**sqlx使用指南**

参考链接: <http://jmoiron.github.io/sqlx/>  
sqlx是一个go语言包，在内置database/sql包之上增加了很多扩展，简化数据库操作代码的书写

**资源**

如果对go语言的sql用法不熟悉，可以到下面网站学习:  
<http://go-database-sql.org/>

如果对于golang语言不熟悉，可以到下面网站学习:  
<https://blog.csdn.net/wdy_yx>  
由于database/sql接口是sqlx的子集，当前文档中所有关于database/sql的用法同样用于sqlx

**开始**

安装sqlx驱动

go get github.com/jmoiron/sqlx

本文访问sqlite数据

go get github.com/mattn/go-sqlite3

**Handle Types**

sqlx设计和database/sql使用方法是一样的。包含有4种主要的handle types:

* sqlx.DB: 和sql.DB相似，表示数据库
* sqlx.Tx: 和sql.Tx相似，表示transacion
* sqlx.Stmt: 和sql.Stmt相似，表示prepared statement.
* sqlx.NamedStmt: 表示prepared statement(支持named parameters)

所有的handler types都提供了对database/sql的兼容，意味着当用调用sqlx.DB.Query时，可以直接替换为sql.DB.Query。这就使得sqlx可以很容易的加入到已有的数据库项目中。

此外，sqlx还有两个cursor类型:

* sqlx.Rows 和sql.Rows类似，Queryx返回。
* sqlx.Row 和 sql.Row类似，QueryRowx返回。

**连接到数据库**

一个DB实例并不是一个链接，但是抽象表示了一个数据库。这就是为什么创建一个DB时并不会返回错误和panic。它内部维护了一个连接池，当需要进行连接的时候尝试连接。你可以通过Open创建一个sqlx.DB或通过NewDb从已存在的sql.DB中创建一个新的sqlx.DB

var db \*sqlx.DB

// exactly the same as the built-in

db = sqlx.Open("sqlite3",":memory:")

// from a pre-existing sql.DB; note the required driverName

db = sqlx.NewDb(sql.Open("sqlite3",":memory:"),"sqlite3")

// force a connection and test that it worked

err = db.Ping()

在一些环境下，你可能需要同时打开一个DB连接。可以调用connect,这个函数打开一个新的DB并尝试Ping。MustConnect函数在链接出错时会panic。

var err error

// open and connect at the same time:

db, err = sqlx.Connect("sqlite3", ":memory:")

// open and connect at the same time, panicing on error

db = sqlx.MustConnect("sqlite3",":memory:")

**Querying 101**

sqlx中的handle types实现了数据库查询相同的基本的操作语法。

* Exec(...) (sql.Result,error) 和database/sql相比没有改变
* Query(...) (\*sql.Rows, error) 和database/sql相比没有改变
* QueryRow(...) \*sql.Row 和database/sql相比没有改变

对内置语法的扩展

* MustExec()sql.Result - Exec, but panic or error
* Queryx(...) (\*sqlx.Rows, error) - Query, but return an sqlx.Rows
* QueryRows(...) \*sqlx.Row - QueryRow, but return an sqlx.Row

还有下面新的语法

* Get(dest interface{},...) error
* Select(dest interface{},...) error  
  下面会详细介绍这些方法的使用

**Exec**

Exec和MustExec从连接池中获取一个连接然后只想对应的query操作。对于不支持ad-hoc query execution驱动，在操作执行的背后会创建一个prepared statement。  
在结果返回前这个connection会返回到连接池中。

schema := `CREATE TABLE place (

country text,

city text NuLL,

telcode integer);`

// execte a query on the server

result, err := db.Exec(schema)

// or, you can use MustExec, which panics on error

cityState := `INSERT INTO place (country, city, telcode) VALUES (?,?)`

countryCity := `INSERT INTO place (country, city, telcode) VALUES (?,?,?)`

db.MustExec(cityState,"Hong Kong", 852)

db.MustExec(cityState, "Singapore", 65)

db.MustExec(countrycity, "South Africa", "Johannesbury", 27)

上面代码中result有两个可能的数据LastInsertd() or RowsAffected()，依赖不同的驱动  
mysql代码中，在含有auth-increment key的表中执行插入操作会得到LatInsertId(),在PostgreSQL中这个信息只有在使用RETURNING语句的row cursor中才会返回

**bindvars**

代码中？ 占位符，称为bindvars,非常重要，你可以总是使用它们来向数据库发送数据，可以用来组织SQL injection攻击。

database/sql并不会对查询语句进行任何的校验，传入什么就发送到server是什么。  
除非driver实现特定的接口，query在数据库执行之前会准备好。不同的数据库的bindvars不一样。

* MySQL使用？
* PostgreSQL使用1，2等等
* SQLite使用? 或$1
* Oracle 使用: name

其他数据库可能还不一样。你可以使用sqlx.DB.Rebind(string) string函数利用? 语法来得到一个合适在当前数据库上执行的query语句

关于binddvars常见的误解是他们用于插值。他们只用于参数化，不允许改变sql语句的合法接口。例如，下面的用法是会报错的。

// doesn't work

db.Query("SELECT \* FROM ?","mytable")

// also doesn't work

db.Query("SELECT ?,? FROM people","name","location")

**Query**

Query是database/sql中执行查询主要使用的方法，该方法返回row结果。Query返回一个sql.Rows对象和一个error对象

// fetch all places from the db

rows, err := db.Query("SELECT country,city, telcode FROM place")

// iterate over each row

for rows.Next() {

var country string

// note that city can be NULL, so we use the NullString type

var telcode int

err = rows.Scan(&country,&city,&telcode)

}

在使用的时候应该把Rows当成一个游标而不是一系列的结果。尽管数据库驱动缓存的方法不一样，  
通赤Next()迭代每次获取一列结果，对于查询结果非常巨大的情况下，可以有效的限制内存的使用，  
Scan()利用reflect把sql每一列结果映射到go语言的数据类型如string, []byte等。如果你没有遍历完全部的rows结果，  
一定要记得在把connection返回到连接池之前调用rows.Close()。

Query返回的error有可能是在server准备查询的时候发生的，也有可能是在执行查询语句的时候发的。例如可能从连接池中获取一个坏的连级（尽管数据库会尝试10次去发现或创建一个工作连接）.  
一般来说，错误主要由错误的sql语句，错误的类似匹配，错误的域名或表名等。

在大部分情况下，Rows.Scan()会把从驱动获取的数据进行拷贝，无论驱动如何使用缓存。特殊类型sql.RawBytes可以用来从驱动返回的数据中获取一个zero-copy的slice byte。当下次调用Next的时候，这个值就不在有效了，因为它指向的内存已经被驱动重写了别的数据。

Query使用的connection在所有的rows通过Next()遍历完后或者调用rows.Close()后释放。  
Queryx和Query行为很相似，不过返回一个sqlx.Rows对象，支持扩展的scan行为。

type Place struct {

Country string

City sql.NullString

TelephoneCode int `db:"telcode`

}

rows, err := db.Queryx("SELECT \* FROM place")

for rows.Next() {

var p Place

err = rows.StructScan(&p)

}

sqlx.Rowx的主要扩展就是StructScan，可以自动把查下结果扫描到对应结构体中的域(fileld)中。  
注意结构体中域(field)必须是可导出(exported)的，这样sqlx才能够写入值到结构体中。  
正如在上面代码中所示，可以利用db结构体标签来指定结构体field映射到数据库中特定的列名，或者用db.MapperFunc()来指定默认的映射。  
db默认对结构体的filed名执行strings.Lower后，和数据库的列名进行匹配。关于StructScan，SliceScan,MapScan更详细的内容请参见后面章节advanced scanning。

**QueryRow**

QueryRow从数据库server中获取一列数据。它从连接池中获取一个连接。然后执行Query,返回一个Row对象，这个对象有一个自已内部的Rows对象。

row := db.QueryRow("SELECT \* FROM place WHERE telcode=?",852)

var telcode int

err = row.Scan(&telcode)

不像Query, QueryRow只返回一个Row类型，并不返回error,如果在执行查询过程中出错，则错误通过Scan返回，如果查询结果为空，则返回sql.ErrNoRows。  
如果Scan本身出错，error同样由scan返回。

QueryRow使用的connection当result返回的时候就关闭了，也就意味着使用QueryRow的时候不能够使用sql.RawByes,因为driver使用sql.RawBytes引用内存，在connection回收后可能也会无效。

QueryRowx返回一个sqlx.Row而不是sql.Row,它实现了跟Rows相同的scan方法如上，同时还有高级的scan方法如下: (更高级的scan方法advanced scanning section)

var p Place

err := db.QueryRows("SELECT city, telcode FROM place LIMIT 1").StructScan(&p)

**Get and Select**

Get和Select是一个非常省时的扩展。它们把query和非常灵活的scan语法结合起来。为了更加清晰的介绍它们，我们先讨论下什么是scannalbe:

a value is scannable if it is not a struct, eg string,int

a value is scannable if it implements sql.Scanner

a value is scannable if it is a struct with no exported fields (eg time.Time)

Get和Select对scannable的类型使用rows.scan,对non-scannable的类型使用rows.StructScan。Get用来获取单个结果然后Scan，Select用来获取结果切片。

p := Place{}

pp := []Place{}

// this will pull the first place directly into p

err = db.Get(&p,"SELECT \* FROM place WHERE telcode > ?", 50)

// they work with regular types as well

var id int

err = db.Get(&id,"SELECT count(\*) FROM place")

// fetch at most 10 place names

var names []string

err = db.Select(&names,"SELECT name FROM place LIMIT 10")

Get和Select在执行查询后就会关闭Rows，并且在执行阶段遇到任何问题都会返回错误。由于它们内部使用的StructScan,所以下文中  
advanced scanning section讲的特征也适用于Get和Select.

Select可以提高编码小路，但是要注意Select和Queryx是有很大不同的，因为Select会把整个结果一次放入内存。如果查询结果没有限制特定的大小，那么最好使用Query/StructScan迭代方法。

**Transactions**

为了使用transactions,必须使用DB.Begin()来创建，下面的代码是错误的:

db.MustExec("BEGIN;")

db.MustExec(...)

db.MustExec("COMMIT;")

Exec和其他查询语句会向DB请求一个connection，执行完后就返回到连接池中，并不能保证每次获取的connection就是BEGIN执行时使用的那个，所以正确的做法要使用DB.Begin:

tx, err := db.BEGIN

err = tx.Exec(...)

err = tx.Commit()

DB除了Begin之外，还可以使用扩展Beginx()和MustBegin(),返回sqlx.Tx:

tx := db.MustBegin()

tx.MustExec(...)

err = tx.Commit()

sqlx.Tx拥有sqlx.DB拥有的所有的haandle extensions.  
由于transaction是一个connection状态，所以Tx对象必须绑定和控制单个connection。一个Tx会在整个生命周期中保存一个connection,然后在调用commit或Rollback()的时候释放掉。你在调用这几个函数的时候必须十分小心，否则connections会一直被占用直到被垃圾回收。  
由于在一个transaction中只能有一个connection，所以每次只能执行一条语句。在执行另外的query操作之前，cursor对象Row\*和Rows必须被Scanned或Closed。如果在数据库给你返回数据的时候你尝试向数据库发送数据，这个操作可能会中断connection。  
最后，Tx对象仅仅执行了一个BEGIN语句和绑定一个connection，它其实并没有在server上执行任何操作。而transaction真实的行为包含locking和isolation,在不同数据库上实现是不同的。

**Prepared Statements**

对于大部分的数据库来说，当一个query执行的时候，在数据库内部statement其实已经准备好了。然后你可以通过sqlx.DB.Prepare()准备statements，便于后面在别的地方使用。

stmt, err := db.Prepare(`SELECT \* FROM place WHERE telcode = ?`)

row = stmt.QueryRow(65)

tx, err := db.Gegin()

txStmt, err := tx.Prepare(`SELECT \* FROM place WHERE telcode = ?`)

row = txStmt.QueryRow(852)

Prepare实际上在数据库上执行preparation操作，所以它需要一个connection和它的connection state。  
database/sql把这部分进行了抽象，自动在新的connection上创建statement,这样开发者就能通过stmt对象在多个connection上并发执行操作。  
Preparex()返回一个sqlx.Stmt对象，包含sqlx.DB和sqlx.Tx所有的handle扩展（方法）.

sql.Tx对象含有一个Stmt()方法，从已存在的statement中返回一个特定于改transaction的statement。  
sqlx.Tx同样含有一个Stmtx()方法，从已有的sql.Stmt或sqlx.Stmt中创建一个特定于transaction的sqlx.Stmt.