青训营大项目答辩汇报文档

一、项目介绍

项目基于Gin与GORM框架实现了**极简版抖音**APP接口，包含基础功能（视频Feed流，视频投稿，个人主页）和方向功能（喜欢列表，用户评论，关系列表，消息）。项目使用1024Code平台提供的MySQL 和 Redis 进行数据存储，使用 Github 作为代码托管平台，团队使用Git进行了全程的代码版本协作管理。

项目服务地址：https://1024code.com/codecubes/hz0kzsk

备用项目服务地址：http://8.130.16.80:8080

GitHub地址：https://github.com/JiangH156/TinyTik

二、项目分工

团队成员	主要贡献
黄江	**组长**，负责 *用户模块* 实现（注册、登录、信息管理），项目协调，技术选型
周帅鹏	负责 *社交模块* 实现（用户关注、粉丝、好友），文档撰写
周灿	负责 *评论和聊天模块* 实现（评论、消息管理）
殷家豪	负责 *喜欢、视频流和发布模块* 实现（喜欢、喜欢列表、视频流、视频投稿发布、发布列表），视频录制，测试

三、项目实现

3.1 技术选型与相关开发文档

技术选型

技术	功能	官网
Gin	Web 框架，路由注册	https://gin-gonic.com/zh-cn/
Gorm	ORM框架，用于对象关系映射	https://gorm.io/zh_CN/
MySQL	关系型数据库	https://www.mysql.com/
Redis	缓存数据库	https://redis.uptrace.dev/zh/guide/
JWT	跨域认证，生成和验证令牌	https://jwt.io/
Viper	配置文件	https://github.com/spf13/viper
Bcrypt	密码加密服务	https://godoc.org/golang.org/x/cryp
Validator	参数校验	https://github.com/go-playground/v
Uuid	ID生成	https://github.com/google/uuid
FFmpeg	封面截取	https://www.ffmpeg.org/download.l

本次开发过程中，团队研究决定选择了 Gin 作为后台的开发框架，并使用 GORM 作为持久层框架，使用 MySQL 作为数据库，并使用 Redis 进行缓存。

Web 框架

本次开发的项目是一个**极简版抖音**APP的后端项目，需要使用 Go 语言实现高性能的 HTTP 服务。所以团队选择了较为流行的 Gin 框架，Gin 是 Go 语言的一个很流行的 Web 开发框架，支持路由、中间件等常用的配置，而且得益于 Go 语言的性能，Gin 不仅性能有保证，运行开销也较低，配置起来也比较简单。

数据存储

本次项目主要需要实现用户、视频、评论、消息、关系等实体的业务逻辑，主要功能都需要实现对象的数据库增删改查操作，所以团队希望使用到 ORM（Object-Relational Mapping）框架简化开发，GORM 是一个强大的 Go 语言 ORM 框架，它提供了简单且直观的 API，帮助我们轻松地与数据库进行交互。

在选择数据库时，考虑到团队成员的技术栈以及 1024Code 平台的支持，团队最终选择了流行的关系型数据库 MySQL 作为数据库，并选择非关系型数据库 Redis 作为缓存。使用 MySQL 作为数据库，可以对数据进行存储，并且可以使用 GORM 进行操作，而对于诸如登录验证等需求，使用 Redis 作为缓存则可以降低数据库访问压力，提高服务效率。

身份验证

为了实现对用户请求进行验证，我们使用 JWT 进行 Token 的生成与验证，JWT 是 JSON Web Token 的缩写，是一种用于在网络应用之间传递声明（Claims）的开放标准（RFC 7519）。它是一种轻量级的安全性传输协议，可用于在客户端和服务器之间传递被数字签名的信息，具有开销少、传输安全等优点。

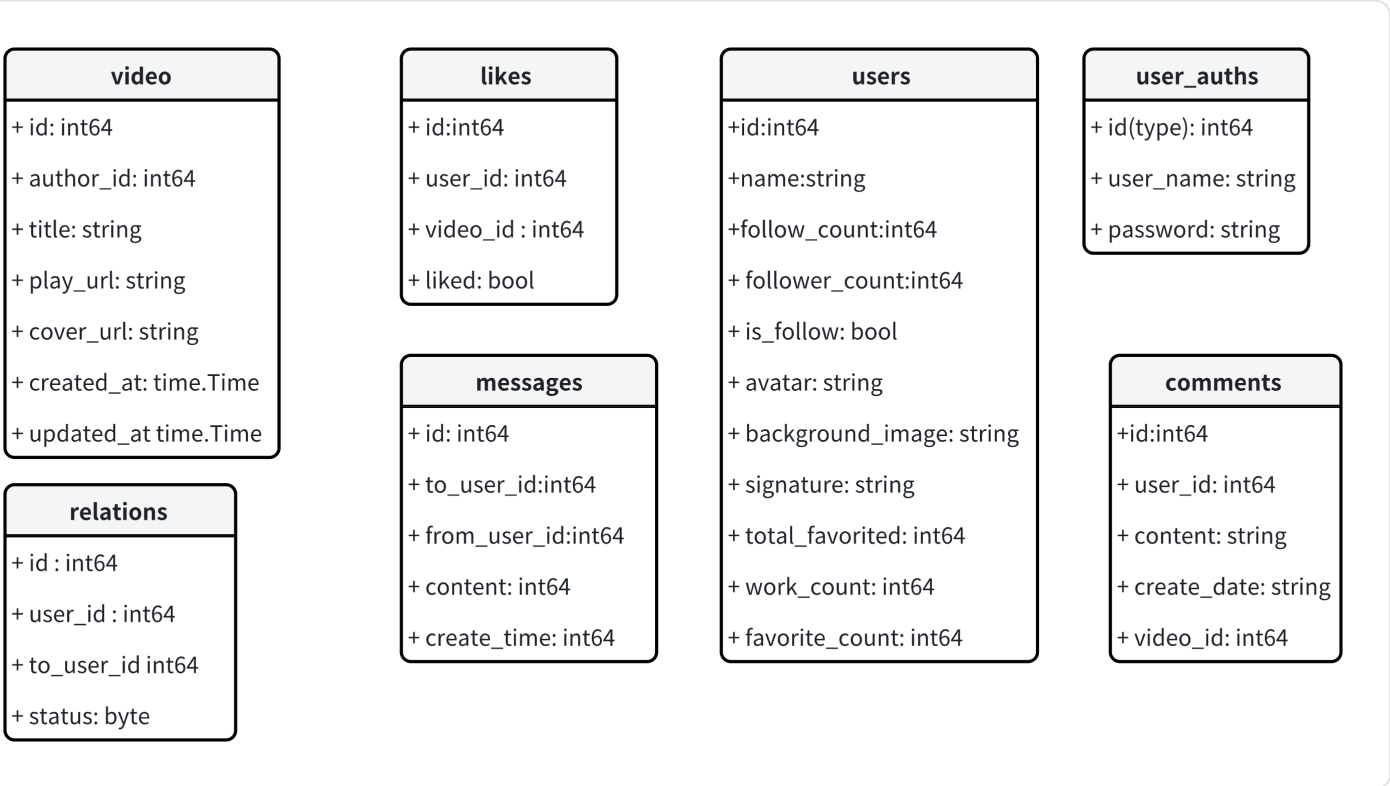
Viper配置文件

Viper 是一个灵活的 Go 语言库，用于处理配置文件，提供多种配置格式支持、合并配置、错误处理和热加载等功能，简化了配置管理和维护，增强了应用程序的可配置性和可扩展性。

视频处理

在用户视频的处理上，我们选用了业界最强大的开源音视频处理库 FFmpeg，它包括了一组用于处理音频、视频、字幕等多媒体数据的库与工具。FFmpeg提供了强大且灵活的功能，可以用于多种多媒体处理任务，如格式转换、编解码、流媒体传输、视频剪辑、视频合并等。

数据库设计



API文档

https://console-docs.apipost.cn/preview/599a053c1085c274/9b93e21dcae61608

3.2 架构设计

本团队成员大多并不熟悉 Go 语言开发，参与此次青训营是第一次使用 Go 语言进行 Web 开发，所以并没有采用更复杂的微服务架构，仍使用传统的单体应用架构实现了整体的功能服务。项目采用经典的 MVC 架构，将应用分为 Model，View 和 Controller 层。

Model 层主要定义了数据实体模型对应的结构体，用户的 HTTP 请求会由 API Router 路由至对应的 Controller，而 Controller 则主要负责请求的解析与返回，中间会调用相应的 Service 执行具体的业务逻辑，如果业务逻辑简单，也可以省略 Service，在 Controller 里进行简单处理后返回响应。部分 Service 会用到 Redis 缓存，会先查询 Redis 缓存，如果命中则读取 Redis 缓存作为结果，否则执行数据库操作。Repository 层主要是对 Model 的数据库增删改查操作进行了封装，Service 可以通过

Repository 实例进行数据的增删改查操作，从而完成业务逻辑的实现。View 对应服务响应的结果，本次项目在 Controller 里定义了对应的结构体。

项目的整体架构如图 3.1 所示。

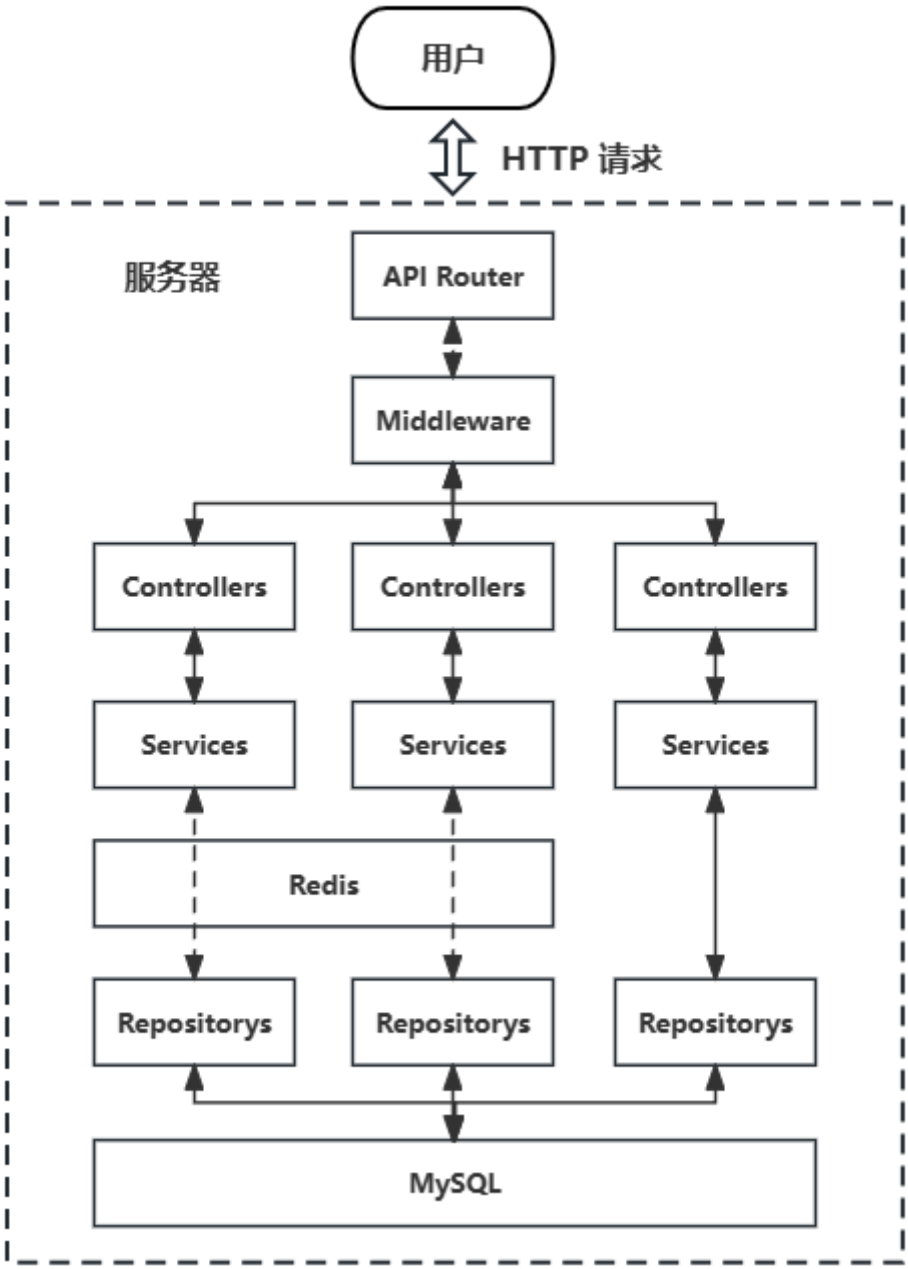


图 3.1 项目架构图

项目目录

```
TinyTik
├─cmd              -- 后台启动管理
├─common           -- 通用的代码或功能
├─config           -- 配置文件
├─controller       -- 控制器（Controller）层，接受请求并处理响应
├─FFmpeg           -- 视频处理文件
├─logs             -- 存放日志
```

```
├─middleware      -- 中间件层，处理请求处理前后的逻辑
├─model           -- 数据库实体（Model）层，处理数据库相关操作
├─public          -- 静态资源文件
├─resp            -- 响应处理层，格式化响应数据和处理异常
├─router          -- 路由（Router）层，负责处理路由和中间件的注册
├─service         -- 服务（Service）层，处理业务逻辑
├─test            -- 项目测试文件
├─utils           -- 工具类和通用函数
```

3.3 项目代码介绍

优化

事务

有些功能涉及多个表，我们使用了事务来保证数据的一致性

单例模式

在初始化配置和返回操作实例时，使用go的单例模式来确保一个类只有一个实例，提供全局唯一的访问点，有助于节省资源和简化代码逻辑。

```
    once.Do(func() {
        //配置mysql
        common.InitDB()
        // 配置logger
        loadLogger()
        // 配置redis
        loadRedis()

    })
```

接口校验

使用 var _ LikeRepositoy = (*likes)(nil) 来确保所有接口都正确实现

防SQL注入

使用参数化查询

```
    err := v.db.Where("author_id=?", userId).Find(&videos).Error
```

输入验证和过滤

用户名和密码字段添加validate的tag，使用validator库进行数据校验

用户模块实现（注册、登录、信息管理）

用户模块

用户模块负责用户信息的管理和登录注册授权验证。该模块被划分为两个子模块：User（用户信息管理）和 Auth（登录注册授权验证）。

User（用户信息管理）模块

User模块提供了用户信息的管理功能，根据前端所需接口提供相应功能，如获取用户信息等。

用户实体

根据项目接口文档定义用户User模型，其中IsF_follow字段是根据当前登录用户实时变化的值，而非User实体属性，放在User模型中是一个冗余字段。此处为了保持与前端接口对应，进行了保留

```
type User struct {
    Id              int64  `json:"id"   gorm:"id;primary_key;autoIncrement;comme
    Name            string `json:"name"   gorm:"name;type:varchar(32);omitempty;
    FollowCount     int64  `json:"follow_count" gorm:"follow_count;omitempty;com
    FollowerCount   int64  `json:"follower_count" gorm:"follower_count;omitempty
    IsFollow        bool   `json:"is_follow" gorm:"is_follow;omitempty;comment:是
    Avatar          string `json:"avatar" gorm:"avatar;omitempty;comment:用户头像
    BackgroundImage string `json:"background_image" gorm:"background_image;omite
    Signature       string `json:"signature" gorm:"signature;omitempty;comment:个
    TotalFavorited  int64  `json:"total_favorited" gorm:"total_favorited;omitemp
    WorkCount       int64  `json:"work_count" gorm:"work_count;omitempty;comment
    FavoriteCount   int64  `json:"favorite_count" gorm:"favorite_count;omitempty
}
```

获取用户信息

由于系统会频繁获取用户信息，为了降低数据库压力和提高效率，在用户登录系统时，后台会缓存用户信息和访问令牌（token）。获取用户信息接口调用时，会直接从Redis中获取用户信息，能够快速响应请求

```
func (r *RedisClient) UserLoginInfo(token string) (userInfo model.User, exist bo
    userBytes, err := r.GetUser(token)
    if err != nil {
        return userInfo, false
```

```
    }
    // json反序列化
    err = json.Unmarshal(userBytes, &userInfo)
    if err != nil {
        return userInfo, false
    }
    return userInfo, true
}
```

Auth（登录注册授权验证）模块

Auth模块负责用户的身份验证和授权管理。主要功能为:用户注册、用户登录

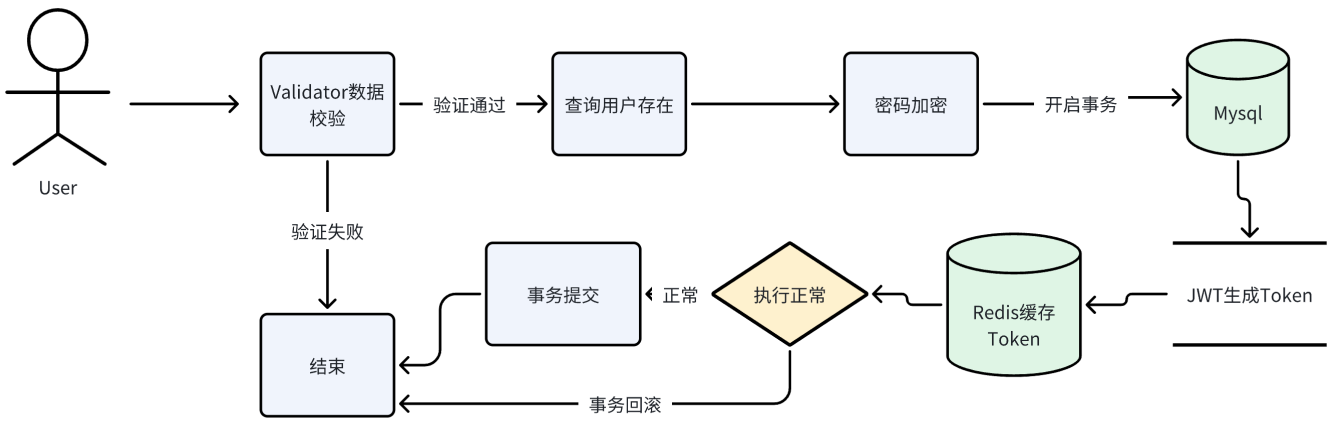
用户授权实体

主键为自动递增ID，因为用户授权模型与用户实体模型具有一定的相关性，所以注册保证用户和用户授权的主键ID字段统一。用户名和密码字段添加validate的tag，使用validator库进行数据校验

```
type UserAuth struct {
    ID       int64  `gorm:"id;primaryKey;autoIncrement;comment:用户id"`
    UserName string `gorm:"user_name;type:varchar(32);not null;comment:用户名称"
    Password string `gorm:"password;varchar(32);not null;comment:用户密码" valida
}
```

用户注册

用户注册流程为数据验证、加密密码、创建用户认证信息、生成访问令牌、存储用户信息等步骤



数据验证使用validator库进行数据校验

```
func ValidateUserAuth(userAuth model.UserAuth) error {
    validate := validator.New()
    if err := validate.Struct(userAuth); err != nil {
```

```
        return err
    }
    return nil
}
```

`GenToken`：该函数用于生成 JWT 令牌，使用用户名作为声明，过期时间默认24h，并使用预定义的签名密钥进行签名。

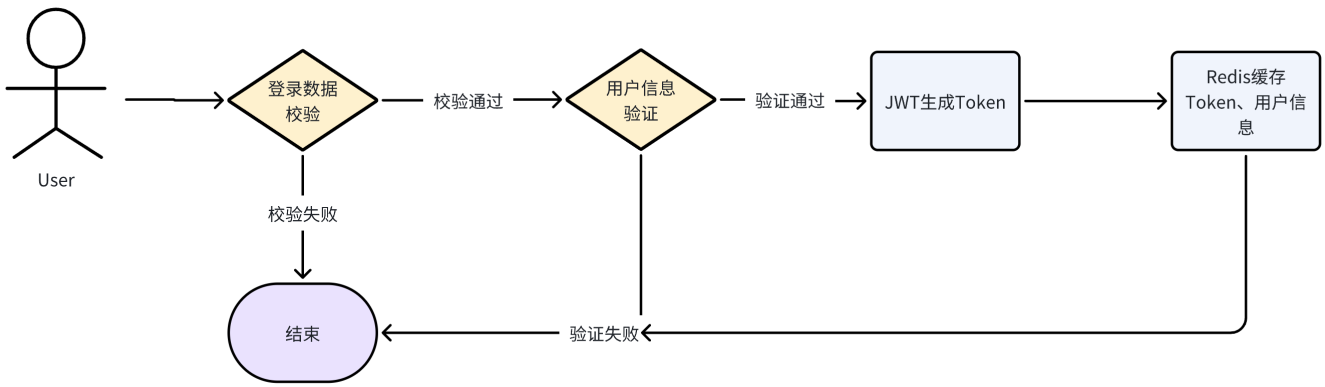
```
type UserClaims struct {
    UserName string
    jwt.RegisteredClaims
}

// JWT过期时间
const TokenExpireDuration = time.Hour * 24

// 用于签名的字符串
var jwtScret = []byte("TinyTik")
func GenToken(UserName string) (string, error) {
    //创建一个自己的声明
    claims := UserClaims{
        UserName,
        jwt.RegisteredClaims{
            //发行者
            Issuer: "TinyTik",
            //主题
            Subject: "TinyTik_TOKEN",
            //过期时间
            ExpiresAt: jwt.NewNumericDate(time.Now().Add(TokenExpireDuration)),
            //生效时间
            NotBefore: jwt.NewNumericDate(time.Now()),
            //发布时间
            IssuedAt: jwt.NewNumericDate(time.Now()),
        },
    }
    //使用指定的签名方法创建签名对象
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    //使用指定的secret签名并获得完整的编码后的字符串token
    tokenString, err := token.SignedString(jwtScret)
    if err != nil {
        return "", err
    }
    return tokenString, nil
}
```

用户鉴权表（UserAuth）和用户表（User)使用依赖注入的方式可以将事务的创建和管理交给外部的调用方，从而实现更灵活的事务控制。调用方可以在需要的时候创建事务，并将其传递给多个方法，以确保这些方法在同一个事务中执行。

```
// 添加用户鉴权
func (auth *AuthRepository) CreateAuth(tx *gorm.DB, userAuth *model.UserAuth) er
    if err := tx.Create(userAuth).Error; err != nil {
        return err
    }
    return nil
}
// 创建用户
func (r *UserRepository) CreateUser(tx *gorm.DB, user model.User) (err error) {
    if err := tx.Create(&user).Error; err != nil {
        return err
    }
    return nil
}
```

用户登录

用户登录流程为数据校验、验证用户信息、生成访问令牌、存储用户信息等步骤



考虑到前台频繁请求用户信息，为了降低数据库（Mysql）的压力，在用户登录时，缓存用户信息在Redis数据库中，提高效率，降低请求响应时间

```
// 缓存登录用户
func (r *RedisClient) SetUser(key string, user []byte) error {
    err := r.Set(key, user, expire)
    if err != nil {
        return err
    }
    return nil
```

```
8 }
```

社交模块实现（用户关注、粉丝、好友）

用户实体

本次项目首先按照接口文档定义了用户类 User 的模型，其中值得注意的是 is_follow 字段，我的理解是这是一个冗余字段，可能是考虑到没有必要因为一个单独的字段而定义 2 个高度相似的 Model（一个作为数据Model，一个作为View Model），在本次的项目实现中我们保留了 is_follow 字段，使用一个 User Model，而 is_follow 字段则在执行数据库查询时填充。

关系实体

用户关系的建模一般有两种常用的方式：基于关系型数据库和基于图形数据库两种。图形数据库的优势是图形数据库以图形结构存储数据，其中节点表示实体，边表示实体之间的关系。这种结构非常适合表示和存储如社交关系这种复杂关系，更加直观和高效。但是很多图形数据库不支持事务，且使用率并不高，为了求稳，本次项目依然选用了关系型数据库，通过对关系建表实现关系的存储。

```
type Relation struct {
    Id       int64 `gorm:"primaryKey;autoIncrement"`
    UserId   int64 `gorm:"primaryKey;foreignKey;reference:users.id"`
    ToUserId int64 `gorm:"primaryKey;foreignKey;reference:users.id"`
    Status   byte
}
```

关系表定义了ID、用户ID和关注用户ID作为复合主键，对关系的有向性进行约束，同时定义了外键约束。在表示用户关注的状态上，我使用了 Status 字段进行了表示，并定义了枚举变量表示用户关系的状态，这样在取关的情况下，不会对关系进行删除而是将字段置为 UNFOLLOW，实现类似于软删除的效果。

```
const (
    UNFOLLOW = iota
    FOLLOW
)
```

关系操作

关系的增删改查操作主要在 RelaRepo 中实现，本次项目中主要实现了是否关注、获取关注列表、获取粉丝列表、获取朋友列表、执行关注/取关操作的功能。RelaRepo 采用 sync.Once 实现了单例模式。

```go
type RelaRepo struct {
    DB *gorm.DB
}

var once sync.Once
var repo *RelaRepo

func GetRelaRepo() *RelaRepo {
    // 采用单例模式
    if repo == nil {
        once.Do(func() {
            repo = &RelaRepo{}
            repo.DB = common.GetDB()
        })
    }
    return repo
}
```

查询关系

使用 Where 和 First 进行关系查询，并通过 Status 字段的值判断是否关注，如果关系不存在视为未关注。

```go
func (r *RelaRepo) Followed(user *model.User, toUser *model.User) bool {
    rel, err := r.GetRelationById(user.Id, toUser.Id)
    if err != nil {
            return false
    }
    if rel.Status == model.FOLLOW {
            return true
    }
    return false
}

func (r *RelaRepo) GetRelationById(id int64, toId int64) (model.Relation, error)
    rel := model.Relation{}
    res := r.DB.Where("user_id = ? AND to_user_id = ?", id, toId).First(&rel)
    if errors.Is(res.Error, gorm.ErrRecordNotFound) {
            return rel, res.Error
    }
    return rel, nil
}
```

获取关注列表

要获取关注列表需要使用到表的 Join，使用 GORM 的 Joins 方法实现，而查询条件很显然是用户表中的用户ID Join 关系表中的被关注ID，同时限制关系表中的用户ID 和 Status。要注意的是 is_follow 字段，很显然关注列表中的用户都是“正在关注”的，也就是说 is_follow 字段为 true。

```
func (r *RelaRepo) GetFollowListById(id int64) ([]model.User, error) {
    users := []model.User{}
    res := r.DB.Model(&model.User{}).
            Select("users.*, true as is_follow").
            Joins("JOIN relations r ON users.id = r.to_user_id AND r.user_id = ?
            Scan(&users)
    return users, res.Error
}
```

获取粉丝列表

所谓A用户的粉丝其实就是关注A用户的用户，所以我们仍然使用 Join 操作。类似的，我们需要注意 is_follow 字段，关注用户A的用户并不一定被用户A关注，所以使用 IF 和 EXISTS 对用户A是否关注进行判断。

```
func (r *RelaRepo) GetFollowerListById(id int64) ([]model.User, error) {
    users := []model.User{}
    res := r.DB.Model(&model.User{}).
            Select("users.*, IF(EXISTS(SELECT * FROM relations r1 WHERE users.id
            Joins("JOIN relations r ON users.id = r.user_id AND r.to_user_id = ?
            Scan(&users)
    return users, res.Error
}
```

获取朋友列表

由于互相关注的用户视为朋友，所以可以类似地通过两次 Join，第一次 Join 临时表 query 筛选关注用户，第二次 Join 筛选出互关用户。

```
func (r *RelaRepo) GetFriendListById(id int64) ([]model.User, error) {
    users := []model.User{}
    query := r.DB.Table("relations").Select("to_user_id").Where("user_id = ?", i
    res := r.DB.Model(&model.User{}).
            Select("users.*, true as is_follow").
            Joins("JOIN (?) q ON users.id = q.to_user_id", query).
            Joins("JOIN relations r ON q.to_user_id = r.user_id AND r.to_user_id
            Scan(&users)
    return users, res.Error
```

```
}
```

执行关注/取关

由于关注和取关操作涉及多个表，为了保证数据的一致性，我们需要确保整个关注操作的原子性，所以这里使用 GORM 提供的事务接口，使用 Updates 方法更新用户表，使用 Save 方法更新关系表（因为关系可能并不存在，需要新增关系）。

```
func (r *RelaRepo) UpdateRelation(user model.User, toUser model.User, follow byt
    // 执行事务
    return r.DB.Transaction(func(tx *gorm.DB) error {
            logger.Debug("取消关注")
            if err := tx.Model(&user).Updates(user).Error; err != nil {
                    // 返回任何错误都会回滚事务
                    return err
            }

            if err := tx.Model(&toUser).Updates(toUser).Error; err != nil {
                    logger.Debug("取消关注失败")
                    return err
            }
            // 更新关系表
            rel, err := r.GetRelationById(user.Id, toUser.Id)
            if errors.Is(err, gorm.ErrRecordNotFound) {
                    rel.UserId = user.Id
                    rel.ToUserId = toUser.Id
            }
            rel.Status = follow
            if err := tx.Save(&rel).Error; err != nil {
                    return err
            }
            // 返回 nil 提交事务
            return nil
    })
}
```

项目还使用到了 GORM 的钩子函数，确保关系更新操作合法。

```
// 使用 Hook 进行合法性检查
func (u *User) BeforeUpdate(tx *gorm.DB) error {
    if u.FollowCount < 0 || u.FollowerCount < 0 {
        logger.Error("Invalid Update: Follow/Unfollow action yield minus value."
        return fmt.Errorf("操作数为负")
```

```
    }
    return nil
}
```

评论和聊天模块实现

评论实体

`Id`：评论的唯一标识符；`User`：评论所属用户的标识符；`User`：评论所属用户的标识符；`CreateDate`：评论的创建日期；`VideoId`：评论所属视频的标识符。

```
type Comment struct {
    Id         int64  `json:"id,omitempty" gorm:"primaryKey;autoIncrement:true"`
    User       int64  `json:"user_id"`
    Content    string `json:"content,omitempty"`
    CreateDate string `json:"create_date,omitempty"`
    VideoId    int64  `json:"video_id,omitempty"`
}
```

消息实体

`Id`：消息的唯一标识符；`ToUserId`：消息的接收者用户标识符；`ToUserId`：消息的接收者用户标识符；`Content`：消息的内容；`CreateTime`：消息的创建时间

```
type Message struct {
    Id         int64  `json:"id,omitempty" gorm:"primary_key;autoIncrement:true"
    ToUserId   int64  `json:"to_user_id,omitempty"`
    FromUserId int64  `json:"from_user_id,omitempty"`
    Content    string `json:"content,omitempty"`
    CreateTime int64  `json:"create_time,omitempty"`
}
```

保存评论

逻辑是将评论对象插入名为"comments"的数据库表中。`tx *gorm.DB`是一个GORM数据库事务对象，用于控制数据库事务的提交和回滚。

```
// 保存评论
func (c *CommentRepository) CreateComment(tx *gorm.DB, comment *model.Comment) (
    result := tx.Table("comments").Create(comment)
```

```
    if result.Error != nil {
        return 0, result.Error
    }
    // 获取插入后的评论对象，包括自动生成的ID
    return comment.Id, nil
}
```

删除评论

通过评论ID从数据库中删除评论数据。

```
// 删除评论
func (c *CommentRepository) DeleteCommentById(tx *gorm.DB, commentID string) err

    var comment model.Comment
    result := tx.Table("comments").Delete(&comment, commentID)
    if result.Error != nil {
        tx.Rollback()
        return result.Error
    }
    return nil
}
```

获取评论列表

用于通过视频ID从数据库中查找评论数据。

```
// 通过视频ID查找评论
func (c *CommentRepository) GetCommentsByVideoID(videoID int64) ([]model.Comment
    var comments []model.Comment
    result := c.DB.Table("comments").Where("video_id = ?", videoID).Find(&commen
    if result.Error != nil {
        return nil, result.Error
    }
    return comments, nil
}
```

发送消息

发送消息的实现是通过把最新的消息存到数据库，再通过查询把最新消息进行返回。

```
// 把消息存到数据库
func (messageRepo *MessageRepository) CreateMessage(tx *gorm.DB, msg *model.Mess
```

```
    if err := tx.Create(msg).Error; err != nil {
        return err
    }
    return nil
}
```

获取消息列表

通过 `preMsgTime` 实现查询 `preMsgTime` 之后的消息并返回，即返回用户未接收到的消息。能够实现历史消息的获取和在聊天界面的实时获取最新数据。

```
func (messageRepo *MessageRepository) GetMessages(tx *gorm.DB, userIdA, userIdB,
    var messages []model.Message
    // 查询
    result := tx.Table("messages").Where("((to_user_id = ? and from_user_id = ?)
        userIdA, userIdB, userIdB, userIdA, preMsgTime).Order("create_time ASC")
    if result.Error != nil {
        // 处理查询错误
        return nil, result.Error
    }
    return messages, nil
}
```

喜欢、发布模块和视频流实现

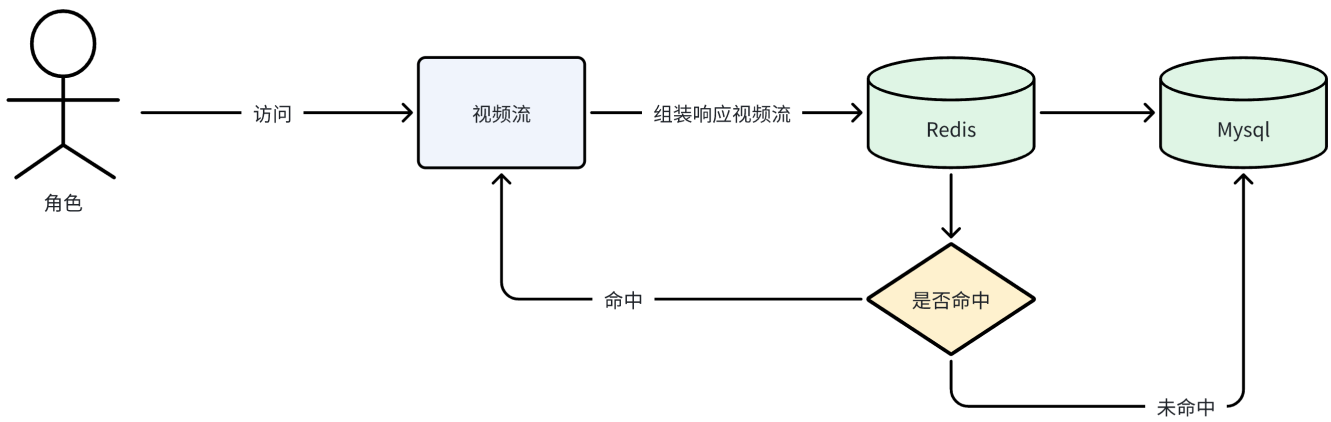
接口设计

```
//video接口设计
type VideoRepositoy interface { //Repositoy层
    Save(ctx context.Context, tx *gorm.DB, video *model.Video) error

    GetVideosByUserID(ctx context.Context, userId int64) (*[]model.Video, error)
    GetVideosByLatestTime(ctx context.Context, latestTime time.Time) (*[]model.V
    GetVideoListByLikeIdList(ctx context.Context, likeList []int64) (*[]model.Vi

    GetCommentCountByVideoId(ctx context.Context, videoId int64) (int64, error)
}

type FeedService interface { //service层
    Feed(c context.Context, latestTime time.Time, userId int64) (*[]VideoList, t
    Publish(c context.Context, video *model.Video) error
    PublishList(c context.Context, userId int64) (*[]VideoList, error)
```

```
    GetRespVideo(ctx context.Context, videoList *[]model.Video, userId int64) (*
    ///
    GetAuthorInfoByredis(c context.Context, userId int64, authorId int64) (*mode
    GetAuthorInfoBymysql(c context.Context, userId int64, authorId int64) (*mode

    GetLikeCountByRedis(c context.Context, videoId int64) (int64, error)
    GetCommentCountByRedis(c context.Context, videoId int64) (int64, error)

    GetIslikeByRedis(c context.Context, videoId int64, userId int64) (bool, erro
    GetIsFollowByRedis(c context.Context, userId int64, authorId int64) (bool, e
}
```

```
//like接口
type LikeRepositoy interface { //Repositoy层
    FavoriteAction(ctx context.Context, tx *gorm.DB, userId int64, videoId int64
    GetlikeIdListByUserId(ctx context.Context, userId int64) ([]int64, error)
    GetLikeCountByVideoId(ctx context.Context, videoId int64) (int64, error)
    GetIslike(ctx context.Context, videoId int64, userId int64) (bool, error)
}
type LikeSerVice interface { //service层
    FavoriteAction(ctx context.Context, userId int64, videoId int64, action_type
    FavoriteList(ctx context.Context, userId int64) (*[]VideoList, error) //favc

}
```

1.视频流模块



不限制登录状态，返回按投稿时间倒序的视频列表，视频数由服务端控制，单次最多30个。

投稿时间返回视频流

```go
//repository层根据投稿时间倒序返回视频列表
func (v *videos) GetVideosByLatestTime(ctx context.Context, latestTime time.Time
    videos := make([]model.Video, 0)
    logger.Debug(latestTime, "上一次时间")
    err := v.db.Model(&model.Video{}).Order("created_at desc").Limit(30).Find(&v

    if err != nil {
        return nil, time.Now(), err

    }

    if len(videos) == 0 {
        logger.Debug("no videos")
        return &videos, time.Now(), nil
    } else {
        return &videos, videos[len(videos)-1].CreatedAt, nil

    }

}
```

在service层将视频列表组装成返回的响应类型，使用协程加快速度

```go
func (v *VideoList) GetRespVideo(ctx context.Context, videoList *[]model.Video,
    var resp []VideoList

    for _, v := range *videoList {

        var respVideo VideoList

        respVideo.Video = v

        wg := sync.WaitGroup{}
        wg.Add(4)
        go func(v *VideoList) {
            defer wg.Done()
            userInfo, err := v.GetAuthorInfoByredis(ctx, userId, v.Video.AuthorI
            if err != nil {
                logger.Debug("获取用户信息错误", err)
                return
            }
            v.User = *userInfo
        }(&respVideo)

```

```
22          go func(v *VideoList) {
23              defer wg.Done()
24              favoriteCount, err := v.GetLikeCountByRedis(ctx, v.Video.Id)
25              if err != nil {
26                  //日志
27                  logger.Debug("获取喜欢数目错误", err)
28                  return
29              }
30              v.FavoriteCount = favoriteCount
31
32          }(&respVideo)
33
34          go func(v *VideoList) {
35              defer wg.Done()
36
37              commentCount, err := v.GetCommentCountByRedis(ctx, v.Video.Id)
38              if err != nil {
39                  //日志
40                  logger.Debug("获取评论数量错误", err)
41                  return
42
43              }
44              v.CommentCount = commentCount
45
46          }(&respVideo)
47
48          go func(v *VideoList) {
49              defer wg.Done() //用户不存在时是默认值false
50              // 用户未登录状态
51              if userId == 0 {
52                  logger.Debug("用户未登录状态")
53                  return
54              }
55              isFavorite, err := v.GetIslikeByRedis(ctx, v.Video.Id, userId)
56
57              if err != nil {
58                  //日志
59                  logger.Debug("获取是否喜欢错误", err)
60                  return
61              }
62              v.IsFavorite = isFavorite
63
64          }(&respVideo)
65
66          wg.Wait()
67
68          resp = append(resp, respVideo)
```

```
69
70      }
71      return &resp, nil
72  }
```

2.发布模块

视频实体

```
1  type Video struct {
2      Id         int64     `json:"id"`
3      AuthorId   int64     `json:"author_id"`
4      Title      string    `json:"title"`
5      PlayUrl    string    `json:"play_url"`
6      CoverUrl   string    `json:"cover_url"`
7      CreatedAt time.Time `json:"created_at"`
8      UpdatedAt time.Time `json:"updated_at"`
9  }
```

发布视频

视频投稿

登录用户选择视频上传，视频存储到本地/public文件夹下(使用uuid确保文件名唯一)，使用 ffmpeg 获取视频的第一帧作为封面

```
1      // 存储视频数据
2      videoHeader, _ := c.FormFile("data")
3      videoPath := fmt.Sprintf("public/%s-%s", uuid.New().String(),
   videoHeader.Filename)
4      if err := c.SaveUploadedFile(videoHeader, videoPath); err != nil {
5          c.JSON(http.StatusInternalServerError, resp.Response{
6              StatusCode: -1,
7              StatusMsg:  "Save file err",
8          })
9          return
10     }
11
12     // 压缩视频
13 func compressVideo(inputVideoPath string) (string, error) {
14     outputVideoPath := strings.TrimSuffix(inputVideoPath, ".mp4") + "CMP.mp4"
```

```
    command := []string{
        "-i", inputVideoPath,
        "-c:v", "libx264",
        //"-b:v", "1M", // 使用比特率代替 -crf
        "-crf", "43", //较高的CRF值会减小文件大小但可能降低视频质量
        "-y", // This option enables overwriting without asking
        outputVideoPath,
    }
    cmd := exec.Command("ffmpeg", command...)

    // 打开已存在的日志文件，如果不存在则创建
    logFile, err := os.OpenFile("logs/video.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return outputVideoPath, err
    }
    defer logFile.Close()

    cmd.Stderr = logFile // 将stderr重定向到指定的日志文件
    // cmd.Stderr = os.Stderr // 将stderr重定向到控制台以查看错误消息

    err = cmd.Run()
    if err != nil {
        logger.Debug("Error compressing video:", err)
        return outputVideoPath, err
    }

    return outputVideoPath, nil
}

// 使用 ffmpeg 获取视频的第一帧作为封面
func generateVideoCover(videoPath string) (string, error) {
    // 使用 ffmpeg 获取视频的第一帧作为封面
    coverFilename := strings.TrimSuffix(videoPath, ".mp4") + "_cover.jpg"
    command := []string{
        "-i", videoPath,
        "-ss", "00:00:01",
        "-vframes", "1",
        coverFilename,
    }
    cmd := exec.Command("ffmpeg", command...)

    // 打开已存在的日志文件，如果不存在则创建
    logFile, err := os.OpenFile("logs/video.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return "", err
```

```
60      }
61      defer logFile.Close()
62
63      cmd.Stderr = logFile // 将stderr重定向到指定的日志文件
64      //cmd.Stderr = os.Stderr // Redirect stderr to console for error messages
65
66      err = cmd.Run()
67      if err != nil {
68          fmt.Println("Error generating cover:", err)
69          return "", err
70      }
71      return coverFilename, nil
72  }
73
74  //保存视频
75  func (v *videos) Save(ctx context.Context, tx *gorm.DB, video *model.Video)
    error {
76      videor := video
77      return tx.Save(&videor).Error
78  }
```

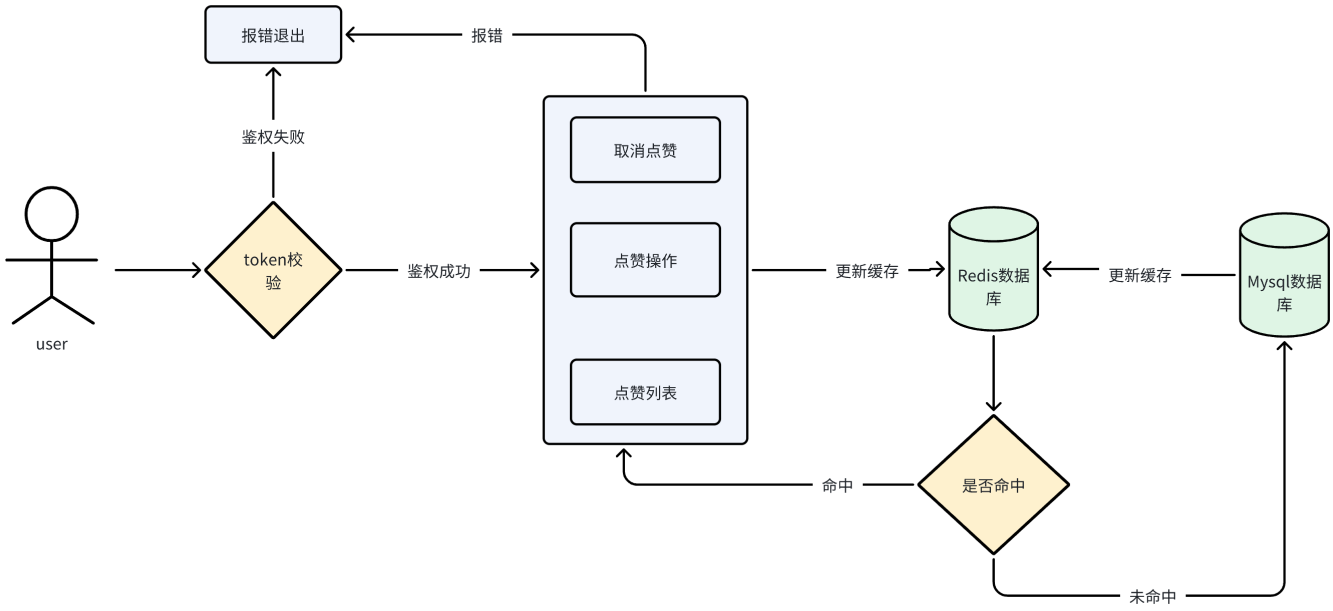
发布列表

列出用户的投稿视频

在repository层查出投稿视频，在service层进行包装（和视频流包装一样）

```
1  func (v *videos) GetVideosByUserID(ctx context.Context, userId int64) (*[]model.
2      var videos []model.Video
3      err := v.db.Where("author_id=?", userId).Find(&videos).Error
4      if err != nil {
5          return nil, err
6      }
7      return &videos, err
8  }
```

3.喜欢模块



喜欢实体

```go
type Like struct {
    Id      int64 `json:"id"`
    UserId  int64 `json:"user_id"`
    VideoId int64 `json:"video_id"`
    Liked   bool  `json:"liked"`
}
```

点赞/取消点赞

登录用户对视频的点赞和取消点赞操作

因为点赞/取消点赞 操作涉及like表和user表，所以使用事务来保证数据的一致性

```go
likeRepositoy := repository.NewLikes()
//以执行点赞操作为例
if action_type == 1 {

        logger.Debug("执行点赞操作")
        //在Redis中记录用户点赞状态
        err := common.RedisA.Set(ctx, fmt.Sprintf("isLike:%d:%d", videoId, userI
        if err != nil {
            return err
```

```
        }
        //刷新redis中的likeCount
        err = common.RedisA.Incr(ctx, fmt.Sprintf("likeCount:%d", videoId)).Err(
        if err != nil {
            logger.Debug(err)
            return err
        }

        //使用事务
        db := common.GetDB()
        err = db.Transaction(func(tx *gorm.DB) error {
            // 在MySQL中保存用户点赞记录
            if err := likeRepositoy.FavoriteAction(ctx, tx, userId, videoId, tru
                return err
            }
            //刷新用户信息
            if err := repository.NewUserRepository().UpdateUserInfo(tx, userId,
                return err
            }

            return nil

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

点赞列表

登录用户的所有点赞视频

首先根据用户id获取点赞的视频列表，然后包装成响应视频格式（和视频流包装一样）

```
func (l *likeSerVice) FavoriteList(ctx context.Context, userId int64) (*[]VideoL

    //错误处理
    likeIdList, err := repository.NewLikes().GetlikeIdListByUserId(ctx, userId)
    if err != nil {
        logger.Debug("GetlikeIdListByUserId")
        return nil, err
    }
    videoList, err := repository.NewFeed().GetVideoListByLikeIdList(ctx, likeIdL
    if err != nil {
```

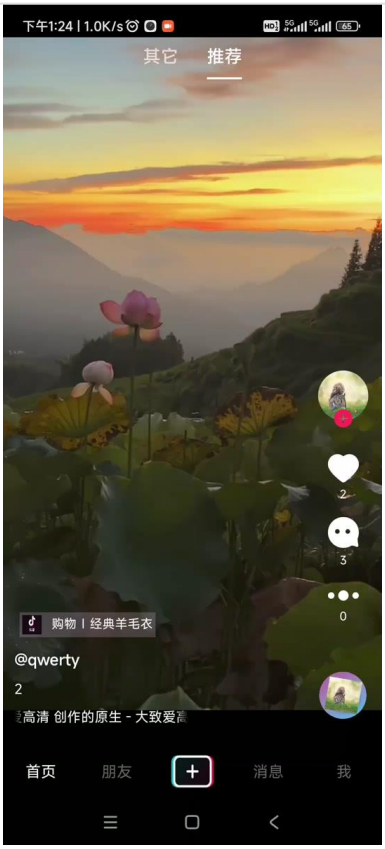
```
11          logger.Debug("GetVideoListByLikeIdList")
12          return nil, err
13      }
14      resp, err := feedService.GetRespVideo(ctx, videoList, userId)
15      if err != nil {
16          logger.Debug("GetRespVideo")
17          return nil, err
18      }
19
20      return resp, nil
21
22  }
```

四、测试结果

功能测试

性能测试

五、Demo 演示视频 （必填）



六、项目总结与反思

1. 目前仍存在的问题
    a. 项目的并发量不够高，在遇到高并发的情境下，会导致系统的崩溃。
    b. 视频播放不够顺畅
2. 已识别出的优化项
    a. 社交模块没有用到 Redis 做缓存，会增加数据库的查询负担，特别是考虑到关注关系可能并不会急剧变化，而获取列表又多次用到了表的 Join，如何对查询进行缓存、索引优化是此次项目还需要进行改进的。
3. 架构演进的可能性
    a. 项目目前采用单体架构，之后可以进一步将功能进行拆分，采用微服务架构进行开发，实现功能的解耦合，并使用消息队列增加系统吞吐量。

4. 项目过程中的反思和总结
    a. 在团队开发过程中，如何使用 Git 进行协同，定义完整规范的 Git 提交规则在一开始时很茫然，本次开发过程中也因此遇到过版本的冲突导致的运行问题，最后还是通过版本的回退进行的解决，所以熟练掌握 Git 的原理是我们在日后进行团队协作必不可少的技能。
    b. 好的项目结构可以让开发效率事半功倍，本次项目在一开始就确定了项目的主要结构和技术选型，在后面开发的过程中也比较顺利，没有出现较大的改动导致的协作困难。
    c. 通过本次项目的开发协作，我们掌握了 Gin 和 GORM 等框架的使用，对 Git 分支协作、Redis缓存等技术有了新的认识，来自字节开发团队的老师们的课程质量也非常高，对于我们的帮助非常大，期望字节青训营可以越办越好。

七、其他补充资料（选填）