# Perl DBI 中文帮助文档

## Contents

## 名称

DBI—Perl 的数据库独立接口。

## 概览

```
use DBI;

@driver_names = DBI->available_drivers;

@data_sources = DBI->data_sources($driver_name);


$dbh = DBI->connect($data_source, $username, $auth);

$dbh = DBI->connect($data_source, $username, $auth, \%attr);


$rv  = $dbh->do($statement);

$rv  = $dbh->do($statement, \%attr);

$rv  = $dbh->do($statement, \%attr, @bind_values);


@row_ary = $dbh->selectrow_array($statement);

$ary_ref = $dbh->selectall_arrayref($statement);


$sth = $dbh->prepare($statement);

$sth = $dbh->prepare_cached($statement);

$rv = $sth->bind_param($p_num, $bind_value);

$rv = $sth->bind_param($p_num, $bind_value, $bind_type);

$rv = $sth->bind_param($p_num, $bind_value, \%attr);

$rv = $sth->execute;

$rv = $sth->execute(@bind_values);
```

```
$rc = $sth->bind_col($col_num, \$col_variable);

$rc = $sth->bind_columns(\%attr, @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;
```

## 注意

该 DBI 是 DBI 1.0.6 版本的。

当前 DBI 规范更新的速度非常快，因此用户需要检查是否具有最新版本的拷贝。RECENT CHANGES 部分描述了用户可见的改变和与 DBI 模块一起提供的改变文件。

另外需要注意的是，无论何时 DBI 改变了，驱动器都需要一点时间来跟上。最近版本的 DBI 可能包含了许多新功能，并且这些可能并不被正在使用的驱动器支持。如果需要这些功能，可以告诉你的驱动器的供应商。

同时请阅读 DBI FAQ，其被安装为 DBI::FAQ，因此可以通过执行 perldoc DBI::FAQ 命令阅读。

## 最近的更改

最近版本中重要的用户可见的改变的总结（如果没有提及最近的版本，那么它仅仅意味着该版本中没有重要的用户可见的改变。）

### DBI 1.00 - 14th August 1998

增加了$dbh->table_info。

### DBI 0.96 - 10th August 1998

增加了$sth->{PRECISION}和$sth->{SCALE}；增加了 DBD::Shell 和 dbish 交互式 DBI shell。任何数据库属性可以通过 DBI->connect(,,, \%attr)设置。为 Perl 驱动器开发者增加了 _get_fbav 和_set_fbav 方法；DBI 跟踪显示增加了`` at yourfile.pl line nnn''；PrintError 和 RaiseError 当前优先考虑驱动器和方法名；增加了$dbh->{Name}；增加了$dbh->quote($value, $data_type)；增加了 DBD::Proxy 和 DBI::ProxyServer；增加了$dbh->selectall_arrayref 和 $dbh->selectrow_array 方法；增加了$dbh->table_info；增加了$dbh->type_info 和$dbh->type_info_all；增加了$h->trace_msg($msg)已写入跟踪文件；增加了@bool = DBI::looks_like_number(@ary)。

### DBI 0.92 - 4th February 1998

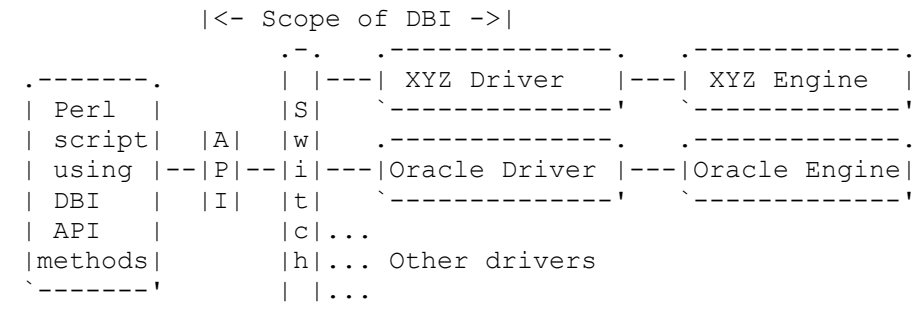
增加了$dbh->prepare 的缓存变种$dbh->prepare_cached()。增加了新的属性：Active, Kids, ActiveKids, CachedKids。增加了对通用目的的'private_'属性的支持。

## 描述

Perl DBI 是一个 Perl 语言的数据库访问 API。DBI 定义了一系列函数，变量和惯例提供一个独立于具体数据库的一致性的数据库接口。

记住 DBI 仅仅是一个接口是很重要的。在应用程序和一个/多个数据库驱动器之间的一个瘦层。驱动器完成实际的工作。DBI 为驱动器在其内工作提供了标准和框架。

## DBI 应用程序的架构



`_'

API 是应用程序 Perl-脚本（编程）接口。调用接口和变量由 DBI 提供到 perl 脚本。API 由 DBI Perl 扩展实施。

'Switch'是用来为实际的执行调度 DBI 方法调用到恰当的驱动器的一段代码。Switch 还负责动态加载驱动器，错误检查/处理以及其他职责。DBI 和 Switch 通常是同义词。

驱动器负责实施支持给定类型的引擎（数据库）。驱动器包含使用相应引擎的私有借口函数编写的 DBI 方法的实现。只有复杂/多数据库应用程序以及通用库函数的作者需要关心驱动器。

## 标记和惯例

DBI 静态顶层类名；

$dbh 数据库处理对象；

$sth 语句处理对象；

$drh 驱动器处理对象（很少可见或在应用程序中使用）

$h 以上任何$??h 处理类型；

$rc 常用返回码（boolean: true=ok, false=error）

$rv 常用返回值（通常为整形）

@ary 从数据库返回的值列表，通常为一行；

$rows 处理的行数；

$fh 文件处理；

undef 在 Perl 中，null 值代表为处理的值；

\%attr 引用一个哈西属性值传递给方法；

注意，如果所有到数据库和语句对象的引用都被删除了 Perl 会自动摧毁它们。

处理对象属性如下：

<PRE> $h-E<gt>{attribute_name} </PRE> (type)

Type 指示属性的值的类型（如果其不是标量）：

\$ 引用一个标量: $h->{attr} or $a = ${$h->{attr}}

\@ 引用一个列表: $h->{attr}->[0] or @a = @{$h->{attr}}

\% 引用一个哈希: $h->{attr}->{a} or %a = %{$h->{attr}}

## 通用接口规则和误解

DBI 没有当前会话的概念。每个会话具有一个从连接方法返回的处理对象（如$dbh）并且那个处理对象用来调用数据库相关的方法。

大多数数据返回给 Perl 脚本是字符串形式的（null 为 undef）。这允许任意精度的数字数据可以被处理而不会丢失精确性。不过需要注意的是，如果字符串作为数字使用，perl 有可能不保留相同的精度。

日期和时间以相应引擎的本地格式的字符串形式返回。时区影响依赖于引擎/驱动器。

Perl 支持在 Perl 字符串中包含二进制数据，并且 DBI 可以传递并且接收二进制数据到引擎，并且不会做任何改变，直到 Driver 实施者决定如何处理这些二进制数据。

多个 SQL 语句可能不绑定在一个语句处理上，如一个单独的$sth。

该版本的 DBI 不支持非顺序读取的纪录。例如，记录只能以它们从数据库返回的顺序被推进并且一旦推进它们就被忘掉。

积极的更新和删除并不被 DBI 直接支持。见 CursorName 属性的描述。

各自的驱动器实施者免费提供他们认为有用的私有函数和处理属性。私有驱动器函数可以使用 DBI func 方法调用。私有驱动器属性的访问和标准的属性相同。

很多方法具有可选的\%attr 参数，其用来传递信息给驱动器实施的方法。除了那些具体文档化声明的\%attr 参数，只能用来传递驱动器相关的提示。通常可以忽略\%attr 参数，或者传递为 undef。

字符集：大多数理解字符集的数据库都有一个默认的全局数据库字符集保存存储的文本。当文本被推进的时候，它应该会自动转换为客户端的字符集。如果一个驱动器需要一个标记得到该行为，那么它应该完成该功能。而不应该要求应用程序完成该功能。

## 命名习惯和名字空间

DBI 包和以及它以下的全部包（(DBI::*)）被保留为 DBI 使用。以 DBD::开头的包名被保留由 DBI 数据库驱动使用。被 DBI 和 DBD 使用的全部环境变量以'DBI_'和'DBD_'开头。

属性名使用的大小写是比较重要的并且在 DBI 脚本的轻便性方面扮演重要的角色。属性名用来提示谁定义了那个名称的名字以及其值。

| 例子 | 意义 |
| --- | --- |
| UPPER_CASE | 标准，如 X/Open, SQL92 |
| MixedCase | DBI API，不使用下划线 |
| lower_case | 驱动器或引擎相关 |

驱动器开发者在定义私有属性时仅仅使用小写字母的属性值是非常重要的。私有属性名必须前缀驱动器名或者合适的缩写，如 ora_，ing_等。

具体驱动器的前缀列表如下：

```
ora_      DBD::Oracle

ing_      DBD::Ingres

odbc_     DBD::ODBC

syb_      DBD::Sybase

db2_      DBD::DB2

ix_       DBD::Informix

csv_      DBD::CSV

file_     DBD::TextFile

xbase_    DBD::XBase

solid_    DBD::Solid

proxy_    DBD::Proxy
```

## 提纲的使用

首先需要加载 DBI 模块：

```
use DBI;
```

同时增加 use strict;（推荐）然后连接到数据源：

```
$dbh = DBI->connect($dsn, $user, $password, { RaiseError => 1,
                    AutoCommit => 0 });
```

DBI 允许应用程序`prepare'语句稍后执行。一个准备的语句由一个语句处理对象识别，如 $sth。

一个 select 语句的典型调用顺序的方法如下：

```
prepare,
  execute, fetch, fetch, ...
  execute, fetch, fetch, ...
```

execute, fetch, fetch, ...

例如：

```
$sth = $dbh->prepare("select foo, bar from table where baz=?");
$sth->execute($baz);
while(@row = $sth->fetchrow_array) {
   print "@row\n";
}
```

非 select 语句的典型调用顺序的方法如下：

prepare,

execute,

execute,

execute.

例如：

```
$sth = $dbh->prepare("insert into table(foo,bar,baz) values (?,?,?)");
while(<CSV>) {
   chop;
   my ($foo,$bar,$baz) = split /,/;
   $sth->execute($foo,$bar,$baz);
}
```

如果 AutoCommit 为 off，则是用以下语句提交改变：

```
$dbh->commit;  # or call $dbh->rollback; to undo changes
```

最后，如果完成了与数据源的工作，应该关闭它：

```
$dbh->disconnect;
```

## 占位符和绑定值

一些驱动器支持占位符和绑定值。这些驱动器允许数据库语句中包含占位符，某些时候也成为参数标记符，指示这些值将在随后提供，不过在准备的语句执行前。例如，应用程序使用以下语句插入一行到 SALES 表中：

```
insert into sales (product_code, qty, price) values (?, ?, ?)
```

或者，从 product 表中选择 description：

```
select description from products where product_code = ?
```

?字符就是占位符。实际值和占位符的关系称为绑定，引用的值称为绑定值。

当占位符与 SQL LIKE 一起使用时，记住占位符会替代整个字符串。所以，需要使用``... LIKE ? ...''并在绑定值中包含全部的配置模式。

### Null 值

Undef 或未定义的值可以用来指示 null 值，但是，如果使用 null 值作为 select 语句的条件时必须注意。如下：

```
select description from products where product_code = ?
```

绑定一个 undef (NULL)到占位符不会选择任何 product_code 为 NULL 的行。参见具体的引擎的 SQL 手册查看原因。应该使用如下 where：

```
... where (product_code = ? or (? is null and product_code is null))
```

并且绑定相同的值到两个占位符。

### 性能

如果没有占位符，以上的插入语句就包含文本常量进行插入并且它需要重新准备和执行。使用占位符，插入语句只需要准备一次，每行的绑定值可以可以在每次执行方法被调用的执行传递。通过避免重新准备通常可以使性能增加 n 倍。如下：

```
my $sth = $dbh->prepare(q{
    insert into sales (product_code, qty, price) values (?, ?, ?)
  }) || die $dbh->errstr;
while (<>) {
    chop;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) || die $dbh->errstr;
```

```
}

$dbh->commit || die $dbh->errstr;
```

察看 execute 和 bind_param 得到更详细的信息。

使用 q{...}可以避免与 SQL 中可能使用的引号冲突。如果希望将变量替代为字符串则使用双引号如 qq{...}操作符。察看 perlop 得到更详细的信息。

察看 bind_column 得到将一个 Perl 变量与一个 select 语句的 output 列相关。

### SQL

大多数 DBI 驱动器要求应用程序使用 SQL 方言与数据库引擎交互（察看具体数据库的 sql 参考手册）。

DBI 本身不强制使用的特定语言，它是语言独立的。在 ODBC 术语中，DBI 在传递模式。唯一的要求是查询和其他语句必须以单个字符串形式传递给 prepare 方法的第一个参数。

## DBI 类

### DBI 类方法

#### 连接

```
$dbh = DBI->connect($data_source, $username, $password) || die $DBI::errstr;

$dbh = DBI->connect($data_source, $username, $password, \%attr) || die
$DBI::errstr;
```

建立一个数据库连接要求提供一个数据源，如果连接成功将返回一个数据库处理对象。使用 $dbh->disconnect 终止连接。

如果连接失败，它将返回 undef 并且设置$DBI::err 和$DBI::errstr（不会设置$!等）。通常应该测试连接的状态并且如果失败应该输出$DBI::errstr。

可以通过 DBI 同时建立到多个数据库的连接，只需要为每个连接创建一个数据库处理对象即可。

$data_source 应该以'dbi:driver_name:'开头，前缀会被截去，driver_name 部分用来声明驱动区（注意大小写敏感）。

为了方便起见，如果$data_source 字段未定义或者为空，DBI 会用环境变量 DBI_DSN 替代。如果 driver_name 部分为空，将会使用环境变量 DBI_DRIVER 替代。如果该环境变量没有定义，那么连接将会失败。

$data_source 值的例子：

```
dbi:DriverName:database_name
```

dbi:DriverName:database_name@hostname:port

dbi:DriverName:database=database_name;host=hostname;port=port

跟随 DriverName 的文本没有标准，每种驱动器可以使用各自的语法。唯一的要求是 DBI 要求提供的信息是一个字符串。可以查看具体的驱动器文档得到具体的语法描述。

如果定义了环境变量 DBI_AUTOPROXY，并且$data_source 不是'Proxy'，那么连接请求将自动被转换为：

dbi:Proxy:$ENV{DBI_AUTOPROXY};dsn=$data_source

并且传递给 DBD::Proxy 模块。DBI_AUTOPROXY 通常为``hostname=...;port=...''格式，察看 DBD 得到更详细的信息。

如果$username 或$password 没有定义而不是空，那么 DBI 会使用 DBI_USER 和 DBI_PASS 环境变量的值代替。出于安全考虑，不推荐使用这些环境变量，仅仅用于简化测试。

如果驱动器没有安装，DBI->connect 将自动安装驱动。驱动器安装通常返回一个有效的驱动器处理器或者失败返回错误消息包括'install_driver'字符串和底层问题。所以 DBI->connect 会在驱动器安装失败时死去，并且仅仅在连接失败上返回 undef，此时，$DBI::errstr 包含错误。

$data_source 参数、$username 和$password 参数随后会传递给驱动器处理。DBI 没有为这些字段定义任何的诊断。驱动器可以以任何方式推断这些值并且提供任何恰当的默认值供引擎处理（如，ORACLE_SID 和 TWO_TASK 等）。

每个连接的 AutoCommit 和 PrintError 属性默认为 on。

\%attr 参数可以用来更改 PrintError, RaiseError, AutoCommit 和其他属性的默认设置。如下：

```
$dbh = DBI->connect($data_source, $user, $pass, {
    PrintError => 0,
    AutoCommit => 0
});
```

轻便的应用程序不应该假设一个单独的驱动器将会同时支持多个会话。

只要可能，每个会话（$dbh）独立于其他会话中的事务。当需要跨事务保持游标打开时这是非常有用的，为长时间打开的游标使用一个会话，另一个则用于较短的更新操作。

为了与旧版本的 DBI 脚本兼容，驱动器名可以声明为 connect 的第四个参数（代替\%attr）：

```
$dbh = DBI->connect($data_source, $user, $pass, $driver);
```

在这种旧格式的连接中，$data_source 不应该以'dbi:driver_name:'开头，即使以它开头，嵌入的 driver_name 会被截去。$dbh->{AutoCommit}属性为 undefined。$dbh->{PrintError}为 off。并且如果 DBI_DSN 没有定义，将会检查 DBI_DBNAME。这种旧格式的连接在将来的版本中会被抛弃。

### available_drivers

```
@ary = DBI->available_drivers;

@ary = DBI->available_drivers($quiet);
```

通过@INC 中的目录搜索所有的 DBD::*模块返回一个所有可用驱动器的列表。默认情况下，如果某些驱动器被先前目录中其他相同名字的驱动器覆盖，将会返回一个警告，可以通过设置$quiet 为 true 屏蔽。

### data_sources

```
@ary = DBI->data_sources($driver);

@ary = DBI->data_sources($driver, \%attr);
```

返回通过命名的驱动器可用的所有数据源（数据库）的列表。如果驱动器还没有加载，将会被加载。如果$driver 为空或者 undef，将使用 DBI_DRIVER 环境变量的值。

数据源将会以适合于传递给 connect 方法的形式返回。包括``dbi:$driver:''前缀。

注意：许多驱动器无法知道对它可用的数据源，因此通常返回空或者不完整的列表。

### Trace

```
DBI->trace($trace_level)

DBI->trace($trace_level, $trace_file)
```

可以使用 DBI 类的方法为所有处理器启用 DBI 跟踪信息，要起用一个特定的处理器的跟踪，可以使用类似于$h->trace 的方法。

使用$trace_level 2 可以看到详细的调用信息包括参数和返回值。跟踪信息是详细的并且通常非常有用。大多数的跟踪输出通过使用 neat 函数格式化因此可能被编辑或截断过。

使用$trace_level 0 禁用跟踪。

原始的跟踪输出被写入 STDERR，如果声明了$trace_filename，那么文件将保持在 append 模式并且所有的跟踪输出都将被冲定向到该文件。没有$trace_filename 的跟踪不会改变跟踪输出传送的地方。

查看$h->trace()和 DEBUGGING 得到关于 DBI_TRACE 环境变量的信息。

## DBI 工具函数

### Neat

```
$str = DBI::neat($value, $maxlen);
```

返回一个包含给定值的简单的表达。

字符串会被引用，内部引号除外。被认为是数字的值不会被引用，未定义的值（NULL）将显示为 undef（无引号）。不可打印字符会被.代替。

对于结构字符串大于$maxlen 的，将被截去为$maxlen-4 和…。如果$maxlen 为 0 或者 undef，那么将使用$DBI::neat_maxlen，默认为 400.

该函数设计用来格式化值供人类使用。其在内部被 DBI 用来 trace 输出。对于给数据库使用的值，通常不应该使用该函数。

### neat_list

```
$str = DBI::neat_list(\@listref, $maxlen, $field_sep);
```

在列表的每个元素上调用 DBI::neat 返回一个与$field_sep 连接的包含结果的字符串，$field_sep 默认为","。

### looks_like_number

```
@bool = DBI::looks_like_number(@array);
```

对于每个看起来像数字的元素，返回 true。对于每个看起来不像数字的元素，返回 false，对于 null 和未定义的元素，返回 undef。

## DBI 动态属性

这些属性通常和最后一个使用的处理器相关。

这里的属性相当于一个方法调用，然后参考方法调用的所有相关文档。

警告：提供这些属性是为了方便，但是他们是有限制的。特别是它们与最后一个处理器相关，因此需要在调用方法后“立刻“使用它们。

如果有任何疑问，使用相应的方法调用。

$DBI::err

等价于$h->err。

$DBI::errstr

等价于$h->errstr。

$DBI::state

等价于$h->state。

$DBI::rows

等价于$h->rows。

## 所有处理器通用的方法

### Err

```
$rv = $h->err;
```

从最后一个驱动器函数调用返回本地数据库引擎错误代码。

### Errstr

```
$str = $h->errstr;
```

从最后一个驱动器函数调用返回本地数据库引擎错误消息。

### State

```
$str = $h->state;
```

以标准的 SQLSTATE 5 个字符的格式返回错误代码。成功的代码返回 00000，并转换为 0。如果驱动器不支持 SQLSTATE，那么对于所有的错误将返回 S1000。

### Trace

```
$h->trace($trace_level);

$h->trace($trace_level, $trace_filename);
```

DBI 跟踪信息可以通过使用跟踪方法在具体的处理器（以及所有其子处理器）上启用。使用 $trace_level 2 可以得到详细的跟踪信息，包括参数和返回值。

用$trace_level 0 禁用跟踪。

原始的跟踪输出被写入 STDERR，如果声明了$trace_filename，那么文件将保持在 append 模式并且所有的跟踪输出都将被冲定向到该文件。没有$trace_filename 的跟踪不会改变跟踪输出传送的地方。

查看$h->trace()和 DEBUGGING 得到关于 DBI_TRACE 环境变量的信息。大多数的跟踪输出通过使用 neat 函数格式化因此可能被编辑或截断过。

### trace_msg

```
$h->trace_msg($message_text);
```

如果跟踪在$h 或 DBI 上启用了，那么写$message_text 到跟踪文件。也可以调用 DBI->trace_msg($msg)。

### Func

```
$h->func(@func_arguments, $func_name);
```

该方法可以用来调用私有非标准的和不可拔插的由驱动器实施的方法。注意，函数名为最后一个参数。

该方法不直接与调用存储过程相关。调用存储过程当前不是由 DBI 实施的，一些驱动器，如 DBD::Oracle，支持不可拔插的方法。查看相关的驱动器文档得到详细的信息。

## 所有处理器通用的属性

这些属性对所有类型的 DBI 处理器都通用。

其中一些属性使用子处理器继承的。这就是说，在一个新创建的语句处理器中的继承属性的值和在父数据库处理器中的值是一样的。改变新语句处理器中的属性不会影响父数据库处理器，并且到数据库处理器的更改不会影响已存在的语句处理器，仅仅会改变之后的语句处理器。

尝试得到或设置一个未知属性的值是致命的，除了私有驱动器相关的属性外（其名称由小写字母开头）。

例如：

```
$h->{AttributeName} = ...;     # set/write

... = $h->{AttributeName};     # get/read
```

### Warn (boolean, inherited)

对某些坏的习惯启用警告，默认为启用。某些仿效层，禁用警告。

### Active (boolean, read-only)

为 true 如果处理其对象为'active'，这在应用程序中很少使用。active 精确的意义此时有些争议。对于数据库处理器，通常表示处理连接到数据库（$dbh->disconnect 应该设置 Active 为 off）。对于语句处理器通常表示处理器是一个 select，并且有更多的数据需要推进（$dbh->finish 或者推进全部数据应该设置 Active 为 false）。

### Kids (integer, read-only)

对于驱动器处理器，Kids 是指由那个驱动器处理器创建的当前存在的数据库处理器的数量。对于数据库处理器，Kids 是指由那个数据库处理器创建的当前存在的语句处理器的数量。

### ActiveKids (integer, read-only)

同上，不过仅仅计算为 Active 的数量。

### CachedKids (hash ref)

对于数据库处理器，返回一个由 prepare_cached 方法创建到语句处理器缓存的引用。对于驱动器处理器，返回一个由 connect_cached 方法创建但还未实施的到语句处理器的缓存的引用。

### CompatMode (boolean, inherited)

被仿效层（如 Oraperl）用来为处理器在底层驱动器（如 DBD::Oracle）上启用兼容行为。并非由应用程序代码正常设置。

### InactiveDestroy (boolean)

该属性用来禁用 DESTROY 一个处理器(它会正常的关闭一个已准备的语句或断开数据库连接)的数据库相关的影响。它主要是为 UNIX 应用程序设计的用来'fork'子进程。父进程或者子进程应该在它们的处理器设置 InactiveDestroy，但不能是全部。对于数据库处理器，该属性不会禁用到 disconnect 方法的显示调用。仅仅是来自 DESTROY 的隐示调用。

### PrintError (boolean, inherited)

该属性用来强制错误除了正常的返回错误码以外产生警告。当设置为 on 的时候，任何导致错误产生的方法都会使 DBI 产生一个警告（``$class $method failed $DBI::errstr''），$class 是驱动器类名，$method 是造成失败的方法名。如：

```
DBD::Oracle::db prepare failed: ... error text here ...
```

默认情况下，DBI->connect 设置 PrintError 为 on。

如果希望，警告可以通过$SIG{__WARN__}处理器或者 CGI::ErrorWrap 模块进行追踪和处理。

### RaiseError (boolean, inherited)

该属性用来强制错误抛出异常而不是简单地返回错误码。默认为 off，如果设置为 on，任何导致发生的错误的方法会使 DBI 有效的产生一个 die(``$class $method failed $DBI::errstr'') $class 是驱动器类名，$method 是造成失败的方法名。如：

```
DBD::Oracle::db prepare failed: ... error text here ...
```

如果 PrintError 为 on，那么 PrintError 将在 RaiseError 前完成。除非定义了__DIE__ handler，在这种情况下，PrintError 将会跳过，因为 die 会打印消息。

如果希望临时关闭 RaiseError，推荐使用以下方法：

```
{
  local $h->{RaiseError} = 0 if $h->{RaiseError};
  ...
}
```

原始值会被 Perl 自动并且可靠的转储，不管其是怎么退出的。... if $h-> {RaiseError}是可选的，不过在通常情况下能使代码更快。相同的逻辑同样应用于其他属性。

在 5.004_04（包括）以前，这种形式不能运行，为兼容性，使用 eval { ... }代替。

### ChopBlanks (boolean, inherited)

该属性用来控制是否从 CHAR 字段截去尾随的空格，其他字段不受影响（即使包括尾随的空格）。

默认为 false。应用程序应该根据需要设置该值

### LongReadLen (unsigned integer, inherited)

该属性可以用来控制在自动推进驱动器从数据库读取的行时'long'/'blob', 'memo'字段的最大长度。

为 0 意味着不会自动推进任何的 long 数据（在 LongReadLen 为 0 时应该返回 undef）。

默认为 0，但是不同的驱动器可能有不能的默认值。大多数应用程序在推进时通常会设置为比最大的 long 字段稍微大一点的值。

在 prepare()后改变一个一个语句处理器的 LongReadLen 值通常没有影响，因此，通常在$dbh 上调用 prepare 前设置 LongReadLen。

LongReadLen 属性仅仅与推进和读取有关，在插入和更新时并不调用。

查看 LongTruncOk 获得截取的行为。

### LongTruncOk (boolean, inherited)

该属性用来控制一个截断的 long 字段的推进的影响。

默认情况下 LongTruncOk 为 false，并且推进一个截断的 long 值会导致推进失败。（应用程序应该在推进一个循环后检查错误，如除 0，long 字段截断导致的推进非预期终止）

如果在 LongTruncOk 为 false 时由于 long 截断导致推进失败，许多驱动器允许继续推进更多的行。

### private_*

DBI 提供了一种方法存储额外的信息在 DBI 处理器中，称为'private'属性。DBI 允许用户存储和提取任何名称以'private_'开头的属性。强烈推荐仅仅使用一个 private 属性（使用哈希引用替代）并且给它一个较长和明确的名称（包括与该属性有关的模块或应用程序）。

# DBI 数据库处理器对象

## 数据库处理器方法

### selectrow_array

```
@row_ary = $dbh->selectrow_array($statement);

@row_ary = $dbh->selectrow_array($statement, \%attr);

@row_ary = $dbh->selectrow_array($statement, \%attr, @bind_values);
```

该工具方法结合 prepare, execute 和 fetchrow_array 组成一个单独的调用。如果在一个列表上下文中调用，将从语句返回第一行数据。如果在标量环境中调用，将从语句返回第一条数据的第一个字段。$statement 可以是先前 prepared 的语句处理器，在这种情况下 prepare 将会跳过。

如果任何语句失败，并且没有设置 RaiseError，selectrow_array 将会返回一个空的列表（在标量环境中为 undef）。

### selectall_arrayref

```
$ary_ref = $dbh->selectall_arrayref($statement);

$ary_ref = $dbh->selectall_arrayref($statement, \%attr);

$ary_ref = $dbh->selectall_arrayref($statement, \%attr, @bind_values);
```

该工具方法结合 prepare, execute 和 fetchall_arrayref 组成一个单独的调用。$statement 可以是先前 prepared 的语句处理器，在这种情况下 prepare 将会跳过。

如果任何语句失败，并且没有设置 RaiseError，selectall_arrayref 将返回 undef。

### Prepare

```
$sth = $dbh->prepare($statement)          || die $dbh->errstr;

$sth = $dbh->prepare($statement, \%attr)  || die $dbh->errstr;
```

为以后数据库引擎执行准备一个单独的语句并返回一个引用给语句处理器对象。

返回的语句处理器可以用来得到语句的属性并调用 execute 方法。见 Statement Handle Methods。

注意：准备永远不会执行一个语句，即使它不是 select 语句，它仅仅是准备执行。（话随这么说，某些驱动器，如 Oracle 在准备 ddl 的时候会执行它，通常这不是问题）

没有准备语句概念的引擎的驱动器通常仅仅存储语句到返回的处理器中并在$sth->execute 调用时处理它。这些驱动器无法给定关于语句的有用信息，如$sth->{NUM_OF_FIELDS}，除非调用了$sth->execute，可拔插应用程序可能需要考虑这一点。

通常，DBI 驱动器并不解析语句的内容（除了计算占位符以外）。语句被直接传递到数据库引擎（pass-thru 模式）。这么做的好处是，可以访问所有将被引擎使用的功能；坏处是，如果使用一个单独的引擎并且希望编写可以在不同引擎之间移植的应用程序就需要额外的考虑。

一些命令行 SQL 工具使用语句终止符，如分号指示语句的结束，这些终止符不能在 DBI 中使用。

### prepare_cached

```
$sth = $dbh->prepare_cached($statement)          || die $dbh->errstr;

$sth = $dbh->prepare_cached($statement, \%attr)  || die $dbh->errstr;
```

和 prepare 一样，只是返回的语句处理器将以与$dbh 哈希相关的方式存储。如果另一个使用相同参数值的调用 prepare_cached，那么相应的缓存$sth 将被返回。（并且不会和数据库服务器联系。）

在某些应用程序中缓存是有用的不过它也会造成问题，因此需要小心使用。当前，如果返回的缓存$sth 是 active 的（select 语句中还有数据需要推进），将会产生一个警告。

缓存可以通过 CachedKids 属性访问和清除。

## do

```
$rc  = $dbh->do($statement)           || die $dbh->errstr;

$rc  = $dbh->do($statement, \%attr)   || die $dbh->errstr;

$rv  = $dbh->do($statement, \%attr, @bind_values) || ...
```

准备和执行一个单独的语句。返回受影响的行数（-1 为未知或者不可用）或者 undef。

该语句对于非 select 语句（不重复执行）最有用，因为它不需要提前准备。对于 select 语句，不应该使用它。

默认 do 方法的逻辑如下：

```
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; # always return true if no error
}
```

例如：

```
my $rows_deleted = $dbh->do(q{
    delete from table
    where status = ?
}, undef, 'DONE') || die $dbh->errstr;
```

在 do 方法中使用占位符和@bind_values 是有用的，因为它避免了在$statement 中正确的引用任何变量。

q{...}风格的引用避免了可能在 SQL 语句中使用引号的冲突。使用双引号如果希望变量引用为字符串。

### Commit

```
$rc  = $dbh->commit     || die $dbh->errstr;
```

如果数据库支持事务，那么提交最近的所有改变。

如果数据库支持事务并且 AutoCommit 为 on，那么执行``commit ineffective with AutoCommit''将会产生一个警告。

见 Transactions。

### Rollback

```
$rc  = $dbh->rollback   || die $dbh->errstr;
```

如果数据库支持事务，那么回滚最近的所有改变。

如果数据库支持事务并且 AutoCommit 为 on，那么执行`` rollback ineffective with AutoCommit''将会产生一个警告。

见 Transactions。

### disconnect

```
$rc = $dbh->disconnect  || warn $dbh->errstr;
```

从数据库处理器断开数据库。通常仅在退出程序时调用。该处理器在数据库断开后几乎没有用处。

断开方法的事务行为是不确定的。一些数据库系统，如 Oracle 和 Ingres 在断开时会自动提交所有未知的改变，而某些数据库，如 Informix 则回滚。因此应用程序应该在调用 disconnect 前显示调用 commit 和 rollback。

如果不再有引用指向处理器，并且数据库仍然是连接的，那么数据库会自动断开（通过 DESTROY 方法），每个驱动器的 DESTROY 方法应该被显示调用回滚任何未提交的改变。这是确保未完成的事务不会简单的提交的重要行为，因为在 Perl 会在每个对象退出前在其上调用 DESTROY。同时不要在全局摧毁期间依赖于对象摧毁的顺序，它是不确定的。

如果在断开一个数据库是还有活动的语句处理器，那么将会得到一个警告。语句处理器应该在断开或调用 finish 方法前被清除。

### ping

```
$rc = $dbh->ping;
```

以一种可理解有效的方式尝试决定，数据库服务器是否仍在运行，连接是否仍在工作。

当前默认的实现总是返回 true 而不做实际的事情，各自的驱动器应该以最适合的方法实施。

很少的应用程序需要使用该方法。查看专门的 Apache::DBI 模块得到一个使用例子。

### table_info *NEW*

警告：该方法仅用来测试，可能会被取消或更改。

```
$sth = $dbh->table_info;
```

返回一个活动的语句处理器，可以用来提取数据库中的表或视图的信息。处理器至少包含以下字段，其他字段也可能出现：

TABLE_QUALIFIER:表 的标识符，如果无法应用到数据源则为 NULL (undef)，如果无法应用到表，则为空。

TABLE_OWNER:表拥有者标识符。如果无法应用到数据源则为 NULL (undef)，如果无法应用到表，则为空。

TABLE_NAME：表名。

TABLE_TYPE:表的类型，``TABLE'', ``VIEW'', ``SYSTEM TABLE'', ``GLOBAL TEMPORARY'', ``LOCAL TEMPORARY'', ``ALIAS'', ``SYNONYM''或数据源相关的类型标识符。

REMARKS:表的描述符，可能为 NULL (undef)。

table_info 可能不会返回任何表的信息，应用程序可以使用任何有效的表而不管它是否是 table_info 返回的。

### tables *NEW*

警告：该方法仅用来测试，可能会被取消或更改。

```
@names = $dbh->tables;
```

返回一个表名和视图名的列表。该列表应该包括所有可以在一个 select 语句中无须更多标识使用的表。通常意味着用户以及用户可以访问的所有表。

table 可能不会返回任何表的信息，应用程序可以使用任何有效的表而不管它是否是 table 返回的。

### type_info_all *NEW*

警告：该方法仅用来测试，可能会被取消或更改。

```
$type_info_all = $dbh->type_info_all;
```

返回一个到数组的引用，其中包括数据库和驱动器支持的各种数据类型变种。

第一个条目是到一个 Name => Index 哈希对的引用。随后的条目引用一个数组，每个条目为一个支持的数据类型变种。开头的哈希定义了在数组列表中的字段的名称和顺序。例如：

```
$type_info_all = [
```

```
    {   TYPE_NAME         => 0,
        DATA_TYPE         => 1,
        PRECISION         => 2,
        LITERAL_PREFIX    => 3,
        LITERAL_SUFFIX    => 4,
        CREATE_PARAMS     => 5,
        NULLABLE          => 6,
        CASE_SENSITIVE    => 7,
        SEARCHABLE        => 8,
        UNSIGNED_ATTRIBUTE=> 9,
        MONEY             => 10,
        AUTO_INCREMENT    => 11,
        LOCAL_TYPE_NAME   => 12,
        MINIMUM_SCALE     => 13,
        MAXIMUM_SCALE     => 14,
    },
    [ 'VARCHAR', SQL_VARCHAR,
        undef, "'","'", undef,0, 1,1,0,0,0,undef,1,255
    ],
    [ 'INTEGER', SQL_INTEGER,
       undef,  "", "", undef,0,  0,1,0,0,0,undef,0,   0
    ],
];
```

注意：在 DATA_TYPE 字段可能会有多行具有相同的值。

该方法通常不直接使用，type_info 方法提供了更加有用的接口。

字段的名称在 type_info 中描述。

### type_info *NEW*

警告：该方法仅用来测试，可能会被取消或更改。

```
@type_info = $dbh->type_info($data_type);
```

返回一个包含一个/多个$data_type 变种的哈希引用列表。

如果$data_type 是 SQL_ALL_TYPES，那么列表将包含所有驱动器和数据库支持的数据类型表中的哈希。

以下条目应该存在：

TYPE_NAME (string)：在 CREATE TABLE 中使用的数据类型名；

DATA_TYPE (integer)：SQL 数据类型号；

PRECISION (integer)：数据类型的最大精度，如果数据类型无法应用将返回 NULL (undef)。

LITERAL_PREFIX (string)：用来前缀一个文本常量的字符。通常字符为'，传给 16 进制的二进制值为 0x。如果数据类型无法应用将返回 NULL (undef)。

LITERAL_SUFFIX (string): 用来前缀一个文本常量的字符。通常字符为'。如果数据类型无法应用将返回 NULL (undef)。

CREATE_PARAMS (string)：数据类型定义的参数。例如，DECIMAL 的 CREATE_PARAMS 为``precision,scale''。对于 VARCHAR，则为``max length''。如果数据类型无法应用将返回 NULL (undef)。

NULLABLE (integer): 指示数据类型是否接受空值。0 = no, 1 = yes, 2 = unknown.

CASE_SENSITIVE (boolean)：指示数据类型是否大小写敏感。

SEARCHABLE (integer)：指示数据类型是否可以在 WHERE 中使用：

0 - 不能使用；

1 - 只能在 like 中使用；

2 - 所有的比较操作符，除了 like；

3 - 可以使用；

UNSIGNED_ATTRIBUTE (boolean)：指示数据类型是否无符号。如果数据类型无法应用将返回 NULL (undef)。

MONEY (boolean): 指示数据类型是否为 money 类型。如果数据类型无法应用将返回 NULL (undef)。

AUTO_INCREMENT (boolean): 指示数据类型是否自动增长。如果数据类型无法应用将返回 NULL (undef)。

LOCAL_TYPE_NAME (string)：TYPE_NAME 的本地版本，用于用户的诊断。

MINIMUM_SCALE (integer)：数据类型的最小长度。如果数据类型长度固定，那么 MAXIMUM_SCALE 也为该值。

MAXIMUM_SCALE (integer)：数据类型的最大长度。如果数据类型长度固定，那么 MINIMUM_SCALE 也为该值。

### Quote

```
$sql = $dbh->quote($value);

$sql = $dbh->quote($value, $data_type);
```

引用一个字符串常量在 SQL 语句中通过转义特定的字符作为文本值使用，并且增加外引用标记所需的类型：

```
$sql = sprintf "select foo from bar where baz = %s",

$dbh->quote("Don't\n");
```

对于大多数数据类型，引号将返回'Don''t'。

一个未定义的$value 值将返回字符串 NULL（没有引用标记）。

如果提供了$data_type，那么它将用来通过使用 type_info 返回的信息确定需要的引用行为。在特定的情况下，标准的数字类型将返回$value 而无需调用 type_info。

引号并非对所有的输入进行转义，如二进制。在占位符和绑定值中无须引用值。

## 数据库处理器属性

这部分描述与数据库处理器相关的属性。

改变这些数据库处理器的属性不会影响其它数据库处理器以及之后的数据库处理器的属性。

尝试得到或设置一个未知属性的值是致命的，除了私有驱动器相关的属性外（其名称由小写字母开头）。

例如：

```
$h->{AttributeName} = ...;    # set/write
```

```
... = $h->{AttributeName};    # get/read
```

### AutoCommit (boolean)

如果为 true，那么数据库改变是不可回滚的。如果为 false，那么必须使用 commit 或 rollback 方法提交或回滚。

驱动器应该经常默认为 AutoCommit（由于 ODBC 和 JDBC 的习惯，强加在 DBI 上的选择）。

尝试设置 AutoCommit 为一个不支持的值是致命的。这是 DBI 的一个重要特征。需要完全事务行为的应用程序可以设置$dbh->{AutoCommit}=0(通过 connect)而无需检查当前分配的值。

根据这种情况，可以将数据库分为三种类型：

- 完全不支持事务的数据库。
- 事务通常是活动的的数据库。
- 事务必须被显示开始('BEGIN WORK')的数据库。

*完全不支持事务的数据库

对于这些数据库尝试设置 AutoCommit 为 off 是致命的。提交和回滚会导致发出警告。

*事务通常是活动的的数据库

这些是支持 ANSI 标准的主流商业数据库的事务的行为。

如果 AutoCommit 为 off，那么到这些数据库的改变将不会有任何的最终影响除非调用了 commit。如果调用了 rollback，任何最后一次 commit 以来的改变都会被回滚。

如果 AutoCommit 为 on,那么该影响同第一种数据库。换句话说，在 AutoCommit 为 off 时显示调用 commit 或 rollback 都是没有意义的，因为所有的改变已经被提交。

在大多数驱动器下，将 AutoCommit 从 off 改变为 on 将执行一个 commit。

将 AutoCommit 从 on 改变为 off 没有立刻的影响。

对于不支持一种具体的自动提交模式的数据库，驱动器必须在每个语句成功或者失败以后调用相应的 COMMIT 或者 ROLLBACK。报告给应用程序的相关信息是相应的语句，除非语句执行成功了，但是 commit 或 rollback 失败了。

*事务必须被显示开始('BEGIN WORK')的数据库

对于这些数据库，目的是让它们的行为和上述数据库一样。为了完成该功能，DBI 驱动器会自动在 AutoCommit 为 off 时自动开始一个事务，并且在提交和回滚后自动开始另一个事务。

在这种情况下，应用程序不需要将这些数据库作为一种特例。

### Name (string)

保留数据库的'name'，通常为用来连接数据库的``dbi:DriverName:...''，但是去掉了开头的``dbi:DriverName:''。

### RowCacheSize (integer) *NEW*

一个用来指示驱动器应用程序希望本地缓存用以以后 select 的行缓冲大小。如果行缓存没有实施，那么 RowCacheSize 的设置会被忽略，并且返回 undef。

具有特定意义的 RowCacheSize 的值如下：

0 - 对于每个 select 自动决定缓存大小；

1 - 禁用本地行缓存；

>1- 缓存这么多行；

<0-对于每个 select 语句，缓存尽可能多的行；

注意：较大的缓存尺寸要求更多的内存(cached rows * maximum size of row)并且较大的缓存可能导致第一次和缓存重新填充时推进较长的延迟。

查看 RowsInCache 语句处理属性得到更多的信息。

## DBI 语句处理器对象

### 语句处理器方法

### bind_param

```
$rc = $sth->bind_param($p_num, $bind_value)  || die $sth->errstr;

$rv = $sth->bind_param($p_num, $bind_value, \%attr)     || ...

$rv = $sth->bind_param($p_num, $bind_value, $bind_type) || ...
```

bind_param 方法可以用来绑定值到先前嵌入在准备的语句的占位符中，占位符通过?指示。如下：

```
$dbh->{RaiseError} = 1;

$sth = $dbh->prepare("select name, age from people where name like ?");

$sth->bind_param(1, "John%");

$sth->execute;

DBI::dump_results($sth);
```

注意：?没有包含在引号中，即使它是字符串。一些驱动器还支持:1，:2，:name 等风格的占位符，不过它们是不可移植的。

一些驱动器不支持占位符。

对于大多数的驱动器，占位符不能是一个语句的任何防止数据库服务器验证语句并创建查询执行计划的元素。例如：

```
"select name, age from ?"         # wrong

"select name, ?   from people"    # wrong
```

同时，占位符只能代表单独的标量值，以下语句是不对的：

```
"select name, age from people where name in (?)"    # wrong
```

\%attr 参数可以用来声明占位符的数据类型。通常驱动器只想知道一个占位符是数字还是字符串。

```
$sth->bind_param(1, $value, { TYPE => SQL_INTEGER });
```

通常，数据类型可以直接传递以代替 attr 哈希引用，如下：

```
$sth->bind_param(1, $value, SQL_INTEGER);
```

在第一次 bind_param 调用后，TYPE 无法改变。TYPE 指示标准的非驱动器相关的类型。为了支持驱动器相关的类型，驱动器可能支持驱动器相关的属性，如：{ ora_type => 97 }。

Perl 只有字符串和数字标量数据类型。所有非数字的数据库数据类型都被绑定为字符串类型，并且必须被格式化为数据库能够理解的格式。

未定义的值或者 undef 通常用来指示 null 值。

bind_param_inout

```
$rc = $sth->bind_param_inout($p_num, \$bind_value, $max_len)  || die $sth->errstr;

$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, \%attr)     || ...

$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, $bind_type) || ...
```

该方法类似于 bind_param，不过它允许值从语句中传出，该语句通常用来调用一个存储过程，$bind_value 必须以传递实参的引用而非副本使用。

注意：与 bind_param 不同，$bind_value 在 bind_param_inout 调用时并不被读取。相反，其中的值在 execute 被调用时执行。

$max_len 参数声明需要分配给$bind_value 的新值的最少内存。如果值太大，那么执行将会失败。如果无法确定返回值的大小，那么定义为一个比可能的值更大的值。

当前版本中，只有少量的驱动器支持该方法，已知的是 DBD::Oracle，DBD::ODBC 在以后的版本中可能会支持。因此该方法不应该在数据库独立的应用中使用。

### Execute

```
$rv = $sth->execute                || die $sth->errstr;

$rv = $sth->execute(@bind_values)  || die $sth->errstr;
```

执行准备的语句。如果发生错误，将会返回 undef。一次成功地执行通常返回 true，无论影响的行数是多少。通常应该检查返回的执行状态（包括其他大部分 DBI 方法）。

对于非 select 语句，Execute 返回影响的行数。如果没有行受到影响，那么 execute 将返回``0E0''，Perl 将会把其作为 0 但是为 true。如果受影响的行数未知，那么将返回-1。

对于 select 语句，execute 仅仅在引擎内部开始查询。在调用 execute 后，使用其中一个推进方法提取数据。Execute 方法不能返回查询将返回的行数（因为大多数引擎不能提前告知），其仅仅返回 true。

如果给定了任何的参数，那么 execute 将会在执行前为每个参数有效的调用 bind_param。以这种方式绑定的值通常被作为 SQL_VARCHAR 类型处理，除非驱动器能够确定正确的类型（几乎很少）或者 bind_param（bind_param_inout）已经声明了类型。

### fetchrow_arrayref

```
$ary_ref = $sth->fetchrow_arrayref;

$ary_ref = $sth->fetch;    # alias
```

提取下一行数据并且返回一个包含字段值的引用给数组。Null 字段以 undef 返回。这是提取数据最快的方法，特别是和$sth->bind_columns 一起使用。

如果没有更多的行可以提取或者发生了错误 fetchrow_arrayref 将会返回 undef，应该使用 $sth->err 或 RaiseError 检查。

注意：当前对于每一次的提取，都将返回相同的数组引用，因此不要存储并在下一次提取后使用数组。

### fetchrow_array

```
@ary = $sth->fetchrow_array;
```

fetchrow_arrayref 的另一种选择。提取下一行数组并将字段保存在数组中返回。Null 字段以 undef 返回。如果没有更多的行可以提取或者发生了错误 fetchrow_arrayref 将会返回 undef，应该使用$sth->err 或 RaiseError 检查。

### fetchrow_hashref

```
$hash_ref = $sth->fetchrow_hashref;

$hash_ref = $sth->fetchrow_hashref($name);
```

fetchrow_arrayref 的另一种选择.提取下一行数据并返回一个包含字段名和字段值对的哈希引用。Null 值返回为 undef。

如果没有更多的行可以提取或者发生了错误 fetchrow_hashref 将会返回 undef，应该使用 $sth->err 或 RaiseError 检查。

可选的$name 参数用于声明用来得到字段名的语句处理器属性的名称。默认为'NAME'。

哈希的键和$sth->{$name}返回的值相同。如果有多个字段具有相同的名称，那么哈希返回的那些字段中只有一个条目。

注意：当前版本的 fetchrow_hashref 在数据库之间是不可移植的，因为不同的数据库对大小写的处理不同，有些返回大写，有些返回小些，另外有一些则是按照 create table 时的输入。这个问题在将来版本的 DBI 中会被解决。

因为 fetchrow_hashref 和 Perl 需要执行额外的工作，因此没有 fetchrow_arrayref 和 fetchrow_array 高效，因此在性能要求很要的系统中不推荐使用。当前对于每行都返回一个新的哈希引用，该行为在以后可能会被更改，因此不要依赖它。

### 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  );
```

fetchall_arrayref 方法用来从已经准备和执行的语句处理器中提取所有数据。其返回一个数组，其中包含了每一行的一个引用。如果没有行返回，fetchall_arrayref 将返回一个到空数组的引用。如果发生了错误，fetchall_arrayref 返回已经提取的数据（应该随后用$sth->err 或 RaiseError 检查）。

当传递一个数组引用时，fetchall_arrayref 使用 fetchrow_arrayref 提取每行作为一个数组饮用。如果参数数组不为空，那么就像片一样通过索引号选择单独的列。

如果没有参数，fetchall_arrayref 就像传递了空的数组引用一样。

当传递一个哈希引用时，fetchall_arrayref 使用 fetchrow_hashref 提取每一行作为一个哈希引用。如果参数哈希不为空，那么它将作为作为一个片通过名称选择单独的列。名称应该小写不管$sth->{NAME}返回的大小写如何。哈希的值应该设置为 1。

例如，仅仅提取第一行数据的第一个列可以使用如下：

```
$tbl_ary_ref = $sth->fetchall_arrayref([0]);
```

提取每行的第二到最后一列，可以使用如下：

```
$tbl_ary_ref = $sth->fetchall_arrayref([-2,-1]);
```

仅提取称为每行称为 foo 和 bar 的字段：

```
$tbl_ary_ref = $sth->fetchall_arrayref({ foo=>1, bar=>1 });
```

前面两个返回一个指向数组引用的数组，最后一个返回一个指向哈希引用的数组。

### finish

```
$rc  = $sth->finish;
```

指示在该语句处理器被重新执行或摧毁以前不再有数据将被提取。这个方法很少使用，不过有时候为了允许服务器释放当前保持的资源，可以使用。

当从一个 select 语句提取完所有的数据后，驱动器会自动调用 finish 方法，因此无需显示调用。

考虑如下一个查询：

```
SELECT foo FROM table WHERE bar=? ORDER BY foo
```

并且仅仅希望选择 foo 值最小的一行，那么在执行时，服务器会使用临时排序空间存储排序的行。如果在执行并选择了一行后，处理器在短时内不会再执行或者完全不会在执行，那么可以使用该函数告诉服务器缓冲空间可以释放。

调用 finish 函数会重置语句的 Active 属性，并且如果他们不能在被访问，一些处理器属性，如 NAME 和 TYPE 将不再可访问。

Finish 方法不会影响会话的事务状态。如果将要摧毁或或者重新执行一个语句处理器，则没有必要调用 finish，见 disconnect 和 Active 属性。

### Rows

```
$rv = $sth->rows;
```

返回最后一个数据库更改命令影响的行数，未知的话为-1。

通常只能在 do 或非 select execute 或推进一个 select 语句的全部行后才能依赖行数。

对于 select 语句，除非全部提取了行，否则无法知道一个 select 将返回的行数。有些驱动器可能知道当前提取了多少行，另一些可能会返回-1。

### bind_col

```
$rc = $sth->bind_col($column_number, \$var_to_bind);
```

```
$rc = $sth->bind_col($column_number, \$var_to_bind, \%attr);
```

绑定 select 语句的输出列到 perl 变量。不需要这么做，不过在有些应用中可能有用。

无论何时行从数据库提取，相应的 Perl 变量都会自动改变。不需要手工推进和分配值。列数从 1 开始。

绑定使用 Perl 别名在非常低的级别进行，因此不会发生额外的拷贝。只要驱动器调用正确的推进函数得到数组即可，它会自动支持列绑定。

为了最大化在驱动器之间的可移植性，应该在执行后调用 bind_col。

bind_param 方法对于输出变量执行相似的函数。

### bind_columns

```
$rc = $sth->bind_columns(\%attr, @list_of_refs_to_vars_to_bind);
```

在 select 语句的每个列上调用 bind_col，如果引用的列数不匹配字段的数量，那么 bind_columns 会 die。不需要这么做，不过在有些应用中可能有用。

为了最大化驱动器之间的可移植性，应该在执行后调用 bind_columns。例如：

```
$dbh->{RaiseError} = 1; # do this, or check every call for errors

$sth = $dbh->prepare(q{ select region, sales from sales_by_region });

$sth->execute;

my ($region, $sales);

# Bind perl variables to columns:

$rv = $sth->bind_columns(undef, \$region, \$sales);

# you can also use perl's \(...) syntax (see perlref docs):

#     $sth->bind_columns(undef, \($region, $sales));

# Column binding is the most efficient way to fetch data

while ($sth->fetch) {print "$region: $sales\n";}
```

### dump_results

```
$rows = $sth->dump_results($maxlen, $lsep, $fsep, $fh);
```

从$sth 提取所有的行，在每一行上调用 DBI::neat_list 并输出结果到$fh（默认为 STDOUT），以$lsep（默认为"\n"）分隔。$fsep 默认为",", $maxlen 默认为 35。

该方法被设计用于原型和测试查询。因为调用了 neat_list，它会使用 neat 格式化字符串，因此不推荐在数据传输的应用程序中使用。

## 语句处理器属性

这部分描述与语句相关的属性，大部分属性是只读的。

改变这些属性不会影响其它处理器以及之后的处理器。

尝试得到或设置一个未知的属性是致命的，除了驱动器相关的属性外（这些属性的名字以小写开头）。

例如：

```
... = $h->{NUM_OF_FIELDS};    # get/read
```

注意：某些驱动器不为这些属性提供有效的值，除非调用了$sth->execute。

查看 finish 得到它可能影响的属性。

### NUM_OF_FIELDS (integer, read-only)

Prepared 语句将返回的字段数。非 select 语句 NUM_OF_FIELDS == 0。

### NUM_OF_PARAMS (integer, read-only)

Prepared 语句中的占位符的数量。查看 SUBSTITUTION VARIABLES 得到更详细的信息。

### NAME (array-ref, read-only)

返回一个到包含每个列的字段名的数组的引用。明子可能包含空格但不应该被截断或有任何尾随空格。

```
print "First column name: $sth->{NAME}->[0]\n";
```

### TYPE (array-ref, read-only) *NEW*

返回一个包含每个列的整形值得数组的引用。该值指示了相应的列的数据类型。

该值对应于 ANSI X3.135 和 ISO/IEC 9075 标准，从术语的角度来说就是 ODBC。不精确匹配标准类型的驱动器类型通常返回和 ODBC 驱动器一样的值。这样可能会返回供应商官方注册的类型号。

TYPE 所有可能的值应该在 type_info_all 方法的输出中至少有一个条目。

### PRECISION (array-ref, read-only) *NEW*

返回一个包含每个列的整形值得数组的引用。对于非数字列，该值通常为最大长度或者定义的长度。对于数字列，该值为小数点后的位数（不包括小数点和正负号）。对于浮点型（REAL, FLOAT, DOUBLE），显示的长度可以最大可以大于精度的 7 个字符。

### SCALE (array-ref, read-only) *NEW*

返回一个包含每个列的整形值得数组的引用。

### NULLABLE (array-ref, read-only)

返回一个数组的引用指示每个列是否返回一个 null：0 = no, 1 = yes, 2 = unknown。

```
print "First column may return NULL\n" if $sth->{NULLABLE}->[0];
```

### CursorName (string, read-only)

如果可用，返回一个与语句处理器相关的游标名。如果不可用活着数据库驱动器不支持"where current of ..."的 SQL 语法，那么将返回 undef。

### Statement (string, read-only) *NEW*

返回传递给 prepare 方法的语句字符串。

### RowsInCache (integer, read-only) *NEW*

如果驱动器支持本地行缓存，那么该属性将用来保存在缓存中还未推进的的行数。如果驱动器不支持，将返回 undef。不过有些驱动器在执行时预推进，还有一些则等待第一次推进。

查看 RowCacheSize 属性。

## 其他的信息

### 事务

事物是任何自动化数据库系统的基础组成部分，它们通过确保在数据库中发生的相关改变原子性来防止错误和数据库中断发生。

这一部分应用于支持事务并且 AutoCommit 为 OFF 的数据库。在 Perl 应用程序中实施自动化事务的推荐方法是使用 eval { ... }（它比"..."更快）。

```
eval {
    foo(...)   # do lots of work here
    bar(...)   # including inserts
    baz(...)   # and updates
};
if ($@) {
    # $@ contains $DBI::errstr if DBI RaiseError caused die
    $dbh->rollback;
    # add other application on-error-clean-up code here
}
```

```
else {
    $dbh->commit;
}
```

在 foo()中的代码或其中执行其他代码，可以通过以下方式实施：

```
$h->method(@args) || die $h->errstr
```

或者设置$h->{RaiseError}属性为 on。如果设置了 RaiseError ，DBI 会在处理器上任何调用的方法失败时 die，因此不需要测试返回码。

Eval 方法的主要好处是如果内部应用程序中的任何代码因为任何原因 die 事物都能被恰当回滚，使用$h->{RaiseError}属性的主要好处是所有的 DBI 调用都会被自动检查，因此这两种技术都被强烈推荐。

## 处理 BLOB/CLOB/memo 字段

许多数据库支持'blob'，'long'和类似的数据类型以支持很长的字符串或者大量的二进制数据。

因为那么大的值通常不会保持在内存中，并且数据库通常无法提前知道一个 select 语句返回的 long 字段的最长长度，因此需要一些特殊的处理。

在这种情况下，$h->{LongReadLen}属性用来确定在推进这些字段时分配的属性。$h->{LongTruncOk}用来决定如果推进的字段不适合于放入缓冲时采取的行为。

当试图插入 long 或者二进制数据时，应该使用占位符因为一个语句的长度通常有限制并且 quote 方法通常不能处理二进制数据。

## 简单的例子

Select 和提取数据的完整的例子如下：

```
my $dbh = DBI->connect("dbi:DriverName:db_name", $user, $password)|| die "Can't connect to $data_source: $DBI::errstr";

my $sth = $dbh->prepare( q{ SELECT name, phone FROM mytelbook }) || die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute || die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";

print "Field names: @{ $sth->{NAME} }\n";

while (($name, $phone) = $sth->fetchrow_array) { print "$name: $phone\n"; }

# check for problems which may have terminated the fetch early
```

```
    die $sth->errstr if $sth->err;

    $dbh->disconnect;
```

从一个文件插入数据的程序如下：

```
    my $dbh = DBI->connect("dbi:DriverName:db_name", $user, $password, { RaiseError =>
1, AutoCommit => 0 });

    my $sth = $dbh->prepare( q{ INSERT INTO table (name, phone) VALUES (?, ?) });

    open FH, "<phone.csv" or die "Unable to open phone.csv: $!";

    while (<FH>) {

        chop;

        my ($name, $phone) = split /,/;

       $sth->execute($name, $phone);

    }

    close FH;

      $dbh->commit;

      $dbh->disconnect;
```

转换推进的 NULLs（未定义的值）为空值：

```
  while($row = $sth->fetchrow_arrayref) {

    # this is a fast and simple way to deal with nulls:

    foreach (@$row) { $_ = '' unless defined }

    print "@$row\n";

  }
```

使用 q{...}是为了避免 SQL 中使用的引用冲突，如果要进行转义，则使用 qq{...}操作符。查看 perlop 得到更详细的信息。

## 线程和线程安全性

Perl 5.004_50(+)在很多平台上支持线程，DBI 应该建立在这些平台上，但是当前没有任何安全性可言。

## 信号处理和取消操作

在当前版本的 Perl 中，信号操作是不安全的。在处理或一个信号时或者之后，通常会有一点 Perl 崩溃和内核 dump 的危险。在版本 5.004_04 中，该威胁减少了，但是仍然存在。

与 DBI 相关的两种常见的操作是用户按下 Ctrl-C 取消操作，或者使用 alarm()和$SIG{ALRM}实施超时。

为了扶助这些方法，DBI 在语句处理器上提供了 cancel 方法。Cancel 应该忽略当前的操作并且设计为从信号处理器调用。

但是它必须确保：在任何一个时间，只有很少的驱动器实施该方法；即使已经实施了，仍然有可能语句处理器和父数据库处理器，在之后不能使用。

如果 cancel 操作返回 true，则表示成功的调用了数据库引擎取消其操作。否则调用失败。

# 调试

除了 trace 方法，可以在启动 Perl 前设置环境变量 DBI_TRACE 启用相同的跟踪。

在类 unix 系统上，可以使用以下一条命令完成该任务：

```
DBI_TRACE=2 perl your_test_script.pl
```

如果 DBI_TRACE 设置为非数字，那么将会假设那个为文件名，并且设置跟踪级别为 2。

# 警告和错误消息

## 致命的错误

### Can't call method "prepare" without a package or object reference

用来调用 prepare 的$dbh 处理器可能未定义，因为先前的连接失败。应该检查 DBI 方法返回的状态，或使用 RaiseError 属性。

### Can't call method "execute" without a package or object reference

用来调用 execute 的$dbh 处理器可能未定义，因为先前的 prepare 失败。应该检查 DBI 方法返回的状态，或使用 RaiseError 属性。

### Database handle destroyed without explicit disconnect

### DBI/DBD internal version mismatch

### DBD driver has not implemented the AutoCommit attribute

### Can't [sg]et %s->{%s}: unrecognised attribute

### panic: DBI active kids (%d) > kids (%d)

### panic: DBI active kids (%d) < 0 or > kids (%d)

### 警告

### DBI Handle cleared whilst still holding %d cached kids!

DBI Handle cleared whilst still active!

DBI Handle has uncleared implementors data

DBI Handle has %d uncleared child handles

## 其他

### 数据库文档

SQL 语言参考手册（如 Oracle SQL,MS SQL,当然 SQL 92/99 也是很重要的。）

### 书籍和日志

Perl 编程（Larry Wall, Tom Christiansen & Randal Schwartz 著），Perl 学习（Randal Schwartz 著）。

### 手册页

perl(1), perlmod(1), perlbook(1)。

### 邮件列表

DBI 邮件列表是 DBI 和相关的模块的用户交流的主要方式，可以通过 www.fugue.com/dbi 订阅。邮件列表归档保存在以下网址：

http://www.rosat.mpe-garching.mpg.de/mailing-lists/PerlDB-Interest/

http://www.xray.mpe.mpg.de/mailing-lists/#dbi

http://outside.organic.com/mail-archives/dbi-users/

http://www.coe.missouri.edu/~faq/lists/dbi.html

## 相关的 www 链接

### DBI 主页：

http://www.arcana.co.uk/technologia/perl/DBI

### 其他相关的联接：

http://eskimo.tamu.edu/~jbaker/dbi-examples.html

http://wdvl.com/Authoring/DB/Intro/toc.html

http://www-ccs.cs.umass.edu/db.html

http://www.odmg.org/odmg93/updates_dbarry.html

http://www.jcc.com/sql_stnd.html

ftp://alpha.gnu.ai.mit.edu/gnu/gnusql-0.7b3.tar.gz

http://www.dbmsmag.com

### 推荐的 Perl 编程链接：

http://language.perl.com/style/

## FAQ

可以使用 perldoc DBI::FAQ 命令查看安装为 DBI::FAQ 模块的 DBI FAQ。

# 作者

DBI 属于 Tim Bunce，该产品文档由 Tim Bunce, J. Douglas Dunlop, Jonathan Leffler 等编写。Perl 属于 Larry Wall 和 perl5 相关人员。

# 版权

可以在 Perl README 中声明的 GNU 或 Artistic 许可的约束下进行发布。

# 翻译

该手册和其他 Perl 模块的德语版可用，在：

http://www.oreilly.de/catalog/perlmodger/manpages.html

# 支持和责任

DBI 是免费的。

The Perl Clinic 提供 Perl 和 DBI, DBD::Oracle 以及 Oraperl 模块的商业支持。查看 http://www.perlclinic.com 得到更详细的信息。

## 未解决的问题

数据类型（ISO 数字类型和类型名习惯）；

错误处理器；

数据库字典方法；

移植性；

Blob 读取；

等等；

## 常见问题

查看 DBI FAQ 得到更详细的 FAQ 列表。使用 perldoc DBI::FAQ 方法阅读。

### DBI 有多快？

为了测试 DBI 和 DBD::Oracle 代码的速度，开发者更改了 DBD::Oracle 以使用户可以设置一个属性，它会导致相同的行被重复提取（不需要调用 Oracle 代码但是为一个提取在代码路径中重复执行所有 DBI 和 DBD::Oracle 代码）。

使用以下代码使提取 50000 行：

```
1 while $csr->fetch;
```

结果是：一个字段：5300 次每 CPU/秒；10 个字段：4000 次每 CPU/秒；

不过的平台结果可能完全不一样。不过为了比较，使用代码：

```
1 while @row = $csr->fetchrow_array;
```

（fetchrow_array 和 ora_fetch 大致相同）如下：

```
one field:   3100 fetches per cpu second (approx)

ten fields:  1000 fetches per cpu second (approx)
```

注意速度的减慢，并且对于更多的列影响程度急剧增加。

改变重复为使用推进执行实际的工作，如下：

```
while(@row = $csr->fetchrow_array) {
```

```
        $hash{++$i} = [ @row ];
    }
```

结果：10 个字段，500 次每 CPU/秒；

因此，可以说 DBI 和 DBD::Oracle 的负载相对于 Perl 语言的负载和可能的数据库负载是很小的。

因此如果认为 DBI 或驱动器比较慢，尝试使用以下语句代替循环，如下：

```
1 while $csr->fetch;
```

之后，如果可以接受，那么就可以检查代码了。如果不能帮助解决问题，那么应该检查数据库，平台，网络等。不过记住在把问题归为 DBI 前考虑清楚。

## 为什么我的 CGI 脚本不能正确工作？

查看以下引用，不要提交 CGI 相关的信息到 dbi-users 邮件列表：

http://www.perl.com/perl/faq/idiots-guide.html

http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml

http://www.perl.com/perl/faq/perl-cgi-faq.html

http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html

http://www.boutell.com/faq/

http://www.perl.com/perl/faq/

常见的问题和建议：

使用 CGI::ErrorWrap 模块。记住需要环境变量不会在 CGI 脚本中被设置。

## 如何维护到一个数据库的 WWW 连接？

关于 Apache httpd 服务器和 mod_perl 模块的信息，查看 http://perl.apache.org/。

## 驱动器建立失败因为找不到 DBIXS.h

在 0.77 中 DBIXS.h 的安装位置更新了（它被安装到了一个错误的目录，但是却是驱动器开发者希望的）。第一件事是检查是否有最新版本的驱动器，驱动器开发者会发布使用新位置的版本。如果有最新版的，那么应该去向 SA 申请。也可以自己编辑 Makefile.PL 文件，更改"-I.../DBI"为"-I.../auto/DBI"，...为非空字符串。

## DBI 和 DBD::Foo 是否已导入 NT / Win32?

最新版本的 DBI，并且至少 DBD::Oracle 模块将会没有改变的建立在 NT/Win32 上，不过确保使用标准的 Perl 5.004 并且不是 ActiveWare 端口。

### ODBC 呢？

DBD::ODBC 模块可用。

### DBI 是否有 2000 年问题？

没有。DBI 完全没有日期概念，也不理解。

各自的驱动器（DBD::*）具有日期处理代码，不过在它们的代码里不太可能有 2000 年问题，但是使用 DBI 和 DBD 驱动器的应用程序如果没有设计和编写的好，可能会有 2000 年问题。

## 已知的驱动器模块

### Altera - DBD::Altera

作者：Dimitrios Souflis

邮件：dsouflis@altera.gr

### ODBC - DBD::ODBC

作者：Tim Bunce

邮件：dbi-users@fugue.com

### Oracle - DBD::Oracle

作者：Tim Bunce

邮件：dbi-users@fugue.com

### Ingres - DBD::Ingres

作者：Henrik Tougaard

邮件：ht@datani.dk, dbi-users@fugue.com

### mSQL - DBD::mSQL

### DB2 - DBD::DB2

### Empress - DBD::Empress

### Informix - DBD::Informix

作者：Jonathan Leffler

邮件：jleffler@informix.com, j.leffler@acm.org, dbi-users@fugue.com

### Solid - DBD::Solid

作者：Thomas Wenrich

邮件：wenrich@site58.ping.at, dbi-users@fugue.com

### Postgres - DBD::Pg

作者：Edmund Mergl

邮件：E.Mergl@bawue.de, dbi-users@fugue.com

### Illustra - DBD::Illustra

作者：Peter Haworth

邮件：pmh@edison.ioppublishing.com, dbi-users@fugue.com

### Fulcrum SearchServer - DBD::Fulcrum

作者：Davide Migliavacca

邮件：davide.migliavacca@inferentia.it

### XBase (dBase) - DBD::XBase

作者：Honza Pazdziora

邮件：adelton@fi.muni.cz

## 其他相关的工作和 Perl 模块

Apache::DBI by E.Mergl@bawue.de

为了与 Apache daemon 和嵌入式 perl 诊断器如 mod_perl 一起使用。建立一个连接用于在整个 http daemon 期间保持打开，对于每次的数据库访问这种方法的 CGI 连接和断开开始成为多余的。

JDBC Server by Stuart 'Zen' Bishop zen@bf.rmit.edu.au

该服务器是 Perl 编写的， 与他交互的客户端类是 Java。因此，Java applet 或者应用程序可以通过 JDBC API 与任何安装了 DBI 驱动的数据库通信。URL 使用 jdbc:dbi://host.domain.etc:999/Driver/DBName 的格式。看起来非常像 jdbcKona 的商业产品。

远程 Proxy DBD 支持

Carl Declerck carl@miskatonic.inbe.net

Terry Greenlaw z50816@mip.mar.lmco.com

SQL Parser

Hugo van der Sanden <hv@crypt.compulink.co.uk>

Stephen Zander stephen.zander@mckesson.com

基于 O'Reilly lex/yacc 书中的例子。

由于本人英文能力有限，可能有一定的不恰当表达。

--译者：张君华

--zhjh256@163.com