

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 Go lang 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					User表		Card表		
ID	Name	Card			Id	Name	Id	CardId	UserId
1	zhangsan	1001			1	zhangsan	1	1001	1
2	zhangsan	1002					2	1002	1
3	zhangsan	1003					3	1003	1
4	zhangsan	1004					4	1004	1
5	zhangsan	1005					5	1005	1
6	zhangsan	1006					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:MemberNumber"`
}

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

```
mysql> select * from users;
```

id	created_at	updated_at	deleted_at	username
1	2022-03-14 17:46:35.772	2022-03-14 17:46:35.772	NULL	zhangsan

```
1 row in set (0.00 sec)
```

```
mysql> mysql> select * from credit_cards;
```

id	created_at	updated_at	deleted_at	number	user_id
1	2022-03-14 17:46:35.772	2022-03-14 17:46:35.772	NULL	0001	1
2	2022-03-14 17:46:35.772	2022-03-14 17:46:35.772	NULL	0002	1

```
2 rows in set (0.00 sec)
```

可以看到在
credit_card表中会自动创建user_id字段与
user表外键关联

3. 一对多Association

- 查找关联
- 使用 Association 方法，需要把 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			Student		Lesson		StudentToLesson		
ID	Name	Lesson	ID	Name	Id	Lesson	Id	userId	lessonId
1	zhangsan	go	1	zhangsan	1	go	1	1	1
2	zhangsan	vue	2	lisi	2	vue	2	1	2
3	lisi	go					3	2	1
4	lisi	vue					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:UserReferID;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 | user_id | language_id | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| user_id | (20) unsigned | NO | PRI | NULL | |
| language_id | (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** 和 **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
}

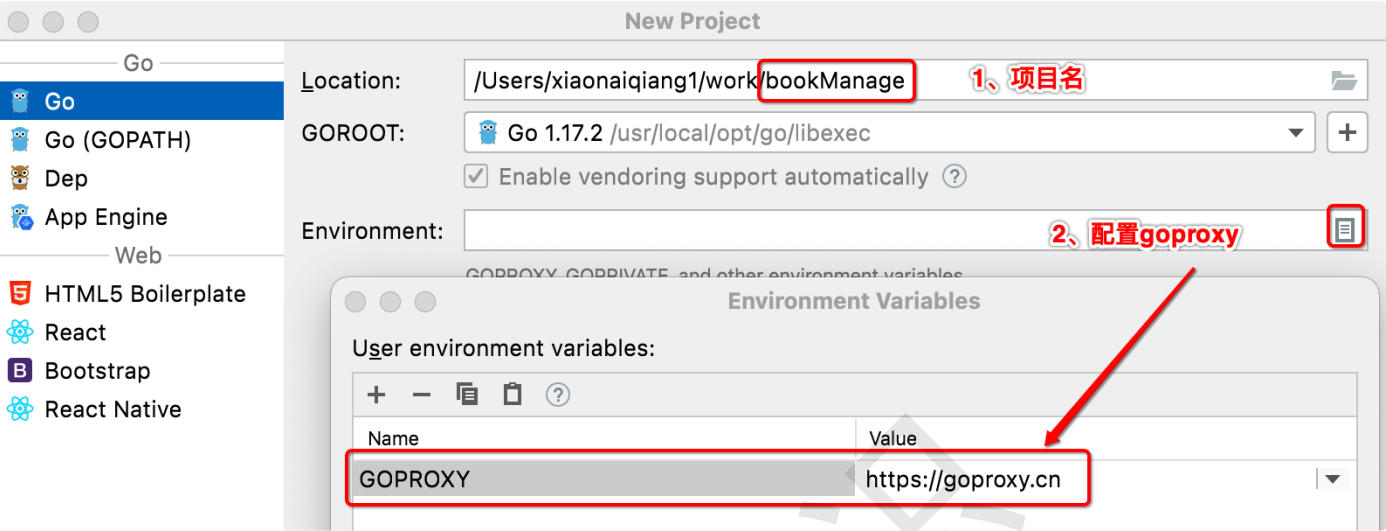
func main() {
    // 0、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/test_db?charset=utf8mb4&parseTime=True&loc=Local"
    db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    // 1、获取 name="zhangsan" 用户的地址
    persons := []Person{}
    db.Preload("Addresses").Find(&persons)
    strPersons, _ := json.Marshal(&persons)
    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）
│   └── 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
├── pkg          // 公共服务（所有模块都能访问的服务）
└── router      // 路由（路由分发）

```

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

```

4.2 添加路由分层

1. main.go

```
package main
import (
    "bookManage/router"
)
func main() {
    // 初始化路由分层
    r := router.InitRouter()
    r.Run(":8888")
}
```

2. router/init_router.go

```
package router
import (
    "github.com/gin-gonic/gin"
)
func InitRouter() *gin.Engine {
    r := gin.Default()
    TestRouters(r)
    // SetupApiRouters(r)
    return r
}
```

3. router/test_router.go

```
package router
import (
    "github.com/gin-gonic/gin"
)
func TestRouters(r *gin.Engine) {
    v1 := r.Group("/api/v1")
    v1.GET("test", TestHandler)
}
// 测试路由访问: http://127.0.0.1:8888/api/v1/test
func TestHandler(c *gin.Context) {
    c.String(200, "ok")
}
```

4.3 初始化mysql连接

1. main.go

```
package main
import (
    "bookManage/dao/mysql"
    "bookManage/router"
)
func main() {
    // 初始化mysql连接
    mysql.InitMysql()
    // 初始化路由分层
    r := router.InitRouter()
    r.Run(":8888")
}
```

2. dao/mysql/mysql.go

```
package mysql

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

var DB *gorm.DB

func InitMysql() {
    // 1、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/books?charset=utf8mb4&parseTime=True&loc=Local"
    db, err := gorm.Open(gmysql.Open(dsn), &gorm.Config{})
    if err != nil {
        fmt.Println("初始化mysql连接错误", err)
    }
    DB = db
}
```

4.4 定义多对多表结构

1. model/user.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. 自动生成表结构

dao/mysql/mysql.go

```
package mysql

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

```

    "gorm.io/gorm"
)

var DB *gorm.DB

func InitMysql() {
    // 1、连接数据库
    dsn := "root:1@tcp(127.0.0.1:3306)/books?charset=utf8mb4&parseTime=True&loc=Local"
    db, err := gorm.Open(gmysql.Open(dsn), &gorm.Config{})
    if err != nil {
        fmt.Println("初始化mysql连接错误", err)
    }
    DB = db
    // 自动创建表结构
    if err := DB.AutoMigrate(model.User{}, model.Book{}); err != nil {
        fmt.Println("自动创建表结构失败: ", err)
    }
}

```

4.5 注册登录

1. router/init_router.go

```

package router

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

func InitRouter() *gin.Engine {
    r := gin.Default()
    TestRouters(r)
    SetupApiRouters(r)
    return r
}

```

2. router/api_router.go

```

package router

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

func SetupApiRouters(r *gin.Engine) {
    r.POST("/register", controller.RegisterHandler)
    r.POST("/login", controller.LoginHandler)
}

```

3. controller/user.go

```

package controller

import (
    "github.com/gin-gonic/gin"
    "bookManage/dao/mysql"
    "bookManage/model"
    "github.com/google/uuid"
)

//注册
func RegisterHandler(c *gin.Context) {
    p := new(model.User)
    if err := c.ShouldBindJSON(p); err != nil {
        c.JSON(400, gin.H{
            "msg": err.Error(),
        })
    }
    //账号密码落库
    tx := mysql.DB.Create(p)
    if tx.Error != nil {
        c.JSON(400, gin.H{
            "msg": tx.Error,
        })
    }
    c.JSON(200, gin.H{
        "msg": "注册成功",
    })
    return
}

//登录
func LoginHandler(c *gin.Context) {
    p := new(model.User)
    if err := c.ShouldBindJSON(p); err != nil {
        c.JSON(400, gin.H{

```

```

        "msg": err.Error(),
    })
}
u := &model.User{
    Username: p.Username,
    Password: p.Password,
}
if rows := mysql.DB.Where(&u).First(&u).Row(); rows == nil {
    c.JSON(403, gin.H{
        "msg": "用户名密码错误",
    })
}
token := uuid.New().String()
mysql.DB.Model(u).Update("token", token)
c.JSON(200, gin.H{
    "token": token,
    "msg": "登录成功",
})
}
}

```

4. 测试注册功能

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

```

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

```

- postman测试

POST http://127.0.0.1:8888/register 请求路由

请求方式

Params Auth Headers (8) Body ● post请求数据在body

raw JSON 数据格式 Beautify

```
1 {
2   ... "username": "lisi",
3   ... "password": "123456"
4 }
```

请求数据

Body 200 OK 5 ms 188 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "msg": {
3     "id": 3,
4     "username": "lisi",
5     "password": "123456",
6     "token": ""
7   }
8 }
```

接口返回数据

5. 登录获取token

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

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

- postman测试

POST http://127.0.0.1:8888/login Send

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   ... "username": "lisi",
3   ... "password": "123456"
4 }
```

Body 200 OK 6 ms 171 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "token": "e0c17504-3796-4c18-8919-20b9fdc1c588"
3 }
```

4.6 图书管理

1. router/api_router.go

```
package router

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

func SetupApiRouters(r *gin.Engine) {
    r.POST("/register", controller.RegisterHandler)
    r.POST("/login", controller.LoginHandler)
    v1 := r.Group("/api/v1")
    v1.POST("book", controller.CreateBookHandler)
    v1.GET("book", controller.GetBookListHandler)
    v1.GET("book/:id", controller.GetBookDetailHandler)
    v1.PUT("book", controller.UpdateBookHandler)
    v1.DELETE("book/:id", controller.DeleteBookHandler)
}
```

2. controller/book.go

```
package controller

import (
    "bookManage/dao/mysql"
    "bookManage/model"
)
```

```

    "github.com/gin-gonic/gin"
    "net/http"
    "strconv"
)

// 新增
func CreateBookHandler(c *gin.Context) {
    p := &model.Book{}
    if err := c.ShouldBindJSON(p); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "msg": err.Error(),
        })
        return
    }
    // 数据库落库
    tx := mysql.DB.Create(&p)
    if tx.Error != nil {
        c.JSON(http.StatusInternalServerError, gin.H{
            "msg": tx.Error.Error(),
        })
        return
    }
    //
    c.JSON(http.StatusOK, gin.H{
        "msg": "书籍创建成功",
    })
}

// 查看列表
func GetBookListHandler(c *gin.Context) {
    books := []*model.Book{}
    tx := mysql.DB.Preload("Users").Find(&books)
    if tx.Error != nil {
        c.JSON(http.StatusInternalServerError, gin.H{
            "msg": tx.Error.Error(),
        })
        return
    }
    c.JSON(http.StatusOK, gin.H{
        "msg": "查询列表成功",
        "data": books,
    })
}

// 查看书籍详情
func GetBookDetailHandler(c *gin.Context) {
    idStr := c.Param("id")
    // string转换成数字
    bookId, _ := strconv.ParseInt(idStr, 10, 64)
    book := &model.Book{

```

```

        Id:    bookId,
    }
}
//数据库查询
tx := mysql.DB.Preload("Users").Find(&book)
if tx.Error != nil {
    c.JSON(http.StatusInternalServerError, gin.H{
        "msg": tx.Error.Error(),
    })
    return
}
c.JSON(http.StatusOK, gin.H{
    "msg": "查询书籍详情成功",
    "data": book,
})
}

//更新书籍
func UpdateBookHandler(c *gin.Context) {
    p := &model.Book{}
    if err := c.ShouldBindJSON(p); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "msg": err.Error(),
        })
        return
    }
    //数据库更新
    oldBook := &model.Book{Id: p.Id}
    newBook := &model.Book{Id: p.Id}
    if p.Name != "" {
        newBook.Name = p.Name
    }
    if p.Desc != "" {
        newBook.Desc = p.Desc
    }
    tx := mysql.DB.Model(&oldBook).Updates(&newBook)
    if tx.Error != nil {
        c.JSON(http.StatusInternalServerError, gin.H{
            "msg": tx.Error.Error(),
        })
        return
    }
    if len(p.Users) > 0 {
        tx := mysql.DB.Model(&newBook).Association("Users").Replace(p.Users)
        if tx.Error() != "" {
            c.JSON(http.StatusInternalServerError, gin.H{
                "msg": tx.Error(),
            })
            return
        }
    }
}
}

```

```

    c.JSON(http.StatusOK, gin.H{
        "msg": "更新书籍成功",
    })
}

// 删除
func DeleteBookHandler(c *gin.Context) {
    idStr := c.Param("id")
    // string转换成数字
    bookId, _ := strconv.Atoi(idStr, 10, 64)
    // 数据库删除, 删除book时, 同时删除第三张表中的用户对关系记录
    tx := mysql.DB.Select("Users").Delete(&model.Book{Id: bookId})
    if tx.Error != nil {
        c.JSON(http.StatusInternalServerError, gin.H{
            "msg": tx.Error.Error(),
        })
        return
    }
    c.JSON(http.StatusOK, gin.H{
        "msg": "删除书籍成功",
    })
}

```

3. 创建图书

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

```

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

```

4. 查看图书列表

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

```

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

```

5. 查看图书详情

- GET: <http://127.0.0.1:8888/api/v1/book/3/>
- 返回结果

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

6. 修改图书信息

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

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

7. 删除图书信息

- DELETE: <http://127.0.0.1:8888/api/v1/book/4/>

4.7 中间件身份验证

1. middleware/auth.go

```
package middleware

import (
    "github.com/gin-gonic/gin"
    "bookManage/dao/mysql"
    "bookManage/model"
)

func AuthMiddleware() func(c *gin.Context) {
    return func(c *gin.Context) {
        token := c.Request.Header.Get("token")
        var u model.User
        // 如果没有当前用户
        row := mysql.DB.Where("token = ?", token).First(&u).RowsAffected
        if row != 1 {
            c.JSON(403, gin.H{"msg": "当前token错误"})
            c.Abort()
            return
        }
    }
}
```

```

    }
    // 将当前请求的userID信息保存到请求的上下文c上
    c.Set("UserId", u.Id)
    c.Next()
}
}

```

2. router/api_router.go

```

package router

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

func SetupApiRouters(r *gin.Engine) {
    r.POST("/register", controller.RegisterHandler)
    r.POST("/login", controller.LoginHandler)
    v1 := r.Group("/api/v1")
    v1.Use(middleware.AuthMiddleware()) // 添加中间验证
    v1.POST("book", controller.CreateBookHandler)
    v1.GET("book", controller.GetBookListHandler)
    v1.GET("book/:id", controller.GetBookDetailHandler)
    v1.PUT("book", controller.UpdateBookHandler)
    v1.DELETE("book/:id", controller.DeleteBookHandler)
}

```

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`

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

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize JSON

```
{
  "msg": "当前token错误"
}
```

5. 携带token访问

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

The screenshot shows the Postman interface for a GET request to `http://127.0.0.1:8888/api/v1/book`. The Headers tab is active, showing a 'token' header with the value '16f65f89-622a-4b1f-a569-d2057c5d5d4f'. The Body tab is selected, displaying a JSON response:

```
{
  "books": [
    {
      "id": 3,
      "Name": "水浒传",
      "desc": "水浒传豪情满怀",
      "Users": [
        {
          "id": 2,
          "username": "zhangsan",
          "password": "123456",
          "token": "8cf14fe3-76f0-4268-b3f2-74e30e774929"
        }
      ]
    }
  ]
}
```


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。

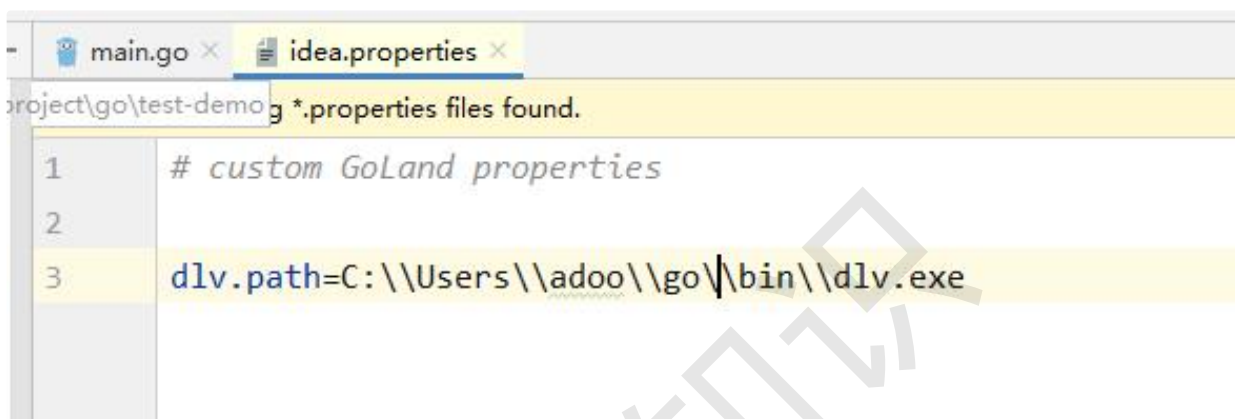
'''3. 4XX客户端错误'''
# 3.1 400 INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误。
# 3.2 401 Unauthorized - [*]: 表示用户没有权限（令牌、用户名、密码错误）。
# 3.3 403 Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
# 3.4 404 NOT FOUND - [*]: 用户发出的请求针对的是不存在的记录。

'''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
```

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\\adon\\go\\bin\\dlv.exe`



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

2. debug示例

