Go 每日一库之 xorm

Original dj GoUpUp 2020-05-08

收录于话题
#Go 每日一库

48个 >

简介

Go 标准库提供的数据库接口 `database/sql` 比较底层，使用它来操作数据库非常繁琐，而且容易出错。因而社区开源了不少第三方库，如上一篇文章中的 `sqlc` 工具，还有各式各样的 ORM （Object Relational Mapping，对象关系映射库），如 `gorm` 和 `xorm` 。本文介绍 `xorm` 。 `xorm` 是一个简单但强大的 Go 语言 ORM 库，使用它可以大大简化我们的数据库操作。

快速使用

先安装：

```
$ go get xorm.io/xorm
```

由于需要操作具体的数据库（本文中我们使用 MySQL），需要安装对应的驱动：

```
$ go get github.com/go-sql-driver/mysql
```

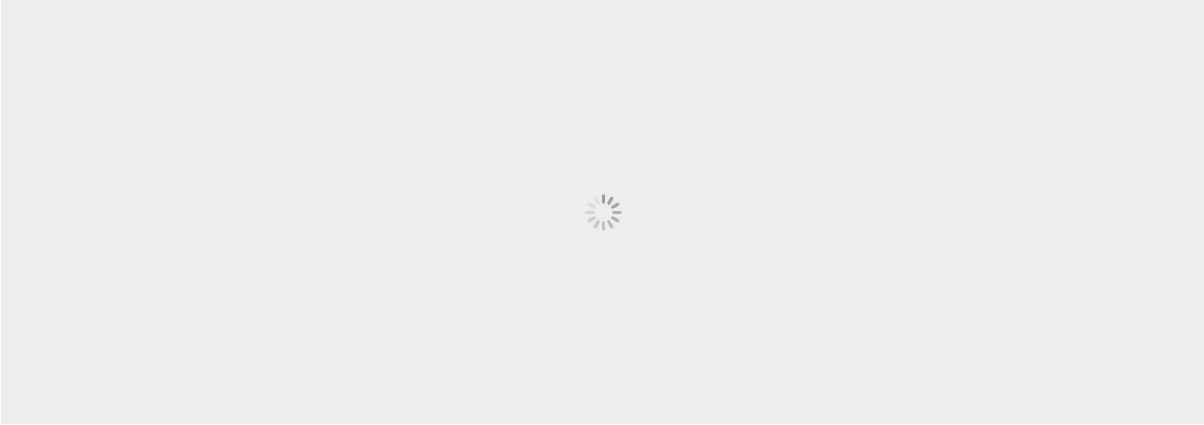
使用：

```
package main

import (
  "log"
  "time"

  _ "github.com/go-sql-driver/mysql"
  "xorm.io/xorm"
)

type User struct {
  Id      int64
  Name    string
  Salt    string
  Age     int
  Passwd  string    `xorm:"varchar(200)"`
  Created time.Time `xorm:"created"`
  Updated time.Time `xorm:"updated"`
}

func main() {
  engine, err := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  if err != nil {
    log.Fatal(err)
```

```
  }

  err = engine.Sync2(new(User))
  if err != nil {
    log.Fatal(err)
  }
}
```

使用 `xorm` 来操作数据库，首先需要使用 `xorm.NewEngine()` 创建一个引擎。该方法的参数与 `sql.Open()` 参数相同。

上面代码中，我们演示了 `xorm` 的一个非常实用的功能，将数据库中的表与对应 Go 代码中的结构体做同步。初始状态下，数据库 `test` 中没有表 `user`，调用 `Sync2()` 方法会根据 `User` 的结构自动创建一个 `user` 表。执行后，通过 `describe user` 查看表结构：

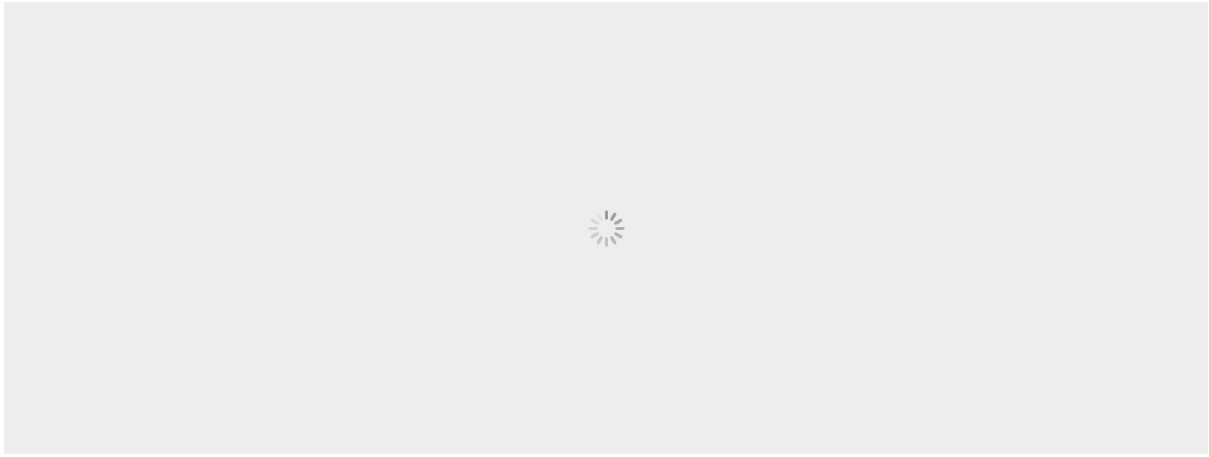


如果表 `user` 已经存在，`Sync()` 方法会对比 `User` 结构与表结构的不同，对表做相应的修改。我们给 `User` 结构添加一个 `Level` 字段：

```
type User struct {
  Id      int64
```

```
    Name    string
    Salt    string
    Age     int
    Level   int
    Passwd  string    `xorm:"varchar(200)"`
    Created time.Time `xorm:"created"`
    Updated time.Time `xorm:"updated"`
}
```

再次执行这个程序后，用`describe user`命令查看表结构：



发现表中多了一个`level`字段。

此修改只限于添加字段。删除表中已有的字段会带来比较大的风险。如果我们`User`结构的`Salt`字段删除，然后执行程序。出现下面错误：

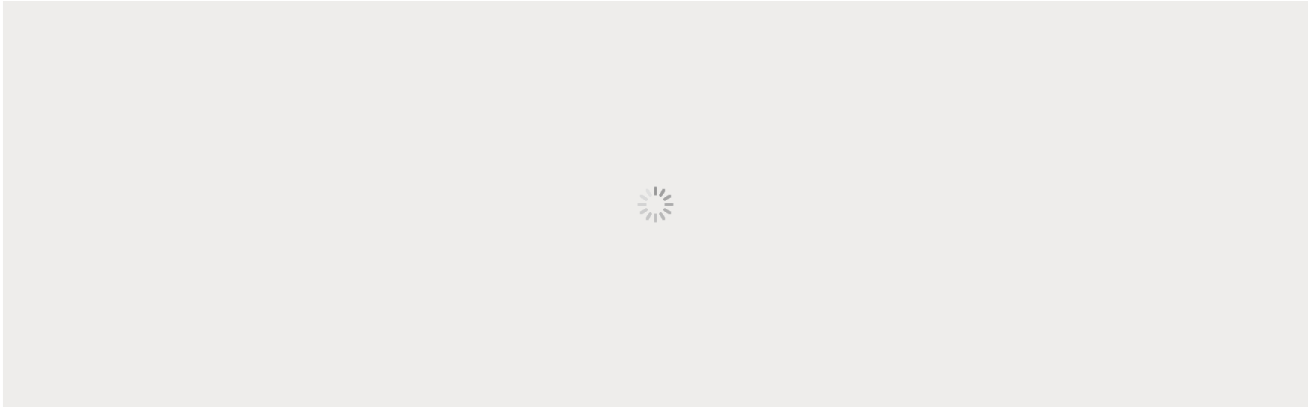
```
[xorm] [warn]  2020/05/07 22:44:38.528784 Table user has column salt but struct has not related field
```

数据库操作

查询&统计

`xorm` 提供了几个查询和统计方法， `Get/Exist/Find/Iterate/Count/Rows/Sum` 。下面逐一介绍。

为了代码演示方便，我在 `user` 表中插入了一些数据：



后面的代码为了简单起见，忽略了错误处理，实际使用中不要漏掉！

Get

`Get()` 方法用于查询单条数据，并使用返回的字段为传入的对象赋值：

```
type User struct {
  Id      int64
```

```
  Name    string
  Salt    string
  Age     int
  Passwd  string    `xorm:"varchar(200)"`
  Created time.Time `xorm:"created"`
  Updated time.Time `xorm:"updated"`
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  user1 := &User{}
  has, _ := engine.ID(1).Get(user1)
  if has {
    fmt.Printf("user1:%v\n", user1)
  }

  user2 := &User{}
  has, _ = engine.Where("name=?", "dj").Get(user2)
  if has {
    fmt.Printf("user2:%v\n", user2)
  }

  user3 := &User{Id: 5}
  has, _ = engine.Get(user3)
  if has {
    fmt.Printf("user3:%v\n", user3)
  }
```

```
    user4 := &User{Name: "pipi"}
    has, _ = engine.Get(user4)
    if has {
        fmt.Printf("user4:%v\n", user4)
    }
}
```

上面演示了 3 种使用 `Get()` 的方式：

- 使用主键：`engine.ID(1)` 查询主键（即 `id`）为 1 的用户；
- 使用条件语句：`engine.Where("name=?", "dj")` 查询 `name = "dj"` 的用户；
- 使用对象中的非空字段：`user3` 设置了 `Id` 字段为 5，`engine.Get(user3)` 查询 `id = 5` 的用户；`user4` 设置了字段 `Name` 为 `"pipi"`，`engine.Get(user4)` 查询 `name = "pipi"` 的用户。

运行程序：

```
user1:&{1 dj salt 18 12345 2020-05-08 21:12:11 +0800 CST 2020-05-08 21:12:11 +0800 CST}
user2:&{1 dj salt 18 12345 2020-05-08 21:12:11 +0800 CST 2020-05-08 21:12:11 +0800 CST}
user3:&{5 mxg salt 54 12345 2020-05-08 21:13:31 +0800 CST 2020-05-08 21:13:31 +0800 CST}
user4:&{3 pipi salt 2 12345 2020-05-08 21:13:31 +0800 CST 2020-05-08 21:13:31 +0800 CST}
```

查询条件的使用不区分调用顺序，但是必须在 `Get()` 方法之前调用。实际上后面介绍的查询&统计方法也是如此，可以在调用实际的方法前添加一些过滤条件。除此之外 `xorm` 支持只返回指定的列（`xorm.Cols()`）或忽略特定的列（`xorm.Omit()`）：

```
func main() {
```

```
    engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

    user1 := &User{}
    engine.ID(1).Cols("id", "name", "age").Get(user1)
    fmt.Printf("user1:%v\n", user1)

    user2 := &User{Name: "pipi"}
    engine.Omit("created", "updated").Get(user2)
    fmt.Printf("user2:%v\n", user2)
}
```

上面第一个查询使用 `Cols()` 方法指定只返回 `id` 、 `name` 、 `age` 这 3 列，第二个查询使用 `Omit()` 方法忽略列 `created` 和 `updated` 。

另外，为了便于排查可能出现的问题， `xorm` 提供了 `ShowSQL()` 方法设置将执行的 SQL 同时在控制台中输出：

```
func main() {
    engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
    engine.ShowSQL(true)

    user := &User{}
    engine.ID(1).Omit("created", "updated").Get(user)
    fmt.Printf("user:%v\n", user)
}
```

运行程序：

```
```

```
[xorm] [info]  2020/05/08 21:38:29.349976 [SQL] SELECT `id`, `name`, `salt`, `age`, `passwd` FROM `user` WHERE `id`=? LIMIT 1 [1] - 4.0
user:&{1 dj salt 18 12345 0001-01-01 00:00:00 +0000 UTC 0001-01-01 00:00:00 +0000 UTC}
```

由输出可以看出，执行的 SQL 语句为：

```
SELECT `id`, `name`, `salt`, `age`, `passwd` FROM `user` WHERE `id`=? LIMIT 1
```

该语句耗时 4.003 ms。在开发中这个方法非常好用！

有时候，调试信息都输出到控制台并不利于我们查询，`xorm` 可以设置日志选项，将日志输出到文件中：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  f, err := os.Create("sql.log")
  if err != nil {
    panic(err)
  }

  engine.SetLogger(log.NewSimpleLogger(f))
  engine.Logger().SetLevel(log.LOG_DEBUG)
  engine.ShowSQL(true)

  user := &User{}
  engine.ID(1).Omit("created", "updated").Get(user)
```

```
    fmt.Printf("user:%v\n", user)
}
```

这样 `xorm` 就会将调试日志输出到 `sql.log` 文件中。注意 `log.NewSimpleLogger(f)` 是 `xorm` 的子包 `xorm.io/xorm/log` 提供的简单日志封装，而非标准库 `log` 。

Exist

`Exist()` 方法查询符合条件的记录是否存在，它的返回与 `Get()` 方法一致，都是 `(bool, error)` 。不同之处在于 `Get()` 会将查询得到的字段赋值给传入的对象。相比之下 `Exist()` 方法效率要高一些。如果不需要获取数据，只要判断是否存在建议使用 `Exist()` 方法。

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  user1 := &User{}
  has, _ := engine.ID(1).Exist(user1)
  if has {
    fmt.Println("user with id=1 exist")
  } else {
    fmt.Println("user with id=1 not exist")
  }

  user2 := &User{}
  has, _ = engine.Where("name=?", "dj2").Get(user2)
  if has {
    fmt.Println("user with name=dj2 exist")
  } else {
```

```
    fmt.Println("user with name=dj2 not exist")
  }
}
```

Find

`Get()` 方法只能返回单条记录，其生成的 SQL 语句总是有 `LIMIT 1` 。 `Find()` 方法返回所有符合条件的记录。 `Find()` 需要传入对象切片的指针或 map 的指针：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  slcUsers:= make([]User, 1)
  engine.Where("age > ? and age < ?", 12, 30).Find(&slcUsers)
  fmt.Println("users whose age between [12,30]:", slcUsers)

  mapUsers := make(map[int64]User)
  engine.Where("length(name) = ?", 3).Find(&mapUsers)
  fmt.Println("users whose has name of length 3:", mapUsers)
}
```

`map` 的键为主键，所以如果表为复合主键就不能使用这种方式了。

Iterate

与 `Find()` 一样， `Iterate()` 也是找到满足条件的所有记录，只不过传入了一个回调去处理每条记录：

```
```

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  engine.Where("age > ? and age < ?", 12, 30).Iterate(&User{}, func(i int, bean interface{}) error {
    fmt.Printf("user%d:%v\n", i, bean.(*User))
    return nil
  })
}
```

如果回调返回一个非 `nil` 的错误，后面的记录就不会再处理了。

Count

`Count()` 方法统计满足条件的记录数量：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  num, _ := engine.Where("age >= ?", 50).Count(&User{})
  fmt.Printf("there are %d users whose age >= 50", num)
}
```

Rows

`Rows()` 方法与 `Iterate()` 类似，不过返回一个 `Rows` 对象由我们自己迭代，更加灵活：

```
func main() {
```

```
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  rows, _ := engine.Where("age > ? and age < ?", 12, 30).Rows(&User{})
  defer rows.Close()

  u := &User{}
  for rows.Next() {
    rows.Scan(u)

    fmt.Println(u)
  }
}
```

`Rows()` 的使用与 `database/sql` 有些类似，但是 `rows.Scan()` 方法可以传入一个对象，比 `database/sql` 更方便。

Sum

`xorm` 提供了两组求和的方法：

- `Sum/SumInt` ：求某个字段的和， `Sum` 返回 `float64` ， `SumInt` 返回 `int64` ；
- `Sums/SumsInt` ：分别求某些字段的和， `Sums` 返回 `[]float64` ， `SumsInt` 返回 `[]int64` 。

例如：

```
type Sum struct {
  Id    int64
  Money int32
  Rate  float32
```

```
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  engine.Sync2(&Sum{})

  var slice []*Sum
  for i := 0; i < 100; i++ {
    slice = append(slice, &Sum{
      Money: rand.Int31n(10000),
      Rate:  rand.Float32(),
    })
  }
  engine.Insert(&slice)

  totalMoney, _ := engine.SumInt(&Sum{}, "money")
  fmt.Println("total money:", totalMoney)

  totalRate, _ := engine.Sum(&Sum{}, "rate")
  fmt.Println("total rate:", totalRate)

  totals, _ := engine.Sums(&Sum{}, "money", "rate")
  fmt.Printf("total money:%f & total rate:%f", totals[0], totals[1])
}
```

插入

使用 `engine.Insert()` 方法，可以插入单条数据，也可以批量插入多条数据：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  user := &User{Name: "lzy", Age: 50}

  affected, _ := engine.Insert(user)
  fmt.Printf("%d records inserted, user.id:%d\n", affected, user.Id)

  users := make([]*User, 2)
  users[0] = &User{Name: "xhq", Age: 41}
  users[1] = &User{Name: "lhy", Age: 12}

  affected, _ = engine.Insert(&users)
  fmt.Printf("%d records inserted, id1:%d, id2:%d", affected, users[0].Id, users[1].Id)
}
```

插入单条记录传入一个对象指针，批量插入传入一个切片。需要注意的是，批量插入时，每个对象的 `Id` 字段不会被自动赋值，所以上面最后一行输出 `id1` 和 `id2` 均为 0。另外，一次 `Insert()` 调用可以传入多个参数，可以对应不同的表。

更新

更新通过 `engine.Update()` 实现，可以传入结构指针或 `map[string]interface{}` 。对于传入结构体指针的情况， `xorm` **只会更新非空的字段**。如果一定要更新空字段，需要使用 `Cols()` 方法显示指定更新的列。使用 `Cols()` 方法指定列后，即使字段为空也会更新：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  engine.ID(1).Update(&User{Name: "ldj"})
```

```
  engine.ID(1).Cols("name", "age").Update(&User{Name: "dj"})

  engine.Table(&User{}).ID(1).Update(map[string]interface{}{"age": 18})
}
```

由于使用 `map[string]interface{}` 类型的参数， `xorm` 无法推断表名，必须使用 `Table()` 方法指定。第一个 `Update()` 方法只会更新 `name` 字段，其他空字段不更新。第二个 `Update()` 方法会更新 `name` 和 `age` 两个字段， `age` 被更新为 0。

删除

直接调用 `engine.Delete()` 删除符合条件的记录，返回删除的条目数量：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  affected, _ := engine.Where("name = ?", "lzy").Delete(&User{})
  fmt.Printf("%d records deleted", affected)
}
```

创建时间、更新时间、软删除

如果我们为 `time.Time/int/int64` 这些类型的字段设置 `xorm:"created"` 标签，**插入数据时**，该字段会自动更新为当前时间；

如果我们为 `tiem.Time/int/int64` 这些类型的字段设置 `xorm:"updated"` 标签，**插入和更新数据时**，该字段会自动更新为当前时间；

如果我们为`time.Time`类型的字段设置了`xorm:"deleted"`标签，**删除数据时**，只是设置删除时间，并不真正删除记录。

```
type Player struct {
  Id int64
  Name string
  Age int
  CreatedAt time.Time `xorm:"created"`
  UpdatedAt time.Time `xorm:"updated"`
  DeletedAt time.Time `xorm:"deleted"`
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  engine.Sync2(&Player{})
  engine.Insert(&Player{Name:"dj", Age:18})

  p := &Player{}
  engine.Where("name = ?", "dj").Get(p)
  fmt.Println("after insert:", p)
  time.Sleep(5 * time.Second)

  engine.Table(&Player{}).ID(p.Id).Update(map[string]interface{}{"age":30})

  engine.Where("name = ?", "dj").Get(p)
  fmt.Println("after update:", p)
  time.Sleep(5 * time.Second)
```

```
  engine.ID(p.Id).Delete(&Player{})

  engine.Where("name = ?", "dj").Unscoped().Get(p)
  fmt.Println("after delete:", p)
}
```

输出：

```
after insert: &{1 dj 18 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:19 +0800 CST 0001-01-01 00:00:00 +0000 UTC}
after update: &{1 dj 30 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:24 +0800 CST 0001-01-01 00:00:00 +0000 UTC}
after delete: &{1 dj 30 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:24 +0800 CST 2020-05-08 23:09:29 +0800 CST}
```

创建时间一旦创建成功就不会再改变了，更新时间每次更新都会变化。已删除的记录必须使用 `Unscoped()` 方法查询，如果要真正 删除某条记录，也可以使用 `Unscoped()` 。

执行原始的 SQL

除了上面提供的方法外， `xorm` 还可以执行原始的 SQL 语句：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  querySql := "select * from user limit 1"
  reuslts, _ := engine.Query(querySql)
  for _, record := range reuslts {
```

```
    for key, val := range record {
      fmt.Println(key, string(val))
    }
  }

  updateSql := "update `user` set name=? where id=?"
  res, _ := engine.Exec(updateSql, "ldj", 1)
  fmt.Println(res.RowsAffected())
}
```

`Query()` 方法返回 `[]map[string][]byte`，切片中的每个元素都代表一条记录，`map` 的键对应列名，`[]byte` 为值。还有 `QueryInterface()` 方法返回 `[]map[string]interface{}`，`QueryString()` 方法返回 `[]map[string]interface{}`。

运行程序：

```
salt salt
age 18
passwd 12345
created 2020-05-08 21:12:11
updated 2020-05-08 22:44:58
id 1
name ldj
1 <nil>
```

总结

本文对 `xorm` 做了一个简单的介绍， `xorm` 的特性远不止于此。 `xorm` 可以定义结构体字段与表列名映射规则、创建索引、执行事务、导入导出 SQL 脚本等。感兴趣可自行探索。好在 `xorm` 有比较详尽的中文文档。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. xorm GitHub：https://github.com/go-xorm/xorm
2. xorm 手册：http://gobook.io/read/gitea.com/xorm/manual-zh-CN/
3. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

我

我的博客：https://darjun.github.io

欢迎关注我的微信公众号【GoUpUp】，共同学习，一起进步~

People who liked this content also liked

Go 每日一库之 reflect

GoUpUp

【割主机厂韭菜】不到20万的买车成本，如何拥有30万级享受？

车叫兽

杨幂、Lisa的“渣女”站姿火了！腰和屁股凹成90度，绝了…

GirlDaily