Создание веб-приложений с быстрым интерфейсом

Оптимизация

JavaScript

производительности

Символ®

O'REILLY® YAHOO! PRESS

Николас Закас

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-213-1, название «JavaScript. Оптимизация производительности» – 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, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

High Performance JavaScript

Nicholas Zakas

O'REILLY®

JavaScript

Оптимизация производительности

Николас Закас

Санкт-Петербург — Москва
2012

Николас Закас

JavaScript. Оптимизация производительности

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*Ю. Бочина*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*

Закас Н.

JavaScript. Оптимизация производительности. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 256 с., ил.

ISBN 978-5-93286-213-1

Если вы относитесь к подавляющему большинству веб-разработчиков, то наверняка широко применяете JavaScript для создания интерактивных веб-приложений с малым временем отклика. Проблема состоит в том, что строки с программным кодом на языке JavaScript могут замедлять работу приложений. Эта книга откроет вам приемы и стратегии, которые помогут в ходе разработки устранить узкие места, влекущие за собой снижение производительности. Вы узнаете, как ускорить выполнение, загрузку, операции с деревом DOM, работу страницы в целом и многое другое.

Николас Закас, программист из компании Yahoo!, специализирующийся на разработке пользовательских интерфейсов веб-приложений, и еще пять экспертов в области использования JavaScript представят оптимальные способы загрузки сценариев и другие приемы программирования, которые помогут вам обеспечить наиболее эффективное и быстрое выполнение программного кода на JavaScript. Вы познакомитесь с наиболее передовыми приемами сборки и развертывания файлов в действующем окружении и с инструментами, которые помогут в поиске проблем.

Книга адресована веб-разработчикам со средним и высоким уровнем владения языком JavaScript, желающим повысить производительность интерфейсов веб-приложений.

ISBN 978-5-93286-213-1
ISBN 978-0-596-80279-0 (англ)

© Издательство Символ-Плюс, 2012
Authorized Russian translation of the English edition of High Performance JavaScript, First Edition ISBN 9780596802790 © 2010 Yahoo!, Inc. This translation is published and sold by permission of O'Reilly Media, 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.
Подписано в печать 26.06.2012. Формат 70×100 1/16.
Печать офсетная. Объем 16 печ. л.

Я посвящаю эту книгу своей семье – маме,
отцу и Грегу, – на чью любовь и поддержку
я всегда мог положиться

Оглавление

Предисловие 11

1. Загрузка и выполнение 21
Местоположение сценария 22
Группировка сценариев 24
Неблокирующая загрузка сценариев 26
Отложенные сценарии 26
Динамические элементы script 27
Инъекция сценариев с помощью XMLHttpRequest 31
Рекомендуемые способы неблокирующей загрузки 32
В заключение 36

2. Доступ к данным 37
Управление областью видимости 38
Цепочки областей видимости и разрешение идентификаторов 39
Производительность разрешения идентификаторов 41
Увеличение цепочки областей видимости 44
Динамические области видимости 46
Замыкания, области видимости и память 47
Члены объектов 50
Прототипы 50
Цепочки прототипов 52
Вложенные члены 54
Кэширование значений членов объектов 55
В заключение 57

3. Работа с деревом DOM 58
Модель DOM в мире броузеров 58
Врожденная медлительность 59
Доступ к дереву DOM и его модификация 59
Свойство innerHTML в сравнении с методами DOM 61
Копирование узлов 65
HTML-коллекции 66
Обход дерева DOM 71

Перерисовывание и перекомпоновка 75
Когда происходит перекомпоновка? 75
Буферизация и применение изменений в дереве отображения 76
Уменьшение количества операций перерисовывания и перекомпоновки 78
Кэширование информации о размещении 81
Исключение элементов из потока отображения для внесения изменений 82
IE и :hover 82
Делегирование обработки событий 83
В заключение 85

4. Алгоритмы и управление потоком выполнения 87

Циклы 87
Типы циклов 88
Производительность цикла 89
Итерации на основе функций 94
Условные инструкции 95
Сравнение if-else и switch 95
Оптимизация инструкций if-else 97
Поисковые таблицы 99
Рекурсия 101
Ограниченность размера стека вызовов 101
Шаблоны реализации рекурсии 103
Итерации 104
Мемоизация 105
В заключение 108

5. Строки и регулярные выражения 109

Конкатенация строк 110
Операторы плюс (+) и плюс-равно (+=) 110
Слияние элементов массива 113
String.prototype.concat() 115
Оптимизация регулярных выражений 116
Как работают регулярные выражения 117
Возвраты 118
Исключение возвратов 121
Примечание к измерению производительности 127
Дополнительные пути повышения производительности регулярных выражений 127
Когда не следует использовать регулярные выражения 131
Усечение строк 132
Усечение с применением регулярных выражений 132
Усечение без применения регулярных выражений 135
Смешанное решение 136

В заключение . . . 138

6. Отзывчивые интерфейсы . . . 139

Поток выполнения пользовательского интерфейса броузера . . . 139
Ограничения броузеров . . . 141
Слишком долго – это сколько? . . . 143
Использование таймеров . . . 145
Основы таймеров . . . 145
Точность таймера . . . 148
Обработка массивов с помощью таймеров . . . 148
Деление заданий . . . 150
Хронометраж выполнения программного кода . . . 152
Таймеры и производительность . . . 154
Фоновые потоки выполнения . . . 155
Окружение фонового потока выполнения . . . 155
Взаимодействие с фоновыми потоками выполнения . . . 156
Загрузка внешних файлов . . . 157
Практическое использование . . . 157
В заключение . . . 159

7. Ajax . . . 160

Передача данных . . . 160
Запрос данных . . . 161
Отправка данных . . . 168
Форматы данных . . . 171
XML . . . 171
JSON . . . 175
HTML . . . 179
Нестандартное форматирование . . . 181
Заключительные выводы о форматах данных . . . 184
Рекомендации по повышению производительности Ajax . . . 185
Кэширование данных . . . 185
Известные ограничения библиотек поддержки Ajax . . . 188
В заключение . . . 189

8. Приемы программирования . . . 191

Предотвращение повторной интерпретации . . . 191
Использование литералов объектов/массивов . . . 193
Предотвращение повторного выполнения работы . . . 194
Отложенная загрузка . . . 195
Предварительная условная загрузка . . . 196
Использование сильных сторон . . . 197
Битовые операторы . . . 197
Встроенные методы . . . 200
В заключение . . . 202

9. Сборка и развертывание высокопроизводительных приложений на JavaScript 203

Apache Ant 204
Объединение JavaScript-файлов 205
Предварительная обработка JavaScript-файлов 206
Минификация JavaScript-файлов 209
Сборка в виде отдельного этапа или во время выполнения 211
Сжатие JavaScript-сценариев 211
Кэширование JavaScript-файлов 213
Решение проблем, связанных с кэшированием 214
Использование сети распространения содержимого 215
Развертывание JavaScript-ресурсов 215
Гибкий процесс сборки JavaScript-файлов 216
В заключение 218

10. Инструменты 219

Профилирование JavaScript-сценариев 220
YUI Profiler 221
Анонимные функции 225
Firebug 226
Панель профилировщика в консоли 227
Прикладной интерфейс консоли 228
Панель Net 229
Инструменты разработчика в Internet Explorer 230
Веб-инспектор в броузере Safari 233
Панель Profiles 233
Панель Resources 236
Инструменты разработчика в Chrome 237
Блокирование сценариями отображения страницы 238
Page Speed 239
Fiddler 241
YSlow 243
dynaTrace Ajax Edition 245
В заключение 248

Алфавитный указатель 249

Предисловие

Когда в 1996 году в Netscape Navigator впервые появилась поддержка JavaScript, производительность еще не имела большого значения. Интернет находился в зачаточном состоянии, и обмен данными происходил очень медленно. Из-за медленных коммутируемых соединений и невысокой вычислительной мощности домашних компьютеров путешествие по Всемирной паутине становилось скорее упражнением на выдержку. Пользователи привыкли ждать загрузки веб-страниц, и когда загрузка успешно завершалась, это воспринималось как праздник.

Первоначальная цель JavaScript заключалась в повышении удобства использования веб-страниц. Вместо того чтобы для решения простых задач, таких как проверка правильности заполнения формы, обращаться к серверу, поддержка JavaScript дала возможность встраивать необходимую функциональность непосредственно в страницы. Это позволило экономить время на обращениях к серверу. Представьте, что вы заполнили длинную форму, отправили ее, ждали ответа 30–60 секунд, и все это только для того, чтобы получить сообщение, что какое-то одно поле заполнено неправильно. Можно смело утверждать, что появление JavaScript сэкономило уйму времени первым пользователям Интернета.

С развитием Интернета

В течение следующего десятилетия росла мощность компьютеров и продолжалось развитие Интернета. Прежде всего, и компьютеры, и Интернет стали намного быстрее. Резкое увеличение быстродействия микропроцессоров, удешевление памяти и появление оптоволоконных каналов открыли новую эру развития Интернета. С расширением доступности высокоскоростных соединений начали «тяжелеть» веб-страницы за счет встраивания в них больших объемов информации и мультимедиа. Всемирная паутина превратилась из множества слабо связанных друг с другом документов в единое целое с различными интерфейсами и дизайнами. Изменилось все, кроме JavaScript.

Инструмент, прежде использовавшийся как средство, позволяющее уменьшить число обращений к серверу, стал применяться все шире. Там, где раньше использовались десятки строк кода на JavaScript, стали применяться сценарии, насчитывающие сотни, а порой и тысячи строк. Появление Internet Explorer 4 и динамического HTML (возмож-

ности изменять некоторые аспекты страниц без их перезагрузки) давало уверенность, что объем программного кода на языке JavaScript в страницах со временем будет только расти.

Последним крупным шагом в развитии броузеров было появление объектной модели документа (Document Object Model, DOM), унифицированного подхода к реализации динамического HTML, выполненной в Internet Explorer 5, Netscape 6 и Opera. Реализации этой модели близко соответствовали третьей версии стандарта ECMA-262. С появлением броузеров, поддерживающих DOM и (более или менее) одну и ту же версию JavaScript, родилась платформа для веб-приложений. Несмотря на огромный скачок в развитии и обобщенный API, используемый при создании сценариев на языке JavaScript, сами реализации JavaScript, выполняющие эти сценарии, практически не развивались.

Зачем нужна оптимизация

Современные веб-приложения с тысячами строк программного кода на JavaScript продолжают выполняться под управлением практически тех же реализаций JavaScript, которые в 1996 году поддерживали веб-страницы с несколькими десятками строк кода. Во многих отношениях развитие JavaScript в броузерах отставало от общего прогресса, что мотивировалось необходимостью обеспечить всеобъемлющую совместимость JavaScript. Особенно ярко это отставание проявилось в броузере Internet Explorer 6, первый выпуск которого был объявлен самым стабильным и быстрым, но позднее был признан самой ужасной платформой для веб-приложений из-за ошибок и низкой производительности.

В действительности, IE6 не был медленнее, просто от него требовалось больше, чем прежде. Веб-приложения, создававшиеся в 2001 году, когда появился IE6, были намного легче и содержали меньше программного кода JavaScript, чем те, что создавались в 2005 году. Увеличение объемов программного кода стало оказывать отрицательное влияние на работу интерпретатора JavaScript в IE6, так как в нем использовалась статическая процедура сборки мусора. Чтобы определить, когда следует запускать сборку мусора, интерпретатор сравнивал количество объектов в памяти с фиксированным числом. Ранее разработчики веб-приложений редко сталкивались с этим пределом, но с увеличением объемов программного кода стало расти число объектов, и сложные веб-приложения стали все чаще превышать установленный порог. Проблема очевидна: разработчики и веб-приложения продолжали свое развитие, тогда как реализации JavaScript оставались на прежнем уровне.

Несмотря на то что другие броузеры имели более логичные процедуры сборки мусора и несколько более высокую производительность, тем не менее сам программный код все еще выполнялся интерпретатором. Выполнение интерпретируемого программного кода всегда протекает медленнее, чем скомпилированного из-за необходимости трансляции исходного программного кода в машинные инструкции. Какими бы умными

и оптимизированными ни были интерпретаторы, их использование всегда будет приводить к снижению производительности.

Компиляторы до краев наполнены всякими оптимизациями, позволяющими разработчикам писать программы, не заботясь особенно об их оптимизации. Опираясь на лексический анализ, компилятор способен определить, что делает программный код, оптимизировать его и произвести быстро выполняющийся машинный код. Интерпретаторы не обладают такими широкими возможностями к оптимизации и обычно выполняют программный код так, как он написан.

В результате язык JavaScript вынуждает разработчика взять на себя заботу об оптимизации, которую обычно выполняют компиляторы в других языках программирования.

Следующее поколение интерпретаторов JavaScript

Первое существенное увеличение производительности интерпретаторы JavaScript получили в 2008 году. Компания Google представила свой, совершенно новый броузер Chrome. Он стал первым броузером, оснащенным оптимизированным интерпретатором JavaScript под кодовым названием V8. Интерпретатор JavaScript V8 включает механизм компиляции во время выполнения (Just-In-Time, JIT), который преобразует программный код на языке JavaScript в машинный код и выполняет его. Это обеспечило чрезвычайно высокую скорость выполнения программного кода JavaScript.

Другие производители броузеров последовали этому примеру и реализовали собственные оптимизированные версии интерпретаторов JavaScript. Броузер Safari 4 был оснащен JIT-интерпретатором JavaScript под названием Squirrel Fish Extreme (иногда его также называют Nitro), а Firefox 3.5 – интерпретатором TraceMonkey, оптимизирующим часто выполняемые фрагменты программного кода.

Все эти новейшие интерпретаторы JavaScript выполняют на этапе компиляции оптимизацию, где только возможно. Когда-нибудь разработчики смогут полностью избавиться от необходимости оптимизировать свой программный код. Однако это время пока не наступило.

Производительность продолжает быть проблемой

Несмотря на существенное улучшение производительности интерпретаторов, в JavaScript еще остаются некоторые аспекты, которые не могут быть оптимизированы этими новыми интерпретаторами. Задержки, обусловленные особенностями работы сети и операциями отображения страниц, являются областью, которая пока не может быть в достаточной

степени оптимизирована броузерами. Такие простые оптимизации, как встраивание функций, свертка кода и конкатенация строк, легко реализуются в компиляторах, однако из-за динамичности и многогранности веб-приложений все эти оптимизации решают проблему производительности лишь частично.

Хотя новые реализации JavaScript позволяют в будущем надеяться на работу в гораздо более быстром Интернете, тем не менее стоящие сегодня задачи повышения производительности в ближайшее время останутся важными и насущными.

Приемы и подходы, обсуждаемые в этой книге, охватывают самые разные аспекты программирования на языке JavaScript, такие как время выполнения, загрузка, взаимодействие с DOM, жизненный цикл страниц и многие другие. Из рассматриваемых проблем лишь малая часть, связанная с производительностью ядра (ECMAScript), может быть устранена за счет усовершенствования интерпретаторов JavaScript, но это еще только должно произойти.

В решении остальных рассмотренных проблем увеличение производительности интерпретаторов JavaScript не решает проблему быстродействия. Это взаимодействие с деревом DOM, сетевые задержки, блокировка, параллельная загрузка сценариев JavaScript и многое другое. Эти вопросы не только не будут сняты в будущем, но станут основным предметом исследований по мере повышения производительности интерпретаторов JavaScript.

Организация книги

Главы в этой книге организованы в соответствии с обычным циклом разработки сценариев на языке JavaScript. Сначала, в главе 1, обсуждаются наиболее оптимальные способы загрузки JavaScript-сценариев в страницы. Главы 2–8 представляют конкретные приемы программирования, которые позволяют максимально увеличить скорость выполнения программного кода. В главе 9 обсуждаются наиболее оптимальные способы создания и развертывания JavaScript-файлов в действующем окружении, а глава 10 охватывает инструменты исследования производительности, способные помочь в поиске проблем, которые могут появиться после развертывания сценариев. Пять из этих глав были написаны другими авторами:

- Глава 3 «Работа с деревом DOM», Стоян Стефанов (Stoyan Stefanov)
- Глава 5 «Строки и регулярные выражения», Стивен Левитан (Steven Levithan)
- Глава 7 «Ajax», Росс Хармс (Ross Harmes)
- Глава 9 «Сборка и развертывание высокопроизводительных приложений на JavaScript», Жюльен Лекомте (Julien Lecomte)
- Глава 10 «Инструменты», Мэтт Суини (Matt Sweeney)

Все эти авторы являются опытными веб-разработчиками, внесшими существенный вклад в веб-разработку. Их имена упоминаются на первых страницах соответствующих глав, давая читателю возможность идентифицировать их труд.

Загрузка JavaScript

Глава 1 «Загрузка и выполнение» начинается с самой основной операции – загрузки программного кода JavaScript в страницу. Повышение производительности JavaScript действительно начинается с выбора наиболее оптимальных способов загрузки программного кода. Эта глава рассматривает проблемы производительности, связанные с загрузкой программного кода, и представляет несколько способов ускорить ее.

Приемы программирования

Плохо написанный программный код, использование неэффективных алгоритмов и утилит являются существенным фактором снижения производительности JavaScript. Следующие семь глав демонстрируют проблемы, связанные с применением тех или иных приемов программирования, и представляют более скоростные способы решения тех же задач.

Глава 2 «Доступ к данным» описывает, как JavaScript-сценарии сохраняют и извлекают данные. Важно не только решить, какие данные хранить, но и где их хранить, и эта глава демонстрирует, как такие понятия, как цепочки областей видимости и цепочки прототипов, могут влиять на производительность сценариев.

Главу 3 «Работа с деревом DOM» написал Стоян Стефанов (Stoyan Stefanov), хорошо знакомый с особенностями работы внутренних механизмов веб-броузера. Стоян объясняет, что из-за особенностей реализации операции с деревом DOM являются самыми медленными в JavaScript. Он рассматривает все аспекты, касающиеся DOM, включая описание того, как перерисовка и реорганизация элементов страниц могут уменьшить скорость выполнения программного кода.

Глава 4 «Алгоритмы и управление потоком выполнения» описывает, как распространенные приемы программирования, такие как циклы и рекурсия, могут работать против вас, снижая производительность во время выполнения. Здесь рассматриваются такие методы оптимизации, как мемоизация, а также некоторые ограничения, накладываемые броузерами.

Многие веб-приложения выполняют сложные операции со строками, обсуждение которых в главе 5 «Строки и регулярные выражения» ведет Стивен Левитан (Steven Levithan), эксперт по обработке текстовых данных. На протяжении многих лет веб-разработчики боролись с не-

эффективной реализацией строковых операций в броузерах, и Стивен объясняет, почему некоторые операции выполняются слишком медленно и как обходить это при разработке.

Глава 6 «Отзывчивые интерфейсы» концентрируется на ощущениях пользователей. В процессе выполнения JavaScript-сценарий может «подвешивать» броузер, вызывая неудовольствие у пользователей. Эта глава рассматривает некоторые приемы, гарантирующие сохранение отзывчивости пользовательского интерфейса в любых условиях.

В главе 7 «Ajax» Росс Хармс (Ross Harmes) обсуждает способы реализации быстрых взаимодействий между клиентом и сервером. Росс показывает, как различные форматы представления данных могут влиять на производительность механизмов Ajax и почему использование объекта `XMLHttpRequest` не всегда является лучшим выбором.

Глава 8 «Приемы программирования» представляет собой коллекцию наиболее эффективных приемов программирования, уникальных для языка JavaScript.

Развертывание

После того как JavaScript-сценарий будет написан и отлажен, наступает момент сделать внесенные изменения доступными всем желающим. Однако недостаточно просто скопировать исходные файлы на действующий сервер. В главе 9 «Сборка и развертывание высокопроизводительных приложений на JavaScript» Жюльен Лекомте (Julien Lecomte) демонстрирует, как повысить производительность сценариев при развертывании. Жюльен обсуждает особенности использования систем сборки, автоматически минимизирующих файлы, и применение функции сжатия в протоколе HTTP при отправке этих файлов броузерам.

Тестирование

Следующий шаг после развертывания JavaScript-сценариев – тестирование их производительности. Методологию тестирования и применяемые при этом инструменты представит Мэтт Суини (Matt Sweeney) в главе 10 «Инструменты». Он расскажет, как с помощью JavaScript измерить производительность, и познакомит с распространенными инструментами прослушивания протокола HTTP, позволяющими определить производительность и выявить имеющиеся проблемы.

Кому адресована эта книга

Эта книга адресована веб-разработчикам со средним и высоким уровнем владения языком JavaScript, желающим повысить производительность интерфейсов веб-приложений.

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Курсив применяется для выделения новых терминов, адресов электронной почты, имен файлов и их расширений.

`Моноширинный шрифт`

Применяется для представления листингов программ, а также для выделения в обычном тексте программных элементов, таких как имена функций и переменных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Используется для выделения команд или текста, который должен быть введен пользователем.

`Моноширинный курсив`

Обозначает элементы в программном коде, которые должны быть замещены значениями, определяемыми реальным контекстом.

Так выделяются советы, предложения или примечания общего характера.

Так выделяются предупреждения или предостережения.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. 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 существенные части 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 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: «High Performance JavaScript, by Nicholas C. Zakas. Copyright 2010 Yahoo!, Inc., 978-0-596-80279-0».

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

Safari® Books Online

Safari Books Online – это виртуальная библиотека, которая позволяет легко и быстро находить ответы на вопросы среди более чем 7500 технических и справочных изданий и видеороликов.

Подписавшись на услугу, вы сможете загружать любые страницы из книг и просматривать любые видеоролики из нашей библиотеки. Читать книги на своих мобильных устройствах и сотовых телефонах. Получать доступ к новинкам еще до того, как они выйдут из печати. Читать рукописи, находящиеся в работе, и посылать свои отзывы авторам. Копировать и вставлять отрывки программного кода, определять свои предпочтения, загружать отдельные главы, устанавливать закладки на ключевые разделы, оставлять примечания, печатать страницы и пользоваться массой других преимуществ, позволяющих экономить ваше время.

Благодаря усилиям O'Reilly Media, данная книга также доступна через услугу Safari Books Online. Чтобы получить полный доступ к электронной версии этой книги, а также книг с похожими темами издательства O'Reilly и других издательств, подпишитесь бесплатно по адресу *http://my.safaribooksonline.com*.

Как с нами связаться

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

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

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

http://www.oreilly.com/catalog/9780596802790

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

В первую очередь мне хотелось бы поблагодарить своих соавторов: Мэтта Суини (Matt Sweeney), Стояна Стефанова (Stoyan Stefanov), Стивена Левитана (Steven Levithan), Росса Хармса (Ross Harmes) и Жюльена Лекомте (Julien Lecomte). Их опыт и знания сделали работу над книгой более увлекательной, а саму книгу – более убедительной.

Спасибо всем знатокам приемов повышения производительности, с которыми мне доводилось встречаться и общаться, особенно Стиву Содерсу (Steve Souders), Тенни Теуреру (Tenni Theurer) и Николь Салливан (Nicole Sullivan). Вы помогли мне расширить кругозор, когда я начинал изучать проблему производительности веб-приложений, за что я безмерно благодарен вам.

Большое спасибо всем, кто рецензировал книгу перед публикацией, включая Райана Грова (Ryan Grove), Оливера Ханта (Oliver Hunt), Мэттью Рассела (Matthew Russell), Теда Родена (Ted Roden), Реми Шарпа (Remy Sharp) и Венкатесварана Юдайашанкара (Venkateswaran Udayasankar). Они отозвались первыми, и их отзывы были неоценимы при подготовке книги к печати.

И огромное спасибо всем сотрудникам O'Reilly и Yahoo!, кто помогал в создании этой книги. Я хотел написать ее для Yahoo! с тех пор, как поступил в эту компанию в 2006 году, а издательство Yahoo! Press дало великолепную возможность реализовать этот замысел.

1

Загрузка и выполнение

Производительность JavaScript в броузере является, пожалуй, одной из самых важных проблем, с которыми приходится сталкиваться разработчикам. Проблема осложняется монопольной природой JavaScript, которая проявляется в том, что на время выполнения JavaScript-сценария все остальное замирает. На практике в большинстве броузеров обновление пользовательского интерфейса и выполнение программного кода на JavaScript происходит в рамках одного процесса, поэтому в каждый конкретный момент времени может выполняться только что-то одно. Чем дольше выполняется JavaScript-сценарий, тем дольше броузер не сможет откликаться на действия пользователя.

В простейшем случае это означает, что присутствия тега `<script>` уже достаточно, чтобы заставить страницу ждать, пока сценарий будет проанализирован и выполнен. При этом неважно, содержит ли этот тег фактический программный код или подключает внешний файл, – загрузка и отображение страницы будут остановлены до завершения работы сценария. Это обязательная часть жизненного цикла страницы, потому что в ходе выполнения сценарий может внести в страницу изменения. Типичным примером может служить применение метода `document.write()` в середине страницы (что часто используется для вывода рекламы). Например:

```
<html>
<head>
  <title>Script Example</title>
</head>
<body>
  <p>
  <script type="text/javascript">
    document.write("The date is " + (new Date()).toDateString());
  </script>
  </p>
```

```
</body>
</html>
```

Когда броузер встречает тег <script>, как в данном примере HTML-страницы, он не может знать, вставит ли JavaScript-сценарий какой-либо текст в тег <p>, добавит ли дополнительные элементы или даже просто закроет тег. Поэтому броузер вынужден остановить обработку страницы, выполнить сценарий и только после этого продолжить анализ и отображение страницы. То же происходит, когда с помощью атрибута src загружается внешний сценарий; броузер сначала должен загрузить программный код из внешнего файла, что занимает время, а затем произвести синтаксический анализ программного кода и выполнить его. На это время процесс отображения и взаимодействия с пользователем полностью блокируется.

Двумя авторитетными источниками информации о влиянии JavaScript-сценариев на скорость загрузки страниц являются группа Yahoo! Exceptional Performance (*http://developer.yahoo.com/performance/*) и Стив Содерс (Steve Souders), автор книг «High Performance Web Sites» (*http://oreilly.com/catalog/9780596529307/*) (O'Reilly) и «Even Faster Web Sites» (*http://oreilly.com/catalog/9780596522315/*) (O'Reilly). Эта глава в значительной мере опирается на их исследования.

Местоположение сценария

Спецификация HTML 4 указывает, что тег <script> в HTML-документах может быть помещен внутрь тега <head> или <body> и может встречаться в любом из них любое количество раз. Традиционно теги <script>, используемые для загрузки внешних JavaScript-файлов, помещаются в тег <head> рядом с тегами <link>, загружающими внешние CSS-файлы, и другой метаинформацией о странице. Теория утверждает, что ссылки на стили и сценарии следует стараться помещать рядом, чтобы загрузить их в первую очередь и обеспечить корректное отображение и работу страницы. Например:

```
<html>
<head>
  <title>Script Example</title>
  <-- Пример неэффективного размещения сценариев -->
  <script type="text/javascript" src="file1.js"></script>
  <script type="text/javascript" src="file2.js"></script>
  <script type="text/javascript" src="file3.js"></script>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

Несмотря на безобидность этого решения, оно порождает несколько проблем, связанных с производительностью: в теге <head> присутствуют три ссылки на JavaScript-файлы, которые требуется загрузить. Поскольку каждый тег <script> блокирует процесс отображения страницы до окончания загрузки и выполнения сценария, создается ощущение снижения скорости выполнения этой страницы. Имейте в виду, что броузеры ничего не отображают, пока не встретят открывающий тег <body>. Помещение сценариев в начало страницы, как в данном примере, обычно приводит к заметной задержке, что часто означает отображение пустой страницы, а для пользователя – отсутствие возможности приступить к чтению или начать как-то иначе взаимодействовать со страницей. Чтобы лучше понять, что происходит на практике, взгляните на следующую временную диаграмму, где показано, в какие моменты загружается каждый из требуемых ресурсов. На рис. 1.1 видно, когда загружается каждый сценарий и файл с таблицей стилей.

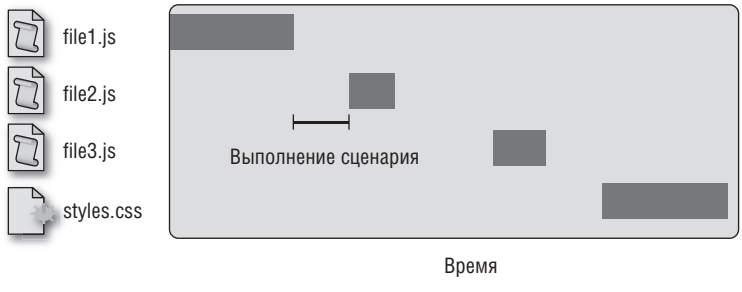


Рис. 1.1. *Выполнение JavaScript-сценария блокирует загрузку остальных файлов*

На рис. 1.1 можно наблюдать интересную картину. На время загрузки первого JavaScript-файла блокируется загрузка других файлов. Далее следует задержка между моментом окончания загрузки файла *file1.js* и началом загрузки файла *file2.js*. Этот интервал времени занимает выполнение программного кода из файла *file1.js*. Перед началом загрузки каждого файла происходит задержка, необходимая для загрузки и выполнения предыдущего файла. И все это время, пока выполняется загрузка файлов, пользователь сидит перед пустым экраном. Таково поведение большинства современных броузеров.

Броузеры Internet Explorer 8, Firefox 3.5, Safari 4 и Chrome 2 поддерживают возможность параллельной загрузки файлов JavaScript. Это здорово, потому что теги <script> не блокируют другие теги <script> из загружающихся внешних ресурсов. Но, к сожалению, загрузка JavaScript-сценариев блокирует загрузку других ресурсов, например изображений. И даже при том что загрузка одного сценария не блокирует загрузку другого сценария, броузер вынужден ждать, пока не будут загружены и выполнены JavaScript-сценарии, прежде чем он сможет продолжить

отображение страницы. Таким образом, повышение производительности в последних версиях броузеров за счет поддержки параллельной загрузки не решает проблему полностью. Блокировка на время загрузки и выполнения сценария все еще остается проблемой.

Поскольку сценарии блокируют загрузку всех остальных ресурсов, рекомендуется помещать все теги <script> как можно ближе к концу тега <body>, чтобы они не влияли на загрузку всей страницы. Например:

```
<html>
<head>
  <title>Script Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>

  <-- Пример рекомендуемого местоположения сценариев -->
  <script type="text/javascript" src="file1.js"></script>
  <script type="text/javascript" src="file2.js"></script>
  <script type="text/javascript" src="file3.js"></script>
</body>
</html>
```

Этот пример представляет рекомендуемое местоположение тегов <script> в HTML-файле. Даже если загрузка одного сценария будет блокировать загрузку остальных, к этому моменту остальная часть страницы уже будет загружена и показана, и у пользователя не возникнет ощущения медленной загрузки. Первое правило, выработанное группой Yahoo! Exceptional Performance, касающееся JavaScript-сценариев: *помещайте сценарии в конец*.

Группировка сценариев

Поскольку каждый тег <script> блокирует отображение страницы в ходе начальной ее загрузки, будет нелишним ограничить общее количество тегов <script> в странице. Это относится и к встроенным, и к внешним сценариям. Каждый раз когда в ходе синтаксического анализа HTML-страницы встречается тег <script>, происходит задержка до окончания выполнения программного кода; минимизация общего количества этих задержек повысит общую скорость загрузки страницы.

Кроме того, Стив Содерс (Steve Souders) обнаружил, что встроенные сценарии, расположенные после тега <link>, ссылающегося на внешнюю таблицу стилей, также заставляют броузер приостановиться до окончания загрузки этой таблицы стилей. Такое поведение предусмотрено с целью обеспечить наличие как можно более полной информации о стилях к тому моменту, когда начнется выполнение встроенного сценария. По этой причине Содерс рекомендует никогда не помещать встроенные сценарии после тегов <link>.

Эта проблема имеет свою специфику в случае внешних JavaScript-файлов. Каждый HTTP-запрос вносит свой вклад в снижение производительности, вследствие чего загрузка одного файла размером в 100 Кбайт будет выполняться быстрее, чем загрузка четырех файлов размером по 25 Кбайт. Поэтому будет полезно ограничить количество внешних файлов сценариев, на которые ссылается страница.

Обычно крупные веб-сайты или веб-приложения требуют загрузки нескольких JavaScript-файлов. Вы можете уменьшить их влияние на производительность, объединив эти файлы в один большой файл и оформив ссылку на этот файл в виде единственного тега `<script>`. Объединение файлов можно выполнить статически с помощью инструментов сборки (обсуждаются в главе 9) или динамически с помощью специализированных инструментов, таких как инструменты, используемые компанией Yahoo!.

Компания Yahoo! создала инструмент динамической сборки для распространения файлов библиотеки Yahoo! User Interface (YUI) через свою службу доставки содержимого (Content Delivery Network, CDN). Любой веб-сайт может объединить произвольное количество файлов из библиотеки YUI, воспользовавшись специализированным URL-адресом и указав требуемые файлы. Например, следующий URL-адрес подключает два файла: *http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js*.

При обращении к данному URL-адресу будут загружены версии 2.7.0 файлов *yahoo-min.js* и *event-min.js*. На сервере эти файлы хранятся отдельно друг от друга и объединяются только при обращении по этому URL-адресу. Вместо использования двух тегов `<script>` (по одному для каждого файла) можно оставить единственный тег `<script>`, загружающий оба файла сразу:

```
<html>
<head>
  <title>Script Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>

  <-- Пример рекомендуемого местоположения сценариев -->
  <script type="text/javascript" src="
http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&
2.7.0/build/event/event-min.js "></script>
</body>
</html>
```

Этот пример страницы, в конце которой присутствует единственный тег `<script>`, загружающий сразу несколько JavaScript-файлов, демонстрирует наиболее оптимальный способ подключения внешних файлов на JavaScript.

Неблокирующая загрузка сценариев

Стремление реализаций JavaScript блокировать процессы в броузере (и HTTP-запросы, и пользовательский интерфейс) является наиболее известной причиной снижения производительности, с которой приходится сталкиваться разработчикам. Уменьшение размеров файлов сценариев и ограничение количества HTTP-запросов – это лишь первый шаг на пути создания отзывчивого веб-приложения. Чем больше функциональности требуется от приложения, тем больше программного кода оно должно включать, и поэтому уменьшить размеры файлов получится не всегда. Объединение всех сценариев в один очень большой файл приведет лишь к тому, что броузер будет блокироваться на продолжительный период времени, несмотря на выполнение единственного HTTP-запроса. Чтобы обойти эту ситуацию, необходимо оформлять загрузку нескольких сценариев так, чтобы не блокировать броузер.

Секрет неблокирующих сценариев заключается в том, что они загружаются после окончания загрузки страницы. Говоря техническим языком, это означает, что программный код загружается уже после возбуждения события `load` в объекте `window`. Достигнуть этого результата можно несколькими способами.

Отложенные сценарии

Спецификация HTML 4 определяет дополнительный атрибут для тега `<script>` с именем `defer`. Атрибут `defer` указывает, что сценарий, содержащийся внутри элемента, не изменяет дерево DOM, и потому его выполнение можно отложить до самого последнего момента. Атрибут `defer` поддерживается только в броузерах Internet Explorer 4+ и Firefox 3.5+, что лишает это решение универсальности. Другие броузеры просто игнорируют атрибут `defer` и обрабатывают тег `<script>` обычным (блокирующим) способом. Тем не менее это решение будет полезно там, где целевые броузеры поддерживают его. Ниже приводится пример использования атрибута:

```
<script type="text/javascript" src="file1.js" defer></script>
```

Тег `<script>` с атрибутом `defer` может находиться в любом месте внутри документа. Загрузка JavaScript-файла, на который он ссылается, начнется в момент, когда броузер встретит тег `<script>`, но выполнение программного кода будет отложено, пока дерево DOM не будет загружено полностью (перед вызовом обработчика события `onload`). На время загрузки отложенного JavaScript-файла другие процессы броузера не будут блокироваться, благодаря чему такие файлы могут загружаться параллельно другому содержимому страницы.

Ни один сценарий в элементах `<script>`, помеченных атрибутом `defer`, не будет выполнен, пока дерево DOM не будет загружено полностью. Это относится и к встроенным, и к внешним сценариям. Следующий при-

мер простой страницы демонстрирует, как атрибут `defer` изменяет поведение сценариев:

```
<html>
<head>
  <title>Script Defer Example</title>
</head>
<body>
  <script defer>
    alert("defer");
  </script>
  <script>
    alert("script");
  </script>
  <script>
    window.onload = function(){
      alert("load");
    };
  </script>
</body>
</html>
```

В процессе обработки этой страницы выводится три диалога. В броузерах, не поддерживающих атрибут `defer`, диалоги выводятся в таком порядке: «defer», «script» и «load». В броузерах, поддерживающих атрибут `defer`, диалоги выводятся в ином порядке: «script», «defer» и «load». Обратите внимание, что сценарий в элементе `<script>` с атрибутом `defer` выполняется после второго сценария, но перед тем как будет вызван обработчик события `onload`.

Если в число целевых броузеров входят только Internet Explorer и Firefox 3.5, то использование отложенных сценариев может оказаться полезным. Если необходимо обеспечить поддержку более широкого круга броузеров, следует использовать другое, более универсальное решение.[1]

Динамические элементы script

Объектная модель документа (Document Object Model, DOM) позволяет динамически создавать из JavaScript-сценариев практически любые части HTML-документа. В своей основе элемент `<script>` ничем не отличается от других элементов: ссылки на них можно получить из дерева DOM, они могут перемещаться в документе с места на место, удаляться из него и даже добавляться. Новый элемент `<script>` легко можно создать с помощью стандартных методов модели DOM:

---

[1] В Internet Explorer 8 (стандартный режим) и Firefox 3.6 было реализовано изменившееся в HTML5 определение атрибута `defer`. В обоих случаях встроенный программный код в элементах `<script>` с атрибутом `defer` теперь игнорируется и требуется использовать атрибут `src`. Таким образом, предыдущий пример будет функционировать не во всех броузерах. При выполнении под управлением Internet Explorer 8 в режиме совместимости будет проявляться устаревшее поведение.

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

Этот новый элемент <script> загрузит файл *file1.js*. Загрузка начнется сразу, как только элемент будет добавлен в страницу. Важно отметить, что при использовании этого приема загрузка и выполнение сценария не будут блокировать другие процессы независимо от того, в каком месте документа была инициирована загрузка. Этот программный код можно даже поместить в тег <head> документа, не опасаясь, что он окажет влияние на загрузку остальной части документа (кроме единственного HTTP-соединения, которое будет создано для загрузки файла).

В общем случае безопаснее добавлять новые узлы <script> в элемент <head>, а не в элемент <body>, особенно если такой программный код выполняется в ходе загрузки страницы. Броузер Internet Explorer может генерировать ошибку «operation aborted» («операция прервана»), если содержимое элемента <body> не было загружено полностью.[1]

Когда файл загружается посредством динамически созданного узла <script>, полученный программный код обычно выполняется немедленно (исключение составляют Firefox и Opera, которые откладывают выполнение, пока не будут выполнены сценарии в предыдущих динамически созданных узлах <script>). Этот прием прекрасно работает в случае с автономными сценариями, но может вызывать проблемы, если загруженный программный код содержит только интерфейсы для использования в других сценариях. В этом случае необходимо следить за тем, когда сценарий будет полностью загружен и будет готов для использования. Реализовать такое слежение можно с помощью события, генерируемого динамическим узлом <script>.

Броузеры Firefox, Opera, Chrome и Safari 3+ генерируют событие load, когда элемент <script> с атрибутом src завершает загрузку сценария. Установив обработчик этого события, можно определить, когда сценарий будет готов к использованию:

```
var script = document.createElement("script")
script.type = "text/javascript";

// Firefox, Opera, Chrome, Safari 3+
script.onload = function(){
    alert("Script loaded!");
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

[1] Более подробное обсуждение этой проблемы дается в статье «The dreaded operation aborted error» по адресу: *http://www.nczonline.net/blog/2008/03/17/the-dreaded-operation-aborted-error/*.

Internet Explorer поддерживает альтернативную реализацию, которая возбуждает событие `readystatechange`. Элемент `<script>` в этом броузере имеет свойство `readyState`, значение которого изменяется на разных этапах загрузки внешнего файла. Свойство `readyState` может иметь пять различных значений:

`"uninitialized"`

Состояние по умолчанию.

`"loading"`

Загрузка началась.

`"loaded"`

Загрузка завершилась.

`"interactive"`

Данные полностью загружены, но пока не готовы к использованию.

`"complete"`

Все данные готовы к использованию.

В документации Microsoft с описанием свойства `readyState` и каждого из возможных его значений указывается, что не все состояния будут обязательно пройдены свойством на протяжении жизненного цикла элемента `<script>`, но при этом ничего не говорится о том, какие из них будут использоваться обязательно. На практике наибольший интерес представляют значения `"loaded"` и `"complete"`. Броузер Internet Explorer ведет себя непоследовательно в том, какое из этих двух значений свойства `readyState` указывает на конечное состояние – иногда элемент `<script>` может достичь состояния `"loaded"` и никогда не перейти в состояние `"complete"`, а иногда он может перейти в состояние `"complete"`, пропустив состояние `"loaded"`. Безопаснее всего проверять оба эти значения при обработке события `readystatechange` и удалять обработчик по достижении одного из этих состояний (чтобы гарантировать, что событие не будет обработано дважды):

```
var script = document.createElement("script")
script.type = "text/javascript";

// Internet Explorer
script.onreadystatechange = function(){
    if (script.readyState == "loaded" || script.readyState == "complete"){
        script.onreadystatechange = null;
        alert("Script loaded.");
    }
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

В большинстве случаев бывает необходимо использовать единый подход к динамической загрузке JavaScript-файлов. Следующая функция реализует оба варианта загрузки – стандартный и характерный для IE:

```
function loadScript(url, callback){

    var script = document.createElement("script")
    script.type = "text/javascript";

    if (script.readyState){ // IE
        script.onreadystatechange = function(){
            if (script.readyState=="loaded" || script.readyState=="complete"){
                script.onreadystatechange = null;
                callback();
            }
        };
    } else { // Другие броузеры
        script.onload = function(){
            callback();
        };
    }

    script.src = url;
    document.getElementsByTagName("head")[0].appendChild(script);
}
```

Эта функция принимает два аргумента: URL-адрес JavaScript-файла и функцию обратного вызова, которую следует вызвать, когда JavaScript-файл будет полностью загружен. С помощью приема определения поддерживаемых особенностей функция выясняет, какое событие следует использовать для мониторинга состояния элемента `<script>`. В заключение она присваивает значение свойству `src` и добавляет элемент `<script>` в страницу. Ниже демонстрируется пример использования функции `loadScript()`:

```
loadScript("file1.js", function(){
    alert("File is loaded!");
});
```

В одной странице можно динамически загружать любое количество JavaScript-файлов, но имейте в виду, что загружаться они могут в разном порядке. Из всех броузеров только Firefox и Opera гарантируют, что сценарии будут выполняться в указанном вами порядке. Другие броузеры будут загружать и выполнять файлы в том порядке, в каком они будут возвращаться сервером. Определенный порядок выполнения можно гарантировать, если объединить операции загрузки в цепочку, например:

```
loadScript("file1.js", function(){
    loadScript("file2.js", function(){
        loadScript("file3.js", function(){
            alert("All files are loaded!");
        });
    });
});
```

Этот пример откладывает загрузку файла *file2.js*, пока не будет загружен файл *file1.js*, и точно так же откладывает загрузку файла *file3.js*,

пока не будет загружен файл *file2.js*. Однако реализация такого подхода может оказаться немного сложнее, когда потребуется организовать загрузку и выполнение большого количества файлов.

Если порядок загрузки большого количества файлов имеет значение, предпочтительнее объединить их в один файл, где они будут располагаться в требуемом порядке. После этого весь необходимый программный код можно будет получить, загрузив этот единый файл (а поскольку динамическая загрузка выполняется асинхронно, размер файла не оказывает влияния на отзывчивость интерфейса пользователя).

Динамическая загрузка сценариев является наиболее часто используемым приемом, позволяющим избежать блокирования благодаря его простоте и совместимости с различными броузерами.

Инъекция сценариев с помощью XMLHttpRequest

Другой способ реализовать неблокирующую загрузку сценариев заключается в получении сценария с помощью объекта XMLHttpRequest (XHR) и последующей вставке его в страницу. Этот прием связан с созданием объекта XMLHttpRequest, загрузкой JavaScript-файла и вставкой полученного программного кода в динамически созданный элемент `<script>`. Ниже приводится простой пример:

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){
            var script = document.createElement("script");
            script.type = "text/javascript";
            script.text = xhr.responseText;
            document.body.appendChild(script);
        }
    }
};
xhr.send(null);
```

Этот фрагмент отправляет GET-запрос на получение файла *file1.js*. Обработчик события `onreadystatechange` ожидает, пока свойство `readyState` получит значение 4, и затем проверяет HTTP-код состояния (любое значение в диапазоне от 200 до 299 свидетельствует об успехе, а значение 304 означает, что ответ был извлечен из кэша). В случае успеха создается новый элемент `<script>`, и его свойству `text` присваивается значение свойства `responseText`, полученное от сервера. При таком подходе фактически создается элемент `<script>` со встроенным сценарием. Как только новый элемент `<script>` будет добавлен в документ, содержащийся в нем программный код будет выполнен и станет готов к использованию.

Основное преимущество этого подхода заключается в возможности загружать JavaScript-сценарии без немедленного их выполнения. Поскольку возвращаемый сервером программный код находится за пределами

тега <script>, он не будет выполнен автоматически после загрузки, что позволяет отложить его выполнение до требуемого момента. Другое преимущество состоит в том, что этот подход реализуется одинаково во всех современных броузерах без исключений.

Основное ограничение этого подхода в том, что JavaScript-файл должен находиться в том же домене, что и страница, запрашивающая его, что делает невозможным загрузку с применением служб CDN. По этой причине прием инъекции сценариев с применением XHR редко используется в крупных веб-приложениях.

Рекомендуемые способы неблокирующей загрузки

Загрузку больших объемов программного кода на языке JavaScript рекомендуется выполнять в два этапа: сначала подключить программный код, необходимый для динамической загрузки, а затем загрузить остальные сценарии, требуемые для инициализации страницы. Поскольку на первом этапе загружается совсем небольшой объем программного кода (фактически достаточно загрузить одну только функцию loadScript()), загрузка и выполнение будут протекать очень быстро и не будут оказывать существенного влияния на общую скорость загрузки страницы. После того как на первом этапе будет получен необходимый программный код, его можно будет использовать для загрузки остальных JavaScript-сценариев. Например:

```
<script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
    loadScript("the-rest.js", function(){
        Application.init();
    });
</script>
```

Поместите этот программный код непосредственно перед закрывающим тегом </body>. Такой подход несет в себе несколько преимуществ. Во-первых, как уже говорилось выше, он гарантирует, что выполнение сценария не будет препятствовать отображению остальной части страницы. Во-вторых, к концу загрузки второго JavaScript-файла дерево DOM будет создано и готово к применению, что позволит избавиться от необходимости использовать другие обработчики событий (такие как window.onload) с целью определить момент, когда страница будет готова для инициализации.

Кроме того, функцию loadScript() можно встроить непосредственно в страницу и тем самым избавиться от лишнего HTTP-запроса. Например:

```
<script type="text/javascript">
    function loadScript(url, callback){

        var script = document.createElement("script")
        script.type = "text/javascript";
```

```
            if (script.readyState){ // IE
                script.onreadystatechange = function(){
                    if (script.readyState == "loaded" ||
                            script.readyState == "complete"){
                        script.onreadystatechange = null;
                        callback();
                    }
                };
            } else { // Другие броузеры
                script.onload = function(){
                    callback();
                };
            }

            script.src = url;
            document.getElementsByTagName("head")[0].appendChild(script);
        }

        loadScript("the-rest.js", function(){
            Application.init();
        });
    </script>
```

При выборе последнего подхода рекомендуется минимизировать начальный сценарий с помощью инструмента, такого как YUI Compressor (глава 9), чтобы уменьшить размер страницы.

После загрузки всех сценариев, необходимых для инициализации страницы, можно использовать функцию `loadScript()` для загрузки в страницу дополнительных функциональных возможностей по мере необходимости.

Подход с использованием YUI 3

Концепция загрузки небольшого начального объема программного кода с последующей загрузкой дополнительной функциональности положена в основу библиотеки YUI 3. Чтобы задействовать YUI 3 в своей странице, в первую очередь необходимо подключить начальный файл YUI:

```
<script type="text/javascript"
  src="http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js"></script>
```

Начальный файл имеет размер около 10 Кбайт (6 Кбайт в виде архива в формате gzip) и включает все необходимое для загрузки других компонентов библиотеки YUI из службы Yahoo! CDN. Например, если потребуются утилиты для работы с деревом DOM, нужно передать методу `use()` из библиотеки YUI имя требуемого модуля (`"dom"`) и функцию, которая будет вызвана, как только загруженный программный код будет готов к использованию:

```
YUI().use("dom", function(Y){
    Y.DOM.addClass(document.body, "loaded");
});
```

Этот пример создает новый экземпляр объекта YUI и вызывает метод use(). Начальный файл содержит всю информацию об именах файлов и зависимостях, поэтому при использовании имени "dom" метод фактически сконструирует специализированный URL-адрес, добавив в него имена всех зависимых файлов, и динамически создаст элемент <script>, который загрузит и выполнит эти файлы. По завершении загрузки программного кода будет выполнена функция обратного вызова, которая получит экземпляр объекта YUI в виде аргумента, что позволит ей немедленно начать использовать новые функциональные возможности.

Библиотека LazyLoad

Райан Гров (Ryan Grove) из Yahoo! Search создал еще более универсальный инструмент – библиотеку LazyLoad (доступна по адресу *http://github.com/rgrove/lazyload/*). Библиотека LazyLoad представляет собой более мощную версию функции loadScript(). В минимизированном состоянии файл библиотеки LazyLoad имеет размер около 1,5 Кбайт (в минимизированном, но не сжатом состоянии). Ниже приводится пример использования этой библиотеки:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js("the-rest.js", function(){
        Application.init();
    });
</script>
```

Библиотека LazyLoad способна также загружать сразу несколько JavaScript-файлов и гарантирует их выполнение в требуемом порядке во всех броузерах. Чтобы загрузить несколько JavaScript-файлов, достаточно просто передать методу LazyLoad.js() массив URL-адресов:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js(["first-file.js", "the-rest.js"], function(){
        Application.init();
    });
</script>
```

Даже при том что прием динамической загрузки сценариев не блокирует отображение страницы, рекомендуется минимизировать количество загружаемых файлов, насколько это возможно. Для загрузки каждого файла требуется выполнить отдельный HTTP-запрос, и функции обратного вызова не будут выполнены, пока не будут загружены и выполнены все файлы.

Библиотека LazyLoad способна также выполнять динамическую загрузку CSS-файлов. Однако обычно в этом нет необходимости, потому что загрузка CSS-файлов всегда выполняется параллельно и не блокирует работу страницы.

Библиотека LABjs

Еще один способ реализовать неблокирующую загрузку сценариев предоставляет свободно распространяемая библиотека LABjs (*http://labjs.com/*), созданная Кайлом Симпсоном (Kyle Simpson) при участии Стива Содерса (Steve Souders). Эта библиотека обеспечивает возможность более тонкого управления процессом загрузки и пытается максимально распараллелить загрузку файлов. Кроме того, библиотека LABjs имеет весьма небольшой размер – 4,5 Кбайт (в минимизированном, но не сжатом состоянии), что обеспечивает минимальное влияние на загрузку страницы. Ниже приводится пример ее использования:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Метод `$LAB.script()` используется с целью определить загружаемый JavaScript-файл, а метод `$LAB.wait()` – чтобы указать функцию, которая должна быть вызвана после загрузки и выполнения файла. Библиотека LABjs поддерживает возможность объединения вызовов методов в цепочки, то есть каждый метод возвращает ссылку на объект `$LAB`. Чтобы загрузить несколько JavaScript-файлов, достаточно объединить в цепочку несколько вызовов метода `$LAB.script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js")
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Отличительной особенностью библиотеки LABjs является ее способность управлять зависимостями. При обычном подключении сценариев с помощью тегов `<script>` они загружаются (последовательно или параллельно, как описывалось выше) и затем последовательно выполняются. В одних случаях это действительно бывает необходимо, а в других – нет.

Библиотека LABjs позволяет с помощью метода `wait()` определить порядок выполнения файлов. В предыдущем примере сценарий в файле *first-file.js* не обязательно будет выполнен перед сценарием в файле *the-rest.js*. Чтобы гарантировать это, следует добавить вызов метода `wait()` после первого вызова метода `script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js").wait()
```

```
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Теперь сценарий в файле *first-file.js* гарантированно будет выполнен перед сценарием в файле *the-rest.js*, однако содержимое файлов будет загружаться параллельно.

В заключение

Управление JavaScript-сценариями в броузерах – не самая простая задача, потому что выполнение программного кода блокирует другие процессы, протекающие в броузере, такие как отображение пользовательского интерфейса. Каждый раз когда броузер встречает тег `<script>`, обработка страницы приостанавливается на время, необходимое для загрузки (если тег ссылается на внешний файл) и выполнения программного кода, после чего обработка страницы продолжается. Однако есть несколько способов уменьшить отрицательное влияние операций загрузки и выполнения JavaScript-сценариев:

- Поместить все теги `<script>` в конец страницы непосредственно перед закрывающим тегом `</body>`. В этом случае, когда дело дойдет до запуска сценариев, страница будет отображена практически полностью.
- Сгруппировать сценарии вместе. Чем меньше тегов `<script>` будет в странице, тем быстрее она будет загружена и станет доступна для взаимодействий с пользователем. Это относится и к встроенным, и к внешним JavaScript-сценариям.
- Имеется несколько способов реализовать загрузку JavaScript-сценариев, не блокируя страницу:
  - Использовать атрибут `defer` в теге `<script>` (только в Internet Explorer и Firefox 3.5+).
  - Динамически создавать элементы `<script>` для загрузки и выполнения сценариев.
  - Загружать JavaScript-сценарии с помощью объекта XHR и вставлять их в страницу.

Используя эти стратегии, можно существенно улучшить субъективное восприятие скорости загрузки веб-приложений с большими объемами программного кода на языке JavaScript.

2

Доступ к данным

Одной из классических задач информатики является определение места хранения данных, обеспечивающего оптимальное чтение и запись. От места хранения данных зависит скорость их извлечения во время выполнения сценария. В языке JavaScript эта задача упрощается небольшим количеством возможных вариантов хранения данных. Тем не менее, как и в других языках программирования, выбор места хранения данных может существенно влиять на скорость доступа к ним. В языке JavaScript имеется четыре основных места, где могут храниться данные:

Литеральные значения

Любое значение, представляющее само себя и не хранящееся в каком-то определенном месте. В языке JavaScript в виде литералов могут быть представлены строки, числа, логические значения, объекты, массивы, функции, регулярные выражения и специальные значения `null` и `undefined`.

Переменные

Любое место хранения данных, определенное разработчиком с помощью ключевого слова `var`.

Элементы массивов

Места хранения данных внутри объекта `Array`, доступ к которым осуществляется с помощью числовых индексов.

Члены объектов

Места хранения данных внутри объекта, доступ к которым осуществляется с помощью строковых индексов.

Каждое из этих мест хранения имеет определенную стоимость операций чтения и записи данных с позиций оценки производительности. В большинстве случаев разница в скорости доступа к данным в литералах и в локальных переменных весьма невелика. Операции доступа к эле-

ментам массивов и членам объектов являются более дорогостоящими, при этом дороговизна этих операций сильно зависит от броузера. На рис. 2.1 показана относительная скорость доступа к 200 000 значений в каждом из перечисленных мест в разных броузерах.

Более старые броузеры, имеющие более традиционные реализации JavaScript, такие как Firefox 3, Internet Explorer и Safari 3.2, тратят больше времени на доступ к значениям по сравнению с броузерами, имеющими оптимизированные реализации JavaScript. Однако все броузеры демонстрируют общую закономерность: доступ к литеральным значениям и локальным переменным выполняется быстрее, чем к элементам массивов и членам объектов. Единственное исключение составляет Firefox 3, в котором благодаря оптимизации доступ к элементам массивов выполняется намного быстрее. Но даже в этом случае рекомендуется везде, где скорость выполнения имеет большое значение, максимально использовать литералы и локальные переменные и ограничивать использование элементов массивов и членов объектов. Для решения этой задачи имеется несколько шаблонов, помогающих оптимизировать программный код.

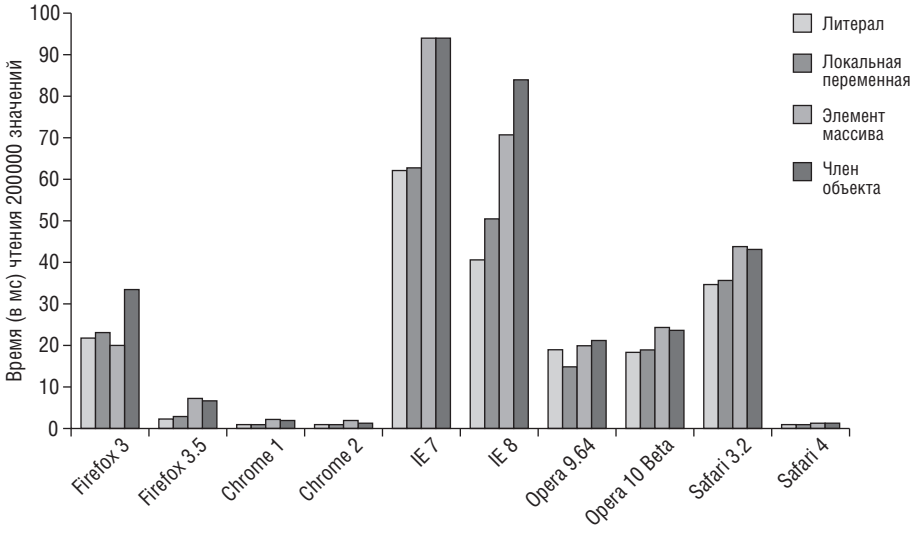


Рис. 2.1. *Время чтения 200000 значений, хранящихся в разных местах*

Управление областью видимости

Концепция области видимости является ключом к пониманию языка JavaScript не только с точки зрения производительности, но и с точки зрения функциональности. Области видимости в языке JavaScript влияют на самые разные аспекты, от определения перечня переменных, доступных функциям, до значения ключевого слова `this`. Области

видимости в JavaScript оказывают определенное влияние на производительность, но чтобы понять, как скорость выполнения связана с областями видимости, необходимо разобраться с особенностями работы областей видимости.

Цепочки областей видимости и разрешение идентификаторов

Каждая функция в языке JavaScript представлена объектом, точнее экземпляром объекта `Function`. Как и любые другие объекты, объекты функций имеют свойства, включая свойства, доступные программному коду, а также внутренние свойства, используемые интерпретатором JavaScript и недоступные программам. Одним из таких свойств является свойство `[[Scope]]`, определяемое третьей редакцией стандарта ECMA-262, (*http://www.ecma-international.org/publications/standards/Ecma-262.htm*).

Внутреннее свойство `[[Scope]]` содержит коллекцию объектов, представляющую область видимости, в которой была создана функция. Эта коллекция называется цепочкой областей видимости функции и определяет, какие данные будут доступны функции. Каждый объект в цепочке областей видимости функции называется *объектом переменных*, и каждый из них содержит пары ключ/значение, представляющие переменные. При создании функции ее цепочка областей видимости заполняется объектами, представляющими данные, доступные в той области видимости, где была создана функция. Например, рассмотрим следующую глобальную функцию:

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

При создании функции `add()` в ее цепочку областей видимости будет включен единственный объект с переменными: глобальный объект, представляющий все глобальные переменные. Этот глобальный объект содержит, например, такие переменные, как `window`, `navigator` и `document`. Взаимосвязь между функциями и объектами переменных показана на рис. 2.2 (при этом на рисунке в глобальном объекте показано лишь несколько глобальных переменных; в действительности их намного больше).

Рис. 2.2. *Цепочка областей видимости для функции add()*

Цепочка областей видимости функции `add` будет использоваться во время выполнения функции. Допустим, что выполняется следующая инструкция:

```
var total = add(5, 10);
```

При вызове функции `add` создается внутренний объект, который называется *контекстом выполнения*. Контекст выполнения определяет окружение, в котором выполняется функция. Каждый контекст выполнения является уникальным для данного конкретного вызова функции, то есть при многократных вызовах одной и той же функции создается множество контекстов выполнения. Объект контекста выполнения уничтожается, как только завершится выполнение функции.

Контекст выполнения имеет собственную цепочку областей видимости, которая используется для разрешения идентификаторов. При создании контекста выполнения его цепочка областей видимости инициализируется объектами, содержащимися в свойстве `[[Scope]]` выполняемой функции. Они копируются в цепочку областей видимости контекста выполнения в порядке их появления в функции. После выполнения этой операции для контекста выполнения создается новый объект, который называется *объектом активации*. Объект активации играет роль объекта переменных для данного вызова функции и содержит все локальные переменные, именованные аргументы, коллекцию `arguments` и ссылку `this`. Затем данный объект помещается в начало цепочки областей видимости. При уничтожении контекста выполнения уничтожается и объект активации. На рис. 2.3 показаны контекст выполнения и его цепочка областей видимости для предыдущего примера вызова функции.

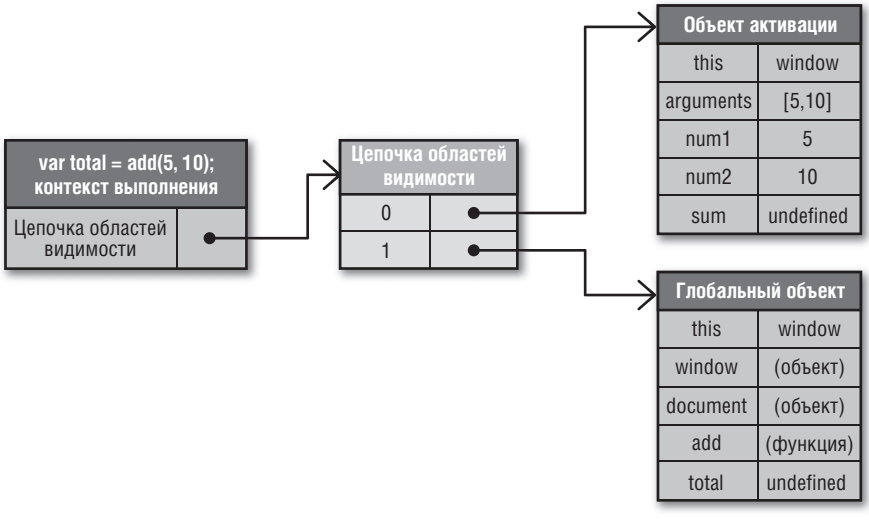


Рис. 2.3. *Цепочка областей видимости при выполнении функции add()*

Каждый раз когда при выполнении функции интерпретатор встречает ссылку на переменную, он выполняет процедуру разрешения идентификатора, чтобы определить, откуда извлекать или где сохранять данные. В ходе этой процедуры выполняется поиск требуемого идентификатора в цепочке областей видимости. Поиск начинается с начала цепочки в объекте активации контекста выполнения. Если поиск увенчается успехом, будет использована найденная переменная; в противном случае поиск продолжится в следующем объекте, входящем в цепочку областей видимости. Поиск продолжается, пока не будет найдена переменная с требуемым идентификатором или пока в цепочке не останется объектов переменных. В последнем случае считается, что идентификатор не был объявлен. Эта процедура выполняется для каждого идентификатора, который будет встречен в ходе выполнения функции, то есть для предыдущего примера эта процедура будет выполнена для идентификаторов `sum`, `num1` и `num2`. Именно эта процедура поиска сказывается на производительности.

Обратите внимание, что в различных частях в цепочке областей видимости могут существовать две переменные с одинаковыми именами. В этом случае будет использоваться переменная, найденная первой при обходе цепочки областей видимости, и говорят, что первая переменная *заслоняет* вторую.

Производительность разрешения идентификаторов

Разрешение идентификаторов – не бесплатная операция; как и любая другая операция, выполняемая компьютером, она сказывается на производительности. Чем глубже в цепочке областей видимости оказывается идентификатор, тем медленнее выполняются операции чтения и записи с его участием. Следовательно, внутри функций локальные переменные обеспечивают самый быстрый доступ к данным, тогда как доступ к глобальным переменным выполняется немного медленнее (оптимизирующие интерпретаторы JavaScript способны выправлять этот недостаток в некоторых ситуациях). Имейте в виду, что глобальные переменные всегда располагаются в последнем объекте переменных в цепочке областей видимости контекста выполнения, то есть они всегда оказываются самыми далекими для процедуры разрешения идентификаторов. На рис. 2.4 и 2.5 показана скорость разрешения идентификаторов в зависимости от глубины их нахождения в цепочке областей видимости. Глубина 1 соответствует локальной переменной.

Для всех броузеров наблюдается общая закономерность – чем глубже в цепочке областей видимости находится искомый идентификатор, тем медленнее будут выполняться операции чтения и записи с его участием. В броузерах с оптимизирующими интерпретаторами JavaScript, таких как Chrome и Safari 4, падение производительности с увеличением глубины вложенности идентификаторов практически не наблюдается,

тогда как в Internet Explorer, Safari 3.2 и в других броузерах наблюдается более существенное влияние. Следует отметить, что в более ранних версиях броузеров, таких как Internet Explorer 6 и Firefox 2, кривые графиков были более крутыми, и их верхние точки оказались бы за пределами представленных здесь диаграмм, если бы эти данные были включены.

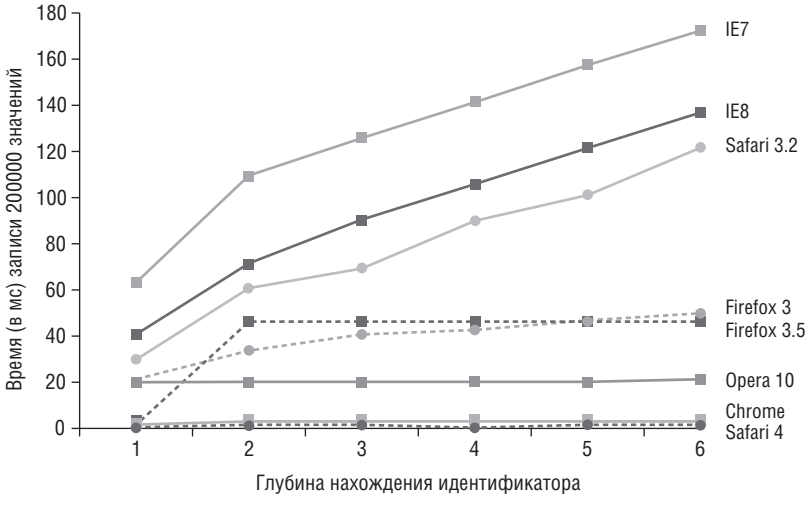


Рис. 2.4. *Разрешение идентификатора для операции записи*

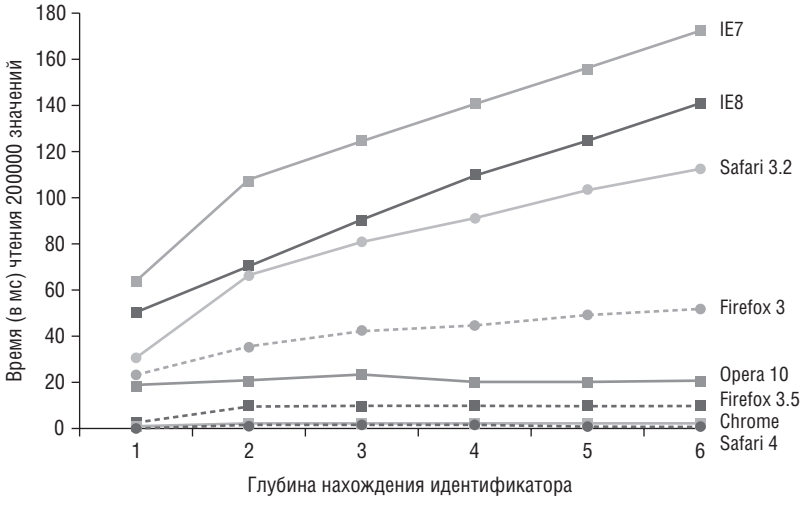


Рис. 2.5. *Разрешение идентификатора для операции чтения*

На основе полученной информации можно рекомендовать максимально использовать локальные переменные, чтобы обеспечить высокую производительность в броузерах, не оснащенных оптимизирующими интерпретаторами JavaScript. Неплохо было бы взять за правило сохранять внешние значения в локальных переменных, если они используются в функции более одного раза. Например:

```
function initUI(){
    var bd = document.body,
        links = document.getElementsByTagName("a"),
        i = 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }

    document.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

Эта функция трижды ссылается на идентификатор `document`, который представляет глобальный объект. Процедура поиска вынуждена будет пройти через всю цепочку областей видимости, прежде чем отыщет его в объекте глобальных переменных. Можно смягчить отрицательное влияние на производительность повторных обращений к глобальной переменной, сохранив при первом обращении ссылку на нее в локальной переменной и далее используя локальную переменную вместо глобальной. Например, предыдущую функцию можно переписать, как показано ниже:

```
function initUI(){
    var doc = document,
        bd = doc.body,
        links = doc.getElementsByTagName("a"),
        i = 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }

    doc.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

Измененная версия функции initUI() сначала сохраняет ссылку на объект document в локальной переменной doc. Теперь количество обращений к глобальной переменной уменьшилось с трех до одного. Доступ к переменной doc выполняется быстрее, чем к переменной document, потому что она является локальной переменной. Разумеется, эта простая функция не покажет значительного увеличения производительности, потому что в ней оптимизировано небольшое количество обращений, но представьте огромные функции с десятками обращений к глобальным переменным, повторяющихся многократно – в таких функциях прирост производительности может оказаться более внушительным.

Увеличение цепочки областей видимости

Вообще говоря, цепочка областей видимости контекста выполнения не изменяется. Однако имеются две инструкции, которые временно увеличивают цепочку областей видимости контекста выполнения в процессе выполнения. Первой из них является инструкция with.

Инструкция with применяется для создания переменных для всех свойств объекта. Она имитирует аналогичную функциональность, имеющуюся в других языках программирования, и выглядит как удобство, позволяющее избежать необходимости многократно писать один и тот же программный код. Например, функцию initUI() можно было бы переписать так:

```
function initUI(){
    with (document){ // избегайте!
        var bd = body,
            links = getElementsByTagName("a"),
            i = 0,
            len = links.length;

        while(i < len){
            update(links[i++]);
        }

        getElementById("go-btn").onclick = function(){
            start();
        };

        bd.className = "active";
    }
}
```

В этой версии функции initUI() используется инструкция with, что позволяет избежать необходимости писать идентификатор document. Хотя такая реализация может показаться более эффективной, в действительности она только ухудшает производительность.

Когда поток выполнения достигает инструкции with, цепочка областей видимости контекста выполнения временно расширяется. Создается

новый объект переменных, содержащий все свойства указанного объекта. Затем этот объект вставляется в начало цепочки, и все локальные переменные функции оказываются во втором объекте в цепочке областей видимости, из-за чего снижается скорость доступа к ним (рис. 2.6).

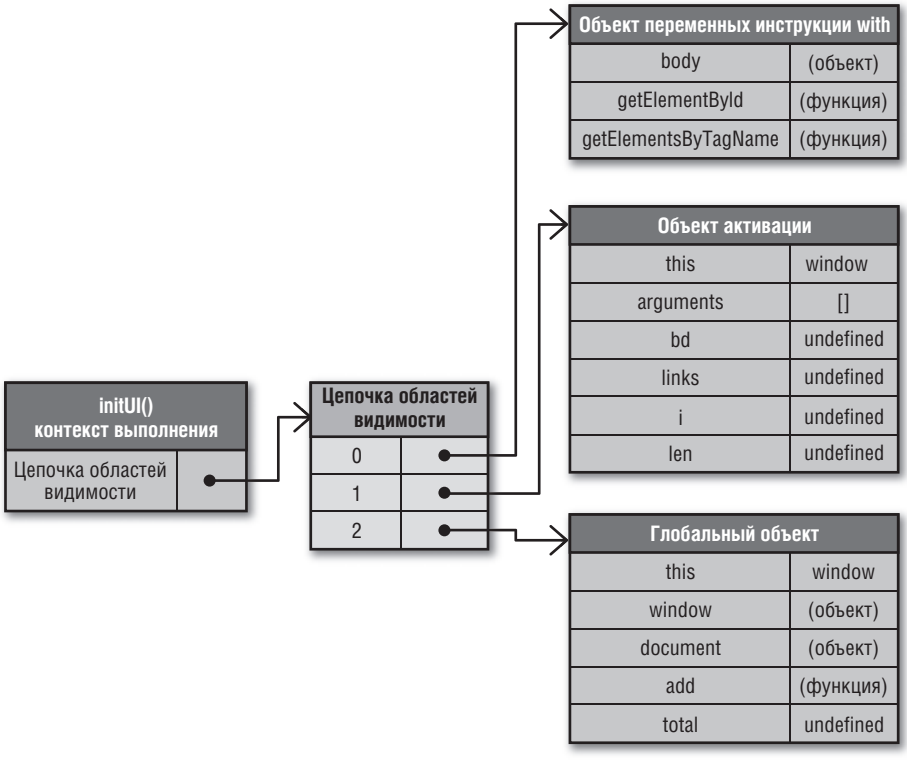


Рис. 2.6. *Расширение цепочки областей видимости в инструкции with*

При передаче объекта `document` инструкции `with` создается новый объект переменных, содержащий все свойства объекта `document`, и он вставляется в начало цепочки областей видимости. Это значительно ускоряет доступ к свойствам объекта `document`, но замедляет доступ к локальным переменным, таким как `bd`. По этой причине лучше стараться не использовать инструкцию `with`. Как было показано выше, ссылку на объект `document` легко можно сохранить в локальной переменной и тем самым получить прирост производительности.

Инструкция `with` не единственная в языке JavaScript, расширяющая цепочку областей видимости контекста выполнения; тем же эффектом обладает предложение `catch` в инструкции `try-catch`. Когда в блоке `try` возникает ошибка, управление немедленно передается в блок `catch`, а в начало цепочки областей видимости вставляется новый объект перемен-

ных, в котором сохраняется объект исключения. Внутри блока `catch` все локальные переменные функции оказываются во втором объекте в цепочке областей видимости. Например:

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    alert(ex.message); // здесь цепочка областей видимости расширена
}
```

Обратите внимание, что по завершении выполнения блока `catch` цепочка областей видимости вернется в прежнее состояние.

Инструкция `try-catch` весьма удобна при правильном применении, и поэтому бессмысленно рекомендовать полностью избегать ее использования. Любой, кто собирается использовать инструкцию `try-catch`, должен представлять себе вероятность появления ошибки. Инструкция `try-catch` никогда не должна применяться как средство против ошибок JavaScript. Если известно, что ошибки будут возникать часто, это говорит о том, что выбран неправильный алгоритм работы и его следует изменить.

Отрицательное влияние предложения `catch` на производительность можно уменьшить, сократив до минимума объем программного кода внутри него. Отличное решение заключается в том, чтобы реализовать метод обработки ошибок, который можно было бы вызывать из предложения `catch`, например:

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    handleError(ex); // передать обработку методу
}
```

Здесь вызов метода `handleError()` является единственной инструкцией, выполняемой внутри предложения `catch`. Этот метод получает объект исключения, сгенерированный в результате ошибки, и может обрабатывать ошибку соответствующим способом. Поскольку здесь выполняется всего одна инструкция и отсутствуют обращения к локальным переменным, временное расширение цепочки областей видимости не оказывает влияния на производительность этого программного кода.

Динамические области видимости

Инструкция `with`, предложение `catch` в инструкции `try-catch`, а также функция, содержащая вызов функции `eval()`, обладают *динамическими областями видимости*. Динамические области видимости существуют только во время выполнения программного кода и потому не обнаруживаются при простом статическом анализе (при просмотре структуры программного кода). Например:

```
function execute(code) {
    eval(code);

    function subroutine(){
        return window;
    }

    var w = subroutine();

    // какое значение получит переменная w?
};
```

Функция `execute()` имеет динамическую область видимости, потому что содержит вызов функции `eval()`. Значение переменной `w` может изменяться в зависимости от значения `code`. В большинстве случаев переменная `w` будет ссылаться на глобальный объект `window`, но взгляните на следующую строку:

```
execute("var window = {};")
```

В этом случае вызов функции `eval()` создаст внутри функции `execute()` локальную переменную `window`, и переменной `w` будет присвоена ссылка не на глобальную, а на локальную переменную `window`. Нет никакой возможности узнать, как в действительности будут обстоять дела, пока не будет выполнен программный код, а это означает, что значение идентификатора `window` нельзя предсказать заранее.

Оптимизирующие реализации интерпретаторов JavaScript, такие как Nitro в броузере Safari, пытаются ускорить разрешение идентификаторов, анализируя программный код, чтобы определить, к каким переменным выполняется доступ в каждый момент. Эти реализации стараются избежать традиционной последовательности поиска в цепочках видимости, индексируя идентификаторы для ускорения их разрешения. Однако с появлением динамической области видимости эта оптимизация становится невозможной. Реализации вынуждены вернуться к использованию более медленного алгоритма разрешения идентификаторов, основанного на использовании хешей, который ближе к традиционному алгоритму поиска в цепочках областей видимости.

По этой причине рекомендуется использовать динамические области видимости, только когда это действительно необходимо.

Замыкания, области видимости и память

Замыкания, являющиеся одной из самых мощных особенностей языка JavaScript, позволяют функциям обращаться к данным за пределами локальной области видимости. Порядок применения замыканий популярно описывается в книгах Дугласа Крокфорда (Douglas Crockford), и в настоящее время без них не обходится ни одно сложное веб-приложение. Однако замыкания также оказывают влияние на производительность.

Чтобы разобраться с влиянием замыканий на производительность, рассмотрим следующий фрагмент:

```
function assignEvents(){

    var id = "xdi9592";

    document.getElementById("save-btn").onclick = function(event){
        saveDocument(id);
    };
}
```

Функция `assignEvents()` присваивает обработчик события единственному элементу в дереве DOM. Этот обработчик события образует замыкание, так как создается во время выполнения функции `assignEvents()`, и получает доступ к переменной `id`, находящейся в объемлющей области видимости. Чтобы обеспечить доступ к переменной `id` из этого замыкания, интерпретатор должен создать соответствующую область видимости.

При вызове функции `assignEvents()` создается объект активации, содержащий помимо всего прочего переменную `id`. Он становится первым объектом в цепочке областей видимости контекста выполнения; вторым за ним следует глобальный объект. Когда создается замыкание, его свойство `[[Scope]]` инициализируется обоими этими объектами (рис. 2.7).

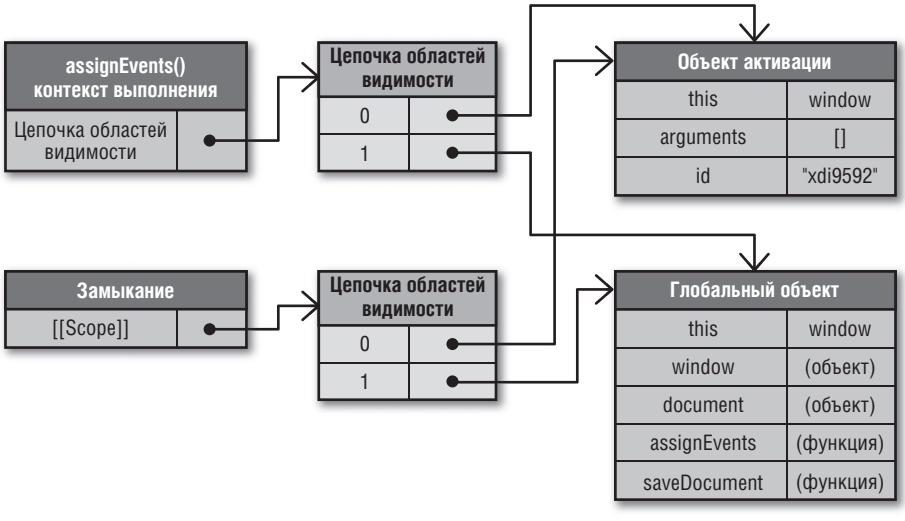


Рис. 2.7. *Цепочки областей видимости контекста выполнения функции assignEvents() и замыкания*

Поскольку свойство `[[Scope]]` замыкания содержит ссылки на те же объекты, что и цепочка областей видимости контекста выполнения, это ведет к появлению побочного эффекта. Обычно объект активации функ-

ции уничтожается одновременно с уничтожением контекста выполнения. Однако при наличии замыкания объект активации не уничтожается, потому что ссылки продолжают существовать внутри свойства `[[Scope]]` замыкания. Это означает, что замыкания потребляют больше памяти, чем функции, не образующие замыканий. В больших веб-приложениях это может превратиться в проблему, особенно при выполнении в Internet Explorer. В IE объекты DOM реализованы не как обычные JavaScript-объекты, вследствие чего использование замыканий может вызывать утечки памяти (подробнее об этом рассказывается в главе 3).

Когда выполняется замыкание, создается контекст выполнения, цепочка областей видимости которого инициализируется теми же двумя объектами цепочки областей видимости, на которые ссылается свойство `[[Scope]]`, а затем создается новый объект активации для самого замыкания (рис. 2.8).

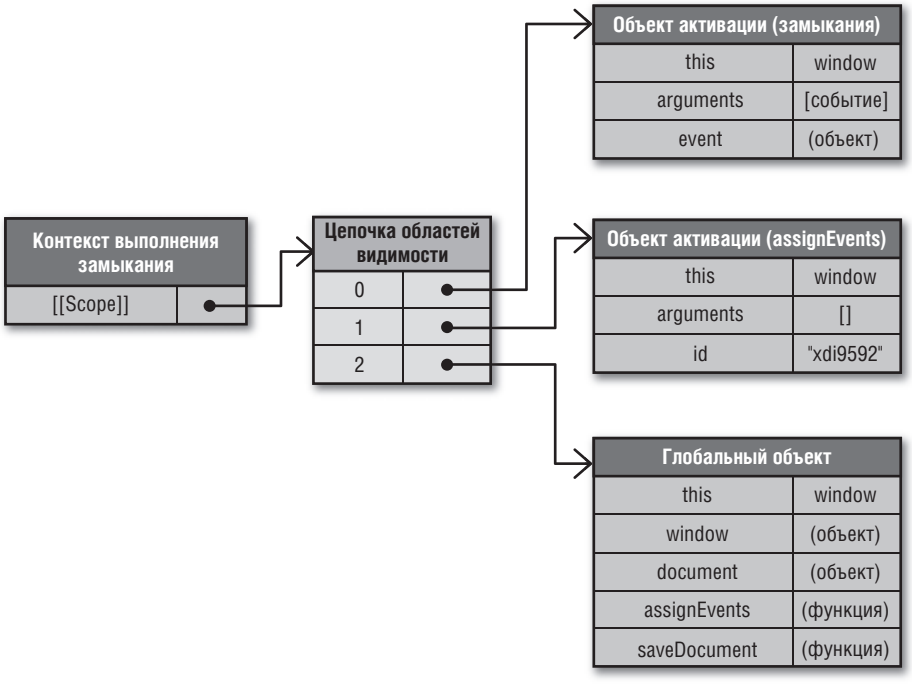


Рис. 2.8. *Выполнение замыкания*

Обратите внимание, что оба идентификатора, `id` и `saveDocument`, используемые в замыкании, идут после первого объекта в цепочке областей видимости. Это является основным фактором снижения производительности в замыканиях: нередко приходится обращаться к большому количеству идентификаторов за пределами основной области видимости,

и каждое такое обращение отрицательно сказывается на производительности.

К замыканиям следует относиться с осторожностью, так как их использование ведет к увеличению потребления памяти и снижению производительности. Однако есть возможность смягчить отрицательное влияние на производительность, если следовать совету, данному выше в этой главе, касающемуся глобальных переменных: сохраняйте ссылки на часто используемые внешние переменные в локальных переменных и используйте локальные переменные.

Члены объектов

Подавляющая часть программного кода на JavaScript пишется в объектно-ориентированном стиле, в котором либо создаются собственные, либо используются встроенные объекты, такие как объекты модели DOM или модели BOM (Browser Object Model – объектная модель броузера). В силу этого в сценариях очень часто осуществляется доступ к членам объектов.

Членами объектов являются и свойства, и методы, которые в языке JavaScript совсем незначительно отличаются друг от друга. Члены объектов могут содержать данные любых типов. Поскольку функции в JavaScript представлены объектами, в дополнение к данным традиционных типов члены объектов могут содержать и функции. Член объекта, ссылающийся на функцию, называют методом, а член объекта, ссылающийся на данные любого другого типа, – свойством.

Как говорилось выше в этой главе, доступ к членам объектов обычно выполняется медленнее, чем доступ к данным в литералах или переменных, и в некоторых броузерах медленнее, чем доступ к элементам массивов. Чтобы понять, почему так происходит, необходимо разобраться в природе объектов в языке JavaScript.

Прототипы

Объекты в языке JavaScript основаны на *прототипах*. Прототип – это объект, который служит основой для других объектов, определяющей и реализующей члены, которые должен иметь новый объект. Эта концепция совершенно не похожа на концепцию классов в традиционных объектно-ориентированных языках программирования, определяющих процесс создания новых объектов. Объекты-прототипы совместно используются всеми экземплярами данного типа объектов, вследствие чего все экземпляры совместно используют члены объекта-прототипа.

Объект хранит ссылку на свой прототип во внутреннем свойстве. В броузерах Firefox, Safari и Chrome это свойство доступно разработчикам под именем `__proto__`; в других броузерах доступ к этому свойству закрыт. Всякий раз, когда создается новый экземпляр встроенного типа,

отличного от `Object`, такого как `Date` или `Array`, он автоматически становится экземпляром типа `Object` – своего прототипа.

Следовательно, объекты могут иметь члены двух типов: члены экземпляра (также называются «собственными» членами) и члены прототипа. Члены экземпляра существуют непосредственно в экземпляре объекта, а члены прототипа наследуются от объекта-прототипа. Рассмотрим следующий пример:

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.toString()); // "[object Object]"
```

Объект `book` в этом примере имеет два члена экземпляра: `title` и `publisher`. Обратите внимание, что здесь отсутствует определение метода `toString()`, но он вызывается и действует, как ожидается, не генерируя ошибку. Метод `toString()` – это член прототипа, наследуемого объектом `book`. Взаимосвязь объектов показана на рис. 2.9.

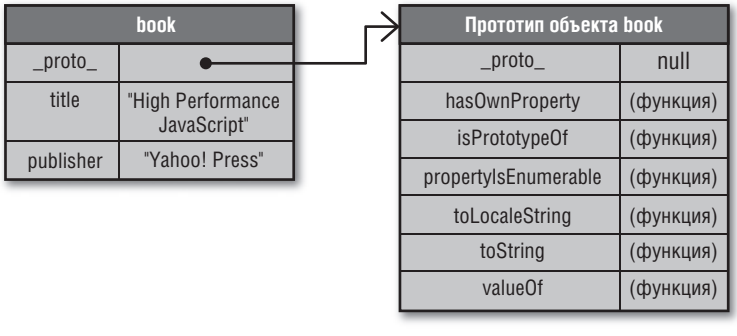


Рис. 2.9. *Взаимосвязь между экземпляром и прототипом*

Процесс разрешения членов объектов очень похож на разрешение переменных. Когда в программном коде встречается вызов `book.toString()`, интерпретатор начинает поиск члена с именем «toString» в экземпляре объекта. Поскольку объект `book` не имеет члена с именем `toString`, поиск продолжится в объекте-прототипе, где будет найден и вызван метод `toString()`. Таким образом, объект `book` обладает доступом ко всем свойствам и методам своего прототипа.

Определить наличие в объекте члена экземпляра с требуемым именем можно с помощью метода `hasOwnProperty()`, принимающего имя члена. Чтобы определить, обладает ли объект доступом к свойству с некоторым именем, можно воспользоваться оператором `in`. Например:

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.hasOwnProperty("title"));    // true
alert(book.hasOwnProperty("toString")); // false

alert("title" in book);    // true
alert("toString" in book); // true
```

В этом фрагменте, когда метод `hasOwnProperty()` получает имя «title», он возвращает `true`, потому что `title` является свойством экземпляра; когда этот метод получает имя «toString», он возвращает `false`, потому что данный член отсутствует в экземпляре. Когда эти же имена свойств участвуют в выражениях с оператором `in`, в обоих случаях возвращается `true`, потому что этот оператор выполняет поиск и в экземпляре, и в прототипе.

Цепочки прототипов

Прототип объекта определяет тип или типы экземпляров этого объекта. По умолчанию все объекты являются экземплярами объекта `Object` и наследуют все основные методы, такие как `toString()`. Имеется возможность создать прототип другого типа, для чего необходимо определить и задействовать конструктор. Например:

```
function Book(title, publisher){
    this.title = title;
    this.publisher = publisher;
}

Book.prototype.sayTitle = function(){
    alert(this.title);
};

var book1 = new Book("High Performance JavaScript", "Yahoo! Press");
var book2 = new Book("JavaScript: The Good Parts", "Yahoo! Press");

alert(book1 instanceof Book);   // true
alert(book1 instanceof Object); // true

book1.sayTitle();        // "High Performance JavaScript"
alert(book1.toString()); // "[object Object]"
```

Здесь для создания экземпляров объекта `Book` используется конструктор `Book`. Прототипом (`__proto__`) экземпляра `book1` является `Book.prototype`, а прототипом объекта `Book.prototype` – объект `Object`. В результате такой реализации образуется цепочка прототипов, от которых объекты `book1` и `book2` наследуют свои члены. Отношения между этими объектами и их прототипами показаны на рис. 2.10.

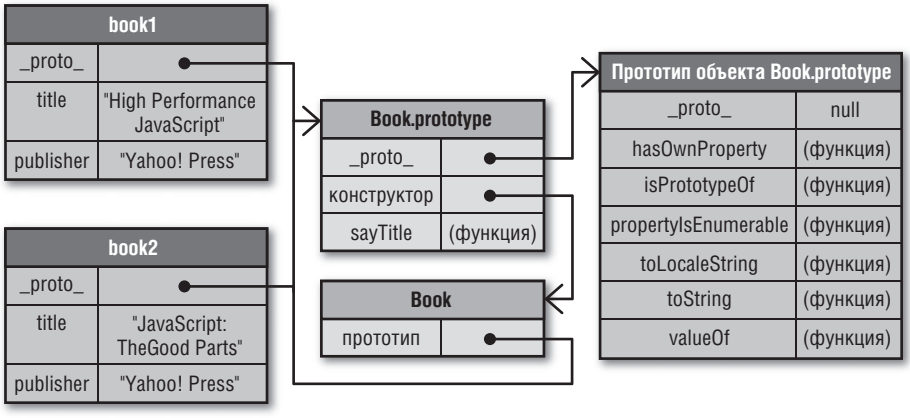


Рис. 2.10. *Цепочка прототипов*

Обратите внимание, что оба экземпляра объекта `Book` используют одну и ту же цепочку прототипов. Каждый экземпляр имеет собственные свойства `title` и `publisher`, а все остальные наследует от прототипов.

Теперь, встретив вызов `book1.toString()`, интерпретатор должен углубиться в цепочку прототипов еще больше, чтобы отыскать член «toString» объекта. Без труда можно догадаться, что чем глубже в цепочке прототипов находится искомый член, тем медленнее происходит доступ к нему. На рис. 2.11 показана зависимость времени доступа к члену от глубины его нахождения в цепочке прототипов.

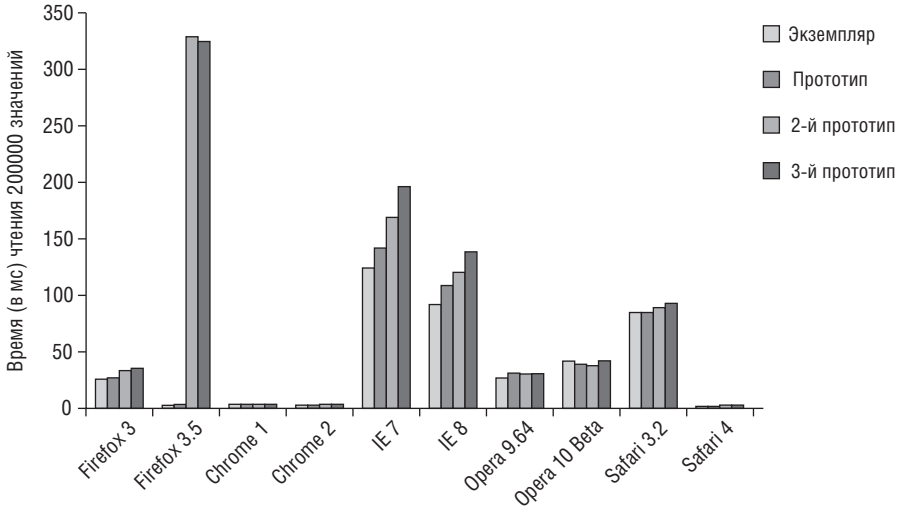


Рис. 2.11. *Время доступа к данным с различной глубиной вложенности в цепочку прототипов*

Новейшие броузеры, оснащенные оптимизирующими интерпретаторами JavaScript, прекрасно справляются с поставленной задачей, однако более старые броузеры – особенно Internet Explorer и Firefox 3.5 – показывают значительное снижение производительности с каждым новым уровнем в цепочке прототипов. Следует понимать, что поиск членов экземпляров в любом случае выполняется медленнее, чем обращение к данным в литералах или локальных переменных, а необходимость выполнять поиск в цепочке прототипов только ухудшает ситуацию.

Вложенные члены

Поскольку члены объектов могут содержать другие члены, в JavaScript-сценариях нередко можно встретить, например, такое обращение: `window.location.href`. Обращение к вложенным членам вынуждает интерпретатор JavaScript выполнять процедуру разрешения членов объектов при встрече каждой точки в выражении. На рис. 2.12 показана зависимость времени доступа к члену от глубины его вложенности.

Нет ничего удивительного, что доступ к членам осуществляется тем медленнее, чем глубже они вложены. Разрешение имени `location.href` всегда выполняется быстрее, чем разрешение имени `window.location.href`, которое, в свою очередь, выполняется быстрее, чем разрешение имени `window.location.href.toString()`. Кроме того, разрешение имен свойств, не являющихся членами экземпляров, выполняется еще дольше из-за необходимости искать их в цепочке прототипов.

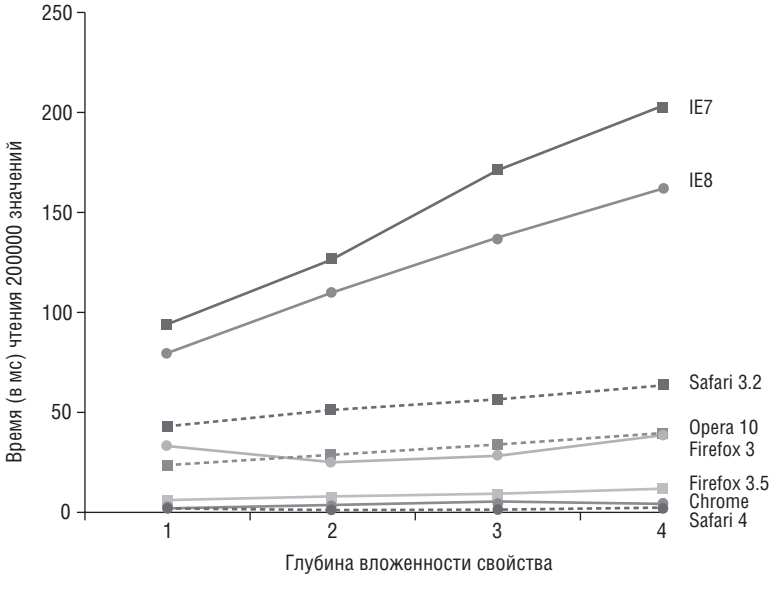


***Рис. 2.12.** Зависимость времени доступа от глубины вложенности свойства*

В большинстве броузеров отсутствуют существенные различия в скорости доступа к членам объектов между способами с применением точечной нотации (`object.name`) и с использованием квадратных скобок (`object["name"]`). Safari – единственный броузер, в котором точечная нотация обеспечивает более высокую скорость, но не настолько, чтобы ради этого стоило отказаться от формы записи с квадратными скобками.

Кэширование значений членов объектов

Познакомившись с перечисленными проблемами производительности операций доступа к членам объектов, легко прийти к выводу, что их следует избегать, насколько это возможно. Точнее, обращаться к членам объектов следует, только когда это действительно необходимо. Например, в следующем примере нет никакой необходимости многократно читать значение члена объекта в функции:

```
function hasEitherClass(element, className1, className2){
    return element.className == className1 || element.className == className2;
}
```

В данном примере функция дважды обращается к свойству `element.className`. Очевидно, что данный программный код дважды выполнит процедуру поиска члена объекта, хотя его значение не изменится в промежутке между обращениями. Есть возможность уменьшить количество операций поиска, сохранив значение в локальной переменной и используя ее вместо свойства:

```
function hasEitherClass(element, className1, className2){
    var currentClassName = element.className;
    return currentClassName == className1 || currentClassName == className2;
}
```

Эта новая версия функции уменьшает количество операций поиска члена объекта до одной. Поскольку цель обеих операций поиска – чтение значения свойства, есть смысл прочитать значение один раз и сохранить его в локальной переменной. Доступ к этой локальной переменной будет осуществляться намного быстрее.

Вообще говоря, если чтение свойства объекта выполняется в функции более одного раза, лучше сохранить значение этого свойства в локальной переменной. После этого вместо свойства можно использовать локальную переменную и избавиться от лишних операций поиска, отрицательно сказывающихся на производительности. Это особенно важно при работе с вложенными членами объектов, доступ к которым особенно сильно снижает скорость выполнения.

Использование пространств имен, как, например, в библиотеке YUI, приводит к частому обращению к вложенным свойствам. Например:

```
function toggle(element){
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
        return false;
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
        return true;
    }
}
```

В этом примере трижды выполняется разрешение имени `YAHOO.util.Dom` для вызова трех разных методов. Для каждого метода трижды выполняется поиск членов объектов. В итоге операция поиска выполняется девять раз, что делает эту функцию весьма неэффективной. Более эффективное решение заключается в том, чтобы сохранить ссылку на `YAHOO.util.Dom` в локальной переменной и затем использовать ее:

```
function toggle(element){
    var Dom = YAHOO.util.Dom;
    if (Dom.hasClass(element, "selected")){
        Dom.removeClass(element, "selected");
        return false;
    } else {
        Dom.addClass(element, "selected");
        return true;
    }
}
```

Общее число операций поиска членов объектов здесь уменьшилось до пяти. Никогда не следует внутри функции искать член объекта более одного раза, если его значение не изменяется между обращениями.

Будьте внимательны: этот прием не рекомендуется использовать для методов объектов. Многие методы объектов используют ссылку `this` для определения контекста вызова, а сохранение ссылки на метод в локальной переменной автоматически свяжет ссылку `this` с объектом `window`. Изменение значения `this` приведет к появлению ошибок, так как интерпретатор JavaScript не сможет отыскать соответствующие члены объектов, которые могут использоваться внутри методов.

В заключение

Место хранения данных, используемых в JavaScript-сценарии, может оказывать заметное влияние на общую производительность программного кода. Существует четыре места, где могут храниться используемые данные: литералы, переменные, элементы массивов и члены объектов. Все они имеют различную скорость доступа.

- Доступ к литералам и локальным переменным выполняется очень быстро, тогда как доступ к элементам массивов и членам объектов занимает больше времени.
- Доступ к локальным переменным выполняется быстрее, чем доступ к внешним переменным, потому что в цепочке областей видимости они находятся в первом объекте переменных. Чем дальше в цепочке находится переменная, тем больше времени требуется для обращения к ней. Доступ к глобальным переменным всегда выполняется медленнее, потому что они всегда находятся в самом конце цепочки областей видимости.
- Избегайте использования инструкции `with`, потому что она удлиняет цепочку областей видимости контекста выполнения. Проявляйте также осторожность при использовании предложения `catch` инструкции `try-catch`, потому что оно имеет тот же эффект.
- Обращения к вложенным членам объектов оказывают существенное влияние на производительность, и их количество следует уменьшить до предела.
- Чем глубже в цепочке прототипов находится свойство или метод, тем медленнее осуществляется доступ к нему.
- Вообще говоря, производительность JavaScript-сценариев можно повысить, сохранив часто используемые члены объектов, элементы массивов и внешние переменные в локальных переменных. Доступ к локальным переменным осуществляется быстрее, чем к оригиналам.

Используя эти принципы, можно существенно увеличить воспринимаемую производительность веб-приложения, для реализации которого требуется большой объем программного кода.

3

Работа с деревом DOM

Стоян Стефанов (Stoyan Stefanov)

Операции с деревом DOM довольно дорогостоящие с позиций влияния на производительность и обычно являются самым узким местом мощных веб-приложений. В этой главе рассматриваются операции с деревом DOM, которые могут оказывать отрицательное влияние на отзывчивость приложений, и даются рекомендации по улучшению времени отклика. Здесь обсуждаются три категории часто решаемых задач:

- Доступ к элементам дерева DOM и их модификация
- Модификации стилей элементов дерева DOM, вызывающие их перерисовку и перекомпоновку
- Обработка действий пользователя посредством событий модели DOM

Но сначала познакомимся с моделью DOM и выясним, почему операции с деревом элементов выполняются так медленно.

Модель DOM в мире броузеров

Объектная модель документа (Document Object Model, DOM) – это независимый от языка прикладной интерфейс (API) для работы с XML- и HTML-документами. В броузерах чаще всего обрабатываются HTML-документы, но при этом веб-приложения нередко получают XML-документы и используют DOM API для доступа к данным, хранящимся в них.

Хотя объектная модель документа определяет API, не зависимый от языка, в броузерах этот прикладной интерфейс реализован на языке JavaScript. Поскольку основной задачей клиентских сценариев является выполнение операций с документом, составляющим основу веб-стра-

ницы, знание модели DOM является важной составляющей успеха при повседневном программировании на языке JavaScript.

Броузеры обычно отделяют друг от друга реализации модели DOM и самого языка JavaScript. В Internet Explorer, например, реализация JavaScript называется JScript и находится в файле библиотеки с именем *jscript.dll*, тогда как реализация модели DOM находится в другой библиотеке, *mshtml.dll* (которая имеет внутреннее название Trident). Такое отделение позволяет другим технологиям и языкам, таким как VBScript, использовать реализацию модели DOM и возможности механизма отображения Trident. В Safari используются реализация модели DOM с названием WebCore и механизм отображения, входящие в состав пакета WebKit, и отдельная реализация интерпретатора с названием JavaScriptCore (в последней версии получила название SquirrelFish). В броузере Google Chrome для отображения страниц также используются библиотеки WebCore из пакета WebKit, но в нем реализована собственная версия интерпретатора JavaScript с названием V8. В Firefox используется реализация JavaScript с названием SpiderMonkey (в последней версии получила название TraceMonkey), являющаяся составной частью механизма отображения Gecko.

Врожденная медлительность

Что все это означает с точки зрения производительности? Само наличие двух отдельных реализаций, взаимодействующих друг с другом, всегда будет сказываться на производительности. Можно провести отличную аналогию, представив реализации DOM и JavaScript (точнее ECMAScript) как два острова, связанные мостом с платным проездом (Джон Хрватин (John Hrvatin), Microsoft, MIX09, *http://videos.visitmix.com/MIX09/T53F*). Каждый раз, когда интерпретатору ECMAScript требуется обратиться к модели DOM, необходимо проехать по мосту и заплатить потерей производительности. Чем больше выполняется операций с деревом DOM, тем больше приходится платить. Поэтому в общем случае можно порекомендовать пересекать этот мост как можно реже и стараться не покидать остров ECMAScript. В оставшейся части главы все внимание будет сконцентрировано на том, что означает эта рекомендация в действительности и что можно сделать, чтобы ускорить операции с пользовательским интерфейсом.

Доступ к дереву DOM и его модификация

Даже простой доступ к элементам DOM имеет определенную цену – «плату за проезд», о которой говорилось выше. А модификация элементов стоит еще дороже, потому что зачастую вызывает необходимость пересчета геометрии страницы броузером.

Естественно, худшим случаем является доступ или модификация элементов в цикле, особенно когда цикл выполняется по коллекции HTML-элементов.

Чтобы получить представление о масштабе проблем, возникающих при работе с деревом DOM, рассмотрим следующий простой пример:

```
function innerHTMLLoop() {
    for (var count = 0; count < 15000; count++) {
        document.getElementById('here').innerHTML += 'a';
    }
}
```

Эта функция изменяет содержимое элемента страницы в цикле. Проблема здесь заключается в том, что в каждой итерации цикла производится два обращения к элементу: при первом обращении читается содержимое свойства `innerHTML`, а при втором выполняется запись в него.

Чтобы повысить эффективность этой функции, можно было бы хранить изменяемое содержимое в локальной переменной и выполнять запись полученного значения только один раз в конце цикла:

```
function innerHTMLLoop2() {
    var content = '';
    for (var count = 0; count < 15000; count++) {
        content += 'a';
    }
    document.getElementById('here').innerHTML += content;
}
```

Эта новая версия будет выполняться намного быстрее во всех броузерах. На рис. 3.1 показаны результаты измерения прироста производительности в различных броузерах. Ось Y на этом рисунке (как и на всех

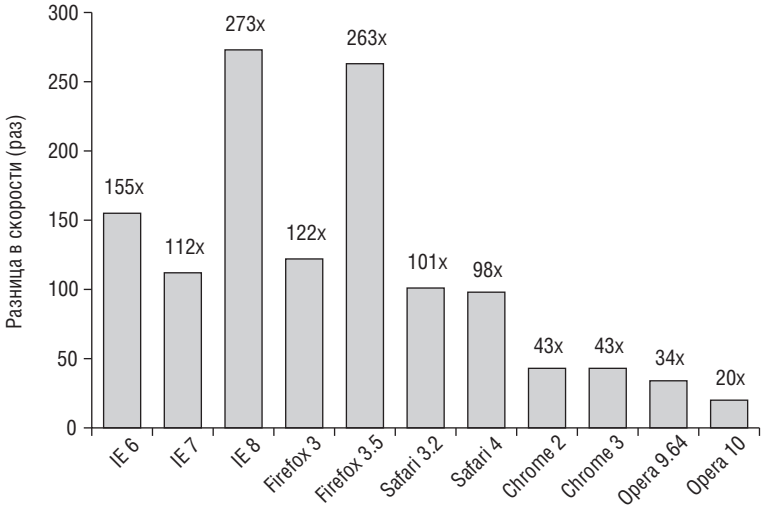


Рис. 3.1. *Одно из преимуществ, которые дает пребывание на территории ECMAScript: функция innerHTMLLoop2() в сотни раз быстрее аналогичной ей функции innerHTMLLoop()*

рисунках в данной главе) определяет прирост производительности, то есть насколько один подход оказался быстрее другого. В данном случае функция `innerHTMLLoop2()` в IE6 оказалась в 155 раз быстрее функции `innerHTMLLoop()`.

Эти результаты наглядно показывают, что увеличение количества обращений к дереву DOM влечет за собой снижение скорости выполнения программного кода. Основной вывод, который следует из этого: касайтесь дерева DOM лишь слегка и оставайтесь на территории ECMAScript как можно дольше.

Свойство innerHTML в сравнении с методами DOM

Долгие годы в сообществе веб-разработчиков не утихали споры по поводу вопроса: что лучше, использовать для изменения содержимого страницы нестандартное, но широко поддерживаемое свойство `innerHTML`, или стандартные методы модели DOM, такие как `document.createElement()`? Имеет ли это какое-нибудь значение для производительности, если не касаться веб-стандартов? Да, имеет, но все меньше и меньше, и тем не менее свойство `innerHTML` действует быстрее во всех броузерах, кроме последних версий, основанных на WebKit (Chrome и Safari).

Рассмотрим пример создания таблицы из 1000 строк двумя способами:

- Посредством конкатенации строк с разметкой HTML и обновления свойства `innerHTML` элемента DOM.
- Только с помощью стандартных методов модели DOM, таких как `document.createElement()` и `document.createTextNode()`.

Содержимое этой таблицы напоминает содержимое, которое может быть получено из системы управления содержимым (Content Management System, CMS). Конечный результат представлен на рис. 3.2.

id	yes?	name	url	action
1	And the answer is... yes	my name is #1	http://example.org/1.html	• edit • delete
2	And the answer is... no	my name is #2	http://example.org/2.html	• edit • delete

Рис. 3.2. *Конечный результат создания HTML-таблицы с 1000 строками и 5 столбцами*

Следующий программный код создает таблицу с помощью свойства `innerHTML`:

```
function tableInnerHTML() {
    var i, h = ['<table border="1" width="100%">'];
```

```
    h.push('<thead>');

    h.push('<tr><th>id<\/th><th>yes?<\/th><th>name<\/th><th>url<\/th>'+
           '<th>action<\/th><\/tr>');
    h.push('<\/thead>');
    h.push('<tbody>');
    for (i = 1; i <= 1000; i++) {
        h.push('<tr><td>');
        h.push(i);
        h.push('<\/td><td>');
        h.push('And the answer is... ' + (i % 2 ? 'yes' : 'no'));
        h.push('<\/td><td>');
        h.push('my name is #' + i);
        h.push('<\/td><td>');
        h.push('<a href="http://example.org/' + i +
               '.html">http://example.org/' + i + '.html<\/a>');
        h.push('<\/td><td>');
        h.push('<ul>');
        h.push(' <li><a href="edit.php?id=' + i + '">edit<\/a><\/li>');
        h.push(' <li><a href="delete.php?id="' + i + '">delete<\/a><\/li>');
        h.push('<\/ul>');
        h.push('<\/td>');
        h.push('<\/tr>');
    }

    h.push('<\/tbody>');
    h.push('<\/table>');

    document.getElementById('here').innerHTML = h.join('');
};
```

Реализация, создающая ту же таблицу с помощью методов модели DOM, получилась более объемной:

```
function tableDOM() {

    var i, table, thead, tbody, tr, th, td, a, ul, li;

    tbody = document.createElement('tbody');

    for (i = 1; i <= 1000; i++) {

        tr = document.createElement('tr');
        td = document.createElement('td');
        td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
        tr.appendChild(td);
        td = document.createElement('td');
        td.appendChild(document.createTextNode(i));
        tr.appendChild(td);
        td = document.createElement('td');
        td.appendChild(document.createTextNode('my name is #' + i));
        tr.appendChild(td);
```

```
        a = document.createElement('a');
        a.setAttribute('href', 'http://example.org/' + i + '.html');
        a.appendChild(document.createTextNode('http://example.org/' + i +
                      '.html'));
        td = document.createElement('td');
        td.appendChild(a);
        tr.appendChild(td);

        ul = document.createElement('ul');
        a = document.createElement('a');
        a.setAttribute('href', 'edit.php?id=' + i);
        a.appendChild(document.createTextNode('edit'));
        li = document.createElement('li');
        li.appendChild(a);
        ul.appendChild(li);
        a = document.createElement('a');
        a.setAttribute('href', 'delete.php?id=' + i);
        a.appendChild(document.createTextNode('delete'));
        li = document.createElement('li');
        li.appendChild(a);
        ul.appendChild(li);
        td = document.createElement('td');
        td.appendChild(ul);
        tr.appendChild(td);

        tbody.appendChild(tr);
    }

    tr = document.createElement('tr');
    th = document.createElement('th');
    th.appendChild(document.createTextNode('yes?'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('id'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('name'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('url'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('action'));
    tr.appendChild(th);

    thead = document.createElement('thead');
    thead.appendChild(tr);
    table = document.createElement('table');
    table.setAttribute('border', 1);
    table.setAttribute('width', '100%');
    table.appendChild(thead);
```

```
    table.appendChild(tbody);

    document.getElementById('here').appendChild(table);
};
```

Результаты сравнения скорости создания HTML-таблицы с использованием свойства `innerHTML` и с помощью методов модели DOM показаны на рис. 3.3. Преимущество свойства `innerHTML` более очевидно в старых версиях броузеров (в IE6 реализация на основе свойства `innerHTML` оказалась в 3,6 раза быстрее) и менее заметно в более новых версиях. А в новейших броузерах, реализованных на основе пакета WebKit, результаты даже получились противоположными: методы модели DOM оказались немного быстрее. Таким образом, выбор того или иного способа будет зависеть от того, какие броузеры используют ваши пользователи, и от ваших личных предпочтений.

В большинстве броузеров использование свойства `innerHTML` позволит повысить скорость выполнения критичных операций, производящих обновление значительной части HTML-страницы. Но в настоящее время в большинстве случаев прирост скорости будет не слишком большой, и поэтому при выборе решения следует подумать о таких факторах, как удобочитаемость, простота сопровождения, коллективные предпочтения и соглашения по оформлению программного кода.

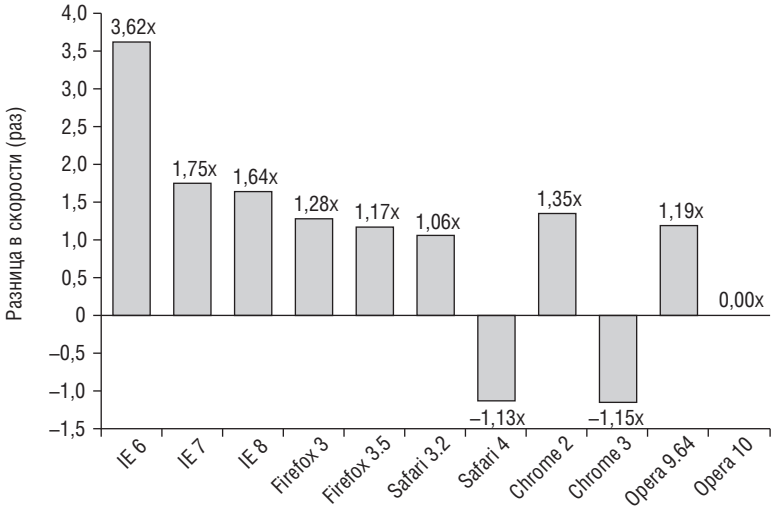


Рис. 3.3. *Преимущество свойства innerHTML перед методами модели DOM при создании таблицы из 1000 строк; в IE6 свойство innerHTML оказывается более чем в три раза быстрее, а в последних броузерах на основе WebKit немного медленнее*

Копирование узлов

Другой способ изменения содержимого страницы с использованием методов модели DOM заключается в копировании существующих DOM-элементов вместо создания новых. Иными словами, – в использовании метода `element.cloneNode()` (где `element` является существующим узлом) вместо метода `document.createElement()`.

В большинстве броузеров операция копирования более эффективна, хотя и незначительно. Реализация создания таблицы из предыдущего примера посредством однократного создания повторяющихся элементов и последующего их копирования дает небольшое увеличение скорости выполнения:

- 2% в IE8, но не изменяется в IE6 и IE7
- До 5,5% в Firefox 3.5 и Safari 4
- 6% в Opera (но никакого увеличения в Opera 10)
- 10% в Chrome 2 и 3% в Chrome 3

Для иллюстрации ниже приводится фрагмент программного кода, генерирующего таблицу с помощью метода `element.cloneNode()`:

```
function tableClonedDOM() {

    var i, table, thead, tbody, tr, th, td, a, ul, li,
        oth = document.createElement('th'),
        otd = document.createElement('td'),
        otr = document.createElement('tr'),
        oa  = document.createElement('a'),
        oli = document.createElement('li'),
        oul = document.createElement('ul');

    tbody = document.createElement('tbody');

    for (i = 1; i <= 1000; i++) {

        tr = otr.cloneNode(false);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
        tr.appendChild(td);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode(i));
        tr.appendChild(td);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode('my name is #' + i));
        tr.appendChild(td);

        // ... остальная часть тела цикла ...

    }

    // ... остальная часть функции создания таблицы ...

}
```

HTML-коллекции

HTML-коллекции – это объекты, подобные массивам, содержащие ссылки на узлы в дереве DOM. Примерами коллекций являются значения, возвращаемые следующими методами:

- `document.getElementsByName()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

Следующие свойства также возвращают HTML-коллекции:

`document.images`

Все элементы `<img>` в странице.

`document.links`

Все элементы `<a>` и `<area>` с атрибутом `<href>`.

`document.forms`

Все формы.

`document.forms[0].elements`

Все поля из первой формы в странице.

Эти методы и свойства возвращают объекты `HTMLCollection`, являющиеся подобными массивам списками. Они не являются настоящими массивами (потому что не имеют методов, таких как `push()` или `slice()`), но подобно массивам имеют свойство `length` и позволяют обращаться к элементам списка по индексам. Например, выражение `document.images[1]` вернет второй элемент коллекции. Как определено стандартом DOM, HTML-коллекции являются «*живыми*, в том смысле что должны автоматически обновляться с изменением содержимого документа» (*http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-75708506*).

HTML-коллекции фактически являются запросами к документу, которые повторно выполняются каждый раз, когда требуется получить самую последнюю информацию, такую как количество элементов в коллекции (то есть значение свойства `length` коллекции). Эта их особенность может приводить к снижению эффективности.

Дорогие коллекции

Чтобы убедиться, что коллекции являются «живыми», рассмотрим следующий фрагмент:

```
// непреднамеренно бесконечный цикл
var alldivs = document.getElementsByTagName('div');
for (var i = 0; i < alldivs.length; i++) {
    document.body.appendChild(document.createElement('div'))
}
```

На первый взгляд этот программный код просто удваивает количество элементов `<div>` в странице. Он обходит в цикле имеющиеся элементы

 и создает новые элементы , добавляя их в конец тела страницы. Но в действительности получился бесконечный цикл, потому что условие выхода из цикла, alldivs.length, увеличивается на единицу с каждой итерацией, отражая текущее состояние документа, лежащего в основе страницы.

Обход в цикле HTML-коллекций подобным образом может не только приводить к логическим ошибкам, но и отрицательно сказываться на производительности из-за того, что в каждой итерации необходимо будет выполнять запрос (рис. 3.4).

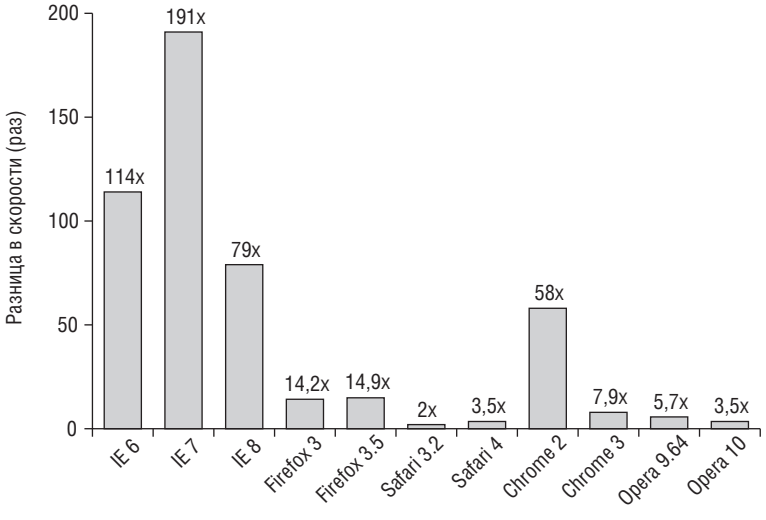


***Рис. 3.4.** Обход массива выполняется существенно быстрее, чем обход HTML-коллекции того же размера и с тем же содержимым*

Как обсуждается в главе 4, не рекомендуется непосредственно обращаться к свойству length массива в условном выражении цикла. Скорость обращения к свойству length коллекции еще ниже, чем к свойству length обычного массива, потому что каждое такое обращение предполагает повторное выполнение запроса. Это демонстрирует следующий пример, который принимает коллекцию coll, копирует ее в массив arr и затем сравнивает скорость итераций по коллекции и массиву.

Определим функцию, копирующую HTML-коллекцию в обычный массив:

```
function toArray(coll) {
    for (var i = 0, a = [], len = coll.length; i < len; i++) {
        a[i] = coll[i];
    }
    return a;
}
```

Теперь создадим коллекцию и скопируем ее в массив:

```
var coll = document.getElementsByTagName('div');
var ar = toArray(coll);
```

Для сравнения будут использоваться следующие две функции:

```
// медленная
function loopCollection() {
    for (var count = 0; count < coll.length; count++) {
        /* ничего не делать */
    }
}

// быстрая
function loopCopiedArray() {
    for (var count = 0; count < arr.length; count++) {
        /* ничего не делать */
    }
}
```

В каждой итерации выполняется обращение к свойству `length` коллекции, что вызывает обновление коллекции и приводит к существенному снижению производительности во всех броузерах. Чтобы оптимизировать реализацию, достаточно просто сохранить длину коллекции в переменной и для проверки условия выхода из цикла использовать эту переменную:

```
function loopCacheLengthCollection() {
    var coll = document.getElementsByTagName('div'),
        len = coll.length;
    for (var count = 0; count < len; count++) {
        /* ничего не делать */
    }
}
```

Эта функция работает так же быстро, как `loopCopiedArray()`.

Во многих случаях, когда требуется выполнить единственный цикл по относительно небольшой коллекции, простого кэширования значения свойства `length` коллекции будет вполне достаточно. Тем не менее обход элементов массива выполняется быстрее, чем обход элементов коллекции, поэтому если предварительно скопировать элементы коллекции в массив, доступ к их свойствам будет выполняться быстрее. Но имейте в виду, что это добавляет в сценарий дополнительный шаг и еще один цикл по элементам коллекции, поэтому важно выполнить профилирование программного кода и на основе полученных результатов решить, будет ли выгоднее использование массива с копией коллекции в вашем конкретном случае.

Внимательно ознакомьтесь с функцией `toArray()`, показанной выше, которая является примером универсальной функции копирования коллекции в массив.

Локальные переменные при обращении к элементам коллекций

В предыдущем примере использовался пустой цикл, но как изменится скорость выполнения, если в теле цикла будут выполняться обращения к элементам коллекции?

Общее правило таково: при любых обращениях к элементам дерева DOM желательно использовать локальные переменные, если одно и то же свойство или метод элемента DOM используется более одного раза. При необходимости выполнить обход коллекции первый шаг оптимизации – сохранить коллекцию и значение ее свойства `length` в локальных переменных, а затем использовать локальную переменную для многократного доступа к элементам коллекции внутри цикла.

В следующем примере внутри цикла выполняются обращения к трем свойствам каждого элемента. Самая медленная версия каждый раз обращается к глобальной переменной `document`, оптимизированная версия сохраняет ссылку на коллекцию, а самая быстрая версия дополнительно сохраняет в локальной переменной еще и текущий элемент коллекции. Все три версии сохраняют значение свойства `length` коллекции в локальной переменной.

```
// медленная
function collectionGlobal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = document.getElementsByTagName('div')[count].nodeName;
        name = document.getElementsByTagName('div')[count].nodeType;
        name = document.getElementsByTagName('div')[count].tagName;
    }
    return name;

};

// быстрая
function collectionLocal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = coll[count].nodeName;
        name = coll[count].nodeType;
        name = coll[count].tagName;
    }
    return name;

};
```

```
// самая быстрая
function collectionNodesLocal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '',
        el = null;
    for (var count = 0; count < len; count++) {
        el = coll[count];
        name = el.nodeName;
        name = el.nodeType;
        name = el.tagName;
    }
    return name;

};
```

На рис. 3.5 показаны результаты сравнения производительности оптимизированных циклов по элементам коллекции. Первый столбик показывает прирост производительности, получаемый за счет обращения к коллекции через локальную переменную, а второй – дополнительный прирост, который дает кэширование элементов коллекции при многократном обращении к ним.

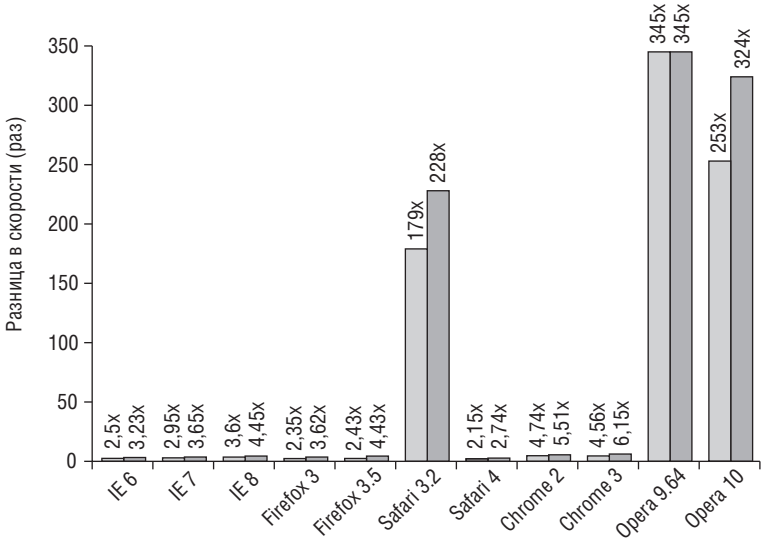


Рис. 3.5. *Прирост производительности за счет использования локальных переменных для хранения ссылок на коллекцию и ее элементы в ходе выполнения циклов*

Обход дерева DOM

Прикладной интерфейс модели DOM обеспечивает множество способов доступа к определенным частям документа. В случаях, когда одну и ту же работу можно выполнить несколькими способами, желательно использовать наиболее эффективный API.

Навигация по дереву DOM

Часто бывает необходимо начать с некоторого определенного элемента DOM и обработать окружающие его элементы, скажем, выполняя рекурсивный обход всех вложенных в него элементов. Такой обход можно реализовать, используя коллекцию `childNodes` или получая братские элементы с помощью свойства `nextSibling`.

Взгляните на две эквивалентные реализации нерекурсивного обхода вложенных элементов:

```
function testNextSibling() {
    var el = document.getElementById('mydiv'),
        ch = el.firstChild,
        name = '';
    do {
        name = ch.nodeName;
    } while (ch = ch.nextSibling);
    return name;
};

function testChildNodes() {
    var el = document.getElementById('mydiv'),
        ch = el.childNodes,
        len = ch.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = ch[count].nodeName;
    }
    return name;
};
```

Имейте в виду, что свойство `childNodes` – это коллекция, требующая осторожного обращения. При выполнении итераций по ней желательно кэшировать значение свойства `length`, чтобы исключить влияние обновления коллекции в каждой итерации.

Оба подхода в разных броузерах обладают практически одинаковой производительностью. Но в IE свойство `nextSibling` действует намного быстрее, чем свойство `childNodes`. В IE6 свойство `nextSibling` действует в 16 раз быстрее, а в IE7 – в 105 раз быстрее. Учитывая это, в приложениях, где требуется высокая производительность, для навигации по дереву DOM в старых версиях IE предпочтительнее использовать свойство `nextSibling`. Во всех остальных случаях выбор того или иного подхода является вопросом личных предпочтений.

Узлы-элементы

Свойства узлов дерева DOM, такие как `childNodes`, `firstChild` и `nextSibling`, не отличают элементы от узлов других типов, например комментариев и текстовых узлов (которые зачастую хранят лишь пробелы между двумя тегами). Во многих случаях бывает необходимо обращаться только к узлам, являющимся элементами, поэтому в цикле нередко приходится предусматривать проверку типов узлов и пропускать узлы, не являющиеся элементами. Для реализации такой проверки типов и фильтрации требуется выполнять лишние обращения к модели DOM.

Многие современные броузеры предлагают прикладные интерфейсы, возвращающие только узлы, являющиеся элементами. Желательно использовать эти интерфейсы, если они имеются, потому что они выполняются быстрее, чем любой другой способ фильтрации, реализованный вручную. Эти удобные свойства узлов дерева DOM перечислены в табл. 3.1.

Таблица 3.1. Свойства узлов дерева DOM, отделяющие элементы (HTML-теги) от всех остальных узлов

Свойство	Используется вместо
`children`	`childNodes`
`childElementCount`	`childNodes.length`
`firstElementChild`	`firstChild`
`lastElementChild`	`lastChild`
`nextElementSibling`	`nextSibling`
`previousElementSibling`	`previousSibling`

Все свойства, перечисленные в табл. 3.1, поддерживаются броузерами Firefox 3.5, Safari 4, Chrome 2 и Opera 9.62. Версии 6 и 7 броузера Internet Explorer поддерживают только свойство `children`.

Обход коллекции `children` происходит быстрее, чем обход коллекции `childNodes`, потому что она обычно содержит меньше элементов. Пробелы, которые имеются в разметке HTML и преобразуются в текстовые узлы, не включаются в коллекцию `children`. Свойство `children` во всех броузерах действует быстрее, чем свойство `childNodes`, хотя обычно разница невелика и составляет от 1,5 до 3 раз. Одним из заметных исключений является IE, где обход коллекции `children` выполняется значительно быстрее, чем обход коллекции `childNodes`, – в 24 раза быстрее в IE6 и в 124 раза быстрее в IE7.

API селекторов

Часто разработчикам требуются более точные инструменты выбора элементов из дерева DOM, чем методы `getElementById()` и `getElementsByTagName()`. Иногда список требуемых элементов можно получить, комби-

нируя эти методы и выполняя обход полученных узлов, но такой подход может оказаться не самым эффективным.

Другим удобным способом идентификации узлов может оказаться выбор по CSS-селекторам – все разработчики знакомы с CSS. Многие JavaScript-библиотеки поддерживают прикладные интерфейсы, позволяющие использовать CSS-селекторы, а теперь и последние версии броузеров предоставляют встроенный DOM-метод `querySelectorAll()`. Естественно, такой подход работает быстрее, чем использование JavaScript и DOM для обхода и фильтрации списка элементов.

Рассмотрим следующий пример:

```
var elements = document.querySelectorAll('#menu a');
```

Переменная `elements` будет содержать список ссылок на все элементы `<a>`, находящиеся в элементе с атрибутом `id="menu"`. Метод `querySelectorAll()` принимает в аргументе строку с CSS-селектором и возвращает объект `NodeList` – объект, подобный массиву, содержащий узлы, которые соответствуют селектору. Значение, возвращаемое методом, не является HTML-коллекцией, поэтому полученный список элементов не является синхронным отображением структуры документа. Это устраняет проблемы снижения производительности (и возможные логические ошибки), свойственные HTML-коллекциям и обсуждавшиеся выше в этой главе.

Чтобы получить те же результаты без применения метода `querySelectorAll()`, потребуется более сложный программный код:

```
var elements = document.getElementById('menu').getElementsByTagName('a');
```

В этом случае переменная `elements` будет содержать HTML-коллекцию, поэтому ее необходимо будет скопировать в массив, чтобы получить такой же статический список, который возвращается методом `querySelectorAll()`.

Метод `querySelectorAll()` еще более удобен, когда требуется объединить несколько запросов. Например, если в странице есть несколько элементов `<div>` с классом «warning», часть из которых имеет класс «notice», метод `querySelectorAll()` позволит получить их все в виде списка:

```
var errs = document.querySelectorAll('div.warning, div.notice');
```

Чтобы получить тот же список без использования метода `querySelectorAll()`, потребуется выполнить большее количество операций. Один из способов заключается в том, чтобы отобрать все элементы `<div>` и отфильтровать ненужные, выполнив обход списка в цикле.

```
var errs = [],
    divs = document.getElementsByTagName('div'),
    classname = '';
for (var i = 0, len = divs.length; i < len; i++) {
    classname = divs[i].className;
```

```
        if (classname === 'notice' || classname === 'warning') {
            errs.push(divs[i]);
        }
    }
```

Сравнение двух представленных подходов показывает, что решение на основе метода `querySelectorAll()` выполняется в 2–6 раз быстрее (рис. 3.6).

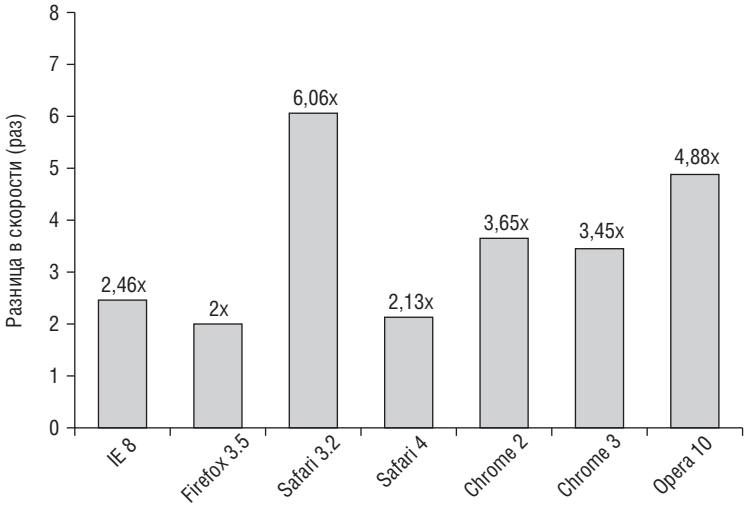


Рис. 3.6. *Преимущество использования API селекторов вместо обхода результатов, возвращаемых методом getElementsbyTagName()*

API селекторов поддерживается броузерами, начиная с версий: Internet Explorer 8, Firefox 3.5, Safari 3.1, Chrome 1 и Opera 10.

Учитывая результаты, представленные на рис. 3.6, было бы желательно проверять наличие поддержки метода `document.querySelectorAll()` и использовать его, если это возможно. Кроме того, если сценарий использует API селекторов, предоставляемый JavaScript-библиотекой, нелишним будет убедиться, что внутри библиотека использует встроенный API. В противном случае, вероятно, будет достаточно просто обновить версию библиотеки.

Можно также использовать другой удобный метод с именем `querySelector()`, возвращающий только первый узел, соответствующий запросу.

Эти два метода являются свойствами узлов дерева DOM, поэтому отобрать узлы из всего документа можно с помощью вызова `document.querySelector('.myclass')`, а выполнить запрос к поддереву можно с помощью вызова `elref.querySelector('.myclass')`, где `elref` – ссылка на элемент DOM.

Перерисовывание и перекомпоновка

После загрузки всех компонентов страницы – разметки HTML, сценариев JavaScript, таблиц CSS, изображений – броузер выполняет синтаксический анализ полученных файлов и создает две внутренние структуры данных:

Дерево DOM

Представляет структуру страницы.

Дерево отображения

Определяет порядок отображения узлов дерева DOM.

Дерево отображения содержит, по крайней мере, по одному узлу для каждого узла в дереве DOM, который требуется отобразить (скрытые элементы дерева DOM не имеют соответствующих им узлов в дереве отображения). Узлы в дереве отображения называются *кадрами*, или *блоками*, в соответствии с моделью CSS, интерпретирующей элементы страницы как блоки, имеющие отступы, поля, рамки и координаты местоположения. Закончив создание деревьев, броузер сможет отобразить («нарисовать») элементы страницы.

Когда в дереве DOM выполняются изменения, затрагивающие геометрию элемента (ширину и высоту), такие как изменение толщины рамки или добавление в абзац еще одной строки текста, броузер должен пересчитать геометрию элемента, а также геометрию и координаты других элементов, которые могли быть затронуты изменениями. Броузер объявляет недействительной часть дерева отображения, затронутую изменениями, и реконструирует ее. Этот процесс называется *перекомпоновка* (*reflow*). По окончании перекомпоновки броузер повторно выводит на экран части, затронутые изменениями в процессе *перерисовывания* (*repaint*).

Не все изменения в дереве DOM оказывают влияние на геометрию элементов. Например, изменение цвета фона элемента не изменяет его ширину или высоту. В этом случае выполняется только перерисовывание (без перекомпоновки), потому что размещение элементов не изменилось.

Перерисовывание и перекомпоновка являются достаточно дорогостоящими операциями и могут ухудшить время реакции пользовательского интерфейса веб-приложения. В силу этого важно стараться избегать ситуаций, вызывающих их, насколько это возможно.

Когда происходит перекомпоновка?

Как упоминалось выше, перекомпоновка необходима, когда изменяется расположение или геометрия элементов. Это происходит, когда:

- В дерево DOM добавляются или удаляются видимые элементы
- Изменяется местоположение элементов
- Изменяются размеры элементов (например, из-за изменения ширины полей или отступов, толщины рамки, ширины, высоты и т. д.)

- Изменяется содержимое, например, когда изменяется текст или одно изображение замещается другим с другими размерами
- Выполняется начальное отображение страницы
- Изменяются размеры окна броузера

В зависимости от природы изменений бывает необходимо пересчитать меньшую или большую часть дерева отображения. Некоторые изменения могут вызвать перекомпоновку всей страницы, например появление полосы прокрутки.

Буферизация и применение изменений в дереве отображения

Из-за высокой стоимости вычислений, связанных с каждой перекомпоновкой, большинство броузеров оптимизируют процесс перекомпоновки, накапливая изменения в буфере и затем применяя их за один прием. Однако иногда (часто непреднамеренно) можно заставить броузер немедленно применить все накопленные в буфере изменения. Применение накопленных в буфере изменений происходит, когда выполняется попытка получить информацию о расположении элементов, то есть при использовании любого из следующих свойств и методов:

- `offsetTop`, `offsetLeft`, `offsetWidth`, `offsetHeight`
- `scrollTop`, `scrollLeft`, `scrollWidth`, `scrollHeight`
- `clientTop`, `clientLeft`, `clientWidth`, `clientHeight`
- `getComputedStyle()` (`currentStyle` в IE)

Эти свойства и методы должны возвращать самую актуальную информацию о размещении элементов, и поэтому броузер применяет все накопленные изменения в дереве отображения и выполняет перекомпоновку, чтобы вернуть корректные значения.

В процессе изменения стилей желательно не использовать свойства и методы, перечисленные выше. Все они вызывают применение накопленных изменений даже при попытке получить информацию о размещении элементов, которая не изменялась и не изменится при применении последних изменений.

Рассмотрим следующий пример, который трижды изменяет одно и то же свойство стиля (этот прием едва ли можно встретить в действующем программном коде, тем не менее он достаточно наглядно иллюстрирует важную тему):

```
// установить и извлечь стили
var computed,
    tmp = ``,
    bodystyle = document.body.style;

if (document.body.currentStyle) { // IE, Opera
    computed = document.body.currentStyle;
```

```
} else { // W3C
    computed = document.defaultView.getComputedStyle(document.body, '');
}

// неэффективный способ изменения одного и того же свойства
// и извлечения информации о стиле сразу после изменения
bodystyle.color = 'red';
tmp = computed.backgroundColor;
bodystyle.color = 'white';
tmp = computed.backgroundImage;
bodystyle.color = 'green';
tmp = computed.backgroundAttachment;
```

В этом примере трижды изменяется цвет переднего плана элемента <body>, и после каждого изменения извлекаются значения свойств вычисляемого стиля. Все извлекаемые свойства – backgroundColor, backgroundImage и backgroundAttachment – никак не связаны с изменяемым цветом. Тем не менее сам факт запроса свойства вычисляемого стиля вынуждает броузер применить изменения, накопленные в буфере, и выполнить перекомпоновку.

Более удачное решение заключается в том, чтобы никогда не запрашивать информацию о размещении элементов, пока не будут выполнены необходимые изменения. Если операции получения свойств вычисляемого стиля переместить в конец, получится такая последовательность инструкций:

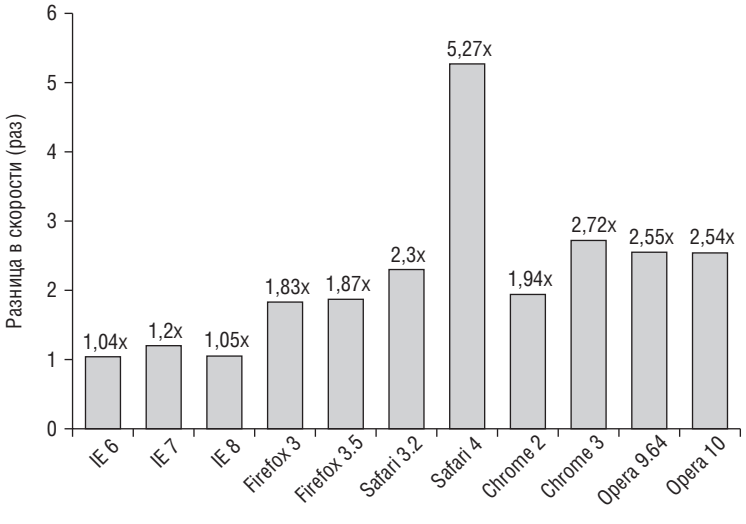


Рис. 3.7. *Преимущество предотвращения перекомпоновки элементов за счет откладывания операций получения информации о размещении*

```
bodystyle.color = 'red';
bodystyle.color = 'white';
bodystyle.color = 'green';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
```

Второй пример будет работать быстрее во всех броузерах, как показывают результаты на рис. 3.7.

Уменьшение количества операций перерисовывания и перекомпоновки

Операции перекомпоновки и перерисовывания могут приводить к существенным потерям производительности, поэтому желательно уменьшать их количество, чтобы улучшить отзывчивость приложения. Для этого следует объединять множественные изменения стилей и элементов в дереве DOM и применять их группами.

Изменение стилей

Рассмотрим следующий пример:

```
var el = document.getElementById('mydiv');
el.style.borderLeft = '1px';
el.style.borderRight = '2px';
el.style.padding = '5px';
```

Здесь изменяются три свойства стиля, каждое из которых оказывает влияние на геометрию элемента. В самом тяжелом случае броузер выполнит операцию перекомпоновки трижды. Большинство современных броузеров оптимизирует такие ситуации и выполняет перекомпоновку только один раз, но в старых броузерах подобная оптимизация может отсутствовать или ее преимущества могут сводиться на нет другим асинхронным процессом (например, обработчиком событий от таймера). Если какой-либо другой программный код запросит информацию о размещении, пока выполняется данный программный код, это может вызвать до трех перекомпоновок. Кроме того, этот пример обращается к модели DOM четыре раза, и это обстоятельство также можно оптимизировать.

Более эффективный способ добиться того же результата заключается в объединении всех изменений и применении их всех сразу путем внесения изменений в дерево DOM только единожды. Реализовать это можно с помощью свойства `cssText`:

```
var el = document.getElementById('mydiv');
el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';
```

Операция изменения свойства `cssText`, как в этом примере, затирает существующую информацию о стиле, поэтому если необходимо сохранить

существующие стили, можно воспользоваться операцией добавления новых значений в конец строки cssText:

```
el.style.cssText += '; border-left: 1px;';
```

Другой способ применения сразу всех изменений стилей заключается в изменении имени CSS-класса вместо встроенных стилей. Этот подход применим в ситуациях, когда стили не зависят от логики работы и производимых вычислений. Прием на основе изменения имени CSS-класса очевиден и прост в сопровождении; он позволяет освободить сценарии от программного кода, определяющего представление элементов, но может оказывать небольшое отрицательное влияние на производительность из-за необходимости выполнять проверки при изменении классов.

```
var el = document.getElementById('mydiv');
el.className = 'active';
```

Группировка изменений в дереве DOM

При применении множества изменений к элементу дерева DOM можно уменьшить количество операций перерисовывания и перекомпоновки, соблюдая следующую последовательность действий:

1. Извлечь элемент из потока отображения документа.
2. Применить множество изменений.
3. Вернуть элемент обратно в документ.

Эта последовательность вызывает две операции перекомпоновки – на шаге 1 и на шаге 3. Если опустить эти шаги, то каждое изменение в шаге 2 может вызывать перекомпоновку.

Ниже приводятся три основных способа выполнения изменений дерева DOM за пределами документа:

- Скрыть элемент, применить изменения и снова отобразить элемент.
- Используя фрагмент документа, сконструировать поддерево за пределами дерева DOM и затем скопировать его в документ.
- Скопировать элемент в узел за пределами документа, изменить копию и затем заменить оригинальный элемент.

Для иллюстрации приема внесения изменений за пределами документа возьмем в качестве примера список ссылок, в который необходимо добавить дополнительную информацию:

```
<ul id="mylist">
   <li><a href="http://phpied.com">Stoyan</a></li>
   <li><a href="http://julienlecomte.com">Julien</a></li>
</ul>
```

Предположим, что дополнительные данные, которые требуется вставить в список, содержатся в следующем объекте:

```
var data = [
  {
```

```
    "name": "Nicholas",
    "url": "http://nczonline.net"
  },
  {
    "name": "Ross",
    "url": "http://techfoolery.com"
  }
];
```

Ниже приводится универсальная функция, добавляющая новые данные в указанный узел:

```
function appendDataToElement(appendToElement, data) {
    var a, li;
    for (var i = 0, max = data.length; i < max; i++) {
        a = document.createElement('a');
        a.href = data[i].url;
        a.appendChild(document.createTextNode(data[i].name));
        li = document.createElement('li');
        li.appendChild(a);
        appendToElement.appendChild(li);
    }
};
```

Наиболее очевидный способ добавления новых данных в список, если не задумываться о перекомпоновке, выглядит так:

```
var ul = document.getElementById('mylist');
appendDataToElement(ul, data);
```

Однако при таком подходе каждый новый элемент из массива `data` будет добавляться непосредственно в дерево DOM и вызывать перекомпоновку. Как обсуждалось выше, один из способов уменьшить количество операций перекомпоновки заключается в том, чтобы временно исключить элемент `<ul>` из потока отображения документа, изменив его свойство `display`, и затем вернуть его:

```
var ul = document.getElementById('mylist');
ul.style.display = 'none';
appendDataToElement(ul, data);
ul.style.display = 'block';
```

Другой способ уменьшить количество операций перекомпоновки – создать и изменить фрагмент документа вообще за пределами документа и затем добавить его в конец оригинального списка. Фрагмент документа – это легковесная версия объекта `document`, предназначенная как раз для решения подобных задач – изменения и перемещения узлов. Одна из удобных синтаксических особенностей фрагментов документа состоит в том, что при добавлении фрагмента в узел добавляется не сам фрагмент, а его дочерние элементы. Следующее решение короче предыдущего на одну строку, вызывает одну операцию перекомпоновки и обращается к действующему дереву DOM только один раз:

```
var fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
document.getElementById('mylist').appendChild(fragment);
```

Третье решение заключается в том, чтобы создать копию узла, внести требуемые изменения в копию и затем заменить старый узел измененной копией:

```
var old = document.getElementById('mylist');
var clone = old.cloneNode(true);
appendDataToElement(clone, data);
old.parentNode.replaceChild(clone, old);
```

В общем случае рекомендуется использовать фрагменты документа (второе решение), если это возможно, потому что при этом выполняется меньше манипуляций с деревом DOM и вызывается меньше операций перекомпоновки. Единственный возможный недостаток заключается в том, что в настоящее время практика использования фрагментов документа распространена недостаточно широко, и кто-то в коллективе может быть не знаком с этим методом.

Кэширование информации о размещении

Как уже упоминалось, броузеры пытаются минимизировать количество перекомпоновок за счет буферизации изменений и применения их группами. Но когда выполняется запрос на получение информации о размещении, такой как смещение, величина прокрутки или значение вычисляемого стиля, броузер вынужден немедленно применить все изменения, накопленные в буфере, чтобы вернуть актуальное значение. По этой причине желательно стараться минимизировать количество запросов на получение информации о размещении, сохранять ее в локальных переменных и работать с локальными значениями.

Рассмотрим пример перемещения элемента `myElement` по диагонали на один пиксел за раз из позиции 100×100 px в позицию 500×500 px. В теле цикла обработки событий от таймера такое перемещение можно было бы реализовать так:

```
// неэффективное решение
myElement.style.left = 1 + myElement.offsetLeft + 'px';
myElement.style.top = 1 + myElement.offsetTop + 'px';
if (myElement.offsetLeft >= 500) {
    stopAnimation();
}
```

Однако это неэффективное решение, потому что при каждом перемещении элемента запрашиваются значения его смещений, что вынуждает броузер в каждом цикле применять изменения и не дает возможности использовать преимущества оптимизации. Более удачный способ сделать то же самое состоит в том, чтобы прочитать начальные значения координат только один раз и сохранить их в локальных переменных,

таких как `var current = myElement.offsetLeft;`. Затем внутри цикла воспроизведения анимации все операции выполнять с использованием переменной `current`, не запрашивая смещения:

```
current++
myElement.style.left = current + 'px';
myElement.style.top = current + 'px';
if (current >= 500) {
    stopAnimation();
}
```

Исключение элементов из потока отображения для внесения изменений

Сокрытие и отображение фрагментов страниц путем их сворачивания/разворачивания является одним из распространенных приемов взаимодействий. При этом часто данная операция сопровождается анимационным эффектом изменения геометрии сворачиваемой области, в результате которой происходит смещение остального содержимого страницы.

Иногда перекомпоновка затрагивает только малую часть дерева отображения, но порой она может вызываться для больших фрагментов или даже для всего дерева. Чем реже выполняется операция перекомпоновки, тем лучше время отклика приложения. Поэтому когда анимационный эффект выполняется в верхней части страницы, смещая вниз почти всю страницу целиком, он может привести к дорогостоящей, масштабной перекомпоновке. Чем больше узлов в дереве отображения подвергаются пересчету, тем хуже становится время отклика приложения.

Соблюдение следующей последовательности действий поможет избежать перекомпоновки значительной части страницы:

1. Определить для элемента, к которому должен быть применен анимационный эффект, режим абсолютного позиционирования, чтобы исключить его из общего потока страницы.
2. Воспроизвести анимационный эффект. Когда элемент развернется, он временно перекроет часть страницы. Это приведет к перерисовыванию малой части страницы вместо перекомпоновки и перерисовывания большого фрагмента.
3. По завершении анимационного эффекта восстановить прежний режим позиционирования, что вызовет смещение остальной части документа, но только один раз.

IE и :hover

Начиная с версии 7 IE позволяет применять псевдоселектор CSS `:hover` к любым элементам (в строгом режиме). Однако наличие большого количества элементов с этим псевдоселектором ухудшает время отклика приложения. Еще более ярко эта проблема наблюдается в IE8.

Например, если создать таблицу с 500–1000 строками и 5 колонками и использовать `tr:hover` для реализации изменения цвета фона и выделения строки под указателем мыши, то, когда пользователь перемещает указатель мыши над таблицей, производительность ощутимо падает. Подсветка строк замедляется, а нагрузка на CPU возрастает до 80–90%. Поэтому старайтесь не использовать этот эффект при работе с большим количеством элементов, например с большими таблицами или длинными списками.

Делегирование обработки событий

Когда в странице имеется большое количество элементов, к каждому из которых подключен один или более обработчиков событий (таких как `onclick`), это может отрицательно сказаться на производительности. За подключение каждого обработчика приходится платить определенную цену – либо в виде утяжеления страницы (увеличения объема кода разметки или программного кода на языке JavaScript), либо в виде увеличения времени выполнения. Чем к большему количеству узлов дерева DOM требуется прикоснуться и изменить их, тем медленнее будет работать приложение, так как подключение обычно выполняется по событию `onload` (или `DOMContentReady`) – самый загруженный этап выполнения любой веб-страницы, предусматривающей богатые возможности взаимодействия с пользователем. Операция подключения обработчика событий требует определенного времени, и, кроме того, броузер должен запомнить связь обработчиков с элементами, что вдобавок ведет к увеличению расхода памяти. И наконец, большое количество этих обработчиков событий может никогда не понадобиться (потому что, например, пользователь должен щелкнуть на одной кнопке или ссылке, а не на каждой из 100 имеющихся), то есть значительная доля работы может не быть востребована.

Прием делегирования обработки событий модели DOM обеспечивает простой и элегантный способ их обработки. Он опирается на тот факт, что события всплывают, благодаря чему могут обрабатываться в родительских элементах. При использовании приема делегирования обработки событий подключается всего один обработчик к вмещающему элементу, который обрабатывает все события, возникшие в дочерних элементах.

Согласно стандарту DOM каждое событие имеет три фазы распространения:

- Перехват
- Передача целевому элементу
- Всплытие

Фаза перехвата не поддерживается в IE, но для реализации приема делегирования вполне достаточно наличия фазы всплытия. Рассмотрим страницу со структурой, изображенной на рис. 3.8.

***Рис. 3.8.** Пример дерева DOM*

Когда пользователь щелкнет на ссылке menu #1, событие `click` сначала будет принято элементом `<a>`. Затем оно начнет всплывать по дереву DOM и будет принято элементом `<li>`, затем элементом `<ul>`, затем `<div>` и т. д. до самой вершины документа и даже до объекта `window`. Эта особенность позволяет подключить единственный обработчик события к родительскому элементу и обрабатывать все события, возникающие в дочерних элементах.

Предположим, что к документу, представленному на рис. 3.8, потребовалось добавить прогрессивную поддержку технологии Ajax. Если пользователь отключит поддержку JavaScript, то ссылки в меню будут действовать как обычно и щелчок на любой из них будет приводить к перезагрузке страницы. Но в случае если поддержка JavaScript включена и броузер обладает необходимыми возможностями, было бы желательно перехватывать все щелчки на ссылках, отменять выполнение действий по умолчанию (то есть переход по ссылке), отправлять Ajax-запрос на получение содержимого и обновлять фрагмент страницы без полной ее перезагрузки. Для реализации желаемого поведения можно воспользоваться приемом делегирования обработки событий, подключив обработчик события `click` к элементу `<ul>` с атрибутом `id="menu"`, который обертывает все ссылки и проверяет все события `click`, ожидая событий от ссылок.

```
document.getElementById('menu').onclick = function(e) {

    // для совместимости с разными броузерами
    e = e || window.event;
    var target = e.target || e.srcElement;

    var pageid, hrefparts;

    // интересуют только события в гиперссылках
    // выйти, если событие click возникло не в ссылке
    if (target.nodeName !== 'A') {
        return;
    }
```

```
        // извлечь идентификатор страницы из ссылки
        hrefparts = target.href.split('/');
        pageid = hrefparts[hrefparts.length - 1];
        pageid = pageid.replace('.html', '');

        // обновить страницу
        ajaxRequest('xhr.php?page=' + pageid, updatePageContents);

        // отменить действие по умолчанию
        // и прервать всплытие события в любом броузере
        if (typeof e.preventDefault === 'function') {
            e.preventDefault();
            e.stopPropagation();
        } else {
            e.returnValue = false;
            e.cancelBubble = true;
        }
    };
```

Как видно из этого примера, прием делегирования обработки событий не сложен в реализации; необходимо только проверять события, чтобы убедиться, что они возникли в требуемых элементах. В примере имеется программный код, обеспечивающий совместимость броузеров, но если перенести его в библиотеку, реализация упростится еще больше. Программный код, обеспечивающий совместимость броузеров, включает реализацию:

- доступа к объекту события и идентификацию источника события (целевой элемент);
- прерывания всплытия события вверх по дереву документа (необязательно);
- отмены действий по умолчанию (необязательно, но в данном случае необходимо, потому что задача состояла в том, чтобы перехватить события от ссылок и отменить переход по ним).

В заключение

Операции доступа к дереву DOM и манипулирование им являются важной частью современных веб-приложений. Но за каждое пересечение моста, соединяющего острова ECMAScript и DOM, приходится платить. Чтобы уменьшить потери производительности, связанные с операциями над деревом DOM, соблюдайте следующие правила:

- Как можно реже обращайтесь к дереву DOM и старайтесь большую часть работы выполнять на территории JavaScript.
- Используйте локальные переменные для хранения ссылок на элементы в дереве DOM, к которым приходится обращаться многократно.
- С осторожностью используйте HTML-коллекции, потому что они представляют синхронную структуру документа. Кэшируйте значе-

ние свойства `length` в переменной и используйте ее при выполнении итераций, а также копируйте коллекцию в массив, когда требуется выполнить большое количество операций с коллекцией.

- Используйте более быстрые API, если доступны, такие как `querySelectorAll()` и `firstElementChild`.
- Не забывайте об операциях перерисовывания и перекомпоновки; группируйте изменения стилей, манипулируйте деревом DOM «в автономном режиме», минимизируйте количество операций чтения информации о размещении и сохраняйте ее в переменных.
- Переводите элементы в режим абсолютного позиционирования перед началом воспроизведения анимационных эффектов.
- Используйте прием делегирования обработки событий для уменьшения количества обработчиков событий.

4

Алгоритмы и управление потоком выполнения

Общая структура программного кода является одним из основных факторов, влияющих на его производительность. Компактность программного кода не является гарантией, что он будет выполняться быстро, а код значительного объема совсем необязательно будет работать медленно. Производительность программного кода во многом зависит от его организации и от выбранного алгоритма решения задачи.

Приемы, описываемые в этой главе, не являются уникальными для языка JavaScript и нередко преподаются как способы оптимизации при обучении другим языкам программирования. Некоторые из рекомендаций, приводимых здесь, не применимы к другим языках программирования, потому что в JavaScript имеется множество механизмов, уникальные особенности которых необходимо учитывать, тем не менее все описываемые приемы основаны на знаниях, накопленных в области информатики.

Циклы

В большинстве языков программирования основную часть времени программный код затрачивает на выполнение циклов. Обход последовательности значений в цикле является одним из самых распространенных шаблонов программирования и, как следствие, одной из областей, где следует сосредоточить усилия по улучшению производительности. Понимание того, как циклы в языке JavaScript влияют на производительность, является особенно важным, потому что бесконечные или долго выполняющиеся циклы оказывают прямое влияние на восприятие пользователя.

Типы циклов

Третья версия спецификации ECMA-262, описывающая базовый синтаксис и особенности JavaScript, определяет четыре типа циклов. Первый – стандартный цикл `for`, который имеет тот же синтаксис, что и в других C-подобных языках:

```
for (var i=0; i < 10; i++){
    // тело цикла
}
```

Цикл `for` наиболее часто используется в языке JavaScript для реализации циклических конструкций. Цикл `for` состоит из четырех частей: инициализация, предварительная проверка условия, завершающая операция и тело цикла. Когда интерпретатор встречает цикл `for`, он сначала выполняет инициализацию, а затем проверяет условие. Если условное выражение вернет `true`, будет выполнено тело цикла. Вслед за телом цикла выполняется завершающая операция. Удобный синтаксис цикла `for` делает его наиболее популярным среди разработчиков.

Обратите внимание, что инструкция `var` в части инициализации цикла `for` создает переменную, видимую не только в теле цикла, но и во всем теле функции. В языке JavaScript отсутствуют области видимости ниже уровня функции, поэтому определение новой переменной внутри цикла `for` равносильно определению новой переменной внутри функции за пределами цикла.

Второй тип циклов – цикл `while`. Цикл `while` – это простой цикл с предварительной проверкой условия продолжения, состоящий из условного выражения и тела цикла:

```
var i = 0;
while(i < 10){
    // тело цикла
    i++;
}
```

Перед выполнением тела цикла вычисляется значение условного выражения. Если условное выражение вернет `true`, будет выполнено тело цикла; в противном случае оно будет пропущено. Любой цикл `for` можно реализовать как цикл `while`, и наоборот.

Третий тип циклов – цикл `do-while`. Цикл `do-while` является единственным в языке JavaScript, выполняющим проверку условия продолжения после выполнения тела цикла, и состоит из двух частей – тела цикла и условного выражения:

```
var i = 0;
do {
    // тело цикла
} while (i++ < 10);
```

Цикл do-while всегда выполняет тело цикла хотя бы один раз, а условное выражение в конце цикла определяет необходимость повторного выполнения цикла.

Четвертый и последний тип циклов – цикл for-in. Этот цикл имеет весьма специфическое назначение: он выполняет обход именованных свойств любого объекта. Базовый синтаксис имеет следующий вид:

```
for (var prop in object){
    // тело цикла
}
```

В каждой итерации цикла переменной prop присваивается имя (в виде строки) очередного свойства объекта. Цикл продолжается, пока не будут перечислены все свойства. В перечислении участвуют и собственные свойства экземпляра объекта, и унаследованные из цепочки прототипов.

Производительность цикла

Вопрос оценки производительности при выборе цикла того или иного типа является постоянным источником дебатов. Из четырех типов циклов, имеющихся в языке JavaScript, только цикл for-in оказывается существенно медленнее остальных.

Поскольку в каждой итерации цикла for-in выполняется операция поиска свойства в экземпляре или в прототипе, он имеет самые значительные накладные расходы и потому выполняется медленнее остальных циклов. Одно и то же количество итераций цикла for-in может выполняться в семь раз медленнее, чем в циклах других типов. По этой причине рекомендуется избегать использования циклов for-in, если только нет необходимости выполнить обход заранее неизвестного количества свойств объекта. Если список свойств, по которым следует выполнить итерации, известен заранее, цикл любого другого типа будет выполняться гораздо быстрее, и в этом случае можно использовать следующий шаблон:

```
var props = ["prop1", "prop2"],
    i = 0;

while (i < props.length){
    process(object[props[i++]]);
}
```

Этот фрагмент создает массив, элементами которого являются имена свойств. Цикл while используется здесь для обхода и обработки этого небольшого числа свойств объекта. Вместо поиска всех и каждого свойства объекта это решение сконцентрировано только на нужных свойствах, экономя время, необходимое на выполнение лишних итераций.

Никогда не используйте цикл for-in для обхода элементов массива.

За исключением цикла `for-in`, все остальные типы циклов имеют одинаковую производительность, поэтому бессмысленно тратить время на выявление наиболее быстрого из них. Выбор типа цикла должен основываться не на производительности, а на других требованиях.

Если тип цикла не влияет на производительность, тогда что влияет? В действительности на производительность влияют всего два фактора:

- Объем работы, выполняемой в одной итерации
- Количество итераций

Уменьшение любого из этих факторов или обоих сразу положительно сказывается на общей производительности цикла.

Уменьшение объема работы, выполняемой в одной итерации

Очевидно, что чем дольше выполняется одна итерация цикла, тем дольше будет выполняться весь цикл. Ограничение количества дорогостоящих операций, выполняемых в теле цикла, является отличным способом увеличения скорости выполнения всего цикла.

Типичный цикл обработки массива можно реализовать на основе любого из трех наиболее быстрых типов циклов. Чаще всего используются следующие шаблоны:

```
// оригинальные циклы
for (var i=0; i < items.length; i++){
    process(items[i]);
}

var j=0;
while (j < items.length){
    process(items[j++]);
}

var k=0;
do {
    process(items[k++]);
} while (k < items.length);
```

Во всех этих циклах в каждой итерации выполняется несколько операций:

1. Поиск свойства (`items.length`) при проверке условия продолжения.
2. Сравнение (`i < items.length`) при проверке условия продолжения.
3. Сравнение для проверки результата условного выражения (`i < items.length == true`).
4. Увеличение значения переменной (`i++`).
5. Поиск элемента массива (`items[i]`).
6. Вызов функции (`process(items[i])`).

В каждой итерации этих простых циклов выполняется множество операций, несмотря на небольшой объем программного кода. Скорость выполнения этих циклов в значительной степени зависит от того, что делает функция `process()` с каждым элементом, но даже в этом случае уменьшение общего количества операций, выполняемых в каждой итерации, может существенно повысить производительность цикла.

Первым шагом к уменьшению объема работы, выполняемой в цикле, является уменьшение числа операций поиска членов объектов и элементов массивов. Как отмечалось в главе 2, в большинстве броузеров эти операции выполняются значительно медленнее, чем операции поиска локальных переменных или извлечения данных из литералов. В предыдущих примерах во всех итерациях в каждом цикле выполняется поиск свойства `items.length`. Это слишком расточительно. Значение данного свойства не изменяется в ходе выполнения цикла, поэтому обращение к нему в каждой итерации является просто тратой времени. Если выполнить поиск свойства один раз, сохранить его значение в локальной переменной и затем использовать эту переменную в условном выражении, это повысит скорость выполнения цикла:

```
// уменьшение числа операций поиска свойства
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}

var j=0,
    count = items.length;
while (j < count){
    process(items[j++]);
}

var k=0,
    num = items.length;
do {
    process(items[k++]);
} while (k < num);
```

Каждая из этих новых версий реализации циклов выполняет единственную операцию поиска свойства `length` массива перед началом цикла. Это позволяет составить выражение проверки условия продолжения цикла исключительно из локальных переменных и тем самым обеспечить более высокую скорость их выполнения. В зависимости от длины массива в большинстве броузеров этот прием позволяет сэкономить до 25% от общего времени выполнения цикла (и до 50% в Internet Explorer).

Производительность можно также увеличить, если изменить направление движения цикла. Часто порядок обработки элементов массива не имеет значения, и потому обработка в направлении от последнего элемента к первому является вполне допустимой альтернативой. Изменение направления движения цикла является распространенным приемом оптимизации производительности во многих языках программиро-

вания, но понимают это немногие. В языке JavaScript изменение порядка движения цикла на обратный позволяет получить небольшой прирост производительности циклов за счет устранения лишних операций:

```
// уменьшение числа операций поиска свойства и обход в обратном направлении
for (var i=items.length; i--; ){
    process(items[i]);
}

var j = items.length;
while (j--){
    process(items[j]);
}

var k = items.length-1;
do {
    process(items[k]);
} while (k--);
```

Циклы в этом примере выполняют обход массива в обратном порядке и совмещают в одном выражении проверку условия продолжения цикла и операцию уменьшения. Теперь в каждом условном выражении просто выполняется сравнение с нулем. Результат условного выражения сравнивается со значением `true`, а так как любое число, отличное от нуля, автоматически преобразуется в `true`, это делает нулевое значение эквивалентом значения `false`. Фактически условное выражение, выполнявшее прежде два сравнения (сравнение переменной цикла с общим количеством и затем сравнение результата со значением `true`), теперь выполняет одно сравнение (значение переменной цикла сравнивается со значением `true`). Уменьшение количества сравнений в каждой итерации еще больше повышает скорость выполнения цикла. Изменение порядка обхода на обратный и уменьшение количества операций поиска свойства позволяют получить увеличение скорости выполнения на 50–60% по сравнению с оригинальными версиями циклов.

Для сравнения с исходным кодом ниже перечислены операции, выполняемые в каждой итерации этих версий циклов:

1. Одно сравнение (`i == true`) в условном выражении.
2. Одна операция уменьшения (`i--`).
3. Поиск элемента массива (`items[i]`).
4. Вызов функции (`process(items[i])`).

Новые версии циклов выполняют на две операции меньше в каждой итерации, что ведет к увеличению производительности, особенно заметному при большом количестве итераций.

Уменьшение объема работы, выполняемой в каждой итерации, дает особенно заметный эффект, когда цикл имеет сложность *O(n)*. Если сложность цикла выше, чем *O(n)*, внимание лучше сосредоточить на уменьшении количества итераций.

Уменьшение количества итераций

Даже самый быстрый программный код будет выполняться достаточно долго, когда в цикле выполняются тысячи итераций. Кроме того, небольшие накладные расходы, связанные с организацией выполнения самих итераций, также увеличивают общее время выполнения. Таким образом, уменьшение количества итераций в цикле может дать существенный прирост производительности. Самым известным приемом ограничения количества итераций является шаблон под названием «Устройство Даффа» (Duff's Device)[1].

«Устройство Даффа» – это прием раскручивания тела цикла так, что каждая итерация фактически выполняет работу нескольких итераций. Считается, что первым перенес реализацию шаблона «Устройство Даффа» с языка C на язык JavaScript Джефф Гринберг (Jeff Greenberg). Типичная реализация шаблона выглядит, как показано ниже:

```
// автор: Джефф Гринберг (Jeff Greenberg)
var iterations = Math.floor(items.length / 8),
    startAt = items.length % 8,
    i       = 0;

do {
    switch(startAt){
        case 0: process(items[i++]);
        case 7: process(items[i++]);
        case 6: process(items[i++]);
        case 5: process(items[i++]);
        case 4: process(items[i++]);
        case 3: process(items[i++]);
        case 2: process(items[i++]);
        case 1: process(items[i++]);
    }
    startAt = 0;
} while (iterations--);
```

Основная идея шаблона «Устройство Даффа» заключается в том, что в каждой итерации цикла допускается выполнить до восьми вызовов функции `process()`. Количество итераций цикла определяется делением общего количества элементов на восемь. Поскольку не все числа делятся на восемь без остатка, остаток записывается в переменную `startAt`, и он определяет количество вызовов функции `process()` в первой итерации цикла. Например, если в массиве имеется 12 элементов, то в первой итерации цикла функция `process()` будет вызвана 4 раза, а во второй – 8 раз, при этом всего будет выполнено две итерации вместо 12.

Чуть более быстрая версия реализации этого алгоритма не имеет инструкции `switch` и отделяет обработку остатка от обработки основной массы элементов:

[1] *http://ru.wikipedia.org/wiki/Устройство_Даффа – Прим. перев.*

```
// автор: Джефф Гринберг (Jeff Greenberg)
var iterations = items.length % 8;
    i          = items.length - 1;

while(iterations){
    process(items[i--]);
    iterations--;
}

iterations = Math.floor(items.length / 8);
while(iterations){
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    iterations--;
}
```

Даже при том что эта реализация выполняет два цикла вместо одного, удаление инструкции `switch` из тела цикла обеспечило увеличение производительности по сравнению с оригинальной версией.

Выигрыш от применения шаблона «Устройство Даффа» в оригинальном или измененном виде в значительной степени зависит от количества выполняемых итераций. В случаях когда в цикле выполняется не более 1000 итераций, прирост производительности едва ли будет значительным по сравнению с использованием обычной конструкции цикла. Однако при количестве итераций выше 1000 эффективность применения шаблона «Устройство Даффа» увеличивается значительно. Например, для цикла, насчитывающего 500 000 итераций, использование этого шаблона уменьшает время выполнения почти на 70%.

Итерации на основе функций

Пятая редакция стандарта ECMA-262 определяет новый метод истинных массивов, который называется `forEach()`. Этот метод выполняет обход элементов массива и для каждого из них вызывает указанную функцию. Функция, вызываемая для каждого элемента и передаваемая методу `forEach()` в виде аргумента, должна принимать три аргумента: значение элемента массива, индекс элемента массива и сам массив. Ниже приводится пример использования этого метода:

```
items.forEach(function(value, index, array){
    process(value);
});
```

Метод `forEach()` реализован в Firefox, Chrome и Safari. Кроме того, многие JavaScript-библиотеки предоставляют логический эквивалент:

```
// YUI 3
Y.Array.each(items, function(value, index, array){
    process(value);
});

// jQuery
jQuery.each(items, function(index, value){
    process(value);
});

// Dojo
dojo.forEach(items, function(value, index, array){
    process(value);
});

// Prototype
items.each(function(value, index){
    process(value);
});

// MooTools
$each(items, function(value, index){
    process(value);
});
```

Несмотря на то что реализация итераций на основе функций обеспечивает более удобный способ их выполнения, в целом она выполняется существенно медленнее обычных циклов. Замедление можно объяснить накладными расходами, связанными с дополнительным вызовом метода для каждого элемента массива. В любом случае итерации на основе функций выполняются до восьми раз дольше обычных циклов и потому не подходят в ситуациях, когда производительность имеет большое значение.

Условные инструкции

Подобно циклам, условные инструкции определяют, как будет выполняться программный код. На выбор между условной инструкцией `if-else` и инструкцией `switch` в языке JavaScript влияют те же факторы, что и в других языках программирования. Однако поскольку разные броузеры по-разному оптимизируют инструкции управления потоком выполнения, выбор между ними не всегда очевиден.

Сравнение if-else и switch

Принято считать, что выбор между инструкциями `if-else` и `switch` зависит от количества выполняемых сравнений: чем больше условий требуется проверить, тем предпочтительнее вместо `if-else` использовать инструкцию `switch`. При этом обычно ссылаются на удобочитаемость про-

граммного кода. Инструкция `if-else` выглядит более удобочитаемой при малом количестве условий, а инструкция `switch` – при большом количестве условий. Взгляните на следующий пример:

```
if (found){
    // некоторые операции
} else {
    // некоторые другие операции
}

switch(found){
    case true:
        // некоторые операции
        break;
    default:
        // некоторые другие операции
}
```

Оба фрагмента решают одну и ту же задачу, однако многие скажут, что инструкция `if-else` выглядит намного проще, чем `switch`. Однако с увеличением количества проверяемых условий мнение обычно меняется на противоположное:

```
if (color == "red"){
    // некоторые операции
} else if (color == "blue"){
    // некоторые операции
} else if (color == "brown"){
    // некоторые операции
} else if (color == "black"){
    // некоторые операции
} else {
    // некоторые операции
}

switch (color){
    case "red":
        // некоторые операции
        break;
case "blue":
        // некоторые операции
        break;
case "brown":
        // некоторые операции
        break;
case "black":
        // некоторые операции
        break;
default:
    // некоторые операции
}
```

В данном случае многие сочтут инструкцию `switch` более удобочитаемой.

Как оказывается, в большинстве случаев инструкция switch выполняется немного быстрее, чем инструкция if-else, но существенный прирост в скорости получается только при большом количестве условий. Основную разницу в производительности этих двух инструкций составляют более высокие накладные расходы на проверку дополнительных условий в инструкции if-else. Таким образом, естественное стремление использовать инструкцию if-else при малом количестве условий и инструкцию switch – при большом с точки зрения производительности является совершенно оправданным.

Вообще говоря, инструкцию if-else лучше использовать, когда имеются два дискретных значения или два диапазона значений, из которых приходится выбирать. Когда требуется проверить более двух дискретных значений, предпочтительнее использовать инструкцию switch.

Оптимизация инструкций if–else

При оптимизации инструкций if-else основной задачей всегда является уменьшение количества проверяемых условий, определяющих выбор направления потока выполнения. Простейшая оптимизация состоит в том, чтобы на первое место поставить проверки наиболее типичных условий. Рассмотрим следующий пример:

```
if (value < 5) {
    // некоторые операции
} else if (value > 5 && value < 10) {
    // некоторые операции
} else {
    // некоторые операции
}
```

Это решение является оптимальным, только если значение переменной value чаще всего оказывается меньше 5. Если обычно значение оказывается больше или равно 10, то каждый раз перед выбором правильной ветки будут проверяться два условия, что увеличит среднее время выполнения этой инструкции. Чтобы обеспечить максимальную скорость выполнения, проверка условий в инструкциях if-else всегда должна выполняться в порядке от наиболее вероятных к наименее вероятным.

Другой способ уменьшения количества проверок заключается в реорганизации инструкции if-else в последовательность вложенных инструкций if-else. Использование единственной большой инструкции if-else обычно приводит к увеличению общего времени выполнения из-за необходимости вычислять все условные выражения. Например:

```
if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
```

```
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 5){
    return result5;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else if (value == 9){
    return result9;
} else {
    return result10;
}
```

В такой инструкции `if-else` выполняется до 10 проверок. Это увеличит общее время выполнения, если значения переменной `value` равномерно распределены в диапазоне от 0 до 10. Чтобы уменьшить количество проверок, этот пример можно реорганизовать в последовательность вложенных инструкций `if-else`, например:

```
if (value < 6){

    if (value < 3){
        if (value == 0){
            return result0;
        } else if (value == 1){
            return result1;
        } else {
            return result2;
        }

    } else {

        if (value == 3){
            return result3;
        } else if (value == 4){
            return result4;
        } else {
            return result5;
        }
    }
} else {
    if (value < 8){
        if (value == 6){
            return result6;
        } else {
            return result7;
        }
```

```
        } else {
            if (value == 8){
                return result8;
            } else if (value == 9){
                return result9;
            } else {
                return result10;
            }
        }
    }
```

Улучшенная версия инструкции `if-else` выполняет не более четырех сравнений для любого значения `value`. Это было достигнуто за счет применения метода половинного деления, когда диапазон возможных значений разбивается на последовательность диапазонов с последующим перебором в каждом диапазоне. Среднее время выполнения этого примера при равномерном распределении возможных значений `value` в диапазоне от 0 до 10 почти в два раза меньше времени выполнения предыдущего примера использования инструкции `if-else`. Этот метод дает еще больший прирост производительности, когда требуется реализовать проверку на принадлежность диапазонам значений (в противоположность дискретным значениям, когда обычно лучше использовать инструкцию `switch`).

Поисковые таблицы

Наилучший принцип, которому стоит следовать при оптимизации условных инструкций `if-else` и `switch`, заключается в том, чтобы вообще избегать их использования. При необходимости проверить большое количество дискретных значений обе инструкции, `if-else` и `switch`, оказываются существенно медленнее, чем прием, основанный на использовании поисковых таблиц. Поисковую таблицу можно создать с помощью массива или обычного объекта, и доступ к данным в поисковой таблице выполняется намного быстрее, чем проверка условия в инструкции `if-else` или `switch`, особенно при большом количестве условий (рис. 4.1).

Поисковые таблицы не только быстрее инструкций `if-else` и `switch`, но также помогают повысить удобочитаемость программного кода, когда требуется проверить огромное количество дискретных значений. Например, инструкции `switch` становятся слишком громоздкими, когда требуется реализовать выбор из большого количества значений:

```
switch(value){
    case 0:
        return result0;
    case 1:
        return result1;
    case 2:
        return result2;
    case 3:
```

```
            return result3;
        case 4:
            return result4;
        case 5:
            return result5;
        case 6:
            return result6;
        case 7:
            return result7;
        case 8:
            return result8;
        case 9:
            return result9;
        default:
            return result10;
    }
```

Объем пространства, занимаемый этой инструкцией `switch` в программном коде, не соответствует ее важности. Всю эту структуру можно заменить, используя массив в качестве поисковой таблицы:

```
// определить массив результатов
var results = [result0, result1, result2, result3, result4, result5, result6,
               result7, result8, result9, result10]

// вернуть нужный результат
return results[value];
```

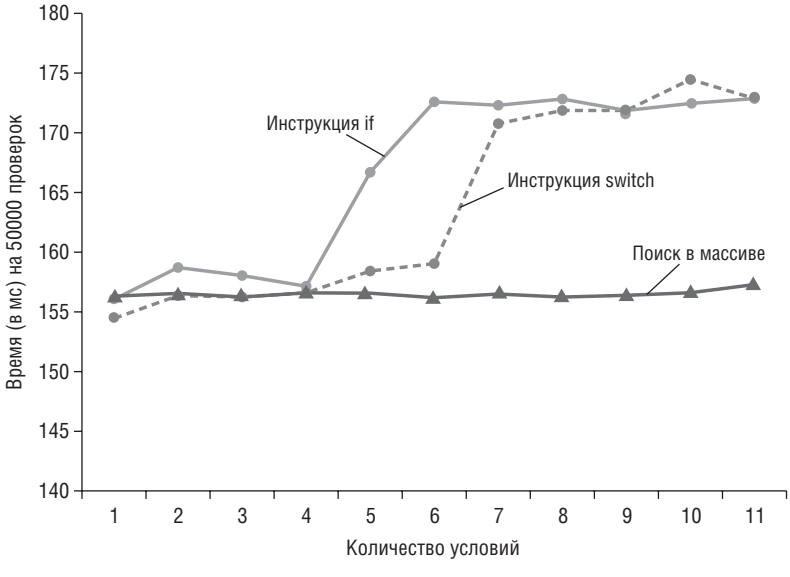


***Рис. 4.1.** Прием поиска элемента в массиве в сравнении с использованием инструкций if-else и switch в Internet Explorer 7*

Использование поисковой таблицы полностью избавляет от необходимости вычислять значения условных выражений. Операция выбора сводится к операции поиска элемента массива или члена объекта. В этом заключается главное преимущество поисковых таблиц: отсутствие необходимости вычислять значения условных выражений не увеличивает или, по крайней мере, незначительно увеличивает накладные расходы с ростом возможных вариантов выбора.

Поисковые таблицы особенно удобно использовать в ситуациях, когда каждому ключу соответствует единственное значение (как в предыдущем примере). Инструкция `switch` больше подходит для ситуаций, когда для каждого ключа требуется выполнить отдельную операцию или набор операций.

Рекурсия

Применение рекурсии обычно упрощает сложные алгоритмы. Существуют классические алгоритмы, реализация которых предполагает использование рекурсии. Например, функция вычисления факториала числа:

```
function factorial(n){
    if (n == 0){
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Проблема рекурсивных функций заключается в неточном или отсутствующем условии завершения, что ведет к увеличению времени выполнения, вызывающему подвисание пользовательского интерфейса. Кроме того, рекурсивные функции могут столкнуться с ограничением размера стека вызовов.

Ограниченность размера стека вызовов

Глубина рекурсии отличается в разных реализациях JavaScript и напрямую зависит от размера стека вызовов. За исключением Internet Explorer, в котором размер стека вызовов зависит от объема доступной памяти в системе, во всех остальных броузерах определяется статический размер стека вызовов. В большинстве современных броузеров стек вызовов имеет достаточно большой размер в сравнении со старыми версиями (в Safari 2, например, стек вызовов вмещал всего 100 записей). На рис. 4.2 показаны размеры стеков вызовов в основных броузерах.

Если размер стека вызовов превысит допустимое значение вследствие слишком глубокой рекурсии, броузер выведет одно из следующих сообщений:

- Internet Explorer: «Stack overflow at line x» (переполнение стека в строке x)
- Firefox: «Too much recursion» (слишком глубокая рекурсия)
- Safari: «Maximum call stack size exceeded» (размер стека вызовов превысил максимальную величину)
- Opera: «Abort (control stack overflow)» (прервано (из-за переполнения стека))

Chrome – единственный броузер, который не выводит сообщение, когда размер стека вызовов превышает максимальное значение.

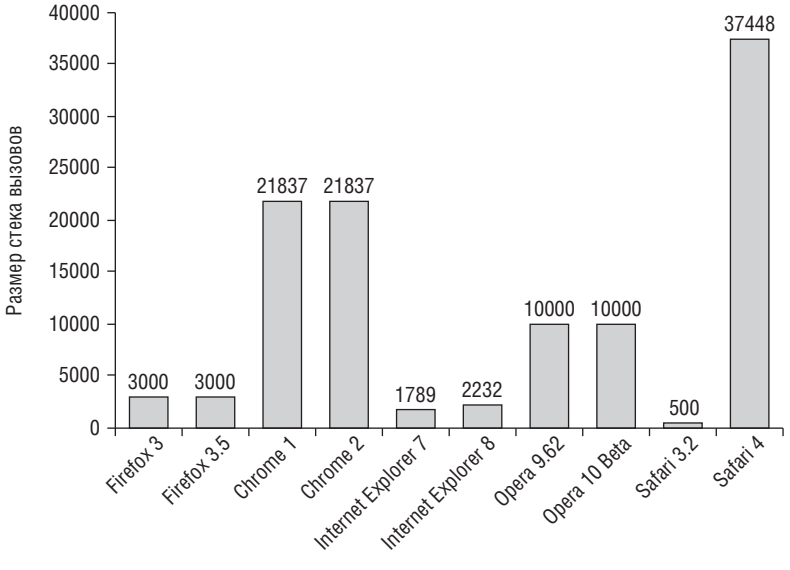


Рис. 4.2. *Размер стека вызовов в различных броузерах*

Самое, пожалуй, интересное в ошибках, связанных с переполнением стека, состоит в том, что в некоторых броузерах они принимают форму исключений JavaScript, которые можно перехватить с помощью инструкции `try-catch`. Тип исключения зависит от типа броузера. В Firefox – `InternalError`, в Safari и Chrome – `RangeError`, а в Internet Explorer возбуждается универсальное исключение `Error`. (Opera не возбуждает исключение, этот броузер просто останавливает работу интерпретатора JavaScript.) Это дает возможность обрабатывать подобные ошибки непосредственно в JavaScript-сценариях:

```
try {
    recurse();
} catch (ex){
    alert("Too much recursion!");
}
```

Если оставить такое исключение необработанным, оно всплывет, как любое другое исключение (в Firefox оно достигнет расширения Firebug и консоли ошибок; в Safari/Chrome будет выведено сообщение в консоли JavaScript). Исключение составляет Internet Explorer – IE не только выведет сообщение об ошибке в консоль JavaScript, но также отобразит диалог, напоминающий диалог `alert()`, с сообщением о переполнении стека.

Несмотря на возможность обработки этих ошибок в JavaScript-сценариях, делать это не рекомендуется. Ни один сценарий, который может исчерпать стек вызовов, не имеет права передаваться в эксплуатацию.

Шаблоны реализации рекурсии

Всякий раз когда приходится сталкиваться с ограниченностью размера стека вызовов, в первую очередь следует выявить все рекурсивные вызовы в сценарии. Для этого необходимо знать о двух основных шаблонах реализации рекурсии. Первый – прямой рекурсивный вызов, как показано в функции `factorial()` выше, когда функция вызывает себя сама. В общем виде этот шаблон выглядит так:

```
function recurse(){
    recurse();
}

recurse();
```

Такие рекурсивные вызовы легко обнаруживаются при возникновении ошибки. Второй, более сложный шаблон, основан на использовании двух функций:

```
function first(){
    second();
}

function second(){
    first();
}

first();
```

В этом шаблоне имеются две функции, вызывающие друг друга, образуя бесконечный цикл. Этот шаблон вызывает больше проблем и труднее поддается выявлению в больших сценариях.

Большая часть ошибок переполнения стека вызовов связана с одним из этих двух шаблонов реализации рекурсии. Часто причиной переполнения стека является ошибка в условии прекращения рекурсии, поэтому после выявления рекурсивных вызовов в первую очередь нужно обратить

внимание на реализацию этого условия. Если условие реализовано правильно, значит, сам алгоритм предусматривает слишком большую глубину рекурсии, чтобы ее можно было безопасно выполнить в броузере, и рекурсивный алгоритм следует заменить другим алгоритмом, основанным на итерациях, мемоизации или обоих приемах сразу.

Итерации

Любой рекурсивный алгоритм также может быть реализован с применением итераций. Итерационные алгоритмы обычно опираются на использование нескольких циклов, решающих различные части задачи, и таким образом порождают собственные проблемы производительности. Однако использование оптимизированных циклов взамен долго выполняющихся рекурсивных функций может способствовать повышению производительности благодаря тому, что накладные расходы на выполнение итераций в циклах несколько ниже накладных расходов на вызов функций.

В качестве примера рассмотрим алгоритм сортировки слиянием, который чаще всего реализуется с использованием рекурсии. Простейшая реализация этого алгоритма на языке JavaScript имеет следующий вид:

```
function merge(left, right){
    var result = [];

    while (left.length > 0 && right.length > 0){
        if (left[0] < right[0]){
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    return result.concat(left).concat(right);
}

function mergeSort(items){

    if (items.length == 1) {
        return items;
    }

    var middle = Math.floor(items.length / 2),
        left = items.slice(0, middle),
        right = items.slice(middle);

     return merge(mergeSort(left), mergeSort(right));
}
```

Реализация сортировки слиянием выглядит чрезвычайно просто, но проблема в том, что функция `mergeSort()` образует слишком глубокую

рекурсию. А рекурсивные функции часто могут вызывать ошибку переполнения стека в броузерах.

Появление ошибки переполнения стека не обязательно означает необходимость полного изменения алгоритма; это означает лишь, что рекурсия в данном случае является не самым лучшим способом реализации. Алгоритм сортировки слиянием можно также реализовать с применением итераций, например:

```
// использует функцию merge() из предыдущего примера
function mergeSort(items){

    if (items.length == 1) {
        return items;
    }

    var work = [];
    for (var i=0, len=items.length; i < len; i++){
        work.push([items[i]]);
    }
    work.push([]); // в случае нечетного числа элементов

    for (var lim=len; lim > 1; lim = Math.floor((lim+1)/2)){
        for (var j=0,k=0; k < lim; j++, k+=2){
            work[j] = merge(work[k], work[k+1]);
        }
        work[j] = [];// в случае нечетного числа элементов
    }

    return work[0];
}
```

Эта реализация функции `mergeSort()` выполняет ту же работу, что и предыдущая ее версия, но без использования рекурсии. Итерационная версия реализации сортировки слиянием может оказаться несколько медленнее рекурсивной, зато она не может привести к переполнению стека вызовов. Переход от рекурсивного алгоритма к итеративному – это одна из возможностей избежать появления ошибки переполнения стека.

Мемоизация

Предотвращение лишней работы – наилучший способ оптимизации производительности. Чем меньше работы выполняет программный код, тем быстрее он действует. Следуя этому правилу, есть смысл реорганизовать программный код, чтобы исключить ненужное многократное выполнение одних и тех же операций, которое ведет лишь к пустой трате времени. Мемоизация – это прием, позволяющий избежать повторного выполнения одной и той же работы за счет сохранения результатов предыдущих вычислений для использования в будущем, что делает его полезным для применения к рекурсивным алгоритмам.

Когда в процессе работы сценария рекурсивная функция вызывается многократно, она выполняет одну и ту же работу множество раз. Ярким примером такого многократного выполнения одной и той же работы рекурсивными функциями может служить функция `factorial()`, представленная в разделе «Рекурсия» выше. Рассмотрим следующий фрагмент:

```
var fact6 = factorial(6);
var fact5 = factorial(5);
var fact4 = factorial(4);
```

Этот фрагмент трижды вычисляет факториалы, в результате чего функция `factorial()` вызывается в общей сложности 18 раз. Худшее в этом фрагменте то, что вся необходимая работа была выполнена уже в первой строке. Поскольку факториал числа 6 – это результат умножения числа 6 на факториал числа 5, факториал числа 5 вычисляется здесь дважды. Более того, факториал числа 4 вычисляется трижды. Гораздо лучше было бы сохранять результаты вычислений и использовать их вместо того, чтобы производить повторные вычисления при каждом новом вызове функции.

Используя прием мемоизации, функцию `factorial()` можно переписать, как показано ниже:

```
function memfactorial(n){

    if (!memfactorial.cache){
        memfactorial.cache = {
            "0": 1,
            "1": 1
        };
    }

    if (!memfactorial.cache.hasOwnProperty(n)){
        memfactorial.cache[n] = n * memfactorial(n-1);
    }

    return memfactorial.cache[n];
}
```

Ключевой особенностью мемоизованной версии функции `factorial()` является создание объекта `cache`, в котором сохраняются результаты вычислений. Этот объект хранится в самой функции и предварительно заполняется результатами факториалов чисел 0 и 1. Перед вычислением факториала функция проверяет содержимое объекта `cache` на наличие готового результата. Отсутствие требуемого значения означает необходимость выполнить вычисления в первый раз и сохранить результат в объекте для последующего использования.

Эта функция используется точно так же, как и оригинальная функция `factorial()`:

```
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

Этот фрагмент также вычисляет факториалы трех разных чисел, но выполняет всего восемь вызовов функции `memfactorial()`. Поскольку все необходимые вычисления выполняются уже в первой строке, следующие две строки не производят рекурсивные вызовы функции благодаря тому, что она возвращает значения из кэша.

Для разных рекурсивных функций процедура мемоизации может несколько отличаться, но в общем и целом применяется один и тот же шаблон. Чтобы упростить мемоизацию функции, можно определить отдельную функцию `memoize()`, содержащую основную функциональность. Например:

```
function memoize(fundamental, cache){
    cache = cache || {};

    var shell = function(arg){
        if (!cache.hasOwnProperty(arg)){
            cache[arg] = fundamental(arg);
        }
        return cache[arg];
    };

    return shell;
}
```

Эта функция `memoize()` принимает два аргумента: функцию для мемоизации и необязательный объект, играющий роль кэша результатов. Объект кэша можно передавать, если необходимо указать некоторые предопределенные результаты; в противном случае будет создан новый объект кэша. Далее создается функция `shell()`, обертывающая оригинальную (`fundamental`) функцию, которая предотвратит повторное вычисление результата, если он уже имеется в кэше. Вызывающей программе возвращается эта функция `shell()`, которую можно вызывать непосредственно, например:

```
// мемоизация функции factorial
var memfactorial = memoize(factorial, { "0": 1, "1": 1 });

// вызов новой функции
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

Подобная обобщенная мемоизация не является такой же оптимальной, как выполненная вручную для каждой конкретной функции, потому что функция `memoize()` сохраняет результаты вызова функции, связывая их с определенными аргументами. Поэтому экономия на рекурсивных

вызовах будет получаться только при многократном вызове функции `shell()` с одними и теми же аргументами. По этой причине для функций, существенно влияющих на производительность, лучше не использовать обобщенное решение мемоизации, а реализовать мемоизацию вручную.

В заключение

Как и в других языках программирования, в языке JavaScript большое влияние на скорость выполнения оказывают организация программного кода и выбор алгоритмов. В отличие от других языков программирования, язык JavaScript обладает ограниченными ресурсами, поэтому применение приемов оптимизации в нем играет еще более важную роль.

- Циклы `for`, `while` и `do-while` имеют примерно одинаковую производительность, и поэтому ни один из них не обладает серьезными достоинствами или недостатками в сравнении с другими.
- Избегайте использования цикла `for-in`, если только не требуется выполнить обход заранее неизвестных свойств объекта.
- Лучший способ повысить производительность циклов – уменьшить количество операций, выполняемых в каждой итерации, и уменьшить общее количество итераций.
- В общем случае инструкция `switch` всегда выполняется быстрее, чем инструкция `if-else`, но ее использование не всегда является лучшим решением.
- Поисковые таблицы являются более быстрой альтернативой многократной проверке условий с помощью инструкции `if-else` или `switch`.
- Фиксированный размер стека вызовов в броузерах ограничивает допустимую глубину рекурсии; ошибка переполнения стека не позволит выполнить оставшийся программный код.
- Столкнувшись с ошибкой переполнения стека, следует подумать о переходе на итеративный алгоритм или использовать прием мемоизации, чтобы исключить возможность повторного выполнения одной и той же работы.

Чем больше объем выполняемого программного кода, тем больший выигрыш в производительности позволит получить применение этих стратегий.

5

Строки и регулярные выражения

Стивен Левитан (Steven Levithan)

Практически все программы на языке JavaScript работают со строками. Например, многие приложения, используя технологию Ajax, получают строки от сервера, преобразуют их для удобства в JavaScript-объекты и затем на их основе генерируют строки с разметкой HTML. Обычно приложениям приходится решать множество задач подобного рода, выполняя объединение, разбиение, переупорядочение, поиск, итерации и другие операции со строками. И по мере роста сложности веб-приложений все большая часть этой обработки выполняется в броузере.

Регулярные выражения в языке JavaScript представляют собой нечто большее, чем простой механизм обработки строк. Поэтому значительная часть данной главы нацелена на то, чтобы помочь вам разобраться в том, каким образом механизмы регулярных выражений[1] обрабатывают строки, и научить приемам создания регулярных выражений с учетом приобретенных знаний.

Кроме того, в этой главе рассказывается о наиболее быстрых способах объединения и усечения строк, совместимых со всеми броузерами, раскрываются приемы увеличения производительности регулярных выражений за счет уменьшения количества возвратов и дается множество других советов и рекомендаций по эффективной обработке строк и использованию регулярных выражений.

---

[1] Механизм – это программное обеспечение, которое обеспечивает работу регулярных выражений. Каждый броузер имеет собственный механизм регулярных выражений (или, если хотите, реализацию) со своими характеристиками производительности.

Конкатенация строк

Операция конкатенации строк может оказывать непредвиденно сильное влияние на производительность. В программах часто возникает необходимость сконструировать строку, последовательно добавляя новые символы в цикле (например, при создании HTML-таблиц или конструировании XML-документов), но такой вид обработки строк печально известен своей низкой производительностью в некоторых броузерах.

Так как же можно оптимизировать выполнение этих операций? Начнем с того, что имеется несколько способов объединения строк (табл. 5.1).

Таблица 5.1. Методы конкатенации строк

Метод	Пример
Оператор +	`str = "a" + "b" + "c";`

Оператор +=	`str = "a";`
	`str += "b";`
	`str += "c";`

`array.join()`	`str = ["a", "b", "c"].join("");`
`string.concat()`	`str = "a";` `str = str.concat("b", "c");`

Все эти методы являются достаточно быстрыми, когда выполняется конкатенация лишь нескольких строк и нечасто, поэтому при редком использовании следует выбирать наиболее подходящий вам способ. Однако с ростом длины и количества объединяемых строк проявляются преимущества отдельных способов.

Операторы плюс (+) и плюс-равно (+=)

Эти операторы предоставляют самый простой способ конкатенации строк, который практически все броузеры, кроме IE7 и более ранних версий, оптимизируют достаточно хорошо, чтобы исключить необходимость рассматривать другие альтернативы. Тем не менее существуют приемы, позволяющие еще больше повысить эффективность этих операторов.

Рассмотрим для начала простой пример. Ниже приводится типичный способ объединения строк:

```
str += "one" + "two";
```

Эта инструкция выполняет четыре действия:

1. Создает временную строку в памяти.
2. Объединенное значение `"onetwo"` сохраняется во временной строке.

3. Временная строка объединяется с текущим значением переменной `str`.
4. Результат присваивается переменной `str`.

Это лишь примерное описание того, как броузеры реализуют данную операцию, но достаточно близкое к действительности.

Следующий фрагмент не создает временную строку (исключаются шаги 1 и 2 в списке выше) за счет прямого добавления строк в конец переменной `str` с помощью двух отдельных инструкций. В большинстве броузеров этот способ работает на 10–40% быстрее:

```
str += "one";
str += "two";
```

На практике такую же производительность можно получить, используя единственную инструкцию, как показано ниже:

```
str = str + "one" + "two";
// эквивалентна инструкции str = ((str + "one") + "two")
```

Здесь не используется временная строка, потому что выражение правее оператора присваивания начинается с переменной `str`, используемой в качестве основы, и добавление производится по одной строке за раз, в направлении слева направо. Если выполнить конкатенацию в другом порядке (например, `str = "one" + str + "two"`), оптимизация будет утеряна. Это обусловлено способом выделения памяти в броузерах при объединении строк. Все броузеры, кроме IE, пытаются увеличить выделение памяти под строку, указанную слева от оператора присваивания, и просто скопировать вторую строку в ее конец (рис. 5.1). Когда базовая строка является крайней левой в выражении, в циклах это позволяет избежать многократного копирования постоянно увеличивающейся базовой строки.

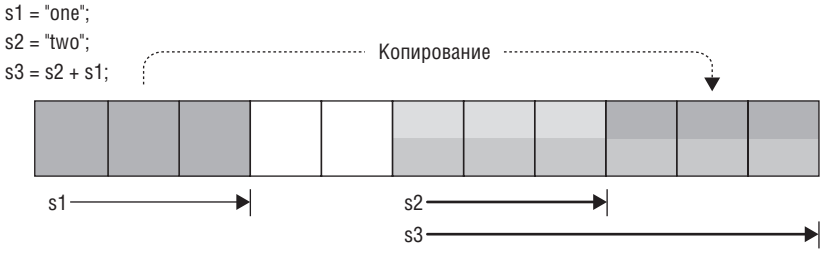


Рис. 5.1. *Пример использования памяти при конкатенации строк: для создания строки s3 строка s1 копируется в конец строки s2; базовая строка s2 не копируется*

Эти приемы не действуют в IE. В IE8 они дают лишь незначительный (если вообще дают) прирост производительности и даже замедляют выполнение в IE7 и в более ранних версиях. Это обусловлено внутренней

реализацией операции конкатенации в IE. Реализация операции конкатенации в IE8 просто сохраняет ссылки на существующие части, составляющие новую строку. Впоследствии (при фактическом использовании объединенной строки) части будут скопированы в новую, «настоящую» строку, которая заменит ранее сохраненные ссылки, чтобы эту сборку не приходилось выполнять при каждом обращении к строке.

Реализация в IE8 будет с успехом проходить искусственные тесты производительности – кажущаяся скорость конкатенации будет выше действительной, – если принудительно не выполнять фактическую конкатенацию после сборки строки. Это можно сделать, например, применением метода `toString()` к полученной строке, проверкой свойства `length` или вставкой строки в дерево DOM.

В IE7 и в более ранних версиях используется более медленная реализация конкатенации, которая всегда копирует пару объединяемых строк в новое место в памяти. Фатальное влияние такой реализации на производительность будет продемонстрировано ниже в разделе «Объединение элементов массива». В версиях IE, предшествовавших IE8, следование советам этого раздела может ухудшить производительность, так как реализация конкатенации в этих версиях броузера показывает более высокую производительность при объединении коротких строк перед слиянием их с длинной базовой строкой (благодаря тому, что исключается необходимость несколько раз копировать длинную строку). Например, при выполнении инструкции `largeStr = largeStr + s1 + s2` IE7 и более ранние версии должны будут скопировать длинную строку `largeStr` дважды – сначала, чтобы объединить ее со строкой `s1`, а затем – со строкой `s2`. Инструкция `largeStr += s1 + s2`, напротив, сначала выполнит конкатенацию двух коротких строк, а затем объединит результат с длинной строкой `largeStr`. Создание промежуточной строки `s1 + s2` гораздо меньше сказывается на производительности, чем двукратное копирование длинной строки.

Firefox и свертка на этапе компиляции

Когда в инструкции присваивания все объединяемые строки представлены константами, Firefox автоматически объединяет их на этапе компиляции. Следующий пример демонстрирует, как это происходит:

```
function foldingDemo() {
    var str = "compile" + "time" + "folding";
    str += "this" + "works" + "too";
    str = str + "but" + "not" + "this";
}

alert(foldingDemo.toString());

/* В Firefox этот программный код будет выглядеть так:
function foldingDemo() {
```

```
    var str = "compiletimefolding";
    str += "thisworkstoo";
    str = str + "but" + "not" + "this";
} */
```

Когда происходит подобная свертка строк, во время выполнения не создается никаких промежуточных строк, а потребление времени и памяти на их конкатенацию уменьшается до нуля. Здорово, когда такое случается, но эта особенность не слишком помогает на практике, потому что гораздо чаще строки конструируются из данных, полученных во время выполнения, а не из констант.

Инструмент YUI Compressor выполняет такую оптимизацию во время сборки. Подробнее об этом инструменте рассказывается в разделе «Уменьшение сценариев JavaScript» в главе 9.

Слияние элементов массива

Метод `Array.prototype.join` объединяет все элементы массива в одну строку и принимает строку-разделитель, которая должна быть вставлена между элементами. Если передать методу пустую строку-разделитель, он выполнит простую конкатенацию всех элементов массива.

В большинстве броузеров объединение элементов массива выполняется медленнее, чем другие способы конкатенации, но важно учитывать, что данный способ является единственным эффективным средством конкатенации большого количества строк в IE7 и в более ранних версиях.

Следующий пример демонстрирует проблему, связанную с производительностью, которая решается с помощью операции объединения элементов массива:

```
var str = "I'm a thirty-five character string.",
    newStr = "",
    appends = 5000;

while (appends--) {
    newStr += str;
}
```

Этот программный код объединяет 5000 строк по 35 символов. На рис. 5.2[1] показано, как долго выполняется этот тест в IE7. В первом испытании выполнялось 5000 операций конкатенации, и это число увеличивалось в каждом последующем испытании.

Простой алгоритм конкатенации строк, реализованный в IE7, многократно выделяет память и копирует все более и более увеличивающиеся

[1] Данные для диаграмм на рис. 5.2 и 5.3 были получены путем усреднения результатов 10 испытаний в IE7, в Windows XP, запущенной в виртуальной машине со скромными ресурсами (ЦПУ: 2 Ггц Core 2 Duo; ОЗУ: 1 Гбайт).

строки в цикле. В результате этого время выполнения операции и потребление памяти возрастают в квадратичной прогрессии.

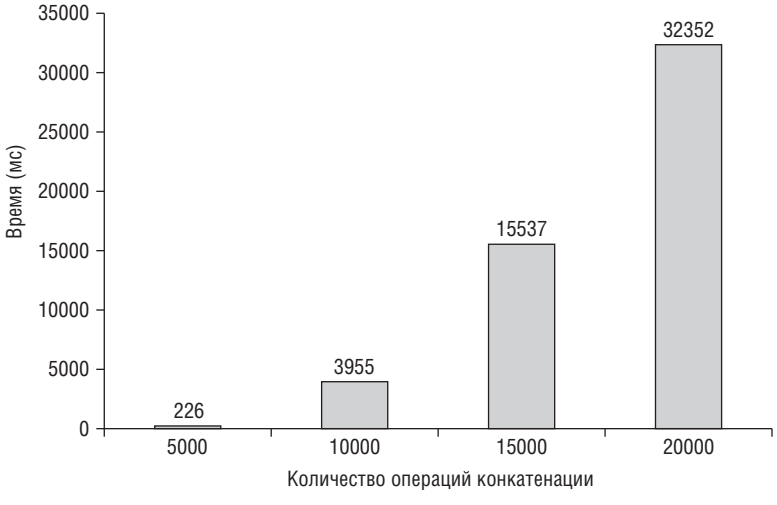


Рис. 5.2. *Время выполнения операций конкатенации в IE7 с использованием оператора +=*

Но есть и хорошие новости – все остальные современные броузеры (включая IE8) проходят этот тест намного лучше и не показывают квадратичную сложность алгоритма, которая в данном случае является просто убийственной. Рассматриваемый пример демонстрирует, какое влияние оказывает на производительность, казалось бы, простая конкатенация строк; даже 226 мс на 5000 операций конкатенации – это существенная потеря производительности, которую хорошо было бы уменьшить, насколько это возможно, а блокирование броузера более чем на 32 секунды для выполнения конкатенации 20000 коротких строк является совершенно недопустимым для любого приложения.

Теперь рассмотрим следующий тест, генерирующий ту же строку посредством слияния элементов массива:

```
var str = "I'm a thirty-five character string.",
    strs = [],
    newStr,
    appends = 5000;

while (appends--) {
    strs[strs.length] = str;
}

newStr = strs.join("");
```

На рис. 5.3 показаны результаты выполнения этого теста в IE7.

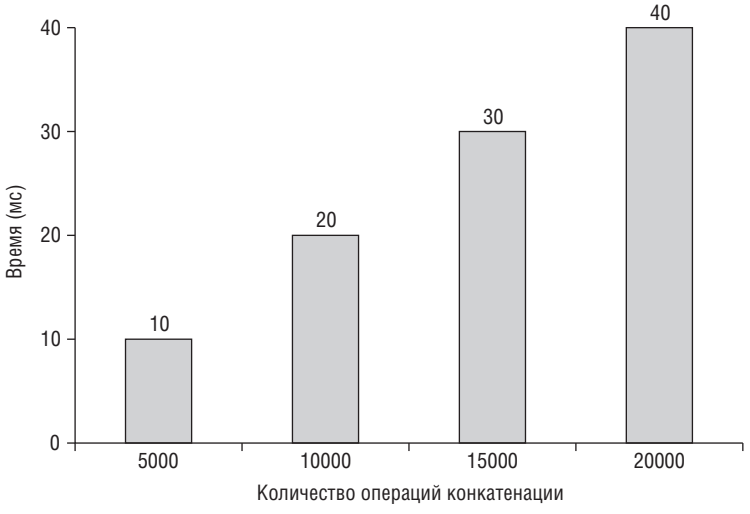


Рис. 5.3. *Время выполнения конкатенации строк в IE7 посредством слияния элементов массива*

Такое существенное увеличение производительности является результатом отказа от многократного выделения памяти и копирования все более увеличивающихся строк. Выполняя операцию слияния элементов массива, броузер сразу выделяет память, достаточную для хранения полной строки, и никогда не копирует одни и те же части строки результата более одного раза.

String.prototype.concat()

Встроенный строковый метод `concat()` принимает произвольное количество аргументов и добавляет их в конец строки, относительно которой этот метод был вызван. Этот способ конкатенации строк считается наиболее гибким, потому что его можно использовать для добавления одной строки, нескольких строк или целого массива строк.

```
// добавить одну строку
str = str.concat(s1);

// добавить три строки
str = str.concat(s1, s2, s3);

// добавить все строки из массива, используя массив как список аргументов
str = String.prototype.concat.apply(str, array);
```

К сожалению, в большинстве случаев метод `concat()` работает несколько медленнее простых операторов + и += и может оказаться существенно медленнее в IE, Opera и Chrome. Кроме того, несмотря на то что применение метода `concat()` для слияния всех строк в массиве напоминает

операцию слияния элементов массива, обсуждавшуюся выше, он обычно работает медленнее (исключение составляет броузер Opera) и страдает той же проблемой катастрофического падения производительности, что и операторы + и +=, при конструировании длинных строк в IE7 и в более ранних версиях.

Оптимизация регулярных выражений

Небрежно сработанные регулярные выражения могут стать причиной падения производительности (ниже в разделе «Исключение возвратов» приводится несколько примеров, демонстрирующих, насколько серьезным может быть это падение), однако существует множество способов повысить эффективность регулярных выражений. Простой факт совпадения двух регулярных выражений с одним и тем же текстом не означает, что они выполняются с одинаковой скоростью.

На эффективность регулярных выражений влияет множество факторов. Самое большое значение имеет текст, к которому применяется регулярное выражение, потому что на анализ текста, имеющего частичные совпадения с регулярным выражением, тратится больше времени, чем на анализ текста, не имеющего таких совпадений. Кроме того, в разных броузерах реализации механизмов регулярных выражений предусматривают свои оптимизации.[1]

Оптимизация регулярных выражений – довольно обширная тема, требующая учета различных нюансов. В этом разделе невозможно охватить все приемы, но тех, что рассмотрены, вполне достаточно, чтобы дать представление об узких местах в производительности регулярных выражений и показать общее направление к овладению искусством конструирования эффективных регулярных выражений.

Обратите внимание: в этом разделе предполагается, что вы уже имеете некоторый опыт работы с регулярными выражениями и вас прежде всего интересует вопрос повышения их эффективности. Если вы не знакомы с регулярными выражениями или вам требуется освежить в памяти основы их применения, обращайтесь к соответствующим ресурсам в Сети или к печатным изданиям. Для тех, кто предпочитает учиться на практических примерах, рекомендуем книгу «Regular Expressions Cookbook»[2] (*http://oreilly.com/catalog/9780596520694/*) (O'Reilly), написанная Яном Гойвертсом (Jan Goyvaerts) и Стивеном Левитаном (Steven Levithan) (это я!), которая демонстрирует приемы создания регулярных выражений в программах на языке JavaScript и некоторых других языках программирования.

---

[1] Вследствие этого незначительные изменения могут ускорить работу регулярного выражения в одном броузере и замедлить в другом.

[2] Ян Гойвертс и Стивен Левитан «Регулярные выражения. Сборник рецептов». – Пер. с англ. – СПб.: Символ-Плюс, 2009.

Как работают регулярные выражения

Занимаясь вопросом повышения эффективности регулярных выражений, важно понимать, как они действуют. Ниже приводится краткий список основных шагов, выполняемых механизмами регулярных выражений:

Шаг 1: компиляция

При создании объекта регулярного выражения (из литерала регулярного выражения или с помощью конструктора `RegExp()`) броузер проверяет шаблон на наличие ошибок и затем преобразует его в промежуточный код, который используется для фактического сопоставления. Если присвоить регулярное выражение переменной, можно избежать многократного выполнения этого этапа для одного и того же шаблона.

Шаг 2: установка начальной позиции

При применении регулярного выражения сначала определяется позиция в испытуемой строке, откуда следует начать поиск. Первоначально эта позиция совпадает с началом строки или определяется значением свойства `lastIndex` объекта регулярного выражения[1], но после возврата из шага 4 (из-за неудачной попытки сопоставления) позиция смещается на один символ дальше.

Оптимизации, встраиваемые производителями броузеров в механизмы регулярных выражений, способны помочь избежать массы ненужной работы на этом этапе, выявляя операции, которые можно пропустить. Например, если регулярное выражение начинается с символа ^, IE и Chrome обычно способны определить, что совпадение не может быть найдено не с начала строки, и отказаться от ненужных операций поиска, начинающихся с любых других позиций в строке. Другой пример: если любые возможные совпадения должны содержать в третьей позиции символ `x`, оптимизированная реализация способна определить этот факт, быстро отыскать следующий символ `x` и установить начальную позицию на две позиции ближе к началу строки (такую оптимизацию, например, предусматривают последние версии Chrome).

Шаг 3: сопоставление каждой лексемы в регулярном выражении

После определения начальной позиции механизм регулярных выражений начинает сопоставлять текст с шаблоном. Когда сопоставление с какой-либо лексемой в шаблоне терпит неудачу, выполняется

[1] Значение свойства `lastIndex` используется только методами `exec()` и `test()` объекта регулярного выражения и только когда регулярное выражение сконструировано с флагом `/g` (global – глобальный поиск). Неглобальные регулярные выражения, а также любые регулярные выражения, передаваемые строковым методам `match()`, `replace()`, `search()` и `split()`, всегда начинают поиск с начала целевой строки.

возврат в предыдущую позицию в уже найденном частичном совпадении и производится попытка найти другие варианты совпадения с регулярным выражением.

Шаг 4: успех или неудача

Если в текущей позиции будет обнаружено полное совпадение, механизм регулярных выражений объявляет об успехе. Если были опробованы все возможные варианты, но совпадение так и не было найдено, происходит возврат к шагу 2 и попытка повторяется, начиная со следующего символа в строке. Только после того, как этот цикл будет выполнен для каждого символа в строке (а также для позиции за последним символом) и совпадение не будет найдено, механизм регулярных выражений объявляет о неудаче.

Знание этого процесса поможет вам понять проблемы, влияющие на производительность регулярных выражений. В следующем разделе более подробно рассматривается ключевая особенность процесса сопоставления на шаге 3 – возвраты.

Возвраты

В большинстве современных реализаций (включая механизм регулярных выражений в языке JavaScript) возврат является основополагающей составляющей процесса сопоставления. Кроме того, именно возвраты обеспечивают значительную долю выразительности и гибкости регулярных выражений. Однако при неосторожном обращении возвраты могут приводить к существенному удорожанию вычислений. Несмотря на то что возвраты являются лишь частью общего уравнения, определяющего уровень производительности, знание того, как они действуют и как уменьшить их количество, является, пожалуй, основным ключом к созданию эффективных регулярных выражений. Поэтому данной теме будет уделено большое внимание в следующих нескольких разделах.

В процессе сопоставления механизм регулярных выражений постепенно движется по испытуемой строке, пытаясь отыскать совпадение в каждой позиции в строке, шагая по элементам регулярного выражения слева направо. Для каждого квантификатора или варианта выбора[1] должно быть принято решение, как продолжать сопоставление. Для квантификатора (такого как `*`, `+?` или `{2,}`) механизм регулярных выражений должен решить, сколько дополнительных символов следует включить в совпадение, а для вариантов выбора (перечисляемых с помощью оператора `|`) он должен найти совпадение с одним из имеющихся вариантов.

[1] Несмотря на то что классы символов, такие как `[a-z]`, и сокращенные формы их записи, такие как `\s` или «точка», допускают различные варианты выбора, они реализованы без использования возвратов и потому не оказывают такого же существенного влияния на производительность.

Каждый раз когда требуется принять такое решение, механизм регулярных выражений запоминает другие возможные варианты, чтобы при необходимости вернуться к ним позже. Если сопоставление с выбранным вариантом оказалось успешным, механизм регулярных выражений продолжит свое движение по шаблону, и в случае успеха сопоставления с остатком шаблона процесс поиска завершится. Но если совпадение с выбранным вариантом не будет найдено или сопоставление потерпит неудачу где-то далее в регулярном выражении, произойдет возврат к последней позиции принятия решения, где остались не опробованные варианты, и будет выполнена попытка найти совпадение с другим вариантом. Так продолжается, пока не будет найдено совпадение или пока безуспешно не будут опробованы все возможные перестановки квантификаторов и вариантов выбора, после чего весь процесс поиска повторится со следующего символа в испытуемой строке.

Варианты выбора и возвраты

Ниже приводится пример, демонстрирующий влияние вариантов выбора на производительность.

```
/h(ello|appy) hippo/.test("hello there, happy hippo");
```

Этому регулярному выражению соответствует текст «hello hippo» или «happy hippo». Сопоставление начинается с поиска символа `h`, который обнаруживается сразу же в начале испытуемой строки. Следующее подвыражение `(ello|appy)` предполагает два варианта совпадения. Сначала выбирается самый левый вариант (перебор вариантов всегда выполняется слева направо) и проверяется совпадение `ello` со следующими символами в строке. Механизм регулярных выражений обнаруживает совпадение с этой подстрокой и со следующим далее пробелом. Однако дальнейшее сопоставление терпит неудачу, потому что символ `h` в слове `hippo` в шаблоне не совпадает со следующим символом `t` в строке. Но механизм регулярных выражений не останавливается на этом, так как были опробованы не все варианты, поэтому он возвращается к точке принятия последнего решения (сразу за первым совпавшим символом `h`) и пытается найти совпадение со вторым вариантом. Совпадение с этим вариантом не обнаруживается, и, так как вариантов больше не осталось, механизм регулярных выражений принимает решение, что совпадение в начале строки отсутствует и переходит ко второму символу, повторяя попытку. Здесь он не находит совпадение с символом `h` и продолжает поиск, пока не встретит 14-й символ, где обнаруживается совпадение с символом `h` в слове «happy». Далее снова начинается сопоставление с вариантами выбора. На этот раз совпадение с вариантом `ello` отсутствует, зато после возврата обнаруживается совпадение со вторым вариантом, и сопоставление продолжается, пока не будет выявлено совпадение с остатком строки «happy hippo» (рис. 5.4). На этом поиск завершается успехом.

Рис. 5.4. *Пример выполнения возвратов при наличии вариантов выбора*

Повторения и возвраты

Следующий пример демонстрирует, как выполняются возвраты при наличии квантификаторов, задающих повторения.

```
var str = "<p>Para 1.</p>" +
          "<img src='smiley.jpg'>" +
          "<p>Para 2.</p>" +
          "<div>Div.</div>";

/<p>.*<\/p>/i.test(str);
```

Первые три символа в этом регулярном выражении соответствуют литералам символов `<p>` в испытуемом тексте. Далее в выражении следует комбинация `.*`. Точке соответствует любой символ, кроме символа конца строки, а максимальный (или «жадный») квантификатор «звездочка» допускает совпадение с точкой ноль и более раз – настолько много, насколько это возможно. Поскольку в испытуемом тексте отсутствуют символы конца строки, эта комбинация «проглатывает» остаток текста! Однако в регулярном выражении имеются и другие символы, которым должен соответствовать испытуемый текст, поэтому далее механизм регулярных выражений пытается отыскать символ `<`. Но так как уже достигнут конец строки, сопоставление терпит неудачу и выполняется возврат на один символ назад. Возвраты в поисках совпадения с символом `<` продолжаются, пока в тексте не будет найден символ `<` в начале тега `</div>`. Затем выполняется попытка найти совпадение с символом `\/` (экранированный символ слэша) со следующим за ним символом `p`, которая терпит неудачу. Тогда снова выполняется возврат, и процесс продолжается, пока не будет найдена последовательность символов `</p>` в конце второго абзаца. После этого возвращается совпавшая часть, простираю-

щаяся от начала первого абзаца до конца последнего, что, вероятно, не совсем то, что подразумевалось при создании регулярного выражения.

Чтобы обеспечить соответствие регулярного выражения отдельным абзацам, следует заменить максимальный квантификатор `*` минимальным (или «нежадным») квантификатором `*?`. При использовании минимальных квантификаторов возвраты выполняются противоположным образом. Когда механизм регулярных выражений встретит в шаблоне `/<p>.*?<\/p>/` комбинацию `.*?`, он сначала постарается вообще пропустить ее и сразу отыскать совпадение с `<\/p>`. Это обусловлено тем, что квантификатор `*?` соответствует минимальному числу повторений предыдущего элемента, от нуля и более раз, а минимальным является ноль повторений. Однако когда сопоставление со следующим символом `<` в шаблоне терпит неудачу, выполняется возврат и повторяется попытка сопоставить комбинацию `.*?` со следующим минимальным количеством символов – одним. Эти возвраты продолжаются, пока не будет найдено совпадение с подвыражением `<\/p>`, следующим за квантификатором, которое обнаруживается в конце первого абзаца.

Таким образом, даже если бы в испытуемом тексте имелся всего один абзац и обе версии данного регулярного выражения, «жадная» и «нежадная», возвращали бы эквивалентные результаты, они отыскивали бы совпадение разными способами (рис. 5.5).

Исключение возвратов

Когда применение регулярного выражения останавливает работу браузера на секунды, минуты или еще больше, проблема, скорее всего, связана с самым тяжелым случаем бесконтрольного возрастания числа возвратов. Чтобы продемонстрировать эту проблему, рассмотрим следующее регулярное выражение, которое должно соответствовать HTML-файлу целиком. Регулярное выражение было разбито на несколько строк, чтобы уместилось по ширине странницы. В отличие от большинства других диалектов регулярных выражений, диалект, реализованный в языке JavaScript, не позволяет «точке» соответствовать любому символу, включая символы конца строки, поэтому в данном примере совпадение с любым символом описывается комбинацией `[\s\S]`.

```
/<html>[\s\S]*?<head>[\s\S]*?<title>[\s\S]*?<\/title>[\s\S]*?<\/head>
[\s\S]*?<body>[\s\S]*?<\/body>[\s\S]*?<\/html>/
```

Это регулярное выражение прекрасно справляется с работой, когда испытуемый текст является допустимой разметкой HTML, но все фатально меняется, когда в тексте отсутствует один или более обязательных тегов. В случае отсутствия тега `</html>`, например, совпадение с последней комбинацией `[\s\S]*?` распространяется до конца текста, и тогда, вместо того чтобы завершить сопоставление с отрицательным результатом, механизм регулярных выражений начинает возвращаться к каждой из предыдущих комбинаций `[\s\S]*?`, расширяя области совпадений

с ними. Сначала он пытается расширить область, охватываемую второй с конца комбинацией `[\s\S]*?`, распространяя ее на тег `</body>`, который прежде совпал с шаблоном `<\/body>` в регулярном выражении, и продолжает расширять ее в поисках второго тега `</body>`, пока снова не достигнет конца строки. Когда все эти попытки не увенчаются успехом, будет выполнена попытка расширить область, охватываемую третьей с конца комбинацией `[\s\S]*?`, и т. д.

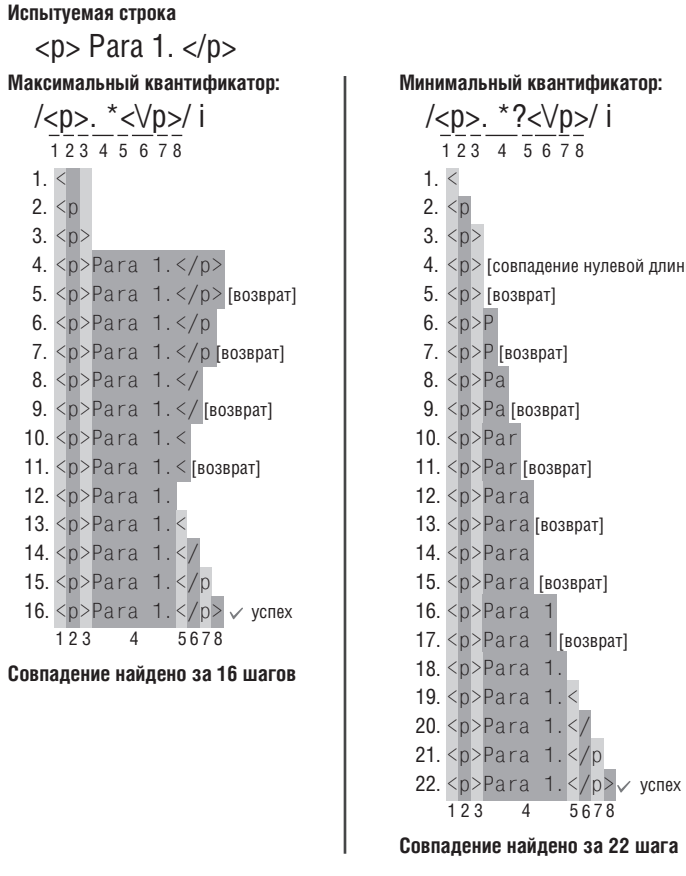


Рис. 5.5. *Пример выполнения возвратов при использовании максимального и минимального квантификаторов*

Решение: уточнение шаблона

Один из способов решения подобных проблем заключается в максимальном уточнении, какие символы могут находиться между обязательными разделителями. Возьмем, например, шаблон `".*?"`, который соответствует строке в двойных кавычках. Заменив комбинацию `.*?`, допускаю-

щую слишком широкую интерпретацию, более точной комбинацией [^"\r\n]*, можно исключить вероятность возвратов, которые вынудят «точку» совпасть с двойной кавычкой и расширить ее область совпадения больше, чем это необходимо.

В примере с разметкой HTML применить это решение не так просто. Здесь нельзя использовать инвертированный символьный класс, такой как [^<], вместо [\s\S], потому что между искомыми тегами может находиться множество других тегов. Однако того же эффекта можно добиться, повторяя несохраняющую группу, содержащую негативную опережающую проверку (блокирующую распространение на следующий обязательный тег) и метапоследовательность [\s\S] (любой символ). Это гарантирует завершение поиска в случае отсутствия обязательного тега в любой промежуточной позиции и, что более важно, предотвратит распространение областей совпадения с шаблонами [\s\S] на теги, обнаруженные негативной опережающей проверкой. Ниже показано, как выглядит регулярное выражение, составленное с применением описываемого подхода:

```
/<html>(?:(?!<head>)[\s\S])*<head>(?:(?!<title>)[\s\S])*<title>
(?:(?!<\/title>)[\s\S])*<\/title>(?:(?!<\/head>)[\s\S])*<\/head>
(?:(?!<body>)[\s\S])*<body>(?:(?!<\/body>)[\s\S])*<\/body>
(?:(?!<\/html>)[\s\S])*<\/html>/
```

Несмотря на то что это регулярное выражение исключает вероятность бесконтрольного увеличения числа возвратов и обеспечивает своевременное прерывание поиска при сопоставлении с неполными строками HTML, оно не сможет победить в гонке за эффективностью. Повторение опережающей проверки для каждого сопоставляемого символа, как в данном примере, само по себе неэффективно и существенно замедляет поиск в случае удачного сопоставления. Такой подход можно использовать при сопоставлении с короткими строками, но в данном случае, где при сопоставлении с HTML-файлом может потребоваться выполнить тысячи опережающих проверок, следует использовать другое, более удачное решение. Это решение, основанное на небольшой хитрости, описывается в следующем разделе.

Имитация атомарной группировки с помощью опережающей проверки и обратных ссылок

Некоторые диалекты регулярных выражений, включая .NET, Java, Oniguruma, PCRE и Perl, поддерживают особенность под названием *атомарная группировка*. Атомарная группировка – записывается как (?>…), где троеточие представляет произвольное регулярное выражение, – является несохраняющей группировкой со специальными характеристиками. Когда механизм регулярных выражений выходит за пределы атомарной группы, он забывает все позиции возврата, имеющиеся внутри группы. Эта особенность обеспечивает более эффективное решение проблемы возвратов в регулярном выражении, соответствующем целому

HTML-файлу: если заключить каждую комбинацию `[\s\S]*?` со следующим за ней HTML-тегом в атомарную группу, то при обнаружении обязательного HTML-тега найденное к этому моменту совпадение будет фактически зафиксировано. Если сопоставление со следующей частью регулярного выражения потерпит неудачу, у механизма регулярных выражений не останется позиций возврата для квантификаторов внутри атомарной группы, благодаря чему комбинация `[\s\S]*?` не будет расширяться за пределы области уже найденного совпадения.

Отличное решение, но JavaScript не поддерживает атомарную группировку и не предоставляет других инструментов, позволяющих исключить ненужные возвраты. Однако, как оказывается, атомарную группировку можно реализовать, задействовав малоизвестную особенность опережающей проверки: опережающая проверка сама являются атомарной группой.[1] Разница лишь в том, что опережающие проверки не поглощают символы при совпадении; они просто проверяют факт совпадения с содержащимися в них шаблонами. Однако такое поведение можно изменить, поместив шаблон опережающей проверки в сохраняющую группу и добавив обратную ссылку за пределами опережающей проверки. Ниже показано, как выглядит реализация этого приема:

```
(?=(шаблон, который требуется сделать атомарным))\1
```

Эту конструкцию можно использовать в любом шаблоне, где потребуется применить атомарную группировку. Только не забывайте следить за номерами обратных ссылок, если регулярное выражение содержит более одной сохраняющей группы.

Ниже показано, как выглядит регулярное выражение, соответствующее HTML-файлу, после применения этого решения:

```
/<html>(?=([\s\S]*?<head>))\1(?=([\s\S]*?<title>))\2(?=([\s\S]*?
<\/title>))\3(?=([\s\S]*?<\/head>))\4(?=([\s\S]*?<body>))\5
(?=([\s\S]*?<\/body>))\6[\s\S]*?<\/html>/
```

Теперь, если в тексте будет отсутствовать завершающий тег `</html>` и совпадение с последней комбинацией `[\s\S]*?` распространится до конца строки, поиск будет завершаться немедленно, из-за отсутствия позиций для возврата назад. Каждый раз когда механизм регулярных выражений будет обнаруживать в испытуемом тексте промежуточный тег и выходить за границы опережающей проверки, он будет забывать позиции возврата, сохраненные внутри опережающей проверки. Обратные ссылки, следующие за опережающими проверками, просто будут поглощать символы, совпавшие с опережающими проверками, включая их в общий результат сопоставления.

---

[1] Можно смело использовать эту особенность опережающих проверок, потому что она поддерживается всеми основными диалектами регулярных выражений и явно требуется стандартами ECMAScript.

Исключение возвратов и вложенные квантификаторы

Так называемые вложенные квантификаторы всегда требуют особого внимания, чтобы исключить вероятность бесконтрольного увеличения числа возвратов. Квантификатор считается вложенным, когда он присутствует внутри группы, которая сама повторяется другим квантификатором (например, `(x+)*`).

Сами по себе вложенные квантификаторы не представляют большой угрозы производительности. Однако при неосторожном обращении они легко могут создать огромное количество вариантов сопоставления текста с внутренним и внешним квантификаторами.

В качестве примера предположим, что потребовалось написать регулярное выражение, соответствующее HTML-тегам, и в результате был создан следующий шаблон:

```
/<(?:[^>"']|"[^"]*"|'[^']*')*>/
```

Это, пожалуй, слишком упрощенный шаблон, потому что он предусматривает не все варианты допустимой и недопустимой разметок, но его может оказаться вполне достаточно при обработке фрагментов допустимой разметки HTML. Он имеет определенные преимущества перед еще более упрощенными регулярными выражениями, такими как `/<[^>]*>/`, так как учитывает возможность появления символов `>` внутри значений атрибутов. Это обеспечивается второй и третьей альтернативами в несохраняющей группе, которой соответствует весь список атрибутов со значениями в двойных или одиночных кавычках, внутри которых допускается указывать любые символы, кроме кавычек соответствующего типа.

На данный момент не наблюдается никакой угрозы бесконтрольного увеличения числа возвратов, несмотря на наличие вложенных квантификаторов `*`. Вторая и третья альтернативы соответствуют точно одной строке в кавычках на каждое повторение группы, поэтому с увеличением испытуемой строки количество позиций возврата возрастает линейно.

Но взгляните на первую альтернативу в несохраняющей группе: `[^>"']`. Она может совпадать только с одним символом, что выглядит несколько неэффективно. Вы можете решить, что добавление квантификатора `+` после символьного класса позволило бы повысить эффективность, так как это обеспечило бы совпадение этой альтернативы сразу с несколькими символами на каждое повторение группы – в позициях внутри испытуемой строки, где обнаружено совпадение, – и вы были бы правы. Возможность совпадения более чем с одним символом позволяет механизму регулярных выражений пропустить множество ненужных шагов в случае успешного сопоставления.

Однако такое изменение может привести к неочевидным отрицательным последствиям. Если механизм регулярных выражений обнаружит открывающий символ `<`, для которого отсутствует парный закрывающий символ `>`, обеспечивающий успех сопоставления, это вызовет бесконтрольный рост количества возвратов из-за огромного количества спосо-

бов комбинирования внутренних квантификаторов с внешним (следующим за несохраняющей группой) при сопоставлении текста, следующего за <. Механизм регулярных выражений должен будет опробовать все возможные перестановки, прежде чем прекратить поиск. Берегитесь!

Еще хуже. Чтобы продемонстрировать еще более экстремальный вариант использования вложенных квантификаторов, приводящий к бесконтрольному увеличению числа возвратов, применим регулярное выражение /(A+A+)+B/ к строке, содержащей только символы A. Данное конкретное выражение эффективнее было бы записать как /AA+B/, однако ради обсуждения представим, что два символа A представляют различные шаблоны, которые могут совпадать с одинаковыми фрагментами испытуемой строки.

Если применить это регулярное выражение к строке из 10 символов A ("AAAAAAAAAA"), механизм регулярных выражений тут же обнаружит совпадение первого шаблона A+ со всеми 10 символами. После этого он вернется на один символ назад, что обеспечит совпадение второго шаблона A+ с последним символом. Затем будет выполнена попытка повторить сопоставление группы, но так как в строке не осталось символов A, а требование наличия хотя бы одного совпадения, предъявляемое квантификатором + группы, уже удовлетворено, механизм регулярных выражений попытается отыскать символ B. Поиск не увенчается успехом, но это не станет причиной отказаться от продолжения поиска, потому что существует еще множество неопробованных вариантов. Что если первому шаблону A+ соответствуют восемь символов, а второму – два? Или первому шаблону A+ соответствуют три символа, второму – два, а группа должна повторяться два раза? А что если в первом повторении группы первому шаблону A+ соответствуют два символа, второму – три, а во втором повторении первому соответствует один символ и второму – четыре? Несмотря на то что и мне, и вам совершенно очевидно, что ни одна из комбинаций не найдет отсутствующий символ B, механизм регулярных выражений педантично проверит все эти бессмысленные комбинации. Сложность этого регулярного выражения в самом тяжелом случае возрастает до O(2n), или до двух в *n*-й степени, где *n* – длина строки. Для случая со строкой из 10 символов A механизму регулярных выражений потребуется выполнить 1024 возврата, а для строки из 20 символов A это число уже превысит миллион. Строки из 35 символов A будет достаточно, чтобы «подвесить» Chrome, IE, Firefox и Opera как минимум на 10 минут (если не дольше), пока они обработают более чем 34 миллиарда возвратов, необходимых, чтобы проверить все возможные комбинации. Исключение составляют последние версии Safari, в которых механизм регулярных выражений способен определить зацикливание и прервать сопоставление (в Safari ограничивается количество допустимых возвратов, при превышении которого попытки сопоставления прерываются).

Ключом к предотвращению подобных проблем является исключение возможности совпадения разных частей регулярного выражения с оди-

наковыми фрагментами испытуемой строки. В данном регулярном выражении эту проблему легко исправить, переписав его в виде /AA+B/, но в сложных регулярных выражениях это может оказаться не так просто. Иногда, как последнее средство, можно использовать имитацию атомарной группировки, хотя в некоторых случаях можно использовать другие решения, которые позволят сохранить регулярные выражения более простыми для понимания. Применение имитации атомарной группировки к данному выражению, после чего оно приобретает вид /((?=(A+A+))\2)+B/, полностью устраняет проблему с возвратами.

Примечание к измерению производительности

Поскольку производительность регулярного выражения может в значительной степени зависеть от текста, к которому оно применяется, нет достаточно однозначного способа сравнить производительность разных регулярных выражений. Для повышения точности сравнения регулярные выражения следует тестировать на строках различной длины, соответствующих, не соответствующих и почти соответствующих выражениям.

Это одна из причин столь длинного описания проблемы возвратов в данной главе. Без твердого понимания механизма возвратов невозможно идентифицировать проблемы, связанные с возвратами. Чтобы как можно раньше выявить проблему бесконтрольного увеличения числа возвратов, всегда следует тестировать регулярные выражения на длинных строках, содержащих частичные совпадения. Подумайте, какие строки будут частично совпадать с вашими регулярными выражениями, и обязательно включите их в свои тесты.

Дополнительные пути повышения производительности регулярных выражений

Ниже перечислены различные дополнительные способы повышения производительности регулярных выражений. Некоторые из них уже упоминались при обсуждении механизма возвратов.

Отсутствие совпадения должно обнаруживаться как можно быстрее

Низкая скорость работы регулярных выражений обычно связана с поздним обнаружением отсутствия совпадения. Если регулярное выражение обнаруживает в длинной строке множество неполных совпадений, оно будет много раз терпеть неудачу, прежде чем найдет полное соответствие. Переделка регулярного выражения так, чтобы совпадение обнаруживалось быстро, а его отсутствие – медленно (например, за счет увеличения количества возвратов для опробования всех возможных комбинаций), обычно является проигрышной стратегией.

Начинайте регулярные выражения с простых, обязательных комбинаций

В идеале первой в регулярном выражении должна быть комбинация, которая быстро проверяется и исключает как можно больше очевидно

несоответствующих вариантов. На эту роль отлично подходят якорные метасимволы (^ или $), конкретные символы (например, x или \u263A), символьные классы (например, [a-z] или такие метасимволы, как \d) и границы слов (\b). Если возможно, следует избегать начинать регулярные выражения с группировок или необязательных комбинаций, а также вариантов выбора, таких как /one|two/, поскольку это вынудит механизм регулярных выражений опробовать множество начальных комбинаций. Броузер Firefox весьма чувствителен к использованию любых квантификаторов в начальных комбинациях и лучше оптимизирует такие выражения, как \s\s*, чем \s+ или \s{1,}. Другие броузеры в основном способны оптимизировать и те, и другие варианты.

Квантифицированные шаблоны и следующие за ними комбинации должны быть взаимоисключающими

Если совпадения с символами и смежными с ними шаблонами или подвыражениями могут перекрываться, это увеличивает количество вариантов разбиения текста между ними. Чтобы избежать этого, необходимо делать шаблоны максимально узкоспециализированными. Не используйте шаблон ".*?" (опирающийся на возвраты), когда в действительности подразумевается "[^"\r\n]*".

Количество и богатство операторов выбора должно быть минимальным

Наличие альтернатив, перечисленных через оператор выбора |, может потребовать проверить все имеющиеся альтернативы для каждой позиции в строке. Часто операторы выбора можно заменить символьными классами и необязательными компонентами или перенести конструкцию выбора ближе к концу регулярного выражения (позволив ему терпеть неудачу прежде, чем будет достигнута конструкция выбора). Примеры реализации этого приема демонстрируются в следующей таблице.

Вместо конструкции	Использовать
cat|bat	[cb]at
red|read	rea?d
red|raw	r(?:ed|aw)
(.|\r|\n)	[\s\S]

Символьные классы, совпадающие с любым символом (такие как [\s\S], [\d \D], [\w\W] или [\0-\uFFFF]) фактически эквивалентны шаблону (?:.|\r|\n| \u2028|\u2029). Они включают четыре символа, которые не совпадают с «точкой» (возврат каретки, перевод строки, разделитель строк и разделитель абзацев).

Символьные классы действуют быстрее оператора выбора, потому что основаны на битовых векторах (или других быстрых реализациях), а не на механизме возвратов. В случаях когда без оператора выбора не обойтись, первой следует указывать наиболее вероятную альтернативу, если это не влияет на логику работы регулярного выражения. Альтернативы опробуются в направлении слева направо, поэтому наиболее вероятные альтернативы будут при сопоставлении просматриваться в первую очередь.

Имейте в виду, что Chrome и Firefox автоматически выполняют некоторые из этих оптимизаций, и поэтому приемы ручной настройки выбора в них дают меньший эффект.

Используйте несохраняющую группировку

При использовании сохраняющей группировки расходуются дополнительные время и память на сохранение и поддержание обратных ссылок. При отсутствии необходимости в обратных ссылках можно ликвидировать сопутствующие им накладные расходы, используя несохраняющую группировку – например, применяя конструкцию `(?:...)` вместо `(...)`. Некоторые предпочитают заключать свои регулярные выражения в сохраняющую группу, когда необходимо получить обратную ссылку на совпадение со всем регулярным выражением. Однако в этом нет необходимости, потому что полное совпадение можно получить, обратившись, например, к нулевому элементу массива, возвращаемого методом `regex.exec()`, или к метасимволу `$&` в замещающем тексте.

Замена сохраняющей группировки ее несохраняющим аналогом дает минимальный эффект в Firefox, но может дать существенный прирост производительности в других броузерах при работе с длинными строками.

Сохраняйте фрагменты текста для уменьшения объема последующей обработки

Как исключение из предыдущего правила, если после применения регулярного выражения потребуется сослаться на отдельные *части* совпадения, то эти части следует сохранять в обратных ссылках. Например, при обработке содержимого строк в кавычках, совпавших с регулярным выражением, следует использовать шаблон `/"([^"]*)"/` и обрабатывать содержимое соответствующей обратной ссылки вместо применения шаблона `/"[^"]*"/` и удаления кавычек из результата вручную. При использовании в цикле такой подход может сэкономить немало времени.

Выделяйте обязательные комбинации

Чтобы помочь механизму регулярных выражений оптимизировать поиск, ему следует помочь определить, какие комбинации являются обязательными. Когда комбинация используется внутри подвыражения или в операторе выбора, механизму регулярных выражений

сложнее определить, являются ли они обязательными, и некоторые механизмы даже не будут пытаться делать какие-либо предположения. Например, выражение `/^(ab|cd)/` выделяет якорный метасимвол привязки к началу строки. Встретив его, IE и Chrome не будут пытаться отыскать совпадение не с начала строки, выполняя поиск практически мгновенно независимо от длины строки. Однако для эквивалентного регулярного выражения `/(^ab|^cd)/`, в котором якорный метасимвол ^ никак не выделяется, IE не будет применять никаких оптимизаций, что может привести к бессмысленному поиску совпадений в каждой позиции в строке.

Используйте наиболее подходящие квантификаторы

Как описывалось выше, в разделе «Повторения и возвраты», максимальные и минимальные квантификаторы выполняют поиск поразному, даже когда они соответствуют одной и той же строке. Использование квантификатора наиболее подходящего типа (исходя из предполагаемого количества возвратов) в случаях, где они дают одинаково корректные результаты, может существенно повысить производительность, особенно на длинных строках.

Минимальные квантификаторы действуют особенно медленно в Opera 9.x и в более ранних версиях, однако этот недостаток был устранен в Opera 10.

Повторно используйте регулярные выражения, присваивая их переменным

Присваивая регулярные выражения переменным, можно избежать лишних накладных расходов на их повторную компиляцию. Некоторые перегибают палку и используют схемы кэширования регулярных выражений, позволяющие избежать повторной компиляции любого регулярного выражения с определенной комбинацией флагов. Применение таких решений не дает особенно большого выигрыша, потому что компиляция регулярных выражений выполняется быстро, а их использование обычно влечет увеличение накладных расходов, от которых они призваны избавлять. Вполне достаточно будет избегать повторной компиляции регулярных выражений в циклах. Иными словами, не делайте, как показано ниже:

```
while (/regex1/.test(str1)) {
    /regex2/.exec(str2);
    ...
}
```

Используйте такой поход:

```
var regex1 = /regex1/,
    regex2 = /regex2/;
while (regex1.test(str1)) {
    regex2.exec(str2);
    ...
}
```

Разбивайте сложные регулярные выражения на более простые составляющие

Не нужно стараться реализовать все необходимое в одном регулярном выражении. Сложные задачи, требующие применения условной логики, решаются обычно проще и эффективнее при разбиении их на два или более регулярных выражения, когда каждое последующее регулярное выражение выполняет поиск по результатам, полученным от предыдущего. Чудовищные регулярные выражения, реализующие все необходимое в одном шаблоне, сложнее в сопровождении и вероятнее порождают проблему, связанную с увеличением числа возвратов.

Когда не следует использовать регулярные выражения

При осторожном обращении регулярные выражения работают очень быстро. Однако они весьма неэффективны при поиске строковых литералов. Это особенно верно для ситуаций, когда заранее известно, какую часть строки требуется проверить. Например, если необходимо проверить, оканчивается ли строка точкой с запятой, можно было бы воспользоваться таким регулярным выражением:

```
endsWithSemicolon = /;$/.test(str);
```

Возможно, кто-то удивится, узнав, что ни один из основных современных броузеров не способен определить, что совпадение возможно только в конце строки. Они будут последовательно просматривать всю строку. Каждый раз, встречая точку с запятой, механизм регулярных выражений будет переходить к следующему метасимволу ($), проверяя совпадение с концом строки. Поиск будет продолжаться до тех пор, пока не будет просмотрена вся строка. Чем длиннее строка (и чем больше в ней точек с запятой), тем дольше будет выполняться поиск.

В данном случае оптимальнее будет пропустить все промежуточные шаги, выполняемые механизмом регулярных выражений, и просто проверить последний в строке:

```
endsWithSemicolon = str.charAt(str.length - 1) == ";";
```

Это решение работает лишь немного быстрее регулярного выражения, когда применяется к коротким строкам, но, что особенно важно, при его использовании длина строки не влияет на время, необходимое для проверки.

В этом примере был использован метод `charAt()`, возвращающий символ в определенной позиции. Строковые методы `slice()`, `substr()` и `substring()` прекрасно подойдут в ситуациях, когда потребуется извлечь и проверить последовательность из нескольких символов в определенной позиции. Кроме того, для определения позиций строк литералов или проверки их присутствия с успехом можно использовать методы `indexOf()` и `lastIndexOf()`. Все эти строковые методы работают быстро и помогут

избежать лишних накладных расходов при поиске строк литералов, при котором нет выигрыша от использования возможностей регулярных выражений.

Усечение строк

На практике часто приходится решать простую задачу удаления из строки начальных и конечных пробелов. Хотя стандарт ECMAScript 5 определяет встроенный строковый метод `trim()` (наличие которого следует проверять в будущих версиях броузеров), исторически язык JavaScript не включает его. В текущих версиях броузеров все еще необходимо определять собственный метод `trim()` или полагаться на библиотеку, включающую его.

Операция усечения строк редко является узким местом в смысле производительности, но ее можно рассматривать как показательный пример оптимизации регулярных выражений, потому что ее можно реализовать разными способами.

Усечение с применением регулярных выражений

Применение регулярных выражений позволяет создать весьма компактную реализацию метода `trim()`, что особенно важно для JavaScript-библиотек, размер которых имеет большое значение. Вероятно, самым лучшим и самым универсальным решением является использование двух операций подстановки: одной – для удаления начальных и другой – для удаления завершающих пробелов. Это обеспечит простоту реализации и высокую скорость, особенно при работе с длинными строками.

```
if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/, "");
    }
}

// проверка нового метода...
// в начале строки присутствуют
// символ табуляции (\t) и символ перевода строки (\n).
var str = " \t\n test string ".trim();
alert(str == "test string"); // выведет "true"
```

Блок `if`, окружающий реализацию, позволяет избежать переопределения уже имеющегося метода `trim()`, который является более оптимизированным и обычно выполняется намного быстрее, чем любая другая реализация на основе функций языка JavaScript. В последующих реализациях данного примера наличие этой условной инструкции предполагается, хотя она и не во всех случаях написана.

Замена регулярного выражения `/\s+$/` на `/\s\s*$/` повышает производительность в Firefox примерно на 35% (в зависимости от длины строки

и ее содержимого)[1]. Несмотря на функциональную идентичность этих двух регулярных выражений, Firefox обеспечивает дополнительную оптимизацию для регулярных выражений, начинающихся с шаблона без квантификатора. В других броузерах оптимизация не дает такой же значительной разницы или вообще не выполняется. Однако замена регулярного выражении, совпадающего с началом строки, на `/^\s\s*/` не дает измеримой разницы, потому что первый якорный метасимвол ^ предотвращает поиск соответствия в других позициях в строке (не позволяя заметить разницу в производительности даже при выполнении более тысячи попыток поиска в длинной строке).

Ниже приводится еще несколько реализаций метода `trim()` на основе регулярных выражений, являющихся наиболее часто используемыми вариантами, которые можно встретить. В табл. 5.2 в конце этого раздела приводятся значения производительности всех описываемых здесь реализаций метода `trim()` в разных броузерах. В действительности существует намного больше вариантов регулярных выражений для усечения строк, чем перечислено здесь, но при работе с длинными строками они неизмеримо медленнее (или как минимум их производительность сильно отличается в разных броузерах), чем две простые операции подстановки строк.

```
// trim 2
String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, "");
}
```

Это, пожалуй, наиболее типичное решение. Оно объединяет два простых регулярных выражения посредством оператора выбора и использует флаг `/g` (global – глобальный), обеспечивающий замену не только первого, но и всех остальных совпадений (если строка содержит пробельные символы и в начале, и в конце, регулярное выражение будет обнаруживать два совпадения). Это не самая худшая версия, но при работе с длинными строками она оказывается медленнее, чем версия, использующая две простых операции подстановки, потому что для каждого символа строки ей приходится проверять две альтернативы.

```
// trim 3
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*?)\s*$/, "$1");
}
```

Это регулярное выражение соответствует всей строке и сохраняет последовательность от первого до последнего непробельного символа (если имеются) в обратной ссылке. Усеченная версия строки получается замещением исходной строки содержимым обратной ссылки.

Этот подход концептуально прост, но минимальный квантификатор внутри сохраняющей группировки вынуждает механизм регулярных

[1] Проверено в Firefox версий 2, 3 и 3.5.

выражений выполнить массу дополнительной работы (то есть возвратов), что делает эту версию медленной при применении к длинным строкам. После входа в сохраняющую группу минимальный квантификатор `*?` требует, чтобы количество совпадений с символьным классом `[\s\S]` было минимальным. Поэтому после сопоставления с очередным символом механизм регулярных выражений останавливается, чтобы сопоставить остаток строки с шаблоном `\s*$`. Если последняя проверка терпит неудачу из-за наличия непробельных символов далее текущей позиции в строке, выполняется переход к следующему символу, производится обновление содержимого обратной ссылки и затем снова производится попытка сопоставить последний шаблон с остатком строки.

Минимальные квантификаторы особенно медленно работают в Opera 9.x и в более ранних версиях. Усечение длинных строк с помощью этого метода в Opera 9.64 выполняется примерно от 10 до 100 раз медленнее, чем в других основных броузерах. Этот давний недостаток был исправлен в Opera 10, благодаря чему этот метод стал показывать производительность, сопоставимую с другими броузерами.

```
// trim 4
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*\S)?\s*$/, "$1");
}
```

Эта версия напоминает предыдущую, но в ней для повышения производительности вместо минимального используется максимальный квантификатор. Чтобы гарантировать совпадение сохраняющей группы до последнего непробельного символа, потребовалось добавить в нее завершающий метасимвол `\S`. Однако из-за необходимости обеспечить совпадение регулярного выражения со строками, состоящими только из пробельных символов, вся сохраняющая группа была сделана необязательной с помощью квантификатора `?`.

Здесь максимальный квантификатор в подвыражении `[\s\S]*` повторяет шаблон, совпадающий с любым символом, пока не будет достигнут конец строки. Затем механизм регулярных выражений начинает возвращать по одному символу, пока продолжается совпадение со следующим шаблоном `\S` или пока он не вернется к первому символу совпадения с группой (после которого он пропускает группу).

В ситуациях, когда завершающих пробельных символов в строке меньше, чем любых других, эта версия оказывается быстрее предыдущей, использующей минимальный квантификатор. В действительности, она настолько быстрая, что в IE, Safari, Chrome и Opera 10 по скорости превосходит даже версию на основе двух операций подстановки. Это обусловлено тем, что некоторые броузеры предусматривают специальную оптимизацию для символьных классов с максимальным квантификатором, совпадающих с любым символом. Механизм регулярных выражений переходит сразу к концу строки, не проверяя промежуточные символы (хотя по-прежнему сохраняет позиции для возвратов), и затем

выполняет возвраты обычным образом. К сожалению, эта версия работает существенно медленнее в Firefox и Opera 9, поэтому использование версии с двумя подстановками выглядит предпочтительнее с точки зрения поддержки различных броузеров.

```
// trim 5
String.prototype.trim = function() {
    return this.replace(/^\s*(\S*(\s+\S+)*)\s*$/, "$1");
}
```

Это достаточно распространенное решение, но оно является самым медленным из демонстрируемых здесь во всех броузерах, поэтому нет особых причин использовать его на практике. Эта версия напоминает две предыдущие в том, что используемое в ней регулярное выражение совпадает со всей строкой и заменяет ее частью, которую требуется получить, но так как при каждом повторении внутренняя группа совпадает только с одним словом, механизм регулярных выражений вынужден выполнить массу мелких шагов. Потери производительности этого регулярного выражения на коротких строках практически незаметны, но на длинных строках, содержащих большое количество слов, эти потери могут оказаться весьма существенными.

Преобразование внутренней сохраняющей в несохраняющую, то есть замена подвыражения (\s+\S+) на (?:\s+\S+), позволяет повысить производительность примерно на 20–45% в Opera, IE и Chrome и незначительно в Safari и Firefox. Однако несохраняющая группировка не спасает эту реализацию. Обратите внимание, что внешняя группировка не может быть преобразована в несохраняющую, потому что она создает ссылку на строку замены.

Усечение без применения регулярных выражений

Несмотря на высокую скорость работы регулярных выражений, есть смысл взглянуть на скорость усечения строк без их использования. Ниже приводится одна из возможных реализаций:

```
// trim 6
String.prototype.trim = function() {
    var start = 0,
        end = this.length - 1,
        ws = " \n\r\t\f\x0b\xa0\u1680\u180e\u2000\u2001\u2002\u2003
\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u202f
\u205f\u3000\ufeff";

    while (ws.indexOf(this.charAt(start)) > -1) {
        start++;
    }
    while (end > start && ws.indexOf(this.charAt(end)) > -1) {
        end--;
    }
    return this.slice(start, end + 1);
}
```

Переменная `ws` в этом примере содержит все пробельные символы, определяемые стандартом ECMAScript 5. Из соображений эффективности копирование каких-либо частей строки не выполняется, пока не будут известны начальная и конечная позиции.

Оказывается, что эта реализация выигрывает в скорости у регулярных выражений, когда на концах строки имеется лишь небольшое количество пробельных символов. Регулярные выражения отлично справляются с удалением пробельных символов в начале строки, но теряют в скорости при удалении пробелов в конце длинных строк. Как отмечалось выше в разделе «Когда не следует использовать регулярные выражения», механизм регулярных выражений не может перейти сразу в конец строки, не проверив промежуточные символы. Однако именно это делает данная реализация, второй цикл в которой начинает движение с конца строки, пока не обнаружит непробельный символ.

Скорость работы данной версии не зависит от длины строки, но она имеет свое слабое место: большое количество начальных и завершающих пробельных символов. Это объясняется необходимостью выполнять циклы по символам и производить проверку на принадлежность к множеству пробельных символов, которая по своей эффективности не может соперничать с оптимизированными процедурами поиска механизмов регулярных выражений.

Смешанное решение

Последнее решение в этом разделе объединяет эффективность регулярных выражений для удаления начальных пробельных символов с производительным способом удаления завершающих пробельных символов без использования регулярных выражений.

```
// trim 7
String.prototype.trim = function() {
    var str = this.replace(/^\s+/, ""),
        end = str.length - 1,
        ws = /\s/;

    while (ws.test(str.charAt(end))) {
        end--;
    }
    return str.slice(0, end + 1);
}
```

Этот смешанный метод остается потрясающе быстрым при удалении небольшого количества начальных пробельных символов и устраняет риск потери производительности при обработке строк с большим количеством начальных пробельных символов и состоящих исключительно из них (хотя слабым местом остается обработка строк с большим количеством завершающих пробельных символов). Обратите внимание, что для проверки совпадения завершающих символов с пробельными в этом

решении используется цикл с регулярным выражением. Использование регулярного выражения несет дополнительные накладные расходы, тем не менее это решение позволило отказаться от длинного списка пробельных символов в пользу краткости программного кода и совместимости с разными броузерами.

Все описанные здесь методы `trim()` имеют одну общую особенность. Производительность методов, основанных на регулярных выражениях, больше зависит от общей длины строки, чем от количества удаляемых символов. Производительность методов, не использующих регулярные выражения и при удалении завершающих пробелов выполняющих цикл начиная с конца строки, не зависит от общей длины строки, но находится в прямой зависимости от количества удаляемых пробельных символов. Простота метода с двумя операциями подстановки на основе регулярных выражений и приличная скорость работы в разных броузерах со строками с различным содержимым и различной длины делают его наиболее привлекательным универсальным решением. Смешанное решение показывает исключительно высокую скорость обработки длинных строк, за которую пришлось заплатить небольшим увеличением объема программного кода; в некоторых броузерах этот метод страдает потерей производительности при усечении большого количества завершающих пробельных символов. Полные сведения о производительности разных методов в разных броузерах приводятся в табл. 5.2.

Таблица 5.2. Производительность различных реализаций метода trim() в разных броузерах

Броузер	Время (мс)[a]						
	Trim 1[b]	Trim 2	Trim 3	Trim 4	Trim 5[c]	Trim 6	Trim 7
IE7	80/80	315/312	547/539	36/42	218/224	14/1015	18/409
IE8	70/70	252/256	512/425	26/30	216/222	4/334	12/205
Firefox 3	136/147	164/174	650/600	1098/1525	1416/1488	21/151	20/144
Firefox 3.5	130/147	157/172	500/510	1004/1437	1344/1394	21/332	18/50
Firefox 3.2.3	253/253	424/425	351/359	27/29	541/554	2/140	5/80
Safari 4	37/37	33/31	69/68	32/33	510/514	<0,5/29	4/18
Opera 9.64	494/517	731/748	9066/9601	901/955	1953/2016	<0,5/210	20/241
Opera 10	75/75	94/100	360/370	46/46	514/514	2/186	12/198
Chrome 2	78/78	66/68	100/101	59/59	140/142	1/37	24/55

[a] Время выполнения 100 операций усечения длинных строк (40 Кбайт); в первом испытании строки содержали по 10 пробельных символов с обоих концов, во втором – по 1000.

[b] Тестирование выполнялось без оптимизации `/\s\s*$/`.

[c] Тестирование выполнялось без замены сохраняющей группировки на несохраняющую.

В заключение

Использование неэффективных строковых операций и невнимательность при создании регулярных выражений могут стать основными причинами потери производительности, однако советы, данные в этой главе, помогут вам избежать типичных ловушек.

- Операция слияния элементов массива является единственной, обеспечивающей приемлемую производительность в IE7 при конкатенации большого количества или длинных строк.
- В броузерах, отличных от IE7 и более ранних версий, операция слияния элементов массива является одним из самых медленных способов конкатенации строк. Вместо нее лучше использовать простые операторы + и += и стараться избегать создания промежуточных строк.
- Возвраты являются неотъемлемой составляющей процесса сопоставления с регулярными выражениями и одновременно потенциальным источником потерь эффективности.
- Бесконтрольное увеличение количества возвратов в регулярных выражениях, которые обычно работают очень быстро, может приводить к существенному их замедлению или даже к отказам при применении к строкам, имеющим частичные совпадения. В число приемов, позволяющих избежать этой проблемы, входят: обеспечение взаимоисключаемости смежных подвыражений, отказ от использования вложенных квантификаторов, обеспечивающих возможность совпадения с одной и той же частью строки множеством способов, и устранение ненужных возвратов, используя атомарную природу опережающих проверок.
- Существует множество дополнительных приемов повышения эффективности, помогающих регулярным выражениям быстрее находить соответствия и тратить меньше времени на оценку позиций, в которых совпадение отсутствует (см. раздел «Дополнительные пути повышения производительности регулярных выражений»).
- Регулярные выражения не всегда являются лучшим инструментом решения задач, особенно когда требуется отыскать строковый литерал.
- Существует большое количество разных способов усечения строки. Однако наиболее оптимальное соотношение краткости и эффективности в разных броузерах при работе со строками разной длины и с разным содержимым обеспечивает прием на основе двух простых регулярных выражений (одно – для удаления начальных, а другое – для удаления завершающих пробельных символов). Отличной альтернативой, менее зависящей от общей длины строк, является цикл поиска первого непробельного символа, начиная с конца строки, или объединение его с регулярными выражениями.

6

Отзывчивые интерфейсы

Нет ничего более обескураживающего, чем отсутствие реакции на щелчок мышью на каком-либо элементе веб-страницы. Эта проблема порождена самой сущностью транзакционных веб-приложений; это она привела к появлению предупреждения «не щелкайте дважды», почти всегда сопровождающего кнопки отправки в большинстве форм. Пользователи испытывают естественное стремление повторить действие, которое не привело к видимым изменениям, и поэтому обеспечение отзывчивости веб-приложений является важной задачей, связанной с производительностью.

В главе 1 была представлена концепция потока выполнения, обслуживающего пользовательский интерфейс броузера. Коротко напомню, что в большинстве броузеров обновление пользовательского интерфейса и выполнение программного кода JavaScript-сценариев производится в рамках единственного процесса. В каждый конкретный момент времени может выполняться только одна из этих операций, то есть пока выполняется программный код на JavaScript, пользовательский интерфейс не может откликаться на действия пользователя, и наоборот. Фактически на время выполнения программного кода сценария пользовательский интерфейс оказывается «заблокированным», поэтому для формирования субъективного восприятия производительности веб-приложения большое значение имеет, сколько времени займет выполнение вашего программного кода.

Поток выполнения пользовательского интерфейса броузера

Процесс, одновременно используемый для выполнения программного кода на языке JavaScript и для обслуживания пользовательского интер-

фейса, часто называют потоком выполнения пользовательского интерфейса (или главным потоком выполнения) броузера (хотя термин «поток выполнения» может по-разному трактоваться в разных броузерах). Поток пользовательского интерфейса обслуживает простую очередь, где хранятся задания, пока процесс занят решением других задач. Как только процесс освобождается, он извлекает из очереди следующее задание и выполняет его. Заданиями могут быть и выполнение некоторого программного кода на JavaScript, и обновление пользовательского интерфейса, в том числе перерисовывание и перекомпоновка (обсуждаются в главе 3). Важно отметить, что каждое отдельное действие пользователя может приводить к добавлению в очередь одно или несколько заданий.

Рассмотрим простой интерфейс, где щелчок на кнопке приводит к отображению сообщения:

```
<html>
<head>
    <title>Browser UI Thread Example</title>
</head>
<body>
    <button onclick="handleClick()">Click Me</button>
    <script type="text/javascript">

        function handleClick(){
            var div = document.createElement("div");
            div.innerHTML = "Clicked!";
            document.body.appendChild(div);
        }

    </script>
</body>
</html>
```

Щелчок на кнопке в этом примере вынудит главный поток выполнения создать два задания и добавить их в очередь. Первое задание – обновить изображение кнопки, придав ей вид нажатой кнопки, и второе – выполнить программный код метода `handleClick()` так, чтобы был выполнен только этот метод, а также любые другие методы, вызванные им. Допустим, что главный поток выполнения не занят другими делами. Он извлечет из очереди первое задание и обновит изображение кнопки. Затем извлечет второе задание и выполнит программный код на языке JavaScript. В процессе выполнения метода `handleClick()` будет создан новый элемент `<div>` и добавлен в конец элемента `<body>`, что фактически приведет к еще одному изменению пользовательского интерфейса. То есть в ходе выполнения программного кода на JavaScript в очередь будет добавлено новое задание обновления пользовательского интерфейса, которое будет выполнено сразу по завершении работы программного кода на языке JavaScript, как показано на рис. 6.1.

Когда главный поток выполнения выполнит все задания, процесс перейдет в холостой режим, ожидая появления новых заданий в очереди.

Холостой режим является идеальным состоянием, потому что в этом случае любые действия пользователя приводят к немедленному обновлению пользовательского интерфейса. Если пользователь попытается взаимодействовать со страницей во время выполнения какого-либо задания, пользовательский интерфейс не только не обновится немедленно, но, возможно, даже не будет создано и поставлено в очередь новое задание обновления пользовательского интерфейса. На практике большинство броузеров прекращают добавление заданий в очередь, пока выполняется программный код на JavaScript, поэтому обработка заданий, связанных с выполнением программного кода, должна завершаться как можно быстрее, чтобы не вызывать негативные впечатления у пользователя.

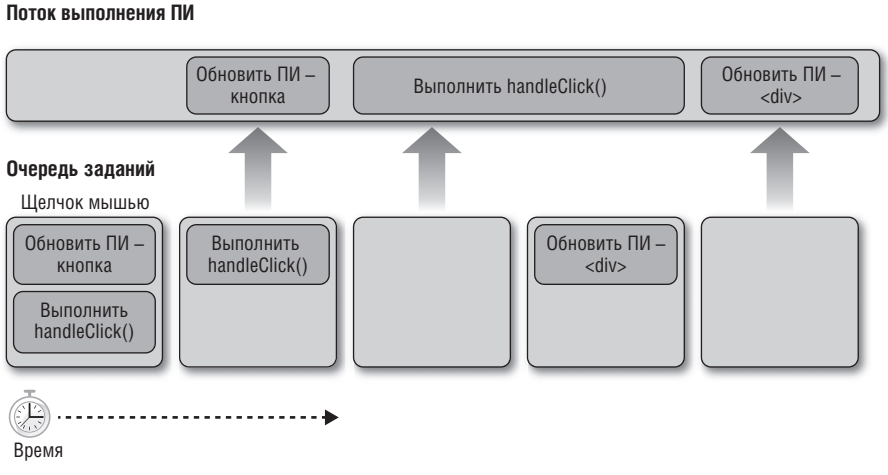


Рис. 6.1. *Добавление заданий главного потока выполнения по мере взаимодействия пользователя со страницей*

Ограничения броузеров

Броузеры устанавливают ограничение на время, которое отводится на выполнение JavaScript-сценария. Это ограничение объясняется необходимостью предотвратить блокирование броузера или компьютера злонамеренным программным кодом, выполняющим бесконечные интенсивные вычисления. Всего существует два таких ограничения: *ограничение размера стека вызовов* (обсуждается в главе 4) и *ограничение времени выполнения сценариев*. Ограничение времени выполнения сценария иногда называют таймером продолжительности выполнения сценариев или таймером защиты от неуправляемых сценариев, и основная его идея основана на том, что броузер следит за продолжительностью работы сценария и прерывает его выполнение, как только будет превышен установленный предел. По достижении предельного значения на экране появляется диалог, показанный на рис. 6.2.

Рис. 6.2. *Предупреждающий диалог, который выводится броузером Internet Explorer после выполнения более 5 миллионов инструкций*

Существует два способа ограничения продолжительности выполнения сценариев. Первый из них заключается в том, чтобы следить за количеством инструкций, выполненных с момента запуска программного кода. При таком подходе максимальная продолжительность работы сценариев на различных компьютерах может отличаться в зависимости от объема доступной памяти и быстродействия процессора, определяющих скорость выполнения одной инструкции. Второй подход заключается в ограничении времени выполнения сценария. Количество инструкций, которые могут быть выполнены в отведенный интервал времени, также может отличаться на разных компьютерах в зависимости от их быстродействия, но в любом случае работа сценария будет прервана по истечении установленного времени. Естественно, все броузеры используют несколько отличающиеся подходы к измерению продолжительности выполнения сценариев:

- Internet Explorer начиная с версии 4 по умолчанию ограничивает продолжительность выполнения сценариев 5 миллионами инструкций; значение этого предела хранится в ключе реестра Windows *HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements*.
- Firefox по умолчанию ограничивает продолжительность выполнения 10 секундами; значение этого предела хранится в настройках броузера (доступных при вводе строки `about:config` в адресную строку) в параметре *dom.max_script_run_time*.
- Safari по умолчанию ограничивает продолжительность выполнения 5 секундами; это значение нельзя изменить, но имеется возможность отключить это ограничение, открыв меню Develop (Разработка) и выбрав пункт Disable Runaway JavaScript Timer (Отключить Runaway JavaScript Timer).
- В Chrome отсутствует отдельная настройка, ограничивающая продолжительность выполнения сценариев. Вместо этого броузер полагается на универсальную систему определения аварийных ситуаций.

- Opera не ограничивает продолжительность выполнения сценариев и будет продолжать выполнять программный код на JavaScript, пока он сам не завершится, тем не менее благодаря архитектурным решениям, реализованным в Opera, это не влияет на стабильность системы в процессе выполнения сценариев.

Когда продолжительность выполнения сценария превысит установленный предел, броузер выведет свое диалоговое окно независимо от наличия обработчиков ошибок в странице. Это переводит проблему в область проблем удобства использования, потому что большинство пользователей Интернета не являются техническими специалистами и потому не смогут верно оценить смысл сообщения об ошибке и правильно выбрать вариант ответа (остановить сценарий или позволить ему продолжить работу).

Если сценарий способен стать причиной вывода такого диалога в какомлибо из броузеров, это означает, что он тратит слишком много времени на решение задачи. Это также говорит о том, что в процессе выполнения сценария броузер слишком долго не сможет откликаться на действия пользователя. Разработчик не имеет возможности обработать ситуацию, вызывающую вывод диалога, – отсутствует возможность определить факт его вывода и, как следствие, отсутствует возможность исправить проблемы, которые могут возникнуть в этом случае. Очевидно, что самый лучший способ обойти ограничения времени выполнения сценария состоит в том, чтобы не приближаться к этим ограничениям.

Слишком долго – это сколько?

Тот факт, что броузер позволяет сценарию выполняться в течение определенного времени, вовсе не означает, что вы тоже должны позволять ему это. На самом деле, чтобы у пользователя создавалось благоприятное впечатление, время работы сценария должно быть намного меньше предела, устанавливаемого броузером. Брендан Эйч (Brendan Eich), создатель языка JavaScript, как-то сказал: «[сценарий], выполняющийся несколько секунд, наверняка делает что-то неправильно...»

Если несколько секунд – это слишком долго, то какая продолжительность выполнения JavaScript-сценария считается приемлемой? Как оказывается, даже одна секунда – это слишком долго. Длительность одной операции, выполняемой JavaScript-сценарием, не должна превышать 100 мс. Это число было получено в ходе исследований, проведенных Робертом Миллером (Robert Miller) в 1968 году.[1] Самое интересное, что Якоб Нильсен (Jakob Nielsen), известный специалист в области удобства и простоты использования, в своей книге «Usability Engineering»

[1] Miller, R. B., «Response time in man-computer conversational transactions», Proc. AFIPS Fall Joint Computer Conference, Vol. 33 (1968), 267–277. Доступна по адресу: *http://portal.acm.org/citation.cfm?id=1476589.1476628*.

(Morgan Kaufmann, 1994) отметил[1], что это число не изменилось со временем и было подтверждено исследованиями Xerox-PARC в 1991 году.[2]

Нильсен отмечает, что если интерфейс откликается на действия пользователя в течение 100 мс, у пользователя будет создаваться ощущение «непосредственного манипулирования объектами пользовательского интерфейса». Любой интервал времени, превышающий 100 мс, будет создавать ощущение оторванности от интерфейса. Поскольку обновление пользовательского интерфейса невозможно во время выполнения программного кода на JavaScript, у пользователя не будет создаваться ощущение участия в управлении интерфейсом, если продолжительность выполнения кода будет составлять более 100 мс.

Дополнительная сложность заключается в том, что во время выполнения JavaScript-сценариев некоторые броузеры даже не добавляют в очередь задания по обновлению пользовательского интерфейса. Например, если щелкнуть на кнопке в то время, когда выполняется некоторый программный код на JavaScript, броузер может не добавить в очередь задание отобразить кнопку в нажатом состоянии или выполнить JavaScript-обработчик щелчка на кнопке. В результате появится ощущение неотзывчивости, или «подвисания», пользовательского интерфейса.

Такое поведение наблюдается во всех броузерах. Во время выполнения сценария пользовательский интерфейс не обновляется в ответ на действия пользователя. Задания вызова JavaScript-обработчиков, порожденные в это время в результате действий пользователя, помещаются в очередь и затем выполняются по порядку, когда длительная операция будет завершена. При этом задания обновления пользовательского интерфейса в ответ на действия пользователя, произведенные в течение этого времени, просто пропускаются, потому что предпочтение отдается динамическим аспектам страницы. То есть если во время выполнения сценария произвести щелчок на кнопке, она никогда не будет отображена в нажатом состоянии, однако ее обработчик `onclick` будет выполнен.

Internet Explorer может пропускать задания выполнить JavaScript-код, порождаемые в ответ на действия пользователя, чтобы обеспечить выполнение только двух одинаковых действий, произведенных подряд. Например, если во время выполнения сценария выполнить четыре щелчка на кнопке, обработчик события `onclick` будет вызван только два раза.

Несмотря на то что в подобных случаях броузеры пытаются действовать логично, все эти особенности поведения разрушают положитель-

---

[1] Доступна по адресу: *www.useit.com/papers/responsetime.html*.

[2] Card, S. K., G.G. Robertson, and J.D. Mackinlay, «The information visualizer: An information workspace», Proc. ACM CHI'91 Conf. (New Orleans: 28 April–2 May), 181–188. Доступна по адресу: *http://portal.acm.org/citation.cfm?id=108874*.

ные впечатления пользователя. Поэтому лучше не допускать подобных ситуаций и ограничить время выполнения любой операции 100 мс. Измерения длительности операций следует производить в самом медленном броузере, который предстоит поддерживать (описание инструментов, позволяющих измерять производительность JavaScript, можно найти в главе 10).

Использование таймеров

Несмотря на все старания, иногда сложность операции не позволяет завершить ее выполнение в течение 100 мс. В таких случаях желательно организовать передачу управления главному потоку выполнения, чтобы обеспечить возможность обновления пользовательского интерфейса. То есть приостановить выполнение JavaScript-сценария и дать пользовательскому интерфейсу шанс обновиться, прежде чем продолжить работу сценария. В этом вам могут помочь JavaScript-таймеры.

Основы таймеров

В сценариях на языке JavaScript таймеры создаются вызовом функций `setTimeout()` или `setInterval()`, принимающих одинаковые аргументы: функцию, которую требуется вызвать, и время (в миллисекундах), которое должно пройти перед ее вызовом. Функция `setTimeout()` создает таймер, срабатывающий только один раз, а функция `setInterval()` создает таймер, срабатывающий неограниченное количество раз через равные интервалы времени.

Особенности взаимодействия таймеров с главным потоком выполнения делают их удобным инструментом деления продолжительных операций на короткие фрагменты. Вызов функции `setTimeout()` или `setInterval()` сообщает интерпретатору JavaScript, что он должен приостановить работу на указанное время и затем добавить в очередь задание выполнить JavaScript-код. Например:

```
function greeting(){
    alert("Hello world!");
}

setTimeout(greeting, 250);
```

Этот программный код добавит в очередь задание вызвать функцию `greeting()` через 250 мс. До этого момента будут выполняться все остальные задания обновления пользовательского интерфейса и выполнения JavaScript-кода. Имейте в виду, что второй аргумент определяет время, через которое указанное задание должно быть добавлено в очередь, и необязательно совпадает со временем, через которое это задание будет выполнено; это задание, как и любое другое, будет ожидать в очереди, пока не будут выполнены все другие задания, уже находящиеся в очереди. Рассмотрим следующий пример:

```
var button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 250);
};
```

Когда пользователь щелкнет на кнопке в этом примере, обработчик вызовет метод `oneMethod()` и установит таймер. Задание вызвать функцию, изменяющую цвет элемента `notice` и переданную таймеру, будет добавлено в очередь через 250 мс. Отсчет этих 250 мс начинается с момента вызова `setTimeout()`, а не с момента завершения внешней функции. То есть если функция `setTimeout()` была вызвана в момент времени n, задание вызвать функцию, переданную таймеру, будет добавлено в очередь в момент времени $n + 250$. Хронология этого процесса с момента щелчка на кнопке показана на рис. 6.3.

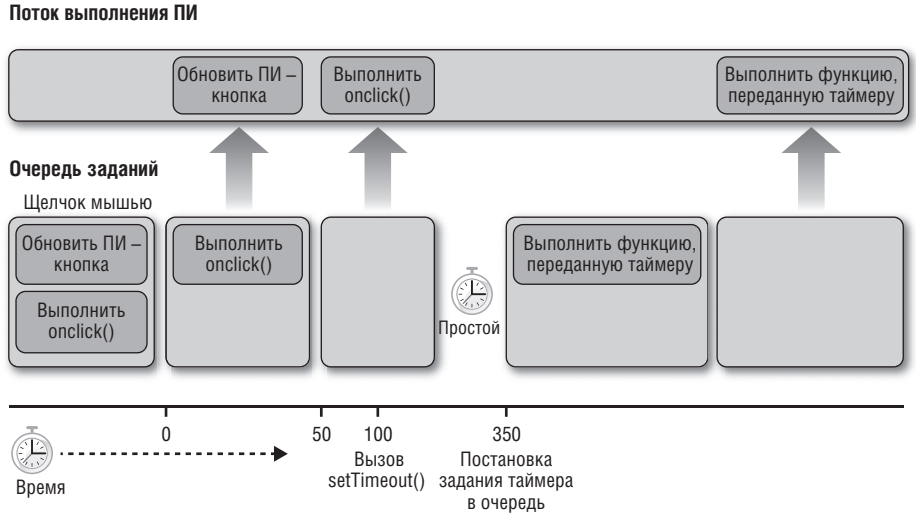


Рис. 6.3. *Второй аргумент функции setTimeout() определяет, когда следует добавить в очередь новое задание выполнить JavaScript-код*

Имейте в виду, что функция, переданная таймеру, не может быть вызвана до того, как завершит работу функция, в которой этот таймер был создан. Например, если в предыдущем примере уменьшить задержку таймера и после создания таймера добавить вызов другой функции, может сложиться ситуация, что таймер сработает раньше и добавит в очередь задание вызвать переданную ему функцию еще до того, как обработчик события `onclick` завершит работу:

```
var button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 50);

    anotherMethod();
};
```

Если метод `anotherMethod()` будет выполняться дольше 50 мс, задание вызвать функцию, переданную таймеру, будет добавлено в очередь еще до того, как обработчик `onclick` завершит работу. В результате функция будет вызвана практически сразу же после завершения обработчика `onclick`, без ощутимой задержки. Эта ситуация изображена на рис. 6.4.

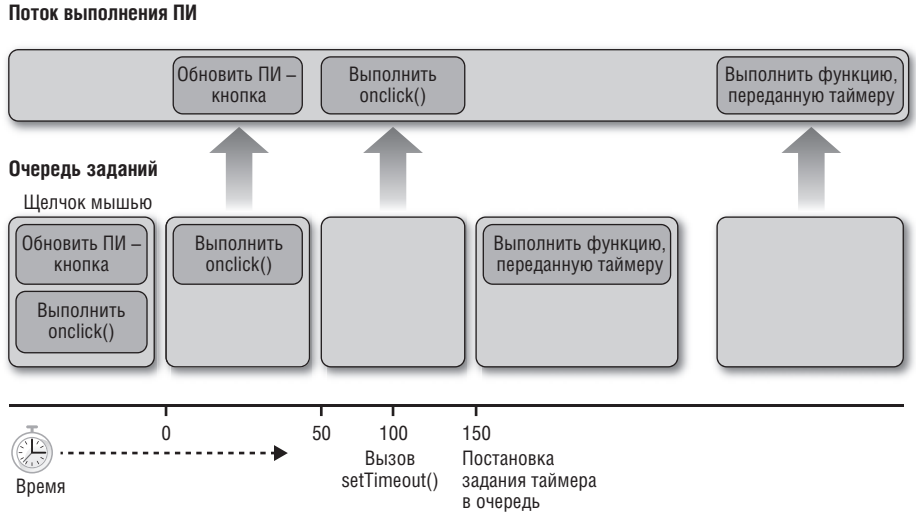


Рис. 6.4. *Задание, добавленное таймером, выполняется практически сразу, если функция, в которой была вызвана функция setTimeout(), выполняется дольше, чем интервал срабатывания таймера*

В любом случае создание таймера приводит к появлению паузы, в течение которой главный поток выполнения переключается с одного задания на другое. Как следствие сбрасываются все счетчики, используемые для слежения за ограничениями, включая таймер продолжительности выполнения сценария. Кроме того, внутри таймера вершина стека вызовов возвращается в исходное положение. Все это делает таймеры идеальным инструментом для работы со сценариями, выполняющимися продолжительное время, в любых броузерах.

Функция `setInterval()` практически идентична функции `setTimeout()`, за исключением того, что она организует многократное добавление в очередь задания выполнить JavaScript-код. Главное отличие состоит в том, что она не добавит новое задание в очередь, если задание, созданное тем же вызовом `setInterval()`, уже присутствует в очереди.

Точность таймера

Таймеры в JavaScript не отличаются высокой точностью и могут ошибаться на несколько миллисекунд в ту или иную сторону. Если при создании таймера указан интервал 250 мс, это еще не означает, что задание будет добавлено в очередь ровно через 250 мс после вызова `setTimeout()`. Все броузеры стараются обеспечить максимально высокую точность, но ошибки на несколько миллисекунд в ту или иную сторону нередки. Поэтому таймеры не считаются надежным средством измерения интервалов времени.

Разрешающая способность таймеров в Windows составляет 15 мс, поэтому задержка 15 мс, указанная при создании таймера, будет интерпретироваться как 0 или 15 мс в зависимости от момента последнего обновления системного времени. Использование задержек менее 15 мс в Internet Explorer может привести к блокировке броузера, поэтому минимальная задержка, которую рекомендуется устанавливать в нем, составляет 25 мс (которая будет интерпретироваться как 15 или 30 мс), чтобы гарантировать минимальную задержку 15 мс.

Такое ограничение минимальной величины задержки также позволит избежать проблем, связанных с разрешающей способностью таймеров в других броузерах и в других системах. Большинство броузеров показывает снижение точности срабатывания таймеров при использовании задержек менее 10 мс.

Обработка массивов с помощью таймеров

Одной из основных причин, приводящих к длительной работе сценариев, являются циклы, выполняющие слишком большой объем работы. Если даже после применения приемов оптимизации циклов, описанных в главе 4, не удается в достаточной мере уменьшить время выполнения, то следующим шагом является привлечение таймеров. Суть этого подхода состоит в том, чтобы разбить всю работу, выполняемую в цикле, на несколько частей. Типичные циклы реализуются по простому шаблону, как показано ниже:

```
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}
```

Циклы с такой структурой могут выполняться очень долго из-за высокой сложности обработки данных в `process()`, размера массива `items` или

обоих факторов сразу. В моей книге «Professional JavaScript for Web Developers, Second Edition» (Wrox, 2009), я предлагаю ответить на два вопроса, чтобы определить, возможно ли выполнять цикл асинхронно с помощью таймеров:

- Должна ли обработка данных выполняться синхронно?
- Должны ли данные обрабатываться последовательно?

Если на оба вопроса получен ответ «нет», исследуемый цикл является отличным кандидатом на применение таймеров с целью разделить работу на фрагменты. Простейший шаблон реализации асинхронного цикла имеет следующий вид:

```
var todo = items.concat(); // создать копию оригинала
setTimeout(function(){

    // извлечь очередной элемент массива и обработать его
    process(todo.shift());

    // если еще остались элементы для обработки, создать другой таймер
    if(todo.length > 0){
        setTimeout(arguments.callee, 25);
    } else {
        callback(items);
    }

}, 25);
```

Основная идея этого шаблона состоит в том, чтобы создать копию оригинального массива и использовать ее как очередь обрабатываемых элементов. Первый вызов `setTimeout()` создаст таймер, обрабатывающий первый элемент массива. Вызов `todo.shift()` вернет первый элемент и одновременно удалит его из массива. Затем это значение будет передано функции `process()`. После обработки элемента проверяется наличие других элементов. Если в массиве `todo` еще остались элементы, создается другой таймер. Поскольку следующий таймер должен вызвать ту же функцию, что и предыдущий, в первом аргументе функции `setTimeout()` передается `arguments.callee`. Это значение указывает на анонимную функцию, в которой находится поток выполнения в данный момент. Если в массиве не осталось элементов для обработки, вызывается функция `callback()`.

Фактическая величина задержки каждого таймера в значительной степени зависит от конкретных условий. В общем случае для выполнения маленьких задержек рекомендуется использовать значение не менее 25 мс, потому что меньшие задержки оставляют слишком короткие интервалы времени для обновления пользовательского интерфейса.

Поскольку для реализации этого шаблона требуется больше программного кода, чем для обычного цикла, его полезно заключить в функцию. Например:

```
function processArray(items, process, callback){
    var todo = items.concat(); // создать копию оригинала

    setTimeout(function(){
        process(todo.shift());

        if (todo.length > 0){
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

Функция `processArray()`, реализующая шаблон, представленный выше, и позволяющая использовать его многократно, принимает три аргумента: обрабатываемый массив, функцию обработки каждого элемента и функцию обратного вызова, которую следует выполнить по окончании обработки массива. Ниже приводится пример использования этой функции:

```
var items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];

function outputValue(value){
    console.log(value);
}

processArray(items, outputValue, function(){
    console.log("Done!");
});
```

В этом примере функция `processArray()` используется для вывода в консоль значений элементов массива и завершающего сообщения по окончании обработки. Благодаря заключению программного кода, выполняющего операции с таймером, в функцию, его можно многократно использовать в разных местах программы без необходимости повторять снова и снова.

Одним из побочных эффектов обработки массивов с помощью таймеров является увеличение времени обработки. Это объясняется тем, что после обработки каждого элемента управление передается главному потоку выполнения, из-за чего возникает задержка перед началом обработки следующего элемента. Однако это совершенно необходимо, чтобы не создавать у пользователя негативные впечатления, возникающие из-за блокировки броузера.

Деление заданий

Нередко одно большое задание можно разделить на последовательность более мелких заданий. Если единственная функция выполняется слишком долго, следует проверить возможность разделить ее на последова-

тельность более мелких функций, выполняющихся быстрее. Часто деление легко можно выполнить, если рассматривать каждую строку программного кода как некоторое атомарное задание, даже когда несколько строк программного кода сгруппированы для решения единой задачи. Некоторые функции легко можно разбить по функциям, которые они вызывают. Например:

```
function saveDocument(id){

    // сохранить документ
    openDocument(id)
    writeText(id);
    closeDocument(id);

    // обновить пользовательский интерфейс, чтобы сообщить об успехе
    updateUI(id);
}
```

Если эта функция будет выполняться слишком долго, ее легко можно разбить на последовательность более мелких шагов, организовав вызов отдельных методов с помощью таймеров. Добиться этого можно, добавив каждую функцию в массив и использовав шаблон, напоминающий шаблон обработки массива из предыдущего раздела:

```
function saveDocument(id){

    var tasks = [openDocument, writeText, closeDocument, updateUI];

    setTimeout(function(){

        // выполнить следующее задание
        var task = tasks.shift();
        task(id);

        // проверить наличие других заданий
        if (tasks.length > 0){
            setTimeout(arguments.callee, 25);
        }
    }, 25);
}
```

Эта версия функции помещает необходимые методы в массив `tasks` и затем вызывает их по одному с помощью таймеров. По сути, это тот же самый шаблон обработки массивов, с той лишь разницей, что обработка элементов связана с вызовом функций, содержащихся в них. Как демонстрировалось в предыдущем разделе, этот шаблон можно заключить в функцию для многократного использования:

```
function multistep(steps, args, callback){

    var tasks = steps.concat(); // скопировать массив

    setTimeout(function(){
```

```
            // выполнить следующее задание
            var task = tasks.shift();
            task.apply(null, args || []);

            // проверить наличие других заданий
            if (tasks.length > 0){
                setTimeout(arguments.callee, 25);
            } else {
                callback();
            }
        }, 25);
    }
```

Функция `multistep()` принимает три аргумента: массив функций, массив аргументов для передачи функциям и функцию обратного вызова, которая должна быть выполнена по завершении. Ниже приводится пример использования этой функции:

```
function saveDocument(id){

    var tasks = [openDocument, writeText, closeDocument, updateUI];
    multistep(tasks, [id], function(){
        alert("Save completed!");
    });
}
```

Обратите внимание, что второй аргумент функции `multistep()` должен быть массивом, поэтому при ее вызове создается массив с единственным элементом `id`. Как и в случае с обработкой массивов, данную функцию лучше использовать, когда асинхронное выполнение задания не повлечет за собой ошибки в зависящем от него программном коде и когда иной способ его выполнения может негативно отразиться на впечатлениях пользователей.

Хронометраж выполнения программного кода

Иногда выполнение по одному заданию за раз оказывается неэффективным. Представьте массив из 1000 элементов, обработка каждого из которых занимает 1 мс. Если каждый элемент обрабатывать с помощью таймера и использовать задержку 25 мс, то общее время обработки массива составит $(25 + 1) \times 1000 = 26000$ мс или 26 секунд. А что если попробовать обрабатывать массив пакетами по 50 элементов с задержкой 25 мс между ними? Тогда общее время обработки составит $(1000/50) + + + 25 + + 1000 = 1500$ мс или 1,5 секунды, и пользователь не будет чувствовать задержек в обновлении интерфейса, потому что обработка одного пакета будет занимать всего 50 мс. Обработка пакетами обычно выполняется быстрее, чем обработка по одному элементу.

Помня о 100 мс как об абсолютном максимуме интервала времени, в течение которого допускается непрерывное выполнение программного кода на JavaScript, можно оптимизировать предыдущие шаблоны. Я реко-

мендую уменьшить это число вдвое и никогда не позволять JavaScript-сценариям непрерывно выполняться дольше 50 мс, чтобы надежно гарантировать, что продолжительность выполнения программного кода никогда не окажется рядом с границей, переход за которую может создавать негативные впечатления.

Продолжительность выполнения программного кода можно определить с помощью встроенного объекта `Date`. Таким способом выполняется большая часть работы по профилированию программного кода:

```
var start = +new Date(),
    stop;

someLongProcess();

stop = +new Date();

if(stop-start < 50){
    alert("Just about right.");
} else {
    alert("Taking too long.");
}
```

Поскольку каждый новый объект `Date` инициализируется текущим системным временем, хронометраж выполнения программного кода можно производить, периодически создавая новые объекты `Date` и сравнивая их значения. Оператор сложения (`+`) преобразует объект `Date` в числовое представление, благодаря чему исключаются все последующие арифметические преобразования. Этот же простой прием можно использовать для оптимизации предыдущих шаблонов, основанных на таймерах.

Метод `processArray()` можно дополнить пакетной обработкой элементов массива, добавив проверку времени:

```
function timedProcessArray(items, process, callback){
    var todo = items.concat(); // создать копию оригинала

    setTimeout(function(){
        var start = +new Date();

        do {
            process(todo.shift());
        } while (todo.length > 0 && (+new Date() - start < 50));

        if (todo.length > 0){
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

Дополнительный цикл `do-while` в этой функции проверяет время после обработки каждого элемента. При вызове функции таймером массив всегда будет содержать хотя бы один элемент, поэтому в данном случае цикл с постусловием подходит больше, чем цикл с предусловием. При выполнении в Firefox 3 эта функция обрабатывает массив из 1000 элементов, где `process()` является пустой функцией, за 38–43 мс; первоначальная версия функции `processArray()` обрабатывает тот же массив более 25000 мс. Хронометраж заданий перед разделением их на более мелкие фрагменты обеспечил весьма существенное ускорение.

Таймеры и производительность

Таймеры могут оказывать огромное влияние на рост производительности программного кода, но злоупотребление ими может иметь негативные последствия. Примеры в этом разделе используют последовательности таймеров так, что в каждый момент времени используется только один таймер, а новый создается только после срабатывания предыдущего. Такое использование таймеров не оказывает отрицательного влияния на производительность.

Проблемы с производительностью начинают проявляться, когда одновременно создается сразу несколько таймеров многократного срабатывания. Поскольку в броузере имеется всего один главный поток выполнения, таймеры начинают конкурировать друг с другом за время, необходимое для выполнения переданных им функций. Нейл Томас (Neil Thomas) из Google Mobile исследовал эту тему, измеряя производительность мобильного приложения Gmail для iPhone и Android.[1]

Томас обнаружил, что таймеры с большим периодом срабатывания – от одной секунды и больше – оказывают незначительное влияние на общее время отклика веб-приложения. Задержки таймеров в этом случае оказываются слишком большими, чтобы оказать отрицательное влияние на производительность главного потока выполнения, и потому с такими задержками без опаски можно использовать таймеры многократного срабатывания. Однако используя несколько таймеров многократного срабатывания с более короткими задержками (от 100 до 200 мс), Томас обнаружил, что мобильное приложение Gmail стало заметно менее отзывчивым и выполнялось медленнее.

Из исследований Томаса следует вывод, что в веб-приложениях необходимо ограничивать количество таймеров многократного срабатывания с короткими задержками. Вместо них Томас предлагает создать единственный таймер многократного срабатывания, выполняющий множество операций при каждом срабатывании.

---

[1] Полный отчет доступен по адресу *http://googlecode.blogspot.com/2009/07/gmail-for-mobile-html5-series-using.html*.

Фоновые потоки выполнения

В то время когда язык JavaScript только появился, в броузерах отсутствовала возможность выполнения программного кода за пределами главного потока выполнения. Прикладной интерфейс фоновых потоков выполнения (Web Workers API) изменил положение дел, предоставив интерфейс, посредством которого можно выполнять программный код, не отнимая время у главного потока выполнения. Определение Web Workers API первоначально входило в состав стандарта HTML 5, а затем было выделено в отдельную спецификацию (*http://www.w3.org/TR/workers/*). Поддержка фоновых потоков выполнения уже реализована в Firefox 3.5, Chrome 3 и Safari 4.

Фоновые потоки потенциально способны существенно повысить производительность веб-приложений благодаря тому, что каждый новый объект `Worker` порождает собственный поток выполнения, в котором выполняется программный код на JavaScript. Это означает, что программный код, выполняемый в фоновом потоке, не оказывает влияния не только на главный поток выполнения, но и на код, выполняемый в других фоновых потоках.

Окружение фонового потока выполнения

Поскольку фоновые потоки выполняются независимо от потока пользовательского интерфейса, они не имеют доступа к многочисленным ресурсам броузера. Одна из причин, почему выполнение JavaScript-сценариев и обновление пользовательского интерфейса производится в одном и том же процессе, состоит в том, что зачастую они тесно связаны друг с другом, и выполнение их не в том порядке может вызвать отрицательные впечатления у пользователя. Фоновые потоки могли бы стать источником множества ошибок в пользовательском интерфейсе, если бы имели возможность манипулировать деревом DOM из-за пределов главного потока выполнения, но этого не происходит, потому что каждый фоновый поток выполнения имеет собственное глобальное окружение, которому доступна только часть возможностей JavaScript. Окружение фонового потока выполнения включает в себя следующее:

- Объект `navigator`, имеющий всего четыре свойства: `appName`, `appVersion`, `userAgent` и `platform`.
- Объект `location` (тот же, что и в объекте `window`, за исключением того, что все его свойства доступны только для чтения).
- Объект `self`, ссылающийся на глобальный объект фонового потока.
- Метод `importScripts()` для загрузки внешних JavaScript-библиотек, используемых в фоновом потоке.
- Все объекты, предусматриваемые стандартом ECMAScript, такие как `Object`, `Array`, `Date` и другие.
- Конструктор `XMLHttpRequest()`.

- Методы `setTimeout()` и `setInterval()`.
- Метод `close()`, немедленно останавливающий работу фонового потока.

Поскольку каждый фоновый поток выполнения имеет собственное глобальное окружение, нельзя создать фоновый поток из любого программного кода JavaScript. То есть необходимо создать отдельный JavaScript-файл, содержащий только программный код, который будет выполняться в фоновом потоке. Чтобы создать фоновый поток, необходимо передать конструктору URL-адрес JavaScript-файла:

```
var worker = new Worker("code.js");
```

Эта инструкция создаст для указанного файла новый поток выполнения с новым окружением. Загрузка файла выполняется асинхронно, и новый поток не будет запущен на выполнение, пока файл не будет загружен и выполнен полностью.

Взаимодействие с фоновыми потоками выполнения

Взаимодействия между фоновым потоком выполнения и веб-страницей выполняются посредством интерфейса событий. Сценарий в веб-странице может передать данные фоновому потоку выполнения, вызвав его метод `postMessage()`, принимающий единственный аргумент с данными для передачи фоновому потоку. Для приема информации внутри фонового потока используется обработчик события `onmessage`. Например:

```
var worker = new Worker("code.js");
worker.onmessage = function(event){
    alert(event.data);
};
worker.postMessage("Nicholas");
```

Данные фоновому потоку будут переданы вместе с событием `message`. Обработчик события `onmessage` получит объект события со свойством `data`, содержащим отправленные данные. Отправить информацию обратно в веб-страницу фоновый поток может с помощью собственного метода `postMessage()`:

```
// внутри code.js
self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

Получившаяся в результате строка будет передана обработчику события `onmessage` объекта `worker`. Такая система обмена сообщениями является единственным способом организации обмена информацией между веб-страницей и фоновым потоком выполнения.

С помощью метода `postMessage()` допускается передавать данные только определенных типов: элементарные значения (строки, числа, логические значения, `null` и `undefined`), а также экземпляры объектов `Object` и `Array`; данные других типов передавать не допускается. Данные допустимого типа сериализуются, передаются фоновому потоку или принимаются

от него и затем преобразуются в обычное представление. Хотя на первый взгляд создается впечатление, что объекты передаются непосредственно, тем не менее передаваемые экземпляры являются отдельными представлениями одних и тех же данных. Попытка передать данные неподдерживаемого типа приведет к возбуждению исключения.

Реализация фоновых потоков выполнения в Safari 4 позволяет передавать с помощью метода `postMessage()` только строки. Уже после появления этого броузера в спецификацию были внесены изменения, предусматривающие сериализацию данных при передаче и учтенные в реализации фоновых потоков в Firefox 3.5.

Загрузка внешних файлов

Загрузка внешних JavaScript-файлов в фоновом потоке выполняется с помощью метода `importScripts()`, который принимает один или более URL-адресов JavaScript-файлов, подлежащих загрузке. Вызов метода `importScripts()` блокирует работу фонового потока, то есть сценарий продолжит выполнение только после загрузки и выполнения всех файлов. Поскольку фоновый поток выполняется за пределами главного потока выполнения, эта блокировка никак не сказывается на отзывчивости пользовательского интерфейса. Например:

```
// внутри code.js
importScripts("file1.js", "file2.js");

self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

Первая строка в этом примере подключает два JavaScript-файла, благодаря чему они будут доступны в контексте фонового потока.

Практическое использование

Фоновые потоки прекрасно подходят для выполнения продолжительных операций над простыми данными и данными, не связанными с пользовательским интерфейсом. На первый взгляд фоновые потоки имеют весьма ограниченную область применения, но многие веб-приложения содержат задачи обработки данных, для решения которых вместо таймеров с успехом можно использовать фоновые потоки выполнения.

Рассмотрим для примера синтаксический анализ длинной строки в формате JSON (синтаксический анализ данных в формате JSON рассматривается далее в главе 7). Допустим, что объем данных настолько велик, что их обработка занимает не менее 500 мс. Это слишком долго, чтобы позволить такому JavaScript-сценарию выполняться на стороне клиента, потому что он будет создавать негативное впечатление у пользователя. Данную задачу трудно разбить на части с помощью таймеров, поэтому

применение фонового потока выполнения в этой ситуации является идеальным решением. Следующий фрагмент демонстрирует реализацию этого решения в веб-странице:

```
var worker = new Worker("jsonparser.js");

// когда данные будут обработаны, будет вызван этот обработчик события
worker.onmessage = function(event){

    // структура JSON, переданная обратно
    var jsonData = event.data;

    // использование структуры JSON
    evaluateData(jsonData);
};

// передать длинную строку в формате JSON для обработки
worker.postMessage(jsonText);
```

Ниже демонстрируется программный код, выполняющий обработку строки в формате JSON в фоновом потоке выполнения:

```
// внутри jsonparser.js
// этот обработчик события вызывается при передаче данных в формате JSON
self.onmessage = function(event){

    // строка в формате JSON поступает в виде значения свойства event.data
    var jsonText = event.data;

    // обработать данные
    var jsonData = JSON.parse(jsonText);

    // отправить результаты обратно
    self.postMessage(jsonData);
};
```

Обратите внимание, что хотя вызов метода `JSON.parse()` займет 500 мс или больше, в данном случае нет необходимости предусматривать дополнительный программный код, который делил бы обработку на более мелкие фрагменты. Обработка будет производиться в отдельном потоке выполнения, поэтому она может продолжаться столько, сколько потребуется, не оказывая влияния на впечатления пользователя.

Страница передает строку в формате JSON фоновому потоку выполнения с помощью метода `postMessage()`. Фоновый поток получает строку в виде значения свойства `event.data` в своем обработчике события `onmessage` и обрабатывает ее. По завершении полученный объект JSON передается обратно странице с помощью вызова метода `postMessage()` внутри фонового потока выполнения. Этот объект в свою очередь поступает в обработчик события `onmessage` страницы в виде значения свойства `event.data`. Имейте в виду, что пока такой подход можно использовать только в Firefox 3.5 и выше, потому что реализации в Safari 4 и Chrome 3 позволяют передавать между страницей и фоновым потоком только строки.

Обработка длинных строк – это лишь один из множества примеров задач, для решения которых с успехом можно использовать фоновые потоки выполнения. В числе других подобных задач можно назвать:

- Кодирование и декодирование длинных строк
- Сложные математические вычисления (включая обработку изображений и видеофайлов)
- Сортировку больших массивов

Всякий раз когда обработка занимает более 100 мс, следует подумать, не является ли решение на основе фоновых потоков выполнения более предпочтительным, чем решение на основе таймеров. Это, конечно, во многом зависит от возможностей броузера.

В заключение

Выполнение сценариев на языке JavaScript и обновление пользовательского интерфейса производятся в рамках одного и того же процесса, поэтому в каждый конкретный момент времени может выполняться только одно задание. Это означает, что пользовательский интерфейс не может реагировать на действия пользователя, пока выполняется программный код на JavaScript, и наоборот. Для успешного управления главным потоком выполнения необходимо гарантировать, что программный код на JavaScript не будет выполняться настолько долго, что это будет замечено пользователем. Для этого необходимо иметь в виду следующее:

- Никакая операция в JavaScript-сценарии не должна выполняться дольше 100 мс. Более длительные операции будут вызывать заметные задержки реакции пользовательского интерфейса и создавать негативные впечатления у пользователя.
- Броузеры по-разному реагируют на действия пользователя, производимые во время выполнения JavaScript-сценариев. Но независимо от поведения броузера у пользователя складываются отрицательные впечатления, когда выполнение сценария продолжается слишком долго.
- Применение таймеров позволяет отложить выполнение программного кода на более поздний срок, что дает возможность делить продолжительные операции на последовательности более мелких заданий.
- В новых версиях броузеров появился новый инструмент – фоновые потоки выполнения, с помощью которых можно выполнять программный код на JavaScript за пределами главного потока выполнения и тем самым предотвратить блокирование пользовательского интерфейса.

Чем сложнее веб-приложение, тем более важным становится активное управление главным потоком выполнения. Никакой программный код нельзя признать настолько важным, чтобы он негативно влиял на впечатления пользователя.

7

Ajax

Росс Хармс (Ross Harmes)

Технология Ajax является одним из важнейших механизмов повышения производительности JavaScript-сценариев. Ее применение может способствовать ускорению загрузки страницы за счет откладывания загрузки объемных ресурсов. С ее помощью можно предотвратить полную перезагрузку страницы, выполняя обмен данными между клиентом и сервером асинхронно. Ее можно использовать даже для извлечения всех ресурсов страницы в единственном HTTP-запросе. Выбирая наиболее подходящий способ передачи и наиболее эффективный формат представления данных, можно существенно улучшить процесс взаимодействия пользователя с сайтом.

В этой главе исследуются наиболее быстрые способы отправки данных и приема их с сервера, а также наиболее эффективные форматы представления данных.

Передача данных

На самом простом уровне Ajax – это способ обмена данными с сервером без необходимости выгрузки текущей страницы; данные могут запрашиваться у сервера или отправляться ему. Существует несколько способов создания такого канала обмена данными, каждый из которых имеет свои преимущества и ограничения. В этом разделе коротко исследуются разные подходы и обсуждается влияние каждого из них на производительность.

Запрос данных

Существует пять основных способов запросить данные с сервера:

- С помощью объекта XMLHttpRequest (XHR)
- С помощью динамических тегов `<script>`
- С помощью фреймов `<iframe>`
- С помощью технологии Comet
- Запрос составных данных с помощью объекта XHR

В современных высокопроизводительных JavaScript-сценариях используются три из них: с помощью объекта XHR, с помощью динамических тегов `<script>` и посредством запроса составных данных с помощью объекта XHR. Применение технологии Comet и фреймов `<iframe>` (в качестве механизма передачи данных) являются весьма ситуативными способами, которые не будут здесь рассматриваться.

XMLHttpRequest

Вне всяких сомнений, наиболее распространенным способом является применение объекта XMLHttpRequest (XHR), который позволяет отправлять и принимать данные асинхронно. Этот объект поддерживается всеми современными броузерами и обеспечивает возможность достаточно точного управления отправкой запросов и полученными данными. К отправляемым запросам (GET и POST) можно добавлять произвольные заголовки и параметры, а из ответов, возвращаемых сервером, можно извлечь не только текст ответа, но и все имеющиеся в нем заголовки. Ниже приводится пример использования этого объекта:

```
var url = '/data.php';
var params = [
    'id=934875',
    'limit=20'
];

var req = new XMLHttpRequest();

req.onreadystatechange = function() {
    if (req.readyState === 4) {
        var responseHeaders = req.getAllResponseHeaders(); // Извлечь заголовки
        var data = req.responseText;                       // Извлечь данные
        // Обработать данные...
    }
}

req.open('GET', url + '?' + params.join('&'), true);
req.setRequestHeader('X-Requested-With', 'XMLHttpRequest'); // Настроить
                                                            // заголовок запроса
req.send(null); // Отправить запрос
```

Этот пример демонстрирует, как выполнять запрос данных по URL-адресу с параметрами и как читать текст и заголовки ответа. Значение 4 в свойстве `readyState` свидетельствует о том, что ответ был получен полностью и доступен для использования.

Ответ сервера можно читать также и в процессе его получения при значении 3 в свойстве `readyState`. Такой прием называется потоковой обработкой данных и является мощным инструментом повышения производительности выполняемых запросов:

```
req.onreadystatechange = function() {

   if (req.readyState === 3) {      // Получены некоторые, но не все данные
      var dataSoFar = req.responseText;
      ...
   }
   else if (req.readyState === 4) { // Получены все данные
      var data = req.responseText;
      ...
   }
}
```

Из-за высокой степени управляемости, которую обеспечивает объект XHR, броузеры накладывают на него некоторые ограничения. Объект XHR нельзя использовать для запроса данных из другого домена, отличного от домена текущей страницы, а старые версии IE не присваивают свойству `readyState` значение 3, что не позволяет использовать этот объект для получения потоковых данных. Данные, полученные в ответ на запрос, могут интерпретироваться либо как строка, либо как XML-документ; это означает, что большие объемы данных будут обрабатываться достаточно медленно.

Несмотря на эти недостатки, способ на основе объекта XHR все еще остается наиболее мощным и чаще других используется для запроса данных. При необходимости организовать обмен данными его следует рассматривать одним из первых.

Сравнение запросов POST и GET при использовании объекта XHR. Когда для запроса данных используется объект XHR, приходится выбирать, какой тип запроса использовать, POST или GET. Запросы, не изменяющие данные на сервере и только инициирующие получение данных (это называется *идемпотентной операцией*), должны выполняться методом GET. GET-запросы кэшируются броузером, что повышает производительность при многократном получении одних и тех же данных.

POST-запросы следует использовать для получения данных, только когда длина строки с URL-адресом и параметрами приближается к 2048 символам или превышает это число. Это обусловлено тем, что Internet Explorer ограничивает длину URL-адреса этим количеством символов и при превышении этого ограничения запрос может оказаться усеченным.

Динамические теги <script>

Этот прием позволяет преодолеть самое большое ограничение объекта XHR: он позволяет обращаться за данными к серверу в другом домене. Это довольно грубый прием; вместо специализированного объекта программный код на JavaScript создает новый тег `<script>` и присваивает его атрибуту `source` строку с URL-адресом сервера в другом домене.

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);
```

Однако способ на основе динамических тегов `<script>` не обеспечивает такой же уровень управления запросами, как объект XHR. Он не позволяет настраивать заголовки в запросе. Параметры могут передаваться только методом GET – метод POST недоступен. Он не позволяет определить предельное время ожидания или повторить запрос; в действительности, невозможно даже определить факт успешного выполнения запроса. Необходимо ждать получения всех данных, прежде чем к ним можно будет получить доступ. Отсутствует доступ к заголовкам ответа или ко всему ответу в виде строки.

Последнее особенно важно. Поскольку содержимое ответа будет добавлено в тег `<script>`, оно *должно* быть сценарием на языке JavaScript. Этим способом нельзя получить XML-документ или даже данные в формате JSON; любые данные, независимо от формата, должны быть заключены в функцию обратного вызова.

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);

function jsonCallback(data) {
    // Обработать данные...
}
```

В этом примере файл *lib.js* должен заключать данные в функции `jsonCallback`:

```
jsonCallback({ "status": 1, "colors": [ "#fff", "#000", "#ff0000" ] });
```

Несмотря на описанные ограничения, данный прием является чрезвычайно быстрым. Ответ сервера выполняется как программный код на языке JavaScript; он не интерпретируется как строка, требующая дополнительной обработки. Благодаря этой своей особенности данный способ является самым быстрым способом получения данных и преобразования их в некоторый вид, в котором они будут доступны на стороне клиента. Сравнение производительности способа на основе динамических тегов `<script>` с производительностью объекта XHR будет проведено далее в этой главе в разделе с описанием формата JSON.

Не следует использовать этот прием для получения данных с посторонних неподконтрольных вам серверов. В языке JavaScript отсутствует

такое понятие, как права доступа, поэтому любой программный код, встраиваемый в страницу с помощью динамических тегов <script>, будет иметь неограниченный доступ к странице. Сюда входят: возможность изменять ее содержимое, выполнять переход на другой сайт и даже следить за действиями пользователя на странице и отправлять данные на сторонний сервер. Программный код, полученный из внешних источников, следует использовать с особой осторожностью.

Запрос составных данных с помощью объекта XHR

Прием запроса составных данных с помощью объекта XHR является самым новым из рассматриваемых здесь; он позволяет получать от сервера сразу несколько ресурсов, выполнив единственный HTTP-запрос. Это достигается за счет упаковывания ресурсов (которые могут быть CSS-файлами, фрагментами разметки HTML, программным кодом на языке JavaScript или изображениями, закодированными в формате base64) на стороне сервера и передаче их клиенту в виде длинной строки символов, где они отделяются друг от друга некоторой согласованной последовательностью символов. JavaScript-сценарий обрабатывает эту строку и анализирует каждый ресурс в соответствии с его MIME-типом и любыми другими «заголовками», переданными вместе с ним.

Проследим этот процесс от начала до конца. Сначала выполняется запрос на получение нескольких изображений:

```
var req = new XMLHttpRequest();

req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = function() {
    if (req.readyState == 4) {
        splitImages(req.responseText);
    }
};
req.send(null);
```

Это очень простой запрос. В этом примере выполняется запрос данных у сценария *rollup_images.php*, которые после получения передаются функции `splitImages()`.

Затем на стороне сервера изображения читаются из файлов и преобразуются в строки:

```
// Прочитать изображения и преобразовать их в строки в формате base64.

$images = array('kitten.jpg', 'sunset.jpg', 'baby.jpg');
foreach ($images as $image) {

    $image_fh = fopen($image, 'r');
    $image_data = fread($image_fh, filesize($image));

    fclose($image_fh);
        $payloads[] = base64_encode($image_data);
```

```
    }
}

// Упаковать эти строки в одну длинную строку и вывести ее.

$newline = chr(1); // Этот символ не может присутствовать в строках,
                   // закодированных в формат base64
echo implode($newline, $payloads);
```

Этот фрагмент программного кода на языке PHP читает три файла с изображениями и преобразует их содержимое в длинные строки с символами в кодировке base64. Затем они объединяются в одну строку с использованием разделителя, состоящего из единственного символа Юникода с кодом 1, и получившаяся строка отправляется клиенту.

На стороне клиента данные обрабатываются с помощью функции `splitImages()`:

```
function splitImages(imageString) {

    var imageData = imageString.split("\u0001");
    var imageElement;

    for (var i = 0, len = imageData.length; i < len; i++) {

        imageElement = document.createElement('img');
        imageElement.src = '' + imageData[i];
        document.getElementById('container').appendChild(imageElement);
    }
}
```

Эта функция получает объединенную строку и разбивает ее обратно на три части. Затем каждая часть используется для создания элемента изображения, и эти элементы вставляются в страницу. Изображение *не* преобразуется из строк в формате base64 обратно в двоичное представление; вместо этого строки передаются элементам изображений посредством URL-адреса `data:` с указанием MIME-типа `image/jpeg`.

В результате броузер получает три изображения, выполнив единственный HTTP-запрос. С таким же успехом можно было получить 20 или 100 изображений; строка ответа получилась бы больше, но для ее получения потребовалось бы выполнить всего один HTTP-запрос. Этот прием можно распространить и на другие типы ресурсов. В один ответ можно объединить JavaScript-файлы, CSS-файлы, фрагменты с разметкой HTML и изображения различных типов. Передать можно данные любых типов, которые могут обрабатываться JavaScript-сценариями как строки. Ниже демонстрируется функция, принимающая строку с программным кодом на языке JavaScript, стилями CSS и изображениями и преобразующая их в ресурсы, которые могут использоваться броузером:

```
function handleImageData(data, mimeType) {
    var img = document.createElement('img');
```

```
    img.src = 'data:' + mimeType + '; base64,' + data;
    return img;
}

function handleCss(data) {
    var style = document.createElement('style');
    style.type = 'text/css';

    var node = document.createTextNode(data);
    style.appendChild(node);
    document.getElementsByTagName('head')[0].appendChild(style);
}

function handleJavaScript(data) {
    eval(data);
}
```

С увеличением объема составных данных возникает необходимость обрабатывать каждый ресурс по мере его получения, а не ждать, пока ответ будет получен полностью. Такая обработка может выполняться, когда свойство `readyState` **получит значение 3:**

```
var req = new XMLHttpRequest();
var getLatestPacketInterval, lastLength = 0;

req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = readyStateHandler;
req.send(null);

function readyStateHandler{
    if (req.readyState === 3 && getLatestPacketInterval === null) {

        // Начать потоковую обработку

        getLatestPacketInterval = window.setInterval(function() {
            getLatestPacket();
        }, 15);
    }

    if (req.readyState === 4) {

        // Остановить потоковую обработку

        clearInterval(getLatestPacketInterval);

        // Получить последний пакет

        getLatestPacket();
    }
}

function getLatestPacket() {
```

```
        var length = req.responseText.length;
        var packet = req.responseText.substring(lastLength, length);

        processPacket(packet);
        lastLength = length;
    }
```

Как только свойство `readyState` впервые получит значение 3, запускается таймер. Обработчик событий от этого таймера каждые 15 мс проверяет появление новых данных. Каждый новый фрагмент данных добавляется в конец буфера, пока не будет встречен символ-разделитель, после чего содержимое буфера обрабатывается как полный ресурс.

Надежная реализация запроса и получения составных данных с помощью объекта XHR выглядит достаточно сложной, но достойной дальнейшего изучения. Полную библиотеку с указанной реализацией легко можно найти в Интернете по адресу *http://techfoolery.com/mxhr/*.

Этот прием имеет несколько недостатков, самый большой из которых заключается в том, что ни один из ресурсов, полученных таким способом, не кэшируется броузером. Если какой-то CSS-файл загружается методом получения составных данных и снова используется другой страницей как самостоятельный ресурс, он не будет повторно загружен из Сети. Это объясняется тем, что упакованные вместе ресурсы передаются в виде одной большой строки и затем разделяются JavaScript-сценарием. А поскольку программный способ добавлять файлы в кэш броузера отсутствует, ни один из полученных ресурсов туда не попадает.

Другой недостаток связан с отсутствием поддержки значения 3 для свойства `readyState` и URL-адресов `data:` в старых версиях Internet Explorer. Обе эти особенности реализованы в Internet Explorer 8, но для поддержки Internet Explorer 6 и 7 необходимо предусматривать обходные решения.

И все же, несмотря на описанные недостатки, существуют ситуации, когда метод получения составных данных способен существенно повысить производительность страницы:

- Страницы с большим количеством ресурсов, которые нигде на сайте больше не используются (и которые по этой причине не должны кэшироваться), особенно это относится к изображениям.
- Сайты, уже использующие комплекты JavaScript-сценариев или CSS-файлов, уникальные для каждой страницы, с целью уменьшить количество HTTP-запросов; поскольку комплекты этих ресурсов являются уникальными для каждой страницы, они никогда не будут читаться из кэша, если конкретная страница не будет полностью перезагружена.

Поскольку HTTP-запросы являются самым узким местом в Ajax, уменьшение их количества оказывает существенное влияние на общую производительность страницы. Это особенно верно, когда появляется возможность заменить 100 запросов на получение изображений единст-

венным запросом на получение составных данных. Специальное тестирование показало, что такой способ загрузки большого количества изображений позволяет сократить время загрузки от 4 до 10 раз по сравнению с приемом загрузки изображений по отдельности. Вы можете сами провести такое тестирование, обратившись по адресу *http://techfoolery.com/mxhr/*.

Отправка данных

Иногда бывает необходимо не получить, а отправить данные на сервер. Например, на сервер можно было бы отправлять неличную информацию о пользователях для последующего анализа или сведения об ошибках, возникающих в сценарии, для регистрации и предупреждения разработчиков. Когда требуется всего лишь отправить данные на сервер, обычно используются два способа: с помощью объекта XHR и с помощью «сигналов».

XMLHttpRequest

Несмотря на то что основная задача объекта XHR – получение данных с сервера, его также можно использовать для отправки данных на сервер. Данные можно отправлять в виде GET- или POST-запросов, а также в различных HTTP-заголовках. Это обеспечивает существенную гибкость. Объект XHR особенно удобно использовать, когда объем отправляемых данных превышает максимально возможную длину URL-адреса. В подобных ситуациях данные можно отправлять в виде POST-запроса:

```
var url = '/data.php';
var params = [
    'id=934875',
    'limit=20'
];

var req = new XMLHttpRequest();

req.onerror = function() {
    // Ошибка.
};

req.onreadystatechange = function() {
    if (req.readyState == 4) {
        // Успех.
    }
};

req.open('POST', url, true);
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

var data = params.join("&");
req.setRequestHeader("Content-Length", data.length);
req.send(data);
```

Как видно из этого примера, сценарий ничего не делает в случае неудачи. Такое решение неплохо подходит, когда объект XHR используется для сбора статистической информации о пользователях. Но когда важно обеспечить надежную доставку данных на сервер, можно добавить программный код, повторяющий попытку отправки в случае неудачи:

```
function xhrPost(url, params, callback) {

    var req = new XMLHttpRequest();

    req.onerror = function() {
        setTimeout(function() {
            xhrPost(url, params, callback);
        }, 1000);
    };

    req.onreadystatechange = function() {
        if (req.readyState == 4) {
            if (callback && typeof callback === 'function') {
                callback();
            }
        }
    };

    req.open('POST', url, true);
    req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

    var data = params.join("&");
    req.setRequestHeader("Content-Length", data.length);
    req.send(data);
}
```

Когда для отправки данных используется объект XHR, более высокая скорость достигается при использовании метода GET. Это объясняется небольшим объемом отправляемых данных – GET-запрос отправляется на сервер в виде единственного пакета. POST-запрос, напротив, отправляется как минимум в виде двух пакетов: в одном отправляются заголовки, а в другом – тело POST-запроса. POST-запросы лучше подходят для отправки больших объемов данных, во-первых, потому что отправка дополнительных пакетов в этом случае будет сказываться на производительности незначительно, и во-вторых, из-за ограничения на длину URL-адреса в Internet Explorer, которое делает невозможным отправку больших объемов данных в GET-запросах.

Сигналы

Этот прием очень напоминает прием на основе динамических тегов `<script>`. JavaScript-сценарий создает новый объект `Image`, присваивает его свойству `src` URL-адрес сценария на стороне сервера. Этот URL-адрес должен содержать отправляемые данные в формате GET-запроса, то

есть в виде пар ключ-значение. Обратите внимание, что при этом не создается элемент <img> и он не добавляется в дерево DOM.

```
var url = '/status_tracker.php';
var params = [
    'step=2',
    'time=1248027314'
];

(new Image()).src = url + '?' + params.join('&');
```

Сервер просто принимает и сохраняет данные – он ничего не отправляет обратно клиенту, потому что созданный объект изображения не предназначен для отображения. Это наиболее эффективный способ отправки данных на сервер. Он имеет весьма небольшие накладные расходы, а ошибки на стороне сервера никак не влияют на клиента.

Простота отправки сигналов с помощью объектов изображений также означает ограниченность этого приема. С его помощью нельзя выполнять POST-запросы, поэтому он может использоваться для отправки лишь небольшого количества символов, ограниченного максимальной длинной URL-адреса. Этот прием *позволяет* организовать прием данных, но весьма ограниченным количеством способов. Можно определить обработчик события load в объекте Image, который будет вызван в случае успешного получения данных со стороны сервера. Можно также проверить высоту и ширину изображения, полученного от сервера (если возвращается изображение), и использовать эти числа как признак состояния сервера. Например, ширина, равная 1, может означать «успех», а равная 2, – необходимость «повторить попытку».

Если от сервера не требуется отправлять данные в ответ на запрос, он должен отправить код ответа 204 No Content без тела сообщения. Это избавит клиента от необходимости ожидать тело ответа, которое никогда не будет послано:

```
var url = '/status_tracker.php';
var params = [
    'step=2',
    'time=1248027314'
];

var beacon = new Image();
beacon.src = url + '?' + params.join('&');

beacon.onload = function() {
    if (this.width == 1) {
        // Успех.
    }
    else if (this.width == 2) {
        // Неудача; повторить попытку отправки сигнала.
    }
};
```

```
beacon.onerror = function() {
    // Ошибка; подождать немного, затем снова отправить сигнал.
};
```

Прием отправки сигналов является самым быстрым и самым эффективным способом передачи данных на сервер. Сервер не отправляет в теле ответа никаких данных, поэтому на стороне клиента не приходится беспокоиться о загрузке данных. Единственный недостаток этого способа в том, что он позволяет получать лишь ограниченный круг типов ответов. При необходимости возвращать клиенту большие объемы данных лучше использовать объект XHR. Если требуется просто отправлять данные на сервер (возможно, с простым ответом), лучше использовать прием отправки сигналов.

Форматы данных

При выборе способа передачи данных необходимо принимать во внимание несколько факторов: широту возможностей, совместимость с разными броузерами, производительность и направление (на сервер или с сервера). При выборе формата данных достаточно будет учесть только один фактор – скорость.

Не существует такого формата данных, который всегда превосходил бы другие. В зависимости от того, какие данные передаются и как они используются страницей, один формат может быстрее загружаться, а другой – быстрее обрабатываться. В этом разделе будет создан визуальный элемент поиска среди пользователей, на основе которого будет продемонстрировано использование всех четырех основных форматов данных. Для этого необходимо реализовать: создание списка пользователей на стороне сервера, передачу его броузеру в ответ на запрос, преобразование этого списка в структуру данных на языке JavaScript и поиск требуемой строки. Рассматриваемые форматы данных будут сравниваться по размеру файла со списком, по скорости преобразования списка и простоте формирования списка на стороне сервера.

XML

Когда технология Ajax приобрела первую популярность, для обмена данными был выбран формат XML. Этот выбор был обусловлен несколькими факторами: полная совместимость (благодаря отличной поддержке этого формата как на стороне сервера, так и на стороне клиента), строгость форматирования и простота проверки. Формат JSON еще не был формализован как формат обмена данными, и почти для каждого языка программирования из числа используемых для создания серверных сценариев имелась библиотека для работы с форматом XML.

Ниже приводится пример списка пользователей, представленного в формате XML:

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
    <user id="1">
        <username>alice</username>
        <realname>Alice Smith</realname>
        <email>alice@alicesmith.com</email>
    </user>
    <user id="2">
        <username>bob</username>
        <realname>Bob Jones</realname>
        <email>bob@bobjones.com</email>
    </user>
    <user id="3">
        <username>carol</username>
        <realname>Carol Williams</realname>
        <email>carol@carolwilliams.com</email>
    </user>
    <user id="4">
        <username>dave</username>
        <realname>Dave Johnson</realname>
        <email>dave@davejohnson.com</email>
    </user>
</users>
```

В сравнении с другими форматами, XML выглядит далеко не самым компактным. Каждый элемент данных в этом формате должен быть представлен довольно подробной структурой, вследствие чего отношение объема данных к объему структуры оказывается слишком маленьким. Кроме того, язык XML имеет несколько неоднозначный синтаксис. Например, при преобразовании данных в формат XML как лучше представить параметры объектов – в виде атрибутов или в виде независимых дочерних элементов? Какие имена тегов выбрать – длинные и описательные или короткие и более эффективные при обработке, но совершенно непонятные? В любом случае порядок синтаксического анализа выглядит неоднозначным, и для его выполнения необходимо заранее знать структуру XML-ответа.

В общем случае для анализа данных в формате XML со стороны программиста на JavaScript требуется приложить немало усилий. Кроме необходимости знать заблаговременно конкретную структуру требуется также точно знать, как разделить эту структуру на составные части и кропотливо собрать их в объект на языке JavaScript. Это не самый простой и автоматизированный процесс, в отличие от трех других форматов данных.

Ниже приводится пример преобразования XML-ответа в объект:

```
function parseXML(responseXML) {

    var users = [];
    var userNodes = responseXML.getElementsByTagName('users');
    var node, usernameNodes, usernameNode, username,
```

```
            realnameNodes, realnameNode, realname,
            emailNodes, emailNode, email;

        for (var i = 0, len = userNodes.length; i < len; i++) {

            node = userNodes[i];
            username = realname = email = '';

            usernameNodes = node.getElementsByTagName('username');
            if (usernameNodes && usernameNodes[0]) {
                usernameNode = usernameNodes[0];
                username = (usernameNode.firstChild) ?
                    usernameNode.firstChild.nodeValue : '';
            }

            realnameNodes = node.getElementsByTagName('realname');
            if (realnameNodes && realnameNodes[0]) {
                realnameNode = realnameNodes[0];
                realname = (realnameNode.firstChild) ?
                    realnameNode.firstChild.nodeValue : '';
            }

            emailNodes = node.getElementsByTagName('email');
            if (emailNodes && emailNodes[0]) {
                emailNode = emailNodes[0];
                email = (emailNode.firstChild) ?
                    emailNode.firstChild.nodeValue : '';
            }

            users[i] = {
                id: node.getAttribute('id'),
                username: username,
                realname: realname,
                email: email
            };
        }

        return users;
    }
```

Как видно из этого примера, необходимо проверить наличие каждого тега, прежде чем пытаться прочитать его значение. Такая реализация оказывается жестко связанной со структурой XML.

Эффективнее было бы представить все значения в виде атрибутов тега `<user>`. Такой подход позволил бы уменьшить размер файла для того же объема данных. Ниже приводится пример списка пользователей, где все значения представлены в виде атрибутов:

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
    <user id="1" username="alice" realname="Alice Smith"
        email="alice@alicesmith.com" />
```

```
    <user id="2" username="bob" realname="Bob Jones"
        email="bob@bobjones.com" />
    <user id="3" username="carol" realname="Carol Williams"
        email="carol@carolwilliams.com" />
    <user id="4" username="dave" realname="Dave Johnson"
        email="dave@davejohnson.com" />
</users>
```

Анализ такого упрощенного XML-ответа реализуется значительно проще:

```
function parseXML(responseXML) {

    var users = [];
    var userNodes = responseXML.getElementsByTagName('users');

    for (var i = 0, len = userNodes.length; i < len; i++) {
        users[i] = {
            id: userNodes[i].getAttribute('id'),
            username: userNodes[i].getAttribute('username'),
            realname: userNodes[i].getAttribute('realname'),
            email: userNodes[i].getAttribute('email')
        };
    }

    return users;
}
```

XPath

Знакомство с языком XPath выходит далеко за рамки этой главы, тем не менее отмечу, что использование XPath может значительно ускорить синтаксический анализ XML-документов по сравнению с применением метода `getElementsByTagName()`. Недостатком этого метода является отсутствие повсеместной поддержки, что заставит дополнительно организовывать выполнение синтаксического анализа в старом стиле с применением методов обхода дерева DOM. На данный момент спецификация «DOM Level 3 XPath» реализована в Firefox, Safari, Chrome и Opera. Internet Explorer 8 имеет похожий, но менее функциональный интерфейс.

Размер ответа и время анализа

Взгляните на таблицу ниже, где приводятся результаты измерения производительности при использовании формата XML.

Формат	Размер	Время загрузки	Время анализа	Общее время
Подробный XML	582960 байт	999,4 мс	343,1 мс	1342,5 мс
Упрощенный XML	437960 байт	475,1 мс	83,1 мс	558,2 мс

Каждый из обсуждаемых форматов данных тестировался со списками, включающими 100, 500, 1000 и 5000 пользователей. Каждый список загружался и анализировался 10 раз в одном и том же броузере, а в качестве значений времени загрузки, анализа и размера файла брались средние значения. Полные результаты для всех форматов данных и приемов передачи можно найти на сайте *http://techfoolery.com/formats/*, где также можно провести свое тестирование.

Как видно из таблицы, использование атрибутов вместо дочерних тегов позволяет уменьшить размер файла и существенно ускорить синтаксический анализ. Это в основном обусловлено заменой операций перемещения XML-структуры по дереву DOM более простыми операциями чтения атрибутов.

Следует ли рассматривать возможность использования XML? Учитывая его распространенность, этот формат нередко оказывается единственно доступным. В таких ситуациях не остается ничего иного, как засучить рукава и написать функции для его анализа. Однако если имеется возможность использовать любой другой формат данных, предпочтение следует отдать ему. Производительность при использовании подробного XML, которая показана в таблице выше, намного ниже производительности более современных приемов. Для броузеров, поддерживающих такую возможность, применение языка XPath позволяет повысить скорость синтаксического анализа, но ценой реализации и сопровождения трех разных способов анализа (один – для броузеров, поддерживающих спецификацию «DOM Level 3 XPath», второй – для Internet Explorer 8 и третий – для всех остальных броузеров). Упрощенный формат XML выглядит более предпочтительным, но его обработка все равно выполняется медленнее, чем обработка других более быстрых форматов. Формат XML не годится для высокопроизводительных приложений, использующих технологию Ajax.

JSON

Формализованный и пропагандируемый Дугласом Крокфордом (Douglas Crockford) формат JSON является легковесным и простым в анализе форматом данных, в котором используется синтаксис литералов объектов и массивов языка JavaScript. Ниже приводится пример списка пользователей в формате JSON:

```
[
    { "id": 1, "username": "alice", "realname": "Alice Smith",
        "email": "alice@alicesmith.com" },
    { "id": 2, "username": "bob", "realname": "Bob Jones",
        "email": "bob@bobjones.com" },
    { "id": 3, "username": "carol", "realname": "Carol Williams",
        "email": "carol@carolwilliams.com" },
    { "id": 4, "username": "dave", "realname": "Dave Johnson",
```

```
        "email": "dave@davejohnson.com" }
]
```

Пользователи в списке представлены объектами, а сам список оформлен в виде массива, как любые другие литералы объектов или массивов в языке JavaScript. Это означает, что при передаче функции `eval()` или при обертывании функцией обратного вызова данные в формате JSON интерпретируются как выполняемый программный код на языке JavaScript. Чтобы выполнить синтаксический анализ строки в формате JSON, в сценарии на языке JavaScript достаточно просто вызвать функцию `eval()`:

```
function parseJSON(responseText) {
    return eval('(' + responseText + ')');
}
```

Примечание, касающееся использования функции `eval()` для анализа данных в формате JSON: использовать функцию `eval()` в сценариях небезопасно, особенно для анализа сторонних данных в формате JSON (которые могут содержать злонамеренный или некорректно работающий программный код). Всякий раз когда это возможно, для анализа строк следует использовать метод `JSON.parse()`. Этот метод перехватывает синтаксические ошибки в данных и позволяет передавать ему функцию для фильтрации и преобразования результатов. В настоящее время этот метод реализован в Firefox 3.5, Internet Explorer 8 и Safari 4. Большинство JavaScript-библиотек поддерживают возможность анализа данных в формате JSON, используя встроенный метод, если он имеется, или собственную, хотя и менее надежную, версию. Пример реализации такой альтернативной версии можно найти в библиотеке *http://json.org/json2.js*. Однако для единообразия в примерах ниже будет использоваться функция `eval()`.

Как и в случае с форматом XML, имеется возможность использовать упрощенную версию описываемого формата. В данном случае можно заменить имена атрибутов более короткими (хотя и менее удобочитаемыми) версиями:

```
[
    { "i": 1, "u": "alice", "r": "Alice Smith", "e": "alice@alicesmith.com" },
    { "i": 2, "u": "bob", "r": "Bob Jones", "e": "bob@bobjones.com" },
    { "i": 3, "u": "carol", "r": "Carol Williams",
        "e": "carol@carolwilliams.com" },
    { "i": 4, "u": "dave", "r": "Dave Johnson", "e": "dave@davejohnson.com" }
]
```

Это те же самые данные, но с более компактной структурой, занимающей меньшее количество байт, передаваемых броузеру. Можно пойти еще дальше и вообще убрать имена атрибутов. Такой формат – более сложный для восприятия, чем два предыдущих, и более хрупкий, но позволяет существенно уменьшить размер файла: почти в два раза по сравнению с первой версией списка в формате JSON.

```
[
    [ 1, "alice", "Alice Smith", "alice@alicesmith.com" ],
    [ 2, "bob", "Bob Jones", "bob@bobjones.com" ],
    [ 3, "carol", "Carol Williams", "carol@carolwilliams.com" ],
    [ 4, "dave", "Dave Johnson", "dave@davejohnson.com" ]
]
```

Для успешного анализа необходимо обеспечить поддержку порядка следования данных. С другой стороны, этот формат совсем несложно преобразовать в формат, поддерживающий те же имена атрибутов, что и первая версия формата JSON:

```
function parseJSON(responseText) {

    var users = [];
    var usersArray = eval('(' + responseText + ')');

    for (var i = 0, len = usersArray.length; i < len; i++) {
        users[i] = {
            id: usersArray[i][0],
            username: usersArray[i][1],
            realname: usersArray[i][2],
            email: usersArray[i][3]
        };
    }

    return users;
}
```

В этом примере для преобразования строки в массив используется функция `eval()`. Затем полученный массив массивов преобразуется в массив объектов. По сути, за маленький размер файла и использование быстрой функции `eval()` приходится платить более сложной функцией анализа. В следующей таблице приводятся результаты измерения производительности обработки списков в трех разных форматах JSON, передача которых осуществлялась с помощью XHR.

Формат	Размер	Время загрузки	Время анализа	Общее время
Подробный JSON	487895 байт	527,7 мс	26,7 мс	554,4 мс
Упрощенный JSON	392895 байт	498,7 мс	29,0 мс	527,7 мс
Массив JSON	292895 байт	305,6 мс	18,6 мс	324,0 мс

JSON-массивы побеждают во всех номинациях: их применение дает самый маленький размер файла, самую высокую среднюю скорость загрузки и самую высокую скорость анализа. Несмотря на то что функции анализа пришлось выполнить обход всех 5000 элементов списка, скорость анализа с ее использованием оказалась почти на 30% выше.

JSON-P

Тот факт, что данные в формате JSON могут выполняться интерпретатором как обычный программный код на языке JavaScript, имеет большое значение для производительности. При использовании объекта XHR данные в формате JSON возвращаются в виде строки. Затем эта строка анализируется с помощью функции `eval()` и преобразуется в обычный объект. Однако при использовании приема на основе динамических тегов `<script>` данные в формате JSON интерпретируются как обычный JavaScript-файл и выполняются как программный код на языке JavaScript. При этом сами данные должны быть обернуты функцией обратного вызова. Такое представление известно как «JSON with padding» (JSON с дополнением), или JSON-P. Ниже приводится пример списка пользователей в формате JSON-P:

```
parseJSON([
    { "id": 1, "username": "alice", "realname": "Alice Smith",
      "email": "alice@alicesmith.com" },
    { "id": 2, "username": "bob", "realname": "Bob Jones",
      "email": "bob@bobjones.com" },
    { "id": 3, "username": "carol", "realname": "Carol Williams",
      "email": "carol@carolwilliams.com" },
    { "id": 4, "username": "dave", "realname": "Dave Johnson",
      "email": "dave@davejohnson.com" }
]);
```

Использование формата JSON-P несколько увеличивает размер файла за счет добавления определения функции-обертки, но такое увеличение выглядит совсем несущественным в сравнении с повышением производительности синтаксического анализа. Поскольку данные интерпретируются как обычный программный код на языке JavaScript, их интерпретация выполняется с соответствующей скоростью. Ниже приводятся результаты измерения производительности обработки тех же трех форматов JSON, переданных как JSON-P.

Формат	Размер	Время загрузки	Время анализа	Общее время
Подробный JSON-P	487913 байт	598,2 мс	0,0 мс	598,2 мс
Упрощенный JSON-P	392913 байт	454,0 мс	3,1 мс	457,1 мс
Массив JSON-P	292912 байт	316,0 мс	3,4 мс	319,4 мс

Размеры файлов и время загрузки остались практически теми же, что и в тестах с использованием XHR, но время, требуемое для анализа, уменьшилось почти в 10 раз. Нулевое время анализа данных в подробном формате JSON-P получилось потому, что анализ в данном случае выполнять не потребовалось – данные уже поступили в требуемом виде.

То же относится и к двум другим версиям – упрощенному JSON-P и массиву JSON-P, но в этих случаях потребовалось выполнить итерации по элементам списка и преобразовать их в подробное представление, которое подробный формат JSON-P обеспечивает изначально.

Самым быстрым в обработке оказался формат JSON-P на основе массивов. Он обрабатывается лишь немногим быстрее, чем данные в формате JSON, переданные с помощью XHR, но эта разница становится более заметной с увеличением размера списка. Если вы работаете над проектом, в котором требуется передавать списки из 10000 или 100000 элементов, предпочтение следует отдавать формату JSON-P.

Однако существует одна причина избегать использования формата JSON-P, которая не связана с производительностью: поскольку данные в формате JSON-P должны представлять собой файл с выполняемым программным кодом на языке JavaScript, они могут быть загружены любым желающим и включены в страницы любого веб-сайта с применением приема динамических тегов `<script>`. Данные в формате JSON, напротив, не интерпретируются как программный код на языке JavaScript, пока не будут переданы функции `eval()`, и могут извлекаться только в виде строки с использованием объекта XHR. Не передавайте важные данные в формате JSON-P, потому что в этом случае нельзя гарантировать их конфиденциальность, даже если использовать случайные URL-адреса или cookies.

Следует ли использовать формат JSON?

Формат JSON имеет ряд преимуществ перед форматом XML. Он намного более компактный и позволяет передавать больше данных в меньшем объеме. Это особенно верно, когда данные хранятся в виде массивов, а не объектов. Формат JSON имеет широкую поддержку благодаря наличию библиотек кодирования/декодирования для большинства языков создания серверных сценариев. Обработка его на стороне клиента является тривиальной задачей, что позволяет потратить больше времени на разработку программного кода, который выполняет некоторые операции с этими данными. И самое важное для веб-разработчиков – он является одним из самых быстрых форматов благодаря небольшому объему данных, которые требуется передавать по сети, а также благодаря высокой скорости их анализа. Формат JSON является краеугольным камнем высокопроизводительных приложений на основе технологии Ajax, особенно при использовании приема на основе динамических тегов `<script>`.

HTML

Часто запрашиваемые данные преобразуются сценарием в разметку HTML для отображения в странице. Преобразование больших объемов данных в простую разметку HTML можно относительно быстро выполнить на языке JavaScript, но то же самое намного быстрее можно сделать

на стороне сервера. Один из приемов, достойных внимания, состоит в том, чтобы сформировать всю необходимую разметку HTML на стороне сервера, а затем передать ее клиенту. После этого JavaScript-сценарий может просто вставить ее на место, воспользовавшись свойством `innerHTML` элемента. Ниже приводится пример списка пользователей, представленный в виде разметки HTML:

```
<ul class="users">
    <li class="user" id="1">
        <a href="http://www.site.com/alice/" class="username">alice</a>
        <span class="realname">Alice Smith</span>
        <a href="mailto:alice@alicesmith.com"
            class="email">alice@alicesmith.com</a>
    </li>
    <li class="user" id="2">
        <a href="http://www.site.com/bob/" class="username">bob</a>
        <span class="realname">Bob Jones</span>
        <a href="mailto:bob@bobjones.com" class="email">bob@bobjones.com</a>
    </li>
    <li class="user" id="3">
        <a href="http://www.site.com/carol/" class="username">carol</a>
        <span class="realname">Carol Williams</span>
        <a href="mailto:carol@carolwilliams.com"
            class="email">carol@carolwilliams.com</a>
    </li>
    <li class="user" id="4">
        <a href="http://www.site.com/dave/" class="username">dave</a>
        <span class="realname">Dave Johnson</span>
        <a href="mailto:dave@davejohnson.com"
            class="email">dave@davejohnson.com</a>
    </li>
</ul>
```

Недостаток такого решения состоит в том, что разметка HTML является слишком расточительным форматом представления данных, даже более расточительным, чем формат XML. Кроме самих данных приходится добавлять вложенные HTML-теги, каждый со своими атрибутами `id`, `class` и другими. Может так получиться, что разметка HTML будет занимать больше места, чем фактические данные, хотя эту ситуацию можно смягчить, используя минимально возможное количество тегов и атрибутов. Исходя из этого данный прием следует использовать, только когда вычислительная мощность клиентов ограничена больше, чем пропускная способность сети.

С одной стороны, имеется самый компактный формат, такой как JSON, позволяющий до минимума уменьшить объем структуры данных, которую требуется анализировать на стороне клиента. При использовании этого формата обеспечивается самое короткое время загрузки на клиентский компьютер, но для обработки и преобразования таких данных в разметку HTML требуется значительный объем процессорного времени

CPU. При этом необходимо выполнить множество строковых операций, которые в языке JavaScript являются одними из самых медленных.

С другой стороны, имеется возможность создать разметку HTML на стороне сервера. Этот формат – намного более объемный, и для его загрузки требуется больше времени, но для отображения данных в этом формате требуется выполнить всего одну операцию:

```
document.getElementById('data-container').innerHTML = req.responseText;
```

В следующей таблице приводятся результаты измерения производительности при использовании формата HTML для представления списка пользователей. Помните о главном отличии этого формата от других: под «анализом» здесь понимается операция вставки разметки HTML в дерево DOM. Кроме того, при использовании разметки HTML обход данных реализуется сложнее и выполняется медленнее, в отличие от обычных JavaScript-массивов.

Формат	Размер	Время загрузки	Время анализа	Общее время
HTML	1063416 байт	273,1 мс	121,4 мс	394,5 мс

Как видно из результатов в таблице, при использовании разметки HTML увеличиваются объем информации, передаваемой по сети, и время, затрачиваемое на синтаксический анализ. Вставка разметки HTML в дерево DOM выполняется одной простой инструкцией, но это обманчивая простота; этой единственной строке программного кода требуется значительное время, чтобы вставить такой большой объем данных в страницу. Эти значения производительности лишь немного будут отличаться от производительности при использовании других форматов, когда конечной целью является не массив данных, а HTML-элементы, отображаемые на странице. Но как бы то ни было, полученные результаты отражают тот факт, что разметка HTML как формат данных слишком расточительна и требует значительного времени для обработки.

Нестандартное форматирование

Идеальным является формат представления данных, включающий минимальный объем информации, необходимой для отделения полей друг от друга. Такой формат легко создать самому, просто объединив данные в строку через символ-разделитель:

```
Jacob;Michael;Joshua;Matthew;Andrew;Christopher;Joseph;Daniel;Nicholas;
Ethan;William;Anthony;Ryan;David;Tyler;John
```

Символы-разделители по сути образуют массив данных, напоминающий список с элементами, разделенными запятыми. Используя различные символы-разделители, можно конструировать многомерные массивы. Ниже приводится пример списка, представленного в формате с символами-разделителями:

```
1:alice:Alice Smith:alice@alicesmith.com;
2:bob:Bob Jones:bob@bobjones.com;
3:carol:Carol Williams:carol@carolwilliams.com;
4:dave:Dave Johnson:dave@davejohnson.com
```

Форматы подобного типа оказываются чрезвычайно компактными и позволяют получить большое значение отношения объема данных к объему структуры (существенно больше, чем позволяют получить другие форматы, исключая простой текст). Данные в собственном формате загружаются быстро и точно так же быстро и легко анализируются на стороне клиента; достаточно просто передать полученную строку методу `split()`, указав символ-разделитель в качестве аргумента. При использовании более сложного форматирования с применением разных символов-разделителей требуется организовать цикл, чтобы разбить исходную строку на отдельные значения (но не забывайте, что циклы в языке JavaScript выполняются очень быстро). Метод `split()` является одной из самых быстрых строковых операций и обычно способен обрабатывать списки из 10000+ элементов за миллисекунды. Ниже приводится пример реализации функции анализа предыдущего формата:

```
function parseCustomFormat(responseText) {

    var users = [];
    var usersEncoded = responseText.split(';');
    var userArray;

    for (var i = 0, len = usersEncoded.length; i < len; i++) {

        userArray = usersEncoded[i].split(':');

        users[i] = {
            id: userArray[0],
            username: userArray[1],
            realname: userArray[2],
            email: userArray[3]
        };
    }

    return users;
}
```

При разработке собственного формата одним из наиболее важных решений является выбор разделителей. В идеале каждый разделитель должен быть представлен единственным символом, и они не должны присутствовать в данных. На эту роль отлично подходят управляющие ASCII-символы, и их легко можно применять в большинстве языков, используемых для создания серверных сценариев. Например, ниже показано, как можно использовать такие ASCII-символы в языке PHP:

```
function build_format_custom($users) {

    $row_delimiter = chr(1);    // \u0001 в JavaScript.
```

```
        $field_delimiter = chr(2); // \u0002 в JavaScript.

        $output = array();
        foreach ($users as $user) {
            $fields = array($user['id'], $user['username'], $user['realname'],
                      $user['email']);
            $output[] = implode($field_delimiter, $fields);
        }

        return implode($row_delimiter, $output);
    }
```

Эти управляющие символы представлены в языке JavaScript в форме записи символов Юникода (например, `\u0001`). Метод `split()` может принимать в аргументе или строку, или регулярное выражение. Если известно, что в данных могут иметься пустые поля, этому методу следует передавать строку-разделитель; если передать ему регулярное выражение, в IE метод `split()` будет игнорировать второй разделитель в каждой паре разделителей, следующих друг за другом. В других броузерах используются эквивалентные типы аргументов.

```
// Роль разделителя играет регулярное выражение.
var rows = req.responseText.split(/\u0001/);

// Строка-разделитель (более надежное решение).
var rows = req.responseText.split("\u0001");
```

Ниже приводятся результаты измерения производительности передачи и обработки данных в нестандартном формате с использованием объекта XHR и динамических тегов `<script>`:

Формат	Размер	Время загрузки	Время анализа	Общее время
Нестандартный формат (XHR)	222892 байт	63,1 мс	14,5 мс	77,6 мс
Нестандартный формат (`<script>`)	222912 байт	66,3 мс	11,7 мс	78,0 мс

С нестандартным форматом можно использовать любой прием получения данных – на основе объекта XHR или на основе динамических тегов `<script>`. Поскольку в обоих случаях ответ представлен строкой, между ними не наблюдается существенной разницы в скорости анализа этой строки. Для очень больших объемов данных этот формат оказывается самым быстрым, превосходящим даже формат JSON и по скорости анализа, и по общему времени загрузки. Применение этого формата позволяет передавать клиенту огромные объемы данных за очень короткое время.

Заключительные выводы о форматах данных

В общем случае предпочтение следует отдавать легковесным форматам; лучшими в этой категории являются JSON и нестандартные форматы на основе символов-разделителей. Если объем данных достаточно велик и время их обработки имеет большое значение, можно применять один из двух следующих приемов:

- Использовать формат JSON-P совместно с приемом динамических тегов `<script>`. В этом случае данные будут интерпретироваться не как строка, а как выполняемый программный код на языке JavaScript, что обеспечит высокую скорость их анализа. Данный прием можно использовать для организации междоменного обмена данными, но его не следует применять для передачи конфиденциальных данных.
- Использовать собственный формат на основе символов-разделителей, получать их с помощью объекта XHR или динамических тегов `<script>` и выполнять преобразование строки с помощью метода `split()`. Данный прием обеспечивает чрезвычайно высокую скорость преобразования данных, даже более высокую, чем прием с использованием формата JSON-P, и в целом позволяет уменьшить объем передаваемой информации.

В следующей таблице и на рис. 7.1 еще раз представлены все результаты измерения производительности (в порядке от более медленных к более быстрым), что поможет вам сравнить производительность, полученную при использовании каждого из форматов. Из результатов был исключен формат HTML, потому что его нельзя напрямую сравнивать с другими форматами. Обратите внимание, что размер и время загрузки приводятся для несжатых данных.

Формат	Размер	Время загрузки	Время анализа	Общее время
Подробный XML	582960 байт	999,4 мс	343,1 мс	1342,5 мс
Подробный JSON-P	487913 байт	598,2 мс	0,0 мс	598,2 мс
Упрощенный XML	437960 байт	475,1 мс	83,1 мс	558,2 мс
Подробный JSON	487895 байт	527,7 мс	26,7 мс	554,4 мс
Упрощенный JSON	392895 байт	498,7 мс	29,0 мс	527,7 мс
Упрощенный JSON-P	392913 байт	454,0 мс	3,1 мс	457,1 мс
Массив JSON	292895 байт	305,6 мс	18,6 мс	324,0 мс
Массив JSON-P	292912 байт	316,0 мс	3,4 мс	319,4 мс
Нестандартный формат (`<script>`)	222912 байт	66,3 мс	11,7 мс	78,0 мс
Нестандартный формат (XHR)	222892 байт	63,1 мс	14,5 мс	77,6 мс

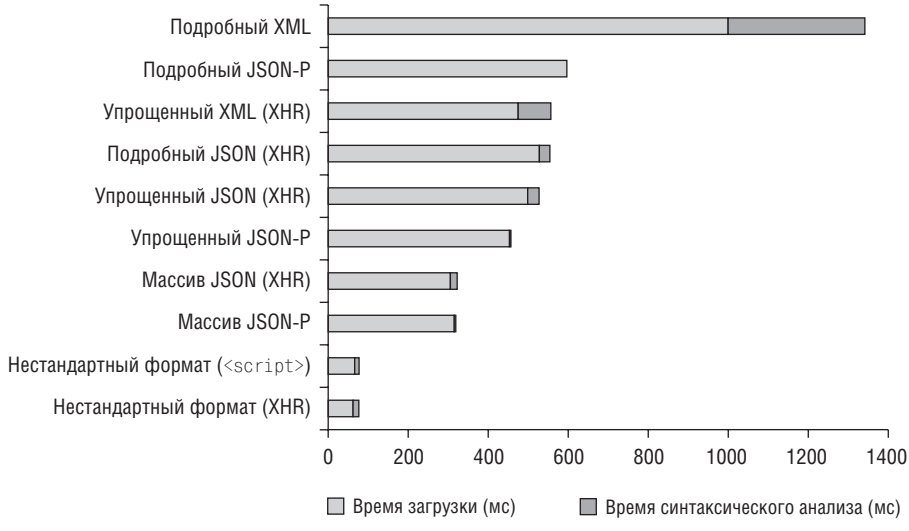


Рис. 7.1. *Сравнение времени загрузки и анализа форматов данных*

Имейте в виду, что эти данные были получены по результатам единственного теста в одном броузере. Эти результаты следует использовать лишь для качественной оценки, а не как фиксированные значения производительности. Вы можете провести собственное тестирование, обратившись по адресу *http://techfoolery.com/formats/*.

Рекомендации по повышению производительности Ajax

Выбрав наиболее подходящие формат и способ передачи данных, можно приступать к рассмотрению других приемов оптимизации. Применимость этих приемов тесно зависит от конкретной ситуации, поэтому перед их использованием следует убедиться, насколько приложение соответствует той или иной ситуации.

Кэширование данных

Самый быстрый Ajax-запрос – тот, который не выполняется. Существует два основных способа предотвратить выполнение ненужных запросов:

- На стороне сервера установить HTTP-заголовки так, чтобы обеспечить сохранение ответа в кэше броузера.
- На стороне клиента сохранять полученные данные локально, чтобы потом не пришлось запрашивать их повторно.

Первый прием является наиболее простым в использовании и сопровождении, тогда как второй обеспечивает более точное управление.

Установка HTTP-заголовков

Если необходимо, чтобы броузер кэшировал Ajax-ответы, запросы должны выполняться методом GET. Но простого использования GET-запросов недостаточно; необходимо также, чтобы в ответе присутствовали нужные HTTP-заголовки. Заголовок `Expires` сообщает броузеру, как долго должен храниться ответ в кэше. Его значением является дата, по истечении которой любые запросы к этому URL-адресу будут направляться не в кэш, а на сервер. Ниже показано, как выглядит заголовок `Expires`:

```
Expires: Mon, 28 Jul 2014 23:30:00 GMT
```

Данный заголовок `Expires` сообщает броузеру, что этот ответ должен храниться в кэше до июля 2014 года. Такие заголовки `Expires` *в далеком будущем* удобно использовать для передачи содержимого, которое никогда не изменится, например изображений или статических данных.

Дата в заголовке `Expires` является датой GMT. На языке PHP такой заголовок можно установить следующим способом:

```
$lifetime = 7 * 24 * 60 * 60; // 7 дней, в секундах.
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

Этот заголовок сообщает броузеру, что он должен хранить данные в кэше 7 дней. Чтобы установить значение заголовка `Expires` в далеком будущем, нужно указать в переменной `$lifetime` более длительный интервал. Следующий пример устанавливает заголовок `Expires`, сообщающий броузеру, что он должен хранить данные 10 лет:

```
$lifetime = 10 * 365 * 24 * 60 * 60; // 10 лет, в секундах.
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

Использование заголовка `Expires` является самым простым способом обеспечить сохранение Ajax-ответов в кэше броузера. При этом не придется вносить какие-либо изменения в программный код клиентского сценария и можно продолжать выполнять Ajax-запросы обычным образом, пребывая в уверенности, что броузер будет отправлять запросы на сервер только в случае отсутствия данных в кэше. Кроме того, поддержку этой оптимизации легко реализовать на стороне сервера, потому что все языки программирования позволяют устанавливать заголовки тем или иным способом. Это самый простой способ обеспечить кэширование данных.

Сохранение данных локально

Вместо того чтобы опираться на механизм кэширования в броузере, можно организовать кэширование данных вручную, сохраняя принимаемые от сервера ответы. Реализовать такое кэширование можно посредством сохранения текста ответа в свойстве объекта, имя которого совпадает с URL-адресом, использовавшимся для его получения. Ниже приводится пример обертки для объекта XHR, которая сначала прове-

ряет наличие в кэше данных для указанного URL-адреса и только потом выполняет запрос:

```
var localCache = {};

function xhrRequest(url, callback) {

    // Проверить наличие в локальном кэше данных для указанного URL.
    if (localCache[url]) {
        callback.success(localCache[url]);
        return;
    }

    // Если данные в кэше отсутствуют, выполнить запрос.

    var req = createXhrObject();
    req.onerror = function() {
        callback.error();
    };

    req.onreadystatechange = function() {
        if (req.readyState == 4) {

            if (req.responseText === '' || req.status == '404') {
                callback.error();
                return;
            }

            // Сохранить ответ в локальном кэше.

            localCache[url] = req.responseText;
            callback.success(req.responseText);
        }
    };

    req.open("GET", url, true);
    req.send(null);
}
```

В общем случае установка заголовка `Expires` является более предпочтительным решением. Оно проще в реализации и позволяет обеспечить кэширование данных между страницами и сеансами. Однако ручное кэширование может пригодиться в ситуациях, когда необходимо программным способом изменять срок хранения данных в кэше и по его истечении получать свежие данные. Представьте ситуацию, когда было бы желательно извлекать данные из кэша для всех запросов, только пока пользователь не выполнит операцию, влекущую за собой изменение данных, вследствие чего ответы в кэше станут недействительными. В подобных ситуациях можно довольно просто организовать удаление таких ответов из кэша:

```
delete localCache['/user/friendlist/'];
delete localCache['/user/contactlist/'];
```

Локальный кэш можно также использовать, когда веб-приложение выполняется на мобильном устройстве. Большинство броузеров для таких устройств имеют очень маленький кэш или вообще не имеют его, поэтому кэширование вручную может оказаться наиболее удачным решением, предотвращающим выполнение лишних запросов.

Известные ограничения библиотек поддержки Ajax

Доступ к объектам поддержки технологии Ajax обеспечивают все JavaScript-библиотеки. Они скрывают различия между броузерами и предоставляют унифицированный прикладной интерфейс. В большинстве ситуаций это огромное благо, поскольку позволяет сконцентрировать внимание не на низкоуровневых особенностях использования объекта XHR в разных броузерах, а на решении прикладных задач. Однако унифицируя интерфейс, библиотеки вынуждены также упрощать его, потому что не все броузеры реализуют все особенности. Это не позволяет использовать объект `XMLHttpRequest` на полную мощность.

Некоторые из приемов, описываемых в этой главе, можно реализовать, только имея непосредственный доступ к объекту XHR. Самым значимым из них является прием потоковой обработки составных данных. Обрабатывая ситуацию, когда свойство `readyState` получает значение 3, можно организовать анализ длинных ответов по частям еще до того, как они будут полностью приняты. Этот прием позволяет обрабатывать фрагменты данных в реальном времени, и такой способ обработки составных данных является одной из причин значительного увеличения производительности. Однако большинство JavaScript-библиотек не дают прямого доступа к событию `readystatechange`, а это означает, что приложение вынуждено ждать окончания приема всего ответа целиком (что может занять существенный промежуток времени), прежде чем приступить к использованию каких-либо его частей.

Непосредственное использование объекта `XMLHttpRequest` не является такой сложной задачей, как может показаться. За исключением небольшого количества особенностей, большинство последних версий всех основных броузеров обеспечивают практически одинаковую поддержку объекта `XMLHttpRequest`, и все предоставляют доступ к различным значениям свойства `readyState`. А поддержку старых версий IE можно реализовать всего несколькими строками программного кода. Ниже приводится пример функции, возвращающей объект XHR, с которым можно взаимодействовать непосредственно (это измененная версия функции, используемой библиотекой YUI 2 Connection Manager):

```
function createXhrObject() {

    var msxml_progid = [
        'MSXML2.XMLHTTP.6.0',
```

```
            'MSXML3.XMLHTTP',
            'Microsoft.XMLHTTP',  // Не поддерживает значение 3 для readyState.
            'MSXML2.XMLHTTP.3.0'  // Не поддерживает значение 3 для readyState.
        ];

        var req;
        try {
            req = new XMLHttpRequest(); // Сначала попробовать стандартный способ.
        }
        catch(e) {
            for (var i = 0, len = msxml_progid.length; i < len; ++i) {
                try {
                    req = new ActiveXObject(msxml_progid[i]);
                    break;
                }
                catch(e2) { }
            }
        }
        finally {
            return req;
        }
    }
```

Эта функция сначала пытается использовать версию объекта `XMLHttpRequest`, поддерживающую значение 3 для свойства `readyState`, а затем переходит к использованию одной из доступных устаревших версий объекта, не поддерживающей эту особенность.

Прямое взаимодействие с объектом XHR также позволяет уменьшить накладные расходы на вызовы функций и еще больше повысить производительность. Но прежде чем отказаться от использования библиотеки поддержки Ajax, следует подумать о последствиях, потому что в этом случае можно столкнуться с проблемами совместимости с устаревшими и малораспространенными броузерами.

В заключение

Для обеспечения высокой производительности Ajax-приложений необходимо знать конкретные требования к конкретной ситуации и правильно выбрать формат данных и способ их передачи.

Форматы данных, такие как простой текст и разметка HTML, очень тесно связаны с конкретной ситуацией, но они могут сэкономить машинные циклы процессора на стороне клиента. Формат XML имеет весьма широкое распространение и поддерживается практически повсеместно, но он слишком расточительный, и на его обработку требуется много времени. Формат JSON более легкий, быстрее обрабатывается (когда интерпретируется не как строка, а как программный код) и имеет почти такое же широкое распространение, как формат XML. Нестандартные форматы, основанные на использовании символов-разделителей,

являются самыми легковесными и позволяют обрабатывать большие объемы данных еще быстрее, но требуют дополнительного программного кода для форматирования данных на стороне сервера и обработки на стороне клиента.

При организации запросов данных объект XHR обеспечивает полный контроль и максимальную гибкость, когда запрашиваемые данные находятся в том же домене, что и сама страница, однако он интерпретирует все входящие данные как строки, что может приводить к снижению производительности. Прием создания динамических тегов `<script>`, напротив, позволяет выполнять междоменные запросы и использовать интерпретатор броузера для обработки программного кода на JavaScript и данных в формате JSON, однако он предоставляет менее надежный интерфейс и не позволяет читать заголовки или коды состояния ответа. Прием получения составных данных с помощью объекта XHR можно использовать для уменьшения числа запросов; он позволяет обрабатывать файлы различных типов в одном ответе, однако не позволяет кэшировать принятые ресурсы. Простым и эффективным способом отправки данных является посылка сигналов с помощью элементов `Image`. Кроме того, можно использовать объект XHR для отправки больших объемов данных в виде POST-запросов.

В дополнение к различным форматам и способам передачи данных существует еще несколько рекомендаций, способных помочь повысить производительность Ajax-приложений:

- Уменьшайте количество запросов, объединяя JavaScript- и CSS-файлы или используя прием запроса составных данных.
- Используйте технологию Ajax для получения второстепенных файлов уже после загрузки страницы, чтобы создать впечатление высокой скорости загрузки.
- Предусматривайте обработку ошибочных ситуаций и обеспечьте решение проблем на стороне сервера.
- Проанализируйте ситуации, когда лучше использовать надежную библиотеку поддержки Ajax, а когда – писать свой низкоуровневый программный код.

Применение технологии Ajax является одной из самых обширных областей повышения производительности сайтов, во-первых, потому что многие сайты широко используют асинхронные запросы и, во-вторых, потому что эта технология может предложить решения проблем, которые не имеют прямого отношения к ней, такие как наличие большого количества загружаемых ресурсов. Творческое применение объекта XHR может превратить вялую и непривлекательную страницу в эффективную и быстро откликающуюся, а раздражающий своей медлительностью сайт – в любимый пользователями.

8

Приемы программирования

В каждом языке программирования имеются свои болевые точки, а также неэффективные шаблоны программирования, которые вырабатываются со временем. Появление их объясняется тем, что многие разработчики переходят в новый язык и начинают расширять его пределы. Начиная с 2005 года, когда появился термин «Ajax», веб-разработчики расширили круг применения языка JavaScript и броузеров значительно больше, чем когда-либо. В результате появились весьма специфические шаблоны программирования, как оптимальные, так и неоптимальные. Эти шаблоны возникают из-за самой роли языка JavaScript в Веб.

Предотвращение повторной интерпретации

Подобно многим языкам сценариев JavaScript позволяет взять строку с программным кодом и выполнить ее в рамках выполняющегося сценария. Это можно реализовать четырьмя стандартными способами: с помощью функции `eval()`, конструктора `Function()`, функций `setTimeout()` и `setInterval()`. Все эти функции позволяют передавать им строку с программным кодом на языке JavaScript и выполнять его. Например:

```
var num1 = 5,
    num2 = 6,

// eval() выполняет строку с программным кодом
result = eval("num1 + num2"),

// Function() выполняет строки с программным кодом
sum = new Function("arg1", "arg2", "return arg1 + arg2");

// setTimeout() выполняет строку с программным кодом
setTimeout("sum = num1 + num2", 100);
```

```
// setInterval() выполняет строку с программным кодом
setInterval("sum = num1 + num2", 100);
```

Всякий раз когда внутри программного кода выполняется другой программный код, на его интерпретацию затрачивается дополнительное время. Сначала интерпретируется и выполняется основной программный код, а затем интерпретируется и выполняется программный код, содержащийся в строке. Повторная интерпретация является довольно дорогостоящей операцией и может занимать больше времени, чем если бы тот же программный код был включен непосредственно в сценарий.

Для сравнения: время доступа к элементу массива отличается в разных броузерах, но разница становится гораздо более драматичной, когда доступ к элементу массива осуществляется с помощью функции `eval()`. Например:

```
// быстрее
var item = array[0];

// медленнее
var item = eval("array[0]");
```

Разница становится еще более значительной, если попробовать выполнить чтение 10000 элементов массива с помощью функции `eval()`. В табл. 8.1 показано время, затрачиваемое на выполнение этой операции.

Таблица 8.1. Время доступа к 10000 элементам массива из обычного программного кода и с помощью функции eval()

Броузер	Обычный программный код (мс)	Функция eval() (мс)
Firefox 3	10,57	822,62
Firefox 3.5	0,72	141,54
Chrome 1	5,7	106,41
Chrome 2	5,17	54,55
Internet Explorer 7	31,25	5086,13
Internet Explorer 8	40,06	420,55
Opera 9.64	2,01	402,82
Opera 10 Beta	10,52	315,16
Safari 3.2	30,37	360,6
Safari 4	22,16	54,47

Такая значительная разница во времени доступа к элементу массива обусловлена необходимостью создания нового экземпляра интерпретатора/компилятора при каждом вызове функции `eval()`. То же самое происходит и при использовании функций `Function()`, `setTimeout()` и `setInterval()`, что автоматически ведет к снижению скорости выполнения.

В большинстве случаев необходимость в функции `eval()` или `Function()` отсутствует, и следует стараться их не использовать, если это возможно. Что касается двух других функций, `setTimeout()` и `setInterval()`, рекомендуется передавать им в первом аргументе не строку, а ссылку на обычную функцию. Например:

```
setTimeout(function(){
    sum = num1 + num2;
}, 100);

setInterval(function(){
    sum = num1 + num2;
}, 100);
```

Предотвращение повторной интерпретации является ключом к достижению максимальной производительности программного кода на языке JavaScript.

Оптимизирующие реализации JavaScript часто кэшируют результаты интерпретации повторяющихся строк с программным кодом в функции `eval()`. При многократном выполнении одной и той же строки с программным кодом можно заметить значительное повышение производительности в Safari 4 и в Chrome.

Использование литералов объектов/массивов

Объекты и массивы в языке JavaScript можно создавать множеством способов, но ни один из них не дает такой же скорости, как создание объектов или массивов с помощью литералов. Типичный способ создания объекта и присваивания значений его свойствам без использования литерала выглядит так:

```
// создать объект
var myObject = new Object();
myObject.name = "Nicholas";
myObject.count = 50;
myObject.flag = true;
myObject.pointer = null;

// создать массив
var myArray = new Array();
myArray[0] = "Nicholas";
myArray[1] = 50;
myArray[2] = true;
myArray[3] = null;
```

С технической точки зрения такой подход вполне допустим, однако литералы интерпретируются намного быстрее. Кроме того, литералы занимают меньше места в программном коде, что уменьшает общий размер файла. Предыдущий фрагмент можно переписать с использованием литералов, как показано ниже:

```
// создать объект
var myObject = {
    name: "Nicholas",
    count: 50,
    flag: true,
    pointer: null
};

// создать массив
var myArray = ["Nicholas", 50, true, null];
```

Результат выполнения этого программного кода будет такой же, что и в предыдущей версии, но он выполняется быстрее практически во всех броузерах (в Firefox 3.5 разница почти отсутствует). Преимущество литералов растет с увеличением числа свойств создаваемых объектов и элементов массивов.

Предотвращение повторного выполнения работы

Один из основных приемов оптимизации производительности в программировании на любом языке – предотвращение выполнения лишней работы. Это понятие подразумевает две вещи: не нужно выполнять работу, в которой нет необходимости, и не нужно повторно выполнять уже произведенную работу. Выполнение ненужной работы обычно легко идентифицируется в ходе реорганизации программного кода. Повторное выполнение ранее произведенной работы идентифицируется обычно сложнее, потому что повторное выполнение одних и тех же действий может производиться в разных местах и по разным причинам.

Пожалуй, самым типичным примером повторного выполнения работы является определение типа броузера. В программах предусматривается большое количество ветвлений программного кода исходя из функциональных особенностей броузера. Рассмотрим в качестве примера добавление и удаление обработчика событий. Типичная реализация, обеспечивающая совместимость с разными броузерами, имеет следующий вид:

```
function addHandler(target, eventType, handler){
    if (target.addEventListener){ // DOM2 Events
        target.addEventListener(eventType, handler, false);
    } else { // IE
        target.attachEvent("on" + eventType, handler);
    }
}

function removeHandler(target, eventType, handler){
    if (target.removeEventListener){ // DOM2 Events
        target.removeEventListener(eventType, handler, false);
    } else { // IE
        target.detachEvent("on" + eventType, handler);
    }
}
```

Функция проверяет поддержку спецификации «DOM Level 2 Events», определяя наличие методов `addEventListener()` и `removeEventListener()`, которые реализованы во всех современных броузерах, кроме Internet Explorer. Если эти методы отсутствуют в объекте `target`, предполагается, что сценарий выполняется в IE и используются методы, характерные для этого броузера.

На первый взгляд эти функции уже оптимизированы. Но проблема потери производительности скрывается в повторении одних и тех же действий при каждом вызове любой из этих функций. Каждый раз, чтобы убедиться в наличии определенного метода, выполняется одна и та же проверка. Если исходить из того, что значениями ссылки `target` могут быть только элементы дерева DOM и пользователь не может как по волшебству поменять броузер во время загрузки страницы, очевидно, что эти действия повторяются. Если наличие метода `addEventListener()` было установлено при первом вызове функции `addHandler()`, то это будет так и при всех остальных вызовах этой функции. Повторное выполнение одной и той же работы при каждом вызове функции является напрасной тратой времени, избежать которой можно двумя способами.

Отложенная загрузка

Первый способ исключить повторное выполнение работы – использовать *отложенную загрузку*. Под отложенной загрузкой понимается отказ от выполнения любой работы, пока не возникнет необходимость в информации, полученной с ее помощью. В предыдущем примере нет необходимости определять, каким способом следует подключать или отключать обработчики событий, пока не будет произведен вызов функции. Ниже приводятся версии предыдущих функций, реализующие шаблон отложенной загрузки:

```
function addHandler(target, eventType, handler){

    // заместить существующую функцию
    if (target.addEventListener){ // DOM2 Events
        addHandler = function(target, eventType, handler){
            target.addEventListener(eventType, handler, false);
        };
    } else { // IE
        addHandler = function(target, eventType, handler){
            target.attachEvent("on" + eventType, handler);
        };
    }

    // вызвать новую функцию
    addHandler(target, eventType, handler);
}

function removeHandler(target, eventType, handler){
```

```
    // заместить существующую функцию
    if (target.removeEventListener){ // DOM2 Events
        removeHandler = function(target, eventType, handler){
            target.addEventListener(eventType, handler, false);
        };
    } else { // IE
        removeHandler = function(target, eventType, handler){
            target.detachEvent("on" + eventType, handler);
        };
    }

    // вызвать новую функцию
    removeHandler(target, eventType, handler);
}
```

Эти две функции реализуют шаблон отложенной загрузки. При первом вызове этих методов определяется соответствующий способ подключения или отключения обработчиков событий. Затем оригинальная функция замещается новой версией, реализующей только необходимые операции. На последнем этапе первого вызова производится вызов новой версии с оригинальными аргументами. Все последующие вызовы функции `addHandler()` или `removeHandler()` уже не будут выполнять проверку, потому что версия с проверкой окажется замещенной новой функцией.

Первый вызов функции, реализующей шаблон отложенной загрузки, всегда занимает больше времени, потому что он должен выполнить проверку и затем вызвать другую функцию для выполнения поставленной задачи. Однако последующие вызовы той же функции будут работать намного быстрее, так как они не содержат логику проверки. Шаблон отложенной загрузки отлично подходит для реализации функций, которые не требуется использовать немедленно.

Предварительная условная загрузка

Альтернативой шаблону отложенной загрузки является шаблон *предварительной условной загрузки*, который выполняет проверку заранее в процессе загрузки сценария, а не в момент вызова функции. В этом шаблоне проверка также выполняется только один раз, но она осуществляется на ранних этапах выполнения. Например:

```
var addHandler = document.body.addEventListener ?
                 function(target, eventType, handler){
                     target.addEventListener(eventType, handler, false);
                 }:
                 function(target, eventType, handler){
                     target.attachEvent("on" + eventType, handler);
                 };

var removeHandler = document.body.removeEventListener ?
                    function(target, eventType, handler){
                        target.removeEventListener(eventType, handler, false);
                    }:
```

```
        function(target, eventType, handler){
            target.detachEvent("on" + eventType, handler);
        };
```

Этот пример проверяет наличие методов `addEventListener()` и `removeEventListener()` и использует эту информацию для присваивания наиболее подходящей функции. Если искомые методы присутствуют, трехместный оператор возвращает функцию, определяемую спецификацией «DOM Level 2 Events», в противном случае возвращает функцию, специфическую для IE. В результате применения этого шаблона все вызовы функций `addHandler()` и `removeHandler()` выполняются одинаково быстро благодаря тому, что проверка выполняется заранее.

Шаблон предварительной условной загрузки гарантирует, что все вызовы функции будут занимать одинаковое время. Но за это приходится платить выполнением проверки в момент загрузки сценария, а не в более позднее время. Предварительная загрузка отлично подходит для функции, которая должна быть готова к использованию немедленно и затем будет использоваться часто на протяжении всего времени жизни страницы.

Использование сильных сторон

Хотя язык JavaScript часто порицают за его медлительность, но некоторые его операции выполняются невероятно быстро. В этом нет ничего удивительного, потому что интерпретаторы JavaScript обычно пишутся на компилирующих языках низкого уровня. Несмотря на то что при медленном выполнении кода легко можно обвинить в медлительности JavaScript, тем не менее сам интерпретатор обычно является самым быстрым компонентом системы, а медленно выполняется именно ваш программный код. В интерпретаторе есть компоненты, которые действуют намного быстрее других и позволяют решать задачи, избегая использования медленных компонентов.

Битовые операторы

Битовые операторы являются одним из самых недооцененных аспектов языка JavaScript. По общему мнению разработчики часто не понимают, как использовать эти операторы, и путают их с логическими эквивалентами. В результате битовые операторы редко применяются в программах на языке JavaScript, несмотря на их преимущества.

В языке JavaScript числа хранятся в 64-битном формате IEEE-754. Однако перед выполнением битовых операций числа преобразуются в 32-битные целые со знаком. После этого каждый оператор действует непосредственно с этим 32-битным представлением. Несмотря на необходимость промежуточного преобразования, битовые операции в языке JavaScript выполняются невероятно быстро по сравнению с математическими и логическими операциями.

Те, кто не знаком с двоичным представлением чисел, могут воспользоваться простой возможностью преобразования числа в строку, содержащую двоичный эквивалент, с помощью метода `toString()`, передав ему число 2. Например:

```
var num1 = 25,
    num2 = 3;

alert(num1.toString(2)); // "11001"
alert(num2.toString(2)); // "11"
```

Обратите внимание на отсутствие ведущих нулей в этом представлении.

В языке JavaScript имеются четыре битовых оператора:

Поразрядное И (AND)

Возвращает число с 1 в каждом бите, где оба исходных числа имеют 1 в соответствующих битах.

Поразрядное ИЛИ (OR)

Возвращает число с 1 в каждом бите, где хотя бы одно из исходных чисел имеет 1 в соответствующем бите.

Поразрядное исключающее ИЛИ (XOR)

Возвращает число с 1 в каждом бите, где только одно из исходных чисел имеет 1 в соответствующем бите.

Поразрядное НЕ (NOT)

Возвращает число с 1 в каждом бите, где исходное число имеет 0 в соответствующем бите, и наоборот.

Ниже показано, как используются эти операторы:

```
// поразрядное И (AND)
var result1 = 25 & 3; //1
alert(result1.toString(2)); // "1"

// поразрядное ИЛИ (OR)
var result2 = 25 | 3; //27
alert(result2.toString(2)); // "11011"

// поразрядное исключающее ИЛИ (XOR)
var result3 = 25 ^ 3; //26
alert(result3.toString(2)); // "11010"

// поразрядное НЕ (NOT)
var result4 = ~25; //-26
alert(result4.toString(2)); // "-11010"
```

Существует пара способов применения битовых операторов в языке JavaScript для увеличения производительности. Первый заключается в использовании битовых операций взамен математических. В практике часто используется прием чередования цветов в строках таблиц путем вычисления результата деления номера строки по модулю 2, например:

```
for (var i=0, len=rows.length; i < len; i++){
    if (i % 2) {
        className = "even";
    } else {
        className = "odd";
    }

    // применить класс
}
```

Результатом деления по модулю 2 является остаток от деления числа на 2. Если взглянуть на 32-битное представление чисел, можно заметить, что четным является любое число, если его первый бит содержит 0, и нечетным – если первый бит содержит 1. Это легко определить с помощью операции поразрядного И (AND) данного числа с числом 1. Если проверяемое число четное, результатом поразрядного И (AND) с 1 будет число 0; если проверяемое число нечетное, результатом поразрядного И (AND) с 1 будет число 1. Это означает, что предыдущий фрагмент можно переписать так:

```
for (var i=0, len=rows.length; i < len; i++){
    if (i & 1) {
        className = "odd";
    } else {
        className = "even";
    }

    // применить класс
}
```

Несмотря на то что программный код изменился совсем немного, версия на основе поразрядного И действует до 50% быстрее первоначальной версии (в зависимости от броузера).

Второй способ заключается в использовании битовых операторов для работы с так называемыми *битовыми масками*. Битовые маски часто используются в программировании, когда имеется несколько логических параметров, которые могут быть установлены одновременно. Идея состоит в том, чтобы использовать каждый бит числа как признак состояния параметра, фактически превратив число в массив логических флагов. Параметры, необходимые для работы с маской, определяются как значения, равные степеням 2. Например:

```
var OPTION_A = 1;
var OPTION_B = 2;
var OPTION_C = 4;
var OPTION_D = 8;
var OPTION_E = 16;
```

Определив набор параметров, с помощью оператора поразрядного ИЛИ (OR) можно создать число, содержащее несколько параметров:

```
var options = OPTION_A | OPTION_C | OPTION_D;
```

Наличие определенного параметра можно проверить с помощью оператора поразрядного И (AND). Оператор вернет значение 0, если проверяемый флаг не установлен, и ненулевое значение в противном случае:

```
// параметр A присутствует в списке?
if (options & OPTION_A){
    // выполнить некоторые операции
}

// параметр B присутствует в списке?
if (options & OPTION_B){
    // выполнить некоторые операции
}
```

Операции с битовыми масками, как в данном примере, выполняются очень быстро, потому что, как упоминалось выше, операция выполняется низкоуровневым машинным кодом. Если в программе имеется несколько параметров, которые требуется хранить все вместе и часто проверять, использование битовых масок поможет поднять общую производительность.

В языке JavaScript также поддерживаются битовые операторы << (сдвиг влево), >> (сдвиг вправо) и >>> (сдвиг вправо с сохранением знака).

Встроенные методы

Как бы вы ни оптимизировали свой программный код, он никогда не будет выполняться быстрее, чем встроенные методы, предоставляемые реализацией JavaScript. Причина проста: все встроенные методы – имеющиеся в броузере до того, как вы напишете хотя бы строчку кода, – написаны на низкоуровневом языке программирования, таком как C++. Это означает, что реализации данных методов скомпилированы в составе броузера в машинный код и потому не имеют таких ограничений, как программный код на языке JavaScript.

Типичной ошибкой неопытных программистов является попытка реализовать сложные математические вычисления с применением операторов языка JavaScript, когда имеется возможность использовать встроенный объект `Math`. Объект `Math` содержит свойства и методы, предназначенные для упрощения математических операций. Он включает несколько математических констант:

Константа	Значение
`Math.E`	Значение числа E, основания натуральных логарифмов
`Math.LN10`	Натуральный логарифм числа 10
`Math.LN2`	Натуральный логарифм числа 2
`Math.LOG2E`	Логарифм по основанию 2 числа E

Константа	Значение
`Math.LOG10E`	Десятичный логарифм числа E
`Math.PI`	Число π
`Math.SQRT1_2`	Корень квадратный из числа $^1/_2$
`Math.SQRT2`	Корень квадратный из числа 2

Каждое из этих значений вычислено заранее, что избавляет от необходимости вычислять их самостоятельно. Существуют также методы для математических вычислений:

Метод	Значение
`Math.abs(`*num*`)`	Абсолютное значение числа *num*
`Math.exp(`*num*`)`	`Math.E`^{num}
`Math.log(`*num*`)`	Натуральный логарифм числа *num*
`Math.pow(`*num*`,`*power*`)`	num^{power}
`Math.sqrt(`*num*`)`	Корень квадратный из числа *num*
`Math.acos(`*x*`)`	Арккосинус числа *x*
`Math.asin(`*x*`)`	Арксинус числа *x*
`Math.atan(`*x*`)`	Арктангенс числа *x*
`Math.atan2(`*y*`,`*x*`)`	Арктангенс числа *y*/*x*
`Math.cos(`*x*`)`	Косинус числа *x*
`Math.sin(`*x*`)`	Синус числа *x*
`Math.tan(`*x*`)`	Тангенс числа *x*

Эти методы действуют быстрее, чем эквивалентный им программный код на языке JavaScript. Всякий раз когда вам потребуется выполнить сложные математические вычисления, взгляните сначала в сторону объекта `Math`.

Другим примером является прикладной интерфейс селекторов, позволяющий извлекать элементы дерева DOM с помощью CSS-селекторов. Методы выполнения CSS-запросов были встроены в JavaScript и получили широкую популярность благодаря JavaScript-библиотеке jQuery. Реализация селекторов в библиотеке jQuery считается самой быстрой, но даже она существенно уступает в скорости встроенным методам. Встроенные методы `querySelector()` и `querySelectorAll()` в среднем справляются со своей задачей в 10 раз быстрее, чем реализация селекторов на языке JavaScript.[1] Для повышения общей производительности многие

[1] Согласно комплекту тестов SlickSpeed по адресу *http://www2.webkit.org/perf/slickspeed/*.

JavaScript-библиотеки в настоящее время перешли на использование встроенных методов, когда они доступны.

Всегда используйте встроенные методы, если они имеются, особенно для математических вычислений и операций с деревом DOM. Чем большая часть работы будет выполняться скомпилированным программным кодом, тем быстрее будет работать ваш программный код.

В Chrome значительная доля встроенной функциональности реализована на языке JavaScript. Поскольку в Chrome для выполнения и встроенного, и вашего программного кода используется синхронный компилятор JavaScript, различия в производительности между ними иногда оказываются минимальными.

В заключение

В языке JavaScript имеются свои уникальные требования к организации программного кода, влияющие на его производительность. С ростом сложности веб-приложений и с увеличением объема программного кода, содержащегося в них, появились шаблоны и антишаблоны программирования. Ниже перечислены некоторые практические рекомендации, которые следует помнить:

- Избегайте повторной интерпретации за счет отказа от использования функции `eval()` и конструктора `Function()`. Кроме того, функциям `setTimeout()` и `setInterval()` желательно передавать не строки, а ссылки на функции.
- Используйте литералы при создании новых объектов и массивов. Таким образом они создаются и инициализируются быстрее, чем при использовании других способов.
- Избегайте повторного выполнения одной и той же работы. Используйте прием отложенной загрузки или предварительной условной загрузки, когда возникает необходимость использовать логику определения типа броузера.
- При выполнении математических операций используйте битовые операторы, которые работают непосредственно с низкоуровневым представлением числа.
- Встроенные методы всегда выполняются быстрее, чем программный код на языке JavaScript. Всегда используйте встроенные методы, когда это возможно.

Как и в случае с другими приемами и подходами, описываемыми в этой книге, самый заметный прирост производительности можно будет заметить, если применить данные рекомендации к часто выполняемому программному коду.

9

Сборка и развертывание высокопроизводительных приложений на JavaScript

Жюльен Лекомте (Julien Lecomte)

Согласно исследованиям, проведенным отделом Exceptional Performance компании Yahoo! в 2007 году, 40–60% пользователей Yahoo! имеют пустой кэш в броузере и примерно 20% всех просмотров страниц выполняется с пустым кэшем (*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*). Кроме того, более свежие исследования, проведенные группой Search компании Yahoo!, которые были подтверждены независимыми исследованиями Стива Содерса (Steve Souders) из Google, показали, что примерно 15% крупных веб-сайтов в США поставляют свое содержимое в несжатом виде.

Эти факты подчеркивают необходимость максимального повышения эффективности доставки веб-приложений на основе JavaScript-сценариев. Часть этой работы выполняется на этапе проектирования и в ходе разработки, однако этап сборки и развертывания, который часто незаслуженно упускается из виду, имеет не менее важное значение. Если не предпринять должных усилий при выполнении этого важного этапа, пострадает производительность приложения независимо от усилий, направленных на оптимизацию производительности.

Цель этой главы – познакомить вас с эффективными способами сборки и развертывания веб-приложений на основе JavaScript. Многие понятия, представленные здесь, иллюстрируются с помощью Apache Ant – инструмента сборки, написанного на языке Java, который быстро превратился в промышленный стандарт сборки приложений для Веб. Ближе

к концу главы в качестве примера будет представлен гибкий инструмент сборки, написанный на языке PHP5.

Apache Ant

Apache Ant (*http://ant.apache.org/*) – это инструмент автоматизации процесса сборки программного обеспечения. Он напоминает утилиту `make`, но реализован на языке Java и для описания процедуры сборки использует XML-файлы, тогда как утилита `make` использует свой формат Makefile. Ant – один из проектов организации Apache Software Foundation (*http://www.apache.org/licenses/*).

Главным преимуществом инструмента Ant перед утилитой `make` и другими инструментами является его переносимость. Инструмент Ant доступен на самых разных платформах, а формат файлов сборки для этого инструмента является платформонезависимым.

Файлы сборки для инструмента Ant пишутся на языке XML и по умолчанию получают имя *build.xml*. Каждый файл сборки содержит описание процедуры сборки единственного проекта и по крайней мере одного задания. Задание может зависеть от других заданий.

Описания заданий состоят из команд – действий, выполняемых автоматически. Ant имеет огромное количество встроенных команд и позволяет при необходимости добавлять дополнительные команды. Кроме того, имеется возможность создавать собственные команды на языке Java для использования в файлах сборки инструмента Ant.

Проект может иметь набор свойств или переменных. Каждое свойство имеет имя и значение. Свойства можно устанавливать в файле сборки с помощью команды `property` или за пределами инструмента Ant. Значение свойства можно извлечь, поместив его имя между `${` и `}`.

Ниже приводится пример файла сборки. При выполнении задания по умолчанию (`dist`) программный код на языке Java, содержащийся в каталоге с исходными текстами, будет скомпилирован и упакован в JAR-архив.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="dist" basedir=".">

  <!-- глобальные свойства для данной сборки -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Создать временную метку -->
    <tstamp/>
    <!-- Создать структуру каталогов сборки для компиляции -->
    <mkdir dir="${build}"/>
  </target>
```

```
  <target name="compile" depends="init" description="compile the source">
    <!-- Скомпилировать java-код из ${src} в ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
          description="generate the distribution">
    <!-- Создать каталог для дистрибутива -->
    <mkdir dir="${dist}/lib"/>
    <!-- Скопировать все из ${build} в файл MyProject-${DSTAMP}.jar -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean" description="clean up">
    <!-- Удалить деревья каталогов ${build} и ${dist} -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>

</project>
```

Для иллюстрации основных понятий в этой главе используется Apache Ant, но существует множество других инструментов для сборки веб-приложений. Из них особого внимания заслуживает инструмент Rake (*http://rake.rubyforge.org/*), получивший популярность в последние годы. Rake – это программа сборки на языке Ruby, по своим возможностям напоминающая утилиту `make`. Наиболее примечательной особенностью этой программы являются файлы сборки Rakefile (версия файлов Makefile, используемых инструментом Rake), для создания которых применяется стандартный синтаксис языка Ruby и которые поэтому являются платформонезависимыми.

Объединение JavaScript-файлов

Согласно исследованиям отдела Exceptional Performance компании Yahoo!, первым и, пожалуй, самым важным шагом по ускорению работы разрабатываемого веб-сайта – особенно применительно к тем, кто посещает его впервые, – является уменьшение количества HTTP-запросов, необходимых для отображения страницы (*http://yuiblog.com/blog/2006/11/28/performance-research-part-1/*). Именно с этого этапа следует начинать оптимизацию, потому что для объединения ресурсов обычно требуется приложить совсем немного усилий, но такое объединение дает самый большой положительный эффект.

Большинство современных веб-сайтов используют в своей работе несколько JavaScript-файлов: обычно небольшую библиотеку, содержащую набор утилит и графических элементов для упрощения разработки интерактивных веб-приложений, совместимых с различными броузерами, и реализацию функциональности веб-сайта, разбитую на несколько логических модулей с целью упростить работу над ними.

Компания CNN (*http://www.cnn.com/*), например, использует библиотеки Prototype и Script.aculo.us. Главная страница сайта этой компании отображает в общей сложности 12 внешних сценариев и более 20 встроенных сценариев. В качестве простейшего шага оптимизации можно было бы объединить некоторые, если не все, сценарии в один внешний JavaScript-файл и тем самым существенно уменьшить количество HTTP-запросов, необходимых для отображения страницы.

Инструмент Apache Ant предоставляет возможность объединить несколько файлов с помощью команды `concat`. Однако важно помнить, что JavaScript-файлы должны объединяться в определенном порядке с учетом имеющихся зависимостей. После выявления всех зависимостей порядок объединения файлов можно определить с помощью элемента `filelist` или комбинации элементов `fileset`. Ниже показано, как выглядит подобное задание в файле сборки Ant:

```
<target name="js.concatenate">
    <concat destfile="${build.dir}/concatenated.js">
        <filelist dir="${src.dir}"
                  files="a.js, b.js"/>
        <fileset dir="${src.dir}"
                 includes="*.js"
                 excludes="a.js, b.js"/>
    </concat>
</target>
```

Данное задание создает файл *concatenated.js* в каталоге сборки, являющийся результатом объединения файлов *a.js* и *b.js* в указанном порядке, за которыми в алфавитном порядке следуют все остальные файлы, имеющиеся в каталоге с исходными текстами.

Обратите внимание: если программный код в каком-либо из исходных файлов (кроме последнего) не завершается точкой с запятой или символом завершения строки, то получившийся объединенный файл может содержать недопустимый программный код. Эту ошибку можно исправить, проинструктировав Ant с помощью атрибута `fixlastline` о необходимости проверять наличие символа перевода строки в конце каждого объединяемого файла:

```
<concat destfile="${build.dir}/concatenated.js" fixlastline="yes">
    ...
</concat>
```

Предварительная обработка JavaScript-файлов

Препроцессор – это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме, пригодной для обработки последующими программами (компилятором). Результат и вид

> обработки зависят от вида препроцессора; так, некоторые препроцессоры могут выполнить только простую текстовую подстановку, другие же по своим возможностям способны сравниться с языками программирования.
>
> – *http://ru.wikipedia.org/wiki/Препроцессор*

Предварительная обработка исходных JavaScript-файлов не повысит производительность веб-приложения, но позволит вам, помимо всего прочего, проанализировать, как выполняется ваш программный код.

Из-за отсутствия препроцессора, специально предназначенного для обработки программного кода на языке JavaScript, следует использовать лексический препроцессор, достаточно гибкий, чтобы позволить настраивать правила лексического анализа, или использовать препроцессор для языка, лексическая грамматика которого достаточно близка к грамматике языка JavaScript. Поскольку синтаксис языка C близок к синтаксису языка JavaScript, можно использовать препроцессор языка C (`cpp`). Ниже показано, как выглядит соответствующее задание в файле сборки Ant:

```
<target name="js.preprocess" depends="js.concatenate">
    <apply executable="cpp" dest="${build.dir}">
        <fileset dir="${build.dir}"
                 includes="concatenated.js"/>
        <arg line="-P -C -DDEBUG"/>
        <srcfile/>
        <targetfile/>
        <mapper type="glob"
                from="concatenated.js"
                to="preprocessed.js"/>
    </apply>
</target>
```

Это задание, которое зависит от задания `js.concatenate`, создает файл *preprocessed.js* в каталоге сборки, являющийся результатом применения препроцессора `cpp` к ранее созданному объединенному файлу. Обратите внимание, что команда `cpp` вызывается со стандартными ключами `-P` (предотвращать создание строк с маркерами) и `-C` (не удалять комментарии). В этом примере также определен макрос `DEBUG`.

Наличие этого задания дает возможность использовать директивы макроопределений (`#define`, `#undef`) и условной компиляции (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`) непосредственно внутри JavaScript-файлов, например для условного включения (или исключения) программного кода для профилирования:

```
#ifdef DEBUG

(new YAHOO.util.YUILoader({
    require: [/profiler'],
    onSuccess: function(o) {
```

```
        YAHOO.tool.Profiler.registerFunction(/foo', window);
    }
})).insert();

#endif
```

Если вы собираетесь использовать многострочные макроопределения, то должны быть уверены, что все строки завершаются символом конца строки (LF) в стиле UNIX. При необходимости для автоматического решения этой задачи можно воспользоваться встроенной командой `fixcrlf` инструмента Ant.

Другим примером, не имеющим прямого отношения к производительности, но демонстрирующим, насколько широкими возможностями обладает предварительная обработка программного кода на языке JavaScript, является использование макроопределений с переменным числом аргументов и директив включения файлов для реализации отладочных проверок. Рассмотрим следующий файл с именем *include.js*:

```
#ifndef _INCLUDE_JS_
#define _INCLUDE_JS_

#ifdef DEBUG
function assert(condition, message) {
    // Обработать ошибку при проверке, отобразив в диалоге alert
    // содержимое message, возможно, включающее трассировку стека.
}
#define ASSERT(x, ...) assert(x, ## __VA_ARGS__)
#else
#define ASSERT(x, ...)
#endif /* DEBUG */

#endif /* _INCLUDE_JS_ */
```

Теперь можно написать такой программный код на языке JavaScript:

```
#include "include.js"

function myFunction(arg) {
    ASSERT(YAHOO.lang.isString(argvar), "arg should be a string");
    ...
#ifdef DEBUG
    YAHOO.log("Log this in debug mode only");
#endif
    ...
}
```

Дополнительный программный код, выполняющий проверки и регистрацию ошибок, будет включаться в сценарии только при наличии макроопределения `DEBUG`, то есть во время разработки. Эти инструкции будут исключены из сценариев в процессе сборки приложения для промышленной эксплуатации.

Минификация JavaScript-файлов

Минификация JavaScript-файлов – это процесс, в ходе которого из JavaScript-файлов удаляется все, что не требуется для его выполнения. Сюда входят комментарии и лишние пробельные символы. Обычно в результате такой обработки размер файла уменьшается вдвое, что приводит к ускорению загрузки. И наличие такой возможности позволяет программистам вставлять в программы более подробные встроенные комментарии с описанием.

Долгое время стандартным средством минификации JavaScript-файлов оставался инструмент JSMin (*http://www.crockford.com/javascript/jsmin.html*), разработанный Дугласом Крокфордом (Douglas Crockford). Однако с ростом размера и сложности веб-приложений многие осознали, что пришло время поднять минификацию JavaScript на новый уровень. Это основная причина, ставшая поводом к разработке инструмента YUI Compressor (*http://developer.yahoo.com/yui/compressor/*), выполняющего все виды операций для достижения более высокой степени компактности, чем могут предложить другие инструменты. Кроме удаления комментариев и ненужных пробельных символов YUI Compressor предлагает следующие возможности:

- Замену имен локальных переменных более короткими (одно-, двух- или трехсимвольными) именами, оптимизированными для gzip-сжатия.
- Замену скобочной нотации точечной нотацией везде, где это возможно (например, замену `foo["bar"]` на `foo.bar`).
- Замену в литералах имен свойств, заключенных в кавычки, на обычные имена свойств, если это возможно (например, замену `{"foo":"bar"}` на `{foo:"bar"}`).
- Замену экранированных кавычек в строках (например, замену `'aaa\'bbb'` на `"aaa'bbb"`).
- Свертку констант (например, замену `"foo"+"bar"` на `"foobar"`).

Обработка JavaScript-файла инструментом YUI Compressor позволяет уменьшить его размер намного больше, чем с помощью JSMin, без каких-либо дополнительных действий. Взгляните на размеры основных файлов библиотеки YUI (версия 2.7.0 доступна по адресу *http://developer.yahoo.com/yui/*):

```
Raw yahoo.js, dom.js и event.js                  192164 байт
yahoo.js, dom.js и event.js + JSMin               47316 байт
yahoo.js, dom.js и event.js + YUI Compressor      35896 байт
```

В этом примере размер файла, обработанного инструментом YUI Compressor, получился на 24% меньше, чем размер файла, обработанного инструментом JSMin. Однако существуют приемы, позволяющие уменьшить размер файлов еще больше. Сохранение локальных ссылок на объекты/значения, образование замыканий, использование констант

для часто используемых значений, а также отказ от использования функции `eval()` (и родственных ей конструктора `Function()` и функций `setTimeout()` и `setInterval()` со строковыми литералами в первом аргументе), ключевого слова `with` и условных комментариев в языке JScript, – все это способствует уменьшению размера файлов. Рассмотрим следующую функцию, предназначенную для переключения класса `selected` в указанном элементе DOM (220 байт):

```
function toggle (element) {
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
    }
}
```

YUI Compressor преобразует этот программный код, как показано ниже (147 байт):

```
function toggle(a){if(YAHOO.util.Dom.hasClass(a,"selected")){
YAHOO.util.Dom.removeClass(a,"selected")}else{YAHOO.util.Dom.
addClass(a,"selected")}};
```

Если реорганизовать первоначальную версию, сохранив ссылку на `YAHOO.util.Dom` в локальной переменной и использовав константу для хранения значения `"selected"`, будет получена следующая функция (232 байт):

```
function toggle (element) {
    var YUD = YAHOO.util.Dom, className = "selected";
    if (YUD.hasClass(element, className)){
        YUD.removeClass(element, className);
    } else {
        YUD.addClass(element, className);
    }
}
```

Эта версия будет сжата инструментом YUI Compressor еще больше (115 байт):

```
function toggle(a){var c=YAHOO.util.Dom,b="selected";if(c.hasClass(a,b)){
c.removeClass(a,b)}else{c.addClass(a,b)}};
```

Степень минификации увеличилась с 33% до 48%, что является превосходным результатом, достигнутым за счет совсем незначительных усилий. Однако следует отметить, что gzip-сжатие, применяемое при отправке содержимого клиенту, может приводить к противоречивым результатам; иными словами, минифицированный файл с наименьшими размерами не всегда будет давать наименьший файл после gzip-сжатия. Этот странный результат является прямым следствием устранения избыточности, присутствующей в первоначальном файле. Кроме того, такая микрооптимизация несколько снижает скорость выполнения из-за того, что вместо литералов используются переменные, на поиск которых будет расходоваться дополнительное время. Поэтому

я обычно рекомендую не злоупотреблять этими приемами, хотя они могут пригодиться в случаях, когда доставка содержимого выполняется клиентам, не поддерживающим gzip-сжатие (или заявляющим об отсутствии его поддержки).

В ноябре 2009 года компания Google выпустила еще более мощный инструмент минификации под названием Closure Compiler (*http://code.google.com/closure/compiler/*). Этот новый инструмент обеспечивает еще более впечатляющие результаты, чем YUI Compressor, при использовании параметров расширенной оптимизации. В этом режиме Closure Compiler становится весьма агрессивным в выборе способов преобразования программного кода и переименования переменных. Несмотря на невероятную экономию, применение этого инструмента требует особой осторожности и необходимости проверять отсутствие изменений в работе полученного программного кода. Кроме того, он затрудняет отладку, потому что изменяет практически все имена. Библиотека Closure поставляется вместе с расширением Closure Inspector для Firebug (*http://code.google.com/closure/compiler/docs/inspector.html*), которое обеспечивает возможность отображения между краткими и первоначальными именами. Однако это расширение доступно только в броузерах Firefox, что может еще больше осложнять отладку веток программного кода, предназначенного для выполнения в других броузерах, и сама отладка все равно оказывается сложнее, чем при использовании других, менее агрессивных инструментов минификации.

Сборка в виде отдельного этапа или во время выполнения

Объединение файлов, предварительная обработка и минификация – все эти шаги могут выполняться как на этапе собственно сборки приложения, так и на этапе выполнения. Сборка на этапе выполнения – полезная возможность в процессе разработки, но такой подход не рекомендуется использовать после передачи в эксплуатацию по причине низкой масштабируемости. Общее правило при сборке высокопроизводительных приложений гласит: все, что можно сделать на этапе собственно сборки, не должно делаться на этапе выполнения.

Инструмент Apache Ant по своей сути является автономной программой сборки, тогда как гибкий инструмент сборки, представленный в конце этой главы, занимает промежуточное положение и может использоваться как в ходе разработки, так и для создания окончательных комплектов файлов для промышленной эксплуатации.

Сжатие JavaScript-сценариев

Когда веб-броузер запрашивает некоторый ресурс, он обычно отправляет HTTP-заголовок `Accept-Encoding` (начиная с версии HTTP/1.1), чтобы

сообщить серверу поддерживаемые кодировки. Эта информация в первую очередь используется с целью обеспечить возможность сжатия документа и тем самым ускорить загрузку и создать у пользователя более благоприятные впечатления. В число возможных значений заголовка `Accept-Encoding` входят: `gzip`, `compress`, `deflate` и `identity` (эти значения утверждены агентством по выделению имен и уникальных параметров протоколов Интернета (Internet Assigned Numbers Authority, IANA)).

Если веб-сервер обнаружит такой заголовок в запросе, он выберет наиболее подходящий метод кодирования и известит веб-броузер о нем с помощью HTTP-заголовка `Content-Encoding`.

`gzip` является самым популярным способом кодирования. Он позволяет уменьшить размер загружаемых данных на 70%, что делает его действенным оружием в борьбе за производительность веб-приложений. Обратите внимание, что gzip-сжатие должно в первую очередь использоваться для текстовых ответов, включая JavaScript-файлы. Файлы других типов, такие как изображения или PDF-файлы, не должны сжиматься описываемым способом, потому что они уже сжаты, и попытка сжать их еще больше просто приведет к напрасной трате вычислительных ресурсов сервера.

Тем, кто использует веб-сервер Apache (безусловно, самый популярный), для поддержки gzip-сжатия необходимо установить и настроить либо модуль `mod_gzip` (для Apache 1.3, доступный по адресу *http://www.schroepl.net/projekte/mod_gzip/*), либо модуль *mod_deflate* (для Apache 2).

Недавние исследования, проведенные независимо группой Search компании Yahoo! и компанией Google, показали, что примерно 15% содержимого, поставляемого крупными веб-сайтами в США, доставляется клиентам в несжатом виде. В основном это обусловлено отсутствием HTTP-заголовка `Accept-Encoding` в запросах, который удаляется корпоративными прокси-серверами, брандмауэрами и даже антивирусным программным обеспечением на персональных компьютерах. Хотя gzip-сжатие является отличным инструментом для веб-разработчиков, требуется учитывать описанный выше факт и следует стараться писать как можно более краткий программный код. Другой прием заключается в использовании альтернативных JavaScript-сценариев для пользователей, которые не могут воспользоваться преимущества gzip-сжатия, но могли бы применять облегченные версии приложений (при этом пользователь должен иметь возможность вернуться к полноценной версии).

В свете вышесказанного стоит упомянуть инструмент Packer (*http://dean.edwards.name/packer/*) – минификатор JavaScript-файлов, разработанный Дином Эдвардсом (Dean Edwards). Packer способен уменьшать размеры JavaScript-файлов еще больше, чем YUI Compressor. Взгляните на следующие результаты, полученные для библиотеки jQuery (версия 1.3.2, доступная по адресу *http://www.jquery.com/*):

```
jQuery                          120180 байт
jQuery + YUI Compressor          56814 байт
```

```
jQuery + Packer                    39351 байт
jQuery без минификации + gzip      34987 байт
jQuery + YUI Compressor + gzip     19457 байт
jQuery + Packer + gzip             19228 байт
```

После gzip-сжатия библиотеки jQuery, обработанной минификатором Packer или YUI Compressor, получаются файлы почти одинакового размера. Однако обработка файлов, минифицированных с помощью Packer, занимает фиксированное время (от 200 до 300 мс на моем современном ноутбуке). То есть комбинация YUI Compressor с gzip-сжатием всегда дает лучшие результаты. Тем не менее Packer с успехом может использоваться для обслуживания пользователей с медленным соединением, не поддерживающих gzip-сжатие, для которых время на распаковывание оказывается незначительным по сравнению со временем загрузки больших объемов программного кода. Единственным недостатком поддержания различных версий JavaScript-сценариев для различных пользователей является необходимость увеличения затрат на контроль качества.

Кэширование JavaScript–файлов

Кэширование HTTP-компонентов существенно повышает скорость загрузки страниц при повторном посещении веб-сайта. Например, при загрузке главной страницы веб-сайта Yahoo! (*http://www.yahoo.com/*) с заполненным кэшем выполняется на 90% меньше HTTP-запросов и загружается на 83% меньше байтов данных, чем в случае с пустым кэшем. Время выполнения запроса (от момента отправки запроса на получение страницы до момента вызова обработчика события `onload`) составляет от 2,4 до 0,9 секунды (*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*). Обычно кэширование используется для сохранения изображений, однако с таким же успехом кэширование можно применять для сохранения всех статических компонентов, включая JavaScript-файлы.

Чтобы известить клиента о том, как долго он может хранить тот или иной ресурс в кэше, веб-серверы используют в ответе HTTP-заголовок `Expires`. Этот заголовок содержит отметку времени в формате RFC 1123. Например: `Expires: Thu, 01 Dec 1994 16:00:00 GMT`. Чтобы пометить ответ как «хранящийся вечно», веб-сервер отправляет в заголовке `Expires` дату, отстоящую от текущего момента примерно на один год в будущем. Согласно HTTP 1.1 RFC (RFC 2616, раздел 14.21), веб-серверы никогда не должны указывать в заголовке `Expires` дату, отстоящую от текущей более чем на один год в будущем.

Если используется веб-сервер Apache, настроить срок хранения содержимого на стороне клиента относительно текущей даты можно с помощью директивы `ExpiresDefault`. В следующем примере показано применение этой директивы к изображениям, JavaScript-файлам и таблицам стилей CSS:

```
<FilesMatch "\.(jpg|jpeg|png|gif|js|css|htm|html)$">
    ExpiresActive on
```

```
    ExpiresDefault "access plus 1 year"
</FilesMatch>
```

Некоторые броузеры, особенно при выполнении на мобильных устройствах, могут ограничивать кэширование. Например, веб-броузер Safari в iPhone не кэширует компоненты, размер которых превышает 25 Кбайт в несжатом виде (*http://yuiblog.com/blog/2008/02/06/iphone-cacheability/*) или 15 Кбайт в iPhone 3.0 OS. В подобных ситуациях можно попробовать разбить HTTP-компоненты на более мелкие фрагменты и ценой увеличения их количества обеспечить возможность кэширования.

Можно также рассмотреть возможность использования механизмов хранения данных на стороне клиента, если эти механизмы доступны, но в этом случае срок хранения данных должны определять сами JavaScript-сценарии.

Наконец, еще один прием состоит в использовании кэша автономных приложений, определяемого стандартом HTML 5 и реализованного в Firefox 3.5, Safari 4.0 и на платформе iPhone начиная с версии iPhone OS 2.1. Этот прием основан на использовании файла манифеста со списком ресурсов, сохраняемых в кэше. Сам файл манифеста объявляется с помощью атрибута `manifest` в теге `<html>` (обратите внимание на объявление `DOCTYPE`, соответствующее стандарту HTML 5):

```
<!DOCTYPE html>
<html manifest="demo.manifest">
```

Файл манифеста содержит список автономных ресурсов, составленный с использованием специального синтаксиса, и обслуживается как ресурс с MIME-типом `text/cache-manifest`. Более подробную информацию о кэшировании автономных веб-приложений можно найти на веб-сайте консорциума W3C: *http://www.w3.org/TR/html5/offline.html*.

Решение проблем, связанных с кэшированием

Управление кэшем на должном уровне может способствовать появлению благоприятного впечатления у пользователя, но у кэширования есть и оборотная сторона: ускоряя работу своего приложения, необходимо обеспечить доставку пользователям самой последней версии статического содержимого. Это достигается за счет переименования статических ресурсов при их изменении.

Чаще всего разработчики добавляют в имена файлов номер версии или сборки. Другие добавляют контрольную сумму. Лично я предпочитаю использовать время и дату. Эту задачу можно автоматизировать с помощью Ant. Следующее задание выполняет переименование JavaScript-файлов, добавляя дату и время в формате `yyyyMMddhhmm`:

```
<target name="js.copy">
    <!-- Создать временную метку -->
    <tstamp/>
```

```
    <!-- Переименовать JavaScript-файлы, добавляя к ним дату и время -->
    <copy todir="${build.dir}">
        <fileset dir="${src.dir}" includes="*.js"/>
        <globmapper from="*.js" to="*-${DSTAMP}${TSTAMP}.js"/>
    </copy>
</target>
```

Использование сети распространения содержимого

Сеть распространения содержимого (Content Delivery Network, CDN) – это сеть компьютеров, разбросанных географически по всему миру, которые отвечают за доставку содержимого конечным пользователям. Основная цель CDN – обеспечить надежность, масштабируемость и, прежде всего, производительность. Доставляя содержимое с сервера, географически расположенного ближе других к пользователю, сети CDN способны существенно уменьшить сетевые задержки.

Некоторые крупные компании создают и поддерживают собственные сети CDN, но в общем случае экономически выгоднее пользоваться услугами сторонних сетей CDN, таких как Akamai Technologies (*http://www.akamai.com/*) или Limelight Networks (*http://www.limelightnetworks.com/*).

Переход на использование CDN обычно реализуется внесением простых изменений в программный код и может обеспечить существенное уменьшение времени отклика у конечного пользователя.

Стоит отметить, что все наиболее популярные JavaScript-библиотеки доступны через CDN. Например, библиотека YUI обслуживается с помощью собственной сети компании Yahoo! (имя сервера: *yui.yahooapis.com*, дополнительные подробности можно найти по адресу: *http://developer.yahoo.com/yui/articles/hosting/*), а библиотеки jQuery, Dojo, Prototype, Script.aculo.us, MooTools, YUI и другие обслуживаются сетью CDN компании Google (имя сервера: *ajax.googleapis.com*, дополнительные подробности можно найти по адресу *http://code.google.com/apis/ajaxlibs/*).

Развертывание JavaScript-ресурсов

Развертывание JavaScript-ресурсов обычно заключается в копировании файлов на один или несколько удаленных компьютеров и иногда в выполнении нескольких команд оболочки на этих компьютерах. Это особенно типично при распространении вновь добавленных файлов по сети доставки с помощью средств самой сети CDN.

Apache Ant предоставляет некоторые возможности для копирования файлов на удаленные серверы. Для копирования файлов на локальные файловые системы можно использовать команду `copy` или воспользоваться дополнительными командами `FTP` или `SCP`. Лично я предпочитаю

напрямую использовать утилиту scp, которая доступна на всех основных платформах. Ниже приводится очень простой пример использования этой утилиты:

```
<apply executable="scp" failonerror="true" parallel="true">
    <fileset dir="${build.dir}" includes="*.js"/>
    <srcfile/>
    <arg line="${live.server}:/var/www/html/"/>
</apply>
```

Наконец, чтобы выполнить команды оболочки на удаленном сервере, на котором выполняется демон SSH, можно воспользоваться дополнительной командой SSHEXEC или просто вызвать утилиту ssh непосредственно, как показано в следующем примере, который реализует перезапуск веб-сервера Apache в UNIX:

```
<exec executable="ssh" failonerror="true">
    <arg line="${live.server}"/>
    <arg line="sudo service httpd restart"/>
</exec>
```

Гибкий процесс сборки JavaScript-файлов

Традиционные инструменты сборки прекрасно справляются со своей задачей, но большинство веб-разработчиков считают их чересчур громоздкими из-за необходимости вручную компилировать проект после каждого изменения программного кода. Вместо этого вполне достаточно было бы обновить окно броузера и полностью пропустить этап компиляции. Как следствие, в некоторых случаях использование приемов, представленных в этой главе, приводит к созданию неважно работающих приложений или веб-сайтов. К счастью, совсем несложно написать инструмент, объединяющий все эти приемы и повышающий эффективность труда веб-разработчиков и вместе с тем все же обеспечивающий высокую производительность их приложений.

smasher – это приложение, написанное на языке PHP5 и опирающееся на инструмент, используемый группой Search компании Yahoo!. Оно объединяет JavaScript-файлы, обрабатывает их и при необходимости минимизирует их содержимое. Приложение можно запускать из командной строки или использовать его в ходе разработки для автоматического объединения ресурсов «на лету» при обработке веб-запросов. Исходные тексты приложения, находящиеся по адресу *http://github.com/jlecomte/smasher*, содержатся в следующих файлах:

smasher.php

Основной файл.

smasher.xml

Файл с настройками.

`smasher`

Обертка для выполнения в командной строке.

`smasher_web.php`

Точка входа веб-сервера.

Приложение `smasher` требует наличие XML-файла с настройками, содержащего определения групп файлов, подлежащих объединению, а также некоторую дополнительную информацию о системе. Ниже приводится пример такого файла:

```
<?xml version="1.0" encoding="utf-8"?>
<smasher>
    <temp_dir>/tmp/</temp_dir>
    <root_dir>/home/jlecomte/smasher/files/</root_dir>
    <java_bin>/usr/bin/java</java_bin>
    <yuicompressor>
        /home/jlecomte/smasher/yuicompressor-2-4-2.jar</yuicompressor>

    <group id="yui-core">
        <file type="css" src="reset.css" />
        <file type="css" src="fonts.css" />
        <file type="js" src="yahoo.js" />
        <file type="js" src="dom.js" />
        <file type="js" src="event.js" />
    </group>

    <group id="another-group">
        <file type="js" src="foo.js" />
        <file type="js" src="bar.js" />
        <macro name="DEBUG" value="1" />
    </group>

    ...

</smasher>
```

Каждый элемент `group` содержит несколько JavaScript- и/или CSS-файлов. Элемент `root_dir` определяет путь к каталогу, где находятся все эти файлы. При необходимости элементы `group` могут также содержать списки макроопределений для предварительной обработки.

После сохранения файла с настройками приложение `smasher` можно запустить из командной строки. Если запустить эту программу без обязательных параметров, перед завершением она выведет информацию о порядке использования. Следующий пример демонстрирует, как можно объединить, обработать и минифицировать основные файлы JavaScript-библиотеки YUI:

```
$ ./smasher -c smasher.xml -g yui-core -t js
```

Если не случится ничего непредвиденного, в рабочем каталоге появится выходной файл, имя которого совпадает с именем группы (в данном

примере: `yui-core`), за которым следует время и дата, а также соответствующее расширение файла (например, *yui-core-200907191539.js*).

Аналогичным образом можно использовать программу `smasher` для обработки веб-запросов в ходе разработки, для чего следует поместить файл *smasher_web.php* в корневой каталог документов веб-сервера и использовать примерно такой URL-адрес:

```
http://<host>/smasher_web.php?conf=smasher.xml&group=yui-core&type=css&nominify
```

При использовании различных URL-адресов комплектов JavaScript- и CSS-файлов для разработки и промышленной эксплуатации возможно обеспечить эффективную работу и при этом получать максимальную выгоду от использования процедуры сборки.

В заключение

Процедура сборки и развертывания может оказывать огромное положительное влияние на производительность приложений, написанных на языке JavaScript. Наиболее важными шагами в этой процедуре являются:

- Объединение JavaScript-файлов с целью уменьшить количество HTTP-запросов.
- Минификация JavaScript-файлов с помощью YUI Compressor.
- Отправка JavaScript-файлов клиентам в сжатом виде (с использованием gzip-сжатия).
- Обеспечение кэширования JavaScript-файлов путем установки соответствующих HTTP-заголовков в ответах и решение проблем кэширования добавлением даты и времени в имена файлов.
- Использование сети доставки содержимого CDN для обслуживания JavaScript-файлов; CDN не только обеспечит более высокую скорость доставки, она также поможет решить вопросы, связанные с кэшированием и сжатием файлов.

Все эти шаги должны автоматизироваться с помощью общедоступных инструментов сборки, таких как Apache Ant, или с помощью собственных инструментов, наиболее полно отвечающих вашим потребностям. Если вам удастся грамотно реализовать процедуру сборки, вы получите существенный прирост производительности веб-приложений и веб-сайтов, использующих в своей работе огромный объем программного кода на языке JavaScript.

10

Инструменты

Мэтт Суини (Matt Sweeney)

Наличие правильных инструментов существенно для определения узких мест в процессе загрузки и выполнения сценариев. Многие производители броузеров и крупные веб-сайты совместно разрабатывают приемы и инструменты, которые помогут вам в создании эффективных и производительных веб-приложений. В этой главе рассказывается о следующих инструментах, имеющихся в свободном доступе:

Инструменты профилирования

Позволяют измерить время выполнения различных функций в ходе выполнения сценария с целью выявления областей для оптимизации.

Инструменты анализа сетевого трафика

Позволяют исследовать, как выполняется загрузка изображений, таблиц стилей и сценариев, и определить ее влияние на общее время, необходимое для загрузки и отображения страницы.

Когда какой-либо сценарий или приложение выполняется недостаточно оптимально, профилировщик помогает выделить области – первые кандидаты для оптимизации. Это может оказаться совсем не просто из-за различий между поддерживаемыми броузерами, однако многие производители теперь встраивают в броузеры собственные комплекты инструментов для отладки, в том числе и профилировщики. В отдельных случаях проблемы производительности могут проявляться только в определенных броузерах; в других случаях симптомы могут обнаруживаться во многих броузерах. Имейте в виду, что оптимизация в одном броузере может положительно сказываться и в других броузерах, но точно так же она может давать совершенно противоположный эф-

фект. Вместо того чтобы строить догадки о том, какие функции или операции выполняются слишком медленно, профилировщик гарантирует, что оптимизации будут подвергнуты самые медленные области системы, которые затрагивают большинство броузеров.

Хотя большая часть главы посвящена инструментам профилирования, тем не менее высокоэффективными в обеспечении максимально быстрой загрузки и выполнения сценариев и страниц могут оказаться инструменты анализа сетевого трафика. Прежде чем углубиться в оптимизацию программного кода, необходимо убедиться, что все сценарии и другие ресурсы загружаются наиболее оптимальным образом. В соответствии с тем, сколько параллельных запросов позволяет выполнять броузер и сколько ресурсов должно быть загружено, скорость загрузки сценариев может зависеть от загрузки изображений и таблиц стилей.

Некоторые из этих инструментов дают рекомендации по повышению производительности веб-страниц. Тем не менее имейте в виду, что информацию, предоставляемую этими инструментами, удастся интерпретировать наилучшим образом, если вы разбираетесь в принципах, стоящих за правилами. Как известно, большинство правил имеют исключения, и глубокое понимание правил позволит вам определить эти исключения.

Профилирование JavaScript-сценариев

Самым основным инструментом профилирования, который входит в состав любой реализации JavaScript, является сам язык. С помощью объекта `Date` можно произвести измерения в любой точке программы. До появления других инструментов использование этого объекта было самым распространенным способом хронометража сценариев и продолжает оставаться полезной возможностью. По умолчанию объект `Date` возвращает текущее время, а операция вычитания одного экземпляра `Date` из другого возвращает разницу в миллисекундах. Рассмотрим следующий пример, где сравниваются прием создания элемента с нуля и путем копирования существующего элемента (глава 3 «Работа с деревом DOM»):

```
var start = new Date(),
    count = 10000,
    i, element, time;

    for (i = 0; i < count; i++) {
        element = document.createElement('div');
    }

    time = new Date() - start;
    alert('created ' + count + ' in ' + time + 'ms');

    start = new Date();
    for (i = 0, i < count; i++) {
```

```
        element = element.cloneNode(false);
    }

    time = new Date() - start;
    alert('created ' + count + ' in ' + time + 'ms');
```

Такой способ профилирования довольно сложен в реализации, так как требует вручную писать программный код, выполняющий измерения. Более удачным решением является реализация объекта `Timer`, автоматизирующего вычисления со временем и сохраняющего полученные результаты.

```
var Timer = {
    _data: {},

    start: function(key) {
        Timer._data[key] = new Date();
    },

    stop: function(key) {
        var time = Timer._data[key];
        if (time) {
            Timer._data[key] = new Date() - time;
        }
    },

    getTime: function(key) {
        return Timer._data[key];
    }
};

Timer.start('createElement');
for (i = 0; i < count; i++) {
    element = document.createElement('div');
}

Timer.stop('createElement');
alert('created ' + count + ' in ' + Timer.getTime('createElement');
```

Как видно из этого примера, использование объекта `Timer` все еще требует вручную писать программный код, выполняющий измерения, но в нем уже угадывается шаблон для создания настоящего профилировщика на языке JavaScript. Путем расширения концепции объекта `Timer` можно сконструировать профилировщик, в котором будет возможность регистрировать функции и производить их хронометраж.

YUI Profiler

Профилировщик YUI Profiler (*http://developer.yahoo.com/yui/profiler/*), созданный Николасом Закасом (Nicholas Zakas), является профилировщиком программного кода на JavaScript, написанным на языке Java-

Script. Вдобавок к возможности хронометража он предоставляет интерфейсы для профилирования функций, объектов и конструкторов, а также для составления подробных отчетов на основе результатов профилирования. Он позволяет выполнять профилирование в различных броузерах и экспортировать данные для создания расширенных отчетов и дальнейшего анализа.

Профилировщик YUI Profiler содержит универсальный таймер, собирающий данные о производительности, и предоставляет статические методы для запуска и остановки именованных испытаний и извлечения результатов измерений.

```
var count = 10000, i, element;

Y.Profiler.start('createElement');

for (i = 0; i < count; i++) {
    element = document.createElement('div');
}

Y.Profiler.stop('createElement');

alert('created ' + count + ' in ' +
      Y.Profiler.getAverage('createElement') + 'ms');
```

Очевидно, что это более совершенный способ, чем подход на основе объекта `Date` и `Timer`, обеспечивающий возможность получения дополнительных данных о количестве вызовов, а также среднего, наименьшего и наибольшего времени выполнения. После сбора данных их можно проанализировать и сопоставить с результатами профилирования, полученными в другой серии испытаний.

Имеется также возможность регистрировать функции для профилирования. Хронометраж зарегистрированных функций выполняется программным кодом, который собирает информацию о производительности. Например, чтобы выполнить профилирование глобального метода `initUI()` из главы 2, достаточно лишь указать его имя:

```
Y.Profiler.registerFunction("initUI");
```

Многие функции привязаны к объектам с целью предотвращения засорения глобального пространства имен. Методы объектов можно зарегистрировать, если передать объект функции `registerFunction()` во втором аргументе. Например, предположим, что имеется объект с именем `uiTest`, реализующий две версии `initUI()` с именами `uiTest.test1()` и `uiTest.test2()`. Каждый из этих методов можно зарегистрировать отдельно:

```
Y.Profiler.registerFunction("test1", uiTest);
Y.Profiler.registerFunction("test2", uiTest);
```

Такой подход вполне работоспособен, но его сложно распространить на случай профилирования множества функций или всего приложения.

Метод `registerObject()` автоматически регистрирует все методы указанного объекта:

```
Y.Profiler.registerObject("uiTest", uiTest);
```

В первом аргументе ему передается имя объекта (для составления отчетов), а во втором – сам объект. Эта инструкция выполнит профилирование всех методов объекта `uiTest`.

Объекты, опирающиеся на механизм наследования от прототипа, требуют особого подхода. YUI Profiler позволяет зарегистрировать функцию-конструктор, которая будет использована для хронометража всех методов всех экземпляров объекта:

```
Y.Profiler.registerConstructor("MyWidget", myNameSpace);
```

Теперь хронометраж и составление отчетов будет выполняться для каждой функции в каждом экземпляре `myNameSpace.MyWidget`. Отдельные отчеты можно извлекать в виде объектов:

```
var initUIReport = Y.Profiler.getReport("initUI");
```

В результате выполнения этой инструкции будет создан объект, содержащий информацию о профилировании, включающую массив *точек измерения* с результатами хронометража каждого вызова в порядке их следования. По этим точкам можно построить график и проанализировать изменение скорости выполнения во времени. Этот объект имеет следующие поля:

```
{
    min: 100,
    max: 250,
    calls: 5,
    avg: 120,
    points: [100, 200, 250, 110, 100]
};
```

Иногда может потребоваться получить единственное значение из определенного поля. Статические методы объекта `Profiler` позволяют получать дискретные данные для отдельных функций и методов:

```
var uiTest1Report = {
    calls: Y.Profiler.getCalls("uiTest.test1"),
    avg: Y.Profiler.getAvg("uiTest.test1")
};
```

Для корректного анализа производительности сценария необходимо иметь некоторое представление, выделяющее самые медленные фрагменты программного кода. Для этого предусмотрен отчет по всем вызывавшимся зарегистрированным функциям в объекте или конструкторе:

```
var uiTestReport = Y.Profiler.getReport("uiTest");
```

Этот метод возвращает объект со следующими данными:

```
{
    test1: {
        min: 100,
        max: 250,
        calls: 10,
        avg: 120
    },
    test2:
        min: 80,
        max: 210,
        calls: 10,
        avg: 90
    }
};
```

Это дает возможность сортировать и представлять данные наглядными способами, выделяя наиболее медленные участки программы для более детального изучения. Также предусмотрена возможность создания полного отчета по всем текущим данным профилирования. Однако такой отчет может содержать бесполезную информацию, например сведения о функциях, вызывавшихся ноль раз, или о функциях, производительность которых не вызывает беспокойства. Чтобы уменьшить объем такой ненужной информации, методу можно передать дополнительную функцию, выполняющую фильтрацию данных:

```
var fullReport = Y.Profiler.getFullReport(function(data) {
    return (data.calls > 0 && data.avg > 5);
});
```

Логическое значение, возвращаемое этой функцией, служит признаком включения в отчет той или иной функции, что позволяет убрать из отчета ненужную информацию.

По окончании профилирования функции, объекты и конструкторы можно исключить из дальнейшего профилирования по отдельности, что дополнительно приводит к удалению данных профилирования:

```
Y.Profiler.unregisterFunction("initUI");
Y.Profiler.unregisterObject("uiTests");
Y.Profiler.unregisterConstructor("MyWidget");
```

Метод `clear()` сохраняет регистрацию функций в профилировщике, но очищает связанные с ними данные. Этот метод можно вызывать для отдельных функций или испытаний:

```
Y.Profiler.clear("initUI");
```

Можно также очистить все данные, вызвав метод без аргумента:

```
Y.Profiler.clear();
```

Поскольку данные генерируются в формате JSON, отчеты профилировщика можно просматривать самыми разными способами. Самый простой из них – в виде HTML-страницы. Данные можно также отправлять

на сервер, где они могли бы сохраняться в базе данных для составления более подробных отчетов. В частности, это может пригодиться для проведения сравнительного анализа различных вариантов оптимизации в разных броузерах.

Следует отметить, что особое неудобство для профилирования доставляют анонимные функции, так как они не имеют имен, к которым можно было бы привязать полученные данные. Однако профилировщик YUI Profiler имеет механизм, позволяющий выполнять профилирование анонимных функций. Метод регистрации анонимной функции возвращает функцию-обертку, которая может вызываться вместо анонимной функции:

```
var instrumentedFunction =
    Y.Profiler.instrument("anonymous1", function(num1, num2){
        return num1 + num2;
    });
instrumentedFunction(3, 5);
```

Благодаря этому данные по профилированию анонимной функции будут добавлены в массив результатов профилировщика, что позволит извлекать их точно так же, как и данные для других функций:

```
var report = Y.Profiler.getReport("anonymous1");
```

Анонимные функции

В некоторых профилировщиках получаемые данные могут искажаться из-за использования анонимных функций или операций присваивания функций переменным. Так как применение анонимных функций является типичным шаблоном программирования на языке JavaScript, это может осложнить или сделать невозможным хронометраж и анализ их производительности. Чтобы обеспечить профилирование анонимных функций, лучше всего дать им имена. Использование указателей на методы объектов вместо замыканий обеспечит самый широкий охват функций профилировщиком.

Сравните способ использования встроенной функции:

```
myNode.onclick = function() {
    myApp.loadData();
};
```

с вызовом метода:

```
myApp._onClick = function() {
    myApp.loadData();
};
myNode.onclick = myApp._onClick;
```

Использование функции как метода объекта позволит любому из рассматриваемых далее профилировщиков автоматически подхватить про-

филирование обработчика события onclick. Однако это не всегда удобно, так как может потребовать значительной реорганизации программного кода при подготовке к профилированию.

При использовании профилировщиков, которые автоматически различают анонимные функции, добавление встроенного имени сделает отчеты более удобочитаемыми:

```
myNode.onclick = function myNodeClickHandler() {
    myApp.loadData();
};
```

Аналогично присваивание функций переменным может помочь некоторым профилировщикам избавиться от необходимости подбирать имена:

```
var onClick = function myNodeClickHandler() {
    myApp.loadData();
};
```

Теперь анонимные функции оказываются *именованными*, что позволит большинству профилировщиков использовать более значащие имена в результатах. Добавление имен требует совсем немного усилий, и этот процесс можно даже автоматизировать, сделав его частью сборки отладочной версии приложения.

Для отладки и профилирования всегда используйте несжатые версии сценариев. Это обеспечит простую идентификацию функций.

Firebug

В среде разработчиков броузер Firefox пользуется особой популярностью, отчасти благодаря расширению Firebug (доступному по адресу *http://www.getfirebug.com/*), которое первоначально было разработано Джо Хьюиттом (Joe Hewitt), а теперь поддерживается организацией Mozilla Foundation. Этот инструмент способен повысить производительность труда любого веб-разработчика, обеспечивая немыслимую прежде возможность проникновения в программный код.

Расширение Firebug предоставляет консоль для вывода отладочных сообщений, обеспечивает возможность просмотра дерева DOM текущей страницы и информации о стилях, исследования свойств элементов дерева DOM и JavaScript-объектов и многое другое. Оно также включает профилировщик и инструмент анализа сетевого трафика, которые будут находиться в центре внимания в данном разделе. Кроме того, расширение Firebug само легко расширяется, позволяя без труда добавлять новые панели.

Панель профилировщика в консоли

В расширении Firebug профилировщик является частью панели Console (Консоль), как показано на рис. 10.1. Он реализует хронометраж и вывод отчетов о выполнении программного кода JavaScript, имеющегося в странице. Отчет включает информацию о каждой функции, которая вызывалась во время работы профилировщика, и предоставляет весьма точные результаты измерения производительности и ценные сведения о том, что могло стать причиной низкой скорости выполнения сценария.

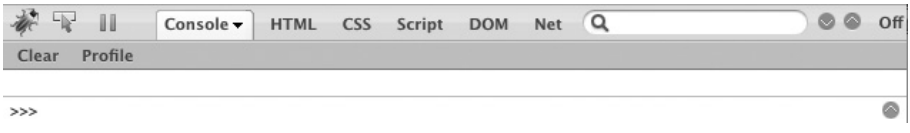


Рис. 10.1. *Панель Console расширения Firebug*

Для запуска профилирования можно щелкнуть на кнопке Profile (Профилировать), запустить сценарий и снова щелкнуть на кнопке Profile (Профилировать), чтобы остановить профилирование. На рис. 10.2 показано, как выглядит типичный отчет с результатами профилирования. Он включает колонки Calls (Вызовы) с количеством вызовов каждой функции, Own Time (Собственное время) со временем, потраченным на выполнение самой функции, и Time (Время) с общим временем, потраченным на выполнение самой функции и любых других функций, которые она вызывала. Профилирование выполняется на уровне интерфейса броузера, поэтому профилирование из панели Console (Консоль) имеет минимальные накладные расходы.

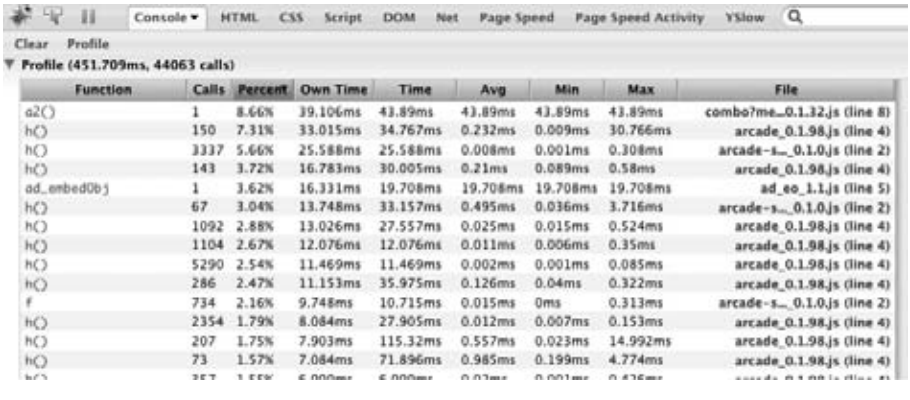


Рис. 10.2. *Панель Profile расширения Firebug*

Прикладной интерфейс консоли

Кроме того, расширение Firebug предоставляет прикладной интерфейс запуска и остановки профилирования из программного кода на JavaScript. Это позволяет более точно организовать хронометраж отдельных фрагментов программного кода. Это также дает возможность присваивать отчетам имена, что может пригодиться для сравнения различных приемов оптимизации.

```
console.profile("regexTest");
regexTest(/foobar', /foo');
console.profileEnd();
console.profile("indexOfTest");
indexOfTest(/foobar', /foo');
console.profileEnd();
```

Возможность запуска и остановки профилирования наиболее интересующих участков программного кода позволяет минимизировать побочные эффекты, обусловленные влиянием других сценариев, которые могут выполняться в процессе работы данного сценария. Однако не следует забывать, что такое использование профилировщика увеличивает накладные расходы. Это объясняется, в первую очередь, затратами времени на создание отчета после каждого вызова метода `profileEnd()`, что вызывает приостановку выполнения сценария до окончания создания отчета. Чем больше отчет, тем больше времени требуется на его создание, поэтому определенный выигрыш можно получить, запуская метод `profileEnd()` с помощью функции `setTimeout()`, чтобы создание отчета выполнялось асинхронно и не блокировало выполнение сценария.

Прикладной интерфейс доступен также через командную строку консоли Firebug.

По окончании профилирования создается новый отчет, показывающий, как долго выполнялась каждая функция, сколько раз они вызывались, процент от общего времени выполнения и другие интересные сведения. Это позволяет понять, куда следует направить усилия по оптимизации, выполнение каких функций следует ускорить и вызовы каких функций желательно минимизировать.

Подобно профилировщику YUI Profiler, функция `console.time()` расширения Firebug может помочь выполнить хронометраж циклов и других операций, которые нельзя отследить с помощью профилировщика. Например, ниже демонстрируется пример хронометража небольшого фрагмента программного кода, содержащего цикл:

```
console.time("cache node");
for (var box = document.getElementById("box"),
        i = 0;
        i < 100; i++) {
```

```
    value = parseFloat(box.style.left) + 10;
    box.style.left = value + "px";
}
console.timeEnd("cache node");
```

По окончании хронометража время выводится в панели Console (Консоль). Это может пригодиться при сравнении различных вариантов оптимизации. В консоль можно вывести дополнительные результаты измерений и тем самым упростить их сопоставление. Например, чтобы сравнить прием кэширования ссылки на узел с приемом кэширования ссылки на свойство `style` узла, достаточно лишь изменить реализацию и вывести новые результаты:

```
console.time("cache style");
for (var style = document.getElementById("box").style,
        i = 0;
        i < 100; i++) {
    value = parseFloat(style.left) + 10;
    style.left = value + "px";
}
console.timeEnd("cache style");
```

Прикладной программный интерфейс консоли дает в руки программистов гибкий инструмент профилирования программного кода на различных уровнях и позволяет объединять результаты в отчеты для дальнейшего анализа различными способами.

Щелчок на имени функции в отчете позволяет перейти к ее реализации в исходном файле, что особенно удобно при работе с анонимными функциями или с функциями, имеющими малопонятные имена.

Панель Net

Сталкиваясь с проблемами производительности, часто бывает полезно отстраниться от программного кода и взглянуть на всю картину в целом. Расширение Firebug позволяет увидеть график загрузки из сети в панели Net (Сеть), как показано на рис. 10.3. В этой панели можно наглядно увидеть паузы между загрузками сценариев и другими ресурсами и получить более полное представление влияния сценариев на загрузку других файлов.

Цветные столбики, следующие за каждым ресурсом, показывают разбиение цикла загрузки на отдельные фазы (поиск в DNS, ожидание ответа и т. д.). Первый столбик (отображаемый синим цветом) отмечает момент, когда было сгенерировано событие `DOMContentLoaded`. Это событие сигнализирует, что дерево DOM страницы полностью сформировано и готово к выполнению операций. Второй столбик (красный) отмечает момент, когда объект `window` сгенерировал событие `load`, которое сигнализирует, что дерево DOM готово к выполнению операций и закончена загрузка всех внешних ресурсов. Это позволяет определить, какая

доля от общего времени отображения страницы была потрачена на синтаксический анализ и выполнение.

На рис. 10.3 показана ситуация загрузки нескольких сценариев. Из графика загрузки видно, что перед запросом на загрузку каждого следующего сценария делается пауза, необходимая для выполнения предыдущего сценария. Самая простая оптимизация, которая позволит уменьшить время загрузки, заключается в уменьшении количества запросов, особенно запросов на загрузку сценариев и таблиц стилей, которые могут блокировать загрузку других ресурсов и увеличивать время, необходимое на отображение страницы. Когда это возможно, следует объединять сценарии в единый файл, чтобы уменьшить общее количество запросов. Это также касается таблиц стилей и изображений.

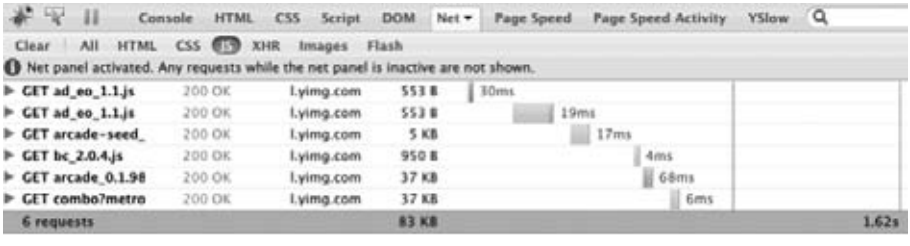


Рис. 10.3. *Панель Net расширения Firebug*

Инструменты разработчика в Internet Explorer

Начиная с версии 8 в состав броузера Internet Explorer входит набор инструментов разработчика, включающий профилировщик. Этот набор инструментов включен непосредственно в IE8, поэтому для его использования не требуется ничего загружать и устанавливать. Подобно расширению Firebug, профилировщик в броузере IE предоставляет функцию профилирования и обеспечивает возможность получения подробных отчетов, включающих количество вызовов, время выполнения и другие характеристики. Он имеет дополнительную возможность просматривать отчет в виде дерева вызовов, профилировать встроенные функции и экспортировать информацию о профилировании. В IE отсутствует инструмент анализа сетевого трафика, однако профилировщик может быть дополнен другими универсальными инструментами, такими как Fiddler, обсуждаемый далее в этой главе. За дополнительной информацией обращайтесь по адресу *http://msdn.microsoft.com/en-us/library/dd565628(VS.85).aspx*.

Профилировщик в IE8 можно найти в инструментах разработчика (Tools (Сервис)→Developer Tools (Средства разработчика)). После щелчка на кнопке Start Profiling (Запуск создания профилей) все последующие операции в программном коде на языке JavaScript будут отслеживаться и хронометр-

рироваться профилировщиком. Щелчок на кнопке Stop Profiling (Остановка создания профилей) (та же кнопка, но с новой надписью) останавливает профилирование и генерирует новый отчет. По умолчанию нажатие на клавишу F5 запускает профилирование, а нажатие комбинации Shift-F5 останавливает его.

Отчет имеет два режима представления: плоское, где выводятся времена выполнения функций, и древовидное, отображающее стек вызовов функций. Древовидное представление позволяет выполнить просмотр стека вызовов и идентифицировать наиболее медленные участки программного кода (рис. 10.4). Профилировщик в IE использует имена переменных, когда имена функций отсутствуют.

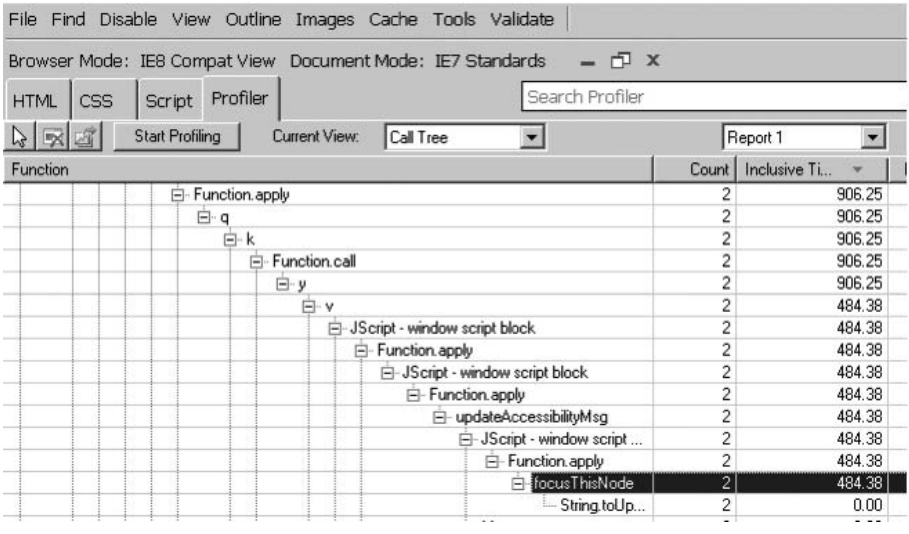


Рис. 10.4. *Дерево вызовов функций в отчете профилировщика IE8*

Щелчок правой кнопкой мыши в области отчета выводит контекстное меню, с помощью которого можно добавлять и удалять столбцы таблицы.

Профилировщик в IE также позволяет выполнять хронометраж методов встроенных JavaScript-объектов. Это дает возможность профилировать встроенные объекты наряду с программным кодом сценария и сравнивать, например, производительность метода `String::indexOf()` с `RegExp::test()` для определения наличия в HTML-элементе свойства `className`, начинающегося с определенного значения:

```
var count = 10000,
    element = document.createElement(/div'),
    result, i, time;
```

```
element.className = /foobar';

for (i = 0; i < count; i++) {
    result = /^foo/.test(element.className);
}

for (i = 0; i < count; i++) {
    result = element.className.search(/^foo/);
}

for (i = 0; i < count; i++) {
    result = (element.className.indexOf(/foo') === 0);
}
```

Из результатов, представленных на рис. 10.5, видно, что методы существенно отличаются по скорости выполнения. Имейте в виду, что среднее время каждого вызова равно нулю. Вообще говоря, встроенные методы – последний адресат для оптимизации, но такая возможность позволяет проводить интересные эксперименты при сравнении различных решений. Также помните, что при маленьких значениях результаты могут получаться неоднозначными из-за ошибок округления и постоянного изменения объема доступной системной памяти.

В настоящее время профилировщик в IE не предлагает прикладной программный интерфейс для доступа из сценариев, но имеет прикладной интерфейс к консоли для вывода информации. Его можно было бы расширить переносом функций `console.time()` и `console.timeEnd()` из Firebug, чтобы обеспечить возможность выполнения тех же испытаний и в IE.

```
if (console && !console.time) {
    console._timers = {};
    console.time = function(name) {
        console._timers[name] = new Date();
    };
    console.timeEnd = function(name) {
        var time = new Date() - console._timers[name];
        console.info(name + ': ' + time + 'ms');
    };
}
```

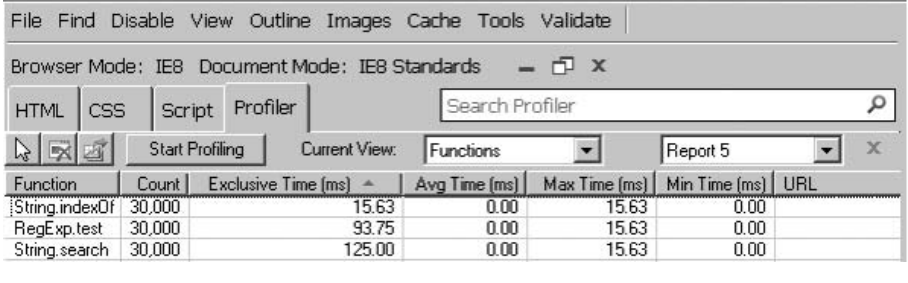


Рис. 10.5. *Результаты профилирования встроенных методов*

Результаты профилирования в IE8 можно экспортировать в формате *.csv*, воспользовавшись кнопкой Export Data (Экспорт данных).

Веб-инспектор в броузере Safari

Начиная с версии 4 в броузере Safari появился свой профилировщик в дополнение к другим инструментам, включающим также инструмент анализа сетевого трафика и входящим в состав веб-инспектора. Подобно инструментам разработчика в Internet Explorer, веб-инспектор позволяет профилировать встроенные функции и получать результаты в виде дерева вызовов функций. Он также предоставляет прикладной программный интерфейс доступа к средствам профилирования консоли, подобно Firebug, и панель Resource (Ресурсы) для анализа сетевого трафика.

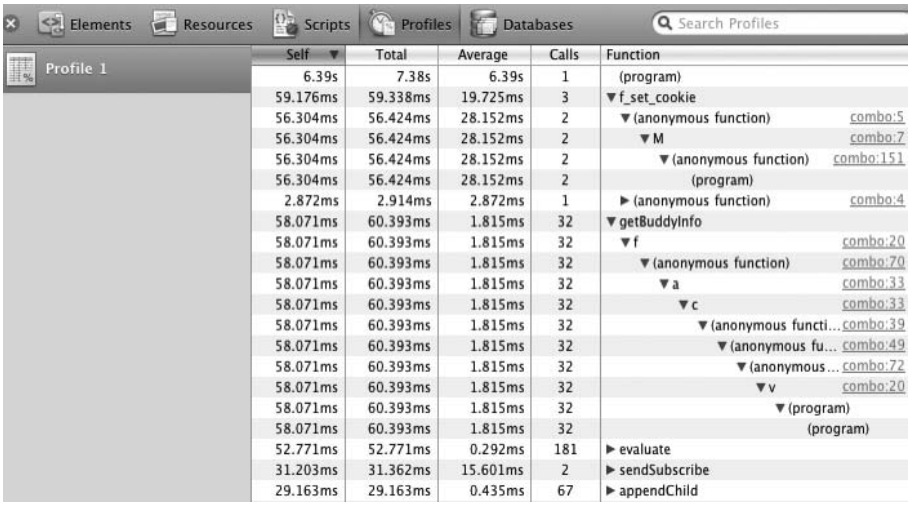
Чтобы получить доступ к веб-инспектору, сначала необходимо сделать доступным меню Develop (Разработка). Чтобы включить меню Develop (Разработка), необходимо открыть диалог Preferences (Настройки)→Advanced (Дополнения) и отметить флажок Show Develop menu in menu bar (Показывать меню «Разработка» в строке меню). После этого можно будет вызвать веб-инспектор выбором пункта меню Develop (Разработка)→Show Web Inspector (Показать веб-инспектор) или нажав комбинацию клавиш Option-Command-I (Mac OS) или Ctrl-Alt-I (Windows).

Панель Profiles

Щелчок на кнопке Profiles (Профили) откроет панель Profiles (Профили), как показано на рис. 10.6. Чтобы активировать панель Profiles (Профили), необходимо щелкнуть на кнопке Enable Profiling (Включить профилирование). Для запуска профилирования необходимо щелкнуть на кнопке Start Profiling (Начать профилирование) (слева внизу, с изображением черного кружка). Чтобы остановить профилирование и отобразить отчет, нужно щелкнуть на кнопке Stop Profiling (Остановить профилирование) (та же самая кнопка, но уже с изображением красного кружка).

Запустить/остановить профилирование можно также комбинацией клавиш Option-Shift-Command-P (Mac OS) или Ctrl-Alt-P (Windows).

Safari имитирует прикладной интерфейс консоли расширения Firebug (`console.profile()`, `console.time()` и другие методы), позволяющий запускать и останавливать профилирование программно. Этот интерфейс обеспечивает выполнение тех же операций, что и расширение Firebug, и дает возможность присваивать отчетам имена для улучшения управления профилями.



Self	Total	Average	Calls	Function
6.39s	7.38s	6.39s	1	(program)
59.176ms	59.338ms	19.725ms	3	▼f_set_cookie
56.304ms	56.424ms	28.152ms	2	▼(anonymous function) combo:5
56.304ms	56.424ms	28.152ms	2	▼M combo:7
56.304ms	56.424ms	28.152ms	2	▼(anonymous function) combo:151
56.304ms	56.424ms	28.152ms	2	(program)
2.872ms	2.914ms	2.872ms	1	▶(anonymous function) combo:4
58.071ms	60.393ms	1.815ms	32	▼getBuddyInfo
58.071ms	60.393ms	1.815ms	32	▼f combo:20
58.071ms	60.393ms	1.815ms	32	▼(anonymous function) combo:70
58.071ms	60.393ms	1.815ms	32	▼a combo:33
58.071ms	60.393ms	1.815ms	32	▼c combo:33
58.071ms	60.393ms	1.815ms	32	▼(anonymous functi... combo:39
58.071ms	60.393ms	1.815ms	32	▼(anonymous fu... combo:49
58.071ms	60.393ms	1.815ms	32	▼(anonymous... combo:72
58.071ms	60.393ms	1.815ms	32	▼v combo:20
58.071ms	60.393ms	1.815ms	32	▼(program)
58.071ms	60.393ms	1.815ms	32	(program)
52.771ms	52.771ms	0.292ms	181	▶evaluate
31.203ms	31.362ms	15.601ms	2	▶sendSubscribe
29.163ms	29.163ms	0.435ms	67	▶appendChild

Рис. 10.6. *Панель Profiles веб-инспектора в Safari*

Имя также можно передать методу `console.profileEnd()`. В этом случае метод остановит профилирование с указанным именем, в случае когда запущено несколько операций профилирования.

Safari обеспечивает два вида представления: Heavy (bottom-up) (По тяжести (снизу-вверх)), для просмотра профилируемых функций, и Tree (top-down) (Дерево (сверху-вниз)), для просмотра стека вызовов. По умолчанию отображается представление Heavy (bottom-up) (По тяжести (снизу-вверх)), отсортированное в порядке, когда самые медленные функции отображаются первыми, и обеспечивающее возможность просмотра стека вызовов. В отличие от Heavy (bottom-up), представление Tree (top-down) (Дерево (сверху-вниз)) позволяет проследить порядок вызова функций сверху вниз. Анализ дерева вызовов может помочь обнаружить скрытые более глубоко проблемы производительности, связанные с тем, как одни функции вызывают другие.

Для нужд профилирования в Safari также была добавлена поддержка свойства `displayName`. Оно позволяет давать имена анонимным функциям для использования в отчетах. Взгляните на следующую функцию, ссылка на которую присваивается переменной `foo`:

```
var foo = function() {
    return /foo!';
};

console.profile(/Anonymous Function');
foo();
console.profileEnd();
```

Как видно на рис. 10.7, отсутствие имен функций может сильно осложнить анализ результатов в отчете. Щелчком на URL-адресе правее функции можно перейти к реализации функции в сценарии.

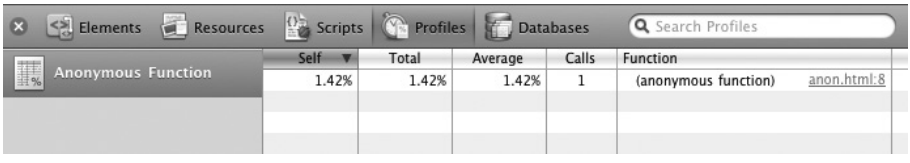


Рис. 10.7. *Отображение анонимных функций в панели Profiles веб-инспектора*

Дополнительное свойство `displayName` позволяет сделать отчеты более удобочитаемыми. Это свойство позволяет также присваивать более описательные имена, не ограничиваясь допустимыми именами функций.

```
var foo = function() {
    return /foo!';
};
foo.displayName = /i am foo';
```

Как видно на рис. 10.8, значение свойства `displayName` заместило имя анонимной функции. Однако это свойство доступно только в броузерах на основе библиотеки Webkit. Кроме того, чтобы использовать это свойство с по-настоящему анонимными функциями, необходимо выполнить реорганизацию программного кода. Как отмечалось выше, добавление встроенных имен является самым простым способом именования анонимных функций, к тому же этот подход с успехом работает и в других профилировщиках:

```
var foo = function foo() {
    return /foo!';
};
```

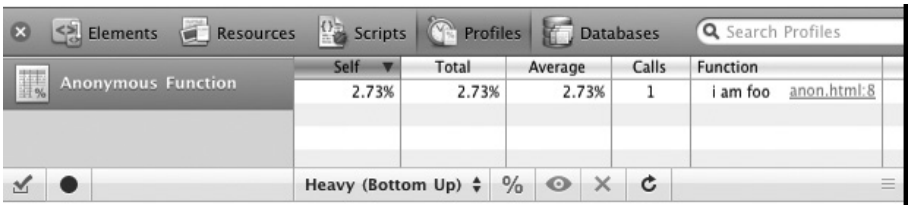


Рис. 10.8. *Отображение значения свойства displayName в панели Profiles веб-инспектора*

Панель Resources

Панель Resources (Ресурсы) помогает лучше понять, как Safari загружает и выполняет сценарии и другие внешние ресурсы. Подобно панели Net (Сеть) в расширении Firebug, она отображает ресурсы и показывает, когда был инициирован запрос и какой отрезок времени заняло его выполнение[1]. Для большей наглядности разные типы ресурсов представлены столбиками разного цвета. Панель Resources (Ресурсы) веб-инспектора масштабирует размеры столбиков, умещая всю протяженность загрузки страницы по ширине окна, уменьшая тем самым визуальные искажения (рис. 10.9).

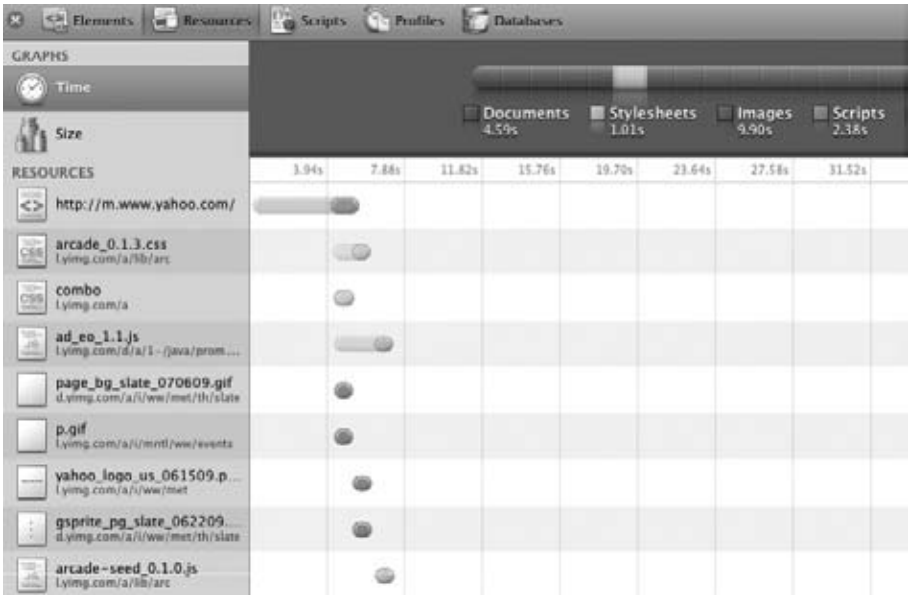


Рис. 10.9. *Панель Resources (Ресурсы) в броузере Safari*

Обратите внимание, что, в отличие от других броузеров, в Safari 4 сценарии загружаются параллельно, не блокируя друг друга. Отсутствие блокировки в Safari объясняется тем, что этот броузер гарантирует выполнение сценариев в соответствующем порядке. Имейте в виду, что это относится только к сценариям, изначально встроенным в разметку HTML; динамически добавляемые сценарии блокируют работу страницы, пока не будут загружены и выполнены (глава 1).

---

[1] В Safari 5 эта информация отображается в новой панели Net (Сеть). – *Прим. перев.*

Инструменты разработчика в Chrome

Компания Google в своем броузере Chrome также предоставляет набор инструментов разработчика, часть которых опирается на функциональные возможности веб-инспектора WebKit/Safari. В дополнение к панели Resources (Ресурсы) для мониторинга сетевого трафика в Chrome ко всем страницам добавляется представление Timeline (График времени) для отображения сетевых событий. В броузере Chrome имеется панель Profiles (Профили) веб-инспектора, в которую добавлена возможность сохранять информацию о распределении памяти в текущий момент. Как и Safari, броузер Chrome позволяет профилировать встроенные функции и поддерживает прикладной программный интерфейс для доступа к консоли, подобный интерфейсу в расширении Firebug, включая методы `console.profile()` и `console.time()`.

Как показано на рис. 10.10, панель Timeline (График времени) обеспечивает возможность просмотра всех операций, подпадающих под одну из категорий: «Loading» (загрузка), «Scripting» (выполнение) или «Rendering» (отображение). Это позволяет разработчику быстро выделить наиболее медленные операции. Некоторые события содержат в себе поддеревья других событий в области представления Records (Записи), которые можно развернуть или свернуть.

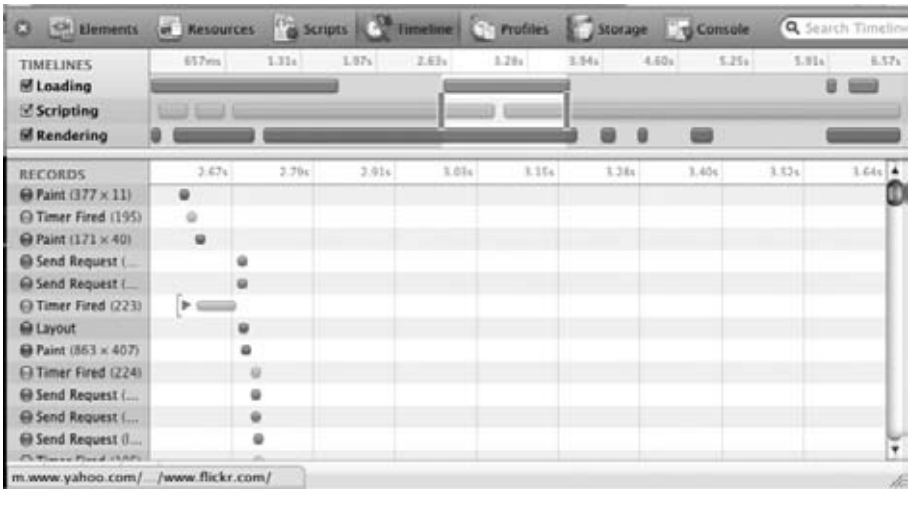


Рис. 10.10. *Панель Timeline из комплекта инструментов разработчика броузера Chrome*

Щелчком на ярлыке с изображением глаза в панели Profiles (Профили) броузера Chrome можно сохранить образ памяти интерпретатора JavaScript (рис. 10.11). Информация в образе группируется по конструкторам, и есть возможность раскрыть любую группу вплоть до отдельных

экземпляров. Образы можно сравнивать, для чего следует выбрать пункт Compared to Snapshot (Сравнить с образом) в раскрывающемся списке в нижней части панели Profiles (Профили). Колонки +/- Count (+/– количество) и Size (Размер) показывают различия между образами.

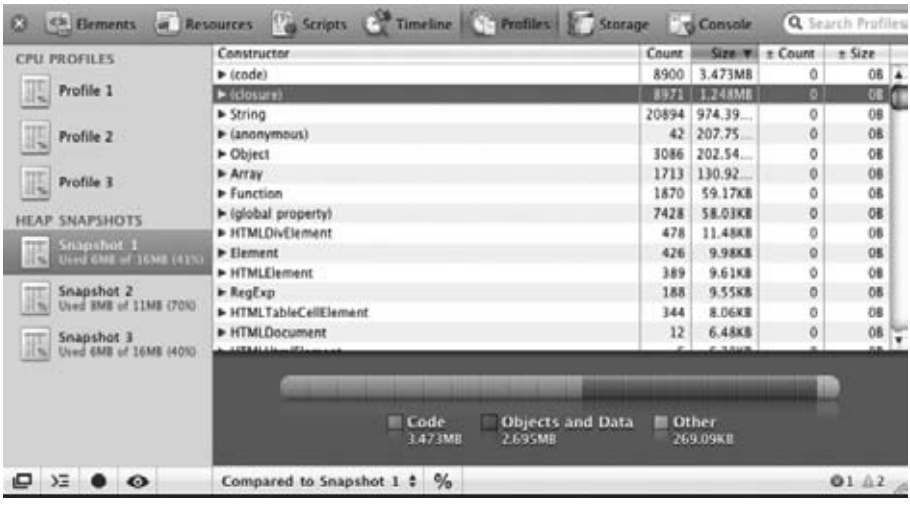


Рис. 10.11. *Создание образов памяти с помощью инструментов разработчика в Chrome*

Блокирование сценариями отображения страницы

Традиционно броузеры выполняют загрузку сценариев по одному. Это обусловлено необходимостью обеспечить удовлетворение зависимостей между файлами. Пока файл, зависящий от других, загружается последним, такой порядок загрузки гарантирует, что все зависимости будут удовлетворены к моменту его запуска. Признаком блокирования отображения страницы сценариями могут служить промежутки между ними. Более новые версии броузеров, такие как Safari 4, IE8, Firefox 3.5 и Chrome, решают эту проблему, производя параллельную загрузку сразу нескольких сценариев, но блокируют их выполнение, чтобы гарантированно удовлетворить возможные зависимости. Такой подход позволяет ускорить загрузку ресурсов, тем не менее процедура отображения страницы все же блокируется до выполнения всех сценариев.

Блокирование отображения страницы сценариями может быть обусловлено медленной процедурой инициализации в одном или нескольких файлах, которую, возможно, следует исследовать с помощью профилировщика и оптимизировать или реорганизовать. Загрузка сценариев может замедлять или приостанавливать отображение страницы, заставляя

пользователя ждать. Выявить и сократить промежутки между ресурсами в ходе их загрузки могут помочь инструменты анализа сетевого трафика. Визуализация этих промежутков поможет определить, какие сценарии выполняются слишком медленно. Загрузку таких сценариев, возможно, стоит отложить на момент, когда страница уже будет отображена, или оптимизировать их, чтобы сократить время выполнения.

Page Speed

Page Speed – это инструмент, первоначально созданный для внутреннего использования в компании Google и позднее выпущенный в виде дополнения к Firebug, которое, подобно панели Net (Сеть) расширения Firebug, отображает информацию о ресурсах, загружаемых веб-страницей. Однако кроме времени загрузки и HTTP-кода состояния он показывает, сколько времени было потрачено на синтаксический анализ и выполнение JavaScript-сценариев, идентифицирует сценарии, загрузку которых можно было бы отложить, и сообщает о неиспользовавшихся функциях. Эта ценная информация может помочь выявить области для дальнейшего исследования, оптимизации и, возможно, реорганизации. Инструкции по установке и другую информацию об этом продукте можно найти по адресу *http://code.google.com/speed/page-speed/*.

Параметр настройки Profile Deferrable JavaScript (Профилирование отложенной загрузки JavaScript), доступный для панели Page Speed, позволяет выявить файлы, загрузку которых можно отложить или которые можно выделить из страницы, чтобы уменьшить объем первоначально загружаемой информации. Часто для первоначального отображения страницы достаточно выполнить очень небольшой сценарий. На рис. 10.12 можно заметить, что основной объем загружаемого программного кода не используется до появления события `load` в объекте `window`. Отложив загрузку сценариев, которые не требуются немедленно для отображения страницы, можно обеспечить более быструю ее загрузку. Позднее сценарии и другие ресурсы можно загружать выборочно по мере необходимости.

Дополнение Page Speed также добавляет в Firebug панель Page Speed Activity (Динамика загрузки страницы). Эта панель напоминает собственную панель Net (Сеть) расширения Firebug, но обеспечивает более подробную информацию о каждом запросе. Эта информация включает подробное разложение жизненного цикла каждого сценария, в том числе фазы синтаксического анализа и выполнения, дающее более точное представление о структуре промежутков между сценариями. Это может помочь в выделении областей, где может потребоваться провести профилирование и реорганизацию программного кода. Как видно из легенды на рис. 10.13, красным цветом показано время, затраченное на синтаксический анализ сценария, а синим – на выполнение. Возможно, причины долгого выполнения сценария заслуживают более подробного изучения с помощью профилировщика.

Рис. 10.12. *Перечень сценариев в Page Speed, загрузку которых можно отложить*

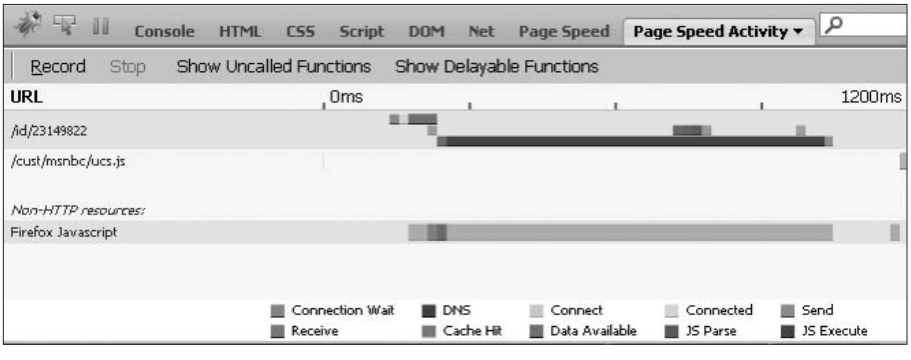


Рис. 10.13. *Время на синтаксический анализ и выполнение в Page Speed*

Может так сложиться, что существенное время будет тратиться на синтаксический анализ и инициализацию сценариев, которые не используются до отображения страницы. Панель Page Speed Activity (Динамика загрузки страницы) может создавать отчеты с перечнем неиспользовавшихся и отложенных функций, исходя из времени, когда они были проанализированы интерпретатором и когда они впервые были вызваны (рис. 10.14).

Эти отчеты показывают, сколько времени было потрачено на инициализацию функций, которые либо ни разу не были вызваны, либо были вызваны позднее. На основе этих отчетов можно подумать о реорганизации программного кода с целью удаления ненужных функций и откладывания загрузки сценариев, не требующихся на этапе первоначальной настройки и отображения страницы.

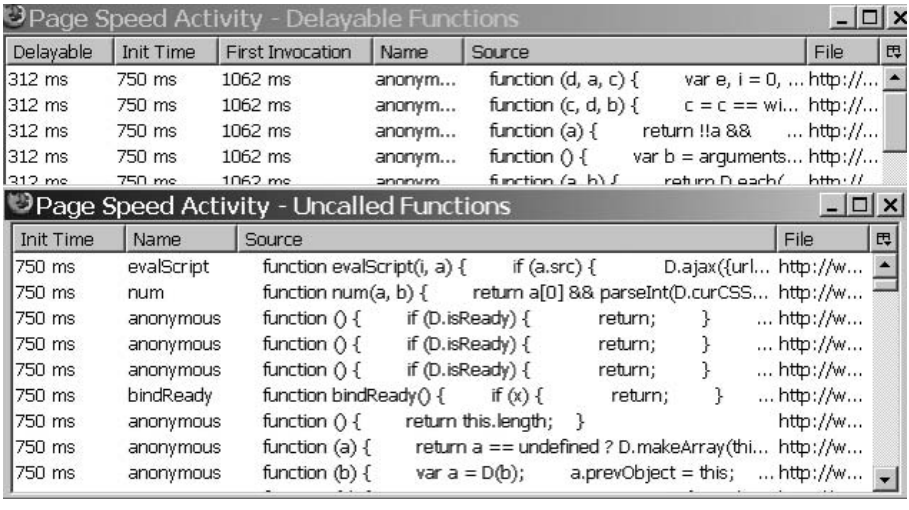


Рис. 10.14. *Отчеты со списками отложенных и не вызывавшихся функций*

Fiddler

Fiddler – это промежуточный программный компонент, предназначенный для отладки HTTP-трафика, который исследует ресурсы, поступающие из сети, и помогает идентифицировать узкие места на этапе загрузки. Этот инструмент анализа сетевого трафика для Windows, созданный Эриком Лоуренсом (Eric Lawrence), позволяет создавать подробные отчеты в любых броузерах или по веб-запросу. Дополнительную информацию по установке и использованию можно найти по адресу *http://www.fiddler2.com/fiddler2/*.

В процессе установки программа Fiddler автоматически интегрируется в IE и Firefox. В результате в Internet Explorer появляется дополнительная кнопка на панели инструментов, а в Firefox – новый пункт в меню Tools (Инструменты). Программу Fiddler можно также запустить вручную. Программа способна проанализировать любой броузер или приложение, выполняющее веб-запросы. В процессе работы весь WinINET-трафик направляется через программу Fiddler, что позволяет ей анализировать скорость загрузки ресурсов. Некоторые броузеры (такие как Opera и Safari) не используют интерфейс WinINET, но они определяются программой Fiddler автоматически, если программа была запущена до запуска броузера. С помощью программы Fiddler можно проанализировать трафик любой программы, способной работать через прокси-сервер, для чего необходимо в качестве прокси-сервера указать программу Fiddler (127.0.0.1, порт: 8888).

Подобно расширениям Firebug, Page Speed и веб-инспектору, программа Fiddler создает временные диаграммы, позволяющие выяснить, за-

грузка каких ресурсов занимает больше всего времени и какие ресурсы могут оказывать влияние на загрузку других ресурсов (рис. 10.15).

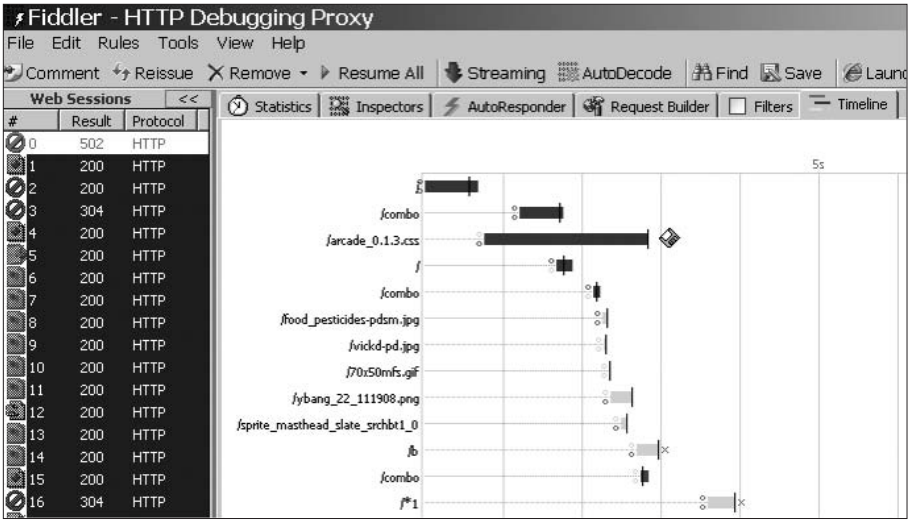


Рис. 10.15. *Временная диаграмма, созданная программой Fiddler*

Если выбрать один или несколько ресурсов в панели слева, информация о них отобразится в основной части окна. Щелкните на вкладке Timeline (Временная диаграмма), чтобы вывести график загрузки ресурсов из сети. Эта диаграмма отображает временной график загрузки ресурсов относительно друг друга, который позволяет выявлять влияние различных стратегий загрузки и делает более очевидными ситуации, когда один ресурс блокирует загрузку других ресурсов.

Вкладка Statistics (Статистика) дает полное представление о фактическом времени выполнения различных операций в ходе загрузки всех выбранных ресурсов, в том числе позволяет узнать время поиска в DNS и выполнения соединения TCP/IP, а также размеры и типы различных запрашиваемых ресурсов (рис. 10.16).

Эта информация поможет определить области приложения дальнейших усилий. Например, большие затраты времени на поиск в DNS и выполнение соединений TCP/IP могут свидетельствовать о проблемах в сети. Круговая диаграмма дает наглядное представление об объемах загрузки ресурсов различных типов и позволяет выявлять наиболее подходящих кандидатов для отложенной загрузки или профилирования (в случаях со сценариями).

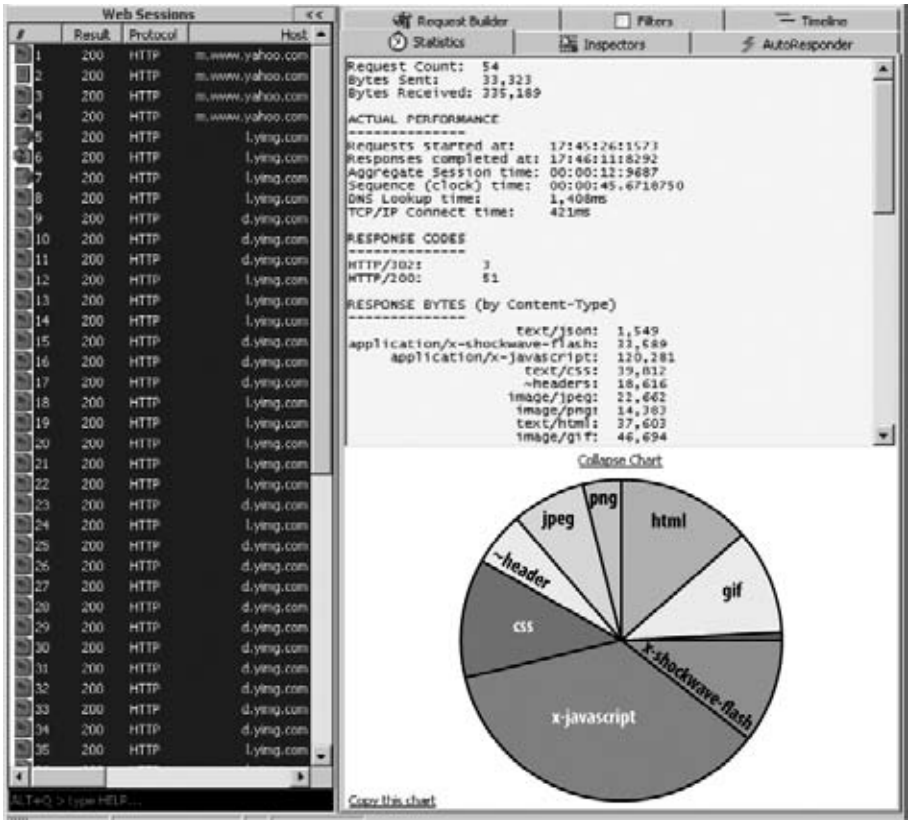


Рис. 10.16. *Вкладка Statistics в окне программы Fiddler*

Программа Fiddler доступна только для Windows. Следует также упомянуть условно-бесплатный продукт под названием Charles Proxy, который может работать и в Windows, и в Mac OS. Получить пробную версию и ознакомиться с описанием этого продукта можно по адресу *http://www.charlesproxy.com/*.

YSlow

Инструмент YSlow позволяет получить подробную информацию о времени загрузки и выполнения сценариев на этапе начального отображения страницы. Этот инструмент был разработан Стивом Содерсом (Steve Souders) для внутренних нужд компании Yahoo! как расширение для Firefox (выполняющееся под управлением расширения GreaseMonkey). Впоследствии он был сделан общедоступным как дополнение для Firebug, поддерживается и регулярно обновляется разработчиками из

компании Yahoo!. Инструкции по установке и описание продукта можно найти по адресу *http://developer.yahoo.com/yslow/*.

YSlow оценивает время загрузки внешних ресурсов, создает отчет о производительности и дает рекомендации по увеличению скорости загрузки страницы. Оценки, предоставляемые программой, основываются на богатом опыте исследований, проводившихся экспертами в области производительности. Применение этих рекомендаций и знакомство с опытом, лежащим в их основе, поможет обеспечить максимально возможную кажущуюся скорость загрузки страницы за счет сокращения числа загружаемых ресурсов до необходимого минимума.

На рис. 10.17 изображено, как выглядит в панели YSlow представление по умолчанию проанализированной веб-страницы. Здесь приводится список рекомендаций по оптимизации времени загрузки и отображения страницы. Каждая рекомендация включает дополнительную информацию и описание предпосылок.

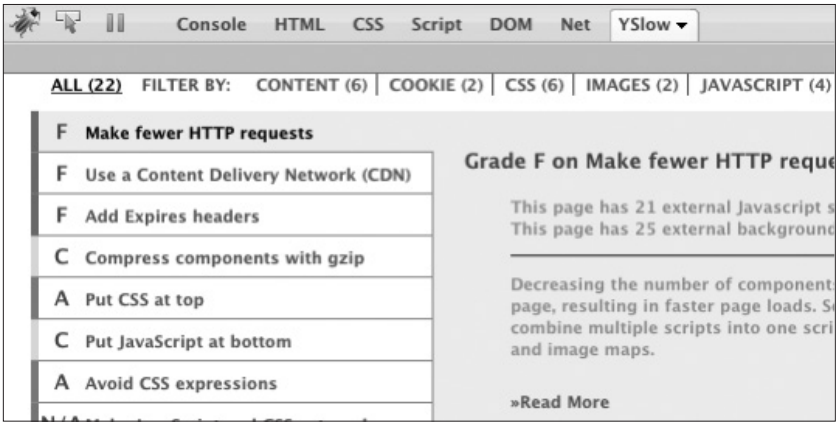


Рис. 10.17. *YSlow: все результаты*

В целом, применение рекомендаций позволит ускорить загрузку и выполнение сценариев. На рис. 10.18 показаны рекомендации, отфильтрованные по параметру JAVASCRIPT, где даются советы по оптимизации загрузки и выполнения сценариев.

Интерпретируя результаты, имейте в виду, что иногда могут возникать исключения, которые придется учитывать. Например, может понадобиться решить, когда загружать сценарии по отдельности, а когда объединять множество сценариев в один сценарий, и загрузку каких сценариев и функций следует отложить до отображения страницы.

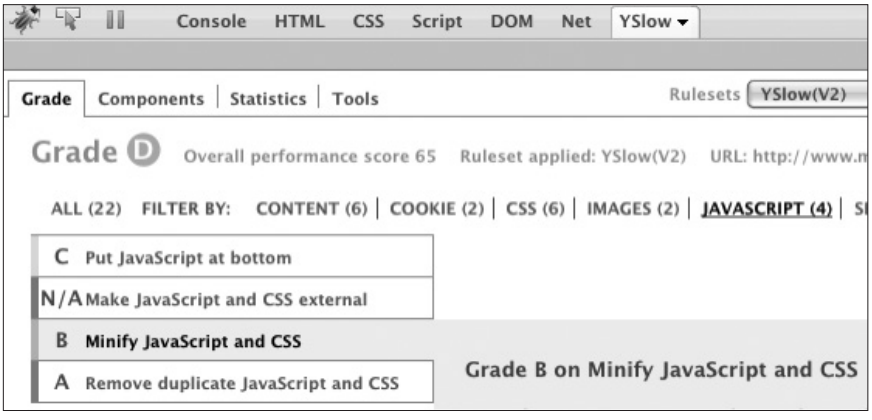


Рис. 10.18. *YSlow: рекомендации по оптимизации JavaScript-сценариев*

dynaTrace Ajax Edition

Разработчики dynaTrace, надежного инструмента диагностики производительности на платформе Java/.NET, выпустили версию «Ajax Edition», которая позволяет выполнять диагностику производительности в Internet Explorer (в ближайшее время ожидается выход версии для Firefox). Этот свободно распространяемый инструмент позволяет выполнять непрерывный анализ производительности – от сетевых операций и отображения страницы до выполнения сценариев и использования центрального процессора. Все аспекты отображаются в едином отчете, что позволяет легко отыскать узкие места. Результаты могут экспортироваться для дальнейшего изучения и анализа. Загрузить dynaTrace можно по адресу *http://ajax.dynatrace.com/pages/*.

Сводный отчет Summary, изображенный на рис. 10.19, дает наглядное представление, позволяющее быстро определить области, требующие дополнительного внимания. Из этого отчета можно перейти к изучению более узкоспециализированных отчетов за получением более подробных сведений о продолжительности различных этапов.

Отчет Network (Сеть), изображенный на рис. 10.20, содержит весьма подробные сведения о продолжительности каждого этапа сетевой активности, включая время поиска в DNS, установления соединения и ответа сервера. Эта информация может подсказать, какие сетевые настройки, возможно, требуется скорректировать. В панелях под отчетом отображаются заголовки запроса и ответа (слева) и фактическое тело ответа (справа).

Отчет JavaScript Triggers (Выполнение JavaScript) содержит подробную информацию о каждом событии, возникшем в процессе анализа (рис. 10.21). Из этого отчета можно перейти к исследованию конкретных событий

(`load`, `click`, `mouseover` и других), чтобы отыскать основную причину низкой производительности.

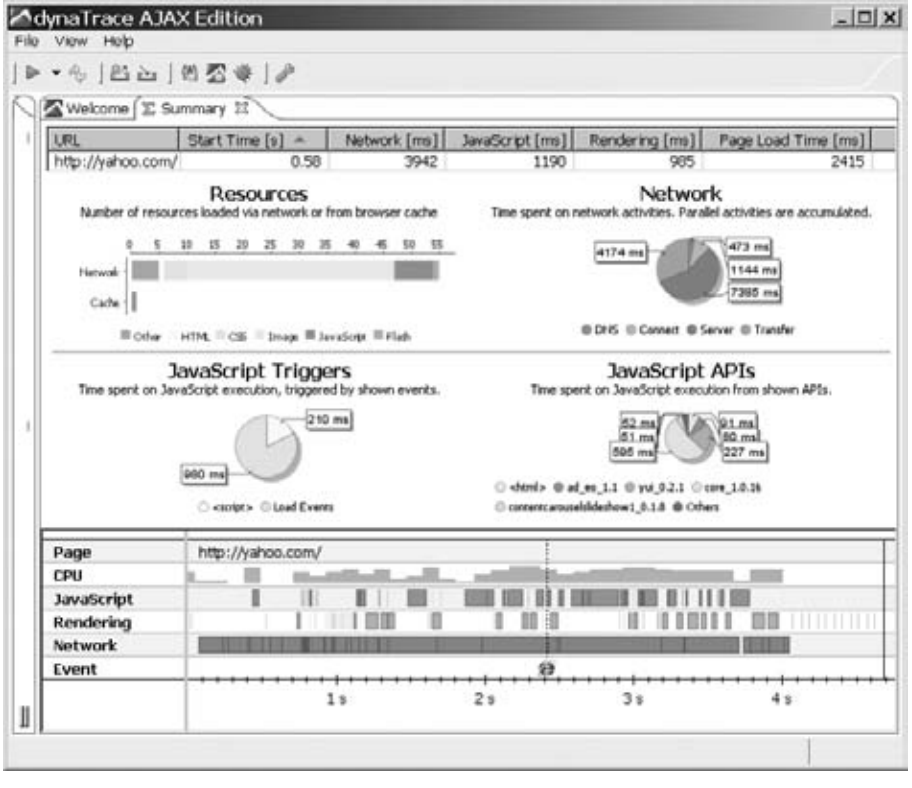


Рис. 10.19. *dynaTrace Ajax Edition: сводный отчет Summary*

Этот отчет включает также информацию обо всех динамических (Ajax) запросах, которые могут генерировать события, и все «обратные вызовы» в сценарии, которые могут производиться по окончании выполнения запросов. Это позволяет лучше понять, из чего складывается общая производительность, что может быть невозможно при профилировании сценария из-за асинхронной природы Ajax.

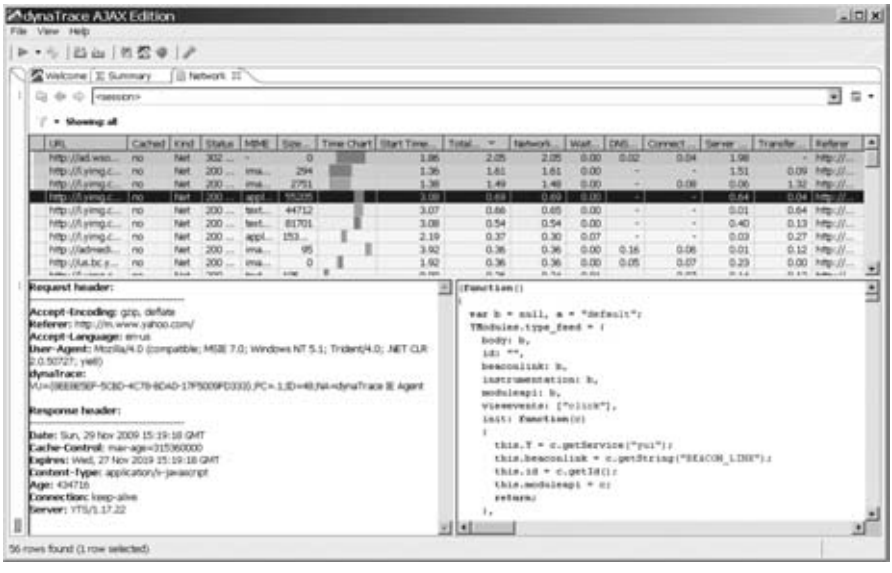


Рис. 10.20. *dynaTrace Ajax Edition: отчет Network*

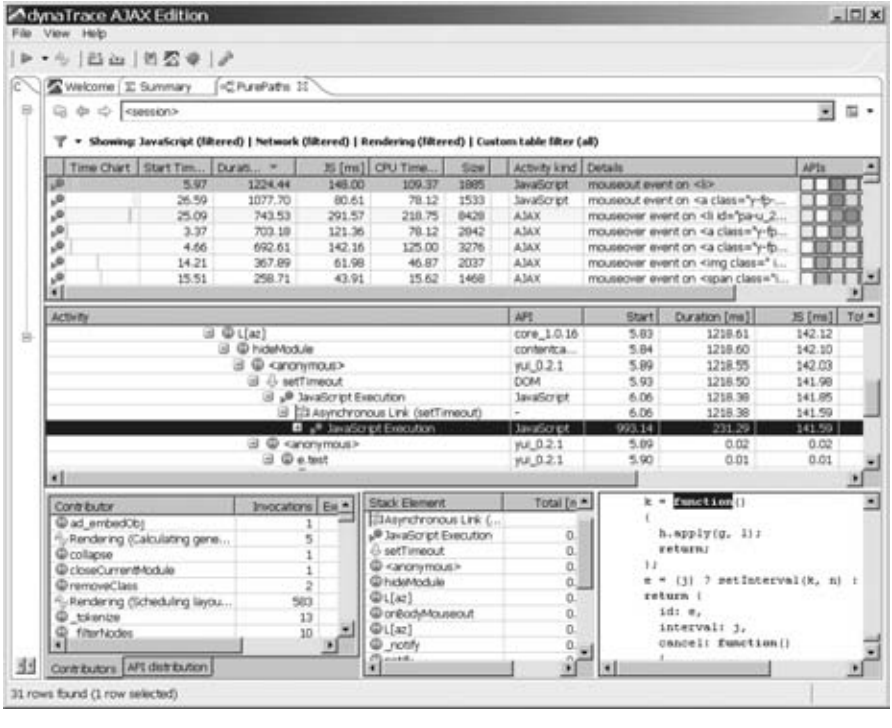


Рис. 10.21. *dynaTrace Ajax Edition: панель PurePaths*

В заключение

Когда веб-страницы или приложения начинают выглядеть медлительными, анализ динамики загрузки ресурсов из сети и профилирование сценариев во время их выполнения позволит сконцентрировать усилия по оптимизации там, где это действительно необходимо.

- Используйте инструменты анализа сетевого трафика для выявления узких мест на этапе загрузки сценариев и других ресурсов; это поможет определить, загрузку каких сценариев лучше отложить, а для каких следует выполнить профилирование.
- Хотя опыт подсказывает, что нужно максимально уменьшить количество HTTP-запросов, тем не менее иногда отложенная загрузка сценариев может обеспечить более высокую скорость отображения страницы и произвести на пользователя более благоприятное впечатление.
- Используйте профилировщики для выявления наиболее медленных участков сценариев. Исследование времени выполнения каждой функции, количества вызовов функций и самого стека вызовов позволит определить направления, в которых должны быть сосредоточены усилия по оптимизации.
- Наиболее ценной обычно является информация о времени выполнения функций и о количестве вызовов, однако внимательное изучение того, как эти функции вызываются, может натолкнуть на мысли о других способах оптимизации.

Инструменты, описанные в этой главе, помогают раскрыть тайны в целом не очень дружелюбных окружений, в которых должны выполняться современные программы. Использование их перед началом оптимизации гарантирует, что время будет потрачено на решение истинных проблем.

Алфавитный указатель

Символы

+ (плюс), оператор, 110
+= (плюс-равно), оператор, 110

A

add(), функция, 39
addEventListener(), функция, 195
Ajax (Asynchronous JavaScript and XML, асинхронный JavaScript и XML), 160
  передача данных, 160
  форматы данных, 171
Apache, веб-сервер
  директива ExpiresDefault, 213
Apache Ant, инструмент сборки, 204
array.join(), метод, 110
assignEvents(), функция, 48

B

BOM (Browser Object Model – объектная модель броузера), 50

C

childNodes, коллекция, 71
Chrome, броузер
  инструменты разработчика, 237
  ограничение времени выполнения сценариев, 142
  синхронный компилятор JavaScript, 202
Closure Compiler, инструмент минификации JavaScript-файлов, 211
Closure Inspector, расширение для Firebug, 211
concat(), метод, 115
console.time(), функция, 228
cssText, свойство, 78
CSS-селекторы, прикладной интерфейс, 72
CSS-файлы, загрузка, 34

D

Date, объект, 220
displayName, свойство, 234
do-while, циклы, 88
dynaTrace, инструмент анализа производительности, 245

E

eval(), функция, 46, 176, 191
execute(), функция, 47
Expires, заголовки, 186
ExpiresDefault, директива, веб-сервер Apache, 213

F

factorial(), функция, 103, 106
Fiddler, инструмент анализа производительности, 241
Firebug, расширение, 226
Firefox, броузер
  ограничение времени выполнения сценариев, 142
  свертка на этапе компиляции, 112
forEach(), метод, 94
for, циклы, 88
for-in, циклы, 89
Function(), конструктор, 191

G

GET- и POST-запросы при использовании объекта XHR, 162
gzip-сжатие, 210, 212

H

hasOwnProperty(), метод, 51
:hover, псевдоселектор, IE, 82
HTML, язык разметки как формат данных, 179
HTML-коллекции
  дорогие коллекции, 66
  локальные переменные, 69
HTTP-заголовки, Ajax, 186

I

IE (Internet Explorer), броузер
  nextSibling, свойство, 71
  выполнение повторяющихся действий, 144
  динамические элементы script, 29
  использование, 230
  объекты XHR, 188
  слияние элементов массива, 113

if-else, инструкция
оптимизация, 97
сравнение с инструкцией switch, 95
initUI(), функция, 44
innerHTML, свойство
в сравнении с методами DOM, 61

J

JavaScript-файлы
кэширование, 213
минификация, 209
объединение, 205
предварительная обработка, 206
jQuery, библиотека
результаты gzip-сжатия, 212
JSMin, инструмент минификации JavaScript-файлов, 209
JSON (JavaScript Object Notation – форма записи объектов на языке JavaScript), 175
JSON-P (JSON with padding – JSON с дополнением), 178

L

$LAB.script(), метод, 35
$LAB.wait(), метод, 35
LABjs, библиотека, загрузка сценариев, 35
LazyLoad, библиотека, загрузка сценариев, 34
length, свойство, 66
loadScript(), функция, 30

M

mergeSort(), функция, 104
message, событие, 156
multistep(), функция, 152

O

onmessage, обработчик события, 156
Opera, броузер
ограничение времени выполнения сценариев, 143

P

Page Speed, инструмент анализа производительности, 239
postMessage(), метод, 156
POST- и GET-запросы при использовании объекта XHR, 162
processArray(), функция, 150
profileEnd(), функция, 228

Q

querySelector(), метод, 201
querySelectorAll(), метод, 73, 201

R

readyState, свойство
использование при обработке составных данных, 166
объекта XHR, 162
removeEventListener(), функция, 195

S

Safari, броузер
загрузка сценариев, 238
запуск и остановка профилирования программным способом, 233
ограничение времени выполнения сценариев, 142
передача данных в виде строк, 157
[[Scope]], свойство, 48
<script>, теги, динамические, 163
<script>, элементы
defer, атрибут, 26
местоположение, 22
setInterval(), функция, 145, 191
setTimeout(), функция, 145, 191
smasher, приложение, 216
string.concat(), метод, 110
switch, инструкция
сравнение с инструкцией if-else, 95

T

this, ссылка в методах объектов, 56
toString(), метод, 51
trim(), метод, 132
try-catch, инструкция, 45, 102

V, W

var, инструкция, 88
while, циклы, 88
with, инструкция, 44

X

XHR (XMLHttpRequest), объект, 161
GET- и POST-запросы, 162
в IE, 188
запрос составных данных, 164
отправка данных, 168
XML, формат данных, 171
XPath, язык запросов к элементам XML-документов, 174

Y

YSlow, инструмент анализа производительности, 243
YUI 3, библиотека, загрузка сценариев, 33
YUI Compressor, инструмент минификации JavaScript-файлов, 209
YUI Profiler, профилировщик, 221

А

алгоритмы, 87
  рекурсия, 101
  условные инструкции, 95, 97
  циклы, 87
анонимные функции, 225
  YUI Profiler, профилировщик, 225
атомарная группировка, имитация, 123

Б

библиотеки
  Ajax, 188
  LABjs, 35
  LazyLoad, 34
  YUI 3, 33
битовые маски, 199
битовые операторы, 197
блокирование отображения страницы сценариями, 238
броузеры, 21
  блокирование отображения страницы сценариями, 238
  главный поток выполнения, 139
  на основе WebKit и свойство innerHTML, 64
  ограничения, 141
  ограниченность размера стека вызовов, 101
  поддержка языка XPath, 174
  поток выполнения пользовательского интерфейса, 139
  производительность, 21, 37
  работа с деревом DOM, 58
  реализации DOM и JavaScript, 58
  сравнение скорости обычного программного кода и функции eval(), 192
  усечение строк, 137

В

варианты выбора
  и возвраты, 119
  производительность, 128
веб-инспектор (Safari), 233
вложенные квантификаторы, 125
  исключение возвратов, 125
вложенные члены объектов, 54
внешние файлы, загрузка, 157
возвраты, 118
  исключение возвратов, 121
время анализа, XML, 174
встроенные методы, 200
выделение обязательных комбинаций, 129

Г

гибкий процесс сборки JavaScript-файлов, 216
глобальные переменные, производительность, 41
группировка изменений в дереве DOM, 79
группировка сценариев, 24

Д

делегирование обработки событий, 83
деление заданий, 150
дерево отображения
  DOM, 75
  перекомпоновка, 75
динамические области видимости, 46
динамические теги <script>, 163
динамические элементы script, 27
доступ к данным, 37
  области видимости, 38
  члены объектов, 50

З

заголовки
  Expires, 186
  HTTP-заголовки, 186
загрузка, 157
  CSS-файлов, 34
  внешних файлов, 157
  сценариев, 32, 238
замыкания, области видимости, 47
запрос данных, Ajax, 161
запросы, HTML-коллекции, 67
значения членов объектов, кэширование, 55

И

идемпотентная операция, 162
измерение производительности регулярных выражений, 127
имитация атомарной группировки, 123
инструкции
  try-catch, 45, 102
  var, 88
  with, 44
инструменты, 219
  dynaTrace, 245
  Fiddler, 241
  Firebug, 226
  Page Speed, 239
  YSlow, 243
  YUI Profiler, профилировщик, 221
  веб-инспектор в броузере Safari, 233
  профилирования анонимных функций, 225
  профилирования JavaScript-сценариев, 220
  разработчика в Chrome, 237
  разработчика в IE (Internet Explorer), 230

инъекция сценариев с помощью XMLHttpRequest, 31
исключение возвратов, 121
использование таймеров, 145
итерации
  и рекурсия, 104
  на основе функций, 94

К

квантификаторы
  вложенные, 125
  производительность, 130
коллекции
  childNodes, 71
  HTML-коллекции, 66
  элементы коллекций, 69
конкатенация строк, 110
константы математические, 200
контекст выполнения, 40
копирование узлов, 65
кэширование
  JavaScript-файлов, 213
  данных при использовании технологии Ajax, 185
  значений членов объектов, 55
  информации о размещении, 81

Л

литеральные значения, определение, 37
локальные переменные
  HTML-коллекции, 69
  производительность, 41, 60

М

массивы
  в сравнении с объектами HTMLCollection, 66
  цикл for-in, 89
математические константы и методы, список, 200
мемоизация, рекурсивных функций, 105
местоположение сценариев, 22
методы, 50, 200
  $LAB.script(), 35
  $LAB.wait(), 35
  array.join(), 110
  concat(), 115
  forEach(), 94
  hasOwnProperty(), 51
  postMessage(), 156
  querySelector(), 201
  querySelectorAll(), 73, 201
  string.concat(), 110
  toString(), 51
  trim(), 132
  встроенные методы, 200
  конкатенации строк, 110
  математические методы, 201
  ссылка this в методах объектов, 56
минификация JavaScript-файлов, 209

Н

навигация по дереву DOM, 71
неблокирующая загрузка сценариев, 26
  динамические элементы script, 27
  инъекция сценариев с помощью XMLHttpRequest, 31
  отложенные сценарии, 26
  рекомендуемый способ, 32
несохраняющая группировка, 129

О

области видимости, 39
  динамические, 46
  и замыкания, 47
обработка массивов с помощью таймеров, 148
обратные ссылки, имитация атомарной группировки, 123
объединение JavaScript-файлов, 205
объект активации, 40
объекты
  Date, 220
  HTMLCollection, 66
  объект активации, 40
  приемы программирования, 193
ограничения продолжительности выполнения сценариев, 142
ограниченность размера стека вызовов, 101
операторы битовые, 197
опережающие проверки, имитация атомарной группировки, 123
отложенная загрузка, 195
отложенные сценарии, 26
отправка данных, Ajax, 168

П

панель
  Net, расширение Firebug, 229
  Profiles (Safari), 233
  Resources (Safari), 236
  профилировщика в консоли, расширение Firebug, 227
передача данных, 160
  запрос данных, 161
  отправка данных, 168
перекомпоновка, 75
  буферизация и применение изменений в дереве отображения, 76
  кэширование информации о размещении, 81
  уменьшение количества операций, 78

переменные, 41
локальные и глобальные, сравнение, 41
определение, 37
перерисовывание, уменьшение количества операций, 78
повторения и возвраты, 120
повторная интерпретация, 191
повторная работа, 194
поисковые таблицы, 99
пользовательские интерфейсы, 139
главный поток выполнения броузера, 139
использование таймеров, 145
поток выполнения пользовательского интерфейса броузера, 139
фоновые потоки выполнения, 155
потоки выполнения
главный поток выполнения броузера, 139
поток выполнения пользовательского интерфейса броузера, 139
предварительная обработка JavaScript-файлов, 206
предварительная условная загрузка, 196
приемы программирования, 191
битовые операторы, 197
встроенные методы, 200
литералы объектов и массивов, 193
отложенная загрузка, 195
повторная интерпретация, 191
повторная работа, 194
предварительная условная загрузка, 196
прикладной интерфейс консоли, расширение Firebug, 228
прикладные интерфейсы
CSS-селекторов, 72
DOM, 58
применение изменений в дереве отображения, 76
производительность
Ajax, 185
JavaScript в броузерах, 21
броузеры, 37
вложенные члены объектов, 54
и замыкания, 47
интерпретаторов JavaScript, 47
и цепочки областей видимости, 44
обработки данных в формате
HTML, 181
JSON, 177
JSON-P, 178
XML, 174
работа с деревом DOM, 58, 59
разрешения идентификаторов, 41
реализаций метода trim(), 137
регулярных выражений, 116, 127
сравнение скорости обычного программного кода и функции eval(), 192
таймеров, 154
циклов, 89
пространства имен, вложенные свойства, 55
прототипы, 50
профилирование JavaScript-сценариев, 220
процесс сборки JavaScript-файлов, 216

P

работа с деревом DOM, 58
HTML-коллекции, 66
innerHTML, свойство, 61
броузеры, 58
делегирование обработки событий, 83
доступ к структуре документа, 71
копирование узлов, 65
перерисовывание и перекомпоновка, 75
развертывание JavaScript-ресурсов, 215
разрешение идентификаторов, области видимости, 39
регулярные выражения, 116
атомарная группировка, 123
возвраты, 118, 121
как работают, 117
когда не следует использовать, 131
повторения, 120
производительность, 116, 127
усечение строк, 132, 137
рекурсия, 101
и итерации, 104
мемоизация, 105
ограниченность размера стека вызовов, 101
шаблоны реализации, 103

C

сборка в виде отдельного этапа или во время выполнения, 211
свертка на этапе компиляции, Firefox, 112
свойства, 50
cssText, 78
displayName, 234
innerHTML, 61
length, 66
прототипы, 50
чтение в функциях, 55
селекторы, CSS, 72
сеть распространения содержимого (Content Delivery Network, CDN), 215
сжатие, 210, 211
сигналы, отправка данных, 169
слияние элементов массива, 113

события
message, 156
onmessage, обработчик события, 156
составные данные, запрос с помощью объекта XHR, 164
стек вызовов, ограниченность размера, 101
стили, перерисовывание и перекомпоновка, 78
строки
конкатенация, 110
усечение строк, 132
сценарии, 21
загрузка, 238
местоположение, 22
отладка и профилирование, 226

Т

таблицы поисковые, 99
таймеры
и производительность, 154
использование, 145
точность, 148
типы данных
функции, методы и свойства, 50
точечная нотация и форма записи с квадратными скобками, сравнение, 55

У

узлы, копирование, 65
узлы-элементы DOM, 72
управление потоком выполнения, 87
рекурсия, 101
условные инструкции, 95, 97
циклы, 87
усечение строк, 132, 137
условные инструкции, 95, 97
if-else, 95
switch, 95
поисковые таблицы, 99

Ф

файлы, 157
загрузка внешних файлов, 157
кэширование JavaScript-файлов, 213
минификация JavaScript-файлов, 209
объединение JavaScript-файлов, 205
предварительная обработка JavaScript-файлов, 206
фоновые потоки выполнения, 155
взаимодействие, 156
загрузка внешних файлов, 157
окружение, 155
практическое использование, 157
форматы данных, 171
HTML, 179
JSON, 175
XML, 171
нестандартные, 181
сравнение производительности, 184
фрагменты документа, группировка изменений в дереве DOM, 80
функции, 150
add(), 39
addEventListener(), 195
assignEvents(), 48
console.time(), 228
eval(), 46, 176, 191
execute(), 47
factorial(), 103, 106
initUI(), 44
loadScript(), 30
mergeSort(), 104
multistep(), 152
processArray(), 150
profileEnd(), 228
removeEventListener(), 195
setInterval(), 145, 191
setTimeout(), 145, 191
анонимные функции, 225
кэширование значений членов объектов, 55
локальные переменные, 40

Ц

цепочки областей видимости
и разрешение идентификаторов, 39
производительность, 44
увеличение, 44
цепочки прототипов, члены объектов, 52
циклы, 87
на основе функций, 94
производительность, 89
типы, 88
хронометраж с помощью функции console.time(), 228

Ч

члены объектов, 50
вложенные, 54
доступ к данным, 50
кэширование значений членов объектов, 55
определение, 37
прототипы, 50
цепочки прототипов, 52

Э

элементы, 69
элементы массивов, определение, 37

Стоян СТЕФАНОВ

JavaScript. Шаблоны

272 стр., книга в продаже

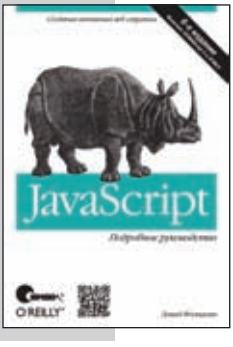
Если вы опытный разработчик, то наверняка часто сталкиваетесь с проблемами, связанными с объектами, функциями, наследованием и другими особенностями JavaScript.

Данная книга призвана вам помочь, продемонстрировав большое количество различных шаблонов и приемов программирования на этом языке. Это могут быть такие подходы, как отказ от глобальных переменных, применение литералов и многие другие. Книга познакомит с различными способами определения функций на языке JavaScript, научит создавать объекты, отличающиеся по сложности от базовых образцов, познакомит с различными приемами повторного использования программного кода и наследования в JavaScript. Рассматриваются примеры реализации распространенных шаблонов проектирования, таких как «единственный объект» (singleton), «фабрика» (factory), «декоратор» (decorator) и других. Вы также познакомитесь с антишаблонами – приемами, которых следует избегать.

Книга написана экспертом по языку JavaScript Стояном Стефановым, ведущим специалистом компании Yahoo!.

Дэвид ФЛЭНАГАН

JavaScript. Подробное руководство, 6-е издание



1080 стр., книга в продаже

Шестое издание бестселлера «JavaScript. Подробное руководство» полностью пересмотрено и дополнено сведениями о JavaScript в соответствии с современным положением дел в разработке приложений для Web 2.0. Эта книга – одновременно и руководство программиста с большим числом практических примеров, и полноценный справочник по базовому языку JavaScript и клиентским прикладным интерфейсам, предоставляемым веб-броузерами.

Издание охватывает стандарты ECMAScript 5 и HTML5. Многие главы переписаны заново, другие дополнены новой информацией, появились новые главы с описанием библиотеки jQuery и поддержки JavaScript на стороне сервера. Часть I – основы JavaScript. В части II описывается среда разработки сценариев, предоставляемая броузерами. Основное внимание уделяется разработке сценариев с применением методики ненавязчивого JavaScript и модели DOM. Часть III – обширный справочник по базовому JavaScript, включающий описания всех классов, объектов, конструкторов, методов, функций, свойств и констант, определенных в JavaScript 1.8, V8 3.0 и ECMAScript 5. Часть IV – справочник по клиентскому JavaScript. Здесь описываются API веб-броузеров, стандарт DOM API Level 3 и недавно вошедшие в стандарт HTML5 технологии WebSockets и WebWorkers, объекты localStorage и sessionStorage, а также теги <audio> и <video>.

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:
московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!