

# 第七节课 留言板实战

## 前言

在这节课开始之前，我们不妨先回忆一下我们已经学过的课程：

前四节课我们学习了golang的语法，大家由零基础到之后可以写出简单的小程序。

第六节课我们学习了gin框架，可以用来写注册登录等简单的后端接口。

第六节课我们学习了mysql的基础语法以及如何使用golang来操作mysql数据库，有了数据库我们就可以把我们想要的数据持久化——保存下来。

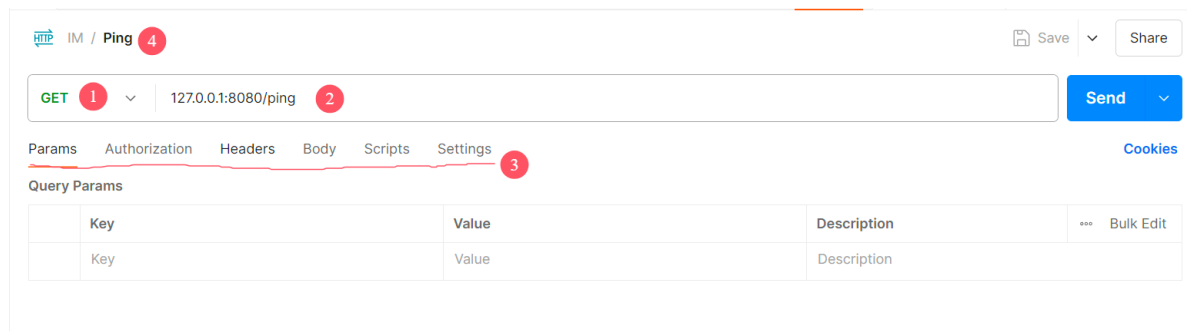
至此为止，作为一个后端开发人员所需要掌握的最基础的知识已经学完了，但是掌握了知识并不代表你就可以熟练应用这些知识了，这节课就带领大家学习一下如何利用前几课的知识，完成一份后端项目：留言板。

## 问题解决

在讲本节课的内容之前，我们先就之前同学们提出的一些问题进行解决

## 如何使用postman/apifox

postman接口操作界面：



①是方法，常用的POST GET DELETE PUT

②是URL，即网页的连接所有都会显示在网页的链接栏上，最好不要放一些信息，过于暴露信息了，十分不安全

③是HTTP请求的不同部分

- **Params**：是包含在请求URL的查询参数
- **Authorization**：是配置请求的授权或者令牌（token），常用于鉴权
- **Headers**：是HTTP头部的信息
- **Body**：是HTTP请求主体的内容，我们传到服务器的参数常在这使用
- **Scripts**：是代码编辑器，常用于写脚本
- **Settings**：是常用的设置

注意事项：params常用于Get请求而不是post请求，想一想都能明白一些传入的参数一般是隐私，如果暴露在url上，嗨客都不需要爆破就能得知你的信息了（

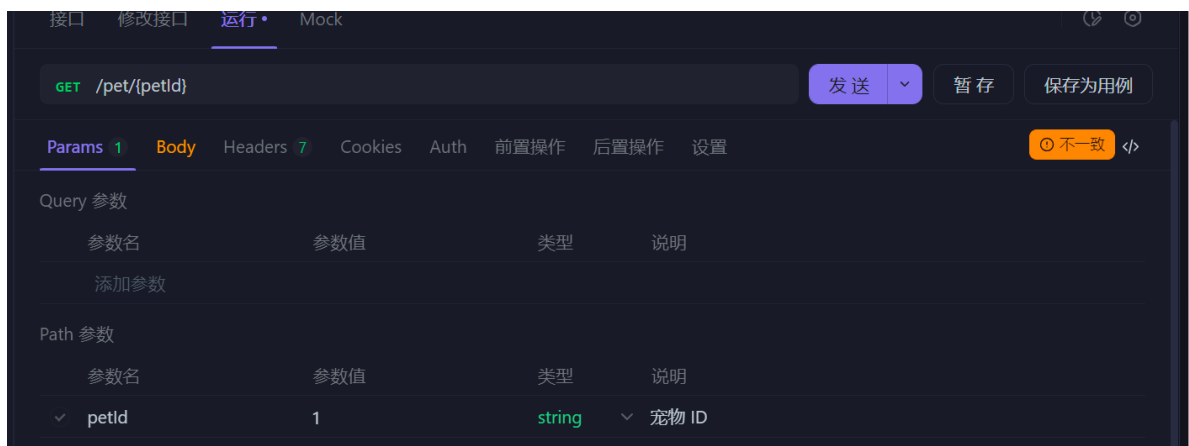
**Body部分：**

Params Authorization Headers (6) Body Scripts Settings

☒ none ☐ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

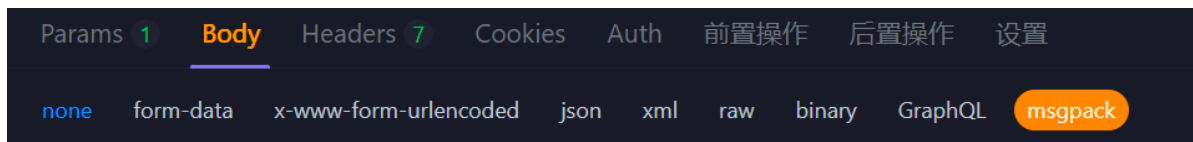
- **none**：没有指定特定的数据格式
  - 适用于 **GET** 请求，或者不需要请求体的请求
- **form-data**：常用于HTML表单提交的数据，数据以键值对的形式编码，通常用于上传文件或简单表单数据
  - 文件上传、表单提交
- **x-www-form-urlencoded**：编码表单数据的方式,数据编码在URL中
  - 简单的表单提交，没有文件上传
- **raw**：数据采用原始、非格式化的格式,如 **JSON**、**XML**、**HTML**
  - 提交 JSON、XML、HTML 或纯文本等
- **binary**：用于二进制数据,如图像、文档或其他非文本内容
  - 上传文件，如图像、音频、PDF 文件等
- **GraphQL**：一种API查询语言,允许客户端请求所需的确切数据
  - 与支持 GraphQL 的 API 进行交互

## Apifox操作界面



与Postman不同的是，他要修改方法需要在修改接口内进行修改

### Body部分：



同上，只是多了个msgpack

- **msgpack**：一种高效的二进制序列化格式，类似于JSON，但比JSON 更小、更快。它通过二进制编码数据，减少了存储空间，并提高了数据传输的速度。常用于需要频繁传输大量数据或对性能有要求的场景。

常用的也就只有 **Params form-data x-www-form-urlencoded raw**

# 包的概念

## 包的基本概念

### 1. 包是代码的组织单位：

- Go 使用包来组织代码。每个 `.go` 文件都属于某个包，该包包含了一组相关的函数、结构体、接口等。
- 一个包的文件都应该存放在同一个文件夹中，并且包名和文件夹名通常是相同的。

### 2. 标准库：

- Go 提供了一个非常丰富的标准库，里面包含了许多功能强大的包，如 `fmt`、`net/http`、`os` 等。
- 这些标准库包大多数已经满足了常见的编程需求。

### 3. 第三方包：

- 除了 Go 的标准库，你也可以使用其他开发者发布的包。这些包可以通过 Go 的包管理工具（如 `go get`）来安装并使用。

### 4. 每个包必须有一个导出标识符：

- 包中的标识符（如变量、函数、类型）只有以大写字母开头时，才可以被其他包访问（即导出）。以小写字母开头的标识符则只能在当前包内使用。

## 包的结构和使用

### 1. 包的声明：

- 每个 `.go` 文件的开头都需要声明它属于哪个包，通常这行声明是文件的第一行。
- 示例：

```
1 | package main
```

### 2. 导入包：

- 通过 `import` 语句可以将其他包导入到当前包中，导入的包可以使用该包提供的功能（函数、类型、变量等）。
- 示例：

```
1 | import "fmt"
```

### 3. 创建和使用包：

- 可以创建自定义包，将相关功能组织在一个文件夹中。
- 包名通常与文件夹名一致，例如一个名为 `math` 的包，通常应该放在 `math` 文件夹下。

### 4. 包的导出和访问：

- Go 中的包可以通过导出标识符（大写字母开头的函数、类型、变量等）来让其他包使用。
- 示例：

```

1 package utils
2
3 import "fmt"
4
5 // 导出的函数
6 func PrintMessage(message string) {
7     fmt.Println(message)
8 }

```

## 5. 使用导入的包：

- 其他包可以通过 `import` 语句导入并使用 `utils` 包中的 `PrintMessage` 函数

```

1 package main
2
3 import (
4     "fmt"
5     "your_project/utils" // 导入自定义包
6 )
7
8 func main() {
9     utils.PrintMessage("Hello, Go!") // 使用 utils 包中的函数
10 }
11

```

## 包的导入路径

Go 中的包导入是基于文件系统路径的，也可以使用远程路径（如 GitHub 上的公开库）：

- 本地包导入：`import "your_project/pkg/mypackage"`
- 第三方包导入：`import "github.com/someone/somelib"`

## 示例：创建和使用自定义包

- 创建包：**创建一个新的文件夹 `mathutil`，并在其中创建一个 Go 文件 `mathutil.go`，并写入以下内容：

## Go 包的导入和管理

### 1. `go.mod` 和模块管理：

- 从 Go 1.11 开始，Go 支持模块（module）系统来管理依赖关系。`go.mod` 文件用于定义和管理 Go 项目的模块路径和版本。**导入第三方包的时候文件目录下必须存在 `go.mod` 文件**

### 2. 安装第三方包：

- 可以使用 `go get` 命令来安装第三方包，Go 会自动处理包的版本和依赖关系。

```

1 go get github.com/sirupsen/logrus

```

### 3. 导入第三方包：

- 安装包后，可以通过 `import` 语句导入并使用它：

```

1 import "github.com/sirupsen/logrus"

```

### 4. 包的命名：

- 可以在导入包的前面加上名字，修改在当前包的引用名，从而避免一个冲突

```
1 import ctx "context" //在当前包下引用这个包内的内容为 ctx.xxx
2 import _ "github.com/go-sql-driver/mysql" // 不直接在代码中引用包中的任何
    函数、类型或变量,通常用来执行包中的初始化逻辑
```

## 数据库的连接参数

```
1 //常见的数据库链接
2 dns := root:114514@tcp(localhost:3306)/homo?
    charset=utf8mb4&parseTime=True&loc=UTC
```

### 链接参数是什么

链接参数就是 `dns` 的 `?` 后面的东西，例如这个例子的参数就是 `charset`、`parseTime`、`loc`

### 常见的链接参数

#### 1. `charset=utf8mb4`

- **作用：**指定数据库连接的字符集，`utf8mb4` 是 MySQL 的字符集，支持完整的 Unicode 字符集，包括表情符号、亚洲字符等。

#### 2. `parseTime=True`

- **作用：**用于解析时间字段。如果设置为 `True`，数据库返回的 `DATETIME` 和 `TIMESTAMP` 类型的字段会被转换为 Go 语言的 `time.Time` 类型。这样可以避免在 Go 代码中手动解析时间字段。

#### 3. `loc=UTC`

- **作用：**设置时区为 UTC。数据库连接时会使用 UTC 时区进行时间存储和计算。这样可以避免因不同地区的时区差异导致的时间错误。

#### 4. `allowAllFiles=True`

- **作用：**允许加载所有文件（如用于加载外部文件到数据库）。某些数据库客户端可能会限制加载文件，`allowAllFiles=True` 可以解除这个限制。

#### 5. `timeout=30s`

- **作用：**设置连接的超时时间。例如，`timeout=30s` 表示如果连接超过 30 秒没有响应，则会抛出超时错误。

#### 6. `ssl-mode=REQUIRED`

- **作用：**指定数据库连接是否使用 SSL 加密。`ssl-mode=REQUIRED` 表示强制使用 SSL 连接。

#### 7. `maxAllowedPacket=16777216`

- **作用：**设置数据库允许的最大数据包大小。此参数有助于调整处理大数据时的限制，避免过大的查询或更新操作失败。

#### 8. `autocommit=true`

- **作用：**设置数据库连接是否启用自动提交事务。如果为 `true`，每次查询后都会自动提交；如果为 `false`，需要手动提交事务。

最常用的就是 `charset=utf8mb4&parseTime=True&loc=UTC`，尤其是时区（有学姐就因为这个时区没有写而犯了错），如果时区没有设置可能会导致时间记录不一样从而导致出错

## 代码规范

同学们写的作业，代码阅读的人可能比较少，只有你自己和我们学姐

当然如果有的同学自己有在github或者其他什么网站的浏览过其他人的项目代码，就知道代码规范的重要性

尤其是开源项目，一般都会专门有个文档对代码进行规范

但是如果接手开发就知道了，一个项目不是一个人去书写的，而是有很多的程序员一起书写的，试想每个人按照自己的个性去书写代码，代码风格不一样，让后续的人去维护代码变得更加困难甚至有人可能会一边写一边吐槽你。所以为了避免留下的代码被人骂所以为了能够更好的维护代码，代码规范就显得尤为重要。

## 文档说明

项目的根目录下应该存在一个 `README.md` 文件，其中应该包含——

- 项目的基本介绍
- 项目的结构
- 使用的技术

同时作为后端的我们，还需要书写**接口文档**，同样常以 `md` 文件进行书写

这时可能就有同学会问了，学姐学姐 `md` 文件是什么？

## markdown

`md` 文件是一种以 `markdown` 语法书写的文件

`Markdown` 是一种轻量级标记语言，使用简单易读易写的语法来格式化文档。它最初由 [John Gruber](#) 和 Aaron Swartz 于 2004 年共同设计，旨在让普通文本更容易转换为 HTML 等格式。

### Markdown 的特点

- **易读性**：即使不转换，原始文本也能清晰表达内容。
- **轻量级**：简单的标记符号即可实现复杂的格式效果。
- **兼容性**：可以轻松转换为 HTML、PDF 等格式。
- **广泛应用**：被广泛用于写作、技术文档、博客、README 文件等。

学姐们好看的pdf课件都是用的 `markdown` 书写的，然后在导出成 `pdf` 文档给你们看的，不过学了这门课后pdf可能就会比较少的出现了，毕竟都教你们了

`markdown` 的语法并不难，我就不过多赘述了

作为一名后端的选手，`markdown` 的基础语法是必须要求会使用的，大家要下去学习一下怎么书写，也不用过于深奥只需要了解怎么写出一个**接口文档**就ok了

[markdown教程](#)

### 可以写markdown的软件

- typora（收费，但是我只能说各凭本事）
- goland
- vscode插件
- 语雀（蚂蚁）
- 飞书（字节，这个很好用说实话）
- 腾讯文档（腾讯）

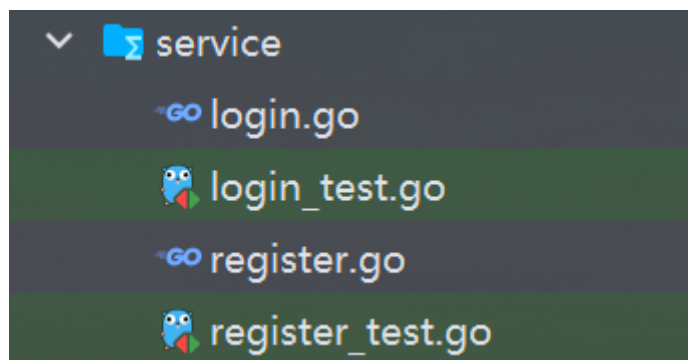
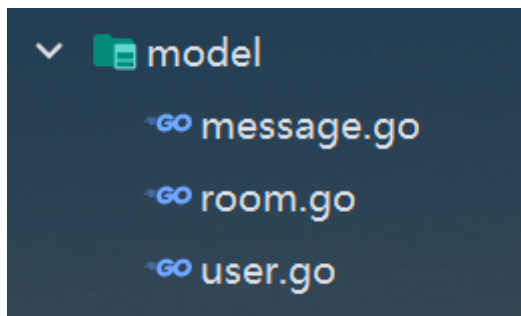
# 如何规范书写

## 命名规范

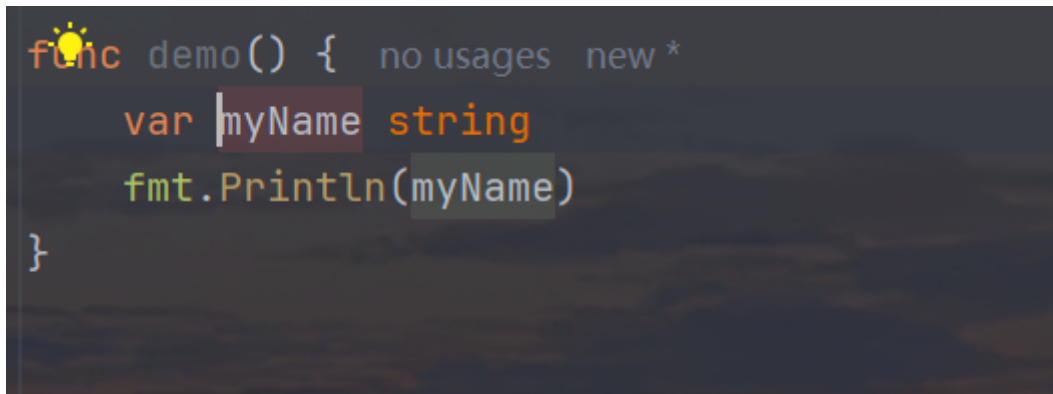
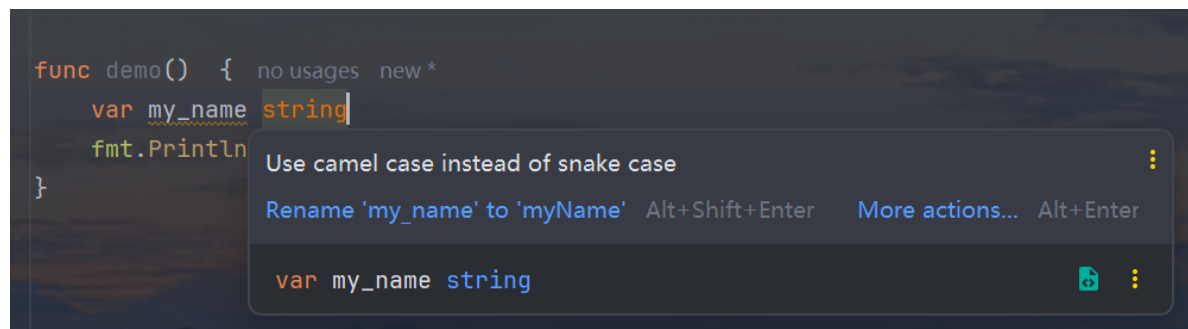
为了能够规范代码，开发中无论是**你的项目名**还是**内部的函数方法结构体等等**，请都最好使用英文命名

- 项目名应全部为小写英文字母，分隔符请使用 - 连接，例如：message-board
- 项目目录命名（文件名）应全部为小写英文字母，分割符使用下划线 \_ 连接，如go源代码中的：  
runtime2\_lockrank.go
- 函数与变量名应该使用英文驼峰式命名法
- package 包名应尽量使用单个单词表达含义

example



当然如果你使用的是goland，如果你的命名是有问题的他还会提醒你，你也可以使用其帮你规范命名



# 切勿if-else无限嵌套

## example

下图是某人的代码可以看到使用了很多 if-else 进行无限嵌套，这样的代码可读性及其的差，对于后续的代码维护者是要到骂娘的地步。

```
if err != nil {
    c.JSON(consts.StatusBadRequest, map[string]string{
        "error": err.Error(),
    })
    return
}
query := "SELECT name,password FROM students WHERE username=?"
rows, err := Db.Query(query, student.Username)
if err != nil {
    c.JSON(consts.StatusBadRequest, map[string]string{
        "error": err.Error(),
    })
    return
}
if !rows.Next() {
    c.JSON(consts.StatusBadRequest, map[string]string{
        "error": "用户名不存在!",
    })
    return
} else {
    err = rows.Scan(&username, &oldPassword)
    if err != nil {
        c.JSON(consts.StatusBadRequest, map[string]string{
            "error": err.Error(),
        })
        return
    }
    if student.OldPassword != oldPassword {
        c.JSON(consts.StatusBadRequest, map[string]string{
            "error": "旧密码错误, 修改失败!",
        })
        return
    } else {
        query = "UPDATE students SET password=? WHERE username=?"
        result, err := Db.Exec(query, student.NewPassword, student.Username)
        if err != nil {
            c.JSON(consts.StatusBadRequest, map[string]string{
                "error": err.Error(),
            })
        }
    }
}
```

## if-else嵌套的坏处:

- 代码可读性差
- 维护困难
- 逻辑错误的风险增加
- 执行效率低
- 不符合单一职责原则
- 可测试性差
- 增强了代码的耦合度

为了避免出现 if-else 嵌套情况的出现，可以用如下的方式去解决问题

1. **提取函数**：将复杂的逻辑提取到独立的函数或方法中，使代码逻辑更加清晰。例如，长嵌套的 if-else 语句可以拆分成多个更简洁、含义明确的小函数。
2. **使用早期返回**：通过早期返回（early return）的方式避免过多的嵌套。这样可以减少代码的层级，使得每个条件判断都能直接给出结果。

## example:



```
1  if condition1 {
2      return result1
3  }
4  if condition2 {
5      return result2
6  }
7  // 继续其他的条件判断
```

## 分层架构

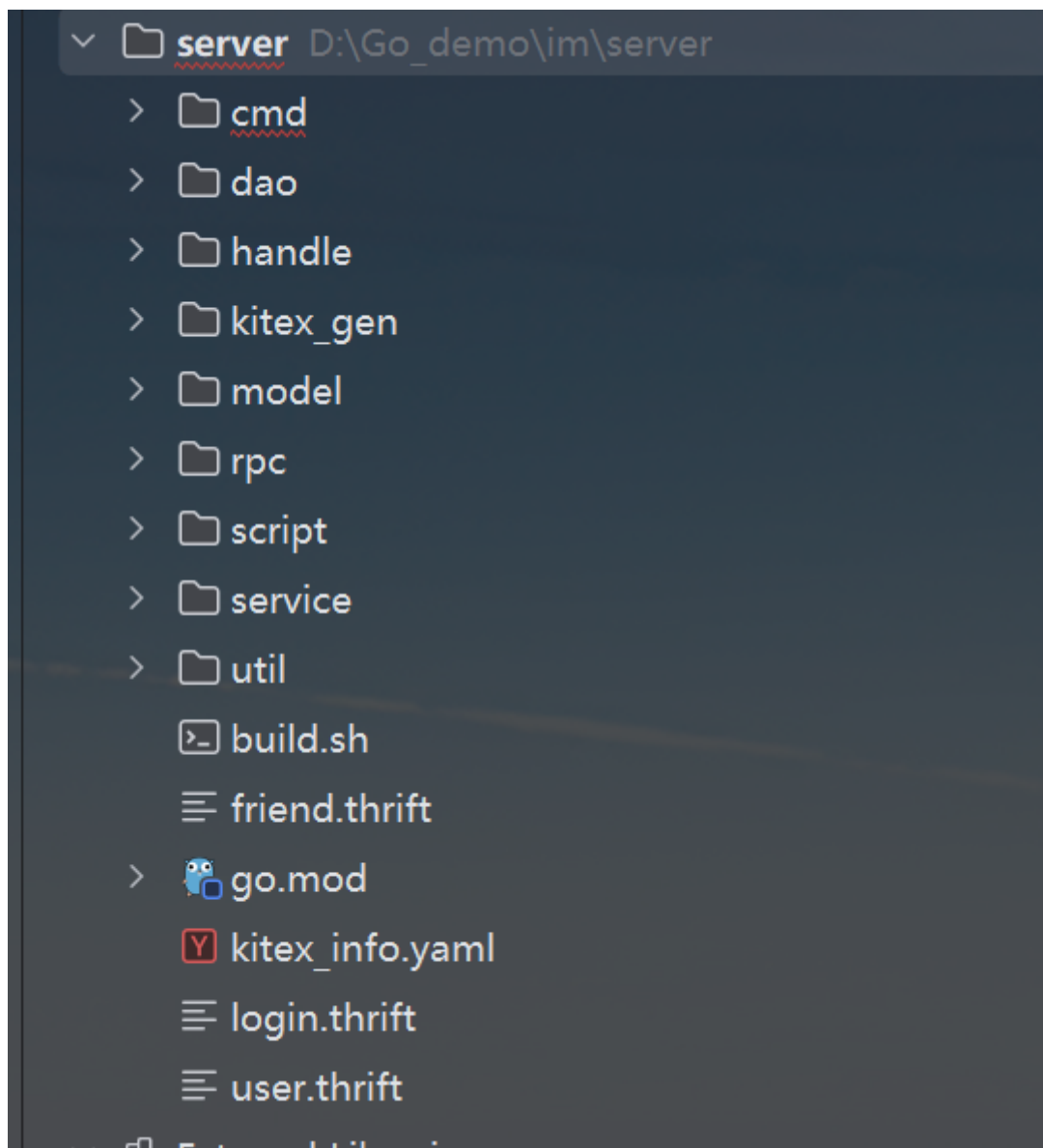
对于一个Go应用程序项目，肯定不只有几个 `.go` 的源文件，在本节课之前大部分人的代码架构是酱紫的格式

```
1  .
2  └─project
3      |   lv1.go
4      |   lv2.go
5      |   lv3.go
6      |
7      └─.idea
```

当然如果你只是为了写一个猜数、计算器等等只使用一个 `main.go` 文件绰绰有余，但是随着你写代码的时间的增长，所写的项目越来越复杂，**你不可能把所有的代码都写到一个文件当中**，你就需要将不同类型的代码分布在不同的文件、文件夹和包中，通过良好的命名，我们甚至可以在不去看具体的代码实现的情况下，仅仅通过包名和文件名就能判断我们在做的事情。这时候你就需要一个良好的代码结构习惯，不然你最终会得到一个凌乱的代码，这就包含大量隐藏的依赖项和全局状态。

### *example*

就像这个我之前搓的demo一样，每个项目都按照作用进行了分区（虽然这个代码只有我一个人在写）但是这种思想很重要



每一种文件都有其对应的位置，而不是全部写在一起。

试想一下当你遇到一个bug的时候，看着日志的定位你更想看到的是main.go:1145 and main.go:14 has some error 还是想看到 dao/user.go:11 and cmd/main.go:45 has error 好

### 分层架构的作用

- `cmd` 仅存放 `main.go` 文件，作为整个项目的入口。
- `api` 存放路由 `router.go`，以及将不同的接口函数按照性质放在不同的文件里，比如 `user.go` 里面存放有关 `user` 相关的接口。`comment.go` 中存放 `comment` 相关接口。这一层是负责参数校验，请求转发。
- `service` 将各个服务封装之后供各个接口调用。可以认为从这里开始，所有的请求参数一定是合法的，业务逻辑和业务流程也都在这一层中。举例来说，比如注册接口需要判断用户名是否重复，那么我们需要写一个 `func IsRepeatUserName(userName string)(bool,error){}` 的函数，放在 `service` 文件夹下的 `user.go` 文件中。
- `model` 存放项目中所有的 `struct`，按照文件名归类。
- `dao` 数据访问层，全称为 `Data Access Object`，所有数据库相关操作全部封装在这一层。将下层存储以更简单的函数、接口形式暴露给 `service` 层来使用。
- `utils` 文件通常用于包含各种辅助函数和工具函数，这些函数在项目中的多个地方可能都会用到。封装代码以提高代码的可维护性和可读性，减少代码重复

至于其他的那就后面再谈，目前来说这几个分层是现有阶段常用的

# 接口文档 (Restful Api规范)

## 什么是接口文档



接口文档 (API 文档) 是用于描述一个系统、模块或应用程序接口的文档。接口文档通常包含接口的详细信息，包括如何访问、请求和响应的格式、参数的含义以及返回的结果等。它为开发人员、测试人员和使用该接口的其他系统或用户提供了操作指南。

通俗一点说就是在产品给我们的需求的基础上，所开发出的接口

一个标准的接口文档需要有如下的内容（结合你的接口是否存在在进行增删）：

1. **接口名称**：描述接口的功能和用途。
2. **接口地址**：具体的URL或端点（Endpoint）。
3. **请求方式**：如 GET、POST、PUT、DELETE 等 HTTP 请求方式。
4. **请求参数**：包括参数名称、类型（如字符串、整数等）、是否必填、默认值和说明。
5. **响应参数**：接口返回的数据结构，包括字段、类型及含义。
6. **请求示例**：演示如何发送请求。
7. **响应示例**：提供接口响应的示例，方便理解返回数据的结构和内容。
8. **错误示例**：接口可能返回的错误代码及其含义。

同时你的接口文档需要符合Restful API规范。

## 什么是Restful API

**RESTful API** 是一种基于 REST（Representational State Transfer，表述性状态转移）架构风格的应用程序接口，它通过 HTTP 协议进行数据交互，广泛用于 Web 服务开发。RESTful API 遵循一套设计规则，使得接口更加清晰、简洁、易于理解和维护。

定义了通用的URI语法：

```
1 | URI = scheme "://" authority "/" path [ "?" query ] [ "#" fragment ]
```

- scheme: 指底层用的协议, 如http、https、ftp
- host: 服务器的IP地址或者域名
- port: 端口, http中默认80
- path: 访问资源的路径, 就是咱们各种web 框架中定义的route路由
- query: 为发送给服务器的参数
- fragment: 锚点, 定位到页面的资源, 锚点为资源id

### 一般只使用如下方法进行操作

- GET: 获取资源
- POST: 新建资源
- PUT: 在服务器更新资源 (向客户端提供改变后的所有资源)
- PATCH: 在服务器更新资源 (向客户端提供改变的属性)
- DELETE: 删除资源

### example

请求方法	URL	含义
GET	/view_message或者/get_message	获取信息
POST	/insert_message或者/create_message	插入信息
POST	/update_message	更新信息
POST	/delete_message	删除信息

如果我们的API遵循 REST 风格, 就会设计成以下格式, 很明显下面遵循RESTful API的接口更加美观

请求方法	URL	含义
GET	/message	获取信息
POST	/message	创建信息
PUT	/message	更新信息
DELETE	/message	删除信息

推荐阅读[阮一峰 理解RESTful架构](#)

## 接口文档如何书写

1. 首先需要在你的接口文档最前面放入对这个项目文档的简要概述
2. 然后就是你所书写的api
  - 最前面需要说明这个api所接受的method (GET、POST、DELETE、PUT.....)
  - 然后是这个接口的名字即作用
    - 也可以对这个接口的作用进行概述
  - 第三个要写的是api的路径
  - 接着就是书写这个api的请求方式、请求体的类型

■ **example:**

**请求方式:**

- POST
  - application/json //是什么就写什么
  - Authorization: Bearer \$token // 鉴权相关
- 第五个就是这个接口所需要的参数，并且带上请求示例（如是json）

■ **example**

**请求参数:**

请求参数	类型	是否必须	说明
xxx	string	yes	xxxx

**请求示例:**

```
1 //json的请求示例
2 {
3     "xxx": "xxxxx"
4 }
```

- 然后就是你的返回参数，并且还要有关于这个返回参数所定义的**状态码**，和详细信息

■ **example**

**返回参数:**

返回参数	类型	说明
111	string	xxx

**返回示例:**

```
1 {
2     "status":10000,
3     "info":xxx,
4     "data":{
5         "111": "xxxx"
6     }
7 }
```

3. 最后需要把该api会返回的所有情况的 json 都写下来，并且标注是什么情况导致的

**example**

**错误示例:**

**内部服务出错:**

```
1 {
2     "status":50000,
3     "info": "internal error"
4 }
```

# 状态码

接口文档的状态码一般由自己定义，但是也尽量不要超脱状态码。

自定义状态码的好处就是可以根据服务端返回的状态码，一瞬间定位到错误的位置，而不需要进行长时间的自检，去寻找bug出现的地方，能够使得代码维护变得更加简单。

而我接下来所讲的状态码是用于表示服务器对客户端请求的处理结果的三位数字代码。

状态码由三位数字组成：

- **第一位数字**：表示响应的类型（即类别）。
- **第二位和第三位数字**：提供更具体的响应信息，通常表示响应的详细情况。

常见的HTTP状态码有以下 几类：

- **1xx：信息性状态码**
  - 这些状态码表示请求已被接受，但尚未处理。它们通常用于告知客户端可以继续发送请求或需要等待进一步的响应。
- **2xx：成功状态码**
  - 这些状态码表示请求已经成功地被服务器处理并响应。
- **3xx：重定向状态码**
  - 这些状态码表示客户端需要进一步操作才能完成请求，通常是指重定向到新的 URL。
- **4xx：客户端错误状态码**
  - 这些状态码表示客户端发送的请求存在错误，服务器无法处理请求。
- **5xx：服务器错误状态码**
  - 这些状态码表示服务器在处理请求时发生了错误，通常是由于服务器本身的故障或配置问题。

更为详细的状态码可以去移步至[httpcat](http://httpcat.com)查看

以下是go在他的标准包net/http定义的状态码

```
1  const (
2      StatusContinue           = 100 // 请求者应继续进行请求
3      StatusSwitchingProtocols = 101 // 服务器已理解请求，并将通过升级协议进行切换
4      StatusProcessing         = 102 // 服务器已接受请求，但尚未处理完成
5      StatusEarlyHints         = 103 // 用于在请求开始处理之前提前提示某些信息
6      StatusOK                 = 200 // 请求成功，服务器已返回请求的数据
7      StatusCreated            = 201 // 请求成功，且已创建新的资源
8      StatusAccepted           = 202 // 请求已接受，但处理未完成
9      StatusNonAuthoritativeInfo = 203 // 请求成功，但返回的元数据可能非原始来源
10     StatusNoContent          = 204 // 请求成功，但无返回内容
11     StatusResetContent       = 205 // 请求成功，且要求请求者重置视图
12     StatusPartialContent     = 206 // 服务器仅部分返回了请求的数据
13     StatusMultiStatus         = 207 // 返回多状态响应，用于WebDAV
14     StatusAlreadyReported     = 208 // 重复的资源已在多状态响应中列出
15     StatusIMUsed              = 226 // 服务器已执行IM扩展操作
16     StatusMultipleChoices     = 300 // 请求有多个可供选择的资源
17     StatusMovedPermanently    = 301 // 请求的资源已永久移动到新位置
18     StatusFound               = 302 // 请求的资源临时位于其他位置
19     StatusSeeOther            = 303 // 请求者应使用GET方法获取资源
20     StatusNotModified         = 304 // 请求的资源未修改，可以使用缓存
21     StatusUseProxy            = 305 // 必须通过代理访问请求的资源
22     StatusTemporaryRedirect   = 307 // 请求的资源临时位于其他位置，但必须使用原请求方
```

法

23	StatusPermanentRedirect	= 308 // 请求的资源永久重定向至新位置
24	StatusBadRequest	= 400 // 服务器无法理解请求的语法
25	StatusUnauthorized	= 401 // 请求需要身份验证
26	StatusPaymentRequired	= 402 // 预留状态码，暂未使用
27	StatusForbidden	= 403 // 服务器拒绝请求
28	StatusNotFound	= 404 // 服务器找不到请求的资源
29	StatusMethodNotAllowed	= 405 // 请求的方法被禁止
30	StatusNotAcceptable	= 406 // 请求的内容特性不被接受
31	StatusProxyAuthRequired	= 407 // 请求要求代理身份验证
32	StatusRequestTimeout	= 408 // 服务器等待请求超时
33	StatusConflict	= 409 // 请求冲突
34	StatusGone	= 410 // 请求的资源已永久删除
35	StatusLengthRequired	= 411 // 请求缺少Content-Length头信息
36	StatusPreconditionFailed	= 412 // 前置条件失败
37	StatusRequestEntityTooLarge	= 413 // 请求实体过大
38	StatusRequestURITooLong	= 414 // 请求的URI过长
39	StatusUnsupportedMediaType	= 415 // 不支持的媒体类型
40	StatusRequestedRangeNotSatisfiable	= 416 // 请求范围无效
41	StatusExpectationFailed	= 417 // 服务器无法满足请求者的期望
42	StatusTeapot	= 418 // 服务器拒绝调制咖啡
43	StatusMisdirectedRequest	= 421 // 请求错误指向的服务器
44	StatusUnprocessableEntity	= 422 // 请求格式正确，但无法处理
45	StatusLocked	= 423 // 请求的资源被锁定
46	StatusFailedDependency	= 424 // 请求因依赖的请求失败
47	StatusTooEarly	= 425 // 请求太早，需稍后再试
48	StatusUpgradeRequired	= 426 // 客户端需升级协议
49	StatusPreconditionRequired	= 428 // 请求需满足条件
50	StatusTooManyRequests	= 429 // 请求过多，需限速
51	StatusRequestHeaderFieldsTooLarge	= 431 // 请求头字段过大
52	StatusUnavailableForLegalReasons	= 451 // 因法律原因无法提供资源。
53	StatusInternalServerError	= 500 // 服务器遇到未知错误
54	StatusNotImplemented	= 501 // 服务器不支持请求的功能
55	StatusBadGateway	= 502 // 网关或代理从上游服务器收到无效响应
56	StatusServiceUnavailable	= 503 // 服务器临时过载或维护中
57	StatusGatewayTimeout	= 504 // 网关或代理等待上游服务器超时
58	StatusHTTPVersionNotSupported	= 505 // 服务器不支持请求所用的HTTP协议版本
59	StatusVariantAlsoNegotiates	= 506 // 服务器配置错误
60	StatusInsufficientStorage	= 507 // 服务器无法存储请求所需内容
61	StatusLoopDetected	= 508 // 检测到循环请求
62	StatusNotExtended	= 510 // 需要进一步扩展
63	StatusNetworkAuthenticationRequired	= 511 // 需要网络认证

当然你也可以自定义状态码，一般来说实际开发中都是自定义状态码的，毕竟有着不同的服务，不同的状态码才可以加快定位的速度

而我们常见的状态码有

- **200 OK**: 表示请求成功，可以返回请求的资源或数据
- **301 Moved Permanently**: 请求的资源已被永久移动到新位置。后续请求应使用新的 URL。
- **302 Found**: 表示临时重定向到了另一个URL。
- **400 Bad Request**: 请求参数错误（如缺少必要参数或参数格式错误）。
- **401 Unauthorized**: 请求未授权，客户端需要提供身份认证信息。常见于需要登录的 API。
- **403 Forbidden**: 服务器理解请求，但拒绝执行。客户端无权访问请求的资源，认证通过，但没有权限访问指定资源（如没有管理员权限）。
- **404 Not Found**: 表示请求的资源不存在，可以返回一个自定义的错误页面或信息。



- **408 Request Timeout**: 请求超时，服务器等待客户端的数据超时。常见于长时间未响应的请求。
- **500 Internal Server Error**: 服务器遇到未预期的错误，无法完成请求。这个错误通常表示程序代码的问题。
- **502 Bad Gateway**: 服务器作为网关或代理，接收到上游服务器返回的无效响应。可能是上游服务器的问题。
- **503 Service Unavailable**: 服务器暂时无法处理请求，通常是由于过载或维护。客户端应稍后重试。

## git的操作和使用

[git的官网](#)可以自行点击进行安装，网上的安装教程很多，我也就不去一一赘述了；

## 如何将代码上传至github

1. 下载并安装好git，并为你的git做好相关配置

```
1 git config --global user.name "你的名字"
2 git config --global user.email "你的邮箱(最好和你当前使用的github为同一个)"
```

2. 首先登录你的github，并且创建一个仓库，[GitHub](#)的官网，可能会很卡，我只能说各显神通吧。


**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

---

*Required fields are marked with an asterisk (\*).*


Owner \* / Repository name \*


 /

Great repository names are short and memorable. Need inspiration? How about **didactic-sniffle** ?

设置公开我们才能看

Description (optional)

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

---

Initialize this repository with:

☐ Add a README file  
This is where you can write a long description for your project. [Learn more about READMEs.](#)


Add .gitignore

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

---

 You are creating a public repository in your personal account.

3. 进入你的项目，并打开终端输入如下命令



```
1 git init //把这个目录变成Git可以管理的仓库
2 git add . //不但可以跟单一文件，还可以跟通配符，更可以跟目录。一个点就把当前目录下所有未追踪的文件全部add了
3 git commit -m "xxx(填本次提交所作的事)" //把文件提交到仓库
4 git remote add origin git@github.com:wangjiax9/practice.git //关联远程仓库
5 git push origin master(根据分支所判断如果是master就写master，如果是main就写main无伤大雅) //把本地库的所有内容推送到远程库上
```

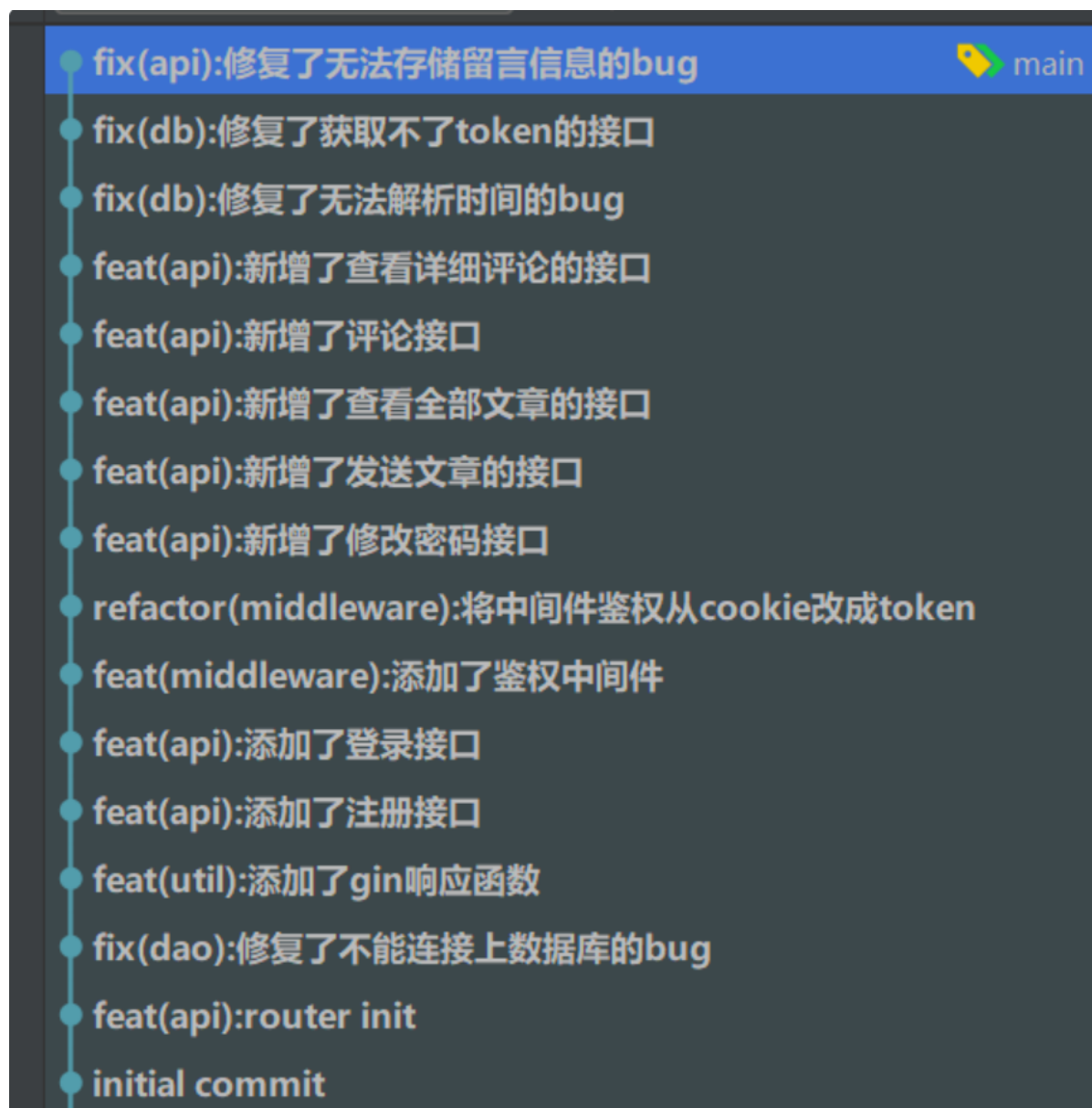
4. 然后就可以看到你的项目在github上出现了

## git commit规范

commit信息应该规范，指出你这次做了什么和本次提交的目的

良好的代码规范对于多人员协同开发都是有好处的，任何人都可以清晰的了解到你本次的提交做了什么

*example*



可能有的同学会把所有的作业一起丢进一个 RedRock-Homework 的仓库内，commit message可能会比较混乱，所以此次作业对commit不做规范要求，但是后续的寒假考核或者是大作业都最好有完整的commit message，如果可以的话每天写的都可以提交一次。

推荐一个挺好的插件gitmoji可以帮助你更好的书写git commit message 并且帮助你规范你的书写

## 常用的git命令

```
1  # 初始化仓库
2  git init                                # 初始化一个新的 Git 仓库
3
4  # 克隆c
5  git clone <repository-url>              # 克隆远程仓库到本地
6  git clone <repository-url> <directory>  # 克隆到指定目录
7  git clone -b <branch-name> <repository-url> # 克隆指定分支
8
9  # 查看状态与记录
10 git status                             # 查看工作区状态
11 git diff                                # 查看未暂存的改动
12 git log                                 # 查看提交日志
13 git log --oneline                       # 简洁模式查看提交日志
14 git log -p <file>                      # 查看某文件的修改历史
15 git blame <file>                       # 查看文件每行代码的提交记录
16
17 # 添加与提交代码
18 git add <file>                          # 添加指定文件到暂存区
19 git add .                               # 添加所有更改文件到暂存区
20 git commit -m "提交信息"                # 提交暂存区的改动到本地仓库
21 git commit -a -m "提交信息"            # 跳过暂存区直接提交
22
23 # 分支管理
24 git branch                              # 查看所有本地分支
25 git branch <branch-name>               # 创建分支
26 git checkout <branch-name>             # 切换到指定分支
27 git checkout -b <branch-name>          # 创建并切换到新分支
28 git branch -d <branch-name>            # 删除本地分支
29 git merge <branch-name>                 # 合并指定分支到当前分支
30
31 # 远程操作
32 git remote -v                           # 查看远程仓库地址
33 git remote add origin <repository-url> # 添加远程仓库
34 git remote remove origin                # 删除远程仓库
35 git push origin <branch-name>          # 推送代码到远程分支
36 git push -u origin <branch-name>       # 第一次推送并关联分支
37 git pull origin <branch-name>          # 拉取远程分支代码到本地
38
39 # 标签管理
40 git tag <tag-name>                      # 创建标签
41 git tag                                  # 查看所有标签
42 git push origin <tag-name>              # 推送标签到远程
43 git tag -d <tag-name>                  # 删除本地标签
44 git push origin --delete <tag-name>     # 删除远程标签
45
46 # 回滚与撤销
47 git reset --hard HEAD~1                 # 回退到上一个版本
48 git reset HEAD <file>                  # 撤销暂存区的改动
49 git checkout -- <file>                  # 丢弃工作区未暂存的改动
50 git push origin <branch-name> --force   # 强制回退远程分支（慎用）
51
52 # 暂存和清理
```

53	<code>git stash</code>	# 暂存当前更改（保存工作区状态）
54	<code>git stash pop</code>	# 恢复暂存的更改
55	<code>git clean -f</code>	# 清理未跟踪文件

## 书写项目

我们不可能拿到项目的时候，就闷着头一个劲的敲代码，项目的形成有一个完整的线的。最好不要直接上手去敲代码，要不然当你写着写着灵光乍现一下就把你写的完全推翻了怎么办，或者就是写着写着没有头绪了，卡在一个地方卡了很久都没有继续动笔。

所以在写代码之前最好先设计一下你的项目应该怎么写，也就是我们常说的需求，有了这个你才有一个完整的思路去写代码，才不会出现上面的情况。

**书写一个项目（后端）的步骤通常包括以下几个主要环节：**（目前来说）

### 1. 需求分析

- **功能需求：**了解和确认后端需要实现的具体功能。例如，API 接口的设计、用户认证、数据处理等。
- **非功能需求：**如性能、可扩展性、并发处理、安全性、容错性等。

### 2. 技术选型

- **编程语言和框架：**选择合适的后端框架（如Gin、Hertz 等）。
- **数据库选择：**选择合适的数据库（关系型如 MySQL、PostgreSQL，非关系型如 MongoDB、Redis）。

### 3. 架构设计

- **API 设计：**设计 RESTful API 接口，确定请求和响应格式。
- **数据库设计：**设计数据库 schema、表关系、索引等，考虑数据的存储、查询效率和一致性。
- **安全性设计：**设计身份验证（如 OAuth、JWT）（下节课会讲的内容——鉴权）和授权机制，保护用户数据和隐私。

### 4. 开发环境准备

- **代码版本管理：**使用 Git 等工具管理代码，搭建 Git 仓库（GitHub或私有仓库）。
- **开发环境搭建：**设置本地开发环境，包括安装必要的软件以及开发框架（go get 相关第三方库）。

### 5. 后端开发

- **业务逻辑开发：**根据需求实现核心功能，如用户注册、数据处理、文件上传等。
- **数据库操作：**编写数据库查询、更新操作，确保性能和数据一致性。
- **接口开发：**根据 API 设计开发后端接口，处理 HTTP 请求和响应，保证接口的性能和易用性。
- **错误处理与日志记录：**确保代码有完善的错误处理机制，并通过日志记录系统来监控异常和关键事件（要学会善用日志来帮你定位错误）。
- **安全性实现：**对敏感数据（如密码，不使用明文存入数据库）进行加密，确保传输和存储的安全性，防止 SQL 注入、XSS 等常见攻击。

### 6. 单元测试与集成测试

- **单元测试：**为每个模块编写单元测试，确保每个功能块的正确性。
- **集成测试：**测试系统中各模块的协作，确保整体功能的完整性。
- **API 测试：**对外部接口进行压力测试和功能测试，确保请求和响应的正确性。
- **代码优化：**优化代码逻辑和数据库查询，减少冗余操作，提高系统响应速度。
- **数据库优化：**优化数据库查询语句、增加索引、分表分库等。

### 7. 维护与迭代

- **Bug 修复：**根据日志监控，修复潜在的 bug 和问题。

- **功能更新与迭代**：根据产品需求和用户反馈，定期进行功能的更新和优化。

## 8. 文档编写

- **API 文档**：编写清晰的 API 文档，方便前端开发人员或其他团队调用接口（可以使用 Swagger 或 Postman 生成）。
- **开发文档**：撰写详细的技术文档，说明系统架构、设计决策、部署流程等，方便后期维护

## 项目实战

---

### 实战环节

---

这就到了这节课的末尾环节了同时也可以作为你们作业的开头环节

虽然大概率代码是敲不完的，但是这节课的主要目的是展示一下代码架构和一些规范等等。

接口仅随机挑了几个比较有代表性的来写，剩下的就需要你们课下去完成了

人人都不看好我  
偏偏我也不争气



## 作业

---

**lv0**：自学markdown语法，并上手自己写一些markdown文件

**lv1**：实现留言板基础功能：用户注册、登录、发表留言、获取所有留言、删除留言，并形成你的接口文档，最后在项目里添加 `README.md` 文件简单讲述一下你的项目实现的功能，也可以写下任何你想写的东西

有良好的代码规范，并且留言板基础功能实现的好的同时接口文档格式书写正确的三位同学（先到先得，但是不是越早就能得到，也得符合要求），我自费给你们买奶茶

```
1 CREATE TABLE `users` (  
2   `id` INT AUTO_INCREMENT PRIMARY KEY,      -- 用户的唯一标识  
3   `nickname` VARCHAR(255)                  -- 用户名  
4   `username` VARCHAR(255) NOT NULL UNIQUE,   -- 账号，确保唯一  
5   `password` VARCHAR(255) NOT NULL,         -- 用户密码  
6   `created_at` DATETIME DEFAULT CURRENT_TIMESTAMP, -- 用户创建时间  
7   `updated_at` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE  
   CURRENT_TIMESTAMP -- 用户更新时间  
8 );  
9  
10 CREATE TABLE `messages` (  
11   `id` INT AUTO_INCREMENT PRIMARY KEY,      -- 留言的唯一标识  
12   `user_id` INT NOT NULL,                  -- 留言的用户ID，外键引用 `users` 表的  
   `id`  
13   `content` TEXT NOT NULL,                 -- 留言内容  
14   `created_at` DATETIME DEFAULT CURRENT_TIMESTAMP, -- 留言时间  
15   `updated_at` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE  
   CURRENT_TIMESTAMP, -- 留言更新时间  
16   `is_deleted` TINYINT(1) DEFAULT 0,      -- 是否删除（逻辑删除），0表示未删除，1  
   表示已删除  
17   `parent_id` INT DEFAULT NULL,            -- 父留言ID，支持回复功能，根留言为NULL  
18   FOREIGN KEY (`user_id`) REFERENCES `users`(`id`) ON DELETE CASCADE, -- 外  
   键，删除用户时自动删除留言  
19   FOREIGN KEY (`parent_id`) REFERENCES `messages`(`id`) ON DELETE CASCADE --  
   外键，删除父留言时自动删除回复  
20 );
```

**lv2:** 新增用户信息更改（修改现有数据库表格），并实现楼中楼回复，并对接口文档进行新增

tips:完成level2可能需要数据结构 树 以及 对树的遍历 相关知识，请自行搜索相关资料并学习

**lv3:** 试着给留言新增点赞功能/留言板的主人可以自由删除所有留言板上的内容（**可选**，二选一即可），修改增删数据库表格实现功能，并完善你的接口文档

**lv4:** 接下来你可以自由发挥完善这个小的demo，自由地想象你的留言板可以变成什么样，然后实现他（**可选**）

## 作业提交方式：

本次作业必须将作业提交至**GitHub**，再将作业的链接发送邮箱至 [be@redrock.team](mailto:be@redrock.team)

作业提交格式 第七次作业-lv1-仙贝-114514

作业体量可能比较大，可以晚点提交捏

大家加油哦，不要放弃马上就要结束了

## 参考文献

1. [markdown教程](#)
2. [阮一峰 理解RESTful架构](#)
3. [git的操作](#)
4. [git常用命令](#)
5. [httpcat](#)

