GORM实践

0 什么是ORM？为什么要用ORM?

什么是ORM

即Object-Relationl Mapping，它的作用是在关系型数据库和对象之间作一个映射，这样，我们在具体的操作数据库的时候，就不需要再去和复杂的SQL语句打交道，只要像平时操作对象一样操作它就可以了 。

ORM解决的主要问题是对象关系的映射。域模型和关系模型分别是建立在概念模型的基础上的。域模型是面向对象的，而关系模型是面向关系的。一般情况下，一个持久化类和一个表对应，类的每个实例对应表中的一条记录，类的每个属性对应表的每个字段。

ORM技术特点：

1. 提高了开发效率。由于ORM可以自动对Entity对象与数据库中的Table进行字段与属性的映射，所以我们实际可能已经不需要一个专用的、庞大的数据访问层。
2. ORM提供了对数据库的映射，不用sql直接编码，能够像操作对象一样从数据库获取数据。

ORM的缺点

- ORM的缺点是会牺牲程序的执行效率和会固定思维模式。
- 从系统结构上来看,采用ORM的系统一般都是多层系统，系统的层次多了，效率就会降低。ORM是一种完全的面向对象的做法，而面向对象的做法也会对性能产生一定的影响。

1 GORM入门指南

gorm是一个使用Go语言编写的ORM框架。它文档齐全，对开发者友好，支持主流数据库。

gorm介绍

Github GORM
中文官方网站内含十分齐全的中文文档，有了它你甚至不需要再继续向下阅读本文。

安装

```
go get -u github.com/jinzhu/gorm
```

连接MySQL

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mysql"
)
func main() {
  db, err := gorm.Open("mysql", "user:password@(localhost)/dbname?charset=utf8mb4&parseTime=True&loc=Local")
  defer db.Close()
}
```

GORM基本示例

GORM操作MySQL

使用GORM连接上面的`db1`进行创建、查询、更新、删除操作。

```go
// 1-mysql-curd.go
package main

import (
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/mysql"
)

// UserInfo 用户信息
type UserInfo struct {
    ID     uint
    Name   string
    Gender string
    Hobby  string
}

// mysql> show tables;
// +----------------+
// | Tables_in_gorm1 |
// +----------------+
// | user_infos      |
// +----------------+
// mysql> desc user_infos;
// +--------+-----------------+------+-----+---------+----------
------+
// | Field  | Type            | Null | Key | Default | Extra
|
// +--------+-----------------+------+-----+---------+----------
------+
// | id     | int(10) unsigned | NO   | PRI | NULL    | auto_incr
ement |
```

```
// | name   | varchar(255)     | YES  |     | NULL    |
|
// | gender | varchar(255)     | YES  |     | NULL    |
|
// | hobby  | varchar(255)     | YES  |     | NULL    |
|
// +--------+------------------+------+-----+---------+----------
------+
// 4 rows in set (0.04 sec)

// gorm 需要提前创建数据库gorm1
// create database gorm1;
func main() {
    fmt.Println("try open mysql connection....")
    // parseTime是查询结果是否自动解析为时间。
    // loc是MySQL的时区设置。
    db, err := gorm.Open("mysql", "root:123456@(localhost:3306)/g
orm1?charset=utf8mb4&parseTime=True&loc=Local")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    // 自动迁移
    // 若该表不存在则创建该表，若该表存在且结构体发生变化则更新表结构
    db.AutoMigrate(&UserInfo{})
    u1 := UserInfo{1, "darren", "男", "篮球"}
    u2 := UserInfo{2, "沙河娜扎", "女", "足球"}
    // 创建记录
    db.Create(&u1)
    db.Create(&u2)
    // 查询
    var u = new(UserInfo)
    db.First(u)
    fmt.Printf("%#v\n", u)
    var uu UserInfo
    db.Find(&uu, "hobby=?", "足球")
    fmt.Printf("%#v\n", uu)
    // 更新
    db.Model(&u).Update("hobby", "双色球")
    // 删除
```

```
65      db.Delete(&u)
66  }
```

2 GORM Model定义

在使用ORM工具时，通常我们需要在代码中定义模型（Models）与数据库中的数据表进行映射，在GORM中模型（Models）通常是正常定义的结构体、基本的go类型或它们的指针。 同时也支持`sql.Scanner`及`driver.Valuer`接口（interfaces）。

gorm.Model

为了方便模型定义，GORM内置了一个`gorm.Model`结构体。`gorm.Model`是一个包含了`ID`，`CreatedAt`，`UpdatedAt`，`DeletedAt`四个字段的Golang结构体。

```
// gorm.Model 定义
type Model struct {
  ID        uint `gorm:"primary_key"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt *time.Time
}
```

你可以将它嵌入到你自己的模型中：

```
// 将 `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`字段注入到`User`模型中
type User struct {
  gorm.Model
  Name string
}
```

当然你也可以完全自己定义模型：

```
// 不使用gorm.Model, 自行定义模型
type User struct {
  ID   int
  Name string
}
```

模型定义示例

```go
type User struct {
  gorm.Model
  Name         string
  Age          sql.NullInt64
  Birthday     *time.Time
  Email        string  `gorm:"type:varchar(100);unique_index"`
  Role         string  `gorm:"size:255"` // 设置字段大小为255
  MemberNumber *string `gorm:"unique;not null"` // 设置会员号（member number）唯一并且不为空
  Num          int     `gorm:"AUTO_INCREMENT"` // 设置 num 为自增类型
  Address      string  `gorm:"index:addr"` // 给address字段创建名为addr的索引
  IgnoreMe     int     `gorm:"-"` // 忽略本字段
}
```

结构体标记（tags）

使用结构体声明模型时，标记（tags）是可选项。gorm支持以下标记:

支持的结构体标记（Struct tags）

结构体标记（Tag）	描述
Column	指定列名
Type	指定列数据类型
Size	指定列大小, 默认值255
PRIMARY_KEY	将列指定为主键
UNIQUE	将列指定为唯一
DEFAULT	指定列默认值
PRECISION	指定列精度
NOT NULL	将列指定为非 NULL
AUTO_INCREMENT	指定列是否为自增类型
INDEX	创建具有或不带名称的索引, 如果多个索引同名则创建复合索引
UNIQUE_INDEX	和 `INDEX` 类似，只不过创建的是唯一索引

结构体标记（Tag）	描述
EMBEDDED	将结构设置为嵌入
EMBEDDED_PREFIX	设置嵌入结构的前缀
-	忽略此字段

关联相关标记（tags）

结构体标记（Tag）	描述
MANY2MANY	指定连接表
FOREIGNKEY	设置外键
ASSOCIATION_FOREIGNKEY	设置关联外键
POLYMORPHIC	指定多态类型
POLYMORPHIC_VALUE	指定多态值
JOINTABLE_FOREIGNKEY	指定连接表的外键
ASSOCIATION_JOINTABLE_FOREIGNKEY	指定连接表的关联外键
SAVE_ASSOCIATIONS	是否自动完成 save 的相关操作
ASSOCIATION_AUTOUPDATE	是否自动完成 update 的相关操作
ASSOCIATION_AUTOCREATE	是否自动完成 create 的相关操作
ASSOCIATION_SAVE_REFERENCE	是否自动完成引用的 save 的相关操作
PRELOAD	是否自动完成预加载的相关操作

范例

```
1 // 2-1-model.go  模型
2 package main
3
4 import (
5     "database/sql"
6     "time"
7
```

```
    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/mysql"
)

type User struct {
    gorm.Model   // 内嵌
    Name         string
    Age          sql.NullInt64
    Birthday     *time.Time
    Email        string  `gorm:"type:varchar(100);unique_index"`
    Role         string  `gorm:"size:255"`        // 设置字段大小为255
    MemberNumber *string `gorm:"unique;not null"` // 设置会员号（member number）唯一并且不为空
    Num          int     `gorm:"AUTO_INCREMENT"`  // 设置 num 为自增类型
    Address      string  `gorm:"index:addr"`      // 给address字段创建名为addr的索引
    IgnoreMe     int     `gorm:"-"`               // 忽略本字段
}

func main() {
    db, err := gorm.Open("mysql", "root:123456@tcp(localhost:3306)/gorm1?charset=utf8mb4&parseTime=True&loc=Local")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    db.AutoMigrate(&User{})
}
```

```
mysql> mysql> desc users;
+--------------+-----------------+------+-----+---------+----------+
| Field        | Type            | Null | Key | Default | Extra    |
+--------------+-----------------+------+-----+---------+----------+
```

```
----------+
| id            | int(10) unsigned | NO   | PRI | NULL    | auto_
increment |
| created_at    | datetime         | YES  |     | NULL    |
|
| updated_at    | datetime         | YES  |     | NULL    |
|
| deleted_at    | datetime         | YES  | MUL | NULL    |
|
| name          | varchar(255)     | YES  |     | NULL    |
|
| age           | bigint(20)       | YES  |     | NULL    |
|
| birthday      | datetime         | YES  |     | NULL    |
|
| email         | varchar(100)     | YES  | UNI | NULL    |
|
| role          | varchar(255)     | YES  |     | NULL    |
|
| member_number | varchar(255)     | NO   | UNI | NULL    |
|
| num           | int(11)          | YES  |     | NULL    |
|
| address       | varchar(255)     | YES  | MUL | NULL    |
|
+---------------+------------------+------+-----+---------+------
----------+
```

3 主键、表名、列名的约定

主键（Primary Key）

GORM **默认会使用名为ID的**字段作为表的主键。

```go
type User struct {
  ID   string // 名为`ID`的字段会默认作为表的主键
  Name string
}
```

```go
// 使用`AnimalID`作为主键
type Animal struct {
  AnimalID int64 `gorm:"primary_key"`
  Name     string
  Age      int64
}
```

表名（Table Name）

表名默认就是结构体名称的复数，例如：

```go
type User struct {} // 默认表名是 `users`
// 将 User 的表名设置为 `profiles`
func (User) TableName() string {
  return "profiles"
}
func (u User) TableName() string {
  if u.Role == "admin" {
    return "admin_users"
  } else {
    return "users"
  }
}
// 禁用默认表名的复数形式，如果置为 true，则 `User` 的默认表名是 `user`
db.SingularTable(true)
```

也可以通过`Table()`指定表名：

```go
// 使用User结构体创建名为`deleted_users`的表
db.Table("deleted_users").CreateTable(&User{})
var deleted_users []User
db.Table("deleted_users").Find(&deleted_users)
//// SELECT * FROM deleted_users;
db.Table("deleted_users").Where("name = ?", "jinzhu").Delete()
//// DELETE FROM deleted_users WHERE name = 'jinzhu';
```

GORM还支持更改默认表名称规则：

```go
gorm.DefaultTableNameHandler = func (db *gorm.DB, defaultTableName string) string  {
  return "prefix_" + defaultTableName;
```

```
}
```

列名（Column Name）

列名由字段名称进行下划线分割来生成

```
type User struct {
  ID        uint      // column name is `id`
  Name      string    // column name is `name`
  Birthday  time.Time // column name is `birthday`
  CreatedAt time.Time // column name is `created_at`
}
```

可以使用结构体tag指定列名：

```
type Animal struct {
  AnimalId    int64     `gorm:"column:beast_id"`         // set co
lumn name to `beast_id`
  Birthday    time.Time `gorm:"column:day_of_the_beast"` // set co
lumn name to `day_of_the_beast`
  Age         int64     `gorm:"column:age_of_the_beast"` // set co
lumn name to `age_of_the_beast`
}
```

时间戳跟踪

CreatedAt

如果模型有 `CreatedAt` 字段，该字段的值将会是初次创建记录的时间。

```
db.Create(&user) // `CreatedAt`将会是当前时间
// 可以使用`Update`方法来改变`CreateAt`的值
db.Model(&user).Update("CreatedAt", time.Now())
```

UpdatedAt

如果模型有 `UpdatedAt` 字段，该字段的值将会是每次更新记录的时间。

```
db.Save(&user) // `UpdatedAt`将会是当前时间
db.Model(&user).Update("name", "jinzhu") // `UpdatedAt`将会是当前时间
```

DeletedAt

如果模型有`DeletedAt`字段，调用`Delete`删除该记录时，将会设置`DeletedAt`字段为当前时间，而不是直接将记录从数据库中删除。

4 CRUD

CRUD通常指数据库的增删改查操作，本文详细介绍了如何使用GORM实现创建、查询、更新和删除操作。

本文中的`db`变量为`*gorm.DB`对象，例如：

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mysql"
)
func main() {
  db, err := gorm.Open("mysql", "user:password@/dbname?charset=utf8&parseTime=True&loc=Local")
  defer db.Close()

  // db.Xx
}
```

创建

创建记录

首先定义模型：

```
type User struct {
    ID           int64
    Name         string
    Age          int64
}
```

使用使用`NewRecord()`查询主键是否存在，主键为空使用`Create()`创建记录：

```
user := User{Name: "xiaomi", Age: 18}
db.NewRecord(user) // 主键为空返回`true`
db.Create(&user)   // 创建user
db.NewRecord(user) // 创建`user`后返回`false`
```

默认值

可以通过 tag 定义字段的默认值，比如：

```
type User struct {
  ID   int64
  Name string `gorm:"default:'小王子'"`
  Age  int64
}
```

注意：通过tag定义字段的默认值，在创建记录时候生成的 SQL 语句会排除没有值或值为 零值 的字段。在将记录插入到数据库后，Gorm会从数据库加载那些字段的默认值。
举个例子：

```
var user = User{Name: "", Age: 99}
db.Create(&user)
```

上面代码实际执行的SQL语句是`INSERT INTO users("age") values('99');`，排除了零值字段`Name`，而在数据库中这一条数据会使用设置的默认值`小王子`作为Name字段的值。
注意：所有字段的零值, 比如`0`，`""`,`false`或者其它`零值`，都不会保存到数据库内，但会使用他们的默认值。 如果你想避免这种情况，可以考虑使用指针或实现 `Scanner/Valuer`接口，比如：

使用指针方式实现零值存入数据库

```
// 使用指针
type User struct {
  ID   int64
  Name *string `gorm:"default:'小王子'"`
  Age  int64
}
user := User{Name: new(string), Age: 18))}
db.Create(&user)  // 此时数据库中该条记录name字段的值就是''
```

使用Scanner/Valuer接口方式实现零值存入数据库

```
// 使用 Scanner/Valuer
type User struct {
    ID int64
    Name sql.NullString `gorm:"default:'小王子'"` // sql.NullString
实现了Scanner/Valuer接口
    Age  int64
}
user := User{Name: sql.NullString{"", true}, Age:18}
```

```
db.Create(&user)  // 此时数据库中该条记录name字段的值就是''
```

扩展创建选项

例如`PostgreSQL`数据库中可以使用下面的方式实现合并插入，有则更新，无则插入。

```
// 为Instert语句添加扩展SQL选项
db.Set("gorm:insert_option", "ON CONFLICT").Create(&product)
// INSERT INTO products (name, code) VALUES ("name", "code") ON CONFLICT;
```

查询

一般查询

```
// 根据主键查询第一条记录
db.First(&user)
//// SELECT * FROM users ORDER BY id LIMIT 1;
// 随机获取一条记录
db.Take(&user)
//// SELECT * FROM users LIMIT 1;
// 根据主键查询最后一条记录
db.Last(&user)
//// SELECT * FROM users ORDER BY id DESC LIMIT 1;
// 查询所有的记录
db.Find(&users)
//// SELECT * FROM users;
// 查询指定的某条记录(仅当主键为整型时可用)
db.First(&user, 10)
//// SELECT * FROM users WHERE id = 10;
```

Where 条件

普通SQL查询

```
// Get first matched record
db.Where("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name = 'jinzhu' limit 1;
// Get all matched records
db.Where("name = ?", "jinzhu").Find(&users)
```

```
//// SELECT * FROM users WHERE name = 'jinzhu';
// <>
db.Where("name <> ?", "jinzhu").Find(&users)
//// SELECT * FROM users WHERE name <> 'jinzhu';
// IN
db.Where("name IN (?)", []string{"jinzhu", "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name in ('jinzhu','jinzhu 2');
// LIKE
db.Where("name LIKE ?", "%jin%").Find(&users)
//// SELECT * FROM users WHERE name LIKE '%jin%';
// AND
db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' AND age >= 22;
// Time
db.Where("updated_at > ?", lastWeek).Find(&users)
//// SELECT * FROM users WHERE updated_at > '2000-01-01 00:00:00';
// BETWEEN
db.Where("created_at BETWEEN ? AND ?", lastWeek, today).Find(&users)
//// SELECT * FROM users WHERE created_at BETWEEN '2000-01-01 00:00:00' AND '2000-01-08 00:00:00';
```

Struct & Map查询

```
// Struct
db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)
//// SELECT * FROM users WHERE name = "jinzhu" AND age = 20 LIMIT 1;
// Map
db.Where(map[string]interface{}{"name": "jinzhu", "age": 20}).Find(&users)
//// SELECT * FROM users WHERE name = "jinzhu" AND age = 20;
// 主键的切片
db.Where([]int64{20, 21, 22}).Find(&users)
//// SELECT * FROM users WHERE id IN (20, 21, 22);
```

提示：当通过结构体进行查询时，GORM将会只通过非零值字段查询，这意味着如果你的字段值为`0`，`''`，`false`或者其他`零值`时，将不会被用于构建查询条件，例如：

```
db.Where(&User{Name: "jinzhu", Age: 0}).Find(&users)
//// SELECT * FROM users WHERE name = "jinzhu";
```

你可以使用指针或实现 Scanner/Valuer 接口来避免这个问题.

```
// 使用指针
type User struct {
  gorm.Model
  Name string
  Age  *int
}
// 使用 Scanner/Valuer
type User struct {
  gorm.Model
  Name string
  Age  sql.NullInt64  // sql.NullInt64 实现了 Scanner/Valuer 接口
}
```

Not 条件

作用与 Where 类似的情形如下：

```
db.Not("name", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu" LIMIT 1;
// Not In
db.Not("name", []string{"jinzhu", "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name NOT IN ("jinzhu", "jinzhu
 2");
// Not In slice of primary keys
db.Not([]int64{1,2,3}).First(&user)
//// SELECT * FROM users WHERE id NOT IN (1,2,3);
db.Not([]int64{}).First(&user)
//// SELECT * FROM users;
// Plain SQL
db.Not("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE NOT(name = "jinzhu");
// Struct
db.Not(User{Name: "jinzhu"}).First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu";
```

Or条件

```
db.Where("role = ?", "admin").Or("role = ?", "super_admin").Find(&users)
//// SELECT * FROM users WHERE role = 'admin' OR role = 'super_admin';
// Struct
db.Where("name = 'jinzhu'").Or(User{Name: "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2';
// Map
db.Where("name = 'jinzhu'").Or(map[string]interface{}{"name": "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2';
```

内联条件

作用与`Where`查询类似，当内联条件与多个立即执行方法一起使用时，内联条件不会传递给后面的立即执行方法。

```
// 根据主键获取记录 (只适用于整形主键)
db.First(&user, 23)
//// SELECT * FROM users WHERE id = 23 LIMIT 1;
// 根据主键获取记录, 如果它是一个非整形主键
db.First(&user, "id = ?", "string_primary_key")
//// SELECT * FROM users WHERE id = 'string_primary_key' LIMIT 1;
// Plain SQL
db.Find(&user, "name = ?", "jinzhu")
//// SELECT * FROM users WHERE name = "jinzhu";
db.Find(&users, "name <> ? AND age > ?", "jinzhu", 20)
//// SELECT * FROM users WHERE name <> "jinzhu" AND age > 20;
// Struct
db.Find(&users, User{Age: 20})
//// SELECT * FROM users WHERE age = 20;
// Map
db.Find(&users, map[string]interface{}{"age": 20})
//// SELECT * FROM users WHERE age = 20;
```

额外查询选项

```
// 为查询 SQL 添加额外的 SQL 操作
db.Set("gorm:query_option", "FOR UPDATE").First(&user, 10)
//// SELECT * FROM users WHERE id = 10 FOR UPDATE;
```

FirstOrInit

获取匹配的第一条记录，否则根据给定的条件初始化一个新的对象 (仅支持 struct 和 map 条件)

```
// 未找到
db.FirstOrInit(&user, User{Name: "non_existing"})
//// user -> User{Name: "non_existing"}
// 找到
db.Where(User{Name: "Jinzhu"}).FirstOrInit(&user)
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
db.FirstOrInit(&user, map[string]interface{}{"name": "jinzhu"})
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Attrs

如果记录未找到，将使用参数初始化 struct.

```
// 未找到
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOr
Init(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}
db.Where(User{Name: "non_existing"}).Attrs("age", 20).FirstOrInit
(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}
// 找到
db.Where(User{Name: "Jinzhu"}).Attrs(User{Age: 30}).FirstOrInit(&
user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Assign

不管记录是否找到，都将参数赋值给 struct.

```
// 未找到
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrInit(&user)
//// user -> User{Name: "non_existing", Age: 20}
// 找到
db.Where(User{Name: "Jinzhu"}).Assign(User{Age: 30}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user -> User{Id: 111, Name: "Jinzhu", Age: 30}
```

FirstOrCreate

获取匹配的第一条记录，否则根据给定的条件创建一个新的记录（仅支持 struct 和 map 条件）

```
// 未找到
db.FirstOrCreate(&user, User{Name: "non_existing"})
//// INSERT INTO "users" (name) VALUES ("non_existing");
//// user -> User{Id: 112, Name: "non_existing"}
// 找到
db.Where(User{Name: "Jinzhu"}).FirstOrCreate(&user)
//// user -> User{Id: 111, Name: "Jinzhu"}
```

Attrs

如果记录未找到，将使用参数创建 struct 和记录.

```
// 未找到
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
//// user -> User{Id: 112, Name: "non_existing", Age: 20}
// 找到
db.Where(User{Name: "jinzhu"}).Attrs(User{Age: 30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// user -> User{Id: 111, Name: "jinzhu", Age: 20}
```

Assign

不管记录是否找到，都将参数赋值给 struct 并保存至数据库.

```
// 未找到
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
//// user -> User{Id: 112, Name: "non_existing", Age: 20}
// 找到
db.Where(User{Name: "jinzhu"}).Assign(User{Age: 30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// UPDATE users SET age=30 WHERE id = 111;
//// user -> User{Id: 111, Name: "jinzhu", Age: 30}
```

高级查询

子查询

基于 `*gorm.expr` 的子查询

```
db.Where("amount > ?", db.Table("orders").Select("AVG(amount)").Where("state = ?", "paid").SubQuery()).Find(&orders)
// SELECT * FROM "orders"  WHERE "orders"."deleted_at" IS NULL AND (amount > (SELECT AVG(amount) FROM "orders"  WHERE (state = 'paid')));
```

选择字段

Select，指定你想从数据库中检索出的字段，默认会选择全部字段。

```
db.Select("name, age").Find(&users)
//// SELECT name, age FROM users;
db.Select([]string{"name", "age"}).Find(&users)
//// SELECT name, age FROM users;
db.Table("users").Select("COALESCE(age,?)", 42).Rows()
//// SELECT COALESCE(age,'42') FROM users;
```

排序

Order，指定从数据库中检索出记录的顺序。设置第二个参数 reorder 为 `true` ，可以覆盖前面定义的排序条件。

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

```
// 多字段排序
db.Order("age desc").Order("name").Find(&users)
//// SELECT * FROM users ORDER BY age desc, name;
// 覆盖排序
db.Order("age desc").Find(&users1).Order("age", true).Find(&users2
)
//// SELECT * FROM users ORDER BY age desc; (users1)
//// SELECT * FROM users ORDER BY age; (users2)
```

数量

Limit，指定从数据库检索出的最大记录数。

```
db.Limit(3).Find(&users)
//// SELECT * FROM users LIMIT 3;
// -1 取消 Limit 条件
db.Limit(10).Find(&users1).Limit(-1).Find(&users2)
//// SELECT * FROM users LIMIT 10; (users1)
//// SELECT * FROM users; (users2)
```

偏移

Offset，指定开始返回记录前要跳过的记录数。

```
db.Offset(3).Find(&users)
//// SELECT * FROM users OFFSET 3;
// -1 取消 Offset 条件
db.Offset(10).Find(&users1).Offset(-1).Find(&users2)
//// SELECT * FROM users OFFSET 10; (users1)
//// SELECT * FROM users; (users2)
```

总数

Count，该 model 能获取的记录总数。

```
db.Where("name = ?", "jinzhu").Or("name = ?", "jinzhu 2").Find(&us
ers).Count(&count)
//// SELECT * from USERS WHERE name = 'jinzhu' OR name = 'jinzhu
 2'; (users)
//// SELECT count(*) FROM users WHERE name = 'jinzhu' OR name = 'j
inzhu 2'; (count)
db.Model(&User{}).Where("name = ?", "jinzhu").Count(&count)
//// SELECT count(*) FROM users WHERE name = 'jinzhu'; (count)
```

```
db.Table("deleted_users").Count(&count)
//// SELECT count(*) FROM deleted_users;
db.Table("deleted_users").Select("count(distinct(name))").Count(&count)
//// SELECT count( distinct(name) ) FROM deleted_users; (count)
```

注意 `Count` 必须是链式查询的最后一个操作，因为它会覆盖前面的 `SELECT`，但如果里面使用了 `count` 时不会覆盖

Group & Having

```
rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) as total").Group("date(created_at)").Rows()
for rows.Next() {
  ...
}
// 使用Scan将多条结果扫描进事先准备好的结构体切片中
type Result struct {
    Date time.Time
    Total int
}
var rets []Result
db.Table("users").Select("date(created_at) as date, sum(age) as total").Group("date(created_at)").Scan(&rets)
rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) as total").Group("date(created_at)").Having("sum(amount) > ?", 100).Rows()
for rows.Next() {
  ...
}
type Result struct {
  Date  time.Time
  Total int64
}
db.Table("orders").Select("date(created_at) as date, sum(amount) as total").Group("date(created_at)").Having("sum(amount) > ?", 100).Scan(&results)
```

连接

Joins，指定连接条件

```
rows, err := db.Table("users").Select("users.name, emails.email").Joins("left join emails on emails.user_id = users.id").Rows()
for rows.Next() {
  ...
}
db.Table("users").Select("users.name, emails.email").Joins("left join emails on emails.user_id = users.id").Scan(&results)
// 多连接及参数
db.Joins("JOIN emails ON emails.user_id = users.id AND emails.email = ?", "jinzhu@example.org").Joins("JOIN credit_cards ON credit_cards.user_id = users.id").Where("credit_cards.number = ?", "411111111111").Find(&user)
```

Pluck

Pluck，查询 model 中的一个列作为切片，如果您想要查询多个列，您应该使用 `Scan`

```
var ages []int64
db.Find(&users).Pluck("age", &ages)
var names []string
db.Model(&User{}).Pluck("name", &names)
db.Table("deleted_users").Pluck("name", &names)
// 想查询多个字段？ 这样做：
db.Select("name, age").Find(&users)
```

扫描

Scan，扫描结果至一个 struct.

```
type Result struct {
  Name string
  Age  int
}
var result Result
db.Table("users").Select("name, age").Where("name = ?", "Antonio").Scan(&result)
var results []Result
db.Table("users").Select("name, age").Where("id > ?", 0).Scan(&results)
// 原生 SQL
db.Raw("SELECT name, age FROM users WHERE name = ?", "Antonio").S
```

```
  can(&result)
```

链式操作相关

链式操作

Method Chaining，Gorm 实现了链式操作接口，所以你可以把代码写成这样：

```
// 创建一个查询
tx := db.Where("name = ?", "jinzhu")
// 添加更多条件
if someCondition {
  tx = tx.Where("age = ?", 20)
} else {
  tx = tx.Where("age = ?", 30)
}
if yetAnotherCondition {
  tx = tx.Where("active = ?", 1)
}
```

在调用立即执行方法前不会生成`Query`语句，借助这个特性你可以创建一个函数来处理一些通用逻辑。

立即执行方法

Immediate methods ， 立即执行方法是指那些会立即生成`SQL`语句并发送到数据库的方法，他们一般是`CRUD`方法，比如：

`Create`，`First`，`Find`，`Take`，`Save`，`UpdateXXX`，`Delete`，`Scan`，`Row`，`Rows`...

这有一个基于上面链式方法代码的立即执行方法的例子：

```
tx.Find(&user)
```

生成的SQL语句如下：

```
SELECT * FROM users where name = 'jinzhu' AND age = 30 AND active
 = 1;
```

范围

`Scopes`，Scope是建立在链式操作的基础之上的。

基于它，你可以抽取一些通用逻辑，写出更多可重用的函数库。

```
func AmountGreaterThan1000(db *gorm.DB) *gorm.DB {
  return db.Where("amount > ?", 1000)
```

```
}
func PaidWithCreditCard(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode_sign = ?", "C")
}
func PaidWithCod(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode_sign = ?", "C")
}
func OrderStatus(status []string) func (db *gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    return db.Scopes(AmountGreaterThan1000).Where("status IN (?)", status)
  }
}
db.Scopes(AmountGreaterThan1000, PaidWithCreditCard).Find(&orders)
// 查找所有金额大于 1000 的信用卡订单
db.Scopes(AmountGreaterThan1000, PaidWithCod).Find(&orders)
// 查找所有金额大于 1000 的 COD 订单
db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid", "shipped"})).Find(&orders)
// 查找所有金额大于 1000 且已付款或者已发货的订单
```

多个立即执行方法

Multiple Immediate Methods，在 GORM 中使用多个立即执行方法时，后一个立即执行方法会复用前一个**立即执行方法**的条件（不包括内联条件）。

```
db.Where("name LIKE ?", "jinzhu%").Find(&users, "id IN (?)", []int{1, 2, 3}).Count(&count)
```

生成的 Sql

```
SELECT * FROM users WHERE name LIKE 'jinzhu%' AND id IN (1, 2, 3)
SELECT count(*) FROM users WHERE name LIKE 'jinzhu%'
```

更新

更新所有字段

`Save()`默认会更新该对象的所有字段，即使你没有赋值。

```
db.First(&user)
```

```
user.Name = "小米"
user.Age = 99
db.Save(&user)
//// UPDATE `users` SET `created_at` = '2020-02-16 12:52:20', `updated_at` = '2020-02-16 12:54:55', `deleted_at` = NULL, `name` = '小米', `age` = 99, `active` = true WHERE `users`.`deleted_at` IS NULL AND `users`.`id` = 1
```

更新修改字段

如果你只希望更新指定字段，可以使用`Update`或者`Updates`

```
// 更新单个属性，如果它有变化
db.Model(&user).Update("name", "hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111;
// 根据给定的条件更新单个属性
db.Model(&user).Where("active = ?", true).Update("name", "hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111 AND active=true;
// 使用 map 更新多个属性，只会更新其中有变化的属性
db.Model(&user).Updates(map[string]interface{}{"name": "hello", "age": 18, "active": false})
//// UPDATE users SET name='hello', age=18, active=false, updated_at='2013-11-17 21:34:10' WHERE id=111;
// 使用 struct 更新多个属性，只会更新其中有变化且为非零值的字段
db.Model(&user).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18, updated_at = '2013-11-17 21:34:10' WHERE id = 111;
// 警告：当使用 struct 更新时，GORM只会更新那些非零值的字段
// 对于下面的操作，不会发生任何更新，"", 0, false 都是其类型的零值
db.Model(&user).Updates(User{Name: "", Age: 0, Active: false})
```

更新选定字段

如果你想更新或忽略某些字段，你可以使用 `Select`，`Omit`

```
db.Model(&user).Select("name").Updates(map[string]interface{}{"name": "hello", "age": 18, "active": false})
//// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111;
```

```
db.Model(&user).Omit("name").Updates(map[string]interface{}{"name"
: "hello", "age": 18, "active": false})
//// UPDATE users SET age=18, active=false, updated_at='2013-11-17
21:34:10' WHERE id=111;
```

无Hooks更新

上面的更新操作会自动运行 model 的 `BeforeUpdate`，`AfterUpdate` 方法，更新 `UpdatedAt` 时间戳，在更新时保存其 `Associations`，如果你不想调用这些方法，你可以使用 `UpdateColumn`，`UpdateColumns`

```
// 更新单个属性，类似于 `Update`
db.Model(&user).UpdateColumn("name", "hello")
//// UPDATE users SET name='hello' WHERE id = 111;
// 更新多个属性，类似于 `Updates`
db.Model(&user).UpdateColumns(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18 WHERE id = 111;
```

批量更新

批量更新时`Hooks（钩子函数）`不会运行。

```
db.Table("users").Where("id IN (?)", []int{10, 11}).Updates(map[st
ring]interface{}{"name": "hello", "age": 18})
//// UPDATE users SET name='hello', age=18 WHERE id IN (10, 11);
// 使用 struct 更新时，只会更新非零值字段，若想更新所有字段，请使用map[strin
g]interface{}
db.Model(User{}).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18;
// 使用 `RowsAffected` 获取更新记录总数
db.Model(User{}).Updates(User{Name: "hello", Age: 18}).RowsAffecte
d
```

使用SQL表达式更新

先查询表中的第一条数据保存至user变量。

```
var user User
db.First(&user)
```

```
db.Model(&user).Update("age", gorm.Expr("age * ? + ?", 2, 100))
```

```
//// UPDATE `users` SET `age` = age * 2 + 100, `updated_at` = '2020-02-16 13:10:20'  WHERE `users`.`id` = 1;
db.Model(&user).Updates(map[string]interface{}{"age": gorm.Expr("age * ? + ?", 2, 100)})
//// UPDATE "users" SET "age" = age * '2' + '100', "updated_at" = '2020-02-16 13:05:51' WHERE `users`.`id` = 1;
db.Model(&user).UpdateColumn("age", gorm.Expr("age - ?", 1))
//// UPDATE "users" SET "age" = age - 1 WHERE "id" = '1';
db.Model(&user).Where("age > 10").UpdateColumn("age", gorm.Expr("age - ?", 1))
//// UPDATE "users" SET "age" = age - 1 WHERE "id" = '1' AND quantity > 10;
```

修改Hooks中的值

如果你想修改 `BeforeUpdate`, `BeforeSave` 等 Hooks 中更新的值，你可以使用 `scope.SetColumn`, 例如：

```
func (user *User) BeforeSave(scope *gorm.Scope) (err error) {
  if pw, err := bcrypt.GenerateFromPassword(user.Password, 0); err == nil {
    scope.SetColumn("EncryptedPassword", pw)
  }
}
```

其它更新选项

```
// 为 update SQL 添加其它的 SQL
db.Model(&user).Set("gorm:update_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Update("name", "hello")
//// UPDATE users SET name='hello', updated_at = '2013-11-17 21:34:10' WHERE id=111 OPTION (OPTIMIZE FOR UNKNOWN);
```

删除

删除记录

警告 删除记录时，请确保主键字段有值，GORM 会通过主键去删除记录，如果主键为空，GORM 会删除该 model 的所有记录。

```
// 删除现有记录
```

```
db.Delete(&email)
//// DELETE from emails where id=10;
// 为删除 SQL 添加额外的 SQL 操作
db.Set("gorm:delete_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Delete(&email)
//// DELETE from emails where id=10 OPTION (OPTIMIZE FOR UNKNOWN);
```

批量删除

删除全部匹配的记录

```
db.Where("email LIKE ?", "%jinzhu%").Delete(Email{})
//// DELETE from emails where email LIKE "%jinzhu%";
db.Delete(Email{}, "email LIKE ?", "%jinzhu%")
//// DELETE from emails where email LIKE "%jinzhu%";
```

软删除

如果一个 model 有 `DeletedAt` 字段，他将自动获得软删除的功能！ 当调用 `Delete` 方法时， 记录不会真正的从数据库中被删除， 只会将`DeletedAt` 字段的值会被设置为当前时间

```
db.Delete(&user)
//// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE id = 111;
// 批量删除
db.Where("age = ?", 20).Delete(&User{})
//// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE age = 20;
// 查询记录时会忽略被软删除的记录
db.Where("age = 20").Find(&user)
//// SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;
// Unscoped 方法可以查询被软删除的记录
db.Unscoped().Where("age = 20").Find(&users)
//// SELECT * FROM users WHERE age = 20;
```

物理删除

```
// Unscoped 方法可以物理删除记录
db.Unscoped().Delete(&order)
//// DELETE FROM orders WHERE id=10;
```

5 gorm-错误处理、事务、SQL构建、通用数据库接口、连接池、复合主键、日志

5.1. 错误处理

执行任何操作后，如果发生任何错误，GORM将其设置为`*DB`的`Error`字段

```
if err := db.Where("name = ?", "jinzhu").First(&user).Error; err
 != nil {
    // 错误处理...
}
// 如果有多个错误发生，用`GetErrors`获取所有的错误，它返回`[]error`
db.First(&user).Limit(10).Find(&users).GetErrors()
// 检查是否返回RecordNotFound错误
db.Where("name = ?", "hello world").First(&user).RecordNotFound()
if db.Model(&user).Related(&credit_card).RecordNotFound() {
    // 没有信用卡被发现处理...
}
```

5.2. 事务

要在事务中执行一组操作，一般流程如下。

```
// 开始事务
tx := db.Begin()
// 在事务中做一些数据库操作（从这一点使用'tx'，而不是'db'）
tx.Create(...)
// ...
// 发生错误时回滚事务
tx.Rollback()
// 或提交事务
tx.Commit()
```

5.2.1. 一个具体的例子

```
func CreateAnimals(db *gorm.DB) err {
  tx := db.Begin()
```

```
  // 注意，一旦你在一个事务中，使用tx作为数据库句柄
  if err := tx.Create(&Animal{Name: "Giraffe"}).Error; err != nil {
     tx.Rollback()
     return err
  }
  if err := tx.Create(&Animal{Name: "Lion"}).Error; err != nil {
     tx.Rollback()
     return err
  }
  tx.Commit()
  return nil
}
```

5.3. SQL构建

5.3.1. 执行原生SQL

```
db.Exec("DROP TABLE users;")
db.Exec("UPDATE orders SET shipped_at=? WHERE id IN (?)", time.Now, []int64{11,22,33})
// Scan
type Result struct {
    Name string
    Age  int
}
var result Result
db.Raw("SELECT name, age FROM users WHERE name = ?", 3).Scan(&result)
```

5.3.2. sql.Row & sql.Rows

获取查询结果为`*sql.Row`或`*sql.Rows`

```
row := db.Table("users").Where("name = ?", "jinzhu").Select("name, age").Row() // (*sql.Row)
row.Scan(&name, &age)
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows() // (*sql.Rows, error)
defer rows.Close()
```

```
for rows.Next() {
    ...
    rows.Scan(&name, &age, &email)
    ...
}
// Raw SQL
rows, err := db.Raw("select name, age, email from users where name = ?", "jinzhu").Rows() // (*sql.Rows, error)
defer rows.Close()
for rows.Next() {
    ...
    rows.Scan(&name, &age, &email)
    ...
}
```

5.3.3. 迭代中使用sql.Rows的Scan

```
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows() // (*sql.Rows, error)
defer rows.Close()
for rows.Next() {
  var user User
  db.ScanRows(rows, &user)
  // do something
}
```

5.4. 通用数据库接口sql.DB

从`*gorm.DB`连接获取通用数据库接口*sql.DB

```
// 获取通用数据库对象`*sql.DB`以使用其函数
db.DB()
// Ping
db.DB().Ping()
```

5.4.1. 连接池

```
db.DB().SetMaxIdleConns(10)
db.DB().SetMaxOpenConns(100)
```

5.5. 复合主键

将多个字段设置为主键以启用复合主键

```
type Product struct {
    ID           string `gorm:"primary_key"`
    LanguageCode string `gorm:"primary_key"`
}
```

5.6. 日志

Gorm有内置的日志记录器支持，默认情况下，它会打印发生的错误

```
// 启用Logger，显示详细日志
db.LogMode(true)
// 禁用日志记录器，不显示任何日志
db.LogMode(false)
// 调试单个操作，显示此操作的详细日志
db.Debug().Where("name = ?", "jinzhu").First(&User{})
```

5.6.1. 自定义日志

参考GORM的默认记录器如何自定义它https://github.com/jinzhu/gorm/blob/master/logger.go

```
db.SetLogger(gorm.Logger{revel.TRACE})
db.SetLogger(log.New(os.Stdout, "\r\n", 0))
```