

## 1 go 语言死锁的讨论

## 2 发送消息流程设计

### 2.1 流程图

### 2.2 引入了 redis 和不引入 redis 的对比

### 2.3 使用 tcp 的效率

### 2.4 使用 udp 的效率

### 2.5 使用 QUIC 的效率

## 3 将 http 协议升级为 websocket 协议

待补充

## 4 发送消息的设计和实现

需要：发送者 ID，接收者 ID，消息类型 (文字、音频、图片等)，消息的内容 (说了什么话、发了什么图片)，发送类型 ()

## 5 redis 的引入

为什么需要引入 redis：对于单机，使用 websocket 可以达到十万的并发。如果想要达到百万并发就需要引入 redis 了。

### 5.1 引入 redis 和不引入 redis 的对比

## 6 token 的引入

待实现 (登陆需要引入 token)

## 7 邮箱和手机号的校验

众所周知,邮箱应该是类似于 xxx@hotmail.com 的形式,手机号应该是 186xxxxxxx 这样的形式,因为在内存中我们采取的是用 string 来存储,因此在注册的时候,应该

进行校验。在这里我们采用的是正则校验表达式。同时看到下图还有几个问题：

1. UID 应该全为数字，类似于 QQ 号
2. 密码和确认密码不应该明文出现，保护隐私

Name	Description
<b>UID</b> * required string (query)	UID <input type="text" value="sc"/>
<b>name</b> * required string (query)	用户名 <input type="text" value="scy"/>
<b>password</b> * required string (query)	密码 <input type="text" value="123456"/>
<b>repassword</b> * required string (query)	确认密码 <input type="text" value="123456"/>
<b>email</b> * required string (query)	email <input type="text" value="scq"/>
<b>phone</b> * required string (query)	phone <input type="text" value="2018"/>

Execute

response

#### Details

Error: Bad Request

#### Response body

"邮箱或者手机号不合法，请检查输入"

正则表达式截图：

```
id:"matches(/~~~~~^1[3456789]\d{9}$/, /~~~~~^(13[0-9]|14[579]|15[0-3,5
```

```
id:"email"`
```

## 8 使用 md5 进行密码加密

为了解决什么问题：内部人员可能看到账号密码/手机号，需要保护用户的隐私，同时也为了防止数据库被入侵，导致密码泄露。

加密流程：随机数 + 用户的密码进行 md5 加密

为什么需要引入随机数：防止彩虹表攻击，通常黑客会有一个表，比如 123456 对应 e10adc3949ba59abbe56e057f20f883e。因此如果黑客能够看到数据库内的数据，可以对照彩虹表来得到原始密码。引入了随机数可以防止这种问题的发生，因为不知道随机数是多少。

Parameters

Name	Description
<b>UID</b> * required string (query)	UID <input type="text" value="99"/>
<b>name</b> * required string (query)	用户名 <input type="text" value="123"/>
<b>password</b> * required string (query)	密码 <input type="text" value="12"/>
<b>repassword</b> * required string (query)	确认密码 <input type="text" value="12"/>
<b>email</b> * required string (query)	email <input type="text" value="2018640800@qq.com"/>
<b>phone</b> * required string (query)	phone <input type="text" value="18612735755"/>

Execute

加密后在数据库中的样子，可以看出是正确的：

2	2024-03-12 21:16:11.640	2024-03-12 21:16:11.640	[NULL]	049	scy	123	123
3	2024-03-12 21:32:43.334	2024-03-12 21:32:43.334	[NULL]	sc	scy	123456	18612735755
4	2024-03-12 22:17:52.192	2024-03-12 22:17:52.192	[NULL]	99	123	02a315bedc95648608e2c1b548aacb4f	18612735755

## 9 实现分布式唯一全局 ID(雪花算法)

待更新

## 10 panic 异常处理和数据库无法连接的问题的解决

有的时候程序会出现一些不符合我们预期的 bug，这个时候我们需要捕获这个 bug 并且进行处理。如果不处理的话，线上服务可能会导致巨大的损失，且会导致后续难以排查问题。golang 使用的是 panic 来解决这个问题。当发生 panic 的时候，报错如下：

```
[error] failed to initialize database, got error Error 1049 (42000): Unknown database 'db'
abase 'db'
panic: 连接mysql 失败,err==Error 1049 (42000): Unknown database 'db'

goroutine 1 [running]:
awesomeProject/utils.InitMySQL()
    C:/Users/Administrator/GoLandProjects/awesomeProject/utils/systems_init.
go:39 +0x3ca
main.main()
    35      )
    36      Temp_db, err := gorm.Open(mysql.Open(viper.GetString(key: "mysql.dns")), &gorm.Config{
    37          Logger: logger})
    38      if err != nil {
    39          panic("连接mysql 失败,err==" + err.Error())
    40      }
```

经过排查发现是因为之前没有创建过这个数据库，使用 mysql 创建一个 db，再进行连接就可以了：

```
func main() {
    10      dsn := "root:123456@tcp(127.0.0.1:3306)/db?charset=utf8mb4&parseTime=
    11      db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    12      if err != nil {
    13          fmt.Printf("数据库连接失败: %v\n", err)
    14      } else {
    15          fmt.Println("数据库连接成功")
    16      }
    17  }
```

Run go build awesomeProject3

GOPATH=C:\Users\Administrator\go #gosetup

D:\Go\bin\go.exe build -o C:\Users\Administrator\AppData\Local\JetBrains\GoLand2023.3\tmp\GoLand\\_\_1go\_build\_awesomeP

awesomeProject3 #gosetup

C:\Users\Administrator\AppData\Local\JetBrains\GoLand2023.3\tmp\GoLand\\_\_1go\_build\_awesomeProject3.exe

数据库连接成功

Process finished with the exit code 0

## 11 golang 分层架构

问题引入：在创建用户的时候，报了循环引用的错误，具体的报错如下：

```
Debug go build awesomeProject
> <3 go setup calls>
package awesomeProject
    imports awesomeProject/router
    imports awesomeProject/service
    imports awesomeProject/models
    imports awesomeProject/utls
    imports awesomeProject/models: import cycle not allowed

Compilation finished with exit code 1
```

问题解决：想起来在字节实习时候经常看到组里的代码是分层的，便去系统学习了一下 golang 的分层架构。

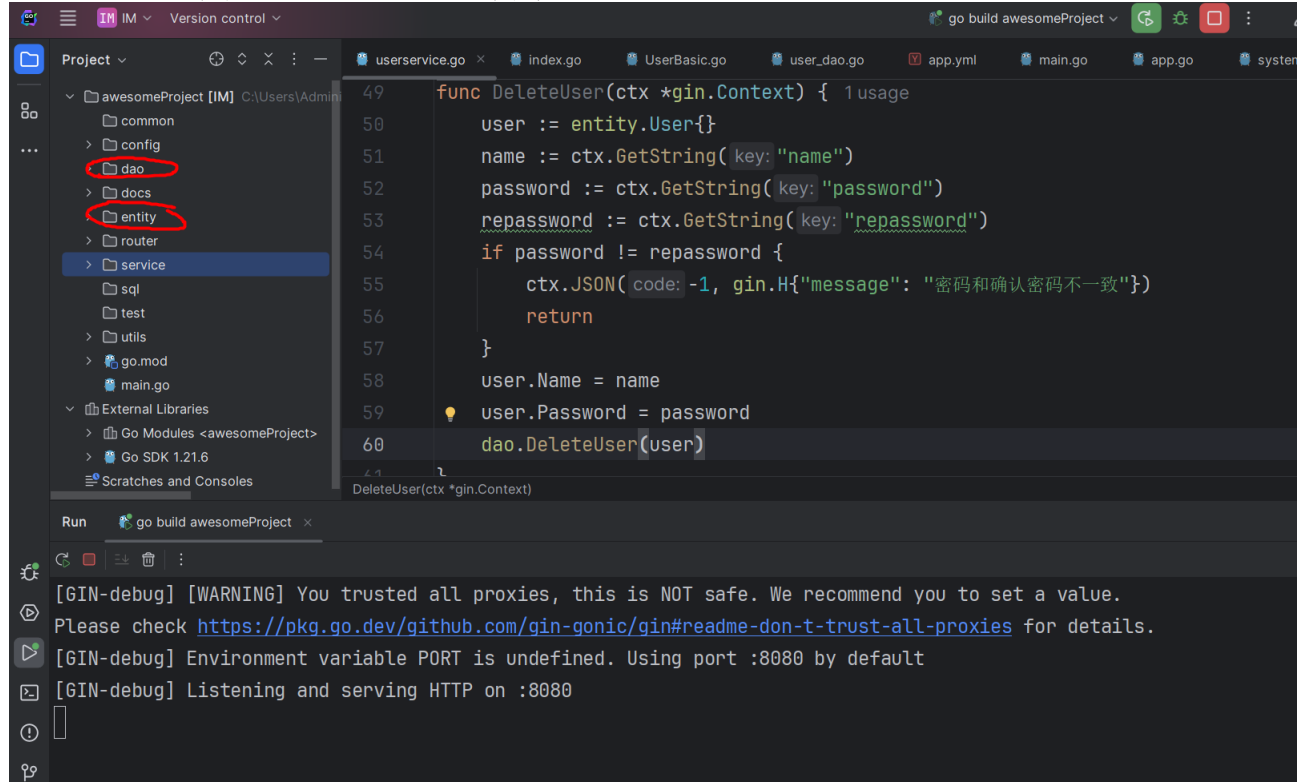
一个比较贴近实际开发流程的分层架构如下：



1. handlers: 处理器只做三件事情：接受请求解析入参、调用 services 完成业务逻辑、构造响应参数。handlers 不包含业务代码逻辑，应该简单地作路由使用。
2. services: 存放业务逻辑相关代码，是整个项目中逻辑最复杂的部分。
3. dao: 只进行对数据的 CRUD，不含有业务逻辑。

4. entity:entity 包存放领域实体及其相关方法及枚举。只能提供最基本的和实体相关的方法，如定义了 User 结构体，提供 IsValidUser 方法判断该 User 是否有效等。

在重新设计分层架构后，可以正常编译通过：



The screenshot shows an IDE interface with a project explorer on the left and a code editor on the right. The project explorer shows a directory structure for 'awesomeProject [IM]'. The 'entity' directory is highlighted with a red circle. The code editor shows the 'userservice.go' file with the following code:

```
49 func DeleteUser(ctx *gin.Context) { 1 usage
50     user := entity.User{}
51     name := ctx.GetString(key: "name")
52     password := ctx.GetString(key: "password")
53     repassword := ctx.GetString(key: "repassword")
54     if password != repassword {
55         ctx.JSON(code: -1, gin.H{"message": "密码和确认密码不一致"})
56         return
57     }
58     user.Name = name
59     user.Password = password
60     dao.DeleteUser(user)
```

The bottom of the IDE shows a terminal window with the following output:

```
Run go build awesomeProject x
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.
[GIN-debug] Environment variable PORT is undefined. Using port :8080 by default
[GIN-debug] Listening and serving HTTP on :8080
```

## 12 插入数据测试

localhost:8080/swagger/index.html#/用户模块/get\_user\_createUser

string (query)	隋春雨
password * required string (query)	密码 123456
repassword * required string (query)	确认密码 123456

Execute

### Responses

Curl

```
curl -X 'GET' \
'http://localhost:8080/user/CreateUser?name=%E9%9A%8B%E6%98%A5%E9%9B%A8&password=123456&repassword=123456' \
-H 'accept: application/json'
```

Request URL

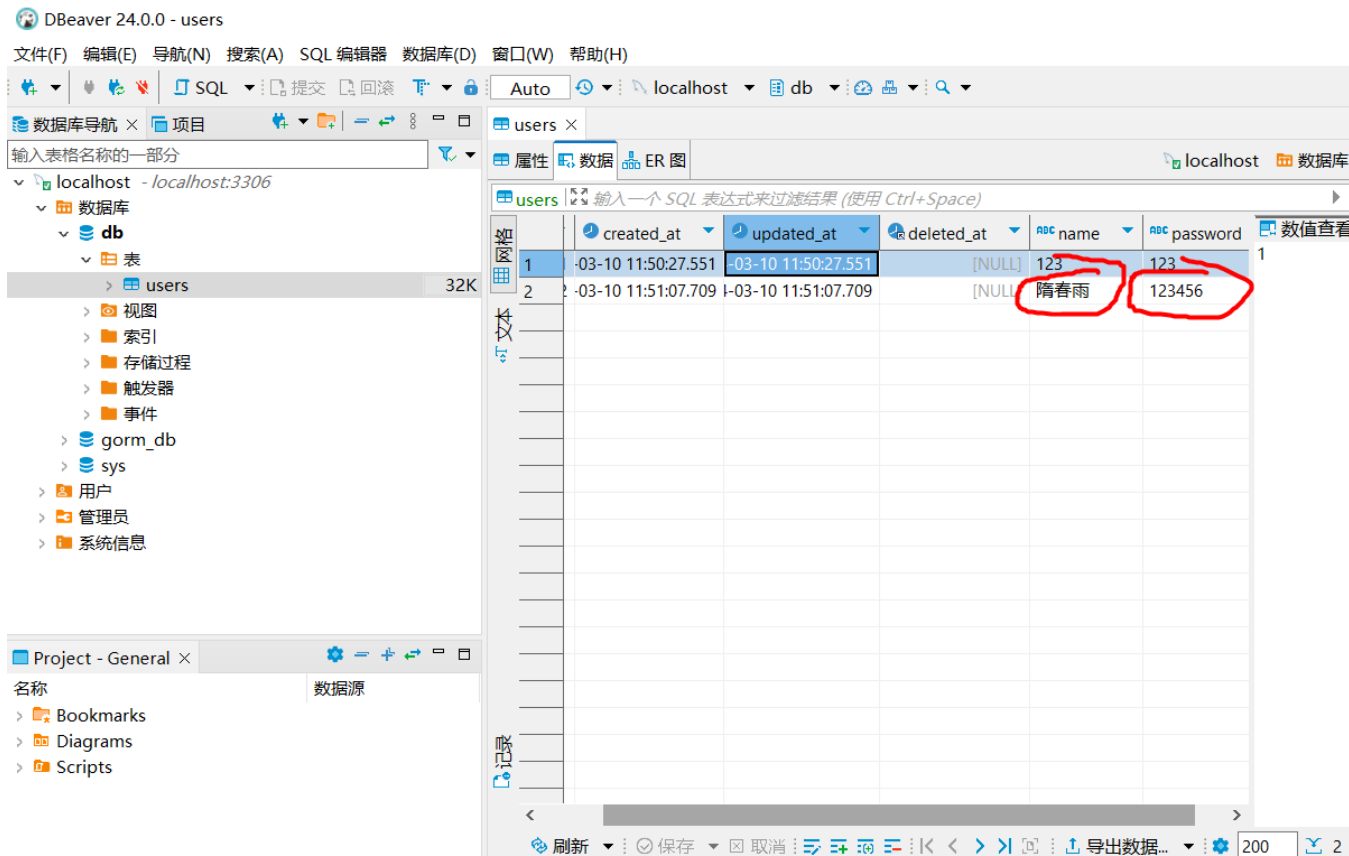
```
http://localhost:8080/user/CreateUser?name=%E9%9A%8B%E6%98%A5%E9%9B%A8&password=123456&repassword=123456
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "message": "新增用户成功" }</pre>

可以看到是成功的插入到了数据库里边：





## 13 golang 比较结构体的各个值是否相等

golang 并不能够直接比较两个结构体是否相等 (编译器没有实现), 可以使用 `reflect.DeepEqual()` 函数来比较, 如下图



## 14 swagger

### 14.1 问题背景

问题背景：之前也用过其他的 API 文档工具，但是最大的问题还是文档和代码是分离的。总是出现文档和代码不同步的情况。

### 14.2 为了解决什么问题

为了解决什么问题：自动化帮写接口说明文档。目前的项目基本都是前后端分离的项目，有时候后端更改完代码，有时候忘记更新了接口的说明文档

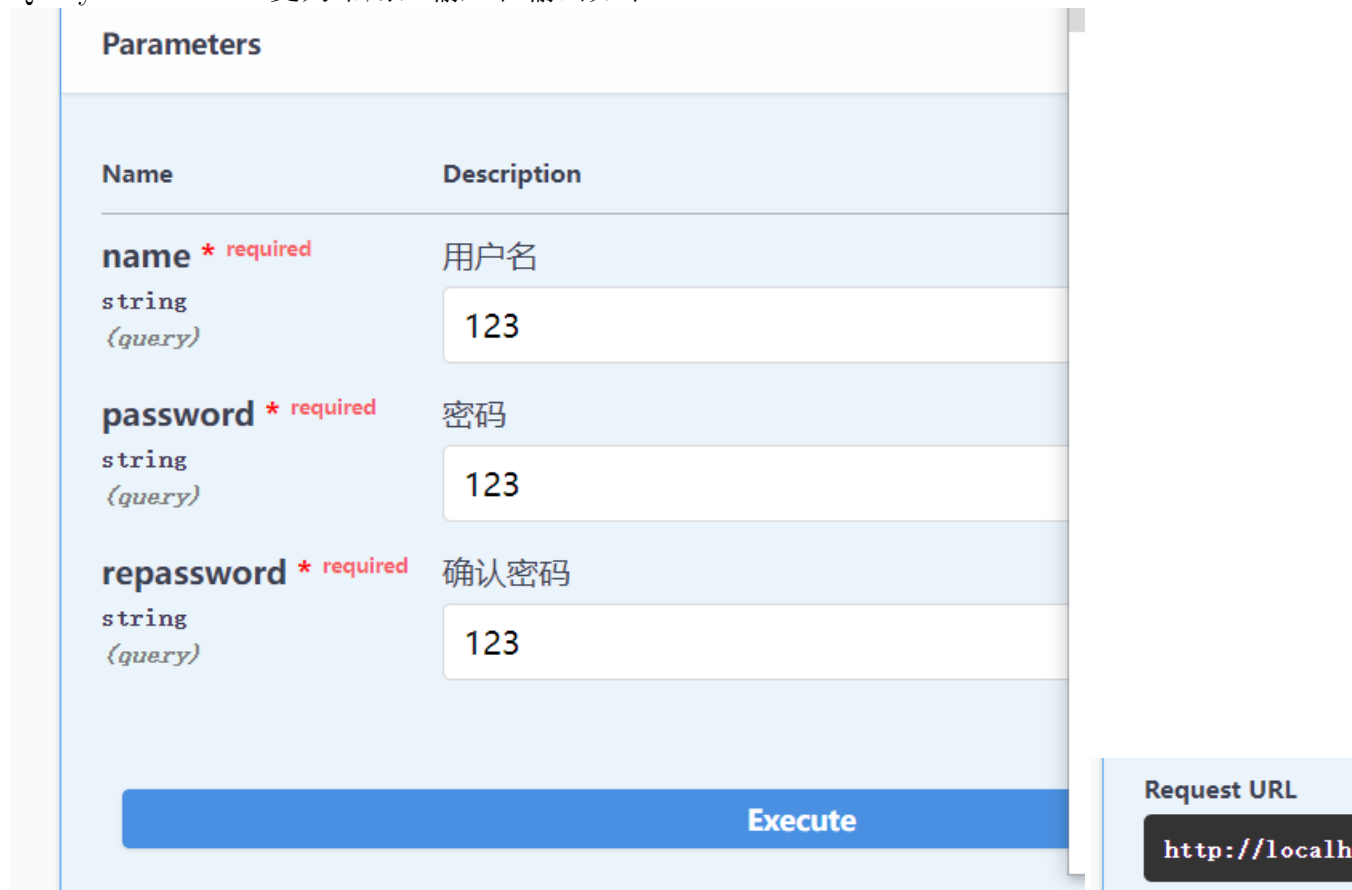
### 14.3 不同的 Parameter Types 的区别

Swagger 的 Parameter Types 可以分为以下几种：

Path Parameters（路径参数）：出现在 URL 路径中，通常用于标识资源。

Query Parameters（查询参数）：出现在 URL 查询字符串中，用于过滤或定位资源。

Query Parameters 更为常用，输入和输出如下：



The image shows a Swagger UI interface for the 'Parameters' section. It lists three query parameters, each with a description and an input field. The parameters are 'name', 'password', and 'repassword', all of which are required and of type 'string'. The input fields contain the value '123'. At the bottom, there is a blue 'Execute' button. To the right, a 'Request URL' field shows 'http://localh'.

Name	Description
<b>name</b> * required string (query)	用户名 123
<b>password</b> * required string (query)	密码 123
<b>repassword</b> * required string (query)	确认密码 123

Execute

Request URL  
http://localh

Header Parameters（头部参数）：出现在请求头部中，用于传递附加信息。

Cookie Parameters (Cookie 参数): 出现在 Cookie 头部中, 用于在客户端和服务端之间传递状态信息。

## 15 HTTP 状态码和 swagger

- 200: OK, 请求成功, 一般用于 GET 与 POST 请求。
- 301: Moved Permanently, 请求的资源已经被永久的移动到新的 URL, 返回信息包括一个新的 URL。
- 304: Not modified, 未修改。查看本地缓存。
- 305: 使用代理。所请求的资源必须通过代理访问。
- 400: 客户端请求语法错误。
- 404: not found。
- 500: 服务器内部错误。

## 16 更新用户模块测试

输入数据:

Name	Description
<b>id</b> * required integer (query)	id <input type="text" value="2"/>
<b>name</b> * required string (query)	name <input type="text" value="隋春雨"/>
<b>password</b> * required string (query)	password <input type="text" value="123456"/>
<b>new_name</b> * required string (query)	new name <input type="text" value="scy"/>
<b>new_password</b> * required string (query)	new password <input type="text" value="qaq"/>

Execute

Responses

Curl

Server response

Code	Details
200	<div>Response body</div> <div>"更新用户信息成功"</div> <div>Response headers</div>

db 之前的数据:

users 输入一个 SQL 表达式来过滤结果 (使用 Ctrl+Space)

	id	created_at	updated_at	deleted_at	name	password
1	1	-03-10 11:50:27.551	-03-10 11:50:27.551	-03-10 16:21:53.998	123	123
2	2	-03-10 11:51:07.709	-03-10 11:51:07.709	[NULL]	隋春雨	123456
3	3	-03-10 14:37:10.954	-03-10 14:37:10.954	[NULL]	隋春雨	123456
4	4	-03-10 15:43:07.561	-03-10 15:43:07.561	-03-10 16:22:28.655	123	123
5	5	-03-10 16:31:03.649	-03-10 16:31:03.649	-03-10 16:32:58.063	123	123

更新之后的数据:

users 输入一个 SQL 表达式来过滤结果 (使用 Ctrl+Space)

	id	created_at	updated_at	deleted_at	name	password
1	1	-03-10 11:50:27.551	-03-10 11:50:27.551	-03-10 16:21:53.998	123	123
2	2	-03-10 11:51:07.709	-03-10 16:57:49.733	[NULL]	scv	qag
3	3	-03-10 14:37:10.954	-03-10 14:37:10.954	[NULL]	隋春雨	123456
4	4	-03-10 15:43:07.561	-03-10 15:43:07.561	-03-10 16:22:28.655	123	123
5	5	-03-10 16:31:03.649	-03-10 16:31:03.649	-03-10 16:32:58.063	123	123

可以看出来，更新成功