Programming the Perl DBI

Alligator Descartes, Tim Bunce

O'REILLY®

Программирование на Perl DBI

Аллигатор Декарт и Тим Банс

Символ

Санкт-Петербург
2001

Аллигатор Декарт, Тим Банс

Программирование на Perl DBI

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Рижий*
Редакторы	*Е. Изотова, Н. Макарова*
Корректура	*С. Беляева*
Верстка	*Е. Изотова, Н. Макарова*

Декарт А., Банс Т.

Программирование на Perl DBI. – Пер. с англ. – СПб: Символ-Плюс, 2001. – 400 с., ил.

ISBN 5-93286-013-8

Данная книга будет полезна как новичкам, которые найдут в ней описание архитектуры DBI и подробные инструкции по написанию программ с помощью DBI, так и знатокам DBI, которым предназначено описание тонкостей использования DBI и специфических особенностей отдельных драйверов DBD.

DBI является основным интерфейсом программирования баз данных на Perl. Любая программа, использующая DBI, может работать с любой базой данных или даже одновременно с несколькими базами данных различных фирм, такими как Oracle, Sybase, Ingres, Informix, MySQL, Access и другие.

Издание содержит полный справочник по DBI. Книга написана с учетом того, что читатель имеет базовые навыки программирования на Perl и может писать простые сценарии.

ISBN 5-93286-013-8
ISBN 1-56592-699-4 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 567-8775. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 31.10.2000. Формат 70x100$^{1}/_{16}$. Бумага офсетная.
Печать офсетная. Объем 25 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в ГПП «Печатный Двор» Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Предисловие

DBI является стандартным интерфейсом баз данных для языка программирования Perl. DBI не зависит от базы данных, т. е. позволяет работать практически с любой базой данных, в том числе с Oracle, Sybase, Informix, Access, MySQL и т. д.

Мы предполагаем у читателей данной книги наличие некоторого опыта работы с Perl, но не ожидаем хорошего знакомства с собственно базами данных. Изложение в книге развертывается неспешно, описывая различные типы баз данных и знакомя читателя с общепринятой терминологией.

Эта книга посвящена не только DBI, она касается более общей темы хранения и извлечения данных из баз различного вида. Это выражается и в том, что книга состоит из двух взаимосвязанных, но независимых частей. В первой части рассказывается о способах хранения и извлечения данных без использования DBI, в то время как вторая, и значительно большая по размеру часть охватывает использование DBI и родственных технологий.

На протяжении всей книги предполагается, что у читателя есть базовые навыки программирования на Perl, и он может самостоятельно писать простые сценарии. Если ваш уровень знания Perl не достаточен, советуем прочесть какие-нибудь книги по Perl, перечисленные в разделе «Дополнительные источники информации» данного предисловия.

Если вы достаточно подготовлены, можете читать книгу не подряд, а в зависимости от того, какие темы вас больше интересуют. Если вас интересует только DBI, можете безболезненно пропустить главу 2. С другой стороны, если вы хорошо разбираетесь в SQL, то, вероятно, пропустите главу 3, которая огорчила бы вас отсутствием описания многих важных тонкостей. В главе 7 сравниваются DBI и ODBC, и она интересна, в основном, «пахарям» от баз данных, ревнителям проектирования и тем, кто отчаянно пытается перенести свои приложения `Win32::ODBC` на DBI.

Вот перечень содержания книги по главам:

Глава 1 «Введение».

Позволяет получить общее представление о книге.

Глава 2 «Основные базы данных, не использующие DBI».

В этой главе рассказывается об основах хранения и извлечения данных с помощью базовых функций Perl посредством одноуровневых баз данных, иначе называемых базами данных на плоских файлах (flat-file database), использующих разделители или поля фиксированной ширины, либо через модули, не использующие DBI, такие как `AnyDBM_File`, `Storable`, `Data::Dumper` и другие. Хотя в этой главе не упоминается непосредственно DBI, способ, которым модули `Storable` и `Data::Dumper` упаковывают структуры данных Perl в строки, может легко быть применен и к DBI.

Глава 3 «SQL и реляционные базы данных».

В этой главе дается обзор основ SQL и реляционных баз данных, а также объясняется, как можно писать простые, но мощные SQL-выражения для построения запросов к базам данных и обработки их результатов. Если вы в какой-то мере знакомы с SQL, можете пропустить эту главу. Если же вы не знаете SQL, советуем прочесть ее, поскольку в последующих главах предполагается наличие базовых знаний об SQL и реляционных базах данных.

Глава 4 «Программирование с помощью DBI».

Эта глава знакомит с DBI путем изложения его архитектуры и основных DBI-операций, таких как соединение с базами данных и обработка ошибок. Эту главу важно прочесть, поскольку в ней описывается структура, которую предоставляет DBI для написания простых, мощных и надежных программ.

Глава 5 «Взаимодействие с базами данных».

В этой главе изложен основной материал по использованию DBI и обсуждается работа с данными в базе – извлечение данных, которые в ней уже хранятся, вставка новых данных, удаление и обновление существующих данных. Мы рассказываем о различных способах, которыми можно осуществлять эти операции, от простого этапа «заставить это работать» до более развитых и оптимизированных приемов работы с данными.

Глава 6 «Более сложные вопросы использования DBI».

Эта глава охватывает более сложные темы в сфере действия DBI, такие как тонкая настройка действия DBI в ваших программах, работа с типами данных LONG/LOB, метаданные команд и баз данных и, наконец, обработка транзакций.

Глава 7 «ODBC и DBI».

В этой главе обсуждается различие архитектур DBI и ODBC – другого переносимого API баз данных. И конечно, особое внимание уделено вопросу, почему программировать с помощью DBI проще.

Глава 8 «Командный процессор для DBI и прокси-доступ к базам данных».

В этой главе обсуждаются две темы, не имеющие непосредственного отношения к ядру DBI как таковому, но знание которых оказывается очень полезным. Сначала мы расскажем об оболочке DBI – инструменте командной строки, позволяющем подключаться к базам данных и делать произвольные запросы. Другая тема – архитектура доверительного доступа (proxy), с помощью которой DBI может, в частности, подключать сценарии на одной машине к базам данных на другой без необходимости установки программного обеспечения для сетевой работы с базами данных. Например, можно подключать сценарии, выполняющиеся на Unix-системе, к базе данных Microsoft Access, работающей на машине под Microsoft Windows.

Приложение A «Спецификация DBI».

Приложение содержит спецификацию DBI, поставляемую с `DBI.pm`.

Приложение B «Характеристики драйверов и баз данных».

В приложении содержатся полезные дополнительные данные обо всех часто используемых DBD и соответствующих им базах данных.

Приложение C «Хартия хранителей священных площадок ASLaN».

В этом приложении содержится Хартия сети хранителей древних святилищ (Ancient Sacred Landscape Network), имеющая целью сохранение древних священных мест, таких как мегалитические площадки, используемые для примеров в этой книге.

Дополнительные источники информации

Чтобы облегчить изучение некоторых тем книги, перечислим дополнительные источники, которые вам могут потребоваться до, во время или после чтения этой книги.

http://www.symbolstone.org/technology/perl/DBI

Домашняя страница DBI. Этот сервер содержит массу полезных сведений о DBI и о том, откуда получить различные модули. Здесь вы также найдете ссылки на активный список рассылки dbi-users и архивы.

http://www.perl.com/CPAN

Сервер объединяет Comprehensive Perl Archive Network – всестороннюю архивную сеть Perl, в которой можно найти массу полезных модулей, в том числе DBI.

C. J. Date «An Introduction to Database Systems»[1].

Стандартный учебник по базам данных, весьма рекомендуемый для чтения.

C. J. Date, Hugh Darwen «A Guide to the SQL Standard».

Отличная книга, подробная, но небольшая по объему и легко читаемая.

http://w3.one.net/~jhoffman/sqltut.htm

http://www.jcc.com/SQLPages/jccs_sql.htm

http://www.contrib.andrew.cmu.edu/~shadow/sql.html

На этих сайтах содержатся информация, спецификации и ссылки по языку запросов SQL, введение в который мы даем в главе 3. Дополнительную информацию можно получить, введя «SQL tutorial» или аналогичное выражение в строку поиска вашей любимой поисковой машины Интернета.

Randal Schwartz, Tom Christiansen «Learning Perl»[2].

Практическое руководство, цель которого помочь вам как можно быстрее начать писать сценарии Perl. Каждую главу сопровождают упражнения (с полными решениями). Большая новая глава знакомит вас с CGI-программированием, затрагивается также использование библиотечных модулей, ссылок и объектно-ориентированных конструкций в Perl.

Larry Wall, Tom Christiansen, Randal Schwartz «Programming Perl»[3].

Авторитетное руководство по Perl 5 – средству создания сценариев, ставшему предпочтительным средством программирования в World Wide Web, системном администрировании Unix и для большого числа других задач. Одним из авторов книги является Ларри Уолл (Larry Wall), создатель Perl.

Tom Christiansen, Nathan Torkington «The Perl Cookbook»[4].

Всеобъемлющее собрание задач, решений и практических примеров для всех, программирующих на Perl. Охватывает темы от вопросов, возникающих у новичков, до приемов, поучительных даже для опытных программистов на Perl. Нечто большее, чем простое собрание советов и ловких приемов, «The Perl Cookbook» является долгожданным дополнением к «Programming Perl», содержащим ранее не публиковавшиеся тайны Perl.

---

[1] К. Дейт «Введение в системы баз данных» 6 издание, изд-во «Диалектика».

[2] Р. Шварц, Т. Кристиансен «Изучаем Perl», изд-во «BHV-Киев».

[3] Л. Уолл, Т. Кристиансен, Р. Шварц «Программирование на Perl» 3 издание, изд-во «Символ-Плюс», февраль 2001 г.

[4] Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста», изд-во «Питер».

Lincoln Stein, Doug MacEachern «Writing Apache Modules with Perl and C».

В этой книге рассказывается, как расширить возможности вашего веб-сервера Apache, независимо от того, используете ли вы в качестве языка программирования Perl или C. В книге излагается архитектура Apache, mod_perl и Apache API. Что касается DBI, то в книге описывается модуль `Apache::DBI`, предоставляющий дополнительные развитые функции DBI для веб-сервисов, таких как создание пула постоянных соединений, оптимизированного для работы с базами данных через Сеть.

«Boutell FAQ» (*http://www.boutell.com/faq/*) и другие.

Эти ссылки имеют большую ценность для тех, кто собирается развертывать веб-сайты под управлением DBI. Обсуждаются общие вопросы, касающиеся того, как надо и как не надо реализовывать CGI-программы.

Randy Jay Yarger, George Reese, Tim King «MySQL & mSQL»[1].

Очень полезная книга для пользователей баз данных MySQL и mSQL. В ней рассказывается не только о самих этих базах данных, но и о драйверах DBI, а также других полезных темах, таких как CGI-программирование.

Типографские обозначения

В данной книге принято следующее использование шрифтов:

`Моноширинный`	Используется в названиях методов, в именах функций, переменных и атрибутов, а также в примерах исходных текстов программ.
Курсив	Используется в именах файлов и машин, в URL, а также для обозначения новых терминов.

Как с нами связаться

Мы опробовали и проверили весь материал книги, насколько было в наших силах, но вы можете столкнуться с тем, что какие-то функции изменились или в издание вкрались ошибки. Просим сообщить о замеченных вами ошибках и предложениях для будущих изданий по адресу:

O'Reilly & Associates

101 Morris Street

[1] Р. Яргер, Дж. Риз, Т. Кинг «MySQL и mSQL», изд-во «Символ-Плюс».

Sebastopol, CA 95472

800-998-9938 (из США или Канады)

707-829-0515 (международные или местные)

707-829-0104 (FAX)

Вы можете также послать нам сообщение электронной почтой. Чтобы быть включенным в наш лист почтовой рассылки или запросить каталог, посылайте письма по адресу:

info@oreilly.com

Вы можете задать технический вопрос или сообщить свои комментарии по поводу этой книги по адресу:

bookquestions@oreilly.com

У данной книги существует веб-сайт, где представлены примеры, ошибки и планы будущих изданий. Эту страницу можно найти на:

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

Дополнительные сведения об этой и других книгах можно найти на сервере O'Reilly:

http://www.oreilly.com

Примеры программ

Вы можете копировать тексты программ из книги и изменять их для собственных нужд. Однако вместо копирования вручную лучше загрузить код с *http://www.oreilly.com/catalog/perldbi*.

Благодарности

Аллигатор хочет поблагодарить свою жену Каролин за терпение к его авторской аффектации и метаниям во время написания книги. Имя Мартина Маккарти (Martin McCarthy) тоже должно быть упомянуто с благодарностью за многочисленные читки корректуры ранних набросков книги. Фил Кайзер (Phil Kizer) заслуживает благодарности за работу серверов, на которых сайт DBI располагался с 1995 до начала 1999 года. Карин и Джон Атвуд (Karin and John Attwood), Энди Бернхэм (Andy Burnham), Энди Норфолк (Andy Norfolk), Крис Твид (Chris Tweed) и многие другие члены списка рассылки stones заслуживают благодарности (и пива) за содействие по сохранению и представлению многих мегалитических площадок на территории Соединенного Королевства. Дополнительная благодарность всем, стоящим за ASLaN, за добровольное участие в трудной работе и хорошее ее исполнение.

Линда Мюи (Linda Mui) решительно заслуживает фирменного пакета O'Reilly и пары старых солнечных очков за изумительную работу по редактированию этой книги. И, наконец, с огромным чувством признательности я обращаюсь к Тиму, благодаря которому книга стала намного лучше, чем если бы я писал ее один.

Тим хотел бы поблагодарить свою жену Мари за то, что она его жена; Ларри Уолла за то, что он дал миру Perl; Теда Лемона (Ted Lemon) за идею, которая много лет спустя превратилась в DBI, и за долгие годы ведения списка рассылки. Спасибо также Тиму О'Рейлли (Tim O'Reilly) за то, что он извел меня требованиями написать книгу о DBI, и Аллигатору за то, что он фактически начал эту работу, а затем позволил мне присоединиться к ней (смирившись с моей склонностью к педантизму), и Линде Мюи за великолепное редактирование.

У DBI долгая история,[1] бесчисленное количество людей внесло свой вклад в его обсуждение и разработку за эти годы. Прежде всего, хотелось бы поблагодарить первопроходцев, в том числе Кевина Стока (Kevin Stock), Баццо Мочетти (Buzz Moschetti), Курта Андерсена (Kurt Andersen), Уильяма Хайлса (William Hails), Гарта Кеннеди (Garth Kennedy), Майкла Пепплера (Michael Peppler), Нейла Бриско (Neil Briscoe), Дэвида Хьюза (David Hughes), Джеффа Стэндера (Jeff Stander) и Форреста Д. Уитчера (Forrest D. Whitcher).

И, конечно, нужно поблагодарить тех несчастных, которые прорывались сквозь бессчетные недокументированные препятствия, чтобы реализовать на практике драйверы DBI. В их рядах находятся Йохан Видман (Jochen Wiedmann), Аллигатор Декарт (Alligator Descartes), Джонатан Леффлер (Jonathan Leffler), Джефф Арлвин (Jeff Urlwin), Майкл Пепплер (Michael Peppler), Хенрик Тугар (Henrik Tougaard), Эдвин Пратомо (Edwin Pratomo), Дэвид Майглваккей (Davide Migliavacca), Ян Паздзиора (Jan Pazdziora), Питер Хэворс (Peter Haworth), Эдмунд Мергл (Edmund Mergl), Стив Уильямс (Steve Williams), Томас Лауэри (Thomas Lowery) и Филип Пламли (Phlip Plumlee). Без них DBI не смог бы стать сегодня реальностью.

Оба мы хотим поблагодарить многочисленных рецензентов, сделавших ценные замечания. Особая благодарность Мэтью Персико (Matthew Persico), Натану Торкингтону (Nathan Torkington), Джеффу Роуи (Jeff Rowe), Деннису Годдарду (Denis Goddard), Хонцу Паздзиора (Honza Pazdziora), Ричу Миллеру (Rich Miller), Ниаму Кеннеди (Niamh Kennedy), Рэнделу Шварцу (Randal Schwartz) и Джеффри Бэйкеру (Jeffrey Baker).

[1] Все это началось 29 сентября 1992 года.

1

Введение

«Базы данных» – большая и сложная тема, охватывающая множество различных понятий структуры, вида и предполагаемого использования. Существует также множество разнообразных способов доступа к данным, хранимым в этих базах, и воздействия на них.

В этой книге описывается интерфейс баз данных для Perl, называемый DBI и предоставляющий унифицированный интерфейс для доступа к данным, хранящимся во многих различных базах. DBI позволяет писать на Perl программы, осуществляющие доступ к данным, без необходимости учитывать специфические для базы данных или платформы вопросы или фирменные интерфейсы.

Мы также рассмотрим способы хранения, извлечения и обработки данных в Perl без применения DBI, поскольку нередки ситуации, когда использование базы данных не является насущной необходимостью, но, тем не менее, требуется некая форма структурированного хранения данных.

Для начала мы обсудим некоторые наиболее частые применения систем баз данных в деловой жизни и место, которое занимают Perl и DBI в этих рамках.

От мэйнфреймов к рабочим станциям

В настоящее время базы данных повсеместно используются в вычислительной практике. В прежние годы они почти исключительно использовались в вычислительных средах мэйнфреймов. Сейчас, когда компьютеры размером с коробку для обуви обладают большей вычислительной мощностью, чем прежние машины, занимавшие целую

комнату, высокопроизводительная обработка баз данных доступна каждому.

Помимо более дешевых и более мощных аппаратных устройств стали доступны меньшие пакеты баз данных, такие как Microsoft Access и mSQL. Эти пакеты позволяют любому пользователю компьютера применять мощную технологию баз данных в повседневной жизни.

В корпоративной рабочей среде также наблюдается резкая децентрализация ресурсов баз данных с коренным уменьшением масштабов операций в некоторых фирмах, в ходе которой производится замена баз данных на мэйнфреймах набором небольших баз данных, распределенных по персональным компьютерам и рабочим станциям. В результате разработчики и пользователи часто становятся ответственными за администрирование и поддержку собственных баз и наборов данных.

Тенденция к смешению и согласованию технологий баз данных имеет целый ряд существенных недостатков. Так, заменив централизованную базу данных кластером рабочих станций и несколькими типами баз данных, компании столкнулись с задачей найма новых квалифицированных администраторов или переобучения уже имеющихся. Кроме того, администраторам приходится теперь изучать, как объединять вместе различные типы баз данных.

В этой обстановке возник новый тип техники программного обеспечения, а именно – *независимые от баз данных* программные интерфейсы. Если уменьшение масштабов создало проблемы для административного персонала, то еще большее влияние оно, возможно, оказало на разработчиков.

В централизованной среде мэйнфрейма предполагается, что программное обеспечение баз данных пишется на стандартном языке, возможно, COBOL или C, и выполняется только на одной машине. В распределенной же среде могут поддерживаться несколько баз данных с различными операционными системами и процессорами, причем каждая команда разработчиков выбирает собственную предпочтительную среду разработки (такую как Visual Basic, PowerBuilder, Oracle Pro*C, Informix E/SQL, C++ с ODBC – список почти бесконечен). Поэтому задача координации и переноса программного обеспечения быстро превратилась из достаточно простой в крайне сложную.

Независимые от баз данных программные интерфейсы пришли на помощь несчастным осажденным разработчикам, снабдив их единым унифицированным интерфейсом, с помощью которого они могут программировать. Они ограждают разработчиков от необходимости изучения каждого типа баз данных, с которым они работают, и позволяют с большей легкостью переносить программное обеспечение с одного типа баз данных на другой. Например, программное обеспечение, написанное первоначально для базы данных на мэйнфрейме, с небольшими модификациями сможет работать с базой данных Oracle. Программное

обеспечение, написанное для Informix, как правило, с небольшими изменениями работает под Oracle. А программное обеспечение, написанное для Microsoft Access, с помощью небольших модификаций будет работать на Sybase.

Если соединить этот независимый от базы данных программный интерфейс с таким языком программирования, как Perl, нейтральным в отношении операционной системы, то появляется перспектива снова получить единую базу программного кода. Как в старые добрые времена, но с большой разницей: теперь вы полностью владеете мощью распределенных баз данных.

Независимые от баз данных программные интерфейсы помогают не только разработчикам. Администраторы могут использовать их для быстрого написания переносимых средств контроля и администрирования баз данных, увеличивая как свою производительность, так и производительность систем и баз данных, за мониторинг которых они отвечают. Этот процесс приводит к лучшей настройке систем и большей их доступности, предоставляя администраторам возможность активной поддержки обслуживаемых систем.

Другая сторона современного стиля ведения корпоративных баз данных касается идеи *хранилищ данных* (*data warehousing*), что означает создание больших хранилищ архивных данных, в которых можно осуществлять просмотр или поиск информации отдельно от активных баз данных. Мощные языки высокого уровня с независимыми от баз данных программными интерфейсами, такие как Perl, приобретают большое значение в создании и сопровождении хранилищ данных. Это связано не только с их способностью без проблем перемещать данные из одной базы в другую, но и с возможностью эффективно просматривать, упорядочивать, преобразовывать и обрабатывать данные.

В целом, базы данных приобретают все большее значение в корпоративной среде, и для того чтобы не допустить раскола этих данных на отдельные местные фрагменты, необходимы мощные интерфейсы. Этот процесс склеивания отдельных кусков может быть облегчен в результате использования программных интерфейсов, независимых от баз данных, таких как DBI, особенно в сочетании с эффективными языками высокого уровня, подобными Perl.

Perl

Perl является языком очень высокого уровня, первоначально разработанным в 1980-х годах Ларри Уоллом (Larry Wall). Сейчас Perl разрабатывается под бдительным оком Ларри группой людей, известных как Perl5-Porters. Одной из многих мощных возможностей Perl является способность обрабатывать произвольные фрагменты текстовых данных, известных как *strings* (строки), разнообразными способами, включая обработку регулярных выражений. Эта возможность де-

лает Perl отличным выбором для программирования баз данных, поскольку большая часть хранящейся в базах данных информации имеет текстовую природу. В отличие от C, не очень хорошо приспособленного для обработки строк, Perl устраняет из процесса программирования эти трудности. Сценарии на Perl оказываются значительно меньше, чем эквивалентные программы на C, и обычно переносимы на другие операционные системы, где программы Perl выполняются после небольших модификаций или вообще без них.

В настоящее время Perl получил возможность динамической загрузки внешних *модулей*, представляющих собой части программного обеспечения, которые могут быть вставлены в Perl для расширения его функциональности. Сейчас таких модулей буквально сотни, от математических до модулей передачи трехмерной графики и модулей, позволяющих взаимодействовать с сетями и сетевым программным обеспечением. DBI является набором модулей для Perl, позволяющих взаимодействовать с базами данных.

В последние годы использование Perl превратилось в стандарт для многих фирм благодаря исключительной полезности его во многих различных приложениях, как своего рода «швейцарского солдатского ножа» языков программирования. Он активно используется системными администраторами, которым нравится его гибкость и пригодность для почти любых мыслимых задач. В сочетании с DBI Perl существенно упрощает загрузку и выгрузку баз данных, а его отличные способности обработки данных позволяют разработчикам с легкостью создавать и обрабатывать данные.

Более того, Perl молчаливо принят de facto как язык для создания CGI-приложений в Сети. Какое отношение это имеет к DBI? С помощью Perl и DBI можно быстро создавать мощные CGI-сценарии, генерирующие динамические веб-страницы из данных, содержащихся в базах данных. Например, каталоги электронной торговли могут храниться в базе данных и представляться покупателям как ряд динамически создаваемых веб-страниц. Примеры программ в этой книге используют базу данных археологических площадок, которую можно разместить в Интернете.

Исходя из такого подтверждения концепции, а также появления новых мощных модулей, таких как DBI и средства быстрой разработки графических интерфейсов пользователя Tk, крупные компании сейчас рассчитывают на то, что Perl обеспечит возможность быстрой разработки эффективных, надежных и переносимых приложений, которые могут быть развернуты в корпоративных сетях интранет и в Интернете.

DBI в реальном мире

Сегодня DBI используется во всем мире многими компаниями, такими как NASA и Motorola, в том числе в крупномасштабных критически важных средах. Приведем примеры свидетельств активных пользователей DBI со всего света:

> Мы разработали для одного своего крупного клиента крупномасштабную систему регистрации и анализа телефонных звонков и поддерживаем ее. Система ежедневно собирает около 1 Гбайт данных по звонкам примерно с 1 200 000 телефонных номеров. К данному времени обработано около 424 Гбайт данных (свыше 6 200 000 000 звонков). Данные обрабатываются и загружаются в Oracle с помощью DBI и DBD::Oracle. В базе данных содержатся сведения примерно о 20 миллионах звонков, которые можно просмотреть. Система создает ежемесячно около 44 000 высококачественных постскриптовских отчетов (около пяти страниц с одиннадцатью цветными графиками и пятью таблицами), получаемых заполнением шаблонов FrameMaker с помощью Perl. (Цифры верны на июль 1999 года и неуклонно растут.)
>
> Система развернута на трех двухпроцессорных машинах Sun SPARC Ultra 2. Одна используется для получения и обработки данных, на второй установлена Oracle, а третья осуществляет главную работу по генерации отчетов, распределенную также и по двум другим машинам. Система почти полностью реализована на Perl.
>
> Имеется лишь одна программа, написанная не на Perl, и то только потому, что она была написана раньше, и не является специфичной для системы. Прочий код, написанный не на Perl, состоит из нескольких небольших библиотек, подключенных к Perl через интерфейс XS.
>
> Цитата из подведения итогов проекта старшим управляющим: «Менее чем через год служба заработала. Это событие было отмечено как один из наиболее быстро прошедших от замысла до запуска проектов такого размера и сложности.»
>
> Система спроектирована, разработана, реализована, инсталлирована и поддерживается Paul Ingram Group, получившей за участие в проекте награду «Rising to the Challenge» («Принявший вызов»). Без Perl система не смогла бы быть разработана в течение требуемых очень сжатых сроков. Опять же, без Perl систему нельзя было бы так легко сопровождать или быстро расширять для удовлетворения изменяющихся требований.
>
> *Тим Банс (Tim Bunce), Paul Ingram Group*

> В 1997 году я разработал для Исследовательского центра NASA в Лэнгли, штат Виргиния, систему, предоставляющую интерфейс для поиска из Сети данных в базе, содержащей сведения о примерно 100 000 элементов оборудования, принадлежащих NASA. Я использовал Apache, DBI, Informix, WDB и mod_perl на Sparc 20. Работала, как в сказке. Она так понравилась в NASA, что ее использовали для демонстраций на совещаниях по реорганизации аэродинамических труб. Дело обстояло таким образом, что каждый раз после показа системы другим людям мне приходилось расширять ее, добавляя вещи, подобные отслежива-

нию ремонтируемого оборудования или выводу графических изображений оборудования, так что когда они теряли листы спецификации, то могли посмотреть их электронный вариант. Когда все работает, успех способствует успеху.

Джефф Роуи (Jeff Rowe)

Я работаю над системой, реализованной с помощью Perl, DBI, Apache (mod_perl) на машине с Red Hat Linux 5.1, с использованием MySQL – облегченной SQL РСУБД.[1] Система предназначена для работы в крупной международной холдинговой компании, владеющей примерно 50 другими компаниями. В компании занято около 30 000 служащих по всему миру, которым нужна надежная система для доступа к веб-ресурсам. Этот первый цикл интранет рассчитан на обработку до сорока запросов веб-объектов в секунду (примерно 200 одновременно работающих пользователей) и работает на однопроцессорном Intel Pentium Pro с 512 Мбайт памяти. Мы ведем разработку на Perl, всюду используя объектно-ориентированную технологию. За последнюю пару лет мы разработали большую библиотеку повторно используемого кода на Perl. Один из наших наиболее полезных модулей создает объектно-реляционную оболочку вокруг DBI, что позволяет нашим разработчикам приложений обращаться к базе данных с использованием объектно-ориентированных методов для доступа к записям или изменения их свойств. Мы сэкономили несчетное количество времени и денег, выбрав Perl вместо более частной системы.

Джесси Эрлбаум (Jesse Erlbaum)

Motorola Commercial Government and Industrial Systems использует Perl вместе с DBI и DBD-Oracle как часть ориентированной на Интернет системы отчетности для большого числа производящих и распределяющих организаций. Использование DBI/DBD-Oracle является составной частью отхода от систем отчетности, базирующихся на Oracle Forms, к платформе, базирующейся чисто на веб. В эксплуатации находится несколько приложений среднего размера, основанных на DBI, включая простые приложения распространения извещений, динамическую маршрутизацию подтверждений и важные бизнес-приложения. Хотя требуется несколько большее «терпение» для разработки веб-приложений и «хорошо выглядящих» пользовательских интерфейсов, мой опыт показывает, что для реализации приложений с использованием DBI требуется несколько меньшее время, чем при альтернативных вариантах. Сроки «исправления» использующих DBI/DBD программ также оказываются короче. Качество программ при использовании подхода DBI/DBD оказывается выше, но это может объясняться различиями в методологии разработки программного обеспечения.

Гарт Кеннеди (Garth Kennedy), Motorola

[1] РСУБД – система управления реляционными базами данных.

Историческое отступление и стоящие камни

По всей книге разбросаны примеры, связанные с обсуждаемыми темами. Для того чтобы не дать примерам сбить вас с толку еще больше, чем это сделает текст, обсудим заранее данные, которыми мы будем манипулировать в примерах.

Преимущественно в Великобритании, но также и в других странах по всему миру есть сооружения из каменных глыб, или *мегалиты*.[1] Камни располагаются кольцами, рядами, одиночно или попарно. Никто не может с полной уверенностью сказать, каково назначение этих сооружений, но существует масса теорий от уклончивого «ритуального» использования до более решительной теории посадочных площадок инопланетян. Наиболее известное и часто посещаемое из этих сооружений – Стоунхендж, расположенный на юге Англии. Однако Стоунхендж является уникальным и нетипичным мегалитическим монументом.

Недостаточное понимание происхождения мегалитов отчасти объясняется тем, что возраст их насчитывает до 5 тысяч лет. Просто не существует письменных источников, описывающих назначение, ритуалы или разумные основания возведения этих сооружений. Однако существует много сайтов, излагающих различные теории их происхождения.

Примеры программ, приведенные в данной книге, и образцы веб-приложений, которые мы также предоставляем, используют базу данных, содержащую сведения об этих сооружениях.

[1] От греческого «большой камень». Для многих сооружений это может быть неудачным названием, поскольку образующие круг камни оказываются не более полуметра в высоту. Однако в некоторых экстремальных случаях, таких как Стоунхендж и Эйвбери, приставка «мега» более чем оправдана.

2

Основные базы данных, не использующие DBI

Существует несколько способов организации данных, содержащихся в базах данных. Наиболее распространенным из них является технология *реляционных баз данных*. Базы данных, использующие реляционную модель, называются *системами управления реляционными базами данных,* или РСУБД. Наиболее популярные в настоящее время системы баз данных (такие как Oracle, Informix и Sybase) являются реляционными по своей архитектуре.

Но что в действительности означает «реляционная»? Реляционной базой данных называется такая база данных, которая воспринимается пользователем как набор таблиц, при этом таблица является неупорядоченным набором строк. (Допуская вольность речи, отношение (relation) является просто математическим термином для такой таблицы.) В каждой строке таблицы содержится фиксированное число полей, и в каждом поле может храниться предопределенный тип данных, например целое, дата или строка.

Другим видом методологии, приобретающим все большее распространение, является объектно-ориентированная методология, или ООСУБД. В объектно-ориентированной модели все, что входит в базу данных, рассматривается как объекты, относящиеся к некоторым классам, внутри которых определены правила обработки заключенных в них данных. Эта методология близко смыкается с объектно-ориентированными языками, такими как Smalltalk, C++ и Java. Однако DBI не поддерживает никакой реально существующей ООСУБД, поэтому в данное время мы не станем углубляться в эту методологию.

Наконец, существует несколько пакетов упрощенных баз данных, поддерживаемых в различных операционных системах. Эти пакеты

простых баз данных, как правило, не обладают теми изощренными функциями, которые предоставляют ядра «настоящих» баз данных. Фактически они являются лишь несколько усложненными программами обработки файлов, а не настоящими пакетами для работы с базами данных. Однако в их пользу говорит то, что они могут работать чрезвычайно быстро, а в некоторых ситуациях усложненные функции, предоставляемые «настоящими» базами данных, являются просто лишней нагрузкой.[1]

В этой главе мы расскажем о некоторых базах данных, с которыми DBI не используется, от простейших файлов данных ASCII до дисковых хэш-файлов с поддержкой дубликатов ключей. Попутно мы обсудим проблемы одновременного доступа и блокировки и некоторые применения довольно полезных модулей `Storable` и `Data::Dumper`. (Хотя эти темы не связаны с DBI непосредственно, мы полагаем, что их освещение может для многих оказаться полезным, и даже ветераны DBI смогут найти несколько удобных приемов.)

Все эти технологии баз данных, от самых сложных до простейших, имеют два основных свойства. Первое состоит в самом определении термина: база данных – это набор хранимых в компьютере данных, поверх которого находятся различные уровни абстракции. Каждый уровень абстракции обычно упрощает организацию хранимых данных и доступ к ним благодаря тому, что разделяются запрос для получения конкретных данных и механизм их получения.

Вторым основным свойством всех систем баз данных является то, что во всех них для получения доступа к хранимым данным используются интерфейсы прикладного программирования (API). В простейших базах данных API состоит просто из вызова функций операционной системы для чтения/записи файлов посредством вашего любимого языка программирования.

API позволяет программистам взаимодействовать со сложными программными модулями по правилам доступа, определенным создателями исходного программного обеспечения. Хорошим примером этого служит Berkeley Database Manager API. Помимо простого доступа к данным этот API позволяет изменять структуру базы данных и сами хранимые в ней данные. Такой более высокий уровень доступа к базе данных дает то преимущество, что не нужно беспокоиться о том, *как* Berkeley Database Manager управляет данными. Вы работаете с абстрагированным представлением через API.

На более высоких уровнях доступа, таких как реализуемые в РСУБД, API доступа и обработки данных полностью отделен от структуры базы данных. Это разделение логической модели и физического пред-

[1] Полезный список свободно распространяемых баз данных имеется на *ftp://ftp.idiom.com/pub/free-databases*.

ставления позволяет писать стандартный код для работы с базой данных, например на SQL, который независим от используемого ядра базы данных.

Администраторы хранения данных и слои программного обеспечения

Современные базы данных, независимо от реализованной в них методологии, обычно состоят из нескольких слоев программного обеспечения. Каждый слой реализует более высокий уровень функциональности, используя интерфейсы и сервисы, имеющиеся в слоях более низкого уровня.

Например, одноуровневые базы данных состоят из пулов данных и очень немногочисленных слоев абстрагирования. Базы данных такого типа позволяют обрабатывать хранимые в них данные посредством прямого изменения того, как эти данные хранятся в самих файлах. Тем самым достигается большая мощь и гибкость ценой трудности использования, ограниченных функциональных возможностей и нервного напряжения, вызываемого отсутствием средств страховки. Вся работа с файлами данных происходит путем использования стандартных файловых операций Perl, в свою очередь, основанных на API операционной системы.

Файловые библиотеки DBM, такие как Berkeley DB, служат примером слоя администратора хранения данных (storage manager), лежащего поверх файлов чистых данных и позволяющего обрабатывать хранимые в базе данные через четко определенный API. Этот администратор хранения данных транслирует вызовы API в операции обработки данных в интересах пользователя, препятствуя непосредственному изменению пользователем структуры данных, в результате которого данные могли бы оказаться поврежденными и нечитаемыми. Управление базой данных через такого администратора хранения данных значительно проще и безопаснее, чем непосредственное ее изменение.

Можно реализовать поверх файлов DBM более мощную систему базы данных. Этот новый слой использовал бы DBM API для реализации более мощных функций и создал бы еще один уровень абстрагирования между пользователем и реальными физическими файлами, содержащими данные.

Использование администраторов более высокого уровня предоставляет многие преимущества. Уровни абстрагирования между программой пользователя и базой данных позволяют производителям баз данных прозрачным образом улучшать оптимизацию, изменять структуру файлов базы данных или осуществлять перенос ядра базы данных на другие платформы, при котором не возникает необходимости в изменении прикладных программ.

Языки запросов и функции данных

Операции работы с базой данных можно разделить на те, в которых производится воздействие на саму базу данных (т. е. логическую и физическую организацию составляющих ее файлов), и те, в которых производится воздействие на сами данные, хранимые в этих файлах. Первая группа в целом специфична для конкретной базы данных и может быть реализована многими способами, а вторая группа обычно выполняется с использованием *языка запросов*.[1]

Все языки запросов от нижнего уровня использования функций Perl для обработки строк и чисел до языка запросов высокого уровня, такого как SQL, реализуют четыре главные операции, с помощью которых обрабатываются данные:

Выборка

Чаще всего работа с базой данных состоит в извлечении хранящихся в ней данных. Эта операция известна как *выборка* (*fetching*). Она возвращает необходимые данные в виде, понятном для базового языка API, используемого для создания запросов к базе данных. Например, при необходимости использовать Perl для запроса данных из базы Oracle эти данные запрашиваются с использованием языка запросов SQL, а возвращаемые данные должны иметь вид строк и чисел Perl. Эта операция известна также как *выбор* (*selecting*), от ключевого слова SQL `SELECT`, используемого при выборке данных из базы.

Сохранение

Для того чтобы данные могли быть выбраны, сначала должно быть произведено их *сохранение* (*storing*). Слой администратора хранения данных переводит величины, представленные на языке программирования, в такие, которые понятны базе данных. После этого администраторы хранения данных записывают эти величины в файлы данных. Эта операция известна также как *вставка* (*inserting*) данных.

Обновление

Если данные записаны в базу, это не значит, что они остаются неизменными. При необходимости их можно изменить. Например, в базе данных, хранящей данные о товарах, со временем могут изменяться сведения о ценах для каждого продукта. Операция изменения значения данных, уже существующих в базе, называется *обновлением* (*updating*). Важно отметить, что при этой операции к

[1] Мы употребляем термин «язык запросов» в очень широком смысле. Мы охватываем им все – от вербальных командных языков типа SQL до жестко закодированной логики в таких языках программирования, как Perl.

базе данных не добавляются новые элементы и не удаляются старые, но изменяются уже существующие.[1]

Удаление

Последней базовой операцией, которую обычно требуется производить с данными, является *удаление* из базы старых или избыточных данных. При этой операции данные полностью удаляются из базы с помощью администратора хранения, вырезающего данные из файлов. После удаления из базы данные нельзя восстановить или заменить, кроме как заново вставив в базу.[2]

Эти операции часто обозначают акронимом C.R.U.D. (Create, Read, Update, Delete). В данной книге эти темы обсуждаются в несколько ином порядке, в основном потому, что, на наш взгляд, большинство читателей, по крайней мере вначале, будет заниматься извлечением данных из существующих баз, а не созданием новых баз для хранения данных.

Стоящие камни и учебная база данных

Наши небольшие учебные базы данных, встречающиеся в этой главе, содержат данные о мегалитах в Великобритании. В последующих главах используется более сложная версия этой базы данных.

В число основных данных о мегалитах[3], которые мы хотим хранить, входят название, местонахождение в стране, уникальный картографический код, тип мегалита (например, круг камней или стоящий камень) и описание того, как выглядит площадка.

Например, мы хотим сохранить в базе данных следующие сведения о Стоунхендже:

Name:

Стоунхендж.

Location:

Уилтшир, Англия.

Map Reference:

SU 123 400.

---

[1] На логическом уровне это так, но на физическом обновление может осуществляться путем удалений и вставок.

[2] Если только вы не используете *транзакции*. Подробнее о них говорится в главе 6.

[3] Хранение чего-либо *на* мегалите является прямым нарушением принципов, изложенных в приложении C. Это замечание приводится на тот случай, если вы пропустили знакомство с мегалитами, представленное нами в главе 1.

Type:

Каменный круг и хендж.

Description:

Самый знаменитый в мире мегалит, образованный земляным валом, или *хенджем*, и несколькими концентрическими кольцами массивных стоящих камней – *дольменами*.

Из этой учебной базы мы можем извлекать различные сведения, например, потребовать данные о всех мегалитах в Уилтшире или о всех стоящих камнях в Оркни и т. д.

Теперь обсудим простейший вид базы данных, который можно использовать, – *одноуровневую базу данных* или *базу данных в плоском файле*.

Одноуровневые базы данных

Простейшим типом базы данных, которую мы можем создать для работы, является старая проверенная *одноуровневая база данных* (*flat-file database*). Эта база данных является, по существу, файлом или группой файлов, содержащих данные в известном стандартном формате, которые программа просматривает в поисках требуемых данных. Модифицирование данных обычно производится путем обновления находящейся в памяти копии данных, которые содержатся в одном или нескольких файлах, с последующей обратной записью на диск всего набора данных. Одноуровневые базы данных обычно являются текстовыми ASCII-файлами, каждая строка которых содержит одну запись данных. Признак конца строки служит разделителем записей.

В этом разделе мы рассмотрим два основных типа баз данных в плоских файлах: файлы, в которых поля отделяются одно от другого символом-разделителем, и файлы, в которых каждое поле имеет фиксированную длину. Мы обсудим преимущества и недостатки каждого типа файла данных и представим примеры программ для работы с ними.

По-видимому, чаще всего одноуровневые базы данных используют формат файла *с разделителем*, в котором все поля отделяются одно от другого символом-разделителем. И, вероятно, наиболее распространенным из этих форматов является файл с *разделителем-запятой* (*comma-separated values* – CSV), в котором поля отделяются одно от другого запятыми. Этот формат воспринимают многие распространенные программы, такие как Microsoft Access и электронные таблицы. Благодаря этому данный формат является отличным переносимым форматом базового уровня, который удобно использовать при совместном доступе к данным из различных приложений.[1]

Другими часто используемыми символами-разделителями являются двоеточие (:), символ табуляции и символ вертикальной черты (|).

Примером файла с разделителями является файл */etc/passwd* операционной системы Unix, в котором разделителем служит двоеточие. На рис. 2.1 показана одна запись из файла */etc/passwd*.

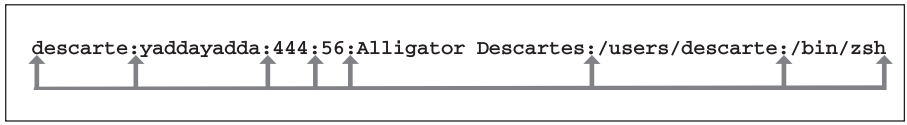

***Рис. 2.1.** Формат записи файла /etc/passwd*

Запросы данных

Поскольку файлы с разделителем относятся к очень низкому уровню администратора хранения данных, все действия с данными должны производиться при помощи функций операционной системы и очень низкоуровневой логики запросов, такой как базовые операции сравнения строк. В следующей программе продемонстрированы открытие файла с разделителем-двоеточием, содержащего данные о мегалитах, поиск нужной площадки и возврат найденных данных:

```
#!/usr/bin/perl -w
#
# ch02/scanmegadata/scanmegadata: Просматривает заданный файл с данными
#                                 по мегалитам в поисках нужной площадки.
#                                Использует двоеточие для разделения полей.
#

### Проверить, задал ли пользователь следующие аргументы:
###     1) Имя файла, содержащего данные
###     2) Название площадки, которую нужно найти
die "Использование: scanmegadata <файл данных> <название площадки>\n"
    unless @ARGV == 2;

my $megalithFile = $ARGV[0];
my $siteName     = $ARGV[1];

### Открытие файла данных для чтения или аварийный выход
open MEGADATA, "<$megalithFile"
    or die "Невозможно открыть $megalithFile: $!\n";

### Объявить переменные полей в строке
my ( $name, $location, $mapref, $type, $description );

### Объявить флаг 'запись найдена'
```

[1] Еще более замечательно, что есть драйвер DBI под названием DBD::CSV, позволяющий писать SQL-команды для обработки данных в плоских файлах формата CSV.

```
my $found;

### Просмотреть все записи в поиске нужной площадки
while ( <MEGADATA> ) {

    ### Удалить символ новой строки, являющийся разделителем записей
    chop;

    ### Разбить запись на отдельные поля
    ( $name, $location, $mapref, $type, $description ) =
         split( /:/, $_ );

    ### Сравнить название площадки с содержащимся в записи
    if ( $name eq $siteName ) {
         $found = $.;  # $. содержит номер текущей строки файла
         last;
    }
}

### Если найдена нужная площадка, вывести данные о ней
if ( $found ) {
     print "Найдена площадка: $name в строке $found\n\n";
     print "Данные по площадке $name ( $type )\n";
     print "===============",
         ( "=" x ( length($name) + length($type) + 5 ) ), "\n";
     print "Местонахождение:      $location\n";
     print "Код карты: $mapref\n";
     print "Описание:   $description\n";
}

### Закрыть файл с данными по мегалитам
close MEGADATA;

exit;
```

Например, если запустить эту программу с файлом, содержащим запись следующего вида:[1]

```
Стоунхендж:Уилтшир:SU 123 400:Каменный круг и хендж:Самый известный
каменный круг
```

и искать термин `Стоунхендж`**, то будут выведены следующие данные:**

```
Найдена площадка: Стоунхендж в строке 1

Данные по площадке Стоунхендж (Каменный круг и хендж)
==================================================
Местонахождение: Уилтшир
```

[1] **В этом примере, как и в нескольких последующих, одна строка разбита на две только для того, чтобы уместиться на странице книги.**

```
Код карты:      SU 123 400
Описание:       Самый известный каменный круг
```

указывающие на то, что наш поиск площадки методом грубой силы работает. Пример программы показывает, что мы использовали собственные функции Perl файлового ввода/вывода и собственные функции Perl обработки строк для расщепления данных с разделителем и проверки их на соответствие критерию поиска.

Недостаток использования файлов с разделителем заключается в том, что если символ-разделитель содержится в самих данных, необходимо проявлять осторожность, чтобы не разбить запись в неверном месте. Использование функции Perl `split()` с простым регулярным выражением в том виде, как это сделано в примере, не принимает этого во внимание и может привести к неверным результатам. Например, следующая запись приведет к тому, что при использовании `split()` мы получим не то, что требуется:

```
Стоунхендж:Уилтшир:SU 123 400:Каменный круг и хендж:Стоунхендж: Самый
известный каменный круг
```

Простейшим способом наскоро исправить положение является замена символа-разделителя, содержащегося в данных, каким-либо другим символом, относительно которого вы уверены, что в данных он встретиться не может. При этом нужно не забыть сделать обратное преобразование при выборке данных.

Другим распространенным способом хранения данных в плоских файлах является использование *записей фиксированной длины*. Это подразумевает помещение каждого элемента данных в поле точно определенного размера в файле данных. При таком формате не требуется использование символа-разделителя между полями. Не требуется также и разделять записи между собой, но мы продолжим использовать в наших примерах символ конца строки для разграничения записей, поскольку в Perl очень легко обрабатывать записи построчно.

Использование полей фиксированной длины напоминает организацию данных в более мощных системах баз данных, таких как РСУБД. Благодаря предварительному выделению пространства для данных администратор хранения данных может делать предположения о размещении данных на диске и производить соответствующую оптимизацию. Для наших мегалитических данных можно установить такие размеры полей:[1]

[1] Тот факт, что каждый размер является степенью двух, имеет лишь то значение, что указывает на почтенный возраст авторов, которые помнят те времена, когда использование степени двух в некоторых случаях оказывалось полезным. Как правило, это больше не является существенным.

```
Поле            Размер в байтах
-----           ---------------
Name            64
Location        64
Map Reference   16
Type            32
Description     256
```

Для хранения данных в таком формате администратор хранения должен использовать несколько иную логику, хотя стандартные функции файлового ввода/вывода Perl по-прежнему можно применять. Для проверки наличия в записи нужных данных необходимо реализовать иной способ извлечения полей. Для файлов с полями фиксированной длины отлично подходит функция Perl `unpack()`. В следующем фрагменте программного кода показано, как функция `unpack()` заменяет использовавшуюся выше `split()`:

```
### Разбить запись на отдельные поля
### с учетом перечисленных выше размеров данных
( $name, $location, $mapref, $type, $description ) =
    unpack( "A64 A64 A16 A32 A256", $_ );
```

Поля фиксированной длины имеют постоянный размер, но помещаемые в них данные могут иметь размер, не совпадающий с размером поля. В этом случае свободное пространство заполняется символом, который обычно в данных не встречается или который можно проигнорировать. Обычно это символ пробела (ASCII 32) или `nul` (ASCII 0).

В нашем примере считается, что данные упакованы с дополнением пробелами, поэтому мы удаляем концевые пробелы из поля, чтобы они не мешали поиску. Это можно сделать, используя в `unpack()` формат с `A` в верхнем регистре.

Если приходится выбирать между полями с символом-разделителем и полями с фиксированной длиной, нужно учитывать следующее:

Основные ограничения

Главное ограничение при использовании символа-разделителя заключается в необходимости добавления специальной обработки символов конца записи и границы полей в данных, чтобы избежать их попадания в значение поля.

Главным ограничением полей фиксированной длины является фиксированная длина. Нужно проверять длину записываемых значений, чтобы она не оказалась слишком велика (или разрешить молчаливое усечение этих значений). Если нужно увеличить размер поля, приходится создавать специальную утилиту для переписывания файла в новом формате, а также просматривать и исправлять все сценарии, непосредственно работающие с файлом.

Занимаемое пространство

Файл с использованием разделителей часто занимает меньше места, чем файл с записями фиксированной длины, хранящий те же данные, причем иногда *значительно меньше*. Все зависит от размера пустых или частично заполненных полей и их количества. Например, некоторые значения полей, такие как URL в Интернете, могут быть очень длинными, но обычно они очень коротки. При хранении их в длинных полях фиксированного размера напрасно расходуется большое количество дискового пространства.

Хотя файлы с разделителями часто занимают меньше места, они все же непроизводительно тратят пространство на все символы-разделители. Если нужно хранить большое число очень маленьких полей, то предпочтение может быть отдано записям фиксированной длины.

Быстродействие

В наши дни вычислительные мощности растут быстрее, чем скорости обмена данными с жесткими дисками. Иными словами, иногда выгоднее использовать более эффективные методы хранения данных, даже если это приводит к большим затратам процессорного времени.

Обычно для последовательного доступа лучше использовать файлы с разделителями, чем файлы с записями фиксированной длины, поскольку уменьшение размера более чем компенсирует увеличение расхода процессорного времени, вызванное необходимостью обработки замененных в тексте символов-разделителей.

Однако у файлов с записями фиксированной длины есть свой козырь: прямой доступ. При желании выбрать из файла с разделителями запись номер 42927 вам *необходимо* читать весь файл и вести подсчет записей, пока вы не доберетесь до нужной. При использовании файла с записями фиксированной длины можно просто умножить 42927 на суммарную длину записи и перейти на нее за один шаг с помощью функции `seek()`.

Более того, если запись найдена, ее можно обновить прямо на месте, записав в нее новые данные. Поскольку новая запись имеет тот же размер, что и прежняя, нет опасности повредить следующую запись.

Вставка данных

Вставлять данные в одноуровневую базу очень просто, и обычно для этого требуется только присоединить новые данные к концу файла. Например, вставка новой записи о мегалите в файл с разделителем-двоеточием может быть осуществлена так:

```
#!/usr/bin/perl -w
#
# ch02/insertmegadata/insertmegadata: Вставляет новую запись в файл
#                                     с данными о мегалитах, используя
#                                     двоеточие в качестве разделителя
#

### Проверить, задал ли пользователь аргументы
###     1) Имя файла, содержащего данные
###     2) Название площадки, данные о которой нужно вставить
###     3) Местонахождение площадки
###     4) Картографический код площадки
###     5) Тип площадки
###     6) Описание площадки
die "Использование: insertmegadata"
    ." <файл данных> <название площадки > <местонахождение> <код карты>
    <тип> <описание>\n"
    unless @ARGV == 6;

my $megalithFile    = $ARGV[0];
my $siteName        = $ARGV[1];
my $siteLocation    = $ARGV[2];
my $siteMapRef      = $ARGV[3];
my $siteType        = $ARGV[4];
my $siteDescription = $ARGV[5];

### Открытие файла данных для дополнения или аварийный выход
open MEGADATA, ">>$megalithFile"
    or die "Невозможно открыть $megalithFile для дополнения: $!\n";

### Создать новую запись
my $record = join( ":", $siteName, $siteLocation, $siteMapRef,
                        $siteType, $siteDescription );

### Вставить новую запись в файл
print MEGADATA "$record\n"
    or die "Ошибка записи в $megalithFile: $!\n";

### Закрыть файл данных по мегалитам
close MEGADATA
    or die "Ошибка закрытия $megalithFile: $!";

print "Вставлена запись по площадке: $siteName\n";

exit;
```

В этом примере просто открывается файл данных в режиме дополнения, и в открытый файл пишется новая запись. Как ни прост этот процесс, у него есть потенциальный недостаток. Эта одноуровневая база данных не обнаруживает вставки нескольких элементов данных с одинаковым ключом для поиска. Это значит, что если бы мы захотели вставить еще одну запись о Стоунхендже, то программа охотно позво-

лила бы нам это сделать, несмотря на то что запись для Стоунхенджа уже существует.

Это может составить проблему с точки зрения целостности данных. Прежде чем дописывать данные, стоило бы осуществить проверку того, что при этом не образуется дубликатов данных. Простым подходом будет объединение программы вставки и программы запроса данных.

Другим потенциальным (и более важным) недостатком является то, что система не сможет надежно обрабатывать те случаи, когда несколько пользователей пытаются добавить данные в базу. Поскольку такие проблемы возникают также при обновлении и удалении данных, мы более подробно обсудим эту тему в одном из последующих разделов этой главы.

Вставлять новые записи в файл с записями фиксированной длины просто. Вместо вывода каждого поля и символа-разделителя в описатель файла Perl для создания из данных записи фиксированной длины можно использовать функцию `pack()`.

Обновление данных

Обновление данных в плоских файлах немного сложнее. При запросе данных из базы мы просто последовательно просматривали базу данных, пока не находили нужную запись. Аналогично при вставке данных мы просто присоединяли новые данные, не имея представления о том, какие данные уже хранятся в базе.

Главная проблема при обновлении данных состоит в том, что должна быть возможность прочесть данные из файла, повозиться с ними некоторое время и затем отправить обратно в файл, не потеряв при этом записей.

Один из подходов состоит в том, чтобы загрузить в память всю базу, провести все обновления в копии базы, находящейся в памяти, а затем сбросить все обратно на диск. Второй подход заключается в том, чтобы поочередно читать из базы данных записи, делать в отдельной записи необходимые изменения и затем сразу вносить запись во временный файл. После того как обработаны все записи, можно заменить исходный файл данных временным файлом. Оба приема жизнеспособны, но мы предпочитаем второй по соображениям производительности: загрузка больших баз данных в память может быть очень ресурсоемкой.

В следующей короткой программе реализована вторая из этих стратегий, для того чтобы обновлять коды карт в базе данных с разделителями записей:

```
#!/usr/bin/perl -w
#
# ch02/updatemegadata/updatemegadata: Обновляет файл с данными о
#                                     мегалитах по указанной площадке.
```

```
#                                        Использует данные с разделителем-
#                                        двоеточием и обновляет поле
#                                         кода карты.

### Проверить, задал ли пользователь аргументы для поиска:
###     1) Имя файла, содержащего данные
###     2) Название искомой площадки
###     3) Новый код карты
die "Использование: updatemegadata <файл данных> <название площадки >
    <новый код карты>\n"
    unless @ARGV == 3;

my $megalithFile = $ARGV[0];
my $siteName     = $ARGV[1];
my $siteMapRef   = $ARGV[2];
my $tempFile     = "tmp.$$";

### Открытие файла данных для чтения или аварийный выход
open MEGADATA, "<$megalithFile"
    or die "Невозможно открыть $megalithFile: $!\n";

### Открыть временный файл с данными мегалитов на запись
open TMPMEGADATA, ">$tempFile"
    or die " Невозможно открыть временный файл $tempFile: $!\n";

### Просмотреть записи и найти нужную площадку
while ( <MEGADATA> ) {

    ### Быстрая предварительная проверка для максимальной скорости:
    ### Пропустить запись, если не то название площадки
    next unless m/^\Q$siteName:/;

    ### Разбить данные записи на отдельные поля
    ### (пусть $description содержит символ новой строки)
    my ( $name, $location, $mapref, $type, $description ) =
         split( /:/, $_ );

    ### Пропустить запись, если название площадки не соответствует.
    ### (Излишне ввиду проведенной выше проверки, но сохранено
    ### для совместимости с другими примерами.)
    next unless $siteName eq $name;

    ### Запись для обновления найдена, обновим значение кода карты
    $mapref = $siteMapRef;

    ### Соберем обновленную запись
    $_ = join( ":", $name, $location, $mapref, $type, $description );

}
continue {
```

```
### Запишем данные во временный файл
   print TMPMEGADATA $_
      or die "Ошибка записи в $tempFile: $!\n";
}

### Закроем входной файл с мегалитическими данными
close MEGADATA;

### Закроем временный выходной файл с мегалитическими данными
close TMPMEGADATA
   or die "Ошибка при закрытии $tempFile: $!\n";

### Теперь мы "зафиксируем" изменения, удалив старый файл...
unlink $megalithFile
   or die "Невозможно удалить прежний файл $megalithFile: $!\n";

### и переименовав новый, чтобы он заменил прежний.
rename $tempFile, $megalithFile
   or die "Невозможно переименовать '$tempFile' в '$megalithFile': $!\n";

exit 0;
```

Как видите, в этом примере мы поиграли в Perl мускулами, использовав для упрощения алгоритма цикл `while ... continue` и добавив для увеличения скорости предварительную проверку.

Эквивалентная программа, которая может работать с файлом записей фиксированной длины, очень похожа, но для изменения содержимого поля используется более быстрое обновление по месту. Принцип аналогичен описанному выше запросу с обработкой по месту: необязательно распаковывать и упаковывать все поля записи, достаточно просто обновить соответствующую часть каждой записи. Например:

```
### Просмотр всех записей в поисках нужной площадки
while ( <MEGADATA> ) {

    ### Предварительная проверка с целью ускорения:
    ### Пропустить запись, если в начале ее нет нужного названия
    next unless m/^\Q$siteName/;

    ### Пропустить запись, если извлеченное название площадки
    ### не соответствует требуемому
    next unless unpack( "x64 x64 A16", $_ ) eq $siteName;

    ### Осуществить замену по месту для обновления поля кода карты
    substr( $_, 64+64, 16, pack( "A16", $siteMapRef ) );

}
```

Такой прием действует быстрее, чем упаковка и распаковка каждой записи, хранимой в файле, поскольку выполняется минимальный

объем действий, необходимый для изменения соответствующих значений полей.

Можно заметить, что предварительная проверка в этом примере надежна не на 100 процентов, но этого и не требуется. Она должна перехватить большинство случаев несоответствия, чтобы оправдать себя в результате сокращения числа раз, когда выполняются более дорогостоящие операции распаковки и проверки значения поля. Можно согласиться, что это не самое убедительное применение идеи, но мы еще вернемся к ней в данной главе для более серьезного обсуждения.

Удаление данных

Последний вид действий, которые можно осуществлять над данными в одноуровневой базе, это удаление данных из базы. Мы будем обрабатывать файл по одной записи, пропуская данные через временный файл, как делали это при обновлении, а не заглатывать все данные в память и сбрасывать их на диск в конце.

При такой технологии запись в базе данных скорее опускается, чем фактически удаляется. Каждая запись считывается из файла, проверяется и записывается во временный файл. Когда встречается запись, которую нужно удалить, она просто *не* пишется во временный файл. В результате из базы данных удаляются всякие ее следы, хотя и довольно безыскусным способом.

Следующая программа может быть использована для удаления соответствующей записи из базы данных с разделителями при задании в качестве аргумента названия площадки, которую нужно удалить:

```
#!/usr/bin/perl -w
#
# ch02/deletemegadata/deletemegadata: Удаляет запись для заданной мега-
#                                     литической площадки. Разделителем
#                                     полей служит двоеточие.
#

### Проверить, задал ли пользователь аргументы:
###     1) Имя файла, содержащего данные
###     2) Название площадки, которую нужно удалить
die "Использование: deletemegadata <имя файла с данными> <название
    площадки>\n"
    unless @ARGV == 2;

my $megalithFile  = $ARGV[0];
my $siteName      = $ARGV[1];
my $tempFile      = "tmp.$$";

### Открытие файла с данными на чтение или аварийный выход
open MEGADATA, "<$megalithFile"
```

```
    or die "Невозможно открыть $megalithFile: $!\n";

### Открыть временный файл с мегалитическими данными на запись
open TMPMEGADATA, ">$tempFile"
    or die "Невозможно открыть временный файл $tempFile: $!\n";

### Просмотреть все записи в поиске нужной площадки
while ( <MEGADATA> ) {

    ### Извлечь из записи название площадки (первое поле)
    my ( $name ) = split( /:/, $_ );

    ### Сравнить название площадки с содержащимся в записи
    if ( $siteName eq $name ) {

        ### Если найдена запись, которую нужно удалить, пропустить ее
        ### и перейти к следующей записи
        next;
    }

    ### Внести исходную запись во временный файл
    print TMPMEGADATA $_
        or die "Ошибка записи $tempFile: $!\n";
    }

### Закроем входной файл с мегалитическими данными
close MEGADATA;

### Закроем временный выходной файл с мегалитическими данными
close TMPMEGADATA
    or die "Ошибка закрытия $tempFile: $!\n";

### Теперь "зафиксируем" изменения, удалив старый файл ...
unlink $megalithFile
    or die "Невозможно удалить старый $megalithFile: $!\n";

### и переименовав новый, чтобы он заменил прежний.
rename $tempFile, $megalithFile
     or die "Невозможно переименовать '$tempFile' в '$megalithFile': $!\
n";

exit 0;
```

Программный код для удаления записей из файла с записями фиксированной длины почти идентичен. Единственное изменение касается, как и можно ожидать, кода для извлечения значения поля:

```
### Извлечь из записи название площадки (первое поле)
my ( $name ) = unpack( "A64", $_ );
```

Как и обновление, удаление данных может стать источником проблем, если несколько пользователей пытается одновременно произвести в

данных изменения. Как бороться с этой проблемой, будет рассмотрено далее в этой главе.

Размещение в плоских файлах сложных данных

При обсуждении «плоских файлов» мы пока сохраняли, извлекали и обрабатывали данные только одного, но самого основного типа, – строкового. Как сохранять более сложные данные, такие как списки, хэши или данные большой степени вложенности, использующие ссылки?

Ответ состоит в преобразовании любых типов данных *в* строки. Технически это известно как *маршаллинг* (*marshalling*), или *сериализация* (*serializing*) данных. Список модулей Perl (Perl Module List)[1] содержит раздел, в котором перечислено несколько модулей Perl, осуществляющих пересылку данных.

Мы рассмотрим два самых популярных модуля: `Data::Dumper` и `Storable`, и увидим, как их можно использовать, для того чтобы внести некоторое оживление в наши плоские файлы. Эти приемы можно использовать и при хранении сложных структур данных Perl в реляционных базах с использованием DBI, поэтому будьте внимательны.

Модуль Perl Data::Dumper

Модуль `Data::Dumper` принимает список переменных Perl и записывает их значения *в виде кода Perl,* который при своем выполнении воссоздаст исходные значения, какими бы сложными они ни были.

Этот модуль позволяет легко и быстро сделать дамп состояния программы на Perl в читаемом виде. С его помощью можно также восстановить состояние программы, выполнив код дампа с помощью функции `eval()` или `do()`.

Проще всего показать, как это происходит, приведя несложный пример:

```
#!/usr/bin/perl -w
#
# ch02/marshal/datadumpertest: Создает несколько переменных Perl
#                              и выводит их в дамп. Затем мы сбрасываем
#                              значения переменных и применяем eval
#                              к переменным из дампа ...

use Data::Dumper;
```

[1] Список модулей Perl можно найти на *http://www.perl.com/CPAN/*.

```
### Настроим стиль вывода Data::Dumper
### Подробности в документации по Data::Dumper
if ($ARGV[0] eq 'flat') {
    $Data::Dumper::Indent = 0;
    $Data::Dumper::Useqq  = 1;
}
$Data::Dumper::Purity = 1;

### Создадим несколько переменных Perl
my $megalith  = 'Стоунхендж';
my $districts = [ 'Уилтшир', 'Оркни', 'Дорсет' ];

### Выведем их
print "Начальные значения: \$megalith  = " . $megalith . "\n" .
      "           \$districts = [ ". join(", ", @$districts) . " ]\n\n";

### Создадим из базы данных новый объект Data::Dumper
my $dumper = Data::Dumper->new( [    $megalith, $districts  ],
                                [ qw( megalith  districts ) ] );

### Выведем значения Perl в переменную
my $dumpedValues = $dumper->Dump();

### Покажем, во что Data::Dumper превратил переменные!
print "Код Perl, созданный Data::Dumper:\n";
print $dumpedValues . "\n";

### Присвоим переменным бессмысленные значения
$megalith = 'Blah! Blah!';
$districts = [ 'Alderaan', 'Mordor', 'The Moon' ];

### Выведем бессмысленные значения
print "Бессмысленные значения: \$megalith  = " . $megalith . "\n" .
      "           \$districts = [ ". join(", ", @$districts) . " ]\n\n";

### Загрузим переменные Perl с помощью eval
eval $dumpedValues;
die if $@;

### Выведем загруженные заново значения
print "Загруженные заново значения: \$megalith  = " . $megalith . "\n" .
      "           \$districts = [ ". join(", ", @$districts) . " ]\n\n";

exit;
```

В этом примере инициализируются две переменные Perl и выводятся их значения. Затем создается объект `Data::Dumper` **с этими значениями, изменяются исходные значения и выводятся новые, показывающие, что все без обмана. Наконец, к результату** `$dumper->Dump()` **применяется функция** `eval`**, в результате чего переменным возвращаются их исход-**

ные значения. И снова мы выводим их, чтобы показать, что никакой ловкости рук не используется:

```
Начальные значения: $megalith  = Стоунхендж
                    $districts = [ Уилтшир, Оркни, Дорсет ]

Код Perl, созданный Data::Dumper:
$megalith = 'Стоунхендж';
$districts = [
            'Уилтшир',
            'Оркни',
            'Дорсет'
          ];

Бессмысленные значения: $megalith  = Blah! Blah!
                        $districts = [ Alderaan, Mordor, The Moon ]
Загруженные заново значения: $megalith  = Стоунхендж
                             $districts = [Уилтшир, Оркни, Дорсет]
```

Как использовать `Data::Dumper` для оживления наших плоских файлов? Прежде всего, нужно попросить `Data::Dumper` создать плоскую выдачу, т. е. без символов новой строки. Для этого нужно установить две глобальные переменные в пакете:

```
$Data::Dumper::Indent = 0; # не использовать символы новой строки
                           # для структуризации выдачи
$Data::Dumper::Useqq  = 1; # заключать строки в двойные кавычки
                           # и использовать "\n" в конце строки
```

В нашей программе это можно сделать, передав ей в командной строке аргумент `flat`. В результате в соответствующем месте программа выдаст следующее:

```
$megalith = "Стоунхендж";$districts = ["Уилтшир","Оркни","Дорсет"];
```

Теперь можно модифицировать наши прежние сценарии просмотра (выборки), вставки, обновления и удаления так, чтобы использовать `Data::Dumper` для форматирования записей вместо функций `join()` или `pack()`, которые мы использовали прежде. Для распаковки записей вместо `split()` или `unpack()` будем использовать `eval`.

Вот главный цикл использовавшегося нами ранее сценария обновления (остальная его часть остается неизменной, за исключением добавления вверху строки `use Data::Dumper` и установки переменных `Data::Dumper`, как было описано выше):

```
### Просмотр всех записей в поисках нужной площадки
while ( <MEGADATA> ) {

    ### Быстрая предварительная проверка для максимальной скорости:
    ### Пропустить запись, если не то название площадки
    next unless m/\Q$siteName/;
```

```
### Выполнить eval со строкой записи для установки ссылки
    ### на массив $fields
    my $fields;
    eval $_;
    die if $@;

    ### Разбить запись данных на отдельные поля
    my ( $name, $location, $mapref, $type, $description ) = @$fields;

    ### Пропустить запись, если извлеченное название площадки
    ### не подходит
    next unless $siteName eq $name;

    ### Найдена запись для обновления
    ### Создать новый массив полей с новым значением кода карты
    $fields = [ $name, $location, $siteMapRef, $type, $description ];

    ### Преобразовать в строку команд Perl, кодирующих нашу строку
    ### записи
    $_ = Data::Dumper->new( [ $fields ], [ 'fields' ] )->Dump();
    $_ .= "\n";

}
```

Чего мы этим добились? Нам удалось избежать скучной необходимости явным образом преобразовывать символы-разделители: `Data::Dumper` делает это за нас. Отсутствуют также ограничения, связанные с использованием полей фиксированного размера.

Однако настоящим большим выигрышем стала возможность хранения практически любых сложных структур данных, даже ссылок на объекты. Есть и другие менее заметные выгоды, которые могут оказаться полезными: можно сохранять и восстанавливать неопределенные значения полей (*null*), и нет необходимости определять в каждой записи все поля (вариантные записи).

Отрицательные стороны? Отрицательные стороны есть всегда. В данном случае они, в основном, сводятся к увеличению времени обработки, необходимого как для дампа данных из записи в строки, так и восстановления их функцией `eval`. Есть версия модуля `Data::Dumper`, написанная на C, которая работает значительно быстрее, но, к сожалению, в ней пока не поддерживается переменная `$Useqq`. Для ускорения времени обработки записей в приведенном примере производится быстрая предварительная проверка, согласно которой пропускаются все строки, не содержащие названия площадки в качестве подстроки *хотя бы в одном* месте.

Возникает также проблема, касающаяся защиты данных. Поскольку мы используем `eval` для выполнения кода Perl, содержащегося в нашем файле данных, существует возможность того, что кто-нибудь от-

редактирует файл данных, добавив свой код, который может произвести вредные действия. К счастью, есть средство для борьбы с этим. Можно использовать прагму Perl `ops`, чтобы ограничить действие `eval` компиляцией кода, содержащего только простые объявления. Более подробные сведения можно получить из документации по `ops`, устанавливаемой вместе с Perl:

```
perldoc ops
```

Модуль Storable

Кроме `Data::Dumper` существуют и другие модули пересылки данных, которые вы можете пожелать изучить, в том числе быстрый и эффективный модуль `Storable`.

В следующем программном коде используется тот же подход, который мы продемонстрировали в примере для `Data::Dumper` в основном цикле записи и извлечения:

```
#!/usr/bin/perl -w
#
# ch02/marshal/storabletest: Создать хэш Perl и сохранить его на внешнем
#                            носителе. После этого мы очищаем хэш и
#                            восстанавливаем его из сохраненного.

use Storable qw( freeze thaw );

### Занесем в хэш некоторые значения
my $megalith = {
    'name' => 'Стоунхендж',
    'mapref' => 'SU 123 400',
    'location' => 'Уилтшир',
};

### Выведем их
print "Начальные значения:  megalith = $megalith->{name}\n" .
      "                     mapref   = $megalith->{mapref}\n" .
      "                     location = $megalith->{location}\n\n";

### Запишем значения в строку
my $storedValues = freeze( $megalith );

### Присвоим переменным бессмысленные значения
$megalith = {
    'name' => 'Flibble Flabble',
    'mapref' => 'XX 000 000',
    'location' => 'Saturn',
};

### Выведем бессмысленные значения
```

```
print "Бессмысленные значения:   megalith = $megalith->{name}\n" .
      "                          mapref   = $megalith->{mapref}\n" .
      "                          location = $megalith->{location}\n\n";

### Извлечем значения из строки
$megalith = thaw( $storedValues );

### Выведем заново загруженные значения
print "Заново загруженные значения: megalith = $megalith->{name}\n" .
      "                             mapref   = $megalith->{mapref}\n" .
      "                            location = $megalith->{location}\n\n";

exit;
```

Программа порождает следующий результат, который показывает, что мы надежно сохраняем данные, а затем их извлекаем:

```
Начальные значения:          megalith = Стоунхендж
                             mapref   = SU 123 400
                             location = Уилтшир

Бессмысленные значения:      megalith = Flibble Flabble
                             mapref   = XX 000 000
                             location = Saturn

Заново загруженные значения: megalith = Стоунхендж
                             mapref   = SU 123 400
                             location = Уилтшир
```

В Storable есть функции для записи и чтения структур данных непосредственно в дисковые файлы. Его также можно использовать для кумулятивной записи файла вместо сохранения всех записей в одной атомарной операции.

Пока все это выглядит очень похоже на Data::Dumper, так в чем же разница? Если кратко, то *в скорости*. Storable работает быстро, очень быстро, причем как при сохранении данных, так и при обратном их получении. Скорость достигается отчасти благодаря реализации на C и непосредственной вставке во внутренние механизмы Perl, отчасти благодаря хранению данных в собственном, очень компактном двоичном формате.

Вот еще одна реализация нашей программы обновления, на этот раз с использованием Storable:

```
#!/usr/bin/perl -w
#
# ch02/marshal/update_storable: Обновляет файл с данными о мегалитах
#                               по указанной площадке. Использует данные
#                               Storable и обновляет поле кода карты.

use Storable qw( nfreeze thaw );
```

```
### Проверить, задал ли пользователь аргументы для поиска:
###     1) Имя файла, содержащего данные
###     2) Название искомой площадки
###     3) Новый код карты
die "Использование: updatemegadata <файл данных> <название площадки >
    <новый код карты>\n"
    unless @ARGV == 3;

my $megalithFile = $ARGV[0];
my $siteName     = $ARGV[1];
my $siteMapRef   = $ARGV[2];
my $tempFile     = "tmp.$$";

### Открытие файла данных для чтения или аварийный выход
open MEGADATA, "<$megalithFile"
    or die "Невозможно открыть $megalithFile: $!\n";

### Открыть временный файл с данными мегалитов на запись
open TMPMEGADATA, ">$tempFile"
    or die "Невозможно открыть временный файл $tempFile: $!\n";

### Просмотреть записи и найти нужную площадку
while ( <MEGADATA> ) {

    ### Преобразовать строку в кодировке ASCII обратно в двоичный код
    ### (pack игнорирует символ конца строки – разделитель записей)
    my $frozen = pack "H*", $_;

    ### Разморозить структуру данных
    my $fields = thaw( $frozen );

    ### Разбить данные записи на отдельные поля
    my ( $name, $location, $mapref, $type, $description ) = @$fields;

    ### Пропустить запись, если название площадки не соответствует
    next unless $siteName eq $name;

    ### Запись для обновления найдена
    ### Создать новый массив полей с новым значением кода карты
    $fields = [ $name, $location, $siteMapRef, $type, $description ];

    ### Заморозить структуру данных в двоичной строке
    $frozen = nfreeze( $fields );

    ### Закодировать двоичную строку как читаемый код ASCII
    ### и добавить символ новой строки
    $_ = unpack( "H*", $frozen ) . "\n";

}
continue {
```

```
    ### Вывести запись во временный файл
    print TMPMEGADATA $_
        or die "Error writing $tempFile: $!\n";
}

### Закрыть входной файл с данными по мегалитам
close MEGADATA;

### Закроем временный выходной файл с мегалитическими данными
close TMPMEGADATA
    or die "Ошибка закрытия $tempFile: $!\n";

### Теперь мы "зафиксируем" изменения , удалив старый файл...
unlink $megalithFile
    or die " Невозможно удалить прежний файл $megalithFile: $!\n";

### и переименовав новый, чтобы он заменил прежний.
rename $tempFile, $megalithFile
    or die " Невозможно переименовать '$tempFile' в '$megalithFile': $!\
n";

exit 0;
```

Поскольку формат `Storable` является двоичным, мы не можем непосредственно производить запись в наш плоский файл. Наш символ-разделитель записей (`"\n"`) может появиться среди двоичных данных и привести к искажению файла. Мы обошли эту проблему путем кодирования двоичных данных как строк, состоящих из пар шестнадцатеричных цифр.

Возможно, вы обратили внимание на использование `nfreeze()` вместо `freeze()`. По умолчанию `Storable` сохраняет числовые данные в самом быстром и простом собственном формате. Проблема в том, что компьютерные системы могут хранить числа разными способами. Использование `nfreeze()` вместо `freeze()` гарантирует запись чисел в формате, переносимом на любые системы.

Возможно, вам интересно узнать, как выглядят эти записи. Ниже представлена запись для мегалитической площадки Каслригг:

```
0302000000050a0a436173746c6572696767580a0743756d62726961580a0a4e59203239312
0323336580a0c53746f6e6520436972636c65580aa34f6e65206f6620746865206c6f76656c
696573742073746f6e6520636972636c65732072656d61696e696e6720746f6461792e20546
869732073697465520697320636f6d707269736564206f66206c6172676520726f756e646564
20626f756c6465727320736574207697468696e2061206e61747572616c20616d706869746
8656174726520666f726d656420627920737572726f756e64696e672068696c6c732e5858
```

Она вся помещается в одной строке файла данных, мы разбили ее на строки, чтобы она уместилась на странице. Чтение не очень захватывающее. Такое представление не позволяет делать быстрые предвари-

тельные проверки типа тех, которые мы осуществляли при использовании `Data::Dumper` и в предыдущих примерах обновления плоских файлов. Можно попробовать делать предварительную проверку после преобразования шестнадцатеричной строки обратно в двоичные данные, но нет никакой гарантии, что строка появляется в выдаче `Storable` в буквальном виде. Сейчас это так, но существует риск, что ситуация может измениться.

Хотя мы обсуждали `Storable` в контексте плоских файлов, эта технология очень полезна при хранении произвольных фрагментов данных Perl в реляционных базах данных, да и в любых других базах данных. `Storable` и `Data::Dumper` — прекрасные инструменты, о существовании которых всегда следует помнить.

Подведем итоги по одноуровневым базам данных

Основное преимущество использования одноуровневых баз для хранения данных заключается в быстроте реализации и скорости работы с небольшими и простыми наборами данных, такими как наша мегалитическая база данных или файл паролей Unix.

Программный код для выборки, вставки, удаления и обновления данных в базе также крайне прост, причем код, производящий разбор данных, может совместно использоваться в разных операциях. Вы сохраняете полный контроль над форматом файлов данных, благодаря чему не можете попасть в положение, когда без вашего ведома изменится формат файла или API для доступа. Эти файлы также легко можно прочесть в стандартном текстовом редакторе (хотя в случае `Storable` это не очень увлекательное чтение).

Недостатки этих баз данных вполне очевидны. Как уже упоминалось, отсутствие возможности одновременного доступа ограничивает возможности таких систем в многопользовательской среде. Кроме того, возникают проблемы масштабирования, обусловленные последовательной природой механизма поиска. Эти проблемы можно решать, усложняя программы (особенно проблему одновременного доступа), но с некоторого момента следует всерьез рассмотреть возможность использования администратора хранения данных более высокого уровня, например файлов DBM. Файлы DBM позволяют также *индексировать* данные и избегать необходимости последовательного доступа.

Перед подробным обсуждением файлов DBM мы в нескольких следующих параграфах рассмотрим примеры более утонченных средств и приемов управления данными, а также способ поддержки одновременного доступа к данным со стороны нескольких пользователей.

Одновременный доступ к базе данных и блокировка

Прежде чем рассматривать управление файловым хранилищем DBM, следует обсудить отмеченные ранее темы, касающиеся одновременного доступа к базам данных в плоских файлах, т. к. эти проблемы возникают со всеми администраторами хранения данных относительно низкого уровня.

Основная проблема состоит в том, что при одновременном доступе к файлам в базе данных могут появляться неопределенные и, как правило, ошибочные данные. Например, если два пользователя решат удалить строку из базы данных по мегалитам с помощью программы, показанной в предыдущем разделе, то на этапе удаления оба пользователя будут работать с первоначальным экземпляром базы данных. Однако после того как один из пользователей первым завершит удаление, результаты его работы будут уничтожены, когда второй пользователь запишет *свою* версию базы данных поверх созданной первым пользователем. Произведенное первым пользователем удаление окажется чудесным образом восстановлено. Эта проблема известна как *ситуация гонок* (*race condition*), и обнаружить ее бывает очень не просто, поскольку трудно воспроизвести условия, в которых она возникает.

Чтобы избежать проблем, возникающих при множественном одновременном доступе, требуется каким-то образом осуществлять монопольный доступ к базе данных при таких потенциально разрушительных операциях, как вставка, обновление и удаление записей. Если бы все программы доступа к базе данных работали в режиме только для чтения, такой проблемы не возникало бы, поскольку никакие данные не изменялись. Если же какой-либо из сценариев должен изменить данные, согласованность всех остальных процессов, обращающихся к данным для чтения или записи, нельзя гарантировать.

Один из способов решения этой проблемы состоит в использовании механизма блокировки файлов, предоставляемого операционной системой, доступ к которому в Perl осуществляется функцией `flock()`. Функция `flock()` реализует кооперативную систему блокировки, которая для своей успешности должна использоваться всеми программами, осуществляющими доступ к данному файлу. Это относится и к сценариям, предназначенным только для чтения, таким как сценарий выборки, приведенный выше, в котором `flock()` можно использовать для проверки того, насколько безопасна попытка чтения базы данных.

Символические константы, используемые в приводимых ниже программах, размещены в пакете `Fcntl` и могут быть импортированы в ваши сценарии для использования с `flock()` при помощи следующей строки:

```
use Fcntl ':flock';
```

`flock()` позволяет осуществлять блокировку в двух режимах – *монопольном* (*exclusive*) и *совместном* (*shared* или *non-exclusive*). Если сценарий осуществил монопольную блокировку, то только он имеет право доступа к файлам базы данных. Любой другой сценарий, желающий получить доступ к базе данных, вынужден ждать освобождения монопольной блокировки, чтобы его запрос блокировки мог быть удовлетворен. Напротив, при совместной блокировке любое количество сценариев может одновременно обращаться к заблокированным файлам, но все попытки получить монопольный доступ будут отвергаться.[1]

Например, сценарий выборки, приведенный в предыдущем разделе, может быть следующим образом усилен благодаря использованию `flock()` для запроса *совместного* доступа к файлам базы данных, чтобы избежать проблем несовместности чтения во время обновления базы данных:

```
### Открыть файл данных для чтения или умереть
open MEGADATA, $ARGV[0] or die "Невозможно открыть $ARGV[0]: $!\n";

print "Получение совместной блокировки...";
flock( MEGADATA, LOCK_SH )
        or die "Невозможно получить совместную блокировку: $!. Аварийный
        выход";
print "Получена совместная блокировка. Готов к чтению базы данных!\n\n";
```

Этот вызов `flock()` не даст выполниться сценарию, пока не будет снята существующая монопольная блокировка на запрашиваемый файл. Когда это произойдет, сценарий выборки получит совместную блокировку и продолжит выборку. Блокировка будет автоматически прекращена при закрытии файла.

Аналогично, сценарий вставки данных можно усилить с помощью `flock()` для запроса *монопольной* блокировки файла до начала его обработки. Необходимо также изменить режим открытия файла: файл должен быть открыт на запись, прежде чем установить на него монопольную блокировку.

Поэтому сценарий вставки можно изменить так:

```
### Открыть файл данных для дополнения или умереть
open MEGADATA, ">>$ARGV[0]"
    or die "Невозможно открыть $ARGV[0] для дополнения: $!\n";

print "Получение монопольной блокировки...";
flock( MEGADATA, LOCK_EX )
       or die "Невозможно получить монопольную блокировку: $!. Аварийный
       выход";
```

[1] Для пользователей Perl под Windows 95 может не быть открытием, что функция *flock( )* в этой системе не поддерживается. Сочувствуем. Попробуйте использовать, например, модуль *LockFile::Simple*.

```
print "Получена блокировка. Готов к обновлению базы данных!\n\n";
```

При этом гарантируется, что никакие операции изменения данных не будут производиться до получения монопольной блокировки файла данных. Аналогичные усовершенствования нужно добавить к сценариям удаления и обновления для гарантии того, что никакие сценарии не «смошенничают» и не проигнорируют процедуры блокировки.

Такая система блокировки действенна во всех системах управления хранением данных, в которых требуется обработка лежащих в их основе файлов базы данных, но нет собственного явного механизма блокировки. Мы вновь обратимся к блокировке при обсуждении системы Berkeley Database Manager, поскольку для нее требуется несколько более сложная стратегия получения описателя файла, к которому должна быть применена функция `flock()`.

Предупреждаем, что `flock()` может оказаться недоступной для вашей конкретной операционной системы. Например, она работает в системах Windows NT/2000, но не в Windows 95/98. Большинством, если не всеми, системами Unix функция `flock()` поддерживается без каких-либо проблем.

Файлы DBM и Berkeley Database Manager

Файлы DBM представляют собой слой администрирования хранения данных, позволяющий программистам сохранять в файлах информацию в виде пар строк – ключа и значения. Файлы DBM являются *двоичными*; строки ключей и значений также могут содержать двоичные данные.

Существует несколько видов файлов DBM, каждый из которых имеет свои сильные и слабые стороны. Perl поддерживает администраторы *ndbm*, *db*, *gdbm*, *sdbm* и *odbm* посредством расширений `NDBM_File`, `DB_File`, `GDBM_File`, `SDBM_File` и `ODBM_File`. Существует также модуль `AnyDBM_File`, который просто использует лучший доступный DBM. В документацию к модулю `AnyDBM_File` входит полезная таблица, в которой сравниваются различные DBM.

Во всех этих расширениях дисковый файл DBM связывается с переменной Perl типа хэш (или *ассоциативный массив*), находящейся в памяти.[1] Простой программный интерфейс, который *выглядит как хэш*, позволяет программистам хранить данные в файлах операционной системы без необходимости вникать в то, как это сделано. Достаточно того, что это работает.

[1] Файлы DBM реализуются посредством библиотечных программ, скомпонованных с расширениями Perl. При этом не создается отдельного серверного процесса.

Программисты заносят значения в хэш и выбирают их оттуда, а обеспечивающий этот процесс слой администрирования хранения данных DBM заботится о том, чтобы перемещать их на диск и обратно.

В этом разделе мы расскажем о самом известном и сложном из этих администраторов хранения данных – Berkeley Database Manager, известном также как Berkeley DB. Доступ к этому программному обеспечению из Perl осуществляется через `DB_File` и расширения Berkeley DB. В системах Windows оно может быть установлено с помощью пакетного администратора Perl – *ppm*. В системах Unix оно компилируется по умолчанию, *только если* на машине уже установлена библиотека Berkeley DB. В Linux обычно так оно и есть, но в большинстве других систем сначала нужно получить и скомпилировать библиотеку Berkeley DB.[1]

Помимо стандартных функций файлов DBM библиотека Berkeley DB и модуль `DB_File` обеспечивают поддержку нескольких различных алгоритмов хранения и извлечения, предназначенных для использования в ситуациях с тонкими различиями. В последних версиях программного обеспечения поддерживаются также одновременный доступ к базам данных и блокировка.

Создание новой базы данных

Прежде чем начать обрабатывать данные в базе Berkeley, необходимо создать новую базу данных или открыть для чтения существующую. Это можно сделать с помощью одного из следующих вызовов функций:

```
tie %hash,  'DB_File', $filename, $flags, $mode, $DB_HASH;
tie %hash,  'DB_File', $filename, $flags, $mode, $DB_BTREE;
tie @array, 'DB_File', $filename, $flags, $mode, $DB_RECNO;
```

Интерес представляет последний параметр этого вызова, т. к. он определяет способ, которым Berkeley DB будет хранить данные в файле базы данных. Сущность этих параметров следующая:

- `DB_HASH` является значением по умолчанию. При этом данные хранятся в соответствии со значением *хэш-функции*, вычисляемой из строки, заданной в качестве ключа. Хэш-таблицы обычно работают очень быстро, поскольку в результате применения хэш-функции к любому заданному значению ключа связанные с этим ключом данные могут быть найдены за одну операцию. Это значительно быстрее, чем последовательный просмотр. Однако хэш-таблицы по

[1] Версию 1 библиотеки Berkeley DB можно получить с *http://www.perl.com/CPAN/src/misc/db.1.86.tar.gz*. Значительно усовершенствованная версия 2 (например *db.2.14.tar.gz*) тоже доступна, но для наших примеров ее наличие не требуется, кроме того, она поддерживается только последними версиями Perl. Вскоре должна появиться версия 3. См. *www.sleepycat.com*.

умолчанию не предоставляют какого-либо полезного упорядочения, а производительность хэш-таблиц может уменьшиться, когда несколько ключей имеет одинаковые значения хэш-функции. Это приводит к тому, что с одним значением хэша связывается несколько записей, что увеличивает время доступа.

- В формате `DB_BTREE` данные в файлах Berkeley DB хранятся в виде сбалансированного двоичного дерева. В технологии хранения B-tree ключи, вводимые в Berkeley DB, по умолчанию сортируются в лексикографическом порядке. При желании можно переопределить порядок сортировки с использованием собственных алгоритмов сортировки.
- Формат `DB_RECNO` позволяет хранить пары ключ/значение в плоских текстовых файлах как с фиксированной, так и с переменной длиной поля. Ключи в этом случае состоят из номеров строк, т. е. номеров записей в базе данных.

При инициализации существующей или новой базы данных Berkeley DB для использования с Perl воспользуйтесь определенным в Perl механизмом `tie`, чтобы связать реальную базу Berkeley DB с хэшем или стандартным скалярным массивом. В результате становятся возможными простая обработка переменных Perl и автоматическое осуществление соответствующих операций над файлами Berkeley DB, избавляя от необходимости ручного программирования Berkeley DB API.

Например, для создания простой базы Berkeley DB можно использовать следующий сценарий Perl:

```
#!/usr/bin/perl -w
#
# ch02/DBM/createdb: Создает БД Berkeley

use strict;

use DB_File;

my  %database;
tie %database, 'DB_File', "createdb.dat"
    or die "Невозможно инициализировать базу данных: $!\n";

untie %database;

exit;
```

Если вы посмотрите каталог, в котором выполняли этот сценарий, то при успешном его выполнении обнаружите новый файл с именем *createdb.dat*. Это дисковый образ вашей базы данных Berkeley, т. е. ваши данные, сохраненные в формате, реализованном в администраторе хранения данных Berkeley DB. Эти файлы обычно называют файлами DBM.

В приведенном примере мы просто указали имя файла, в котором должна храниться база данных, и проигнорировали остальные аргументы. Это вполне допустимо, если значения по умолчанию нас удовлетворяют. Значения дополнительных аргументов по умолчанию представлены в табл. 2.1.

Таблица 2.1. Значения по умолчанию аргументов модуля DB_File

Аргумент	Значение по умолчанию
`$filename`	`undef`*
`$flags`	`O_CREAT | O_RDWR`
`$mode`	`0666`
`$storage_type`	`$DB_HASH`

* Если имя файла задано как `undef`, то база данных будет создана только в памяти. Она ведет себя так же, как если бы записывалась в файл, но после выхода из программы база данных перестает существовать.

Аргумент `$flags` принимает значения, ассоциированные со стандартной функцией Perl `sysopen()`, а аргумент `$mode` имеет вид восьмеричного числа прав доступа к файлу, которые вы хотите присвоить файлу DBM. Если значение по умолчанию равно `0666`, то соответствующие права в Unix будут:

```
-rw-rw-rw-
```

Это означает, что пользователь, группа и все остальные имеют право чтения и записи в этой файл.[1] Можно задать более строгие ограничения для уверенности в том, что неавторизованные пользователи не испортят файлы DBM.

На других платформах, таких как Win32, система прав доступа может не использоваться, в таких случаях режим доступа просто игнорируется.

Исходя из того, что создание новой базы данных является весьма важной операцией, может иметь смысл реализовать механизм монопольного доступа, защищающий файлы базы данных во время первоначального создания и загрузки. Как и в случае баз данных в плоских файлах, для осуществления блокировки на уровне файлов следует использовать функцию Perl `flock()`, но между блокировкой стандартных файлов и файлов DBM есть некоторые различия.

[1] Мы не учитываем модификацию прав, которая может быть произведена `umask`.

Стратегии блокировки

Проблемы безопасного доступа к базам данных, досаждающие при работе с базами данных в плоских файлах, остаются в силе для баз данных Berkeley. Поэтому необходимо реализовать стратегию блокировки, делающую безопасным многопользовательский доступ к базам данных, если это требуется для ваших приложений.

Способ, которым `flock()` используется в отношении файлов DBM, несколько отличен от блокировки стандартных дескрипторов файлов в Perl, поскольку при создании файла DBM в сценарии Perl не возвращается прямая ссылка на дескриптор файла, в котором находится база данных.

К счастью, в модуле `DB_File` есть метод, который можно использовать для нахождения *дескриптора файла*, соответствующего файлу DBM, что позволяет использовать `flock()`. Для получения дескриптора нужно вызвать метод `fd()`, применив его к ссылке на объект, возвращаемой при инициализации базы данных методом `tie()`. Например:

```
### Создание новой базы данных ...
$db = tie %database, 'DB_File', "megaliths.dat"
    or die "Невозможно инициализировать базу данных: $!\n";

### Получить дескриптор файла DBM
my $fd = $db->fd();

### Аккуратно произвести open() с этим дескриптором,
### чтобы получить описатель файла для Perl
open DATAFILE, "+<&=$fd"
      or die "Невозможно безопасно открыть файл: $!\n";

### И заблокировать его до начала загрузки данных ...
print "Получение монопольной блокировки...";
flock( DATAFILE, LOCK_EX )
        or die "Невозможно получить монопольную блокировку: $!.
        Аварийный выход";
print "Получена блокировка. Готов к обновлению базы данных!\n\n";
```

Этот код выглядит несколько неприятно, особенно добавленный вызов `open()`. Вызов написан так, чтобы не повредить исходный дескриптор файла, полученный при создании базы данных для файла DBM. В результате дескриптор файла базы данных связывается с дескриптором файла Perl неразрушающим образом. Это позволяет использовать дескриптор файла с `flock()` как обычно.

Однако после составления этого описания и всех примеров с использованием стандартного документированного способа блокировки файлов Berkeley DBM было обнаружено, что при одновременном доступе к данным сохраняется незначительный риск их повреждения. Суть проблемы состоит в том, что код DBM при первом открытии файла

прочитывает некоторую его часть до того, как вы успеваете его заблокировать.

Есть способ быстро исправить положение, если только в вашей системе поддерживается флаг O_EXLOCK, как это делается в FreeBSD и, вероятно, в большинстве версий Linux. Просто добавьте флаг O_EXLOCK в операцию привязки:

```
use Fcntl;     # импортировать O_EXLOCK, если есть возможность
$db = tie %database, 'DB_File', "megaliths.dat", O_EXLOCK;
```

Дополнительные сведения по этому вопросу и более общий способ его решения можно найти по адресу:

http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/1999-09/msg00954.html

и цепочке сообщений, которая за ним следует.

Вставка и извлечение значений

Вставка данных в базу данных Berkeley с помощью модуля Perl DB_File осуществляется очень просто благодаря использованию *связанного хэша* (*tied hash*) или *связанного массива* (*tied array*). Связь между файлом DBM и структурой данных Perl создается при открытии базы данных. Благодаря этому управление содержимым базы данных осуществляется просто путем изменения содержимого структур данных Perl.

Такая система позволяет с легкостью сохранять данные в файле DBM, а также абстрагирует от сценариев фактические действия с данными в файлах. Поэтому Berkeley DB является администратором хранения данных более высокого уровня, чем простые базы данных в плоских файлах, обсуждавшиеся ранее в этой главе.

В следующем сценарии показаны вставка и извлечение данных из файла DBM с использованием связанного хэша. Этот хэш в Perl состоит из пар ключ/значение. Значения сохраняются в хэш-таблице с уникальными ключами. Благодаря этому достигаются исключительно быстрое извлечение данных и элемент индексирования, в противоположность последовательному доступу. Например:

```
#!/usr/bin/perl -w
#
# ch02/DBM/simpleinsert: Создает БД Berkeley, вставляет некоторые
#                        проверочные данные и затем делает их дамп

use DB_File;
use Fcntl ':flock';

### Инициализация Berkeley DB
```

```
my %database;
my $db = tie %database, 'DB_File', "simpleinsert.dat",
         O_CREAT | O_RDWR, 0666
    or die "Невозможно инициализировать базу данных: $!\n";

my $fd = $db->fd();
open DATAFILE, "+<&=$fd"
    or die "Невозможно безопасно открыть файл: $!\n";
print "Получение монопольной блокировки...";
flock( DATAFILE, LOCK_EX )
    or die "Невозможно получить блокировку: $!. Аварийный выход";
print "Блокировка осуществлена. Готов к обновлению базы данных!\n\n";


### Вставим несколько строк данных
$database{'Каланиш I'} =
"Эта площадка, часто называемая "Северным Стоунхенджем", имеет форму
изогнутого кельтского креста.";

$database{'Эйвбери'} =
"Эйвбери является обширной размашистой площадкой, на которой помимо
прочих чудес находится самый большой каменный круг в Великобритании.
Сам хендж столь велик, что окружает почти целиком деревню Эйвбери.";

$database{'Ландин Линкс'} =
"Ландин Линкс является мегалитической диковиной с тремя грубо отесанными
и очень высокими каменными глыбами, возможно, элементами конструкции
из четырех столбов. Каждый монолит имеет высоту свыше 5 метров.";

### Отвязать базу данных
undef $db;
untie %database;

### Закрыть описатель файла, чтобы снять блокировку
close DATAFILE;


### Заново привязать базу данных для уверенности в том,
### что читаются действительно сохраненные данные
$db = tie %database, 'DB_File', "simpleinsert.dat", O_RDWR, 0444
    or die "Невозможно инициализировать базу данных: $!\n";

### На этот раз нужна блокировка только в совместном режиме, поскольку
### мы не обновляем данные ...
$fd = $db->fd();
open DATAFILE, "+<&=$fd"
     or die "Невозможно безопасно открыть файл: $!\n";
print "Установка совместной блокировки...";
flock( DATAFILE, LOCK_SH )
       or die "Невозможно получить блокировку: $!. Аварийный выход";
print "Установлена блокировка. Готов к чтению базы данных!\n\n";
```

```
### Сделать дамп базы
foreach my $key ( keys %database ) {
    print "$key\n", ( "=" x ( length( $key ) + 1 ) ), "\n\n";
    print "$database{$key}\n\n";
}

### Закрыть базу данных Berkeley
undef $db;
untie %database;

### Закрыть описатель файла, чтобы снять блокировку
close DATAFILE;

exit;
```

При работе этот сценарий создает следующую выходную информацию, показывающую, что значения извлекаются действительно из базы данных:

```
Получение монопольной блокировки...Блокировка осуществлена. Готов к обновлению базы данных!

Установка совместной блокировки...Установлена блокировка. Готов к чтению базы данных!

Каланиш I
============

Эта площадка, часто называемая Северным Стоунхенджем, имеет форму
изогнутого кельтского креста.

Эйвбери
========

Эйвбери является обширной размашистой площадкой, на которой помимо
прочих чудес находится самый большой каменный круг в Великобритании.
Сам хендж столь велик, что окружает почти целиком деревню Эйвбери.

Ландин Линкс
=============

Ландин Линкс является мегалитической диковиной с тремя грубо отесанными
и очень высокими каменными глыбами, возможно, элементами конструкции
из четырех столбов. Каждый монолит имеет высоту свыше 5 метров.
```

Возможно, вы заметили, что мы слегка смошенничали в этом примере. Мы сохранили только описания площадок, а не все данные, такие как код карты и местоположение. Это проблема, свойственная базам данных, которые образованы парами ключ/значение: с данным ключом можно сохранить только одно значение. Обойти это ограничение

можно простой конкатенацией значений в одну строку и записью этой строки, подобно тому, как мы делали это с помощью `join()`, `pack()`, `Data::Dumper` и `Storable` ранее в этой главе.

Такого рода фокус при хранении можно выполнить, по крайней мере, двумя способами.[1] Один состоит в том, чтобы вручную объединить данные в строку, а при необходимости вручную разобрать. Другой способ несколько более сложен и использует объект Perl, инкапсулирующий обрабатываемый мегалит и скрывающий операции упаковки и распаковки.

Локализованное хранение и извлечение

Первый способ – обработка приложением объединения в строку и расщепления – является, конечно, наиболее самодостаточным. Здесь нужно сделать небольшое отступление.

Самодостаточность может быть благотворна, поскольку в результате логика сценария сосредоточивается в нем самом, несколько облегчая понимание. К несчастью, такая локализация может послужить источником больших огорчений. Хорошим примером является наша мегалитическая база данных. В предыдущем разделе мы написали на Perl четыре разных сценария для решения четырех основных задач обработки данных. При локализации логики вы, в сущности, реализуете один и тот же код для записи и извлечения в четырех разных местах.

Более того, если вы решите изменить формат данных, то вам придется согласованно переписать четыре различных сценария. Исходя из того, что вы, вероятно, добавите новые сценарии для выполнения более специфических функций (например, создания веб-страниц) над соответствующими данными из базы, возникнет еще больше мест потенциального отказа вашей базы данных и неизбежного повреждения данных.

Вернемся к нашему примеру. Можно довольно легко записывать сложные данные в файл DBM, используя либо `join()` для создания строки с разделителями, либо `pack()` для создания записей фиксированной длины. Получить желаемый результат с помощью `join()` можно следующим образом:

```
### Вставим несколько строк данных
$database{'Каланиш I'} =
    join( ':', 'Каланиш I', 'Каланиш, Западные острова', 'NB 213 330',
```

[1] Как всегда в Perl, «есть не один способ сделать это» – There's More Than One Way To Do It (столь распространенная фраза в Perl, что ее часто можно встретить как акроним TMTOWTDI). Мы излагаем здесь эти идеи, потому что они прежде всего пришли нам в голову. Ваше решение может оказаться значительно более диковинным и неясным или мучительно очевидным. Таков Perl.

```
                'Каменный круг', 'Описание Каланиша I' );
$database{'Эйвбери'} =
    join( ':', 'Эйвбери', 'Уилтшир', 'SU 103 700',
                'Каменный круг и хендж',
                'Описание Эйвбери' );

$database{'Ландин Линкс'} =
    join( ':', 'Ландин Линкс', 'Файф', 'NO 404 027', 'Стоящие камни',
                'Описание Ландин Линкс' );

### Сделать дамп базы данных
foreach my $key ( keys %database ) {
    my ( $name, $location, $mapref, $type, $description ) =
         split( /:/, $database{$key} );
    print "$name\n", ( "=" x length( $name ) ), "\n\n";
    print "Местонахождение:      $location\n";
    print "Код карты: $mapref\n";
    print "Описание:   $description\n\n";
}
```

Сохранять данные в виде записей фиксированной длины так же просто, но при этом база данных довольно быстро пожирает пространство. Более того, главной причиной использования записей фиксированной длины часто является стремление увеличить скорость доступа, однако при хранении данных в файле DBM запросы и обновления по месту просто не дают заметного увеличения скорости.

В следующем фрагменте программного кода показано, как вставлять и выводить данные при использовании записей фиксированной длины:

```
### Шаблон для упаковки и распаковки.
$PACKFORMAT = 'A64 A64 A16 A32 A256';

### Вставим несколько строк данных
$database{'Каланиш I'} =
    pack( $PACKFORMAT, 'Каланиш I', 'Каланиш, Западные острова',
                'NB 213 330', 'Каменный круг', 'Описание Каланиша I');

$database{'Эйвбери'} =
    pack( $PACKFORMAT'Эйвбери', 'Уилтшир', 'SU 103 700',
                'Каменный круг и хендж', 'Описание Эйвбери');

$database{'Ландин Линкс'} =
    pack( $PACKFORMAT, 'Ландин Линкс', 'Файф', 'NO 404 027',
                'Стоящие камни', 'Описание Ландин Линкс');

### Сделать дамп базы данных
foreach my $key ( keys %database ) {
    my ( $name, $location, $mapref, $type, $description ) =
        unpack( $PACKFORMAT, $database{$key} );
    print "$name\n", ( "=" x length( $name ) ), "\n\n";
    print "Местонахождение:      $location\n";
```

```
    print "Код карты: $mapref\n";
    print "Описание:   $description\n\n";
}
```

Реальный код механизма хранения и извлечения не намного более ужасен, чем при использовании разделителей в записях, но он создает невнятность в виде шаблона для `pack()`, о котором легко можно позабыть или ввести с ошибками. При этом фактически не решается проблема локализации логики, а сопровождение превращается в уже упоминавшийся кошмар.

Как можно улучшить этот код?

Упаковка в объектах Perl

Одним из решений как проблемы локализованного кода, так и проблемы хранения нескольких величин в одной паре хэша ключ/значение является использование *объектов* Perl для инкапсуляции данных и укрытия некоторых неприятных вещей.[1]

В следующей программе на Perl определяется объект класса `Megalith`. Этот пакетный объектный модуль может затем использоваться во всех наших программах, которые не нужно переписывать, если мы изменим характер работы модуля:

```
#!/usr/bin/perl -w
#
# ch02/DBM/Megalith.pm: Класс Perl, инкапсулирующий мегалит

package Megalith;

use strict;
use Carp;

### Создает новый объект megalith и инициализирует поля-члены.
sub new {
    my $class = shift;
    my ( $name, $location, $mapref, $type, $description ) = @_;
    my $self = {};
    bless $self => $class;

    ### Если аргумент только один, предполагаем, что в $name находится
    ###  строка, содержащая все значения полей, и распаковываем ее
```

[1] Здесь может возникать некоторая путаница в отношении Perl. Использование объектов, методов доступа и маскирования данных тесно связано с объектно-ориентированным подходом. В нашей программе смешано удобство обычного программирования с элементами чисто объектно-ориентированного программирования. Традиционные ОО-программисты часто поднимают большой шум, когда программисты на Perl публично обсуждают такого рода вещи.

```
    if ( @_ == 1 ) {
        $self->unpack( $name );
    }
    else {
        $self->{name} = $name;
        $self->{location} = $location;
        $self->{mapref} = $mapref;
        $self->{type} = $type;
        $self->{description} = $description;
    }
    return $self;
}

### Упаковывает текущие значения полей в запись с
### разделителем-двоеточием и возвращает ее
sub pack {
    my ( $self ) = @_;

    my $record = join( ':', $self->{name}, $self->{location},
                            $self->{mapref}, $self->{type},
                            $self->{description} );

    ### Простая проверка того, что в полях не содержатся двоеточия
    croak "Поле записи содержит символ-разделитель ':' "
        if $record =~ tr/:/:/ != 4;

    return $record;
}

### Распаковывает заданную строку в поля-члены
sub unpack {
    my ( $self, $packedString ) = @_;

    ### Примитивный разбор...Предполагается отсутствие
    ### разделителей в полях
    my ( $name, $location, $mapref, $type, $description ) =
         split( ':', $packedString, 5 );

    $self->{name} = $name;
    $self->{location} = $location;
    $self->{mapref} = $mapref;
    $self->{type} = $type;
    $self->{description} = $description;
}

### Вывод данных о мегалите
sub dump {
    my ( $self ) = @_;

    print "$self->{name} ( $self->{type} )\n",
          "=" x ( length( $self->{name} ) +
```

```
                    length( $self->{type} ) + 5 ), "\n";
    print " Местонахождение:      $self->{location}\n";
    print " Код карты: $self->{mapref}\n";
    print " Описание:   $self->{description}\n\n";
}

1;
```

В формате записи, определенном в модуле, содержатся элементы данных, относящиеся к каждой мегалитической площадке, которые программы могут запрашивать и изменять. Новый объект `Megalith` можно создать в Perl с помощью оператора `new`, например:

```
### Создать новый объект, инкапсулирующий Стоунхендж
$stonehenge =
    new Megalith( 'Стоунхендж', 'Описание Стоунхенджа',
                  'Уилтшир', 'SU 123 400' );

### Вывести название площадки, записанное в объекте...
print "Название: $stonehenge->{name}\n";
```

Было бы прекрасно, если бы эти объекты `Megalith` можно было непосредственно записывать в файл DBM. Попробуем написать несложный код, который просто помещает объект в хэш:

```
### Создать новый объект, инкапсулирующий Стоунхендж
$stonehenge =
    new Megalith'Стоунхендж', 'Описание Стоунхенджа',
                  'Уилтшир', 'SU 123 400');

### Сохраним объект в хэше базы данных
$database{'Стоунхендж'} = $stonehenge;

### Посмотрим, что находится в записи базы данных
print "Ключ: $database{'Стоунхендж'}\n";
```

Результат получается, по меньшей мере, странный:

```
Key: Megalith=HASH(0x80e9aec)
```

Оказывается, вместо самого объекта в базу данных Berkeley помещается строка, *описывающая* ссылку на объект Perl!

Полученному результату не следует удивляться, т. к. система DBM предназначена для хранения одиночных строк и не имеет встроенных способов упаковки сложных объектов в одну величину. Она просто превращает все ключи и значения в строки.

К счастью, проблему сохранения объекта Perl можно обойти путем упаковки, или *маршаллинга*, значений всех полей объекта `Megalith` в одну строку, а затем вставки этой строки в базу данных. Аналогично,

после извлечения строки из базы данных можно создать новый Megalith и распаковать строку в его соответствующие поля.

Чтобы реализовать это, можно написать следующий код, использующий наш удачно определенный класс Megalith (обратите внимание на вызов метода pack()):

```
$database{'Каланиш I'} =
    new Megalith( 'Каланиш I',
                  'Западные острова',
                  'NB 213 330',
                  'Каменный круг',
                  'Описание Каланиша I' )->pack();

### Сделаем дамп базы данных
foreach $key ( keys %database ) {

    ### Распакуем запись в новый объект Megalith
    my $megalith = new Megalith( $database{$key} );

    ### И выведем запись
    $megalith->dump();
}
```

В объекте Megalith определены два метода с именами pack() и unpack(). Они просто упаковывают поля в одну строку с разделителями и распаковывают строку в соответствующие поля объекта, как это требуется. Если объект Megalith создается при передаче ему единственного аргумента, являющегося такой строкой, метод unpack() вызывается косвенным образом, а внутренние детали управления хранением данных скрыты от программиста.

Аналогично, фактический способ упаковки и распаковки данных скрыт от пользователя модуля. Это означает, что при необходимости сделать изменения в структуре базы данных это можно осуществить внутри модуля без необходимости обновления сценариев работы с базой данных.

Если вы читали ранее в этой главе раздел о том, как помещать сложные данные в плоские файлы, то знаете, что есть не один способ сделать это.

Поэтому, несмотря на то что в начале приходится проделать немного большую работу, фактически хранить в файлах DBM объекты Perl (и другие сложные виды данных) достаточно просто.

Методы доступа к объекту

Чтобы навести окончательный глянец на класс Megalith, следует добавить *методы доступа*, позволяющие управлять доступом к значениям, хранящимся в объектах. В приведенном выше примере кода дос-

туп к переменным, являющимся членами объекта, осуществлялся явным образом:

```
print "Название мегалита: $megalith->{name}\n";
```

При таком способе могут возникнуть проблемы, если внутренняя структура объекта `Megalith` претерпит какие-либо изменения. Кроме того, если по ошибке написать что-либо вроде `$megalith->{nme}`, никаких сообщений об ошибке или предупреждений выдано не будет. Определим метод доступа `getName()`, например, таким образом:

```
### Возвращает название мегалита
sub getName {
    my ( $self ) = @_;
    return $self->{name};
}
```

Можно утверждать, что программа становится в результате более удобочитаемой:

```
print "Название мегалита: " . $megalith->getName() . "\n";
```

и обеспечивается корректность кода приложения, поскольку фактическая логика снова перемещается в объект.

Ограничения в запросах к данным файлов DBM и хэш-таблицам

Несмотря на возможность вставлять в файл Berkeley DB сложные данные (хотя и несколько окружным путем), существует фундаментальное ограничение на это программное обеспечение баз данных: извлекать значения можно только по одному ключу. Это означает, что поиск в нашей базе данных по мегалитам можно осуществлять только по имени, а не по коду карты или местонахождению.

Это может составить довольно большую проблему, если вы хотите выполнять такие запросы, как, скажем, «найти все площадки в Уилтшире», не указывая точного названия. В этом случае нужно проверять каждую запись: удовлетворяет ли она критерию. При этом вместо индексированного поиска по ключу используется последовательный доступ.

Решением проблемы может быть создание вторичных *ссылочных хэшей*, ключами которых будут различные поля, поиск по которым вы хотите осуществлять. Для каждого ключа значением будет *ссылка* на исходный хэш, а не на отдельное значение. Это позволяет изменять значение в исходном хэше, а новое значение будет автоматически отражаться в ссылочных хэшах. В следующем фрагменте программы показано, как можно создавать и выводить ссылочный хэш, ключом которого является местоположение мегалитической площадки:

```
### Построить ссылочный хэш, основываясь на местонахождении
### каждого памятника
$locationDatabase{'Wiltshire'}     = \$database{'Эйвбери'};
$locationDatabase{'Western Isles'} = \$database{'Каланиш I'};
$locationDatabase{'Fife'}          = \$database{'Ландин Линкс'};

### Сделать дамп базы данных местонахождений
foreach $key ( keys %locationDatabase ) {

    ### Распаковать запись в новый объект Megalith
    my $megalith = new Megalith( ${ $locationDatabase{$key} } );

    ### Вывести запись
    $megalith->dump();
}
```

У такого решения есть, конечно, свои недостатки. Наиболее очевидным из них является тот, что при всяком удалении или вставке данных необходимо произвести соответствующую операцию в каждом вторичном ссылочном хэше.

Но самая большая проблема при таком подходе связана с тем, что ключи в данных могут не быть уникальными. Если мы храним записи для Стоунхенджа и Эйвбери, то надо иметь в виду, что обе площадки расположены в Уилтшире. В таком случае последняя введенная запись всегда перезаписывает ранее введенные в хэш. Для решения этой проблемы можно использовать функцию файлов Berkeley DB, позволяющую создавать *цепочки значений*.

Создание цепочек значений в хэше

Одной из больших проблем при работе с файлами DBM и использовании механизма хранения `DB_HASH` является необходимость уникальности ключей, с которыми сохраняются данные. Например, если мы записываем два различные значения с ключом «Уилтшир», например, для Стоунхенджа и Эйвбери, то обычно в базе данных хранится последнее значение, введенное в хэш. Это, по меньшей мере, проблематично.

В хорошем проекте базы данных для ускоренного поиска в любой структуре данных должен быть уникальный *первичный ключ*. Но в наскоро и неаккуратно созданных базах данных, плохо спроектированных или с низким качеством данных, такая уникальность может быть не обеспечена. Аналогичным образом, эта проблема возникает при использовании ссылочных хэш-таблиц для поиска в базе данных по ключу, не являющемуся первичным.

В Perl решение проблемы состоит в том, чтобы поместить множественные значения в массив, который хранится в элементе хэша. Пока программа выполняется, этот прием работает прекрасно, но если база данных записывается на диск и снова загружается, данные теряются.

Для решения проблемы рассмотрим использование DB_BTREE, другого метода администрирования хранения данных в Berkeley DB, в котором ключи упорядочиваются перед вставкой. В этом механизме допускается использование одинаковых ключей, поскольку используемый в нем файл DBM имеет вид массива, а не хэш-таблицы. К счастью, обращение к файлу DBM по-прежнему происходит через хэш-таблицу Perl, поэтому DB_BTREE использовать не сложнее. Главным недостатком использования DB_BTREE является снижение производительности, поскольку извлечение данных из бинарного дерева обычно несколько медленнее, чем из хэш-таблицы.

В следующей короткой программе создается база данных Berkeley с использованием механизма хранения DB_BTREE, а также устанавливается флаг, показывающий, что допускается использование повторяющихся ключей. Вводится ряд строк с повторяющимися ключами, а затем делается дамп базы, чтобы показать, что ключи сохранены:

```
#!/usr/bin/perl -w
#
# ch02/DBM/dupkey1: Создает Berkeley DB с механизмом DB_BTREE и
#                   допускает повторяющиеся ключи. Затем вставляются
#                   некоторые тестовые данные с повторяющимися ключами
#                   и делается дамп полученной базы данных.

use DB_File;
use Fcntl ':flock';
use Megalith;

### Установить режим Berkeley DB BTree для обработки повторяющихся
### ключей
$DB_BTREE->{'flags'} = R_DUP;

### Удалить существующие файлы базы данных
unlink 'dupkey2.dat';

### Открыть базу данных
my %database;
my $db = tie %database, 'DB_File', "dupkey2.dat",
              O_CREAT | O_RDWR, 0666, $DB_BTREE
    or die "Невозможно инициализировать базу данных: $!\n";

### Установить монопольную блокировку базы данных,
### чтобы никто не имел к ней доступа
my $fd = $db->fd();
open DATAFILE, "+<&=$fd"
    or die "Невозможно безопасно открыть файл: $!\n";
print "Получение монопольной блокировки...";
flock( DATAFILE, LOCK_EX )
    or die "Невозможно получить блокировку: $!. Аварийный выход";
print "Блокировка осуществлена. Готов к обновлению базы данных!\n\n";
```

```
### Создать, упаковать и вставить несколько строк с повторяющимися
### ключами
$database{'Уилтшир'} =
  new Megalith( 'Эйвбери',
                'Уилтшир',
                'SU 103 700',
                'Каменный круг и хендж',
             'Самый большой каменный круг в Великобритании' )->pack();

$database{'Уилтшир'} =
  new Megalith( 'Стоунхендж',
                'Уилтшир',
                'SU 123 400',
                'Каменный круг и хендж',
                'Самый общеизвестный каменный круг в мире' )->pack();

$database{'Уилтшир'} =
  new Megalith( 'Святилище',
                'Уилтшир',
                'SU 118 680',
                'Каменный круг (разрушен)',
                'Описание отсутствует' )->pack();

### Сделать дамп базы данных
foreach my $key ( keys %database ) {

    ### Распаковать запись в новый объект Megalith
    my $megalith = new Megalith( $database{$key} );

    ### И вывести запись
    $megalith->dump();
}

### Закрыть базу данных
undef $db;
untie %database;

### Закрыть дескриптор файла, чтобы снять блокировку
close DATAFILE;

exit;
```

В результате выполнения этой программы получается не совсем то, на что мы рассчитывали:

```
Получение монопольной блокировки...Блокировка осуществлена.
Готов к обновлению базы данных!

Святилище ( Каменный круг (разрушен) )
==============================================
Местонахождение:   Уилтшир
Код карты:         SU 118 680
```

```
Описание:          Описание отсутствует

Святилище ( Каменный круг (разрушен) )
==========================================
Местонахождение:   Уилтшир
Код карты:         SU 118 680
Описание:          Описание отсутствует

Святилище ( Каменный круг (разрушен) )
==========================================
Местонахождение:   Уилтшир
Код карты:         SU 118 680
Описание:          Описание отсутствует
```

Похоже на то, что вместо трех разных записей мы умудрились записать три экземпляра одной и той же записи!

К счастью, это не так. Мы правильно записали три различные записи с одним и тем же ключом в файле DBM. Проблема в том, как мы пытаемся прочесть эти записи из файла DBM. Очевидно, не работает разадресация с использованием ключа хэша, поскольку, как мы уже знаем, в Perl для каждого ключа хранится только одно значение.

Чтобы обойти это ограничение, можно использовать метод `seq()`, объявленный в модуле `DB_File`, который используется для прохода по *связанным в цепочку* записям, хранящимся в одном элементе хэша. На рис. 2.2 показан принцип просмотра связанных в цепочку записей в элементе хэша.

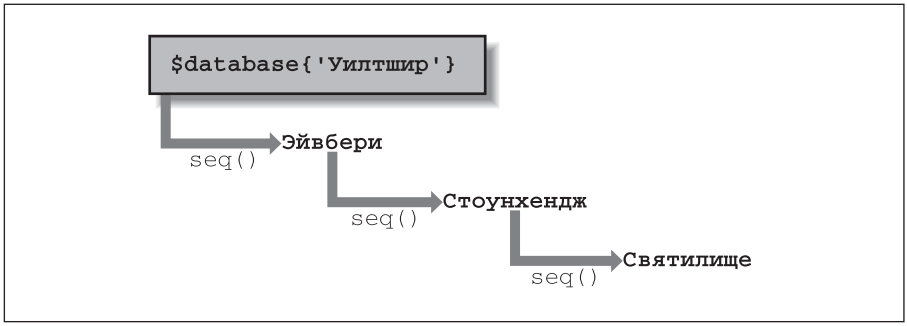

Рис. 2.2. *Просмотр связанных в цепочку записей*

С использованием `seq()` исправленный код дампа записей выглядит следующим образом:

```
### Сделать дамп базы данных
my ($key, $value, $status) = ('', '', 0);
for ( $status = $db->seq( $key, $value, R_FIRST ) ;
      $status == 0 ;
      $status = $db->seq( $key, $value, R_NEXT ) ) {
```

```
    ### Распаковать запись в новый объект Megalith
    my $megalith = new Megalith( $value );

    ### И вывести запись
    $megalith->dump();
}
```

При выполнении этой исправленной версии получается тот результат, который мы ожидали, т. е. записи для трех различных мегалитических площадок.

Метод `seq()` просто понять и использовать, и он хорошо работает в соединении с циклом `for`, как показано выше. Метод принимает три аргумента: ключ хэша, значение хэша и флаг, указывающий, какой элемент в цепочке нужно возвратить. В качестве первых двух аргументов методу `seq()` действительно передаются ключ хэша и правильное значение хэша соответственно. Возвращаемое значение хэша зависит от значения третьего аргумента:

- `R_FIRST` возвращает первую запись в цепочке.
- `R_LAST` возвращает последнюю запись в цепочке.
- `R_NEXT` возвращает следующую запись в цепочке. Используется для прохода цепочки в прямом направлении.
- `R_PREV` возвращает предыдущую запись в цепочке. Используется для прохода цепочки в обратном направлении.
- `R_CURSOR` возвращает запись, в которой обнаружено *частичное соответствие* с ключом. Этим достигается некоторая степень «нечеткого соответствия» ключей. Эта функция не всегда работает корректно и может возвращать ближайшее соответствие требуемому ключу вместо точного совпадения. Например, если искать все площадки в Уилтшире (Wiltshire) и просить частичного совпадения с «wilt», то при отсутствии точного совпадения могут быть возвращены записи с местонахождением «Western Isles», поскольку это ближайшее соответствие ключу поиска.

В показанном примере дампа базы данных мы начинаем с первой записи цепочки и проходим цепочку в прямом направлении. Можно было осуществлять поиск в обратном направлении, написав такой код:

```
for ( $status = $db->seq( $key, $value, R_LAST ) ;
      $status == 0 ;
      $status = $db->seq( $key, $value, R_PREV ) ) {
    ...
}
```

Существует более простой и быстрый метод запроса повторяющихся значений: `get_dup()`. Этот метод возвращает либо число записей с заданным ключом, либо массив или хэш, содержащий соответствующие

записи. Например, если в нашей базе данных есть три записи с ключом Уилтшир, то в этом можно убедиться, написав такой код:

```
### Выводит число записей с ключом "Уилтшир"
my $numRecords = $db->get_dup( 'Уилтшир' );
print "Число записей с ключом Уилтшир: $numRecords\n";
```

Удаление значений

Последняя операция, которую можно осуществлять над файлами DBM, – это удаление значений. Обновление сводится к присвоению других значений соответствующим ключам в базе данных, удаление столь же просто. Эта операция осуществляется с помощью стандартной функции Perl `delete` по соответствующему ключу в базе данных. `delete` удаляет его из хэша, представляющего базу данных, а поскольку хэш привязан к файлу DBM, ключ удаляется и оттуда.

Следующая программа вставляет в Berkeley DB три записи и делает дамп базы, чтобы показать, что эти записи в ней есть. Затем удаляется одна запись и делается повторный дамп базы данных, чтобы продемонстрировать удаление. Вот текст программы:

```
#!/usr/bin/perl -w
#
# ch02/DBM/delete: Создает Berkeley DB, вставляет тестовые данные
#                  и затем удаляет некоторые из них

use strict;

use DB_File;

### Инициализировать Berkeley DB
my  %database;
tie %database, 'DB_File', "delete.dat"
    or die "Невозможно инициализировать базу данных: $!\n";

### Вставить несколько строк данных
$database{'Каланиш I'}    = "Западные острова";
$database{'Эйвбери'}      = "Уилтшир";
$database{'Ландин Линкс'} = "Файф";

### Сделать дамп базы данных
print "Полный дамп базы данных...\n";
foreach my $key ( keys %database ) {
    printf "%15s - %s\n", $key, $database{$key};
}
print "\n";

### Удалить строку
delete $database{'Эйвбери'};
```

```
### Сделать повторный дамп базы данных
print "Дамп базы данных после удалений...\n";
foreach my $key ( keys %database ) {
    printf "%15s - %s\n", $key, $database{$key};
}

### Закрыть Berkeley DB
untie %database;

exit;
```

Программа выводит то, что и ожидалось:

```
Полный дамп базы данных...
       Каланиш I - Западные острова
         Эйвбери - Уилтшир
    Ландин Линкс - Файф

Дамп базы данных после удалений...
       Каланиш I - Западные острова
    Ландин Линкс - Файф
```

Это означает, что заданная строка перманентно удалена из базы данных в результате удаления соответствующего элемента хэша.

Модуль MLDBM

Модуль `MLDBM` очень полезен для быстрой записи сложных структур данных Perl в файлы DBM для постоянного хранения. `ML` в `MLDBM` происходит от *multilevel* (многоуровневость) и указывает на способность хранения сложных многоуровневых структур данных. Это возможность, которой обычные хэши, даже привязанные к файлам DBM, не обладают.

Модуль `MLDBM` являет собой отличный пример многоуровневого администратора хранения данных. Он выступает как тонкий слой поверх другого модуля DBM, перехватывающий операции чтения и записи для автоматической *сериализации* (или десериализации) данных с использованием другого модуля.[1]

Модуль работает путем автоматической сериализации структур данных Perl, которые вы хотите сохранить, в одну строку, записываемую затем в файл DBM. Обратное восстановление данных производится десериализацией сохраненной строки в подходящий объект Perl. Интерфейс для указания сохраняемых и извлекаемых данных идентичен

[1] Мы обсуждали сериализацию, а также модули `Data::Dumper` и `Storable` ранее в этой главе.

API для файлов DBM. Благодаря этому очень легко «подкинуть» MLDBM вместо существующего модуля DBM.

В следующем примере показано, как можно использовать DB_File для записи и Data::Dumper для вывода извлеченных данных:

```
#!/usr/bin/perl -w
#
# ch02/mldbmtest: Демонстрирует запись сложных структур данных
#                 в файл DBM с помощью модуля MLDBM.

use MLDBM qw( DB_File Data::Dumper );
use Fcntl;

### Удалить тестовый файл, если он уже существует ...
unlink 'mldbmtest.dat';

tie my %database1, 'MLDBM', 'mldbmtest.dat', O_CREAT | O_RDWR, 0666
    or die "Невозможно инициализировать файл MLDBM: $!\n";

### Создадим в базе данных несколько мегалитических записей
%database1 = (
    'Эйвбери' => {
        name => 'Эйвбери',
        mapref => 'SU 103 700',
        location => 'Уилтшир'
    },
    'Кольцо Бродгара' => {
        name => 'Кольцо Бродгара',
        mapref => 'HY 294 133',
        location => 'Оркни'
    }
);

### Отвязать и привязать заново, чтобы показать,
### что данные находятся в файле
untie %database1;

tie my %database2, 'MLDBM', 'mldbmtest.dat', O_RDWR, 0666
    or die " Невозможно инициализировать файл MLDBM: $!\n";

### Сделать дамп сохраненного через Data::Dumper...
print Data::Dumper->Dump( [ \%database2 ] );

untie %database2;

exit;
```

Результат выполнения программы таков:

```
$VAR1 = {
      'Эйвбери' => {
```

```
                    'name' => 'Эйвбери',
                    'location' => 'Уилтшир',
                    'mapref' => 'SU 103 700'
                  },
    'Кольцо Бродгара' => {
                          'name' => 'Кольцо Бродгара',
                          'location' => 'Оркни',
                          'mapref' => 'HY 294 133'
                        }
};
```

Это показывает, что вложенные данные в исходном хэше восстановлены верно.

Заключение

Глава получилась длинной как для читателей, так и для писателей. Мы осветили много тем и надеемся, что помогли вам понять основы баз данных и снабдили некоторыми новыми для вас приемами работы.

Мы почти готовы к тому, чтобы перейти к изложению собственно DBI, но перед этим хотим познакомить вас с прелестями SQL.

3

SQL и реляционные базы данных

Структурированный язык запросов (*Structured Query Language*), или *SQL*,[1] – это язык, предназначенный для обработки данных в базах данных.

В 1970 году Е. Ф. Кодд (E. F. Codd), работавший в IBM, опубликовал ставшую классической работу «Реляционная модель данных для больших совместно используемых банков данных» («A Relational Model of Data for Large Shared Data Banks»), в которой заложил ряд теоретических принципов управления данными, ныне известными как *реляционная модель*. В эту статью уходит корнями вся сфера технологий реляционных баз данных.

Одним из многих направлений исследований, вызванных этой статьей, стала разработка и реализация языка, позволяющего легко взаимодействовать с реляционными базами данных. Этот язык позволил программистам избежать написания ужасно сложных программ для работы с такими базами данных.[2]

Данная глава ставит своей задачей дать неофиту баз данных весьма ограниченный обзор того, что представляет собой SQL и как с его помощью можно решать ряд простых задач. Многие сложные детали ар-

---

[1] Официально произносится «эс-кью-эль», хотя распространено также произношение «сиквел». Иногда можно услышать и «сквил» (squeal – визг), но обычно лишь от людей, впервые увидевших его синтаксис или только что удаливших все свои данные.

[2] В главе 2 показаны многочисленные примеры того, каким многословным и негибким может быть взаимодействие с базами данных.

хитектуры SQL и его работы в этом обзоре опущены или их изложение в значительной мере упрощено, чтобы дать новичкам знания, необходимые для использования DBI простым, но эффективным образом. В предисловии к книге перечислены другие книги и веб-серверы, посвященные SQL и технологиям реляционных баз данных.

Методология реляционных баз данных

В модели реляционных баз данных активно используются единицы хранения данных, называемые *таблицами*, с которыми связан ряд атрибутов, известных как *колонки*. Например, нам может понадобиться хранить в таблице `megaliths` название мегалитической площадки, ее местонахождение, тип площадки и на какой карте ее можно найти. Каждый из этих элементов данных будет представляться отдельной колонкой.

Как правило, в больших базах данных таблицы создаются в рамках структур, называемых *схемами*. Схема является группой логических структур данных, или *объектов схемы*, в число которых входят таблицы и представления (views). В некоторых базах данных схема соответствует пользователю, создаваемому внутри базы данных. В других базах данных она является более общим способом группировки связанных таблиц. Например, в нашей базе данных по мегалитам при использовании Oracle мы создаем пользователя с именем *stones*. Внутри схемы пользователя *stones* создаются различные таблицы, образующие вместе базу данных по мегалитам.

Данные в таблице хранятся в виде *строк*. Это означает, что данные, относящиеся к одной площадке, хранятся в одной строке, содержащей в каждой колонке соответствующие значения. Такое размещение данных точно соответствует метафоре «строка-колонка», используемой в электронных таблицах, бухгалтерских книгах и даже в простых разграфленных блокнотных страницах, где можно писать от руки.

Вот пример такого списка, содержащего данные по мегалитам:

```
Площадка       Местонахождение   Тип                    Код карты
--------       ---------------   ---                    ----------
Каланиш I      Западные острова  Каменный круг и ряды   NB 213 330
Стоунхендж     Уилтшир           Каменный круг и хендж  SU 123 422
Эйвбери        Уилтшир           Каменный круг и хендж  SU 103 700
Санхани        Абердиншир        Круг лежащих камней    NJ 716 058
Ландин линкс   Файф              Четыре столба          NO 404 027
```

Такая система хорошо подходит для обобщенного запроса типа «Сообщить названия всех мегалитов» или «Сообщить картографический код всех мегалитов в Уилтшире». Для выполнения таких запросов нужно просто указать колонки, которые мы хотели бы видеть, и усло-

вия, которым должна удовлетворять каждая колонка в строке, включаемой в результат запроса.

Подобный же синтаксис можно использовать для управления данными, например, «Вставить в таблицу `megaliths` новую строку со следующими значениями...» или «Удалить все строки с мегалитами в Файф».

Явная простота SQL находится в противоречии с тем фактом, что он обладает исключительно мощным синтаксисом для управления данными, хранимыми в базах, и способствует поддержке логической структуры данных.

Главный упор в архитектуре реляционных баз данных сделан на том, что родственные данные должны храниться либо в одном и том же месте, либо в разных местах, но связанных с оригиналом каким-либо явным образом. Одним из принципов является также то, что данные в базе не должны дублироваться.

Используя в качестве примера нашу базу данных по мегалитам, мы решили хранить все данные, непосредственно относящиеся к каждой мегалитической площадке, в таблице `megaliths`, а все мультимедийные клипы – в отдельной таблице. Это хороший пример реляционной базы данных, хотя и маленькой, поскольку если бы мы хранили мультимедийные клипы в таблице `megaliths`, то пришлось бы многократно дублировать данные по мегалитам – фактически для каждого клипа. Это приводит к *избыточности данных* – одной из проблем, избежать которую позволяет реляционная модель.

Мы также выделили категории площадок в отдельную таблицу с именем `site_types`, чтобы еще сильнее сократить избыточность данных.

Процесс рационального размещения данных в таблицах для уменьшения избыточности данных носит название *нормализации*. В некоторых ситуациях желательно проведение обратной операции, называемой *денормализацией*.

Данные, хранимые в нескольких нормализованных таблицах, извлекаются с помощью *соединений* таблиц, позволяющих запросам извлекать колонки из всех таблиц, участвующих в соединении. Соединения, к примеру, позволят нам извлечь название мегалитической таблицы и URL мультимедийного клипа в одном запросе. Это эффективный способ хранения данных, при котором хранится ровно столько данных, сколько необходимо для извлечения необходимой информации.

Недостатком систематического использования многотабличных соединений является снижение производительности при работе с базами, содержащими большой объем данных. Для связи строк между таблицами требуется дополнительное дисковое пространство, а базе данных может оказаться затруднительно установить такую связь наилучшим образом.

Это важная проблема в отрасли знаний, известной как создание *хранилищ данных* (*data warehousing*), в которых большие объемы знаний хранятся таким образом, чтобы дать возможность пользователям создавать отчеты и производить анализ этих данных. Типичным решением в таких ситуациях является создание широких *денормализованных* таблиц, содержащих значительное количество дублирующейся информации. В результате значительно возрастает производительность за счет увеличения расхода дискового пространства. Поскольку данные, содержащиеся в хранилищах, обычно используются только для чтения, не приходится беспокоиться о синхронизации изменений в данных.

Для демонстрации различных способов использования SQL для выборки и управления данными в этих главах используется небольшая база данных, состоящая из трех таблиц. Эти таблицы называются `megaliths`, `media` и `site_types`. На рис. 3.1 показана структура этих трех таблиц.

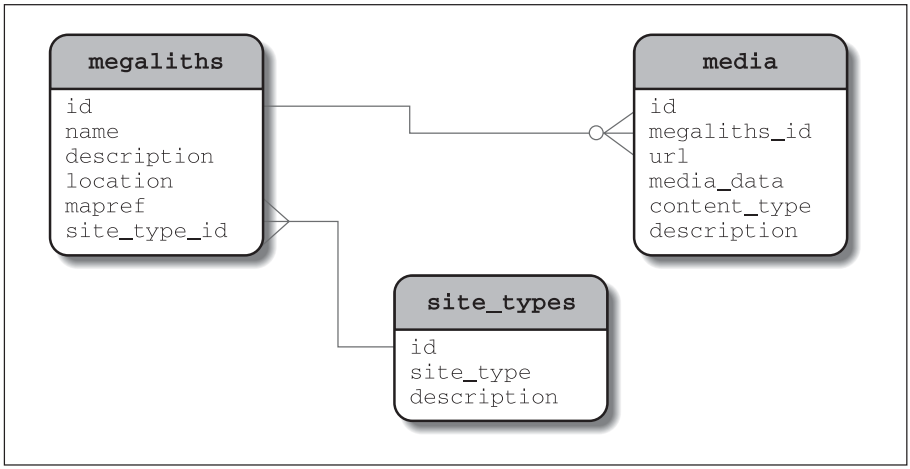

Рис. 3.1. *База данных по мегалитам*

Таблицы предназначены для хранения данных по мегалитическим площадкам и относящимся к ним мультимедийным клипам. На практике с каждой мегалитической площадкой связываются ноль или более мультимедийных клипов, и каждая площадка относится ровно к одному типу. Эта небольшая база данных ляжет в основу примеров, приводимых на протяжении всей оставшейся части книги.

Типы данных и значения NULL

Одним из наиболее важных аспектов структур, определяемых в базе данных, таких как таблицы и представления, является *тип данных* для каждой колонки. Perl является слабо типизированным языком, в

то время как SQL – сильно типизированным. Поэтому каждое поле или значение имеет установленный тип данных, который определяет способ сравнения полей и значений. Например, поле `mapref` (код карты) в таблице `megaliths` оказалось бы не очень полезным, если бы в нем можно было хранить только даты!

Поэтому важно назначить каждой колонке правильный тип данных. Благодаря этому можно избежать возможной путаницы в отношении того, как должны интерпретироваться значения, хранимые в каждой колонке, а также устанавливается порядок сравнения этих величин в условных выражениях запросов.

Существует несколько распространенных типов данных. Наиболее широко используемые из них можно сгруппировать следующим образом:

Числовые типы данных

В группу числовых входят такие типы, как целые числа и числа с плавающей запятой (действительные). В зависимости от базы данных в их число могут входить `FLOAT`, `REAL`, `INTEGER` и `NUMBER`. Сравнение числовых типов производится очевидным образом, т. е. проверяются фактические значения.

Символьные типы

Символьные типы используются для хранения и обработки текстовых данных. В символьном типе данных могут храниться любые символы, будь то буквы или цифры.

Однако если в символьном типе хранятся цифры, они будут рассматриваться как символьные строки, а не числа. Например, они будут сортироваться и упорядочиваться как строки, а не числа, поэтому «10» окажется меньше, чем «9».

В зависимости от конкретной базы данных могут существовать различные виды символьных типов, например `CHAR`, `VARCHAR`, `VARCHAR2` и т. д. В большинстве баз данных поддерживается по крайней мере основной из них – `CHAR`.

При сравнении символьных типов данных обычно используется лексикографический порядок соответственно набору символов, используемому базой данных.

Тип данных «дата»

В большинстве СУБД реализован хотя бы один тип данных для хранения дат, отличный от символьного типа данных, используемого для хранения строкового представления даты. Он позволяет легко производить различные арифметические операции над датами. Например, можно выбрать строки, в которых поле, содержащее дату, соответствует понедельнику.

При сравнении дат большей из них считается более поздняя. Кроме того, часто встречаются типы для хранения времени и временных меток (дата плюс время).

Типы данных двоичных объектов

Типы данных двоичных объектов появились в системах баз данных сравнительно недавно. Они позволяют хранить в базах данных большие неструктурированные массивы данных, такие как изображения, видео- и аудиоклипы. Фактические типы данных двоичных объектов часто различны в разных базах данных, но если они есть, то обычно называются LOB (large objects). Например, тип данных BLOB хранит двоичные данные, а тип CLOB хранит большие объемы символьных данных в ASCII. Как правило, сравнивать типы LOB между собой нельзя.

Значение NULL

NULL является значением особого типа, свидетельствующим о фактическом отсутствии значения. Если в колонке содержится NULL, это означает, что в данной строке эта колонка не содержит значения. Значение NULL используется, когда фактическое значение неизвестно или несущественно.

При создании таблицы для каждой колонки можно определить, допустимо ли иметь в ней значения NULL независимо от типа данных колонки.

Значения NULL не следует смешивать с числовым значением «нуль». Это не одно и то же. «Нуль» означает нуль, а NULL означает отсутствие значения.[1]

При вычислении выражения, в которое входит величина NULL (но не в составе различных специальных функций обработки NULL), результатом всегда является NULL. При сравнении величин с NULL всегда нужно использовать `IS NULL` и `IS NOT NULL`.[2] Будьте внимательны!

Значение NULL участвует в так называемых таблицах «трехзначной логики», которые используются при расчете таблиц истинности условных предложений, о чем будет сказано далее в этой главе. Благодаря этому условные выражения SQL могут иметь значения *true*, *false* или *NULL*.

[1] Впрочем, некоторые базы данных воспринимают вводимые пустые строки как NULL.

[2] В некоторых базах данных, например mSQL, можно использовать `= NULL`.

Запрос данных

Первая (и, вероятно, наиболее полезная) операция, которую SQL позволяет осуществлять над данными, – это выборка и возврат строк из таблиц, хранимых в базах данных. Эта функция представляет собой самую суть того, что собственно представляет собой база данных – большое хранилище информации, в котором возможен поиск.

Во всех запросах SQL, сколь бы простыми или сложными они ни были, используется ключевое слово `SELECT` для указания колонок, которые должны быть выбраны, таблиц, в которых находятся эти колонки, и условий, которым должны удовлетворять извлекаемые строки. `SELECT` относится к группе команд, которая называется *языком обработки данных* (*Data Manipulation Language*), или *DML*.

Полный синтаксис `SELECT` может показаться для новичка устрашающим в основном потому, что в нем используется много способов модификации запросов. Например, можно потребовать возврата только уникальных строк данных, группировки строк или даже задать способ сортировки возвращаемых строк.

Мы сейчас рассмотрим только простейшие случаи.

В нашем примере для команд SQL и других зарезервированных слов используются прописные и строчные буквы для имен объектов базы данных (таблиц, колонок и т. д.). В большинстве баз данных команды SQL не чувствительны к регистру, но в именах объектов базы данных регистр может учитываться.

Простые запросы

В простейшем запросе SQL запрашиваются некоторые колонки из всех строк таблицы. Синтаксис `SELECT` для такого запроса можно описать следующим образом:

```
SELECT column, column, ..., column
FROM table
```

или:

```
SELECT *
FROM table
```

если нужно выбрать все колонки указанной таблицы.

Таким образом, для выбора строк данных из нескольких колонок нашей таблицы `megaliths` можно использовать следующую команду SQL:

```
SELECT name, location, mapref
FROM megaliths
```

которая возвращает следующие данные:

```
+-----------------------------------------------------------------+
| name          | location                          | mapref      |
+-----------------------------------------------------------------+
| Каланиш I     | Каланиш, о. Льюис                 | NB 213 330 |
| Ландин Линкс | Ландин Линкс, Файф, Шотландия     | NO 404 027 |
| Стоунхендж    | Близ Эймсбери, Уилтшир, Англия    | SU 123 400 |
| Эйвбери       | Эйвбери, Уилтшир, Англия          | SU 103 700 |
| Санхани       | Близ Инш, Абердиншир              | NJ 716 058 |
+-----------------------------------------------------------------+
```

Итак, даже в простейшем запросе SQL, который можно вообразить, присущая синтаксису гибкость позволяет легко указать, какая в точности информация из базы данных нам требуется без необходимости мучительного написания многих программных строк для ее извлечения.

Здесь проявляется еще одна сторона методологии реляционных баз данных: хотя в базе данных содержится информация во всех колонках некоторой таблицы, в конкретных случаях требуется извлечение данных только из некоторого подмножества. Следовательно, можно извлекать *только* те данные, которые нужны в данном запросе, и никакие другие. Это очень мощная функция, позволяющая разделять фактические данные, хранящиеся в базе данных, от желательного представления этих данных.

Запросы и условные предложения

В предыдущем примере извлекались все строки таблицы, тогда как в повседневных операциях с базами данных обычно требуется более точный отбор отдельных строк. Например, запрос «Сообщить названия всех каменных кругов в Уилтшире» является более узким, чем «Сообщить обо всех каменных кругах в базе данных». Для решения этой задачи SQL предоставляет возможность задания *условий*, которые для строк, возвращаемых пользователю, должны быть удовлетворены.

Синтаксис SQL для условных предложений столь же прост, как для указания колонок, представляющих интерес. Условные предложения, сужающие запрос, помещаются *после* списка таблиц, из которых извлекаются данные, т. е. после предложения `FROM` и списка таблиц.

Поэтому запрос, возвращающий колонки `name` и `location` в строках, содержащих строку «Уилтшир» в колонке `location`, можно записать следующим образом:

```
SELECT name, location
FROM megaliths
WHERE location LIKE '%Уилтшир%'
```

Этот запрос возвращает следующие данные:

```
+-------------------------------------------------+
| name          | location                        |
+-------------------------------------------------+
| Стоунхендж    | Близ Эймсбери, Уилтшир, Англия  |
| Эйвбери       | Эйвбери, Уилтшир, Англия        |
+-------------------------------------------------+
```

В возвращенных данных присутствуют только заданные колонки и только для тех площадок, значение `location` которых содержит строку «Уилтшир». Ключевое слово `WHERE` стоит в начале списка условий, которые должны быть удовлетворены для каждой возвращаемой строки. Условие означает, что колонка `location` должна содержать нужную строку «Уилтшир».[1]

В следующей таблице перечислены все *операторы сравнения*, используемые в SQL для проверки условий. Эти условия в основном напоминают синтаксис Perl и не должны вызывать затруднений.

Оператор	Назначение

=	Этот оператор проверяет точное равенство значений колонок и/или литералов. Например, запрос:
	`SELECT name, location`
	`FROM megaliths`
	`WHERE location = 'Файф'`
	возвращает все строки, в которых значение колонки `location` равно значению "Файф".

<>	Этот оператор проверяет наличие неравенства значений колонок и/или литералов. Например, запрос:
	`SELECT name, location`
	`FROM megaliths`
	`WHERE location <> 'Файф'`
	возвращает все строки, в которых значение колонки `location` *не* равно значению "Файф". В некоторых базах данных вместо <> используются альтернативные операторы `!=`, `^=` или `~=`.

> и <	Эти операторы представляют проверку отношений «больше» и «меньше» между значениями колонок и/или литералов. Например, запрос:
	`SELECT name, location`
	`FROM megaliths`
	`WHERE id < 10`
	`AND id > 5`

[1] В данном случае символ «%» используется в качестве стандартного символа-заместителя SQL, соответствующего любому числу символов. Иногда в базах данных в том же качестве используется символ «*».

(продолжение)

Оператор	Назначение
	возвращает название и местонахождение для всех площадок, у которых значение `id` меньше 10 и больше 5. Тип сравнения зависит от типа данных участвующих величин. Числовые величины сравниваются как числа, строковые величины сравниваются как строки, а величины типа «дата» сравниваются как даты. Существуют также родственные операторы `<=` и `>=`, осуществляющие проверки «больше или равно» и «меньше или равно», соответственно.

`IN`	Это ключевое слово служит для проверки равенства колонки и/или литерала одному из заданного множества значений. Например, в запросе:
	`SELECT name, location`
	`FROM megaliths`
	`WHERE location IN ( 'Западные острова', 'Файф' )`
	проверяется равенство заданной колонки и каждого элемента множества. Поэтому этот запрос возвращает строки, в которых значением колонки `location` является `"Западные острова"` или `"Файф"`.

`LIKE`	Оператор `LIKE` допускает ограниченное использование символов-заместителей для поиска соответствия строк и значений колонок и/или литералов. Например, запрос:
	`SELECT name, description`
	`FROM megaliths`
	`WHERE description LIKE '%Самый большой%'`
	возвращает строки с колонками `name` и `description`, в которых колонка `description` содержит в какой-либо позиции строку `"Самый большой"`.
	Символами-заместителями могут быть либо знак процента (`%`) для задания любого числа символов, либо подчеркивание (`_`) для соответствия одному символу.*

* В некоторых базах данных вместо `LIKE` используются другие ключевые слова, например `MATCHES` или `CONTAINS`, а также другие символы-заместители, например «`*`» или «`?`».

Со временем мегалитическая база данных может разрастись и включать в себя данные по тысячам площадок в стране. Поэтому для быстрого поиска записей может потребоваться ужесточить критерии поиска, включив другие условия, которым должны удовлетворять возвращаемые записи. Например, при намерении найти данные по всем каменным кругам в Уилтшире простой запрос по всем площадкам в Уилтшире возвратил бы сотни записей, разбираться с которыми пришлось бы вручную. Можно сузить поиск, задав дополнительное условие, чтобы колонка `mapref` начиналась со строки `SU 123`:

```
SELECT name, location
FROM megaliths
WHERE location LIKE '%Уилтшир%'
AND mapref LIKE 'SU 123%'
```

В этом примере второе условие просто добавляется в конец списка условий, которые должны быть удовлетворены. Два условия соединяются логическим оператором `AND`. Теперь эту команду можно прочесть как «Выдать название и местонахождение всех мегалитических площадок, находящихся в Уилтшире в картографическом районе SU 123», в результате чего будут возвращены поля `name` и `location` для строки «Стоунхендж», но не для строки «Эйвбери».

Таким образом условия можно объединять в списки, связанные логическими булевыми операторами, управляющими тем, как вычисляется истинность условия.

В следующей таблице описаны определенные в SQL булевы (или логические) операторы, которые можно использовать для соединения условных предложений.

Оператор	Функция			
AND	Возвращает логическое И двух предложений с обеих сторон ключевого слова. Для определения значения объединенного предложения можно использовать следующую таблицу истинности.			
		TRUE	FALSE	NULL
	TRUE	true	false	null
	FALSE	false	false	false
	NULL	null	false	null
OR	Возвращает логическое ИЛИ двух предложений с обеих сторон ключевого слова. Для определения значения объединенного предложения можно использовать следующую таблицу истинности.			
		TRUE	FALSE	NULL
	TRUE	true	true	true
	FALSE	true	false	null
	NULL	true	null	null
NOT	Отрицает логическое значение следующего за ним выражения. Его действие показано в следующей таблице истинности.			
		TRUE	FALSE	NULL
		false	true	null

Истинность всего условия определяется истинностью каждого отдельного элемента и применением операторов AND, OR и NOT.

Теперь можно вычислить результат наличия в команде нескольких условных предложений. Например, следующее условное предложение:

```
WHERE location LIKE '%Уилтшир%'
AND mapref LIKE 'SU 123%'
```

вычисляется для этой строки следующим образом:

```
+-----------------------------------------------------------+
| name          | mapref     | location                     |
+-----------------------------------------------------------+
| Эйвбери       | SU 103 700 | Эйвбери, Уилтшир, Англия     |
+-----------------------------------------------------------+

location LIKE '%Уилтшир%'      => TRUE
mapref LIKE 'SU 123%'           => FALSE

TRUE AND FALSE => FALSE
```

Поэтому для этой строки возвращается значение false, и она отклоняется запросом. Однако следующая строка:

```
+-------------------------------------------------------------+
| name         | mapref     | location                        |
+-------------------------------------------------------------+
| Стоунхендж   | SU 123 400 | Близ Эймсбери, Уилтшир, Англия  |
+-------------------------------------------------------------+
```

приводит к следующим вычислениям:

```
location LIKE '%Уилтшир%'     => TRUE
mapref LIKE 'SU 123%'           => TRUE

TRUE AND TRUE => TRUE
```

что обеспечивает ее возврат запросом.

При сочетании нескольких логических операторов важно учитывать их сравнительный *приоритет*. Приоритет операторов определяет порядок, в котором они выполняются при вычислении результата. В стандарте SQL определено, что наивысший приоритет имеет оператор NOT, за ним следует AND и затем OR. Для изменения порядка следования группы операторов могут заключаться в скобки.

Например, вы хотите выбрать все мегалитические площадки, находящиеся в Уилтшире или в Файф, для которых описание площадки содержит слова «внушающий благоговение».

Этот запрос можно (ошибочно) записать как:

```
SELECT name, location
```

```
FROM megaliths
WHERE location LIKE '%Уилтшир%' OR location LIKE '%Файф%'
AND description LIKE '%внушающий благоговение%'
```

На первый взгляд он выглядит корректно, но на самом деле не учитывает порядка, в котором сочетаются условные предложения. Он выбирает все внушающие благоговение площадки в Файф, но также и *все* площадки в Уилтшире независимо от типа.

Это происходит потому, что у оператора `AND` больший приоритет, чем у `OR`, и он выполняется первым. Поэтому наша команда SQL сначала выполняет `AND` над `location LIKE %Файф%` и `description LIKE %внушающий благоговение%`. Затем оператор `OR` объединяет результат операции `AND` с `location LIKE %Уилтшир%`. Это совсем не то, что мы имели в виду.

Более корректно записать этот запрос с использованием скобок для логической группировки операторов. Например:

```
SELECT name, location
FROM megaliths
WHERE ( location LIKE '%Уилтшир%' OR location LIKE '%Файф%' )
AND description LIKE '%внушающий благоговение%'
```

Это изменяет порядок вычисления условных предложений. Сначала вычисляются предложения, объединенные в одну группу, а затем этот результат используется вместо отдельных значений истинности для каждого предложения, входящего в группу.

Наконец, достаточно часто используется другой, еще более сложный способ задания условных предложений. Он состоит в том, чтобы получить используемые при сравнении величины в результате выполнения вложенного запроса.[1] Например:

```
SELECT name, description
FROM megaliths
WHERE name IN
    ( SELECT tourist_sitename
      FROM wiltshire_tourist_sites )
```

Если бы заранее было известно, что вложенный запрос может вернуть только одну строку данных, то вместо `IN` можно было бы использовать оператор `=`.

Запросы к нескольким таблицам

В предыдущем разделе рассказывалось о структуре команд SQL в целом и о том, как можно использовать SQL для запроса данных из од-

[1] Поддержка этой функции может отсутствовать в некоторых системах баз данных, поддерживаемых DBI и его драйверами.

ной таблицы базы данных. Однако из изложения теории реляционных баз данных ранее в этой главе вам должно быть понятно, что мощь и гибкость архитектуры реляционных баз данных заключается в возможности объединения таблиц, т. е. в связывании отдельных записей, располагающихся в разных таблицах, с целью уменьшения дублирования данных. Это связывание записей является важнейшей частью работы с реляционными базами данных.

Для иллюстрации этого понятия нужно познакомиться с другими таблицами, которые мы будем использовать в наших примерах, а именно: с таблицей `media` и таблицей `site_types`.

Таблица `media` содержит данные о том, где находятся мультимедийные клипы для указанных площадок, что позволяет внешним приложениям показывать или проигрывать эти клипы, пока пользователь изучает текстовую информацию о площадке, хранящуюся в таблице `megaliths`.

Аналогично `site_types` является справочной таблицей (lookup table) для различных категорий мегалитических площадок, описанных в базе данных.

Для задания команды `SELECT` из двух или более таблиц базы данных мы просто добавляем их имена после ключевого слова `FROM`. Поэтому запрос на получение всех строк в двух таблицах теоретически должен выглядеть примерно так:

```
SELECT name, description, location, url, content_type
FROM megaliths, media
```

Однако такой запрос возвращает несколько путаные на вид данные. Для каждой строки первой таблицы выбираются *все* строки второй таблицы![1] Это означает, что мультимедийные записи для «Ландин Линкс» могут быть возвращены одновременно с информацией о площадке «Эйвбери».

Как добиться того, чтобы значения в полях второй таблицы были связаны со значениями из первой, т. е. клипы для Стоунхенджа возвращались только вместе с информацией по Стоунхенджу?

В таблице `megaliths` мы уже определили колонку с именем `id`, содержащую уникальный идентификатор для каждой строки, хранимой в таблице. Аналогично в таблице `media` есть колонка с именем `id`, служащая той же цели. Более того, в таблице `media` есть еще колонка с именем `megaliths_id`. При вставке строки в таблицу `megaliths` или `media` в колонку

[1] Это ситуация известна как «декартово произведение». Если в каждой таблице 100 строк, то в результате возвращается 10 000 строк. Если в каждой таблице 1 000 000 строк, то возвращается 1 000 000 000 000 строк. Чтобы избежать этого, при наличии `n` таблиц следует иметь хотя бы `n-1` условий соединения.

id помещается уникальный идентификатор. Кроме того, при вставке строки в таблицу media в колонку megaliths_id помещается уникальный идентификатор мегалитической площадки, к которой относится мультимедийный клип.

Эта связь между полями обычно называется связью между *первичным ключом* и *внешним ключом*. Первичный ключ – это уникальное значение, хранящееся в «главной» таблице. Внешний ключ – это то же самое значение, содержащееся в строках другой, «справочной» таблицы.

Теперь можно написать запрос для извлечения из базы данных нужной информации, *соединив* две таблицы по *связанным* полям. Этим обеспечивается связь мультимедийных клипов с правильными площадками:

```
SELECT name, description, url, content_type
FROM megaliths, media
WHERE megaliths.id = megaliths_id
```

Этим иллюстрируется другая сторона условных предложений SQL: вместо сравнений произвольных значений с колонками таблицы можно сравнивать значения различных колонок. В вышеприведенном примере мы проверяем соответствие первичного и внешнего ключей двух таблиц и в случае совпадения возвращаем объединенную запись, состоящую из колонок двух таблиц.

Обратите также внимание на то, как мы квалифицировали имя поля id в условном предложении, предварив его именем таблицы с точкой. Без этого база данных не могла бы определить, обращаемся мы к полю id в таблице megaliths или в таблице media, и выдала бы ошибку.

Аналогично, если мы пожелаем выбрать поля id из обеих таблиц, то следующая команда просто собьет с толку базу данных, и мы снова получим ошибку:

```
SELECT id, id, megaliths_id
FROM megaliths, media
WHERE id = megaliths_id
```

Поэтому в тех случаях, когда может создаваться двусмысленность, следует явно указывать имя таблицы, к которой принадлежит поле. Например:

```
SELECT megaliths.name, megaliths.description,
       media.url, media.content_type
FROM megaliths, media
WHERE megaliths.id = media.megaliths_id
```

Конечно, неприятно, что нужно вводить много символов с клавиатуры. Более разумной альтернативой является использование *псевдонимов* имен таблиц. Для этого нужно просто добавить псевдоним после имени таблицы в предложении FROM. Например:

```
SELECT mega.name, mega.description, med.url, med.content_type
FROM megaliths mega, media med
WHERE mega.id = med.megaliths_id
```

Чаще в качестве псевдонима используется первый символ имени таблицы, если при этом сохраняется уникальность псевдонимов.

Эта теория соединения таблиц распространяется на любое количество таблиц.[1] На практике случается, что некоторые таблицы в базе данных содержат в колонках только внешние ключи, которые делают многотабличные соединения более эффективными. Главное – нужно помнить, что все таблицы, участвующие в запросе, должны соединяться с другими по какой-либо колонке. В противном случае база данных возвращает большое число очень странных результатов!

Например, чтобы получить мультимедийные данные, связанные с площадкой, и данные о типе площадки, можно использовать такой запрос:

```
SELECT mega.name, mega.description, st.site_type,
       med.url, med.description
FROM megaliths mega, media med, site_types st
WHERE mega.id = med.megaliths_id
AND mega.site_type_id = st.id
```

Другим типом соединения, которое позволяет производить SQL, является *внешнее соединение*. В дополнение к результатам, возвращаемым простым соединением, внешнее соединение возвращает строки одной из таблиц, для которых в другой таблице не нашлось строк, удовлетворяющих условию соединения. При этом во всех строках второй таблицы, в которой не нашлось значений, соответствующих значениям первой таблицы, возвращаются значения `NULL`.

Допустим, мы хотим получить данные по всем площадкам, расположенным в Уилтшире, а также URL для связанных с ними мультимедийных клипов, если они существуют. Использование простого соединения:

```
SELECT mega.name, mega.location, med.url
FROM megaliths mega, media med
WHERE mega.id = med.megaliths_id
AND mega.location LIKE '%Уилтшир%'
```

возвращает только те площадки в Уилтшире, с которыми связаны мультимедийные клипы, а площадки без клипов исключаются. Эту проблему можно решить путем использования *внешнего соединения*.

[1] В действительности базы данных, поддерживающие соединения, нередко имеют определенную верхнюю границу на количество соединяемых таблиц.

Официальным стандартным способом обозначения внешнего соединения является использование вместо запятой между таблицами фразы `LEFT OUTER JOIN` (левое внешнее соединение) или `RIGHT OUTER JOIN` (правое внешнее соединение) и добавление предложения `ON условное_выражение`.[1]

Например, стандартный запрос для извлечения требуемой нам информации может быть написан так:

```
SELECT mega.name, mega.location, med.url
FROM megaliths mega
     LEFT OUTER JOIN media med ON mega.id = med.megaliths_id
WHERE mega.location LIKE '%Уилтшир%'
```

В этом примере мы получили левое внешнее соединение по колонкам `id` и `megaliths_id`, поскольку для всех площадок без мультимедийных клипов нет соответствующих записей в таблице `media`. Левое внешнее соединение гарантирует, что даже при отсутствии записей по мультимедийным клипам будут возвращены, по крайней мере, название и местонахождение всех площадок в Уилтшире. Правое внешнее объединение в этом запросе возвратило бы значения, для которых не существует записей в таблице `megaliths`.

Наконец, стоит упомянуть о некоторых способах получения эффективных соединений таблиц. В наших примерах для осуществления соединения мы вставляли в таблицы дополнительные колонки, но можно было просто добавить в таблицу `media` колонку, содержащую название мегалитической площадки.

Существует несколько веских причин, по которым мы этого не сделали:

1. Если бы название мегалитической площадки по какой-то причине изменилось, например, в случае исправления обнаруженной орфографической ошибки, название, хранящееся в таблице `media`, стало бы устаревшим и неверным. В результате связь между двумя таблицами по этой строке была бы разорвана.

2. Целочисленные ключи занимают значительно меньше места, чем строки при построении индексов. Меньшее место означает, что в блоке дискового пространства помещается больше записей, а значит, требуется меньше операций чтения с диска. Меньший размер индекса и меньшее число операций чтения с диска компенсируют некоторое увеличение размера главной таблицы данных и обычно дают выигрыш как в скорости, так и в дисковом пространстве.

[1] Во многих базах данных внешнее соединение либо не поддерживается, либо использует другой синтаксис. Например, в Oracle 7 внешние соединения выглядят так же, как и внутренние, но к одной из сторон условия соединения добавляются три символа «+».

3. Сравнение строк осуществляется медленнее, чем сравнение чисел, особенно целых. Поэтому в хорошо спроектированных базах данных целые числа часто используются для первичных и внешних ключей, поскольку операторы сравнения быстрее работают с ними. Строки требуют сравнения всех символов, что может отнимать много времени.

Следовательно, для увеличения скорости выполнения запросов можно проектировать базу данных таким образом, чтобы соединения осуществлялись с использованием целочисленных колонок. Часто полезно строить индексы по колонкам внешних ключей в «справочных» таблицах, если база данных поддерживает такую возможность.

Группировка и упорядочение данных

Часто желательно иметь несколько больший контроль над тем, как возвращаются отобранные данные. Наиболее часто используются два способа организации данных – упорядочение извлекаемых данных по одной или более колонкам и группировка извлеченных строк с применением функций к группам, а не к отдельным строкам.

Perl может успешно решать такие задачи в вашей программе, но осуществление группировки и упорядочения через SQL переложит эту задачу на сервер базы данных, а также убережет вас от необходимости создания и использования приемов организации данных, которые, возможно, не будут оптимальными. Поэтому, как правило, следует использовать SQL, а не собственный код уровня приложения.

Упорядочение данных

Упорядочение данных, извлекаемых командой `SELECT`, можно осуществить с помощью предложения `ORDER BY`. Это предложение всегда помещается в конце запроса после задания всех условий соединения.

Предложение `ORDER BY` состоит из разделенного запятыми списка колонок, которые должны использоваться при упорядочении данных. Например, предложение `ORDER BY` вида:

```
ORDER BY name, location
```

упорядочит строки по названию, а если названия площадок одинаковы, то для вторичного упорядочения будет использована колонка местонахождения. Направление упорядочения можно изменить с установленного по умолчанию порядка «по возрастанию» (т. е. от A до Z) на порядок «по убыванию», дополнив любое имя поля в предложении `ORDER BY` ключевым словом `DESC`.

Группировка данных

Возможность группировки элементов данных очень полезна при создании итоговых отчетов. В SQL входит предложение `GROUP BY`, позволяющее группировать строки с одинаковым набором значений и применять к ним *групповые функции*.

Хорошим примером такой операции является случай, когда требуется суммировать значения в некоторой колонке таблицы. Например, можно использовать групповую функцию `sum()` для суммирования стоимости заказов, принятых в течение определенного дня:

```
SELECT order_date, sum( net ), sum( vat ), sum( total )
FROM sales
GROUP BY order_date
```

Как и в предложении `ORDER BY`, группировки можно объединять в списке через запятую, чтобы создавать сложные подгруппы колонок.

Изменение данных в таблицах

Базы, открытые только для чтения (т. е. позволяющие применять к содержащимся в них данным только команду `SELECT`), очень полезны. *Хранилища данных* обычно являются большими базами, открытыми только для чтения и заполненными архивными данными в виде, удобном для составления отчетов. Однако в базах данных, предназначенных для обработки транзакций, возможность быстрой и эффективной *модификации* данных имеет первостепенное значение.

Существует несколько базовых операций, входящих в более широкое понятие модификации данных, а именно:

- *вставка* новых данных в базу;
- *удаление* данных из базы;
- *обновление*, или модификация, данных, существующих в базе.

Каждая из этих операций относится к группе команд языка обработки данных, или DML, наряду с `SELECT`.

Мы поочередно рассмотрим эти задачи и применим теорию к нашей учебной базе данных.

Вставка данных

Для того чтобы база данных могла принести какую-то реальную пользу, в нее должны быть внесены некоторые данные в результате ручного ввода или автоматизированной процедуры загрузки. Операция вставки данных означает необходимость ввода в базу данных совершенно новой записи. Если запись уже существует и требуется лишь

модификация данных в какой-либо колонке, то следует использовать операцию *обновления*.

Вставка данных в реляционной модели производится на построчной основе: каждая запись или элемент информации, которые вы загружаете в базу данных, соответствуют новой строке в существующей таблице. Поскольку каждая вставляемая запись соответствует новой строке в таблице, вставка нескольких записей невозможна.[1]

Ключевое слово SQL `INSERT` обеспечивает простой механизм вставки новых строк в базу. Например, в предположении, что таблица `megaliths` уже присутствует в базе и содержит шесть колонок, показанных ранее на рис. 3.1, можно вставить в нее одну строку с помощью команды SQL:

```
INSERT INTO megaliths VALUES ( 0, 'Каланиш I',
                               '"Северный Стоунхендж"',
                               'Западные острова',
                               'NB 213 330', 1 )
```

Если снова выбрать все строки таблицы командой `SELECT`, то среди них должна быть видна только что введенная строка.

Подобно тому как в команде `SELECT` можно указать колонки таблицы, которые должен вернуть запрос, можно (и является хорошей практикой) указать, в какие колонки таблицы должны быть вставлены значения. Неуказанные колонки получат значение по умолчанию (обычно `NULL`). Например, если вы хотите задать только колонки `id` и `name`, допуская, что `description` и `location` могут иметь значение `NULL`, команда SQL будет следующей:

```
INSERT INTO megaliths ( id, name ) VALUES ( 0, 'Каланиш I' )
```

Должно быть точное соответствие между числом колонок и значениями колонок, указанными в команде SQL. Важно также обеспечить соответствие типов данных передаваемых значений и соответствующих колонок.

Использование INSERT для передачи данных

Одним из наиболее скрытых применений ключевого слова `INSERT` является передача данных из одной таблицы или колонки в другую в результате одной простой операции. Это выглядит как противоречие предыдущему утверждению, что каждая команда `INSERT` может вста-

[1] Это не совсем точно в наше время, поскольку серверы баз данных становятся более интеллектуальными. Например, в Oracle допускается вставка в представления equi-join, а также поддерживаются триггеры «INSTEAD OF», с помощью которых можно делать почти все, что угодно.

вить только одну строку, но на самом деле корректно следует правилам (хотя и коварным образом).

Например, если мы хотим быстро скопировать таблицу megaliths в новую таблицу с именем megaliths_copy, то можно воспользоваться следующей командой SQL:

```
INSERT INTO megaliths_copy
    SELECT *
    FROM megaliths
```

В результате этого процесса все строки, возвращаемые командой SELECT, одна за другой вставляются в новую таблицу, пока не будет создана точная копия исходной таблицы. Эта функция INSERT очень полезна, если нужно быстро сделать копию таблиц, чтобы произвести над оригиналом какие-либо разрушительные действия, например, удалить избыточные данные. Для работы этой команды SQL исходная и результирующая таблицы должны иметь одинаковую структуру.

Можно уточнить эту операцию, указав условия, которым должны удовлетворять перемещаемые строки, чтобы быть вставленными. Например, чтобы скопировать только строки данных для мегалитов, находящихся в Уилтшире:

```
INSERT INTO megaliths_copy
    SELECT *
    FROM megaliths
    WHERE location LIKE '%Уилтшир%'
```

Более того, можно делать из таблиц *извлечение* данных в новые таблицы, явно указывая целевые колонки новой таблицы. Это полезно при создании больших денормализованных таблиц для использования в хранилищах данных. Поэтому если бы у нас была таблица по имени megalocations, содержащая две колонки с именами name и location, можно было бы заполнить эту новую таблицу на основе таблицы megaliths следующим образом:

```
INSERT INTO megalocations
    SELECT name, location
    FROM megaliths
```

Можно даже выбирать данные для вставки из нескольких таблиц. Денормализованная таблица, содержащая строки, сращенные из данных таблиц megaliths и media, может содержать две колонки – name и url. Заполнить эту таблицу командой INSERT просто:

```
INSERT INTO megamedia
    SELECT name, url
    FROM megaliths, media
    WHERE megaliths.id = media.megaliths_id
```

Однако заполнение таблицы с помощью команд INSERT обычно осуществляется программами пакетной загрузки, генерирующими подходящие команды SQL и выполняющими их в базе данных, например SQL*Loader в Oracle. Конечно, Perl дает хороший пример языка программирования, весьма облегчающего загрузку данных из файла через DBI.

Удаление данных

Теперь, когда вы, не пожалев времени, загрузили свои таблицы данными, следующее, что вам понадобится, – это привести их в порядок и удалить ненужные или избыточные данные.

Определенное в SQL ключевое слово DELETE – это как раз то, что вам нужно. SQL обеспечивает простой синтаксис для удаления из таблиц строк раз и навсегда. Как и в случае команды INSERT, удаление строк применяется одновременно только к одной таблице. Поэтому если вы хотите удалить строки, на которые ссылаются записи в других таблицах, сначала нужно удалить из вторичных таблиц ассоциированные записи с *внешними ключами*. Эта операция, известная как *каскадное удаление*,[1] сохраняет *ссылочную целостность* данных. В некоторых базах данных поддерживается механизм каскадного удаления, автоматически осуществляющий эти дополнительные удаления.

Например, при каскадном удалении строк из таблицы megaliths понадобится также удалить соответствующие строки из таблицы media, для которых выполняется следующее условие соединения:

```
megaliths.id = media.megaliths_id
```

Однако команды DELETE не имеют ограничения «одна строка за одну операцию», которым страдают команды INSERT. Командой DELETE можно за один раз очистить всю таблицу. Например, удалить все строки таблицы megaliths можно следующей командой:

```
DELETE FROM megaliths
```

Разумеется, может оказаться желательным удалить не все, а только некоторые строки таблицы. Это делается знакомым нам образом – заданием списка условий, которым должны удовлетворять данные в удаляемой строке. Поэтому если нужно удалить из таблицы megaliths все строки с площадками, расположенными в Уилтшире, отлично сработает следующая команда:

[1] Этот процесс аналогичен удалению в Unix файла, на который указывают несколько символических ссылок. То же самое можно сказать о системах Windows и Macintosh, в которых существуют ярлыки для документов.

```
DELETE FROM megaliths
WHERE location LIKE '%Уилтшир%'
```

Для того чтобы удалить все строки, относящиеся к каменным кругам, можно сузить критерий, которому должна удовлетворять строка, указав, что тип площадки должен быть каменным кругом. Более жесткий запрос выглядит так:

```
DELETE FROM megaliths
WHERE location LIKE '%Уилтшир%'
AND site_type_id IN = (SELECT id FROM site_types
                       WHERE site_type = 'Каменный круг')
```

Следует обратить внимание на то, что при удалении из таблицы всех строк сама таблица не удаляется из базы данных. Таблица остается на месте, но в ней нет строк.[1]

Более действенным способом определения строк данных, подлежащих удалению, является использование вложенного запроса, возвращающего целевые строки. Хорошим примером таких действий является удаление из таблицы внешних ключей при удалении первичных ключей. Эта операция может быть разбита на две отдельные команды DELETE, первая из которых удаляет строки с внешними ключами, а вторая – с первичными ключами. В следующем примере из обеих таблиц – media и megaliths – удаляются строки, относящиеся к площадкам, расположенным в Уилтшире:

```
DELETE FROM media
WHERE megaliths_id IN (
    SELECT id
    FROM megaliths
    WHERE location LIKE '%Уилтшир%'
  )

DELETE FROM megaliths
WHERE location LIKE '%Уилтшир%'
```

Итак, удаление данных из таблиц благодаря ключевому слову DELETE осуществляется очень просто (возможно, слишком просто!). Далее в этой главе мы более подробно остановимся на процессе удаления с точки зрения базы данных и обсудим немаловажную возможность отмены ошибочно произведенных удалений.

[1] В некоторых базах данных есть более быстрый и удобный способ удаления из таблицы всех строк с помощью ключевого слова TRUNCATE TABLE. Но будьте осторожны: в некоторых базах данных это ключевое слово удаляет и все индексы.

Обновление данных

Последний способ, с помощью которого можно произвести модификацию данных в таблицах, – это осуществление модификации существующих данных по месту путем обновления данных в конкретных колонках конкретных строк. Команда UPDATE не вставляет и не удаляет строки, структура таблицы в результате ее выполнения тоже не меняется.

Команды UPDATE являются очень мощными, позволяя обновлять множественные строки данных с помощью одной команды. При желании новые значения могут быть заданы как возвращаемые командой SELECT, следуя синтаксису команды INSERT.

Самая простая и полезная команда UPDATE изменяет значение колонки в одной строке. Например, если нужно обновить местонахождение «Эйвбери» в строке таблицы megaliths, можно использовать такую команду:

```
UPDATE megaliths
SET location = 'Близ Дивайзис, Уилтшир'
WHERE name = 'Эйвбери'
```

Обратите внимание на условное предложение, указанное в команде. Если бы в команде не проверялось название площадки, то команда UPDATE была бы выполнена с каждой строкой таблицы, что могло бы вызвать катастрофическое повреждение данных. Условные предложения задаются точно таким же способом, как и в других командах SQL, например DELETE и SELECT.

Командой UPDATE можно одновременно обновлять несколько колонок, просто перечисляя обновляемые колонки через запятую. Например, чтобы обновить поля name и description в таблице megaliths, можно написать следующую команду SQL:

```
UPDATE megaliths
SET location = 'Каланиш, Остров Льюис',
    description = 'Сложная площадка в форме изогнутого кельтского креста'
WHERE name = 'Каланиш I'
```

В некоторых системах баз данных можно обновлять несколько колонок одновременно, используя вложенный запрос, возвращающий список значений из другой таблицы. Эти значения используются в качестве новых значений для заданных колонок. Например, если пожелать синхронизировать нашу мегалитическую базу данных с базой данных Туристского совета Уилтшира, то можно воспользоваться такой командой SQL:

```
UPDATE megaliths
SET ( name, location ) =
    ( SELECT tourist_site_name, tourist_site_location
      FROM tourist_sites
```

```
        WHERE tourist_site_name LIKE '%Эйвбери%'
        AND tourist_site_type = 'Каменный круг' )
WHERE name = 'Эйвбери'
```

Эта команда обновляет поля `name` и `location` в таблице `megaliths` значениями, которые возвращает запрос, выполняющийся в другой таблице. Важно отметить, что вложенный запрос должен возвращать только одну строку данных, иначе команда `UPDATE` не выполнится и выдаст ошибку.

Фиксация и откат изменений

Итак, что же происходит при совершении ужасной ошибки во время модификации данных в базе? Является ли единственным выходом уход в отставку? Не надо бояться! Некоторые базы данных предоставляют возможность, называемую *откатом транзакции*, которая позволяет спасти не только вашу голову, но и данные.

Принцип отката весьма прост. При каждом изменении данных в строке ее копия предварительно записывается в журнал, в котором регистрируются все модификации. Если вы решите, что изменения произведены правильно, то можно *зафиксировать* изменения в базе данных. Если окажется, что зафиксированные изменения ошибочны, тогда ваши дела плохи: можете наводить порядок в столе и вспоминать, как пишется резюме.

Однако если по чистой случайности вы найдете ошибку в модифицированных строках до того, как зафиксируете изменения, то можно сделать откат произведенных изменений, возвратив строкам те значения, которые были в них до того, как вы начали их изменять. Ваше рабочее место остается за вами.

Можно порадоваться и тому, что изменения, которые вы делали во время транзакции, не видел никто другой, работавший с базой данных. Поэтому никто не узнает о вашей ошибке, и ваша репутация останется незапятнанной.

Большинство баз данных автоматически фиксирует изменения при отсоединении от базы, если только не выполнен явным образом откат. Поэтому если выполняемая программа не осуществляет должного контроля ошибок при изменении данных, может произойти отсоединение от базы данных и непреднамеренная фиксация ошибочных изменений в базе данных. Вывод из этого только один – *всегда проверяйте наличие ошибок*!

В некоторых базах данных нет такой сложной функции, как откат или undo. В таких случаях еще более важно сделать резервную копию вашей базы данных, прежде чем дать волю драматичным манипуляциям данными с помощью SQL. Создание резервных копий всегда является правильной мыслью, даже если база данных поддерживает транзакции.

Создание и удаление таблиц

В предыдущем разделе обсуждались операции, которые можно осуществлять с помощью SQL над данными, хранимыми в виде строк таблиц базы данных. Однако существует отдельный набор команд для действий с таблицами и другими объектами в самих базах данных. Эти команды известны как *язык определения данных* (*Data Definition Language*), или *DDL*.

Операции, которые можно производить над таблицами, весьма фундаментальны, поскольку имеют далеко идущие последствия. Существуют две наиболее простые операции:

Создание новой таблицы

Осуществляется командой `CREATE TABLE`, синтаксис которой зависит от используемой базы данных. Однако в общем случае в этой команде задаются имя создаваемой таблицы и определения всех колонок таблицы (имена и типы данных).

Например, для создания в нашей базе данных таблицы `megaliths` использовалась следующая команда SQL:

```
CREATE TABLE megaliths (
    id              INTEGER NOT NULL,
    name            VARCHAR(64),
    location        VARCHAR(64),
    description     VARCHAR(256),
    site_type_id    INTEGER,
    mapref          VARCHAR(16)
)
```

`CREATE TABLE` создаст новую таблицу согласно данному определению, которая будет совершенно пуста, пока вы не вставите в нее строки.

Удаление существующей таблицы

Это наиболее радикальная операция модификации данных. Полностью удаляются как все строки данных таблицы, так и сама структура таблицы в базе данных. Обычно эту операцию нельзя отменить. После ввода этой фатальной команды указанная таблица утрачивается навсегда (если только не была сделана резервная копия).

Синтаксис для удаления таблиц является достаточно стандартным в различных базах данных и крайне прост.[1] Чтобы полностью избавиться от таблицы `megaliths`, можно выполнить такую команду SQL:

```
DROP TABLE megaliths
```

[1] У такой смертельной операции должен был быть гораздо более сложный синтаксис!

Существуют и другие способы манипуляций с определениями таблиц, а также другие структуры в базе данных, которые можно создавать (например, представления и индексы), но они находятся за рамками данной книги. Более подробные сведения можно получить в документации по вашей базе данных.

4

Программирование с помощью DBI

В этой главе мы подробно обсудим реальный интерфейс программирования, определенный в модуле DBI. Мы начнем с самой архитектуры DBI, объясним, как использовать дескрипторы (handles), создаваемые DBI для взаимодействия с базами, затем осветим такие простые задачи, как соединение с базами данных и отключение от них. Наконец, мы остановимся на важной теме обработки ошибок и расскажем о некоторых методах и функциях DBI. В последующих главах мы опишем, как обрабатывать данные, находящиеся в базе, а также другие, более сложные функции.

Архитектура DBI

В архитектуре DBI выделяются две большие группы программного обеспечения: собственно DBI и *драйверы*. DBI определяет фактический интерфейс программирования, маршрутизирует вызовы методов к соответствующим драйверам и обеспечивает для них различные вспомогательные сервисы. Для каждого типа базы данных реализован конкретный драйвер, осуществляющий реальные операции в базе. Архитектура DBI проиллюстрирована на рис. 4.1.

При создании программного обеспечения с помощью программного интерфейса DBI используемые методы определены в модуле DBI. Модуль DBI определяет, какой драйвер должен осуществлять выполнение метода, и передает метод соответствующему драйверу для фактического выполнения. Это становится более очевидным, если понять,

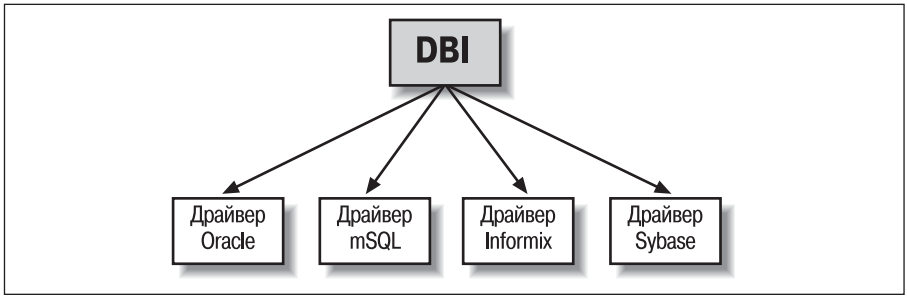

Рис. 4.1. *Архитектура DBI*

что сам модуль DBI не работает с базой данных и даже не знает, какие типы баз данных существуют. На рис. 4.2 показан поток данных от сценария Perl к базе данных.

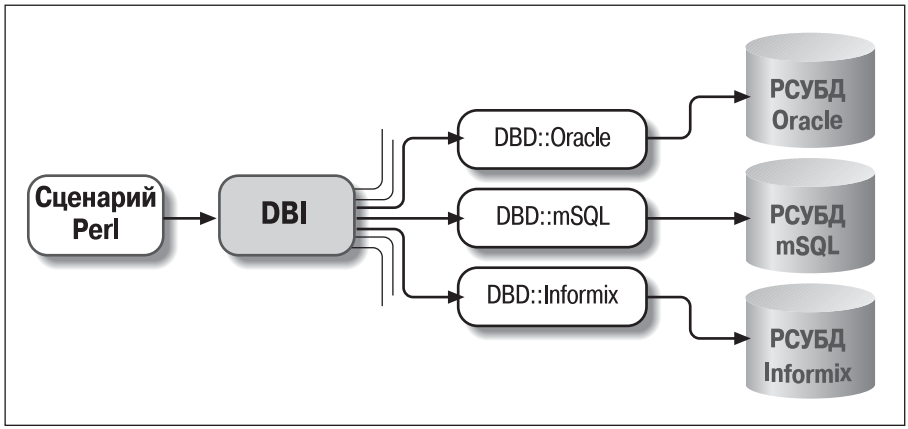

Рис. 4.2. *Поток данных через DBI*

При такой архитектуре довольно просто реализовать драйвер для базы данных любого типа. Все, что требуется – это реализовать методы, определенные в спецификации DBI[1] и поддерживаемые в модуле DBI способом, который поддается интерпретации в базе данных. Возвращаемые этим модулем данные передаются обратно модулю DBI, а из него возвращаются программе на Perl. Все данные, передаваемые между DBI и его драйверами, имеют стандартные типы данных Perl, благодаря чему поддерживается изоляция модуля DBI от необходимости что-либо знать о базах данных.

[1] На практике нужно реализовать небольшое число методов, поскольку для большинства из них в DBI есть подходящие значения по умолчанию. В модуле `DBI::DBD` есть документация для отважных разработчиков драйверов.

Отделение драйверов от собственно DBI превращает DBI в мощный интерфейс программирования, который можно расширить для использования с практически любой имеющейся сегодня базой данных. В настоящее время существуют драйверы для многих распространенных баз данных, в том числе Oracle, Informix, mSQL, MySQL, Ingres, Sybase, DB2, Empress, SearchServer и PostgreSQL. Есть даже драйверы для файлов XBase и CSV.

Эти драйверы можно менять, внося в программы незначительные изменения. Соединение такой переносимости на уровне баз данных с переносимостью сценариев Perl в широком спектре операционных систем дает средство быстрой разработки приложений, достойное внимания.

Драйверы называют также драйверами баз данных, или DBD, согласно пространству имен, в котором они определены. Например, Oracle использует `DBD::Oracle`, Informix использует `DBD::Informix` и т. д. Архитектуру DBI легче запомнить, если считать, что DBI – это сокращение от *D*ata*B*ase *I*ndependent (независимый от базы данных), а DBD – сокращение от *D*ata*B*ase *D*ependent (зависимый от базы данных).

Поскольку в DBI используются объектно-ориентированные функции Perl, инициализировать DBI для использования в своих программах очень просто. Это можно сделать, добавив в верхней части программы строку:

```
use DBI;
```

При наличии этой строки загружается главный модуль DBI. Модули с драйверами отдельных баз данных загружаются по мере надобности. Обычно не требуется загружать их явным образом.

Дескрипторы

В DBI определены три главных типа объектов, которые можно использовать для взаимодействия с базами данных. Эти объекты известны как *дескрипторы*. Есть дескрипторы для драйверов, которые используются DBI для создания дескрипторов соединений с базами данных, которые, в свою очередь, можно использовать для создания дескрипторов отдельных команд базы данных. На рис. 4.3 показана общая структура связей дескрипторов, а их значение описывается в следующих разделах.

Дескрипторы драйверов

Дескрипторы драйверов описывают используемые драйверы и создаются, когда DBI загружает и инициализирует драйвер. Для каждого загруженного драйвера есть в точности один дескриптор. Изначально

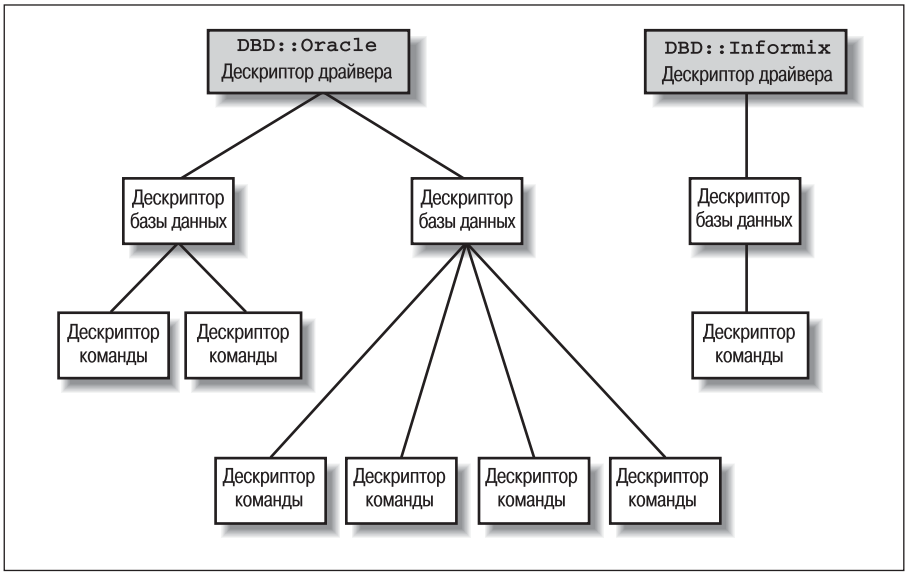

Рис. 4.3. *Дескрипторы DBI*

дескриптор драйвера является единственной связью DBI с драйвером, и на данном этапе с помощью этого драйвера еще не установлена связь ни с одной базой данных.

Единственными существенными методами, доступными через дескриптор драйвера, являются: `data_sources()`, служащий для перечисления объектов, с которыми возможно соединение, и `connect()`, использующийся для установления реального соединения. Однако чаще эти методы вызываются как методы класса DBI, что мы обсудим более подробно далее в этой главе.

Поскольку дескриптор драйвера полностью инкапсулирует драйвер, нет причин, по которым одновременно не может быть загружено несколько драйверов. Это одна из составляющих мощности DBI как интерфейса.

Например, если программисту ставится задача перенесения базы данных Oracle в базу данных Informix, можно написать программу DBI, которая соединяется одновременно с обеими базами данных и легко перемещает между ними необходимые данные. В этом случае создаются два дескриптора драйверов – один для Oracle и второй для Informix. При этом не возникает никаких проблем, поскольку каждый дескриптор драйвера является совершенно самостоятельным объектом Perl.

В спецификации DBI на дескриптор драйвера обычно ссылаются как на `$drh`.

Как правило, в программах не следует ссылаться на дескрипторы драйверов. Фактическое создание экземпляров дескрипторов драйве-

ров происходит «за кулисами» DBI, как правило, при вызове `DBI->connect()`.

Дескрипторы баз данных

Дескрипторы баз данных – это первый шаг к реальной работе с базами данных, поскольку дескриптор инкапсулирует отдельное соединение с конкретной базой данных. Прежде чем выполнять команды SQL для базы данных, нужно получить реальное *соединение* с этой базой. Обычно это осуществляется с помощью метода DBI `connect()`:

```
$dbh = DBI->connect( $data_source, ... );
```

Большинство современных баз данных позволяет работать в многопользовательском режиме, допускающем существование нескольких одновременных соединений. Этому соответствует и архитектура дескрипторов баз данных. Примером может служить приложение для наблюдения за биржевыми котировками, работающее с таблицами базы данных, доступ к которым осуществляется из различных учетных записей пользователей. Сценарий DBI может осуществить несколько соединений с базой данных, по одному для каждой учетной записи пользователя, и в каждом соединении выполнять команды SQL. Дескрипторы баз данных являются полностью инкапсулированными объектами, и это значит, что транзакции из одного дескриптора базы данных не могут «пересечься» или «просочиться» в другой.

Дескрипторы баз данных являются *потомками* соответствующего дескриптора драйвера, что соответствует представлению о реализации множественных одновременных соединений с базами данных как различных типов, так и одного и того же типа. Например, в более сложном сценарии DBI можно создать по два соединения с каждой из баз данных Oracle и Informix для осуществления вышеуказанного управления данными. На рис. 4.3 проиллюстрирована возможность наличия нескольких дескрипторов базы данных для соединения с базой данных Oracle через дескриптор драйвера.

Заметьте, что если бы программа была написана на C, то потребовалось бы два экземпляра кода: один для программного интерфейса Oracle и другой для Informix. DBI выравнивает игровое поле.

В спецификации DBI и примерах программ на дескрипторы баз данных обычно ссылаются как на `$dbh`.

Дескрипторы команд

Дескрипторы команд – это последний тип объектов, которые определяются в DBI для взаимодействия с базами и обработки данных. В

этих дескрипторах инкапсулируются отдельные команды SQL, которые должны выполняться в базе данных.

Дескрипторы команд являются *потомками* соответствующих дескрипторов баз данных. Поскольку дескрипторы команд являются самостоятельными объектами, данные внутри одной команды защищены от вмешательства других обработчиков команд.

Для каждого дескриптора базы данных практически не существует предела количества дескрипторов команд, которые можно создать и выполнить.[1] В одном сценарии можно создать и выполнить много команд и обработать возвращаемые данные. Хорошим примером может служить робот, осуществляющий «добычу данных». Он соединяется с базой данных и выполняет большое количество запросов, возвращающих разнообразную информацию. Вместо написания сложного кода SQL для поиска корреляции в данных сценарий Perl получает данные, возвращаемые многими запросами, и осуществляет их анализ, используя полнофункциональные процедуры обработки текста и данных, предоставляемые Perl.

В спецификации DBI и примерах программ на дескрипторы команд обычно ссылаются как на `$sth`.

Имена источников данных

При соединении с базой данных через DBI ему необходимо указать, где находится база данных, с которой нужно соединиться. Например, драйверу базы данных может потребоваться имя базы данных или имя физической машины, на которой располагается база данных. Эти сведения называются *именем источника данных* (*data source name*) и являются, вероятно, наиболее трудной для стандартизации стороной DBI по причине большого многообразия синтаксиса соединений.

DBI требует, чтобы имя источника данных начиналось с символов `dbi:` (подобно тому, как URL начинается с `http:`), за которыми должно следовать имя драйвера и еще одно двоеточие, например `dbi:Oracle:`. Любой последующий текст передается собственному методу драйвера `connect()`, который, в свою очередь, интерпретирует этот текст соответствующим образом. Большинство драйверов рассчитывает получить просто имя базы данных либо, чаще всего, одну или несколько пар имя/значение, разделенных двоеточиями. Несколько примеров часто встречающихся случаев приведены далее в этой главе.

[1] На практике количество одновременно существующих дескрипторов команд зависит от базы данных, с которой производятся действия. Данные о том, сколько дескрипторов команд одновременно поддерживает ваша база данных, можно найти в приложении B.

Например, для mSQL требуется передать имя узла, имя базы данных и, возможно, номер порта TCP/IP для соединения с сервером базы данных. Для Oracle же может потребоваться лишь одно слово, являющееся псевдонимом более сложного идентификатора соединения, хранящегося в отдельных файлах конфигурации Oracle.

DBI предлагает два полезных метода, позволяющих узнать, какие источники данных доступны для каждого драйвера, установленного в системе.

Во-первых, можно получить список всех драйверов, установленных на машине, с помощью метода `DBI->available_drivers()`. Каждый элемент возвращаемого списка начинается с префикса установленного драйвера,[1] например `dbi:Informix:`.

Во-вторых, можно вызвать метод `DBI->data_sources()` одного или нескольких драйверов, возвращенных методом `DBI->available_drivers()`, чтобы перечислить источники данных, известные этому драйверу.[2] Обращение к методу `data_sources()` влечет фактическую загрузку указанного драйвера и проверку того, что он установлен полностью и корректно. Поскольку DBI при невозможности загрузить и инициализировать драйвер прекращает свою работу, этот метод следует вызывать внутри блока `eval{}`, чтобы ошибку можно было перехватить.

Следующий сценарий перечисляет все драйверы и все источники данных для каждого драйвера в системе:

```
#!/usr/bin/perl -w
#
# ch04/listdsns: Перечисляет все источники данных
# и все установленные драйверы

use DBI;

### Проверить, какие драйверы установлены в DBI
my @drivers = DBI->available_drivers();

die "Драйверы не обнаружены!\n" unless @drivers; # этого быть не должно

### Проход по всем драйверам и вывод списка имен источников данных
### для каждого из них
```

[1] В действительности определение «установленный драйвер» является несколько вольным. DBI лишь просматривает каталоги из `@INC` в поиске подкаталогов DBD, содержащих файлы с расширением *.pm*. Предполагается, что такие файлы являются драйверами. Он не проверяет, установлены ли эти модули правильно и полностью. На практике такая процедура работает быстро и успешно.

[2] Учтите, что могут быть возвращены не все источники данных, доступные через этот драйвер. И наоборот, перечисление источника данных не обязательно означает, что в данное время он доступен для соединения.

```
foreach my $driver ( @drivers ) {
    print "Драйвер: $driver\n";
    my @dataSources = DBI->data_sources( $driver );
    foreach my $dataSource ( @dataSources ) {
        print "\tИсточником данных является $dataSource\n";
    }
    print "\n";
}

exit;
```

На моей машине этот сценарий выводит следующие данные:

```
Драйвер: ADO

Драйвер: CSV
    Источником данных является DBI:CSV:f_dir=megaliths
    Источником данных является DBI:CSV:f_dir=pictish_stones

Драйвер: ExampleP
    Источником данных является dbi:ExampleP:dir=.

Драйвер: File
    Источником данных является DBI:File:f_dir=megaliths
    Источником данных является DBI:File:f_dir=pictish_stones

Драйвер: ODBC

Драйвер: Proxy

Драйвер: XBase
    Источником данных является dbi:XBase:.
```

Это говорит о том, что установлены стандартные драйверы `DBD::Proxy`, `DBD::ADO`, `DBD::File` и `DBD::ExampleP`, а также `DBD::ODBC`, `DBD::Xbase` и `DBD::CSV`.

Теоретический интерес это может представлять, но на практике вам редко понадобится использовать эти методы. Большинство приложений пишется в расчете на один источник данных, который либо жестко задан в коде приложения, либо каким-то образом передается ему в качестве параметра.

При указании источника данных для базы текст, следующий за префиксом драйвера, должен иметь формат, соответствующий конкретному типу базы данных, с которой вы хотите соединиться. Он является сугубо специфичным для каждой базы данных, но в следующей таблице приведено несколько примеров.[1]

[1] Отличным примером приложения, получающего имена источников данных во время исполнения, является `dbish`, о котором более подробно рассказывается в главе 8.

База данных	Пример синтаксиса соединения

mSQL	`dbi:mSQL:hostname:database:port_number`
	Например, для соединения с базой данных под именем *archaeo* на машине с именем *fowliswester.arcana.co.uk* на порту с номером 1114 следует использовать аргумент `$data_source`:
	`dbi:mSQL:fowliswester.arcana.co.uk:archaeo:1114`

Oracle	`dbi:Oracle:connection_descriptor`
	В Oracle существует менее шаблонный способ указания идентификаторов соединения из-за того, что программное обеспечение базы данных Oracle может обрабатывать соединения различными способами.
	Если об этой кошмарной теме сказать в двух словах, то Oracle может использовать два типа соединений. Для локальных соединений Oracle использует единственный элемент данных, которым может быть дескриптор соединения, имя базы данных или псевдоним базы данных, определенный в файлах конфигурации Oracle. Для сетевых соединений Oracle обычно требуется знать псевдоним дескриптора соединения, заданный в файлах конфигурации Oracle, либо, если вы склонны к мазохизму, можно задать дескриптор соединения целиком… Но, поверьте мне, это малоприятно.
	Например, простым значением `$data_source` для Oracle может быть следующее:
	`dbi:Oracle:archaeo`

CSV	`dbi:CSV:f_dir=/datafiles` Модуль `DBD::CSV` рассматривает в качестве базы данных расположенную в одном каталоге группу файлов со значениями, разделяемыми запятой. Источник данных для этого драйвера может содержать параметр `f_dir`, указывающий каталог, в котором расположены эти файлы.

Если аргумент `$data_source` пуст или содержит неопределенное значение, то допустимое значение ищется в переменной окружения `DBI_DSN`. Если эта переменная не определена или содержит недопустимое значение, DBI вызывает функцию `die()`.

Соединение и отсоединение

Основная работа в программировании баз данных обычно состоит в выполнении команд SQL в базе данных. Однако для выполнения этой задачи сначала должно быть установлено *соединение*. Более того, признаком хорошего тона считается *отсоединение* от базы данных по

окончании работы, чтобы освободить как ресурсы своей машины, так и, что более важно, дорогостоящие ресурсы базы данных.

Соединение

В случае простых баз данных, таких как плоские файлы или файлы Berkeley DB, «соединение» обычно состоит лишь в том, чтобы открыть файлы на чтение или использовать механизм связывания. Однако в более крупных системах баз данных соединение бывает значительно более сложным.

Относительно простой является РСУБД mSQL, метод соединения с которой не сложен: для соединения программа подключается к порту TCP/IP на том компьютере, на котором работает база данных. В результате устанавливается активное соединение с базой данных. Однако в более сложных системах, таких как Oracle, при соединении нужно выполнить значительно большую работу по обеспечению безопасности и выполнению служебных функций. Кроме того, в таких системах программа должна сообщить при соединении значительно больше данных, например, имя пользователя, под которым вы хотите подключиться, и пароль.

При рассмотрении широкого спектра систем баз данных сведения, необходимые для соединения, можно свести к следующим составляющим:

1. *Имя источника данных* – строка с указанием драйвера, который нужно использовать, базы данных, с которой нужно соединиться, и, возможно, ее местонахождения. Формат этого аргумента в значительной мере зависит от типа базы данных и обсуждался в предыдущем разделе.
2. *Имя пользователя*, под которым вы хотите соединиться с базой данных. Если коснуться понятия имен пользователей несколько глубже, то нужно отметить, что в некоторых системах база данных разделяется на отдельные области, называемые схемами, в которых различные пользователи могут создавать таблицы и обрабатывать данные. Одни пользователи не могут изменять таблицы или данные, созданные другими пользователями. Такая настройка аналогична системе учетных записей в многопользовательских компьютерных системах, где пользователям разрешено создавать собственные файлы, с которыми они могут работать, но не обязательно доступные другим пользователям. На практике пользователи могут запретить всем остальным доступ к своим файлам или таблицам либо разрешить доступ избранным группам или всем пользователям.[1]

[1] Обычно это соблюдается. Однако в некоторых системах баз данных, например в MySQL, поддерживается множество пользователей, но только одна схема.

Как правило, в больших системах баз данных осуществляется аналогичная политика безопасности, при этом обычно администратор имеет доступ к учетной записи, позволяющей читать, изменять и удалять таблицы и данные любого пользователя. Все остальные пользователи должны соединяться под своими именами. При этом имя пользователя для соединения с базой данных не обязательно должно быть таким же, как имя регистрации в системе.

В более ограниченных системах баз данных может отсутствовать понятие аутентификации, основанной на имени пользователя, тем не менее аргументы имени пользователя и пароля должны присутствовать, чаще всего в виде пустых строк.

3. *Пароль*, относящийся к указанному имени пользователя.

В свете этих общих аргументов синтаксис для соединения с базами данных при использовании DBI состоит в вызове `connect()`, определяемом так:

```
$dbh = DBI->connect( $data_source, $username, $password, \%attr );
```

Последний аргумент — `\%attr` — является необязательным и может быть опущен. `\%attr` является ссылкой на хэш, содержащий атрибуты дескриптора, применяемые в этом соединении. Один из важнейших элементов данных, передаваемый в этом хэше, указывает, должен ли DBI предоставлять автоматическую обработку ошибок. Подробнее мы обсудим это в следующем разделе, но сейчас отметим, что обычно задаются два атрибута с именами `RaiseError` и `PrintError`, которые определяют, должен ли DBI прекратить выполнение или автоматически вывести предупреждение при обнаружении ошибки базы данных.

Этот метод возвращает при своем вызове дескриптор базы данных, если соединение было успешно установлено, либо значение `undef` в случае неудачи.

Чтобы проиллюстрировать использование метода `DBI->connect()`, предположим, что имеется база данных Oracle с именем `archaeo`. Для соединения с этой базой данных можно использовать следующий код:

```
#!/usr/bin/perl -w
#
# ch04/connect/ex1: Осуществляет соединение с базой данных Oracle.

use DBI;            # Загрузка модуля DBI

### Осуществить соединение с помощью драйвера Oracle
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
or die "Невозможно соединение с базой данных Oracle: $DBI::errstr\n";

exit;
```

Этот простой пример иллюстрирует использование метода `DBI->connect()` для установления одного соединения с базой данных. Мы также производим проверку ошибок при вызове, чтобы убедиться, что соединение произведено. В случае неудачи выводится сообщение об ошибке, а также специфическая для базы данных причина отказа, содержащаяся в переменной `$DBI::errstr`.[1]

Приведем более сложный пример, в котором выполняются два соединения с одной и той же базой данных в одном сценарии:

```
#!/usr/bin/perl -w
#
# ch04/connect/ex2: Осуществляет одновременное соединение с двумя
#                   базами данных Oracle с одинаковыми аргументами.
#                   Это делается для того, чтобы показать, что все
#                   дескрипторы баз данных, даже с одинаковыми
#                   аргументами, совершенно независимы один от другого.

use DBI;            # Загрузить модуль DBI

### Осуществить соединение с помощью драйвера Oracle
my $dbh1 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Невозможно осуществить первое соединение с базой данных:
            $DBI::errstr\n";

my $dbh2 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Невозможно осуществить второе соединение с базой данных:
            $DBI::errstr\n";

exit;
```

А вот пример одновременного соединения с двумя различными базами данных:

```
#!/usr/bin/perl -w
#
# ch04/connect/ex3: Осуществляет одновременное соединение
#                   с двумя различными базами данных Oracle

use DBI;            # Загрузить модуль DBI

### Осуществить соединение с помощью драйвера Oracle
my $dbh1 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Невозможно осуществить соединение с первой
    базой данных Oracle: $DBI::errstr\n";

my $dbh2 = DBI->connect( "dbi:Oracle:seconddb", "username", "password" )
```

[1] На самом деле сообщение об ошибке будет выведено дважды по причинам, которые будут объяснены в разделе «Обработка ошибок» далее в этой главе.

```
    or die "Невозможно осуществить соединение со второй
    базой данных Oracle: $DBI::errstr\n";

exit;
```

Первый пример достаточно интересен, поскольку даже если мы используем одинаковые аргументы в `DBI->connect()`, два создаваемых дескриптора совершенно различны и не имеют общих данных.

Последний пример использования `DBI->connect()` демонстрирует соединение с двумя различными базами данных (одна из них Oracle, другая – mSQL) из одного сценария. В этом случае механизм DBI автоматического вывода сообщений об ошибках отключается для базы данных mSQL путем передачи методу `connect()` хэша с атрибутами, как показано ниже:

```
#!/usr/bin/perl -w
#
# ch04/connect/ex4: Одновременное соединение с двумя базами данных -
#                   Oracle и mSQL. В дескрипторе mSQL отключено
#                   автоматическое сообщение об ошибках.

use DBI;            # Загрузить модуль DBI

### Осуществить соединение с помощью драйвера Oracle
my $dbh1 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
or die "Невозможно соединение с базой данных Oracle: $DBI::errstr\n";

my $dbh2 = DBI->connect( "dbi:mSQL:seconddb", "username", "password" , {
            PrintError => 0
        } )
or die "Невозможно соединение с базой данных mSQL: $DBI::errstr\n";

exit;
```

Аргументы `$username` и `$password` должны быть заданы, но могут быть пустыми (`''`), если они не требуются. Как уже говорилось, аргумент `$data_source` тоже может быть неопределенным, и в этом случае используется значение переменной окружения `DBI_DSN`, если она установлена.

Отсоединение

Явное отключение от базы данных не является строго обязательным при выходе из вашей программы после завершения всей работы, но будет правильным поступком. Мы весьма рекомендуем, чтобы явное отсоединение от базы данных вошло у вас в привычку.

DBI предоставляет метод, с помощью которого программист может отсоединить заданный дескриптор базы данных от его базы данных. Его

полезно использовать, особенно в программах, в которых осуществлено несколько соединений или будет выполнено несколько последовательных соединений.

Метод, осуществляющий отсоединение, выглядит так:

```
$rc = $dbh->disconnect();
```

Согласно этому определению `disconnect()` вызывается относительно заданного дескриптора базы данных. При этом сохраняется полная независимость дескрипторов баз данных. Если одновременно действуют несколько дескрипторов баз данных, то каждый из них должен быть отсоединен явным образом.

Приведем пример использования метода `disconnect()`:

```
#!/usr/bin/perl -w
#
# ch04/disconnect/ex1: Соединение с базой данных Oracle с отключением
#                      автоматического вывода сообщений об ошибках,
#                      затем отсоединение явным образом.

use DBI;            # Загрузить модуль DBI

### Осуществить соединение с использованием драйвера Oracle
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
            PrintError => 0
        } )
    or die "Невозможно соединиться с базой данных Oracle:
    $DBI::errstr\n";

### Теперь отсоединиться от базы данных
$dbh->disconnect
    or warn "Отсоединение не выполнено: $DBI::errstr\n";

exit;
```

При успешном отсоединении возвращается значение true, в противном случае – false. На практике неудача при отсоединении означает, что соединение уже по какой-то причине потеряно. После отсоединения дескриптор базы данных уже нельзя использовать для получения каких-либо результатов.

Что произойдет, если не отсоединиться явным образом? Поскольку дескрипторы DBI являются ссылками на объекты Perl, вступит в действие собственный сборщик мусора Perl, который подчистит все брошенные объекты. Он делает это, вызывая метод объекта `DESTROY`, когда в сценарии не остается больше ссылок на объект или Perl завершает свою работу.

Для дескриптора базы данных метод `DESTROY` вызовет `disconnect()`, чтобы произвести аккуратное отсоединение, если дескриптор соединен с

базой данных. Однако при этом он пожалуется на то, что ему пришлось это сделать, выдав сообщение:

```
Database handle destroyed without explicit disconnect.
(Дескриптор базы данных уничтожен без явного отсоединения.)
```

Основное предупреждение, которое нужно сделать в отношении метода `disconnect()`, касается автоматической фиксации транзакций при отсоединении. Например, если программа обновила данные, но не вызвала `commit()` или `rollback()` перед обращением к `disconnect()`, то различные базы данных будут вести себя по-разному. Oracle автоматически зафиксирует модификацию данных, в то время как Informix может этого не сделать. Для правильной работы метод `DESTROY` должен вызывать `rollback()` перед `disconnect()`, если режим `AutoCommit` не включен. В главе 6 мы более подробно обсудим действие `disconnect()` и `DESTROY` в транзакциях.

Обработка ошибок

Обработка ошибок внутри программ или отсутствие ее весьма часто вызывает вопросы при программировании в DBI. Когда кто-то спрашивает, почему не работает его программа, ответ обычно заключается в том, что нужно обрабатывать ошибки. Можно с уверенностью утверждать, что в девяти случаях из десяти при добавлении проверки ошибок выводится соответствующее сообщение об ошибке, и причина ее становится очевидной.

Сравнение автоматической и ручной проверки ошибок

В ранних версиях DBI программист должен был производить собственную проверку ошибок традиционным образом, аналогичным тому, как производилось соединение с базой данных в приведенных ранее примерах. После каждого метода, возвращающего тот или иной индикатор успеха или неудачи при его выполнении, должна была следовать команда, проверяющая наличие ошибки. Это хороший способ программирования, несколько напоминающий C, но он достаточно быстро утомляет, и возрастает соблазн опустить проверку ошибок.

Сейчас в DBI есть значительно более простая возможность проверки ошибок в стиле *исключительных ситуаций* (*exceptions*). Это означает, что если DBI обнаруживает ошибку собственными средствами, то может автоматически вывести предупреждение или завершить работу, выполнив `warn()` или `die()` с соответствующим сообщением. Тем самым бремя проверки ошибок перекладывается с плеч программиста на сам

DBI, который осуществляет эту работу надежным и неутомительным образом, как вам того и хотелось.

Ручную проверку ошибок осуществляют в некоторых приложениях, в которых предполагается возникновение частых неудач. Например, при неудаче соединения программа может обнаружить ошибку, сделать пятиминутную паузу и автоматически попытаться еще раз установить соединение. При автоматической же проверке программа просто завершит работу, сообщив о невозможности установить соединение.

DBI позволяет осуществлять проверку ошибок в смешанном стиле путем избирательного включения или отключения автоматической проверки для каждого дескриптора.

Ручная проверка ошибок

Разумеется, DBI по-прежнему позволяет вручную осуществлять проверку ошибок в программах при выполнении методов DBI. Такой способ проверки ошибок более присущ классическому программированию на C и Perl, когда проверяется успешность выполнения каждого важного оператора, позволяя программе осуществлять маневр в случае неудачи.

По умолчанию DBI осуществляет автоматическую базовую проверку ошибок в результате активации атрибута `PrintError`. Чтобы отключить эту функцию, установите значение этого атрибута равным `0` в дескрипторе, после того как он создан, либо, в случае дескриптора базы данных, через хэш атрибутов метода `connect()`.

Например:

```
### Атрибуты, передаваемые DBI->connect()
%attr = (
    PrintError => 0,
    RaiseError => 0
);

### Соединение...
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" ,
          \%attr );

### Включить автоматическую обработку ошибок уровня предупреждений...
$dbh->{PrintError} = 1;
```

Большинство методов DBI возвращает при неудаче выполнения значение ошибочного состояния, как правило `undef`. В Perl его легко проверить:

```
### Попытка соединения с базой данных
my $dbh = DBI->connect( ... )
    or die "Невозможно соединиться с базой данных: $DBI::errstr!\";
```

В следующей программе отключена автоматическая обработка ошибок. Для выявления ошибок служат наши собственные проверки. В этом примере установка атрибутов перемещена в сам метод `connect()` — аккуратный стиль, который часто используется:

```
#!/usr/bin/perl -w
#
# ch04/error/ex1: Небольшой пример ручной обработки ошибок.

use DBI;             # Загрузить модуль DBI

### Осуществить соединение с помощью драйвера Oracle
my $dbh = DBI->connect( undef, "stones", "stones", {
    PrintError => 0,
    RaiseError => 0
} ) or die "Невозможно соединиться с базой данных: $DBI::errstr\n";

### Подготовить к выполнению команду SQL
my $sth = $dbh->prepare( "SELECT * FROM megaliths" )
    or die "Невозможно подготовить команду SQL: $DBI::errstr\n";

### Выполнить команду в базе данных
$sth->execute
    or die "Невозможно выполнить команду SQL: $DBI::errstr\n";

### Извлечь возвращенные строки данных
my @row;
while ( @row = $sth->fetchrow_array() ) {
    print "Строка: @row\n";
}
warn "Получение данных прервано с ошибкой: $DBI::errstr\n"
    if $DBI::err;

### Отсоединиться от базы данных
$dbh->disconnect
    or warn "Ошибка при отсоединении: $DBI::errstr\n";

exit;
```

Как можно видеть из примера, код для проверки наличия ошибок при выполнении методов DBI фактически длиннее, чем код для выполнения самих действий с данными. Кроме того, вполне может случиться, что вы просто забудете написать проверку после команды, в результате чего программа может повести себя крайне странным образом, выдавая непонятные ошибки, а вы потратите на отладку многие часы!

Автоматическая проверка ошибок

Автоматическая проверка ошибок в DBI действует на двух уровнях. Атрибут дескриптора `PrintError` предписывает DBI вызывать функцию Perl `warn()` (что обычно имеет следствием вывод на экран возника-

ющих ошибок), а атрибут дескриптора RaiseError предписывает DBI вызывать при возникновении ошибки функцию Perl die() (в результате чего выполнение сценария обычно немедленно прекращается).

Поскольку используются стандартные функции Perl warn() и die(), результат действия PrintError и RaiseError можно изменить с помощью обработчиков сигналов $SIG{__WARN__} и $SIG{__DIE__}. Аналогично функцию die(), выполняемую при установке RaiseError, можно перехватить с помощью eval {...}.

Эти уровни автоматической проверки ошибок могут быть включены в любом дескрипторе, хотя чаще всего и с большей пользой они применяются в дескрипторах баз данных. Для активации требуемого стиля автоматической проверки ошибок можно установить значение каждого из следующих атрибутов:

```
$h->{PrintError} = 1;
$h->{RaiseError} = 1;
```

Аналогично, чтобы отключить автоматическую проверку ошибок, нужно установить значение этих атрибутов в 0.

Если установлены оба атрибута – RaiseError и PrintError, то при возникновении ошибки будут поочередно выполнены warn() и die(). Если не задан обработчик $SIG{__DIE__}, выполнение warn() опускается, чтобы не выводить сообщение об ошибке дважды.[1]

Эти атрибуты чаще указываются в необязательном хэше атрибутов, передаваемом DBI->connect() при соединении с базой данных. Рекомендованным стилем написания программ с использованием DBI является автоматическая проверка ошибок, поэтому в DBI->connect() атрибут PrintError по умолчанию установлен. Можете считать это подспорьем для новичков и авторов сценариев, создаваемых на скорую руку. Те, кто работает более серьезно, обычно устанавливают RaiseError или сбрасывают PrintError и проверяют ошибки самостоятельно.

В следующем коротком примере показано, как использовать RaiseError вместо ручной проверки ошибок:

```
#!/usr/bin/perl -w
#
# ch04/error/ex2: Небольшой пример с использованием автоматической
#                 обработки ошибок с RaiseError, т.е. выполнение
#                 программы прекращается при обнаружении любых ошибок.

use DBI;            # Загрузить модуль DBI

my ($dbh, $sth, @row);
```

[1] В будущих версиях может измениться порядок действий при установке обоих атрибутов. Это нужно учитывать, если код помещен внутрь eval.

```
### Осуществить соединение с помощью драйвера Oracle
$dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    PrintError => 0,    ### Не сообщать об ошибках через warn()
    RaiseError => 1     ### Сообщать об ошибках через die()
} );

### Подготовить команду SQL для выполнения
$sth = $dbh->prepare( "SELECT * FROM megaliths" );

### Выполнить команду в базе данных
$sth->execute();

### Извлечь возвращенные строки данных
while ( @row = $sth->fetchrow_array() ) {
    print "Строка: @row\n";
}

### Отсоединиться от базы данных
$dbh->disconnect();

exit;
```

Этот пример читается короче и легче, чем предыдущий пример с ручной проверкой ошибок. Логика программы более понятна. Наиболее очевидное дополнительное преимущество состоит в том, что можно забыть о ручной обработке ошибок после операций DBI, поскольку DBI сделает эту проверку за нас.

Смешанная проверка ошибок

В одной программе можно применять смешанные стили проверки ошибок, поскольку автоматическую проверку можно легко включать и выключать для каждого дескриптора. Смешанная проверка оказывается полезной во многих случаях. Допустим, к примеру, что ваша программа работает непрерывно, каждую пару минут опрашивая базу данных в поисках новых биржевых котировок.

И вот происходит катастрофа: крах базы данных! В идеальном случае при неудаче очередного соединения с базой данных программа не прекращает работу, а ждет несколько минут и повторяет попытку. После соединения с базой данных проверка ошибок должна просто извещать о неудачном выполнении команды, а не завершать работу программы.

Эту смешанную проверку ошибок можно разбить на две части: ручная проверка для вызова `DBI->connect()` и автоматическая проверка через установленный атрибут `PrintError` для всех остальных команд. Это иллюстрируется следующим примером:

```
#!/usr/bin/perl -w
#
```

```
# ch04/error/mixed1: Демонстрация смешанного режима проверки ошибок.

use DBI;            # Загрузить модуль DBI

### Атрибуты, которые нужно передать DBI->connect() для отключения
### автоматического контроля ошибок
my %attr = (
    PrintError => 0,
    RaiseError => 0,
);

### Программа продолжает выполняться еще, еще и еще...
while ( 1 ) {
    my $dbh;

    ### Попытка соединения с базой данных. Если соединиться
    ### не удалось, сделать паузу и повторить попытку ...
    until (
        $dbh = DBI->connect( "dbi:Oracle:archaeo", "username",
               "password", \%attr )
    ) {
        warn "Невозможно соединиться: $DBI::errstr. Пауза перед
              повторной попыткой.\n";
        sleep( 5 * 60 );
    }

    eval {      ### Перехват _всех_ отказов внутри кода

        ### Установить автоматический контроль ошибок
        ### для дескриптора базы данных
        $dbh->{RaiseError} = 1;

        ### Подготовить команду SQL к выполнению
        my $sth = $dbh->prepare( "SELECT stock, value FROM
        current_values" );

        while (1) {

            ### Выполнить команду в базе данных
            $sth->execute();

            ### Извлечь возвращенные строки данных
            while ( my @row = $sth->fetchrow_array() ) {
                print "Строка: @row\n";
            }

            ### Ждем изменения котировок
            sleep 60;
        }

    };
```

```
        warn "Мониторинг прерван с ошибкой: $@\n" if $@;

        ### Короткая пауза, чтобы не тревожить базу данных
        sleep 5;
    }

    exit;
```

Эта программа показывает, что в DBI можно наряду с автоматической проверкой ошибок легко написать код для явной проверки ошибок и восстановления.

Диагностика ошибок

Возможность перехвата ошибок в DBI очень полезна как при автоматическом, так и при ручном контроле, но сами по себе выводимые сведения не имеют большой ценности. Истинно полезной является возможность определить, в чем состоит ошибка, чтобы проследить ее и исправить.

С этой целью в DBI определены несколько методов диагностики ошибок, которые можно использовать со всеми допустимыми дескрипторами, драйверами, базами данных и командами. Эти методы сообщают программисту код ошибки и текстовые данные от последнего вызывавшегося метода DBI. Таковыми методами являются:

```
$rv  = $h->err();
$str = $h->errstr();
$str = $h->state();
```

Эти методы возвращают следующие элементы данных, которые можно использовать для более точной отладки ошибок:

- `$h->err()` возвращает номер ошибки, связанный с текущей ошибкой, отмеченной в дескрипторе `$h`. Возвращаемые значения целиком определяются соответствующей СУБД. В некоторых системах может не поддерживаться передача значимой информации. Например, ошибки mSQL всегда имеют значение `-1`. Oracle немного любезнее: отказ в соединении может выставить сообщение об ошибке ORA-12154, для которой вызов `$h->err()` возвращает значение `12154`. Хотя обычно это значение является числом, не следует думать, что это всегда будет так.
- `$h->errstr()` является несколько более полезным методом, поскольку возвращает строку, содержащую описание ошибки, предоставляемое базой данных. Строка должна соответствовать номеру ошибки, возвращаемому `$h->err()`.

  Например, mSQL в качестве номера ошибки всегда возвращает `-1`, что не является особенно информативным. Однако вызов `$h->errstr()` предоставляет значительно более полезные данные. При

ошибке соединения с базой данных может быть возвращено сообщение:

```
ERROR : Can't connect to local MSQL server
```

В Oracle при неудачном соединении с кодом ошибки `12154` будет возвращено следующее описание ошибки:

```
ORA-12154: TNS:could not resolve service name (DBD ERROR: OCIServerAttach)
```

- `$h->state()` возвращает строку в стандартном формате SQLSTATE из пяти символов. Многие драйверы не полностью поддерживают этот метод и при его вызове возвращают значение:

  ```
  S1000
  ```

  Особый код общего успеха операции `00000` транслируется в `0`, поэтому если не выставлена ошибка, этот метод возвращает значение `false`.

DBI сбрасывает сведения об ошибке для конкретного дескриптора перед вызовом большинства своих методов. Поэтому важно проверять ошибки выполнения методов, прежде чем вызывать новые методы для того же самого дескриптора. Если сведения об ошибках могут понадобиться позднее, необходимо самостоятельно их сохранять.

Перепишем предыдущий пример, чтобы показать, как можно использовать специальные методы дескриптора для сообщения об ошибках:

```
#!/usr/bin/perl -w
#
# ch04/error/ex3: Небольшой пример с использованием ручной проверки
#                 ошибок, в котором применяются специальные методы
#                 дескрипторов для сообщения об ошибках.

use DBI;             # Загрузить модуль DBI

### Атрибуты, передаваемые DBI->connect() для отключения
### автоматической проверки ошибок
my %attr = (
    PrintError => 0,
    RaiseError => 0,
);

### Осуществить соединение с помощью драйвера Oracle
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , \
%attr )
    or die "Невозможно соединение с базой данных: ", $DBI::errstr,"\n";

### Подготовить команду SQL к выполнению
my $sth = $dbh->prepare( "SELECT * FROM megaliths" )
    or die "Невозможно подготовить команду SQL: ", $dbh->errstr(), "\n";
### Выполнить команду в базе данных
```

```
$sth->execute
    or die "Невозможно выполнить команду SQL: ", $sth->errstr(), "\n";

### Извлечь возвращенные строки данных
while ( my @row = $sth->fetchrow_array() ) {
    print "Строка: @row\n";
}
warn "Проблемы при вызове fetchrow_array(): ", $sth->errstr(), "\n"
    if $sth->err();

### Отсоединиться от базы данных
$dbh->disconnect
    or warn "Неудача при отсоединении: ", $dbh->errstr(), "\n";

exit;
```

Как видите, это еще более скучная процедура, чем при использовании переменной `$DBI::errstr`, которую можно хотя бы прямо вставлять в сообщения об ошибках.

Помимо этих трех методов, которые позволяют вести тонкую обработку ошибок на уровне дескрипторов, есть три соответствующие переменные, содержащие те же сведения, но на уровне класса DBI:

```
$DBI::err
$DBI::errstr
$DBI::state
```

Эти переменные используются, по сути, так же, как `$h->err()` и другие, но значения относятся к *последнему использовавшемуся дескриптору*. Особенно удобно вставлять их в строки с сообщениями об ошибках.

Поскольку эти переменные связаны с последним дескриптором, использовавшимся в DBI, их срок жизни еще короче, чем у методов обработки ошибок дескрипторов, и пользоваться ими нужно сразу после вызова метода, приведшего к неудаче. В противном случае весьма вероятно, что они будут содержать информацию об ошибках, вводящую в заблуждение.

Очень удобно использовать переменные для анализа ошибок соединения. При возникновении этих ошибок дескриптор, из которого можно получить информацию об ошибке, не возвращается. Поскольку в сценариях не используются внутренние дескрипторы драйверов, переменная `$DBI::errstr` дает очень простой и эффективный способ получить сообщение об ошибке при неудачном выполнении `connect()`.

Итак, для большинства приложений рекомендуется использование автоматического контроля ошибок с помощью атрибутов `RaiseError` и/или `PrintError`. Либо можно использовать ручную проверку и без труда вставлять `$DBI::errstr` в сообщения. Для более сложных приложений можно использовать методы дескрипторов.

Вспомогательные методы и функции

Завершая базовое введение в DBI, мы расскажем о некоторых вспомогательных методах и функциях, которые несколько облегчат вашу жизнь. В их число входят очень полезный метод *обработки кавычек* (*quote escaping*), трассировка выполнения DBI и различные функции для приведения в порядок данных.

Специфическая обработка кавычек

Без сомнения, самым важным вспомогательным методом является `quote()`, который расставляет кавычки и управляющие символы в командах SQL так, как это принято в ядре базы данных, с которым ведется работа. Эта функция приобретает важность, если у вас есть строка Perl, которую нужно вставить в базу данных, поскольку в большинстве случаев данные должны быть заключены в кавычки.

Ситуация осложняется тем, что различные базы данных требуют использования разного формата этих внешних кавычек. DBI решает эту проблему путем объявления метода `quote()`, который выполняется применительно к дескриптору базы данных, что обеспечивает применение корректных правил расстановки кавычек.

Этот метод, выполненный с дескриптором базы данных, преобразует строку, переданную в качестве аргумента, согласно определенным правилам и возвращает строку с правильно расставленными для конкретной базы данных управляющими символами.

Например:

```
#!/usr/bin/perl -w
#
# ch04/util/quote1: Демонстрирует использование метода $dbh->quote()

use DBI;

### Строка для расстановки кавычек
my $string = "Don't view in monochrome (it looks 'fuzzy')!";

### Соединиться с базой данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1
} );

### Расставить управляющие символы ...
my $quotedString = $dbh->quote( $string );
### Использовать строку в кавычках как строковый литерал в команде SQL
my $sth = $dbh->prepare( "
    SELECT *
    FROM media
```

```
    WHERE description = $quotedString
  " );
$sth->execute();

exit;
```

Например, если расставляются кавычки в строке Perl, содержащей текст `Do it!`, через дескриптор базы данных Oracle будет возвращено значение `'Do it!'`. Однако метод `quote()` учитывает и такие случаи, как `Don't do it!`, который для большинства баз данных должен транслироваться в `'Don''t do it!'`. Простое добавление окружающих кавычек произвело бы `'Don't do it!'`, что не является допустимым литералом SQL.

Некоторые базы данных требуют более сложного метода `quote()`, а некоторые драйверы, хотя и не все, располагают методом `quote()`, который может справляться с многострочными последовательностями и даже двоичными данными.

В особом случае, если аргумент представляет собой `undef`, метод `quote()` возвращает строку `NULL` без кавычек. Это соответствует использованию в DBI `undef` для представления значений `NULL` и использованию значений `NULL` в SQL.

Трассировка выполнения DBI

DBI демонстрирует крайне полезную возможность генерации трассировочной информации времени выполнения относительно выполняемых им действий, которая может помочь сберечь массу времени при прослеживании непонятных проблем в ваших программах DBI.

На самом верхнем уровне вы вызываете метод `DBI->trace()`, который включает трассировку всех операций DBI с момента своего вызова. Допустимы несколько уровней трассировки:

0 Трассировка отключается.

1 Трассируется выполнение методов DBI с показом возвращаемых значений и ошибок.

2 То же, что и 1, а также вход в метод и параметры.

3 То же, что и 2, а также дополнительные данные внутренней трассировки драйвера.

4 Уровни 4 и выше могут включать больше подробностей, чем оказывается полезным.

Метод `trace()` можно использовать с двумя видами аргументов, указывая только уровень трассировки либо также и файл, к которому дописывается трассировочная информация. Следующий пример демонстрирует использование `DBI->trace()`:

```
#!/usr/bin/perl -w
#
# ch04/util/trace1: Демонстрирует использование трассировки DBI.

use DBI;

### Удалить прежние файлы трассировки
unlink 'dbitrace.log' if -e 'dbitrace.log';

### Соединиться с базой данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" );

### Установить уровень трассировки в 1 и подготовиться
DBI->trace( 1 );
doPrepare();

### Установить уровень трассировки в 2 и подготовиться
DBI->trace( 2, 'dbitrace.log' );
doPrepare();

### Установить выдачу трассировочных данных снова на STDERR на уровне 2
### и подготовиться
DBI->trace( 2, undef );
doPrepare();

exit;

### Подготовить команду (недопустимую, чтобы продемонстрировать
### трассировку)
sub doPrepare {
    print "Подготовка и выполнение команды\n";
    my $sth = $dbh->prepare( "
        SELECT * FROM megalith
    " );
    $sth->execute();
    return;
}

exit;
```

Эта программа генерирует довольно много трассировочной информации, из которой мы покажем лишь малую часть:

```
  -> prepare for DBD::Oracle::db (DBI::db=HASH(0xcd45c)~0xcd4a4 '
      SELECT * FROM megalith
  ') thr0
  <- prepare= DBI::st=HASH(0xcd648) at trace1 line 30.
  -> execute for DBD::Oracle::st (DBI::st=HASH(0xcd648)~0x16afec) thr0
  dbd_st_execute SELECT (out0, lob0)...
  !! ERROR: 942 'ORA-00942: table or view does not exist (DBD ERROR:
     OCIStmtExecute)'
```

```
    <- execute= undef at trace1 line 33.
DBD::Oracle::st execute failed: ORA-00942: table or view does not exist
(DBD ERROR: OCIStmtExecute) at trace1 line 33.
```

Эта трассировочная информация была создана при установленном уровне трассировки 2 и показывает операции, которые осуществил DBI при попытке подготовить и выполнить команду. Строки, начинающиеся с `->`, записываются при входе в метод, а строки, начинающиеся с `<-`, записываются при выходе из метода. Выдача трассировки DBI имеет отступ в четыре пробела, чтобы отличить ее от выдачи программы.

Вы можете видеть вызов метода `prepare()` и его параметры: дескриптор базы данных и команду SQL, которую нужно подготовить.[1] В следующей строке показан возврат дескриптора команды методом `prepare()`. Показаны также файл и номер строки, из которой был вызван `prepare()`. Затем мы видим, как вызывается `execute()`, строку трассировки самого драйвера и возврат из метода после регистрации ошибки. Наконец, мы видим предупреждение, выведенное DBI в соответствии с атрибутом `PrintError`, установленным по умолчанию.

На уровне 1 генерируется аналогичная трассировочная информация. Основное отличие состоит в том, что не показываются строки входа в метод (`->`).

Один из недостатков такого вида трассировки состоит в том, что если в программе используется много дескрипторов, объем трассировочной информации может быть очень большим. Опять же, если вы проследили возникшую у вас проблему до конкретной операции над базой данных, вам желательно протрассировать отдельно именно ее.

Метод `trace()` доступен на уровне дескриптора, что позволяет отдельно трассировать операции над дескрипторами баз данных или команд. Поэтому можно трассировать операции с дескриптором базы данных на уровне 1 и с дескриптором отдельной команды на уровне 2. Например:

```
### Соединение с базой данных...
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" );
### Трассировка дескриптора базы данных на уровне 1 на экране
$dbh->trace( 1 );

### Создать новую команду
my $sth = ...;
```

[1] Если Perl, который вы используете, был скомпилирован с включенной многопоточностью, то в строке для каждого метода показан также номер потока, например `thr0`. В DBI реализован мьютекс для каждого драйвера, поэтому одновременно в каждый драйвер DBD может войти только один поток. Каким образом это действует, можно увидеть, установив уровень трассировки равным 4 или выше.

```
### Трассировать команду на уровне 2 в файл 'trace.lis'
$sth->trace( 2, 'trace.lis' );
```

Обратите внимание, что если при вызове `trace()` указано имя файла, то в данный момент в этот файл перенаправляется выдача *всех* дескрипторов.

Если ваши программы ведут себя странным образом или систематически выдают ошибки, то для решения проблем следует рассмотреть возможность использования встроенных функций трассировки DBI. Это средство очень полезно, поскольку позволяет увидеть, какие именно данные передаются в базу данных, и убедиться, что они имеют правильный формат.

Наконец, управлять трассировкой можно с помощью переменной окружения `DBI_TRACE`, которая действует аналогично методу `DBI->trace()`. Имеется в виду, что она трассирует все дескрипторы, используемые в программе. Эту переменную окружения можно использовать тремя способами, которые сведены в следующую таблицу.

Значение DBI_TRACE	Воздействие на DBI
`1`	`DBI->trace(1);`
`dbitrace.log`	`DBI->trace(2, 'dbitrace.log');`
`4=dbitrace.log`	`DBI->trace(4, 'dbitrace.log');`

Если уровень трассировки не указан в переменной окружения `DBI_TRACE`, то по умолчанию он принимается равным `2`, как показано в приведенной таблице.

Ясное и аккуратное форматирование

В DBI есть пара вспомогательных функций, которые можно использовать для приведения строк к виду, облегчающему чтение. Эти две функции называются `neat()` и `neat_list()`, первая из них действует на одиночное скалярное значение, а вторая – на список скалярных величин.

Например, для использования `neat()` с целью приведения в порядок нескольких строк можно написать такой код:

```
#!/usr/bin/perl -w
#
# ch04/util/neat1: Поверка вспомогательной функции DBI::neat().
#

use DBI;
```

```
### Зададим несколько строк для форматирования
my $str1 = "Alligator's an extremely neat() and tidy person";
my $str2 = "Oh no\nhe's not!";

### Отформатировать первую строку, ограничив длину 40 символами
print "Строка: " . DBI::neat( $str1, 40 ) . "\n";

### Отформатировать вторую строку, по умолчанию длина 400
print "Строка: " . DBI::neat( $str2 ) . "\n";

### Отформатировать число
print "Число: " . DBI::neat( 42 * 9 ) . "\n";

### Отформатировать неопределенное значение
print "Неопределенное значение: " . DBI::neat( undef ) . "\n";

exit;
```

Программа выводит следующее:

```
Строка: 'Alligator's an extremely neat() and...'
Строка: 'Oh no
he's not!'
Число: 378
Неопределенное значение: undef
```

Она показывает, что строковые значения заключаются в кавычки,[1] а числовые значения – нет. Первая строка усечена до указанной длины с добавлением «`...`». Неопределенные значения распознаются и возвращаются в виде строки `undef` без кавычек.

Функцию `neat()` удобно использовать для одиночных значений, а функцию `neat_list()` – для списков. Она просто вызывает `neat()` для каждого элемента заданного списка, а затем соединяет значения в список с использованием заданной строки-разделителя. Например:

```
#!/usr/bin/perl -w
#
# ch04/util/neat2: Демонстрация работы вспомогательной
#                  функции DBI::neat_list()
use DBI qw( neat_list );

### Объявим несколько строк для форматирования
my @list = ( 'String-a-string-a-string-a-string', 42, 0, '', undef );
```

[1] Обратите внимание на то, что внутренние кавычки не преобразуются. Это связано с тем, что `neat()` предназначена для создания результатов, которые можно читать, и делать это она должна быстро, поскольку используется внутренними механизмами трассировки. Если вы хотите преобразовать кавычки, то нужно воспользоваться методом `quote()`.

```
### Отформатируем строки в массив
print neat_list( \@list, 40, ", " ), "\n";

exit;
```

Эта программа выводит следующее:

```
'String-a-string-a-string-a-string-a...', 42, 0, '', undef
```

В этом примере показано также, что вспомогательные функции можно импортировать в пакет, и потому опустить префикс `DBI::`.

DBI использует `neat()` и `neat_list()` внутри себя для форматирования результатов, производимых при трассировке. Это важно знать, чтобы не удивляться, почему при трассировке ваши гигантские команды SQL сокращаются до 400 символов.[1]

Проверка чисел

Последняя вспомогательная функция DBI, которую мы рассмотрим, довольно любопытна и называется `looks_like_number()`. Эта функция просто сообщает, похожа величина на число или нет.

`looks_like_number()` принимает в качестве аргумента список величин и возвращает новый массив, в котором для соответствующих величин исходного массива указана одна из трех возможностей: они являются числом, они не являются числом или это не определено.

Может показаться, что в такой функции едва ли возникнет потребность, но при обработке больших объемов данных полезно определить, для каких значений может потребоваться преобразовать кавычки с помощью метода `quote()`.

Возвращаемый массив содержит то же число значений, что и исходный, а элементы его принимают одно из трех значений, имеющих следующий смысл:

```
true        Исходная величина является числом.
false       Исходная величина не является числом.
undef       Исходная величина пустая или неопределенная.
```

В следующем примере проиллюстрирована работа этой функции:

```
#!/usr/bin/perl -w
#
# ch04/util/lookslike1: Демонстрация работы функции
#                       DBI::looks_like_number().
```

[1] 400 является значением по умолчанию переменной `$DBI::neat_maxlen`, которая задает максимальную длину по умолчанию для функции `neat()`.

```
use DBI;

### Объявим список величин
my @values = ( 333, 'Choronzon', 'Tim', undef, 'Alligator', 1234.34,
               'Linda', 0x0F, '0x0F', 'Larry Wall' );

### Проверим, какие из них являются числами!
my @areNumbers = DBI::looks_like_number( @values );

for (my $i = 0; $i < @values; ++$i ) {

    my $value = (defined $values[$i]) ? $values[$i] : "undef";

    print "values[$i] -> $value ";

    if ( defined $areNumbers[$i] ) {
        if ( $areNumbers[$i] ) {
            print "является числом!\n";
        }
        else {
            print "совершенно не похожа на число и должна быть
                   заключена в кавычки!\n";
        }
    }
    else {
        print "не определена!\n";
    }
}

exit;
```

Результат работы этой программы показывает, как обрабатываются величины. Видно, что шестнадцатеричные значения не воспринимаются как числа:

```
values[0] -> 333 является числом!
values[1] -> Choronzon совершенно не похожа на число и должна быть
                   заключена в кавычки!
values[2] -> Tim совершенно не похожа на число и должна быть
                   заключена в кавычки!
values[3] -> undef не определена!
values[4] -> Alligator совершенно не похожа на число и должна быть
                   заключена в кавычки!
values[5] -> 1234.34 является числом!
values[6] -> Linda совершенно не похожа на число и должна быть
                   заключена в кавычки!
values[7] -> 15 является числом!
values[8] -> 0x0F совершенно не похожа на число и должна быть
                   заключена в кавычки!
values[9] -> Larry Wall совершенно не похожа на число и должна быть
                   заключена в кавычки!
```

Первая величина `0x0F` в списке воспринята как число, поскольку Perl преобразовал ее в `15` при компиляции сценария. Вторая такая же величина не воспринята как число, т. к. функция `looks_like_number()` воспринимает только целые числа и числа с плавающей запятой.

На этом мы заканчиваем введение в DBI и его архитектуру. В следующей главе мы подробно поговорим о том, как делать практические вещи с помощью DBI.

5

Взаимодействие с базами данных

Изучая DBI, мы к настоящему времени разобрались в том, как устанавливать и завершать соединение с базами данных различных типов в программах на Perl, а также в способах обнаружения и исправления ошибок при вызове методов DBI.

До сих пор мы не обсуждали, как обрабатывать данные в базах, т. е. извлекать, обновлять и удалять информацию (помимо прочего). В этой главе рассказывается, как это можно делать с помощью DBI и как использовать мощные функции обработки данных Perl для эффективной работы с данными.

Вспомните обсуждение архитектуры DBI в главе 4, особенно тему *дескрипторов команд*. Эти дескрипторы и связанные с ними методы обеспечивают функции обработки данных в базах.

Выполнение простых запросов

Чаще всего взаимодействие программы и базы данных заключается в извлечении данных. В стандартном SQL этот процесс осуществляется с помощью ключевого слова `SELECT`. Используя Perl и DBI, мы получаем значительно больший контроль над способами извлечения данных из базы. Мы также получаем значительно больший контроль над тем, как будут обрабатываться данные после извлечения.

Извлечение данных с использованием DBI представляет собой цикл, состоящий из четырех этапов:

1. На *подготовительном* этапе производится синтаксический разбор команды SQL, осуществляется проверка ее допустимости и возвращается дескриптор, представляющий эту команду в базе данных.

2. Если на подготовительном этапе был возвращен действительный дескриптор команды, то на следующем этапе эта команда *выполняется* в базе данных. Происходит фактическое выполнение запроса и заполнение структур запрошенными данными внутри базы данных. Однако на этом этапе у программы Perl нет доступа к запрошенным данным.

3. Третий этап известен как этап *выборки*, во время которого из базы данных выбираются фактические данные с использованием дескриптора команды. На этапе выборки запрошенные данные строка за строкой вытаскиваются в структуры данных Perl, такие как скаляры и хэши, и потом могут обрабатываться вашей программой.

   Этап выборки заканчивается, когда оказываются выбранными все данные, или он может быть досрочно прерван с помощью метода `finish()`.

   Если позднее возникнет необходимость в повторном выполнении (re-`execute()`) запроса, например, с другими параметрами, то можно сохранить дескриптор команды, выполнить снова `execute()` и в результате вернуться на этап 2.

4. Последним этапом цикла извлечения данных является этап *освобождения ресурсов*. В сущности, это внутренняя операция чистки, в которой DBI и драйвер освобождают ресурсы, занятые дескриптором команды и связанными с ним данными. Для некоторых драйверов этот процесс может включать в себя необходимость обратиться к базе данных и потребовать у нее освобождения всех ресурсов, которые могут быть связаны с командой.

   Все это производится автоматически в результате действия собственного механизма *сборки мусора* в Perl.

Этот цикл происходит с каждой выполняемой командой SQL `SELECT`. В других командах SQL, например `INSERT`, `UPDATE` и `DELETE`, этап выборки пропускается и выполняются только этапы подготовки, выполнения и освобождения ресурсов (о чем мы расскажем далее в этой главе).

Чтобы понять, как этот четырехэтапный цикл выборки вписывается в ваши программы, рассмотрим более подробно каждый этап в отдельности.

Подготовка команд SQL

На первом этапе цикла извлечения данных из базы осуществляется *подготовка* дескриптора команды SQL. Этот этап, в основном, соответ-

ствует этапу *синтаксического разбора*, происходящему внутри ядра базы данных.

При этом обычно команда SQL посылается базе данных в виде строки символов через действующий дескриптор базы данных. Затем база данных производит синтаксический анализ строки, чтобы убедиться в допустимости команды SQL как в отношении синтаксиса, так и в отношении объектов базы данных, на которые ссылается команда (т. е. проверяется существование таблиц и наличие прав доступа к ним, если они существуют).

Если база данных сможет «заглотить» команду, будет возвращена некоторая структура данных, специфичная для конкретной базы, в которой инкапсулируется разобранная команда. Эту специфическую для базы данных структуру DBI инкапсулирует дальше в виде *дескриптора команды*. Этот процесс проиллюстрирован на рис. 5.1.

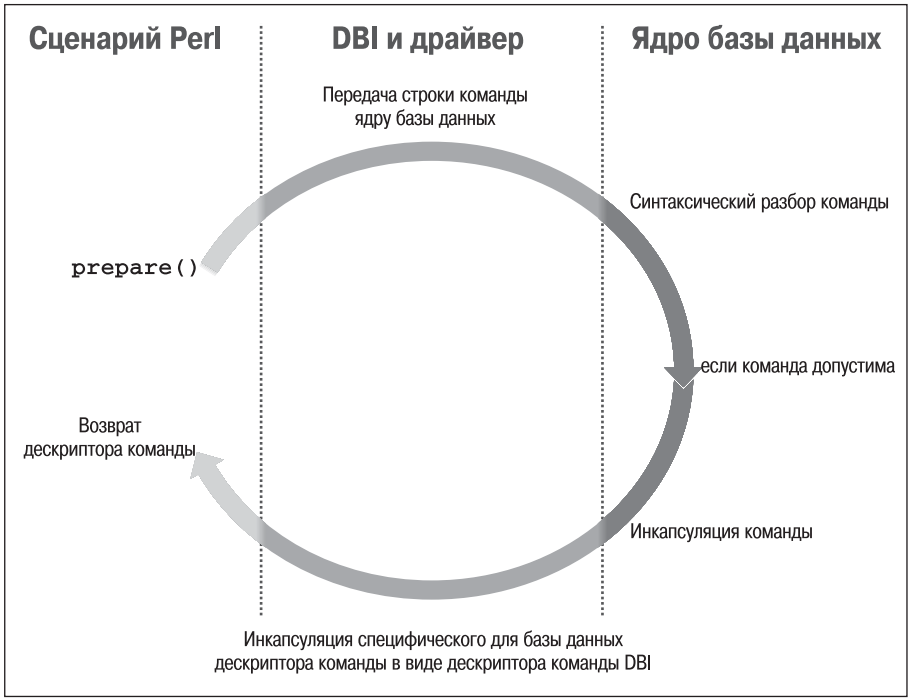

Рис. 5.1. *Команда подготовки потока данных посредством DBI*

С помощью этого дескриптора команды DBI и осуществляется оставшаяся часть цикла выборки данных.

В терминах DBI подготовка команды осуществляется с помощью метода `prepare()`, который выполняется через дескриптор базы данных. Например, простую программу DBI, создающую дескриптор команды, можно написать так:

```
#!/usr/bin/perl -w
#
# ch05/prepare/ex1: Создание дескриптора базы данных
#                   и дескриптора команды

use DBI;

### Дескриптор базы данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" );

### Дескриптор команды
my $sth = $dbh->prepare( "SELECT id, name FROM megaliths" );

exit;
```

При этом предполагается, конечно, что разбор команды произведен успешно. Может оказаться, что вы сделаете опечатку при вводе команды SQL, либо база данных не сможет провести синтаксический разбор команды по каким-то другим причинам. В этом случае вызов `prepare()` возвращает значение `undef`, что указывает на неудачу анализа.

Помимо возврата неопределенного значения DBI напечатает сообщение об ошибке, поскольку в дескрипторах баз данных, возвращаемых `DBI->connect()`, по умолчанию устанавливается атрибут `PrintError`. Подробнее о `PrintError` см. в главе 4.

Отметим важную особенность процесса подготовки команд. Некоторые драйверы могут откладывать фактическое выполнение этапа подготовки до вызова `execute()`, поскольку в некоторых базах данных другого способа не предоставляется. Поэтому все сказанное о методе `prepare()` – что он делает и почему его выполнение может оказаться неудачным – может относиться лишь к тому моменту, когда осуществляется вызов `execute()`.

Создание команд «на лету»

Можно создавать команды SQL «на лету», используя возможности обработки строк в Perl, а затем передавать их методу `prepare()`. Хорошим примером демонстрации такой работы может служить использование DBI для интеграции баз данных с веб-сайтами.

Предположим, что вы сделали базу данных по мегалитическим площадкам доступной для просмотра в Сети. Когда пользователь вводит название площадки, оно передается в виде строки сценарию CGI. Затем эта строка используется в команде SQL для извлечения данных по площадке из базы.

Для реализации такого рода взаимодействия нужно иметь возможность создавать команды SQL, и одним из способов осуществления этого является использование функций обработки строк в Perl.[1]

[1] Часто лучшим способом будет использование *связанных значений*, о чем мы расскажем далее в этой главе.

Следующая программа иллюстрирует этот принцип:

```
### Этой переменной каким-то образом присваивается значение из формы...
my $siteNameToQuery = $CGI->param( "SITE_NAME" );

### Нужно обеспечить правильную расстановку кавычек для использования
### строки в команде SQL
my $siteNameToQuery_quoted = $dbh->quote( $siteNameToQuery );

### Теперь вставим переменную в команду SQL, заключенную
### в двойные кавычки
$sth = $dbh->prepare( "
            SELECT meg.name, st.site_type, meg.location, meg.mapref
            FROM megaliths meg, site_types st
            WHERE name = $siteNameToQuery_quoted
            AND meg.site_type_id = st.id
          " );
$sth->execute();
@row = $sth->fetchrow_array();
...
```

Более того, *любую* часть этого запроса можно создать «на лету», поскольку на этом этапе команда SQL является просто строкой Perl. Другим хитрым приемом является настройка колонок, участвующих в запросе к базе данных, в зависимости от того, какие поля помечены в броузере для вывода. На рис. 5.2 показана веб-страница, на которой пользователь отмечает нужные ему колонки.

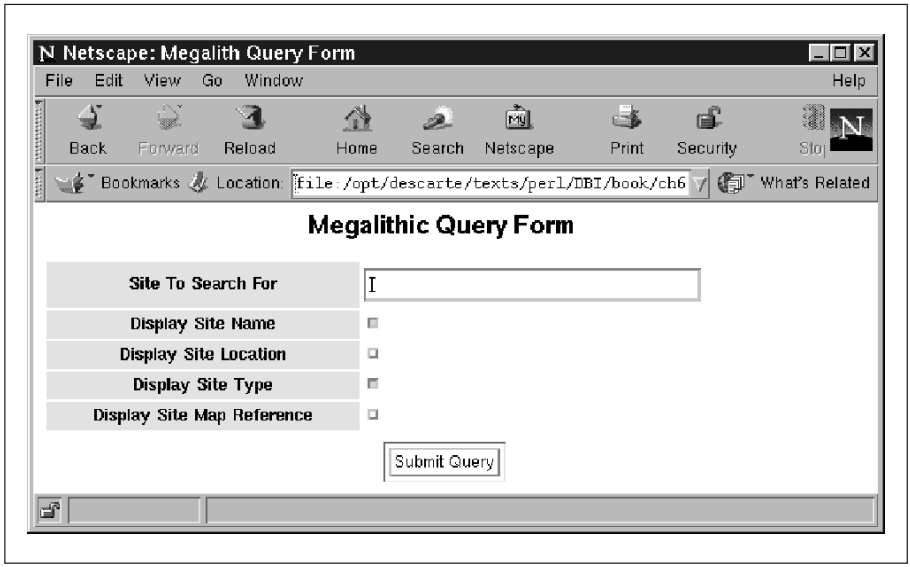

Рис. 5.2. *Страница поиска мегалитических площадок*

Программу, необходимую для управления генерацией таких команд SQL, можно написать следующим образом:

```
### Создадим список имен выбранных полей
@fields = ();

### Определим, какие пункты были выбраны
push @fields, "name"     if $nameCheckbox     eq "CHECKED";
push @fields, "location" if $locationCheckbox eq "CHECKED";
push @fields, "type"     if $typeCheckbox     eq "CHECKED";
push @fields, "mapref"   if $maprefCheckbox   eq "CHECKED";

### Проверим, что хоть что-то было выбрано
die "Не выбраны поля для запроса!\n"
    unless @fields;

### Теперь составим команду SQL
$statement = sprintf "SELECT %s FROM megaliths WHERE name = %s",
    join(", ", @fields), $dbh->quote($siteNameToQuery);

### Выполним запрос
$sth = $dbh->prepare( $statement );
$sth->execute();
@row = $sth->fetchrow_array();
...
```

Таким образом, весь запрос SQL, начиная от выбираемых колонок и кончая условиями, которые должны выполняться для выбираемых данных, создан динамически и передан на обработку в базу данных.

Страница, выводимая на броузер пользователя после выполнения этого запроса, показана на рис. 5.3.

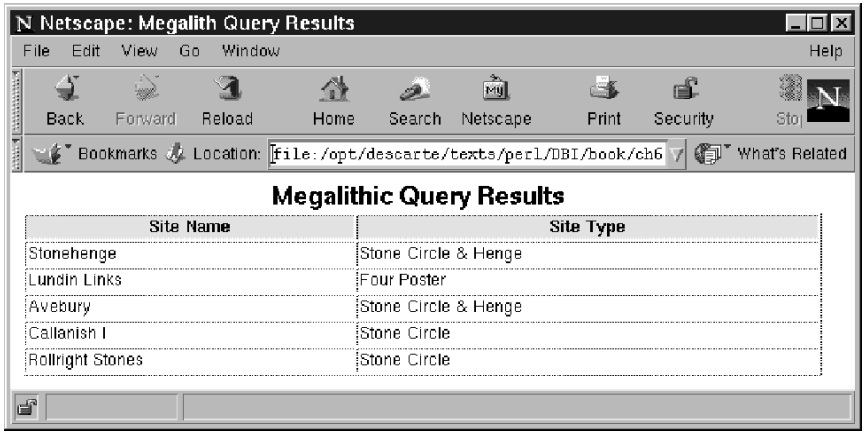

Рис. 5.3. *Страница результатов поиска по базе мегалитических площадок*

Используя обработку строк в Perl для создания команд SQL на основе данных, вводимых пользователем, и DBI, можно управлять достаточно сложными веб-формами очень простым и гибким способом.

Выполнение команд Select

На втором этапе цикла выборки данных нужно сообщить базе данных о необходимости *выполнить* подготовленную команду SQL. На этом этапе выполнения от базы данных требуется осуществить запрос и начать собирать результирующий набор данных.

Выполнение команды SQL происходит посредством действующего дескриптора команды, создаваемого при успешном завершении метода `prepare()`. Например, выполнение команды SQL можно записать следующим простым способом:

```
### Создать дескриптор команды
my $sth = $dbh->prepare( "SELECT id, name FROM megaliths" );

### Выполнить дескриптор команды
$sth->execute();
```

Если выполнение команды пройдет успешно, вызов метода `execute()` возвратит значение `true`, в противном случае возвращается значение `undef`.

Как и в большинстве методов DBI, если установлен атрибут `PrintError`, то через метод `warn()` будет выведено сообщение об ошибке. С другой стороны, если установлен атрибут `RaiseError`, будет сгенерирована исключительная ситуация посредством метода `die()`. Какой бы способ вы не использовали, всегда неплохо проверить наличие ошибок.[1]

Если происходит успешный возврат из `execute()`, это не обязательно означает, что база данных завершила выполнение команды `SELECT`; оно могло только начаться. Представьте себе, что мегалиты встречаются очень часто, и в нашей таблице `megaliths` десять миллионов строк. В ответ на `execute()` база данных могла бы лишь установить указатель, известный также как *курсор*, прямо над первой строкой таблицы.

Это значит, что после успешного выполнения метода база данных и драйвер готовы возвращать результаты, но ваша программа на Perl их пока не получила. Об этом важно помнить. Чтобы извлечь данные из базы, нужно явным образом осуществить их *выборку*. В этом состоит третий этап цикла.

[1] Мы не всегда явно проверяем возникновение ошибок в приводимых примерах программ. В таких случаях можете считать, что мы пристегнулись к нашему катапультируемому креслу `RaiseError`.

Выборка данных

Выборка данных является главной целью подачи запросов базе данных. Прекрасно, что можно потренировать базу данных на выполнении запросов, но пока вы не извлечете данные фактически, ваша программа не сможет ими воспользоваться.

Данные, возвращаемые запросом SQL, известны как *результирующий набор*, или *результирующее множество* (что связано с математической *теорией множеств*, на которой основана теория реляционных баз данных). Результирующий набор возвращается в программу на Perl в результате перебора всех записей (строк) в наборе и переноса в программу значений из каждой строки. Такой вид выборки данных результирующего набора на построчной основе обычно называется *курсором*.

Курсоры полезны при последовательной выборке: записи выбираются в том порядке, в котором они хранятся в результирующем наборе. При этом нельзя пропустить запись или перейти к произвольно выбранной. Более того, после того как адресованная курсором строка выбрана, курсор о ней «забывает». Таким образом, курсоры не могут проходить по результирующему набору в обратном направлении.

Поэтому обычным способом выборки данных из результирующего набора базы данных является циклический проход по записям, возвращаемым с помощью дескриптора команды, и обработка каждой строки, пока строк для выборки больше не останется. Это можно записать в виде следующего псевдокода:

```
while ( записи, выбираемые из $sth ) {
    ### Выбрать из курсора текущую строку
    @columns = получить значения колонок;
    ### Вывести значения...
    print "Выбранная строка: @columns\n";
}
```

DBI еще больше упрощает этот процесс путем объединения в одном методе проверки наличия данных и выборки данных.

Существует несколько способов извлечения строк из результирующего набора с использованием различных типов данных Perl. Например, можно выбрать строку как простой список значений, ссылку на массив значений или ссылку на хэш из пар «имя поля/значение». Все эти способы извлекают из курсора текущую строку, но возвращают данные в программу Perl в различных форматах.

Проще всего выборка данных осуществляется с помощью метода `fetchrow_array()`, возвращающего массив, или, скорее, список, содержащий поля строки. Допустим, что нужно извлечь из нашей базы данных по мегалитам название площадки и ее тип. Для выборки этих данных из таблицы служит следующий код:

```
### Подготовить команду SQL ( предполагается существование $dbh)
$sth = $dbh->prepare( "
            SELECT meg.name, st.site_type
            FROM megaliths meg, site_types st
            WHERE meg.site_type_id = st.id
          " );

### Выполнить команду SQL и создать результирующий набор
$sth->execute();

### Выбрать из базы каждую строку результирующих данных как список
while ( ( $name, $type ) = $sth->fetchrow_array ) {
    ### Вывести маленькое сообщение....
    print "Мегалитическая площадка $name имеет тип $type\n";
}
```

Вместо списка скалярных переменных можно выбрать результирующие данные в массив переменных посредством метода `fetchrow_array()`:

```
while ( @row = $sth->fetchrow_array ) {
    ### Вывести маленькое сообщение
    print " Мегалитическая площадка $row[0] имеет тип $row[1]\n";
}
```

что функционально одинаково.

Важно помнить, что поля в результирующем наборе располагаются в том порядке, в каком были перечислены колонки в команде SQL. Поэтому в приведенном примере поле `name` было запрошено перед полем `site_type`. В результате первым элементом массива или скалярного списка является значение поля `name`, за которым следует значение поля `site_type`.

Цикл `while` выполняется, пока значением выражения в скобках не станет `false`. Естественно, мы хотим прекратить выполнение цикла, когда больше не останется данных для выборки, и метод `fetchrow_array()` обеспечивает нам это. Когда данных больше нет, он возвращает пустой список. Perl воспринимает это как значение `false`, тем самым прекращая выполнение цикла.

При работе с циклами выборки важно помнить, что методы выборки возвращают одно и то же значение и тогда, когда данных больше нет, и тогда, когда возникает ошибка. Поэтому ошибка во время выборки вызывает выход из цикла, как если бы все данные были выбраны. Если не установлен атрибут `RaiseError`, то полезно проверить наличие ошибки сразу после выхода из цикла. Это показано в примере, который приведен ниже.[1]

[1] В остальных примерах циклов выборки в данной книге предполагается, что установлен атрибут `RaiseError`.

Другим способом, с помощью которого можно осуществлять выборку данных из базы, является использование метода `fetchrow_arrayref()`, возвращающего не сам массив, а ссылку на него. Этот метод имеет более высокую производительность по сравнению с `fetchrow_array()`, поскольку возвращаемые данные не копируются в новый массив для каждой выбранной строки. Например:

```
### Выбрать из базы данных строки результирующего набора
### в виде ссылки на массив....
while ( $array_ref = $sth->fetchrow_arrayref ) {
    ### Вывести маленькое сообщение....
    print "Мегалитическая площадка $arrayref->[0] имеет тип
           $array_ref->[1]\n";
}
die "Ошибка выборки данных $DBI::errstr" if $DBI::err;
```

Важно учитывать, что теперь для всех строк, выбираемых из базы данных через текущий дескриптор команды, используется одна и та же ссылка на массив. Если вы собираетесь хранить данные строк для будущего использования, об этом очень важно помнить. Например, в следующей программе возвращаемые данные накапливаются в постоянном хранилище, чтобы ими можно было воспользоваться по завершении выборки:

```
### «Заначка» для строк...
my @stash;

### Выбрать ссылки на строки и «заначить» их!
while ( $array_ref = $sth->fetchrow_arrayref ) {
    push @stash, $array_ref;        # XXX НЕВЕРНО!
}

### Вывести содержимое «заначки»!
foreach $array_ref ( @stash ) {
    print "Строка: @$array_ref\n";
}
```

Здесь происходит нечто странное. Все строки, выведенные из «заначки», оказываются одинаковыми. Это происходит потому, что сохранялись не данные строки, а ссылка на строку, а поскольку DBI использует для каждой строки одну и ту же ссылку, получается совсем не тот результат, который ожидался. Нужно сохранить копии значений массива, на который указывает ссылка, а не саму ссылку, как показано в примере:

```
### «Заначка» для строк...
my @stash;

### Выбрать ссылки на строки и «заначить» их!
while ( $array_ref = $sth->fetchrow_arrayref ) {
```

```
    push @stash, [ @$array_ref ];  # Копирование содержимого массива
}

### Вывести содержимое «заначки»!
foreach $array_ref ( @stash ) {
    print "Строка: @$array_ref\n";
}
```

Метод `fetchrow_arrayref()` особенно полезно использовать в сочетании с *привязкой колонок*, о которой мы расскажем далее в этой главе.

И последним методом выборки строк результирующего набора из базы данных на основе использования курсора является захват его в ссылке на хэш. Это осуществляется с помощью метода `fetchrow_hashref()`, используемого аналогично `fetchrow_arrayref()`. Например:

```
### Осуществить выборку текущей строки в ссылку на хэш
while ( $hash_ref = $sth->fetchrow_hashref ) {
    ...
```

Хэш, на который указывает ссылка, использует в качестве ключей имена выбираемых полей, а значения этих полей хранятся как значения хэша. Поэтому при выборке из базы данных полей `name` и `site_type` можно обращаться к элементам хэша следующим образом:

```
### Выборка строк в ссылку на хэш
while ( $hash_ref = $sth->fetchrow_hashref ) {
    print "Мегалитическая площадка $hash_ref->{name} имеет тип
           $hash_ref->{site_type}\n";
}
```

Как можно ожидать, при использовании этого метода вас подстерегает ряд опасностей. Самое важное – проявлять осторожность в отношении фактического имени поля, которое вы выбираете. Некоторые базы данных производят с именами полей неожиданные вещи, например, переводят все символы в верхний или в нижний регистр, что приводит к неправильному определению ключа хэша. Решить эту проблему можно, передав `fetchrow_hashref()` имя *атрибута*, который нужно использовать при передаче имен полей. Например, можно использовать `NAME` по умолчанию, `NAME_uc` использовать для перевода имен полей в верхний регистр и `NAME_lc` для перевода их в нижний регистр. Например, переносимый способ использования ссылок на хэш может быть таким:

```
### Выборка строк в ссылку на хэш с именами полей в верхнем регистре
while ( $hash_ref = $sth->fetchrow_hashref('NAME_lc') {
    print "Мегалитическая площадка $hash_ref->{name} имеет тип
           $hash_ref->{site_type}\n";
}
```

Рекомендуется указывать NAME_uc или NAME_lc; на производительность работы это не оказывает никакого влияния.

Мы отметим еще несколько опасностей при использовании fetchrow_hashref(). Если в команде SELECT используется полностью квалифицированное имя поля, например:

```
SELECT megaliths.id, ...
```

то большинство баз данных возвращает в качестве имени поля только строку id. Обычно это не создает проблем, но вы можете споткнуться, если одно и то же имя имеют несколько полей:

```
SELECT megaliths.id, media.id ...
```

Поскольку хэш, возвращаемый fetchrow_hashref(), может иметь только один ключ id, вы не сможете получить значения обоих полей и даже не можете быть уверенными в том, значение которого из двух полей id вы получили. Вам придется либо использовать другой метод выборки строк, либо применить *псевдонимы* для названий колонок. Присваивание псевдонимов колонкам аналогично присваиванию псевдонимов таблицам, о котором мы говорили в главе 3. Псевдоним можно указать после выражения, задающего колонку:

```
SELECT megaliths.id meg_id, media.id med_id ...
```

хотя некоторые базы данных требуют несколько более многословную форму:

```
SELECT megaliths.id AS meg_id, media.id AS med_id ...
```

Использование псевдонимов очень удобно, когда выбираются выражения, например:

```
SELECT megaliths.id + 1 ...
```

поскольку базы данных по-разному именуют колонки, содержащие выражения. Использование псевдонимов не только облегчает ссылки на колонки, но и делает приложение более переносимым.

При обсуждении метода fetchrow_arrayref() мы отметили, что в текущей версии для каждой строки он возвращает одну и ту же ссылку на массив. Что касается fetchrow_hashref(), то он не возвращает одну и ту же ссылку на хэш для каждой строки, но наверняка будет делать это в следующей версии. (В результате он станет также быстрее работать, ибо сейчас работает несколько медленнее, чем хотелось бы.)

Существуют другие способы выборки данных из базы, которые связаны либо с *пакетной выборкой* (*batch fetching*), либо с *атомарной выборкой* (*atomic fetching*), и обсуждаются ниже в этой главе.

Быстрый способ выборки и вывода

В DBI имеется вспомогательный метод dump_results(), выбирающий все строки результирующего набора дескриптора команды и выводящий их. Этот метод вызывается через подготовленный и выполненный дескриптор команды и действует, выбирая и выводя все строки результирующего набора. Каждая выбираемая строка форматируется согласно правилам, установленным по умолчанию или заданным в программе. По окончании работы dump_results() выводится число строк, выбранных из базы данных, и сообщение об ошибке, если она возникла. Метод возвращает число выбранных строк.

Например, чтобы быстро вывести результаты обработки запроса, можно написать такой код:

```
$sth = $dbh->prepare( ¨
            SELECT name, mapref, location
            FROM megaliths
          ¨ );
$sth->execute();
$rows = $sth->dump_results();
```

который выведет следующие данные:

```
'Balbirnie', 'NO 285 029', 'Balbirnie Park, Markinch, Fife'
'Castlerigg', 'NY 291 236', 'Near Keswick, Cumbria, England'
'Sunhoney', 'NJ 716 058', 'Near Insch, Aberdeenshire'
'Avebury', 'SU 103 700', 'Avebury, Wiltshire, England'
4 rows
```

Можно управлять способом форматирования выходных данных, задавая максимальную длину каждого поля в строке, символы, разделяющие поля в строке, и символы, разделяющие строки. Можно также задать дескриптор файла Perl, в который должна быть записана выдача.

Значениями по умолчанию для этих параметров являются:

```
1:  Максимальная длина поля  -   35
2:  Разделитель строк        -   ¨\n¨
3:  Разделитель полей        -   ¨,¨
4:  Дескриптор файла выдачи  -   STDOUT
```

Поэтому для вывода в файл полей размером в 80 символов, разделяемых двоеточиями, можно написать такой код:

```
### Подготовить и выполнить запрос
$sth = $dbh->prepare( ¨
            SELECT name, location, mapref
            FROM megaliths
          ¨ );
$sth->execute();
```

```
### Открыть выходной файл
open FILE, ">results.lis" or die "Невозможно открыть results.lis: $!";

### Сделать дамп отформатированных результатов в файл
$rows = $sth->dump_results( 80, '\n', ':', \*FILE );

### Закрыть выходной файл
close FILE or die "Ошибка закрытия файла с результатами: $!\n";
```

Метод `dump_results()` использует внутри себя для форматирования вспомогательную функцию `neat_list()`, описанную в предыдущей главе. Поэтому не следует использовать выходные данные `dump_results()` для передачи или обработки данных. Она предназначена только для просмотра выдачи пользователем.

Прерывание выборки данных

Когда дескриптор команды для `SELECT` успешно выполнен, он называется *активным*. Есть даже булев атрибут дескриптора команды с именем `Active`, доступный для чтения. Активность означает, что сервер базы данных производит какие-то действия в интересах этого дескриптора.

При повторном вызове метода выборки после выборки последней строки данных драйвер автоматически прекращает активность сервера базы данных от имени этого `execute()` и сбрасывает атрибут `Active`. На практике большинству драйверов в этом случае не приходится ничего делать, поскольку серверу известно, что драйвер извлек последнюю строку, поэтому сервер автоматически освобождает все ресурсы, использовавшиеся для хранения результирующего набора.

Поскольку эта завершающая работа выполняется автоматически при возврате методом выборки состояния «конец данных», обычно нет необходимости думать об этом. Однако есть два случая, когда уместно взять ситуацию под свой контроль и вызвать метод `finish()` дескриптора команды. (Имейте в виду, что `finish()` не означает конца самого дескриптора команды, завершается только текущее ее *выполнение*. Позднее вы можете снова вызвать `execute()` этого дескриптора.)

Первая ситуация несколько неопределенна и связана с добропорядочным поведением при работе с базой данных. Если для хранения вашего результирующего набора сервер базы данных временно использует значительный объем дискового пространства, *и* вы извлекли не все записи, *и* вы не будете в скором времени уничтожать или повторно выполнять дескриптор команды, то тогда уместно вызвать `finish()`. Благодаря этому сервер сможет освободить дисковое пространство, временно выделенное для хранения ваших результатов.[1]

Второй тип ситуации является менее неопределенным, поскольку DBI выдает ворчливое предупреждение типа одного из `disconnect()`:

```
disconnect invalidates 1 active statement handle
(either destroy statement handles or call finish on them before discon-
necting)
(отсоединение аннулирует 1 активный дескриптор команды;
прежде чем отсоединяться, уничтожьте дескрипторы команд или вызовите для
них finish)
```

В данном случае DBI предупреждает, что имеется активный дескриптор команды, возможно, содержащий данные для выборки, который аннулируется, т. е. становится непригодным для использования в результате отсоединения от базы данных.

С какой целью DBI старается вас предупредить? Идея в том, чтобы помочь обнаружить места неперехваченных и необработанных ошибок выборки, в которых цикл выборки был прерван раньше, чем были извлечены все данные. Некоторые ошибки выборки строк, такие как прерывание выполнения транзакции, означают, что из этого дескриптора команды невозможно получить дополнительные строки. В таких случаях драйвер сбрасывает флаг `Active`. В других случаях, например, при делении на ноль в выражении для колонки или усечении длинного значения поля, можно продолжить извлечение строк, поэтому драйвер сохраняет флаг `Active` в установленном состоянии.

На практике существуют ситуации и помимо циклов выборки, в которых дескриптор команды может остаться активным как при нормальном ходе событий, так и в исключительной ситуации.

Самой простой является ситуация, когда вы хотите выбрать только n строк, поскольку знаете, что их всего n. Большинство драйверов не могут определить, что извлечена последняя строка, и поэтому не могут сбросить флаг `Active`. Это аналогично ситуации «добронамеренного отношения» к базе данных, обсуждавшейся выше. В следующем примере показано использование метода `finish()` после выборки единственной строки, представляющей интерес:

```
sub select_one_row {
    my $sth = shift;
    $sth->execute(@_) or return;
    my @row = $sth->fetchrow_array();
    $sth->finish();
    return @row;
}
```

Еще одна исключительная ситуация часто связана с использованием `RaiseError`. При возбуждении исключительной ситуации, например,

[1] Классическим примером является команда `SELECT dialled_number, count(*) FROM calls WHERE subscriber = ? GROUP BY dialled_number ORDER BY count(*) DESC`, когда нужно выбрать лишь несколько первых строк из тех тысяч, которые база данных записала во временный буфер и отсортировала.

когда DBI обнаруживает ошибку в дескрипторе с установленным атрибутом RaiseError или когда другая часть программы вызывает die(), происходит внезапная передача управления из текущей точки в ближайший окаймляющий блок eval. Вполне возможно, что в результате останутся дескрипторы с незаконченной обработкой данных.

Вывод предупреждения, выдаваемого disconnect(), и большинства других предупреждений DBI можно подавить для каждого дескриптора, сбросив его атрибут Warn. Такая практика обычно не одобряется, но если в этом есть необходимость, можно ее использовать.

Запомните, что вызов finish() не является необходимым, не разрушает сам объект дескриптора команды в Perl, не обязателен для устранения утечек памяти и не мешает повторному вызову execute() для дескриптора. Все это распространенные заблуждения, переходящие из одной книги в другую. Мы обсудим то, как на самом деле разрушаются дескрипторы, в следующем разделе.

Освобождение ресурсов дескрипторов

При подготовке команды возвращаемый дескриптор команды связан с выделенными для него ресурсами памяти как в сценарии Perl, так и на сервере базы данных, с которым установлено соединение. Когда дескриптор команды вам больше не нужен, его следует уничтожить. Звучит страшно, но в действительности его нужно просто освободить.

Дескрипторы команд на практике представляются объектами Perl и подпадают в этом качестве под действие сборщика мусора Perl. Это означает, что когда не остается больше ссылок на дескриптор команды (например, переменная дескриптора вышла из области видимости или получила новое значение), Perl сам разрушит объект и вернет себе использовавшиеся им ресурсы.

Вот пример недолговечного дескриптора команды:

```
if ($fetch_new_data) {
    my $sth = $dbh->prepare( ... );
    $sth->execute();
    $data = $sth->fetchall_arrayref();
}
```

Обратите внимание, что нет необходимости явным образом освобождать или удалять дескриптор команды. Это делает за нас Perl. Переменная my $sth содержит единственную ссылку на данный объект дескриптора команды. Когда в конце блока переменная $sth прекращает свое существование, удаляется последняя ссылка и вступает в дело сборщик мусора Perl. Аналогично при выходе из сценария прекращают свое существование все глобальные переменные, а все объекты, на которые они ссылаются, высвобождаются точно таким же образом.

Вот несколько иной пример:

```
### Подача команд SQL для выбора площадок по типу
foreach ( 'Каменный круг', 'Монолит', 'Хендж' ) {
    my $sth = $dbh->prepare( ... $_ ... );
    $sth->execute();
    $sth->dump_results();
}
```

Во время второго и последующих проходов цикла переменной `$sth` присваивается ссылка на новый дескриптор команды, в результате чего удаляется прежде содержавшаяся в ней ссылка. И снова, поскольку это была единственная ссылка на дескриптор и она уничтожена, ресурсы дескриптора освобождаются.

Ваше приложение во время выполнения может готовить, использовать и выбрасывать тысячи (или сотни тысяч) дескрипторов команд. Если бы ресурсы базы данных, выделенные для этих команд, высвобождались лишь при закрытии соединения с базой данных, они могли бы в скором времени истощиться.

Практически единственный случай, когда можно создать перегрузку базы данных, – это хранение дескрипторов команд в массивах или хэшах. Если вы не позаботитесь об удалении или переписывании старых значений, то дескрипторы могут накапливаться.

Чтобы следить за тем, сколько дескрипторов команд размещено в памяти для данного дескриптора базы данных (например, для обнаружения утечек памяти), можно использовать атрибуты дескриптора базы данных `Kids` и `ActiveKids`. Оба они возвращают целые числа. Первый подсчитывает все дескрипторы команд, а второй – только те, у которых установлен атрибут `Active`.

Выполнение команд, отличных от SELECT

В главе 3 мы обсуждали различные методы обработки данных, которые могут вам потребоваться. До сих пор в этой главе мы касались наиболее часто используемой операции обработки данных – выборки. А как насчет вставки, удаления и обновления данных?

Эти операции осуществляются несколько отличным от запросов способом, поскольку в них не используется понятие курсора для прохода через результирующий набор. Они воздействуют на строки данных, хранимые в таблицах, не возвращая в программу никаких строк. Поэтому для таких операций не подходит полный цикл подготовки–выполнения–выборки–освобождения. Этап выборки здесь просто неприменим.

Поскольку обычно требуется вызывать такие команды только единожды, было бы весьма утомительно вызывать `prepare()` для получения дескриптора команды, и затем вызывать для этого дескриптора метод `execute()`, чтобы сразу после вызова выбросить этот дескриптор.

К счастью, в DBI определена сокращенная форма выполнения этих операций – метод `do()`, вызываемый для действующего дескриптора базы данных. Использовать `do()` крайне просто. Например, для удаления некоторых строк из таблицы `megaliths` требуется всего лишь следующий программный код:

```
### Предполагая, что существует действующий дескриптор базы данных ....
### Удалить строки для Стоунхенджа!
$rows = $dbh->do( "
            DELETE FROM megaliths
            WHERE name = 'Стоунхендж'
        " );
```

Чтобы оповестить об успешном выполнении команды SQL, вызов возвращает значение, показывающее число строк, обработанных командой SQL, или содержащее `undef`, если произошла ошибка.

Некоторые базы данных и некоторые команды не могут возвращать число строк, обработанных командой; в таких случаях возвращается величина `-1`.

Особым случаем является возврат нулевого числа строк в виде строки `0E0`, являющейся лишь причудливым способом изображения нуля. Возвращая `0E0` вместо `0`, метод `do()` возвращает значение, интерпретируемое Perl как `true` даже в том случае, когда не было обработано ни одной строки.[1] Метод `do()` возвращает значение `false` только в случае ошибки.

Хорошим методом DBI, о котором следует помнить, является `quote()`, особенно при динамическом создании команд SQL и особенно при вставке новых данных с помощью `do()`. Этот метод правильно расставляет кавычки в строках литералов в команде SQL, прежде чем передать ее базе данных. Мы рассказывали о нем в главе 4.

Привязка параметров к командам

Одним из вопросов, о которых мы упоминали в рассказе о подготовке дескрипторов команд, является *привязка значений* (*bind values*). Вы могли также встречать термины *заполнители* (*placeholders*), *параметры* (*parameters*) и *связывание* (*binding*). Что за ними скрывается?

Связанным является значение, которое можно привязать к заполнителю, объявленному в команде SQL. Это похоже на создание команд SQL «на лету», например:

[1] В действительности Perl использует специальную логику, чтобы применять в таких случаях строку `0 but true`. В DBI она не используется, поскольку когда программа выводит сообщение типа: `print Удалено $rows строк\n`, то `Удалено 0E0 строк` выглядит несколько лучше, чем `Удалено 0 but true строк`.

```
$sth = $dbh->prepare( "
            SELECT name, location
            FROM megaliths
            WHERE name = " . $dbh->quote( $siteName ) . "
        " );
```

Однако вместо вставки полученного значения в команду SQL вы указываете заполнитель, к которому затем привязываете получаемое значение, например:

```
$sth = $dbh->prepare( "
            SELECT name, location
            FROM megaliths
            WHERE name = ?
        " );
$sth->bind_param( 1, $siteName );
```

Метод `bind_param()` осуществляет фактическую связь передаваемой величины с указанным заполнителем. База данных правильно вычленит заполнитель и зарезервирует для него пространство, которое будет «заполнено» при вызове `bind_param()`. Важно помнить о необходимости вызова `bind_param()` *до* вызова `execute()`; в противном случае отсутствующее значение так и останется пустым, и команда не сможет быть выполнена.

Столь же просто задать в одной команде несколько связанных значений, поскольку `bind_param()` принимает индекс параметра (нумерация начинается с 1), с которым должно быть связано передаваемое значение, например:

```
$sth = $dbh->prepare( "
            SELECT name, location
            FROM megaliths
            WHERE name = ?
            AND mapref = ?
            AND type LIKE ?
        " );
$sth->bind_param( 1, "Эйвбери" );
$sth->bind_param( 2, $mapreference );
$sth->bind_param( 3, "%Каменный круг%" );
```

Возможно, вы обратили внимание, что мы не вызывали метод `quote()` для обработки значений. Связанные значения передаются в базу данных отдельно от команды SQL,[1] поэтому нет необходимости «оборачивать» значение согласно правилам SQL.

[1] Это не совсем верно, т. к. некоторые драйверы эмулируют заполнители, осуществляя их замену связанными значениями, до передачи команды SQL базе данных. В таких драйверах используются внутренние установки Perl, чтобы для каждого значения попытаться угадать, требуется ли применение `quote`. Более подробно об этом сообщается в документации драйвера.

Некоторые драйверы баз данных распознают заполнители в виде `:1`, `:2` и т. д. либо даже в виде `:имя` или `:некоторое_значение`, но при этом не гарантирована переносимость приложения между базами данных. Единственной гарантированно переносимой формой заполнителя является отдельный вопросительный знак – `?`. Конечно, если конкретная база данных не поддерживает связывание, драйвер может вообще оказаться не в состоянии произвести правильный синтаксический разбор команды.

Сравнение связанных величин и интерполируемых команд

Итак, зачем используются связанные величины? В чем состоит реальное отличие использования связанных значений от вставки значений в команды SQL «на лету»?

На первый взгляд, очевидных различий нет. При создании команд с помощью вставки (интерполяции) используются функции Perl обработки строк для создания законченной команды SQL, посылаемой базе данных. Связанные значения посылаются базе данных вслед за командой SQL, но перед ее выполнением. В обоих случаях получается один и тот же результат.

Реальное отличие состоит в том, как базы данных обрабатывают связанные значения, если они вообще это делают. Например, большинством крупных систем баз данных поддерживается структура данных, известная как «совместно используемый кэш SQL», в которой хранятся команды SQL наряду с дополнительными данными, такими как *план выполнения запроса*.

Идея заключается в том, что если команда уже есть в кэше SQL, то базе данных не нужно заново обрабатывать команду, чтобы вернуть для нее дескриптор: можно повторно использовать данные, хранимые в кэше. В результате такой процедуры резко возрастает производительность в тех случаях, когда одна и та же команда SQL выполняется многократно.[1]

Допустим, к примеру, что нам нужно получить общие данные о сотне мегалитических площадок, используя для поиска поле названия. С этой целью можно написать такую команду SQL:

```
SELECT name, location, mapref
FROM megaliths
WHERE name = <search_term>
```

[1] Я сталкивался с ситуацией, когда база данных работала дольше минуты лишь для того, чтобы выработать «достаточно хороший» план выполнения для сложного запроса SQL. В таких случаях повторное использование дескриптора обработанного запроса значительно увеличивает производительность.

При использовании SQL со вставкой мы фактически подадим базе данных сто различных команд SQL. Несмотря на их идентичность, они являются различными для базы данных, которая будет производить синтаксический разбор каждой из них, не используя кэшированную информацию. При использовании связанных значений один и тот же SQL и «план выполнения» будут применяться многократно, хотя для каждого запроса используются свои связанные значения.

Поэтому для баз данных, поддерживающих связанные значения, дескрипторы подготовленных команд могут резко увеличить производительность приложений и эффективность работы базы данных. Это особенно существенно при вставке большого числа записей.

Несмотря на сказанное, могут быть веские причины для использования команд SQL со вставкой, а не связанных значений. Одной из причин может быть просто то, что ваша база данных не поддерживает связанные значения! Менее очевидной причиной может быть наличие в базе данных ограничений на то, в каких частях команд SQL допускается использование заполнителей.

В приведенных примерах мы продемонстрировали использование связанных значений в условных предложениях запросов. Предположим, для вредности, что мы хотим перебрать список таблиц в базе данных и вернуть количество строк в каждой из таблиц. Следующий фрагмент программного кода иллюстрирует использование для этой цели команды SQL со вставкой:

```
foreach $tableName ( qw( megaliths, media, site_types ) ) {
    $sth = $dbh->prepare( "
                SELECT count(*)
                FROM $tableName
              " );
    $sth->execute();
    my $count = $sth->fetchrow_array();
    print "Таблица $tableName содержит $count строк\n";
}
```

При использовании команды со вставкой этот код правильно выполнится и даст желаемые результаты, хотя и ценой разбора и выполнения в базе данных четырех различных команд SQL. Можно переписать код с использованием связанных значений, что теоретически должно быть более эффективно:

```
    $sth = $dbh->prepare( "
                SELECT count(*)
                FROM ?
              " );
    $sth->bind_param( 1, $tableName );
    ...
```

На самом деле для большинства баз данных вызов `prepare()` не сможет произвести синтаксический анализ команды, поскольку заполнители обычно используются только для значений литералов. Это связано с тем, что база данных должна обладать достаточной информацией для создания плана выполнения запроса и не может сделать этого при неполной информации, например, не зная имени таблицы.

Следующая программа также не сможет быть выполнена, поскольку производится привязка не просто литеральных выражений:

```
$sth = $dbh->prepare( "
            SELECT count(*)
            FROM megaliths
            ?
" );
$sth->bind_param( 1, "WHERE name = 'Эйвбери'" );
...
```

Конечно, может оказаться, что ваш драйвер поддерживает такого рода запросы, но не следует всерьез полагаться на то, что это сработает с другими базами данных!

Связанные значения и типы данных

Perl является слабо типизированным языком, т. е. у вас есть строки и есть числа. Числа могут выступать как строки, а строки иногда могут оказаться числами. Над строками можно производить арифметические операции. Подобное может сбить с толку и программиста, так что можете себе представить положение драйвера, когда он сталкивается со связанными значениями.

Чтобы помочь драйверу разобраться с тем, какого типа данные передаются ему в связанной величине, можно задать дополнительный аргумент, определяющий тип данных. Например, в следующей программе с выполняемой в базе данных командой связаны надлежащим образом типизированные величины:

```
use DBI qw(:sql_types);

$sth = $dbh->prepare( "
            SELECT meg.name, meg.location, st.site_type, meg.mapref
            FROM megaliths meg, site_types st
            WHERE name = ?
            AND id = ?
            AND mapref = ?
            AND meg.site_type_id = st.id
          " );
### Тип данных для этого значения не требуется, это строка
$sth->bind_param( 1, "Эйвбери" );
```

```
### Очевидно, это число; снова не указываем тип
$sth->bind_param( 2, 21 );

### А здесь строка, выглядящая как число
$sth->bind_param( 3, 123500, { TYPE => SQL_VARCHAR } );

### Альтернативная сокращенная форма предыдущей команды
$sth->bind_param( 3, 123500, SQL_VARCHAR );

### Теперь со всеми заполнителями связаны значения,
### можно выполнить команду
$sth->execute();
```

Команда `use DBI qw(:sql_types);` запрашивает импорт стандартных типов SQL в виде имен, а фактически – подпрограмм, возвращающих стандартные целые значения типов SQL. Например, `SQL_VARCHAR` возвращает значение `12`. Если вы не хотите импортировать имена типов SQL, можете добавить префикс `DBI::` и вместо `SQL_VARCHAR` использовать `DBI::SQL_VARCHAR`. Однако делать это не рекомендуется, поскольку теряются преимущества проверки во время компиляции по директиве `use strict;`.

Если тип указан, драйвер должен воспринять его как настоятельный совет относительно того, что нужно делать. Но это всего лишь совет. Некоторые драйверы не обращают никакого внимания на указанный тип данных, а те, которые это делают, в основном используют его для различения строк, чисел и типов LONG/LOB. Это относительно новая область для DBI и драйверов, которая медленно прогрессирует.

Обычно базы данных поддерживают значительно более широкий диапазон типов данных, чем числа и строки. Очень распространены типы данных «дата», имеющие весьма разнообразные форматы. В настоящее время DBI успешно справляется с ними, уклоняясь от проблемы и рассчитывая на получение строк, содержащих данные, отформатированные согласно требованиям конкретной базы данных для соответствующего типа данных.[1]

Связывание входных и выходных параметров

В дополнение к методу `bind_param()` существует метод `bind_param_inout()`, который можно использовать для того, чтобы «стащить» значения, возвращаемые командой. Обычно он полезен только в связи с хранимыми процедурами, принимающими входные параметры и возвращающими значения. Более того, этот метод поддерживают немногие базы данных и еще меньшее число драйверов, так что будьте осторожны.

[1] В будущих версиях могут появиться функции `escape` в стиле ODBC.

bind_param_inout() ведет себя сходным с bind_param() образом, но использует не само связываемое значение, а *ссылку* на него. Это позволяет заменять связанное значение тем, которое возвращает команда.

Необходимо также задать дополнительный аргумент, устанавливающий максимальный размер возвращаемого значения. Если размер возвращаемого значения превосходит максимально допустимый, это приводит к неудаче при выполнении execute(). Поэтому, если вы не уверены в том, какой размер может иметь возвращаемое значение, следует придерживаться пессимистической оценки и задать для этого параметра большое значение. Единственным недостатком при этом станет расход большего объема памяти, чем требуется.

Последний и необязательный аргумент задает тип данных связанного значения. Его действие аналогично установлению типа данных в bind_param(). О том, как задавать значения для этого аргумента, можно узнать из предыдущего раздела.

Ориентированный на Oracle пример, демонстрирующий работу bind_param_inout(), использует следующую хранимую процедуру, которая для заданного входного числа возвращает ближайшее к нему целое число:

```
-- Пример хранимой процедуры, написанной на Oracle PL/SQL
PROCEDURE ceiling_floor (value IN NUMBER, c OUT NUMBER, f OUT NUMBER) IS
BEGIN
    c := CEIL(value);
    f := FLOOR(value);
END;
```

Программу DBI для получения значений, возвращаемых этой процедурой, можно написать так:

```
### Переменные, в которые будут занесены возвращенные значения ...
my $ceiling;
my $floor;

$sth = $dbh->prepare( "BEGIN ceiling_floor( ?, ?, ? ); END;" );
$sth->bind_param( 1, 42.3 );
$sth->bind_param_inout( 2, \$ceiling, 50 );
$sth->bind_param_inout( 3, \$floor, 50 );
$sth->execute();
print "Хранимая процедура возвратила $ceiling, $floor\n";
```

С одним и тем же дескриптором команды можно использовать как bind_param(), так и bind_param_inout(). Разумеется, если вы используете bind_param(), когда ожидается возврат значения, это значение будет утрачено.

Стоит упомянуть об одном тонком различии между bind_param() и bind_param_inout(). При вызове bind_param() связываемое значение, которое вы передаете, копируется и не меняется, пока вы снова не вызовете

bind_param(). При вызове же bind_param_inout() копируется *ссылка*. Фактическое значение, на которое указывает ссылка, не считывается, пока не будет произведен вызов execute().

Связывание значений без использования bind_param()

Вызов bind_param() для каждого заполнителя может оказаться довольно скучным и утомительным делом, если заполнителей много, поэтому DBI предоставляет более простой способ сделать это с помощью метода execute(). При вызове execute() можно просто передать список значений, и execute() вызовет bind_param() для каждого из них.

Кроме того, описанный выше метод do(), а также методы selectrow_array() и selectall_arrayref(), о которых мы вскоре расскажем, тоже так или иначе вызывают execute() и также могут принимать список связанных значений.

Следующий пример иллюстрирует передачу связанных значений методу execute():

```
$sth = $dbh->prepare( "
            SELECT name, location, mapref
            FROM megaliths
            WHERE name = ? OR description LIKE ?
          " );
$sth->execute( "Эйвбери", "%крупнейший каменный круг%" );
...
```

При указании связанных значений таким способом явное определение типа передаваемых значений невозможно. В некоторых случаях драйвер правильно угадывает тип данных, но, как правило, все значения передаются базе данных с типом SQL_VARCHAR. Однако если ранее вы уже вызывали bind_param() или bind_param_inout() для каких-либо заполнителей с явным указанием типа данных, то будет использоваться именно этот тип данных. Например, в результате выполнения

```
$sth->prepare( "
            SELECT name, location, mapref
            FROM megaliths
            WHERE id = ?
          " );
$sth->bind_param( 1, 42, SQL_INTEGER );
$sth->execute( 123456 );
...
```

значение 123456 будет передано базе данных как величина, связанная с типом SQL_INTEGER, а не SQL_VARCHAR.

Связывание выходных колонок

В примерах выборки данных, с которыми мы до сих пор сталкивались, использовался метод fetch(), возвращавший значения, которые копировались в переменные Perl. Например:

```
while( ( $foo, $bar ) = $sth->fetchrow_array ) { ... }
```

Этот синтаксис работает прекрасно, но становится несколько запутанным, если возвращается много полей. Кроме того, осуществляется лишнее копирование данных, что может оказаться накладным, если извлекается много больших строк.

DBI поддерживает функцию, упрощающую выборку данных и позволяющую избежать дополнительного копирования. В результате достигается эффект быстрой выборки. Эта функция известна как *связывание колонок* и работает путем указания переменных Perl, в которые непосредственно записываются значения колонок при выборке. В результате при выборке данных из базы с помощью метода fetch()[1] переменные Perl, связанные с каждой из колонок, автоматически обновляются выбранными значениями.

Лучше всего проиллюстрировать этот процесс на примере:

```
### Переменные Perl, в которые будут записаны данные из полей
my ( $name, $location, $type );

### Подготовить и выполнить команду SQL
$sth = $dbh->prepare( "
            SELECT meg.name, meg.location, st.site_type
            FROM megaliths meg, site_types st
            WHERE meg.site_type_id = st.id
          " );
$sth->execute();

### Связать переменные Perl с каждой из выходных колонок
$sth->bind_col( 1, \$name );
$sth->bind_col( 2, \$location );
$sth->bind_col( 3, \$type );

### Выбрать данные из результирующего набора
while ( $sth->fetch ) {
    print "$name имеет тип $type и находится в $location\n";
}
```

Метод, который мы использовали для явного связывания переменных Perl с выходными колонками, называется bind_col(). Он принимает

[1] fetch() является просто удобным коротким псевдонимом для fetchrow_arrayref().

номер связываемой колонки, начиная с 1, и ссылку на переменную Perl, с которой эту колонку нужно связать. В результате по завершении вызова `fetch()` связанные переменные Perl будут автоматически обновлены без явного присвоения выбранных значений. Это очень эффективный способ выборки данных из базы как с точки зрения программирования, так и с точки зрения производительности. `bind_col()` использует ссылки на переменные Perl, благодаря чему при использовании связанных выходных колонок нет необходимости в дополнительном выделении памяти и размещении объектов.

Для обеспечения максимальной переносимости следует применять `bind_col()` к выполненному дескриптору команды. Например, если ваша база данных не возвращает фактические данные при вызове `prepare()`, то у `bind_col()` будет недостаточно информации для того, чтобы успешно связать выходные колонки с переменными Perl. Это может привести к крайне неожиданным результатам.

Явное использование `bind_col()` для отдельной привязки каждой колонки может оказаться несколько утомительным, особенно если используется много колонок. К счастью, в DBI имеется другой метод с именем `bind_columns()`, который можно использовать для быстрой привязки колонок за одно обращение.

`bind_columns()` работает почти так же, как `bind_col()`, за исключением того, что вместо явного указания номера колонки, к которой должна быть привязана переменная Perl, нужно просто перечислить переменные Perl, и назначение колонок произойдет автоматически. Например, предыдущую программу можно переписать следующим образом с использованием метода `bind_columns()`:

```
### Переменные Perl, в которые будут записаны данные из полей
my ( $name, $location, $type );

### Подготовить и выполнить команду SQL
$sth = $dbh->prepare( "
            SELECT meg.name, meg.location, st.site_type
            FROM megaliths meg, site_types st
            WHERE meg.site_type_id = st.id
          " );
$sth->execute();

### Связать переменные Perl с каждой из выходных колонок
$sth->bind_columns( undef, \$name, \$location, \$type );

### Выбрать данные из результирующего набора
while ( $sth->fetch ) {
    print "$name имеет тип $type и находится в $location \n";
}
```

Важно отметить, что число колонок, заданных в команде SQL, и число переменных Perl, заданных в `bind_columns()`, должны в точности совпа-

дать. Нельзя отобрать колонки, из которых должна производиться выборка, как это можно сделать с использованием bind_col().[1]

Поскольку bind_columns() внутри себя использует bind_col(), правила использования этих методов одинаковы.

Отметим в заключение, что связывание значений, задаваемых для команды SQL, никак не связано с возможностью привязки переменных Perl к выходным колонкам команды SQL. Это отдельные операции. Связанные значения действуют на входном уровне базы данных, в то время как привязка выходных колонок действует исключительно на выходном уровне Perl.

Сравнение do() и prepare()

Как говорилось в предыдущем разделе, использование метода do() значительно облегчает выполнение команд, отличных от SELECT, по сравнению с периодической подготовкой и выполнением команд. Это достигается просто благодаря включению этапов подготовки и выполнения в один составной метод.

Однако такой способ имеет отрицательную сторону – снижение производительности. При повторяющемся вызове do() для вставки в таблицу большого числа строк подготовка дескриптора команды может происходить большее число раз, чем это действительно необходимо, особенно если команда содержит заполнители. Например, в следующем сценарии вставляется несколько строк в таблицу megaliths:

```
### Повторить с различными данными...
foreach $item ( qw( Стонхендж Эйвбери Каслригг Санхани ) ) {
    ### ... и вставить их в таблицу
    $dbh->do( "INSERT INTO megaliths ( name ) VALUES ( ? )",
              undef, $name );
}
```

В результате для каждой вставляемой строки создается новый дескриптор команды, команда подготавливается, выполняется и в конце концов уничтожается. Поэтому в данном цикле осуществляется четыре вызова подготовки, четыре исполнения и четыре уничтожения. Однако поскольку при каждом выполнении цикла используется связанное значение, базе данных, возможно, требуется лишь один раз произвести разбор команды, а затем использовать эту команду из совместно используемого кэша SQL. Поэтому наша программа, по существу, напрасно производит три подготовки этой команды.

[1] Первым аргументом bind_columns() является undef, что связано с историческими причинами. Если вы используете DBI 1.08 или более позднюю версию, его задавать не требуется.

Это весьма неэффективный процесс. В данном случае было бы лучше вручную подготовить дескриптор команды и при каждом проходе цикла повторно осуществить выполнение для дескриптора, например:

```
### Настроить команду для повторного выполнения
$sth = $dbh->prepare( "INSERT INTO megaliths ( name ) VALUES ( ? )" );

### Повторить с различными данными...
foreach $item ( qw( Stonehenge Avebury Castlerigg Sunhoney ) ) {
    ### ... и вставить их в таблицу
    $sth->execute( $name );
}
```

В этой программе подготовка команды осуществляется только один раз, а выполнение ее осуществляется четыре раза – по одному для каждой вставляемой строки. Такую программу несколько менее удобно писать, но выполнение ее обычно происходит значительно быстрее.

Если уж мы заговорили о скорости вставки, то важно отметить, что существует и более быстрый способ. Использование сценария Perl совместно с DBI едва ли сможет достичь той скорости массовой загрузки записей, которую обеспечивают специально оптимизированные средства поставщиков баз данных, такие как SQL*Loader Oracle или BCP Sybase. Но не падайте духом; Perl является идеальным средством для создания и обработки файлов данных, используемых этими загрузчиками.

Атомарная и пакетная выборки

Атомарная и *пакетная* выборки представляют собой два несколько более интересных способа извлечения данных из базы. Эти процедуры несколько схожи между собой в том смысле, что могут значительно облегчить вам жизнь, но осуществляют это принципиально различными способами.

Атомарная выборка

Если вам нужно извлечь только одну строку данных, атомарная выборка позволяет уплотнить состоящий из четырех этапов цикл выборки (описанный выше) в одном методе. Для атомарной выборки можно использовать два метода – `selectrow_array()` и `select_row_arrayref()`. Они ведут себя аналогично своим ориентированным на строки родственникам – `fetchrow_array()` и `fetchrow_arrayref()`, и главное отличие состоит в том, что для работы этих двух атомарных методов не требуется наличия подготовленного и выполненного дескриптора и, что более важно, они возвращают только одну строку данных.

Поскольку эти методы не требуют использования дескриптора команды, они вызываются через дескриптор базы данных. Например, чтобы выбрать поля `name` и `type` из произвольной строки нашей таблицы `megaliths`, можно написать такой код:

```
### Предполагая существование активного $dbh...
( $name, $mapref ) =
    $dbh->selectrow_array( "SELECT name, mapref
                            FROM megaliths" );
print "Мегалит $name расположен в $mapref\n";
```

Это значительно удобнее, чем использовать методы `prepare()`, `execute()` и затем `fetchrow_array()` или `fetchrow_arrayref()` для извлечения одной строки.

Наконец, можно использовать связанные значения, что способствует эффективному использованию ресурсов базы данных.

Пакетная выборка

Пакетная выборка дает возможность получить результирующий набор запроса SQL целиком за одно обращение, в противоположность использованию построчных методов, таких как `fetchrow_array()`, при последовательном проходе по результирующему набору.

В DBI для этой цели определено несколько методов, в том числе `fetchall_arrayref()` и `selectall_arrayref()`, которые, по существу, извлекают весь результирующий набор в структуру данных Perl, которую можно обрабатывать.

`fetchall_arrayref()` может работать в трех различных режимах в зависимости от переданных ему аргументов. Можно вызвать его без аргументов со ссылкой на срез массива в качестве аргумента или со ссылкой на срез хэша. Эти режимы мы обсудим в следующих разделах.

Без аргументов

Когда метод `fetchall_arrayref()` вызывается без аргументов, он возвращает ссылку на массив, содержащий ссылки на каждую строку результирующего набора. Каждая из этих ссылок указывает на массив, содержащий значения полей в этой строке. На рис. 5.4 показана возвращаемая структура данных.

Это кажется довольно запутанным, но на самом деле доступ к данным, хранимым в этой структуре, осуществляется очень просто. Например, следующий код показывает, как разыменовывать структуру данных, возвращаемую `fetchall_arrayref()` при запуске этого метода без аргументов:

```
#!/usr/bin/perl -w
#
```

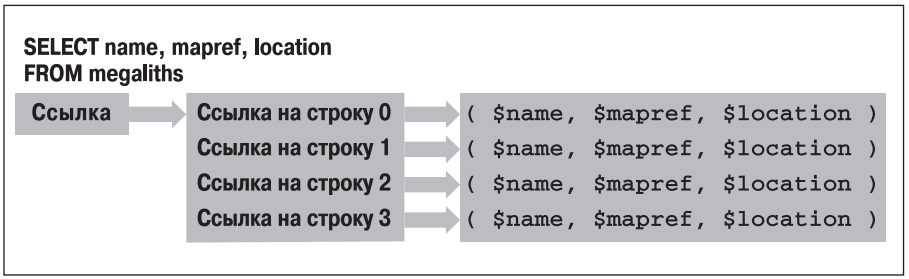

***Рис. 5.4.** Структура данных fetchall_arrayref( )*

```
# ch05/fetchall_arrayref/ex1: Полный пример, осуществляющий соединение
#                             с базой данных, выполняющий команду SQL
#                             и затем осуществляющий выборку всех
#                             строк данных в структуру данных. Затем
#                             осуществляется проход по этой структуре
#                             и ее вывод.

use DBI;

### Дескриптор базы данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1
});

### Дескриптор команды
my $sth = $dbh->prepare( " SELECT name, location, mapref FROM megaliths " );

### Выполнение команды
$sth->execute();

### Выборка всех данных в структуру данных Perl
my $array_ref = $sth->fetchall_arrayref();

### Проход по структуре данных и вывод дампа каждой части
###
### Для каждой строки в возвращенной ссылке на массив...
foreach my $row (@$array_ref) {
    ### Расщепить строку и вывести каждое поле ...
    my ( $name, $type, $location ) = @$row;
    print "\tМегалитическая площадка $name, находится в $location,
           имеет тип $type\n";
}

exit;
```

Следовательно, при необходимости сделать выборку всего результирующего набора `fetchall_arrayref()` предоставляет эффективный и простой способ сделать это. Это тем более правильно, если вы собираетесь

создавать в памяти структуру данных, содержащую возвращенные строки для последующей их обработки. Вместо того чтобы делать это самостоятельно, можно просто воспользоваться результатом работы `fetchall_arrayref()`.

Передача ссылки на срез массива

Можно использовать `fetchall_arrayref()` для возврата структуры данных, содержащей лишь определенные колонки из каждой строки результирующего набора. Например, можно создать команду SQL, выбирающую поля `name`, `site_type`, `location` и `mapref`, но создать при этом в памяти структуру, содержащую только поля `name` и `location`.

В стандартном режиме `fetchall_arrayref()` без аргументов этого сделать нельзя, но можно легко осуществить, передав `fetchall_arrayref()` в качестве аргумента срез массива.

Если исходная команда SQL имеет вид:

```
SELECT meg.name, st.site_type, meg.location, meg.mapref
FROM megaliths meg, site_types st
WHERE meg.site_type_id = st.id
```

то индексы в массиве для каждой возвращаемой строки имеют следующее соответствие:

```
name       -> 0
site_type  -> 1
location   -> 2
mapref     -> 3
```

Зная эти индексы массива для колонок, можно просто написать:

```
### Извлечь поля name и location...
$array_ref = $sth->fetchall_arrayref( [ 0, 2 ] );
```

Индексы массива указываются в форме, стандартной для Perl, поэтому в особых случаях можно использовать диапазоны и отрицательные индексы, например:

```
### Извлечь предпоследнюю и последнюю колонки
$array_ref = $sth->fetchall_arrayref( [ -2, -1 ] );

### Извлечь колонки с первой по третью
$array_ref = $sth->fetchall_arrayref( [ 0 .. 2 ] );
```

Фактическая структура данных, создаваемая при таком использовании `fetchall_arrayref()`, аналогична по форме той структуре, которая создается при вызове `fetchall_arrayref()` без аргументов.

Передача ссылки на срез хэша

Последний способ, которым можно использовать fetchall_arrayref() для избирательной записи колонок в ссылку на массив, состоит в передаче в качестве аргумента ссылки на хэш, где содержатся нужные колонки. Это похоже на использование метода fetchrow_hashref(), но при этом возвращается ссылка на массив, содержащий ссылки на хэши для всех строк результирующего набора.

Если нужно сохранить только колонки name и location из команды SQL, определенной как

```
SELECT name, location, mapref
FROM megaliths
```

то можно сообщить fetchall_arrayref() о необходимости сохранить соответствующие поля, передав ему в качестве аргумента анонимный хэш. Этот хэш должен быть инициализирован таким образом, чтобы в нем содержались имена колонок, которые нужно сохранить.

Например, для записи колонок name и location можно написать такой код:

```
### Сохранить колонки name и location
$array_ref = $sth->fetchall_arrayref( { name => 1, location => 1 } );
```

Структура данных, создаваемая fetchall_arrayref() при работе в таком режиме, является ссылкой на массив ссылок на хэши, для которых ключом является имя колонки, а значением – значение, содержащееся в этой колонке и соответствующей строке. Просмотреть эту структуру данных очень просто. В следующей программе показано, как это можно сделать:

```
#!/usr/bin/perl -w
#
# ch05/fetchall_arrayref/ex3: Полный пример, осуществляющий соединение
#                             с базой данных, выполняющий команду SQL
#                             и осуществляющий выборку всех строк
#                             данных в структуру данных. Затем
#                             осуществляется проход по этой структуре
#                             и ее вывод.

use DBI;

### Дескриптор базы данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1,
} );

### Дескриптор команды
my $sth = $dbh->prepare( " SELECT name, location, mapref FROM megaliths " );
```

```
### Выполнение команды
$sth->execute();

### Выборка всех данных в ссылку на массив ссылок на хэши!
my $array_ref = $sth->fetchall_arrayref( { name => 1, location => 1 } );

### Проход по структуре данных и вывод дампа каждой части
###
### Для каждой строки в возвращенной ссылке на массив.....
foreach my $row (@$array_ref) {
    ### Выбрать соответствующие поля из ссылки на хэш и вывести их...
    print "\ tМегалитическая площадка $row->{name}, находится в
          $row->{location}\n";
}

exit;
```

Есть пара важных моментов, на которые нужно обратить внимание при работе с такой формой выборки результирующего набора:

- При выполнении команды SQL, в которой несколько колонок имеют одинаковое имя, в возвращаемых хэшах для всех колонок будет только один элемент. Ранее созданные элементы будут перезаписаны и утеряны. То же замечание относится к `fetchrow_hashref()`, поскольку этот метод внутренне вызывается `fetchall_arrayref()` при передаче ему среза хэша.

  Вот пример SQL, с которым возникнут проблемы:

  ```
  SELECT m.name, c.name
  FROM megaliths m, countries c
  WHERE m.country_id = c.id
  ```

  В этом примере хэши для строк будут содержать значение колонки либо из `countries`, либо из `megaliths`, но не оба одновременно.

- Второе замечание относительно использования `fetchall_arrayref()` состоит в том, что названия колонок, хранимые в возвращаемом кэше, всегда находятся в нижнем регистре. Регистр, используемый базой данных, и регистр, используемый в параметре, передаваемом `fetchall_arrayref()`, игнорируются.

В заключение отметим, что пакетная выборка значений является удобным способом извлечения всех данных результирующего набора в структуры данных Perl для последующей их обработки. Следует, однако, помнить, что большие результирующие наборы съедают и большой объем памяти. Если попытаться осуществить выборку слишком большого набора данных, то память израсходуется раньше, чем произойдет возврат из метода. Вашему системному администратору это может прийтись не по душе.

6

Более сложные вопросы использования DBI

В этой главе рассказывается о некоторых более сложных темах, связанных с использованием DBI, в том числе о возможности изменения «на лету» способов функционирования базы данных и дескрипторов команд, а также о том, как использовать явную обработку транзакций в базе данных. Эти темы не являются строго необходимыми для базового использования DBI, но в них содержатся полезные сведения, позволяющие увеличить возможности программ DBI.

Атрибуты дескрипторов и метаданные

Помимо методов, связанных с дескрипторами баз данных и команд, в DBI также определены *атрибуты* этих дескрипторов, позволяющие разработчикам производить исследование или тонкую настройку среды, в которой действуют эти дескрипторы. Некоторые атрибуты присущи только отдельным дескрипторам баз данных или команд, а некоторые являются общими для тех и других.

Значения атрибутов дескрипторов можно представлять себе в виде хэша, содержащего пары ключ/значение, и с ними можно работать через ссылку так же, как с обычным хэшем. Вот несколько примеров использования атрибута `AutoCommit`:

```
### Установить атрибут дескриптора базы данных "AutoCommit"
### равным 1 (т.е. включить)
$dbh->{AutoCommit} = 1;

### Получить текущее значение "AutoCommit" из дескриптора
$foo = $dbh->{AutoCommit};
```

Выборка атрибутов как значений хэша, а не в результате вызова методов, имеет дополнительное преимущество, состоящее в том, что подстановку из хэша можно *интерполировать* в строках, заключенных в двойные кавычки:

```
### Вывести текущее значение "AutoCommit" из дескриптора
print "AutoCommit: $dbh->{AutoCommit}\n";
```

Если `AutoCommit` включен, будет выведено:

```
AutoCommit: 1
```

как и ожидалось. Фактически, поскольку `AutoCommit` является *булевым* атрибутом, `1` будет выведена для любого значения, которое Perl сочтет *истинным* (*true*).

Если атрибуту присвоено значение *ложь* (*false*), разумно ожидать, что будет выведен `0`, но, к вашему удивлению, вы увидите:

```
AutoCommit:
```

Это связано с тем, что для Perl внутреннее представление `false` может быть одновременно и числом ноль и пустой строкой. При использовании в строковом контексте выводится пустая строка, в числовом контексте выводится ноль.

При чтении или установке значения атрибута DBI автоматически проверяет имя атрибута и генерирует ошибку, если имя неизвестно.[1] Аналогично любая попытка установить атрибут, определенный только для чтения, приводит к возникновению ошибки. Учтите, однако, что об этих ошибках сообщается через `die()` независимо от состояния атрибута `RaiseError`, что потенциально ведет к выходу из программы. Это еще одна причина для использования блоков `eval {...}`, о чем говорилось в главе 4 «Программирование с использованием DBI».

Дескриптор команды является потомком своего родительского дескриптора базы данных. Аналогично сами дескрипторы баз данных являются потомками своих родительских дескрипторов драйверов. Дескрипторы-потомки наследуют от своих родителей значения некоторых атрибутов. Правила этого наследования определены в соответствии с соображениями здравого смысла и являются следующими:

- Дескриптор команды наследует (копирует) текущие значения некоторых атрибутов родительского дескриптора базы данных.

[1] Специфические для драйвера атрибуты, например, те, которые начинаются со строчной буквы, представляют собой особый случай. Всякие чтение или установка специфического для драйвера атрибута, не обработанные драйвером, обрабатываются DBI без возбуждения ошибки. Это облегчает жизнь разработчикам драйверов, но требует повышенного внимания для указания имени без ошибок.

- Изменение значений атрибутов этого нового дескриптора команды не оказывает никакого влияния ни на родительский дескриптор базы данных, ни на другие дескрипторы команд. Все изменения относятся исключительно к измененному дескриптору команды.
- Изменения атрибутов дескриптора базы данных не оказывают влияния на существующие дескрипторы команд, являющиеся его потомками. Изменения атрибутов дескриптора базы данных будут учтены только в дескрипторах команд, которые будут порождены им позднее.

Полные данные о том, какие атрибуты наследуются, можно получить в спецификации DBI, представленной в приложении А.

Передача атрибутов методам DBI

Дескрипторы содержат набор текущих значений атрибутов, которые часто используются для управления действием методов. Многие методы также могут принимать в качестве необязательного аргумента ссылку на хэш, содержащий значения атрибутов.

Это является, в основном, запасным механизмом для разработчиков и пользователей драйверов, и потому не всегда работает так, как вы этого ожидаете. Например, вы можете полагать, что программа:

```
$dbh->{RaiseError} = 1;
...
$dbh->do( $sql_statement, undef, { RaiseError => 0 } ); # НЕВЕРНО
```

сбросит `RaiseError` при вызове метода `do()`. Но этого не происходит! Параметры атрибута *игнорируются* DBI при вызовах *всех* методов дескрипторов баз данных и команд. Вы даже не получите предупреждения о том, что атрибут проигнорирован.

Если параметры игнорируются, то зачем тогда вообще они нужны? Что касается DBI, то он их игнорирует, но драйвер `DBD`, обрабатывающий метод, может и не игнорировать. А может и игнорировать! Параметры хэша атрибутов для методов являются *советами* драйверу и обычно приносят пользу, только если содержат специфические для драйвера атрибуты.[1]

Сказанное не относится к методу `DBI->connect()`, поскольку это метод не драйвера, а DBI. Его параметр с хэшем атрибутов `\%attr` *используется* для установки атрибутов вновь создаваемого дескриптора базы данных. Мы дали несколько примеров использования `RaiseError` в главе 4 и приведем еще целый ряд в следующем разделе.

[1] Не исключено, что в будущих версиях DBI станут учитываться некоторые неспецифичные для драйверов атрибуты, такие как `RaiseError`.

Установление соединения с использованием атрибутов

Одной из многочисленных крылатых фраз Perl стал афоризм «существует более одного способа сделать это», и DBI не является исключением. Кроме возможности установки атрибутов дескриптора путем простого присваивания и использования параметра атрибутов метода `connect()` (как было показано раньше), DBI предоставляет еще один способ.

Можно включить присвоение значений атрибутам в параметр имени источника данных метода `connect()`, например:

```
$dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1
});
```

можно также записать в виде:

```
$dbh = DBI->connect( "dbi:Oracle(RaiseError=>1):archaeo", '', '');
```

Нельзя оставлять пробелы перед открывающей скобкой или между закрывающей и последующим двоеточием, но внутри скобок пробелы можно помещать. Можно также при желании использовать «=» вместо «=>». При необходимости установки более чем одного параметра их следует разделять запятыми.

Установка атрибутов в параметре имени источника данных имеет приоритет перед установкой в параметре атрибутов. Это очень удобно, когда нужно переопределить жестко зашитую в код установку таких атрибутов, как `PrintError`. Например, в следующей программе `PrintError` остается включенным:

```
$dbh = DBI->connect( "dbi:Oracle(PrintError=>1):archaeo", '', '', {
    PrintError => 0
});
```

Но какой смысл жестко программировать установку атрибута в двух разных местах? В таком виде этот пример не очень полезен, но мы могли бы позволить приложению получать имя источника данных из параметров командной строки либо оставить его пустым и использовать переменную окружения `DBI_DSN`. В результате приложение становится значительно более гибким.

Важность регистра

Вы могли обратить внимание, что в некоторых именах атрибутов используются только буквы верхнего регистра, как в `NUM_OF_FIELDS`, а в других используется смешанный регистр, как в `RaiseError`. Если вы

смотрели описания отдельных драйверов баз данных, то могли заметить, что в некоторых именах атрибутов используются только буквы нижнего регистра, например `ado_conn` и `ora_type`.

За этой кажущейся непоследовательностью скрывается важная система. Регистр букв, используемых в именах атрибутов, имеет значение и играет важную роль в переносимости сценариев DBI и расширяемости самого DBI. Регистр букв имени атрибута используется для обозначения того, *кто* определил смысл этого имени и его значение, по следующей схеме:

Верхний регистр (UPPER_CASE)

Имена атрибутов, в которых используются только прописные буквы и символ подчеркивания, определены во внешних стандартах, таких как ISO SQL или ODBC.

Хорошим примером служит атрибут `TYPE` дескриптора команды. Это атрибут верхнего регистра, поскольку возвращаемые им значения являются стандартными переносимыми номерами типов данных, как они определены в ISO SQL и ODBC, а не непереносимыми числами собственных типов базы данных.

Смешанный регистр (mixedCase)

Имена атрибутов, начинающиеся с буквы в верхнем регистре, но включающие также буквы нижнего регистра, определены в спецификации DBI.

Нижний регистр (lower_case)

Имена атрибутов, начинающиеся с букв в нижнем регистре, определены в отдельных драйверах баз данных. Они известны как специфические для драйвера атрибуты.

Поскольку их назначение устанавливается авторами драйверов без какого-либо централизованного контроля, важно, чтобы два автора драйверов не выбрали одинаковое имя для атрибутов, имеющих разное назначение. Для этого все специфические для драйвера атрибуты начинаются с префикса, указывающего на конкретный драйвер. Например, все атрибуты драйвера `DBD::ADO` начинаются с `ado_`, атрибуты `DBD:: Informix` начинаются с `ix_` и т. д.

В большинстве драйверов есть специфическая версия атрибута `TYPE` дескриптора команды, которая возвращает не числа стандартных типов данных, а присущие конкретной базе данных типы. В `DBD::Oracle` она называется `ora_type`, в `DBD::Ingres` это `ing_ingtype`, а в `DBD::mysql` она называется `mysql_type`. Префикс также облегчает нахождение в приложениях специфического для драйвера кода при необходимости их сопровождения.

Специфические для драйвера атрибуты играют важную роль в DBI. Они являются аварийным клапаном. Они позволяют в большей ме-

ре раскрывать имеющиеся в них функции и данные без необходимости втискивать их в узкие рамки DBI.

Общие атрибуты

Общими являются атрибуты, которые можно запрашивать и устанавливать как в дескрипторах баз данных, так и в дескрипторах команд. В этом разделе описываются некоторые наиболее часто используемые атрибуты, в том числе:

PrintError

Будучи установлен, атрибут `PrintError` побуждает DBI выдать предупреждение, когда метод DBI возвращает состояние ошибки. Эта функция очень полезна для осуществления быстрой отладки программ, поскольку явная проверка возвращаемого значения может осуществляться в программе не после каждой команды DBI.

В выводимой строке сообщения об ошибке перечисляются класс драйвера базы данных, через который был направлен метод DBI, метод, вызвавший возникновение ошибки, и значение `$DBI::errstr`. Следующее сообщение было выведено после безуспешного выполнения метода `prepare()` в базе данных Oracle7 с использованием драйвера `DBD::Oracle`:

```
DBD::Oracle::db prepare failed: ORA-00904:
   invalid column name (DBD: error possibly near <*> indicator
   at char 8 in '
           SELECT <*>nname, location, mapref
           FROM megaliths
     ') at /opt/WWW/apache/cgi-bin/megalith/megadump line 79.
```

Для выдачи сообщения об ошибке `PrintError` использует стандартную функцию Perl с именем `warn()`. Поэтому возможно использование обработчика ошибок `$SIG{__WARN__}` или модуля обработки ошибок, например `CGI::ErrorWrap`, для перенаправления ошибок от `PrintError`.

По умолчанию этот атрибут установлен.

RaiseError

Атрибут `RaiseError` по стилю аналогичен `PrintError`, но несколько отличается по действию. В то время как `PrintError` просто выводит сообщение, когда DBI обнаруживает, что произошла ошибка, `RaiseError` обычно «убивает» программу наповал.

`RaiseError` использует стандартную функцию Perl `die()` для генерации исключительной ситуации и выхода. Это означает, что можно использовать для перехвата исключительной ситуации `eval` и са-

мостоятельно ее обрабатывать.[1] Это важная и ценная стратегия обработки ошибок для крупных приложений, весьма рекомендуемая при использовании транзакций.

Формат сообщения об ошибке, выводимой RaiseError, идентичен используемому PrintError. Если определены оба атрибута PrintError и RaiseError, то PrintError пропускается, если не установлен обработчик $SIG{__DIE__}.[2]

RaiseError по умолчанию выключен.

ChopBlanks

Этот атрибут регулирует поведение драйвера базы данных в отношении типа данных CHAR в символьных колонках фиксированной ширины с дополнением пробелами. При установке значения этого атрибута в истинное в любых колонках типа CHAR, возвращаемых командой SELECT, завершающие пробелы будут отсекаться. На другие типы данных этот атрибут не действует, даже если присутствуют концевые пробелы.

ChopBlanks обычно устанавливается, когда нужно просто удалить завершающие пробелы, не используя при этом явно специального кода для усечения ни в исходной команде SQL, ни в Perl.

Это может оказаться очень полезным механизмом при работе со старыми базами данных, в которых чаще используются типы данных CHAR с фиксированной длиной и дополнением пробелами, чем типы VARCHAR. Дописываемые базой данных пробелы попадают под действие этого атрибута.

В настоящее время этот атрибут по умолчанию выключен.

LongReadLen и LongTruncOk

Многие базы данных поддерживают типы данных BLOB (большой двоичный объект), LONG и аналогичные им для хранения в одном поле очень длинных строк или большого объема двоичных данных. Некоторые базы данных поддерживают значения переменной длины размером свыше 2 000 000 000 байт.

Поскольку значения такого размера обычно невозможно хранить в памяти, а базы данных, как правило, не могут заранее сообщить, каков максимальный размер данных типа LONG, которые могут быть возвращены командой SELECT (в отличие от других типов данных), необходимы специальные методы обработки. В таких случаях используется атрибут LongReadLen, чтобы определить, какой

[1] Это также позволяет определить обработчик $SIG{__DIE__}, который обрабатывает вызов die() вместо стандартного поведения Perl.

[2] В будущих версиях может пропускаться PrintError, если установлен RaiseError и текущий код выполняется в блоке eval.

объем памяти должен быть выделен для буфера при выборке таких полей.

LongReadLen по умолчанию устанавливается равным 0 или небольшому значению типа 80, в результате чего выбирается небольшая часть данных типа LONG или их выборка не происходит вообще. Если вы хотите осуществлять выборку каких-либо данных типа LONG, следует присвоить атрибуту LongReadLen в вашем приложении значение несколько большее, чем размер самого длинного поля, которое вы предполагаете извлечь. Установка слишком большого значения приводит только к непроизводительному расходу памяти.[1]

Атрибут LongTruncOk используется для определения действий, производимых в случае, когда выбранное значение поля оказывается больше по размеру, чем буфер, заданный атрибутом LongReadLen. Например, если LongTruncOk имеет значение «истина» (т. е. «усечение разрешено»), то сверхдлинное значение молчаливо усекается до длины, указанной в LongReadLen, без возбуждения ошибки.

С другой стороны, если LongTruncOk имеет значение «ложь», то выборка данных типа LONG с размером большим, чем LongReadLen, рассматривается как ошибка. Если RaiseError не включен, то происходит обычный сбой вызова выборки данных, который выглядит так, будто бы достигнут конец данных.

LongTruncOk по умолчанию устанавливается как «ложь», что приводит к ошибке при выполнении выборки чрезмерно длинных данных. Устанавливайте RaiseError или проверяйте наличие ошибок после выполнения циклов выборки.

Далее в этой главе мы подробнее обсудим, как можно работать с данными типа LONG.

В спецификации DBI, приведенной в приложении A, содержится полный список всех общих атрибутов, определенных в DBI.

Атрибуты дескриптора базы

Атрибуты дескриптора базы данных специфичны для дескрипторов баз данных и не определены для других дескрипторов. В их число входят:

[1] При использовании значения, являющегося степенью двух, например, 64 Кбайт, 512 Кбайт, 8 Мбайт и т. д., фактически в некоторых системах может быть занят вдвое больший объем из-за нерационального механизма выделения памяти. Это связано с тем, что для хранения служебной информации требуется несколько дополнительных байт, а поскольку глупая программа распределения памяти работает только со степенями двух, она удваивает объем выделяемой памяти для их размещения.

AutoCommit

Атрибут дескриптора базы данных `AutoCommit` используется для того, чтобы позволить программам осуществлять более тонкий контроль над транзакциями (в отличие от поведения по умолчанию, означающего «фиксировать все»).

Функционирование этого атрибута тесно связано с тем, как DBI осуществляет контроль над транзакциями. Поэтому далее в этой главе можно найти полное описание этого параметра.

Name

Атрибут дескриптора базы данных `Name` содержит «имя» базы данных. Обычно оно совпадает со строкой «`dbi:DriverName:...`», используемой для соединения с базой данных, но без ведущих символов «`dbi:DriverName:`».

В спецификации DBI, приведенной в приложении А, содержится полный список всех атрибутов дескриптора базы данных, определенных в DBI. Атрибуты дескрипторов команд мы обсудим чуть позже, а сначала коснемся метаданных базы данных.

Метаданные базы данных

Метаданные базы данных являются информацией высокого уровня, «данными о данных», хранимыми в базе данных и описывающими эту базу данных. Эти данные крайне полезны для динамического создания команд SQL и даже создания динамических представлений содержимого базы данных.

Хранимые в базе данных метаданные и способ их хранения весьма разнообразны в различных системах баз данных. В большинстве основных систем имеется *системный каталог*, состоящий из набора таблиц и представлений, запросы к которым позволяют получить сведения обо всех объектах базы данных, включая таблицы и представления. При осуществлении непосредственных запросов к системному каталогу часто возникают две проблемы: данные оказываются слишком сложными и трудными для запросов, и запросы являются непереносимыми на другие типы баз данных.

В DBI следовало бы включить ряд удобных методов для доступа к этим данным в переносимой форме, и когда-нибудь это произойдет, однако сейчас он предоставляет лишь два метода, которые можно выполнить для действующего дескриптора базы данных, чтобы извлечь из нее метаданные об объектах.

Первый из этих методов называется `tables()` и возвращает массив, содержащий имена таблиц и представлений в базе данных, соответствующей дескриптору. Использование этого метода показано в следующем примере:

```
### Соединиться с базой данных
my $dbh = DBI->connect( 'dbi:Oracle:archaeo', 'stones', 'stones' );

### Получить список таблиц и представлений
my @tables = $dbh->tables();

### Вывести его
foreach my $table ( @tables ) {
    print "Таблица: $table\n";
}
```

При соединении с базой данных MySQL будет получено следующее:

```
Таблица: megaliths
Таблица: media
Таблица: site_types
```

Однако при соединении с базой данных Oracle результат будет таким:

```
Таблица: STONES.MEGALITHS
Таблица: STONES.MEDIA
Таблица: STONES.SITE_TYPES
```

В том и другом случае, если в базе данных содержатся другие таблицы, они будут включены в выдачу.

Oracle по умолчанию записывает все имена в верхнем регистре, чем объясняется одно из отличий в выдаче, но что означает «`STONES.`» перед именем каждой таблицы?

Oracle, как и большинство других больших систем баз данных, поддерживает концепцию *схемы*. Схема представляет собой способ группировки связанных между собой таблиц и других объектов базы данных в именованную совокупность. В Oracle каждый пользователь получает свою собственную схему с именем, совпадающим с именем пользователя. (Такой подход применяется не во всех базах данных, поддерживающих схемы.)

Если бы другой пользователь Oracle, отличный от *stones*, захотел обратиться к таблице `media`, то по умолчанию ему потребовалось бы *полностью квалифицировать* имя таблицы, добавляя имя схемы, например `stones.media`. Если этого не делать, то база данных полагает, что ссылка указывает на таблицу `media` в собственной схеме пользователя.

Итак, `STONES` в выдаче является именем схемы, в которой определены таблицы. То, что возвращаются полностью квалифицированные имена, важно, поскольку метод `tables()` возвращает имена всех таблиц, которыми владеют все обнаруженные им пользователи.

Другим методом, используемым для извлечения метаданных базы данных, является `table_info()`, возвращающий более подробные сведения о таблицах и представлениях, хранимых в базе данных.

При вызове `table_info()` возвращается подготовленный и выполненный дескриптор команды, который можно использовать для выборки данных о таблицах и представлениях, находящихся в базе данных. Каждая строка, выбираемая через этот дескриптор команды, содержит *по меньшей мере* следующие поля в указанном порядке:[1]

TABLE_QUALIFIER

В этом поле содержится идентификатор квалификатора таблицы. В большинстве случаев возвращается `undef` (NULL).

TABLE_OWNER

В этом поле содержится имя владельца таблицы. Если база данных не поддерживает множественные схемы или владение таблицами, в этом поле содержится `undef` (NULL).

TABLE_NAME

В этом поле содержится имя таблицы, которое *никогда* не должно быть `undef`.

TABLE_TYPE

В этом поле содержится «тип» объекта, описываемого этой строкой. Возможными значениями могут быть `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, `SYNONYM` или специфичный для драйвера базы данных идентификатор.

REMARKS

В этом поле содержатся описание или комментарии, относящиеся к таблице. Поле может содержать `undef` (NULL).

Для того чтобы вывести некоторые основные сведения о таблицах, содержащихся в текущей схеме или базе данных, можно написать следующую программу, в которой используется `table_info()` для извлечения всей информации о таблицах, а затем форматируется выдача:

```
#!/usr/bin/perl -w
#
# ch06/dbhdump: Делает дамп сведений о команде SQL.

use DBI;

### Соединиться с базой данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1
} );

### Создать новый дескриптор команды для выборки данных о таблицах
my $tabsth = $dbh->table_info();
```

[1] Драйверы баз данных могут включать в результирующие данные дополнительные колонки сведений.

```
### Вывод заголовка
print "Квалификатор  Владелец  Название таблицы  Тип  Примечания\n";
print "============  ========  ================  ===  ==========\n\n";

### Перебор всех таблиц...
while ( my ( $qual, $owner, $name, $type, $remarks ) =
        $tabsth->fetchrow_array() ) {

    ### Привести в порядок поля NULL
    foreach ($qual, $owner, $name, $type, $remarks) {
        $_ = "N/A" unless defined $_;
    }

    ### Вывести метаданные таблицы...
    printf "%-9s  %-9s %-32s %-6s %s\n", $qual, $owner, $name, $type,
    $remarks;
}

exit;
```

Выполнение этой программы с нашей мегалитической базой данных в Oracle выводит следующий результат:

```
Квалификатор  Владелец  Название таблицы  Тип    Примечания
============  ========  ================  =====  ==========

N/A           STONES    MEDIA             TABLE  N/A
N/A           STONES    MEGALITHS         TABLE  N/A
N/A           STONES    SITE_TYPES        TABLE  N/A
```

Такой вид метаданных не слишком полезен, поскольку в нем перечислены только метаданные об объектах, находящихся в базе данных, а не структура самих объектов (например имена колонок). Чтобы извлечь структуру каждой таблицы или представления, нам нужен другой тип метаданных, который можно получить через дескрипторы команд.

Атрибуты дескриптора команды, или метаданные команды

Атрибуты дескриптора команды специфичны для дескрипторов команд и наследуют ряд атрибутов от родительских дескрипторов баз данных. Многие атрибуты дескрипторов команд определены только для чтения, поскольку лишь описывают подготовленную команду или ее результаты.

Теоретически эти атрибуты должны быть определены, когда дескриптор команды подготовлен, но практически можно рассчитывать на правильность их значений только тогда, когда дескриптор команды

подготовлен *и выполнен*. Кроме того, для некоторых драйверов выборка всех данных из команды SELECT или явный вызов метода finish() для дескриптора команды делают атрибуты дескриптора команды недоступными.

В спецификации DBI, приведенной в приложении А, содержится полный список всех атрибутов дескриптора команды, определенных в DBI.

Statement

Этот атрибут содержит строку команды, переданную методу prepare().

NUM_OF_FIELDS

В этом атрибуте содержится число колонок, которые будут возвращены командой SELECT. Например:

```
$sth = $dbh->prepare( "
            SELECT name, location, mapref
            FROM megaliths
          " );
$sth->execute();
print "Команда SQL содержит $sth->{NUM_OF_FIELDS} колонок\n";
```

Для команд, не являющихся SELECT, этот атрибут содержит нулевое значение. Это позволяет производить быструю проверку того, имеет ли команда тип SELECT.

NAME

NAME_uc

NAME_lc

В атрибуте NAME содержатся имена выбранных в команде колонок. Значением атрибута является ссылка на массив, длина которого равна количеству полей в исходной команде.

Например, можно так перечислить все имена колонок в таблице:

```
$sth = $dbh->prepare( "SELECT * FROM megaliths" );
$sth->execute();

for ( $i = 1 ; $i <= $sth->{NUM_OF_FIELDS} ; $i++ ) {
    print "Колонка $i называется $sth->{NAME}->[$i-1]\n";
}
```

Имена, содержащиеся в массиве атрибутов, являются именами колонок, возвращенными базой данных.

Есть два дополнительных атрибута, относящиеся к именам колонок. NAME_uc содержит те же имена колонок, что и атрибут NAME, но все символы нижнего регистра переведены в нем в верхний регистр. Аналогично в атрибуте NAME_lc все символы верхнего регистра пере-

ведены в нижний регистр. Обычно предпочтительнее использовать эти атрибуты, а не NAME.

TYPE

Атрибут TYPE содержит ссылку на массив целых величин, представляющих международный стандарт значений для соответствующих типов данных. Массив целых чисел имеет длину, равную числу колонок, выбранных исходной командой, и обращаться с ним можно так же, как в приведенном выше примере с атрибутом NAME.

Стандартными значениями распространенных типов данных являются следующие:

```
SQL_CHAR             1
SQL_NUMERIC          2
SQL_DECIMAL          3
SQL_INTEGER          4
SQL_SMALLINT         5
SQL_FLOAT            6
SQL_REAL             7
SQL_DOUBLE           8
SQL_DATE             9
SQL_TIME            10
SQL_TIMESTAMP       11
SQL_VARCHAR         12
SQL_LONGVARCHAR     -1
SQL_BINARY          -2
SQL_VARBINARY       -3
SQL_LONGVARBINARY   -4
SQL_BIGINT          -5
SQL_TINYINT         -6
SQL_BIT             -7
SQL_WCHAR           -8
SQL_WVARCHAR        -9
SQL_WLONGVARCHAR   -10
```

Хотя эти числа достаточно стандартизованы,[1] соответствие, устанавливаемое драйверами между собственными типами данных и этими стандартными значениями, может быть весьма различным. Собственные типы данных, плохо соответствующие какому-либо из этих типов, могут отображаться в диапазоне, официально зарезервированном для использования Perl DBI: от –9999 до –9000.

[1] Некоторые являются стандартом ISO, другие являются стандартом de facto Microsoft ODBC. Зайдите на *ftp://jerry.ece.umassd.edu/isowg3/dbl/SQL_Registry* и поищите «SQL Data Types» или имена интересующих вас типов на *http://search.microsoft.com/us/dev/*.

PRECISION

Атрибут PRECISION содержит ссылку на массив целых чисел, представляющих длину или размер колонок в команде SQL.

Есть два общих способа вычисления точности представления данных в колонке. Для строковых типов, таких как CHAR и VARCHAR, возвращается максимальная длина колонки. Например, для колонки, определенной в таблице как:

```
location        VARCHAR2(1000)
```

возвращается значение 1000.

Числовые типы данных обрабатываются несколько иным способом, при котором возвращается число *значащих цифр*. Оно может не иметь прямой связи с объемом пространства, отводимого для хранения числа. Например, в Oracle числа хранятся с точностью до 38 знаков, но используется внутренний формат с переменной длиной размером от 1 до 21 байта.

Для типов с плавающей запятой, таких как REAL, FLOAT и DOUBLE, максимальный «размер отображения» может быть на семь символов больше, чем точность, из-за символа присоединения, десятичной точки, буквы «E», знака и двух или трех цифр показателя степени.

SCALE

Атрибут SCALE содержит ссылку на массив целых чисел, представляющих количество десятичных разрядов в колонке. Очевидно, он имеет смысл только для чисел с плавающей точкой. Целые и нечисловые типы данных возвращают ноль.

NULLABLE

Атрибут NULLABLE содержит ссылку на массив целых чисел, указывающих на то, может ли соответствующая колонка иметь значение NULL. Элементы массива могут принимать одно из трех значений:

0 Колонка не может иметь значение NULL.

1 Колонка может иметь значение NULL.

2 Неизвестно, может ли колонка иметь значение NULL.

NUM_OF_PARAMS

Атрибут NUM_OF_PARAMS содержит число параметров (заполнителей), заданных в команде.

Обычное использование этих атрибутов дескриптора команды состоит в том, чтобы динамически форматировать и отображать данные, получаемые из запросов, а также получать информацию о таблицах, содержащихся в базе данных.

В следующем сценарии последняя операция осуществлена путем создания дескриптора команды, выбирающей информацию по всем таблицам, как было описано ранее в разделе «Метаданные базы данных», и последующего перебора всех таблиц и вывода структуры таблицы через метаданные команды:

```
#!/usr/bin/perl -w
#
# ch06/tabledump: Дамп сведений обо всех таблицах.

use DBI;

### Соединиться с базой данных
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    RaiseError => 1
});

### Создать новый дескриптор команды для выборки информации о таблице
my $tabsth = $dbh->table_info();

### Перебор всех таблиц...
while ( my ( $qual, $owner, $name, $type ) = $tabsth->fetchrow_array() ) {

    ### Таблица, данные о которой нужно выбрать
    my $table = $name;

    ### Создать полное имя таблицы, заключив при необходимости в кавычки
    $table = qq{"$owner"."$table"} if defined $owner;

    ### Команда SQL для выборки метаданных таблицы
    my $statement = "SELECT * FROM $table";

    print "\n";
    print "Информация о таблице\n";
    print "====================\n\n";
    print "Команда:     $statement\n";

    ### Подготовить и выполнить команду SQL
    my $sth = $dbh->prepare( $statement );
    $sth->execute();

    my $fields = $sth->{NUM_OF_FIELDS};
    print "КОЛИЧЕСТВО_ПОЛЕЙ: $fields\n\n";

    print "Название колонки  Тип  Точность  степень  Null возможен?\n";
    print "----------------  ---  --------  -------  --------------\n\n";

    ### Перебор всех полей и дамп сведений о каждом поле
    for ( my $i = 0 ; $i < $fields ; $i++ ) {

        my $name = $sth->{NAME}->[$i];
```

```
    ### Описать значение NULLABLE
    my $nullable = ("Да", "Нет", "Неизвестно")[ $sth->{NULLABLE}->[$i] ];
    ### Преобразовать другие значения, не предоставляемые некоторыми
    ### драйверами
    my $scale = $sth->{SCALE}->[$i];
    my $prec  = $sth->{PRECISION}->[$i];
    my $type  = $sth->{TYPE}->[$i];

    ### Вывести сведения о поле
    printf "%-30s %5d      %4d   %4d   %s\n",
           $name, $type, $prec, $scale, $nullable;
}

### Явным образом освободить ресурсы команды,
### поскольку мы не выбрали все данные
    $sth->finish();
}

exit;
```

При выполнении команды с нашей мегалитической базой данных выводятся следующие данные:

```
Информация о таблице
====================

Команда: SELECT * FROM STONES.MEDIA
КОЛИЧЕСТВО_ПОЛЕЙ: 5

Название колонки  Тип  Точность  Степень  Null возможен?
----------------  ---  --------  -------  --------------

ID                  3        38        0  Нет
MEGALITH_ID         3        38        0  Да
URL                12      1024        0  Да
CONTENT_TYPE       12        64        0  Да
DESCRIPTION        12      1024        0  Да

Информация о таблице
====================

Команда: SELECT * FROM STONES.MEGALITHS
КОЛИЧЕСТВО_ПОЛЕЙ: 6

Название колонки  Тип  Точность  Степень  Null возможен?
----------------  ---  --------  -------  --------------

ID                  3        38        0  Нет
NAME               12       512        0  Да
DESCRIPTION        12      2048        0  Да
LOCATION           12      2048        0  Да
MAPREF             12        16        0  Да
SITE_TYPE_ID        3        38        0  Да
```

```
Информация о таблице
====================

Команда: SELECT * FROM STONES.SITE_TYPES
КОЛИЧЕСТВО_ПОЛЕЙ: 3

Название колонки  Тип  Точность  Степень  Null возможен?
----------------  ---  --------  -------  --------------

ID                  3        38        0  Нет
SITE_TYPE          12       512        0  Да
DESCRIPTION        12      2048        0  Да
```

В выводимой информации представлены сведения о структуре объектов нашей базы данных. Того же результата мы могли бы добиться, опрашивая системные таблицы нашей базы данных. Мы получили бы при этом больше информации, но такая программа не была бы переносимой.

Обработка данных типа LONG/LOB

Чтобы извлекать из базы данные типа LONG/LOB (большие объекты), DBI требуется получить некоторую дополнительную информацию. Как уже говорилось ранее в разделе об атрибутах `LongReadLen` и `LongTruncLen`, DBI не может определить, какого размера буфер должен быть выделен для выборки колонок, содержащих данные LOB. Поэтому мы не можем просто подать команду `SELECT` и рассчитывать, что она выполнится.

Выбор данных LOB осуществляется просто и, в сущности, идентичен выбору колонок любых других типов, за тем важным исключением, что нужно установить, по крайней мере, атрибут `LongReadLen`, прежде чем готовить команду, которая должна возвратить LOB. Например:

```
### Мы не ждем двоичных данных размером свыше 512 Кбайт...
$dbh->{LongReadLen} = 512 * 1024;

### Выбор сырых данных мультимедиа из базы данных
$sth = $dbh->prepare( "
            SELECT mega.name, med.media_data
             FROM megaliths mega, media med
             WHERE mega.id = med.megaliths_id
       " );
$sth->execute();
while ( ($name, $data) = $sth->fetchrow_array ) {
    ...
}
```

Без крайне важной установки LongReadLen вызов fetchrow_array(), по всей вероятности, привел бы к неудаче во время выборки уже первой строки, поскольку значение LongReadLen по умолчанию очень мало – обычно 80 или меньше.

Что будет, если в базе данных окажется «норовистая» колонка, которая больше, чем LongReadLen? Как справится с ней код в предыдущем примере? Что произойдет?

Когда размер данных выбранного LOB превосходит значение LongReadLen, возникает ошибка, *если* атрибут LongTruncOk *не* установлен в значение «истина». По умолчанию DBI присваивает LongTruncOk значение «ложь», чтобы гарантировать ошибку при случайном усечении данных.

Однако при этом может возникнуть проблема, если атрибут RaiseError не установлен. Как поведет себя приведенный выше фрагмент программы, если он попытается выбрать строку с полем LOB, размер которого превышает значение LongReadLen? Метод fetchrow_array() возвращает пустой список, если во время выборки строки возникла ошибка. Но fetchrow_array() возвращает пустой список и в том случае, когда выбраны все данные. Цикл while просто завершит свою работу, и будет выполнен код, который следует за ним. Если в цикле должны были быть выбраны 50 записей, то его выполнение могло прекратиться после выборки 45-й записи, если 46-я оказалась слишком большой. Без проверки ошибок вы можете никогда не обнаружить, что некоторые строки отсутствуют! То же самое справедливо для других методов fetchrow, например fetchrow_hashref().

Не все помнят, что после циклов выборки нужно проверять наличие ошибок, и это является частой причиной проблем, возникающих с программами, обрабатывающими поля LONG/LOB. *Всегда*, даже тогда, когда не обрабатываются поля специальных типов, полезно проверить наличие ошибок после циклов выборки или позволить DBI сделать это за вас путем установки атрибута RaiseError, как обсуждалось в главе 4.

Возвращаясь к нашему маленькому программному фрагменту, допустим, что нас удовлетворит, если значения, большие чем LongReadLen, будут молча усекаться, не вызывая ошибку. Следующая программная заглушка правильно обработает этот случай:

```
### Нас интересуют первые 512 Кбайт данных
$dbh->{LongReadLen} = 512 * 1024;
$dbh->{LongTruncOk} = 1;    ### Нас удовлетворяет усечение излишков

### Выбираем из базы сырые мультимедиаданные
$sth = $dbh->prepare( ¨
            SELECT mega.name, med.media_data
            FROM megaliths mega, media med
            WHERE mega.id = med.megaliths_id
```

```
            " );
$sth->execute();
while ( ($name, $data) = $sth->fetchrow_arrayref ) {
    ...
}
```

Единственное изменение, которое мы произвели, не считая комментариев, это добавление строки, устанавливающей атрибут `LongTruncOk` в истинное значение.

Возможность усечения данных LOB, которые слишком велики, весьма полезна для текста и некоторых видов двоичных данных, но не для всех. При хранении потокового мультимедиа, интерпретируемого на временной основе, вы не сильно пострадаете от усечения, поскольку поток данных можно будет просматривать или прослушивать до той точки, в которой он был обрезан. Однако такие двоичные файлы, как ZIP, в конце которых хранится контрольная сумма, станут в результате усечения нерабочими. Для данных такого типа не рекомендуется включать атрибут `LongTruncOk`, поскольку он позволяет обрезать данные и делать их в результате непригодными без всякого сообщения о наличии проблемы. В такой ситуации вы не сможете определить, содержит ли колонка искаженные данные или она была усечена DBI. Caveat emptor – пусть покупатель будет осторожен!

Для того чтобы писать переносимые программы, извлекающие данные LOB из базы, нужно помнить, что формат этих данных может быть различным в разных базах данных и для разных типов в одной базе данных. Например, в Oracle для колонки с типом данных LONG RAW, а не просто LONG, данные переносятся закодированными как пары шестнадцатеричных цифр для каждого байта, поэтому после выборки очередной шестнадцатеричной строки требуется декодировать ее с помощью `unpack("H*",...)`, чтобы получить исходное двоичное значение. В силу исторических причин атрибут `LongReadLen` для этих типов данных относится к длине двоичных данных, поэтому можно производить выборку шестнадцатеричных строк вдвое большей длины, чем указанная атрибутом длина.

В настоящее время в DBI нет способов выборки значений LONG/LOB *по частям*, т. е. кусок за куском. Это значит, что вы ограничены выборкой только таких данных, которые умещаются в доступной памяти. Кроме того, вы не можете организовать *поток* данных во время выборки их из базы. В некоторых драйверах реализован неофициальный метод `blob_read()`, поэтому если вам необходима выборка данных по кускам, проверьте ее возможность по документации к конкретному драйверу.

Вставка и обновление колонок LONG/LOB

Некоторые базы данных позволяют вставлять данные в колонки LONG/LOB, используя команды SQL, с помощью литеральных строк, например:

```
INSERT INTO table_name (key_num, long_description) VALUES (42, '...')
```

Если забыть о переносимости, то это хорошо для коротких текстовых строк, но для любых других данных сразу возникают проблемы. Во-первых, в большинстве баз данных существует предел максимальной длины команды SQL, который обычно значительно ниже, чем максимальный размер колонки LONG/LOB. Во-вторых, в большинстве баз данных существуют ограничения на то, какие символы могут включаться в литералы. Метод `quote()` драйвера DBD сделает все, что в его силах, но часто невозможно включить в строку произвольные двоичные значения. Наконец, возвращаясь к переносимости, многие базы данных строго относятся к типизации данных и просто не позволяют присваивать литеральные строки колонкам LONG/LOB.

Как же избежать этих проблем? Здесь снова на помощь нам приходят *заполнители*. Мы достаточно подробно обсуждали заполнители в главе 5, поэтому здесь мы скажем только о том, что относится к полям LONG/LOB.

Чтобы использовать заполнители, следует реализовать с помощью DBI вышеприведенную команду следующим образом:

```
use DBI qw(:sql_types);

$sth = $dbh->prepare( "
    INSERT INTO table_name (key_num, long_description) VALUES (?, ?)
" );
$sth->bind_param( 1, 42 );
$sth->bind_param( 2, $long_description, SQL_LONGVARCHAR);
$sth->execute();
```

Передача `SQL_LONGVARCHAR` в качестве необязательного параметра `TYPE` методу `bind_param()` дает драйверу серьезное указание на то, что вы осуществляете связывание с типом LONG/LOB. Для некоторых драйверов такое указание не является необходимым, но всегда полезно включить его.

В настоящее время DBI не предоставляет способа вставки или обновления полей LONG/LOB *по частям*, т. е. кусок за куском. Это означает, что при обработке таких величин вы ограничены доступной вам памятью.

Транзакции, блокировка и изоляция

Последняя тема в этой главе относится к важной (волосы дыбом встают!) проблеме *обработки транзакций*.

Обработка транзакций поддерживается достаточно мощными системами баз данных, в которых команды SQL могут быть объединены в логические группы. Каждая такая группа называется *транзакцией*, а осуществляемые ею действия гарантированно являются атомарной операцией при восстановлении исходного состояния. Согласно стандарту ANSI/ISO SQL транзакция начинается с первой выполнимой команды SQL и заканчивается при явной фиксации или откате.

В результате фиксации данные записываются в таблицы базы данных и становятся видимыми для других пользователей, работающих в то же время. В результате отката аннулируются все изменения, произведенные в любых таблицах с начала текущей транзакции.

Стандартным примером, разъясняющим действие транзакции, является банковский перевод, при котором клиент переводит $1000 с одного банковского счета на другой. Банковский перевод состоит из трех отдельных этапов:

1. Уменьшение исходного счета на требуемую сумму.
2. Увеличение целевого счета на требуемую сумму.
3. Регистрация перевода в журнале.

Если взглянуть на перевод как на три отдельных этапа, то возможность катастрофы становится вполне очевидной. Предположим, что между этапами 1 и 2 выключается электричество. Несчастный клиент обеднел на $1000, поскольку деньги не достигли целевого счета и не зарегистрированы в журнале переводов. Банк стал на $1000 богаче.[1]

Разумеется, если авария питания произошла между этапами 2 и 3, то у клиента образуются нужные суммы на нужных счетах, однако у банка не сохраняется записей об операции. Это ведет к возникновению различных бухгалтерских проблем.

Выход состоит в том, чтобы рассматривать три отдельных этапа как одну логическую единицу, или транзакцию. Так, в начале выполнения первого этапа автоматически запускается транзакция. Эта транзакция продолжается, пока не будет завершен этап 3, после чего транзакцию можно завершить, а все изменения либо зафиксировать в базе данных, либо сделать откат и аннулировать их. Поэтому если в какой-либо момент во время выполнения транзакции произойдет отключение питания, для всей транзакции может автоматически быть произведен откат, когда база данных будет снова запущена, и в данных не будет произведено никаких постоянных изменений.

[1] Не удивительно, что деньги сначала снимаются со счета.

Транзакция представляет собой ситуацию «все или ничего». Либо срабатывает все, либо ничего, что представляет собой приятную новость для нашего несчастного клиента банка.

Иногда говорят, что транзакции обладают свойствами A.C.I.D. (Atomic, Consistent, Isolated, Durable – атомарны, непротиворечивы, изолированны, постоянны).

Атомарность

Изменения, производимые транзакцией в базе данных, являются атомарными: либо происходит все, либо не происходит ничего.

Непротиворечивость

Транзакция является корректным преобразованием состояния. Действия, рассматриваемые как группа, не нарушают каких-либо требований целостности, связанных с состоянием.

Изолированность

Несмотря на то что транзакции могут выполняться одновременно, каждой из них представляется, что остальные транзакции выполнены до нее или после нее.

Постоянство

После того как транзакция успешно завершена (т. е. `commit()` возвращает значение, соответствующее успеху), произведенные ею в базе данных изменения сохраняются после возникающих в будущем отказов.

Реализация базой данных обработки транзакций со свойствами ACID требует использования файла с журналом регистрации, некоторых сложных технических приемов, а также тщательного программирования. Поэтому среди свободно распространяемых баз данных редко можно найти такие, которые поддерживают транзакции со свойствами ACID (примечательное исключение составляет PostgreSQL), а в случае поддержки приходится расплачиваться снижением производительности.

С другой стороны, полная поддержка транзакций обеспечивает значительно большую безопасность при отказах питания, отказах клиентов, отказах баз данных и других распространенных видах аварий. Как мы увидим ниже, простые механизмы блокировки не обеспечивают того же уровня защиты и восстанавливаемости.

Поскольку не все системы баз данных поддерживают обработку транзакций, не исключено, что вам не удастся воспользоваться возможностью отката при непреднамеренном повреждении данных или защитить себя от возможных аварий электропитания. Но если ваша база данных поддерживает транзакции, то DBI позволит легко управлять ими, создавая при этом переносимый код.

Автоматическая обработка транзакций

Стандартом ISO для SQL определена точная модель транзакции. В нем сказано, что соединение с базой данных *всегда* происходит в рамках транзакции. Каждая транзакция заканчивается фиксацией или откатом, а каждая новая транзакция начинается с очередной исполняемой команды. В большинстве систем определен также механизм автоматической фиксации, который действует так, как если бы `commit()` автоматически вызывался после каждой команды.

В стандарте DBI делается попытка найти способ позволить всем драйверам для всех баз данных предоставить, насколько это осуществимо, одинаковые возможности. При этом он опирается на то, что на практике существует слабая разница между базой данных, поддерживающей транзакции, но работающей в режиме автоматической фиксации, и базой данных, которая вообще не поддерживает транзакции.

Стандарт DBI пытается также гарантировать, чтобы приложение, написанное с использованием транзакций, не могло быть случайно выполнено с базой данных, которая их не поддерживает. Он делает это, рассматривая попытку отключить автоматическую фиксацию транзакций как фатальную ошибку для такой базы данных.

Исходя из важности возможности включения и отключения автоматической фиксации транзакций, DBI определяет атрибут дескриптора базы данных с именем `AutoCommit`, который управляет тем, должен ли DBI *делать вид*, что осуществляет автоматическую принудительную фиксацию изменения данных после каждой команды.

Например, если выполняется такая команда, как `$dbh->do()`, в результате действия которой удаляются какие-либо данные из базы, и `AutoCommit` имеет значение «истина», то откатить изменения нельзя, даже если база данных поддерживает транзакции.

По умолчанию DBI устанавливает `AutoCommit` во включенное состояние, в результате чего такое потенциально опасное поведение становится автоматическим, если не отключить его явным образом. Это обусловлено прецедентом, установленным ODBC и JDBC. Возможно, в этом случае было ошибкой ставить совместимость DBI со стандартами выше, чем соображения безопасности. В будущей версии DBI, возможно, будет выводиться предупреждение, если `AutoCommit` не указывается в качестве атрибута для метода `DBI->connect()`, поэтому стоит уже сейчас привыкать к его использованию.

Действия при изменении этого атрибута зависят от того, какой тип обработки транзакций поддерживает база данных. Существуют три возможности:

Транзакции не поддерживаются

Для баз данных, не имеющих поддержки транзакций, AutoCommit всегда включен. Попытка отключить AutoCommit приведет к фатальной ошибке.

Транзакции поддерживаются постоянно

В эту группу баз данных входит основная масса коммерческих РСУБД, таких как Oracle, поддерживающая стандарт ANSI/ISO на проведение транзакций.

Если значение AutoCommit изменяется с включенного на выключенное, то никаких немедленных действий не происходит. Каждая новая команда становится частью новой транзакции, для которой нужно произвести фиксацию или откат.

Если значение AutoCommit изменяется с выключенного на включенное, то все незавершенные изменения в базе данных автоматически фиксируются.

Явная поддержка транзакций

В некоторых базах данных, например Informix, поддерживается идея, что транзакции являются необязательными и должны явным образом запускаться приложениями по мере необходимости.

DBI пытается работать с такими системами, как с базами данных, для которых транзакции активны постоянно. Для этого DBI требует, чтобы драйвер автоматически начинал транзакцию, когда AutoCommit переводится из включенного в выключенное состояние. Когда транзакция фиксируется или откатывается, драйвер автоматически начинает новую транзакцию.

Следовательно, несмотря на свою независимость от баз данных, DBI предлагает как простую автоматическую фиксацию транзакций, так и мощное ручное управление обработкой транзакций.

Принудительная фиксация

В DBI определен метод с именем commit() для явной фиксации всех незафиксированных данных в текущей транзакции. Этот метод выполняется применительно к действующему дескриптору базы данных:

```
$dbh->commit();
```

Если commit() вызывается, когда включен атрибут AutoCommit, выводится предупреждение следующего вида:

```
commit ineffective with AutoCommit
```

которое просто информирует о том, что изменения в базе данных уже зафиксированы. Это предупреждение выводится и тогда, когда commit() вызывается применительно к базе данных, которая не поддержи-

вает транзакции, поскольку в этом случае атрибут AutoCommit по определению включен.

Откат изменений

Операцией, дополнительной к фиксации данных в базе, является откат. В DBI определен метод с именем rollback(), который можно использовать для отката последних незафиксированных изменений в базе данных.

Как и commit(), метод rollback() выполняется применительно к дескриптору базы данных:

```
$dbh->rollback();
```

Аналогично, если rollback() вызывается, когда AutoCommit включен, выводится сообщение следующего типа:

```
rollback ineffective with AutoCommit
```

указывающее на то, что изменения в базе данных уже зафиксированы. Это предупреждение выводится и тогда, когда rollback() вызывается применительно к базе данных, которая не поддерживает транзакции, поскольку в этом случае атрибут AutoCommit по определению включен.

Отсоединение явное или случайное

Действие транзакции при явном отсоединении от базы данных, когда AutoCommit выключен, к сожалению, является неопределенным. Некоторые базы данных, такие как Oracle и Ingres, автоматически фиксируют все незавершенные изменения. Однако в других системах баз данных, таких как Informix, производится откат незавершенных изменений. Поэтому приложения, не использующие AutoCommit, должны *всегда* явным образом вызывать commit() или rollback(), перед тем как вызвать disconnect().

Что же происходит, если disconnect() не удается вызвать явным образом или такой возможности не представилось, т. к. программа завершила работу после die? Поскольку дескрипторы DBI являются ссылками на объекты, можно быть уверенным, что Perl сам вызовет метод DESTROY для каждого дескриптора, и если программа заканчивает работу, дескриптор выходит из области видимости или единственный экземпляр дескриптора перезаписывается другим значением.

Фактическая реализация метода DESTROY зависит от автора драйвера. Если дескриптор базы данных все еще сохраняет соединение, он *должен* автоматически вызвать rollback() (если только не включен AutoCommit) перед вызовом disconnect(). Вызов rollback() в DESTROY является важным. Если драйвер этого не делает, то программа, аварийно завер-

шаемая методом `die` посреди транзакции, может на деле «случайно» зафиксировать незавершенную транзакцию! К счастью, все известные нам драйверы, поддерживающие транзакции, действуют правильно.

В качестве дополнительной проверки при отсоединении от базы данных при наличии активных дескрипторов команд выводится предупредительное сообщение. Активные дескрипторы команд и связанные с ними вопросы мы обсуждали в главе 5.

Соединение автоматической обработки ошибок с транзакциями

Транзакции, как вы сейчас, наверное, осознали, тесно связаны с обработкой ошибок. Это особенно верно, когда после обнаружения ошибки приходится восстанавливать порядок, возвращая базу данных в то состояние, в котором она была перед началом транзакции.

В главе 4 мы достаточно подробно обсуждали обработку ошибок и воспевали хвалу использованию атрибута `RaiseError` для автоматического обнаружения ошибок.

Представьте себе, что можно соединить автоматическое *обнаружение* ошибок при установке атрибута DBI `RaiseError` с *перехватом* ошибок в блоке Perl `eval { ... }` и возможностями *обработки* ошибок в транзакциях. В результате получается простой, но мощный способ написания устойчивых приложений на Perl.

Существует довольно распространенная структура такого рода приложений, поэтому в помощь изложению этой темы мы включили соответствующий пример.

В этом наброске программы обрабатываются файлы CSV, содержащие данные по продажам в какой-либо стране, с некоторого веб-сайта загружается курс обмена валют и добавляется к данным, затем осуществляется ряд вставок, выборок, обновлений и еще ряд вставок, изменяющих базу данных. Обработка повторяется для нескольких стран.

Вот этот код:

```
### Соединиться с базой данных, поддерживающей транзакции
### и обработку ошибок
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , {
    AutoCommit => 0,
    RaiseError => 1,
} );

### Вести счет отказов. Используется для установки статуса выхода
### из программы
my @failed;
```

```
foreach my $country_code ( qw(US CA GB IE FR) ) {

    print "Обрабатывается $country_code\n";

    ### Все действия для одной страны производятся в блоке eval
    eval {

        ### Прочесть, разобрать и проверить файл данных (например,
        ### используя DBD::CSV)
        my $data = load_sales_data_file( "$country_file.csv" );

        ### Добавить данные из Web (например, с помощью модулей LWP)
        add_exchange_rates( $data, $country_code,
                            "http://exchange-rate-service.com" );

        ### Выполнить загрузку базы данных (например, с помощью
        ### DBD::Oracle)
        insert_sales_data( $dbh, $data );
        update_country_summary_data( $dbh, $data );
        insert_processed_files( $dbh, $country_code );

        ### С этим файлом все в порядке, зафиксировать изменения
        ### в базе данных
        $dbh->commit();

    };

    ### Если что-то не так...
    if ($@) {

        ### Сообщить пользователю, что есть ошибки, и какие ошибки
        warn "Невозможно обработать страну $country_code: $@\n";
        ### Отменить изменения в базе данных, произведенные
        ### до возникновения ошибки
        $dbh->rollback();

        ### Вести учет ошибок
        push @failed, $country_code;

    }
}
$dbh->disconnect();

### Выйти, сообщив пользователю статус
exit @failed ? 1 : 0;
```

Ниже мы приведем некоторые замечания относительно того, чем обусловлена такая структура программы, и обсудим некоторые связанные с этим вопросы:

Единица работы

Главным вопросом проектирования является принятие решения относительно того, что должна представлять собой «единица работы». Иными словами, после какой части работы нужно осуществлять фиксацию, и какая часть работы аннулируется при откате в случае ошибки? Наименьшая единица должна соответствовать наименьшей логически завершенной группе изменений в базе данных. В нашем примере это соответствует полной обработке файла для одной страны, что мы и выбрали здесь в качестве единицы работы.

Мы могли выбрать и более крупный кусок работы. При другом очевидном возможном выборе в качестве единицы работы принимается обработка всех файлов. В этом случае нам просто потребовалось бы поместить цикл `foreach` внутрь блока `eval`. Следует помнить, что в большинстве баз данных существует предел объема изменений в базе данных, которые можно произвести, не фиксируя их. Обычно он велик и может конфигурироваться, но нужно знать, что он ограничен и при его превышении могут возникнуть проблемы.

Где делать фиксацию

Важно поместить `commit()` внутрь блока `eval`. Вызов фиксации является наиболее важной частью транзакции. Не рассчитывайте на то, что `commit()` успешно выполнится только потому, что предшествующие команды выполнились без ошибок. Базы данных могут откладывать большую часть фактических действий до того момента, когда осуществляется фиксация.

Вызов `commit()` должен находиться в самом конце блока `eval`. Иногда приходится применять более сложные уловки. Представьте себе, что задача изменилась, и вам предложено удалять в своем сценарии файлы после их обработки. Где бы вы поместили вызов `unlink()`? До или после `commit()`? Поразмышляйте об этом некоторое время. Вспомните, что всегда существует опасность того, что либо `commit()`, либо `unlink()` не выполнятся. Необходимо оценить риск в каждом из этих случаев и возможные последствия.

Вот что может происходить в нашем примере. Если вы сначала осуществляете фиксацию, а затем удаление файла, и удаление не выполняется, то при следующем запуске сценария вы снова обрабатываете тот же самый файл. Если сначала вы удаляете файл, а затем происходит отказ фиксации, теряются данные. Очевидно, меньшим злом будет сначала выполнить фиксацию, в результате чего может произойти повторная обработка, которой, к тому же, можно попытаться избежать, сравнивая данные в файле с теми, которые уже находятся в базе данных.

Обстоятельства реальной жизни могут быть значительно более сложными. Однако есть множество творческих подходов к этой

проблеме *фиксации в двух системах*. Самое главное – помнить, что когда некоторые изменения вне базы данных должны фиксироваться одновременно с изменениями в базе данных, возникает проблема, которую нужно решать.

Действия при возникновении ошибок

Первое, что нужно сделать в блоке `if ($@) {...}`, – это вывести сообщение об ошибке. Код, выводящий ошибку, документирует то, с чем столкнулся блок обработки ошибок. Сделав это сразу, вы избежите того, что прежде чем вы выведете сообщение, произойдет другая фатальная ошибка, которая скроет первоначальную проблему.

Окажите любезность себе самому и вашим пользователям, *всегда* включая в свое сообщение об ошибке как можно больше полезных данных. Кажется, это просто, но вновь и вновь мы сталкиваемся с кодом типа:

```
$dbh->commit() or die "commit не сработал!"; # БЕССМЫСЛЕННО!
```

Использование `RaiseError` оказывается здесь полезным, поскольку создается сообщение (или присваивается значение переменной Perl `$@`), в котором содержится ошибка, сообщенная драйвером, а также имена драйвера и метода.[1]

Итак, если вы перехватываете ошибку с использованием `eval`, не забывайте вывести содержимое `$@`. Но не ограничивайтесь этим. В большинстве приложений имеются переменные, указывающие на то, какие данные в текущий момент обрабатываются. Включение одной или нескольких из них, в нашем случае `$country_code`, добавляет ценный контекст к сообщению об ошибке.

По общему правилу в каждый `die` или `warn` нужно интерполировать, по крайней мере, одну переменную, не считая `$!` или `$@`.

Защита отката транзакции

Итак, возникла ошибка, вы напечатали соответствующее сообщение, и теперь пора выполнять `rollback()` – откат. Остановитесь на мгновение. Вспомните, что у используемого вами дескриптора базы данных установлен атрибут `RaiseError`. Это означает, что если сама операция отката вызовет ошибку, будет возбуждена новая исключительная ситуация, и выполнение сценария сразу прекратится либо перейдет в охватывающий блок `eval`.

Задайте себе вопрос, будете ли вы удовлетворены, если ошибка отката запустит новую исключительную ситуацию. Если нет, то заключите ее в блок `eval`:

[1] В него входят также имя файла и номер строки программы, в которой произошла ошибка, но при желании их можно удалить с помощью регулярных выражений.

```
eval { $dbh->rollback };
```

Вероятными причинами отказа `rollback()` могут быть остановка сервера базы данных или ошибка связи в сети, каждая из которых может оказаться причиной той ошибки, которую вы в данный момент обрабатываете. Неуспех при выполнении отката обычно означает, что у сервера баз данных возникли проблемы и продолжать попытки не стоит, поэтому поведение по умолчанию часто является тем, что требуется.

Статус завершения программы

Возврат сценариями надежного статуса успех/ошибка при выходе служит признаком хорошего проектирования. Мы пропагандируем хорошее проектирование.

Смерть в результате несчастного случая

Одна из важных вещей, которые нужно помнить о транзакциях, состоит в том, что *любой* вызов внутри блока `eval` может послужить причиной аварийного выхода из программы, и эти вызовы могут вовсе не относиться к DBI. Поэтому, используя `eval` для перехвата *всех* возникающих ошибок, вы обеспечиваете корректный откат всех незавершенных транзакций.

7

ODBC и DBI

Open Database Connectivity – интерфейс открытого доступа к базам данных, широко известный как ODBC, определяет независимый от баз данных API. Администратор драйверов ODBC поддерживает API и управляет одним или более подключаемыми драйверами для обмена данными с различными типами баз данных. Соответствующая архитектура показана на рис. 7.1.

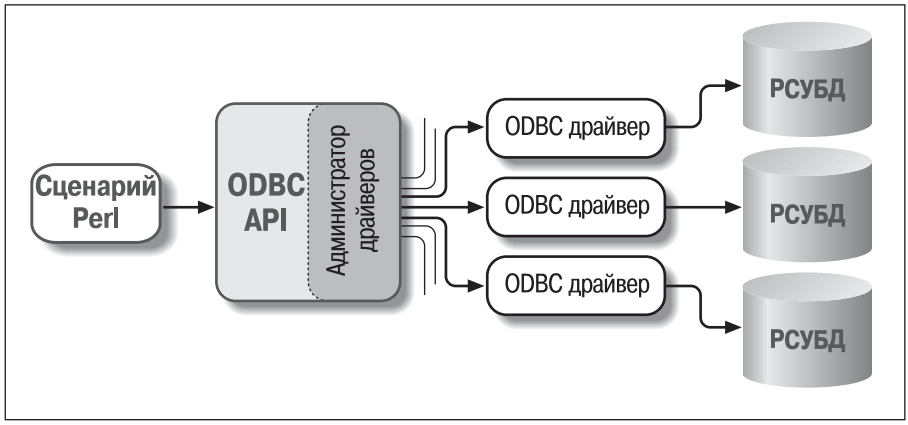

Рис. 7.1. *Архитектура ODBC*

Не звучит ли это знакомо? Менеджер драйверов ODBC и драйверы выполняют ту же работу, что и DBI со своими драйверами: определяют и реализуют независимый от баз данных интерфейс прикладного программирования.

В результате возникает целый ряд вопросов. В чем состоит различие? Почему не использовать просто ODBC, а не DBI? Могут ли DBI и ODBC

работать совместно? Какие преимущества имеет каждый из этих интерфейсов?

Прежде чем отвечать на эти вопросы, взглянем в перспективе на историю и задачи ODBC и DBI, а также посмотрим на модуль `Win32::ODBC`, реализующий непосредственный интерфейс к ODBC для Perl.

ODBC – результат принятия и расширения

В начале 1990-х годов консорциум поставщиков образовал SQL Access Group для поддержки возможности работы SQL в различных системах. В октябре 1992 и октябре 1993 годов большая часть этой работы была опубликована в виде проекта стандарта под названием «Интерфейс уровня вызовов» (Call Level Interface), или CLI, что является другим наименованием интерфейса прикладного программирования, или API. Однако стандарт CLI, разработанный SQL Access Group, на практике никогда не был реализован, по крайней мере, в тогдашней его форме.

Microsoft требовалось реализовать аналогичную концепцию, чтобы избежать необходимости выпуска различных версий программных продуктов, которым требовалось обмениваться данными с несколькими базами данных. Они увидели стандарт CLI SQL Access Group и «приняли и расширили» его, причем радикально. В результате появился интерфейс Open Database Connectivity, быстро ставший стандартом de facto. По справедливости Microsoft превратила незаконченный проект стандарта в полностью функциональную практическую реальность.

DBI – проработан и изменен

Тем временем, с 29 сентября 1992 года, еще до появления ODBC, группа заинтересованных участников работала над проработкой спецификации независимого от баз данных интерфейса для Perl 4. После примерно восемнадцати месяцев работы DBperl (как он тогда назывался) вполне устоялся, и должна была начаться его реализация.[1] Однако в это время Ларри Уолл начал выпускать альфа-версии Perl 5.

Вскоре стало очевидно, что объектно-ориентированные возможности Perl 5 можно использовать для реализации значительно улучшенного интерфейса баз данных. Поэтому работа над DBI приняла новое направление, и одновременно он стал свободно создаваться как надмно-

[1] Археологи могут найти спецификацию этой очень старой версии на *http://www.perl.com/CPAN/modules/dbperl/DBI dbispec.v05*.

жество стандарта CLI SQL Access Group. Поэтому на высоком уровне у DBI много общего как с этим стандартом, так и с ODBC, каким он нам известен сегодня.

Механизм ODBC

Рассмотрим основные характеристики ODBC, делающие его отличным от DBI и позволяющие успешно работать в качестве интерфейса, независимого от баз данных. Четырьмя главными характеристиками являются:

- стандартный синтаксис SQL;
- стандартные коды ошибок;
- богатые метаданные;
- многочисленные атрибуты и опции.

Стандартный синтаксис SQL

Стандартизация синтаксиса SQL является чем-то вроде чаши Грааля. В драйверах ODBC она обычно хорошо реализована, в то время как DBI полностью уклоняется от этой задачи! Проблема в том, что хотя в теории SQL может быть стандартизованным языком, на практике в базах данных большинства поставщиков он достаточно далек от стандарта, и это вызывает сложности при переносе.

Например, даже простая задача конкатенации двух полей базы данных должна записываться по следующему образцу (в базах данных, удовлетворяющих стандарту SQL-92):

```
SELECT first_name || ' ' || last_name FROM table
```

Разные базы данных требуют употребления одной из следующих форм:

```
SELECT first_name + ' ' + last_name FROM table
SELECT CONCAT(first_name, ' ', last_name) FROM table
SELECT CONCAT(CONCAT(first_name, ' ') last_name) FROM table
SELECT first_name CONCAT  ' ' CONCAT last_name FROM table
```

Диалекты SQL, используемые различными базами данных, изобилуют такими несовместимостями, не говоря уже о бесчисленных «расширениях» стандарта. Это основная головная боль для разработчиков, стремящихся к тому, чтобы их приложения работали с любыми базами данных.

Подход ODBC к этой проблеме весьма элегантен. Он позволяет осуществлять переносимость при использовании стандартного SQL, но не мешает иметь доступ к функциям, специфическим для каждой базы

данных. Когда приложение передает драйверу команду SQL, тот осуществляет ее синтаксический разбор как команды SQL-92, а затем перерабатывает в соответствии с фактическим синтаксисом используемой базы данных.

Если синтаксический разбор команды оказывается неудачным из-за несоответствия стандарту, то исходный SQL передается базе данных без изменений. Благодаря этому осуществляется доступ к специфическим для базы данных функциям, чему не мешает разбор, производимый ODBC.

DBI уклонился от этой задачи полностью, поскольку для ее решения потребовалось бы писать значительно более сложные драйверы, чем те, которыми он сейчас обходится. Синтаксический разбор и переформирование SQL является нетривиальной задачей, поэтому DBI не пытается предлагать переносимость на уровне SQL. В результате не возникает больших проблем. Perl позволяет легко создавать в приложениях такие команды SQL, которые требуются для используемой базы данных, как было показано в главе 5.

Стандартные коды ошибок

Если команда `INSERT` заканчивается неудачей, как можно определить, не вызвано ли это тем, что в таблице уже есть запись с тем же первичным ключом? В ODBC нужно проверить значение индикатора ошибки `SQLSTATE` и сравнить его с «23 000», независимо от того, какая база данных используется. В DBI вы предоставлены самому себе.

В ODBC определено большое число стандартных кодов ошибок, которые можно использовать для определения того, что произошло, с достаточной степенью подробности. Потребность в них возникает не слишком часто, но их наличие очень полезно. Эта идиллическая картина портится тем обстоятельством, что многие коды зависят от версии используемого драйвера ODBC. Например, драйвер ODBC 2.x возвращает «S0011», если в команде `CREATE INDEX` указано имя уже существующего индекса, а драйвер ODBC 3.x возвращает в этом случае «42S11». Вот такая стандартизация!

В DBI вам приходится проверять различные значения `$DBI::err` или строки `$DBI::errstr`, в зависимости от используемого драйвера базы данных. DBI предоставляет переменную `$DBI::state` и метод `$h->state()`, которые драйверы могут использовать для вывода стандартных кодов ошибок, но немногие из них делают это в настоящее время.

Богатые метаданные

В ODBC определен большой набор функций метаданных, предоставляющих сведения как о структуре данных в базе, так и о типах данных,

поддерживаемых базой данных. В следующей таблице перечислены эти функции и отмечено, какие из них поддерживаются в модулях `Win32::ODBC` и DBI.

Функция ODBC	Win32::ODBC	DBI
Tables	x	x
TablePrivileges		
Columns	x	
ColumnPrivileges		
SpecialColumns	x	
Statistics	x	
PrimaryKeys	x	
ForeignKeys	x	
Procedures		
ProcedureColumns		
GetTypeInfo	x	x

Как видите, DBI отстает от модуля `Win32::ODBC`. К тому времени, когда вы будете читать эту книгу, в DBI могут быть определены интерфейсы для некоторых функций, но сколько времени потребуется на то, чтобы драйверы действительно их реализовали, неизвестно. Возможно, лидерами станут модули `DBD::ODBC` и `DBD::Oracle`.

Многочисленные атрибуты и опции

В попытке стать всеобъемлющим интерфейсом для очень широкого спектра существующих источников данных ODBC предоставляет способ сообщить приложению о мельчайших деталях драйвера и источника данных, с которым он соединен. Через функцию `GetInfo()` можно получить столько подробностей (свыше 200 по последним подсчетам), что мы не станем расходовать бумагу, перечисляя их.

Хотя некоторые книги включают этот список, чтобы добиться внушительного объема, мы просто адресуем вас к соответствующей электронной версии Microsoft:

http://msdn.microsoft.com/library/sdkdoc/dasdk/odch5fu7.htm

Если этот URL прекратил существование, можно обратиться к средствам поиска MSDN на

http://msdn.microsoft.com/us/dev/

и поискать SQLGetInfo returns, используя параметр поиска exact phrase. Ссылка, которую вы найдете, вероятно, будет называться просто SQLGetInfo.[1]

ODBC предоставляет также большое число средств для точной настройки поведения драйвера в вашем приложении. Доступ к ним осуществляется посредством следующих функций:

```
GetEnvAttr       SetEnvAttr       --  4 attributes
GetConnectAttr   SetConnectAttr   -- 16 attributes
GetStmtAttr      SetStmtAttr      -- 33 attributes
```

До ODBC 3.x существовал более старый набор функций с именами, оканчивающимися на Option вместо Attr. Эти функции почти идентичны перечисленным, за исключением небольшого числа атрибутов. Чтобы узнать о подробностях использования этих функций, можно использовать описанную выше процедуру поиска в Microsoft MSDN.

Модуль Win32::ODBC предоставляет доступ к функциям GetInfo(), Get/SetConnect_Option() и Get/SetStmtOption(). В DBI определено крайне ограниченное подмножество этих функций посредством набора атрибутов дескрипторов DBI.

Доступ к ODBC из Perl

Итак, мы определили, что стандарт ODBC является хорошей вещью, но как ее использовать?

Для использования ODBC из Perl есть только две практические возможности: модуль Win32::ODBC и DBI с модулем DBD::ODBC. Сначала мы опишем DBD::ODBC, а затем более пристально рассмотрим Win32::ODBC.

DBD::ODBC

Модуль DBD::ODBC написали Тим Банс (Tim Bunce) и Джефф Арлвин (Jeff Urlwin), основываясь на исходном коде, который создал Томас К. Венрих (Thomas K. Wenrich). Это расширение Perl, написанное на C и не привязанное к платформам Microsoft Win32, что делает его хорошим выбором для прямого использования ODBC в Unix, VMS и других не-Windows системах.

Будучи драйвером DBI, модуль DBD::ODBC имеет главной целью реализацию функций, требуемых DBI, а не просто предоставление доступа к ODBC из Perl.

Драйвер DBD::ODBC более подробно описан в приложении B «Характеристики драйверов и баз данных».

[1] Все функции Microsoft ODBC имеют префикс SQL.

Win32::ODBC

Модуль Win32::ODBC написал Дэйв Рот (Dave Roth), основываясь на исходном коде, который создал Дэн Де Маггио (Dan DeMaggio). Это расширение Perl, написанное на C++ и тесно связанное с платформой Win32.

Главной задачей модуля Win32::ODBC является предоставление прямого доступа к функциям ODBC. С этой точки зрения Win32::ODBC представляет довольно тонкий интерфейс низкого уровня.

Вот пример программы, использующей Win32::ODBC:

```
use Win32::ODBC;

### Соединиться с источником данных
$db = new Win32::ODBC("DSN=MyDataDSN;UID=me;PWD=secret")
    or die Win32::ODBC::Error();

### Подготовить и выполнить команду
if ($db->Sql("SELECT item, price FROM table")) {
    print "Ошибка SQL: " . $db->Error() . "\n";
    $db->Close();
    exit;
}

### Выбрать строку из источника
while ($db->FetchRow) {
  my ($item, $price) = $db->Data(); ### Получить значения данных из строки
    print "Изделие $item = $price\n";
}

### Отсоединиться
$db->Close();
```

Наиболее значительными недостатками Win32::ODBC по сравнению с DBD::ODBC являются следующие:

Нет отдельного дескриптора команды

Для хранения деталей текущей команды используется дескриптор соединения с базой данных. Отдельного дескриптора команды нет, поэтому с дескриптором базы данных можно выполнить только одну команду. Это не столь страшно, как может показаться, поскольку можно клонировать дескрипторы базы данных, в результате чего несколько дескрипторов могут совместно использовать одно соединение с базой данных ODBC.

Нет отдельных этапов подготовки и выполнения команды

Нельзя подготовить команду, чтобы выполнить ее позднее. Метод Sql(), подобно методу do() в DBI, объединяет оба этапа вместе.

Заполнители и связанные параметры не поддерживаются

Это, вероятно, самый большой недостаток Win32::ODBC. Все значения должны передаваться в командах SQL в виде литералов.

Отсутствие поддержки заполнителей, особенно вкупе с невозможностью подготовки команд, ведет к тому, что нетривиальные приложения, основанные на Win32::ODBC, создают более тяжелую нагрузку для серверов баз данных и потому медленнее выполняются.

Это также создает проблемы при вставке двоичных данных, таких как изображения.

Выборка строк является двухэтапным процессом

Метод FetchRow() фактически не возвращает данные в сценарий. Чтобы получить строку значений данных, нужно выполнить либо метод Data() для получения простого списка (как в fetchrow_array()), либо метод DataHash() для получения хэша (как в fetchrow_hashref()).

Это, скорее, неудобство, чем существенный недостаток. Кроме того, это вторая причина, по которой использование Win32::ODBC несколько замедляет работу в сравнении с DBI.

Отсутствует автоматическая обработка ошибок

В ODBC не существует эквивалента механизму DBI RaiseError и PrintError. Необходимо явным образом проверять статус, возвращаемый при вызове методов Win32::ODBC, чтобы писать надежные приложения.

Отсутствие автоматической обработки ошибок делает Win32::ODBC менее пригодным для нетривиальных приложений, для которых важна надежность. Это особенно относится к приложениям, использующим транзакции.

Win32::ODBC работает несколько медленнее, чем DBD::ODBC

Даже при выполнении простых запросов Win32::ODBC оказывается несколько медлительнее, чем DBD::ODBC, для одной и той же платформы и базы данных. Как и всегда при использовании тестов, получаемые результаты могут различаться, поэтому произведите собственную проверку, если этот вопрос для вас важен.

Планируется заняться некоторыми из этих недостатков при подготовке следующих версий. Наиболее существенными преимуществами Win32::ODBC в сравнении с DBD::ODBC являются следующие:

Можно использовать большую часть ODBC API

В данное время это самое большое преимущество Win32::ODBC перед DBD::ODBC.

Остальные пункты в этом списке относятся, в сущности, к важным функциям ODBC, а не самого модуля Win32::ODBC, но если они не

поддерживаются в DBD::ODBC, то могут считаться преимуществами Win32::ODBC.

Имеются атрибуты, опции и метаданные

Они описаны в предыдущем разделе. Имеется широкий диапазон функций метаданных, а также функций, управляющих многими атрибутами и опциями.

Поддерживаются прокручиваемые курсоры

Прокручиваемые курсоры позволяют читать данные, возвращаемые запросом, в любом порядке. Можно перейти на последнюю строку и читать данные в обратном направлении. Можно перейти на любую строку по ее абсолютному номеру или относительно текущей строки. Это очень удобно для использования в приложениях, осуществляющих интерактивный просмотр данных.

Альянс DBI с ODBC

Сильное влияние на DBI оказали ODBC и лежащие в его основе международные стандарты (X/Open SQL CLI и ISO/IEC 9075-3:1995 SQL/CLI). Разработка модуля DBD::ODBC дала DBI более прочную поддержку в мире ODBC.

Спецификация DBI развивается во времени естественным образом. Стандарт ODBC дает ей основанные на стандартах рамки развития. Например, если необходимо добавить метод, возвращающий сведения о типах данных, поддерживаемых базой данных, то значительно разумнее следовать образцу проверенной стандартной функции, чем идти своим путем. Поэтому реализация метода DBI type_info очень напоминает функцию ODBC Get_TypeInfo.

По мере развития модулей DBI и DBD::ODBC происходит их естественное сближение. Поскольку есть два отличных переносимых менеджера драйверов Open Source,[1] модуль DBD::ODBC должен стать таким же переносимым, как сам DBI. Поэтому вполне может иметь смысл объединить их вместе.

Вопросы и предпочтения

Надеемся, что теперь мы ответили на все ваши вопросы относительно использования ODBC в Perl, кроме одного. Если нужно сделать выбор между Win32::ODBC и DBD::ODBC, что следует предпочесть?

[1] Их можно получить из проекта FreeODBC на: *http://users.ids.net/~bjepson/FreeODBC/*.

На самом деле этот выбор достаточно прост. Если вам требуется использовать больше функций ODBC API, чем предоставляет DBD::ODBC, используйте Win32::ODBC. В противном случае комбинация DBI с DBD::ODBC может оказаться лучшим выбором.

Переход с Win32::ODBC на DBI

Допустим, что у вас написано приложение, использующее Win32::ODBC, но теперь вам необходимо, чтобы оно работало на машине без Windows, или вы хотите использовать DBI по какой-то другой причине.

С технической точки зрения, лучше всего переписать приложение заново, но в жизни мы не всегда можем позволить себе такую роскошь. Более практичным решением явится использование какого-либо эмулятора Win32::ODBC, в котором «под капотом» находится DBI. Как раз это делает модуль Win32::DBIODBC, поставляемый с DBI.

В модуле Win32::DBIODBC реализована достаточная часть Win32::ODBC API, чтобы многие несложные приложения были перенесены в результате изменения всего лишь одной строки. Вместо:

```
use Win32::ODBC;
```

нужно написать:

```
use Win32::DBIODBC;
```

В остальных местах своей программы вы по-прежнему ссылаетесь на модуль Win32::ODBC. Есть простой способ проверки приложений, при котором их вообще не нужно изменять. Просто загрузите Win32::DBI-ODBC из командной строки:

```
perl -MWin32::DBIODBC your_script_name
```

Win32::DBIODBC одурачит Perl, заставив его думать, что загружен Win32::ODBC, поэтому строка use Win32::ODBC; в вашем сценарии не будет иметь никакого действия.

Использовав эмуляцию, вы можете приступить к неспешной переработке своего приложения для прямого использования DBI.

А как в отношении ADO?

ADO (ActiveX Data Objects) является новейшим видом собственного API Microsoft для доступа к данным. Говорят, что «ADO является стратегическим для Microsoft интерфейсом высокого уровня для доступа ко всем видам данных».

Если это будет для вас облегчением, можете считать ADO глянцем, наведенным на ODBC, хотя в действительности он основывается на OLE

DB API, созданном в Microsoft. ADO обеспечивает доступ к базам данных ODBC, а также ко многим новым источникам данных, прежде не доступным через ODBC. Он является объектно-ориентированным и теоретически разработанным с целью облегчения использования.

Можно использовать ADO в Perl через модуль `Win32::OLE`. Вот пример:

```
use Win32::OLE;
$conn = Win32::OLE->new("ADODB.Connection");
$conn->Open("DSN=MyDSN;UID=MyUID;PWD=MyPwd");
$RS = $conn->Execute("SELECT isbn, title FROM books");
if (!$RS) {
    $Errors = $conn->Errors();
    die "Ошибки:\n", map { "$_->{Description}\n" } keys %$Errors;
}

while ( !$RS->EOF ) {
    my ($isbn, $title) = (
        $RS->Fields('isbn')->Value,
        $RS->Fields('title')->Value,
    );
    print "$isbn : $title\n";
    $RS->MoveNext();
}
$RS->Close();
$conn->Close();
```

Чтобы уберечь вас от необходимости изучения еще одного API-доступа, DBI приходит к вам на помощь с `DBD::ADO`. Драйвер `DBD::ADO` позволяет соединяться с любым источником данных ADO и осуществлять в нем выборку данных с использованием переносимого кода DBI Perl. Не нужно изучать новый API, а ваша жизнь будет значительно облегчена при необходимости переноса приложений в/из ADO.

8

Командный процессор для DBI и прокси-доступ к базам данных

В этой главе рассматриваются два важных дополнения к Perl DBI: командный процессор для баз данных и прокси-драйверы для доступа к драйверам удаленных баз данных по сети.

dbish – командный процессор DBI (Shell)

DBI Shell, или `dbish`, является средством командной строки, позволяющим выполнять произвольные команды SQL и осуществлять диагностику баз данных без написания законченных программ на Perl.

Допустим, к примеру, что нам требуется быстро получить список всех мегалитов в Уилтшире. Для этого мы могли бы написать законченную программу на Perl, которая соединяется с базой данных, подготавливает и выполняет необходимую команду SQL, осуществляет выборку данных, форматирует их и отсоединяется от базы данных.

Посредством DBI этот процесс осуществляется легко, однако это утомительно, если нужно всего лишь быстро получить какие-то данные.

Здесь вступает в дело `dbish`, позволяющий соединиться с источником данных и прямо передать ему команду. `dbish` сам осуществляет необходимые операции соединения, подготовки, выполнения, а вам выдает готовые результаты.

Запуск dbish

dbish является исполняемой программой, поставляемой с DBI. Запустить ее можно, введя следующее:

```
dbish
```

в результате чего появляется приглашение вида:

```
DBI::Shell 10.5 using DBI 1.14

WARNING: The DBI::Shell interface and functionality are
=======  very likely to change in subsequent versions!

Available DBI drivers:
 1: dbi:ADO
 2: dbi:ExampleP
 3: dbi:Oracle
 4: dbi:Proxy
Enter driver name or number, or full 'dbi:...:...' DSN:

(DBI::Shell 10.5 использующий DBI 1.14

ПРЕДУПРЕЖДЕНИЕ: Интерфейс DBI::Shell и исполняемые функции с большой
==============  вероятностью изменятся в последующих версиях!

Доступные драйверы DBI:
 1: dbi:ADO
 2: dbi:ExampleP
 3: dbi:Oracle
 4: dbi:Proxy
Введите имя драйвера или номер, или полное имя 'dbi:...:...' DSN:)
```

Некоторым драйверам для соединения с базой данных необходимы корректные имя пользователя и пароль. Для выполнения этого требования можно передать dbish дополнительные аргументы в виде:

```
dbish <источник_данных> [имя_пользователя] [пароль]
```

Например:

```
dbish '' stones stones
```

или:

```
dbish dbi: stones stones
```

В этом случае мы не указали драйвер и поэтому будем выбирать его интерактивно через меню. Можно обойтись без использования меню, указав имя источника данных для нужной базы данных:

```
dbish dbi:Oracle:archaeo stones stones
```

Если не указывать драйвер в командной строке, то выводится меню, позволяющее выбрать тип базы данных согласно списку имеющихся драйверов. Например, если мегалитическая база данных содержится в Oracle, то выбрать источник данных `dbi:Oracle` можно, введя `3`. В результате у заданного драйвера баз данных запрашиваются имеющиеся источники данных, например:

```
Enter data source to connect to:
 1: dbi:Oracle:archaeo
 2: dbi:Oracle:sales
Enter data source or number, or full 'dbi:...:...' DSN:

(Задайте источник данных для подсоединения:
 1: dbi:Oracle:archaeo
 2: dbi:Oracle:sales
Задайте источник данных или номер, или полное имя 'dbi:...:...' DSN:)
```

Этот пример показывает, что драйверу базы данных Oracle известны две локально сконфигурированные базы данных Oracle. Наши данные по мегалитам содержатся в базе данных `archaeo`, поэтому введите `1`.

Теперь `dbish` попытается соединиться с базой данных. После успешного соединения с источником данных вы увидите приглашение:

```
stones@dbi:Oracle:archaeo>
```

говорящее о том, что вы соединились с базой данных `dbi:Oracle:archaeo` в качестве пользователя `stones`, и `dbish` готов принять от вас команду.

Находясь в `dbish`, можно осуществить соединение с другой базой данных, выполнив команду `/connect`. Например:

```
stones@dbi:Oracle:archaeo> /connect dbi:Oracle:sales dbusername
Disconnecting from dbi:Oracle:archaeo.
Connecting to 'dbi:Oracle:sales' as 'dbusername'...
Password for 'dbusername' (not echoed to screen): ......
stones@dbi:Oracle:sales>

(stones@dbi:Oracle:archaeo> /connect dbi:Oracle:sales dbusername
Отсоединение от dbi:Oracle:archaeo.
Подсоединение к 'dbi:Oracle:sales' под именем 'dbusername'...
Пароль для 'dbusername' (на экране не отображается): ......
stones@dbi:Oracle:sales>)
```

К сожалению, одновременное соединение с несколькими базами данных пока не поддерживается в `dbish`.

Обработка команд

Чаще всего `dbish` используют для того, чтобы выполнить какую-то специальную команду в базе данных, проверить работоспособность команды перед включением ее в сценарий Perl или просто быстро полу-

чить ответ на какой-то вопрос. Это как раз те задачи, для решения которых был разработан dbish.

Команды dbish вводятся после косой черты (/), за которой следуют имя команды и, возможно, какие-либо дополнительные аргументы, например:

```
/help
```

Все вводимые данные, которые не начинаются с косой черты, рассматриваются как часть команды SQL и дописываются в «буфер команды». Когда команда SQL завершена, можно выполнить ее, и результаты, если они есть, будут выведены на экран.

Например, для получения названий всех площадок в базе данных мегалитов введите:

```
stones@dbi:Oracle:archaeo> SELECT name FROM megaliths
Current statement buffer (enter '/' to execute or '/help' for help):
(Буфер текущей команды (введите '/' для выполнения или '/help' для подсказки):)

SELECT name FROM megaliths

stones@dbi:Oracle:archaeo> /
'Эйвбери'
'Стоунхендж'
'Ландин Линкс'
...
[132 rows of 1 fields returned]
stones@dbi:Oracle:archaeo>
```

Обратите внимание, что для выполнения команды SQL можно ввести одну косую черту. После выполнения команды буфер команд очищается. Предположим, однако, что мы начинаем ввод нового запроса, а потом решаем, что нам нужны другие данные:

```
stones@dbi:Oracle:archaeo> SELECT name FROM megaliths
Current statement buffer (enter '/' to execute or '/help' for help):
SELECT name FROM megaliths
stones@dbi:Oracle:archaeo> SELECT name, mapref FROM megaliths
Current statement buffer (enter '/' to execute or '/help' for help):
SELECT name FROM megaliths
SELECT name, mapref FROM megaliths

stones@dbi:Oracle:archaeo>
```

Это совершенно неверно! К счастью, можно очистить буфер от старых команд и начать все заново с помощью команды /clear. Выполненные команды удаляются из буфера, но могут быть снова вызваны с помощью инструкции /history. Можно даже выполнить инструкцию /edit, чтобы запустить внешний редактор для редактирования SQL.

Формат вывода результатов команды SELECT может конфигурироваться командой /format. В настоящее время есть две формы этой команды – /format neat и /format box. По умолчанию установлен формат neat, использующий для форматирования данных функцию DBI::neat_list(). Например, команда:

```
stones@dbi:Oracle:archaeo> SELECT name, mapref FROM megaliths /
```

выводит на экран следующее:

```
'Эйвбери', 'SU 102 699'
'Стоунхендж' 'SU 123 422',
'Ландин Линкс', 'NO 404 027'
...
[132 rows of 1 fields returned]
```

Формат box выглядит более приятно:

```
+--------------+------------+
| name         | mapref     |
+--------------+------------+
| Эйвбери      | SU 102 699 |
+--------------+------------+
| Стоунхендж   | SU 123 422 |
+--------------+------------+
| Ландин Линкс | NO 404 027 |
+--------------+------------+
```

dbish позволяет выполнять команды SQL, не являющиеся SELECT, с помощью директивы /. Вам нужно удалить из таблицы все строки? Просто введите:

```
stones@dbi:Oracle:archaeo> delete from megaliths /
[132 rows affected]
stones@dbi:Oracle:archaeo>
```

Быстро, легко и смертельно! Таким способом можно выполнять все команды, отличные от SELECT, в том числе CREATE TABLE и даже вызовы хранимых процедур, если ваша база данных их поддерживает.[1]

Разные команды dbish

dbish является процессором командной строки с достаточно полными функциями,[2] в число которых входят некоторые удобные команды,

[1] Существует команда /do, в результате которой выполняется метод do(), а не prepare(), и затем execute(). На практике надобность в этом возникает редко.

[2] Мощными средствами редактирования командной строки dbish обязан модулям Term::Readline и Term::Readline::Gnu. Устанавливать их для использования dbish не обязательно.

позволяющие фиксировать и откатывать изменения в базе данных, повторно вызывать команды SQL и собственные инструкции, выполнявшиеся раньше, и даже выполнять произвольные команды Perl.

Одной из наиболее полезных является команда `/table_info`, перечисляющая таблицы в базе данных, к которой вы подключены. Это незаменимая команда, когда вы мучительно пытаетесь вспомнить, как называется эта мерзкая таблица!

Полный список всех команд можно получить, введя `/help`.

`dbish` является удобным инструментом для выполнения коротких задач в базе данных. Со временем он превратится в незаменимый инструмент администратора баз данных и разработчика программ для них, подобно таким фирменным утилитам, как SQL*Plus Oracle.

Доверительный доступ к базам данных

Доверительный доступ к базам данных (database proxying) означает возможность направления базе данных запросов с использованием промежуточного программного средства – прокси – и возврат результатов запросов без использования клиентской программой драйверов баз данных.

Например, прокси часто используются для того, чтобы клиентская программа, работающая на машине Unix, могла, скажем, посылать запрос базе данных Microsoft Access, находящейся на машине, работающей под Windows. Может оказаться, что на машине Unix нет программного обеспечения или драйверов ODBC и поэтому об ODBC ей ничего не известно. Это означает, что все запросы она должна пересылать прокси-серверу, которому известно об ODBC и базе данных Access. Прокси-сервер подает запрос и забирает результаты, которые возвращает обратно клиентской программе.

Эта очень мощная функция, поскольку позволяет получать доступ к базам данных практически в любых операционных системах из любой другой операционной системы при условии, что на обеих системах выполняются Perl и DBI. Дополнительное преимущество связывается с распространением программного обеспечения: если на клиентских PC используются сценарии Perl для доступа к базам данных Oracle, размещенным на центральном сервере Unix, то не требуется производить потенциально сложную инсталляцию клиента Oracle. Возможности прокси DBI делают это клиентское программное обеспечение лишним.

Более того, можно автоматически добавить «сетевые возможности» для баз данных, которые иным образом такие функции не поддерживают. Например, используя прокси и DBI, можно выполнять на машине Windows сценарий Perl, который запрашивает данные из файла CSV через сеть.

Наконец, архитектура прокси в DBI позволяет осуществлять сжатие «на лету» данных запросов и результатов, а также осуществлять шифрование этих данных. Эти две возможности делают DBI мощным средством передачи больших результирующих наборов данных через медленные сетевые соединения, например, через модем. Кроме того, DBI становится средством для передачи конфиденциальных данных. Более подробно мы обсудим эти две темы в одном из следующих разделов.

Архитектура доверительного доступа к базам данных

DBI поддерживает прокси-доступ к базам данных с помощью двух модулей – `DBD::Proxy` и `DBI::ProxyServer`. `DBD::Proxy` используется клиентскими программами для обмена данными с прокси-сервером, реализованным в модуле `DBI::ProxyServer`. Эта архитектура показана на рис. 8.1.

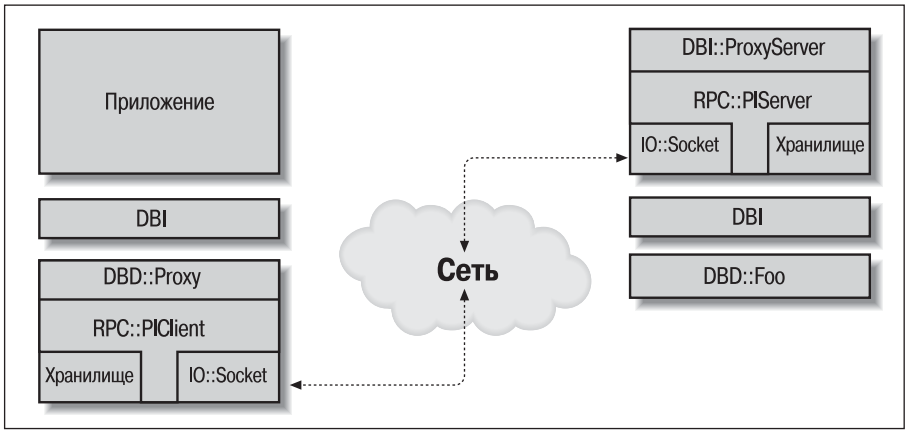

Рис. 8.1. *Архитектура прокси-доступа DBI*

Поскольку модуль `DBI::ProxyServer` использует драйверы баз данных для фактического интерфейса к базам данных, через прокси можно посылать запросы и обрабатывать данные в любых базах данных, включая файлы CSV и XBase (DBF). Архитектура прокси-доступа DBI не ограничивает вас использованием только баз данных высшего ряда, таких как Oracle или Informix.

Так как же использовать этот прокси-сервер? Рассмотрим пример часто встречающегося случая, когда программа Perl, работающая на машине Unix, должна послать запрос базе данных Microsoft Access, работающей на машине под Windows.

Настройка прокси-сервера

Прокси-сервер DBI является просто слоем поверх DBI; он может быть сервером только для источников данных, с которыми может соединяться собственно DBI. Поэтому, прежде чем заниматься настройкой прокси-сервера для установления соединений с клиентами, необходимо установить драйверы тех баз данных, которые понадобятся клиентам. В нашем примере соединения с базой данных Access необходимо установить модуль `DBD::ODBC`.[1]

Вам потребуется также сконфигурировать свой источник данных ODBC через панель управления Windows. Назовем источник данных ODBC для нашей мегалитической базы данных `archaeo`.

Можно убедиться в том, что этот источник данных правильно сконфигурирован, с помощью командного процессора DBI `dbish`, установленного на машине с Windows:

```
dbish dbi:ODBC:archaeo
```

либо с помощью короткого сценария, выполняемого на машине под Windows:

```
use DBI;
$dbh = DBI->connect( "dbi:ODBC:archaeo", "username", "password" );
$dbh->disconnect();
```

Если `dbish` производит соединение или при выполнении сценария не возникает ошибок, то, вероятно, все инсталлировано и сконфигурировано правильно.

Простейшим способом настройки прокси-сервера DBI является использование сценария `dbiproxy`, поставляемого с основным модулем DBI. Для модуля `DBI::ProxyServer`, используемого `dbiproxy`, есть обязательные модули, которые должны быть предварительно инсталлированы: `PlRPC` и `Net::Daemon`. Их можно загрузить и инсталлировать с CPAN с помощью:

```
perl -MCPAN -e 'install Bundle::DBI'
```

Либо, если вы работаете с ActiveState Perl под Windows, можно инсталлировать эти модули отдельно через PPM (поскольку PPM в данное время не поддерживает механизм связывания).

Главные данные, которые нужны `dbiproxy`, – это номер порта, на котором прокси-сервер должен слушать входящие клиентские соедине-

[1] При наличии компилятора можно получить его исходный код из CPAN и собрать самостоятельно или в Windows получить и инсталлировать откомпилированную версию с помощью средства PPM, предоставляемого с ActiveState Perl.

ния. Если номер порта – 3333, то можно запустить прокси-сервер с помощью команды:

```
dbiproxy --localport 3333
```

В результате запускается сервер, который ожидает соединений. Если вы хотите действительно убедиться в том, что сервер запущен и работает, можете запустить его с флагом --debug и необязательным флагом --logfile, указывающим, куда должна направляться отладочная информация.

Например, в результате:

```
dbiproxy --localport 3333 --debug
```

выдача отладочной информации будет осуществляться в окно командной подсказки на машине Windows или через syslog(1) на машине Unix. На машинах под Unix можно перенаправить выдачу в текущий терминал с помощью команды:

```
dbiproxy --localport 3333 --debug --logfile /dev/tty
```

Она должна правильно работать на большинстве платформ Unix.

Соединение с прокси-сервером

Теперь, когда мы сконфигурировали наш прокси-сервер, чтобы он ждал клиентов на порту 3333 нашей машины под Windows, необходимо сообщить клиентской программе Perl на машине Unix, что она должна использовать этот прокси-сервер, а не пытаться устанавливать с базой данных непосредственное соединение.

Например, приведенный ранее сценарий для проверки ODBC соединяется с базой данных напрямую через модуль DBD::ODBC с помощью следующего вызова DBI->connect():

```
$dbh = DBI->connect( "dbi:ODBC:archaeo", "username", "password" );
```

Для локальных соединений это прекрасно, но как перевести это в нечто, используемое прокси-сервером?

DBD::Proxy использует ряд необязательных аргументов, которые можно добавить к DSN при указании базы данных, с которой нужно соединиться. DBD::Proxy позволяет задать имя узла той машины, на которой работает прокси-сервер, номер порта, который прослушивает прокси-сервер, и источник данных, к которому прокси-сервер должен вас присоединить.

Поэтому чтобы соединиться с базой данных ODBC с именем archaeo на машине Windows с именем fowliswester и прокси-сервером, прослушивающим порт 3333, нужно использовать следующий синтаксис DBI->connect():

```
$dsn   = "dbi:ODBC:archaeo";
$proxy = "hostname=fowliswester;port=3333";
$dbh = DBI->connect( "dbi:Proxy:$proxy;dsn=$dsn", '', '' );
```

Хотя этот код выглядит довольно длинным, на самом деле он является компактным и переносимым способом установления сквозных соединений с удаленными базами данных через прокси-сервер.

Когда вы соединитесь с прокси-сервером, а тот соединится с нужным источником данных, вы получите активный дескриптор базы данных, позволяющий передавать базе данных запросы так, как если бы вы соединились с ней напрямую. Поэтому при использовании прокси-сервера изменяется только вызов `DBI->connect()`, что совершенно равносильно переключению с одной базы данных на другую.

В дополнение к сказанному можно использовать прокси-сервер, вообще не внося изменений в программы. Требуется лишь установить переменную окружения `DBI_AUTOPROXY`, а DBI сделает все остальное. В предыдущем примере можно сохранить команду `connect()`, обращающуюся к `dbi:ODBC:archaeo`, и просто присвоить переменной окружения `DBI_AUTOPROXY` следующее значение:

```
dbi:Proxy:hostname=fowliswester;port=3333
```

Значение, содержащееся в `DBI_AUTOPROXY`, конкатенируется с DSN, указанным в вызове `DBI->connect()`, для создания правильного прокси-DSN. Например:

```
$ENV{DBI_AUTOPROXY} = 'dbi:Proxy:hostname=fowliswester;port=3333';
$dbh = DBI->connect( "dbi:ODBC:archaeo", "username", "password" );
```

приведет к тому, что сценарий будет пытаться соединиться со следующим DSN:

```
dbi:Proxy:hostname=fowliswester;port=3333;dsn=dbi:ODBC:archaeo
```

Другой важный момент, который нужно отметить: на клиентской машине вообще не нужно устанавливать никаких серверов баз данных. Драйверы баз данных используются только прокси-сервером.

Более сложные темы

Архитектура прокси-доступа в DBI реализована поверх сетевых модулей Perl низкого уровня, таких как `PlRPC`, и в случае `DBI::ProxyServer`, `Net::Daemon`. Эти модули имеют множество функций, которые в результате становятся доступными в архитектуре DBI, например, мощные средства конфигурирования списков доступа, а также сжатие и шифрование «в линию».

Каждую из этих тем мы рассмотрим более подробно и объясним, как может быть организовано их эффективное использование в ваших программах.

Конфигурирование доступа

Модули `Net::Daemon`, `RPC::PlServer` и `DBI::ProxyServer` имеют общую систему файлов конфигурации в силу того, что `RPC::PlServer` наследует модулю `Net::Daemon`, а `DBI::ProxyServer` наследует `RPC::PlServer`.[1]

Файлы конфигурации для этих модулей являются сценариями Perl, в которых устанавливаются различные опции. Наиболее полезными являются опции, позволяющие использовать списки доступа. Списки доступа позволяют управлять тем, какие машины могут соединяться с прокси-сервером, и устанавливать режим работы сетевого транспортного протокола, по которому осуществляется связь этих машин с прокси-сервером.

Например, если у вас есть защищенная корпоративная локальная сеть, содержащая сервер баз данных и компьютеры клиентов, то можно разрешить машинам клиентов соединяться с центральной базой данных через прокси-сервер без использования аутентификации и шифрования. Это означает, что соединение PC с сетью является *доверительным*.

Однако компьютерам, установленным дома у служащих, которым нужен доступ к базе данных, нельзя доверять, поскольку поток данных через телефонные линии, в принципе, может быть перехвачен конкурентами. Поэтому данные, которыми обмениваются эти машины и центральный сервер базы данных, шифруются.

Простой файл конфигурации прокси-сервера может выглядеть так:

```
{
    facility => 'daemon',
    pidfile  => '/var/dbiproxy/dbiproxy.pid',
    user  => 'nobody',
    group => 'nobody',
    localport => '3333',
    mode      => 'fork',

    # Контроль доступа
    clients => [
        # Доступ из локальной сети ( 192.168.1.* )
        {
            mask   => '^192\.168\.1\.\d+$',
            accept => 1
        },
```

[1] Все эти модули, в том числе `DBD::Proxy`, разработал и реализовал один автор, Jochen Wiedmann. Спасибо, Jochen.

```
        # Доступ удаленных машин ( 192.168.2.* ),но с применением
        # шифрования
        {
            mask   => '^192\.168\.2\.\d+$',
            accept => 1,
            # Вскоре мы обсудим используемые алгоритмы шифрования
            cipher => Crypt::IDEA->new( 'be39893df23f98a2' )
        },
        # Больше никого не пускать
        {
            mask   => '.*',
            accept => 0
        }
    ]
}
```

Сценарий `dbiproxy` можно следующим образом запустить со специальным файлом конфигурации:

```
dbiproxy --configfile <filename>
```

Например, если сохранить приведенный выше файл конфигурации как `proxy.config`, то можно запустить `dbiproxy` командой:

```
dbiproxy --configfile proxy.config
```

Более того, файл конфигурации `DBI::ProxyServer` позволяет применять списки доступа к отдельным типам команд SQL. Например, вы можете потребовать, чтобы группа служащих могла получать данные об объеме продаж, но не могла их изменять. Это можно осуществить с помощью таких параметров конфигурации:

```
# Разрешить только запросы SELECT
# от рабочих станций ( 192.168.3.* )
clients => [
    {
        mask   => '^192\.168\.3\.\d+$',
        accept => 1,
        sql    => {
            select => 'SELECT name, mapref FROM megaliths WHERE name = ?'
        }
    },
]
```

В других параметрах, ограничивающих использование команд, можно использовать `insert`, `update` и `delete`. Например, если вы хотите разрешить выполнение только команд `DELETE` определенного вида, можно задать контроль доступа так:

```
sql => {
    delete => 'DELETE FROM megaliths WHERE id = ?'
}
```

В результате будут отклоняться все команды DELETE, которые не удовлетворяют заданной маске, в том числе зловредные команды DELETE FROM megaliths.

Итак, функции контроля доступа, имеющиеся в DBI::ProxyServer и его родительских модулях, могут использоваться для того, чтобы легко и быстро строить сложные (но гибкие) сетевые системы баз данных.

Сжатие

В предыдущем примере мы обсуждали возможность передачи запросов серверу через модем и прокси-сервер. Предположим, что пользователь подает запрос, который возвращает 100 000 строк данных, каждая из которых имеет размер около 1 Кбайт. Для медленного сетевого соединения это слишком большой объем информации.

Чтобы ускорить работу, можно сконфигурировать прокси-сервер так, чтобы он осуществлял линейное сжатие данных (с помощью модуля Compress::Zlib) для тех клиентов, которые обращаются к базе данных через соединение по коммутируемой линии. В результате резко сокращается объем данных, передаваемых с помощью модемов. Сделать это можно, запустив сценарий dbiproxy с дополнительным аргументом:

```
--compression gzip
```

который требует, чтобы использовался метод GNU gzip.

Чтобы ваш клиент мог передавать и принимать сжатые данные от прокси-сервера DBI, нужно и прокси-серверу сообщить о необходимости использовать сжатие. Это делается путем задания в DSN дополнительной пары ключ/значение в виде compression=gzip при соединении с базой данных, например:

```
$proxyloc = 'hostname=fowliswester;port=3333';
$compression = 'compression=gzip';
$dsn = 'dbi:ODBC:archaeo';
$dbh = DBI->connect( "dbi:Proxy:$proxyloc;$compression;dsn=$dsn",
                     "username", "password" );
```

Расплатой является затрата времени центральных процессоров на компрессию и декомпрессию данных прокси-сервером и прокси-клиентом, соответственно. С точки зрения клиента это, вероятно, не составляет проблемы, но на прокси-сервер может оказать воздействие, особенно если одновременно выполняется несколько запросов и для каждого требуется сжатие.

Тем не менее, компрессия является полезной и очевидной функцией, способной повысить эффективность сетей и баз данных при использовании прокси-сервера DBI.

Шифрование

Последней темой конфигурации прокси-сервера, о которой мы расскажем, является линейное шифрование данных.

Эта функция полезна при реализации защищенной сетевой среды базы данных, в которой операции с базой данных могут происходить через незащищенные сетевые связи, например, с использованием телефонных линий и общественного провайдера Интернета. Например, служащий, находящийся дома, может пожелать использовать свой вход в Интернет для доступа к защищенной базе данных компании. Либо вам понадобится осуществить электронную коммерческую сделку между двумя финансовыми учреждениями.

Оба примера являются кандидатами на использование механизма шифрования `DBI::ProxyServer`. Шифрование реализовано в модулях `RPC::PlClient` и `RPC::PlServer`. Это позволяет `DBD::Proxy` и `DBI::ProxyServer` использовать эти механизмы посредством наследования. Фактические процедуры шифрования используют для генерации ключей и сравнения внешние модули `Crypt::IDEA` или `Crypt::DES`.[1]

Самой общей предпосылкой образования потока зашифрованных данных является генерация клиентом и сервером *ключей,* которыми они обмениваются друг с другом. Когда клиент хочет передать данные серверу, он зашифровывает их ключом сервера. Аналогично при необходимости передать данные от сервера к клиенту сервер сначала использует ключ клиента для их зашифровки. Такая система позволяет клиенту и серверу надежно декодировать поступающие данные. Поскольку данные шифруются до их передачи и расшифровываются после получения, при всяком перехвате сетевого трафика будут видны лишь зашифрованные данные.

Поэтому для поддержки шифрования в прокси-доступе через DBI необходимо конфигурирование как клиента, соединяющегося с прокси-сервером, так и сервера, для использования одинакового шифрования.

Конфигурирование клиента тривиально и состоит в указании дополнительных аргументов при вызове `DBI->connect()`. Например, для использования `Crypt::IDEA` в качестве метода шифрования можно записать:

```
use Crypt::IDEA;

### Ключ является произвольным, но шестнадцатеричным числом!
$key = 'b3a6d83ef3187ac4';
```

[1] Технические различия, а также входные и выходные данные этих алгоритмов находятся за пределами данной книги. Вам нужно посмотреть документацию к этим модулям, чтобы получить ссылки на тексты, в которых обсуждаются различные криптографические алгоритмы.

```
### Соединиться с прокси-сервером
$dbh = DBI->connect( "dbi:Proxy:cipher=IDEA;key=$key;...", '', '' );
```

Фактическое образование ключа происходит при создании нового объекта указанного типа шифрования (в данном случае `Crypt::IDEA`) с заданным значением ключа. Этот шифровальный объект передается прокси-серверу. Можно использовать модуль `Crypt::DES` для обеспечения криптографических сервисов, если заменить `cipher=IDEA` на `cipher=DES`.[1] Это показывает, как можно настраивать механизмы шифрования прокси-сервера DBI.

Например, при передаче конфиденциальных, но не секретных данных из базы данных на компьютере, находящемся у кого-либо дома, можно использовать относительно слабый механизм шифрования, предоставляемый `Crypt::DES`. Однако при передаче важных данных можно перейти на более сильное, но медленное шифрование, осуществляемое `Crypt::IDEA`.

Конфигурирование прокси-сервера столь же просто и осуществляется путем задания правил шифрования в файле конфигурации прокси-сервера. Например, простая конфигурация прокси-сервера, при которой весь трафик зашифровывается с помощью `Crypt::IDEA`, может быть записана так:

```
require Crypt::IDEA;

### Ключ, которым должны шифроваться данные
$key = 'b3a6d83ef3187ac4';

{
    clients => [ {
        'accept' => 1,
        'cipher' => IDEA->new( pack( "H*", $key ) )
      } ]
}
```

Важным аспектом этого файла конфигурации является то, что ключ, используемый для создания объекта `Crypt::IDEA`, соответствует тому, который используется клиентскими программами для соединения с этим прокси-сервером. Если ключи не совпадают, соединение не будет осуществлено, поскольку клиент и сервер окажутся не в состоянии расшифровывать данные из потока, проходящего через сетевое соединение.

[1] Потребуется также соответствующим образом изменить строку `use Crypt::IDEA`.

A

Спецификация DBI

Это приложение является слегка отредактированной версией спецификации DBI, «живым документом», медленно, но верно развивающимся по мере выпуска новых версий DBI. Данный текст основан на спецификации DBI для версии DBI 1.14.

Хотя мы отдаем себе отчет в том, что это приложение окажется несколько устаревшим к тому моменту, когда вы станете его читать, мы все-таки включили в книгу спецификацию, поскольку это важный справочный материал, и мы считаем, что без него книга была бы неполной. Для получения самой современной информации обратитесь к электронной документации для установленной у вас версии DBI. Обычно доступ к электронной документации осуществляется командой `perldoc DBI`. Файл *Changes* (изменения), поставляемый в дистрибутиве DBI, содержит подробные сведения об изменениях.

Учтите, что при всяком внесении изменений в DBI требуется некоторое время, чтобы они отразились в драйверах. В последних версиях DBI появились новые функции (помеченные в тексте как *NEW* (новый)), которые могут еще не поддерживаться в используемых вами драйверах. Если вам нужна поддержка новых функций, свяжитесь с авторами этих драйверов.

Краткий обзор

```
use DBI;

@driver_names = DBI->available_drivers;
@data_sources = DBI->data_sources($driver_name);

$dbh = DBI->connect($data_source, $username, $auth, \%attr);
```

```
$rv  = $dbh->do($statement);
$rv  = $dbh->do($statement, \%attr);
$rv  = $dbh->do($statement, \%attr, @bind_values);

$ary_ref = $dbh->selectall_arrayref($statement);
@row_ary = $dbh->selectrow_array($statement);
$ary_ref = $dbh->selectcol_arrayref($statement);

$sth = $dbh->prepare($statement);
$sth = $dbh->prepare_cached($statement);

$rv = $sth->bind_param($p_num, $bind_value);
$rv = $sth->bind_param($p_num, $bind_value, $bind_type);
$rv = $sth->bind_param($p_num, $bind_value, \%attr);

$rv = $sth->execute;
$rv = $sth->execute(@bind_values);

$rc = $sth->bind_col($col_num, \$col_variable);
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);

@row_ary  = $sth->fetchrow_array;
$ary_ref  = $sth->fetchrow_arrayref;
$hash_ref = $sth->fetchrow_hashref;

$ary_ref  = $sth->fetchall_arrayref;

$rv  = $sth->rows;

$rc  = $dbh->commit;
$rc  = $dbh->rollback;

$sql = $dbh->quote($string);

$rc  = $h->err;
$str = $h->errstr;
$rv  = $h->state;

$rc  = $dbh->disconnect;
```

Как получить помощь

Если у вас возникли вопросы в отношении использования DBI, можно обратиться за помощью посредством списка рассылки *dbi-users@isc.org*. Подписаться на этот список можно, посетив страницу:

http://www.isc.org/dbi-lists.html

Стоит также посетить домашнюю страницу DBI по адресу:

http://www.symbolstone.org/technology/perl/DBI

Прежде чем задавать вопросы, перечитайте этот документ, просмотрите архивы и прочтите DBI FAQ. Архивы перечислены в конце этого документа. FAQ инсталлируется как модуль `DBI::FAQ`, и читать его можно, выполнив команду `perldoc DBI::FAQ`.

Учтите, что Тим Банс не поддерживает списки рассылки и веб-страницу (это делают великодушные добровольцы). Поэтому не шлите письма прямо ему, у него просто нет времени на то, чтобы отвечать лично. В почтовом списке *dbi-users* есть много опытных людей, которые смогут вам помочь.

Описание

DBI является модулем доступа к базам данных для языка программирования Perl. В нем определен ряд методов, переменных и соглашений, составляющих единообразный интерфейс баз данных, не зависящий от конкретной используемой базы данных.

Важно помнить, что DBI представляет собой просто интерфейс, нечто вроде прослойки «клея» между приложением и одним или несколькими модулями *драйверов* баз данных. Самую большую часть работы выполняют именно драйверы. DBI предоставляет стандартный интерфейс и структуру, в рамках которой действуют драйверы.

Архитектура приложения DBI

API, или интерфейс прикладного программирования, определяет интерфейс вызовов и переменные, которые должны использоваться сценариями Perl. API реализован в расширении Perl DBI (рис. A.1).

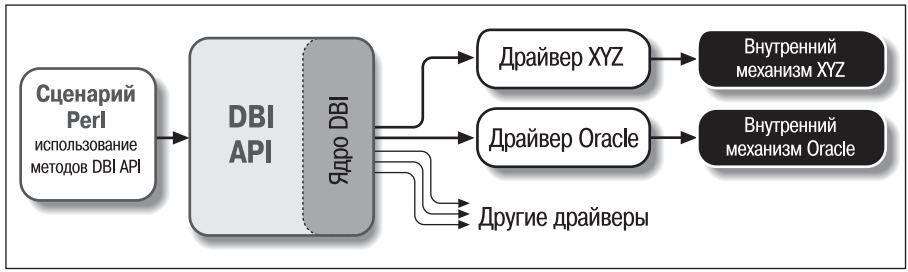

***Рис. A.1.** Архитектура приложения DBI*

DBI «пересылает» вызовы методов надлежащим драйверам для фактического исполнения. DBI отвечает также за динамическую загрузку драйверов, проверку наличия ошибок и их обработку, предоставление выполняемых по умолчанию реализаций методов и выполняет другие задачи, не связанные с конкретными базами данных.

В каждом драйвере реализованы методы DBI с использованием функций частного интерфейса ядра соответствующей базы данных. Заботиться о драйверах нужно только разработчикам сложных приложений, работающих с несколькими базами данных, или разработчикам общих библиотечных функций.

Обозначения и соглашения

В данном документе используются следующие соглашения:

`$dbh`

Объект – дескриптор базы данных.

`$sth`

Объект – дескриптор команды.

`$drh`

Объект – дескриптор драйвера (редко используется в приложениях).

`$h`

Любой из перечисленных выше дескрипторов (`$dbh`, `$sth` или `$drh`).

`$rc`

Общий код возврата (булев: `true`=OK, `false`=ошибка).

`$rv`

Общее возвращаемое значение (обычно целое число).

`@ary`

Список значений, возвращаемых базой данных; обычно строка данных.

`$rows`

Число обработанных строк (если оно доступно, иначе `-1`).

`$fh`

Дескриптор файла.

`undef`

Значения NULL представляются в Perl как неопределенные.

`\%attr`

Ссылка на хэш значений атрибутов, передаваемых методам.

Заметим, что Perl автоматически уничтожает объекты – дескрипторы баз данных и команд при удалении всех ссылок на них.

Основные принципы работы

Чтобы использовать DBI, нужно сначала загрузить модуль DBI:

```
use DBI;
use strict;
```

(Директива `use strict;` не является обязательной, но настоятельно рекомендуется к использованию.)

Затем необходимо *соединиться* с источником данных и получить *дескриптор* соединения:

```
$dbh = DBI->connect($dsn, $user, $password,
                    { RaiseError => 1, AutoCommit => 0 });
```

Поскольку установление соединения может потребовать времени, оно обычно производится в начале программы, а в конце происходит отсоединение.

Настоятельно рекомендуется явно определить поведение при автоматической фиксации – `AutoCommit`, что в будущих версиях может стать обязательным. Этим атрибутом устанавливается, должны ли изменения автоматически фиксироваться в базе данных при выполнении команд или фиксация должна осуществляться позже явным образом.

DBI позволяет приложению «готовить» команды для последующего выполнения. Подготовленная команда отождествляется с дескриптором команды, находящимся в переменной Perl. В наших примерах эта переменная будет называться `$sth`.

Типичная последовательность вызова методов для команды `SELECT` следующая:

```
prepare,
  execute, fetch, fetch, ...
  execute, fetch, fetch, ...
  execute, fetch, fetch, ...
```

Например:

```
$sth = $dbh->prepare("SELECT foo, bar FROM table WHERE baz=?");

$sth->execute( $baz );

while ( @row = $sth->fetchrow_array ) {
  print "@row\n";
}
```

Типичная последовательность вызова методов для команды, отличной от `SELECT`, следующая:

```
prepare,
  execute,
  execute,
  execute.
```

Например:

```
$sth = $dbh->prepare("INSERT INTO table(foo,bar,baz) VALUES (?,?,?)");

while(<CSV>) {
  chop;
  my ($foo,$bar,$baz) = split /,/;
      $sth->execute( $foo, $bar, $baz );
}
```

Метод `do()` можно использовать для неповторяющихся *не*-`SELECT` команд (или с драйверами, не поддерживающими заполнители):

```
$rows_affected = $dbh->do("UPDATE your_table SET foo = foo + 1");
```

Чтобы зафиксировать изменения в базе данных (когда `AutoCommit` отключен):

```
$dbh->commit;  # или вызвать $dbh->rollback; для отмены изменений
```

Наконец, если вы завершили работу с источником данных, следует отсоединиться от него:

```
$dbh->disconnect;
```

Общие правила интерфейса и предостережения

В DBI нет понятия «текущего сеанса». У каждого сеанса есть дескриптор, т. е. `$dbh`, возвращенный методом `connect`. Этот объект используется для вызова методов, относящихся к базе данных.

Большинство данных возвращается сценарию Perl в виде строк. (Значения Null возвращаются как `undef`.) Это позволяет обрабатывать произвольные числовые данные без потери точности. Учтите, что Perl может не обеспечивать такой же точности, когда строка используется как число.

Дата и время возвращаются как строки символов в «родном» формате соответствующего ядра базы данных. Эффект временного пояса зависит от базы данных и драйвера.

Perl поддерживает двоичные данные в своих строках, а DBI обменивается двоичными данными с драйвером без изменений. Как обрабатывать такие двоичные данные, решают разработчики драйвера.

В большинстве баз данных, способных работать с несколькими наборами символов, устанавливается глобальный набор символов по умолчанию. Хранящийся в базе данных текст должен состоять из символов этого набора. Если это не так, то виновата база данных или приложение, осуществлявшее вставку данных. Во время выборки текст должен автоматически преобразовываться в набор символов клиента, предположительно с учетом национальных установок. Если для осу-

ществления такого поведения драйверу необходимо установить флаг, то он должен сделать это сам, а не требовать подобных действий от приложения.

Несколько команд SQL нельзя помещать в один дескриптор команды (`$sth`), хотя некоторые базы данных и драйверы поддерживают такую возможность (в том числе Sybase и SQL Server).

В данной версии DBI не поддерживается чтение записей в произвольном порядке, т. е. записи выбираются в том порядке, в котором их возвратила база данных, а после выборки они забываются.

DBI напрямую не поддерживает обновление и удаление по месту. Альтернативное решение можно найти в описании атрибута `CursorName`.

Разработчики отдельных драйверов могут предоставлять любые собственные функции и/или атрибуты дескрипторов, которые сочтут полезными. Собственные функции драйверов вызываются с помощью метода DBI `func()`. К собственным атрибутам драйвера доступ осуществляется так же, как и к стандартным атрибутам.

У многих методов есть необязательный параметр `\%attr`, его можно использовать для передачи данных драйверу, в котором реализован метод. За исключением особо оговоренных в документации случаев, параметр `\%attr` можно использовать только для передачи специфических для драйвера указаний. Обычно можно игнорировать параметр `\%attr` или передавать его как `undef`.

Соглашения по именам и пространство имен

Имя пакета DBI и все имена пакетов внутри него (`DBI::*`) зарезервированы для использования DBI. В расширениях и связанных модулях используется пространство имен `DBIx::` (см. *http://www.perl.com/CPAN/modules/by-module/DBIx/*). Имена пакетов, начинающиеся с `DBD::`, зарезервированы для использования драйверами баз данных DBI. Все переменные окружения, используемые DBI или отдельными DBD, начинаются с `DBI_` или `DBD_`.

Регистр букв в именах атрибутов имеет значение и играет важную роль в переносимости сценариев DBI. Регистр в имени атрибута используется для указания того, кто определил смысл этого атрибута и его возможные значения, как показано в следующей таблице.

Регистр имени	Кто определяет значение
ВЕРХНИЙ_РЕГИСТР	Стандарты, например X/Open, ISO SQL92 и т. д. (переносимы)
СмешанныйРегистр	DBI API (переносимый), подчеркивание не используется
нижний_регистр	Специфические для драйвера или ядра базы данных (не переносимы)

Крайне важно, чтобы разработчики драйверов при определении частных атрибутов использовали только имена на нижнем регистре. Частные имена атрибутов должны иметь префикс, образованный из имени драйвера или подходящего сокращения (например, `ora` для Oracle, `ing` для Ingres и т. д.).

Вот список специфических для драйверов префиксов:

```
ado    DBD::ADO
best   DBD::BestWins
csv    DBD::CSV
db2    DBD::DB2
f      DBD::File
file   DBD::TextFile
ib     DBD::InterBase
ing    DBD::Ingres
ix     DBD::Informix
msql   DBD::mSQL
mysql  DBD::mysql
odbc   DBD::ODBC
ora    DBD::Oracle
proxy  DBD::Proxy
solid  DBD::Solid
syb    DBD::Sybase
tuber  DBD::Tuber
xbase  DBD::XBase
```

SQL – язык запросов

Большинство драйверов DBI требует, чтобы приложения использовали для взаимодействия с ядром базы данных диалект SQL (Structured Query Language – структурированный язык запросов). По следующим адресам можно найти полезные сведения об SQL и дополнительные ссылки:

http://www.altavista.com/query?q=sql+tutorial

http://www.jcc.com/sql_stnd.html

http://www.contrib.andrew.cmu.edu/~shadow/sql.html

Сам DBI не требует использования какого-либо особого языка, он независим от языка. В терминах ODBC DBI находится в «прозрачном» (pass-thru) режиме, хотя для отдельных драйверов это может быть не так. Единственное требование состоит в том, чтобы запросы и другие команды представлялись одной строкой символов, передаваемой в качестве первого аргумента методам `prepare` или `do`.

Интересное изложение *действительной* истории РСУБД и SQL, основанное на воспоминаниях людей, причастных к их появлению, можно найти на:

http://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-018-html/sqlr95.html

Историю SQL вы узнаете, пройдя по ссылкам «And the rest» (И остальное) и «Intergalactic dataspeak» (Межгалактический обмен данными).

Заполнители и связанные значения

Некоторые драйверы поддерживают заполнители и связанные значения. *Заполнители*, называемые также параметрическими отметками, используются для обозначения в командах баз данных тех величин, которые будут переданы позже, перед выполнением подготовленных команд. Например, приложение может использовать следующую команду для вставки строки данных в таблицу `sales`:

```
INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
```

или такую команду для выборки описания продукта:

```
SELECT description FROM products WHERE product_code = ?
```

Символы `?` являются заполнителями. Объединение фактических значений с заполнителями называется связыванием (*binding*), а значения называют связанными (*bind values*).

При использовании заполнителей с квалификатором SQL `LIKE` нужно помнить, что заполнитель замещает целую строку, поэтому нужно писать `"... LIKE ? ..."` и включать символы-заместители в значение, связываемое с заполнителем.

Значения Null

Неопределенные величины, или `undef`, можно использовать для указания значений null. Однако следует проявлять осторожность в частном случае использования null при детализации команды `SELECT`.

Например:

```
SELECT description FROM products WHERE product_code = ?
```

Связывание `undef` (NULL) с заполнителем *не* приведет к выбору строк, в которых `productcode` имеет значение NULL. (Обратитесь к руководству по SQL для ядра вашей базы данных или любой книге по SQL за разъяснением причин этого.) Чтобы явным образом выбрать строки со значением NULL, нужно написать `"WHERE product_code IS NULL"`, а в общем случае нужно сказать:

```
... WHERE (product_code = ? OR (? IS NULL AND product_code IS NULL))
```

и связать одно и то же значение с обоими заполнителями.

Производительность

Если не использовать заполнители, то приведенная выше команда вставки должна содержать вставляемые значения в виде литералов, и для вставки каждой новой строки ее нужно заново готовить и выполнять. При использовании заполнителей команду вставки нужно готовить только один раз. Связанные значения для каждой строки можно передавать методу `execute` при каждом его вызове. При отсутствии необходимости повторной подготовки команды для каждой строки приложение обычно работает во много раз быстрее.

Вот пример:

```
my $sth = $dbh->prepare(q{
  INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
}) || die $dbh->errstr;
while (<>) {
    chop;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) || die $dbh->errstr;
}
$dbh->commit || die $dbh->errstr;
```

Дальнейшие подробности можно найти в описании `execute` и `bind_param`.

Использование расстановки кавычек в стиле `q{...}` в этом примере позволяет избежать путаницы с кавычками, которые могут использоваться в командах SQL. Используйте двойные кавычки, как в операторе `qq{...}`, если вам требуется вставлять переменные в строку. Более подробно см. об этом в разделе «Quote and Quote-Like Operators» на странице электронного руководства *perlop*.

См. также метод `bind_column`, который используется для установления связи между переменными Perl и выходными колонками команды `SELECT`.

Класс DBI

В этом разделе мы расскажем о методах класса DBI, вспомогательных функциях и динамических атрибутах, связываемых с общими дескрипторами DBI.

Методы класса DBI

Класс DBI предоставляет следующие методы:

connect

```
$dbh = DBI->connect($data_source, $username, $password)
```

```
        || die $DBI::errstr;
$dbh = DBI->connect($data_source, $username, $password, \%attr)
        || die $DBI::errstr;
```

`connect` устанавливает соединение с базой данных, или сеанс, в соответствии с заданным источником данных `$data_source`. Если соединение успешно установлено, возвращается объект дескриптора базы данных. Для закрытия соединения используйте метод `$dbh->disconnect`.

Если попытка установления соединения заканчивается неудачей (см. ниже), метод возвращает `undef` и устанавливает `$DBI::err` и `$DBI::errstr`. (Он *не* устанавливает `$!` и прочие переменные.) Обычно следует проверять значение, возвращаемое `connect`, и в случае появления ошибки выполнять `print $DBI::errstr`.

С помощью DBI можно осуществлять множественные одновременные соединения с несколькими базами данных через несколько драйверов. Нужно лишь выполнить вызов `connect` для каждой из баз данных и сохранить по экземпляру каждого возвращенного дескриптора базы данных.

Значение `$data_source` должно начинаться с `dbi:`*`driver_name`*`:`. При этом *`driver_name`* указывает на драйвер, который должен использоваться для установления соединения (регистр знаков имеет значение).

Если параметр `$data_source` имеет неопределенное значение или пуст, то DBI подставит значение из переменной окружения `DBI_DSN`. Если пустой является только часть *`driver_name`* (т. е. префиксом `$data_source` является `dbi::`), то используется переменная окружения `DBI_DRIVER`. Если ни одна из переменных не установлена, то `connect` приводит к выходу из программы.

Вот примеры значений `$data_source`:

```
dbi:DriverName:database_name
dbi:DriverName:database_name@hostname:port
dbi:DriverName:database=database_name;host=hostname;port=port
```

Не существует стандарта для текста, который следует за именем драйвера. Каждый драйвер вправе использовать любой синтаксис. Единственное требование DBI состоит в том, чтобы все данные располагались в одной строке. Следует выяснить в документации к используемым вами драйверам, какой синтаксис для них требуется. (Если автору драйвера требуется определить синтаксис для `$data_source`, рекомендуется использовать стиль ODBC, показанный в последнем приведенном примере.)

Если определена переменная окружения `DBI_AUTOPROXY` (а драйвером в `$data_source` не является `Proxy`), то запрос на соединение автоматически изменяется следующим образом:

```
dbi:Proxy:$ENV{DBI_AUTOPROXY};dsn=$data_source
```

и передается модулю DBD::Proxy. DBI_AUTOPROXY обычно устанавливается в виде «hostname=... ;port=...». Подробности см. в документации по DBD::Proxy.

Если $username или $password имеют неопределенные значения (а не просто пусты), то DBI подставляет значения из переменных окружения DBI_USER и DBI_PASS соответственно. DBI выдает предупреждение, если переменные окружения не определены. Однако систематическое использование этих переменных не рекомендуется по соображениям безопасности. Этот механизм предназначен, в основном, для облегчения тестирования.

DBI->connect автоматически инсталлирует драйвер, если он еще не инсталлирован. При инсталляции драйвера возвращается действующий дескриптор драйвера, либо происходит аварийное завершение программы с выводом сообщения об ошибке, состоящего из строки install_driver и описания возникшей проблемы. Таким образом, DBI->connect аварийно завершается при неудачной инсталляции драйвера, а при неудаче соединения просто возвращается undef, и в $DBI::errstr помещается сообщение об ошибке.

Затем драйверу передаются на обработку аргументы $data_source (с удаленным префиксом «dbi:...:»), $username и $password. В DBI не определена какая-либо интерпретация содержимого этих полей. Драйвер может интерпретировать поля $data_source, $username и $password любым образом и устанавливать любые нужные значения по умолчанию, соответственно ядру базы данных, к которому осуществляется доступ. (Например, Oracle использует переменные окружения ORACLE_SID и TWO_TASK, если $data_source не задана.)

Атрибуты AutoCommit и PrintError по умолчанию включены для каждого соединения. (Дальнейшие подробности см. в описании AutoCommit и PrintError.) Однако весьма рекомендуется явным образом определить AutoCommit, а не полагаться на значения по умолчанию. В будущих версиях DBI может выдаваться предупреждение, если AutoCommit не задан явно.

Чтобы изменить установки по умолчанию для PrintError, RaiseError, AutoCommit и других атрибутов, можно использовать параметр \%attr. Например:

```
$dbh = DBI->connect($data_source, $user, $pass, {
      PrintError => 0,
      AutoCommit => 0
});
```

Можно также задать значения атрибутов соединения в параметре $data_source. Например:

```
dbi:DriverName(PrintError=>0,Taint=>1):...
```

Значения отдельных атрибутов, заданные таким образом, имеют приоритет над значениями, указанными в параметре `\%attr` метода `connect`.

Когда это возможно, каждый сеанс (`$dbh`) независим от транзакций в других сеансах. Это полезно при необходимости держать в транзакциях открытые курсоры, например, если вы используете один сеанс для долгоживущих курсоров (обычно открытых только для чтения), а в другом сеансе используете короткие транзакции для обновления.

Для совместимости со старыми сценариями DBI можно указывать драйвер в качестве четвертого аргумента `connect` (вместо `\%attr`):

```
$dbh = DBI->connect($data_source, $user, $pass, $driver);
```

В этой «старомодной» форме `connect` параметр `$data_source` не должен начинаться с `dbi:`*`driver_name:`*. (В противном случае встроенный *`driver_name`* будет проигнорирован.) Заметим также, что в этой старой форме `connect` атрибут `$dbh->{AutoCommit}` имеет *неопределенное значение*, атрибут `$dbh->{PrintError}` выключен, а если `DBI_DSN` не определена, то проверяется старая переменная окружения `DBI_DBNAME`. Учтите, что из будущих версий DBI этот `connect` «в старом стиле» может быть исключен.

connect_cached (новый)

```
$dbh = DBI->connect_cached($data_source, $username, $password)
          || die $DBI::errstr;
$dbh = DBI->connect_cached($data_source, $username, $password, \%attr)
          || die $DBI::errstr;
```

`connect_cached` аналогичен *connect*, за исключением того, что возвращаемый дескриптор базы данных хранится в хэше с указанными параметрами. Если делается повторный вызов `connect_cached` с теми же значениями параметров, то возвращается соответствующий `$dbh` из кэша, если он еще годен к употреблению. Кэшированный дескриптор базы данных заменяется новым соединением, если произошло отсоединение или выполнение метода `ping` оказалось неудачным.

Обратите внимание, что относительно некоторых вещей этот метод действует иначе, чем постоянные соединения, реализованные в `Apache::DBI`.

В некоторых приложениях кэширование приносит пользу, но оно может также послужить источником проблем, и использовать его нужно с осторожностью. Поведение этого метода может измениться, поэтому при намерении использовать его в реально работающих приложениях следует проконсультироваться в листе рассылки *dbi-users*.

К кэшу можно обращаться (и очищать его) с помощью атрибута `CachedKids`.

available_drivers

```
@ary = DBI->available_drivers;
@ary = DBI->available_drivers($quiet);
```

Возвращает список всех имеющихся драйверов путем поиска модулей `DBD::*` в каталогах, указанных в `@INC`. По умолчанию выводится предупреждение в случае, если какие-либо драйверы замаскированы одноименными, находящимися в каталогах, просматриваемых ранее. Передача значения `true` в `$quiet` подавляет выдачу предупреждения.

data_sources

```
@ary = DBI->data_sources($driver);
@ary = DBI->data_sources($driver, \%attr);
```

Возвращает список всех источников данных (баз данных), доступных через указанный драйвер. Драйвер будет загружен, если это не было сделано раньше. Если значение `$driver` пустое или `undef`, то используется значение переменной окружения `DBIDRIVER`.

Источники данных возвращаются в форме, пригодной для передачи методу `connect` (т. е. имеют префикс `dbi:$driver:`).

Имейте в виду, что многие драйверы не умеют определять, какие источники данных доступны. Такие драйверы возвращают пустой или неполный список.

trace

```
DBI->trace($trace_level)
DBI->trace($trace_level, $trace_filename)
```

Используя метод `trace` класса DBI можно включить данные трассировки DBI для любых дескрипторов. Для включения трассировки конкретного дескриптора используйте аналогичный метод `$h->trace`.

Уровень трассировки может быть установлен следующим образом:

0 Трассировка отключена.

1 Трассировать вызовы методов DBI с показом возвращаемых результатов или ошибок.

2 Трассировать вход в методы с параметрами и возвращаемыми результатами.

3 То же, что выше, плюс некоторые данные высокого уровня от драйвера и некоторые внутренние данные.

4 То же, что выше, плюс подробная информация от драйвера. Также включается информация о мьютексах DBI при использовании Perl с потоками.

5 и выше

То же, что выше, со все более и более неясными сведениями.

Уровень трассировки 1 лучше всего подходит для простого просмотра происходящего. Уровень 2 хорошо выбирать для трассировки общего назначения. Уровни 3 и выше (вплоть до 9) лучше использовать для исследования особых проблем, когда необходимо посмотреть, что происходит «внутри» драйвера и DBI.

Вывод результатов трассировки подробен и обычно очень полезен. Его большая часть форматируется с помощью функции `neat`, поэтому строки могут быть отредактированы и укорочены.

Изначально вывод результатов трассировки производится на `STDERR`. Если задать `$trace_filename`, открывается файл в режиме дописывания, и вся информация (в том числе от других дескрипторов) перенаправляется в этот файл. Последующие вызовы `trace` без указания `$trace_filename` не меняют место, куда выводятся данные трассировки. Если значение `$trace_filename` является неопределенным, то данные трассировки посылаются на `STDERR`, а предыдущий файл с данными трассировки закрывается.

См. также в описаниях методов `$h->trace` и `$h->trace_msg` данные о переменной окружения `DBI_TRACE`.

Вспомогательные функции DBI

Кроме методов, перечисленных в предыдущем разделе, пакет DBI предоставляет следующие вспомогательные функции:

neat

```
$str = DBI::neat($value, $maxlen);
```

Возвращает строку, содержащую очищенное и аккуратное представление переданной величины.

Строки заключаются в кавычки, хотя внутренние кавычки не преобразуются с помощью управляющих символов. Величины, распознаваемые как числа, не заключаются в кавычки. Неопределенные значения (NULL) выводятся как `undef` (без кавычек). Невыводимые символы заменяются точками (`.`).

Результирующие строки, превышающие по длине `$maxlen`, укорачиваются до длины `$maxlen-4` и в конце добавляется «`...'`». Если значение `$maxlen` равно `0` или `undef`, то длина устанавливается равной `$DBI::neat_maxlen`, которая, в свою очередь, равна по умолчанию `400`.

Эта функция предназначена для форматирования значений в виде, пригодном для просмотра человеком. Она используется внутри DBI для форматирования выдачи trace. Обычно ее не следует использовать для форматирования значений в базе данных (см. также quote).

neat_list

```
$str = DBI::neat_list(\@listref, $maxlen, $field_sep);
```

Вызывает DBI::neat для каждого элемента из списка и возвращает результаты, соединенные строкой $field_sep, значением которой по умолчанию является ", ".

looks_like_number

```
@bool = DBI::looks_like_number(@array);
```

Возвращает true для каждого элемента, который выглядит как число. Возвращает false для каждого элемента, который не выглядит как число. Возвращает undef для каждого элемента, который не определен или пуст.

Динамические атрибуты DBI

Динамические атрибуты всегда связаны с последним использовавшимся дескриптором (ниже он обозначается как $h).

В случаях, когда атрибут эквивалентен вызову метода, см. документацию по вызову метода.

Предупреждение: эти атрибуты представлены для удобства, но они имеют определенные ограничения. В частности, у них короткий срок жизни: поскольку они связаны с последним использовавшимся дескриптором, их нужно использовать *немедленно* после вызова метода, который их «установил». Если возникают сомнения, используйте соответствующий метод.

$DBI::err

Эквивалентен $h->err.

$DBI::errstr

Эквивалентен $h->errstr.

$DBI::state

Эквивалентен $h->state.

$DBI::rows

Эквивалентен $h->rows. Пожалуйста, посмотрите документацию по методу rows.

Методы, являющиеся общими для всех дескрипторов

Следующие методы можно использовать со всеми типами дескрипторов DBI:

err

```
$rv = $h->err;
```

Возвращает *собственный* код ошибки ядра базы данных для последнего вызывавшегося метода драйвера. Обычно код является целым числом, но полагаться на это нельзя.

DBI сбрасывает `$h->err` в `undef` перед большинством вызовов методов DBI, поэтому срок существования этой величины короток. Кроме того, большинство драйверов использует одну переменную ошибки для всех своих дескрипторов, и вызов метода для одного дескриптора обычно сбрасывает ошибку во всех остальных дескрипторах, являющихся потомками этого драйвера.

Если вам необходимо проверять отдельные ошибки *и* ваша программа должна быть переносима на различные типы баз данных, вам нужно выяснить, каковы коды ошибок в каждой конкретной базе данных и проверять их все.

errstr

```
$str = $h->errstr;
```

Возвращает *собственное* сообщение об ошибке ядра базы данных для последнего вызывавшегося метода драйвера. Относительно срока существования справедливо все сказанное выше о методе `err`.

state

```
$str = $h->state;
```

Возвращает код ошибки в стандартном пятисимвольном формате SQLSTATE. Учтите, что особый код успеха `00000` транслируется в `0` (false). Если драйвер не поддерживает SQLSTATE (а таких большинство), то `state` возвращает `S1000` (общая ошибка) для всех ошибок.

trace

```
$h->trace($trace_level);
$h->trace($trace_level, $trace_filename);
```

Трассировка данных DBI может быть включена для любого конкретного дескриптора (и всех его будущих потомков) путем установки уровня трассировки с помощью метода trace.

Для просмотра происходящего лучше всего использовать уровень трассировки 1. Уровень 2 больше подходит для трассировки общего назначения. Уровни 3 и выше (вплоть до 9) лучше использовать для исследования особых проблем, когда необходимо посмотреть, что происходит «внутри» драйвера и DBI.

Вывод результатов трассировки подробен и обычно весьма полезен. Его большая часть форматируется с помощью функции neat, поэтому строки могут быть отредактированы и укорочены.

Изначально вывод результатов трассировки производится на STDERR. Если задать $trace_filename, открывается файл в режиме дописывания, и вся информация (в том числе от других дескрипторов) перенаправляется в этот файл. Последующие вызовы trace без указания $trace_filename не меняют то, куда выводятся данные трассировки. Если значение $trace_filename является неопределенным, то данные трассировки посылаются на STDERR, а предыдущий файл с данными трассировки закрывается.

См. также в описаниях метода DBI->TRACE данные о переменной окружения DBI_TRACE.

trace_msg

```
$h->trace_msg($message_text);
$h->trace_msg($message_text, $min_level);
```

Записывает $message_text в файл трассировки, если включена трассировка для $h или DBI в целом. Можно осуществлять вызов в форме DBI->trace_msg($msg). См. trace.

В случае когда определен $min_level, сообщение выводится, только если уровень трассировки не меньше, чем $min_level. По умолчанию $min_level равен 1.

func

```
$h->func(@func_arguments, $func_name);
```

Метод func можно использовать для вызова частных нестандартных и непереносимых методов, реализованных в драйвере. Обратите внимание, что имя функции указывается в последнем аргументе.

Этот метод не связан напрямую с вызовом хранимых процедур. Вызов хранимых процедур в настоящее время в DBI не определен. Некоторые драйверы, например DBD::Oracle, поддерживают его непереносимым способом.

Подробности ищите в документации по драйверу.

Атрибуты, общие для всех дескрипторов

Эти атрибуты являются общими для всех типов дескрипторов DBI.

Некоторые атрибуты наследуются дескрипторами-потомками. Это означает, что значение наследуемого атрибута во вновь созданном дескрипторе такое же, как значение в родительском дескрипторе базы данных. Изменение значений атрибутов этого нового дескриптора команды не оказывает никакого влияния на родительский дескриптор базы данных, а изменения атрибутов дескриптора базы данных не оказывают влияния на существующие дескрипторы команд, а только на те дескрипторы команд, которые будут порождены им позже.

Попытка установить или получить значение неизвестного атрибута приводит к фатальному результату, если только это не частный атрибут драйвера (имя которого начинается с буквы в нижнем регистре).

Например:

```
$h->{AttributeName} = ...;     # установить/записать
... = $h->{AttributeName};     # получить/прочесть
```

Warn (булев, наследуемый)

Включает выдачу полезных предупреждений при обнаружении каких-либо неудачных приемов программирования. По умолчанию включен. Некоторые слои эмуляции, особенно для интерфейсов Perl 4, отключают вывод предупреждений. Поскольку предупреждения генерируются функцией Perl `warn`, они могут перехватываться с помощью `$SIG{__WARN__}`.

Active (булев, только для чтения)

Имеет значение true, если дескриптор является «активным». Редко используется в приложениях. Точное значение в настоящее время не вполне ясно. Для дескриптора базы данных обычно означает, что дескриптор соединен с базой данных (`$dbh->disconnect` сбрасывает `Active`). Для дескриптора команды обычно означает, что команда представляет собой `SELECT` и в ней могут быть еще данные для выборки. (Выборка всех данных или вызов `$sth->finish` сбрасывает `Active`.)

Kids (целое число, только для чтения)

Для дескриптора драйвера `Kids` является числом существующих в данный момент дескрипторов базы данных, созданных из этого дескрип-

тора драйвера. А для дескриптора базы данных `Kids` – число существующих в данный момент дескрипторов команд, созданных из этого дескриптора базы данных.

ActiveKids (целое число, только для чтения)

Подобен `Kids`, но учитываются только активные (в указанном выше отношении) потомки.

CachedKids (ссылка на хэш)

Для дескриптора базы данных возвращает ссылку на кэш (хэш) дескрипторов команд, созданных методом `prepare_cached`. Для дескриптора драйвера возвращает ссылку на кэш (хэш) дескрипторов баз данных, созданных методом `connect_cached`.

CompatMode (булев, наследуемый)

Используется в слоях эмуляции (например Oraperl) для включения совместимого режима работы соответствующего драйвера (например `DBD::Oracle`) для этого дескриптора. Обычно не устанавливается в коде приложения.

InactiveDestroy (булев)

Этот атрибут можно использовать для отключения связанного с *ядром базы данных* эффекта уничтожения дескриптора (при котором обычно закрывается подготовленная команда, происходит отсоединение от базы данных и т. д.).

Для дескриптора базы данных этот атрибут не отключает *явный* вызов метода disconnect, а только скрытый в DESTROY.

Этот атрибут предназначен для использования в приложениях Unix, которые «разветвляются» на дочерние процессы. Либо родительский, либо порожденный процесс, но не оба одновременно, должны установить `InactiveDestroy` для всех совместно используемых дескрипторов.

PrintError (булев, наследуемый)

Этот атрибут можно использовать для принудительного вывода сообщений об ошибках (с использованием `warn`) в дополнение к обычному возврату кодов ошибок. Когда атрибут установлен, любой метод, приведший к возникновению ошибки, заставит DBI выполнить `warn("$class $method failed: $DBI::errstr")`, где `$class` является драйвером класса, а `$method` – именем вызвавшего ошибку метода.

Например:

```
DBD::Oracle::db prepare failed: ... здесь текст ошибки ...
```

По умолчанию `DBI->connect` устанавливает `PrintError` во включенное состояние.

При желании предупредительные сообщения можно перехватывать и обрабатывать с помощью обработчика `$SIG{__WARN__}` или модулей `CGI::Carp` и `CGI::ErrorWrap`.

RaiseError (булев, наследуемый)

Этот атрибут можно использовать для того, чтобы заставить DBI возбуждать исключительные ситуации, а не просто возвращать код ошибки обычным образом. По умолчанию он «выключен». При «включении» любой метод, вызвавший ошибку, приводит к тому, что DBI вызывает `die("$class $method failed: $DBI::errstr")`, где `$class` является классом драйвера, а `$method` является именем метода, вызвавшего ошибку, например:

```
DBD::Oracle::db prepare failed: ... здесь текст ошибки ...
```

Если при этом включен и `PrintError`, то он обрабатывается перед `RaiseError`, если только не определен обработчик `__DIE__`, в этом случае `PrintError` пропускается, поскольку `die` выведет сообщение об ошибке.

Если требуется временно выключить `RaiseError` (например, внутри библиотечной функции, которая может вызвать ошибку), рекомендуется делать это так:

```
{
  local $h->{RaiseError}; # локализовать и отключить для этого блока
  ...
}
```

Perl автоматически восстановит исходное значение, независимо от того, каким образом произойдет выход из блока. Такой же алгоритм применим и к другим атрибутам, включая `PrintError`.

К сожалению, он не срабатывает в версиях Perl вплоть до 5.004_04. Для обратной совместимости можно вместо этого использовать блок `eval { ... }`.

ChopBlanks (булев, наследуется)

Этот атрибут можно использовать для управления обрезанием хвостовых пробелов в символьных полях фиксированной длины (CHAR). На другие типы полей он не действует, даже если они содержат хвостовые пробелы.

По умолчанию атрибут имеет значение false (хотя значение по умолчанию может изменяться). Приложения, требующие особого поведения, должны устанавливать этот атрибут в нужное им значение. Эмулирующие интерфейсы должны устанавливать этот атрибут в соответствии с поведением интерфейса, который они эмулируют.

Драйверы не обязаны поддерживать этот атрибут, но любой не поддерживающий его драйвер должен возвращать в качестве его значения `undef`.

LongReadLen (целое без знака, наследуется)

Этот атрибут можно использовать для управления максимальной длиной больших полей («blob», «memo» и т. д.), которые драйвер автоматически читает из базы данных во время выборки. Атрибут `LongReadLen` участвует только при выборке и чтении больших полей, на вставку и обновление он не влияет.

Значение `0` определяет, что длинные данные не должны автоматически выбираться (`fetch` должен возвращать `undef` в качестве значения больших полей, когда `LongReadLen` равен `0`).

По умолчанию обычно устанавливается значение `0`, но это зависит от драйвера. Приложения, выбирающие длинные поля, должны устанавливать это значение несколько большим, чем ожидаемый максимальный размер выбираемого поля.

Некоторые базы данных возвращают определенные типы длинных полей как пары шестнадцатеричных цифр. Для этих типов `LongReadLen` относится к размеру данных, а не к удвоенной длине закодированной строки.

Обычно изменение значения `LongReadLen` для дескриптора команды после того, как для него выполнен `prepare`, не оказывает никакого эффекта, поэтому установка `LongReadLen` для `$dbh` обычно производится перед вызовом `prepare`.

Учтите, что задаваемое для этого атрибута значение оказывает непосредственное воздействие на объем памяти, используемой приложением, поэтому не будьте слишком щедры при его установке.

Дополнительные сведения о процедуре усечения полей см. в описании `LongTruncOk`.

LongTruncOk (булев, наследуется)

Этот атрибут можно использовать для управления результатом выборки большого поля при его усечении (обычно в результате того, что оно длиннее, чем `LongReadLen`).

По умолчанию LongTruncOk имеет значение false, что приводит к возникновению ошибки при выборке поля, которое приходится усекать. (Приложения должны всегда проверять наличие ошибок после цикла выборки, чтобы обнаруживать случаи, когда выборка преждевременно заканчивается в результате деления на ноль или усечения длинного поля.)

Если выборка приводит к ошибке из-за усечения длинного поля при LongTruncOk, установленном в false, то многие драйверы позволяют продолжить выборку строк дальше.

См. также LongReadLen.

Taint (булев, наследуется)

Если этот атрибут установлен в истинное значение *и* Perl работает в режиме меченых данных (запущен с опцией -T), то все данные, выбираемые из базы, помечаются, и при вызове большинства методов DBI проверяется наличие помеченных аргументов. *Такое поведение может в будущем измениться.*

По умолчанию этот атрибут сброшен, даже если Perl запущен с опцией -T. Подробнее об этом режиме Perl см. на странице электронного руководства *perlsec*. Если Perl запущен не в режиме меченых данных, этот атрибут не действует.

При выборке данных, которым вы доверяете, можно отключить атрибут taint для дескриптора команды на время работы цикла выборки.

В настоящее время помечаются только данные выборки. Возможно, что в будущих версиях результаты вызовов других методов DBI и значения выбираемых атрибутов также будут помечаться. Такое изменение может нарушить работу ваших приложений, если вы не побеспокоитесь об этом уже сейчас. Если вы используете режим пометки данных в DBI, не поленитесь сообщить о вашем опыте и предложениях для последующих изменений.

private_*

DBI предоставляет способ хранения дополнительных данных в дескрипторах в качестве «частных» атрибутов. DBI разрешает записывать и извлекать значение любого атрибута, имя которого начинается с private_. Настоятельно рекомендуется использовать только *один* закрытый атрибут (например, используйте ссылку на хэш) и давать ему длинное и недвусмысленное имя, включающее имя модуля или приложения, к которому относится этот атрибут, например private_YourFullModuleName_thingy.

Объекты дескрипторов баз данных в DBI

В этом разделе рассказывается о методах и атрибутах, относящихся к дескрипторам баз данных.

Методы дескрипторов баз данных

Для дескрипторов баз данных в DBI определены следующие методы:

do

```
$rc  = $dbh->do($statement)           || die $dbh->errstr;
$rc  = $dbh->do($statement, \%attr)   || die $dbh->errstr;
$rv  = $dbh->do($statement, \%attr, @bind_values) || ...
```

Подготавливает и выполняет одну команду. Возвращает число задействованных строк или `undef` в случае ошибки. Возврат значения `-1` означает, что число строк неизвестно или не применимо.

Обычно полезнее всего использовать этот метод для *не*-`SELECT` команд, которые либо нельзя подготовить заранее в силу ограничений драйвера, либо не требуется выполнять многократно. Его не рекомендуется применять с командами `SELECT`, поскольку он не возвращает дескриптор команды и, следовательно, не позволяет осуществлять выборку данных.

Метод `do`, выполняемый по умолчанию, логически эквивалентен такой последовательности:

```
sub do {
    my($dbh, $statement, $attr, @bind_values) = @_;
    my $sth = $dbh->prepare($statement, $attr) or return undef;
    $sth->execute(@bind_values) or return undef;
    my $rows = $sth->rows;
    ($rows == 0) ? "0E0" : $rows; # возврат true, если нет ошибок
}
```

Например:

```
my $rows_deleted = $dbh->do(q{
    DELETE FROM table
    WHERE status = ?
}, undef, 'DONE') || die $dbh->errstr;
```

Использование заполнителей и `@bind_values` с методом `do` может принести пользу, поскольку позволяет избежать необходимости правильной расстановки кавычек в переменных, входящих в `$statement`. Однако если вы собираетесь выполнять команду неоднократно, то эффективнее будет однажды подготовить команду, а затем выполнять ее требуемое количество раз.

Расстановка кавычек с помощью q{...}, используемая в этом примере, позволяет избежать конфликтов с кавычками, которые могут использоваться в команде SQL. Если вам нужно вставить переменную в строку, используйте оператор двойных кавычек qq{...}. Подробности см. в разделе «Quote and Quote-Like Operators» страницы электронного руководства *perlop*.

selectrow_array

```
@row_ary = $dbh->selectrow_array($statement);
@row_ary = $dbh->selectrow_array($statement, \%attr);
@row_ary = $dbh->selectrow_array($statement, \%attr, @bind_values);
```

Этот вспомогательный метод объединяет в одном вызове prepare, execute и fetchrow_array. При вызове в контексте списка он возвращает первую строку данных из команды. При вызове в скалярном контексте он возвращает первое поле первой строки. Параметр $statement может быть ранее подготовленным дескриптором команды, в этом случае prepare пропускается.

При неудаче вызова какого-либо из методов и не установленном атрибуте RaiseError метод selectrow_array возвращает пустой список.

В скалярном контексте selectrow_array возвращает значение первого поля. Если подходящих строк нет или произошла ошибка, возвращается undef. Поскольку нельзя определить в этом случае, был ли undef возвращен из-за того, что значением первого поля является NULL, вызывать selectrow_array в скалярном контексте следует с осторожностью.

selectall_arrayref

```
$ary_ref = $dbh->selectall_arrayref($statement);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr, @bind_values);
```

Этот вспомогательный метод объединяет в одном вызове prepare, execute и fetchall_arrayref. Он возвращает ссылку на массив, содержащий ссылки на массивы для каждой строки выбранных данных.

Параметр $statement может быть ранее подготовленным дескриптором команды, в этом случае prepare пропускается. Подготовка рекомендуется, если команда должна выполняться многократно.

При ошибке выполнения какого-либо метода, кроме fetch, и неустановленном атрибуте RaiseError метод selectall_arrayref возвращает undef. При ошибке в fetch и неустановленном атрибуте RaiseError происходит возврат с теми данными, которые были выбраны к моменту ошибки.

selectcol_arrayref

```
$ary_ref = $dbh->selectcol_arrayref($statement);
$ary_ref = $dbh->selectcol_arrayref($statement, \%attr);
$ary_ref = $dbh->selectcol_arrayref($statement, \%attr, @bind_values);
```

Этот вспомогательный метод объединяет в одном вызове prepare, execute и выборку одной колонки из всех строк. Он возвращает ссылку на массив, содержащий значения первой колонки из каждой строки.

Параметр $statement может быть ранее подготовленным дескриптором команды, в этом случае prepare пропускается. Подготовка рекомендуется, если команда должна выполняться многократно.

При ошибке выполнения какого-либо метода, кроме fetch, и неустановленном атрибуте RaiseError метод selectcol_arrayref возвращает undef. При ошибке в fetch и неустановленном атрибуте RaiseError происходит возврат с теми данными, которые были выбраны к моменту ошибки.

prepare

```
$sth = $dbh->prepare($statement)          || die $dbh->errstr;
$sth = $dbh->prepare($statement, \%attr)  || die $dbh->errstr;
```

Подготавливает одну команду для выполнения ядром базы данных в будущем и возвращает ссылку на объект дескриптора команды.

Возвращаемый дескриптор команды можно использовать для получения атрибутов команды и вызова метода execute. См. «Методы дескриптора команды».

Драйверы баз данных, не поддерживающие подготовку команд, обычно просто записывают команду в возвращаемый дескриптор и выполняют ее при вызове $sth->execute. Такие драйверы не могут возвращать многие полезные данные о команде, например $sth->{NUM_OFFIELDS}, пока не будет выполнен $sth->execute. Переносимые приложения должны принимать во внимание этот факт.

Обычно драйверы DBI не производят синтаксического анализа команды, кроме простого подсчета количества заполнителей. Команда передается непосредственно ядру базы данных, что иногда называют режимом ретрансляции (pass-thru mode). Это имеет как преимущества, так и недостатки. К преимуществам относится то, что можно полностью использовать функции ядра базы данных. Недостатки состоят в том, что ваши возможности ограничиваются при использовании простого ядра, а при создании приложений, предназначенных для работы с несколькими ядрами баз данных, приходится принимать дополнительные меры предосторожности.

Переносимые приложения не должны рассчитывать на то, что новую команду можно подготовить и/или выполнить, пока не выбраны все результаты предыдущей команды.

Некоторые средства для работы с SQL из командной строки используют символы окончания команды, обычно точку с запятой. Такие символы окончания команды, как правило, не должны использоваться в DBI.

prepare_cached

```
$sth = $dbh->prepare_cached($statement)
$sth = $dbh->prepare_cached($statement, \%attr)
$sth = $dbh->prepare_cached($statement, \%attr, $allow_active)
```

Подобен `prepare`, за исключением того, что возвращаемый дескриптор команды хранится в хэше, связанном с `$dbh`. Если производится новый вызов `prepare_cached` с теми же значениями `$statement` и `%attr`, возвращается соответствующий `$sth` из кэша без обращения к серверу базы данных.

В некоторых приложениях такое кэширование может оказаться полезным, однако оно может также послужить источником проблем, и пользоваться им нужно с осторожностью. Если возвращенный `$sth` является активным, т. е. имеется команда `SELECT`, из которой извлечены не все данные, выдается предупреждение. Подавить выдачу предупреждения можно, установив значение `$allow_active` в «истину». Доступ (и очистка) кэша могут быть осуществлены через атрибут `CachedKids`.

Вот пример возможного использования `prepare_cached`:

```
while ( ($field, $value) = each %search_fields ) {
    push @sql,    "$field = ?";
    push @values, $value;
}
$qualifier = "";
$qualifier = "where ".join(" and ", @sql) if @sql;
$sth = $dbh->prepare_cached("SELECT * FROM table $qualifier");
$sth->execute(@values);
```

commit

```
$rc  = $dbh->commit     || die $dbh->errstr;
```

Фиксирует (делает постоянными) серию последних изменений в базе данных, если она поддерживает транзакции и `AutoCommit` отключен.

Если AutoCommit включен, то вызов commit влечет выдачу предупреждения «commit ineffective with AutoCommit» (commit не действует при включенном AutoCommit).

rollback

```
$rc  = $dbh->rollback   || die $dbh->errstr;
```

Производит откат (аннулирует) последнюю серию незафиксированных в базе данных изменений, если база данных поддерживает транзакции и AutoCommit выключен.

Если AutoCommit включен, то при вызове rollback выводится предупреждение «rollback ineffective with AutoCommit» (rollback не действует при включенном AutoCommit).

disconnect

```
$rc = $dbh->disconnect  || warn $dbh->errstr;
```

Отсоединяет базу данных от дескриптора базы данных. disconnect обычно используется только перед выходом из программы. После отсоединения дескриптор становится бесполезным.

Действие транзакции при явном отсоединении от базы данных, когда AutoCommit выключен, является, к сожалению, неопределенным. Некоторые базы данных, такие как Oracle и Ingres, автоматически фиксируются при всех незавершенных изменениях. Однако в других системах баз данных, таких как Informix, производится откат незавершенных изменений. Поэтому приложения, не использующие AutoCommit, должны *всегда* явным образом вызывать commit() или rollback() перед тем, как вызвать disconnect().

База данных автоматически отсоединяется с помощью метода DESTROY, если не остается ссылок на дескриптор, а соединение сохранено. Метод DESTROY для каждого драйвера должен скрыто вызывать rollback для отмены всех незафиксированных изменений. Это важно для того, чтобы незавершенные транзакции не были зафиксированы только потому, что Perl вызывает DESTROY для всех объектов перед выходом. Не следует также полагаться на порядок удаления объектов при «глобальном удалении», поскольку он не определен.

Обычно, если необходимо произвести фиксацию или откат изменений при отсоединении, следует явным образом вызвать commit или rollback перед тем, как отсоединяться от базы данных.

При отсоединении от базы данных с активными дескрипторами команд выдается предупреждение. Дескрипторы команд должны быть очищены (удалены) перед отсоединением, либо для каждого из них должен быть вызван метод finish.

ping

```
$rc = $dbh->ping;
```

Пытается определить достаточно эффективным способом, работает ли по-прежнему сервер базы данных и работает ли соединение с ним. Отдельные драйверы должны реализовывать эту функцию способом, наиболее подходящим для соответствующего ядра базы данных.

Текущая реализация *по умолчанию* всегда возвращает true, не делая ничего в действительности. На самом деле, она возвращает «`0 but true`», т. е. «истину, но ноль». Благодаря этому можно отличить подлинное возвращенное значение от значения по умолчанию. Драйверы должны переопределять этот метод, чтобы он выполнял действия, соответствующие их типу базы данных.

Не многие приложения могут потребовать использования этого метода. Пример использования можно найти в модуле `Apache::DBI`.

table_info (новый)

```
$sth = $dbh->table_info;
```

Предупреждение: это экспериментальный метод, который может подвергнуться изменениям.

Возвращает активный дескриптор команды, который можно использовать для выборки сведений о таблицах и представлениях, существующих в базе данных.

Дескриптор имеет, как минимум, следующие поля, расположенные в указанном ниже порядке. После этих полей могут присутствовать и другие.

TABLE_CAT

Идентификатор таблицы в каталоге. Это поле содержит NULL (`undef`), если не применимо к источнику данных, как это чаще всего бывает. Поле является пустым, если не применимо к таблице.

TABLE_SCHEM

Имя схемы, содержащей значение TABLE_NAME. Это поле содержит NULL (`undef`), если не применимо к источнику данных, и является пустым, если не применимо к таблице.

TABLE_NAME

Имя таблицы (или представления, синонима и т. д.).

TABLE_TYPE

Имеет одно из следующих значений: «TABLE», «VIEW», «SYSTEM TABLE», «GLOBAL TEMPORARY», «LOCAL TEMPORARY», «ALI-

AS», «SYNONYM» или идентификатор типа, специфический для источника данных.

REMARKS

Описание таблицы. Может быть NULL (`undef`).

Учтите, что `table_info` может возвращать записи не для всех таблиц. Приложения могут использовать любые существующие таблицы независимо от того, возвращает ли `table_info` данные о них. См. также `tables`.

Более подробные данные о полях и их назначении ищите по адресу:

http://msdn.microsoft.com/library/psdk/dasdk/odch6wqb.htm

Если этот URL не работает, используйте поисковые средства MSDN на:

http://search.microsoft.com/us/dev/

и ищите `SQLTables returns` с использованием опции exact phrase. Ссылка, которую вы получите, вероятно, будет называться `SQLTables` и будет указывать на Data Access SDK.

tables (новый)

```
@names = $dbh->tables;
```

Предупреждение: это экспериментальный метод, который может подвергнуться изменениям.

Возвращает список имен таблиц и представлений, возможно, вместе с префиксом схемы. В списке должны быть все таблицы, которые можно использовать в команде `SELECT` без дополнительной квалификации.

Учтите, что `tables` может возвращать записи не для всех таблиц. Приложения могут использовать любые существующие таблицы независимо от того, возвращает ли `tables` данные о них. См. также `table_info`.

type_info_all (новый)

```
$type_info_all = $dbh->type_info_all;
```

Предупреждение: это экспериментальный метод, который может подвергнуться изменениям.

Возвращает ссылку на массив, содержащий сведения о каждом варианте типа данных, поддерживаемого базой данных и драйвером. Массив и его содержимое предназначены только для чтения.

Первым элементом является ссылка на хэш пар `Name => Index`. Последующие элементы являются ссылками на массивы, по одному для каждого типа данных. В первом хэше определены имена и порядок полей в последующем списке массивов, например:

```
$type_info_all = [
  {   TYPE_NAME         => 0,
      DATA_TYPE         => 1,
      COLUMN_SIZE       => 2,      # первоначально был PRECISION
      LITERAL_PREFIX    => 3,
      LITERAL_SUFFIX    => 4,
      CREATE_PARAMS     => 5,
      NULLABLE          => 6,
      CASE_SENSITIVE    => 7,
      SEARCHABLE        => 8,
      UNSIGNED_ATTRIBUTE=> 9,
      FIXED_PREC_SCALE  => 10,     # первоначально был MONEY
      AUTO_UNIQUE_VALUE => 11,     # первоначально был AUTO_INCREMENT
      LOCAL_TYPE_NAME   => 12,
      MINIMUM_SCALE     => 13,
      MAXIMUM_SCALE     => 14,
      NUM_PREC_RADIX    => 15,
  },
  [ 'VARCHAR', SQL_VARCHAR,
      undef, "'","'", undef,0,  1,1,0,0,0,undef,1,255, undef
  ],
  [ 'INTEGER', SQL_INTEGER,
      undef,  "", "", undef,0,  0,1,0,0,0,undef,0,  0, 10
  ],
];
```

Обратите внимание, что несколько строк могут иметь одинаковое значение в поле DATA_TYPE, если имя типа можно записывать разными способами и/или есть разновидности типа с разными атрибутами (например, с установленным или нет AUTO_UNIQUE_VALUE, с установленным или нет UNSIGNED_ATTRIBUTE и т. д.).

Строки упорядочены сначала по DATA_TYPE, а затем по степени близости к соответствующему типу ODBC SQL.

Смысл полей описывается в документации по методу type_info. Приведенные значения индексов (например NULLABLE => 6) служат только для иллюстрации. Драйверы могут определять поля в другом порядке.

Обычно этот метод не используется непосредственно. Метод type_info предоставляет более полезный интерфейс к данным.

type_info (новый)

```
@type_info = $dbh->type_info($data_type);
```

Предупреждение: это экспериментальный метод, который может подвергнуться изменениям.

Возвращает список ссылок на хэши, содержащие сведения об одном или более вариантах $datatype. Список упорядочен сначала по DATA_TYPE, а затем по степени близости к соответствующему типу ODBC SQL.

При вызове в скалярном контексте возвращается только первый (лучший) элемент.

Если $data_type не определен или равен SQL_ALL_TYPES, то в списке будут содержаться хэши для всех разновидностей типов данных, поддерживаемых базой данных и драйвером.

Если $data_type является ссылкой на массив, то type_info возвращает данные для *первого* типа в массиве, которому найдено любое соответствие.

Ключи хэша следуют тем же соглашениям по регистру букв, что и весь DBI (см. «Соглашения по именам и пространство имен»). Должны существовать следующие элементы:

TYPE_NAME (строка)

Имя типа данных для использования в CREATE TABLE и других командах.

DATA_TYPE (целое число)

Номер типа данных SQL.

COLUMN_SIZE (целое число)

Для числовых типов данных это либо общее количество цифр (если NUM_PREC_RADIX равно 10), либо общее количество битов, допустимых в колонке (если NUM_PREC_RADIX равно 2).

Для строковых типов это максимальное количество байтов в строке.

Для типов даты и интервала это максимальное количество символов, необходимое для вывода значения.

LITERAL_PREFIX (строка)

Символы, используемые в качестве префикса для литералов. Обычно префиксом является «'» (одинарная кавычка) для символов или «0x» для двоичных данных, передаваемых в шестнадцатеричном виде. Для типов, к которым это не применимо, возвращается NULL (undef).

LITERAL_SUFFIX (строка)

Символы, используемые в качестве суффикса для литералов. Для символов это обычно «'» (одинарная кавычка). Для типов, к которым это не применимо, возвращается NULL (undef).

CREATE_PARAMS (строка)

Параметры определения типа данных. Например, CREATE_PARAMS для DECIMAL будет «precision,scale», если тип DECIMAL должен быть объявлен как DECIMAL(*precision,scale*), где *precision* и *scale* являются целыми числами. Для VARCHAR это «max length». Для типов, к которым это не применимо, возвращается NULL (undef).

NULLABLE (целое число)

Указывает на то, может ли тип данных принимать значение NULL: `0` = нет, `1` = да, `2` = неизвестно.

CASE_SENSITIVE (булев)

Указывает на то, является ли тип данных чувствительным к регистру при сортировке и сравнении.

SEARCHABLE (целое число)

Указывает на то, как тип данных может использоваться в предложении `WHERE`:

0 Не может использоваться в предложении `WHERE`.

1 Только в предикате `LIKE`.

2 Во всех операторах сравнения, кроме `LIKE`.

3 Может использоваться в предложении `WHERE` с любыми операторами сравнения.

UNSIGNED_ATTRIBUTE (булев)

Указывает на то, может ли тип данных быть беззнаковым. Для типов, к которым это не применимо, возвращается NULL (`undef`).

FIXED_PREC_SCALE (булев)

Указывает на то, всегда ли тип данных имеет одинаковые точность и масштаб (например, для денежного типа). Для типов, к которым это не применимо, возвращается NULL (`undef`).

AUTO_UNIQUE_VALUE (булев)

Указывает на то, должна ли колонка этого типа автоматически получать уникальное значение при вставке строки. Для типов, к которым это не применимо, возвращается NULL (`undef`).

LOCAL_TYPE_NAME (строка)

Локализованная версия `TYPE_NAME` для использования в диалоге с пользователем. Если локализованное имя недоступно, возвращается NULL (`undef`) (в этом случае нужно использовать `TYPE_NAME`).

MINIMUM_SCALE (целое)

Минимальный масштаб типа данных. Если тип данных имеет фиксированный масштаб, то в `MAXIMUM_SCALE` содержится то же самое значение. Для типов, к которым это не применимо, возвращается NULL (`undef`).

MAXIMUM_SCALE (целое)

Максимальный масштаб типа данных. Если тип данных имеет фиксированный масштаб, то в `MINIMUM_SCALE` содержится то же самое значение. Для типов, к которым это не применимо, возвращается NULL (`undef`).

SQL_DATA_TYPE (целое)

Эта колонка такая же, как DATA_TYPE, за исключением типов данных даты и интервала. Для типов данных интервала и даты поле SQL_DATA_TYPE возвращает SQL_INTERVAL или SQL_DATETIME, а поле SQL_DATETIME_SUB (см. ниже) возвращает субкод конкретного типа данных интервала или даты. Если это поле NULL, значит, драйвер не поддерживает либо не сообщает о субтипах интервала или даты.

SQL_DATETIME_SUB (целое)

Для типов данных интервала или даты, в которых поле SQL_DATA_TYPE содержит SQL_INTERVAL или SQL_DATETIME, в этом поле содержится субкод конкретного типа данных интервала либо даты, в противном случае имеет значение NULL (undef).

NUM_PREC_RADIX (целое)

Значение основания системы исчисления для типа данных. Для приблизительных числовых типов NUM_PREC_RADIX содержит 2, а COLUMN_SIZE содержит количество битов. Для точных числовых типов NUM_PREC_RADIX содержит значение 10, а COLUMN_SIZE содержит количество десятичных цифр. Для типов, к которым это не применимо, или если драйвер не может сообщить эти сведения, возвращается NULL (undef).

INTERVAL_PRECISION (целое)

Директивная точность интервала для интервальных типов. Для типов, к которым это не применимо, или в случае, если драйвер не может сообщить эти сведения, возвращается NULL (undef).

Поскольку драйверы DBI и ODBC по-разному отображают свои типы в стандартные типы ISO, может потребоваться поиск в более чем одном типе. Вот пример поиска типа, пригодного для хранения даты:

```
$my_date_type = $dbh->type_info( [ SQL_DATE, SQL_TIMESTAMP ] );
```

Аналогично, чтобы с большей надежностью находить тип для хранения данных small integer, можно использовать список, состоящий из SQL_SMALLINT, SQL_INTEGER, SQL_DECIMAL и т. д.

Более подробные данные об этих полях и их назначении смотрите на

http://msdn.microsoft.com/library/psdk/dasdk/ odch6yy7.htm

Если этот URL не работает, используйте поисковые средства MSDN на:

http://search.microsoft.com/us/dev/

и ищите SQLGetTypeInfo returns с использованием опции exact phrase. Ссылка, которую вы получите, вероятно, будет называться SQLGetTypeInfo (их может оказаться несколько).

Отдельные типы данных описаны здесь:

http://msdn.microsoft.com/library/psdk/dasdk/odap8fcj.htm

Если этот URL не работает, используйте поисковые средства MSDN, как описано выше, и ищите `SQL Data Types`.

quote

```
$sql = $dbh->quote($value);
$sql = $dbh->quote($value, $data_type);
```

Расставляет кавычки в строковом литерале для использования в качестве литерала в команде SQL, заменяя специальные символы, содержащиеся в строке (например кавычки), и добавляя внешние кавычки нужного типа.

```
$sql = sprintf "SELECT foo FROM bar WHERE baz = %s",
               $dbh->quote("Don't");
```

Для большинства баз данных функция `quote` должна вернуть `'Don''t'` (вместе с внешними кавычками).

Для неопределенного значения `$value` возвращается строка `NULL` (без кавычек), что соответствует представлению NULL в SQL.

Если задан аргумент `$data_type`, то он используется при попытке определить правила расстановки кавычек с помощью данных, возвращаемых `type_info`. Особым случаем являются стандартные числовые типы, они оптимизированы для возврата `$value` без обращения к `type_info`.

Функция `quote` может не справиться с произвольными входными данными (например, двоичными или содержащими символы перевода строки) и никак не связана с преобразованием метасимволов оболочки. Обрабатывать значения, используемые в заполнителях и связанных величинах, не нужно.

Атрибуты дескриптора базы данных

В этом разделе описаны атрибуты, специфичные для дескрипторов баз данных.

Изменения в этих атрибутах дескрипторов баз данных не влияют на уже существующие или создаваемые впоследствии дескрипторы баз данных.

Попытка установить или получить значение неизвестного атрибута приводит к фатальному результату, если только это не частный атрибут, специфический для драйвера (у всех таких атрибутов имена начинаются с буквы в нижнем регистре). Например:

```
$h->{AutoCommit} = ...;       # установить/записать
... = $h->{AutoCommit};       # получить/прочитать
```

AutoCommit (булев)

Если он установлен, то в базе данных нельзя произвести откат изменений. Если его значение равно false, то изменения в базе данных автоматически происходят в «транзакции», которую нужно зафиксировать или откатить с помощью методов `commit` или `rollback`.

По умолчанию драйверы должны всегда устанавливать режим `AutoCommit` (неудачный выбор для DBI, на который в значительной мере повлияли соглашения, принятые в ODBC и JDBC.)

Попытка установить для `AutoCommit` не поддерживаемое значение приводит к фатальной ошибке. Это важная особенность DBI. Приложениям, требующим полной поддержки транзакций, можно устанавливать `$dbh->{AutoCommit} = 0` (или устанавливать `AutoCommit` равным `0` через `connect`) без необходимости проверки того, что значение успешно присвоено.

Чтобы описать действие этого атрибута, можно разделить базы данных на три категории:

Базы данных, вообще не поддерживающие транзакции.

Базы данных, в которых транзакции действуют постоянно.

Базы данных, в которых транзакции должны запускаться явным образом (`'BEGIN WORK'`).

Базы данных, вообще не поддерживающие транзакции

Для этих баз данных попытка сбросить `AutoCommit` приводит к фатальной ошибке. `commit` и `rollback` выдают предупреждения о том, что они не действуют, когда включен `AutoCommit`.

Базы данных, в которых транзакции действуют постоянно

Обычно это основные коммерческие реляционные базы данных, в которых транзакции реализованы согласно стандарту ANSI. Если `AutoCommit` выключен, то изменения в базе данных становятся постоянными только после выполнения вызова `commit` (см. также описание `disconnect`). При вызове `rollback` все изменения, произведенные вслед за последним `commit`, отменяются.

Если `AutoCommit` включен, то это оказывает такое же действие, как если бы DBI автоматически вызывал `commit` после каждой успешной операции с базой данных. Иными словами, явный вызов `commit` или `rollback` при включенном `AutoCommit` не действует, поскольку изменения уже зафиксированы.

Изменение `AutoCommit` из выключенного состояния во включенное в большинстве драйверов приводит к вызову `commit`.

Изменение `AutoCommit` из включенного состояния в выключенное не должно оказывать немедленного воздействия.

Для баз данных, не поддерживающих режим автоматической фиксации, драйвер должен автоматически фиксировать каждую команду путем явного вызова commit после ее успешного завершения (и производить откат путем явного вызова rollback при неудачном выполнении команды). Передаваемая приложению информация об ошибке соответствует выполненной команде, если только при ее удачном выполнении не произошло ошибки при вызове commit или rollback.

Базы данных, в которых транзакции должны запускаться явным образом

Для этих баз данных DBI стремится обеспечить такое же поведение, как для баз данных, в которых транзакции действуют постоянно (как описано выше).

Для этого драйвер DBI автоматически начинает транзакцию, когда AutoCommit сбрасывается (из состояния по умолчанию «включен») и автоматически начинает новую транзакцию после выполнения commit или rollback. Благодаря этому приложению не приходится отдельно учитывать работу с такими базами данных.

См. важные дополнительные замечания о транзакциях в описании disconnect.

Driver (дескриптор)

Содержит дескриптор родительского драйвера. Рекомендуется использовать только для определения имени драйвера:

```
$dbh->{Driver}->{Name}
```

Name (строка)

Содержит «имя» базы данных. Обычно (рекомендуемое значение) совпадает со строкой «dbi:DriverName:...», используемой для соединения с базой данных, но с удалением ведущих символов dbi:DriverName:.

RowCacheSize (целое) (новый)

Подсказка драйверу, указывающая желательный для приложения размер локального кэша строк, используемого драйвером в последующих командах SELECT. Если кэш строк не реализован, установка значения RowCacheSize игнорируется, а чтение возвращает значение undef.

Значения RowCacheSize имеют следующий смысл:

0 Автоматически определять разумный размер кэша для каждой команды SELECT.

1 Отключить локальный кэш строк.

>1 Кэшировать заданное количество строк.

<0 Кэшировать для каждой команды SELECT столько строк, сколько помещается в памяти.

Учтите, что большой размер кэша может потребовать очень много памяти (количество кэшированных строк умножается на максимальный размер строки). Кроме того, при большом кэше потребуется большая задержка не только при первой выборке, но и при каждой новой загрузке кэша.

См. также атрибут RowsInCache дескриптора команды.

Объекты DBI дескрипторов команд

В этом разделе перечисляются методы и атрибуты дескрипторов команд DBI.

Методы дескрипторов команд

В DBI определены следующие методы, используемые с дескрипторами команд:

bind_param

```
$rc = $sth->bind_param($p_num, $bind_value)  || die $sth->errstr;
$rv = $sth->bind_param($p_num, $bind_value, \%attr)     || ...
$rv = $sth->bind_param($p_num, $bind_value, $bind_type) || ...
```

Метод bind_param можно использовать для связывания значения с заполнителем в подготовленной команде. Заполнители обозначаются символом «вопросительный знак» (?). Например:

```
$dbh->{RaiseError} = 1;          # избежать необходимости проверок после
каждого вызова
$sth = $dbh->prepare("SELECT name, age FROM people WHERE name LIKE ?");
$sth->bind_param(1, "John%");  # заполнители нумеруются, начиная с 1
$sth->execute;
DBI::dump_results($sth);
```

Обратите внимание, что ? не заключается в кавычки, даже если заполнитель представляет собой строку. Некоторые драйверы баз данных помимо ? распознают заполнители в виде *:name* или *:n* (:1, :2 и т. д.), но при этом не гарантирована переносимость. Для указания значений null используются неопределенные связанные значения или undef.

Некоторые драйверы не поддерживают заполнители.

В большинстве драйверов нельзя использовать заполнители для любого элемента команды, поскольку это не позволило бы серверу базы данных проверить команду и создать план ее выполнения. Например:

```
"SELECT name, age FROM ?"          # неверно (может возникнуть ошибка)
"SELECT name, ?   FROM people"     # неверно (хотя ошибки не будет)
```

Кроме того, заполнители могут представлять только отдельные скалярные величины. Например, следующая команда не даст ожидаемого результата, если будет передано более одного значения:

```
"SELECT name, age FROM people WHERE name IN (?)"    # неверно
```

Типы данных для заполнителей

Для указания того, какой тип данных должен иметь заполнитель, можно использовать параметр \%attr. Обычно драйверу нужно знать только то, должен ли заполнитель быть связан со строкой или числом, например:

```
$sth->bind_param(1, $value, { TYPE => SQL_INTEGER });
```

Краткой записью этого частого случая является непосредственная передача типа данных вместо ссылки на хэш \%attr. Следующая строка эквивалентна предыдущей:

```
$sth->bind_param(1, $value, SQL_INTEGER);
```

Значение TYPE указывает на стандартный (не специфический для драйвера) тип параметра. Для указания специфического для драйвера типа он может поддерживать специфический атрибут, например { ora_type => 97 }. Тип данных заполнителя нельзя менять после первого обращения к bindparam. Однако его можно не указывать, в результате чего он возвращается к предшествующему значению.

В Perl из типов данных есть только строка и числовой скаляр. Все типы данных, не являющиеся числами, связываются как строки и должны иметь формат, воспринимаемый базой данных.

Альтернативой заданию типа данных при вызове bind_param является передача значения драйвером как типа, установленного по умолчанию (VARCHAR). Для преобразования типа в команде можно использовать функции SQL, например:

```
INSERT INTO price(code, price) VALUES (?, CONVERT(MONEY,?))
```

Функция CONVERT использована здесь только в качестве примера. Реальная функция и ее синтаксис могут быть различными в разных базах данных и поэтому код не является переносимым.

Дополнительные сведения можно найти в разделе «Заполнители и связанные значения».

bind_param_inout

```
$rc = $sth->bind_param_inout($p_num, \$bind_value, $max_len)  || die $sth->
errstr;
$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, \%attr)     ||
...
$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, $bind_type) ||
...
```

Этот метод действует подобно bind_param, но позволяет команде обновлять значения. Команда обычно является вызовом хранимой процедуры. $bind_value нужно передавать в виде ссылки на фактически используемое значение.

Обратите внимание, что, в отличие от bind_param, переменная $bind_value не считывается при вызове bind_param_inout. Значение переменной считывается в тот момент, когда вызывается execute.

Дополнительный параметр $max_len задает минимальное количество памяти, выделяемое для нового значения $bind_value. Если величина, возвращаемая базой данных, слишком велика, то выполнение приводит к ошибке. При неуверенности в том, какое значение использовать, выберите значение побольше, то есть свыше самого длинного значения, которое может быть возвращено. Издержками использования слишком большого значения будет только непроизводительный расход памяти.

Вероятно, немногие драйверы будут использовать этот метод. Единственный известный в настоящее время драйвер, который его реализует, это DBD::Oracle (DBD::ODBC, возможно, будет осуществлять поддержку в следующей версии). Поэтому данный метод не следует использовать в приложениях, которые не должны зависеть от базы данных.

Для задания значений null используются неопределенные величины или undef. Дополнительные сведения можно найти в разделе «Заполнители и связанные значения».

execute

```
$rv = $sth->execute                || die $sth->errstr;
$rv = $sth->execute(@bind_values)  || die $sth->errstr;
```

Осуществляет действия, необходимые для выполнения подготовленной команды. При возникновении ошибки возвращается значение undef. При успешном выполнении execute всегда возвращает true вне зависимости от числа задействованных строк, даже если оно нулевое (см. ниже). Всегда важно проверять значение, возвращаемое execute (и большинством других методов DBI) на предмет наличия ошибок.

Для *не*-SELECT команды метод execute возвращает число задействованных строк, если оно известно. Если ни одна строка не была изменена, execute возвращает 0E0, что Perl воспринимает как 0 и одновременно true. Заметьте, что если ни одна строка не была изменена командой, то это не ошибка. Если число измененных строк неизвестно, execute возвращает -1.

Для команд SELECT метод execute просто «запускает» запрос в ядре базы данных. Для извлечения данных после вызова execute используйте один из методов выборки. Метод execute *не* возвращает число строк, которые вернет запрос (поскольку большинство баз данных не может указать его заранее), он возвращает просто true.

При задании аргументов метод execute, прежде чем выполнять команду, вызывает для каждого значения bind_param. Значения, связанные таким способом, обычно рассматриваются как имеющие тип SQL_VARCHAR, если только драйвер не сможет определить истинный тип (что бывает редко), или если для указания типа ранее уже не использовались bind_param (или bind_param_inout).

fetchrow_arrayref

```
$ary_ref = $sth->fetchrow_arrayref;
$ary_ref = $sth->fetch;    # псевдоним
```

Осуществляет выборку очередной строки данных и возвращает ссылку на массив, содержащий значения полей. Поля со значением null возвращаются в массиве в виде значений undef. Это самый быстрый способ выборки данных, особенно в сочетании с $sth->bind_columns.

Если строк больше нет или происходит ошибка, fetchrow_arrayref возвращает undef. Впоследствии следует проверить значение $sth->err (или использовать атрибут RaiseError), чтобы определить, не было ли значение undef возвращено вследствие ошибки.

В текущей версии при каждой выборке возвращается одна и та же ссылка на массив. Не следует запоминать ссылку и использовать ее после каждой последующей выборки.

fetchrow_array

```
@ary = $sth->fetchrow_array;
```

Альтернативный для fetchrow_arrayref метод. Осуществляет выборку очередной строки данных и возвращает ее в виде списка, содержащего значения полей. Поля со значением null возвращаются в списке в виде значений undef.

Если строк больше нет или происходит ошибка, fetchrow_array возвращает пустой список. Впоследствии следует проверить значение $sth->

err (или использовать атрибут RaiseError), чтобы определить, не был ли пустой список возвращен вследствие ошибки.

В скалярном контексте fetchrow_array возвращает значение первого поля. Если строк больше нет или происходит ошибка, возвращается undef. Поскольку нельзя отличить undef, возвращенный из-за того, что значением первого поля был NULL, от undef, возвращенного вследствие ошибки, следует проявлять осторожность при использовании fetchrow_array в скалярном контексте.

fetchrow_hashref

```
$hash_ref = $sth->fetchrow_hashref;
$hash_ref = $sth->fetchrow_hashref($name);
```

Альтернативный для fetchrow_arrayref метод. Осуществляет выборку очередной строки данных и возвращает ее в виде ссылки на хэш, содержащий пары имя поля/значение поля. Поля со значением null возвращаются в хэше в виде значений undef.

Если строк больше нет или происходит ошибка, fetchrow_hashref возвращает undef. Впоследствии следует проверить значение $sth->err (или использовать атрибут RaiseError), чтобы определить, не было ли undef возвращено вследствие ошибки.

Необязательный параметр $name задает имя атрибута дескриптора команды. В силу исторических причин его значение по умолчанию NAME; однако для реализации переносимости рекомендуется использовать NAME_lc или NAME_uc.

Ключами хэша являются имена, возвращаемые $sth->{$name}. Если есть несколько полей с одинаковым именем, то в возвращаемом хэше окажется для них только один элемент.

Из-за дополнительной работы, которую приходится делать fetchrow_hashref и Perl, использование этого метода не столь эффективно, как fetchrow_arrayref или fetchrow_array.

В текущей версии для каждой строки возвращается новая ссылка на хэш. *В будущем это изменится*, и каждый раз будет возвращаться одна и та же ссылка, поэтому не полагайтесь на текущий характер выполнения.

fetchall_arrayref

```
$tbl_ary_ref = $sth->fetchall_arrayref;
$tbl_ary_ref = $sth->fetchall_arrayref( $slice_array_ref );
$tbl_ary_ref = $sth->fetchall_arrayref( $slice_hash_ref  );
```

Метод fetchall_arrayref можно использовать для того, чтобы осуществить выборку всех строк из подготовленного и выполненного зап-

роса. Он возвращает ссылку на массив, содержащий ссылки для каждой из строк.

Если возвращаемых строк нет, fetchall_arrayref возвращает ссылку на пустой массив. При возникновении ошибки в fetchall_arrayref происходит возврат с теми данными, которые были выбраны к моменту ошибки, – возможно, отсутствующими. Впоследствии следует проверить значение $sth->err (или использовать атрибут RaiseError), чтобы определить, переданы данные полностью или частично вследствие ошибки.

При передаче ссылки на массив fetchall_arrayref использует fetchrow_arrayref для выборки каждой строки в качестве ссылки на массив. Если передаваемый массив не пуст, то используется как срез для передачи отдельных колонок по номерам.

Без параметров fetchall_arrayref действует так, как если бы ему был передан пустой массив.

При передаче ссылки на хэш fetchall_arrayref использует fetchrow_hashref для выборки каждой строки как ссылки на хэш. Если хэш не пуст, он используется как срез для выборки отдельных колонок по имени. Имена должны быть заданы в нижнем регистре независимо от регистра, возвращаемого в $sth->{NAME}. Значения хэша должны быть установлены в 1.

Например, извлечь только первую колонку каждой строки:

```
$tbl_ary_ref = $sth->fetchall_arrayref([0]);
```

Извлечь предпоследнюю и последнюю колонки каждой строки:

```
$tbl_ary_ref = $sth->fetchall_arrayref([-2,-1]);
```

Извлечь только поля «foo» и «bar» из каждой строки:

```
$tbl_ary_ref = $sth->fetchall_arrayref({ foo=>1, bar=>1 });
```

В первых двух примерах возвращается ссылка на массив ссылок на массив. В последнем примере возвращается ссылка на массив ссылок на хэш.

finish

```
$rc  = $sth->finish;
```

Указывает на то, что из этого дескриптора команды не будет больше делаться выборка и он будет выполнен заново или разрушен. Потребность в методе finish возникает редко, но иногда он может оказаться полезным в особых ситуациях, когда нужно позволить серверу освободить ресурсы (например, буферы сортировки).

После выборки из команды SELECT всех данных драйвер автоматически вызывает finish, поэтому обычно нет надобности делать это явным образом.

Рассмотрим такой запрос:

```
SELECT foo FROM table WHERE bar=? ORDER BY foo
```

Требуется выбрать только первое (самое маленькое) значение «foo» из очень большой таблицы. При выполнении запроса сервер будет вынужден использовать временное буферное пространство для хранения отсортированных строк. Если после выполнения дескриптора и выборки одной строки дескриптор некоторое время не будет выполнен заново и не будет разрушен, то можно использовать метод finish, чтобы сообщить серверу, что буферное пространство можно освободить.

При вызове finish сбрасывается атрибут Active команды. При этом некоторые атрибуты команды, например NAME и TYPE, могут сделаться недоступны, если к ним еще не осуществлялся доступ, и поэтому они не кэшированы.

Метод finish не оказывает влияния на статус транзакции в соединении с базой данных. Он не имеет отношения к выполнению транзакций. Это, в основном, вспомогательный внутренний метод, применение которого редко требуется. Нет необходимости вызывать finish, если вы собираетесь разрушить дескриптор команды или выполнить его снова. См. также описание disconnect и атрибута Active.

rows

```
$rv = $sth->rows;
```

Возвращает число строк, обработанных последней командой, воздействующей на строки, или -1, если число строк неизвестно или недоступно.

Обычно полагаться на счетчик строк можно только при выполнении команд *не*-SELECT-типа (для таких операций, как UPDATE и DELETE), либо после выборки всех строк команды SELECT.

Для команды SELECT обычно невозможно узнать, сколько строк она возвращает, пока не будут выбраны все строки. Некоторые драйверы возвращают число строк, выбранных к данному моменту, а другие могут возвращать -1, пока не будут выбраны все строки. Поэтому использовать методы rows или $DBI::rows с командами типа SELECT не рекомендуется.

Один из альтернативных способов получения числа строк для SELECT заключается в том, чтобы выполнить команду SQL «SELECT COUNT(*) FROM ...» с таким же условием «...», как в вашем запросе, и получить из нее число строк.

bind_col

```
$rc = $sth->bind_col($column_number, \$var_to_bind);
```

Привязывает выходную колонку (поле) команды SELECT к переменной Perl. Пример можно видеть в описании bind_columns. Заметьте, что нумерация колонок начинается с 1.

Когда производится выборка строки из базы данных, автоматически обновляется соответствующая переменная Perl. Нет необходимости выбирать и присваивать значения вручную. Привязка осуществляется в Perl на самом низком уровне с использованием псевдонимов, поэтому дополнительного копирования не происходит. Благодаря этому использование связанных переменных очень эффективно.

Для максимальной переносимости между драйверами bind_col нужно вызывать после execute. В будущих версиях DBI это ограничение может быть снято.

Для того чтобы осуществлять выборку данных, необязательно привязывать выходные колонки, но это полезно делать в некоторых приложениях, требующих максимальной производительности или большей ясности кода. Метод bind_param осуществляет аналогичную, но противоположную функцию для входных переменных.

bind_columns

```
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);
```

Вызывает bind_col для каждой колонки в команде SELECT. Метод bind_columns приводит к аварийному выходу из программы, если число ссылок не соответствует числу полей.

Для максимальной переносимости программы между драйверами метод bind_columns должен использоваться после вызова execute.

Например:

```
$dbh->{RaiseError} = 1; # Выполнить или проверить каждый вызов на наличие
ошибок
$sth = $dbh->prepare(q{ SELECT region, sales FROM sales_by_region });
$sth->execute;
my ($region, $sales);

# Связать переменные Perl с колонками:
$rv = $sth->bind_columns(\$region, \$sales);

# Можно также использовать синтаксис Perl \(...)
# (см.документы perlref):
#     $sth->bind_columns(\($region, $sales));

# Связывание колонок - наиболее эффективный способ выборки данных
```

```
while ($sth->fetch) {
    print "$region: $sales\n";
}
```

Для обеспечения совместимости со старыми сценариями первый параметр игнорируется, если он является undef или ссылкой на хэш.

dump_results

```
$rows = $sth->dump_results($maxlen, $lsep, $fsep, $fh);
```

Делает выборку всех строк из $sth, вызывает DBI::neat_list для каждой строки и выводит результаты в $fh (по умолчанию STDOUT) с разделителем $lsep (по умолчанию "\n"). $fsep имеет значение по умолчанию ", ", а $maxlen по умолчанию равно 35.

Этот метод предназначен для использования в качестве удобной утилиты при создании прототипов и тестировании запросов. Поскольку строки в нем форматируются и редактируются для чтения людьми с помощью neat_list, не рекомендуется применять его для передачи данных в приложениях.

Атрибуты дескрипторов команд

В этом разделе описываются атрибуты, специфические для дескрипторов команд. Большинство их предназначено только для чтения.

Изменения атрибутов дескрипторов команд не оказывают влияния на уже существующие или создаваемые в будущем дескрипторы команд.

Попытка установить или получить значение неизвестного атрибута приводит к фатальному результату, если только это не частный атрибут драйвера (имя которого начинается с буквы в нижнем регистре).

Например:

```
... = $h->{NUM_OF_FIELDS};    # получить/прочитать
```

Обратите внимание, что некоторые драйверы не могут обеспечить верные значения всех или части атрибутов до того, как будет сделан вызов $sth->execute.

См. также описание метода finish, в котором говорится о его воздействии на некоторые атрибуты.

NUM_OF_FIELDS (целое число, только для чтения)

Число полей (колонок), которое возвратит подготовленная команда. Для команд, отличных от SELECT, возвращается значение 0.

NUM_OF_PARAMS (целое число, только для чтения)

Число параметров (заполнителей) в подготовленной команде. Подробности см. в разделе «Переменные подстановки» далее в этом приложении.

NAME (ссылка на массив, только для чтения)

Возвращает ссылку на массив имен полей во всех колонках. Имена могут содержать пробелы, но не должны усекаться или иметь пробелы в конце. Обратите внимание, что имена полей имеют тот регистр (верхний, нижний, смешанный), который возвращает используемый драйвер. В переносимых приложениях нужно использовать `NAME_lc` или `NAME_uc`. Например:

```
print "Имя первой колонки: $sth->{NAME}->[0]\n";
```

NAME_lc (ссылка на массив, только для чтения)

Подобен `NAME`, но всегда возвращает имена в нижнем регистре.

NAME_uc (ссылка на массив, только для чтения)

Подобен `NAME`, но всегда возвращает имена в верхнем регистре.

TYPE (ссылка на массив, только для чтения) (новый)

Возвращает ссылку на массив целых чисел для каждой из колонок. Значение указывает на тип данных в соответствующей колонке.

Значения соответствуют международным стандартам (ANSI X3.135 и ISO/IEC 9075), что, в целом, согласуется с ODBC. Для специфических типов данных драйверов, не соответствующих стандартным типам, обычно должны возвращаться те же значения, что и в драйвере ODBC, поставляемом производителем базы данных. Это могут быть частные типы данных в диапазонах, официально зарегистрированных поставщиком.

Дополнительные сведения можно получить на:

ftp://jerry.ece.umassd.edu/isowg3/dbl/SQL_Registry

Если не существует предоставляемого поставщиком базы данных драйвера ODBC, драйвер DBI может использовать номера типов в диапазоне, официально зарегистрированном для использования в DBI: от −9999 до −9000.

Для всех возможных значений TYPE должна иметься хотя бы одна запись в выдаче метода type_info_all (см. этот метод).

PRECISION (ссылка на массив, только для чтения) (новый)

Возвращает ссылку на массив целых чисел для каждой из колонок. Для нечисловых колонок значение числа обычно характеризует максимальную или заданную ширину колонки. Для числовых колонок значения относятся к максимальному числу значащих цифр, используемых в типе данных (без учета знака и десятичной точки). Обратите внимание, что для типов данных с плавающей запятой (REAL, FLOAT, DOUBLE) превышение значения над точностью может быть до семи знаков из-за присоединения знака, десятичной точки, буквы «E», знака и двух или трех цифр показателя степени.

SCALE (ссылка на массив, только для чтения) (новый)

Возвращает ссылку на массив целых чисел для каждой из колонок. Значения NULL (undef) обозначают колонки, для которых масштаб неприменим.

NULLABLE (ссылка на массив, только для чтения)

Возвращает ссылку на массив чисел, указывающих на возможность содержать NULL в колонке. Возможными значениями являются: 0 = нет, 1 = да, 2 = неизвестно. Например:

```
print "Первая колонка может возвращать NULL\n" if $sth->{NULLABLE}->[0];
```

CursorName (строка, только для чтения)

Возвращает имя курсора, связанного с дескриптором команды, если это возможно. Если оно недоступно или база данных не поддерживает синтаксис SQL "where current of ...", возвращается undef.

Statement (строка, только для чтения) (новый)

Возвращает строку команды, переданную методу prepare.

RowsInCache (целое число, только для чтения)

Если драйвер поддерживает локальный кэш строк для команд SELECT, то в этом атрибуте содержится число невыбранных строк в кэше. Если не поддерживает, то возвращается undef. Учтите, что одни драйверы

производят предварительную выборку строк при выполнении команды, а другие ожидают первой выборки.

См. также описание атрибута `RowCacheSize` дескриптора базы данных.

Дополнительная информация

Потоки и безопасность потоков

В Perl версии 5.004_50 и выше включена экспериментальная поддержка многопоточности для ряда платформ. Если DBI скомпилирован при включенной многопоточности Perl, то в нем используются мьютексы для драйверов, обеспечивающие одновременную работу только одного потока для каждого драйвера. Пожалуйста, учтите, что поддержка потоков в Perl все еще является экспериментальной и в ней замечены некоторые существенные проблемы. Использовать ее не рекомендуется.

Обработка сигналов и отмена операций

Первое, что нужно отметить: обработка сигналов в Perl все еще не безопасна. Всегда есть некоторый риск аварийного завершения работы Perl и/или получения дампа памяти во время или после обработки сигнала. (Риск был уменьшен в версии 5.004_04, но все еще присутствует.)

Два наиболее частых случая использования сигналов в связи с DBI – это отмена операций при нажатии пользователем <Ctrl>+<C> (прерывание) и реализация обработки тайм-аута с использованием `alarm()` и `$SIG{ALRM}`.

В помощь реализации этих действий DBI предоставляет метод `cancel` для дескрипторов команд. Метод `cancel` должен прерывать текущую операцию и предназначен для вызова из обработчика сигнала.

Однако необходимо подчеркнуть: а) в данный момент эти функции реализованы в малом числе драйверов (в DBI есть метод по умолчанию, который просто возвращает `undef`); и б) даже если эти функции реализованы, существует возможность того, что впоследствии дескриптор команды, а возможно, и родительский дескриптор базы данных, окажутся неработоспособными.

Если `cancel` возвращает true, значит, он успешно вызвал собственную функцию `cancel` ядра базы данных. Если он возвращает false, значит, отмена операции не удалась. Если возвращается `undef`, значит, в ядре базы данных не реализована функция отмены операции.

Дополнительные источники информации

Документация по драйверам и базам данных

Читайте документацию по используемому вами драйверу DBD.

Читайте справочное руководство по языку SQL для используемого вами ядра базы данных.

Книги и журналы

«Programming the Perl DBI», Alligator Descartes and Tim Bunce («Программирование на Perl DBI», Аллигатор Декарт и Тим Банс).

«Programming Perl, 3rd Ed», Larry Wall, Tom Christiansen and Randal Schwartz («Программирование на Perl», 3-е издание, Ларри Уолл, Том Кристиансен, Рэндал Шварц).

«Learning Perl», Randal Schwartz («Изучаем Perl», Рэндал Шварц).

«Dr Dobb's Journal», November 1996 («Журнал д-ра Доббса», ноябрь, 1996).

«The Perl Journal», April 1997 («Журнал Perl», апрель 1997).

Страницы электронного руководства

См. страницы руководства *perl*, *perlmod* и *perlbook*.

Почтовый список рассылки

Основным средством общения пользователей DBI и родственных модулей служит список рассылки *dbi-users*. Подписаться и прекратить подписку можно через:

http://www.isc.org/dbi-lists.html

Обычно в месяц проходит от 700 до 900 сообщений. Для того чтобы посылать сообщения, нужно подписаться. Однако можно выбрать подписку «только отправка».

Архивы списка рассылки находятся по адресам:

http://www.xray.mpe.mpg.de/mailing-lists/dbi/

http://www.egroups.com/list/dbi-users/info.html

http://www.bitmechanic.com/mail-archives/dbi-users/

Различные близкие ссылки в Интернете

Домашняя страница DBI:

http://www.symbolstone.org/technology/perl/DBI

Другие ссылки, относящиеся к DBI:

http://tegan.deltanet.com/~phlip/DBUIdoc.html
http://dc.pm.org/perl_db.html
http://wdvl.com/Authoring/DB/Intro/toc.html
http://www.hotwired.com/webmonkey/backend/tutorials/tutorial1.html

Другие ссылки, относящиеся к базам данных:

http://www.jcc.com/sql_stnd.html
http://cuiwww.unige.ch/OSG/info/FreeDB/FreeDB.home.html

Коммерческие ссылки и ссылки на хранилища данных:

http://www.dwinfocenter.org
http://www.datawarehouse.com
http://www.datamining.org
http://www.olapcouncil.org
http://www.idwa.org
http://www.knowledgecenters.org/dwcenter.asp

Рекомендуемая ссылка по программированию на Perl:

http://language.perl.com/style/

FAQ

Прочтите, пожалуйста, DBI FAQ, устанавливаемый как модуль `DBI::FAQ`. Для его чтения можно использовать `perldoc`, выполнив команду `perldoc DBI::FAQ`.

Авторы

Создателем DBI является Тим Банс (Tim Bunce). Авторы текста: Тим Банс, Дж. Дуглас Данлоп (J. Douglas Dunlop), Джонатан Леффлер (Jonathan Leffler) и другие. Создателями Perl являются Ларри Уолл и `perl5-porters`.

Copyright

Распространение DBI допускается в рамках GNU General Public License или Artistic License, как указано в файле README Perl.

Благодарности

Я хотел бы выразить благодарность за неоценимый вклад, внесенный многими людьми, с которыми я работал над проектом DBI, особенно в ранние годы (1992–1994). Порядок не имеет значения: Кевин Сток, Баз Мочетти, Курт Андерсен, Тед Лемон, Уильям Хейлс, Гарт Кеннеди, Майкл Пепплер, Нейл Бриско, Джефф Урлвин, Дэвид Хьюз, Джефф Стэндер, Форрест Уитчер, Ларри Уолл, Джефф Фрайд, Рой Джонсон, Пол Хадсон, Георг Рехфелд, Стив Сайзмор, Рон Пул, Джон Мик, Том Кристиансен, Стив Баумгартен, Рэндал Шварц и многие другие.

И, конечно, нужно поблагодарить тех несчастных, которые прорывались через бессчетные недокументированные препятствия, чтобы реализовать на практике драйверы DBI. В их рядах находятся Йохан Видман, Аллигатор Декарт, Джонатан Леффлер, Джефф Урлвин, Майкл Пепплер, Хенрик Тугарт, Эдвин Прамото, Дэвид Мигливакка, Ян Паздзиора, Питер Хэйворт, Эдмунд Мергл, Стив Уильямс, Томас Лоури и Филип Пламли. Без них DBI не смог бы стать сегодня реальностью. Я особенно благодарен Аллигатору Декарту, начавшему работу над данной книгой и позволившему мне включиться в нее.

Переводы

Благодаря возможности, предоставленной O'Reilly, можно получить перевод на немецкий язык данного текста и другой документации по модулям Perl (все переводы могут оказаться несколько устаревшими) по адресу:

http://www.oreilly.de/catalog/perlmodger

Некоторые другие переводы:

Испанский – http://cronopio.net/perl/
Японский – http://member.nifty.ne.jp/hippo2000/dbimemo.htm

Поддержка/гарантии

DBI является свободно распространяемым программным обеспечением. *Оно распространяется без каких бы то ни было гарантий*.

О коммерческой поддержке Perl и модулей DBI, `DBD::Oracle` и Oraperl можно договориться с The Perl Clinic. Подробности можно узнать на

http://www.perlclinic.com

Обучение

Вот ссылки, по которым можно выяснить возможность пройти обучение по DBI (это не является рекомендацией):

http://www.treepax.co.uk/
http://www.keller.com/dbweb/

B

Характеристики драйверов и баз данных

В этом приложении мы надеемся дать вам почувствовать особенности и функциональность различных драйверов DBI и соответствующих баз данных.

Граница между функциональностью и отличительными особенностями драйвера, с одной стороны, и функциональностью и отличительными особенностями соответствующей ему базы данных – с другой, несколько размыта. В некоторых случаях в ядре базы данных имеются функции, которые драйвер не использует или не может использовать; в других случаях драйвер эмулирует функции, не поддерживаемые базой данных, например, использование заполнителей. Поэтому, встречая в дальнейшем термины *драйвер* или *база данных*, относитесь к ним скептически.

Мы ставим себе следующие главные цели:

- дать простой обзор каждого драйвера и базы данных;
- помочь в первоначальном выборе подходящего драйвера DBI и базы данных для создаваемых приложений;
- помочь выявить проблемы, с которыми можно столкнуться при переносе существующих приложений из одной комбинации драйвера и базы данных в другую.

Мы не пытаемся подробно описывать драйверы и базы данных и не воспроизводим их документацию. Нас интересуют только основные характеристики, которые наиболее часто используются или имеют отношение к нашим задачам. Для этих характеристик мы предоставляем всего лишь общее руководство, иногда едва выходящее за рамки

указателя. Полную детализацию следует искать в документации по драйверу и базе данных.

В сотрудничестве с авторами драйверов мы разработали описания для следующих драйверов и баз данных:

DBD::ADO

Microsoft «Active Data Objects»

DBD::CSV

ASCII-файлы со значениями, разделяемыми запятой.

DBD::DB2

IBM DB2.

DBD::Empress

Empress.

DBD::Informix

Informix.

DBD::Ingres

Ingres.

DBD::InterBase

InterBase.

DBD::mysql и DBD::mSQL

Базы данных MySQL и mSQL.

DBD::ODBC

Для любых источников данных ODBC.

DBD::Oracle

Oracle.

DBD::Pg

PostgreSQL.

DBD::SearchServer

Fulcrum Search Server.

DBD::Sybase

Для Sybase и Microsoft SQL Server.

DBD::XBase

Для файлов XBase (dBase и т. д.)

Для каждого из этих драйверов мы постарались охватить один и тот же круг тем в одинаковом порядке.

Темы включают в себя:

- Общие сведения о драйвере.

- Способ соединения.
- Поддерживаемые типы данных, их диапазон и выполняемые функции.
- Особенности диалекта SQL и поведение по умолчанию.
- Расширенные функции базы данных.
- Способ доступа к метаданным базы данных.

Читать все подряд – занятие не для робких. Рекомендуем читать по мере надобности именно то, что вам необходимо в связи с возникающими вопросами и проблемами.

Получение DBI и драйверов

Прежде чем начать использовать модуль драйвера DBI, его, очевидно, нужно откуда-то получить и установить в системе.

Если вы работаете под Microsoft Windows и используете версию Perl ActiveState, то сначала нужно испытать администратор пакетов Perl – *Perl Package Manager*, или *PPM*. Утилита PPM инсталлируется с ActiveState Perl и значительно упрощает загрузку и инсталляцию прекомпилированных модулей. При инсталляции драйвера DBI с помощью PPM также автоматически инсталлируется DBI, если это не было сделано раньше. Дополнительную информацию можно получить по адресу:

http://www.activestate.com/PPM/

Это простое решение срабатывает не всегда. Если вы не используете ActiveState Perl под Microsoft Windows или требуемый вам драйвер не входит в число прекомпилированных для загрузки через PPM, то путь оказывается более долгим: нужно загрузить исходный код драйвера и скомпилировать его самостоятельно. Обычно это не столь трудно, как может показаться.

Исходный код драйверов DBI можно загрузить с любого узла, входящего в полную архивную сеть Perl «Comprehensive Perl Archive Network» (CPAN). Вот несколько хороших URL, с которых можно начать:

http://www.perl.com/CPAN/modules/by-module/DBD/

http://www.perl.org/CPAN/modules/by-module/DBD/

http://search.cpan.org/search?mode=module&query=DBD

Если вы еще не инсталлировали DBI, это необходимо проделать в первую очередь. Заменив в приведенных выше адресах DBD на DBI, вы найдете исходный код для модуля DBI.

Учтите, что многие драйверы систем баз данных требуют, чтобы при компиляции драйвера на машине имелось некоторое специфическое

клиентское программное обеспечение. Информация обо всех необходимых компонентах должна содержаться в документации к драйверу.

DBD::ADO

Общие сведения

Версия драйвера

DBD::ADO версия 0.03.

На момент написания этой книги драйвер DBD::ADO и собственно ADO остаются относительно новыми. Их характеристики, несомненно, изменятся, поэтому читайте новейшую документацию.

Краткая характеристика

Поскольку DBD::ADO выступает в качестве интерфейса к другим драйверам низкого уровня для баз данных в Windows, значительная часть его поведения определяется этими драйверами.

Транзакции	В зависимости от источника данных
Блокировка	В зависимости от источника данных
Соединения таблиц	В зависимости от источника данных
LONG/LOB-типы данных	В зависимости от источника данных
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Пока нет
Хранимые процедуры	Ограниченная поддержка, без параметров
Связывание выходных значений	Нет
Регистр символов в именах таблиц	В зависимости от источника данных
Регистр символов в именах полей	В зависимости от источника данных
Преобразование недопустимых имен	В зависимости от источника данных
Нечувствительный к регистру оператор "LIKE"	В зависимости от источника данных
Псевдоколонка ROW ID в серверных таблицах	В зависимости от источника данных
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	В зависимости от источника данных

Автор и как с ним связаться

Драйвер сопровождают Томас Лоури (Thomas Lowery) и Филип Пламли (Phlip Plumlee). Связаться с ними можно через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Для надежной работы модулю DBD::ADO требуется Microsoft ADO версии 2.1 или выше. Рекомендуется использовать NT с Service Pack 4.

Модуль написан на чистом Perl и использует модуль `Win32::OLE` для обработки запросов ADO.

Модуль `DBD::ADO` поддерживает использование команд SQL для запросов к любым источникам данных, поддерживаемым ADO. Это могут быть драйверы Jet для различных форматов файлов Microsoft Office, любые драйверы данных ODBC или экспериментальные поставщики данных, представляющие иерархии папок файловой системы или службы каталогов Интернета как источники данных.

Каждая предоставляющая данные система поддерживает некоторый вид SQL либо в собственном формате, например MS-SQL Server Transact SQL, либо в качестве слоя эмуляции в поставщике данных, как, например, при чтении данных электронной таблицы Excel драйвером Jet.

Сведения об ADO можно найти на *http://www.microsoft.com/data/ado/*.

Отличия от спецификации DBI

`DBD::ADO` – очень новый и в настоящее время неполный драйвер. Однако он быстро развивается, и поскольку написан на чистом Perl с использованием `Win32::OLE`, он легко доступен для усовершенствования.

Синтаксис соединения

Формат `DBI->connect()` «Имя источника данных» (*Data Source Name*), или DSN, следующий:

```
dbi:ADO:DSN
```

DSN должен быть именем источника данных ODBC, зарегистрированного приложением панели управления ODBC Data Sources. Если приложение DBI выполняется как системный сервис или демон, например, как сценарий CGI, то DSN должен присутствовать на вкладке «System DSN».

Для метода `DBI->connect()` нет специфических для драйвера атрибутов. `DBD::ADO` поддерживает неограниченное число одновременных соединений с одним или несколькими источниками данных, ограничения могут быть обусловлены лишь самими источниками данных.

Типы данных

Типы данных – числовые, строковые, даты/времени и LONG/LOB – зависят от взаимодействия четырех составляющих: поддержки в «скалярах» Perl, способа трансляции VARIANT в скаляры слоем `Win32::OLE`, типов, допустимых самим VARIANT, и типов, предоставляемых конечным поставщиком данных.

Пользователь/программист должен изучить эти четыре характеристики по соответствующей документации, чтобы выяснить, будет ли DBD::ADO правильно передавать типы.

Транзакции, изоляция и блокировка

DBD::ADO предоставляет пользователю возможности, которыми обладает собственное *соединение*. Если поставщик данных поддерживает транзакции, то они осуществляются в рамках соединения: все команды, являющиеся потомком одного соединения, могут «видеть» обновления данных, ожидающие команды COMMIT. Для остальных соединений с источником данных эти незавершенные изменения будут не видны.

Диалект SQL

Поскольку DBD::ADO выступает как интерфейс к другим драйверам баз данных, от этих драйверов и баз данных, с которыми они осуществляют соединение, зависят следующие характеристики:

- чувствительность к регистру оператора LIKE;
- синтаксис соединения таблиц;
- имена таблиц и колонок;
- серверная псевдоколонка Row ID;
- автоматическая генерация ключей и последовательностей;
- автоматическая нумерация строк и предельное количество строк.

Дополнительные сведения могут быть получены в документации по используемым драйверам и базам данных.

Драйвер DBD::ADO не поддерживает обновление и удаление по месту.

Связывание параметров

Связывание параметров пока не поддерживается DBD::ADO.

Хранимые процедуры

Вызов хранимых процедур поддерживается DBD::ADO с использованием синтаксиса в стиле ODBC {call procedure_ name()}.

Метаданные таблиц

DBD::ADO в данное время не поддерживает метод table_info(). Для его реализации требуется дополнительное время и активные добровольцы.

Специфические для драйверов атрибуты и методы

Доступ к объекту ADO *соединение* может осуществляться из дескрипторов базы данных и команд через атрибут `ado_conn`.

Доступ к объекту ADO *RecordSet* осуществляется из дескриптора команды через атрибут `ado_rs`.

DBD::CSV

Общие сведения

Версия драйвера

`DBD::CSV` версия 0.1019.

Краткая характеристика

Транзакции	Нет
Блокировка	Неявная, только покомандная
Соединения таблиц	Нет
LONG/LOB-типы данных	Да, до 4 Гбайт
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?"
Хранимые процедуры	Нет
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Учитывается, зависит от файловой системы
Регистр символов в именах полей	Учитывается, сохраняется исходный регистр
Преобразование недопустимых имен	Нет
Нечувствительный к регистру оператор "LIKE"	Да, "CLIKE"
Псевдоколонка ROW ID в серверных таблицах	Нет
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Йохан Видман (Jochen Wiedmann). С ним можно связаться через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Драйвер `DBD::CSV` использует сервисы, предоставляемые несколькими другими модулями. Модуль `Text::CSV_XS` используется для чтения и записи файлов CSV. Абстрактный родовой класс `DBD::File` предоставляет драйверу структуру для работы с *плоскими файлами*. В свою очередь, он использует модуль `SQL::Statement` для синтаксического анализа и оценки простых команд SQL.

Важно отметить, что хотя все считают, будто им известен формат файлов CSV, на самом деле формального определения этого формата не существует, зато есть многочисленные тонкие различия.

Одно из описаний файлов CSV можно найти здесь:

http://www.whatis.com/csvfile.htm

Отличия от спецификации DBI

`DBD::CSV` не производит полный синтаксический анализ команды до ее выполнения. Поэтому такие атрибуты, как `$sth->{NUM_OF_FIELDS}`, недоступны, пока не будет вызван `$sth->execute()`. Это нормальное поведение, но его нужно учитывать при переносе приложений, первоначально написанных для других драйверов.

Атрибуты дескриптора команд `PRECISION`, `SCALE` и `TYPE` не поддерживаются. Обратите также внимание, что многие атрибуты становятся недоступными после выборки всех результирующих строк или вызова метода `finish()`.

Синтаксис соединения

Синтаксис `DBI->connect()` «Имя источника данных» (DSN) может быть следующим:

```
dbi:CSV:
dbi:CSV:attrs
```

где `attrs` является необязательным списком пар *ключ=значение*, разделяемых точкой с запятой.

Количество дескрипторов баз данных ограничено только объемом памяти. Рекомендуется использовать различные дескрипторы баз данных для каждого из форматов таблиц.

Часто используются следующие атрибуты:

`f_dir=directory`

По умолчанию в качестве таблиц рассматриваются файлы в текущем каталоге. Атрибут `f_dir` заставляет модуль открывать файлы в заданном каталоге.

`csv_eol`

`csv_sep_char`

`csv_quote_char`

`csv_escape_char`

Эти атрибуты применяются для описания используемого формата файла CSV. Например, чтобы открыть как таблицу файл */etc/passwd*, в котором значения разделяются двоеточиями, а строки оканчиваются символом новой строки, следует использовать следующие значения:

```
csv_eol=\n;csv_sep_char=:
```

Значениями по умолчанию являются `\r\n`, запятая (`,`), двойная кавычка (`"`) и двойная кавычка (`"`), соответственно. Все эти атрибуты и значения по умолчанию наследуются из модуля `Text::CSV_XS`.

Типы данных

Обработка числовых данных

Несомненно, основным недостатком модуля `DBD::CSV` является отсутствие надлежащей обработки типов. При чтении таблицы CSV нет способа надежно определить правильный тип данных полей. Все поля выглядят как строки и по умолчанию так и обрабатываются.

Модуль `SQL::Statement`, а следовательно, и драйвер `DBD::CSV`, принимает числовые типы `INTEGER` и `REAL` в командах `CREATE TABLE`, но они всегда хранятся как строки и по умолчанию извлекаются как строки.

Отдельные колонки можно читать как целые или вещественные числа, и тогда они конвертируются во внутренние типы Perl IV и NV – целые и числовые значения, соответственно. Беззнаковые значения не поддерживаются.

Для назначения колонкам определенных типов нужно создать *определения метаданных*. В следующем примере читается таблица *table_name* с колонками `I`, `N` и `P` типов `INTEGER`, `DOUBLE` и `STRING`, соответственно:

```
my $dbh = DBI->connect("DBI:CSV:", '', '');
$dbh->{csv_tables}->{table_name}->{types} = [
    Text::CSV_XS::IV(), Text::CSV_XS::NV(), Text::CSV_XS::PV()
];
my $sth = $dbh->prepare("SELECT id, sales, description FROM table_name");
```

Обработка строковых данных

Как и в случае числовых типов, `DBD::CSV` допускает в командах `CREATE TABLE` больше типов данных, чем реально поддерживается. Можно использовать `CHAR(n)` и `VARCHAR(n)` с произвольными числами `n`, `BLOB` или `TEXT`, но на самом деле это всегда `BLOB` в некотором свободном виде.

Лежащий в основе строковый тип может хранить любые двоичные данные, в том числе встроенные символы NUL. Однако для многих других средств работы с файлами CSV такие данные могут оказаться неприемлемыми.

Обработка данных типа «дата»

Прямой поддержки типов данных для даты или времени нет.

Обработка данных типа LONG/BLOB

Данные `BLOB` эквивалентны строкам. Размер ограничивается доступной памятью.

Другие вопросы обработки данных

Поддерживается метод `type_info()`.

Транзакции, изоляция и блокировка

Драйвер не поддерживает транзакции.

Блокировки явно не поддерживаются. Таблицы блокируются на время выполнения команд, но блокировка снимается сразу по выполнении команды.

Диалект SQL

Чувствительность к регистру оператора LIKE

Поддерживаются два различных оператора `LIKE`. Оператор `LIKE` чувствителен к регистру, а `CLIKE` — нет.

Синтаксис соединения таблиц

Соединение таблиц не поддерживается.

Имена таблиц и колонок

Имена таблиц и колонок чувствительны к регистру. Однако следует учитывать, что имена таблиц являются, в сущности, именами файлов, поэтому таблицы *Foo* и *foo* могут существовать одновременно с идентичными данными. Тем не менее для них могут существовать различные определения метаданных в `$dbh->{csv_tables}`.

См. ниже «Метаданные таблиц» для получения дополнительных сведений об именах таблиц и колонок.

Row ID

Серверная псевдоколонка Row ID не поддерживается.

Автоматическая генерация ключей и последовательностей

Автоматические ключи и счетчики не поддерживаются.

Автоматическая нумерация строк и предельное число строк

Ни автоматическая нумерация строк, ни предельное число строк не поддерживаются.

Обновление и удаление по месту

Обновление и удаление по месту не поддерживаются.

Связывание параметров

В качестве заполнителей используются вопросительные знаки:

```
$dbh->do("INSERT INTO A VALUES (?, ?)", undef, $id, $name);
```

Заполнители в стиле `:1` не поддерживаются.

Хранимые процедуры

Хранимые процедуры не поддерживаются.

Метаданные таблиц

По умолчанию драйвер предполагает, что имена колонок хранятся в первой строке таблицы, например:

```
login:password:uid:gid:comment:shell:homedir
root:s34hj34n34jh:0:0:Superuser:/bin/bash:/root
```

Если имен колонок в таблице нет, можно задать их следующим образом:

```
$dbh->{csv_tables}->{$table}->{skip_rows} = 0;
$dbh->{csv_tables}->{$table}->{col_names} =
    [qw(login password uid gid comment shell homedir)];
```

В этом случае считается, что в первой строке содержатся данные.

Если не передавать имена колонок и они не считываются из первой строки, то автоматически генерируются имена *col0*, *col1* и т. д.

Имена колонок можно извлечь через стандартный атрибут `$sth->{NAME}`. Атрибут NULLABLE возвращает массив, состоящий из единиц. Прочие атрибуты метаданных не поддерживаются.

Имена таблиц или файлов можно как обычно получать через `$dbh->table_info()` или `$dbh->tables()`.

Специфические для драйверов атрибуты и методы

Помимо атрибутов `f_dir`, `csv_eol`, `csv_sep_char`, `csv_quote_char` и `csv_sep_char`, о которых уже говорилось выше, наиболее важным атрибутом дескриптора базы данных является следующий:

```
$dbh->{csv_tables}
```

`csv_tables` можно использовать для задания метаданных таблицы. Это ссылка на хэш, ключами которого являются имена таблиц, а значениями – ссылки на хэши со следующими атрибутами:

`file`

Имя файла, связанного с таблицей. По умолчанию именем файла является `$dbh->{f_dir}/$table`.

`col_names`

Ссылка на массив имен колонок.

`skip_rows`

Число строк, которые нужно прочесть в начале файла перед чтением данных таблицы. Первая из строк будет рассматриваться в качестве массива имен колонок, однако атрибут `col_names` имеет более высокий приоритет.

`types`

Это ссылка на массив `Text::CSV_XS` значений типов для соответствующих колонок. Поддерживается три типа, значения которых определяются функциями `IV()`, `NV()` и `PV()` пакета `Text::CSV_XS`.

Для всех дескрипторов отсутствуют специфические для драйвера атрибуты и закрытые методы.

DBD::DB2

Общие сведения

Версия драйвера

`DBD::DB2` версия 0.71.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да, "?" (собственные)
Хранимые процедуры	Да
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Не учитывается, хранятся в верхнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет
Псевдоколонка ROW ID в серверных таблицах	Нет
Обновление/удаление по месту	Да
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Поддержка драйвера DBD::DB2 обеспечивается IBM в соответствии с соглашениями по обслуживанию DB2 UDB. Комментарии, предложения и просьбы по развитию можно посылать электронной почтой на *db2perl@ca.ibm.com*. Дополнительные сведения можно найти на сайте:

http://www.ibm.com/data/db2/perl

Поддерживаемые версии и варианты баз данных

Драйвер DBD::DB2 поддерживает DB2 UDB V5.2 и более поздние.

Вот несколько URL, содержащих специфические для базы данных/драйвера сведения:

http://www.software.ibm.com/data/db2/perl

http://www.software.ibm.com/data/db2

http://www.software.ibm.com/data/db2/library

http://www.software.ibm.com/data/db2/udb/ad

Отличия от спецификации DBI

Единственным существенным отличием от текущей спецификации DBI является способ, которым типы данных указываются в методе bind_param(). Пожалуйста, посмотрите далее в этом разделе, как использовать метод bind_param() с драйвером DBD::DB2.

Синтаксис соединения

Синтаксис DBI->connect() «Имя источника данных» (DSN) следующий:

```
dbi:DB2:database_name
```

Специфических для драйвера атрибутов метода DBI->connect() нет.

DBD::DB2 поддерживает одновременные соединения с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

DB2 UDB поддерживает следующие числовые типы данных:

```
SMALLINT
INTEGER
BIGINT
REAL
DOUBLE
FLOAT
DECIMAL или NUMERIC
```

SMALLINT является двухбайтовым целым в диапазоне от –32768 до +32767. Максимальная точность равна 5. Масштаб не поддерживается.

INTEGER является четырехбайтовым целым в диапазоне от –2147483648 до +2147483647. Максимальная точность равна 10. Масштаб не поддерживается.

BIGINT является восьмибайтовым целым в диапазоне от –9223372036854775808 до +9223372036854775807. Максимальная точность равна 19. Масштаб не поддерживается.

REAL является 32-битовым приближением вещественного числа. Число может быть равно 0 или находиться в диапазоне от –3,402e+38 до –1,175e–37 или от +1,175e–37 до +3,402e+38. Максимальная точность равна 7. Масштаб не поддерживается.

DOUBLE или FLOAT является 64-битовым приближением вещественного числа. Число может быть равно 0 или находиться в диапазоне от –1,79769e+308 до –2,225e–307 или от 2,225e–307 до 1,79769e+308. Максимальная точность равна 15. Масштаб не поддерживается.

DECIMAL или NUMERIC является упакованным десятичным числом с неявной десятичной точкой в диапазоне от –10**31+1 до +10**31–1. Максимальная точность – 31 знак. Масштаб не может быть отрицательным или большим, чем максимальная точность.

Обратите внимание, что DB2 поддерживает числа за пределами обычного допустимого диапазона чисел в Perl. Это не вызывает проблем, поскольку DBD::DB2 всегда возвращает числа как строки.

Обработка строковых данных

DB2 UDB поддерживает следующие строковые типы данных:

```
CHAR
CHAR FOR BIT DATA
VARCHAR
VARCHAR FOR BIT DATA
GRAPHIC
VARGRAPHIC
```

CHAR является символьной строкой фиксированной длины до 254 байтов. VARCHAR является символьной строкой переменной длины до 32 672 байтов. Варианты FOR BIT DATA используются с данными, не связанными с определенным набором кодировки символов.

GRAPHIC является строкой фиксированной длины двухбайтовых символов длиной до 127 символов.

VARGRAPHIC является строкой переменной длины двухбайтовых символов длиной до 16 336 символов.

Типы CHAR и GRAPHIC являются строками фиксированной длины с добавлением пробелов.

В DB2 UDB поля CHAR могут иметь смешанные наборы кодировки (национальные наборы символов). Символы не-ASCII обрабатываются согласно определению кодовой страницы. Например, символы Shift-JIS в диапазоне от 0x81 до 0x9F и от 0xE0 до 0xFC являются первыми байтами DBCS (двухбайтовой кодировки), а символы в диапазоне от 0xA0 до 0xDF являются однобайтовыми символами Катаканы. Дописка пробелов в поля CHAR всегда производится пробелом ASCII (однобайтовым). Для UTF-8 символы с установленным знаковым разрядом интерпретируются согласно определению UTF-8.

Типы GRAPHIC хранятся как чисто двухбайтовые в кодовой странице базы данных, установленной по умолчанию, или в UCS-2 для базы в кодировке Unicode. Дописка пробелов для полей GRAPHIC всегда производится пробелом DBCS для соответствующей кодовой страницы или пробелом UCS-2 (U+0020) для баз данных в кодировке Unicode.

DB2 UDB автоматически производит преобразование кодовой страницы между кодовой страницей клиента и кодовой страницей базы данных.

Поддержка Unicode обеспечивается DB2 UDB Version 5 + FixPak 7 (DB2 UDB V5.2 является фактически DB2 UDB V5 + FixPak 6). В базе данных Unicode типы данных CHAR хранятся в формате UTF-8, а типы данных GRAPHIC хранятся в формате UCS-2.

В DB2 UDB версии 6.1 функция VARCHAR() расширена таким образом, что преобразует типы данных графических строк в VARCHAR, исключая LONG VARGRAPHIC и DBCLOB. Эта функция допустима только для баз данных UCS-2. Для баз данных не в формате Unicode она недопустима.

Все символьные типы могут хранить строки с байтами nul("\0").

Конкатенация строк может осуществляться с помощью оператора || или функции SQL CONCAT(s1,s2).

Обработка данных типа «дата»

DB2 UDB поддерживает следующие типы данных для хранения даты, времени и даты/времени:

```
DATE
TIME
TIMESTAMP
```

DATE состоит из трех частей, представляющих год, месяц и день. Диапазон хранения года – от 0001 до 9999. Год, состоящий из двух цифр, нельзя использовать в DB2 UDB. Он должен задаваться всеми четырьмя цифрами.

TIME состоит из трех частей, представляющих час, минуты и секунды на 24-часовом циферблате.

TIMESTAMP состоит из семи частей, представляющих год, месяц, день, час, минуты, секунды и микросекунды в обозначении даты и времени,

как описано выше, за исключением дробной части в микросекундах. Если задать значение типа TIMESTAMP без указания времени, то устанавливается время 00:00:00 (полночь).

Текущие дату, время и дату/время можно получить с помощью специальных регистров CURRENT DATE, CURRENT TIME и CURRENT TIMESTAMP.

DB2 UDB поддерживает следующие форматы даты, времени и даты/времени:

```
ISO   (International Standards Organization)
USA   (IBM USA standard)
EUR   (IBM European standard)
JIS   (Japanese Industrial Standard Christian era)
LOC   (по месту, зависит от кода страны базы данных)
```

Можно вводить значения даты, времени и даты/времени в любом поддерживаемом формате, например:

```
create table datetest(dt date);
insert into datetest('1991-10-27');
insert into datetest('10/27/1991');
```

Форматом по умолчанию для вывода DATE, TIME и TIMESTAMP является тот, который ассоциирован с кодом страны базы данных (указанный выше формат LOC). Можно также использовать функцию CHAR() и указать альтернативный формат.

Значения даты/времени можно инкрементировать, декрементировать и вычитать одно из другого. DB2 UDB предоставляет широкий спектр функций даты, в том числе DAY(), DAYOFWEEK(), DAYOFYEAR(), MONTHNAME() и TIMESTAMPDIFF(). Другие функции можно найти в документации по DB2 UDB.

Следующее выражение SQL можно использовать для преобразования целого числа «секунд после 1 января 1970 года» к соответствующему значению дата/время в базе данных (местное не-GMT время):

```
TIMESTAMP('1970-01-01','00:00') + seconds_since_epoch
```

Простого выражения, осуществляющего обратное преобразование, нет. Вычитание time_stamp('1970-01-01','00:00') из другой временной метки дает длительность временной метки, являющуюся значением DECIMAL(20,6) в формате yyyymmddhhmmss.zzzzzz.

DB2 не производит автоматической настройки временного пояса.

Обработка данных типа LONG/BLOB

DB2 UDB поддерживает следующие типы данных LONG/BLOB:

```
BLOB
CLOB
DBCLOB
```

```
LONG VARCHAR
LONG VARCHAR FOR BIT DATA
LONG VARGRAPHIC
```

BLOB (большой двоичный объект) является строкой переменной длины, измеряемой в байтах и имеющей длину до 2 Гбайт. BLOB предназначен, главным образом, для хранения необычных данных, таких как изображения, голос и смешанные мультимедиа. BLOB не связывается с набором кодировки символов (аналогично символьным строкам FOR BIT DATA; см. ниже).

CLOB (большой символьный объект) является строкой переменной длины, измеряемой в байтах и имеющей длину до 2 Гбайт. CLOB используется для хранения больших объемов символьных данных.

DBCLOB (большой объект двухбайтовых символов) является строкой переменной длины двухбайтовых символов до 1 073 741 823 символов. DBCLOB используется для хранения больших объемов символов DBCS.

LONG VARCHAR является строкой переменной длины, которая может иметь длину до 32 700 байт. LONG VARCHAR FOR BIT DATA используется с данными, не связанными с набором кодировки символов.

LONG VARGRAPHIC является строкой переменной длины двухбайтовых символов, которая может иметь длину до 16 350 символов.

Ни для одного из этих типов данных не требуется обмен с базой данных в виде пар шестнадцатеричных цифр.

К сожалению, драйвер DBD::DB2 пока не поддерживает атрибуты LongReadLen и LongTruncOk. Вставлять и извлекать можно данные максимально большого для соответствующего типа данных размера, хотя ресурсы системы могут накладывать ограничения.

Драйвер DBD::DB2 необычен тем, что требует усиленного использования атрибутов связанных параметров как для обычных типов данных, так и для типов LONG/BLOB. (См. раздел «Связывание параметров», в котором рассказывается о хэшах атрибутов.) Вот, например, хэш атрибутов для CLOB, который в данном приложении имеет размер до 100 Кбайт:

```
$attrib_clob = {
  ParamT => SQL_PARAM_INPUT,
  Ctype  => SQL_C_CHAR,
  Stype  => SQL_CLOB,
  Prec   => 100000
  Scale  => 0,
};
```

Другие вопросы обработки данных

Драйвер DBD::DB2 пока не поддерживает метод type_info().

DB2 UDB не производит автоматическое преобразование строк в числа и чисел в строки.

Транзакции, изоляция и блокировка

DB2 UDB поддерживает транзакции и четыре уровня изоляции транзакций: Repeatable Read, Read Stability, Cursor Stability, Uncommited Read. По умолчанию установлен уровень изоляции Cursor Stability (устойчивость курсора).

Для драйвера `DBD::DB2` уровень изоляции может быть изменен установкой желаемого значения для ключевого слова TXNISOLATION в файле *db2cli.ini*. Это ключевое слово устанавливается в секции, относящейся к специфической базе данных, что означает его воздействие на все приложения, осуществляющие соединение с этой базой данных. Изменить уровень изоляции транзакций с помощью SQL нельзя.

Режим по умолчанию для чтения и записи основывается на уровне изоляции. Строки, возвращаемые командой `SELECT`, можно заблокировать явным образом, дополнив ее словами `FOR UPDATE` и списком имен полей, например:

```
SELECT colname1, colname2
FROM tablename
WHERE colname1 = 'testvalue'
FOR UPDATE OF colname1, colname2
```

Чтобы явным образом заблокировать всю таблицу, можно использовать команду `LOCK TABLE table_name IN lock_mode`.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор `LIKE` чувствителен к регистру.

Синтаксис соединения таблиц

Внутреннее соединение осуществляется с использованием обычного синтаксиса `WHERE a.field = b.field`. Можно также использовать следующий синтаксис:

```
SELECT tablea.col1, tableb.col1
FROM tablea INNER JOIN tableb
ON tableb.name = tablea.name
```

DB2 UDB поддерживает левые внешние соединения, правые внешние соединения и полные внешние соединения. Например, для левого внешнего соединения можно использовать такую команду:

```
SELECT tablea.col1, tablea.col2, tableb.col1, tableb.col2
```

```
FROM tablea LEFT OUTER JOIN tableb
ON tableb.name = tablea.name
```

Другие формы внешнего соединения получаются в результате замены «LEFT» на «RIGHT» или «FULL».

Имена таблиц и колонок

В DB2 UDB версии 5.2 максимальная длина имени таблицы или колонки равна 18 символам. В DB2 UDB версии 6.1 максимальная длина имен таблиц будет увеличена до 128, а максимальная длина имени колонки до 30.

Первым символом имени должна быть буква, но остальные символы могут представлять собой любое сочетание букв верхнего регистра, цифр и символов подчеркивания.

Имена таблиц и полей могут заключаться в двойные кавычки (") и содержать описанные выше символы плюс буквы на нижнем регистре.

Имена таблиц и колонок хранятся в каталогах в верхнем регистре, если не используются ограничители. Идентификаторы с ограничителями сохраняют регистр. Два последовательных символа кавычки используются для представления одной кавычки в идентификаторе с ограничителями.

В именах таблиц и колонок могут использоваться национальные символы.

Row ID

DB2 UDB не поддерживает псевдоколонку «идентификатор строки таблицы».

Автоматическая генерация ключей и последовательностей

Для генерации в таблице уникальных значений (ключей) может использоваться функция GENERATE_UNIQUE. Например:

```
CREATE TABLE EMP_UPDATE (
  UNIQUE_ID CHAR(13) FOR BIT DATA, -- обратите внимание на "FOR BIT DATA"
  EMPNO CHAR(6),
  TEXT VARCHAR(1000)
)
INSERT INTO EMP_UPDATE VALUES
  (GENERATE_UNIQUE(), '000020', 'Обновление компоненты...'),
  (GENERATE_UNIQUE(), '000050', 'Обновление компоненты...')
```

К сожалению, DB2 не предоставляет возможности узнать последнее значение, сгенерированное GENERATE_UNIQUE.

DB2 UDB не поддерживает именованные генераторы последовательностей.

Автоматическая нумерация строк и предельное число строк

Не существует псевдоколонки, которую можно использовать для последовательной нумерации строк, выбираемых командой SELECT. Однако пронумеровать строки результирующего набора можно с помощью функции OLAP ROWNUMBER. Например:

```
SELECT ROWNUMBER() OVER (order by lastname) AS number, lastname, salary
FROM employee ORDER BY number;
```

В результате возвращаются строки таблицы employee с номерами, присваиваемыми в порядке возрастания lastname, упорядоченные по номеру строки.

Курсор можно объявить с предложением FETCH FIRST n ROWS ONLY, чтобы ограничить число возвращаемых строк.

Обновление и удаление по месту

DB2 UDB поддерживает обновление и удаление по месту. Поскольку специальной проверки этой возможности с драйвером DBD::DB2 не проводилось, официально она не поддерживается, однако возникновения проблем не ожидается.

Синтаксис обновления по месту такой (у DELETE аналогичный синтаксис):

```
"UPDATE ... WHERE CURRENT OF $sth->{CursorName}"
```

Связывание параметров

Связывание параметров непосредственно поддерживается DB2 UDB. Допустимы только стандартные заполнители в стиле ?.

Драйвер DBD::DB2 не поддерживает атрибут TYPE в точно такой же форме, которая описана в спецификации DBI. Для передачи данных методу bind_param() используются хэши атрибутов. Хэш атрибутов является просто совокупностью сведений о конкретном типе данных. (См. в документации по DBD::DB2 список предопределенных хэшей атрибутов.)

Ниже следует пример того, как можно создать новый хэш атрибутов:

```
$attrib_char = {
  ParamT => SQL_PARAM_INPUT,
  Ctype  => SQL_C_CHAR,
  Stype  => SQL_CHAR,
  Prec   => 254,
  Scale  => 0,
};
```

Хранимые процедуры

Хранимые процедуры вызываются с помощью следующего синтаксиса SQL:

```
CALL procedure-name(argument, ...)
```

Метаданные таблиц

DBD::DB2 пока не поддерживает метод table_info().

Представление SYSCAT.COLUMNS содержит по одной строке для каждой колонки из каждой таблицы и представления в базе данных.

Представление SYSCAT.INDEXES содержит по одной строке для каждого индекса, определенного для каждой таблицы в базе данных. Первичные ключи реализованы как уникальные индексы.

Специфические для драйверов атрибуты и методы

DBD::DB2 не имеет специфических для драйвера атрибутов или методов.

DBD::Empress и DBD::EmpressNet

Общие сведения

Версия драйвера

DBD::Empress версии 0.51.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да, "?" (собственные)
Хранимые процедуры	Да
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Учитывается, хранятся в заданном виде
Регистр символов в именах полей	Учитывается, хранятся в заданном виде
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Да, "MATCH"
Псевдоколонка ROW ID в серверных таблицах	Да, "MS_RECORD_NUMBER"
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Да, с некоторыми ограничениями

Автор и как с ним связаться

Драйвер написал Стив Уильямс (Steve Williams). С ним можно связаться через *swilliam@empress.com*.

Поддерживаемые версии и варианты баз данных

`DBD::Empress` поддерживает Empress V6.10 и более поздние версии. Дополнительные сведения можно получить на:

http://www.empress.com

Эти драйверы используют одинаковый интерфейс Perl, но различный внутренний интерфейс к базе данных. `DBD::Empress` предназначен для прямого доступа к базе данных, а `DBD::Empress_Net` предназначен для распределенных баз данных, соединение с которыми устанавливается через Empress Connectivity Server (называемый в Empress v8.10 и более ранних версиях сервером Empress ODBC).

Отличия от спецификации DBI

Существенных отличий нет.

Синтаксис соединения

Формат `DBI->connect()` «Имя источника данных» (DSN) может быть следующим:

```
dbi:Empress:physical_database
dbi:EmpressNet:logical_database
dbi:EmpressNet:SERVER=server_name;DATABASE=physical_database;PORT=port_number
```

Специфических для драйвера атрибутов метода `DBI->connect()` нет.

`DBD::EmpressNet` поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

`DBD::Empress` тоже поддерживает множественные одновременные соединения с одной или несколькими базами данных, однако эти соединения имитируются, поэтому существует ряд ограничений. Большинство этих ограничений связано с обработкой транзакций: 1) `AutoCommit` должен быть одновременно включен или выключен для всех соединений; 2) при переключении обработки с одной базы данных на другую все транзакции в первой базе данных автоматически фиксируются.

Типы данных

Обработка числовых данных

РСУБД Empress поддерживает следующие числовые типы данных:

```
DECIMAL(p,s)      от 1 до 15 цифр
DOLLAR(p,type)    от 1 до 13 цифр
```

```
REAL            Обычно 4-байтовое с плавающей точкой одинарной точности
FLOAT(p)        Обычно 4- или 8-байтовое с плавающей точкой
LONGFLOAT       Обычно 8-байтовое с плавающей точкой двойной точности
SHORTINTEGER            от -127 до 127
INTEGER               от -32767 до 32767
LONGINTEGER     от -2147483647 до 2147483647
```

Драйвер DBD поддерживает только родовые типы данных Empress. Это означает, что все данные определенной группы типов возвращаются как один и тот же тип данных. Например, SHORTINTEGER, INTEGER и LONGINTEGER возвращаются как LONG_ INTEGER.

DBD::Empress возвращает все числа как строки.

Обработка строковых данных

СУРБД Empress поддерживает следующие строковые типы данных:

```
CHAR (length, type)
NLSCHAR (length, type)
TEXT (display_length, primary, overflow, extent)
NLSTEXT (display_length, primary, overflow, extent)
```

Все аргументы имеют значения по умолчанию. Подробности можно найти в справочнике по SQL Empress SQL Reference (A4). Обычно максимальный размер для всех строковых типов составляет 2**31–1 байт (2 Гбайт). Ни один из строковых типов не дополняется пробелами.

NLSCHAR и NLSTEXT могут использоваться для хранения 8-битовых и многобайтовых символов, но UTF-8 в настоящее время не поддерживается.

Конкатенацию строк можно осуществлять с помощью функции SQL s1 CONCAT(s2).

Обработка данных типа «дата»

РСУБД Empress поддерживает следующие типы даты/времени:

```
DATE(t)           = 0000-01-01 to 9999-12-31 с разрешением 1 день
TIME(t)           = 1970-01-01 to 2035-12-31 с разрешением 1 секунда
MICROTIMESTAMP(t) = 0000-01-01 to 9999-12-31 с разрешением 1 микросекунда
```

По умолчанию форматом вывода является (t). Это один из девяти типов, определенных в разделе форматов даты/времени.

Empress поддерживает девять форматов даты/времени:

```
Type  Date                 Time                          MicroTimestamp

0     yyyymmdd             yyyymmddhhmmss                yyyymmddhhmmssffffff
1     dd aaaaaaaaa yyyy    dd aaaaaaaaa yyyy hh:mm:ss     dd aaaaaaaaa yyyy
                                                         hh:mm:ss.fffff
2     aaaaaaaaa dd, yyyy   aaaaaaaaa dd, yyyy hh:mm:ss   aaaaaaaaa dd, yyyy
                                                         hh:mm:ss.fffff
```

3	mm/dd/yy	mm/dd/yy hh:mm:ss	mm/dd/yy hh:mm:ss.ffffff
4	dd/mm/yy	dd/mm/yy hh:mm:ss	dd/mm/yy hh:mm:ss.ffffff
5	dd aaa yy	dd aaa yy hh:mm:ss	dd aaa yy hh:mm:ss.ffffff
6	aaa dd, yy	aaa dd, yy hh:mm:ss	aaa dd, yy hh:mm:ss.fffff
7	mm/dd/yyyy	mm/dd/yyyy hh:mm:ss	mm/dd/yyyy hh:mm:ss.ffffff
8	dd/mm/yyyy	dd/mm/yyyy hh:mm:ss	dd/mm/yyyy hh:mm:ss.ffffff

Для всех типов часть, относящуюся к дате, опускать нельзя. Если задается значение без указания времени, то по умолчанию устанавливается 00:00:00 (полночь). Если вводятся только две цифры года, то век основывается на переменной Empress MSDATELIMIT. Для Empress v8.*xx* и последующих ее значением по умолчанию служит 1950. В более ранних версиях оно равнялось 1900.

Для ввода даты в Empress можно использовать любой из девяти указанных форматов. Единственное ограничение состоит в невозможности ввести четыре цифры года в формат, использующий две цифры: при вводе даты в нем всегда используется MSDATELIMIT.

При выводе DBD::Empress использует yyyymmddhhmmssffffff, а DBD::EmpressNet использует yyyy-mm-dd hh:mm:ss.ffffff. Empress не позволяет изменить формат вывода по умолчанию. Значение дата/время нельзя отформатировать в другом стиле для вывода. Лучшим выходом является выбор составляющих даты/времени с помощью таких функций SQL, как DAYOF(d) и MONTHOF(d), с последующим форматированием в Perl.

Текущие дату и время на сервере можно получить с помощью псевдоконстант NOW и TODAY. NOW возвращает текущие дату и время. TODAY возвращает только дату.

Арифметические действия с датами и временем выполняются с помощью операторов даты/времени Empress. Например:

```
NOW + 2 MINUTES + 5 SECONDS
TODAY - 3 DAYS
```

Empress предоставляет большой набор функций дат, в том числе DAYOF(), MONTHOF(), YEAROF(), HOUROF(), MINUTEOF(), SECONDOF(), WEEKOFYEAR(), DAYNAME(), DAYOFWEEK(), DAYOFYEAR() и DATENEXT().

Следующее выражение SQL:

```
'1 jan 1970' + unix_time_field SECONDS
```

осуществляет преобразование в местное время, основываясь на 1 января 1970, но прямо привести к GMT нельзя.

Получить для местного часового пояса (не GMT) количество секунд, прошедших с 1 января 1970 года, с точностью до суток можно, используя выражение:

```
(date_field - '1 jan 1970') * 86400
```

Empress не производит автоматические настройки в соответствии с часовым поясом.

Обработка данных типа LONG/BLOB

РСУБД Empress поддерживает следующие типы данных LONG:

```
TEXT      Данные переменной длины из 7-битовых символов
NLSTEXT   Как TEXT, но допустимы 8-битовые символы
BULK      Интерпретируются пользователем (Byte Stream)
```

Обычно максимальный размер для всех этих типов равен 2**31–1 байт (2 Гбайт).

`LongReadLen` работает согласно определению в `DBD::EmpressNet`, но игнорируется в `DBD::Empress`. Максимальное значение `LongReadLen` обычно ограничено 2 Гбайт. `LongTruncOk` не реализован.

Для привязки типов данных LONG/BLOB не требуется специальных действий. Атрибут `TYPE` в данное время не используется при привязке параметров. Максимальный размер параметров в `bind_param()` ограничен возможностями операционной системы и размером типа `int` в C, в зависимости от того, что из них меньше.

Другие вопросы обработки данных

Метод `type_info()` не поддерживается.

Empress автоматически преобразует строки в числа и даты, а числа и даты в строки по мере необходимости.

Транзакции, изоляция и блокировка

`DBD::Empress` поддерживает транзакции. Уровнем изоляции по умолчанию является Serializable.

Другие уровни изоляции транзакций явным образом не поддерживаются. Однако поддерживается Read Uncommited на уровне отдельных запросов. Этот уровень включается добавлением параметра `BYPASS` к каждой команде SQL.

Например:

```
SELECT BYPASS * FROM table_name
```

По умолчанию осуществляется блокировка на уровне записи. Блокировка по чтению не блокирует другие блокировки по чтению, но блокирует запись, а блокировка по записи блокирует все остальные бло-

кировки. Блокировки по записи могут обходиться при чтении с помощью параметра BYPASS.

В режиме транзакции (AutoCommit отключен) выбранные строки автоматически блокируются для обновления, если только в команде SELECT не используется параметр BYPASS.

Для явной блокировки таблицы можно использовать команду LOCK TABLE table_name IN lock_mode. Режим блокировки может быть EXCLUSIVE или SHARE. SHARE требует, чтобы пользователь имел в отношении таблицы права SELECT или UPDATE. EXCLUSIVE требует, чтобы пользователь имел в отношении таблицы права UPDATE, INSERT или DELETE.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE чувствителен к регистру. Оператор MATCH не чувствителен к регистру.

Синтаксис соединения таблиц

Для создания внешних соединений перед таблицами, управляющими соединением, должно помещаться ключевое слово OUTER, например:

```
SELECT customer_name, order_date
FROM OUTER customers, orders
WHERE customers.cust_id = orders.cust_id;
```

Эта команда возвращает все строки таблицы customers, для которых нет соответствующих строк в таблице orders. Empress возвращает NULL для всех выражений в списке выборки, содержащих колонки таблицы orders.

Имена таблиц и колонок

Длина имен идентификаторов Empress, обозначающих таблицы и колонки, не может превышать 32 символов. Первым символом должна быть буква, но затем может следовать любая комбинация букв, цифр и символов подчеркивания (_). Имена таблиц/колонок в Empress хранятся так, как они заданы. Они чувствительны к регистру.

Таблицы и поля Empress могут содержать большую часть символов ASCII (кроме $ и ?), если они заключены в кавычки.

В базовом продукте могут использоваться символы ISO-Latin. Специальные продукты для других языков, например японского, могут обрабатывать соответствующие наборы символов.

Row ID

Для ссылки на идентификатор строки используется MS_RECORD_NUMBER. Во время выборки идентификатор строки можно рассматривать как

строку; однако при использовании в предложении WHERE его нужно рассматривать как целое число. Он используется только для явной выборки, неравенства недопустимы:

```
SELECT * FROM table_name WHERE MS_RECORD_NUMBER = ?
```

Автоматическая генерация ключей и последовательностей

В Empress нет механизма автоинкрементирования или системной генерации ключей. Генераторы последовательностей не поддерживаются.

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

Обновление и удаление по месту не поддерживаются.

Связывание параметров

Связывание параметров Empress поддерживается напрямую. Поддерживается только стандартный заполнитель ?.

DBD::Empress распознает атрибут bind_param() TYPE SQL_BINARY. Все остальные типы автоматически правильно связываются без использования TYPE. Неподдерживаемые типы игнорируются без выдачи предупреждения.

Хранимые процедуры

DBD::Empress не поддерживает хранимые процедуры явным образом. Неявная поддержка хранимых процедур возможна в командах SQL. Например:

```
$sth->prepare("SELECT func(attribute) FROM table_name");
```

Метаданные таблиц

DBD::Empress не поддерживает метод table_info().

Для получения подробных данных о колонках таблицы можно использовать системные таблицы SYS_ATTRS и SYS_TABLES. Например:

```
SELECT * FROM sys_attrs
WHERE attr_tabnum = (SELECT tab_number FROM sys_tables WHERE tab_name='x')
```

Однако для этого требуется наличие прав на выполнение SELECT в отношении этих системных таблиц.

В данное время нельзя получить через DBD::Empress подробные данные о ключах и индексах. Для получения этих данных можно интерпрети-

ровать содержание системных таблиц, хотя это сопряжено с некоторыми трудностями.

Специфические для драйверов атрибуты и методы

DBD::Empress не имеет существенных специфических атрибутов для дескрипторов драйвера и частных методов.

DBD::Informix

Общие сведения

Версия драйвера

DBD::Informix версия 0.62.

Краткая характеристика

Транзакции	Да, если разрешены при создании базы
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да, "?" (собственный)
Хранимые процедуры	Да
Связывание выходных значений	Да
Регистр символов в именах таблиц	Конфигурируем
Регистр символов в именах полей	Конфигурируем
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет
Псевдоколонка ROW ID в серверных таблицах	Да, "ROWID"
Обновление/удаление по месту	Да
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Джонатан Леффлер (Jonathan Leffler). С ним можно связаться через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль DBD::Informix поддерживает Informix OnLine и SE, начиная с версии 5.00. Есть некоторые ограничения поддержки IUS (также известной как IDS/UDO). Он использует Informix-ESQL/C (также известный как Informix ClientSDK). Для того чтобы иметь возможность скомпилировать код DBD::Informix, нужно иметь лицензию разработчика для Informix-ESQL/C (или версии C-кода для Informix-4GL).

Дополнительные данные можно получить на узлах:

http://www.informix.com

http://www.iiug.org

Отличия от спецификации DBI

Если изменить значение `AutoCommit` после подготовки команды, можно столкнуться с неожиданными проблемами, поэтому лучше этого не делать.

Читайте документацию по `DBD::Informix` для получения подробностей по этому и другим разнообразным тонким вопросам совместимости.

Синтаксис соединения

Формат `DBI->connect()` «Имя источника данных» (DSN) следующий:

```
dbi:Informix:connect_string
```

Здесь `connect_string` является любой допустимой строкой, которую можно передать команде Informix `CONNECT` (или команде `DATABASE` в версиях 5.*x*). В число допустимых записей входят:

```
dbase
dbase@server
@server
/path/to/dbase
//machine/path/to/dbase
```

Специфических для драйвера атрибутов метода `DBI->connect()` нет.

Если вы используете ESQL/C версии 6.00 или выше, то число дескрипторов баз данных ограничено только вашим воображением и физическими возможностями компьютера. В версии 5.*x* вы одновременно можете использовать только одно соединение.

Типы данных

Обработка числовых данных

Informix поддерживает следующие числовые типы данных:

```
INTEGER           - 32-битовое целое со знаком, исключая -2**31
SERIAL            - синоним INTEGER в отношении масштаба
SMALLINT          - 16-битовое целое со знаком,исключая -2**15
FLOAT             - C 'double'
SMALLFLOAT        - C 'float'
REAL              - синоним SMALLFLOAT
DOUBLE PRECISION  - синоним FLOAT
DECIMAL(s)        - s-значное число с плавающей точкой (базы данных не-ANSI)
DECIMAL(s)        - s-значное целое (базы данных MODE ANSI)
```

```
DECIMAL(s,p)    - s-значное число с фиксированной точкой
                  с p десятичными разрядами
MONEY(s)        - s-значное число с фиксированной точкой
                  с 2 десятичными разрядами
MONEY(s,p)      - s-значное число с фиксированной точкой
                  с p десятичными разрядами
NUMERIC(s)      - синоним DECIMAL(s)
NUMERIC(s,p)    - синоним DECIMAL(s,p)
INT8            - 64-битовое целое со знаком, исключая -2**63 (IDS/UDO)
SERIAL8         - синоним INT8 в отношении масштаба
```

DBD::Informix возвращает все числа как строки. Поэтому драйвер не накладывает ограничений на размер PRECISION или SCALE.

Обработка строковых данных

Informix поддерживает следующие строковые типы данных:

```
VARCHAR(size)
NVARCHAR(size)
CHAR
CHAR(size)
NCHAR
NCHAR(size)
CHARACTER VARYING(size)
NATIONAL CHARACTER VARYING(size)
NATIONAL CHARACTER(size)
CHARACTER(size)
VARCHAR(size,min)   -- и синонимы для этого типа
NVARCHAR(size,min)  -- и синонимы для этого типа
LVARCHAR            -- только в IDS/UDO
```

Возможно, здесь следовало также поместить большие объекты TEXT и BYTE, т. к. они автоматически преобразуются в строки и обратно.

Типы CHAR имеют ограничение длины 32 767 байтов в OnLine и IDS и несколько меньшее значение (32*5xx*) в SE. Для типов VARCHAR предел равен 255. Колонки LVARCHAR ограничены 2 Кбайт; когда они используются для передачи других типов данных, размер ограничен 32 Кбайт. В DBD::Informix 0.61 поддержка LVARCHAR работает не полностью.

Типы CHAR и NCHAR имеют фиксированную длину и дополняются пробелами.

Обработка национальных наборов символов зависит от версии базы данных (и различна для версий 5, версий 6 и 7.1*x* и версий, начиная с 7.2*x*). Детали для версии 8.*x* различаются в зависимости от *x*. Это зависит от местных установок, определяемых большим числом стандартных (например LANG, LC_COLLATE) и нестандартных (например DBNLS, CLIENT_LOCALE) переменных окружения. Подробности можно узнать в соответствующем руководстве. Кодировка Unicode в данное время в Informix не поддерживается (по данным на 28.02.99).

Конкатенация строк осуществляется с помощью оператора ||.

Обработка данных типа «дата»

Есть два основных типа для работы с датой/временем: DATE и DATETIME. DATE поддерживает даты в диапазоне от 01/01/0001 до 31/12/9999. Этот тип достаточно гибок в отношении формата ввода/вывода. Внутренне представляется количеством дней, истекших с 31 декабря 1899, поэтому 1 января 1900 является днем 1. Он не понимает изменения календаря в 1752, 1582–1584 годах и начале первого тысячелетия и устанавливает для этих ранних дат 1970-01-01.

DATETIME должен определяться двумя элементами из множества:

```
YEAR MONTH DAY HOUR MINUTE SECOND FRACTION FRACTION(n) для n = 1..5
```

Дата хранится согласно формату ISO 8601 для констант. Например, DATE("29/02/ 2000") эквивалентно:

```
DATETIME("2000-02-29") YEAR TO DAY,
```

Век для систем POSIX можно выразить следующим образом:

```
DATETIME(1970-01-01 00:00:00) YEAR TO SECOND
```

Прямой поддержки временных поясов нет.

Формат даты/времени по умолчанию зависит от местных установок среды окружения, версии и типа данных. Типы DATETIME строго придерживаются ISO 8601, за исключением преобразования одной или двух цифр года в четырехзначный эквивалент, что зависит от версии и окружения.

Обработка двузначных лет зависит от версии, исправленных ошибок и окружения. В целом (для текущей версии), если переменная окружения DBCENTURY не установлена или установлена в 'R', используется текущее столетие. Если DBCENTURY установлена в 'F', дата относится к следующему столетию; если DBCENTURY имеет значение 'P', то дата относится к предыдущему столетию; если DBCENTURY имеет значение 'C', то это будет ближайшая дата в 50-летнем интервале, исходя из текущего дня, месяца и года без учета времени суток.

Текущие дата и время возвращаются функцией CURRENT, обычно задаваемой как CURRENT YEAR TO SECOND.

Informix не предоставляет простых способов ввода и вывода даты и времени в других форматах. На эту тему можно написать многие главы.

Informix поддерживает черновую версию типа данных SQL2 INTERVAL:

```
INTERVAL start[(p1)] [TO end[(p2)]]
```

(где [ ] обозначает необязательные части.)

Интервалы могут быть заданы следующим образом:

```
YEAR, YEAR TO MONTH,
MONTH,
DAY, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND,
HOUR, HOUR TO MINUTE, HOUR TO SECOND,
MINUTE, MINUTE TO SECOND,
SECOND, FRACTION
```

p1 задает число цифр в самой старшей части величины, максимум – 9 и по умолчанию – 2 (кроме YEAR, имеющего по умолчанию 4 цифры). p2 задает число цифр в дробной части секунд, максимум – 5 и по умолчанию – 3.

Значения интервалов в литералах можно задавать с использованием следующего синтаксиса:

```
INTERVAL value start[(p1)] [TO end[(p2)]]
```

Например:

```
INTERVAL(2) DAY
INTERVAL(02:03) HOUR TO MINUTE
INTERVAL(12345:67.891) MINUTE(5) TO FRACTION(3)
```

Выражение «2 UNITS DAY» эквивалентно первой из приведенных строк. Аналогичные выражения можно использовать с любыми основными типами.

Над датами и интервалами можно производить полный набор операций, например, datetime–datetime=interval, datetime+interval=datetime, interval/number=interval.

Следующее выражение SQL можно использовать для преобразования целого числа «секунд после 1 января 1970 GMT» в соответствующее значение дата/время для базы данных:

```
DATETIME(1970-01-01 00:00:00) YEAR TO SECOND + seconds_since_epoch UNITS SECOND
```

Простого встроенного выражения для произведения обратных действий нет. Используйте хранимую процедуру; обратитесь к архивам *comp.databases.informix* на DejaNews или к серверу Informix International Users Group (IIUG) на *http://www.iiug.org*.

В Informix нет простого способа работы с несколькими часовыми поясами.

Обработка данных типа LONG/BLOB

В Informix поддерживаются следующие типы больших объектов:

```
BYTE  - двоичные данные    max 2 Гбайт
TEXT  - текстовые данные   max 2 Гбайт
BLOB  - двоичные данные    max 2 Гбайт (возможно, больше); только IDS/UDO
CLOB  - символьные данные  max 2 Гбайт (возможно, больше); только IDS/UDO
```

DBD::Informix в данное время не поддерживает типы данных BLOB и CLOB, но поддерживает типы BYTE и TEXT.

Атрибуты DBI LongReadLen и LongTruncOk не реализованы. Если выбранные данные имеют тип BYTE или TEXT, то они сохраняются в соответствующей переменной Perl и единственным ограничением является память, предел которой 2 Гбайт.

Максимальный размер параметра, используемого с bind_param() при вставке данных типа BYTE или TEXT, составляет 2 Гбайт. Для выборки или вставки не требуется особых приемов. UPDATE просто не работает.

Метод bind_param() не обращает внимания на атрибут TYPE. Передаваемая строка автоматически преобразуется в требуемый тип. Если это не строковый тип, то должна существовать возможность его преобразования системой. Использовать UPDATE в DBD::Informix с этими типами данных нельзя, только в версии 7.30 IDS обеспечивается возможность обработки больших объектов.

Другие вопросы обработки данных

Метод type_info() не поддерживается.

Как правило, типы, не являющиеся BLOB, автоматически конвертируются в строки и обратно. Informix поддерживает также автоматическое преобразование числовых типов между собой в тех случаях, когда оно корректно. Преобразование из DATETIME или INTERVAL в числовые типы данных не является автоматическим.

Транзакции, изоляция и блокировка

Базы данных Informix могут создаваться с поддержкой транзакций или без нее.

Informix поддерживает несколько уровней изоляции транзакций: REPEATABLE READ, CURSOR STABILITY, COMMITTED READ и DIRTY READ. Обратитесь к документации Informix, если вас интересует точный смысл этих уровней. Уровни изоляции имеются только в ONLINE и IDS и родственных им; в SE поддерживается только уровень, находящийся где-то посередине между COMMITTED READ и DIRTY READ.

Уровень изоляции, устанавливаемый по умолчанию, зависит от типа базы данных, с которой вы соединяетесь. Для изменения уровня изоляции можно воспользоваться командой SET ISOLATION TO *level*. Если в базе данных не ведется журнал (т. е. не поддерживаются транзакции), то уровень изоляции установить нельзя. В некоторых более новых версиях можно установить транзакцию в режим READ ONLY.

Принятый по умолчанию режим блокировки по чтению и записи зависит от уровня изоляции, определения таблицы и того, создавалась ли база данных с поддержкой транзакций.

Строки, возвращаемые командой SELECT, можно блокировать для предотвращения их изменения в другой транзакции путем добавления к команде слов FOR UPDATE. Можно также дополнительно задать список обновляемых колонок, указав его в скобках после предложения FOR UPDATE.

Для явной блокировки таблицы используется команда LOCK TABLE table_name IN lock_mode. Режим блокировки может быть SHARED или EXCLUSIVE. Существуют ограничения на то, когда можно разблокировать таблицы и когда можно применять блокировки. Блокировка строк/страниц производится с курсорами FOR UPDATE. В некоторых типах баз данных некоторые курсоры неявно создаются в режиме FOR UPDATE.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE чувствителен к регистру.

Синтаксис соединения таблиц

Во всех версиях Informix поддерживается базовый стиль задания соединений WHERE a.field = b.field. Наличие поддержки записи соединений в стиле SQL-92 зависит от версии РСУБД и обычно отсутствует.

Внешние соединения поддерживаются. Основная форма такая:

```
SELECT * FROM A, OUTER B WHERE a.col1 = b.col2
```

При этом будут выбраны все строки из A. Если в B есть строки, соответствующие строкам A по условию соединения, они также выводятся. Если в B нет соответствующих строк, то в *B*-колонках списка SELECT возвращается NULL. Есть и другие особенности, такие как осложнения с критериями в предложении WHERE или вложенными внешними соединениями.

Имена таблиц и колонок

В большинстве версий максимальный размер имени таблицы или колонки составляет 18 символов в соответствии с SQL-86. В новейших версиях (Centaur, временно 9.2 или 7.4), размер равен 128, как это требуется в SQL-92. Длина имен владельцев (схем) может составлять 8 символов в старых версиях и 32 в версиях с длинными именами таблиц/колонок.

Первым символом должна быть буква, но затем может следовать любая комбинация букв, цифр и символов подчеркивания (_).

Если установлена переменная окружения DELIMIDENT, то имена таблиц, колонок и владельцев можно заключать в двойные кавычки, при этом становятся допустимыми любые символы. Для того чтобы вставить в

имя двойную кавычку, используйте смежные двойные кавычки, например `"I said, ""Don't"""`. (Обычно Informix достаточно вольно рассматривает двойные и одиночные кавычки как эквивалентные, поэтому часто можно с таким же успехом написать: `'I said, "Don''t"'`. Однако если установлена переменная `DELIMIDENT`, следует проявлять большую осторожность.) Имена владельцев для большей надежности следует заключать в двойные кавычки.

Режим сохранения и различения регистра в именах таблиц и колонок зависит от окружения и механизма использования кавычек.

Поддержка наборов национальных символов в именах зависит от версии и окружения (местных установок).

Row ID

В большинстве таблиц есть виртуальная колонка ROWID, которую можно выбирать. В таблицах, разбитых на куски, ее нет, если только при создании таблицы или изменении ее структуры не указывалось предложение `WITH ROWIDS`. Тогда это физическая колонка ROWID, которая в других случаях является виртуальной (т. е. команда `SELECT *` ее не видит).

Как и любой другой тип, за исключением `BLOB`, ROWID можно преобразовать в строку и в таком виде использовать. Обратите внимание, что значения ROWID не обязаны быть смежными или начинаться с нуля либо единицы.

Автоматическая генерация ключей и последовательностей

Типы данных SERIAL и SERIAL8 являются «автоинкрементируемыми ключами». Если вы вставляете в такую колонку ноль, то ей присваивается очередное неиспользованное значение ключа, и откат этого произвести нельзя. Нельзя использовать NULL, необходимо вставлять ноль. При вставке в колонку ненулевого значения используется это значение. Обычно на колонку накладывается ограничение уникальности, чтобы предотвратить появление повторяющихся значений.

Чтобы узнать, какое значение было только что вставлено, можно использовать синтаксис:

```
$sth->{ix_sqlerrd}[1]
```

Informix не поддерживает непосредственно генераторы последовательностей, но для этого можно использовать собственные хранимые процедуры.

Автоматическая нумерация строк и предельное число строк

Informix не поддерживает автоматической нумерации возвращаемых строк.

В некоторых последних версиях Informix поддерживается директива FIRST, ограничивающая число возвращаемых строк в команде SELECT:

```
SELECT FIRST num_of_rows ...
```

Обновление и удаление по месту

Обновление и удаление по месту поддерживаются с использованием синтаксиса WHERE CURRENT OF. Например:

```
$dbh->do("UPDATE ... WHERE CURRENT OF $sth->{CursorName}");
```

Связывание параметров

В Informix связывание параметров обеспечивается непосредственно. Поддерживаются только стандартные заполнители ?.

Поддержка атрибута TYPE для bind_param() в настоящее время отсутствует, но ожидается в следующей версии.

Хранимые процедуры

Некоторые хранимые процедуры можно использовать как функции в обычном SQL :

```
SELECT proc1(Col1) FROM SomeTable WHERE Col2 = proc2(Col3);
```

Все хранимые процедуры могут выполняться с помощью команды SQL EXECUTE PROCEDURE. Если процедура не возвращает значений, ее можно просто выполнить. Если процедура возвращает хотя бы одно значение через оператор RETURN, ее можно рассматривать как команду SELECT. Поэтому после вызова execute() можно делать выборку из дескриптора команды, как если бы была выполнена команда SELECT, например:

```
$sth = $dbh->prepare("EXECUTE PROCEDURE CursoryProcedure(?,?)");
$sth->execute(1, 12);
$ref = $sth->fetchall_arrayref();
```

Метаданные таблиц

В настоящее время метод DBI table_info() не поддерживается. Для получения списка всех таблиц или его подмножества можно использовать частный метод _tables().

Характеристики колонок таблицы можно получать с помощью частного метода _columns().

Ключи и индексы таблицы можно получить из запросов к системному каталогу.

Дополнительные сведения по этому и другим вопросам можно получить из телеконференции *comp.databases.informix* и через междуна-

родную группу пользователей International Informix User Group (IIUG) на *http://www.iiug.org*.

Специфические для драйверов атрибуты и методы

За подробностями специфических для драйвера атрибутов дескрипторов баз данных и команд обратитесь к документации по `DBD::Informix`.

Доступ к описанию таблиц и колонок легко получить с помощью частных методов `_tables()` и `_columns()`.

Другие существенные характеристики базы данных и драйвера

Во время сеанса работы с базой данных можно создавать временные таблицы, автоматически удаляемые по окончании сеанса, если это не было сделано раньше явным образом. Это очень удобная функция.

В последних версиях Informix (IDS/UDO, IUS) поддерживаются определяемые пользователем подпрограммы и типы данных, которые могут быть реализованы на сервере с использованием C или в последнее время Java.

Синтаксис SQL-92 «CASE WHEN» некоторыми версиями серверов Informix поддерживается. Благодаря этому значительно упрощаются некоторые виды запросов.

DBD::Ingres

Общие сведения

Версия драйвера

`DBD::Ingres` версия 0.16 и, где это отмечено, релиз 0.20.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да, "?" и ":1" (собственные)
Хранимые процедуры	Да
Связывание выходных значений	Да
Регистр символов в именах таблиц	Не учитывается, хранятся в верхнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки

```
Нечувствительный к регистру оператор "LIKE"   Нет
Псевдоколонка ROW ID в серверных таблицах     Да, "tid"
Обновление/удаление по месту                  Нет
Одновременное использование нескольких        Не ограничено
дескрипторов
```

Автор и как с ним связаться

Автор драйвера – Хенрик Тугарт (Henrik Tougaard). С ним можно связаться через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль `DBD::Ingres` поддерживает Ingres 6.4 и OpenIngres (1.*x* и II). Дополнительные сведения об Ingres можно получить на странице:

http://www.cai.com/products/ingres.htm

Отличия от спецификации DBI

Подготовленные команды нельзя использовать в разных транзакциях, поскольку commit/rollback закрывают и делают недействительными все подготовленные команды. Проводится работа по устранению этого недостатка.

Синтаксис соединения

Синтаксис `DBI->connect()` «Имя источника данных» (DSN) может быть следующим:

```
dbi:Ingres:dbname
dbi:Ingres:vnode::dbname
dbi:Ingres:dbname;options
```

Здесь `options` являются дополнительными флагами SQL, как определено в *CA-OpenIngres System Reference Guide*.

Специфических для драйвера атрибутов метода `DBI->connect()` не существует.

`DBD::Ingres` поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

База данных и драйвер поддерживают одно-, двух- и четырехбайтовые целые (`INTEGER`), четырех- и восьмибайтовые числа с плавающей запятой (`FLOAT`) и денежный тип. База данных и драйвер (начиная с версии 0. 20) поддерживают числовой тип `DECIMAL`.

Тип	Описание	Диапазон
INTEGER1	1-байтовое целое	от −128 до +127
SMALLINT	2-байтовое целое	от −32 678 до +32 767
INTEGER	4-байтовое целое	от −2 147 483 648 до +2 147 483 647
FLOAT4	4-байтовое с плавающей точкой	от −1,0e+38 до 1,0e+38 (7 цифр)
FLOAT	8-байтовое с плавающей точкой	от −1,0e+38 до 1,0e+38 (16 цифр)
MONEY	8-байтовый денежный	от $−999 999 999 999,99 до $999 999 999 999,99
DECIMAL	число с фиксированной точкой	Зависит от точности (максимум 31) и масштаба

`DBD::Ingres` всегда возвращает числа в виде чисел Perl, причем по возможности целых либо с плавающей точкой. Поэтому возможна потеря точности при выборке типа `DECIMAL` с точностью, большей чем у чисел Perl (обычно 16). В этом случае можно преобразовать значение в строку в выражении, используемом в `SELECT`.

Обработка строковых данных

Ingres и `DBD::Ingres` поддерживают следующие строковые типы данных:

```
VARCHAR(size)
CHAR(size)
TEXT(size)
C(size)
```

Для всех строковых типов максимальная длина составляет 2000 байтов. Типы `CHAR`, `TEXT` и `C` имеют фиксированную длину и дополняются пробелами.

Во всех строковых типах могут использоваться национальные наборы символов. Тип `C` принимает только изображаемые символы. `CHAR` и `VARCHAR` могут содержать любые символы, в том числе символы `nul` (`"\0"`) в середине строки. Unicode формально пока не поддерживается.

Конкатенация строк производится с помощью оператора SQL `+`.

Обработка данных типа «дата»

В Ingres всего один тип данных для дат: `DATE`. Однако в нем могут содержаться как абсолютная дата и время, так и временной интервал. Дата и время имеют разрешение в одну секунду в интервале примерно

от 1 января 1581 года до 31 декабря 2382 года. Интервалы хранятся с разрешением в одну секунду.

Ingres поддерживает ряд форматов даты в зависимости от установки системной переменной II_DATE_FORMAT. Формат вывода по умолчанию американский (US): DD-MMM-YYYY HH:MM:SS.

Допустимы несколько форматов ввода. В качестве формата по умолчанию могут использоваться: MM/DD/YYYY, DD-MMM-YYYY, MM-DD-YYYY, YYYY.MM.DD, YYYY_MM_DD, MMDDYY, MM-DD и MM/DD.

При задании значения DATE без указания времени устанавливается время по умолчанию 00:00:00 (полночь). При задании значения DATE без указания даты датой по умолчанию является первое число текущего месяца. Если в формате даты год указан двумя цифрами, например YY в DD-MON-YY (часто устанавливается по умолчанию), то дата всегда принадлежит текущему столетию.

Поддерживаются следующие функции работы с датами:

```
DATE(string)              - преобразует строку в дату
DATE_TRUNC(unit, date)    - значение даты обрезается по указанную
                            составляющую
DATE_PART(unit, date)     - целое, содержащее указанную часть
DATE_GMT(date)            - преобразует дату в строку
                            "YYYY_MM_DD HH:MM:SS GMT"
INTERVAL(unit, interval)  - выражает интервал числом указанных единиц
```

Текущие дату и время возвращает функция DATE('now'). Текущую дату возвращает функция DATE('today').

Следующее выражение SQL можно использовать для преобразования целого числа секунд после 1 января 1970 GMT в соответствующее значение дата/время в базе данных:

```
DATE('01.01.1970 00:00 GMT')+DATE(CHAR(seconds_since_epoch)+' seconds')
```

а для обратного преобразования использовать:

```
INT4(INTERVAL('seconds', DATE('now')-DATE('01.01.1970 00:00 GMT')))
```

К дате может прибавляться трехбуквенное обозначение временного пояса (из ограниченного набора). Если временной пояс не указан, предполагается, что это временной пояс текущего *клиента*. Все значения дата/время хранятся в базе данных как GMT и преобразуются в местное время *клиента*, осуществляющего выборку данных. Все сравнения дат на сервере производятся в GMT.

Обработка данных типа LONG/BLOB

Ingres поддерживает следующие типы LONG:

```
LONG VARCHAR  - символьные данные переменной длины до 2 Гбайт
LONG BYTE     - произвольные двоичные данные переменной длины до 2 Гбайт
```

Однако драйвер DBD::Ingres эти типы пока не поддерживает.

Другие вопросы обработки данных

Драйвер DBD::Ingres поддерживает метод type_info().

Ingres поддерживает автоматическое преобразование типов данных, в случае, если оно корректно.

Транзакции, изоляция и блокировка

Ingres поддерживает транзакции. По умолчанию установлен уровень изоляции транзакций Serializable. OpenIngres II поддерживает уровни изоляции Repeatable Read, Read Commited и Serializable.

Чтение записи устанавливает блокировку по чтению, которая не позволяет производящим запись в базу данных изменять эту запись, а в зависимости от глубины блокировки, возможно, и другие записи. Другие чтения при этом производятся беспрепятственно. При записи устанавливается блокировка, не позволяющая другим ни писать, ни читать.

Команда SET LOCKMODE позволяет изменить глубину блокировки. Ее можно установить в следующие значения:

```
ROW    - блокировка только задействованных строк (только OpenIngres II)
PAGE   - блокировка страницы, содержащей задействованную строку
TABLE  - блокировка всей таблицы
```

С помощью команды SET LOCKMODE SESSION WHERE READLOCK=NOLOCK можно, но *не* рекомендуется, установить уровень изоляции в Read Uncommited.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE чувствителен к регистру.

Синтаксис соединения таблиц

OpenIngres поддерживает внешние соединения согласно синтаксису ANSI SQL-92. Ingres 6.4 не поддерживает внешних соединений.

Имена таблиц и колонок

Имена идентификаторов не должны превышать в длину 32 символов. Первым символом должна быть буква или символ подчеркивания (_), остальные могут быть любой комбинацией букв, цифр, символов ($), (#) и (@).

Однако если идентификатор заключен в двойные кавычки ("), он может содержать любые допустимые символы, в том числе и пробелы, но

не двойные кавычки. В Ingres 6.4 подобное именование не поддерживается.

Различение верхнего и нижнего регистра определяется установками, сделанными при инсталляции Ingres.

Использование национальных символов в идентификаторах допускается, если они заключены в двойные кавычки.

Row ID

В Ingres псевдоколонка «ID-строка» называется `tid`. Это целое число, которое можно использовать без специальной обработки, например:

```
SELECT * FROM table WHERE tid=1029;
```

Автоматическая генерация ключей и последовательностей

OpenIngres II поддерживает колонки «логических ключей». Они определяются с помощью специального типа данных `TABLE_KEY WITH SYSTEM MAINTAINED`. В Ingres 6.4 поддержка этой функции предоставлялась за отдельную плату.

Колонка может быть определена как `TABLE_KEY` или `OBJECT_KEY`. `Table_key` являются уникальными в таблице, а `object_key` уникальны в базе данных в целом.

`DBD::Ingres` в настоящее время не может показать значение последнего введенного автоматического ключа, хотя в будущем такая поддержка может быть включена, если об этом вежливо попросит достаточно большое количество людей или кто-то сам напишет код.

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

Обновление и удаление по месту поддерживаются в `DBD::Ingres` версии 0.20 посредством синтаксиса `WHERE CURRENT OF`. Например:

```
$dbh->do("UPDATE ... WHERE CURRENT OF $sth->{CursorName}");
```

`CursorName` автоматически определяется `DBD::Ingres` для каждой подготовленной команды.

Связывание параметров

Связывание параметров Ingres поддерживается непосредственно. Поддерживается только стандартный заполнитель `?`.

При использовании метода bind_param() можно с помощью атрибута TYPE определить обычные целые, с плавающей запятой и символьные типы. Для неподдерживаемых значений атрибута TYPE выводится предупреждение.

Хранимые процедуры

Вызов хранимых процедур осуществляется с помощью команды execute procedure. Например:

```
$dbh->do("execute procedure my_proc(param1='value')");
```

Возврат результатов пока невозможен.

Метаданные таблиц

DBD::Ingres версии 0.20 поддерживает метод table_info().

Каталог IICOLUMNS содержит данные обо всех колонках таблицы.

Каталог IIINDEXES содержит подробные сведения обо всех индексах базы данных, по одной строке для каждого индекса. Каталог IIINDEX_COLUMNS содержит сведения о колонках, образующих каждый индекс.

Первичные ключи отмечены в поле key_sequence каталога IICOLUMNS.

Специфические для драйверов атрибуты и методы

В DBD::Ingres нет специфических для драйвера атрибутов дескриптора базы данных. Однако при этом поддерживается несколько атрибутов дескриптора команды. Каждый из них возвращает ссылку на массив значений, по одному для каждой колонки в результатах выборки. Вот эти атрибуты:

ing_type

: 'i' для целочисленных колонок, 'f' для чисел с плавающей точкой и 's' для строк.

ing_ingtype

: Числовой тип Ingres для колонок.

ing_length

: Размер колонки в Ingres (используемый в базе данных).

DBD::Ingres поддерживает один частный метод:

get_dbevent()

: Этот частный метод вызывает GET DBEVENT и INQUIRE_INGRES для выборки незавершенного события базы данных. При вызове без аргумента вызывается блокирующий GET DBEVENT WITH WAIT. Задание числового аргумента приводит к вызову GET DBEVENT WITH WAIT= :seconds.

DBD::InterBase

Общие сведения

Версия драйвера

DBD::InterBase версия 0.021.

Эта версия драйвера DBD::InterBase является просто оболочкой Perl вокруг модуля IBPerl. Автор работает над прямой версией XS, поэтому обязательно прочтите новейшую документацию.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 4 Гбайт
Доступность атрибутов дескриптора команды	После выборки первой строки
Заполнители	Да, "?" (собственный)
Хранимые процедуры	Да
Связывание выходных значений	Да
Регистр символов в именах таблиц	Не учитывается, хранятся в верхнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет
Псевдоколонка ROW ID в серверных таблицах	Нет
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Эдвин Прамото (Edwin Pratomo). С ним можно связаться через список рассылки *dbi-users* или по *ed.pratomo@computer.org*.

Поддерживаемые версии и варианты баз данных

DBD::InterBase использовался для доступа к InterBase 4.0 под Linux и InterBase 5.5 под NT, но должен также работать с любыми версиями InterBase после 3.3, поддерживаемыми IBPerl. DBD::InterBase унаследовал все ограничения IBPerl 0.7, в том числе отсутствие метаданных.

За дополнительными сведениями по InterBase обращайтесь по адресам:

http://www.interbase.com

http://www.interbase.com/products/dsqlsyntax.html

Отличия от спецификации DBI

DBD::InterBase не имеет доступа к метаданным команды, пока команда не выполнена и не *выбрана ее первая строка*. Поэтому такие атрибуты, как $sth->{NUM_OF_FIELDS}, недоступны, пока не выполнен $sth->execute() и не вызван метод выборки. Будем надеяться, что в следующую версию будут внесены необходимые исправления.

Синтаксис соединения

Формат DBI->connect() «Имя источника данных» (DSN) следующий:

```
dbi:InterBase:attrs
```

где attrs является списком пар *ключ=значение*, разделяемых двоеточием. Допустимо использование следующих атрибутов:

database

Задает полный путь на сервере к базе данных, которая должна быть сделана базой данных по умолчанию.

host (необязательный)

Задает имя узла сервера InterBase, с которым нужно соединиться. По умолчанию это localhost (локальный сервер).

role (необязательный)

Задает имя роли SQL; поддерживается только в InterBase 5.0 и выше.

charset (необязательный)

Задает набор символов клиента, который нужно использовать. Полезен, если набор символов клиента отличается от используемого сервером. При использовании этого параметра включается автоматическое преобразование символов из одного набора в другой. Значение по умолчанию – NONE.

DBD::InterBase поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

InterBase поддерживает числовые типы INTEGER, SMALLINT, FLOAT, DOUBLE PRECISION, NUMERIC(p,s) и DECIMAL(p,s).

FLOAT и INTEGER всегда представляются 32 битами, а SMALLINT является 16-битным. DOUBLE PRECISION зависит от платформы, но обычно имеет размер 64 бита. Точность для NUMERIC/DECIMAL находится в пределах от 1 до 15, а масштаб находится в пределах от 1 до 15.

DBD::InterBase всегда возвращает числа в виде строк.

Обработка строковых данных

InterBase поддерживает следующие строковые типы данных:

```
CHAR(размер)      фиксированной длины с дополнением пробелами
VARCHAR(размер)   переменной, но ограниченной длины
```

Размер может составлять от 1 до 32 767 байт. Можно также задать набор символов для каждого поля, например:

```
CHAR(size) CHARACTER SET "ISO8859_1"
VARCHAR(size) CHARACTER SET "ISO8859_1"
```

InterBase поддерживает также `NCHAR(size)` и `NCHAR(size) VARYING` в качестве псевдонимов для `CHARACTER SET "ISO8859_1"`.

Обработка данных типа «дата»

В InterBase поддерживается один гибкий тип данных `DATE`, в котором можно хранить дату, время или одновременно дату и время. Дата, которую нужно сохранить как тип `DATE`, должна иметь формат `DD MON YYYY HH:MM:SS` или `DD-MON-YYYY HH:MM:SS`. Части `DD` и `MON` должны быть определены явно, остальные части по умолчанию устанавливаются равными текущим значениям года/времени.

Части `DD MON YYYY` могут принимать любые значения от 1 января 100 года до 29 февраля 32 768 года. `HH:MM:SS` находится в диапазоне от 00:00:00 до 23:59:59.

Год должен указываться четырьмя цифрами. Если заданы только две цифры, InterBase определит год, используя алгоритм скользящего окна: вычесть введенные две последние цифры года из последних двух цифр текущего года; если абсолютное значение разности больше 50, то первыми двумя цифрами становятся 20, в противном случае – 19.

Выбираемые из базы значения типа `DATE` форматируются с использованием строки формата `strftime()`. Эту строку можно задать в атрибуте `DateFormat` при вызове метода `prepare()`. Если атрибут не задан, в качестве строки формата используется «`%c`», например:

```
$stmt = "SELECT * FROM BDAY";
$opt = { 'DateFormat' => "%d %B %Y" };
$array_ref = $dbh->selectall_arrayref($stmt, $opt);
```

InterBase не поддерживает непосредственно определенные в SQL-92 типы данных `DATE`, `TIME` и `TIMESTAMP`.

В InterBase используются следующие литералы дат: `NOW`, `TODAY`, `YESTERDAY` и `TOMORROW`. Например:

```
CREATE TABLE SALES (
    ORDER_ID INTEGER NOT NULL,
    SHIP_DATE DATE DEFAULT "NOW" NOT NULL,
PRIMARY KEY(ORDER_ID));
```

Обработка данных типа LONG/BLOB

InterBase поддерживает тип данных BLOB. DBD::InterBase может обрабатывать поля BLOB размером до 4 Гбайт, если, конечно, в вашей машине есть такой объем памяти.

Для колонки BLOB может быть установлен двоичный или текстовый тип данных, а для текстового типа также указан набор символов. Данные BLOB хранятся в *сегментах*, размер которых (до 64 Кбайт) также может быть установлен для каждой колонки BLOB.

Другие вопросы обработки данных

InterBase поддерживает автоматическое преобразование типов данных там, где оно обосновано.

Транзакции, изоляция и блокировка

InterBase поддерживает транзакции. Уровень изоляции транзакций можно изменить командой SET TRANSACTION ISOLATION LEVEL x. Полное описание можно найти в руководстве InterBase DSQL.

Строки, возвращаемые колонкой SELECT, можно заблокировать для изменений другими транзакциями, добавив к команде выборки предложение FOR UPDATE. Дополнительно после предложения FOR UPDATE можно указать в скобках список блокируемых колонок.

Явной команды LOCK TABLE нет.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE чувствителен к регистру символов.

Синтаксис соединения таблиц

Поддерживаются внешние и внутренние соединения, записываемые на SQL в стандарте ISO.

Имена таблиц и колонок

Максимальный размер имен таблиц и колонок не может превышать 31 символ. Допустимо использование только буквенно-цифровых символов, первым символом должна быть буква.

InterBase преобразует все идентификаторы в верхний регистр.

Row ID

Понятие «идентификатора строки» отсутствует.

Автоматическая генерация ключей и последовательностей

Механизм создания уникальных последовательных чисел, автоматически вставляемых в таких операциях SQL, как INSERT, UPDATE, носит название GENERATOR. Например:

```
CREATE GENERATOR generator_name
SET GENERATOR generator_name TO integer_value
```

где integer_value является целым числом в диапазоне от −2**31 до 2**31 − 1. Команда SET GENERATOR устанавливает начальное значение заново создаваемого генератора или переустанавливает значение уже существующего.

Для использования генератора следует вызвать функцию InterBase GEN_ID. Например:

```
INSERT INTO SALES (PO_NUMBER) VALUES (GEN_ID(generator_name, step))
```

Команды DROP GENERATOR не существует; вот как можно удалить GENERATOR:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = 'generator_name'
```

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

InterBase не поддерживает обновления и удаления по месту.

Связывание параметров

Связывание параметров InterBase поддерживает непосредственно. Драйвер DBD::InterBase поддерживает заполнители в стиле ?.

Атрибут TYPE в методе bind_param(), а также метод type_info() в настоящее время не поддерживаются.

Хранимые процедуры

InterBase поддерживает хранимые процедуры, но пока ни DBD::InterBase, ни IBPerl их не поддерживают.

Метаданные таблиц

DBD::InterBase пока не поддерживает метод table_info().

Специфические для драйверов атрибуты и методы

Существенных специфических для драйвера атрибутов дескриптора базы данных в DBD::InterBase не сущетвует.

DBD::mysql и DBD::mSQL

Общие сведения

Версия драйвера

DBD::mysql и DBD::mSQL версий 1.20*xx* и 1.21_*xx*.

Версии 1.20*xx* (четные номера) являются стабильными, и изменения касаются только исправления ошибок и переносимости. Версии 1.21_*xx* (нечетные номера) используются для разработки драйверов: все новые функции и изменения интерфейса производятся в этой линии, пока она не станет в конце концов 1.22*xx*.

Краткая характеристика

Транзакции	Нет
Блокировка	Да, явная (только MySQL)
Соединения таблиц	Да, внутренние и внешние (для mSQL только внутренние)
LONG/LOB-типы данных	Да, до 4 Гбайт
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?" (эмулируется)
Хранимые процедуры	Нет
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Зависит от файловой системы, хранятся как определены
Регистр символов в именах полей	Нечувствителен/чувствителен (MySQL/mSQL), хранятся как определены
Преобразование недопустимых имен	Нет
Нечувствительный к регистру оператор "LIKE"	Возможны варианты, см. ниже
Псевдоколонка ROW ID в серверных таблицах	Да, "_rowid" (только в mSQL)
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Йохан Видман (Jochen Wiedmann). Связаться с ним можно через список рассылки *Msql-Mysql-modules@lists.mysql.com*.

Поддерживаемые версии и варианты баз данных

MySQL и mSQL являются небольшими эффективными и свободно распространяемыми серверами баз данных. MySQL имеет богатый набор характеристик, в то время как mSQL – самый минимальный.

Драйвер DBD::mysql версии 1.20*xx* поддерживает все версии MySQL, начиная примерно с 3.20. Драйвер DBD::mysql версии 1.21_*xx* поддерживает MySQL 3.22 и более поздние.

Драйверы DBD::mSQL версий 1.20*xx* и 1.21_*xx* поддерживают все версии mSQL вплоть до mSQL 2.0.*x*.

Здесь можно получить дополнительные сведения о MySQL:

http://www.mysql.com/

Здесь можно получить дополнительные сведения о mSQL:

http://www.blnet.com/msqlpc

http://www.hughes.com.au/

Отличия от спецификации DBI

Оба драйвера – DBD::mysql и DBD::mSQL – не производят полного разбора команды до ее выполнения. Такие атрибуты, как $sth->{NUM_OF_FIELDS}, недоступны до вызова $sth->execute(). Это нормальный режим, но важно помнить о нем при переносе приложений, написанных изначально для других драйверов.

Учтите также, что многие атрибуты команд становятся недоступными после выборки всех результатов или вызова метода finish().

Синтаксис соединения

Формат DBI->connect() «Имя источника данных» (DSN) может быть следующим:

```
DBI:mysql:attrs
DBI:mSQL:attrs
```

где attrs является списком пар *ключ=значение*, разделяемых двоеточием. Допустимо использование следующих атрибутов:

database=$database

Имя базы данных, с которой вы хотите соединиться.

host=$host

Имя машины, на которой выполняется сервер для базы данных, с которой вы хотите соединиться, по умолчанию *localhost*.

msql_configfile=$file

Загрузить из указанного файла специфические для драйвера установки, по умолчанию *InstDir/msql.conf*.

mysql_compression=1

Для низкоскоростных соединений может оказаться желательным сжимать данные при передаче между клиентом и ядром. Если ядро MySQL поддерживает сжатие, его можно включить данным атрибутом. По умолчанию сжатие отключено.

Специфических для драйвера атрибутов метода connect() не существует. Количество дескрипторов баз данных и команд зависит только от объема памяти. Ограничений на их одновременное использование нет.

Типы данных

Обработка числовых данных

В MySQL есть пять размеров целых типов данных, каждый из которых может быть со знаком (по умолчанию) или без знака (при добавлении слова UNSIGNED после имени типа).

Имя типа	Кол-во бит	Диапазон со знаком	Диапазон без знака
TINYINT	8	–128..127	0..255
SMALLINT	16	–32768..32767	0..65535
MEDIUMINT	24	–8388608..8388607	0..16777215
INTEGER	32	–2147483648..2147483647	0..4294967295
BIGINT	64	–(2*63)..(2**63-1)	0..(2**64)

Тип INT используется как псевдоним для INTEGER. Другими псевдонимами служат INT1=TINYINT, INT2=SMALLINT, INT3=MEDIUMINT, INT4=INT, INT8=BIGINT и MIDDLEINT=MEDIUMINT.

Обратите внимание, что арифметические операции производятся над величинами BIGINT или DOUBLE со знаком, поэтому не следует использовать целые без знака, которые больше самого большого целого со знаком (кроме как в битовых операциях). В операциях -, + и * используется арифметика с BIGINT, если оба аргумента имеют тип INTEGER. Это значит, что при перемножении больших целых чисел (или результатов, возвращаемых целочисленными функциями) можно получить неожиданные значения, если результат превысит 9223372036854775807.

MySQL имеет три основных типа данных для чисел, не являющихся целыми: FLOAT, DOUBLE и DECIMAL. Можно также использовать псевдонимы FLOAT4 для FLOAT и FLOAT8 для DOUBLE.

В последующем изложении буква M используется для обозначения *максимального размера отображения*, или *PRECISION* в терминологии ODBC и DBI. Буква D используется для обозначения числа знаков, которые могут следовать за десятичной точкой. (SCALE – масштаб в терминологии ODBC и DBI.)

Максимальный размер отображения (PRECISION) и количество знаков в дробной части (SCALE) обычно не требуются. Например, если вы просто используете «DOUBLE», то будут применены значения по умолчанию.

DOUBLE(M,D)

Числа с плавающей точкой обычного размера (с двойной точностью). Допустимы значения от –1,7976931348623157e+308 до –2,2250738585072014e–308, 0 и от 2,225073858 5072014e–308 до 1,7976931348623157e+308.

REAL и DOUBLE PRECISION используются как псевдонимы DOUBLE.

FLOAT(M,D)

Маленькое (одинарной точности) число с плавающей точкой. Допустимы значения от –3. 402823466e+38 до –1.175494351e-38, 0 и от –1.175494351e–38 до 3.40282346 6e+38.

FLOAT(M)

Число с плавающей точкой. Точность (M) может равняться 4 или 8. FLOAT(4) является числом с одинарной точностью, а FLOAT(8) – с двойной. Эти типы похожи на описанные выше FLOAT и DOUBLE. FLOAT(4) и FLOAT(8) имеют тот же диапазон значений, что и соответствующие типы FLOAT и DOUBLE, но их размер отображения и количество десятичных знаков не определены.

DECIMAL(M,D)

Тип DECIMAL является неупакованным числовым типом с плавающей точкой. NUMERIC является псевдонимом для DECIMAL. Он ведет себя как колонка CHAR; «неупакованный» означает, что число хранится в виде строки, по одному символу для каждой цифры и десятичной точки. Если D равно 0, то у величин не будет десятичной точки или дробной части. Максимальный размер DECIMAL такой же, как у DOUBLE, но фактический диапазон значений в колонке DECIMAL может быть ограничен путем выбора M и D.

NUMERIC может использоваться как псевдоним для DECIMAL.

В mSQL используется значительно меньшее количество числовых типов:

INTEGER соответствует типу INTEGER MySQL.

UINT соответствует типу INTEGER UNSIGNED MySQL.

REAL соответствует типу REAL MySQL.

Все типы данных, включая числа, возвращаются драйвером в виде строк. Поэтому нет ограничений на величину PRECISION и SCALE.

Обработка строковых данных

MySQL поддерживает следующие строковые типы, как указано в *mysql.info*; здесь *M* обозначает максимальный отображаемый размер или PRECISION:

`CHAR(M)`

Строка фиксированной длины, всегда дополняемая справа пробелами. `M` может находиться в диапазоне от 1 до 255 символов.

`VARCHAR(M)`

Строка переменной длины. Замыкающие пробелы удаляются базой данных при записи значения, что отличается от спецификации ANSI SQL. `M` может находиться в диапазоне от 1 до 255 символов.

`ENUM('value1','value2',...)`

Перечисление. Строковый объект, у которого может быть только одно значение, выбранное из заданного списка (или NULL). В перечислении может быть до 65 535 различных значений.

`SET('value1','value2',...)`

Множество. Строковый объект, у которого может быть несколько значений (или ни одного), каждое из которых должно выбираться из указанного списка. SET может иметь максимум 64 элемента.

Длина типов `CHAR` и `VARCHAR` ограничена 255 байтами. Все строковые типы поддерживают двоичные символы, включая NUL. (Для литеральных строк используйте метод `$dbh->quote()`.)

Поддерживаются также следующие псевдонимы:

```
BINARY(num)      CHAR(num) BINARY
CHAR VARYING     VARCHAR
LONG VARBINARY   BLOB
LONG VARCHAR     TEXT
VARBINARY(num)   VARCHAR(num) BINARY
```

В `DBD::mysql` атрибут `ChopBlanks` всегда включен. Ядро MySQL само удаляет пробелы, находящиеся в конце строки. Другая характерная особенность состоит в том, что при сравнениях и сортировке регистр символов в колонках `CHAR` и `VARCHAR` не учитывается, если только не задать атрибут `BINARY`, например:

```
CREATE TABLE foo (A VARCHAR(10) BINARY)
```

В версиях MySQL после 3.23 можно осуществлять сравнение строк без учета регистра с помощью модификатора `BINARY`:

```
SELECT * FROM table WHERE BINARY column = "A"
```

Национальные символы обрабатываются в сравнениях соответственно системе кодировки, указанной во время компиляции, по умолчанию ISO-8859-1. Системы кодировки, не соответствующие ISO, в частности UTF-16, не поддерживаются.

Конкатенация строк производится с помощью функции SQL `CONCAT(s1, s2, ...)`.

Ядро mSQL (а следовательно, и драйвер DBD::mSQL) из строковых типов поддерживает только CHAR(M), соответствующий VARCHAR(M) в MySQL, и тип TEXT(M), являющийся гибридом CHAR и BLOB. Для всех строковых типов mSQL удаляет хвостовые пробелы. Кроме того, в mSQL нет возможности конкатенации строк.

Обработка данных типа «дата»

MySQL, согласно *mysql.info*, поддерживает следующие типы даты и времени:

DATE

Дата в диапазоне с 0000-01-01 по 9999-12-31. MySQL выводит значения DATE в формате YYYY-MM-DD, но позволяет присваивать значения колонкам DATE в таких форматах:

```
YYMMDD
YYYYMMDD
YY.MM.DD
YYYY.MM.DD
```

Здесь «.» может быть любым нецифровым разделителем, а при задании года двумя цифрами предполагается год 20YY, если YY меньше 70.

DATETIME

Комбинация даты и времени. Поддерживаемый диапазон от 0000-01-01 00:00:00 до 9999-12-31 23:59:59. MySQL выводит значения DATETIME в формате YYYY-MM-DD HH:MM:SS, но допускает присвоение значений колонкам DATETIME с использованием показанных выше форматов для DATE с добавлением HH:MM:SS.

TIMESTAMP(M)

Временная метка. Диапазон от 1970-01-01 00:00:00 и до некоторого момента в 2032 или 2106 году, в зависимости от специфического для операционной системы типа time_t. MySQL выводит значения TIMESTAMP в форматах YYYYMMDDHHMMSS, YYMMDDHHMMSS, YYYYMMDD или YYMMDD, в соответствии со значением M, равным 14 (или же отсутствием подобного значения), 12, 8 или 6, но присваивать значения колонкам TIMESTAMP позволяет с помощью строк или чисел. Этот формат вывода не согласуется с руководством, поэтому его нужно проверять в каждой версии, т. к. он может измениться.

Колонки типа TIMESTAMP удобно использовать для регистрации времени операций INSERT или UPDATE, поскольку если для них не указывается значение, им присваивается время последней операции. Текущее время устанавливается и при попытке присвоить значение NULL.

TIME

Время в диапазоне от -838:59:59 до 838:59:59. MySQL выводит значения времени в формате HH:MM:SS. Присваивать значения колонкам TIME можно с помощью формата [[[DAYS] [H]H:]MM:]SS[.fraction] или [[[[[H]H]H]H]MM]SS [.fraction].

YEAR

Год. Допустимыми значениями являются 1901–2155 и 0000 в формате четырех знаков и 1970–2069 при использовании двух цифр (70–69). При вводе двузначные годы в диапазоне 00–69 воспринимаются как 2000–2069. (YEAR является новым типом данных в MySQL 3.22.)

При использовании двух цифр года в формате YY-MM-DD (дата) или YY (год) они преобразуются в 2000–2069 или 1970–1999, соответственно. Таким образом, в MySQL нет проблемы 2000 года, но есть проблема 2070 года!

В MySQL 3.23 такой режим будет изменен на 2000–2068 и 1969–1999 согласно стандарту X/ Open Unix.[1]

Функция NOW() и ее псевдоним SYSDATE позволяют указывать текущие дату и время в командах SQL.

Для форматирования значений даты и времени можно использовать функцию DATE_FORMAT(date, format) с применением строк формата в стиле printf.

В MySQL много функций для работы с датой и временем, в том числе DAYOFWEEK(date) (1 = воскресенье, ..., 7 = суббота), WEEKDAY(date) (0 = понедельник, ..., 6 = воскресенье), DAYOFMONTH(date), DAYOFYEAR(date), MONTH(date), DAYNAME (date), MONTHNAME(date), WEEK(date), YEAR(date), HOUR(time), MINUTE (time), SECOND(time), DATE_ADD(date, interval) (interval задается в духе «2 HOURS») и DATE_SUB(date, interval).

Следующее выражение SQL можно использовать для преобразования целого числа «секунд после 1 января 1970 GMT» в соответствующее значение дата/время в базе данных:

```
FROM_UNIXTIME(seconds_since_epoch)
```

и обратно:

```
UNIX_TIMESTAMP(timestamp)
```

MySQL не производит автоматических настроек в соответствии с часовым поясом.

База данных mSQL поддерживает следующие типы даты/времени:

[1] См. *http://www.unix-systems.org/version2/whatsnew/year2000.html*.

```
DATE  - соответствует типу DATE в MySQL
TIME  - соответствует типу TIME в MySQL
```

В mSQL поддерживается единственный формат даты `DD-MMM-YYYY`, где `MMM` является трехсимвольным английским сокращением названия месяца. Единственный поддерживаемый в mSQL формат времени – `HH:MM:SS`.

Обработка данных типа LONG/BLOB

В MySQL поддерживаются следующие типы BLOB согласно *mysql.info*:

```
TINYBLOB   / TINYTEXT    максимальный размер 255 (2**8 - 1)
BLOB       / TEXT        максимальный размер 65535 (2**16 - 1)
MEDIUMBLOB / MEDIUMTEXT  максимальный размер 16777215 (2**24 - 1)
LONGBLOB   / LONGTEXT    максимальный размер 4294967295 (2**32 - 1)
```

Во всех типах BLOB допускаются двоичные символы. Атрибуты `LongReadLen` и `LongTruncOk` не поддерживаются.

Максимальный размер значений параметров метода `bind_param()` ограничен только максимальным размером команды SQL. По умолчанию это 1 Мбайт, но можно его увеличить почти до 24 Мбайт, изменив в *mysqld* переменную `max_allowed_packet`.

Передавать `TYPE` или другие атрибуты методу `bind_param()` при связывании этих типов не нужно.

В mSQL поддерживается один тип `BLOB`, а именно – `TEXT`. Это гибрид обычного `VARCHAR` и `BLOB`. Задается *средняя* длина, и данные, превышающие эту дину, автоматически помещаются в особую область таблицы.

Другие вопросы обработки данных

Версии драйверов 1.21_*xx* и выше поддерживают метод `type_info()`.

MySQL поддерживает автоматическое преобразование типов данных там, где оно уместно. mSQL, напротив, его не поддерживает.

Транзакции, изоляция и блокировка

Как mSQL, так и MySQL *не* поддерживают транзакции.

Поскольку текущие версии как mSQL, так и MySQL выполняют команды нескольких клиентов поочередно (атомарно) и не поддерживают транзакции, в режиме блокировки по умолчанию нет необходимости в защите изоляции транзакций.

В MySQL можно явным образом устанавливать блокировку таблиц, например:

```
LOCK TABLES table1 READ, table2 WRITE
```

Снятие блокировки происходит при выполнении новой команды LOCK TABLES, при отсоединении или явно командой:

```
UNLOCK TABLES
```

Существуют также определяемые пользователем блокировки, управлять которыми можно с помощью функций SQL GET_LOCK() и RELEASE_LOCK(). Нельзя автоматически блокировать строки или таблицы во время выполнения команд SELECT, это нужно делать явным образом.

Как можно догадаться, mSQL в данное время не поддерживает блокировки.

Диалект SQL

Чувствительность к регистру оператора LIKE

В MySQL для чувствительности к регистру *всех* операторов, сравнивающих символы, в том числе LIKE, необходимо присутствие атрибута BINARY хотя бы в одном месте – либо в типе поля команды CREATE TABLE, либо рядом с именем поля в операторе сравнения. Однако чувствительность к регистру всегда можно подавить, используя функцию TOLOWER.

В mSQL три оператора LIKE: LIKE чувствителен к регистру, CLIKE не чувствителен и RLIKE использует регулярные выражения в стиле Unix.

Синтаксис соединения таблиц

Соединения поддерживаются с использованием обычного синтаксиса:

```
SELECT * FROM a,b WHERE a.field = b.field
```

или иначе:

```
SELECT * FROM a JOIN b USING field
```

Внешние соединения поддерживаются в MySQL, но не mSQL:

```
SELECT * FROM a LEFT OUTER JOIN b ON condition
```

Внешние соединения в MySQL всегда являются левыми внешними соединениями.

Имена таблиц и колонок

В MySQL имена таблиц и колонок должны быть не длиннее 64 символов. В mSQL имена таблиц и колонок ограничены 35 символами.

В MySQL можно заключать имена таблиц и колонок в одиночные кавычки (использование стандартных двойных кавычек допустимо, если база данных запущена с ключом --ansi-mode). Это необходимо де-

лать, если имя содержит специальные символы или зарезервированное слово. В mSQL заключение имен в кавычки не поддерживается.

Ограничения на имена таблиц вызваны тем, что таблицы хранятся в файлах и имена таблиц являются, в действительности, именами файлов. В частности, чувствительность имен таблиц к регистру определяется файловой системой, также недопустимы некоторые символы, например «.» и «/».

В именах колонок регистр не различается в MySQL и различается в mSQL, но в обеих системах имена сохраняются без преобразования регистра.

В MySQL в именах могут использоваться символы национальных алфавитов (с установленным восьмым битом). В mSQL это недопустимо.

Row ID

В MySQL нет идентификаторов строк, в mSQL есть псевдоколонка `_rowid`.

Колонка mSQL `_rowid` является числовой, и, поскольку mSQL не производит автоматического преобразования строк в числа, нужно следить за тем, чтобы значение не заключалось в кавычки при последующем использовании в командах `SELECT`.

Имейте в виду, что поскольку транзакции и блокировка не поддерживаются, возрастает опасность того, что строка, которую вы идентифицируете значением `_rowid`, окажется удаленной или замененной к тому времени, когда вы решите использовать идентификатор строки.

Автоматическая генерация ключей и последовательностей

Для всех целочисленных полей в MySQL можно установить атрибут `AUTO_INCREMENT`. Это значит, что если есть таблица:

```
CREATE TABLE a (
  id INTEGER AUTO_INCREMENT NOT NULL PRIMARY KEY,
  ...)
```

и команда:

```
INSERT INTO a (id, ...) VALUES (NULL, ...)
```

то автоматически генерируется уникальный ID (то же, если поле ID вообще не указывать в команде вставки). Сгенерированный ID позднее можно извлечь:

```
$sth->{mysql_insertid}                (1.21_xx)
$sth->{insertid}                      (1.20_xx)
```

Либо, если вы используете `$dbh->do`, а не `prepare/execute`, выполните команды:

```
$dbh->{mysql_insertid}                (1.21_xx)
$dbh->do("SELECT LAST_INSERT_ID()");  (1.20_xx)
```

MySQL не поддерживает непосредственно генераторы последовательностей, но они легко могут эмулироваться (детали можно найти в руководстве по MySQL). Например:

```
UPDATE seq SET id=last_insert_id(id+1)
```

База данных mSQL поддерживает последовательности, по одной на таблицу. Выполнив:

```
CREATE SEQUENCE on A
```

впоследствии можно извлечь значение с помощью команды:

```
SELECT _seq FROM A
```

Нельзя непосредственно ссылаться на последовательность в команде вставки, нужно сделать выборку ее значения, а затем выполнить вставку с полученной величиной.

Автоматическая нумерация строк и предельное число строк

Ни то ни другое ядро не поддерживает автоматической нумерации строк в результирующем наборе команды `SELECT`.

mSQL и MySQL поддерживают ограничение числа возвращаемых строк, например:

```
SELECT * FROM A LIMIT 10
```

возвращает только первые 10 строк, но только MySQL поддерживает синтаксис:

```
SELECT * FROM A LIMIT 20, 10
```

для того, чтобы вернуть строки 20–29, начав нумерацию с 0.

Обновление и удаление по месту

Обновления и удаления по месту в MySQL и mSQL не поддерживаются.

Связывание параметров

Ни то ни другое ядро не поддерживает заполнители, но драйверы `DBD::mysql` и `DBD::mSQL` предоставляют полную их эмуляцию. В качестве заполнителей используются вопросительные знаки:

```
$dbh->do("INSERT INTO table VALUES (?, ?)", undef, $id, $name);
```

Заполнители вида «`:1`» не поддерживаются.

В приведенном примере драйвер пытается угадать тип данных вводимых значений, отыскивая в собственных внутренних строках Perl указания на числовой тип данных. В MySQL это удается, поскольку в нем можно использовать такие выражения:

```
INSERT INTO table (id_number) VALUES ('2')
```

где id_number является числовой колонкой. mSQL рассматривает такую команду как ошибочную, поэтому иногда приходится задавать тип данных, используя команду типа:

```
$dbh->do("INSERT INTO table VALUES (?, ?)", undef, int($id), "$name");
```

или используя атрибут TYPE метода bind_param():

```
use DBI qw(:sql_types);
$sth = $dbh->prepare("INSERT INTO table VALUES (?, ?)");
$sth->bind_param(1, $id,   SQL_INTEGER);
$sth->bind_param(2, $name, SQL_VARCHAR);
$sth->execute();
```

В текущих версиях драйверов при задании неподдерживаемых значений атрибута TYPE предупреждение не генерируется.

Хранимые процедуры

В mSQL и MySQL отсутствует понятие хранимых процедур, хотя планируется добавить некоторые возможности хранимых процедур в MySQL.

Метаданные таблиц

Поддержка метода table_info() была включена в драйверы, начиная с версии 1.21_*xx*.

Для получения сведений о таблице можно выполнить запрос:

```
LISTFIELDS $table
```

При этом возвращается дескриптор команды без результирующих строк. Таблица описывается атрибутами TYPE, NAME и т. д.

В MySQL можно использовать команду:

```
SHOW INDEX FROM $table
```

для извлечения сведений об индексах таблицы, в частности первичном ключе. Данные возвращаются в виде строк. Драйвер DBD::mSQL поддерживает аналогичную функцию в виде:

```
LISTINDEX $table $index
```

где $index указывает имя требуемого индекса.

Специфические для драйверов атрибуты и методы

Поддерживаются следующие специфические для драйвера атрибуты дескриптора базы данных:

mysql_info

mysql_thread_id

mysql_insertid

Эти атрибуты относятся к C-вызовам mysql_info(), mysql_thread_id() и mysql_ insertid(), соответственно.

Поддерживаются следующие специфические для драйвера атрибуты дескриптора команды:

mysql_use_result

mysql_store_result

В DBD::mysql получение драйвером результатов с сервера может производиться двумя способами. Если включен mysql_store_result, производится выборка всех строк, в памяти создается таблица с результатами, которая и возвращается (100 % кэш строк).

Если установлен mysql_use_result, строки возвращаются приложению по мере выборки. При этом более экономно расходуется память клиента, но использовать такой режим, когда к базе обращается несколько пользователей, нельзя, поскольку могут оказаться заблокированными другие приложения (не путать с блокировкой данных!).

mysql_insertid

Ранее сгенерированное значение колонки auto_increment.

mysql_is_blob

mysql_is_key

msql_is_num

mysql_is_num

msql_is_pri_key

mysql_is_pri_key

Эти атрибуты возвращают ссылки на массивы значений флагов для колонок результирующего набора. Их можно использовать в запросе LISTFIELDS для получения сведений о колонках таблицы.

mysql_max_length

В отличие от атрибута PRECISION, этот атрибут возвращает фактический максимальный размер конкретных данных в текущем результирующем наборе. Это может быть использовано, например, при выводе таблиц ASCII.

Этот атрибут не действует при установленном mysql_use_result, поскольку ему требуется просмотреть все данные.

msql_table

mysql_table

Аналогичны NAME, но возвращаются имена таблиц, а не колонок.

`msql_type`

`mysql_type`

Аналогичны `TYPE`, но возвращаются собственные типы соответствующего ядра.

`msql_type_name`

`mysql_type_name`

Аналогичны `msql_type` и `mysql_type`, но возвращаются имена колонок, которые можно использовать в команде `CREATE TABLE`.

Поддерживается один частный метод с именем `admin()`. Он предоставляет ряд административных функций:

```
$rc = $drh->func('createdb', $db, $host, $user, $password, 'admin');
$rc = $drh->func('dropdb',   $db, $host, $user, $password, 'admin');
$rc = $drh->func('shutdown',      $host, $user, $password, 'admin');
$rc = $drh->func('reload',        $host, $user, $password, 'admin');

$rc = $dbh->func('createdb', $database, 'admin');
$rc = $dbh->func('dropdb',   $database, 'admin');
$rc = $dbh->func('shutdown',            'admin');
$rc = $dbh->func('reload',              'admin');
```

Они соответствуют командам `mysqladmin` и `msqladmin`.

DBD::ODBC

Общие сведения

Версия драйвера

`DBD::ODBC` версия 0.20.

Краткая характеристика

Поскольку `DBD::ODBC` действует как интерфейс к другим базам данных, его работа во многом определяется этими драйверами.

Транзакции	В зависимости от источника данных
Блокировка	В зависимости от источника данных
Соединения таблиц	В зависимости от источника данных
LONG/LOB-типы данных	В зависимости от источника данных
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да
Хранимые процедуры	Да
Связывание выходных значений	Нет

Регистр символов в именах таблиц	В зависимости от источника данных
Регистр символов в именах полей	В зависимости от источника данных
Преобразование недопустимых имен	В зависимости от источника данных
Нечувствительный к регистру оператор "LIKE"	В зависимости от источника данных
Псевдоколонка ROW ID в серверных таблицах	В зависимости от источника данных
Обновление/удаление по месту	Да
Одновременное использование нескольких дескрипторов	В зависимости от источника данных

Автор и как с ним связаться

Авторы драйвера – Джефф Урлвин (Jeff Urlwin) и Тим Банс (Tim Bunce). Первоначально работа основывалась на ранней версии `DBD::Solid`, который создал Томас Венрих (Thomas Wenrich). Связаться с авторами можно через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль `DBD::ODBC` поддерживает ODBC версий 2.*x* и 3.*x* под Unix и Win32. Для всех платформ, *кроме* модуля `DBD::ODBC`, требуются администратор драйверов ODBC *и* драйверы ODBC.

Для Win32 администратор драйверов включен в операционную систему. В Unix исходный код администратора драйверов iODBC находится в каталоге `iodbcsrc`. Пока iODBC действует в качестве *администратора* драйверов, вам необходимо найти фактический драйвер для вашей платформы и базы данных.

В число поставщиков драйверов входят:

Intersolv: *http://www.intersolv.com*

OpenLink: *http://www.openlinksw.com*

Это не полный список, есть и другие поставщики. Вот еще ссылки, относящиеся к ODBC:

http://www.genix.net/unixODBC

http://www.openlinksw.com/iodbc

http://www.microsoft.com/data/odbc

Чтобы подписаться на список рассылки для разработчиков *freeodbc*, пошлите письмо на *freeodbc-request@as220.org* с единственным словом `subscribe` в теле сообщения.

Отличия от спецификации DBI

DBD::ODBC в данное время не поддерживает связывание «выходных» параметров. Эта функция будет реализована в будущих выпусках.

Синтаксис соединения

Формат DBI->connect() «Имя источника данных» (DSN) может быть следующим:

```
dbi:ODBC:odbc_dsn
dbi:ODBC:driver=Microsoft Access Driver (*.mdb);dbq=\\server\share\access.mdb
```

В первом случае odbc_dsn – это имя источника данных ODBC (DSN). DSN является просто именем, используемым для ссылки на набор специфических для драйвера параметров соединения, определенных где-то в другом месте. В параметры соединения обычно входят имя используемого драйвера ODBC, имя базы данных и все элементы, необходимые для соединения.

В Win32 создавать ODBC DSN лучше всего с помощью приложения ODBC32 в панели управления Windows. В Unix обычно требуется отредактировать текстовый файл *.odbc.ini* в исходном каталоге. Более подробная информация должна содержаться в документации к администратору драйверов.

Во втором примере соединения используется специфическая для драйвера строка соединения. Задав все необходимые сведения, можно обойтись без определенного ранее DSN. В нашем примере используется драйвер «Microsoft Access Driver (*.mdb)» для доступа к базе данных Access, находящейся в каталоге *server**share**access.mdb*.

В настоящее время нет специфических для драйвера атрибутов метода DBI->connect().

Большинство драйверов ODBC и баз данных позволяют осуществлять несколько одновременных соединений с одной и той же базой данных, но есть и такие, в которых не предусмотрена эта возможность.

Некоторые драйверы ODBC и базы данных, особенно Sybase и SQL Server, не позволяют подготовить и выполнить новый дескриптор команды, пока производится выборка данных через другой дескриптор команды, связанный с тем же дескриптором базы данных.

Типы данных

Обработка числовых данных

Обработка числовых данных в ODBC зависит от множества факторов. Одним из этих решающих факторов является конечная база данных. Например, Oracle поддерживает иные числовые типы, нежели Sybase, которая, в свою очередь, поддерживает иные типы данных, чем файл

CSV. Дополнительные сведения вы можете почерпнуть в документации к своей базе данных.

К несчастью, вторым важным набором факторов являются производитель драйвера ODBC и версия драйвера. Например, я видел множество различий в обработке числовых данных в разных версиях драйверов ODBC для Oracle. То, что работает в одной версии, может, к сожалению, не работать даже в более поздних версиях драйверов для Oracle. Дополнительные сведения ищите в документации к своему драйверу ODBC.

Методы DBI `type_info()` и `type_info_all()` предоставляют сведения о типах данных, поддерживаемых базой данных, и используемом драйвере.

Обработка строковых данных

Как и в случае обработки чисел, обработка строк зависит от базы данных и драйвера. Дополнительные сведения приведены выше в разделе «Обработка числовых данных».

Конкатенация строк осуществляется с помощью функции SQL `CONCAT(s1,s2)`.

Обработка данных типа «дата»

Как и в случае обработки чисел, обработка дат зависит от базы данных и драйвера. Дополнительные сведения приведены выше в разделе «Обработка числовых данных».

Можно использовать управляющие последовательности ODBC для установки даты независимым от базы данных способом. Например, чтобы ввести в таблицу дату 21 января 1998 года, можно использовать команду:

```
INSERT INTO table_name (date_field) VALUES ({d '1998-01-21'});
```

Вместо литералов можно использовать заполнители в управляющих последовательностях, например:

```
INSERT INTO table_name (date_field) VALUES ({d ?});
```

Аналогичные управляющие последовательности определены для других типов дата/время. Вот их полный набор:

```
{d 'YYYY-MM-DD'}                    - дата
{t 'HH:MM:SS'}                      - время
{ts 'YYYY-MM-DD HH:MM:SS'}          - временная метка
{ts 'YYYY-MM-DD HH:MM:SS.FFFFFFF'}  - временная метка
```

Если задать значение DATE без составляющей времени, то устанавливается время по умолчанию 00:00:00 (полночь). Есть также управляющее предложение для интервалов, которое используется так:

```
{interval [+|-] 'value' [interval_qualifier]}
```

Например:

```
{interval '200-11' YEAR(3) TO MONTH}
```

Более подробные сведения можно найти в справочнике по ODBC.

Текущие дату и время на сервере можно получить с помощью управляющей последовательности для вызова соответствующей скалярной функции ODBC, например:

```
INSERT INTO table_name (date_field) VALUES ({fn CURDATE});
```

Управляющая последовательность `{fn ...}` не требуется, если команда SQL полностью удовлетворяет уровню грамматики SQL-92, который поддерживается вашим драйвером ODBC.

В число других родственных функций входят `CURTIME()`, `NOW()`, `CURRENT_DATE()`, `CURRENT_TIME()` и `CURRENT_TIMESTAMP()`. Для трех последних требуется драйвер ODBC версии 3.

В число других функций, относящихся к дате/времени, входят: `DAYNAME()`, `DAYOFMONTH()`, `DAY_OFWEEK()`, `DAYOFYEAR()`, `EXTRACT()`, `HOUR()`, `MINUTE()`, `MONTH()`, `MONTH_NAME()`, `SECOND()`, `WEEK()` и `YEAR()`.

Базовые арифметические действия с датой/временем можно выполнять с помощью функций `TIMESTAMPADD()` и `TIMESTAMPDIFF()`.

Следующее выражение SQL можно использовать для преобразования целого числа секунд, прошедших с 1 января 1970 года, в соответствующее значение дата/время в базе данных:

```
TIMESTAMPADD(SQL_TSI_SECOND, seconds_since_epoch, {d '1970-01-01'})
```

Для обратного преобразования можно использовать выражение:

```
TIMESTAMPDIFF(SQL_TSI_SECOND, {d '1970-01-01'}, date_field)
```

В самом ODBC нет поддержки временных поясов, но в базах данных, с которыми производится соединение, она может присутствовать.

Обработка данных типа LONG/BLOB

Поддержка типов данных `LONG/BLOB` и их максимальные размеры сильно зависят от базы данных, с которой осуществляется соединение.

Атрибуты `LongReadLen` и `LongTruncOk` действуют согласно определению. Однако реализация драйвера оказывает на них влияние. Некоторые данные неверно сообщают о произведенном ими усечении данных или предоставляют больше данных, чем было фактически извлечено.

Для типов данных `LONG/BLOB` не требуется специальной обработки. С ними можно работать как с любыми другими полями при выборке, вставке и т. п.

Другие вопросы обработки данных

Драйвер DBD::ODBC поддерживает метод type_info().

Транзакции, изоляция и блокировка

DBD::ODBC поддерживает транзакции, если их поддерживает база данных, с которой осуществляется соединение.

Поддерживаемые уровни изоляции транзакций, уровень, устанавливаемый по умолчанию, и режим блокировки зависят от базы данных, с которой осуществляется соединение.

Диалект SQL

Поскольку DBD::ODBC действует как интерфейс к другим драйверам баз данных, следующие вопросы определяются этими драйверами и базами данных, соединение с которыми они осуществляют:

- чувствительность к регистру оператора LIKE;
- имена таблиц и колонок;
- Row ID;
- автоматическая генерация ключей и последовательностей;
- автоматическая нумерация строк и предельное число строк.

За дополнительными сведениями обратитесь к документации по используемым драйверам и базам данных.

Синтаксис соединения таблиц

Синтаксис соединения таблиц частично зависит от базы данных, с которой осуществлено соединение, и используемого драйвера ODBC. Стандартный SQL ODBC определяет стандартный синтаксис внутренних соединений и управляющих последовательностей, используемых для внешних соединений:

```
{oj outer_join}
```

где outer_join определяется как

```
table_name [LEFT | RIGHT | FULL]
  OUTER JOIN [ table_name | outer_join] ON condition
```

Запрос внешнего соединения должен стоять после предложения FROM команды SELECT, но перед предложением WHERE, если оно есть.

Обновление и удаление по месту

Зависит от базы данных, с которой производится соединение. Обновление и удаление по месту поддерживаются в ODBC SQL с использованием синтаксиса WHERE CURRENT OF.

Вот пример:

```
$dbh->do("UPDATE ... WHERE CURRENT OF $sth->{CursorName}");
```

Связывание параметров

Связывание параметров в DBD::ODBC поддерживается, если его обеспечивает соответствующий драйвер ODBC. Поддерживается лишь стандартный заполнитель ?.

Атрибут TYPE метода bind_param() поддерживается.

Хранимые процедуры

Хранимые процедуры можно вызывать, используя следующие управляющие последовательности ODBC:

```
{call procedure1_name}
{call procedure2_name(?, ?)}
{?= call procedure3_name(?, ?)}
```

Последняя форма должна использоваться для получения значений, возвращаемых процедурой, но DBD::ODBC в настоящей версии не поддерживает выходные параметры.

Метаданные таблиц

DBD::ODBC поддерживает метод table_info().

DBD::ODBC поддерживает также многие функции *метаданных* ODBC, которые можно использовать для получения сведений о таблицах, имеющихся в базе данных. Доступ к ним осуществляется как к специфическим для драйвера частным методам:

```
SQLGetTypeInfo   --  $dbh->func(xxx,         'GetTypeInfo')
SQLDescribeCol   --  $sth->func(colno,       'DescribeCol')
SQLColAttributes --  $sth->func(xxx, colno, 'ColAttributes')
SQLGetFunctions  --  $dbh->func(xxx,         'GetFunctions')
SQLColumns       --  $dbh->func(catalog, schema, table, column, 'columns')
SQLStatistics    --  $dbh->func(catalog, schema, table, unique, 'Statistics')
SQLPrimaryKeys   --  $dbh->func(catalog, schema, table, 'PrimaryKeys')
SQLForeignKeys   --  $dbh->func(pkc, pks, pkt, fkc, fks, fkt, 'ForeignKeys')
```

В ближайшее время в DBI должны быть включены стандартные методы для этих функций; возможно, в тот момент, когда вы читаете эту книгу, они уже имеются в нем.

Специфические для драйверов атрибуты и методы

В DBD::ODBC нет специфических для драйвера атрибутов дескрипторов.

Кроме частных методов, описанных выше в разделе «Метаданные таблиц», можно использовать частный метод `GetInfo()` для получения подробных сведений об используемых драйвере и базе данных.

DBD::Oracle

Общие сведения

Версия драйвера

`DBD::Oracle` версия 1.03.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 4 Гбайт
Доступность атрибутов дескриптора команды	После prepare()
Заполнители	Да, "?" и ":1" (собственные)
Хранимые процедуры	Да
Связывание выходных значений	Да
Регистр символов в именах таблиц	Не учитывается, хранятся в верхнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет
Псевдоколонка ROW ID в серверных таблицах	Да, "ROWID"
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Тим Банс (Tim Bunce). С ним можно связаться через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль `DBD::Oracle` поддерживает Oracle 7 и Oracle 8.

При компиляции модуля по умолчанию включается поддержка нового интерфейса Oracle 8 OCI, позволяющего использовать некоторые функции Oracle 8, в том числе LOB и «`INSERT ... RETURNING ...`».

С `DBD::Oracle` поставляется модуль эмуляции старого программного обеспечения Perl4 oraperl, что позволяет легко обновлять сценарии oraperl до Perl5.

Дополнительные сведения об Oracle можно получить на серверах:

http://www.oracle.com

http://technet.oracle.com

Отличия от спецификации DBI

Существенных отличий работы `DBD::Oracle` от текущей спецификации DBI не отмечено.

Синтаксис соединения

Синтаксис `DBI->connect()` «Имя источника данных» (DSN) может быть следующим:

```
dbi:Oracle:tnsname
dbi:Oracle:sidname
dbi:Oracle:host=hostname;sid=sid
```

Некоторые другие, более редкие форматы также можно использовать, если они поддерживаются в используемой версии клиента Oracle.

Существенных специфических для драйвера атрибутов метода `DBI->connect()` нет.

`DBD::Oracle` поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

В Oracle есть только один, но гибкий числовой тип NUMBER. Но Oracle поддерживает в качестве псевдонимов несколько типов данных стандарта ANSI и IBM, в том числе:

```
INTEGER      = NUMBER(38)
INT          = NUMBER(38)
SMALLINT     = NUMBER(38)
DECIMAL(p,s) = NUMBER(p,s)
NUMERIC(p,s) = NUMBER(p,s)
FLOAT        = NUMBER
FLOAT(b)     = NUMBER(p)   где b - двоичная точность, от 1 до 126
REAL         = NUMBER(18)
```

Тип данных NUMBER позволяет хранить положительные и отрицательные числа с фиксированной и плавающей запятой в диапазоне от $1{,}0 \times 10^{-130}$ до $9{,}9\ldots9 \times 10^{125}$ (38 девяток и 88 нулей) с 38-значной точностью.

Можно задать число с фикисированной точкой в виде: `NUMBER(p,s)`, где `s` является масштабом или числом цифр справа от десятичной точки. Масштаб может изменяться от −84 до 127.

Целое число можно задать в виде NUMBER(p). Это число с фиксированной точкой с точностью p и масштабом 0. Эквивалентная запись – NUMBER(p,0).

Число с плавающей точкой можно задать как NUMBER. Это число с плавающей точкой и десятичной точностью 38. Величина масштаба неприменима к числам с плавающей точкой.

DBD::Oracle возвращает все числа в виде строк. Поэтому драйвер не накладывает ограничений на величину PRECISION или SCALE.

Обработка строковых данных

Oracle поддерживает следующие строковые типы данных:

```
VARCHAR2(size)
NVARCHAR2(size)
CHAR
CHAR(size)
NCHAR
NCHAR(size)
RAW(size)
```

Тип RAW представляется в виде шестнадцатеричных цифр. Содержимое воспринимается как несимвольные двоичные данные и потому не подвергается перекодировке интерфейсами шлюзов или при преобразовании наборов символов.

Типы CHAR и RAW имеют максимальный размер 2000 байт. Для VARCHAR предельный размер равен 2000 байт в Oracle 7 и 4000 в Oracle 8.

Типы NVARCHAR2 и NCHAR предназначены для хранения строк символов национального алфавита (только в Oracle 8). Для этих типов максимальное число хранимых символов может быть меньше, если используются наборы многобайтовых символов.

Типы CHAR и NCHAR имеют фиксированную длину и дополняются пробелами.

Oracle автоматически преобразует символьные данные из набора символов базы данных, установленного при создании базы данных, в набор символов клиента, определяемый параметром NLS_LANG для типов данных CHAR и VARCHAR2 или параметром NLS_NCHAR для типов NCHAR и NVARCHAR2.

Чтобы преобразовать строки из одного набора символов в другой, можно использовать функцию CONVERT(string, dest_char_set, source_char_set). Oracle 8 поддерживает 180 наборов символов, в которых может осуществляться хранение. Поддерживается UTF-8. См. в разделе «Поддержка национальных языков» справочного руководства Oracle подробности использования наборов символов.

Конкатенация строки осуществляется с помощью функции SQL CONCAT(s1,s2,...) или оператора ||.

Обработка данных типа «дата»

Oracle поддерживает один гибкий тип данных дата/время: DATE. Даты могут иметь значение в диапазоне от 1 января 4712 до нашей эры до 31 декабря 4712 года нашей эры с разрешением в одну секунду.

Oracle поддерживает очень широкий спектр форматов даты и может использовать один из нескольких календарей (арабская хиджра, английская хиджра, грегорианский, японский имперский, персидский, ROC Official Республики Китай и таиландский буддийский. Мы будем рассматривать здесь только грегорианский календарь.

Формат по умолчанию для вывода типа DATE определяется параметром конфигурации NLS_DATE_FORMAT, но обычно в большинстве инсталляций на западе это DD-MON-YY, например 20-FEB-99. Формат по умолчанию для ввода данных типа DATE тот же, что и для вывода. Распознается только один этот формат.

Если задать значение DATE без указания времени, то устанавливается время по умолчанию 00:00:00 (полночь). Если задать значение DATE без даты, по умолчанию дата устанавливается как первое число текущего месяца. Если в формате даты присутствуют только две цифры года, например YY в DD-MON-YY (часто используется по умолчанию), то возвращаемая дата принадлежит текущему столетию. Можно использовать формат RR для смещения даты на пятьдесят лет.

Формат даты по умолчанию задается явно через параметр инициализации NLS_ DATE_FORMAT или неявно с помощью параметра инициализации NLS_TERRITORY. Описание этих параметров можно найти в руководстве по Oracle 8.

Для данного сеанса можно изменить формат даты, установленный по умолчанию, с помощью команды ALTER SESSION. Например:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'MM/DD/YYYY'
```

Функцию TO_DATE() можно использовать для разбора строки символов, содержащей дату в известном формате, например:

```
UPDATE table SET date_field = TO_DATE('1999-02-21', 'YYYY-MM-DD')
```

Для форматирования даты можно использовать функцию TO_CHAR(), например:

```
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD') FROM DUAL
```

Текущие дату и время возвращает функция SYSDATE().

К значениям типа DATE можно прибавлять числа. Число интерпретируется как количество дней. Например, SYSDATE + 1 означает то же самое время завтрашнего дня, а SYSDATE - (3/1440) означает время три минуты назад. Можно вычитать даты одну из другой, чтобы найти разницу между ними в днях.

Oracle предоставляет большой набор функций для работы с датами, в том числе ROUND(), TRUNC(), NEXT_ DAY(), ADD_MONTHS(), LAST_DAY() (последний день месяца) и MONTHS_BETWEEN().

Следующее выражение SQL можно использовать для преобразования целого числа секунд, прошедших с 1 января 1970 года, в соответствующее значение дата/время в базе данных:

```
to_date(trunc(:unixtime/86400, 0) + 2440588, 'J') -- часть, относящаяся к дате
+(mod(:unixtime,86400)/86400)                     -- часть, относящаяся к времени
```

Для обратного преобразования можно использовать выражение:

```
(date_time_field - TO_DATE('01-01-1970','DD-MM-YYYY')) * 86400
```

Oracle не производит автоматических настроек в соответствии с часовым поясом, однако он предоставляет функцию NEW_TIME(), которая рассчитывает настройки времени для ряда часовых поясов. NEW_TIME(d, z1, z2) возвращает дату и время в часовом поясе z2, если дата и время в часовом поясе z1 представлены значением d.

Обработка данных типа LONG/BLOB

Oracle поддерживает следующие типы данных LONG/BLOB:

```
LONG      - символьные данные переменной длины
LONG RAW  - сырые двоичные данные переменной длины
CLOB      - большой объект, содержащий однобайтовые символы
NCLOB     - большой объект, содержащий символы из национального набора
BLOB      - большой двоичный объект
BFILE     - локатор для внешнего файла больших двоичных объектов
```

Типы LONG могут хранить до 2 Гбайт данных. Другие типы (LOB и FILE) могут хранить до 4 Гбайт. Типами LOB и FILE можно пользоваться только при работе с Oracle 8 OCI.

Типы LONG RAW и RAW передаются в/из базы данных как строки пар шестнадцатеричных цифр.

Атрибуты LongReadLen и LongTruncOk работают согласно определению. Однако атрибут LongReadLen оказывается ограниченным 65 535 байтами на большинстве платформ при работе с Oracle 7. Компиляция DBD::Oracle вместе с Oracle 8 OCI увеличивает этот предел до 4 Гбайт.

Максимальный размер значения параметра метода bind_param(), которое можно использовать при вставке данных типа LONG, оказывается ограниченным 65 535 байтами на большинстве платформ при работе с Oracle 7. Компиляция DBD::Oracle вместе с Oracle 8 OCI увеличивает этот предел до 4 Гбайт.

Значение атрибута TYPE, равное SQL_LONGVARCHAR, указывает на тип Oracle LONG. Значение SQL_LONGVARBINARY указывает на тип Oracle LONG RAW. Эти значения требуются не всегда, но их использование весьма рекомендовано.

Другой специальной обработки для данных типа LONG/BLOB не требуется. С ними можно обращаться, как с любыми другими полями при выборке или вставке и т. д.

Другие вопросы обработки данных

Драйвер DBD::Oracle поддерживает метод type_info().

Oracle поддерживает автоматическое преобразование типов данных там, где оно уместно.

Транзакции, изоляция и блокировка

DBD::Oracle поддерживает транзакции. По умолчанию установлен уровень изоляции транзакций READ COMMITED.

Oracle поддерживает уровни изоляции READ COMMITED и SERIALIZABLE. Для каждой транзакции уровень можно один раз изменить, выполнив команду SET TRANSACTION ISOLATION LEVEL x (где x является названием требуемого уровня изоляции транзакций).

Oracle поддерживает также непротиворечивость чтения на уровне транзакции. Включение производится командой SET TRANSACTION с параметром READ ONLY.

В Oracle по умолчанию установлен режим, при котором блокировка не мешает другим пользователям осуществлять запросы к таблице. Запрос не блокирует таблицу. При чтении не блокируется запись, а при записи не блокируется чтение.

Строки, возвращаемые командой SELECT, можно заблокировать для изменения в других транзакциях путем приписывания предложения FOR UPDATE к команде SELECT. Дополнительно после предложения FOR UPDATE можно указать в скобках список блокируемых колонок.

Для явной установки блокировки на всю таблицу можно использовать команду LOCK TABLE table_name IN lock_mode. Поддерживается ряд блокировок строк и таблиц.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE чувствителен к регистру.

Синтаксис соединения таблиц

Oracle поддерживает внутренние соединения, используя обычный синтаксис:

```
SELECT * FROM a, b WHERE a.field = b.field
```

Чтобы написать запрос, который производит внешнее соединение таблиц A и B и возвращает все строки таблицы A, нужно применить оператор Oracle для внешнего соединения (+) ко всем колонкам B, фигурирующим в условии соединения. Например:

```
SELECT customer_name, order_date
FROM customers, orders
WHERE customers.cust_id = orders.cust_id (+);
```

Для всех строк в таблице `customers`, не имеющих соответствующих строк в таблице `orders`, Oracle возвращает `NULL` для всех выражений списка выборки, содержащих колонки таблицы `orders`.

Имена таблиц и колонок

Имена идентификаторов Oracle, в частности таблиц и колонок, не могут превосходить в длину 30 символов.

Первым символом должна быть буква, но остальные могут представлять любую комбинацию букв, цифр, знаков ($) и (#) и символов подчеркивания (_).

Однако если заключить идентификатор Oracle в двойные кавычки ("), то он может содержать любые допустимые символы, в том числе и пробелы, за исключением кавычек.

Oracle преобразует все идентификаторы к верхнему регистру, если они не заключены в двойные кавычки. При использовании кавычек можно также применять национальные символы.

Row ID

Псевдоколонка идентификатора строки в Oracle называется ROWID. Колонки ROWID – буквенно-цифровые и учитывают регистр. Можно обращаться с ними, как с обычными строками, и использовать для быстрой выборки строк.

Автоматическая генерация ключей и последовательностей

Oracle поддерживает «генераторы последовательностей». В базе данных можно создать любое число именованных генераторов последовательностей с помощью команды SQL `CREATE SEQUENCE seq_name`. В каждой из них есть псевдоколонки `NEXTVAL` и `CURRVAL`. Вот образец обычного использования:

```
INSERT INTO table (k, v) VALUES (seq_name.nextval, ?)
```

Чтобы получить только что использованное значение, можно выполнить команду:

```
SELECT seq_name.currval FROM DUAL
```

Oracle не поддерживает автоматическую генерацию ключей типа автоинкрементируемых или генерируемых системой. Однако их можно эмулировать с помощью триггеров и генераторов последовательностей. Например:

```
CREATE TRIGGER trigger_name
  BEFORE INSERT ON table_name FOR EACH ROW
  DECLARE newid integer;
BEGIN
  IF (:NEW.key_field_name IS NULL)
  THEN
    SELECT sequence_name.NextVal INTO newid FROM DUAL;
    :NEW.key_field_name := newid;
  END IF;
END;
```

Oracle8i (8.1.0 и выше) поддерживает генерацию глобальных уникальных идентификаторов согласно проекту IETF, используя новую функцию `SYS_GUID()`. Применение `GUID` в распределенных базах данных более полезно, чем генераторов последовательностей, поскольку никакие два узла не сгенерируют один и тот же `GUID`.

Автоматическая нумерация строк и предельное число строк

Псевдоколонку `ROWNUM` можно использовать для последовательной нумерации выбранных строк (начиная с 1). К сожалению, `ROWNUM` в Oracle обладает некоторыми обескураживающими ограничениями. Смотрите документацию по Oracle SQL.

Обновление и удаление по месту

Oracle не поддерживает обновление и удаление по месту.

Связывание параметров

Связывание параметров в Oracle поддерживается непосредственно. Поддерживаются заполнители двух видов – `?` и `:1`. Форма `:name` также поддерживается, но она не является переносимой.

Атрибут `TYPE` метода `bind_param()` можно использовать для указания типа, с которым должен связываться параметр. Эти типы SQL связываются как `VARCHAR2`: `SQL_NUMERIC`, `SQL_ DECIMAL`, `SQL_INTEGER`, `SQL_SMALLINT`, `SQL_FLOAT`, `SQL_REAL`, `SQL_DOUBLE` и `SQL_VARCHAR`. Oracle автоматически преобразует `VARCHAR2` в требуемый тип.[1]

[1] При работе со строками и предоставлении Oracle возможности производить преобразование вы получите многие преимущества. Формат упакованных десятичных чисел в Oracle является компактным, быстрым в работе и обладает значительно большими точностью и масштабом представления, чем собственные числовые величины Perl.

Тип SQL_CHAR связывается как CHAR, что позволяет использовать семантику сравнения с фиксированным размером и дополнением пробелами.

Типы SQL_BINARY и SQL_VARBINARY связываются как RAW. SQL_LONGVAR_BINARY связывается как LONG RAW и SQL_LONGVARCHAR – как LONG.

При задании неподдерживаемых значений атрибута TYPE выводятся предупреждения.

Обратитесь к документации по DBD::Oracle за подробностями относительно привязки LOB и CURSOR.

Хранимые процедуры

Хранимые процедуры Oracle реализуются на языке Oracle PL/SQL.[1]

Модуль DBD::Oracle можно использовать для выполнения блоков программного кода PL/SQL, которые начинаются словом BEGIN и заканчиваются словом END;. Блоки PL/SQL используются для вызова хранимых процедур. Вот простой пример вызова хранимой процедуры «foo», которой передаются два параметра:

```
$sth = $dbh->prepare("BEGIN foo(:1, :2) END;");
$sth->execute("Baz", 24);
```

Ниже приведен более сложный пример, в котором хранимая процедура вызывается с двумя параметрами и возвращает значения. Второй параметр процедуры определен как IN OUT, поэтому мы связываем его с помощью метода bind_param_inout(), чтобы позволить обновление переменной Perl:

```
$sth = $dbh->prepare("BEGIN :result = func_name(:id, :changeme) END;");
$sth->bind_param(":id", "FooBar");
my ($result, $changeme) = (41, 42);
$sth->bind_param_inout(":result",  \$result,  100);
$sth->bind_param_inout(":changeme", \$changeme, 100);
$sth->execute();
print "func_name возвратила '$result' и изменила значение changeme на '$changeme'\n";
```

Метаданные таблиц

DBD::Oracle поддерживает метод table_info().

Представление ALL_TABLES содержит подробные сведения обо всех таблицах в базе данных, по одной строке на каждую таблицу.

[1] Процедурное расширение SQL с поддержкой переменных, ветвления выполнения, пакетов, исключительных ситуаций и др. В Oracle8i хранимые процедуры можно реализовывать на Java.

Представление ALL_TAB_COLUMNS содержит подробные сведения обо всех колонках всех таблиц в базе данных, по одной строке на каждую таблицу.

Представление ALL_INDEXES содержит подробные сведения обо всех индексах в базе данных, в том числе первичных ключах, по одной строке на каждый индекс.

Представление ALL_IND_COLUMNS содержит сведения о колонках, образующих каждый индекс.

(Обратите внимание, что для всех этих представлений поля, содержащие статистику, полученную из фактических данных таблицы, обновляются только при выполнении команды SQL ANALYSE для этой таблицы.)

Специфические для драйверов атрибуты и методы

В DBD::Oracle нет существенных специфических для драйвера атрибутов дескрипторов баз данных или команд.

Поддерживаются следующие частные методы:

plsql_errstr

```
$plsql_errstr = $dbh->func('plsql_errstr');
```

Возвращает текст ошибки из таблицы USER_ERRORS.

dbms_output_enable

```
$dbh->func('dbms_output_enable');
```

Включает пакет DBMS_OUTPUT. Пакет DBMS_OUTPUT обычно используется для извлечения трассировочных и информационных сообщений из хранимых процедур.

dbms_output_get

```
$msg  = $dbh->func('dbms_output_get');
@msgs = $dbh->func('dbms_output_get');
```

Получает отдельную строку или все имеющиеся строки с помощью DBMS_OUTPUT.GET_LINE.

dbms_output_put

```
$msg  = $dbh->func('dbms_output_put', @msgs);
```

Помещает сообщения с помощью DBMS_OUTPUT.PUT_LINE.

DBD::Pg

Общие сведения

Версия драйвера

DBD::Pg версия 0.91.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, только внутренние
LONG/LOB-типы данных	Да, максимальный размер зависит от файловой системы
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?" и ":1" (собственные)
Хранимые процедуры	Нет
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Не учитывается, хранятся в нижнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в нижнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет, но есть нечувствительное к регистру регулярное выражение "~*"
Псевдоколонка ROW ID в серверных таблицах	Да, "oid"
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Эдмунд Мергл (Edmund Mergl). С ним можно связаться через список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль DBD-Pg-0.91 поддерживает PostgreSQL 6.4. Дополнительные сведения можно получить на сервере:

http://www.postgresql.org

Отличия от спецификации DBI

DBD::Pg не производит полный синтаксический анализ команды до ее выполнения. Поэтому такие атрибуты, как $sth->{NUM_OF_FIELDS}, недоступны, пока не будет вызван $sth->execute(). Это нормальное поведение, но его нужно учитывать при переносе приложений, первоначально написанных для других драйверов.

Синтаксис соединения

Синтаксис DBI->connect() «Имя источника данных» (DSN) может быть следующим:

```
dbi:Pg:dbname=$dbname
dbi:Pg:dbname=$dbname;host=$host;port=$port;options=$options;tty=$tty
```

Для всех параметров команды connect, в том числе userid и password, есть значения по умолчанию, жестко зашитые в программный код, которые можно переопределить путем установки соответствующих значений переменных окружения.

Специфических для драйвера атрибутов метода DBI->connect() не существует.

DBD::Pg поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

PostgreSQL поддерживает следующие числовые типы данных:

PostgreSQL	Диапазон
int2	от −32768 до +32767
int4	от −2147483648 до +2147483647
float4	6 десятичных знаков
float8	15 десятичных знаков

На некоторых платформах поддерживается также тип int8. DBD::Pg всегда возвращает числа в виде строк.

Обработка строковых данных

PostgreSQL поддерживает следующие строковые типы данных:

```
CHAR            отдельный символ
CHAR(size)      фиксированной длины с дополнением пробелами
VARCHAR(size)   переменной, но ограниченной длины
TEXT            переменной длины
```

Для всех строковых типов длина ограничена 4096 байтами. Тип CHAR имеет фиксированную длину и дополняется пробелами.

Специальной обработки символов с установленным восьмым битом нет. Они сохраняются в базе данных в неизменном виде. Ни в одном из

символьных типов нельзя хранить встроенные символы null. Кодировка Unicode формально не поддерживается.

Строки можно конкатенировать с помощью оператора ||.

Обработка данных типа «дата»

PostgreSQL поддерживает следующие типы даты/времени:

Тип данных	Размер	Характеристика	Описание
abstime	4 байта	первоначальные дата и время	ограниченный диапазон
date	4 байта	тип SQL92	большой диапазон
datetime	8 байт	наилучший общий тип даты и времени	большой диапазон, высокая точность
interval	12 байт	тип SQL92	эквивалентен timespan
reltime	4 байта	первоначальный временной интервал	ограниченный диапазон, низкая точность
time	4 байта	тип SQL92	большой диапазон
timespan	12 байт	наилучший тип общего интервала времени	большой диапазон, высокая точность
timestamp	4 байта	тип SQL92	ограниченный диапазон

Тип данных	Диапазон		Разрешение
abstime	1901-12-14	2038-01-19	1 секунда
date	4713 до н. э.	32767	1 день
datetime	4713 до н. э.	1465001	1 микросекунда
interval	−178 000 000 лет	+178 000 000 лет	1 микросекунда
reltime	−68 лет	+68 лет	1 секунда
time	00:00:00:00	23:59:59:99	1 микросекунда
timespan	−178 000 000 лет	178 000 000 лет	1 микросекунда
timestamp	1901-12-14	2038-01-19	1 секунда

PostgreSQL поддерживает ряд форматов даты:

Название	Пример
ISO	1997-12-17 0:37:16-08
SQL	12/17/1997 07:37:16.00 PST
Postgres	Wed Dec 17 07:37:16 1997 PST
European	17/12/1997 15:37:16.00 MET
NonEuropean	12/17/1997 15:37:16.00 MET
US	12/17/1997 07:37:16.00 MET

Формат вывода по умолчанию не зависит от местных установок клиента/сервера. Он зависит, в порядке возрастания приоритета, от установки переменной окружения PGDATESTYLE на сервере, переменной окружения PGDATESTYLE у клиента и команды SQL SET DATESTYLE.

Все описанные выше форматы можно использовать для ввода. Можно также использовать большое число других форматов. Особого формата для ввода по умолчанию нет. Если формат ввода даты оказывается двусмысленным, то для разрешения используется значение DATESTYLE.

Если задать значение даты/времени без указания времени, по умолчанию устанавливается время 00:00:00 (полночь). Задать дату/время без указания даты нельзя. Если в дате вводятся две цифры, обозначающие год, то если год меньше 70, к нему прибавляется 2000, в противном случае – 1900.

Текущие дата/время возвращаются с помощью ключевых слов 'now' или 'current', которые должны быть преобразованы в допустимый тип данных:

```
SELECT 'now'::datetime
```

PostgreSQL поддерживает ряд функций даты/времени для преобразования типов данных, извлечения составляющих даты/времени, усечения до заданной единицы и т. д. Со значениями дат и интервалов можно производить обычные арифметические действия, например, дата-дата=интервал и т. д.

Следующее выражение SQL можно использовать для преобразования целого числа секунд после 1 января 1970 года GMT в соответствующее значение дата/время в базе данных:

```
DATETIME(unixtime_field)
```

и для обратного преобразования:

```
DATE_PART('epoch', datetime_field)
```

Сервер хранит все данные приведенными к GMT. Время преобразуется к местному времени на сервере базы данных перед тем, как быть отправленным клиентскому приложению, поэтому по умолчанию время соответствует часовому поясу сервера.

Для задания часового пояса по умолчанию сервер использует переменную окружения `TZ`. На клиентской стороне используется переменная окружения `PGTZ`, которая содержит сведения о часовом поясе, посылаемые серверу при соединении. Для установки временного пояса для текущего сеанса можно использовать команду SQL `SET TIME ZONE`.

Обработка данных типа LONG/BLOB

PostgreSQL обрабатывает `BLOB` с помощью типа данных «large objects». Обработка этого типа отлична от всех остальных типов данных. Данные разбиваются на куски, хранимые в базе данных в кортежах. Доступ к большим объектам предоставляется через интерфейс, созданный в близком соответствии с файловой системой Unix. Максимальный размер ограничивается размером файла операционной системы.

Если просто выбрать поле, то получаются не сами данные, а «идентификатор большого объекта». Атрибуты `LongReadLen` и `LongTruncOk` не реализованы, поскольку в такой конструкции они теряют смысл. Единственный метод, реализованный в драйвере, – это недокументированный метод DBI `blob_read()`.

Другие вопросы обработки данных

Драйвер `DBD::Pg` поддерживает метод `type_info()`.

PostgreSQL поддерживает автоматическое преобразование типов данных там, где оно уместно.

Транзакции, изоляция и блокировка

PostgreSQL поддерживает транзакции. В настоящее время по умолчанию установлен уровень изоляции транзакций Serializable, реализованный путем блокировки на уровне таблиц. То и другое может измениться в будущих версиях. Других уровней изоляции транзакций не поддерживается.

При включенном `AutoCommit` запросы не блокируют таблицы. При чтении не блокируется запись, а при записи не блокируется чтение. Этот режим изменяется с началом транзакции (при выключенном `AutoCommit`). Запрос устанавливает на таблицу блокировку с совместным доступом до окончания транзакции.

Можно использовать команду `LOCK TABLE table_name` для явного блокирования таблицы. Она действует только внутри транзакции (при выключенном `AutoCommit`).

Чтобы гарантировать неизменность выбираемых данных до того момента, когда они позднее будут изменены в транзакции, нужно явным образом блокировать таблицу командой `LOCK TABLE` перед тем, как выполнять выборку.

Диалект SQL

Чувствительность к регистру оператора LIKE

В PostgreSQL есть следующие операторы сравнения строк:

Обозначение	Описание	Пример
~~	То же, что оператор LIKE в SQL	`'scrappy,marc' ~~ '%scrappy%'`
!~~	То же, что оператор NOT LIKE в SQL	`'bruce' !~~ '%al%'`
~	Совпадение (регулярное выражение) с учетом регистра	`'thomas' ~ '.*thomas.*'`
~*	Совпадение (регулярное выражение) без учета регистра	`'thomas' ~* '.*Thomas.*'`
!~	Несовпадение (регулярное выражение) с учетом регистра	`'thomas' !~ '.*Thomas.*'`
!~*	Несовпадение (регулярное выражение) без учета регистра	`'thomas' !~ '.*vadim.*'`

Синтаксис соединения таблиц

Внешние соединения не поддерживаются. Для внутренних соединений используется обычный синтаксис.

Имена таблиц и колонок

Максимальный размер имен таблиц и колонок не может превышать 31 символ. Допустимо использование только буквенно-числовых символов, причем первым символом должна быть буква.

Если идентификатор заключен в двойные кавычки (¨), он может состоять из любых символов, кроме двойных кавычек.

PostgreSQL преобразует все идентификаторы в нижний регистр, если они не заключены в двойные кавычки. Символы национального алфавита можно использовать, если они не заключены в двойные кавычки.

Row ID

Псевдоколонка с идентификатором строки в PostgreSQL называется *oid* (объектный идентификатор). С ней можно работать как со строкой, и использовать для быстрой выборки строк.

Автоматическая генерация ключей и последовательностей

PostgreSQL не поддерживает автоматической генерации ключей, такой как автоинкрементирование или ключи, генерируемые системой.

Однако в PostgreSQL поддерживаются «генераторы последовательностей». В базе данных можно создать любое число именованных генераторов последовательностей. Последовательности используются с помощью функций NEXTVAL и CURRVAL. Вот типичное их использование:

```
INSERT INTO table (k, v) VALUES (NEXTVAL('seq_name'), ?);
```

Чтобы получить только что использованное значение, можно выполнить в том же сеансе соответствующую команду SQL currval() или такую команду:

```
SELECT last_value FROM seq_name
```

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

PostgreSQL не поддерживает обновление и удаление по месту.

Связывание параметров

Связывание параметров эмулируется драйвером. Поддерживаются заполнители ? и :1.

Можно использовать атрибут TYPE метода bind_param() для управления тем, как обрабатываются параметры. Следующие типы SQL связываются как VARCHAR: SQL_NUMERIC, SQL_DECIMAL, SQL_INTEGER, SQL_SMALLINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE и SQL_VARCHAR.

Тип SQL_CHAR связывается как CHAR, что позволяет использовать семантику сравнения с фиксированным размером и дополнением пробелами.

При задании неподдерживаемых значений атрибута TYPE выводятся предупреждения.

Хранимые процедуры

DBD::Pg не поддерживает хранимые процедуры.

Метаданные таблиц

DBD::Pg поддерживает метод table_info().

Таблица pg_attribute содержит подробные сведения обо всех колонках всех таблиц в базе данных, по одной строке на каждую таблицу.

pg_index содержит подробные сведения обо всех индексах в базе данных, в том числе первичных ключах, по одной строке на каждый индекс.

Специфические для драйверов атрибуты и методы

В DBD::Pg нет существенных специфических для драйвера атрибутов дескрипторов баз данных.

В DBD::Pg присутствуют следующие специфические для драйвера атрибуты дескрипторов команд:

pg_size

: Возвращает ссылку на массив целых чисел, соответствующих колонкам. Целое число означает размер колонки в байтах для хранения (не для вывода). Колонки переменного размера обозначаются величиной −1.

pg_type

: Возвращает ссылку на массив строк, соответствующих колонкам. В строке содержится имя типа данных колонки.

pg_oid_status

: Возвращает OID последней выполненной команды INSERT.

pg_cmd_status

: Возвращает имя типа последней команды; возможные значения: INSERT, DELETE, UPDATE и SELECT.

DBD::Pg не имеет частных методов.

Другие существенные характеристики базы данных и драйвера

PostgreSQL предоставляет существенные дополнительные возможности благодаря включению четырех базовых концепций, позволяющих пользователям легко расширять систему: классы, наследование, типы и функции.

Есть и другие функции, увеличивающие мощь и гибкость системы: ограничения, триггеры, правила, целостность транзакций, процедурные языки и большие объекты.

Кроме того, данное программное обеспечение является программным обеспечением с открытым кодом (Open Source) и активно развивается сообществом разработчиков.

DBD::SearchServer

Общие сведения

Версия драйвера

DBD::SearchServer версия 0.20.

Ранее этот драйвер был известен как DBD::Fulcrum.

Краткая характеристика

Транзакции	Нет
Блокировка	Да, явная и неявная
Соединения таблиц	Нет, но см. описание ниже
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?" и ":1" (эмулируются)
Хранимые процедуры	Нет
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Не учитывается, хранятся в верхнем регистре
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Да, "LIKE"
Псевдоколонка ROW ID в серверных таблицах	Да, "FT_CID"
Обновление/удаление по месту	Да
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Дэвид Мигливакка (Davide Migliavacca). С ним можно связаться через список рассылки *dbi-users*. Дэвид Мигливакка не имеет отношения к PCDOCS/Fulcrum, где создан SearchServer, и, в частности, не связан с поддержкой продукта для клиентов PCDOCS/Fulcrum.

Поддерживаемые версии и варианты баз данных

Модуль DBD::SearchServer поддерживает PCDOCS/Fulcrum SearchServer версий от 2.*x* до 3.5.

Fulcrum SearchServer является очень мощной системой извлечения текста с интерфейсом SQL. Не рассчитывайте найти в нем полноценную РСУБД с поддержкой SQL. Подробные сведения о языке запросов можно найти в документации по продукту.

Дополнительные данные о SearchServer можно получить на сервере:

http://www.pcdocs.com

Отличия от спецификации DBI

DBD::SearchServer не производит полного разбора команды до ее выполнения. Такие атрибуты, как $sth->{NUM_OF_FIELDS}, недоступны до вызова $sth->execute(). Это нормальный режим, но важно помнить о нем при переносе приложений, написанных изначально для других драйверов.

Синтаксис соединения

В Unix можно задать для SearchServer местонахождение таблиц базы данных с помощью переменных окружения: FULSEARCH, FULCREATE и FULTEMP. Поэтому строка соединения всегда выглядит просто как:

```
dbi:SearchServer:
```

В WIN32 можно использовать полностью квалифицированный синтаксис DSN с указанием имени источника данных ODBC в качестве третьей составляющей строки соединения:

```
dbi:SearchServer:DSN
```

Специфических для драйвера атрибутов метода DBI->connect() не существует.

DBD::SearchServer поддерживает неограниченное количество одновременных соединений с базами данных одного и того же сервера.

Типы данных

Обработка числовых данных

В SearchServer есть два числовых типа данных: INTEGER и SMALLINT. INTEGER (или INT) является беззнаковым 32-битным двоичным целым с 10-ю цифрами точности. SMALLINT является 16-битным двоичным целым со знаком с 5-ю цифрами точности.

Обработка строковых данных

SearchServer поддерживает следующие строковые типы данных:

```
CHAR(size)
VARCHAR(size)
APVARCHAR(size)
```

Колонки CHAR имеют фиксированный размер, а колонка VARCHAR может иметь переменный размер вплоть до заданного максимального. Если размер не задан, он полагается равным 1. Максимальный размер для колонок CHAR или VARCHAR составляет 32 767.

APVARCHAR является особым типом данных. Можно иметь в таблице не более одной колонки типа APVARCHAR; он используется для хранения

полного текста, предназначенного для индексирования, а также в запросах для извлечения текста. В конечном счете, он модифицируется для отметки мест, в которых запрос выявил совпадение. Максимальный размер колонки APVARCHAR составляет 2 147 483 647.

Тип CHAR имеет фиксированную длину и дополняется пробелами.

В SearchServer есть собственные средства преобразования для национальных наборов символов. Весь текст считается заданным в одном из трех внутренних наборов символов (FTICS). Задача правильного использования наборов символов возлагается на приложение. Программы чтения документов (используемые SearchServer для фактического доступа к документам при индексировании) несут ответственность за транслирование других наборов символов в FTICS. Вместе с пакетом распространяется ряд фильтров для «трансляции».

Поддерживается ISO Latin 1 (8859-1). Подробности использования наборов символов можно найти в разделе «Наборы символов» справочного руководства по SearchSQL.

Обработка данных типа «дата»

SearchServer поддерживает один тип даты/времени: DATE. Значением DATE может быть дата от 1 января 100 года до н. э. до 31 декабря 2047 года с разрешением в один день. Строки таблиц содержат автоматическую колонку только для чтения FT_TIMESTAMP с лучшим разрешением, но ее тип не DATE, а INTEGER. Кроме того, с колонками DATE можно использовать только литералы дат.

Форматом даты служит YYYY-MM-DD (стандарт ISO). Есть заготовки для использования других форматов, но их применение не поощряется.

При вводе допустим только формат даты ISO.

При вводе двузначного года к его значению прибавляется 1900. Однако это не поддерживаемая функция, что является правильным.

Арифметических функций даты/времени нет, как и поддержки временных поясов.

Обработка данных типа LONG/BLOB

Тип APVARCHAR может иметь размер до 2 Гбайт.

LongReadLen и LongTruncOk игнорируются, поскольку семантика типа APVARCHAR совершенно особенная.

Для выборки данных из колонки APVARCHAR нужно использовать недокументированный метод blob_read(). Вставка в колонку APVARCHAR происходит косвенным образом при указании внешнего документа в зарезервированной колонке FT_SFNAME. Данные документа не вставляются в действительности в таблицы, они индексируются. Однако впоследствии можно осуществлять выборку документа из колонки APVARCHAR.

Другие вопросы обработки данных

Драйвер DBD::SearchServer не поддерживает метод type_info().

Транзакции, изоляция и блокировка

DBD::SearchServer не поддерживает транзакции.

Блокировка осуществляется с учетом характеристик таблицы, установленных при ее создании или измененных позднее внешней утилитой ftlock.

По умолчанию используется режим ROWLOCKING, означающий создание временных блокировок при обычных операциях, в число которых входят выборка, обновление с поиском и удаление. Эти блокировки не должны мешать чтению задействованных строк, но предотвращают выполнение одновременных изменений и переиндексирование блокированных строк.

Если установлен режим NOLOCKING, ядро не осуществляет блокировки этой таблицы, и обязанность поддержания целостности данных возлагается на приложение. Пожалуйста, внимательно прочтите документацию, прежде чем экспериментировать с этими параметрами; есть дополнительные эффекты, создаваемые режимами индексирования PERIODIC и IMMEDIATE.

Строки, возвращаемые командой SELECT, можно заблокировать для изменений другими транзакциями, добавив к команде выборки предложение FOR UPDATE.

Средства явной блокировки таблиц нет. Можно запретить модификацию, удаление и даже реиндексирование *схемы* таблиц с помощью команды PROTECT TABLE, но при этом не затрагивается изменение на уровне строк, которое остается разрешенным. Команда UNPROTECT TABLE восстанавливает обычный режим.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE не чувствителен к регистру.

Синтаксис соединения таблиц

SearchServer фактически не поддерживает соединения, однако поддерживает механизм типа *представлений*.

В представлениях таблицы должны располагаться на одном узле и принадлежать одной схеме. Для представлений обеспечивается доступ в режиме только для чтения, а описание их помещается в файл с использованием специального синтаксиса. Более подробные сведения о

представлениях можно найти в руководстве по администрированию и подготовке данных.

Имена таблиц и колонок

В идентификаторах допускается использование букв, цифр и символов подчеркивания (_). Максимальный размер имен таблиц и колонок в данное время не известен.

SearchServer преобразует все идентификаторы в верхний регистр. Имена таблиц и колонок не чувствительны к регистру. Допустимо использование в идентификаторах национальных символов.

Row ID

В SearchServer псевдоколонка идентификатора строки называется `FT_CID` и имеет тип данных `INTEGER`. В предложении WHERE допускается использование `FT_CID`, но только с оператором равенства `=`.

Автоматическая генерация ключей и последовательностей

SearchServer не поддерживает автоматическую генерацию ключей в виде автоинкрементирования или системной генерации ключей. Однако целочисленная псевдоколонка `FT_CID` не возобновляется при удалении строк.

Генераторы последовательностей не поддерживаются.

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

Обновление и удаление по месту поддерживаются с использованием синтаксиса `WHERE CURRENT OF`. Например:

```
$dbh->do("UPDATE ... WHERE CURRENT OF $sth->{CursorName}");
```

Связывание параметров

Заполнители `?` и `:1` поддерживаются путем эмуляции драйвером.

Атрибут `TYPE` метода `bind_param()` игнорируется, поэтому не выводятся предупреждения при использовании неподдерживаемых значений.

Хранимые процедуры

В SearchServer отсутствуют хранимые процедуры и функции.

Метаданные таблиц

DBD::SearchServer поддерживает метод table_info().

В системной таблице COLUMNS содержатся подробные сведения обо всех колонках всех таблиц в базе данных, по одной строке на каждую таблицу. В системной таблице COLUMNS колонка INDEX_MODE указывает на индексированные колонки и режим их индексирования.

Специфические для драйверов атрибуты и методы

В DBD::SearchServer нет специфических для драйвера атрибутов дескрипторов баз данных. Есть один специфический для драйвера атрибут дескрипторов команд:

ss_last_row_id

> Этот атрибут доступен только для чтения и действует после команд INSERT, DELETE или UPDATE. Он содержит значение FT_CID (идентификатора строки) для последней строки, задействованной в команде. Чтобы получить этот атрибут, нужно текущую команду осуществить с командой операции подготовки/выполнения (а не просто выполнить метод do()).

Частных методов нет.

DBD::Sybase – для Sybase и Microsoft SQL Server

Общие сведения

Версия драйвера

DBD::Sybase версия 0.14.

Краткая характеристика

Транзакции	Да
Блокировка	Да, явная и неявная
Соединения таблиц	Да, внутренние и внешние
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?" (собственные), см. ниже
Хранимые процедуры	Да
Связывание выходных значений	Нет, все значения возвращаются через методы выборки
Регистр символов в именах таблиц	Учитывается, хранятся в заданном виде, конфигурируем
Регистр символов в именах полей	Учитывается, хранятся в заданном виде, конфигурируем
Преобразование недопустимых имен	Да, через двойные кавычки
Нечувствительный к регистру оператор "LIKE"	Нет
Псевдоколонка ROW ID в серверных таблицах	Нет

Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Ограничения на дескрипторы команд, см. ниже

Автор и как с ним связаться

Автор драйвера – Майкл Пепплер (Michael Peppler). С ним можно связаться через список рассылки *dbi-users* или по адресу *mpeppler@peppler.org*.

Поддерживаемые версии и варианты баз данных

Модуль `DBD::Sybase` поддерживает Sybase 10.*x* и 11.*x* и предлагает ограниченную поддержку доступа к Sybase 4.*x* и серверам Microsoft MS-SQL в предположении наличия клиента Sybase OpenClient или библиотек FreeTDS.

Доступ к стандартной версии MS-SQL 7 неосуществим из библиотек Sybase, но возможен с использованием библиотек FreeTDS. Есть патч для MS-SQL 7, позволяющий соединяться с ним клиентам Sybase:

http://support.microsoft.com/support/kb/articles/q239/8/83.asp

Библиотеки FreeTDS (*www.freetds.org*) являются проектом Open Source по восстановлению протокола TDS (Tabular Data Stream), используемого как Sybase, так и Microsoft. FreeTDS пока имеется в альфа-версии, но `DBD::Sybase` хорошо компилируется с последним выпуском и поддерживает многие функции (кроме заполнителей `?`).

Вот некоторые URL для получения более специальной информации по драйверу/базе данных:

http://www.sybase.com

http://techinfo.sybase.com

http://sybooks.sybase.com

http://www.microsoft.com/sql

http://www.freetds.org

Отличия от спецификации DBI

Атрибуты `LongReadLen` и `LongTruncOk` не поддерживаются.

Обратите внимание, что `DBD::Sybase` не производит полного разбора команды до ее выполнения. Поэтому такие атрибуты, как `$sth->{NUM_OF_FIELDS}`, недоступны до вызова `$sth->execute()`. Это нормальный режим, но важно помнить о нем при переносе приложений, написанных изначально для других драйверов.

Синтаксис соединения

Синтаксис `DBI->connect()` «Имя источника данных» (DSN) следующий:

```
dbi:Sybase:attrs
```

где attrs является списком пар *ключ=значение*, разделяемых двоеточием. Допустимо использование следующих атрибутов:

server

Указывает сервер Sybase, с которым нужно соединиться.

database

Указывает базу данных на сервере, которая должна быть назначена базой данных по умолчанию для этого сеанса (через USE database).

charset

Указывает используемый клиентом набор символов. Полезен, если клиентский набор символов по умолчанию отличен от используемого сервером. В результате задания этого атрибута будет производиться автоматическое преобразование из одного набора символов в другой.

DBD::Sybase поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Обычно клиенты Sybase не могут готовить/выполнять новый дескриптор команды, пока выполняется выборка из другого дескриптора команды, связанного с тем же дескриптором базы данных. Однако DBD::Sybase эмулирует этот процесс путем установления нового соединения, которое будет автоматически закрыто при уничтожении дескриптора новой команды. Следует иметь в виду, что в результате такого подхода возникают некоторые тонкие, но важные проблемы, связанные с обработкой транзакций.

Типы данных

Обработка числовых данных

Драйвер поддерживает типы данных INTEGER, SMALLINT, TINYINT, MONEY, SMALLMONEY, FLOAT, REAL, DOUBLE, NUMERIC(p,s) и DECIMAL(p,s).

INTEGER является 32-разрядным целым, SMALLINT – 16-разрядным и TINYINT – 8-разрядным. Все остальные типы данных, кроме NUMERIC/DECIMAL, зависят от аппаратной платформы. Точность NUMERIC/DECIMAL находится в пределах от 1 до 38, а масштаб – от 0 до 38.

По умолчанию значения NUMERIC/DECIMAL возвращаются как строки Perl, даже если масштаб равен 0, а точность достаточно мала, чтобы уместиться в значении целого типа. Все остальные числа возвращаются в собственном формате.

Обработка строковых данных

DBD::Sybase поддерживает CHAR, VARCHAR, BINARY и VARBINARY, при этом длина каждого из них не превышает 255 символов. Тип CHAR имеет фиксированную длину и дополняется пробелами.

Sybase производит автоматическое преобразование данных CHAR и VARCHAR между набором символов сервера (см. системную таблицу syscharset) и набором символов клиента, определяемым его местными установками. Типы данных BINARY и VARBINARY не преобразуются. UTF-8 поддерживается.

Дополнительно о проблемах использования наборов символов можно прочесть в руководстве по Sybase OpenClient.

Конкатенация строк производится с помощью оператора SQL «+».

Обработка данных типа «дата»

Sybase поддерживает типы DATETIME и SMALLDATETIME. Тип DATETIME может иметь значение от 1 января 1753 года до 31 декабря 9999 года с разрешением в одну трехсотую секунды. Тип SMALLDATETIME имеет диапазон от 1 января 1900 года до 6 июня 2079 года с разрешением в одну минуту.

Текущую дату на сервере можно получить с помощью функции SQL GETDATE().

Формат даты в Sybase зависит от установок местности клиента. Формат даты по умолчанию основан на «C»:

```
Feb 16 1999 12:07PM
```

Эта же установка допускает использование в Sybase еще нескольких форматов даты в дополнение к выше указанному:

```
2/16/1998 12:07PM
1998/02/16 12:07
1998-02-16 12:07
19980216 12:07
```

Если время опущено, оно устанавливается равным 00:00. Если опущена дата, она устанавливается равной 1 января 1900 года. Если опущен век и год меньше 50, то предполагается 2000, а если год больше или равен 50, то предполагается 1900.

Особый частный метод _date_fmt() (вызов которого происходит через $dbh->func()) используется для изменения формата ввода и вывода даты. Форматы основаны на стандартных функциях преобразования Sybase. Реализовано следующее подмножество имеющихся форматов:

```
LONG        - Nov 15 1998 11:30:11:496AM
SHORT       - Nov 15 1998 11:30AM
DMY4_YYYY   - 15 Nov 1998
MDY1_YYYY   - 11/15/1998
DMY1_YYYY   - 15/11/1998
HMS         - 11:30:11
```

Для преобразования значений даты и времени из других форматов используйте функцию SQL CONVERT(), например:

```
UPDATE a_table
   SET date_field = CONVERT(datetime_field, '1999-02-21', 105)
```

CONVERT() является общей функцией преобразования, позволяющей осуществлять взаимное преобразование между большинством типов данных.

Арифметические операции над данными типа дата/время производятся с помощью функций DATEADD(), DATEPART() и DATEDIFF() языка Transact SQL, например:

```
SELECT DATEDIFF(ss, date1, date2)
```

возвращает разницу в секундах между date1 и date2.

Sybase совсем не воспринимает часовые пояса, исключая функцию SQL GETDATE(), возвращающую дату во временном поясе, в котором работает сервер (через localtime).

Следующее выражение SQL можно использовать для преобразования целого числа секунд, истекших с 1 января 1970 года («время Unix»), в соответствующее значение даты/времени в базе данных:

```
DATEADD(ss, unixtime_field, 'Jan 1 1970')
```

Учтите, однако, что сервер не воспринимает временные пояса и потому выдаст локальное время Unix на сервере, а не правильное значение для часового пояса GMT.

Если известно, что сервер и клиент работают в одном часовом поясе, можно использовать код:

```
use Time::Local;
$time_to_database = timegm(localtime($unixtime));
```

для преобразования значения времени Unix перед его отправкой серверу Sybase.

Для обратного преобразования даты в базе данных в время Unix можно использовать:

```
DATEDIFF(ss, 'Jan 1 1970', datetime_field)
```

То же предостережение относительно местного времени и GMT действует и в этом случае. Если известно, что сервер и клиент работают в одном часовом поясе, можно преобразовать возвращенное значение в правильное время GMT следующим выражением Perl:

```
use Time::Local;
$time = timelocal(gmtime($time_from_database));
```

Обработка данных типа LONG/BLOB

Sybase поддерживает типы IMAGE и TEXT для данных LONG/BLOB. Каждый из типов может хранить до 2 Гбайт двоичных данных, в том числе символов nul. Основное отличие между типами колонок IMAGE и TEXT заключается в том, как библиотеки клиента обрабатывают входные и выходные данные. Данные TEXT вводятся и возвращаются такими, как есть («as is»). Данные IMAGE возвращаются в виде длинной строки шестнадцатеричных символов и вводиться должны в таком же виде.

Атрибуты LongReadLen и LongTrunkOk не действуют. По умолчанию максимальный размер данных TEXT/IMAGE составляет 32 Кбайт, но его можно изменить командой Transact-SQL SET TEXTSIZE.

Связанные параметры нельзя использовать для ввода в Sybase данных типа TEXT или IMAGE.

Другие вопросы обработки данных

Драйвер DBD::Sybase пока не поддерживает метод type_info().

Sybase не производит автоматического преобразования чисел в строки и обратно. Необходимо явным образом вызывать функцию SQL CONVERT. Однако для заполнителей не требуется специальной обработки, поскольку драйверу DBD::Sybase известно, какой тип должен иметь каждый заполнитель.

Транзакции, изоляция и блокировка

DBD::Sybase поддерживает транзакции. По умолчанию установлен уровень изоляции транзакций READ COMMITTED.

Sybase поддерживает уровни изоляции READ COMMITED, READ UNCOMMITED и SERIALIZABLE. Уровень можно изменить для соединения или команды, выполнив SET TRANSACTION_ISOLATION LEVEL x, где x равен 0 для READ UNCOMMITED, 1 для READ COMMITED и 3 для SERIALIZABLE.

По умолчанию запрос на чтение устанавливает совместную блокировку для каждой считанной страницы. Это позволяет другим процессам производить чтение из таблицы, но блокирует процессы, пытающиеся получить монопольную блокировку (для обновления). Совместная блокировка устанавливается только на время, которое реально необходимо серверу для чтения страницы, а не на все время выполнения команды SELECT. (Серверы версии 11.9.2 и более поздних имеют различные дополнительные механизмы блокировки.)

Явной команды LOCK TABLE нет. Добавление предложения WITH HOLDLOCK к команде SELECT можно использовать для принудительной установки монопольной блокировки таблицы, но это редко требуется.

Чтобы выполнить в Sybase обновление, в котором участвует несколько таблиц, лучше всего заключить всю операцию в транзакцию. Тем самым гарантируется, что блокировки будут установлены в должном порядке и никакой посторонний процесс не изменит строки, которые ваша операция модифицирует в данное время.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор `LIKE` чувствителен к регистру.

Синтаксис соединения таблиц

Внешние соединения осуществляются с помощью операторов `=*` (правое внешнее объединение) и `*=` (левое внешнее объединение):

```
SELECT customers.customer_name, orders.order_date
FROM customers, orders
WHERE customers.cust_id =* orders.cust_id
```

Для всех строк в таблице `customers`, не имеющих соответствующих строк в таблице `orders`, Sybase возвращает NULL для всех выражений списка выборки, содержащих колонки таблицы `orders`.

Имена таблиц и колонок

Имена идентификаторов, в частности таблиц и колонок, не могут превосходить в длину 30 символов.

Первым символом должна быть буква (в соответствии с текущим набором символов сервера) или символ подчеркивания (_). Последующие символы могут быть буквами или символами (@), (#) и (_). Идентификаторы не могут иметь в своем составе пробелы или символы (%), (!), (^), (*), (.). Кроме того, идентификаторы не должны входить в список зарезервированных слов (их полный список можно найти в документации по Sybase).

Имена таблиц или колонок можно заключать в кавычки, если включена опция `set quoted_identifier`. При этом пользователь может обойти запрещение использовать зарезервированные слова. Если эта опция установлена, строки символов в двойных кавычках рассматриваются как идентификаторы, а в одинарных кавычках – как литералы.

По умолчанию различается регистр символов в идентификаторах. Этот режим можно изменить, установив для сервера другой порядок сортировки по умолчанию.

Символы национального алфавита можно использовать в идентификаторах без заключения их в кавычки.

Row ID

Sybase не поддерживает псевдоколонку «идентификатор строки».

Автоматическая генерация ключей и последовательностей

Sybase поддерживает функцию `IDENTITY` для автоматической генерации ключей. Если объявить в таблице колонку `IDENTITY`, то при каждой вставке будет генерироваться новое значение. Присваиваемые значения всегда возрастают, но их непрерывность не гарантируется.

Чтобы получить значение, сгенерированное и использованное при последней вставке, можно выполнить команду:

```
SELECT @@IDENTITY
```

Sybase не поддерживает генераторы последовательностей, хотя достаточно просто написать хранимые процедуры для генерации последовательностей.[1]

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

Sybase не поддерживает обновления и удаления по месту.

Связывание параметров

Связывание параметров в Sybase поддерживается непосредственно. Однако есть два недостатка, о которых нужно знать.

Во-первых, `DBD::Sybase` создает внутреннюю хранимую процедуру для каждого вызова `prepare()` с заполнителями `?`. Эти хранимые процедуры помещаются в базу данных *tempdb* и уничтожаются только при закрытии соединения. Можно легко превысить допустимые размеры *tempdb*, если в сценарии выполняется много вызовов `prepare()` с заполнителями.

Во-вторых, поскольку все временные хранимые процедуры создаются в *tempdb*, это может вызвать неприятности из-за блокировки системных таблиц в *tempdb*. Эта проблема, вероятно, будет устранена в следующих выпусках Sybase (11.9.4 или 12.0).

Заполнители вида `:1` не поддерживаются, и атрибут `TYPE` метода `bind_param()` в данное время игнорируется, поэтому для неподдерживаемых

[1] См. на *http://techinfo.sybase.com/css/techinfo.nsf/DocID/ID=860* полное изложение различных имеющихся возможностей.

значений не генерируются предупреждения. И наконец, попытка связать тип данных TEXT или IMAGE окажется неудачной.

Хранимые процедуры

В Sybase хранимые процедуры пишутся на Transact-SQL, являющемся процедурным расширением Sybase для SQL.

Хранимые процедуры вызываются точно так же, как обычные команды SQL, и могут возвращать результаты тех же типов (т. е. SELECT в хранимой процедуре можно извлекать через $sth->fetch()).

Если хранимая процедура возвращает данные через параметры OUTPUT, то вначале их нужно объявить:

```
$sth = $dbh->prepare(qq[
   declare \@name varchar(50)
   exec getName 1234, \@name output
]);
```

Хранимые процедуры нельзя вызывать с параметрами (?). Следующий код недопустим:

```
$sth = $dbh->prepare("exec my_proc ?");  # недопустимо
$sth->execute($foo);
```

Вместо этого нужно использовать код:

```
$sth = $dbh->prepare("exec my_proc '$foo'");
$sth->execute();
```

Поскольку хранимые процедуры Sybase почти всегда возвращают более одного результирующего набора, следует выполнять цикл, пока значение syb_more_results не станет равно 0:

```
do {
  while($data = $sth->fetch) {
     ...
  }
} while($sth->{syb_more_results});
```

Метаданные таблиц

DBD::Sybase поддерживает метод table_info().

В таблице syscolumns есть одна строка для каждой колонки каждой таблицы. Подробности см. в определениях системных таблиц Sybase. Однако простейшим способом получения метаданных таблиц является использование хранимой процедуры sp_help.

Оптимальным способом получения подробных сведений об индексах таблицы является использование хранимой процедуры sp_helpindex (или sp_helpkey).

Специфические для драйверов атрибуты и методы

В DBD::Sybase есть следующие специфические для драйвера атрибуты дескрипторов баз данных:

syb_show_sql

Если он установлен, то в строку, возвращаемую $dbh->errstr, включается текущая команда SQL.

syb_show_eed

Если он установлен, то в строку, возвращаемую $dbh->errstr, включается расширенная информация об ошибке. В нее, например, может входить сообщение об индексе, вызвавшем ошибку при вставке повторяющегося значения.

В DBD::Sybase есть следующие специфические для драйвера атрибуты дескрипторов команд:

syb_more_results

Описание в другом месте данного документа.

syb_result_type

Возвращает численное значение типа текущего результирующего набора. Полезен при выполнении хранимых процедур для определения того, какой тип данных может быть извлечен в данный момент (обычные строки выборки, выходные параметры, статус окончания и т. д.).

Обеспечивается один частный метод:

_date_fmt

Устанавливает форматы преобразования и отображения дат, используемые по умолчанию. См. описание в данном документе.

Другие существенные характеристики базы данных и драйвера

Sybase и DBD::Sybase позволяют в одном вызове подготовить несколько команд и в одном вызове их выполнить. Результаты возвращаются клиенту как поток табулированных данных. Хранимые процедуры могут также возвращать поток множественных наборов данных. Каждый отдельный набор результатов рассматривается как один обычный результирующий набор, поэтому fetch() возвращает undef в конце каждого набора. Чтобы узнать, остались ли еще наборы данных, можно проверить значение атрибута syb_more_results. Вот типичный цикл, в котором используется эта специфическая особенность Sybase:

```
do {
  while($d = $sth->fetch) {
    ... do something with the data
```

```
    }
} while($sth->{syb_more_results});
```

Sybase обладает богатыми и мощными функциями хранимых процедур и триггеров, использование которых поощряется.

DBD::XBase

Общие сведения

Версия драйвера

DBD::XBase версия 0.145.

Краткая характеристика

Транзакции	Нет
Блокировка	Нет
Соединения таблиц	Нет
LONG/LOB-типы данных	Да, до 2 Гбайт
Доступность атрибутов дескриптора команды	После execute()
Заполнители	Да, "?" и ":1" (эмулируются)
Хранимые процедуры	Нет
Связывание выходных значений	Нет
Регистр символов в именах таблиц	Учитывается, хранятся в заданном виде
Регистр символов в именах полей	Не учитывается, хранятся в верхнем регистре
Преобразование недопустимых имен	Нет
Нечувствительный к регистру оператор "LIKE"	Да, "LIKE"
Псевдоколонка ROW ID в серверных таблицах	Нет
Обновление/удаление по месту	Нет
Одновременное использование нескольких дескрипторов	Не ограничено

Автор и как с ним связаться

Автор драйвера – Ян Паздзиора (Jan Pazdziora). С ним можно связаться через *adelton@fi.muni.cz* или список рассылки *dbi-users*.

Поддерживаемые версии и варианты баз данных

Модуль DBD::XBase поддерживает разновидности файлов *dbf:* dBaseIII и IV, варианты Fox*, в том числе файлы полей memo *dbt* и *fpt*.

Очень подробные сведения о формате XBase и многочисленные ссылки можно найти на сервере:

http://www.e-bachmann.dk/docs/xbase.htm

Отличия от спецификации DBI

DBD::XBase не производит полного разбора команды до ее выполнения. Поэтому такие атрибуты, как $sth->{NUM_OF_FIELDS}, недоступны до вызова $sth->execute(). Это нормальный режим, но важно помнить о нем при переносе приложений, написанных изначально для других драйверов.

Синтаксис соединения

DBI->connect() «Имя источника данных» (DSN) должен в третьей части содержать каталог, где находятся файлы *dbf*:

```
dbi:XBase:/path/to/directory
```

По умолчанию это текущий каталог.

Специфических для драйвера атрибутов метода DBI->connect() нет.

DBD::XBase поддерживает неограниченное число одновременных соединений с одной или несколькими базами данных.

Типы данных

Обработка числовых данных

DBD::XBase поддерживает общие типы NUMBER(p,s), FLOAT(p,s) и INTEGER(l). Максимальные масштаб и точность ограничиваются возможностями Perl по обработке чисел. В файлах *dbf* числа хранятся как строки ASCII, двоичные целые или числа с плавающей запятой.

В существующих файлах *dbf* типы полей определяются в заголовке. Числовые типы могут храниться в виде строк ASCII или некотором двоичном формате. DBD::XBase (с помощью XBase.pm) разбирает эти данные и производит чтение и запись в поля в этом формате.

При создании нового файла *dbf* с помощью команды CREATE TABLE числовые поля всегда создаются традиционным для XBase способом в виде строки ASCII. (Модуль XBase.pm предоставляет возможности управления этим процессом.)

Числовые поля всегда возвращаются как числовые величины Perl, а не строки. Следовательно, невозможно использовать числа вне диапазона, допустимого в Perl. В будущем это ограничение, возможно, будет снято.

Обработка строковых данных

В DBD::XBase есть типы CHAR(length) и VARCHAR(length).

Для обоих типов максимальная длина составляет 65 535 символов.[1]

Добавление пробелов производится и в типе CHAR, и в типе VARCHAR, поэтому ChopBlanks применим к обоим.

Обработка данных с установленным восьмым битом производится прозрачным образом. Преобразования национальных наборов символов не производятся. Поскольку строковые типы могут хранить двоичные данные, возможно хранение строк Unicode.

Обработка данных типа «дата»

DBD::XBase поддерживает следующие типы даты/времени:

```
DATE
DATETIME
TIME
```

Тип DATE содержит строку из восьми символов в формате YYYYMMDD. Только этот формат можно использовать при вводе и выводе. DBD::XBase не проверяет допустимость вводимых значений.

Типы DATETIME и TIME внутренне хранят 4-байтовое целое значение дня и 4-байтовое целое значение секунд (в 1/1000 секунды). DBD::XBase вводит и выводит эти типы в стиле Unix с использованием числа с плавающей запятой, представляющего «число секунд, истекших с начала эпохи» (возможно, с дробными знаками). В будущем этот режим может быть изменен.

Способ получения текущих даты/времени отсутствует, функции SQL для работы с датами/временем не поддерживаются. Понятие временных поясов также отсутствует.

Обработка данных типа LONG/BLOB

DBD::XBase поддерживает тип данных MEMO. В качестве псевдонима MEMO можно использовать BLOB. В полях типа MEMO можно хранить строки размером до 2 Гбайт (для всех типов memo-файлов XBase).

В файлах *dbt* формата dBaseIII поля memo не могут содержать байт 0x1A. В файлах dbt/fpt форматов dBaseIV и Fox* могут содержаться любые символы.

Для выборки и вставки полей типа MEMO не требуется специальных методов обработки. Атрибуты LongReadLen и LongTruncOk в данное время игнорируются.

[1] Эта граница действует несмотря на то, что в старых версиях dBases допускалась длина, равная лишь 254 символам. Поэтому вновь созданные файлы *dbf* могут не работать со старыми XBase-совместимыми программами.

Другие вопросы обработки данных

Драйвер DBD::XBase поддерживает метод type_info().

DBD::XBase поддерживает автоматическое преобразование типов данных там, где оно уместно.

Транзакции, изоляция и блокировка

DBD::XBase не поддерживает транзакции и не производит блокировки таблиц, с которыми работает.

Диалект SQL

Чувствительность к регистру оператора LIKE

Оператор LIKE не чувствителен к регистру.

Синтаксис соединения таблиц

DBD::XBase не поддерживает соединения таблиц.

Имена таблиц и колонок

В формате XBase каждая таблица хранится в отдельном файле. Поля MEMO хранятся в дополнительных файлах. Имена таблиц ограничены максимальной длиной имени файла, поддерживаемой файловой системой. Они сохраняются и обрабатываются в том виде, в котором введены. Чувствительность к регистру зависит от свойств файловой системы, в которой хранятся файлы.

Названия колонок ограничиваются одиннадцатью символами. Они хранятся в верхнем регистре и нечувствительны к регистру.

Имена таблиц и колонок должны начинаться с буквы, за которой может следовать любая комбинация букв, цифр и символов подчеркивания. Допустимо использование символов национального алфавита.

DBD::XBase не поддерживает заключение в кавычки имен таблиц и колонок.

Row ID

DBD::XBase не поддерживает псевдоколонку «идентификатор строки».

Автоматическая генерация ключей и последовательностей

DBD::XBase не поддерживает автоматическую генерацию ключей и генераторы последовательностей из-за ограничений формата XBase.

Автоматическая нумерация строк и предельное число строк

Автоматическая нумерация строк и ограничение числа строк не поддерживаются.

Обновление и удаление по месту

DBD::XBase не поддерживает обновления и удаления по месту.

Связывание параметров

Связывание параметров реализовано в драйвере и поддерживает заполнители вида ?, :1 и :name.

Атрибут TYPE метода bind_param() игнорируется. Следовательно, неподдерживаемые значения атрибута TYPE не приводят к выводу предупреждений.

Хранимые процедуры

Хранимые процедуры неприменимы в формате XBase.

Метаданные таблиц

DBD::XBase поддерживает метод table_info.

В настоящее время единственный способ получить подробные сведения о колонках таблицы – это выполнить команду SELECT * FROM table и воспользоваться атрибутами NAME и TYPE дескриптора команды.

Ключи и индексы в данное время не поддерживаются.

Специфические для драйверов атрибуты и методы

DBD::XBase обладает одним специфическим для драйвера атрибутом, относящимся как к дескрипторам баз данных, так и к дескрипторам команд:

xbase_ignorememo

> Игнорировать файлы memo, чтобы прочесть таблицу, для которой файл memo потерян или поврежден.

В DBD::XBase нет полезных частных методов.

C

Хартия священных площадок ASLaN

Если данная книга вызвала у вас интерес к мегалитическим площадкам, пожалуйста, прочтите документ, в котором рассказывается, как нужно вести себя в этих местах. За прошедшие века многие площадки были безвозвратно утрачены нами из-за вандализма и преднамеренного разрушения. Нам бы хотелось, чтобы немногие сохранившиеся до наших дней мегалиты не подвергались дальнейшему уничтожению.

В попытках сохранить наши разрушающиеся мегалитические площадки, книги, посвященные им, часто включают целый ряд просьб. Простая рекомендация «не портить мегалитические площадки» больше подошла бы для таких книг, но, к сожалению, ее часто игнорируют. Мы включили этот документ в свою книгу, надеясь, что его влияние на умы будет достаточно сильным и приведет к глубокому пониманию имеющихся сегодня проблем.

Мы хотели бы также обратить ваше внимание на то, что многие площадки находятся под охраной государства, а потому разрушительное воздействие на них является *противозаконным*.

Отложив в сторону возможное наказание и хандру, вспомним о том, что площадки существуют для того, чтобы дарить радость и открывать свой тайный смысл разным людям. Радуйтесь!

- Пожалуйста, постарайтесь при посещении площадок оставлять их в таком состоянии, в котором их было бы приятно обнаружить следующим посетителям. Уважайте землю и ее обитателей – людей, животных и растения.
- Рытье ям с любыми целями вредит растениям, а возможно, насекомым и археологическим останкам. Разрушение археологических

останков затруднит нам и нашим потомкам изучение исторического прошлого площадки. Ущерб, нанесенный площадке с любой стороны, наносит ущерб духу местности.

- Разжигание костров может нанести такой же урон, как и копание земли. Огонь может повредить стоящие камни – при перегреве они раскалываются. Летом возможно быстрое распространение огня, при котором погибает природа, а проверить, что костер действительно погас, бывает очень трудно. Костры наносят ущерб археологическим изысканиям, мешают геофизическим исследованиям и загрязняют археологические слои пеплом и древесным углем. Перегрев, свечной воск и надписи повреждают мхи и лишайники, для восстановления которых могут понадобиться десятилетия. Ущерб, наносимый кострами, губит дух местности.
- Если вам покажется уместным совершить жертвоприношение, подумайте обо всех его последствиях. Не оставляйте искусственные материалы. Тщательно отберите предметы жертвоприношения, чтобы их нельзя было принять за мусор. Не закапывайте вещи. Разлагаемые микроорганизмами жертвы гниют – имейте это в виду, оставляя их. Если на площадке уже есть жертвоприношения, подумайте о том, стоит ли оставлять еще.
- Пожалуйста, не уносите с площадки ничего, кроме мусора. Многие растения, произрастающие вокруг священных площадок, являются необычными или редкими, поэтому не рвите цветы. Не уносите с собой камни – они могут быть важной частью площадки, что не всегда очевидно.
- В прежние времена соблюдалась традиция не оставлять следов каких-либо ритуалов из-за возможности преследования. Стоит возродить эту традицию, поскольку в ней проявляется уважение к природе и духу места.
- Не надо ничего менять на площадке; пусть площадка изменит вас.

ASLaN – это Сообщество древних священных ландшафтов (Ancient Sacred Landscape Network), образованное для того, чтобы обратить внимание общественности на необходимость защиты и сохранения священных площадок, их состояния, ухода за ними и осуществления доступа на них. Дополнительную информацию об ASLaN можно найти на сервере:

http://www.symbolstone.org/archaeology/aslan

Алфавитный указатель

D

З

И

К

Л

М