Спецификация DBI

Это слегка отредактированная версия спецификации DBI, «живой документ», медленно, но верно развивающаяся 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 этот документ окажется несколько устаревшим к тому моменту, когда вы станете его читать, мы все-таки считаем, что это важный справочный материал. 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 c*hanges* (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;
@dnver_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_nuin     $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_vanable);
$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 OBI FAQ, и читать его можно, выполнив команду `perldoc DBIFAQ`,
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 гобой 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. 1).
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.

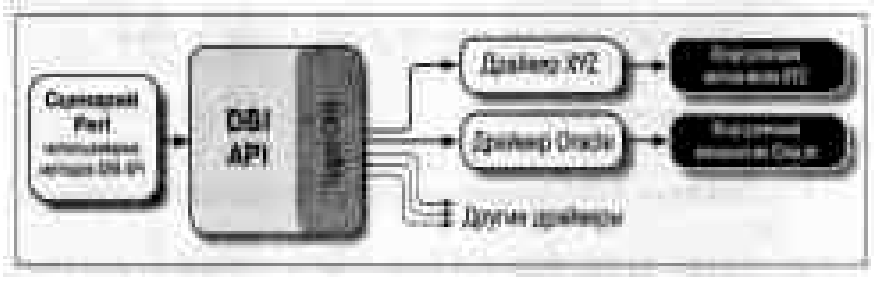

Рис. .1. Архитектура приложения DBI

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:

```
  $stn = $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 осуществления 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 $stn = $dDh->prepare(q{INSERT INTO sales (product_code, qty,
  price) VALUES (?,?,?)}) || die $dbh->errstr;

while (<>) {

   chop;
   my ($product_code,  $qty,  $pnce) = 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;
$dbh = D8I->connect($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 *$data_source* имеет неопределенное значение или пуст, то DBI подставит значение из переменной окружения *DBI_DSN*. Если пустой является только часть *dnver_name* (т. е. префиксом *$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->comect 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 *AutoCoimt* и *PrintError* по умолчанию включены для каждого соединения. (Дальнейшие подробности см. в описании *AutoCommit* и *PrintErrror*.) Однако весьма рекомендуется явным образом определить *AutoComit*, 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 *$data_source* не должен начинаться с *dbi:dnver_name* . (В противном случае встроенный *dnver_name* будет проигнорирован.) Заметим также, что в этой старой форме connect атрибут *$dbh->{AutoCommit}* имеет *неопределенное значение,* атрибут *$dbh->{PnntError}* выключен, а если *DBI_DSN* не определен, то проверяется старая переменная окружения *DBI_DBNAME*. Учтите, что из будущих версий DBI этот connect «в старом стиле» может быть исключен.

connect_cached (новый)

```
$dbh = D8I->connect_cached($data_source,  $username, $password)
        || die $DBI::errstr;
$dbn = DBI->connect_cached($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 списке рассылки *dbi-users,*

К кэшу можно обращаться (и очищать его) с помощью атрибута *Cached-Kids.*

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($dnver);
@ary = DBI->data_sources($dnver, \%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 *DBI_DRIVER*.

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). Неотображаемые 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*).

neatlist

```
$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_nufrber(@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:: errst

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

```
Sstr = $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);
$n->trace($trace level   $trace_filenane);
```

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-XTRACE* 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->{AttnbuteName} = ...          # установить/записать
...= $h->{AttnbuteName};          # 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 дескриптора 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 *pnvate_*. 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 *pnvate_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_ret=   $dbh->selectall_arrayref($statement, \%attr)
$ary_ref=   $dbh->selectall_arrayref($statement, \%attr,  @bind_values);
```

Этот вспомогательный метод объединяет в одном вызове *prepare, execute* и *fetchall_artayref* .
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 *Cached-Kids*.

Вот пример возможного использования *prepare_cached*:

```
while (($field,  $value) = each %search_fields ) {
   push @sql,  "$field = ?";
   push ©values,  Svalue;
}
$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», «ALIAS», «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
     LOCALJYPEJAME      => 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 «Ox» 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 «*maxlength*». Для типов, к которым это не применимо, возвращается NULL (*undef*).

NULLABLE (целое число)

Указывает на то, может ли тип данных принимать значение NULL: O = 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 = spnntf "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->{AutoComimt};          # 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_pararo($p_num,$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_pararfi(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 *bind_param*. 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_nu[n, \$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_pararn_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 *$stn->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 запроса. 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 = $stn->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 стать недоступными, 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->birid_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 "Sregion: $sales\n"; }
```

Для обеспечения совместимости со старыми сценариями первый параметр игнорируется, если он является undef или ссылкой на хэш.

dump_results

```
$rows = $sth->dump__results($roaxlen,  $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 *$fsep* (по умолчанию *"\n"*). *$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 Бане).

«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-archwes/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/tutoriall.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

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 Уильяме, 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://cronoplo.net/perl/
Японский - http://member.nifty.ne.ip/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/

DBD::mysql и DBD::mSQL

Общие сведения-

Версия драйвера

`DBD::mysql и DBD::mSQL версий 1.20xx и 1.21_xx.`

Версии *1.20xx* (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 *1.22xx*.

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	*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	*Возможны варианты см ниже*
Псевдоколонка ROW ID в серверных	*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 *"i.20xx* 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 *l.20xx* и *1.21_xx* поддерживают все версии mSQL вплоть до mSQL2.0.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	-147483648..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 FLOATS 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 2,2250738585072014e-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 *NULL*. (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 DATE-TIME 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), HO-UR(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:

```
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)
LONT6LOB / LONGTEXT        максимальный размер 4294967295 (2**32 - 1)
```

[1] См. *http://www.unix-systems.org/version2/whatsnew/year2000.html.*

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 *TO-LOWER*.

В mSQL три оператора LIKE: LIKE чувствителен к регистру, CLIKE не чувствителен и PLIKE 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 делать, 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, Sname, 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:

```
LIST FIELDS $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:

```
L1STINDEX Stable $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 *LIST FIELDS* 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*.