



下载APP



## 30 | ORM : CURD 神器 GORM 包介绍及实战

2021-08-03 孔令飞

《Go 语言项目开发实战》

课程介绍 &gt;



讲述：孔令飞

时长 17:48 大小 16.31M



你好，我是孔令飞。

在用 Go 开发项目时，我们免不了要和数据库打交道。每种语言都有优秀的 ORM 可供选择，在 Go 中也不例外，比如 [gorm](#)、[xorm](#)、[gorose](#)等。目前，GitHub 上 star 数最多的是 GORM，它也是当前 Go 项目中使用最多的 ORM。

IAM 项目也使用了 GORM。这一讲，我就来详细讲解下 GORM 的基础知识，并介绍 iam-apiserver 是如何使用 GORM，对数据进行 CURD 操作的。



### GORM 基础知识介绍

GORM 是 Go 语言的 ORM 包，功能强大，调用方便。像腾讯、华为、阿里这样的大厂，都在使用 GORM 来构建企业级的应用。GORM 有很多特性，开发中常用的核心特性如下：

功能全。使用 ORM 操作数据库的接口，GORM 都有，可以满足我们开发中对数据库调用的各类需求。

支持钩子方法。这些钩子方法可以应用在 Create、Save、Update、Delete、Find 方法中。

开发者友好，调用方便。

支持 Auto Migration。

支持关联查询。

支持多种关系数据库，例如 MySQL、Postgres、SQLite、SQLServer 等。

GORM 有两个版本，[🔗V1](#)和[🔗V2](#)。遵循用新不用旧的原则，IAM 项目使用了最新的 V2 版本。

## 通过示例学习 GORM

接下来，我们先快速看一个使用 GORM 的示例，通过该示例来学习 GORM。示例代码存放在[🔗marmotedu/gopractise-demo/gorm/main.go](#)文件中。因为代码比较长，你可以使用以下命令克隆到本地查看：

 复制代码

```
1 $ mkdir -p $GOPATH/src/github.com/marmotedu
2 $ cd $GOPATH/src/github.com/marmotedu
3 $ git clone https://github.com/marmotedu/gopractise-demo
4 $ cd gopractise-demo/gorm/
```

假设我们有一个 MySQL 数据库，连接地址和端口为 127.0.0.1:3306，用户名为 iam，密码为 iam1234。创建完 main.go 文件后，执行以下命令来运行：

 复制代码

```
1 $ go run main.go -H 127.0.0.1:3306 -u iam -p iam1234 -d test
2 2020/10/17 15:15:50 totalcount: 1
```

```
3 2020/10/17 15:15:50 code: D42, price: 100
4 2020/10/17 15:15:51 totalcount: 1
5 2020/10/17 15:15:51 code: D42, price: 200
6 2020/10/17 15:15:51 totalcount: 0
```

在企业级 Go 项目开发中，使用 GORM 库主要用来完成以下数据库操作：

连接和关闭数据库。连接数据库时，可能需要设置一些参数，比如最大连接数、最大空闲连接数、最大连接时长等。

插入表记录。可以插入一条记录，也可以批量插入记录。

更新表记录。可以更新某一个字段，也可以更新多个字段。

查看表记录。可以查看某一条记录，也可以查看符合条件的记录列表。


删除表记录。可以删除某一个记录，也可以批量删除。删除还支持永久删除和软删除。

在一些小型项目中，还会用到 GORM 的表结构自动迁移功能。

GORM 功能强大，上面的示例代码展示的是比较通用的一种操作方式。

上述代码中，首先定义了一个 GORM 模型（Models），Models 是标准的 Go struct，用来代表数据库中的一个表结构。我们可以给 Models 添加 TableName 方法，来告诉 GORM 该 Models 映射到数据库中的哪张表。Models 定义如下：

```
1 type Product struct {
2     gorm.Model
3     Code string `gorm:"column:code"`
4     Price uint `gorm:"column:price"`
5 }
6
7 // TableName maps to mysql table name.
8 func (p *Product) TableName() string {
9     return "product"
10 }
```

 复制代码

如果没有指定表名，则 GORM 使用结构体名的蛇形复数作为表名。例如：结构体名为 DockerInstance，则表名为 dockerInstances。

在之后的代码中，使用 Pflag 来解析命令行的参数，通过命令行参数指定数据库的地址、用户名、密码和数据库名。之后，使用这些参数生成建立 MySQL 连接需要的配置文件，并调用 `gorm.Open` 建立数据库连接：

[复制代码](#)

```
1 var (
2     host      = pflag.StringP("host", "H", "127.0.0.1:3306", "MySQL service hos
3     username = pflag.StringP("username", "u", "root", "Username for access to
4     password = pflag.StringP("password", "p", "root", "Password for access to
5     database = pflag.StringP("database", "d", "test", "Database name to use")
6     help      = pflag.BoolP("help", "h", false, "Print this help message")
7 )
8
9 func main() {
10     // Parse command line flags
11     pflag.CommandLine.SortFlags = false
12     pflag.Usage = func() {
13         pflag.PrintDefaults()
14     }
15     pflag.Parse()
16     if *help {
17         pflag.Usage()
18         return
19     }
20
21     dns := fmt.Sprintf(`%s:%s@tcp(%s)/%s?charset=utf8&parseTime=%t&loc=%s`,
22         *username,
23         *password,
24         *host,
25         *database,
26         true,
27         "Local")
28     db, err := gorm.Open(mysql.Open(dns), &gorm.Config{})
29     if err != nil {
30         panic("failed to connect database")
31     }
32 }
```

创建完数据库连接之后，会返回数据库实例 `db`，之后就可以调用 `db` 实例中的方法，完成数据库的 CURD 操作。具体操作如下，一共可以分为六个操作：

第一个操作，自动迁移表结构。

[复制代码](#)

```
1 // 1. Auto migration for given models
db.AutoMigrate(&Product{})
```

**我不建议你在正式的代码中自动迁移表结构。**因为变更现网数据库是一个高危操作，现网数据库字段的添加、类型变更等，都需要经过严格的评估才能实施。这里将变更隐藏在代码中，在组件发布时很难被研发人员感知到，如果组件启动，就可能会自动修改现网表结构，也可能会因此引起重大的现网事故。

GORM 的 `AutoMigrate` 方法，只对新增的字段或索引进行变更，理论上是没有风险的。在实际的 Go 项目开发中，也有很多人使用 `AutoMigrate` 方法自动同步表结构。但我更倾向于规范化、可感知的操作方式，所以我在实际开发中，都是手动变更表结构的。当然，具体使用哪种方法，你可以根据需要自行选择。

第二个操作，插入表记录。

[复制代码](#)

```
1 // 2. Insert the value into database
2 if err := db.Create(&Product{Code: "D42", Price: 100}).Error; err != nil {
3     log.Fatalf("Create error: %v", err)
4 }
5 PrintProducts(db)
```

通过 `db.Create` 方法创建了一条记录。插入记录后，通过调用 `PrintProducts` 方法打印当前表中的所有数据记录，来测试是否成功插入。

第三个操作，获取符合条件的记录。

[复制代码](#)

```
1 // 3. Find first record that match given conditions
2 product := &Product{}
3 if err := db.Where("code= ?", "D42").First(&product).Error; err != nil {
4     log.Fatalf("Get product error: %v", err)
5 }
```

`First` 方法只会返回符合条件的记录列表中的第一条，你可以使用 `First` 方法来获取某个资源的详细信息。

## 第四个操作，更新表记录。

[复制代码](#)

```
1 // 4. Update value in database, if the value doesn't have primary key, will in
2 product.Price = 200
3 if err := db.Save(product).Error; err != nil {
4     log.Fatalf("Update product error: %v", err)
5 }
6 PrintProducts(db)
```

通过 Save 方法，可以把 product 变量中所有跟数据库不一致的字段更新到数据库中。具体操作是：先获取某个资源的详细信息，再通过 product.Price = 200 这类赋值语句，对其中的一些字段重新赋值。最后，调用 Save 方法更新这些字段。你可以将这些操作看作一种更新数据库的更新模式。

## 第五个操作，删除表记录。

通过 Delete 方法删除表记录，代码如下：

[复制代码](#)

```
1 // 5. Delete value match given conditions
2 if err := db.Where("code = ?", "D42").Delete(&Product{}).Error; err != nil {
3     log.Fatalf("Delete product error: %v", err)
4 }
5 PrintProducts(db)
```

这里需要注意，因为 Product 中有 gorm.DeletedAt 字段，所以，上述删除操作不会真正把记录从数据库表中删除掉，而是通过设置数据库 product 表 deletedAt 字段为当前时间的方法来删除。

## 第六个操作，获取表记录列表。

[复制代码](#)

```
1 products := make([]*Product, 0)
2 var count int64
3 d := db.Where("code like ?", "%D%").Offset(0).Limit(2).Order("id desc").Find(&
4 if d.Error != nil {
5     log.Fatalf("List products error: %v", d.Error)
```



```
6 }
```

在 `PrintProducts` 函数中，会打印当前的所有记录，你可以根据输出，判断数据库操作是否成功。

## GORM 常用操作讲解

看完上面的示例，我想你已经初步掌握了 GORM 的使用方法。接下来，我再来给你详细介绍下 GORM 所支持的数据库操作。

## 模型定义

GORM 使用模型 (Models) 来映射一个数据库表。默认情况下，使用 ID 作为主键，使用结构体名的 `snake_cases` 作为表名，使用字段名的 `snake_case` 作为列名，并使用 `CreatedAt`、`UpdatedAt`、`DeletedAt` 字段追踪创建、更新和删除时间。

使用 GORM 的默认规则，可以减少代码量，但我更喜欢的方式是**直接指明字段名和表名**。例如，有以下模型：

[复制代码](#)

```
1 type Animal struct {
2     AnimalID int64      // 列名 `animal_id`
3     Birthday time.Time  // 列名 `birthday`
4     Age       int64      // 列名 `age`
5 }
```

上述模型对应的表名为 `animals`，列名分别为 `animal_id`、`birthday` 和 `age`。我们可以通过以下方式来重命名表名和列名，并将 `AnimalID` 设置为表的主键：

[复制代码](#)

```
1 type Animal struct {
2     AnimalID int64      `gorm:"column:animalID;primaryKey"` // 将列名设为 `animal
3     Birthday time.Time `gorm:"column:birthday"`           // 将列名设为 `birthc
4     Age       int64      `gorm:"column:age"`               // 将列名设为 `age`
5 }
6
7 func (a *Animal) TableName() string {
8     return "animal"
9 }
```

上面的代码中，通过 `primaryKey` 标签指定主键，使用 `column` 标签指定列名，通过给 Models 添加 `TableName` 方法指定表名。

数据库表通常会包含 4 个字段。

ID：自增字段，也作为主键。

CreatedAt：记录创建时间。

UpdatedAt：记录更新时间。

DeletedAt：记录删除时间（软删除时有用）。

GORM 也预定义了包含这 4 个字段的 Models，在我们定义自己的 Models 时，可以直接内嵌到结构体内，例如：

[复制代码](#)

```
1 type Animal struct {
2     gorm.Model
3     AnimalID int64      `gorm:"column:animalID"` // 将列名设为 `animalID`
4     Birthday time.Time `gorm:"column:birthday"` // 将列名设为 `birthday`
5     Age      int64      `gorm:"column:age"`      // 将列名设为 `age`
6 }
```

Models 中的字段能支持很多 GORM 标签，但如果我们不使用 GORM 自动创建表和迁移表结构的功能，很多标签我们实际上是用不到的。在开发中，用得最多的是 `column` 标签。

## 连接数据库

在进行数据库的 CURD 操作之前，我们首先需要连接数据库。你可以通过以下代码连接 MySQL 数据库：

[复制代码](#)

```
1 import (
2     "gorm.io/driver/mysql"
3     "gorm.io/gorm"
```



```
4 )
5
6 func main() {
7     // 参考 https://github.com/go-sql-driver/mysql#dsn-data-source-name 获取详情
8     dsn := "user:pass@tcp(127.0.0.1:3306)/dbname?charset=utf8mb4&parseTime=True&
9     db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
10 }
```

如果需要 GORM 正确地处理 `time.Time` 类型，在连接数据库时需要带上 `parseTime` 参数。如果要支持完整的 UTF-8 编码，可将 `charset=utf8` 更改为 `charset=utf8mb4`。

GORM 支持连接池，底层是用 `database/sql` 包来维护连接池的，连接池设置如下：

[复制代码](#)

```
1 sqlDB, err := db.DB()
2 sqlDB.SetMaxIdleConns(10)           // 设置MySQL的最大空闲连接数（推荐100）
3 sqlDB.SetMaxOpenConns(100)         // 设置MySQL的最大连接数（推荐100）
4 sqlDB.SetConnMaxLifetime(time.Hour) // 设置MySQL的空闲连接最大存活时间（推荐10s）
```

上面这些设置，也可以应用在大型后端项目中。

## 创建记录

我们可以通过 `db.Create` 方法来创建一条记录：

[复制代码](#)

```
1 type User struct {
2     gorm.Model
3     Name      string
4     Age       uint8
5     Birthday  *time.Time
6 }
7 user := User{Name: "Jinzhu", Age: 18, Birthday: time.Now()}
8 result := db.Create(&user) // 通过数据的指针来创建
```

`db.Create` 函数会返回如下 3 个值：

`user.ID`：返回插入数据的主键，这个是直接赋值给 `user` 变量。

`result.Error`：返回 `error`。

`result.RowsAffected` : 返回插入记录的条数。

当需要插入的数据量比较大时，可以批量插入，以提高插入性能：

[复制代码](#)

```
1 var users = []User{{Name: "jinzhu1"}, {Name: "jinzhu2"}, {Name: "jinzhu3"}}
2 DB.Create(&users)
3
4 for _, user := range users {
5     user.ID // 1,2,3
6 }
```

## 删除记录

我们可以通过 `Delete` 方法删除记录：

[复制代码](#)

```
1 // DELETE from users where id = 10 AND name = "jinzhu";
2 db.Where("name = ?", "jinzhu").Delete(&User{})
```

GORM 也支持根据主键进行删除，例如：

[复制代码](#)


```
1 // DELETE FROM users WHERE id = 10;
2 db.Delete(&User{}, 10)
```

不过，我更喜欢使用 `db.Where` 的方式进行删除，这种方式有两个优点。

第一个优点是删除方式更通用。使用 `db.Where` 不仅可以根据主键删除，还能够随意组合条件进行删除。

第二个优点是删除方式更显式，这意味着更易读。如果使用 `db.Delete(&User{}, 10)`，你还需要确认 `User` 的主键，如果记错了主键，还可能会引入 Bug。

此外，GORM 也支持批量删除：

 复制代码


```
1 db.Where("name in (?)", []string{"jinzhu", "colin"}).Delete(&User{})
```

GORM 支持两种删除方法：软删除和永久删除。下面我来分别介绍下。

## 1. 软删除

软删除是指执行 Delete 时，记录不会被从数据库中真正删除。GORM 会将 DeletedAt 设置为当前时间，并且不能通过正常的方式查询到该记录。如果模型包含了一个 gorm.DeletedAt 字段，GORM 在执行删除操作时，会软删除该记录。


下面的删除方法就是一个软删除：

 复制代码

```
1 // UPDATE users SET deleted_at="2013-10-29 10:23" WHERE age = 20;
2 db.Where("age = ?", 20).Delete(&User{})
3
4 // SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;
5 db.Where("age = 20").Find(&user)
```

可以看到，GORM 并没有真正把记录从数据库删除掉，而是只更新了 deleted\_at 字段。在查询时，GORM 查询条件中新增了 AND deleted\_at IS NULL 条件，所以这些被设置过 deleted\_at 字段的记录不会被查询到。对于一些比较重要的数据，我们可以通过软删除的方式删除记录，软删除可以使这些重要的数据后期能够被恢复，并且便于以后的排障。


我们可以通过下面的方式查找被软删除的记录：

 复制代码

```
1 // SELECT * FROM users WHERE age = 20;
2 db.Unscoped().Where("age = 20").Find(&users)
```

## 2. 永久删除

如果想永久删除一条记录，可以使用 Unscoped：


 复制代码

```
1 // DELETE FROM orders WHERE id=10;
2 db.Unscoped().Delete(&order)
```

或者，你也可以在模型中去掉 gorm.DeletedAt。

## 更新记录


GORM 中，最常用的更新方法如下：

 复制代码

```
1 db.First(&user)
2
3 user.Name = "jinzhu 2"
4 user.Age = 100
5 // UPDATE users SET name='jinzhu 2', age=100, birthday='2016-01-01', updated_a
6 db.Save(&user)
```


上述方法会保留所有字段，所以执行 Save 时，需要先执行 First，获取某个记录的所有列的值，然后再对需要更新的字段设置值。

还可以指定更新单个列：

 复制代码

```
1 // UPDATE users SET age=200, updated_at='2013-11-17 21:34:10' WHERE name='colin'
2 db.Model(&User{}).Where("name = ?", "colin").Update("age", 200)
```

也可以指定更新多个列：

 复制代码

```
1 // UPDATE users SET name='hello', age=18, updated_at = '2013-11-17 21:34:10' W
2 db.Model(&user).Where("name", "colin").Updates(User{Name: "hello", Age: 18, Ac
```

这里要注意，这个方法只会更新非零值的字段。

## 查询数据

GORM 支持不同的查询方法，下面我来讲解三种在开发中经常用到的查询方式，分别是检索单个记录、查询所有符合条件的记录和智能选择字段。

### 1. 检索单个记录

下面是检索单个记录的示例代码：

[复制代码](#)

```
1 // 获取第一条记录（主键升序）
2 // SELECT * FROM users ORDER BY id LIMIT 1;
3 db.First(&user)
4
5 // 获取最后一条记录（主键降序）
6 // SELECT * FROM users ORDER BY id DESC LIMIT 1;
7 db.Last(&user)
8 result := db.First(&user)
9 result.RowsAffected // 返回找到的记录数
10 result.Error        // returns error
11
12 // 检查 ErrRecordNotFound 错误
13 errors.Is(result.Error, gorm.ErrRecordNotFound)
```

如果 model 类型没有定义主键，则按第一个字段排序。

### 2. 查询所有符合条件的记录


示例代码如下：

[复制代码](#)

```
1 users := make([]*User, 0)
2
3 // SELECT * FROM users WHERE name <> 'jinzhu';
4 db.Where("name <> ?", "jinzhu").Find(&users)
```

### 3. 智能选择字段

你可以通过 Select 方法，选择特定的字段。我们可以定义一个较小的结构体来接受选定的字段：

 复制代码

```
1 type APIUser struct {  
2     ID    uint  
3     Name string  
4 }  
5  
6 // SELECT `id`, `name` FROM `users` LIMIT 10;  
7 db.Model(&User{}).Limit(10).Find(&APIUser{})
```


除了上面讲的三种常用的基本查询方法，GORM 还支持高级查询，下面我来介绍下。

## 高级查询

GORM 支持很多高级查询功能，这里我主要介绍 4 种。

### 1. 指定检索记录时的排序方式

示例代码如下：

 复制代码

```
1 // SELECT * FROM users ORDER BY age desc, name;  
2 db.Order("age desc, name").Find(&users)
```

### 2. Limit & Offset


Offset 指定从第几条记录开始查询，Limit 指定返回的最大记录数。Offset 和 Limit 值为 -1 时，消除 Offset 和 Limit 条件。另外，Limit 和 Offset 位置不同，效果也不同。

 复制代码

```
1 // SELECT * FROM users OFFSET 5 LIMIT 10;  
2 db.Limit(10).Offset(5).Find(&users)
```

### 3. Distinct


Distinct 可以从数据库记录中选择不同的值。

 复制代码

```
1 db.Distinct("name", "age").Order("name, age desc").Find(&results)
```

## 4. Count

Count 可以获取匹配的条数。

 复制代码

```
1 var count int64
2 // SELECT count(1) FROM users WHERE name = 'jinzhu'; (count)
3 db.Model(&User{}).Where("name = ?", "jinzhu").Count(&count)
```

GORM 还支持很多高级查询功能，比如内联条件、Not 条件、Or 条件、Group & Having、Joins、Group、FirstOrInit、FirstOrCreate、迭代、FindInBatches 等。因为 IAM 项目中没有用到这些高级特性，我在这里就不展开介绍了。你如果感兴趣，可以看下 [GORM 的官方文档](#)。


## 原生 SQL

GORM 支持原生查询 SQL 和执行 SQL。原生查询 SQL 用法如下：

 复制代码

```
1 type Result struct {
2     ID    int
3     Name  string
4     Age   int
5 }
6
7 var result Result
8 db.Raw("SELECT id, name, age FROM users WHERE name = ?", 3).Scan(&result)
```

原生执行 SQL 用法如下；

 复制代码



```
1 db.Exec("DROP TABLE users")
2 db.Exec("UPDATE orders SET shipped at=? WHERE id TN ?". time.Now(). Truncate(1000000000). UnixMilli())
```

## GORM 钩子

GORM 支持钩子功能，例如下面这个在插入记录前执行的钩子：

```
1 func (u *User) BeforeCreate(tx *gorm.DB) (err error) {
2     u.UUID = uuid.New()
3
4     if u.Name == "admin" {
5         return errors.New("invalid name")
6     }
7     return
8 }
```

复制代码

GORM 支持的钩子见下表：



钩子	触发时机
BeforeSave	Save前执行
AfterSave	Save后执行
BeforeCreate	插入记录前执行
AfterCreate	插入记录后执行
BeforeDelete	删除记录前执行
AfterDelete	删除记录后执行
BeforeUpdate	更新记录前执行
AfterUpdate	更新记录后执行
AfterFind	查询记录后执行

## iam-apiserver 中的 CURD 操作实战

接下来，我来介绍下 iam-apiserver 是如何使用 GORM，对数据进行 CURD 操作的。

**首先**，我们需要配置连接 MySQL 的各类参数。iam-apiserver 通过

🔗 [NewMySQLOptions](#) 函数创建了一个带有默认值的 🔗 [MySQLOptions](#) 类型的变量，将该变量传给 🔗 [NewApp](#) 函数。在 App 框架中，最终会调用 MySQLOptions 提供的 AddFlags 方法，将 MySQLOptions 提供的命令行参数添加到 Cobra 命令行中。

**接着**，在 🔗 [PrepareRun](#) 函数中，调用 🔗 [GetMySQLFactoryOr](#) 函数，初始化并获取仓库层的实例 🔗 [mysqlFactory](#)。实现了仓库层 🔗 [store.Factory](#) 接口：

[📄 复制代码](#)

```
1 type Factory interface {
2     Users() UserStore
3     Secrets() SecretStore
4     Policies() PolicyStore
5     Close() error
6 }
```

GetMySQLFactoryOr 函数采用了我们在 🔗 [11 讲](#) 中提过的单例模式，确保 iam-apiserver 进程中只有一个仓库层的实例，这样可以减少内存开支和系统的性能开销。

GetMySQLFactoryOr 函数中，使用 🔗 [github.com/marmotedu/iam/pkg/db](https://github.com/marmotedu/iam/pkg/db) 包提供的 New 函数，创建了 MySQL 实例。New 函数代码如下：

[📄 复制代码](#)

```
1 func New(opts *Options) (*gorm.DB, error) {
2     dns := fmt.Sprintf(`%s:%s@tcp(%s)/%s?charset=utf8&parseTime=%t&loc=%s`,
3         opts.Username,
4         opts.Password,
5         opts.Host,
6         opts.Database,
7         true,
8         "Local")
9
10    db, err := gorm.Open(mysql.Open(dns), &gorm.Config{
11        Logger: logger.New(opts.LogLevel),
12    })
13    if err != nil {
14        return nil, err
15    }
```

```
16
17     sqlDB, err := db.DB()
18     if err != nil {
19         return nil, err
20     }
21
22     // SetMaxOpenConns sets the maximum number of open connections to the data
23     sqlDB.SetMaxOpenConns(opts.MaxOpenConnections)
24
25     // SetConnMaxLifetime sets the maximum amount of time a connection may be
26     sqlDB.SetConnMaxLifetime(opts.MaxConnectionLifeTime)
27
28     // SetMaxIdleConns sets the maximum number of connections in the idle conn
29     sqlDB.SetMaxIdleConns(opts.MaxIdleConnections)
30
31     return db, nil
32 }
```

上述代码中，我们先创建了一个 `*gorm.DB` 类型的实例，并对该实例进行了如下设置：

通过 `SetMaxOpenConns` 方法，设置了 MySQL 的最大连接数（推荐 100）。

通过 `SetConnMaxLifetime` 方法，设置了 MySQL 的空闲连接最大存活时间（推荐 10s）。

通过 `SetMaxIdleConns` 方法，设置了 MySQL 的最大空闲连接数（推荐 100）。

`GetMySQLFactoryOr` 函数最后创建了 `datastore` 类型的变量 `mysqlFactory`，该变量是仓库层的变量。`mysqlFactory` 变量中，又包含了 `*gorm.DB` 类型的字段 `db`。

**最终**，我们通过仓库层的变量 `mysqlFactory`，调用其 `db` 字段提供的方法来完成数据库的 CURD 操作。例如，创建密钥、更新密钥、删除密钥、获取密钥详情、查询密钥列表，具体代码如下（代码位于 [secret.go](#) 文件中）：

 复制代码

```
1 // Create creates a new secret.
2 func (s *secrets) Create(ctx context.Context, secret *v1.Secret, opts metav1.C
3     return s.db.Create(&secret).Error
4 }
5
6 // Update updates an secret information by the secret identifier.
7 func (s *secrets) Update(ctx context.Context, secret *v1.Secret, opts metav1.U
8     return s.db.Save(secret).Error
9 }
```

```
10 // Delete deletes the secret by the secret identifier.
11 func (s *secrets) Delete(ctx context.Context, username, name string, opts meta
12     if opts.Unscoped {
13         s.db = s.db.Unscoped()
14     }
15
16     err := s.db.Where("username = ? and name = ?", username, name).Delete(&v1.Se
17     if err != nil && !errors.Is(err, gorm.ErrRecordNotFound) {
18         return errors.WithCode(code.ErrDatabase, err.Error())
19     }
20
21     return nil
22 }
23
24 // Get return an secret by the secret identifier.
25 func (s *secrets) Get(ctx context.Context, username, name string, opts metav1.
26     secret := &v1.Secret{}
27     err := s.db.Where("username = ? and name = ?", username, name).First(&secret)
28     if err != nil {
29         if errors.Is(err, gorm.ErrRecordNotFound) {
30             return nil, errors.WithCode(code.ErrSecretNotFound, err.Error())
31         }
32
33         return nil, errors.WithCode(code.ErrDatabase, err.Error())
34     }
35
36     return secret, nil
37 }
38
39 // List return all secrets.
40 func (s *secrets) List(ctx context.Context, username string, opts metav1.ListO
41     ret := &v1.SecretList{}
42     ol := gormutil.Unpointer(opts.Offset, opts.Limit)
43
44     if username != "" {
45         s.db = s.db.Where("username = ?", username)
46     }
47
48     selector, _ := fields.ParseSelector(opts.FieldSelector)
49     name, _ := selector.RequiresExactMatch("name")
50
51     d := s.db.Where(" name like ?", "%"+name+"%").
52         Offset(ol.Offset).
53         Limit(ol.Limit).
54         Order("id desc").
55         Find(&ret.Items).
56         Offset(-1).
57         Limit(-1).
58         Count(&ret.TotalCount)
59
60     return ret, d.Error
61 }
```

```
62 }
```

上面的代码中，`s.db` 就是 `*gorm.DB` 类型的字段。

上面的代码段执行了以下操作：

通过 `s.db.Save` 来更新数据库表的各字段；

通过 `s.db.Unscoped` 来永久性从表中删除一行记录。对于支持软删除的资源，我们还可以通过 `opts.Unscoped` 选项来控制是否永久删除记录。`true` 永久删除，`false` 软删除，默认软删除。

通过 `errors.Is(err, gorm.ErrRecordNotFound)` 来判断 GORM 返回的错误是否是没有找到记录的错误类型。

通过下面两行代码，来获取查询条件 `name` 的值：

[复制代码](#)

```
1 selector, _ := fields.ParseSelector(opts.FieldSelector)
2 name, _ := selector.RequiresExactMatch("name")
```


我们的整个调用链是：控制层 -> 业务层 -> 仓库层。这里你可能要问：**我们是如何调用到仓库层的实例 `mysqlFactory` 的呢？**

这是因为我们的控制层实例包含了业务层的实例。在创建控制层实例时，我们传入了业务层的实例：

[复制代码](#)

```
1 type UserController struct {
2     srv srvv1.Service
3 }
4
5 // NewUserController creates a user handler.
6 func NewUserController(store store.Factory) *UserController {
7     return &UserController{
8         srv: srvv1.NewService(store),
9     }
10 }
```

业务层的实例包含了仓库层的实例。在创建业务层实例时，传入了仓库层的实例：


 复制代码

```
1 type service struct {
2     store store.Factory
3 }
4
5 // NewService returns Service interface.
6 func NewService(store store.Factory) Service {
7     return &service{
8         store: store,
9     }
10 }
```

通过这种包含关系，我们在控制层可以调用业务层的实例，在业务层又可以调用仓库层的实例。这样，我们最终通过仓库层实例的 db 字段（\*gorm.DB 类型）完成数据库的 CURD 操作。

## 总结

在 Go 项目中，我们需要使用 ORM 来进行数据库的 CURD 操作。在 Go 生态中，当前最受欢迎的 ORM 是 GORM，IAM 项目也使用了 GORM。GORM 有很多功能，常用的功能有模型定义、连接数据库、创建记录、删除记录、更新记录和查询数据。这些常用功能的常见使用方式如下：

 复制代码

```
1 package main
2
3 import (
4     "fmt"
5     "log"
6
7     "github.com/spf13/pflag"
8     "gorm.io/driver/mysql"
9     "gorm.io/gorm"
10 )
11
12 type Product struct {
13     gorm.Model
14     Code string `gorm:"column:code"`
15     Price uint `gorm:"column:price"`
16 }
17
```

```
18 // TableName maps to mysql table name.
19 func (p *Product) TableName() string {
20     return "product"
21 }
22
23 var (
24     host      = pflag.StringP("host", "H", "127.0.0.1:3306", "MySQL service host
25     username = pflag.StringP("username", "u", "root", "Username for access to my
26     password = pflag.StringP("password", "p", "root", "Password for access to my
27     database = pflag.StringP("database", "d", "test", "Database name to use")
28     help      = pflag.BoolP("help", "h", false, "Print this help message")
29 )
30
31 func main() {
32     // Parse command line flags
33     pflag.CommandLine.SortFlags = false
34     pflag.Usage = func() {
35         pflag.PrintDefaults()
36     }
37     pflag.Parse()
38     if *help {
39         pflag.Usage()
40         return
41     }
42
43     dns := fmt.Sprintf(`%s:%s@tcp(%s)/%s?charset=utf8&parseTime=%t&loc=%s`,
44         *username,
45         *password,
46         *host,
47         *database,
48         true,
49         "Local")
50     db, err := gorm.Open(mysql.Open(dns), &gorm.Config{})
51     if err != nil {
52         panic("failed to connect database")
53     }
54
55     // 1. Auto migration for given models
56     db.AutoMigrate(&Product{})
57
58     // 2. Insert the value into database
59     if err := db.Create(&Product{Code: "D42", Price: 100}).Error; err != nil {
60         log.Fatalf("Create error: %v", err)
61     }
62     PrintProducts(db)
63
64     // 3. Find first record that match given conditions
65     product := &Product{}
66     if err := db.Where("code= ?", "D42").First(&product).Error; err != nil {
67         log.Fatalf("Get product error: %v", err)
68     }
69 }
```



```
70 // 4. Update value in database, if the value doesn't have primary key, will
71 product.Price = 200
72 if err := db.Save(product).Error; err != nil {
73     log.Fatalf("Update product error: %v", err)
74 }
75 PrintProducts(db)
76
77 // 5. Delete value match given conditions
78 if err := db.Where("code = ?", "D42").Delete(&Product{}).Error; err != nil {
79     log.Fatalf("Delete product error: %v", err)
80 }
81 PrintProducts(db)
82 }
83
84 // List products
85 func PrintProducts(db *gorm.DB) {
86     products := make([]*Product, 0)
87     var count int64
88     d := db.Where("code like ?", "%D%").Offset(0).Limit(2).Order("id desc").Find
89     if d.Error != nil {
90         log.Fatalf("List products error: %v", d.Error)
91     }
92
93     log.Printf("totalcount: %d", count)
94     for _, product := range products {
95         log.Printf("\tcode: %s, price: %d\n", product.Code, product.Price)
96     }
97 }
```

此外，GORM 还支持原生查询 SQL 和原生执行 SQL，可以满足更加复杂的 SQL 场景。

GORM 中，还有一个非常有用的功能是支持 Hooks。Hooks 可以在执行某个 CURD 操作前被调用。在 Hook 中，可以添加一些非常有用的功能，例如生成唯一 ID。目前，GORM 支持 BeforeXXX、AfterXXX 和 AfterFind Hook，其中 XXX 可以是 Save、Create、Delete、Update。

最后，我还介绍了 IAM 项目的 GORM 实战，具体使用方式跟总结中的示例代码大体保持一致，你可以返回文稿查看。

## 课后练习

1. GORM 支持 AutoMigrate 功能，思考下，你的生产环境是否可以使用 AutoMigrate 功能，为什么？

2. 查看 [GORM 官方文档](#)，看下如何用 GORM 实现事务回滚功能。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

👍 赞 1

🔗 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 控制流（下）：iam-apiserver 服务核心功能实现讲解

下一篇 31 | 数据流：通过 iam-authz-server 设计，看数据流服务的设计

## 更多课程推荐

# 说透区块链

拨开迷雾，还原区块链真相

赵铭

区块链服务平台资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金奖励**。

## 精选留言 (5)

💬 写留言

我不  
在线

二三闲语

2021-08-09

## 怎么配置主从，多库

展开 ▾

作者回复: 主从，是mysql实例层保证的，应用层可以不用关注。

多库，可以参照现有的实现，在Store层添加一个新的db实例

1



**shuff1e**

2021-08-04

这个方法只会更新非零值的字段。

---

如果要更新某个字段为零值，怎么办比较好。

展开 ▾

1



**友**

2021-08-03

目前用过几款orm。有beego自带的 gorm ent

展开 ▾



**那一刻**

2021-08-03

请问老师，在链接数据库New方法里，是否要调用下sqlDB.ping来确定链接是否成功呢？

作者回复: 不用哈，如果没连接成功功能测试时，你是能感觉到的，那时候可以修复下。

不过加上也可以



**pedro**

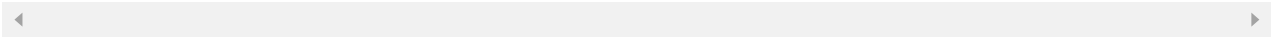
2021-08-03

GORM 支持 AutoMigrate 功能，思考下，你的生产环境是否可以使用 AutoMigrate 功能，为什么？

肯定不能啊，生产环境严格的要死，提交的SQL都要一圈的审核，怎么能直接偏移，爆炸了年终奖全都没了

展开

作者回复: 是的呀，最好别用AutoMigrate



1

