抖音汇报文档

任务划分

任务	姓名
（user）用户登录；注册；用户信息	张逸峥
（video）视频投稿；发布列表；视频流；	孙艳红
（favourite）点赞操作；点赞列表	赵心治
（comment）评论操作；评论列表	雷焌槟
（relation）关注/取消；用户关注列表；用户粉丝列表	张运哲

技术栈

接口描述语言：thrift

RPC框架：kitex

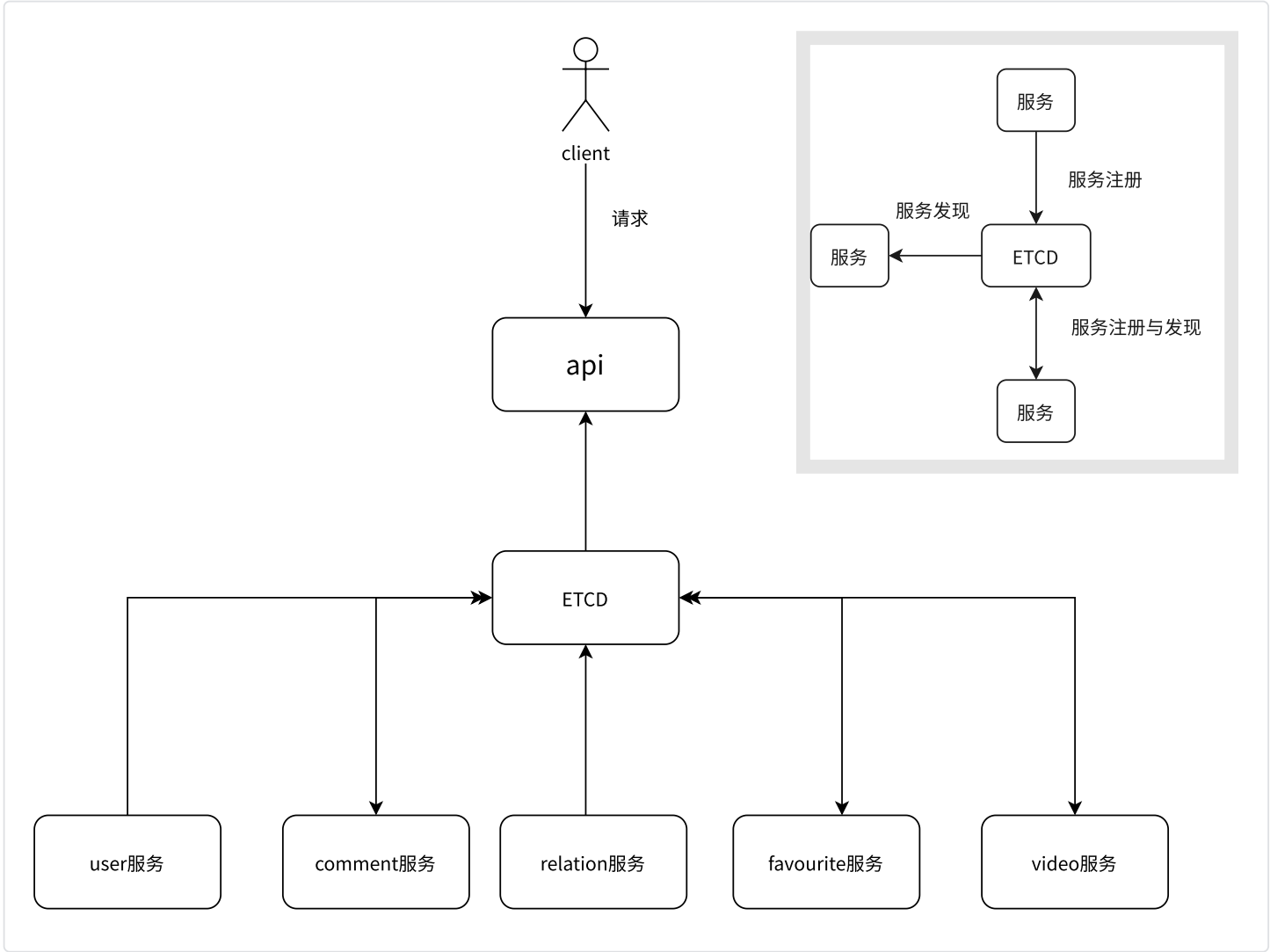
web框架：gin

中间缓存：redis，消息队列kafka，手写cache

文件加密：bcrypt

封面生成：GoCV

整体结构


client
请求
api
ETCD
user服务
comment服务
relation服务
favourite服务
video服务
服务
服务注册
服务发现
服务
ETCD
服务注册与发现
服务

数据库表

user

```sql
CREATE TABLE `auth` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `app_key` varchar(20) DEFAULT '' COMMENT 'Key',
    `app_secret` varchar(50) DEFAULT '' COMMENT 'Secret',
    # 此处请写入公共字段
    PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='认证管理';

CREATE TABLE users (
    id BIGINT NOT NULL AUTO_INCREMENT COMMENT '用户ID',
    u_name VARCHAR(30) NOT NULL COMMENT '用户名字',
    passwd VARCHAR(18) NOT NULL COMMENT '用户密码',
    # nickname      VARCHAR(30)   DEFAULT '用户'+id      COMMENT '昵称',
    follow_count BIGINT unsigned DEFAULT '0' COMMENT '关注数',
    fans_count BIGINT unsigned DEFAULT '0' COMMENT '粉丝数',
    PRIMARY KEY (id),
    index index_name(u_name(11))
) ENGINE = InnoDB;
```

favourite

```sql
CREATE TABLE favorite (
    u_id BIGINT NOT NULL COMMENT '用户ID',
    v_id BIGINT NOT NULL COMMENT '视频ID',
    PRIMARY KEY (u_id, v_id)
) ENGINE = InnoDB;
```

video

```sql
CREATE TABLE videos
(
  id              BIGINT        NOT NULL  AUTO_INCREMENT  ,
  u_id            BIGINT        NOT NULL  COMMENT '用户ID',
  play_url        VARCHAR(255)  NOT NULL  COMMENT '视频地址',
  cover_url       VARCHAR(255)  NOT NULL  COMMENT '封面地址',
  title           VARCHAR(255)  NOT NULL  COMMENT '视频标题',
  favorite_count  BIGINT        unsigned DEFAULT '0'  COMMENT '点赞数',
  comment_count   BIGINT        unsigned DEFAULT '0'  COMMENT '评论数',
  create_time     BIGINT        NOT NULL  COMMENT '创建时间',
  PRIMARY KEY (id),
  index index_uid(u_id),
  index index_createTime(create_time)
) ENGINE=InnoDB;
```

relation

```
CREATE TABLE relation
(
  follower1       BIGINT  NOT NULL   COMMENT '用户A',
  follower2       BIGINT  NOT NULL   COMMENT '用户B',
  tag   INT NOT NULL COMMENT '0:标记删除,互相不关注 1:A关注B 2:B关注A 3:相互关注',
  PRIMARY KEY (follower1,follower2),
  index index_follower1_tag(follower1,tag),
  index index_follower2_tag(follower2,tag)
)ENGINE=InnoDB;
```

comment

```
CREATE TABLE comment
(
  id                BIGINT  NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  v_id              BIGINT  NOT NULL COMMENT '视频ID',
  comment_number    BIGINT  NOT NULL COMMENT '评论数量',
  create_time       BIGINT  NOT NULL COMMENT '创建时间',
  PRIMARY KEY (id),
  index undex_vid(v_id),
  index index_createTime(create_time)
) ENGINE=InnoDB;

CREATE TABLE comment
(
  id                BIGINT  NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  c_id                      BIGINT  NOT NULL COMMENT '评论ID'
  u_id              BIGINT  NOT NULL COMMENT '用户ID',
  v_id              BIGINT  NOT NULL COMMENT '视频ID',
  state             INT     NOT NULL COMMENT '状态（0-正常，1-隐藏）',
  content           TEXT    NOT NULL COMMENT '评论内容',
  create_time       BIGINT  NOT NULL COMMENT '创建时间',
  PRIMARY KEY (id),
  index undex_vid(v_id),
  index index_createTime(create_time)
) ENGINE=InnoDB;
```

各部分服务特色（基本功能均实现）

user

随机加盐密码认证（项目特色）

密码加密的原因

为了防止数据库意外泄露/破坏和出于保护用户隐私的目的, 不应该在数据空中存入明文密码.

常用做法是通过哈希算法对明文密码加密后存入数据库. 但是人们有使用便于记忆的密码习惯, 并且不同应用也往往使用相同的密码. 因此简单的加密不能应对彩虹表攻击. 仍然存在密码泄露的风险.

可以通过向密码中加盐的方式提高密码的保护等级.

"盐"是一个随机生成的字节数组, 盐与序列化的密码相加后得到加盐密码, 增加了密码的随机性, 再对加盐密码去哈希后存入数据库.

实现

本项目中, 基于 `"golang.org/x/crypto/bcrypt"` 实现密码加盐加密.

- `bcrypt.GenerateFromPassword(password, cost)` 函数实现密码加盐后哈希;
- `bcrypt.CompareHashAndPassword(hashedPassword, password)` ` 函数实现加密密码验证.

密码串解析

```
$2a$10$ESkb/bwSyISLgq1bOH0C2utXdb.hcH9oBQD1hUnfDOzm4bMKK6EX2
$ 为分隔符
2a bcrypt加密版本号
10 Cost值
ESkb/bwSyISLgq1bOH0C2utXdb 盐
hcH9oBQD1hUnfDOzm4bMKK6EX2 密码密文

```

源码

```go
// GenerateFromPassword returns the bcrypt hash of the password at the given
// cost. If the cost given is less than MinCost, the cost will be set to
// DefaultCost, instead. Use CompareHashAndPassword, as defined in this package,
// to compare the returned hashed password with its cleartext version.
func GenerateFromPassword(password []byte, cost int) ([]byte, error) {
        p, err := newFromPassword(password, cost)
        if err != nil {
                return nil, err
        }
        return p.Hash(), nil
}
// CompareHashAndPassword compares a bcrypt hashed password with its possible
// plaintext equivalent. Returns nil on success, or an error on failure.
func CompareHashAndPassword(hashedPassword, password []byte) error {
        p, err := newFromHash(hashedPassword)
        if err != nil {
                return err
        }
        otherHash, err := bcrypt(password, p.cost, p.salt)
        if err != nil {
                return err
        }
        otherP := &hashed{otherHash, p.salt, p.cost, p.major, p.minor}
        if subtle.ConstantTimeCompare(p.Hash(), otherP.Hash()) == 1 {
                return nil
        }
        return ErrMismatchedHashAndPassword
}

```

relation

数据库建表思路（项目特色）

方案一

数据库表的结构：（follower1，follower2）

如果A关注B，即插入A B

如果B关注A，即插入B A

方案一存在的并发问题

业务需求：A，B两个用户，如果相互关注，则赋予一些权限（比如将A，B插入friend表），可以互相发消息；否则只是单向关注关系。

session1(A关注B)	session1(B关注A)

begin;
select * from 'relation'
where follower1= B and follower2= A;
返回空	

	begin;
	select * from 'relation'
	where follower1= A and follower2= B;
	返回空

	insert into 'relation'(follower1,follower2) values(B,A);
insert into 'relation'(follower1,follower2)values(A,B);	
commit;	
	commit;

由于一开始A和B之间没有关注关系，所以两个事务里面的select语句查出来的结果都是空。因此，session 1的逻辑就是“既然B没有关注A，那就只插入一个单向关注关系”。session 2也同样是这个逻辑。这个结果对业务来说就是bug了。因为在业务设定里面，这两个逻辑都执行完成以后，是应该赋予一些权限的。方案一第1步即使使用了排他锁也不行，因为记录不存在，行锁无法生效。

方案二（我们的思路）

数据库表结构：（follower1，follower2，tag）

保证follower1严格小于follower2

即A关注B，若A>B，则插入B A 2

若A< B,则插入A B 1

tag可取值0，1，2，3

值是0的时候，表示互相没关注（此时，删除此行）

值是1的时候，表示follower1关注 follower2;

值是2的时候，表示follower2关注 follower1;

值是3的时候，表示互相关注。

如何解决存在的并发问题

如果A<B，就执行下面的逻辑：

```
begin; /*启动事务*/
insert into `relation`
(follower1, follower2, tag)
values(A, B, 1) on duplicate key
update tag = tag| 1;

select tag from `relation`
where follower1=A and follower2=B;
/*代码中判断返回的 tag,
  如果是1，事务结束，执行 commit。
  如果是3，则执行赋予权力的操作。
  */

commit;
```

如果A>B，则执行下面的逻辑：

```
begin; /*启动事务*/
insert into  `relation`
(follower1, follower2, tag)
 values(B, A, 2) on duplicate key
 update tag = tag| 2;

select tag from `relation`
where follower1=B and follower2=A;
/*代码中判断返回的 tag,
  如果是2，事务结束，执行 commit
  如果是3，则执行赋予权力的操作。
*/
commit;
```

这个设计里，让“relation”表里的数据保证follower1< follower2，这样不论是A关注B，还是B关注A，在操作“relation”表的时候，如果反向的关系已经存在，就会出现行锁冲突。然后，insert … on duplicate语句，确保了在事务内部，执行了这个SQL语句后，就强行占住了这个行锁，之后的select 判断tag这个逻辑时就确保了是在行锁保护下的读操作。

方案二（我们的思路）的优点

1. 减少了数据库的冗余，存储量减少了将近一半
2. 解决了上诉的并发问题

手写缓存（MiniCache）的设计（项目特色）

由于自己的服务器配置差且只有一台，使用redis的话服务上线需要较高的运行成本，不太好管理。所以，relation服务和vedio服务自己实现了一套单机缓存MiniCache。

MiniCache，拥有懒清理和哨兵清理两种清理机制，内存淘汰策略使用W-TinyLFU方法。

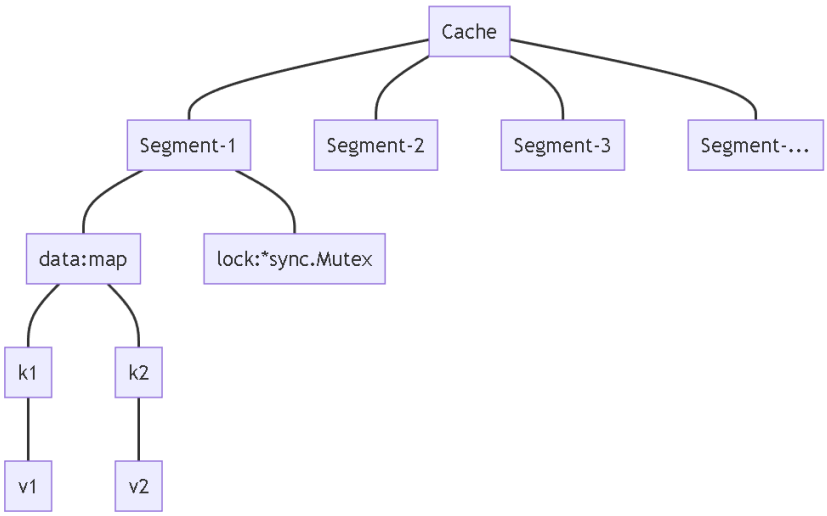
MiniCache功能特性

- 引入 **option function** 模式，可定制化各种操作的过程
- 引入**分片机制**，以此来减少锁的粒度，提高并发
- 拥有**懒清理和哨兵清理**两种清理机制
- 手写 **singleflight 机制**，减少缓存穿透的伤害
- 手写**布隆过滤器**记录全量数据，不存在直接返回，减少对DB的访问压力（单元测试通过，待加入relation服务）
- 支持**W-TinyLFU内存淘汰算法**，解决LRU热点数据命中率不高以及LFU难以应对突发流量，可能存在旧数据长期不被淘汰的问题
- 支持使用**随机过期时间**，减少缓存失效的伤害

目前支持带过期时间的key支持懒清理和哨兵清理两种清理机制，当内存到达分段上限后，随机淘汰；不带过期时间的key支持W-TinyLFU内存淘汰算法；将来会支持带过期时间的key支持W-TinyLFU内存淘汰算法（待加入MiniCache）

MiniCache的大致架构

第一层是缓存容器（Cache）。缓存容器下属多个分片（Segment）,使用自定义的哈希算法，将键值对分配到各个段中。段采用内建的map数据结构进行二次哈希，并保存到map数据结构中。当写入操作发生时，只对该数据所属的段加锁，而不是对整个缓存容器加锁，这样可以提升并发访问的性能。

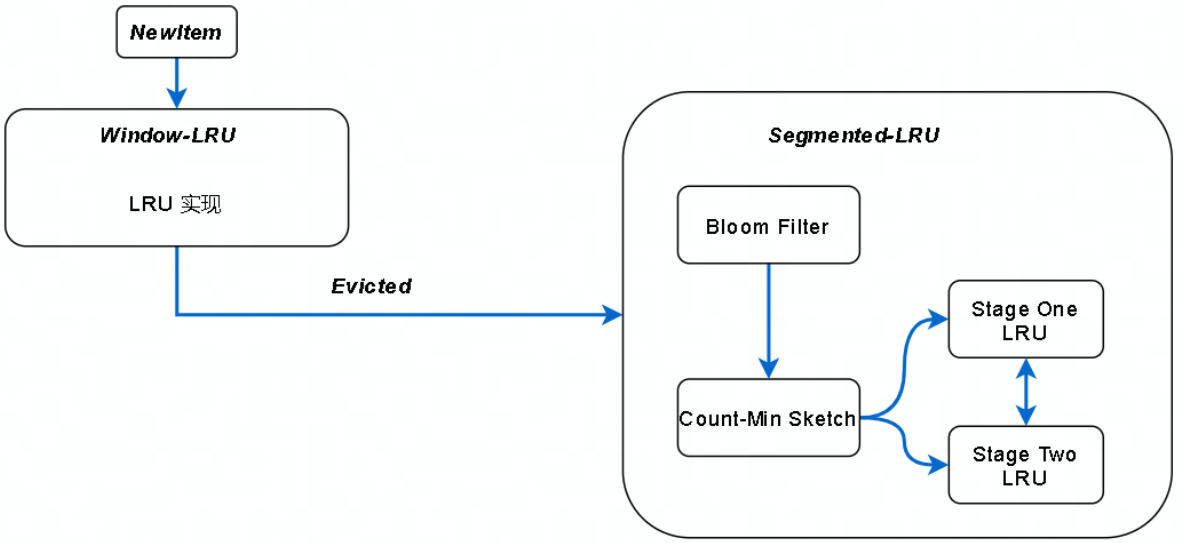


W-TinyLFU方法

主缓存（main cache）使用 SLRU 逐出策略和 TinyLFU 接纳策略，而窗口缓存（window cache）采用 LRU 逐出策略而没有任何接纳策略。

主缓存根据 SLRU 策略静态划分为 A1 和 A2 两个区域，80%的空间分配给热门项目（A2），并从 20%的非热门项目（A1）中挑选 victim。所有请求的 key 都会被允许进入窗口缓存，而窗口缓存的 victim 则有机会被允许进入主缓存。如果被接受，则 W-TinyLFU 的 victim 是主缓存的 victim，否则是窗口缓存的 victim。

窗口缓存的大小初始为总缓存大小的 1%，主缓存的大小为 99%。



W-TinyLFU淘汰算法的优势

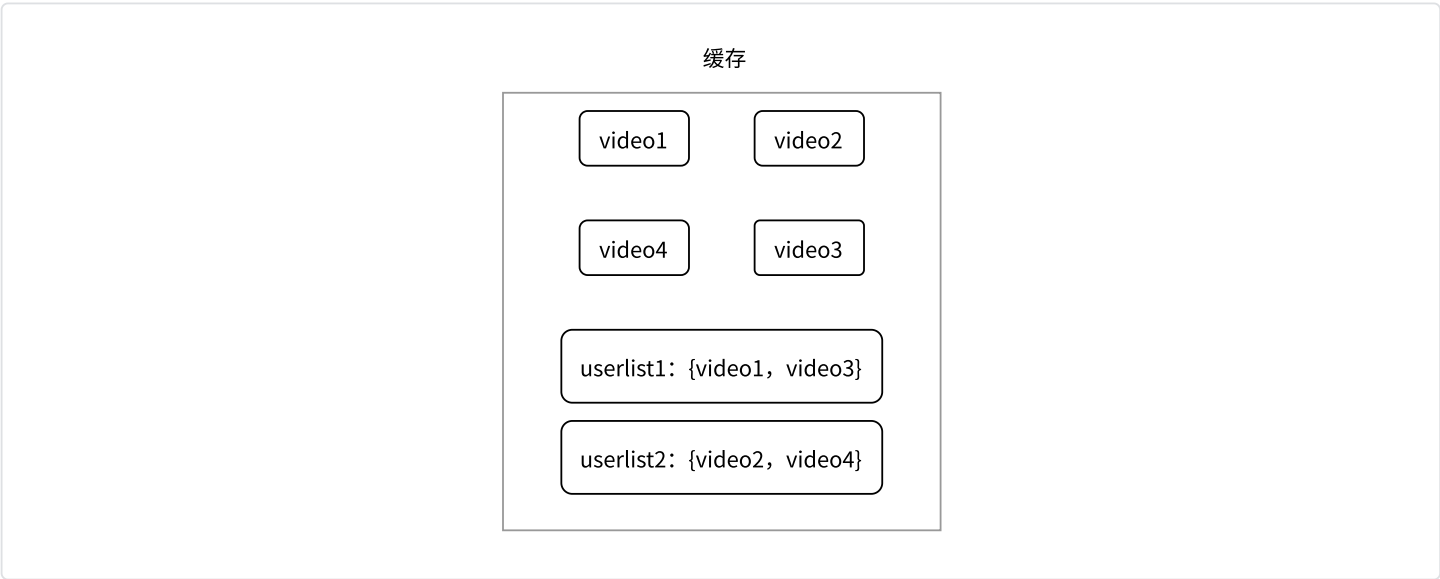
1. 针对只访问一次的数据，在window-LRU里很快就淘汰了，不占用缓存空间
2. 可以应对突发的流量
3. 真正的热点数据，可以进入Stage Two LRU中，并经过保鲜机制存活下来

Video

缓存的设计

缓存采用了relation部分的miniCache。

在video部分功能中 需要根据用户id获取发布列表，根据时间排序获取feed流。因此缓存中需要存储视频信息，及用户发布的视频信息列表。**考虑到热点视频的作者就是热点用户的可能性。并将视频信息与用户列表解耦，减少视频信息在缓存中重复** 。 缓存中记录的用户发布列表，里面仅含是视频id而不包含视频的详细信息。缓存内容如下图。



查询

在查询用户发布列表时，如果用户发布列表缓存命中，则根据列表里的视频Id查询缓存里的详细信息，如果视频信息没命中则查询数据库。如果用户发布列表没命中，则根据用户id查询数据库，返回视频id列表（覆盖索引）。

在查询feed流时，首先查询数据库，按时间顺序返回视频ID列表（这里用到了覆盖索引，将视频创建时间建了个索引）然后再根据视频Id查询缓存，，如果没有命中则根据视频Id查询数据库。查询的视频流中多是最新的热点数据，所以这里缓存的作用是减少回表次数。

更新

用户发布新视频，这里采用先更新数据库，然后删除缓存的策略。

favourite

comment

通信 RPC 接口

```
CreateCommentResponse CreateComment(1:CreateCommentRequest req) // 创建评论
DeleteCommentResponse DeleteComment(1:DeleteCommentRequest req) // 删除评论
QueryCommentsResponse QueryComments(1:QueryCommentsRequest req) // 查询评论列表
QueryCommentNumberResponse QueryCommentNumber(1:QueryCommentNumberRequest req // 查询评论数
```

支持功能

发布评论：

仅支持一层回复，即视频下回复，不支持楼中楼回复

仅支持文字编码回复

不支持修改评论

读取评论：按照时间排序（默认分页，每页最多返回10000条评论）

删除评论：评论本人删除

敏感词、评论审核（文字内容安全）：腾讯云安全：1000QPS

拓展需求：（并没有完成）

如果拓展成为评论框架或者评论中台，需要添加人工管理相关功能，包括数据检索、导出、人工审核和删除等功能

评论点赞、按照赞数排序读取评论

实现细节

redis 缓存实现

1. 简单的在查询时如果缓存未命中，发送一个消息，收到后在 redis 中创建一个12小时的缓存。
2. 如果评论数据发送更新，则数据库更新后发送信息更新缓存。
3. 关于一致性：在我对这个业务的理解中，每个人发评论对于自身评论是更加关注的，而对于同时间发送评论人的评论并不关心，所以在客户端方面实现一个服务端确认后增加自己评论数目的功能就已经满足了需求，而后端的主要任务就是处理大量的数据并保证缓存的同步，同时这个同步可以慢一些，并不需要即时完成（但需要保证数据的可靠性，保证评论内容一定写入数据库之中，同时完成同步）

归并回源实现

1. 利用 go 官方的 singleflight 包实现同一时段相同请求的合并，互联网黑话——归并回源，具体实现和用法参考 go 官方文档。
2. 在评论系统中，主要在查询评论列表、查询评论数量、删除评论这三个服务上引入这个特性，也即这三个服务操作本身具有幂等性（互联网黑话又一，其本质就是多次执行同一操作结果和后台数据不变）

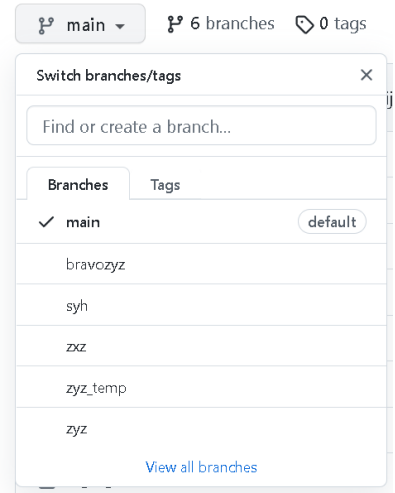
消息队列作用

1. 将可能巨大的对数据库的写流量缓冲下来，减少数据库压力。
2. 将数据库写操作独立出来后，可以单线程消费，以解决我这个简单（垃圾）的redis缓存的一致性问题。
3. 可以在写数据和缓存层面上实现对存储层的解耦（简单理解就是可以随时更换下面的存储引擎，不需要对服务层说明，服务层只需要按照一定格式向消息队列发送消息就可以了）。（这个特性在存储层实现读写分离后会更加明显，收益更高）

关于日志选用 zap 而不顺手使用 klog 的原因

1. zap 及其推荐第三方库对于文件日志的支持更加全面，通过简单配置就可以实现了日志文件的大小限制、超大小后自动新建文件、日志存储时间、超时自动删除、分级日志写入等管理功能，而 klog 没有这样功能抑或实现较为麻烦。
2. zap 对于日志格式的自定义个人比较喜欢，也可以比较简单的使用和易读（代码的可读性）。（个人原因，非功能考虑）

团队协作



每个人新建自己的分支，使用git命令团结合作。

以下是我们经常用到的命令总结：

https://juejin.cn/post/7106125483506401287

常用的git命令 从0到0.1 | 青训营笔记 - 掘金

这是我参与「第三届青训营 -后端场」笔记创作活动的第5篇笔记。我在大项目开发过程中git命令的简单使用

安全问题

1. GORM 使用 `database/sql` 的参数占位符来构造 SQL 语句，这可以自动转义参数，**避免 SQL 注入数据**

2. token的验证：基于JWT实现API访问控制，token验证的过程实现为一个中间件，在调用功能接口之前进行token验证。用户正确注册和登录后，由User服务返回给用户一个token。在token的claim中规定了token的有效时间，只有在有效期内才能正常访问功能。token过期后，用户需要通过重新登录获取新的token。**解决了用户越权风险**。
3. 敏感词检测：基于腾讯云内容安全接口，**避免用户传播不良信息**。
4. 数据库敏感信息的保护（**如密码的保护**）：随机加盐密码认证（详细见user的服务特色）

成果展示