Подробные решения на восьми языках программирования

Включает учебное руководство

Регулярные выражения

Сборник рецептов

СИМВОЛ®

O'REILLY®

Ян Гойвертс
Стивен Левитан

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-181-3, название «Регулярные выражения. Сборник рецептов» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Regular Expressions Cookbook

Jan Goyvaerts, Steven Levithan

O'REILLY®

Регулярные выражения
Сборник рецептов

Ян Гойвертс, Стивен Левитан

Санкт-Петербург — Москва
2010

Ян Гойвертс, Стивен Левитан

Регулярные выражения Сборник рецептов

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Гойвертс Я., Левитан С.

Регулярные выражения. Сборник рецептов. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 608 с., ил.

ISBN 978-5-93286-181-3

Сборник содержит более 100 рецептов, которые помогут научиться эффективно оперировать данными и текстом с применением регулярных выражений. Книга знакомит читателя с функциями, синтаксисом и особенностями этого важного инструмента в различных языках программирования: C#, Java, JavaScript, Perl, PHP, Python, Ruby и VB.NET. Предлагаются пошаговые решения наиболее часто встречающихся задач: работа с адресами URL и путями в файловой системе, проверка и форматирование ввода пользователя, обработка текста, а также обмен данными и работа с текстами в форматах HTML, XML, CSV и др.

Данное руководство поможет как начинающему, так и уже опытному специалисту расширить свои знания о регулярных выражениях, познакомиться с новыми приемами, узнать все тонкости работы с ними, научиться избегать ловушек и ложных совпадений. Освоив материал этой книги, вы сможете полнее использовать все те возможности, которые предоставляет умелое применение регулярных выражений, и тем самым сэкономить свое время.

ISBN 978-5-93286-181-3
ISBN 978-0-596-52068-7 (англ)

© Издательство Символ-Плюс, 2010
Authorized translation of the English edition © 2009 O'Reilly. This translation is published and sold by permission of O'Reilly, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 30.10.2009. Формат 70×100 1/16. Печать офсетная.
Объем 38 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 11

1. Введение в регулярные выражения 19

Определение регулярных выражений 19
Поиск с заменой с помощью регулярных выражений 25
Инструменты для работы с регулярными выражениями 27

2. Основные навыки владения регулярными выражениями 48

2.1. Соответствие литеральному тексту 49
2.2. Соответствие непечатным символам 52
2.3. Сопоставление с одним символом из нескольких 54
2.4. Сопоставление с любым символом 59
2.5. Сопоставление в начале и/или в конце строки 62
2.6. Сопоставление с целыми словами 68
2.7. Кодовые пункты Юникода, свойства, блоки и алфавиты 71
2.8. Сопоставление с одной из нескольких альтернатив 85
2.9. Группы и сохранение части совпадения 87
2.10. Повторный поиск соответствия с ранее совпавшим текстом 91
2.11. Сохранение и именованные части совпадения 93
2.12. Повторение части регулярного выражения определенное число раз 97
2.13. Выбор минимального или максимального числа повторений 100
2.14. Устранение бесполезных возвратов 104
2.15. Предотвращение бесконтрольных повторений 108
2.16. Проверка соответствия без включения его в общее соответствие 111
2.17. Совпадение с одной из двух альтернатив по условию 119
2.18. Добавление комментариев в регулярные выражения 122
2.19. Вставка текстового литерала в замещающий текст 124
2.20. Вставка совпадения с регулярным выражением в замещающий текст 128
2.21. Вставка части совпадения с регулярным выражением в замещающий текст 129
2.22. Вставка контекста совпадения в замещающий текст 133

3. Программирование с применением регулярных выражений 135

Языки программирования и диалекты регулярных выражений 135
3.1. Литералы регулярных выражений в исходных текстах 142
3.2. Импортирование библиотеки регулярных выражений 149
3.3. Создание объектов регулярных выражений 151
3.4. Установка параметров регулярных выражений 159
3.5. Проверка возможности совпадения в пределах испытуемой строки 168
3.6. Проверка совпадения со всей испытуемой строкой 175
3.7. Извлечение текста совпадения 181
3.8. Определение позиции и длины совпадения 188
3.9. Извлечение части совпавшего текста 194
3.10. Извлечение списка всех совпадений 202
3.11. Обход всех совпадений в цикле 208
3.12. Проверка полученных совпадений в программном коде 215
3.13. Поиск совпадения внутри другого совпадения 219
3.14. Замена всех совпадений 224
3.15. Замена совпадений с повторным использованием частей совпадений 232
3.16. Замена совпадений фрагментами, сгенерированными в программном коде 238
3.17. Замещение всех совпадений внутри совпадений с другим регулярным выражением 245
3.18. Замещение всех совпадений между совпадениями с другим регулярным выражением 247
3.19. Разбиение строки 253
3.20. Разбиение строки, сохранение совпадений с регулярным выражением 264
3.21. Построчный поиск 269

4. Проверка и форматирование 274

4.1. Проверка адресов электронной почты 274
4.2. Проверка и форматирование телефонных номеров 282
4.3. Проверка международных телефонных номеров 288
4.4. Проверка дат в традиционных форматах 290
4.5. Точная проверка дат в традиционных форматах 295
4.6. Проверка времени в традиционных форматах 300
4.7. Проверка даты и времени в формате ISO 8601 303
4.8. Ограничение возможности ввода алфавитно-цифровыми символами 308
4.9. Ограничение длины текста 312
4.10. Ограничение числа строк в тексте 317

4.11. Проверка утвердительных ответов ... 323
4.12. Проверка номеров социального страхования ... 325
4.13. Проверка номеров ISBN ... 328
4.14. Проверка почтовых индексов ... 337
4.15. Проверка почтовых индексов, используемых в Канаде ... 338
4.16. Проверка почтовых индексов, используемых в Великобритании ... 339
4.17. Поиск адресов, содержащих номер почтового ящика ... 339
4.18. Преобразование имен из формата «имя фамилия» в формат «фамилия, имя» ... 341
4.19. Проверка номеров кредитных карт ... 346
4.20. Европейские регистрационные номера плательщиков НДС ... 353

5. Слова, строки и специальные символы ... 361

5.1. Поиск определенного слова ... 361
5.2. Поиск любого слова из множества ... 364
5.3. Поиск похожих слов ... 367
5.4. Поиск любых слов, за исключением некоторых ... 371
5.5. Поиск любого слова, за которым не следует указанное слово ... 373
5.6. Поиск любого слова, которому не предшествует определенное слово ... 375
5.7. Поиск близко расположенных слов ... 379
5.8. Поиск повторяющихся слов ... 387
5.9. Удаление повторяющихся строк ... 389
5.10. Совпадение с полными строками, содержащими определенное слово ... 395
5.11. Совпадение с полными строками, не содержащими определенное слово ... 397
5.12. Удаление ведущих и завершающих пробельных символов ... 398
5.13. Замена повторяющихся пробельных символов единственным пробелом ... 402
5.14. Экранирование метасимволов регулярных выражений ... 403

6. Числа ... 409

6.1. Целые числа ... 409
6.2. Шестнадцатеричные числа ... 413
6.3. Двоичные числа ... 416
6.4. Удаление ведущих нулей ... 418
6.5. Числа в определенном диапазоне ... 419
6.6. Шестнадцатеричные числа в определенном диапазоне ... 427
6.7. Вещественные числа ... 429

6.8. Числа с разделителями групп разрядов 433
6.9. Римские числа 434

7. URL, пути и адреса в Интернете 438

7.1. Проверка адресов URL 438
7.2. Поиск адресов URL в тексте 442
7.3. Поиск в тексте адресов URL, заключенных в кавычки 444
7.4. Поиск в тексте адресов URL, заключенных в скобки 446
7.5. Преобразование адресов URL в ссылки 448
7.6. Проверка строк URN 449
7.7. Проверка универсальных адресов URL 452
7.8. Извлечение схемы из адреса URL 458
7.9. Извлечение имени пользователя из URL 460
7.10. Извлечение имени хоста из URL 462
7.11. Извлечение номера порта из URL 465
7.12. Извлечение пути из адреса URL 467
7.13. Извлечение строки запроса из URL 471
7.14 Извлечение фрагмента из URL 472
7.15. Проверка доменных имен 473
7.16. Сопоставление с адресами IPv4 476
7.17. Сопоставление с адресами IPv6 479
7.18. Проверка путей в Windows 495
7.19. Выделение элементов путей в Windows 498
7.20. Извлечение буквы устройства из путей в Windows 503
7.21. Извлечение имени сервера и разделяемого ресурса из пути в формате UNC 504
7.22. Извлечение имен папок из путей в Windows 506
7.23. Извлечение имени файла из пути Windows 508
7.24. Извлечение расширения имени файла из пути Windows 510
7.25. Удаление недопустимых символов из имен файлов 511

8. Разметка и обмен данными 513

8.1. Поиск тегов XML 521
8.2. Заменить тег <b> тегом 541
8.3. Удаление всех XML-подобных тегов, за исключением и 545
8.4. Сопоставление с именами XML 549
8.5. Преобразование простого текста в HTML добавлением тегов <p> и
 557
8.6. Поиск определенных атрибутов в XML-подобных тегах 560
8.7. Добавление атрибута cellspacing в теги <table>, где этот атрибут отсутствует 566

8.8. Удаление XML-подобных комментариев 569
8.9. Поиск слов в XML-подобных комментариях 574
8.10. Изменение разделителя, используемого в файлах CSV 579
8.11. Извлечение полей CSV из определенного столбца 584
8.12. Сопоставление с заголовком раздела в файле INI 588
8.13. Сопоставление с разделом в файле INI 589
8.14. Сопоставление с парами имя-значение в файле INI 591

Алфавитный указатель 593

Предисловие

В течение последнего десятилетия наблюдался неуклонный рост интереса к регулярным выражениям. На сегодняшний день все популярные языки программирования включают мощные библиотеки поддержки этого инструмента или даже имеют реализацию этой поддержки, встроенной непосредственно в сам язык. Многие разработчики с успехом используют поддержку регулярных выражений, предоставляя пользователям своих приложений возможности поиска или фильтрации данных с их помощью. Регулярные выражения присутствуют практически везде.

Было издано множество книг, помогающих освоить регулярные выражения. Большинство из них прекрасно справляются с задачей описания их синтаксиса, дополнительно предоставляя некоторые примеры и справочную информацию. Но пока еще не существует книги, которая демонстрировала бы решения, основанные на использовании регулярных выражений, примененные к широкому кругу практических задач, связанных с обработкой текста, возникающих в разнообразных интернет-приложениях. Мы, Стив и Ян, решили данной книгой восполнить этот пробел.

Мы стремились показать, как можно применять регулярные выражения в ситуациях, в которых тем, кто имеет недостаточный опыт работы с этим инструментом, может показаться, что их применить невозможно, а искушенные программисты могли бы посчитать, что для решения подобных задач их не стоило использовать.

Так как в настоящее время регулярные выражения получили повсеместную поддержку, они часто оказываются удобным инструментом, который может использоваться конечными пользователями даже без посредничества программистов. Да и сами программисты могут сэкономить время, используя регулярные выражения для решения задач извлечения и изменения информации, не тратя сил и времени на создание соответствующего процедурного программного кода или изучение описаний возможностей различных вспомогательных библиотек.

Проблемы, связанные с многообразием версий

Как и для любых других инструментов, популярных в мире информационных технологий, существует множество различных реализаций поддержки регулярных выражений, обладающих различной степенью совместимости. В результате это привело к появлению множества различных *диалектов* регулярных выражений, которые не всегда одинаково интерпретируют одно и то же регулярное выражение, если вообще способны интерпретировать его.

Во многих книгах отмечается наличие диалектов и указываются некоторые из их отличий. Но нередко какие-то диалекты в определенных ситуациях вообще не упоминаются – особенно если в этих диалектах отсутствуют какие-то возможности – тогда как можно было бы показать альтернативное решение или обходной путь. Это становится препятствием, когда приходится работать с различными диалектами регулярных выражений в различных приложениях или языках программирования.

Отдельные заявления, встречающиеся в литературе, такие как «в настоящее время все используют регулярные выражения в стиле языка Perl», к сожалению, выводят из рассмотрения широкий круг несовместимостей. Даже между пакетами реализации регулярных выражений «в стиле языка Perl» имеются существенные различия, к тому же сам язык Perl непрерывно развивается дальше. Чрезмерно упрощенное освещение проблемы может приводить к тому, что программист будет проводить часы бесплодных поисков в отладчике, вместо того чтобы исследовать особенности используемой реализации регулярных выражений. И даже когда он обнаружит, что та или иная особенность отсутствует в реализации, он не будет знать, как обойти этот недостаток.

Данная книга является первой, где на всем протяжении параллельно рассматриваются наиболее распространенные и обладающие богатыми возможностями диалекты регулярных выражений,.

Для кого написана эта книга

Издание предназначено для тех, кто постоянно работает с текстовой информацией на компьютере, то есть обрабатывает огромные массивы документов, работает с текстом в текстовых редакторах или разрабатывает программное обеспечение, реализующее анализ и управление текстовой информацией. Регулярные выражения являются превосходным инструментом для решения таких задач. Эта книга содержит все, что необходимо знать об этом инструменте. От читателя не требуется иметь какой-либо опыт работы с регулярными выражениями, потому что в книге объясняются даже самые основные их положения.

Если вы уже знакомы с регулярными выражениями, то найдете здесь множество подробностей, о которых часто умалчивают другие книги или электронные издания. Если вам уже приходилось сталкиваться с ситуацией, когда одно и то же регулярное выражение работает в одном приложении, но не работает в другом, вы найдете весьма ценным подробное описание семи диалектов наиболее распространенных регулярных выражений, представленное в этой книге. Мы организовали книгу как сборник рецептов, поэтому вы можете начинать чтение непосредственно с интересующей вас темы. А если вы прочтете это издание от корки до корки, вы станете мастером регулярных выражений мирового уровня.

Эта книга даст вам все необходимые сведения о регулярных выражениях, даже если вы не занимаетесь программированием. Если вам требуется использовать регулярные выражения в текстовом редакторе, в инструменте поиска или в каком-либо другом приложении, где имеется поле ввода с надписью «regex» (регулярное выражение), вы можете приступать к чтению этой книги, даже не обладая опытом программирования. В большинстве рецептов, что приводятся в книге, даны решения, основанные исключительно на применении одного или нескольких регулярных выражений.

Для программистов в главе 3 приводятся полные сведения, необходимые для реализации регулярных выражений в исходном программном коде. Данная глава предполагает, что читатель обладает знаниями об основных особенностях выбранного им языка программирования, но не требует наличия опыта использования регулярных выражений в исходном программном коде.

Охватываемые технологии

.NET, Java, JavaScript, PCRE, Perl, Python и Ruby – это не просто броские названия. Они представляют семь диалектов регулярных выражений, рассматриваемых в данной книге. Мы в равной степени уделим внимание всем семи диалектам. В особенности мы позаботимся о том, чтобы указать все несовместимости, существующие между ними, которые нам удалось обнаружить.

Глава, имеющая отношение к программированию (глава 3) , содержит листинги исходных текстов на языках C#, Java, JavaScript, PHP, Perl, Python, Ruby и VB.NET. Здесь также для каждого рецепта приводятся решения и их описания для всех восьми языков. Так как это приводит к тому, что в главе многое повторяется не один раз, вы легко можете пропускать обсуждения, связанные с языками программирования, которые вас не интересуют, – достаточно следить за тем, что относится к выбранному вами языку.

Организация книги

В первых трех главах описываются наиболее ценные инструменты и даются основные сведения, позволяющие начать использовать регулярные выражения. Каждая последующая глава представляет различные регулярные выражения, подробно исследуя какой-то один круг задач обработки текста.

Глава 1 «Введение в регулярные выражения» объясняет назначение регулярных выражений и знакомит с набором инструментов, способных облегчить их изучение, создание и отладку.

Глава 2 «Основные навыки владения регулярными выражениями» описывает каждый элемент и каждую особенность регулярных выражений, а также содержит рекомендации по их эффективному использованию.

Глава 3 «Программирование с применением регулярных выражений» демонстрирует приемы программирования и содержит листинги с исходными текстами на всех языках программирования, описываемых в этой книге, где используются регулярные выражения.

Глава 4 «Проверка и форматирование» содержит рецепты обработки информации, вводимой пользователем, такой как даты, номера телефонов и почтовые индексы, используемые в разных странах.

Глава 5 «Слова, строки и специальные символы» рассматривает наиболее типичные задачи обработки текста, такие как проверка наличия или отсутствия в строках определенных слов.

Глава 6 «Числа» показывает, как определять целые и вещественные числа при вводе в различных форматах.

Глава 7 «URL, пути и адреса в Интернете» демонстрирует, как анализировать и манипулировать строками, часто используемыми в Интернете и в системе Windows в процессе поиска.

Глава 8 «Разметка и обмен данными» охватывает приемы манипулирования текстом в форматах HTML, XML, CSV (comma-separated values – данные, разделенные запятыми) и конфигурационными файлами в формате INI.

Типографские соглашения

В этой книге используются следующие типографские соглашения:

Курсив

Курсивом выделяются новые термины, адреса URL, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`

Используется для представления листингов программ; элементов программного кода, таких как имена переменных или функций;

значений, возвращаемых в результате применения регулярных выражений, а также обрабатываемых элементов или текста, к которым применяется регулярное выражение. Это может быть содержимое поля ввода в приложении, файла на диске или содержимое строковой переменной.

Моноширинный курсивный шрифт

Используется для представления текста, который должен замещаться значениями, вводимыми пользователем, или для представления значений, определяемых контекстом.

‹Регулярное•выражение›

Так представляются регулярные выражения сами по себе или в том виде, в каком они вводятся в поле ввода в приложениях. Пробелы внутри регулярных выражений обозначаются кружочком, за исключением тех случаев, когда количество пробелов не имеет большого значения.

«Замещаемый•текст»

Так будет представлен текст, соответствующий регулярному выражению, который будет замещен при выполнении операции поиска с заменой. Пробелы в замещаемом тексте обозначаются кружочком.

Совпавший текст

Так будет представлен фрагмент текста, соответствующий регулярному выражению.

...

Многоточие внутри регулярных выражений указывает место, которое должно быть заполнено вами, прежде чем вы сможете воспользоваться ими. Сопутствующий текст поясняет, что должно быть подставлено вместо многоточия.

CR, LF и CRLF

Обозначения CR, LF и CRLF в рамках представляют фактические разрывы текстовых строк, а не специальные экранированные последовательности, такие как \r, \n и \r\n. Такие строки могут создаваться в многострочных полях ввода нажатием клавиши Enter или при использовании многострочных строковых констант в исходных текстах программ, таких как строковые литералы в языке C#, или в строках, окруженных тройными кавычками, в языке Python.

↵

Эту стрелку вы можете увидеть на клавише Return или Enter, имеющейся на клавиатуре. Она говорит о том, что текст был перенесен исключительно для того, чтобы уместить его по ширине книжной страницы. При вводе такого текста не нужно нажимать клавишу Enter — просто продолжайте ввод в той же самой строке.

Этот значок отмечает советы, предложения и примечания общего характера.

Этот значок отмечает предупреждения и предостережения.

Использование программного кода примеров

Данная книга призвана оказать помощь в решении ваших задач. Вообще вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить значительные отрывки программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Для цитирования данной книги или примеров из нее, при ответах на вопросы не требуется получение разрешения. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Regular Expressions Cookbook by Jan Goyvaerts and Steven Levithan. Copyright 2009 Jan Goyvaerts and Steven Levithan, 978-0-596-2068-7».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Safari® Books Online

Если на обложке технической книги есть пиктограмма «Safari® Books Online», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://www.regexcookbook.com

или

http://oreilly.com/catalog/9780596520687

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Благодарности

Мы выражаем свою признательность Энди Ораму (Andy Oram), нашему редактору из издательства O'Reilly Media, Inc., за то, что он сопровождал этот проект от начала и до конца. Мы также благодарим Джеффри Фридла (Jeffrey Friedl), Зака Грента (Zak Greant), Николая Линдберга (Nikolaj Lindberg) и Яна Морса (Ian Morse), за техническое рецензирование, что позволило сделать эту книгу более последовательной и точной.

1

Введение в регулярные выражения

Раз уж вы открыли эту книгу, то наверняка захотите ввести в свой программный код какие-то из несуразных строк, состоящих из круглых скобок и вопросительных знаков, которые вам встретятся в этих главах. Если вы готовы включиться в игру, будьте нашим гостем: практические регулярные выражения и их описания вы найдете в главах с 4 по 8.

Однако и начальные главы помогут вам сэкономить немало времени в будущем. Например, в этой главе будут представлены некоторые утилиты (часть из них создана Яном, одним из авторов книги), которые позволят вам проверять и отлаживать регулярные выражения перед включением их в программный код, где отыскать возможные ошибки будет намного труднее. Кроме того, начальные главы покажут вам, как использовать различные особенности и возможности регулярных выражений, способные облегчить вашу жизнь, дадут понимание регулярных выражений с точки зрения их оптимизации, покажут тонкие отличия в том, как обрабатываются регулярные выражения в разных языках программирования и даже в различных версиях одного и того же языка.

Мы приложили немалые усилия при подготовке этих базовых сведений в надежде на то, что вы начнете изучение с них или, если вы уже столкнулись с проблемами, связанными с использованием регулярных выражений, то приняли решение со всем этим разобраться.

Определение регулярных выражений

В контексте этой книги под *регулярными выражениями* понимаются определенные виды текстовых шаблонов, которые могут использоваться во многих современных приложениях и языках программирования. Вы можете использовать их, чтобы убедиться, что введенный текст вписывается в заданный шаблон; отыскивать внутри больших объемов текста фрагменты, соответствующие шаблону; замещать текст, соответствующий шаблону, другим текстом или переупорядочивать фраг-

менты совпавшего текста; разбивать блоки текста на списки более мелких фрагментов или даже делать не столь разумные вещи. Эта книга поможет вам точно понимать, что вы делаете, и избегать ошибок.

История появления термина «регулярное выражение»

Термин «регулярное выражение» пришел из математики и теоретической информатики, где он отражает свойство математических выражений, называемое *регулярностью*. Такие выражения могут быть реализованы программно с использованием детерминированных конечных автоматов (ДКА). ДКА – это механизм с конечным числом состояний, не поддерживающий возможность возврата (backtracking).

Текстовые шаблоны, использовавшиеся в самых ранних версиях инструментов *grep*, были регулярными выражениями в математическом смысле. Несмотря на то, что имя прижилось, современные регулярные выражения в стиле языка Perl фактически не являются регулярными выражениями в математическом смысле. Они реализованы с применением механизма недетерминированных конечных автоматов (НКА). Вы вскоре поближе познакомитесь с понятием возврата. Программистам-практикам достаточно усвоить из этого примечания следующее: высоколобым компьютерным теоретикам приходится мириться с тем, что чистые теоретические модели заменяются технологиями, более полезными в реальном мире.

При грамотном использовании регулярные выражения способны упростить задачи программирования и обработки текстов, а также обеспечить решение многих задач, которые вообще не могли бы быть решены без использования регулярных выражений. Вам могут потребоваться десятки, если не сотни, строк процедурного программного кода, чтобы извлечь все адреса электронной почты из документа, и такой код будет очень сложно написать и сопровождать. Однако с применением регулярных выражений, как показано в рецепте 4.1, решение этой задачи может быть реализовано всего лишь парой строк кода, а возможно, даже и одной строкой.

Но если попытаться взвалить слишком много на единственное регулярное выражение или задействовать регулярные выражения там, где они малопригодны, вы поймете, почему некоторые говорят[1]:

[1] Историю этой цитаты Джеффри Фридл прослеживает в своем блоге по адресу: *http://regex.info/blog/2006-09-15/247*.

> Когда кто-то, сталкиваясь с проблемой, думает: «А-а, понимаю, здесь поможет регулярное выражение», то в результате он получает две проблемы.

Вторая проблема, которую получают те, кто так думает, состоит в том, что они не читали справочное руководство, которое вы держите в руках. Продолжайте читать его. Регулярные выражения – это очень мощный инструмент. Если решаемая задача связана с обработкой или извлечением текста на компьютере, то владение регулярными выражениями поможет вам сэкономить массу времени.

Множество диалектов регулярных выражений

Итак, заголовком предыдущего раздела мы ввели вас в заблуждение. Мы не дали определение регулярных выражений. И мы не можем сделать этого. В мире не существует официального стандарта, который точно определял бы, какой текстовый шаблон является регулярным выражением, а какой – нет. Как вы можете себе представить, каждый, кто проектирует свой язык программирования, и каждый разработчик приложений, обрабатывающих тексты, имеет свое представление о том, какими должны быть регулярные выражения. Поэтому в результате мы имеем массу *диалектов* регулярных выражений.

К счастью, большинство проектировщиков и разработчиков оказываются достаточно ленивы. Зачем создавать нечто совершенно новое, когда можно скопировать то, что уже существует? В результате почти все современные диалекты регулярных выражений, включая и те, что рассматриваются в этой книге, ведут свою историю от языка программирования Perl. Мы называем эти диалекты *регулярными выражениями в стиле языка Perl*. Синтаксис этих диалектов прост и главным образом совместим, хотя и не полностью.

Писатели тоже ленивы. Единственное регулярное выражение мы обычно называем *regex* или *regexp*, а термин regexes используем для обозначения множественного числа.

Диалекты регулярных выражений не связаны жестко с какими-то определенными языками программирования. Языки сценариев обычно имеют свои собственные, встроенные диалекты регулярных выражений. Другие языки программирования опираются на использование библиотек, реализующих поддержку регулярных выражений. Некоторые библиотеки доступны для нескольких языков программирования, а некоторые языки программирования предоставляют возможность выбора из нескольких библиотек.

Эта вводная глава рассматривает диалекты регулярных выражений без всякой связи с языками программирования. Листинги с исходными текстами начнут появляться в главе 3; в разделе «Языки программирования и диалекты регулярных выражений» вы узнаете, с какими диалектами вам предстоит работать. А пока оставим в стороне все вопросы,

связанные с программированием. Инструменты, перечисленные в следующем разделе, обеспечивают простоту исследования синтаксиса регулярных выражений, «обучая в процессе работы».

Диалекты регулярных выражений, рассматриваемые в этой книге

Для этой книги мы отобрали наиболее распространенные диалекты регулярных выражений, используемых в наши дни. Все они являются диалектами *регулярных выражений языка Perl*. Некоторые диалекты имеют более широкие возможности по сравнению с другими. Но, как правило, когда два диалекта обладают одинаковым набором возможностей, они используют один и тот же синтаксис. Мы укажем на некоторые из досадных несовместимостей, с которыми нам пришлось столкнуться.

Все рассматриваемые здесь диалекты являются частью какого-либо языка программирования или библиотеки, которые продолжают активно разрабатываться. Из списка диалектов, что приводится ниже, вы узнаете, какие версии рассматриваются в этой книге. На протяжении всей книги мы будем упоминать диалекты без их версий, если представленные регулярные выражения одинаково работают во всех диалектах, что случается практически всегда. Помимо исправлений ошибок, проявляющихся в крайних случаях, диалекты обычно не изменяются, за исключением добавления новых особенностей и введения новых синтаксических конструкций, которые в предыдущих версиях трактовались как ошибка:

Perl

То, что язык Perl имеет встроенную поддержку регулярных выражений, является главной причиной их популярности в наши дни. В этой книге рассматриваются версии Perl 5.6, 5.8 и 5.10.

Многие приложения и библиотеки реализации регулярных выражений, которые, как утверждается, поддерживают регулярные выражения языка Perl или совместимы с ними, на самом деле просто используют регулярные выражения в стиле языка Perl. Они поддерживают синтаксис, похожий на синтаксис Perl, но не поддерживают тот же набор особенностей регулярных выражений. Наиболее вероятно, что они используют один из диалектов, что приводятся в списке ниже. Все эти диалекты являются диалектами в стиле языка Perl.

PCRE

PCRE – это библиотека реализации «Perl-Compatible Regular Expressions» (Perl-совместимых регулярных выражений), написанная на языке C Филиппом Хейзелем (Philip Hazel). Вы можете получить исходные тексты библиотеки по адресу: *http://www.pcre.org*. В этой книге рассматриваются версии PCRE с 4 по 7.

Несмотря на утверждение, что библиотека PCRE совместима с диалектом регулярных выражений языка Perl, что в отношении ее даже более справедливо, чем для других диалектов, тем не менее, она всего лишь поддерживает стиль языка Perl. Некоторые особенности, такие как поддержка Юникода, в ней отличаются, и она не допускает внедрение программного кода на языке Perl в регулярные выражения, как это допускается в самом языке Perl.

Благодаря свободной лицензии и надежности библиотека PCRE нашла применение во многих языках программирования и приложениях. Она встроена в язык PHP и включена в состав многих компонентов Delphi. Если заявлено, что приложение поддерживает «Perl-совместимые» регулярные выражения без указания конкретного используемого диалекта, наиболее вероятно, что используется библиотека PCRE.

.NET

Платформа Microsoft .NET Framework обеспечивает поддержку полноценного диалекта регулярных выражений в стиле языка Perl в виде пакета *System.Text.RegularExpressions*. В этом издании описываются версии .NET начиная с 1.0 по 3.5. Строго говоря, существует всего две версии пакета *System.Text.RegularExpressions*: 1.0 и 2.0. В версиях .NET 1.1, 3.0 и 3.5 классы Regex не претерпели никаких изменений.

Любые языки программирования, поддерживаемые платформой .NET, включая C#, VB.NET, Delphi for .NET и даже COBOL.NET, обладают полным доступом ко всем особенностям диалекта регулярных выражений .NET. Если приложение, разработанное на основе платформы .NET, обеспечивает поддержку регулярных выражений, можно быть совершенно уверенным, что оно использует диалект .NET, даже если заявлено, что используются «регулярные выражения Perl». Исключение составляет сама среда разработки Visual Studio (VS). Сама среда разработки VS по-прежнему использует устаревший диалект регулярных выражений, использовавшийся с самого начала, который вообще не является диалектом в стиле языка Perl.

Java

Java 4 – это первая версия, в которой присутствует встроенная поддержка регулярных выражений, реализованная в виде пакета java.util.regex. Эта реализация быстро затмила различные сторонние библиотеки для Java. Кроме того, что этот пакет является стандартным и встроенным, он также предлагает полноценную реализацию Perl-совместимого диалекта регулярных выражений и имеет превосходную производительность, сопоставимую даже с приложениями, написанными на языке C. В этой книге рассматривается пакет java.util.regex, входящий в состав Java 4, 5 и 6.

Если вы пользуетесь программным обеспечением, написанным на языке Java не более чем пару лет тому назад, любые регулярные выражения, поддерживаемые им, весьма вероятно используют диалект Java.

JavaScript

Название *JavaScript* используется в этой книге для обозначения диалекта регулярных выражений, определяемого стандартом ECMA-262 версии 3. Этот стандарт определяет язык программирования ECMAScript, который в различных веб-броузерах более широко известен под названиями JavaScript и JScript. Все броузеры: Internet Explorer с версии 5.5 по 8.0, Firefox, Opera и Safari – реализуют третью версию стандарта ECMA-262. Однако у каждого из броузеров имеются свои отклонения от стандарта. Мы будем указывать на эти отклонения в ситуациях, когда это будет иметь значение.

Если веб-сайт позволяет выполнять поиск или фильтрацию содержимого с помощью регулярных выражений, не вынуждая ожидать ответа веб-сервера, следовательно он использует диалект JavaScript регулярных выражений, который поддерживается броузерами на стороне клиента. Этот диалект используется даже в языках программирования Microsoft VBScript и Adobe ActionScript.

Python

Язык программирования Python обеспечивает поддержку регулярных выражений посредством модуля *re*. В этой книге рассматриваются версии Python 2.4 и 2.5. Поддержка регулярных выражений в языке Python остается неизменной уже в течение нескольких лет.

Ruby

Поддержка регулярных выражений в языке Ruby реализована как часть самого языка, так же как и в Perl. В этой книге рассматриваются версии Ruby 1.8 и 1.9. Версия Ruby 1.8 по умолчанию использует диалект регулярных выражений, реализация которого включена непосредственно в исходные тексты Ruby. Версия Ruby 1.9 по умолчанию использует библиотеку регулярных выражений Onigurama. Имеется возможность скомпилировать Ruby 1.8 с поддержкой библиотеки Onigurama, а Ruby 1.9 – с поддержкой старого диалекта регулярных выражений Ruby. В этой книге мы будем обозначать встроенный диалект Ruby как Ruby 1.8, а диалект Onigurama как Ruby 1.9.

Чтобы определить, какой диалект регулярных выражений Ruby используется у вас, попробуйте выполнить регулярное выражение `‹a++›`. Если поддерживается диалект Ruby 1.8, будет выдано сообщение об ошибке, потому что он не поддерживает захватывающие квантификаторы, тогда как в диалекте Ruby 1.9 этому выражению соответствует строка, состоящая из одного или более символов.

Библиотека Onigurama предусматривает обратную совместимость с диалектом Ruby 1.8, просто добавляя новые возможности, которые не влияют на работу существующих регулярных выражений. Разработчики даже оставили некоторые особенности, которые следовало бы изменить, например выражение `(?m)` означает «точка соответствует концу строки», тогда как в других диалектах для этой же цели используется выражение `(?s)`.

Поиск с заменой с помощью регулярных выражений

Поиск с заменой – это типичная задача, которая решается с применением регулярных выражений. Функция, реализующая поиск с заменой, принимает строку, в которой производится поиск, регулярное выражение и строку замены, которая будет замещать найденные фрагменты. Результатом функции является строка, где все участки, соответствующие регулярному выражению, заменяются замещающим текстом.

Даже при том, что замещающий текст не является регулярным выражением, тем не менее, допускается использовать специальный синтаксис с целью динамического создания замещающего текста. Как это делается, описывается в рецептах 2.20 и 2.21. Некоторые диалекты также поддерживают вставку совпавшего содержимого в замещающий текст, как показано в рецепте 2.22. В главе 3 в рецепте 3.16 показано, как с помощью программного кода генерировать различные замещающие тексты для каждого совпадения.

Множество диалектов определения замещающего текста

Многообразие идей, выдвигаемых различными разработчиками реализаций регулярных выражений, привело к появлению широкого диапазона диалектов регулярных выражений, каждый из которых обладает своими особенностями. История выработки синтаксиса определения замещающего текста также не является исключением. В действительности диалектов определения замещающего текста оказалось даже больше, чем диалектов самих регулярных выражений. Создание механизма регулярных выражений – задача не из легких. Большинство программистов предпочитают использовать один из существующих, а прикрутить функцию поиска с заменой к существующему механизму регулярных выражений значительно проще. В результате появилось множество диалектов определения замещающего текста для библиотек поддержки регулярных выражений, в которых отсутствуют собственные реализации функции поиска с заменой.

К счастью, все диалекты регулярных выражений, представленные в этой книге, обладают собственными возможностями выполнения по-

иска с заменой, за исключением PCRE. Этот недостаток PCRE усложняет жизнь тем программистам, которые используют диалекты, созданные на его основе. Свободно распространяемая библиотека PCRE не содержит встроенных функций, реализующих возможность замены. Поэтому все приложения и языки программирования, опирающиеся на использование PCRE, должны предоставлять собственные реализации функции поиска с заменой. Многие программисты пытаются копировать существующие реализации, но практически всегда делают это немного по-разному.

Ниже перечислены диалекты определения замещающего текста, рассматриваемые в этой книге. За дополнительной информацией о соответствующих им диалектах регулярных выражений обращайтесь к разделу «Множество диалектов регулярных выражений»:

Perl

Язык Perl обладает встроенной поддержкой замещения текста с помощью регулярных выражений, реализованной в виде инструкции `s/regex/replace/`. Диалект определения замещающего текста в языке Perl соответствует его диалекту регулярных выражений. В этой книге рассматриваются версии от Perl 5.6 по Perl 5.10. В последней версии была добавлена поддержка именованных обратных ссылок в замещающем тексте как дополнение к добавленной в регулярные выражения поддержке именованных сохранений.

PHP

В этой книге рассматривается реализация поиска с заменой на языке PHP в виде функции `preg_replace`. Данная функция использует диалект регулярных выражений PCRE и диалект PHP определения замещающего текста.

В других языках программирования, опирающихся на библиотеку PCRE, используются отличные от PHP диалекты определения замещающего текста. В зависимости от того, где черпали вдохновение разработчики того или иного языка программирования, синтаксис определения замещающего текста может быть похожим на диалект PHP или на любой другой диалект определения замещающего текста из представленных в этой книге.

В языке PHP имеется также функция `ereg_replace`. Эта функция использует другой диалект регулярных выражений (POSIX ERE) и иной диалект определения замещающего текста. Семейство функций `ereg` из языка PHP не будет рассматриваться в этой книге.

.NET

Пакет *System.Text.RegularExpressions* содержит несколько функций, реализующих поиск с заменой. Диалект определения замещающего текста в .NET соответствует диалекту регулярных выражений .NET. Все версии .NET используют один и тот же диалект определения замещающего текста. Новые особенности регулярных вы-

ражений, появившиеся в .NET 2.0, никак не повлияли на синтаксис определения замещающего текста.

Java

Пакет *java.util.regex* содержит собственные встроенные функции поиска с заменой. В этой книге рассматриваются версии Java 4, 5 и 6. Во всех этих версиях используется один и тот же синтаксис определения замещающего текста.

JavaScript

Под названием *JavaScript* в этой книге мы будем подразумевать как диалект определения замещающего текста, так и диалект регулярных выражений, определяемый стандартом ECMA-262 версии 3.

Python

Модуль *re* в языке Python реализует поиск с заменой в виде функции `sub()`. Диалект определения замещающего текста в языке Python соответствует диалекту регулярных выражений Python. В этой книге рассматриваются версии Python 2.4 и 2.5. Реализация поддержки регулярных выражений в языке Python остается неизменной на протяжении уже многих лет.

Ruby

Поддержка регулярных выражений в языке Ruby встроена непосредственно в сам язык, включая и функцию поиска с заменой. В этой книге рассматриваются версии Ruby 1.8 и 1.9. Версия Ruby 1.8 по умолчанию использует диалект регулярных выражений, реализация которого включена непосредственно в исходные тексты Ruby, тогда как версия Ruby 1.9 по умолчанию использует библиотеку регулярных выражений Onigurama. Имеется возможность скомпилировать Ruby 1.8 с поддержкой библиотеки Onigurama, а Ruby 1.9 – с поддержкой старого диалекта регулярных выражений Ruby. В этой книге мы будем обозначать встроенный диалект Ruby как Ruby 1.8, а диалект Onigurama как Ruby 1.9.

Для Ruby 1.8 и 1.9 используется один и тот же синтаксис определения замещающего текста, за исключением того, что в версии Ruby 1.9 добавлена поддержка именованных обратных ссылок в замещающем тексте. Именованные сохранения – это новая особенность, появившаяся в поддержке регулярных выражений Ruby 1.9.

Инструменты для работы с регулярными выражениями

Если у вас нет опыта программирования регулярных выражений, мы рекомендуем для начала поэкспериментировать с ними с помощью дополнительных инструментов и только потом включать их в программ-

ный код. В этой главе представлены примеры простых регулярных выражений, которые не содержат дополнительных экранированных последовательностей, те что необходимы при использовании в языках программирования (даже в языке командной оболочки UNIX). Эти регулярные выражения можно вводить непосредственно в поле поиска приложения.

В главе 3 описывается, как оформляются регулярные выражения в исходных текстах программ. Оформление регулярных выражений в виде строк еще больше осложняет их восприятие человеком, потому что к правилам экранирования последовательностей внутри регулярных выражений добавляются правила экранирования служебных символов в строках. Мы не будем касаться этой проблемы до рецепта 3.1. Как только вы разберетесь с основами регулярных выражений, вы сможете увидеть лес за частоколом обратных слэшей.

Инструменты, описываемые в этом разделе, кроме всего прочего обеспечивают возможность отладки, проверки синтаксиса и другие удобства, которые недоступны в большинстве сред программирования. По этой причине в процессе разработки регулярных выражений для своих приложений вы можете найти для себя удобным сначала создать сложное регулярное выражение в одном из этих инструментов и потом перенести его в свою программу.

RegexBuddy

Программа RegexBuddy (рис. 1.1) – это один из наиболее полнофункциональных инструментов, доступных на момент написания этих строк, позволяющий создавать, тестировать и реализовывать регулярные выражения. Он обладает уникальной возможностью имитировать все диалекты регулярных выражений, рассматриваемые в этой книге, и даже преобразовывать их из одного диалекта в другой.

Программа RegexBuddy была спроектирована и разработана Яном Гойвертсом (Jan Goyvaerts), одним из авторов этой книги. Работа над программой RegexBuddy сделала Яна экспертом в области регулярных выражений, а использование RegexBuddy помогло довести его соавтора Стивена до этапа передачи этой книги в издательство O'Reilly.

Если интерфейс программы (рис. 1.1) покажется вам перегруженным, то это только потому, что мы распахнули большую часть из имеющихся панелей, чтобы продемонстрировать обширные возможности RegexBuddy. По умолчанию все панели аккуратно свернуты в строку вкладок. Существует также возможность перемещать панели на второй монитор.

Чтобы опробовать какое-либо регулярное выражение из представленных в этой книге, просто введите его в поле редактирования, расположенное в верхней части окна RegexBuddy. Программа автоматически

подсветит синтаксис введенного регулярного выражения, ярко выделив возможные ошибки и непарные скобки.

По мере ввода регулярного выражения в панели Create (Создать) автоматически будет создаваться подробный отчет на английском языке. Двойной щелчок мышью на любом из описаний в дереве будет вызывать переход к редактированию соответствующей части регулярного выражения. Вы можете вставлять новые элементы в регулярное выражение вручную или, щелкнув на кнопке Insert Token (Вставить лексему), выбирать нужный элемент из меню. Например, если вы не помните достаточно сложный синтаксис позитивных опережающих проверок, можно попросить RegexBuddy вставить соответствующие символы.

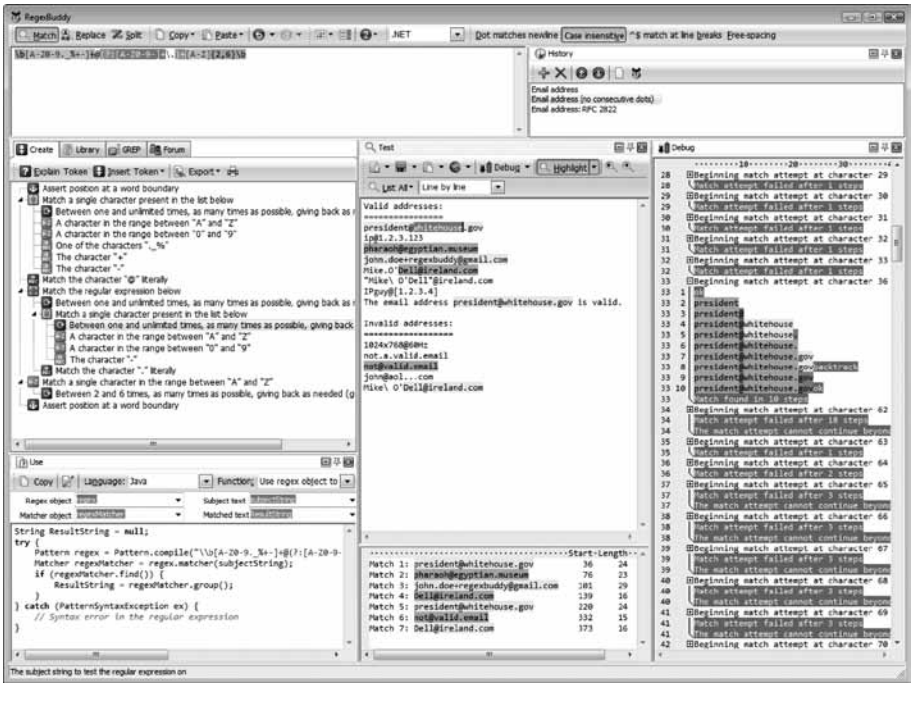


Рис. 1.1. *RegexBuddy*

После ввода или вставки некоторого примерного текста в панель Test (Проверка) при нажатой кнопке Highlight (Выделить) программа RegexBuddy автоматически выделит фрагменты текста, соответствующие регулярному выражению.

В число кнопок, которые вам наверняка придется использовать чаще всего, входят:

List All (Перечислить все)

Отображает список всех совпадений.

Replace (Заменить)

Кнопка Replace (Заменить), расположенная в верхней части окна, дает возможность ввести замещающий текст. Кнопка Replace (Заменить) в панели Test (Проверка) дает возможность увидеть, как будет выглядеть первоначальный текст после выполнения операции замены.

Split (Разделить) (кнопка в панели Test, а не в верхней части окна)

Интерпретирует регулярное выражение как разделитель и разбивает исходный текст на лексемы по совпадениям, найденным с помощью регулярного выражения.

Щелкните на любой из этих кнопок и в меню List All (Перечислить все) выберите пункт Update Automatically (Обновлять автоматически), чтобы обеспечить автоматическую синхронизацию результатов по мере редактирования регулярного выражения или текста примера.

Чтобы в точности увидеть, как работает (или не работает) регулярное выражение, следует щелкнуть в панели Test (Проверка) на выделенном совпадении или в месте, где обнаружено отсутствие совпадения, и затем щелкнуть на кнопке Debug (Отладка). Программа RegexBuddy переключится в панель Debug (Отладка) и по шагам продемонстрирует процесс поиска совпадения. Щелкнув мышью в пределах вывода отладчика, можно определить, какому фрагменту регулярного выражения соответствует текст, на котором был выполнен щелчок. Щелкнув мышью в пределах регулярного выражения, можно выделить фрагмент текста в панели отладчика, соответствующий данной части регулярного выражения.

В панели Use (Использование) можно выбрать предпочтительный язык программирования и функцию и тут же получить исходный программный код реализации регулярного выражения. Исходные тексты шаблонов для RegexBuddy полностью доступны для редактирования во встроенном редакторе шаблонов. Благодаря этому можно добавлять новые функции и даже новые языки программирования или изменять существующие.

Чтобы испытать работу регулярного выражения на большом объеме данных, можно переключиться в панель GREP и произвести поиск (возможно, с заменой) в любом числе файлов и каталогов.

Обнаружив регулярное выражение в исходном тексте сопровождаемой программы, можно скопировать его в буфер обмена, включая ограничивающие его кавычки или слэши, затем в окне программы RegexBuddy щелкнуть на кнопке Paste (Вставить), расположенной в верхней части, и выбрать стиль оформления строк для вашего языка программирования. После этого регулярное выражение появится в окне RegexBuddy как обычное регулярное выражение, без дополнительных кавычек и экранирующих последовательностей, необходимых для оформления

литералов строк. Чтобы создать в буфере обмена строку с соблюдением синтаксических правил, достаточно щелкнуть на кнопке Copy (Копировать), расположенной в верхней части, которую затем можно будет вставить в исходный текст программы.

По мере накопления опыта в панели Library (Библиотека) можно создавать собственную библиотеку регулярных выражений. Не забывайте добавлять подробное описание и тестовые примеры при сохранении регулярного выражения. Регулярные выражения могут выглядеть сложными для понимания даже опытных экспертов.

В случае неудачи при создании собственного регулярного выражения можно щелкнуть на вкладке панели Forum (Форум) и затем на кнопке Login (Зарегистрироваться). В случае если вы приобрели программу RegexBuddy на законных основаниях, на экране появится форма входа, заполнив которую и щелкнув на кнопке OK вы подключитесь к форуму пользователей RegexBuddy. Стивен и Ян часто бывают здесь.

Программа RegexBuddy работает под управлением операционных систем Windows 98, ME, 2000, XP и Vista. Пользователи Linux и Apple могут запускать ее под управлением VMware, Parallels, CrossOver Office и, с некоторыми усилиями, под WINE. Загрузить оценочную версию RegexBuddy можно по адресу *http://www.regexbuddy.com/RegexBuddyCookbook.exe*. Оценочная версия обладает полным набором функциональных возможностей в течение семи дней фактического использования, за исключением доступа к форуму.

RegexPal

RegexPal (рис. 1.2) – это веб-приложение, выполняющее тестирование регулярных выражений. Оно было написано Стивеном Левитаном (Steven Levithan), одним из авторов этой книги. Все, что необходимо для работы с этим приложением, это современный веб-броузер. Приложение RegexPal полностью написано на языке JavaScript. Вследствие этого оно поддерживает только диалект регулярных выражений JavaScript, реализованный в веб-броузере, используемом для доступа к веб-приложению.

Чтобы опробовать какое-либо регулярное выражение из этой книги, откройте страницу *http://www.regexpal.com* и введите регулярное выражение в поле с текстом «Enter regex here». Приложение RegexPal автоматически подсветит синтаксис введенного регулярного выражения, что сразу же позволит определить синтаксические ошибки. Приложение RegexPal учитывает проблемы несовместимости броузеров, которые могут нарушить ваши планы при разработке регулярных выражений для JavaScript. Если какая-то синтаксическая конструкция работает некорректно в некоторых броузерах, RegexPal выделит ее как ошибку.

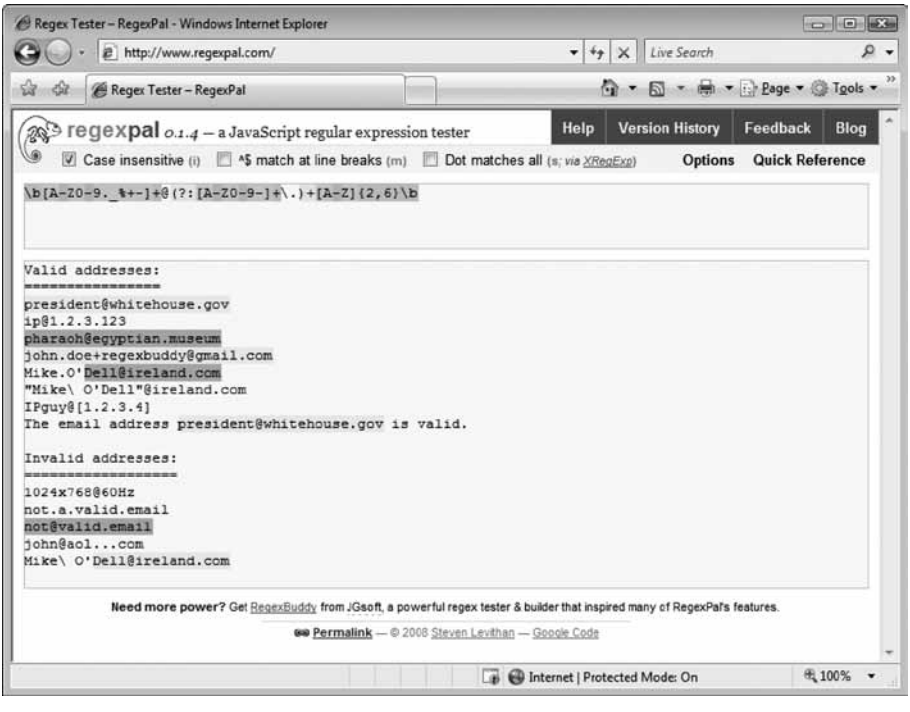


Рис. 1.2. *RegexPal*

Теперь можно ввести текст примера в поле с текстом «Enter test data here», после чего RegexPal автоматически выделит фрагменты текста, соответствующие регулярному выражению.

Здесь нет никаких кнопок, на которых требовалось бы было бы щелкать, что делает RegexPal одним из самых удобных веб-приложений тестирования регулярных выражений.

Другие примеры веб-приложений тестирования регулярных выражений

Создать простое веб-приложение для тестирования регулярных выражений достаточно просто. Если у вас имеются основные навыки веб-разработки, в главе 3 вы найдете все, что вам будет необходимо для создания такого приложения. Сотни людей уже сделали это, причем некоторые из них дополнили свои приложения дополнительными возможностями, которые делают их заслуживающими внимания.

regex.larsolavtorvik.com

Ларс Олав Торвик (Lars Olav Torvik) создал небольшое, но замечательное приложение тестирования регулярных выражений, доступное по адресу *http://regex.larsolavtorvik.com* (рис. 1.3).

Рис. 1.3. *regex.larsolavtorvik.com*

Для начала выберите требуемый диалект регулярных выражений, щелкнув на названии диалекта в верхней части страницы. Ларс предлагает выбор между диалектами PHP PCRE, PHP POSIX и JavaScript. PHP PCRE – это диалект PCRE, рассматриваемый в этой книге и используемый в языке PHP семейством функций `preg`. POSIX – это устаревший диалект с ограниченными возможностями, используемый семейством функций `ereg` в языке PHP, который не рассматривается в данной книге. В случае выбора JavaScript вы будете работать с реализацией JavaScript в вашем броузере.

Введите регулярное выражение в поле с меткой `Pattern` (Шаблон) и текст примера в поле `Subject` (Испытуемый текст). Спустя мгновение в поле `Matches` (Совпадения) появится текст примера с выделенными фрагментами, соответствующими регулярному выражению. В поле `Code` (Программный код) появится строка программного кода, которая применяет регулярное выражение к испытуемому тексту. Операция копирования и вставки этой строки в текстовом редакторе позволит вам

сэкономить время на преобразование вручную регулярного выражения в строковый литерал. Любая строка или массив, возвращаемые программным кодом, отображаются в поле Result (Результат). Так как при создании своего веб-приложения Ларс использовал технологию Ajax, результаты появляются всего через несколько мгновений для любого выбранного диалекта. Для работы с этим инструментом необходимо подключение к сети, так как интерпретатор PHP выполняется на сервере, а не в броузере.

Во второй колонке отображается список команд и параметров регулярных выражений. Перечень доступных команд и параметров зависит от выбранного диалекта регулярных выражений. В число команд обычно входят такие операции, как поиск совпадений, замена и разбиение. В число параметров входят такие, как чувствительность к регистру символов, а также ряд других параметров, характерных для реализации. Эти команды и параметры описываются в главе 3.

Nregex

http://www.nregex.com (рис. 1.4) – это простое веб-приложение, предназначенное для тестирования регулярных выражений, созданное Дэвидом Сераянгом (David Seruyange) на основе технологии .NET. Несмотря на то, что на сайте явно не говорится, какой диалект реализован, – это .NET 1.x, по крайней мере, на момент написания этих строк.

Расположение элементов на странице может немного сбивать с толку. Регулярное выражение следует вводить в поле с надписью Regular Expression (Регулярное выражение), а параметры выражения определяются с помощью флажков под ним. Испытуемый текст вводится в поле, расположенное ниже, где первоначально находится текст «`If I just had $5.00 then "she" wouldn't be so @#$! mad.`». Если предметом испытаний является веб-страница, можно ввести адрес URL в поле Load Target From URL (Загрузить испытуемый текст с адреса URL) и щелкнуть на кнопке Load (Загрузить), расположенной под полем ввода. Если предметом испытаний является файл, расположенный на жестком диске, можно щелкнуть на кнопке Browse (Просмотреть), отыскать требуемый файл и щелкнуть на кнопке Load (Загрузить), расположенной под полем ввода.

Испытуемый текст появится в поле Matches & Replacements (Совпадения и замены), расположенном в центре страницы, в котором будут выделены фрагменты, соответствующие регулярному выражению. Если ввести какой-либо текст в поле Replacement String (Строка замены), то будет отображен результат выполнения операции поиска с заменой. В случае наличия ошибки в регулярном выражении будет выведено многоточие (...).

Поиск соответствий регулярному выражению производится программным кодом платформы .NET, выполняющимся на стороне сервера, поэтому для работы с сайтом необходимо подключение к сети. Если автоматическое обновление страницы выполняется слишком медленно,

Рис. 1.4. *Nregex*

наиболее веротно, что это происходит из-за большого объема испытуемого текста. В этом случае отметьте флажок Manually Evaluate Regex (Производить вычисления вручную), расположенный выше поля ввода регулярного выражения. В результате на странице появится кнопка Evaluate (Вычислить), щелчок на которой будет приводить к обновлению содержимого поля Matches & Replacements (Совпадения и замены).

Rubular

По адресу *http://www.rubular.com* можно найти небольшое веб-приложение, выполняющее проверку регулярных выражений (рис. 1.5), которое использует диалект Ruby 1.8.

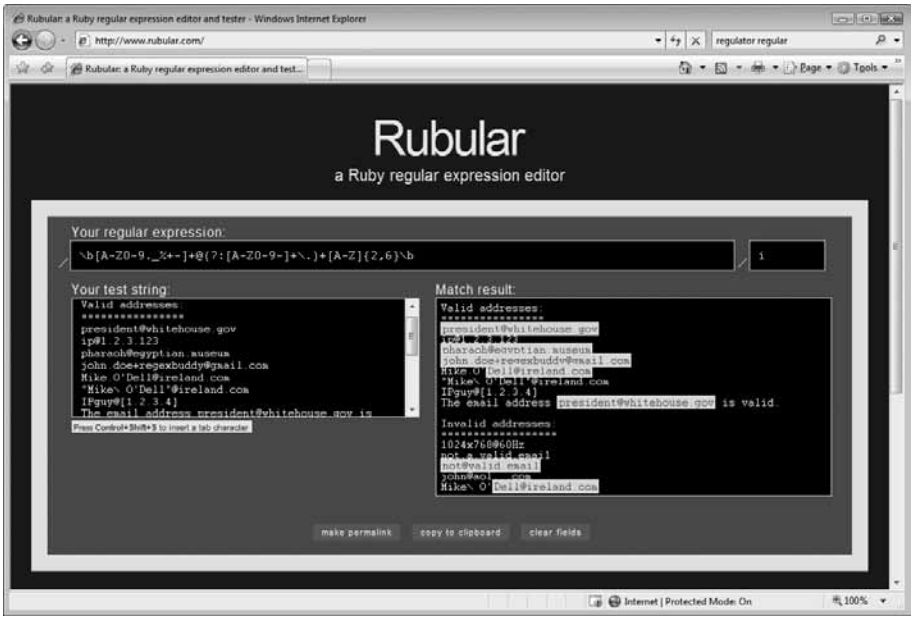


Рис. 1.5. *Rubular*

Регулярное выражение вводится в поле Your regular expression (Ваше регулярное выражение), расположенное между двумя символами слэша. Отключить чувствительность к регистру символов можно, введя символ `i` в небольшое поле ввода, находящееся правее второго символа слэша. При желании точно так же можно определить параметр «точка соответствует концу строки», введя символ m в то же самое поле. Комбинация символов im активизирует оба параметра. Эти соглашения могут показаться недружественными для тех, кто не знаком с языком Ruby, однако они соответствуют синтаксису `/regex/im`, используемому в исходных текстах программ на языке Ruby.

Введите или скопируйте испытуемый текст в поле Your test string (Ваша тестовая строка) и подождите мгновение. В поле Match result (Результат поиска соответствия) появится испытуемый текст с выделенными фрагментами, соответствующими регулярному выражению.

myregexp.com

Сергеем Евдокимовым (Sergey Evdokimov) было создано несколько приложений для разработчиков на языке Java, позволяющих тестировать регулярные выражения. На домашней странице *http://www.myregexp.com* (рис. 1.6) предоставляется доступ к веб-приложению тестирования регулярных выражений. Это Java-апплет, который выполняется броузером. При этом необходимо, чтобы на компьютере клиента была установлена среда выполнения Java 4 (или более поздней версии). Для оцен-

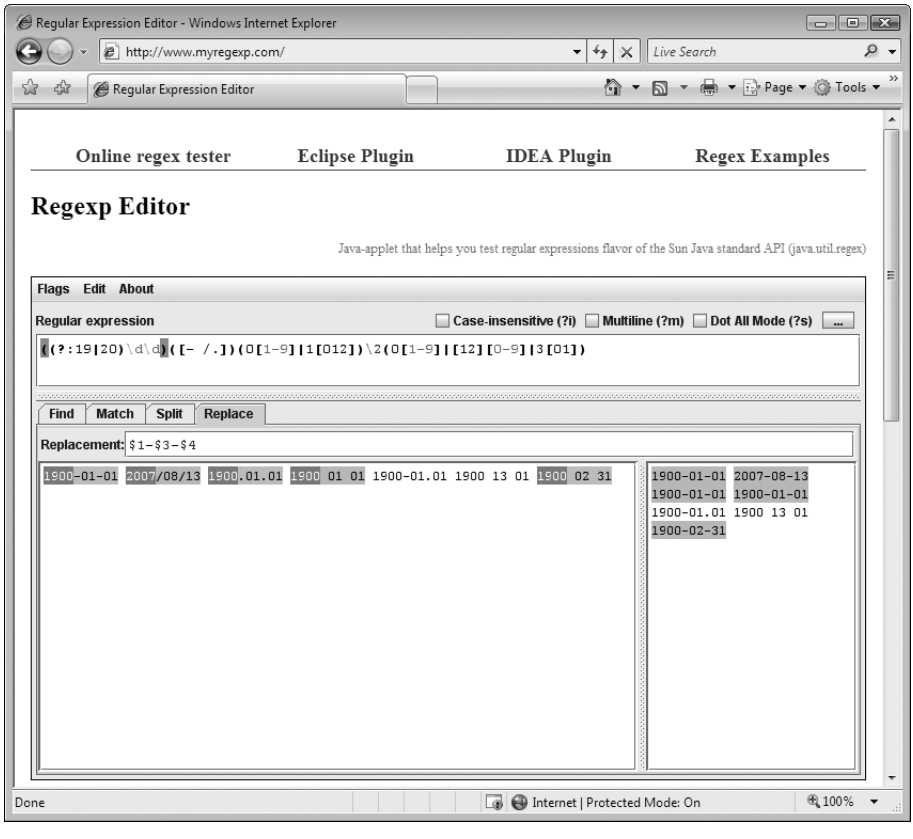


Рис. 1.6. *myregexp.com*

ки регулярных выражений используется пакет `java.util.regex`, который впервые появился в версии Java 4. Под диалектом регулярных выражений «Java» подразумевается именно этот пакет.

Регулярное выражение вводится в поле Regular Expression (Регулярное выражение). Желаемые параметры регулярного выражения устанавливаются с помощью меню Flags (Флаги). Кроме того, три параметра имеют собственные отдельные флажки.

Если необходимо проверить регулярное выражение, уже имеющееся в исходном программном коде на языке Java, достаточно просто скопировать строку целиком в буфер обмена, затем на странице *myregexp.com* щелкнуть на пункте меню Edit (Правка) и выбрать пункт Paste Regex from Java String (Вставить регулярное выражение из строки на языке Java). По завершении редактирования регулярного выражения в том же самом меню можно выбрать пункт Copy Regex for Java Source (Копировать регулярное выражение в формате строки на языке Java). В меню Edit (Правка) имеются похожие команды для JavaScript и XML.

Под полем ввода регулярного выражения имеются четыре вкладки, позволяющие производить разные виды испытаний:

Find (Поиск)

Выделяет все соответствия регулярному выражению, обнаруженные в испытуемом тексте. Эти соответствия отыскиваются с помощью метода `Matcher.find()` языка Java.

Match (Соответствие)

Проверяет, соответствует ли весь испытуемый текст целиком указанному регулярному выражению. Если соответствует, выделяется весь текст целиком. Этот тест демонстрирует работу методов `String.matches()` и `Matcher.matches()`.

Split (Разбиение)

Второе поле справа отображает массив строк, возвращаемых методом `String.split()` или `Pattern.split()` при использовании указанных вами регулярного выражения и испытуемого текста.

Replace (Замена)

После ввода замещающего текста в поле справа будет выведен текст, возвращаемый методом `String.replaceAll()` или `Matcher.replaceAll()`.

Другие приложения тестирования регулярных выражений, созданные Сергеем, можно отыскать с помощью ссылок, расположенных в верхней части страницы *http://www.myregexp.com*. Одно из них является модулем расширения для среды программирования Eclipse, а другое – модулем расширения для среды программирования IntelliJ IDEA.

reAnimator

Еще одно приложение – reAnimator *http://osteele.com/tools/reanimator* (рис. 1.7), созданное Оливером Стилом (Oliver Steele), вовсе не предназначено для оживления умерших регулярных выражений. Это лишь небольшой забавный инструмент, дающий графическое изображение конечных автоматов, которые используются механизмом регулярных выражений для поиска соответствия регулярному выражению.

Синтаксис регулярных выражений, поддерживаемых приложением reAnimator, весьма ограничен. Он совместим со всеми диалектами, рассматриваемыми в этой книге. Любое регулярное выражение, которое можно воспроизвести с помощью reAnimator, будет работать с любым из диалектов, представленных в книге, но это не означает, что обратное утверждение тоже верно. Это обусловлено тем, что регулярные выражения, поддерживаемые приложением reAnimator, являются регулярными в математическом смысле. Краткие пояснения по этому вопросу приводятся во врезке «История появления термина „регулярное выражение"» на стр. 20.

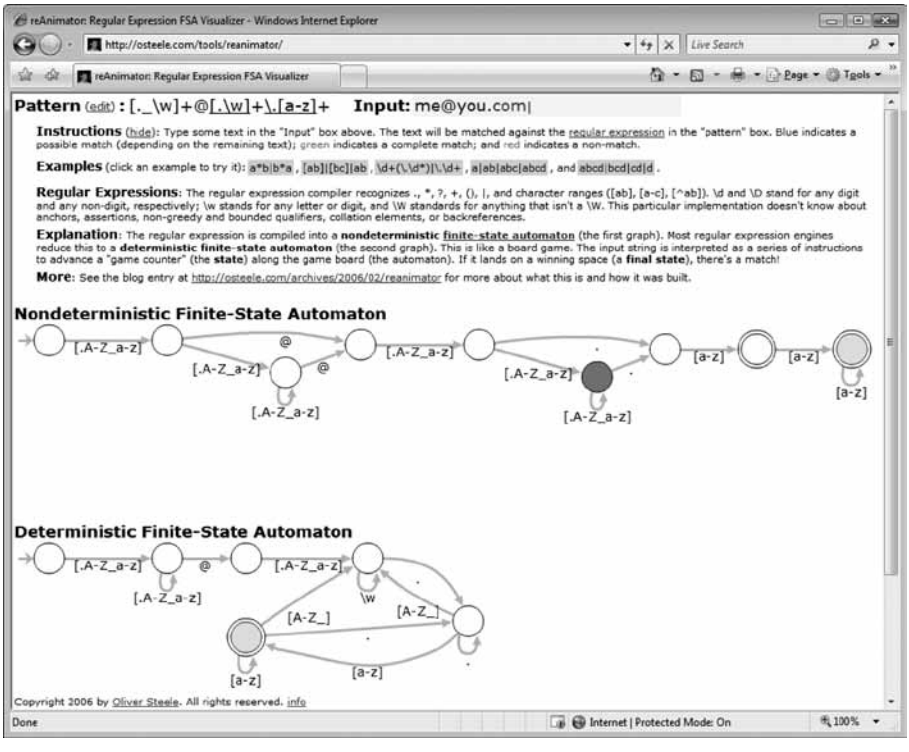


Рис. 1.7. *reAnimator*

Сначала необходимо перейти к полю Pattern (Шаблон) и щелкнуть на кнопке Edit (Правка). Ввести регулярное выражение в поле Pattern (Шаблон) и щелкнуть на кнопке Set (Установить). Затем следует медленно вводить испытуемый текст в поле Input (Ввод).

По мере ввода каждого символа цветные шарики будут перемещаться по изображению конечного автомата, отмечая конечную позицию, достигнутую автоматом к настоящему моменту. Голубые шарики показывают, что конечный автомат принял ввод, но при этом для достижения полного совпадения требуется дополнительный ввод. Зеленые шарики показывают, что ввод полностью соответствует шаблону. Отсутствие шариков говорит о том, что конечный автомат не может отыскать соответствие.

Приложение reAnimator обнаруживает соответствия, только если регулярному выражению соответствует вся строка целиком, как если бы она находилась между якорными элементами ‹^› и ‹$›. Это еще одна особенность выражений, которые являются регулярными в математическом смысле.

Примеры настольных приложений тестирования регулярных выражений

Expresso

Expresso (не следует путать с кофе espresso (эспрессо), богатым кофеином) – это приложение для платформы .NET, позволяющее создавать и тестировать регулярные выражения. Загрузить его можно по адресу *http://www.ultrapico.com/Expresso.htm*. Для его работы на компьютере должна быть установлена платформа .NET 2.0 или более поздней версии.

Для свободной загрузки доступна 60-дневная оценочная версия. По окончании оценочного периода необходимо зарегистрироваться, в противном случае программа Expresso перестанет работать. Регистрация выполняется бесплатно, но требует указать парням из Ultrapico свой адрес электронной почты. Регистрационный ключ высылается по электронной почте.

Внешний вид приложения Expresso приводится на рис. 1.8. Поле Regular Expression (Регулярное выражение), предназначенное для ввода регулярных выражений, постоянно находится в зоне видимости. Подсветка синтаксиса отсутствует. В поле Regex Analyzer (Анализатор регулярного выражения) автоматически создается краткий отчет о регулярном выражении на английском языке. Это поле также отображается постоянно.

В нижней части вкладки Design Mode (Режим проектирования) имеется возможность устанавливать такие параметры поиска совпадений, как Ignore Case (Игнорировать регистр символов). Большую часть экранного пространства занимает линия вкладок, где можно выбирать элементы для добавления в регулярное выражение. Те, кто имеет два монитора или один большой монитор, могут щелчком мыши на кнопке Undock (Отделить) отделить линию вкладок, преобразовав ее в плавающую панель. Кроме того, создавать регулярные выражения можно также во вкладке Test Mode (Режим тестирования).

Во вкладке Test Mode (Режим тестирования) следует ввести или вставить испытуемый текст в левый нижний угол и щелкнуть на кнопке Run Match (Выполнить поиск соответствий), чтобы получить в поле Search Results (Результаты поиска) полный список всех соответствий. В испытуемом тексте ничего не выделяется. Чтобы выделить совпадение в испытуемом тексте, следует щелкнуть на совпадении в наборе результатов.

Во вкладке Expression Library (Библиотека выражений) выводится список примеров регулярных выражений и список последних регулярных выражений. Испытуемое регулярное выражение добавляется в этот список всякий раз, когда выполняется щелчок на кнопке Run Match (Выпол-

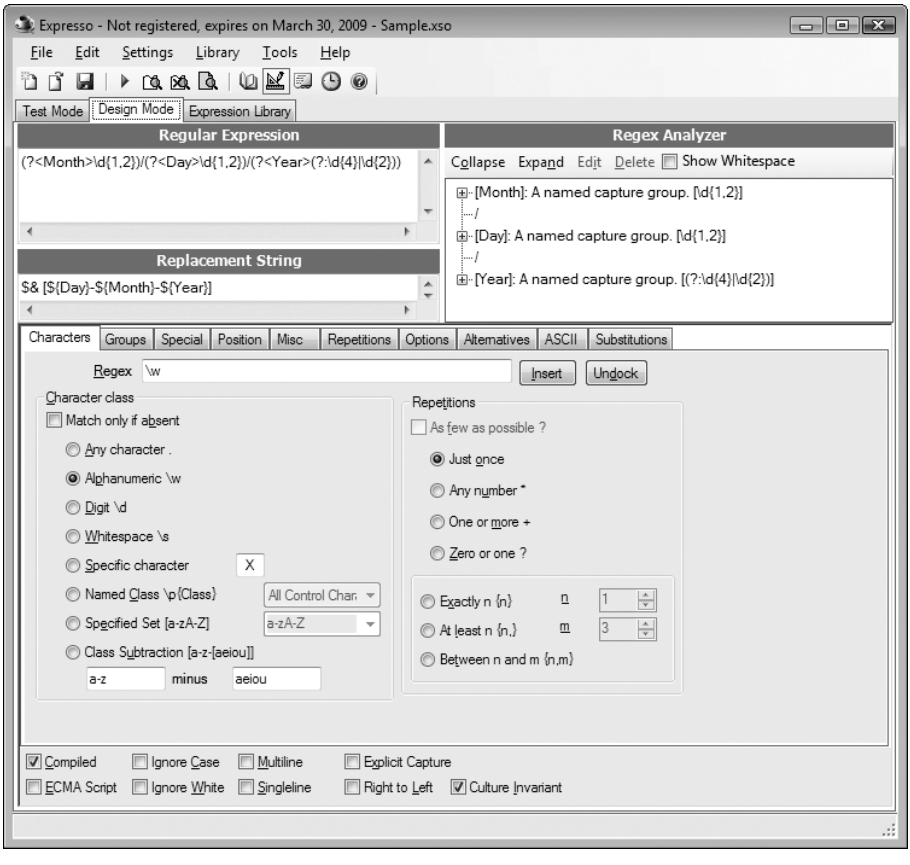


Рис. 1.8. *Expresso*

нить поиск соответствий). Библиотека допускает возможность редактирования посредством меню Library (Библиотека) в главном меню программы.

The Regulator

Приложение Regulator, которое можно загрузить по адресу *http://sourceforge.net/projects/regulator*, не может использоваться в дыхательных аппаратах для подводного плавания или в газовых баллонах – это еще одно приложение на платформе .NET, предназначенное для создания и тестирования регулярных выражений. Последняя версия приложения требует наличия установленной версии .NET 2.0 или более поздней. Кроме того, доступны для загрузки более старые версии приложения для .NET 1.x. Regulator – это открытое программное обеспечение, для приобретения которого не требуется платить деньги или проходить процедуру регистрации.

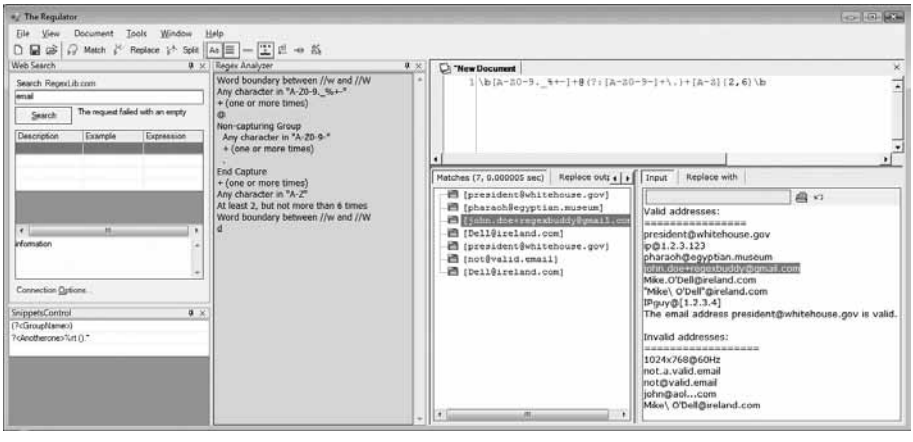


Рис. 1.9. *The Regulator*

Вся функциональность приложения Regulator доступна на единственном экране (рис. 1.9). Вкладка New Document (Новый документ) – это место, где вводится регулярное выражение. Подсветка синтаксиса выполняется автоматически, но синтаксические ошибки, присутствующие в регулярном выражении, не выделяются. Щелчком правой кнопки мыши вызывается контекстное меню, в котором можно выбирать добавляемые элементы регулярного выражения. Параметры регулярного выражения можно устанавливать с помощью кнопок на главной панели инструментов. Ярлыки на кнопках могут ввести в заблуждение, поэтому дожидайтесь появления всплывающих подсказок, чтобы определить назначение каждой кнопки.

Ниже и правее области ввода регулярного выражения имеется кнопка Input (Ввод), щелчок на которой приводит к появлению области, куда можно вставить испытуемый текст. Щелчок на кнопке Replace with (Заменить на) открывает область ввода замещающего текста, используемого в операции поиска с заменой. Ниже и левее области ввода регулярного выражения выводятся результаты работы регулярного выражения. Результат не обновляется автоматически – чтобы обновить результат, необходимо щелкнуть на кнопке Match (Соответствие), Replace (Заменить) или Split (Разбить) в панели инструментов. Фрагменты в испытуемом тексте не выделяются. Щелчок на совпадении в списке результатов приводит к выделению соответствующего фрагмента в испытуемом тексте.

В панели Regex Analyzer (Анализатор регулярного выражения) выводится простой отчет о регулярном выражении на английском языке, но этот отчет не обновляется автоматически и не является интерактивным. Чтобы обновить отчет, следует выбрать пункт Regex Analyzer (Анализатор регулярного выражения) в меню View (Вид), даже если панель анализатора видима. Щелчок мышью в панели анализатора приводит лишь к перемещению текстового курсора.

grep

Название *grep* происходит от команды `g/re/p`, которая выполняет поиск по регулярному выражению в текстовом редакторе *ed* в системе UNIX, одном из первых приложений, обеспечивших поддержку регулярных выражений. Эта команда оказалась настолько востребованной во всех системах UNIX, что была выделена в отдельную утилиту grep, обеспечивающую поиск в файлах с использованием регулярных выражений. Если вы используете систему UNIX, Linux или OS X, введите команду `man grep` в терминале, чтобы получить подробное описание этой утилиты.

Следующие три инструмента являются приложениями Windows, которые работают точно так же, как и утилита grep, и обладают дополнительными возможностями.

PowerGREP

Программа PowerGREP была разработана Яном Гойвертсом (Jan Goyvaerts), одним из авторов этой книги. Это, пожалуй, одна из наиболее мощных реализаций утилиты *grep* для платформы Windows (рис. 1.10). Программа PowerGREP использует собственный диалект регулярных выражений, соединяющий в себе лучшие черты диалектов, рассматриваемых в этой книге. Этот диалект в программе RegexBuddy помечен как «JGsoft».

Чтобы приступить к поиску с применением регулярного выражения, достаточно просто выбрать пункт Clear (Очистить) в меню Action (Операция) и ввести регулярное выражение в поле Search (Найти), находящееся в панели Action (Операция). В панели File Selector (Выбор файла) щелчком мыши следует выбрать требуемый каталог, а в меню File Selector (Выбор файла) выбрать пункт Include File or Folder (Включить файл или папку) или Include Folder and Subfolders (Включить папку и подпапки). После этого можно выбрать пункт Execute (Выполнить) в меню Action (Операция) и тем самым запустить процедуру поиска.

Чтобы запустить поиск с заменой, после очистки операции следует выбрать в раскрывающемся списке Action type (Тип операции), находящемся в левом верхнем углу панели Action (Операция), пункт Search-and-replace (Поиск с заменой). После этого ниже поля Search (Найти) появится поле ввода Replace (Заменить). В это поле следует ввести замещающий текст. Все остальные шаги совпадают с выполнением операции поиска.

Программа PowerGREP обладает уникальной возможностью одновременно использовать до трех списков регулярных выражений, причем в каждом из списков может находиться произвольное число регулярных выражений. В предыдущих двух параграфах описывается, как выполняются простые операции поиска, которые могут быть выполнены с помощью любой утилиты grep, однако чтобы оценить весь потен-

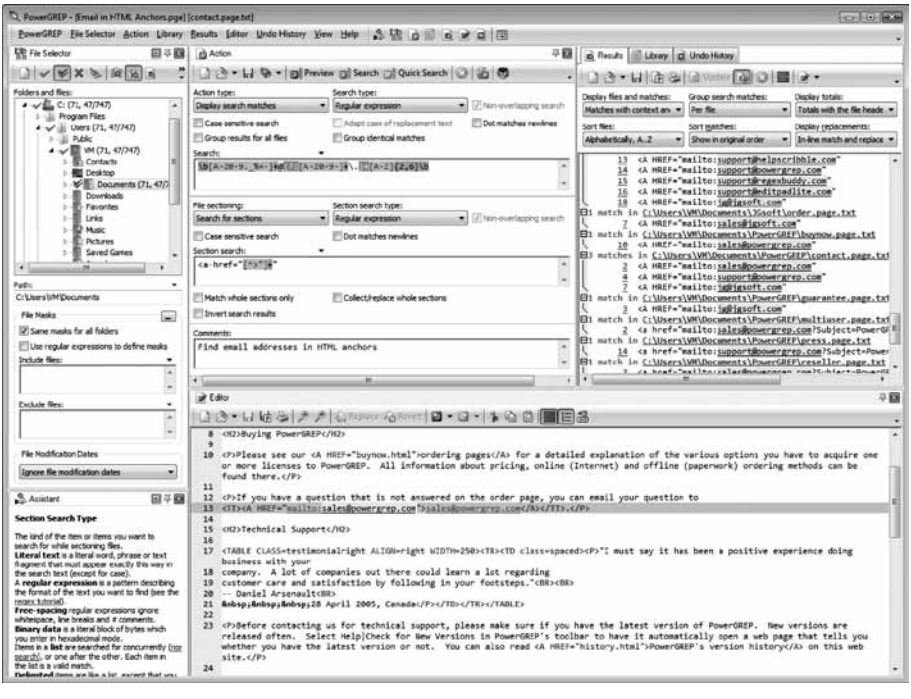


Рис. 1.10. PowerGREP

циал программы PowerGREP, придется заняться чтением обширной документации к программе.

Программа PowerGREP работает в системах Windows 98, ME, 2000, XP и Vista. Вы можете бесплатно загрузить оценочную версию программы по адресу *http://www.powergrep.com/PowerGREPCookbook.exe*. За исключением возможности сохранять результаты и библиотеки, оценочная версия обладает полной функциональностью в течение 15 дней фактического использования. Но хотя оценочная версия не позволяет сохранять результаты, отображаемые в панели Results (Результаты), тем не менее она будет изменять содержимое файлов при выполнении операции поиска с заменой, как и полнофункциональная версия.

Windows Grep

Программа Windows Grep (*http://www.wingrep.com*) – это один из старейших grep-инструментов для Windows. Возраст программы можно определить по пользовательскому интерфейсу (рис. 1.11), но она прекрасно справляется со всем, что заявлено. Она поддерживает ограниченный диалект регулярных выражений под названием POSIX ERE. Следует отметить, что этот диалект поддерживает тот же синтаксис, что и диалекты, рассматриваемые в этой книге. Программа Windows Grep от-

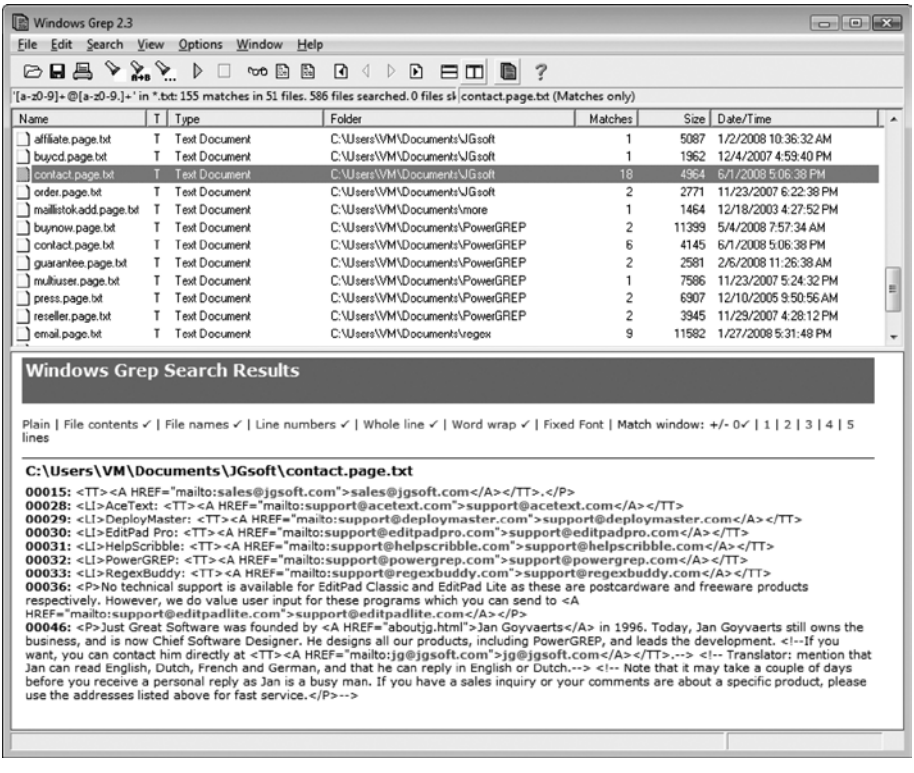


Рис. 1.11. *Windows Grep*

носится к категории условно бесплатного программного обеспечения (shareware), то есть вы можете бесплатно загрузить ее, но предполагается, что вы заплатите за нее, если пожелаете пользоваться ею.

Чтобы выполнить поиск, следует выбрать пункт Search (Найти) в меню Search (Найти). Содержимое окна программы отличается в зависимости от выбранного в меню Options (Параметры) режима: Beginner Mode (Режим новичка) или Expert Mode (Режим эксперта). Для новичков предоставляется пошаговый мастер, а для экспертов – диалог с вкладками.

По окончании подготовительных операций программа Windows Grep немедленно приступает к поиску и представляет список файлов, в которых были обнаружены совпадения. Если щелкнуть на файле в списке, в нижней панели можно увидеть найденные в нем соответствия, а двойной щелчок открывает файл. Чтобы нижняя панель была видна постоянно, следует выбрать пункт All Matches (Все соответствия) в меню View (Вид).

Чтобы произвести поиск с заменой, в меню Search (Найти) следует выбрать пункт Replace (Заменить).

RegexRenamer

Программа RegexRenamer (рис. 1.12) в действительности не является grep-инструментом. Она не выполняет поиск по содержимому файлов, вместо этого она отыскивает и изменяет имена файлов. Загрузить программу можно по адресу *http://regexrenamer.sourceforge.net*. Программа RegexRenamer требует наличия установленной версии 2.0 платформы Microsoft .NET.

Регулярное выражение вводится в поле Match (Соответствие), а замещающий текст – в поле Replace (Заменить). Чтобы включить режим нечувствительности к регистру символов, следует пометить флажок /i, при включенном флажке /g производится замена не только первого, но и всех остальных найденных соответствий в каждом имени файла. Флажок /x включает режим игнорирования дополнительных пробелов, что само по себе малополезно, так как в поле регулярного выражения можно ввести только одну строку.

Дерево каталогов слева используется для выбора папки, в которой хранятся файлы, которые требуется переименовать. В правом верхнем углу можно определить маску для файлов или регулярное выражение фильтра. Это позволяет ограничить перечень файлов, к именам которых будет применяться операция поиска с заменой. Возможность использования одного регулярного выражения для фильтрации, а другого для выполнения операции поиска с заменой – преимущество по срав-

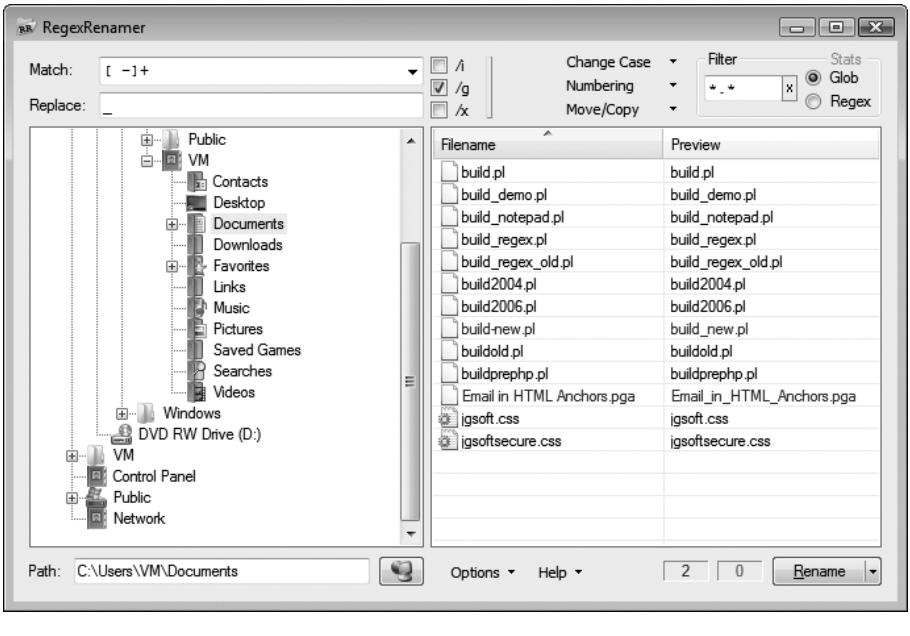


Рис. 1.12. *RegexRenamer*

нению с попыткой решения обеих задач с помощью единственного регулярного выражения.

Популярные текстовые редакторы

Большинство современных текстовых редакторов обладают, по меньшей мере, самой примитивной поддержкой регулярных выражений. В диалоге поиска или поиска с заменой обычно присутствует флажок, включающий режим использования регулярного выражения. Некоторые редакторы, такие как EditPad Pro, также используют регулярные выражения для обработки текста, например для организации подсветки синтаксиса или создания списков классов и функций. Все эти особенности редакторов описываются в сопроводительной документации. Ниже приводится список некоторых текстовых редакторов, обладающих поддержкой регулярных выражений:

- Boxer Text Editor (PCRE)
- Dreamweaver (JavaScript)
- EditPad Pro (собственный диалект, вобравший в себя лучшие черты диалектов, рассматриваемых в этой книге. В программе RegexBuddy именуется «JGsoft»)
- Multi-Edit (PCRE, если выбран параметр Perl)
- NoteTab (PCRE)
- UltraEdit (PCRE)
- TextMate (Ruby 1.9 [Oniguruma])

2

Основные навыки владения регулярными выражениями

Задачи, представленные в этой главе, не являются реальными задачами, которые будут ставить перед вами ваш босс или ваши заказчики. Это, скорее, технические задачи, с которыми вы будете сталкиваться при создании регулярных выражений, предназначенных для решения практических задач. Например, в первом рецепте объясняется, как с помощью регулярного выражения отыскать определенный текст. Собственно само по себе это не может быть целью, потому что нет никакой необходимости использовать регулярные выражения, когда требуется отыскать определенный текст. Однако при создании регулярных выражений вам наверняка потребуется отыскивать соответствия с определенным текстом, поэтому вам необходимо знать, какие символы следует экранировать, а как раз об этом и рассказывается в рецепте 2.1.

Первыми приводятся рецепты, описывающие очень простые приемы использования регулярных выражений. Если у вас уже имеется некоторый опыт работы с регулярными выражениями, вы можете просто бегло просмотреть их или вообще пропустить. Последующие рецепты в этой главе определенно будут содержать новую для вас информацию, если, конечно, вы не прочитали книгу «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E. F. Friedl), выпущенную издательством O'Reilly,[1] от корки до корки.

Мы придумывали рецепты для этой главы так, чтобы каждый из них описывал один аспект синтаксиса регулярных выражений. Все вместе они образуют законченный учебник по регулярным выражениям. Рекомендуем прочитать его от начала до конца, чтобы твердо усвоить основы регулярных выражений. Однако можно сразу перейти к регулярным выражениям, имеющим практическую ценность, которые

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

приводятся в главах с 4 по 8, и возвращаться по ссылкам к этой главе при встрече с незнакомыми синтаксическими конструкциями.

В этой вводной главе рассматриваются только регулярные выражения и полностью исключаются из рассмотрения какие-либо вопросы, связанные с программированием. Следующая глава наполнена листингами программного кода. Вы можете заглянуть в раздел «Языки программирования и диалекты регулярных выражений» главы 3, чтобы узнать, какой диалект регулярных выражений используется в вашем языке программирования. Сами диалекты, о которых будет говориться в этой главе, были представлены в разделе «Диалекты регулярных выражений, рассматриваемые в этой книге» на стр. 22.

2.1. Соответствие литеральному тексту

Задача

Создать регулярное выражение, в точности соответствующее следующему восхитительному предложению: `The punctuation characters in the ASCII table are: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

Решение

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
!"#\$%&'\(\)\*\+,-\./:;<=>\?@\[\\]\^_`\{\|}~
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любое регулярное выражение, в котором отсутствуют символы `$()*+.?[\^{|`, просто соответствует само себе. Например, чтобы отыскать в тексте фразу `Mary had a little lamb`, достаточно воспользоваться регулярным выражением `‹Mary•had•a•little•lamb›`. При этом совершенно неважно, отмечен ли флажок `Regular Expression` (Регулярное выражение) в диалоге поиска текстового редактора.

Двенадцать знаков препинания, которые обеспечивают всю магическую силу регулярных выражений, называются *метасимволами*. Если необходимо, чтобы регулярное выражение находило соответствие с этими символами буквально, их следует *экранировать*, помещая перед ними символ обратного слэша. То есть регулярному выражению

```
\$\(\)\*\+\.\?\[\\\^\{\|
```

соответствует текст

```
$()*+.?[\^{|
```

Обратите внимание, что в списке отсутствуют: закрывающая квадратная скобка], дефис - и закрывающая фигурная скобка }. Первые два символа интерпретируются как метасимволы, только если они стоят после неэкранированной открывающей квадратной скобки [, а } – только после неэкранированного символа {. Нет никакой необходимости экранировать символ }. Правила использования метасимволов в блоках, заключенных между символами [и], описываются в рецепте 2.3.

Экранирование любых других не алфавитно-цифровых символов не влияет на работу регулярного выражения, по крайней мере, это утверждение справедливо для всех диалектов, рассматриваемых в этой книге. Экранирование алфавитно-цифровых символов либо придает им особое значение, либо рассматривается как синтаксическая ошибка.

Те, кто плохо знаком с регулярными выражениями, часто стремятся экранировать каждый знак препинания, попадающий в поле зрения. Не старайтесь показать всем, что вы – новичок. Разумно используйте экранирование. Частокол ненужных символов обратного слэша осложняет чтение регулярного выражения, особенно когда все эти символы обратного слэша требуется удваивать, чтобы представить регулярное выражение как строковый литерал в исходном тексте программы.

Варианты

Экранирование блока

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
\Q!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~\E
```

Параметры: нет
Диалекты: Java 6, PCRE, Perl

Диалекты Perl, PCRE и Java поддерживают конструкции ‹\Q› и ‹\E›. Конструкция ‹\Q› подавляет интерпретацию всех метасимволов, включая и символ обратного слэша, пока не встретится конструкция ‹\E›. Если опустить конструкцию ‹\E›, все символы, находящиеся в регулярном выражении после конструкции ‹\Q›, будут интерпретироваться буквально.

Единственное преимущество выражения ‹\Q...\E› состоит в том, что его проще читать, чем ‹\.\.\.›.

Несмотря на то, что версии Java 4 и 5 поддерживают эту особенность, тем не менее, лучше ею не пользоваться. Ошибки в реализации приводят к тому, что регулярному выражению с конструкцией ‹\Q...\E› соответствует совсем не то, что соответствует этому же регулярному выражению в диалектах PCRE, Perl и Java 6. Эти ошибки были исправлены в версии Java 6, благодаря чему этот диалект дает тот же результат, что и диалекты PCRE и Perl.

Поиск без учета регистра символов

```
ascii
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
(?i)ascii
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

По умолчанию регулярные выражения чувствительны к регистру символов. Выражению ‹regex› соответствует текст regex, но не Regex, REGEX или ReGeX. Чтобы регулярному выражению ‹regex› соответствовали все эти варианты, необходимо включить режим нечувствительности к регистру символов.

В большинстве приложений для этого достаточно установить или сбросить флажок. Все языки программирования, обсуждаемые в следующей главе, имеют флаг или свойство, с помощью которого можно управлять режимом чувствительности регулярного выражения к регистру символов. В рецепте 3.4, в следующей главе, описывается, как в программном коде применять параметры регулярного выражения, соответствующие приводимому в книге регулярному выражению.

Если режим нечувствительности к регистру символо не был включен за пределами регулярного выражения, это можно сделать с помощью модификатора режима ‹(?i)›, например ‹(?i)regex›. Этот прием работает в диалектах .NET, Java, PCRE, Perl, Python и Ruby.

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают локальные модификаторы режима, которые воздействуют только на часть регулярного выражения. Например, регулярному выражению ‹sensitive(?i)caseless(?-i)sensitive› соответствует текст sensitiveCASELESSsensitive, но не SENSITIVEcaselessSENSITIVE. Модификатор ‹(?i)› включает режим нечувствительности к регистру символов до конца регулярного выражения, а модификатор ‹(?-i)› отключает его до конца регулярного выражения. Они действуют как простые переключатели.

В рецепте 2.10 показано, как использовать локальные модификаторы режима с группами.

См. также

Рецепты 2.3 и 5.14.

2.2. Соответствие непечатным символам

Задача

Написать регулярное выражение, которому соответствовала бы строка, состоящая из следующих управляющих символов ASCII: bell, escape, form feed, line feed, carriage return, horizontal tab, vertical tab. Эти символы имеют следующие шестнадцатеричные коды ASCII: 07, 1B, 0C, 0A, 0D, 09, 0B.

Решение

```
\a\e\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Python, Ruby

```
\x07\x1B\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Python, Ruby

```
\a\e\f\n\r\t\0x0B
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Для представления семи наиболее часто используемых управляющих символов ASCII имеются специальные экранированные последовательности. Все они состоят из символа обратного слэша и следующей за ним буквы. Это тот же самый синтаксис, что используется во многих языках программирования для представления управляющих символов в строковых литералах. В табл. 2.1 приводится перечень наиболее часто используемых непечатных символов и их представления.

Таблица 2.1. Непечатные символы

Представление	Значение	Шестнадцатеричное представление
‹ \a ›	bell	0x07
‹ \e ›	escape	0x1B
‹ \f ›	form feed	0x0C
‹ \n ›	line feed (newline)	0x0A
‹ \r ›	carriage return	0x0D

‹\t›	horizontal tab	0x09
‹\v›	vertical tab	0x0B

Стандарт ECMA-262 не поддерживает последовательности ‹\a› и ‹\e›. Поэтому в примерах для JavaScript мы использовали другой синтаксис, хотя многие броузеры поддерживают ‹\a› и ‹\e›. Язык Perl не поддерживает экранированную последовательность ‹\v› (вертикальная табуляция), поэтому в Perl ее следует заменить экранированной последовательностью с шестнадцатеричным (\x0B) или восьмеричным (\013) кодом.

Эти управляющие символы и альтернативный синтаксис, который демонстрируется в следующем разделе, с одинаковым успехом могут использоваться в регулярных выражениях как внутри, так и за пределами символьных классов.

Варианты представления непечатных символов

26 управляющих символов

```
\cG\x1B\cL\cJ\cM\cI\cK
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Ruby 1.9

С помощью последовательностей от ‹\cA› до ‹\cZ› определяется соответствие с любым из 26 управляющих символов, занимающим позицию от 1 до 26 в таблице символов ASCII. Символ c должен указываться в нижнем регистре. Буква, следующая за ним, в большинстве диалектов может указываться в любом регистре, но мы рекомендуем всегда использовать символы верхнего регистра. В диалекте Java это обязательное условие.

Этот синтаксис может пригодиться, если вы привыкли вводить управляющие символы в консоли, нажимая клавишу Ctrl одновременно с алфавитной клавишей. Комбинация клавиш Ctrl-H в терминалах соответствует символу забоя (backspace). В регулярных выражениях символу забоя соответствует последовательность ‹\cH›.

Этот синтаксис не поддерживается механизмом регулярных выражений языка Python и классическим механизмом Ruby в Ruby 1.8, но он поддерживается механизмом Onigurama в Ruby 1.9.

Управляющий символ escape, занимающий позицию 27 в таблице ASCII, не имеет эквивалента в английском алфавите, поэтому в наших регулярных выражениях он представлен последовательностью ‹\x1B›.

7-битный набор символов

```
\x07\x1B\x0C\x0A\x0D\x09\x0B
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Последовательности из экранированного символа `\x` в нижнем регистре, за которым следуют две шестнадцатеричные цифры в верхнем регистре, соответствует один символ из набора ASCII. На рис. 2.1 показано соответствие шестнадцатеричных комбинаций от ‹`\x00`› до ‹`\x7F`› и символов из набора ASCII. Таблица упорядочена так, чтобы первая цифра в комбинации возрастала по направлению вниз, а вторая – вправо.

Соответствие комбинациям от ‹`\x80`› до ‹`\xFF`› зависит от того, как они интерпретируются вашим механизмом регулярных выражений и в какой кодировке набран испытуемый текст. Мы не рекомендуем вам использовать последовательности от ‹`\x80`› до ‹`\xFF`›. Вместо них лучше использовать кодовые пункты Юникода, описываемые в рецепте 2.7.

В случае использования Ruby 1.8 или библиотеки PCRE, скомпилированной без поддержки кодировки UTF-8, воспользоваться кодовыми пунктами Юникода будет невозможно. Ruby 1.8 и PCRE без поддержки UTF-8 представляют собой 8-битовые механизмы регулярных выражений. Они ничего не знают о кодировках символов и многобайтовых символах. Последовательность ‹`\xAA`› в этих механизмах просто соответствует байту со значением 0xAA независимо от того, представляет ли этот байт отдельный символ с кодом 0xAA или является частью многобайтового символа.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рис. 2.1. *Таблица символов ASCII*

См. также

Рецепт 2.7.

2.3. Сопоставление с одним символом из нескольких

Задача

Создать регулярное выражение, которому соответствовали бы любые ошибочные написания слова `calendar`, чтобы иметь возможность отыскивать это слово в документе, не полагаясь на грамотность автора. Пред-

полагается, что на месте любой гласной буквы может использоваться символ a или e. Создать второе регулярное выражение, которому соответствовала бы единственная шестнадцатеричная цифра. Создать третье регулярное выражение, которому соответствовал бы единственный символ, не являющийся шестнадцатеричной цифрой.

Решение

Ошибки в слове calendar

```
c[ae]l[ae]nd[ae]r
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричная цифра

```
[a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Нешестнадцатеричная цифра

```
[^a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Последовательность, заключенная в квадратные скобки, называется *символьным классом*. Символьному классу соответствует единственный символ, совпадающий с любым из перечисленных символов. В первом регулярном выражении указано три класса, каждому из которых соответствует символ a или e. Это обеспечивает свободу выбора. Когда с помощью этого регулярного выражения проверяется слово calendar, первому символьному классу соответствует символ a, второму – e и третьему – a.

За пределами символьных классов метасимволами являются двенадцать знаков препинания. Внутри символьных классов только четыре имеют специальное назначение: \, ^, - и]. В диалектах Java и .NET открывающая квадратная скобка [также является метасимволом внутри символьных классов. Все остальные символы интерпретируются буквально и просто добавляют себя в символьный класс. Регулярному выражению ‹[$()*+.?{|]› соответствует любой из девяти символов, перечисленных между квадратными скобками.

Символ обратного слэша всегда экранирует символ, следующий за ним, так же как и за пределами символьного класса. Экранированный символ может представлять единственный символ, а также начало или ко-

нец диапазона. Другие четыре метасимвола приобретают специальное назначение, только если находятся в определенной позиции. Они могут представлять литералы символов в символьных классах без дополнительного экранирования путем помещения их в позицию, где они утрачивают специальное назначение. Выражение ‹[][^-]› иллюстрирует эту особенность, по крайней мере, если вы не используете реализацию JavaScript, строго придерживающуюся стандарта. Однако мы рекомендуем всегда экранировать эти метасимволы; предыдущее регулярное выражение следовало бы записать так: ‹[\]\[\^\-]›. Экранирование метасимволов упрощает понимание регулярного выражения.

Алфавитно-цифровые символы не должны экранироваться обратным слэшем, так как в этом случае либо возникает ошибка, либо создается специальный символ (имеющий специальное значение в регулярных выражениях). В наших пояснениях к некоторым специальным символам, например в рецепте 2.2, мы указывали, что они могут использоваться внутри символьных классов. Все эти специальные символы состоят из слэша и символа, иногда сопровождаемых рядом других символов. Так, символьному классу ‹[\r\n]› соответствует символ возврата каретки (\r) или перевода строки (\n).

Символ *крышки* (^) в символьном классе означает отрицание, если он следует непосредственно за открывающей квадратной скобкой. Он обеспечивает соответствие символьному классу любых символов, которые *отсутствуют* в списке. Инвертированному символьному классу соответствуют символы конца строки, если они отсутствуют в инвертированном символьном классе.

Символ *дефиса* (-) определяет *диапазон*, когда находится между двумя символами. Диапазон в символьном классе включает в себя символ, стоящий перед дефисом, символ, стоящий после дефиса, и все символы, расположенные между ними в порядке следования их числовых кодов. Чтобы узнать, какие числовые коды соответствуют символам, необходимо обратиться к таблице символов ASCII или Юникода. Символьный класс ‹[A-z]› включает в себя все символы, расположенные в таблице ASCII между символом верхнего регистра A и символом нижнего регистра z. Этот диапазон также включает в себя некоторые знаки препинания; так, символьному классу ‹[A-Z\[\\\]\^_`a-z]› соответствуют те же самые символы, только объявлен он более явно. Мы рекомендуем создавать диапазоны, включающие только цифры или только буквы, причем только верхнего или только нижнего регистра.

Перевернутые диапазоны, такие как ‹[z-a]›, не допускаются.

Варианты

Сокращения

[a-fA-F\d]

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Шесть специальных символов, состоящих из обратного слэша и следующего за ним алфавитного символа, представляют *сокращенную* форму записи символьных классов. Их можно использовать как внутри, так и за пределами символьных классов. Последовательностям ‹\d› и ‹[\d]› соответствует один цифровой символ. Каждой сокращенной форме записи, включающей символ нижнего регистра, соответствует сокращенная форма записи, включающая символ верхнего регистра, имеющая противоположный смысл. Так, последовательности ‹\D› соответствует любой символ, не являющийся цифрой, а ее эквивалентом является запись ‹[^\d]›.

Метасимволу ‹\w› соответствует один *символ слова*. Символ слова – это символ, который может входить в состав слова. Сюда относятся алфавитные символы, цифры и символ подчеркивания. Такой набор символов может показаться странным, но он был выбран таким, потому что из этих символов обычно составляются идентификаторы в языках программирования. Метасимволу ‹\W› соответствует любой символ, который не может являться частью такого слова.

В диалектах Java, JavaScript, PCRE и Ruby метасимвол ‹\w› всегда идентичен классу ‹[a-zA-Z0-9_]›. В .NET и Perl он также включает алфавитно-цифровые символы любых других алфавитов (кириллица, тайский алфавит и прочие). В Python символы других алфавитов включаются, только если при создании регулярного выражения передавался флаг UNICODE или U. К метасимволу ‹\d› применяются те же правила во всех диалектах. В .NET и Perl цифры из других алфавитов всегда включаются в класс, тогда как в Python они включаются, только если передавался флаг UNICODE или U.

Метасимволу ‹\s› соответствует любой *пробельный символ*. Сюда входят символы пробела, табуляции и символы конца строки. В .NET, Perl и JavaScript метасимволу ‹\s› также соответствуют все символы, определяемые стандартом Юникода как пробельные. Примечательно, что в диалекте JavaScript для ‹\s› используется Юникод, а для ‹\d› и ‹\w› – ASCII. Метасимволу ‹\S› соответствуют все символы, не соответствующие метасимволу ‹\s›.

Еще одно противоречие несет в себе метасимвол ‹\b›. Метасимвол ‹\b› не является краткой формой представления символьного класса, он обозначает границу слова. Можно было бы ожидать, что ‹\b› поддерживает Юникод, когда эту кодировку поддерживает метасимвол ‹\w›, и только ASCII, когда ‹\w› поддерживает только ASCII, но это не всегда так. Подробнее об этом рассказывается в подразделе «Символы слов» на стр. 70, в рецепте 2.6.

Поиск без учета регистра символов

```
(?i)[A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?i)[^A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Нечувствительность к регистру символов, определяемая с помощью внешнего флага (рецепт 3.4) или с помощью модификатора режима внутри регулярного выражения (рецепт 2.1), также оказывает воздействие и на символьные классы. Только что представленные регулярные выражения эквивалентны регулярным выражениям, представленным в первоначальном решении.

Диалект JavaScript также следует этому правилу, но он не поддерживает модификатор ‹(?i)›. Чтобы отключить чувствительность регулярного выражения к регистру символов в JavaScript, необходимо установить флаг /i при его создании.

Особенности, характерные для разных диалектов

Разность символьных классов в .NET

```
[a-zA-Z0-9-[g-zG-Z]]
```

Данному регулярному выражению соответствует одна шестнадцатеричная цифра, но реализовано оно косвенным образом. Базовому символьному классу соответствует любой алфавитно-цифровой символ, а вложенному классу, который вычитается из базового, символы от g до z. Вложенный вычитаемый класс должен следовать за базовым классом и предваряться символом дефиса: ‹[class-[subtract]]›.

Операцию *вычитания* символьных классов удобно использовать, например при работе со свойствами Юникода, блоками и алфавитами. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\P{N}› соответствует любой символ, у которого отсутствует свойство Number. Разности этих двух классов соответствует любая из 10 цифр тайского алфавита.

Объединение, разность и пересечение символьных классов в Java

```
[a-f[A-F][0-9]]
[a-f[A-F[0-9]]]
```

Диалект Java допускает вкладывать один символьный класс в другой. При непосредственном включении одного символьного класса в другой получается *объединение* двух классов. Допускается создавать столько

уровней вложения, сколько потребуется. Два приведенных выше выражения дают тот же эффект, что и следующее, в котором отсутствуют дополнительные квадратные скобки:

```
[\w&&[a-fA-F0-9\s]]
```

Это выражение могло бы получить приз в соревновании на самое запутанное выражение. Базовому символьному классу соответствует любой символ слова. Вложенному классу соответствует любая шестнадцатеричная цифра и любой пробельный символ. Результатом является *пересечение* двух классов, которому соответствуют только шестнадцатеричные цифры и ничто иное. Так как базовому классу не соответствуют пробельные символы, а вложенному классу не соответствуют символы ‹[g-zG-Z_]›, они исключаются из окончательного символьного класса и остаются только шестнадцатеричные цифры:

```
[a-zA-Z0-9&&[^g-zG-Z]]
```

Данному регулярному выражению соответствует одна шестнадцатеричная цифра и тоже косвенным образом. Базовому символьному классу соответствует любой алфавитно-цифровой символ, а вложенному классу, который вычитается из базового, символы от g до z. Этот вложенный класс инвертируется и ему предшествуют два символа амперсанда: ‹[class&&[^subtract]]›.

Пересечение и разность символьных классов удобно использовать, в частности, при работе со свойствами Юникода, блоками и алфавитами. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\p{N}› соответствует любой символ, у которого присутствует свойство Number. Последовательности ‹[\p{InThai}&&[\p{N}]]› соответствует любая из 10 цифр тайского алфавита.

Если вас интересуют различия между возможными комбинациями ‹\p› в регулярных выражениях, их описание вы найдете в рецепте 2.7.

См. также

Рецепты 2.1, 2.2 и 2.7.

2.4. Сопоставление с любым символом

Задача

Создать регулярное выражение, которому соответствует любой символ, заключенный в апострофы. Одному решению должен соответствовать любой одиночный символ, за исключением конца строки, заключенный в апострофы. Второму решению должен соответствовать действительно любой символ, включая символы конца строки.

Решение

Любой символ, за исключением символа конца строки

'.'

Параметры: нет (параметр «точке соответствуют границы строк» должен быть сброшен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любой символ, включая символы конца строки

'.'

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

'[\s\S]'

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любой символ за исключением символа конца строки

Точка – это один из старейших и простейших элементов регулярных выражений. Она означает обязательное соответствие любому одиночному символу.

Однако не всегда очевидно, что означает фраза *любой символ*. Самые ранние инструменты, обеспечивающие поддержку регулярных выражений, обрабатывали содержимое файлов строку за строкой, поэтому отсутствовала потребность искать соответствие с концом строки в испытуемом тексте. Языки программирования, рассматриваемые в этой книге, обрабатывают текст целиком, как единое целое, независимо от наличия в нем разрывов строк. Если действительно возникает потребность производить обработку текста построчно, можно дополнить программу кодом, который будет разбивать исходный текст на массив строк и затем применять регулярное выражение к каждой строке в массиве. Как это делается, показано в рецепте 3.21 в следующей главе.

Ларри Уолл (Larry Wall), создатель языка Perl, сохранил в языке Perl традиционное поведение инструментов построчной обработки текста, где точка никогда не соответствовала концу строки (\n). Все остальные диалекты, рассматриваемые в этой книге, последовали его примеру. То есть регулярному выражению ‹.› соответствует любой символ, *за исключением* символа перевода строки.

Любой символ, включая символы конца строки

Если необходимо позволить регулярному выражению обрабатывать сразу несколько строк, следует включить параметр «точке соответствуют границы строк». Этот параметр маскируется под разными названиями. В языке Perl и во многих других он носит сбивающее с толку название «режим единственной строки» (single line mode), тогда как в Java он называется «режим точка – это все» (dot all mode). Подробное описание этого вопроса вы найдете в рецепте 3.4 в следующей главе. Независимо от того, как называется этот параметр в вашем любимом языке программирования, называйте его про себя режимом «точке соответствуют границы строк». Это все, что дает данный параметр.

Для JavaScript, где отсутствует параметр «точке соответствуют границы строк», необходимо альтернативное решение. Как описывается в рецепте 2.3, метасимволу ‹\s› соответствуют любые пробельные символы, тогда как метасимволу ‹\S› соответствуют любые другие символы, не соответствующие метасимволу ‹\s›. Объединив их, можно получить символьный класс ‹[\s\S]›, которому будут соответствовать любые символы, включая символы конца строки. Тот же эффект дают символьные классы ‹[\d\D]› и ‹[\w\W]›.

Злоупотребление точкой

Точка является элементом регулярных выражений, которым злоупотребляют наиболее часто. Выражение ‹\d\d.\d\d.\d\d› представляет не лучший способ поиска дат. Ему прекрасно соответствует дата 05/06/08, но ему также соответствует текст 99/99/99. Хуже того, ему соответствует число 12345678.

Регулярное выражение, которому соответствуют только корректные даты, – это тема для более поздней главы. Но заменить точку на более подходящий символьный класс совсем несложно. Выражение ‹\d\d[/.\-]\d\d[/.\-]\d\d› позволяет использовать в качестве символа-разделителя слэш, точку или дефис. Данному регулярному выражению по-прежнему соответствует текст 99/99/99, но число 12345678 уже не соответствует.

Это просто совпадение, что в предыдущем примере точка включена в символьные классы. Внутри символьных классов точка интерпретируется буквально. В этом есть определенный смысл, так как в некоторых странах, например в Германии, точка используется в качестве символа-разделителя в датах.

Точку следует использовать, только когда действительно допускается совпадение с любым символом. Во всех остальных случаях лучше использовать символьные классы и инвертированные символьные классы.

Варианты

`(?s)'.'`

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python

`(?m)'.'`

Параметры: нет
Диалект: Ruby

Если не удалось включить режим «точке соответствуют границы строк» вне регулярного выражения, можно добавить модификатор режима в начало выражения. В рецепте 2.1 в подразделе «Поиск без учета регистра символов» на стр. 51 мы объяснили концепцию модификаторов режима и говорили, что в JavaScript они не поддерживаются.

`‹(?s)›` – это модификатор режима «точке соответствуют границы строк» в .NET, Java, PCRE, Perl и Python. Символ `s` здесь обозначает режим «single line» (единственная строка), под которым в языке Perl подразумевается режим «точке соответствуют границы строк».

Терминология настолько запутывающая, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Для включения режима «точке соответствуют границы строк» в Ruby используется модификатор `‹(?m)›`. Здесь используется другой символ, но функциональность та же самая. Новый механизм в Ruby 1.9 продолжает использовать модификатор `‹(?m)›` для включения режима «точке соответствуют границы строк». Значение модификатора `‹(?m)›` в языке Perl описывается в рецепте 2.5.

См. также

Рецепты 2.3, 3.4 и 3.21.

2.5. Сопоставление в начале и/или в конце строки

Задача

Создать четыре регулярных выражения. Первому выражению должно соответствовать слово `alpha`, но только если оно находится в самом начале испытуемого текста. Второму выражению должно соответствовать слово `omega`, но только если оно находится в самом конце испытуемого текста. Третьему выражению должно соответствовать слово `begin`, но только если оно находится в начале строки. Четвертому выражению должно соответствовать слово `end`, но только если оно находится в конце строки.

Решение

В начале испытуемого текста

```
^alpha
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\Aalpha
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В конце испытуемого текста

```
omega$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
omega\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В начале строки

```
^begin
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В конце строки

```
end$
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Якорные метасимволы и строки

Метасимволы регулярных выражений ‹^›, ‹$›, ‹\A›, ‹\Z› и ‹\z› называются *якорями*. Они не соответствуют каким-либо символам, вместо этого они обозначают определенные позиции, фактически осуществляя привязку регулярного выражения к этим позициям.

Строка – это часть испытуемого текста, которая находится между начальной и конечной границами строки. Если в испытуемом тексте отсутствуют границы строк, то весь текст рассматривается как одна строка. Так, следующий текст состоит из четырех строк, по одной для

слов one и two, затем следует пустая строка, и далее строка со словом four:

```
one
two

four
```

В программном коде этот текст можно было бы представить как one[LF] two[LF][LF]four.

В начале испытуемого текста

Якорному метасимволу ‹\A› всегда соответствует точка самого начала испытуемого текста, непосредственно перед первым символом. Это единственное место в тексте, соответствующее данному метасимволу. Поместив ‹\A› в начало регулярного выражения, можно проверить, начинается ли испытуемый текст с искомого текста. Символ «A» должен быть символом верхнего регистра.

Диалект JavaScript не поддерживает метасимвол ‹\A›.

Якорный метасимвол ‹^› является эквивалентом метасимволу ‹\A› при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр.

Если вы не используете диалект JavaScript, мы рекомендуем всегда вместо метасимвола ‹^› использовать метасимвол ‹\A›. Значение метасимвола ‹\A› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В конце испытуемого текста

Якорным метасимволам ‹\Z› и ‹\z› всегда соответствует позиция в конце испытуемого текста непосредственно за последним символом. Поместив ‹\Z› или ‹\z› в конец регулярного выражения, можно проверить, оканчивается ли испытуемый текст искомым текстом.

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают оба метасимвола ‹\Z› и ‹\z›. Диалект Python поддерживает только метасимвол ‹\Z›. Диалект JavaScript не поддерживает ни один из них.

Различия между метасимволами ‹\Z› и ‹\z› начинают проявляться, когда в испытуемом тексте последним символом является символ конца строки. В этом случае метасимволу ‹\Z› соответствует самый конец испытуемого текста, после завершающего символа конца строки, а также позиция непосредственно перед этим символом конца строки. Преимущество состоит в том, что можно выполнить поиск ‹omega\Z›, не беспокоясь об удалении завершающего символа конца строки в конце испытуемого текста. Читая содержимое файла строку за строкой, одни

инструменты включают символ конца строки, другие – нет; метасимвол ‹\Z› позволяет ликвидировать это различие. Метасимволу ‹\z› соответствует только самый конец испытуемого текста, поэтому он не будет совпадать с текстом в регулярном выражении, если в нем имеется завершающий символ конца строки.

Якорный метасимвол ‹$› является эквивалентом метасимволу ‹\Z›, при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр. Так же как и ‹\Z›, метасимволу ‹$› соответствует самый конец испытуемого текста, а также позиция перед завершающим символом конца строки.

Чтобы прояснить все эти тонкости и неясности, рассмотрим пример на языке Perl. Предположим, что переменная $/ (текущий символ-разделитель записей) имеет значение по умолчанию \n, тогда следующая инструкция на языке Perl прочитает одну строку, введенную с терминала (из потока стандартного ввода):

```
$line = <>;
```

В языке Perl символ перевода строки будет сохранен в переменной $line. Вследствие этого регулярное выражение, такое как ‹end•of•input.\z›, не совпадет с содержимым переменной. Но совпадения будут обнаружены выражениями ‹end•of•input.\Z› и ‹end•of•input.$›, потому что они игнорируют завершающий символ перевода строки.

Чтобы упростить обработку данных, программисты на языке Perl часто принудительно отбрасывают символ перевода строки инструкцией:

```
chomp $line;
```

После выполнения этой операции совпадение будет обнаружено всеми тремя выражениями. (Технически инструкция chomp удаляет из строки текущий символ-разделитель записей.)

Если вы не пользуетесь JavaScript, мы рекомендуем всегда использовать метасимвол ‹\Z› вместо ‹$›. Значение метасимвола ‹\Z› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В начале строки

По умолчанию метасимволу ‹^› соответствует позиция в начале испытуемого текста, как и метасимволу ‹\A›. Только в Ruby метасимволу ‹^› всегда соответствует начало строки. Во всех остальных диалектах необходимо включать дополнительный параметр, чтобы символ крышки и знак доллара совпадали с границами строк. Этот параметр обычно называется «многострочным» режимом.

Не следует путать этот режим с режимом «единственной строки», который более известен как режим «точке соответствуют границы строк». «Многострочный» режим оказывает влияние только на поведение символа крышки и знака доллара, а режим «единственной строки» оказывает влияние только на поведение точки, как разъяснялось в рецепте 2.4. Нет ничего невозможного в том, чтобы одновременно включить «многострочный» режим и режим «единственной строки». По умолчанию оба параметра отключены.

При корректной установке параметра метасимволу ‹^› будет соответствовать позиция в начале каждой строки испытуемого текста. Строго говоря, метасимвол будет совпадать с позицией непосредственно перед самым первым символом в файле, что происходит всегда, а также с позициями после каждого символа конца строки. Символ крышки в выражении ‹\n^› избыточен, потому что ‹^› всегда совпадает с позицией после ‹\n›.

В конце строки

По умолчанию метасимволу ‹$› соответствует позиция только в конце испытуемого текста или перед завершающим символом конца строки, как и метасимволу ‹\Z›. Только в Ruby метасимволу ‹$› всегда соответствует конец каждой строки. Во всех остальных диалектах необходимо включать дополнительный «многострочный» режим, чтобы символ крышки и знак доллара совпадали с границами строк.

При корректной установке параметра метасимволу ‹$› будет соответствовать позиция в конце каждой строки испытуемого текста. (Безусловно, он также будет совпадать с позицией за самым последним символом в тексте, потому что эта позиция всегда считается концом строки.) Символ доллара в выражении ‹$\n› избыточен, потому что ‹$› всегда совпадает с позицией перед ‹\n›.

Совпадения нулевой длины

Нет ничего необычного в том, чтобы регулярное выражение содержало только один или более якорных метасимволов. Такое регулярное выражение будет находить совпадение нулевой длины в каждой позиции, где будет обнаруживаться соответствие якорному метасимволу. При объединении нескольких якорных метасимволов все они должны совпадать с одной и той же позицией, чтобы регулярное выражение находило соответствие.

Такое регулярное выражение может использоваться в операции поиска с заменой. Замещение метасимволов ‹\A› или ‹\Z› позволяет добавлять что-либо в начало или в конец испытуемого текста. Замещение метасимволов ‹^› или ‹$› в режиме «символам ^ и $ соответствуют границы строк» позволяет добавлять что-либо в начало или в конец каждой строки испытуемого текста.

Комбинация двух якорных метасимволов позволит проверить наличие пустых строк или отсутствие ввода. Комбинации ‹\A\Z› соответствует пустая строка, а также строка, состоящая из единственного символа перевода строки. Комбинации ‹\A\z› соответствует только пустая строка. Комбинации ‹^$› в режиме «символам ^ и $ соответствуют границы строк» будет соответствовать каждая пустая строка в испытуемом тексте.

Варианты

`(?m)^begin`

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python

`(?m)end$`

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python

Если нет возможности включить режим «символам ^ и $ соответствуют границы строк» вне регулярного выражения, в начало регулярного выражения можно поместить модификатор режима. О модификаторах режима и об отсутствии их поддержки в JavaScript рассказывалось в рецепте 2.1 в подразделе «Поиск без учета регистра символов» на стр. 51.

‹(?m)› — это модификатор режима «символам ^ и $ соответствуют границы строк» в диалектах .NET, Java, PCRE, Perl и Python. Символ m соответствует названию «multiline» (многострочный) режим, под которым в языке Perl подразумевается режим «символам ^ и $ соответствуют границы строк».

Как уже говорилось выше, терминология оказалась настолько запутывающей, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Модификатор ‹(?m)› в языке Ruby используется для включения режима «точке соответствуют границы строк». То есть в языке Ruby модификатор ‹(?m)› не оказывает влияния на поведение якорных метасимволов ‹^› и ‹$›. В Ruby метасимволам ‹^› и ‹$› всегда соответствуют начало и конец каждой строки.

За исключением путаницы в символах модификаторов, в языке Ruby выбор применения метасимволов ‹^› и ‹$› исключительно к границам строк следует признать удачным. Если вы не используете JavaScript, мы рекомендуем придерживаться такого же выбора в своих регулярных выражениях.

Ян следовал этой идее при разработке программ EditPad Pro и PowerGREP. В них вы не найдете флажок с надписью «^ and $ match at line breaks» (символам ^ и $ соответствуют границы строк), хотя имеется флажок с надписью dot matches newlines (точке соответствуют границы строк). Если добавить в начало регулярного выражения моди-

фикатор ‹(?-m)›, то, чтобы привязать регулярное выражение к началу или к концу файла, необходимо будет использовать метасимволы ‹\A› и ‹\Z›.

См. также

Рецепты 3.4 и 3.21.

2.6. Сопоставление с целыми словами

Задача

Создать регулярное выражение, которому соответствовало бы слово cat в тексте My cat is brown, но которое не находило бы соответствие в словах category или bobcat. Создать еще одно регулярное выражение, которому соответствовала бы последовательность cat в тексте staccato, но которое не находило бы соответствие ни в одном из трех предыдущих случаев.

Решение

На границах слова

```
\bcat\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Не на границах слова

```
\Bcat\B
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

На границах слова

Метасимвол ‹\b› называется *границей слова*. Ему соответствует позиция начала или конца слова. Сам по себе этот метасимвол дает совпадение нулевой длины. Метасимвол ‹\b› – является якорным, точно так же, как и метасимволы, представленные в предыдущем разделе.

Строго говоря, метасимволу ‹\b› соответствуют следующие три позиции:

- Перед первым символом испытуемого текста, если первый символ является символом слова.

- После последнего символа испытуемого текста, если последний символ является символом слова.
- Между двумя символами испытуемого текста, когда один из них является символом слова, а другой – нет.

Ни в одном из диалектов, обсуждаемых в этой книге, нет отдельных метасимволов, которым соответствовала бы позиция только в начале или только в конце слова. Однако в них нет никакой необходимости, если только вам не потребуется регулярное выражение, не содержащее ничего, кроме границы слова. Символы перед или после метасимвола ‹\b› в регулярном выражении помогут указать, с какой позицией будет совпадать ‹\b›. Метасимвол ‹\b› в выражениях ‹\bx› и ‹!\b› мог бы совпадать только с началом слова. Метасимвол ‹\b› в выражениях ‹x\b› и ‹\b!› мог бы совпадать только с концом слова. А выражения ‹x\bx› и ‹!\b!› никогда не будут находить соответствий.

Чтобы выполнить поиск «только целого слова» с помощью регулярного выражения, нужно просто поместить искомое слово между двумя границами слов, как это предложено в решении ‹\bcat\b›. Первый метасимвол ‹\b› требует, чтобы символ ‹c› находился в самом начале строки или после символа, не являющегося символом слова. Второй метасимвол ‹\b› требует, чтобы символ ‹t› находился в самом конце строки или перед символом, не являющимся символом слова.

Символы конца строки не являются символами слова. Поэтому метасимвол ‹\b› будет совпадать с позицией после символа конца строки, если сразу же вслед за ним располагается символ слова. Он также будет совпадать с позицией перед символом конца строки, если ему непосредственно предшествует символ слова. Благодаря этому слово, занимающее всю строку, будет обнаружено в процессе поиска «только целых слов». На поведение метасимвола ‹\b› не влияет выбор «многострочного» режима или наличие модификатора ‹(?m)›, что является одной из причин, почему в этой книге «многострочный» режим сопоставляется с режимом «символам ^ и $ соответствуют границы строк».

Не на границах слова

Метасимволу ‹\B› соответствует любая позиция в испытуемом тексте, которая не соответствует метасимволу ‹\b›. Отсюда следует, что метасимволу ‹\B› соответствует любая позиция, не являющаяся началом или концом слова.

Строго говоря, метасимволу ‹\B› соответствуют следующие пять позиций:

- Перед первым символом испытуемого текста, если первый символ не является символом слова.
- После последнего символа испытуемого текста, если последний символ не является символом слова.

- Между двумя символами слова.
- Между двумя символами, не являющимися символами слова.
- Пустая строка.

Выражению ‹\Bcat\B› соответствует последовательность символов cat в тексте staccato, но не в текстах My cat is brown, category или bobcat.

Чтобы выполнить поиск с условием, противоположным требованию «только целое слово» (то есть чтобы совпадение обнаруживалось в словах staccato, category и bobcat, но не обнаруживалось в тексте My cat is brown), необходимо с помощью операции выбора объединить подвыражения ‹\Bcat› и ‹cat\B› в выражение ‹\Bcat|cat\B›. Подвыражению ‹\Bcat› соответствует последовательность символов cat в словах staccato и bobcat. Подвыражению ‹cat\B› соответствует последовательность символов cat в слове category (и в слове staccato, если подвыражение ‹\Bcat› не позаботилось о нем раньше). Оператор выбора рассматривается в рецепте 2.8.

Символы слов

За всеми этими разговорами о границах слов мы ничего не сказали о том, что такое *символ слова*. Символ слова – это символ, который может являться частью слова. В рецепте 2.3 в подразделе «Сокращения» на стр. 57 говорилось о том, какие символы включены в состав символьного класса ‹\w›, который соответствует одному символу слова. К сожалению, для метасимвола ‹\b› история выглядит иначе.

Несмотря на то, что все диалекты, рассматриваемые в этой книге, поддерживают метасимволы ‹\b› и ‹\B›, тем не менее, они по-разному определяют, какие символы являются символами слова.

В диалектах .NET, JavaScript, PCRE, Perl, Python и Ruby метасимволу ‹\b› соответствует позиция между двумя символами, один из которых соответствует символьному классу ‹\w›, а другой – символьному классу ‹\W›. Метасимволу ‹\B› всегда соответствует позиция между двумя символами, когда оба они соответствуют либо символьному классу ‹\w›, либо символьному классу ‹\W›.

В диалектах JavaScript, PCRE и Ruby символами слов считаются только символы ASCII. Символьный класс ‹\w› в них идентичен классу ‹[a-zA-Z0-9_]›. При использовании этих диалектов можно выполнить поиск «только целого слова» в тексте, написанном на языке, где используются исключительно символы от A до Z без диакритических знаков, например на английском. Но эти диалекты не могут использоваться для поиска «только целых слов» в текстах на других языках, таких как испанский или русский.

Диалекты .NET и Perl интерпретируют буквы и цифры из любых алфавитов как символы слова. В этих диалектах можно выполнить поиск

«только целого слова» в текстах на любом языке, включая и те, которые не используют латинский алфавит.

В языке Python имеется возможность выбора. Символы, не входящие в набор ASCII, могут включаться в поиск, только если при создании регулярного выражения был передан флаг `UNICODE` или `U`. Этот флаг в равной степени воздействует на метасимволы `‹\b›` и `‹\w›`.

Диалект Java отличается некоторой непоследовательностью. Классу `‹\w›` в нем соответствуют только символы ASCII. Однако метасимвол `‹\b›` поддерживает Юникод и может применяться при работе с любыми алфавитами. В Java выражению `‹\b\w\b›` соответствует одна буква латинского алфавита, цифра или символ подчеркивания, который не является частью слова ни на одном языке. Выражение `‹\bкошка\b›` корректно совпадет со словом `кошка` в тексте на русском языке, потому что метасимвол `‹\b›` поддерживает Юникод. Но выражение `‹\w+›` не совпадет ни с одним русским словом, потому что класс `‹\w›` поддерживает только символы ASCII.

См. также

Рецепт 2.3.

2.7. Кодовые пункты Юникода, свойства, блоки и алфавиты

Задача

С помощью регулярного выражения отыскать символ торговой марки (™), указав в нем кодовый пункт Юникода вместо копирования и вставки фактического символа. При желании можно копировать и вставлять сам символ; в конце концов символ торговой марки – это всего лишь обычный литерал несмотря на то, что его нельзя ввести с клавиатуры непосредственно. Литералы обсуждались в рецепте 2.1.

Создать регулярное выражение, которому соответствовали бы любые символы, обладающие свойством Юникода «Currency Symbol» (символ денежной единицы). Свойства Юникода также называются категориями.

Создать регулярное выражение, которому соответствовали бы любые символы, принадлежащие блоку Юникода «Greek Extended».

Создать регулярное выражение, которому соответствовали бы любые символы, которые в соответствии со стандартом Юникода принадлежат греческому алфавиту.

Создать регулярное выражение, которому соответствовали бы графемы, то есть символы, состоящие из базового символа и дополнительных комбинационных знаков.

Решение

Кодовый пункт Юникода

```
\u2122
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python
Данное регулярное выражение будет работать в Python, только если оно будет оформлено, как строка Юникода: `u"\u2122"`.

```
\x{2122}
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона `/u`. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Свойство, или категория Юникода

```
\p{Sc}
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона `/u`. Диалекты JavaScript и Python не поддерживают свойства Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Блок Юникода

```
\p{IsGreekExtended}
```

Параметры: нет
Диалекты: .NET, Perl

```
\p{InGreekExtended}
```

Параметры: нет
Диалекты: Java, Perl

Диалекты JavaScript, PCRE, Python и Ruby не поддерживают блоки Юникода.

Алфавит в Юникоде

```
\p{Greek}
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

Для работы с алфавитами необходима библиотека PCRE версии 6.5 или выше, а кроме того, библиотека должна быть скомпилирована с поддержкой UTF-8. В языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты .NET, JavaScript и Python не поддерживают свойства Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Графема в Юникоде

```
\X
```

Параметры: нет
Диалекты: PCRE, Perl

В диалектах PCRE и Perl имеется специальный метасимвол, совпадающий с графемами, но при этом поддерживается альтернативный способ, связанный с использованием свойств Юникода.

```
\P{M}\p{M}*
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты JavaScript и Python не поддерживают свойства Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Обсуждение

Кодовый пункт Юникода

Кодовый пункт – это одна запись в базе данных символов Юникода. Кодовый пункт – это не *символ*, впрочем, это зависит от того, какой смысл вкладывается в слово «символ». То, что на экране выглядит как символ, в Юникоде называется *графемой*.

Кодовый пункт Юникода U+2122 представляет символ «знака торговой марки». Отыскать его можно с помощью конструкции ‹\u2122› или ‹\x{2122}›, в зависимости от используемого диалекта регулярных выражений.

При использовании метасимвола ‹\u› необходимо указывать точно четыре шестнадцатеричные цифры. Это означает, что его можно использовать для поиска кодовых пунктов Юникода в диапазоне от U+0000 по U+FFFF. При использовании метасимвола ‹\x› допускается указывать любое число шестнадцатеричных цифр, что обеспечивает поддержку всех кодовых пунктов в диапазоне от U+000000 по U+10FFFF. Кодовый пункт U+00E0 будет соответствовать как выражению ‹\x{E0}›, так и выражению ‹\x{00E0}›. Кодовые пункты U+100000 используются очень ред-

ко и в настоящее время весьма слабо поддерживаются шрифтами и операционными системами.

Кодовые пункты могут использоваться внутри символьных классов и за их пределами.

Свойства, или категории Юникода

Каждый кодовый пункт Юникода обладает точно одним *свойством Юникода*, или принадлежит единственной *категории Юникода*. Эти термины являются взаимозаменяемыми. Всего существует 30 категорий Юникода, сгруппированных в 7 суперкатегорий:

`‹\p{L}›`

Любая буква любого языка.

`‹\p{Ll}›`

Любая строчная буква, для которой имеется заглавный вариант.

`‹\p{Lu}›`

Заглавная буква, для которой имеется строчный вариант.

`‹\p{Lt}›`

Буква, с которой начинается слово, когда только первая буква слова записывается как заглавная.

`‹\p{Lm}›`

Специальный символ, который используется как буква.

`‹\p{Lo}›`

Символ или идеограмма, для которого отсутствуют варианты нижнего и верхнего регистра.

`‹\p{PM}›`

Символ, который должен объединяться с другим символом (диакритические знаки, умляуты, описывающие рамки и прочее).

`‹\p{Mn}›`

Символ, который должен объединяться с другим символом, не требующим дополнительного пространства (например, диакритические знаки, умляуты и прочее).

`‹\p{Mc}›`

Символ, который должен объединяться с другим символом, требующим дополнительного пространства (гласные буквы во многих восточных алфавитах).

`‹\p{Me}›`

Символ, описанный вокруг другого символа (окружность, квадрат, клавишный колпачок и прочее).

‹\p{Z}›

Любые типы пробельных символов или невидимых символов-разделителей.

‹\p{Zs}›

Невидимые пробельные символы, занимающие дополнительное пространство.

‹\p{Zl}›

Символ-разделитель строк U+2028.

‹\p{Zp}›

Символ-разделитель параграфов U+2029.

‹\p{S}›

Математические символы, знаки денежных единиц, декоративные элементы, символы для рисования рамок, и прочее.

‹\p{Sm}›

Любые математические символы.

‹\p{Sc}›

Любые знаки денежных единиц.

‹\p{Sk}›

Комбинационный знак (метка) как самостоятельный символ.

‹\p{So}›

Различные символы, которые не являются математическими знаками, знаками денежных единиц или комбинационными символами.

‹\p{N}›

Различные числовые символы в любом алфавите.

‹\p{Nd}›

Цифры от 0 до 9 в любом алфавите, за исключением идеографических алфавитов.

‹\p{Nl}›

Числа, которые выглядят как буквы, например римские цифры.

‹\p{No}›

Надстрочные или подстрочные цифры или числа, не являющиеся цифрами 0...9 (кроме чисел из идеографических алфавитов).

‹\p{P}›

Различные знаки пунктуации.

‹\p{Pd}›

Различные тире и дефисы.

‹\p{Ps}›

Различные открывающие скобки.

‹\p{Pe}›

Различные закрывающие скобки.

‹\p{Pi}›

Различные открывающие кавычки.

‹\p{Pf}›

Различные закрывающие кавычки.

‹\p{Pc}›

Знаки пунктуации, например подчеркивание, соединяющие слова.

‹\p{Po}›

Различные знаки пунктуации, не являющиеся дефисами, тире, скобками, кавычками или символами, соединяющими слова.

‹\p{C}›

Неотображаемые управляющие символы и неиспользуемые кодовые пункты.

‹\p{Cc}›

Управляющие символы 0x00...0x1F в кодировке ASCII и 0x80...0x9F – в кодировке Latin-1.

‹\p{Cf}›

Неотображаемые символы, являющиеся признаками форматирования.

‹\p{Co}›

Кодовые пункты, предназначенные для закрытого использования.

‹\p{Cs}›

Одна половина суррогатной пары в кодировке UTF-16.

‹\p{Cn}›

Кодовые пункты, которым не присвоены символы.

Комбинация ‹\p{Ll}› совпадает с единственным кодовым пунктом, обладающим свойством Ll, или «lowercase letter» (символ нижнего регистра). Комбинация ‹\p{L}› является сокращенной формой записи символьного класса ‹[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]›, которому соответствует любой кодовый пункт из категории «letter» (буква).

Метасимвол ‹\P› – это инвертированная версия метасимвола ‹\p›. Комбинации ‹\P{Ll}› соответствует единственный кодовый пункт, не имеющий свойства Ll. Комбинации ‹\P{L}› соответствует единственный кодовый пункт, не имеющий какого-либо свойства из суперкатегории «letter»

(буква). Это не аналог символьного класса ‹[\P{Ll}\P{Lu}\P{Lt}\P{Lm}\P{Lo}]›, которому соответствует любой кодовый пункт. Комбинации ‹\P{Ll}› соответствуют кодовые пункты, обладающие свойством Lu (или любым другим свойством, за исключением Ll), тогда как комбинация ‹\P{Lu}› будет совпадать и с кодовыми пунктами, имеющими свойство Ll. Объединение только этих двух комбинаций в символьный класс уже обеспечивает совпадение с любыми возможными кодовыми пунктами.

Блок Юникода

Все кодовые пункты в базе данных символов Юникода подразделяются на блоки. Каждый содержит единственный диапазон кодовых пунктов. Кодовые пункты с U+0000 по U+FFFF делятся на 105 кодовых блоков:

```
U+0000…U+007F    <\p{InBasic_Latin}>
U+0080…U+00FF    <\p{InLatin-1_Supplement}>
U+0100…U+017F    <\p{InLatin_Extended-A}>
U+0180…U+024F    <\p{InLatin_Extended-B}>
U+0250…U+02AF    <\p{InIPA_Extensions}>
U+02B0…U+02FF    <\p{InSpacing_Modifier_Letters}>
U+0300…U+036F    <\p{InCombining_Diacritical_Marks}>
U+0370…U+03FF    <\p{InGreek_and_Coptic}>
U+0400…U+04FF    <\p{InCyrillic}>
U+0500…U+052F    <\p{InCyrillic_Supplementary}>
U+0530…U+058F    <\p{InArmenian}>
U+0590…U+05FF    <\p{InHebrew}>
U+0600…U+06FF    <\p{InArabic}>
U+0700…U+074F    <\p{InSyriac}>
U+0780…U+07BF    <\p{InThaana}>
U+0900…U+097F    <\p{InDevanagari}>
U+0980…U+09FF    <\p{InBengali}>
U+0A00…U+0A7F    <\p{InGurmukhi}>
U+0A80…U+0AFF    <\p{InGujarati}>
U+0B00…U+0B7F    <\p{InOriya}>
U+0B80…U+0BFF    <\p{InTamil}>
U+0C00…U+0C7F    <\p{InTelugu}>
U+0C80…U+0CFF    <\p{InKannada}>
U+0D00…U+0D7F    <\p{InMalayalam}>
U+0D80…U+0DFF    <\p{InSinhala}>
U+0E00…U+0E7F    <\p{InThai}>
U+0E80…U+0EFF    <\p{InLao}>
U+0F00…U+0FFF    <\p{InTibetan}>
U+1000…U+109F    <\p{InMyanmar}>
```

```
U+10A0…U+10FF    <\p{InGeorgian}>
U+1100…U+11FF    <\p{InHangul_Jamo}>
U+1200…U+137F    <\p{InEthiopic}>
U+13A0…U+13FF    <\p{InCherokee}>
U+1400…U+167F    <\p{InUnified_Canadian_Aboriginal_Syllabics}>
U+1680…U+169F    <\p{InOgham}>
U+16A0…U+16FF    <\p{InRunic}>
U+1700…U+171F    <\p{InTagalog}>
U+1720…U+173F    <\p{InHanunoo}>
U+1740…U+175F    <\p{InBuhid}>
U+1760…U+177F    <\p{InTagbanwa}>
U+1780…U+17FF    <\p{InKhmer}>
U+1800…U+18AF    <\p{InMongolian}>
U+1900…U+194F    <\p{InLimbu}>
U+1950…U+197F    <\p{InTai_Le}>
U+19E0…U+19FF    <\p{InKhmer_Symbols}>
U+1D00…U+1D7F    <\p{InPhonetic_Extensions}>
U+1E00…U+1EFF    <\p{InLatin_Extended_Additional}>
U+1F00…U+1FFF    <\p{InGreek_Extended}>
U+2000…U+206F    <\p{InGeneral_Punctuation}>
U+2070…U+209F    <\p{InSuperscripts_and_Subscripts}>
U+20A0…U+20CF    <\p{InCurrency_Symbols}>
U+20D0…U+20FF    <\p{InCombining_Diacritical_Marks_for_Symbols}>
U+2100…U+214F    <\p{InLetterlike_Symbols}>
U+2150…U+218F    <\p{InNumber_Forms}>
U+2190…U+21FF    <\p{InArrows}>
U+2200…U+22FF    <\p{InMathematical_Operators}>
U+2300…U+23FF    <\p{InMiscellaneous_Technical}>
U+2400…U+243F    <\p{InControl_Pictures}>
U+2440…U+245F    <\p{InOptical_Character_Recognition}>
U+2460…U+24FF    <\p{InEnclosed_Alphanumerics}>
U+2500…U+257F    <\p{InBox_Drawing}>
U+2580…U+259F    <\p{InBlock_Elements}>
U+25A0…U+25FF    <\p{InGeometric_Shapes}>
U+2600…U+26FF    <\p{InMiscellaneous_Symbols}>
U+2700…U+27BF    <\p{InDingbats}>
U+27C0…U+27EF    <\p{InMiscellaneous_Mathematical_Symbols-A}>
U+27F0…U+27FF    <\p{InSupplemental_Arrows-A}>
U+2800…U+28FF    <\p{InBraille_Patterns}>
U+2900…U+297F    <\p{InSupplemental_Arrows-B}>
```

```
U+2980…U+29FF    <\p{InMiscellaneous_Mathematical_Symbols-B}>
U+2A00…U+2AFF    <\p{InSupplemental_Mathematical_Operators}>
U+2B00…U+2BFF    <\p{InMiscellaneous_Symbols_and_Arrows}>
U+2E80…U+2EFF    <\p{InCJK_Radicals_Supplement}>
U+2F00…U+2FDF    <\p{InKangxi_Radicals}>
U+2FF0…U+2FFF    <\p{InIdeographic_Description_Characters}>
U+3000…U+303F    <\p{InCJK_Symbols_and_Punctuation}>
U+3040…U+309F    <\p{InHiragana}>
U+30A0…U+30FF    <\p{InKatakana}>
U+3100…U+312F    <\p{InBopomofo}>
U+3130…U+318F    <\p{InHangul_Compatibility_Jamo}>
U+3190…U+319F    <\p{InKanbun}>
U+31A0…U+31BF    <\p{InBopomofo_Extended}>
U+31F0…U+31FF    <\p{InKatakana_Phonetic_Extensions}>
U+3200…U+32FF    <\p{InEnclosed_CJK_Letters_and_Months}>
U+3300…U+33FF    <\p{InCJK_Compatibility}>
U+3400…U+4DBF    <\p{InCJK_Unified_Ideographs_Extension_A}>
U+4DC0…U+4DFF    <\p{InYijing_Hexagram_Symbols}>
U+4E00…U+9FFF    <\p{InCJK_Unified_Ideographs}>
U+A000…U+A48F    <\p{InYi_Syllables}>
U+A490…U+A4CF    <\p{InYi_Radicals}>
U+AC00…U+D7AF    <\p{InHangul_Syllables}>
U+D800…U+DB7F    <\p{InHigh_Surrogates}>
U+DB80…U+DBFF    <\p{InHigh_Private_Use_Surrogates}>
U+DC00…U+DFFF    <\p{InLow_Surrogates}>
U+E000…U+F8FF    <\p{InPrivate_Use_Area}>
U+F900…U+FAFF    <\p{InCJK_Compatibility_Ideographs}>
U+FB00…U+FB4F    <\p{InAlphabetic_Presentation_Forms}>
U+FB50…U+FDFF    <\p{InArabic_Presentation_Forms-A}>
U+FE00…U+FE0F    <\p{InVariation_Selectors}>
U+FE20…U+FE2F    <\p{InCombining_Half_Marks}>
U+FE30…U+FE4F    <\p{InCJK_Compatibility_Forms}>
U+FE50…U+FE6F    <\p{InSmall_Form_Variants}>
U+FE70…U+FEFF    <\p{InArabic_Presentation_Forms-B}>
U+FF00…U+FFEF    <\p{InHalfwidth_and_Fullwidth_Forms}>
U+FFF0…U+FFFF    <\p{InSpecials}>
```

Блок Юникода – это единый и неразрывный диапазон кодовых пунктов. Несмотря на то, что многим блокам присвоены имена алфавитов и категорий Юникода, тем не менее, здесь нет 100% совпадения. Название блока лишь указывает на его основное назначение.

Блок Currency не включает в себя знаки доллара и йены. По исторически сложившимся причинам они находятся в блоках Basic_Latin и Latin-1_Supplement. Но при этом оба знака имеют свойство Currency Symbol. Поэтому для поиска любых знаков денежных единиц вместо комбинации ‹\p{InCurrency}› следует использовать ‹\p{Sc}›.

Большинство блоков включают в себя кодовые пункты с не присвоенными символами, которые покрываются свойством ‹\p{Cn}›. Никакое другое свойство Юникода и никакой алфавит не включают кодовые пункты с не присвоенными символами.

Комбинация ‹\p{InBlockName}› может использоваться в диалектах .NET и Perl. В диалекте Java надо использовать комбинацию ‹\p{IsBlockName}›.

Диалект Perl поддерживает вариант Is, но лучше использовать синтаксис In, чтобы избежать путаницы. Для работы с алфавитами в Perl поддерживаются комбинации ‹\p{Script}› и ‹\p{IsScript}›, но не ‹\p{InScript}›.

Алфавит Юникода

Каждый кодовый пункт, за исключением кодовых пунктов с не присвоенными символами, является частью точно одного алфавита Юникода. Кодовые пункты с не присвоенными символами не являются частью ни одного алфавита. Кодовые пункты со значениями до U+FFFF и с присвоенными символами принадлежат следующим алфавитам:

```
<\p{Common}>
<\p{Arabic}>
<\p{Armenian}>
<\p{Bengali}>
<\p{Bopomofo}>
<\p{Braille}>
<\p{Buhid}>
<\p{CanadianAboriginal}>
<\p{Cherokee}>
<\p{Cyrillic}>
<\p{Devanagari}>
<\p{Ethiopic}>
<\p{Georgian}>
<\p{Greek}>
<\p{Gujarati}>
<\p{Gurmukhi}>
<\p{Han}>
<\p{Hangul}>
<\p{Hanunoo}>
<\p{Hebrew}>
<\p{Hiragana}>
<\p{Katakana}>
<\p{Khmer}>
<\p{Lao}>
<\p{Latin}>
<\p{Limbu}>
<\p{Malayalam}>
<\p{Mongolian}>
<\p{Myanmar}>
<\p{Ogham}>
<\p{Oriya}>
<\p{Runic}>
<\p{Sinhala}>
<\p{Syriac}>
<\p{Tagalog}>
<\p{Tagbanwa}>
<\p{TaiLe}>
<\p{Tamil}>
<\p{Telugu}>
<\p{Thaana}>
<\p{Thai}>
<\p{Tibetan}>
```

```
<\p{Inherited}>                    <\p{Yi}>
<\p{Kannada}>
```

Алфавит – это группа кодовых пунктов, используемых в различных системах письменности. Одни алфавиты, такие как Thai, применяются только в одном языке человеческого общения. Другие, такие как Latin, используются в нескольких языках. В некоторых языках применяется несколько алфавитов. Например, в Юникоде нет алфавита Japanese, вместо этого предлагаются алфавиты Hiragana, Katakana, Han и Latin, символы из которых обычно используются при составлении документов на японском языке.

Самым первым в списке, в нарушение алфавитного порядка следования, указан алфавит Common. Этот алфавит содержит все символы, общие для широкого диапазона алфавитов, такие как знаки препинания, пробел и различные вспомогательные символы.

Графема Юникода

Различия между кодовыми пунктами и символами становятся заметны, когда в игру вступают *комбинированные знаки*. Кодовому пункту U+0061 соответствует «строчная буква а латинского алфавита», а кодовому пункту U+00E0 – «строчная буква а латинского алфавита с диакритическим знаком». В комбинации они представляют то, что большинство людей назвали бы символом.

Кодовый пункт U+0300 – это диакритический знак. Он может использоваться только после буквы. Строка, состоящая из кодовых пунктов U+0061 U+0300, будет отображаться, как символ à, так же, как и кодовый пункт U+00E0. Комбинационный знак U+0300 отображается над символом U+0061.

Причина появления двух разных способов отображения символа с диакритическим знаком состоит в том, что уже давно существует множество кодировок символов, в которых символы с диакритическими знаками присутствуют в виде единственных символов. Разработчики Юникода сочли, что будет полезно сохранить точное отображение устаревших наборов символов в дополнение к способу, когда диакритические знаки отделяются от базовых символов, который позволяет создавать произвольные комбинации, не поддерживаемые устаревшими кодировками.

Для вас, как для пользователя регулярных выражений, это особенно важно, так как все диалекты регулярных выражений, рассматриваемые в этой книге, оперируют кодовыми пунктами, а не графическими символами. Когда мы говорим, что ‹.› соответствует единственному символу, в действительности подразумевается соответствие единственному кодовому пункту. Если испытуемый текст будет состоять из двух кодовых пунктов U+0061 U+0300, который в языках программирования, таких как Java, может быть представлен строковым литералом “\u0061\u0300”, точке будет соответствовать только кодовый пункт

U+0061, или символ a без диакритического знака U+0300. А обоим кодовым пунктам будет соответствовать регулярное выражение ‹..›.

Диалекты Perl и PCRE предлагают специальный метасимвол ‹\X›, которому соответствует любая одиночная графема Юникода. По сути, это Юникод-версия метасимвола ‹.›. Ему соответствует любой кодовый пункт, который не является комбинационным знаком, вместе с любым комбинационным знаком, следующим за ним. Тот же эффект дает комбинация ‹\P{M}\p{M}*›, использующая синтаксис свойств Юникода. Метасимвол ‹\X› обнаружит два совпадения в строке àà независимо от того, как она закодирована. Если эта строка закодирована как "\u00E0\u0061\u0300", тогда первое совпадение будет обнаружено с "\u00E0", а второе – с "\u0061\u0300".

Варианты

Инвертированный вариант

Метасимвол ‹\P› является обратным для метасимвола ‹\p›. Например, комбинации ‹\P{Sc}› соответствует любой символ, не имеющий свойства Юникода «Currency Symbol». Метасимвол ‹\P› поддерживается всеми диалектами, поддерживающими метасимвол ‹\p›, и применим для всех поддерживаемых свойств, блоков и алфавитов.

Символьные классы

Все диалекты, поддерживающие метасимволы ‹\u›, ‹\x›, ‹\p› и ‹\P›, допускают их использование в символьных классах. В этом случае в класс добавляется символ, представленный кодовым пунктом, или символы, принадлежащие категории, блоку или алфавиту. Например, следующему регулярному выражению будет соответствовать символ, являющийся либо открывающей кавычкой, либо закрывающей кавычкой, либо символом знака торговой марки (U+2122):

```
[\p{Pi}\p{Pf}\x{2122}]
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Перечисление всех символов

Если диалект регулярных выражений не поддерживает категории Юникода, блоки или алфавиты, можно просто перечислить символы, составляющие требуемую категорию, блок или алфавит в виде символьного класса. Для блоков это выполняется очень просто: каждый блок – это просто непрерывный диапазон, заключенный между двумя кодовыми пунктами. Например, блок Greek Extended включает символы от U+1F00 по U+1FFF:

```
[\u1F00-\u1FFF]
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, Python

```
[\x{1F00}-\x{1FFF}]
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

Для большинства категорий и многих алфавитов эквивалентные символьные классы будут представлять длинные списки из отдельных кодовых пунктов и коротких диапазонов. Символы, входящие в любую из категорий и во многие алфавиты, разбросаны по всей таблице Юникода. Ниже приводится аналог алфавита Greek:

```
[\u0370-\u0373\u0375\u0376-\u0377\u037A\u037B-\u037D\u0384\u0386↵
\u0388-\u038A\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03F5\u03F6↵
\u03F7-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61\u1D66-\u1D6A\u1DBF\u1F00-\u1F15↵
\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D\u1F50-\u1F57\u1F59\u1F5B\u1F5D↵
\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FBC\u1FBD\u1FBE\u1FBF-\u1FC1↵
\u1FC2-\u1FC4\u1FC6-\u1FCC\u1FCD-\u1FCF\u1FD0-\u1FD3\u1FD6-\u1FDB↵
\u1FDD-\u1FDF\u1FE0-\u1FEC\u1FED-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFC↵
\u1FFD-\u1FFE\u2126]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python

Мы создали это регулярное выражение, скопировав список кодовых пунктов алфавита Greek из документа *http://www.unicode.org/Public/UNIDATA/Scripts.txt* и выполнив поиск с заменой с помощью трех следующих регулярных выражений:

1. С помощью операции поиска по регулярному выражению ‹;.*› с заменой найденных совпадений пустой строкой были удалены комментарии. Если это выражение удалит все, отмените изменения и отключите параметр «точке соответствуют границы строк».
2. С помощью операции поиска по регулярному выражению ‹^›, при включенном параметре «символам ^ и $ соответствуют границы строк» с заменой найденных совпадений на строку “\u” в начало каждого кодового пункта был добавлен префикс \u. А замена ‹\.\.› строкой “-\u” обеспечила корректное оформление диапазонов.
3. Наконец, операция замены фрагментов, соответствующих выражению ‹\s+›, пустой строкой удалила символы перевода строки. Добавление квадратных скобок вокруг символьного класса завершило создание регулярного выражения. Вероятно, вам придется вручную добавить префикс \u в начало символьного класса и/или удалить лишнюю комбинацию \u в конце, в зависимости от того, включили ли вы начальную и конечную пустые строки при копировании списка из файла Scripts.txt.

На первый взгляд может показаться, что был проделан большой объем работы, но в действительности на все это у Яна ушло меньше минуты. Чтобы написать это пояснение, потребовалось гораздо больше времени.

Сделать то же самое для случая использования синтаксиса \x{} ничуть не сложнее:

1. С помощью операции поиска по регулярному выражению ‹;.*› с заменой найденных совпадений пустой строкой были удалены комментарии. Если это выражение удалит все, отмените изменения и отключите параметр «точке соответствуют границы строк».
2. С помощью операции поиска по регулярному выражению ‹^› при включенном параметре «символам ^ и $ соответствуют границы строк» с заменой найденных совпадений на строку "\x{", в начало каждого кодового пункта был добавлен префикс \x{. А замена ‹\.\.› строкой "}-\x{" обеспечила корректное оформление диапазонов.
3. Наконец, операция замены фрагментов, соответствующих выражению ‹\s+›, строкой "}" добавила закрывающие фигурные скобки и удалила символы перевода строки. Добавление квадратных скобок вокруг символьного класса завершило создание регулярного выражения. Вероятно, вам придется вручную добавить префикс \x{ в начало символьного класса и/или удалить лишнюю комбинацию \x{ в конце в зависимости от того, включили ли вы начальную и конечную пустые строки при копировании списка из файла Scripts.txt.

Результат приводится ниже:

```
[\x{0370}-\x{0373}\x{0375}\x{0376}-\x{0377}\x{037A}\x{037B}-\x{037D}↵
\x{0384}\x{0386}\x{0388}-\x{038A}\x{038C}\x{038E}-\x{03A1}↵
\x{03A3}-\x{03E1}\x{03F0}-\x{03F5}\x{03F6}\x{03F7}-\x{03FF}↵
\x{1D26}-\x{1D2A}\x{1D5D}-\x{1D61}\x{1D66}-\x{1D6A}\x{1DBF}↵
\x{1F00}-\x{1F15}\x{1F18}-\x{1F1D}\x{1F20}-\x{1F45}\x{1F48}-\x{1F4D}↵
\x{1F50}-\x{1F57}\x{1F59}\x{1F5B}\x{1F5D}\x{1F5F}-\x{1F7D}↵
\x{1F80}-\x{1FB4}\x{1FB6}-\x{1FBC}\x{1FBD}\x{1FBE}\x{1FBF}-\x{1FC1}↵
\x{1FC2}-\x{1FC4}\x{1FC6}-\x{1FCC}\x{1FCD}-\x{1FCF}\x{1FD0}-\x{1FD3}↵
\x{1FD6}-\x{1FDB}\x{1FDD}-\x{1FDF}\x{1FE0}-\x{1FEC}\x{1FED}-\x{1FEF}↵
\x{1FF2}-\x{1FF4}\x{1FF6}-\x{1FFC}\x{1FFD}-\x{1FFE}\x{2126}↵
\x{10140}-\x{10174}\x{10175}-\x{10178}\x{10179}-\x{10189}↵
\x{1018A}\x{1D200}-\x{1D241}\x{1D242}-\x{1D244}\x{1D245}]
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

См. также

http://www.unicode.org – официальный веб-сайт консорциума Unicode Consortium, где можно загрузить любые официальные документы, касающиеся стандарта Юникод, таблицы символов и прочее.

Юникод – это обширная тема, которой посвящены целые книги. Одна из таких книг называется «Unicode Explained», автор Джукка Корпела (Jukka K. Korpella) (O'Reilly).

Мы не можем в единственном разделе описать все, что вам потребуется знать о кодовых пунктах Юникода, свойствах, блоках и алфавитах. Мы даже не можем объяснить, зачем вам это нужно – вам это просто необходимо. Комфортная простота расширенной таблицы ASCII – это пустынное место в современном глобализованном мире.

2.8. Сопоставление с одной из нескольких альтернатив

Задача

Создать регулярное выражение, которому при многократном применении к тексту `Mary, Jane, and Sue went to Mary's house` будут соответствовать Mary, Jane, Sue и снова Mary. Последующие попытки применения должны терпеть неудачу.

Решение

```
Mary|Jane|Sue
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Символ *вертикальной черты* служит в регулярных выражениях для разделения нескольких *альтернатив*. При каждом последующем применении выражению `‹Mary|Jane|Sue›` будут соответствовать слова Mary, или Jane, или Sue. Всякий раз ему будет соответствовать только одно имя, но каждый раз новое.

Все диалекты регулярных выражений, рассматриваемые в этой книге, используют механизмы, управляемые регулярными выражениями. *Механизм* – это обычное программное обеспечение, реализующее поддержку регулярных выражений. Термин *управляемый регулярными выражениями*[1] означает, что прежде чем будет выполнен переход к следую-

[1] Другая разновидность механизмов – это механизмы, *управляемые текстом*. Главное отличие состоит в том, что механизм, управляемый текстом, посещает каждый символ в испытуемом тексте только один раз, тогда как механизм, управляемый регулярным выражением, может посещать каждый символ несколько раз. Механизмы, управляемые текстом, обладают более высокой производительностью, но поддерживают выражения, которые являются регулярными только в математическом смысле, как описывалось в начале главы 1. Замысловатые регулярные выражения в стиле языка Perl, которые делают эту книгу такой захватывающей, могут быть реализованы только при использовании механизма, управляемого регулярными выражениями.

щему символу, для каждого текущего символа в испытуемом тексте будут опробованы все возможные перестановки регулярного выражения.

Когда выражение ‹Mary|Jane|Sue› впервые применяется к тексту Mary, Jane, and Sue went to Mary's house, тут же обнаруживается совпадение Mary в начале строки.

Когда то же самое выражение применяется к остатку строки, то есть после щелчка на кнопке Find Next (Найти далее) в текстовом редакторе, механизм регулярных выражений попытается сопоставить ‹Mary› с первой запятой в строке. Эта попытка завершится неудачей. Затем будет выполнена попытка найти совпадение с ‹Jane› в той же позиции, которая также окончится неудачей. Попытка найти совпадение с ‹Sue› с запятой также провалится. Только после этого механизм регулярных выражений передвинется на один символ вперед. Сопоставление начнется с первого пробела, в результате поиск всех трех альтернатив также не увенчается успехом.

Когда начнется сопоставление с символа J, попытка найти соответствие первой альтернативе ‹Mary› потерпит неудачу. Затем, опять же с символа J, начнется сопоставление альтернативы ‹Jane›, В результате будет найдено соответствие Jane и механизм регулярных выражений объявит о победе.

Обратите внимание, что соответствие Jane было найдено несмотря на то, что в испытуемом тексте имеется еще одно вхождение имени Mary, а альтернатива ‹Mary› находится в регулярном выражении перед альтернативой ‹Jane›. По крайней мере в данном случае порядок следования альтернатив в регулярном выражении не имеет значения. Регулярное выражение находит *самое первое слева* соответствие. Оно просматривает текст слева направо, на каждом шаге пытается применить все альтернативы и останавливает попытки в первой же позиции в испытуемом тексте, где было обнаружено соответствие любой из альтернатив.

Если выполнить еще одну попытку поиска в оставшейся части строки, будет найдено соответствие Sue. Четвертая попытка поиска обнаружит еще одно соответствие Mary. Если попробовать выполнить пятую попытку поиска, она завершится неудачей, потому что ни одна из трех альтернатив не обнаружит совпадение в оставшейся части строки , house.

Порядок следования альтернатив в регулярном выражении имеет значение, только когда две из них могут обнаружить соответствие в одной и той же позиции в строке. Например, регулярное выражение ‹Jane|Janet› имеет две альтернативы, которые совпадают с одной и той же позицией в тексте Her name is Janet. В регулярном выражении отсутствуют метасимволы границ слова. Поэтому тот факт, что ‹Jane› совпадает со словом Janet в тексте Her name is Janet только частично, не имеет никакого значения.

Выражению ‹Jane|Janet› соответствует последовательность Jane в тексте Her name is Janet, потому что механизм, управляемый регулярным

выражением, отличается *нетерпеливостью*. Кроме того, что испытуемый текст просматривается слева направо, в поисках первого встретившегося совпадения, альтернативы в регулярном выражении точно так же просматриваются слева направо. Механизм прекращает поиск, как только обнаруживает первую совпавшую альтернативу.

Когда выражение ‹Jane|Janet› достигает символа J в тексте Her name is Janet, обнаруживается соответствие с первой альтернативой, ‹Jane›. Вторая альтернатива даже не испытывается. Если предложить механизму попытаться отыскать второе соответствие, в испытуемом тексте для дальнейшего просмотра останется только символ t. В результате не будет найдено соответствие ни с одной из альтернатив.

Существует два способа воспрепятствовать альтернативе ‹Jane› похитить соответствие у альтернативы ‹Janet›. Первый способ состоит в том, чтобы поместить более длинную альтернативу на первое место: ‹Janet|Jane›. Более надежное решение заключается в том, чтобы явно определить выполняемые действия: в данном случае выполняется поиск имен, а имена – это целые слова. Регулярные выражения не распознают слова, зато они распознают границы слов.

Так, оба выражения ‹\bJane\b|\bJanet\b› и ‹\bJanet\b|\bJane\b› совпадут со словом Janet в тексте Her name is Janet. Благодаря привязке к границам слова совпасть смогла только одна альтернатива. Порядок следования альтернатив опять оказался несущественным.

В рецепте 2.12 описывается более надежное решение: ‹\bJanet?\b›.

См. также

Рецепт 2.9.

2.9. Группы и сохранение части совпадения

Задача

Улучшить регулярное выражение, соответствующее именам Mary, Jane и Sue, так чтобы найденные совпадения были целыми словами. Использовать группировку, чтобы добиться этого эффекта за счет использования одной пары метасимволов границы слова во всем регулярном выражении вместо одной пары для каждой альтернативы.

Создать регулярное выражение, которому соответствовали бы любые даты в формате yyyy-mm-dd, и сохраняющее по отдельности год, месяц и день. Цель состоит в том, чтобы облегчить работу с отдельными значениями в программном коде, обрабатывающем совпадение. Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно исключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте.

Решение

```
\b(Mary|Jane|Sue)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(\d\d\d\d)-(\d\d)-(\d\d)\b
```

Параметры: None
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Оператор выбора, о котором говорилось в предыдущем разделе, имеет самый низший приоритет среди всех операторов регулярных выражений. Например, если записать такое выражение: ‹\bMary|Jane|Sue\b›, в нем получится три альтернативы: ‹\bMary›, ‹Jane› и ‹Sue\b›. Данное регулярное выражение будет обнаруживать совпадение Jane в тексте Her name is Janet.

Чтобы в регулярном выражении что-то исключить из альтернатив, следует *сгруппировать* альтернативы. Группировка выполняется с помощью круглых скобок. Они имеют наивысший приоритет среди всех операторов регулярных выражений, как и в большинстве языков программирования. В выражении ‹\b(Mary|Jane|Sue)\b› имеется три альтернативы, ‹Mary›, ‹Jane› и ‹Sue›, заключенные в метасимволы границ слова. Это регулярное выражение не находит соответствий в тексте Her name is Janet.

Когда механизм регулярных выражений достигнет символа J в слове Janet внутри испытуемого текста, первый метасимвол границы слова обнаружит совпадение. После этого механизм перейдет к группе. Попытка сопоставления с первой альтернативой в группе, ‹Mary›, потерпит неудачу. Сопоставление со второй альтернативой, ‹Jane›, увенчается успехом. Механизм покинет группу. Теперь непроверенным в выражении остается только метасимвол ‹\b›. Сопоставление границы слова с позицией между символами e и t в конце испытуемого текста потерпит неудачу. Поэтому и вся попытка найти соответствие начиная с позиции символа J также потерпит неудачу.

Пара круглых скобок – это не просто группа, а *сохраняющая группа*. Для случая регулярного выражения Mary-Jane-Sue сохранение не дает никаких преимуществ, потому что оно охватывает все регулярное выражение. Возможность сохранения приобретает особое значение, когда оно охватывает только часть регулярного выражения, например: ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b›.

Регулярное выражение совпадает с датами в формате yyyy-mm-dd. В точности то же самое делает регулярное выражение ‹\b\d\d\d\d-\d\d-\d\d\b›. Поскольку в этом регулярном выражении отсутствуют операто-

ры выбора или повторения, функция группировки оказывается невостребованной, но вот функция сохранения оказывается полезной.

В регулярном выражении ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b› имеется три сохраняющих группы. Группы нумеруются порядковыми номерами открывающих круглых скобок, присваиваемыми слева направо. Подвыражение ‹(\d\d\d\d)› – это группа с номером 1. Подвыражение ‹(\d\d)› – группа с номером 2. Второе подвыражение ‹(\d\d)› – группа с номером 3.

В процессе поиска совпадений, когда механизм регулярных выражений покидает группу, достигнув закрывающей круглой скобки, он сохраняет текст, совпавший с сохраняющей группой. Когда регулярное выражение совпадает с текстом 2008-05-24, фрагмент 2008 сохраняется в первом сохранении, фрагмент 05 – во втором сохранении и фрагмент 24 – в третьем.

Существует три способа использования сохраненного текста. В рецепте 2.10 описывается, как повторно проверить соответствие сохраненного текста тому же самому регулярному выражению. В рецепте 2.21 показано, как вставлять сохраненный текст в замещающий текст при выполнении операции поиска с заменой. В рецепте 3.9, в следующей главе, описывается, как совпадения с частями регулярных выражений можно использовать в приложениях.

Варианты

Несохраняющие группы

В регулярном выражении ‹\b(Mary|Jane|Sue)\b› круглые скобки потребовались исключительно для нужд группировки. Поэтому вместо сохраняющей группировки можно было бы использовать несохраняющую группировку:

```
\b(?:Mary|Jane|Sue)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Несохраняющая группа начинается с трех символов ‹(?:›. Закрывающая круглая скобка ‹)› закрывает ее. Несохраняющая группировка обеспечивает ту же самую возможность группировки, но без сохранения совпадений.

При подсчете открывающих круглых скобок для выяснения порядковых номеров сохраняющих групп скобки, открывающие несохраняющие группы, не учитываются. Это главное преимущество несохраняющих групп: они могут добавляться в существующие регулярные выражения, не оказывая влияния на нумерованные ссылки, указывающие на сохраняющие группы.

Еще одно преимущество несохраняющей группировки заключается в скорости работы. Если в выражении не предполагается использовать обратные ссылки на определенные группы (рецепт 2.10), вставлять сохраненные в них фрагменты в замещающий текст (рецепт 2.21) или извлекать сохраненные фрагменты в программном коде (рецепт 3.9), сохраняющая группировка просто будет давать ненужную нагрузку, которую можно ликвидировать, задействовав несохраняющую группировку. На практике выигрыш в скорости работы регулярного выражения от такой замены трудно заметить, если регулярное выражение не используется в глубоком цикле или не обрабатывает большой массив данных.

Группы с модификаторами режима

В рецепте 2.1 в описании варианта «Поиск без учета регистра символов» говорилось, что диалекты .NET, Java, PCRE, Perl и Ruby поддерживают локальные модификаторы режима, которые работают как простые переключатели: ‹sensitive(?i)caseless(?-i)sensitive›. Хотя синтаксис модификаторов включает круглые скобки, например ‹(?i)›, они не имеют никакого отношения к группировке.

Вместо применения переключающих модификаторов можно использовать их внутри несохраняющих групп:

```
\b(?i:Mary|Jane|Sue)\b
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
sensitive(?i:caseless)sensitive
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

При добавлении модификатора режима в несохраняющую группу режим, определяемый этим модификатором, распространяется только на часть регулярного выражения, заключенную в группу. По достижении закрывающей круглой скобки восстанавливаются предыдущие параметры режима. Так как по умолчанию поиск соответствий выполняется с учетом регистра символов, только часть выражения внутри

```
(?i:···)
```

будет нечувствительна к регистру символов.

Допускается объединять несколько модификаторов режима, например: ‹(?ism:group)›. Для отключения режима, определяемого модификатором, следует использовать символ дефиса: выражение ‹(?-ism:group)› отключает все три режима. Выражение ‹(?i-sm:group)› включает режим нечувствительности к регистру символов (i) и отключает режимы «точке соответствуют границы строк» (s) и «символам ^ и $ соответствуют границы строк» (m). Эти режимы подробно описаны в рецептах 2.4 и 2.5.

См. также

Рецепты 2.10, 2.11, 2.21 и 3.9.

2.10. Повторный поиск соответствия с ранее совпавшим текстом

Задача

Создать регулярное выражение, совпадающее с «магическими» датами в формате yyyy-mm-dd. Магической считается дата, когда последние две цифры года, месяц и день являются одним и тем же числом. Например, магической считается дата 2008-08-08. Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно исключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте. Требуется отыскать только магические даты.

Решение

```
\b\d\d(\d\d)-\1-\1\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Чтобы в регулярном выражении повторно проверить совпадение текста, уже совпавшего ранее, для начала необходимо сохранить предыдущее совпадение. Сделать это можно с помощью сохраняющей группы, как было показано в рецепте 2.9. После этого с помощью *обратных ссылок* можно проверить совпадение с этим текстом в любом месте регулярного выражения. Ссылки на первые девять сохраняющих групп оформляются символом обратного слэша, за которым следует единственная цифра от одного до девяти. Для групп от 10 до 99 используются формы записи от ‹\10› до ‹\99›.

Не следует использовать форму записи ‹\01›. Она будет интерпретироваться как экранированная форма записи восьмеричного числа или как ошибка. Мы вообще не будем использовать экранированные формы записи восьмеричных чисел в этой книге, потому что экранированная форма записи шестнадцатеричных чисел ‹\xFF› более наглядна.

Когда регулярное выражение ‹\b\d\d(\d\d)-\1-\1\b› встретит магическую дату, например 2008-08-08, первое подвыражение ‹\d\d› совпадет с фрагментом 20. Затем механизм регулярных выражений войдет в сохраняющую группу, запомнив текущую позицию в испытуемом тексте.

Подвыражение ‹\d\d› внутри сохраняющей группы совпадет с фрагментом 08 и механизм обнаружит круглую скобку, закрывающую группу. В этой позиции совпавший фрагмент 08 будет сохранен в группе с номером 1. Следующий элемент регулярного выражения – дефис, который совпадает с самим собой. Далее следует обратная ссылка. Механизм регулярных выражений проверит содержимое первой сохраняющей группы: 08. Затем попробует сопоставить этот текст буквально. Если регулярное выражение нечувствительно к регистру символов, сохраненный текст сопоставляется таким способом. В данном случае сопоставление с обратной ссылкой увенчается успехом. Следующий дефис и обратная ссылка также совпадут. Наконец, метасимвол границы слова совпадет с концом испытуемого текста и в качестве общего совпадения будет найдена дата 2008-08-08. Сохраняющая группа по-прежнему будет хранить фрагмент 08.

Если сохраняющая группа повторяется в результате применения квантификатора (рецепт 2.12) или возврата (рецепт 2.13), каждый раз новое совпадение с сохраняющей группой затирает предыдущее. Обратная ссылка на группу совпадает только с текстом, сохраненным этой группой.

Если то же самое выражение встретит текст 2008-05-24 2007-07-07, первый раз сохраняющая группа выполнит сохранение, когда подвыражение ‹\b\d\d(\d\d)› совпадет с фрагментом 2008, и сохранит фрагмент 08 для первой (и единственной) сохраняющей группы. Затем дефис совпадет сам с собой. Затем обратная ссылка попытается сопоставить сохраненный фрагмент ‹08› с фрагментом текста 05 и потерпит неудачу.

Поскольку в выражении отсутствуют альтернативы, механизм закончит попытку сопоставления. На этом этапе будут очищены все сохраняющие группы. Когда механизм предпримет вторую попытку сопоставления, начиная с первого символа 0 в испытуемом тексте, в группе ‹\1› вообще не будет ничего.

В следующий раз сохраняющая группа выполнит сохранение, когда подвыражение ‹\b\d\d(\d\d)› совпадет с фрагментом 2007, и сохранит фрагмент 07. Затем дефис совпадет сам с собой. Потом обратная ссылка попытается сопоставить ‹07›. На этот раз попытка увенчается успехом, как и последующие попытки сопоставления дефиса, обратной ссылки и границы слова. В результате будет найдено соответствие 2007-07-07.

Так как механизм регулярных выражений производит обработку слева направо, сохраняющие круглые скобки должны находиться перед обратной ссылкой. Регулярные выражения ‹\b\d\d\1-(\d\d)-\1› и ‹\b\d\d\1-\1-(\d\d)\b› никогда не найдут соответствие. Поскольку обратная ссылка располагается перед сохраняющей группой, в сохраняющей группе еще ничего сохранено не было. Если вы не используете JavaScript, сопоставление обратной ссылки всегда будет приводить к неудаче, если она указывает на группу, которая еще не участвовала в сопоставлении.

Группа, не участвовавшая в сопоставлении, это не то же самое, что группа, сохранившая совпадение нулевой длины. Обращение к обратной ссылке на группу, сохранение в которой имеет нулевую длину, всегда оканчивается успехом. Когда выражение ‹(^)\1› сопоставляется с началом строки, первая сохраняющая группа сохраняет совпадение с метасимволом ‹^›, имеющее нулевую длину; в результате попытка сопоставления с ‹\1› завершается успехом. На практике такое может случаться, когда содержимое сохраняющей группы является необязательным.

JavaScript – это единственный известный нам диалект, который поступает вопреки десятилетним традициям обратных ссылок в регулярных выражениях. В диалекте JavaScript или, по крайней мере, в реализациях, следующих стандарту JavaScript, обратная ссылка на сохраняющую группу, не участвовавшую в сопоставлении, всегда оканчивается успехом, так же как и обратная ссылка на группу, сохранение которой имеет нулевую длину. По этой причине выражение ‹\b\d\d\1-\1-(\d\d)\b› в JavaScript может совпасть с текстом 12--34.

См. также

Рецепты 2.9, 2.11, 2.21 и 3.9.

2.11. Сохранение и именованные части совпадения

Задача

Создать регулярное выражение, которому соответствовали бы любые даты в формате yyyy-mm-dd и сохраняющее по отдельности год, месяц и день. Цель состоит в том, чтобы облегчить работу с отдельными значениями в программном коде, обрабатывающем совпадение. Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Содействуя достижению этой цели, необходимо присвоить сохраняемым фрагментам текста описательные имена «year», «month» и «day».

Создать другое регулярное выражение, совпадающее с «магическими» датами в формате yyyy-mm-dd. Магической считается дата, когда последние две цифры года, месяц и день являются одним и тем же числом. Например, магической считается дата 2008-08-08. Необходимо сохранить магическое число (08 в примере) и пометить его именем «magic».

Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно ис-

ключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте.

Решение

Именованное сохранение

```
\b(?<year>\d\d\d\d)-(?<month>\d\d)-(?<day>\d\d)\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'year'\d\d\d\d)-(?'month'\d\d)-(?'day'\d\d)\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<year>\d\d\d\d)-(?P<month>\d\d)-(?P<day>\d\d)\b
```

Параметры: нет
Диалекты: PCRE 4 and later, Perl 5.10, Python

Именованные обратные ссылки

```
\b\d\d(?<magic>\d\d)-\k<magic>-\k<magic>\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?'magic'\d\d)-\k'magic'-\k'magic'\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?P<magic>\d\d)-(?P=magic)-(?P=magic)\b
```

Параметры: нет
Диалекты: PCRE 4 and later, Perl 5.10, Python

Обсуждение

В рецептах 2.9 и 2.10 были продемонстрированы *сохраняющие группы* и *обратные ссылки*. Если говорить точнее, в этих рецептах использовались *нумерованные* сохраняющие группы и нумерованные обратные ссылки. Каждой группе автоматически присваивается свой номер, который затем используется в обратных ссылках.

В дополнение к нумерованным группам современные диалекты регулярных выражений поддерживают *именованные* сохраняющие группы. Единственное отличие между именованными и нумерованными группами заключается в возможности присваивать описательные имена вместо того, чтобы иметь дело с автоматически присваиваемыми номерами. Именованные группы делают регулярные выражения более удобочитаемыми и более простыми в сопровождении. Вставка новой

сохраняющей группы в существующее регулярное выражение может изменить номера всех сохраняющих групп. Имена же, присваиваемые вами, остаются неизменными.

Python стал первым диалектом регулярных выражений, в котором появилась поддержка именованных сохранений. Синтаксис определения именованной сохраняющей группы выглядит так: ‹(?P<name>regex)›. Имя должно состоять из символов слова, соответствующих метасимволу ‹\w›. Комбинация ‹(?P<name>› – это открывающая скобка группы, а ‹)› – закрывающая.

Разработчики класса Regex в платформе .NET выработали собственный синтаксис определения именованных сохранений, использовав два взаимозаменяемых варианта. Синтаксис ‹(?<name>regex)› имитирует синтаксис диалекта Python, в котором отсутствует символ P. Имя должно состоять из символов слова, соответствующих метасимволу ‹\w›. Комбинация ‹(?<name>› – это открывающая скобка группы, а ‹)› – закрывающая.

Угловые скобки в имени сохранения здорово раздражают, когда необходимо записать выражение в документе XML, как это было при написании данной книги в формате DocBook XML. По этой причине в .NET появился альтернативный синтаксис определения именованных сохранений: ‹(?'name'regex)›. Угловые скобки в нем были заменены апострофами. Выбирайте, какой синтаксис будет для вас удобнее. Их функциональность совершенно идентична.

Вероятно благодаря более высокой популярности .NET по сравнению с Python разработчики других библиотек поддержки регулярных выражений предпочитают воспроизводить синтаксис .NET. Этот синтаксис поддерживается в Perl 5.10 и механизмом Onigurama в Ruby 1.9.

В библиотеке PCRE синтаксис диалекта Python воспроизведен уже давно, когда диалект Perl вообще не поддерживал именованные сохранения. Версия PCRE 7, добавляющая новые особенности в Perl 5.10, поддерживает как синтаксис диалекта .NET, так и синтаксис диалекта Python. Возможно, из-за успеха библиотеки PCRE в смысле сохранения обратной совместимости, Perl 5.10 также поддерживает синтаксис диалекта Python. В PCRE и Perl 5.10 функциональность синтаксиса .NET и Python именованных сохранений полностью идентична.

Выбирайте тот синтаксис, который будет наиболее удобным для вас. Если вы используете язык PHP и стремитесь сохранить совместимость со старыми версиями PHP, включающими старые версии библиотеки PCRE, используйте синтаксис диалекта Python. Если совместимость со старыми версиями не требуется, и при этом вам приходится работать с платформой .NET или Ruby, используйте синтаксис диалекта .NET, чтобы проще было переносить регулярные выражения между программами на разных языках. В случае сомнений при работе с PHP/PCRE используйте синтаксис диалекта Python. Те, кому придется пересобирать ваши программы с поддержкой более старых версий PCRE, будут

недовольны, если вдруг регулярные выражения в вашем программном коде окажутся неработоспособны. При копировании регулярного выражения в программу, где используется диалект .NET или Ruby, удалить несколько лишних символов `P` будет совсем несложно.

Документация к PCRE 7 и Perl 5.10 лишь вскользь упоминает синтаксис диалекта Python, но это не означает, что его не следует использовать. Более того, мы рекомендуем использовать его при работе с PCRE и PHP.

Именованные обратные ссылки

В паре с именованными сохранениями идут именованные обратные ссылки. Так же, как функциональность именованных сохраняющих групп идентична функциональности нумерованных сохраняющих групп, функциональность именованных обратных ссылок идентична функциональности нумерованных обратных ссылок. Зато они более удобочитаемые и проще в сопровождении.

В диалекте Python для создания обратной ссылки на группу `name` используется синтаксис `‹(?P=name)›`. Несмотря на то, что здесь используются круглые скобки, это обратная ссылка, а не группа. В этом определении нельзя ничего помещать между именем и закрывающей круглой скобкой. Обратная ссылка `‹(?P=name)›` — это особый элемент регулярного выражения, так же как и `‹\1›`.

В диалекте .NET используются синтаксические конструкции `‹\k<name>›` и `‹\k'name'›`. Оба варианта совершенно идентичны по своей функциональности и являются взаимозаменяемыми. Обратная ссылка, определенная с применением апострофов, может ссылаться на именованную группу, созданную с применением угловых скобок, и наоборот.

Мы настоятельно рекомендуем не смешивать именованные и нумерованные группы в одном регулярном выражении. Различные диалекты следуют разным правилам нумерации неименованных групп, находящихся между именованными группами. В Perl 5.10 и Ruby 1.9 используется синтаксис диалекта .NET, но они не следуют правилам нумерации, принятым в диалекте .NET для именованных сохраняющих групп или в случае смешивания нумерованных и именованных сохраняющих групп. Вместо того чтобы попытаться объяснить различия, я просто рекомендую не смешивать именованные и нумерованные группы. Избегайте путаницы и либо давайте имена всем неименованным группам, либо делайте их несохраняющими.

См. также

Рецепты 2.9, 2.10, 2.21 и 3.9.

2.12. Повторение части регулярного выражения определенное число раз

Задача

Создать регулярное выражение, которое совпадает со следующими разновидностями чисел:

- Гугол (десятичное число, состоящее из 100 цифр).
- 32-битное шестнадцатеричное число.
- 32-битное шестнадцатеричное число с необязательным суффиксом `h`.
- Вещественное число с необязательной целой частью, обязательной дробной частью и необязательной экспонентой. В каждой части допускается любое число цифр.

Решение

Гугол

```
\b\d{100}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричное число

```
\b[a-f0-9]{1,8}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричное число с необязательным суффиксом

```
\b[a-f0-9]{1,8}h?\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Вещественное число

```
\d*\.\d+(e\d+)?
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Фиксированное число повторений

Квантификатор `{n}`, где *n* – это положительное число, повторяет предшествующий ему элемент регулярного выражения *n* раз. Подвыра-

жение ‹\d{100}› в выражении ‹\b\d{100}\b› совпадает со строкой из 100 цифр. Тот же эффект можно получить, напечатав ‹\d› 100 раз.

Квантификатор ‹{1}› повторит предшествующий элемент один раз, как если бы квантификатора не было вообще. Выражение ‹ab{1}c› идентично выражению ‹abc›.

Квантификатор ‹{0}› повторит предшествующий элемент ноль раз, что фактически равносильно удалению его из регулярного выражения. Выражение ‹ab{0}c› идентично выражению ‹ac›.

Переменное число повторений

Для организации *повторений переменное число раз* используется квантификатор ‹{*n*,*m*}›, где *n* – это положительное число и *m* – число, большее *n*. Выражению ‹\b[a-f0-9]{1,8}\b› соответствует шестнадцатеричное число длиной от одной до восьми цифр. В случае переменного числа повторений становится существенным порядок следования альтернатив. Подробнее об этом рассказывается в рецепте 2.13.

Если числа *n* и *m* равны, выполняется фиксированное число повторений. Регулярное выражение ‹\b\d{100,100}\b› эквивалентно выражению ‹\b\d{100}\b›.

Бесконечное число повторений

Квантификатор ‹{*n*,}›, где *n* – это положительное число, позволяет выполнять *бесконечное число повторений*. По сути, бесконечное число повторений – это переменное число повторений, для которого не указан верхний предел.

Выражение ‹\d{1,}› совпадает с одной или более цифрами, то же самое делает и выражение ‹\d+›. Символ «плюс», следующий за элементом регулярного выражения, означает «один или более раз». В рецепте 2.13 описывается значение символа «плюс», следующего за квантификатором.

Выражение ‹\d{0,}› совпадает с любым числом цифр, в том числе и с нулевым, то же самое делает и выражение ‹\d*›. Символ «звездочка» всегда означает «ноль или более раз». Кроме того, что символ «звездочка» обеспечивает возможность бесконечного числа повторений, он еще делает предшествующий ему элемент необязательным.

Необязательные элементы

При использовании переменного числа повторений, когда число *n* равно нулю, элемент, предшествующий квантификатору, фактически делается необязательным. Выражение ‹h{0,1}› совпадает с ‹h› ноль или один раз. Если в испытуемом тексте отсутствует символ h, выражение ‹h{0,1}› вернет совпадение нулевой длины. Если использовать выражение ‹h{0,1}› как самостоятельное, оно будет обнаруживать совпадение

нулевой длины перед каждым символом в испытуемом тексте, отличном от h. Для каждого символа h будет обнаруживаться соответствие длиной в один символ (сам символ h).

Конструкция ‹h?› это то же самое, что и ‹h{0,1}›. Знак вопроса, следующий за допустимым элементом регулярного выражения, не являющимся квантификатором, означает «ноль или один раз». В следующем рецепте описывается значение знака вопроса, следующего за квантификатором.

Знак вопроса или любой другой квантификатор, следующий за открывающей круглой скобкой, вызывает синтаксическую ошибку. В Perl и диалектах, копирующих его, эта особенность используется для добавления «расширений Perl» в синтаксис регулярных выражений. В предыдущих рецептах демонстрируются такие особенности, как несохраняющие группы и именованные сохраняющие группы, в синтаксисе которых используется знак вопроса, следующий за открывающей круглой скобкой. Эти знаки вопроса не являются квантификаторами, они просто являются частью синтаксиса несохраняющих групп и именованных сохраняющих групп. В следующих рецептах будут показаны дополнительные способы группировки, где используется синтаксис ‹(?›.

Повторение групп

Если поместить квантификатор после круглой скобки, закрывающей группу, повторению подвергнется вся группа. Выражение ‹(?:abc){3}› идентично выражению ‹abcabcabc›.

Квантификаторы могут быть вложенными. Выражение ‹(e\d+)?› совпадет с символом e, за которым следует одна или более цифр, или обнаружит совпадение нулевой длины. В регулярном выражении, совпадающем с вещественным числом, этому подвыражению соответствует экспоненциальная часть числа.

Сохраняющие группы допускают возможность повторения. Как пояснялось в рецепте 2.9, совпадение с группой сохраняется всякий раз, когда механизм выходит за пределы группы, перезаписывая текст, совпавший с группой ранее. Выражению ‹(\d\d){1,3}› соответствует строка из двух, четырех или шести цифр. Механизм покидает группу трижды. Когда регулярное выражение совпадет со строкой 123456, сохраняющая группа будет хранить фрагмент 56, потому что фрагмент 56 был сохранен на последней итерации группы. Другие два совпадения с группой, 12 и 34, будут утрачены.

Выражение ‹(\d\d){3}› сохранит тот же текст, что и выражение ‹\d\d\d\d(\d\d)›. Если необходимо, чтобы сохраняющая группа сохраняла все две, четыре или шесть цифр, а не только две последние, то вместо того, чтобы повторять сохраняющую группу, следует поместить квантифика-

тор в эту группу: ‹((?:\d\d){1,3})›. Здесь была использована несохраняющая группа, чтобы иметь возможность вынести функцию группировки из сохраняющей группы. Точно так же можно было бы использовать сохраняющие группы: ‹((\d\d){1,3})›. Когда данное выражение совпадет со строкой 123456, в ‹\1› будет храниться фрагмент 123456, а в ‹\2› — 56.

Механизм регулярных выражений платформы .NET является единственным, который позволяет извлекать все итерации повторяющейся сохраняющей группы. Если напрямую обратиться к свойству Value группы, которое возвращает строку, будет получено значение 56, как и в любых других механизмах регулярных выражений. Обратные ссылки в регулярном выражении и в замещающем тексте также подставят фрагмент 56, но если воспользоваться свойством CaptureCollection группы, можно получить доступ к стеку со значениями 56, 34 и 12.

См. также

Рецепты 2.9, 2.13, 2.14.

2.13. Выбор минимального или максимального числа повторений

Задача

Создать регулярное выражение, совпадающее с парой тегов и языка разметки XHTML и текстом между ними. Текст между этими тегами может содержать другие теги XHTML.

Решение

```
<p>.*?</p>
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Все квантификаторы, рассматривавшиеся в рецепте 2.12, являются *максимальными*, то есть они стремятся выполнить максимально возможное число повторений, возвращая часть соответствия, только если это необходимо для обеспечения совпадения всего регулярного выражения.

Это может осложнить поиск пары тегов языка разметки XHTML (который является разновидностью языка XML и потому требующий, чтобы для каждого открывающего тега имелся соответствующий закрывающий тег). Рассмотрим следующий простой фрагмент текста на языке XHTML:

```
<p>
The very <em>first</em> task is to find the beginning of a paragraph.
</p>
<p>
Then you have to find the end of the paragraph
</p>
```

В этом фрагменте имеется два открывающих тега и два закрывающих . Необходимо обеспечить соответствие первого тега первому тегу , потому что они образуют единый параграф. Обратите внимание, что этот параграф содержит вложенный тег , поэтому регулярное выражение не может останавливаться, просто встретив первый символ <.

Взгляните на неправильное решение задачи в этом рецепте:

```
<p>.*</p>
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Единственное отличие этого неправильного решения заключается в отсутствии дополнительного знака вопроса после звездочки. Неправильное решение использует ту самую максимальную звездочку, что описывалась в рецепте 2.12.

После того как будет найдено соответствие с тегом в испытуемом тексте, механизм достигает конструкции ‹.*›. Точке соответствует любой символ, включая символы перевода строки. Звездочка повторяет попытки найти соответствие с точкой ноль или более раз. Звездочка является максимальным квантификатором, поэтому комбинация ‹.*› совпадает со всем, что попадается на ее пути до самого конца испытуемого текста. Повторюсь еще раз: комбинация ‹.*› «съест» весь файл XHTML, начиная от первого параграфа.

Когда ‹.*› насытится полностью, механизм попытается найти соответствие ‹<› в конце испытуемого текста. Эта попытка окончится неудачей. Но это еще не конец: механизм регулярных выражений выполнит *возврат*.

Звездочка предпочитает захватить столько текста, сколько это возможно, но ее вполне устроит и нулевое число совпадений. При каждом повторении квантификатора после превышения минимального числа повторений квантификатора регулярное выражение запоминает позицию для последующего возможного возврата. Механизм регулярных выражений может возвращаться к этим позициям, если часть регулярного выражения, следующая за квантификатором, не находит соответствия.

Когда сопоставление ‹<› терпит неудачу, механизм возвращается назад, заставляя ‹.*› вернуть один символ из своего совпадения. После этого снова предпринимается попытка найти соответствие ‹<› с последним символом в файле. И снова неудача, и снова возврат на один

символ, и попытка найти соответствие ‹<› с предпоследним символом в файле. Этот процесс продолжается, пока сопоставление ‹<› не увенчается успехом. Если для элемента выражения ‹<› так и не будет найдено соответствия даже после исчерпания всех возможных возвратов, то попытка найти соответствие для всего регулярного выражения будет считаться неудачной.

Если в ходе выполнения возвратов элемент ‹<› совпал в некоторой позиции, производится попытка найти соответствие элементу ‹/›. В случае неудачи механизм опять начинает выполнять возвраты. Этот процесс повторяется, пока элемент ‹</p›› не совпадет полностью.

Так в чем же собственно проблема? Так как звездочка является максимальным квантификатором, неправильному регулярному выражению будет соответствовать все содержимое файла XHTML, начиная от первого тега и заканчивая последним тегом . Но правильное решение состоит в том, чтобы отыскать параграф XHTML, то есть необходимо найти соответствие первого тега с первым тегом , следующим за ним.

В таких случаях применяются *минимальные* квантификаторы. Любой квантификатор можно сделать минимальным, поместив за ним знак вопроса: ‹*?›, ‹+?›, ‹??› и ‹{7,42}?› – все это минимальные квантификаторы.

Минимальные квантификаторы тоже позволяют выполнять возврат, но в другом режиме. Минимальный квантификатор совпадает минимально возможное число раз, сохраняет позицию возврата и позволяет механизму продолжить сопоставление остальной части регулярного выражения. Если сопоставление с остатком регулярного выражения терпит неудачу и механизм выполняет возврат, минимальный квантификатор повторяется еще раз. Если регулярное выражение продолжает выполнять возвраты, квантификатор продолжает сопоставление, пока не будет достигнуто максимально возможное число повторений или пока повторяемый элемент не потерпит неудачу.

В выражении ‹ .*? › используется минимальный квантификатор, чтобы обеспечить корректное обнаружение единственного параграфа XHTML. После того как будет найдено совпадение с ‹ ›, минимальный квантификатор ‹.*?› просто запоминает позицию для возврата. Если подвыражение ‹ › обнаруживает совпадение сразу же вслед за , результатом сопоставления будет пустой параграф. В противном случае механизм регулярных выражений возвращается к подвыражению ‹.*?›, которому соответствует единственный символ. Если после этого сопоставление с ‹ › опять потерпит неудачу, будет выполнено сопоставление ‹.*?› со следующим символом. Эти действия будут выполняться, пока не будет обнаружено совпадение с ‹ › или ‹.*?› потерпит неудачу. Поскольку точке соответствуют любые символы, неудачное сопоставление с ‹.*?› возможно только по достижении конца файла XHTML.

Квантификаторы ‹*› и ‹*?› дают одинаковые совпадения регулярного выражения. Единственное различие между ними заключается в том, в каком порядке проверяются возможные совпадения. Максимальный квантификатор обнаружит максимально возможное соответствие. Минимальный квантификатор обнаружит минимально возможное соответствие.

Если это возможно, лучшее решение заключается в том, чтобы обеспечить существование единственно возможного соответствия. Регулярные выражения, совпадающие с числами, которые приводятся в рецепте 2.12, по-прежнему будут совпадать с теми же числами, если все имеющиеся в них квантификаторы заменить их минимальными версиями. Причина состоит в том, что части этих регулярных выражений, где имеются квантификаторы, и части, следующие за ними, являются взаимоисключающими. Метасимвол ‹\d› совпадает с цифрой, а метасимвол ‹\b›, следующий за ‹\d›, совпадает, только если следующий символ не является цифрой (или буквой).

Сравнение результатов работы выражений ‹\d+\b› и ‹\d+?\b› с различными текстами может помочь понять принцип действия минимальных и максимальных квантификаторов. Максимальная и минимальная версии воспроизводят одинаковые результаты, но испытуемый текст при этом просматривается по-разному.

В случае использования выражения ‹\d+\b› для сопоставления с текстом 1234 элемент ‹\d+› совпадет со всеми цифрами. После этого будет обнаружено совпадение с метасимволом ‹\b› и в результате будет найдено общее совпадение с выражением. В случае использования выражения ‹\d+?\b›, элемент ‹\d+?› сначала совпадет только с цифрой 1. Сопоставление метасимвола ‹\b› с позицией между 1 и 2 потерпит неудачу. Затем соответствие с ‹\d+?› будет расширено до 12 и снова сопоставление ‹\b› потерпит неудачу. Так будет продолжаться, пока ‹\d+?› не совпадет с 1234, после чего успешно будет обнаружено совпадение с ‹\b›.

Если в качестве испытуемого текста взять 1234X, первое регулярное выражение, ‹\d+\b›, по-прежнему будет совпадать с фрагментом 1234. Но сопоставление с ‹\b› потерпит неудачу. Подвыражение ‹\d+› вернет одно совпадение, оставив 123. Сопоставление с ‹\b› снова потерпит неудачу. Подвыражение ‹\d+› будет возвращать совпадения, пока не останется его минимум, то есть 1, но и в этом случае сопоставление с ‹\b› будет терпеть неудачу. В результате попытка найти совпадение для всего выражения будет считаться неудачной.

При применении выражения ‹\d+?\b› к тексту 1234X подвыражение ‹\d+?› сначала совпадет только с 1. Сопоставление метасимвола ‹\b› с позицией между 1 и 2 потерпит неудачу. Затем соответствие с ‹\d+?› будет расширено до 12, и снова сопоставление ‹\b› потерпит неудачу. Так будет продолжаться, пока ‹\d+?› не совпадет с 1234, но и после это-

го сопоставление с ‹\b› будет терпеть неудачу. Подвыражение ‹\d+?› попытается расширить соответствие, но сопоставление подвыражения ‹\d› с символом X потерпит неудачу. В результате попытка найти совпадение для всего выражения будет считаться неудачной.

Когда между границами слова помещается подвыражение ‹\d+›, оно должно будет либо совпасть со всеми цифрами в испытуемом тексте, либо потерпеть неудачу. Использование минимальной версии квантификатора не изменит результирующего совпадения или неудачи. Фактически, использование выражения ‹\b\d+\b› было бы более предпочтительно, если бы оно могло обнаруживать совпадение вообще без возвратов. В следующем рецепте описывается, как для достижения этого эффекта можно использовать захватывающий квантификатор ‹\b\d++\b›, по крайней мере в некоторых диалектах.

См. также

Рецепты 2.8, 2.9, 2.12, 2.14 и 2.15.

2.14. Устранение бесполезных возвратов

Задача

В предыдущем рецепте описывались различия между максимальными и минимальными квантификаторами, и как они выполняют возврат. В некоторых ситуациях в возвратах нет никакой необходимости.

В выражении ‹\b\d+\b› используется максимальный квантификатор, а в выражении ‹\b\d+?\b› – минимальный. Оба они совпадают с целым числом. В одном и том же тексте они обнаруживают одно и то же совпадение. Любые выполняемые ими возвраты не являются необходимыми. Нужно переписать это регулярное выражение и явно ликвидировать все возвраты, что обеспечит более высокую производительность.

Решение

```
\b\d++\b
```

Параметры: нет
Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Простейшее решение заключается в использовании захватывающего квантификатора. Но такие квантификаторы поддерживаются не всеми современными диалектами регулярных выражений.

```
\b(?>\d+)\b
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

Атомарная группировка оказывает тот же эффект. Она имеет менее удобочитаемый синтаксис, но поддерживается более широким кругом диалектов, чем захватывающие квантификаторы.

JavaScript и Python не поддерживают ни захватывающие квантификаторы, ни атомарную группировку. В этих двух диалектах нет никакой возможности ликвидировать бесполезные возвраты.

Обсуждение

Захватывающий квантификатор напоминает максимальный квантификатор: он пытается отыскать максимально возможное число соответствий. Различие состоит в том, что захватывающий квантификатор не возвращает совпадения, даже когда возврат является единственным способом, который мог бы обеспечить совпадение для остальной части выражения. Захватывающие квантификаторы на запоминают позиции совпадений для возврата.

Любой квантификатор можно сделать захватывающим, поместив вслед за ним символ «плюс». Например, все следующие квантификаторы являются захватывающими: ‹*+›, ‹++›, ‹?+› и ‹{7,42}+›.

Захватывающие квантификаторы поддерживаются Java 4 и выше с момента включения пакета `java.util.regex`. Все версии PCRE, рассматриваемые в этой книге (с 4 по 7), поддерживают захватывающие квантификаторы. Диалект Perl поддерживает их, начиная с версии Perl 5.10. Классический механизм регулярных выражений Ruby не поддерживает захватывающие квантификаторы, а механизм Onigurama, по умолчанию используемый в Ruby 1.9, обеспечивает их поддержку.

Обертывание максимального квантификатора *атомарной группировкой* дает тот же эффект, что и захватывающий квантификатор. Когда механизм регулярных выражений покидает атомарную группу, все позиции возврата, сохраненные квантификатором, и варианты выбора внутри группы отбрасываются. Атомарная группировка имеет синтаксис: ‹(?>*regex*)›, где ‹*regex*› – любое регулярное выражение. Атомарная группа по своей сути является несохраняющей группой, которая дополнительно отвергает возвраты. Знак вопроса не является квантификатором, просто открывающая скобка состоит из трех символов ‹(?>›.

В случае применения выражения ‹\b\d++\b› (с захватывающим квантификатором) к тексту `123abc 456` метасимвол ‹\b› совпадет с началом испытуемого текста, а подвыражение ‹\d++› совпадет с фрагментом 123. В этом оно не отличается от выражения ‹\b\d+\b› (с максимальным квантификатором). Но затем при сопоставлении с позицией между 3 и a второй метасимвол ‹\b› потерпит неудачу.

Захватывающий квантификатор не сохраняет позиции возвратов. Поскольку в этом выражении отсутствуют другие квантификаторы или варианты выбора, у механизма не остается возможности попробовать

найти иные соответствия, когда сопоставление со второй границей слова терпит неудачу. В результате механизм тут же объявляет о неудаче поиска соответствия в позиции символа 1.

Механизм попытается сопоставить регулярное выражение, начиная со следующего символа в строке, и использование захватывающего квантификатора не повлияет на такое его поведение. Если необходимо обеспечить совпадение регулярного выражения со всем испытуемым текстом, следует использовать якорные метасимволы, как обсуждалось в рецепте 2.5. В конечном счете механизм регулярных выражений дойдет до попытки сопоставить выражение, начиная с позиции символа 4, и обнаружит соответствие 456.

Отличие выражения с максимальным квантификатором состоит в том, что в случае первой неудачной попытки сопоставить метасимвол ‹\b› максимальный квантификатор уступит одно свое соответствие. После этого механизм проверит (без всякой пользы) совпадение ‹\b› с позицией между символами 2 и 3, а затем между символами 1 и 2.

Процесс поиска совпадения с применением атомарной группировки в сущности выполняется точно так же. При применении выражения ‹\b(?>\d+)\b› (захватывающего) к тексту 123abc 456 граница слова совпадет с началом испытуемого текста. Механизм регулярных выражений войдет в атомарную группу и подвыражение ‹\d+› совпадет с фрагментом 123. Затем механизм выйдет из атомарной группы и все позиции возврата, сохраненные подвыражением ‹\d+›, будут утрачены. Когда сопоставление со вторым метасимволом ‹\b› потерпит неудачу, у механизма регулярных выражений не останется вариантов, как сразу же объявить о неудачной попытке. Как и при использовании захватывающего квантификатора, в конечном итоге это выражение обнаружит соответствие 456.

Мы говорили, что захватывающий квантификатор не сохраняет позиции возврата, а атомарная группировка отбрасывает их. Это позволяет лучше понять процесс сопоставления, но не стоит заострять свое внимание на различиях, так как в некоторых диалектах их вообще может не быть. Во многих диалектах ‹x++› – это всего лишь сокращенная форма записи ‹(?>x+)›, и оба выражения работают совершенно одинаково. Для конечного результата совершенно неважно, сразу механизм не запоминает позиции возврата или отбрасывает их позднее.

Основные же различия между захватывающими квантификаторами и атомарной группировкой заключаются в том, что захватывающий квантификатор применяется к единственному элементу регулярного выражения, тогда как атомарная группа может заключать в себя целое регулярное выражение.

Выражения ‹\w++\d++› и ‹(?>\w+\d+)› – это не одно и то же. Выражение ‹\w++\d++›, эквивалентом которого является выражение ‹(?>\w+)(?>\d+)›,

не совпадет с текстом abc123. Подвыражение ‹\w++› совпадет со всем текстом abc123, но затем механизм регулярных выражений попытается найти в конце испытуемого текста соответствие для ‹\d++›. Поскольку в тексте не останется символов, которые могли бы соответствовать, подвыражение ‹\d++› потерпит неудачу. В отсутствие позиций возврата попытка сопоставить регулярное выражение провалится.

В выражении ‹(?>\w+\d+)› используются два максимальных квантификатора, заключенные в такую же атомарную группу. Внутри этой группы возвраты выполняются как обычно. Позиции возвратов отбрасываются только после того, как механизм регулярных выражений покинет группу. При применении этого выражения к тексту abc123 подвыражение ‹\w+› совпадет с фрагментом abc123. Максимальные квантификаторы запоминают позиции возвратов. Когда сопоставление с ‹\d+› потерпит неудачу, подвыражение ‹\w+› уступит один символ. В результате для ‹\d+› будет найдено соответствие 3. После этого механизм регулярных выражений покинет группу и отбросит все позиции возврата, сохраненные подвыражениями ‹\w+› и ‹\d+›. Поскольку был достигнут конец регулярного выражения, различия становятся несущественными. Общее совпадение с регулярным выражением обнаружено.

Если бы конец регулярного выражения не был достигнут, как, например, при использовании выражения ‹(?>\w+\d+)\d+›, механизм регулярных выражений оказался бы в той же ситуации, что и в случае с выражением ‹\w++\d++›. Для второго подвыражения ‹\d+› не остается символов для сопоставления в конце испытуемого текста. А так как позиции возврата уже были отброшены, механизму регулярных выражений остается только объявить о неудаче.

Захватывающие квантификаторы и атомарная группировка не только оптимизируют регулярные выражения. Они могут оказывать влияние на соответствия, обнаруживаемые регулярным выражением, устраняя результаты, которые могли бы быть получены за счет возвратов.

Этот рецепт продемонстрировал, что применение захватывающих квантификаторов и атомарной группировки позволяет выполнить минимальную оптимизацию, которая может даже не повлиять на результаты тестов на скорость выполнения. В следующем рецепте будет показано, что атомарная группировка может оказывать весьма существенное влияние.

См. также

Рецепты 2.12 и 2.15.

2.15. Предотвращение бесконтрольных повторений

Задача

Создать единственное регулярное выражение, совпадающее с целым файлом HTML, которое будет проверять наличие тегов `html`, `head`, `title` и `body` и их вложенность. Выражение не должно совпадать с файлами HTML, в которых отсутствуют требуемые теги.

Решение

```
<html>(?>.*?<head>)(?>.*?<title>)(?>.*?</title>)↵
(?>.*?</head>)(?>.*?<body[^>]*>)(?>.*?</body>).*?</html>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Ruby

Диалекты JavaScript и Python не поддерживают атомарную группировку. В них нет никакой возможности устранить бесполезные возвраты. При программировании на JavaScript или Python эту проблему можно решить за счет поиска литералов тегов по одному, выполняя поиск следующего тега по остатку испытуемого текста после того, как будет найден предыдущий.

Обсуждение

Правильное решение этой задачи будет проще понять, если начать с более простого решения:

```
<html>.*?<head>.*?<title>.*?</title>↵
.*?</head>.*?<body[^>]*>.*?</body>.*?</html>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если проверить работу этого выражения, применив его к корректному файлу HTML, оно прекрасно справится со своей задачей. Подвыражение `<.*?>` пропустит все, потому включен параметр «точке соответствуют границы строк». Минимальная звездочка гарантирует, что регулярное выражение будет продвигаться вперед по одному символу за раз, каждый раз проверяя совпадение со следующим тегом. Все это объясняется в рецептах 2.4 и 2.13.

Однако проблемы возникнут, если применить это выражение к испытуемому тексту, в котором отсутствуют какие-либо теги HTML. Самый худший случай – отсутствие тега `</html>`.

Предположим, что механизм регулярных выражений обнаружил соответствия со всеми предыдущими тегами и теперь занят разворачиванием последнего подвыражения ‹.*?›. Поскольку подвыражение ‹</html›› никогда не обнаружит соответствие, совпадение с ‹.*?› будет расширяться вплоть до конца файла. Когда это произойдет, попытка найти совпадение со всем регулярным выражением будет признана неудачной.

Но это еще не конец. Другие шесть подвыражений ‹.*?› имеют сохраненные позиции возврата, которые позволяют расширять их соответствия. Когда последнее подвыражение ‹.*?› потерпит неудачу, расширяться начнет предшествующее ему, постепенно захватывая совпадение <u></body></u>. Тот самый текст, который ранее уже был определен, как совпадение с литералом ‹</body›› в регулярном выражении. Это подвыражение ‹.*?› тоже будет расширено до самого конца файла, как и все предыдущие точки с минимальным квантификатором. Только когда первое подвыражение ‹.*?› достигнет конца файла, механизм регулярных выражений объявит о неудаче.

Это регулярное выражение имеет сложность $O(n^7)$ для самого тяжелого случая, – то есть длина испытуемого текста в седьмой степени. В выражении имеется семь точек с минимальным квантификатором, потенциально способных пройти попытки сопоставления до конца файла. Если размер файла увеличится в два раза, регулярному выражению может понадобиться до 128 раз больше шагов, чтобы обнаружить отсутствие совпадения.

Мы называем это *катастрофическими возвратами*. Такое большое число возвратов может привести к тому, что регулярное выражение навечно «застрянет» в процедуре поиска или вызовет аварийное завершение приложения. Некоторые реализации регулярных выражений способны прервать неконтролируемую последовательность повторений раньше, но даже в этом случае регулярное выражение будет способно до минимума снизить производительность приложения.

Катастрофические возвраты – это пример класса явлений, известного как *комбинаторный взрыв*, когда пересекаются несколько независимых условий и необходимо проверить все возможные комбинации. Можно даже сказать, что в этом случае регулярное выражение – это *декартово произведение* различных операторов повторения.

Решение этой проблемы заключается в использовании атомарной группировки, предотвращающей бесполезные возвраты. Нет никакого смысла расширять соответствие для шестого подвыражения ‹.*?›, после того как будет найдено совпадение для ‹</body››. Если попытка найти совпадение с с ‹</html›› не увенчалась успехом, расширение соответствия для шестой точки с минимальным квантификатором не приведет к появлению закрывающего тега html.

Чтобы остановить расширение соответствия для квантифицированного элемента регулярного выражения, когда было найдено совпадение со следующим за ним разделителем, необходимо квантифицированную часть выражения и разделитель поместить в атомарную группу: ‹(?>.*?</body>)›. Теперь механизм регулярных выражений будет отбрасывать все позиции, совпавшие с ‹(?>.*?</body>)›, после того как будет найдено соответствие для ‹</body›. Если позднее сопоставление с ‹</html›› потерпит неудачу, механизм регулярных выражений забудет о существовании ‹(?>.*?</body>)› и дальнейшее расширение соответствия происходить не будет.

Если точно так же оформить все остальные подвыражения ‹.*?›, ни одно из них не будет расширять свое соответствие. Хотя в выражении по-прежнему останутся все семь точек с минимальными квантификаторами, их совпадения никогда больше не будут перекрываться. Это снизит сложность регулярного выражения до O(*n*), которая имеет линейную зависимость от длины испытуемого текста. Никакое регулярное выражение не может быть более эффективным, чем это.

Варианты

При желании катастрофическое действие возвратов можно наблюдать, применив регулярное выражение ‹(x+x+)+y› к тексту xxxxxxxxxx. Если оно потерпит неудачу слишком быстро, добавьте в испытуемый текст еще один символ x. Повторяйте добавлять символы, пока регулярное выражение не будет выполняться слишком долго или пока приложение не завершится аварийно. Для этого не потребуется добавлять слишком много символов x, если не использовать Perl для тестирования.

Из всех диалектов регулярных выражений, обсуждаемых в этой книге, только Perl способен определить, что выражение слишком сложное и прервать попытки поиска соответствий без аварийного завершения приложения.

Это регулярное выражение имеет сложность $O(2^n)$. Когда попытка найти соответствие для ‹y› потерпит неудачу, механизм регулярных выражений опробует все возможные перестановки повторений для каждого элемента ‹x+› и группы, содержащей их. Например, одна такая перестановка, возникающая в процессе сопоставления, – это когда первому подвыражению ‹x+› соответствует фрагмент xxx, второму подвыражению ‹x+› – символ x, и группа повторяется три или более раз для каждого совпадения ‹x+› с x. Для текста с десятью символами x будет выполнено 1024 перестановки. Если увеличить число символов в испытуемом тексте до 32, будет получено более 4 миллиардов возможных перестановок, что наверняка приведет к исчерпанию доступной памяти механизмом регулярных выражений, если в нем отсутствует «аварийная кнопка», которая позволит ему сдаться и заявить, что регулярное выражение слишком сложное.

В данном случае бессмысленное регулярное выражение легко можно записать, как ‹xx+y›, которое будет отыскивать те же совпадения, но зависимость сложности от длины текста будет уже линейной. На практике в случае более сложных регулярных выражений правильные решения могут быть не такими очевидными.

Суть состоит в том, чтобы научиться видеть ситуации, когда две или более частей регулярного выражения могут совпасть с одним и тем же текстом. В таких ситуациях может потребоваться применить атомарную группировку, чтобы исключить для механизма регулярных выражений возможность попытаться разделить испытуемый текст между этими частями выражения всеми возможными способами. Регулярные выражения всегда следует проверять на (длинных) испытуемых текстах, содержащих фрагменты, частично, но не полностью соответствующие регулярному выражению.

См. также

Рецепты 2.13 и 2.14.

2.16. Проверка соответствия без включения его в общее соответствие

Задача

Отыскать любое слово, расположенное между парой тегов и HTML, без включения этих тегов в общее соответствие регулярному выражению. Например, для испытуемого текста My cat is furry, правильным соответствием будет cat.

Решение

```
(?<=<b>)\w+(?=</b>)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 поддерживают опережающую проверку ‹(?=)›, но не поддерживают ретроспективную ‹(?<=)›.

Обсуждение

Проверка соседних символов

Четыре вида групп, выполняющих *проверку соседних символов* и поддерживаемых современными диалектами регулярных выражений, имеют специальный механизм отказа от текста, совпавшего с частью регулярного выражения внутри таких групп. По сути проверка сосед-

них символов позволяет выяснить, совпадает ли близлежащий текст, не помещая его в совпадение.

Проверка предыдущих символов называется *ретроспективной*. Это единственная конструкция в регулярных выражениях, которая просматривает текст справа налево, а не слева направо. *Позитивная ретроспективная проверка* имеет синтаксис ‹(?<=text)›. Четыре символа ‹(?<=› образуют открывающую скобку. Что допускается включать в содержимое ретроспективной проверки, показанное здесь как ‹text›, зависит от используемого диалекта регулярных выражений. Но как бы то ни было, простые строковые литералы, такие как ‹(?<=)›, использовать можно в любом диалекте.

Ретроспективная проверка проверяет, совпадает ли текст, указанный в ней, с текстом, расположенным непосредственно перед позицией, где механизм регулярных выражений встретил ретроспективную проверку. Применительно к тексту My cat is furry ретроспективная проверка ‹(?<=)› будет терпеть неудачу, пока не будет выполнена попытка сопоставления в позиции символа c. Тогда механизм регулярных выражений войдет в группу ретроспективной проверки, предложив ей проверить текст слева. Элемент ‹› совпадет с текстом левее символа c. Механизм покинет ретроспективную проверку в этой позиции и исключит текст, совпавший с ретроспективной проверкой, из общего совпадения. Говоря другими словами, общее соответствие возвращается к состоянию, которое было на момент, когда механизм вошел в ретроспективную проверку. В данном случае перед попыткой сопоставления с символом c в испытуемой строке текущее совпадение имело нулевую длину. Ретроспективная проверка лишь проверяет наличие соответствия с ‹›. При этом текст, совпавший с ней, в общее совпадение не включается. По этой причине проверки соседних символов называют еще *проверками с нулевой длиной совпадения*.

После совпадения с ретроспективной проверкой предпринимается попытка сопоставить символьный класс ‹\w+› с одним или более символами. Он совпадет со словом cat. Подвыражение ‹\w+› не находится внутри какой-либо проверки соседних символов или группы, поэтому оно совпадет с текстом cat обычным способом. Мы говорим, что ‹\w+› совпадает и *поглощает* текст cat, тогда как проверка соседних символов может совпадать, но никогда ничего не поглощает.

Проверка может также выполняться в прямом направлении, в каком регулярное выражение просматривает текст. Такая проверка называется *опережающей*. Опережающая проверка в равной степени поддерживается всеми диалектами регулярных выражений, рассматриваемыми в этой книге. *Позитивная опережающая проверка* имеет синтаксис ‹(?=*regex*)›. Первые три символа ‹(?=› образуют открывающую скобку группы. Все, что допускается использовать в регулярных выражениях, можно использовать внутри опережающей проверки вместо ‹*regex*›.

Когда подвыражение ‹\w+› в выражении ‹(?<=)\w+(?=)› совпадет с фрагментом cat в тексте My cat is furry, механизм регулярных выражений войдет в опережающую проверку. Единственная особенность поведения опережающей проверки в этой точке заключается в том, что механизм регулярных выражений запоминает, какая часть текста уже совпала, связывая ее с опережающей проверкой. Совпадение с ‹› будет обнаружено как обычно, и механизм регулярных выражений покинет опережающую проверку. Регулярное выражение внутри опережающей проверки обнаружило совпадение, поэтому и сама опережающая проверка считается совпавшей. Механизм регулярных выражений отбросит текст, совпавший с опережающей проверкой, и восстановит соответствие в состояние, существовавшее на момент входа в опережающую проверку. Общее текущее совпадение будет содержать текст cat. Поскольку на этом регулярное выражение заканчивается, в качестве окончательного результата поиска соответствия принимается текст cat.

Негативная проверка

Форма записи ‹(?!*regex*)› с восклицательным знаком вместо знака равенства – это *негативная опережающая проверка*. Негативная опережающая проверка работает точно так же, как и позитивная, за исключением того, что позитивная опережающая проверка совпадает, когда совпадает регулярное выражение, находящееся внутри нее, а негативная опережающая проверка совпадает, когда регулярное выражение внутри нее терпит неудачу.

Процесс сопоставления протекает точно так же. Механизм сохраняет текущее найденное совпадение при входе в негативную опережающую проверку и пытается сопоставить регулярное выражение внутри нее обычным способом. Если подвыражение совпадает, опережающая проверка терпит неудачу и механизм регулярных выражений выполняет возврат. Если совпадение с подвыражением обнаружено не будет, механизм восстанавливает прежнее соответствие и продолжает сопоставление оставшейся части регулярного выражения.

Точно так же форма записи ‹(?<!*text*)› является *негативной ретроспективной проверкой*. Негативная ретроспективная проверка совпадает, когда ни одна из альтернатив внутри ретроспективной проверки не обнаруживает совпадение с текстом, предшествующим позиции в испытуемом тексте, которая была достигнута механизмом регулярных выражений.

Различные уровни ретроспективной проверки

Опережающая проверка выполняется довольно просто. Все диалекты регулярных выражений, рассматриваемые в этой книге, позволяют помещать в нее полноценные регулярные выражения. Все конструкции, какие только можно использовать в регулярном выражении, можно использовать и в опережающей проверке. Внутрь опережающей проверки

можно даже вкладывать другие опережающие и ретроспективные проверки. От таких нагромождений голова может пойти кругом, но механизм регулярных выражений справится с этим без труда.

Ретроспективная проверка – это совсем другая история. Программное обеспечение поддержки регулярных выражений всегда предназначалось для поиска по тексту только в направлении слева направо. Ретроспективный поиск часто реализуется довольно грубо: механизм регулярных выражений определяет количество символов, помещенных внутрь ретроспективной проверки, переходит назад на указанное число символов и затем сравнивает текст в ретроспективной проверке с испытуемым текстом слева направо.

По этой причине первые реализации допускали вставлять внутрь ретроспективной проверки только текст фиксированной длины. Диалекты Perl, Python и Ruby 1.9 пошли на один шаг вперед, позволив использовать операторы выбора и символьные классы и дав тем самым возможность вставлять в ретроспективную проверку несколько строковых литералов фиксированной длины. Выражение ‹(?<=one|two|three|forty-two|gr[ae]y)› содержит все, с чем могут иметь дело ретроспективные проверки.

Внутренние реализации диалектов Perl, Python и Ruby 1.9 разложат это выражение на шесть ретроспективных проверок. Сначала они вернутся назад на три символа, чтобы проверить варианты ‹one|two›, затем на четыре символа, чтобы проверить варианты ‹gray|grey›, затем на пять символов, чтобы проверить ‹three›, и, наконец, на девять символов, чтобы проверить ‹forty-two›.

Диалекты PCRE и Java пошли еще на один шаг вперед. Они позволяют использовать внутри ретроспективных проверок любые регулярные выражения, имеющие конечную длину совпадения. Это означает, что внутри ретроспективной проверки можно использовать все, за исключением бесконечных квантификаторов ‹*›, ‹+› и ‹{42,}›. Внутренние реализации диалектов PCRE и Java вычисляют минимальную и максимальную длину текста, который возможно будет соответствовать регулярному выражению внутри ретроспективной проверки. Затем они возвращаются на минимальное число символов и применяют регулярное выражение из ретроспективной проверки в направлении слева направо. Если совпадение не будет обнаружено, они возвращаются еще на один символ назад и повторяют попытку, пока либо ретроспективная проверка не совпадет, либо пока не будет выполнена попытка сопоставить максимальное число символов.

Если все это на ваш взгляд выглядит неэффективным, пусть так. Ретроспективная проверка удобна в использовании, но с ее применением невозможно поставить рекорд в скорости выполнения. Ниже мы представим решение для диалектов JavaScript и Ruby 1.8, которые вообще не поддерживают ретроспективные проверки. В действительности же это

решение намного более эффективно, чем решение на основе применения ретроспективной проверки.

Механизм регулярных выражений платформы .NET является единственным в мире[1], который фактически способен обрабатывать полноценные регулярные выражения внутри ретроспективной проверки и в действительности он применяет такие регулярные выражения в направлении справа налево. И само регулярное выражение в ретроспективной проверке, и испытуемый текст просматриваются справа налево.

Двойная проверка на совпадение с одним и тем же текстом

При использовании ретроспективной проверки в начале регулярного выражения или опережающей проверки в конце возникает ситуация, когда необходимо, чтобы что-то находилось перед или после совпадения с регулярным выражением, без включения этого «что-то» в общее совпадение. Если проверка соседних символов выполняется в середине регулярного выражения, можно применять несколько проверок на совпадение с одним и тем же текстом.

В рецепте 2.3 в разделе «Особенности, характерные для разных диалектов» на стр. 58 (подраздел рецепта 2.3) было показано, как использовать разность символьных классов для поиска цифр алфавита Thai. Операцию вычитания символьных классов поддерживают только .NET и Java.

Символ является тайской цифрой, если он является символом (любым) алфавита Thai и цифрой (любого алфавита). С помощью опережающей проверки можно проверить соответствие обоим требованиям для одного и того же символа:

```
(?=\p{Thai})\p{N}
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

Это регулярное выражение работает только в трех диалектах, поддерживающих алфавиты Юникода, как уже описывалось в рецепте 2.7. Но сам принцип использования опережающей проверки для сопоставления одного и того же символа более одного раза может использоваться во всех диалектах, описываемых в этой книге.

---

[1] Механизм регулярных выражений, реализованный в программе RegexBuddy, также позволяет помещать полноценные регулярные выражения внутрь ретроспективной проверки, Но в нем (пока) отсутствует особенность, которая напоминала бы свойство `RegexOptions.RightToLeft`, имеющееся в реализации платформы .NET, используемое для переворачивания всего регулярного выражения.

Когда механизм регулярных выражений выполняет поиск совпадения для выражения ‹(?=\p{Thai})\p{N}›, он начинает с того, что входит в опережающую проверку для каждой позиции в испытуемой строке, откуда начинается попытка сопоставления. Если символ в данной позиции отсутствует в алфавите Thai (то есть проверка ‹\p{Thai}› терпит неудачу), опережающая проверка терпит неудачу. Это приводит к тому, что вся попытка сопоставления признается неудачной и механизм регулярных выражений вынужден начинать новую попытку со следующего символа.

Когда регулярное выражение достигает символа из алфавита Thai, проверка ‹\p{Thai}› проходит успешно. Вследствие этого и вся опережающая проверка ‹(?=\p{Thai})› признается совпавшей. Выходя из опережающей проверки, механизм регулярных выражений восстанавливает найденное ранее соответствие в прежнее состояние. В данном случае перед первым найденным символом алфавита Thai общее совпадение имело нулевую длину. Затем выполняется проверка ‹\p{N}›. Поскольку опережающая проверка отменяет свое совпадение, подвыражение ‹\p{N}› сравнивается с тем же символом, с которым уже совпала проверка ‹\p{Thai}›. Если этот символ имеет свойство Number Юникода, проверка ‹\p{N}› совпадет. Поскольку подвыражение ‹\p{N}› находится за пределами какой-либо проверки соседних символов, оно поглощает символ и выражение обнаруживает тайскую цифру.

Проверки являются атомарными

Когда механизм регулярных выражений покидает группу проверки соседних символов, он отказывается от текста, совпавшего с проверкой. Поскольку текст совпадения отвергается, все позиции возврата, сохраненные операторами выбора или квантификаторами внутри проверки, также отвергаются. Это обеспечивает атомарность опережающих и ретроспективных проверок. В рецепте 2.15 во всех подробностях описывается атомарная группировка.

В большинстве ситуаций атомарная природа проверки остается невостребованной. Проверка соседних символов – это всего лишь попытка узнать сам факт, совпадет ли регулярное выражение, находящееся внутри нее, или нет. Сколькими способами может произойти это совпадение, не имеет никакого значения, потому что проверка не поглощает ни одного символа из испытуемого текста.

Атомарность начинает обретать значение только при использовании сохраняющих групп внутри опережающей (и ретроспективной, если это допускает используемый диалект) проверки. Даже при том, что опережающая проверка не поглощает текст, механизм регулярных выражений запоминает, какая часть текста совпала с любой из сохраняющих групп внутри опережающей проверки. Если опережающая проверка находится в конце регулярного выражения, в сохраняющих

группах может оказаться совпавший с ними текст, не совпавший с самим регулярным выражением. Если опережающая проверка находится в середине регулярного выражения, может случиться так, что в сохраняющих группах окажутся перекрывающиеся участки испытуемого текста.

Единственная ситуация, когда атомарность проверки может повлиять на общее совпадение регулярного выражения, – это когда за пределами проверки используются обратные ссылки на сохраняющие группы, расположенные внутри проверки. Рассмотрим следующее регулярное выражение:

```
(?=(\d+))\w+\1
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

На первый взгляд это регулярное выражение должно совпадать с текстом 123x12. Элемент ‹\d+› мог бы сохранить фрагмент 12 в первой сохраняющей группе, затем элемент ‹\w+› мог бы совпасть с фрагментом 3x, и, наконец, ссылка ‹\1› могла бы совпасть с фрагментом 12.

Но этого никогда не произойдет. Механизм регулярных выражений входит в проверку и сохраняющую группу. Максимальный метасимвол ‹\d+› совпадает с фрагментом 123. Это совпадение сохраняется в первой сохраняющей группе. Затем механизм выходит из опережающей проверки и переустанавливает позицию поиска совпадений в начало строки, отвергая все позиции возврата, созданные максимальным квантификатором, но запоминает текст 123, сохраненный первой сохраняющей группой.

Теперь предпринимается попытка сопоставить максимальный элемент ‹\w+› с начала строки. Он поглощает весь текст 123x12. Сопоставление с обратной ссылкой ‹\1›, ссылающейся на текст 123, терпит неудачу в конце строки. Подвыражение ‹\w+› уступает один символ. Сопоставление с ‹\1› опять терпит неудачу. Подвыражение ‹\w+› продолжает уступать символы, пока у него не останется минимально возможный фрагмент 1. Сопоставление с обратной ссылкой ‹\1› в позиции, следующей за первым символом 1, также терпит неудачу.

Заключительный фрагмент текста 12 мог бы совпасть с обратной ссылкой ‹\1›, если бы механизм регулярных выражений мог вернуться в опережающую проверку и отказаться от совпадения 123 в пользу совпадения с 12, но механизм регулярных выражений этого не делает.

Механизм регулярных выражений исчерпал все позиции возврата. Подвыражение ‹\w+› уступило все, что могло, а опережающая проверка, выполнившая подвыражение ‹\d+›, отбросила все позиции возврата. В результате попытка отыскать совпадение с регулярным выражением терпит неудачу.

Решение без использования ретроспективной проверки

Все предыдущие снадобья будут бесполезны, если вы пользуетесь Python или JavaScript, потому что вы вообще не сможете воспользоваться ретроспективными проверками. Нет никакого способа решить эту проблему при использовании этих диалектов, но можно обойтись без ретроспективной проверки, использовав сохраняющие группы. Это альтернативное решение также будет работать во всех других диалектах регулярных выражений.

```
(<b>)(\w+)(?=</b>)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Вместо ретроспективной проверки мы использовали сохраняющую группу для открывающего тега ‹<b>›. Интересующую нас часть регулярного выражения, ‹\w+›, мы также поместили в сохраняющую группу.

Если применить это регулярное выражение к тексту `My <b>cat</b> is furry`, общим совпадение для всего регулярного выражения будет фрагмент `<b>cat`. Первая сохраняющая группа захватит фрагмент `<b>`, а вторая – фрагмент `cat`.

Если требуется, чтобы совпадение содержало только слово `cat` (слово между тегами `<b>`), например, чтобы выделить из текста только слово, добиться этого можно простым сохранением не всего совпадения с регулярным выражением, а только текста, совпавшего со второй сохраняющей группой.

Если требуется выполнить поиск с заменой и заменить только слово между тегами, можно просто использовать обратную ссылку на первую сохраняющую группу, чтобы повторно вставить в текст открывающий тег в замещаемый текст. На самом деле, в данном случае сохраняющая группа вообще не нужна, так как открывающий тег всегда остается одним и тем же. Но когда тег может изменяться, сохраняющая группа вставит именно тот тег, который совпал с ней. Подробнее этот прием рассматривается в рецепте 2.21.

Наконец, при желании можно имитировать ретроспективную проверку с помощью двух регулярных выражений. Первое выполняет поиск без ретроспективной проверки. Когда обнаруживается совпадение, нужно скопировать часть испытуемого текста, находящуюся перед совпадением, в новую строковую переменную. Второе регулярное выражение реализует проверку, которая выполнялась в ретроспективной проверке, при этом в конец выражения следует добавить привязку к концу строки (‹\z› или ‹$›). Якорный метасимвол обеспечит совпадение конца второго выражения с концом строки. Поскольку строка извлекается из текста в позиции, где совпало первое выражение, это обеспечит выполнение второй проверки непосредственно перед совпадением с первым выражением.

В JavaScript этот алгоритм можно было бы реализовать так:

```
var mainregexp = /\w+(?=<\/b>)/;
var lookbehind = /<b>$/;
if (match = mainregexp.exec("My <b>cat</b> is furry")) {
    // Найдено слово перед закрывающим тегом </b>
    var potentialmatch = match[0];
    var leftContext = match.input.substring(0, match.index);
    if (lookbehind.exec(leftContext)) {
        // Ретроспективная проверка совпала:
        // содержимое переменной potentialmatch находится между тегами <b>
    } else {
        // Ретроспективная проверка не совпала:
        // переменная potentialmatch содержит не то, что требуется
    }
} else {
    // Слово перед закрывающим тегом </b> не найдено
}
```

См. также

Рецепты 5.5 и 5.6.

2.17. Совпадение с одной из двух альтернатив по условию

Задача

Создать регулярное выражение, совпадающее со списком слов `one`, `two` и `three`, разделенных запятыми. Каждое слово может присутствовать в списке не менее одного раза.

Решение

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}(?(1)|(?!))(?(2)|(?!))(?(3)|(?!))
```

Параметры: нет
Диалекты: .NET, JavaScript, PCRE, Perl, Python

Диалекты Java и Ruby не поддерживают условные операторы. При программировании на языках Java и Ruby (или любых других) можно использовать регулярные выражения без условных операторов, написав дополнительный программный код, проверяя совпадения с каждой из трех сохраняющих групп.

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Диалекты .NET, JavaScript, PCRE, Perl и Python поддерживают *условные операторы* при использовании нумерованных сохраняющих групп. Выражение ‹(?(1)*then*|*else*)› – это условный оператор, проверяющий совпадение с первой сохраняющей группой. Если группа участвовала в совпадении, механизм регулярных выражений попытается найти совпадение ‹*then*›. Если сохраняющая группа до сих пор не участвовала в совпадении, выполняется попытка сопоставить часть ‹*else*›.

Круглые скобки, знак вопроса и вертикальная черта – все эти символы составляют синтаксис условного оператора. В данном случае их значение отличается от обычного. В частях ‹*then*› и ‹*else*› допускается использовать любые регулярные выражения. Единственное ограничение состоит в том, что при необходимости использовать оператор выбора в какой-либо из частей альтернативы следует заключать в группу. В условном операторе допускается использовать только один символ вертикальной черты.

При желании можно опустить любую из частей ‹*then*› или ‹*else*›. Пустое регулярное выражение всегда имеет совпадение нулевой длины. В решении этой задачи используются три условных оператора, в которых опущена часть ‹*then*›. Если сохраняющая группа участвовала в сопоставлении, считается, что условие выполняется.

Пустая негативная опережающая проверка, ‹(?!)›, заполняет часть ‹*else*›. Поскольку пустое регулярное выражение всегда совпадает, негативная опережающая проверка, содержащая пустое регулярное выражение, всегда будет терпеть неудачу. То есть условие ‹(?(1)|(?!))› никогда не будет выполняться, если первая сохраняющая группа не совпала с чем-нибудь.

Поместив каждую из трех обязательных альтернатив в отдельную сохраняющую группу, можно использовать три условных оператора в конце регулярного выражения, чтобы проверить наличие совпадений для каждой сохраняющей группы.

Диалект .NET поддерживает также именованные условные операторы. Выражение ‹(?(*name*)*then*|*else*)› проверяет, участвовала ли сохраняющая группа с именем name в сопоставлении к настоящему моменту.

Чтобы лучше понять, как работают условные операторы, рассмотрим регулярное выражение ‹(a)?b(?(1)c|d)›. Это, по сути, более сложная форма записи ‹abc|bd)›.

Если испытуемый текст начинается с символа a, он сохраняется в первой сохраняющей группе. В противном случае первая сохраняющая группа вообще не участвует в сопоставлении. Знак вопроса, находящийся за пределами сохраняющей группы, имеет важное значение, так как он делает всю группу необязательной. Если символ a отсутствует,

группа повторяется ноль раз и никогда не получит возможность сохранить что-либо. Она не может сохранить строку нулевой длины.

Если бы использовалось подвыражение ‹(a?)›, группа всегда участвовала бы в сопоставлении. Так как в этом подвыражении после группы нет квантификатора, она повторялась бы точно один раз. Группа сохранила бы символ a либо не сохранила бы ничего.

Независимо от наличия совпадения с ‹a›, далее идет следующая лексема ‹b›. За ней следует проверка условия. Если сохраняющая группа участвовала в сопоставлении, даже если она сохранит строку нулевой длины (что здесь невозможно), будет предпринята попытка найти соответствие для ‹c›. В противном случае будет произведена попытка сопоставления с ‹d›.

Говоря простым языком, выражение ‹(a)?b(?(1)c|d)› либо совпадает с фрагментом ab, за которым следует c, либо совпадает с b, за которым следует d.

В диалектах .NET, PCRE и Perl, но не в диалекте Python, условные операторы могут также выполнять проверку соседних символов. Выражение ‹(?(?=*if*)*then*|*else*)› сначала выполняет ‹(?=*if*)›, как обычную опережающую проверку. Как это работает, описывается в рецепте 2.16. Если проверка увенчалась успехом, выполняется попытка сопоставления с частью ‹*then*›. В противном случае – с частью ‹*else*›. Поскольку проверка ничего не поглощает из испытуемого текста, обе части ‹*then*› и ‹*else*› применяются к одной и той же позиции в испытуемом тексте, где производилась проверка условия ‹*if*›, независимо от результата проверки.

В условном операторе вместо опережающей проверки можно использовать ретроспективную проверку. Можно также использовать негативные проверки, хотя мы не рекомендуем делать это, так как можно только все запутать, перевернув значения подвыражений «then» и «else».

Выражение с условием, использующим проверку, может быть записано без применения условного оператора, например: ‹(?=*if*)*then*|(?!*if*)*else*›. Если положительная опережающая проверка дает положительный результат, выполняется попытка сопоставления части ‹*then*›. В противном случае выполняется попытка сопоставить альтернативный вариант. Затем то же самое делает негативная опережающая проверка. Негативная опережающая проверка завершается успехом, когда подвыражение ‹*if*› не находит соответствия, что происходит гарантированно, так как проверка ‹(?=*if*)› уже потерпела неудачу. В результате предпринимается попытка сопоставить подвыражение ‹*else*›. Размещение опережающей проверки в условном операторе экономит время, так как в этом случае условие ‹*if*› проверяется всего один раз.

См. также

Рецепты 2.9 и 2.16.

2.18. Добавление комментариев в регулярные выражения

Задача

Выражение `‹\d{4}-\d{2}-\d{2}›` совпадает с датами в формате yyyy-mm-dd, но не выполняет проверку чисел на корректность. Такое простое регулярное выражение вполне может использоваться на практике, когда заранее известно, что текст не может содержать некорректные даты. Необходимо добавить комментарии в это регулярное выражение, чтобы описать, что делают его части.

Решение

```
\d{4} # Год
-     # Разделитель
\d{2} # Месяц
-     # Разделитель
\d{2} # День
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Режим свободного форматирования

Регулярные выражения быстро становятся очень сложными и трудными для понимания. Так же как и при работе с исходными текстами программ, желательно комментировать все регулярные выражения, за исключением самых тривиальных.

Все диалекты регулярных выражений, рассматриваемые в этой книге, за исключением JavaScript, предлагают альтернативный синтаксис оформления регулярных выражений, который существенно упрощает добавление комментариев в регулярные выражения. Перейти на использование этого синтаксиса можно, включив параметр *свободного форматирования*. В различных языках программирования этот параметр носит разные имена.

В .NET необходимо включить параметр `RegexOptions.IgnorePatternWhitespace`. В Java – передать флаг `Pattern.COMMENTS`. В Python этот флаг называется `re.VERBOSE`, в Perl и Ruby для этих целей используется флаг `/x`.

Включение режима свободного форматирования влечет за собой появление двух эффектов. В этом режиме в число метасимволов включается символ решетки (#), имеющий специальное назначение за пределами символьных классов. С символа решетки начинаются комментарии, которые продолжаются до конца строки или до конца регулярного выражения (в зависимости от того, какое условие выполнится первым). Символ решетки и все, что следует за ним, просто игнорируются механизмом регулярных выражений. Чтобы оформить поиск совпадения с символом решетки, его следует поместить в символьный класс ‹[#]› или экранировать ‹\#›.

Другой эффект состоит в том, что пробельные символы, в число которых входят пробелы, символы табуляции и символы перевода строки, также игнорируются за пределами символьных классов. Чтобы оформить поиск совпадения с символом пробела, его следует поместить в символьный класс ‹[•]› или экранировать ‹\•›. Для обеспечения большей удобочитаемости можно использовать шестнадцатеричную экранированную последовательность ‹\x20› или кодовый пункт Юникода ‹\u0020› или ‹\x{0020}›. Чтобы оформить поиск совпадения с символом табуляции, следует использовать метасимвол ‹\t›. Для поиска конца строки следует использовать комбинацию ‹\r\n› (Windows) или ‹\n› (UNIX/Linux/OS X).

Режим свободного форматирования не влияет на порядок оформления символьных классов. Символьный класс – это единый элемент. Любые пробельные символы или символы решетки внутри символьных классов рассматриваются как литералы, добавленные в символьный класс. Нет никакой возможности разбить символьный класс на отдельные части, чтобы прокомментировать их.

Символьные классы Java, поддерживающие свободное форматирование

Регулярные выражения едва ли заслужили бы столь высокую репутацию, если бы один диалект был несовместим с другими. В данном случае Java – исключение из правил.

В диалекте Java символьные классы не являются едиными и неделимыми элементами. При включении режима свободного форматирования диалект Java начинает игнорировать пробельные символы в символьных классах, а символы решетки в символьных классах играют роль начала комментариев. Это означает потерю возможности использовать ‹[•]› и ‹[#]› для поиска соответствий с литералами этих символов. Вместо них придется использовать экранированные последовательности ‹\u0020› и ‹\#›.

Варианты

```
(?#Year)\d{4}(?#Separator)-(?#Month)\d{2}-(?#Day)\d{2}
```

Параметры: нет
Диалекты: .NET, PCRE, Perl, Python, Ruby

Если по каким-то причинам невозможно или нежелательно использовать режим свободного форматирования, можно добавлять комментарии в форме ‹(?#*comment*)›. Все символы между ‹(?#› и ‹)› будут игнорироваться.

К сожалению, диалект JavaScript, который единственный из рассматриваемых в этой книге не поддерживает режим свободного форматирования, также не поддерживает и альтернативный синтаксис оформления комментариев. Диалект Java также не поддерживает его.

```
(?x)\d{4} # Год
-         # Разделитель
\d{2}     # Месяц
-         # Разделитель
\d{2}     # День
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Если нет возможности включить режим свободного форматирования за пределами регулярного выражения, можно в самое начало регулярного выражения добавить модификатор ‹(?x)›. Обратите внимание, что перед модификатором ‹(?x)› не должно быть пробелов. Режим свободного форматирования включается только начиная с позиции этого модификатора – любые пробелы перед ним имеют значение.

Подробнее о модификаторах рассказывается в рецепте 2.1 в подразделе «Поиск без учета регистра символов» на стр. 51.

2.19. Вставка текстового литерала в замещающий текст

Задача

Для любого регулярного выражения, выполняющего поиск с заменой совпадения, подготовить замещающий текст со следующими символами: $%*$1\1.

Решение

```
$%\*$$1\1
```

Диалекты замещающего текста: .NET, JavaScript

```
\$%\\*\$1\\1
Диалект замещающего текста: Java

$%\*\$1\\1
Диалект замещающего текста: PHP

\$%\*\$1\\1
Диалект замещающего текста: Perl

$%\*$1\\1
Диалекты замещающего текста: Python, Ruby
```

Обсуждение

Когда и как экранировать символы в замещающем тексте

Данный рецепт демонстрирует разные правила экранирования, используемые в разных диалектах замещающего текста. Единственные два символа, которые может потребоваться экранировать в замещающем тексте, – это знак доллара и символ обратного слэша. Символами экранирования также являются знак доллара и символ обратного слэша.

Знак процента и звездочка в этом примере всегда интерпретируются, как обычные литералы, однако предшествующий символ обратного слэша может интерпретироваться как экранирующий символ, а не как литерал. «$1» и/или «\1» являются обратными ссылками на сохраняющую группу. В рецепте 2.21 описывается, в каком диалекте какой синтаксис используется для оформления обратных ссылок.

Факт, что эта задача имеет пять решений для семи диалектов замещающего текста, демонстрирует отсутствие стандартного синтаксиса для замещающего текста.

.NET и JavaScript

Диалекты .NET и JavaScript всегда интерпретируют символ обратного слэша как литерал. Его не требуется экранировать другим символом обратного слэша, в противном случае замещающий текст будет содержать два символа обратного слэша.

Одиночный знак доллара рассматривается как литерал. Знак доллара требуется экранировать, только если за ним следуют цифра, амперсанд, обратный апостроф, прямая кавычка, символ подчеркивания, знак плюс или другой знак доллара. Чтобы экранировать знак доллара, перед ним нужно поместить еще один знак доллара.

Все знаки доллара можно продублировать, если вам кажется, что это сделает замещающий текст более удобочитаемым. Следующее решение совершенно идентично тому, что приводится выше.

```
$$%\*$$1\1
```

Диалект замещающего текста: .NET, JavaScript

Кроме того, в диалекте .NET необходимо экранировать знак доллара, предшествующий открывающей фигурной скобке. В диалекте .NET последовательность «${group}» интерпретируется как именованная обратная ссылка. Диалект JavaScript не поддерживает именованные обратные ссылки.

Java

В диалекте Java для экранирования обратного слэша и знака доллара в замещающем тексте используется символ обратного слэша. Все литералы обратного слэша и литералы знака доллара должны экранироваться символом обратного слэша. Если этого не сделать, будет возбуждено исключение.

PHP

PHP требует, чтобы все цифры и знаки доллара, предшествующие цифре или открывающей фигурной скобке, экранировались символом обратного слэша.

Обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа слэша, их следует записать как «\\\\». Все остальные символы обратного слэша интерпретируются как литералы.

Perl

Диалект Perl несколько отличается от других диалектов замещающего текста: в действительности в нем нет диалекта замещающего текста. Тогда как в других языках программирования в процедурах поиска с заменой присутствует специальная логика, выполняющая подстановку для таких конструкций, как «$1», в языке Perl это обычная интерпретация переменной. В замещающем тексте необходимо экранировать символом обратного слэша все литералы знака доллара, как если бы они присутствовали в строке, ограниченной кавычками.

Единственное исключение состоит в том, что в языке Perl поддерживается синтаксис «\1» оформления обратных ссылок. Поэтому необходимо экранировать символ обратного слэша, за которым следует цифра, если обратный слэш должен интерпретироваться как литерал. Обратный слэш, за которым следует знак доллара, также должен экранироваться, чтобы предотвратить экранирование знака доллара.

Обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа слэша, их следует записать как «\\\\». Все остальные символы обратного слэша интерпретируются как литералы.

Python и Ruby

В диалектах Python и Ruby знак доллара не имеет специального значения в замещающем тексте. Символы обратного слэша должны экранироваться другими обратными слэшами, если за ними следует символ, придающий обратному слэшу специальное значение.

В диалекте Python конструкции с «`\1`» по «`\9`» и «`\g<`» образуют обратные ссылки. Эти символы обратного слэша необходимо экранировать.

В диалекте Ruby необходимо экранировать символ обратного слэша, за которым следует цифра, амперсанд, обратный апостроф, прямая кавычка или знак плюс.

Кроме того, в обоих языках обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа слэша, их следует записать как «`\\\\`». Все остальные символы обратного слэша интерпретируются как литералы.

Дополнительные правила экранирования для строковых литералов

Не забывайте, что в этой главе мы имеем дело только с регулярными выражениями и замещающим текстом как таковыми. Языки программирования и строковые литералы будут рассматриваться в следующей главе.

Любой замещающий текст из продемонстрированных выше сможет применяться, только когда функции `replace()` передается фактическая строковая переменная, хранящая этот текст. Другими словами, если в приложении имеется поле ввода, куда пользователь вводит замещающий текст, эти решения демонстрируют, что́ должен вводить пользователь, чтобы функция поиска с заменой работала, как ожидается. Если проверить команды поиска с заменой в RegexBuddy или в другой программе проверки регулярных выражений, этот рецепт продемонстрирует ожидаемые результаты.

Но те же замещающие тексты не будут работать, если вставить их непосредственно в исходный текст программы и окружить кавычками. Строковые литералы в языках программирования определяют собственные правила экранирования и этим правилам необходимо следовать, применяя их поверх правил экранирования, определяемых диалектом замещающего текста. На практике в конечном итоге может получиться частокол из символов обратного слэша.

См. также

Рецепт 3.14.

2.20. Вставка совпадения с регулярным выражением в замещающий текст

Задача

Выполнить поиск с заменой, в процессе которого все адреса URL будут преобразованы в ссылки HTML, указывающие на эти адреса, и использовать обнаруженные адреса URL как замещающий текст. Для данного упражнения примем, что адреса URL начинаются с последовательности «http:», за которой следуют любые непробельные символы. Например, текст `Please visit http://www.regexcookbook.com` должен превратиться в текст `Please visit <a href="http://www.regexcookbook.com">http://www.regexcookbook.com</a>`.

Решение

Регулярное выражение

```
http:\S+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
<a•href="$&">$&</a>
```

Диалекты замещающего текста: .NET, JavaScript, Perl

```
<a•href="$0">$0</a>
```

Диалекты замещающего текста: .NET, Java, PHP

```
<a•href="\0">\0</a>
```

Диалекты замещающего текста: PHP, Ruby

```
<a•href="\&">\&</a>
```

Диалект замещающего текста: Ruby

```
<a•href="\g<0>">\g<0></a>
```

Диалект замещающего текста: Python

Обсуждение

Вставка всего совпадения с регулярным выражением обратно в замещающий текст обеспечивает простой способ вставки нового текста перед, после или вокруг совпавшего текста или даже между несколькими копиями совпавшего текста. Чтобы использовать совпадение со всем регулярным выражением, не потребуется добавлять какие-либо сохраняющие группы в регулярное выражение, за исключением диалекта Python.

В языке Perl комбинация «$&» фактически является переменной. В языке Perl в случае успеха в этой переменной сохраняется общее совпадение с регулярным выражением.

Диалекты .NET и JavaScript поддерживают синтаксис «$&» для вставки совпадения с регулярным выражением в замещающий текст. В диалекте Ruby в аналогичной конструкции вместо знака доллара используется символ обратного слэша, поэтому общее совпадение может быть вставлено с помощью последовательности «\&».

В диалектах Java, PHP и Python отсутствует специальный элемент для вставки совпадения со всем регулярным выражением, но они позволяют вставлять в замещающий текст совпадение с сохраняющей группой, как описывается в следующем разделе. Все регулярное выражение в них рассматривается как неявная сохраняющая группа с номером 0. Чтобы сослаться на нулевую сохраняющую группу в языке Python, необходимо использовать синтаксис именованных обратных ссылок. Python не поддерживает синтаксис «\0».

Диалекты .NET и Ruby также поддерживают синтаксис обращения к нулевой сохраняющей группе, но в них совершенно не важно, какой синтаксис используется. Результат будет тем же самым.

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1 и рецепт 3.15.

2.21. Вставка части совпадения с регулярным выражением в замещающий текст

Задача

Найти совпадение с любой непрерывной последовательностью из 10 цифр, например 1234567890. Преобразовать эту последовательность в формат представления телефонных номеров, например (123) 456-7890.

Решение

Регулярное выражение

```
\b(\d{3})(\d{3})(\d{4})\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
($1)•$2-$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl

```
(${1})•${2}-${3}
```
Диалекты замещающего текста: .NET, PHP, Perl

```
(\1)•\2-\3
```
Диалекты замещающего текста: PHP, Python, Ruby

Обсуждение

Замещение с использованием сохраняющих групп

В рецепте 2.10 описывалось, как использовать сохраняющие группы в регулярном выражении, чтобы получить более одного совпадения с одним и тем же текстом. Текст, совпавший с каждой из сохраняющих групп, также будет доступен после каждого успешного совпадения. Текст из некоторых или всех сохраняющих групп можно вставлять в любом порядке, и даже несколько раз, в замещающий текст.

Некоторые диалекты, такие как Python и Ruby, используют одинаковый синтаксис «\1» обратных ссылок как в регулярных выражениях, так и в замещающем тексте. Другие диалекты используют синтаксис Perl «$1», где вместо обратного слэша применяется знак доллара. PHP поддерживает оба варианта.

В языке Perl ссылки «$1» и с иными цифрами фактически являются переменными, значение которых устанавливается после каждого успешного совпадения регулярного выражения. Их можно использовать в любом месте программного кода, пока не будет предпринята новая попытка сопоставления регулярного выражения. Диалекты .NET, Java, JavaScript и PHP поддерживают синтаксис «$1» только в замещающем тексте. Для доступа к содержимому сохраняющих групп в программном коде на этих языках программирования предоставляются иные способы. Более подробно об этом рассказывается в главе 3.

$10 и выше

Все диалекты регулярных выражений, рассматриваемые в этой книге, поддерживают до 99 сохраняющих групп в регулярных выражениях. В замещающем тексте могут возникать неоднозначность интерпретации обратных ссылок для «$10» или «\10» и выше. Они могут интерпретироваться и как обратная ссылка на группу с номером 10, и как обратная ссылка на первую группу, за которой следует литерал «0».

Диалекты .NET, PHP и Perl позволяют помещать номер сохраняющей группы в фигурные кавычки, устраняя неоднозначность. Последовательность «${10}» всегда обозначает обратную ссылку на сохраняющую группу с номером 10, а последовательность «${1}0» всегда обозначает обратную ссылку на первую сохраняющую группу, за которой следует литерал «0».

Диалекты Java и JavaScript пытаются правильно интерпретировать последовательность «$10». Если в регулярном выражении существует сохраняющая группа с указанным двузначным номером, такая последо-

вательность будет интерпретироваться, как обратная ссылка на сохраняющую группу. Если число сохраняющих групп меньше, только первая цифра будет использоваться как ссылка на группу, а ноль будет интерпретироваться, как литерал. То есть последовательность «$23» будет означать ссылку на сохраняющую группу с номером 23, только если таковая существует в регулярном выражении. В противном случае она будет обозначать ссылку на вторую сохраняющую группу, за которой следует литерал «3».

Диалекты .NET, PHP, Perl, Python и Ruby всегда интерпретируют последовательность «$10» или «\10» как ссылку на сохраняющую группу с номером 10 независимо от ее существования. Если такой группы не существует, вступают в действия правила, применяемые к ссылкам на несуществующие группы.

Ссылки на несуществующие группы

Регулярное выражение, представленное в решении для этого рецепта, содержит три сохраняющих группы. Если в замещающем тексте указать последовательность «$4» или «\4», она будет интерпретироваться как ссылка на несуществующую сохраняющую группу. Это вызовет три варианта действий.

Java и Python завопят о нарушении, возбудив исключение или вернув сообщение об ошибке. В этих диалектах не следует использовать ошибочные обратные ссылки. (В действительности ошибочные обратные ссылки не следует использовать ни в одном из диалектов.) Если необходимо, чтобы последовательность «$4» или «\4» интерпретировалась буквально, необходимо экранировать знак доллара или символ обратного слэша. Подробнее об этом рассказывается в рецепте 2.19.

PHP, Perl и Ruby подставляют в замещающий текст все ссылки, включая и те, что указывают на несуществующие сохраняющие группы. Несуществующие группы ничего не сохраняют, и потому ссылки, указывающие на такие группы, замещаются пустым текстом.

Наконец, .NET и JavaScript интерпретируют обратные ссылки на несуществующие, как литеральный текст.

Все диалекты выполняют замещение ссылок на сохраняющие группы, существующие в регулярном выражении, но не сохранившие ничего. Такие ссылки замещаются пустым текстом.

Решение, использующее именованные сохранения

Регулярное выражение

```
\b(?<area>\d{3})(?<exchange>\d{3})(?<number>\d{4})\b
```

Параметры: нет

Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'area'\d{3})(?'exchange'\d{3})(?'number'\d{4})\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<area>\d{3})(?P<exchange>\d{3})(?P<number>\d{4})\b
```

Параметры: нет
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

Замещающий текст

```
(${area})•${exchange}-${number}
```

Диалект замещающего текста: .NET

```
(\g<area>)•\g<exchange>-\g<number>
```

Диалект замещающего текста: Python

```
(\k<area>)•\k<exchange>-\k<number>
```

Диалект замещающего текста: Ruby 1.9

```
(\k'area')•\k'exchange'-\k'number'
```

Диалект замещающего текста: Ruby 1.9

```
($1)•$2-$3
```

Диалекты замещающего текста: .NET, PHP, Perl 5.10

```
(${1})•${2}-${3}
```

Диалекты замещающего текста: .NET, PHP, Perl 5.10

```
(\1)•\2-\3
```

Диалекты замещающего текста: PHP, Python, Ruby 1.9

Диалекты, поддерживающие именованные сохранения

Диалекты .NET, Python и Ruby 1.9 позволяют использовать в замещающем тексте именованные обратные ссылки, если в регулярном выражении используются именованные сохраняющие группы.

В диалектах .NET и Python синтаксис именованных обратных ссылок для работы с именованными сохраняющими группами похож на синтаксис обратных ссылок, указывающих на нумерованные сохраняющие группы. Просто в фигурных или угловых скобках указывается имя или номер группы.

В диалекте замещающего текста Ruby используется тот же синтаксис обратных ссылок, что и в регулярных выражениях. Для именованных групп в Ruby 1.9 синтаксис имеет вид: «`\k<group>`» или «`\k'group'`». Выбор между угловыми скобками и апострофами – это лишь вопрос личных предпочтений.

Perl 5.10 и PHP (при использовании библиотеки PCRE) поддерживают именованные сохраняющие группы в регулярных выражениях, но не в замещающем тексте. Чтобы обратиться к именованным группам, присутствующим в регулярном выражении, в замещающем тексте можно использовать нумерованные обратные ссылки. Диалекты Perl 5.10 и PCRE присваивают номера именованным и нумерованным группам в направлении слева направо.

Диалекты .NET, Python и Ruby 1.9 также позволяют использовать нумерованные ссылки на именованные группы. Однако для именованных групп в диалекте .NET используется иная схема нумерации, как описывается в рецепте 2.11. Смешивание именованных и нумерованных групп в диалектах .NET, Python и Ruby не рекомендуется. Желательно, чтобы либо все группы были именованными, либо все – нумерованными. Для обращения к именованным группам всегда следует использовать именованные обратные ссылки.

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1 и рецепты 2.9, 2.10, 2.11 и 3.15.

2.22. Вставка контекста совпадения в замещающий текст

Задача

Создать замещающий текст, который заместит совпадение с регулярным выражением текстом, расположенным перед совпадением, всем испытуемым текстом и остатком испытуемого текста, расположенным после совпадения. Например, если в тексте `BeforeMatchAfter` был найден фрагмент `Match`, его следует заменить текстом `BeforeBeforeMatchAfterAfter`, в результате должен получиться текст `BeforeBeforeBeforeMatchAfterAfterAfter`.

Решение

```
$`$_$'
```

Диалекты замещающего текста: .NET, Perl

```
\`\`\&\'\'
```

Диалект замещающего текста: Ruby

```
$`$`$&$'$'
```

Диалект замещающего текста: JavaScript

Обсуждение

Под термином *контекст* подразумевается испытуемый текст, к которому применялось регулярное выражение. Контекст делится на три части: испытуемый текст перед совпадением с регулярным выражением, испытуемый текст после совпадения и весь испытуемый текст. Текст, расположенный перед совпадением, иногда называют *левым контекстом*, а текст, расположенный после совпадения, соответственно, *правым контекстом*. Весь испытуемый текст образуется левым контекстом, совпадением и правым контекстом.

Диалекты .NET и Perl поддерживают конструкции «$`», «$'» и «$_», которые вставляют все три составляющие контекста в замещающий текст. Фактически в языке Perl эти конструкции являются переменными, значения которых устанавливаются после успешного совпадения с регулярным выражением, и доступными в любом месте в программном коде, пока не будет выполнена новая попытка сопоставления. Комбинация «$`» обозначает левый контекст. Символ обратного апострофа вводится с клавиатуры в раскладке U.S. нажатием на клавишу, расположенную левее клавиши с цифрой 1, в левом верхнем углу клавиатуры. Комбинация «$'» обозначает правый контекст. В качестве прямой кавычки здесь используется обычный апостроф. На клавиатуре, в раскладке U.S., клавиша с этим символом находится между клавишей с точкой с запятой и клавишей Enter. Комбинация «$_» обозначает весь испытуемый текст. Подобно диалектам .NET и Perl диалект JavaScript использует комбинации «$`» и «$'» для обозначения левого и правого контекстов. Однако в JavaScript отсутствует элемент, соответствующий всему испытуемому тексту. При работе с этим диалектом можно восстановить весь испытуемый текст, вставив текст совпадения с помощью конструкции «$&» между левым и правым контекстами.

В диалекте Ruby доступ к левому и правому контекстам поддерживаются конструкциями «\`» и «\'», а текст совпадения с регулярным выражением вставляется конструкцией «\&». Как и в JavaScript, здесь нет элемента, с помощью которого можно было бы вставить весь испытуемый текст.

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1 и рецепт 3.15.

3

Программирование с применением регулярных выражений

Языки программирования и диалекты регулярных выражений

В этой главе описывается, как реализовать регулярные выражения в различных языках программирования. рецепты в этой главе предполагают, что в вашем распоряжении уже имеется готовое регулярное выражение; в этом вы можете опираться на предыдущую главу. Теперь перед вами стоит задача поместить регулярное выражение в исходный текст программы и заставить его работать.

В этой главе мы приложили максимум усилий, чтобы объяснить, как и почему каждый фрагмент программного кода работает тем или иным способом. Высокий уровень детализации может сделать чтение этой главы от начала до конца достаточно утомительным занятием. Если вы читаете книгу «Регулярные выражения, сборник рецептов» в первый раз, мы рекомендуем бегло просмотреть эту главу, чтобы получить общее представление о том, что может потребоваться. Позднее, когда появится необходимость реализовать какое-либо из регулярных выражений, рассматриваемых в последующих главах, можно будет вернуться сюда и подробнее узнать, как интегрировать регулярные выражения в программу, написанную на том или ином языке программирования.

В главах с 4 по 8 демонстрируются регулярные выражения, предназначенные для решения практических задач. Эти главы основное внимание уделяют самим регулярным выражениям и во многих рецептах программный код вообще не демонстрируется. Чтобы заставить какое-то регулярное выражение, приведенное в этих главах, работать, просто нужно включить его в один из фрагментов программного кода, которые демонстрируются в этой главе.

Поскольку последующие главы основное внимание уделяют именно регулярным выражениям, они представляют решения для определенных диалектов регулярных выражений, а не для языков программирования. Между диалектами регулярных выражений и языками программирования нет точного соответствия. Языки сценариев обычно используют собственный встроенный диалект регулярных выражений, другие языки программирования опираются на библиотеки, реализующие поддержку регулярных выражений. Некоторые библиотеки могут использоваться в нескольких языках программирования, а в некоторых языках программирования имеется возможность использовать разные библиотеки. В разделе «Множество диалектов регулярных выражений» на стр. 21 описываются все диалекты регулярных выражений, рассматриваемые в этой книге. В разделе «Множество диалектов определения замещающего текста» на стр. 25 перечислены диалекты определения замещающего текста, используемого в операциях поиска с заменой, с применением регулярных выражений. Все языки программирования, описываемые в этой главе, используют один из этих диалектов.

Языки, рассматриваемые в этой главе

В этой главе рассматриваются восемь языков программирования. Каждый рецепт имеет отдельные решения для всех языков, а во многих рецептах также даются отдельные пояснения для всех восьми языков. Если применяемый прием относится сразу к нескольким языкам, мы повторим его в пояснениях для каждого из этих языков. Вы без всякого ущерба для себя можете пропускать пояснения для языков программирования, которые вас не интересуют.

C#

Язык программирования C# используется платформой Microsoft .NET. Классы `System.Text.RegularExpressions` используют диалект регулярных выражений и диалект замещающего текста «.NET». В этой книге охватываются версии C# с 1.0 по 3.5 или Visual Studio с версии 2002 по 2008.

VB.NET

Под названиями VB.NET и Visual Basic.NET, используемыми в этой книге, подразумевается Visual Basic 2002 и более поздние версии, чтобы обозначить отличие этих версий от Visual Basic 6 и более ранних. В настоящее время Visual Basic использует платформу Microsoft .NET. Классы `System.Text.RegularExpressions` используют диалект регулярных выражений и диалект замещающего текста «.NET». В этой книге охватываются версии Visual Basic с 2002 по 2008.

Java

Версия Java 4 стала первой версией, обеспечившей встроенную поддержку регулярных выражений в виде пакета `java.util.regex`. Па-

кет `java.util.regex` использует диалект регулярных выражений и диалект замещающего текста «Java». В этой книге охватываются версии Java 4, 5 и 6.

JavaScript

Этот диалект регулярных выражений используется в языке программирования, который наиболее широко известен под названием JavaScript. Все современные веб-броузеры реализуют поддержку этого языка программирования: Internet Explorer (начиная с версии 5.5), Firefox, Opera, Safari и Chrome. Многие приложения также используют JavaScript в качестве языка сценариев.

Строго говоря, в этой книге под названием *JavaScript* подразумевается язык программирования, определяемый третьей версией стандарта ECMA-262. Этот стандарт дает определение языка программирования ECMAScript, который более известен в различных веб-броузерах по его реализациям JavaScript и JScript.

Кроме того, стандарт ECMA-262v3 дает определение диалектов регулярных выражений и замещающего текста, используемых в языке JavaScript. В данной книге эти диалекты обозначаются названием «JavaScript».

PHP

В языке PHP имеется три набора функций, предназначенных для работы с регулярными выражениями. Мы настоятельно рекомендуем использовать семейство функций `preg`. В этой книге рассматриваются только функции семейства `preg`, которые были встроены в язык PHP, начиная с версии 4.2.0. В этой книге охватываются версии PHP 4 и 5. Функции `preg` в языке PHP являются обертками вокруг функций библиотеки PCRE. Диалект регулярных выражений PCRE в этой книге обозначается «PCRE». Поскольку в библиотеке PCRE отсутствуют функции, реализующие поиск с заменой, разработчиками языка PHP был выработан собственный синтаксис определения замещающего текста для функции `preg_replace()`. Данный диалект замещающего текста в этой книге обозначается «PHP».

Семейство функций `mb_ereg` является составной частью семейства «многобайтовых» функций языка PHP, предназначенных для работы с текстом на языках, для которых традиционно используется многобайтовое представление символов, таких как японский и китайский. В версии PHP 5 функции семейства `mb_ereg` используют библиотеку поддержки регулярных выражений Onigurama, которая изначально предназначалась для языка программирования Ruby. Диалект регулярных выражений Onigurama в этой книге обозначается как «Ruby 1.9». Использовать функции семейства `mb_ereg` рекомендуется только в случае необходимости работы с многобайтовыми представлениями символов при условии, что вы уже знакомы с функциями `mb_` языка PHP.

Семейство функций `ereg` является старейшим семейством функций поддержки регулярных выражений в языке PHP и официально не рекомендуется к использованию, начиная с версии PHP 5.3.0. Эти функции не зависят от внешних библиотек и реализуют диалект POSIX ERE. Данный диалект поддерживает лишь ограниченный набор возможностей и в этой книге не рассматривается. Диалект POSIX ERE – это ограниченное подмножество диалектов Ruby 1.9 и PCRE. Регулярные выражения, применяемые в вызовах функций `ereg`, можно использовать в функциях `mb_ereg` и `preg`. При этом в случае функций `preg` придется добавить разделители в стиле языка Perl (рецепт 3.1).

Perl

То, что язык Perl имеет встроенную поддержку регулярных выражений, является главной причиной их популярности в наши дни. Диалекты регулярных выражений и замещающего текста, используемые операторами `m//` и `s///` языка Perl, в этой книге обозначаются названием «Perl». В этой книге рассматриваются версии Perl 5.6, 5.8 и 5.10.

Python

Язык программирования Python обеспечивает поддержку регулярных выражений посредством модуля `re`. Диалекты регулярных выражений и замещающего текста, используемые этим модулем, в этой книге обозначаются как «Python». В этой книге рассматриваются версии Python 2.4 и 2.5.

Ruby

Поддержка регулярных выражений в языке Ruby реализована как часть самого языка. В этой книге рассматриваются версии Ruby 1.8 и 1.9. В этих двух версиях Ruby по умолчанию используются разные реализации механизма регулярных выражений. В версии Ruby 1.9 по умолчанию используется механизм Onigurama, обладающий более широкими возможностями, чем классический механизм в версии Ruby 1.8. Более подробно об этом рассказывается в разделе «Диалекты регулярных выражений, рассматриваемые в этой книге», на стр. 22.

В этой книге мы не будем углубляться в различия между версиями Ruby 1.8 и 1.9. В этой книге демонстрируются лишь самые простые регулярные выражения, в которых не используются новые особенности версии Ruby 1.9. Поскольку поддержка регулярных выражений включена непосредственно в язык Ruby, программный код на этом языке, использующий регулярные выражения, остается неизменным независимо от того, скомпилирован ли Ruby с поддержкой классического механизма регулярных выражений или механизма Onigurama. В случае необходимости версия Ruby 1.8 может быть скомпилирована с поддержкой механизма Onigurama.

Другие языки программирования

Языки программирования, перечисленные ниже, не рассматриваются в этой книге, но они используют тот или иной диалект регулярных выражений из описываемых здесь. Если вы используете один из этих языков, можете просто пропустить эту главу, при этом другие главы попрежнему будут для вас полезны.

ActionScript

Язык программирования ActionScript – это реализация стандарта ECMA-262, выполненная компанией Adobe. Начиная с версии 3.0, язык ActionScript полностью поддерживает регулярные выражения стандарта ECMA-262v3. Данный диалект регулярных выражений в этой книге обозначается как «JavaScript». Кроме того, язык ActionScript очень близок к языку JavaScript. Примеры из этой книги, которые приводятся для языка JavaScript, легко могут быть адаптированы под язык ActionScript.

C

В программах на языке C можно использовать широкий круг библиотек поддержки регулярных выражений. Из диалектов, рассматриваемых в книге, наилучшим, пожалуй, выбором будет библиотека PCRE, распространяемая с открытыми исходными текстами. Загрузить исходные тексты библиотеки на языке C можно на сайте *http://www.pcre.org*. Программный код библиотеки написан так, что позволяет компилировать его с помощью самых разных компиляторов на самых разных платформах.

C++

В программах на языке C++ можно использовать широкий круг библиотек поддержки регулярных выражений. Из диалектов, рассматриваемых в книге, наилучшим, пожалуй, выбором будет библиотека PCRE, распространяемая с открытыми исходными текстами. В программах можно напрямую использовать C API библиотеки или прибегнуть к помощи классов-оберток на языке C++, включенных в состав PCRE и доступных для загрузки (подробности смотрите на сайте *http://www.pcre.org*).

Delphi for Win32

К моменту написания этих строк версия Delphi для платформы Win32 не имела встроенной поддержки регулярных выражений. Однако имеется множество компонентов VCL, обеспечивающих эту поддержку. Я рекомендую выбирать компоненты, опирающиеся на использование PCRE. В Delphi имеется возможность связывать объектные файлы на языке C с приложением, и многие компоненты VCL, являющиеся обертками для библиотеки PCRE, используют эти файлы. Эта возможность позволяет собрать программу как единый файл *.exe*.

Загрузить мой собственный компонент TPerlRegEx можно по адресу *http://www.regexp.info/delphi.html*. Это компонент VCL, который устанавливается непосредственно в палитру компонентов, благодаря чему его можно просто бросать на форму. Еще одной популярной оберткой вокруг библиотеки PCRE для Delphi является класс TJclRegEx, входящий в состав библиотеки JCL (*http://www.delphi-jedi.org*). Класс TJclRegEx наследует класс TObject, поэтому его нельзя просто бросить на форму.

Обе библиотеки распространяются с открытыми исходными текстами на условиях лицензии Mozilla Public License.

Delphi Prism

В Delphi Prism можно использовать поддержку регулярных выражений, предоставляемую платформой .NET. Для этого достаточно добавить модуль System.Text.RegularExpressions в предложение uses в любом модуле Delphi Prism, где предполагается использовать регулярные выражения.

После этого можно будет использовать те же приемы, которые демонстрируются в этой главе, во фрагментах программного кода на языках C# и VB.NET.

Groovy

В программах на языке Groovy можно использовать регулярные выражения, поддержка которых обеспечивается пакетом java.util.regex, как в языке Java. В действительности все решения, представленные в этой главе для языка Java, также должны работать и в языке Groovy. Собственный синтаксис регулярных выражений в языке Groovy просто обеспечивает сокращенную форму записи. Литерал регулярного выражения, части которого отделяются друг от друга символами слэша, является экземпляром класса java.lang.String, а оператор =~ создает экземпляр класса java.util.regex.Matcher. Вы свободно можете смешивать синтаксис языка Groovy со стандартным синтаксисом языка Java – классы и объекты при этом остаются теми же самыми.

PowerShell

Язык PowerShell – это язык сценариев, разработанный компанией Microsoft, опирающийся на платформу .NET. Операторы -match и -replace, встроенные в PowerShell, используют диалект регулярных выражений и замещающего текста .NET, который описывается в этой книге.

R

Проект R поддерживает регулярные выражения с помощью функций grep, sub и regexpr в пакете base. Все эти функции принимают необязательный параметр с именем perl, который получает значение FALSE, если он опущен. Если передать в нем значение TRUE, будет ис-

пользоваться диалект PCRE, описываемый в этой книге. Регулярные выражения, демонстрируемые для версии PCRE 7, будут работать в версии R 2.5.0 и выше. В более ранних версиях R следует использовать регулярные выражения, помеченные в этой книге как «PCRE 4 и выше». Диалекты «базовый» и «расширенный», поддерживаемые языком R, являются устаревшими диалектами регулярных выражений с ограниченными возможностями и не рассматриваются в этой книге.

REALbasic

Язык REALbasic имеет встроенный класс `RegEx`. Реализация этого класса опирается на версию библиотеки PCRE с поддержкой кодировки UTF-8. Это означает, что имеется возможность задействовать поддержку Юникода в библиотеке PCRE, но при этом придется использовать класс `TextConverter` языка REALbasic для преобразования текста в кодировку UTF-8, прежде чем передавать его классу `RegEx`.

Все регулярные выражения, демонстрируемые в этой книге для версии PCRE 6, будут работать и в программах на языке REALbasic. Единственная особенность REALbasic состоит в том, что параметры «нечувствительность к регистру символов» и «символам ^ и $ соответствуют границы строк» («многострочный» режим) включены по умолчанию. Если появится необходимость использовать регулярные выражения из этой книги, для которых явно не говорится, что они используют эти режимы сопоставления, их необходимо будет явно отключить в программе на языке REALbasic.

Scala

Язык Scala обладает встроенной поддержкой регулярных выражений посредством модуля `scala.util.matching`. Эта поддержка построена на основе механизма регулярных выражений пакета `java.util.regex` языка Java. Диалекты регулярных выражений и замещающего текста, используемые в языках Java и Scala, обозначаются в этой книге как «Java».

Visual Basic 6

Версия Visual Basic 6 – это последняя версия Visual Basic, которая не требует наличия платформы .NET. Это также означает, что Visual Basic 6 не использует превосходную поддержку регулярных выражений, реализованную в платформе .NET. Примеры программного кода на языке VB.NET, которые приводятся в этой книге, вообще не будут работать в VB 6.

Visual Basic 6 обеспечивает простоту использования функциональных возможностей, предоставляемых библиотеками ActiveX и COM. Одной из таких библиотек является библиотека Microsoft VBScript, которая, начиная с версии 5.5, обладает неплохой поддержкой регулярных выражений. Библиотека реализует тот же диалект регу-

лярных выражений, что используется в языке JavaScript, описываемый стандартом ECMA-262v3. Данная библиотека является частью Internet Explorer 5.5 и выше. Она доступна на любом компьютере, работающем под управлением операционной системы Windows XP или Vista, а также в предыдущих версиях Windows, при условии, что там установлена версия Internet Explorer 5.5 или выше. В эту категорию входят практически все персональные компьютеры, работающие под управлением Windows и используемые для подключения к Интернету.

Чтобы использовать эту библиотеку в приложении на языке Visual Basic, нужно выбрать пункт меню Project|References в среде разработки VB. Прокрутить список и отыскать в нем пункт «Microsoft VBScript Regular Expressions 5.5», расположенный сразу же под пунктом «Microsoft VBScript Regular Expressions 1.0». Убедитесь, что выбрали версию 5.5, а не 1.0. Версия 1.0 предоставляется только для сохранения обратной совместимости, а ее возможности менее чем удовлетворительны.

После добавления ссылки можно будет увидеть, какие классы предоставляются библиотекой и какие члены входят в состав этих классов. Для этого нужно выбрать пункт меню View|Object Browser. В окне Object Browser и в раскрывающемся списке в левом верхнем углу нужно выбрать библиотеку «VBScript_RegExp_55».

3.1. Литералы регулярных выражений в исходных текстах

Задача

Имеется регулярное выражение ‹[$"'\n\d/\\]›, являющееся решением некоторой задачи. Это регулярное выражение представляет собой символьный класс, которому соответствуют знак доллара, кавычка, апостроф, символ перевода строки, любая цифра в диапазоне от 0 до 9, символ слэша или символ обратного слэша. Необходимо вставить это регулярное выражение в исходный текст программы в виде строковой константы или оператора регулярного выражения.

Решение

C#

В виде обычной строки:

```
"[$\"'\n\\d/\\\\]"
```

В виде строкового литерала:

```
@"[$""'\n\d/\\]"
```

VB.NET

```
“[$””’\n\d/\\]”
```

Java

```
“[$\”’\n\\d/\\\\]”
```

JavaScript

```
/[$”’\n\d\/\\]/
```

PHP

```
‘%[$”\’\n\d/\\\\]%’
```

Perl

Оператор сопоставления с шаблоном:

```
/[\$”’\n\d\/\\]/
m![\$”’\n\d/\\]!
```

Оператор подстановки:

```
s![\$”’\n\d/\\]!!
```

Python

Строка в тройных кавычках:

```
r”””[$”’\n\d/\\]”””
```

Обычная строка:

```
“[$\”’\n\\d/\\\\]”
```

Ruby

Литерал регулярного выражения, разделенный символами слэша:

```
/[$”’\n\d\/\\]/
```

Литерал регулярного выражения, разделенный знаками пунктуации, выбираемыми пользователем:

```
%r![$”’\n\d/\\]!
```

Обсуждение

Когда в этой книге демонстрируется регулярное выражение само по себе (в противоположность регулярному выражению, входящему в состав программного кода), оно всегда приводится в чистом виде. Данный рецепт является единственным исключением. При использовании программы тестирования регулярных выражений, такой как RegexBuddy

или RegexPal, регулярное выражение можно было бы ввести в таком виде. Если некоторое приложение принимает регулярные выражения, вводимые пользователем, пользователь должен был бы вводить его в таком виде.

Но если необходимо поместить регулярное выражение в исходный текст программы, придется выполнить дополнительные действия. Небрежность, проявленная при копировании регулярного выражения из программы тестирования регулярных выражений в исходный текст вашей программы или обратно, часто может заставить поломать голову над тем, почему оно работает в программе тестирования, но не работает в исходном тексте программы. Или почему программа тестирования сообщает об ошибке в регулярном выражении, которое было скопировано из чьего-то программного кода. Все языки программирования, рассматриваемые в этой книге, требуют, чтобы литерал регулярного выражения был разграничен определенным способом. В одних языках регулярные выражения должны быть оформлены как строки, а в других – как специальные константы регулярных выражений. Если регулярное выражение содержит символы, которые являются разделителями или имеют специальное значение в конкретном языке программирования, их необходимо экранировать.

Наиболее часто в качестве экранирующего символа используется символ обратного слэша. Именно по этой причине большинство решений этой проблемы включают гораздо большее число обратных слэшей по сравнению с оригинальными регулярными выражениями.

C#

В языке C# литерал регулярного выражения может передаваться конструктору `Regex()` и различным функциям-членам класса `Regex`. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

Язык C# поддерживает два типа строковых литералов. Наиболее часто используются строки в кавычках, хорошо известные по таким языкам программирования, как C++ и Java. Внутри таких строк кавычки и обратные слэши должны экранироваться символом обратного слэша. В строках также допускается указывать экранированные последовательности, обозначающие неотображаемые символы, такие как ‹\n›. При использовании свойства `RegexOptions.IgnorePatternWhitespace` (рецепт 3.4) для включения режима свободного форматирования, описываемого в рецепте 2.18, наблюдаются различия между “\n” и “\\n”. “\n” – это строка с символом перевода строки, который игнорируется как пробельный символ. “\\n” – это строка, включающая метасимвол ‹\n› регулярного выражения, совпадающий с символом перевода строки.

Строковые литералы начинаются со знака @ и кавычки и заканчиваются кавычкой. Чтобы включить кавычку в строковый литерал, ее необходимо продублировать. Символы обратного слэша не требуется экра-

нировать, в результате регулярное выражение получается гораздо более удобочитаемым. @"\n" – это всегда метасимвол ‹\n› регулярного выражения, который совпадает с символом перевода строки, даже в режиме свободного форматирования. Строковые литералы не поддерживают ‹\n› на уровне строки, зато они могут располагаться на нескольких строках. Это делает строковые литералы идеальными для представления регулярных выражений в режиме свободного форматирования.

Выбор очевиден: чтобы поместить регулярное выражение в исходный текст программы на языке C#, предпочтительнее использовать строковые литералы.

VB.NET

В языке VB.NET литерал регулярного выражения может передаваться конструктору Regex() и различным функциям-членам класса Regex. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

В Visual Basic используются строки в кавычках. Кавычки внутри строк должны дублироваться. Экранировать какие-либо другие символы в них не требуется.

Java

В языке Java литерал регулярного выражения может передаваться фабричному методу класса Pattern.compile() и различным функциям класса String. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

В Java используются строки в кавычках. Кавычки и символы обратного слэша внутри строк должны экранироваться символом обратного слэша. В строках также допускается указывать экранированные последовательности, обозначающие неотображаемые символы, такие как ‹\n›, и кодовые пункты Юникода, такие как ‹\uFFFF›.

При использовании свойства Pattern.COMMENTS (рецепт 3.4) для включения режима свободного форматирования, описываемого в рецепте 2.18, наблюдаются различия между "\n" и "\\n". "\n" – это строка с символом перевода строки, который игнорируется как пробельный символ. "\\n" – это строка, включающая метасимвол ‹\n› регулярного выражения, совпадающий с символом перевода строки.

JavaScript

В языке JavaScript регулярные выражения лучше всего создавать с использованием специального синтаксиса объявления литералов регулярных выражений. Для этого нужно просто поместить регулярное выражение между двумя символами слэша. Если символы слэша присутствуют в самом регулярном выражении, их следует экранировать символом обратного слэша.

Несмотря на то, что имеется возможность создавать объект `RegExp` из строки, тем не менее, нет смысла использовать строковую форму записи для представления регулярных выражений в программном коде. В этом случае потребовалось бы экранировать кавычки и символы обратного слэша, что обычно приводит к появлению частокола из обратных слэшей.

PHP

Синтаксис литералов регулярных выражений, используемый функциями `preg` в языке PHP, представляет собой довольно хитроумное изобретение. В отличие от JavaScript и Perl, в языке отсутствует специализированный тип регулярного выражения. Регулярные выражения всегда должны быть представлены в виде строк. То же самое относится к функциям `ereg` и `mb_ereg`. Но в своем стремлении к подражанию языку Perl разработчики функций-оберток для библиотеки PCRE в языке PHP добавили дополнительное требование.

Внутри строки регулярное выражение должно записываться как литерал регулярного выражения в языке Perl. Это означает, что там, где в языке Perl регулярное выражение записывается как `/regex/`, при передаче его функциям `preg` в языке PHP оно должно записываться как `'/regex/'`. Как и в языке Perl, в качестве разделителей допускается использовать любую пару знаков пунктуации. Если символ-разделитель присутствует внутри регулярного выражения, его необходимо экранировать символом обратного слэша. Чтобы избежать этого, следует выбирать символ-разделитель, отсутствующий внутри регулярного выражения. В этом рецепте я использовал знак процента, потому что символ слэша присутствует в регулярном выражении, а знак процента – нет. Если символ слэша отсутствует в регулярном выражении, можно использовать его, так как это наиболее часто используемый разделитель в языке Perl и обязательный разделитель в языках JavaScript и Ruby.

Язык PHP поддерживает строки в кавычках и в апострофах. В обоих случаях необходимо экранировать кавычки (или апострофы), а также обратные слэши, присутствующие в регулярном выражении, символом обратного слэша. При использовании строк в кавычках дополнительно требуется экранировать знак доллара. Для записи регулярных выражений предпочтительнее использовать строки в апострофах, если не требуется интерпретировать переменные внутри регулярных выражений.

Perl

В языке Perl литералы регулярных выражений используются в операторе сопоставления с шаблоном и в операторе подстановки. Оператор сопоставления с шаблоном содержит два символа слэша с регулярным выражением, заключенным между ними. Символы слэша внутри регулярного выражения должны экранироваться символами обратного слэ-

ша. Какие-либо другие символы экранировать не требуется, за исключением, возможно, символов $ и @, по причинам, описываемым в конце этого раздела.

Существует альтернативная форма записи оператора сопоставления с шаблоном, когда регулярное выражение помещается между парой любых других знаков пунктуации, в этом случае оператор должен начинаться с символа m. При использовании в качестве разделителей любых типов открывающих и закрывающих знаков пунктуации (круглые, квадратные или фигурные скобки), они должны соответствовать друг другу, например m{*regex*}. Если используется другой знак пунктуации, следует просто использовать его дважды. В решении к этому рецепту используется восклицательный знак. Это позволило сэкономить на экранировании символов слэша в регулярном выражении. Если в качестве разделителей используются знаки пунктуации, имеющие открывающую и закрывающую формы, тогда экранировать необходимо только закрывающий знак пунктуации, если он используется в регулярном выражении как литерал.

Оператор подстановки похож на оператор сопоставления с шаблоном. Только вместо символа m он начинается с символа s и принимает замещающий текст. При использовании квадратных скобок или подобных им знаков пунктуации необходимо использовать две пары: s[*regex*][*replace*]. Во всех остальных случаях знак пунктуации используется трижды: s/*regex*/*replace*/.

Операторы сопоставления с шаблоном и подстановки в языке Perl интерпретируются как строки в кавычках. Если регулярное выражение записано как m/I am $name/ и переменная $name хранит строку "Jan", в результате будет получено регулярное выражение ‹I•am•Jan›. Кроме того, в языке Perl $" – это переменная, поэтому потребовалось экранировать знак доллара в символьном классе регулярного выражения для этого рецепта.

Никогда не следует экранировать знак доллара, который используется как якорный метасимвол (рецепт 2.5). Экранированный знак доллара всегда интерпретируется как литерал. Язык Perl в состоянии отличать знаки доллара, используемые как якорные метасимволы, от знаков доллара, используемых для интерпретации переменных, имея в виду тот факт, что якорный метасимвол может использоваться только в конце группы, или в конце всего регулярного выражения, или перед символом перевода строки. Не требуется экранировать знак доллара в выражении ‹m/^*regex*$/›, если требуется проверить, соответствует ли регулярному выражению «regex» строка целиком.

Символ @ не имеет специального назначения в регулярных выражениях, но в языке Perl он используется для интерпретации переменных.

Его необходимо экранировать в литералах регулярных выражений, включенных в программный код на языке Perl, как если бы это были строки в кавычках.

Python

Функции модуля re в языке Python принимают литералы регулярных выражений в виде строк. Допускается использовать любые способы определения строк, которые допускаются языком Python. В зависимости от того, какие символы встречаются в регулярном выражении, различные формы записи строк могут уменьшить необходимость экранирования с помощью символов обратного слэша.

Обычно наилучшим выбором являются «сырые» (raw) строки. Сырые строки в языке Python вообще не требуют экранирования каких-либо символов. В случае использования сырых строк отпадает необходимость дублировать символы обратного слэша в регулярных выражениях. Строка r"\d+" читается проще, чем "\\d+", особенно если регулярное выражение длинное.

Единственный случай, когда сырые строки перестают быть лучшим выбором, – когда регулярное выражение включает в себя кавычки и апострофы. Невозможно использовать сырую строку, заключенную в пару апострофов или кавычек, так как нет никакой возможности экранировать кавычки внутри регулярного выражения. В такой ситуации могут помочь строки в тройных кавычках, как это сделано в решении для языка Python в данном рецепте. Обычная строка показана в решении для сравнения.

Если в регулярном выражении необходимо использовать возможности поддержки Юникода, описанные в рецепте 2.7, то надлежит использовать строки Юникода. Превратить обычную строку в строку Юникода можно, добавив перед открывающей кавычкой символ u.

Сырые строки не поддерживают возможность определения неотображаемых символов, таких как \n. Они интерпретируют экранированные последовательности как последовательность литералов. Это не является проблемой для модуля re. Он поддерживает эти экранированные последовательности как часть синтаксиса регулярных выражений, а также как часть синтаксиса замещающего текста. Литерал \n в сырой строке по-прежнему будет интерпретироваться как символ перевода строки в регулярных выражениях и в замещающем тексте.

При использовании атрибута re.VERBOSE (рецепт 3.4) для включения режима свободного форматирования, описываемого в рецепте 2.18, наблюдаются различия между строкой "\n" с одной стороны и строкой "\\n" и сырой строкой r"\n" – с другой. "\n" – это строка с символом перевода строки, который игнорируется как пробельный символ. "\\n" и r"\n" – это строки, включающие метасимвол ‹\n› регулярного выражения, совпадающий с символом перевода строки.

При использовании режима свободного форматирования сырые строки в тройных кавычках, такие как r"""\n""", являются лучшим выбором, потому что они могут располагаться на нескольких строках. Кро-

ме того, ‹\n› не интерпретируется на уровне строки, поэтому он может интерпретироваться на уровне регулярного выражения как метасимвол, совпадающий с границей строки.

Ruby

В языке Ruby регулярные выражения лучше всего создавать с использованием специального синтаксиса объявления регулярных выражений. Для этого нужно просто поместить регулярное выражение между двумя символами слэша. Если символ слэша присутствует внутри самого регулярного выражения, его следует экранировать символом обратного слэша.

Если нежелательно экранировать символы слэша в регулярном выражении, можно поставить перед началом регулярного выражения префикс r% и затем использовать в качестве разделителя любой другой символ пунктуации.

Несмотря на то, что существует возможность создавать объекты Regex из строки, тем не менее, нет смысла использовать строковую форму записи для представления регулярных выражений в программном коде. В этом случае потребовалось бы экранировать кавычки и символы обратного слэша, что обычно приводит к появлению частокола из обратных слэшей.

В этом отношении язык Ruby очень напоминает язык JavaScript, за исключением имени класса, которое в языке Ruby записывается как одно слово Regexp, а в языке JavaScript – как RegExp, в стиле «camel caps».

См. также

В рецепте 2.3 объясняется, как работают символьные классы и почему в регулярных выражениях необходимо записывать два символа обратного слэша, чтобы включить только один символ в символьный класс.

В рецепте 3.4 описывается, как установить параметры регулярного выражения, которые в отдельных языках программирования реализованы как часть литералов регулярных выражений.

3.2. Импортирование библиотеки регулярных выражений

Задача

Прежде чем появится возможность использовать регулярные выражения в приложении, необходимо импортировать в программный код библиотеку регулярных выражений или пространство имен.

В остальных фрагментах исходных текстов в этой книге предполагается, что импортирование уже было выполнено, если оно было необходимо.

Решение

C#

```
using System.Text.RegularExpressions;
```

VB.NET

```
Imports System.Text.RegularExpressions
```

Java

```
import java.util.regex.*;
```

Python

```
import re
```

Обсуждение

В некоторых языках программирования поддержка регулярных выражений встроена в сам язык. При использовании этих языков не требуется импортировать что-либо, чтобы обеспечить поддержку регулярных выражений. В других языках поддержка регулярных выражений обеспечивается с помощью библиотек, которые необходимо импортировать в исходных текстах. Некоторые языки программирования вообще не имеют поддержки регулярных выражений. Для таких языков программирования поддержку регулярных выражений необходимо скомпилировать и связать самостоятельно.

C#

Если поместить в начало файла с исходным программным кодом на языке C# инструкцию `using`, появится возможность напрямую обращаться к классам, обеспечивающим функциональные возможности регулярных выражений без необходимости полностью квалифицировать их имена. Например, обращение `System.Text.RegularExpressions.Regex()` можно будет записать просто как `Regex()`.

VB.NET

Если поместить в начало файла с исходным программным кодом на языке VB.NET инструкцию `Imports`, появится возможность напрямую обращаться к классам, обеспечивающим функциональные возможности регулярных выражений без необходимости полностью квалифици-

ровать их имена. Например, обращение `System.Text.RegularExpressions.Regex()` можно будет записать просто как `Regex()`.

Java

Чтобы в языке Java воспользоваться встроенной библиотекой регулярных выражений, необходимо импортировать в приложение пакет `java.util.regex`.

JavaScript

Встроенная поддержка регулярных выражений в языке JavaScript доступна всегда.

PHP

Функции `preg` встроены в сам язык и всегда доступны в версии PHP 4.2.0 и выше.

Perl

Встроенная поддержка регулярных выражений в языке Perl доступна всегда.

Python

Чтобы получить возможность использовать функции регулярных выражений в языке Python, необходимо в сценарии импортировать модуль `re`.

Ruby

Встроенная поддержка регулярных выражений в языке Ruby доступна всегда.

3.3. Создание объектов регулярных выражений

Задача

Необходимо создать объект регулярного выражения или как-то иначе скомпилировать регулярное выражение, чтобы его можно было эффективно использовать в приложении.

Решение

C#

Если известно, что регулярное выражение не содержит ошибок:

```
Regex regexObj = new Regex("regex pattern");
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной `UserInput`):

```
try {
    Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

Если известно, что регулярное выражение не содержит ошибок:

```
Dim RegexObj As New Regex("regex pattern")
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной `UserInput`):

```
Try
    Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
    ' Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Если известно, что регулярное выражение не содержит ошибок:

```
Pattern regex = Pattern.compile("regex pattern");
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной `userInput`):

```
try {
    Pattern regex = Pattern.compile(userInput);
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы иметь возможность использовать регулярное выражение в виде строки, необходимо создать объект Matcher:

```
Matcher regexMatcher = regex.matcher(subjectString);
```

Чтобы использовать регулярное выражение в виде другой строки, необходимо создать новый объект Matcher, как только что было продемонстрировано, или повторно использовать существующий объект:

```
regexMatcher.reset(anotherSubjectString);
```

JavaScript

Литерал регулярного выражения в программном коде:

```
var myregexp = /regex pattern/;
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной `userinput`:

```
var myregexp = new RegExp(userinput);
```

Perl

```
$myregex = qr/regex pattern/
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной `$userinput`:

```
$myregex = qr/$userinput/
```

Python

```
reobj = re.compile("regex pattern")
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной `userinput`:

```
reobj = re.compile(userinput)
```

Ruby

Литерал регулярного выражения в программном коде:

```
myregexp = /regex pattern/;
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной `userinput`:

```
myregexp = Regexp.new(userinput);
```

Обсуждение

Прежде чем механизм регулярных выражений сможет выполнить сопоставление регулярного выражения со строкой, оно должно быть скомпилировано. Компиляция выражения производится во время работы приложения. Конструктор объекта регулярного выражения или функция, выполняющая компилирование, производит синтаксический анализ строки, в которой хранится регулярное выражение, и преобразует ее в древовидную структуру или в конечный автомат. Функция, выполняющая фактическое сопоставление с шаблоном, выполняет обход этого дерева или конечного автомата в процессе просмотра строки с испытуемым текстом. В языках программирования, поддерживающих ли-

тералы регулярных выражений, производят компиляцию, когда поток выполнения достигает оператора регулярного выражения.

.NET

В языках C# и VB.NET скомпилированное регулярное выражение хранится в экземпляре класса `System.Text.RegularExpressions.Regex`. Самый простой конструктор принимает всего один параметр: строку с регулярным выражением.

Если в регулярном выражении обнаруживаются синтаксические ошибки, конструктор `Regex()` возбуждает исключение `ArgumentException`. Сообщение исключения точно укажет, какая ошибка произошла. Если регулярные выражения вводятся пользователем приложения, очень важно предусмотреть обработку этого исключения, в ходе которой можно отобразить текст исключения и предложить пользователю исправить ошибку.

Если регулярное выражение включено в текст программы в виде строкового литерала, можно опустить действия по обработке исключения, если с помощью инструментальных средств анализа программного кода было проверено, что при выполнении строка не возбуждает исключения. Совершенно невозможно изменить состояние или режим так, чтобы один и тот же литерал регулярного выражения компилировался в одной ситуации и не компилировался в другой. Обратите внимание, что в том случае, если регулярное выражение содержит синтаксическую ошибку, исключение будет возбуждено на этапе выполнения программы, а не на этапе ее компиляции.

Если регулярное выражение предполагается использовать в цикле или многократно задействовать его в приложении, следует использовать объект `Regex`. Процесс конструирования объекта регулярного выражения создает дополнительную нагрузку. Статические члены класса `Regex`, которые принимают регулярное выражение в виде строкового параметра, в любом случае создают объект `Regex`, поэтому лучше сделать это в своем программном коде явно и потом хранить ссылку на созданный объект.

Если планируется использовать регулярное выражение один-два раза, можно пользоваться статическими членами класса `Regex`, чтобы сэкономить несколько строк программного кода. Статические члены класса `Regex` не стирают объект регулярного выражения, созданный внутренней реализацией, – они сохраняют в кэше 15 последних использовавшихся регулярных выражений. Существует возможность изменить размер кэша, установив значение свойства `Regex.CacheSize`. Поиск в кэше организован как поиск строки регулярного выражения. Но не следует во всем полагаться на кэш. Если в программе часто требуется большое число регулярных выражений, предпочтительнее будет создать собственный кэш, в котором можно организовать более эффективный поиск, чем поиск строки.

Java

В языке Java класс Pattern хранит одно скомпилированное регулярное выражение. Экземпляры этого класса можно создавать с помощью фабричной функции класса Pattern.compile(), которая принимает единственный параметр: строку с регулярным выражением.

Если в регулярном выражении обнаружится синтаксическая ошибка, функция Pattern.compile() возбудит исключение PatternSyntaxException. Сообщение исключения точно укажет, какая ошибка произошла. Если регулярные выражения вводятся пользователем приложения, очень важно предусмотреть обработку этого исключения, в ходе которой можно отобразить текст исключения и предложить пользователю исправить ошибку. Если регулярное выражение включено в текст программы в виде строкового литерала, можно опустить действия по обработке исключения, если с помощью инструментальных средств анализа программного кода было проверено, что при выполнении строка не возбуждает исключения. Совершенно невозможно изменить состояние или режим так, чтобы один и тот же литерал регулярного выражения компилировался в одной ситуации и не компилировался в другой. Обратите внимание, что в том случае, если регулярное выражение содержит синтаксическую ошибку, исключение будет возбуждено на этапе выполнения программы, а не на этапе ее компиляции.

Если регулярное выражение предполагается использовать многократно, следует создать объект Pattern вместо того, чтобы использовать статические члены класса String. Даже при том, что для этого потребуется несколько дополнительных строк программного кода, этот код будет работать гораздо эффективнее. Статические члены выполняют рекомпиляцию каждого регулярного выражения и при каждом обращении. В действительности статические члены реализованы исключительно для решения простейших задач применения регулярных выражений.

Объект Pattern просто хранит скомпилированную версию регулярного выражения – он не выполняет никакой фактической работы. Собственно сопоставление регулярного выражения производится классом Matcher. Чтобы создать экземпляр класса Matcher, необходимо вызвать метод matcher() объекта скомпилированного регулярного выражения. Метод matcher() принимает единственный аргумент – строку испытуемого текста.

Метод matcher() можно вызывать неограниченное число раз, чтобы применить одно и то же регулярное выражение к множеству строк. Можно работать с несколькими объектами, выполняющими сопоставление и использующими одно и то же регулярное выражение, при условии, что все они будут работать в одном потоке выполнения. Классы Pattern и Matcher не предусматривают возможность выполнения в многопоточной среде. Если необходимо использовать одно и то же регулярное выражение в нескольких потоках выполнения, функция Pattern.compile() должна вызываться в каждом таком потоке.

Если регулярное выражение было применено к одной строке и необходимо применить то же самое регулярное выражение к другой строке, чтобы повторно использовать объект `Matcher`, необходимо вызвать его метод `reset()`, передав ему следующую строку с испытуемым текстом. Такой подход эффективнее, чем создание нового объекта `Matcher`. Метод `reset()` возвращает тот же объект `Matcher`, которому принадлежит метод, что позволяет производить повторную инициализацию объекта и задействовать его в одной строке программного кода, например, `regexMatcher.reset(nextString).find()`.

JavaScript

Форма записи регулярных выражений, как показано в рецепте 3.2, уже создает новый объект регулярного выражения. Чтобы повторно использовать тот же самый объект, нужно просто присвоить его переменной.

Если регулярное выражение хранится в строковой переменной (например, регулярное выражение, введенное пользователем), для его компиляции следует использовать конструктор `RegExp()`. Обратите внимание, что регулярное выражение внутри строки не окружается символами слэша. Эти слэши являются частью синтаксиса JavaScript записи объектов `RegExp` в виде литералов, а вовсе не частью самого регулярного выражения.

Так как запись литерала регулярного выражения в переменную является тривиальной операцией, в большинстве решений для JavaScript в этой главе эта строка кода опущена и напрямую используется литерал регулярного выражения. Если предполагается многократно использовать одно и то же регулярное выражение, его необходимо присвоить переменной и использовать эту переменную, вместо того чтобы много раз вставлять один и тот же литерал регулярного выражения. Это повысит производительность и упростит сопровождение программного кода.

PHP

Язык PHP не предоставляет возможность сохранять скомпилированное регулярное выражение в переменной. Поэтому всякий раз, когда возникает потребность выполнить какую-либо операцию с применением регулярного выражения, функциям `preg` придется передавать регулярное выражение в виде строки.

Функции семейства `preg` хранят до 4 096 скомпилированных регулярных выражений в своем кэше. Несмотря на то, что поиск в кэше выполняется на основе хешей, тем не менее, скорость поиска намного ниже, чем скорость обращения к переменной, однако производительность от этого падает не так сильно, как она могла бы упасть, если бы

одно и то же регулярное выражение пришлось компилировать снова и снова.

Perl

В языке Perl имеется возможность с помощью оператора qr// скомпилировать регулярное выражение и присвоить его переменной. Этот оператор имеет тот же синтаксис, что и оператор сопоставления, описанный в рецепте 3.1, за исключением того, что он начинается не с символа m, а с символов qr.

Вообще язык Perl весьма эффективен в части многократного использования ранее скомпилированных регулярных выражений. Поэтому мы не будем использовать оператор qr// во фрагментах программного кода в этой главе. Его использование будет продемонстрировано только в рецепте 3.5.

Оператор qr// удобно использовать, когда необходимо интерпретировать переменные внутри регулярного выражения или когда само регулярное выражение является строкой (например, когда оно получено в результате ввода пользователем). При использовании qr/$regexstring/ имеется возможность управлять процессом повторной компиляции, чтобы отразить новое содержимое переменной $regexstring. Оператор m/$regexstring/ выполняет повторную компиляцию регулярного выражения всякий раз, тогда как m/$regexstring/o никогда не будет компилировать его повторно. Значение модификатора /o объясняется в рецепте 3.4.

Python

В языке Python, в модуле re, имеется функция compile(), которая принимает строку с регулярным выражением и возвращает объект со скомпилированным регулярным выражением.

Если то же самое регулярное выражение планируется использовать многократно, необходимо явно вызвать функцию compile(). Все функции, имеющиеся в модуле re, в самом начале вызывают функцию compile(), а затем вызывают требуемую функцию объекта скомпилированного регулярного выражения.

Функция compile() сохраняет ссылки на 100 последних скомпилированных регулярных выражений. Тем самым операция повторной компиляции любого из 100 последних использовавшихся регулярных выражений замещается операцией поиска по словарю. Когда кэш заполняется полностью, он очищается целиком.

Если проблема производительности не стоит слишком остро, быстродействия кэша будет вполне достаточно, чтобы напрямую использовать функции модуля re. Но когда производительность имеет большое значение, предпочтительнее пользоваться функцией compile().

Ruby

Форма записи регулярных выражений, продемонстрированная в рецепте 3.2, уже создает новый объект регулярного выражения. Чтобы повторно использовать тот же самый объект, нужно просто присвоить его переменной.

Если регулярное выражение хранится в строковой переменной (например, регулярное выражение, введенное пользователем), для его компиляции следует использовать фабричную функцию `Regexp.new()` или ее синоним `Regexp.compile()`. Обратите внимание, что регулярное выражение внутри строки не окружается символами слэша. Эти слэши являются частью синтаксиса Ruby записи объектов `Regexp` в виде литералов, а вовсе не частью самого регулярного выражения.

Так как запись литерала регулярного выражения в переменную является тривиальной операцией, в большинстве решений для Ruby в этой главе эта строка кода опущена и напрямую используется литерал регулярного выражения. Если предполагается многократно использовать одно и то же регулярное выражение, его необходимо присвоить переменной и использовать эту переменную, вместо того чтобы много раз вставлять один и тот же литерал регулярного выражения. Это повысит производительность и упростит сопровождение программного кода.

Компилирование регулярных выражений до уровня языка CIL

C#

```
Regex regexObj = new Regex("regex pattern", RegexOptions.Compiled);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern", RegexOptions.Compiled)
```

Обсуждение

При создании объекта `Regex` на платформе .NET без передачи дополнительных параметров регулярное выражение компилируется именно так, как описывалось в разделе «Обсуждение» на стр. 154. Если вторым параметром конструктору `Regex()` передать значение `RegexOptions.Compiled`, класс `Regex` выполнит нечто иное: он скомпилирует регулярное выражение до уровня языка CIL, известного также как MSIL. Название CIL происходит от «Common Intermediate Language» (общий промежуточный язык). Это низкоуровневый язык программирования, который гораздо ближе к языку ассемблера, чем к C# или Visual Basic. Все

компиляторы .NET производят программный код на языке CIL. Когда приложение запускается первый раз, платформа .NET компилирует программный код на языке CIL в машинные инструкции, подходящие для компьютера пользователя.

Преимущество компилирования регулярных выражений с параметром `RegexOptions.Compiled` заключается в том, что они могут выполняться до 10 раз быстрее, чем регулярные выражения, скомпилированные без этого параметра.

Недостаток такого подхода состоит в том, что компиляция может выполняться на два порядка медленнее, чем простое преобразование строки с регулярным выражением в древовидную структуру. Кроме того, программный код на языке CIL становится постоянной частью приложения и сохраняется, пока приложение не завершит работу. Программный код CIL не убирается сборщиком мусора.

Параметр `RegexOptions.Compiled` следует использовать, только если регулярное выражение слишком сложное или необходимо обрабатывать большие объемы текста, когда задержки, возникающие во время выполнения операции с регулярным выражением, становятся слишком заметными для пользователя. Дополнительные затраты на компиляцию и сборку не имеют смысла, если регулярное выражение выполняет свою работу за доли секунды.

См. также

Рецепты 3.1, 3.2 и 3.4.

3.4. Установка параметров регулярных выражений

Задача

Необходимо скомпилировать регулярное выражение со всеми доступными режимами сопоставления: «свободное форматирование», «нечувствительность к регистру символов», «точке соответствуют границы строк» и «символам ^ и $ соответствуют границы строк».

Решение

C#

```
Regex regexObj = new Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
    RegexOptions.Singleline | RegexOptions.Multiline);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
    RegexOptions.Singleline Or RegexOptions.Multiline)
```

Java

```
Pattern regex = Pattern.compile("regex pattern",
    Pattern.COMMENTS | Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
    Pattern.DOTALL | Pattern.MULTILINE);
```

JavaScript

Литерал регулярного выражения в программном коде:

```
var myregexp = /regex pattern/im;
```

Регулярное выражение, получено от пользователя в виде строки:

```
var myregexp = new RegExp(userinput, "im");
```

PHP

```
regexstring = '/regex pattern/simx';
```

Perl

```
m/regex pattern/simx;
```

Python

```
reobj = re.compile("regex pattern",
    re.VERBOSE | re.IGNORECASE |
    re.DOTALL | re.MULTILINE)
```

Ruby

Литерал регулярного выражения в программном коде:

```
myregexp = /regex pattern/mix;
```

Регулярное выражение, получено от пользователя в виде строки:

```
myregexp = Regexp.new(userinput,
    Regexp::EXTENDED or Regexp::IGNORECASE or
    Regexp::MULTILINE);
```

Обсуждение

Многие регулярные выражения из тех, что приводятся в книге, и из тех, что встретятся вам на практике, написаны с учетом использования в определенном режиме. Почти все современные диалекты регулярных выражений поддерживают четыре основных режима. К сожалению,

в некоторых диалектах параметры, реализующие те или иные режимы, имеют противоречивые или запутывающие названия. Использование ошибочных режимов обычно отрицательно сказывается на работоспособности регулярного выражения.

Во всех решениях в этом рецепте для установки режимов используются флаги или параметры, предоставляемые языком программирования или классом регулярных выражений. Другой способ установки режимов заключается в использовании модификаторов режимов внутри регулярного выражения. Модификаторы режимов внутри регулярного выражения всегда имеют высший приоритет перед параметрами или флагами, которые устанавливаются за пределами регулярного выражения.

.NET

Конструктор `Regex()` принимает второй необязательный аргумент с параметрами регулярного выражения. Доступные параметры можно увидеть в перечислении `RegexOptions`.

Свободное форматирование: `RegexOptions.IgnorePatternWhitespace`
Нечувствительность к регистру символов: `RegexOptions.IgnoreCase`
Точке соответствуют границы строк: `RegexOptions.Singleline`
Символам ^ и $ соответствуют границы строк: `RegexOptions.Multiline`

Java

Фабричная функция `Pattern.compile()` принимает второй необязательный аргумент с параметрами регулярного выражения. Класс `Pattern` определяет несколько констант для установки различных режимов. Имеется возможность одновременно установить несколько параметров, объединив их оператором «поразрядное ИЛИ» `|`.

Свободное форматирование: `Pattern.COMMENTS`
Нечувствительность к регистру символов: `Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE`
Точке соответствуют границы строк: `Pattern.DOTALL`
Символам ^ и $ соответствуют границы строк: `Pattern.MULTILINE`

В действительности существует два параметра, определяющие режим нечувствительности к регистру символов, и, чтобы обеспечить полную нечувствительность к регистру символов, следует установить оба. Если установить только `Pattern.CASE_INSENSITIVE`, то сопоставление без учета регистра символов будет производиться только для латинских символов от A до Z.

Если установить оба параметра, то все символы из всех алфавитов будут сопоставляться без учета регистра символов. Единственная причина, чтобы отказаться от использования параметра `Pattern.UNICODE_CASE`, заключается в стремлении обеспечить высокую производительность, если заранее известно, что испытуемый текст содержит только сим-

волы ASCII. При использовании модификаторов режима внутри регулярных выражений модификатор ‹(?i)› включает режим нечувствительности к регистру только для символов ASCII, а модификатор ‹(?iu)› – режим полной нечувствительности к регистру символов.

JavaScript

Чтобы определить, какие-либо параметры в языке JavaScript, необходимо добавить один или более односимвольных флагов в литерал объекта RegExp после символа слэша, завершающего регулярное выражение. В документации эти флаги обычно записываются как /i и /m, хотя любой флаг – это всего лишь один символ. Перед флагами не должно добавляться никаких символов слэша.

При использовании конструктора RegExp() для компиляции строки в объект регулярного выражения можно передать второй необязательный аргумент с флагами регулярного выражения. Второй аргумент должен быть строкой с символами параметров, которые требуется установить. В этой строке не должно присутствовать никаких символов слэша.

Свободное форматирование: не поддерживается в JavaScript
Нечувствительность к регистру символов: /i
Точке соответствуют границы строк: не поддерживается в JavaScript
Символам ^ и $ соответствуют границы строк: /m

PHP

В рецепте 3.1 говорилось, что функции семейства preg в языке PHP требуют, чтобы литерал регулярного выражения был заключен между двумя символами пунктуации, обычно символами слэша, и был отформатирован как строковый литерал.

Чтобы определить параметры регулярного выражения, нужно добавить в конец строки один или более односимвольных модификаторов. То есть символы модификаторов должны следовать за разделителем, закрывающим регулярное выражение, но оставаться внутри строки в апострофах или в кавычках. В документации эти флаги обычно записываются как /x, хотя любой флаг – это всего лишь один символ, а разделитель между регулярным выражением и модификаторами необязательно должен быть символом слэша.

Свободное форматирование: /x
Нечувствительность к регистру символов: /i
Точке соответствуют границы строк: /s
Символам ^ и $ соответствуют границы строк: /m

Perl

Параметры регулярного выражения определяются добавлением одного или более односимвольных модификаторов в конец оператора сопостав-

ления с шаблоном или подстановки. В документации эти флаги обычно записываются как /x, хотя любой флаг – это всего лишь один символ, а разделитель между регулярным выражением и модификаторами необязательно должен быть символом слэша.

Свободное форматирование: /x
Нечувствительность к регистру символов: /i
Точке соответствуют границы строк: /s
Символам ^ и $ соответствуют границы строк: /m

Python

Функция compile() (описывается в предыдущем рецепте) принимает второй необязательный аргумент с параметрами регулярного выражения. Этот аргумент необходимо собирать с помощью оператора |, чтобы объединить значения констант, определяемых в модуле re. Многие другие функции в модуле re, которые принимают в качестве аргумента литерал регулярного выражения, также принимают параметры выражения в виде последнего необязательного аргумента.

Для каждого параметра регулярных выражений определена пара констант. То есть каждый параметр может быть представлен константой с полным именем параметра или константой с именем из одного символа. Значения констант внутри пары идентичны. Единственное отличие состоит в том, что использование полных имен делает программный код более удобочитаемым для других разработчиков, не знакомых с алфавитом параметров регулярных выражений. Односимвольные имена констант параметров, перечисленные в этом разделе, соответствуют их аналогам в языке Perl.

Свободное форматирование: re.VERBOSE или re.X
Нечувствительность к регистру символов: re.IGNORECASE или re.I
Точке соответствуют границы строк: re.DOTALL или re.S
Символам ^ и $ соответствуют границы строк: re.MULTILINE или re.M

Ruby

Чтобы определить какие-либо параметры в языке Ruby, необходимо добавить один или более односимвольных флагов в литерал объекта Regexp после символа слэша, завершающего регулярное выражение. В документации эти флаги обычно записываются как /i и /m, хотя любой флаг – это всего лишь один символ. Перед флагами не должно добавляться никаких символов слэша.

При использовании фабричной функции Regexp.new() для компиляции строки в объект регулярного выражения можно передать второй необязательный аргумент с флагами регулярного выражения. Второй аргумент должен быть либо значением nil, чтобы отключить все параметры, либо комбинацией констант класса Regexp, объединяемых оператором or.

Свободное форматирование: /r или Regexp::EXTENDED

Нечувствительность к регистру символов: /i или Regexp::IGNORECASE

Точке соответствуют границы строк: /m или Regexp::MULTILINE. В языке Ruby действительно используются модификатор «m» и название «multi line», тогда как в других диалектах для обозначения режима «точке соответствуют границы строк», используется модификатор «s» или название «single line».

Символам ^ и $ соответствуют границы строк: символ крышки и знак доллара в языке Ruby всегда совпадают с границами строк. Этот режим невозможно отключить. Поэтому соответствие началу и концу строки с испытуемым текстом следует проверять с помощью метасимволов ‹\A› и ‹\Z›.

Дополнительные параметры, характерные для различных языков

.NET

Параметр RegexOptions.ExplicitCapture превращает все группы, за исключением именованных, в несохраняющие. С этим параметром конструкция ‹(group)› становится эквивалентной конструкции ‹(?:group)›. Если сохраняющие группы всегда оформляются как именованные, включение этого параметра обеспечит более высокую эффективность регулярного выражения, без необходимости использовать синтаксическую конструкцию ‹(?:group)›. Вместо включения параметра RegexOptions.ExplicitCapture можно добавить в начало регулярного выражения модификатор режима ‹(?n)›. Подробнее о группировке рассказывается в рецепте 2.9, а в рецепте 2.11 рассказывается об именованных группах.

Если одно и то же регулярное выражение используется в программном коде .NET и JavaScript, и необходимо обеспечить одинаковое поведение выражения, можно включить параметр RegexOptions.ECMAScript. Этот параметр особенно удобен при разработке клиентской части веб-приложения на JavaScript и серверной части на ASP.NET. Наиболее важный эффект, производимый этим параметром, состоит в том, что метасимволы \w и \d ограничиваются набором символов ASCII, как в языке JavaScript.

Java

В языке Java имеется уникальный параметр Pattern.CANON_EQ, который включает режим «канонической эквивалентности». Как уже говорилось в разделе «Графема Юникода» на стр. 81, Юникод обеспечивает различные способы представления символов с диакритическими знаками. При включении этого параметра регулярное выражение будет со-

впадать с символами, даже если в испытуемом тексте они имеют различные представления. Например, регулярное выражение ‹\u00E0› будет совпадать и с последовательностью “\u00E0”, и с последовательностью “\u0061\u0300”, потому что они канонически эквивалентны. Оба представления выводятся на экран как символ «à» и неразличимы для конечного пользователя. При отключенном режиме канонической эквивалентности регулярное выражение ‹\u00E0› не будет совпадать с последовательностью “\u0061\u0300”. Именно такое поведение проявляют другие диалекты регулярных выражений, рассматриваемые в книге.

Наконец, при включенном параметре Pattern.UNIX_LINES только символ ‹\n› будет интерпретироваться как граница строки, а точка, крышка и знак доллара – нет. По умолчанию все символы разрыва строк в Юникоде интерпретируются, как символы границы строки.

JavaScript

Если необходимо повторно применять регулярное выражение к той же самой строке, то есть выполнить обход всех совпадений или отыскать и заменить все совпадения, а не только первое, следует указать флаг /g, или «global» (глобальный).

PHP

Флаг /u предписывает библиотеке PCRE интерпретировать регулярное выражение и строку испытуемого текста, как строки в кодировке UTF-8. Этот модификатор также позволяет использовать в регулярном выражении такие метасимволы Юникода, как ‹\p{FFFF}› и ‹\p{L}›. Они подробно описаны в рецепте 2.7. Без этого модификатора PCRE интерпретирует каждый байт как отдельный символ, а метасимволы Юникода вызывают появление ошибки.

Флаг /U меняет поведение «максимальных» и «минимальных» квантификаторов на противоположное, добавляя дополнительный знак вопроса. Обычно квантификатор ‹.*› является максимальным, а квантификатор ‹.*?› – минимальным. При наличии флага /U квантификатор ‹.*› становится минимальным, а квантификатор ‹.*?› – максимальным.

Я настоятельно рекомендую никогда не использовать этот флаг, так как он будет сбивать с толку программистов, которым позднее придется читать ваш программный код и которые пропустят дополнительный модификатор /U, уникальный для PHP. Кроме того, не путайте флаги /U и /u, если они встретятся где-нибудь в программном коде. Модификаторы режима чувствительны к регистру символов.

Perl

Если необходимо повторно применять регулярное выражение к той же самой строке (например, выполнить обход всех совпадений или оты-

скать и заменить все совпадения, а не только первое), следует указать флаг /g, или «global» (глобальный).

Если в регулярном выражении имеется переменная, например m/I am $name/, Perl будет компилировать регулярное выражение при каждом его использовании, потому что содержимое переменной $name может измениться. Отменить это поведение можно с помощью модификатора /o. Выражение m/I am $name/o будет компилироваться только при первом его использовании, а при последующих использованиях будет задействована скомпилированная версия. Если содержимое переменной $name изменится, это никак не отразится на регулярном выражении. Информация о том, как управлять повторной компиляцией регулярных выражений, приводится в рецепте 3.3.

Python

В языке Python имеется два дополнительных параметра, которые изменяют значение метасимвола границы слова (рецепт 2.6) и сокращенных форм записи символьных классов ‹\w›, ‹\d› и ‹\s›, а также их инвертированные версии (рецепт 2.3). По умолчанию эти метасимволы обслуживают только алфавитные символы ASCII, цифры и пробельные символы.

Параметр re.LOCALE, или re.L, обеспечивает чувствительность этих метасимволов к текущим региональным настройкам. Региональные настройки определяют, какие символы будут интерпретироваться этими метасимволами регулярных выражений как алфавитные символы, цифры и пробельные символы. Этот параметр необходимо использовать, когда испытуемый текст не является строкой Юникода и необходимо, чтобы также обрабатывались такие символы, как символы с диакритическими знаками.

Параметр re.UNICODE, или re.U, обеспечивает зависимость метасимволов от стандарта Юникод. Все символы, определяемые в Юникоде как буквы, цифры и пробельные символы, будут соответственно интерпретироваться этими метасимволами. Этот параметр необходимо использовать, когда испытуемый текст, к которому применяется регулярное выражение, является строкой Юникода.

Ruby

Фабричная функция Regexp.new() принимает необязательный третий аргумент, в котором указывается кодировка символов, поддерживаемая регулярным выражением. Если не указывать кодировку, будет применяться та же кодировка, которая используется в исходном файле. В большинстве случаев использование кодировки исходных файлов является правильным выбором.

Чтобы выбрать кодировку явно, необходимо передать в этом аргументе единственный символ. Аргумент не чувствителен к регистру символов. Возможные значения приводятся ниже:

`n`

Происходит от слова «None» (нет). Каждый байт строки будет интерпретироваться, как отдельный символ. Это значение следует использовать для текста в кодировке ASCII.

`e`

Активизирует кодировку «EUC» для языков Дальнего Востока.

`s`

Активизирует японскую кодировку «Shift-JIS».

`u`

Активизирует кодировку UTF-8, в которой для представления одного символа может использоваться от одного до четырех байтов и поддерживаются все языки, поддерживаемые стандартом Юникод (в число которых входят все живые языки, независимо от их значимости).

При использовании литералов регулярных выражений, кодировка может выбираться с помощью модификаторов `/n`, `/e`, `/s` и `/u`. В каждом отдельном регулярном выражении может использоваться только один из этих модификаторов. Они могут использоваться в комбинации с любыми другими модификаторами `/x`, `/i` и `/m`.

Не путайте модификатор `/s` в языке Ruby с похожим модификатором в языках Perl, Java или .NET. В Ruby модификатор `/s` активизирует применение кодировки «Shift-JIS». В Perl и в большинстве других диалектов регулярных выражений он включает режим «точке соответствуют границы строки». В Ruby этот режим включается с помощью флага `/m`.

См. также

Эффекты, производимые режимами сопоставления, подробно описываются в главе 2. Там же описывается порядок использования модификаторов режима внутри регулярного выражения.

`Свободное форматирование`: рецепт 2.18
`Нечувствительность к регистру символов`: «Поиск без учета регистра символов» на стр. 51 в рецепте 2.1
`Точке соответствуют границы строк`: рецепт 2.4
`Символам ^ и $ соответствуют границы строк`: рецепт 2.5

В рецептах 3.1 и 3.3 описывается порядок использования литералов регулярных выражений в исходных текстах программ и создания объектов регулярных выражений. Параметры регулярных выражений устанавливаются на этапе их создания.

3.5. Проверка возможности совпадения в пределах испытуемой строки

Задача

Необходимо проверить, будет ли найдено совпадение с определенным регулярным выражением в определенной строке. Достаточно частичного совпадения. Например, регулярному выражению ‹regex•pattern› соответствует часть строки The regex pattern can be found. По условиям задачи не требуется беспокоиться о каких-либо особенностях сопоставления, достаточно просто узнать, имеется совпадение или нет.

Решение

C#

Для быстрых однократных проверок можно использовать статические функции:

```
bool foundMatch = Regex.IsMatch(subjectString, "regex pattern");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
bool foundMatch = false;
try {
    foundMatch = Regex.IsMatch(subjectString, UserInput);
} catch (ArgumentNullException ex) {
    // Нельзя передавать значение null в качестве регулярного выражения
    // или строки с испытуемым текстом
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект Regex:

```
Regex regexObj = new Regex("regex pattern");
bool foundMatch = regexObj.IsMatch(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex с полноценной обработкой исключений:

```
bool foundMatch = false;
try {
    Regex regexObj = new Regex(UserInput);
    try {
        foundMatch = regexObj.IsMatch(subjectString);
    } catch (ArgumentNullException ex) {
```

```
            // Нельзя передавать значение null в качестве регулярного выражения
            // или строки с испытуемым текстом
        }
    } catch (ArgumentException ex) {
        // Синтаксическая ошибка в регулярном выражении
    }
```

VB.NET

Для быстрых однократных проверок можно использовать статические функции:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "regex pattern")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
Dim FoundMatch As Boolean
Try
    FoundMatch = Regex.IsMatch(SubjectString, UserInput)
Catch ex As ArgumentNullException
    'Нельзя передавать значение Nothing в качестве регулярного выражения
    'или строки с испытуемым текстом
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект `Regex`:

```
Dim RegexObj As New Regex("regex pattern")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Функции `IsMatch()` обязательно должна передаваться строка `SubjectString` в качестве единственного аргумента, и вызов должен производиться относительно экземпляра класса `RegexObj`, а не как статическая функция класса `Regex`:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Regex` с полноценной обработкой исключений:

```
Dim FoundMatch As Boolean
Try
    Dim RegexObj As New Regex(UserInput)
    Try
        FoundMatch = Regex.IsMatch(SubjectString)
    Catch ex As ArgumentNullException
        'Нельзя передавать значение Nothing в качестве регулярного выражения
        'или строки с испытуемым текстом
```

```
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Единственный способ проверить частичное совпадение заключается в том, чтобы создать объект класса `Matcher`:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.find();
```

Если регулярное выражение вводится конечным пользователем, следует предусмотреть полноценную обработку исключений:

```
boolean foundMatch = false;
try {
    Pattern regex = Pattern.compile(UserInput);
    Matcher regexMatcher = regex.matcher(subjectString);
    foundMatch = regexMatcher.find();
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
if (/regex pattern/.test(subject)) {
    // Успешное совпадение
} else {
    // Попытка сопоставления завершилась неудачей
}
```

PHP

```
if (preg_match('/regex pattern/', $subject)) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Perl

Когда испытуемая строка хранится в специальной переменной `$_`:

```
if (m/regex pattern/) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Когда испытуемая строка хранится в переменной `$subject`:

```
if ($subject =~ m/regex pattern/) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

При использовании регулярного выражения, скомпилированного ранее:

```
$regex = qr/regex pattern/;
if ($subject =~ $regex) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
if re.search("regex pattern", subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

Чтобы повторно использовать то же регулярное выражение, следует использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("regex pattern")
if reobj.search(subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

Ruby

```
if subject =~ /regex pattern/
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
end
```

Следующий фрагмент делает то же самое:

```
if /regex pattern/ =~ subject
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
End
```

Обсуждение

Наиболее часто регулярные выражения используются для проверки соответствия строки регулярному выражению. В большинстве языков программирования вполне достаточно, чтобы регулярное выражение совпало хотя бы с частью строки, чтобы функция сопоставления вернула истинное значение. Функция сопоставления просматривает всю строку с испытуемым текстом, чтобы определить, совпадает ли регулярное выражение с какой-либо ее частью. Функция возвращает истинное значение, как только будет найдено первое совпадение. Ложное значение возвращается, когда был достигнут конец строки и при этом не было найдено ни одного совпадения.

Примеры программного кода в этом рецепте могут использоваться для проверки наличия в строке некоторых данных. Если требуется проверить, соответствует ли вся строка определенному шаблону (например, с целью проверки ввода пользователя), следует использовать следующий рецепт.

C# и VB.NET

Класс `Regex` предоставляет нам четыре перегруженных версии метода `IsMatch()`, два из которых являются статическими. Это позволяет вызывать метод `IsMatch()` с различными аргументами. Испытуемая строка всегда передается в первом параметре. Это та самая строка, в которой регулярное выражение будет пытаться отыскать совпадение. Первый аргумент не может иметь значение `null`. В противном случае метод `IsMatch()` возбудит исключение `ArgumentNullException`.

Проверку можно выполнить в единственной строке программного кода, вызвав статический метод `Regex.IsMatch()`, без создания объекта `Regex`. В этом случае регулярное выражение передается во втором аргументе, а параметры регулярного выражения – в необязательном третьем аргументе. Если регулярное выражение содержит синтаксическую ошибку, метод `IsMatch()` возбудит исключение `ArgumentException`. Если регулярное выражение не содержит ошибок, метод вернет значение `true` в случае, когда в строке имеется совпадение, и `false` – если совпадение не было найдено.

Если одно и то же регулярное выражение требуется применить для проверки множества строк, можно повысить производительность программного кода, предварительно создав объект `Regex`, и вызывая метод `IsMatch()` этого объекта. Этот метод принимает единственный обязательный аргумент – испытуемую строку. Во втором необязательном аргументе можно указать позицию символа, начиная с которого следует выполнять поиск. Фактически число, указываемое во втором аргументе, – это количество символов от начала испытуемой строки, которые должны игнорироваться регулярным выражением. Это

может быть удобно, когда часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть больше или равно нулю и меньше или равно длине строки. В противном случае метод `IsMatch()` возбудит исключение `ArgumentOutOfRangeException`.

Статические перегруженные версии методов не имеют аргумента, с помощью которого можно было бы указать, с какой позиции в строке следует начинать поиск совпадения. Не существует перегруженной версии метода `IsMatch()` которая позволяла бы указывать, в какой позиции перед концом строки следует останавливать поиск. Если это необходимо, можно вызвать метод `Regex.Match("subject", start, stop)` и проверить значение свойства `Success` возвращаемого объекта `Match`. Подробнее об этом рассказывается в рецепте 3.8.

Java

Чтобы проверить наличие совпадения регулярного выражения с полной строкой или с ее частью, необходимо создать объект `Matcher`, как описывается в рецепте 3.3. После этого можно вызвать метод `find()` вновь созданного или повторно инициализированного объекта `Matcher`.

Методы `String.matches()`, `Pattern.matches()` и `Matcher.matches()` не пригодны для решения поставленной здесь задачи, так как они требуют, чтобы регулярное выражение совпадало с полной строкой.

JavaScript

Чтобы проверить наличие совпадения регулярного выражения с частью строки, необходимо вызвать метод `test()` регулярного выражения. Испытуемая строка передается этому методу в виде единственного аргумента.

Метод `regexp.test()` возвращает `true`, если регулярному выражению соответствует вся испытуемая строка или ее часть, и `false` — в противном случае.

PHP

Функция `preg_match()` может использоваться для разных целей. В простейшем случае ей передаются всего два обязательных аргумента: строка с регулярным выражением и строка с испытуемым текстом, который необходимо проверить с помощью регулярного выражения. Если совпадение было найдено, функция `preg_match()` вернет значение `1`, если в испытуемом тексте не было обнаружено ни одного совпадения, она вернет значение `0`.

Необязательные аргументы, которые могут передаваться функции `preg_match()`, описываются в последующих рецептах.

Perl

В языке Perl синтаксическая конструкция m// фактически является не просто контейнером для хранения регулярного выражения, а оператором регулярного выражения. При использовании оператора //m самого по себе в качестве испытуемого текста используется содержимое переменной $_.

Если необходимо применить оператор сопоставления m// к содержимому другой переменной, следует использовать оператор связывания =~, который свяжет оператор регулярного выражения с указанной переменной. Оператор сопоставления с шаблоном вернет значение true, если регулярному выражению соответствует часть испытуемой строки, и false – если ни одного совпадения не было найдено.

Если необходимо убедиться в отсутствии совпадений с регулярным выражением в строке, можно использовать оператор !~, который является инвертированной версией оператора =~.

Python

В модуле re имеется функция search(), которая выполняет поиск совпадений частей строки с регулярным выражением. Регулярное выражение передается функции в первом аргументе, а испытуемая строка – во втором. Параметры регулярного выражения можно передать в третьем необязательном аргументе.

Функция re.search() вызывает функцию re.compile() и затем вызывает метод search() объекта скомпилированного регулярного выражения. Этот метод принимает единственный аргумент: испытуемую строку.

Если регулярное выражение обнаружит совпадение, функция search() вернет экземпляр класса MatchObject. Если совпадение не будет найдено, функция search() вернет значение None. При использовании возвращаемого значения в условном операторе if объект MatchObject оценивается, как значение True, а значение None оценивается, как False. В последующих рецептах этой главы демонстрируется, как использовать информацию, хранящуюся в объекте MatchObject.

Не следует путать функции search() и match(). Функция match() не может использоваться для поиска совпадения в середине строки. Функция match() используется в следующем рецепте.

Ruby

Оператор =~ – это оператор сопоставления с шаблоном. Чтобы отыскать в строке первое совпадение с регулярным выражением, необходимо поместить его между регулярным выражением и строкой. Оператор возвращает номер позиции найденного совпадения. Если совпадение не будет обнаружено, оператор вернет значение nil.

Этот оператор реализован в обоих классах, `Regexp` и `String`. В версии Ruby 1.8 не имеет никакого значения, экземпляр какого класса будет располагаться слева или справа от оператора. В версии Ruby 1.9 с этим связан побочный эффект, имеющий отношение к именованным сохраняющим группам. Подробнее об этом рассказывается в рецепте 3.9.

Во всех других фрагментах программного кода на языке Ruby в этой книге мы будем помещать испытуемую строку слева от оператора `=~`, а регулярное выражение – справа. Это соответствует синтаксису языка Perl, из которого Ruby заимствовал оператор `=~`, и позволяет избежать неожиданностей версии Ruby 1.9, связанных с именованными сохраняющими группами, которые для кого-нибудь могут оказаться неожиданными.

См. также

Рецепты 3.6 и 3.7.

3.6. Проверка совпадения со всей испытуемой строкой

Задача

Необходимо проверить, соответствует ли строка целиком некоторому регулярному выражению. То есть необходимо убедиться, что регулярное выражение содержит шаблон, которому соответствует строка целиком, от начала до конца. Например, допустим, что имеется регулярное выражение `‹regex•pattern›`, оно будет совпадать с входным текстом, состоящим из строки `regex pattern`, но не с более длинной строкой `The regex pattern can be found`.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
bool foundMatch = Regex.IsMatch(subjectString, @"\Aregex pattern\Z");
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex(@"\Aregex pattern\Z");
bool foundMatch = regexObj.IsMatch(subjectString);
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "\Aregex pattern\Z")
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
Dim RegexObj As New Regex("\Aregex pattern\Z")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Функции IsMatch() обязательно должна передаваться строка SubjectString в качестве единственного аргумента, и вызов должен производиться относительно экземпляра класса RegexObj, а не как статическая функция класса Regex:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Java

Для однократной проверки можно использовать статическую функцию:

```
boolean foundMatch = subjectString.matches("regex pattern");
```

В случае многократного использования того же самого регулярного выражения предпочтительнее скомпилировать регулярное выражение и создать объект совпадения:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.matches(subjectString);
```

JavaScript

```
if (/^regex pattern$/.test(subject)) {
    // Успешное совпадение
} else {
    // Попытка сопоставления завершилась неудачей
}
```

PHP

```
if (preg_match('/\Aregex pattern\Z/', $subject)) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Perl

```
if ($subject =~ m/\Aregex pattern\Z/) {
    # Успешное совпадение
```

```
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
if re.match(r"regex pattern\Z", subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"regex pattern\Z")
if reobj.match(subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

Ruby

```
if subject =~ /\Aregex pattern\Z/
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
End
```

Обсуждение

Обычно успех сопоставления с регулярным выражением говорит о том, что *где-то* внутри испытуемого текста имеется совпадение с шаблоном. Во многих ситуациях бывает необходимо убедиться, что регулярному выражению соответствует не какая-то часть, а весь испытуемый текст *целиком*. Пожалуй, наиболее типичная ситуация, когда необходимо выяснить, совпадает ли текст целиком, – это проверка ввода. Если пользователь вводит номер телефона или IP-адрес и включает посторонние символы, необходимо отклонить его ввод.

Решения, где используются якорные метасимволы ‹$› и ‹\Z›, также могут использоваться при построчной обработке файлов (рецепт 3.21), когда используемый механизм извлечения строк оставляет символы завершения строки в конце строки. Как описывается в рецепте 2.5, эти якорные метасимволы также совпадают с последним символом завершения строки, фактически позволяя игнорировать его.

В последующих подразделах подробно описываются решения для различных языков программирования.

C# и VB.NET

Класс Regex в платформе .NET не имеет функции, позволяющей убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать тот же самый метод IsMatch(), описанный в предыдущем рецепте.

Java

В языке Java имеются три метода с именем matches(). Все они проверяют, совпадает ли регулярное выражение со строкой целиком. Эти методы обеспечивают быстрый способ проверки ввода без заключения регулярного выражения в якорные метасимволы, совпадающие с началом и с концом строки.

Метод matches() класса String принимает регулярное выражение в виде единственного аргумента. Он возвращает значение true или false как признак совпадения регулярного выражения со всей строкой целиком. Метод matches() класса Pattern принимает две строки: первая – это регулярное выражение, а вторая – испытуемая строка. В качестве испытуемой строки методу Pattern.matches() в действительности можно передавать любой объект типа CharSequence. Это единственная причина отказа от использования String.matches() в пользу Pattern.matches().

Оба метода, String.matches() и Pattern.matches(), при каждом обращении производят компиляцию регулярного выражения, вызывая Pattern.compile("regex").matcher(subjectString).matches(). Так как регулярное выражение компилируется каждый раз, эти методы желательно использовать только при однократном использовании регулярного выражения (например, для проверки одного поля ввода в форме) или когда не требуется высокая производительность. Эти методы не предоставляют возможность указывать параметры сопоставления за пределами регулярного выражения. Если регулярное выражение содержит синтаксическую ошибку, возбуждается исключение PatternSyntaxException.

В случае многократного использования одного и того же регулярного выражения для повышения эффективности необходимо скомпилиро-

вать его и создать объект Matcher, как описывается в рецепте 3.3. После этого можно будет вызывать метод matches() созданного экземпляра класса Matcher. Этот метод не принимает никаких аргументов, потому что испытуемая строка указывается при создании объекта сопоставления.

JavaScript

В языке JavaScript отсутствует функция, позволяющая убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹^›, а в конец выражения – якорный метасимвол ‹$›. При этом не следует устанавливать флаг /m регулярного выражения. Только при отсутствии флага /m символ крышки и знак доллара будут совпадать исключительно с началом и с концом испытуемой строки. При установленном флаге /m они также будут совпадать с границами строк в середине испытуемого текста.

После добавления якорных метасимволов в регулярное выражение его можно будет использовать с методом regexp.test(), описанным в предыдущем рецепте.

PHP

В языке PHP отсутствует функция, позволяющая убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать функцию preg_match(), описанную в предыдущем рецепте.

Perl

В языке Perl имеется всего один оператор сопоставления с шаблоном, который удовлетворяется совпадением с частью строки. Если необходимо проверить, соответствует ли регулярному выражению вся строка целиком, нужно в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как

‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, его можно будет использовать, как описано в предыдущем рецепте.

Python

Функция match() близко напоминает функцию search(), описанную в предыдущем рецепте. Основное отличие состоит в том, что функция match() требует, чтобы регулярное выражение совпадало исключительно с начала испытуемой строки. Если регулярное выражение не совпадает с началом строки, функция match() тут же возвращает значение None. Функция search(), в свою очередь, будет пытаться отыскать совпадение с регулярным выражением в каждой последующей позиции в строке, пока либо не обнаружит совпадение, либо не достигнет конца испытуемой строки.

Функция match() не требует, чтобы вся строка целиком соответствовала регулярному выражению. Она допускает совпадение с частью строки, при условии, что это совпадение начинается в начале строки. Если необходимо проверить, соответствует ли регулярному выражению вся строка целиком, нужно в конец выражения добавить якорный метасимвол ‹\Z›, совпадающий с концом строки.

Ruby

Класс Regex в языке Ruby не имеет функции, позволяющей убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать оператор =~, описанный в предыдущем рецепте.

См. также

Подробно о том, как работают якорные метасимволы, рассказывается в рецепте 2.5.

В рецептах 2.8 и 2.9 описываются оператор выбора и группировка. Если регулярное выражение использует конструкцию выбора за пределами

какой-либо группы, ее необходимо заключить в группу перед добавлением якорных метасимволов. Если регулярное выражение не использует конструкцию выбора или использует ее только в пределах групп, в этих случаях, чтобы обеспечить корректную работу якорей, дополнительная группировка не требуется.

Когда является допустимым совпадение с регулярным выражением части строки, вам может пригодиться рецепт 3.5.

3.7. Извлечение текста совпадения

Задача

Имеется регулярное выражение, совпадающее с частью испытуемого текста. Необходимо извлечь совпавшие фрагменты текста. Если для регулярного выражения в строке имеется более одного совпадения, необходимо извлечь только первое из них. Например, в результате применения регулярного выражения `<\d+>` к строке `Do you like 13 or 42?` должно быть извлечено совпадение 13.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString, @"\d+").Value;
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
string resultString = null;
try {
resultString = Regex.Match(subjectString, @"\d+").Value;
} catch (ArgumentNullException ex) {
        // Нельзя передавать значение null в качестве регулярного выражения
        // или строки с испытуемым текстом
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект `Regex`:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Match(subjectString).Value;
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex с полноценной обработкой исключений:

```
string resultString = null;
try {
    Regex regexObj = new Regex(@"\d+");
    try {
        resultString = regexObj.Match(subjectString).Value;
    } catch (ArgumentNullException ex) {
        // Нельзя передавать значение null в качестве
        // строки с испытуемым текстом
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString, "\d+").Value
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
ResultString = Regex.Match(SubjectString, "\d+").Value
Catch ex As ArgumentNullException
    'Нельзя передавать значение Nothing в качестве регулярного выражения
    'или строки с испытуемым текстом
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект Regex:

```
Dim RegexObj As New Regex("\d+")
Dim ResultString = RegexObj.Match(SubjectString).Value
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex с полноценной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
Dim RegexObj As New Regex("\d+")
Try
ResultString = RegexObj.Match(SubjectString).Value
```

```
Catch ex As ArgumentNullException
    ‘Нельзя передавать значение Nothing в качестве
    ‘строки с испытуемым текстом
End Try
Catch ex As ArgumentException
    ‘Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Чтобы отыскать и сохранить совпадение, необходимо создать объект `Matcher`:

```
String resultString = null;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group();
}
```

Если регулярное выражение вводится конечным пользователем, следует предусматривать полноценную обработку исключений:

```
String resultString = null;
try {
    Pattern regex = Pattern.compile("\\d+");
    Matcher regexMatcher = regex.matcher(subjectString);
    if (regexMatcher.find()) {
        resultString = regexMatcher.group();
    }
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
var result = subject.match(/\d+/);
if (result) {
    result = result[0];
} else {
    result = '';
}
```

PHP

```
if (preg_match('/\d+/', $subject, $groups)) {
    $result = $groups[0];
} else {
    $result = '';
}
```

Perl

```
if ($subject =~ m/\d+/) {
    $result = $&;
} else {
    $result = '';
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search("regex pattern", subject)
if matchobj:
    result = matchobj.group()
else:
    result = ""
```

Чтобы иметь возможность использовать то же самое регулярное выражение повторно, следует создать объект скомпилированного регулярного выражения:

```
reobj = re.compile("regex pattern")
matchobj = reobj.search(subject)
if match:
    result = matchobj.group()
else:
    result = ""
```

Ruby

Можно использовать оператор =~ и его специальную переменную `$&`:

```
if subject =~ /regex pattern/
    result = $&
else
    result = ""
end
```

Как вариант, можно задействовать метод `match()` объекта `Regexp`:

```
matchobj = /regex pattern/.match(subject)
if matchobj
    result = matchobj[0]
else
    result = ""
end
```

Обсуждение

Извлечение фрагмента строки, соответствующего шаблону, – это еще одна из основных задач, решаемых с помощью регулярных выражений. Во всех языках программирования, рассматриваемых в этой книге, имеется простой способ получения первого совпадения с регулярным выражением в строке. Функция начнет попытки применения регулярного выражения с начала строки и продолжит просматривать ее, пока не будет обнаружено совпадение.

.NET

В классе `Regex` платформы .NET отсутствует функция-член, которая возвращала бы фрагмент, совпавший с регулярным выражением. Но в нем имеется метод `Match()`, возвращающий экземпляр класса `Match`. Объекты класса `Match` имеют свойство с именем `Value`, в котором хранится текст, совпавший с регулярным выражением. Если попытка найти совпадения с регулярным выражением потерпела неудачу, этот метод все равно вернет объект `Match`, свойство `Value` которого будет содержать пустую строку.

Всего имеется пять перегруженных версий метода `Match()`. В первом аргументе всегда передается строка с испытуемым текстом, в котором требуется отыскать совпадения с регулярным выражением. Этот аргумент не может иметь значение `null`. В противном случае метод `Match()` возбудит исключение `ArgumentNullException`.

Если регулярное выражение необходимо использовать небольшое число раз, можно воспользоваться статическим методом. В этом случае вторым аргументом методу передается регулярное выражение. В третьем необязательном аргументе можно передать параметры регулярного выражения. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение `ArgumentException`.

Если одно и то же регулярное выражение требуется применить для проверки множества строк, можно повысить производительность программного кода, предварительно создав объект `Regex` и затем вызывая метод `Match()` этого объекта. Этот метод принимает единственный обязательный аргумент – испытуемую строку. Во втором необязательном аргументе можно указать позицию символа, начиная с которого следует выполнять поиск. Фактически число, указываемое во втором аргументе – это количество символов от начала испытуемой строки, которые должны игнорироваться регулярным выражением. Это удобно, когда часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть в диапазоне от нуля до величины длины строки. В противном случае метод `Match()`возбудит исключение `ArgumentOutOfRangeException`.

Если методу передается второй аргумент с начальной позицией, то можно также передать третий аргумент – длину подстроки, в пределах которой следует искать совпадения с регулярным выражением. Это число должно быть больше или равно нулю и не превышать длины испытуемой строки (первый аргумент) минус начальное смещение (второй аргумент). Например, вызов `regexObj.Match("123456", 3, 2)` попытается отыскать совпадение в подстроке "45". Если величина третьего аргумента будет больше, чем длина испытуемой строки, метод `Match()` возбудит исключение `ArgumentOutOfRangeException`. Если величина третьего аргумента не будет превышать длину испытуемой строки, но сумма второго и третьего аргументов окажутся больше длины строки, то будет возбуждено другое исключение: `IndexOutOfRangeException`. Если вы позволяете конечному пользователю указывать начальную и конечную позиции, тогда либо проверяйте полученные значения перед вызовом метода `Match()`, либо предусмотрите обработку обоих типов исключений.

Статические методы не имеют аргументов, с помощью которых можно было бы указать, в какой части строки следует выполнять поиск совпадения с регулярным выражением.

Java

Чтобы получить часть строки, совпавшей с регулярным выражением, необходимо создать объект класса `Matcher`, как описывается в рецепте 3.3, и затем вызвать метод `find()` этого объекта без аргументов. Если метод `find()` вернет значение `true`, следует вызвать метод `group()` без аргументов, чтобы извлечь текст, совпавший с регулярным выражением. Если метод `find()` вернет значение `false`, метод `group()` вызывать не следует, так как в этом случае будет возбуждено исключение `IllegalStateException`.

Метод `Matcher.find()` принимает один необязательный аргумент с начальной позицией в испытуемой строке. Этот аргумент можно использовать, чтобы организовать поиск, начиная с определенной позиции в строке. Если указать значение ноль, поиск начнется с начала строки. Если указать значение меньше нуля или больше длины испытуемой строки, будет возбуждено исключение `IndexOutOfBoundsException`.

Если опустить аргумент, функция `find()` всякий раз будет начинать поиск с позиции, следующей за предыдущим совпадением, найденным ею. Если вызвать функцию `find()` сразу после вызова метода `Pattern.matcher()` или `Matcher.reset()`, она начнет поиск с начала строки.

JavaScript

Метод `string.match()` принимает единственный аргумент с регулярным выражением. Регулярное выражение можно передавать в виде литерала регулярного выражения, в виде объекта регулярного выражения или в виде строки. Если регулярное выражение передается в виде строки, метод `string.match()` создаст временный объект `regexp`.

Если попытка сопоставления потерпит неудачу, метод `string.match()` вернет значение `null`. Это позволяет отличать случаи, когда в строке отсутствуют совпадения с регулярным выражением и когда для регулярного выражения обнаруживается совпадение с нулевой длиной. Это означает, что невозможно будет напрямую отобразить результат, так как на экране может появиться текст «null» или сообщение об ошибке обращения к несуществующему объекту.

Если попытка сопоставления увенчается успехом, метод `string.match()` вернет массив с полной информацией о совпадении. Нулевой элемент массива – это строка с текстом, совпавшим с регулярным выражением.

Регулярное выражение не должно содержать флаг `/g`. В противном случае поведение метода `string.match()` будет отличаться от описанного здесь, как объясняется в рецепте 3.10.

PHP

Функция `preg_match()`, рассматривавшаяся в двух предыдущих рецептах, принимает необязательный третий аргумент, в котором будет сохраняться текст, совпавший с регулярным выражением и с его сохраняющими группами. Если функция `preg_match()` возвращает значение `1`, в указанной переменной будет храниться массив строк. В нулевом элементе массива хранится совпадение со всем регулярным выражением. Остальные элементы массива будут описаны в рецепте 3.9.

Perl

Когда оператор `m//` сопоставления с шаблоном обнаруживает совпадение, он записывает значения в несколько специальных переменных. Одна из таких переменных, `$&`, хранит часть строки, совпавшей с регулярным выражением. Другие специальные переменные будут описаны в последующих рецептах.

Python

В рецепте 3.5 описывается функция `search()`. В этом рецепте экземпляр класса `MatchObject`, возвращаемый функцией `search()`, сохранялся в переменной. Чтобы получить часть строки, совпавшей с регулярным выражением, следует вызвать метод `group()` объекта совпадения без аргументов.

Ruby

В рецепте 3.8 описываются переменная `$~` и объект `MatchData`. В строковом контексте этот объект преобразуется в текст, совпавший с регулярным выражением. В контексте массива этот объект преобразуется в массив, нулевой элемент которого хранит совпадение со всем регулярным выражением.

$& — это специальная переменная, доступная только для чтения. Она представляет собой псевдоним для $~[0], где хранится совпадение со всем регулярным выражением.

См. также

Рецепты 3.5, 3.8, 3.9, 3.10 и 3.11.

3.8. Определение позиции и длины совпадения

Задача

Вместо извлечения подстроки, совпавшей с регулярным выражением, как показано в предыдущем рецепте, необходимо определить начальную позицию и длину совпадения. При наличии этой информации можно будет извлечь совпадение другими средствами или выполнить обработку части оригинальной строки, совпавшей с регулярным выражением.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
int matchstart, matchlength = -1;
Match matchResult = Regex.Match(subjectString, @"\d+");
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
int matchstart, matchlength = -1;
Regex regexObj = new Regex(@"\d+");
Match matchResult = regexObj.Match(subjectString).Value;
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim MatchResult = Regex.Match(SubjectString, “\d+”)
If MatchResult.Success Then
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim RegexObj As New Regex(“\d+”)
Dim MatchResult = Regex.Match(SubjectString, “\d+”)
If MatchResult.Success Then
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

Java

```
int matchStart, matchLength = -1;
Pattern regex = Pattern.compile(“\\d+”);
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    matchStart = regexMatcher.start();
    matchLength = regexMatcher.end() - matchStart;
}
```

JavaScript

```
var matchstart = -1;
var matchlength = -1;
var match = /\d+/.exec(subject);
if (match) {
    matchstart = match.index;
    matchlength = match[0].length;
}
```

PHP

```
if (preg_match(‘/\d+/’, $subject, $groups, PREG_OFFSET_CAPTURE)) {
    $matchstart = $groups[0][1];
    $matchlength = strlen($groups[0][0]);
}
```

Perl

```
if ($subject =~ m/\d+/g) {
    $matchlength = length($&);
```

```
    $matchstart = length($`);
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search(r"\d+", subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
matchobj = reobj.search(subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

Ruby

Можно использовать оператор =~ и его специальную переменную $~:

```
if subject =~ /regex pattern/
    matchstart = $~.begin()
    matchlength = $~.end() - matchstart
end
```

Как вариант, можно задействовать метод `match()` объекта `Regexp`:

```
matchobj = /regex pattern/.match(subject)
if matchobj
    matchstart = matchobj.begin()
    matchlength = matchobj.end() - matchstart
end
```

Обсуждение

.NET

Для получения позиции и длины совпадения в решении использован тот же метод `Regex.Match()`, что был описан в предыдущем рецепте. На этот раз использовались свойства `Index` и `Length` объекта `Match`, возвращаемого методом `Regex.Match()`.

Свойство `Index` хранит номер позиции в испытуемой строке, где начинается совпадение с регулярным выражением. Если совпадение начи-

нается с начала строки, значением свойства Index будет ноль. Если совпадение начинается со второго символа в строке, значением свойства Index будет один. Максимально возможное значение свойства Index равно длине строки. Такое может произойти, когда для регулярного выражения обнаруживается совпадение нулевой длины в конце строки. Например, регулярное выражение, состоящее только из якорного метасимвола ‹\Z›, всегда совпадает только в конце строки.

Свойство Length хранит число совпавших символов. Вполне допустимым считается, когда совпадение имеет нулевую длину. Например, регулярное выражение, состоящее только из метасимвола границы слова ‹\b›, будет обнаруживать совпадение нулевой длины в начале первого слова в строке.

Если попытка сопоставления не увенчалась успехом, метод Regex.Match() все равно вернет объект Match, оба свойства Index и Length которого будут равны нулю. Эти значения могут также появляться и в случае успеха сопоставления. Регулярное выражение, состоящее только из якорного метасимвола начала строки ‹\A›, будет обнаруживать совпадение нулевой длины в начале строки. Вследствие этого нельзя полагаться на свойства Match.Index и Match.Length, чтобы определить, была ли попытка сопоставления успешной. Вместо этого лучше использовать свойство Match.Success.

Java

Чтобы определить позицию и длину совпадения, следует вызвать метод Matcher.find(), как было описано в предыдущем рецепте. Если метод find() вернет значение true, можно вызвать метод Matcher.start() без аргументов, чтобы получить индекс первого символа совпадения с регулярным выражением. При вызове метода end() без аргументов возвращается индекс первого символа, следующего за совпадением. Чтобы определить длину, достаточно вычесть индекс первого символа совпадения из индекса первого символа, следующего за совпадением, которая может оказаться равной нулю. Если вызвать метод start() или end(), не вызвав перед этим метод find(), будет возбуждено исключение IllegalStateException.

JavaScript

Вызов метода exec() объекта regexp возвращает массив с полной информацией о совпадении. Этот массив обладает рядом дополнительных свойств. Свойство index хранит позицию в испытуемой строке, с которой начинается совпадение с регулярным выражением. Если совпадение начинается с начала строки, свойство index будет иметь значение ноль. Нулевой элемент массива хранит строку совпадения со всем регулярным выражением. Чтобы определить длину этой строки, можно воспользоваться ее свойством length.

Если для регулярного выражения нет ни одного совпадения в строке, метод `regexp.exec()` вернет значение `null`.

Не следует использовать свойство `lastIndex` массива, возвращаемого методом `exec()`, для определения позиции последнего символа в совпадении. В строгой реализации JavaScript свойство `lastIndex` вообще отсутствует в возвращаемом массиве, оно имеется только у самого объекта `regexp`. Но и свойство `regexp.lastIndex` тоже не следует использовать. Оно – ненадежный источник информации из-за различий между броузерами (подробнее об этом рассказывается в рецепте 3.11). Вместо этого, чтобы определить позицию, где закончилось совпадение, нужно просто сложить значения свойств `match.index` и `match[0].length`.

PHP

В предыдущем рецепте описывалось, как получить текст, совпавший с регулярным выражением, передавая третий аргумент функции `preg_match()`. Получить позицию совпадения можно, передав в четвертом аргументе значение `PREG_OFSET_CAPTURE`. Этот аргумент определяет, что именно будет сохранять функция `preg_match()` в третьем аргументе, когда она возвращает значение `1`.

Если четвертый аргумент опущен или в нем передается нулевое значение, переменная, передаваемая в третьем аргументе, принимает массив строк. Когда в четвертом аргументе передается значение `PREG_OFSET_CAPTURE`, переменная принимает массив массивов. Нулевой элемент массива все так же хранит общее совпадение (смотрите предыдущий рецепт), а первый и последующие элементы, как и прежде, хранят содержимое сохраняющих групп с номерами один и выше (смотрите следующий рецепт). Но вместо простой строки совпадения с регулярным выражением или сохраняющей группой элементы массива хранят массивы из двух значений: текст совпадения и позицию в строке, где это совпадение было обнаружено.

Так, для общего совпадения нулевой подэлемент нулевого элемента хранит текст, совпавший с регулярным выражением. Если этот текст передать функции `strlen()`, можно узнать его длину. Подэлемент с индексом 1 нулевого элемента хранит целое число, определяющее позицию в испытуемой строке, где начинается совпадение.

Perl

Чтобы получить длину совпадения, достаточно просто вычислить длину содержимого переменной `$&`, которая хранит общее совпадение с регулярным выражением. Чтобы определить позицию начала совпадения, нужно вычислить длину содержимого переменной `$\``, которая хранит текст в строке, предшествующий совпадению.

Python

Метод `start()` объекта `MatchObject` возвращает позицию в строке, где начинается совпадение с регулярным выражением. Метод `end()`возвращает позицию первого символа в строке, следующего за совпадением. Оба метода возвращают одно и то же значение в случае, когда обнаружено совпадение нулевой длины.

Методы `start()` и `end()` могут принимать аргумент и возвращать фрагмент текста, совпавшего с одной из сохраняющих групп в регулярном выражении. Например, вызов `start(1)` вернет позицию начала совпадения для первой сохраняющей группы, вызов `end(2)` – позицию конца совпадения для второй группы и так далее. Язык Python поддерживает до 99 сохраняющих групп. Группе с номером 0 соответствует совпадение со всем регулярным выражением. Передача методам `start()` и `end()` любого числа, за исключением нуля, превышающего число сохраняющих групп в регулярном выражении (которых может быть до 99), вызывает исключение `IndexError`. Если передан номер группы, существующей в регулярном выражении, но эта группа не участвовала в совпадении, методы `start()` и `end()` вернут для этой группы значение `-1`.

При желании можно сохранить начальную и конечную позиции в кортеже, вызвав метод `span()` объекта совпадения.

Ruby

В рецепте 3.5 для поиска первого совпадения с регулярным выражением использовался оператор `=~`. Попутно этот оператор записывает в специальную переменную `$~` экземпляр класса `MatchData`. Эта переменная является локальной для потоков выполнения и локальной для методов. Это означает, что содержимое этой переменной может использоваться только в пределах метода и пока в этом методе снова не будет использован оператор `=~`; при этом можно не беспокоиться, что содержимое переменной будет затерто в каком-нибудь другом потоке выполнения или каким-нибудь другим методом в этом потоке выполнения.

При желании можно сохранять информацию о нескольких совпадениях с регулярными выражениями, вызывая метод `match()` объекта `Regexp`. Этот метод принимает испытуемую строку в виде единственного аргумента. Если совпадение было найдено, метод возвращает экземпляр класса `MatchData` или значение `nil` – в противном случае. Кроме того, он сохраняет тот же самый экземпляр `MatchData` в специальной переменной `$~`, но не затирает другие экземпляры класса `MatchData`, хранящиеся в других переменных. Объект класса `MatchData` содержит полную информацию о совпадении с регулярным выражением. В рецептах с 3.7 по 3.9 описывается, как извлечь текст, совпавший с регулярным выражением и с сохраняющими группами.

Метод `begin()` возвращает позицию в испытуемой строке, где начинается совпадение с регулярным выражением. Метод `end()` возвращает по-

зицию первого символа в испытуемой строке, следующего за совпадением. Метод `offset()` возвращает массив начальных и конечных позиций. Все три метода принимают единственный аргумент. Чтобы получить позиции, относящиеся к совпадению со всем регулярным выражением, следует передать значение `0`. Если передается положительное число, возвращаются позиции, относящиеся к совпадению с указанной сохраняющей группой. Например, вызов `begin(1)` вернет начало совпадения первой сохраняющей группы.

Не следует использовать методы `length()` и `size()`, чтобы получить длину совпадения. Оба эти метода возвращают число элементов массива, в который преобразуется объект класса `MatchData` в контексте, когда ожидается массив, как будет описано в рецепте 3.9.

См. также

Рецепты 3.5 и 3.9.

3.9. Извлечение части совпавшего текста

Задача

Как и в рецепте 3.7, имеется регулярное выражение, совпадающее с подстрокой в испытуемом тексте, но на этот раз необходимо извлечь только одну часть совпадения из этой подстроки. Чтобы отделить части совпадений друг от друга, в регулярном выражении используется сохраняющая группировка, как описано в рецепте 2.9.

Например, в строке `Please visit http://www.regexcookbook.com for more information` регулярному выражению `‹http://([a-z0-9.-]+)›` соответствует фрагмент `http://www.regexcookbook.com`. Части регулярного выражения, заключенной в первую сохраняющую группу, соответствует фрагмент `www.regexcookbook.com`. Необходимо извлечь это доменное имя, сохраненное первой сохраняющей группой, в строковую переменную.

Для иллюстрации принципов работы с сохраняющими группами мы будем использовать это простое регулярное выражение. В главе 7 приводится более точное регулярное выражение, которое может использоваться для поиска адресов URL.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString,
                      "http://([a-z0-9.-]+)").Groups[1].Value;
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex("http://([a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups[1].Value;
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString,
                   "http://([a-z0-9.-]+)").Groups(1).Value
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("http://([a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups(1).Value
```

Java

```
String resultString = null;
Pattern regex = Pattern.compile("http://([a-z0-9.-]+)");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group(1);
}
```

JavaScript

```
var result = "";
var match = /http:\/\/([a-z0-9.-]+)/.exec(subject);
if (match) {
    result = match[1];
} else {
    result = '';
}
```

PHP

```
if (preg_match('%http://([a-z0-9.-]+)%', $subject, $groups)) {
    $result = $groups[1];
} else {
    $result = '';
}
```

Perl

```
if ($subject =~ m!http://([a-z0-9.-]+)!) {
    $result = $1;
} else {
```

```
    $result = '';
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search("http://([a-z0-9.-]+)", subject)
if matchobj:
    result = matchobj.group(1)
else:
    result = ""
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("http://([a-z0-9.-]+)")
matchobj = reobj.search(subject)
if match:
    result = matchobj.group(1)
else:
    result = ""
```

Ruby

Можно использовать оператор =~ и его специальные нумерованные переменные, такие как $1:

```
if subject =~ %r!http://([a-z0-9.-]+)!
    result = $1
else
    result = ""
end
```

Как вариант, можно задействовать метод `match()` объекта `Regexp`:

```
matchobj = %r!http://([a-z0-9.-]+)!.match(subject)
if matchobj
    result = matchobj[1]
else
    result = ""
end
```

Обсуждение

В рецептах 2.10 и 2.21 описывалось, как использовать нумерованные обратные ссылки в регулярных выражениях и в замещающем тексте, чтобы можно было описать совпадение с тем же текстом еще раз или вставить часть совпадения с регулярным выражением в замещающий

текст. Те же самые номера ссылок можно использовать в программном коде, чтобы извлекать совпадения с сохраняющими группами.

Нумерация сохраняющих групп в регулярных выражениях ведется начиная с единицы. В языках программирования нумерация элементов списков или массивов обычно начинается с нуля. Во всех языках программирования, описываемых в этой книге, где сохраняющие группы запоминаются в массиве или в списке, используется та же нумерация сохраняющих групп, что и в регулярном выражении, то есть начиная с единицы. Нулевой элемент в массиве или в списке используется для сохранения совпадения со всем регулярным выражением. Это означает, что если в регулярном выражении имеется три сохраняющих группы, массив, хранящий совпадения с ними, будет иметь четыре элемента. Нулевой элемент будет хранить общее совпадение со всем регулярным выражением, а первый, второй и третий элементы – текст, совпавший с тремя сохраняющими группами.

.NET

Чтобы извлечь информацию о сохраняющих группах, потребуется обратиться к одной из версий функции-члена `Regex.Match()`, впервые описанной в рецепте 3.7. Возвращаемый ею объект `Match` имеет свойство `Groups`. Это свойство представляет собой коллекцию типа `GroupCollection`. В коллекции хранится информация обо всех сохраняющих группах, присутствующих в регулярном выражении. Элемент `Groups[1]` хранит информацию о первой сохраняющей группе, элемент `Groups[2]` – о второй группе, и так далее.

Коллекция `Groups` хранит по одному объекту `Group` для каждой сохраняющей группы. Класс `Group` имеет те же свойства, что и класс `Match`, за исключением свойства `Groups`. При обращении к `Match.Groups[1].Value` возвращается текст, совпавший с первой сохраняющей группой, точно так же, как при обращении к `Match.Value` возвращается совпадение со всем регулярным выражением. При обращении к `Match.Groups[1].Index` и `Match.Groups[1].Length` возвращаются начальная позиция и длина текста, совпавшего с группой. Подробнее о свойствах `Index` и `Length` говорится в рецепте 3.8.

Элемент `Groups[0]` хранит информацию о совпадении со всем регулярным выражением, которая также хранится в самом объекте совпадения. Свойства `Match.Value` и `Match.Groups[0].Value` полностью эквивалентны.

Коллекция `Groups` не возбуждает исключение при использовании ошибочного номера группы. Например, при обращении к элементу `Groups[-1]` также будет получен объект `Group`, но свойства этого объекта будут свидетельствовать об отсутствии совпадения с этой несуществующей сохраняющей группой под номером `-1`. Лучший способ проверить это заключается в использовании свойства `Success`. При обращении к `Groups[-1].Success` будет возвращено значение `false`.

Чтобы определить количество имеющихся сохраняющих групп, следует проверить свойство `Match.Groups.Count`. Свойство `Count` следует тем же соглашениям, что и свойства `Count` коллекций всех типов в платформе .NET: при обращении к нему возвращается число элементов в коллекции, которое определяется, как наибольший индекс плюс один. В данном примере коллекция Groups хранит элементы `Groups[0]` и `Groups[1]`. А свойство `Groups.Count` имеет значение `2`.

Java

Программный код, извлекающий текст совпадения с сохраняющей группой или информацию о сохраняющей группе, практически не отличается от программного кода, показанного в предыдущих примерах, извлекающего ту же информацию о совпадении со всем регулярным выражением. Методы `group()`, `start()` и `end()` класса `Matcher` принимают по одному необязательному аргументу. Без этого аргумента, или когда в нем передается значение ноль, они возвращают совпадение или позиции, относящиеся ко всему регулярному выражению.

При передаче положительного числа возвращается информация о соответствующей сохраняющей группе. Нумерация групп начинается с единицы, так же как и нумерация обратных ссылок в самом регулярном выражении. Если указать число, превышающее число сохраняющих групп в регулярном выражении, все три функции возбудят исключение `IndexOutOfBoundsException`. Если сохраняющая группа с указанным номером присутствует в регулярном выражении, но она не участвовала в сопоставлении, метод `group(n)` вернет значение `null`, а методы `start(n)` и `end(n)` вернут `-1`.

JavaScript

Как описывалось в предыдущем рецепте, метод `exec()` объекта регулярного выражения возвращает массив с информацией о совпадении. Нулевой элемент этого массива хранит совпадение со всем регулярным выражением. Первый элемент хранит текст, совпавший с первой сохраняющей группой, второй элемент – со второй сохраняющей группой, и так далее.

Если в строке отсутствует совпадение с регулярным выражением, метод `regexp.exec()` вернет значение `null`.

PHP

В рецепте 3.7 описывалось, как получить текст, совпавший с регулярным выражением, передавая третий аргумент функции `preg_match()`. Когда функция `preg_match()` возвращает значение `1`, в этот аргумент записывается массив. Нулевой элемент этого массива хранит строку совпадения со всем регулярным выражением.

Первый элемент хранит текст, совпавший с первой сохраняющей группой, второй элемент – со второй сохраняющей группой и так далее. Длина массива равна числу сохраняющих групп плюс один. Индексы в массиве соответствуют номерам обратных ссылок в регулярном выражении.

Если в четвертом аргументе функции `preg_match()` передать константу `PREG_OFFSET_CAPTURE`, как описывалось в предыдущем рецепте, длина массива по-прежнему будет равна числу сохраняющих групп плюс один. Но вместо строк в каждом элементе массива будет храниться подмассив из двух элементов. Нулевой подэлемент хранит строку с текстом совпадения для всего регулярного выражения или сохраняющей группы. Первый подэлемент хранит целое число, определяющее позицию начала совпадения в испытуемой строке.

Perl

Когда оператор `m//` сопоставления с шаблоном находит совпадение, он устанавливает значения нескольких специальных переменных. В число таких переменных входят `$1`, `$2`, `$3` и так далее, которые хранят части строки, совпавшие с сохраняющими группами в регулярном выражении.

Python

Решение этой задачи во многом идентично решению, что приводится в рецепте 3.7. Только вместо вызова метода `group()` без параметров следует указать номер интересующей сохраняющей группы. Вызов `group(1)` вернет текст, совпавший с первой сохраняющей группой, `group(2)` – со второй группой, и так далее. В языке Python поддерживается до 99 сохраняющих групп. Группа с номером 0 представляет совпадение со всем регулярным выражением. Если передать число, превышающее количество сохраняющих групп в регулярном выражении, метод `group()` возбудит исключение `IndexError`. Если группа с указанным номером существует, но она не участвовала в сопоставлении, метод `group()` вернет значение `None`.

Имеется возможность передавать методу `group()` сразу несколько номеров сохраняющих групп, чтобы получить совпадения с несколькими группами за одно обращение. В этом случае результатом работы метода будет список строк.

Если необходимо получить кортеж с текстами совпадений для всех сохраняющих групп, можно воспользоваться методом `groups()` объекта `MatchObject`. Кортеж будет хранить значения `None` для тех групп, которые не участвовали в сопоставлении. Если передать методу `groups()` некоторое значение, оно будет использовано вместо значения `None` для групп, не участвовавших в сопоставлении.

Если вместо кортежа требуется получить словарь совпадений с сохраняющими группами, вместо метода groups() можно вызвать метод groupdict(). Если передать методу groupdict() некоторое значение, оно будет помещаться в словарь вместо значения None для групп, не участвовавших в сопоставлении.

Ruby

В рецепте 3.8 описывались переменная $~ и объект MatchData. В контексте, где ожидается массив, этот объект интерпретируется как массив совпадений со всеми сохраняющими группами, имеющимися в регулярном выражении. Нумерация сохраняющих групп начинается с 1, как и обратных ссылок в регулярном выражении. Нулевой элемент массива хранит совпадение со всем регулярным выражением.

Переменные $1, $2 и так далее — это специальные переменные, доступные только для чтения. $1 — это сокращенная форма записи для $~[1]; в этой переменной хранится текст совпадения с первой сохраняющей группой. В переменной $2 хранится текст совпадения со второй группой и так далее.

Именованное сохранение

Если в регулярном выражении используются именованные сохраняющие группы, то для получения совпадений в программном коде можно использовать их имена.

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString,
                    "http://(?<domain>[a-z0-9.-]+)").Groups["domain"].Value;
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex("http://(?<domain>[a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups["domain"].Value;
```

В языке C# нет отличий между способом получения объекта Group для нумерованной и для именованной группы. В последнем случае коллекция Groups индексируется не целыми числами, а строками. Кроме того, в данном случае платформа .NET не будет возбуждать исключение, если указанная группа не существует. При обращении Match.Groups["nosuchgroup"].Success просто будет возвращено значение false.

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString,
                   "http://(?<domain>[a-z0-9.-]+)").Groups("domain").Value
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("http://(?<domain>[a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups("domain").Value
```

В языке VB.NET нет отличий между способом получения объекта `Group` для нумерованной и для именованной группы. В последнем случае коллекция `Groups` индексируется не целыми числами, а строками. Кроме того, в данном случае платформа .NET не будет возбуждать исключение, если указанная группа не существует. При обращении `Match.Groups["nosuchgroup"].Success` просто будет возвращено значение `false`.

PHP

```
if (preg_match('%http://(?P<domain>[a-z0-9.-]+)%', $subject, $groups)) {
    $result = $groups['domain'];
} else {
    $result = '';
}
```

Если в регулярном выражении имеются именованные сохраняющие группы, переменной `$groups` будет присвоен ассоциативный массив. Текст совпадения для каждой сохраняющей группы будет добавлен в массив дважды. Извлекать совпадения из массива можно с использованием как номеров групп, так и их имен. В примере программного кода элемент `$groups[0]` хранит совпадение со всем регулярным выражением, тогда как элементы `$groups[1]` и `$groups['domain']` хранят текст совпадения с сохраняющей группой, единственной в регулярном выражении.

Perl

```
if ($subject =~ '!http://(?<domain>[a-z0-9.-]+)%!) {
    $result = $+{'domain'};
} else {
    $result = '';
}
```

Именованная сохраняющая группировка поддерживается в языке Perl, начиная с версии 5.10. В хеше `$+` хранятся тексты совпадений со всеми именованными сохраняющими группами. В языке Perl именованные

группы нумеруются вместе с нумерованными группами. В данном примере переменные $1 и $+{name} хранят текст совпадения с сохраняющей группой, единственной в регулярном выражении.

Python

```
matchobj = re.search("http://(?P<domain>[a-z0-9.-]+)", subject)
if matchobj:
    result = matchobj.group("domain")
else:
    result = ""
```

Если в регулярном выражении имеются именованные сохраняющие группы, то вместо номера методу group() можно передавать имена групп.

См. также

Рецепт 2.9, где описываются нумерованные сохраняющие группы.

Рецепт 2.11, где описываются именованные сохраняющие группы.

3.10. Извлечение списка всех совпадений

Задача

Во всех предыдущих рецептах этой главы речь шла только о первом совпадении, обнаруженном регулярным выражением в испытуемой строке. Но во многих случаях регулярное выражение, совпадающее с частью строки, может обнаруживать еще одно совпадение в оставшейся части строки. А за вторым совпадением может следовать третье и так далее. Например, в строке The lucky numbers are 7, 13, 16, 42, 65, and 99 регулярное выражение ‹\d+› может обнаружить шесть совпадений: 7, 13, 16, 42, 65 и 99.

Необходимо извлечь список всех подстрок, которые обнаруживаются регулярным выражением при многократном применении его к фрагменту строки, остающемуся после каждого очередного совпадения.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
MatchCollection matchlist = Regex.Matches(subjectString, @"\d+");
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex(@"\d+");
MatchCollection matchlist = regexObj.Matches(subjectString);
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim matchlist = Regex.Matches(SubjectString, "\d+")
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("\d+")
Dim MatchList = RegexObj.Matches(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    resultList.add(regexMatcher.group());
}
```

JavaScript

```
var list = subject.match(/\d+/g);
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
$result = $result[0];
```

Perl

```
@result = $subject =~ m/\d+/g;
```

Этот прием работает только, если регулярное выражение не содержит сохраняющих групп, поэтому в подобных случаях следует использовать несохраняющие группы. Подробности приводятся в рецепте 2.9.

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.findall(r"\d+", subject)
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
result = reobj.findall(subject)
```

Ruby

```
result = subject.scan(/\d+/)
```

Обсуждение

.NET

Метод Matches() класса Regex многократно применяет регулярное выражение к строке, пока не будут найдены все совпадения. Он возвращает объект MatchCollection, в котором хранятся все совпадения. Первым аргументом всегда передается испытуемая строка. В этой строке будут разыскиваться совпадения с регулярным выражением. Первый аргумент не может иметь значение null. В противном случае метод Matches() возбудит исключение ArgumentNullException.

Если требуется выполнить поиск совпадений с регулярным выражением в небольшом числе строк, можно использовать статическую перегруженную версию метода Matches(). В первом аргументе этому методу передается испытуемая строка, а во втором – регулярное выражение. Параметры регулярного выражения можно передавать в третьем необязательном аргументе.

Если предстоит обработать большое число строк, предпочтительнее будет сначала создать объект Regex и затем использовать его для вызова метода Matches(). Единственным обязательным аргументом для этого метода является испытуемая строка. Во втором необязательном аргументе можно передать номер позиции символа, с которого должны начинаться попытки сопоставления с регулярным выражением. Фактически число, указываемое во втором аргументе, – это количество символов от начала испытуемой строки, которые должны игнорироваться регулярным выражением. Это может быть удобно, когда часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть между нулем и длиной строки. В противном случае метод Matches()возбудит исключение ArgumentOutOfRangeException.

Статические перегруженные версии методов не имеют аргумента, с помощью которого можно было бы указать, с какой позиции в строке следует начинать поиск совпадения. Не существует перегруженной версии метода Matches(), которая позволяла бы указывать, в какой позиции перед концом строки следует останавливать поиск. В случае необходимости можно вызывать метод Regex.Match("subject", start, stop) в цикле, как показано в следующем рецепте, и вручную добавлять в список все найденные совпадения.

Java

В языке Java отсутствует функция, которая извлекала бы список совпадений. Но ее легко можно реализовать, адаптировав решение из ре-

цепта 3.7. Для этого вызов метода find() нужно производить не в условном операторе if, а в цикле while.

Чтобы использовать классы List и ArrayList, как в этом примере, нужно добавить инструкцию import java.util.*; в начало файла с программным кодом.

JavaScript

В данном решении вызывается метод string.match(), точно так же, как в решении к рецепту 3.7. Но здесь есть одно маленькое, но важное отличие: флаг /g. Подробнее о флагах регулярных выражений рассказывается в рецепте 3.4.

Флаг /g предписывает функции match() выполнить итерации по всем совпадениям в строке и поместить их в массив. В приведенном фрагменте программного кода элемент list[0] будет хранить первое совпадение с регулярным выражением, list[1] – второе и так далее. Узнать количество совпадений можно с помощью свойства list.length. В случае отсутствия совпадений метод string.match() вернет значение null, как обычно.

Элементами этого массива являются строки. При использовании флага /g метод string.match() не предоставляет никакой дополнительной информации о совпадениях с регулярным выражением. Если необходимо получить дополнительные сведения о совпадениях, следует выполнить итерации по совпадениям, как описывается в рецепте 3.11.

PHP

Во всех решениях на языке PHP в предыдущих рецептах использовалась функция preg_match(), которая отыскивает первое совпадение с регулярным выражением в строке. Функция preg_match_all()очень похожа на нее. Главное ее отличие состоит в том, что она отыскивает в строке все совпадения и возвращает целое число, обозначающее число найденных совпадений с регулярным выражением.

Первые три аргумента функции preg_match_all() идентичны первым трем аргументам функции preg_match(): строка с регулярным выражением, строка, в которой следует произвести поиск, и переменная, в которую будет записан массив с результатами. Единственное отличие состоит в том, что третий аргумент является обязательным, а массив всегда многомерный.

В четвертом аргументе допускается передавать либо значение константы PREG_PATTERN_ORDER, либо PREG_SET_ORDER. Если опустить четвертый аргумент, по умолчанию будет использоваться значение PREG_PATTERN_ORDER.

При использовании значения PREG_PATTERN_ORDER будет возвращаться массив, хранящий информацию об общем совпадении в нулевом элементе, а информацию о сохраняющих группах в первом и последующих элементах. Массив будет иметь длину, равную числу сохраня-

ющих групп плюс один. Тот же порядок построения массива используется в функции preg_match(). Разница заключается в том, что вместо каждого элемента, хранящего строку с единственным совпадением, найденным функцией preg_match(), используются подмассивы со всеми совпадениями, найденными функцией preg_match_all(). Длина каждого подмассива совпадает со значением, возвращаемым функцией preg_match_all().

Чтобы получить список всех совпадений с регулярным выражением в строке, отбросив совпадения с сохраняющими группами, следует указать значение PREG_PATTERN_ORDER и извлечь нулевой элемент массива. Если интерес представляет только текст совпадения с определенной сохраняющей группой, можно использовать значение PREG_PATTERN_ORDER и номер сохраняющей группы. Например, после вызова preg_match_all('%http://([a-z0-9.-]+)%', $subject, $result) элемент $result[1] будет хранить список доменных имен всех адресов URL, имеющихся в испытуемой строке.

Если указано значение PREG_SET_ORDER, массив заполняется теми же строками, но несколько иным способом. Длина массива равна значению, возвращаемому функцией preg_match_all(). Каждый элемент массива представляет собой подмассив, нулевой элемент которого содержит совпадение со всем регулярным выражением, а первый и последующие элементы – совпадения с сохраняющими группами. Когда указано значение PREG_SET_ORDER, элемент $result[0] будет хранить тот же самый массив, что возвращает функция preg_match().

Допускается комбинировать флаг PREG_OFFSET_CAPTURE с флагом PREG_PATTERN_ORDER или PREG_SET_ORDER. В этом случае возникает тот же эффект, что и при передаче флага PREG_OFFSET_CAPTURE функции preg_match() в четвертом аргументе. То есть в каждом элементе массива содержится не строка, а массив из двух элементов, в которых хранятся строка совпадения и смещение этого совпадения от начала испытуемой строки.

Perl

В рецепте 3.4 говорилось, что необходимо добавить модификатор /g, чтобы обеспечить возможность поиска в испытуемой строке нескольких совпадений с регулярным выражением. При использовании регулярного выражения с модификатором /g в списочном контексте оно будет отыскивать и возвращать все совпадения в испытуемой строке. В данном рецепте списочный контекст обеспечивается переменной-списком, расположенной слева от оператора присваивания.

Если в регулярном выражении отсутствуют какие-либо сохраняющие группы, список будет содержать только совпадения со всем регулярным выражением. Если в регулярном выражении присутствуют сохраняющие группы, список будет содержать совпадения со всеми сохраняющими группами для каждого совпадения с регулярным выражением. Совпадение со всем регулярным выражением не включает-

ся в список, если выражение целиком не заключить в сохраняющую группу. Если необходимо получить список всех совпадений с регулярным выражением целиком, все сохраняющие группы следует заменить несохраняющими. Обе разновидности группировки описываются в рецепте 2.9.

Python

Поиск всех совпадений с регулярным выражением выполняет функция `findall()` из модуля `re`. Первым аргументом функции передается регулярное выражение, а вторым – испытуемая строка. Параметры регулярного выражения можно передать функции в третьем необязательном аргументе.

Функция `re.findall()` вызывает функцию `re.compile()`, после чего вызывает метод `findall()` объекта скомпилированного регулярного выражения. Этот метод принимает единственный обязательный аргумент: испытуемую строку.

Метод `findall()` имеет два необязательных аргумента, которые не поддерживаются глобальной функцией `re.findall()`. Вслед за испытуемой строкой можно передать позицию символа в строке, с которого метод `findall()` должен начинать поиск. Если опустить этот аргумент, поиск будет выполняться методом `findall()` во всей строке. Если указана начальная позиция поиска, можно также определить конечную позицию. Если конечная позиция не определена, поиск будет выполняться до конца строки.

Независимо от того, с какими аргументами будет вызываться метод `findall()`, в результате всегда будет возвращаться список всех найденных совпадений. Если в регулярном выражении имеются сохраняющие группы, метод вернет список кортежей, содержащих совпадения для всех сохраняющих групп для каждого совпадения с регулярным выражением.

Ruby

Метод `scan()` класса `String` принимает регулярное выражение в виде единственного аргумента. Он отыскивает в строке все совпадения с регулярным выражением. Когда метод `scan()` вызывается без блока, он возвращает массив всех совпадений с регулярным выражением.

Если в регулярном выражении отсутствуют сохраняющие группы, метод `scan()` возвратит массив строк. Каждый элемент этого массива соответствует совпадению с регулярным выражением и содержит текст совпадения.

При наличии сохраняющих групп метод `scan()` вернет массив массивов. Каждый элемент этого массива соответствует совпадению с регулярным выражением и представляет собой массив с совпадениями для всех сохраняющих групп. В нулевом подэлементе будет храниться со-

впадение с первой сохраняющей группой, в первом – со второй сохраняющей группой и так далее. Совпадения со всем регулярным выражением не включаются в массив. Если необходимо включить совпадения со всем регулярным выражением, следует все регулярное выражение заключить в одну сохраняющую группу.

В языке Ruby отсутствует возможность заставить метод `scan()` возвращать массив строк, когда регулярное выражение содержит сохраняющие группы. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы на несохраняющие.

См. также

Рецепты 3.7, 3.11 и 3.12.

3.11. Обход всех совпадений в цикле

Задача

В предыдущем рецепте рассказывалось, как многократно применить регулярное выражение к строке, чтобы получить список совпадений. Теперь требуется выполнить обход всех совпадений в цикле.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной matchResult
    matchResult = matchResult.NextMatch();
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной matchResult
    matchResult = matchResult.NextMatch();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
    'Здесь можно выполнить обработку очередного совпадения,
    'хранящегося в переменной matchResult
    MatchResult = MatchResult.NextMatch
End While
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
    'Здесь можно выполнить обработку очередного совпадения,
    'хранящегося в переменной matchResult
    MatchResult = MatchResult.NextMatch
End While
```

Java

```
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в объекте regexMatcher
}
```

JavaScript

Если совпадения с регулярным выражением могут иметь нулевую длину или когда это заранее неизвестно, следует предусмотреть обработку проблем, связанных с различной для разных броузеров реакцией метода exec() на совпадения с нулевой длиной:

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Не дать возможность таким броузерам, как Firefox, попасть
    // в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной match
}
```

Если заранее известно, что для регулярного выражения никогда не будут обнаруживаться совпадения нулевой длины, можно выполнять итерации непосредственно по регулярному выражению:

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной match
}
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($result[0]); $i++) {
    # Текст совпадения = $result[0][$i];
}
```

Perl

```
while ($subject =~ m/\d+/g) {
    # текст совпадения = $&
}
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
for matchobj in re.finditer(r"\d+", subject):
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной matchobj
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной matchobj
```

Ruby

```
subject.scan(/\d+/) {|match|
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной match
}
```

Обсуждение

.NET

В рецепте 3.7 описывалось, как с помощью функции-члена Match() класса Regex получить первое совпадение с регулярным выражением. Чтобы обойти в цикле все совпадения, имеющиеся в строке, необходимо снова вызвать функцию Match() для получения информации о первом совпадении. Функция Match() возвращает экземпляр класса Match, который в представленном решении сохраняется в переменной matchResult. Если свойство Success объекта matchResult содержит значение true, можно начинать итерации.

В начале цикла можно использовать свойства объекта Match для получения информации о первом совпадении. В рецепте 3.7 описывается свойство Value, в рецепте 3.8 – свойства Index и Length и в рецепте 3.9 – свойство-коллекция Groups.

Закончив работу с первым совпадением, можно вызвать функцию-член NextMatch() относительно переменной matchResult. Функция Match.NextMatch() возвращает экземпляр класса Match, как и функция Regex.Match(). Вновь созданный экземпляр хранит информацию о втором совпадении.

Присваивание результата, полученного при вызове matchResult.NextMatch(), одной и той же переменной matchResult упрощает обход всех совпадений. Каждый раз в цикле выполняется проверка значений свойства matchResult.Success, чтобы убедиться, что функция NextMatch() действительно обнаружила еще одно совпадение. Когда функция NextMatch() терпит неудачу, она все равно возвращает объект Match, но в этом случае его свойство Success будет иметь значение false. Используя одну и ту же переменную matchResult, нам удалось совместить в одной инструкции while проверку наличия первого совпадения с проверками наличия всех последующих совпадений после обращений к функции NextMatch().

Функция NextMatch() не изменяет объект Match, относительно которого она вызывается. В случае необходимости можно сохранять объекты Match для каждого совпадения с регулярным выражением.

Метод NextMatch() не имеет аргументов. Он использует то же регулярное выражение и испытуемую строку, которая передавались методу Regex.Match(). Объект Match сохраняет ссылки на регулярное выражение и испытуемую строку.

Статическую версию метода Regex.Match() можно использовать, даже когда испытуемая строка содержит очень большое число совпадений с регулярным выражением. Метод Regex.Match() скомпилирует регулярное выражение всего один раз, а возвращаемый объект Match будет хранить ссылку на скомпилированное регулярное выражение. Метод Match.NextMatch()использует это скомпилированное ранее регулярное

выражение, на которое ссылается объект Match, даже если вызывался статический метод Regex.Match(). Создавать экземпляр класса Regex действительно необходимо только в том случае, если метод Regex.Match() требуется вызывать многократно, например чтобы применить одно и то же регулярное выражение к большому числу строк.

Java

Обход всех совпадений в языке Java реализуется очень просто. Достаточно просто в цикле while вызывать метод find(), представленный в рецепте 3.7. Каждый вызов метода find() обновляет информацию, хранящуюся в объекте Matcher, сведениями о совпадении и о позиции, откуда должна начинаться следующая попытка сопоставления.

JavaScript

Если возникнет потребность использовать регулярное выражение в цикле, не забудьте указать флаг /g. Назначение этого флага разъяснялось в рецепте 3.4. Цикл while (regexp.exec()) отыскивает все числа в испытуемой строке, когда regexp = /\d+/g. Если бы регулярное выражение имело вид /\d+/, то цикл while (regexp.exec()) находил бы первое число в строке снова и снова, пока не произошло бы аварийное завершение сценария или пока сценарий не был бы принудительно завершен броузером.

Примечательно, что цикл while (/\d+/g.exec()) (в котором используется литерал регулярного выражения с флагом /g) также превратился бы в бесконечный цикл, по крайней мере, в некоторых реализациях JavaScript, потому что регулярное выражение компилировалось бы заново в каждой итерации цикла while. Когда выполняется компиляция регулярного выражения, начальная позиция для предстоящей попытки сопоставления сбрасывается в начало строки. Присваивание регулярного выражения переменной за пределами цикла гарантирует, что оно будет скомпилировано всего один раз.

Объект, возвращаемый методом regexp.exec(), описывается в рецептах 3.8 и 3.9. Это один и тот же объект, даже когда метод exec() вызывается в цикле. Допускается делать с этим объектом все, что потребуется.

Единственный эффект действия флага /g заключается в обновлении значения свойства lastIndex объекта regexp, чей метод exec() вызывается. Это эффект действует даже при использовании литерала регулярного выражения, как показано во втором решении для этого рецепта. При каждом последующем обращении к методу exec() попытка сопоставления начинается с позиции lastIndex. Если присвоить свойству lastIndex новое значение, попытка сопоставления начнется с указанной позиции.

Однако со свойством lastIndex связана одна очень существенная проблема. Если понимать стандарт ECMA-262v3 для JavaScript буквально, то метод exec() должен записывать в свойство lastIndex позицию перво-

го символа, следующего за совпадением. Это означает, что когда совпадение имеет нулевую длину, следующая попытка сопоставления должна начинаться с позиции, где только что было найдено совпадение, что в результате приводит к бесконечному циклу.

Все механизмы регулярных выражений, рассматриваемые в этой книге (за исключением JavaScript), решают эту проблему, автоматически начиная следующую попытку сопоставления со следующего символа в строке, если предыдущее совпадение имело нулевую длину. Именно по этой причине в рецепте 3.8 было дано предостережение от использования свойства `lastIndex` для определения конца совпадения, так как в Internet Explorer оно дает некорректное значение.

Разработчики Firefox несмотря ни на что неуклонно следуют буквальному толкованию требований стандарта ECMA-262v3, даже если соблюдение этих требований приведет к тому, что обращения к методу `regexp.exec()` в результате попадут в бесконечный цикл. А ведь такой исход весьма вероятен. Например, для обхода всех строк в многострочном тексте можно использовать инструкции `re = /^.*$/gm; while (re.exec())`, но если в тексте имеется пустая строка, броузер Firefox «споткнется» о нее.

Решение проблемы состоит в том, чтобы увеличивать на 1 значение свойства `lastIndex`, если функция `exec()` этого еще не сделала. Как это делается, показано в первом решении для JavaScript. Если вы не уверены в том, какой выбор сделать, просто вставьте эту строку программного кода и вы будете застрахованы от этой проблемы.

Данная проблема не наблюдается при использовании метода `string.match()` (рецепт 3.14) или при поиске всех совпадений с помощью метода `string.replace()` (рецепт 3.10). Для этих методов, внутренняя реализация которых использует свойство `lastIndex`, стандарт ECMA-262v3 оговаривает, что значение `lastIndex` должно увеличиваться на 1 при каждом совпадении нулевой длины.

PHP

Функция `preg_match()` принимает пятый необязательный аргумент, определяющий позицию в строке, откуда следует начинать попытки сопоставления. Для решения поставленной задачи можно было бы адаптировать решение из рецепта 3.8 и передавать функции `preg_match()` сумму `$matchstart + $matchlength` в виде пятого аргумента при попытке найти второе совпадение в строке, и продолжать повторять то же самое в третьей и последующих попытках, пока `preg_match()` не вернет `0`. Этот прием используется в рецепте 3.18.

В дополнение к необходимости вычислять в программном коде начальное смещение для каждой попытки сопоставления многократное обращение к функции `preg_match()` само по себе неэффективно – из-за отсутствия возможности сохранять скомпилированное регулярное выражение в переменной. Функция `preg_match()` вынуждена искать

скомпилированное регулярное выражение в своем кэше при каждом вызове.

Более простое и более эффективное решение заключается в использовании функции `preg_match_all()`, как описано в предыдущем рецепте, и выполнении итераций в массиве с результатами сопоставления.

Perl

В рецепте 3.4 говорилось, что в регулярное выражение необходимо добавить модификатор `/g`, чтобы отыскать в испытуемой строке все совпадения. Если регулярное выражение с модификатором `/g` используется в скалярном контексте, следующая попытка сопоставления будет начинаться в конце предыдущего совпадения. В этом рецепте инструкция `while` обеспечивает скалярный контекст. Все специальные переменные, такие как `$&` (описываются в рецепте 3.7), существуют только в области видимости цикла `while`.

Python

Функция `finditer()` из модуля `re` возвращает итератор, который может использоваться для поиска всех совпадений с регулярным выражением. Первым аргументом эта функция принимает регулярное выражение, а вторым – испытуемую строку. Параметры регулярного выражения могут передаваться в третьем, необязательном аргументе.

Функция `re.finditer()` вызывает функцию `re.compile()`, после чего вызывает метод `finditer()` объекта скомпилированного регулярного выражения. Этот метод имеет единственный обязательный аргумент: испытуемую строку.

Метод `finditer()` принимает два необязательных аргумента, которые не поддерживаются глобальной функцией `re.finditer()`. Вслед за аргументом с испытуемой строкой можно передать позицию символа в строке, откуда метод `finditer()` должен начинать поиск. Если этот аргумент отсутствует, поиск будет выполняться во всей строке. Если указана начальная позиция, то можно указать и конечную позицию. Если конечная позиция не определена, поиск выполняется до конца строки.

Ruby

Метод `scan()` класса `String` принимает регулярное выражение в виде единственного аргумента и выполняет обход всех совпадений с регулярным выражением в строке. При вызове с блоком появляется возможность обрабатывать каждое совпадение, как только оно будет найдено.

Если регулярное выражение не содержит сохраняющих групп, в блоке следует указать только одну переменную цикла. Эта переменная будет принимать текст совпадения с регулярным выражением.

Если регулярное выражение содержит одну или более сохраняющих групп, необходимо указать по одной переменной для каждой группы. Первая переменная будет принимать текст совпадения с первой сохраняющей группой, вторая переменная – со второй сохраняющей группой и так далее. Ни в одну из переменных не будет записываться совпадение со всем регулярным выражением. Если необходимо включить совпадение со всем регулярным выражением, его следует заключить в дополнительную сохраняющую группу.

```
subject.scan(/(a)(b)(c)/) {|a, b, c|
    # a, b и c хранят совпадения с тремя сохраняющими группами
}
```

Если переменных указано меньше, чем сохраняющих групп в регулярном выражении, то получить доступ можно будет только к тем сохраняющим группам, для которых имеются переменные. Если переменных указано больше, чем сохраняющих групп, в лишние переменные будет записываться значение `nil`.

Если указана только одна переменная цикла, тогда как в регулярном выражении имеется одна или более сохраняющих групп, переменная будет заполняться массивом строк. Каждая строка в массиве будет соответствовать сохраняющей группе. Если в выражении имеется всего одна сохраняющая группа, массив будет состоять из единственного элемента:

```
subject.scan(/(a)(b)(c)/) {|abc|
    # в элементах abc[0], abc[1] и abc[2] хранится текст
    # совпадений с тремя сохраняющими группами
}
```

См. также

Рецепты 3.7, 3.8, 3.10 и 3.12.

3.12. Проверка полученных совпадений в программном коде

Задача

В рецепте 3.10 показано, как получить список всех совпадений с регулярным выражением, применяя его к фрагменту строки, остающемуся после каждого сопоставления. Теперь требуется получить список совпадений, удовлетворяющих некоторому критерию, который невозможно (достаточно просто) выразить в виде регулярного выражения. Например, при получении списка счастливых номеров требуется оставить только те, которые без остатка делятся на 13.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
StringCollection resultList = new StringCollection();
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
StringCollection resultList = new StringCollection();
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultList = New StringCollection
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
        ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim ResultList = New StringCollection
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
```

```
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
        ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    if (Integer.parseInt(regexMatcher.group()) % 13 == 0) {
        resultList.add(regexMatcher.group());
    }
}
```

JavaScript

```
var list = [];
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Не позволять таким броузерам, как Firefox,
    // попадать в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Здесь можно обработать совпадение, хранящееся в переменной match
    if (match[0] % 13 == 0) {
        list.push(match[0]);
    }
}
```

PHP

```
preg_match_all('/\d+/', $subject, $matchdata, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($matchdata[0]); $i++) {
    if ($matchdata[0][$i] % 13 == 0) {
        $list[] = $matchdata[0][$i];
    }
}
```

Perl

```
while ($subject =~ m/\d+/g) {
    if ($& % 13 == 0) {
        push(@list, $&);
    }
}
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
list = []
for matchobj in re.finditer(r"\d+", subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
list = []
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

Ruby

```
list = []
subject.scan(/\d+/) {|match|
    list << match if (Integer(match) % 13 == 0)
}
```

Обсуждение

Регулярное выражение имеет дело с текстом. Хотя регулярному выражению `‹\d+›` соответствует то, что мы называем числами, тем не менее, для механизма регулярных выражений это всего лишь строки, состоящие из одной или более цифр.

Если требуется отыскать какие-то определенные числа, например числа, кратные 13, намного проще написать универсальное регулярное выражение, отыскивающее совпадения со всеми числами, а затем добавить немного программного кода, который будет отбрасывать совпадения, не представляющие интереса.

Все решения для этого рецепта основаны на решениях из предыдущего рецепта, где демонстрируется, как выполнять обход всех совпадений. Совпадение с регулярным выражением внутри цикла преобразуется в число.

Некоторые языки программирования делают это автоматически, другие требуют явно вызвать функцию, которая преобразует строку в число. Далее проверяется, делится ли целое число на 13 без остатка. Если делится, совпадение добавляется в список. Если нет, совпадение пропускается.

См. также

Рецепты 3.7, 3.10 и 3.11.

3.13. Поиск совпадения внутри другого совпадения

Задача

Необходимо отыскать все совпадения с определенным регулярным выражением, но только внутри некоторого раздела испытуемой строки. Имеется еще одно регулярное выражение, которому соответствуют все разделы в строке.

Допустим, что имеется файл HTML, в котором различные сообщения выводятся жирным шрифтом, с помощью тегов . Необходимо отыскать все числа, заключенные в этот тег. Если текст, заключенный в тег содержит несколько чисел, все они должны рассматриваться как отдельные совпадения. Например, для строки 1 2 3 4 5 6 7 должно быть найдено четыре совпадения: 2, 5, 6 и 7.

Решение

C#

```
StringCollection resultList = new StringCollection();
Regex outerRegex = new Regex("<b>(.*?)</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex(@"\d+");
// Найти первый раздел
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
    // Получить совпадения внутри раздела
    Match innerMatch = innerRegex.Match(outerMatch.Groups[1].Value);
    while (innerMatch.Success) {
        resultList.Add(innerMatch.Value);
        innerMatch = innerMatch.NextMatch();
    }
    // Найти следующий раздел
    outerMatch = outerMatch.NextMatch();
}
```

VB.NET

```
Dim ResultList = New StringCollection
Dim OuterRegex As New Regex("<b>(.*?)</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("\d+")
'Найти первый раздел
```

```
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Получить совпадения внутри раздела
    Dim InnerMatch = InnerRegex.Match(OuterMatch.Groups(1).Value)
    While InnerMatch.Success
        ResultList.Add(InnerMatch.Value)
        InnerMatch = InnerMatch.NextMatch
    End While
    OuterMatch = OuterMatch.NextMatch
End While
```

Java

Проще всего реализовать итерации можно с использованием двух объектов совпадений, и этот прием будет работать в версии Java 4 и выше:

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    Matcher innerMatcher = innerRegex.matcher(outerMatcher.group());
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}
```

Следующий фрагмент обладает большей эффективностью (потому что объект `innerMatcher` создается всего один раз), но он будет работать только в версии Java 5 или выше:

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
Matcher innerMatcher = innerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    innerMatcher.region(outerMatcher.start(), outerMatcher.end());
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}
```

JavaScript

```
var result = [];
var outerRegex = /<b>([\s\S]*?)<\/b>/g;
var innerRegex = /\d+/g;
var outerMatch = null;
```

```
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex)
        outerRegex.lastIndex++;
    var innerSubject = subject.substr(outerMatch.index,
                                      outerMatch[0].length);
    var innerMatch = null;
    while (innerMatch = innerRegex.exec(innerSubject)) {
        if (innerMatch.index == innerRegex.lastIndex)
            innerRegex.lastIndex++;
        result.push(innerMatch[0]);
    }
}
```

PHP

```
$list = array();
preg_match_all('%<b>(.*?)</b>%s', $subject, $outermatches,
               PREG_PATTERN_ORDER);
for ($i = 0; $i < count($outermatches[0]); $i++) {
    if (preg_match_all('/\d+/', $outermatches[0][$i], $innermatches,
                       PREG_PATTERN_ORDER)) {
        $list = array_merge($list, $innermatches[0]);
    }
}
```

Perl

```
while ($subject =~ m!<b>(.*?)</b>!gs) {
    push(@list, ($& =~ m/\d+/g));
}
```

Этот фрагмент будет работать, только если внутреннее регулярное выражение (‹\d+› в данном примере) не содержит сохраняющих групп, поэтому следует использовать несохраняющую группировку. Подробнее об этом рассказывается в рецепте 2.9.

Python

```
list = []
innerre = re.compile(r"\d+")
for outermatch in re.finditer("(?s)<b>(.*?)</b>", subject):
    list.extend(innerre.findall(outermatch.group(1)))
```

Ruby

```
list = []
subject.scan(/<b>(.*?)<\/b>/m) {|outergroups|
    list += outergroups[0].scan(/\d+/)
}
```

Обсуждение

Регулярные выражения прекрасно подходят для выделения лексем из текста, но они плохо подходят для парсинга текста. Под выделением лексем здесь понимается идентификация различных частей строки, таких как числа, слова, символы, теги, комментарии и прочее. Регулярное выражение выполняет сканирование строки слева направо и пытается отыскать соответствие, опробуя различные альтернативы и различные количества символов. С такой работой регулярные выражения справляются просто прекрасно.

Под парсингом понимается процесс выявления взаимоотношений между лексемами. Например, в языках программирования комбинации лексем формируют инструкции, функции, классы, пространства имен и прочее. Отслеживание назначения лексем внутри широкого контекста лучше реализовать в программном коде. В частности, регулярные выражения не в состоянии отслеживать нелинейный контекст, например вложенные конструкции[1].

Поиск одной лексемы внутри другой лексемы часто пытаются реализовать с помощью регулярных выражений. Пара HTML-тегов `<b>` отыскивается с помощью простого регулярного выражения `‹<b>(.*?)</b>›`[2]. Числа отыскиваются с помощью еще более простого регулярного выражения `‹\d+›`. Но если попытаться объединить эти два выражения в одно, можно в результате получить нечто совсем иное:

```
\d+(?=(?:(?!<b>).)*</b>)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Хотя это регулярное выражение решает задачу, поставленную в этом рецепте, понять его будет очень непросто. Даже знаток регулярных выражений должен будет тщательнейшим образом исследовать его, чтобы понять, что оно делает; возможно даже воспользоваться каким-либо инструментом, чтобы протестировать его. А это всего лишь комбинация двух простых регулярных выражений.

---

[1] В некоторых современных диалектах регулярных выражений была предпринята попытка ввести средства сбалансированного, или рекурсивного сопоставления. Однако применение этих средств приводит к настолько сложным регулярным выражениям, что это лишний раз подтверждает мое замечание о том, что парсинг лучше всего выполнять с помощью программного кода.

[2] Чтобы иметь возможность идентифицировать тег, располагающийся на нескольких строках, следует включить режим «точке соответствуют границы строк». Для JavaScript можно воспользоваться выражением `‹<b>([\s\S]*?)</b>›`.

Лучшее решение состоит в том, чтобы сохранить два регулярных выражения в первоначальном виде, а для их объединения использовать программный код. Получившийся при этом программный код будет немного длиннее, но намного проще для понимания, а простота программного кода является одной из основных причин, по которым используют регулярные выражения. Выражение, подобное `‹<b>(.*?)</b>›`, легко поймет любой обладающий минимальным опытом работы с регулярными выражениями; к тому же такое выражение быстро сделает то, что в противном случае потребовало бы больше строк программного кода, более сложных в сопровождении.

Несмотря на то, что решения в этом рецепте являются одними из самых сложных в этой главе, тем не менее, они очень прямолинейны. Используются два регулярных выражения. «Внешнее» регулярное выражение используется для поиска HTML-тегов `<b>` и текста между ними, причем текст между тегами сохраняется первой сохраняющей группой. Работа с этим выражением реализована, как было показано в рецепте 3.11. Единственное отличие состоит в том, что комментарий, говорящий о том, где можно использовать совпадения, замещен программным кодом, позволяющим «внутреннему» регулярному выражению выполнить свою работу.

Второму регулярному выражению соответствуют цифры. Работа с этим выражением реализована, как было показано в рецепте 3.10. Единственное отличие состоит в том, что вместо целой строки второе регулярное выражение применяется только к части испытуемой строки, совпавшей с первой сохраняющей группой во внешнем регулярном выражении.

Существует два способа ограничить внутреннее регулярное выражение текстом, совпавшим (с первой сохраняющей группой) во внешнем выражении. В некоторых языках программирования имеется функция, позволяющая применять регулярное выражение к части строки. Это может помочь сэкономить на копировании строк, если функция поиска соответствий не производит автоматическое заполнение структуры текстом, соответствующим сохраняющим группам. Всегда имеется возможность извлечь подстроку, совпавшую с сохраняющей группой, и применить к ней внутреннее регулярное выражение.

В любом случае совместное использование двух регулярных выражений в цикле будет работать быстрее, чем использование одного регулярного выражения с вложенными опережающими проверками. Последнее требует, чтобы механизм регулярных выражений выполнял большое число возвратов. Для больших файлов единое регулярное выражение будет работать намного медленнее, так как ему потребуется определять границы раздела (HTML-теги `<b>`) для каждого числа, присутствующего в испытуемом тексте, включая числа, находящиеся вне тегов `<b>`. Решение, использующее два регулярных выражения, даже не пытается искать числа, пока не будут обнаружены границы раздела, время поиска которых зависит линейно от размера файла.

См. также

Рецепты 3.8, 3.10 и 3.11.

3.14. Замена всех совпадений

Задача

Необходимо заменить все совпадения с регулярным выражением ‹before› замещающим текстом «after».

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, “before”, “after”);
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
string resultString = null;
try {
    resultString = Regex.Replace(subjectString, “before”, “after”);
} catch (ArgumentNullException ex) {
    // Не допускается передавать значение null в качестве регулярного
    // выражения, испытуемой строки или замещающего текста
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex(“before”);
string resultString = regexObj.Replace(subjectString, “after”);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex с полной обработкой исключений:

```
string resultString = null;
try {
    Regex regexObj = new Regex(“before”);
    try {
        resultString = regexObj.Replace(subjectString, “after”);
    } catch (ArgumentNullException ex) {
```

```
        // Не допускается передавать значение null в качестве регулярного
        // выражения или замещающего текста
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString, "before", "after")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
    ResultString = Regex.Replace(SubjectString, "before", "after")
Catch ex As ArgumentNullException
    'Не допускается передавать значение null в качестве регулярного
    'выражения, испытуемой строки или замещающего текста
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("before")
Dim ResultString = RegexObj.Replace(SubjectString, "after")
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Regex` с полной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
    Dim RegexObj As New Regex("before")
    Try
        ResultString = RegexObj.Replace(SubjectString, "after")
    Catch ex As ArgumentNullException
        'Не допускается передавать значение null в качестве регулярного
        'выражения или замещающего текста
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
String resultString = subjectString.replaceAll("before", "after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
try {
    String resultString = subjectString.replaceAll("before", "after");
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
} catch (IllegalArgumentException ex) {
    // Синтаксическая ошибка в замещающем тексте (неэкранированный символ $
?)
} catch (IndexOutOfBoundsException ex) {
    // В замещающем тексте использована несуществующая обратная ссылка
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Matcher`:

```
Pattern regex = Pattern.compile("before");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Matcher` с полной обработкой исключений:

```
String resultString = null;
try {
    Pattern regex = Pattern.compile("before");
    Matcher regexMatcher = regex.matcher(subjectString);
    try {
        resultString = regexMatcher.replaceAll("after");
    } catch (IllegalArgumentException ex) {
        // Синтаксическая ошибка в замещающем тексте
        // (неэкранированный символ $ ?)
    } catch (IndexOutOfBoundsException ex) {
        // В замещающем тексте использована несуществующая обратная ссылка
    }
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
result = subject.replace(/before/g, "after");
```

PHP

```
$result = preg_replace('/before/', 'after', $subject);
```

Perl

Испытуемая строка хранится в специальной переменной $_, результат возвращается в переменной $_:

```
s/before/after/g;
```

Испытуемая строка хранится в переменной $subject, результат возвращается в переменной $subject:

```
$subject =~ s/before/after/g;
```

Испытуемая строка хранится в переменной $subject, результат возвращается в переменной $result:

```
($result = $subject) =~ s/before/after/g;
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub("before", "after", subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile("before")
result = reobj.sub("after", subject)
```

Ruby

```
result = subject.gsub(/before/, 'after')
```

Обсуждение

.NET

При реализации поиска с заменой с помощью регулярного выражения на платформе .NET всегда есть возможность использовать метод Regex.Replace(). Метод Replace() имеет 10 перегруженных версий. Половина из них принимают замещающий текст в виде строки – здесь рассматриваются эти методы. Другая половина принимает замещающий текст в виде делегата MatchEvaluator – эти методы рассматриваются в рецепте 3.16.

Первым аргументом, который ожидает получить метод Replace(), всегда является строка, хранящая оригинальный испытуемый текст, в ко-

тором требуется выполнить операцию поиска с заменой. Этот аргумент не может иметь значение `null`. В противном случае метод `Replace()` возбудит исключение `ArgumentNullException`. Метод `Replace()` всегда возвращает строку, в которой выполнены замены найденных совпадений.

Если регулярное выражение предполагается использовать всего несколько раз, можно применять статическую версию метода. Вторым аргументом этой версии метода передается используемое регулярное выражение. Замещающий текст передается в третьем аргументе. Параметры регулярного выражения можно передать в необязательном четвертом аргументе. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение `ArgumentException`.

Если одно и то же регулярное выражение предполагается использовать для обработки большого числа строк, можно повысить эффективность программного кода, создав сначала объект `Regex`, а затем вызывая метод `Replace()` этого объекта. Первым аргументом этому методу передается испытуемая строка, а вторым – замещающий текст. Это единственные обязательные аргументы.

При вызове метода `Replace()` относительно экземпляра класса `Regex` ему можно передать дополнительные аргументы, ограничивающие замещение совпадений. Если опустить эти аргументы, будут замещены все совпадения в испытуемой строке. Статическая версия метода `Replace()` не принимает этих дополнительных аргументов, она всегда замещает все совпадения.

В третьем необязательном аргументе, следующем за испытуемым и замещающим текстом, можно передать число выполняемых замен. Если число больше единицы, то оно обозначает максимально возможное число замен. Например, вызов `Replace(subject, replacement, 3)` заменит только первые три совпадения с регулярным выражением, а все остальные будут игнорироваться. Если в строке будет найдено менее трех совпадений, то замещены будут все совпадения. При этом нет никакой возможности определить ситуацию, когда число замен оказалось меньше, чем было запрошено. Если в третьем аргументе передать ноль, ни одной замены не будет произведено и будет возвращена строка в первоначальном виде. Если передать значение `-1`, будут замещены все совпадения с регулярным выражением. Если передать число меньше `-1`, метод `Replace()` возбудит исключение `ArgumentOutOfRangeException`.

Если методу был передан третий аргумент с числом замен, то можно передать четвертый необязательный аргумент, определяющий позицию символа, начиная с которого должно выполняться сопоставление с регулярным выражением.

По сути, число, которое передается в четвертом аргументе, представляет собой число символов в начале испытуемого текста, которые должны игнорироваться регулярным выражением. Этот аргумент может использоваться, когда строка уже обработана до некоторой позиции и необходимо выполнить поиск с заменой в оставшейся части строки. Зна-

чение четвертого аргумента должно быть числом в диапазоне от нуля до длины испытуемой строки. В противном случае метод Replace() возбудит исключение ArgumentOutOfRangeException. В отличие от метода Match(), Replace() не позволяет передавать аргумент, в котором можно было бы указать длину подстроки, где регулярному выражению позволено выполнять замены.

Java

Если требуется выполнить поиск с заменой в одной строке с помощью одного и того же регулярного выражения, можно воспользоваться методом replaceFirst() или replaceAll() данной строки. Оба метода принимают два аргумента: строку с регулярным выражением и строку с замещающим текстом. Эти функции удобно вызывать как: Pattern.compile("before").matcher(subjectString).replaceFirst("after") и Pattern.compile("before").matcher(subjectString).replaceAll("after").

Если одно и то же регулярное выражение предполагается использовать для обработки множества строк, следует создать объект Matcher, как описано в рецепте 3.3, а затем вызывать метод replaceFirst() или replaceAll() этого объекта, передавая ему замещающий текст в качестве единственного аргумента.

Если регулярное выражение и замещающий текст вводятся конечным пользователем, можно столкнуться с тремя классами исключений. Исключение PatternSyntaxException возбуждается методами Pattern.compile(), String.replaceFirst() и String.replaceAll(), если регулярное выражение содержит синтаксическую ошибку. Исключение ArgumentException возбуждается методами replaceFirst() и replaceAll(), если синтаксическая ошибка присутствует в замещающем тексте. Если замещающий текст синтаксически оформлен правильно, но в нем имеются обратные ссылки на несуществующие сохраняющие группы, возбуждается исключение IndexOutOfBoundsException.

JavaScript

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать метод replace() этой строки. Первым аргументом этому методу передается регулярное выражение, а вторым – строка с замещающим текстом. Метод replace() возвращает новую строку с произведенными заменами.

Если необходимо выполнить замену всех совпадений с регулярным выражением, при создании объекта регулярного выражения следует установить флаг /g. Как это делается, описывается в рецепте 3.4. В отсутствие флага /g замещено будет только первое совпадение.

PHP

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать функцию preg_replace(). Регулярное

выражение передается этой функции в первом аргументе, замещающий текст – во втором, а испытуемый текст – в третьем. Возвращаемое значение является строкой с испытуемым текстом, в котором выполнены замены.

С помощью четвертого необязательного аргумента можно ограничить число производимых замен. Если опустить этот аргумент или передать в нем значение -1, замещены будут все совпадения. Если указать значение 0, ни одной замены произведено не будет. Если указать положительное число, функция preg_replace() произведет число замен, не превышающее указанного значения. Если число совпадений окажется меньше указанного числа, все имеющиеся совпадения будут замещены без появления ошибки.

Если потребуется узнать количество произведенных замен, можно передать функции пятый аргумент. В этот аргумент будет записано целое число фактически произведенных замен.

Особенность функции preg_replace() заключается в том, что в первых трех аргументах вместо строк она может принимать массивы. Если в третьем аргументе вместо единственной строки передается массив, функция preg_replace() вернет массив строк, произведя поиск с заменой во всех строках.

Если в первом аргументе передать массив строк с регулярными выражениями, функция preg_replace() будет использовать эти регулярные выражения одно за другим, выполняя операции поиска с заменой в испытуемом тексте. Если функции передается массив испытуемых строк, все регулярные выражения будут применены ко всем испытуемым строкам. Когда выполняется поиск с помощью массива регулярных выражений, в качестве замещающего текста можно указать единственную строку (которая будет использоваться всеми регулярными выражениями) или массив строк. При использовании двух массивов функция preg_replace() обойдет оба массива регулярных выражений и строк замещающего текста и будет использовать различные строки замещающего текста для каждого регулярного выражения. Обход строк в массивах будет выполняться в том порядке, в каком они хранятся в памяти, что не всегда совпадает с порядком следования индексов в массивах. Если элементы добавлялись в массивы не в порядке следования индексов, массивы регулярных выражений и замещающего текста следует обработать функцией ksort(), прежде чем передавать их функции preg_replace().

В следующем примере массив $replace создается в обратном порядке:

```
$regex[0] = '/a/';
$regex[1] = '/b/';
$regex[2] = '/c/';
$replace[2] = '3';
$replace[1] = '2';
```

```
$replace[0] = '1';

echo preg_replace($regex, $replace, "abc");
ksort($replace);
echo preg_replace($regex, $replace, "abc");
```

Первый вызов функции preg_replace() вернет строку 321, которая может оказаться не тем, что вы ожидали. После обработки массива функцией ksort() операция замены вернет строку 123, как и требовалось. Функция ksort() изменяет саму переменную, которая ей передается – не следует передавать ее возвращаемое значение (true или false) функции preg_replace().

Perl

В языке Perl конструкция s/// в действительности является оператором подстановки. Если оператор s/// используется самостоятельно, поиск с заменой будет выполняться в переменной $_, а результат будет помещаться обратно в переменную $_.

Если необходимо использовать другую переменную с оператором подстановки, следует применять оператор связывания =~, чтобы ассоциировать оператор подстановки с требуемой переменой. Связывание оператора подстановки со строкой приводит к немедленному выполнению операции поиска с заменой. Результат сохраняется в переменной, где хранился испытуемый текст.

Оператор s/// всегда изменяет переменную, с которой он связывается. Если необходимо сохранить результат поиска с заменой в другой переменной так, чтобы исходная переменная не изменялась, то сначала нужно присвоить другой переменной содержимое исходной переменной, а затем связать оператор подстановки с этой переменной. Решение на языке Perl демонстрирует, как эти две операции можно выполнить в одной строке программного кода.

Чтобы выполнить замену всех совпадений с регулярным выражением, нужно использовать флаг /g, как описывается в рецепте 3.4. Без этого флага Perl заменит только первое совпадение.

Python

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать функцию sub() из модуля re. Регулярное выражение передается функции в первом аргументе, замещающий текст – во втором, а испытуемая строка – в третьем. Глобальная функция sub() не позволяет указывать параметры регулярного выражения.

Функция re.sub() вызывает функцию re.compile() и затем вызывает метод sub() объекта скомпилированного регулярного выражения. Этот метод принимает два обязательных аргумента: замещающий текст и испытуемую строку.

Обе формы функции `sub()` возвращают строку, в которой замещены все совпадения с регулярным выражением. Обе принимают необязательный аргумент, который можно использовать для ограничения числа производимых замен. Если опустить этот аргумент или передать в нем нулевое значение, замещены будут все совпадения. Если передать положительное число, оно будет ограничивать максимальное число замещаемых совпадений. Если число совпадений окажется меньше указанного числа, все имеющиеся совпадения будут замещены без появления ошибки.

Ruby

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать метод `gsub()` класса `String`. Регулярное выражение передается методу в первом аргументе, а замещающий текст – во втором. Возвращаемым значением метода является строка, с выполненными в ней заменами. Если в испытуемой строке совпадения с регулярным выражением отсутствуют, метод `gsub()` вернет оригинальную строку.

Метод `gsub()` не изменяет строку, относительно которой он вызывается. Если необходимо изменить оригинальную строку, следует вызывать метод `gsub!()`. Если в испытуемой строке совпадения с регулярным выражением отсутствуют, метод `gsub!()` вернет значение `nil`. В противном случае он вернет строку, относительно которой был произведен вызов, с выполненными в ней заменами.

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1 и рецепт 3.15.

3.15. Замена совпадений с повторным использованием частей совпадений

Задача

Необходимо выполнить поиск с заменой, при котором части совпадений с регулярным выражением будут вставляться обратно. Части, которые должны вставляться обратно, необходимо изолировать с помощью сохраняющих групп в регулярном выражении, как описывается в рецепте 2.9.

Например, необходимо отыскать пары слов, разделенных знаком равенства, и поменять эти слова местами.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, @"(\w+)=(\w+)", "$2=$1");
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex(@"(\w+)=(\w+)");
string resultString = regexObj.Replace(subjectString, "$2=$1");
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString, "(\w+)=(\w+)", "$2=$1")
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("(\w+)=(\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "$2=$1")
```

Java

При обработке единственной строки одним и тем же регулярным выражением можно использовать функцию `String.replaceAll()`:

```
String resultString = subjectString.replaceAll("(\\w+)=(\\w+)", "$2=$1");
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Matcher`:

```
Pattern regex = Pattern.compile("(\\w+)=(\\w+)");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("$2=$1");
```

JavaScript

```
result = subject.replace(/(\w+)=(\w+)/g, "$2=$1");
```

PHP

```
$result = preg_replace('/(\w+)=(\w+)/', '$2=$1', $subject);
```

Perl

```
$subject =~ s/(\w+)=(\w+)/$2=$1/g;
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub(r"(\w+)=(\w+)", r"\2=\1", subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile(r"(\w+)=(\w+)")
result = reobj.sub(r"\2=\1", subject)
```

Ruby

```
result = subject.gsub(/(\w+)=(\w+)/, '\2=\1')
```

Обсуждение

Регулярное выражение `‹(\w+)=(\w+)›` совпадает с парой слов и каждое из них сохраняется в своей собственной сохраняющей группе. Слово, стоящее перед знаком равенства, сохраняется в первой сохраняющей группе, второе слово, стоящее после знака равенства, сохраняется во второй сохраняющей группе.

В замещающем тексте необходимо указать сначала текст, совпавший со второй сохраняющей группой, затем знак равенства, а затем текст, совпавший с первой сохраняющей группой. Сделать это можно с помощью специальных символов-заполнителей. В различных языках программирования синтаксис определения замещающего текста может изменяться в широких пределах. Диалекты замещающего текста описываются в разделе «Поиск с заменой с помощью регулярных выражений» главы 1, а в рецепте 2.21 описывается, как ссылаться на сохраняющие группы в замещающем тексте.

.NET

В платформе .NET можно использовать тот же самый метод `Regex.Replace()`, что был описан в предыдущем рецепте, используя в качестве замещающего текста строку. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста .NET, как описывается в рецепте 2.21.

Java

В языке Java можно использовать те же самые методы `replaceFirst()` и `replaceAll()`, что были описаны в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Java, как описывается в этой книге.

JavaScript

В языке JavaScript можно использовать тот же самый метод `string.replace()`, что был описан в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста JavaScript, как описывается в этой книге.

PHP

В языке PHP можно использовать ту же самую функцию `preg_replace()`, что была описана в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста PHP, как описывается в этой книге.

Perl

В языке Perl часть `replace` в операторе `s/regex/replace/` интерпретируется как строка в кавычках. В замещающей строке допускается использовать специальные переменные `$&`, `$1`, `$2` и так далее, как описывается в рецепте 3.7 и в рецепте 3.9. Значения этих переменных устанавливаются сразу же, как только будет найдено совпадение с регулярным выражением, непосредственно перед выполнением замены. Эти переменные можно использовать в любом другом программном коде на языке Perl. Их значения остаются неизменными, пока не будет предпринята другая попытка сопоставления с регулярным выражением.

Все остальные языки программирования, рассматриваемые в этой книге, предоставляют функцию, которая принимает замещающий текст в виде строки. Функция производит синтаксический анализ этой строки, чтобы обработать обратные ссылки, такие как `$1` или `\1`. Но конструкция `$1` не имеет в этих языках специального назначения за пределами замещающего текста.

Python

В языке Python можно использовать ту же самую функцию `sub()`, что была описана в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Python, как описывается в этой книге.

Ruby

В языке Ruby можно использовать тот же самый метод `String.gsub()`, что был описан в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Ruby, как описывается в этой книге.

В языке Ruby не предоставляется возможность интерпретировать такие переменные, как `$1`, в замещающем тексте. Поэтому в Ruby интерпретация переменных выполняется перед вызовом `gsub()`. Но перед вы-

зовом gsub() еще не найдено ни одно совпадение, поэтому невозможно выполнить подстановку обратных ссылок. Если попробовать интерпретировать переменную $1, будет получен текст совпадения с первой сохраняющей группой в последнем регулярном выражении, использовавшемся перед вызовом gsub().

Поэтому в качестве обратной ссылки в замещающем тексте следует использовать такие метасимволы, как «\1». Функция gsub() подставляет текст совпадения с соответствующей сохраняющей группой на место этих метасимволов для каждого найденного совпадения с регулярным выражением. Я рекомендую заключать замещающий текст в апострофы. В строках, окруженных кавычками, символ обратного слэша интерпретируется как экранирующий символ и экранирует восьмеричные цифры. Во фрагментах '\1' и "\\1" используется текст совпадения с первой сохраняющей группой, тогда как фрагмент "\1" будет замещен символом с кодом 0x01.

Именованное сохранение

При использовании в регулярном выражении именованных сохраняющих групп имеется возможность ссылаться из замещающего текста на группы по их именам.

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString,
                      @"(?<left>\w+)=(?<right>\w+)", "${right}=${left}");
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex(@"(?<left>\w+)=(?<right>\w+)");
string resultString = regexObj.Replace(subjectString, "${right}=${left}");
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString,
                   "(?<left>\w+)=(?<right>\w+)", "${right}=${left}")
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Dim RegexObj As New Regex("(?<left>\w+)=(?<right>\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "${right}=${left}")
```

PHP

```
$result = preg_replace('/(?P<left>\w+)=(?P<right>\w+)/', '$2=$1', $subject);
```

Семейство функций preg в языке PHP использует библиотеку PCRE, которая поддерживает именованные сохранения. Функции preg_match() и preg_match_all() добавляют совпадения с именованными сохраняющими группами в массив с результатами. К сожалению, функция preg_replace() не предоставляет возможность использовать именованные обратные ссылки в замещающем тексте. Если регулярное выражение имеет именованные сохраняющие группы, нужно сосчитать именованные и нумерованные группы слева направо, чтобы определить номера обратных ссылок для каждой группы. Эти номера затем можно использовать в замещающем тексте.

Perl

```
$subject =~ s/(?<left>\w+)=(?<right>\w+)/$+{right}=$+{left}/g;
```

Именованные сохраняющие группы поддерживаются в Perl, начиная с версии 5.10. В хеше $+ хранятся совпадения со всеми именованными сохраняющими группами, имеющимися в последнем использовавшемся регулярном выражении. Этот хеш можно использовать в строке с замещающим текстом, как и в любом другом месте.

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub(r"(?P<left>\w+)=(?P<right>\w+)", r"\g<right>=\g<left>",
                subject)
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"(?P<left>\w+)=(?P<right>\w+)")
result = reobj.sub(r"\g<right>=\g<left>", subject)
```

Ruby

```
result = subject.gsub(/(?<left>\w+)=(?<right>\w+)/, '\k<left>=\k<right>')
```

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1, где описываются диалекты замещающего текста.

Рецепт 2.21, где объясняется, как добавлять обратные ссылки на сохраняющие группы в замещающий текст.

3.16. Замена совпадений фрагментами, сгенерированными в программном коде

Задача

Необходимо заменить все совпадения с регулярным выражением новым текстом, который генерируется программным кодом. При этом необходимо иметь возможность заменять все совпадения разными строками, основываясь на фактическом тексте совпадения.

Например, предположим, что необходимо заменить все числа в строке на их произведения с числом два.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, @"\d+",
                      new MatchEvaluator(ComputeReplacement));
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Replace(subjectString,
                      new MatchEvaluator(ComputeReplacement));
```

В обоих фрагментах вызывается функция `ComputeReplacement`. Этот метод необходимо добавить в класс, реализующий это решение:

```
public String ComputeReplacement(Match matchResult) {
    int twiceasmuch = int.Parse(matchResult.Value) * 2;
    return twiceasmuch.ToString();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = Regex.Replace(SubjectString, "\d+", MyMatchEvaluator)
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("\d+")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = RegexObj.Replace(SubjectString, MyMatchEvaluator)
```

В обоих фрагментах вызывается функция `ComputeReplacement`. Этот метод необходимо добавить в класс, реализующий это решение:

```
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    Dim TwiceAsMuch = Int.Parse(MatchResult.Value) * 2;
    Return TwiceAsMuch.ToString();
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    Integer twiceasmuch = Integer.parseInt(regexMatcher.group()) * 2;
    regexMatcher.appendReplacement(resultString, twiceasmuch.toString());
}
regexMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/\d+/g,
                             function(match) { return match * 2; }
                            );
```

PHP

Используя объявленную ранее функцию обратного вызова:

```
$result = preg_replace_callback('/\d+/', compute_replacement, $subject);

function compute_replacement($groups) {
    return $groups[0] * 2;
}
```

Используя анонимную функцию обратного вызова:

```
$result = preg_replace_callback(
    '/\d+/',
    create_function(
        '$groups',
        'return $groups[0] * 2;'
    ),
    $subject
);
```

Perl

```
$subject =~ s/\d+/$& * 2/eg;
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub(r"\d+", computereplacement, subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile(r"\d+")
result = reobj.sub(computereplacement, subject)
```

В обоих фрагментах вызывается функция `computereplacement`. Эта функция должна быть объявлена до того, как она будет передана функции `sub()`.

```
def computereplacement(matchobj):
    return str(int(matchobj.group()) * 2)
```

Ruby

```
result = subject.gsub(/\d+/) {|match|
    Integer(match) * 2
}
```

Обсуждение

Когда в качестве замещающего текста используется строка, имеется возможность выполнять только простую подстановку текста. Чтобы каждое совпадение можно было заменить чем-то уникальным, основываясь на замещаемом совпадении, замещающий текст необходимо генерировать программным кодом.

C#

В рецепте 3.14 рассматривались различные способы вызова метода `Regex.Replace()`, когда ему в качестве замещающего текста передается строка. Статическая версия метода принимает замещающий текст в третьем аргументе, после испытуемого текста и регулярного выражения. Если передать регулярное выражение конструктору `Regex()`, появится возможность вызывать метод `Replace()` созданного объекта, который принимает замещающий текст во втором аргументе.

Вместо строки с замещающим текстом во втором или третьем аргументе допускается передавать делегат `MatchEvaluator`. Этот делегат является ссылкой на функцию-член, которая добавляется в класс, выполняющий операцию поиска с заменой. Создание делегата производится с помощью ключевого слова `new` и вызова конструктора `MatchEvaluator()`. Функция-член передается конструктору `MatchEvaluator()` в виде единственного аргумента.

Функция, которая будет использоваться для создания делегата, должна возвращать строку и принимать объект класса System.Text.RegularExpressions.Match в виде единственного аргумента. Это тот самый объект класса Match, который возвращается функцией-членом Regex.Match(), используемой почти во всех предыдущих рецептах в этой главе.

Когда метод Replace() вызывается с делегатом MatchEvaluator в качестве замещающего текста, для каждого совпадения с регулярным выражением, которое требуется заменить, будет вызываться ваша функция. Эта функция должна возвращать замещающий текст. В процессе создания замещающего текста допускается использовать любые свойства объекта Match. В примере, показанном выше, для извлечения строки совпадения со всем выражением используется свойство matchResult.Value. На практике для сборки замещающего текста из совпадений с сохраняющими группами, имеющимися в регулярном выражении, часто используется коллекция matchResult.Groups[].

Если какие-то определенные совпадения не должны замещаться, функция должна возвращать значение свойства matchResult.Value. Если функция вернет значение null или пустую строку, совпадение с регулярным выражением будет замещено пустой строкой (то есть будет удалено).

VB.NET

В рецепте 3.14 рассматривались различные способы вызова метода Regex.Replace(), когда ему в качестве замещающего текста передается строка. Статическая версия метода принимает замещающий текст в третьем аргументе, после испытуемого текста и регулярного выражения. Если использовать ключевое слово Dim для создания переменной с регулярным выражением, появится возможность вызывать метод Replace() этого объекта, который принимает замещающий текст во втором аргументе.

Вместо строки с замещающим текстом во втором или третьем аргументе допускается передавать объект класса MatchEvaluator. Этот объект содержит ссылку на функцию, которая добавляется в класс, выполняющий операцию поиска с заменой. Создание объекта класса MatchEvaluator производится с помощью ключевого слова Dim. Конструктору MatchEvaluator() передается ключевое слово AddressOf, за которым следует имя функции-члена. Оператор AddressOf возвращает ссылку на функцию, не производя фактический вызов этой функции.

Функция, которая будет использоваться в объекте MatchEvaluator, должна возвращать строку и принимать объект класса System.Text.RegularExpressions.Match в виде единственного аргумента. Это тот самый объект класса Match, который возвращается функцией-членом Regex.Match(), используемой почти во всех предыдущих рецептах в этой главе. Аргумент передается функции по значению, поэтому он объявлен с ключевым словом ByVal.

Когда метод Replace() вызывается с объектом MatchEvaluator в качестве замещающего текста, для каждого совпадения с регулярным выражением, которое требуется заменить, будет вызываться ваша функция. Эта функция должна возвращать замещающий текст. В процессе создания замещающего текста допускается использовать любые свойства объекта Match. В примере, показанном выше, для извлечения строки совпадения со всем выражением используется свойство MatchResult.Value. На практике для сборки замещающего текста из совпадений с сохраняющими группами, имеющимися в регулярном выражении, часто используется коллекция MatchResult.Groups[].

Если какие-то определенные совпадения не должны замещаться, функция должна возвращать значение свойства MatchResult.Value. Если функция вернет значение Nothing или пустую строку, совпадение с регулярным выражением будет замещено пустой строкой (т. е. будет удалено).

Java

Решение для Java выглядит очень прямолинейным. Здесь выполняются итерации через все совпадения с регулярным выражением, как описано в рецепте 3.11. Внутри цикла вызывается метод appendReplacement() объекта Matcher. Когда метод find() не может отыскать последующие совпадения, вызывается метод appendTail(). Применение методов appendReplacement() и appendTail() существенно упрощает возможность замены каждого совпадения уникальным замещающим текстом.

Метод appendReplacement() принимает два аргумента. Первый аргумент имеет тип StringBuffer и служит временным хранилищем результатов в процессе выполнения операции поиска с заменой. Второй аргумент – это замещающий текст для последнего совпадения, которое будет обнаружено методом find(). Этот замещающий текст может включать ссылки на сохраняющие группы, такие как "$1". Если в замещающем тексте имеется синтаксическая ошибка, будет возбуждено исключение IllegalArgumentException. Если в замещающем тексте имеется ссылка на несуществующую сохраняющую группу, будет возбуждено исключение IndexOutOfBoundsException. Если вызвать метод appendReplacement() после неудачной попытки отыскать совпадение функцией find(), будет возбуждено исключение IllegalStateException.

В случае успешного завершения вызова метода appendReplacement() выполняются два действия. Во-первых, он копирует в буфер текст, находящийся между предыдущим и текущим совпадениями, не выполняя никаких преобразований. Если текущее совпадение было первым, копируется весь текст, находящийся перед ним. Во-вторых, он добавляет замещающий текст, подставляя на место обратных ссылок совпадения с соответствующими сохраняющими группами.

Если требуется удалить какое-то совпадение, его достаточно просто заменить пустой строкой. Если необходимо оставить текст со-

впадения без изменения, нужно просто пропустить вызов метода appendReplacement() для этого совпадения. Когда я говорю «предыдущее совпадение», я подразумеваю предыдущее совпадение, для которого вызывался метод appendReplacement(). Если для каких-то совпадений метод appendReplacement() не вызывался, они становятся частью текста между замещаемыми совпадениями, который копируется в буфер без изменения.

Завершив замещение совпадений, следует вызвать метод appendTail(). Он скопирует текст в конце строки, находящийся после последнего совпадения, для которого вызывался метод appendReplacement().

JavaScript

В языке JavaScript функции являются обычными объектами, которые можно присваивать переменным. Вместо строкового литерала или переменной, которая хранит строку, в функцию string.replace() можно передать свою функцию, которая возвращает строку. Эта функция будет вызываться всякий раз, когда будет обнаруживаться необходимость выполнить замену.

Функцию, возвращающую замещающий текст, можно объявить так, что она будет принимать один или более аргументов. В этом случае в первом аргументе ей будет передаваться текст совпадения с регулярным выражением. Если в регулярном выражении имеются сохраняющие группы, во втором аргументе будет передан текст совпадения с первой сохраняющей группой, в третьем аргументе будет передан текст совпадения со второй сохраняющей группой и так далее. Благодаря этим аргументам можно использовать фрагменты совпадения с регулярным выражением для создания замещающего текста.

Функция, возвращающая замещающий текст, в решении на языке JavaScript для этого рецепта, просто принимает текст совпадения с регулярным выражением и возвращает число, умноженное на два. Преобразования из строки в число и обратно в языке JavaScript выполняются неявно.

PHP

Функция preg_replace_callback() действует точно так же, как и функция preg_replace(), которая была описана в рецепте 3.14. Она принимает регулярное выражение, замещение, испытуемый текст, необязательную начальную позицию и необязательное максимальное число замен. Регулярное выражение и испытуемый текст могут быть представлены обычными строками или массивами строк.

Отличие функции preg_replace_callback() состоит в том, что в качестве замещения она принимает не строку или массив строк, а функцию. Функция, возвращающая замещающий текст, может быть объ-

явлена как именованная функция или как анонимная функция с помощью функции `create_function()`. Функция должна принимать один аргумент и возвращать строку (или что-то, что может быть преобразовано в строку).

Всякий раз, когда функция `preg_replace_callback()` обнаруживает совпадение с регулярным выражением, она обращается к функции обратного вызова. Аргумент заполняется массивом строк. Нулевой элемент этого массива содержит совпадение со всем регулярным выражением, остальные элементы хранят совпадения с соответствующими сохраняющими группами. Этот массив можно применять для сборки замещающего, используя текст совпадения с регулярным выражением или с одной или более сохраняющими группами.

Perl

Оператор `s///` поддерживает один дополнительный модификатор, который игнорируется оператором `m//`: `/e`. Модификатор `/e`, или «execute» (выполнить), предписывает оператору подстановки выполнить часть оператора, содержащую замещающий текст, как программный код на языке Perl, а не интерпретировать ее, как строку в кавычках. С помощью этого модификатора легко можно извлечь текст из переменной `$&` и умножить его на два. Результат работы программного кода используется как замещающий текст.

Python

Функция `sub()` в языке Python позволяет передавать функцию вместо строки с замещающим текстом. Эта функция будет вызываться для замещения каждого совпадения с регулярным выражением.

Эту функцию следует объявить прежде, чем на нее можно будет сослаться. Она должна принимать один аргумент, экземпляр класса `MatchObject`, который является тем же самым объектом, что возвращает функция `search()`. Этот объект может использоваться для извлечения совпадений (или их частей) с регулярным выражением и конструирования замещающего текста на их основе. Подробнее об этом рассказывается в рецепте 3.7 и в рецепте 3.9.

Функция обязана возвращать строку с замещающим текстом.

Ruby

В двух предыдущих рецептах использовался метод `gsub()` класса `String` с двумя аргументами: регулярное выражение и замещающий текст. Для этого метода существует блочная форма.

В блочной форме метод `gsub()` принимает единственный аргумент с регулярным выражением. Он присваивает первой переменной цикла строку с текстом совпадения для всего регулярного выражения. Если доба-

вить дополнительные переменные цикла, они будут получать значение nil, даже если регулярное выражение содержит сохраняющие группы.

Внутрь блока следует поместить выражение, создающее строку, которая будет использоваться в качестве замещающего текста. Внутри блока допускается использовать специальные переменные, которые заполняются в процессе сопоставления с регулярным выражением, такие как $~, $& и $1. Подробнее об этом рассказывается в рецептах 3.7, 3.8 и 3.9.

В замещающем тексте не допускается использовать такие метасимволы, как «\1». Они будут интерпретироваться как обычный текст.

См. также

Рецепты 3.9 и 3.15.

3.17. Замещение всех совпадений внутри совпадений с другим регулярным выражением

Задача

Необходимо заменить все совпадения с определенным регулярным выражением, но только внутри определенных участков испытуемого текста. Искомыми участками являются совпадения с другим регулярным выражением.

Допустим, что имеется файл HTML, в котором различные фрагменты выделены тегами . Внутри каждой пары тегов необходимо заменить все совпадения с регулярным выражением <before> замещающим текстом <after>. Например, строка before first before before before before должна быть преобразована в строку: before first after before after after.

Решение

C#

```
Regex outerRegex = new Regex("<b>.*?</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex("before");
string resultString = outerRegex.Replace(subjectString,
                      new MatchEvaluator(ComputeReplacement));

public String ComputeReplacement(Match matchResult) {
    // Выполнить поиск с заменой
    // для каждого совпадения с "внешним" регулярным выражением
    return innerRegex.Replace(matchResult.Value, "after");
}
```

VB.NET

```
Dim OuterRegex As New Regex("<b>.*?</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("before")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = OuterRegex.Replace(SubjectString, MyMatchEvaluator)

Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    'Выполнить поиск с заменой
    'для каждого совпадения с "внешним" регулярным выражением
    Return InnerRegex.Replace(MatchResult.Value, "after");
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<b>.*?</b>");
Pattern innerRegex = Pattern.compile("before");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    outerMatcher.appendReplacement(resultString,
        innerRegex.matcher(outerMatcher.group()).replaceAll("after"));
}
outerMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/<b>.*?<\/b>/g,
                             function(match) {
                                 return match.replace(/before/g, "after");
                             }
                            );
```

PHP

```
$result = preg_replace_callback('%<b>.*?</b>%',
                                replace_within_tag, $subject);

function replace_within_tag($groups) {
    return preg_replace('/before/', 'after', $groups[0]);
}
```

Perl

```
$subject =~ s%<b>.*?</b>%($match = $&) =~ s/before/after/g; $match;%eg;
```

Python

```
innerre = re.compile("before")
def replacewithin(matchobj):
    return innerre.sub("after", matchobj.group())
```

```
result = re.sub("<b>.*?</b>", replacewithin, subject)
```

Ruby

```
innerre = /before/
result = subject.gsub(/<b>.*?<\/b>/) {|match|
    match.gsub(innerre, 'after')
}
```

Обсуждение

Данное решение представляет собой комбинацию решений двух предыдущих рецептов и основано на использовании двух регулярных выражений. «Внешнему» регулярному выражению `‹<b>.*?</b>›` соответствуют пары тегов `<b>` и текст между ними. «Внутреннему» регулярному выражению соответствует слово «before», которое замещается словом «after».

В рецепте 3.16 описывается, как выполнить поиск с заменой, конструируя замещающий текст для каждого совпадения с помощью программного кода. Здесь этот прием применяется для замещения совпадений с внешним регулярным выражением. Каждый раз, когда внешнее регулярное выражение обнаруживает открывающий и закрывающий теги `<b>`, запускается операция поиска с заменой, как это делалось уже в рецепте 3.14. Испытуемым текстом для операции поиска с заменой во внутреннем регулярном выражении является текст, совпавший с внешним регулярным выражением.

См. также

Рецепты 3.11, 3.13 и 3.16.

3.18. Замещение всех совпадений между совпадениями с другим регулярным выражением

Задача

Необходимо заменить все совпадения с некоторым регулярным выражением, но только внутри определенных участков испытуемого текста. Искомыми участками являются фрагменты, расположенные между совпадениями с другим регулярным выражением. Другими словами, необходимо выполнить поиск с заменой во всех фрагментах испытуемого текста, которые не совпали с другим регулярным выражением.

Предположим, что имеется файл HTML, в котором необходимо заменить все прямые кавычки (`"`) фигурными (“ ”), но только за пределами тегов HTML. Кавычки внутри тегов должны оставаться прямыми кавычками ASCII, в противном случае веб-броузер окажется не в состоянии выполнить парсинг разметки HTML. Например, строка `"text"`

‹span class="middle"›"text"‹/span› "text" **должна быть преобразована в** "text" ‹span class="middle"›"text"‹/span› "text".

Решение

C#

```
string resultString = null;
Regex outerRegex = new Regex("<[^<>]*>");
Regex innerRegex = new Regex("\"([^\"]*)\"");
// Найти первый фрагмент
int lastIndex = 0;
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
    // Выполнить поиск с заменой в тексте между этим
    // и предыдущим совпадениями
    string textBetween =
        subjectString.Substring(lastIndex, outerMatch.Index - lastIndex);
    resultString = resultString +
        innerRegex.Replace(textBetween, "\u201C$1\u201D");
    lastIndex = outerMatch.Index + outerMatch.Length;
    // Скопировать текст фрагмента без изменений
    resultString = resultString + outerMatch.Value;
    // Найти следующий фрагмент
    outerMatch = outerMatch.NextMatch();
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
string textAfter = subjectString.Substring(lastIndex,
                   subjectString.Length - lastIndex);
resultString = resultString + innerRegex.Replace(textAfter,
                                                 "\u201C$1\u201D");
```

VB.NET

```
Dim ResultString As String = Nothing
Dim OuterRegex As New Regex("<[^<>]*>")
Dim InnerRegex As New Regex("""([^""]*)""")
'Найти первый фрагмент
Dim LastIndex = 0
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Выполнить поиск с заменой в тексте между этим
    'и предыдущим совпадениями
    Dim TextBetween = SubjectString.Substring(LastIndex,
                      OuterMatch.Index - LastIndex);
    ResultString = ResultString + InnerRegex.Replace(TextBetween,
                   ChrW(&H201C) + "$1" + ChrW(&H201D))
```

```
    LastIndex = OuterMatch.Index + OuterMatch.Length
    ‘Скопировать текст фрагмента без изменений
    ResultString = ResultString + OuterMatch.Value
    ‘ Найти следующий фрагмент
    OuterMatch = OuterMatch.NextMatch
End While
‘Выполнить поиск с заменой в оставшейся части строки
‘после последнего совпадения
Dim TextAfter = SubjectString.Substring(LastIndex,
                                        SubjectString.Length - LastIndex);
ResultString = ResultString +
    InnerRegex.Replace(TextAfter, ChrW(&H201C) + “$1” + ChrW(&H201D))
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile(“<[^<>]*>”);
Pattern innerRegex = Pattern.compile(“\”([^\”]*)\””);
Matcher outerMatcher = outerRegex.matcher(subjectString);
int lastIndex = 0;
while (outerMatcher.find()) {
    // Выполнить поиск с заменой в тексте между этим
    // и предыдущим совпадениями
    String textBetween = subjectString.substring(lastIndex,
                                                 outerMatcher.start());
    Matcher innerMatcher = innerRegex.matcher(textBetween);
    resultString.append(innerMatcher.replaceAll(“\u201C$1\u201D”));
    lastIndex = outerMatcher.end();
    // Добавить совпадение с регулярным выражением, не изменяя его
    resultString.append(outerMatcher.group());
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
String textAfter = subjectString.substring(lastIndex);
Matcher innerMatcher = innerRegex.matcher(textAfter);
resultString.append(innerMatcher.replaceAll(“\u201C$1\u201D”));
```

JavaScript

```
var result = “”;
var outerRegex = /<[^<>]*>/g;
var innerRegex = /”([^”]*)”/g;
var outerMatch = null;
var lastIndex = 0;
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex) outerRegex.lastIndex++;
    // Выполнить поиск с заменой в тексте между этим
    // и предыдущим совпадениями
```

```
        var textBetween = subject.substring(lastIndex, outerMatch.index);
        result = result + textBetween.replace(innerRegex, "\u201C$1\u201D");
        lastIndex = outerMatch.index + outerMatch[0].length;
        // Добавить совпадение с регулярным выражением, не изменяя его
        result = result + outerMatch[0];
    }
    // Выполнить поиск с заменой в оставшейся части строки
    // после последнего совпадения
    var textAfter = subject.substr(lastIndex);
    result = result + textAfter.replace(innerRegex, "\u201C$1\u201D");
```

PHP

```
$result = '';
$lastindex = 0;
while (preg_match('/<[^<>]*>/', $subject, $groups, PREG_OFFSET_CAPTURE,
                  $lastindex)) {
    $matchstart = $groups[0][1];
    $matchlength = strlen($groups[0][0]);
    // Выполнить поиск с заменой в тексте между этим
    // и предыдущим совпадениями
    $textbetween = substr($subject, $lastindex, $matchstart-$lastindex);
    $result .= preg_replace('/"([^"]*)"/', '"$1"', $textbetween);
    // Добавить совпадение с регулярным выражением, не изменяя его
    $result .= $groups[0][0];
    // Переместить начальную позицию для поиска следующего совпадения
    $lastindex = $matchstart + $matchlength;
    if ($matchlength == 0) {
        // Устранить вероятность бесконечного цикла в случае,
        // когда регулярное выражение допускает возможность появления
        // совпадений нулевой длины
        $lastindex++;
    }
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
$textafter = substr($subject, $lastindex);
$result .= preg_replace('/"([^"]*)"/', '"$1"', $textafter);
```

Perl

```
use encoding "utf-8";
$result = '';
while ($subject =~ m/<[^<>]*>/g) {
    $match = $&;
    $textafter = $';
    ($textbetween = $`) =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
    $result .= $textbetween . $match;
```

```
}
$textafter =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
$result .= $textafter;
```

Python

```
innerre = re.compile('"([^"]*)"')
result = "";
lastindex = 0;
for outermatch in re.finditer("<[^<>]*>", subject):
    # Выполнить поиск с заменой в тексте между этим
    # и предыдущим совпадениями
    textbetween = subject[lastindex:outermatch.start()]
    result += innerre.sub(u"\u201C\\1\u201D", textbetween)
    lastindex = outermatch.end()
    # Добавить совпадение с регулярным выражением, не изменяя его
    result += outermatch.group()
# Выполнить поиск с заменой в оставшейся части строки
# после последнего совпадения
textafter = subject[lastindex:]
result += innerre.sub(u"\u201C\\1\u201D", textafter)
```

Ruby

```
result = '';
textafter = ''
subject.scan(/<[^<>]*>/) {|match|
    textafter = $'
    textbetween = $`.gsub(/"([^"]*)"/, '"\1"')
    result += textbetween + match
}
result += textafter.gsub(/"([^"]*)"/, '"\1"')
```

Обсуждение

В рецепте 3.13 говорилось, как задействовать два регулярных выражения, чтобы отыскать совпадения (со вторым регулярным выражением) внутри определенных фрагментов в файле (совпавших с первым регулярным выражением). В решениях для этого рецепта используется тот же самый прием поиска и замены совпадений только в пределах определенных фрагментов испытуемого текста.

Очень важно, чтобы регулярное выражение, которое используется для поиска фрагментов, разделяющих испытуемый текст на части, работало с оригинальной строкой. Если бы оригинальная строка модифицировалась, необходимо было бы смещать начальную позицию поиска для первого регулярного выражения, отыскивающего разделы, по мере добавления и удаления символов вторым регулярным выражением. Еще более важно, что модификации могут вносить нежелательные побоч-

ные эффекты. Например, если внешнее регулярное выражение использует якорный метасимвол ‹^›, чтобы привязать совпадение к началу строки, а внутреннее регулярное выражение вставляет символ перевода строки в конец раздела, найденного первым регулярным выражением, то метасимвол ‹^› будет совпадать сразу же за предыдущим разделом с вновь добавленным символом перевода строки.

Несмотря на то, что решения получились довольно длинными, все они достаточно прямолинейны. В решениях используются два регулярных выражения. «Внешнему» регулярному выражению ‹<[^<>]*>› соответствуют пары угловых скобок и все, что находится между ними, за исключением самих угловых скобок. Это грубый способ поиска любых тегов HTML. Это регулярное выражение замечательно будет справляться со своей задачей при условии, что файл HTML не содержит каких-либо угловых скобок, которые не были (по ошибке) оформлены как сущности. Реализация работы с этим регулярным выражением в точности повторяет программный код из рецепта 3.11. Единственное отличие состоит в том, что комментарии, сообщающие, где можно использовать найденное совпадение, замещены программным кодом, который выполняет операцию поиска с заменой.

Поиск с заменой в цикле реализован точно так же, как решение для рецепта 3.14. Испытуемая строка, в которой выполняется поиск с заменой, содержит фрагмент текста между предыдущим и текущим совпадениями с внешним регулярным выражением. Результат работы внутренней операции поиска с заменой добавляется в конец строки с общим результатом. Кроме того, в конец строки с общим результатом в неизменном виде добавляется текущее совпадение с внешним регулярным выражением.

Когда попытка отыскать совпадение с внешним регулярным выражением терпит неудачу, в последний раз запускается операция поиска с заменой в тексте, следующем за последним совпадением с внешним регулярным выражением.

Регулярному выражению ‹“([^”]*)”›, используемому внутри цикла для выполнения поиска с заменой, соответствуют пары кавычек и все, что находится между ними, за исключением самих кавычек. Текст между кавычками сохраняется первой сохраняющей группой.

В замещающем тексте используется ссылка на первую сохраняющую группу, которая помещена внутри пары фигурных кавычек. Фигурным кавычкам соответствуют пункты Юникода U+201C и U+201D. Обычно фигурные кавычки можно вставлять непосредственно в исходные тексты. Однако Visual Studio 2008, претендуя на интеллектуальность, автоматически замещает фигурные кавычки прямыми.

Внутри регулярного выражения сопоставление с кодовыми пунктами Юникода можно описать как ‹\u201C› или ‹\x{201C}›, но ни в одном

из языков программирования, рассматриваемых в этой книге, не поддерживается возможность использовать такие конструкции в замещающем тексте. Если конечный пользователь пожелает вставить фигурные кавычки в замещающий текст, который он набирает в поле ввода, ему придется вставлять эти символы, копируя их из таблицы символов. В исходном тексте программы, внутри замещающего текста, можно использовать экранированные последовательности Юникода, если они поддерживаются языком программирования в строковых литералах. Например, языки C# и Java поддерживают конструкцию `\u201C` на уровне строки, но VB.NET не обеспечивает возможность добавления экранированных последовательностей Юникода в строки. В VB.NET для преобразования кодового пункта Юникода в символ можно воспользоваться функцией `ChrW()`.

Perl и Ruby

В решениях на языках Perl и Ruby используются две специальные переменные, доступные в этих языках, о которых еще не рассказывалось. Переменная `$` ` (знак доллара и обратный апостроф) хранит часть текста, расположенную слева от совпадения, а переменная `$'` (знак доллара и апостроф) хранит часть текста, расположенную справа от совпадения. Вместо того чтобы выполнять итерации по совпадениям в оригинальной строке с испытуемым текстом, мы начинаем новый поиск в части строки, следующей за предыдущим совпадением. При таком подходе с помощью переменной `$` ` легко можно извлечь текст, расположенный между предыдущим и текущим совпадениями.

Python

Результатом работы этого фрагмента кода является строка символов Юникода, потому что замещающий текст объявляется как строка Юникода. Чтобы отобразить ее, может потребоваться задействовать функцию `encode()`, как показано ниже:

```
print result.encode('1252')
```

См. также

Рецепты 3.11, 3.13 и 3.16.

3.19. Разбиение строки

Задача

Необходимо разбить строку с помощью регулярного выражения. После разбиения должен получиться массив строк, содержащих фрагменты текста между совпадениями с регулярным выражением.

Например, требуется разбить строку, содержащую теги HTML, по этим тегам. Результатом разбиения строки I•like•bold•and•italic•fonts должен быть массив из пяти строк: I•like•, bold, •and•, italic и •fonts.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string[] splitArray = Regex.Split(subjectString, "<[^<>]*>");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
string[] splitArray = null;
try {
    splitArray = Regex.Split(subjectString, "<[^<>]*>");
} catch (ArgumentNullException ex) {
    // Не допускается передавать значение null в качестве регулярного
    // выражения или испытуемой строки
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex("<[^<>]*>");
string[] splitArray = regexObj.Split(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex с полной обработкой исключений:

```
string[] splitArray = null;
try {
    Regex regexObj = new Regex("<[^<>]*>");
    try {
        splitArray = regexObj.Split(subjectString);
    } catch (ArgumentNullException ex) {
        // Не допускается передавать значение null в качестве регулярного
        // выражения или испытуемой строки
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
Dim SplitArray As String()
Try
    SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
Catch ex As ArgumentNullException
    'Не допускается передавать значение null в качестве регулярного
    'выражения или испытуемой строки
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("<[^<>]*>")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Regex` с полной обработкой исключений:

```
Dim SplitArray As String()
Try
    Dim RegexObj As New Regex("<[^<>]*>")
    Try
        SplitArray = RegexObj.Split(SubjectString)
    Catch ex As ArgumentNullException
        'Не допускается передавать значение null в качестве регулярного
        'выражения или испытуемой строки
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Если необходимо разбить всего одну строку с одним и тем же регулярным выражением, можно использовать функцию `String.Split()`:

```
String[] splitArray = subjectString.split("<[^<>]*>");
```

Если регулярное выражение вводится конечным пользователем, следует предусмотреть полную обработку исключений:

```
try {
    String[] splitArray = subjectString.split("<[^<>]*>");
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Pattern`:

```
Pattern regex = Pattern.compile("<[^<>]*>");
String[] splitArray = regex.split(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Pattern` с полной обработкой исключений:

```
String[] splitArray = null;
try {
    Pattern regex = Pattern.compile("<[^<>]*>");
    splitArray = regex.split(subjectString);
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

Разбивать строки с применением регулярного выражения можно с помощью функции `string.split()`:

```
result = subject.split(/<[^<>]*>/);
```

К сожалению, существует масса проблем, связанных с использованием регулярного выражения в функции `string.split()` в разных броузерах. Наиболее надежный способ заключается в создании списка вручную:

```
var list = [];
var regex = /<[^<>]*>/g;
var match = null;
var lastIndex = 0;
while (match = regex.exec(subject)) {
    // Исключить вероятность попадания таких броузеров, как Firefox,
    // в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Добавить в список текст перед совпадением
    list.push(subject.substring(lastIndex, match.index));
    lastIndex = match.index + match[0].length;
}
// Добавить в список остаток строки после последнего совпадения
list.push(subject.substr(lastIndex));
```

PHP

```
$result = preg_split('/<[^<>]*>/', $subject);
```

Perl

```
@result = split(m/<[^<>]*>/, $subject);
```

Python

При разбиении небольшого числа строк можно использовать глобальную функцию:

```
result = re.split("<[^<>]*>", subject))
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile("<[^<>]*>")
result = reobj.split(subject)
```

Ruby

```
result = subject.split(/<[^<>]*>/)
```

Обсуждение

При разбиении строки с помощью регулярного выражения по сути решается задача, противоположная рецепту 3.10. Вместо получения списка всех совпадений с регулярным выражением в данном случае требуется получить список фрагментов текста между совпадениями, включая текст перед первым совпадением и после последнего совпадения. Сами совпадения с регулярным выражением исключаются из конечного результата.

C# и VB.NET

Чтобы разбить строку с помощью регулярного выражения, в платформе .NET всегда имеется возможность использовать метод `Regex.Split()`. Первым аргументом методу всегда передается строка, хранящая оригинальный испытуемый текст, который требуется разбить. В этом аргументе не допускается передавать значение `null`. В противном случае метод `Split()` возбудит исключение `ArgumentNullException`. Возвращаемым значением метода `Split()` всегда является массив строк.

Если регулярное выражение предполагается использовать всего несколько раз, можно использовать статическую версию метода. Вторым аргументом этой версии метода передается используемое регулярное выражение. Параметры регулярного выражения можно передавать в третьем необязательном аргументе. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение `ArgumentException`.

Если одно и то же регулярное выражение предполагается использовать для обработки большого числа строк, можно повысить эффективность программного кода, создав сначала объект Regex, а затем вызывая метод Split() этого объекта. Единственным обязательным аргументом этого метода является испытуемая строка.

При вызове метода Split() относительно экземпляра класса Regex ему можно передать дополнительные аргументы, ограничивающие количество разбиений. Если опустить эти аргументы, строка будет разбита по всем имеющимся совпадениям. Статическая версия метода Split() не принимает эти дополнительные аргументы, она всегда разбивает строку по всем имеющимся совпадениям.

Во втором необязательном аргументе, следующем за испытуемым текстом, можно передать максимальное число разбиений. Например, вызов regexObj.Split(subject, 3) вернет массив, содержащий не более трех строк. Метод Split() попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива.

Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод Split() выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов regexObj.Split(subject, 1) вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка. Вызов regexObj.Split(subject, 0) выполнит разбиение по всем имеющимся совпадениям, как если бы метод Split() был вызван без второго аргумента. Если во втором аргументе передать отрицательное число, метод Split() возбудит исключение ArgumentOutOfRangeException.

Когда методу передается второй аргумент, определяющий максимальное число строк в возвращаемом массиве, можно также указать необязательный третий аргумент, определяющий позицию в испытуемой строке, с которой следует начинать поиск совпадений. По сути, число, которое передается в третьем аргументе, представляет собой число символов в начале испытуемого текста, которые должны игнорироваться регулярным выражением. Этот аргумент может использоваться, когда строка уже обработана до некоторой позиции и необходимо выполнить разбиение оставшейся части строки.

Символы, которые будут пропущены регулярным выражением, тем не менее, будут добавлены в возвращаемый массив. Первая строка в массиве – это вся подстрока, предшествующая первому обнаруженному после указанной начальной позиции совпадению, включая символы, находящиеся перед начальной позицией. Значением третьего аргумента должно быть число в диапазоне от нуля до длины испытуемой строки. В противном случае метод Split() возбудит исключение

ArgumentOutOfRangeException. В отличие от метода Match() метод Split() не позволяет указывать длину подстроки, в которой можно выполнять поиск соответствий с регулярным выражением.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеется два совпадения, следующих непосредственно друг за другом, без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Java

Если требуется выполнить разбиение единственной строки, можно воспользоваться методом split() данной строки. Этот метод принимает регулярное выражение в виде единственного аргумента. Данный метод просто вызывает метод Pattern.compile("regex").split(subjectString).

Если необходимо выполнить разбиение множества строк, следует создать объект Pattern с помощью фабричного метода Pattern.compile(). При таком подходе достаточно будет скомпилировать регулярное выражение всего один раз. После этого можно будет вызывать метод split() полученного экземпляра класса Pattern, передавая ему испытуемую строку в качестве аргумента. В данном случае нет необходимости создавать экземпляр класса Matcher. Класс Matcher вообще не имеет метода split().

Метод Pattern.split() может принимать второй необязательный аргумент, а метод String.split() — нет. Во втором аргументе можно указать максимальное число фрагментов, которое требуется получить. Например, вызов Pattern.split(subject, 3) вернет массив, содержащий не более трех строк. Метод split() попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод split() выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов Pattern.split(subject, 1) вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеется два совпадения, следующих непосредственно друг за другом, без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Однако Java ликвидирует пустые строки в конце массива. Если необходимо, чтобы пустые строки включались в массив, методу `Pattern.split()` следует передать отрицательное число во втором аргументе. В этом случае разбиение будет выполнено по всем найденным совпадениям, и пустые строки в конце массива останутся на месте. Фактическая величина второго аргумента, когда в нем передается отрицательное число, не имеет значения. В языке Java невозможно одновременно ограничить число разбиений и оставить пустые строки в конце массива.

JavaScript

Чтобы выполнить разбиение строки в языке JavaScript, можно использовать метод `split()` этой строки. Чтобы получить массив строк, содержащий максимальное число разбиений, достаточно передать методу `split()` единственный аргумент с регулярным выражением. Во втором необязательном аргументе можно указать максимальное число строк в возвращаемом массиве. Это должно быть положительное число. Если в этом аргументе передать ноль, метод вернет пустой массив. Если опустить второй аргумент или передать в нем отрицательное число, строка будет разбита по всем найденным совпадениям. Наличие флага `/g` в регулярном выражении (рецепт 3.4) не оказывает никакого влияния.

К сожалению, ни один из популярных веб-броузеров не реализует все особенности метода `split()`, определяемые стандартом JavaScript. В частности, одни броузеры включают в возвращаемый массив совпадения с сохраняющими группами, другие – нет. Те броузеры, что включают совпадения с сохраняющими группами, некорректно обрабатывают ситуацию, когда группы не участвуют в совпадении. Чтобы избежать подобных проблем, в регулярных выражениях, передаваемых методу `split()`, желательно использовать только несохраняющие группы (рецепт 2.9).

Некоторые реализации JavaScript не включают пустые строки в возвращаемый массив. Пустые строки должны включаться в массив, когда два совпадения с регулярным выражением следуют непосредственно друг за другом или когда совпадение находится в начале или в конце разбиваемой строки. Поскольку решить эту проблему невозможно за счет простого изменения регулярного выражения, вероятно, надежнее будет использовать длинное решение на языке JavaScript, которое приводится в этом рецепте. Данное решение включает все пустые строки в возвращаемый массив, но его легко можно изменить так, чтобы пустые строки не включались.

Длинное решение является адаптацией рецепта 3.12. Оно добавляет в массив текст, находящийся между двумя совпадениями с регулярным выражением, и сами совпадения. Чтобы получить только текст между совпадениями, мы использовали информацию о совпадениях, как описывалось в рецепте 3.8.

Если вам потребуется реализация `String.prototype.split`, которая следует требованиям стандарта и работает во всех броузерах, обратите внимание на решение, предложенное Стивеном Левитаном (Steven Levithan), по адресу: *http://blog.stevenlevithan.com/archives/cross-browser-split*.

PHP

Для разбиения строки на массив строк по совпадениям с регулярным выражением можно использовать функцию `preg_split()`. Регулярное выражение передается этой функции в первом аргументе, а испытуемая строка – во втором. Если второй аргумент опущен, в качестве испытуемой строки используется специальная переменная `$_`.

В третьем необязательном аргументе можно указать максимальное число фрагментов, которое требуется получить. Например, вызов `preg_split($regex, $subject, 3)` вернет массив, содержащий не более трех строк. Функция `preg_split()` попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, функция `preg_split()` выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Если опустить третий аргумент или передать в нем значение `-1`, строка будет разбита по всем найденным совпадениям.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеется два совпадения, следующих непосредственно друг за другом, без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку. По умолчанию функция `preg_split()` включает пустые строки в возвращаемый массив. Если наличие пустых строк в массиве нежелательно, можно передать константу `PREG_SPLIT_NO_EMPTY` в четвертом аргументе.

Perl

Для разбиения строки на массив строк по совпадениям с регулярным выражением можно использовать функцию `split()`. Оператор регулярного выражения передается этой функции в первом аргументе, а испытуемая строка – во втором.

В третьем необязательном аргументе выражения можно указать максимальное число фрагментов, которое требуется получить. Например, вызов `split(/regex/, subject, 3)` вернет массив, содержащий не более трех

строк. Функция split() попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, функция split() выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного.

Если опустить третий аргумент, интерпретатор Perl самостоятельно определит соответствующий предел. Если результат присваивается переменной-массиву, как это реализовано в решении к данному рецепту, строка будет разбита по всем найденным совпадениям. Если результат присваивается списку скалярных переменных, будет установлен предел, равный числу переменных в списке плюс один. Другими словами, будет выполнена попытка заполнить все переменные, а возможный остаток строки будет отброшен. Например, инструкция ($one, $two, $three) = split(/,/) выполнит разбиение содержимого переменной $_ на 4 части.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеется два совпадения, следующих непосредственно друг за другом, без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Python

Чтобы разбить строку с помощью регулярного выражения, можно использовать функцию split() из модуля re. Регулярное выражение передается этой функции в первом аргументе, а испытуемая строка – во втором. Глобальная функция split() не позволяет передавать параметры регулярного выражения.

Функция re.split() вызывает функцию re.split() и затем метод split() объекта скомпилированного регулярного выражения. Этот метод имеет единственный обязательный аргумент: испытуемую строку.

Обе формы функции split() возвращают список с фрагментами текста между совпадениями с регулярным выражением. Обе принимают один необязательный аргумент, в котором можно указать максимальное число разбиений. Если опустить этот аргумент или передать в нем число ноль, строка будет разбита по всем найденным совпадениям. Если передать положительное число, оно будет определять максимальное количество совпадений с регулярным выражением, по которым следует разбить строку. Возвращаемый список будет содержать на одну стро-

ку больше, чем указано в аргументе. Последняя строка в списке будет представлять неразбитый остаток испытуемой строки после последнего совпадения. Если число найденных совпадений окажется меньше указанного значения, строка будет разбита по всем совпадениям без вывода сообщения об ошибке.

Ruby

Чтобы разбить испытуемую строку на массив строк по совпадениям с регулярным выражением, можно вызвать метод `split()` этой строки и передать ему регулярное выражение в первом аргументе.

Метод `split()` может принимать второй необязательный аргумент, в котором указывается максимальное число фрагментов, которое требуется получить. Например, вызов `subject.split(re, 3)` вернет массив, содержащий не более трех строк. Метод `split()` попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод `split()` выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов `split(re, 1)` вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеется два совпадения, следующих непосредственно друг за другом, без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Однако Ruby ликвидирует пустые строки в конце массива. Если необходимо, чтобы пустые строки включались в массив, методу `split()` следует передать отрицательное число во втором аргументе. В этом случае разбиение будет выполнено по всем найденным совпадениям, и пустые строки в конце массива останутся на месте. Фактическая величина второго аргумента, когда в нем передается отрицательное число, не имеет значения. В языке Ruby невозможно одновременно ограничить число разбиений и оставить пустые строки в конце массива.

См. также

Рецепт 3.20.

3.20. Разбиение строки, сохранение совпадений с регулярным выражением

Задача

Требуется выполнить разбиение строки с использованием регулярного выражения. В результате должен быть получен массив или список строк, содержащих фрагменты текста между совпадениями, а также с самими совпадениями.

Предположим, что требуется разбить строку с тегами HTML по этим тегам и дополнительно сохранить сами теги. Например, после разбиения строки I•like•bold•and•italic•fonts должен получиться массив из девяти строк: I•like•, , bold, , •and•, , italic, и •fonts.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string[] splitArray = Regex.Split(subjectString, "(<[^<>]*>)");
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex("(<[^<>]*>)");
string[] splitArray = regexObj.Split(subjectString);
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim SplitArray = Regex.Split(SubjectString, "(<[^<>]*>)")
```

В случае использования того же самого регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Dim RegexObj As New Regex("(<[^<>]*>)")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("<[^<>]*>");
Matcher regexMatcher = regex.matcher(subjectString);
int lastIndex = 0;
while (regexMatcher.find()) {
```

```
        resultList.add(subjectString.substring(lastIndex,
                                                regexMatcher.start()));
        resultList.add(regexMatcher.group());
        lastIndex = regexMatcher.end();
    }
    resultList.add(subjectString.substring(lastIndex));
```

JavaScript

```
var list = [];
var regex = /<[^<>]*>/g;
var match = null;
var lastIndex = 0;
while (match = regex.exec(subject)) {
    // Исключить вероятность попадания таких броузеров, как Firefox,
    // в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Добавить текст перед совпадением и само совпадение
    list.push(subject.substring(lastIndex, match.index), match[0]);
    lastIndex = match.index + match[0].length;
}
// Добавить остаток строки после последнего совпадения
list.push(subject.substr(lastIndex));
```

PHP

```
$result = preg_split('/(<[^<>]*>)/', $subject, -1,
                     PREG_SPLIT_DELIM_CAPTURE);
```

Perl

```
@result = split(m/(<[^<>]*>)/, $subject);
```

Python

При разбиении небольшого числа строк можно использовать глобальную функцию:

```
result = re.split("(<[^<>]*>)", subject))
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile("(<[^<>]*>)")
result = reobj.split(subject)
```

Ruby

```
list = []
lastindex = 0;
subject.scan(/<[^<>]*>/) {|match|
```

```
        list << subject[lastindex..$~.begin(0)-1];
        list << $&
        lastindex = $~.end(0)
    }
    list << subject[lastindex..subject.length()]
```

Обсуждение

Метод `Regex.Split()` в платформе .NET включает в возвращаемый массив текст, совпавший с сохраняющими группами. В версиях .NET 1.0 и 1.1 в массив включается только совпадение с первой сохраняющей группой. В версиях .NET 2.0 и выше совпадения со всеми сохраняющими группами включаются в массив в виде отдельных строк. Если требуется включить в массив совпадение со всем регулярным выражением, его нужно заключить в общую сохраняющую группу. В версиях .NET 2.0 и выше необходимо, чтобы все остальные группы были несохраняющими, в противном случае совпадения с ними также попадут в массив.

Сохраняющие группы не участвуют в подсчете совпадений, когда методу `Split()` передается аргумент, определяющий максимальное число разбиений. Вызов `regexObj.Split(subject, 4)` с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: `I•like•`, `<b>`, `bold`, `</b>`, `•and•`, `<i>` и `italic</i>•fonts`. Если бы регулярное выражение содержало 10 сохраняющих групп, вызов `regexObj.Split(subject, 4)` в версии .NET 2.0 или выше вернул бы массив с 34 строками.

В .NET отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы несохраняющими группами. Это легко можно сделать, передав параметр `RegexOptions.ExplicitCapture` и заменив в регулярном выражении все именованные сохраняющие группы обычными (то есть группами в простых круглых скобках).

Java

Метод `Pattern.split()` в языке Java не обеспечивает возможность добавления совпадений с регулярным выражением в возвращаемый массив. Вместо этого можно адаптировать решение из рецепта 3.12, чтобы добавлять в список не только совпадения с регулярным выражением, но и фрагменты текста между совпадениями. Чтобы получить текст между совпадениями, в решении используется дополнительная информация о совпадениях, как описывается в рецепте 3.8.

JavaScript

Функция `string.split()` в языке JavaScript не предоставляет возможность управлять включением совпадений с регулярным выражением в массив. В соответствии со стандартом JavaScript в массив должны добавляться совпадения со всеми сохраняющими группами. К сожалению, популярные веб-броузеры либо не делают этого, либо делают несовместимыми способами.

Чтобы получить решение, которое будет работать во всех броузерах, мы адаптировали решение для рецепта 3.12 и добавляем в список не только совпадения с регулярным выражением, но и фрагменты текста между совпадениями. Чтобы получить текст между совпадениями, в решении используется дополнительная информация о совпадениях, как описывается в рецепте 3.8.

PHP

Если функции `preg_split()` передать в четвертом аргументе значение `PREG_SPLIT_DELIM_CAPTURE`, в возвращаемый массив будут включены совпадения с сохраняющими группами. Значение `PREG_SPLIT_DELIM_CAPTURE` можно объединить со значением `PREG_SPLIT_NO_EMPTY` с помощью оператора `|`.

Сохраняющие группы не участвуют в подсчете совпадений, когда функции `preg_split()` передается третий аргумент, определяющий максимальное число разбиений. Если установить предел числа разбиений, равный четырем, для испытуемой строки и регулярного выражения, которые приводятся в решении для данного рецепта, функция вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: I•like•, , bold, , •and•, и italic•fonts.

Perl

Функция `split()` в языке Perl включает в массив совпадения со всеми сохраняющими группами. Если потребуется включить в массив совпадение со всем регулярным выражением, его нужно целиком заключить в сохраняющую группу.

Сохраняющие группы не участвуют в подсчете совпадений, когда функции `split()` передается аргумент, определяющий максимальное число разбиений. Вызов `split(/(<[^<>]*>)/, $subject, 4)` с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами

в регулярном выражении. Проще говоря, будет получен массив со строками: I•like•, , bold, , •and•, и italic•fonts. Если бы регулярное выражение содержало 10 сохраняющих групп, вызов split($regex, $subject, 4) вернул бы массив с 34 строками.

В языке Perl отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы несохраняющими группами.

Python

Функция split() в языке Python включает в массив совпадения со всеми сохраняющими группами. Если потребуется включить в массив совпадение со всем регулярным выражением, его нужно целиком заключить в сохраняющую группу.

Сохраняющие группы не участвуют в подсчете совпадений, когда функции split() передается аргумент, определяющий максимальное число разбиений. Вызов split("(<[^<>]*>)", subject, 3) с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: "I like", "", "bold", "", " and ", "" и "italic fonts". Если бы регулярное выражение содержало 10 сохраняющих групп, вызов split(regex, subject, 3) вернул бы массив с 34 строками.

В языке Python отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы несохраняющими группами.

Ruby

Метод String.split() в языке Ruby не предоставляет возможность управлять включением совпадений с регулярным выражением в массив. Вместо этого можно адаптировать решение из рецепта 3.12, чтобы добавлять в список не только совпадения с регулярным выражением, но и фрагменты текста между совпадениями. Чтобы получить текст между совпадениями, в решении используется дополнительная информация о совпадениях, как описывается в рецепте 3.8.

См. также

Описание сохраняющей и несохраняющей группировки в рецепте 2.9.

Описание именованной группировки в рецепте 2.11.

3.21. Построчный поиск

Задача

Традиционные инструменты grep применяют регулярное выражение к одной строке текста за один раз и отображают строки, совпавшие (или не совпавшие) с регулярным выражением. Допустим, что имеется некоторый массив строк или многострочный текст, который требуется обработать именно таким способом.

Решение

C#

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
string[] lines = Regex.Split(subjectString, "\r?\n");
```

Затем выполнить обход строк в массиве:

```
Regex regexObj = new Regex("regex pattern");
for (int i = 0; i < lines.Length; i++) {
    if (regexObj.IsMatch(lines[i])) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

VB.NET

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
Dim Lines = Regex.Split(SubjectString, "\r?\n")
```

Затем выполнить обход строк в массиве:

```
Dim RegexObj As New Regex("regex pattern")
For i As Integer = 0 To Lines.Length - 1
    If RegexObj.IsMatch(Lines(i)) Then
        'Строка Lines[i] соответствует регулярному выражению
    Else
        'Строка Lines[i] не соответствует регулярному выражению
    End If
Next
```

Java

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
String[] lines = subjectString.split("\r?\n");
```

Затем выполнить обход строк в массиве:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher("");
for (int i = 0; i < lines.length; i++) {
    regexMatcher.reset(lines[i]);
    if (regexMatcher.find()) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

JavaScript

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку. Как уже упоминалось в рецепте 3.19, некоторые броузеры исключают пустые строки из массива.

```
var lines = subject.split(/\r?\n/);
```

Затем выполнить обход строк в массиве:

```
var regexp = /regex pattern/;
for (var i = 0; i < lines.length; i++) {
    if (lines[i].match(regexp)) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

PHP

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
$lines = preg_split('/\r?\n/', $subject)
```

Затем выполнить обход строк в массиве:

```
foreach ($lines as $line) {
    if (preg_match('/regex pattern/', $line)) {
```

```
            // Строка line соответствует регулярному выражению
        } else {
            // Строка line не соответствует регулярному выражению
        }
    }
```

Perl

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
@lines = split(m/\r?\n/, $subject)
```

Затем выполнить обход строк в массиве:

```
foreach $line (@lines) {
    if ($line =~ m/regex pattern/) {
        # Строка $line соответствует регулярному выражению
    } else {
        # Строка $line не соответствует регулярному выражению
    }
}
```

Python

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
lines = re.split("\r?\n", subject);
```

Затем выполнить обход строк в массиве:

```
reobj = re.compile("regex pattern")
for line in lines[:]:
    if reobj.search(line):
        # Строка line соответствует регулярному выражению
    else:
        # Строка line соответствует регулярному выражению
```

Ruby

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
lines = subject.split(/\r?\n/)
```

Затем выполнить обход строк в массиве:

```
re = /regex pattern/
lines.each { |line|
```

```
    if line =~ re
        # Строка line соответствует регулярному выражению
    else
        # Строка line не соответствует регулярному выражению
}
```

Обсуждение

При необходимости выполнять построчную обработку данных можно обезопасить себя от большого числа возможных проблем, если разбить данные на отдельные строки вместо того, чтобы пытаться работать с одной большой строкой, содержащей символы перевода строки. После этого можно применять фактическое регулярное выражение к каждой строке в массие, не беспокоясь о том, что будет обнаружено совпадение одновременно с несколькими строками. Кроме того, такой подход упрощает отслеживание взаимоотношений между строками. Например, можно с помощью одного регулярного выражения отыскивать строки с заголовками, а с помощью другого – строки нижнего колонтитула. После этого при обнаружении разделяющих строк можно использовать третье регулярное выражение для поиска интересующих строк данных. На первый взгляд реализация такого алгоритма может показаться слишком трудоемкой, однако он достаточно прямолинейный, а программный код, реализующий его, обладает высокой эффективностью. Создать единственное регулярное выражение, которое будет отыскивать заголовки, данные и нижние колонтитулы, намного сложнее, и такое выражение будет работать намного медленнее.

Кроме того, при построчной обработке текста легко можно инвертировать регулярное выражение. Нет простого способа реализовать в регулярном выражении условие «соответствует строке, которая не содержит это или это слово». Легко инвертировать можно только символьные классы. Но когда текст уже разбит на строки, поиск строк, которые не содержат некоторое слово, становится таким же простым делом, как поиск текстового литерала во всех строках и удаление строк, в которых присутствует искомое слово.

В рецепте 3.19 показано, как легко можно разбить строку, превратив ее в массив строк. Регулярному выражению `‹\r\n›` соответствуют пары символов CR и LF, которые разделяют строки в Microsoft Windows. Регулярному выражению `‹\n›` соответствует символ LF, разделяющий строки в системе UNIX и ее производных, таких как Linux и даже OS X. Поскольку эти два регулярных выражения по сути являются простым текстом, для разбиения текста на строки можно даже вообще не использовать их. Если язык программирования позволяет разбивать строки по текстовым литералам, обязательно следует использовать эту возможность.

Если заранее нет уверенности, как разделяются строки в используемых данных, можно выполнить разбиение текста с помощью регуляр-

ного выражения ‹\r?\n›. Так как здесь символ CR объявлен необязательным, это регулярное выражение будет соответствовать как границам строк CRLF в Windows, так и границам строк LF в UNIX.

Как только строки окажутся в массиве, их легко можно будет обойти в цикле. Внутри цикла можно будет определить, какие строки соответствуют регулярному выражению, а какие нет, следуя рецепту 3.5.

См. также

Рецепты 3.11 и 3.19.

4

Проверка и форматирование

В этой главе содержатся рецепты реализации проверки и форматирования наиболее типичных видов данных, которые вводятся пользователем. Некоторые решения демонстрируют, как выполнить проверку данных, допускающих альтернативные форматы представления, такие как почтовые индексы США, которые могут состоять из пяти или из девяти цифр. Другие предназначены для преобразования введенных данных, таких как номера телефонов, даты и номера кредитных карт, в наиболее общеупотребимые формы представления.

Эти рецепты помогут вам не только отклонять неправильный ввод, но и повысить удобство ваших приложений для пользователей. Такое сообщение, как «вводить без дефисов и пробелов», рядом с полем ввода номера телефона или номера кредитной карты, нередко просто игнорируются пользователями. К счастью, во многих случаях регулярные выражения помогут обеспечить пользователям возможность вводить данные в удобных и привычных для них форматах и не потребуют больших усилий с вашей стороны.

В некоторых языках программирования имеются функциональные возможности, похожие на те, что предлагаются в ряде рецептов этой главы, и реализованные через их собственные классы или библиотеки. В конкретных ситуациях бывает предпочтительнее использовать эти встроенные возможности, поэтому мы будем указывать на них по ходу обсуждения.

4.1. Проверка адресов электронной почты

Задача

Имеется форма на веб-сайте или диалог в приложении, где у пользователя запрашивается адрес электронной почты. Необходимо с помощью регулярного выражения проверить корректность адреса, прежде

чем пытаться отправить сообщение по этому адресу. Это позволит сократить число возвратов недоставленных сообщений.

Решение

Простое

Первое решение выполняет очень простую проверку. Оно проверяет лишь наличие символа @ и отсутствие пробельных символов в адресе:

```
^\S+@\S+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A\S+@\S+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Простая проверка с ограничением допустимого набора символов

Доменное имя, часть адреса, следующая за символом @, может содержать только те символы, которые допускается использовать в доменных именах. *Имя пользователя*, часть адреса, предшествующая символу @, может содержать только те символы, которые допускается использовать в именах пользователей электронной почты, количество которых намного меньше, чем могут принимать большинство клиентов и серверов электронной почты:

```
^[A-Z0-9+_.-]+@[A-Z0-9.-]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9+_.-]+@[A-Z0-9.-]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Простая проверка без ограничения допустимого набора символов

Данное регулярное выражение является расширенной версией предыдущего и считает допустимым применять в именах пользователей более широкий набор символов. Не все программы электронной почты в состоянии обрабатывать все эти символы, но мы включили все символы, которые считаются допустимыми в соответствии с документом RFC 2822, в котором определяется формат сообщений электронной почты. Среди допустимых символов имеются такие, которые представляют угрозу, например апострофы (') и символы вертикальной черты

(|). Символы, которые могут иметь специальное назначение, обязательно необходимо экранировать при передаче адресов электронной почты другим программам, чтобы предотвратить такие виды нападений, как инъекция кода SQL:

```
^[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Без ведущих, завершающих или следующих друг за другом точек

В именах пользователей и в доменных именах допускается использовать один или более символов точки, но две точки, следующие подряд, считаются недопустимыми. Кроме того, точки не могут быть первыми и последними символами в имени пользователя и в доменном имени:

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Имя домена верхнего уровня может содержать от двух до шести букв

Это регулярное выражение добавляет к предыдущим версиям следующее требование к доменному имени: оно должно включать как минимум одну точку и после точки могут следовать только алфавитные символы. То есть имя домена должно состоять, как минимум, из двух уровней, например `secondlevel.com` или `thirdlevel.secondlevel.com`. Имя домена верхнего уровня `.com` должно содержать от двух до шести алфавитных символов. Все коды стран, используемые в качестве домена верхнего уровня, состоят из двух букв. Имена универсальных доменов верхнего уровня должны содержать от трех (`.com`) до шести букв (`.museum`):

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Об адресах электронной почты

Если на ваш взгляд для чего-то концептуально простого, такого как проверка адресов электронной почты, должно существовать простое решение, охватывающее все возможные вариации, то вы глубоко неправы. Этот рецепт представляет собой яркий пример ситуации, когда прежде чем браться за создание регулярного выражения, необходимо *точно* решить, что именно требуется проверить. Не существует универсального правила, согласно которому можно было бы однозначно сказать, какой адрес является корректным, а какой – нет. Все зависит от определения, *что считать правильным*.

Согласно документу RFC 2822, в котором определяется синтаксис адресов электронной почты, адрес `asdf@asdf.asdf` является корректным. Но он не будет считаться правильным, если под правильным понимать адрес, который примет сообщение электронной почты. В мире не существует домена верхнего уровня `asdf`.

Главная проблема проверки состоит в том, что невозможно точно утверждать, что адрес `john.doe@somewhere.com` правильный и способен принимать сообщения электронной почты, не попытавшись отправить письмо по этому адресу и получить ответ на него. Но даже после этого нельзя быть полностью уверенным в чем-либо, потому что домен `somewhere.com` может просто уничтожить послание на несуществующий почтовый ящик, не сообщая об ошибке, или John Doe может нажать клавишу Delete на клавиатуре, или письмо может быть уничтожено спам-фильтром.

Поскольку в конечном счете все равно придется проверять существование адреса, послав на него сообщение, можно ограничиться более простым и менее строгим регулярным выражением. Принимать некорректные адреса может оказаться предпочтительнее, чем вызывать недовольство пользователей, отвергая корректные адреса. Учитывая это обстоятельство, можно отдать предпочтение регулярному выражению, выполняющему «простую проверку без ограничения допустимого набора символов». Хотя оно и пропускает то, что не является адресом электронной почты, например `#$%@.-`, тем не менее, регулярное выражение само по себе очень простое и быстрое, и оно никогда не заблокирует допустимые адреса.

Если надо избежать слишком большого числа возвратов недоставленных писем и при этом не блокировать действительные адреса электронной почты, регулярное выражение «имя домена верхнего уровня может содержать от двух до шести букв» на стр. 276 будет лучшим выбором.

Сложность регулярного выражения также необходимо принимать во внимание. Если выполняется проверка ввода пользователя, предпочтение стоит отдать более сложному регулярному выражению, потому что пользователь может ввести все, что угодно. Но если адреса извлекаются из файлов базы данных, в отношении которых заранее известно, что они содержат корректные адреса, можно использовать очень простое регулярное выражение, которое просто отделяет адреса электронной почты от остальных данных. В этом случае достаточно будет даже регулярного выражения из раздела «Простое».

Наконец, необходимо учитывать пригодность регулярного выражения в будущем. В прошлом можно было ограничить имя домена верхнего уровня двухбуквенными кодами стран и перечислением универсальных доменов, например ‹com|net|org|mil|edu›. Однако, учитывая постоянное появление новых доменов, такие регулярные выражения быстро становятся непригодными.

Синтаксис регулярного выражения

Регулярные выражения, представленные в этом рецепте, демонстрируют все основные элементы синтаксиса регулярных выражений в действии. Если вы изучили эти элементы в главе 2, вы уже можете выполнить 90% заданий, которые лучше всего решаются с помощью регулярных выражений.

Все регулярные выражения требуют, чтобы был включен режим нечувствительности к регистру символов. В противном случае допустимы будут только символы верхнего регистра. В этом режиме можно будет вместо ‹[A-Za-z]› вводить ‹[A-Z]›, экономя время на нажатиях клавиш. При использовании одного из двух последних выражений режим нечувствительности к регистру символов становится особенно удобным. В противном случае пришлось бы заменить все символы ‹X› на ‹[Xx]›.

Как отмечалось в рецепте 2.3, метасимволы ‹\S› и ‹\w› являются сокращенными формами записи символьных классов. Метасимволу ‹\S› соответствуют любые непробельные символы, а метасимволу ‹\w› соответствуют символы слов.

Конструкции ‹@› и ‹\.› соответствуют символу @ и точке, соответственно. Поскольку точка является метасимволом, при использовании ее за пределами символьных классов ее необходимо экранировать символом обратного слэша. Символ @ не имеет специального назначения ни в одном из диалектов регулярных выражений, рассматриваемых в этой книге. Перечень всех метасимволов, которые необходимо экранировать, приводится в рецепте 2.1.

‹[A-Z0-9.-]› и другие последовательности, заключенные в квадратные скобки, являются символьными классами. Данному символьному классу соответствует любая буква от A до Z, любая цифра от 0 до 9, а также точка и дефис. Обычно дефис используется в символьных классах для определения диапазона, но когда дефис стоит последним символом в символьном классе, он интерпретируется как литерал.

О символьных классах рассказывается в рецепте 2.3, включая возможность комбинирования с сокращенными формами записи, как в этом случае: ‹[\w!#$%&'*+/=?`{|}~^.-]›. Данному классу соответствует символ слова, а также любой из 19 перечисленных знаков пунктуации.

Когда символы ‹+› и ‹*› используются за пределами символьных классов, они играют роль квантификаторов. Знак плюс повторяет предшествующий ему элемент один или более раз. В этих регулярных выражениях квантифицируемыми элементами являются символьные классы и иногда – группы. Таким образом, подвыражению ‹[A-Z0-9.-]+› соответствует одна или более букв, цифр, точек и/или дефисов.

Например, группе ‹(?:[A-Z0-9-]+\.)+› соответствуют одна или более алфавитных символов, цифр и/или дефисов, за которыми следует один литерал точки. Символ плюс обеспечивает повторение этой группы один или более раз. Группа должна обнаруживать, как минимум, одно совпадение, но может совпадать столько раз, сколько это возможно. Более подробно механизм работы подобных конструкций описывается в рецепте 2.12.

‹(?:group)› – это несохраняющая группа. Она используется для группировки части регулярного выражения, благодаря чему можно применять квантификатор к группе целиком. Сохраняющая группа ‹(group)› делает то же самое, но имеет более простой синтаксис. Во всех регулярных выражениях, которые использовались до сих пор, можно заменить все комбинации символов ‹(?:› на ‹(› и результат от этого не изменится.

Однако нам не требуется сохранять какие-либо части адресов электронной почты, а несохраняющая группировка обладает большей эффективностью, хотя и делает регулярное выражение менее удобочитаемым. Полная информация о сохраняющих и несохраняющих группах приводится в рецепте 2.9.

Якорные метасимволы ‹^› и ‹$› вынуждают регулярное выражение искать совпадение в начале и в конце испытуемого текста, соответственно. Если поместить все регулярное выражение между этими метасимволами, тем самым будет поставлено требование совпадения с регулярным выражением всей испытуемой строки целиком.

Это очень важное условие при проверке ввода пользователя. Едва ли кто-то пожелает принять строку drop database; -- joe@server.com haha! как допустимый адрес электронной почты. Без якорных метасимволов эта строка будет соответствовать всем предыдущим регулярным вы-

ражениям, поскольку они будут обнаруживать совпадение joe@server.com в середине данной строки. Подробнее об этом рассказывается в рецепте 2.5. Это также объясняет, почему должен быть отключен режим «символам ^ и $ соответствуют границы строк».

В диалекте регулярных выражений Ruby символ крышки и знак доллара всегда совпадают с границами строк. Регулярные выражения, использующие символ крышки и знак доллара, будут корректно работать в языке Ruby, но только если проверяемая строка не содержит символов перевода строки. Если строка может содержать символы перевода строки, все регулярные выражения, использующие метасимволы ‹^› и ‹$›, будут совпадать с адресом электронной почты в строке drop database; -- LFjoe@server.comLF haha!, где обозначение LF представляет символ перевода строки.

Чтобы избежать этого, следует использовать якорные метасимволы ‹\A› и ‹\Z›. Они соответствуют только началу и концу строки независимо от используемых режимов во всех диалектах, рассматриваемых в этой книге, за исключением диалекта JavaScript. Диалект JavaScript вообще не поддерживает метасимволы ‹\A› и ‹\Z›. Описание этих метасимволов можно найти в рецепте 2.5.

Проблема выбора между ‹^› и ‹$›, и ‹\A› и ‹\Z› свойственна для всех регулярных выражений, осуществляющих проверку ввода. Многие из них приводятся в этой книге. Хотя периодически мы будем напоминать об этой проблеме, тем не менее, мы не будем делать это постоянно или приводить отдельные решения для JavaScript и Ruby в каждом рецепте. Во многих случаях мы будем показывать единственное решение, использующее символ крышки и знак доллара, и указывать диалект Ruby в списке совместимых диалектов. Если вы пользуетесь Ruby, не забывайте применять метасимволы ‹\A› и ‹\Z›, если необходимо избежать совпадения с одной строкой в многострочном тексте.

Пошаговое создание регулярного выражения

Этот рецепт демонстрирует, как можно создавать регулярное выражение шаг за шагом. Этот прием особенно удобно использовать при использовании интерактивного инструмента тестирования регулярных выражений, таких как RegexBuddy.

Для начала надо загрузить в инструмент набор допустимых и недопустимых примеров данных. В данном случае это могут быть списки допустимых и недопустимых адресов электронной почты.

Затем пишется простое регулярное выражение, которое совпадает со всеми допустимыми адресами. На этом этапе можно игнорировать возможные совпадения с недопустимыми адресами. Конструкция ‹^\S+@\

S+$› уже определяет основную структуру адреса электронной почты: имя пользователя, символ @ и доменное имя.

Определив основную структуру текстового шаблона, можно приступать к усовершенствованию каждой части структуры, пока регулярное выражение не прекратит совпадать с недопустимыми данными. Если предполагается применять регулярное выражение к уже существующим данным, это задание можно будет выполнить достаточно быстро. Если предполагается, что регулярное выражение будет использоваться для обработки ввода пользователя, работа над регулярным выражением, пока оно не окажется достаточно строгим, чтобы соответствовать только допустимым данным, может оказаться довольно сложным делом.

Варианты

Если задача состоит не в том, чтобы определять, является ли вся введенная строка адресом электронной почты, а в том, чтобы отыскать адреса в объемном тексте, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›. Однако простое удаление их из регулярного выражения не станет правильным решением. В этом случае получившееся регулярное выражение, которое требует, чтобы имя домена верхнего уровня содержало только алфавитные символы, будет совпадать, например, с фрагментом asdf@asdf.as в строке asdf@asdf.as99. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что начало имени пользователя и имя домена верхнего уровня не могут быть частью более длинных слов.

Это легко реализуется с помощью пары метасимволов границ слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например, в результате такой замены выражение ‹^[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$› превратится в выражение ‹\b[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\b›.

В действительности данное регулярное выражение объединяет в себе шаблон имени пользователя из примера «Простая проверка с ограничением допустимого набора символов» на стр. 275 и шаблон доменного имени из примера «Имя домена верхнего уровня может содержать от двух до шести букв» на стр. 276. Мы на практике убедились в эффективности этого регулярного выражения.

См. также

Документ RFC 2822 определяет структуру и синтаксис сообщений электронной почты, включая адреса, используемые в сообщениях. Загрузить RFC 2822 можно по адресу *http://www.ietf.org/rfc/rfc2822.txt*.

4.2. Проверка и форматирование телефонных номеров

Задача

Необходимо определить, является ли ввод пользователя номером телефона в одном из форматов, используемых в Северной Америке, включая код города. В число этих форматов входят: 1234567890, 123-456-7890, 123.456.7890, 123 456 7890, (123) 456 7890 и их комбинации. Если ввод представляет собой допустимый номер телефона, его следует привести к стандартному формату (123) 456-7890, чтобы обеспечить единообразие отображения телефонных номеров.

Решение

С помощью регулярного выражения легко можно проверить, ввел ли пользователь что-то похожее на допустимый телефонный номер. Применив сохраняющие группы для сохранения каждого набора цифр, то же самое регулярное выражение можно будет использовать для приведения испытуемого текста к желаемому формату.

Регулярное выражение

```
^\(?([0-9]{3})\)?[-.•]?([0-9]{3})[-.•]?([0-9]{4})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
($1)•$2-$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
(\1)•\2-\3
```

Диалекты замещающего текста: Python, Ruby

C#

```
Regex regexObj =
    new Regex(@"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$");

if (regexObj.IsMatch(subjectString)) {
    string formattedPhoneNumber =
        regexObj.Replace(subjectString, "($1) $2-$3");
} else {
    // Недопустимый номер телефона
}
```

JavaScript

```
var regexObj = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;
if (regexObj.test(subjectString)) {
    var formattedPhoneNumber =
        subjectString.replace(regexObj, "($1) $2-$3");
} else {
    // Недопустимый номер телефона
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецептах 3.5 и 3.15.

Обсуждение

Этому регулярному выражению соответствуют три группы цифр. Первая группа может быть заключена в необязательные круглые скобки, и за первыми двумя группами могут следовать символы-разделители трех видов (дефис, точка или пробел). Ниже приводится структура регулярного выражения, разбитого на отдельные части, с опущенными дополнительными группами цифр:

```
^        # Проверка совпадения с началом строки.
\(       # Соответствует литералу "("...
  ?      #     ноль или один раз.
(        # Сохранение вложенного совпадения для обратной ссылки 1...
  [0-9]  #     Соответствует цифре...
    {3}  #         точно три раза.
)        # Конец сохраняющей группы 1.
\)       # Соответствует литералу ")"...
  ?      #     ноль или один раз.
[-. ]    # Соответствует одному символу из набора "-. "...
  ?      #     ноль или один раз.
...      # [Соответствует остальным цифрам и разделителю.]
$        # Проверка совпадения с концом строки.
```

Рассмотрим каждую из этих частей более подробно.

Символы ‹^› и ‹$› в начале и в конце регулярного выражения – это метасимволы специального назначения, которые называются *якорными* метасимволами, или *проверками*. Этим метасимволам соответствует не текст, а определенные позиции в тексте. В частности, метасимволу ‹^› соответствует начало текста, а метасимволу ‹$› – конец. Они гарантируют, что регулярное выражение не будет совпадать с более длинным текстом, таким как 123-456-78901.

Как уже неоднократно говорилось, круглые скобки в регулярных выражениях имеют специальное назначение, но в данном случае необхо-

димо разрешить пользователю вводить круглые скобки, и регулярное выражение должно распознавать их. Это наглядный пример ситуации, когда необходимо использовать символ обратного слэша для экранирования специальных символов, чтобы механизм регулярных выражений интерпретировал их как обычные символы. То есть последовательностям ‹\(› и ‹\)›, охватывающим первую группу цифр, соответствуют символы круглых скобок в тексте. Обе последовательности сопровождаются вопросительным знаком, который делает совпадения с ними необязательными. Подробнее о знаке вопроса мы поговорим после того, как рассмотрим остальные части этого регулярного выражения.

Круглые скобки, перед которыми не стоят символы обратного слэша, образуют сохраняющие группы и используются для запоминания фрагментов текста, совпавших с подвыражениями, заключенными в скобки, чтобы позднее можно было извлечь эти фрагменты. В данном случае обратные ссылки на сохраненные фрагменты используются в замещающем тексте, благодаря чему мы легко можем привести телефонный номер к нужному формату.

В этом регулярном выражении используются еще два типа элементов – символьные классы и квантификаторы. Символьные классы позволяют описать совпадение с любым символом из определенного набора. Конструкция ‹[0-9]› – это символьный класс, которому соответствует одна цифра. Все диалекты регулярных выражений, описываемые в этой книге, позволяют использовать сокращенную форму записи символьного класса ‹\d›, которой также соответствует одна цифра, но в некоторых диалектах метасимволу ‹\d› соответствует цифра из любого набора символов или алфавита, а это не совсем то, что нам в данном случае требуется. Подробнее о метасимволе ‹\d› рассказывается в рецепте 2.3.

Конструкция ‹[-.•]› – это еще один символьный класс, которому соответствует любой из трех допустимых символов-разделителей. Местоположение символа дефиса в символьном классе имеет большое значение. Когда дефис находится между двумя символами, он образует диапазон символов, как в символьном классе ‹[0-9]›. Другой способ обеспечить интерпретацию дефиса в символьном классе как литерала заключается в том, чтобы экранировать его символом обратного слэша. Так, символьный класс ‹[.\-•]› эквивалентен предыдущему.

Наконец, квантификаторы позволяют описать повторяемость элемента регулярного выражения или группы. ‹{3}› – это квантификатор, который говорит, что предшествующий ему элемент должен повторяться точно три раза. То есть регулярное выражение ‹[0-9]{3}› эквивалентно выражению ‹[0-9][0-9][0-9]›, но оно короче и легче читается. Знак вопроса (упоминавшийся выше) – это специальный квантификатор, который обеспечивает повторение предшествующего ему элемента ноль или один раз. Этот квантификатор можно также записать как ‹{0,1}›.

Любой квантификатор, допускающий нулевое число повторений предшествующего элемента, фактически делает этот элемент необязательным. Поскольку знак вопроса указан после каждого разделителя, допускается, что цифры в телефонном номере будут следовать непосредственно друг за другом.

Следует заметить, что несмотря на то, что в данном рецепте говорится о форматах представления телефонных номеров, используемых в Северной Америке, фактически он предназначен для работы с номерами в системе *NANP* (North American Numbering Plan – североамериканская схема присвоения номеров). NANP – это схема присвоения номеров для стран, в которых используется код страны «1». В их число входят Соединенные Штаты и их территории, Канада, Бермудские острова и 16 государств Карибского бассейна. Мексика и государства Центральной Америки не входят в эту систему.

Варианты

Устранение недопустимых номеров

Итак, регулярному выражению соответствуют любые 10-значные номера. Если необходимо ограничиться совпадением только с номерами, соответствующими североамериканской схеме присвоения номеров, необходимо соблюсти следующие требования:

- *Код зоны* должен начинаться с цифры в диапазоне 2–9, за которой следует цифра из диапазона 0–8 и третья цифра может быть любой.
- Вторая группа из трех цифр, известная как *центральная станция*, или *коммутатор*, должна начинаться с цифры в диапазоне 2–9, за которой следуют любые две цифры.
- Последняя группа из четырех цифр, известная как *код станции*, не имеет ограничений.

Эти требования легко можно реализовать с помощью нескольких символьных классов.

```
^\(?([2-9][0-8][0-9])\)?[-.•]?([2-9][0-9]{2})[-.•]?([0-9]{4})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Помимо только что перечисленных основных правил существуют правила для различных зарезервированных, неприсвоенных и ограниченных номеров. Если не стоит задача отфильтровывать максимально возможное число телефонных номеров, то не надо и пытаться отбрасывать неиспользуемые номера. Новые коды зон, соответствующие вышеперечисленным правилам, появляются регулярно; при этом, даже если телефонный номер соответствует правилам, это еще не значит, что он действительно используется.

Поиск телефонных номеров в документах

Два небольших изменения в предыдущем регулярном выражении позволят использовать его для поиска телефонных номеров внутри текста большого объема:

```
\(?\b([0-9]{3})\)?[-.•]?([0-9]{3})[-.•]?([0-9]{4})\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Здесь были убраны проверки ‹^› и ‹$› привязывавшие регулярное выражение к началу и концу текста. На их место были вставлены метасимволы границы слова (‹\b›), чтобы гарантировать, что совпавший фрагмент текста не является частью более длинного числа или слова.

Подобно метасимволам ‹^› и ‹$›, метасимвол ‹\b› также соответствует определенной позиции, а не какому-то фактическому тексту. В частности, метасимволу ‹\b› соответствует позиция между символом слова и либо не символом слова, либо началом и концом текста. Алфавитные, цифровые символы и символ подчеркивания – все они считаются символами слова (рецепт 2.6).

Обратите внимание, что первый метасимвол границы слова стоит после необязательной открывающей скобки. Это важно, так как между двумя не символами слова, например между пробелом и следующей за ним открывающей круглой скобкой, не может быть границы слова. Проверка первой границы слова обретает важное значение, только когда совпавший номер не окружен круглыми скобками, так как позиция между открывающей круглой скобкой и первой цифрой номера всегда будет совпадать с метасимволом границы слова.

Допустимость ведущего символа «1»

Можно предусмотреть возможность наличия в номере необязательной первой цифры «1», обозначающей код страны (который используется всеми странами в регионе, охватываемом североамериканской схемой присвоения номеров), дополнив регулярное выражение, как показано ниже:

```
^(?:\+?1[-.•]?)?\(?([0-9]{3})\)?[-.•]?([0-9]{3})[-.•]?([0-9]{4})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В дополнение к форматам телефонных номеров, показанных выше, этому регулярному выражению также соответствуют такие строки, как +1 (123) 456-7890 и 1-123-456-7890. Оно использует несохраняющую группу, которая оформляется как ‹(?:…)›. Когда вопросительный знак следует за неэкранированной открывающей круглой скобкой, как в данной конструкции, он не является квантификатором, – вместо этого он помогает идентифицировать тип группировки. Стандартные сохраняющие

группы требуют от механизма регулярных выражений следить за обратными ссылками, поэтому несохраняющие группы имеют более высокую эффективность, так как совпавший с группой текст не требуется сохранять для обеспечения возможности сослаться на него позднее. Еще одна причина, почему здесь используется несохраняющая группа, состоит в желании сохранить неизменной строку с замещающим текстом. Если бы в регулярное выражение была добавлена сохраняющая группа, пришлось бы в замещающем тексте, показанном выше в этом рецепте, обратную ссылку `$1` заменить на `$2` (и так далее).

В эту версию регулярного выражения была добавлена конструкция `‹(?:\+?1[-.•]?)?›`. Символу «1» в этом шаблоне предшествует необязательный знак плюс, а вслед за символом «1» может следовать один из трех символов-разделителей (дефис, точка или пробел). Вся добавленная несохраняющая группа также объявлена необязательной, но, так как символ «1» внутри группы является обязательным, предшествующий ему символ плюс и следующий за ним символ-разделитель являются недопустимыми, если в номере отсутствует ведущий символ «1».

Допустимость семизначных номеров телефонов

Чтобы позволить совпадение с номерами телефонов, в которых отсутствует код зоны, необходимо первую группу цифр вместе с окружающими их круглыми скобками и следующим за ними символом-разделителем заключить в необязательную несохраняющую группу:

```
^(?:\(?([0-9]{3})\)?[-.•]?)?([0-9]{3})[-.•]?([0-9]{4})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поскольку теперь код зоны уже не является обязательной частью совпадения, простая замена любого совпадения замещающим текстом «($1)•$2-$3» может дать в результате, например, такую строку: `() 123-4567`, с пустыми круглыми скобками. Чтобы обойти эту проблему, нужно добавить программный код, который будет проверять наличие совпадения с первой группой и соответствующим образом изменять замещающий текст.

См. также

В рецепте 4.3 демонстрируется, как выполнять проверку телефонных номеров в международном формате.

Североамериканская схема присвоения номеров (NANP) действует на территории США, Канады, Бермудских островов и 16 государств Карибского бассейна. Дополнительную информацию об этой схеме можно найти на сайте *http://www.nanpa.com*.

4.3. Проверка международных телефонных номеров

Задача

Необходимо проверить корректность международного телефонного номера. Эти номера телефонов должны начинаться со знака плюс, за которым следует код страны, а затем телефонный номер внутри страны.

Решение

Регулярное выражение

```
^\+(?:[0-9]•?){6,14}[0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
function validate (phone) {
    var regex = /^\+(?:[0-9] ?){6,14}[0-9]$/;
    if (regex.test(phone)) {
        // Допустимый международный телефонный номер
    } else {
        // Недопустимый международный телефонный номер
    }
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

Правила и соглашения, используемые при выводе международных телефонных номеров, значительно варьируются в зависимости от страны, поэтому довольно сложно реализовать надежную проверку международных телефонных номеров без принятия строгого формата. К счастью, существует стандартная форма записи, определяемая документом ITU-T E.123. Данная форма записи требует, чтобы международные номера телефонов включали начальный знак плюс (известный как символ, соответствующий *международному префиксу*) и для отделения групп цифр допускает использовать только пробелы. Несмотря на то, что внутри телефонного номера может употребляться символ тильды (~) для обозначения дополнительного сигнала готовности к набору но-

мера, его следует исключить из регулярного выражения, поскольку это всего лишь процедурный элемент (другими словами, он фактически не набирается) и используется достаточно редко. Согласно системе нумерации телефонных номеров (ITU-T E.164) номер не может состоять более, чем из 15 цифр. Самые короткие международные телефонные номера содержат семь цифр.

Учитывая вышесказанное, рассмотрим регулярное выражение, опять же разбитое на фрагменты. Так как эта версия записана с использованием режима свободного форматирования, литералы символов пробелов были заменены на последовательность ‹\x20›:

```
^          # Проверка совпадения с началом строки.
\+         # Соответствует литералу символа “+”.
(?:        # Несохраняющая группа...
  [0-9]    #    Соответствует цифре.
  \x20     #    Соответствует символу пробела...
    ?      #         Ноль или один раз.
)          # Конец несохраняющей группы.
  {6,14}   #     Повторить предыдущую группу от 6 до 14 раз.
[0-9]      # Соответствует цифре.
$          # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Якорные метасимволы ‹^› и ‹$› по границам регулярного выражения гарантируют, что регулярное выражение будет совпадать со всем испытуемым текстом. Несохраняющей группе, заключенной в конструкцию ‹(?:…)›, соответствует одна цифра, за которой следует необязательный символ пробела. Повторением этой группы с помощью интервального квантификатора ‹{6,14}› гарантируется соблюдение правила о минимальном и максимальном количестве цифр в номере, при этом допускается появление пробела-разделителя в любом месте внутри номера. Второй символьный класс ‹[0-9]› завершает реализацию правила о количестве цифр (увеличивая допустимое число цифр с интервала от 6 до 14 до интервала от 7 до 15) и гарантирует, что телефонный номер не будет заканчиваться пробелом.

Варианты

Проверка международного телефонного номера в формате EPP

```
^\+[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение следует формату записи международных телефонных номеров, определяемому расширяемым протоколом представления информации (Extensible Provisioning Protocol, EPP). Этот протокол был разработан относительно недавно (работа над ним была завершена в 2004 году) и предназначен для обмена информацией между регистратурами и регистраторами доменных имен. Он используется продолжающими расширяться регистратурами доменных имен, включая *.com*, *.info*, *.net*, *.org* и *.us*. Особая его значимость обусловлена тем, что формат EPP записи международных телефонных номеров приобретает все большее распространение и признание, благодаря чему он представляет собой серьезный альтернативный формат хранения (и проверки) международных телефонных номеров.

Для представления телефонных номеров протоколом EPP используется формат *+CCC.NNNNNNNNNNxEEEE*, где *C* – это код страны, состоящий от 1 до 3 цифр, фрагмент *N* может включать до 14 цифр и *E* – это необязательное расширение. Ведущий знак плюс и точка, следующая за кодом страны, являются обязательными элементами. Символ «x» должен присутствовать только при наличии расширения.

См. также

В рецепте 4.2 описываются дополнительные возможности проверки номеров в североамериканской схеме нумерации.

Документ «ITU-T recommendation E.123» («Notation for national and international telephone numbers, e-mail addresses and Web addresses» – форма записи национальных и международных телефонных номеров, адресов электронной почты веб-адресов) можно загрузить по адресу *http://www.itu.int/rec/T-REC-E.123*.

Документ «ITU-T Recommendation E.164» («The international public telecommunication numbering plan» – международная схема нумерации телекоммуникаций) можно загрузить по адресу *http://www.itu.int/rec/T-REC-E.164*.

Национальные схемы нумерации можно загрузить по адресу *http://www.itu.int/ITU-T/inr/nnp*.

Документ RFC 4933 определяет синтаксис и семантику контактных идентификаторов EPP, включая международные телефонные номера. Загрузить RFC 4933 можно по адресу *http://tools.ietf.org/html/rfc4933*.

4.4. Проверка дат в традиционных форматах

Задача

Необходимо выполнить проверку дат в традиционных форматах mm/dd/yy, mm/dd/yyyy, dd/mm/yy и dd/mm/yyyy. Необходимо использо-

вать простое регулярное выражение, которое просто проверяет, выглядит ли ввод как дата, не пытаясь проверить корректность таких дат, как 31 февраля.

Решение

Соответствует любому из указанных форматов представления дат, допуская отсутствие ведущих нулей:

```
^[0-3]?[0-9]/[0-3]?[0-9]/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует любому из указанных форматов представления дат, требуя наличие ведущих нулей:

```
^[0-3][0-9]/[0-3][0-9]/(?:[0-9][0-9])?[0-9][0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует форматам m/d/yy и mm/dd/yyyy. Допускает любые комбинации из одной или двух цифр в номере дня и месяца и две или четыре цифры в номере года:

```
^(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует формату mm/dd/yyyy, требует наличия ведущих нулей:

```
^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует форматам d/m/yy и dd/mm/yyyy. Допускает любые комбинации из одной или двух цифр в номере дня и месяца и две или четыре цифры в номере года:

```
^(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует формату dd/mm/yyyy, требует наличия ведущих нулей:

```
^(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует любому из этих форматов представления дат с большей точностью, допускает отсутствие ведущих нулей:

```
^(?:(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])|↵
(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9]))/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует любому из этих форматов представления дат с большей точностью, требует наличия ведущих нулей:

```
^(?:(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])|↵
(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9]))/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Режим свободного форматирования немного упрощает чтение следующих двух выражений:

```
^(?:
  # m/d или mm/dd
  (1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])
|
  # d/m или dd/mm
  (3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])
)
# /yy или /yyyy
/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:
  # mm/dd
  (1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])
|
  # dd/mm
  (3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])
)
# /yyyy
/[0-9]{4}$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

На первый взгляд, проверку чего-то концептуально простого, такого как даты, легко реализовать с помощью регулярного выражения. Но это далеко не так по следующим двум причинам. Во-первых, даты –

вещь настолько обыденная, что люди очень небрежно обращаются с их записью. Так, для кого-то запись 4/1 может обозначать первоапрельский «День смеха». Для кого-то другого это может быть первый рабочий день в году[1], если Новый Год приходится на пятницу. Решения, которые приводятся здесь, обеспечивают совпадение с самыми распространенными форматами представления дат.

Другая проблема заключается в том, что регулярные выражения не работают с числами напрямую. Невозможно определить в регулярном выражении правило, которое, к примеру, гласило бы: «соответствовать числу в диапазоне от 1 до 31». Регулярные выражения работают с символами. Приходится использовать выражение ‹3[01]|[12][0-9]|0?[1-9]›, чтобы описать совпадение с цифрой 3, за которой следует цифра 0 или 1, или совпадение с 1 или 2, за которой следует любая цифра, или совпадение с необязательной цифрой 0, за которой следует цифра в диапазоне от 1 до 9. С помощью символьных классов, таких как ‹[1-9]›, можно определять диапазоны для одной цифры. Это возможно благодаря тому, что символы, обозначающие цифры от 0 до 9, следуют непосредственно друг за другом в таблицах символов ASCII и Юникод. Подробнее о сопоставлении регулярных выражений со всеми типами чисел рассказывается в главе 6.

Вследствие всего вышеизложенного необходимо выбирать между простотой и точностью регулярного выражения. Если заранее известно, что испытуемый текст не содержит ошибочных дат, можно использовать тривиальное регулярное выражение, такое как ‹\d{2}/\d{2}/\d{4}›. Тот факт, что это выражение совпадает с датами, такими как 99/99/9999, не имеет большого значения, если такие даты не могут появиться в испытуемом тексте. Такое выражение можно быстро ввести с клавиатуры, и оно отработает очень быстро.

Первые два решения в этом рецепте тоже относятся к простым и быстрым, и они также совпадают с некорректными датами, такими как 0/0/00 и 31/31/2008. В них используются только литералы символов, соответствующие символам-разделителям в датах, и символьные классы (рецепт 2.3), соответствующие цифрам, а также знак вопроса (рецепт 2.12), чтобы обеспечить необязательность некоторых цифр. Регулярное выражение ‹(?:[0-9]{2})?[0-9]{2}› допускает, что номер года может состоять из двух или четырех цифр. Выражению ‹[0-9]{2}› соответствуют точно две цифры. Выражению ‹(?:[0-9]{2})?› соответствуют ноль или две цифры. Несохраняющая группировка (рецепт 2.9) является обязательной, потому что вопросительный знак должен применяться к символьному классу и квантификатору ‹{2}› одновременно. Выражению ‹[0-9]{2}?› соответствуют точно две цифры, как и выражению ‹[0-9]{2}›. Без группировки вопросительный знак делает предшествующий ему квантификатор минимальным, что не оказывает никакого

[1] 4 января. – *Прим. перев.*

эффекта, потому что квантификатор ‹{2}› не позволяет повторять предшествующий ему элемент больше или меньше, чем два раза.

В решениях с 3 по 6 номер месяца ограничивается диапазоном от 1 до 12, а число – диапазоном от 1 до 31. Для описания совпадения с разными парами цифр при оформлении диапазонов двузначных чисел использовалась конструкция выбора (рецепт 2.8), заключенная в группу. Здесь использовалась сохраняющая группировка, потому что число, месяц и год могут потребоваться позднее.

Последние два решения имеют немного большую сложность, поэтому они представлены в режиме свободного форматирования. Единственное отличие этого режима заключается в более высокой удобочитаемости. Диалект JavaScript не поддерживает режим свободного форматирования. Заключительные решения поддерживают все форматы представления дат, как и первые два примера. Различие между ними состоит в том, что в последних двух выражениях используется дополнительный уровень конструкции выбора, позволяющий пропускать даты 12/31 и 31/12 и запрещающий недопустимые номера месяцев, такие как 31/31.

Варианты

Если вместо того чтобы определять, является ли введенная строка датой, необходимо отыскать даты в тексте документа, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›.

Однако простое удаление их из регулярного выражения не является правильным решением. В этом случае получившееся регулярное выражение будет совпадать, например, с фрагментом 12/12/2001 в строке 9912/12/200199. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что даты не могут быть частью более длинных последовательностей цифр.

Это легко реализуется с помощью пары метасимволов границ слов. В регулярном выражении цифры интерпретируются как символы слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например:

```
\b(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Рецепты 4.5, 4.6 и 4.7.

4.5. Точная проверка дат в традиционных форматах

Задача

Необходимо проверить даты в традиционных форматах mm/dd/yy, mm/dd/yyyy, dd/mm/yy и dd/mm/yyyy. Требуется, чтобы регулярное выражение не пропускало недопустимые даты, такие как 31 февраля.

Решение

C#

Сначала месяц, потом день:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Недопустимая дата
    }
}
```

Сначала день, потом месяц:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<day>[0-3]?[0-9])/(?<month>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Недопустимая дата
    }
```

```
}
```

Perl

Сначала месяц, потом день:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {
    $month = $1;
    $day = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Сначала день, потом месяц:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {
    $day = $1;
    $month = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Проверка исключительно средствами регулярного выражения

Сначала месяц, потом день:

```
^(?:
  # Февраль (29 дней в любом году)
  (?<month>0?2)/(?<day>[12][0-9]|0?[1-9])
```

```
|
  # месяцы продолжительностью 30 дней
  (?<month>0?[469]|11)/(?<day>30|[12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 31 день
  (?<month>0?[13578]|1[02])/(?<day>3[01]|[12][0-9]|0?[1-9])
)
# Год
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET

```
^(?:
  # Февраль (29 дней в любом году)
  (0?2)/([12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 30 дней
  (0?[469]|11)/(30|[12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 31 день
  (0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9])
)
# Год
/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(0?2)/([12][0-9]|0?[1-9])|(0?[469]|11)/(30|[12][0-9]|0?[1-9])|↵
(0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]))/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Сначала день, потом месяц:

```
^(?:
  # Февраль (29 дней в любом году)
(?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
|
  # месяцы продолжительностью 30 дней
  (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[469]|11)
|
  # месяцы продолжительностью 31 день
  (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[13578]|1[02])
)
# Год
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET

```
^(?:
  # Февраль (29 дней в любом году)
  ([12][0-9]|0?[1-9])/(0?2)
|
  # месяцы продолжительностью 30 дней
  (30|[12][0-9]|0?[1-9])/([469]|11)
|
  # месяцы продолжительностью 31 день
  (3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02])
)
# Год
/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([12][0-9]|0?[1-9])/(0?2)|(30|[12][0-9]|0?[1-9])/([469]|11)|↵
(3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02]))/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

По сути, имеется два способа точной проверки дат с помощью регулярных выражений. Первый метод заключается в использовании простого регулярного выражения, которое лишь сохраняет группы чисел, выглядящие как комбинация месяц/день/год, и последующем использовании программного кода, который выполнит проверку этих чисел на корректность. Я использовал первое регулярное выражение из предыдущего рецепта, которое считает допустимыми любые числа в диапазоне от 0 до 39 для обозначения дня и месяца. Это упрощает переход от формата mm/dd/yyyy к формату dd/mm/yyyy простой сменой сохраняющей группы, которая интерпретируется, как номер месяца.

Главное преимущество этого метода заключается в простоте наложения дополнительных ограничений, таких как ограничение дат определенным периодом. Во многих языках программирования имеются средства для работы с датами. В решении для языка C# для проверки допустимости даты и преобразования ее в удобный формат за одно действие используется структура `DateTime`, используемая в платформе .NET.

Другой метод целиком опирается на использование регулярного выражения. Решение становится более или менее простым, если считать каждый год високосным. Здесь можно использовать тот же самый при-

ем, основанный на выборе альтернатив, как это было сделано в заключительных решениях, представленных в предыдущем рецепте.

Проблема решения, опирающегося исключительно на использование регулярного выражения, в том, что для большей точности определения оно больше не сохраняет день и месяц в единственной сохраняющей группе. Теперь имеются три сохраняющих группы для месяца и три сохраняющих группы для дня. Когда регулярное выражение обнаруживает совпадение с датой, только три группы из семи будут хранить некоторые данные. Для февраля месяц и день будут сохраняться в группах 1 и 2. Для месяца продолжительностью 30 дней месяц и день будут сохраняться в группах 3 и 4. Для месяца продолжительностью 31 день месяц и день будут сохраняться в группах 5 и 6. Группа 7 всегда будет хранить год.

Только диалект .NET способен помочь в этой ситуации. В .NET допускается иметь несколько именованных групп (рецепт 2.11) с одним и тем же именем и использовать общее пространство хранения для групп с одинаковыми именами. Если использовать только решение для .NET с именованными сохранениями, можно просто извлекать совпавший текст, используя имена групп «month» и «day», не беспокоясь о том, сколько дней в месяце. Все остальные диалекты, рассматриваемые в этой книге, либо не поддерживают именованное сохранение, либо не допускают наличие в одном регулярном выражении двух групп с одинаковыми именами, либо возвращают текст, сохраненный последней группой с данным именем. Для этих диалектов нумерованное сохранение является единственным выходом.

Решение, опирающееся исключительно на использование регулярного выражения, представляет интерес только в ситуациях, когда регулярное выражение – это единственное, что можно использовать. Например, в приложении, предлагающем единственное поле для ввода регулярного выражения. При программировании многие вещи гораздо проще решить с помощью дополнительного программного кода. Это особенно удобно, если позднее может потребоваться добавить дополнительные проверки дат. Ниже приводится решение, основанное исключительно на применении регулярного выражения, которому соответствуют даты в диапазоне от 2 мая 2007 года до 29 августа 2008 года, в формате d/m/yy или dd/mm/yyyy:

```
# от 2 мая 2007 до 29 августа 2008
^(?:
  # от 2 мая 2007 до 31 декабря 2007
  (?:
    # от 2 мая до 31 мая
    (?<day>3[01]|[12][0-9]|0?[2-9])/(?<month>0?5)/(?<year>2007)
  |
    # от 1 июня до 31 декабря
    (?:
```

```
        # месяцы продолжительностью 30 дней
        (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[69]|11)
      |
        # месяцы продолжительностью 31 день
        (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[78]|1[02])
      )
      /(?<year>2007)
    )
  |
    # от 1 января 2008 до 29 августа 2008
    (?:
      # от 1 августа до 29 августа
      (?<day>[12][0-9]|0?[1-9])/(?<month>0?8)/(?<year>2008)
    |
      # от 1 января до 30 июня
      (?:
        # Февраль
        (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
      |
        # месяцы продолжительностью 30 дней
        (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[46])
      |
        # месяцы продолжительностью 31 день
        (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[1357])
      )
      /(?<year>2008)
    )
  )$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

См. также

Рецепты 4.4, 4.6 и 4.7.

4.6. Проверка времени в традиционных форматах

Задача

Необходимо проверить ввод времени в различных традиционных форматах записи времени, таких как hh:mm и hh:mm:ss в 12-часовом и 24-часовом форматах.

Решение

Часы и минуты в 12-часовом формате:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы и минуты в 24-часовом формате:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды в 12-часовом формате:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды в 24-часовом формате:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Знаки вопроса во всех предыдущих регулярных выражениях делает ведущие нули необязательными. Чтобы сделать их обязательными, достаточно удалить знаки вопроса.

Обсуждение

Реализовать проверку времени определенно проще, чем проверку дат. Продолжительность каждого часа составляет 60 минут, а продолжительность каждой минуты – 60 секунд. Это означает, что нет необходимости строить сложные конструкции выбора в регулярном выражении. Для минут и секунд конструкция выбора вообще не требуется. Выражению `‹[0-5]?[0-9]›` соответствует цифра в диапазоне от 0 до 5, за которой следует цифра в диапазоне от 0 до 9. То есть этому выражению соответствуют числа в диапазоне от 0 до 59. Знак вопроса после первого символьного класса делает его необязательным. Тем самым единственная цифра также рассматривается как допустимое обозначение числа минут или секунд между 0 и 9. Если первые 10 минут или секунд должны записываться как значения от 00 до 09, следует удалить знак вопроса. В рецептах 2.3 и 2.12 подробно описываются символьные классы и квантификаторы, такие как знак вопроса.

Для поиска совпадений со значением часов нам действительно потребуется использовать конструкцию выбора (рецепт 2.8). Вторая цифра

считается допустимой в разных диапазонах, в зависимости от первой цифры. В 12-часовом формате, если первая цифра 0, вторая цифра может быть любой из 10 цифр, но если первая цифра 1, вторая цифра может быть только 0, 1 или 2. В регулярном выражении соответствующая конструкция записана как ‹1[0-2]|0?[1-9]›. В 24-часовом формате, если первая цифра 0 или 1, вторая цифра может быть любой из 10 цифр, но если первая цифра 2, вторая цифра может быть только цифрой в диапазоне от 0 до 3. В регулярном выражении эта проверка выглядит, как ‹2[0-3]|[01]?[0-9]›. Здесь также знак вопроса позволяет записывать первые 10 часов одной цифрой. Если требуется, чтобы час всегда записывался двумя цифрами, знак вопроса следует удалить.

Мы заключили в круглые скобки части регулярного выражения, соответствующие часам, минутам и секундам. Тем самым мы обеспечили простой способ извлечения цифр, обозначающих часы, минуты и секунды без символов двоеточия. В рецепте 2.9 описывается, как с помощью круглых скобок оформляются сохраняющие группы. В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде.

Круглые скобки, заключающие подвыражение, которому соответствует час, объединяют две альтернативные формы записи. Если удалить эти скобки, регулярное выражение будет работать некорректно. Удаление скобок, окружающих минуты и секунды, не окажет никакого эффекта, кроме того, что сделает невозможным извлекать их значения по отдельности.

Варианты

Если вместо того, чтобы определять, является ли введенная строка значением времени, необходимо отыскать значения времени в объемном тексте, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›. Однако простое удаление их из регулярного выражения не является правильным решением. В этом случае получившееся регулярное выражение будет совпадать, например, с фрагментом 12:12 в строке 9912:1299. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что значения времени не могут быть частью более длинных последовательностей цифр.

Это легко реализуется с помощью пары метасимволов границ слов. В регулярном выражении цифры интерпретируются как символы слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например:

```
\b(2[0-3]|[01]?[0-9]):([0-5]?[0-9])\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Границы слов не отклоняют все подряд – они отклоняют только символы, цифры и символы подчеркивания. Только что продемонстрированное регулярное выражение, совпадающее с часами и минутами в 24-часовом формате, обнаружит соответствие 16:08 в испытуемом тексте `The time is 16:08:42 sharp`. Пробел не является символом слова, тогда как `1` является им, поэтому граница слова обнаружит совпадение между ними. Цифра `8` является символом слова, а двоеточие – нет, поэтому метасимвол ‹`\b`› также совпадет с позицией между ними.

Если наряду с символами слова необходимо отклонить и двоеточия, следует использовать проверку соседних символов (рецепт 2.16). Следующее регулярное выражение не совпадет ни с одной частью строки `The time is 16:08:42 sharp`. Оно будет работать только в диалектах, поддерживающих ретроспективную проверку:

```
(?<![:\w])(2[0-3]|[01]?[0-9]):([0-5]?[0-9])(?![:\w])
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

См. также

Рецепты 4.4, 4.5 и 4.7.

4.7. Проверка даты и времени в формате ISO 8601

Задача

Необходимо обеспечить совпадение с датой и/или временем в официальном формате ISO 8601, который является основой для многих стандартизованных форматов представления даты и времени. Например, в языке XML Schema встроенные типы `date`, `time` и `dateTime` основаны на стандарте ISO 8601.

Решение

Следующим выражениям соответствует календарный месяц, например `2008-08`. Дефис обязателен:

```
^([0-9]{4})-(1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-(?<month>1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<year>[0-9]{4})-(?P<month>1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: PCRE, Python

Календарная дата, например 2008-08-30. Дефис необязателен. Это регулярное выражение считает допустимыми формы записи YYYYMMDD и YYYYMM-DD, которые не соответствуют стандарту ISO 8601:

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[0-1]|0[1-9]|[1-2][0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?↵
(?<day>3[0-1]|0[1-9]|[1-2][0-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Календарная дата, например 2008-08-30. Дефис необязателен. Это регулярное выражение использует условную конструкцию для исключения форм записи YYYY-MMDD и YYYYMM-DD. Для первого дефиса добавлена дополнительная сохраняющая группа:

```
^([0-9]{4})(-)?(1[0-2]|0[1-9])(?(2)-)(3[0-1]|0[1-9]|[1-2][0-9])$
```

Параметры: нет
Диалекты: .NET, PCRE, Perl, Python

Календарная дата, например 2008-08-30. Дефис необязателен. Это регулярное выражение использует конструкцию выбора для исключения форм записи YYYY-MMDD и YYYYMM-DD. Для значения месяца определено две сохраняющих группы:

```
^([0-9]{4})(?:(1[0-2]|0[1-9])|-?(1[0-2]|0[1-9])-?)↵
(3[0-1]|0[1-9]|[1-2][0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Неделя года, например 2008-W35. Дефис необязателен:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Неделя и дата, например 2008-W35-6. Дефис необязателен:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])-?([1-7])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])-?(?<day>[1-7])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

День года, например 2008-243. Дефис необязателен:

```
^([0-9]{4})-?(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?↵
(?<day>36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Часы и минуты, например 17:21. Двоеточие необязательно:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Часы, минуты и секунды, например 17:21:59. Двоеточие необязательно:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?↵
(?<second>[0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Указатель часового пояса, например Z, +07 или +07:00. Двоеточие и минуты необязательны:

```
^(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды с указателем часового пояса, например 17:21:59+07:00. Все двоеточия являются необязательными. Минуты в указателе часового пояса также являются необязательными:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])↵
(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?(?<sec>[0-5]?[0-9])↵
(?<timezone>Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Дата с необязательным указателем часового пояса, например 2008-08-30 или 2008-08-30+07:00. Дефисы являются обязательными. Это тип date в языке XML Schema:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])↵
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↵
(?<day>3[0-1]|0[1-9]|[1-2][0-9])↵
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Время с необязательными долями секунды и часовым поясом, например 01:45:36 или 01:45:36.123+07:00. Это тип данных time в языке XML Schema:

```
^(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↵
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[0-1][0-9]):(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])↵
(?<ms>\.[0-9]+)?(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Дата и время с необязательными долями секунды и часовым поясом, например 2008-08-30T01:45:36 или 2008-08-30T01:45:36.123Z. Это тип dateTime в языке XML Schema:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])↵
T(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↵
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↵
(?<day>3[0-1]|0[1-9]|[1-2][0-9])T(?<hour>2[0-3]|[0-1][0-9]):↵
(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])(?<ms>\.[0-9]+)?↵
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Обсуждение

Стандарт ISO 8601 определяет широкий диапазон форматов представления даты и времени. Регулярные выражения, представленные здесь, охватывают наиболее часто используемые форматы, но в большинстве систем, использующих стандарт ISO 8601, применяют только ограниченный набор. Например, в языке XML Schema дефисы и двоеточия в представлениях даты и времени являются обязательными элементами. Чтобы обеспечить обязательность дефисов и двоеточий, достаточно просто удалить знак вопроса, стоящий после них. Чтобы запретить использование символов и двоеточий, нужно удалить эти символы и знаки вопроса, следующие за ними. Будьте внимательны с несохраняющими группами, которые определяются как ‹(?:group)›. Если знак вопроса и символ двоеточия следуют за открывающей круглой скобкой, эти три символа образуют открывающую скобку несохраняющей группы.

Отдельные дефисы и двоеточия, которые не имеют точного соответствия стандарту ISO 8601, в регулярных выражениях объявлены необязательными. Например, фрагмент 1733:26 согласно стандарту ISO 8601 не является допустимой формой записи времени, но он будет принят регулярными выражениями. Требование, чтобы все дефисы и двоеточия либо присутствовали, либо отсутствовали, немного усложняет регулярное выражение. Мы реализовали это требование в качестве примера, но на практике, например в типах данных языка XML Scheme, разделители скорее будут являться либо обязательными, либо запрещенными, чем необязательными.

Все числовые части регулярного выражения мы заключили в круглые скобки, упростив тем самым возможность извлечения года, месяца, дня, часов, минут, секунд и часового пояса. В рецепте 2.9 описывается, как с помощью круглых скобок создаются сохраняющие группы.

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с этими сохраняющими группами, в программном коде.

Кроме того, для большинства регулярных выражений демонстрируется альтернатива, основанная на применении именованного сохранения. Некоторые из приведенных форматов представления даты и времени могут оказаться незнакомыми для вас или ваших коллег. Именованное сохранение делает регулярные выражения проще для понимания. Диалекты .NET, PCRE 7, Perl 5.10 и Ruby 1.9 поддерживают синтаксис `‹(?<name>group)›`. Все версии PCRE и Python, рассматриваемые в этой книге, поддерживают альтернативный синтаксис `‹(?P<name>group)›`, где присутствует дополнительный символ `‹P›`. Дополнительная информация об этом приводится в рецептах 2.11 и 3.9.

Во всех регулярных выражениях введены ограничения на числовые диапазоны. Например, дни месяца ограничены диапазоном чисел от 01 до 31. Выражения никогда не совпадут с 32-м числом или 13-м месяцем. Ни одно из приведенных регулярных выражений не пытается исключить недопустимые комбинации числа и месяца, такие как 31 февраля. В рецепте 4.5 описывается, как решить эту проблему.

Несмотря на то, что некоторые из этих регулярных выражений довольно велики, тем не менее, они достаточно прямолинейны и используют приемы, описанные в рецептах 4.4 и 4.6.

См. также

Рецепты 4.4, 4.5 и 4.6.

4.8. Ограничение возможности ввода алфавитно-цифровыми символами

Задача

В приложении требуется ограничить пользователя возможностью вводить только алфавитно-цифровые символы латинского алфавита.

Решение

Благодаря регулярным выражениям решение получается тривиально простым. Сначала создается символьный класс, включающий требуемый диапазон символов. Затем добавляются квантификатор, повторяющий символьный класс один или более раз, и якорные метасимволы, привязывающие совпадение к началу и к конце строки. Вот и все.

Регулярное выражение

```
^[A-Z0-9]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ruby

```
if subject =~ /^[A-Z0-9]+$/i
    puts "Текст содержит только алфавитно-цифровые символы"
else
    puts "Текст содержит не только алфавитно-цифровые символы"
end
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецептах 3.4 и 3.5.

Обсуждение

Рассмотрим это регулярное выражение, разделив его на четыре части:

```
^          # Проверка совпадения с началом строки.
[A-Z0-9]   # Соответствует символу от "A" до "Z" или от "0" до "9"...
  +        #     от одного до неограниченного числа раз.
$          # Проверка совпадения с концом строки
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Проверки ‹^› и ‹$› в начале и в конце регулярного выражения гарантируют, что будет проверена вся введенная строка. Без них регулярное выражение могло бы совпасть лишь с частью более длинной строки, пропустив недопустимые символы. Квантификатор ‹+› повторяет предшествующий ему элемент один или более раз. Если потребуется обеспечить совпадение регулярного выражения с пустой строкой, можно просто заменить квантификатор ‹+› на ‹*›. Квантификатор ‹*› выполняет ноль или более повторений, делая предшествующий ему элемент необязательным.

Варианты

Ограничение возможности ввода символами ASCII

Следующее регулярное выражение ограничивает возможность ввода 128 символами из 7-битовой таблицы ASCII. В их число входят 33 неотображаемых символа:

```
^[\x00-\x7F]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ограничение возможности ввода неуправляющими символами ASCII и символами разрыва строки

Следующее регулярное выражение можно использовать, чтобы обеспечить возможность ввода только отображаемых и пробельных символов из таблицы ASCII, исключая управляющие символы. Символ перевода строки и возврата каретки (с кодами 0x0A и 0x0D, соответственно) являются наиболее часто используемыми управляющими символами, поэтому они явно были добавлены с помощью метасимволов ‹\n› (перевод строки) и ‹\r› (возврат каретки):

```
^[\n\r\x20-\x7E]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ограничение возможности ввода символами, общими для кодировок ISO-8859-1 и Windows-1252

ISO-8859-1 и Windows-1252 (часто называется ANSI) – это две наиболее широко используемые кодировки восьмибитовых символов, основанные на стандарте Latin-1 (если быть более точными, ISO/IEC 8859-1). Однако они несовместимы между собой по символам с кодами от 0x80 до 0x9F. В кодировке ISO-8859-1 эти коды отведены под управляющие символы, тогда как в кодировке Windows-1252 они используются для определения расширенного диапазона символов и знаков пунктуации.

Эти отличия порой приводят к сложностям в отображении символов, особенно в документах, не объявляющих свою кодировку, или когда получатель текста использует операционную систему, отличную от Windows. Следующее регулярное выражение может использоваться для ограничения ввода набором символов, общих для кодировок ISO-8859-1 и Windows-1252 (включая общие управляющие символы):

```
^[\x00-\x7F\xA0-\xFF]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричная форма записи может осложнить чтение этого регулярного выражения, но данный символьный класс действует точно так же, как и символьный класс ‹[A-Z0-9]›, показанный выше. Ему соответствуют символы в двух диапазонах: \x00-\x7F и \xA0-\xFF.

Ограничение ввода алфавитно-цифровыми символами из любого языка

Это регулярное выражение ограничивает возможность ввода алфавитно-цифровыми символами из любого языка или алфавита. В нем использу-

ется символьный класс, который включает свойства для всех кодовых пунктов Юникода, относящихся к категории букв или цифр:

```
^[\p{L}\p{N}]+$
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

К сожалению, свойства Юникода поддерживаются не всеми диалектами регулярных выражений, рассматриваемыми в этой книге. В частности, это регулярное выражение не будет работать в JavaScript, Python и Ruby 1.8. Кроме того, чтобы это регулярное выражение работало в диалекте PCRE, необходимо, чтобы библиотека PCRE была скомпилирована с поддержкой UTF-8. Свойства Юникода могут использоваться в семействе функций `preg` языка PHP (которые опираются на библиотеку PCRE) при добавлении флага `/u` в конец регулярного выражения.

Следующее регулярное выражение демонстрирует обходное решение для диалекта Python:

```
^[^\W_]+$
```

Параметры: поддержка Юникода
Диалект: Python

Отсутствие поддержки свойств Юникода в диалекте Python компенсируется использованием флага `UNICODE` или `U` при создании регулярного выражения. Этот флаг изменяет действие некоторых метасимволов регулярных выражений, вынуждая их использовать таблицу символов Юникода. Метасимвол `‹\w›` составляет основную часть решения, поскольку ему соответствуют алфавитно-цифровые символы и символ подчеркивания. Используя его инвертированную версию `‹\W›` в инвертированном символьном классе, можно удалить символ подчеркивания из допустимого набора символов. Двойное отрицание, как в этом выражении, иногда бывает весьма полезно в регулярных выражениях, хотя и осложняет их понимание.[1]

См. также

В рецепте 4.9 демонстрируется, как ограничить уже не набор символов, а длину текста.

---

[1] Для пущей забавы (если это попадает под ваше понятие забавы) попробуйте создать три, четыре или даже больше уровней отрицания, добавив негативную проверку соседних символов (рецепт 2.16) и разность символьных классов (раздел «Особенности, характерные для разных диалектов» на стр. 58 в рецепте 2.3).

4.9. Ограничение длины текста

Задача

Необходимо убедиться, что строка имеет длину от 1 до 10 символов и содержит только символы от A до Z.

Решение

Все языки программирования, рассматриваемые в этой книге, предоставляют простой и эффективный способ проверки длины текста. Например, строки в JavaScript имеют свойство `length`, которое хранит целое число, определяющее длину строки. Однако в некоторых ситуациях бывает удобно проверять длину строки в регулярных выражениях, особенно когда длина является одним из нескольких факторов, определяющих совпадение испытуемого текста с требуемым шаблоном. Следующее регулярное выражение гарантирует совпадение со строкой, длина которой составляет от 1 до 10 символов, и эта строка содержит только заглавные буквы от A до Z. Это регулярное выражение можно модифицировать под собственные нужды, установив иные значения минимальной и максимальной длины, или сделать допустимыми символы из других диапазонов, отличных от A–Z.

Регулярное выражение

```
^[A-Z]{1,10}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($ARGV[0] =~ /^[A-Z]{1,10}$/) {
    print "Допустимый ввод\n";
} else {
    print "Недопустимый ввод\n";
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

Ниже приводится это очень прямолинейное регулярное выражение, разложенное на составляющие:

```
^          # Проверка совпадения с началом строки.
[A-Z]      # Соответствует символу в диапазоне от "A" до "Z"...
```

```
  {1,10}  #     от 1 до 10 раз.
$         # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Якорные метасимволы ‹^› и ‹$› гарантируют, что будет проверена вся испытуемая строка, в противном случае регулярное выражение могло бы совпасть лишь с 10 символами более длинной строки. Символьному классу ‹[A-Z]› соответствует любой символ верхнего регистра в диапазоне от A до Z, а интервальный квантификатор ‹{1,10}› повторяет символьный класс от 1 до 10 раз. Благодаря комбинации интервального квантификатора с якорными метасимволами, совпадающими с началом и концом строки, регулярное выражение будет терпеть неудачу, если длина испытуемого текста выйдет за указанные пределы.

Примечательно, что символьный класс ‹[A-Z]› явно допускает наличие в строке только заглавных букв. Если в число допустимых потребуется добавить строчные буквы в диапазоне от a до z, достаточно будет просто заменить этот символьный класс на ‹[A-Za-z]› или применить режим нечувствительности к регистру символов. В рецепте 3.4 демонстрируется, как это сделать.

Начинающие пользователи регулярных выражений часто допускают ошибку, пытаясь сэкономить на вводе с клавиатуры, определяя символьный класс ‹[A-z]›. На первый взгляд такая конструкция выглядит как удачный трюк, позволяющий определить диапазон всех символов верхнего и нижнего регистров. Однако между диапазонами A–Z и a–z в таблице символов ASCII имеется несколько символов пунктуации. Поэтому символьный класс ‹[A-z]› в действительности является эквивалентом символьного класса ‹[A-Z[\]^_`a-z]›.

Варианты

Ограничение длины для произвольного шаблона

Так как такие квантификаторы, как ‹{1,10}›, применяются только к предшествующему элементу, ограничение количества символов, соответствующих шаблону, включающему более одного элемента, требует иного подхода.

Как описывалось в рецепте 2.16, опережающие проверки (и противоположные им ретроспективные проверки) являются проверками специального вида, которые, как и метасимволы ‹^› и ‹$›, совпадают с позициями внутри испытуемой строки и не поглощают никакие символы. Опережающие проверки могут быть позитивными и негативными, то есть они могут проверять наличие или отсутствие совпадения с шаблоном за текущей позицией. Позитивная опережающая проверка, записываемая как ‹(?=…)›, может использоваться в начале шаблона, что-

бы проверить, попадает ли длина строки в заданный диапазон. Остальная часть регулярного выражения может проверять соответствие требуемому шаблону, не беспокоясь о длине текста. Ниже приводится простой пример реализации такого приема:

```
^(?=.{1,10}$).*
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?=[\S\s]{1,10}$)[\S\s]*
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Очень важно, что якорный метасимвол ‹$› находится внутри опережающей проверки, потому что проверка максимальной длины будет работать, только если гарантируется отсутствие дополнительных символов сверх указанного предела. Так как опережающая проверка в начале регулярного выражения уже ограничивает длину испытуемого текста, следующий за ней шаблон может накладывать другие ограничения. В данном случае шаблон ‹.*› (или ‹[\S\s]*›, в версии, добавляющей поддержку JavaScript) используется, чтобы просто обеспечить совпадение с испытуемым текстом без дополнительных ограничений.

Первое регулярное выражение должно использоваться в режиме «точке соответствуют границы строк», чтобы обеспечить корректную работу регулярного выражения, когда испытуемая строка содержит разрывы строк. Как активировать этот режим в разных языках программирования, подробно рассказывается в рецепте 3.4. В JavaScript отсутствует режим «точке соответствуют границы строк», поэтому во втором регулярном выражении используется символьный класс, совпадающий с любым символом. Дополнительная информация приводится в разделе «Любой символ, включая символы конца строки» на стр. 60 в рецепте 2.4.

Ограничение числа непробельных символов

Следующему регулярному выражению соответствует любая строка, содержащая от 10 до 100 непробельных символов:

```
^\s*(?:\S\s*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В диалектах Java, PCRE, Python и Ruby метасимволу ‹\s› соответствуют только пробельные символы ASCII, а метасимволу ‹\S› соответствуют все остальные символы. В Python можно обеспечить совпадение метасимвола ‹\s› всем пробельным символам Юникода, передав флаг UNICODE или U при создании регулярного выражения. В диалектах Java,

PCRE и Ruby 1.9, если необходимо исключить учет пробельных символов Юникода при определении предельного числа символов, можно применить следующую версию регулярного выражения, в которой используются свойства Юникода (описываются в рецепте 2.7):

```
^[\p{Z}\s]*(?:[^\p{Z}\s][\p{Z}\s]*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Чтобы это регулярное выражение работало в диалекте PCRE, библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP включить поддержку UTF-8 можно с помощью модификатора шаблона `/u`.

Это последнее регулярное выражение объединяет свойство Separator Юникода, ‹`\p{Z}`›, с метасимволом ‹`\s`›, совпадающим с любым пробельным символом. Диапазоны символов, соответствующих ‹`\p{Z}`› и ‹`\s`›, не перекрываются полностью. В диапазон ‹`\s`› входят символы с кодами от 0x09 до 0x0D (табуляция, перевод строки, вертикальная табуляция, перевод формата и возврат каретки), которые согласно стандарту Юникода не имеют свойства Separator. Объединение метасимволов ‹`\p{Z}`› и ‹`\s`› в символьный класс гарантирует совпадение с любыми пробельными символами.

В обоих регулярных выражениях квантификатор ‹`{10,100}`› применяется к предшествующей ему несохраняющей группе, а не к единственному элементу. Группе соответствует любой непробельный символ, за которым могут следовать ноль или более пробельных символов. Интервальный квантификатор способен надежно проследить количество совпавших непробельных символов, потому что в каждой итерации возможно совпадение только с одним непробельным символом.

Ограничение числа слов

Следующее регулярное выражение очень похоже на предыдущий пример ограничения числа непробельных символов, за исключением того, что здесь выполняется подсчет слов, а не отдельных непробельных символов. Оно совпадает с текстом, содержащим от 10 до 100 слов, пропуская любые символы, не являющиеся символами слова, включая знаки пунктуации и пробельные символы:

```
^\W*(?:\w+\b\W*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В диалектах Java, JavaScript, PCRE и Ruby метасимвол ‹`\w`›, обозначающий символ слова, в этом регулярном выражении будет совпадать только с символами A–Z, a–z, 0–9 и _ и поэтому он не может использоваться для подсчета слов, содержащих буквы и цифры, не входящие

в набор ASCII. В диалектах .NET и Perl метасимвол ‹\w› опирается на таблицу символов Юникода (так же, как и его инвертированная версия ‹\W›, и граница слова ‹\b›) и будет совпадать со всеми буквами и цифрами из всех алфавитов Юникода. В диалекте Python имеется возможность выбирать, будут эти метасимволы использовать Юникод или нет, в зависимости от того, устанавливается ли флаг UNICODE или U при создании регулярного выражения.

Если необходимо подсчитать количество слов, содержащих буквы и цифры, не входящие в набор ASCII, эту возможность обеспечат следующие регулярные выражения для дополнительных диалектов регулярных выражений.

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+\b[^\p{L}\p{N}_]*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, Perl

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+(?:[^\p{L}\p{N}_]+|$)){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Для выполнения этой работы библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. Включить поддержку UTF-8 в языке PHP можно с помощью модификатора шаблона /u.

Как было отмечено, причина появления различных (но эквивалентных) версий регулярных выражений кроется в различных интерпретациях метасимволов, обозначающих символы слова и границу слова, о чем подробнее рассказывается в разделе «Символы слов» на стр. 70, в рецепте 2.6.

В последних двух регулярных выражениях используются символьные классы, включающие отдельные свойства Юникода для букв и цифр (‹\p{L}› и ‹\p{N}›), и во все классы вручную добавлен символ подчеркивания, чтобы обеспечить их эквивалентность предыдущим регулярным выражениям, опиравшимся на использование метасимволов ‹\w› и ‹\W›.

Каждое повторение несохраняющей группы в первых двух регулярных выражениях из трех совпадает с целым словом, за которым следует ноль или более символов, не являющихся символами слова. Метасимвол ‹\W› (или символьный класс ‹[^\p{L}\p{N}_]›) внутри группы может повторяться нулевое число раз в случае, если строка заканчивается символом слова. Однако, так как в результате этого последовательность символов, не являющихся символами слова, фактически делается необязательной для всего процесса сопоставления, становится необходимой проверка границы слова ‹\b› между ‹\w› и ‹\W› (или между ‹[\p{L}\p{N}_]› и ‹[^\p{L}\p{N}_]›), чтобы гарантировать, что при каждом повторении группа действительно будет совпадать с целым словом.

Без проверки границы слова группа при некотором повторении могла бы совпадать с любой частью слова, а при последующих повторениях – с дополнительными частями.

Третья версия регулярного выражения (где была добавлена поддержка диалектов PCRE и Ruby 1.9) действует несколько иначе. Вместо квантификатора звездочки (ноль или более повторений) в ней используется квантификатор плюс (одно или более повторений) и явно допускается совпадение с нулевым числом символов, только когда процесс сопоставления достигнет конца строки. Это позволяет избежать использования метасимвола границы слова, который был необходим, чтобы обеспечить точность подсчета, так как в диалектах PCRE или Ruby метасимвол `‹\b›` не поддерживает Юникод. В диалекте Java метасимвол `‹\b›` поддерживает Юникод, притом, что метасимвол `‹\w›` в этом же диалекте его не поддерживает.

К сожалению, ни одна из этих возможностей не позволяет корректно обрабатывать слова, в которых используются буквы и цифры, не входящие в набор ASCII, в диалектах JavaScript и Ruby 1.8. Одно из возможных решений заключается в том, чтобы переориентировать регулярное выражение на подсчет пробельных символов, а не последовательностей символов слов, как показано ниже:

```
^\s*(?:\S+(?:\s+|$)){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Perl, PCRE, Python, Ruby

Во многих случаях это выражение будет давать те же результаты, что и предыдущие решения, хотя они не являются точными эквивалентами. Например, одно из различий заключается в том, что составные слова, разделенные дефисом (такие как «far-reaching»), в этом случае будут интерпретироваться не как два слова, а как одно.

См. также

Рецепты 4.8 и 4.10.

4.10. Ограничение числа строк в тексте

Задача

Необходимо убедиться, что испытуемый текст содержит не больше пяти строк независимо от того, сколько символов содержится в этом тексте.

Решение

Символы или последовательности символов, применяемые для разделения строк, могут отличаться в разных операционных системах в зависимости от принятых в них соглашений, используемых приложений,

предпочтений пользователей и так далее. Поэтому для выработки идеального решения необходимо ответить на вопрос о том, какие соглашения об обозначении начала новой строки должны поддерживаться. Решения, следующие ниже, поддерживают соглашения о разрывах строк, принятые в MS-DOC/Windows (‹\r\n›), в устаревшей версии Mac OS (‹\r›) и в UNIX/Linux/OS X (‹\n›).

Регулярное выражение

Следующие три регулярных выражения для разных диалектов имеют два отличия. В первом регулярном выражении вместо несохраняющей группировки, которая оформляется как ‹(?:…)›, используется атомарная группировка, которая оформляется как ‹(?›…)›, потому что атомарная группировка обладает несколько более высокой эффективностью в тех диалектах, которые ее поддерживают. Другое отличие заключается в метасимволе, применяемом для проверки начала и конца строки (метасимволы ‹\A› или ‹^› проверяют совпадение с началом строки, а метасимволы ‹\z›, ‹\Z› или ‹$› – с концом строки). Подробнее эти различия будут обсуждаться ниже в этом рецепте. Все три регулярных выражения в точности соответствуют одним и тем же строкам:

```
\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\A(?:(?:\r\n?|\n)?[^\r\n]*){0,5}\Z
```

Параметры: нет
Диалект: Python

```
^(?:(?:\r\n?|\n)?[^\r\n]*){0,5}$
```

Параметры: нет
Диалект: JavaScript

PHP (PCRE)

```
if (preg_match('/\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z/', $_POST['subject'])) {
    print 'Испытуемый текст содержит не более пяти строк';
} else {
    print ' Испытуемый текст содержит более пяти строк ';
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

Все регулярные выражения, продемонстрированные к настоящему моменту в этом рецепте, используют группировку для поиска совпадений с последовательностями, обозначающими разрывы строк, оформленными в стиле MS-DOS/Windows, устаревшей версии Mac OS или UNIX/Linux/OS X, за которыми следует произвольное число любых других символов. Группировка повторяется от нуля до пяти раз, поскольку согласно условиям требуется обеспечить совпадение с текстом, содержащим не более пяти строк.

Ниже приводится версия регулярного выражения для JavaScript, разбитая на отдельные части. Мы использовали версию для JavaScript потому, что элементы этого выражения наверняка будут знакомы широкому кругу читателей. Отличия в версиях для других диалектов мы объясним ниже:

```
^          # Проверить совпадение с началом строки.
(?:        # Группировка, но не сохраняющая...
  (?:      #   Группировка, но не сохраняющая...
    \r     #     Соответствует символу возврата каретки
           #     (CR, ASCII-код 0x0D).
    \n     #     Соответствует символу перевода строки
           #     (LF, ASCII-код 0x0A)...
      ?    #       ноль или один раз.
  |        #     или...
    \n     #       Соответствует символу перевода строки.
  )        #   Конец сохраняющей группы.
    ?      #     Повторить предыдущую группу ноль или один раз.
  [^\r\n]  #   Соответствует любому символу, кроме CR и LF...
    *      #     от нуля до неограниченного числа раз.
)          # Конец несохраняющей группы.
  {0,5}    #   Повторить предыдущую группу от нуля до пяти раз.
$          # Проверить совпадение с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Первый метасимвол ‹^› совпадает с позицией начала текста. Это позволит гарантировать, что весь текст будет содержать не более пяти строк, потому что, если не привязать регулярное выражение к началу текста, оно может совпасть с любыми пятью строками внутри более объемного текста.

Следующая несохраняющая группа включает в себя комбинацию из последовательности конца строки и любых других символов, не являющихся символами конца строки. Сразу за ней следует квантификатор, позволяющий этой группе повторяться от нуля до пяти раз (нулевое число повторений соответствует полностью пустой строке). Необя-

зательная подгруппа внутри внешней группы совпадает с последовательностью конца строки. За ней следует символьный класс, который совпадает с произвольным числом символов, не являющихся символами конца строки.

Обратите внимание на порядок следования элементов во внешней группе (сначала идет последовательность конца строки, а затем последовательность символов, не являющихся символами конца строки). Если элементы поставить в обратном порядке, то есть чтобы группа выглядела как ‹(?:[^\r\n]*(?:\r\n?|\n)?)›, для пятого повторения потребовалось бы наличие последовательности символов конца строки в конце текста. Фактически, такое выражение позволяло бы наличие в тексте шестой пустой строки.

Подгруппа определяет три последовательности конца строки:

- Символ возврата каретки, за которым следует символ перевода строки (‹\r\n›, последовательность конца строки, обычная для MS-DOS/Windows)
- Отдельный символ возврата каретки (‹\r›, символ конца строки в устаревшей версии Mac OS)
- Отдельный символ перевода строки (‹\n›, символ конца строки, обычный для UNIX/Linux/OS X)

Теперь перейдем к различиям в разных диалектах.

Первая версия регулярного выражения (совместимая со всеми диалектами, за исключением Python и JavaScript) использует не простую несохраняющую группировку, а атомарную. В некоторых случаях использование атомарной группировки может оказывать более существенное влияние, но в данной ситуации она просто позволяет механизму регулярных выражений избежать ненужных возвратов, необходимость в которых может возникнуть, если попытка совпадения начнет терпеть неудачу (подробнее об атомарной группировке рассказывается в рецепте 2.15).

Еще одно отличие между версиями для разных диалектов – это метасимволы, используемые для проверки совпадения с началом и концом текста. Непригодность метасимволов ‹^› и ‹$› для этих целей уже была продемонстрирована выше. Несмотря на то, что эти якорные метасимволы поддерживаются всеми диалектами регулярных выражений, рассматриваемыми здесь, тем не менее, альтернативные решения в этом разделе используют вместо них метасимволы ‹\A›, ‹\Z› и ‹\z›. Если говорить коротко, то это объясняется тем, что значения этих метасимволов несколько отличаются в разных диалектах. Для подробного объяснения придется погрузиться в историю развития регулярных выражений....

Когда в языке Perl выполнялось чтение строк из файла, получающаяся строка заканчивалась символом конца строки. Вследствие этого в Perl

было «расширено» традиционное значение метасимвола ‹$›, которое затем было скопировано большинством других диалектов регулярных выражений. В дополнение к совпадению с абсолютным концом всего текста метасимвол ‹$› в языке Perl совпадает также с позицией непосредственно перед символом конца строки. Кроме того, в язык Perl были введены еще две проверки, совпадающие с концом текста: ‹\Z› и ‹\z›. Якорный метасимвол ‹\Z› в языке Perl имеет почти то же значение, что и метасимвол ‹$›, за исключением того, что на него не оказывает влияния режим, позволяющий метасимволам ‹^› и ‹$› совпадать с символами конца строки. Метасимвол ‹\z› всегда совпадает только с абсолютным концом текста, без всяких исключений. Поскольку в этом рецепте мы имеем дело с символами конца строки, с помощью которых производится подсчет строк в тексте, то чтобы убедиться в отсутствии шестой строки, здесь используется проверка ‹\z›, в диалектах, поддерживающих ее.

Большинство других диалектов регулярных выражений скопировало из языка Perl якорные метасимволы, обозначающие конец строки/текста. Диалекты .NET, Java, PCRE и Ruby поддерживают оба метасимвола, ‹\Z› и ‹\z›, которые в этих диалектах имеют то же значение, что и в языке Perl. В Python имеется только метасимвол ‹\Z› (в верхнем регистре), но он имеет немного сбивающее с толку значение, совпадая только с абсолютным концом текста, как метасимвол ‹\z› (в нижнем регистре) в языке Perl. В диалекте JavaScript отсутствуют якорные метасимволы «z», но, в отличие от всех остальных диалектов, рассматриваемых в этой книге, якорный метасимвол ‹$› в этом диалекте совпадает только с абсолютным концом текста (когда отключен режим, допускающий совпадение метасимволов ‹^› и ‹$› с границами строк).

В случае с метасимволом ‹\A› ситуация немного лучше. Он всегда совпадает только с началом текста и одинаково интерпретируется во всех диалектах, рассматриваемых в этой книге, за исключением JavaScript (который не поддерживает его).

Несмотря на существование досадных разногласий между диалектами, одно из преимуществ регулярных выражений, которые приводятся в этой книге, состоит в том, что вам вообще не придется беспокоиться по поводу этих различий. Кровавые подробности, подобные только что описанным, мы включаем только для тех, кто желает получить более глубокие знания.

Варианты

Работа с необычными разделителями строк

Регулярные выражения, продемонстрированные выше, ограничиваются поддержкой последовательностей конца строки, обычных для MS-DOS/Windows, устаревшей версии Mac OS и UNIX/Linux/OS X. Одна-

ко существуют некоторые другие редко встречающиеся завершающие пробельные символы, с которыми можно столкнуться. Следующие регулярные выражения учитывают возможность появления этих дополнительных символов, допуская возможность появления в тексте не более пяти строк.

```
\A(?>\R?\V*){0,5}\z
```

Параметры: нет
Диалекты: PCRE 7 (with the PCRE_BSR_UNICODE option), Perl 5.10

```
\A(?>(?>\r\n?|[\n-\f\x85\x{2028}\x{2029}])?↵
[^\n-\r\x85\x{2028}\x{2029}]*){0,5}\z
```

Параметры: нет
Диалекты: PCRE, Perl

```
\A(?>(?>\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\z
```

Параметры: нет
Диалекты: .NET, Java, Ruby

```
\A(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\Z
```

Параметры: нет
Диалект: Python

```
^(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}$
```

Параметры: нет
Диалект: JavaScript

Все эти регулярные выражения обрабатывают разделители, перечисленные в табл. 4.1 вместе с их кодовыми пунктами и именами в Юникоде.

Таблица 4.1. Разделители строк

Последовательность в Юникоде	Эквивалент в регулярных выражениях	Название	Когда используется
U+000D U+000A	`\r\n`	Возврат каретки и перевод строки (CRLF)	Текстовые файлы в Windows и MS-DOS
U+000A	`\n`	Перевод строки (LF)	Текстовые файлы в UNIX, Linux и OS X
U+000B	`\v`	Вертикальная табуляция (VT)	(Редко)

Последовательность в Юникоде	Эквивалент в регулярных выражениях	Название	Когда используется
U+000C	‹\f›	Перевод формата (FF)	(Редко)
U+000D	‹\r›	Возврат каретки (CR)	Текстовые файлы в Mac OS
U+0085	‹\x85›	Следующая строка (NEL)	Текстовые файлы в суперкомпьютерах IBM (Редко)
U+2028	‹\x2028› или ‹\x{2028}›	Разделитель строк	(Редко)
U+2029	‹\x2029› или ‹\x{2029}›	Разделитель параграфов	(Редко)

См. также

Рецепт 4.9.

4.11. Проверка утвердительных ответов

Задача

Необходимо проверить параметры настройки или ответ, введенный в командной строке, на совпадение с положительным утверждением. Требуется обеспечить достаточную гибкость в принятии ответов, чтобы такие ответы, как `true`, `t`, `yes`, `y`, `okay`, `ok` и `1`, принимались в любых комбинациях символов верхнего и нижнего регистров.

Решение

Использовать регулярное выражение, объединяющее в себе все допустимые формы и позволяющее выполнить проверку с помощью одного простого критерия.

Регулярное выражение

```
^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
var yes = /^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$/i;
if (yes.test(subject)) {
    alert("Yes");
} else {
    alert("No");
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецептах 3.4 и 3.5.

Обсуждение

Ниже приводится регулярное выражение, разложенное на отдельные составляющие. Комбинации элементов, которые проще читать вместе, приводятся в одной строке:

```
^            # Проверка совпадения с началом строки.
(?:          # Группировка, но несохраняющая...
  1          #   Проверка соответствия литералу "1".
 |           #  или...
  t(?:rue)?  #   Проверка соответствия с "t",
             #   за которым может следовать "rue".
 |           #  или...
  y(?:es)?   #   Проверка соответствия с "y",
             #   за которым может следовать "es".
 |           #  или...
  ok(?:ay)?  #   Проверка соответствия с "ok",
             #   за которым может следовать "ay".
)            # Конец несохраняющей группы.
$            # Проверка совпадения с концом строки.
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Регулярное выражение фактически проверяет совпадение с одним из семи литералов без учета регистра символов. Его можно записать разными способами.

Например, выражение ‹^(?:[1ty]|true|yes|ok(?:ay)?)$› ничуть не хуже предыдущего. Простая проверка совпадения с одной из семи альтернатив, например ‹^(?:1|t|true|y|yes|ok|okay)$›, также прекрасно справляется с поставленной задачей, хотя, с точки зрения производительности, вообще лучше минимизировать количество альтернатив, перечис-

ляемых с помощью оператора выбора ‹|› в символьных классах и необязательных элементах (к которым применяется квантификатор ‹?›). В данном случае падение производительности наверняка не будет превышать нескольких микросекунд, но для пользы дела всегда следует помнить о проблемах, связанных с производительностью регулярных выражений. Иногда различия между этими способами реализации могут вызывать неподдельное удивление.

Во всех этих примерах потенциальные совпадения заключены в несохраняющую группу, чтобы ограничить область действия операторов выбора. Если убрать группировку и использовать, например, такое выражение, как ‹^true|yes$›, механизм регулярных выражений будет искать «начало строки, за которым следует фрагмент «true» или «yes», за которым следует конец строки». Выражение ‹^(?:true|yes)$› предписывает механизму регулярных выражений отыскать начало строки, потом отыскать «true» или «yes», а потом – конец строки.

См. также

Рецепты 5.2 и 5.3.

4.12. Проверка номеров социального страхования

Задача

Необходимо проверить, является ли некоторый текст допустимым номером социального страхования.

Решение

Если требуется просто убедиться, что текст следует формату записи номеров социального страхования и отвергнуть очевидно недопустимые значения, решить эту задачу легко можно с помощью приведенного ниже регулярного выражения. Если требуется более строгое решение, которое выполняет сверку с данными Управления социального обеспечения, чтобы определить, принадлежит ли этот номер живущему человеку, обращайтесь к разделу «См. также» этого рецепта.

Регулярное выражение

```
^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↵
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Python

```
if re.match(r"^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↵
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$", sys.argv[1]):
    print "Корректный номер социального страхования"
else:
    print "Некорректный номер социального страхования"
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецептах 3.5.

Обсуждение

Номера социального страхования в Соединенных Штатах состоят из девяти цифр и имеют формат AAA-GG-SSSS:

- Первые три цифры определяют географический регион и называются *номером зоны*. Номер зоны не может иметь значение 000 или 666 и к моменту написания этих строк недопустимыми считались все номера социального страхования, содержащие номера зон со значениями выше 772.
- Четвертая и пятая цифры называются *номером группы*, который может изменяться в диапазоне от 01 до 99.
- Последние четыре цифры составляют *личный номер*, который может изменяться в диапазоне от 0001 до 9999.

Решение для этого рецепта соблюдает все ограничения, перечисленные выше. Ниже снова приводится регулярное выражение, на этот раз разложенное на отдельные составляющие:

```
^           # Проверка совпадения с началом строки.
(?!000|666) # Гарантировать отсутствие совпадения с "000" или "666" здесь.
(?:         # Группировка, но несохраняющая...
  [0-6]     #   Соответствует символу в диапазоне от "0" до "6".
  [0-9]{2}  #   Соответствует цифре точно два раза.
 |          #  или...
  7         #   Соответствует литералу "7".
  (?:       #   Группировка, но несохраняющая...
    [0-6]   #     Соответствует символу в диапазоне от "0" до "6".
    [0-9]   #     Соответствует цифре.
   |        #    или...
    7       #     Соответствует литералу "7".
    [0-2]   #     Соответствует символу в диапазоне от "0" до "2".
  )         #   Конец несохраняющей группы.
)           # Конец несохраняющей группы.
-           # Соответствует литералу "-".
(?!00)      # Гарантировать отсутствие совпадения с "00" здесь.
```

```
[0-9]{2}    # Соответствует цифре точно два раза.
-           # Соответствует литералу "-".
(?!0000)    # Гарантировать отсутствие совпадения с "0000" здесь.
[0-9]{4}    # Соответствует цифре точно четыре раза.
$           # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Помимо метасимволов ‹^› и ‹$›, выполняющих проверку совпадения с началом и концом строки, это регулярное выражение можно разбить на три группы цифр, разделенных дефисом. Первая группа самая сложная. Вторая и третья группы просто совпадают с двух- и четырехзначными числами, соответственно, но используют негативную опережающую проверку, чтобы обеспечить невозможность совпадения со значениями, состоящими из одних нулей.

Первая группа цифр является самой сложной и самой неудобочитаемой по сравнению с другими, потому что она описывает совпадение с диапазоном чисел. Вначале она использует негативную опережающую проверку ‹(?!000|666)›, чтобы исключить значения «000» и «666». Затем решается задача устранения любых чисел выше 772.

Так как регулярному выражению приходится иметь дело с текстом, а не с числами, мы вынуждены проверять соответствие диапазону чисел цифра за цифрой. Во-первых, известно, что допускается совпадение с любыми трехзначными числами, начинающимися с цифр от 0 до 6, благодаря тому, что предыдущая негативная опережающая проверка уже исключила недопустимые числа 000 и 666. Эта первая часть задачи легко решается с помощью нескольких символьных классов и квантификатора: ‹[0-6][0-9]{2}›. Поскольку диапазон чисел, начинающихся с 7, ограничен, шаблон, который только что был построен, мы поместили в группу ‹(?:[0-6][0-9]{2}|7)›, чтобы ограничить область действия оператора выбора.

Числа, начинающиеся с 7, считаются допустимыми, только если они находятся в диапазоне от 700 до 772, поэтому следующий шаг состоит в том, чтобы выделить все числа, второй знак в которых начинается с цифры 7. Если вторая цифра попадает в диапазон от 0 до 6, тогда третья цифра может быть любой. Если вторая цифра 7, третья цифра будет считаться допустимой, только если она находится в диапазоне от 0 до 2. Объединив эти правила для чисел, начинающихся с цифры 7, получаем выражение ‹7(?:[0-6][0-9]|7[0-2])›, которое совпадает с цифрой 7, за которой следует одна из двух возможных альтернатив для второй и третьей цифры.

Наконец, вставив его во внешнюю группу, соответствующую первой группе цифр, получаем выражение ‹(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))›. То есть только что мы благополучно создали регулярное выражение, которому соответствуют трехзначные числа в диапазоне от 000 до 772.

Варианты

Поиск номеров социального страхования в документах

Если требуется отыскать номера социального страхования в большом документе или в длинной строке ввода, следует заменить якорные метасимволы ‹^› и ‹$› метасимволами границы слова. Механизмы регулярных выражений интерпретируют все алфавитно-цифровые символы и символ подчеркивания как символы слова.

```
\b(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↵
(?!00)[0-9]{2}-(?!0000)[0-9]{4}\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

На сайте Управления социального обеспечения *http://www.socialsecurity.gov* можно найти ответы на наиболее типичные вопросы, а также самые свежие списки присвоенных номеров зон и групп.

Служба проверки номеров социального страхования (Social Security Number Verification Service, SSNVS), по адресу: *http://www.socialsecurity.gov/employer/ssnv.htm*, предлагает два способа убедиться, что имя и номер социального страхования соответствуют записи в Управлении социального обеспечения.

Более полное обсуждение сопоставления с числовыми диапазонами, включая примеры сопоставления с диапазонами, включающими переменное число цифр, можно найти в рецепте 6.5.

4.13. Проверка номеров ISBN

Необходимо проверить корректность международного стандартного номера книги (International Standard Book Number, ISBN), который может быть записан в старом (ISBN-10) или в текущем (ISBN-13) формате. Требуется обеспечить возможность разделения идентификатора ISBN и других частей номера символом дефиса или пробела. Номера `ISBN 978-0-596-52068-7`, `ISBN-13: 978-0-596-52068-7`, `978 0 596 52068 7`, `9780596520687`, `ISBN-10 0-596-52068-9` и `0-596-52068-9` являются примерами допустимых номеров ISBN.

Решение

Невозможно выполнить полную проверку номера ISBN исключительно средствами регулярного выражения, потому что последняя цифра вычисляется с применением алгоритма получения контрольной суммы.

Регулярные выражения, которые приводятся в этом разделе, проверяют формат записи номера ISBN, тогда как последующие примеры программного кода включают проверку последней цифры.

Регулярные выражения

ISBN-10:

```
^(?:ISBN(?:-10)?:?•)?(?=[-0-9X•]{13}$|[0-9X]{10}$)[0-9]{1,5}[-•]?↵
(?:[0-9]+[-•]?){2}[0-9X]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-13:

```
^(?:ISBN(?:-13)?:?•)?(?=[-0-9•]{17}$|[0-9]{13}$)97[89][-•]?[0-9]{1,5}↵
[-•]?(?:[0-9]+[-•]?){2}[0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-10 или ISBN-13:

```
^(?:ISBN(?:-1[03])?:?•)?(?=[-0-9•]{17}$|[-0-9X•]{13}$|[0-9X]{10}$)↵
(?:97[89][-•]?)?[0-9]{1,5}[-•]?(?:[0-9]+[-•]?){2}[0-9X]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
// `регулярное выражение` проверяет соответствие формату ISBN-10 или ISBN-13
var regex = /^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|[-0-9X ]{13}$|↵
[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$/;

if (regex.test(subject)) {
    // Удалить цифры, не имеющие отношение к ISBN, и разбить номер на части
    // представив их в виде массива
    var chars = subject.replace(/[^0-9X]/g, “”).split(“”);
    // Удалить последнюю цифру ISBN из массива `chars`,
    // и присвоить ее переменной `last`
    var last = chars.pop();
    var sum = 0;
    var digit = 10;
    var check;

    if (chars.length == 9) {
        // Вычислить контрольную цифру ISBN-10
        for (var i = 0; i < chars.length; i++) {
            sum += digit * parseInt(chars[i], 10);
```

```
            digit -= 1;
        }
        check = 11 - (sum % 11);
        if (check == 10) {
            check = “X”;
        } else if (check == 11) {
            check = “0”;
        }
    } else {
        // Вычислить контрольную цифру ISBN-13
        for (var i = 0; i < chars.length; i++) {
            sum += (i % 2 * 2 + 1) * parseInt(chars[i], 10);
        }
        check = 10 - (sum % 10);
        if (check == 10) {
            check = “0”;
        }
    }
    if (check == last) {
        alert(“Корректный номер ISBN”);
    } else {
        alert(“Некорректная контрольная цифра номера ISBN”);
    }
} else {
    alert(“Некорректный номер ISBN”);
}
```

Python

```
import re
import sys

# регулярное выражение проверяет соответствие формату ISBN-10 или ISBN-13
regex = re.compile(“^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|↵
[-0-9X ]{13}$|[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?↵
(?:[0-9]+[- ]?){2}[0-9X]$”)

subject = sys.argv[1]

if regex.search(subject):
    # Удалить цифры, не имеющие отношение к ISBN, и разбить номер на части,
    # представив их в виде массива
    chars = re.sub(“[^0-9X]”, “”, subject).split(“”)
    # Удалить последнюю цифру ISBN из массива `chars`
    # и присвоить ее переменной `last`
    last = chars.pop()
```

```
        if len(chars) == 9:
            # Вычислить контрольную цифру ISBN-10
            val = sum((x + 2) * int(y) for x,y in enumerate(reversed(chars)))
        check = 11 - (val % 11)
        if check == 10:
            check = "X"
        elif check == 11:
            check = "0"
        else:
            # Вычислить контрольную цифру ISBN-13
            val = sum((x % 2 * 2 + 1) * int(y) for x,y in enumerate(chars))
            check = 10 - (val % 10)
            if check == 10:
                check = "0"

        if (str(check) == last):
            print "Корректный номер ISBN"
        else:
            print "Некорректная контрольная цифра номера ISBN"
    else:
        print "Некорректный номер ISBN"
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

ISBN – это уникальный идентификатор продаваемых книг и подобных им продуктов. 10-значный формат ISBN был опубликован как международный стандарт ISO 2108 в 1970 году. Все номера ISBN, которые присваиваются начиная с 1 января 2007 года, состоят из 13 цифр.

Номера в форматах ISBN-10 и ISBN-13 делятся на четыре и на пять элементов, соответственно. Три элемента имеют переменную длину, а остальные один или два элемента – фиксированную. Обычно все части отделяются друг от друга дефисами или пробелами. Ниже приводится краткое описание каждого элемента:

- 13-значные номера ISBN начинаются с префикса 978 или 979.
- *Идентификатор группы* определяет группу стран, говорящих на одном языке. Этот идентификатор может иметь длину от одной до пяти цифр.
- *Идентификатор издательства* имеет переменную длину и присваивается национальным агентством ISBN.

- *Идентификатор книги* также имеет переменную длину и выбирается издателем.
- Заключительный символ называется *контрольной цифрой* и вычисляется с применением алгоритма получения контрольной суммы. В формате ISBN-10 контрольная цифра может быть числом от 0 до 9 или символом X (число 10 в римской системе записи чисел), тогда как в формате ISBN-13 контрольная цифра может быть только числом в диапазоне от 0 до 9. Так как для разных форматов ISBN используются разные алгоритмы вычисления контрольной цифры, допустимые символы в них отличаются.

Ниже приводится регулярное выражение из решения «ISBN-10 или ISBN-13», разложенное на отдельные составляющие. Так как это регулярное выражение записано в режиме свободного форматирования, литералы пробелов в нем экранированы символами обратного слэша. В языке Java требуется, чтобы в режиме свободного форматирования пробелы экранировались даже внутри символьных классов:

```
^                 # Проверка совпадения с началом строки.
(?:               # Группировка, но несохраняющая...
  ISBN            #   Соответствует фрагменту “ISBN”.
  (?:-1[03])?     #   Необязательное совпадение с фрагментом “-10” или “-13”.
  :?              #   Необязательное совпадение с литералом “:”.
  \               #   Соответствует символу пробела (экранирован).
)?                # Повторить группу ноль или один раз.
(?=               # Опережающая проверка возможности следующих совпадений...
  [-0-9\ ]{17}$   #   Соответствует 17 дефисам, цифрам и пробелам,
 |                #  за которыми следует конец строки. Или...
  [-0-9X\ ]{13}$  #   Соответствует 13 дефисам, цифрам, “X” и пробелам,
 |                #  за которыми следует конец строки. Или...
  [0-9X]{10}$     #   Соответствует цифре и “X”, с последующим концом строки.
)                 # Конец позитивной опережающей проверки.
(?:               # Группировка, но несохраняющая...
  97[89]          #   Соответствует фрагменту “978” или “979”.
  [-\ ]?          #   Необязательное совпадение с дефисом или пробелом.
)?                # Повторить группу ноль или один раз.
[0-9]{1,5}        # Соответствует от одной до пяти цифр.
[-\ ]?            # Необязательное совпадение с дефисом или пробелом.
(?:               # Группировка, но несохраняющая...
  [0-9]+          #   Соответствует один или более раз.
  [-\ ]?          #   Необязательное совпадение с дефисом или пробелом.
){2}              # Повторить группу точно два раза.
[0-9X]            # Соответствует цифре или “X”.
$                 # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Первое подвыражение ‹(?:ISBN(?:-1[03])?:?•)?› содержит три необязательных элемента, что позволяет ему совпадать с любым из семи фрагментов, следующих ниже (в конце всех фрагментов, за исключением варианта с пустой строкой, имеется пробел):

- ISBN•
- ISBN-10•
- ISBN-13•
- ISBN:•
- ISBN-10:•
- ISBN-13:•
- *Пустая строка (префикс ISBN отсутствует)*

Далее следует позитивная опережающая проверка ‹(?=[-0-9•]{17}$|[-0-9X•]{13}$|[0-9X]{10}$)›, гарантирующая соблюдение одного из трех ограничений (разделенных оператором выбора ‹|›), определяющих длину и допустимый набор символов для остальной части совпадения. Все три ограничения (показаны ниже) оканчиваются якорным метасимволом ‹$›, гарантирующим отсутствие последующего текста, не укладывающегося ни в один из шаблонов:

‹[-0-9•]{17}$›

Формат ISBN-13 с четырьмя разделителями (всего 17 символов)

‹[-0-9X•]{13}$›

Формат ISBN-13 без разделителей или формат ISBN-10 с тремя разделителями (всего 13 символов)

‹[0-9X]{10}$›

Формат ISBN-10 без разделителей (всего 10 символов)

После того как позитивная опережающая проверка проверит длину и набор символов, можно произвести сопоставление отдельных элементов номера ISBN, не беспокоясь об их суммарной длине. Подвыражению ‹(?:97[89][-•]?)?› соответствует префикс «978» или «979», обязательный в формате ISBN-13. Совпадение с несохраняющей группой сделано необязательным, потому что она не обнаружит соответствия в испытуемой строке, имеющей формат ISBN-10. Подвыражению ‹[0-9]{1,5}[-•]?› соответствуют от одной до пяти цифр идентификатора группы и необязательный разделитель, возможно следующий за ними. Подвыражению ‹(?:[0-9]+[-•]?){2}› соответствуют идентификаторы издательства и книги, имеющие переменную длину, и их необязательные разделители. Наконец, подвыражению ‹[0-9X]$› соответствует контрольная цифра в конце строки.

Несмотря на то, что регулярное выражение в состоянии убедиться, что в качестве последней цифры используется корректный символ (циф-

ра или символ X), оно не сможет определить, представляет ли эта цифра корректную контрольную сумму ISBN. Для обеспечения гарантии, что цифры номера ISBN были введены без ошибок и не являются случайным набором цифр, используется один из двух алгоритмов вычисления контрольной суммы (в зависимости от того, в каком формате записан номер – ISBN-10 или ISBN-13). Реализации обоих алгоритмов показаны в примерах программного кода на языках JavaScript и Python. В следующих разделах описываются правила определения контрольных сумм, чтобы вы могли реализовать их в других языках программирования.

Контрольная сумма ISBN-10

Контрольная сумма для номеров в формате ISBN-10 может принимать значения в диапазоне от 0 до 10 (значение 10 записывается Римской цифрой X). Она вычисляется следующим образом:

1. Каждая из первых 9 цифр умножается на число, убывающее в порядке от 10 до 2, а результаты умножения суммируются.
2. Сумма делится на 11.
3. Остаток деления (не частное) вычитается из числа 11.
4. Если результат равен 11, в качестве контрольной суммы используется число 0, для представления результата, равного 10, используется символ X.

Ниже приводится пример вычисления контрольной цифры для номера `0-596-52068-?`, в формате ISBN-10:

```
Шаг 1:
sum = 10×0 + 9×5 + 8×9 + 7×6 + 6×5 + 5×2 + 4×0 + 3×6 + 2×8
    =    0 +  45 +  72 +  42 +  30 +  10 +   0 +  18 +  16
    = 233
Шаг 2:
    233 ÷ 11 = 21, остаток 2
Шаг 3:
    11 - 2 = 9
Шаг 4:
    9 [подстановка иного значения не требуется]
```

Была получена контрольная цифра 9, поэтому полный номер имеет вид: `ISBN 0-596-52068-9`.

Контрольная сумма ISBN-13

Контрольная сумма для номеров в формате ISBN-13 может принимать значения в диапазоне от 0 до 9 и вычисляется похожим способом:

1. По мере перемещения по номеру вправо каждая из первых 12 цифр умножается на 1 или на 3, а результаты умножения суммируются.

2. Сумма делится на 10.
3. Остаток деления (не частное) вычитается из числа 10.
4. Если результат равен 10, в качестве контрольной суммы используется число 0.

Ниже приводится пример вычисления контрольной цифры для номера `978-0-596-52068-?`, в формате ISBN-13:

```
Шаг 1:
sum = 1×9 + 3×7 + 1×8 + 3×0 + 1×5 + 3×9 + 1×6 + 3×5 + 1×2 + 3×0 + 1×6 + 3×8
    =   9 +  21 +   8 +   0 +   5 +  27 +   6 +  15 +   2 +   0 +   6 +  24
    = 123
Шаг 2:
    123 ÷ 10 = 12, остаток 3
Шаг 3:
    10 - 3 = 7
Шаг 4:
    7 [подстановка иного значения не требуется]
```

Была получена контрольная цифра 7, поэтому полный номер имеет вид: `ISBN 978-0-596-52068-7`.

Варианты

Поиск номеров ISBN в документах

В этой версии регулярного выражения «ISBN-10 или ISBN-13» якорные метасимволы были заменены метасимволом границы слова, чтобы его можно было использовать для поиска отдельных номеров ISBN в объемном тексте. Кроме того, в этой версии идентификатор «ISBN» сделан обязательным по двум причинам. Во-первых, обязательное наличие идентификатора позволит избежать ложных совпадений (без этого требования регулярное выражение может совпадать с 10- и 13-значными числами) и во-вторых, идентификатор ISBN официально является обязательным при выводе на печать:

```
\bISBN(?:-1[03])?:?•(?=[-0-9•]{17}$|[-0-9X•]{13}$|[0-9X]{10}$)↵
(?:97[89][-•]?)?[0-9]{1,5}[-•]?(?:[0-9]+[-•]?){2}[0-9X]\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пропуск некорректных идентификаторов ISBN

Предыдущие регулярные выражения допускают совпадение с номером ISBN-10, которому предшествует идентификатор «ISBN-13», и наоборот. В следующем регулярном выражении используется условная конструкция (рецепт 2.17), чтобы убедиться, что за идентификаторами «ISBN-10» и «ISBN-13» следует номер ISBN соответствующего типа. Оно допускает совпадение с любым из форматов ISBN-10 и ISBN-13, если тип

не указан явно. В большинстве случаев это регулярное выражение будет чрезмерно избыточным, потому что те же самые результаты можно получить, последовательно используя более простые регулярные выражения, отдельные для ISBN-10 и ISBN-13, продемонстрированные выше. Оно представлено здесь исключительно для демонстрации возможностей регулярных выражений:

```
^
(?:ISBN(-1(?:(0)|3))?:?\ )?
(?(1)
  (?(2)
    (?=[-0-9X ]{13}$|[0-9X]{10}$)
    [0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
   |
    (?=[-0-9 ]{17}$|[0-9]{13}$)
    97[89][- ]?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9]$
  )
 |
  (?=[-0-9 ]{17}$|[-0-9X ]{13}$|[0-9X]{10}$)
  (?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
)
$
```

Параметры: режим свободного форматирования
Диалекты: .NET, PCRE, Perl, Python

См. также

Самую последнюю версию руководства пользователя «ISBN User's Manual» можно найти на сайте Международного агентства ISBN *http://www.isbn-international.org*.

Официальный список числовых идентификаторов групп на странице *http://www.isbn-international.org/en/identifiers/allidentifiers.html* поможет идентифицировать страну или область происхождения книги по номеру зоны, включающему первые от 1 до 5 цифр, в ее номере ISBN.

4.14. Проверка почтовых индексов

Задача

Необходимо выполнить проверку ZIP-кода (почтовый индекс в США), который может записываться в двух форматах: как пятизначное число или как девятизначное число (ZIP + 4). Регулярное выражение должно совпадать со строками 12345 и 12345-6789 и не совпадать со строками 1234, 123456, 123456789 или 1234-56789.

Решение

Регулярное выражение

```
^[0-9]{5}(?:-[0-9]{4})?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

VB.NET

```
If Regex.IsMatch(subjectString, "^[0-9]{5}(?:-[0-9]{4})?$") Then
    Console.WriteLine("Допустимый ZIP-код")
Else
    Console.WriteLine("Недопустимый ZIP-код")
End If
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

Ниже приводится регулярное выражение проверки ZIP-кода, разложенное на отдельные составляющие:

```
^          # Проверка совпадения с началом строки.
[0-9]{5}   # Соответствует цифре точно пять раз.
(?:        # Группировка, но несохраняющая...
  -        #   Соответствует литералу "-".
  [0-9]{4} #   Соответствует цифре точно четыре раза.
)          # Конец несохраняющей группы.
  ?        #   Повторить предшествующую группу ноль или один раз.
$          # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это регулярное выражение весьма прямолинейное, поэтому нам нечего добавить к его описанию. Чтобы использовать его для поиска ZIP-кодов внутри более длинной строки, следует заменить якорные метасимволы ‹^› и ‹$› метасимволом границы слова, что в результате даст выражение ‹\b[0-9]{5}(?:-[0-9]{4})?\b›.

См. также

Рецепты 4.15, 4.16 и 4.17.

4.15. Проверка почтовых индексов, используемых в Канаде

Задача

Необходимо проверить, является ли строка почтовым индексом, используемым в Канаде.

Решение

```
^(?!.*[DFIOQU])[A-VXY][0-9][A-Z]•[0-9][A-Z][0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Негативная опережающая проверка, находящаяся в начале этого регулярного выражения, предотвращает появление в испытуемой строке символов D, F, I, O, Q или U. Символьный класс `‹[A-VXY]›`, следующий далее, исключает возможность появления символа W или Z в начале индекса. Кроме этих двух исключений, в канадских почтовых индексах просто используется чередующаяся последовательность из шести алфавитных и цифровых символов с пробелом в середине. Например, регулярное выражение будет совпадать с последовательностью K1A 0B1, которая является почтовым индексом центрального почтового отделения города Оттава, Канада.

См. также

Рецепты 4.14, 4.16 и 4.17.

4.16. Проверка почтовых индексов, используемых в Великобритании

Задача

Необходимо написать регулярное выражение, которое будет совпадать с почтовым индексом, используемым в Великобритании.

Решение

```
^[A-Z]{1,2}[0-9R][0-9A-Z]?•[0-9][ABD-HJLNP-UW-Z]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Почтовые индексы в Великобритании могут содержать от пяти до семи алфавитно-цифровых символов, разделенных пробелом. Правила, определяющие, какие символы и в каких позициях могут появляться, отличаются большой сложностью и имеют множество исключений. Данное регулярное выражение реализует основные из этих правил.

См. также

Британский стандарт BS7666 по адресу *http://www.govtalk.gov.uk/gdsc/html/frames/PostCode.htm*, где описываются правила составления почтовых индексов.

Рецепты 4.14, 4.16 и 4.17.

4.17. Поиск адресов, содержащих номер почтового ящика

Задача

Необходимо отыскать адреса, содержащие номер почтового ящика, и предупредить пользователя о том, что информация о доставке должна содержать адрес улицы.

Решение

Регулярное выражение

```
^(?:Post•(?:Office•)?|P[.•]?O\.?•)?Box\b
```

Параметры: нечувствительность к регистру, символам ^ и $ соответствуют границы строк

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

C#

```
Regex regexObj = new Regex(
    @"^(?:Post (?:Office )?|P[. ]?O\.? )?Box\b",
    RegexOptions.IgnoreCase | RegexOptions.Multiline
);
if (regexObj.IsMatch(subjectString) {
    Console.WriteLine("Строка не выглядит, как адрес улицы");
} else {
    Console.WriteLine("Можно отправлять");
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.5.

Обсуждение

Следующее объяснение записано в режиме свободного форматирования, поэтому каждый значимый символ пробела в регулярном выражении экранирован символом обратного слэша:

```
^                # Проверка совпадения с началом строки.
(?:              # Группировка, но несохраняющая...
  Post\          #   Соответствует тексту “Post “.
  (?:Office\ )?  #   Необязательное соответствие тексту “Office “.
 |               #  или...
  P[.\ ]?        #   Соответствует “P” и, возможно, запятой или пробелу.
  O\.?\          #   Соответствует “O”, возможно точке и пробелу.
)?               # Повторить группу ноль или один раз.
Box              # Соответствует тексту “Box”.
\b               # Проверить совпадение с границей слова.
```

Параметры: нечувствительность к регистру, символам ^ и $ соответствуют границы строк

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Регулярному выражению соответствуют все примеры строк, следующие ниже, если они находятся в начале строки:

- Post Office Box
- post box
- P.O. box
- P O Box
- Po. box
- PO Box
- Box

Несмотря на все предпринятые меры предосторожности, в реальной жизни можно столкнуться с ситуацией, когда почтовое отправление доставляется не по тому адресу или когда при указанном верном адресе отправление возвращается обратно, потому что многие люди достаточно вольно трактуют адреса.

Чтобы уменьшить этот риск, лучше заранее предупредить, что начинать адрес с указания почтового ящика недопустимо. Если будет получено совпадение с этим регулярным выражением, можно предупредить пользователя, что был введен номер почтового ящика, но не было указано, как до него добраться.

См. также

Рецепты 4.14, 4.15 и 4.16.

4.18. Преобразование имен из формата «имя фамилия» в формат «фамилия, имя»

Задача

Требуется преобразовать имена людей из формата «имя фамилия» в формат «фамилия, имя» для составления списка, отсортированного по фамилиям в алфавитном порядке. При этом необходимо принять во внимание другие элементы, составляющие полное имя, например преобразовать полные имена из формата «имя отчество приставка фамилия дополнение» в формат «фамилия, имя отчество приставка дополнение».

Решение

К сожалению, с помощью регулярных выражений невозможно произвести надежный парсинг имен. Регулярные выражения задают жесткие условия совпадения, тогда как имена людей могут быть настолько необычными, что даже человек может понимать их неправильно. Для определения структуры имен или способов сортировки в алфавитном порядке часто требуется принимать во внимание национальные традиции и даже личные предпочтения. Однако, если есть возможность сделать определенные предположения относительно обрабатываемых данных и свести количество ошибок до приемлемого уровня, регулярные выражения помогут найти быстрое решение.

При составлении следующего регулярного выражения мы не пытались реализовать обработку крайних случаев и преднамеренно сохранили его простым.

Регулярное выражение

```
^(.+?)•([^\s,]+)(,?•(?:[JS]r\.?|III?|IV))?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
$2,•$1$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\2,•\1\3
```

Диалекты замещающего текста: Python, Ruby

JavaScript

```
function formatName (name) {
    return name.replace(/^(.+?) ([^\s,]+)(,? (?:[JS]r\.?|III?|IV))?$/i,
                        "$2, $1$3");
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.15.

Обсуждение

Для начала рассмотрим это регулярное выражение по частям. Комментарии, поясняющие, какие части имени соответствуют тем или иным сегментам регулярного выражения, будут даны ниже.

Так как здесь регулярное выражение записано в режиме свободного форматирования, литералы пробелов экранированы символами обратного слэша:

```
^             # Проверка совпадения с началом строки.
(             # Сохранение совпадения для обратной ссылки 1...
  .+?         # Соответствует одному или более символам,
              #   минимальное число раз.
)             # Конец сохраняющей группы.
\             # Соответствует литералу пробела.
(             # Сохранение совпадения для обратной ссылки 2...
  [^\s,]+     # Соответствует одному или более непробельным символам
              # или запятым.
)             # Конец сохраняющей группы.
(             # Сохранение совпадения для обратной ссылки 3...
  ,?\         # Соответствует ", " или " ".
  (?:         # Группировка, но не сохраняющая...
    [JS]r\.?  # Соответствует "Jr", "Jr.", "Sr" или "Sr.".
   |          # или...
    III?      # Соответствует "II" или "III".
   |          # или...
    IV        # Соответствует "IV".
  )           # Конец несохраняющей группы.
)?            # Повторить группу ноль или один раз.
$             # Проверка совпадения с концом строки.
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это регулярное выражение реализовано с учетом следующих предположений об испытуемых данных:

- Испытуемая строка содержит, по меньшей мере, имя и фамилию (остальные части полного имени считаются необязательными).
- Имя записано перед фамилией.
- Дополнение к имени, если оно присутствует, имеет одно из следующих значений: «Jr», «Jr.», «Sr», «Sr.», «II», «III» или «IV» с необязательной предшествующей запятой.

В этом регулярном выражении не решены некоторые проблемы, которые стоит рассмотреть:

- Регулярное выражение не в состоянии определять составные фамилии, в которых не используется символ дефиса. Например, полное имя `Sacha Baron Cohen` будет преобразовано в `Cohen, Sacha Baron`, тогда как правильное представление `Baron Cohen, Sacha`.
- Оно не сохраняет приставки перед фамилией, хотя иногда этого требуют национальные традиции или личные предпочтения (например, правильное представление имени «Charles de Gaulle» (Шарль де Голль) имеет вид «de Gaulle, Charles» согласно 15 изданию «Chicago Manual of Style», которое противоречит биографическому словарю Мерриам-Вебстер (Merriam-Webster's Biographical Dictionary)).
- Поскольку якорные метасимволы `‹^›` и `‹$›` привязывают совпадение к началу и концу строки, никаких замещений не может быть выполнено, если весь испытуемый текст не соответствует шаблону. Следовательно, если совпадение не будет обнаружено (например, когда испытуемый текст содержит одно только имя), имя останется без изменений.

Что касается работы регулярного выражения, оно делит полное имя на части, используя три сохраняющие группы. Затем с помощью обратных ссылок в замещающем тексте эти части собираются в требуемом порядке.

В первой сохраняющей группе используется максимально гибкий шаблон `‹.+?›` для извлечения имени вместе с любым количеством вторых имен и приставок к фамилии, таких как немецкое «von» или французское «de». Эти части полного имени рассматриваются как единое целое, поскольку они последовательно перечисляются в строке результата. Кроме того, объединение первого и второго имен помогает избежать ошибок, потому что регулярное выражение не в состоянии отличать составные имена, такие как «Mary Lou» или «Norma Jeane», от имени, за которым следует второе имя. Даже люди не всегда в состоянии визуально отличать их.

Подвыражению `‹[^\s,]+›` во второй сохраняющей группе соответствует фамилия. Гибкость этого символьного класса, подобно точке в первой сохраняющей группе, позволяет ему совпадать с любыми акцентированными и любыми другими символами, не входящими в набор Latin.

Третьей сохраняющей группе соответствует необязательное дополнение, такое как «Jr.» или «III», из предопределенного списка допустимых значений. Дополнение обрабатывается отдельно от фамилии, потому что оно должно находиться в конце преобразованной строки с полным именем.

Вернемся на мгновение к первой сохраняющей группе. Почему за точкой в первой сохраняющей группе следует минимальный квантификатор ‹+?›, тогда как за символьным классом во второй сохраняющей группе следует максимальный квантификатор ‹+›? Если бы первая группа (которая обрабатывает имя переменной длины и потому должна продвинуться по строке с полным именем как можно дальше) использовала максимальный квантификатор, то третья сохраняющая группа (которой соответствует дополнение) вообще не получила бы возможность участвовать в сопоставлении. Точка из первой группы могла бы совпасть со всеми символами до самого конца строки, а поскольку третья группа является необязательной, механизм регулярных выражений выполнял бы возврат только для того, чтобы обеспечить совпадение со второй группой, после чего объявил бы об успехе. Вторая группа может использовать максимальный квантификатор, потому что она содержит более ограниченный символьный класс, позволяющий группе совпадать только с одним элементом имени.

В табл. 4.2 приводятся некоторые примеры форматирования имен с помощью данного регулярного выражения и строки замещающего текста.

Таблица 4.2. Форматированные имена

Оригинал	Результат
Robert Downey, Jr.	Downey, Robert, Jr.
John F. Kennedy	Kennedy, John F.
Scarlett O'Hara	O'Hara, Scarlett
Pepé Le Pew	Pew, Pepé Le
J.R.R. Tolkien	Tolkien, J.R.R.
Catherine Zeta-Jones	Zeta-Jones, Catherine

Варианты

Список приставок к фамилии в начале полного имени

Добавив небольшой фрагмент, как показано в следующем регулярном выражении, можно обеспечить вывод приставок к фамилии из предо-

пределенного списка перед фамилией. В частности, это регулярное выражение распознает следующие значения: «De», «Du», «La», «Le», «St», «St.», «Ste», «Ste.», «Van» и «Von». Допускаются любые комбинации из этих значений (например, «de la»):

```
^(.+?)•((?:(?:D[eu]|L[ae]|Ste?\.?|V[ao]n)•)*[^\s,]+)↵
(,?•(?:[JS]r\.?|III?|IV))?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
$2,•$1$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\2,•\1\3
```

Диалекты замещающего текста: Python, Ruby

4.19. Проверка номеров кредитных карт

Представьте, что вы получили задание реализовать форму заказа для компании, принимающей оплату кредитными картами. Так как за каждую попытку операции с кредитной картой, даже неудачную, взимается плата, вы решили задействовать регулярное выражение, чтобы заранее отсеивать недопустимые номера кредитных карт.

Кроме того, это создаст дополнительные удобства для клиентов. Регулярное выражение в состоянии обнаруживать очевидные опечатки сразу, как только клиент завершит заполнение веб-формы. Обращение же к службе обработки кредитных карт легко может занять от 10 до 30 секунд.

Решение

Удаление пробелов и дефисов

Необходимо извлечь номер кредитной карты, введенный клиентом, и сохранить его в переменной. Перед выполнением проверки следует отыскать и удалить пробелы и дефисы. Для этого все совпадения со следующим регулярным выражением следует заменить пустым замещающим текстом:

```
[•-]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецепте 3.14 показано, как выполнить эту предварительную замену.

Проверка номера

После удаления пробелов и дефисов можно воспользоваться регулярным выражением, следующим ниже, чтобы проверить введенный номер кредитной карты на соответствие формату любой из шести основных компаний, осуществляющих обслуживание кредитных карт. В нем используются именованные сохранения, позволяющие определить название кредитной компании, услугами которой пользуется клиент:

```
^(?:
(?<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?<mastercard>5[1-5][0-9]{14}) |
(?<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?<amex>3[47][0-9]{13}) |
(?<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?<jcb>(?:2131|1800|35\d{3})\d{11})
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?:
(?P<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?P<mastercard>5[1-5][0-9]{14}) |
(?P<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?P<amex>3[47][0-9]{13}) |
(?P<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?P<jcb>(?:2131|1800|35\d{3})\d{11})
)$
```

Параметры: режим свободного форматирования
Диалекты: PCRE, Python

Диалекты Java, Perl 5.6, Perl 5.8 и Ruby 1.8 не поддерживают именованное сохранение. Поэтому в этих диалектах необходимо использовать нумерованное сохранение. Группа 1 сохраняет номер карты Visa, группа 2 – MasterCard и так далее, вплоть до группы 6 – JCB:

```
^(?:
(4[0-9]{12}(?:[0-9]{3})?) |        # Visa
(5[1-5][0-9]{14}) |                # MasterCard
(6(?:011|5[0-9][0-9])[0-9]{12}) |  # Discover
(3[47][0-9]{13}) |                 # AMEX
(3(?:0[0-5]|[68][0-9])[0-9]{11}) | # Diners Club
((?:2131|1800|35\d{3})\d{11})      # JCB
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Диалект JavaScript не поддерживает режим свободного форматирования. Удалив пробелы и комментарии, мы получим:

```
^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↵
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|↵
(3(?:0[0-5]|[68][0-9])[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если нет необходимости определять тип кредитной карты, можно убрать сохраняющие группы:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? |        # Visa
5[1-5][0-9]{14} |                # MasterCard
6(?:011|5[0-9][0-9])[0-9]{12} |  # Discover
3[47][0-9]{13} |                 # AMEX
3(?:0[0-5]|[68][0-9])[0-9]{11} | # Diners Club
(?:2131|1800|35\d{3})\d{11}      # JCB
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Или для JavaScript:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9][0-9])[0-9]{12}|↵
3[47][0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Рецепт 3.6 поможет добавить это регулярное выражение в форму заказа, чтобы реализовать проверку номера карты. Если для обработки разных типов карт используются разные службы или если необходимо сохранить некоторую информацию для статистики, можно воспользоваться рецептом 3.9, чтобы узнать, какой именованной или нумерованной сохраняющей группе принадлежит совпадение. Это поможет узнать название кредитной компании, услугами которой пользуется клиент.

Пример веб-страницы со сценарием JavaScript

```
<html>
<head>
<title>Credit Card Test</title>
</head>
<body>
<h1>Проверка кредитной карты</h1>
```

```
<form>
<p>Пожалуйста, введите номер вашей кредитной карты:</p>

<p><input type="text" size="20" name="cardnumber"
  onkeyup="validatecardnumber(this.value)"></p>

<p id="notice">(номер карты не был введен)</p>
</form>

<script>
function validatecardnumber(cardnumber) {
  // Удалить пробелы и дефисы
  cardnumber = cardnumber.replace(/[ -]/g, '');
  // Проверить корректность номера карты
  // Регулярное выражение сохранит номер в одной из сохраняющих групп
  var match = /^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↵
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|[68][0-9])↵
[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$/.exec(cardnumber);
  if (match) {
    // Список типов карт, в том же порядке, что и в регулярном выражении
    var types = ['Visa', 'MasterCard', 'Discover', 'American Express',
                 'Diners Club', 'JCB'];
    // Отыскать совпавшую сохраняющую группу
    // Пропустить нулевой элемент в массиве совпадений (общее совпадение)
    for (var i = 1; i < match.length; i++) {
      if (match[i]) {
        // Вывести тип карты для данной группы
        document.getElementById('notice').innerHTML = types[i - 1];
        break;
      }
    }
  } else {
    document.getElementById('notice').innerHTML = '(ошибка в номере карты)';
  }
}
</script>
</body>
</html>
```

Обсуждение

Удаление пробелов и дефисов

Цифры рельефного номера на самой кредитной карте обычно делятся на группы, по четыре цифры в каждой. Это упрощает чтение номера. Вполне естественно, что многие пытаются ввести номер карты в таком же виде, включая пробелы.

Написать регулярное выражение, которое будет проверять номер кредитной карты, допуская возможность появления пробелов, дефисов и прочих разделителей, намного сложнее, чем регулярное выражение, которое допускает только появление цифр. Поэтому, чтобы не утруждать клиента, заставляя его повторно вводить номер без пробелов или дефисов, можно просто выполнить операцию поиска с заменой и удалить их, прежде чем приступать к проверке номера и выполнять его передачу службе обработки кредитных карт.

Регулярному выражению ‹[•-]› соответствует символ пробела или дефиса. Замена всех совпадений с этим регулярным выражением пустым текстом обеспечит удаление всех пробелов и дефисов.

Номера кредитных карт могут состоять только из цифр. Поэтому вместо символьного класса ‹[•-]›, позволяющего удалить только пробелы и дефисы, можно воспользоваться метасимволом ‹\D›, что позволит удалить все символы, не являющиеся цифрами.

Проверка номера

Каждая компания, обслуживающая кредитные карты, использует свой формат номера. Мы использовали это обстоятельство, чтобы не заставлять клиента вводить название компании вместе с номером – название компании определяется по номеру. Ниже приводятся описания форматов для каждой компании:

Visa

13 или 16 цифр, номер начинается с 4.

MasterCard

16 цифр, номер начинается с чисел от 51 до 55.

Discover

16 цифр, номер начинается с 6011, или с 65.

American Express

15 цифр, номер начинается с 34 или 37.

Diners Club

14 цифр, номер начинается с чисел от 300 до 305, 36 или 38.

JCB

15 цифр, номер начинается с чисел 2131 или 1800, или 16 цифр, номер начинается с 35.

Если необходимо обеспечить прием кредитных карт только определенных типов, нежелательные типы можно удалить из регулярного выражения. При удалении JCB не забудьте также удалить из регулярного выражения последний оператор ‹|›. Если регулярное выражение будет содержать в конце последовательность ‹||› или ‹|)›, оно будет

интерпретировать пустые строки как допустимые номера кредитных карт.

Например, чтобы обеспечить прием кредитных карт только компаний Visa, MasterCard и AMEX, можно воспользоваться следующим регулярным выражением:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? | # Visa
5[1-5][0-9]{14} |         # MasterCard
3[47][0-9]{13}            # AMEX
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Альтернативное решение:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13})$
```

Regex options: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если необходимо отыскать номера кредитных карт в тексте документа, якорные метасимволы следует заменить метасимволом ‹\b› границы слова.

Внедрение решения в веб-страницу

Пример в разделе «Пример веб-страницы со сценарием JavaScript» на стр. 347 демонстрирует, как можно было бы добавить эти два регулярных выражения в форму заказа. Для поля ввода номера кредитной карты определен обработчик события `onkeyup`, который вызывает функцию `validatecardnumber()`. Эта функция извлекает номер кредитной карты из поля ввода, удаляет из него пробелы и дефисы и выполняет проверку с помощью регулярного выражения с нумерованными сохраняющими группами. Результат проверки отображается путем замещения содержимого последнего параграфа на странице.

Если регулярное выражение не обнаружит совпадения, метод `regexp.exec()` вернет значение `null` и на странице будет выведен текст (`ошибка в номере карты`). Если совпадение будет найдено, метод `regexp.exec()` вернет массив строк. Нулевой элемент этого массива хранит совпадение со всем регулярным выражением. Элементы с 1 по 6 хранят фрагменты, совпавшие с соответствующими им сохраняющими группами.

В нашем регулярном выражении присутствует шесть сохраняющих групп, разделенных операторами выбора. Это означает, что только одна группа будет участвовать в совпадении и сохранит номер кредитной карты. Остальные элементы массива будут содержать пустой текст (значение `undefined` или пустую строку, в зависимости от используемого броузера). Функция проверяет шесть групп, одну за другой. Когда об-

наруживается непустая группа, номер карты опознается и на странице выводится ее тип.

Дополнительная проверка с применением алгоритма Луна

Имеется возможность произвести дополнительную проверку номера кредитной карты, прежде чем приступать к обработке заказа. Последняя цифра в номере кредитной карты является контрольной суммой, вычисляемой в соответствии с *алгоритмом Луна*. Так как этот алгоритм требует выполнения арифметических операций, его невозможно реализовать средствами регулярных выражений.

Проверку с применением алгоритма Луна можно добавить в пример веб-страницы, который приводился в этом рецепте, вставив вызов функции `luhn(cardnumber);` перед инструкцией `else` в функции `validatecardnumber()`. В этом случае проверка с применением алгоритма Луна будет выполняться, только когда регулярное выражение будет обнаруживать совпадение и после определения типа кредитной карты. При этом для алгоритма Луна определять тип кредитной карты не требуется. Для всех кредитных карт используется один и тот же метод вычисления контрольной суммы.

На языке JavaScript алгоритм Луна можно реализовать, как показано ниже:

```
function luhn(cardnumber) {
  // Создать массив с цифрами, составляющими номер кредитной карты
  var getdigits = /\d/g;
  var digits = [];
  while (match = getdigits.exec(cardnumber)) {
    digits.push(parseInt(match[0], 10));
  }
  // Выполнить расчет по алгоритму Луна для массива
  var sum = 0;
  var alt = false;
  for (var i = digits.length - 1; i >= 0; i--) {
    if (alt) {
      digits[i] *= 2;
      if (digits[i] > 9) {
        digits[i] -= 9;
      }
    }
    sum += digits[i];
    alt = !alt;
  }
  // Сообщить о корректности номера карты
```

```
  if (sum % 10 == 0) {
    document.getElementById("notice").innerHTML += '; проверка Луна пройде-
на;
  } else {
    document.getElementById("notice").innerHTML +=
        '; проверка Луна не пройдена ';
  }
}
```

Эта функция принимает строку с номером кредитной карты в виде аргумента. Номер кредитной карты может содержать только цифры. В нашем примере функция `validatecardnumber()` уже удалила пробелы и дефисы и определила, что номер карты содержит допустимое число цифр.

Сначала функция задействует регулярное выражение `‹\d›`, чтобы в цикле обойти все цифры, содержащиеся в строке. Обратите внимание на модификатор `/g`. Внутри цикла обращением к `match[0]` извлекается совпавшая цифра. Так как регулярные выражения работают только с текстом (со строками), вызывается функция `parseInt()`, чтобы сохранить в элементе массива целое число, а не строку Если этого не сделать, в результате в переменной `sum` окажется строка, собранная из цифр, а не сумма чисел.

Фактический алгоритм работает с массивом и вычисляет контрольную сумму. Если остаток от деления контрольной суммы на число 10 равен нулю, номер карты считается верным. В противном случае номер содержит ошибку.

4.20. Европейские регистрационные номера плательщиков НДС

Задача

Вам дано задание реализовать веб-форму заказа для компании, работающей в Евросоюзе.

Европейские налоговые законы предусматривают следующее: когда компания, имеющая регистрационный номер плательщика НДС (ваш клиент) и размещающаяся в одной из стран Евросоюза, приобретает какой-либо товар у поставщика (ваша компания), размещающегося в любой другой стране Евросоюза, поставщик не должен закладывать НДС (налог на добавленную стоимость) в стоимость товара. Если покупатель не имеет регистрационного номера плательщика НДС, поставщик должен включить НДС в стоимость и перевести сумму НДС местной налоговой службе. Поставщик обязан использовать регистрационный номер плательщика НДС покупателя как доказательство для налоговой службы, что он не должен перечислять НДС. Это означает, что

для поставщика очень важно проверить регистрационный номер плательщика НДС покупателя, прежде чем выполнить сделку, не облагаемую налогом.

Самой часто встречающейся причиной некорректности регистрационных номеров являются обычные опечатки, допущенные клиентом. Чтобы сделать процесс оформления заказа более дружественным и быстрым, необходимо использовать регулярное выражение и проверить регистрационный номер, пока клиент заполняет форму заказа. Реализовать такую проверку можно на стороне клиента, в JavaScript, или на стороне сервера, с помощью сценария, принимающего форму заказа. Если номер не соответствует регулярному выражению, клиент сможет тут же исправить опечатку.

Решение

Удаление пробелов и знаков пунктуации

Необходимо извлечь регистрационный номер плательщика НДС, введенный клиентом, и сохранить его в переменной. Прежде чем выполнять проверку корректности номера, следует выполнить глобальную операцию поиска с заменой, чтобы заменить пустыми строками все совпадения со следующим регулярным выражением:

```
[-.•]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецепте 3.14 показано, как выполнить эту предварительную замену. Мы предполагаем, что клиент не может вводить какие-либо знаки пунктуации, за исключением дефисов, точек и пробелов. Любые другие недопустимые символы будут обнаруживаться при последующей проверке.

Проверка номера

После удаления пробелов и знаков пунктуации можно выполнить проверку регистрационного номера для любой из 27 стран, входящих в Евросоюз, с помощью следующего регулярного выражения:

```
^(
(AT)?U[0-9]{8} |                          # Австрия
(BE)?0?[0-9]{9} |                         # Бельгия
(BG)?[0-9]{9,10} |                        # Болгария
(CY)?[0-9]{8}L |                          # Кипр
(CZ)?[0-9]{8,10} |                        # Чешская республика
(DE)?[0-9]{9} |                           # Германия
(DK)?[0-9]{8} |                           # Дания
(EE)?[0-9]{9} |                           # Эстония
```

```
(EL|GR)?[0-9]{9} |                          # Греция
(ES)?[0-9A-Z][0-9]{7}[0-9A-Z] |             # Испания
(FI)?[0-9]{8} |                             # Финляндия
(FR)?[0-9A-Z]{2}[0-9]{9} |                  # Франция
(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3}) | # Соединенное Королевство
(HU)?[0-9]{8} |                             # Венгрия
(IE)?[0-9]S[0-9]{5}L |                      # Ирландия
(IT)?[0-9]{11} |                            # Италия
(LT)?([0-9]{9}|[0-9]{12}) |                 # Литва
(LU)?[0-9]{8} |                             # Люксембург
(LV)?[0-9]{11} |                            # Латвия
(MT)?[0-9]{8} |                             # Мальта
(NL)?[0-9]{9}B[0-9]{2} |                    # Нидерланды
(PL)?[0-9]{10} |                            # Польша
(PT)?[0-9]{9} |                             # Португалия
(RO)?[0-9]{2,10} |                          # Румыния
(SE)?[0-9]{12} |                            # Швеция
(SI)?[0-9]{8} |                             # Словения
(SK)?[0-9]{10}                              # Словакия
)$
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В этом регулярном выражении используется режим свободного форматирования, чтобы упростить его изменение в будущем. Время от времени к Евросоюзу присоединяются новые страны, и страны-участницы меняют свои правила, касающиеся регистрационных номеров плательщиков НДС. К сожалению, в JavaScript не поддерживается режим свободного форматирования. Для этого языка программирования все регулярное выражение следует поместить в одну строку:

```
^((AT)?U[0-9]{8}|(BE)?0?[0-9]{9}|(BG)?[0-9]{9,10}|(CY)?[0-9]{8}L|↵
(CZ)?[0-9]{8,10}|(DE)?[0-9]{9}|(DK)?[0-9]{8}|(EE)?[0-9]{9}|↵
(EL|GR)?[0-9]{9}|(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]|(FI)?[0-9]{8}|↵
(FR)?[0-9A-Z]{2}[0-9]{9}|(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})|↵
(HU)?[0-9]{8}|(IE)?[0-9]S[0-9]{5}L|(IT)?[0-9]{11}|↵
(LT)?([0-9]{9}|[0-9]{12})|(LU)?[0-9]{8}|(LV)?[0-9]{11}|(MT)?[0-9]{8}|↵
(NL)?[0-9]{9}B[0-9]{2}|(PL)?[0-9]{10}|(PT)?[0-9]{9}|(RO)?[0-9]{2,10}|↵
(SE)?[0-9]{12}|(SI)?[0-9]{8}|(SK)?[0-9]{10})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Рецепт 3.6 поможет добавить это регулярное выражение в форму заказа.

Обсуждение

Удаление пробелов и знаков пунктуации

Чтобы упростить чтение регистрационных номеров плательщиков НДС, люди часто при вводе добавляют дополнительные знаки пунктуации, чтобы разбить их на группы цифр. Например, клиент из Германии может ввести свой номер как `DE 123.456.789`.

Практически невозможно написать единственное регулярное выражение, которому соответствуют любые регистрационные номера для любой из 27 стран, введенные в любом формате. Поскольку знаки пунктуации служат исключительно для повышения удобочитаемости, намного проще будет сначала удалить их и только потом выполнить проверку получившегося регистрационного номера.

Регулярному выражению `‹[-.•]›` соответствует дефис, точка или пробел. Операция замены всех совпадений с этим регулярным выражением пустой строкой фактически удалит знаки пунктуации, часто используемые при вводе регистрационных номеров плательщиков НДС.

Регистрационные номера состоят только из букв и цифр. Поэтому, вместо выражения `‹[-.•]›` для удаления любых недопустимых символов можно использовать выражение `‹[^A-Z0-9]›`.

Проверка номера

Оба регулярных выражения, проверяющие регистрационный номер, являются идентичными. Единственное отличие состоит в том, что в первом используется режим свободного форматирования, что делает его более удобочитаемым, и в нем присутствуют комментарии, указывающие на страны. Диалект JavaScript не поддерживает режим свободного форматирования, тогда как другие диалекты предоставляют такую возможность.

Чтобы обеспечить совпадение с регистрационным номером для любой из 27 стран, в регулярном выражении используется оператор выбора. Форматы номеров имеют следующий вид:

Австрия

U99999999

Бельгия

999999999 или 0999999999

Болгария

999999999 или 9999999999

Кипр

99999999L

Чешская республика

99999999, 999999999 или 9999999999

Германия

999999999

Дания

99999999

Эстония

999999999

Греция

999999999

Испания

X9999999X

Финляндия

99999999

Франция

XX999999999

Великобритания

999999999, 999999999999 или XX999

Венгрия

99999999

Ирландия

9S99999L

Италия

99999999999

Литва

999999999 или 99999999999

Люксембург

99999999

Латвия

99999999999

Мальта

99999999

Нидерланды

999999999B99

Польша

999999999

Португалия

999999999

Румыния

99, 999, 9999, 99999, 999999, 9999999, 99999999, 999999999 или 9999999999

Швеция

99999999999

Словения

99999999

Словакия

999999999

Строго говоря, двухбуквенные обозначения стран являются частью регистрационного номера. Однако люди часто опускают их, так как название страны уже указывается в счете. Регулярное выражение будет принимать регистрационные номера и без кода страны. Если необходимо, чтобы код страны вводился в обязательном порядке, следует убрать из регулярного выражения все знаки вопроса. В этом случае при выводе сообщения об ошибке необходимо известить клиента, что код страны требуется указывать в обязательном порядке.

Если предполагается принимать заказы только из определенных стран, можно убрать из регулярного выражения те страны, которые не разрешено выбрать в форме заказа. Удаляя альтернативы из регулярного выражения, не забывайте удалять и операторы выбора ‹|›, отделяющие удаляемую альтернативу от предыдущей или последующей альтернативы. Если этого не сделать, в регулярном выражении окажутся два оператора выбора, следующие друг за другом ‹||›. Последовательность ‹||› добавит в регулярное выражение альтернативу, которой соответствует пустая строка, сделав возможной ситуацию, когда форма заказа будет воспринимать пустой регистрационный номер, как корректный.

Все 27 альтернатив сгруппированы в одну группу. Эта группа помещена между символом крышки и знаком доллара, которые привязывают совпадение с регулярным выражением к началу и к концу испытуемой строки. Вследствие этого вся введенная строка будет проверяться как регистрационный номер.

Если возникнет необходимость отыскивать регистрационные номера в тексте документа, якорные метасимволы следует заменить метасимволом ‹\b› границы слова.

Варианты

Преимущество регулярного выражения, выполняющего проверку сразу для всех 27 стран, состоит в том, что в форму заказа достаточно будет добавить всего одно выражение. Можно было бы расширить форму заказа, используя 27 отдельных регулярных выражений. При этом сначала можно было бы определять страну, указанную в расчетном адресе, а затем выбирать регулярное выражение, соответствующее этой стране:

Австрия

```
‹^(AT)?U[0-9]{8}$›
```

Бельгия

```
‹^(BE)?0?[0-9]{9}$›
```

Болгария

```
‹^(BG)?[0-9]{9,10}$›
```

Кипр

```
‹^(CY)?[0-9]{8}L$›
```

Чешская республика

```
‹^(CZ)?[0-9]{8,10}$›
```

Германия

```
‹^(DE)?[0-9]{9}$›
```

Дания

```
‹^(DK)?[0-9]{8}$›
```

Эстония

```
‹^(EE)?[0-9]{9}$›
```

Греция

```
‹^(EL|GR)?[0-9]{9}$›
```

Испания

```
‹^(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]$›
```

Финляндия

```
‹^(FI)?[0-9]{8}$›
```

Франция

```
‹^(FR)?[0-9A-Z]{2}[0-9]{9}$›
```

Великобритания

```
‹^(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})$›
```

Венгрия

```
‹^(HU)?[0-9]{8}$›
```

Ирландия

```
‹^(IE)?[0-9]S[0-9]{5}L$›
```

Италия

```
‹^(IT)?[0-9]{11}$›
```

Литва

```
‹^(LT)?([0-9]{9}|[0-9]{12})$›
```

Люксембург

```
‹^(LU)?[0-9]{8}$›
```

Латвия

```
‹^(LV)?[0-9]{11}$›
```

Мальта

```
‹^(MT)?[0-9]{8}$›
```

Нидерланды

```
‹^(NL)?[0-9]{9}B[0-9]{2}$›
```

Польша

```
‹^(PL)?[0-9]{10}$›
```

Португалия

```
‹^(PT)?[0-9]{9}$›
```

Румыния

```
‹^(RO)?[0-9]{2,10}$›
```

Швеция

```
‹^(SE)?[0-9]{12}$›
```

Словения

```
‹^(SI)?[0-9]{8}$›
```

Словакия

```
‹^(SK)?[0-9]{10}$›
```

Пример реализации проверки регистрационного номера выбранным регулярным выражением можно найти в рецепте 3.6. Такая реализация позволит определить корректность регистрационного номера для страны, указанной клиентом.

Главное преимущество использования отдельных регулярных выражений состоит в том, что появляется возможность вставлять код страны в начало номера, не требуя этого от клиента. Если регулярное выражение соответствует введенному номеру, можно проверить содержимое первой сохраняющей группы. Как это сделать, описывается в рецепте 3.9. Если первая сохраняющая группа ничего не содержит, следовательно, клиент не ввел код страны. В этом случае можно до-

бавить код страны перед номером, прежде чем сохранять его у себя в базе данных.

Греческие регистрационные номера могут содержать один из двух кодов страны. Код `EL` традиционно используется в греческих регистрационных номерах, а код `GR` является стандартизованным кодом страны для Греции.

См. также

Регулярное выражение просто проверяет, выглядит ли введенная последовательность как регистрационный номер плательщика НДС. Этого вполне достаточно, чтобы выявить непредумышленные ошибки. Регулярное выражение не проверяет, принадлежит ли введенный регистрационный номер компании, указанной в форме заказа. Проверить принадлежность регистрационного номера, если таковой вообще был кому-то присвоен, можно на странице *http://ec.europa.eu/taxation_customs/vies/vieshome.do*.

Приемы, использованные в регулярном выражении, рассматриваются в рецептах 2.3, 2.5 и 2.8.

5

Слова, строки и специальные символы

В этой главе содержатся рецепты, связанные с поиском и манипулированием текстом в различных контекстах. Одни рецепты демонстрируют, как реализовать расширенные функции поисковых механизмов, такие как поиск одного из нескольких слов или поиск слов, находящихся рядом друг с другом. Другие помогут вам отыскивать целые строки, содержащие определенные слова, удалять повторяющиеся слова или экранировать метасимволы регулярных выражений.

Основная цель этой главы состоит в том, чтобы продемонстрировать различные приемы и конструкции регулярных выражений в действии. Чтение этой главы сродни тренировке в применении большого числа особенностей регулярных выражений, и должно помочь вам наработать навыки применения регулярных выражений в решении стоящих перед вами задач. Во многих случаях мы ограничиваемся поиском простого текста, но предлагаемые нами шаблоны решений могут быть адаптированы вами под решение конкретных проблем.

5.1. Поиск определенного слова

Задача

Перед вами стоит простая задача – отыскать все вхождения слова «cat», без учета регистра символов. Выражение должно обнаруживать только те вхождения, которые являются отдельными словами. Оно не должно обнаруживать вхождения, являющиеся частью других слов, таких как `hellcat`, `application` или `Catwoman`.

Решение

Решить эту проблему помогут метасимволы границы слова:

```
\bcat\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Как можно использовать это регулярное выражение для поиска всех совпадений, демонстрируется в рецепте 3.7. Как заменить найденные совпадения другим текстом, демонстрируется в рецепте 3.14.

Обсуждение

Метасимволы границы слова, находящиеся с обоих концов регулярного выражения, гарантируют, что совпадение с текстом `cat` будет обнаруживаться, только если он появляется только как самостоятельное слово. Точнее, границы слова требуют, чтобы фрагмент `cat` отделялся от остального текста началом или концом строки, пробельными символами, знаками пунктуации или другими символами, не являющимися символами слова.

Механизмы регулярных выражений интерпретируют буквы, цифры и символ подчеркивания, как символы слова. Более подробно границы слова обсуждаются в рецепте 2.6.

В диалектах JavaScript, PCRE и Ruby могут возникнуть проблемы с интернациональным текстом, так как в этих диалектах при определении границ слова в учет принимаются только символы кодировки ASCII. Другими словами, границы слова обнаруживаются только в позициях между совпадениями с выражениями `‹^|[^A-Za-z0-9_]›` и `‹[A-Za-z0-9_]›` или между совпадениями с выражениями `‹[A-Za-z0-9_]›` и `‹[^A-Za-z0-9_]|$›`. То же относится и к Python, если при создании регулярного выражения не был установлен флаг `UNICODE` или `U`. Эта особенность препятствует использованию метасимвола `‹\b›` для поиска «только целого слова» в тексте, содержащем символы из алфавитов, отличных от алфавита Latin. Например, в JavaScript, PCRE и Ruby регулярное выражение `‹\büber\b›` будет обнаруживать совпадение внутри слова `darüber`, и не будет обнаруживать внутри текста `dar über`. В большинстве случаев такой результат представляет собой полную противоположность желаемому. Проблема обусловлена тем, что символ `ü` интерпретируется не как символ слова, поэтому метасимвол границы слова обнаруживает совпадение между символами `rü`. В свою очередь, между пробелом и символом `ü` граница слова не обнаруживается, потому что они образуют непрерывную последовательность символов, не являющихся символами слова.

Решить эту проблему можно, применив вместо метасимвола границы слова опережающую и ретроспективную проверки (*проверки соседних символов*). Подобно метасимволу границы слова проверка соседних символов обеспечивает совпадение нулевой длины с некоторой позицией. В диалектах PCRE (когда эта библиотека скомпилирована с поддержкой UTF-8) и Ruby 1.9 имеется возможность имитировать поддержку границы слова в кодировке Юникод, например `‹(?<=\P{L}|^)cat(?=\P{L}|$)›`. В этом регулярном выражении использованы инверти-

рованные метасимволы свойства Letter Юникода (‹\P{L}›), которые рассматривались в рецепте 2.7. Проверка соседних символов рассматривается в рецепте 2.16. Если потребуется, чтобы проверка соседних символов интерпретировала цифры и символ подчеркивания как символы слова (подобно метасимволу ‹\b›), достаточно будет заменить две конструкции ‹\P{L}› символьным классом ‹[^\p{L}\p{N}_]›.

В JavaScript и Ruby 1.8 не поддерживаются ни ретроспективная проверка, ни свойства Юникода. Обойти отсутствие поддержки ретроспективной проверки в этих диалектах можно сопоставлением с символом, не являющимся символом слова, перед каждым совпадением, а затем либо удалять его из каждого совпадения, либо возвращать его обратно в строку при выполнении операции замещения (примеры использования фрагментов совпадений в замещающем тексте приводятся в рецепте 3.15). Кроме того, отсутствие поддержки свойств Юникода (вместе с тем фактом, что в обоих языках программирования метасимволы ‹\w› и ‹\W› поддерживают только символы ASCII) означает, что придется обходиться менее универсальным решением. Кодовые пункты из категории Letter рассеяны по всему набору символов Юникода, поэтому потребовались бы тысячи символов, чтобы имитировать конструкцию ‹\p{L}› с помощью экранированных последовательностей Юникода и символьных классов. Неплохим компромиссом мог бы стать символьный класс ‹[A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF]›, которому соответствуют все буквы Юникода в восьмибитовом адресном пространстве, то есть первые 256 кодовых пунктов Юникода со значениями от 0x00 до 0xFF (список соответствующих символов приводится на рис. 5.1). Этот символьный класс будет совпадать со множеством (или, в инвертированной форме, исключать множество) наиболее часто используемых символов, находящихся за пределами адресного пространства семибитного набора ASCII.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
⋮																
4		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z					
6		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z					
⋮																
A											ª					
B						µ					º					
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö		Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö		ø	ù	ú	û	ü	ý	þ	ÿ

Рис. 5.1. *Алфавитные символы Юникода в восьмибитном адресном пространстве*

Ниже демонстрируется, как можно заменить все вхождения слова «cat» словом «dog» в JavaScript. Этот пример корректно опознает все наиболее типичные акцентированные символы, поэтому слово ëcat останется без изменений. Для этого потребовалось сконструировать свой собственный символьный класс и отказаться от использования встроенных метасимволов ‹\b› и ‹\w›:

```
// 8-битные алфавитные символы
var L = 'A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF';
var pattern = '([^{L}]|^)cat([^{L}]|$)'.replace(/{L}/g, L);
var regex = new RegExp(pattern, 'gi');

// заменить cat на dog и вернуть в строку
// любые совпавшие дополнительные символы
subject = subject.replace(regex, '$1dog$2');
```

Примечательно, что для вставки специальных символов в строковый литерал в JavaScript используется форма записи \x*HH* (где *HH* – это двухзначное шестнадцатеричное число). Следовательно, переменная L, которая передается в регулярное выражение, будет содержать литеральные версии символов. Если необходимо, чтобы в регулярное выражение передавались сами метапоследовательности \x*HH*, их необходимо экранировать в строковом литерале с помощью символа обратного слэша (например, "\\x*HH*") . Однако в данном случае это не имеет большого значения и не окажет влияния на совпадение с регулярным выражением.

См. также

Рецепты 5.2, 5.3 и 5.4.

5.2. Поиск любого слова из множества

Задача

Необходимо отыскать любое слово из списка, не выполняя несколько операций поиска в испытуемом тексте.

Решение

С использованием конструкции выбора

Самое простое решение заключается в создании конструкции выбора из требуемых слов:

```
\b(?:one|two|three)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Более сложные примеры поиска похожих слов рассматриваются в рецепте 5.3.

Пример решения на языке JavaScript

```
var subject = 'One times two plus one equals three.';

var regex = /\b(?:one|two|three)\b/gi;

subject.match(regex);
// вернет массив с четырьмя совпадениями: ['One','two','one','three']

// Эта функция делает то же самое,
// но принимает искомые слова в виде массива.
// Любые метасимволы регулярных выражений,
// присутствующие внутри искомых слов,
// перед поиском экранируются символами обратного слэша.

function match_words (subject, words) {
    var regex_metachars = /[(){}[\]*+?.\\^$|,\-]/g;

    for (var i = 0; i < words.length; i++) {
        words[i] = words[i].replace(regex_metachars, '\\$&');
    }

    var regex = new RegExp('\\b(?:' + words.join('|') + ')\\b', 'gi');

    return subject.match(regex) || [];
}

match_words(subject, ['one','two','three']);
// вернет массив с четырьмя совпадениями: ['One','two','one','three']
```

Обсуждение

С использованием конструкции выбора

Это регулярное выражение состоит из трех частей: границы слова с обоих концов, несохраняющая группа и список слов (отделяются друг от друга оператором выбора ‹|›). Границы слова гарантируют, что регулярное выражение не совпадет с частью более длинного слова. Несохраняющая группировка ограничивает область действия операторов выбора – в противном случае для достижения того же эффекта регулярное выражение пришлось бы записать как ‹\bone\b|\btwo\b|\bthree\b›. Каждое из слов просто соответствует самому себе.

Так как механизм регулярных выражений пытается найти совпадение с каждым словом из списка, просматривая его слева направо, мож-

но заметить некоторое увеличение производительности, если в начало списка поместить слова, которые могут встречаться в испытуемом тексте с большей долей вероятности. Так как слова с обеих сторон окружены границами слова, они могут появляться в любом порядке. Однако, если бы в выражении отсутствовали границы слова, тогда важно было бы первыми указать наиболее длинные слова, в противном случае выражение `‹awe|awesome›` никогда не смогло бы найти совпадение со словом «awesome», так как оно всегда обнаруживало бы совпадение с «awe» в начале слова.

Обратите внимание, что это регулярное выражение предназначено лишь для того, чтобы в общих чертах продемонстрировать возможность совпадения с одним словом из списка.

Поскольку в данном примере оба слова `‹two›` и `‹three›` начинаются с одного и того же символа, есть возможность помочь механизму регулярных выражений, переписав регулярное выражение как `‹\b(?:one|t(?:wo|hree))\b›`. В рецепте 5.3 приводятся дополнительные примеры, демонстрирующие, как эффективнее выполнять поиск одного слова из списка похожих слов.

Пример решения на языке JavaScript

Пример на языке JavaScript сопоставляет тот же список слов двумя разными способами. В первом случае просто создается регулярное выражение и с помощью метода `match()`, имеющегося у строк в языке JavaScript, выполняется поиск в испытуемом тексте. При вызове методу `match()` передается регулярное выражение, использующее флаг `/g` («global» – глобальный). Он возвращает массив всех совпадений, обнаруженных в строке, или значение `null`, если не было найдено ни одного совпадения.

Во втором случае используется функция (`match_words()`), принимающая испытуемую строку, внутри которой требуется выполнить поиск, и массив искомых слов. Эта функция сначала экранирует любые метасимволы регулярных выражений, которые могут присутствовать в искомых словах, и затем компонует из списка слов новое регулярное выражение, используемое для поиска в строке.

Функция возвращает массив обнаруженных совпадений или пустой массив, если сгенерированным регулярным выражением не было найдено ни одного совпадения. Благодаря применению флага нечувствительности к регистру символов (`/i`) искомые слова могут состоять из символов верхнего и нижнего регистра в любой комбинации.

См. также

Рецепты 5.1, 5.3 и 5.4.

5.3. Поиск похожих слов

Задача

В данном случае необходимо решить несколько задач:

- Отыскать все вхождения обоих слов `color` и `colour` в строке.
- Отыскать любое из трех слов, оканчивающееся на `bat`, `cat` или `rat`.
- Отыскать любое слово, оканчивающееся на `phobia`.
- Отыскать наиболее распространенные варианты записи имени «Steven»: `Steve`, `Steven` и `Stephen`.
- Отыскать совпадение с любой формой термина «regular expression».

Решение

Ниже приводятся регулярные выражения, решающие поставленные задачи в порядке их следования. Во всех решениях используется режим нечувствительности к регистру символов.

Color или colour

```
\bcolou?r\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Bat, cat или rat

```
\b[bcr]at\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Слова, заканчивающиеся на «phobia»

```
\b\w*phobia\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Steve, Steven или Stephen

```
\bSte(?:ven?|phen)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Варианты написания термина «regular expression»

```
\breg(?:ular•expressions?|ex(?:ps?|e[sn])?)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Использование границ слова для обеспечения совпадения с целыми словами

Для обеспечения совпадения с целыми словами во всех пяти регулярных выражениях используются метасимволы границы слова (‹\b›) . Во всех шаблонах используются разные способы сопоставления с вариантами слов, которым они соответствуют.

Рассмотрим каждое из решений подробнее.

Color или Colour

Это регулярное выражение будет совпадать со словом color или colour, но не будет совпадать со словом colorblind. В нем используется квантификатор ‹?›, чтобы сделать необязательным совпадение с предшествующим ему символом «u». Квантификаторы, такие как ‹?›, действуют не как шаблонные символы, знакомые многим. Они непосредственно связаны с предшествующими им элементами, которые могут представлять собой либо одиночный элемент (как литерал символа «u» в данном случае), либо группу элементов, заключенных в круглые скобки. Квантификатор ‹?› повторяет предшествующий ему элемент ноль или один раз. Механизм регулярных выражений сначала пытается сопоставить элемент, с которым связан квантификатор, и если эта попытка оказывается неудачной, механизм продолжает движение вперед, без сопоставления элемента. Любой квантификатор, допускающий нулевое число повторений, фактически делает необязательным совпадение с предшествующим ему элементом, и это как раз то, что требуется в данном случае.

Bat, cat или rat

В этом регулярном выражении для совпадения с символом «b», «c» или «r» используется символьный класс, за которым следуют литералы символов «at». То же самое можно было бы сделать с помощью выражений ‹\b(?:b|c|r)at\b›, ‹\b(?:bat|cat|rat)\b› или ‹\bbat\b|\bcat\b|\brat\b›. Однако всякий раз, когда выбор между возможными совпадениями заключается в выборе единственного символа из списка, лучше использовать символьные классы. Мало того, что символьные классы обеспечивают более компактную и удобочитаемую форму записи (благодаря отсутствию операторов выбора и возможности использовать диапазоны, такие как A–Z), большинство механизмов регулярных выражений также обеспечивают превосходную оптимизацию обработки символьных классов. Операторы выбора требуют от механизма регулярных выраже-

ний использовать дорогостоящий, в смысле вычислительных ресурсов, алгоритм возвратов, тогда как для сопоставления с символьными классами задействуется более простой метод поиска.

Тем не менее, несколько слов предостережения. Символьные классы относятся к элементам регулярных выражений, которые наиболее часто применяются неверно. Возможно, проблема в недостаточно подробной документации или, возможно, в недостаточно внимательном ознакомлении с нею. Как бы то ни было, не повторяйте ошибок, характерных для начинающих. Символьные классы способны обеспечить совпадение лишь с одним из символов, перечисленных в них, – и только это.

Ниже приводятся два наиболее типичных примера неправильного использования символьных классов:

Помещение слов в символьные классы

Вне всяких сомнений, выражение `<[cat]{3}>` будет обнаруживать совпадение со словом cat, но оно также будет совпадать со словами act, ttt и любыми другими комбинациями из перечисленных символов. То же относится и к инвертированным классам, таким как `<[^cat]>`, которому соответствует любой символ, за исключением `c`, `a` и `t`.

Попытка использовать оператор выбора в символьном классе

Символьный класс по определению обеспечивает выбор между символами, перечисленными в нем. Конструкции `<[a|b|c]>` соответствует один символ из множества «abc|». Что, вполне возможно, совсем не то, что вы подразумевали. Но даже если это не так, данный символьный класс содержит лишние символы вертикальной черты.

Все подробности, которые могут пригодиться для правильного и эффективного использования символьных классов, приводятся в рецепте 2.3.

Слова, заканчивающиеся на «phobia»

Как и в предыдущем регулярном выражении, здесь также используется квантификатор, позволяющий обнаруживать совпадения с разными вариантами слов в строке. Например, регулярному выражению соответствуют такие слова, как arachnophobia и hexakosioihexekontahexaphobia, а так как квантификатор `‹*›` допускает нулевое число повторений, ему также будет соответствовать само слово phobia. Если необходимо, чтобы перед окончанием «phobia» присутствовал хотя бы один символ, следует заменить квантификатор `‹*›` на `‹+›`.

Steve, Steven или Stephen

Данное регулярное выражение включает в себя пару особенностей, использованных в предыдущих примерах. Несохраняющая группировка, записываемая как `‹(?:…)›`, ограничивает область действия оператора выбора `‹|›`. Квантификатор `‹?›`, применяемый к первой альтернативе

внутри группы, обеспечивает необязательность предшествующего ему символа ‹n›. Это увеличивает эффективность (и краткость) выражения по сравнению с эквивалентным выражением ‹\bSte(?:ve|ven|phen)\b›. Те же соображения эффективности заставляют поместить строковый литерал ‹Ste› в начало регулярного выражения, а не повторять его трижды, как в выражениях ‹\b(?:Steve|Steven|Stephen)\b› или ‹\bSteve\b|\bSteven\b|\bStephen\b›. Некоторые механизмы возвратов недостаточно интеллектуальны, чтобы заметить, что любой текст, соответствующий этим двум последним выражениям, должен начинаться с последовательности символов Ste. По мере продвижения механизма по испытуемой строке в поисках совпадения он сначала попытается отыскать границу слова и затем убедиться, что следующий символ – это символ S. Если совпадение не найдено, механизм вынужден будет опробовать все возможные альтернативы, присутствующие в регулярном выражении, прежде чем он сможет переместиться к следующей позиции в строке и повторить попытку. Если человек легко может заметить, что это будет напрасной тратой сил (так как все альтернативы в регулярном выражении начинаются с последовательности «Ste»), то механизм даже не подозревает об этом. Если же выражение записано как ‹\bSte(?:ven?|phen)\b›, механизм тут же сообразит, что он не сможет отыскать соответствие в строке, которая не начинается с указанных символов.

Более подробно о работе механизма возвратов рассказывается в рецепте 2.13.

Варианты написания термина «regular expression»

Последний пример в этом рецепте соединяет в себе конструкцию выбора, символьные классы и квантификаторы, чтобы отыскать совпадение со всеми типичными вариантами записи термина «regular expression». Поскольку на первый взгляд это регулярное выражение может показаться немного сложным, разобьем его на составляющие и рассмотрим их по отдельности.

Регулярное выражение, которое приводится ниже, записано в режиме свободного форматирования, который не поддерживается диалектом JavaScript. Поскольку в режиме свободного форматирования пробелы игнорируются, литералы пробелов были экранированы символами обратного слэша:

```
\b             # Проверка совпадения с границей слова.
reg            # Соответствует “reg”.
(?:            # Группировка, но несохраняющая...
  ular\        #   Соответствует “ular “.
  expressions? #   Соответствует “expression” или “expressions”.
 |             #  или...
  ex           #   Соответствует “ex”.
  (?:          #   Группировка, но несохраняющая...
```

```
    ps?        #     Соответствует “p” или “ps”.
   |           #    или...
    e          #     Соответствует “e”.
    [sn]       #     Соответствует одному из символов из множества “sn”.
  )            #   Конец несохраняющей группы.
    ?          #     Повторить предшествующую группу ноль или один раз.
)              # Конец несохраняющей группы.
\b             # Проверка совпадения с границей слова.
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Данный шаблон совпадет с любой из следующих семи строк:

- regular expressions
- regular expression
- regexps
- regexp
- regexes
- regexen
- regex

См. также

Рецепты 5.1, 5.2 и 5.4.

5.4. Поиск любых слов, за исключением некоторых

Задача

Необходимо с помощью регулярного выражения отыскать совпадения с любыми полными словами, за исключением слова `cat`. Слово `Catwoman` и другие подобные слова, которые просто содержат в себе символы «cat», должны совпадать, а слово `cat` – нет.

Решение

Исключить совпадения с нежелательными словами можно с помощью негативной опережающей проверки, которая является ключевым элементом следующего регулярного выражения:

```
\b(?!cat\b)\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Несмотря на то, что с помощью инвертированного символьного класса (записывается как ‹[^···]›) легко можно исключить возможность совпадения с определенными символами, тем не менее, нельзя просто записать ‹[^cat]›, и надеяться, что будут исключены совпадения со словом cat. ‹[^cat]› — это допустимое регулярное выражение, но ему соответствует любой символ, за исключением символов c, a и t. Но, хотя выражение ‹\b[^cat]+\b› могло бы помочь исключить совпадения со словом cat, оно точно так же исключило бы совпадения со словом cup, потому что в этом слове содержится запрещенный символ c. Регулярное выражение ‹\b[^c][^a][^t]\w*› ничуть не лучше, потому что оно будет отвергать любые слова, в которых первым следует символ c, вторым — a или третьим — t. Кроме того, это регулярное выражение не накладывает ограничений на первые три символа слова — оно лишь совпадает со словами, содержащими, по крайней мере, три символа, поскольку ни один из инвертированных символьных классов не является необязательным.

Учитывая все вышесказанное, рассмотрим, как регулярное выражение, представленное в начале этого рецепта, позволяет решить поставленную задачу:

```
\b    # Проверка соответствия границе слова.
(?!   # Проверка, что регулярное выражение, следующее ниже, не найдет
      # совпадения, начиная с этой позиции...
  cat #   Соответствие “cat”.
  \b  #   Проверка соответствия границе слова.
)     # Конец негативной опережающей проверки.
\w+   # Соответствует одному или более символу слова.
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Ключевым элементом этого шаблона является негативная опережающая проверка, которая имеет синтаксис ‹(?!···)›. Негативная опережающая проверка отклоняет возможность совпадения с последовательностью символов cat, которая следует за границей слова, и не запрещает регулярному выражению совпадать с этими же символами, если они не следуют именно в таком порядке, или когда они составляют часть более длинного или более короткого слова. В конце регулярного выражения отсутствует проверка совпадения с границей слова, потому что ее присутствие не повлияет на возможность совпадения с регулярным выражением. Квантификатор ‹+› в подвыражении ‹\w+› повторит метасимвол символа слова максимально возможное число раз, то есть он всегда будет обнаруживать совпадение до ближайшей границы слова.

Если применить это регулярное выражение к испытуемому тексту categorically match any word except cat, оно обнаружит пять совпадений: categorically, match, any, word и except.

Варианты

Поиск слов, не содержащих других слов

Если вместо того, чтобы пытаться отыскать совпадения с любыми словами, кроме слова cat, необходимо отыскать любые слова, которые *не содержат* в себе слово cat, требуется немного иной подход:

```
\b(?:(?!cat)\w)+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В предыдущем разделе этого рецепта метасимвол границы слова в начале регулярного выражения обеспечивал удобную привязку, которая позволила просто выполнить негативную опережающую проверку в начале слова.

Решение, используемое здесь, не столь эффективно; тем не менее, эта конструкция часто используется, чтобы обеспечить возможность совпадения с любым словом, за исключением определенного слова или шаблона. Это достигается за счет повторения группы, содержащей негативную опережающую проверку и единственный метасимвол, обозначающий символ слова. Перед сопоставлением с каждым символом механизм регулярных выражений проверяет отсутствие совпадения со словом cat, начиная с текущей позиции.

В отличие от предыдущего регулярного выражения, в данном выражении требуется присутствие метасимвола границы слова. В противном случае оно могло бы совпасть с первой частью слова перед вхождением в него слова cat.

См. также

Рецепт 2.16, где приводится более полное обсуждение проверок соседних символов (позитивных и негативных, опережающих и ретроспективных проверок).

Рецепты 5.1, 5.5, 5.6 и 5.11.

5.5. Поиск любого слова, за которым не следует указанное слово

Задача

Необходимо отыскать совпадение с любым словом, непосредственно за которым не следует слово cat, игнорируя присутствующие между ними любые пробельные символы, знаки пунктуации и другие символы, не являющиеся символами слова.

Решение

Секретным ингредиентом этого регулярного выражения является негативная опережающая проверка:

```
\b\w+\b(?!\W+cat\b)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецептах 3.7 и 3.14 показано, как это регулярное выражение можно реализовать в программном коде.

Обсуждение

Как и во многих других рецептах этой главы, совпадение с целым словом обеспечивается совместной работой метасимволов границы слова (‹\b›) и символа слова (‹\w›). Более подробное описание этих особенностей можно найти в рецепте 2.6.

Вторая часть этого выражения заключена в конструкцию ‹(?!…)›, которая представляет собой негативную опережающую проверку. Опережающая проверка предписывает механизму регулярных выражений временно пройти вперед по строке, чтобы проверить возможность совпадения текста, находящегося непосредственно за текущей позицией, с шаблоном, находящимся внутри опережающей проверки. При этом любые символы, совпавшие с выражением внутри опережающей проверки, не поглощаются – проверка просто убеждается, что совпадение возможно. Поскольку здесь используется негативная опережающая проверка, результат проверки инвертируется.

Другими словами, если шаблон внутри опережающей проверки может обнаружить совпадение непосредственно за текущей позицией, попытка сопоставления считается неудачной и механизм регулярных выражений продвигается на одну позицию вперед, чтобы предпринять новую попытку сопоставления со всем регулярным выражением, начиная со следующего символа в испытуемой строке. Более подробное обсуждение опережающей (и ретроспективной) проверки можно найти в рецепте 2.16.

Что касается шаблона внутри опережающей проверки: конструкции ‹\W+› соответствует один или более символов, не являющихся символами слова, которые находятся перед словом ‹cat›, а завершающий метасимвол границы слова гарантирует, что совпадение будет обнаружено только со словами, за которыми не следует последовательность символов cat, в виде самостоятельного слова, но допускается, что следующее слово начинается с символов cat.

Следует заметить, что это регулярное выражение будет совпадать даже со словом cat, при условии, что за ним не следует второе слово cat. Если потребуется избежать совпадения со словом cat, следует объединить

это регулярное выражение с регулярным выражением из рецепта 5.4, что в результате даст выражение ‹\b(?!cat\b)\w+\b(?!\W+cat\b)›.

Варианты

Если необходимо обеспечить совпадение только со словами, за которыми следует слово cat (не включая в совпадение само слово cat и предшествующие ему символы, не являющиеся символами слова), замените негативную опережающую проверку на позитивную и радуйтесь жизни:

```
\b\w+\b(?=\W+cat\b)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Рецепт 2.16, где приводится более полное обсуждение проверок соседних символов (позитивных и негативных, опережающих и ретроспективных проверок).

Рецепты 5.4 и 5.6.

5.6. Поиск любого слова, которому не предшествует определенное слово

Задача

Необходимо отыскать любое слово, которое не следует непосредственно за словом cat, игнорируя присутствующие между ними любые символы, не являющиеся символами слова.

Решение

Взгляд назад

Ретроспективная проверка позволяет проверить наличие некоторого текста непосредственно перед текущей позицией. Она предписывает механизму регулярных выражений временно вернуться назад по строке, чтобы проверить возможность совпадения текста, оканчивающегося в позиции, где находится ретроспективная проверка. Более подробное обсуждение ретроспективной проверки можно найти в рецепте 2.16.

Следующие три регулярных выражения используют негативную ретроспективную проверку, которая записывается как ‹(?‹!…)›. К сожалению, диалекты регулярных выражений, рассматриваемые в этой книге, отличаются допустимыми шаблонами, которые можно помещать внутрь ретроспективной проверки. В результате получившиеся

решения работают немного по-разному. Дополнительные подробности можно найти в разделе «Пояснения» на стр. 11, в этом рецепте.

Слова, не предшествующие слову «cat»

```
(?<!\bcat\W+)\b\w+
```

Параметры: нечувствительность к регистру символов
Диалект: .NET

```
(?<!\bcat\W{1,9})\b\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE

```
(?<!\bcat)(?:\W+|^)(\w+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Имитация ретроспективной проверки

Диалекты JavaScript и Ruby 1.8 вообще не поддерживают ретроспективную проверку, хотя и предоставляют возможность опережающей проверки. Однако, благодаря тому, что в данной задаче ретроспективная проверка находится в самом начале регулярного выражения, имеется отличная возможность имитировать ретроспективную проверку, разбив регулярное выражение на две части, как показано в следующем примере на JavaScript:

```
var subject = 'My cat is furry.',
    main_regex = /\b\w+/g,
    lookbehind = /\bcat\W+$/i,
    lookbehind_type = false, // негативная ретроспективная проверка
    matches = [],
    match,
    left_context;

while (match = main_regex.exec(subject)) {
    left_context = subject.substring(0, match.index);

    if (lookbehind_type == lookbehind.test(left_context)) {
        matches.push(match[0]);
    } else {
        main_regex.lastIndex = match.index + 1;
    }
}

// совпадения: ['My','cat','furry']
```

Обсуждение

Фиксированная, конечная и бесконечная длина совпадения с ретроспективной проверкой

В первом регулярном выражении используется негативная ретроспективная проверка ‹(?‹!\bcat\W+)›. Так как квантификатор ‹+›, используемый в ретроспективной проверке, не имеет верхнего предела числа совпавших символов, эта версия будет работать только в диалекте регулярных выражений .NET. Все остальные диалекты, рассматриваемые в этой книге, требуют, чтобы совпадение с шаблоном в ретроспективной проверке имело фиксированную или конечную длину.

Во втором регулярном выражении квантификатор ‹+› в ретроспективной проверке был заменен квантификатором ‹{1,9}›. Поэтому такое регулярное выражение может использоваться в диалектах .NET, Java и PCRE, которые поддерживают возможность совпадения переменной длины с ретроспективной проверкой, если известен заранее верхний предел числа совпадающих символов. В этом случае для отделения слов друг от друга я разрешил совпадение с не более чем девятью символами, не являющимися символами слова. Это допускает возможность появления нескольких пробелов и знаков пунктуации между словами. Если только испытуемый текст не представляет собой нечто необычное, это регулярное выражение будет работать точно так же, как и решение, применимое только в диалекте .NET. Однако даже в диалекте .NET указание верхнего предела повторений для любых квантификаторов внутри ретроспективной проверки наверняка сделает регулярное выражение более эффективным, хотя бы потому, что в данном случае уменьшается число непредвиденных возвратов, которые могут возникнуть внутри ретроспективной проверки.

Третье регулярное выражение было преобразовано так, чтобы обеспечить для ретроспективной проверки возможность просматривать строку фиксированной длины, благодаря чему оно может использоваться с еще большим диапазоном диалектов регулярных выражений. Для этого метасимвол ‹\W›, обозначающий символьный класс, совпадающий с символами, не являющимися символами слова, был вынесен за пределы ретроспективной проверки. Это означает, что символы, не являющиеся символами слова (такие как знаки пунктуации и пробельные символы) и предшествующие искомому слову, станут частью фрагмента, который фактически будет совпадать (и возвращаться) с регулярным выражением. Чтобы упростить возможность игнорировать эту часть каждого совпадения, заключительная последовательность символов слова была заключена в сохраняющую группу. Добавив немного программного кода, можно будет читать не все совпадение целиком, а только значение обратной ссылки 1, что даст тот же результат, что и предыдущие регулярные выражения. Программный код, реализующий работу с обратными ссылками, приводится в рецепте 3.9.

Имитация ретроспективной проверки

Диалект JavaScript не поддерживает ретроспективную проверку, но в примере программного кода на JavaScript показано, как можно имитировать обратную проверку, которая должна выполняться в начале регулярного выражения, с помощью двух регулярных выражений. В этом примере нет ограничений на длину совпадения с (имитируемой) ретроспективной проверкой.

Для начала мы разбили оригинальное регулярное выражение `‹(?‹!\bcat\W+)\b\w+›` на две части: выражение внутри ретроспективной проверки `‹\bcat\W+›` и выражение, следующее за ней (`‹\b\w+›`). Затем добавили метасимвол `‹$›` в конец регулярного выражения, выполняющего ретроспективную проверку. Если регулярное выражение `lookbehind` потребуется применять в режиме «символ крышки и знак доллара соответствуют границам строк» (модификатор `/m`), тогда вместо `‹$›` следует вставить конструкцию `‹$(?!\s)›`, чтобы гарантировать совпадение только с концом испытуемого текста. Переменная `lookbehind_type` указывает, является имитируемая ретроспективная проверка позитивной или негативной: значение `true` обозначает позитивную, а `false` – негативную ретроспективную проверку.

После инициализации всех переменных с помощью методов `main_regex()` и `exec()` выполняется обход испытуемого текста (описание этого процесса приводится в рецепте 3.11). После того как будет найдено соответствие, фрагмент испытуемого текста, предшествующий совпадению, копируется в новую строковую переменную (`left_context`) и производится поиск совпадений с регулярным выражением `lookbehind` в этой строке. Так как в конец регулярного выражения `lookbehind` был добавлен якорь, это гарантирует, что второе совпадение будет находиться непосредственно перед первым. Сравнивая результат выполнения ретроспективной проверки с переменной `lookbehind_type`, можно определить, соответствует ли установленный критерий успешному совпадению.

В заключение выполняется одно из двух возможных действий. Если совпадение признано успешным, совпавший фрагмент добавляется в массив `matches`. Если нет, то позиция, с которой будет продолжен поиск совпадения (для этого используется свойство `main_regex.lastIndex`), смещается на один символ вперед от начальной позиции последнего совпадения объекта `main_regex`, чтобы метод `exec` не начал следующую итерацию от конца текущего совпадения.

Все! Мы закончили!

Этот сложный трюк использует свойство `lastIndex`, значение которого динамически обновляется для регулярных выражений, использующих флаг `/g` («global» – глобальный). Обычно обновление и сброс значения свойства `lastIndex` происходят автоматически. Здесь мы воспользовались им, чтобы взять под свой контроль порядок прохождения испытуемого текста регулярным выражением, перемещая позицию поис-

ка совпадения вперед или назад, по мере необходимости. Эта хитрость помогает имитировать только ретроспективную проверку, находящуюся в начале регулярного выражения. С некоторыми изменениями этот программный код мог бы также использоваться для имитации ретроспективной проверки, находящейся в самом конце регулярного выражения. Однако он не может обеспечить полную поддержку ретроспективной проверки. Из-за необходимости организовать взаимодействие между механизмом ретроспективной проверки и механизмом возвратов такой подход не может использоваться для точной имитации поведения ретроспективной проверки, находящейся в середине регулярного выражения.

Варианты

Если необходимо отыскать совпадение только со словами, которым предшествует слово `cat` (исключив из совпадения само слово `cat` и разделяющие их символы, не являющиеся символами слова), следует заменить негативную ретроспективную проверку позитивной:

```
(?<=\bcat\W+)\b\w+
```

Параметры: нечувствительность к регистру символов
Диалект: .NET

```
(?<=\bcat\W{1,9})\b\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE

```
(?<=\bcat)(?:\W+|^)(\w+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

См. также

Рецепт 2.16, где приводится более полное обсуждение проверок соседних символов (позитивных и негативных, опережающих и ретроспективных проверок).

Рецепты 5.4 и 5.5.

5.7. Поиск близко расположенных слов

Задача

Необходимо с помощью регулярного выражения имитировать NEAR-поиск слов, стоящих поблизости. Для тех, кто не знаком с этим термином, поясню: в некоторых средствах поиска, использующих логические операторы, такие как NOT или OR, имеется также специальный опе-

ратор NEAR. При попытке выполнить поиск по строке «word1 NEAR word2» будут найдены слова word1 и word2, следующие в любом порядке, при условии, что они расположены друг от друга в пределах некоторого расстояния.

Решение

Если требуется просто отыскать два разных слова, можно объединить два регулярных выражения, одно из которых совпадает со словом word1, находящимся перед word2, и второе, совпадающее с искомыми словами, следующими в обратном порядке. Следующее регулярное выражение позволяет отыскивать пары слов, которые могут разделять до пяти других слов:

```
\b(?:word1\W+(?:\w+\W+){0,5}?word2|word2\W+(?:\w+\W+){0,5}?word1)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(?:
  word1                # первое искомое слово
  \W+ (?:\w+\W+){0,5}? # до пяти слов
  word2                # второе искомое слово
 |                     #    или шаблон из тех же слов в обратном порядке...
  word2                # второе искомое слово
  \W+ (?:\w+\W+){0,5}? # до пяти слов
  word1                # первое искомое слово
)\b
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Во втором регулярном выражении использован режим свободного форматирования и добавлены дополнительные символы пробела для повышения удобочитаемости. Во всем остальном эти два регулярных выражения совершенно идентичны. Диалект JavaScript не поддерживает режим свободного форматирования, но другие диалекты предоставляют такую возможность. В рецептах 3.5 и 3.7 показано, как можно добавить эти регулярные выражения в форму поиска или в другой программный код.

Обсуждение

Это регулярное выражение содержит две инвертированные копии одного и того же шаблона, следующие друг за другом, и окружает их границами слова. Первое подвыражение совпадает со словом word1 и следующим за ним словом word2, между которыми могут находиться от нуля до

пяти других слов. Второе подвыражение совпадает с теми же словами word1 и word2, но следующими в обратном порядке.

В обоих подвыражениях присутствует минимальный квантификатор ‹{0,5}?›. Он заставляет регулярное выражение соответствовать минимально возможному числу слов, расположенных между искомыми словами. Если попытаться сопоставить регулярное выражение с испытуемым текстом word1 word2 word2, оно совпадет с фрагментом word1 word2, потому что между первым и последним словами в совпадении присутствует минимальное (ноль) число слов. Если появится необходимость настроить расстояние между искомыми словами, можно заменить 0 и 5 внутри квантификаторов на нужные значения. Например, если заменить их значениями ‹{1,15}?›, это позволит отыскивать требуемые слова, между которыми могут находиться до 15 слов и которые обязательно должны отделяться хотя бы одним словом.

Конструкции сокращенной формы записи символьных классов, которые используются для сопоставления с символами слова и символами, не являющимися символами слова (‹\w› и ‹\W›, соответственно), следуют причудливому определению регулярных выражений относительно того, какие символы считаются символами слова (буквы, цифры и символ подчеркивания).

Варианты

Использование условных конструкций

Нередко одно и то же регулярное выражение можно записать несколькими способами. В этой книге мы старались удерживать баланс между переносимостью, краткостью, эффективностью и некоторыми другими характеристиками. Однако иногда даже далеко не идеальные решения тем не менее могут служить наглядными примерами. В следующих регулярных выражениях демонстрируются альтернативные способы поиска слов, расположенных недалеко друг от друга. Мы не рекомендуем использовать их на практике, так как, хотя они будут совпадать с теми же самыми фрагментами текста, но эту работу будут выполнять немного дольше. Кроме того, они будут работать не во всех диалектах регулярных выражений.

В первом регулярном выражении вместо простого объединения инвертированных шаблонов используется условная конструкция, выясняющая наличие совпадения со словом word1 или word2 в конце регулярного выражения. Условная конструкция проверяет, участвовала ли первая сохраняющая группа в сопоставлении; это означает, что совпадение началось со слова word2:

```
\b(?:word1|(word2))\W+(?:\w+\W+){0,5}?(?(1)word1|word2)\b
```

Параметры: нет
Диалекты: .NET, PCRE, Perl, Python

Следующая версия также использует условную конструкцию, чтобы определить, совпадение с каким словом следует искать в конце, но добавляет еще две особенности регулярных выражений:

```
\b(?:(?<w1>word1)|(?<w2>word2))\W+(?:\w+\W+){0,5}?(?(w2)(?&w1)|(?&w2))\b
```

Параметры: нет
Диалекты: PCRE 7, Perl 5.10

Здесь первые вхождения ‹word1› и ‹word2› заключены в именованные сохраняющие группы, которые записываются как ‹(?<name>…)›. Это позволяет задействовать синтаксис *обращения к подпрограмме* ‹(?&name)›, позволяющий повторно использовать подвыражение, обращаясь к нему по имени. Это далеко не то же самое, что обратная ссылка на именованную группу. Именованная обратная ссылка, такая как ‹\k<name>› (.NET, PCRE 7, Perl 5.10) или ‹(?P=name)› (PCRE 4 и выше, Perl 5.10, Python), позволяет добавить повторное сопоставление с текстом, который уже совпал с именованной сохраняющей группой. Подпрограмма же, такая как ‹(?&name)›, позволяет повторно задействовать фактический шаблон, заключенный в соответствующую сохраняющую группу. Здесь нельзя использовать обратные ссылки, потому что в данном случае они обеспечили бы сопоставление со словами, которые уже совпали. Подпрограммы внутри условной конструкции, находящейся в конце регулярного выражения, обеспечивают сопоставление со словом, совпадения с которым *еще не было*, избавляя от необходимости употреблять искомые слова еще раз. Это означает, что, если потребуется использовать это регулярное выражение для поиска других пар слов, достаточно будет изменить их лишь в одном месте.

Сопоставление с тремя или более словами, расположенными поблизости

Экспоненциальный рост числа перестановок. Сопоставление с двумя словами, расположенными поблизости, является достаточно простой задачей. В конце концов, существует всего два способа их взаимного расположения. Но что, если потребуется отыскать три слова, которые могут следовать в произвольном порядке? В этом случае имеется шесть возможных вариантов их взаимного расположения (рис. 5.2). Количество вариантов взаимного расположения слов равно *n!*, или произведению последовательности целых чисел от 1 до *n* (*n факториал)*. Для четырех слов возможно 24 варианта их расположения. Когда вы доберетесь до 10 слов, число вариантов превысит несколько миллионов. Просто нереально реализовать сопоставление более чем с несколькими словами, расположенными поблизости друг от друга, используя приемы регулярных выражений, обсуждавшиеся до сих пор.

Некрасивое решение. Один из способов решить эту проблему заключается в том, чтобы повторять группу, совпадающую с искомыми словами или любым другим словом (только после того, как будет обнаружено

совпадение с искомым словом), и затем с помощью условных конструкций препятствовать успешному завершению попытки сопоставления, пока не будут найдены все искомые слова. Ниже приводится пример сопоставления с тремя словами, следующими в любом порядке, которые могут разделять до пяти слов:

```
\b(?:(?>(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!))))\w+)\b\W*?){3,8}↵
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, PCRE, Perl

```
Два значения:
    [ 12, 21 ]
    = 2 возможных перестановки
Три значения:
     [ 123, 132,
       213, 231,
       312, 321 ]
    = 6 возможных перестановок
Четыре значения:
     [ 1234, 1243, 1324, 1342, 1423, 1432,
       2134, 2143, 2314, 2341, 2413, 2432,
       3124, 3142, 3214, 3241, 3412, 3421,
       4123, 4132, 4213, 4231, 4312, 4321 ]
    = 24 возможных перестановки
Факториалы:
    2! = 2 × 1                                  =       2
    3! = 3 × 2 × 1                              =       6
    4! = 4 × 3 × 2 × 1                          =      24
    5! = 5 × 4 × 3 × 2 × 1                      =     120
    ...
    10! = 10 × 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1 = 3628800
```

Рис. 5.2. *Количество возможных перестановок*

Ниже приводится еще одно регулярное выражение, в котором стандартная несохраняющая группировка была заменена атомарной группировкой (рецепт 2.14) с целью обеспечить поддержку диалекта Python:

```
\b(?:(?:(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!))))\w+)\b\W*?){3,8}↵
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, PCRE, Perl, Python

Квантификаторы ‹{3,8}›, используемые в приведенных выше регулярных выражениях, просто учитывают три обязательных слова и тем са-

мым допускают наличие от нуля до пяти слов между ними. Пустые негативные опережающие проверки, такие как ‹(?!)›, никогда не будут обнаруживать совпадения и используются, чтобы блокировать дальнейшее сопоставление, пока не будет найдено совпадение хотя бы с одним обязательным словом. Логика управления сопоставлением реализована на базе двух наборов вложенных условных конструкций. Первый набор, используя конструкцию ‹\w+›, препятствует совпадению с любым из прежних слов, пока не будет найдено совпадение хотя бы с одним обязательным словом. Второй набор условных конструкций, в конце выражения, вынуждает механизм регулярных выражений производить возврат или терпеть неудачу, если не было найдено совпадение со всеми обязательными словами.

Мы привели краткий обзор того, как это все работает, но, вместо того чтобы дальше погружаться в детали и выяснять, как можно добавить дополнительные обязательные слова, рассмотрим улучшенную реализацию, которая обеспечивает поддержку большего числа диалектов регулярных выражений и использует некоторые хитрости.

Использование пустых обратных ссылок. Некрасивое решение вполне работоспособно, но оно способно победить в конкурсе самых малопонятных регулярных выражений на звание самого неудобочитаемого и сложного в сопровождении. Оно ухудшится еще больше, если попытаться добавить в него дополнительные обязательные слова.

К счастью, регулярные выражения предоставляют одну зацепку, которую можно использовать, чтобы упростить выражение, и при этом она добавляет поддержку диалектов Java и Ruby (ни один из которых не поддерживает условные конструкции).

Прием, описываемый в этом разделе, должен с особой осторожностью использоваться в окончательных версиях приложений. Мы используем здесь особенности поведения регулярных выражений, недокументированные в большинстве библиотек.

```
\b(?:(?>word1()|word2()|word3()|(?>\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|(?:\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

С помощью этой конструкции легко можно добавлять обязательные слова. Ниже приводится пример, позволяющий находить четыре обязательных слова, следующие в произвольном порядке, которые могут разделять до пяти слов:

```
\b(?:(?>word1()|word2()|word3()|word4()|↵
(?>\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|word4()|↵
(?:\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Эти регулярные выражения специально используют пустые сохраняющие группы после каждого обязательного слова. Так как попытка сопоставления с обратной ссылкой, такой как ‹\1›, будет терпеть неудачу, если соответствующая ей сохраняющая группа еще не участвовала в сопоставлении, то обратные ссылки на пустые сохраняющие группы могут использоваться для управления процессом перемещения механизма регулярных выражений по шаблону, как это делают условные конструкции в регулярных выражениях, продемонстрированных выше. Если соответствующая группа уже участвовала в сопоставлении, то, когда механизм регулярных выражений достигнет обратной ссылки на эту группу, он просто обнаружит совпадение с пустой строкой и двинется дальше.

Группировка ‹(?›\1|\2|\3)› в данном примере препятствует совпадению слова с ‹\w+›, пока не будет найдено хотя бы одно совпадение с обязательным словом. В конце регулярного выражения обратные ссылки повторяются еще раз, чтобы воспрепятствовать окончательному совпадению, пока не будут найдены все три обязательных слова.

Диалект Python не поддерживает атомарную группировку, поэтому в примерах, где Python перечислен в списке поддерживаемых диалектов, атомарные группировки заменены обычными несохраняющими группами. Хотя это приводит к снижению эффективности регулярного выражения, но не влияет на результат сопоставления. Самая внешняя группировка не может быть атомарной ни в одном из диалектов, потому что для выполнения поставленной задачи механизму регулярных выражений требуется иметь возможность выполнять возвраты внутри внешней группы, если обратные ссылки в конце регулярного выражения терпят неудачу при сопоставлении.

В JavaScript используются свои правила для работы с обратными ссылками. Хотя диалект JavaScript поддерживает все синтаксические конструкции, присутствующие в версии этого шаблона для Python, он имеет два ограничения, препятствующие возможности использовать эту хитрость, как в других диалектах. Первая проблема состоит в том, что обратная ссылка на сохраняющую группу будет обнаруживать совпадение, даже если она еще не участвовала в сопоставлении. Спецификация на JavaScript требует, чтобы такие обратные ссылки соответ-

ствовали пустой строке или, говоря другими словами, они всегда обнаруживают совпадение. Практически во всех других диалектах все происходит с точностью до наоборот: они никогда не обнаруживают совпадение, в результате чего механизм регулярных выражений вынужден производить возвраты, либо пока не потерпит неудачу совпадение со всем регулярным выражением, либо пока группы, на которые они ссылаются, не примут участие в сопоставлении и не обеспечат возможность совпадения для обратных ссылок.

Вторая проблема в диалекте JavaScript связана со значениями, которые запоминаются сохраняющими группами, вложенными в повторяющуюся внешнюю группу, например ‹((a)|(b))+›. В большинстве диалектов регулярных выражений значениями, запоминаемыми сохраняющими группами, находящимися внутри повторяющейся внешней группы, являются совпадения, обнаруженные в последней попытке сопоставления. Так, если при попытке сопоставления выражение ‹(?:(a)|(b))+› обнаружило совпадение с фрагментом ab, значением первой обратной ссылки будет символ a. Однако согласно спецификации на JavaScript, значения обратных ссылок на вложенные группы должны сбрасываться всякий раз, когда выполняется очередное повторение внешней группы. По этой причине выражение ‹(?:(a)|(b))+› также будет совпадать с фрагментом ab, но по окончании сопоставления обратная ссылка 1 будет ссылаться на сохраняющую группу, не участвовавшую в сопоставлении, и в JavaScript будет совпадать с пустой строкой в пределах самого регулярного выражения и возвращать значение `undefined`, например в массиве, возвращаемом методом `RegExp.prototype.exec()`.

Любого из этих отличий в поведении диалекта JavaScript достаточно, чтобы избегать возможности имитировать условные конструкции с помощью пустых сохраняющих групп, как было описано выше.

Множество слов на любом расстоянии друг от друга

Если необходимо просто проверить наличие некоторых слов в испытуемом тексте, не принимая во внимание их близость, можно воспользоваться опережающей проверкой, чтобы реализовать поиск в одной операции.

Во многих случаях гораздо проще и эффективнее выполнять несколько операций поиска, по одной для каждого искомого слова, просто следя за тем, чтобы все проверки дали положительный результат.

```
\A(?=.*?\bword1\b)(?=.*?\bword2\b).*\Z
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?=[\s\S]*?\bword1\b)(?=[\s\S]*?\bword2\b)[\s\S]*$
```

Параметры: нечувствительность к регистру символов (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Эти регулярные выражения совпадают со всем испытуемым текстом, если внутри него присутствуют все искомые слова. В противном случае никаких совпадений обнаружено не будет. Программисты на JavaScript не смогут использовать первую версию, потому что JavaScript не поддерживает якорные метасимволы ‹\A› и ‹\Z› и режим «точке соответствуют границы строк».

Реализовать эти регулярные выражения можно, следуя примеру в рецепте 3.6. Перед его использованием необходимо заменить ‹word1› и ‹word2› словами, которые требуется отыскать. Если необходимо проверить наличие большего числа слов, можно добавить в начало регулярного выражения столько опережающих проверок, сколько потребуется. Например, выражение ‹\A(?=.*?\bword1\b)(?=.*?\bword2\b)(?=.*?\bword3\b).*\Z› отыскивает три слова.

См. также

Рецепты 5.5 и 5.6.

5.8. Поиск повторяющихся слов

Задача

Вы редактируете некоторый документ, и вам потребовалось проверить наличие ошибочно повторяющихся слов. При поиске не должны учитываться различия в регистре символов, такие как «The the». Кроме того, требуется обеспечить возможность разделения слов разным числом пробелов, даже если при этом слова будут переноситься на другую строку.

Решение

Обратная ссылка соответствует тому, что уже совпало раньше, благодаря чему она является ключевым ингредиентом данного рецепта:

```
\b([A-Z]+)\s+\1\b
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если предполагается использовать это выражение, чтобы сохранить первое слово и удалить все последующие дубликаты, следует заменить все совпадения с обратной ссылкой 1. Другой подход заключает-

ся в том, чтобы выделить совпадения, окружив их другими символами (например, тегами HTML), чтобы при последующем осмотре их легко можно было идентифицировать. В рецепте 3.15 показано, как использовать обратные ссылки в замещающем тексте, который потребуется для реализации любого из этих подходов.

Если требуется просто отыскать повторяющиеся слова, чтобы самому проверить, действительно ли это ошибка, можно воспользоваться рецептом 3.7, где приводится программный код, необходимый для этого. Текстовый редактор или инструмент, подобный grep, подобный тем, что перечислены в разделе «Инструменты для работы с регулярными выражениями» главы 1, помогут вам отыскать повторяющиеся слова, а доступность окружающего текста позволит определить, действительно ли повторение слов является ошибкой, которую следует исправить.

Обсуждение

Чтобы обнаружить повторное совпадение с тем, что уже совпало ранее, необходимы сохраняющая группа и обратная ссылка. Нужно просто поместить в сохраняющую группу то, что может совпадать более одного раза, и затем добавить сопоставление с обратной ссылкой. Этот прием работает иначе, чем простое повторение элемента или группы с помощью квантификатора. Рассмотрим различия между простыми регулярными выражениями ‹(\w)\1› и ‹\w{2}›. В первом выражении для обнаружения повторного совпадения с тем же самым символом слова используются сохраняющая группа и обратная ссылка, тогда как во втором выражении использование квантификатора обеспечивает совпадение с любыми двумя символами слова. Подробнее об обратных ссылках рассказывается в рецепте 2.10.

Вернемся к нашей задаче. Решение, представленное в этом рецепте, отыскивает повторяющиеся слова, состоящие только из символов от A до Z и от a до z (поскольку включен режим нечувствительности к регистру символов). Чтобы обеспечить возможность поиска повторяющихся слов, содержащих символы из других алфавитов, можно использовать свойство Letter Юникода (‹\p{L}›), если диалект регулярных выражений поддерживает такую возможность (раздел «Свойства или категории Юникода» на стр. 72).

Элементу ‹\s+›, расположенному между сохраняющей группой и обратной ссылкой, соответствует любое число пробельных символов, таких как пробел, табуляция или разрыв строки. Если требуется ограничить набор символов, которые могут разделять повторяющиеся слова, горизонтальными пробельными символами (то есть исключить из совпадения разрывы строк), нужно заменить ‹\s+› на ‹[^\S\r\n]›. Это позволит избежать совпадения с повторяющимися словами, расположенными в разных строках текста. Диалекты PCRE 7 и Perl 5.10 включают метасимвол ‹\h›, представляющий сокращенную форму записи символь-

ного класса, который предпочтительнее в данной ситуации, так как он специально создан, чтобы соответствовать горизонтальному пробельному символу.

Наконец, границы слова в начале и в конце регулярного выражения гарантируют, что совпадения не будут обнаруживаться в середине других слов, например «this thistle».

Следует заметить, что повторяющиеся слова в тексте не всегда являются ошибкой, поэтому простое удаление их без визуального просмотра может оказаться неправильным. Например, в разговорном английском языке допустимыми считаются такие конструкции, как «that that» или «had had». Омонимы, названия, звукоподражательные слова (такие как «oink oink» или «ha ha») и некоторые другие конструкции иногда образуются повторяющимися словами. Поэтому в большинстве случаев необходимо визуально исследовать каждое совпадение.

См. также

Рецепт 2.10, где подробно рассматриваются обратные ссылки.

Рецепт 5.9, где показано, как отыскивать повторяющиеся строки.

5.9. Удаление повторяющихся строк

Задача

Имеется некоторый файл журнала, результат выполнения запроса к базе данных или какой-нибудь другой файл или текст с повторяющимися строками. Необходимо с помощью текстового редактора или похожего инструмента удалить все дубликаты, оставив единственную строку из каждой группы повторений.

Решение

Существуют различные программы (включая утилиту `uniq` командной строки в системе UNIX и команду `Get-Unique` оболочки PowerShell в системе Windows), способные помочь удалить дубликаты строк, встречающиеся в файле или в тексте. В следующем разделе будут представлены три варианта решения задачи, основанные на использовании регулярных выражений, которые могут быть особенно полезны при попытке решить эту задачу в текстовом редакторе, обеспечивающем поддержку регулярных выражений и операции поиска с заменой.

При программировании второе и третье решения использовать не следует, поскольку они неэффективны в сравнении с другими доступными средствами, такими как использование хешей[1] для хранения уникальных строк. Однако первое решение (которое требует, чтобы стро-

[1] Ассоциативные массивы. – *Прим. перев.*

ки были отсортированы заранее, что позволит удалить не только рядом стоящие дубликаты) может быть вполне применимо, так как отличается простотой и высокой скоростью работы.

Решение 1: сортировка строк и удаление смежных дубликатов

Если есть возможность отсортировать строки в файле или в тексте, чтобы все повторяющиеся строки оказались следующими друг за другом, ее необходимо использовать, при условии, что не требуется сохранить первоначальный порядок следования строк. Это позволит использовать для удаления дубликатов более простую и эффективную по сравнению с другими операцию поиска с заменой.

Чтобы удалить дубликаты строк после сортировки, можно использовать следующее регулярное выражение и замещающий текст:

```
^(.*)(?:(?:\r?\n|\r)\1)+$
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Диалекты замещающего текста: Python, Ruby

Для совпадения с двумя или более дубликатами строк, следующими друг за другом, в этом регулярном выражении используются сохраняющая группа и обратная ссылка (помимо других ингредиентов). В замещающем тексте используется обратная ссылка, которая вставляет первую строку обратно в текст. В рецепте 3.15 приводится пример программного кода, который можно использовать для реализации этого решения.

Решение 2: оставить в несортированном файле последнее вхождение каждой повторяющейся строки

В случае использования текстового редактора, в котором отсутствует возможность сортировать строки, или если важно сохранить оригинальный порядок следования строк, удалить дубликаты поможет следующее решение, даже если повторяющиеся строки разделены другими строками:

```
^([^\r\n]*)(?:\r?\n|\r)(?=.*^\1$)
```

Параметры: точке соответствуют границы строк, символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Ниже приводится то же самое решение для диалекта JavaScript, которое не требует включения режима «точке соответствуют границы строк»:

```
^(.*)(?:\r?\n|\r)(?=[\s\S]*^\1$)
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

(*Пустая строка, то есть ничего.*)

Диалекты замещающего текста: неприменимо в данном случае

Решение 3: оставить в несортированном файле первое вхождение каждой повторяющейся строки

Если необходимо сохранить первое вхождение из каждой группы повторяющихся строк, необходимо использовать иной подход. Сначала приведем регулярное выражение и замещающий текст, которые мы будем использовать:

```
^([^\r\n]*)$(.*?)(?:(?:\r?\n|\r)\1$)+
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Чтобы преодолеть несовместимость этого регулярного выражения с диалектом JavaScript, из-за отсутствия в нем поддержки режима «точке соответствуют границы строк», необходимо внести пару изменений.

```
^(.*)$([\s\S]*?)(?:(?:\r?\n|\r)\1$)+
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
$1$2
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1\2
```

Диалекты замещающего текста: Python, Ruby

В отличие от регулярных выражений в решениях 1 и 2, эта версия не может удалить все дубликаты строк в ходе одной операции поиска с заменой. Вам потребуется постоянно повторять операцию «заменить все», пока регулярное выражение не перестанет обнаруживать совпадения, то есть пока в тексте не останется повторяющихся строк. Дополнительные подробности приводятся в разделе «Пояснения» в этом рецепте.

Обсуждение

Решение 1: сортировка строк и удаление смежных дубликатов

Это регулярное выражение удаляет все повторяющиеся строки, следующие друг за другом, за исключением первой. Оно не удаляет дубликаты, если между ними присутствуют другие строки. Рассмотрим порядок работы выражения поближе.

Во-первых, символ крышки (‹^›) в начале регулярного выражения совпадает с началом строки. Обычно он совпадает только с началом всего испытуемого текста, поэтому необходимо включить режим «символам ^ и $ соответствуют границы строк» (в рецепте 3.4 показано, как устанавливать различные режимы). Затем конструкция ‹.*› внутри сохраняющих круглых скобок совпадает со всем содержимым одной строки (даже если строка пустая), а значение сохраняется как обратная ссылка 1. Чтобы не возникало ошибок, необходимо, чтобы режим «точке соответствуют границы строки» был выключен, в противном случае точка со звездочкой совпадут со всем текстом.

Внутри внешней несохраняющей группы мы использовали конструкцию ‹(?:\r?\n|\r)›, которой соответствуют разделители строк в текстовых файлах, используемые в Windows (‹\r\n›), в UNIX/Linux/OS X (‹\n›) или устаревшей версии Mac OS (‹\r›). Затем обратная ссылка ‹\1› пытается найти совпадение со строкой, сопоставление с которой только что было завершено. Если в этой позиции не будет найдена та же самая строка, попытка сопоставления с регулярным выражением терпит неудачу и механизм регулярных выражений перемещается в следующую позицию. Если совпадение обнаруживается, квантификатор ‹+› выполняет повтор группы (составленной из последовательности разрыва строки и обратной ссылки 1), чтобы попытаться найти совпадение с дополнительными дубликатами строк.

Наконец, мы использовали знак доллара в конце регулярного выражения, чтобы проверить совпадение с концом строки в тексте. Это гарантирует совпадение только с идентичными строками, но не со строками, начинающимися с тех же символов, которые составляют содержимое предыдущей строки.

Поскольку выполняется операция поиска с заменой, из текста удаляется каждое совпавшее вхождение (включая оригинальную строку и раз-

делители строк). Замещающий текст содержит обратную ссылку 1, которая вставляет оригинальную строку обратно в текст.

Решение 2: оставить в несортированном файле последнее вхождение каждой повторяющейся строки

Это решение несколько отличается от решения 1, которое приводится выше в этом рецепте и которое отыскивает дубликаты строк, только если они следуют непосредственно друг за другом. Во-первых, в версии для диалектов, отличных от JavaScript, в данном решении точка внутри сохраняющей группы заменена конструкцией `‹[^\r\n]›` (совпадает с любым символом, кроме разрыва строки) и был включен режим «точке соответствуют границы строк». Сделано это потому, что далее в регулярном выражении точка используется, чтобы обеспечить совпадение с любым символом, включая разрывы строк. Во-вторых, для поиска дубликатов строки, которые могут встретиться далее в испытуемом тексте, была добавлена опережающая проверка. Поскольку опережающая проверка не поглощает символы, регулярное выражение всегда будет совпадать с единственной строкой (вместе с завершающим ее разрывом строки), которая повторно встречается ниже в испытуемом тексте. Замена всех совпадений пустыми строками удалит все дубликаты строк, при этом нетронутыми останутся только последние вхождения.

Решение 3: оставить в несортированном файле первое вхождение каждой повторяющейся строки

Поскольку ретроспективная проверка поддерживается не так широко, как опережающая (а там, где поддерживается, она может оказаться не в состоянии заглядывать назад настолько далеко, как это необходимо), регулярное выражение в решении 3 существенно отличается от регулярного выражения в решении 2. Вместо поиска строк, которые уже встречались выше (что можно было бы сравнить с тактикой решения 2), это регулярное выражение совпадает со строкой, с первым дубликатом этой строки, который встречается ниже в тексте, и всеми строками между ними. Оригинальная строка сохраняется как обратная ссылка 1, а строки между дубликатами (если таковые вообще имеются) – как обратная ссылка 2. Операция замещения совпадения обеими обратными ссылками 1 и 2 возвращает в текст те части совпадения, которые требуется сохранить, и отбрасывает строку-дубликат и предшествующий ей разрыв строки.

Этот альтернативный подход таит в себе две проблемы. Во-первых, так как каждое совпадение с дубликатами строк может включать строки, находящиеся между ними, вполне вероятно, что среди них окажутся другие строки, которые также могут повторяться, но будут пропущены в процессе выполнения операции замены. Во-вторых, если одна и та же строка встречается в тексте более двух раз, то регулярное выражение сначала совпадет с первым и со вторым дубликатами, но все

последующие дубликаты будут восприниматься им как другие наборы дубликатов, которые соответствуют регулярному выражению по мере продвижения по тексту. Вследствие этого операция замены в лучшем случае удалит только каждое второе вхождение той или иной строки. Чтобы ликвидировать эти проблемы и гарантировать удаление всех дубликатов, необходимо выполнять операцию поиска с заменой для всего испытуемого текста, пока регулярное выражение не перестанет обнаруживать совпадения. Рассмотрим, как будет работать это регулярное выражение, применительно к следующему тексту:

```
value1
value2
value2
value3
value3
value1
value2
```

Для удаления всех дубликатов строк из этого текста потребуется выполнить три прохода. Результаты каждого из проходов приводятся в табл. 5.1.

Таблица 5.1. Результаты операции поиска с заменой после каждого прохода

Проход 1	**Проход 2**	**Проход 3**	**Окончательный текст**
value1	value1	value1	value1
value2	value2	value2	value2
value2	~~value2~~	value3	value3
value3	value3	~~value2~~	
value3	~~value3~~		
~~value1~~	value2		
value2			
Одно совпадение/замена	Два совпадения/замены	Одно совпадение/замена	Дубликатов не осталось

См. также

Рецепт 2.10, где подробно рассматриваются обратные ссылки.

Рецепт 5.8, где показано, как отыскивать повторяющиеся слова.

5.10. Совпадение с полными строками, содержащими определенное слово

Задача

Необходимо обеспечить совпадение с любыми строками, содержащими слово `ninja` в любом месте внутри этих строк.

Решение

```
^.*\bninja\b.*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Часто бывает удобно находить совпадения с целыми строками для их отбора или удаления. Мы начнем наши попытки отыскать совпадение с любой строкой, содержащей слово `ninja`, с выражения `‹\bninja\b›`. Как описывалось в рецепте 2.6, метасимволы границы слова, находящиеся с обоих концов регулярного выражения, гарантируют, что совпадение будет обнаруживаться с последовательностью символов «ninja», только когда она представляет собой отдельное слово.

Чтобы расширить область совпадения с регулярным выражением до полной строки, добавим конструкцию `‹.*›` с обоих концов. Последовательности точки со звездочкой соответствуют ноль или более символов внутри текущей строки. Звездочка является максимальным квантификатором, поэтому она обеспечит совпадение максимально возможной длины. Первой точке со звездочкой будет соответствовать часть строки до последнего вхождения слова «ninja», а второй точке со звездочкой – все остальные символы, не являющиеся разрывами строк, следующие за искомым словом.

Наконец, поместим символ крышки и знак доллара в начало и в конец регулярного выражения, соответственно, чтобы обеспечить совпадение с полной строкой. Строго говоря, якорный метасимвол `‹$›` в конце выражения не нужен, потому что точка и максимальная звездочка всегда обнаруживают совпадение только до конца строки. Однако знак доллара не повредит и сделает регулярное выражение немного более очевидным. Добавление якорных метасимволов в регулярные выражения там, где это уместно, иногда может помочь избежать неожиданностей, поэтому этот прием стоит взять на вооружение. Следует заметить, что в отличие от знака доллара, символ крышки в начале регулярного выражения не всегда бывает избыточен, так как он гарантирует, что регу-

лярное выражение будет совпадать только с полными строками, даже когда по каким-то причинам поиск начинается с середины строки.

Не следует забывать, что три ключевых метасимвола, используемые для ограничения совпадения единственной строкой (якорные метасимволы ‹^› и ‹$›, и точка), не имеют фиксированного значения. Чтобы переориентировать их на работу со строками, следует включить режим, позволяющий метасимволам ‹^› и ‹$› совпадать с границами строк, а режим, когда точке соответствуют границы строк, следует отключить. В рецепте 3.4 показано, как переключать эти режимы в программном коде. В случае использования диалектов JavaScript или Ruby одним режимом, который надо контролировать, становится меньше, потому что в JavaScript отсутствует режим, позволяющий точке совпадать с границами строк, а в Ruby символ крышки и знак доллара всегда совпадают с границами строк.

Варианты

Чтобы отыскать строки, содержащие любое из нескольких слов, можно использовть конструкцию выбора:

```
^.*\b(one|two|three)\b.*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение совпадает с любой строкой, содержащей хотя бы одно из трех слов: «one», «two» или «three». Круглые скобки, окружающие слова, служат двум целям. Во-первых, они ограничивают область действия конструкции выбора и, во-вторых, они сохраняют в обратной ссылке 1 слово, которое фактически было встречено в строке. Если в строке присутствует более чем одно из этих слов, в обратной ссылке будет сохранено слово, стоящее ближе к правому концу строки. Это обусловлено тем, что квантификатор звездочка, расположенный перед круглыми скобками, является максимальным, и будет расширять совпадение с точкой до тех пор, пока это возможно. Если сделать звездочку минимальной, как в выражении ‹^.*?\b(one|two|three)\b.*$›, обратная ссылка 1 будет содержать слово из списка, стоящее ближе к левому концу строки.

Чтобы отыскать строки, которые должны содержать сразу несколько искомых слов, можно использовать опережающую проверку:

```
^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).+$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы обеспечить совпадение со строками, содержащими в себе три обязательных слова, в этом регулярном выражении используется позитивная опережающая проверка. Конструкция ‹.+› в конце обеспечивает фактическое совпадение со строкой после того, как опережающие проверки определят, что строка соответствует требованиям.

См. также

Рецепт 5.11, где показано, как обеспечить совпадение с полными строками, *не* содержащими определенное слово.

5.11. Совпадение с полными строками, не содержащими определенное слово

Задача

Необходимо обеспечить совпадение с полными строками, не содержащими слово `ninja`.

Решение

```
^(?:(?!\bninja\b).)*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)

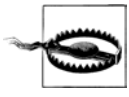
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Чтобы обеспечить совпадение со строкой, *не* содержащей некоторую последовательность символов, можно использовать негативную опережающую проверку (описывается в рецепте 2.16). Обратите внимание, что в этом регулярном выражении негативная опережающая проверка и точка объединены в повторяющуюся несохраняющую группу. Это обеспечивает проверку отсутствия совпадения с подвыражением ‹\bninja\b› в каждой позиции строки. Якорные метасимволы ‹^› и ‹$›, находящиеся по обоим концам регулярного выражения, гарантируют совпадение со всей строкой целиком.

Режимы, применяемые к этому регулярному выражению, определяют, будет ли совпадение включать в себя весь испытуемый текст или только одну строку. В случае, когда включен режим «символам ‹^› и ‹$› соответствуют границы строк» и выключен режим «точке соответствуют границы строк», это выражение действует, как только что было описано, и совпадает с единственной строкой. Если изменить эти два режима

на обратные, регулярное выражение будет совпадать с любым текстом целиком, если в нем отсутствует слово «ninja».



Выполнение негативной опережающей проверки в каждой позиции строки или текста слишком неэффективно. Это решение может использоваться только в случаях, когда регулярное выражение является единственным доступным инструментом, например в приложении, которое нельзя изменить. При создании собственного приложения обратите внимание на рецепт 3.21, где представлено намного более эффективное решение.

См. также

Рецепт 5.10, где показано, как обеспечить совпадение с полными строками, *содержащими* определенное слово.

5.12. Удаление ведущих и завершающих пробельных символов

Задача

Необходимо удалить из строки ведущие и завершающие пробелы.

Решение

Чтобы сделать решение максимально простым и быстрым, лучше всего использовать два регулярных выражения, одно из которых будет удалять ведущие пробельные символы, а другое – завершающие:

```
^\s+
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\s+$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Достаточно просто заменить пустыми строками совпадения, найденные обоими регулярными выражениями. Как это сделать, демонстрируется в рецепте 3.14. В обоих случаях необходимо заменить только первое найденное совпадение, поскольку эти регулярные выражения совпадают со всеми ведущими или завершающими пробельными символами сразу.

Обсуждение

Удаление ведущих и завершающих пробельных символов – это достаточно простая, но часто встречающаяся задача. Каждое из представленных регулярных выражений делится на три части: якорный метасимвол, проверяющий совпадение с началом или концом строки (‹^› и ‹$›, соответственно), символьный класс в сокращенной форме записи, совпадающий с любым пробельным символом (‹\s›), и квантификатор, повторяющий символьный класс один или более раз (‹+›).

Многие языки программирования предоставляют специальную функцию, обычно называемую `trim()` или `strip()`, которая может удалять ведущие и завершающие пробельные символы. В табл. 5.2 показано, как использовать эту встроенную функцию или метод в различных языках программирования.

Таблица 5.2. Стандартные функции для удаления ведущих и завершающих пробельных символов

Язык(и) программирования	Пример использования
C#, VB.NET	`String.Trim([chars])`
Java	`string.trim()`
PHP	`trim($string)`
Python, Ruby	`string.strip()`

В стандартных библиотеках для JavaScript и Perl отсутствует эквивалентная функция, но ее легко можно создать самому:

В Perl:

```
sub trim {
    my $string = shift;
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
```

В JavaScript:

```
function trim (string) {
    return string.replace(/^\s+/, '').replace(/\s+$/, '');
}

// Альтернативный вариант, может использоваться, чтобы добавить
// метод trim ко всем строковым объектам:
String.prototype.trim = function () {
```

```
    return this.replace(/^\s+/, '').replace(/\s+$/, '');
};
```

В обоих языках программирования, Perl и JavaScript, метасимволу ‹\s› соответствует любой символ, определяемый стандартом Юникод как пробельный, в дополнение к символам пробела, табуляции, перевода строки и возврата каретки, которые наиболее часто рассматриваются как пробельные.

Варианты

В действительности существует масса способов записать регулярное выражение, которое поможет усекать строку. Однако при работе с длинными строками (когда производительность имеет самое большое значение) они неизменно оказываются медленнее, чем решение, опирающееся на использование двух простых регулярных выражений. Ниже приводятся некоторые из наиболее типичных альтернативных решений, с которыми можно столкнуться на практике. Все они написаны на JavaScript, а поскольку в JavaScript отсутствует режим «точке соответствуют границы строк», совпадение с любым единственным символом, включая разрывы строк, обеспечивается конструкцией ‹[\s\S]›. В других диалектах вместо этой конструкции можно использовать точку и включать режим «точке соответствуют границы строк».

```
string.replace(/^\s+|\s+$/g, '');
```

Это, пожалуй, наиболее часто используемое решение. Оно объединяет два простых регулярных выражения в одну конструкцию выбора (рецепт 2.8) и использует флаг /g («global» – глобальный), чтобы получить возможность заменить все, а не только первые совпадения (совпадений будет два – с ведущими и завершающими пробельными символами в строке). Это неплохое решение, но при работе с длинными строками оно медленнее решения, опирающегося на использование двух простых регулярных выражений.

```
string.replace(/^\s*([\s\S]*?)\s*$/, '$1')
```

Это регулярное выражение сопоставляется со всей строкой и сохраняет последовательность от первого до последнего непробельного символа (если таковые присутствуют) в обратной ссылке 1. После замены всей строки содержимым обратной ссылки 1 останется усеченная версия строки.

Этот подход концептуально проще, но минимальный квантификатор внутри сохраняющей группы заставляет механизм регулярных выражений выполнить массу дополнительной работы, что обусловливает более низкую скорость работы этого решения на длинных строках. После того как в процессе сопоставления механизм регулярных выражений войдет в сохраняющую группу, минимальный квантификатор

требует, чтобы символьный класс ‹[\s\S]› повторялся минимально возможное число раз. Поэтому при каждом проходе механизм регулярных выражений добавляет в совпадение единственный символ и останавливается, пытаясь сопоставить оставшуюся часть шаблона (‹\s*$›). Если эта попытка терпит неудачу, потому что где-то за текущей позицией в строке присутствует непробельный символ, механизм добавляет к совпадению с классом еще один символ и опять повторяет попытку сопоставления оставшейся части шаблона.

```
string.replace(/^\s*([\s\S]*\S)?\s*$/, '$1')
```

Это решение напоминает предыдущее, но в нем, из соображений производительности, минимальный квантификатор заменен максимальным. Чтобы гарантировать совпадение сохраняющей группы вплоть до последнего непробельного символа, в конец ее был добавлен метасимвол ‹\S›. Однако, из-за того что регулярное выражение должно совпадать со строками, состоящими только из пробельных символов, вся сохраняющая группа сделана необязательной добавлением квантификатора ‹?› после нее.

Вернемся на шаг назад и рассмотрим, как в действительности работает это выражение. Здесь максимальная звездочка в конструкции ‹[\s\S]*› повторяет шаблон «любой символ», пока не будет достигнут конец строки. Затем механизм регулярных выражений выполняет возвраты по одному символу, начиная с конца строки, пока не будет обнаружено совпадение со следующим метасимволом ‹\S› или пока в процессе возвратов не будет достигнут первый символ, совпавший с сохраняющей группой. Если число завершающих пробельных символов не превышает длину текста, совпавшую к данному моменту, то в общем случае это решение оказывается быстрее предыдущего, где используется минимальный квантификатор. Однако и это решение нельзя сравнить по производительности с решением, опирающимся на использование двух простых регулярных выражений.

```
string.replace(/^\s*(\S*(?:\s+\S+)*)\s*$/, '$1')
```

Это довольно часто встречающееся решение, поэтому оно включено в перечень – в качестве предупреждения. У него нет никаких преимуществ, потому что оно медленнее всех остальных решений, показанных выше. Это регулярное выражение похоже на два предыдущих, в том смысле, что оно совпадает со всей строкой и замещает ее той частью, которую требуется сохранить, но, так как внутренняя несохраняющая группа при каждом повторении совпадает с единственным словом, механизму регулярных выражений приходится выполнить множество мелких шагов. Проигрыш этого решения в производительности едва ли будет заметен при обработке коротких строк, но на очень длинных строках, содержащих большое число слов, это регулярное выражение может превратиться в «узкое место».

Некоторые реализации механизмов регулярных выражений содержат оптимизации, изменяющие внутренние процессы сопоставления, описанные здесь, вследствие чего эти решения могут показывать производительность чуть выше или чуть ниже, чем мы предположили. Однако простое решение, опирающееся на использование двух регулярных выражений, неизменно показывает приличную производительность на строках любой длины и с любым содержимым, и поэтому оно является лучшим.

См. также

Рецепт 5.13.

5.13. Замена повторяющихся пробельных символов единственным пробелом

Задача

Продолжая очищать ввод пользователя от лишних символов, требуется заменить повторяющиеся пробельные символы единственным пробелом. Любые символы табуляции, разрыва строки или другие пробельные символы также должны замещаться пробелом.

Решение

Чтобы решить эту задачу, достаточно просто заменить все совпадения со следующими регулярными выражениями единственным пробелом. В рецепте 3.14 показано, как реализовать это решение в программном коде.

Удаление любых пробельных символов

```
\s+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Удаление символов горизонтальной табуляции

```
[•\t]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Довольно часто в процедуре очистки ввода бывает необходимо заменить повторяющиеся пробельные символы единственным пробелом. Напри-

мер, при отображении документов HTML повторяющиеся пробельные символы просто игнорируются (с некоторыми исключениями), поэтому удаление повторяющихся пробельных символов может помочь уменьшить размеры файлов страниц без каких-либо отрицательных последствий.

Удаление любых пробельных символов

В этом решении любые последовательности пробельных символов (разрывы строк, символы табуляции, пробелы и прочие) замещаются единственным символом пробела. Поскольку квантификатор ‹+› повторяет символьный класс, совпадающий с любым пробельным символом (‹\s›), один или более раз, даже единственный символ табуляции, например, будет замещен пробелом. Если заменить квантификатор ‹+› на ‹{2,}›, замещаться будут только последовательности из двух или более пробельных символов. Это может привести к уменьшению числа замен и, как следствие, к увеличению производительности, но в этом случае останутся незамеченными символы табуляции или разрывы строк, которые в противном случае были бы замещены символами пробела. Поэтому какой подход считать лучшим, зависит от желаеиого результата.

Удаление символов горизонтальной табуляции

Это решение действует точно так же, как и предыдущее, за исключением того, что оно оставляет разрывы строк нетронутыми. Замещаются только символы табуляции и пробелы.

См. также

Рецепт 5.12.

5.14. Экранирование метасимволов регулярных выражений

Задача

Необходимо реализовать возможность использовать в регулярных выражениях строковые литералы, получаемые от пользователя или из других источников. Однако, прежде чем встраивать эту строку в регулярное выражение, необходимо экранировать все метасимволы регулярных выражений, чтобы избежать появления непредвиденных эффектов.

Решение

После добавления символа обратного слэша перед каждым символом, который потенциально может иметь специальное назначение в регу-

лярном выражении, можно безопасно использовать получившийся шаблон для поиска литеральной последовательности символов. Все языки программирования, описываемые в этой книге, за исключением JavaScript, имеют встроенные функции или методы, выполняющие эту задачу (перечислены в табл. 5.3). Однако ради полноты обсуждения, мы покажем, как решить эту задачу с помощью собственного регулярного выражения даже в языках, имеющих готовые решения.

Встроенные решения

В табл. 5.3 перечислены встроенные функции, предназначенные для решения этой задачи.

Таблица 5.3. Встроенные функции, выполняющие экранирование метасимволов регулярных выражений

Язык программирования	Функция
C#, VB.NET	`Regex.Escape(str)`
Java	`Pattern.quote(str)`
Perl	`quotemeta(str)`
PHP	`preg_quote(str, [delimiter])`
Python	`re.escape(str)`
Ruby	`Regexp.escape(str)`

Обратите внимание на отсутствие языка JavaScript в этом списке – в нем отсутствует встроенная функция, предназначенная для этой цели.

Регулярное выражение

Несмотря на то, что лучше использовать встроенное решение, если оно существует, эту же задачу можно решить с помощью следующего регулярного выражения и замещающего текста (приводится ниже). При этом необходимо выполнить замещение всех совпадений, а не только первого.

В рецепте 3.15 приводится программный код, производящий замещение совпадений строкой, которая содержит обратную ссылку. Обратная ссылка необходима для того, чтобы вернуть в строку совпавший специальный символ с предшествующим ему символом обратного слэша:

```
[[\]{}()*+?.\\|^$\-,&#\s]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

Следующие строки с замещающим текстом содержат литерал обратного слэша. Строки приводятся без дополнительных символов обратного слэша, которые могут потребоваться для экранирования обратных слэшей, при их использовании в строковых литералах в некоторых языках программирования. Подробнее о диалектах замещающего текста рассказывается в рецепте 2.19.

```
\$&
```

Диалекты замещающего текста: .NET, JavaScript

```
\\$&
```

Диалект замещающего текста: Perl

```
\\$0
```

Диалекты замещающего текста: Java, PHP

```
\\\0
```

Диалекты замещающего текста: PHP, Ruby

```
\\\&
```

Диалект замещающего текста: Ruby

```
\\\g<0>
```

Диалект замещающего текста: Python

Пример функции на JavaScript

Ниже приводится пример, демонстрирующий, как использовать регулярное выражение и строку замещающего текста для создания статического метода `RegExp.escape()` в JavaScript:

```
RegExp.escape = function (str) {
    return str.replace(/[[\]{}()*+?.\\|^$\-,&#\s]/g, "\\$&");
};

// Проверка метода...
var str = "Hello.World?";
var escaped_str = RegExp.escape(str);
alert(escaped_str == "Hello\\.World\\?"); // -> true
```

Обсуждение

Регулярное выражение в этом рецепте помещает все метасимволы внутрь единого символьного класса. Рассмотрим каждый из этих символов и разберемся, почему их следует экранировать. Для одних символов потребность экранировать их не так очевидна, как для других:

`[] {} ()`

Квадратные скобки, <[> и <]>, образуют символьные классы. Фигурные скобки, <{> и <}>, образуют интервальные квантификаторы, а также используются в некоторых специальных конструкциях, таких как свойства Юникода. Круглые скобки, <(> и <)>, используются для группировки, сохранения и в других специальных конструкциях.

`* + ?`

Эти три символа являются квантификаторами, которые повторяют предшествующий им элемент ноль или более, один или более и ноль или один раз, соответственно. Знак вопроса также используется после открывающей круглой скобки для образования специальных видов группировок и других конструкций (то же самое относится и к символу звездочки в Perl 5.10 и PCRE 7).

`. \ |`

Точке соответствует любой символ в испытуемом тексте, обратный слэш экранирует специальные символы или делает литерал символа специальным, а вертикальная черта обеспечивает возможность выбора между несколькими альтернативами.

`^ $`

Символ крышки и знак доллара являются якорными метасимволами, соответствующими началу или концу текста или строки. Кроме того, символ крышки может использоваться для инвертирования символьного класса.

Остальные символы, соответствующие регулярному выражению, обретают специальное назначение только в специальных случаях. Они включены в список из предосторожности.

`-`

Дефис образует диапазон внутри символьного класса. Он экранируется здесь, чтобы избежать непреднамеренного создания диапазонов, когда встраиваемый текст оказывается внутри символьного класса. Следует иметь в виду, что при встраивании текста внутрь символьного класса получившееся регулярное выражение не будет совпадать со встроенной строкой, оно будет совпадать с одним из символов, присутствующих во встраиваемой строке.

,

Запятая используется внутри интервальных квантификаторов, таких как <{1,5}>. Так как большинство диалектов регулярных выражений интерпретируют фигурные скобки как литералы символов, если они не образуют допустимый квантификатор, существует вероятность (хотя и небольшая) образовать квантификатор там, где его не было перед вставкой литерального текста в регулярное выражение, если не экранировать запятые.

&

Символ амперсанда включен в этот перечень по той причине, что в диалекте Java два амперсанда, следующие друг за другом, обозначают операцию пересечения символьных классов. В других языках программирования можно безопасно удалить амперсанд из списка символов, которые требуется экранировать, но не будет ошибкой и оставить его.

и пробельные символы

Символ решетки и пробельные символы (соответствующие классу <\s>) являются метасимволами, только если включен режим свободного форматирования. Эти символы также не будет ошибкой экранировать в любом случае.

Что касается замещающего текста, одна из пяти конструкций («$&», «\&», «$0», «\0» или «\g‹0›») может использоваться для восстановления совпавшего символа вместе с символом обратного слэша. В Perl конструкция $& является переменной, использование которой с любым регулярным выражением приводит к потере его производительности. Если переменная $& используется где-то в другом месте программы на языке Perl, в этом нет ничего особенного, потому что вы уже заплатили полную цену за это. В противном случае лучше будет обернуть все выражение сохраняющей группировкой и вместо $& использовать в замещающем тексте обратную ссылку $1.

Варианты

Как уже объяснялось в разделе «Экранирование блока» в рецепте 2.1 на стр. 50, можно экранировать целый блок текста, использовав конструкцию ‹\Q···\E›. Возможность экранирования блоков поддерживается только в диалектах Java, PCRE и Perl, но даже в этих языках экранирование блока не обеспечивает защиту от ошибок. Чтобы обеспечить полную безопасность, все равно потребуется экранировать все вхождения \E в строке, которую предполагается внедрить в регулярное выражение. В большинстве случаев гораздо проще использовать описанный выше прием, не зависящий от используемого языка программирования, основанный на экранировании всех метасимволов регулярных выражений.

См. также

Рецепт 2.1, где обсуждается, как обеспечить совпадение с литералами символов. Список символов, обязательных для экранирования, приводимый в этом рецепте, немного короче, потому что в нем не учитываются символы, которые может потребоваться экранировать в режиме свободного форматирования или в случае, когда они могут вставляться произвольно, при работе с длинными шаблонами.

6

Числа

Регулярные выражения предназначены для работы с текстом и не способны интерпретировать строки из цифр как числа. Для регулярного выражения `56` – это не число пятьдесят шесть, а строка, содержащая два символа, которые отображаются как цифры 5 и 6. Механизм регулярных выражений знает, что это цифры, потому что им соответствует класс `‹\d›` (рецепт 2.3). Но это все, что он знает. Он не знает, что `56` имеет более глубокий смысл, точно так же, как он не знает, что `:-)` – это не просто три знака пунктуации, соответствующие выражению `‹\p{P}{3}›`.

Однако числа – одна из наиболее важных разновидностей вводимой информации, с которыми приходится сталкиваться на практике, и иногда возникает необходимость обрабатывать их с помощью регулярных выражений, вместо того чтобы передавать программному коду, когда требуется ответить на такой вопрос, как: «Попадает ли число в диапазон от 1 до 100?». Поэтому мы посвятили целую главу вопросам сопоставления регулярных выражений со всеми видами чисел. Сначала мы рассмотрим несколько рецептов, которые могут показаться тривиальными, но на самом деле объясняют важные базовые понятия. В последующих рецептах, где приводятся более сложные регулярные выражения, предполагается, что вы владеете этими основными понятиями.

6.1. Целые числа

Задача

Требуется отыскать различные целые десятичные числа в объемном документе или проверить, представляет ли строковая переменная целое десятичное число.

Решение

Поиск любых положительных целых десятичных чисел в объемном документе:

```
\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли текстовая строка целым положительным десятичным числом:

```
\A[0-9]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск любых отдельных целых положительных десятичных чисел в объемном документе:

```
(?<=^|\s)[0-9]+(?=$|\s)
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Поиск любых отдельных целых положительных десятичных чисел в объемном документе. При этом допускается попадание ведущих пробельных символов в совпадение:

```
(^|\s)([0-9]+)(?=$|\s)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск любых целых десятичных чисел, которым может предшествовать знак плюс или минус:

```
[+-]?\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли текстовая строка целым десятичным числом, которому может предшествовать знак плюс или минус:

```
\A[+-]?[0-9]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[+-]?[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск любого целого десятичного числа с необязательным знаком. При этом допускается наличие пробельного символа между знаком и числом, но в числах без знака ведущие пробелы не допускаются:

```
([+-]•*)?\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Целое число – это непрерывная последовательность из одной или более цифр от нуля до девяти. Это легко можно выразить с помощью символьного класса (рецепт 2.3) и квантификатора (рецепт 2.12): ‹[0-9]+›.

Мы предпочитаем явно указывать диапазон ‹[0-9]› и не использовать сокращенную форму записи ‹\d›. В диалектах .NET и Perl метасимвол ‹\d› соответствует любой цифре из любого алфавита, тогда как класс ‹[0-9]› всегда соответствует одной из 10 цифр в кодировке ASCII. Если заранее известно, что в испытуемом тексте отсутствуют цифры, не входящие в диапазон ASCII, можно сэкономить несколько нажатий на клавиши и использовать ‹\d› вместо ‹[0-9]›.

Если заранее неизвестно, включает ли текст цифры, не входящие в таблицу ASCII, необходимо задуматься над тем, что делать с найденными совпадениями и что ожидает получить пользователь, чтобы определить, использовать ли метасимвол ‹\d› или символьный класс ‹[0-9]›. Если предполагается преобразовывать текст совпадения с регулярным выражением в целое число, нужно проверить, сможет ли функция, имеющаяся в языке программирования и выполняющая преобразование из строки в число, интерпретировать цифры, не являющиеся символами ASCII. Пользователи, создающие документы с использованием своих родных алфавитов, будут ожидать, что ваше программное обеспечение распознает цифры в этих документах.

Кроме того что число является последовательностью цифр, оно также должно быть обособлено. Например, A4 – это размер бумаги, а не число. Существует несколько способов, позволяющих гарантировать, что регулярное выражение будет совпадать только с числами.

Если необходимо проверить, является ли некоторый текст числом, достаточно окружить регулярное выражение якорными символами, совпадающими с началом и с концом текста. Лучше всего для этого выби-

рать метасимволы ‹\A› и ‹\Z›, потому что их смысл не изменяется. К сожалению, диалект JavaScript их не поддерживает. В JavaScript можно использовать метасимволы ‹^› и ‹$›, но при этом не следует указывать флаг /m, который вынуждает символ крышки и знак доллара совпадать с разрывами строк. В Ruby символ крышки и знак доллара всегда совпадают с разрывами строк, поэтому их использование не может вынудить регулярное выражение совпадать со всем текстом.

Когда поиск чисел производится в пределах объемного текстового документа, простое решение могут дать метасимволы границы слова (рецепт 2.6). Если поместить их до или после элемента регулярного выражения, соответствующего цифре, границы слова обеспечат отсутствие символов слова до или после совпавшей цифры. Например, выражение ‹4› совпадет с цифрой 4 в строке A4. Выражение ‹4\b› тоже обнаружит совпадение, потому что в этой строке после цифры 4 отсутствуют какие-либо символы слова. Выражения ‹\b4› и ‹\b4\b› не обнаружат совпадения в строке A4, потому что проверка ‹\b› будет терпеть неудачу при сопоставлении с позицией между символами A и 4. В регулярных выражениях к символам слова относятся буквы, цифры и знак подчеркивания.

Если в регулярное выражение включается сопоставление с символом, не являющимся символом слова, таким как знак плюс или минус, или пробельный символ, следует быть очень осторожным в использовании границ слова. Чтобы найти совпадение с последовательностью +4 и при этом исключить совпадение с последовательностью +4B, вместо выражения ‹\b\+4\b› следует использовать выражение ‹\+4\b›. Первое выражение не будет совпадать с фрагментом +4, потому что в испытуемой строке перед знаком плюс отсутствует символ слова, чтобы удовлетворить границу слова. Выражение ‹\b\+4\b› обнаружит совпадение +4 в строке 3+4, потому что 3 – это символ слова, а + – нет.

Выражение ‹\+4\b› требует наличия единственной границы слова. Первый метасимвол ‹\b› в выражении ‹\+\b4\b› является избыточным. При сопоставлении этого регулярного выражения первый метасимвол ‹\b› всегда будет обнаруживать совпадение с позицией между + и 4 и поэтому никогда ничего не исключит. Первый метасимвол ‹\b› становится необходимым, когда знак плюс является необязательным. Выражение ‹\+?\b4\b› не совпадет с цифрой 4 в тексте A4, а выражение ‹\+?4\b› совпадет.

Границы слова не всегда обеспечивают верное решение. Рассмотрим испытуемый текст $123,456.78. Если попытаться в цикле сопоставить с этой строкой регулярное выражение ‹\b[0-9]+\b›, оно обнаружит совпадения 123, 456 и 78. Знак доллара, запятая и десятичная точка не являются символами слова, поэтому границы слова будут обнаруживать совпадения между цифрами и всеми этими символами. Иногда это именно то, что требуется, а иногда – нет.

Если требуется всего лишь отыскать целые числа, окруженные пробельными символами или началом и концом текста, вместо границ слова следует использовать проверку соседних символов. Проверке ‹(?=$|\s)› соответствует позиция в самом конце текста или перед пробельным символом (разрывы строк также входят в число пробельных символов). Проверке ‹(?‹=^|\s)› соответствует позиция в самом начале текста или после пробельного символа. Метасимвол ‹\s› можно заменить символьным классом, совпадающим с любым желаемым символом, который может появляться перед числом или после него. Порядок работы проверок соседних символов описывается в рецепте 2.16.

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Чтобы проверить, находится ли число в начале строки или предшествует ли ему пробельный символ, вместо ретроспективной проверки можно использовать обычную группировку. Недостаток такого подхода состоит в том, что пробельный символ будет включен в совпадение, если число не находится в начале текста. Эту проблему можно легко решить, если часть регулярного выражения, совпадающую с числом, поместить в сохраняющую группу. Пятое регулярное выражение в разделе «Решение» сохраняет пробельный символ в первой сохраняющей группе, а совпадение с целым числом – во второй.

См. также

Рецепты 2.3 и 2.12.

6.2. Шестнадцатеричные числа

Задача

Требуется отыскать шестнадцатеричные числа в объемном текстовом документе или проверить, является ли значение строковой переменной шестнадцатеричным числом.

Решение

Поиск любых шестнадцатеричных чисел в текстовом документе:

```
\b[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b[0-9A-Fa-f]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка шестнадцатеричным числом:

```
\A[0-9A-F]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9A-F]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск шестнадцатеричных чисел с префиксом `0x`:

```
\b0x[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричных чисел с префиксом `&H`:

```
&H[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричных чисел, оканчивающихся символом `H`:

```
\b[0-9A-F]+H\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения байта, или 8-битного числа:

```
\b[0-9A-F]{2}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения слова, или 16-битного числа:

```
\b[0-9A-F]{4}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения двойного слова, или 32-битного числа:

```
\b[0-9A-F]{8}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения четверного слова, или 64-битного числа:

```
\b[0-9A-F]{16}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки шестнадцатеричных значений байтов (то есть четного числа шестнадцатеричных цифр):

```
\b(?:[0-9A-F]{2})+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

При сопоставлении шестнадцатеричных чисел с регулярными выражениями используются те же приемы, что и при сопоставлении целых десятичных чисел. Единственное отличие состоит в том, что символьный класс, соответствующий одной цифре, на этот раз должен включать буквы от A до F. При этом необходимо решить, должны ли эти символы использоваться в верхнем или в нижнем регистре или допускается смешивание регистров символов. Регулярные выражения, рассматриваемые здесь, допускают смешивание регистров символов.

По умолчанию регулярные выражения различают регистр символов. Классу ‹[0-9a-f]› соответствуют шестнадцатеричные цифры только в нижнем регистре, а классу ‹[0-9A-F]› – только в верхнем регистре. Чтобы обеспечить возможность совпадения без учета регистра символов, можно использовать класс ‹[0-9a-fA-F]› или включить режим нечувствительности к регистру символов. Как это сделать в языках программирования, описываемых в этой книге, рассказывается в рецепте 3.4. Первое регулярное выражение в решениях показано дважды – в них используются разные способы обеспечения нечувствительности к регистру символов. Во всех остальных регулярных выражениях используется только второй способ.

Если необходимо, чтобы в шестнадцатеричных числах были допустимы только символы верхнего регистра, можно использовать приведенные регулярные выражения с выключенным режимом нечувствительности к регистру символов. Чтобы обеспечить допустимость только символов нижнего регистра, следует отключить режим нечувствительности к регистру символов и заменить фрагмент ‹A-F› на ‹a-f›.

Выражение ‹(?:[0-9A-F]{2})+› соответствует четному числу шестнадцатеричных цифр. Выражение ‹[0-9A-F]{2}› соответствует точно двум шестнадцатеричным цифрам. Выражение ‹(?:[0-9A-F]{2})+› повторяет это сопоставление один или более раз. Применение несохраняющей группировки (рецепт 2.9) здесь совершенно необходимо, потому что квантификатор ‹+› должен повторять комбинацию символьного клас-

са и квантификатора ‹{2}›. Выражение ‹[0-9A-F]{2}+› не будет рассматриваться как ошибочное в диалектах Java, PCRE и Perl 5.10, но оно делает совсем не то, что ожидается. Дополнительный символ ‹+› превращает квантификатор ‹{2}› в захватывающий. В данном случае это не дает никакого эффекта, потому что квантификатор ‹{2}› не может повторить предшествующий ему элемент менее двух раз.

В некоторых решениях показано, как обеспечить совпадения с шестнадцатеричными числами, имеющими префикс или окончание, часто используемые для идентификации шестнадцатеричных чисел. Они используются, чтобы обозначить разницу между десятичными и шестнадцатеричными числами, которые могут состоять исключительно из десятичных цифр. Например, 10 может быть десятичным числом между 9 и 11 или шестнадцатеричным числом между F и 11.

В большинстве решений используются метасимволы границы слова (рецепт 2.6). Границы слова можно использовать при поиске чисел в текстовых документах. Следует отметить, что регулярное выражение, допускающее наличие префикса &H, не имеет метасимвола границы слова в начале. Это обусловлено тем, что амперсанд не является символом слова. Если бы в начале регулярного выражения присутствовал метасимвол границы слова, оно отыскивало бы только шестнадцатеричные числа, следующие сразу же за символом слова.

Если необходимо проверить, является ли весь текст шестнадцатеричным числом, достаточно просто окружить регулярное выражение метасимволами, совпадающими с началом и с концом текста. Лучше всего для этого выбирать метасимволы ‹\A› и ‹\Z›, потому что их смысл не изменяется. К сожалению, диалект JavaScript их не поддерживает. В JavaScript можно использовать метасимволы ‹^› и ‹$›, но при этом не следует указывать флаг /m, который вынуждает символ крышки и знак доллара совпадать с разрывами строк. В Ruby символ крышки и знак доллара всегда совпадают с разрывами строк, поэтому их использование не может вынудить регулярное выражение совпадать со всем текстом.

См. также

Рецепты 2.3 и 2.12.

6.3. Двоичные числа

Задача

Требуется отыскать двоичные числа в объемном текстовом документе или проверить, является ли значение строковой переменной двоичным числом.

Решение

Поиск двоичного числа в текстовом документе:

```
\b[01]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка двоичным числом:

```
\A[01]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[01]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного числа, оканчивающегося символом `B`:

```
\b[01]+B\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного значения байта, или 8-битного числа:

```
\b[01]{8}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного значения слова, или 16-битного числа:

```
\b[01]{16}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки значений байтов (то есть двоичных цифр, количество которых кратно 8):

```
\b(?:[01]{8})+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Во всех этих регулярных выражениях используются те же приемы, что и в двух предыдущих рецептах. Основное отличие состоит в том, что теперь каждая цифра может быть `0` или `1`. Это легко можно обеспе-

чить с помощью символьного класса, включающего всего два символа: ‹[01]›.

См. также

Рецепты 2.3 и 2.12.

6.4. Удаление ведущих нулей

Задача

Требуется написать регулярное выражение, совпадающее с целым числом, которое либо возвращает число без ведущих нулей, либо удаляет ведущие нули.

Решение

Регулярное выражение

```
\b0*([1-9][0-9]*|0)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl

```
\1
```

Диалекты замещающего текста: PHP, Python, Ruby

Извлечение чисел в Perl

```
while ($subject =~ m/\b0*([1-9][0-9]*|0)\b/g) {
    push(@list, $1);
}
```

Удаление ведущих нулей в PHP

```
$result = preg_replace('/\b0*([1-9][0-9]*|0)\b/', '$1', $subject);
```

Обсуждение

Для отделения числа от ведущих нулей мы используем сохраняющую группировку. Подвыражение ‹0*› перед группой совпадает с ведущими нулями, если таковые имеются. Конструкции ‹[1-9][0-9]*› внутри группы соответствует число, состоящее из одной или более цифр, при-

чем первая цифра числа не может быть нулем. Число может начинаться с нуля, только если это сам ноль. Границы слова гарантируют, что совпадение будет обнаружено с целым числом, а не с его частью, о чем уже говорилось в рецепте 6.1.

Чтобы получить список всех чисел без ведущих нулей, находящихся в испытуемом тексте, необходимо выполнить обход всех совпадений с регулярным выражением, как описывается в рецепте 3.11. Внутри цикла число следует извлекать из первой (и единственной) сохраняющей группы, как описывается в рецепте 3.9. В этом решении показано, как реализовать эту операцию на языке Perl.

Удаление ведущих нулей легко можно выполнить с помощью операции поиска с заменой. Наше регулярное выражение отделяет число от ведущих нулей с помощью сохраняющей группировки. Если заменить все совпадение с регулярным выражением (число вместе с ведущими нулями) текстом совпадения с первой сохраняющей группой, мы фактически удалим ведущие нули. В решении показано, как можно реализовать эту операцию на языке PHP.

См. также

Рецепты 3.15 и 6.1.

6.5. Числа в определенном диапазоне

Задача

Требуется обеспечить совпадение с целым числом, попадающим в определенный диапазон. При этом необходимо, чтобы регулярное выражение определяло точный диапазон значений, а не ограничивало количество цифр.

Решение

От 1 до 12 (час или месяц)

```
^(1[0-2]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 24 (час):

```
^(2[0-4]|1[0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 31 (день месяца):

```
^(3[01]|[12][0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 53 (неделя года):

```
^(5[0-3]|[1-4][0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 59 (минута или секунда):

```
^[1-5]?[0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 100 (процент):

```
^(100|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 100:

```
^(100|[1-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 32 до 126 (коды отображаемых символов ASCII):

```
^(12[0-6]|1[01][0-9]|[4-9][0-9]|3[2-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 127 (неотрицательное значение байта со знаком):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От –128 до 127 (значение байта со знаком):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9]|-(12[0-8]|1[01][0-9]|[1-9]?[0-9]))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 255 (значение байта без знака):

```
^(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 366 (день года):

```
^(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|[1-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1900 до 2099 (год):

```
^(19|20)[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 32767 (неотрицательное значение слова со знаком):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От –32768 до 32767 (значение слова со знаком):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]|-(3276[0-8]|327[0-5][0-9]|32[0-6][0-9]{2}|↵
3[01][0-9]{3}|[12][0-9]{4}|[1-9][0-9]{1,3}|[0-9]))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 65535 (значение слова без знака):

```
^(6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В предыдущих рецептах выполнялось сопоставление с целыми числами, состоящими из произвольного или определенного количества цифр. Они допускали появление любых цифр в любом разряде числа. Это очень простые регулярные выражения.

Сопоставление с числами в определенном диапазоне, например с числами от 0 до 255, – не такая простая задача для регулярных выражений. Нельзя написать выражение ‹[0-255]›. Нет, вы, конечно, можете записать такое выражение, но оно не будет совпадать с числами в диапазоне от 0 до 255. Это будет символьный класс, эквивалентный классу ‹[0125]›, соответствующий единственному символу, который является цифрой 0, 1, 2 или 5.

Так как эти регулярные выражения получились немного длинными, во всех решениях используются якорные метасимволы, чтобы сделать выражения пригодными для проверки, является ли вся строка, например полученная от пользователя, допустимым числом. В рецепте 6.1 объясняется, как вместо якорных метасимволов можно использовать границы слова и проверки соседних символов в случае применения регулярных выражений для других целей. В обсуждении регулярные выражения будут приводиться без якорных метасимволов и основное внимание будет уделяться работе с диапазонами. Если у вас появится желание использовать одно из этих регулярных выражений, вы сможете добавить якорные метасимволы или границы слова, чтобы обеспечить совпадение регулярного выражения с полным числом, а не с его частью.

Регулярные выражения сопоставляются с текстом символ за символом. Если необходимо получить совпадение с числом, состоящим более чем из одной цифры, придется предусмотреть все возможные комбинации цифр. Основными стандартными строительными блоками таких выражений являются символьные классы (рецепт 2.3) и конструкции выбора (рецепт 2.8).

В символьных классах можно использовать диапазоны для одной цифры, такие как ‹[0-5]›. Это возможно благодаря тому, что цифры от 0 до 9 занимают непрерывную область в таблицах символов ASCII и Юникод. Классу ‹[0-5]› соответствует одна из шести цифр, точно так же, как классам ‹[j-o]› и ‹[\x09-\x0E]› соответствуют различные диапазоны из шести символов.

Когда число из определенного диапазона присутствует в тексте, оно занимает некоторое число следующих друг за другом позиций. Каждую позицию занимает некоторая цифра из определенного диапазона. Числа из одних диапазонов, например от 12 до 24, занимают фиксированное количество позиций. Числа из других диапазонов, например от 1 до 12, могут занимать переменное количество позиций. Диапазон цифр, допустимых в каждой позиции, может зависеть или не зависеть от цифр, стоящих в других позициях. В диапазоне чисел от 40 до 59 позиции не зависят друг от друга. В диапазоне чисел от 44 до 55 позиции уже являются взаимозависимыми.

Проще всего представить те диапазоны, числа из которых занимают фиксированное число независимых позиций, такие как от 40 до 59. Чтобы записать такие числа в виде регулярного выражения, достаточно просто объединить несколько символьных классов. При этом нужно использовать один символьный класс для каждой позиции, указывая в них допустимые диапазоны цифр в той или иной позиции.

```
[45][0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Для представления чисел из диапазона от 40 до 59 требуются две цифры. Поэтому нам потребовались два символьных класса. Первая цифра может быть 4 или 5. Символьный класс ‹[45]› соответствует любой из этих цифр. Вторая цифра может быть любой из 10 цифр. Совпадение с этой цифрой обеспечивается классом ‹[0-9]›.

Вместо класса ‹[0-9]› можно было бы использовать его сокращенную форму записи ‹\d›. Мы используем явный диапазон ‹[0-9]›, чтобы сохранить единство стиля записи символьных классов и поддержать удобочитаемость. Уменьшение числа символов обратного слэша в регулярных выражениях также очень полезно при работе с такими языками программирования, как Java, которые требуют экранировать обратные слэши в строковых литералах.

Числа в диапазоне от 44 до 55 также занимают две позиции, но они не являются взаимонезависимыми. Первая цифра может быть 4 или 5. Если первая цифра 4, вторая цифра должна попадать в диапазон от 4 до 9, охватывая числа от 44 до 49. Если первая 5, вторая цифра должна попадать в диапазон от 0 до 5, охватывая числа от 50 до 55. В регулярном выражении можно просто объединить эти два диапазона чисел с помощью конструкции выбора:

```
4[4-9]|5[0-5]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

С помощью конструкции выбора мы сообщили механизму регулярных выражений, что требуется найти совпадение с ‹4[4-9]› или ‹5[0-5]›. Среди всех операций регулярных выражений оператор выбора имеет наименьший приоритет, благодаря чему не требуется группировать цифры, как в выражении ‹(4[4-9])|(5[0-5)›.

При необходимости в конструкции выбора можно объединить столько диапазонов, сколько потребуется. Числа в диапазоне от 34 до 65 также занимают две взаимозависимые позиции. Первая цифра может быть любой цифрой в диапазоне от 3 до 6. Если первая цифра 3, вторая долж-

на попадать в диапазон от 4 до 9. Если первая цифра 4 или 5, вторая цифра может быть любой. Если первая цифра 6, вторая должна попадать в диапазон от 0 до 5:

```
3[4-9]|[45][0-9]|6[0-5]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точно так же, как мы использовали конструкцию выбора, чтобы разбить диапазон с взаимозависимыми позициями на несколько диапазонов с взаимонезависимыми позициями, мы можем использовать конструкцию выбора, чтобы разбить диапазон чисел с переменным числом позиций на несколько диапазонов с фиксированным числом позиций. Числа в диапазоне от 1 до 12 могут занимать одну или две позиции. Выделим из него диапазон от 1 до 9, занимающий одну позицию, и диапазон от 10 до 12, занимающий две позиции. Позиции в каждом из этих двух диапазонов являются взаимонезависимыми, поэтому нам не потребуется разбивать их в свою очередь:

```
1[0-2]|[1-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Мы поместили диапазон чисел с двумя цифрами перед диапазоном с одной цифрой. Это было сделано преднамеренно, потому что механизм регулярных выражений *нетерпелив* по своей природе. Он выполняет просмотр выражения слева направо и останавливается, как только обнаруживает первое же совпадение. Если представить, что имеется испытуемый текст `12`, выражение ‹`1[0-2]|[1-9]`› обнаружит совпадение 12, тогда как выражение ‹`[1-9]|1[0-2]`› совпадет только с цифрой ‹1›. Первой будет опробована альтернатива ‹`[1-9]`›. Поскольку она благополучно совпадет с 1, механизм регулярных выражений не будет даже пытаться проверить альтернативу ‹`1[0-2]`›, которая может обеспечить «лучшее» решение.

Диапазон от 85 до 117 включает числа, которые могут иметь две разные длины. Числа в диапазоне от 85 до 99 занимают две позиции, а числа в диапазоне от 100 до 117 – три. Позиции в этих диапазонах являются взаимозависимыми, поэтому необходимо произвести дополнительное деление. В случае диапазона, занимающего две позиции: если первая цифра 8, вторая должна попадать в диапазон от 5 до 9. Если первая цифра 9, вторая может быть любой цифрой. В случае диапазона, занимающего три позиции: первая цифра может быть только 1. Если во второй позиции находится цифра 0, тогда в третьей позиции может находиться любая цифра. Но если вторая цифра 1, тогда третья должна попадать в диапазон от 0 до 7. В результате мы получаем четыре диапазона: от 85 до 89, от 90 до 99, от 100 до 109 и от 110 до 117. Несмотря на то,

Некоторые механизмы регулярных выражений не являются нетерпеливыми

POSIX-совместимые механизмы регулярных выражений и механизмы ДКА не следуют этому правилу. Они опробуют все альтернативы и возвращают самое длинное совпадение. Однако все диалекты, рассматриваемые в этой книге, являются механизмами НКА, которые не выполняют дополнительную работу, требуемую стандартом POSIX. Все они сообщат, что выражение ‹[1-9]|1[0-2]› совпадает с 1 в тексте 12.

На практике список альтернатив обычно окружается якорными метасимволами или границами слова. В этом случае порядок следования альтернатив не имеет никакого значения. Оба выражения, ‹^([1-9]|1[0-2])$› и ‹^(1[0-2]|[1-9])$›, в любом из диалектов, описываемых в этой книге, обнаружат совпадение с 12 в тексте 12, так же как POSIX-совместимые механизмы регулярных выражений и механизмы ДКА. Якорные метасимволы требуют, чтобы регулярное выражение совпало либо со всей строкой целиком, либо не совпало вообще. Определения механизмов ДКА и НКА приводятся во врезке «История появления термина «регулярное выражение»», на стр. 20 в главе 1.

что количество диапазонов увеличилось, регулярное выражение остается таким же прямолинейным, как и предыдущие:

```
8[5-9]|9[0-9]|10[0-9]|11[0-7]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы обеспечить совпадение регулярного выражения с диапазонами чисел, необходимо просто выполнять деление на диапазоны, пока все диапазоны не будут иметь фиксированное число взаимонезависимых позиций. Если следовать этому правилу, всегда будут получаться корректные регулярные выражения, которые легко читать и сопровождать, даже если они будут иметь внушительную длину.

Существует несколько приемов, которые позволят уменьшить длину регулярных выражений. Например, регулярное выражение, совпадающее с числами из диапазона от 0 до 65535, с помощью предыдущего правила можно записать как:

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9][0-9]|6[0-4][0-9][0-9][0-9]|↵
[1-5][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9]|[1-9][0-9][0-9]|↵
[1-9][0-9]|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение прекрасно справляется со своей задачей, и невозможно придумать регулярное выражение, которое выполнялось бы быстрее. Любые оптимизации, какие можно было бы применить (например, в выражении присутствует несколько альтернатив, начинающихся с 6), будут выполнены механизмом регулярных выражений в процессе компиляции выражения. Нет никакой необходимости тратить свое время впустую, чтобы усложнить выражение в надежде сделать его быстрее. Однако есть возможность сделать регулярное выражение короче, чтобы сэкономить время на вводе с клавиатуры и при этом сохранить удобочитаемость.

В некоторых альтернативах присутствуют идентичные символьные классы, следующие друг за другом. Существует возможность убрать дубликаты, воспользовавшись квантификаторами. Полная информация о квантификаторах приводится в рецепте 2.12.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часть ‹[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]› регулярного выражения содержит три очень похожих альтернативы и во всех них присутствует одна и та же пара символьных классов. Единственное различие заключается в количестве повторений второго класса. Мы легко можем объединить их в конструкцию ‹[1-9][0-9]{1,3}›.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любые последующие ухищрения только ухудшат удобочитаемость. Например, можно было бы отделить ведущую цифру 6 от первых четырех альтернатив:

```
6(?:553[0-5]|55[0-2][0-9]|5[0-4][0-9]{2}|[0-4][0-9]{3})|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Но в действительности это регулярное выражение получилось на один символ длиннее, потому что в него пришлось добавить несохраняющую группировку, чтобы отделить цифру 6 от других альтернатив. Этот прием не дает преимуществ в скорости ни в одном из диалектов регуляр-

ных выражений, рассматриваемых в этой книге. Все они выполняют эту оптимизацию внутренними средствами.

См. также

Рецепты 2.8, 4.12 и 6.1.

6.6. Шестнадцатеричные числа в определенном диапазоне

Задача

Требуется обеспечить совпадение с шестнадцатеричным числом, попадающим в определенный диапазон. При этом необходимо, чтобы регулярное выражение определяло точный диапазон значений, а не ограничивало количество цифр.

Решение

От 1 до C (от 1 до 12: час или месяц):

```
^[1-9a-c]$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 18 (от 1 до 24: час):

```
^(1[0-8]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 1F (от 1 до 31: день месяца):

```
^(1[0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 35 (от 1 до 53: неделя года):

```
^(3[0-5]|[12][0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 3B (от 0 до 59: минута или секунда):

```
^(3[0-9a-b]|[12]?[0-9a-f])$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 64 (от 0 до 100: процент):

```
^(6[0-4]|[1-5]?[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 64 (от 1 до 100):

```
^(6[0-4]|[1-5][0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 20 до 7E (от 32 до 126: коды отображаемых символов ASCII):

```
^(7[0-9a-e]|[2-6][0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 7F (от 0 до 127: 7-битное число):

```
^[1-7]?[0-9a-f]$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до FF (от 0 до 255: 8-битное число):

```
^[1-9a-f]?[0-9a-f]$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 16E (от 1 до 366: день года):

```
^(16[0-9a-e]|1[0-5][0-9a-f]|[1-9a-f][0-9a-f]?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 76C до 833 (от 1900 до 2099: год):

```
^(83[0-3]|8[0-2][0-9a-f]|7[7-9a-f][0-9a-f]|76[c-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 7FFF: (от 0 до 32767: 15-битное число):

```
^([1-7][0-9a-f]{3}|[1-9a-f][0-9a-f]{1,2}|[0-9a-f])$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до FFFF: (от 0 до 65535: 16-битное число):

```
^([1-9a-f][0-9a-f]{1,3}|[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Между сопоставлением с диапазонами десятичных чисел и шестнадцатеричных чисел нет никакой разницы. Как уже говорилось в предыдущем разделе, необходимо просто выполнять деление на диапазоны, пока каждый из получившихся диапазонов не будет иметь фиксированное число взаимонезависимых позиций. После это останется только выбрать символьный класс для каждой позиции и объединить диапазоны с помощью конструкции выбора.

Так как буквы и цифры занимают отдельные области в таблицах символов ASCII и Юникод, невозможно использовать такой класс, как `‹[0-F]›`, для сопоставления с любой из шестнадцатеричных и десятичных цифр. В действительности этот символьный класс будет совпадать с шестнадцатеричными и десятичными цифрами, но он также будет совпадать со знаками пунктуации, расположенными в таблице ASCII между цифрами и буквами. Поэтому следует использовать символьный класс с двумя диапазонами: `‹[0-9A-F]›`.

Другая проблема связана с чувствительностью к регистру символов. По умолчанию регулярные выражения различают символы разных регистров. Классу `‹[0-9A-F]›` соответствуют только символы верхнего регистра, а классу `‹[0-9a-f]›` — только нижнего. Класс `‹[0-9A-Fa-f]›` обеспечивает совпадение с символами любого регистра.

Явно указывать диапазоны символов верхнего и нижнего регистров в каждом классе быстро может стать утомительным занятием. Гораздо проще включить режим нечувствительности к регистру символов. Как это сделать в том или ином языке программирования, рассказывается в рецепте 3.4.

См. также

Рецепты 2.8 и 6.2.

6.7. Вещественные числа

Требуется обеспечить совпадение с вещественным числом и иметь возможность определять, являются ли знак, целая, дробная и экспоненциальная части числа обязательными, необязательными или недопусти-

мыми. Здесь не требуется, чтобы регулярное выражение ограничивалось определенным диапазоном значений, потому что эту проверку лучше реализовать в программном коде, как описывалось в рецепте 3.12.

Решение

Обязательны: знак, целая, дробная и экспоненциальная части:

```
^[-+][0-9]+\.[0-9]+[eE][-+]?[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обязательны: знак, целая и дробная части, экспоненциальная часть недопустима:

```
^[-+][0-9]+\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Знак необязателен, целая и дробная части обязательны, экспоненциальная часть недопустима:

```
^[-+]?[0-9]+\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны: знак и целая часть, дробная часть обязательна, экспоненциальная часть недопустима:

```
^[-+]?[0-9]*\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка также должна быть опущена. Экспоненциальная часть недопустима.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка становится необязательной. Экспоненциальная часть недопустима.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка также должна быть опущена. Экспоненциальная часть необязательна.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка становится необязательной. Экспоненциальная часть необязательна.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Предыдущее регулярное выражение, исправленное для поиска числа в текстовом документе:

```
[-+]?(\b[0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+\b)?
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Все регулярные выражения заключаются в якорные метасимволы (рецепт 2.5), чтобы обеспечить проверку, является ли весь испытуемый текст вещественным числом, в противовес поиску вещественного числа в текстовом документе. Если потребуется выполнить поиск вещественного числа в объемном тексте, можно использовать границы слова или проверки соседних символов, как описывается в рецепте 6.1.

Решения, не допускающие наличия необязательных частей, достаточно прямолинейны: они просто перечисляют требуемые соответствия слева направо. Символьные классы (рецепт 2.3) совпадают со знаком, цифрами и символом `e`. Квантификаторы `‹+›` и `‹?›` (рецепт 2.12) допускают появление любого количества цифр и необязательный знак экспоненты.

Сделать необязательными знак и целую часть числа достаточно просто. Знак вопроса, следующий за символьным классом с символами знака, делает его необязательным. Замена квантификатора `‹+›` на `‹*›` обеспечит возможность повторения цифр целой части числа ноль или более раз вместо одного или более раз.

Сложности начинают возникать, когда необходимо сделать необязательными знак, целую и дробную части числа. Хотя все они сами по себе и являются необязательными, тем не менее, они не могут быть необязательными одновременно, так как пустая строка не является до-

пустимым вещественным числом. Простое решение ‹[-+]?[0-9]*\.?[0-9]*› будет совпадать со всеми допустимыми вещественными числами, но оно также будет совпадать и с пустой строкой, а так как мы опустили якорные метасимволы, это регулярное выражение будет совпадать с пустыми строками между любыми двумя символами в испытуемом тексте. Если это регулярное выражение применить в операции поиска с заменой к тексту 123abc456, используя замещающий текст «{$&}», в результате будет получена строка {123}{}a{}b{}c{456}{}. Регулярное выражение корректно совпадет с фрагментами 123 и 456, но оно также обнаружит совпадения нулевой длины во всех остальных попытках.

Создавая регулярное выражение, когда каждый элемент сам по себе может быть необязательным, очень важно учесть, могут ли другие элементы оставаться необязательными, если один из них фактически будет опущен. Вещественное число должно содержать хотя бы одну цифру.

В решениях к этому рецепту подробно поясняется, что, когда целая или дробная часть является необязательной, другая становится обязательной. Кроме того, отдельно поясняется, какие решения интерпретируют последовательность 123. как вещественное число, а какие – как целое число, за которым следует точка, не являющаяся частью этого числа. Например, в языках программирования завершающая точка может рассматриваться как оператор конкатенации или как первая точка в операторе диапазона, состоящем из двух точек.

Чтобы удовлетворить требование, не допускающее ситуации, когда целая и дробная части могут быть опущены одновременно, для реализации двух ситуаций мы использовали конструкцию выбора (рецепт 2.8) внутри группировки (рецепт 2.9). Выражение ‹[0-9]+(\.[0-9]+)?› соответствует ситуации с обязательной целой частью и необязательной дробной. Выражение ‹\.[0-9]+› соответствует одной дробной части.

Объединение ‹[0-9]+(\.[0-9]+)?|\.[0-9]+› охватывает все три возможные ситуации. Первая альтернатива совпадает с числами, содержащими целую и дробную части, а также с числами без дробной части. Вторая альтернатива совпадает только с дробной частью. Поскольку оператор выбора имеет самый низкий приоритет, мы заключили обе альтернативы в группу, после чего получили возможность добавлять эту конструкцию в более длинные регулярные выражения.

Выражение ‹[0-9]+(\.[0-9]+)?|\.[0-9]+› требует, чтобы десятичная точка была опущена в случае отсутствия дробной части. Если точка может присутствовать в числе даже в отсутствие дробной части, можно использовать выражение ‹[0-9]+(\.[0-9]*)?|\.[0-9]+›. Первая альтернатива в этом регулярном выражении по-прежнему объединена с квантификатором ‹?›, который обеспечивает ее необязательность. Различие заключается в том, что теперь сами цифры дробной части объявлены необязательными. Мы заменили квантификатор ‹+› (один или более) на ‹*› (ноль или более). В результате первая альтернатива этого регулярного

выражения совпадает с целой и необязательной дробной частью, причем дробная часть может быть представлена десятичной точкой с цифрами или одной десятичной точкой. Вторая альтернатива осталась без изменений.

Этот последний пример интересен тем, что изменение одного из требований привело к изменению квантификатора в регулярном выражении. Изменилось требование к точке, которая теперь стала необязательной сама по себе, а не в комбинации с цифрами дробной части. Чтобы удовлетворить изменившемуся требованию, мы поменяли квантификатор у символьного класса, соответствующего цифрам дробной части. Такое решение стало возможным благодаря тому, что точка и символьный класс уже были сгруппированы, что делает их необязательными одновременно.

См. также

Рецепты 2.3, 2.8, 2.9 и 2.12.

6.8. Числа с разделителями групп разрядов

Задача

Требуется обеспечить совпадение с числами, в которых запятая используется для разделения групп разрядов, а точка – для отделения дробной части.

Решение

Обязательные целая и дробная части:

```
^[0-9]{1,3}(,[0-9]{3})*\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обязательная целая часть и необязательная дробная. Если дробная часть опущена, десятичная точка также должна быть опущена.

```
^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательная целая часть и необязательная дробная. Если дробная часть опущена, десятичная точка также должна быть опущена.

```
^([0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?|\.[0-9]+)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Предыдущее регулярное выражение, исправленное для поиска числа в текстовом документе:

```
\b[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?\b|\.[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Так как все эти регулярные выражения предназначены, чтобы обеспечить совпадение с вещественными числами, в них используются те же приемы, что и в предыдущем рецепте. Единственное отличие состоит в том, что теперь для сопоставления с целой частью вместо выражения `‹[0-9]+›` используется выражение `‹[0-9]{1,3}(,[0-9]{3})*›`. Это регулярное выражение совпадает с последовательностью от 1 до 3 цифр, за которой следует ноль или более групп, состоящих из запятой и трех цифр.

Мы не можем использовать выражение `‹[0-9]{0,3}(,[0-9]{3})*›`, чтобы обеспечить необязательность целой части, потому что оно совпадает с числами, начинающимися с запятой, например ,123. Это та же ловушка со всеми необязательными элементами, которая описывалась в предыдущем рецепте. Чтобы сделать целую часть необязательной, мы не изменяем элемент регулярного выражения, соответствующий целой части числа, а всю целую часть целиком делаем необязательной. Последние два регулярных выражения в решении делают это с помощью конструкции выбора. Регулярное выражение для необязательной целой части и обязательной дробной объединено с регулярным выражением, совпадающим с дробной частью, при отсутствующей целой части. В результате было получено регулярное выражение, в котором и целая, и дробная части являются необязательными, но не одновременно.

См. также

Рецепты 2.3, 2.9 и 2.12.

6.9. Римские числа

Задача

Необходимо обеспечить совпадение с римскими числами, такими как IV, XIII и MVIII.

Решение

Римские числа без проверки на корректность:

```
^[MDCLXVI]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Современные римские числа, строгое соответствие:

```
^(?=[MDCLXVI])M*(C[MD]|D?C{0,3})(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Современные римские числа, гибкое соответствие:

```
^(?=[MDCLXVI])M*(C[MD]|D?C*)(X[CL]|L?X*)(I[XV]|V?I*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простые римские числа:

```
^(?=[MDCLXVI])M*D?C{0,4}L?X{0,4}V?I{0,4}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Римские числа записываются с помощью символов M, D, C, L, X, V и I, представляющих значения 1000, 500, 100, 50, 10, 5 и 1 соответственно. Первому регулярному выражению соответствуют любые строки, состоящие из этих символов, без проверки порядка следования и количества символов, допустимых в римских числах.

В наше время (имеются в виду последние несколько сотен лет) запись римских чисел регулируется сводом строгих правил. Эти правила обеспечивают точное положение римских цифр в числах. Например, 4 всегда записывается как IV, и никогда как IIII.

Второе регулярное выражение в решении соответствует только римским числам, которые следуют этим правилам.

Каждая ненулевая цифра в десятичном представлении в римской форме записи записывается отдельно. Число 1999 записывается как MCMXCIX, где M – это 1000, CM – это 900, XC – это 90 и IX – это 9.

Тысячи записываются просто: по одному символу M на каждую тысячу, что легко можно выразить как `‹M*›`.

Сотни могут записываться 10 разными способами, совпадения с которыми мы обеспечили с помощью конструкций выбора. Выражение `‹C[MD]›` соответствует формам записи CM и CD, которые представляют числа 900 и 400. `‹D?C{0,3}›` соответствует формам записи DCCC, DCC, DC, D, CCC, CC, C и пустой строке, представляющим 800, 700, 600, 500, 300, 200, 100 и «ничто». Это дает нам все 10 цифр в разряде сотен.

Совпадение с десятками обеспечивает выражение ‹X[CL]|L?X{0,3}›, а с единицами – выражение ‹I[XV]|V?I{0,3}›. В них используется тот же синтаксис, но используются другие символы.

Все четыре части регулярного выражения могут быть необязательными, потому что в любом из разрядов может быть ноль. У римлян не было символа или слова, представляющего ноль. Поэтому ноль в римских числах не записывается. Несмотря на то, что каждая часть римского числа сама по себе может быть необязательной, тем не менее, все они не могут быть необязательными одновременно. Мы должны гарантировать невозможность совпадений нулевой длины с регулярным выражением. Для этого мы добавили в начало регулярного выражения опережающую проверку ‹(?=[MDCLXVI])›. Эта опережающая проверка, как описывается в рецепте 2.16, проверяет возможность совпадения хотя бы с одним символом. Опережающая проверка не поглощает символы, поэтому совпадения с ней могут повторно совпасть с остальной частью регулярного выражения.

Третье регулярное выражение немного гибче. Оно допускает наличие таких цифр, как IIII, и при этом принимает IV.

Четвертое регулярное выражение соответствует римским числам, которые записаны без применения операции вычитания, из-за чего все символы должны быть записаны в порядке убывания их значений. Число 4 должно записываться как IIII, а не как IV. Сами римляне записывали числа именно таким способом.

Все регулярные выражения заключены в якорные метасимволы (рецепт 2.5), чтобы обеспечить проверку, является ли весь испытуемый текст римским числом, в противовес поиску римского числа в текстовом документе. Если потребуется выполнить поиск римского числа в объемном тексте, можно заменить ‹^› и ‹$› на ‹\b›.

Преобразование римских чисел в десятичные

Следующая функция на языке Perl использует «строгое» регулярное выражение из этого рецепта, чтобы убедиться, что полученная ею строка является римским числом. Затем она использует регулярное выражение ‹[MDLV]|C[MD]?|X[CL]?|I[XV]?›, чтобы обойти в цикле все цифры в числе и сложить их значения:

```
sub roman2decimal {
    my $roman = shift;
    if ($roman =~
        m/^(?=[MDCLXVI])
          (M*)                  # 1000
          (C[MD]|D?C{0,3})      # 100
          (X[CL]|L?X{0,3})      # 10
```

```
            (I[XV]|V?I{0,3})     # 1
            $/ix)
        {
            # Найдено римское число
            my %r2d = ('I' => 1, 'IV' => 4, 'V' => 5, 'IX' => 9,
                       'X' => 10, 'XL' => 40, 'L' => 50, 'XC' => 90,
                       'C' => 100, 'CD' => 400, 'D' => 500, 'CM' => 900,
                       'M' => 1000);
            my $decimal = 0;
            while ($roman =~ m/[MDLV]|C[MD]?|X[CL]?|I[XV]?/ig) {
                $decimal += $r2d{uc($&)};
            }
            return $decimal;
        } else {
            # Это не римское число
            return 0;
        }
    }
```

См. также

Рецепты 2.3, 2.8, 2.9, 2.12, 2.16, 3.9 и 3.11.

7

URL, пути и адреса в Интернете

Помимо чисел, которые рассматривались в предыдущей главе, еще одной важной темой, которая касается широкого круга программ, является поиск совпадений с различными путями и адресами, которые используются для поиска данных:

- Адреса URL, URN и связанные с ними строки
- Доменные имена
- IP-адреса
- Имена файлов и папок в операционной системе Windows

Формат записи адресов URL оказался настолько гибким и удобным, что был принят для обозначения адресов самых разных ресурсов, которые вообще не имеют отношения к Всемирной паутине (World Wide Web). По этой причине приемы синтаксического анализа текста с помощью регулярных выражений, которые приводятся в этой главе, будут весьма полезны в самых разных ситуациях.

7.1. Проверка адресов URL

Задача

Необходимо проверить, является ли заданный фрагмент текста адресом URL, допустимым в некоторой ситуации.

Решение

Допускает практически любые адреса URL:

```
^(https?|ftp|file)://.+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(https?|ftp|file)://.+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Требует доменное имя и не допускает наличия имени пользователя или пароля:

```
\A                        # Якорь
(https?|ftp)://           # Схема
[a-z0-9-]+(\.[a-z0-9-]+)+ # Домен
([/?].*)?                 # Путь и/или параметры
\Z                        # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+↵
([/?].+)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Требует доменное имя и не допускает наличия имени пользователя или пароля. Позволяет опускать схему (http или ftp), если она может быть определена из имени поддомена (www или ftp):

```
\A                          # Якорь
((https?|ftp)://|(www|ftp)\.) # Схема адресации или поддомен
[a-z0-9-]+(\.[a-z0-9-]+)+     # Домен
([/?].*)?                     # Путь и/или параметры
\Z                            # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^((https?|ftp)://|(www|ftp)\.)[a-z0-9-]+(\.[a-z0-9-]+)+([/?].*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Требует доменное имя и путь, ведущий к файлу с изображением. Имя пользователя, пароль или параметры не допускаются:

```
\A                        # Якорь
(https?|ftp)://           # Схема адресации
[a-z0-9-]+(\.[a-z0-9-]+)+ # Домен
(/[\w-]+)*                # Путь
/[\w-]+\.(gif|png|jpg)    # Файл
\Z                        # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+(/[\w-]+)*/[\w-]+\.(gif|png|jpg)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Невозможно создать регулярное выражение, совпадающее со всеми допустимыми адресами URL и не совпадающее с каким-нибудь недопустимым адресом URL. Причина состоит в том, что практически все что угодно может стать допустимым адресом URL в еще не изобретенной схеме адресации.

Проверка адреса URL может быть полезной, только когда известен контекст, в котором этот адрес может быть признан допустимым. В этом случае можно ограничить круг допустимых адресов URL в соответствии со схемами адресации, поддерживаемыми используемым программным обеспечением. Все регулярные выражения в этом рецепте предназначены для проверки URL, используемых веб-броузерами. Эти адреса URL записываются в следующем виде:

```
scheme://user:password@domain.name:80/path/file.ext?param=value&param2↵
=value2#fragment
```

Все эти части на практике являются необязательными. В URL типа `file:` используется только путь. В URL `http:` обязательным является только доменное имя.

Первое регулярное выражение в этом решении проверяет, начинается ли адрес URL с названия одной из схем адресации, наиболее часто используемых веб-броузерами: `http`, `https`, `ftp` и `file`. Якорный метасимвол ‹^› привязывает регулярное выражение к началу текста (рецепт 2.5). Конструкция выбора (рецепт 2.8) используется для представления списка схем. ‹https?› — это интеллектуальный способ представить конструкцию ‹http|https›.

Поскольку первое регулярное выражение допускает достаточно разные схемы адресации, такие как `http` и `file`, оно в действительности не может проверить корректность текста, следующего за именем схемы. Выражение ‹.+$› просто поглощает все, что встретится ему до конца текста, при условии, что текст не содержит символов разрыва строки.

По умолчанию точке (рецепт 2.4) соответствуют все символы, за исключением символов разрыва строки. В этом отношении диалект Ruby является исключением. В Ruby символ крышки и знак доллара всегда совпадают с символами разрыва строки, и поэтому вместо них мы использовали метасимволы ‹\A› и ‹\Z› (рецепт 2.5). Строго говоря, то же самое следовало бы сделать со всеми регулярными выражениями в этом рецепте, чтобы адаптировать их для работы с диалектом Ruby. Следова-

ло бы, – если бы входной текст мог бы содержать несколько строк и требовалось бы избежать совпадения с URL, занимающим одну из строк в тексте.

Следующие два регулярных выражения записаны в режиме свободного форматирования (рецепт 2.18) и в обычном виде. Выражение, записанное в режиме свободного форматирования, проще читать, тогда как обычное регулярное выражение легче вводить. Диалект JavaScript не поддерживает режим свободного форматирования регулярных выражений.

Эти два регулярных выражения принимают только адреса URL для веб и FTP и требуют, чтобы за именем схемы HTTP или FTP следовало нечто, похожее на допустимое доменное имя. Доменное имя должно записываться символами ASCII. Интернационализированные домены (IDN) не принимаются. За доменом может следовать путь или список параметров, отделяемые от домена символом слэша или знаком вопроса. Поскольку знак вопроса находится внутри символьного класса (рецепт 2.3), его не требуется экранировать. В символьных классах знак вопроса интерпретируется как обычный символ, а символ слэша интерпретируется как обычный символ в любом месте регулярного выражения. (Если его и можно встретить в исходных текстах программ экранированным обратным слэшем, то только потому, что в языке Perl и некоторых других символ слэша используется как разделитель в литералах регулярных выражений.)

В выражении не предпринимается попыток проверить корректность пути или параметров. Элемент `‹.*›` просто совпадает со всем подряд, кроме символов разрыва строки. Поскольку и путь, и параметры являются необязательными, подвыражение `‹[/?].*›` заключено в группу, которая сделана необязательной с помощью квантификатора `‹?›` (рецепт 2.12).

Эти регулярные выражения и те, что следуют ниже, не допускают возможности передачи имени пользователя или пароля в строке адреса URL. Размещение информации о пользователе в строке URL считается плохой практикой из соображений безопасности.

Большинство веб-броузеров принимают адреса URL, которые не содержат имени схемы адресации, и правильно определяют ее, исходя из доменного имени. Например, адрес `www.regexbuddy.com` – это сокращенная форма записи адреса `http://www.regexbuddy.com`. Чтобы обеспечить возможность совпадения с такими адресами URL, мы просто расширили в регулярном выражении список допустимых схем, включив в него поддомены `www.` и `ftp.`.

Выражение `‹(https?|ftp)://|(www|ftp)\.›` прекрасно справляется с этим. В этом списке присутствуют две альтернативы, каждая из которых начинается с двух других альтернатив. Первая группа альтернатив обеспечивает совпадение с `‹https?›` и `‹ftp›`, за которыми должна следовать последовательность `‹://›`. Вторая группа альтернатив обеспечивает совпадение с `‹www›` и `‹ftp›`, за которыми должна следовать точка. Вы лег-

ко можете отредактировать оба списка, чтобы изменить названия схем адресации и имена поддоменов.

Последние два регулярных выражения требуют наличия в URL схемы, доменного имени, записанного символами ASCII, пути и имени файла с изображением в формате GIF, PNG или JPEG. Путь и имя файла могут состоять из символов и цифр любого алфавита, а также включать символы подчеркивания и дефисы. Все это, кроме дефиса, включает сокращенная форма записи символьного класса `‹\w›` (рецепт 2.3).

Какое из этих регулярных выражений следует использовать? Это во многом зависит от того, что вы собираетесь делать. Во многих ситуациях ответом на этот вопрос может быть отказ от использования регулярных выражений вообще. Вместо этого нужно просто попытаться выполнить запрос по проверяемому адресу. Если в ответ будет возвращено допустимое содержимое, этот адрес можно признать корректным. Если будет получена ошибка с кодом 404, адрес можно отвергнуть. В конечном счете, это единственный способ проверить действительность адреса URL.

См. также

Рецепты 2.3, 2.8, 2.9 и 2.12.

7.2. Поиск адресов URL в тексте

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут окружаться такими знаками пунктуации, как круглые скобки, не являющиеся частью адреса.

Решение

Адреса URL без пробелов:

```
\b(https?|ftp|file)://\S+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Адреса URL без пробелов или завершающих знаков пунктуации:

```
\b(https?|ftp|file)://[-A-Z0-9+&@#/%?=~_|$!:,.;]*↵
[A-Z0-9+&@#/%=~_|$]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Адреса URL без пробелов или завершающих знаков пунктуации. Наименование схемы адресации может отсутствовать в адресах, начинающихся с имени поддомена `www` или `ftp`:

```
\b((https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*↵
[A-Z0-9+&@#/%=~_|$]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Пусть имеется текст:

```
Visit http://www.somesite.com/page, where you will find more information.
```

Какая его часть является адресом URL?

Прежде чем указать на фрагмент `http://www.somesite.com/page`, подумайте о следующем: знаки пунктуации и пробелы являются допустимыми символами в адресах URL. Запятые, точки и даже пробелы необязательно должны экранироваться (`%20`). Литералы пробелов вполне допустимы. Некоторые визуальные средства создания веб-страниц позволяют пользователям вставлять пробелы в имена файлов и папок и включают эти пробелы в виде литералов в ссылки на эти файлы.

Это означает, что регулярное выражение, совпадающее со всеми допустимыми адресами URL, в предыдущем тексте должно совпасть с фрагментом:

```
http://www.somesite.com/page, where you will find more information.
```

Вероятность, что человек, напечатавший это предложение, предполагал включить пробелы в адрес URL, весьма невелика, потому что использование неэкранированных пробелов в URL – большая редкость. Первое регулярное выражение в решении исключает их с помощью сокращенной формы записи символьного класса `‹\S›`, который совпадает со всеми символами, не являющимися пробельными. Даже при том, что предусматривается работа регулярного выражения в режиме «нечувствительности к регистру символов», символ `S` должен быть символом верхнего регистра, потому что метасимвол `‹\S›` – это далеко не то же самое, что метасимвол `‹\s›`. В действительности они имеют совершенно противоположные значения. Подробнее о них рассказывается в рецепте 2.3.

Первое регулярное выражение все еще достаточно сырое. Оно включит в совпадение запятую, которая следует за адресом URL в примере. Несмотря на то, что запятые и другие знаки пунктуации не являются чем-то необычным в адресах URL, тем не менее, в конце адреса они используются крайне редко.

В следующем регулярном выражении вместо одного метасимвола `‹\S›` используются два символьных класса. Первый класс включает в себя большее число знаков пунктуации, чем второй. Второй класс содержит знаки пунктуации, которые в соответствии с правилами английского языка с наибольшей вероятностью могут появиться в предложении после адреса URL. Первый класс имеет квантификатор «звездочка» (ре-

цепт 2.12), который обеспечивает возможность совпадения с URL произвольной длины. Второй символьный класс не имеет квантификатора, требуя тем самым, чтобы адрес URL оканчивался одним из символов, перечисленных в этом классе. Символьный класс не содержит алфавитных символов нижнего регистра, потому что предполагается работа в режиме «нечувствительности к регистру символов». Порядок установки режимов в том или ином языке программирования описывается в рецепте 3.4.

Второе регулярное выражение будет неправильно определять адреса URL, в которых некорректно используются знаки пунктуации, совпадая лишь с частью адреса. Но это регулярное выражение решает наиболее типичную проблему, связанную с запятой или точкой, следующей сразу за адресом URL.

Большинство веб-броузеров принимают адреса URL, в которых отсутствует указание схемы адресации, корректно определяя ее, исходя из доменного имени. Например, адрес `www.regexbuddy.com` является сокращенной формой записи адреса `http://www.regexbuddy.com`. Чтобы обеспечить возможность совпадения с такими адресами URL, последнее регулярное выражение расширяет список допустимых схем адресации, включая в него поддомены `www.` и `ftp.`.

Подвыражение `‹(https?|ftp)://|(www|ftp)\.›` прекрасно справляется с этой задачей. В этом списке присутствуют две альтернативы, каждая из которых, в свою очередь, начинается с двух альтернатив. Первая альтернатива совпадает с `‹https?›` и `‹ftp›`, за которыми следует последовательность символов `‹://›`. Вторая альтернатива совпадает с `‹www›` или `‹ftp›`, за которыми следует точка. Список схем и поддоменов, принимаемых регулярным выражением, может быть изменен в соответствии с конкретной задачей.

См. также

Рецепты 2.3 и 2.6.

7.3. Поиск в тексте адресов URL, заключенных в кавычки

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут быть окружены знаками пунктуации, являющимися частью окружающего текста, а не самого адреса URL. Необходимо предоставить пользователю возможность заключать адреса URL в кавычки, чтобы можно было явно показать, когда пробелы и знаки пунктуации являются частью URL.

Решение

```
\b(?:(?:https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*
                                        [-A-Z0-9+&@#/%=~_|$]
|"(?:(?:https?|ftp|file)://|(www|ftp)\.)[^"\r\n]+"
|'(?:(?:https?|ftp|file)://|(www|ftp)\.)[^'\r\n]+'
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов, точке соответствуют границы строк, якорным метасимволам соответствуют границы строк

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В предыдущем рецепте рассматривалась задача поиска адресов URL в тексте, и как отличать знаки препинания в тексте от символов URL. И хотя решение в предыдущем рецепте пригодно для использования в большинстве случаев, тем не менее, ни одно регулярное выражение не может гарантировать абсолютную надежность во всех ситуациях.

Если предполагается, что регулярное выражение будет применяться к создаваемым впоследствии текстовым документам, то можно предоставить пользователям возможность заключать адреса URL в кавычки. Решение, представленное здесь, позволяет заключать адреса URL в одинарные или двойные кавычки. Если адрес URL помещен в кавычки, он должен начинаться с одной из схем: `‹https?|ftp|file›` или с одного из поддоменов `‹www|ftp›`. Вслед за схемой или поддоменом регулярное выражение допускает наличие любых символов, за исключением разрывов строк и кавычек, окружающих URL.

Регулярное выражение делится на три альтернативы. Первая альтернатива – это регулярное выражение из предыдущего рецепта, совпадающее с адресом URL без кавычек и пытающееся отличать знаки пунктуации и символы URL. Вторая альтернатива совпадает с адресом URL в кавычках. Третья альтернатива совпадает с адресом URL в апострофах. Мы использовали две альтернативы вместо одной, – с сохраняющей группой, включающей открывающую кавычку и обратной ссылкой на закрывающую, потому что обратную ссылку нельзя использовать внутри инвертированного символьного класса, исключающего кавычки из URL.

Мы использовали кавычки и апострофы, потому что таким образом обычно оформляются адреса URL в файлах HTML и XHTML. Заключение адресов в кавычки естественно для тех, кто занимается разработкой веб-страниц, но вы легко сможете добавить в регулярное выражение другие пары символов, ограничивающих адреса URL.

См. также

Рецепты 2.8 и 2.9.

7.4. Поиск в тексте адресов URL, заключенных в скобки

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут окружаться знаками пунктуации, являющимися частью окружающего текста, а не самого адреса URL. Необходимо обеспечить корректное совпадение с адресами URL, содержащими пары круглых скобок, при этом не включая в совпадение скобки, окружающие адрес URL.

Решение

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.])*
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)↵
|[-A-Z0-9+&@#/%=~_|$?!:,.])*(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|↵
[A-Z0-9+&@#/%=~_|$])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В адресах URL допустимым считается практически любой символ, включая и круглые скобки. Однако скобки очень редко встречаются в адресах, и именно поэтому мы не включили их в регулярные выражения, которые приводились в предыдущих рецептах. Но на некоторых крупных веб-сайтах скобки все же используются:

```
http://en.wikipedia.org/wiki/PC_Tools_(Central_Point_Software)
http://msdn.microsoft.com/en-us/library/aa752574(VS.85).aspx
```

Одно из решений заключается в том, чтобы потребовать от пользователя заключать такие адреса URL в кавычки. Другое – расширить регулярное выражение так, чтобы оно принимало такие URL. Самое сложное состоит в том, чтобы определить, какая из закрывающих скобок является частью URL, а какая является знаком пунктуации, завершающим URL, как в следующем примере:

```
RegexBuddy's web site (at http://www.regexbuddy.com) is really cool.
```

Так как вполне возможно, что одна из скобок является ограничивающим знаком пунктуации, а другая нет, мы не можем использовать при-

ем заключения адреса в кавычки из предыдущего рецепта. Наиболее простое решение состоит в том, чтобы разрешить появление скобок внутри адресов URL, только когда они не являются вложенными парами открывающих и закрывающих скобок. Адреса URL Википедии и Microsoft отвечают этим требованиям.

Оба регулярных выражения, которые приводятся в решении, делают одно и то же. Первое из них записано в режиме свободного форматирования, чтобы обеспечить более высокую удобочитаемость.

Эти регулярные выражения, по сути, представляют собой последнее регулярное выражение из рецепта 7.2. Они делятся на три части: список схем адресации, за которым следует тело URL, в котором используется квантификатор «звездочка», чтобы обеспечить совпадение с URL любой длины, и завершающий символ URL, для которого не предусмотрен квантификатор (то есть этот символ должен появляться точно один раз). В оригинальном регулярном выражении в рецепте 7.2 и та, и другая часть – тело URL и завершающий символ – состояли из единственного символьного класса.

В решениях для этого рецепта они были заменены двумя более сложными символьными классами. Символьный класс в середине:

```
[-A-Z0-9+&@#/%=~_|$?!:,.]
```

превратился в

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.]
```

Последний символьный класс:

```
[A-Z0-9+&@#/%=~_|$]
```

превратился в

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$]
```

Оба символьных класса заменили конструкции выбора (рецепт 2.8). Поскольку оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений, для объединения альтернатив мы использовали несохраняющую группировку (рецепт 2.9).

В оба символьных класса была добавлена альтернатива ‹`\([-A-Z0-9+&@#/%=~_|$?!:,.]* \)`›, при этом оригинальный класс остался в качестве другой альтернативы. Новая альтернатива совпадает с парами круглых скобок с произвольным числом символов, допустимых в адресах URL, между ними.

Заключительный символьный класс был объединен с той альтернативой, которая позволяет адресам URL оканчиваться текстом в круглых скобках или единственным символом, который наименее вероятно может служить знаком пунктуации.

В результате объединения получилось регулярное выражение, совпадающее с адресами URL, включающими любое число круглых скобок,

а также с адресами URL без круглых скобок и даже с адресами URL, не содержащими ничего, кроме скобок, при условии, что они парные.

В часть регулярного выражения, совпадающего с телом URL, мы добавили квантификатор «звездочка» для всей несохраняющей группы. Это обеспечило возможность появления любого числа пар круглых скобок в URL. Так как теперь вся несохраняющая группа получила квантификатор «звездочка», отпала необходимость в квантификаторе «звездочка» для оригинального символьного класса. На самом деле, звездочку необходимо убрать в обязательном порядке.

Средняя часть регулярного выражения, представленного в решении, имеет вид `‹(ab*c|d)*›`, где `‹a›` и `‹c›` – это литералы круглых скобок, а `‹b›` и `‹d›` – символьные классы. Записать это регулярное выражение как `‹(ab*c|d*)*›` будет ошибкой. На первый взгляд оно может выглядеть логично, потому что допускает совпадение с любым числом символов из класса `‹d›`, но внешний квантификатор `‹*›` уже обеспечивает повторение класса `‹d›`. Если для класса `‹d›` добавить еще одну звездочку, сложность регулярного выражения станет экспоненциальной. Выражение `‹(d*)*›` может совпасть со строкой `dddd` множеством способов. Например, внешняя звездочка может выполнить четыре повторения, а внутренняя звездочка по одному повторению каждый раз. Внешняя звездочка может выполнить три повторения, а внутренняя звездочка – 2-1-1, 1-2-1 или 1-1-2. Внешняя звездочка может выполнить два повторения, а внутренняя звездочка – 2-2, 1-3 или 3-1. Нетрудно понять, что с ростом длины испытуемой строки количество возможных комбинаций увеличивается лавинообразно. Мы называем этот эффект катастрофическим возвратом, этот термин был введен в рецепте 2.15. Эта проблема проявляется, когда регулярное выражение не в состоянии отыскать допустимое совпадение, например, потому что в регулярное выражение было что-то добавлено, чтобы получить возможность находить адреса URL, после или внутри которых содержится что-то специфическое, отвечающее требованиям конкретной задачи.

См. также

Рецепты 2.8 и 2.9.

7.5. Преобразование адресов URL в ссылки

Задача

Имеется текстовый документ, который может содержать один или более адресов URL. Необходимо преобразовать эти адреса в ссылки, добавив в текст HTML-теги `‹a›`, заключив в них найденные URL. Сами адреса URL должны присутствовать в тексте документа как в виде адреса, на который указывает ссылка, так и в виде текста ссылки.

Решение

Чтобы отыскать адреса URL в тексте, можно воспользоваться одним из регулярных выражений из рецептов 7.2 или 7.4. А в качестве замещающего текста использовать:

```
<a•href="$&">$&</a>
```

Диалекты замещающего текста: .NET, JavaScript, Perl

```
<a•href="$0">$0</a>
```

Диалекты замещающего текста: .NET, Java, PHP

```
<a•href="\0">\0</a>
```

Диалекты замещающего текста: PHP, Ruby

```
<a•href="\&">\&</a>
```

Диалект замещающего текста: Ruby

```
<a•href="\g<0>">\g<0></a>
```

Диалект замещающего текста: Python

При работе над программой реализовать эту операцию поиска с заменой можно, как описывается в рецепте 3.15.

Обсуждение

Эта задача решается достаточно просто. Задействуется регулярное выражение, совпадающее с адресом URL, и затем найденное совпадение замещается текстом «‹a•href="*URL*"›*URL*‹/a›», где *URL* представляет найденный адрес URL. В различных языках программирования используется различный синтаксис определения замещающего текста, этим обусловлен такой длинный список решений для этой задачи. Но все они, по сути, делают одно и то же. Описание синтаксиса замещающего текста приводится в рецепте 2.20.

См. также

Рецепты 2.21, 3.15, 7.2 и 7.4.

7.6. Проверка строк URN

Задача

Требуется проверить, является ли строка допустимым унифицированным именем ресурса (Uniform Resource Name, URN), в соответствии с определением в RFC 2141, или отыскать строки URN в текстовом документе.

Решение

Проверить, является ли строка допустимым унифицированным именем ресурса:

```
\Aurn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^urn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск строк URN в текстовом документе:

```
\burn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Парметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки URN в текстовом документе, при этом предполагается, что знак пунктуации в конце URN является частью окружающего текста (английского), а не самой строки URN:

```
\burn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Строка URN состоит из трех частей. Первая часть – это четыре символа `urn:`, которые можно добавить в регулярное выражение как литералы.

Вторая часть – это идентификатор пространства имен (Namespace Identifier, NID). Он занимает от 1 до 32 символов. Первый символ должен быть буквой или цифрой. Остальные символы могут быть буквами, цифрами или дефисами. Совпадение с идентификатором обеспечивается с помощью двух символьных классов (рецепт 2.3): первому классу соответствует буква или цифра, а второму – от 0 до 31 букв, цифр и дефисов. Идентификатор должен отделяться от остальной части URN двоеточием, которое также можно добавить в регулярное выражение в виде литерала.

Третья часть URN – это строка из определенного пространства имен (Namespace Specific String, NSS). Она может иметь любую длину и в дополнение к буквам и цифрам может включать любые знаки пунктуации. Сопоставление с ней легко можно организовать с помощью другого символьного класса. Знак плюс, находящийся после символьного класса, повторяет его один или более раз (рецепт 2.12).

Если необходимо проверить, является ли заданная строка допустимой строкой URN, достаточно лишь добавить якорные метасимволы в начало и в конец регулярного выражения, чтобы обеспечить его привязку к началу и концу текста. Сделать это можно с помощью `‹^›` и `‹$›` во всех диалектах, за исключением Ruby, или с помощью `‹\A›` и `‹\Z›` во всех диалектах, за исключением JavaScript. Подробнее об этих якорных метасимволах рассказывается в рецепте 2.5.

Задача становится чуть сложнее, когда есть потребность отыскать строки URN в текстовом документе. Проблема знаков пунктуации, присущая адресам URL и обсуждавшаяся в рецепте 7.2, также свойственна и строкам URN. Предположим, что имеется текст:

```
The URN is urn:nid:nss, isn't it?
```

Проблема заключается в определении, является ли запятая частью URN. Строки URN, оканчивающиеся запятой, синтаксически являются допустимыми, но любой человек, читающий это предложение, поймет, что запятая является знаком препинания, а не частью URN. Последнее регулярное выражение в разделе «Решение» решает эту проблему, накладывая ограничение, не предусмотренное документом RFC 2141. Оно ограничивает последний символ строки URN набором символов, допустимых в части NSS, но маловероятных в качестве знаков препинания, которые могут появляться в предложении на английском языке, где упоминается эта строка URN.

Этого легко добиться, заменив квантификатор «плюс» (один или более раз) звездочкой (ноль или более раз) и добавив второй символьный класс, совпадающий с последним символом. Если бы мы добавили второй символьный класс, не изменив квантификатор, тем самым мы потребовали бы, чтобы часть NSS содержала, как минимум, два символа, а это не совсем то, что нам требуется.

См. также

Рецепты 2.3 и 2.12.

7.7. Проверка универсальных адресов URL

Задача

Необходимо проверить, является ли заданный фрагмент текста адресом URL в соответствии с требованиями RFC 3986.

Решение

```
\A
(# Схема
 [a-z][a-z0-9+\-.]*:
 (# Авторизация и путь
  //
  ([a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
  ([a-z0-9\-._~%]+                           # Имя хоста
  |\[[a-f0-9:.]+\]                           # IP-адрес хоста версии IPv6
  |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии
                                             # IPv...
  (:[0-9]+)?                                 # Порт
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?         # Путь
 |# Путь без авторизации
  (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (# Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\A
(# Схема
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (# Авторизация и путь
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
  (?<host>[a-z0-9\-._~%]+                           # Имя хоста
  |       \[[a-f0-9:.]+\]                           # IP-адрес версии IPv6
  |       \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес
                                                    # будущей версии
                                                    # IPv...
  (?<port>:[0-9]+)?                                 # Порт
  (?<path>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)       # Путь
 |# Путь без авторизации
  (?<path>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
          (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?<path>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET

```
\A
(# Схема
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (#Авторизация и путь
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
  (?<host>[a-z0-9\-._~%]+                           # Имя хоста
  |       \[[a-f0-9:.]+\]                           # IP-адрес версии IPv6
  |       \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес
                                                    # будущей версии
                                                    # IPv...
  (?<port>:[0-9]+)?                                 # Порт
  (?<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)   # Путь
```

```
 |# Путь без авторизации
  (?<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
                (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?<relpath>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(# Схема
 (?P<scheme>[a-z][a-z0-9+\-.]*):
 (# Авторизация и путь
  //
  (?P<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
  (?P<host>[a-z0-9\-._~%]+                           # Имя хоста
  |        \[[a-f0-9:.]+\]                           # IP-адрес версии IPv6
  |        \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес
                                                     # будущей версии
                                                     # IPv...
  (?P<port>:[0-9]+)?                                 # Порт
  (?P<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)   # Путь
 |# Путь без авторизации
  (?P<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
                (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?P<relpath>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(?P<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?P<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
^([a-z][a-z0-9+\-.]*:(\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/?[a-z0-9\-._~%!$&'()*+,;=:@]+↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?)?)|([a-z0-9\-._~%!$&'()*+,;=@]+↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)↵
+\/?))
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Большинство предыдущих рецептов в этой главе, имевших отношение к адресам URL, и регулярные выражения, приведенные в них, были рассчитаны на особые разновидности адресов URL. Некоторые из регулярных выражений были адаптированы для конкретных нужд, таких как определение, является ли знак пунктуации частью URL или текста, где приводится этот адрес URL.

Регулярные выражения в данном рецепте предусматривают работу с универсальными адресами URL. Они не ориентированы на поиск адресов URL в текстовых документах, а предназначены для проверки строк, которые, как предполагается, являются адресами URL, и деления адресов URL на отдельные части. Они решают все эти задачи для любых типов URL, при этом на практике часто требуется более тонкая настройка регулярных выражений. рецепты, следующие далее, демонстрируют такие специфические регулярные выражения.

В документе RFC 3986 описывается, как должен выглядеть допустимый адрес URL. Он охватывает все возможные разновидности адресов URL, включая относительные URL и адреса URL для схем адресации, которые еще не используются. В результате документ RFC 3986 получился чрезвычайно обширным, а регулярные выражения, реализующие его требования, – очень длинными. Регулярные выражения в этом рецепте реализуют только самые основные требования. Они достаточно надежно разбивают URL на различные части, но не выполняют проверку каждой из этих частей. Проверка всех частей могла бы потребовать точных сведений о каждой схеме адресации, используемой в адресах URL.

Документ RFC 3986 охватывает не все адреса URL, которые могут встречаться на практике. Например, многие веб-серверы и веб-броузеры принимают адреса URL с литералами пробелов в них, хотя RFC 3986 требует, чтобы пробелы были представлены как `%20`.

Абсолютный адрес URL должен начинаться с названия схемы, такой как `http:` или `ftp:`. Первый символ в названии схемы должен быть буквой. Последующие символы могут быть буквами, цифрами и некоторыми знаками пунктуации. Соблюдение этого требования легко можно реализовать с помощью двух символьных классов: ‹`[a-z][a-z0-9+\-.]*`›.

Во многих схемах адресации URL требуется то, что в RFC 3986 называется «авторизацией» («authority»). Авторизация – это доменное имя или IP-адрес сервера, которому может предшествовать имя пользователя и за которым может следовать номер порта.

Имя пользователя может состоять из букв, цифр и знаков пунктуации. Оно должно отделяться от доменного имени или IP-адреса знаком @. Подвыражение ‹`[a-z0-9\-._~%!$&'()*+,;=]+@`› совпадает с именем пользователя и разделительным знаком.

Документ RFC 3986 достаточно либерально относится к набору символов, которые могут составлять доменное имя. В рецепте 7.15 описывается, какие символы обычно допустимы в доменных именах: буквы, цифры, дефисы и точки. Кроме того, в RFC 3986 допускается использовать тильду и любые другие символы в формате записи со знаком процента. Доменное имя должно преобразовываться в кодировку UTF-8, а любые байты, которые не являются буквами, цифрами, дефисами или тильдой, должны быть представлены в формате *%FF*, где *FF* – это шестнадцатеричное представление значения байта.

Чтобы сохранить простоту нашего оригинального регулярного выражения, мы отказались от проверки, что за каждым знаком процента следуют точно две шестнадцатеричные цифры. Такого рода проверку лучше выполнить после того, как будут выделены отдельные части адреса URL. Так, мы обеспечиваем совпадение с именем хоста с помощью простого выражения ‹`[a-z0-9\-._~%]+`›, которому также соответствуют адреса IPv4 (допустимые в соответствии с RFC 3986).

Кроме доменного имени и адреса IPv4, хост может определяться адресом IPv6, заключенным в квадратные скобки, или даже версией IP-адресов, возможной в будущем. Совпадение с адресами IPv6 обеспечивается с помощью выражения ‹`\[[a-f0-9:.]+\]`›, а с версией, возможной в будущем, – с помощью выражения ‹`\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]`›. Хотя мы не можем проверить IP-адреса для версии, которая еще не существует, но мы могли бы более строго подойти к адресам IPv6. Однако и эту работу лучше оставить для второго регулярного выражения, которую можно выполнить после извлечения адреса из URL. рецепт 7.17 показывает, что проверка адресов IPv6 выполняется весьма тривиально.

Номер порта, если он указан, – это простое десятичное число, отделяемое от имени хоста двоеточием. Выражение ‹`:[0-9]+`› – это все, что нам необходимо.

Если авторизация определена, за ней должен следовать либо абсолютный путь, либо путь вообще не указывается. Абсолютный путь начинается с символа слэша, за которым следует один или более сегментов, отделяемых друг от друга слэшами. Сегмент состоит из одной или более букв, цифр или знаков пунктуации. Здесь символы слэша не могут следовать подряд. Путь может заканчиваться символом слэша. Совпадение с такими путями обеспечивает выражение ‹(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›.

Если в URL отсутствует авторизация, путь может быть как абсолютным, так и относительным или отсутствовать вообще. Абсолютный путь начинается с символа слэша, тогда как относительный – нет. Поскольку на этот раз ведущий символ слэша становится необязательным, нам потребовалось написать более длинное регулярное выражение, совпадающее сразу с абсолютными и относительными путями: ‹/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›.

Относительные адреса URL не определяют схему адресации, а, следовательно, и авторизацию. В этом случае путь становится обязательной частью адреса, при этом он может быть абсолютным или относительным. Поскольку в URL отсутствует схема, первый сегмент относительного пути не может содержать символы двоеточия. В противном случае этот символ двоеточия интерпретировался бы как символ, отделяющий наименование схемы. Поэтому, чтобы обеспечить совпадение с путем в относительном адресе URL, нам потребовались два регулярных выражения. Совпадение с относительными путями обеспечивает выражение ‹[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›. Оно очень похоже на выражение, соответствующее путям со схемой, но без авторизации. Отличия состоят в необязательности ведущего символа слэша, который отсутствует, и в содержимом первого символьного класса, из которого исключен символ двоеточия. Совпадение с абсолютным путем обеспечивает выражение ‹(/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?›. Это то же самое регулярное выражение, которое использовалось для совпадения с путями в адресах URL, в которых задается схема адресации и авторизация, за исключением того, что звездочка, повторяющая сегменты пути, заменена на плюс. Относительные пути требуют наличия хотя бы одного сегмента.

Часть со строкой запроса является необязательной. Если она присутствует, то должна начинаться со знака вопроса. Строка запроса продолжается до первого символа решетки или до конца адреса. Поскольку знак решетки не попадает в число знаков пунктуации, допустимых в строке запроса URL, мы легко можем обеспечить сопоставление с ней с помощью выражения ‹\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*›. Оба знака вопроса в этом регулярном выражении являются литералами. Первый из них находится за пределами символьного класса и должен экранироваться. Второй находится внутри символьного класса, где он всегда интерпретируется как литерал.

Заключительная часть URL – это фрагмент, который также является необязательным. Он начинается со знака решетки и тянется до конца адреса URL. Соответствие с ним обеспечивает выражение `‹\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*›`.

Чтобы упростить доступ к различным частям URL, мы использовали именованные сохраняющие группы. В рецепте 2.11 описывается, как работают именованное сохранение в различных диалектах регулярных выражений, рассматриваемых в этой книге. Диалект .NET – единственный, который допускает наличие в выражении нескольких групп с одинаковыми именами, как если бы они были одной группой. Это очень удобно в данной ситуации, потому что наше регулярное выражение предусматривает множество способов совпадения с путем в URL в зависимости от того, указывается ли схема и/или авторизация. Дав этим трем группам одно и то же имя, мы можем просто обратиться к группе с именем «path», чтобы получить путь независимо от того, указана ли в адресе URL схема и/или авторизация.

Другие диалекты не поддерживают такую возможность для именованного сохранения, даже при том, что многие придерживаются того же синтаксиса определения именованных сохранений. Для других диалектов все сохраняющие группы, совпадающие с путями, имеют разные имена. После того, как будет найдено совпадение, только одна из них будет действительно хранить путь. Две другие не будут участвовать в сопоставлении.

См. также

Рецепты 2.3, 2.8, 2.9 и 2.12.

7.8. Извлечение схемы из адреса URL

Задача

Требуется из строки, в которой хранится URL, извлечь схему адресации. Например, необходимо извлечь текст `http` из строки `http://www.regexcookbook.com`.

Решение

Извлечение схемы из заведомо допустимого адреса URL

```
^([a-z][a-z0-9+\-.]*):
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение схемы при проверке адреса URL

```
\A
([a-z][a-z0-9+\-.]*):
(# Авторизация и путь
 //
 ([a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
 ([a-z0-9\-._~%]+                           # Имя хоста
 |\[[a-f0-9:.]+\]                           # IP-адрес версии IPv6
 |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии
                                            # IPv...
 (:[0-9]+)?                                 # Порт
 (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?         # Путь
|# Путь без авторизации
 (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
)
# Строка запроса
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*):(//([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|(/?[a-z0-9\-._~%!$&'()*+,;=:@]+↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?)(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечь схему из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Схема URL всегда находится в самом начале адреса URL. Соблюдение этого требования в регулярном выражении обеспечивает символ крышки (рецепт 2.5). Схема должна начинаться с буквы, за которой могут следовать дополнительные буквы, цифры, знаки плюс, дефисы и точки. Совпадение с ними мы обеспечили с помощью двух символьных классов `‹[a-z][a-z0-9+\-.]*›` (рецепт 2.3).

Схема отделяется от остальной части URL символом двоеточия. Мы добавили его в регулярное выражение, чтобы гарантировать совпадение со схемой, только если адрес URL действительно начинается с нее. Относительные адреса URL не имеют схемы. Синтаксис URL, определя-

емый в RFC 3986, гарантирует отсутствие символов двоеточия в относительных адресах URL, если только этим двоеточиям не предшествуют символы, которые не допускается использовать в схемах. Именно по этой причине мы исключили символ двоеточия из символьного класса, используемого для сопоставления с путем, в рецепте 7.7. Если применить регулярные выражения из этого рецепта к действительному, но относительному URL, они вообще не обнаружат совпадения.

Поскольку регулярное выражение совпадает не только с названием схемы (в совпадение включается символ двоеточия), мы добавили в регулярное выражение сохраняющую группировку. Если регулярное выражение обнаружит совпадение, название схемы без двоеточия можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 7.7. Поскольку по условию задачи требуется извлечь схему, можно исключить из регулярного выражения сопоставление с относительными адресами URL, в которых наименование схемы не указывается. Это позволило упростить регулярное выражение.

Поскольку регулярное выражение совпадает со всем адресом URL, мы добавили дополнительную сохраняющую группировку, заключив в нее часть регулярного выражения, совпадающую со схемой. Извлекая текст, совпавший с первой сохраняющей группой, можно получить схему URL.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.9. Извлечение имени пользователя из URL

Задача

Требуется извлечь имя пользователя из строки, хранящей адрес URL. Например, нужно извлечь имя jan из `ftp://jan@www.regexcookbook.com`.

Решение

Извлечение имени пользователя из заведомо допустимого адреса URL

```
^[a-z0-9+\-.]+://([a-z0-9\-._~%!$&'()*+,;=]+)@
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение имени пользователя при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                        # Схема
([a-z0-9\-._~%!$&'()*+,;=]+)@                # Пользователь
([a-z0-9\-._~%]+                             # Имя хоста
|\[[a-f0-9:.]+\]                             # IP-адрес версии IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии IPv...
(:[0-9]+)?                                   # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?           # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?          # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?          # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+)@([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь имя пользователя из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Имя пользователя, если оно присутствует в URL, располагается правее схемы и двух символов слэша, с которых в URL начинается сегмент «авторизации». Имя пользователя отделяется от следующего за ним имени хоста знаком @. Поскольку знак @ не может появляться в именах хостов, мы можем быть уверены, что извлекли имя пользователя из URL, если обнаружили знак @ после двух символов слэша и перед следующим слэшем в URL. Символы слэша не могут использоваться в именах пользователей, поэтому нам не потребовалось выполнять дополнительную проверку.

Все эти правила подразумевают, что мы очень легко можем извлечь имя пользователя, если заранее известно, что испытуемый текст представляет допустимый адрес URL. Мы пропускаем схему с помощью регулярного выражения `<[a-z0-9+\-.]+>` и `://`. После этого извлекаем имя пользователя, следующее дальше. Если было найдено совпадение со знаком @, можно быть уверенными, что символы перед ним составляют имя пользователя. Символьный класс `<[a-z0-9\-._~%!$&'()*+,;=]>` перечисляет все символы, которые считаются допустимыми в именах пользователей.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит имя пользователя. В этом случае в совпадение с регулярным выражением попадут обе части URL, схема и имя пользователя. Поэтому мы добавили в регулярное выражение сохраняющую группировку. Когда регулярное выражение обнаружит совпадение, имя пользователя без символов-разделителей или других частей URL можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 7.7. Поскольку по условию задачи требуется извлечь имя пользователя, можно исключить из регулярного выражения сопоставление с адресами URL, в которых отсутствует авторизация. Фактически регулярное выражение, которое приводится в решении, совпадает только с адресами URL, которые содержат авторизацию, включающую имя пользователя. Требование к наличию сегмента авторизации позволило упростить регулярное выражение. Более того, оно получилось даже проще, чем в рецепте 7.8

Поскольку это регулярное выражение совпадает со всем адресом URL, мы добавили дополнительную сохраняющую группировку, заключив в нее часть регулярного выражения, совпадающую с именем пользователя. Извлекая текст, совпавший с первой сохраняющей группой, можно получить имя пользователя URL.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат имени пользователя, можно воспользоваться одним из регулярных выражений в рецепте 7.7. Первое регулярное выражение в рецепте 7.7 сохраняет имя пользователя (если оно указано) в третьей сохраняющей группе. Эта группа сохраняет также символ @. Если необходимо получить имя пользователя без символа @, можно добавить в регулярное выражение еще одну сохраняющую группировку.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.10. Извлечение имени хоста из URL

Задача

Требуется извлечь имя хоста из строки, хранящей адрес URL. Например, имя `www.regexcookbook.com` из `http://www.regexcookbook.com`.

Решение

Извлечение имени хоста из заведомо допустимого адреса URL

```
\A
[a-z][a-z0-9+\-.]*://             # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?    # Имя пользователя
([a-z0-9\-._~%]+                  # Имя хоста или IP-адрес IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес IPv6+
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-z0-9\-._~%!$&'()*+,;=:]+\])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение имени хоста при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                        # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?               # Имя пользователя
([a-z0-9\-._~%]+                             # Имя хоста
|\[[a-f0-9:.]+\]                             # IP-адрес IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])  # IP-адрес будущей версии IPv...
(:[0-9]+)?                                   # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?           # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?          # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?          # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь имя хоста из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Мы использовали

якорные метасимволы ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][az0-9+\-.]*://› пропускает схему, а выражение ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› пропускает необязательное имя пользователя. Сразу же вслед за ними следует имя хоста.

В RFC 3986 допускается две различные формы записи хоста. Доменные имена и адреса IPv4 указываются без квадратных скобок, тогда как адреса IPv6 и IP-адреса возможных будущих версий указываются в квадратных скобках. Эти две формы записи необходимо обрабатывать отдельно, потому что форма записи в квадратных скобках допускает большее число знаков пунктуации, чем форма записи без скобок. В частности, в квадратных скобках допускается использовать символ двоеточия, а в доменных именах и адресах IPv4 – нет. Двоеточие также используется для отделения именно хоста (в обеих формах записи) от номера порта.

Выражению ‹[a-z0-9\-._~%]+› соответствуют доменные имена и адреса IPv4. Выражение ‹\[[a-z0-9\-._~%!$&'()*+,;=:]+\]› обрабатывает адреса версии IPv6 и выше. Мы объединили эти два регулярных выражения, используя конструкцию выбора (рецепт 2.8), в группу. Кроме этого, сохраняющая группа позволяет извлекать имя хоста.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит имя хоста. В этом случае в совпадение с регулярным выражением попадут схема, имя пользователя и имя хоста. Когда регулярное выражение обнаружит совпадение, имя хоста без символов-разделителей или других частей URL можно будет извлечь из второй сохраняющей группы. Для адресов IPv6 в совпадение с сохраняющей группой будут включаться квадратные скобки. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 7.7. Поскольку по условию задачи требуется извлечь имя хоста, можно исключить из регулярного выражения сопоставление с адресами URL, в которых отсутствует авторизация. Это существенно упрощает регулярное выражение. Оно очень похоже на регулярное выражение, которое приводилось в рецепте 7.9. Единственное отличие состоит в том, что теперь имя пользователя опять стало необязательным, как и в рецепте 7.7.

Кроме того, это регулярное выражение использует конструкцию выбора для различных форм записи имени хоста, заключенную в сохраняющую группу. Извлекая текст, совпавший со второй сохраняющей группой, можно получить имя хоста URL.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат имени хоста,

можно воспользоваться одним из регулярных выражений из рецепта 7.7. Первое регулярное выражение в рецепте 7.7 сохраняет имя хоста (если оно указано) в четвертой сохраняющей группе.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.11. Извлечение номера порта из URL

Задача

Требуется извлечь номер порта из строки, хранящей адрес URL. Например, нужно извлечь число 80 из адреса `http://www.regexcookbook.com:80/`.

Решение

Извлечение имени хоста из заведомо допустимого адреса URL

```
\A
[a-z][a-z0-9+\-.]*://             # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?    # Имя пользователя
([a-z0-9\-._~%]+                  # Имя хоста или адрес IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # Адрес IPv6+
:(?<port>[0-9]+)                  # Номер порта
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?↵
([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]):([0-9]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение имени хоста при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                      # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?             # Имя пользователя
([a-z0-9\-._~%]+                           # Имя хоста
|\[[a-f0-9:.]+\]                           # Адрес IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии IPv...
:([0-9]+)                                  # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?         # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?        # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?        # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*:\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?↵
([a-z0-9\-._~%]+|\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]↵
+\]):([0-9]+)(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?↵
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь номер порта из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Мы использовали якорные метасимволы ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][a-z0-9+\-.]*://› пропускает схему, а выражение ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› пропускает необязательное имя пользователя. Выражение ‹([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\])› пропускает имя хоста.

Номер порта отделяется от имени хоста символом двоеточия, который мы добавили в регулярное выражение как литерал. Номер порта – это просто строка цифр, совпадение с которой легко обеспечить с помощью выражения ‹[0-9]+›.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит номер порта. В этом случае в совпадение с регулярным выражением попадут схема, имя пользователя, имя хоста и номер порта. Когда регулярное выражение обнаружит совпадение, номер порта без символов-разделителей или других частей URL можно будет извлечь из третьей сохраняющей группы.

Другие две группы обеспечивают необязательность имени пользователя и объединение альтернативных форм записи имени хоста. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, что испытуемый текст является допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 7.7. Поскольку по условию задачи требуется извлечь номер порта, можно исключить из регулярного выражения сопоставление с адресами URL, в которых отсутствует авторизация. Это существенно упрощает регулярное выражение. Оно очень похоже на регулярное выражение, которое приводилось в рецепте 7.10.

Единственное отличие состоит в том, что теперь номер порта уже не является необязательным, а еще мы переместили границу сохраняющей

группировки, соответствующей номеру порта, исключив из нее двоеточие, отделяющее номер порта от имени хоста. Номер сохраняющей группы – 3.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат номера порта, можно воспользоваться одним из регулярных выражений из рецепта 7.7. Первое регулярное выражение в рецепте 7.7 сохраняет номер порта (если он указан) в пятой сохраняющей группе.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.12 Извлечение пути из адреса URL

Задача

Требуется извлечь путь из строки, хранящей адрес URL. Например, нужно извлечь фрагмент `/index.html` из адреса `http://www.regexcookbook.com/index.html` или из адреса `/index.html#fragment`.

Решение

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Следующее регулярное выражение совпадает со всеми URL, даже с теми, в которых путь не указан:

```
\A
# Пропустить схему и авторизацию, если имеются
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
# Путь
([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Совпадение должно обеспечиваться только с адресами URL, содержащими путь:

```
\A
# Пропустить схему и авторизацию, если имеются
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
```

```
# Путь
(/?[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)
# Строка запроса, фрагмент или конец URL
([#?]|\Z)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?(/?[a-z0-9\-._~%!$&'()*+,;=@]+↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)([#?]|$)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Чтобы обеспечить совпадение только с адресами URL, содержащими путь, использовать атомарную группировку:

```
\A
# Пропустить схему и авторизацию, если имеются
(?>([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?)
# Путь
([a-z0-9\-._~%!$&'()*+,;=:@/]+)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

Обсуждение

Регулярное выражение, извлекающее путь, можно существенно упростить, если известно, что испытуемый текст является допустимым адресом URL. В рецепте 7.7 предусмотрено три различных способа совпадения с путем, в зависимости от наличия сегментов схемы и/или авторизации в URL, но в регулярном выражении, извлекающем путь из заведомо допустимого адреса URL достаточно единственного сопоставления с путем.

Регулярное выражение начинается с якорного метасимвола ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][a-z0-9+\-.]*:› перешагивает через схему, а выражение ‹//[^/?#]+› — через авторизацию. Мы смогли использовать такое простое выражение, соответствующее авторизации, потому что уже знаем, что URL допустим и нам не требуется извлекать имя пользователя, имя хоста или номер порта из авторизации. Сегмент авторизации начинается с двух символов слэша и простирается до начала пути (символ слэша), строки запроса (знак вопроса) или фрагмента (символ решетки). Инвертированный символьный класс обеспечивает совпадение с любыми символами, кроме первого символа слэша, знака вопроса и решетки (рецепт 2.3).

Поскольку сегмент авторизации является необязательным, мы поместили его в группу, за которой следует квантификатор «знак вопроса»: ‹(//[^/?#]+)?›. Схема также является необязательным элементом URL. Если схема опущена, авторизация тоже должна быть опущена. Чтобы обеспечить соблюдение этого правила, мы поместили части регулярного выражения, совпадающие со схемой и необязательным сегментом авторизации в другую группу, которую также сделали необязательной с помощью вопросительного знака.

Нам заранее известно, что испытуемый текст является допустимым адресом URL, поэтому совпадение с сегментом пути легко можно обеспечить с помощью единственного символьного класса ‹[a-z0-9\-._~%!$&'()*+,;=:@/]*›, который включает символ слэша. Нам не требуется проверять наличие двойных символов слэша, недопустимых в пути.

Мы целенаправленно использовали звездочку, а не плюс в качестве квантификатора символьного класса для сегмента пути. Может показаться странным, что совпадение с путем сделано необязательным в регулярном выражении, единственная цель которого извлечь путь из URL. Фактически совпадение с сегментом пути должно быть необязательным из-за упрощений, сделанных нами при перескакивании сегментов схемы и авторизации.

В универсальном регулярном выражении в рецепте 7.7 имеется три разных способа сопоставления с сегментом пути в зависимости от наличия в URL схемы и/или авторизации. Это гарантирует, что схема по ошибке не будет воспринята, как путь.

Но сейчас мы стараемся упростить выражение, используя единственный символьный класс для сегмента пути. Рассмотрим URL http://www.regexcookbook.com, где имеются сегменты схемы и авторизации, но отсутствует путь. Первая часть нашего регулярного выражения благополучно совпадет со схемой и авторизацией. Механизм регулярных выражений попытается сопоставить символьный класс для пути, но в тексте больше не осталось символов. Если путь объявлен необязательным (с помощью квантификатора «звездочка»), механизм регулярных выражений будет вполне удовлетворен отсутствием символов для пути. Он достигнет конца регулярного выражения и объявит, что общее совпадение было найдено.

Но если бы совпадение с символьным классом пути было обязательным, механизм регулярных выражений начал бы выполнять возвраты. (Те, кто не знаком с механизмом возвратов, могут обратиться к рецепту 2.13.) Он вспомнит, что части регулярного выражения, совпадающие с сегментами схемы и авторизации, объявлены необязательными, поэтому механизм скажет: попробуем еще раз, но на этот раз исключим возможность совпадения с ‹(//[^/?#]+)?›. Тогда подвыражение ‹[a-z0-9\-._~%!$&'()*+,;=:@/]+› совпадет с фрагментом //www.regexcookbook.com, интерпретировав его как путь, что совершенно не то, что нам тре-

буется. Если бы для сопоставления с сегментом пути использовалось более точное регулярное выражение, не допускающее появления двойных символов слэша, механизм регулярных выражений смог бы выполнить возврат еще раз и попробовать вариант с отсутствующей схемой. Тогда, при использовании точного выражения для пути, фрагмент `http` адреса был бы воспринят как путь. Для предотвращения этого нам пришлось бы добавить дополнительную проверку наличия строки запроса и фрагмента за сегментом пути. Если реализовать все это, мы получим регулярные выражения, обозначенные в разделе «Решение» как «совпадающие только с адресами URL, действительно содержащими путь». Они получились немного сложнее, только чтобы обеспечить отсутствие совпадения с адресом URL без пути.

Если используемый диалект регулярных выражений поддерживает атомарную группировку, она обеспечит более простое решение. Атомарную группировку (рецепт 2.15) поддерживают все диалекты, рассматриваемые в книге, за исключением JavaScript и Python. Атомарная группировка сообщает механизму регулярных выражений, что не следует выполнять возврат. Если подвыражения, совпадающие с сегментами схемы и авторизации, заключить в атомарную группу, механизм регулярных выражений будет вынужден сохранить совпадения со схемой и авторизацией, как только они будут обнаружены, даже если в результате этого не останется символов для символьного класса, обеспечивающего совпадение с путем. Это решение такое же эффективное, как и предыдущее, с необязательным сегментом пути.

Путь можно будет извлечь из третьей сохраняющей группы при использовании любого регулярного выражения из этого рецепта. Если использовать первые два выражения, в которых совпадение с путем является необязательным, в третьей сохраняющей группе может возвращаться пустая строка или значение `null` в JavaScript.

Если заранее неизвестно, что испытуемый текст является допустимым адресом URL, можно воспользоваться регулярным выражением из рецепта 7.7. При работе с платформой .NET можно применить выражение, работающее только с диалектом .NET и содержащее три группы с именем «path» в трех подвыражениях, способных совпасть с сегментом пути в URL. В других диалектах, поддерживающих именованные сохранения, путь будет сохраняться в одной из трех групп: «hostpath», «schemepath» или «relpath». Так как только одна из трех групп действительно будет содержать что-нибудь, для извлечения пути можно воспользоваться простой хитростью, выполнив операцию конкатенации строк, возвращаемых тремя группами. Две из них будут возвращать пустые строки, поэтому фактически никакой конкатенации выполняться не будет.

Если используемый диалект не поддерживает именованное сохранение, можно использовать первое регулярное выражение из рецепта 7.7. Оно сохраняет путь в группе 6, 7 или 8. В этом случае также можно использовать трюк с конкатенацией фрагментов, сохраняемых этими тремя

группами, потому что две из них будут возвращать пустые строки. Однако в JavaScript этот прием работать не будет. Для групп, не участвовавших в сопоставлении, в JavaScript возвращается значение `undefined`.

В рецепте 3.9 приводится немало информации об извлечении текста, совпавшего с нумерованными или именованными сохраняющими группами в разных языках программирования.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.13. Извлечение строки запроса из URL

Задача

Требуется извлечь строку запроса из строки, хранящей адрес URL. Например, нужно извлечь фрагмент param=value из `http://www.regexcookbook.com?param=value` или из `/index.html?param=value`.

Решение

```
^[^?#]+\?([^#]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение строки запроса из URL – достаточно тривиальная задача, если заведомо известно, что испытуемый текст является допустимым адресом URL. Запрос отделяется от предшествующей части URL знаком вопроса. То есть первым знаком вопроса, встретившимся в URL. Благодаря этому с помощью регулярного выражения ‹`^[^?#]+\?`› можно сразу же перейти к первому вопросительному знаку. Знак вопроса в регулярных выражениях интерпретируется как метасимвол только за пределами символьных классов, но не внутри, поэтому нам потребовалось экранировать литерал знака вопроса за пределами символьного класса. Первый символ ‹`^`› – это якорь (рецепт 2.5), тогда как второй символ ‹`^`› инвертирует символьный класс (рецепт 2.3).

Знак вопроса может появляться в адресах URL как часть (необязательного) фрагмента, следующего за строкой запроса. Поэтому, чтобы убедиться, что это первый знак вопроса в URL и он не является частью фрагмента в адресе URL, в котором отсутствует строка запроса, вместо простого подвыражения ‹`\?`› пришлось использовать ‹`^[^?#]+\?`›.

Строка запроса простирается до начала фрагмента или до конца строки URL, если в ней отсутствует сегмент фрагмента. Фрагмент отделяется от остальной части URL знаком решетки. Так как знак решетки не

допускается нигде, кроме фрагмента, выражения ‹[^#]+› будет вполне достаточно для совпадения с запросом. Инвертированный символьный класс совпадает с любыми символами до первого встретившегося знака решетки или до конца испытуемого текста, если в нем нет ни одного знака решетки.

Это регулярное выражение будет совпадать только с адресами URL, которые действительно содержат строку запроса. Когда будет обнаруживаться совпадение с URL, в его состав будет входить все, от начала URL, поэтому мы поместили часть ‹[^#]+› регулярного выражения, совпадающую со строкой запроса, в сохраняющую группу. Когда регулярное выражение обнаружит совпадение, строку запроса без символов-разделителей или других частей URL можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать одно из регулярных выражений из рецепта 7.7. Первое регулярное выражение в рецепте 7.7 сохраняет строку запроса, если она присутствует в строке URL, в сохраняющей группе под номером 12.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.14. Извлечение фрагмента из URL

Задача

Требуется извлечь сегмент фрагмента из строки, хранящей адрес URL. Например, нужно извлечь фрагмент top из `http://www.regexcookbook.com#top` или из `/index.html#top`.

Решение

```
#(.+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение фрагмента из URL – достаточно тривиальная задача, если заведомо известно, что испытуемый текст является допустимым адресом URL. Фрагмент отделяется от предшествующей части URL знаком

решетки. Фрагмент – это единственная часть URL, где допускается присутствие знака решетки, и фрагмент – всегда последняя часть URL. Поэтому мы легко можем извлечь фрагмент, отыскав первый знак решетки и выбрав все символы до конца строки. С этой задачей прекрасно справляется выражение `‹#.+›`. Не забудьте выключить режим свободного форматирования, в противном случае потребуется экранировать литерал знака решетки символом обратного слэша.

Это регулярное выражение будет совпадать только с адресами URL, которые действительно содержат фрагмент. В совпадении будет присутствовать только фрагмент, но оно включает в себя знак решетки, отделяющий фрагмент от остальной части URL. Чтобы получить возможность извлекать один только фрагмент, без отделяющего символа #, в решение была добавлена сохраняющая группа.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать одно из регулярных выражений из рецепта 7.7. Первое регулярное выражение в рецепте 7.7 сохраняет фрагмент, если он присутствует в строке URL, в сохраняющей группе под номером 13.

См. также

Рецепты 2.9, 3.9 и 7.7.

7.15. Проверка доменных имен

Задача

Необходимо проверить, выглядит ли испытуемый текст как полное квалифицированное доменное имя, или отыскать такие доменные имена в текстовом документе.

Решение

Проверить, выглядит ли строка как допустимое доменное имя:

```
^([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Отыскать допустимые доменные имена в текстовом документе:

```
\b([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверить, не превышает ли длина каждой части доменного имени 63 символов:

```
\b((?=[a-z0-9-]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Разрешить использование интернационализированных доменных имен в кодировке punycode:

```
\b((xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверить, не превышает ли длина каждой части доменного имени 63 символов, и разрешить использование интернационализированных доменных имен в кодировке punycode:

```
\b((?=[a-z0-9-]{1,63}\.)(xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Доменное имя имеет вид `domain.tld` или `subdomain.domain.tld`, или может включать любое число дополнительных поддоменов. Домен верхнего уровня (top-level domain, `tld`) состоит из двух или более букв. Это сама простая часть регулярного выражения: ‹`[a-z]{2,}`›.

Имена доменов и любых поддоменов состоят из букв, цифр и дефисов. Дефисы не могут следовать парами и не могут быть первыми символами в именах доменов. Совпадение с ними обеспечивается регулярным выражением ‹`[a-z0-9]+(-[a-z0-9]+)*`›. Это регулярное выражение допускает любое число букв и цифр, за которыми может следовать любое число групп символов, состоящих из дефисов, за которыми следуют другие последовательности букв и цифр. Необходимо помнить, что внутри символьных классов дефис интерпретируется как метасимвол (рецепт 2.3), но за их пределами он воспринимается как обычный символ, поэтому нам не потребовалось экранировать дефисы в этом регулярном выражении.

Имена доменов и поддоменов отделяются друг от друга символом точки, совпадение с которой обеспечивает фрагмент ‹`\.`› в регулярном выражении. Так как в URL помимо имени домена может присутствовать любое число имен поддоменов, мы поместили подвыражение, совпадающее с именем домена и с литералом точки в повторяемую группу:

‹([a-z0-9]+(-[a-z0-9]+)*\.)+›. Так как имена поддоменов следуют тому же синтаксису, что и имена доменов, эта группа обрабатывает и те, и другие имена.

Если необходимо проверить, является ли заданная строка допустимым именем домена, достаточно добавить якорные метасимволы в начало и в конец регулярного выражения, чтобы обеспечить его привязку к началу и концу текста. Сделать это можно с помощью ‹^› и ‹$› во всех диалектах за исключением Ruby или с помощью ‹\A› и ‹\Z› во всех диалектах за исключением JavaScript. Подробнее об этих якорных метасимволах рассказывается в рецепте 2.5.

Если потребуется отыскать доменные имена в текстовом документе, можно добавить границы слова ‹\b› (рецепт 2.6).

Первый набор регулярных выражений не проверяет, превышает ли длина каждой части доменного имени 63 символа. Реализовать такую проверку не так просто, потому что наше регулярное выражение, совпадающее с каждой частью доменного имени, ‹[a-z0-9]+(-[a-z0-9]+)*›, содержит в себе три квантификатора и нет никакой возможности сообщить механизму регулярных выражений, что в сумме они не должны производить более 63 повторений.

Можно было бы использовать выражение ‹[-a-z0-9]{1,63}›, чтобы ограничить длину каждой части доменного имени 63 символами, или ‹\b([-a-z0-9]{1,63}\.)+[a-z]{2,63}›, для всего доменного имени. Но тогда мы лишились бы возможности исключать доменные имена с символами дефиса в недопустимых местах.

Однако можно задействовать опережающую проверку, чтобы дважды проверить совпадение с одним и тем же текстом. Те, кто не знаком с опережающей проверкой, могут сначала ознакомиться с рецептом 2.16. Мы использовали то же самое регулярное выражение ‹[a-z0-9]+(-[a-z0-9]+)*\.›, совпадающее с доменным именем, в котором дефисы находятся в допустимых местах, и добавили в опережающую проверку выражение ‹[-a-z0-9]{1,63}\.›, чтобы убедиться, что длина имени не превышает 63 символов. В результате получилось выражение ‹(?=[-a-z0-9]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.›.

Опережающая проверка ‹(?=[-a-z0-9]{1,63}\.)› сначала убеждается, что до ближайшего символа точки имя содержит от 1 до 63 букв, цифр и дефисов. Точка в опережающей проверке имеет большое значение. Без нее доменные имена длиннее 63 символов по-прежнему могли бы удовлетворять ограничению в 63 символа, накладываемому опережающей проверкой. Только добавив литерал точки внутрь опережающей проверки, мы можем обеспечить соблюдение ограничения длины 63 символами.

Опережающая проверка не поглощает символы совпавшего с ней текста. Поэтому если опережающая проверка преуспевает, выражение ‹[a-z0-9]+(-[a-z0-9]+)*\.› применяется к тексту, уже совпавшему с опережающей проверкой. Мы уже убедились, что эта часть имени не пре-

вышает 63 символов в длину, и теперь осталось лишь убедиться, что она представляет собой допустимую комбинацию дефисов и других символов.

Интернационализированные доменные имена (Internationalized Domain Names, IDN) теоретически могут содержать почти любые символы. Фактически же список допустимых символов зависит от регистратора, управляющего доменом верхнего уровня. Например, в домене `.es` допускается использовать доменные имена, содержащие испанские символы.

На практике интернационализированные доменные имена часто кодируются с помощью схемы, называемой *punycode*. Несмотря на всю сложность алгоритма punycode, здесь важно упомянуть, что в результате его применения получаются доменные имена, состоящие из букв, цифр и дефисов и отвечающие требованиям, которые мы уже удовлетворили в нашем регулярном выражении для доменных имен. Единственное отличие в том, что доменные имена, воспроизводимые алгоритмом punycode, начинаются с префикса `xn--`. Чтобы добавить поддержку таких доменов в наше регулярное выражение, нам достаточно добавить `‹(xn--)?›` в группу, совпадающую с частями доменного имени.

См. также

Рецепты 2.3, 2.12 и 2.16.

7.16. Сопоставление с адресами IPv4

Необходимо проверить, является ли некоторая строка допустимым адресом IPv4 в форме записи 255.255.255.255. Дополнительно необходимо преобразовать этот адрес в 32-битное целое число.

Решение

Регулярное выражение

Простое регулярное выражение, выполняющее проверку IP-адресов:

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, выполняющее проверку IP-адресов:

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простое регулярное выражение, извлекающее IP-адреса из текстового документа:

```
\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, извлекающее IP-адреса из текстового документа:

```
\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простое регулярное выражение, сохраняющее четыре части IP-адреса:

```
^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, сохраняющее четыре части IP-адреса:

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($subject =~ m/^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})/)
{
    $ip = $1 << 24 | $2 << 16 | $3 << 8 | $4;
}
```

Обсуждение

IP-адреса в версии 4 обычно записываются в формате 255.255.255.255, где каждая часть должна быть представлена числом от 0 до 255. Сопоставление с такими IP-адресами реализуется очень просто.

В решении представлены четыре регулярных выражения. Два из них обозначены как «простые», а два других – как «точные».

В простых регулярных выражениях для сопоставления с каждым из четырех блоков цифр IP-адреса используется подвыражение `‹[0-9]{1,3}›`. Оно фактически совпадает с числами в диапазоне от 0 до 999, а не от 0 до 255. Простые регулярные выражения более эффективны, когда

заранее известно, что строка содержит только допустимые IP-адреса, и требуется только отделить IP-адреса от остального текста.

В точных регулярных выражениях для сопоставления с каждым из четырех блоков цифр IP-адреса используется подвыражение ‹25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?›. Это регулярное выражение точно соответствует числу в диапазоне от 0 до 255, содержащему необязательный ведущий ноль для чисел в диапазоне от 10 до 99 и два необязательных ведущих нуля для чисел в диапазоне от 0 до 9. Выражение ‹25[0-5]› совпадает с числами от 250 до 255, выражение ‹2[0-4][0-9]› совпадает с числами от 200 до 249 и выражение ‹[01]?[0-9][0-9]?› совпадает с числами от 0 до 199, включая необязательные ведущие нули. Подробнее о сопоставлении с числовыми диапазонами рассказывается в рецепте 6.5.

Если необходимо проверить, является ли вся строка допустимым IP-адресом, следует использовать одно из регулярных выражений, начинающихся символом крышки и заканчивающихся знаком доллара. Эти якорные метасимволы, совпадающие с началом и концом строки, описываются в рецепте 2.5. Если необходимо выполнить поиск IP-адресов в текстовом документе, следует использовать регулярные выражения, начинающиеся и заканчивающиеся метасимволом границы слова ‹\b› (рецепт 2.6).

Первые четыре регулярных выражения имеют вид ‹(?:number\.){3}number›. Первые три числа в IP-адресе совпадают с несохраняющей группой (рецепт 2.9), которая повторяется три раза (рецепт 2.12). Группе соответствует число и литерал точки, которая трижды встречается в IP-адресе. Последней части регулярного выражения соответствует заключительное число IP-адреса. Использование несохраняющей группы и трехкратное ее повторение делает регулярное выражение короче и эффективнее.

Чтобы преобразовать текстовое представление IP-адреса в целое число, необходимо сохранить все четыре числа отдельно. Это реализовано в двух последних регулярных выражениях. Вместо одной группы, повторяющейся трижды, в них используются четыре сохраняющих группы, по одной для каждого числа. Это единственный способ сохранить по отдельности все четыре числа, присутствующие в IP-адресе.

После сохранения чисел объединить их в 32-битное целое число не составляет труда. Текст, совпавший с четырьмя сохраняющими группами регулярного выражения, в языке Perl хранится в специальных переменных $1, $2, $3 и $4. Порядок извлечения содержимого сохраняющих групп в других языках программирования описывается в рецепте 3.9. В языке Perl строковые переменные, соответствующие сохраняющим группам, автоматически преобразуются в числа при применении к ним оператора битового сдвига (<<). В других языках программирования может потребоваться вызвать метод String.toInteger() или похожий на него, прежде чем можно будет выполнить сдвиг чисел и объединить их.

См. также

Рецепты 2.3, 2.8, 2.9 и 2.12.

7.17. Сопоставление с адресами IPv6

Необходимо проверить, является ли некоторая строка допустимым адресом IPv6 в стандартной, компактной и/или смешанной форме записи.

Решение

Стандартная форма записи

Обеспечить совпадение с адресом IPv6 в стандартной форме записи, когда адрес состоит из восьми 16-битовых слов в шестнадцатеричной нотации, разделенных двоеточиями (например: `1762:0:0:0:0:B03:1:AF18`). Ведущие нули являются необязательными.

Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной форме записи:

```
^(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Отыскать в текстовом документе адреса IPv6 в стандартной форме записи:

```
(?<![:.\w])(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Смешанная форма записи

Обеспечить совпадение с адресом IPv6 в смешанной форме записи, когда адрес состоит из шести 16-битовых слов в шестнадцатеричной нотации, за которыми следуют четыре байта в десятичной нотации. Слова отделяются двоеточиями, а байты – точками. Слова и байты отделяются двоеточием. Ведущие нули в шестнадцатеричных словах и в десятичных байтах являются необязательными. Данная форма записи используется в случае смешивания адресов формата IPv4 и IPv6 и когда адреса IPv4 дополняются до формата IPv6. Пример адреса IPv6 в смешанной форме записи: 1762:0:0:0:0:B03:127.32.67.15.

Проверить, является ли весь испытуемый текст адресом IPv6 в смешанной форме записи:

```
^(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Отыскать в текстовом документе адреса IPv6 в смешанной форме записи:

```
 (?<![:.\w])(?:[A-F0-9]{1,4}:){6}↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Стандартная или смешанная форма записи

Обеспечить совпадение с адресом IPv6 в стандартной или смешанной форме записи.

Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной или смешанной форме записи:

```
\A                                          # Начало текста
(?:[A-F0-9]{1,4}:){6}                       # 6 слов
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}               # 2 слова
```

```
|  (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}  # или 4 байта
   (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)\Z                                                   # Конец текста
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?))$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной или смешанной форме записи:

```
(?<![:.\w])                                           # Привязка к адресу
(?:[A-F0-9]{1,4}:){6}                                 # 6 слов
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}                        # 2 слова
|  (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} # или 4 байта
   (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)(?![:.\w])                                           # Привязка к адресу
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b                                                    # Граница слова
(?:[A-F0-9]{1,4}:){6}                                 # 6 слов
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}                        # 2 слова
|  (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} # или 4 байта
   (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)\b                                                   # Граница слова
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?))\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Компактная форма записи

Обеспечить совпадение с адресом IPv6 в компактной форме записи. Компактная форма записи идентична стандартной, за исключением того, что в компактной форме одна из последовательностей из одного или более слов, содержащих ноль, может быть опущена, но при этом двоеточия до и после опущенных нулей остаются. Опознать адрес в компактной форме записи можно по двум двоеточиям, следующим подряд. Опущена может быть только одна последовательность нулей, в противном случае будет невозможно определить, как много слов было опущено в каждой последовательности. Если опущенная последовательность нулей находится в начале или в конце IP-адреса, он будет начинаться или оканчиваться двумя двоеточиями. Если в IP-адресе все числа равны нулю, такой адрес в компактной форме будет содержать всего два двоеточия, без каких либо цифр.

Например, 1762::B03:1:AF18 — это адрес 1762:0:0:0:0:B03:1:AF18 в компактной форме записи. Регулярные выражения в этом разделе будут совпадать как с компактной, так и со стандартной формой записи адреса IPv6. Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной или компактной форме записи:

```
\A(?:
 # Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    \Z) # и конец текста
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|↵
(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}$)(([0-9A-F]{1,4}:){1,7}|:)↵
((:[0-9A-F]{1,4}){1,7}|:))$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной или компактной форме записи:

```
(?<![:.\w])(?:
 # Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
```

```
 # и не более 1 пары двоеточий
(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
 # Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↵
[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4})↵
{1,7}|:))(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Смешанная компактная форма записи

Обеспечить совпадение с адресом IPv6 в смешанной компактной форме записи. Компактная смешанная форма записи идентична смешанной, за исключением того, что в компактной форме записи одна из последовательностей из одного или более слов, содержащих ноль, может быть опущена, при этом двоеточия до и после опущенных нулей оставлены. Четыре десятичных байта должны присутствовать всегда, даже если они равны нулю. Опознать адрес в компактной смешанной форме записи можно по двум двоеточиям, следующим подряд, в первой части адреса, и по трем точкам во второй части. Опущена может быть только одна последовательность нулей, в противном случае будет невозможно определить, как много слов было опущено в каждой последователь-

ности. Если опущенная последовательность нулей находится в начале IP-адреса, он будет начинаться двумя двоеточиями, а не цифрой.

Например, 1762::B03:127.32.67.15 — это компактная смешанная форма записи адреса 1762:0:0:0:0:B03:127.32.67.15. Регулярные выражения в этом разделе будут совпадать с адресами IPv6 в смешанной форме записи, как в компактной, так и не в компактной.

Проверить, является ли весь испытуемый текст адресом IPv6 в компактной или некомпактной смешанной форме записи:

```
\A
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, число двоеточий не более 6
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    \Z) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.)↵
{3}[0-9]{1,3}$)(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]↵
[0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в смешанной компактной или некомпактной форме записи:

```
(?<![:.\w])
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, число двоеточий не более 6
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
```

```
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, не более 6 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↵
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Стандартная, смешанная или компактная форма записи

Обеспечить совпадение с адресом IPv6 в любой из форм записи, описанных выше: стандартной, смешанной и компактной.

Проверить, является ли весь испытуемый текст адресом IPv6:

```
\A(?:
# Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     \Z) # и привязка
  # и не более 1 пары двоеточий
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    \Z) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↵
[0-9]{1,3}$)(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)|(?:[A-F0-9]{1,4}:){7}↵
[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}$)↵
(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:))$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной, смешанной или компактной форме записи:

```
(?<![:.\w])(?:
# Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     (?![:.\w])) # и привязка
```

```
  # и не более 1 пары двоеточий
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
# Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     (?![:.\w])) # и привязка
  # и не более 1 пары двоеточий
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
 (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↵
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|↵
[01]?[0-9][0-9]?)|(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|↵
(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}(?![:.\w]))↵
(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:))(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Из-за наличия нескольких форм записи сопоставление с адресами IPv6 реализовать не так просто, как сопоставление с адресами IPv4. Выбор допустимых форм записи оказывает большое влияние на сложность регулярного выражения. Обычно используются две формы записи: стандартная и смешанная. Если вы решите считать допустимой только одну из них, то получите три набора регулярных выражений.

И стандартная, и смешанная нотации имеют компактную форму записи, в которой опускаются нулевые значения. Поддержка компактной формы записи дает еще три набора регулярных выражений.

Для проверки, является ли заданная строка допустимым адресом IPv6, и для поиска IP-адресов в текстовом документе потребуются немного отличающиеся регулярные выражения. При проверке допустимости IP-адреса мы использовали якорные метасимволы, описываемые в рецепте 2.5. Диалект JavaScript использует якорные метасимволы ‹^› и ‹$›, тогда как диалект Ruby использует якорные метасимволы ‹\A› и ‹\Z›. Все остальные диалекты поддерживают оба набора якорных метасимволов. Диалект Ruby также поддерживает метасимволы ‹^› и ‹$›, но допускает их совпадение с символами разрыва строки, присутствующими в тексте. При использовании диалекта Ruby символ крышки и знак доллара должны применяться, только если испытуемый текст не содержит разрывов строк.

Для обеспечения возможности отыскивать адреса IPv6 в текстовом документе мы использовали негативную ретроспективную проверку ‹(?<![:.\w])› и негативную опережающую проверку ‹(?![:.\w])›, чтобы убедиться в отсутствии символа слова (буква, цифра или символ подчеркивания), двоеточия или точки перед адресом и после него. Это гарантирует, что совпадение не будет обнаружено в длинной последовательности цифр и двоеточий. Порядок работы ретроспективной и опережающей проверок описывается в рецепте 2.16. Если проверка ближайших символов недоступна, то для проверки отсутствия символа

слова перед адресом и после него можно использовать границы слова, но только если первый и последний символ адреса являются (шестнадцатеричной) цифрой. В компактной форме записи адреса могут начинаться и заканчиваться символом двоеточия. В этом случае, чтобы получить совпадение с границей слова перед двоеточием или после него, потребовалось бы наличие соседней буквы или цифры, что совсем не то, что нам требуется. Полное обсуждение границ слова приводится в рецепте 2.6.

Стандартная форма записи

Стандартная форма записи адресов IPv6 достаточно просто поддается обработке с помощью регулярного выражения. В этом случае необходимо обеспечить совпадение с восемью словами в шестнадцатеричной нотации, разделенных двоеточиями. Выражение `‹[A-F0-9]{1,4}›` совпадает с последовательностью от 1 до 4 шестнадцатеричных символов, которые составляют 16-битное слово с необязательными ведущими нулями. В символьном классе (рецепт 2.3) перечислены только символы верхнего регистра. Совпадение с символами нижнего регистра обеспечивается за счет режима нечувствительности к регистру символов. Порядок установки режимов в различных языках программирования описывается в рецепте 3.4.

Несохраняющая группа `‹(?:[A-F0-9]{1,4}:){7}›` совпадает с шестнадцатеричным словом, за которым следует символ двоеточия. Квантификатор повторяет группу семь раз. Первое двоеточие в этом регулярном выражении является частью синтаксиса определения несохраняющей группы, как описывается в рецепте 2.9, а второй – это литерал двоеточия. Символ двоеточия интерпретируется в регулярных выражениях как метасимвол только в некоторых ситуациях, когда он является частью лексемы регулярного выражения. Поэтому нет необходимости экранировать литерал двоеточия символом обратного слэша. Мы могли бы экранировать его, но это только сделало бы регулярное выражение менее удобочитаемым.

Смешанная форма записи

Регулярное выражение для сопоставления с адресами IPv6 в смешанной форме записи состоит из двух частей. Подвыражение `‹(?:[A-F0-9]{1,4}:){6}›` совпадает с шестью шестнадцатеричными словами, за каждым из которых следует символ двоеточия, точно такое же подвыражение используется для сопоставления с последовательностью из семи шестнадцатеричных слов в стандартной форме записи.

Теперь вместо еще одного шестнадцатеричного слова в конце адреса присутствует полный адрес IPv4. Для сопоставления с ним мы использовали «точное» регулярное выражение из рецепта 7.16.

Стандартная или смешанная форма записи

Чтобы обеспечить совпадение и со стандартной, и со смешанной формами записи, требуется более длинное регулярное выражение. Эти формы записи отличаются только представлением последних 32 битов в адресе IPv6. В стандартной форме записи используются два 16-битовых слова, тогда как в смешанной – 4 десятичных байта, как в адресе IPv4.

Первая часть регулярного выражения совпадает с шестью шестнадцатеричными словами, как и в регулярном выражении, поддерживающем только смешанную форму записи. Вторая часть регулярного выражения теперь представлена несохраняющей группой с двумя альтернативами представления последних 32 битов. Как описывается в рецепте 2.8, оператор выбора (вертикальная черта) имеет самый низкий приоритет среди всех операторов регулярных выражений. Поэтому пришлось использовать несохраняющую группировку, чтобы исключить из выбора шесть слов.

Первая альтернатива, слева от вертикальной черты, совпадает с двумя шестнадцатеричными словами и символом двоеточия между ними. Вторая альтернатива совпадает с адресом IPv4.

Компактная форма записи

Когда необходимо обеспечить совпадение с компактной формой записи, дело осложняется еще больше. Причина в том, что компактная форма записи позволяет опускать переменное число нулей. Так, `1:0:0:0:0:6:0:0`, `1::6:0:0` и `1:0:0:0:0:6::` – это три разных способа записи одного и того же адреса IPv6. Адрес может содержать до восьми слов, но может и не содержать ни одного. Если в адресе присутствует менее восьми слов, в нем должна иметься одна последовательность из двух символов двоеточия, представляющая опущенные нулевые значения.

Переменное число повторений легко реализовать в регулярных выражениях. Если в адресе IPv6 имеется пара двоеточий, следующих друг за другом, то в адресе может присутствовать не более семи слов перед и после двойного двоеточия. Это легко записать следующим образом:

```
(
 ([0-9A-F]{1,4}:){1,7} # от 1 до 7 слов слева
| :                    # или двойное двоеточие в начале
)
(
 (:[0-9A-F]{1,4}){1,7} # от 1 до 7 слов слева
| :                    # или двойное двоеточие в конце
)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это регулярное выражение совпадает со всеми адресами IPv6 в компактной форме записи, но оно не будет совпадать с адресами в стандартной, некомпактной форме.

Это регулярное выражение, как и следующие ниже в этом разделе, также будут работать и в диалекте JavaScript, если убрать из него комментарии и лишние пробельные символы. Диалект JavaScript поддерживает все функциональные возможности, задействованные в этих регулярных выражениях, за исключением режима свободного форматирования, который мы используем здесь, чтобы упростить понимание регулярных выражений.

Это очень простое регулярное выражение. Первая его часть совпадает с последовательностью от 1 до 7 слов, за которой следует символ двоеточия, или с двоеточием для адресов, в которых нет ни одного слова левее двойного двоеточия.

Вторая часть совпадает с последовательностью от 1 до 7 слов, которой предшествует символ двоеточия, или с двоеточием для адресов, в которых нет ни одного слова правее двойного двоеточия. Все вместе обеспечивает допустимость совпадения с самим двоеточием, с двоеточием и последовательностью от 1 до 7 слов только слева, с двоеточием и последовательностью от 1 до 7 слов только справа, и с двоеточием и последовательностью от 1 до 7 слов слева и справа.

Эта последняя часть доставляет определенные неприятности. Регулярное выражение допускает совпадение с последовательностью от 1 до 7 слов как слева, так и справа, как собственно и должно быть, но оно не позволяет указать, что суммарное количество слов слева и справа не должно превышать 7. В адресе IPv6 должно быть 8 слов. Двойное двоеточие свидетельствует, что было опущено, по крайней мере, одно слово, поэтому общее число слов не должно превышать 7.

Регулярные выражения не производят математических вычислений. Они могут ограничить появление некоторого фрагмента от 1 до 7 раз. Но они не в состоянии ограничить суммарное появление двух фрагментов, распределив эти 7 раз между двумя фрагментами в какой-нибудь пропорции.

Чтобы лучше понять эту проблему, рассмотрим простую аналогию. Допустим, что необходимо обеспечить совпадение с чем-то, что записывается в формате `aaaaxbbb`. Строка должна иметь длину от 1 до 8 символов и содержать от 0 до 7 символов `a`, точно один символ `x` и от 0 до 7 символов `b`.

Решить эту проблему с помощью регулярных выражений можно двумя способами. Один из них заключается в том, чтобы описать все возможные альтернативы. Этот способ используется в следующем разделе, где обсуждается сопоставление с компактной смешанной формой записи. Это может привести к созданию весьма длинного регулярного выражения, но его легко понять.

```
\A(?:a{7}x
 | a{6}xb?
 | a{5}xb{0,2}
 | a{4}xb{0,3}
 | a{3}xb{0,4}
 | a{2}xb{0,5}
 | axb{0,6}
 | xb{0,7}
)\Z
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В этом регулярном выражении для каждого возможного числа символов a предусмотрена своя альтернатива. Каждая альтернатива описывает допустимое количество символов b, после того как было найдено совпадение с заданным числом символов a и символом x. Другое решение состоит в том, чтобы использовать опережающую проверку. Этот метод используется в регулярном выражении, в разделе «Решение», совпадающем с адресом IPv6 в компактной форме записи. Те, кто не знаком с с опережающей проверкой, могут сначала ознакомиться с рецептом 2.16. Опережающая проверка обеспечивает возможность сопоставления с одним и тем же фрагментом текста дважды, с проверкой соблюдения одного из двух условий.

```
\A
  (?=[abx]{1,8}\Z)
  a{0,7}xb{0,7}
\Z
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Якорный метасимвол ‹\A› в начале регулярного выражения привязывает его к началу испытуемого текста. Далее следует позитивная опережающая проверка. Она проверяет возможность совпадения с последовательностью от 1 до 8 букв ‹a›, ‹b› и/или ‹x›, с тем, что эта последовательность простирается до конца текста. Метасимвол ‹\Z› внутри опережающей проверки имеет крайне важное значение. Чтобы регулярное выражение могло ограничить длину строки восемью символами, опережающая проверка должна убедиться в отсутствии дополнительных символов после совпавших.

В другой ситуации вместо ‹\A› и ‹\Z› можно было бы использовать другие символы-ограничители. Если необходимо было бы выполнить поиск последовательности aaaaxbbb и ее разновидностей «только как целых слов», можно было бы использовать границы слов. Но в любом случае, чтобы ограничить длину совпадения, необходимо использовать какой-то ограничитель, и этот ограничитель, совпадающий с концом строки, необходимо поместить и в опережающую проверку, и в конец

регулярного выражения. Если этого не сделать, регулярное выражение будет совпадать с частью более длинной строки.

Когда опережающая проверка удовлетворит свои требования, она вернет обратно совпавшие с ней символы. То есть, когда механизм регулярных выражений перейдет к подвыражению `‹a{0,7}›`, он окажется в начале испытуемого текста. Тот факт, что опережающая проверка не поглощает текст, совпавший с ней, является главным ее отличием от несохраняющей группы и позволяет применять два шаблона к одному и тому же фрагменту текста.

Несмотря на то, что подвыражение `‹a{0,7}xb{0,7}›` само по себе могло бы совпасть с последовательностью длиной до 15 символов, тем не менее, в данном случае оно может совпасть только с 8-ю, потому что опережающая проверка уже гарантирует наличие всего 8 символов. Все, что остается на долю подвыражения `‹a{0,7}xb{0,7}›`, – это проверка правильного порядка следования символов. При этом подвыражение `‹a*xb*›` могло бы обеспечить тот же эффект, что и подвыражение `‹a{0,7}xb{0,7}›`.

Второй метасимвол `‹\Z›` в конце регулярного выражения тоже имеет важное значение. Если в опережающей проверке необходимо проверить отсутствие лишних символов, то во второй проверке необходимо гарантировать верный порядок следования символов. Это дает уверенность, что регулярное выражение не совпадет с чем-нибудь вроде `axba`, когда длина текста находится в диапазоне от 1 до 8 символов и удовлетворяет требованиям опережающей проверки.

Компактная смешанная форма записи

Смешанную форму записи, так же как и стандартную, можно сжать. Несмотря на то, что четыре байта в конце должны быть указаны всегда, даже когда они равны нулю, количество шестнадцатеричных слов перед ними может быть переменным. Если все шестнадцатеричные слова равны нулю, адрес IPv6 может в конечном итоге выглядеть, как адрес IPv4, которому предшествуют два двоеточия.

При создании регулярного выражения, совпадающего с адресами в компактной смешанной форме записи, приходится решать те же проблемы, что и в случае с компактной стандартной формой записи. Все они были описаны в предыдущем разделе.

Главное отличие между регулярными выражениями для сопоставления с компактной смешанной и с компактной (стандартной) формами записи заключается в том, что при сопоставлении с адресами в компактной смешанной нотации необходимо проверять наличие адреса IPv4 после шести шестнадцатеричных слов. Эту проверку мы выполняем в конце регулярного выражения, используя точное выражение из рецепта 7.16, совпадающее с адресами IPv4, использовавшееся в этом рецепте, в выражении для сопоставления с некомпактной смешанной формой записи.

Проверка совпадения с адресом IPv4 в конце выражения необходима, но точно так же она необходима и внутри опережающей проверки, чтобы убедиться, что в адресе IPv6 присутствует не более шести двоеточий или шести шестнадцатеричных слов. Так как точное сопоставление уже выполняется в конце регулярного выражения, в опережающей проверке вполне достаточно простой проверки совпадения с адресом IPv4. Точная проверка адреса IPv4 в опережающей проверке не требуется, так как это делает основное регулярное выражение. Но она должна обеспечивать совпадение с частью IPv4, чтобы якорный символ, совпадающий с концом текста внутри опережающей проверки, мог выполнить свою работу.

Стандартная, смешанная или компактная форма записи

Заключительный набор регулярных выражений объединяет в себе все вышесказанное. Они совпадают с адресами IPv6 в любой форме записи: стандартной или смешанной, компактной или нет.

Регулярные выражения в этом наборе составлены из выражений для компактной смешанной и компактной (стандартной) форм записи с применением конструкции выбора. В них, в свою очередь, также используются конструкции выбора для сопоставления с компактной и некомпактной формами записи адресов IPv6.

В результате получилось регулярное выражение с тремя конструкциями выбора на верхнем уровне, первая из которых в свою очередь содержит два варианта. Первая конструкция соответствует адресам IPv6 в смешанной форме записи, как компактной, так и некомпактной. Вторая конструкция соответствует адресам IPv6 в стандартной форме записи. Третья конструкция соответствует адресам в компактной (стандартной) форме записи.

Мы создали три конструкции выбора верхнего уровня вместо двух, со своими собственными конструкциями выбора, потому что нет никаких причин группировать в конструкцию выбора стандартную и компактную формы записи. В случае смешанной формы записи необходимость объединения компактной и некомпактной форм записи обусловлена возможностью сэкономить на проверке части адреса IPv4.

Фактически мы объединили выражение:

```
^(6words|compressed6words)ip4$
```

и выражение:

```
^(8words|compressed8words)$
```

в выражение:

```
^((6words|compressed6words)ip4|8words|compressed8words)$
```

вместо:

```
^((6words|compressed6words)ip4|(8words|compressed8words))$
```

См. также

Рецепты 2.16 и 7.16.

7.18. Проверка путей в Windows

Задача

Необходимо проверить, является ли некоторая строка допустимой строкой пути к папке или к файлу в операционной системе Microsoft Windows.

Решение

Буква устройства в пути

```
\A
(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\ # Устройство
(?:[^\\/:*?"<>|\r\n]+\\)*                   # Папка
[^\\/:*?"<>|\r\n]*                          # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\(?:[^\\/:*?"<>|\r\n]+\\)*↵
[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства и пути в формате UNC

```
\A
(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\| # Устройство
(?:[^\\/:*?"<>|\r\n]+\\)*                    # Папка
[^\\/:*?"<>|\r\n]*                           # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\(?:[^\\/:*?"<>|\r\n]+\\)*↵
[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства, UNC и относительные пути

```
\A
(?:(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\| # Устройство
\\?[^\\/:*?"<>|\r\n]+\\?)                       # Относительный путь
(?:[^\\/:*?"<>|\r\n]+\\)*                       # Папка
[^\\/:*?"<>|\r\n]*                              # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\|\\?[^\\/:*?"<>|\r\n]+\\?)↵
(?:[^\\/:*?"<>|\r\n]+\\)*[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Буква устройства в пути

Сопоставление с полным путем к файлу или папке на устройстве, в котором присутствует буква устройства, выполняется очень просто. Устройство обозначается единственной буквой, за которой следуют двоеточие и обратный слэш. Это совпадение обеспечивает простое выражение ‹[a-z]:\\›. Обратный слэш является метасимволом в регулярных выражениях, поэтому, чтобы обеспечить совпадение с литералом, его необходимо экранировать другим обратным слэшем.

Имена папок и файлов в Windows могут содержать любые символы, за исключением следующих: \/:*?"‹›|. Разрывы строк в именах недопустимы. Мы легко можем описать совпадение с любым символом, за исключением указанных, с помощью инвертированного символьного класса ‹[^\\/:*?"‹›|\r\n]+›. Обратный слэш в символьных классах также является метасимволом, поэтому его необходимо экранировать. Последовательности ‹\r› и ‹\n› – это два символа разрыва строки. Подробнее о символьных классах рассказывается в рецепте 2.3. Квантификатор «плюс» (рецепт 2.12) указывает, что требуется совпадение с одним или более символами.

Имена папок в пути отделяются друг от друга обратными слэшами. Совпадение с последовательностью из нуля или более имен папок можно описать с помощью выражения ‹(?:[^\\/:*?"‹›|\r\n]+\\)*›, которое помещает выражение, соответствующее имени папки, и литерал обратного слэша в несохраняющую группу (рецепт 2.9) , повторяемую ноль или более раз с помощью звездочки (рецепт 2.12).

Для сопоставления с именем файла используется выражение ‹[^\\/:*?"‹›|\r\n]*›. Звездочка обеспечивает необязательность имени файла, благо-

даря чему допускается завершение пути символом обратного слэша. Если необходимо, чтобы обратный слэш отсутствовал в конце пути, нужно заменить последний символ ‹*› в регулярном выражении на ‹+›.

Буква устройства и пути в формате UNC

Пути к файлам на сетевых устройствах, которые не отображаются на буквы устройств, можно записывать в соответствии с универсальным соглашением об именах (Universal Naming Convention, UNC). Пути в формате UNC имеют вид *\\server\share\folder\file*.

Мы легко можем адаптировать регулярное выражение, совпадающее с путями, включающими букву устройства, для сопоставления с путями в формате UNC. Для этого достаточно заменить подвыражение ‹[a-z]:›, совпадающее с буквой устройства, на подвыражение, совпадающее с буквой устройства или с именем сервера.

Такое совпадение обеспечивает выражение ‹(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)›. Вертикальная черта – это оператор выбора (рецепт 2.8). Он реализует выбор между сопоставлением ‹[a-z]:› с буквой устройства и сопоставлением ‹\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+› с именем сервера и разделяемого ресурса. Оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений. Чтобы объединить два варианта, мы использовали несохраняющую группировку. Как объясняется в рецепте 2.9, символы ‹(?:› образуют сложного вида открывающую скобку несохраняющей группы. Вопросительный знак после открывающей круглой скобки теряет свой обычный смысл.

Остальная часть регулярного выражения может оставаться той же самой. Имя разделяемого ресурса в формате UNC будет совпадать с частью регулярного выражения, которая соответствует именам папок.

Буква устройства, UNC и относительные пути

Относительный путь начинается с имени папки (возможно, со специального имени папки ..., соответствующего родительской папке) или содержит единственное имя файла. Чтобы обеспечить поддержку относительных путей, необходимо добавить третью конструкцию в раздел регулярного выражения, совпадающего с «устройством». Эта конструкция должна совпадать с началом относительного пути, вместо буквы устройства или имени сервера.

Выражению ‹\\?[^\\/:*?"‹›|\r\n]+\\?› соответствует начало относительного пути. Путь может начинаться с символа обратного слэша, но это необязательно. Подвыражение ‹\\?› совпадет с символом обратного слэша, если он присутствует, или ни с чем в противном случае. Подвыражение ‹[^\\/:*?"‹›|\r\n]+› совпадает с именем папки или файла. Если относительный путь содержит только имя файла, заключительное подвыражение ‹\\?› не будет совпадать ни с чем, так же как и обе части регулярного выражения, соответствующие «папке» или «файлу», каж-

дая из которых является необязательной. Если относительный путь определяет каталог, заключительное подвыражение ‹\\?› будет совпадать с обратным слэшем, отделяющим первую папку в относительном пути от остальной части пути. Часть регулярного выражения, соответствующая «папке», в этом случае совпадет с остальными именами папок в пути, если они присутствуют, а часть регулярного выражения, соответствующая «файлу», совпадет с именем файла.

Регулярное выражение, соответствующее относительному пути, уже не имеет отдельных частей, соответствующих различным компонентам испытуемого текста. Часть регулярного выражения, помеченная как «относительный путь», фактически будет совпадать с именем папки или файла, если путь относительный. Если в относительном пути присутствует одно или более имен папок, часть регулярного выражения «относительный путь» совпадет с первой папкой, а части «папка» и «файл» совпадут с тем, что останется. Если относительный путь состоит только из имени файла, ему будет соответствовать часть «относительный путь», а на долю частей «папка» и «файл» ничего не останется. Так как нас интересует только проверка корректности пути, это обстоятельство не имеет никакого значения. Комментарии в регулярном выражении играют роль меток, чтобы помочь вам лучше понять его.

Если потребуется извлекать отдельные части пути в сохраняющие группы, то придется приложить больше усилий, чтобы обеспечить совпадение с буквой устройства, именем папки или файла по отдельности. Эта проблема решается в следующем рецепте.

См. также

Рецепты 2.3, 2.8, 2.9 и 2.12.

7.19. Выделение элементов путей в Windows

Задача

Необходимо проверить, выглядит ли испытуемый текст, как допустимый путь к папке или к файлу в операционной системе Microsoft Windows. Если окажется, что испытуемый текст хранит допустимый путь Windows, то необходимо извлечь букву устройства, часть пути с именами папок и имя файла по отдельности.

Решение

Буква устройства в пути

```
\A
(?<drive>[a-z]:)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
```

```
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
([a-z]:)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:)\\((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства и пути в формате UNC

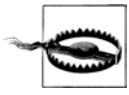
```
\A
(?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

Буква устройства, UNC и относительные пути



Следующие регулярные выражения могут совпадать с пустой строкой. Дополнительные сведения и альтернативное решение приводятся в разделе «Пояснения».

```
\A
(?<drive>[a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
([a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)↵
((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Регулярные выражения в этом рецепте очень похожи на выражения в предыдущем рецепте. Здесь предполагается, что вы уже прочитали и поняли пояснения, которые приводились в предыдущем рецепте.

Буква устройства в пути

В регулярное выражение, совпадающее со строкой пути, включающей букву устройства, было внесено единственное изменение по сравнению

с выражением из предыдущего рецепта. Мы добавили три сохраняющие группы, которые можно использовать для извлечения различных частей пути: `‹drive›` (устройство), `‹folder›` (папка) и `‹file›` (файл). Если используемый диалект регулярных выражений поддерживает именованные сохранения (рецепт 2.11), можно использовать эти имена. В противном случае можно ссылаться на сохраняющие группы по их номерам: 1, 2 и 3. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Буква устройства и пути в формате UNC

В выражение, совпадающее с путями в формате UNC, мы добавили те же три сохраняющие группы.

Буква устройства, UNC и относительные пути

Регулярное выражение становится сложнее, когда возникает необходимость обеспечить совпадение с относительными путями. В предыдущем рецепте просто была добавлена третья конструкция в часть выражения, совпадающую с началом относительного пути. В данном случае это невозможно. При совпадении с относительными путями сохраняющая группа, соответствующая букве устройства, должна оставаться пустой.

На этот раз символ обратного слэша, следовавший в выражении из раздела «буква устройства и пути в формате UNC» вслед за сохраняющей группой с буквой устройства, переместился в эту сохраняющую группу. Мы добавили его в конец конструкций, соответствующих букве устройства и сетевому имени разделяемого ресурса. Мы также добавили третью конструкцию с необязательным обратным слэшем для совпадения с относительными путями, которые могут начинаться с символа обратного слэша. Так как третья конструкция является необязательной, то и вся группа, совпадающая с устройством, фактически становится необязательной.

Получившееся регулярное выражение будет совпадать с любыми путями в операционной системе Windows. Проблема состоит в том, что сделав необязательной часть, совпадающую с устройством, мы получили регулярное выражение, в котором все части являются необязательными. Части, совпадающие с именами папок и файлов, уже были необязательными в выражениях, поддерживавших только абсолютные пути. Другими словами: это регулярное выражение может совпадать с пустой строкой.

Если необходимо гарантировать невозможность совпадения с пустой строкой, необходимо добавить дополнительные альтернативы для работы с относительными путями, которые определяют папку (когда имя файла является необязательным), и с относительными путями, которые не определяют папку (когда имя файла является обязательным):

```
\A
(?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file2>[^\\/:*?"<>|\r\n]*)
| (?<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?:
  (?P<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?P<file>[^\\/:*?"<>|\r\n]*)
| (?P<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?P<file2>[^\\/:*?"<>|\r\n]*)
| (?P<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
(?:
  ([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  ((?:[^\\/:*?"<>|\r\n]+\\)*)
  ([^\\/:*?"<>|\r\n]*)
| (\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  ([^\\/:*?"<>|\r\n]*)
| ([^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\((?:[^\\/:*?"<>|\r\n]+\\)*)↵
([^\\/:*?"<>|\r\n]*)|(\\?(?:[^\\/:*?"<>|\r\n]+\\)+)([^\\/:*?"<>|\r\n]*)|↵
([^\\/:*?"<>|\r\n]+))$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Из-за того, что была исключена возможность совпадения с пустой строкой, мы получили шесть сохраняющих групп, которые сохраняют три различные части пути. В каждом конкретном случае следует определиться заранее, что будет проще – выполнить дополнительную проверку на отсутствие символов в строке перед применением этого выражения или приложить дополнительные усилия для обработки нескольких сохраняющих групп после того, как совпадение уже будет найдено.

Диалект .NET допускает возможность присваивания одного и того же имени нескольким именованным группам. Однако диалект .NET единственный способен обрабатывать группы с одинаковыми именами, как если бы это была единственная сохраняющая группа. Используя следующее регулярное выражение в диалекте .NET, можно просто извлекать совпадение с группой ‹folder› (папка) или ‹file› (файл), не беспокоясь о том, какая из двух групп ‹folder› или какая из трех групп ‹file› фактически участвовала в совпадении:

```
\A
(?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<folder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<file>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET

См. также

Рецепты 2.9, 2.11, 3.9 и 7.18.

7.20. Извлечение буквы устройства из путей в Windows

Задача

Имеется строка, в которой хранится (синтаксически) допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь букву устройства, если таковая присутствует. Например, необходимо извлечь букву c из пути c:\folder\file.ext.

Решение

```
^([a-z]):
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение буквы устройства из строки, которая заведомо содержит допустимый путь, является тривиальной задачей, даже если заранее не известно, действительно ли путь начинается с буквы устройства. Путь может быть или относительным, или в формате UNC.

Двоеточие в строках путей в системе Windows является недопустимым символом, за исключением случая, когда оно используется для отделения буквы устройства. То есть, если в начале строки имеется буква, за которой следует двоеточие, можно быть уверенным, что эта буква является буквой устройства.

Якорный метасимвол ‹^› совпадает с началом строки (рецепт 2.5). Тот факт, что в диалекте Ruby символ крышки также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Символьный класс ‹[a-z]› совпадает с единственной буквой (рецепт 2.3). Мы поместили символьный класс между парой круглых скобок (образующих сохраняющую группу), благодаря чему получили возможность извлекать букву устройства без символа двоеточия, который также совпадает с регулярным выражением. Мы добавили двоеточие в регулярное выражение, чтобы гарантировать совпадение с буквой устройства, а не с первой буквой в относительном пути.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Рецепт 7.19, если заранее не известно, хранит ли испытуемая строка допустимый путь Windows.

7.21. Извлечение имени сервера и разделяемого ресурса из пути в формате UNC

Задача

Имеется строка, в которой хранится (синтаксически) допустимый путь к файлу или папке на диске в системе Windows или в сети. Если путь

указан в формате UNC, необходимо извлечь имя сетевого сервера и имя разделяемого ресурса на сервере, содержащиеся в пути. Например, необходимо извлечь server и share из пути \\server\share\folder\file.ext.

Решение

```
^\\\\([a-z0-9_.$]+)\\([a-z0-9_.$]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени сетевого сервера и разделяемого ресурса из строки, которая заведомо содержит допустимый путь, является простой задачей, даже если заранее не известно, действительно ли строка содержит путь в формате UNC. Путь может быть относительным или начинаться с буквы устройства.

Пути в формате UNC начинаются с двух символов обратного слэша. Два обратных слэша в строках путей в системе Windows являются недопустимыми, за исключением случая, когда они стоят в начале пути в формате UNC. То есть, если в начале строки имеются два символа обратного слэша, можно быть уверенным, что за ними следуют имя сервера и разделяемого ресурса

Якорный метасимвол ‹^› совпадает с началом строки (рецепт 2.5). Тот факт, что в диалекте Ruby символ крышки также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Подвыражение ‹\\\\› совпадает с двумя литералами обратного слэша. Так как обратный слэш в регулярных выражениях является метасимволом, его необходимо экранировать другим символом обратного слэша, чтобы обеспечить совпадение с литералом. Первый символьный класс ‹[a-z0-9_.$]+› совпадает с именем сетевого сервера. Второй, следующий за другим литералом обратного слэша, совпадает с именем разделяемого ресурса. Мы поместили символьные классы между парами круглых скобок, образующими сохраняющие группы, чтобы иметь возможность извлекать имя сервера из первой сохраняющей группы, а имя разделяемого ресурса – из второй. Все регулярное выражение будет совпадать с фрагментом \\server\share.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Рецепт 7.19, если заранее не известно, хранит ли испытуемая строка допустимый путь Windows.

7.22. Извлечение имен папок из путей в Windows

Имеется строка, в которой хранится (синтаксически) допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути имя папки. Например, необходимо извлечь фрагмент `\folder\subfolder\` из пути `c:\folder\subfolder\file.ext` или из пути `\\server\share\folder\subfolder\file.ext`.

Решение

```
^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?((?:\\|^)↵
(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени папки из строки, содержащей путь в Windows, может оказаться непростым делом, если требуется обеспечить поддержку формата UNC, потому что нельзя просто извлечь часть пути между символами обратного слэша. В этом случае в совпадение попадут имя сетевого сервера и имя разделяемого ресурса.

Первая часть регулярного выражения, ‹`^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?`›, перепрыгивает через букву устройства или имя сетевого сервера и разделяемого ресурса, находящиеся в начале пути. Эта часть регулярного выражения состоит из сохраняющей группы с двумя конструкциями. Первая конструкция совпадает с буквой устройства, как описывается в рецепте 7.20, а вторая – с именем сервера и разделяемого ресурса в пути, представленном в формате UNC, как описывается в рецепте 7.21. Оператор выбора описывается в рецепте 2.8.

Знак вопроса, следующий за группой, делает ее необязательной. Это позволяет обеспечить поддержку относительных путей, в которых отсутствует буква устройства или имя разделяемого сетевого ресурса.

Совпадение с папками легко обеспечить с помощью подвыражения ‹`(?:[^\\/:*?"<>|\r\n]+\\)+`›. Символьный класс совпадает с именем папки. Несохраняющая группа совпадает с именем папки, за которым следует символ обратного слэша, отделяющий папки друг от друга и от имени файла. Группа повторяется один или более раз. Это означает, что данное регулярное выражение будет совпадать только с теми путями, в которых действительно присутствует имя папки. Пути, в которых указывается только имя файла, устройство или имя сетевого ресурса, совпадать с регулярным выражением не будут.

Если путь начинается с буквы устройства или с имени сетевого ресурса, за ними должен следовать символ обратного слэша. Относительные пути могут начинаться с обратного слэша. Поэтому в начало группы,

в части выражения, совпадающей с папкой, необходимо добавить необязательный обратный слэш. Так как данное регулярное выражение предполагается использовать только с заведомо допустимыми путями, нет необходимости требовать наличия обратного слэша в случае наличия в пути буквы устройства или имени сетевого ресурса. Достаточно лишь допустить его присутствие.

Регулярное выражение должно совпадать, по меньшей мере, с одной папкой, поэтому мы обеспечили невозможность совпадения с фрагментом e\, как с именем папки, в пути \\server\share\. Кроме того, чтобы добавить совпадение с необязательным обратным слэшем в начало сохраняющей группы, совпадающей с именем папки, мы использовали подвыражение ‹(\\|^)›, а не ‹\\?›.

Если вам интересно узнать, почему фрагмент \\server\shar может совпадать как буква устройства, а фрагмент e\ – как имя папки, загляните в рецепт 2.13. Это обусловлено тем, что механизмы регулярных выражений выполняют возвраты. Рассмотрим следующее регулярное выражение:

```
^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?↵
((?:\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Это регулярное выражение, так же, как и регулярное выражение, представленное в решении, требует наличия в пути хотя бы одного символа, отличного от обратного слэша, и одного обратного слэша. Если регулярное выражение обнаружит в пути \\server\share совпадение \\server\share с буквой устройства и затем потерпит неудачу при сопоставлении группы, совпадающей с именем папки, оно не остановится на достигнутом, а попытается выполнить различные перестановки.

В этом случае механизм вспомнит, что символьный класс ‹[a-z0-9_.$]+›, соответствующий имени сетевого ресурса, необязательно должен соответствовать всем доступным символам. Чтобы удовлетворить квантификатор ‹+›, достаточно одного символа. Механизм выполнит возврат, заставив символьный класс уступить один символ, и повторит попытку.

Когда механизм продолжит сопоставление, в испытуемой строке останется два символа, способных обеспечить совпадение с папкой: e\. Этих двух символов вполне достаточно, чтобы удовлетворить подвыражение ‹(?:[^\\/:*?"‹›|\r\n]+\\)+›, в результате чего будет обеспечено совпадение для всего регулярного выражения. Но это не то совпадение, которое ожидалось.

Использовав подвыражение ‹(\\|^)› вместо ‹\\?›, мы решили эту проблему. Оно также допускает наличие обратного слэша, но в случае его отсутствия оно требует, чтобы в начале строки находилось имя папки. Это означает, что если будет обнаружено совпадение с буквой устройства и механизм регулярных выражений таким образом продвинется

дальше начала строки, наличие обратного слэша становится обязательным. Если далее не будет найдено совпадение ни с одной папкой, механизм регулярных выражений также попытается выполнить возврат, но безуспешно, потому что сопоставление с подвыражением ‹(\\|^)› потерпит неудачу. Механизм регулярных выражений будет производить возвраты, пока не достигнет начала строки. Сохраняющая группа, совпадающая с буквой устройства и именем сетевого ресурса, является необязательной, поэтому механизм регулярных выражений сможет попытаться выполнить сопоставление для части выражения, совпадающей с именем папки, с начала строки. Хотя подвыражение ‹(\\|^)› сможет обнаружить там совпадение, остальная часть регулярного выражения не допустит этого, потому что подвыражение ‹(?:[^\\/:*?"‹›|\r\n]+\\)+› делает невозможным совпадение с двоеточием, следующим за буквой устройства или двойного обратного слэша, с которого начинается имя сетевого ресурса.

Мы не использовали этот прием в рецептах 7.18 и 7.19, потому что в тех регулярных выражениях совпадение с папкой было необязательным. Так как в тех регулярных выражениях все, что следовало за частью, совпадающей с устройством, объявлено необязательным, механизм регулярных выражений никогда не будет выполнять никаких возвратов. Конечно, такая необязательность может привести к другим проблемам, как описывается в рецепте 7.19.

Когда регулярное выражение обнаружит совпадение, первая сохраняющая группа будет содержать букву устройства или имя сетевого ресурса, а вторая сохраняющая группа – имя папки. При совпадении с относительным путем первая сохраняющая группа будет содержать пустую строку. Вторая сохраняющая группа всегда будет содержать хотя бы одно имя папки. Если применить это регулярное выражение к пути, в котором папка отсутствует, оно вообще не будет обнаруживать совпадение.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Рецепт 7.19, если заранее не известно, хранит ли испытуемая строка допустимый путь Windows.

7.23. Извлечение имени файла из пути Windows

Имеется строка, в которой хранится (синтаксически) допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути имя файла. Например, необходимо извлечь фрагмент file.ext из пути c:\folder\file.ext.

Решение

```
[^\\/:*?"<>|\r\n]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени файла из строки, которая заведомо содержит допустимый путь, является тривиальной задачей, даже если заранее неизвестно, действительно ли путь оканчивается именем файла.

Имя файла всегда находится в конце строки. Оно не может содержать символы двоеточия или обратного слэша, поэтому его невозможно перепутать с именем папки, с буквой устройства или с именем сетевого ресурса, в которых используются символы обратного слэша и/или двоеточия.

Якорный метасимвол `‹$›` совпадает с концом строки (рецепт 2.5). Тот факт, что в диалекте Ruby знак доллара также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Инвертированный символьный класс `‹[^\\/:*?"‹›|\r\n]+›` (рецепт 2.3) совпадает с символами, которые могут присутствовать в именах файлов. Несмотря на то, что механизм регулярных выражений сканирует испытуемую строку в направлении слева направо, якорь в конце регулярного выражения гарантирует, что только совпадение с последними символами в строке будет интерпретироваться как имя файла.

Если испытуемая строка оканчивается обратным слэшем, как в путях, где не указываются имена файлов, регулярное выражение вообще не будет обнаруживать совпадение. Когда оно действительно будет обнаруживать соответствие, совпадение будет содержать только имя файла, поэтому нет необходимости в использовании сохраняющей группы для отделения имени файла от остальной части пути.

См. также

Рецепт 3.7, где можно узнать, как извлекать текст, совпавший с регулярным выражением, в различных языках программирования.

Рецепт 7.19, если заранее не известно, хранит ли испытуемая строка допустимый путь Windows.

7.24. Извлечение расширения имени файла из пути Windows

Имеется строка, в которой хранится (синтаксически) допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути расширение имени файла, если оно имеется. Например, необходимо извлечь фрагмент `.ext` из пути `c:\folder\file.ext`.

Решение

```
\.[^.\\/:*?"<>|\r\n]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Для извлечения расширения имени файла можно использовать тот же прием, который применялся для извлечения всего имени файла в рецепте 7.23.

Единственное отличие заключается в том, как обрабатывать точку. Регулярное выражение в рецепте 7.23 не включает никаких точек. Инвертированный символьный класс в выражении из рецепта 7.23 просто обеспечивает совпадение с любыми точками, которые могут встретиться в имени файла.

Расширение имени файла начинается с точки. Поэтому мы добавили ‹`\.`› в начало регулярного выражения, чтобы обеспечить совпадение с точкой.

Такие имена файлов, как `Version 2.0.txt`, могут содержать несколько точек. Последняя точка является той, что отделяет расширение от имени файла. Само расширение не может содержать точек. Мы указали на это обстоятельство, поместив точку в символьный класс. Внутри символьных классов точка играет роль обычного литерала, поэтому ее можно не экранировать. Якорь ‹`$`› в конце регулярного выражения гарантирует, что с выражением будет совпадать фрагмент `.txt`, а не `.0`.

Если испытуемая строка оканчивается обратным слэшем или именем файла, не содержащим ни одной точки, регулярное выражение вообще не будет обнаруживать совпадение. Когда оно действительно будет обнаруживать соответствие, совпадение будет содержать расширение, включая точку, которая отделяет его от имени файла.

См. также

Рецепт 7.19, если заранее не известно, хранит ли испытуемая строка допустимый путь Windows.

7.25. Удаление недопустимых символов из имен файлов

Задача

Необходимо удалить последовательности символов, которые недопустимы в именах файлов Windows. Например, имеется строка с названием документа, которую требуется использовать как имя файла по умолчанию, когда пользователь в первый раз щелкнет на кнопке Save (Сохранить).

Решение

Регулярное выражение

```
[\\/:"*?<>|]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

Оставить пустой строку с замещающим текстом.

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl, Python, Ruby

Обсуждение

В именах файлов в системе Windows не допускается использовать символы \/:"*?‹›|. Эти символы используются для отделения имен устройств и папок, для заключения путей в кавычки, в качестве шаблонных символов и для перенаправления ввода-вывода в командной строке.

Совпадение с этими символами легко можно обеспечить с помощью символьного класса ‹[\\/:"*?‹›|]›. Обратный слэш в символьных классах является метасимволом, его необходимо экранировать другим символом обратного слэша. Все остальные символы всегда интерпретируются как литералы в символьных классах.

Для повышения эффективности мы повторяем символьный класс с помощью квантификатора ‹+›. Поэтому если строка содержит непрерывную последовательность недопустимых символов, то вся эта последовательность будет удалена за один проход. Разница в производительности едва ли будет заметна при работе с короткими строками, такими как имена файлов, но это отличный прием, который следует иметь в виду при работе с большими объемами данных, где наверняка будут встречаться длинные последовательности символов, которые следует удалить.

Так как требуется всего лишь удалить нежелательные символы, в качестве замещающего текста в операции поиска с заменой следует использовать пустую строку.

См. также

Рецепт 3.14, где описывается, как реализовать операцию поиска с заменой с использованием фиксированного замещающего текста в разных языках программирования.

8

Разметка и обмен данными

Основное внимание в заключительной главе будет уделено решению типичных задач, которые возникают при работе с различными языками разметки и форматами: HTML, XHTML, XML, CSV и INI. Несмотря на то, что глава предполагает хотя бы базовое знакомство с этими технологиями, тем не менее, в начале главы приводятся краткие описания каждой из них, чтобы можно было освежить свои знания, прежде чем погружаться в эти технологии. Описания концентрируются на основных синтаксических правилах, необходимых для обеспечения корректного поиска структур данных в каждом из форматов. Остальные подробности будут вводиться по мере необходимости, когда мы будем сталкиваться с соответствующими проблемами.

Это не всегда очевидно, но некоторые из этих форматов могут оказаться неожиданно сложными в обработке, по крайней мере, с помощью регулярных выражений. Как правило, для решения большинства задач, рассматриваемых в этой главе, вместо регулярных выражений лучше использовать специализированные парсеры и функции, особенно, если точность имеет важное значение (например, если результаты обработки могут повлиять на безопасность). Однако в этих рецептах приводятся интересные приемы, которые могут использоваться во многих задачах быстрой обработки данных.

Итак, посмотрим, с чем нам придется иметь дело. Большая часть сложностей, рассматриваемых в этой главе, связана с необходимостью обрабатывать ситуации, возникающие при разнообразных отклонениях от приведенных далее правил.

Язык разметки гипертекста (Hypertext Markup Language, HTML)

Язык HTML используется для описания структуры, семантики и представления миллиардов веб-страниц и других документов. Поэтому совершенно естественно попытаться привлечь регулярные выражения к обработке HTML-документов, но следует знать заранее, что этот язык плохо подходит для обработки жесткими и точными

регулярными выражениями. Это особенно верно в отношении увечной разметки HTML, которая часто встречается в веб-страницах, частично благодаря чрезвычайной терпимости веб-броузеров к определенным ошибкам в конструировании разметки HTML.

В этой главе мы сосредоточимся на правилах обработки ключевых компонентов корректно оформленного документа HTML: элементов (и атрибутов, содержащихся в них), ссылок на символы, комментариях и на объявлениях типов документов. В этой книге рассматривается версия HTML 4.01, выпущенная в 1999 году, которая на момент написания этих строк оставалась последней законченной версией стандарта.

Основные строительные блоки разметки HTML называются *элементами*. Элементы записываются с помощью *тегов*, которые окружаются угловыми скобками. Элементы подразделяются на блочные (такие как параграфы, заголовки, списки, таблицы и формы) и встраиваемые (такие как гиперссылки, цитаты, выделение курсивом и элементы ввода). Обычно элементы имеют как открывающий тег (например, `<html>`), так и закрывающий (например, `</html>`). Открывающий тег элемента может содержать *атрибуты*, которые будут описаны позже. Между тегами находится *содержимое* элемента, которое может состоять из текста и других элементов или оставаться пустым. Элементы могут вкладываться друг в друга, но они не могут перекрываться (например, `<div><div></div></div>` – это допустимое взаиморасположение элементов, а `<div><span></div></span>` – нет). Для некоторых элементов (таких как `<p>`, который обозначает параграф) закрывающий тег является необязательным. Элементы с необязательным закрывающим тегом закрываются автоматически, открывающим тегом нового блочного элемента. Имеется несколько элементов (включая `<br>`, завершающий строку), которые не имеют содержимого и никогда не используют закрывающие теги. Однако даже пустой элемент может иметь атрибуты. Имена элементов HTML начинаются с символов A–Z. Во всех допустимых именах элементов используются только буквы и цифры. Имена элементов нечувствительны к регистру символов.

Элементы `<script>` и `<style>` заслуживают особого внимания, потому что они позволяют встраивать в документы программный код на языке сценариев и таблицы стилей. Эти элементы закрываются ближайшим вхождением закрывающих тегов `</style>` или `</script>`, даже если они присутствуют внутри комментария или строки, в пределах описания стиля или программного кода.

Атрибуты определяются внутри открывающего тега элемента, сразу же вслед за его именем, и отделяются одним или несколькими пробельными символами. Большинство атрибутов записываются в виде пар имя-значение. Так, в следующем примере показан элемент `<a>` (anchor – якорь) с двумя атрибутами и содержимым «Click me!»:

```
<a href="http://www.regexcookbook.com"
   title = 'Regex Cookbook'>Click me!</a>
```

Как видно из этого примера, имя атрибута и значение отделяются друг от друга знаком равенства и необязательным пробельным символом. Значение заключается в апострофы или в кавычки. Чтобы использовать в значениях атрибутов кавычки того же типа, что и объемлющие, необходимо использовать ссылки на символы (описываются ниже). Значения атрибутов можно не заключать в кавычки или апострофы, если они содержат только символы A-Z, a-z, 0-9, символ подчеркивания, точку, двоеточие и дефис (в виде регулярного выражения это требование выглядит, как `<^[-.0-9:A-Z_a-z]+$>`). Некоторые атрибуты (такие как `selected` и `checked`, используемые некоторыми элементами форм) оказывают воздействие на элемент, содержащий их, уже своим присутствием и не требуют указывать значение. В таких случаях знак равенства, отделяющий имя атрибута от значения, также опускается. Как вариант, в качестве значений таких «минимизированных» атрибутов допускается использовать их имена (например, `selected="selected"`). Имена атрибутов начинаются с символов A–Z. Во всех допустимых именах атрибутов используются только буквы и дефисы. Атрибуты могут следовать в любом порядке, а их имена нечувствительны к регистру символов.

В языке HTML версии 4 определяются 252 *мнемонические ссылки на символы* и более миллиона *числовых ссылок на символы* (все вместе мы будем называть их *ссылками на символы*). Числовые ссылки на символы определяют соответствующие им символы с помощью кодовых пунктов Юникода и записываются в формате `&#`*nnnn*`;` или `&#x`*hhhh*`;`, где *nnnn* – это одна или более десятичных цифр в диапазоне 0-9, а *hhhh* – это одна или более шестнадцатеричных цифр в диапазоне 0-9 и A-F (без учета регистра символов). Мнемонические ссылки на символы записываются в формате `&`*имямнемоники*`;` (с учетом регистра символов, в отличие от большинства других аспектов языка HTML) и особенно удобны, когда требуется ввести литерал символа, имеющего специальное назначение в некоторых контекстах, например угловые скобки (`<` и `>`), кавычки (`"`) и амперсанд (`&`).

Также часто используется мнемоника ` ` (неразрывный пробел, код 0xA0), удобство которой обусловлено тем, что в представлении документа HTML отображаются все вхождения этого символа, даже если они следуют друг за другом. Пробелы, символы табуляции, разрывы строк обычно отображаются как единственный пробельный символ, даже если подряд следует множество таких символов. Амперсанд (`&`) не может использоваться иначе как в ссылках на символы.

Комментарии в разметке HTML оформляются следующим образом:

```
<!-- это комментарий -->
<!-- это тоже комментарий, но он располагается
     на нескольких строках -->
```

Содержимое комментария не имеет специального значения и не отображается средствами просмотра. Между закрывающей последовательностью дефисов -- и символом > допускается вставлять пробельные символы. Для совместимости с ранними (до 1995 года) броузерами некоторые разработчики окружают комментариями HTML содержимое элементов <script> и <style>. Современные броузеры игнорируют такие комментарии и обрабатывают сценарии или таблицы стилей обычным образом.

Наконец, документы HTML часто начинаются с объявления типа документа (неофициально «DOCTYPE»), который содержит спецификацию допустимого и недопустимого содержимого документа. Объявление DOCTYPE по своему виду напоминает элемент HTML, как видно в следующей строке, используемой для документов, строго следующих определению HTML 4.01 Strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

Такова, в двух словах, физическая структура документа HTML. Помните, что в действительности разметка HTML часто изобилует отклонениями от этих правил, которые успешно распознаются большинством броузеров. Кроме того, следует отметить, что каждый элемент накладывает свои ограничения на содержимое и свои внутренние атрибуты, при использовании которых документ HTML рассматривается как допустимый. Рассмотрение подобных правил, предъявляемых к содержимому, выходит далеко за рамки этой книги, но издательство O'Reilly выпустило книгу Чака Муссиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedy) «HTML & XHTML: The Definitive Guide»[1], которая является отличным источником этой информации.

Так как по своей структуре разметка HTML очень похожа на разметку XHTML и XML (описываются ниже), многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки.

Расширяемый язык разметки гипертекста (Extensible Hypertext Markup Language, XHTML)

Язык XHTML разрабатывался как развитие HTML 4.01 с целью ухода от связи HTML с SGML и перехода на основы XML. Однако в настоящее время продолжается независимое развитие HTML, поэтому более правильным будет считать язык XHTML альтернативой языку HTML. В этой книге охватываются версии XHTML 1.0 и 1.1.

[1] Чак Муссиано, Билл Кеннеди «HTML и XHTML. Подробное руководство», 6-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

Несмотря на то, что эти версии стандарта во многом сохраняют обратную совместимость с HTML, тем не менее, существует несколько важных отличий от структуры HTML, описанной выше:

- Документы XHTML могут начинаться с *объявления XML*, такого как `<?xml version="1.0" encoding="UTF-8"?>`.
- Непустые элементы должны завершаться закрывающими тегами. Пустые элементы должны иметь закрывающий тег или оканчиваться последовательностью />.
- Имена элементов и атрибутов чувствительны к регистру и должны записываться символами нижнего регистра.
- Благодаря использованию префиксов пространства имен XML имена элементов и атрибутов могут помимо символов, допустимых в именах элементов и атрибутов HTML, содержать двоеточие.
- Не допускается использование значений атрибутов, не заключенных в кавычки. Значения атрибутов должны окружаться кавычками или апострофами.
- Все атрибуты должны иметь соответствующие им значения.

Между HTML и XHTML существует еще ряд различий, которые касаются в основном крайних случаев и обработки ошибок, но в большинстве своем не оказывают влияния на регулярные выражения в этой главе. Дополнительную информацию о различиях между HTML и XHTML можно найти по адресу *http://www.w3.org/TR/xhtml1/#diffs* и *http://wiki.whatwg.org/wiki/HTML_vs._XHTML*.

Синтаксис языка разметки XHTML очень похож на синтаксис HTML и следует правилам языка XML, поэтому многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки. рецепты, в которых упоминается «(X)HTML», в равной степени относятся к HTML и XHTML. Как правило, бывает желательно избежать зависимости от использования соглашений только языка HTML или только XHTML, поскольку они часто смешиваются между собой в документах и веб-броузеры обычно воспринимают такую смесь.

Расширяемый язык разметки (Extensible Markup Language, XML)

Язык XML – это многоцелевой язык, предназначенный, в первую очередь, для обмена структурированными данными. Он используется как основа для создания широкого диапазона языков разметки, включая XHTML, который только что рассматривался. В этой книге охватываются версии XML 1.0 и 1.1. Полное описание особенностей языка XML и его грамматики выходит далеко за рамки этой книги, но применительно к нашему случаю существует всего несколько важных отличий от структуры языка HTML, описанного выше:

- Документы XML могут начинаться с объявления XML, такого как `<?xml version="1.0" encoding="UTF-8"?>`, и содержать другие инструкции по обработке, имеющие похожий формат. Например, инструкция `<?xml-stylesheet type="text/xsl" href="transform.xslt"?>` указывает, что к документу должен применяться файл *transform.xslt* преобразований XSL.
- Объявление DOCTYPE может включать внутренние объявления, заключенные в квадратные скобки. Например:

```
<!DOCTYPE example [
  <!ENTITY copy "&#169;">
  <!ENTITY copyright-notice "Copyright &copy; 2008, O'Reilly Media">
]>
```

- *Разделы CDATA* используются для экранирования блоков текста. Они начинаются строкой `<![CDATA[` и завершаются первым вхождением последовательности символов `]]>`.
- Непустые элементы должны иметь закрывающий тег. Пустые элементы должны либо иметь закрывающий тег, либо оканчиваться символами `/>`.
- *Имена* в языке XML (которые подчиняются правилам, применяемым к именам элементов, атрибутов и мнемонических ссылок) чувствительны к регистру символов и могут использовать символы Юникода. В число допустимых символов входят A–Z, a–z, двоеточие (:) и символ подчеркивания (_), а после первого символа также допускается использовать 0–9, дефис (-) и точку (.). Подробнее об этом рассказывается в рецепте 8.4.
- Употребление значений атрибутов без кавычек не допускается. Значения атрибутов должны заключаться в апострофы или в кавычки.
- Все атрибуты должны сопровождаться значениями.

Существует множество других правил, которых следует придерживаться, чтобы обеспечить создание правильно оформленных документов XML или, если возникнет необходимость, создать свой собственный парсер XML. Однако только что описанных правил (в дополнение к структуре документов HTML, которая была описана выше) обычно бывает вполне достаточно для реализации простых регулярных выражений.

Так как по своей структуре разметка XML очень похожа на разметку HTML и составляет основу разметки XHTML, многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки. рецепты, в которых упоминается «XML», в равной степени относятся к языкам XML, XHTML и HTML.

Значения, разделенные запятыми (Comma-Separated Values, CSV)

CSV – это старый, но все еще широко применяемый формат файлов, используемый для представления табличных данных. Формат CSV поддерживается большинством приложений электронных таблиц и систем управления базами данных и пользуется большой популярностью при организации обмена данными между приложениями. Несмотря на отсутствие какой-либо официальной спецификации формата CSV, в октябре 2005 года была предпринята попытка дать ему общее определение в документе RFC 4180, а кроме того, этот формат был зарегистрирован организацией IANA, как тип MIME «text/csv». До публикации указанного документа RFC стандартом де-факто считались соглашения, используемые в Microsoft Excel. Так как RFC определяет правила, очень похожие на те, что используются программой Excel, такое положение вещей не представляет большой проблемы.

Эта глава охватывает формат CSV, определяемый документом RFC 4180 и используемый приложением Microsoft Excel 2003 и более поздних версий.

Как следует из названия, файлы в этом формате содержат список значений, или *полей*, разделенных запятыми. Каждая строка, или *запись*, располагается в одной текстовой строке. За последним полем в записи символ запятой не указывается. За последней записью в файле может следовать разрыв строки. Все записи в файле должны содержать одинаковое число полей.

Значение каждого поля в формате CSV может заключаться в кавычки. Кроме того, поля вообще могут быть пустыми. Любое поле, содержащее запятые, кавычки или разрывы строк, должно заключаться в кавычки. Кавычки внутри поля должны экранироваться другой, предшествующей ей, кавычкой.

Первая запись в файле CSV иногда используется как заголовок, содержащий имена всех столбцов. Это невозможно определить программным способом, исходя из содержимого файла CSV, поэтому некоторые приложения предлагают пользователю указать, как следует интерпретировать первую строку.

Документ RFC 4180 определяет, что начальные и конечные пробелы в значениях поля являются частью значения. В некоторых старых версиях Excel эти пробелы игнорируются, но Excel 2003 и более поздние версии следуют этому указанию RFC. Документ RFC не определяет порядок обработки ошибок, связанных с наличием неэкранированных кавычек или с чем-то другим. В редких случаях обработка этой ошибки, выполняемая в Excel, может приводить к непредсказуемым результатам, поэтому очень важно гарантировать экранирование кавычек. Поля, содержащие кавычки, сами должны заключаться в кавычки, а поля в кавычках не должны содержать начальных или конечных пробелов за пределами кавычек.

Ниже приводится пример фрагмента в формате CSV, демонстрирующий многие из правил, которые мы только что рассмотрели. Он содержит две записи, с тремя полями в каждой:

```
aaa,b b,"""c"" cc"
1,,"333, three,
still more threes"
```

В табл. 8.1 только что продемонстрированный фрагмент CSV отображается более наглядно.

Таблица 8.1. Пример отображения фрагмента в формате CSV

aaa	b b	"c" cc
1	(пустое поле)	333, three, still more threes

Несмотря на то, что мы описали правила CSV, которые учитываются в рецептах этой главы, существует огромное количество вариаций, как различные программы читают и записывают файлы CSV. Многие приложения даже позволяют в файлах с расширением «csv» использовать в качестве разделителей полей не только запятые, но и другие символы. В число других типичных отклонений входит порядок включения запятых (или других разделителей полей), кавычек и разрывов строк в значения полей и порядок интерпретации начальных и завершающих пробелов в полях без кавычек – они могут игнорироваться или интерпретироваться как литералы.

Файлы инициализаци (INI)

Простой формат INI часто используется для оформления файлов с настройками. Этот формат плохо определен, и, как результат, существует масса вариантов интерпретации этого формата различными программами и системами. Регулярные выражения в этой главе придерживаются наиболее распространенных соглашений по оформлению файлов INI, которые описываются ниже.

Параметры в файле INI – это пары имя-значение, разделяемые знаком равенства, с возможными дополнительными пробелами и символами табуляции. Значения могут заключаться в апострофы или в кавычки, что позволяет включать в значения начальные и конечные пробелы и другие специальные символы.

Параметры могут группироваться в *разделы*, которые начинаются с имени раздела, заключенного в квадратные скобки и находящегося в отдельной строке. Раздел простирается, пока не будет встречено объявление следующего раздела или конец файла. Разделы не могут быть вложенными.

Точка с запятой отмечает начало *комментария*, который простирается до конца строки. Комментарии могут присутствовать в строках

с параметрами или объявлениями разделов. Содержимое комментариев не имеет никакого значения.

Ниже приводится пример файла INI с вводным комментарием (отмечающий дату последнего изменения файла), двумя разделами («user» и «post») и тремя параметрами («name», «title» и «content»):

```
; last modified 2008-12-25
[user]
name=J. Random Hacker

[post]
title = Regular Expressions Rock!
content = "Let me count the ways..."
```

8.1. Поиск тегов XML

Задача

Требуется обеспечить совпадение с любыми тегами HTML, XHTML или XML, присутствующими в тексте, с целью удалить, изменить, подсчитать их количество или выполнить какие-либо другие операции.

Решение

Самое лучшее решение зависит от нескольких факторов, включая точность, эффективность и терпимость к приемлемым для вас ошибкам в разметке. Определившись с требованиями, предъявляемыми к конкретной ситуации, можно перейти к определению перечня того, что хотелось бы сделать с результатами. Однако неважно, предполагается ли удалять теги, выполнять поиск внутри них, добавлять или удалять атрибуты или заменять теги альтернативной разметкой, – в любом случае сначала нужно отыскать их.

Имейте в виду, что это будет длинный рецепт, полный тонкостей, исключений и вариаций. Если вы ищете быстрое решение и не желаете затрачивать усилия на поиск решения, наилучшим образом отвечающего вашим потребностям, возможно, вам стоит сразу перейти к разделу «Теги (X)HTML (не строго)» этого рецепта, где предлагается решение, сочетающее в себе приличную устойчивость к ошибкам в разметке и строгость подхода.

На скорую руку

Это решение является самым простым и используется гораздо чаще, чем можно было бы предположить, но оно включено сюда больше для сравнения и объяснения его недостатков. Оно может быть достаточно хорошим, если заранее известно, с какого рода содержимым придется работать, и последствия неправильной обработки не причинят существенных хлопот. Это регулярное выражение начинается сопоставле-

нием с символом <, а затем просто продолжает сопоставление до первого встреченного символа >:

```
<[^>]*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Допускается наличие символа > в значениях атрибутов

Следующее регулярное выражение тоже достаточно простое; оно дает корректные результаты не во всех случаях. Однако оно прекрасно справляется с возложенной на него задачей, если используется только для обработки фрагментов допустимой разметки (X)HTML. Его преимущество перед предыдущим регулярным выражением состоит в том, что оно корректно проходит символы > в значениях атрибутов:

```
<(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится это же регулярное выражение, дополненное комментариями и отступами для большей удобочитаемости:

```
<
(?: [^>"']   # Символ не в кавычках или...
  | “[^"]*”  # Значение атрибута в кавычках или...
  | ‘[^']*’  # Значение атрибута в апострофах
)*
>
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Оба эти регулярных выражения работают идентично, поэтому у себя можно использовать любое из них. Те, кто пишут на JavaScript, могут использовать только первый вариант, потому что в JavaScript отсутствует поддержка режима свободного форматирования.

Теги (X)HTML (не строго)

Помимо поддержки символов > в значениях атрибутов следующее регулярное выражение имитирует поведение броузеров, которые обычно реализуют не очень строгие требования к именам тегов (X)HTML. Это позволяет регулярному выражению обходить содержимое, которое не выглядит, как теги, включая комментарии, объявления DOCTYPE и литералы символов < в тексте. В нем используется тот же прием обработки атрибутов и других случайных символов, появляющихся внутри тега, который применялся в предыдущем выражении, но в него добавлена специальная обработка имени тега. В частности, это выражение требует, чтобы имя начиналось с буквы английского алфавита. Имя тега со-

храняется в обратной ссылке 1 на тот случай, если его потребуется извлечь:

```
</?([A-Za-z][^\s>/]*)(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И в режиме свободного форматирования:

```
<
/?                  # Разрешить закрывающие теги
([A-Za-z][^\s>/]*) # Сохранить имя тега в обратной ссылке 1
(?: [^>"']          # Символ не в кавычках или...
  | "[^"]*"         # Значение атрибута в кавычках или...
  | '[^']*'         # Значение атрибута в апострофах
)*
>
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Оба эти регулярных выражения работают идентично, хотя второй вариант не может использоваться в диалекте JavaScript, так как в нем отсутствует поддержка режима свободного форматирования.

Теги (X)HTML (строго)

Это регулярное выражение более сложное, чем те, что приводились выше, потому что оно фактически следует правилам оформления тегов (X)HTML, описанным во вводном разделе этой главы. Это не всегда бывает желательно, так как броузеры не строго следуют этим правилам. Другими словами, данное регулярное выражение не будет совпадать с содержимым, которое не выглядит как допустимый тег (X) HTML, что в результате может приводить к потере части содержимого, которое броузерами интерпретируется как тег (например, если в разметке используется имя атрибута, включающее недопустимые символы, или если атрибуты включены в закрывающий тег). Здесь одновременно соблюдаются правила оформления тегов HTML и XHTML, так как для них характерно смешивание соглашений. Имя тега сохраняется в обратной ссылке 1 или 2 (в зависимости от типа тега – открывающий или закрывающий), на тот случай, если его потребуется извлечь:

```
<(?:([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*(?:"[^"]*"|↵
'[^']*'|[-.:\w]+))?)*\s*/?|/([A-Z][-:A-Z0-9]*)\s*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы сделать выражение менее загадочным, ниже приводится это же выражение в режиме свободного форматирования с комментариями:

```
<                    #
(?:                  # Вариант выбора для открывающих тегов...
  ([A-Z][-:A-Z0-9]*) #   Сохранить имя открывающего тега в обратной ссылке 1
  (?:                #   Позволить присутствие нуля или более атрибутов...
    \s+              #   ...разделенных пробельными символами
    [A-Z][-:A-Z0-9]* #     Имя атрибута
    (?:              #
      \s*=\s*        #     Разделитель имени-значения в атрибуте
      (?: “[^”]*”    #       Значение атрибута в кавычках
        | ‘[^’]*’    #       Значение атрибута в апострофах
        | [-.:\w]+   #       Значение атрибута без кавычек (HTML)
      )              #
    )?               #     Позволить атрибуты без значений (HTML)
  )*                 #
  \s*                #   Позволить завершающие пробельные символы
  /?                 #   Позволить самозакрывающиеся теги (XHTML)
 |                   # Вариант выбора для закрывающих тегов...
  /                  #
  ([A-Z][-:A-Z0-9]*) #   Сохранить имя закрывающего тега в обратной ссылке 2
  \s*                #   Позволить завершающие пробельные символы
) #
> #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Теги XML (строго)

Язык XML имеет очень точные спецификации, и это означает, что пользовательские программы должны строго следовать его правилам. Это существенное отличие от HTML и многострадальных броузеров, обычно используемых для его обработки:

```
<(?:([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*(?:"[^"]*"|'[^']*'))*\s*↵
/?|/([_:A-Z][-.:\w]*)\s*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И снова это же регулярное выражение в режиме свободного форматирования с дополнительными комментариями:

```
<                    #
(?:                  # Вариант выбора для открывающих тегов...
  ([_:A-Z][-.:\w]*)  #   Сохранить имя открывающего тега в обратной ссылке 1
  (?:                #   Позволить присутствие нуля или более атрибутов...
    \s+              #   ...разделенных пробельными символами
    [_:A-Z][-.:\w]*  #     Имя атрибута
    \s*=\s*          #     Разделитель имени-значения в атрибуте
    (?: “[^”]*”      #       Значение атрибута в кавычках
```

```
      | '[^']*'      #       Значение атрибута в апострофах
    )                #
  )*                 #
  \s*                #   Позволить завершающие пробельные символы
  /?                 #   Позволить самозакрывающиеся теги
 |                   # Вариант выбора для закрывающих тегов...
  /                  #
  ([_:A-Z][-.:\w]*) #   Сохранить имя закрывающего тега в обратной ссылке 2
  \s*                #   Позволить завершающие пробельные символы
)                    #
>                    #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Подобно двум предыдущим решениям, предназначенным для поиска тегов (X)HTML, эти регулярные выражения сохраняют имя тега в обратной ссылке 1 или 2, в зависимости от типа совпавшего тега – открывающий или закрывающий. Регулярное выражение, совпадающее с тегами XML, немного короче, чем версия для (X)HTML, потому что здесь не приходится иметь дело с особенностями, характерными только для разметки HTML (минимизированные атрибуты и значения без кавычек). Кроме того, оно допускает использовать в именах элементов и атрибутов более широкий диапазон символов.

Обсуждение

Некоторые предостережения

Несмотря на то, что достаточно часто возникает потребность производить поиск XML-подобных тегов с помощью регулярных выражений, тем не менее, для обеспечения надежности требуется сбалансированность и внимательное отношение к данным, с которыми предстоит работать. Из-за этих сложностей некоторые стараются воздерживаться от использования регулярных выражений при работе с различными разновидностями разметки XML или (X)HTML, предпочитая специализированные парсеры и функции. Такую возможность обязательно нужно учитывать, потому что подобные инструменты обычно оптимизированы для быстрого выполнения возложенных на них задач, а кроме того, они включают устойчивые механизмы определения или обработки некорректной разметки. В мире броузеров, например, для манипулирования разметкой HTML, как правило, лучше воспользоваться преимуществами древовидной объектной модели документа (Document Object Model, DOM). В других случаях можно было бы воспользоваться парсером SAX или XPath. Однако время от времени возникают ситуации, когда решения на базе регулярных выражений работают просто прекрасно и их использование имеет определенный смысл.

Учтя эту «реплику в сторону», приступим к анализу регулярных выражений, которые приведены в этом рецепте. Первые два решения для большинства случаев оказываются чересчур упрощенными, но XML-подобные языки разметки они обрабатывают одинаково. Последние три следуют более строгим правилам и ориентированы на определенные языки разметки. Однако даже в последних решениях учитываются соглашения относительно тегов HTML и XHTML, потому что они часто смешиваются в одном документе, нередко по неосторожности. Например, автор может использовать в документе HTML теги в стиле XHTML, самозакрывающийся тег `<br />` или по ошибке использовать символы верхнего регистра в именах элементов в документе XHTML.

На скорую руку

Преимущество этого решения заключается в его простоте, благодаря которой выражение легко запоминается и вводится, а кроме того, быстро работает. Цена этой простоте – некорректная обработка некоторых допустимых и недопустимых конструкций XML и (X)HTML. Если вы работаете с разметкой, которую пишете сами и знаете, что появление таких конструкций в вашем испытуемом тексте невозможно, или если вас не беспокоят последствия ошибок, такая цена может быть вполне приемлемой. Другой пример, когда это решение может оказаться достаточно приемлемым, – когда вы работаете с текстовым редактором, позволяющим предварительно просмотреть совпадения с регулярным выражением.

Регулярное выражение начинается с литерала `‹<›` (совпадающего с началом тега). Далее следует инвертированный символьный класс и максимальный квантификатор звездочка `‹[^>]*›`, которые обеспечивают совпадение с нулем или более следующих далее символов, отличных от >. Им соответствуют имя тега, атрибуты и ведущий или замыкающий символ /. Здесь можно было бы использовать минимальный квантификатор (`‹[^>]*?›`), но это не даст ничего, кроме замедления работы регулярного выражения, потому что в этом случае возникает большее число возвратов (причина описывается в рецепте 2.13). Конец тега в регулярном выражении сопоставляется с литералом `‹››`.

Если вы предпочитаете использовать точку вместо инвертированного символьного класса `‹[^>]›`, это вполне возможно. Точка будет прекрасно выполнять свои функции при условии, что вместе с ней будет использоваться минимальная звездочка (`‹.*?›`) и активирован режим «точке соответствуют границы строк» (в диалекте JavaScript вместо точки можно использовать конструкцию `‹[\s\S]*?›`). Точка с максимальной звездочкой (шаблон `‹<.*››`) изменяет смысл регулярного выражения, вынуждая его некорректно совпадать со всем подряд от первого символа < до самого последнего символа > в испытуемом тексте, даже если для этого регулярному выражению придется попутно поглотить многочисленные теги.

Настало время обратиться к примерам. Это регулярное выражение будет совпадать с каждой из следующих строк текста целиком:

```
<div>
</div>
<div class="box">
<div id="pandoras-box" class="box" />
<!-- comment -->
<!DOCTYPE html>
<< < w00t! >
<>
```

Следует заметить, что этот шаблон совпадает не только с тегами. Хуже того, он не сможет правильно найти совпадение с целыми тегами `<input>` в испытуемых строках `<input type="button" value=">>">` или `<input type="button" onclick="alert(2 > 1)">`. Вместо этого регулярное выражение обнаруживает совпадения только до первого символа >, который появляется в значениях атрибутов. Аналогичные проблемы наблюдаются с комментариями, разделами CDATA в разметке XML, объявлениями DOCTYPE, с программным кодом в элементах `<script>` и везде, где встречаются символы >.

Если приходится обрабатывать что-то более сложное, чем самая простая разметка, особенно, когда испытуемый текст происходит из разных или неизвестных источников, лучше воспользоваться одним из более надежных решений, которые приводятся далее в этом рецепте.

Допускается наличие символа > в значениях атрибутов

Следующее регулярное выражение, подобно только что описанному и созданному на скорую руку, включено, прежде всего, для сравнения с последующими, более надежными решениями. Однако оно соблюдает самые основные правила, необходимые для сопоставления с XML-подобными тегами, и поэтому может лучше соответствовать вашим потребностям при обработке фрагментов допустимой разметки, включающей только элементы и текст. Отличие от предыдущего выражения заключается в том, что оно пропускает символы >, встречающиеся в значениях атрибутов. Например, оно корректно будет совпадать с целыми тегами `<input>` в предыдущих примерах испытуемого текста: `<input type="button" value=">>">` и `<input type="button" onclick="alert(2 > 1)">`.

Первый вариант выбора – это инвертированный символьный класс `<[^>"']>`, совпадающий с любым одиночным символом, отличным от правой угловой скобки (которая закрывает тег), кавычки или апострофа (оба типа кавычек указывают на начало значения атрибута). Первый вариант выбора отвечает за совпадение с именами тегов и атрибутов, а также с другими символами, находящимися за пределами значений в кавычках. Такой порядок следования вариантов выбран намеренно, для обеспечения более высокой производительности. Меха-

низмы регулярных выражений опробуют варианты в регулярном выражении в направлении слева направо, поэтому первым следует вариант, вероятность совпадения с которым выше, чем вероятность совпадения с вариантами для значений в кавычках (тем более, что сопоставление производится по одному символу за раз).

Далее следуют варианты для сопоставления со значениями атрибутов в кавычках и апострофах (‹"[^"]*"› и ‹'[^']*'›). Использование инвертированных символьных классов в них позволяет им продолжать сопоставление, невзирая на наличие символов >, разрывов строк и всего остального, что не является закрывающей кавычкой.

Следует отметить, что в этом решении не предусматривается никакой специальной обработки комментариев, что позволяло бы исключать комментарии или обеспечивать корректное совпадение с ними и с другими специальными узлами в документе. Прежде чем применять это регулярное выражение, обязательно следует определить, содержимое какого вида требуется обработать.

Безопасная(!) оптимизация эффективности

После прочтения этого раздела может сложиться впечатление, что можно заставить регулярное выражение работать быстрее, добавив квантификатор ‹*› или ‹+› после первого инвертированного символьного класса (‹[^>"']›). В позициях в испытуемом тексте, где регулярное выражение будет находить совпадение, это действительно так. Возможность совпадения сразу с несколькими символами за раз позволит механизму регулярных выражений пропускать большое число ненужных шагов на пути к успешному совпадению.

Однако такое изменение может привести к менее очевидным отрицательным последствиям в тех позициях, где механизм регулярных выражений обнаруживает лишь частичное совпадение. Когда регулярное выражение обнаруживает совпадение с открывающим символом <, за которым не следует символ >, наличие которого обеспечило бы успешное завершение попытки сопоставления, возникает проблема «катастрофических возвратов», описанная в рецепте 2.15. Она обусловлена огромным числом способов объединения внутреннего квантификатора с внешним (после несохраняющей группы) для сопоставления с текстом, следующим за символом <, каждый из которых механизм регулярных выражений вынужден будет опробовать, прежде чем убедиться в безуспешности попытки сопоставления. Будьте внимательны!

В диалектах регулярных выражений, поддерживающих захватывающие квантификаторы или атомарную группировку (диалекты

JavaScript и Python не поддерживают ни то, ни другое), можно избежать этой проблемы, пользуясь при этом преимуществами высокой производительности сопоставления сразу с несколькими символами не в кавычках. Более того, есть возможность пойти еще дальше и уменьшить число потенциальных возвратов в другом месте регулярного выражения. Если используемый диалект регулярных выражений поддерживает обе возможности, лучше использовать захватывающие квантификаторы (показаны здесь во втором регулярном выражении), потому что они позволяют сохранить регулярное выражение более коротким и удобочитаемым.

С атомарной группировкой:

```
<(?>(?:(?>[^>"']+)|"[^"]*"|'[^']*')*)>
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

С захватывающими квантификаторами:

```
<(?:[^>"']++|"[^"]*"|'[^']*')*+>
```

Параметры: нет
Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Теги (X)HTML (не строго)

Благодаря простому дополнению это регулярное выражение намного ближе имитирует соблюдение свободных правил, которые используют веб-броузеры при идентификации тегов (X)HTML в исходных текстах. Это решение хорошо подходит для ситуаций, когда необходимо копировать поведение броузера и не приходится беспокоиться о том, действительно ли соответствуют совпадающие теги всем правилам, определяющим допустимую разметку. Однако следует иметь в виду, что может попасться ужасающе недопустимая разметка HTML, которую данное выражение не сможет обрабатывать способом, похожим на тот, которым это делают броузеры, потому что синтаксический анализ крайних случаев ошибочной разметки в броузерах производится их собственными, уникальными способами.

Самое существенное отличие этого регулярного выражения от предыдущего решения заключается в том, что оно требует, чтобы за левой угловой скобкой (<) следовала буква в диапазоне A–Z или a–z, которой может предшествовать символ / (для закрывающих тегов). Это ограничение исключает возможность совпадения со случайными символами < в тексте, а также с комментариями, с объявлениями DOCTYPE, с объявлениями и инструкциями обработки XML, с разделами CDATA и так

далее. Этот прием не защищает от совпадения с чем-то, что выглядит как тег *внутри* комментариев, в программном коде на языке сценариев, в содержимом элементов <textarea> и так далее. В разделе «Игнорирование необычных разделов (X)HTML и XML» на стр. 534 показано, как обойти эту проблему. Но сначала рассмотрим, как работает это регулярное выражение.

Символ ‹<› начинает совпадение с литерала левой угловой скобки. Конструкция ‹/?›, следующая далее, совпадает с необязательным символом слэша в закрывающих тегах. Далее следует сохраняющая группа ‹([A-Za-z][^\s>/]*)›, совпадающая с именем тега и запоминающая его как обратную ссылку 1. Если ссылаться на имя тега не потребуется (например, когда нужно просто удалить все теги), сохраняющие круглые скобки можно удалить (достаточно лишь сохранить шаблон, заключенный в них). Внутри группы имеются два символьных класса. Первый класс, ‹[A-Za-z]›, устанавливает правило первого символа в имени тега. Следующий класс, ‹[^\s>/]›, допускает совпадение почти с любыми символами, которые являются частью имени. Единственное исключение – пробельные символы (‹\s›, которые отделяют имя тега от следующих за ним атрибутов), > (который заканчивает тег) и / (используемый перед завершающим символом > в единичных тегах XHTML). Любые другие символы (даже включая кавычки) интерпретируются как часть имени тега. Это может показаться слишком свободным требованием, но именно так действует большинство броузеров. Ошибочные теги могут не оказывать никакого влияния на способ отображения страницы, но они, тем не менее, будут доступны в дереве DOM и не будут отображаться как текст, хотя любое содержимое, имеющееся внутри них, будет отображаться.

Вслед за именем тега следует подвыражение, обрабатывающее атрибуты, которое было взято непосредственно из предыдущего регулярного выражения: ‹(?:[^>"']|"[^"]*"|'[^']*')*›. Остается только добавить правую угловую скобку, завершающую тег, и дело сделано.

Следующие регулярные выражения демонстрируют, как можно адаптировать этот шаблон для совпадения только с открывающими, только с закрывающими или с одиночными (самозакрывающимися) тегами:

Открывающие теги

```
<([A-Za-z][^\s>/]*)(?:[^>"'/]|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В этой версии внутрь первого инвертированного символьного класса в несохраняющей группе добавлен символ слэша (/), чтобы предотвратить возможность совпадения со слэшами за пределами кавычек, окружающих значения атрибутов.

Одиночные теги

```
<([A-Za-z][^\s>/]*)(?:[^>"']|"[^"]*"|'[^']*')*/>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Здесь непосредственно перед завершающей правой угловой скобкой добавлен обязательный символ слэша.

Открывающие и одиночные теги

```
<([A-Za-z][^\s>/]*)(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Здесь нет никаких дополнений. Вместо этого была удалена конструкция ‹/?›, следовавшая за открывающей угловой скобкой ‹<› в оригинальном выражении.

Закрывающие теги

```
</([A-Za-z][^\s>/]*)(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В данном случае символ слэша за открывающей угловой скобкой сделан обязательной частью совпадения. Следует отметить, что мы умышленно допустили возможность появления атрибутов в закрывающих тегах, потому что это регулярное выражение основано на «не строгом» решении. Даже при том, что броузеры не реагируют на атрибуты, содержащиеся в закрывающих тегах, тем менее, они не возражают против их существования.

Во врезке «Безопасная(!) оптимизация эффективности» на стр. 528 показано, как повысить скорость сопоставления с тегами при помощи атомарной группировки и захватывающих квантификаторов. В данном случае производительность потенциально может быть улучшена еще больше, потому что наборы символов, которые могут совпасть с символьным классом ‹[^\s>/]› и с последней частью выражения, перекрываются, обеспечивая тем самым огромный набор возможных комбинаций, которые механизму регулярных выражений придется опробовать, прежде чем оставить неудачную попытку совпадения.

Если в используемом диалекте регулярных выражений доступна атомарная группировка или захватывающие квантификаторы, за счет их использования здесь можно существенно повысить производительность. Следующие изменения также могут быть перенесены в регулярные выражения, совпадающие с открывающими/закрывающими/одиночными тегами, которые приводятся чуть выше:

```
</?([A-Za-z](?>[^\s>/]*))(?>(?:(?>[^>"']+)|"[^"]*"|'[^']*')*)>
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
</?([A-Za-z][^\s>/]*+)(?:[^>"']++|"[^"]*"|'[^']*')*+>
```

Параметры: нет
Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Теги (X)HTML (строго)

Говоря о строгости этого решения, мы подразумеваем, что оно пытается следовать синтаксическим правилам HTML и XHTML, описанным во вводном разделе этой главы, вместо того чтобы имитировать работу броузеров при парсинге исходного кода разметки документа. По сравнению с предыдущими регулярными выражениями эта строгость добавляет следующие правила:

- Имена тегов и атрибутов должны начинаться с буквы A–Z или a–z, а внутри имен могут использоваться только символы A–Z, a–z, 0–9, дефис и двоеточие (в виде регулярного выражения это выглядит как: `‹^[-:A-Za-z0-9]+$›`).
- Ошибочные, случайные символы недопустимы в именах тегов. За именем тега могут следовать только пробельные символы, атрибуты (со значениями или без) и необязательный завершающий символ слэша (/).
- В значениях атрибутов, не заключенных в кавычки, могут использоваться только символы A–Z, a–z, 0–9, символ подчеркивания, дефис, точка и двоеточие (в виде регулярного выражения это выглядит как: `‹^[-.:A-Za-z0-9_]+$›`).
- Закрывающие теги не могут включать атрибуты.

Так как шаблон разбит на два варианта выбора (первый – для открывающих и одиночных тегов, а второй – для закрывающих), имя тега сохраняется в обратной ссылке 1 или 2, в зависимости от типа совпавшего тега. Оба набора сохраняющих круглых скобок можно убрать, если обратные ссылки на имена тегов не нужны.

В следующих примерах два варианта выбора шаблона выделены в отдельные регулярные выражения. Оба они сохраняют имя тега в обратной ссылке 1:

Открывающие и одиночные теги

```
<([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*↵
(?:"[^"]*"|'[^']*'|[-.:\w]+))?)*\s*/?>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Конструкция ‹/?›, стоящая непосредственно перед закрывающей угловой скобкой ‹>›, позволяет регулярному выражению совпадать как с открывающими, так и с одиночными тегами. Если ее удалить, выражение будет совпадать только с открывающими тегами. Если удалить только квантификатор знак вопроса (сделав обязательным совпадение с символом ‹/›), оно будет совпадать только с одиночными тегами.

Закрывающие теги

```
</([A-Z][-:A-Z0-9]*)\s*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В последних двух разделах говорилось, как увеличить потенциальную производительность за счет добавления атомарной группировки или захватывающих квантификаторов. Строго определенные пути поиска совпадений с этим регулярным выражением подразумевают отсутствие потенциальных совпадений с одной и той же строкой более чем одним способом, вследствие этого снижается вероятность возвратов, которые могут порождать проблемы. Это регулярное выражение не *рассчитывает* на возможность возвратов, поэтому если возникнет желание, можно каждый последний квантификатор ‹*›, ‹+› и ‹?› сделать захватывающим (или получить тот же эффект с помощью атомарной группировки). Хотя этот прием может оказаться только полезным, мы откажемся от вариантов его применения для данного и последующих выражений, чтобы постараться в этом рецепте сохранить контроль над сумасшедшим числом возможностей.

На следующей странице в разделе «Игнорирование необычных разделов (X)HTML и XML» демонстрируется, как избежать совпадения с тегами внутри комментариев, в тегах <script> и т. п.

Теги XML (строго)

Язык XML не допускает возможность «не строгого» решения, вследствие его точной спецификации и требований к парсерам не обрабатывать неправильно оформленную разметку. Хотя можно использовать предыдущие регулярные выражения при обработке документов XML, их упрощенный подход не обеспечивает более высокую надежность поиска, потому что не существует нестрогих программ для работы с разметкой XML, чье поведение можно было бы имитировать.

Это регулярное выражение в своей основе является упрощенной версией регулярного выражения из раздела «Теги (X)HTML (строго)», потому что появилась возможность убрать из него поддержку двух особенностей HTML, недопустимых в XML: значений атрибутов без кавычек и минимизированных атрибутов (атрибутов, не сопровождаемых значением). Еще одно отличие заключено в наборе символов, которые допускается использовать в именах тегов и атрибутов. Фактически правила определения имен в языке XML (применяемые к именам тегов и атри-

бутов) более свободны, чем показано здесь, и допускают использование сотен тысяч дополнительных символов Юникода. Если потребуется обеспечить допустимость этих символов при поиске, можно заменить три вхождения конструкции ‹[_:A-Z][-.:\w]*› на один из шаблонов из рецепта 8.4. Примечательно, что список допустимых символов может изменяться в зависимости от используемой версии XML.

Как и в случае с регулярными выражениями для (X)HTML, имя тега сохраняется в обратной ссылке 1 или 2, в зависимости от типа совпавшего тега, открывающий/одиночный или закрывающий. Здесь также можно удалить сохраняющие круглые скобки, если ссылки на имена тегов не требуются.

В следующих примерах два альтернативных шаблона выделены в отдельные регулярные выражения. В результате каждый из них сохраняет имя тега в обратной ссылке 1:

Открывающие и одиночные теги

```
<([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*↵
(?:"[^"]*"|'[^']*'))*\s*/?>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Конструкция ‹/?›, стоящая непосредственно перед закрывающей угловой скобкой ‹>›, позволяет регулярному выражению совпадать как с открывающими, так и с одиночными тегами. Если ее удалить, выражение будет совпадать исключительно с открывающими тегами. Если удалить только квантификатор знак вопроса, оно будет совпадать исключительно с одиночными тегами.

Закрывающие теги

```
</([_:A-Z][-.:\w]*)\s*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В следующем разделе демонстрируется, как избежать совпадения с тегами внутри комментариев, в разделах CDATA и в объявлениях DOCTYPE.

Игнорирование необычных разделов (X)HTML и XML

При попытке сопоставления с XML-подобными тегами внутри исходного файла или в строке наибольшую проблему представляет необходимость избежать совпадения с содержимым, которое выглядит как тег, когда его местоположение или контекст явно свидетельствуют, что это не тег. Регулярные выражения для работы с разметкой (X)HTML и XML, которые демонстрировались в этом рецепте, избегают совпадения с проблематичным содержимым за счет ограничений, накладываемых на первый символ в имени элемента. Некоторые идут еще дальше,

требуя, чтобы теги полностью соответствовали синтаксическим правилам (X)HTML или XML. Однако для полной надежности необходимо также избегать совпадения с содержимым, находящимся в комментариях, в программном коде на языке сценариев (где могут использоваться символы «больше» и «меньше» в математических операциях), в разделах CDATA языка XML и в ряде других конструкций. Эту проблему можно решить, если сначала отыскать эти проблематичные разделы, а затем выполнять поиск тегов только за пределами этих совпадений.

В рецепте 3.18 показано, как реализовать поиск между совпадениями с другим регулярным выражением. В этом рецепте используется два шаблона: внутреннее и внешнее регулярное выражение. Решения, которые были продемонстрированы выше, будут играть роль внутреннего выражения. Внешнее регулярное выражение показано ниже, с отдельными шаблонами для (X)HTML и XML. Этот прием скрывает проблематичные разделы от внутреннего регулярного выражения и тем самым позволяет сохранить реализацию достаточно простой.

Внешнее регулярное выражение для (X)HTML. Следующее регулярное выражение совпадает с комментариями, а также с элементами `<script>`, `<style>`, `<textarea>` и `<xmp>`[1] (включая их содержимое):

```
<!--.*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|↵
'[^']*')*?(?:/>|>.*?</\1\s*>)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Для тех, кому эта строка покажется неудобочитаемой, ниже приводится это регулярное выражение в режиме свободного форматирования с дополнительными комментариями:

```
# Комментарий
<!-- .*? --\s*>

|

# Специальные элементы и их содержимое
<( script | style | textarea | xmp )\b
    (?: [^>"']  # Символы не в кавычках
      | "[^"]*" # Значение в кавычках
      | '[^']*' # Значение в апострофах
    )*?
```

[1] `<xmp>` – это малоизвестный, но широко поддерживаемый элемент, похожий на `<pre>`. Подобно `<pre>` он сохраняет все пробельные символы и по умолчанию использует моноширинный шрифт, но кроме этого отображается все его содержимое (включая теги HTML), как простой текст. Элемент `<xmp>` был объявлен устаревшим в HTML 3.2 и был полностью удален в HTML 4.

```
(?: # Одиночный тег
    />
   | # Иначе включить содержимое элемента и отыскать закрывающий тег
    > .*? </\1\s*>
)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Ни одно из выражений, приведенных выше, не будет работать корректно в JavaScript, потому в JavaScript отсутствуют режимы «точке соответствуют границы строк» и «режим свободного форматирования». Следующее регулярное выражение снова возвращено к неудобочитаемому представлению, и в нем символ точки заменила конструкция ‹[\s\S]›, благодаря чему его можно использовать в диалекте JavaScript:

```
<!--[\s\S]*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|↵
'[^']*')*?(?:/>|>[\s\S]*?</\1\s*>)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Эти регулярные выражения представляют нечто вроде дилеммы: так как они совпадают с тегами <script>, <style>, <textarea> и <xmp>, эти теги никогда не будут обнаруживаться вторым (внутренним) регулярным выражением, хотя оно способно находить все теги. Однако решение этой проблемы – всего лишь вопрос добавления дополнительного программного кода, обрабатывающего эти теги отдельно. Эти (внешние) регулярные выражения уже сохраняют имена тегов в обратной ссылке 1, с помощью которой можно проверить, было ли найдено совпадение с комментарием или с тегом (и если это тег, то можно узнать его имя).

Внешнее регулярное выражение для XML. Это регулярное выражение совпадает с комментариями, с разделами CDATA и с объявлениями DOCTYPE. В каждом из этих случаев совпадение обеспечивается отдельными шаблонами, которые объединяются в общее регулярное выражение с помощью метасимвола выбора ‹|›:

```
<!--.*?--\s*>|<!\[CDATA\[.*?]]>|<!DOCTYPE\s(?:[^<>"']|"[^"]*"|↵
'[^']*'|<!(?:[^>"']|"[^"]*"|'[^']*')*>)*>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это же выражение в режиме свободного форматирования:

```
# Комментарий
<!-- .*? --\s*>
```

```
|

# Раздел CDATA
<!\[CDATA\[ .*? ]]>

|

# Объявление типа документа
<!DOCTYPE\s
    (?: [^<>"'] # Неспециальный символ
      | “[^”]*” # Значение в кавычках
      | ‘[^’]*’ # Значение в апострофах
      | <!(?:[^>"']|"[^"]*"|'[^']*')*> # Объявление разметки
    )*
>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

А это версия, работающая в диалекте JavaScript (где отсутствуют режимы «точке соответствуют границы строк» и «режим свободного форматирования»):

```
<!--[\s\S]*?--\s*>|<!\[CDATA\[[\s\S]*?]]>|<!DOCTYPE\s(?:[^<>"']|"[^"]*"|↵
‘[^’]*’|<!(?:[^>"']|"[^"]*"|'[^']*')*>)*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Варианты

Совпадение с допустимыми тегами HTML 4

Иногда бывает необходимо ограничить круг поиска только допустимыми элементами HTML, особенно при поиске тегов в документах не в формате HTML, где желательны дополнительные меры предосторожности против ложных совпадений.

Следующее регулярное выражение совпадает только с 91 допустимым элементом HTML 4. Этот список не включает нестандартные теги HTML, такие как <blink>, <bgsound>, <embed> и <nobr>. Он также не включает элементы, присутствующие только в версии XHTML 1.1 (в версию XHTML 1.0 не было добавлено новых тегов), или совершенно новые элементы, которые планируется включить в состав HTML 5:

```
</?(a|abbr|acronym|address|applet|area|b|base|basefont|bdo|big|blockquote|↵
body|br|button|caption|center|cite|code|col|colgroup|dd|del|dfn|dir|div|↵
dl|dt|em|fieldset|font|form|frame|frameset|h1|h2|h3|h4|h5|h6|head|hr|html|↵
i|iframe|img|input|ins|isindex|kbd|label|legend|li|link|map|menu|meta|↵
noframes|noscript|object|ol|optgroup|option|p|param|pre|q|s|samp|script|↵
```

```
select|small|span|strike|strong|style|sub|sup|table|tbody|td|textarea|↵
tfoot|th|thead|title|tr|tt|u|ul|var)\b(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Можно ускорить работу этого регулярного выражения, уменьшив число вариантов выбора, разделенных метасимволом ‹|›. Везде, где только возможно, мы использовали символьные классы и необязательные окончания. Эти изменения могут существенно уменьшить количество возвратов, необходимых механизму регулярных выражений. Рассмотрим, что произойдет, когда механизм встретит в испытуемом тексте последовательность символов <0. Эта последовательность не может быть началом тега, потому что ни в одном из тегов имя не начинается с цифры 0, но прежде чем механизм регулярных выражений сможет исключить это невозможное совпадение, он должен будет проверить, не начинается ли каждый из 91 вариантов выбора с символа 0. Уменьшив число вариантов до минимума (по одному для каждой буквы, с которой может начинаться имя тега), мы уменьшаем число шагов, которые должны быть выполнены после совпадения с левой угловой скобкой, с 91 до 19. Ниже показано, как выглядит такое регулярное выражение:

```
</?(a(?:bbr|cronym|ddress|pplet|rea)?|b(?:ase(?:font)?|do|ig|lockquote|↵
ody|r|utton)?|c(?:aption|enter|ite|o(?:de|l(?:group)?))|d(?:[dlt]|el|fn|↵
i[rv])|em|f(?:ieldset|o(?:nt|rm)|rame(?:set)?)|h(?:[1-6r]|ead|tml)|↵
i(?:frame|mg|n(?:put|s)|sindex)?|kbd|l(?:abel|egend|i(?:nk)?)|m(?:ap|↵
e(?:nu|ta))|no(?:frames|script)|o(?:bject|l|p(?:tgroup|tion))|p(?:aram|↵
re)?|q|s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|t(?:able|↵
body|[dhrt]|extarea|foot|head|itle)|ul?|var)\b(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В таком нагромождении сложно разобраться, но режим свободного форматирования может немного помочь:

```
<
/? # Разрешить совпадение с закрывающими тегами
(  # Сохранить имя тега в обратной ссылке 1

   a(?:bbr|cronym|ddress|pplet|rea)?|
   b(?:ase(?:font)?|do|ig|lockquote|ody|r|utton)?|
   c(?:aption|enter|ite|o(?:de|l(?:group)?))|
   d(?:[dlt]|el|fn|i[rv])|
   em|
   f(?:ieldset|o(?:nt|rm)|rame(?:set)?)|
   h(?:[1-6r]|ead|tml)|
   i(?:frame|mg|n(?:put|s)|sindex)?|
   kbd|
   l(?:abel|egend|i(?:nk)?)|
   m(?:ap|e(?:nu|ta))|
```

```
   no(?:frames|script)|
   o(?:bject|l|p(?:tgroup|tion))|
   p(?:aram|re)?|
   q|
   s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|
   t(?:able|body|[dhrt]|extarea|foot|head|itle)|
   ul?|
   var

) \b        # Не допускается частичное совпадение с именами
(?: [^>"']  # Любой символ, кроме >, “, или ‘
  | “[^”]*” # Значение атрибута в кавычках
  | ‘[^’]*’ # Значение атрибута в апострофах
)*
>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Для поклонников красоты в оформлении ниже приводится еще один способ записи того же регулярного выражения в режиме свободного форматирования, которое читается еще легче:

```
<
/? # Разрешить совпадение с закрывающими тегами
(  # Сохранить имя тега в обратной ссылке 1

   a (?: bbr                  #
       | cronym               #
       | ddress               #
       | pplet                #
       | rea                  #
     )?|                      # Необязательная группа (допускается тег <a>)
   b (?: ase (?:font)?        # <base>, <basefont>
       | do                   #
       | ig                   #
       | lockquote            #
       | ody                  #
       | r                    #
       | utton                #
     )?|                      # Необязательная группа (допускается тег <b>)
   c (?: aption               #
       | enter                #
       | ite                  #
       | o (?:de|l(?:group)?) # <code>, <col>, <colgroup>
     ) |                      #
   d (?: [dlt]                # <dd>, <dl>, <dt>
       | el                   #
       | fn                   #
       | i[rv]                # <dir>, <div>
     ) |                      #
   em |                       #
```

```
f (?: ieldset               #
    | o (?:nt|rm)           # <font>, <form>
    | rame (?:set)?         # <frame>, <frameset>
  ) |                       #
h (?: [1-6r]                # <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <hr>
    | ead                   #
    | tml                   #
  ) |                       #
i (?: frame                 #
    | mg                    #
    | n (?:put|s)           # <input>, <ins>
    | sindex                #
  )?|                       # Необязательная группа (допускается тег <i>)
kbd |                       #
l (?: abel                  #
    | egend                 #
    | i (?:nk)?             # <li>, <link>
  ) |                       #
m (?: ap                    #
    | e (?:nu|ta)           # <menu>, <meta>
  ) |                       #
no (?: frames               #
    | script                #
  ) |                       #
o (?: bject                 #
    | l                     #
    | p (?:tgroup|tion)     # <optgroup>, <option>
  ) |                       #
p (?: aram                  #
    | re                    #
  )?|                       # Необязательная группа (допускается тег <p>)
q |                         #
s (?: amp                   #
    | cript                 #
    | elect                 #
    | mall                  #
    | pan                   #
    | t (?:rike|rong|yle)   # <strike>, <strong>, <style>
    | u[bp]                 # <sub>, <sup>
  )?|                       # Необязательная группа (допускается тег <s>)
t (?: able                  #
    | body                  #
    | [dhrt]                # <td>, <th>, <tr>, <tt>
    | extarea               #
    | foot                  #
    | head                  #
    | itle                  #
  ) |                       #
ul? |                       # <u>, <ul>
var                         #
```

```
) \b       # Не допускается частичное совпадение с именами
(?: [^>"'] # Любой символ, кроме >, “ или ‘
| “[^”]*”  # Значение атрибута в кавычках
| ‘[^’]*’  # Значение атрибута в апострофах
)*
>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Если вы работаете с разметкой XHTML, следует учесть, что, в версии XHTML 1.0 не было добавлено новых тегов, но были удалены следующие 14 тегов: `<applet>`, `<basefont>`, `<center>`, `<dir>`, `<font>`, `<frame>`, `<frameset>`, `<iframe>`, `<isindex>`, `<menu>`, `<noframes>`, `<s>`, `<strike>` и `<u>`.

В версии XHTML 1.1 сохранились все элементы из версии XHTML 1.0 и были добавлены шесть новых (все они имеют отношение к отображению текста на азиатских языках): `<rb>`, `<rbc>`, `<rp>`, `<rt>`, `<rtc>` и `<ruby>`. Создание регулярного выражения, совпадающего только с допустимыми элементами XHTML версий 1.0 или 1.1, мы оставляем за вами.

См. также

Поиск совпадения с любым или со всеми тегами – распространенная задача, но нередко бывает необходимо обеспечить совпадение с каким-нибудь определенным тегом или с одним из небольшого набора; в рецепте 8.2 показано, как решить обе эти задачи.

Рецепт 8.4, где описываются символы, допустимые в именах элементов XML и их атрибутов.

8.2. Заменить тег <b> тегом

Задача

Необходимо заменить в испытуемом тексте все открывающие и закрывающие теги `<b>` соответствующими тегами `<strong>`, сохраняя любые существующие атрибуты.

Решение

Следующее регулярное выражение совпадает с открывающим и закрывающим тегами `<b>` как с атрибутами, так и без них:

```
<(/?)b\b((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
< #
(/?)           # Сохранить необязательный ведущий слэш в обратной ссылке 1
b \b           # Дополнить имя тега границей слова
(              # Сохранить любые атрибуты в обратной ссылке 2
   (?: [^>"']  #     Любые символы, кроме >, “ или ‘
     | “[^"]*” #     Значение атрибута в кавычках
     | ‘[^']*’ #     Значение атрибута в апострофах
   )*          #
)              #
>              #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Чтобы сохранить все атрибуты при изменении имени тега, можно использовать следующий замещающий текст:

```
<$1strong$2>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong\2>
```

Диалекты замещающего текста: Python, Ruby

Чтобы удалить все атрибуты, выполняя ту же процедуру, достаточно опустить обратную ссылку 2 в замещающем тексте:

```
<$1strong>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong>
```

Диалекты замещающего текста: Python, Ruby

Программный код, необходимый для реализации этой задачи, приводится в рецепте 3.15.

Обсуждение

Предыдущий рецепт включает подробное обсуждение множества способов сопоставления с XML-подобными тегами. Это освобождает данный рецепт от необходимости рассматривать прямолинейный подход к проблеме поиска конкретного тега. Тег `<b>` и его замена `<strong>` предложены только в качестве примера, но вы можете подставить на их место имена двух любых других тегов.

Регулярное выражение начинается сопоставлением с литералом `<` – с первым символом любого тега. Далее следует сопоставление с не-

обязательным символом слэша, который обнаруживается в закрывающих тегах, с помощью конструкции ‹/?›, заключенной в сохраняющие круглые скобки. Сохранение результата сопоставления с этим шаблоном (который будет либо пустой строкой, либо символом слэша) позволит легко восстановить слэш в замещающем тексте без использования какой-либо условной логики.

Затем производится сопоставление самого имени тега, <b>. При желании здесь можно использовать имя любого другого тега. Мы задействовали режим нечувствительности к регистру символов, чтобы также обеспечить совпадение с символом верхнего регистра B.

Про границу слова (‹\b›), следующую за именем тега, легко забыть, но это один из наиболее важных элементов регулярного выражения. Граница слова позволяет обеспечить совпадение только с тегами <b>, но не с <br>, <body>, <blockquote> или любыми другими тегами, которые просто начинаются с буквы «b». Для решения этой же проблемы после имени можно было бы использовать сопоставление с пробельным символом (‹\s›), но в этом случае выражение не будет совпадать с тегами, в которых отсутствуют атрибуты и которые поэтому не имеют пробельных символов после имени. Граница слова позволяет решить эту задачу просто и элегантно.

При работе с XML и XHTML не забывайте, что в именах тегов XML допускается использовать двоеточие для отделения пространства имен, а также дефис и некоторые другие символы, которые образуют границу слова. Например, регулярное выражение может обнаружить совпадение с чем-нибудь, похожим на <b-sharp>. Если в вашем случае это имеет значение, вместо границы слова можно воспользоваться опережающей проверкой ‹(?=[\s/>])›. Она позволит обеспечить невозможность совпадения с частью имени тега, причем это решение более надежное.

Шаблон ‹((?:[^>"']|"[^"]*"|'[^']*')*)›, следующий за именем тега, используется для совпадения со всем остальным внутри тега, вплоть до закрывающей угловой скобки. Если обернуть этот шаблон в сохраняющую группировку, как было сделано в этом примере, можно будет легко вставлять обратно любые атрибуты и другие символы (такие как завершающий символ слэша в одиночных тегах) в замещающем тексте. Внутри сохраняющих круглых скобок шаблон повторяет сопоставление с несохраняющей группой, содержащей три варианта выбора. Первый, ‹[^>"']›, совпадает с любым одиночным символом, кроме >, " или '. Другие два варианта совпадают с целой строкой, заключенной в кавычки или апострофы, что позволяет обеспечить совпадение с любыми значениями атрибутов, содержащими правые угловые скобки, которые не смогут интерпретироваться регулярным выражением, как конец тега.

Варианты

Замена списков тегов

Если необходимо отыскать любой тег из списка, потребуется внести простое изменение. Нужно все желаемые имена тегов заключить в группу и разделить их оператором выбора. Группа в данном случае используется для ограничения области действия операторов выбора (‹|›).

Следующее регулярное выражение совпадает с открывающими и закрывающими тегами <b>, <i>, и <big>. Замещающий текст, следующий еще ниже, замещает все эти теги соответствующими тегами или , при этом сохраняются все атрибуты:

```
<(/?)([bi]|em|big)\b((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится то же выражение в режиме свободного форматирования:

```
<                 #
(/?)              # Сохранить необязательный ведущий слэш в обратной ссылке 1
([bi]|em|big) \b  # Сохранить имя тега в обратной ссылке 2
(                 #     Сохранить атрибуты и пр. в обратной ссылке 3
    (?: [^>"']    #     Любой символ, кроме >, “ или ‘
      | “[^"]*”   #     Значение атрибута в кавычках
      | ‘[^']*’   #     Значение атрибута в апострофах
)*                #
)                 #
>                 #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Для совпадения с тегами <b> и <i> мы использовали символьный класс ‹[bi]›, вместо указания их имен по отдельности через символ оператора выбора (|), как это сделано для тегов и <big>. Символьный класс работает быстрее, чем оператор выбора, потому что для выполнения своей работы ему не требуется выполнять возвраты. Когда искомые варианты выбора отличаются единственным символом, лучше использовать символьный класс.

Кроме того, для имени тега была добавлена сохраняющая группа, в результате чего группа, сопоставляемая с атрибутом, теперь сохраняет свое совпадение в обратной ссылке 3. Несмотря на то, что в операции замены всех совпадений тегом нет никакой необходимости ссылаться на имя тега, тем не менее, наличие сохраненного имени тега в отдельной обратной ссылке поможет определять тип совпавшего тега в случае необходимости.

Чтобы сохранить все атрибуты при замене имени тега, можно использовать следующие примеры замещающего текста:

```
<$1strong$3>
```

```
Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP
```

```
<\1strong\3>
```

```
Диалекты замещающего текста: Python, Ruby
```

Чтобы удалить атрибуты совпавшего тега, достаточно убрать обратную ссылку 3 из замещающего текста:

```
<$1strong>
```

```
Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP
```

```
<\1strong>
```

```
Диалекты замещающего текста: Python, Ruby
```

См. также

Рецепт 8.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Рецепт 8.3, который является противоположностью по отношению к этому рецепту и показывает, как отыскивать любые теги, за исключением тех, что в списке.

8.3. Удаление всех XML-подобных тегов, за исключением и

Задача

Необходимо удалить все теги в испытуемом тексте, за исключением `<em>` и `<strong>`.

В другом случае требуется не только удалить все теги, отличные от `<em>` и `<strong>`, но также удалить и теги `<em>` и `<strong>`, содержащие атрибуты.

Решение

Это прекрасный повод задействовать негативную опережающую проверку (описывается в рецепте 2.16). Применительно к этой задаче негативная опережающая проверка позволит обеспечить совпадение с фрагментами текста, похожими на тег, *за исключением* случаев, когда сразу за открывающей угловой скобкой `<` или `</` следует определенное слово. Если затем все найденные совпадения заменить пустыми строками

(как это сделать, показано в рецепте 3.14), в тексте останутся только требуемые теги.

Решение 1: совпадение с тегами, за исключением и

```
</?(?!(?:em|strong)\b)[a-z](?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
< /?                  # Разрешить совпадение с закрывающими тегами
(?!                   # Негативная опережающая проверка
    (?: em | strong ) #     Список тегов, исключаемых из сопоставления
    \b                #     Граница слова позволит исключить совпадение
                      #     с частью тега
)                     #
[a-z]                 # Первый символ в имени тега должен быть a-z
(?: [^>"']            #     Любой символ, кроме >, " или '
  | "[^"]*"           #     Значение атрибута в кавычках
  | '[^']*'           #     Значение атрибута в апострофах
)*                    #
>                     #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Решение 2: совпадение с тегами, за исключением и , и с любыми тегами, содержащими атрибуты

Внеся одно изменение (заменив ‹\b› на ‹\s*›), можно заставить это регулярное выражение совпадать также с любыми тегами и , содержащими атрибуты:

```
</?(?!(?:em|strong)\s*>)[a-z](?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И снова это же регулярное выражение в режиме свободного форматирования:

```
< /?                  # Разрешить совпадение с закрывающими тегами
(?!                   # Негативная опережающая проверка
    (?: em | strong ) #     Список тегов, исключаемых из сопоставления
    \s* >             #     Исключить теги, содержащие атрибуты
)                     #
[a-z]                 # Первый символ в имени тега должен быть a-z
(?: [^>"']            #     Любой символ, кроме >, " или '
  | "[^"]*"           #     Значение атрибута в кавычках
```

```
  | '[^']*'          #     Значение атрибута в апострофах
)*                   #
>                    #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Регулярные выражения в этом рецепте имеют много общего с выражениями, приводившимися выше в этой главе и обеспечивающими совпадение с XML-подобными тегами. Кроме негативной опережающей проверки, добавленной с целью предотвратить совпадение с некоторыми тегами, эти регулярные выражения практически эквивалентны выражениям из раздела «Теги (X)HTML (не строго)» в рецепте 8.1. Другое значительное отличие заключается в том, что здесь имя тега не сохраняется в обратной ссылке 1.

Рассмотрим поближе нововведения в этом рецепте. Выражение в Решении 1 никогда не будет совпадать с тегами `<em>` и `<strong>` независимо от того, имеют они атрибуты или нет, но будет совпадать с любыми другими тегами. Выражение в решении 2 будет совпадать с теми же тегами, что и выражение в решении 1, и дополнительно совпадать с тегами `<em>` и `<strong>`, содержащими один или более атрибутов. В табл. 8.2 приводится несколько примеров испытуемых строк, иллюстрирующих вышесказанное.

Таблица 8.2. Несколько примеров испытуемых строк

Испытуемая строка	Решение 1	Решение 2
`<i>`	Совпадает	Совпадает
`</i>`	Совпадает	Совпадает
`<i style="font-size:500%; color:red;">`	Совпадает	Совпадает
`<em>`	Не совпадает	Не совпадает
`</em>`	Не совпадает	Не совпадает
`<em style="font-size:500%; color:red;">`	Не совпадает	Совпадает

Поскольку цель применения этих регулярных выражений состоит в том, чтобы заменить совпадения пустыми строками (то есть удалить теги), решение 2 надежнее защищает от возможности с помощью допустимых тегов `<em>` и `<strong>` изменять форматирование неожиданным образом и творить другие глупости.

В этом рецепте (до сих пор) мы преднамеренно избегали выражения «белый список», описывая, как оставить нетронутыми всего несколько тегов, потому что у этой фразы есть второе значение, связанное с темой безопасности. Существует множество способов обойти ограничения этого шаблона, используя специально подготовленный, злонамеренный код разметки HTML. Если вас беспокоит проблема возможных атак с применением злонамеренного кода HTML и атак типа «межсайтовый скриптинг» (XSS), безопаснее всего будет преобразовать все символы <, > и & в соответствующие им мнемонические ссылки (<, > и &) и вставить обратно теги, которые, как известно, безопасны (пока они не содержат атрибуты или используют только атрибуты из списка допустимых атрибутов). style – пример небезопасного атрибута, потому что некоторые броузеры позволяют встраивать в таблицы CSS фрагменты программного кода на языке сценариев. Например, чтобы после замены <, > и & мнемоническими ссылками вставить теги обратно, можно выполнить поиск с помощью регулярного выражения ‹<(/?)em>› и заменить совпадения с помощью замещающего текста «<$1em>» (для диалектов Python и Ruby, «<\1em>»).

Варианты

«Белый список» для определенных атрибутов

Рассмотрим следующие новые требования: необходимо обеспечить совпадение со всеми тегами, кроме <a>, и , с двумя исключениями. Все теги <a>, имеющие атрибуты, отличные от href или title, должны совпадать, кроме того, должны совпадать теги и , имеющие вообще хоть какие-нибудь атрибуты. Все совпавшие фрагменты должны быть удалены.

Другими словами, следует удалить все теги, кроме перечисленных в «белом списке» (<a>, и). Единственными допустимыми атрибутами являются href и title, причем они допускаются только внутри тегов <a>. В случае появления любого другого атрибута в любом теге весь тег должен удаляться.

Следующее регулярное выражение, которое решает эту задачу, показано как в обычном режиме, так и в режиме свободного форматирования:

```
<(?!(?:em|strong|a(?:\s+(?:href|title)\s*=\s*(?:"[^"]*"|'[^']*'))*)\s*>)↵
[a-z](?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
< /?            # Разрешить совпадение с закрывающими тегами
(?!             # Негативная опережающая проверка
```

```
    (?: em       #   Не совпадать с <em>...
      | strong   #   или <strong>...
      | a        #   или <a>...
        (?:      #     причем только с теми тегами <a>,
                 #     которые не содержат...
            \s+  #       атрибуты, отличные от href и/или title
            (?:href|title)
            \s*=\s*
            (?:"[^"]*"|'[^']*') # Значение атрибута в кавычках
                                # или апострофах
        )*
    )
    \s* >        # Избегать совпадения с этими тегами, когда они содержат...
)                #   только атрибуты, перечисленные выше
[a-z]            # Первый символ в имени тега должен быть буквой a-z
(?: [^>"']       #   Любой символ, кроме >, " или '
  | "[^"]*"      #   Значение атрибута в кавычках
  | '[^']*'      #   Значение атрибута в апострофах
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Мы сами определили границы области, для которой есть смысл применять такое сложное регулярное выражение. Если правила становятся более сложными, чем это, вероятно, лучше будет написать дополнительный программный код, основываясь на рецептах 3.11 и 3.16, и реализовать в нем проверку каждого совпавшего тега, чтобы определить, как его обрабатывать (основываясь на имени тега, содержащихся в нем атрибутов или какой-то иной информации).

См. также

Рецепт 8.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Рецепт 8.2, который является противоположностью по отношению к этому рецепту и показывает, как отыскивать только перечисленные теги.

8.4. Сопоставление с именами XML

Задача

Требуется проверить, является ли испытуемая строка допустимым *именем* в языке разметки XML (обобщенной синтаксической конструкцией). В языке XML точно определено, какие символы могут составлять имена, и эти правила применяются к именам элементов, атрибутов

и мнемоник, к инструкциям обработки и ко многому другому. Первым символом имени может быть буква, символ подчеркивания или двоеточие, остальная часть имени может представлять собой любую комбинацию букв, цифр, символов подчеркивания, двоеточий, дефисов и точек. Это приблизительное описание, но очень близкое к истине. Точный список допустимых символов зависит от используемой версии XML.

Кроме того, может потребоваться добавить шаблон, совпадающий с допустимыми именами, в другие регулярные выражения, предназначенные для работы с XML, когда дополнительная точность влечет за собой дополнительную сложность.

Ниже приводятся некоторые примеры допустимых имен:

- `thing`
- `_thing_2_`
- `:Российские-Вещь`
- `fantastic4:the.thing`
- 日本の物

Следует отметить, что в число допустимых входят также символы из алфавитов, отличных от Latin, включая даже иероглифы, как видно в последнем примере. Точно так же вслед за первым символом может следовать любая цифра Юникода, а не только арабские цифры 0–9.

Для сравнения ниже приводятся несколько примеров недопустимых имен, которые не должны совпадать с выражением:

- `thing!`
- `thing with spaces`
- `.thing.with.a.dot.in.front`
- `-thingamajig`
- `2nd_thing`

Решение

Подобно идентификаторам во многих языках программирования в языке XML определено множество символов, которые могут появляться в именах, причем только часть их может употребляться в качестве первого символа. Эти перечни символов существенно отличаются для версии XML 1.0, четвертое издание (и ниже), и версий XML 1.1 и 1.0, пятое издание. По сути в версии XML 1.1 в именах могут использоваться любые символы, допустимые в четвертом издании версии 1.0, плюс более миллиона других символов. Однако большинство дополнительных символов – не что иное, как просто позиции в таблице Юникода. Многие из них еще не присвоены символам, но уже считаются допу-

стимыми для совместимости в будущем, по мере расширения базы данных символов Юникода.

Ради лаконичности, когда в этом рецепте мы будем ссылаться на версию XML 1.0, будут подразумеваться издания с первого по четвертое версии XML 1.0. Когда будет говориться об именах XML 1.1, также будет подразумеваться пятое издание версии XML 1.0. Пятое издание стало официальной рекомендацией консорциума W3C только в конце ноября 2008 года, почти через пять лет после появления XML 1.1.

Регулярные выражения в этом рецепте содержат якорные метасимволы, совпадающие с началом и концом текста (‹^···$› или ‹\A···\Z›), что обеспечивает возможность совпадения с испытуемым текстом только целиком. Если у вас появится необходимость вставлять эти шаблоны в более длинные регулярные выражения, выполняющие сопоставление с элементами XML, не забудьте удалить якоря в начале и в конце шаблона. Якорные метасимволы описываются в рецепте 2.5.

Имена XML 1.0 (приблизительно)

```
\A[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, чтобы можно было использовать свойства Юникода (‹\p{···}›). В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона /u.

Свойства Юникода не поддерживаются в JavaScript, Python и Ruby 1.8. Регулярное выражение для имен XML 1.1, которое следует ниже, не использует свойства Юникода и поэтому может быть лучшим выбором в этих языках программирования. В разделе «Обсуждение» этого рецепта на стр. 553 подробно описывается, почему может быть лучше использовать решение для XML 1.1, даже в диалектах, поддерживающих свойства Юникода.

Имена XML 1.1 (точно)

Ниже следуют три версии одного и того же регулярного выражения, учитывающие различия между диалектами. Единственное отличие между первым и вторым заключается в якорных метасимволах, используемых в начале и в конце регулярного выражения. В третьей версии для определения кодовых пунктов Юникода выше шестнадцатеричного значения FF (десятичное число 255) вместо <\u> используется конструкция ‹\x{···}›.

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-\u1FFF\u200C↵
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-\uFFFD]↵
[:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-\u1FFF↵
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-↵
\uFDCF\uFDF0-\uFFFD]*\Z
```

Параметры: нет
Диалекты: .NET, Java, Python, Ruby 1.9

```
^[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-\u1FFF\u200C↵
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-\uFFFD]↵
[:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-\u1FFF↵
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-↵
\uFDCF\uFDF0-\uFFFD]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, Python

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\x{2FF}\x{370}-\x{37D}\x{37F}-\x{1FFF}↵
\x{200C}\x{200D}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}↵
\x{F900}-\x{FDCF}\x{FDF0}-\x{FFFD}][:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6↵
\xF8-\x{36F}\x{370}-\x{37D}\x{37F}-\x{1FFF}\x{200C}\x{200D}\x{203F}↵
\x{2040}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}\x{F900}-↵
\x{FDCF}\x{FDF0}-\x{FFFD}]*\Z
```

Параметры: нет
Диалекты: PCRE, Perl

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, чтобы можно было использовать метапоследовательности ‹\x{…}› для значений выше шестнадцатеричного числа FF. В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона /u.

В Ruby 1.8 поддержка Юникода регулярными выражениями отсутствует полностью, однако в разделе «Варианты» этого рецепта на стр. 555 приводятся возможные альтернативные решения, обладающие меньшей точностью.

Несмотря на то, что мы утверждали, что эти регулярные выражения точно следуют правилам, применяемым к именам в версии XML 1.1, тем не менее, в действительности это справедливо только для символов с 16-битными кодами (позиции от 0x0 до 0xFFFF).

Дополнительно в XML 1.1 допускается использовать после первого символа имени еще 917503 кодовых пунктов, расположенных между позициями 0x10000 и 0xEFFFF. Только PCRE, Perl и Python способны ссылаться на кодовые пункты выше 0xFFFF, но вам едва ли доведется встретиться с ними в природе (прежде всего потому, что большая часть позиций в этом диапазоне вообще не присвоена каким-либо символам). Если возникнет необходимость добавить поддержку для этих дополнительных кодовых пунктов, в PCRE и Perl можно добавить ‹\x{10000}-

\x{EFFFF}› в конец второго символьного класса, а в Python можно добавить ‹\U00010000-\U000EFFFF› (обратите внимание, что за символом верхнего регистра U обязательно должно следовать восемь шестнадцатеричных цифр). Но даже без добавления этого массивного диапазона перечень символов имен в XML 1.1 намного обширнее, чем в XML 1.0.

Обсуждение

Так как многие регулярные выражения в этой главе выполняют сопоставление с элементами XML, этот рецепт в значительной степени служит целям обеспечения полноты обсуждения шаблонов, которые могут использоваться, когда требуется четко определить, как следует выполнять сопоставление с именами тегов и атрибутов. В остальных рецептах мы стараемся придерживаться более простых и менее точных шаблонов, отдавая предпочтение удобочитаемости и эффективности.

Итак, погрузимся чуть глубже в правила, которые стоят за этими шаблонами.

Имена XML 1.0

Для определения правил, предъявляемых к именам, в спецификации XML 1.0 используется прием «белого списка» и явно перечисляются все допустмые символы. Первым символом имени может быть двоеточие (:), подчеркивание (_) и почти любой символ из следующих категорий Юникода:

- Строчная буква (Ll)
- Заглавная буква (Lu)
- Буква в регистре названия (Lt)
- Буква без регистра (Lo)
- Цифра (Nl)

Вслед за первым символом в дополнение к уже упомянутым символам могут использоваться дефис (-), точка (.) и любой другой символ из следующих категорий:

- Маркер (M), объединяющий следующие подкатегории: маркер, не занимающий места (Mn), маркер, занимающий место (Mc), и объемлющий маркер (Me)
- Модифицированная буква (Lm)
- Десятичная цифра (Nd)

Эти правила приводят к первому регулярному выражению в разделе «Решение» этого рецепта. Это регулярное выражение еще раз приводится ниже в режиме свободного форматирования:

```
\A                                # Начало текста
[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}] # Первый символ имени
```

```
[:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*     # Символы имени (ноль или более)
\Z                                 # Конец текста
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Снова напомним, что библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона `/u`.

Следует отметить, что во втором символьном классе вместо отдельных подкатегорий букв (Ll, Lu, Lt, Lo и Lm) использована их базовая категория ‹`\p{L}`›.

Ранее уже отмечалось, что здесь используются приблизительные правила. Это обусловлено двумя причинами. Во-первых, спецификация XML 1.0 (здесь не имеется в виду пятое издание или более поздняя версия) перечисляет ряд исключений из этих правил. Во-вторых, списки символов в XML 1.0 основаны на стандарте Юникода 2.0, который был выпущен в 1996 году. В последующих версиях стандарта на Юникод была добавлена поддержка новых алфавитов, символы из которых не допускается использовать в именах XML 1.0. Однако для приведения регулярного выражения к версии Юникода, используемой механизмом регулярных выражений, то есть для ограничения его возможностью сопоставления с символами Юникода 2.0, пришлось бы написать монстроподобный шаблон, заполненный сотнями диапазонов и кодовых пунктов. Если вы действительно захотите создать такого монстра, обращайтесь к спецификации на четвертое издание XML 1.0 (*http://www.w3.org/TR/2006/REC-xml-20060816*), раздел 2.3 «Common Syntactic Constructs» и приложение B «Character Classes».

Ниже приводятся несколько способов сократить представленные выше регулярные выражения в разных диалектах.

Диалекты Perl и PCRE позволяют объединить строчные буквы (Ll), заглавные буквы (Lu) и буквы в регистре названия (Lt) в специальную категорию «буквы, имеющие регистр» (L&). Кроме того, эти диалекты позволяют опускать фигурные скобки в экранированной последовательности ‹`\p{…}`›, если внутри скобок используется только один символ. Мы задействовали эту особенность в следующем регулярном выражении, использовав конструкцию ‹`\pL\pM`› вместо ‹`\p{L}\p{M}`›:

```
\A[:_\p{L&}\p{Lo}\p{Nl}][:_\-.\pL\pM\p{Nd}\p{Nl}]*\Z
```

Параметры: нет
Диалекты: PCRE, Perl

Диалект .NET поддерживает операцию вычитания символьных классов, которая используется здесь для вычитания подкатегории Lm из категории L вместо того, что перечислять все остальные подкатегории букв:

```
\A[:_\p{L}\p{Nl}-[\p{Lm}]][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*\Z
```

Параметры: нет
Диалект: .NET

Диалект Java, так же, как PCRE и Perl, позволяет опускать фигурные скобки вокруг односимвольных имен категорий Юникода. Кроме того, следующее регулярное выражение использует преимущества более сложной операции вычитания символьных классов (реализованной как пересечение с инвертированным классом) для вычитания подкатегории Lm из L:

```
\A[:_\pL\p{Nl}&&[^\p{Lm}]][:_\-.\pL\pM\p{Nd}\p{Nl}]*\Z
```

Параметры: нет
Диалект: Java

Диалекты JavaScript, Python и Ruby 1.8 вообще не поддерживают категории Юникода. Диалект Ruby 1.9 не поддерживает эти только что описанные интересные особенности, но поддерживает более переносимую версию этих регулярных выражений, продемонстрированную в разделе «Решение» этого рецепта.

Имена XML 1.1

В спецификации XML 1.0 была допущена ошибка явной привязки к стандарту Юникода 2.0. Более поздние версии Юникода добавили поддержку еще большего количества символов, часть которых принадлежит алфавитам, которые раньше вообще не принимались во внимание (например, Чероки, Эфиопский и Монгольский). Так как XML претендует на положение универсального формата, была предпринята попытка ликвидировать эту проблему в версиях XML 1.1 и в пятом издании XML 1.0. При определении символов, допустимых в именах, в этих более поздних версиях был выполнен переход от стратегии «белого списка» к стратегии «черного списка», чтобы обеспечить поддержку не только тех символов, что были добавлены, начиная с версии Юникода 2.0, но и тех, что будут добавлены в будущем.

Эта новая стратегия, когда разрешено все, что явно не запрещено, улучшает будущую совместимость, а также упрощает и сокращает регулярные выражения, точно следующие правилам. Именно поэтому регулярные выражения для работы с именами XML 1.1 помечены как точные, тогда как регулярные выражения для XML 1.0 – как приблизительные.

Варианты

В некоторых рецептах этой главы (например, рецепт 8.1), сегменты выражения, имеющие дело с именами XML, или не используют никакие ограничения, или запрещают дополнительные алфавиты и другие сим-

волы, которые в действительности являются допустимыми. Это было сделано, чтобы упростить их изучение. Однако если необходимо разрешить использование других алфавитов, обеспечивая при этом базовый уровень ограничений (и вам не требуется более точная проверка имен из первых регулярных выражений в этом рецепте), можно воспользоваться следующими регулярными выражениями.

Мы оставили якорные метасимволы начала и конца текста в этих регулярных выражениях, потому что они должны использоваться не самостоятельно, а как часть более длинного выражения.

Это первое регулярное выражение просто исключает возможность совпадения с символами, которые могут использоваться в тегах как символы-разделители, а также не допускает возможность совпадения с цифрой в первом символе:

```
[^\d\s"'/<=>][^\s"'/<=>]*
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится еще один, более короткий способ добиться того же самого. Вместо использования двух отдельных символьных классов в нем применяется негативная опережающая проверка, чтобы отвергнуть совпадение с цифрой в первом символе. Этот запрет применяется только к первому символу, хотя квантификатор ‹+› после символьного класса позволяет регулярному выражению совпадать с неограниченным числом символов:

```
(?!\d)[^\s"'/<=>]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Джон Коуэн (John Cowan), один из разработчиков спецификации XML 1.1, объясняет, какие символы недопустимы в именах XML 1.1 и почему, в своем сообщении по адресу: *http://recycledknowledge.blogspot.com/2008/02/which-characters-are-excluded-in-xml.html*.

Документ «Background to Changes in XML 1.0, 5th Edition» по адресу *http://www.w3.org/XML/2008/02/xml10_5th_edition_background.html*, где обсуждается обоснование обратного переноса правил, применяемых к именам в XML 1.1, в версию XML 1.0, пятое издание.

8.5. Преобразование простого текста в HTML добавлением тегов <p> и

Задача

Имеется самый обычный текст, такой как многострочное значение, отправляемое с помощью формы, который необходимо преобразовать во фрагмент разметки HTML для отображения в веб-странице. Параграфы, отделяемые друг от друга двумя разрывами строки, должны быть заключены в теги `<p>…</p>`. Кроме того, разрывы строк следует заменить тегами `<br>`.

Решение

Эту проблему можно решить, выполнив четыре простых шага. В большинстве языков программирования только на этапе выполнения двух средних шагов можно извлечь выгоду от применения регулярных выражений.

Шаг 1: замена специальных символов HTML мнемоническими ссылками

Так как выполняется преобразование простого текста в разметку HTML, на первом этапе необходимо заменить три специальных символа HTML – `&`, `<` и `>` – мнемоническими ссылками (табл. 8.3). В противном случае получившаяся разметка может приводить к неожиданным результатам при отображении в веб-броузере.

Таблица 8.3. Замена специальных символов HTML

Искомый символ	Замещается на
‹&›	«&»
‹<›	«<»
‹>›	«>»

В первую очередь следует заменить символы амперсанда (`&`), потому что в ходе операции замены в испытуемый текст будут добавлены дополнительные амперсанды, являющиеся частью мнемонических ссылок.

Шаг 2: замена всех разрывов строк тегом

Регулярное выражение:

```
\r\n?|\n
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\R
```

Параметры: нет
Диалекты: PCRE 7, Perl 5.10

Замещающий текст:

```
<br>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Шаг 3: замена двойных тегов
 на

Регулярное выражение:

```
<br>\s*<br>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
</p><p>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Шаг 4: заключить весь текст в теги ...

Этот шаг является простой операцией конкатенации строк и не требует применения регулярных выражений.

Пример для JavaScript

Чтобы объединить все четыре шага, создадим функцию JavaScript с именем `html_from_plaintext`. Эта функция принимает строку, обрабатывает ее, выполняя только что описанные шаги, и возвращает новую строку с разметкой HTML:

```
function html_from_plaintext (subject) {
    // шаг 1 (поиск простого текста)
    subject = subject.replace(/&/g, "&amp;").
                      replace(/</g, "&lt;").
                      replace(/>/g, "&gt;");
    // шаг 2
    subject = subject.replace(/\r\n?|\n/g, "<br>");

    // шаг 3
    subject = subject.replace(/<br>\s*<br>/g, "</p><p>");
```

```
    // шаг 4
    subject = "<p>" + subject + "</p>";

    return subject;
}

/*
html_from_plaintext("Test.")            -> "<p>Test.</p>"
html_from_plaintext("Test.\n")          -> "<p>Test.<br></p>"
html_from_plaintext("Test.\n\n")        -> "<p>Test.</p><p></p>"
html_from_plaintext("Test1.\nTest2.")   -> "<p>Test1.<br>Test2.</p>"
html_from_plaintext("Test1.\n\nTest2.") -> "<p>Test1.</p><p>Test2.</p>"
html_from_plaintext("< AT&T >")         -> "<p>&lt; AT&amp;T &gt;</p>"
*/
```

В конец фрагмента программного кода добавлено несколько примеров результатов работы этой функции применительно к различным испытуемым строкам. Тем, кто не знаком с языком JavaScript, следует обратить внимание на модификатор `/g`, добавляемый в конец каждого литерала регулярного выражения, который вынуждает метод `replace` заместить все совпадения с шаблоном, а не только самое первое. Метапоследовательность `\n`, присутствующая в строке с испытуемым текстом, в языке JavaScript добавляет символ перевода строки (позиция 0x0A в таблице ASCII) в строковый литерал.

Обсуждение

Шаг 1: замена специальных символов HTML мнемоническими ссылками

Проще всего реализовать этот шаг, выполнив три отдельные операции поиска с заменой (в табл. 8.3, выше, приводится список замещающих мнемоник). Для глобальных операций поиска с заменой в JavaScript всегда используются регулярные выражения, но в других языках программирования можно получить более высокую производительность, выполняя обычную операцию поиска с заменой по простому тексту.

Шаг 2: замена всех разрывов строк тегом

На этом шаге для поиска разрывов строк в соответствии с соглашениями для Windows/MS-DOS (CRLF), UNIX/Linux/OS X (LF) и устаревшей версии Mac OS (CR) используется регулярное выражение `‹\r\n?|\n›`. Пользователи Perl 5.10 и PCRE 7 могут использовать специальный метасимвол `‹\R›` (обратите внимание на верхний регистр символа `R`) вместо сопоставления с различными последовательностями, обозначающими разрывы строк.

Замена всех разрывов строк тегами `<br>` перед добавлением тегов параграфов на следующем шаге позволяет сохранить простоту всего действия, так как дает возможность добавить пробельный символ между

тегами `</p><p>` в последующей операции поиска с заменой. Это поможет сохранить код разметки HTML удобочитаемым, не смешивая все вместе.

Если предпочтительнее использовать одиночные теги в стиле XHTML, то вместо замещающего текста «`<br>`» следует использовать «`<br•/>`». Кроме того, чтобы учесть это изменение, необходимо изменить регулярное выражение, применяемое на третьем шаге.

Шаг 3: замена двойных тегов
 на

Наличие двух разрывов строки, следующих подряд, отмечает конец одного параграфа и начало другого, поэтому замещающий текст для этого шага содержит закрывающий тег `</p>`, за которым следует открывающий тег `<p>`. Если испытуемый текст состоит из единственного параграфа (то есть в нем отсутствуют двойные разрывы строк), подстановка производиться не будет. На шаге 2 было заменено несколько разрывов строк (вместо которых в тексте были оставлены теги `<br>`), поэтому на данном шаге можно было бы выполнить операцию поиска с заменой простого текста. Однако применение регулярного выражения упрощает работу и позволяет игнорировать пробельные символы между разрывами строки. Дополнительные символы пробела все равно не отображаются в документе HTML.

Если создается фрагмент разметки XHTML и поэтому разрывы строк были замещены текстом «`<br•/>`», вместо «`<br>`», необходимо заменить регулярное выражение, используемое на этом шаге, на `‹<br•/>\s*<br•/>›`.

Шаг 4: заключить весь текст в теги …

Шаг 3 добавляет разметку между параграфами. Теперь необходимо добавить тег `<p>` в самое начало испытуемого текста и закрывающий тег `</p>` в самый конец. Этот шаг завершает процесс, будь в тексте 1 параграф или 100.

См. также

Рецепт 4.10, включающий дополнительные сведения о метасимволе `‹\R›` из диалектов Perl и PCRE и показывающий, как организовать совпадение с дополнительными, необычными разделителями строк, которые поддерживаются метасимволом `‹\R›`.

8.6. Поиск определенных атрибутов в XML-подобных тегах

Задача

Требуется отыскать внутри файла (X)HTML или XML теги, содержащие определенный атрибут, такой как `id`.

Этот рецепт охватывает несколько вариантов одной и той же задачи. Предположим, что необходимо обеспечить совпадение со всеми типами строк, приведенными ниже, с помощью отдельных регулярных выражений:

- Теги с атрибутом `id`.
- Теги `<div>` с атрибутом `id`.
- Теги с атрибутом `id`, имеющим значение `my-id`.
- Теги, содержащие имя класса `my-class` внутри значения атрибута `class` (имена классов отделяются пробельными символами).

Решение

Теги с атрибутом id (на скорую руку)

Если необходимо быстро выполнить поиск в текстовом редакторе, обеспечивающем возможность предварительного просмотра результатов, добиться цели поможет следующее (максимально упрощенное) регулярное выражение:

```
<[^>]+\sid\b[^>]*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже то же регулярное выражение разложено в режиме свободного форматирования:

```
<        # Начало тега
[^>]+    # Имя тега, атрибуты и пр.
\s id \b # Имя искомого атрибута как целое слово
[^>]*    # Оставшаяся часть тега, включая значение атрибута id
>        # Конец тега
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Теги с атрибутом id (более надежно)

В отличие от только что рассмотренного регулярного выражения, следующий подход к этой же задаче поддерживает поиск значений атрибутов в кавычках, содержащих литерал >, и не будет выдавать совпадение для тегов, в которых последовательность символов `id` содержится в одном из значений атрибутов:

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
<                       #
(?: [^>"']              # Имена тега и атрибута и пр.
  | "[^"]*"             #     ...и значения атрибутов в кавычках
  | '[^']*'             #
)+?                     #
\s id                   # Имя искомого атрибута как целое слово
\s* = \s*               # Разделитель имени-значения атрибута
( "[^"]*" | '[^']*' ) # Сохранить значение атрибута в обратной ссылке 1
(?: [^>"']              # Остальные символы
  | "[^"]*"             #     ...и значения атрибутов в кавычках
  | '[^']*'             #
)*                      #
>                       #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это регулярное выражение сохраняет значение атрибута id и окружающие его кавычки в обратной ссылке 1. Это позволит использовать значение в программном коде за пределами регулярного выражения или в строке замещающего текста. Если значение атрибута не потребуется, можно вместо сохраняющей группировки использовать несохраняющую или вообще заменить всю последовательность ‹\s*=\s*("[^"]*"|'[^']*')› метасимволом ‹\b›. В этом случае совпадение с оставшейся частью тега, в том числе и со значением атрибута id, обеспечит остальная часть регулярного выражения.

Теги с атрибутом id

Чтобы отыскать определенный тег, в начало предыдущего регулярного выражения необходимо добавить его имя и внести еще пару незначительных изменений. В следующем примере после открывающей угловой скобки ‹<› мы добавили ‹div\s›. Метасимвол ‹\s› (пробельный символ) гарантирует невозможность совпадения с тегами, чьи имена просто начинаются с трех букв «div». Заранее известно, что за именем тега обязательно будет следовать пробельный символ, потому что у тегов, которые требуется отыскать, имеется, по крайней мере, один атрибут (id). Дополнительно мы заменили последовательность ‹+?\sid› на ‹*?\bid›, чтобы выражение работало, когда id является первым атрибутом тега и отсутствуют дополнительные отделяющие символы (кроме первого пробела) после имени тега:

```
<div\s(?:[^>"']|"[^"]*"|'[^']*')*?\bid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же выражение в режиме свободного форматирования:

```
<div \s                # Имя тега и следующий за ним пробельный символ
(?: [^>"']             # Имена тега и атрибута и пр.
  | "[^"]*"            #     ...и значения атрибутов в кавычках
  | '[^']*'            #
)*?                    #
\b id                  # Имя искомого атрибута как целое слово
\s* = \s*              # Разделитель имени-значения атрибута
( "[^"]*" | '[^']*' ) # Сохранить значение атрибута в обратной ссылке 1
(?: [^>"']             # Остальные символы
  | "[^"]*"            #     ...и значения атрибутов в кавычках
  | '[^']*'            #
)*                     #
>                      #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Теги с атрибутом id, имеющим значение «my-id»

На этот раз, по сравнению с регулярным выражением из раздела «Теги с атрибутом id (более надежно)» на стр. 561 мы удалили сохраняющие круглые скобки вокруг значения атрибута `id`, потому что оно уже известно заранее. В частности, подвыражение `‹("[^"]*"|'[^']*')›` мы заменили на `‹(?:"my-id"|'my-id')›`:

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*(?:"my-id"|'my-id')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И версия в режиме свободного форматирования:

```
< #
(?: [^>"']    # Имена тега и атрибута и пр.
  | "[^"]*"   # ...и значения атрибутов в кавычках
  | '[^']*'   #
)+?           #
\s id         # Имя искомого атрибута как целое слово
\s* = \s*     # Разделитель имени-значения атрибута
(?: "my-id"   # Искомое значение атрибута
  | 'my-id' ) #     ...окруженное кавычками или апострофами
(?: [^>"']    # Остальные символы
  | "[^"]*"   #     ...и значения атрибутов в кавычках
  | '[^']*'   #
)*            #
>             #
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Если взглянуть на подвыражение ‹(?:"my-id"|'my-id')›, то здесь можно было бы избежать повторения «my-id» (немного потеряв в эффективности), использовав подвыражение ‹(["'])my-id\1›. Использование сохраняющей группировки и обратной ссылки гарантирует, что искомое значение начинается и заканчивается кавычками одного и того же типа.

Теги, содержащие «my-class» в значении атрибута class

Если в предыдущих регулярных выражениях планка оправданности применения регулярного выражения еще не была преодолена, то в данном случае, очевидно, мы достигли границы, за которой неразумно пытаться решать задачу с помощью единственного регулярного выражения. Разбиение процесса на несколько регулярных выражений поможет в достижении поставленной цели, поэтому мы разобьем поиск на три части. Первое регулярное выражение совпадает с тегами, второе отыскивает в них атрибут class (и сохраняет его значение в обратной ссылке) и, наконец, выполняется поиск значения my-class.

Поиск тегов:

```
<(?:[^>"']|"[^"]*"|'[^']*')+>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Рецепт 8.1 посвящен вопросам сопоставления с XML-подобными тегами. Там объясняется, как действует только что представленное регулярное выражение, и демонстрируются альтернативные решения, обеспечивающие различную степень сложности и точности.

Затем, с помощью программного кода, который приводится в рецепте 3.13, можно выполнить поиск атрибута class внутри каждого совпадения, используя следующее регулярное выражение:

```
^(?:[^>"']|"[^"]*"|'[^']*')+?\sclass\s*=\s*("[^"]*"|'[^']*')
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Оно сохраняет все значение атрибута class и окружающие его кавычки в обратной ссылке 1. Совпадение со всем, что находится перед атрибутом class, обеспечивает подвыражение ‹^(?:[^>"']|"[^"]*"|'[^']*')+?›, которое за один шаг совпадает со значением в кавычках, чтобы избежать попыток поиска слова «class» внутри значений других атрибутов. Правая часть выражения заканчивает совпадение, как только будет достигнут конец значения атрибута class. Все, что в теге следует дальше,

для нас не имеет никакого значения, поэтому нет никаких причин продолжать сопоставление до конца найденного тега.

Символ крышки в начале регулярного выражения привязывает его к началу испытуемого текста. Это никак не влияет на совпадение, но он препятствует механизму регулярных выражений начинать новые попытки сопоставления (которые неизбежно потерпят неудачу) во всех позициях символов в строке, если сопоставление с начала строки оказалось неудачным.

Наконец, если оба предыдущих регулярных выражения благополучно обнаружили совпадение, можно выполнить поиск в значении обратной ссылки 1 второго регулярного выражения, используя следующий шаблон:

```
(?:^|\s)my-class(?:\s|$)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Так как имена классов отделяются пробельными символами, имя `my-class` должно ограничиваться с обеих сторон либо пробельным символом, либо вообще ничем. Если бы не тот факт, что имена классов могут включать дефисы, вместо двух несохраняющих групп можно было бы использовать метасимволы границы слова. Однако дефис создает границы слова и поэтому подвыражение `‹\bmy-class\b›` может совпасть с частью имени `not-my-class`.

Обсуждение

В разделе «Решение» этого рецепта уже подробно описано, как действуют эти регулярные выражения, поэтому мы не будет повторяться. Не следует забывать, что регулярные выражения часто бывают не идеальным решением задачи поиска в разметке, особенно когда достигается уровень сложности, как в этом рецепте. Прежде чем задействовать такие регулярные выражения, всегда следует посмотреть – не лучше ли использовать альтернативные решения, такие как применение XPath, парсера SAX или использование DOM. Мы включили эти регулярные выражения, потому что не так уж необычно для людей стараться справиться с решением подобных задач, но не говорите, что мы вас не предупреждали. Хотелось бы надеяться, что мы смогли показать некоторые из проблем, возникающих при поиске в разметке, и помогли вам избежать еще более непродуманных решений.

См. также

Рецепт 8.7, решающий концептуально противоположную задачу и отыскивающий теги, не содержащие определенный атрибут.

8.7. Добавление атрибута cellspacing в теги <table>, где этот атрибут отсутствует

Задача

Необходимо отыскать в файле (X)HTML и добавить `cellspacing="0"` во все таблицы, не имеющие атрибута `cellspacing`.

Этот рецепт служит примером добавления атрибутов в XML-подобные теги, которые еще не имеют этих атрибутов. Вы можете заменить имя тега и атрибута, а также значение атрибута на любое другое.

Решение

Выражение 1: упрощенное решение

Для совпадения с тегами `<table>`, не содержащими слово `cellspacing`, можно использовать негативную опережающую проверку, как показано ниже:

```
<table\b(?![^>]*?\scellspacing\b)([^>]*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же выражение в режиме свободного форматирования:

```
<table \b           # Совпадение с "<table",
                    # за которым следует граница слова
(?!                 # Проверить, что выражение ниже не совпадет здесь
  [^>]              #   Совпадение с любым символом, кроме ">"...
    *?              #     ноль или более раз, как можно меньше (минимальный)
  \s cellspacing \b #   Совпадение с "cellspacing", как с полным словом
)                   #
(                   # Сохранить совпадение с выражением ниже в обр. ссылке 1
  [^>]              #   Совпадение с любым символом, кроме ">"...
    *               #     Ноль или более раз, как можно больше
                    #     (максимальный)
)                   #
>                   # Совпадение с литералом ">" в конце тега
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Выражение 2: более надежное решение

В следующем регулярном выражении мы заменили оба экземпляра инвертированного символьного класса `‹[^>]›` из упрощенного решения на `‹(?:[^>"']|"[^"]*"|'[^']*')›`. Это повысило надежность регулярного выражения, во-первых, потому что была добавлена поддержка атрибутов в кавычках, которые могут содержать литерал «>», а во-вторых, новый

шаблон препятствует совпадению с тегом, где слово «cellspacing» просто содержится внутри значения атрибута.

Ниже приводится регулярное выражение со всеми только что описанными изменениями:

```
<table\b(?!(?:[^>"']|"[^"]*"|'[^']*')*?\scellspacing\b)↵
((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

А ниже это же выражение в режиме свободного форматирования:

```
<table \b           # Совпадение с "<table", за которым следует
                    # граница слова
(?!                 # Проверить, что выражение ниже не совпадет здесь
  (?: [^>"']        #   Совпадение с любым символом, кроме >, " и '
    | "[^"]*"       #     Или значение в кавычках
    | '[^']*'       #     Или значение в апострофах
  )*?               #   Ноль или более раз, как можно меньше (минимальный)
  \s cellspacing \b #   Совпадение с "cellspacing" как с полным словом
)                   #
(                   # Сохранить совпадение с выражением ниже в обр. ссылке 1
  (?: [^>"']        #   Совпадение с любым символом, кроме >, " и '
    | "[^"]*"       #     Или значение в кавычках
    | '[^']*'       #     Или значение в апострофах
  )*                #   Ноль или более раз, как можно больше (максимальный)
)                   #
>                   #
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Вставка нового атрибута

Все регулярные выражения, показанные в этом рецепте, могут использовать один и тот же замещающий текст, потому что все регулярные выражения сохраняют атрибуты совпавшего тега `<table>` (если таковые имеются) в обратной ссылке 1. Это позволяет вставлять атрибуты обратно, как часть замещающего значения, при добавлении нового атрибута `cellspacing`. Ниже приводятся необходимые замещающие строки:

```
<table•cellspacing="0"$1>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<table•cellspacing="0"\1>
```

Диалекты замещающего текста: Python, Ruby

В рецепте 3.15 показано, как реализовать поиск с заменой, когда в замещающем тексте используется обратная ссылка.

Обсуждение

Чтобы понять, как действуют эти регулярные выражения, разберем сначала упрощенное решение. Это решение, как вы увидите, содержит четыре логические части.

Первая часть, ‹<table\b›, совпадает со строкой литералов <table, за которой следует граница слова (‹\b›). Граница слова предотвращает возможность совпадения с тегами, имена которых просто начинаются с последовательности символов «table». Несмотря на то, что это может показаться необязательным при работе с (X)HTML (поскольку в этом языке разметки отсутствуют теги с именами, например, «tablet», «tableau» или «tablespoon»), тем не менее, это хорошая практика, которая поможет избежать появления ошибок, когда потребуется адаптировать это регулярное выражение для поиска других тегов.

Вторая часть выражения, ‹(?![^>]*?\scellspacing\b)›, — это негативная опережающая проверка. Она не поглощает символы испытуемого текста, совпавшие с ней, она проверяет, потерпит ли неудачу попытка сопоставления, если слово cellspacing появится где-нибудь в пределах открывающего тега. Так как предполагается добавить атрибут cellspacing во все совпадения, то нам необходимо исключить совпадение с тегами, уже имеющими этот атрибут.

Так как опережающая проверка заглядывает за текущую позицию попытки сопоставления, в ней используется начальная конструкция ‹[^>]*?›, позволяющая в процессе поиска заглядывать вперед настолько, насколько потребуется, пока не будет обнаружено нечто, похожее на конец тега (первое вхождение символа >). Остальная часть подвыражения в опережающей проверке (‹\scellspacing\b›) просто совпадает со строкой литералов «cellspacing», как с полным словом. Мы предусмотрели проверку совпадения с начальным пробельным символом (‹\s›), потому что он всегда должен отделять имя атрибута от имени тега или от предыдущих атрибутов. Мы предусмотрели проверку совпадения с границей слова вместо совпадения с другим пробельным символом, потому что граница слова полностью обеспечивает совпадение со строкой cellspacing как с целым словом, даже если атрибут не имеет значения или если непосредственно за именем атрибута следует знак равенства.

Двигаясь дальше, мы переходим к третьей части регулярного выражения: ‹([^>]*)›. Это инвертированный символьный класс со следующим за ним квантификатором «ноль или более раз», обернутый в сохраняющую группу. Сохранение этой части совпадения позволяет легко возвращать назад атрибуты, присутствующие в совпавшем теге, как часть замещающего текста. В отличие от негативной опережающей проверки, эта часть фактически добавляет атрибуты тега в строку совпадения с регулярным выражением.

Наконец, регулярное выражение совпадает с литералом ‹>› в конце тега.

Регулярное выражение 2, так называемая более надежная версия, действует точно так же, как и только что описанное выражение, за исключением того, что оба экземпляра инвертированного символьного класса ‹[^>]› были заменены на ‹(?:[^>"']|"[^"]*"|'[^']*')›. Этот более длинный шаблон перешагивает через значения атрибутов в кавычках или апострофах за один шаг.

Что касается замещающего текста, он может использоваться в обеих версиях регулярного выражения и будет замещать каждый совпавший тег `<table>` новым тегом, включающим `cellspacing` в качестве первого атрибута, за которым следуют все остальные атрибуты, имевшиеся в оригинальном теге (обратная ссылка 1).

См. также

Рецепт 8.6, решающий концептуально противоположную задачу и отыскивающий теги, содержащие определенный атрибут.

8.8. Удаление XML-подобных комментариев

Задача

Требуется удалить комментарии из документа (X)HTML или XML. Например, требуется удалить комментарии из веб-страницы, прежде чем передать ее броузеру, чтобы уменьшить размер файла страницы и тем самым уменьшить время загрузки для тех, у кого медленное подключение к Интернету.

Решение

Отыскать комментарии совсем несложно благодаря наличию минимальных квантификаторов. Ниже приводится выражение, выполняющее эту работу:

```
<!--.*?-->
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Достаточно прямолинейно. Как обычно, из-за отсутствия в JavaScript режима «точке соответствуют границы строк» необходимо заменить точку символьным классом, включающим все символы, чтобы регулярное выражение могло совпадать с многострочными комментариями. Ниже приводится версия выражения для JavaScript:

```
<!--[\s\S]*?-->
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы удалить комментарии, нужно заменить все совпадения пустыми строками (то есть ничем). В рецепте 3.14 приводится программный код, выполняющий замену всех совпадений с регулярным выражением.

Обсуждение

Принцип действия

В начале и в конце этого регулярного выражения присутствуют строки литералов ‹<!--› и ‹-->›. Поскольку ни один из этих символов не имеет специального значения в регулярных выражениях (за исключением дефиса, который в символьных классах образует диапазоны), их не требуется экранировать. Остается только исследовать фрагмент ‹.*?› или ‹[\s\S]*?› в середине выражения.

Благодаря режиму «точке соответствуют границы строк», точка в первом регулярном выражении совпадает с любым одиночным символом. В версии для JavaScript место точки занимает символьный класс ‹[\s\S]›. Однако эти два регулярных выражения полностью эквивалентны. Метасимволу ‹\s› соответствуют любые пробельные символы, а метасимволу ‹\S› соответствуют все остальные символы. Их объединение обеспечивает совпадение с любым символом.

Минимальный квантификатор ‹*?› повторяет предшествующий ему элемент «любой символ» ноль или более раз, минимально возможное число раз. Благодаря этому предшествующая конструкция будет повторяться, пока не будет встречено первое вхождение -->, – вместо того, чтобы сразу совпасть до конца испытуемого текста, а затем выполнять возвраты, пока не будет встречено последнее вхождение -->. (Подробнее о том, как выполняются возвраты с минимальными и максимальными квантификаторами, рассказывается в рецепте 2.13.) Эта стратегия прекрасно работает, потому что XML-подобные комментарии не могут вкладываться друг в друга. Другими словами, комментарии всегда оканчиваются самым первым (левым) вхождением -->.

Когда нельзя удалять комментарии

Большинству веб-разработчиков известно, что комментарии HTML внутри элементов <script> и <style> используются для обеспечения обратной совместимости с древними версиями броузеров. В настоящее время это стало бессмысленным шаманством, но оно продолжает жить, отчасти благодаря простому копированию фрагментов разметки при создании новых страниц. Предположим, что при удалении комментариев из документа (X)HTML вам совсем не хочется уничтожить программный

код JavaScript и таблицы CSS. Вероятно, вам также захочется оставить содержимое элементов <textarea>, разделов CDATA и значения атрибутов внутри.

Выше говорилось, что удалить комментарии совсем несложно. Как оказывается, эти слова справедливы, только если игнорировать некоторые необычные области разметки (X)HTML или XML, где изменяются синтаксические правила. Другими словами, если игнорировать сложные части проблемы, то все просто.

Конечно, в некоторых случаях можно было бы оценить разметку, которую требуется обработать, и решить, что эти проблемы вполне можно игнорировать, возможно, потому, что вы сами писали эту разметку и знаете, чего в ней можно ожидать. Упрощенный подход может использоваться при выполнении операции поиска с заменой в текстовом редакторе, позволяющем проверять каждое совпадение перед его удалением.

Но вернемся назад и посмотрим, как обойти эти проблемы. В разделе «Игнорирование необычных разделов (X)HTML и XML» на стр. 534 некоторые из этих проблем уже рассматривались в контексте сопоставления с XML-подобными тегами. Подобный подход можно использовать для поиска комментариев. Чтобы сначала отыскать необычные разделы, можно использовать программный код из рецепта 3.18 и регулярное выражение, которое приводится ниже, а затем замещать комментарии, найденные между совпадениями, пустыми строками (то есть удалить комментарии):

```
<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|'[^']*')*?↵
(?:/>|>.*?</\1\s*>)|<[a-z](?:[^>"']|"[^"]*"|'[^']*')*>|<!\[CDATA\[.*?]]>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Добавим несколько пробелов и комментариев в режиме свободного форматирования, чтобы сделать это регулярное выражение более простым для понимания:

```
# Специальный элемент: тег и содержимое
<( script | style | textarea | xmp )\b
    (?: [^>"']   # Совпадает с любыми именами атрибутов
      | “[^"]*”  #     ...и значениями
      | ‘[^']*’  #
    )*?
(?: # Одиночный тег
    />
  | # Иначе, включить в совпадение содержимое элемента и закрывающий тег
    > .*? </\1\s*>
)

|
```

```
# Обычный элемент: только тег
<[a-z]           # Первый символ имени тега
    (?: [^>"']   # Совпадает с остальной частью имени тега
      | “[^”]*”  #     ...вместе с именами атрибутов
      | ‘[^’]*’  #     ...и значениями
    )*
>

|

# Раздел CDATA
<!\[CDATA\[ .*? ]]>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Ниже приводится эквивалентная версия для диалекта JavaScript, в котором отсутствуют оба режима: «точке соответствуют границы строк» и «режим свободного форматирования»:

```
<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|'[^']*')*?↵
(?:/>|>[\s\S]*?</\1\s*>)|<[a-z](?:[^>"']|"[^"]*"|'[^']*')*>|<!\[CDATA\[↵
[\s\S]*?]]>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Варианты

Поиск допустимых XML-подобных комментариев

В действительности существует еще несколько синтаксических правил, касающихся комментариев в разметке (X)HTML и XML, помимо того, что комментарии начинаются с `<!--` и заканчиваются `-->`. В частности:

- Комментарии не могут содержать два дефиса, следующие подряд. Например, `<!-- com--ment -->` – это недопустимый комментарий, потому что в нем имеются два дефиса, следующие подряд.
- Закрывающей последовательности не может предшествовать дефис, являющийся частью комментария. Например, `<!-- comment --->` – это недопустимый комментарий, но полностью пустой комментарий `<!--->` допускается.
- Между `--` и `>` в закрывающей последовательности может присутствовать пробельный символ. Например, `<!-- comment -- >` – это полностью допустимый комментарий.

Эти правила легко оформить в виде регулярного выражения:

```
<!--[^-]*(?:-[^-]+)*--\s*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обратите внимание, что совпадение с содержимым комментария между открывающей и закрывающей последовательностями является необязательным, поэтому данное выражение будет совпадать с пустым комментарием `<!---->`. Однако если между ограничителями появится дефис, за ним должен следовать хотя бы один символ, отличный от дефиса. А так как внутри комментария не может присутствовать два дефиса, идущих подряд, минимальный квантификатор в регулярном выражении в начале рецепта был заменен максимальными квантификаторами. Минимальные квантификаторы тоже будут работать, но их использование здесь приведет к ненужным возвратам (рецепт 2.13).

Некоторые читатели могут, посмотрев на это новое регулярное выражение, спросить, почему здесь дважды используется инвертированный символьный класс `‹[^-]›` вместо того, чтобы просто поместить дефис в несохраняющую, необязательную группу (то есть, `‹<!--(?:-?[^-]+)*--\s*>›`). На то есть отличная причина, которая возвращает нас к обсуждению «катастрофических возвратов» из рецепта 2.15.

Так называемые *вложенные квантификаторы* всегда требуют особого внимания, чтобы избежать ситуации катастрофических возвратов. Квантификатор считается вложенным, когда он присутствует внутри группы, которая сама повторяется квантификатором. Например, шаблон `‹(?:-?[^-]+)*›` содержит два вложенных квантификатора: знак вопроса, следующий за дефисом, и знак плюс, следующий за инвертированным символьным классом.

Однако в действительности вложение квантификаторов само по себе не несет угрозы производительности. Опасность, скорее, заключается в потенциально огромном числе способов, которыми внешний квантификатор `‹*›` может комбинироваться с внутренними квантификаторами в процессе сопоставления с текстом. Если механизму регулярных выражений не удается отыскать `-->` в конце частичного совпадения (как это требуется, если включить данный сегмент шаблона в регулярное выражение, совпадающее с комментариями), он должен опробовать все возможные комбинации повторений, прежде чем убедиться в безуспешности попытки и перейти к следующей позиции. Число возможных вариантов увеличивается чрезвычайно быстро с каждым дополнительным символом, который механизм должен попробовать сопоставить. Однако во вложенных квантификаторах нет ничего опасного, если подобная ситуация невозможна. Например, шаблон `‹(?:-[^-]+)*›` не представляет угрозы, несмотря на то, что содержит вложенный квантификатор `‹+›`, потому что теперь сопоставление с дефисом должно производиться точно один раз на каждом повторении группы и потенциальное число точек возврата увеличивается линейно с увеличением длины испытуемого текста.

Еще один способ избежать проблемы возвратов, описанной только что, заключается в использовании атомарной группировки. Следующее выражение эквивалентно первому в этом разделе, но оно на несколько символов короче и не поддерживается диалектами JavaScript и Python:

```
<!--(?>-?[^-]+)*--\s*>
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

Подробное описание атомарной группировки (и близких к ней захватывающих квантификаторов) приводится в рецепте 2.14.

Поиск С-подобных комментариев

Подобного рода шаблон может использоваться для поиска других, не вложенных многострочных комментариев. Комментарии в языке С либо начинаются символами /* и заканчиваются первым вхождением */, либо начинаются символами // и продолжаются до конца строки. Следующее регулярное выражение обеспечивает совпадение с комментариями обоих типов за счет объединения шаблонов для каждой разновидности комментариев с помощью оператора выбора:

```
/\*[\s\S]*?\*/|//.*
```

Параметры: нет (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Рецепт 8.9, где показано, как отыскивать определенные слова, встречающиеся внутри XML-подобных комментариев.

8.9. Поиск слов в XML-подобных комментариях

Задача

Требуется отыскать все вхождения слова TODO (что сделать) внутри комментариев (X)HTML или XML. Например, требуется отыскать подчеркнутый текст в следующей строке:

```
This "TODO" is not within a comment, but the next one is. <!-- TODO: ↵
Come up with a cooler comment for this example. -->
```

Решение

Существует, по меньшей мере, два подхода к решению этой проблемы, и каждый из них имеет свои преимущества. Первый, который описы-

вается в разделе «Двухступенчатый подход» на стр. 41, состоит в том, чтобы отыскать комментарии с помощью внешнего регулярного выражения, а затем внутри каждого совпадения выполнить поиск с помощью другого регулярного выражения или даже с применением инструментов простого поиска текста. Этот подход дает лучшие результаты, если программный код, выполняющий эту работу, пишете вы сами, потому что деление решения задачи на два этапа упростит реализацию и сделает ее быстрой. Однако если поиск в файлах выполняется с помощью текстового редактора или инструмента grep, подход с делением задачи на два этапа работать не будет, если только используемый инструмент не предлагает специальную возможность поиска внутри совпадений с другим регулярным выражением[1].

Если необходимо выполнить поиск слов внутри комментариев с помощью единственного регулярного выражения, эту задачу можно решить с помощью проверки соседних символов. Этот подход демонстрируется в разделе «Одноступенчатый подход».

Двухступенчатый подход

Когда это возможно, лучше разбить решение этой задачи на два этапа: найти комментарии и затем найти в них слово TODO.

Так можно найти комментарии:

```
<!--.*?-->
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Диалект JavaScript не поддерживает режим «точке соответствуют границы строк», но в нем вместо точки можно использовать символьный класс, соответствующий любому символу, как показано ниже:

```
<!--[\s\S]*?-->
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Затем в каждом из комментариев, найденном с помощью только что представленных регулярных выражений, можно поискать текст, совпадающий с литералами ‹TODO›. При желании это регулярное выражение можно использовать в режиме нечувствительности к регистру символов и с границами слова по обеим сторонам, чтобы обеспечить совпадение только с полным словом TODO, например:

```
\bTODO\b
```

[1] PowerGREP, который описывается в разделе «Инструменты для работы с регулярными выражениями» в главе 1, является одним из таких инструментов, способных выполнять поиск внутри совпадений.

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецепте 3.13 показано, как реализовать поиск в совпадениях с внешним регулярным выражением

Одноступенчатый подход

Опережающая проверка (описывается в рецепте 2.16) позволяет решить эту задачу с помощью единственного регулярного выражения, хотя и менее эффективно. В следующем регулярном выражении позитивная опережающая проверка используется, чтобы убедиться, что за словом TODO находится последовательность -->, закрывающая комментарий. Сама она не может сказать, находится ли слово внутри комментария или просто за ним следует комментарий, поэтому используется вложенная негативная опережающая проверка, чтобы убедиться в отсутствии открывающей последовательности <!-- перед -->:

```
\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Так как диалект JavaScript не поддерживает режим «точке соответствуют границы строк», вместо точки можно использовать ‹[\s\S]›:

```
\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Двухступенчатый подход

В рецепте 3.13 приводится программный код, реализующий поиск внутри совпадений с другим регулярным выражением. Он принимает внутреннее и внешнее регулярное выражение. Регулярное выражение, совпадающее с комментарием, используется как внешнее выражение, а ‹\bTODO\b› – как внутреннее.

Главное, что следует отметить, это минимальный квантификатор ‹*?›, который следует за символом точки или за символьным классом в регулярном выражении, совпадающем с комментариями. Как описывается в рецепте 2.13, этот квантификатор обеспечивает совпадение до первого вхождения последовательности --> (завершающей комментарий), а не до самого последнего.

Одноступенчатый подход

Это решение более сложное и медленное. Его преимуществом является то, что оно объединяет оба этапа из предыдущего подхода в одно регулярное выражение. Благодаря этому данное решение может использоваться в текстовых редакторах, в средах разработки или с другими инструментами, не позволяющими производить поиск внутри совпадений с другим регулярным выражением

Разобьем регулярное выражение на составляющие в режиме свободного форматирования и поближе рассмотрим каждую его часть:

```
\b TODO \b      # Совпадение с символами "TODO", как с полным словом
(?=             # Проверить, что выражение ниже может совпасть здесь
  (?:           #   Группировка, но несохраняющая...
    (?! <!-- ) #     Проверить, что "<!--" не совпадет здесь
    .           #     Совпадение с любым единственным символом
  )*?           #   Ноль или более раз, минимально возможное число раз
  -->           #   Совпадение с символами "-->"
) #
```

Параметры: точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Эта прокомментированная версия регулярного выражения не будет работать в JavaScript из-за отсутствия в этом диалекте обоих режимов, «свободного форматирования» и «точке соответствуют границы строк».

Следует отметить, что регулярное выражение содержит негативную опережающую проверку, вложенную внутрь позитивной опережающей проверки. Это позволяет потребовать, чтобы любое совпадение с TODO предшествовало последовательности --> и между ними отсутствовало совпадение с <!--.

Если вам понятно, как все это работает, замечательно: вы можете пропустить оставшуюся часть данного раздела. Но если что-то кажется вам непонятным, давайте вернемся немного назад и шаг за шагом построим внешнюю позитивную опережающую проверку в этом регулярном выражении.

Допустим на минуту, что нам требуется просто отыскать вхождения слова TODO, за которыми с некоторой позиции в строке следует последовательность символов -->. В результате мы получим выражение ‹\bTODO \b(?=.*?-->)› (в режиме «точке соответствуют границы строк»), которое совпадет с подчеркнутым фрагментом в строке <!--TODO-->. Конструкция ‹.*?› в начале опережающей проверки совершенно необходима, потому что в противном случае регулярное выражение будет совпадать, только когда за словом TODO *сразу же* будет следовать -->, без каких-либо символов между ними. Квантификатор ‹*?› повторяет точку ноль или более

раз, причем, минимально возможное число раз, что просто замечательно, так как нам требуется совпадение только до первого вхождения `-->`.

К настоящему моменту регулярное выражение может быть записано как ‹`\bTODO(?=.*?-->)\b`›, где второй метасимвол ‹`\b`› перенесен за опережающую проверку, без какого-либо влияния на текст совпадения. Это обусловлено тем, что и граница слова, и опережающая проверка являются проверками с нулевой длиной совпадения (раздел «Проверка соседних символов» на стр. 111). Однако лучше поставить границу слова первой, для большей удобочитаемости и производительности. В середине частичного совпадения механизм регулярных выражений быстрее проверит наличие границы слова, потерпит неудачу и двинется дальше, пытаясь начать сопоставление со следующего символа в строке, не тратя время на опережающую проверку, в которой нет необходимости, если слово `TODO` не является самостоятельным словом.

Итак, похоже, что регулярное выражение ‹`\bTODO\b(?=.*?-->)`› работает замечательно, но что, если применить его к испытуемой строке `TODO <!-- separate comment -->`? Выражение точно так же обнаружит совпадение со словом `TODO`, потому что за ним следует строка `-->`, невзирая на то, что оно находится за пределами комментария. То есть необходимо заменить символ точки внутри опережающей проверки, совпадающий с любым символом, на конструкцию, совпадающую с любым символом, не являющимся частью строки `<!--`, которая открывает новый комментарий. Мы не можем использовать инвертированный символьный класс, такой как ‹`[^<!-]`›, потому что требуется разрешить вхождение символов `<`, `!` и `-`, если они не объединены в точную последовательность `<!--`.

Это как раз та ситуация, когда на помощь может прийти негативная опережающая проверка. Конструкции ‹`(?!<!--).`› соответствует любой одиночный символ, не являющийся частью последовательности, открывающей комментарий. Поместив этот шаблон внутрь несохраняющей группировки, как ‹`(?:(?!<!--).)`›, можно повторять всю последовательность с помощью минимального квантификатора ‹`*?`›, который ранее применялся к точке.

Объединив все вместе, получаем окончательное регулярное выражение, которое было приведено, как решение этой задачи: ‹`\bTODO\b(?=(?:(?!<!--).)*?-->)`›. В JavaScript, где отсутствует режим «точке соответствуют границы строк», эквивалентное выражение имеет вид: ‹`\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)`›.

Варианты

Хотя регулярное выражение из «Одноступенчатого подхода» и проверяет, что за любым совпадением со словом `TODO` следует строка `-->` без `<!--` между ними, но оно не проверяет обратное условие: что искомому слову предшествует строка `<!--` без `-->` между ними. Есть несколько причин, почему мы пропустили это правило:

- Обычно можно отказаться от выполнения двойной проверки, тем более, что одноступенчатое регулярное выражение предназначено для использования в текстовых редакторах и с другими инструментами, где можно визуально проверить результаты поиска.
- Чем меньше количество проверок, тем меньше на их выполнение тратится времени (то есть достигается выигрыш за счет отказа от дополнительных проверок).
- Самое главное: так как заранее не известно, как далеко перед словом TODO находится начало комментария, для заглядывания назад может потребоваться использовать бесконечную ретроспективную проверку, которая поддерживается только диалектом .NET.

При использовании диалекта .NET и при наличии желания включить эту дополнительную проверку можно задействовать следующее регулярное выражение:

```
(?<=<!--(?:(?!-->).)*?)\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк

Диалекты: .NET

Это более строгое регулярное выражение, поддерживаемое только диалектом.NET, содержит в самом начале дополнительную позитивную ретроспективную проверку, которая действует точно так же, как опережающая проверка в конце, но в обратном порядке. Так как ретроспективная проверка заглядывает назад, отыскивая <!--, она содержит вложенную опережающую проверку, которая позволяет ей совпадать с любыми символами, не являющимися частью последовательности -->.

Так как опережающая и ретроспективная проверки в начале выражения являются проверками с нулевой длиной совпадения, окончательное совпадение будет содержать только слово TODO. Совпадения с проверками не включаются в текст окончательного совпадения.

См. также

Рецепт 8.8, где подробно обсуждается, как обеспечить совпадение с XML-подобными комментариями.

8.10. Изменение разделителя, используемого в файлах CSV

Задача

Требуется заменить все запятые, разделяющие поля в файле CSV, на символы табуляции. Запятые, присутствующие внутри полей, окруженных кавычками, должны остаться без изменений.

Решение

Следующее ниже регулярное выражение совпадает с отдельным полем CSV и с предшествующим ему символом-разделителем, если таковой имеется. Обычно предшествующий разделитель является запятой, но он может быть и пустой строкой (то есть ничем) при совпадении с первым полем в первой записи или разрывом строки при совпадении с первым полем любой последующей записи. Каждый раз, когда обнаруживается совпадение, само поле, включая возможные окружающие его кавычки, сохраняется в обратной ссылке 2, а предшествующий ему разделитель – в обратной ссылке 1.

Регулярные выражения в этом рецепте предназначены для работы только с допустимыми файлами CSV, с точки зрения правил, которые приводились ранее в описании «Значения, разделенные запятыми (Comma-Separated Values, CSV)».

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится это же регулярное выражение в режиме свободного форматирования::

```
( , | \r?\n | ^ )  # Сохраняющая группа 1 совпадает с разделителями полей
                   #   или с началом строки
(                  # Сохраняющая группа 2 совпадает с единственным полем:
  [^",\r\n]+       #   не заключенным в кавычки
|                  #  или...
  " (?:[^"]|"")* " #   заключенным в кавычки (может содержать
                   #   экранированные кавычки)
)?                 # Группа является необязательной, так как
                   # поле может быть пустым
```

Параметры: режим свободного форматирования, (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Используя это регулярное выражение и программный код из рецепта 3.11, можно реализовать итерации по содержимому файла CSV и проверять значение обратной ссылки 1 после каждого совпадения. Значение строки с замещающим текстом для каждого совпадения зависит от значения этой обратной ссылки. Если это запятая – она замещается символом табуляции. Если обратная ссылка содержит пустую строку или разрыв строки, это значение остается на месте (то есть не делается ничего или полученное значение вставляется обратно, как часть строки замещающего текста). Так как поля CSV сохраняются в обратной ссылке 2 как часть общего совпадения, их также необходимо встав-

лять в каждую строку замещающего текста. Единственное, что действительно необходимо замещать, – это запятые, сохраняемые в обратной ссылке 1.

Пример на языке JavaScript

Следующий ниже фрагмент представляет законченную веб-страницу, включающую два многострочных поля ввода и кнопку с надписью Replace (Заменить) между ними. После щелчка на кнопке из первого текстового поля (помеченного как Input (Вход)) извлекается введенная в него строка, а затем с помощью только что продемонстрированного регулярного выражения все запятые-разделители преобразуются в символы табуляции и получившаяся строка помещается во второе текстовое поле (помеченное как Output (Выход)). Если в первое поле было введено допустимое содержимое в формате CSV, оно должно появиться во втором текстовом поле с запятыми, замещенными символами табуляции. Чтобы выполнить испытания, сохраните этот фрагмент кода в файле с расширением «.html» и откройте его в веб-броузере:

```
<html>
<head>
<title>Change CSV delimiters from commas to tabs</title>
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Replace" onclick="commas_to_tabs()"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function commas_to_tabs () {
    var input = document.getElementById('input'),
        output = document.getElementById('output'),
        regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
        result = '',
        match;

    while (match = regex.exec(input.value)) {
        // Проверить значение обратной ссылки 1
        if (match[1] == ',') {
            // Добавить в результат символ табуляции (на место
            // совпавшей запятой) и содержимое обратной ссылки 2.
            // Если обратная ссылка 2 не определена (когда из-за
            // необязательности вторая сохраняющая группа не участвовала
            // в сопоставлении), использовать пустую строку.
            result += '\t' + (match[2] || '');
```

```
            } else {
                // Добавить все совпадение в результат
                result += match[0];
            }
            // Предотвратить попадание некоторых броузеров в бесконечный цикл
            if (match.index == regex.lastIndex) regex.lastIndex++;
        }
        output.value = result;
    }
    </script>
    </body>
    </html>
```

Обсуждение

Подход, описанный в этом рецепте, позволяет передавать каждое отдельное поле CSV (включая внутренние разрывы строк, экранированные кавычки и запятые) по одному за раз. Благодаря этому каждое последующее сопоставление будет начинаться с позиции непосредственно перед следующим разделителем полей.

Первая сохраняющая группа в регулярном выражении, `‹(,|\r?\n|^)›`, совпадает с запятой, разрывом строки или с позицией в начале испытуемого текста. Так как механизм регулярных выражений опробует варианты выбора слева направо, эти варианты перечислены в порядке уменьшения вероятности встречи с ними в файле CSV. Эта сохраняющая группа – единственная часть регулярного выражения, требующая обязательного совпадения. То есть возможна ситуация, когда общее совпадение с регулярным выражением будет пустой строкой, поскольку якорь `‹^›` всегда совпадает один раз. Значение, совпавшее с этой первой сохраняющей группой, должно быть проверено в программном коде за пределами регулярного выражения, который замещает запятые требуемым разделителем (например, символом табуляции).

Мы еще не прошли через всё регулярное выражение, но подход, описанный до сих пор, уже начинает казаться замысловатым. Может возникнуть вопрос – почему бы не написать регулярное выражение, совпадающее только с запятыми, которые необходимо заменить символами табуляции. Если бы было сделано именно так, простая замена всех совпадений помогла бы избавиться от программного кода за пределами регулярного выражения, проверяющего, совпадает ли сохраняющая группа 1 с запятой или какими-либо другим текстом. В конце концов, определить, находится ли запятая внутри или за пределами поля CSV в кавычках, можно с помощью опережающей и ретроспективной проверки, разве не так?

К сожалению, для реализации такого регулярного выражения потребовалась бы ретроспективная проверка бесконечной длины, чтобы точно определить, какие запятые находятся за пределами полей в ка-

вычках, что возможно только в диалекте .NET (описание различных ограничений ретроспективной проверки можно найти в разделе «Различные уровни ретроспективной проверки» на стр. 113). Однако даже разработчикам для платформы .NET следует избегать решений, основанных на проверках соседних символов, потому что это может приводить к существенному усложнению и снижению производительности регулярного выражения.

Возвращаясь назад к описанию принципа действия регулярного выражения, заметим, что большая часть его заключена в следующую пару круглых скобок – сохраняющую группу 2. Эта вторая группа совпадает с единственным полем CSV, включая окружающие его кавычки. В отличие от предыдущей группы, эта является необязательной и может совпадать с пустыми полями.

Обратите внимание, что группа 2 содержит два альтернативных подвыражения, разделенных метасимволом ‹|›. Первая альтернатива, ‹[^",\r\n]+›, – это инвертированный символьный класс, за которым следует квантификатор «один или более раз» (‹+›), а все вместе это соответствует полю, не заключенному в кавычки. Чтобы соответствовать этому подвыражению, поле не должно содержать кавычки, запятые или разрывы строк.

Вторая альтернатива внутри группы 2, ‹"(?:[^"]|"")*"›, соответствует полю, заключенному в кавычки. Точнее, она соответствует символу кавычки, за которой следует ноль или более символов, не являющихся кавычкой, и повторяющихся (экранированных) кавычек, за которыми следует закрывающая кавычка.

Квантификатор ‹*› в конце внутренней несохраняющей группы повторяет сопоставление двух внутренних вариантов выбора максимально возможное число раз, пока не будет найдена кавычка, не дублированная и потому завершающая поле.

Предположим, что выражение применяется к допустимому файлу CSV. Тогда первое совпадение, обнаруженное регулярным выражением, будет найдено в самом начале испытуемого текста, а каждое последующее должно начинаться непосредственно за концом предыдущего совпадения.

См. также

Рецепт 8.11, где описывается, как использовать регулярное выражение из этого рецепта для извлечения полей CSV, принадлежащих определенному столбцу.

8.11. Извлечение полей CSV из определенного столбца

Задача

Необходимо извлечь все поля из третьего столбца файла CSV.

Решение

Регулярное выражение из рецепта 8.10 можно использовать для обхода всех полей в испытуемом тексте CSV. Добавив немного программного кода, можно подсчитывать поля в каждой строке (или *записи*) и извлекать поля только из определенной позиции.

Следующее регулярное выражение (показано в обычном режиме и в режиме свободного форматирования) совпадает с единственным полем CSV и предшествующим ему разделителем, сохраняя их в двух отдельных сохраняющих группах. Так как внутри полей в кавычках могут появляться разрывы строк, простой поиск от начала до конца в каждой строке CSV может не давать точных результатов. Сопоставляя и перешагивая через поля по одному, легко определить, какие разрывы строк находятся за пределами полей в кавычках и благодаря этому начинать отсчет полей в новой записи.

Регулярные выражения в этом рецепте предназначены для работы только с допустимыми файлами CSV, с точки зрения правил, которые приводились в описании формата «Значения, разделенные запятыми (Comma-Separated Values, CSV)» на стр. 519.

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
( , | \r?\n | ^ )  # Сохраняющая группа 1 совпадает с разделителями полей
                   #   или с началом строки
(                  # Сохраняющая группа 2 совпадает с единственным полем:
  [^",\r\n]+       #   не заключенным в кавычки
|                  # или...
  " (?:[^"]|"")* " #   заключенным в кавычки (может содержать
                   #    экранированные кавычки)
)?                 # Группа является необязательной, так как
                   # поле может быть пустым
```

Параметры: режим свободного форматирования, (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это в точности те же самые регулярные выражения, которые приводились в рецепте 8.10, и они могут использоваться для решения широкого круга других задач обработки файлов CSV. Следующий фрагмент программного кода демонстрирует, как можно использовать версию выражения не в режиме свободного форматирования для извлечения столбца CSV.

Пример на языке JavaScript

Следующий ниже фрагмент представляет законченную веб-страницу, включающую два многострочных поля ввода и кнопку с надписью Extract Column 3 (Извлечь столбец 3) между ними. После щелчка на кнопке из первого текстового поля Input (Вход) извлекается введенная в него строка, а затем с помощью регулярного выражения, продемонстрированного выше, из каждой записи извлекается значение третьего поля, после чего весь столбец (где значения полей отделяются разрывами строк) помещается в поле Output (Выход). Чтобы выполнить испытания, сохраните этот фрагмент кода в файле с расширением «.html» и откройте его в веб-броузере:

```
<html>
<head>
<title>Extract the third column from a CSV string</title>
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Extract Column 3"
    onclick="display_csv_column(2)"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function display_csv_column (index) {
    var input = document.getElementById('input'),
        output = document.getElementById('output'),
        column_fields = get_csv_column(input.value, index);

    if (column_fields.length > 0) {
        // Вывести каждую запись в отдельной строке, отделив ее от других
        // записей символом перевода строки (\n)
        output.value = column_fields.join('\n');
    } else {
        output.value = '[No data found to extract]';
    }
}
```

```
// Вернуть массив полей из переданного текста CSV,
// индексация элементов в котором начинается с нуля
function get_csv_column (csv, index) {
    var regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
        result = [],
        column_index = 0,
        match;

    while (match = regex.exec(csv)) {
        // Проверить значение обратной ссылки 1. Если это запятая,
        // увеличить значение column_index. Иначе сбросить его в ноль.
        if (match[1] == ',') {
            column_index++;
        } else {
            column_index = 0;
        }
        if (column_index == index) {
            // Добавить поле (обр. ссылка 2) в конец массива результата
            result.push(match[2]);
        }
        // Предотвратить попадание некоторых броузеров в бесконечный цикл
        if (match.index == regex.lastIndex) regex.lastIndex++;
    }
    return result;
}
</script>
</body>
</html>
```

Обсуждение

Так как здесь повторно используется регулярное выражение из рецепта 8.10, мы не будем опять подробно описывать, как оно действует. Однако этот рецепт включает новый пример программного кода на JavaScript, где данное регулярное выражение используется для извлечения полей с определенным порядковым номером из каждой записи CSV в испытуемом тексте.

Функция `get_csv_column()` в представленном коде выполняет итерации через совпадения в испытуемом тексте. После каждого совпадения проверяется содержимое обратной ссылки 1. Если она содержит запятую, следовательно, было найдено совпадение не с первым полем в записи, поэтому производится наращивание переменной `column_index`, где хранится порядковый номер текущего столбца. Если обратная ссылка 1 содержит нечто отличное от запятой (то есть пустую строку или разрыв строки), следовательно, было найдено совпадение с первым полем в новой строке, поэтому значение переменной `column_index` сбрасывается в ноль.

Затем программный код проверяет, не достиг ли счетчик `column_index` индекса искомого столбца, который требуется извлечь. Каждый раз, когда проверка дает положительный результат, в массив с результатами добавляется значение обратной ссылки 2 (все, что следует за разделителем полей). После того как будет выполнен обход всего испытуемого текста, функция `get_csv_column()` вернет массив, содержащий значения всего заданного столбца (третьего, в этом примере). Список совпадений записывается во второе текстовое поле на странице, при этом каждое значение отделяется символом перевода строки (`\n`).

Этот пример можно было бы немного улучшить, дав пользователю возможность указывать номер столбца, который следует извлечь, с помощью дополнительного поля ввода или посредством запроса. Обсуждаемая функция `get_csv_column()`написана в предположении такой возможности и позволяет указывать индекс желаемого столбца, отсчет которых начинается с нуля, в виде второго аргумента (`index`).

Варианты

Хотя использование программного кода для обхода текста по одному полю CSV за раз обеспечивает дополнительную гибкость, при использовании текстового редактора может оказаться вполне достаточным ограничиться операцией поиска с заменой. В этой ситуации можно добиться похожих результатов, производя замещение совпадения с каждой полной записью значением поля с требуемым индексом (с помощью обратной ссылки). Следующие регулярные выражения иллюстрируют этот прием для различных индексов столбцов, замещая каждую запись полем из заданного столбца.

Все эти регулярные выражения не будут совпадать с записями, число полей в которых меньше номера искомого столбца, поэтому такие записи останутся в неизменном виде.

Соответствует записи CSV и сохраняет поле из столбца с номером 1 в обратной ссылке 1

```
^([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует записи CSV и сохраняет поле из столбца с номером 2 в обратной ссылке 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?,([^",\r\n]+|"(?:[^"]|"")*")?↵
(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует записи CSV и сохраняет поле из столбца с номером 3 или выше в обратной ссылке 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?){1},↵
([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Увеличив число в квантификаторе ‹{1}›, можно заставить регулярное выражение обрабатывать поля с порядковыми номерами выше 3. Например, если изменить значение квантификатора на ‹{2}›, будут сохраняться поля из столбца с номером 4, если изменить на ‹{3}› – будут сохраняться поля из столбца с номером 5 и так далее. Если предполагается работать с полями из столбца 3, квантификатор ‹{1}› можно просто удалить, так как в этом случае он не оказывает никакого воздействия.

Замещающий текст

Со всеми этими регулярными выражениями используется один и тот же замещающий текст (обратная ссылка 1). В результате замещения каждого совпадения содержимым обратной ссылки 1 должны остаться только искомые поля.

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Диалекты замещающего текста: Python, Ruby

8.12. Сопоставление с заголовком раздела в файле INI

Задача

Необходимо отыскать совпадения со всеми заголовками разделов в файле INI.

Решение

Здесь все просто. Заголовки разделов в файлах INI находятся в начале строки текста и окружены квадратными скобками (например, [Section1]). Эти правила легко оформить в виде регулярного выражения:

```
^\[[^\]\r\n]+]
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В этом регулярном выражении не так много частей, поэтому его легко будет разобрать:

- Благодаря включенному режиму «символам ^ и $ соответствуют границы строк» начальный символ ‹^› совпадает с началом каждой строки в тексте.
- ‹\[› совпадает с литералом [. Чтобы символ [не интерпретировался как начало символьного класса, он экранируется обратным слэшем.
- ‹[^\]\r\n]› – это инвертированный символьный класс, которому соответствует любой символ, кроме], возврата каретки (\r) и перевода строки (\n). Вслед за классом сразу же следует квантификатор ‹+›, позволяющий классу совпадать с одним или более символами, пока не будет обнаружено совпадение с....
- Завершающему символу ‹]› соответствует литерал] в конце заголовка раздела. Этот символ не требуется экранировать, так как он не находится внутри символьного класса.

Если требуется просто отыскать определенный заголовок раздела, то решение выглядит еще проще. Следующее регулярное выражение совпадает с заголовком раздела Section1:

```
^\[Section1]
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Единственное отличие от сопоставления с простым текстом «[Section1]» состоит в том, что совпадение должно находиться в начале строки текста. Это предотвращает возможность совпадения с закомментированным заголовком раздела (которому предшествует символ точки с запятой) или с фрагментом, который выглядит как заголовок, но фактически является частью значения параметра (например, Item1=[Value1]).

См. также

Рецепт 8.13, где описывается, как обеспечивать совпадение с разделами в файлах INI.

Рецепт 8.14, где описывается, как добиться того же самого в отношении пар имя-значение.

8.13. Сопоставление с разделом в файле INI

Задача

Требуется обеспечить совпадение со всем разделом в файле INI (то есть с заголовком раздела и со всеми парами имя-значение в этом разделе),

чтобы разбить файл INI на блоки или для обработки блоков по отдельности.

Решение

В рецепте 8.12 показано, как обеспечить совпадение с заголовком раздела в файле INI. Чтобы обеспечить совпадение со всем разделом, мы начнем сопоставление точно так же, как в предыдущем рецепте, но будем продолжать сопоставление, пока не будет достигнут конец текста или символ [в начале строки (так как это указывает на начало нового раздела):

```
^\[[^\]\r\n]+](?:\r?\n(?:[^[\r\n].*)?)*
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Или в режиме свободного форматирования:

```
^ \[ [^\]\r\n]+ ] # Совпадение с заголовком раздела
(?:               # Далее следует остальная часть раздела...
  \r?\n           #   Совпадение с последовательностью символов разрыва
строки
  (?:             #   После каждого начала строки совпадает...
    [^[\r\n]      #     С любым символом, кроме “[“ и разрыва строки
    .*            #     Совпадение с остатком строки
  )?              #   Необязательная, чтобы совпадать с пустыми строками
)*                # Продолжать, пока не будет найден конец раздела
```

Параметры: символам ^ и $ соответствуют границы строк, режим свободного форматирования (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Это регулярное выражение начинается сопоставлением заголовка раздела в файле INI с шаблоном ‹^\[[^\]\r\n]+]› и продолжает построчное сопоставление, пока не будет встречена строка, начинающаяся с символа [. Рассмотрим следующий текст:

```
[Section1]
Item1=Value1
Item2=[Value2]

; [SectionA]
; Заголовок раздела SectionA был закомментирован

ItemA=ValueA ; ItemA не закомментирован и является частью раздела Section1
```

```
[Section2]
Item3=Value3
Item4 = Value4
```

В данном тексте это регулярное выражение обнаружит два совпадения. Первое будет простираться от начала текста до заголовка «[Section2]», включая пустую строку перед ним. Второе совпадение будет простираться от начала заголовка Section2 до конца текста.

См. также

Рецепт 8.12, где описывается, как обеспечить совпадение с заголовком раздела в файле INI.

Рецепт 8.14, где описывается, как добиться совпадения с парами имя-значение.

8.14. Сопоставление с парами имя–значение в файле INI

Задача

Требуется обеспечить совпадение с парами имя-значение в файле INI (например, `Item1=Value1`), разбивая каждое совпадение на две части с помощью сохраняющих групп. Обратная ссылка 1 должна хранить имя параметра (`Item1`), а обратная ссылка 2 – значение (`Value1`).

Решение

Ниже приводится регулярное выражение, выполняющее эту работу (далее следует то же самое регулярное выражение в режиме свободного форматирования):

```
^([^=;\r\n]+)=([^;\r\n]*)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^               # Начало строки текста
( [^=;\r\n]+ ) # Сохранить имя в обратной ссылке 1
=               # Разделитель между именем и значением
( [^;\r\n]* )  # Сохранить значение в обратной ссылке 2
```

Параметры: символам ^ и $ соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Как и в других рецептах этой главы, имеющих отношение к файлам INI, здесь мы работает с весьма простыми составляющими. Регулярное выражение начинается с символа ‹^›, который обеспечивает совпадение с началом строки в тексте (при этом должен быть включен режим «символам ^ и $ соответствуют границы строк»). Очень важно, чтобы совпадение начиналось с начала строки, потому что в противном случае можно было бы найти совпадение с частью закомментированной строки.

Затем регулярное выражение использует сохраняющую группу, содержащую инвертированный символьный класс (‹[^=;\r\n]›), за которым следует квантификатор «один или более раз» (‹+›), что обеспечивает совпадение с именем параметра и сохранение его в виде обратной ссылки 1. Инвертированному классу соответствует любой символ, кроме следующих четырех: знак равенства, точка с запятой, возврат каретки (‹\r›) и перевод строки (‹\n›). Символы возврата каретки и перевода строки используются как признак конца параметра в файле INI, точка с запятой отмечает начало комментария, а знак равенства отделяет имя параметра от значения.

После сопоставления с именем параметра далее в регулярном выражении следуют сопоставление с литералом знака равенства (разделитель имени и значения) и затем со значением параметра. Совпадение со значением обеспечивает вторая сохраняющая группа, в которой используется шаблон, похожий на шаблон имени параметра, но имеющий на два ограничения меньше. Во-первых, этот второй шаблон допускает совпадение со знаком равенства как частью значения (то есть инвертированный символьный класс на один символ короче). Во-вторых, здесь используется квантификатор ‹*›, чтобы устранить необходимость соответствовать хотя бы одному символу (так как, в отличие от имен, значения могут быть пустыми).

Вот и все.

См. также

Рецепт 8.12, где описывается, как обеспечить совпадение с заголовком раздела в файле INI.

Рецепт 8.13, где описывается, как обеспечивать совпадение с разделами в файлах INI.

Алфавитный указатель

Символы

7-битный набор символов, 53
$ (знак доллара)
  как метасимвол, 49
  как якорный метасимвол, 63, 65, 66, 147, 279, 283
  экранирование, 406
    в замещающем тексте, 125
$~[0], переменная, 188
$`, переменная, 134, 192, 253
$_, переменная, 134
$', переменная, 134, 253
$&, переменная, 129, 187, 192
$~, переменная (Ruby), 187
  сохраняющие группы, 200
$+, хеш, 201, 237
& (амперсанд)
  экранирование, 407
| (вертикальная черта)
  как оператор выбора, 85, 301, 365
  экранирование, 406
- (дефис)
  диапазоны внутри символьных классов, 55
  как метасимвол, 49
  удаление из номера кредитной карты, 345, 348
  экранирование, 406
, (запятая)
  как метасимвол, 57
  как разделитель групп разрядов, 433
  экранирование, 407
* (звездочка)
  как квантификатор, 279
  как максимальный квантификатор, 101
  как метасимвол, 49
  экранирование, 406
? (знак вопроса)
  для обозначения минимальных квантификаторов, 102
  как квантификатор, 99
  как метасимвол, 49
  экранирование, 406
+ (знак плюс)
  для обозначения захватывающего квантификатора, 104
  как квантификатор, 279
  как метасимвол, 49
  экранирование, 406
[] (квадратные скобки)
  для определения символьных классов, 55
  как метасимволы, 49
  символьные классы, 279
  экранирование, 406
@ (коммерческое at)
  представление в Perl, 147
  совпадение, 278
() (круглые скобки)
  атомарная группировка, 108, 318
  как метасимволы, 49
  повторный поиск соответствия с ранее совпавшим текстом, 91
  сохраняющая группировка, 87
  экранирование, 406
^ (крышка)
  как метасимвол, 49
  как якорный метасимвол, 63, 65, 279, 283
  отрицание внутри символьных классов, 55
  экранирование, 406
(?:), несохраняющая группа, 279
\\ (обратный слэш)
  в исходных текстах, 142
  внутри символьных классов, 55
  как метасимвол, 49
  как символ экранирования, 49
  экранирование, 125, 406
  экранирование символов, 403
!~, оператор (Perl), 174
=~, оператор (Perl), 174
=~, оператор (Ruby), 174, 193
(решетка)
  оформление комментариев, 123
  # (решетка)экранирование, 407
™ (символ торговой марки), сопоставление, 71

. (точка)
злоупотребление, 61
как метасимвол, 49
соответствует любому символу, 59
экранирование, 406
{} (фигурные скобки)
для организации повторений, 97
как метасимволы, 49
экранирование, 406

A

\\A, якорный метасимвол, 63, 64, 65, 280
\\a (управляющий символ ASCII bell), 52
ActionScript, язык программирования, 139
appendReplacement(), метод, 242
appendTail(), метод, 242

B

\\b, метасимвол, граница слова, 57, 68, 286, 302, 362
\\B, не граница слова, 69
<b>, тег, замещение тегом , 541
begin(), метод, Ruby (класс MatchData), 193

C

\\cA по \\cZ, значения, 53
CacheSize, свойство (.NET), 154
CANON_EQ, константа (Java), 164
CASE_INSENSITIVE, константа (Java), 161
CDATA, разделы XML, 518
cellspacing, атрибут, добавление в теги <table>, 566
CIL (Common Intermediate Language общий промежуточный язык), 158
Comma-Separated Values (значения, разделенные запятыми, CSV), 519
извлечение полей из определенного столбца, 584
изменение разделителя, 579
COMMENTS, константа (Java), 161
compile(), метод
Java (класс Pattern), 145, 155, 161
Python (модуль re), 157, 163
Ruby (класс Regexp), 158
Count, свойство (Match.Groups), 198
Currency, блок Юникода, 79
строка из определенного пространства имен (Namespace Specific String, NSS), 451
C, язык программирования, 139
C#, язык программирования, 136
импортирование библиотеки регулярных выражений, 150
литералы регулярных выражений, 144
создание объектов регулярных выражений, 154
C++, язык программирования, 139

D

\\D, метасимвол, 57
\\d, метасимвол, 57, 411
Delphi for Win32, язык программирования, 139
Delphi Prism, язык программирования, 140
DOCTYPE, объявление (HTML), 516
DOTALL, константа (Java), 161

E

\\E, лексема, для подавления метасимволов, 50
\\e (управляющий символ ASCII escape), 52
ECMA-262, стандарт, 137, 139
ECMAScript, значение (RegexOptions), 164
ECMAScript язык программирования, 137
end(), метод
Java (класс Matcher), 191, 198
Python (класс MatchObject), 193
Ruby (класс MatchData), 193
EPP (Extensible Provisioning Protocol расширяемый протокол представления информации), 290
ereg, семейство функций (PHP), 138
литералы регулярных выражений, 146
exec(), метод, JavaScript, 191, 198
обход всех совпадений в цикле, 212
ExplicitCapture, значение (RegexOptions), 164
EXTENDED, константа (Regexp), 164
Extensible Hypertext Markup Language (расширяемый язык разметки гипертекста, XHTML), 516
добавление атрибутов в теги <table>, 566

поиск тегов, 521
удаление всех тегов, за исключением выбранных, 545
Extensible Markup Language (расширяемый язык разметки, XML), 517
добавление атрибутов в теги <table>, 566
поиск определенных атрибутов, 560
поиск слов в комментариях, 574
поиск тегов, 521, 524
сопоставление с именами, 549
удаление всех тегов, за исключением выбранных, 545
удаление комментариев, 569

F

\\f (управляющий символ ASCII form feed), 52
findall(), метод, Python (модуль re), 207
finditer(), метод, Python (модуль re), 214
find(), метод, Java (класс Matcher), 173, 186, 191, 205
обход всех совпадений в цикле, 212

G

Get-Unique, команда (PowerShell), 389
grep, функция, поддержка в проекте R, 140
Groovy, язык программирования, 140
GroupCollection, класс (.NET), 197
groupdict(), метод, Python (класс MatchObject), 200
groups(), метод, Python (класс MatchObject), 199
Groups, свойство .NET (функция Match()), 197
Group, класс (.NET), 197
group(), метод
Java (класс Matcher), 198
re, модуль (Python), 199
gsub(), метод, Ruby (класс String), 232, 244
обратные ссылки, 235

H

Hypertext Markup Language (язык разметки гипертекста, HTML), 513
добавление атрибутов в теги <table>, 566
добавление тегов <p> в простой текст, 557
замещение тега <b> тегом , 541
поиск тегов, 521
проверка тегов, 537
удаление всех тегов, за исключением выбранных, 545

I

(?-i), модификатор режима, 51
(?i), модификатор режима, 51
IGNORECASE, константа (Regexp), 164
IgnoreCase, параметр (RegexOptions), 161
IgnorePatternWhitespace, параметр (RegexOptions), 161
IndexOutOfBoundsException, исключение, 229
index, свойство
JavaScript, 191
.NET, 190
INI, файлы
сопоставление с заголовком раздела, 588
сопоставление с парами имя-значение, 591
сопоставление с разделом, 589
IPv4, адреса, сопоставление, 476
IPv6, адреса, сопоставление, 479
IsMatch(), метод (.NET), 172
ISO 8601, форматы, 303

J

JavaScript, язык программирования, 137
библиотека регулярных выражений, 151
литералы регулярных выражений, 145
определение замещающего текста, 27
параметры регулярных выражений, 162, 165
поддержка регулярных выражений, 24, 137
создание объектов регулярных выражений, 156
java.util.regex, пакет, 136
для использования регулярных выражений, 151
Java, язык программирования, 136
импортирование библиотеки регулярных выражений, 151
литералы регулярных выражений, 145
определение замещающего текста, 27

параметры регулярных выражений, 161, 164
поддержка регулярных выражений, 23, 136
создание объектов регулярных выражений, 155

K

ksort(), функция (PHP), 231

L

lastIndex, свойство, JavaScript (класс RegExp), 192, 212, 378
length(), метод, Ruby (класс MatchData), 194
length, свойство
JavaScript, 191
.NET, 190

M

(?m), модификатор режима, 67
MatchData, класс (Ruby), 187
begin(), метод, 193
end(), метод, 193
length(), метод, 194
offset(), метод, 194
size(), метод, 194
Matcher, класс (Java), 156
end(), метод, 191, 198
find(), метод, 173, 186, 191, 205, 212
group(), метод, 198
reset(), метод, 156, 186
start(), метод, 191, 198
Matches(), метод (.NET), 204
MatchEvaluator, класс (.NET), 240
MatchObject, класс (Python)
end(), метод, 193
start(), метод, 193
Match, класс (.NET)
NextMatch(), метод, 211
свойства Index и Length, 190
match(), метод
Ruby (класс Regexp), 193
match(), метод (JavaScript), 205
Match(), метод (.NET), 190
Groups, свойство, 197
Value, свойство, 185
обход свех совпадений в цикле, 211
mb_ereg, семейство функций (PHP), 137
литералы регулярных выражений, 146
Microsoft VBScript, библиотека, 141
MULTILINE, константа
Java, 161
Regexp, 164
Multiline, параметр (RegexOptions), 161
m//, оператор (Perl), 174, 187
сохраняющие группы, 199

N

.NET, платформа, 136
параметры регулярных выражений, 161, 164
создание объектов регулярных выражений, 154
\\n (символ перевода строки)
представление в C#, 144
представление в Java, 145
представление в Python, 143, 148
\\n (управляющий символ ASCII newline), 52
NEAR-поиск, 379
new(), метод (Ruby), 158, 163
Nregex, веб-приложение для тестирования регулярных выражений, 34

O

offset(), метод, Ruby (класс MatchData), 194
Onigurama, библиотека, 138

P

\\P{}, для проверки категории Юникода, 76
\\p{}, определение категории Юникода, 72
PatternSyntaxException, исключение, 155
Pattern, класс (Java), 155
compile(), метод, 145, 155, 161
split(), метод, 259
константы параметров, 161, 164
PCRE (Perl-Compatible Regular Expressions – Perl-совместимые регулярные выражения), библиотека, 22, 137
определение замещающего текста, 26
Perl, язык программирования, 138
библиотека регулярных выражений, 151
литералы регулярных выражений, 146
определение замещающего текста, 26
параметры регулярных выражений, 163, 166

поддержка регулярных выражений, 21, 138
создание объектов регулярных выражений, 157
PHP, язык программирования, 137
импортирование библиотеки регулярных выражений, 151
литералы регулярных выражений, 146
определение замещающего текста, 26
параметры регулярных выражений, 162, 165
функции регулярных выражений, 137
POSIX-совместимые механизмы регулярных выражений, 425
PowerGREP, приложение, 43
PowerShell, язык сценариев, 140
PREG_OFFSET_CAPTURE, константа, 192, 199, 206
PREG_PATTERN_ORDER, константа, 205
PREG_SET_ORDER, константа, 205
PREG_SPLIT_DELIM_CAPTURE, константа, 267
PREG_SPLIT_NO_EMPTY, константа, 261, 267
preg, семейство функций (PHP), 137
preg_match_all(), функция, 205, 214
preg_match(), функция, 173, 187, 213
preg_replace_callback(), функция, 243
preg_replace(), функция, 137, 229, 235, 237
preg_split(), 267
preg_split(), функция, 261
доступность, 151
литералы регулярных выражений, 146
параметры регулярных выражений, 162
создание объектов регулярных выражений, 156
Python, язык программирования, 138
импортирование библиотеки регулярных выражений, 151
литералы регулярных выражений, 148
определение замещающего текста, 27
параметры регулярных выражений, 163, 166
поддержка регулярных выражений, 24
создание объектов регулярных выражений, 157
функции регулярных выражений, 138

Q

\\Q, лексема, для подавления метасимволов, 50
qr//, оператор (Perl), 157

R

\\r (управляющий символ ASCII carriage return), 52
REALbasic, язык программирования, 141
RegexOptions, перечисление, 161, 164
RegexOptions, перечисление Compiled, значение, 158
regexpr, функция, поддержка в проекте R, 140
RegExp, класс (JavaScript)
exec(), метод, 198, 212
index, свойство, 191
lastIndex, свойство, 192, 212, 378
length, свойство, 191
test(), метод, 173
Regexp, класс (Ruby)
compile(), метод, 158
new(), метод, 158, 163
RegExp(), конструктор (JavaScript), 156
RegexRenamer, приложение, 46
Regex, класс (.NET)
IsMatch(), метод, 172
Matches(), метод, 204
Replace(), метод, 227, 234, 240, 241
Split(), метод, 257, 266
RegEx, класс (REALbasic), 141
Regex(), конструктор (.NET), 144, 154
RegexOptions, перечисление (параметры регулярных выражений), 158
replaceAll(), метод, Java (класс String), 229
обратные ссылки, 234
replaceFirst(), метод, Java (класс String), 229
обратные ссылки, 234
replace(), метод, JavaScript (класс String), 229
обратные ссылки, 235
Replace(), метод (.NET), 227, 240, 241
обратные ссылки, 234
replace, функция (класс String)
удаление пробельных символов, 400

reset(), метод, Java (класс Matcher), 156, 186
re, модуль (Python), 27, 138, 148
compile(), функция, 157
findall(), метод, 207
finditer(), метод, 214
group(), метод, 199
search(), функция, 174, 187
split(), метод, 262, 268
sub(), метод, 231, 235, 244
импортирование, 151
параметры регулярных выражений, 163
RFC 3986, стандарт, 455
RFC 4180, стандарт, 519
Ruby, язык программирования, 138
библиотека регулярных выражений, 151
литералы регулярных выражений, 149
определение замещающего текста, 27
параметры регулярных выражений, 163, 166
поддержка регулярных выражений, 24
создание объектов регулярных выражений, 158
функции регулярных выражений, 138
R, проект, 140

S

<script>, элементы (HTML), 514
<style>, элементы (HTML), 514
\\S, метасимвол, 278
\\s, метасимвол, 315
\\s, пробельный символ, 57
scala.util.matching, пакет, 141
Scala, язык программирования, 141
scan(), метод, Ruby (класс String), 214
search(), функция, Python (модуль re), 174, 187
Singleline, параметр (RegexOptions), 161
size(), метод, Ruby (класс MatchData), 194
split(), метод,
JavaScript (класс String), 260
Java (класс String), 259
.NET, 257
обратные ссылки, 266
Python (модуль re), 262
и сохраняющая группировка, 268
Ruby (класс String), 263
и сохраняющая группировка, 268
Perl, 261
и сохраняющая группировка, 267
start(), метод,
Java (класс Matcher), 191, 198
Python (класс MatchObject), 193
String, класс (Java)
replaceAll(), метод, 229, 234
replaceFirst(), метод, 229, 234
String, класс (JavaScript)
match(), метод, 186, 205
replace(), метод, 229, 235
replace, функция, класс String, 400
split(), метод, 260
String, класс (Ruby)
gsub(), метод, 232, 235, 244
scan(), метод, 214
split(), метод, 263, 268
strlen(), функция, 192
sub(), метод, Python (модуль re), 27, 231, 244
обратные ссылки, 235
sub, функция, поддержка в проекте R, 140
System.Text.RegularExpressions.Regex, класс, 154
s///, оператор (Perl), 231
/e, модификатор, 244
обратные ссылки, 235

T

<table>, тег, добавление атрибутов, 566
\\t (управляющий символ ASCII horizontal tab), 52
test(), метод (JavaScript), 173
TPerlRegEx, компонент, 140

U

\\u, метасимвол для представления кодовых пунктов Юникода в строках Java, 145
UNC, пути, 497, 501, 504
UNICODE_CASE, константа (Java), 161
UNICODE (или U), флаг, 57, 311
uniq, утилита (UNIX), 389
UNIX_LINES, константа (Java), 165
URL, адреса
извлечение имени пользователя, 460
извлечение имени хоста, 462
извлечение номера порта, 465
извлечение пути, 467

извлечение строки запроса, 471
извлечение схемы, 458
извлечение фрагмента, 472
поиск в тексте, 442, 444, 446
проверка, 438
универсальные, проверка, 452

V

\\v (управляющий символ ASCII vertical tab), 52
Value, свойство, .NET (класс Match), 185
VB.NET, язык программирования, 136, 141
импортирование библиотеки регулярных выражений, 150
литералы регулярных выражений, 145
создание объектов регулярных выражений, 154
Visual Basic 6, язык программирования, 141

W

Windows Grep, приложение, 44

X

\\W, метасимвол, 57, 70
ограничение числа слов, 315
\\w, метасимвол, 278
ограничение числа слов, 315
\\w, символ слова, 57, 70
XML 1.0, спецификация, 550, 551, 553
XML 1.1, спецификация, 550, 551, 555

Z

\\Z, якорный метасимвол, 63, 64, 66, 280
\\z, якорный метасимвол, 63, 64

А

алгоритм Луна, 351
алфавитно-цифровые символы, ограничение ввода, 308
алфавиты Юникода, 72, 80
перечисление всех символов, 82
атомарная группировка, 108, 110, 318
атомарность опережающей и ретроспективной проверок, 116
атрибуты (языки разметки), 514
белые списки, 548
поиск определенных атрибутов, 560
атрибуты, языки разметки

Б

белые списки, 548
бесконечная длина совпадения с ретроспективной проверкой, 377
бесконечное число повторений, 98
библиотеки регулярных выражений, импортирование, 149
блоки Юникода, 72, 77
перечисление всех символов, 82
буквы устройств, извлечение из путей в Windows, 503

В

ведущие нули, удаление, 418
ведущие пробельные символы, удаление, 398
вещественные числа, 429
вложенные квантификаторы, 573
вложенные символьные классы, 58
возвраты, 101
бесполезные, устранение, 104
катастрофические, 109
время
в традиционных форматах, проверка, 300
вставка контекста совпадения в замещающий текст, 133
вставка совпадения с регулярным выражением в замещающий текст, 128
вставка части совпадения с регулярным выражением в замещающий текст, 129
выбор
поиск слов из списка, 364
выделение лексем, 222

Г

гиперссылки, включение адресов URL, 448
границы слова, 68, 302
при поиске чисел, 412
сопоставление в символьных классах, 57
графемы Юникода, 73, 81
группировка и сохранение, 87
атомарная группировка, 108, 110, 318
именованные сохранения, 131
имитация ретроспективной проверки, 118
повторение групп, 99
проверка соседних символов, 111, 313, 362

проверка условия, 119
пустые обратные ссылки, 384
ссылки на несуществующие группы, 131

Д

двоичные числа, поиск, 416
десятичные числа, преобразование римских чисел, 436
диакритические знаки, 81
диалекты
определения замещающего текста, 25
регулярных выражений, 22, 138
диапазоны внутри символьных классов, 55
ДКА, механизм регулярных выражений, 425
длина совпадения, определение, 188
длина строки, проверка, 312
доменные имена, проверка, 473
дубликаты строк, удаление, 389

Е

европейские регистрационные номера плательщиков НДС, проверка, 352
единственная строка (single line), режим, 61

З

завершающие пробельные символы, удаление, 398
замена всех совпадений, 224
замена совпадений с повторным использованием частей совпадений, 232
замещающий текст
вставка контекста совпадения, 133
вставка совпадения с регулярным выражением, 128
вставка части совпадения с регулярным выражением, 129
сгенерированный в программном коде, 238
экранирование символов, 125
замещение всех совпадений
внутри совпадений с другим регулярным выражением, 245
между совпадениями с другим регулярным выражением, 247
записи, в файлах CSV, 519
захватывающие квантификаторы, 104
знаки пунктуации
категория Юникода, 75
значения, разделенные запятыми (Comma-Separated Values, CSV), 519
извлечение полей из определенного столбца, 584
изменение разделителя, 579

И

идентификатор пространства имен (Namespace Identifier, NID), 451
извлечение списка всех совпадений, 202
извлечение текста совпадения, 181
извлечение части совпавшего текста, 194
изменение разделителя, используемого в файлах CSV, 579
имена в языке разметки XML, 518
сопоставление, 549
имена папок, извлечение из путей в Windows, 506
имена файлов
извлечение из путей в Windows, 508
удаление недопустимых символов, 511
имена, форматирование, 341
именованные обратные ссылки, 96
именованные сохранения, 93, 94, 131, 200, 236
группы с одинаковыми именами (.NET), 299
имитация ретроспективной проверки, 376, 378
импортирование библиотеки регулярных выражений, 149
имя пользователя, извлечение из адреса URL, 460
имя сервера, извлечение из путей в Windows, 504
имя хоста, извлечение из адреса URL, 462
инструменты для работы с регулярными выражениями, 27
Expresso, 40
grep, 43
myregexp.com, 36
Nregex, 34
reAnimator, 38
RegexBuddy, 28
RegexPal, 31
Rubular, 35
The Regulator, 41
популярные текстовые редакторы, 47
интервальный квантификатор, 315

исходные тексты, литералы регулярных выражений, 142
катастрофические возвраты, 109

К

категории Юникода, 72, 74
квантификаторы, 284
бесконечного числа повторений, 98
вложенные, 573
переменного числа повторений, 98
фиксированного числа повторений, 97
кодовые пункты Юникода, 72, 73
блоки, 77
которым не присвоены символы, 76
комбинированные знаки, 81
комментарии
в HTML, 515
в XML, 569, 570, 574
в файлах INI, 520
в регулярных выражениях, 122
компактная форма записи (IPv6), 482, 490
смешанная, 483, 485, 493, 494
конечная длина совпадения с ретроспективной проверкой, 377
конструкция выбора
поиск чисел в определенном диапазоне, 423
контекст совпадения, 133
контрольная сумма ISBN-10, 334
контрольная сумма ISBN-13, 334

Л

левый контекст, 134
литералы регулярных выражений в исходных текстах, 142
литеральный текст, сопоставление, 49
локальные модификаторы режима, 51
Луна алгоритм, 351

М

максимальное число повторений, 100
максимальные квантификаторы, 100
устранение бесполезных возвратов, 104
международные телефонные номера, проверка, 288
метасимволы, 49
внутри символьных классов, 55
экранирование, 403
механизмы, управляемые регулярными выражениями, 85
механизмы, управляемые текстом, 85
минимальное число повторений, 100
минимальные квантификаторы, 100, 102
устранение бесполезных возвратов, 104
мнемоники, 515, 557
мнемонические ссылки на символы, 515, 557
многострочный (multiline) режим, 66
многострочный текст
проверка числа строк, 317
модификаторы режима, 51
используемые с несохраняющими группами, 90

Н

негативная опережающая проверка, 113, 371, 374
негативная ретроспективная проверка, 113, 375
неотображаемые символы
представление в Python, 148
непечатные символы, сопоставление, 52
несохраняющая группа (?:), 279
несохраняющая группировка, 89
нечувствительность к регистру символов, режим, установка, 161
НКА, механизм регулярных выражений, 425
номера кредитных карт, проверка, 345
номера социального страхования, проверка, 325
нули
удаление ведущих нулей, 418

О

обратные ссылки, 91
именованные, 94, 96
использование пустых обратных ссылок, 384
на несуществующие группы, 131
повторное использование частей совпадений, 232
поиск повторяющихся слов, 387
обход всех совпадений в цикле, 208
объединение символьных классов, 58
объекты регулярных выражений, создание, 151
объявление типа документа (HTML), 516
ограничение числа непробельных символов, 314

ограничение числа слов, 315
ограничение числа строк в тексте, 317
оператор выбора, 85
оператор подстановки (Perl), 147
оператор сопоставления с шаблоном (Perl), 146
опережающая проверка, 112, 313
  негативная, 113, 371, 374
  позитивная, 112
определение позиции и длины совпадения, 188
относительные пути в Windows, 500, 501
отрицание внутри символьных классов, 55

П

параметры, в файлах INI, 520
параметры регулярных выражений, установка, 159
парсинг текста, 222
первое вхождение строки, сохранение, 391, 393
переменное число повторений, 98
пересечение символьных классов, 58
повторение при сопоставлении
  минимальное и максимальное число повторений, 100
  повторяющиеся части регулярного выражения, 97
  предотвращение бесконтрольных повторений, 108
повторное использование частей совпадений, 232
повторяющиеся пробельные символы, удаление, 402
повторяющиеся слова, поиск, 387
позитивная опережающая проверка, 112
  поиск слов на любом расстоянии друг от друга, 386
позитивная ретроспективная проверка, 112
позиция совпадения, определение, 188
поиск, 397
  адресов URL в тексте, 442, 444, 446
  адресов, содержащих номер почтового ящика, 339
  определенных атрибутов в тегах XML, 560
  построчный, 269
  слов, 364, 367, 371, 373, 375, 379, 387, 361
  строк, 389, 395, 397
  тегов XML, 521
  чисел, 409, 413, 416
поиск с заменой, 25
  вставка текстового литерала в замещающий текст, 124
  замена всех совпадений, 224
  замена совпадений фрагментами, сгенерированными в программном коде, 238
  замена специальных символов HTML мнемоническими ссылками, 557
  замещение всех совпадений внутри совпадений с другим регулярным выражением, 245
  замещение всех совпадений между совпадениями с другим регулярным выражением, 247
  повторное использование частей совпадений, 232
  тегов в языках разметки, 541
поля, в файлах CSV, 519
  извлечение, 584
порты, извлечение из адреса URL, 465
последнее вхождение строки, сохранение, 390, 393
построчный поиск, 269
похожие слова, поиск, 367
почтовые индексы, проверка
  Канады, 338
  Соединенного Королевства 338
  США, 336
правый контекст, 134
предотвращение бесконтрольных повторений, 108
преобразование имен из одного формата в другой, 341
пробельные символы
  в режиме свободного форматирования, 123
  категория Юникода, 74
  повторяющиеся, удаление, 402
  сопоставление в символьных классах, 57
  удаление ведущих и завершающих пробельных символов в строках, 398
  удаление из европейских регистрационных номеров плательщиков НДС, 353, 355
  удаление из номера кредитной карты, 345, 348
проверки, 283
  адресов URL, 438
  адресов электронной почты, 274

возможности совпадения в пределах испытуемой строки, 168
времени в традиционных форматах, 300
дат в традиционных форматах, 290
дат и времени в формате ISO 8601, 303
доменных имен, 473
европейских регистрационных номеров плательщиков НДС, 352
комментариев XML, 570
международных телефонных номеров, 288
номеров социального страхования, 325
полученных совпадений в программном коде, 215
путей в Windows, 495
с нулевой длиной совпадения, 112
совпадения со всей испытуемой строкой, 175
соседних символов, 111, 313, 362
негативная, 113
проверка условия, 121
строк URN, 449
телефонных номеров, 282
универсальных адресов URL, 452
условия, при сопоставлении, 119
элементов HTML, 537
проверка ввода, 312, 317, 323, 328, 336, 345, 351
номеров ISBN, 328
номеров кредитных карт, 345
номеров социального страхования, 325
ограничение возможности ввода алфавитно-цифровыми символами, 308
ограничение возможности ввода символами ANSI, 310
ограничение длины текста, 312
ограничение числа строк в тексте, 317
почтовых индексов, 336
утвердительных ответов, 323
простой текст, преобразование в HTML, 557
пустые обратные ссылки, 384
пути в файловой системе (Windows)
выделение элементов, 498
извлечение буквы устройства, 503
извлечение имени сервера, 504
извлечение имен папок, 506
извлечение имен файлов, 508
извлечение расширений файлов, 510
проверка, 495
удаление недопустимых символов, 511
пути, извлечение из адреса URL, 467

Р

разбиение строки, 253
с сохранением совпадений, 264
разделители в файлах CSV, изменение, 579
разделы, в файлах INI, 520
разность символьных классов, 58
разрывы строк, 321
в документах HTML, 557
расширения (файлов), извлечение из путей в Windows, 510
расширяемый протокол представления информации (Extensible Provisioning Protocol, EPP), 290
расширяемый язык разметки (Extensible Markup Language, XML), 517
добавление атрибутов в теги <table>, 566
поиск слов в комментариях, 574
поиск тегов, 521, 524
сопоставление с именами, 549
удаление всех тегов, за исключением выбранных, 545
удаление комментариев, 569
расширяемый язык разметки гипертекста (Extensible Hypertext Markup Language, XHTML), 516
добавление атрибутов в теги <table>, 566
поиск тегов, 521
удаление всех тегов, за исключением выбранных, 545
регистр символов
нечувствительность к регистру символов, 278
в символьных классах, 58
поиск без учета регистра символов, 51
регулярные выражения
экранирование метасимволов, 403
в стиле языка Perl, 21
режим свободного форматирования, 122
в Python, 148
рекурсивное сопоставление, 222
ретроспективная проверка, 112, 313, 363
имитация, 376, 378

применением сохраняющей группировки, 118
негативная, 113, 375
позитивная, 112
римские числа, 434

С

сбалансированное сопоставление, 222
свободное форматирование, режим, 122
установка, 161
свойства Юникода, 72
символам ^ и $ соответствуют границы строк, режим, 159
символ торговой марки, сопоставление, 71
символы
ANSI, ограничение ввода, 310
ISO-8859-1, ограничение ввода, 310
Windows-1252, ограничение ввода, 310
категории Юникода, 72
слов, 57, 70
символьные классы, 55, 284
в режиме свободного форматирования, 123
метасимволы Юникода, 82
нечувствительность к регистру символов, 58
объединение и пересечение, 58
разность, 58
сокращения, 56
слова, поиск
близко расположенных слов, 379
в XML-подобных комментариях, 574
любого слова, за которым не следует указанное слово, 373
любого слова, которому не предшествует определенное слово, 375
любых слов, за исключением некоторых, 371
одного слова из списка, 364
определенного слова, 361
повторяющихся слов, 387
похожих слов, 367
поиск строк, не содержащих определенное слово, 397
поиск строк, содержащих определенное слово, 395
сопоставление с целыми словами, 68
экранирование метасимволов, 403
смешанная форма записи (IPv6), 480, 489
компактная, 483, 485, 493, 494
совпадения внутри другого совпадения, 219
совпадения нулевой длины, 66
создание объектов регулярных выражений, 151
сокращенные формы записи символьных классов, 56
сопоставление, 397
без учета регистра символов, 51, 278
в начале и/или в конце строки, 62
внутри совпадений с другим регулярным выражением, 245
вставка контекста совпадения в замещающий текст, 133
замена совпадений с повторным использованием частей совпадений, 232
обход всех совпадений в цикле, 208
поиск совпадения внутри другого совпадения, 219
проверка условия, 119
реукрсивное, 222
с адресами IPv4, 476
с адресами IPv6, 479
сбалансированное, 222
с заголовком раздела в файле INI, 588
с именами XML, 549
с литеральным текстом, 49
с любым символом, 59
с непечатными символами, 52
с одной из нескольких альтернатив, 85
с одном символом из нескольких, 54
с парами имя-значение в файле INI, 591
с повторением, 97, 100, 108
с разделом в файле INI, 589
с ранее совпавшим текстом, 91
с тегами в языках разметки, 521
с целыми словами, 68
сортировка строк, 390, 392
сохраняющая группировка, 87
извлечение информации о совпадении, 194
именованное сохранение, 93, 200, 236, 299
именованные сохранения, 131
имитация ретроспективной проверки, 118
использование пустых обратных ссылок, 384
повторное использование частей совпадений, 232

повторный поиск соответствия с ранее совпавшим текстом, 91
поиск повторяющихся слов, 387
проверка условия, 119
ссылки на несуществующие группы, 131
стандартная форма записи (IPv6), 479
строки
строки в апострофах (PHP), 146
строки в кавычках
C#, 144
Java, 145
Perl, 147
PHP, 146
VB.NET, 145
строки в тройных кавычках (Python), 148
проверка адресов URL, 438
проверка длины ввода, 312
проверка строк URN, 449
разбиение, 253
экранирование метасимволов, 403
строки запросов, извлечение из адреса URL, 471
строки текста, 389
не содержащие определенное слово, 397
содержащие определенное слово, 395
сортировка с целью удаления дубликатов, 390, 392
удаление ведущих и завершающих пробельных символов, 398
удаление дубликатов, 389
удаление повторяющихся пробельных символов, 402
схемы, извлечение из адреса URL, 458
сырые строки (Python), 148

Т

теги (языки разметки), 514
добавление атрибутов в теги <table>, 566
замещение одного тега другим, 541
сопоставление, 521
удаление всех тегов, за исключением выбранных, 545
телефонные номера
международные, 288
проверка, 282
точка это все (dot all), режим, 61
точке соответствуют границы строк, режим, 61, 159

У

удаление
ведущих нулей, 418
недопустимых символов из имен файлов, 511
повторяющихся пробельных символов, 402
универсальные адреса URL, проверка, 452
унифицированное имя ресурса (Uniform Resource Name, URN), проверка, 449
управляющие символы
категория Юникода, 76
условные конструкции, 381
установка параметров регулярных выражений, 159
утвердительные ответы, 323

Ф

файлы инициализаци (INI), 520
фиксированная длина совпадения с ретроспективной проверкой, 377
фиксированное число повторений, 97
формат даты и времени ISO 8601, 303
форматирование
проверка времени, 300, 303
проверка дат, 290, 303
телефонных номеров, 282
форматирование имен, 341
форматы дат
проверка, 290
фрагмент, извлечение из адреса URL, 472

Ц

целые числа
в определенном диапазоне, поиск, 419, 427
поиск и идентификация, 409
с разделителями групп разрядов, 433
удаление ведущих нулей, 418
цифры (числа), 411
категория Юникода, 75
соответствия в символьных классах, 57

Ч

числа
IPv4, адреса, сопоставление, 476
IPv6, адреса, сопоставление, 479
вещественные, 429
двоичные, поиск, 416

европейские регистрационные номера плательщиков НДС, проверка, 352
номера ISBN, проверка, 328
числа
номера кредитных карт, проверка, 345
проверка номеров социального страхования, 325
римские, 434
с разделителями групп разрядов, 433
удаление ведущих нулей, 418
целые, 409, 419
шестнадцатеричные, 427
поиск, 413
числовых ссылок на символы, поиск, 515

Ш

шестнадцатеричные числа, поиск, 413
в определенном диапазоне, 427

Э

экранирование, 49
блока, 50, 407
внутри символьных классов, 55
как и когда, 403
символы в замещающем тексте, 125
электронная почта, проверка адресов, 274
элементы, языки разметки, 514

Ю, Я

Юникод, 71
языки, алфавиты Юникода, 80
язык разметки гипертекста (Hypertext Markup Language, HTML), 513
добавление атрибутов в теги <table>, 566
добавление тегов <p> в простой текст, 557
замещение тега <b> тегом , 541
поиск тегов, 521
проверка тегов, 537
удаление всех тегов, за исключением выбранных, 545
якори, 63

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-181-3, название «Регулярные выражения. Сборник рецептов» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.