GORM

Gorm 连接数据库实例

什么是 ORM

- O：object 对象
- R ： relationship 关系
- M：map 映射

即对象和关系型数据库之间的映射

优点：提升开发效率

缺点：1. 牺牲性能

2.不够灵活

3.弱化 sql 能力

Golang 数据库连接以及 CURD 测试

```go
package test

import (
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
    "log"
    "os"
    "testing"
)

type UserInfo struct { // 数据库的表名为 user_infos
    ID    int
    Name  string
    Hobby string
}

const (
    dbDriver = "mysql"
    dbSource = "root:123456@tcp(127.0.0.1:3306)/blog?charset=utf8mb4&parseTime=True&loc=Local"
)

var DB *gorm.DB

// TestMain:Golang 约定 TestMain 函数是所有单元测试的入口
func TestMain(m *testing.M) {
    db, err := gorm.Open(mysql.Open(dbSource), &gorm.Config{})
    if err != nil {
        log.Fatal("can not connect db:", err)
    }
    DB = db
    os.Exit(m.Run())
}

// Go中的每个单元测试函数都必须以 Test 开头，并且以 testing.T 作为输入参数。

// TestAddUser 新增用户
func TestAddUser(t *testing.T) {
    u1 := UserInfo{2, "max", "run"}
    DB.Create(u1)
}

// TestUpdateUser 更新用户
func TestUpdateUser(t *testing.T) {
    u1 := UserInfo{2, "max2", "run"}
```

```
    DB.Updates(u1)
}

// TestDeleteUser 删除用户
func TestDeleteUser(t *testing.T) {
    DB.Delete(UserInfo{2, "max2", "run"})
}

// TestGetUser 获取用户
func TestGetUser(t *testing.T) {
    DB.Find(UserInfo{2, "max", "run"})
}
```

Note:

> TestMain:Golang 约定 TestMain 函数是所有单元测试的入口
>
> Go中的每个单元测试函数都必须以 Test 开头，并且以 testing.T 作为输入参数

Gorm 模型定义

在使用 ORM 工具时，通常我们需要在代码中定义 Model 与数据库中的数据表进行映射，在 GORM 中模型通常是**正常定义的结构体、基本的 Go 类型或它们的指针（如果结构体比较大时，传递指针会比较节约性能）**。

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` 四个字段的结构体

gorm.Model

```go
// gorm.Model
type Model struct {
    ID        uint `gorm:"primarykey"`
    CreatedAt time.Time
    UpdatedAt time.Time
    DeletedAt DeletedAt `gorm:"index"`
}
// 嵌入其他结构体
//  type User struct {
//    gorm.Model
//  }
```

这个模型中已经定义了主键，你可以选择使用或不使用，使用时可以直接嵌入其他结构体中。如果不使用这个模型，需要自己通过添加结构体 tag: `gorm:"primarykey"` 来指定主键。

模型定义的默认值

主键默认值：GORM 会默认使用结构体中的 `ID` 作为主键，除非你特意声明了其他字段

表名默认值：

1. 表名默认为结构体名称的复数。
2. 也可以通过 `tableName()` 自定义表名

```go
type UserInfo struct { // 数据库的表名为 user_infos
    ID int
    Name  string
    Hobby string
}
// 将表名 userinfos 改为 users
(UserInfo)TableName()string {
    return "users"
    }
```

3. 还可以通过 `db.Table()` 指定结构体的表名：

```Go
type User struct{
    ID int
    Name  string
    Hobby string
}
// 使用 User 结构体创建一个 userTabel 表
db.Table("userTabel").CreateTable(&User{})
```

4. 可以通过 `DefaultTableNameHandler` 修改表名创建时的默认规则。

```Go
gorm.DefaultTableNameHandler = func (db *gorm.DB,defaultTableName string)string {
    return "SMS_"  + defaultTableName; // 注意，这个方法只能作用与你使用 GORM 默认的表名时才会生效
}
```

列名默认值：

1. 如果你的字段名是由两个单词组成的，GORM 会自动使用下划线分隔。

比如：`CreatedAt ===> created_at`

2. 也可以使用 tag 来指定列名

```Go
Age `gorm:"column:user_age" default:18`
```

Gorm 增删改查

在执行操作前加入 `debug()` 函数可以打印 sql 语句。

增加：

创建记录：

```go
u1 := table.UserInfo{2, "max", "run"}
DB.Create(u1)
```

数据库字段插入时可以指定一个默认值比如：

```go
type User struct {
    Name string `gorm:"default:'max'"`
}
```

如果没有指定 `Name` 的值，也就是该字段的值为空。GORM 会忽略该字段，但是如果设定了默认值，就会将空值改为默认值。

如果一定要将某个字段的值置为空，有两种方法：

可以使用指针类型的数据来实现：

```go
type User struct {
    Name *string `gorm:"default:'max'"`
}
// 在创建时：
u := User{Name:new(string),age:19}
```

还可以使用 `Scanner/Valuer`：

```go
type User struct {
    Name sql.NullString `gorm:"default:'max'"` //这个结构体类型实现了 Scanner/Valuer 接口
}
// 创建
u := User{Name:sql.NullString{String:"",Valid:true},age:18}

```

查询

`db.First()` 根据主键排序，查询第一条数据，这里必须传入指针类型的结构体

`db.Last()` 与上面查询相反

`db.Find()` 需要传入结构体切片的指针，返回的是所有记录

```go
func TestGetUser(t *testing.T) {

    var u table.UserInfo
    // 根据主键查询第一个记录
    DB.First(&u) // First 和 Last 会根据主键排序，分别查询第一条和最后一条数据。只有在目标 struct 是指针或者通过 db.Model() 指定 model 时，该方法才有效。
    //  这里只能传指针，我的理解是查询到的结果需要传入原结构体而不是副本。
    fmt.Println(u)
    // DB.Find() 也是需要传入结构体切片的指针，返回的是所有记录
    //DB.Find(table.UserInfo{2, "max", "run"})


    // works because model is specified using `db.Model()`
    result := map[string]interface{}{}
    db.Model(&User{}).First(&result)

   // doesn't work
    result := map[string]interface{}{}
    db.Table("users").First(&result)
}
```

> 关于 first 为什么必须传入指针，官方文档的解释：
>
> The First and Last methods will find the first and last record (respectively) as ordered by primary key. **They only work when a pointer to the destination struct is passed to the methods as argument or when the model is specified using db.Model().** Additionally, if no primary key is defined for relevant model, then the model will be ordered by the first field.

根据 struct 和 map 查询：

```go
// Struct
db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)
// SELECT * FROM users WHERE name = "jinzhu" AND age = 20 ORDER BY id LIMIT 1;

// Map 可以查询零值，但改变结构体后，map对应的 key 也需要修改
db.Where(map[string]interface{}{"name": "jinzhu", "age": 20}).Find(&users)
// SELECT * FROM users WHERE name = "jinzhu" AND age = 20;

// Slice of primary keys 不够清晰，不常用这个方法的话比较难理解
db.Where([]int64{20, 21, 22}).Find(&users)
// SELECT * FROM users WHERE id IN (20, 21, 22);
```

注意：当通过**结构体查询时**，GORM 只会通过非零字段进行查询，如果你的某个字段值为零，那么将不会作为构建查询的条件。

```go
// 获取第一条匹配的记录，或者根据给定的条件初始化一个实例（仅支持 sturct 和 map 条件）
db.FirstOrInit(&user, User{Name: "non_existing"})
//// user -> User{Name: "non_existing"}

```

Limit 指定从数据库中检索出的最大记录数

```go
db.Limit(3).Find(&users)
// SELECT * FROM users LIMIT 3;
```

Offset 指定开始返回记录前要跳过的记录数

```go
db.Offset(3).Find(&users)
// SELECT * FROM users OFFSET 3;
```

Count 该 model 能获取的记录总数

```go
db.Where("name = ?","max").Or("name = ?","maxs").Find(&users).Count()&count
```

注意： `Count` 必须是链式查询的最后一个操作，因为他会覆盖前面的 `SELECT` ，但是如果里面使用了 `count` 时不会覆盖。

注意：如果你需要把数据赋值给结构体切片，记得传入结构体切片的指针

更新

保存所有字段

`db.Save()` 默认更新表里的所有字段，即使字段是零值

```go
// 对主键为 3 的字段进行 update
user.ID = 3
user.Name = "xiaoming"
user.Hobby = "game"
db.Save(&user)
// 未指定主键，使用 create
user.Name = "xiaoming"
user.Hobby = "game"
db.Save(&user)
```

`Save()` 是一个组合型函数，**如果保存的值里不包含主键，它就会执行 `Create()` 函数，否则就会在所有字段上执行 `Update()` 函数。**

> NOTE：不要在使用 Save() 时使用 Model()，这是未定义的行为。

更新单个列

`db.Update()` 更新单个字段。当使用 `Update()` 更新单个列时，必须指定条件，否则会出现 `ErrMissingWhereClause` 错误，当使用了 Model 方法，且该对象主键有值，该值会被用于构建条件。

```go
// 使用 update 更新指定字段
db.Model(&table.UserInfo{}).Where("name = ?", "max").Update("name", "hello")

// 当指定 记录的主键时，主键会自动用来成为更新的条件
user.ID = 1
db.Model(&user).Update("name", "max")
// 表中 name=max 的字段都会被更新为 name=hello
```

更新多个列

`db.Updates()` 更新多个字段

```go
// 使用 map or struct 更新多个字段，只能更新非零值字段
//db.Model(&user).Where("id = ?", 1).Updates(map[string]interface{}{"name": "max", "hobby": "ride"})
db.Model(&user).Where("id = ?", 1).Updates(table.UserInfo{
    ID:    1,
    Name:  "max2",
    Hobby: "code",
})
```

> 注意：
>
> 1. 使用 map or struct 更新多个字段，
> 2. struct 只能更新非零值字段，如果要确保更新某个字段，可以使用 map 或者 `SELECT` 指定

更新选定字段

指定字段更新：`Select()`

忽略字段：`Omit()`

```go
// 指定字段更新:
user.ID = 1
//这里只选择了 name 字段，所以哪怕其他字段不同也只更新 name
db.Model(&user).Select("name").Updates(map[string]interface{}{"name": "max1", "hobby": "cook"})
// 这里忽略了 name，所以会更新其他字段
db.Model(&user).Omit("name").Updates(map[string]interface{}{"name": "test", "hobby": "cook"})

// 更新零值字段:(注意提前指定 user 的主键)
db.Model(&user).Select("name").Updates(table.UserInfo{
        ID:    1,
        Name:  "",
        Hobby: "code",
})
```

更新 Hook

对于更新操作，GORM 支持 BeforeSave、BeforeUpdate、AfterSave、AfterUpdate 钩子，这些方法将在更新记录时被调用。

```go
func (u *UserInfo) BeforeUpdate(tx *gorm.DB) (err error) {
    if u.ID == 1 {
        return errors.New("ID = 1  的字段不能被修改")
    }
    return
}
```

批量更新

如果不指定结构体的主键的值，那么 GORM 会进行批量更新:

```Go
// 更新所有 name = xiaoming 的字段
db.Model(&table.UserInfo{}).Where("name = ?", "xiaoming").Updates(table.UserInfo{
    Name:  "xiaohua",
    Hobby: "IT",
})
```

更新的记录数

```Go
// 通过 `RowsAffected` 得到更新的记录数
result := db.Model(User{}).Where("ID = ?", "1").Updates(User{Name: "hello", hobby: "code"})
// UPDATE users SET name='hello', age=18 WHERE role = 'admin;

result.RowsAffected // 更新的记录数
result.Error        // 更新的错误
```

删除

删除一条记录

删除之前必须指定字段，否则会触发批量删除

```Plain Text
// 删除一条记录
user.ID = 4
db.Delete(&user)
// 条件删除
db.Where("name = ?", "xiaohua").Delete(&user) // 删除了 id = 4,name = xiaohua 的字段
```

根据主键删除

> GORM 允许通过内联条件指定主键来检索对象，但只支持整型数值，因为 string 可能导致 SQL 注入。

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

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

db.Delete(&users, []int{1,2,3})
// DELETE FROM users WHERE id IN (1,2,3);

```

批量删除

如果删除时不指定主键，GORM 会删除所有匹配的字段

```go
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

如果您的模型包含了一个 gorm.deletedat 字段（gorm.Model 已经包含了该字段），它就会获得软删除的特性。

拥有软删除能力的模型调用 Delete 时，**记录不会被从数据库中真正删除**。但 GORM 会将 **DeletedAt 置为当前时间， 并且你不能再通过正常的查询方法找到该记录。**

可以使用 `unscoped()` 找到该记录并删除

```Go
// 找到被软删除的记录
db.Unscoped().Where("age = 20").Find(&users)
// SELECT * FROM users WHERE age = 20;
//硬删除:
db.Unscoped().Delete(&order)
// DELETE FROM orders WHERE id=10;
```

注意事项

后端不要相信前端传递的任何数据，每次传递数据都需要验证。

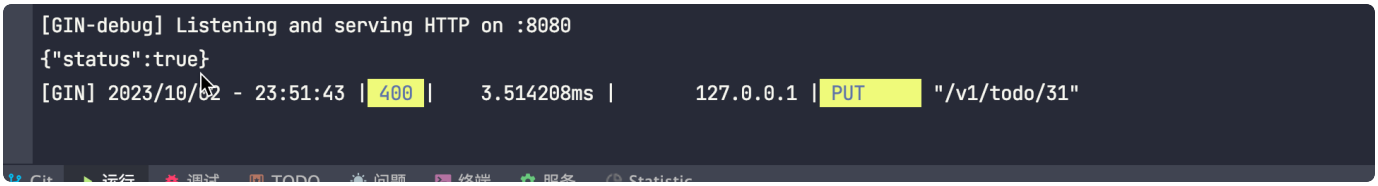
在开发时遇到的有意思的点:

在编写 修改事项 的 API 时，比较好奇前端传入的参数到底是一个结构体还是其他内容，如果是结构体，就感觉多传了一次 ID，如果不是结构体，那么我在底层插入数据库时又不知道该怎么处理。

最终找到了相应的答案:

这里前端传入的确实是一个结构体，但是这个结构体的内容只有 `Status` 字段的值:

```Vue
datas, _ := c.GetRawData()
  fmt.Println(string(datas))
```



但是只有结构体和 `Status` 字段的值我该如何去修改数据库内容？在网上查阅资料后发现一些资料，整理为测试:

```go
type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func TestJsonDecode(t *testing.T) {
    str := `{"age":18}`                    // 模拟前端传来的某条参数;
    person := Person{Name: "老六", Age: 27} // 模拟数据库中已有的记录;
    err := json.NewDecoder(strings.NewReader(str)).Decode(&person)
    if err == nil {
        fmt.Printf("解码结果: %+v", person)
    } // 解码结果: {Name:老六 Age:18}
}


```

这里 `json.NewDecoder` 会返回一个实例化的 `Decode` ,然后将输入流存储到缓冲区里

Go

```go
// 调用 newDecoder 会直接返回一个带有 io.Reader 的 Decoder 实例和响应的缓冲区
func NewDecoder(r io.Reader) *Decoder {
    return &Decoder{r: r}
}


type Decoder struct {
    r       io.Reader
    buf     []byte  //
    d       decodeState
    scanp   int   // start of unread data in buf
    scanned int64 // amount of data already scanned
    scan    scanner
    err     error

    tokenState int
    tokenStack []int
}

func (dec *Decoder) Decode(v any) error {
    if dec.err != nil {
        return dec.err
    }

    if err := dec.tokenPrepareForDecode(); err != nil {
        return err
    }

    if !dec.tokenValueAllowed() {
        return &SyntaxError{msg: "not at beginning of value", Offset: dec.InputOffset()}
    }

    // Read whole value into buffer.
    n, err := dec.readValue()
    if err != nil {
        return err
    }
    dec.d.init(dec.buf[dec.scanp : dec.scanp+n])
    dec.scanp += n

    // Don't save err from unmarshal into dec.err:
    // the connection is still usable since we read a complete JSON
    // object from it before the error happened.
    err = dec.d.unmarshal(v)    //在这里将输入流的内容反向解析到 v
```

```
45
46          // fixup token streaming state
47          dec.tokenValueEnd()
48
49          return err
50      }
```

`Decode` 会将缓冲区的数据作为输入源解码到 person 结构体中，这样我就可以调用 `Save` 存储到数据库了。

可能这部分源码还是没看太懂，后面需要继续确认一下。