# 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.

# 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.