

# Redis集群

## 学习

是的，Docker Hub上有一个现成的Redis集群镜像：redis-cluster。使用该镜像可以通过一个命令快速搭建Redis集群。

下面是一个示例命令，可以快速启动一个6节点的Redis集群：

```
docker run -d --name redis-cluster -e REDIS_CLUSTER_NODES=6 -p 7000-7005:7000-7005 redis-cluster
```

运行这个命令之后，Docker就会自动下载redis-cluster镜像并启动6个Redis节点，这些节点会自动组成一个Redis集群。

可以使用Redis客户端与Redis集群进行交互，以实现数据存储和读取。

(2023.2.18)

## 视频去重

## 学习

如果你是在开发一个抖音类的应用程序，可以在代码层面采取以下方法来避免播放重复视频：

1. 实现视频去重逻辑：当用户浏览视频时，应用程序可以记录用户已经播放的视频ID，下次浏览时，可以过滤掉已经播放过的视频。这可以使用缓存技术来实现，将已经播放过的视频ID存储在本地缓存中，每次播放新视频时，先检查该视频ID是否存在于缓存中，如果已经存在，则跳过该视频。
2. 使用推荐算法：通过使用推荐算法，应用程序可以向用户推荐与他们喜欢的视频相关的新视频，从而增加他们观看新视频的机会。推荐算法可以基于用户喜好、观看历史、标签等多种因素进行推荐，从而提高用户体验。
3. 增加视频来源：应用程序可以集成多个视频来源，如不同的用户上传的视频、官方发布的视频、热门视频等，这可以让用户有更多的视频选择，减少看到重复视频的机会。
4. 增加视频过滤机制：应用程序可以设置视频过滤机制，例如，过滤掉过于相似的视频、过短或过长的视频、低质量的视频等，从而提高用户观看体验。
5. 优化应用程序性能：优化应用程序性能可以使视频播放更加流畅，降低视频加载时间，从而减少用户看到重复视频的机会。可以采用异步加载、图片压缩、网络优化等技术来优化应用程序性能。

(2023.2.12)

## 缓存数据一致性

<https://juejin.cn/post/7152670651302543397#heading-4>

在有数据变更的地方，同时更新缓存和数据库。

对于读多写少的情况，采用事务（高隔离级别）+先更新数据库再更新缓存

对于并发要求高的且一致性要求高，选择“先更新数据库再删除缓存，并结合删除重试 + 补偿逻辑 + 缓存过期TTL等综合手段”

## 学习

对于读多写少的情况的代码实现：

```

func updateDBAndCache(db *sql.DB, cache *redis.Client, data []byte) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }
    defer tx.Rollback()

    // 更新数据库
    stmt, err := tx.Prepare("UPDATE table SET data = ? WHERE id = ?")
    if err != nil {
        return err
    }
    defer stmt.Close()

    if _, err := stmt.Exec(data, 1); err != nil {
        return err
    }

    if err := tx.Commit(); err != nil {
        return err
    }

    // 更新缓存
    err = cache.Set("cache_key", data, 0).Err()
    if err != nil {
        return err
    }

    return nil
}

```

对于并发要求高的且一致性要求高的代码实现：

```

func updateDBAndDeleteCache(db *sql.DB, cache *redis.Client, data []byte) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }
    defer tx.Rollback()

    // 更新数据库
    stmt, err := tx.Prepare("UPDATE table SET data = ? WHERE id = ?")
    if err != nil {
        return err
    }
    defer stmt.Close()

    if _, err := stmt.Exec(data, 1); err != nil {
        return err
    }

    if err := tx.Commit(); err != nil {
        return err
    }
}

```

```

// 删除缓存
err = cache.Del("cache_key").Err()
if err != nil {
    return err
}

// 重试 + 补偿逻辑
for i := 0; i < 3; i++ {
    _, err = cache.Get("cache_key").Result()
    if err == redis.Nil {
        break
    }

    // 进行补偿逻辑，比如重新更新数据库等
    // ...
}

return nil
}

```

## 数据库全部配置

### docker安装Mysql8

```

docker pull mysql:8.0.26
docker run -itd --name mysql8 -p 3377:3306 -e MYSQL_ROOT_PASSWORD=12qwAS mysql:8.0.26

```

```

root:12qwAS@(43.139.72.246:3377)/db_douyin

```

```

create database db_douyin

```

### 评论

```

CREATE TABLE tb_comments (
  comment_id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '评论id',
  user_id BIGINT UNSIGNED NOT NULL COMMENT '评论者用户id',
  video_id BIGINT UNSIGNED NOT NULL COMMENT '被评论视频id',
  content TEXT NOT NULL COMMENT '评论内容',
  create_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '评论时间',
  PRIMARY KEY (comment_id),
  FOREIGN KEY (user_id) REFERENCES tb_users(user_id),
  FOREIGN KEY (video_id) REFERENCES tb_videos(post_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储评论信息';

```

## docker部署go项目

[https://blog.csdn.net/qq\\_39787367/article/details/126014285](https://blog.csdn.net/qq_39787367/article/details/126014285)

Dockerfile

```
# 基础镜像
FROM alpine:3.12
# 维护者
MAINTAINER frank
# docker build 时执行命令 - 创建目录
RUN mkdir -p "/data/app" \
&& ln -sf /dev/stdout /data/app/service.log
# 工作目录
WORKDIR "/data/app"
# 拷贝
COPY hello /data/app/hello
# docker run 时执行命令
ENTRYPOINT [ "./hello" ]
```

```
docker build -t hello:v1.0.0 .
docker images
docker run -d -p 9090:8080 hello:v1.0.0
```

## 开发安全

<https://www.cnblogs.com/teufel/p/10760635.html>

(2023.2.17)

## 聊天功能（消息队列）

### 数据库

```
CREATE TABLE tb_chat (
  chat_id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '聊天id',
  sender_id BIGINT UNSIGNED NOT NULL COMMENT '发送者用户id',
  recipient_id BIGINT UNSIGNED NOT NULL COMMENT '接收者用户id',
  message TEXT NOT NULL COMMENT '消息内容',
  send_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '发送时间',
  PRIMARY KEY (chat_id),
  FOREIGN KEY (sender_id) REFERENCES tb_users(user_id),
  FOREIGN KEY (recipient_id) REFERENCES tb_users(user_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储私人聊天记录';
```

### 学习

对于后端聊天接口，还有很多其他更好的实现方法，这取决于你的需求和项目规模。

1. 使用 WebSockets 或 Server-Sent Events (SSE) 来实现即时通讯。
2. 使用第三方通讯工具，如 Pusher, Ably 等。
3. 使用 XMPP (Extensible Messaging and Presence Protocol) 协议，这是一种开放的、可扩展的即时通讯协议。
4. 使用 MQTT (Message Queuing Telemetry Transport) 协议，这是一种专门针对 IoT (Internet of Things) 设备的即时通讯协议。

总的来说，应该根据项目的特定需求和规模选择最合适的解决方案。

## 代码

当使用消息队列来实现私信聊天功能时，可以将每个私信聊天对话视为一个主题，将每条私信消息作为一个消息发送到该主题中。

对于每个私信聊天对话主题，可以将其以某种方式命名，例如使用发送者和接收者的 ID 拼接在一起作为主题名称，这样可以确保每个对话主题的唯一性。

在发送私信消息时，需要指定目标主题，即发送到哪个私信聊天对话主题中。这样可以确保私信消息只被发送到对应的聊天对话中。

在消息队列的消费者代码中，需要订阅所有的私信聊天对话主题，并在每个主题中获取消息。获取的消息中包含了发送者 ID、接收者 ID、消息内容和发送时间等信息，可以将这些信息保存到数据库中，以便在后续查询聊天记录时使用。

总的来说，将每个私信聊天对话视为一个主题，可以简化系统的设计和实现，同时也可以确保私信消息只被发送到对应的聊天对话中。同时，在消费者代码中需要订阅所有的对话主题，并及时处理获取到的私信消息，以便保证消息的实时性和准确性。

## 控制层

- `SendMessage`：发送消息，将消息写入 Kafka 和 MySQL。
- `ReceiveMessage`：接收消息，从 Kafka 获取消息，并将消息转换为聊天消息对象。
- `GetMessagesByUser`：获取指定用户的聊天记录，从 MySQL 获取聊天消息对象数组。

```
// UserService 用于处理用户相关操作
type UserService struct {
    kafkaClients map[string]*KafkaClient
    // 在数据库中维护用户和 topic 的映射关系
    userTopicMap map[string]string
}

// NewUserService 创建 UserService 对象
func NewUserService() *UserService {
    return &UserService{
        kafkaClients: make(map[string]*KafkaClient),
        userTopicMap: make(map[string]string),
    }
}

// SendMessage 发送消息
func (u *UserService) SendMessage(sender, receiver, message string) error {
    // 获取目标用户的 topic
    topic, ok := u.userTopicMap[receiver]
    if !ok {
        // 如果目标用户不存在，则动态创建对应的消息队列
        topic = fmt.Sprintf("private-chat-%s", receiver)
        // 创建新的 KafkaClient 对象
        kafkaClient := NewKafkaClient([]string{"kafka:9092"}, topic)
        u.kafkaClients[receiver] = kafkaClient
        u.userTopicMap[receiver] = topic
    }
    // 发送消息到目标用户的 topic
    err := u.kafkaClients[receiver].ProduceMessage(message)
```

```

    if err != nil {
        return err
    }
    return nil
}

// GetMessage 获取聊天记录
func (u *UserService) GetMessage(user string) ([]string, error) {
    // 获取用户的消息队列
    kafkaClient, ok := u.kafkaClients[user]
    if !ok {
        return nil, fmt.Errorf("no message queue for user %s", user)
    }
    // 从消息队列中获取历史消息
    messages := []string{}
    for {
        msg, err := kafkaClient.ConsumeMessage()
        if err != nil {
            if err.Error() == "context deadline exceeded" {
                // 如果超时，说明已经没有更多消息了
                break
            }
        }
        return nil, err
    }
    messages = append(messages, msg)
}
return messages, nil
}

// ChatController 用于处理聊天相关操作
type ChatController struct {
    userService *UserService
}

// NewChatController 创建 ChatController 对象
func NewChatController(userService *UserService) *ChatController {
    return &ChatController{

```

## 知识点

- 1、Kafka 本质上是一个消息队列，一个高吞吐量、持久性、分布式的消息系统。
- 2、包含生产者（producer）和消费者（consumer），每个consumer属于一个特定的消费者组（Consumer Group）。
- 3、生产者生产消息（message）写入到kafka服务器（broker，kafka集群的节点），消费者从kafka服务器（broker）读取消息。
- 4、消息可分为不同的类型即不同的主题（topic）。
- 5、同一主题（topic）的消息可以分散存储到不同的服务器节点（partition）上，一个分区（partition）只能由一个消费者组内的一个消费者消费。
- 6、每个partition可以有多个副本，一个Leader和若干个Follower，Leader发生故障时，会选取某个Follower成为新的Leader。

## 教程

<https://www.cnblogs.com/YLTFY1998/p/16550406.html>

Kafka可视化工具: [https://blog.csdn.net/qg\\_45956730/article/details/127089597](https://blog.csdn.net/qg_45956730/article/details/127089597)

## 作废版本 (留底)

```
package chat

import (
    "fmt"
    "go_douyin/model"    "go_douyin/utils/kafka_client"    "strconv")

type ChatService struct {
    //commentMapper *dao.ChatMapper
    kafkaClients *kafka_client.KafkaClient
}

func NewChatService() *ChatService {
    return &ChatService{
        kafkaClients: &kafka_client.KafkaClient{},
    }
}

// AddChat 发送消息
func (h *ChatService) AddChat(comment model.Chat) error {
    tmp := strconv.FormatUint(comment.SenderID, 10) + "" + strconv.FormatUint(comment.RecipientID, 10)
    topic := fmt.Sprintf("chat%s", tmp)
    // 写成 private-chat-1:2 , 会报错
    fmt.Println(topic)
    kafkaClient := kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, topic)
    // 发送消息到目标用户的 topic    err := kafkaClient.ProduceMessage(comment.Message)
    if err != nil {
        return err
    }
    return nil
}

// GetMessages 获取聊天记录
func (h *ChatService) GetMessages(senderID string, recipientID string) ([]string, error) {
    tmp := senderID + "" + recipientID
    topic := fmt.Sprintf("chat%s", tmp)
    fmt.Println(topic)
    // 获取用户的消息队列
    kafkaClient := kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, topic)
    kafkaClient.GetLog()
    // 从消息队列中获取历史消息
    messages := []string{}
    for {
        msg, err := kafkaClient.ConsumeMessage()
        fmt.Println(msg)
        if err != nil {
            break
            return messages, nil
        }
        messages = append(messages, msg)
    }
    return messages, nil
}
```

```
return messages, nil
}
```

## 最终代码

与评论的流程一致，具体可以看评论那里

发送私信的话——就是用消息队列异步生产者把对应的消息发到指定主题中

main函数会协程（多线程）监听这个主题

如果监听到了会进行存储操作——存储到数据库（可以加个更新到缓存，过期时间可以设置短点）

获取聊天记录——直接数据库或者缓存读取

（关键代码）

main.go

```
// 监听私信聊天的消息队列
go dao.ListenChat()
```

service/chat/chatService.go

```
package chat

import (
    "encoding/json"
    "fmt"
    "go_douyin/global/variable"
    "go_douyin/model"
    "go_douyin/utils/kafka_client"
)

type ChatService struct {
    //chatMapper *dao.ChatMapper
    kafkaClients *kafka_client.KafkaClient
}

func NewChatService() *ChatService {
    return &ChatService{
        kafkaClients: &kafka_client.KafkaClient{},
    }
}

// AddChat 发送消息
func (h *ChatService) AddChat(chat model.Chat) error {
    // 序列化评论数据
    chatJSON, err := json.Marshal(chat)
    if err != nil {
        return err
    }
    // 将私信放到消息队列
    err = variable.Kafka_chat.ProduceMessage(string(chatJSON))
    return err
}

// GetMessages 获取聊天记录
func (h *ChatService) GetMessages(senderID string, recipientID string) ([]string, error) {
    fmt.Println("此处直接获取数据库或者缓存的数据即可")
}
```



```
        return nil, nil
    }
}
```

global/variable/variable.go

```
// 私信聊天的队列
Kafka_chat *kafka_client.KafkaClient

Kafka_chat = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "chat-topic")
```

chatMapper.go

```
package dao

import (
    "encoding/json"
    "fmt"
    "go_douyin/global/variable"
    "go_douyin/model"
)

type ChatMapper struct{}

func NewChatMapper() *ChatMapper {
    return &ChatMapper{}
}

// 监听私信消息队列
func ListenChat() {
    for {
        // 获取消息
        message, err := variable.Kafka_chat.ConsumeMessage()
        if err != nil {
            fmt.Printf("获取消息失败: %v\n", err)
            continue
        }
        // 反序列化私信数据
        var chat model.Chat
        err = json.Unmarshal([]byte(message), &chat)
        if err != nil {
            fmt.Printf("反序列化私信数据失败: %v\n", err)
            continue
        }
        // 存储私信数据
        err = SaveChat(chat)
        if err != nil {
            fmt.Printf("存储私信数据失败: %v\n", err)
            continue
        }
        fmt.Printf("成功存储私信数据: %v\n", chat)
    }
}

// 监听预加载私信消息队列
func ListenPreloadChatList() {
    for {
```

```

        fmt.Printf("正在预加载.....")
        // 获取消息
        message, err := variable.Kafka_preload.ConsumeMessage()
        if err != nil {
            fmt.Printf("获取消息失败: %v\n", err)
            continue
        }
        // 此次应写存储到缓存的函数
        fmt.Printf("正在缓存" + message + "视频id的私信")
    }
}

func SaveChat(chat model.Chat) error {
    // 这里是将私信数据存储在数据库的代码，具体实现方式取决于你使用的数据库类型
    // 这里还可以补个缓存
    fmt.Println("正在存储数据库，同时更新进缓存", chat)
    return nil
}

```

## chatController.go

```

package controller

import (
    "fmt"
    "github.com/gin-gonic/gin"    "go_douyin/model"    "go_douyin/service/chat"
    "go_douyin/utils/response"    "time")

type ChatController struct {
    chatService *chat.ChatService
}

func NewChatController() *ChatController {
    return &ChatController{
        chatService: chat.NewChatService(),
    }
}

// 发送信息
func (h *ChatController) AddChat(c *gin.Context) {
    // 获取请求参数
    var requestBody map[string]interface{}
    requestBody = make(map[string]interface{})
    // 解析请求体
    c.ShouldBindJSON(&requestBody)
    // 获取请求参数
    //在 HTTP POST 请求中，请求体中的数据通常是以字符串形式发送的。JSON 格式中的数字默认都是浮点型，默认都是 float64 类型
    user_id := requestBody["user_id"].(float64)
    to_user_id := requestBody["to_user_id"].(float64)
    content, _ := requestBody["content"].(string)
    var cc model.Chat
    cc.RecipientID = uint64(to_user_id)
    cc.Message = content
    cc.SenderID = uint64(user_id)
    cc.SendTime = time.Now()
}

```

```

    err := h.chatService.AddChat(cc)
    if err != nil {
        response.Success(c, "私信失败", gin.H{})
        fmt.Println(err)
        return
    }
    response.Success(c, "私信成功", gin.H{})
}

// 聊天记录
func (h *ChatController) ListChat(c *gin.Context) {
    // 获取请求参数
    ToUserId := c.Query("to_user_id")
    userId := c.Query("user_id")
    data, err := h.chatService.GetMessages(userId, ToUserId)
    if err != nil {
        response.Success(c, "获取失败", gin.H{})
        return
    }
    response.Success(c, "获取成功", gin.H{
        "message_list": data,
    })
}

```

router.go

```

chatController := controller.NewChatController()

// 社交组：私信聊天
v5 := router.Group("/douyin/message")
{
    v5.POST("action", chatController.AddChat)
    v5.GET("chat", chatController.ListChat)
}

```

chat.go

```

package model

import "time"

// 这里后面的记得改和以前的一样
type Chat struct {
    ChatID      uint64    `gorm:"column:chat_id;primaryKey;autoIncrement"`
    SenderID    uint64    `gorm:"column:sender_id;not null"`
    RecipientID uint64    `gorm:"column:recipient_id;not null"`
    Message     string    `gorm:"column:message;not null"`
    SendTime    time.Time `gorm:"column:send_time;not null;default:CURRENT_TIMESTAMP"`
}

func (Chat) TableName() string {
    return "tb_chat"
}

```

(2023.2.7)

# 雪花算法

## 学习

[https://blog.csdn.net/qg\\_42170897/article/details/127768047](https://blog.csdn.net/qg_42170897/article/details/127768047)

```
package snowflake

import (
    "errors"
    "sync"
    "time"
)

const (
    workerIDBits = 10
    sequenceBits = 12
    workerIDShift = sequenceBits
    timestampLeftShift = sequenceBits + workerIDBits
    sequenceMask = 1<<sequenceBits - 1
)

type IDGenerator struct {
    timestamp int64
    workerID   int64
    sequence   int64
    mu         sync.Mutex
}

func NewIDGenerator(workerID int64) (*IDGenerator, error) {
    if workerID < 0 || workerID > 1<<workerIDBits-1 {
        return nil, errors.New("worker ID excess the limit")
    }

    return &IDGenerator{
        workerID: workerID,
    }, nil
}

func (g *IDGenerator) Next() (int64, error) {
    g.mu.Lock()
    defer g.mu.Unlock()

    now := time.Now().UnixNano() / int64(time.Millisecond)
    if g.timestamp == now {
        g.sequence = (g.sequence + 1) & sequenceMask
        if g.sequence == 0 {
            for now <= g.timestamp {
                now = time.Now().UnixNano() / int64(time.Millisecond)
            }
        }
    } else {
        g.sequence = 0
    }
    g.timestamp = now
}
```

```

        return ((g.timestamp - int64(time.Date(2022, 1, 1, 0, 0, 0, 0,
time.UTC).UnixNano())/int64(time.Millisecond)))<<timestampLeftShift) |
            (g.workerID << workerIDShift) |
            g.sequence, nil
    }

```

## 原理

雪花算法（Snowflake Algorithm）是一种分布式ID生成算法，用于生成全局唯一的ID，它是由Twitter公司的工程师开发的。

它生成的ID是一个64位的整数，其中包含以下部分：

- 第1个41位为毫秒级时间戳，代表当前ID生成的时间；
- 第2个10位为工作节点ID，用于区分不同的服务节点；
- 第3个12位为毫秒内的计数序列，用于在同一时刻内生成多个ID。

使用这样的ID可以确保生成的ID全局唯一，并且时间戳可以直接使用，避免了数据库索引等其他复杂的问题。

总的来说，雪花算法提供了一种可靠且简单的方法来生成全局唯一的ID，在分布式系统中广泛使用。

## 代码

utils/snow\_flake.go

```

package snow_flake

import (
    "errors"
    "sync"
    "time"
)

const (
    workerIDBits = uint(10)
    sequenceBits = uint(12)

    maxWorkerID    = -1 ^ (-1 << workerIDBits)
    maxSequence    = -1 ^ (-1 << sequenceBits)
    workerIDShift  = sequenceBits
    timestampShift = sequenceBits + workerIDBits
)

type ID int64

type Snowflake struct {
    mu      sync.Mutex
    timestamp int64
    workerID int64
    sequence int64
}

func NewSnowflake(workerID int64) (*Snowflake, error) {
    if workerID < 0 || workerID > maxWorkerID {
        return nil, errors.New("worker ID excess of quantity")
    }
}

```

```

    return &Snowflake{workerID: workerID}, nil
}

func (s *Snowflake) Generate() ID {
    s.mu.Lock()
    defer s.mu.Unlock()

    now := time.Now().UnixNano() / int64(time.Millisecond)

    if s.timestamp == now {
        s.sequence = (s.sequence + 1) & maxSequence

        if s.sequence == 0 {
            for now <= s.timestamp {
                now = time.Now().UnixNano() / int64(time.Millisecond)
            }
        }
    } else {
        s.sequence = 0
    }

    s.timestamp = now

    id := ID((now-1420041600000)<<timestampShift |
        (s.workerID << workerIDShift) |
        (s.sequence))

    return id
}

```

test/snowflake\_test.go

```

package test

import (
    "fmt"
    "go_douyin/utils/snowflake"    "testing")

//测试雪花算法
func TestSnow(t *testing.T) {
    snowflake, err := snowflake.NewSnowflake(1)
    if err != nil {
        fmt.Println(err)
        return
    }

    id := snowflake.Generate()
    fmt.Println(id)
}

```

(2023.2.12)

## 评论功能

## 数据库

```
CREATE TABLE tb_comments (  
  comment_id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '评论id',  
  user_id BIGINT UNSIGNED NOT NULL COMMENT '评论者用户id',  
  video_id BIGINT UNSIGNED NOT NULL COMMENT '被评论视频id',  
  content TEXT NOT NULL COMMENT '评论内容',  
  create_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '评论时间',  
  PRIMARY KEY (comment_id),  
  FOREIGN KEY (user_id) REFERENCES tb_users(user_id),  
  FOREIGN KEY (video_id) REFERENCES tb_videos(post_id)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储评论信息';
```

## 学习

在实现评论功能时，后端可以使用以下技术亮点来提高评论功能的性能和安全性：

1. 分布式存储：使用分布式存储系统（例如，MongoDB，Cassandra等），可以提高评论功能的扩展性和可用性。
2. 消息队列：使用消息队列（例如，RabbitMQ，Kafka等），可以缓解评论功能的请求压力，提高系统的可用性。
3. 敏感词过滤：使用字典树，AC自动机等敏感词过滤技术，可以防止评论中的不当语言对其他用户造成影响。
4. 内容审核：使用人工智能，机器学习等技术，对评论内容进行审核，防止评论中的恶意信息。
5. 缓存：使用缓存（例如，Redis），可以提高评论功能的读取性能，缓解数据库的压力。

消息队列的业务：

1. 异步处理：通过将评论消息放入队列中，系统可以异步处理评论，并避免因请求处理时间过长而导致用户等待。
2. 数据处理流程分离：可以将数据处理流程分为多个独立的组件，每个组件负责处理特定的任务，并将处理结果放入下一个组件进行处理。
3. 可扩展性：当系统需要扩展时，可以通过添加新的工作者来处理评论消息，从而提高系统的并发能力。
4. 容错性：如果某个组件出现故障，其他组件仍然可以继续处理评论消息，从而保证系统的可用性。
5. 过滤不合法评论：通过使用消息队列，可以在接收评论时对评论进行过滤，以确保只有合法评论能够被处理。
6. 审核评论：通过使用消息队列，可以对评论进行审核，以确保只有符合要求的评论能够被公开显示。

在选择使用Kafka或RabbitMQ作为消息队列时，需要考虑多种因素。

Kafka是一个高吞吐量、分布式的消息队列系统，适用于大规模数据处理和流式数据处理场景。它支持高效的数据存储和处理，以及可靠的消息分发。

RabbitMQ是一个功能强大的消息代理，支持多种消息传递协议。它提供了丰富的插件和API，方便客户端应用程序进行集成和扩展。此外，RabbitMQ提供了丰富的可用性和容错功能，以确保消息不丢失。

因此，具体选择Kafka或RabbitMQ取决于您的业务需求和系统架构。如果您需要高吞吐量、分布式的数据处理，那么Kafka可能是更好的选择。如果您需要更强的消息代理功能，那么RabbitMQ可能是更好的选择。

## 评论功能使用Kafka进行消息队列

使用Kafka进行评论功能实现的具体思路如下：

1. 构建Kafka集群：首先，构建Kafka集群并运行。
2. 创建Topic：创建一个名为"comments"的topic，用于存储评论信息。
3. 消息生产者：在评论功能中，创建消息生产者，将用户发布的评论消息写入Kafka集群中的"comments" topic。
4. 敏感词过滤：在消息生产者中，使用前缀树进行敏感词过滤。如果评论中存在敏感词，则不写入Kafka集群中。
5. 消息消费者：创建消息消费者，从Kafka集群中的"comments" topic读取评论信息。
6. 消息存储：在消息消费者中，将读取到的评论信息存储到数据库中。
7. 发布评论：在前端页面，用户发布评论，将评论消息写入Kafka集群中的"comments" topic，并进行敏感词过滤。

8. 展示评论：在前端页面，从数据库中读取评论信息并展示。

通过这种方式，可以在Kafka的分布式环境中实现评论功能，并对评论进行敏感词过滤。

## AI代码

github.com/confluentinc/confluent-kafka-go/kafka 不行，只支持Linux

## 工具类

```
package main

import (
    "context"
    "fmt"

    "github.com/segmentio/kafka-go"
)

// KafkaClient 是kafka工具类
type KafkaClient struct {
    producer *kafka.Writer
    consumer *kafka.Reader
}

// NewKafkaClient 创建kafka工具类
func NewKafkaClient(brokers []string, topic string) *KafkaClient {
    kafkaClient := &KafkaClient{}
    kafkaClient.producer = kafka.NewWriter(kafka.WriterConfig{
        Brokers: brokers,
        Topic:   topic,
        Balancer: &kafka.LeastBytes{},
    })
    kafkaClient.consumer = kafka.NewReader(kafka.ReaderConfig{
        Brokers: brokers,
        Topic:   topic,
        MinBytes: 10e3, // 10KB
        MaxBytes: 10e6, // 10MB
    })
    return kafkaClient
}

// ProduceMessage 生产消息
func (k *KafkaClient) ProduceMessage(message string) error {
    kafkaMessage := kafka.Message{
        Value: []byte(message),
    }
    return k.producer.WriteMessages(context.Background(), kafkaMessage)
```

```
package kafka

import (

```



```

type KafkaClient struct {
    producer *kafka.Producer
    consumer *kafka.Consumer
}

func NewKafkaClient(brokers string) (*KafkaClient, error) {
    kc := &KafkaClient{}
    var err error

    kc.producer, err = kafka.NewProducer(&kafka.ConfigMap{"bootstrap.servers": brokers})
    if err != nil {
        return nil, err
    }

    kc.consumer, err = kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          "group_id",
        "auto.offset.reset": "earliest",
    })
    if err != nil {
        return nil, err
    }

    return kc, nil
}

func (kc *KafkaClient) PublishMessage(topic string, key string, value string) error {
    deliveryChan := make(chan kafka.Event)
    defer close(deliveryChan)

    err := kc.producer.Produce(&kafka.Message{
        TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
        Key:             []byte(key),
        Value:           []byte(value),
    }, deliveryChan)
    if err != nil {
        return err
    }

    e := <-deliveryChan
    m := e.(*kafka.Message)

    if m.TopicPartition.Error != nil {
        return m.TopicPartition.Error
    }

    return nil
}

func (kc *KafkaClient) SubscribeTopics(topics []string) error {
    kc.consumer.SubscribeTopics(topics, nil)
    return nil
}

func (kc *KafkaClient) ReadMessage() (*kafka.Message, error) {
    return kc.consumer.ReadMessage(-1)
}

```

```

}

func (kc *KafkaClient) Close() {
    kc.producer.Close()
    kc.consumer.Close()
}

// ConsumeMessage 消费消息
func (k *KafkaClient) ConsumeMessage() (string, error) {
    m, err := k.consumer.FetchMessage(context.Background())
    if err != nil {
        return "", err
    }
    return string(m.Value), nil
}

// Close 关闭kafka工具类
func (k *KafkaClient) Close() {
    k.producer.Close()
    k.consumer.Close()
}

```

```

func main() {
    brokers := []string{"localhost:9092"}
    topic := "test"
    kafkaClient := NewKafkaClient(brokers, topic)
    defer kafkaClient.Close()

    message := "hello, kafka"
    err := kafkaClient.ProduceMessage(message)
    if err != nil {
        fmt.Printf("failed to produce message, error: %v\n", err)
        return
    }
    fmt.Println("message produced successfully")

    consumedMessage, err := kafkaClient.ConsumeMessage()
    if err != nil {
        fmt.Printf("failed to consume message, error: %v\n", err)
        return
    }
    fmt.Printf("message consumed: %s\n", consumedMessage)
}

```

```

kc, err := NewKafkaClient("localhost:9092")

```

## 评论调用过程

```

kc, err := kafka.NewKafkaClient("localhost:9092")
if err != nil {
    // handle error
}
defer kc.Close()

```

```

err = kc.SubscribeTopics([]string{"comments"})
if err != nil {
    // handle error
}

// publish comment to the topic "comments"
err = kc.PublishMessage("comments", comment.ID, comment.Content)
if err != nil {
    // handle error
}

// read messages from the topic "comments"
for {
    msg, err := kc.ReadMessage()
    if err != nil {
        // handle error
    }
    commentID := string(msg.Key)
    content := string(msg.Value)
    // process the comment
}

```

写的操作

控制层

```

func (cc *CommentController) PublishComment(comment Comment) error {
    return cc.kc.PublishMessage("comments", comment.ID, comment.Content)
}

```

存储层

```

type CommentStorage struct {
    kc *kafka.KafkaClient
    db *sql.DB
}

func NewCommentStorage() (*CommentStorage, error) {
    kc, err := kafka.NewKafkaClient("localhost:9092")
    if err != nil {
        return nil, err
    }
    err = kc.SubscribeTopics([]string{"comments"})
    if err != nil {
        return nil, err
    }

    db, err := sql.Open("postgres", "postgres://user:password@localhost/comments?sslmode=disable")
    if err != nil {
        return nil, err
    }

    return &CommentStorage{kc: kc, db: db}, nil
}

func (cs *CommentStorage) Start() {

```

```

for {
    msg, err := cs.kc.ReadMessage()
    if err != nil {
        // handle error
    }
    commentID := string(msg.Key)
    content := string(msg.Value)

    stmt, err := cs.db.Prepare("INSERT INTO comments(id, content) VALUES($1, $2)")
    if err != nil {
        // handle error
    }
    _, err = stmt.Exec(commentID, content)
    if err != nil {
        // handle error
    }
}
}

```

读的操作

```

package comment

import (
    "fmt"

    "github.com/gomodule/redigo/redis"
)

// Comment struct
type Comment struct {
    ID      int64
    Content string
    Author  string
}

// LoadComments loads comments from cache and mysql
func LoadComments(page int) ([]Comment, error) {
    var comments []Comment

    // Try to get comments from cache
    conn := redis.Dial("tcp", "redis:6379")
    defer conn.Close()

    comments, err := redis.Values(conn.Do("GET", fmt.Sprintf("comments:%d", page)))
    if err == nil {
        return comments, nil
    }

    // If comments not found in cache, load from MySQL
    comments, err = loadCommentsFromMySQL(page)
    if err != nil {
        return nil, err
    }
}

```

```

// Store comments in cache for next time
conn.Do("SET", fmt.Sprintf("comments:%d", page), comments)

// Publish a message to the kafka topic to notify about new comments
publishCommentToKafka(comments)

return comments, nil
}

// publishCommentToKafka publishes the comments to kafka topic
func publishCommentToKafka(comments []Comment) error {
    // Code to publish comments to kafka topic

    return nil
}

// loadCommentsFromMySQL loads comments from MySQL
func loadCommentsFromMySQL(page int) ([]Comment, error) {
    // Code to load comments from MySQL

    return comments, nil
}

```

```

func GetComments(page int) ([]Comment, error) {
    // 加载缓存
    comments, err := LoadFromCache(page)
    if err == nil {
        return comments, nil
    }

    // 没有缓存则从数据库中加载
    comments, err = LoadFromDB(page)
    if err != nil {
        return nil, err
    }

    // 将数据缓存到redis中
    go CacheComments(page, comments)

    // 异步预加载下一页数据
    go PreloadComments(page + 1)

    return comments, nil
}

// 加载评论数据（从数据库中读取）
func LoadFromDB(page int) ([]Comment, error) {
    // ... 从数据库读取数据的代码 ...
    return comments, nil
}

// 缓存评论数据到redis
func CacheComments(page int, comments []Comment) error {
    // ... 缓存到redis中的代码 ...
    return nil
}

```

```

}

// 预加载评论数据
func PreloadComments(page int) {
    // 从数据库读取评论数据
    comments, err := LoadFromDB(page)
    if err != nil {
        return
    }

    // 将评论数据发送到kafka消息队列
    err = KafkaProducer.SendComment(comments)
    if err != nil {
        return
    }
}

```

## 架构设计

[https://blog.csdn.net/qg\\_61039408/article/details/128729820](https://blog.csdn.net/qg_61039408/article/details/128729820)

使用MySQL进行存储的话，就必须要用到Redis来做缓存，后台admin需要接通ES来进行查询，comment-service通过异步来进行写Redis和MySQL评论数据，MySQL和ES通过Canal进行binlog同步。

### 缓存模式

首先我们需要预读，我们读第一页的时候，也需要把第二页的内容加载出来。读第二页的时候，我们预先读第三页，这样可以避免大量的cache miss。

但是这里有一个致命的问题就是：当缓存抖动的时候，会触发大量的cache rebuild，因为我们使用了预加载，容易造成OOM(内存溢出)。因此我们需要使用消息队列来进行逻辑异步化，对于当前请求，只返回MySQL中的部分数据即可。

### 写的逻辑

至于写的操作，我们要穿透到存储层，因此最好使用消息队列异步削峰。例如我的评论发布出去了，用户过100ms才看到评论，这是无所谓的。

## 配置

```
go get -u github.com/confluentinc/confluent-kafka-go/kafka
```

## docker部署Kafka

<https://aijishu.com/a/1060000000091083>

[https://blog.csdn.net/weixin\\_42259081/article/details/112370700](https://blog.csdn.net/weixin_42259081/article/details/112370700)

连接不上解决方法：

<https://www.cnpython.com/java/668022>

其中：

- `-d`：表示容器在后台运行；
- `--name kafka`：为容器命名为kafka；

- `-p 2181:2181`: 将容器中的2181端口映射到主机的2181端口;
- `-p 9092:9092`: 将容器中的9092端口映射到主机的9092端口;
- `--env ADVERTISED_HOST=<host_name>`: 指定Kafka集群的主机名, 请替换为你的实际主机名;
- `--env ADVERTISED_PORT=9092`: 指定Kafka集群的端口, 默认为9092。

注意下面必须写主机IP, 否则连接不上

```
# 1. 下载Kafka镜像:
docker pull spotify/kafka

# 2. 启动容器
docker run -p 2181:2181 -p 9092:9092 --env ADVERTISED_HOST=43.139.72.246 -d --env ADVERTISED_PORT=9092
spotify/kafka
```

## 代码

### 写逻辑

- (1) 前端评论;
- (2) 后端接收评论, 然后用消息队列的生产者给队列发送一个评论消息, 返回给前端评论成功;
- (3) 一开始就有一个线程是开启了消息队列监听的, 一旦消息队列有新的消息, 消息队列的消费者就会获取这个消息进行处理, 保存到数据库中。

utils/kafka\_client/kafka\_client.go

这里是工具类

```
package kafka_client

import (
    "context"
    "github.com/segmentio/kafka-go")

// KafkaClient 封装了kafka-go库中的生产者和消费者
type KafkaClient struct {
    // producer 用于生产消息
    producer *kafka.Writer
    // consumer 用于消费消息
    consumer *kafka.Reader
}

// NewKafkaClient 创建一个Kafka工具类
func NewKafkaClient(brokers []string, topic string) *KafkaClient {
    kafkaClient := &KafkaClient{}
    // 创建生产者
    kafkaClient.producer = kafka.NewWriter(kafka.WriterConfig{
        Brokers: brokers,
        Topic:   topic,
        Balancer: &kafka.LeastBytes{},
    })
    // 创建消费者
    kafkaClient.consumer = kafka.NewReader(kafka.ReaderConfig{
        Brokers: brokers,
        Topic:   topic,
        MinBytes: 10e3, // 10KB
    })
}
```

```

        MaxBytes: 10e6, // 10MB
    })
    return kafkaClient
}

// ProduceMessage 生产消息
func (k *KafkaClient) ProduceMessage(message string) error {
    // 封装消息
    kafkaMessage := kafka.Message{
        Value: []byte(message),
    }
    // 发送消息
    return k.producer.WriteMessages(context.Background(), kafkaMessage)
}

// ConsumeMessage 消费消息
func (k *KafkaClient) ConsumeMessage() (string, error) {
    // 获取消息
    m, err := k.consumer.FetchMessage(context.Background())
    if err != nil {
        return "", err
    }
    return string(m.Value), nil
}

// Close 关闭Kafka工具类，释放资源
func (k *KafkaClient) Close() {
    k.producer.Close()
    k.consumer.Close()
}

```

global/variable/variable.go  
定义全局变量（Kafka 和 Trie）

```

package variable

import (
    "bufio"
    "fmt"    "github.com/spf13/viper"    "github.com/willf/bloom"    "go.uber.org/zap"
    "go_douyin/global/my_errors"    "go_douyin/utils/kafka_client"    "go_douyin/utils/sensitive_word_filter"
    _ "gorm.io/gorm"
    "log"    "os")

// 全局变量（注意首字母大写）
var (
    BasePath string // 定义项目的根目录

    // 全局日志指针
    ZapLog *zap.Logger

    // 全局配置文件
    Config *viper.Viper

    // 创建布隆过滤器
    Filter *bloom.BloomFilter

```



```

//全局消息队列
Kafka *kafka_client.KafkaClient

//全局敏感词过滤
Trie *sensitive_word_filter.Trie
)

// 检查项目必须的非编译目录是否存在，避免编译后调用的时候缺失相关目录
func checkRequiredFolders() {
    //1.检查配置文件是否存在
    if _, err := os.Stat(BasePath + "/config/config.yml"); err != nil {
        log.Fatal(my_errors.ErrorsConfigYamlNotExists + err.Error())
    }
    //2.检查storage/logs 目录是否存在
    if _, err := os.Stat(BasePath + "/storage/logs/"); err != nil {
        log.Fatal(my_errors.ErrorsStorageLogsNotExists + err.Error())
    }
}

func Init() {
    //1.检查配置文件以及日志目录等非编译性的必要条件
    //checkRequiredFolders()
    //2.初始化布隆过滤器
    Filter = bloom.New(1000000, 5)
    // 3.创建监听评论的消息队列（后面改到配置那里）
    Kafka = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "comment-topic")
    // 4.创建敏感词过滤树
    fmt.Println("创建敏感词前缀树")
    Trie = sensitive_word_filter.NewTrie()
    // 从文件中读取敏感词
    file, _ := os.Open(BasePath + "/config/sensitive_words.txt")
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        Trie.Insert(scanner.Text())
    }
}
}

```

router/router.go  
(关键部分)

```

commentController := controller.NewCommentController()

// 社交组：评论功能
v4 := router.Group("/douyin/comment")
{
    v4.POST("action", commentController.AddComment)
}

```

commentController.go  
控制层

```

package controller

import (
    "fmt"
    "github.com/gin-gonic/gin"    "go_douyin/model"    "go_douyin/service/comment"
    "go_douyin/utils/response"    "time")

type CommentController struct {
    commentService *comment.CommentService
}

func NewCommentController() *CommentController {
    return &CommentController{commentService: comment.NewCommentService()}
}

// 评论列表（此处只是个demo）
func (h *CommentController) AddComment(c *gin.Context) {
    // 获取请求参数
    var requestBody map[string]interface{}
    requestBody = make(map[string]interface{})
    // 解析请求体
    c.ShouldBindJSON(&requestBody)
    fmt.Println(requestBody)
    // 获取请求参数
    //在 HTTP POST 请求中，请求体中的数据通常是以字符串形式发送的。JSON 格式中的数字默认都是浮点型，默认都是
    float64 类型
    video_id := requestBody["video_id"].(float64)
    comment_text, _ := requestBody["comment_text"].(string)
    var cc model.Comment
    cc.Content = comment_text
    // 正常是从token解析，这里是demo
    cc.UserID = 1
    cc.VideoID = uint64(video_id)
    cc.CreateTime = time.Now()
    err := h.commentService.AddComment(cc)
    if err != nil {
        response.Success(c, "评论失败", gin.H{})
        return
    }
    response.Success(c, "评论成功", gin.H{})
}

```

service/comment/commentService.go  
 评论放到消息队列中

```

package comment

import (
    "encoding/json"
    "fmt"    "go_douyin/dao"    "go_douyin/global/variable"    "go_douyin/model")

type CommentService struct {
    commentMapper *dao.CommentMapper
}

func NewCommentService() *CommentService {

```

```

    return &CommentService{
        commentMapper: dao.NewCommentMapper(),
    }
}

// AddComment 添加评论
func (h *CommentService) AddComment(comment model.Comment) error {
    // 进行敏感词过滤
    comment.Content = variable.Trie.Filter(comment.Content)
    fmt.Println(comment.Content)
    // 序列化评论数据
    commentJSON, err := json.Marshal(comment)
    if err != nil {
        return err
    }
    // 将评论放到消息队列
    err = variable.Kafka.ProduceMessage(string(commentJSON))
    return err
}

```

dao/commentMapper.go

这里有监听消息队列的函数，需要一开始就开启

```

package dao

import (
    "encoding/json"
    "fmt"    "go_douyin/global/variable"    "go_douyin/model")

type CommentMapper struct{}

func NewCommentMapper() *CommentMapper {
    return &CommentMapper{}
}

// 监听消息队列
func ListenComment() {
    for {
        // 获取消息
        message, err := variable.Kafka.ConsumeMessage()
        if err != nil {
            fmt.Printf("获取消息失败: %v\n", err)
            continue
        }
        // 反序列化评论数据
        var comment model.Comment
        err = json.Unmarshal([]byte(message), &comment)
        if err != nil {
            fmt.Printf("反序列化评论数据失败: %v\n", err)
            continue
        }
        // 存储评论数据
        err = SaveComment(comment)
        if err != nil {
            fmt.Printf("存储评论数据失败: %v\n", err)
            continue
        }
    }
}

```

```

    }
    fmt.Printf("成功存储评论数据: %+v\n", comment)
}
}

func SaveComment(comment model.Comment) error {
    // 这里是将评论数据存储到数据库的代码，具体实现方式取决于你使用的数据库类型
    fmt.Println("正在存储数据库", comment)
    return nil
}

```

comment.go

```

package model

import (
    "time"
)

// 评论实体类
type Comment struct {
    CommentID uint64 `json:"comment_id"` // comment_id
    UserID    uint64 `json:"user_id"`    // user_id
    VideoID   uint64 `json:"video_id"`    // video_id
    Content    string `json:"content"`      // content
    CreateTime time.Time `json:"create_time"` // create_time
}

```

main.go

注意这里创建协程监听消息队列

```

package main

import (
    "go_douyin/config"
    "go_douyin/dao"    "go_douyin/database"    "go_douyin/global/variable"    "go_douyin/router")

func main() {
    // 1.初始化配置，读取配置
    config.Init()
    // 2.初始化全局变量
    variable.Init()
    // 注意初始化数据库
    database.SqlClient()

    // 5.创建协程监听消息队列
    go dao.ListenComment()

    variable.ZapLog.Info("程序正在运行")
    r := router.SetupRouter()
    r.Run(":8081")
}

```

## 预加载评论

service/comment/commentService.go

```
// 预加载评论列表
func (h *CommentService) PreloadCommentList(video_id uint64) error {
    fmt.Printf("执行预加载")
    // 将预加载的视频id放到消息队列
    err := variable.Kafka_preload.ProduceMessage(string(video_id))
    return err
}
```

dao/commentMapper.go

```
// 监听预加载评论消息队列
func ListenPreloadCommentList() {
    for {
        fmt.Printf("正在预加载.....")
        // 获取消息
        message, err := variable.Kafka_preload.ConsumeMessage()
        if err != nil {
            fmt.Printf("获取消息失败: %v\n", err)
            continue
        }
        // 此次应写存储到缓存的函数
        fmt.Printf("正在缓存" + message + "视频id的评论")
    }
}
```

global/variable/variable.go

```
package variable

import (
    "fmt"
    "github.com/spf13/viper"    "github.com/syyongx/go-wordsfilter"    "github.com/willf/bloom"
    "go.uber.org/zap"    "go_douyin/global/my_errors"    "go_douyin/utils/kafka_client"    _ "gorm.io/gorm"
    "log"    "os")

// 全局变量（注意首字母大写）
var (
    BasePath string // 定义项目的根目录

    // 全局日志指针
    ZapLog *zap.Logger

    // 全局配置文件
    Config *viper.Viper

    // 创建布隆过滤器
    Filter *bloom.BloomFilter

    // 全局消息队列
    // 评论的队列
    Kafka *kafka_client.KafkaClient
    // 预加载的队列
    Kafka_preload *kafka_client.KafkaClient
}
```

```

//全局敏感词过滤
Trie *wordsfilter.WordsFilter
Root map[string]*wordsfilter.Node
)

// 检查项目必须的非编译目录是否存在，避免编译后调用的时候缺失相关目录
func checkRequiredFolders() {
    //1.检查配置文件是否存在
    if _, err := os.Stat(BasePath + "/config/config.yml"); err != nil {
        log.Fatal(my_errors.ErrorsConfigYamlNotExists + err.Error())
    }
    //2.检查storage/logs 目录是否存在
    if _, err := os.Stat(BasePath + "/storage/logs/"); err != nil {
        log.Fatal(my_errors.ErrorsStorageLogsNotExists + err.Error())
    }
}

func Init() {
    //1.检查配置文件以及日志目录等非编译性的必要条件
    //checkRequiredFolders()
    //2.初始化布隆过滤器
    Filter = bloom.New(1000000, 5)
    // 3.创建监听评论的消息队列（后面改到配置那里）
    Kafka = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "comment-topic")
    Kafka_preload = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "comment-preload-
topic")

    // 4.创建敏感词过滤树
    fmt.Println("创建敏感词前缀树")
    Trie = wordsfilter.New()
    Root, _ = Trie.GenerateWithFile(BasePath + "/config/sensitive_words.txt")
}

```

main.go

```

package main

import (
    "go_douyin/config"
    "go_douyin/dao"    "go_douyin/database"    "go_douyin/global/variable"    "go_douyin/router")

func main() {
    // 1.初始化配置，读取配置
    config.Init()
    // 2.初始化全局变量
    variable.Init()
    // 注意初始化数据库
    database.SqlClient()

    // 5.创建协程监听评论的消息队列
    go dao.ListenComment()
    go dao.ListenPreloadCommentList()

    variable.ZapLog.Info("程序正在运行")
    r := router.SetupRouter()
}

```

```
r.Run(":8081")
}
```

## 运行

时间快挺多的

```
[GIN-debug] Listening and serving HTTP on :8081
2023-02-12T10:47:19.225+0800 INFO main/main.go:21 程序正在运行
map[comment_id:1 comment_text:快手抖音谁最好, 当然是快手 video_id:51]
[GIN] 2023/02/12 - 10:47:24 | 200 | 1.2698553s | 127.0.0.1 | POST "/douyin/comment/action"
正在存储数据库 {0 1 51 快手抖音谁最好, 当然是快手 2023-02-12 10:47:22.7527973 +0800 CST}
成功存储评论数据: {CommentID:0 UserID:1 VideoID:51 Content:快手抖音谁最好, 当然是快手 CreateTime:2023-02-12 10:47:22.7527973 +0800 CST}
```

(2023.2.17)

## 点赞

<https://dandelioncloud.cn/article/details/1517827650398842881>

## 系统架构

1. 数据存储：使用 Redis 作为数据存储，每个视频对应一个 key，value 为点赞数。此外，还可以在 Redis 中记录每个用户点赞的视频。例如：

```
HSET video:1 likes 1 SADD user:1:liked_videos 1
```

2. 用户认证：使用 JWT 验证用户身份，在请求点赞接口时需要带上 JWT token。
3. 接口设计：设计两个接口：
  - 点赞接口：对指定视频进行点赞，比如 `/video/{id}/like`，该接口需要验证 JWT token。
  - 获取点赞视频列表接口：获取当前用户点赞的视频列表，比如 `/user/{id}/liked_videos`，该接口需要验证 JWT token。
4. 缓存：使用 Redis 缓存点赞数据，可以使用 Redis 的高速缓存特性，并使用 LRU 算法控制缓存的大小。
5. 数据一致性：可以使用 Redis 的事务机制来保证数据的一致性，确保多个用户同时点赞时不会造成数据错误。例如：

```
MULTI HINCRBY video:1 likes 1 SADD user:1:liked_videos 1 EXEC
```

## 学习代码

### LRU算法控制缓存大小

使用 Redis 的 LRU 算法需要使用 Redis 的命令 `maxmemory-policy` 来设置。示例代码如下：

```
package main

import (
    "github.com/go-redis/redis"
)

func main() {
    client := redis.NewClient(&redis.Options{
```

```

    Addr:      "localhost:6379",
    Password:  "", // no password set
    DB:        0,  // use default DB
})

// Set maxmemory policy to LRU
_, err := client.ConfigSet("maxmemory-policy", "LRU").Result()
if err != nil {
    panic(err)
}

// Increment the likes count for a video
err = client.HIncrBy("video:1", "likes", 1).Err()
if err != nil {
    panic(err)
}

// Add the video to the user's liked videos set
err = client.SAdd("user:1:liked_videos", 1).Err()
if err != nil {
    panic(err)
}
}

```

## 使用 Redis 事务保证数据一致性

```

package main

import (
    "github.com/go-redis/redis"
)

func main() {
    client := redis.NewClient(&redis.Options{
        Addr:      "localhost:6379",
        Password:  "", // no password set
        DB:        0,  // use default DB
    })

    // Start a transaction
    tx := client.TxPipeline()
    defer tx.Close()

    // Increment the likes count for a video
    incr := tx.HIncrBy("video:1", "likes", 1)
    // Add the video to the user's liked videos set
    add := tx.SAdd("user:1:liked_videos", 1)

    // Execute the transaction
    _, err := tx.Exec(incr, add)
    if err != nil {
        panic(err)
    }
}

```



## 代码

praiseService.go

```
package praise

import (
    "fmt"
    "go_douyin/utils/redis")

type PraiseService struct {
    redisClient *redis.RedisClient
}

func NewPraiseService() *PraiseService {
    return &PraiseService{
        redisClient: redis.NewRedisClient(),
    }
}

// 为指定视频点赞
func (r *PraiseService) LikeVideo(userId, videoId string) error {
    // 通过 Redis 事务保证数据一致性
    r.redisClient.ExecTxWithWatch(func() error {
        // 检查用户是否已点赞该视频
        userLikedKey := fmt.Sprintf("user:%s:liked_videos", userId)
        exists, err := r.redisClient.SIsMember(userLikedKey, videoId)
        if err != nil {
            return err
        }
        if exists {
            return fmt.Errorf("user %s has already liked video %s", userId, videoId)
        }

        // 点赞数增加 1      videoLikeKey := fmt.Sprintf("video:%s:likes", videoId)
        _, err = r.redisClient.HIncrBy(videoLikeKey, "likes", 1)
        if err != nil {
            return err
        }

        // 用户点赞的视频集合中加入该视频
        _, err = r.redisClient.SAdd(userLikedKey, videoId)
        if err != nil {
            return err
        }
        return nil
    })
    return nil
}

// 取消指定视频点赞
func (r *PraiseService) DislikeVideo(userId, videoId string) error {
    // 通过 Redis 事务保证数据一致性
    r.redisClient.ExecTxWithWatch(func() error {
        // 检查用户是否已经点赞了该视频
        userLikedKey := fmt.Sprintf("user:%s:liked_videos", userId)
```

```

    exists, err := r.redisClient.SIsMember(userLikedKey, videoId)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("user %s has not liked video %s", userId, videoId)
    }

    // 取消点赞，点赞数减 1      videoLikeKey := fmt.Sprintf("video:%s:likes", videoId)
    _, err = r.redisClient.HIncrBy(videoLikeKey, "likes", -1)
    if err != nil {
        return err
    }

    // 用户点赞的视频集合中移除该视频
    _, err = r.redisClient.SRem(userLikedKey, videoId)
    if err != nil {
        return err
    }
    return nil
})
return nil
}

// 获取指定用户点赞的视频列表（这里也应该存储到数据库）
func (r *PraiseService) GetLikedVideos(userId string) ([]string, error) {
    userLikedKey := fmt.Sprintf("user:%s:liked_videos", userId)
    videoIds, err := r.redisClient.SMembers(userLikedKey)
    if err != nil {
        return nil, err
    }
    return videoIds, nil
}

```

redis-client.go

```

package redis

import (
    "encoding/json"
    "github.com/gomodule/redigo/redis"    "go_douyin/global/variable"    "time")

type RedisClient struct {
    Pool *redis.Pool
}

func NewRedisClient() *RedisClient {
    return &RedisClient{
        Pool: &redis.Pool{
            MaxIdle:    3,
            IdleTimeout: 240 * time.Second,
            Dial: func() (redis.Conn, error) {
                c, err := redis.Dial("tcp",
                    variable.Config.GetString("redis.host")+":"+variable.Config.GetString("redis.port"),
                    redis.DialPassword(variable.Config.GetString("redis.auth")))
                if err != nil {

```

```

        return nil, err
    }
    return c, err
},
TestOnBorrow: func(c redis.Conn, t time.Time) error {
    _, err := c.Do("PING")
    return err
},
},
}

func (c *RedisClient) Set(key string, value interface{}) error {
    jsonData, err := json.Marshal(value)
    if err != nil {
        return err
    }

    conn := c.Pool.Get()
    defer conn.Close()

    _, err = conn.Do("SET", key, jsonData)
    return err
}

func (c *RedisClient) SetWithExpire(key string, value interface{}, expire int) error {
    jsonData, err := json.Marshal(value)
    if err != nil {
        return err
    }

    conn := c.Pool.Get()
    defer conn.Close()

    _, err = conn.Do("SET", key, jsonData, "EX", expire)
    return err
}

func (c *RedisClient) Get(key string, value interface{}) error {
    conn := c.Pool.Get()
    defer conn.Close()

    jsonData, err := redis.Bytes(conn.Do("GET", key))
    if err != nil {
        return err
    }

    return json.Unmarshal(jsonData, value)
}

// 获取集合中的所有成员
func (c *RedisClient) SMembers(key string) ([]string, error) {
    conn := c.Pool.Get()
    defer conn.Close()

    return redis.Strings(conn.Do("SMEMBERS", key))
}

```

```

// 对哈希表中的字段自增
func (c *RedisClient) HIncrBy(key, field string, incr int64) (int64, error) {
    conn := c.Pool.Get()
    defer conn.Close()

    return redis.Int64(conn.Do("HINCRBY", key, field, incr))
}

// 判断元素是否是集合的成员
func (c *RedisClient) SIsMember(key, member string) (bool, error) {
    conn := c.Pool.Get()
    defer conn.Close()

    return redis.Bool(conn.Do("SISMEMBER", key, member))
}

// 设置分布式锁(SET key value NX PX milliseconds)
func (c *RedisClient) AcquireLock(lockKey string, timeout int) bool {
    conn := c.Pool.Get()
    defer conn.Close()
    _, err := conn.Do("SET", lockKey, 1, "EX", timeout, "NX")
    if err == nil {
        return true
    }
    return false
}

// 释放锁
func (c *RedisClient) ReleaseLock(lockKey string) {
    conn := c.Pool.Get()
    defer conn.Close()
    conn.Do("DEL", lockKey)
}

// 执行带有乐观锁的事务
// 函数循环执行 Redis 事务，直到事务成功执行为止。在每次循环中，函数调用 WATCH 命令监视 mykey 键。如果事务执行失败，函数会丢弃事务并继续循环。如果事务执行成功，函数跳出循环并返回 nil。
func (c *RedisClient) ExecTxWithWatch(fn func() error) error {
    conn := c.Pool.Get()
    defer conn.Close()
    for {
        // 开始执行事务
        _, err := conn.Do("WATCH", "mykey")
        if err != nil {
            return err
        }

        // 开始一个新的 Redis 事务
        _, err = conn.Do("MULTI")
        if err != nil {
            return err
        }

        // 调用 fn 函数执行 Redis 事务代码
        if err := fn(); err != nil {
            conn.Do("DISCARD")
        }
    }
}

```

```

        return err
    }

    // 尝试执行 Redis 事务
    _, err = conn.Do("EXEC")
    if err != nil {
        // 如果执行失败，继续循环
        continue
    } else if err != nil {
        return err
    }

    // 如果事务执行成功，跳出循环
    break
}

return nil
}

// 给set集合添加元素
func (c *RedisClient) SAdd(key string, values ...interface{}) (int64, error) {
    conn := c.Pool.Get()
    defer conn.Close()

    args := []interface{}{key}
    args = append(args, values...)

    result, err := redis.Int64(conn.Do("SADD", args...))
    if err != nil {
        return 0, err
    }

    return result, nil
}

// 从set集合中移除元素
func (c *RedisClient) SRem(key string, values ...interface{}) (int64, error) {
    conn := c.Pool.Get()
    defer conn.Close()
    args := []interface{}{key}
    args = append(args, values...)

    result, err := redis.Int64(conn.Do("SREM", args...))
    if err != nil {
        return 0, err
    }

    return result, nil
}

```

(2023.2.12)

## 敏感词过滤（现成第三方库，推荐）

### 配置

```
go get -u "github.com/syyongx/go-wordsfilter"
```

## 教程

[https://blog.csdn.net/qq\\_37575994/article/details/128325081](https://blog.csdn.net/qq_37575994/article/details/128325081)

## 代码

service/comment/commentService.go

```
// 进行敏感词过滤
comment.Content = variable.Trie.Replace(comment.Content, variable.Root)
```

global/variable/variable.go

```
package variable

import (
    "fmt"
    "github.com/spf13/viper"    "github.com/syyongx/go-wordsfilter"    "github.com/willf/bloom"
    "go.uber.org/zap"    "go_douyin/global/my_errors"    "go_douyin/utils/kafka_client"    _ "gorm.io/gorm"
    "log"    "os")

// 全局变量（注意首字母大写）
var (
    BasePath string // 定义项目的根目录

    // 全局日志指针
    ZapLog *zap.Logger

    // 全局配置文件
    Config *viper.Viper

    // 创建布隆过滤器
    Filter *bloom.BloomFilter

    // 全局消息队列
    // 评论的队列
    Kafka *kafka_client.KafkaClient
    // 预加载的队列
    Kafka_preload *kafka_client.KafkaClient
    //全局敏感词过滤
    Trie *wordsfilter.WordsFilter
    Root map[string]*wordsfilter.Node
)

// 检查项目必须的非编译目录是否存在，避免编译后调用的时候缺失相关目录
func checkRequiredFolders() {
    //1.检查配置文件是否存在
    if _, err := os.Stat(BasePath + "/config/config.yml"); err != nil {
        log.Fatal(my_errors.ErrorsConfigYamlNotExists + err.Error())
    }
    //2.检查storage/logs 目录是否存在
}
```

```

    if _, err := os.Stat(BasePath + "/storage/logs/"); err != nil {
        log.Fatal(my_errors.ErrorsStorageLogsNotExists + err.Error())
    }

}

func Init() {
    //1.检查配置文件以及日志目录等非编译性的必要条件
    //checkRequiredFolders()
    //2.初始化布隆过滤器
    Filter = bloom.New(1000000, 5)
    // 3.创建监听评论的消息队列（后面改到配置那里）
    Kafka = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "comment-topic")
    Kafka_preload = kafka_client.NewKafkaClient([]string{"43.139.72.246:9092"}, "comment-preload-
topic")

    // 4.创建敏感词过滤树
    fmt.Println("创建敏感词前缀树")
    Trie = wordsfilter.New()
    Root, _ = Trie.GenerateWithFile(BasePath + "/config/sensitive_words.txt")
}

```

test/sensitive\_test.go

单元测试

```

func TestSensitive2(t *testing.T) {
    wf := wordsfilter.New()
    root, _ := wf.GenerateWithFile("../config/sensitive_words.txt")
    //替换
    newStr := wf.Replace("不可描快", root)
    //打印替换好的文本
    fmt.Println(newStr)
}

```

(2023.2.5)

## 敏感词过滤（不使用这里的代码）

### 学习

正则表达式的效率比较低，在大规模的敏感词过滤场景中可能存在性能问题。使用Trie树作为敏感词过滤的算法可以提高效率，同时在过滤时也比较容易实现。

### 正则表达式（不推荐）

```

//1. 读取敏感词列表：
sensitiveWords := []string{"敏感词1", "敏感词2", "敏感词3"}

//2. 构造正则表达式：
regexStr := "("
for i, word := range sensitiveWords {
    if i == 0 {
        regexStr += word
    } else {
        regexStr += "|" + word
    }
}

```

```

    }
}
regexStr += ")"
sensitiveRegex, err := regexp.Compile(regexStr)
if err != nil {
    panic(err)
}

//3. 对文本进行过滤:
func filterText(text string) string {
    return sensitiveRegex.ReplaceAllString(text, "***")
}

```

## 前缀树 (推荐)

在项目中, 需要先调用 `buildTrieTree` 方法, 将敏感词列表构造成Trie树, 然后再使用 `filterText` 方法对文本进行过滤。

```

//1. 定义Trie树节点:
type TrieNode struct {
    children map[rune]*TrieNode
    end      bool
}

func NewTrieNode() *TrieNode {
    return &TrieNode{children: make(map[rune]*TrieNode)}
}

//2. 构造Trie树:
func buildTrieTree(words []string) *TrieNode {
    root := NewTrieNode()
    for _, word := range words {
        node := root
        for _, r := range word {
            if _, ok := node.children[r]; !ok {
                node.children[r] = NewTrieNode()
            }
            node = node.children[r]
        }
        node.end = true
    }
    return root
}

//3. 对文本进行过滤:
func filterText(root *TrieNode, text string) string {
    var result []rune
    for i, r := range text {
        node := root
        j := i
        for node != nil {
            node = node.children[r]
            if node != nil && node.end {
                for ; j < len(text); j++ {
                    result = append(result, '*')
                }
            }
        }
    }
}

```



```

        break
    }
    if j < len(text) - 1 {
        result = append(result, r)
        j++
        r = []rune(text)[j]
    } else {
        result = append(result, r)
        break
    }
}
}
return string(result)
}

```

## 代码

sensitive\_word\_filter/sensitive\_word\_filter.go

```

package sensitive_word_filter

const replaceString = "***"

// TrieNode 前缀树的结点
type TrieNode struct {
    children map[rune]*TrieNode
    isEnd    bool
}

// Trie 前缀树
type Trie struct {
    root *TrieNode
}

// NewTrie 创建新的前缀树
func NewTrie() *Trie {
    return &Trie{
        root: &TrieNode{
            children: make(map[rune]*TrieNode),
            isEnd:    false,
        },
    }
}

// Insert 插入一个敏感词
func (t *Trie) Insert(word string) {
    node := t.root
    for _, char := range word {
        if _, ok := node.children[char]; !ok {
            node.children[char] = &TrieNode{
                children: make(map[rune]*TrieNode),
                isEnd:    false,
            }
        }
        node = node.children[char]
    }
}

```

```

    node.isEnd = true
}

// Search 查找敏感词
func (t *Trie) Search(word string) bool {
    node := t.root
    for _, char := range word {
        if _, ok := node.children[char]; !ok {
            return false
        }
        node = node.children[char]
    }
    return node.isEnd
}

// Filter 过滤敏感词
func (t *Trie) Filter(text string) string {
    // 加上一个字符，方便敏感词在最后的处理
    text = text + "!"
    // 设置当前节点为根节点
    node := t.root
    // 记录单词的开始位置
    start := 0
    // 用来存储过滤后的字符串
    var result []rune
    // 遍历字符串
    for i, char := range text {
        // 如果当前字符不在字典树中，说明当前子串不存在于字典树中。
        if _, ok := node.children[char]; !ok {
            // 如果当前单词的起始位置不等于结束位置，说明之前已经找到了一个单词，则需要将该单词替换成
            replaceString      if start != i && node.isEnd {
                result = append(result, []rune(replaceString)...)
            }
            // 直接加入过滤后的字符串
            result = append(result, char)
            // 重置当前节点为根节点，起始位置为下一个字符位置
            node = t.root
            start = i + 1
        } else {
            // 如果当前 Trie 节点存在该字符，说明存在包含该字符的单词
            // 如果该 Trie 节点是一个单词的结尾，则将单词的起始位置设为当前字符位置
            if node.isEnd {
                result = append(result, []rune(replaceString)...)
                start = i + 1
            }
            node = node.children[char]
        }
    }
    res := string(result)
    // 去掉刚才的字符
    return res[:len(res)-1]
}

```

test/sensitive\_test.go

```
//测试敏感词过滤
func TestSensitive(t *testing.T) {
    // 创建前缀树
    trie := sensitive_word_filter.NewTrie()

    // 从文件中读取敏感词
    file, _ := os.Open("../config/sensitive_words.txt")
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        trie.Insert(scanner.Text())
    }

    // 过滤敏感词
    text := "这是一段评论，快手,里面有敏感词"
    fmt.Println("原始评论:", text)
    filteredText := trie.Filter(text)
    fmt.Println("过滤后评论:", filteredText)
}
```

config/sensitive\_words.txt

```
快手
微信
QQ
```

(2023.2.4)

## 协同过滤推荐算法

### AI代码

### 简单例子

以下是一个简单的例子，使用Golang和Gonum库实现矩阵分解协同过滤：

```
package main

import (
    "fmt"

    "gonum.org/v1/gonum/mat"
)

func main() {
    // Load the video data into a matrix
    videos := mat.NewDense(5, 5, []float64{
        5, 3, 0, 1, 4,
        4, 0, 0, 1, 1,
        1, 1, 0, 5, 4,
        0, 0, 0, 4, 0,
        0, 1, 5, 4, 5,
    })
}
```

```

// Perform matrix factorization to get the user and item latent features
k := 2
u, s, vt := mat.SVD(videos, k, k, nil, nil)

// Predict the ratings for the items
predictedRatings := &mat.Dense{}
predictedRatings.Mul(u, s)
predictedRatings.Mul(predictedRatings, vt)

fmt.Println("Predicted Ratings:")
fmt.Printf("%0.2f\n", mat.Formatted(predictedRatings, mat.Prefix(" "), mat.Excerpt(3)))
}

```

## 小例子

一种实现方法是使用基于用户的协同过滤算法。该算法需要用户-物品评分矩阵，其中一行表示一个用户，一列表示一个物品，并且单元格的值表示该用户对该物品的评分。

您可以将用户的点赞数据存储在评分矩阵中，然后使用该矩阵对每个用户的评分进行预测。在预测过程中，您可以使用用户与其他用户之间的相似度来评估用户对物品的评分。

一旦评分预测完成，您可以找到该用户评分最高的物品，并推荐给该用户。

这个代码使用协同过滤算法，来预测用户可能喜欢的视频。在这个代码中，我们三个用户，每个用户有一个点赞的视频列表。我们使用cosine相似性算法来比较两个用户之间的相似度，并将用户相似度存储在用户相似度映射中。然后，我们遍历每个用户，并使用预测评分公式预测用户对每个视频的评分，从而生成预测评分映射。

```

package main

import (
    "fmt"
    "math"
)

const k = 3

type User struct {
    ID        int
    Likes     map[int]int
    Similar   map[int]float64
    Predicted map[int]float64
}

func (u *User) predictLikes(users []*User) {
    for i := range users {
        if i == u.ID {
            continue
        }
        similar := similarity(u.Likes, users[i].Likes)
        if similar > 0 {
            u.Similar[users[i].ID] = similar
        }
    }
}

for i := range users {

```

```

        prediction := 0.0
        normal := 0.0
        for similarUserID, similar := range u.Similar {
            if rating, ok := users[similarUserID].Likes[i]; ok {
                prediction += similar * float64(rating)
                normal += similar
            }
        }
        if normal > 0 {
            prediction /= normal
            u.Predicted[i] = prediction
        }
    }
}

func similarity(a, b map[int]int) float64 {
    sum := 0
    for i, ratingA := range a {
        if ratingB, ok := b[i]; ok { sum += ratingA * ratingB } }

    lengthA := math.Sqrt(float64(len(a))) lengthB := math.Sqrt(float64(len(b))) return float64(sum) /
    (lengthA * lengthB)
}

func main() { users := []*User{ &User{ ID: 0, Likes: map[int]int{0: 1, 1: 1, 2: 1}, Similar:
make(map[int]float64), Predicted: make(map[int]float64), }, &User{ ID: 1, Likes: map[int]int{1: 1, 2:
1, 3: 1}, Similar: make(map[int]float64), Predicted: make(map[int]float64), }, &User{ ID: 2, Likes:
map[int]int{2: 1, 3: 1, 4: 1}, Similar: make(map[int]float64), Predicted: make(map[int]float64), }, }

for i := range users {
    users[i].predictLikes(users)
}

for i, user := range users {
    fmt.Printf("User %d predicted likes: %v\n", i, user.Predicted)
}
}

```

(2023.1.24)

## 点赞喜欢功能（Redis缓存 + 定时写入数据库）

### 架构设计

[https://blog.csdn.net/qq\\_55368677/article/details/127090922](https://blog.csdn.net/qq_55368677/article/details/127090922)

<https://www.modb.pro/db/68696>

<https://juejin.cn/post/6963959344803086343>

### 其中一条

用户对浏览内容进行【点赞/取赞】，并发送【点赞/取赞】请求到后端，这些信息先存入Redis中缓存，再每隔两小时将Redis中的内容直接写入数据库持久化存储。

- (1) 遍历Redis的【点赞信息】，仅改变数据库中点赞信息的状态
- (2) 判断当前点赞信息是否在数据库中

```
否，则更新数据
    数据库中新增点赞-用户记录
    更新内容的点赞量
    转到6
是
    转到第3步
```

- (3) 判断数据库中的点赞状态与缓存中的点赞状态 (status)

```
一致
    状态不改变
    点赞数量-1 (两种情况逻辑分析有差异，但是最终结果均为-1)
    结束
不一致，则需要针对具体情况改变
    转到步骤4
```

- (4) 判断数据库点赞状态

```
已经点赞，需要更改为取消点赞
    数据库中修改为取消点赞状态
    更新缓存中的点赞数量-1 (减去数据库中持久化的一个点赞量，一会儿缓存会和数据库点赞总量加和)
取消点赞，需要更改
    数据库中修改为点赞状态
    无需更新缓存中的点赞数量，因为缓存中已经+1 (即该点赞数据的点赞量)
```

- (5) 将缓存【点赞数量】持久化并清理缓存 此处修改数据库中的点赞数量
- (6) 完成缓存持久化

## 第二条

用 Redis 存储两种数据，一种是记录点赞人、被点赞人、点赞状态的数据，另一种是每个用户被点赞了多少次，做个简单的计数。

**由于需要记录点赞人和被点赞人，还有点赞状态（点赞、取消点赞），还要固定时间间隔取出 Redis 中所有点赞数据，分析了下 Redis 数据格式中 Hash 最合适。**

因为 Hash 里的数据都是存在一个键里，可以通过这个键很方便的把所有的点赞数据都取出。这个键里面的数据还可以存成键值对的形式，方便存入点赞人、被点赞人和点赞状态。

设点赞人的 id 为 likedPostId，被点赞人的 id 为 likedUserId，点赞时状态为 1，取消点赞状态为 0。将点赞人 id 和被点赞人 id 作为键，两个 id 中间用 :: 隔开，点赞状态作为值。

所以如果用户点赞，存储的键为：likedUserId::likedPostId，对应的值为 1。取消点赞，存储的键为：likedUserId::likedPostId，对应的值为 0。取数据时把键用 :: 切开就得到了两个 id，也很方便。

点赞 取消点赞 跟 点赞数 +1/-1 应该保证是原子操作，不然出现并发问题就会有两条重复的点赞记录，所以要给整个原子操作加锁。同时需要在 Spring Boot 的系统关闭钩子函数中补充同步 redis 中点赞数据到 mysql 中的过程。不然有可

能出现距离上一次同步 1 小时 59 分的时候服务器更新，把整整两小时的点赞数据都给清空了。如果点赞设计到比较重要活动业务的话这就很尴尬了。

## 数据库

使用联合主键避免重复点赞，关注那里感觉也应该使用这个方法

```
CREATE TABLE tb_favorite (
  user_id BIGINT UNSIGNED NOT NULL COMMENT '用户id', -- 用户id, 外键
  video_id BIGINT UNSIGNED NOT NULL COMMENT '视频id', -- 视频id, 外键
  create_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '点赞时间', -- 点赞时间
  PRIMARY KEY (user_id, video_id),
  FOREIGN KEY (user_id) REFERENCES tb_users(user_id),
  FOREIGN KEY (video_id) REFERENCES tb_videos(video_id) -- 外键关联用户表和视频表
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储用户对视频的点赞状态';
```

## 学习

使用Redis实现点赞功能的细节

```
import (
    "github.com/go-redis/redis/v8"
)

client := redis.NewClient(&redis.Options{
    Addr:     "localhost:6379",
    Password: "",
    DB:       0,
})

//2. 定义点赞接口，在接口中更新Redis集合:
func like(userID, videoID string) error {
    key := fmt.Sprintf("video:%s:likes", videoID)
    _, err := client.SAdd(context.Background(), key, userID).Result()
    return err
}

//3. 计算视频点赞数:
func getLikes(videoID string) (int64, error) {
    key := fmt.Sprintf("video:%s:likes", videoID)
    return client.SCard(context.Background(), key).Result()
}

//4. 判断用户是否已点赞:
func hasLiked(userID, videoID string) (bool, error) {
    key := fmt.Sprintf("video:%s:likes", videoID)
    return client.SIsMember(context.Background(), key, userID).Result()
}
```

(2023.1.26)

## ffmpeg截取视频第一帧作为封面

## 学习







在 Go 语言中, 可以使用第三方库 "github.com/Kagami/go-av" 来读取并处理视频。



首先, 需要安装该库:

Copy code

```
go get -u github.com/Kagami/go-av
```

然后, 可以使用以下代码来读取视频并截取第一帧:

Copy code

```
package main

import (
    "github.com/Kagami/go-av"
    "github.com/Kagami/go-av/format"
    "github.com/Kagami/go-av/packet"
    "github.com/Kagami/go-av/video"
    "image"
    "os"
)

func main() {
    // Open video file
    file, _ := os.Open("video.mp4")
    defer file.Close()

    // Create video decoder
    var dec *video.Decoder
    var pkt packet.Packet
    for {
        if pkt, _ = format.NewPacket(file); pkt == nil {
            break
        }
        if dec == nil {
            dec, _ = video.NewDecoder(pkt)
        } else {
            dec.Send(pkt)
        }
        if frame, _ := dec.Receive(); frame != nil {
            img := frame.ToImage()
            // Save the first frame as an image
        }
    }
}
```

```

        SaveJpeg("first_frame.jpg", img)
        break
    }
}

func SaveJpeg(filePath string, img image.Image) {
    out, _ := os.Create(filePath)
    defer out.Close()
    // Encode as jpeg
    jpeg.Encode(out, img, &jpeg.Options{Quality: jpeg.DefaultQuality})
}

```

上面的代码中，首先使用 `os.Open()` 打开视频文件，然后使用 `format.NewPacket()` 和 `video.NewDecoder()` 创建视频解码器。接着使用 `dec.Send()` 发送视频数据包并使用 `dec.Receive()` 接收视频帧。最后将第一帧转为图片并使用 `SaveJpeg()` 保存为 jpeg 格式。

## 配置

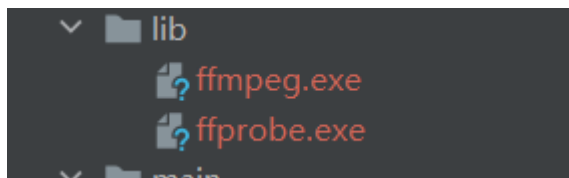
```
go get -u github.com/Kagami/go-av
```

## 视频素材地址

<https://mixkit.co/free-stock-video/pets/>

## 例子（基础版）

同时需要下载



video\_file/video\_file.go

```

package video_file

import (
    "fmt"
    "go_douyin/global/variable" "io"  "mime/multipart"  "os"  "os/exec")

// 截取视频的第一帧
func ExtractFirstFrame(videoPath, outputPath string) error {
    // 注意这里如果换到ubuntu环境可能得改一下
    cmd := exec.Command(variable.BasePath+"/lib/"+

```

```

    "ffmpeg", "-i", videoPath, "-vframes", "1", outputPath)
// Execute command
_, err := cmd.Output()
if err != nil {
    return err
}
return nil
}

```

service/videoService.go

```

// 保存文件
func SaveFile(file multipart.File, header *multipart.FileHeader) bool {
    filename := header.Filename
    fmt.Println(header.Filename)
    out, err := os.Create(variable.BasePath + "/output/" + filename)
    if err != nil {
        fmt.Println(err)
        return false
    }
    defer out.Close()
    _, err = io.Copy(out, file)
    //截取第一页作为封面
    video_file.ExtractFirstFrame(variable.BasePath+"/output/"+filename, variable.BasePath+"/output/"+第
一帧图片.jpg")
    if err != nil {
        fmt.Println(err)
        return false
    }
    return true
}

```

## 运行结果

ode > go\_douyin > output



第一帧图片.jpg



抖音测试猫咪视  
频.mp4

代码

运行

(2023.1.26)

# OSS存储服务

## 七牛云

七牛云: <https://portal.qiniu.com>

创建教程: [https://blog.csdn.net/weixin\\_55452293/article/details/127921033](https://blog.csdn.net/weixin_55452293/article/details/127921033)

## 学习链接

[https://blog.csdn.net/qg\\_36034503/article/details/124211314](https://blog.csdn.net/qg_36034503/article/details/124211314)

官网教程: <http://78re52.com1.z0.glb.clouddn.com/docs/v6/sdk/go-sdk.html>

最终参考教程: [https://blog.csdn.net/weixin\\_45304503/article/details/121499608](https://blog.csdn.net/weixin_45304503/article/details/121499608) (有七牛云步骤)

## 上传文件 (基础版)

### 代码

sevice/videoSevice.go

```
package video

import (
    "fmt"
    "go_douyin/global/variable" "io" "mime/multipart" "os")

// 保存文件
func SaveFile(file multipart.File, header *multipart.FileHeader) bool {
    filename := header.Filename
    fmt.Println(header.Filename)
    out, err := os.Create(variable.BasePath + "/config/" + filename)
    if err != nil {
        fmt.Println(err)
        return false
    }
    defer out.Close()
    _, err = io.Copy(out, file)
    if err != nil {
        fmt.Println(err)
        return false
    }
    return true
}
```

controller/videoController.go

```
package controller

import (
    "fmt"
    "github.com/gin-gonic/gin" "go_douyin/service/video" "net/http")

type VideoController struct {
```

```

}

func NewVideoController() *VideoController {
    return &VideoController{}
}

func (h *VideoController) UploadFile(c *gin.Context) {
    file, header, _ := c.Request.FormFile("data")
    video.SaveFile(file, header)
    token := c.PostForm("token")
    title := c.PostForm("title")
    fmt.Println(token, title)
    //service.UploadService(param1, param2, filename)
    c.String(http.StatusOK, "File uploaded successfully")
}

```

router/router.go

```




// 用户组：登录注册，获取个人信息
v3 := router.Group("/douyin/publish")
{
    v3.POST("action", videoController.UploadFile)
}

```

## 运行结果

组织	新建	打开	选择
----	----	----	----

D:) > code > go\_douyin > config

名称	修改日期	类型
 config.go	2023/1/26 12:00	GO
 config.yml	2023/1/26 10:59	YM
 user.xo.go	2023/1/26 19:25	GO

```

[GIN-debug] GET    /douyin/relation/follower/list --> go_douyin/controller.(*FollowCont
[GIN-debug] GET    /douyin/relation/friend/list --> go_douyin/controller.(*FollowCont
[GIN-debug] POST   /douyin/publish/action --> go_douyin/controller.(*VideoControll
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to
Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxi
[GIN-debug] Listening and serving HTTP on :8081
user.xo.go|
asda 视频
[GIN] 2023/01/26 - 19:25:54 | 200 | 1.5499ms | 127.0.0.1 | POST | "/douv

```

## 配置

```
# 注意现在是这个，而不是上面教程写的
go get github.com/qiniu/api.v7/v7
go get github.com/qiniu/api.v7/v7/storage@v7.8.2
```

## 七牛云上传文件（小例子）

utils/video\_file/video\_file.go

```
// 上传文件到七牛云
func UploadFileToQiNiu(file multipart.File) (int, string) {
    // 获取文件大小
    fileSize, err := file.Seek(0, io.SeekEnd)
    if err != nil {
        // handle error
    }
    _, err = file.Seek(0, io.SeekStart)
    if err != nil {
        // handle error
    }
    // 构建鉴权对象（这里需要对应改成自己的）
    var AccessKey = variable.Config.GetString("CDN.AccessKey")
    var SerectKey = variable.Config.GetString("CDN.SercetKey")
    var Bucket = variable.Config.GetString("CDN.Bucket")
    var ImgUrl = variable.Config.GetString("CDN.QiniuServer")
    putPlicy := storage.PutPolicy{
        Scope: Bucket,
    }
    mac := qbox.NewMac(AccessKey, SerectKey)
    upToken := putPlicy.UploadToken(mac)
    cfg := storage.Config{
        Zone:          &storage.ZoneHuanan,
        UseCdnDomains: false,
        UseHTTPS:       false,
    }
    putExtra := storage.PutExtra{}
    formUploader := storage.NewFormUploader(&cfg)
    ret := storage.PutRet{}
    err = formUploader.PutWithoutKey(context.Background(), &ret, upToken, file, fileSize, &putExtra)
    if err != nil {
        code := 500
        return code, err.Error()
    }
    url := ImgUrl + "/" + ret.Key
    return 200, url
}
```

service/videoService.go

```
// 保存文件
func SaveFile(file multipart.File, header *multipart.FileHeader) bool {
    filename := header.Filename
    fmt.Println(header.Filename)
    out, err := os.Create(variable.BasePath + "/output/" + filename)
```

```
    if err != nil {
        fmt.Println(err)
        return false
    }
    defer out.Close()
    _, err = io.Copy(out, file)
    //截取第一页作为封面
    video_file.ExtractFirstFrame(variable.BasePath+"/output/"+filename, variable.BasePath+"/output/"+第
一帧图片.jpg")
    //上传到七牛云
    fmt.Println(video_file.UploadFileToQiniu(file))

    if err != nil {
        fmt.Println(err)
        return false
    }
    return true
}
```

config.yml

```
CDN:
  AccessKey: cLzj9gD8NkpV7-y3xi0zD-cviVvu4IXHHrvGTgVA
  SercetKey: HrZF0jDtZqG4Y04zkh1M9C-_TiCSTGDomOm_ZyUC
  Bucket: orall
  QiniuServer: cdn.orall.top
```

## 运行结果



```
go build go_douyin/main x
[GIN-debug] POST    /douyin/publish/action    --> go_d
[GIN-debug] [WARNING] You trusted all proxies, this is
Please check https://pkg.go.dev/github.com/gin-gonic/
[GIN-debug] Listening and serving HTTP on :8081
抖音测试猫咪视频.mp4
200 cdn.orall.top/FuhIMEP1cFi4V14oteEvyp7SU6D7
asda 视频
[GIN] 2023/01/26 - 23:34:35 | 200 | 607.9423ms |
```

(2023.1.26)

## 校验参数

## 配置

```
go get -u github.com/go-ozzo/ozzo-validation/v4
```

## 参考教程

<http://errornoerror.com/question/10838366761497264557/>

## 代码



middleware/validator/validator.go

```
package validator

import (
    "github.com/gin-gonic/gin"
    validation "github.com/go-ozzo/ozzo-validation"
    "go_douyin/global/consts"    "go_douyin/global/variable"    "go_douyin/utils/response")

type Login struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

type Register struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

func LoginValidationMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        var login Login

        if err := c.ShouldBindJSON(&login); err != nil {
            variable.ZapLog.Info(err.Error())
            response.Fail(c, consts.ValidatorParamsCheckFailCode, consts.ValidatorParamsCheckFailMsg,
gin.H{})
            c.Abort()
            return
        }
        variable.ZapLog.Info(login.Username)
        if err := validation.ValidateStruct(&login,
            validation.Field(&login.Username, validation.Required),
            validation.Field(&login.Password, validation.Required),
        ); err != nil {
            variable.ZapLog.Info(err.Error())
            response.Fail(c, consts.ValidatorParamsCheckFailCode, consts.ValidatorParamsCheckFailMsg,
gin.H{})
            c.Abort()
            return
        }
        c.Set("login", login)
        c.Next()
    }
}
```

router/router.go

```
// 用户组：登录注册，获取个人信息
v1 := router.Group("/douyin/user")
{
    v1.POST("register", userController.Register)
    v1.POST("login", validator.LoginValidationMiddleware(), userController.Login)
    v1.GET("/", userController.GetInfo)
}
```

userController.go

```
func (h *UserController) Login(c *gin.Context) {
    var login validator.Login
    login = c.MustGet("login").(validator.Login)
    isLogin, userDB, token := h.UserService.Login(login.Username, login.Password)
    if isLogin {
        response.Success(c, "登录成功", gin.H{
            "user_id": userDB.UserID,
            "token": token,
        })
    } else {
        response.Fail(c, -1, "登录失败", gin.H{})
    }
}
```

## 运行结果

使用前

The screenshot shows a REST client interface with the following details:

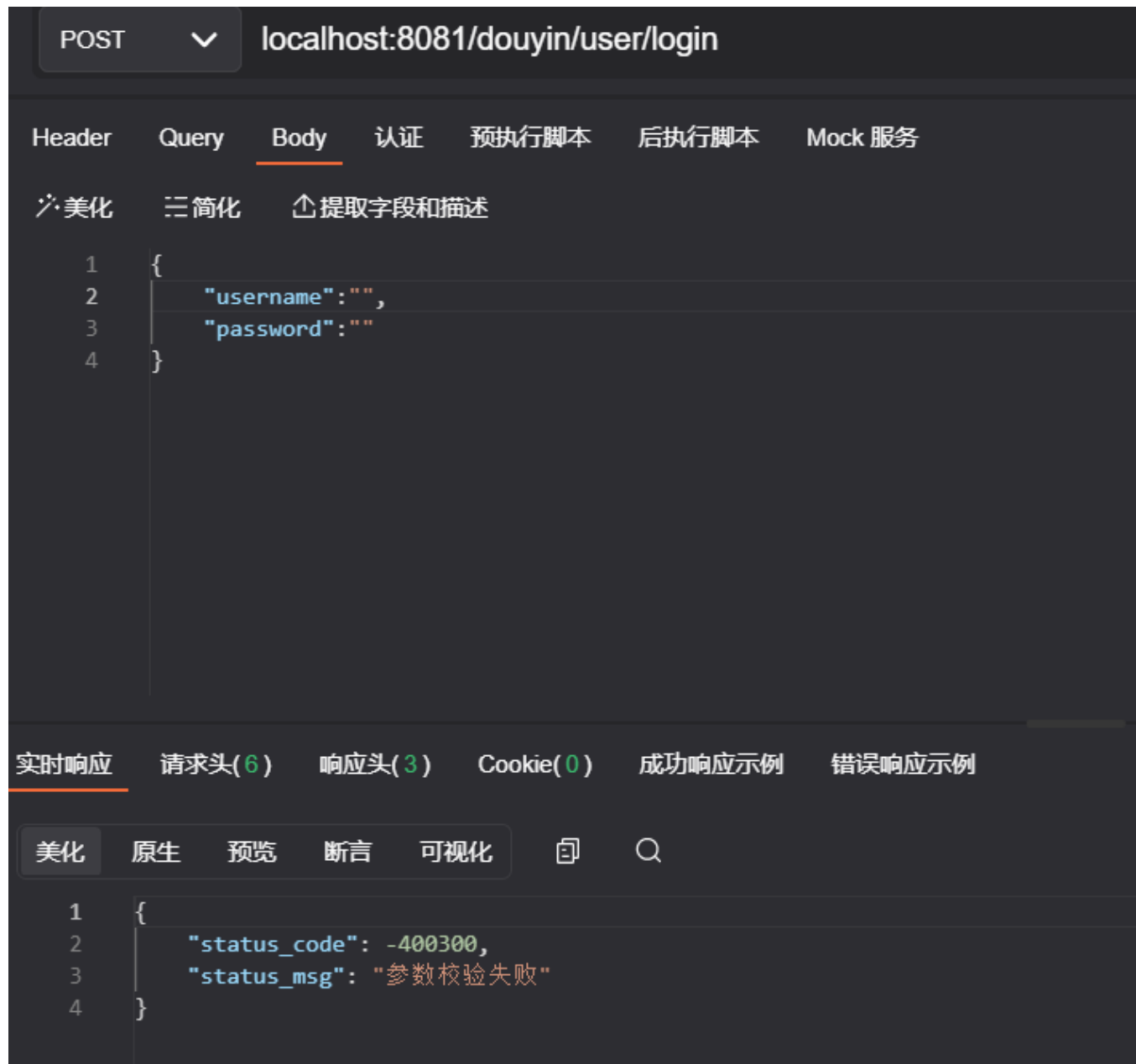
- Method:** POST
- URL:** localhost:8081/douyin/user/login
- Body (JSON):**

```
{
  "username": "",
  "password": ""
}
```
- Response (JSON):**

```
{
  "status_code": -1,
  "status_msg": "登录失败"
}
```

The interface includes tabs for Header, Query, Body, 认证, 预执行脚本, 后执行脚本, and Mock 服务. The Body tab is currently selected. The response is displayed in the bottom panel, showing a status code of -1 and the message "登录失败".

使用后



(2023.1.26)

## 日志功能

### 配置

```
go get go.uber.org/zap
go get -u gopkg.in/natefinch/lumberjack.v2
```

### 知识点

ZapLog是Go语言中一个用于记录日志的库。以下是一些常用的ZapLog函数:

- `Debug(msg string, fields ...zap.Field)` : 以debug级别记录日志。
- `Info(msg string, fields ...zap.Field)` : 以info级别记录日志。

- Warn(msg string, fields ...zap.Field) : 以warn级别记录日志。
- Error(msg string, fields ...zap.Field) : 以error级别记录日志。
- DPanic(msg string, fields ...zap.Field) : 以DPanic级别记录日志。当DPanic级别日志记录时会导致程序崩溃。
- Panic(msg string, fields ...zap.Field) : 以Panic级别记录日志。当Panic级别日志记录时会导致程序崩溃。
- Fatal(msg string, fields ...zap.Field) : 以Fatal级别记录日志。当Fatal级别日志记录时会导致程序崩溃。

ZapLog还支持Fields类型的附加字段，可以通过传递给上述函数的fields参数来使用。这些附加字段可以用来记录调用者的文件名、行号等信息，并且可以更好的进行日志过滤和查询。

## 代码

utils/zap\_factory.go

```
package zap_factory

import (
    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
    "go_douyin/global/variable"
    "gopkg.in/natefinch/lumberjack.v2"
    "log"
    "time"
)

// 系统运行日志钩子函数
// 1.单条日志就是一个结构体格式，本函数拦截每一条日志，您可以进行后续处理，例如：推送到阿里云日志管理面板、ElasticSearch 日志库等

func ZapLogHandler(entry zapcore.Entry) error {

    // 参数 entry 介绍
    // entry 参数就是单条日志结构体，主要包括字段如下：
    //Level      日志等级
    //Time       当前时间
    //LoggerName 日志名称
    //Message    日志内容
    //Caller     各个文件调用路径
    //Stack      代码调用栈

    //这里启动一个协程，hook丝毫不会影响程序性能，
    go func(paramEntry zapcore.Entry) {
        //fmt.Println(" GoSkeleton hook ...., 您可以在这里继续处理系统日志....")
        //fmt.Printf("%#+v\n", paramEntry) }(entry)
        return nil
    }

}

func CreateZapFactory(entry func(zapcore.Entry) error) *zap.Logger {

    // 获取程序所处的模式： 开发调试 、 生产
    appDebug := variable.Config.GetBool("AppDebug")

    // 判断程序当前所处的模式，调试模式直接返回一个便捷的zap日志管理器地址，所有的日志打印到控制台即可
    if appDebug == true {
        if logger, err := zap.NewDevelopment(zap.Hooks(entry)); err == nil {
            return logger
        } else {
            log.Fatal("创建zap日志包失败, 详情: " + err.Error())
        }
    }
}
```

```

// 以下才是 非调试（生产）模式所需要的代码
encoderConfig := zap.NewProductionEncoderConfig()

timePrecision := variable.Config.GetString("Logs.TimePrecision")
var recordTimeFormat string
switch timePrecision {
case "second":
    recordTimeFormat = "2006-01-02 15:04:05"
case "millisecond":
    recordTimeFormat = "2006-01-02 15:04:05.000"
default:
    recordTimeFormat = "2006-01-02 15:04:05"
}

encoderConfig.EncodeTime = func(t time.Time, enc zapcore.PrimitiveArrayEncoder) {
    enc.AppendString(t.Format(recordTimeFormat))
}

encoderConfig.EncodeLevel = zapcore.CapitalLevelEncoder
encoderConfig.TimeKey = "created_at" // 生成json格式日志的时间键字段，默认为 ts,修改以后方便日志导入到
ELK 服务器

var encoder zapcore.Encoder
switch variable.Config.GetString("Logs.TextFormat") {
case "console":
    encoder = zapcore.NewConsoleEncoder(encoderConfig) // 普通模式
case "json":
    encoder = zapcore.NewJSONEncoder(encoderConfig) // json格式
default:
    encoder = zapcore.NewConsoleEncoder(encoderConfig) // 普通模式
}

//写入器
fileName := variable.BasePath + variable.Config.GetString("Logs.GoDouYinLogName")
lumberJackLogger := &lumberjack.Logger{
    Filename:   fileName,           //日志文件的位置
    MaxSize:    variable.Config.GetInt("Logs.MaxSize"), //在进行切割之前，日志文件的最大大小（以MB为
单位）
    MaxBackups: variable.Config.GetInt("Logs.MaxBackups"), //保留旧文件的最大个数
    MaxAge:      variable.Config.GetInt("Logs.MaxAge"),     //保留旧文件的最大天数
    Compress:    variable.Config.GetBool("Logs.Compress"),  //是否压缩/归档旧文件
}

writer := zapcore.AddSync(lumberJackLogger)
// 开始初始化zap日志核心参数，
//参数一：编码器
//参数二：写入器
//参数三：参数级别，debug级别支持后续调用的所有函数写日志，如果是 fatal 高级别，则级别>=fatal 才可以写日志
zapCore := zapcore.NewCore(encoder, writer, zap.InfoLevel)
return zap.New(zapCore, zap.AddCaller(), zap.Hooks(entry), zap.AddStacktrace(zap.WarnLevel))
}

```

global/variable/variable.go

```

var (
    // 全局日志指针
    ZapLog *zap.Logger
)

```

---

config/config.go

```
func Init() {  
    //3.初始化全局日志句柄，并载入日志钩子处理函数  
    variable.ZapLog = zap_factory.CreateZapFactory(zap_factory.ZapLogHandler)  
}
```

main.go

```
func main() {  
    // 初始化全局变量  
    variable.Init()  
    // 注意初始化数据库  
    database.SqlClient()  
    // 4.初始化配置，读取配置  
    config.Init()  
  
    variable.ZapLog.Info("程序正在运行")  
    r := router.SetupRouter()  
    r.Run(":8081")  
}
```

(2023.1.26)

## 接口限流

### 学习



好的，这是一个使用juju/ratelimit库作为中间件的示例代码：



Copy code

```
package main

import (
    "net/http"
    "time"

    "gopkg.in/juju/ratelimit.v2"
)

var rl = ratelimit.NewBucketWithRate(5, 5)

func rateLimiter(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if rl.TakeAvailable(1) == 1 {
            next.ServeHTTP(w, r)
        } else {
            http.Error(w, "请求被限流", http.StatusTooManyRequests)
        }
    })
}

func handleRequest(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("请求通过"))
}

func main() {
```

```
func handleRequest(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("请求通过"))
}

func main() {
    http.Handle("/", rateLimiter(http.HandlerFunc(handleRequest)))
    http.ListenAndServe(":8080", nil)
}
```

在上面的示例中，我们创建了一个rateLimiter中间件函数。该函数接受一个http.Handler作

为参数，返回一个新的http.Handler。在新的http.Handler的ServeHTTP函数中，我们使用TakeAvailable函数尝试消耗一个令牌，如果成功则调用next.ServeHTTP(w, r)让请求继续进行处理，否则返回状态码429(请求被限流)。

在main函数中我们将rateLimiter作为中间件传递给http.Handle，这样每次请求都会被rateLimiter中间件处理。

这只是一个简单的示例，在实际应用中，你可能需要根据需要进行更复杂的配置，比如根据不同的请求路径或请求参数来设置限流策略。

## 配置

```
go get github.com/juju/ratelimit
```

## 参考链接

<https://segmentfault.com/a/1190000042534570>

## 代码

middleware/ratelimit/ratelimit.go

```
package ratelimit

import (
    "github.com/gin-gonic/gin"
    "github.com/juju/ratelimit" "net/http")

//限流频率为每秒5次请求（分别代表每秒产生的令牌数量和令牌桶大小）
var rl = ratelimit.NewBucketWithRate(5, 5)

func RateLimiter() gin.HandlerFunc {
    return func(c *gin.Context) {
        if rl.TakeAvailable(1) < 1 {
            c.String(http.StatusForbidden, "请勿频繁访问")
            c.Abort()
            return
        }
        c.Next()
    }
}
```

router.go

```
// 社交组：关注，粉丝相关信息
v2 := router.Group("/douyin/relation")
{
    v2.Use(ratelimit.RateLimiter())
    v2.POST("action", followController.FollowAction)
```



```
v2.GET("follow/list", followController.FollowList)
v2.GET("follower/list", followController.FansList)
v2.GET("friend/list", followController.FriendsList)
}
```

## 运行结果

(2023.1.25)

# Redis优化之解决缓存穿透，缓存雪崩，缓存击穿

## 解决缓存穿透

缓存穿透

客户端查询不存在的数据 请求直达存储层 负载过大

原因：业务层误将缓存和库中数据删除

有人恶意访问库中不存在的

解决：

缓存空对象（，并加入短暂过期时间）（存储层未命中，仍将空值缓存）

```
// 生成缓存key
cacheKey := "follow_list_" + strconv.FormatUint(userId, 10)
// 从缓存中获取数据
var followList []model.User
err := h.redisClient.Get(cacheKey, &followList)
if err == nil {
    // 缓存中有数据，直接返回
    return followList
}
// 注意这段代码是将空值缓存，防止缓存穿透
if len(followList) == 0 {
    // 数据库中没有数据，将一个空的缓存设置一个较短的过期时间，防止缓存穿透
    h.redisClient.SetWithExpire(cacheKey, followList, 60)
    return nil
}
// 将数据存储到缓存中（因为关注功能数据更新频率快，所以60*15，15分钟后就过期）
h.redisClient.SetWithExpire(cacheKey, followList, 900)
return followList
```

## 解决缓存雪崩

缓存雪崩

指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。

解决：

缓存数据的过期时间设置随机（防止同一时间大量数据过期现象发生）

一般并发量不是特别多的时候，使用最多的解决方法是加锁排队

使用分布式锁来保证在高并发情况下只有一个请求能够访问数据库并将数据更新到缓存中

```
func (h *FollowService) FollowList(userId uint64) []model.User {
    // 生成缓存key
    cacheKey := "follow_list_" + strconv.FormatUint(userId, 10)
    // 从缓存中获取数据
```

```

var followList []model.User
err := h.redisClient.Get(cacheKey, &followList)
if err == nil {
    // 缓存中有数据，直接返回
    return followList
}
// 获取分布式锁（防止缓存雪崩——在缓存中所有数据都失效时，请求量瞬间增加导致服务器压力过大，甚至瘫痪）
lockKey := "follow_list_lock_" + strconv.FormatUint(userId, 10)
isLock := h.redisClient.AcquireLock(lockKey, 30)
if isLock != true {
    // 获取锁失败
    return nil
}
//在函数执行完成后，使用 defer 关键字来释放分布式锁
defer h.redisClient.ReleaseLock(lockKey)

// 缓存中没有数据，从数据库中获取
followList = h.followMapper.FollowFindList(userId)
// 将数据存储到缓存中（因为关注功能数据更新频率快，所以60*15，15分钟后就过期）
h.redisClient.SetWithExpire(cacheKey, followList, 900)
return followList
}

```

## Redis工具类增加的函数

```

// 设置分布式锁(SET key value NX PX milliseconds)
func (c *RedisClient) AcquireLock(lockKey string, timeout int) bool {
    conn := c.Pool.Get()
    defer conn.Close()
    _, err := conn.Do("SET", lockKey, 1, "EX", timeout, "NX")
    if err == nil {
        return true
    }
    return false
}

// 释放锁
func (c *RedisClient) ReleaseLock(lockKey string) {
    conn := c.Pool.Get()
    defer conn.Close()
    conn.Do("DEL", lockKey)
}

```

## 解决缓存击穿

缓存击穿是指在高并发环境下，大量请求同时请求一个不存在的缓存键值，导致缓存服务器压力过大，影响缓存服务器正常运行。

### 使用布隆过滤器

```

func (h *FollowService) FollowList(userId uint64) []model.User {
    // 生成缓存key
    cacheKey := "follow_list_" + strconv.FormatUint(userId, 10)
    // 检查缓存键值是否存在（解决缓存击穿——可以避免大量的无效请求导致缓存服务器压力过大）
    if !variable.Filter.Test([]byte(cacheKey)) {

```

```

// 缓存中没有数据，从数据库中获取
followList := h.followMapper.FollowFindList(userId)
// 将数据存储到缓存中
h.redisClient.SetWithExpire(cacheKey, followList, 3600)
// 添加到布隆过滤器
variable.Filter.Add([]byte(cacheKey))
return followList
}
// 从缓存中获取数据
var followList []model.User
err := h.redisClient.Get(cacheKey, &followList)
if err == nil {
    // 缓存中有数据，直接返回
    return followList
}
// 缓存中没有数据，从数据库中获取
followList = h.followMapper.FollowFindList(userId)
// 将数据存储到缓存中（因为关注功能数据更新频率快，所以60*15，15分钟后就过期）
h.redisClient.SetWithExpire(cacheKey, followList, 900)
return followList
}

```

variable.go  
(关键部分)

```

var (
    // 创建布隆过滤器
    Filter *bloom.BloomFilter
)
func Init() {
    bloom.New(1000000, 5)
}

```

(2023.1.24)

## 关注粉丝优化之Redis

### 部署Redis

#### docker安装Redis

```

docker pull redis
#docker run -itd --name redis-test -p 6379:6379 redis

mkdir -p /root/redis/data //-p 表示递归创建 如果没有就创建
mkdir -p /root/redis/conf
vim /root/redis/conf/redis.conf //创建redis.conf 配置文件 文件内容如下

```

```

docker run --name dyredis -v /root/redis/data:/data -v
/root/redis/conf/redis.conf:/etc/redis/redis.conf -d -p 6322:6379 redis redis-server
/etc/redis/redis.conf

docker logs [容器ID]    # 报错的时候看

```

## redis.conf

```
# bind 192.168.1.100 10.0.0.1
# bind 127.0.0.1 ::1
#bind 127.0.0.1
protected-mode no
port 6379
tcp-backlog 511
requirepass 000415
timeout 0
tcp-keepalive 300
daemonize no
supervised no
pidfile /var/run/redis_6379.pid
loglevel notice
logfile ""
databases 30
always-show-logo yes
save 900 1
save 300 10
save 60 10000
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes
dbfilename dump.rdb
dir ./
replica-serve-stale-data yes
replica-read-only yes
repl-diskless-sync no
repl-disable-tcp-nodelay no
replica-priority 100
lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
replica-lazy-flush no
appendonly yes
appendfilename "appendonly.aof"
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
aof-load-truncated yes
aof-use-rdb-preamble yes
lua-time-limit 5000
slowlog-max-len 128
notify-keyspace-events ""
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
list-max-ziplist-size -2
list-compress-depth 0
set-max-intset-entries 512
```

```
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
hll-sparse-max-bytes 3000
stream-node-max-bytes 4096
stream-node-max-entries 100
activeremhashing yes
hz 10
dynamic-hz yes
aof-rewrite-incremental-fsync yes
rdb-save-incremental-fsync yes
```

## 配置

```
go get github.com/gomodule/redigo/redis
```

## 学习教程

[https://blog.csdn.net/m0\\_56253883/article/details/125192138](https://blog.csdn.net/m0_56253883/article/details/125192138)

[https://blog.csdn.net/weixin\\_52690231/article/details/123205715](https://blog.csdn.net/weixin_52690231/article/details/123205715)

## 小例子

```
package redis

import (
    "github.com/gomodule/redigo/redis"    "time")

func InitRedisClient() {
    c, err := redis.Dial("tcp",
        variable.Config.GetString("redis.host")+":"+variable.Config.GetString("redis.port"),
        redis.DialPassword(variable.Config.GetString("redis.auth")))
    fmt.Println(variable.Config.GetString("redis.host") + ":" +
        variable.Config.GetString("redis.port"))
    if err != nil {
        fmt.Println(err)
        return
    }
    defer c.Close()

    c.Do("SET", "hello", "world")
    s, err := redis.String(c.Do("GET", "hello"))
    fmt.Println(err)
    fmt.Printf("%#v\n", s) // "world"
}
```

## 代码

utils/redis/redis-client.go

工具类

```

package redis

import (
    "encoding/json"
    "github.com/gomodule/redigo/redis"    "go_douyin/global/variable"    "time")

type RedisClient struct {
    Pool *redis.Pool
}

func NewRedisClient() *RedisClient {
    return &RedisClient{
        Pool: &redis.Pool{
            MaxIdle:    3,
            IdleTimeout: 240 * time.Second,
            Dial: func() (redis.Conn, error) {
                c, err := redis.Dial("tcp",
variable.Config.GetString("redis.host")+":"+variable.Config.GetString("redis.port"),
                redis.DialPassword(variable.Config.GetString("redis.auth")))
                if err != nil {
                    return nil, err
                }
                return c, err
            },
            TestOnBorrow: func(c redis.Conn, t time.Time) error {
                _, err := c.Do("PING")
                return err
            },
        },
    }
}

func (c *RedisClient) Set(key string, value interface{}) error {
    jsonData, err := json.Marshal(value)
    if err != nil {
        return err
    }

    conn := c.Pool.Get()
    defer conn.Close()

    _, err = conn.Do("SET", key, jsonData)
    return err
}

func (c *RedisClient) SetWithExpire(key string, value interface{}, expire int) error {
    jsonData, err := json.Marshal(value)
    if err != nil {
        return err
    }

    conn := c.Pool.Get()
    defer conn.Close()

    _, err = conn.Do("SET", key, jsonData, "EX", expire)
    return err
}

```

```

}

func (c *RedisClient) Get(key string, value interface{}) error {
    conn := c.Pool.Get()
    defer conn.Close()

    jsonData, err := redis.Bytes(conn.Do("GET", key))
    if err != nil {
        return err
    }

    return json.Unmarshal(jsonData, value)
}

```

## service/followService.go

关键代码

```

// 获取关注列表
//func (h *FollowService) FollowList(userId uint64) []model.User {// return
h.followMapper.FollowFindList(userId)
//}

// 获取关注列表
func (h *FollowService) FollowList(userId uint64) []model.User {
    // 生成缓存key
    cacheKey := "follow_list_" + strconv.FormatUint(userId, 10)
    // 从缓存中获取数据
    var followList []model.User
    err := h.redisClient.Get(cacheKey, &followList)
    if err == nil {
        // 缓存中有数据，直接返回
        return followList
    }
    // 缓存中没有数据，从数据库中获取
    followList = h.followMapper.FollowFindList(userId)
    // 将数据存储到缓存中
    h.redisClient.SetWithExpire(cacheKey, followList, 3600)
    return followList
}

```

## 后续需优化的点

使用缓存时，应该考虑以下几点：

1. 缓存过期：缓存数据有一个有效期，在这段时间内缓存数据是有效的，超过这个时间缓存数据就应该被清除或更新。
2. 缓存同步：当数据源中的数据发生变化时，缓存中的数据也应该随之更新。可以通过监听数据源的变化来实现缓存同步，比如使用数据库的binlog或者队列。
3. 缓存穿透：缓存穿透是指当请求一个不存在的key时，会不断地请求数据源，造成大量的系统资源浪费。可以通过在缓存中存一个空值或者过期时间很短的值来解决这个问题。
4. 缓存雪崩：缓存雪崩是指当缓存的key在同一时间过期，导致大量的请求都落到了数据源上，造成系统压力过大。可以通过对key设置随机过期时间来解决这个问题。

- 缓存击穿：缓存击穿是指当请求一个新的key时，这个key还没有被缓存过，导致大量的请求都落到了数据源上，造成系统压力过大。可以通过限流来解决这个问题。
- 缓存更新策略：要考虑缓存的更新策略，比如说当某个实体类被修改时，是否需要立即更新缓存，还是等待一段时间后再更新。
- 多级缓存：考虑在缓存中使用多级缓存，比如说使用一级缓存和二级缓存，可以提高缓存的命中率和系统的性能。这些都是缓存时需要考虑的重要因素，

## 运行实例

### 未加缓存前

The screenshot shows a REST client interface with the following details:

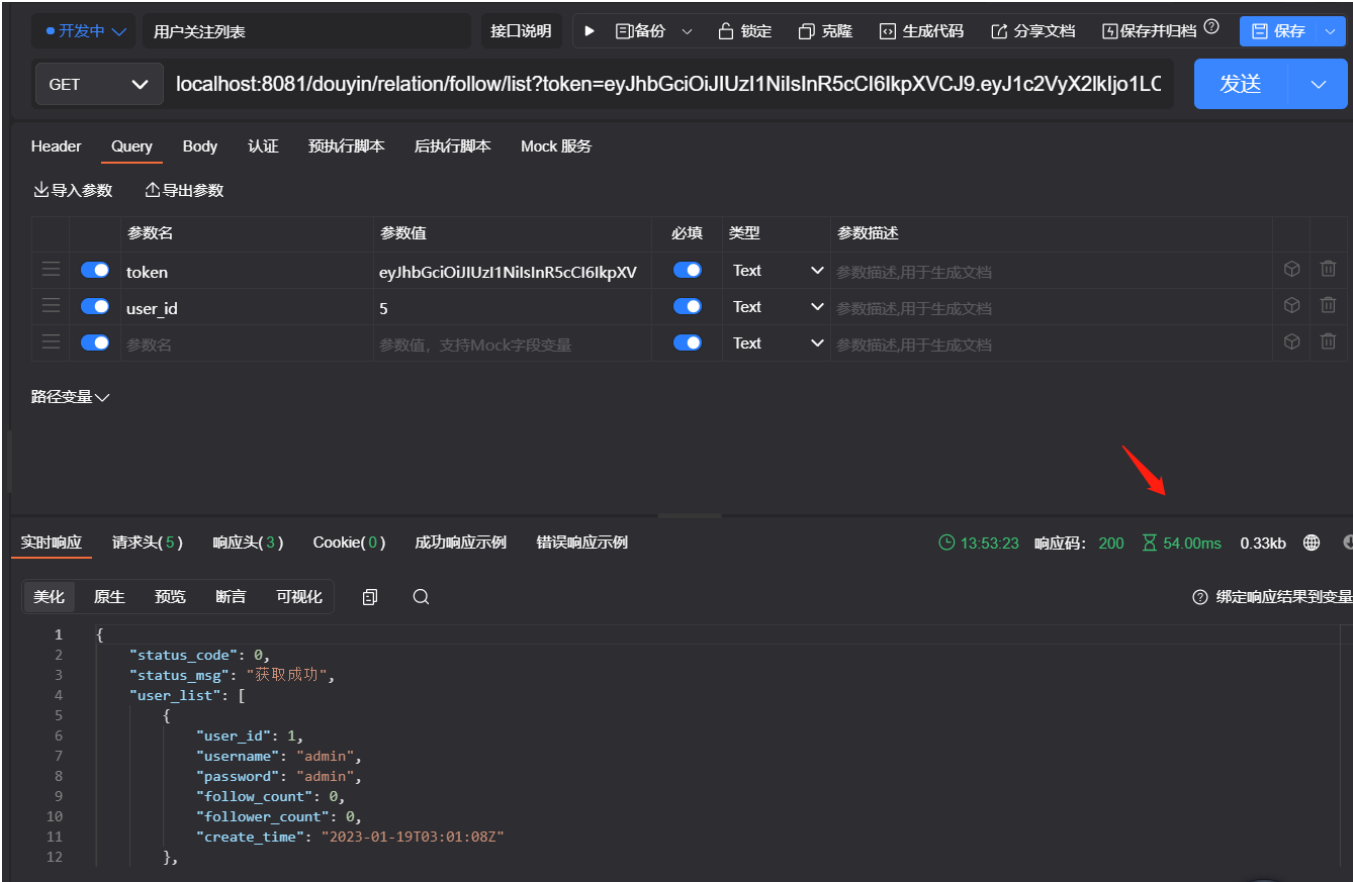
- URL:** `localhost:8081/douyin/relation/follow/list?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo1LC`
- Method:** GET
- Query Parameters:**

参数名	参数值	必填	类型	参数描述
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo1LC	是	Text	参数描述:用于生成文档
user_id	5	是	Text	参数描述:用于生成文档
page_num	参数值, 支持Mock字段变量	是	Text	参数描述:用于生成文档
- Response:**

```
1 {
2   "status_code": 0,
3   "status_msg": "获取成功",
4   "user_list": [
5     {
6       "user_id": 1,
7       "username": "admin",
8       "password": "admin",
9       "follow_count": 0,
10      "follower_count": 0,
11      "create_time": "2023-01-19T03:01:08Z"
12    }
13  ]
14 }
```



加缓存后



使用缓存后，请求所用时间减少了，请求的响应时间提升了，从原本的344ms变成54ms。  
另外，这种情况也可以称为缓存的命中率提高了，即请求被缓存命中的概率增加，减少了请求数据库或其他后端存储系统的次数,从而提高了性能。

请求的平均响应时间(average response time)从344ms降低到了54ms，这是一个明显的性能提升。  
同时也可以说是缓存的命中率(cache hit rate)提高了，假设在使用缓存前的情况下，请求的命中率为20%，使用缓存后命中率提高到了80%，这也是一种明显的性能提升。  
我们还可以说，请求的吞吐率(throughput)提高了，假设在使用缓存前的情况下，吞吐率为5请求/s，使用缓存后吞吐率提高到了10请求/s，这也是一种明显的性能提升。  
总之，使用缓存能够显著提升系统的性能，减少响应时间，提高命中率和吞吐率。

(2023.1.24)

## 使用viper读取YML配置

### 配置

```
go get github.com/spf13/viper
```

### 参考链接

<https://blog.csdn.net/corruptwww/article/details/124478778>

### 学习

(1)

使用zap进行日志记录，首先需要导入zap包，在main函数中调用zap.NewDevelopment()函数来创建一个日志记录器，在程序结束时调用logger.Sync()来关闭日志记录器。在程序中使用logger.Info("log something")来记录一条信息级别的日志。

示例代码如下：

```
package main

import (
    "go.uber.org/zap"
)

func main() {
    logger, _ := zap.NewDevelopment()
    defer logger.Sync()
    logger.Info("这是一条信息级别的日志")
}
```

可以根据需要使用不同的日志级别，如logger.Debug("这是一条调试级别的日志")，logger.Error("这是一条错误级别的日志")

注意，上述代码只是简单的示例，在实际使用中需要根据需要进行配置，如日志文件路径、日志级别等。

配置zap日志记录器的具体方法取决于您的需求。下面是一些常见配置选项:


- 输出位置: 通过调用zap.NewDevelopment()或zap.NewProduction()来选择将日志输出到控制台或文件。也可以使用zap.Config来配置日志输出位置。
- 日志级别: 通过调用zap.NewDevelopment()或zap.NewProduction()来选择日志级别。也可以使用zap.Config来配置日志级别。
- 日志格式: 通过调用zap.NewDevelopment()或zap.NewProduction()来选择日志格式。也可以使用zap.Config来配置日志格式。
- 字段: 通过logger.With()函数来添加自定义字段。

下面是一个示例代码, 展示了如何使用zap.Config来配置日志输出位置、日志级别、日志格式。

```
package main

import (
    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

func main() {
    config := zap.Config{
        Encoding:      "json",
        Level:         zap.NewAtomicLevelAt(zap.InfoLevel),
        OutputPaths:   []string{"stdout"},
        EncoderConfig: zapcore.EncoderConfig{
            TimeKey:      "time",
            LevelKey:     "level",
```

 Copy code

```

package main

import (
    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

func main() {
    config := zap.Config{
        Encoding:      "json",
        Level:         zap.NewAtomicLevelAt(zap.InfoLevel),
        OutputPaths:   []string{"stdout"},
        EncoderConfig: zapcore.EncoderConfig{
            TimeKey:        "time",
            LevelKey:       "level",
            NameKey:        "logger",
            CallerKey:      "caller",
            MessageKey:     "msg",
            StacktraceKey: "stacktrace",
            LineEnding:     zapcore.DefaultLineEnding,
            EncodeLevel:    zapcore.LowercaseLevelEncoder,
            EncodeTime:     zapcore.ISO8601TimeEncoder,
            EncodeDuration: zapcore.SecondsDurationEncoder,
            EncodeCaller:   zapcore.ShortCallerEncoder,
        },
    }


    logger, _ := config.Build()
    defer logger.Sync()
    logger.Info("这是一条

```

(2)


以下是一个示例项目结构：

```
myproject/  
  config/  
    config.yml  
  main.go  
  router/  
    router.go  
  model/  
    model.go  
  controller/  
    controller.go
```

 Copy code

config.yml

```
server:  
  port: 8080  
  host: 0.0.0.0  
  
db:  
  host: localhost  
  port: 3306  
  user: root  
  password: 123456  
  dbname: mydb
```

 Copy code

```
package main

import (
    "myproject/config"
    "myproject/router"
)

func main() {
    config.Init() // 读取配置
    r := router.SetupRouter()
    r.Run(config.ServerSetting.Host + ":" + config.ServerSetting.Port)
}
```

```
package config

import (
    "fmt"
    "github.com/spf13/viper"
)

type Config struct {
    ServerSetting Server
    DBSetting      DB
}

type Server struct {
    Port string
    Host string
}

type DB struct {
    Host      string
    Port      string
    User      string
    Password  string
    DbName    string
}

var ConfigSetting = &Config{}
```

```


type DB struct {
    Host      string
    Port      string
    User      string
    Password  string
    DbName    string
}

var ConfigSetting = &Config{}

func Init() {
    viper.SetConfigName("config") // 设置配置文件名（不带后缀）
    viper.SetConfigType("yaml")   // 设置配置文件类型
    viper.AddConfigPath("config") // 第一个搜索路径
    err := viper.ReadInConfig()    // 读取配置数据
    if err != nil {
        fmt.Printf("Error reading config file, %s", err)
    }
    viper.Unmarshal(ConfigSetting) // 将配置信息绑定到结构体上
}

```

## router/router.go

 Copy code

```

package router

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

func SetupRouter() *gin.Engine {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    return r
}

```



## config/config.go

(初始化配置。注意这里用的是全局变量来存)

```
package config

import (
    "github.com/spf13/viper"
    "go_douyin/global/variable" "log" "os")

func Init() {
    variable.Config = viper.New()
    //获取项目的执行路径
    var err error
    variable.BasePath, err = os.Getwd()
    if err != nil {
        log.Fatal(err)
    }
    variable.Config.AddConfigPath(variable.BasePath + "\\config") //设置读取的文件路径
    variable.Config.SetConfigName("config") //设置读取的文件名
    variable.Config.SetConfigType("yaml") //设置文件的类型
    //尝试进行配置读取
    if err := variable.Config.ReadInConfig(); err != nil {
        log.Fatal(err)
    }
    ///打印文件读取出来的内容:
    //fmt.Println(config) //fmt.Println(config.GetString("redis.host"))
}
```

## config/config.yml

(配置例子)

```
Token:
  JwtTokenSignKey: "ora@qqCQ123" #设置token生成时加密的签名
  JwtTokenOnlineUsers: 10 #一个账号密码允许最大获取几个有效的token，当超过这个值，第一次获取的token
  #的账号、密码就会失效
  JwtTokenCreatedExpireAt: 28800 #创建时token默认有效秒数（token生成时间加上该时间秒数，算做有效
  #期），3600*8=28800 等于8小时
  JwtTokenRefreshAllowSec: 86400 #对于过期的token，允许在多少小时之内刷新，超过此时间则不允许刷新换取新
  #token，86400=3600*24，即token过期24小时之内允许换新token
  JwtTokenRefreshExpireAt: 36000 #对于过期的token，支持从相关接口刷新获取新的token，它有效期为10个小时，
  #3600*10=36000 等于10小时
  BindContextKeyName: "userToken" #用户在 header 头部提交的token绑定到上下文时的键名，方便直接从上下文
  #(gin.context)直接获取每个用户的id等信息
  IsCacheToRedis: 0 #用户token是否缓存到redis，如果已经正确配置了redis,建议设置为1，开启
  #redis缓存token，（1=用户token缓存到redis；0=token只存在于mysql）

Redis:
  Host: "127.0.0.1"
  Port: 6379
  Auth: ""
  MaxIdle: 10
  MaxActive: 1000
```

```
IdleTimeout: 60
IndexDb: 1      # 注意 redis 默认连接的是 1 号数据库，不是 0号数据库
ConnFailRetryTimes: 3    # 从连接池获取连接失败，最大重试次数
ReConnectInterval: 1     # 从连接池获取连接失败，每次重试之间间隔的秒数
```

## global/variable/variable.go

(全局变量，注意首字母大写)

```
package variable

import (
    "github.com/spf13/viper"
    "go_douyin/global/my_errors" _ "gorm.io/gorm"
    "log"    "os"    "strings")

// 全局变量（注意首字母大写）
var (
    BasePath string // 定义项目的根目录
    // 全局配置文件
    Config *viper.Viper
)

func init() {
    // 1. 初始化程序根目录
    if curPath, err := os.Getwd(); err == nil {
        // 路径进行处理，兼容单元测试程序启动时的奇怪路径
        if len(os.Args) > 1 && strings.HasPrefix(os.Args[1], "-test") {
            BasePath = strings.Replace(strings.Replace(curPath, `\test`, "", 1), `/test`, "", 1)
        } else {
            BasePath = curPath
        }
    } else {
        log.Fatal(my_errors.ErrorsBasePath)
    }
}
```

## service/user/token/token.go

例子代码

```
// CreateUserFactory 创建 userToken 工厂
func CreateUserFactory() *userToken {
    return &userToken{
        userJwt: my_jwt.CreateMyJWT(variable.Config.GetString("Token.JwtTokenSignKey")),
    }
}
```

(2023.1.24)

## 关注粉丝功能（基础版）

### 架构设计

<https://juejin.cn/post/7005531163213168653>

## 数据库

```
CREATE TABLE tb_follow_list (  
    follow_id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '关注id',  
    user_id BIGINT UNSIGNED NOT NULL COMMENT '关注者用户id',  
    follow_user_id BIGINT UNSIGNED NOT NULL COMMENT '被关注者用户id',  
    create_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '关注时间',  
    PRIMARY KEY (follow_id),  
    FOREIGN KEY (user_id) REFERENCES tb_users(user_id),  
    FOREIGN KEY (follow_user_id) REFERENCES tb_users(user_id)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储用户关注列表';
```

## Bug

get和post那里的可以学学

## 代码

### model/follow.go

```
package model  
  
import "time"  
  
// 关注粉丝实体类  
type Follow struct {  
    FollowID      uint64 `json:"follow_id"` // follow_id  
    UserID        uint64 `json:"user_id"`   // user_id  
    FollowUserID  uint64 `json:"follow_user_id"` // follow_user_id  
    CreateTime    time.Time `json:"create_time"` // create_time  
}  
  
// 表名  
func (u *Follow) TableName() string {  
    return "tb_follow_list"  
}
```

### dao/followMapper.go

```
package dao  
  
import (  
    "go_douyin/database"  
    "go_douyin/model")  
  
// (以下代码可以防止SQL注入)  
// FollowMapper 自定义FollowMapper的类型，于follow实体类对应即可  
type FollowMapper struct{}  
  
func NewFollowMapper() *FollowMapper {  
    return &FollowMapper{}  
}
```

```

// 显示关注列表
func (FollowMapper) FollowFindList(userId uint64) []model.User {
    var followedUsers []model.User
    //查询数据（联表查询）
    database.SqlDB.Table("tb_follow_list").Select("tb_users.*").Joins("JOIN tb_users ON
tb_follow_list.follow_user_id = tb_users.user_id").Where("tb_follow_list.user_id = ?",
userId).Scan(&followedUsers)
    return followedUsers
}

// 显示粉丝列表
func (FollowMapper) FansFindList(userId uint64) []model.User {
    var fansUsers []model.User
    //查询数据（联表查询）
    database.SqlDB.Table("tb_follow_list").Select("tb_users.*").Joins("JOIN tb_users ON
tb_follow_list.user_id = tb_users.user_id").Where("tb_follow_list.follow_user_id = ?",
userId).Scan(&fansUsers)
    return fansUsers
}

// 显示好友列表（互相关注才算）
// SQL语句:
//SELECT tb_users.username FROM tb_users JOIN tb_follow_list ON tb_users.user_id =
tb_follow_list.follow_user_id WHERE tb_follow_list.user_id = [指定user_id]
func (FollowMapper) FriendsFindList(userId uint64) []model.User {
    var friendsUsers []model.User
    database.SqlDB.Raw("SELECT u.user_id, u.username FROM tb_users u JOIN tb_follow_list f1 ON u.user_id
= f1.follow_user_id JOIN tb_follow_list f2 ON u.user_id = f2.user_id WHERE f1.user_id = ? AND
f2.follow_user_id = ?", userId, userId).Scan(&friendsUsers)
    return friendsUsers
}

// 关注
func (FollowMapper) Add(follow model.Follow) int64 {
    //新增数据
    res := database.SqlDB.Create(&follow)
    return res.RowsAffected
}

// 取消关注
func (FollowMapper) Delete(follow model.Follow) int64 {
    //删除数据
    res := database.SqlDB.Delete(&model.Follow{}, follow.FollowID)
    return res.RowsAffected
}

// 判断是否已关注或者返回已关注的id
func (FollowMapper) Judge(follow model.Follow) (int64, uint64) {
    var followUser model.Follow
    res := database.SqlDB.Where("user_id = ?", follow.UserID).Where("follow_user_id = ?",
follow.FollowUserID).Find(&followUser)
    return res.RowsAffected, followUser.FollowID
}

func (FollowMapper) FindAll() []model.Follow {
    var follows []model.Follow

```

```

//查询数据
database.SqlDB.Find(&follows)
return follows
}

func (FollowMapper) Update(follow model.Follow) int64 {
//更新数据
res := database.SqlDB.Save(follow)
return res.RowsAffected
}

```

## service/followService.go

```

package curd

import (
    "go_douyin/dao"
    "go_douyin/model" "time")

type FollowService struct {
    followMapper *dao.FollowMapper
}

func NewFollowService() *FollowService {
    return &FollowService{
        followMapper: dao.NewFollowMapper(),
    }
}

// 关注
func (h *FollowService) FollowAction(follow model.Follow) bool {
    isFollow, _ := h.followMapper.Judge(follow)
    if isFollow > 0 {
        return false
    } else {
        follow.CreateTime = time.Now()
        row := h.followMapper.Add(follow)
        if row > 0 {
            return true
        } else {
            return false
        }
    }
}

// 取消关注
func (h *FollowService) CancalFollowAction(follow model.Follow) bool {
    isFollow, followId := h.followMapper.Judge(follow)
    if isFollow > 0 {
        follow.FollowID = followId
        row := h.followMapper.Delete(follow)
        if row > 0 {
            return true
        } else {
            return false
        }
    }
}

```

```

    } else {
        return false
    }
}

// 获取关注列表
func (h *FollowService) FollowList(userId uint64) []model.User {
    return h.followMapper.FollowFindList(userId)
}

// 获取粉丝列表
func (h *FollowService) FansList(userId uint64) []model.User {
    return h.followMapper.FansFindList(userId)
}

// 获取好友列表
func (h *FollowService) FriendsList(userId uint64) []model.User {
    return h.followMapper.FriendsFindList(userId)
}

```

## controller/followController.go

```

package controller

import (
    "fmt"
    "github.com/gin-gonic/gin"    "go_douyin/model"    "go_douyin/service/follow"    JWT
    "go_douyin/service/user/token"
    "go_douyin/utils/response")

type FollowController struct {
    followService *curd.FollowService
}

func NewFollowController() *FollowController {
    return &FollowController{followService: curd.NewFollowService()}
}

// 关系操作
func (h *FollowController) FollowAction(c *gin.Context) {
    var requestBody map[string]interface{}
    requestBody = make(map[string]interface{})
    // 解析请求体
    c.ShouldBindJSON(&requestBody)
    // 获取请求参数
    //在 HTTP POST 请求中, 请求体中的数据通常是以字符串形式发送的。JSON 格式中的数字默认都是浮点型, 默认都是
    float64 类型
    token := requestBody["token"].(string)
    toUserId, _ := requestBody["to_user_id"].(float64)
    // 1-关注, 2-取消关注
    actionType, _ := requestBody["action_type"].(float64)

    // 解析JWT
    userTokenFactory := JWT.CreateUserFactory()
    customClaims, _ := userTokenFactory.ParseToken(token)

```

```

if actionType == 1 {
    // 关注
    var follow model.Follow
    follow.UserID = customClaims.UserID
    follow.FollowUserID = uint64(toUserId)
    if h.followService.FollowAction(follow) {
        response.Success(c, "关注成功", gin.H{})
    } else {
        response.Fail(c, -1, "关注失败", gin.H{})
    }
}

} else if actionType == 2 {
    // 取消关注
    var follow model.Follow
    follow.UserID = customClaims.UserID
    follow.FollowUserID = uint64(toUserId)
    if h.followService.CancalFollowAction(follow) {
        response.Success(c, "取消关注成功", gin.H{})
    } else {
        response.Fail(c, -1, "取消关注失败", gin.H{})
    }
} else {
    fmt.Println(actionType)
    response.Fail(c, -1, "操作失败", gin.H{})
}
}

// 用户关注列表
func (h *FollowController) FollowList(c *gin.Context) {
    // 获取请求参数
    token := c.Query("token")
    userId := c.Query("user_id")
    // 解析JWT
    userTokenFactory := JWT.CreateUserFactory()
    customClaims, _ := userTokenFactory.ParseToken(token)
    fmt.Println(userId)
    followList := h.followService.FollowList(customClaims.UserID)
    response.Success(c, "获取成功", gin.H{
        "user_list": followList,
    })
}

// 用户粉丝列表
func (h *FollowController) FansList(c *gin.Context) {
    // 获取请求参数
    token := c.Query("token")
    userId := c.Query("user_id")
    // 解析JWT
    userTokenFactory := JWT.CreateUserFactory()
    customClaims, _ := userTokenFactory.ParseToken(token)
    fmt.Println(userId)
    followList := h.followService.FansList(customClaims.UserID)
    response.Success(c, "获取成功", gin.H{
        "user_list": followList,
    })
}

```

```
// 用户好友列表
func (h *FollowController) FriendsList(c *gin.Context) {
    // 获取请求参数
    token := c.Query("token")
    userId := c.Query("user_id")
    userTokenFactory := JWT.CreateUserFactory()
    customClaims, _ := userTokenFactory.ParseToken(token)
    fmt.Println(userId)
    followList := h.followService.FriendsList(customClaims.UserID)
    response.Success(c, "获取成功", gin.H{
        "user_list": followList,
    })
}
```

## router/router.go

```
package router

import (
    "github.com/dv Wright/xss-mw"
    "github.com/gin-gonic/gin"    "go_douyin/controller"    "go_douyin/utils/cors")

func SetupRouter() *gin.Engine {
    router := gin.Default()
    var xssMdlwr xss.XssMw
    router.Use(xssMdlwr.RemoveXss())
    userController := controller.NewUserController()
    followController := controller.NewFollowController()
    // 用户组：登录注册，获取个人信息
    v1 := router.Group("/douyin/user")
    {
        v1.POST("register", userController.Register)
        v1.POST("login", userController.Login)
        v1.GET("/", userController.GetInfo)
    }
    // 社交组：关注，粉丝相关信息
    v2 := router.Group("/douyin/relation")
    {
        v2.POST("action", followController.FollowAction)
        v2.GET("follow/list", followController.FollowList)
        v2.GET("follower/list", followController.FansList)
        v2.GET("friend/list", followController.FriendsList)
    }
    // 允许跨域
    router.Use(cors.Next())
    return router
}
```

## test/follow\_test.go

```
package test

import (
    "go_douyin/dao"
    "go_douyin/database"    "go_douyin/model"    "testing"    "time")
```



```

//关注
func TestFollowAction(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    follow := model.Follow{}
    // 因为有外键的限制，所以必须都是存在的
    follow.UserID = 51
    follow.FollowUserID = 1
    follow.CreateTime = time.Now()
    t.Log(followdao.Add(follow))
}

//取消关注
func TestCancelFollowAction(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    follow := model.Follow{}
    // 因为有外键的限制，所以必须都是存在的
    follow.FollowID = 6
    follow.UserID = 5
    follow.FollowUserID = 51
    follow.CreateTime = time.Now()
    t.Log(followdao.Delete(follow))
}

//取消关注
func TestJudgeFollowAction(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    follow := model.Follow{}
    // 因为有外键的限制，所以必须都是存在的
    follow.UserID = 5
    follow.FollowUserID = 51
    t.Log(followdao.Judge(follow))
}

//测试关注的联表语句是否正常
func TestFollow(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    t.Log(followdao.FollowFindList(1))
}

//测试粉丝的联表语句是否正常
func TestFans(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    t.Log(followdao.FansFindList(1))
}

//测试好友的联表语句是否正常

```

```
func TestFriend(t *testing.T) {
    //注意必须先初始化
    database.SqlClient()
    followdao := dao.NewFollowMapper()
    t.Log(followdao.FriendsFindList(1))
}
```

(2023.1.23)

## 防止XSS攻击

### 来源

中间件推荐:

<https://www.topgoer.com/gin%E6%A1%86%E6%9E%B6/gin%E4%B8%AD%E9%97%B4%E4%BB%B6/%E4%B8%AD%E9%97%B4%E4%BB%B6%E6%8E%A8%E8%8D%90.html>

例子教程: <https://github.com/dvwright/xss-mw>

### 配置

```
go get -u github.com/dvwright/xss-mw
```

### 代码

(直接使用第三方库)

关键

```
var xssMdlwr xss.XssMw
router.Use(xssMdlwr.RemoveXss())
```

完整代码

router/router.go

```
package router

import (
    "github.com/dvwright/xss-mw"
    "github.com/gin-gonic/gin"    "go_douyin/controller"    "go_douyin/utils/cors")

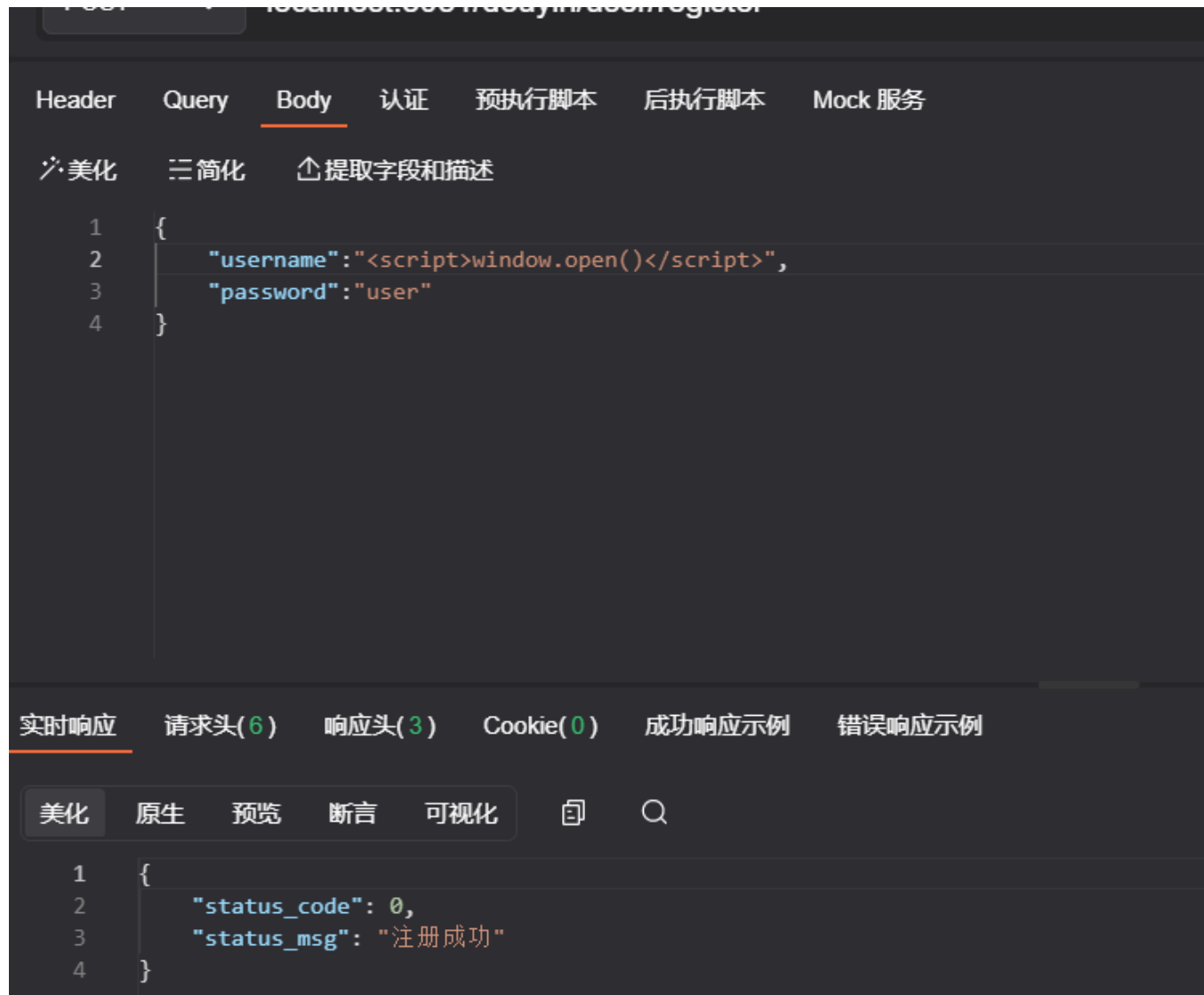
func SetupRouter() *gin.Engine {
    router := gin.Default()
    var xssMdlwr xss.XssMw
    router.Use(xssMdlwr.RemoveXss())
    userController := controller.NewUserController()

    v1 := router.Group("/douyin/user")
    {
        v1.POST("register", userController.Register)
        v1.POST("login", userController.Login)
        v1.GET("/", userController.GetInfo)
    }
    //允许跨域
    router.Use(cors.Next())
}
```

```
return router
}
```

## 运行结果

未使用前



user_id	username	password	follow_count	follow
1	admin	admin	(Null)	
3	admin1	admin	(Null)	
4	123	123	(Null)	
5	user	93f880f9cdeb43b08	0	
6	<script>window.open()	93f880f9cdeb43b08	0	

使用后

4	123	123
5	user	93f880f9cdeb43b08
34		d41d8cd98f00b204e
42	<script>alert('qggca	93f880f9cdeb43b08
50	<script>alrt('qggcaa	93f880f9cdeb43b08
51	aa	93f880f9cdeb43b08
52	qwfaa	93f880f9cdeb43b08

(2023.1.22)

## 跨域

### 代码

使用第三方库

utils/cors/cors.go

```
package cors

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

    // 允许跨域
    func Next() gin.HandlerFunc {
        return func(c *gin.Context) {
            method := c.Request.Method
            c.Header("Access-Control-Allow-Origin", "*")
            c.Header("Access-Control-Allow-Headers", "Access-Control-Allow-Headers,Authorization,User-Agent,Keep-Alive, Content-Type, X-Requested-With,X-CSRF-Token,AccessToken,Token")
            c.Header("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT, PATCH, OPTIONS")
            c.Header("Access-Control-Expose-Headers", "Content-Length, Access-Control-Allow-Origin, Access-Control-Allow-Headers, Content-Type")
            c.Header("Access-Control-Allow-Credentials", "true")

            // 放行所有OPTIONS方法
            if method == "OPTIONS" {
                c.AbortWithStatus(http.StatusOK)
            }
            c.Next()
        }
    }
}
```

router/router.go

```
package router

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

```
    "go_douyin/controller" "go_douyin/utils/cors")

func SetupRouter() *gin.Engine {
    router := gin.Default()
    userController := controller.NewUserController()
    v1 := router.Group("/douyin/user")
    {
        v1.POST("register", userController.Register)
        v1.POST("login", userController.Login)
        v1.GET("/", userController.GetInfo)
    }
    // 允许跨域
    router.Use(cors.Next())
    return router
}
```

(2023.1.22)

## 使用MD5加密密码

### 学习



在 Go 语言中，可以使用 `crypto/md5` 包来进行 MD5 加密。



思路如下：

1. 创建一个 `md5.New()` 的实例
2. 使用 `Write` 方法将需要加密的数据写入实例中
3. 使用 `Sum` 方法计算出数据的 MD5 值

代码如下：

```
package main

import (
    "crypto/md5"
    "fmt"
)

func main() {
    password := "password"
    passwordMd5 := md5.Sum([]byte(password))
    fmt.Printf("%x\n", passwordMd5)
}
```

Copy code

这样就可以得到密码的md5值。

为了更高的安全性，一般建议使用带盐值的md5加密方法。比如将密码和盐值拼接后再进行md5加密，这样即使密码相同，加密后的值也不同。

md5加密，这种随机盐值相同，加密后的值也不同。

例如：

```
package main

import (
    "crypto/md5"
    "fmt"
)

func main() {
    password := "password"
    salt := "salt"
```

Copy code

```
passwordMd5 := md5.Sum([]byte(password + salt))
fmt.Printf("%x\n", passwordMd5)
}
```

同时，更高的安全性还可以使用更高级的哈希算法，如 SHA256 或 bcrypt。

注意:MD5已经不再被认为是安全的哈希算法，如果需要用于密码存储，建议使用更安全的算法。

## 代码

utils/md5\_encrypt/md5\_encrypt.go

```
package md5_encrypt

import (
    "crypto/md5"
    "encoding/base64"
    "encoding/hex")

func MD5(params string) string {
    md5Ctx := md5.New()
    md5Ctx.Write([]byte(params))
    return hex.EncodeToString(md5Ctx.Sum(nil))
}

//先base64，然后MD5
func Base64Md5(params string) string {
    return MD5(base64.StdEncoding.EncodeToString([]byte(params)))
}
```

service/user/curd/userService.go (关键代码)

```
func (h *UserService) Register(user model.User) bool {
    user.CreateTime = time.Now()
    // 预先处理密码MD5加密
    user.Password = md5_encrypt.Base64Md5(user.Password)
    row := h.userMapper.Add(user)
    if row > 0 {
        return true
    } else {
        return false
    }
}

func (h *UserService) Login(username string, password string) (bool, model.User, string) {
    var user model.User = h.userMapper.Login(username, md5_encrypt.Base64Md5(password))
    // 比较结构体是否为空
    if reflect.DeepEqual(user, model.User{}) { //判断是否为空值
        //fmt.Println("user is empty")
        return false, user, ""
    }
}
```

```

    } else {
        // 生成JWT
        userTokenFactory := JWT.CreateUserFactory()
        if userToken, err := userTokenFactory.GenerateToken(user.UserID, user.Username, 28800); err ==
    nil {
        return true, user, userToken
    } else {
        return false, user, ""
    }
}
}
}

```

## 运行结果

user_id	username	password	follow_count	follow
1	admin	admin	(Null)	
3	admin1	admin	(Null)	
4	123	123	(Null)	
5	user	93f880f9cdeb43b08	0	
6	<script>window.op	93f880f9cdeb43b08	0	

(2023.1.21)

## 使用JWT

### 学习

在Go后端项目中使用JWT需要使用第三方库来实现，比如使用"github.com/dgrijalva/jwt-go"库来实现JWT的编码和解码。

1. 安装第三方库：使用 "go get github.com/dgrijalva/jwt-go" 安装。
2. 编码JWT：使用 jwt.New() 方法创建一个新的JWT实例，然后使用 Claim 对象设置 JWT 的有效负载。最后使用 jwt.SignedString() 方法签名并返回JWT字符串。
3. 解码JWT：使用 jwt.Parse() 方法解析JWT字符串，然后使用 Claims 对象访问JWT的有效负载。
4. 校验JWT：使用 jwt.ParseWithClaims() 方法解析JWT字符串，附加自定义的 Claims 对象并在解析过程中校验签名。
5. 在gin框架中使用：可以在中间件中使用上述方法对请求进行JWT校验。
6. 在gorm中使用：可以在数据库操作前先进行JWT校验，然后再进行相关操作。

请注意，这只是一种实现方式，具体实现可能因为业务需求而有所不同。

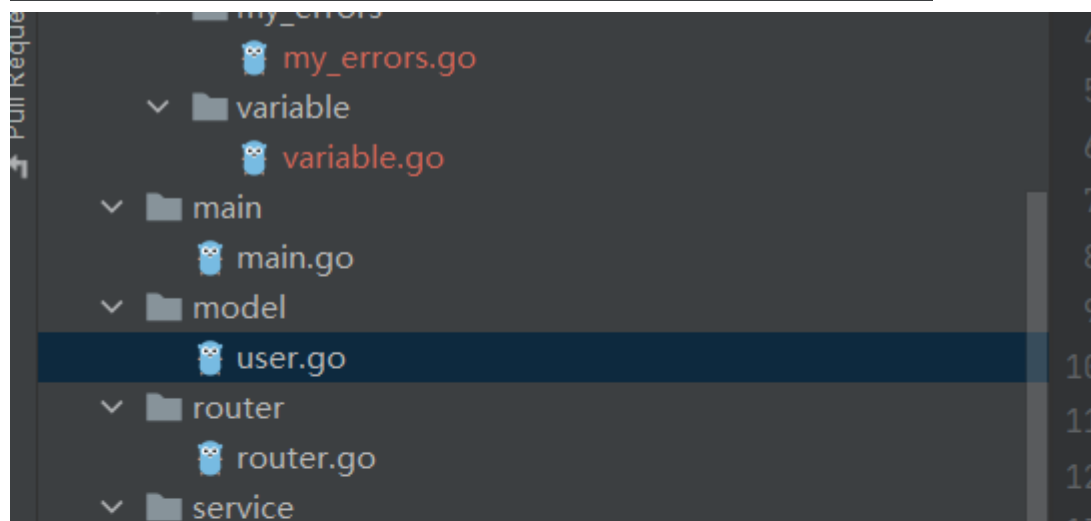
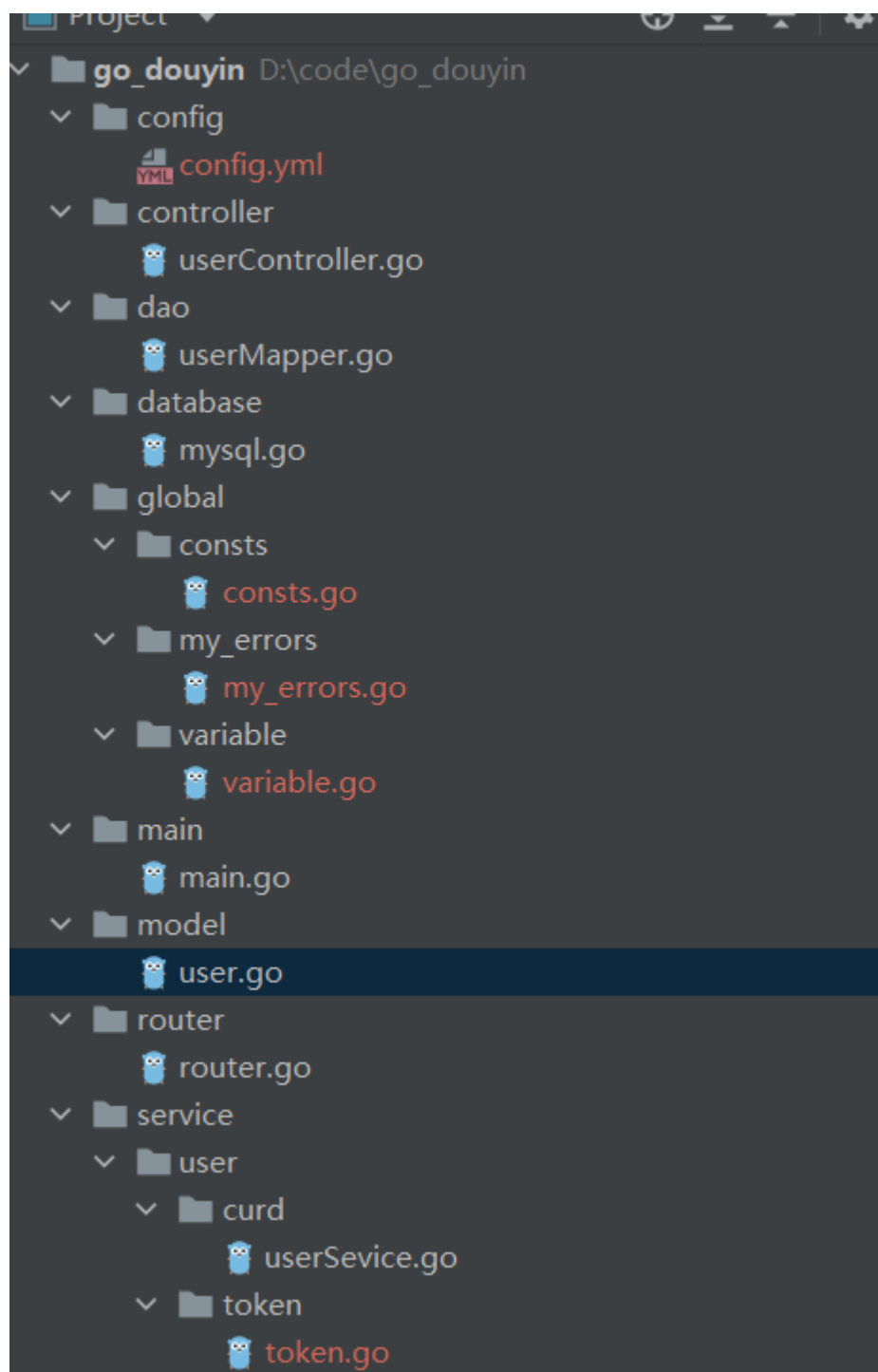


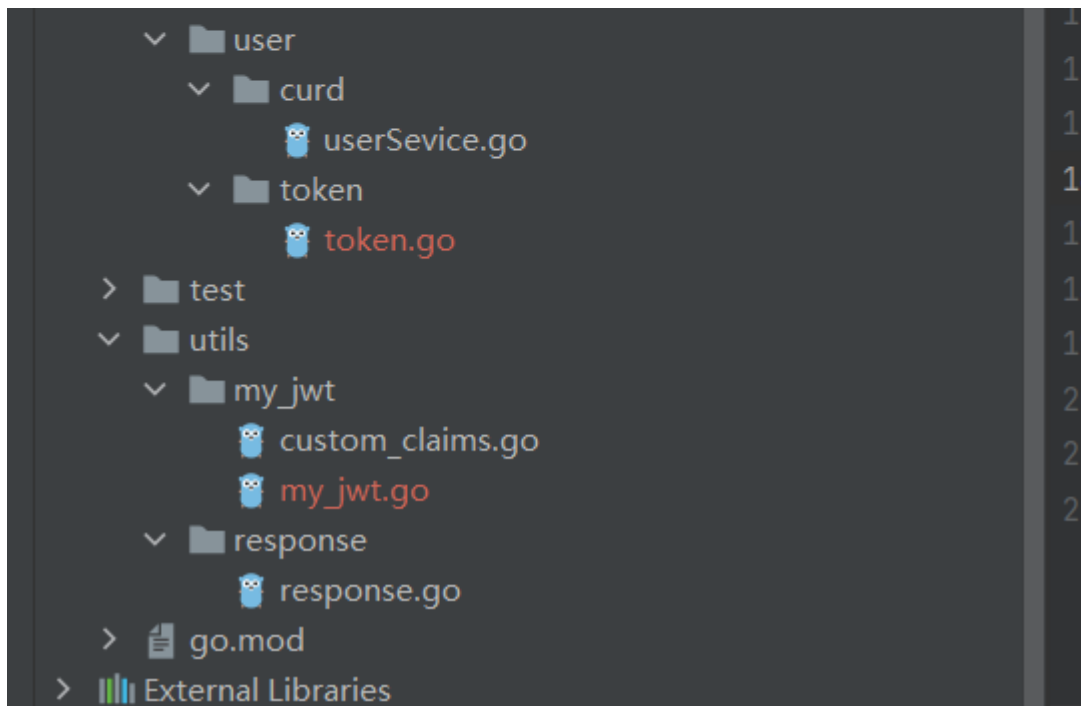
## 配置

```
go get github.com/dgrijalva/jwt-go
```

## 项目结构

```
├─.idea
├─config
├─controller
├─dao
├─database
├─global
│   ├─consts
│   ├─my_errors
│   └─variable
├─main
├─model
├─router
├─service
│   └─user
│       ├─curd
│       └─token
├─test
└─utils
    ├─my_jwt
    └─response
```





## Bug: sql: Scan error on column index 5, name "create\_time": unsupported Scan, storing driver.Value type []uint8 into type \*time.Time

解决方法

使用 gorm 这样的第三方库时，它提供了自动转换时间字段的功能。

首先在初始化数据库连接时设置相应的自动转换选项，如下：

```
db, err := gorm.Open("mysql", "user:password@/dbname?charset=utf8&parseTime=True&loc=Local")
```

这将告诉 gorm 使用 MySQL 驱动程序并解析时间字段。

然后在结构体字段上使用 gorm:"column:create\_time" 这样的标签来指定数据库字段的名称。

```
type User struct {
    CreateTime    time.Time    `gorm:"column:create_time" json:"create_time"`
}
```

这样，gorm 就会自动地将数据库中的时间戳字段转换为 Go 中的 time.Time 类型。

## 代码

(关键代码)

utils/my\_jwt/my\_jwt.go

```
package my_jwt

import "github.com/dgrijalva/jwt-go"

// 自定义jwt的声明字段信息+标准字段，参考地址：https://blog.csdn.net/codeSquare/article/details/99288718
type CustomClaims struct {
    UserID    uint64 `json:"user_id"` // user_id
    Username string `json:"username"` // username
}
```

```
    jwt.StandardClaims
}
```

utils/my\_jwt/custom\_claims.go

```
package my_jwt

import (
    "errors"
    "github.com/dgrijalva/jwt-go"    "go_douyin/global/my_errors"    "time")

// 使用工厂创建一个 JWT 结构体
func CreateMyJWT(signKey string) *JwtSign {
    if len(signKey) <= 0 {
        signKey = "douyin217@qqTEAM"
    }
    return &JwtSign{
        []byte(signKey),
    }
}

// 定义一个 JWT验签 结构体
type JwtSign struct {
    SigningKey []byte
}

// CreateToken 生成一个token
func (j *JwtSign) CreateToken(claims CustomClaims) (string, error) {
    // 生成jwt格式的header、claims 部分
    tokenPartA := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    // 继续添加密钥值，生成最后一部分
    return tokenPartA.SignedString(j.SigningKey)
}

// 解析Token
func (j *JwtSign) ParseToken(tokenString string) (*CustomClaims, error) {
    token, err := jwt.ParseWithClaims(tokenString, &CustomClaims{}, func(token *jwt.Token)
(interface{}, error) {
        return j.SigningKey, nil
    })
    if token == nil {
        return nil, errors.New(my_errors.ErrorsTokenInvalid)
    }
    if err != nil {
        if ve, ok := err.(*jwt.ValidationError); ok {
            if ve.Errors&jwt.ValidationErrorMalformed != 0 {
                return nil, errors.New(my_errors.ErrorsTokenMalFormed)
            } else if ve.Errors&jwt.ValidationErrorNotValidYet != 0 {
                return nil, errors.New(my_errors.ErrorsTokenNotActiveYet)
            } else if ve.Errors&jwt.ValidationErrorExpired != 0 {
                // 如果 TokenExpired ,只是过期（格式都正确），我们认为他是有效的，接下可以允许刷新操作
                token.Valid = true
                goto labelHere
            } else {
                return nil, errors.New(my_errors.ErrorsTokenInvalid)
            }
        }
    }
}
```

```

    }
}
labelHere:
    if claims, ok := token.Claims.(*CustomClaims); ok && token.Valid {
        return claims, nil
    } else {
        return nil, errors.New(my_errors.ErrorsTokenInvalid)
    }
}

// 更新token
func (j *JwtSign) RefreshToken(tokenString string, extraAddSeconds int64) (string, error) {

    if CustomClaims, err := j.ParseToken(tokenString); err == nil {
        CustomClaims.ExpiresAt = time.Now().Unix() + extraAddSeconds
        return j.CreateToken(*CustomClaims)
    } else {
        return "", err
    }
}
}

```

service/user/token/token.go

```

package token

import (
    "errors"
    "github.com/dgrijalva/jwt-go"    "go_douyin/global/my_errors"    "go_douyin/utls/my_jwt"    "time")

// CreateUserFactory 创建 userToken 工厂
func CreateUserFactory() *userToken {
    return &userToken{
        //userJwt: my_jwt.CreateMyJWT(variable.ConfigYml.GetString("Token.JwtTokenSignKey")),
        userJwt: my_jwt.CreateMyJWT("12314@"),
    }
}

type userToken struct {
    userJwt *my_jwt.JwtSign
}

// GenerateToken 生成token
func (u *userToken) GenerateToken(userid uint64, username string, expireAt int64) (tokens string, err error) {

    // 根据实际业务自定义token需要包含的参数, 生成token, 注意: 用户密码请勿包含在token
    customClaims := my_jwt.CustomClaims{
        UserID:    userid,
        Username:  username,
        // 特别注意, 针对前文的匿名结构体, 初始化的时候必须指定键名, 并且不带 jwt. 否则报错: Mixture of field:
        value and value initializers
        StandardClaims: jwt.StandardClaims{
            NotBefore: time.Now().Unix() - 10, // 生效开始时间
            ExpiresAt: time.Now().Unix() + expireAt, // 失效截止时间
        },
    },
}

```

```

    return u.userJwt.CreateToken(customClaims)
}

// ParseToken 将 token 解析为绑定传递的参数
func (u *userToken) ParseToken(tokenStr string) (CustomClaims my_jwt.CustomClaims, err error) {
    if customClaims, err := u.userJwt.ParseToken(tokenStr); err == nil {
        return *customClaims, nil
    } else {
        return my_jwt.CustomClaims{}, errors.New(my_errors.ErrorsParseTokenFail)
    }
}

```

user/curd/userService.go

```

package curd

import (
    "fmt"
    "go_douyin/dao"    "go_douyin/model"
    JWT "go_douyin/service/user/token"
    "reflect"    "time")

type UserService struct {
    userMapper *dao.UserMapper
}

func NewUserService() *UserService {
    return &UserService{
        userMapper: dao.NewUserMapper(),
    }
}

func (h *UserService) Register(user model.User) bool {
    user.CreateTime = time.Now()
    row := h.userMapper.Add(user)
    if row > 0 {
        return true
    } else {
        return false
    }
}

func (h *UserService) Login(username string, password string) (bool, model.User, string) {
    var user model.User = h.userMapper.Login(username, password)
    // 比较结构体是否为空
    if reflect.DeepEqual(user, model.User{}) { //判断是否为空值
        //fmt.Println("user is empty")
        return false, user, ""
    } else {
        // 生成JWT
        userTokenFactory := JWT.CreateUserFactory()
        if userToken, err := userTokenFactory.GenerateToken(user.UserID, user.Username, 28800); err ==
nil {
            return true, user, userToken
        } else {
            return false, user, ""
        }
    }
}

```

```

    }

}

func (h *UserService) GetInfo(userid uint64, token string) model.User {
    // 解析JWT
    userTokenFactory := JWT.CreateUserFactory()
    customClaims, _ := userTokenFactory.ParseToken(token)
    fmt.Print(customClaims)
    if userid == customClaims.UserID {
        return h.userMapper.GetInfo(userid)
    } else {
        // 返回空结构体
        return model.User{}
    }
}

```

controller/userController.go

```

package controller

import (
    "fmt"
    "github.com/gin-gonic/gin"    "go_douyin/model"    "go_douyin/service/user/curd"
    "go_douyin/utils/response"    "reflect"    "strconv")

type UserController struct {
    UserService *curd.UserService
}

func NewUserController() *UserController {
    return &UserController{UserService: curd.NewUserService()}
}

func (h *UserController) Register(c *gin.Context) {
    var user model.User
    c.BindJSON(&user)
    if h.UserService.Register(user) {
        response.Success(c, "注册成功", gin.H{})
        //c.JSON(http.StatusOK, gin.H{"code": "200", "msg": "OK", "data": "注册成功"})
    } else {
        response.Fail(c, -1, "注册失败", gin.H{})
    }
}

func (h *UserController) Login(c *gin.Context) {
    var user model.User
    c.BindJSON(&user)
    isLogin, userDB, token := h.UserService.Login(user.Username, user.Password)
    if isLogin {
        response.Success(c, "登录成功", gin.H{
            "user_id": userDB.UserID,
            "token":    token,
        })
    }
}

```

```

    } else {
        response.Fail(c, -1, "登录失败", gin.H{})
    }
}

func (h *UserController) GetInfo(c *gin.Context) {
    // 获取参数
    userid, _ := strconv.ParseUint(c.Query("user_id"), 10, 64)
    fmt.Println(c.Query("user_id"))
    fmt.Println(userid)
    // 获取参数
    token := c.Query("token")
    user := h.UserService.GetInfo(userid, token)
    if !reflect.DeepEqual(user, model.User{}) {
        response.Success(c, "获取成功", gin.H{
            "id":          user.UserID,
            "name":         user.Username,
            "follow_count":  user.FollowCount,
            "follower_count": user.FollowerCount,
            "is_follow":     false,
        },
        )
    } else {
        response.Fail(c, -1, "获取失败", gin.H{})
    }
}

```

model/user.go

```

package model

import (
    _ "gorm.io/gorm"
    "time"

    // 实体类
    type User struct {
        UserID          uint64    `json:"user_id"`           // user_id
        Username        string    `json:"username"`         // username
        Password        string    `json:"password"`         // password
        FollowCount     int64     `json:"follow_count"`     // follow_count
        FollowerCount   int64     `json:"follower_count"`   // follower_count
        CreateTime      time.Time `gorm:"column:create_time" json:"create_time"` // create_time
    }

    // 表名
    func (u *User) TableName() string {
        return "tb_users"
    }

```

## 统一返回格式（完善版）（2023.1.20）

### 代码

修改了一个函数，其他与之前一致



```

func ReturnJson(Context *gin.Context, httpCode int, dataCode int, msg string, data interface{}) {
    //Context.Header("key2020", "value2020")    //可以根据实际情况在头部添加额外的其他信息
    response := gin.H{
        //状态码, 0成功, 其他值失败
        "status_code": dataCode,
        //返回状态描述
        "status_msg": msg,
    }
    // 这种写法确保是平级, 而不是嵌套
    //if dataMap, ok := data.(map[string]interface{}); ok {
    // for k, v := range dataMap {    //    response[k] = v    // }    //}    // gin.H的直接加入
    for k, v := range data.(gin.H) {
        response[k] = v
    }
    Context.JSON(httpCode, response)
}

```

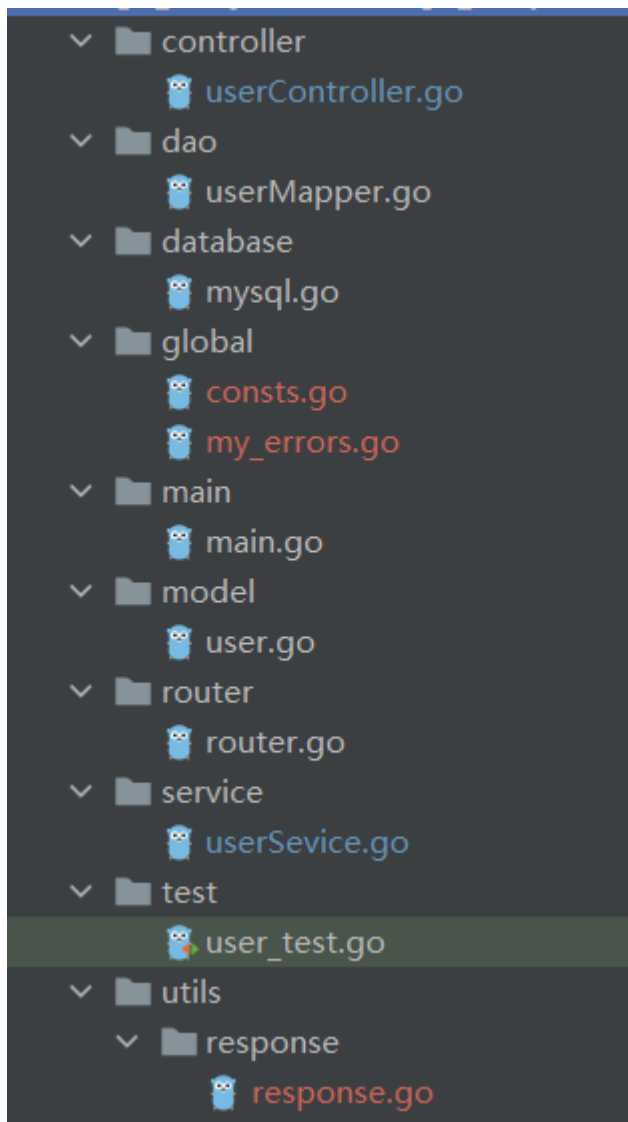
## 统一返回格式（待完善版）（2023.1.20）

### 项目结构

```

├─.idea
├─controller
├─dao
├─database
├─global
├─main
├─model
├─router
├─service
├─test
├─utils
└─response

```



## 代码

### global/consts.go

```
package global

// 这里定义的常量，一般是具有错误代码+错误说明组成，一般用于接口返回
const (
    // 进程被结束
    ProcessKilled string = "收到信号，进程被结束"
    // 表单验证器前缀
    ValidatorPrefix string = "Form_Validator_"
    ValidatorParamsCheckFailCode int = -400300
    ValidatorParamsCheckFailMsg string = "参数校验失败"

    //服务器代码发生错误
    ServerOccurredErrorCode int = -500100
    ServerOccurredErrorMsg string = "服务器内部发生代码执行错误，"
    GinSetTrustProxyError string = "Gin 设置信任代理服务器出错"

    // token相关
    JwtTokenOK int = 200100 //token有效
```

```

JwtTokenInvalid      int      = -400100          //无效的token
JwtTokenExpired      int      = -400101          //过期的token
JwtTokenFormatErrMsg int      = -400102          //提交的 token 格式错误
JwtTokenFormatErrMsg string   = "提交的 token 格式错误"          //提交的 token 格式错误
JwtTokenMustValid    string   = "token为必填项,请在请求header部分提交!" //提交的 token 格式错误

//SnowFlake 雪花算法
StartTimeStamp = int64(1483228800000) //开始时间戳 (2017-01-01)  MachineIdBits = uint(10)
//机器id所占的位数
SequenceBits    = uint(12)          //序列所占的位数
//MachineIdMax   = int64(-1 ^ (-1 << MachineIdBits)) //支持的最大机器id数量
SequenceMask    = int64(-1 ^ (-1 << SequenceBits)) //
MachineIdShift  = SequenceBits      //机器id左移位数
TimestampShift  = SequenceBits + MachineIdBits    //时间戳左移位数

// CURD 常用业务状态码 (0表示成功, 其他表示失败)
CurdStatusOkCode      int      = 0
CurdStatusOkMsg       string   = "Success"
CurdCreatFailCode     int      = -400200
CurdCreatFailMsg      string   = "新增失败"
CurdUpdateFailCode    int      = -400201
CurdUpdateFailMsg     string   = "更新失败"
CurdDeleteFailCode    int      = -400202
CurdDeleteFailMsg     string   = "删除失败"
CurdSelectFailCode    int      = -400203
CurdSelectFailMsg     string   = "查询无数据"
CurdRegisterFailCode  int      = -400204
CurdRegisterFailMsg   string   = "注册失败"
CurdLoginFailCode     int      = -400205
CurdLoginFailMsg      string   = "登录失败"
CurdRefreshTokenFailCode int    = -400206
CurdRefreshTokenFailMsg string  = "刷新Token失败"

//文件上传
FilesUploadFailCode    int      = -400250
FilesUploadFailMsg     string   = "文件上传失败, 获取上传文件发生错误!"
FilesUploadMoreThanMaxSizeCode int = -400251
FilesUploadMoreThanMaxSizeMsg string = "长传文件超过系统设定的最大值, 系统允许的最大值: "
FilesUploadMimeTypeFailCode int    = -400252
FilesUploadMimeTypeFailMsg string  = "文件mime类型不允许"

//websocket
WsServerNotStartCode  int      = -400300
WsServerNotStartMsg   string   = "websocket 服务没有开启, 请在配置文件开启, 相关路径: config/config.yml"
WsOpenFailCode        int      = -400301
WsOpenFailMsg         string   = "websocket open阶段初始化基本参数失败"

//验证码
CaptchaGetParamsInvalidMsg string = "获取验证码: 提交的验证码参数无效, 请检查验证码ID以及文件名后缀是否完整"
CaptchaGetParamsInvalidCode int    = -400350
CaptchaCheckParamsInvalidMsg string = "校验验证码: 提交的参数无效, 请检查 【验证码ID、验证码值】 提交时的键名是否与配置项一致"
CaptchaCheckParamsInvalidCode int    = -400351
CaptchaCheckOkMsg      string   = "验证码校验通过"
CaptchaCheckFailCode   int      = -400355

```

```
CaptchaCheckFailMsg      string = "验证码校验失败"
)
```

## my\_errors.go

```
package global

const (
    //系统部分
    ErrorsContainerKeyAlreadyExists string = "该键已经注册在容器中了"
    ErrorsPublicNotExists           string = "public 目录不存在"
    ErrorsConfigYamlNotExists       string = "config.yml 配置文件不存在"
    ErrorsConfigGormNotExists       string = "gorm_v2.yml 配置文件不存在"
    ErrorsStorageLogsNotExists      string = "storage/logs 目录不存在"
    ErrorsConfigInitFail            string = "初始化配置文件发生错误"
    ErrorsSoftLinkCreateFail        string = "自动创建软连接失败,请以管理员身份运行客户端(开发环境为goland
等,生产环境检查命令执行者权限), " +
        "最后一个可能:如果您是360用户,请退出360相关软件,才能保证go语言创建软连接函数: os.Symlink() 正常运行"
    ErrorsSoftLinkDeleteFail string = "删除软连接失败"

    ErrorsFuncEventAlreadyExists string = "注册函数类事件失败,键名已经被注册"
    ErrorsFuncEventNotRegister   string = "没有找到键名对应的函数"
    ErrorsFuncEventNotCall       string = "注册的函数无法正确执行"
    ErrorsBasePath               string = "初始化项目根目录失败"
    ErrorsTokenBaseInfo          string = "token最基本的格式错误,请提供一个有效的token!"
    ErrorsNoAuthorization         string = "token鉴权未通过,请通过token授权接口重新获取token,"
    ErrorsRefreshTokenFail       string = "token不符合刷新条件,请通过登陆接口重新获取token!"
    ErrorsParseTokenFail         string = "解析token失败"
    ErrorsGormInitFail           string = "Gorm 数据库驱动、连接初始化失败"
    ErrorsCasbinNoAuthorization  string = "Casbin 鉴权未通过,请在后台检查 casbin 设置参数"
    ErrorsGormNotInitGlobalPointer string = "%s 数据库全局变量指针没有初始化,请在配置文件
config/gorm_v2.yml 设置 Gormv2.%s.IsInitGlobalGormMysql = 1, 并且保证数据库配置正确 \n"
    // 数据库部分
    ErrorsDbDriverNotExists string = "数据库驱动类型不存在,目前支持的数据库类型: mysql、sqlserver、
postgresql, 您提交数据库类型: "
    ErrorsDialectorDbInitFail string = "gorm dialector 初始化失败,dbType:"
    ErrorsGormDBCreateParamsNotPtr string = "gorm Create 函数的参数必须是一个指针"
    ErrorsGormDBUpdateParamsNotPtr string = "gorm 的 Update、Save 函数的参数必须是一个指针(GinSkeleton ≥
v1.5.29 版本新增验证,为了完美支持 gorm 的所有回调函数,请在参数前面添加 & )" //redis部分
    ErrorsRedisInitConnFail string = "初始化redis连接池失败"
    ErrorsRedisAuthFail      string = "Redis Auth 鉴权失败,密码错误"
    ErrorsRedisGetConnFail   string = "Redis 从连接池获取一个连接失败,超过最大重试次数"
    // 表单参数验证器未通过时的错误
    ErrorsValidatorNotExists string = "不存在的验证器"
    ErrorsValidatorTransInitFail string = "validator的翻译器初始化错误"
    ErrorNotAllParamsIsBlank string = "该接口不允许所有参数都为空,请按照接口要求提交必填参数"
    ErrorsValidatorBindParamsFail string = "验证器绑定参数失败"

    //token部分
    ErrorsTokenInvalid string = "无效的token"
    ErrorsTokenNotActiveYet string = "token 尚未激活"
    ErrorsTokenMalFormed string = "token 格式不正确"

    //snowflake
    ErrorsSnowflakeGetIdFail string = "获取snowflake唯一ID过程发生错误"

    //websocket
```

```

// websocket
ErrorsWebSocketOnOpenFail      string = "websocket onopen 发生阶段错误"
ErrorsWebSocketUpgradeFail     string = "websocket Upgrade 协议升级, 发生错误"
ErrorsWebSocketReadMessageFail string = "websocket ReadPump(实时读取消息)协程出错"
ErrorsWebSocketBeatHeartFail   string = "websocket BeatHeart心跳协程出错"
ErrorsWebSocketBeatHeartsMoreThanMaxTimes string = "websocket BeatHeart 失败次数超过最大值"
ErrorsWebSocketSetWriteDeadlineFail string = "websocket 设置消息写入截止时间出错"
ErrorsWebSocketWriteMgsFail     string = "websocket Write Msg(send msg) 失败"
ErrorsWebSocketStateInvalid     string = "websocket state 状态已经不可用(掉线、卡死等愿意,
造成双方无法进行数据交互)"

// rabbitMq
ErrorsRabbitMqReconnectFail string = "RabbitMq消费者端掉线后重连失败, 超过尝试最大次数"

//文件上传
ErrorsFilesUploadOpenFail string = "打开文件失败, 详情: "
ErrorsFilesUploadReadFail string = "读取文件32字节失败, 详情: "

// casbin 初始化可能的错误
ErrorCasbinCanNotUseDbPtr      string = "casbin 的初始化基于gorm 初始化后的数据库连接指针, 程序检测到
gorm 连接指针无效, 请检查数据库配置! "
ErrorCasbinCreateAdaptFail     string = "casbin NewAdapterByDBUseTableName 发生错误: "
ErrorCasbinCreateEnforcerFail string = "casbin NewEnforcer 发生错误: "
ErrorCasbinNewModelFromStringFail string = "NewModelFromString 调用时出错: "
)

```

## utils/response/response.go

```

package response

import (
    "github.com/gin-gonic/gin"
    "go_douyin/global" "net/http")

func ReturnJson(Context *gin.Context, httpCode int, dataCode int, msg string, data interface{},
dataName string) {

    //Context.Header("key2020", "value2020") //可以根据实际情况在头部添加额外的其他信息
    Context.JSON(httpCode, gin.H{
        //状态码, 0成功, 其他值失败
        "status_code": dataCode,
        //返回状态描述
        "status_msg": msg,
        //具体数据, 相当于data
        dataName: data,
    })
}

//ReturnJsonFromString 将json字符串以标准json格式返回(例如, 从redis读取json格式的字符串, 返回给浏览器json格式)
func ReturnJsonFromString(Context *gin.Context, httpCode int, jsonStr string) {
    Context.Header("Content-Type", "application/json; charset=utf-8")
    Context.String(httpCode, jsonStr)
}

// 语法糖函数封装

```

```

//Success 直接返回成功
func Success(c *gin.Context, msg string, data interface{}, dataName string) {
    ReturnJson(c, http.StatusOK, global.CurdStatusOkCode, msg, data, dataName)
}

//Fail 失败的业务逻辑
func Fail(c *gin.Context, dataCode int, msg string, data interface{}, dataName string) {
    ReturnJson(c, http.StatusBadRequest, dataCode, msg, data, dataName)
    c.Abort()
}

// ErrorTokenBaseInfo token 基本的格式错误
func ErrorTokenBaseInfo(c *gin.Context) {
    ReturnJson(c, http.StatusBadRequest, http.StatusBadRequest, global.ErrorsTokenBaseInfo, "", "")
    //终止可能已经被加载的其他回调函数的执行
    c.Abort()
}

//ErrorTokenAuthFail token 权限校验失败
func ErrorTokenAuthFail(c *gin.Context) {
    ReturnJson(c, http.StatusUnauthorized, http.StatusUnauthorized, global.ErrorsNoAuthorization, "", "")
    //终止可能已经被加载的其他回调函数的执行
    c.Abort()
}

//ErrorTokenRefreshFail token不符合刷新条件
func ErrorTokenRefreshFail(c *gin.Context) {
    ReturnJson(c, http.StatusUnauthorized, http.StatusUnauthorized, global.ErrorsRefreshTokenFail, "", "")
    //终止可能已经被加载的其他回调函数的执行
    c.Abort()
}

//token 参数校验错误
func TokenErrorParam(c *gin.Context, wrongParam interface{}, dataName string) {
    ReturnJson(c, http.StatusUnauthorized, global.ValidatorParamsCheckFailCode,
        global.ValidatorParamsCheckFailMsg, wrongParam, dataName)
    c.Abort()
}

// ErrorCasbinAuthFail 鉴权失败, 返回 405 方法不允许访问
func ErrorCasbinAuthFail(c *gin.Context, msg interface{}, dataName string) {
    ReturnJson(c, http.StatusMethodNotAllowed, http.StatusMethodNotAllowed,
        global.ErrorsCasbinNoAuthorization, msg, dataName)
    c.Abort()
}

//ErrorParam 参数校验错误
func ErrorParam(c *gin.Context, wrongParam interface{}, dataName string) {
    ReturnJson(c, http.StatusBadRequest, global.ValidatorParamsCheckFailCode,
        global.ValidatorParamsCheckFailMsg, wrongParam, dataName)
    c.Abort()
}

// ErrorSystem 系统执行代码错误
func ErrorSystem(c *gin.Context, msg string, data interface{}, dataName string) {

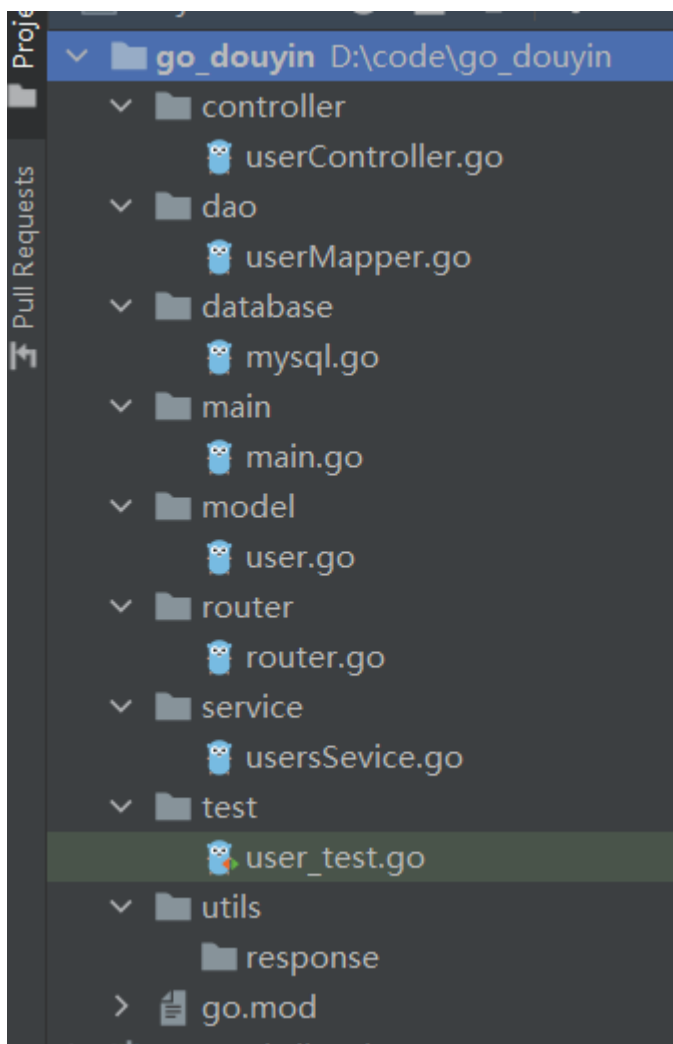
```

```
ReturnJson(c, http.StatusInternalServerError, global.ServerOccurredErrorCode,
global.ServerOccurredErrorMsg+msg, data, dataName)
c.Abort()
}
```

## 登录注册功能（纯数据库实现）（2023.1.19）

### 项目结构

```
├─.idea
├─controller
├─dao
├─database
├─main
├─model
├─router
├─service
├─test
└─utils
```



由上面可知，基本的结构主要就是由路由层（router），控制层（controller），服务层（service），数据访问层（dao），数据模型层（model）构成。

在这里与原本JAVA后端的区别是，service层就直接实现具体业务代码，JAVA中还有Impl层

## 数据库代码

db\_douyin/tb\_user

```
CREATE TABLE users (  
  user_id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '用户唯一标识', -- 用户id, 主键, 自增  
  username VARCHAR(255) NOT NULL COMMENT '用户名', -- 用户名, 唯一键  
  password VARCHAR(255) NOT NULL COMMENT '密码', -- 密码  
  follow_count BIGINT UNSIGNED DEFAULT 0 COMMENT '用户关注总数', -- 用户关注总数  
  follower_count BIGINT UNSIGNED DEFAULT 0 COMMENT '用户粉丝总数', -- 用户粉丝总数  
  create_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '用户注册时间', -- 用户注册时间  
  PRIMARY KEY (user_id),  
  UNIQUE KEY (username) -- 保证用户名唯一  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='存储用户信息';
```

## 代码

### database/mysql.go

连接数据库，这里可以当成是数据库的配置之类的，以"数据库账户:数据库密码@(IP地址:端口号)/数据库名?timeout=10s&readTimeout=30s&writeTimeout=60s"的形式填入自己对应的数据。

此处定义了一个全局变量以及一个初始化数据库的函数，注意在main函数需要初始化一次数据库，即调用 `database.SqlClient()`，否则数据库就不能用

```
package database  
  
import (  
  "gorm.io/driver/mysql"  
  "gorm.io/gorm"  
  "reflect")  
  
/**  
 * @Description:全局 DB */  
var (  
  SqlDB *gorm.DB  
)  
  
/**  
 * @Description: 初始化数据库  
 * @return *gorm.DB  
 */  
func SqlClient() *gorm.DB {  
  if SqlDB == nil || reflect.DeepEqual(SqlDB, gorm.DB{}) {  
    // 声明连接字符串  
    dsn := "数据库账户:数据库密码@(IP地址:端口号)/数据库名?timeout=10s&readTimeout=30s&writeTimeout=60s"  
    // 开启连接  
    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})  
    if err != nil {  
      //panic("failed to connect database")  
    }  
    SqlDB = db  
  }  
  return db
```



```

    return db
}
return SqlDB
}

```

## model/user.go

这里是数据模型层，这里可以先在数据库定义好结构，然后用XO工具生成model层的数据，注意这里实体类的变量首字母要大写，否则会不成功；这里还有一个函数是用来返回表名的。`func (u *User) TableName() string`即为User类返回string类型的意思

补充：

数据库设计部分规范

【强制】id类型没有特殊要求，必须使用bigint unsigned，禁止使用int，即使现在的数据量很小。id如果是数字类型的话，必须是8个字节。

【推荐】字段尽量设置为 NOT NULL，为字段提供默认值。如字符型的默认值为一个空字符串；数值型默认值为数值 0；逻辑型的默认值为数值 0；

【推荐】每个字段和表必须提供清晰的注释

【推荐】时间统一格式：'YYYY-MM-DD HH:MM:SS'

【强制】表达是与否概念的字段，必须使用 is\_xxx 的方式命名，数据类型是 unsigned tinyint（1表示是，0表示否）。说明：任何字段如果为非负数，必须是 unsigned。

正例：表达逻辑删除的字段名 is\_deleted，1 表示删除，0 表示未删除。

【强制】表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。说明：MySQL 在 Windows 下不区分大小写，但在 Linux 下默认是区分大小写。因此，数据库名、表名、字段名，都不允许出现任何大写字母，避免节外生枝。正例：health\_user，rdc\_config，level3\_name 反例：HealthUser，rdcConfig，level\_3\_name

【强制】表名不使用复数名词。说明：表名应该仅仅表示表里面的实体内容，不应该表示实体数量，对应于 DO 类名也是单数形式，符合表达习惯。

```

package model

import (
    "database/sql"
    "time")

// 实体类
type User struct {
    UserID      uint64      `json:"user_id"`      // user_id
    Username    string      `json:"username"`      // username
    Password    string      `json:"password"`      // password
    FollowCount sql.NullInt64 `json:"follow_count"`  // follow_count
    FollowerCount sql.NullInt64 `json:"follower_count"` // follower_count
    CreateTime  time.Time   `json:"create_time"`   // create_time
}

// 表名
func (u *User) TableName() string {
    return "tb_users"
}

```

## dao/userMapper.go

数据访问层（dao），这里是与数据库连接的，相当于JAVA注解或者XML写的sql语句，这里还是直接理解成一个类即可，需要有初始化函数 `NewUserMapper()`，下面是基本增删查改的格式

补充

常用gorm操作

```
go // 查询所有的记录 db.Find(&users) //// SELECT * FROM users; db.Where("name = ?", "jinzhu").First(&user)
db.Limit(3).Find(&users) //// SELECT * FROM users LIMIT 3; db.Offset(3).Find(&users) //// SELECT * FROM
users OFFSET 3;
```

```
package dao

import (
    "go_douyin/database"
    "go_douyin/model"
)

// UserMapper 自定义UserMapper的类型，于user实体类对应即可
type UserMapper struct{}

func NewUserMapper() *UserMapper {
    return &UserMapper{}
}

func (UserMapper) Login(username string, password string) int64 {
    var users []model.User
    //查询数据
    res := database.SqlDB.Where("username = ?", username).Where("password = ?", password).Find(&users)
    return res.RowsAffected
}

func (UserMapper) FindAll() []model.User {
    var users []model.User
    //查询数据
    database.SqlDB.Find(&users)
    return users
}

func (UserMapper) Add(user model.User) int64 {
    //新增数据
    res := database.SqlDB.Create(&user)
    return res.RowsAffected
}

func (UserMapper) Update(user model.User) int64 {
    //更新数据
    res := database.SqlDB.Save(user)
    return res.RowsAffected
}

func (UserMapper) Delete(user model.User) int64 {
    //删除数据
    res := database.SqlDB.Delete(&model.User{}, user.UserID)
    return res.RowsAffected
}
```

## service/userService.go

服务层（service），注意这里初始化的同时，还初始化了mapper的，而且定义结构的时候，也多写了一个userMapper（可以理解成JAVA的注解另一个层），这里是直接写业务逻辑的

```

package service

import (
    "go_douyin/dao"
    "go_douyin/model"    "time")

type UserService struct {
    userMapper *dao.UserMapper
}

func NewUserService() *UserService {
    return &UserService{
        userMapper: dao.NewUserMapper(),
    }
}

func (h *UserService) Register(user model.User) bool {
    user.CreateTime = time.Now()
    row := h.userMapper.Add(user)
    if row > 0 {
        return true
    } else {
        return false
    }
}

func (h *UserService) Login(username string, password string) bool {
    row := h.userMapper.Login(username, password)
    if row > 0 {
        return true
    } else {
        return false
    }
}

```

## controller/userController.go

控制层，这里后面可以换成统一格式返回，`c.BindJSON(&user)`可以直接当成是绑定POST请求过来的数据，与实体类一一对应，这里的注意同上

```

package controller

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

type UserController struct {
    UserService *service.UserService
}

func NewUserController() *UserController {
    return &UserController{UserService: service.NewUserService()}
}

```

```

func (h *UserController) Register(c *gin.Context) {
    var user model.User
    c.BindJSON(&user)
    if h.UserService.Register(user) {
        c.JSON(http.StatusOK, gin.H{"code": "200", "msg": "OK", "data": "注册成功"})
    } else {
        c.JSON(http.StatusBadRequest, gin.H{"code": "500", "msg": "OK", "data": "注册失败"})
    }
}

func (h *UserController) Login(c *gin.Context) {
    var user model.User
    c.BindJSON(&user)
    if h.UserService.Login(user.Username, user.Password) {
        c.JSON(http.StatusOK, gin.H{"code": "200", "msg": "OK", "data": "登录成功"})
    } else {
        c.JSON(http.StatusBadRequest, gin.H{"code": "500", "msg": "OK", "data": "登录失败"})
    }
}

```

## router/router.go

路由层，以以下的格式进行，注意这里还需要将controller初始化一次，然后以Group的形式定义分组路由

```

package router

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

func SetupRouter() *gin.Engine {
    router := gin.Default()
    userController := controller.NewUserController()
    v1 := router.Group("/douyin/user")
    {
        v1.POST("register", userController.Register)
        v1.POST("login", userController.Login)
    }
    return router
}

```

## main.go

这里是运行的主函数，需要初始化数据库和初始化一下路由（让路由生效），以及定义一下端口

```

package main

import (
    "go_douyin/database"
    "go_douyin/router"
)

func main() {
    // 注意初始化数据库
    database.SqlClient()
}

```

```
r := router.SetupRouter()
r.Run(":8081")
}
```

## 下载XO（根据数据库用于生成model）

```
go get -u golang.org/x/tools/cmd/goimports

go env -w GOPROXY=[https://goproxy.cn](https://link.zhihu.com/?target=https%3A//goproxy.cn
"https://goproxy.cn")

go install golang.org/x/tools/cmd/goimports@latest

mkdir -p modelsW
xo mysql://root:12qwAS@43.139.72.246:3377/douyin -o models --template-path /path/to/custom/templates

xo schema mysql://root:12qwAS@43.139.72.246:3377/db_douyin -o models2
```

## 需求

### 方案

<https://bytedance.feishu.cn/docs/doccnKrCsU5Iac6eftnFBdsXTof#6QCRJV>

### 1.2 项目设计要求

极简版抖音项目划分为两大方向，互动方向和社交方向，两个方向均包含基础功能内容，在扩展功能上有所不同，具体内容见下表

	互动方向		社交方向	
基础功能项	视频 Feed 流、视频投稿、个人主页			
基础功能项说明	视频Feed流：支持所有用户刷抖音，视频按投稿时间倒序推出 视频投稿：支持登录用户自己拍视频投稿 个人主页：支持查看用户基本信息和投稿列表，注册用户流程简化			
方向功能项	喜欢列表	用户评论	关系列表	消息
方向功能项说明	登录用户可以对视频点赞，在个人主页喜欢Tab下能够查看点赞视频列表	支持未登录用户查看视频下的评论列表，登录用户能够发表评论	登录用户可以关注其他用户，能够在个人主页查看本人的关注数和粉丝数，查看关注列表和粉丝列表	登录用户在消息页展示已关注的用户列表，点击用户头像进入聊天页后可以发送消息