第二阶段：Gin Web框架（下篇）

> *讲师：阿杜*

一、GORM入门

1.1 什么是ORM?

`orm是一种术语而不是软件`

- orm英文全称object relational mapping,就是 对象映射关系 程序
- 简单来说类似python这种面向对象的程序来说一切皆对象，但是我们使用的数据库却都是关系型的
- 为了保证一致的使用习惯，通过 orm将编程语言的对象模型和数据库的关系模型建立映射关系
- 这样我们直接 使用编程语言的对象模型进行操作数据库 就可以了，而不用直接使用sql语言

1.2 什么是GORM?

参考文档：https://gorm.io/zh_CN/docs/index.html

`GORM是一个神奇的，对开发人员友好的 Golang ORM 库`

- 全特性 ORM (几乎包含所有特性)
- 模型关联 (一对一， 一对多，一对多 （反向）， 多对多， 多态关联)
- 钩子 (Before/After Create/Save/Update/Delete/Find)
- 预加载
- 事务
- 复合主键

- SQL 构造器
- 自动迁移
- 日志
- 基于GORM回调编写可扩展插件
- 全特性测试覆盖
- 开发者友好

1.3 GORM(v2)基本使用

1. 安装

```
go get -u gorm.io/gorm
```

2. 连接MySQL

- 先创建一个数据库

```
mysql>  create database test_db charset utf8;  # 创建数据库
mysql>  use test_db;                           # 切换到数据库

mysql>  show tables;                           # 查看是否生成表
+------------------+
| Tables_in_test_db |
+------------------+
| users            |
+------------------+

mysql> desc users;                        # 查看表的字段是否正常
+----------+------------+------+-----+---------+----------------+
| Field    | Type       | Null | Key | Default | Extra          |
+----------+------------+------+-----+---------+----------------+
| id       | bigint(20) | NO   | PRI | NULL    | auto_increment |
| username | longtext   | YES  |     | NULL    |                |
| password | longtext   | YES  |     | NULL    |                |
+----------+------------+------+-----+---------+----------------+
```

- 创建mysql连接

参考文档：https://gorm.io/zh_CN/docs/connecting_to_the_database.html

```
package main

import (
    "fmt"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

func main() {
    //parseTime是查询结果是否自动解析为时间
    //loc是Mysql的时区设置
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
charset=utf8mb4&parseTime=True&loc=Local"
    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    if err != nil {
      fmt.Println(err)
    }
    fmt.Println(db)  // &{0xc00018a630 <nil> 0 0xc000198380 1}
}
```

3. 自动创建表

参考文档：https://gorm.io/zh_CN/docs/models.html

```
package main

import (
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

// User 表的结构体ORM映射
type User struct {
```

```
    Id    int64 `gorm:"primary_key" json:"id"`
    Username  string
    Password  string
}

func main() {
    // 1、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 2、自动创建表
    db.AutoMigrate(
      User{},
    )
}
```

4. 基本增删改查

参考文档：https://gorm.io/zh_CN/docs/index.html

```
package main

import (
    "fmt"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

// User 表的结构体ORM映射
type User struct {
    Id    int64 `gorm:"primary_key" json:"id"`
    Username string
    Password string
}

func main() {
    // 0、连接数据库
```

```
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    db.AutoMigrate(User{})
    // 1、增
    db.Create(&User{
        //Id:    3,
        Username: "zhangsan",
        Password: "123456",
    })
    // 2、改
    db.Model(User{Id: 3,}).Update("username", "lisi")
    //db.Model(User{}).Where("id = 1").Update("username", "lisi")
    // 3、查
    // 3.1 过滤查询
    u := User{Id: 3}
    db.First(&u)
    fmt.Println(u)
    // 3.2 查询所有数据
    users := []User{}
    db.Find(&users)
    fmt.Println(users)  // [{2 zhangsan 123456} {3 lisi 123456}]
    // 4、删
    // 4.1 删除 id = 3 的用户
    db.Delete(&User{Id: 3})
    // 4.2 条件删除
    db.Where("username = ?", "zhangsan").Delete(&User{})
}
```

1.4 模型定义

参考文档：https://gorm.io/zh_CN/docs/models.html

1. 模型定义

- 模型一般都是普通的 Golang 的结构体，Go的基本数据类型，或者指针。
- 例子：

```
type User struct {
    Id           int64       `gorm:"primary_key" json:"id"`
    Name         string
    CreatedAt      *time.Time  `json:"createdAt"
gorm:"column:create_at"`
    Email        string     `gorm:"type:varchar(100);unique_index"`
// 唯一索引
    Role       string     `gorm:"size:255"`                    // 设置
字段的大小为255个字节
    MemberNumber  string       `gorm:"unique;not null"`
 // 设置memberNumber 字段唯一且不为空
    Num          int        `gorm:"AUTO_INCREMENT"`                 //
设置 Num字段自增
    Address      string     `gorm:"index:addr"`                   // 给
Address 创建一个名字是 `addr`的索引
    IgnoreMe     int        `gorm:"-"`                          //忽略这个
字段
}
```

```
mysql> desc users;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| id           | bigint       | NO   | PRI | NULL    | auto_increment |
| name         | longtext     | YES  |     | NULL    |                |
| create_at    | datetime(3)  | YES  |     | NULL    |                |
| email        | varchar(100) | YES  |     | NULL    |                |
| role         | varchar(255) | YES  |     | NULL    |                |
| member_number| varchar(191) | NO   | UNI | NULL    |                |
| num          | bigint       | YES  |     | NULL    |                |
| address      | varchar(191) | YES  | MUL | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
```

2. 支持结构标签

- 标签是声明模型时可选的标记

标签	说明
Column	指定列的名称
Type	指定列的类型
Size	指定列的大小，默认是 255
PRIMARY_KEY	指定一个列作为主键
UNIQUE	指定一个唯一的列
DEFAULT	指定一个列的默认值
PRECISION	指定列的数据的精度
NOT NULL	指定列的数据不为空
AUTO_INCREMENT	指定一个列的数据是否自增
INDEX	创建带或不带名称的索引，同名创建复合索引
UNIQUE_INDEX	类似 索引 ，创建一个唯一的索引
EMBEDDED	将 struct 设置为 embedded
EMBEDDED_PREFIX	设置嵌入式结构的前缀名称
-	忽略这些字段

二、一对多关联查询

参考文档：https://gorm.io/zh_CN/docs/has_many.html

2.1 一对多入门

1. has many介绍

- `has many` 关联就是创建和另一个模型的一对多关系
- 例如， 例如每一个用户都拥有多张信用卡，这样就是生活中一个简单的一对多关系

User表	一个用户可以有多个信用卡	
ID	Name	Card
1	zhangsan	1001
2	zhangsan	1002
3	zhangsan	1003
4	zhangsan	1004
5	zhangsan	1005
6	zhangsan	1006

User表		Card表		
Id	Name	Id	CardId	UserId
1	zhangsan	1	1001	1
		2	1002	1
		3	1003	1
		4	1004	1
		5	1005	1
		6	1006	1

```
// 用户有多张信用卡，UserID 是外键
type User struct {
    gorm.Model
    CreditCards []CreditCard
}
type CreditCard struct {
    gorm.Model
    Number  string
    UserID  uint   // 默认会在 CreditCard 表中生成 UserID 字段作为 与User表关联的外键ID
}
```

2. 外键

- 为了定义一对多关系， 外键是必须存在的，默认外键的名字是 所有者类型的名字加上它的主键(UserId) 。
- 就像上面的例子，为了定义一个属于 User 的模型，外键就应该为 UserID 。
- 使用其他的字段名作为外键， 你可以通过 foreignkey 来定制它， 例如:

```
type User struct {
  gorm.Model
  CreditCards []CreditCard `gorm:"foreignKey:UserRefer"`
}

type CreditCard struct {
  gorm.Model
  Number    string
  UserRefer uint
}
```

3. 外键关联

- GORM 通常使用所有者的主键作为外键的值， 在上面的例子中，它就是 User 的 ID 。
- 当你分配信用卡给一个用户， GORM 将保存用户 ID 到信用卡表的 UserID 字段中。
- 你能通过 association_foreignkey 来改变它

```
type User struct {
    gorm.Model
    MemberNumber string
    // 默认CreditCard会使用User表的Id作为外键,
association_foreignkey:MemberNumber
    // 指定使用MemberNumber 作为外键关联
    CreditCards []CreditCard
`gorm:"foreignkey:UserMemberNumber;association_foreignkey:MemberNumbe
r"`
}
type CreditCard struct {
    gorm.Model
    Number       string
    UserMemberNumber  string
}
```

2.2 创建一对多表

参考文档：https://gorm.io/zh_CN/docs/associations.html#%E8%87%AA%E5%8A%A8%E5%88%9B%E5%BB%BA%E3%80%81%E6%9B%B4%E6%96%B0

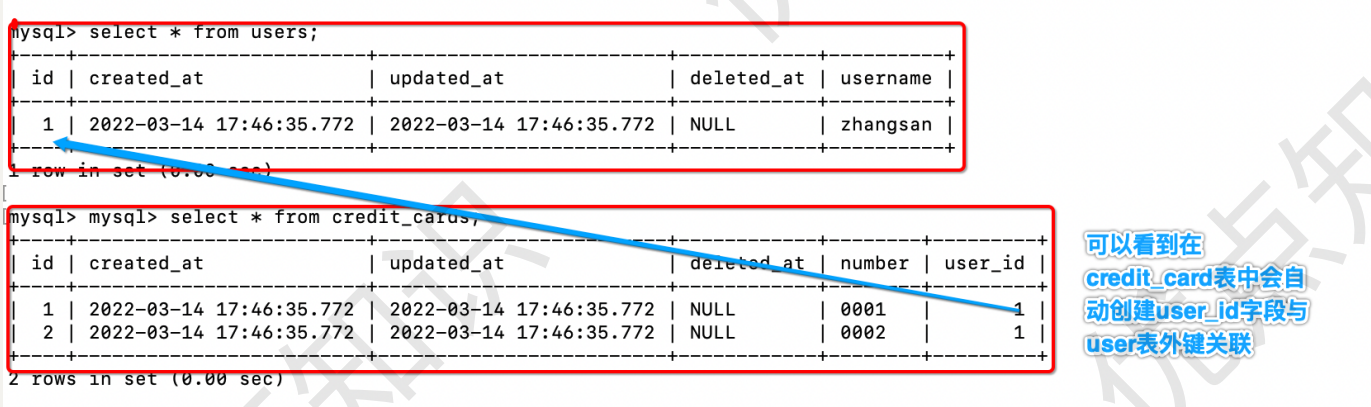
1. 表结构定义

```
package main

import (
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)
```

```
/*
constraint:OnUpdate:CASCADE 【当User表更新，也会同步给CreditCards】 // 外键约束
OnDelete:SET NULL 【当User中数据被删除时，CreditCard关联设置为 NULL，不删除记录】
*/
type User struct {
    gorm.Model
    Username   string    `json:"username" gorm:"column:username"`
    CreditCards []CreditCard
  `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
}
type CreditCard struct {
    gorm.Model
    Number string
    UserID uint
}
func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 创建表结构
    db.AutoMigrate(User{}, CreditCard{})
    // 1、创建一对多
    user := User{
        Username: "zhangsan",
        CreditCards: []CreditCard{
        {Number: "0001"},
        {Number: "0002"},
      },
    }
    db.Create(&user)
    // 2、为已存在用户添加信用卡
    u := User{Username: "zhangsan"}
    db.First(&u)
    //fmt.Println(u.Username)
}
```

2. 创建结果说明

- 我们没有指定 foreignkey，所以会与 UserID字段自动建立外键关联关系



3. 一对多Association

- 查找关联
- 使用 `Association` 方法，需要把把 `User` 查询好，然后根据 `User` 定义中指定的 `AssociationForeignKey` 去查找 `CreditCard`

```
package main

import (
    "encoding/json"
    "fmt"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

/*
constraint:OnUpdate:CASCADE 【当User表更新，也会同步给CreditCards】
OnDelete:SET NULL 【当User中数据被删除时，CreditCard关联设置为 NULL，不删除记录】
*/
type User struct {
    gorm.Model
    Username    string     `json:"username" gorm:"column:username"`
    CreditCards []CreditCard
  `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
```

```
}
type CreditCard struct {
    gorm.Model
    Number string
    UserID uint
}
func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、查找 用户名为 zhangsan 的所有信用卡信息
    u := User{Username: "zhangsan"}  // Association必须要先查出User才能
关联查询对应的CreditCard
    db.First(&u)
    err :=
db.Model(&u).Association("CreditCards").Find(&u.CreditCards)
    if err != nil {
        fmt.Println(err)
    }
    err =
db.Model(&u).Association("CreditCards").Append([]CreditCard{
        {Number: "0003"},
    })
    if err != nil {
        fmt.Println(err)
    }
    strUser, _ := json.Marshal(&u)
    fmt.Println(string(strUser))
}
```

- 打印结果如下

```
{
    "ID":1,
    "username":"zhangsan",
    "CreditCards":[
        {
            "ID":1,
            "Number":"0001",
            "UserID":1
        },
        ...
    ]
}
```

4. 一对多Preload

- 预加载
- 使用`Preload`方法，在查询`User`时先去获取`CreditCard`的记录

```
package main

import (
    "encoding/json"
    "fmt"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)
/*
constraint:OnUpdate:CASCADE 【当User表更新，也会同步给CreditCards】
OnDelete:SET NULL 【当User中数据被删除时，CreditCard关联设置为 NULL，不删除记录】
*/
type User struct {
    gorm.Model
    Username    string    `json:"username" gorm:"column:username"`
    CreditCards []CreditCard
  `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET
    NULL;"`
```

```
}
type CreditCard struct {
    gorm.Model
    Number string
    UserID uint
}
func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、预加载：查找 user 时预加载相关 CreditCards
    //users := User{Username: "zhangsan"} // 只查找张三用户的信用卡信息
    users := []User{}
    db.Preload("CreditCards").Find(&users)
    strUser, _ := json.Marshal(&users)
    fmt.Println(string(strUser))
}
```

- 查询结果

```
[
     {
        "ID":1,
        "username":"zhangsan",
        "CreditCards":[
          {
            "ID":1,
            "Number":"0001",
            "UserID":1
          },
        ...
       ]
     }
]
```

三、多对多

3.1 多对多入门

参考文档：https://gorm.io/zh_CN/docs/many_to_many.html

1. Many To Many

一个学生可以选择多个课程，一个课程又包含多个学生（go、vue）：**双向一对多**

Student表

ID	Name	Lesson
1	zhangsan	go
2	zhangsan	vue
3	lisi	go
4	lisi	vue

Student

ID	Name
1	zhangsan
2	lisi

Lesson

Id	Lesson
1	go
2	vue

StudentToLesson

Id	userId	lessonId
1	1	1
2	1	2
3	2	1
4	2	2

- Many to Many 会在两个 model 中添加一张连接表。
- 例如，您的应用包含了 user 和 language，且一个 user 可以说多种 language，多个 user 也可以说一种 language。
- 当使用 GORM 的 AutoMigrate 为 User 创建表时，GORM 会自动创建连接表

```
// User 拥有并属于多种 language，`user_languages` 是连接表
type User struct {
    gorm.Model
    Languages []Language `gorm:"many2many:user_languages;"`
}
type Language struct {
    gorm.Model
    Name string
}
```

2. 反向引用

```
// User 拥有并属于多种 language，`user_languages` 是连接表
type User struct {
  gorm.Model
  Languages []*Language `gorm:"many2many:user_languages;"`
}
type Language struct {
    gorm.Model
    Name string
    Users []*User `gorm:"many2many:user_languages;"`
}
```

3. 重写外键

- 对于`many2many`关系，连接表会同时拥有两个模型的外键

```
type User struct {
    gorm.Model
    Languages []Language `gorm:"many2many:user_languages;"`
}
type Language struct {
    gorm.Model
    Name string
}
// 连接表：user_languages
//  foreign key: user_id, reference: users.id
//  foreign key: language_id, reference: languages.id
```

- 若要重写它们，可以使用标签`foreignKey`、`references`、`joinforeignKey`、`joinReferences`。
- 当然，您不需要使用全部的标签，你可以仅使用其中的一个重写部分的外键、引用

```
type User struct {
    gorm.Model
    Profiles []Profile
`gorm:"many2many:user_profiles;foreignKey:Refer;joinForeignKey:UserRe
ferID;References:UserRefer;joinReferences:ProfileRefer"`
    Refer   uint    `gorm:"index:,unique"`
}
type Profile struct {
    gorm.Model
    Name    string
    UserRefer uint `gorm:"index:,unique"`
}
// 会创建连接表：user_profiles
//  foreign key: user_refer_id, reference: users.refer
//  foreign key: profile_refer, reference: profiles.user_refer
```

3.2 创建多对多表

1. m2m生成第三张表

```
package main

import (
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

// User 拥有并属于多种 language，`user_languages` 是连接表
type User struct {
    gorm.Model
    Languages []Language `gorm:"many2many:user_languages;"`
}

type Language struct {
    gorm.Model
    Name string
```

```
}
func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、自动创建多对多表结构
    db.AutoMigrate(
        User{},
        Language{},
    )
}
```

- 生成如下三张表

```
mysql>
mysql> show tables;
+------------------+
| Tables_in_test_db |
+------------------+
| languages
| user_languages
| users
+------------------+
3 rows in set (0.00 sec)

mysql> desc user_languages;
+-------------+----------------------+------+-----+---------+-------+
| Field       |                      | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| user_id     | bigint(20) unsigned  | NO   | PRI | NULL    |       |
| language_id | bigint(20) unsigned  | NO   | PRI | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

会自动创建第三张关联表 user_languages

关联表中默认为 user_id language_id

2. 自定义第三张表

```
package main

import (
    "time"
    "gorm.io/driver/mysql"
```

```
    "gorm.io/gorm"
)

type Person struct {
    ID     int
    Name    string
    Addresses []Address `gorm:"many2many:person_addresses;"`
}

type Address struct {
    ID  uint
    Name string
}

type PersonAddress struct {
    PersonID  int `gorm:"primaryKey"`
    AddressID int `gorm:"primaryKey"`
    CreatedAt time.Time
    DeletedAt gorm.DeletedAt
}

func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、自动创建多对多表结构
    db.AutoMigrate(
        Person{},
        Address{},
    )
    // 2、添加数据
    persons := Person{
        ID:  1,
        Name: "zhangsan",
        Addresses: []Address{
            {ID: 1, Name: "bj"},
            {ID: 2, Name: "sh"},
        },
```

```
    }
    db.Create(&persons)
}
```

- 生成三张表如下

```
mysql>
mysql> show tables;
+-------------------+
| Tables_in_test_db |
+-------------------+
| addresses         |
| people            |
| person_addresses  |
+-------------------+
3 rows in set (0.00 sec)

mysql> desc person_addresses;
+------------+---------------------+------+-----+---------+-------+
| Field      | Type                | Null | Key | Default | Extra |
+------------+---------------------+------+-----+---------+-------+
| person_id  | bigint(20)          | NO   | PRI | NULL    |       |
| address_id | bigint(20) unsigned | NO   | PRI | NULL    |       |
+------------+---------------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

3. 多对多Preload

- 预加载

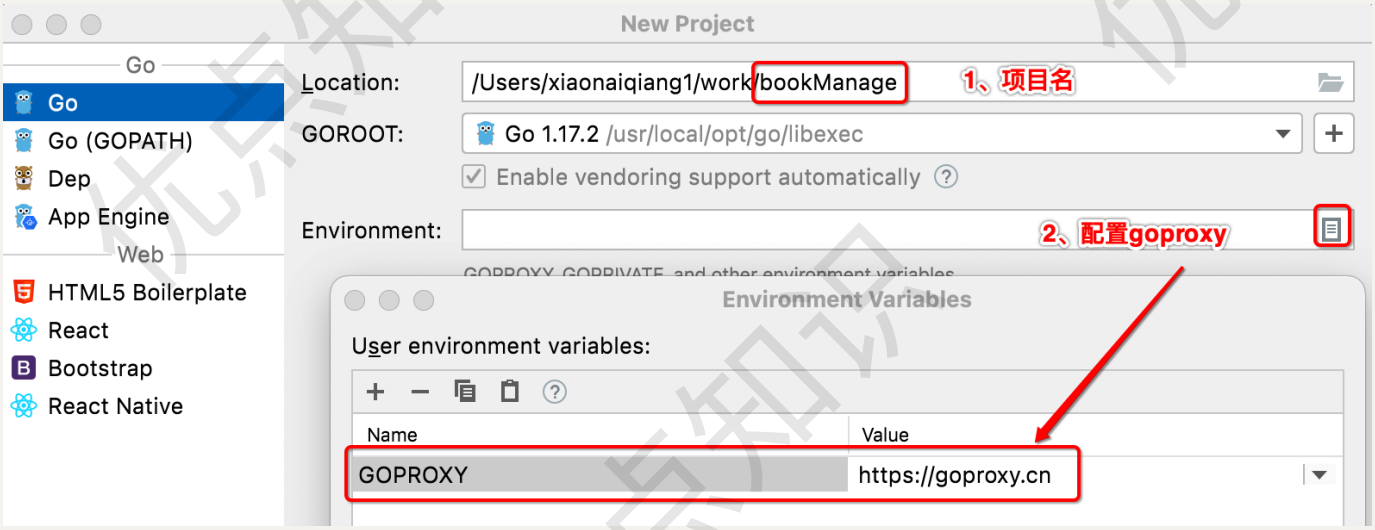
```
package main

import (
    "encoding/json"
    "fmt"
    "time"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

type Person struct {
```

```
    ID     int
    Name    string
    Addresses  []Address `gorm:"many2many:person_addresses;"`
}

type Address struct {
    ID  uint
    Name string
}

type PersonAddress struct {
    PersonID  int `gorm:"primaryKey"`
    AddressID int `gorm:"primaryKey"`
    CreatedAt time.Time
    DeletedAt gorm.DeletedAt
}

func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?
    charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、获取 name="zhangsan" 用户的地址
    persons := []Person{}
    db.Preload("Addresses").Find(&persons)
    strPersons, _ := json.Marshal(&persons)
    fmt.Println(string(strPersons))
    // [{"ID":1,"Name":"zhangsan","Addresses":[{"ID":1,"Name":"bj"},
{"ID":2,"Name":"sh"}]}]
    // 2、获取 name="zhangsan" 用户的地址
    person := Person{Name: "zhangsan"}
    db.Preload("Addresses").Find(&person)
    strPerson, _ := json.Marshal(&person)
    fmt.Println(string(strPerson))
    // {"ID":1,"Name":"zhangsan","Addresses":[{"ID":1,"Name":"bj"},
{"ID":2,"Name":"sh"}]}
}
```

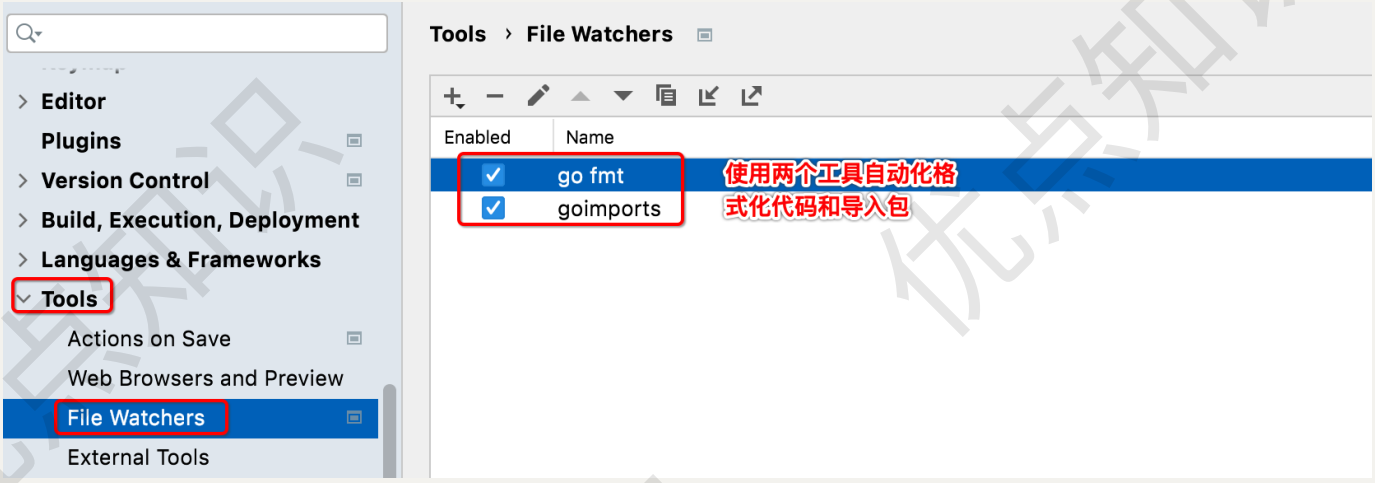
四、图书管理系统

4.1 初始化项目环境

1. 创建项目配置goproxy



2. 添加格式化工具



3. go常用项目结构

```
.
├── Readme.md    // 项目说明（帮助你快速的属性和了解项目）
├── config     // 配置文件（mysql配置 ip 端口 用户名 密码，不能写死到代码中）
├── controller   // CLD：服务入口，负责处理路由、参数校验、请求转发
├── service    // CLD：逻辑（服务）层，负责业务逻辑处理
├── dao      // CLD：负责数据与存储相关功能（mysql、redis、ES等）
├── db       // CLD：负责数据与存储初始化（mysql、redis、ES等）
│   ├── mysql
├── model     // 模型（定义表结构）
├── logging    // 日志处理
├── main.go    // 项目启动入口
├── middleware   // 中间件
```

4. 创建数据库

```
mysql> create database books charset utf8;
```

5. 当前项目结构

```
go mod init bookManage
```

- 图书管理服务
    - 用户服务：登录，注册
    - 书籍服务：对书籍的增删改查的操作

```
.
├── controller       // CLD：服务入口，负责处理路由、参数校验、请求转发
│   ├── book.go
│   ├── user.go
│   └── router.go
├── service       // CLD：逻辑（服务）层，负责业务逻辑处理
│   ├── book.go
│   └── user.go
```

```
├── dao            // CLD：负责数据与存储相关功能（mysql、redis、ES等）
│   ├── book.go
│   └── user.go
├── db             // CLD：负责数据与存储初始化（mysql、redis、ES等）
│   └── mysql
│       └── mysql.go
├── model          // 模型
│   ├── book.go
│   ├── user.go
│   └── user_m2m_book.go
├── middleware     // 中间件：token验证
│   └── auth.go
├── main.go        // 项目启动入口
```

4.2 定义多对多表结构

1. model/user.go

```go
package model

type User struct {
    Id    int64  `gorm:"primary_key" json:"id"`
    Username string `gorm:"not null" json:"username"
binding:"required"`
    Password string `gorm:"not null" json:"password"
binding:"required"`
    Token   string `json:"token"`
}

func (User) TableName() string {
    return "user"
}
```

2. model/book.go

```
package model

type Book struct {
    Id   int64  `gorm:"primary_key" json:"id"`
    Name  string `gorm:"not null" json:"Name" binding:"required"`
    Desc  string `json:"desc"`
    Users []User `gorm:"many2many:book_users;"`
}

func (Book) TableName() string {
  return "book"
}
```

3. model/user_m2m_book.go

```
package model

type BookUser struct {
    UserID int64 `gorm:"primaryKey"`
    BookID int64 `gorm:"primaryKey"`
}
```

4. 初始化mysql连接

`dao/mysql/mysql.go`

```
package mysql

import (
  "bookManage/model"
  "fmt"
  "gorm.io/driver/mysql"
  "gorm.io/gorm"
)
```

```
var DB *gorm.DB

func InitMysql() {
  //拼接配置
  dsn := "root:123456@tcp(127.0.0.1:3306)/books?
charset=utf8mb4&parseTime=True&loc=Local"
  db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
  //defer db.Close()
  if err != nil {
    fmt.Println(err)
  }
  DB = db
  err = DB.AutoMigrate(model.Book{}, model.User{})
  if err != nil {
    fmt.Println(err)
  }
}
```

4.3 用户登录及注册

1. dao/user.go

```
package dao

import (
  "bookManage/db/mysql"
  "bookManage/model"
  "errors"
  "fmt"
  "github.com/wonderivan/logger"
  "gorm.io/gorm"
)

var User user
```

```
type user struct{}

//新增
func (*user) Add(user *model.User) error {
  tx := mysql.DB.Create(&user)
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("添加User失败, %v\n", tx.Error))
    return errors.New(fmt.Sprintf("添加User失败, %v\n", tx.Error))
  }
  return nil
}

//查询单个
func (*user) Has(name string) (*model.User, bool, error) {
  data := &model.User{}
  tx := mysql.DB.Where("username = ?", name).First(&data)
  if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
    return nil, false, nil
  }
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("查询User失败, %v\n", tx.Error))
    return nil, false, errors.New(fmt.Sprintf("查询User失败, %v\n",
tx.Error))
  }

  return data, true, nil
}
//查询单个
func (*user) GetByToken(token string) (*model.User, bool, error) {
  data := &model.User{}
  tx := mysql.DB.Where("token = ?", token).First(&data)
  if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
    return nil, false, nil
  }
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("查询User失败, %v\n", tx.Error))
    return nil, false, errors.New(fmt.Sprintf("查询User失败, %v\n",
tx.Error))
  }
```

```
  return data, true, nil
}

//更新
func (*user) UpdateToken(user *model.User, token string) error {
  tx := mysql.DB.Model(user).Where("username = ? and password = ?",
user.Username, user.Password).Update("token", token)
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("更新User Token失败, %v\n", tx.Error))
    return errors.New(fmt.Sprintf("更新User Token失败, %v\n",
tx.Error))
  }
  return nil
}
```

2. service/user.go

```
package service

import (
  "bookManage/dao"
  "bookManage/model"
  "errors"
  "github.com/google/uuid"
)

var User user

type user struct{}

//新增
func (*user) Add(user *model.User) error {
  _, has, err := dao.User.Has(user.Username)
  if err != nil {
    return err
```

```
  }
  if has {
    return errors.New("该数据已存在，请重新添加")
  }
  if err := dao.User.Add(user); err != nil {
    return err
  }
  return nil
}

//更新token
func (*user) UpdateToken(user *model.User) error {
  token := uuid.New().String()
  return dao.User.UpdateToken(user, token)
}

//登录
func (*user) Login(name, password string) error {
  data, has, err := dao.User.Has(name)
  if err != nil {
    return err
  }
  if !has {
    return errors.New("用户名不存在")
  }
  if data.Password != password {
    return errors.New("用户名或密码错误")
  }
  return nil
}
```

3. controller/user.go

```
package controller

import (
```

```
  "bookManage/model"
  "bookManage/service"
  "github.com/gin-gonic/gin"
  "net/http"
)

var User user

type user struct{}

//注册
func (*user) Register(c *gin.Context) {
  p := &model.User{}
  if err := c.ShouldBindJSON(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90400,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  //创建用户
  if err := service.User.Add(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "注册用户成功",
    "data": nil,
  })
}

//登录
func (*user) Login(c *gin.Context) {
```

```
    p := &model.User{}
    if err := c.ShouldBindJSON(p); err != nil {
        c.JSON(http.StatusOK, gin.H{
            "code": 90400,
            "msg":  err.Error(),
            "data": nil,
        })
        return
    }
    //校验账号密码,通过账号密码查记录
    if err := service.User.Login(p.Username, p.Password); err != nil {
        c.JSON(http.StatusOK, gin.H{
            "code": 90500,
            "msg":  err.Error(),
            "data": nil,
        })
        return
    }
    if err := service.User.UpdateToken(p); err != nil {
        c.JSON(http.StatusOK, gin.H{
            "code": 90500,
            "msg":  err.Error(),
            "data": nil,
        })
        return
    }
    c.JSON(http.StatusOK, gin.H{
        "code": 0,
        "msg":  "登录成功",
        "data": nil,
    })
}
```

4. controller/router.go

```
package controller

import (
  "github.com/gin-gonic/gin"
)

var Router router

type router struct{}

func (r *router) InitApiRouter(router *gin.Engine) {
  router.POST("/register", User.Register)
  router.POST("/login", User.Login)
}
```

5. main.go

```
package main

import (
  "bookManage/controller"
  "bookManage/db/mysql"
  "github.com/gin-gonic/gin"
)

func main() {
  //初始化数据库
  mysql.InitMysql()
  //初始化gin
  r := gin.Default()
  //注册路由
  controller.Router.InitApiRouter(r)
  //启动server
  r.Run(":8888")
```

```
}
```

6. 测试注册功能

- POST: http://127.0.0.1:8888/register

```
{
  "username": "lisi",
  "password": "123456"
}
```

- postman测试

5. 登录获取token

- POST: http://127.0.0.1:8888/login

```
{
  "username": "lisi",
  "password": "123456"
}
```

- postman测试

4.4 图书管理

1. dao/book.go

```
package dao

import (
  "bookManage/db/mysql"
  "bookManage/model"
  "errors"
```

```
  "fmt"
  "github.com/wonderivan/logger"
  "gorm.io/gorm"
)

var Book book

type book struct{}

//列表
func (*book) List() ([]*model.Book, error) {
  books := []*model.Book{}
  tx := mysql.DB.Preload("Users").Find(&books)
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("获取Book列表失败, %v\n", tx.Error))
    return nil, errors.New(fmt.Sprintf("获取Book列表失败, %v\n",
tx.Error))
  }
  return books, nil
}

//新增
func (*book) Add(book *model.Book) error {
  tx := mysql.DB.Omit("Users").Create(&book)
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("添加Book失败, %v\n", tx.Error))
    return errors.New(fmt.Sprintf("添加Book失败, %v\n", tx.Error))
  }
  if len(book.Users) > 0 {
    err :=
mysql.DB.Model(&book).Association("Users").Append(book.Users)
    if err != nil {
      logger.Error(fmt.Sprintf("关联新增Book:Users失败, %v\n", err))
      return errors.New(fmt.Sprintf("关联新增Book:Users失败, %v\n",
err))
    }
  }
  return nil
}
```

```
//查询单个，用于新建时判断是否存在
func (*book) Has(name string) (*model.Book, bool, error) {
  data := &model.Book{}
  tx := mysql.DB.Where("name = ?", name).First(&data)
  if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
    return nil, false, nil
  }
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("查询Book失败, %v\n", tx.Error))
    return nil, false, errors.New(fmt.Sprintf("查询Book失败, %v\n",
tx.Error))
  }

  return data, true, nil
}

//查询单个
func (*book) Get(id int) (*model.Book, bool, error) {
  data := &model.Book{}
  tx := mysql.DB.Where("id = ?", id).First(&data)
  if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
    return nil, false, nil
  }
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("查询Book失败, %v\n", tx.Error))
    return nil, false, errors.New(fmt.Sprintf("查询Book失败, %v\n",
tx.Error))
  }

  return data, true, nil
}

//更新
func (*book) Update(book *model.Book) error {
  tx := mysql.DB.Model(&model.Book{}).Where("id = ?",
book.ID).Updates(&book)
  if tx.Error != nil {
    logger.Error(fmt.Sprintf("更新Book失败, %v\n", tx.Error))
```

```
    return errors.New(fmt.Sprintf("更新Book失败, %v\n", tx.Error))
  }
  if len(book.Users) > 0 {
    //关联更新用户, Association.Replace
    err :=
mysql.DB.Model(&book).Association("Users").Replace(book.Users)
    if err != nil {
      logger.Error(fmt.Sprintf("关联更新Book:Users失败, %v\n", err))
      return errors.New(fmt.Sprintf("关联更新Book:Users失败, %v\n",
err))
    }
  } else {
    //关联清空用户, 使用Association.Clear
    err := mysql.DB.Model(&book).Association("Users").Clear()
    if err != nil {
      logger.Error(fmt.Sprintf("关联更新Book:Users失败, %v\n", err))
      return errors.New(fmt.Sprintf("关联更新Book:Users失败, %v\n",
err))
    }
  }
  return nil
}

//删除
func (*book) Delete(id int) error {
  data := &model.Book{
    ID: id,
  }
  err := mysql.DB.Model(&data).Association("Users").Clear()
  if err != nil {
    logger.Error("关联删除Role:Permission失败, " + err.Error())
    return errors.New("关联删除Role:Permission失败, " + err.Error())
  }
  tx := mysql.DB.Delete(&data)
  if tx.Error != nil {
    logger.Error("删除Book失败, " + tx.Error.Error())
    return errors.New("删除Book失败, " + tx.Error.Error())
  }
  return nil
```

```
}
```

2. service/book.go

```
package service

import (
  "bookManage/dao"
  "bookManage/model"
  "errors"
)

var Book book

type book struct{}

//列表
func (*book) List() ([]*model.Book, error) {
  return dao.Book.List()
}

//查询单个
func (*book) Has(name string) (*model.Book, bool, error) {
  return dao.Book.Has(name)
}

//查询单个
func (*book) Get(id int) (*model.Book, bool, error) {
  return dao.Book.Get(id)
}

//新增
func (*book) Add(book *model.Book) error {
  _, has, err := dao.Book.Has(book.Name)
  if err != nil {
    return err
```

```
  }
  if has {
    return errors.New("该数据已存在，请重新添加")
  }
  if err := dao.Book.Add(book); err != nil {
    return err
  }
  return nil
}

//更新
func (*book) Update(book *model.Book) error {
  return dao.Book.Update(book)
}

//删除
func (*book) Delete(id int) error {
  return dao.Book.Delete(id)
}
```

3. controller/book.go

```
package controller

import (
  "bookManage/model"
  "bookManage/service"
  "github.com/gin-gonic/gin"
  "net/http"
)

var Book book

type book struct{}

//新增
```

```go
func (*book) Add(c *gin.Context) {
  p := &model.Book{}
  if err := c.ShouldBindJSON(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90400,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  if err := service.Book.Add(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "创建书籍成功",
    "data": nil,
  })
}

//查看列表
func (*book) List(c *gin.Context) {
  data, err := service.Book.List()
  if err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "查询列表成功",
```

```
    "data": data,
  })
}

//查看书籍详情
func (*book) GetDetail(c *gin.Context) {
  p := new(struct {
    ID int `form:"id"`
  })
  if err := c.Bind(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90400,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  data, _, err := service.Book.Get(p.ID)
  if err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "查询书籍详情成功",
    "data": data,
  })
}

//更新书籍
func (*book) Update(c *gin.Context) {
  p := &model.Book{}
  if err := c.ShouldBindJSON(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90400,
```

```
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  if err := service.Book.Update(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "更新书籍成功",
    "data": nil,
  })
}

//删除
func (*book) Delete(c *gin.Context) {
  p := new(struct {
    ID int `json:"id"`
  })
  if err := c.ShouldBindJSON(p); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90400,
      "msg":  err.Error(),
      "data": nil,
    })
    return
  }
  if err := service.Book.Delete(p.ID); err != nil {
    c.JSON(http.StatusOK, gin.H{
      "code": 90500,
      "msg":  err.Error(),
      "data": nil,
    })
```

```
    return
  }
  c.JSON(http.StatusOK, gin.H{
    "code": 0,
    "msg":  "删除书籍成功",
    "data": nil,
  })
}
```

4. controller/router.go

```
package controller

import (
  "github.com/gin-gonic/gin"
)

var Router router

type router struct{}

func (r *router) InitApiRouter(router *gin.Engine) {
  v1 := router.Group("/api/v1")
  v1.POST("/book/add", Book.Add)
  v1.GET("/book/list", Book.List)
  v1.GET("/book/detail", Book.GetDetail)
  v1.PUT("/book/update", Book.Update)
  v1.DELETE("/book/del", Book.Delete)
}
```

5. 测试创建图书

- POST：http://127.0.0.1:8888/api/v1/book/add
- 请求数据

```
{
  "name": "西游记",
  "desc": "大师兄师傅被妖怪抓走了"
}
```

6. 测试查看图书列表

- GET：http://127.0.0.1:8888/api/v1/book/list
- 返回数据

```
{
  "books": [
    {
      "id": 3,
      "Name": "水浒传",
      "desc": "水浒传豪情满怀",
      "Users": [
      ]
    }
 ]
}
```

7. 测试查看图书详情

- GET：http://127.0.0.1:8888/api/v1/book/detail
- 返回结果

```
{
  "books": {
    "id": 3,
    "Name": "水浒传",
    "desc": "水浒传豪情满怀",
    "Users": null
 }
}
```

6. 测试修改图书信息

- PUT：http://127.0.0.1:8888/api/v1/book/update
- 携带数据

```
{
  "id": 4,
  "name": "西游记后传"
}
```

7. 测试删除图书信息

- DELETE： http://127.0.0.1:8888/api/v1/book/del
- 携带数据

```
{
    "id": 4
}
```

4.5 中间件身份验证

1. middleware/auth.go

```
package middleware

import (
  "bookManage/dao"
  "github.com/gin-gonic/gin"
  "net/http"
)

func AuthMiddleWare() gin.HandlerFunc {
  return func(c *gin.Context) {
    token := c.Request.Header.Get("token")
    //验证token是否正确
```

```
    user, has, err := dao.User.GetByToken(token)
    if err != nil {
      c.JSON(http.StatusOK, gin.H{
        "code": 90500,
        "msg":  err.Error(),
        "data": nil,
      })
      c.Abort()
    }
    if !has {
      c.JSON(http.StatusOK, gin.H{
        "code": 90403,
        "msg":  "鉴权不通过",
        "data": nil,
      })
      c.Abort()
    } else {
      c.Set("UserID", user.ID)
    }
  }
}
```

2. router/router.go

```
package controller

import (
  "bookManage/middleware"
  "github.com/gin-gonic/gin"
)

var Router router

type router struct{}

func (r *router) InitApiRouter(router *gin.Engine) {
```

```
    router.POST("/register", User.Register)
    router.POST("/login", User.Login)
    v1 := router.Group("/api/v1")
    v1.Use(middleware.AuthMiddleWare())
    v1.POST("/book/add", Book.Add)
    v1.GET("/book/list", Book.List)
    v1.GET("/book/detail", Book.GetDetail)
    v1.PUT("/book/update", Book.Update)
    v1.DELETE("/book/del", Book.Delete)
}
```

3. 测试登录功能

- POST: http://127.0.0.1:8888/login
- 请求数据

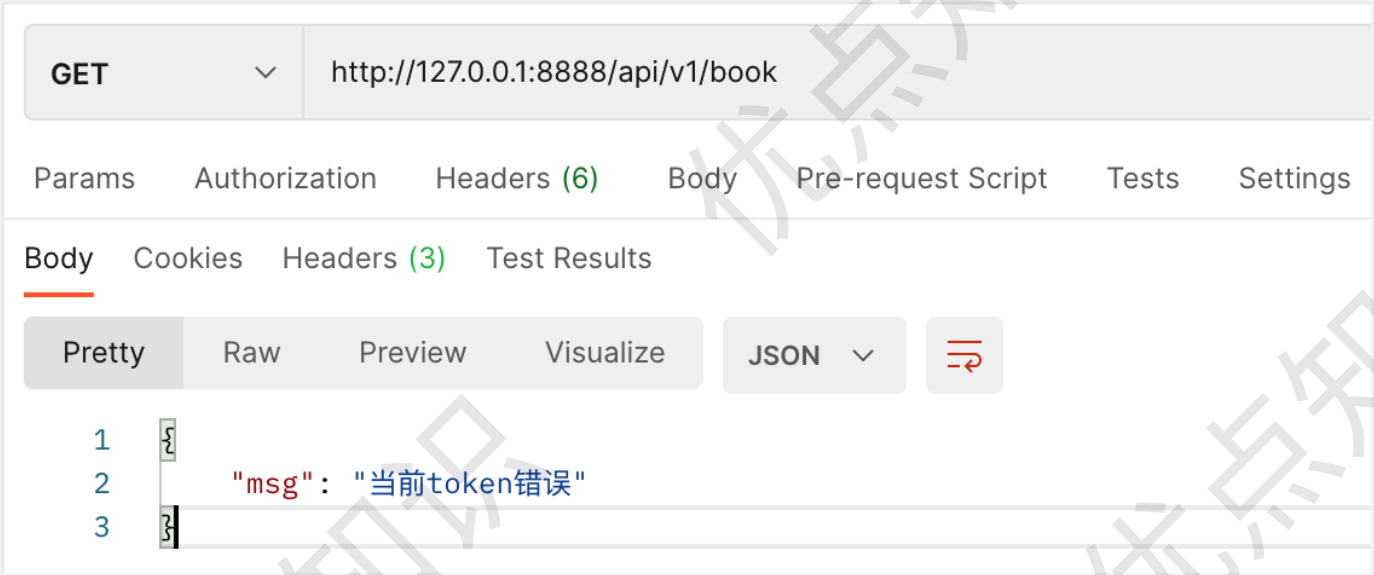
```
{
    "username": "lisi",
    "password": "123456"
}
```

- 请求返回

```
{
    "token": "16f65f89-622a-4b1f-a569-d2057c5d5d4f"
}
```

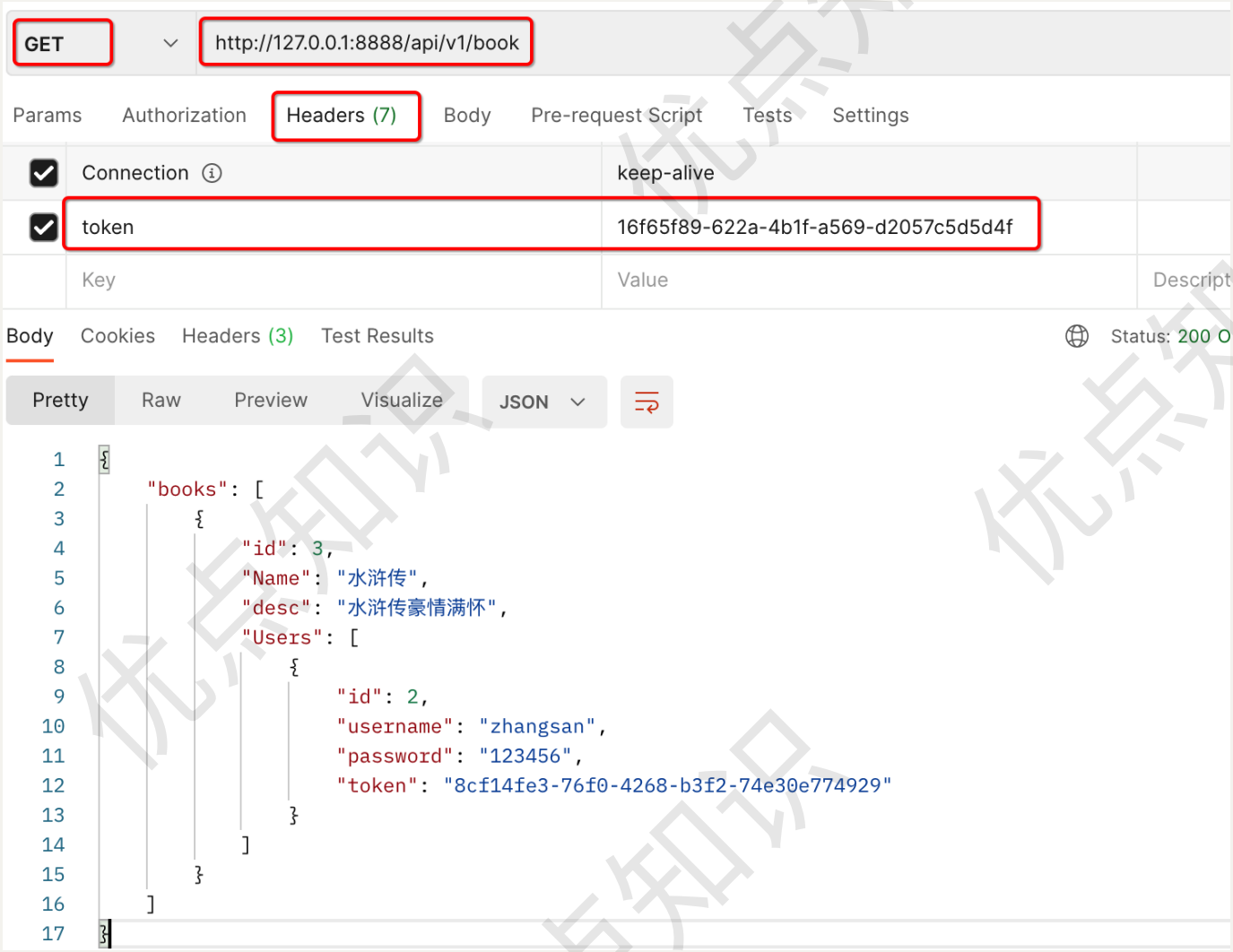
4. 测试无token获取图书列表

- GET: http://127.0.0.1:8888/api/v1/book



5. 携带token访问

- GET: http://127.0.0.1:8888/api/v1/book
- token: "16f65f89-622a-4b1f-a569-d2057c5d5d4f"



五、Restful风格

5.1 什么是RESTful风格

参考文档：https://www.cnblogs.com/xiaonq/p/10053234.html

1. 什么是RESTful

- REST与技术无关，代表的是 一种软件架构风格 （REST是Representational State Transfer的简称，中文翻译为“表征状态转移”）
- REST从 资源 的角度类审视整个网络，它将分布在网络中某个节点的 资源通过URL进行标识

- 所有的数据，不管是通过网络获取的还是 操作（增删改查） 的数据，都是 资源，将一切数据视为资源 是REST区别与其他架构风格的最本质属性
- 对于REST这种面向资源的架构风格，有人提出一种全新的结构理念，即：面向资源架构（ROA：Resource Oriented Architecture）

2. web开发本质

- 对数据库中的表进行增删改查操作
- Restful风格就是把所有数据都当做资源， 对表的操作就是对资源操作
- 在url同通过 资源名称来指定资源
- 通过(增删改查) get/post/put/delete /patch 对资源的操作
    - get : 获取一条数据 （一个学生信息） 、或者是获取数据列表（所有学生信息）
    - post：添加一条数据
    - put：修改一些信息
    - delete：删除一条数据

5.2 RESTful设计规范

1. URL路径

- 面向资源编程： 路径，视网络上任何东西都是资源，均使用名词表示（可复数），不要使用动词

```
# 不好的例子：url中含有动词
/getProducts
/listOrders
# 正确的例子：地址使用名词复数
GET /products     # 将返回所有产品信息
POST /products    # 将新建产品信息
GET /products/4    # 将获取产品4
PUT /products/4    # 将更新产品4
```

2. 请求方式

- 访问同一个URL地址，采用不同的请求方式，代表要执行不同的操作
- 常用的HTTP请求方式有如下四种：

请求方式	说明
GET	获取资源数据(单个或多个)
POST	新增资源数据
PUT	修改资源数据
DELETE	删除资源数据

- 例如

```
GET /books       # 获取所有图书数据
POST /books       # 新建一本图书数据
GET /books/<id>/     # 获取某个指定的图书数据
PUT /books/<id>/     # 更新某个指定的图书数据
DELETE /books/<id>/    # 删除某个指定的图书数据
```

3. 过滤信息

- `过滤，分页，排序`：通过在url上传参的形式传递搜索条件
- 常见的参数：

```
?limit=10           # 指定返回记录的数量。
?offset=10           # 指定返回记录的开始位置。
?page=2&pagesize=100        # 指定第几页，以及每页的记录数。
?sortby=name&order=asc        # 指定返回结果按照哪个属性排序，以及排序顺序。
```

4. 响应状态码

- 重点状态码

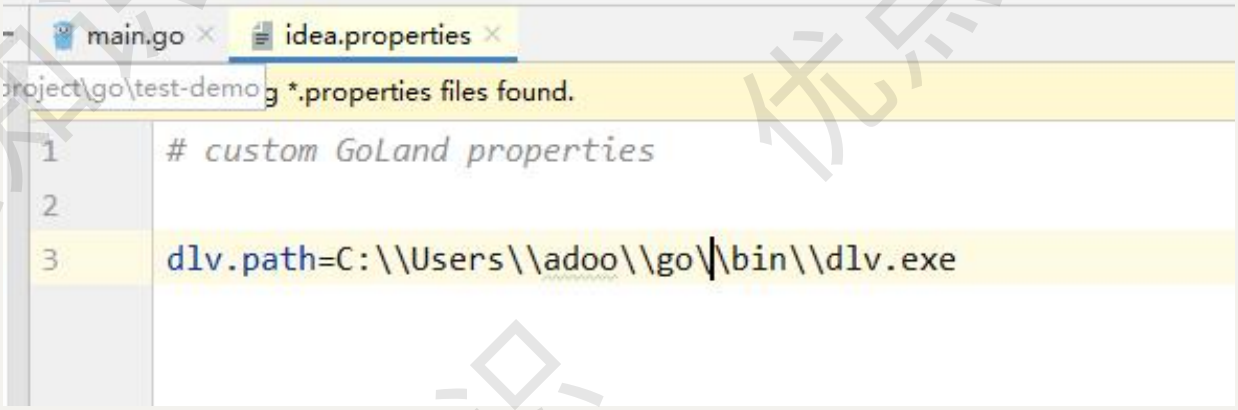
```
'''1. 2XX请求成功'''
# 1.1 200 请求成功，一般用于GET与POST请求
```

```
# 1.2 201 Created – [POST/PUT/PATCH]：用户新建或修改数据成功。
# 204 NO CONTENT – [DELETE]：用户删除数据成功。

'''3. 4XX客户端错误'''
# 3.1 400 INVALID REQUEST – [POST/PUT/PATCH]：用户发出的请求有错误。
# 3.2 401 Unauthorized – [*]：表示用户没有权限（令牌、用户名、密码错误）。
# 3.3 403 Forbidden – [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
# 3.4 404 NOT FOUND – [*]：用户发出的请求针对的是不存在的记录。

'''4. 5XX服务端错误'''
# 500 INTERNAL SERVER ERROR – [*]：服务器内部错误，无法完成请求
# 501 Not Implemented    服务器不支持请求的功能，无法完成请求

更多状态码参考：https://www.runoob.com/http/http-status-codes.html
```

六、Debug

1. 解决Version of Delve is too old问题

- `go get github.com/go-delve/delve/cmd/dlv`
- Goland->Help->Edit Custom Properties，点击后没有则创建
- 添加`dlv.path=C:\\Users\\adoo\\go\\bin\\dlv.exe`



- 其中`C:\\Users\\adoo\\go`是GOPATH，根据自身GOPATH路径去设置

- 重启Goland

2. debug示例