Ruby 正则表达式 Ruby DBI Read 操作

Ruby 数据库访问 - DBI 教程

本章节将向您讲解如何使用 Ruby 访问数据库。*Ruby DBI* 模块为 Ruby 脚本提供了类似于 Perl DBI 模块的独立于数据库的接口。

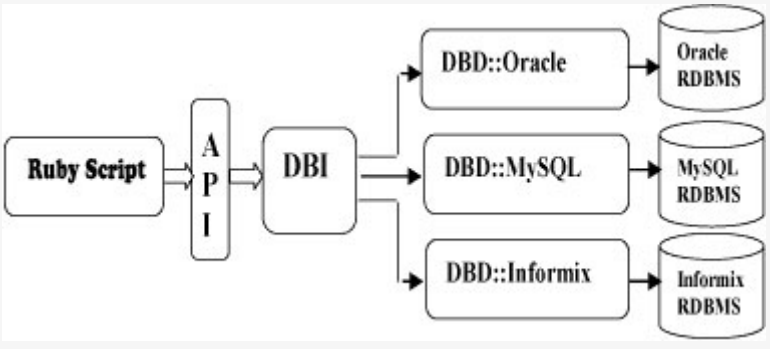
DBI 即 Database independent interface，代表了 Ruby 独立于数据库的接口。DBI 在 Ruby 代码与底层数据库之间提供了一个抽象层，允许您简单地实现数据库切换。它定义了一系列方法、变量和规范，提供了一个独立于数据库的一致的数据库接口。

DBI 可与下列进行交互：

- ADO (ActiveX Data Objects)
- DB2
- Frontbase
- mSQL
- MySQL
- ODBC
- Oracle
- OCI8 (Oracle)
- PostgreSQL
- Proxy/Server
- SQLite
- SQLRelay

DBI 应用架构

DBI 独立于任何在后台中可用的数据库。无论您使用的是 Oracle、MySQL、Informix，您都可以使用 DBI。下面的架构图清晰地说明了这点。



Ruby DBI 一般的架构使用两个层：

- 数据库接口（DBI）层。该层是独立于数据库，并提供了一系列公共访问方法，方法的使用不分数据库服务器类型。
- 数据库驱动（DBD）层。该层是依赖于数据库，不同的驱动提供了对不同的数据库引擎的访问。MySQL、PostgreSQL、InterBase、Oracle 等分别使用不同的驱动。每个驱动都负责解释来自 DBI 层的请求，并把这些请求映射为适用于给定类型的数据库服务器的请求。

安装

如果您想要编写 Ruby 脚本来访问 MySQL 数据库，您需要先安装 Ruby MySQL 模块。

安装 Mysql 开发包

```
# Ubuntu
sudo apt-get install mysql-client
sudo apt-get install libmysqlclient15-dev
# Centos
yum install mysql-devel
```

Mac OS 系统需要修改 ~/.bash_profile 或 ~/.profile 文件，添加如下代码:

```
MYSQL=/usr/local/mysql/bin
export PATH=$PATH:$MYSQL
export DYLD_LIBRARY_PATH=/usr/local/mysql/lib:$DYLD_LIBRARY_PATH
```

或者使用软连接：

```
sudo ln -s /usr/local/mysql/lib/libmysqlclient.18.dylib /usr/lib/libmysqlclient.
18.dylib
```

使用 RubyGems 安装 DBI（推荐）

RubyGems大约创建于2003年11月，从Ruby 1.9版起成为Ruby标准库的一部分。更多详情可以查看：Rub
y RubyGems

使用 gem 安装 dbi 与 dbd-mysql：

```
sudo gem install dbi
sudo gem install mysql
sudo gem install dbd-mysql
```

使用源码安装（Ruby版本 小于1.9的使用此方法）

该模块是一个 DBD，可从 http://tmtm.org/downloads/mysql/ruby/ 上下载。

下载后最新包，解压进入到目录，执行以下命令安装:

```
ruby extconf.rb
或者
ruby extconf.rb --with-mysql-dir=/usr/local/mysql
或者
ruby extconf.rb --with-mysql-config
```

然后编译：

```
make
```

获取并安装 Ruby/DBI

您可以从下面的链接下载并安装 Ruby DBI 模块：

https://github.com/erikh/ruby-dbi

在开始安装之前，请确保您拥有 root 权限。现在，请安装下面的步骤进行安装：

步骤 1

```
git clone https://github.com/erikh/ruby-dbi.git
```

或者直接下再 zip 包并解压。

步骤 2

进入目录 *ruby-dbi-master*，在目录中使用 *setup.rb* 脚本进行配置。最常用的配置命令是 config 参数后不跟任何参数。该命令默认配置为安装所有的驱动。

```
ruby setup.rb config
```

更具体地，您可以使用 --with 选项来列出了您要使用的特定部分。例如，如果只想配置主要的 DBI 模块和 MySQL DBD 层驱动，请输入下面的命令：

```
ruby setup.rb config --with=dbi,dbd_mysql
```

步骤 3

最后一步是建立驱动器，使用下面命令进行安装：

```
ruby setup.rb setup
ruby setup.rb install
```

数据库连接

假设我们使用的是 MySQL 数据库，在连接数据库之前，请确保：

- 您已经创建了一个数据库 TESTDB。
- 您已经在 TESTDB 中创建了表 EMPLOYEE。
- 该表带有字段 FIRST_NAME、LAST_NAME、AGE、SEX 和 INCOME。
- 设置用户 ID "testuser" 和密码 "test123" 来访问 TESTDB
- 已经在您的机器上正确地安装了 Ruby 模块 DBI。
- 您已经看过 MySQL 教程，理解了 MySQL 基础操作。

下面是连接 MySQL 数据库 "TESTDB" 的实例：

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
```

```
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
"testuser", "test123")
# 获取服务器版本字符串，并显示
row = dbh.select_one("SELECT VERSION()")
puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

当运行这段脚本时，将会在 Linux 机器上产生以下结果。

```
Server version: 5.0.45
```

如果建立连接时带有数据源，则返回数据库句柄（Database Handle），并保存到 **dbh** 中以便后续使用，否则 **dbh** 将被设置为 nil 值，*e.err* 和 *e::errstr* 分别返回错误代码和错误字符串。

最后，在退出这段程序之前，请确保关闭数据库连接，释放资源。

INSERT 操作

当您想要在数据库表中创建记录时，需要用到 INSERT 操作。

一旦建立了数据库连接，我们就可以准备使用 **do** 方法或 **prepare** 和 **execute** 方法创建表或创建插入数据表中的记录。

使用 do 语句

不返回行的语句可通过调用 **do** 数据库处理方法。该方法带有一个语句字符串参数，并返回该语句所影响的行数。

```
dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
FIRST_NAME CHAR(20) NOT NULL,
LAST_NAME CHAR(20),
AGE INT,
SEX CHAR(1),
INCOME FLOAT )" );
```

同样地，您可以执行 SQL *INSERT* 语句来创建记录插入 EMPLOYEE 表中。

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
```

```
"testuser", "test123")
dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME,
LAST_NAME,
AGE,
SEX,
INCOME)
VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
puts "Record has been created"
dbh.commit
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
dbh.rollback
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

使用 *prepare* 和 *execute*

您可以使用 DBI 的 *prepare* 和 *execute* 方法来执行 Ruby 代码中的 SQL 语句。

创建记录的步骤如下：

- 准备带有 INSERT 语句的 SQL 语句。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** API 来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

下面是使用这两种方法的语法：

实例

```
sth = dbh.prepare(statement)
sth.execute
... zero or more SQL operations ...
sth.finish
```

这两种方法可用于传 **bind** 值给 SQL 语句。有时候被输入的值可能未事先给出，在这种情况下，则会用到绑定值。使用问号（**?**）替代实际值，实际值通过 execute() API 来传递。

下面的实例在 EMPLOYEE 表中创建了两个记录：

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
"testuser", "test123")
sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME,
```

```
LAST_NAME,
AGE,
SEX,
INCOME)
VALUES (?, ?, ?, ?, ?)" )
sth.execute('John', 'Poul', 25, 'M', 2300)
sth.execute('Zara', 'Ali', 17, 'F', 1000)
sth.finish
dbh.commit
puts "Record has been created"
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
dbh.rollback
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

如果同时使用多个 INSERT，那么先准备一个语句，然后在一个循环中多次执行它要比通过循环每次调用 do 有效率得多。

READ 操作

对任何数据库的 READ 操作是指从数据库中获取有用的信息。

一旦建立了数据库连接，我们就可以准备查询数据库。我们可以使用 **do** 方法或 **prepare** 和 **execute** 方法从数据库表中获取值。

获取记录的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 逐一获取结果，并输出这些结果。这将通过使用 **fetch** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。

下面的实例从 EMPLOYEE 表中查询所有工资（salary）超过 1000 的记录。

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
"testuser", "test123")
sth = dbh.prepare("SELECT * FROM EMPLOYEE
WHERE INCOME > ?")
sth.execute(1000)
sth.fetch do |row|
printf "First Name: %s, Last Name : %s\n", row[0], row[1]
printf "Age: %d, Sex : %s\n", row[2], row[3]
```

```
printf "Salary :%d \n\n", row[4]
end
sth.finish
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

这将产生以下结果：

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000

First Name: John, Last Name : Poul
Age: 25, Sex : M
Salary :2300
```

还有很多从数据库获取记录的方法，如果您感兴趣，可以查看 Ruby DBI Read 操作。

Update 操作

对任何数据库的 UPDATE 操作是指更新数据库中一个或多个已有的记录。下面的实例更新 SEX 为 'M' 的所有记录。在这里，我们将把所有男性的 AGE 增加一岁。这将分为三步：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
"testuser", "test123")
sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1
WHERE SEX = ?")
sth.execute('M')
sth.finish
dbh.commit
rescue DBI::DatabaseError => e
```

```
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
dbh.rollback
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

DELETE 操作

当您想要从数据库中删除记录时，需要用到 DELETE 操作。下面的实例从 EMPLOYEE 中删除 AGE 超过 20 的所有记录。该操作的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中删除所需的记录。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

实例

```
#!/usr/bin/ruby -w
require "dbi"
begin
# 连接到 MySQL 服务器
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
"testuser", "test123")
sth = dbh.prepare("DELETE FROM EMPLOYEE
WHERE AGE > ?")
sth.execute(20)
sth.finish
dbh.commit
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
dbh.rollback
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

执行事务

事务是一种确保交易一致性的机制。事务应具有下列四种属性：

- **原子性（Atomicity）：**事务的原子性指的是，事务中包含的程序作为数据库的逻辑工作单位，它所做的对数据修改操作要么全部执行，要么完全不执行。

- **一致性（Consistency）：**事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。假如数据库的状态满足所有的完整性约束，就说该数据库是一致的。
- **隔离性（Isolation）：**事务的隔离性指并发的事务是相互隔离的，即一个事务内部的操作及正在操作的数据必须封锁起来，不被其它企图进行修改的事务看到。
- **持久性（Durability）：**事务的持久性意味着当系统或介质发生故障时，确保已提交事务的更新不能丢失。即一旦一个事务提交，它对数据库中数据的改变应该是永久性的，耐得住任何数据库系统故障。持久性通过数据库备份和恢复来保证。

DBI 提供了两种执行事务的方法。一种是 *commit* 或 *rollback* 方法，用于提交或回滚事务。还有一种是 *transaction* 方法，可用于实现事务。接下来我们来介绍这两种简单的实现事务的方法：

方法 I

第一种方法使用 DBI 的 *commit* 和 *rollback* 方法来显式地提交或取消事务：

实例

```
dbh['AutoCommit'] = false # 设置自动提交为 false.
begin
dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
WHERE FIRST_NAME = 'John'")
dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
WHERE FIRST_NAME = 'Zara'")
dbh.commit
rescue
puts "transaction failed"
dbh.rollback
end
dbh['AutoCommit'] = true
```

方法 II

第二种方法使用 *transaction* 方法。这个方法相对简单些，因为它需要一个包含构成事务语句的代码块。*transaction* 方法执行块，然后根据块是否执行成功，自动调用 *commit* 或 *rollback*：

实例

```
dbh['AutoCommit'] = false # 设置自动提交为 false
dbh.transaction do |dbh|
dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
WHERE FIRST_NAME = 'John'")
dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
WHERE FIRST_NAME = 'Zara'")
end
dbh['AutoCommit'] = true
```

COMMIT 操作

Commit 是一种标识数据库已完成更改的操作，在这个操作后，所有的更改都不可恢复。

下面是一个调用 **commit** 方法的简单实例。

```
dbh.commit
```

ROLLBACK 操作

如果您不满意某个或某几个更改，您想要完全恢复这些更改，则使用 **rollback** 方法。

下面是一个调用 **rollback** 方法的简单实例。

```
dbh.rollback
```

断开数据库

如需断开数据库连接，请使用 disconnect API。

```
dbh.disconnect
```

如果用户通过 disconnect 方法关闭了数据库连接，DBI 会回滚所有未完成的事务。但是，不需要依赖于任何 DBI 的实现细节，您的应用程序就能很好地显式调用 commit 或 rollback。

处理错误

有许多不同的错误来源。比如在执行 SQL 语句时的语法错误，或者是连接失败，又或者是对一个已经取消的或完成的语句句柄调用 fetch 方法。

如果某个 DBI 方法失败，DBI 会抛出异常。DBI 方法会抛出任何类型的异常，但是最重要的两种异常类是 *DBI::InterfaceError* 和 *DBI::DatabaseError*。

这些类的 Exception 对象有 *err*、*errstr* 和 *state* 三种属性，分表代表了错误号、一个描述性的错误字符串和一个标准的错误代码。属性具体说明如下：

- **err：**返回所发生的错误的整数表示法，如果 DBD 不支持则返回 *nil*。例如，Oracle DBD 返回 *ORA-XXXX* 错误消息的数字部分。
- **errstr：**返回所发生的错误的字符串表示法。
- **state：**返回所发生的错误的 SQLSTATE 代码。SQLSTATE 是五字符长度的字符串。大多数的 DBD 并不支持它，所以会返回 nil。

在上面的实例中您已经看过下面的代码：

```
rescue DBI::DatabaseError => e
puts "An error occurred"
puts "Error code: #{e.err}"
puts "Error message: #{e.errstr}"
dbh.rollback
ensure
# 断开与服务器的连接
dbh.disconnect if dbh
end
```

为了获取脚本执行时有关脚本执行内容的调试信息，您可以启用跟踪。为此，您必须首先下载 dbi/trace 模块，然后调用控制跟踪模式和输出目的地的 *trace* 方法：

```
require "dbi/trace"
..............
trace(mode, destination)
```

mode 的值可以是 0（off）、1、2 或 3，destination 的值应该是一个 IO 对象。默认值分别是 2 和 STDERR。

方法的代码块

有一些创建句柄的方法。这些方法通过代码块调用。使用带有方法的代码块的优点是，它们为代码块提供了句柄作为参数，当块终止时会自动清除句柄。下面是一些实例，有助于理解这个概念。

- **DBI.connect ：**该方法生成一个数据库句柄，建议在块的末尾调用 *disconnect* 来断开数据库。
- **dbh.prepare ：**该方法生成一个语句句柄，建议在块的末尾调用 *finish*。在块内，您必须调用 *execute* 方法来执行语句。
- **dbh.execute ：**该方法与 dbh.prepare 类似，但是 dbh.execute 不需要在块内调用 execute 方法。语句句柄会自动执行。

实例 1

DBI.connect 可带有一个代码块，向它传递数据库句柄，且会在块的末尾自动断开句柄。

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                  "testuser", "test123") do |dbh|
```

实例 2

dbh.prepare 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.prepare("SHOW DATABASES") do |sth|
sth.execute
puts "Databases: " + sth.fetch_all.join(", ")
end
```

实例 3

dbh.execute 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.execute("SHOW DATABASES") do |sth|
puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* 方法也可带有一个代码块，这在上面的章节中已经讲解过了。

特定驱动程序的函数和属性

DBI 让数据库驱动程序提供了额外的特定数据库的函数，这些函数可被用户通过任何 Handle 对象的 *func* 方法进行调用。

使用 **[]=** or **[]** 方法可以设置或获取特定驱动程序的属性。

DBD::Mysql 实现了下列特定驱动程序的函数：

序号	函数 & 描述
1	**dbh.func(:createdb, db_name)** 创建一个新的数据库。
2	**dbh.func(:dropdb, db_name)** 删除一个数据库。
3	**dbh.func(:reload)** 执行重新加载操作。
4	**dbh.func(:shutdown)** 关闭服务器。
5	**dbh.func(:insert_id) => Fixnum** 返回该连接的最近 AUTO_INCREMENT 值。
6	**dbh.func(:client_info) => String** 根据版本返回 MySQL 客户端信息。
7	**dbh.func(:client_version) => Fixnum** 根据版本返回客户端信息。这与 :client_info 类似，但是它会返回一个 fixnum，而不是返回字符串。
8	**dbh.func(:host_info) => String** 返回主机信息。
9	**dbh.func(:proto_info) => Fixnum** 返回用于通信的协议。
10	**dbh.func(:server_info) => String** 根据版本返回 MySQL 服务器端信息。
11	**dbh.func(:stat) => Stringb>** **返回数据库的当前状态。**
12	**dbh.func(:thread_id) => Fixnum** 返回当前线程的 ID。

```
#!/usr/bin/ruby
```

```ruby
require "dbi"
begin
   # 连接到 MySQL 服务器
   dbh = DBI.connect("DBI:Mysql:TESTDB:localhost"
,
                     "testuser", "test123")
   puts dbh.func(:client_info)
   puts dbh.func(:client_version)
   puts dbh.func(:host_info)
   puts dbh.func(:proto_info)
   puts dbh.func(:server_info)
   puts dbh.func(:thread_id)
   puts dbh.func(:stat)
rescue DBI::DatabaseError => e
   puts "An error occurred"
   puts "Error code:    #{e.err}"
   puts "Error message: #{e.errstr}"
ensure
   dbh.disconnect if dbh
end
```

这将产生以下结果：

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
```

```
Opens: 324  Flush tables: 1  Open tables: 64  \
Queries per second avg: 2.860
```