No.1 文档概要

在Golang中使用SQL或类似SQL的数据库的惯用方法是通过 database/sql 包操作。它为面向行的数据库提供了轻量级的接口。这篇文章是关于如何使用它，最常见的参考。

为什么需要这个？包文档告诉你每件事情都做了什么，但它并没有告诉你如何使用这个包。我们很多人都希望自己能快速参考和入门的方法，而不是讲故事。欢迎捐款；请在这里发送请求。

在Golang中你用sql.DB访问数据库。你可以使用此类型创建语句和事务，执行查询，并获取结果。下面的代码列出了sql.DB是一个结构体，点击 database/sql/sql.go 查看官方源码。

首先你应该知道一个sql.DB不是一个数据库的连接。它也没有映射到任何特点数据库软件的"数据库"或"模式"的概念。它是数据库的接口和数据库的抽象，它可能与本地文件不同，可以通过网络连接访问，也可以在内存和进程中访问。

sql.DB为你在幕后执行一些重要的任务：

- 通过驱动程序打开和关闭实际的底层数据库的连接。
- 它根据需要管理一个连接池，这可能是如上所述的各种各样的事情。

sql.DB抽象旨在让你不必担心如何管理对基础数据存储的并发访问。一个连接在使用它执行任

务时被标记为可用，然后当它不在使用时返回到可用的池中。这样的后果之一是，如果你无法将连接释放到池中，则可能导致db.SQL打开大量连接，可能会耗尽资源(连接太多，打开的文件句柄太多，缺少可用网络端口等)。稍后我们将进一步讨论这个问题。

在创建sql.DB之后，你可以用它来查询它所代表的数据库，以及创建语句和事务。

No.2 导入数据库驱动

要使用 database/sql，你需要 database/sql 自身，以及需要使用的特定的数据库驱动。

你通常不应该直接使用驱动包，尽管有些驱动鼓励你这样做。(在我们看来，这通常是个坏主意。) 相反的，如果可能，你的代码应该仅引用 database/sql 中定义的类型。这有助于避免使你的代码依赖于驱动，从而可以通过最少的代码来更改底层驱动(因此访问的数据库)。它还强制你使用Golang习惯用法，而不是特定驱动作者可能提供的特定的习惯用法。

在本文档中，我们将使用@julienschmidt 和 @arnehormann中优秀的MySql驱动。

将以下内容添加到Go源文件的顶部(也就是package name下面)：

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

注意我们正在加载的驱动是匿名的，将其限定符别名为_，因此我们的代码中没有一个导出的名称可见。在引擎下，驱动将自身注册为可用于 database/sql 包，但一般来说没有其他情况发生。

现在你已经准备好访问数据库了。

No.3 访问数据库

现在你已经加载了驱动包，就可以创建一个数据库对象sql.DB。创建一个sql.DB你可以使用sql.Open()。Open返回一个*sql.DB。

```
func main() {
    db, err := sql.Open("mysql",
        "user:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()
}
```

在示例中，我们演示了几件事:

1. sql.Open的第一个参数是驱动名称。这是驱动用来注册database/sql的字符串，并且通常与包名相同以避免混淆。例如，它是github.com/go-sql-driver/mysql的MySql驱动(作者:jmhodges)。某些驱动不遵守公约的名称，例如github.com/mattn/go-sqlite3的sqlite3(作

者:matte)和github.com/lib/pq的postgres(作者:mjibson)。

2. 第二个参数是一个驱动特定的语法，它告诉驱动如何访问底层数据存储。在本例中，我们将连接本地的MySql服务器实例中的"hello"数据库。

3. 你应该(几乎)总是检查并处理从所有database/sql操作返回的错误。有一些特殊情况，我们稍后将讨论这样做事没有意义的。

4. 如果sql.DB不应该超出该函数的作用范围，则延迟函数defer db.Close()是惯用的。

也许是反直觉的，sql.Open()不建立与数据库的任何连接，也不会验证驱动连接参数。相反，它只是准备数据库抽象以供以后使用。首次真正的连接底层数据存储区将在第一次需要时懒惰地建立。如果你想立即检查数据库是否可用(例如，检查是否可以建立网络连接并登陆)，请使用db.Ping()来执行此操作，记得检查错误：

```
err = db.Ping()
if err != nil {
    // do something here
}
```

虽然在完成数据库之后Close()数据库是惯用的，但是sql.DB对象被设计为长连接。不要经常Open()和Close()数据库。相反，为你需要访问的每个不同的数据存储创建一个sql.DB对象，并保留它，直到程序访问数据存储完毕。在需要时传递它，或在全局范围内使其可用，但要保持开放。并且不要从短暂的函数中Open()和Close()。相反，通过sql.DB作为参数传递给该短暂的

函数。

如果你不把sql.DB视为长期存在的对象，则可能会遇到诸如重复使用和连接共享不足，耗尽可用的网络资源以及由于TIME_WAIT中剩余大量TCP连接而导致的零星故障的状态。这些问题表明你没有像设计的那样使用database/sql的迹象。

现在是时候使用你的sql.DB对象了。

No.4 检索结果集

有几个惯用的操作来从数据存储中检索结果。

1. 执行返回行的查询。

2. 准备重复使用的语句，多次执行并销毁它。

3. 以一次关闭的方式执行语句，不准备重复使用。

4. 执行一个返回单行的查询。这种特殊情况有一个捷径。

Golang的database/sql函数名非常重要。如果一个函数名包含查询Query()，它被设计为询问数据库的问题，并返回一组行，即使它是空的。不返回行的语句不应该使用Query()函数；他们应该使用Exec()。

从数据库获取数据

让我们来看一下如何查询数据库，使用Query的例子。我们将向用户表查询id为1的用户，并打印出用户的id和name。我们将使用rows.Scan()将结果分配给变量，一次一行。

```
var (
    id int
    name string
)
rows, err := db.Query("select id, name from users where id = ?", 1)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    err := rows.Scan(&id, &name)
    if err != nil {
        log.Fatal(err)
    }
    log.Println(id, name)
}
err = rows.Err()
if err != nil {
    log.Fatal(err)
}
```

下面是上面代码中正在发生的事情：

1. 我们使用db.Query()将查询发送到数据库。我们像往常一样检查错误。

2. 我们用defer内置函数推迟了rows.Close()的执行。这个非常重要。

3. 我们用rows.Next()遍历了数据行。

4. 我们用rows.Scan()读取每行中的列变量。

5. 我们完成遍历行之后检查错误。

这几乎是Golang中唯一的办法。例如，你不能将一行作为映射来获取。这是因为所有东西都是强类型的。你需要创建正确类型的变量并将指针传递给它们，如图所示。

其中的几个部分很容易出错，可能会产生不良后果。

• 你应该总是检查rows.Next()循环结尾处的错误。如果循环中出现错误，则需要了解它。不要仅仅假设循环遍历，直到你已经处理了所有的行。

• 第二，只要有一个打开的结果集(由行代表)，底层连接就很忙，不能用于任何其他查询。这意味着它在连接池中不可用。如果你使用rows.Next()遍历所有行，最终将读取最后一行，rows.Next()将遇到内部EOF错误，并为你调用rows.Close()。但是，如果由于某种原因退出该循环-提前返回，那么行不会关闭，并且连接保持打开状态。(如果rows.Next()由于错误而返回false，则会自动关闭)。这是一种简单耗尽资源的方法。

• rows.Close()是一种无害的操作，如果它已经关闭，所以你可以多次调用它。但是请注意，我们首先检查错误，如果没有错误，则调用rows.Close()，以避免运行时的panic。

• 你应该总是用延迟语句defer推迟rows.Close()，即使你也在循环结束时调用rows.Close()，这不是一个坏主意。

• 不要在循环中用defer推迟。延迟语句在函数退出之前不会执行，所以长时间运行的函数不应该使用它。如果你这样做，你会慢慢积累记忆。如果你在循环中反复查询和使用结果集，则在完成每个结果后应显示的调用rows.Close()，而不用延迟语句defer。

Scan()如何工作

当你遍历行并将其扫描到目标变量中时，Golang会在幕后为你执行数据类型转换。它基于目标变量的类型。意识到这一点可以干净你的代码，并帮助避免重复工作。

例如，假设你从表中选择了一些行，这是用字符串列定义的。如varchar(45)或类似的列。然而，你碰巧知道表格总是包含数字。如果传递指向字符串的指针，Golang会将字节复制到字符串中。现在可以使用strconv.ParseInt()或类似的方式将值转换为数字。你必须检查SQL操作中的错误以及解析整数的错误。这又乱又糟糕。

或者，你可以通过Scan()指向一个整数即可。Golang会检测到并为你调用strconv.ParseInt()。如果有转换错误，则调用Scan()将返回它。你的代码现在更小更整洁。这是推荐使用database/sql的方法。

准备查询

一般来说，你应该总是准备多次使用查询。准备查询的结果是一个准备语句，可以为执行语句

时提供的参数，提供占位符(a.k.a bind值)。这比连接字符串更好，出于所有通常的理由(例如避免SQL注入攻击)。

在MySql中，参数占位符为？，在PostgreSql中为$N,其中N为数字。SQLite接受这两者之一。在Oracle中占位符以冒号开始，并命名为:param1。本文档中我们使用？占位符，因为我们使用MySql作为示例。

```
stmt, err := db.Prepare("select id, name from users where id = ?")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()
rows, err := stmt.Query(1)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    // ...
}
if err = rows.Err(); err != nil {
    log.Fatal(err)
}
```

在引擎下，db.Query()实际上准备，执行和关闭一个准备好的语句。这是数据库的三次往返。如果你不小心，可以使应用程序的数据库交互数量增加三倍！有些驱动可以在特定情况下避免这种情况，但并非所有驱动都可以这样做。点击prepared statements查看更多声明。

单行查询

如果一个查询返回最多一行，可以使用一些快速的样板代码：

```
var name string
err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
if err != nil {
    log.Fatal(err)
}
fmt.Println(name)
```

来自查询的错误将被推迟到Scan()，然后返回。你也可以在准备的语句中调用QueryRow()：

```
stmt, err := db.Prepare("select name from users where id = ?")
if err != nil {
    log.Fatal(err)
}
var name string
err = stmt.QueryRow(1).Scan(&name)
if err != nil {
    log.Fatal(err)
}
fmt.Println(name)
```

No.5 修改数据和使用事务

现在我们已经准备好了如何修改数据和处理事务。如果你习惯于使用“statement”对象来获取行并更新数据，那么这种区别可能视乎是认为的，但是在Golang中有一个重要的原因。

修改数据的statements

使用Exec()，最好用一个准备好的statement来完成INSERT,UPDATE,DELETE或者其他不返回行的语句。下面的示例演示如何插入行并检查有关操作的元数据：

```
stmt, err := db.Prepare("INSERT INTO users(name) VALUES(?)")
if err != nil {
    log.Fatal(err)
}
res, err := stmt.Exec("Dolly")
if err != nil {
    log.Fatal(err)
}
lastId, err := res.LastInsertId()
if err != nil {
    log.Fatal(err)
}
rowCnt, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}
log.Printf("ID = %d, affected = %d\n", lastId, rowCnt)
```

执行该语句将生成一个sql.Result，该语句提供对statement元数据的访问：最后插入的ID和行数受到影响。

如果你不在乎结果怎么办？如果你只想执行一个语句并检查是否有错误，但忽略结果该怎么办？下面两个语句不会做同样的事情吗？

```
_, err := db.Exec("DELETE FROM users")  // OK
_, err := db.Query("DELETE FROM users") // BAD
```

答案是否定的。他们不做同样的事情，你不应该使用Query()。Query()将返回一个sql.Rows，它保留数据库连接，直到sql.Rows关闭。由于可能有未读数据(例如更多的数据行)，所以不能使用连接。在上面的示例中，连接将永远不会被释放。垃圾回收器最终会关闭底层的net.Conn，但这可能需要很长时间。此外，database/sql包将继续跟踪池中的连接，希望在某个时候释放它，以便可以再次使用连接。因此，这种反模式是耗尽资源的好方法(例如连接数太多)。

事务处理

在Golang中，事务本质上是保留与数据存储的连接的对象。它允许你执行我们迄今为止所看到的所有操作，但保证它们将在同一连接上执行。

你可以通过调用db.Begin()开始一个事务，并在结果Tx变量上用Commit()或Rollback()方法关闭它。在封面下，Tx从池中获取连接，并保留它仅用于该事务。Tx上的方法一对一到可以调用数据本本身的方法，例如Query()等等。

在事务中创建的Prepare语句仅限于该事务。点击prepared statements查看更多准备的声明。

你不应该在SQL代码中混合BEGIN和COMMIT相关的函数(如Begin()和Commit()的SQL语句)，可能会导致悲剧：

• Tx对象可以保持打开状态，从池中保留连接而不返回。
• 数据库的状态可能与代表它的Golang变量的状态不同步。
• 你可能会认为你是在事务内部的单个连接上执行查询，实际上Golang已经为你创建了几个连接，而且一些语句不是事务的一部分。

当你在事务中工作时，你应该注意不要对Db变量进行调用。应当使用db.Begin()创建的Tx变量进行所有调用。Db不在一个事务中，只有Tx是。如果你进一步调用db.Exec()或类似的函数，那么这些调用将发生在事务范围之外，是在其他的连接上。

如果你需要处理修改连接状态的多个语句，即使你不希望事务本身，也需要一个Tx。例如：

• 创建仅在一个连接中可见的临时表。

• 设置变量，如MySql's SET @var := somevalue语法。

• 更改连接选项，如字符集或超时。

如果你需要执行任何这些操作，则需要把你的作业(也可以说Tx操作语句)绑定到单个连接，而在Golang中执行此操作的唯一方法是使用Tx。

No.6 使用预处理语句

准备语句(db.Prepare()或者tx.Prepare())在Golang中具有所有常见的优点：安全性，效率，方便性。但是他们的实现方式与你习惯的方式可能有所不同，特别是关于它们如何与database/sql的一些内部组件进行交互的方式。

准备语句和连接

在数据库级别，将准备好的语句绑定到单个数据库连接。典型的流程是：客户端向服务器发送带有占位符的SQL语句以进行准备，服务器使用语句ID进行响应，然后客户端通过发送其ID和参数来执行该语句。

然而在Golang中，连接不会直接暴露给database/sql包的用户。你不准备连接上语句。你准备好在一个db或tx。并且database/sql具有一些便捷的行为，如自动重试。由于这些原因，准备好的语句和连接(存在于驱动级别)之间的潜在关联被隐藏在代码中。

下面是它的工作原理：

1. 准备一个语句时，它会在池中的连接上准备好。

2. Stmt对象记住使用哪个连接。

3. 当你执行Stmt时，它试图使用Stmt对象记住的那个连接(后面我们将这里的连接称为原始连接)。如果它不可用，因为它关闭或忙于做其他事情，它从池中获取另一个连接，并在另一个连接上重新准备与数据库的语句。

因为在原始连接繁忙时，会根据需要重新准备语句，因此数据库的高并发使用可能会导致大量连接繁忙，从而创建大量的准备语句。这会导致语句的明显泄露，正在准备和重新准备的语句比你想象的更多，甚至会影响到服务器端对语句数量的限制。

避免准备好的语句

Golang将为你在封面下创建准备好的声明。例如，一个简单的db.Query(sql,param1,param2)通过准备sql，然后使用参数执行它，最后关闭语句。

有时，准备好的语句并不是你想要的。这可能有几个原因。

1. 数据库不支持准备好的语句。例如，当使用MySql驱动时，你可以连接到MemSql和Sphinx，因为它们支持MySql线路协议。但是它们不支持包含准备语句的"二进制"协议，因此它们会以混乱的方式失败。

2. 这些语句没有重用到足以使它们变得有价值，而安全问题则以其他方式处理，因此性能开销是不需要的。这方面点击VividCortex博客可以看到一个例子。

如果不想使用预处理语句，则需要使用fmt.Sprint()或类似的方法来组合SQL，并将其作为db.Query()或db.QueryRow()的唯一参数传递。你的驱动需要支持明文查询执行，这是通过执行器(Execer是一个结构体)和查询器(Queryer是一个结构体)接口在Golang 1.1中添加的，在此记录。

事务中的准备语句

在Tx中创建的准备语句仅限于它，因此早期关于重新准备的注意事项不适用。当你对Tx对象进行操作时，你的操作直接映射到它下面唯一的一个连接上。

这也意味着在Tx内创建的准备语句不能与之分开使用。同样，在DB中创建的准备语句不能再事务中使用，因为它们将被绑定到不同的连接。

要在Tx中使用事务外的预处理语句，可以使用Tx.Stmt()，它将从事务外部准备一个新的特定于事务的语句。它通过采用现有的预处理语句，设置与事务的连接，并在执行时重新准备所有语句。这个行为及其实现是不可取的，甚至在databse/sql源代码中有一个TODO来改进它；我们建议不要使用这个。

在处理事务中的预处理语句时，必须小心谨慎。请考虑下面的示例：

```
tx, err := db.Begin()
if err != nil {
    log.Fatal(err)
}
defer tx.Rollback()
stmt, err := tx.Prepare("INSERT INTO foo VALUES (?)")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close() // danger!
for i := 0; i < 10; i++ {
    _, err = stmt.Exec(i)
    if err != nil {
        log.Fatal(err)
```

```
    }
}
err = tx.Commit()
if err != nil {
    log.Fatal(err)
}
// stmt.Close() runs here!
```

之前Golang1.4关闭*sql.Tx将与之关联的连接返还到池中，但是，在预处理语句结束后，延迟调用时在那之后发生的，这可能导致并发访问底层的连接，使连接状态不一致。如果使用Golang1.4或更高的版本，则应确保在提交事务或回滚之前声明始终关闭。点击查看这个issues。

参数占位符语法

预处理语句中的占位符参数的语法是特定于数据库的。例如，比较MySql,PostgreSQL,Oracle：

```
MySQL               PostgreSQL           Oracle
=====               ==========           ======
WHERE col = ?       WHERE col = $1       WHERE col = :col
VALUES(?, ?, ?)     VALUES($1, $2, $3)   VALUES(:val1, :val2, :val3)
```

No.7 错误处理

几乎所有使用database/sql类型的操作都会返回一个错误作为最后一个值。你应该总是检查这

些错误，千万不要忽视它们。有几个地方错误行为是特殊情况，还有一些额外的东西可能需要知道。

遍历结果集的错误

请思考下面的代码：

```
for rows.Next() {
    // ...
}
if err = rows.Err(); err != nil {
    // handle the error here
}
```

来自rows.Err()的错误可能是rows.Next()循环中各种错误的结果。除了正常完成循环之外，循环可能会退出，因此你总是需要检查循环是否正常终止。异常终止自动调用rows.Close()，尽管多次调用它是无害的。

关闭结果集的错误

如上所述，如果你过早的退出循环，则应该总是显式的关闭sql.Rows。如果循环正常退出或通过错误，它会自动关闭，但你可能会错误的执行此操作：

```
for rows.Next() {
    // ...
    break; // whoops, rows is not closed! memory leak...
```

```
}
// do the usual "if err = rows.Err()" [omitted here]...
// it's always safe to [re?]close here:
if err = rows.Close(); err != nil {
    // but what should we do if there's an error?
    log.Println(err)
}
```

rows.Close()返回的错误是一般规则的唯一例外，最好是捕获并检查所有数据库操作中的错误。如果rows.Close()返回错误，那么你应该怎么做。记录错误信息或panic可能是唯一明智的事情，如果这不明智，那么也许你应该忽略错误。

QueryRow()的错误

思考下面的代码来获取一行数据：

```
var name string
err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
if err != nil {
    log.Fatal(err)
}
fmt.Println(name)
```

如果没有id = 1的用户怎么办？那么结果中不会有行，而.Scan()不会将值扫描到name中。那会怎么样？

Golang定义了一个特殊的错误常量，称为sql.ErrNoRows，当结果为空时，它将从QueryRow()返回。这在大多数情况下需要作为特殊情况来处理。空的结果通常不被应用程序代码认为是错误的，如果不检查错误是不是这个特殊常量，那么会导致你意想不到的应用程序代码错误。

来自查询的错误被推迟到调用Scan()，然后从中返回。上面的代码可以更好地写成这样：

```
var name string
err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
if err != nil {
    if err == sql.ErrNoRows {
        // there were no rows, but otherwise no error occurred
    } else {
        log.Fatal(err)
    }
}
fmt.Println(name)
```

有人可能会问为什么一个空的结果集被认为是一个错误。空集没有什么错误。原因是QueryRow()方法需要使用这种特殊情况才能让调用者区分是否QueryRow()实际上找到一行；没有它，Scan(0不会做任何事情，你可能不会意识到你的变量毕竟没有从数据库中获取任何值。

当你使用QueryRow()时，你应该只会遇到此错误。如果你在别处遇到这个错误，你就做错了什么。

识别特定的数据库错误

像下面这样编写代码是很有诱惑力的：

```
rows, err := db.Query("SELECT someval FROM sometable")
// err contains:
// ERROR 1045 (28000): Access denied for user 'foo'@'::1' (using password: NO)
if strings.Contains(err.Error(), "Access denied") {
    // Handle the permission-denied error
}
```

这不是最好的方法。例如，字符串值可能会取决于服务器使用什么语言发送错误消息。比较错误编号以确定具体错误是啥要好得多。

但是，驱动的机制不同，因为这不是database/sql本身的一部分。在本教程重点介绍的MySql驱动中，你可以编写以下代码：

```
if driverErr, ok := err.(*mysql.MySQLError); ok { // Now the error number is accessible direc
    if driverErr.Number == 1045 {
        // Handle the permission-denied error
    }
}
```

再次，这里的MySQLError类型由此特定驱动程序提供，并且驱动程序之间的.Number字段可能不同。然而，该值是从MySql的错误消息中提取的，因此是特定于数据库的，而不是特定于驱动的。

这段代码还是很丑相对于1045，一个魔术数字是一种代码气味。一些驱动(虽然不是MySql的驱

动程序，因为这里的主题的原因)提供错误标识符的列表。例如Postgres pg驱动程序在error.go中。还有一个由VividCortex维护的MySql错误号的外部包。使用这样的列表，上面的代码写的更漂亮：

```
if driverErr, ok := err.(*mysql.MySQLError); ok {
    if driverErr.Number == mysqlerr.ER_ACCESS_DENIED_ERROR {
        // Handle the permission-denied error
    }
}
```

处理连接错误

如果与数据库的连接被丢弃，杀死或发生错误该怎么办?

当发生这种情况时，你不需要实现任何逻辑来重试失败的语句。作为database/sql连接池的一部分，处理失败的连接是内置的。如果你执行查询或其他语句，底层连接失败，则Golang将重新打开一个新的连接(或从连接池中获取另一个连接)，并重试10次。

然而，可能会产生一些意想不到的后果。当某些类型错误可能会发生其他错误条件。这也可能是驱动程序特定的。MySql驱动程序发生的一个例子是使用KILL取消不需要的语句(例如长时间运行的查询)会导致语句被重试10次。

No.8 使用空值

可以为空的字段是令人烦恼的，并导致很多丑陋的代码。如果可以，避开它们。如果没有，那么你需要使用database/sql包中的特殊类型来处理它们，或者定义你自己的类型。

有可以空的布尔值，字符串，整数和浮点数的类型。下面是你使用它们的方法：

```
for rows.Next() {
    var s sql.NullString
    err := rows.Scan(&s)
    // check err
    if s.Valid {
       // use s.String
    } else {
       // NULL value
    }
}
```

可以空的类型的限制和避免的理由的情况下你需要更有说服力的可以为空的列:

1. 没有sql.NullUint64或sql.NullYourFavoriteType。你需要为这个定义你自己的。

2. 可空性可能会非常棘手，并不是未来的证明。如果你认为某些内容不会为空，但是你错了，你的程序将会崩溃，也许很少会发生错误。

3. Golang的好处之一是为每个变量设置一个有用的默认零值。这不是空的工作方式。

如果你需要定义自己的类型来处理NULLS，则可以复制sql.NullString的设计来实现。

如果你不能避免在你的数据库中具有空值，周围有多数数据库系统支持的另一项工作是COALESCE()。像下面这样的东西可能是可以使用的，而不需要引入大量的sql.Null*类型

```
rows, err := db.Query(`
    SELECT
        name,
        COALESCE(other_field, '') as other_field
    WHERE id = ?
`, 42)

for rows.Next() {
    err := rows.Scan(&name, &otherField)
    // ..
    // If `other_field` was NULL, `otherField` is now an empty string. This works with other
}
```

No.9 使用未知列

Scan()函数要求你准确传递正确数目的目标变量。如果你不知道查询将返回什么呢?
如果你不知道查询将返回多少列，则可以使用Columns()来查询列名称列表。你可以检查此列表的长度以查看有多少列，并且可以将切片传递给具有正确数值的Scan()。列如，MySql的某些fork为SHOW PROCESSLIST命令返回不同的列，因此你必须为此准备好，否则将导致错误，这是一种方法；还有其他的方法：

```go
cols, err := rows.Columns()
if err != nil {
    // handle the error
} else {
    dest := []interface{}{ // Standard MySQL columns
        new(uint64), // id
        new(string), // host
        new(string), // user
        new(string), // db
        new(string), // command
        new(uint32), // time
        new(string), // state
        new(string), // info
    }
    if len(cols) == 11 {
        // Percona Server
    } else if len(cols) > 8 {
        // Handle this case
    }
    err = rows.Scan(dest...)
    // Work with the values in dest
}
```

如果你不知道这些列或者它们的类型，你应该使用sql.RawBytes。

```go
cols, err := rows.Columns() // Remember to check err afterwards
vals := make([]interface{}, len(cols))
for i, _ := range cols {
    vals[i] = new(sql.RawBytes)
}
for rows.Next() {
    err = rows.Scan(vals...)
    // Now you can check each element of vals for nil-ness,
    // and you can use type introspection and type assertions
```

```
    // to fetch the column into a typed variable.
}
```

No.10 连接池

database/sql包中有一个基本的连接池。没有很多的控制或检查能力，但这里有一些你可能会发现有用的知识：

• 连接池意味着在单个数据库上执行两个连续的语句可能会打开两个链接并单独执行它们。对于程序员来说，为什么它们的代码行为不当，这是相当普遍的。例如，后面跟着INSERT的LOCK TABLES可能会被阻塞，因为INSERT位于不具有表锁定的连接上。

• 连接是在需要时创建的，池中没有空闲连接。

• 默认情况下，连接数量没有限制。如果你尝试同时执行很多操作，可以创建任意数量的连接。这可能导致数据库返回错误，例如"连接太多"。

• 在Golang1.1或更新版本中，你可以使用db.SetMaxIdleConns(N)来限制池中的空闲连接数。这并不限制池的大小。

• 在Golang1.2.1或更新版本中，可以使用db.SetMaxOpenConns(N)来限制于数据库的总打开连接数。不幸的是，一个死锁bug（修复）阻止db.SetMaxOpenConns(N)在1.2中安全使用。

• 连接回收相当快。使用db.SetMaxIdleConns(N)设置大量空闲连接可以减少此流失，并有助于

保持连接以重新使用。

• 长期保持连接空闲可能会导致问题（例如在微软azure上的这个问题）。尝试db.SetMaxIdleConns(0)如果你连接超时，因为连接空闲时间太长。

No.11 惊喜，反模式和限制

虽然database/sql很简单，但一旦你习惯了它，你可能会对它支持的用例的微妙之处感到惊讶。这是Golang的核心库通用的。

资源枯竭

如本网站所述，如果你不按预期使用database/sql，你一定会为自己造成麻烦，通常是通过消耗一些资源或阻止它们被有效的重用：
• 打开和关闭数据库可能会导致资源耗尽。

• 没有读取所有行或使用rows.Close()保留来自池的连接。

• 对于不返回行的语句，使用Query()将从池中预留一个连接。

• 没有意识到预处理语句如何工作会导致大量额外的数据库活动。

巨大的uint64值

这里有一个令人吃惊的错误。如果设置了高位，就不能将大的无符号整数作为参数传递给语

句：

```
_, err := db.Exec("INSERT INTO users(id) VALUES", math.MaxUint64) // Error
```

这将抛出一个错误。如果你使用uint64值要小心，因为它们可能开始小而且无错误的工作，但会随着时间的推移而增加，并开始抛出错误。

连接状态不匹配

有些事情可以改变连接状态，这可能导致的问题有两个原因：

1. 某些连接状态，比如你是否处于事务中，应该通过Golang类型来处理。

2. 你可能假设你的查询在单个连接上运行。

例如，使用USE语句设置当前数据库对于很多人来说是一个典型的事情。但是在Golang中，它只会影响你运行的连接。除非你处于事务中，否则你认为在该连接上执行的其他语句实际上可能在从池中获取的不同的连接上运行，因此它们不会看到这些更改的影响。

此外，在更改连接后，它将返回到池，并可能会污染其他代码的状态。这就是为什么你不应该直接将BEGIN或COMMIT语句作为SQL命令发出的原因之一。

数据库特定的语法

database/sql API提供了面向行的数据库抽象，但是具体的数据库和驱动程序可能会在行为或语法上有差异，例如预处理语句占位符。

多个结果集

Golang驱动程序不以任何方式支持单个查询中的多个结果集，尽管有一个支持大容量操作（如批量复制）的功能请求似乎没有任何计划。

这意味着，除了别的以外，返回多个结果集的存储过程将无法正常工作。

调用存储过程

调用存储过程是特定于驱动程序的，但在MySql驱动程序中，目前无法完成。看来你可以调用一个简单的过程来返回一个单一的结果集，通过执行如下的操作：

```
1 | err := db.QueryRow("CALL mydb.myprocedure").Scan(&result) // Error
```

事实上这行不通。你将收到以下错误1312：PROCEDURE mydb.myprocedure无法返回给定上下文中的结果集。这是因为MySql希望将连接设置为多语句模式，即使单个结果，并且驱动程序当前没有执行此操作（尽管看到这个问题）。

多个声明支持

database/sql没有显式的拥有多个语句支持，这意味着这个行为是后端依赖的：

```
_, err := db.Exec("DELETE FROM tbl1; DELETE FROM tbl2") // Error/unpredictable result
```

服务器可以解释它想要的，它可以包括返回的错误，只执行第一个语句，或执行两者。

同样，在事务中没有办法批处理语句。事务中的每个语句必须连续执行，并且结果中的资源（如行或行）必须被扫描或关闭，以便底层连接可供下一个语句使用。这与通常不在事务中工作时的行为不同。在这种情况下，完全可以执行查询，循环遍历行，并在循环中对数据库进行查询（这将发生在一个新的连接上）：

```
rows, err := db.Query("select * from tbl1") // Uses connection 1
for rows.Next() {
    err = rows.Scan(&myvariable)
    // The following line will NOT use connection 1, which is already in-use
    db.Query("select * from tbl2 where id = ?", myvariable)
}
```

但是事务只绑定到一个连接，所以事务不可能做到这一点：

```
tx, err := db.Begin()
rows, err := tx.Query("select * from tbl1") // Uses tx's connection
for rows.Next() {
    err = rows.Scan(&myvariable)
    // ERROR! tx's connection is already busy!
```

```
    tx.Query("select * from tbl2 where id = ?", myvariable)
}
```

不过，Golang不会阻止你去尝试。因此，如果你试图在第一个释放资源并自行清理之前尝试执行另一个语句，可能会导致一个损坏的连接。这也意味着事务中的每个语句都会产生一组单独的网络往返数据库。

No.12 相关资料

以下是我们发现有帮助的一些外部信息来源。

- 官方database/sql源码可能需要vpn打开
- MySql驱动作者jmoiron关于驱动的说明
- jmoiron内置接口
- pregresql驱动作者VividCortex博客透明加密

我们希望本教程是有帮助的。如果你有任何改进意见，请在https://github.com/VividCortex/go-database-sql-tutorial.database-sql-tutorial)的ISSUE中提出。

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下

尼古拉斯河马

总资产1 (约0.12元)　共写了1.1W字　获得19个赞　共17个粉丝

关注

写下你的评论...

全部评论 3

只看作者

按时间倒序　按时间正序

前端无聊

4楼 2019.11.10 19:20

最近在学这个，不过不打算看你的文章，如果我实在搞不懂再看你的文章，我还是以文

档和自己探索为主

赞 回复

小赵营

3楼 2019.10.21 10:39

很全很详细，赞赞赞

赞 回复

z2423236821