现代程序离不开数据存储，现阶段很热的所谓大数据处理、云盘等，更是以存储为依托。有数据存储，自然需要进行数据交换，已达到数据共享等目的。

关系型数据库发展了很长一段时间，SQL/SQL-like 已经很成熟，使用也很广泛，Go 语言标准库提供了对 SQL/SQL-like 数据库的操作的标准接口，即 [database/sql](http://docs.studygolang.com/pkg/database/sql" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank) 包。

在数据交换方面，有很多成熟的协议可以使用，常用的有：JSON、XML 等，似乎 Java 社区更喜欢 XML，而目前似乎使用更多的是 JSON。在交换协议选择方面，考虑的主要这几个方面因素：性能、跨语言（通用性）、传输量等。因此，对于性能要求高的场景，会使用 protobuf、msgpack 之类的协议。由于 JSON 和 XML 使用很广泛，Go 语言提供了解析它们的标准库；同时，为了方便 Go 程序直接数据交换，Go 专门提供了 [gob](http://docs.studygolang.com/pkg/" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank) 这种交换协议。

# 1. database/sql — SQL/SQL-Like 数据库操作接口

这是 Go 提供的操作 SQL/SQL-Like 数据库的通用接口，但 Go 标准库并没有提供具体数据库的实现，需要结合第三方的驱动来使用该接口。本书使用的是 mysql 的驱动：[github.com/go-sql-driver/mysql](https://github.com/go-sql-driver/mysql" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank)。

*注：该包有一个子包：driver，它定义了一些接口供数据库驱动实现，一般业务代码中使用 database/sql 包即可，尽量避免使用 driver 这个子包。*

## 1.1. database/sql 是什么？

很明显，[database/sql](http://docs.studygolang.com/pkg/database/sql" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank) 首先是 Go 标准库提供的一个包，用于和 SQL/SQL-Like 数据库 ( 关系或类似关系数据库）通讯。它提供了和 ODBC、Perl 的 DBI、Java 的 JDBC 和 PHP 的 PDO 类似的功能。然而，它的设计却不太一样，掌握了它有利于构建健壮、高性能的基于 database 的应用。

另一方面，database/sql 提供的是抽象概念，和具体数据库无关，具体的数据库实现，有驱动来做，这样可以很方便的更换数据库。

该包提供了一些类型（概括性的），每个类型可能包括一个或多个概念。

* DB：sql.DB 类型代表了一个数据库。这点和很多其他语言不同，它并不代表一个到数据库的具体连接，而是一个能操作的数据库对象，具体的连接在内部通过连接池来管理，对外不暴露。这点是很多人容易误解的：每一次数据库操作，都产生一个 sql.DB 实例，操作完 Close。
* Results：定义了三种结果类型：sql.Rows、sql.Row 和 sql.Result，分别用于获取多个多行结果、一行结果和修改数据库影响的行数（或其返回 last insert id）。
* Statements：sql.Stmt 代表一个语句，如：DDL、DML 等。
* Transactions：sql.Tx 代表带有特定属性的一个事务。

## 1.2. sql.DB 的使用

官方文档关于 DB 的描述：

是一个数据库句柄，代表一个具有零到多个底层连接的连接池，它可以安全的被多个 goroutine 同时使用。  
sql 包会自动创建和释放连接；它也会维护一个闲置连接的连接池。如果数据库具有单连接状态的概念，该状态只有在事务中被观察时才可信。一旦调用了 BD.Begin，返回的 Tx 会绑定到单个连接。当调用事务 Tx 的 Commit 或 Rollback 后，该事务使用的连接会归还到 DB 的闲置连接池中。连接池的大小可以用 SetMaxIdleConns 方法控制。

由于 DB 并非一个实际的到数据库的连接，而且可以被多个 goroutine 并发使用，因此，程序中只需要拥有一个全局的实例即可。所以，经常见到的示例代码：

db, err := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")  
if err != nil {  
 panic(err)  
}  
defer db.Close()

实际中，defer db.Close() 可以不调用，官方文档关于 DB.Close 的说明也提到了：Close 用于关闭数据库，释放任何打开的资源。一般不会关闭 DB，因为 DB 句柄通常被多个 goroutine 共享，并长期活跃。当然，如果你确定 DB 只会被使用一次，之后不会使用了，应该调用 Close。

所以，实际的 Go 程序，应该在一个 go 文件中的 init 函数中调用 sql.Open 初始化全局的 sql.DB 对象，供程序中所有需要进行数据库操作的地方使用。

前面说过，sql.DB 并不是实际的数据库连接，因此，sql.Open 函数并没有进行数据库连接，只有在驱动未注册时才会返回 err != nil。

例如：db, err := sql.Open("mysql", "root:@tcp23(localhost233:3306)/test?charset=utf8")。虽然这里的 dsn 是错误的，但依然 err == nil，只有在实际操作数据库（查询、更新等）或调用 Ping 时才会报错。

关于 Open 函数的参数，第一个是驱动名，为了避免混淆，一般和驱动包名一致，在驱动实现中，会有类似这样的代码：

func init() {

sql.Register("mysql", &MySQLDriver{})

}

其中 mysql 即是注册的驱动名。由于注册驱动是在 init 函数中进行的，这也就是为什么采用 \_ "github.com/go-sql-driver/mysql" 这种方式引入驱动包。第二个参数是 DSN（数据源名称），这个是和具体驱动相关的，database/sql 包并没有规定，具体书写方式参见驱动文档。

### 1.2.1. 连接池的工作原理

获取 DB 对象后，连接池是空的，第一个连接在需要的时候才会创建。可以通过下面的代码验证这一点：

db, \_ := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")  
fmt.Println("please exec show processlist")  
time.Sleep(10 \* time.*Second*)  
fmt.Println("please exec show processlist again")  
db.Ping()  
time.Sleep(10 \* time.*Second*)

在 Ping 执行之前和之后，show processlist 多了一条记录，即多了一个连接，Command 列是 Sleep。

连接池的工作方式：当调用一个函数，需要访问数据库时，该函数会请求从连接池中获取一个连接，如果连接池中存在一个空闲连接，它会将该空闲连接给该函数；否则，会打开一个新的连接。当该函数结束时，该连接要么返回给连接池，要么传递给某个需要该连接的对象，知道该对象完成时，连接才会返回给连接池。相关方法的处理说明（假设 sql.DB 的对象是 db）：

* ****db.Ping()**** 会将连接立马返回给连接池。
* ****db.Exec()**** 会将连接立马返回给连接池，但是它返回的 Result 对象会引用该连接，所以，之后可能会再次被使用。
* ****db.Query()**** 会传递连接给 sql.Rows 对象，直到完全遍历了所有的行或 Rows 的 Close 方法被调用了，连接才会返回给连接池。
* ****db.QueryRow()**** 会传递连接给 sql.Row 对象，当该对象的 Scan 方法被调用时，连接会返回给连接池。
* ****db.Begin()**** 会传递连接给 sql.Tx 对象，当该对象的 Commit 或 Rollback 方法被调用时，该链接会返回给连接池。

从上面的解释可以知道，大部分时候，我们不需要关心连接不释放问题，它们会自动返回给连接池，只有 Query 方法有点特殊，后面讲解如何处理。

注意：如果某个连接有问题（broken connection)，database/sql 内部会进行[最多 2 次](https://github.com/golang/go/blob/release-branch.go1.13/src/database/sql/sql.go" \l "L1390" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank) 的重试，从连接池中获取或新开一个连接来服务，因此，你的代码中不需要重试的逻辑。

### 1.2.2.控制连接池

Go1.2.1 之前，没法控制连接池，Go1.2.1 之后，提供了两个方法来控制连接池（Go1.2 提供了控制，不过有 bug）。

* ****db.SetMaxOpenConns(n int)**** 设置连接池中最多保存打开多少个数据库连接。注意，它包括在使用的和空闲的。如果某个方法调用需要一个连接，但连接池中没有空闲的可用，且打开的连接数达到了该方法设置的最大值，该方法调用将堵塞。默认限制是 0，表示最大打开数没有限制。
* ****db.SetMaxIdleConns(n int)**** 设置连接池中能够保持的最大空闲连接的数量。[默认值是 2](http://docs.studygolang.com/src/database/sql/sql.go?s=13724:13743" \l "L501" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter07/_blank)

上面的两个设置，可以用程序实际测试。比如通过下面的代码，可以验证 MaxIdleConns 是 2：

db, \_ := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")  
  
// 去掉注释，可以看看相应的空闲连接是不是变化了  
// db.SetMaxIdleConns(3)  
  
for i := 0; i < 10; i++ {  
 go func() {  
 db.Ping()  
 }()  
}  
  
time.Sleep(20 \* time.*Second*)

通过 show processlist 命令，可以看到有两个是 Sleep 的连接。