

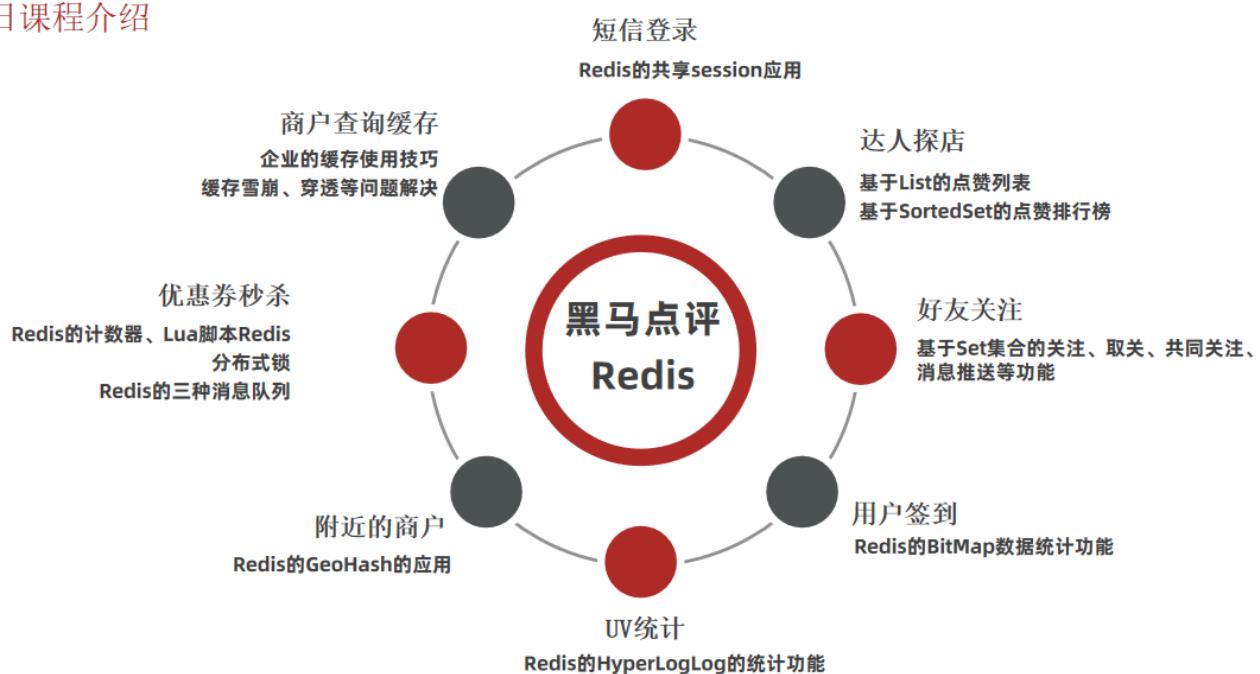
黑马点评

项目介绍

黑马点评是一个大量使用Redis的项目，该项目的功能类似大众点评。

- 短信登录：使用redis共享session来实现
- 商户查询缓存：理解缓存击穿，缓存穿透，缓存雪崩等问题
- 优惠券秒杀：Redis的计数器功能，结合Lua完成高性能的redis操作，同时学会Redis分布式锁的原理，包括Redis的三种消息队列
- 打人探店：基于List来完成点赞列表的操作，同时基于SortedSet来完成点赞的排行榜功能
- 好友关注：基于Set集合的关注、取消关注，共同关注等等功能
- 附近的商户：利用Redis的GEOHash来完成对于地理坐标的操作
- 用户签到：使用Redis的BitMap数据统计功能
- UV统计：使用Redis来完成统计功能

今日课程介绍



时长分布

- 基础篇：P1-P22 3h59m
- 实战篇：P24-P95 19h40m
 - 短信登录：24-34 2h20m (✓)
 - 商户查询：35-47 3h22m (✓)
 - 优惠券秒杀：48-77 8h27m (✓)
 - 达人探店：78-81 1h9m (✓)
 - 好友关注：82-87 1h56m (✓)
 - 附近商户：88-90 1h3m (✓)
 - 用户签到：91-95 1h12m (✓)
- 高级篇：P96-P144 9h29m
- 原理篇：P145-P175 9h38m

TODO

1. 拦截器要画个图
2. 一人一单感觉也需要

启动项目

1. 没有worksapce.xml的话，自己新建一个-->如果没有service界面

2. 刷新pom.xml文件

3. 修改application的配置文件，mysql密码和redis的host

4. 后端运行项目 [HmDianPingApplication](#)

- 报错显示： [警告：源发行版 9 需要目标发行版 9/无效的源发行版：9](#)

解决办法：https://blog.csdn.net/weixin_45716968/article/details/129436663?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-129436663-blog-121019126.235%5Ev36%5Epc_relevant_default_base3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-129436663-blog-121019126.235%5Ev36%5Epc_relevant_default_base3&utm_relevant_index=2

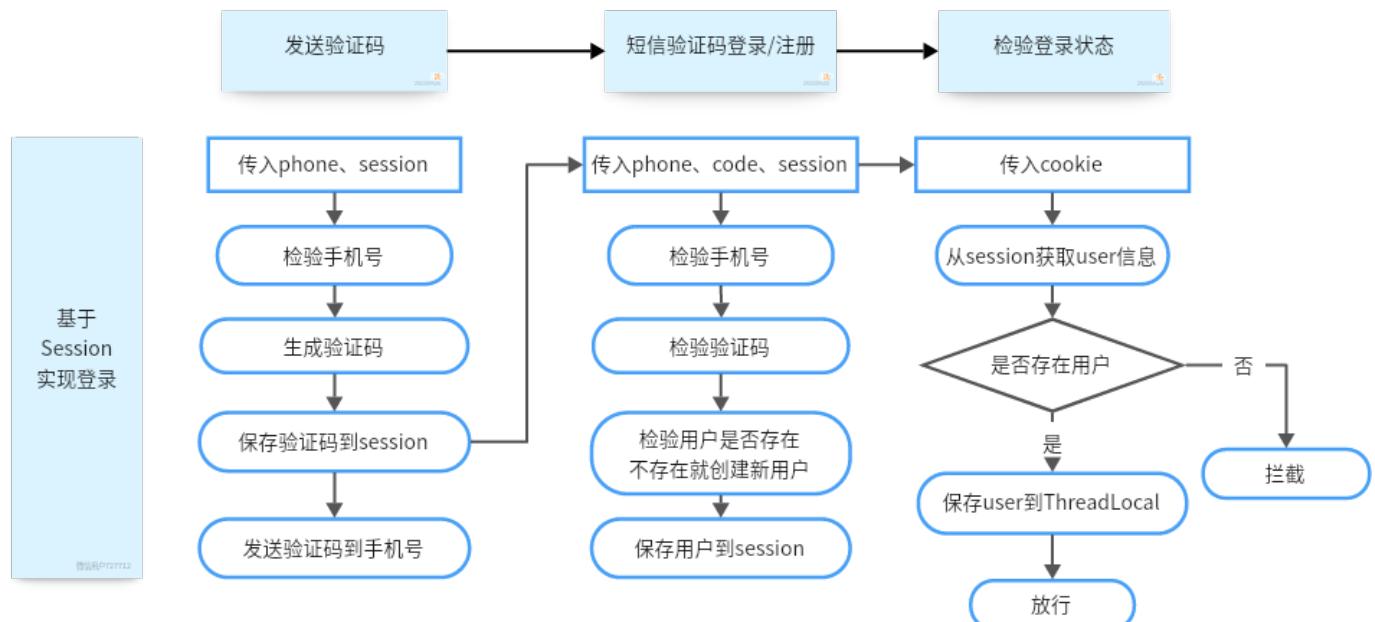
5. 打开：..\nginx-1.18.0 输入cmd，再输入 start nginx.exe （闪屏也没关系），访问<http://localhost:8080>

- 如果后台没有进程，就去conf改端口
- 有进程但是报404的错，可能是你改了端口但是访问地址没改端口
- 出错显示 [Whitelabel Error Page](#) ——>一般是后端的问题
 - 注意运行 <http://localhost:8081/shop-type/list>，看是否有json格式数据，有的话说明后端跑起来了
 - 没有的话检查一下第三步是否完成
- 前端只显示框架不显示具体数据，list接口和hot接口报错——> sos哥们你看看你后端跑了没

短信登陆

1、基于Session实现登录流程

整体业务逻辑：



发送短信验证码

代码实现：

```

@Override
public Result sendCode(String phone, HttpSession session) {
    //1.校验手机号是否合法
    if(RegexUtils.isPhoneInvalid(phone)){
        //2.若不符合，返回错误信息
        return Result.fail("手机号格式错误");
    }
    //3.若符合，生成验证码
    String code = RandomUtil.randomNumbers(6);
    //4.保存验证码到session
    session.setAttribute("code",code);
    //5.发送验证码（要调用第三方，这里不做）
    log.debug("发送短信验证码: {}",code);
    return Result.ok();
}
  
```

短信登录、注册功能

代码实现：

```

@Override
public Result login(LoginFormDTO loginForm, HttpSession session) {
    String phone = loginForm.getPhone();
    String code = loginForm.getCode();
    //校验手机号
    if(RegexUtils.isPhoneInvalid(phone)){
        return Result.fail("手机号格式错误");
    }
  
```

```

//校验验证码
Object cacheCode = session.getAttribute("code");
if(cacheCode==null||!code.equals(cacheCode.toString())){
    return Result.fail("验证码错误");
}
//查数据库
LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
queryWrapper.eq(StringUtils.isNotBlank(phone),User::getPhone,phone);
User user = userMapper.selectOne(queryWrapper);
//判断用户是否存在，不存在则创建一个
if(user==null){
    user=createUserWithPhone(phone);
}
//脱敏，剔除user中的敏感信息，保存一个UserDTO到session中
session.setAttribute("user", BeanUtil.copyProperties(user, UserDTO.class));
return Result.ok();
}

```

实现拦截器

拦截器代码：LoginInterceptor

```

public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        //1.获取session
        HttpSession session = request.getSession();
        //2.获取session中的用户
        Object user = session.getAttribute("user");
        //3.判断用户是否存在
        if(user == null){
            //4.不存在，拦截，返回401状态码
            response.setStatus(401);
            return false;
        }
        //5.存在，保存用户信息到Threadlocal
        UserHolder.saveUser((User)user);
        //6.放行
        return true;
    }
}

```

让拦截器生效

```

@Configuration
public class MvcConfig implements WebMvcConfigurer {

```

```

@Resource
private StringRedisTemplate stringRedisTemplate;

@Override
public void addInterceptors(InterceptorRegistry registry) {
    // 登录拦截器
    registry.addInterceptor(new LoginInterceptor())
        .excludePathPatterns(
            "/shop/**",
            "/voucher/**",
            "/shop-type/**",
            "/upload/**",
            "/blog/hot",
            "/user/code",
            "/user/login"
        ).order(1);
    // token刷新的拦截器
    registry.addInterceptor(new
RefreshTokenInterceptor(stringRedisTemplate)).addPathPatterns("/**").order(0);
}
}

```

ThrealLocal工具类

1. 在拦截器的时候就把当前登录用户 save 一个 ThreadLocal 对象里面，每个线程都有自己独立的 `ThreadLocal` 值，因此它只会返回当前线程中的 `UserDTO`，而不会混淆不同线程的数据。之后要用到的时候再调用 `getUser` 获取当前登录用户
2. 【`ThreadLocal` 是一个线程级别的变量，它允许你在每个线程中存储和访问不同的数据。在你的代码中，你使用 `ThreadLocal` 存储了 `UserDTO` 对象，使得每个线程都可以独立地访问自己的 `UserDTO` 实例。】

```

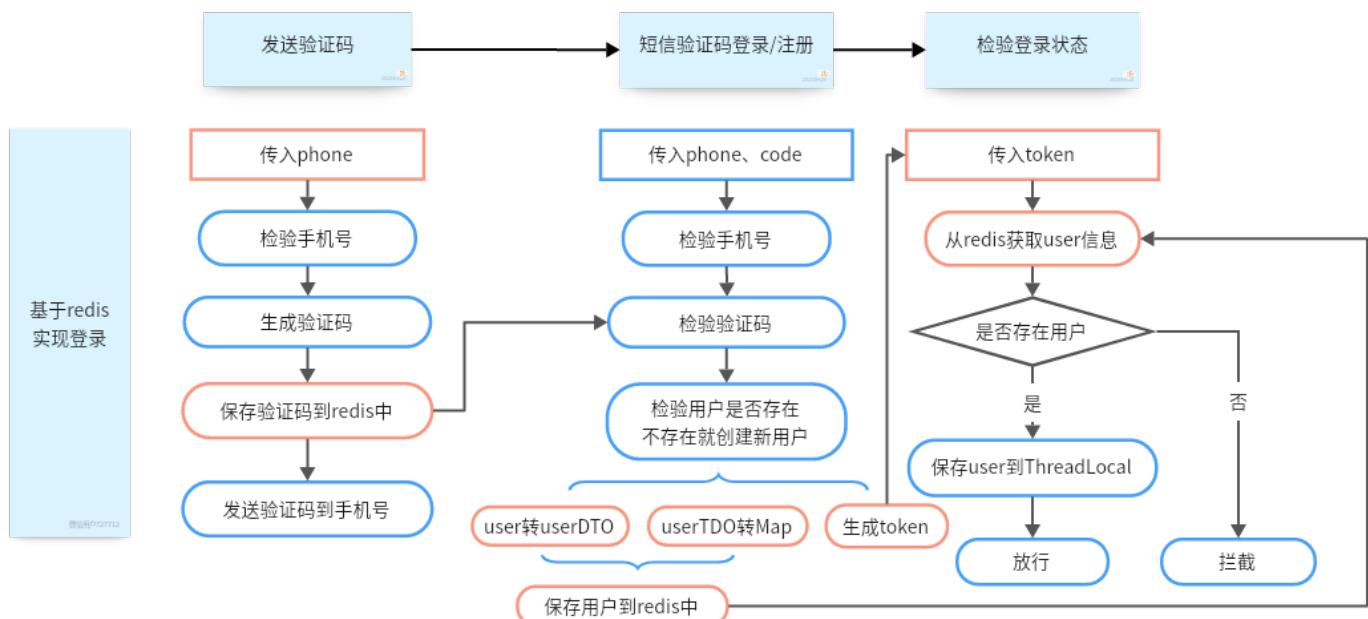
public class UserHolder {
    private static final ThreadLocal<UserDTO> tl = new ThreadLocal<>();

    public static void saveUser(UserDTO user){tl.set(user);}
    public static UserDTO getUser(){return tl.get();}
    public static void removeUser(){tl.remove();}
}

```

2、基于Redis实现共享session登录流程

整体业务逻辑：（橙色为修改部分）



发送短信验证码

代码实现：UserServiceimpl——sendCode

```

@Override
public Result sendCode(String phone) {
//    1. 检验手机号
    if (RegexUtils.isPhoneInvalid(phone)) {
//        这里抛出异常和return fail有什么区别吗？ —> 有区别，抛出异常会被全局异常处理器捕获，返回fail不会
        throw new RuntimeException("手机号格式不正确");
    }
//    2. 生成验证码
    String code = RandomUtil.randomNumbers(6);
//    3. 保存验证码到session —> 保存到redis中, redis名字、值、过期时间、时间单位
//    session.setAttribute("code", code);
    stringRedisTemplate.opsForValue().set(LOGIN_CODE_KEY + phone, code,
    LOGIN_CODE_TTL, TimeUnit.MINUTES);

//    4. 发送验证码到手机
//    注意这里的log是lombok的@Slf4j注解生成的，不然只能写一个参数
    log.debug("发送验证码: {}, 到手机: {}", code, phone);
    return Result.ok();
}

```

短信登录、注册功能

代码实现：UserServiceimpl——login

```

@Override
public Result login(LoginFormDTO loginForm) {
//    1. 检验手机号 —> 因为每个请求都是单独的，使用还要再检查一次
}

```

```
String phone = loginForm.getPhone();
if (RegexUtils.isPhoneInvalid(phone)) {
    return Result.fail("手机号格式错误!");
}
// 2. 检验验证码 -- 从redis中获取
Object cacheCode = session.getAttribute("code");
String cacheCode = stringRedisTemplate.opsForValue().get(LOGIN_CODE_KEY + phone);
String code = loginForm.getCode();
if (cacheCode == null || !cacheCode.equals(code)) {
    System.out.println("cacheCode = " + cacheCode);
    return Result.fail("验证码错误");
}
// 3. 检验用户是否存在
// 法1：最简洁的用法，但是有硬编码
User user = query().eq("phone", phone).one();
// 法2：使用lambda表达式，减少硬编码
User user = this.lambdaQuery().eq(User::getPhone,
loginForm.getPhone()).one();
// 法3：复杂一点，但多了一个isNotBlank的动态查询
LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
queryWrapper.eq(StringUtils.isNotBlank(phone), User::getPhone, phone);
User user = userMapper.selectOne(queryWrapper);
// 如果不存在则创建用户
if (user == null) {
    user = createUserWithPhone(phone);
}

// 4. 保存用户到session -- 保存到token中
// session.setAttribute("user", BeanUtil.copyProperties(user,
UserDTO.class));
String token = UUID.randomUUID().toString(true);
UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
// 因为user的id是long类型的，但是StringRedisTemplate只支持String类型的key-value，因此要需要自定义map映射规将user转成map后进行hash存储
// userDTO：要转换为Map的Java对象           new HashMap<>(): 存储转换后的Map的容器
Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new HashMap<>(),
// 忽略userDTO对象中的空值属性，即那些值为null的属性不会被放入userMap
中
    CopyOptions.create().setIgnoreNullValue(true)
        // 将属性值放入userMap前，将属性值转换为其字符串表示形式
        .setFieldValueEditor((fieldName, fieldValue) ->
fieldValue.toString()));

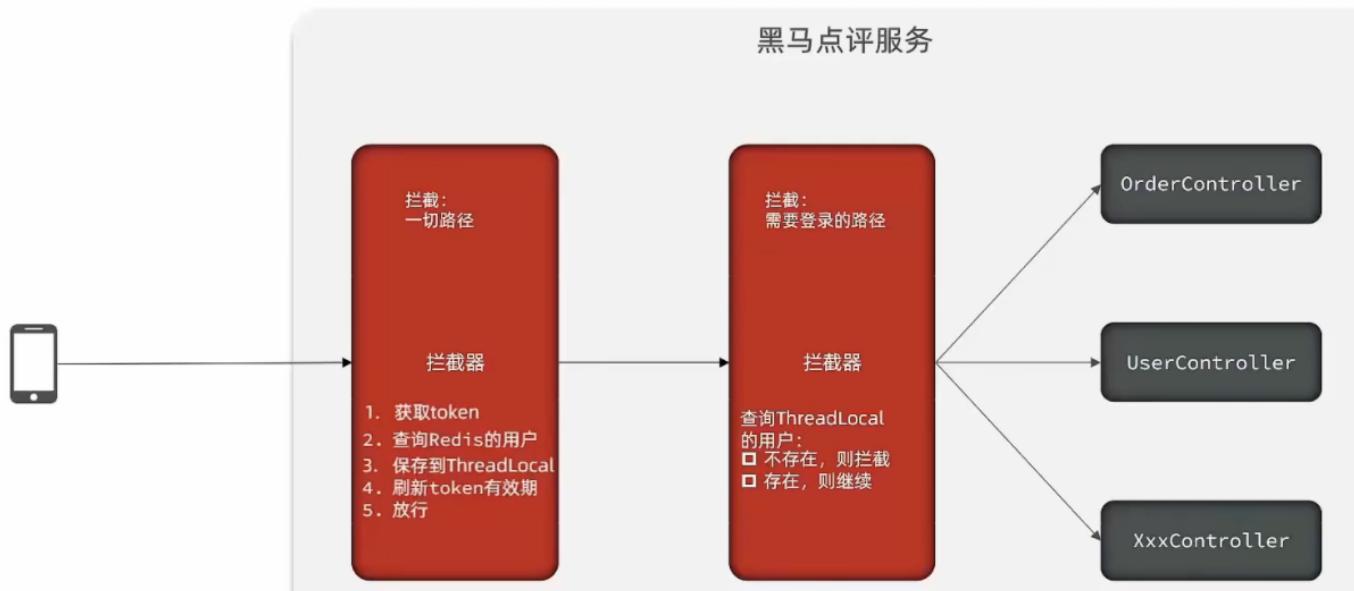
// 5. 存储
String tokenKey = LOGIN_USER_KEY + token;
stringRedisTemplate.opsForHash().putAll(tokenKey, userMap);
// 设置过期时间
stringRedisTemplate.expire(tokenKey, LOGIN_USER_TTL, TimeUnit.MINUTES);
return Result.ok(token);
}
```

优化拦截器

在这个方案中，他确实可以使用对应路径的拦截，同时刷新登录token令牌的存活时间，但是现在这个拦截器他只是拦截需要被拦截的路径，假设当前用户访问了一些不需要拦截的路径，那么这个拦截器就不会生效，所以此时令牌刷新的动作实际上就不会执行，所以这个方案他是存在问题的

业务逻辑：

登录拦截器的优化



代码实现：

RefreshTokenInterceptor

```

public class RefreshTokenInterceptor implements HandlerInterceptor {

    private StringRedisTemplate stringRedisTemplate;

    public RefreshTokenInterceptor(StringRedisTemplate stringRedisTemplate) {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        // 1. 获取请求头中的token
        String token = request.getHeader("authorization");
        if (StrUtil.isBlank(token)) {
            return true;
        }
        // 2. 基于TOKEN获取redis中的用户
        String key = LOGIN_USER_KEY + token;
        Map<Object, Object> userMap =
        stringRedisTemplate.opsForHash().entries(key);
        // 3. 判断用户是否存在
        if (userMap.isEmpty()) {
    
```

```

        return true;
    }
    // 5.将查询到的hash数据转为UserDTO
    UserDTO userDTO = BeanUtil.fillBeanWithMap(userMap, new UserDTO(), false);
    // 6.存在, 保存用户信息到 ThreadLocal
    UserHolder.saveUser(userDTO);
    // 7.刷新token有效期
    stringRedisTemplate.expire(key, LOGIN_USER_TTL, TimeUnit.MINUTES);
    // 8.放行
    return true;
}

@Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
    // 移除用户
    UserHolder.removeUser();
}
}

```

LoginInterceptor

```

public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        // 1.判断是否需要拦截 (ThreadLocal中是否有用户)
        if (UserHolder.getUser() == null) {
            // 没有, 需要拦截, 设置状态码
            response.setStatus(401);
            return false;
        }
        // 有用户, 则放行
        return true;
    }
}

```

MvcConfig: 拦截器不生效的可能原因：MvcConfig没加@Configuration注解

```

@Configuration
public class MvcConfig implements WebMvcConfigurer {

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    // 添加拦截器
    public void addInterceptors(InterceptorRegistry registry) {

```

```

// 登录拦截器
    registry.addInterceptor(new LoginInterceptor(stringRedisTemplate))
        .excludePathPatterns(
            "/shop/**",
            "/voucher/**",
            "/shop-type/**",
            "/upload/**",
            "/blog/hot",
            "/user/code",
            "/user/login",
            "/user/me"
        ).order(1);
// token刷新拦截器 order为0, 最高
    registry.addInterceptor(new RefreshTokenInterceptor(stringRedisTemplate))
        .addPathPatterns("/**").order(0);
}
}

```

3、补充

用户登出

```

@Override
public Result logout(HttpServletRequest request) {
    String token = request.getHeader("Authorization");
    if (token == null) {
        return Result.ok("尚未登录! 无法退出");
    }
    // 去掉redis的记录
    String tokenKey = LOGIN_USER_KEY + token;
    stringRedisTemplate.delete(tokenKey);
    return Result.ok();
}

```

根据手机号创建新用户

```

private User createUserWithPhone(String phone) {
    User user = new User();
    user.setPhone(phone);
    user.setNickName(USER_NICK_NAME_PREFIX + RandomUtil.randomString(7));
    this.save(user);
    return user;
}

```

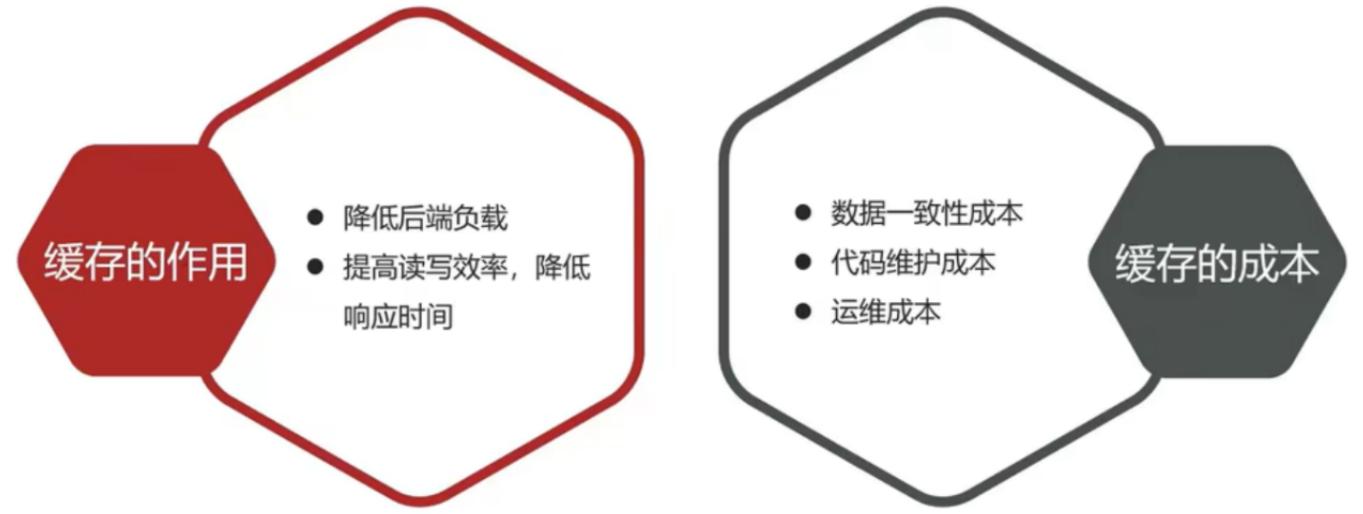
商户查询缓存

0、为什么要缓存

速度快,好用

缓存数据存储于代码中,而代码运行在内存中,内存的读写性能远高于磁盘,缓存可以大大降低**用户访问并发量带来的服务器读写压力**

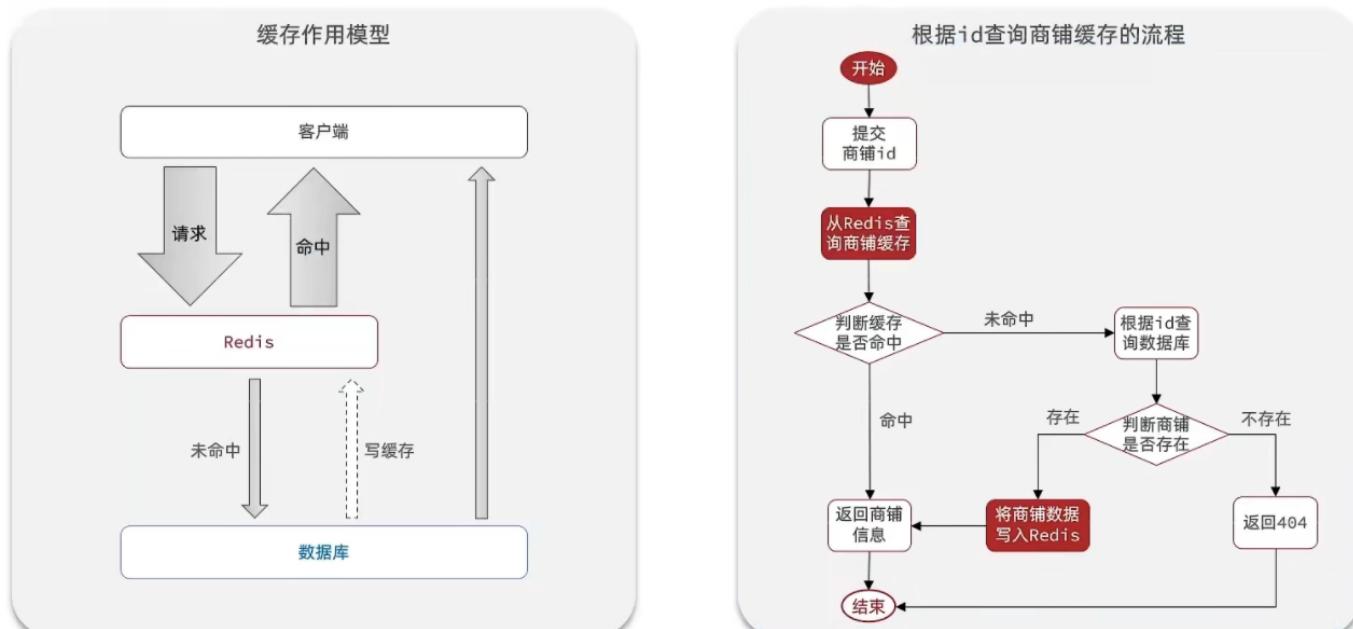
实际开发过程中,企业的数据量,少则几十万,多则几千万,这么大数据量,如果没有缓存来作为“避震器”,系统是几乎撑不住的,所以企业会大量运用到缓存技术



0.5 添加缓存到redis

业务逻辑:

添加Redis缓存



代码实现:

```

@Override
public Result queryById(Long id) {
    //查询redis, 若存在则转换成对象后返回
    String key = CACHE_SHOP_KEY + id;
    String shopJson = stringRedisTemplate.opsForValue().get(key);
}

```

```

if (StringUtils.isNotBlank(shopJson)) {
    Shop shop = JSONUtil.toBean(shopJson, Shop.class);
    return Result.ok(shop);
}
//不存在则查询数据库，然后转成以json串存入redis后，返回
Shop shop = shopMapper.selectById(id);
if(shop==null){
    return Result.fail("店铺不存在");
}

stringRedisTemplate.opsForValue()
.set(key,JSONUtil.toJsonStr(shop));
return Result.ok(shop);
}

```

1、实现数据库和缓存双写一致

(1) 缓存更新策略

	内存淘汰	超时剔除	主动更新
说明	不用自己维护，利用Redis的内存淘汰机制，当内存不足时自动淘汰部分数据。下次查询时更新缓存。	给缓存数据添加TTL时间，到期后自动删除缓存。下次查询时更新缓存。	编写业务逻辑，在修改数据库的同时，更新缓存。 
一致性	差	一般	好
维护成本	无	低	高

业务场景：

- 低一致性需求：使用内存淘汰机制。例如店铺类型的查询缓存
- 高一致性需求：主动更新，并以超时剔除作为兜底方案。例如店铺详情查询的缓存

(2) 数据库和缓存不一致采用什么方案

综合考虑使用方案一，但是方案一调用者如何处理呢？这里有几个问题

操作缓存和数据库时有三个问题需要考虑：

如果采用第一个方案，那么假设我们每次操作数据库后，都操作缓存，但是中间如果没有人查询，那么这个更新动作实际上只有最后一次生效，中间的更新动作意义并不大，我们可以把缓存删除，等待再次查询时，将缓存中的数据加载出来

- 删除缓存还是更新缓存？
 - 更新缓存：每次更新数据库都更新缓存，无效写操作较多
 - 删除缓存：更新数据库时让缓存失效，查询时再更新缓存
- 如何保证缓存与数据库的操作的同时成功或失败？

- 单体系统，将缓存与数据库操作放在一个事务
- 分布式系统，利用TCC等分布式事务方案

应该具体操作缓存还是操作数据库，我们应当是先操作数据库，再删除缓存，原因在于，如果你选择第一种方案，在两个线程并发来访问时，假设线程1先来，他先把缓存删了，此时线程2过来，他查询缓存数据并不存在，此时他写入缓存，当他写入缓存后，线程1再执行更新动作时，实际上写入的就是旧的数据，新的数据被旧数据覆盖了。

- 先操作缓存还是先操作数据库？
 - 先删除缓存，再操作数据库
 - 先操作数据库，再删除缓存

(3) 具体实现

代码实现：修改**ShopServiceImpl**的queryById和update

```
@Override
public Result queryById(Long id) {
    //查询redis，若存在则转换成对象后返回
    String key = CACHE_SHOP_KEY + id;
    String shopJson = stringRedisTemplate.opsForValue().get(key);
    if (StringUtils.isNotBlank(shopJson)) {
        Shop shop = JSONUtil.toBean(shopJson, Shop.class);
        return Result.ok(shop);
    }
    //不存在则查询数据库，然后转成以json串存入redis后，返回
    Shop shop = shopMapper.selectById(id);
    if(shop==null){
        return Result.fail("店铺不存在");
    }

    stringRedisTemplate.opsForValue()
        .set(key,JSONUtil.toJsonStr(shop),CACHE_SHOP_TTL, TimeUnit.MINUTES);
    return Result.ok(shop);
}

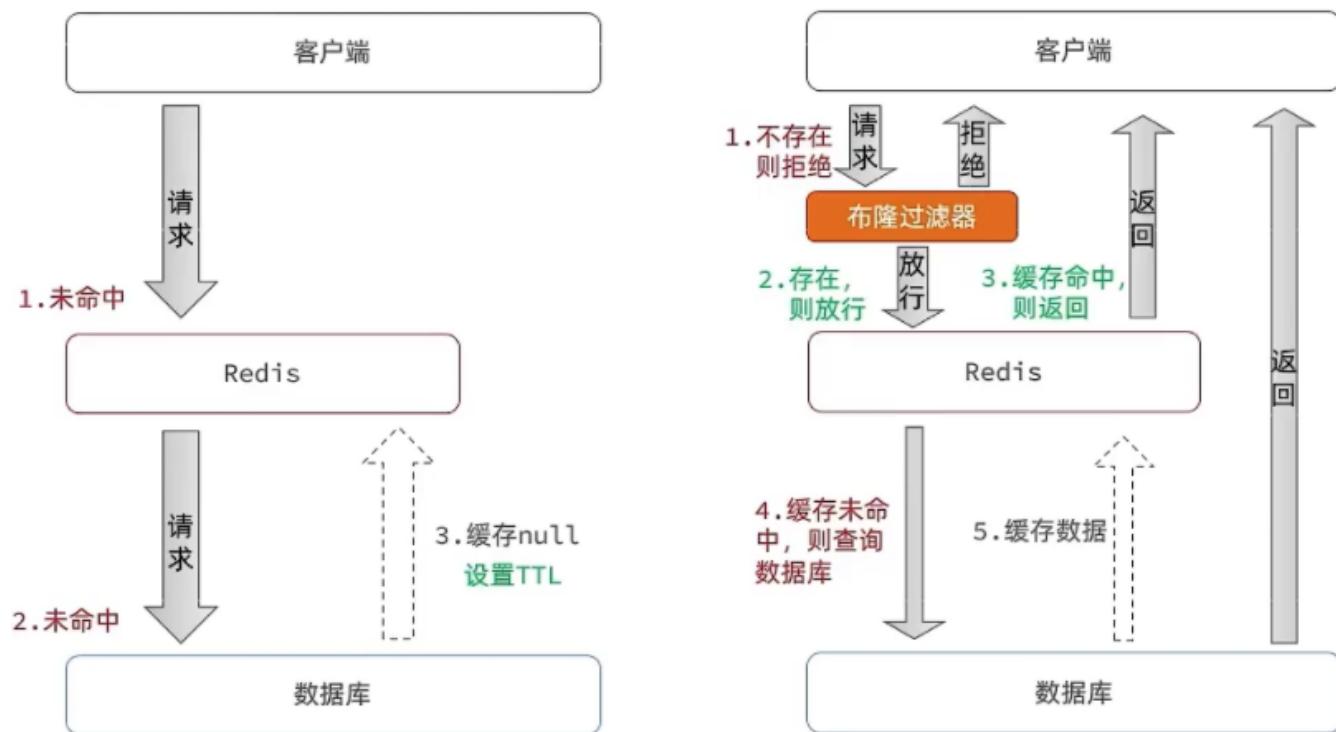
@Override
@Transactional //保证原子性
public Result update(Shop shop) {
    Long id = shop.getId();
    if(id==null){
        return Result.fail("店铺id不能为空");
    }
    //先更新数据库，再删除缓存
    shopMapper.updateById(shop);
    stringRedisTemplate.delete(CACHE_SHOP_KEY+ id);
    return Result.ok();
}
```

2、缓存穿透

缓存穿透：是指客户端请求的数据在缓存中和数据库中都不存在，这样缓存永远不会生效，这些请求都会打到数据库。

常见的解决方案有两种：

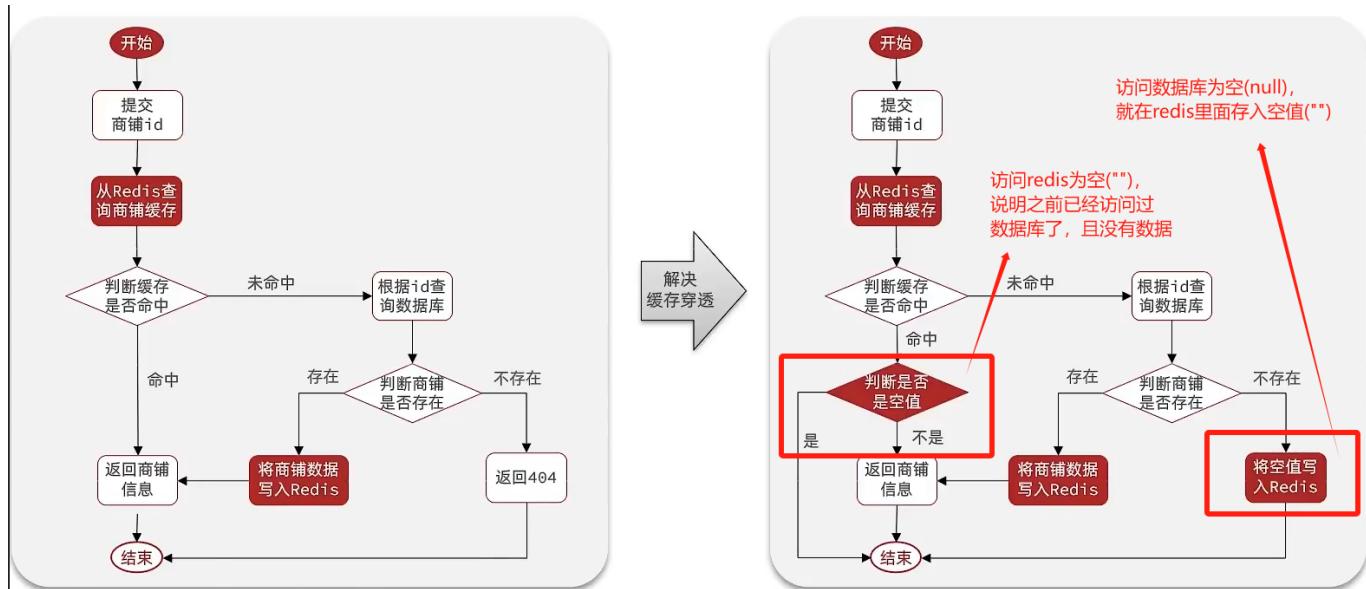
- 缓存空对象
 - 优点：实现简单，维护方便
 - 缺点：
 - 额外的内存消耗
 - 可能造成短期的不一致
- 布隆过滤
 - 优点：内存占用较少，没有多余key
 - 缺点：
 - 实现复杂
 - 存在误判可能（哈希冲突）
- 增强id的复杂度，避免被猜测id规律
- 做好数据的基础格式校验
- 加强用户权限校验
- 做好热点参数的限流



业务逻辑：

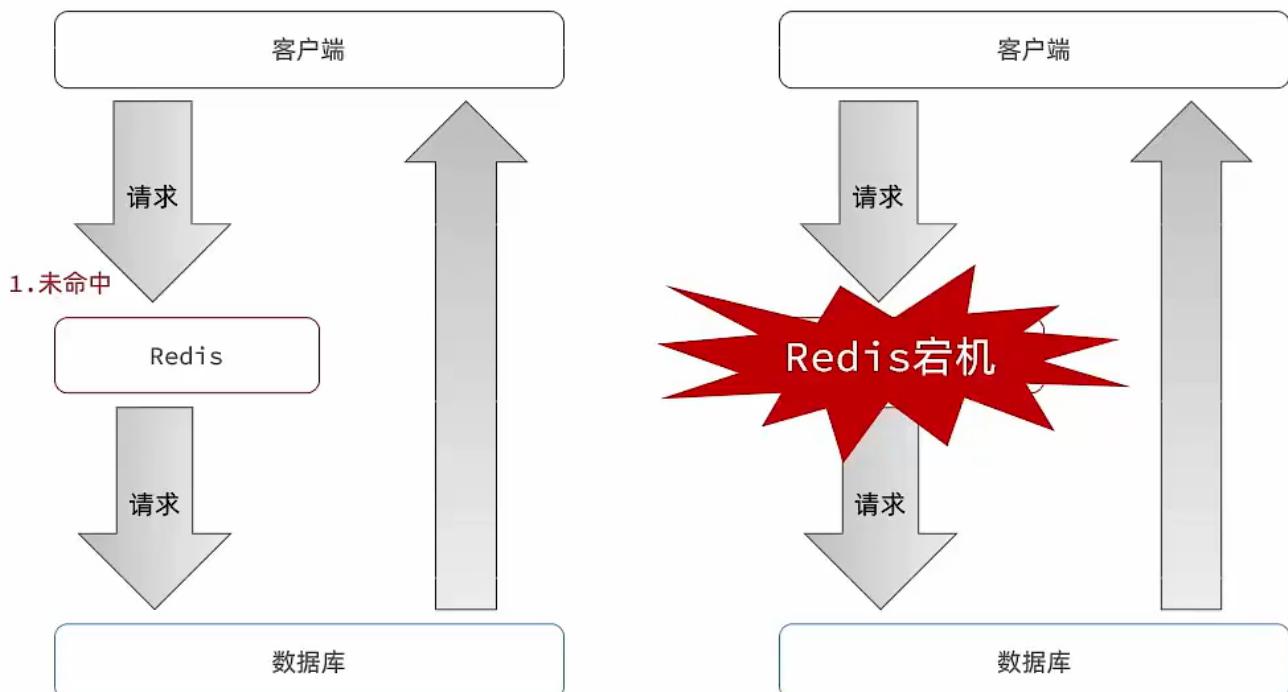
在原来的逻辑中，我们如果发现这个数据在mysql中不存在，直接就返回404了，这样是会存在缓存穿透问题的

现在的逻辑中：如果这个数据不存在，我们不会返回404，还是会把这个数据写入到Redis中，并且将value设置为空，每当再次发起查询时，我们如果发现命中之后，判断这个value是否是null，如果是null，则是之前写入的数据，证明是缓存穿透数据，如果不是，则直接返回数据。



3、缓存雪崩

缓存雪崩: 是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。



解决方案：

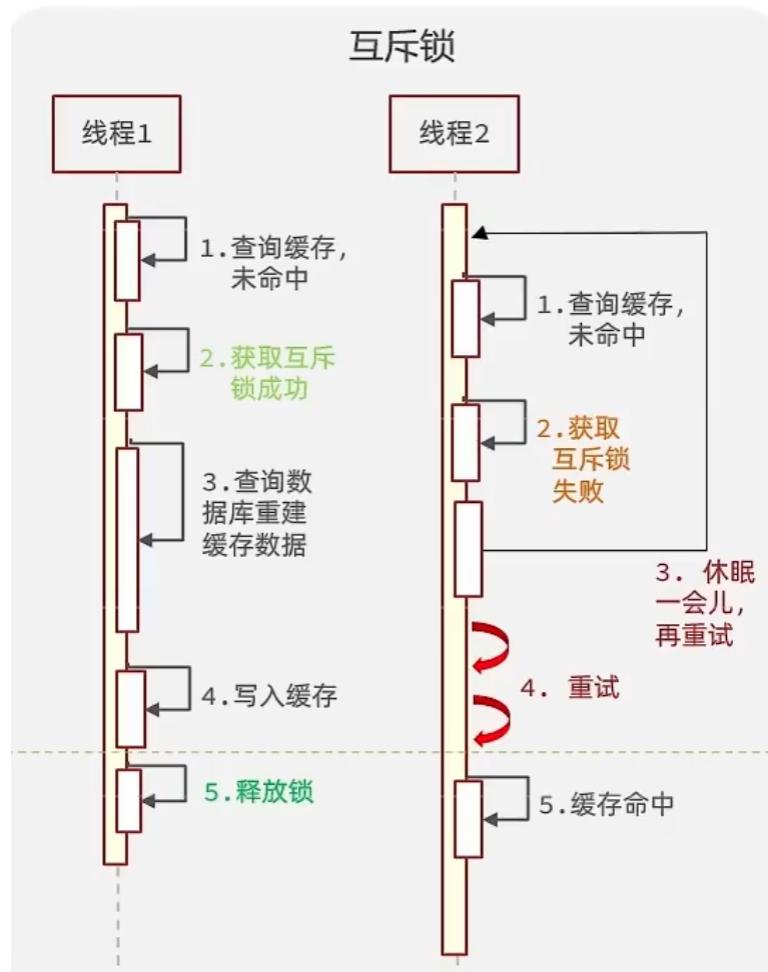
- 给不同的Key的TTL添加随机值
 - 利用Redis集群提高服务的可用性
 - 给缓存业务添加降级限流策略
 - 给业务添加多级缓存

4、缓存击穿

缓存击穿问题也叫热点Key问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击

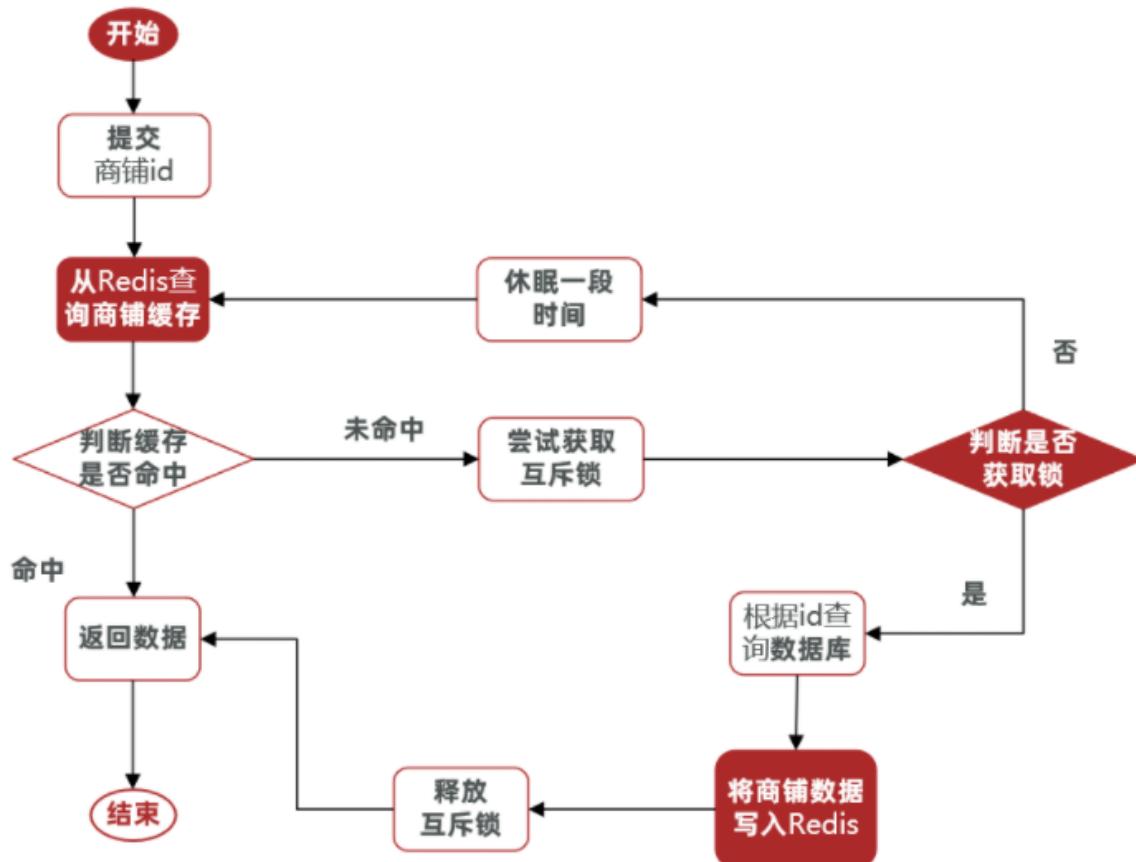
法1：互斥锁

其实就是一个时间段只能一个线程去处理问题，然后给个锁不让其他线程进来处理，让他去睡觉，睡一会儿等锁被释放了再来处理



业务逻辑：

需求：修改根据id查询商铺的业务，基于互斥锁的方式来解决缓存击穿问题



代码实现：

1、写锁

```

private boolean tryLock(String key) {
    Boolean flag = stringRedisTemplate.opsForValue().setIfAbsent(key, "1", 10,
TimeUnit.SECONDS);
    return BooleanUtil.isTrue(flag);
}

private void unlock(String key) {
    stringRedisTemplate.delete(key);
}
  
```

2、查询带锁

```

public Shop queryWithMutex(Long id) {
    String key = CACHE_SHOP_KEY + id;
    // 1、从redis中查询商铺缓存
    String shopJson = stringRedisTemplate.opsForValue().get("key");
    // 2、判断是否存在
    if (StrUtil.isNotBlank(shopJson)) {
        // 存在,直接返回
        return JSONUtil.toBean(shopJson, Shop.class);
    }
}
  
```

```

    }
    //判断命中的值是否是空值
    if (shopJson != null) {
        //返回一个错误信息
        return null;
    }
    // 4.实现缓存重构
    //4.1 获取互斥锁
    String lockKey = "lock:shop:" + id;
    Shop shop = null;
    try {
        boolean isLock = tryLock(lockKey);
        // 4.2 判断否获取成功
        if(!isLock){
            //4.3 失败，则休眠重试
            Thread.sleep(50);
            return queryWithMutex(id);
        }
        //4.4 成功，根据id查询数据库
        shop = getById(id);
        // 5.不存在，返回错误
        if(shop == null){
            //将空值写入redis

stringRedisTemplate.opsForValue().set(key, "",CACHE_NULL_TTL,TimeUnit.MINUTES);
            //返回错误信息
            return null;
        }
        //6.写入redis

stringRedisTemplate.opsForValue().set(key,JSONUtil.toJsonStr(shop),CACHE_NULL_TTL,
TimeUnit.MINUTES);

    }catch (Exception e){
        throw new RuntimeException(e);
    }
    finally {
        //7.释放互斥锁
        unlock(lockKey);
    }
    return shop;
}

```

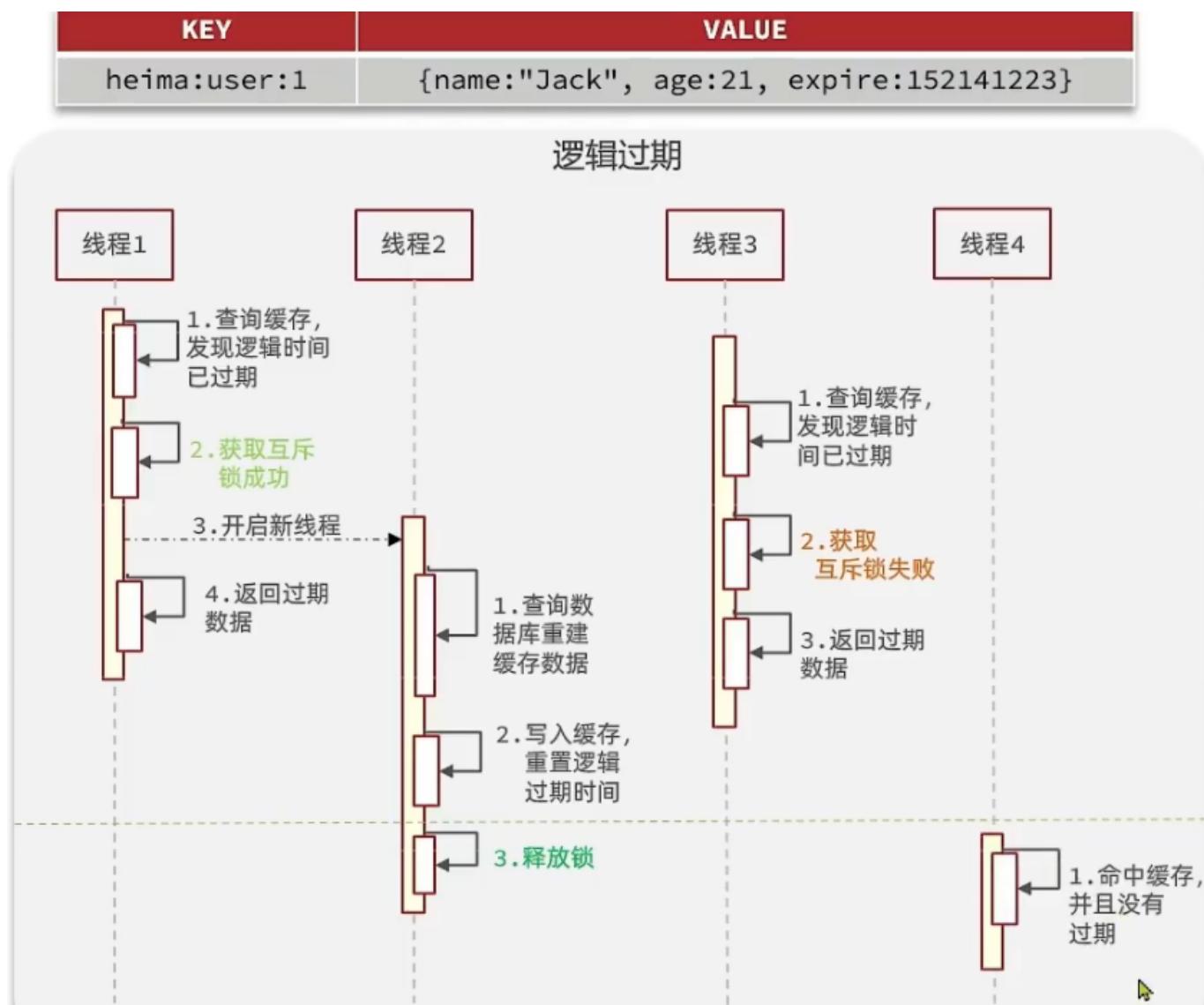
法2：逻辑过期

既然是高并发访问那干脆就直接redis里面一直都不要删除了，再加个逻辑过期时间，过期的话就开个独立线程去更新数据写入redis，在没更新完之前访问到的都是redis里面的旧数据

but意想不到的难点居然是如何给数据添加过期时间的字段

方案一：新建一个RedisData类，这个类有过期时间字段，然后让pojo继承这个类/直接在原有的类直接加字段，但是这样就改变基础代码了

so方案二：新建一个RedisData类，这个类有过期时间字段，同时有另一个Object字段（ob就是牛啦存什么都可以）用来存数据。相当于给数据又加了一层封装



两种方法对比

解决方案	优点	缺点
互斥锁	<ul style="list-style-type: none"> • 没有额外的内存消耗 • 保证一致性 • 实现简单 	<ul style="list-style-type: none"> • 线程需要等待，性能受影响 • 可能有死锁风险
逻辑过期	<ul style="list-style-type: none"> • 线程无需等待，性能较好 	<ul style="list-style-type: none"> • 不保证一致性 • 有额外内存消耗 • 实现复杂

5、封装Redis工具类

基于StringRedisTemplate封装一个缓存工具类，满足下列需求：

- 方法1：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间
- 方法2：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，用于处理缓存击穿问题
- 方法3：根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题
- 方法4：根据指定的key查询缓存，并反序列化为指定类型，需要利用逻辑过期解决缓存击穿问题

将逻辑进行封装

```
/*
 * 将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间
 *
 * @param key    redis键
 * @param value  redis值
 * @param time   缓存时间
 * @param unit   时间单位
 */
public void set(String key, Object value, Long time, TimeUnit unit) {
    // 注意这里要把value转为string类型
    stringRedisTemplate.opsForValue().set(key, JSONUtil.toJsonStr(value),
time, unit);
}

/*
 * 将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，  

* 用于处理缓存击穿问题
 *
 * @param key    redis键
 * @param value  redis值
 * @param time   缓存时间
 * @param unit   时间单位
 */
public void setWithLogicalExpire(String key, Object value, Long time, TimeUnit
unit) {
    // 设置逻辑过期
    RedisData redisData = new RedisData();
    redisData.setData(value);
    // 注意转second

    redisData.setExpireTime(LocalDateTime.now().plusSeconds(unit.toSeconds(time)));

    //写入redis
    stringRedisTemplate.opsForValue().set(key, JSONUtil.toJsonStr(redisData));
}

/*
 * 根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题
 *
```

```
* @param keyPrefix 键前缀
* @param id 就id啦
* @param type 要转换的数据类型
* @param dbFallback 查询数据库的函数
* @param time 时间
* @param unit 时间单位
* @param <R> 数据类型
* @param <ID> id类型
*/
// 返回值不确定 — 使用泛型 (先定义泛型Class<R> type, 再返回类型<R>R)
// id也不确定 — 还是泛型, 用ID, 泛型类型定义改成<R, ID>R
// 查数据库的逻辑不确定 — 用Function<ID, R> :ID是入参, R是返回值
// 过期时间也不要写死 — 用Long time, TimeUnit unit
public <R, ID> R queryWithPassThrough(String keyPrefix, ID id, Class<R> type,
Function<ID, R> dbFallback, Long time, TimeUnit unit) {
    //查询redis, 若存在则转换成对象后返回
    String key = keyPrefix + id;
    String Json = stringRedisTemplate.opsForValue().get(key);

    //这里判断的是Json是否真的有值, 不包括空值
    if (StringUtils.isNotBlank(Json)) {
        return JSONUtil.toBean(Json, type);
    }

    // 判断缓存是否命中(命中的是否是空值)。
    // 如果isNotBlank + !=null, 说明命中, 之前就请求过了且redis设为了"", 这种情况
    // 也不要再请求redis了, 直接返回错误
    if (Json != null) {
        return null;
    }
    //不存在则查询数据库, 然后转成以json串存入redis后, 返回
    R r = dbFallback.apply(id);
    if (r == null) {
        // 将空值写入redis
        stringRedisTemplate.opsForValue().set(key, "", CACHE_NULL_TTL, unit);
        return null;
    }
    this.set(key, r, time, unit);
    return r;
}

/**
 * 根据指定的key查询缓存, 并反序列化为指定类型, 利用逻辑过期的方式解决缓存击穿问题
 */
public <R, ID> R queryWithLogicalExpire(String keyPrefix, ID id, Class<R>
type, Function<ID, R> dbFallback, Long time, TimeUnit unit) {
    //查询redis, 这里的shopJson是(Object)RedisData类型的
    String key = keyPrefix + id;
    String Json = stringRedisTemplate.opsForValue().get(key);

    //未命中, 说明不是热点key
    if (StringUtils.isBlank(Json)) {
```

```
        return null;
    }

    // 命中的话再判断是否逻辑过期
    RedisData redisData = JSONUtil.toBean(Json, RedisData.class);
    R r = JSONUtil.toBean((JSONObject) redisData.getData(), type);
    LocalDateTime expireTime = redisData.getExpireTime();
    // 未过期直接返回shop
    if (LocalDateTime.now().isBefore(expireTime)) {
        return r;
    }

    // 过期了就重建缓存：先获取锁，再开个独立线程处理
    String lockKey = keyPrefix + id;
    boolean lock = tryLock(lockKey);
    if (lock) {
        CACHE_REBUILD_EXECUTOR.submit(() -> {
            try {
                // 模拟重建延迟 saveShop2Redis
                // 1. 查数据库
                R r1 = dbFallback.apply(id);
                // 2. 带逻辑过期地写入redis
                this.setWithLogicalExpire(key, r1, time, unit);
            } catch (Exception e) {
                throw new RuntimeException(e);
            } finally {
                unLock(lockKey);
            }
        });
    }
    // 返回旧数据
    return r;
}

/**
 * 根据指定的key查询缓存，并反序列化为指定类型，利用互斥锁的方式解决缓存击穿问题
 */
public <R, ID> R queryWithMutex(String keyPrefix, ID id, Class<R> type, String lockKeyPrefix, Function<ID, R> dbFallback, Long time, TimeUnit unit) {

    // 查询redis，若存在则转换成对象后返回
    String key = keyPrefix + id;
    String json = stringRedisTemplate.opsForValue().get(key);

    // 这里判断的是shopJson是否真的有值，不包括空值
    if (StringUtils.isNotBlank(json)) {
        return JSONUtil.toBean(json, type);
    }

    // 判断缓存是否命中(命中的是否是空值)。
    // 如果isNotBlank + !=null，说明命中，之前就请求过了且redis设为了“”，这种情况也不要再请求redis了，直接返回错误
    if (json != null) {
        return null;
    }
}
```

```
        }
        // 未命中，进行缓存穿透处理
        // 先加锁，防止缓存穿透
        String lockKey = lockKeyPrefix + id;
        R r1;
        try {
            boolean lock = tryLock(lockKey);
            while (!lock) {
                // 获取锁失败，偷偷睡一觉，再重新查询
                TimeUnit.MILLISECONDS.sleep(50);
                lock = tryLock(lockKey);
            }
            // DoubleCheck(因为此时有可能别的线程已经重新构建好缓存)
            json = stringRedisTemplate.opsForValue().get(key);
            if (StringUtils.isNotBlank(json)) {
                r1 = JSONUtil.toBean(json, type);
                return r1;
            }
        }

        // 模拟重建延迟
        try {
            TimeUnit.MILLISECONDS.sleep(100);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        // 不存在则查询数据库，然后转成以json串存入redis后，返回
        r1 = JSONUtil.toBean(json, type);
        if (r1 == null) {
            // 将空值写入redis，解决缓存穿透问题
            stringRedisTemplate.opsForValue().set(key, "", 10,
TimeUnit.MINUTES);
            return null;
        }
        stringRedisTemplate.opsForValue().set(key, JSONUtil.toJsonStr(r1),
time, unit);

    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        unLock(lockKey);
    }

    return r1;
}

// 获取锁
private boolean tryLock(String lockKey) {
    // 首先尝试获取锁，获取不到返回false
    Boolean flag = stringRedisTemplate.opsForValue().setIfAbsent(lockKey, "1",
LOCK_SHOP_TTL, TimeUnit.SECONDS);
    // 不直接返回Boolean类型，避免自动拆箱时出现空指针异常。setIfAbsent内部是long转
    boolean再转Boolean，可能会出现空指针异常的
    // 条件表达式，只有满足不为null和为true时才返回true
```

```

        // return flag != null && flag;
        return BooleanUtil.isTrue(flag);
    }

    //释放锁
    private void unLock(String lockKey) {
        stringRedisTemplate.delete(lockKey);
    }
}

```

在ShopServiceImpl 中

```

@Resource
private CacheClient cacheClient;

@Override
public Result queryById(Long id) {
    // 解决缓存穿透
    Shop shop = cacheClient
        .queryWithPassThrough(CACHE_SHOP_KEY, id, Shop.class,
this::getById, CACHE_SHOP_TTL, TimeUnit.MINUTES);

    // 互斥锁解决缓存击穿
    // Shop shop = cacheClient
    //         .queryWithMutex(CACHE_SHOP_KEY, id, Shop.class, this::getById,
CACHE_SHOP_TTL, TimeUnit.MINUTES);

    // 逻辑过期解决缓存击穿
    // Shop shop = cacheClient
    //         .queryWithLogicalExpire(CACHE_SHOP_KEY, id, Shop.class,
this::getById, 20L, TimeUnit.SECONDS);

    if (shop == null) {
        return Result.fail("店铺不存在! ");
    }
    // 7.返回
    return Result.ok(shop);
}

```

优惠券秒杀

1、生成全局唯一ID

如果使用数据库自增ID就存在一些问题：

- id的规律性太明显
- 受单表数据量的限制

全局ID生成器，是一种在分布式系统下用来生成全局唯一ID的工具，一般要满足下列特性：唯一性、安全性、递增性、高性能、高可用

为了增加ID的安全性，我们可以不直接使用Redis自增的数值，而是**拼接一些其它信息**



ID的组成部分：

符号位：1bit，永远为0

时间戳：31bit，以秒为单位，可以使用69年

序列号：32bit，秒内的计数器，支持每秒产生 2^{32} 个不同ID

代码实现：utils/RedisIdWorker.java

```

@Component
public class RedisIdWorker {
    /**
     * 开始时间戳
     */
    private static final long BEGIN_TIMESTAMP = 1640995200L;
    /**
     * 序列号的位数
     */
    private static final int COUNT_BITS = 32;

    private StringRedisTemplate stringRedisTemplate;

    public RedisIdWorker(StringRedisTemplate stringRedisTemplate) {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    public long nextId(String keyPrefix) {
        // 1.生成时间戳
        LocalDateTime now = LocalDateTime.now();
        long nowSecond = now.toEpochSecond(ZoneOffset.UTC);
        long timestamp = nowSecond - BEGIN_TIMESTAMP;

        // 2.生成序列号
        // 2.1.获取当前日期，精确到天
        String date = now.format(DateTimeFormatter.ofPattern("yyyy:MM:dd"));
        // 2.2.自增长
        long count = stringRedisTemplate.opsForValue().increment("incr:" +
keyPrefix + ":" + date);

        // 3.拼接并返回
        return timestamp << COUNT_BITS | count;
    }
}

```

```
    }  
}
```

2、添加优惠券

普通券直接save

秒杀券要保存更多信息到Voucher表

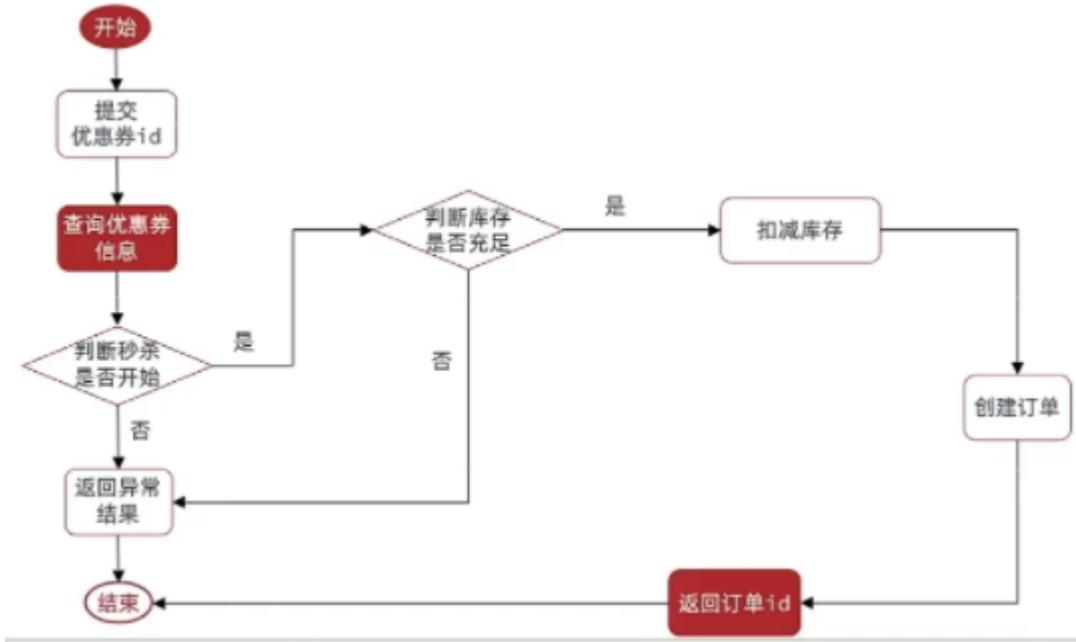
代码实现：

```
@Override  
@Transactional  
public void addSeckillVoucher(Voucher voucher) {  
    // 保存优惠券  
    save(voucher);  
    // 保存秒杀信息  
    SeckillVoucher seckillVoucher = new SeckillVoucher();  
    seckillVoucher.setVoucherId(voucher.getId());  
    seckillVoucher.setStock(voucher.getStock());  
    seckillVoucher.setBeginTime(voucher.getBeginTime());  
    seckillVoucher.setEndTime(voucher.getEndTime());  
    seckillVoucherService.save(seckillVoucher);  
    // 保存秒杀库存到Redis中  
    stringRedisTemplate.opsForValue().set(SECKILL_STOCK_KEY + voucher.getId(),  
    voucher.getStock().toString());  
}
```

3、实现秒杀下单

下单时需要判断两点：

- 秒杀是否开始或结束，如果尚未开始或已经结束则无法下单
- 库存是否充足，不足则无法下单



代码实现：VoucherOrderServiceImpl

```

@Override
public Result seckillVoucher(Long voucherId) {
    // 1. 查询优惠券
    SeckillVoucher voucher = seckillVoucherService.getById(voucherId);
    // 2. 判断秒杀是否开始
    if (voucher.getBeginTime().isAfter(LocalDateTime.now())) {
        // 尚未开始
        return Result.fail("秒杀尚未开始!");
    }
    // 3. 判断秒杀是否已经结束
    if (voucher.getEndTime().isBefore(LocalDateTime.now())) {
        // 尚未开始
        return Result.fail("秒杀已经结束!");
    }
    // 4. 判断库存是否充足
    if (voucher.getStock() < 1) {
        // 库存不足
        return Result.fail("库存不足!");
    }
    // 5. 扣减库存
    boolean success = seckillVoucherService.update()
        .setSql("stock= stock -1")
        .eq("voucher_id", voucherId).update();
    if (!success) {
        // 扣减库存
        return Result.fail("库存不足!");
    }
    // 6. 创建订单
    VoucherOrder voucherOrder = new VoucherOrder();
    // 6.1. 订单id
    long orderId = redisIdWorker.nextId("order");
    voucherOrder.setId(orderId);
    // 6.2. 用户id
  
```

```

        Long userId = UserHolder.getUser().getId();
        voucherOrder.setUserId(userId);
        // 6.3. 代金券id
        voucherOrder.setVoucherId(voucherId);
        save(voucherOrder);

        return Result.ok(orderId);

    }

```

4、解决超卖问题

假设线程1过来查询库存，判断出来库存大于1，正准备去扣减库存，但是还没有来得及去扣减，此时线程2过来，线程2也去查询库存，发现这个数量一定也大于1，那么这两个线程都会去扣减库存，最终多个线程相当于一起去扣减库存，此时就会出现库存的超卖问题。

解决思路：



悲观锁：

悲观锁可以实现对于数据的串行化执行，比如syn，和lock都是悲观锁的代表，同时，悲观锁中又可以再细分为公平锁，非公平锁，可重入锁，等等

乐观锁：

乐观锁：会有一个版本号，每次操作数据会对版本号+1，再提交回数据时，会去校验是否比之前的版本大1，如果大1，则进行操作成功，这套机制的核心逻辑在于，如果在操作过程中，版本号只比原来大1，那么就意味着操作过程中没有人对他进行过修改，他的操作就是安全的，如果不<1，则数据被修改过

修改代码方案一、

VoucherOrderServiceImpl 在扣减库存时，改为：

```

boolean success = seckillVoucherService.update()
    .setSql("stock= stock -1") //set stock = stock -1
    .eq("voucher_id", voucherId).eq("stock", voucher.getStock()).update();
//where id = ? and stock = ?

```

以上逻辑的核心含义是：只要我扣减库存时的库存和之前我查询到的库存是一样的，就意味着没有人在中间修改过库存，那么此时就是安全的，但是以上这种方式通过测试发现会有很多失败的情况，失败的原因在于：在使用乐观锁过程中假设100个线程同时都拿到了100的库存，然后大家一起去进行扣减，但是100个人中只有1个人能扣减成功，其他的人在处理时，他们在扣减时，库存已经被修改过了，所以此时其他线程都会失败

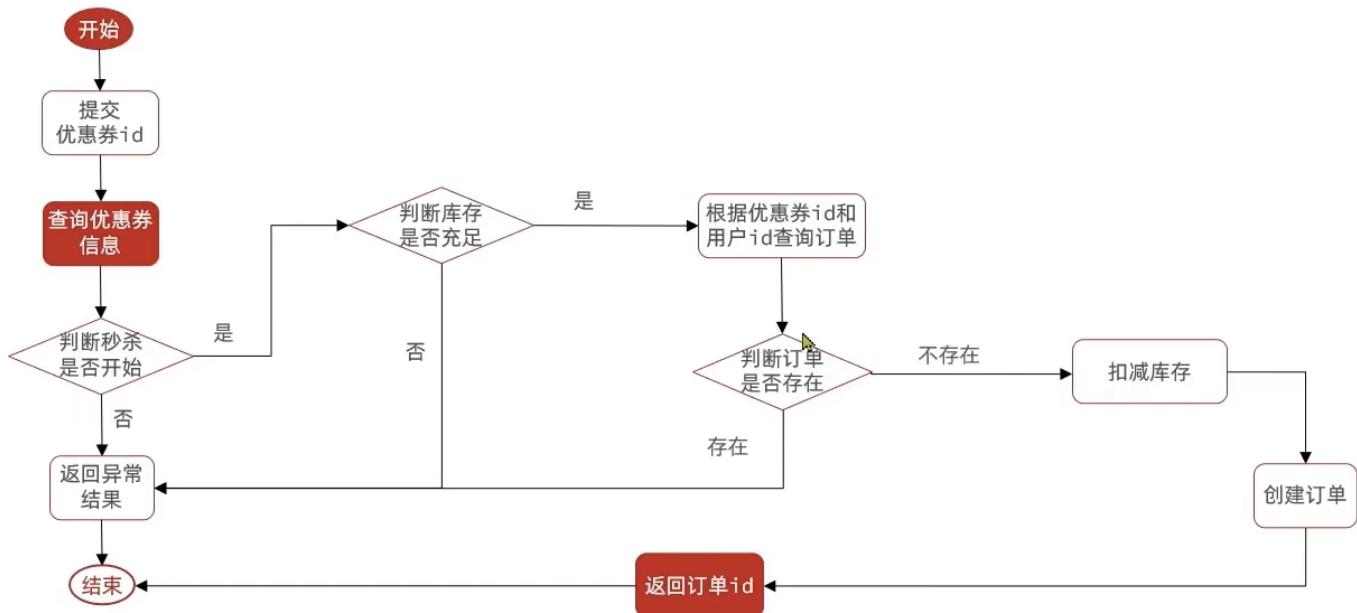
修改代码方案二、

之前的方式要修改前后都保持一致，但是这样我们分析过，成功的概率太低，所以我们的乐观锁需要变一下，改成stock大于0 即可

```
boolean success = seckillVoucherService.update()
    .setSql("stock= stock -1")
    .eq("voucher_id", voucherId).update().gt("stock",0); //where id = ?
and stock > 0
```

5、实现一人一单

业务逻辑



代码实现：VoucherOrderServiceImpl

```
@Override
public Result seckillVoucher(Long voucherId) {
    SeckillVoucher voucher = seckillVoucherService.getById(voucherId);

    // 判断是否在秒杀时间内
    if (voucher.getBeginTime().isAfter(LocalDateTime.now()) ||
        voucher.getEndTime().isBefore(LocalDateTime.now())) {
        return Result.fail("不在秒杀时间内");
    }

    // 判断是否还有库存
```

```
if (voucher.getStock() < 1) {
    return Result.fail("库存不足");
}

Long userId = UserHolder.getUser().getId();
// 通过userId控制锁的粒度，只有相同用户才会加锁
// synchronized是java内置的一个线程同步关键字，可以卸载需要同步的对象、方法或者
特定的代码块中
// intern()方法是将字符串放入常量池中，这样相同的字符串就会指向同一个对象，从而
实现锁的粒度控制
synchronized (userId.toString().intern()) {
    // 通过AopContext.currentProxy()获取代理对象，从而实现事务控制
    IVoucherOrderService proxy = (IVoucherOrderService)
AopContext.currentProxy();
    return proxy.createVoucherOrder(voucherId);
}

public Result createVoucherOrder(Long voucherId) {
    // 一人一单逻辑
    Long userId = UserHolder.getUser().getId();

    int count = query().eq("user_id", userId).eq("voucher_id",
voucherId).count();
    // 判断是否存在
    if (count > 0) {
        return Result.fail("用户已经购买过一次! ");
    }

    // 减库存
    boolean success = seckillVoucherService.update()
        .setSql("stock = stock-1")
        .eq("voucher_id", voucherId)
        // 乐观锁解决超卖问题
        // .eq("stock", voucher.getStock())
        .gt("stock", 0)
        .update();

    if (!success) {
        return Result.fail("减库存失败");
    }

    // 创建订单：保存订单信息到数据库中
    VoucherOrder voucherOrder = new VoucherOrder();

    // 生成订单id、用户id、代金券id
    long orderId = redisIdWorker.nextId("order");
    voucherOrder.setId(orderId);
    voucherOrder.setUserId(UserHolder.getUser().getId());
    voucherOrder.setVoucherId(voucherId);

    save(voucherOrder);
    return Result.ok(orderId);
}
```

细节描述：

1. 关于synchronized和Lock的区别

synchronized采用的是悲观锁机制

Lock采用的是乐观锁机制

项	synchronized	Lock
特性	Java的关键字，在JVM层面	J.U.C包中的接口
获取	A获得锁，B等待。A阻塞，B一直等待	可尝试获得锁，线程可以不用一直等待
释放	执行完同步代码或者发生异常，被动释放	在finally中释放锁，避免死锁
状态	无法判断	可以判断
类型	可以重入，不可中断，非公平	可重入，可判断，可公平，可非公平
性能	少量同步	大量同步

知乎 @Tom弹架构

2. 实现代理对象

防止事务失效，所以要用代理对象

pom.xml添加

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

application.java添加

```
@EnableAspectJAutoProxy(exposeProxy = true) //暴露代理对象
```

VoucherOrderServiceImpl实现

```
synchronized (userId.toString().intern()) {
    // 通过AopContext.currentProxy()获取代理对象，从而实现事务控制
    IVoucherOrderService proxy = (IVoucherOrderService)
        AopContext.currentProxy();
```

```

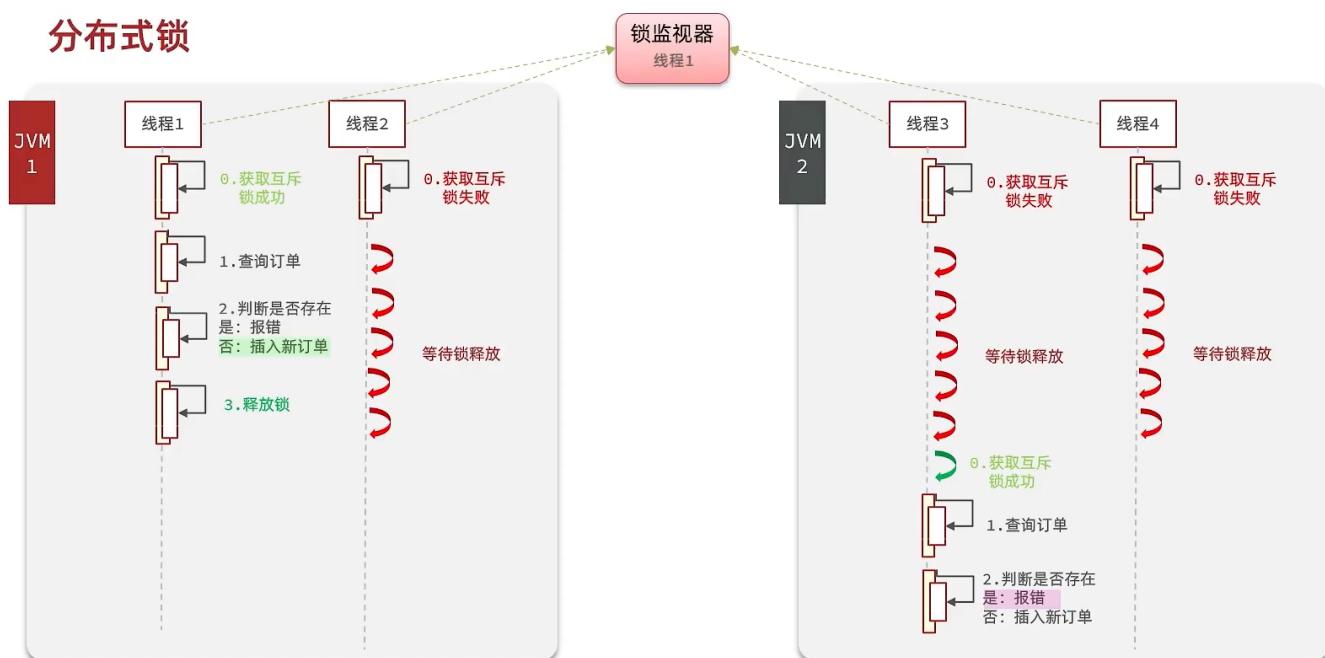
        return proxy.createVoucherOrder(voucherId);
    }
}

```

分布式锁

为什么需要分布式锁

由于现在我们部署了多个tomcat，每个tomcat都有一个属于自己的jvm，现在是服务器B的tomcat内部，又有两个线程，但是他们的锁对象写的虽然和服务器A一样，但是锁对象却不是同一个，所以线程3和线程4可以实现互斥，但是却无法和线程1和线程2实现互斥，这就是 **集群环境下syn锁失效**的原因，在这种情况下，我们就需要使用分布式锁来解决这个问题。



简单介绍

分布式锁：满足分布式系统或集群模式下多进程可见并且互斥的锁。

核心思想就是让大家都使用同一把锁锁住线程

满足条件：

- 可见性：**多个线程都能看到相同的结果，注意：这个地方说的可见性并不是并发编程中指的内存可见性，只是说多个进程之间都能感知到变化的意思
- 互斥：**互斥是分布式锁的最基本的条件，使得程序串行执行
- 高可用：**程序不易崩溃，时时刻刻都保证较高的可用性
- 高性能：**由于加锁本身就让性能降低，所有对于分布式锁本身需要有较高的加锁性能和释放锁性能
- 安全性：**安全也是程序中必不可少的一环

常见的分布式锁有三种

Mysql: mysql本身就带有锁机制，但是由于mysql性能本身一般，所以采用分布式锁的情况下，其实使用mysql作为分布式锁比较少见

Redis: redis作为分布式锁是非常常见的一种使用方式，现在企业级开发中基本都使用redis或者zookeeper作为分布式锁，利用setnx这个方法，如果插入key成功，则表示获得到了锁，如果有人插入成功，其他人插入失败则表示无法获得到锁，利用这套逻辑来实现分布式锁

Zookeeper: zookeeper也是企业级开发中较好的一个实现分布式锁的方案，由于本套视频并不讲解zookeeper的原理和分布式锁的实现，所以不过多阐述

	MySQL	Redis	Zookeeper
互斥	利用mysql本身的互斥锁机制	利用setnx这样的互斥命令	利用节点的唯一性和有序性实现互斥
高可用	好	好	好
高性能	一般	好	一般
安全性	断开连接，自动释放锁	利用锁超时时间，到期释放	临时节点，断开连接自动释放

redis使用：利用redis 的setNx 方法，确保每次只有一个锁存在，同时增加过期时间，防止死锁，此方法可以保证加锁和增加过期时间具有原子性

代码实现

锁的基本接口：utils/ILock

```
public interface ILock {
    /**
     * 尝试获取锁
     * @param timeoutSec 锁持有的超时时间，过期后自动释放
     * @return true代表获取锁成功；false代表获取锁失败
     */
    boolean tryLock(long timeoutSec);

    /**
     * 释放锁
     */
    void unlock();
}
```

锁的实现：utils/SimpleRedisLock

```
public class SimpleRedisLock implements ILock {
```

```

private final String name;
private final StringRedisTemplate stringRedisTemplate;

public SimpleRedisLock(String name, StringRedisTemplate stringRedisTemplate) {
    this.name = name;
    this.stringRedisTemplate = stringRedisTemplate;
}

// 获取锁
@Override
public boolean tryLock(long timeoutSec) {
    // 获取线程标示，因为我们要区分到底是哪个线程拿到了这个锁，方便之后unlock的时候
    // 进行判断
    String threadId = Thread.currentThread().getId();
    // 获取锁
    Boolean success = stringRedisTemplate.opsForValue()
        .setIfAbsent(KEY_PREFIX + name, threadId, timeoutSec,
    TimeUnit.SECONDS);
    return Boolean.TRUE.equals(success);
}

// 释放锁
@Override
public void unlock() {
    // 通过del删除锁
    stringRedisTemplate.delete(KEY_PREFIX + name);
}
}

```

VoucherOrderServiceImpl

```

@Override
public Result seckillVoucher(Long voucherId) {
    SeckillVoucher voucher = seckillVoucherService.getById(voucherId);

    // 判断是否在秒杀时间内
    .....
    // 判断是否还有库存
    .....
    Long userId = UserHolder.getUser().getId();
    // 之前的：没有考虑集群模式下的锁问题
    .....

    // 【完善代码】：考虑集群模式下的锁问题
    // 创建锁对象
    SimpleRedisLock lock = new SimpleRedisLock("order:" + userId,
    stringRedisTemplate);
    // 获取锁对象
    boolean isLock = lock.tryLock(1200);
    // 加锁失败
    if (!isLock) {

```

```

        return Result.fail("不允许重复下单");
    }

    // 这里就是为了调用createVoucherOrder方法，但是要考虑到事务的问题，所以要通过代理对象来调用
    try {
        // 获取代理对象(事务)
        IVoucherOrderService proxy = (IVoucherOrderService)
AopContext.currentProxy();
        return proxy.createVoucherOrder(voucherId);
    } finally {
        // 释放锁
        lock.unlock();
    }
}
}

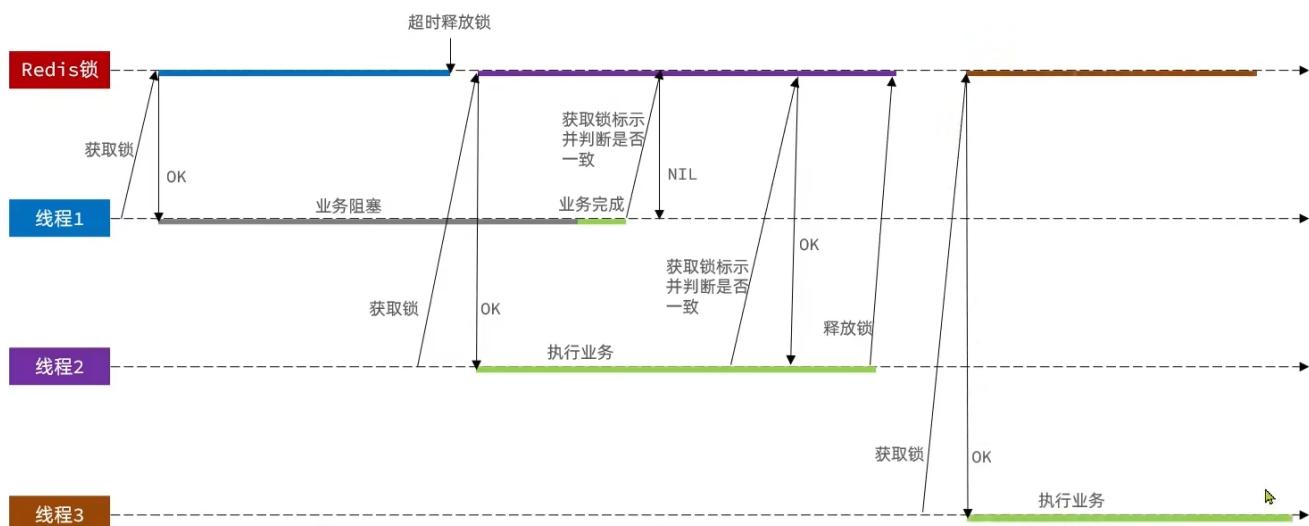
```

误删问题！

问题描述

持有锁的线程在锁的内部出现了阻塞，导致他的锁自动释放，这时其他线程，线程2来尝试获得锁，就拿到了这把锁，然后线程2在持有锁执行过程中，线程1反应过来，继续执行，而线程1执行过程中，走到了删除锁逻辑，此时就会把本应该属于线程2的锁进行删除，这就是误删别人锁的情况说明

解决方案：解决方案就是在每个线程释放锁的时候，去判断一下当前这把锁是否属于自己，如果属于自己，才进行锁的删除，否则不做删除。



解决思路：在获取锁时存入线程标示（可以用UUID表示），一致放锁，不一致不放锁

代码实现：

```

private static final String ID_PREFIX = UUID.randomUUID().toString(true) + "-";
@Override
public boolean tryLock(long timeoutSec) {
    // 获取线程标示

```

```

String threadId = ID_PREFIX + Thread.currentThread().getId();
// 获取锁
Boolean success = stringRedisTemplate.opsForValue()
    .setIfAbsent(KEY_PREFIX + name, threadId, timeoutSec,
TimeUnit.SECONDS);
return Boolean.TRUE.equals(success);
}

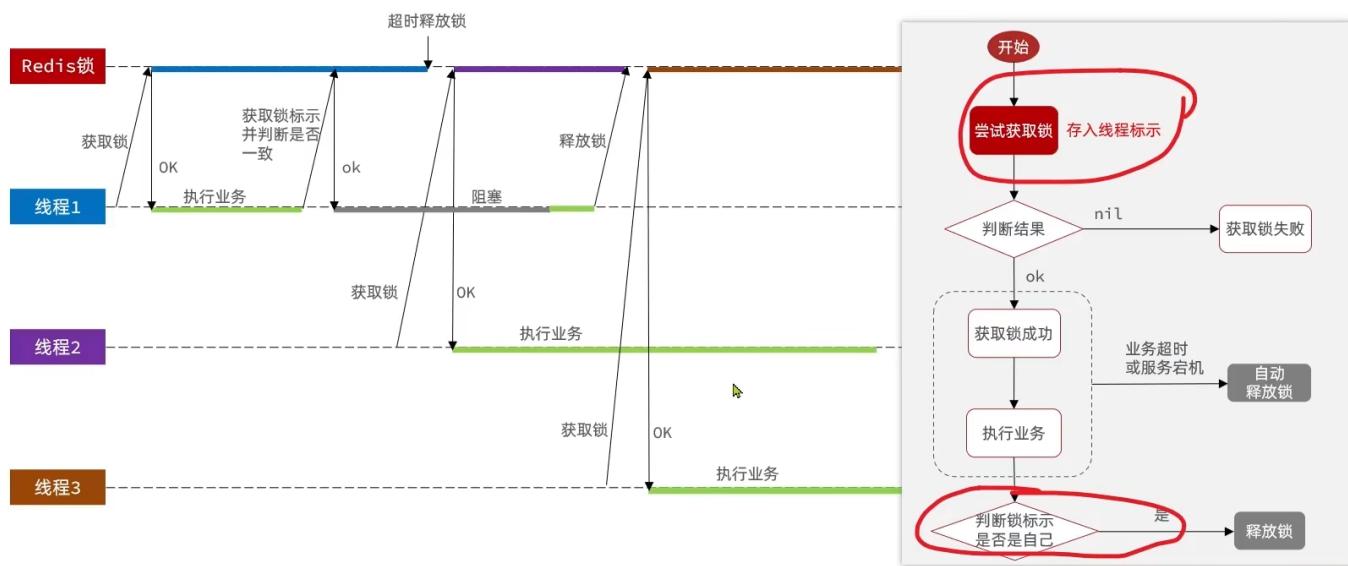
public void unlock() {
// 获取线程标示
String threadId = ID_PREFIX + Thread.currentThread().getId();
// 获取锁中的标示
String id = stringRedisTemplate.opsForValue().get(KEY_PREFIX + name);
// 判断标示是否一致
if(threadId.equals(id)) {
// 释放锁
stringRedisTemplate.delete(KEY_PREFIX + name);
}
}
}

```

分布式锁的原子性问题

更为极端的误删逻辑说明：

线程1现在持有锁之后，在执行业务逻辑过程中，他正准备删除锁，而且已经走到了条件判断的过程中，比如他已经拿到了当前这把锁确实是属于他自己的，正准备删除锁，但是此时他的锁到期了，那么此时线程2进来，但是线程1他会接着往后执行，当他卡顿结束后，他直接就会执行删除锁那行代码，相当于条件判断并没有起到作用，这就是删锁时的原子性问题，**之所以有这个问题，是因为线程1的拿锁，比锁，删锁，实际上并不是原子性的，我们要防止刚才的情况发生**



解决思路：Lua脚本解决多条命令原子性问题

Redis提供了Lua脚本功能，在一个脚本中编写多条Redis命令，确保多条命令执行时的原子性。

Lua是一种编程语言，它的基本语法大家可以参考网站：<https://www.runoob.com/lua/lua-tutorial.html>

这里重点介绍Redis提供的调用函数，我们可以使用lua去操作redis，又能保证他的原子性，这样就可以实现拿锁比锁删锁是一个原子性动作了，作为Java程序员这一块并不作一个简单要求，并不需要大家过于精通，只需要知道他有什么作用即可。

redis使用：RedisTemplate中可以利用**execute方法**去执行lua脚本，对应参数如下：【脚本名称、key、参数数组】

```

RedisTemplate.java
    /**
     * (non-Javadoc)
     * @see org.springframework.data.redis.core.RedisOperations#execute(org.springframework.data.redis.core.RedisScript<T>, java.util.List, Object...)
     */
    @Override
    public <T> T execute(RedisScript<T> script, List<K> keys, Object... args) {
        return scriptExecutor.execute(script, keys, args);
    }
}

127.0.0.1:6379> help @scripting
EVAL script numkeys key [key ...] arg [arg ...]
summary: Execute a Lua script server side
since: 2.6.0

```

代码实现：utils/unlock.lua

```

-- 锁的key
-- local id = "lock:order:5"
-- 当前线程标识
-- local threadId = "asdavewrfawe-26"
-- 因为不能写死，所以我们要进行参数传递

-- 这里的 KEYS[1] 就是锁的key，这里的ARGV[1] 就是当前线程标示，脚标从1开始
-- 获取锁中的标示，判断是否与当前线程标示一致
if (redis.call('GET', KEYS[1]) == ARGV[1]) then
    -- 一致，则删除锁
    return redis.call('DEL', KEYS[1])
end
-- 不一致，则直接返回
return 0

```

VoucherOrderServiceImpl

```

// 提前在静态代码块中加载lua脚本，， 提高性能
private static final DefaultRedisScript<Long> UNLOCK_SCRIPT;
static {
    UNLOCK_SCRIPT = new DefaultRedisScript<>();
    UNLOCK_SCRIPT.setLocation(new ClassPathResource("unlock.lua"));
    UNLOCK_SCRIPT.setResultType(Long.class);
}

```

```

    }

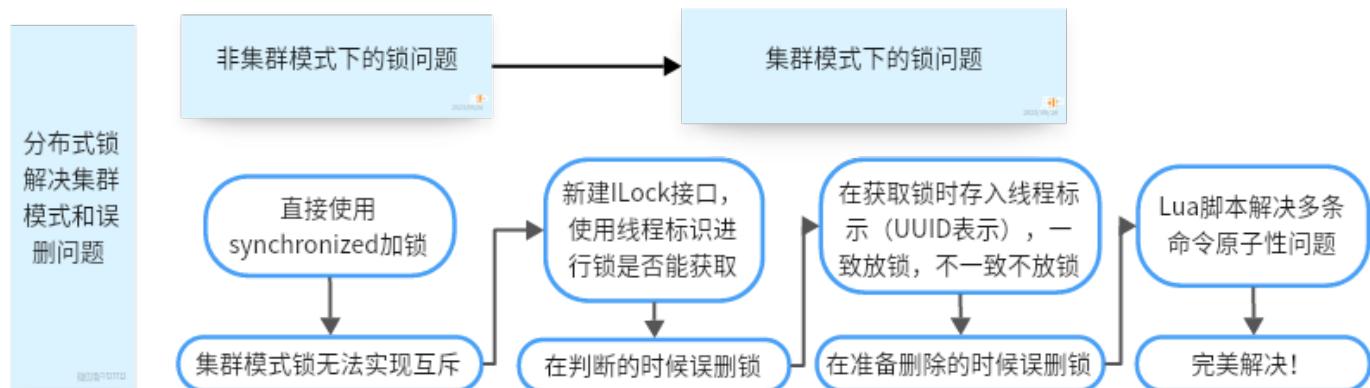
public void unlock() {
    // 调用lua脚本解决误删问题，确保原子性
    stringRedisTemplate.execute(
        UNLOCK_SCRIPT,
        // 字符串转集合
        Collections.singletonList(KEY_PREFIX + name),
        // 线程标识
        ID_PREFIX + Thread.currentThread().getId());
}

```

小总结

基于Redis的分布式锁实现思路：

- 利用set nx ex获取锁，并设置过期时间，保存线程标示
- 释放锁时先判断线程标示是否与自己一致，一致则删除锁
 - 特性：
 - 利用set nx满足互斥性
 - 利用set ex保证故障时锁依然能释放，避免死锁，提高安全性
 - 利用Redis集群保证高可用和高并发特性

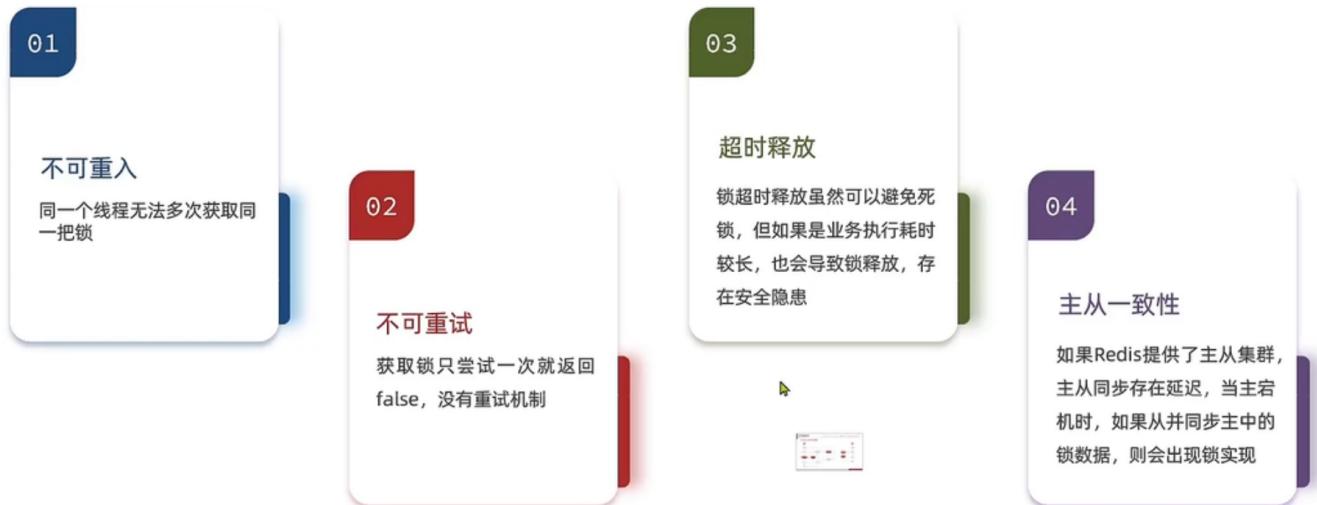


分布式锁-redission

1、简单介绍

基于setnx实现的分布式锁存在下面的问题：

基于setnx实现的分布式锁存在下面的问题：



重入问题：所以可重入锁的主要意义是防止死锁，我们的synchronized和Lock锁都是可重入的。

不可重试：我们认为合理的情况是：当线程在获得锁失败后，他应该能再次尝试获得锁。

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务，其中就包含了各种分布式锁的实现。

Redisson提供了分布式锁的多种多样的功能【不算白雪！】

8. 分布式锁 (Lock) 和同步器 (Synchronizer)

- 8.1. 可重入锁 (Reentrant Lock)
- 8.2. 公平锁 (Fair Lock)
- 8.3. 联锁 (MultiLock)
- 8.4. 红锁 (RedLock)
- 8.5. 读写锁 (ReadWriteLock)
- 8.6. 信号量 (Semaphore)
- 8.7. 可过期性信号量 (PermitExpirableSemaphore)
- 8.8. 闭锁 (CountDownLatch)

2、快速入门

引入依赖：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>
```

配置Redisson客户端：

```
@Configuration
public class RedissonConfig {

    @Bean
    public RedissonClient redissonClient(){
        // 配置
        Config config = new Config();
        config.useSingleServer().setAddress("redis://192.168.150.101:6379")
            .setPassword("123321");
        // 创建RedissonClient对象
        return Redisson.create(config);
    }
}
```

使用Redisson的分布式锁

```
@Resource
private RedissonClient redissonClient;

@Test
void testRedisson() throws Exception{
    //获取锁(可重入), 指定锁的名称
    RLock lock = redissonClient.getLock("anyLock");
    //尝试获取锁, 参数分别是: 获取锁的最大等待时间(期间会重试), 锁自动释放时间, 时间单位
    boolean isLock = lock.tryLock(1,10,TimeUnit.SECONDS);
    //判断获取锁成功
    if(isLock){
        try{
            System.out.println("执行业务");
        }finally{
            lock.unlock();
        }
    }
}
```

优化VoucherOrderServiceImpl中的seckillVoucher加锁

```
// 这里修改成了redisson的分布式锁
// SimpleRedisLock lock = new SimpleRedisLock("order:" + userId,
stringRedisTemplate);
RLock lock = redissonClient.getLock("order:" + userId);
```

3、可重入锁原理

同一个线程获取两次或以上锁就算是重入了

redis使用：在分布式锁中，redisson采用hash结构用来存储锁，其中大key表示表示这把锁是否存在，用小key (filed) 表示当前这把锁被哪个线程持有，还要记录这把锁的重入了几次 (value)

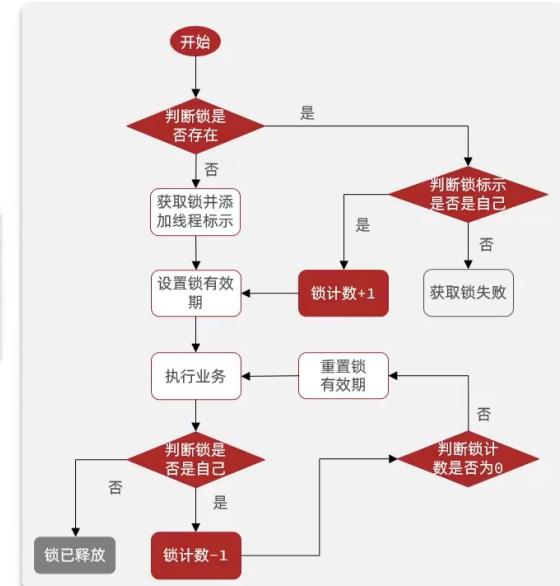
对于可重入锁，在删除的时候必须注意重入次数 (value) 是否为0，防止误删

业务逻辑：注意两个重置有效期时间

```
// 创建锁对象
RLock lock = redissonClient.getLock("lock");

@Test
void method1() {
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 1");
        return;
    }
    try {
        log.info("获取锁成功, 1");
        method2();
    } finally {
        log.info("释放锁, 1");
        lock.unlock();
    }
}
void method2(){
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 2");
        return;
    }
    try {
        log.info("获取锁成功, 2");
    } finally {
        log.info("释放锁, 2");
        lock.unlock();
    }
}
```

KEY	VALUE	
	field	value
lock	thread1	1



代码实现：好复杂，所以我们选择lua脚本实现

获取锁

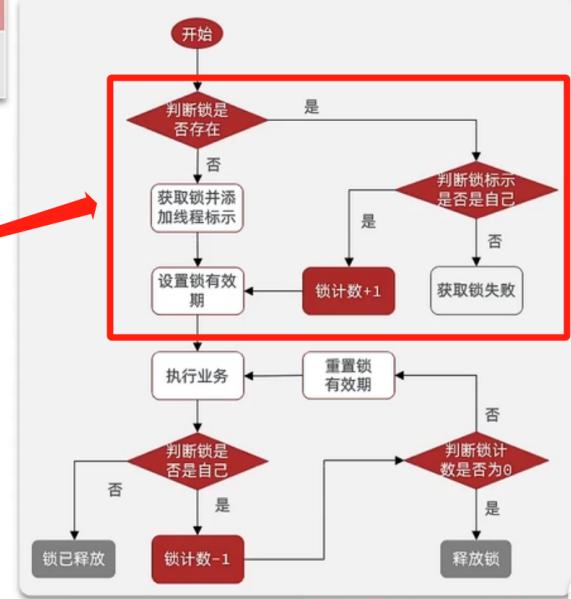
用exists手动判断是否存在【? why not nx】

Redisson可重入锁原理

获取锁的Lua脚本：

```
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断是否存在
if(redis.call('exists', key) == 0) then
    -- 不存在, 获取锁
    redis.call('hset', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
-- 锁已经存在, 判断threadId是否是自己
if(redis.call('hexists', key, threadId) == 1) then
    -- 不存在, 获取锁, 重入次数+1
    redis.call('hincrby', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
return 0; -- 代码走到这里, 说明获取锁的不是自己, 获取锁失败
```

KEY	VALUE	
	field	value
lock	thread1	0



释放锁

Redisson可重入锁原理

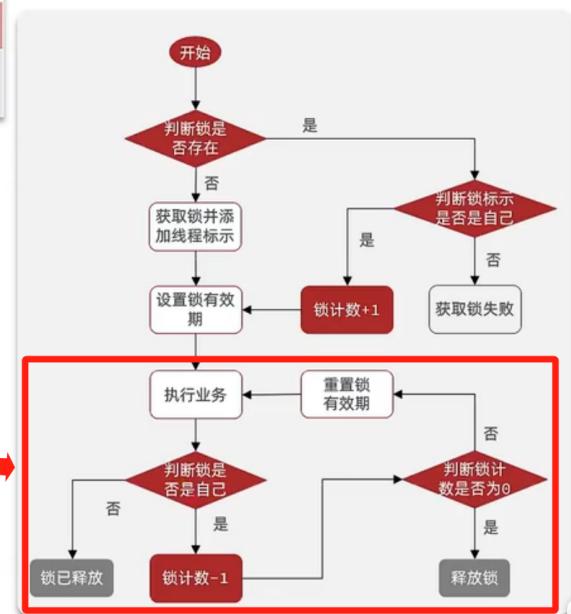
释放锁的Lua脚本：

```

local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断当前锁是否还是被自己持有
if (redis.call('HEXISTS', key, threadId) == 0) then
    return nil; -- 如果已经不是自己，则直接返回
end;
-- 是自己的锁，则重入次数-1
local count = redis.call('HINCRBY', key, threadId, -1);
-- 判断是否重入次数是否已经为0
if (count > 0) then
    -- 大于0说明不能释放锁，重置有效期然后返回
    redis.call('EXPIRE', key, releaseTime);
    return nil;
else -- 等于0说明可以释放锁，直接删除
    redis.call('DEL', key);
    return nil;
end;

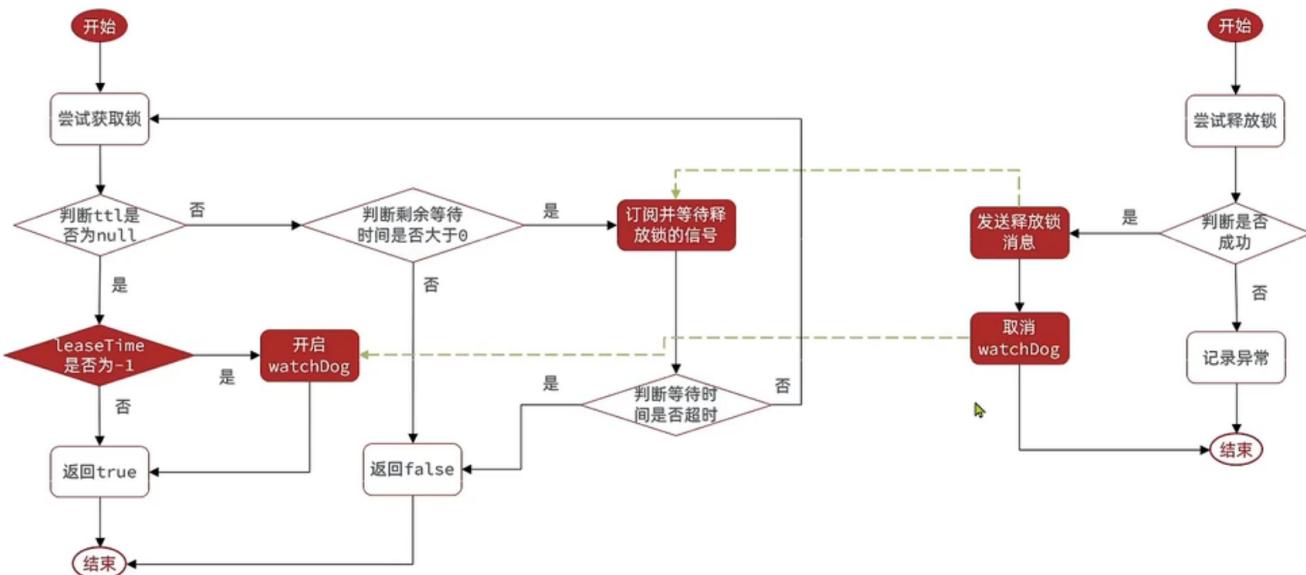
```

KEY	VALUE	
	field	value
lock:order	thread1	0



4、锁重试和WatchDog机制

Redisson分布式锁原理



部分关键代码【不太懂、、、反正就是实现了可重试和超时续约】

```

private RFuture<Boolean> tryAcquireOnceAsync(long waitTime, long leaseTime,
TimeUnit unit, long threadId) {
    if (leaseTime != -1L) {
        return this.tryLockInnerAsync(waitTime, leaseTime, unit, threadId,
RedisCommands.EVAL_NULL_BOOLEAN);
    } else {
        RFuture<Boolean> ttlRemainingFuture = this.tryLockInnerAsync(waitTime,
this.commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout(),
TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL_NULL_BOOLEAN);
        ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
            if (e == null) {

```

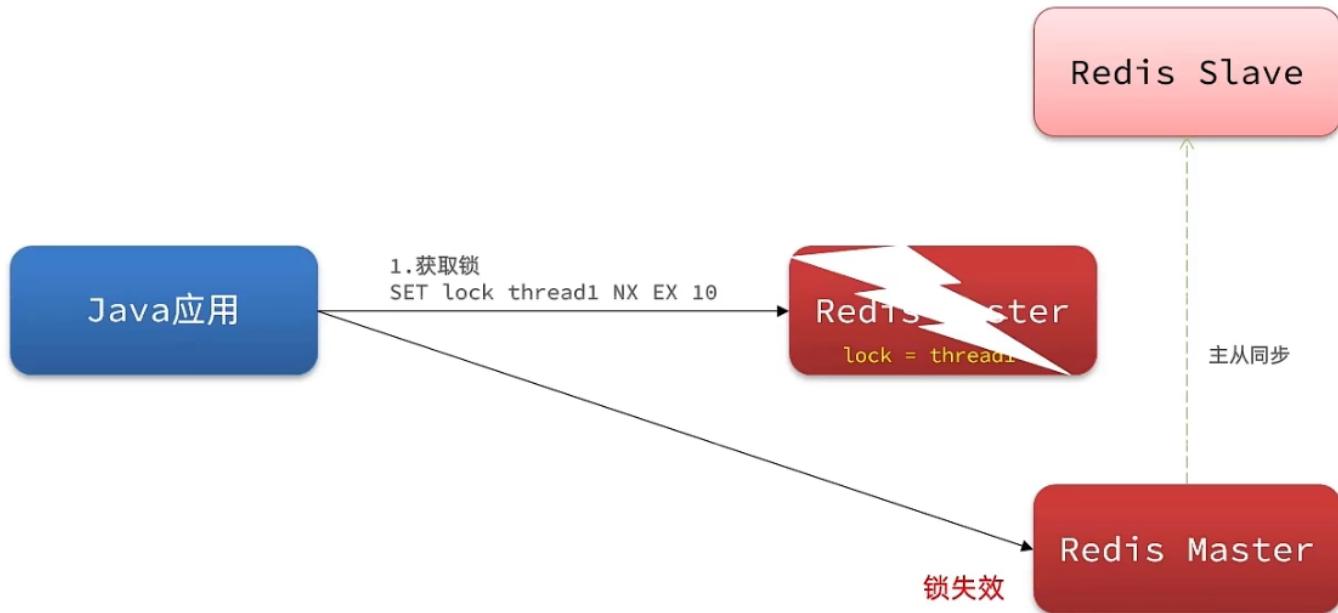
```
        if (ttlRemaining) {
            this.scheduleExpirationRenewal(threadId);
        }
    });
    return ttlRemainingFuture;
}
}

<T> RFuture<T> tryLockInnerAsync(long waitTime, long leaseTime, TimeUnit unit,
long threadId, RedisStrictCommand<T> command) {
    this.internalLockLeaseTime = unit.toMillis(leaseTime);
    return this.evalWriteAsync(this.getName(), LongCodec.INSTANCE, command,
        "if (redis.call('exists', KEYS[1]) == 0) then "+
        "redis.call('hincrby', KEYS[1], ARGV[2], 1); "+
        "redis.call('pexpire', KEYS[1], ARGV[1]); return nil; "+
        "end; "+
        "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) "+
        "then redis.call('hincrby', KEYS[1], ARGV[2], 1); "+
        "redis.call('pexpire', KEYS[1], ARGV[1]); "+
        "return nil; "+
        "end; "+
        "return redis.call('pttl', KEYS[1]);",
        Collections.singletonList(this.getName()),
        this.internalLockLeaseTime, this.getLockName(threadId));
}
```

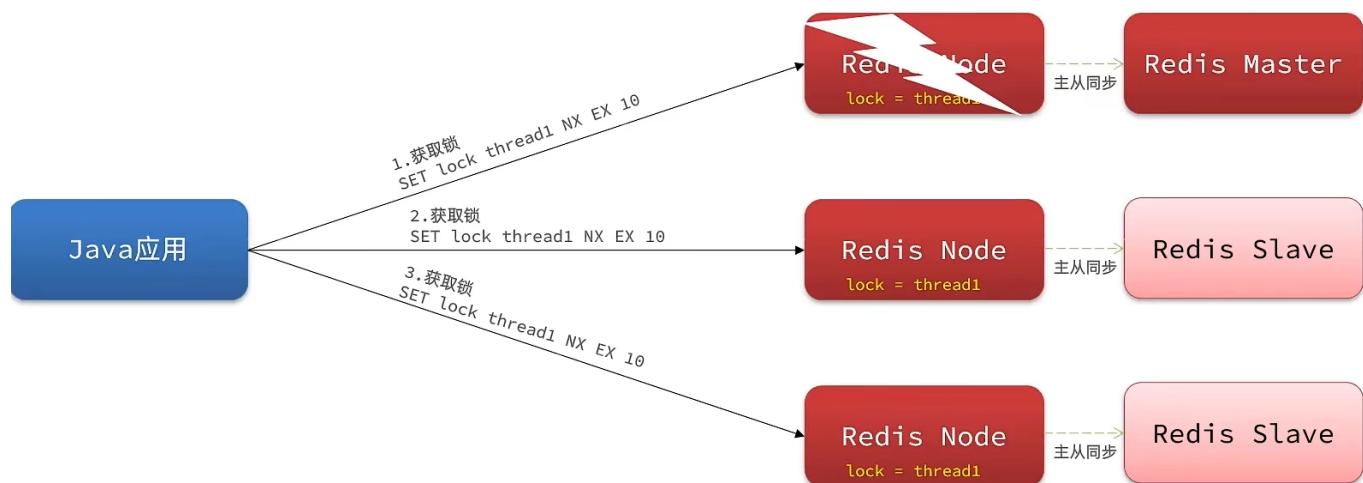
5、MultiLock原理

主从不一致问题：

使用主从：我们去写命令写在主机上，主机会将数据同步给从机，但是假设在主机还没有来得及把数据写入到从机去的时候，此时主机宕机，哨兵会发现主机宕机，并且选举一个slave变成master，而此时新的master中实际上并没有锁信息，此时锁信息就已经丢掉了



为了解决这个问题，redisson提出了MutliLock锁，使用这把锁每个节点的地位都是一样的，这把锁加锁的逻辑需要写入到每一个主从节点上，**只有所有的服务器都写入成功，此时才是加锁成功**，假设现在某个节点挂了，那么他去获得锁的时候，只要有一个节点拿不到，都不能算是加锁成功，就保证了加锁的可靠性。



代码实现：

1. RedissonConfig

```
@Configuration
public class RedissonConfig {

    @Bean
    public RedissonClient redissonClient(){
        // 配置
        Config config = new Config();
        // 添加redis地址，这里选择添加单点地址，也可以使用config.useClusterServers()添加集群地址
        config.useSingleServer().setAddress("redis://127.0.0.1:6379");
        // 创建RedissonClient对象
        return Redisson.create(config);
    }
}
```

```
}

@Bean
public RedissonClient redissonClient2(){
    //同上，改变地址就行
}

@Bean
public RedissonClient redissonClient3(){
    //同上，改变地址就行
}

}
```

2. RedissonTest

```
@Resource
private RedissonClient redissonClient;
@Resource
private RedissonClient redissonClient2;
@Resource
private RedissonClient redissonClient3;

@BeforeEach
void setUp() {
    RLock lock1 = redissonClient.getLock("order");
    RLock lock2 = redissonClient2.getLock("order");
    RLock lock3 = redissonClient3.getLock("order");

    // 创建联锁multiLock
    RLock lock = redissonClient.getMultiLock(lock1, lock2, lock3);
}

.....//其他的都不用动
```

6、总结：Redisson分布式锁原理

1) 不可重入Redis分布式锁：

- ◆ 原理：利用setnx的互斥性；利用ex避免死锁；释放锁时判断线程标示
- ◆ 缺陷：不可重入、无法重试、锁超时失效

2) 可重入的Redis分布式锁：

- ◆ 原理：利用hash结构，记录线程标示和重入次数；利用watchDog延续锁时间；利用信号量控制锁重试等待
- ◆ 缺陷：redis宕机引起锁失效问题

3) Redisson的multiLock：

- ◆ 原理：多个独立的Redis节点，必须在所有节点都获取重入锁，才算获取锁成功
- ◆ 缺陷：运维成本高、实现复杂

可重入：利用hash记录线程id和重入次数

可重试：利用信号量和PubSub功能实现等待、唤醒，获取锁失败的重试机制

超时续约：利用watchDog，每隔一段时间 (releaseTime/3)，重置超时时间

主从一致：利用multiLock，多个独立的Redis节点，必须在所有节点都获取重入锁才算获取锁成功

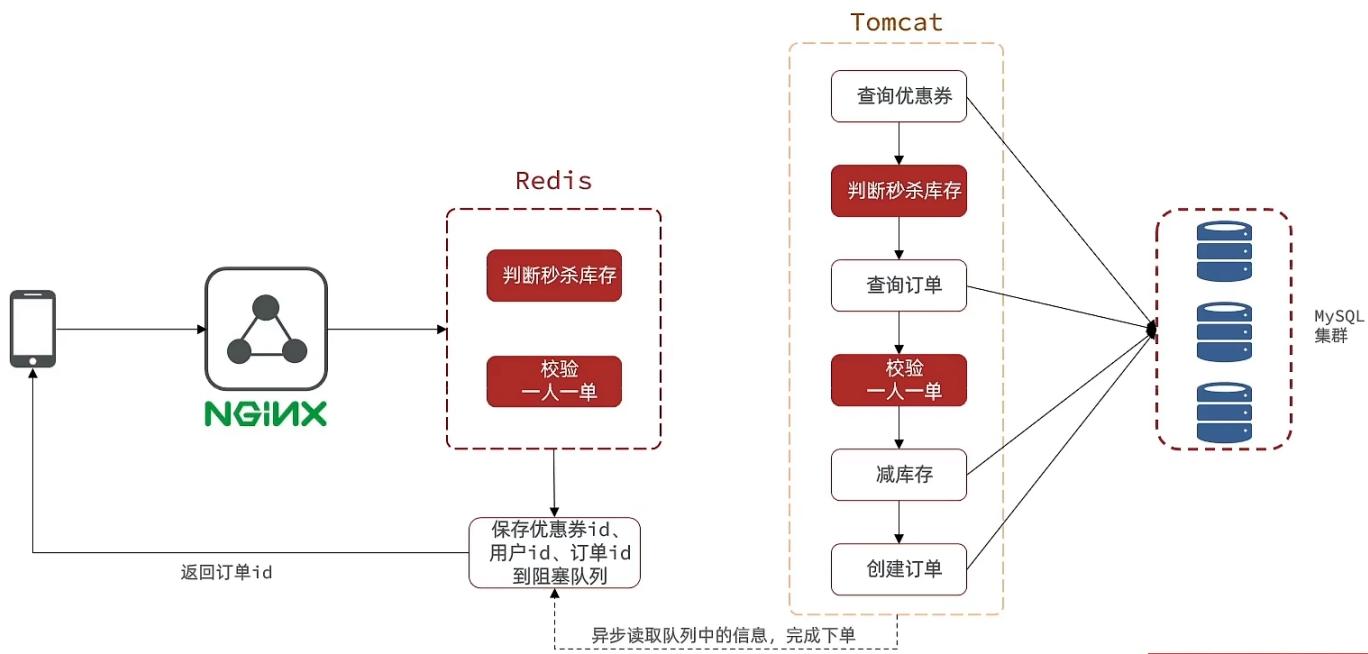
优化秒杀

1、异步秒杀思路

下单流程步骤：查询优惠卷 -- 判断秒杀库存是否足够 -- 查询订单 -- 校验是否是一人一单 -- 扣减库存 -- 创建订单

在这六步操作中，又有很多操作是要去操作数据库的，而且还是一个线程串行执行，这样就会导致我们的程序执行的很慢，所以我们需要**异步程序执行**

异步程序执行流程：



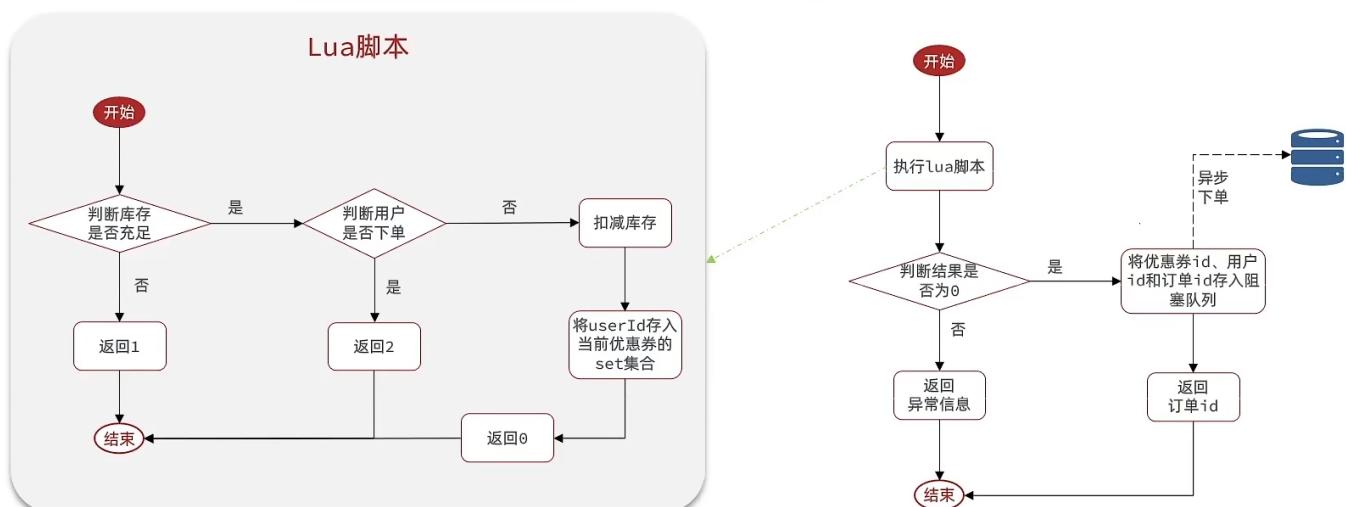
当用户下单之后，**判断库存是否充足只需要去redis中去根据key找对应的value是否大于0即可**，如果不充足，则直接结束，如果充足，继续在redis中判断用户是否可以下单

KEY	VALUE	KEY	VALUE
stock:vid:7	100	order:vid:7	1 , 2 , 3 , 5 , 7 , 8

如果set集合中没有这条数据，说明他可以下单，如果set集合中没有这条记录，则将userId和优惠卷存入到redis中，并且返回0，我们可以使用lua语言来操作保证整个过程的原子性

Redis优化秒杀

KEY	VALUE	KEY	VALUE
stock:vid:7	100	order:vid:7	1 , 2 , 3 , 5 , 7 , 8



2、Redis完成秒杀资格判断

需求：

- 新增秒杀优惠券的同时，将优惠券信息保存到Redis中
- 基于Lua脚本，判断秒杀库存、一人一单，决定用户是否抢购成功

- 如果抢购成功，将优惠券id和用户id封装后存入阻塞队列
- 开启线程任务，不断从阻塞队列中获取信息，实现异步下单功能

代码实现：

1、seckill.lua

```
-- 秒杀优化需求二：基于Lua脚本，判断秒杀库存、一人一单，决定用户是否有购买资格
-- 1.参数列表
-- 1.1.优惠券id
local voucherId = ARGV[1]
-- 1.2.用户id
local userId = ARGV[2]
-- 1.3.订单id
local orderId = ARGV[3]

-- 2.数据key
-- 2.1.库存key ..lua的字符串拼接
local stockKey = 'seckill:stock:' .. voucherId
-- 2.2.订单key
local orderKey = 'seckill:order:' .. voucherId

-- 3.脚本业务
-- 3.1.判断库存是否充足 get stockKey tonumber()将字符串转换为数字
if(tonumber(redis.call('get', stockKey)) <= 0) then
    -- 3.2.库存不足，返回1
    return 1
end
-- 3.2.判断用户是否下单 SISMEMBER：判断set集合中是否存在某个元素
if(redis.call('sismember', orderKey, userId) == 1) then
    -- 3.3.存在，说明是重复下单，返回2
    return 2
end
-- 3.4.扣库存 incrby stockKey -1
redis.call('incrby', stockKey, -1)
-- 3.5.下单 (保存用户) sadd orderKey userId
redis.call('sadd', orderKey, userId)
-- 3.6.发送消息到队列中， XADD stream.orders * k1 v1 k2 v2 ...
redis.call('xadd', 'stream.orders', '*', 'userId', userId, 'voucherId', voucherId,
'id', orderId)

return 0
```

2、VoucherOrderServiceImpl

```
@Override
public Result seckillVoucher(Long voucherId) {
    //获取用户
    Long userId = UserHolder.getUser().getId();
    long orderId = redisIdWorker.nextId("order");
```

```

// 1.执行lua脚本
Long result = stringRedisTemplate.execute(
    SECKILL_SCRIPT,
    Collections.emptyList(),
    voucherId.toString(), userId.toString(), String.valueOf(orderId)
);
int r = result.intValue();
// 2.判断结果是否为0
if (r != 0) {
    // 2.1.不为0 , 代表没有购买资格
    return Result.fail(r == 1 ? "库存不足" : "不能重复下单");
}
//TODO 保存阻塞队列
// 3.返回订单id
return Result.ok(orderId);
}

```

3、基于阻塞队列实现秒杀优化

代码实现：VoucherOrderServiceImpl【sorry我没太搞懂这里在干嘛、、、】

```

//异步处理线程池
private static final ExecutorService SECKILL_ORDER_EXECUTOR =
Executors.newSingleThreadExecutor();

//在类初始化之后执行，因为当这个类初始化好了之后，随时都是有可能要执行的
@PostConstruct
private void init() {
    SECKILL_ORDER_EXECUTOR.submit(new VoucherOrderHandler());
}

// 用于线程池处理的任务
// 当初始化完毕后，就会去从对列中去拿信息
private class VoucherOrderHandler implements Runnable{

    @Override
    public void run() {
        while (true){
            try {
                // 1.获取队列中的订单信息
                VoucherOrder voucherOrder = orderTasks.take();
                // 2.创建订单
                handleVoucherOrder(voucherOrder);
            } catch (Exception e) {
                log.error("处理订单异常", e);
            }
        }
    }
}

private void handleVoucherOrder(VoucherOrder voucherOrder) {
    //1.获取用户，注意是从voucherOrder中取，因为现在是在多线程中
    Long userId = voucherOrder.getUserId();
}

```

```
// 2.创建锁对象
RLock redisLock = redissonClient.getLock("lock:order:" + userId);
// 3.尝试获取锁
boolean isLock = redisLock.lock();
// 4.判断是否获得锁成功
if (!isLock) {
    // 获取锁失败，直接返回失败或者重试
    log.error("不允许重复下单! ");
    return;
}
try {
    //注意：由于是spring的事务是放在threadLocal中，此时的是多线程，事务会
失效
    proxy.createVoucherOrder(voucherOrder);
} finally {
    // 释放锁
    redisLock.unlock();
}
}
//a
private BlockingQueue<VoucherOrder> orderTasks =new ArrayBlockingQueue<>(1024
* 1024);
@Override
public Result seckillVoucher(Long voucherId) {
    Long userId = UserHolder.getUser().getId();
    long orderId = redisIdWorker.nextId("order");
    // 1.执行lua脚本
    Long result = stringRedisTemplate.execute(
        SECKILL_SCRIPT,
        Collections.emptyList(),
        voucherId.toString(), userId.toString(), String.valueOf(orderId)
    );
    int r = result.intValue();
    // 2.判断结果是否为0
    if (r != 0) {
        // 2.1.不为0，代表没有购买资格
        return Result.fail(r == 1 ? "库存不足" : "不能重复下单");
    }
    VoucherOrder voucherOrder = new VoucherOrder();
    // 2.3.订单id
    long orderId = redisIdWorker.nextId("order");
    voucherOrder.setId(orderId);
    // 2.4.用户id
    voucherOrder.setUserId(userId);
    // 2.5.代金券id
    voucherOrder.setVoucherId(voucherId);
    // 2.6.放入阻塞队列
    orderTasks.add(voucherOrder);
    //3.获取代理对象
    proxy = (IVoucherOrderService)AopContext.currentProxy();
    //4.返回订单id
    return Result.ok(orderId);
}
```

```

    @Transactional
    public void createVoucherOrder(VoucherOrder voucherOrder) {
        Long userId = voucherOrder.getUserId();
        // 5.1.查询订单
        int count = query().eq("user_id", userId).eq("voucher_id",
        voucherOrder.getVoucherId()).count();
        // 5.2.判断是否存在
        if (count > 0) {
            // 用户已经购买过了
            log.error("用户已经购买过了");
            return ;
        }

        // 6.扣减库存
        boolean success = seckillVoucherService.update()
            .setSql("stock = stock - 1") // set stock = stock - 1
            .eq("voucher_id", voucherOrder.getVoucherId()).gt("stock", 0) //
        where id = ? and stock > 0
            .update();
        if (!success) {
            // 扣减失败
            log.error("库存不足");
            return ;
        }
        save(voucherOrder);

    }

```

小总结：

秒杀业务的优化思路是什么？

- 先利用Redis完成库存余量、一人一单判断，完成抢单业务
- 再将下单业务放入阻塞队列，利用独立线程异步下单
- 基于阻塞队列的异步秒杀存在哪些问题？
 - 内存限制问题
 - 数据安全问题

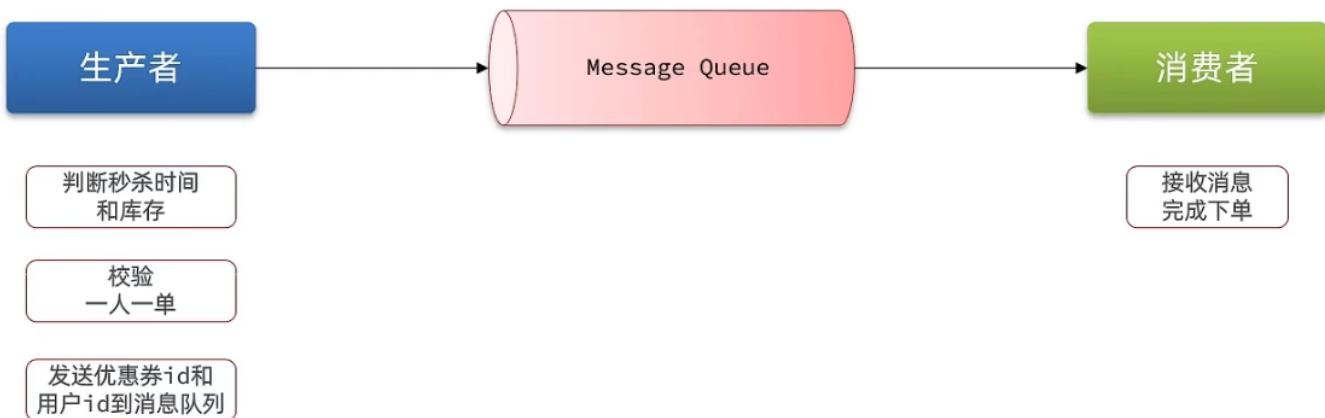
所以我们请出redis消息队列(^_)

redis消息队列

1、认识消息队列

什么是消息队列：字面意思就是存放消息的队列。最简单的消息队列模型包括3个角色：

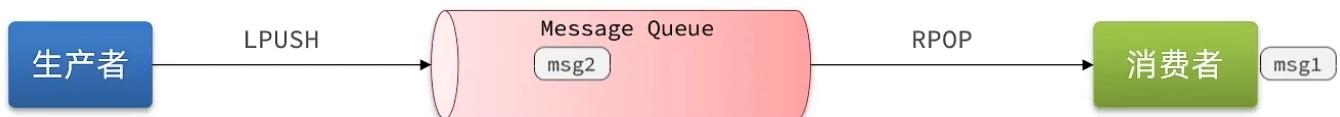
- 消息队列：存储和管理消息，也被称为消息代理（Message Broker）
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取消息并处理消息



使用队列的好处在于 **解耦**：**所谓解耦在我们秒杀中就是：我们下单之后，利用redis去进行校验下单条件，再通过队列把消息发送出去，然后再启动一个线程去消费这个消息，完成解耦，同时也加快我们的响应速度。

2、基于List实现消息队列

Redis的list数据结构是一个双向链表，很容易模拟出队列效果。



队列是入口和出口不在一边，因此我们可以利用：LPUSH 结合 RPOP、或者 RPUSH 结合 LPOP来实现。不过要注意的是，当队列中没有消息时RPOP或LPOP操作会返回null，并不像JVM的阻塞队列那样会阻塞并等待消息。因此这里应该使用BRPOP或者BLPOP来实现阻塞效果。

基于List的消息队列有哪些优缺点？优点：

- 利用Redis存储，不受限于JVM内存上限
- 基于Redis的持久化机制，数据安全性有保证
- 可以满足消息有序性

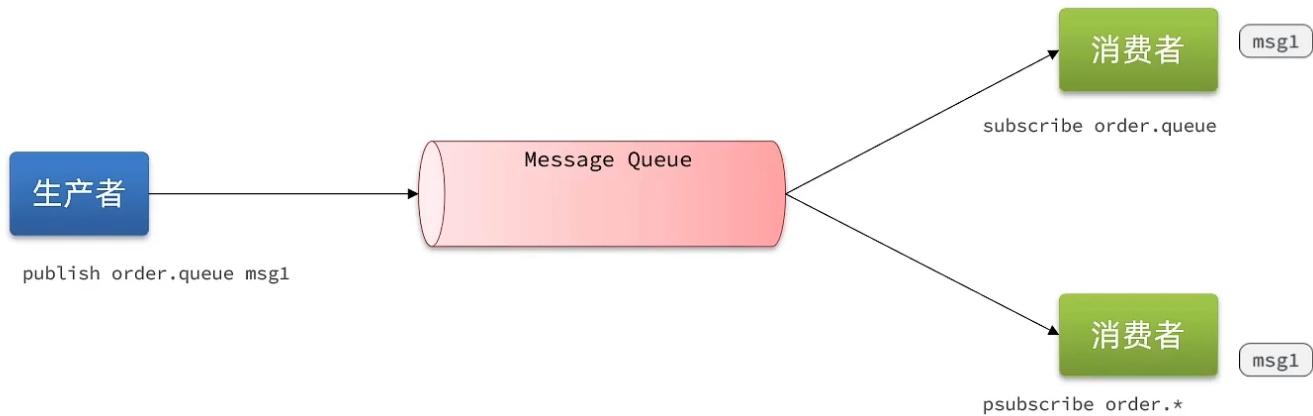
缺点：

- 无法避免消息丢失
- 只支持单消费者

3、基于PubSub的消息队列

消费者可以订阅一个或多个channel，生产者向对应channel发送消息后，所有订阅者都能收到相关消息。

SUBSCRIBE channel [channel]：订阅一个或多个频道
 PUBLISH channel msg：向一个频道发送消息
 PSUBSCRIBE pattern[pattern]：订阅与pattern格式匹配的所有频道



基于PubSub的消息队列有哪些优缺点？ 优点：

- 采用发布订阅模型，支持多生产、多消费

缺点：

- 不支持数据持久化
- 无法避免消息丢失
- 消息堆积有上限，超出时数据丢失

4、基于Stream的消息队列

发送消息：XADD

```
127.0.0.1:6379> help xadd
XADD key [NOMKSTREAM] [MAXLEN|MINID [=|~] threshold [LIMIT count]] *|ID field value [field value ...]
summary: Appends a new entry to a stream
since: 5.0.0
group: stream
```

消息的唯一id, *代表由Redis自动生成。格式是“时间戳-递增数字”，例如“1644804662707-0”

如果队列不存在，是否自动创建队列

设置消息队列的最大消息数量

发送到队列中的消息，称为Entry。格式就是多个key-value键值对

示例：返回时间戳和序号

```
## 创建名为 users 的队列，并向其中发送一个消息，内容是：{name=jack,age=21}，并且使用Redis自动生成ID
127.0.0.1:6379> XADD users * name jack age 21
"1644805700523-0"
```

读取消息：XREAD

```
127.0.0.1:6379> help XREAD
XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]
summary: Return never seen elements in multiple streams, with IDs greater than
the ones reported by the caller for each stream. Can block.
since: 5.0.0
group: stream
```

每次读取消息的最大数量

当没有消息时，是否阻塞、阻塞时长

要从哪个队列读取消息，key就是队列名

起始id，只返回大于该ID的消息
0：代表从第一个消息开始
\$：代表从最新的消息开始

示例

```
127.0.0.1:6379> XREAD COUNT 1 STREAMS users 0
1) 1) "users"
   2) 1) 1) "1644805700523-0"
      2) 1) "name"
         2) "jack"
         3) "age"
         4) "21"
```

XREAD阻塞方式，读取最新的消息：

```
127.0.0.1:6379> XREAD COUNT 1 BLOCK 1000 STREAMS users $
(nil)
(1.07s)
```

在业务开发中，我们可以循环的调用XREAD阻塞方式来查询最新消息，从而实现持续监听队列的效果，伪代码如下：

```
1 while(true){
2     // 尝试读取队列中的消息，最多阻塞2秒
3     Object msg = redis.execute("XREAD COUNT 1 BLOCK 2000 STREAMS users $");
4     if(msg == null){
5         continue;
6     }
7     // 处理消息
8     handleMessage(msg);
9 }
```

注意：当我们指定起始ID为\$时，代表读取最新的消息，如果我们处理一条消息的过程中，又有超过1条以上的消息到达队列，则下次获取时也只能获取到最新的一条，会出现漏读消息的问题

STREAM类型消息队列的XREAD命令特点：

- 消息可回溯
- 一个消息可以被多个消费者读取
- 可以阻塞读取
- 有消息漏读的风险

5、基于Stream的消息队列-消费者组

消费者组（Consumer Group）：将多个消费者划分到一个组中，监听同一个队列。具备下列特点：

01

消息分流

队列中的消息会分流给组内的不同消费者，而不是重复消费，从而加快消息处理的速度

02

消息标示

消费者组会维护一个标示，记录最后一个被处理的消息，哪怕消费者宕机重启，还会从标示之后读取消息。确保每一个消息都会被消费

03

消息确认

消费者获取消息后，消息处于 pending 状态，并存入一个 pending-list。当处理完成后需要通过 XACK 来确认消息，标记消息为已处理，才会从 pending-list 移除。

相关语法：

```
// 创建消费者组
XGROUP CREATE key groupName ID[MKSTREAM]
// - key: 队列名称
// - groupName: 消费者组名称
// - ID: 起始ID标示, $代表队列中最后一个消息, 0则代表队列中第一个消息
// - MKSTREAM: 队列不存在时自动创建队列

// 删除指定的消费者组
XGROUP DESTORY key groupName

// 给指定的消费者组添加消费者
XGROUP CREATECONSUMER key groupname consumername

// 删除消费者组中的指定消费者
XGROUP DELCONSUMER key groupname consumername

// 从消费者组读取消息
XREADGROUP GROUP group consumer [COUNT count] [BLOCK milliseconds] [NOACK] STREAMS
key [key ...] ID [ID ...]

ex: XPENDING mystream group55 - + 10 // - + 表示a11
// group: 消费组名称
// consumer: 消费者名称, 如果消费者不存在, 会自动创建一个消费者
// count: 本次查询的最大数量
// BLOCK milliseconds: 当没有消息时最长等待时间
// NOACK: 无需手动ACK, 获取到消息后自动确认【一般不配置】
// STREAMS key: 指定队列名称
// ID: 获取消息的起始ID:
    // ">": 从下一个未消费的消息开始
    // 其它: 根据指定id从pending-list中获取已消费但未确认的消息, 例如0, 是从pending-list
    中的第一个消息开始

// 确认消息
XACK key group id [id ...]
```

```
// 获取pending-list的消息
XPENDING key group [[IDLE min-idle-time] start end count [consumer]]
    // - IDLE: 确认时间
    // - start&end: 确认消息的起始和末尾
    // - count: 确认数量
```

消费者监听消息的基本思路：【伪代码】

```
1 while(true){
2     // 尝试监听队列，使用阻塞模式，最长等待 2000 毫秒
3     Object msg = redis.call("XREADGROUP GROUP g1 c1 COUNT 1 BLOCK 2000 STREAMS s1 >");
4     if(msg == null){ // null说明没有消息，继续下一次
5         continue;
6     }
7     try {
8         // 处理消息，完成后一定要ACK
9         handleMessage(msg);
10    } catch(Exception e){
11        while(true){
12            Object msg = redis.call("XREADGROUP GROUP g1 c1 COUNT 1 STREAMS s1 0");
13            if(msg == null){ // null说明没有异常消息，所有消息都已确认，结束循环
14                break;
15            }
16            try {
17                // 说明有异常消息，再次处理
18                handleMessage(msg);
19            } catch(Exception e){
20                // 再次出现异常，记录日志，继续循环
21                continue;
22            }
23        }
24    }
25 }
```

STREAM类型消息队列的XREADGROUP命令特点：

- 消息可回溯
- 可以多消费者争抢消息，加快消费速度
- 可以阻塞读取
- 没有消息漏读的风险
- 有消息确认机制，保证消息至少被消费一次

全部对比：【stream最好！】

	List	PubSub	Stream
消息持久化	支持	不支持	支持
阻塞读取	支持	支持	支持
消息堆积处理	受限于内存空间，可以利用多消费者加快处理	受限于消费者缓冲区	受限于队列长度，可以利用消费者组提高消费速度，减少堆积
消息确认机制	不支持	不支持	支持
消息回溯	不支持	不支持	支持

6、基于Redis的Stream结构作为消息队列，实现异步秒杀下单

需求：

- 创建一个Stream类型的消息队列，名为stream.orders
- 修改之前的秒杀下单Lua脚本，在认定有抢购资格后，直接向stream.orders中添加消息，内容包含 voucherId、userId、orderId
- 项目启动时，开启一个线程任务，尝试获取stream.orders中的消息，完成下单

代码实现：

1、seckill.lua

```
-- 秒杀优化需求二：基于Lua脚本，判断秒杀库存、一人一单，决定用户是否有购买资格
-- 1.参数列表
-- 1.1.优惠券id
local voucherId = ARGV[1]
-- 1.2.用户id
local userId = ARGV[2]
-- 1.3.订单id
local orderId = ARGV[3]

-- 2.数据key
-- 2.1.库存key ..lua的字符串拼接
local stockKey = 'seckill:stock:' .. voucherId
-- 2.2.订单key
local orderKey = 'seckill:order:' .. voucherId

-- 3.脚本业务
-- 3.1.判断库存是否充足 get stockKey tonumber()将字符串转换为数字
if(tonumber(redis.call('get', stockKey)) <= 0) then
    -- 3.2.库存不足，返回1
    return 1
end
-- 3.2.判断用户是否下单 SISMEMBER：判断set集合中是否存在某个元素
```

```

if(redis.call('sismember', orderKey, userId) == 1) then
    -- 3.3.存在, 说明是重复下单, 返回2
    return 2
end
-- 3.4.扣库存 incrby stockKey -1
redis.call('incrby', stockKey, -1)
-- 3.5.下单 (保存用户) sadd orderKey userId
redis.call('sadd', orderKey, userId)

return 0

```

2、VoucherOrderServiceImpl

```

private static final DefaultRedisScript<Long> SECKILL_SCRIPT;

static {
    SECKILL_SCRIPT = new DefaultRedisScript<>();
    SECKILL_SCRIPT.setLocation(new ClassPathResource("seckill.lua"));
    SECKILL_SCRIPT.setResultType(Long.class);
}

//异步处理线程池
private static final ExecutorService SECKILL_ORDER_EXECUTOR =
Executors.newSingleThreadExecutor();

IVoucherOrderService proxy;

//在类初始化之后执行, 因为当这个类初始化好了之后, 随时都是有可能要执行的
@PostConstruct
private void init() {
    SECKILL_ORDER_EXECUTOR.submit(new VoucherOrderHandler());
}

// 用于线程池处理的任务
// 当初始化完毕后, 就会去从对列中去拿信息
private class VoucherOrderHandler implements Runnable,
com.hmdp.service.impl.VoucherOrderHandler {
    @Override
    public void run() {
        while (true) {
            try {
                // 之前: 1.获取阻塞队列中的订单信息
                // VoucherOrder voucherOrder = orderTasks.take();
                // 2.创建订单
                // handleVoucherOrder(voucherOrder);

                // 现在: 1.获取消息队列中的订单信息 XREADGROUP GROUP g1 c1 COUNT
                1 BLOCK 2000 STREAMS s1 >
                    List<MapRecord<String, Object, Object>> list =
                    stringRedisTemplate.opsForStream().read(Consumer.from("g1", "c1"),

```

```
StreamReadOptions.empty().count(1).block(Duration.ofSeconds(2)),  
StreamOffset.create("stream.orders", ReadOffset.lastConsumed()));  
        // 2.判断订单信息是否为空  
        if (list == null || list.isEmpty()) {  
            // 如果为null, 说明没有消息, 继续下一次循环  
            continue;  
        }  
        // 解析数据 string就是消息id  
        MapRecord<String, Object, Object> record = list.get(0);  
        Map<Object, Object> value = record.getValue();  
        // 把map转成order对象  
        VoucherOrder voucherOrder = BeanUtil.fillBeanWithMap(value,  
new VoucherOrder(), true);  
        // 3.创建订单  
        createVoucherOrder(voucherOrder);  
        // 4.确认消息 XACK  
        stringRedisTemplate.opsForStream().acknowledge("s1", "g1",  
record.getId());  
  
    } catch (Exception e) {  
        log.error("处理订单异常", e);  
        // 处理异常消息  
        handlePendingList();  
    }  
}  
  
}  
  


```
private void handlePendingList() {
 while (true) {
 try {
 // 1.获取pending-list中的订单信息 XREADGROUP GROUP g1 c1 COUNT
1 BLOCK 2000 STREAMS s1 0
 // 注意这里是0
 List<MapRecord<String, Object, Object>> list =
stringRedisTemplate.opsForStream().read(Consumer.from("g1", "c1"),
StreamReadOptions.empty().count(1), StreamOffset.create("stream.orders",
ReadOffset.from("0")));
 // 2.判断订单信息是否为空
 if (list == null || list.isEmpty()) {
 // 如果为null, 说明没有异常消息, 结束循环
 break;
 }
 // 解析数据
 MapRecord<String, Object, Object> record = list.get(0);
 Map<Object, Object> value = record.getValue();
 VoucherOrder voucherOrder = BeanUtil.fillBeanWithMap(value,
new VoucherOrder(), true);
 // 3.创建订单
 createVoucherOrder(voucherOrder);
 // 4.确认消息 XACK
 stringRedisTemplate.opsForStream().acknowledge("s1", "g1",
record.getId());
 } catch (Exception e) {
 }
```


```

```
        log.error("处理pending订单异常", e);
        try {
            Thread.sleep(20);
        } catch (InterruptedException ex) {
            throw new RuntimeException(ex);
        }

    }
}

}

@Override
public Result seckillVoucher(Long voucherId) {
    Long userId = UserHolder.getUser().getId();
    // 订单id
    long orderId = redisIdWorker.nextId("order");
    // 1.执行lua脚本
    Long result = stringRedisTemplate.execute(SECKILL_SCRIPT,
Collections.emptyList(), voucherId.toString(), userId.toString(),
String.valueOf(orderId), String.valueOf(orderId));
    int r = result.intValue();
    // 2.判断结果是否为0
    if (r != 0) {
        // 2.1.不为0 , 代表没有购买资格
        return Result.fail(r == 1 ? "库存不足" : "不能重复下单");
    }
    // VoucherOrder voucherOrder = new VoucherOrder();
    // voucherOrder.setId(orderId);
    // // 2.4.用户id
    // voucherOrder.setUserId(userId);
    // // 2.5.代金券id
    // voucherOrder.setVoucherId(voucherId);
    // // 2.6.放入阻塞队列
    // orderTasks.add(voucherOrder);
    //3.获取代理对象
    proxy = (IVoucherOrderService) AopContext.currentProxy();
    //4.返回订单id
    return Result.ok(orderId);
}

@Transactional
public void createVoucherOrder(VoucherOrder voucherOrder) {
    Long userId = voucherOrder.getUserId();
    // 5.1.查询订单
    int count = query().eq("user_id", userId).eq("voucher_id",
voucherOrder.getVoucherId()).count();
    // 5.2.判断是否存在
    if (count > 0) {
        // 用户已经购买过了
        log.error("用户已经购买过了");
        return;
    }
}
```

```

    // 6. 扣减库存
    boolean success = seckillVoucherService.update().setSql("stock = stock -
1") // set stock = stock - 1
        .eq("voucher_id", voucherOrder.getVoucherId()).gt("stock", 0) //
where id = ? and stock > 0
        .update();
    if (!success) {
        // 扣减失败
        log.error("库存不足");
        return;
    }
    save(voucherOrder);

}
}

```

7、tip：保存1k个token到JMeter中

Test

```

/**
 * 在Redis中保存1000个用户信息并将其token写入文件中，方便测试多人秒杀业务
 */
@Test
void testMultiLogin() throws IOException {
    List<User> userList = userService.lambdaQuery().last("limit 1000").list();
    for (User user : userList) {
        String token = UUID.randomUUID().toString();
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new
HashMap<>(),
            CopyOptions.create().ignoreNullValue()
                .setFieldValueEditor((fieldName, fieldValue) ->
fieldValue.toString()));
        String tokenKey = LOGIN_USER_KEY + token;
        stringRedisTemplate.opsForHash().putAll(tokenKey, userMap);
        stringRedisTemplate.expire(tokenKey, 30, TimeUnit.MINUTES);
    }
    Set<String> keys = stringRedisTemplate.keys(LOGIN_USER_KEY + "*");
    @Cleanup FileWriter fileWriter = new
FileWriter(System.getProperty("user.dir") + "\\tokens.txt");
    @Cleanup BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
    assert keys != null;
    for (String key : keys) {
        String token = key.substring(LOGIN_USER_KEY.length());
        String text = token + "\n";
        bufferedWriter.write(text);
    }
}

```

JMeter配置如下，在CSV数据文件设置处设置token



注意HTTP信息头管理器也要把authorization换成\${token}变量



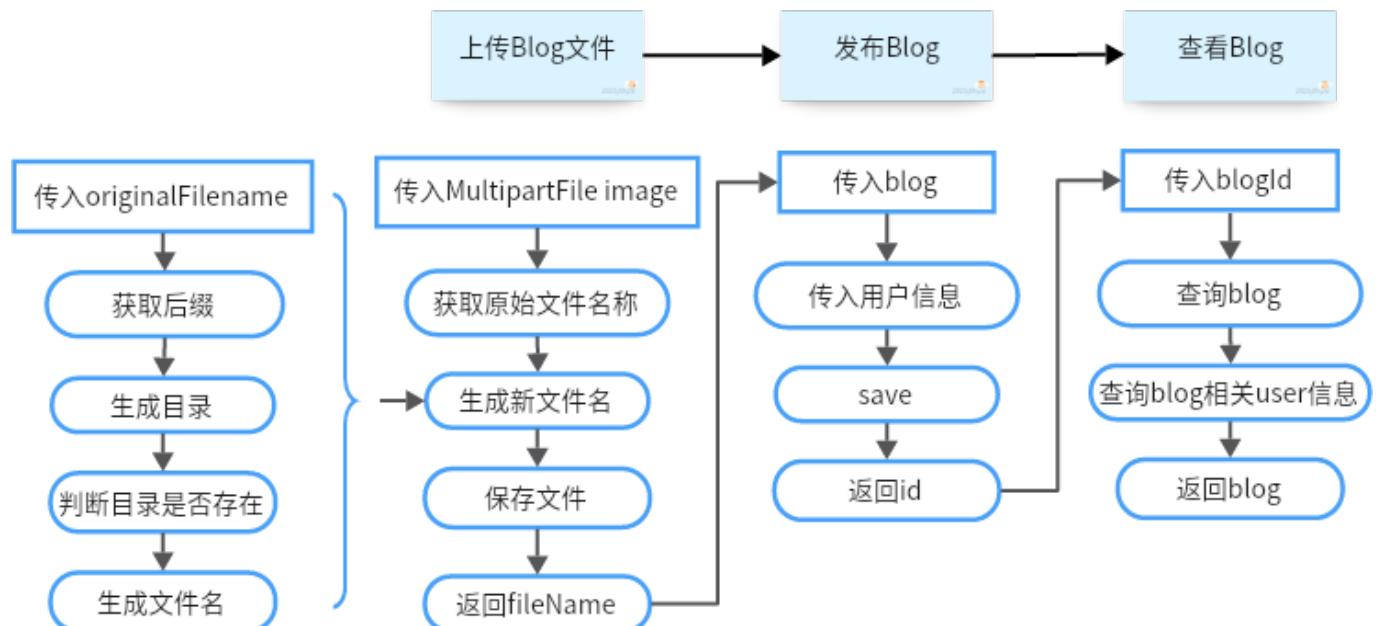
结果：实现了抢购且没有超卖等问题（但是好像也没有快很多Σ(口||)

Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 KB/...	发送 KB/...
HTTP请求	1000	405	363	799	893	989	41	1019	78.20%	962.5/sec	1956.11	57.58
总体	1000	405	363	799	893	989	41	1019	78.20%	962.5/sec	1956.11	57.58

达人探店

1、发布、查看探店笔记

业务逻辑：



代码实现:

上传blog文件: UploadController

```

@PostMapping("blog")
public Result uploadImage(@RequestParam("file") MultipartFile image) {
    try {
        // 获取原始文件名称
        String originalFilename = image.getOriginalFilename();
        // 生成新文件名
        String fileName = createNewFileName(originalFilename);
        // 保存文件
        image.transferTo(new File(SystemConstants.IMAGE_UPLOAD_DIR,
        fileName));
        // 返回结果
        log.debug("文件上传成功, {}", fileName);
        return Result.ok(fileName);
    } catch (IOException e) {
        throw new RuntimeException("文件上传失败", e);
    }
}

// createNewFileName 生成新的文件名
private String createNewFileName(String originalFilename) {
    // 获取后缀
    String suffix = StrUtil.subAfter(originalFilename, ".", true);
    // 生成目录
    String name = UUID.randomUUID().toString();
    int hash = name.hashCode();
    int d1 = hash & 0xF;
    int d2 = (hash >> 4) & 0xF;
    // 判断目录是否存在
    File dir = new File(SystemConstants.IMAGE_UPLOAD_DIR,
    StrUtil.format("/blogs/{}/{}/", d1, d2));
    if (!dir.exists()) {
}
  
```

```
        dir.mkdirs();
    }
    // 生成文件名
    return StrUtil.format("/blogs/{}/{}/{}.{}", d1, d2, name, suffix);
}
```

发布blog: BlogServiceImpl

```
@Override
public Result saveBlog(Blog blog) {
    if (blog.getShopId() == null || blog.getTitle() == null ||
blog.getContent() == null) {
        return Result.fail("提交前请把Blog全部信息填写完整(●'◡'●)");
    }
    // 1. 获取登录用户
    UserDTO user = UserHolder.getUser();
    blog.setUserId(user.getId());
    // 2.保存探店笔记
    boolean isSuccess = save(blog);
    if (!isSuccess) {
        return Result.fail("新增笔记失败!");
    }
    return Result.ok(blog.getId());
}
```

查看blog: BlogServiceImpl

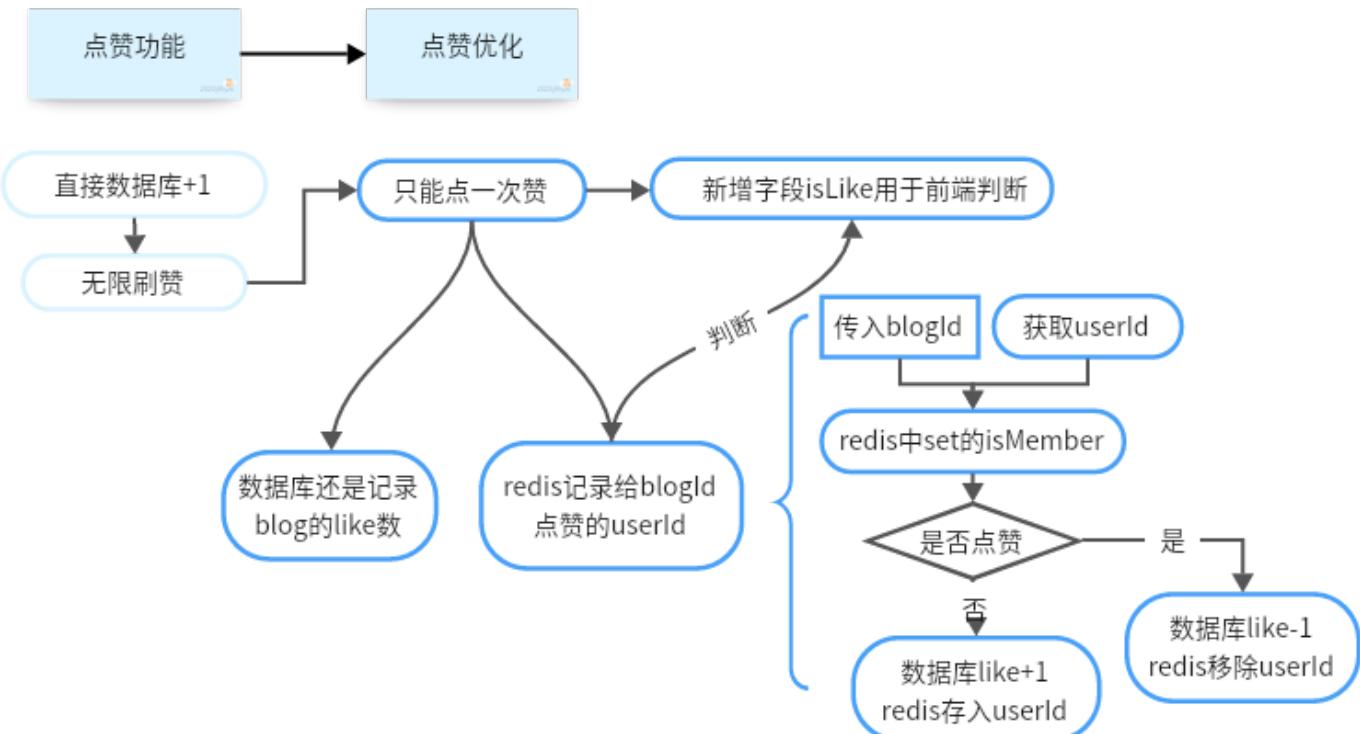
```
@Override
public Result queryBlogById(Long id) {
    Blog blog = this.getById(id);
    if (blog == null) {
        return Result.fail("笔记不存在! ");
    }
    // 因为blog表里面是没有username和usericon的，所以要再query
    queryBlogUser(blog);

    return Result.ok(blog);
}

// 查询blog有关的用户
private void queryBlogUser(Blog blog) {
    Long userId = blog.getUserId();
    User user = userService.getById(userId);
    blog.setName(user.getNickName());
    blog.setIcon(user.getIcon());
}
```

2、点赞笔记

业务逻辑:



代码实现:

1、Blog：添加字段

```

@TableField(exist = false)
private Boolean isLike;

```

2、BlogServiceImpl

```

@Override
public Result likeBlog(Long id){
    // 修改点赞数量,直接update数据库,但是这样可以无限刷赞
    // blogService.update().setSql("liked = liked + 1").eq("id", id).update();

    // 1.获取登录用户
    Long userId = UserHolder.getUser().getId();
    // 2.判断当前登录用户是否已经点赞
    String key = BLOG_LIKED_KEY + id;
    Boolean isMember = stringRedisTemplate.opsForSet().isMember(key,
    userId.toString());
    if(BooleanUtil.IsFalse(isMember)){
        //3.如果未点赞, 可以点赞
        //3.1 数据库点赞数+1
        boolean isSuccess = update().setSql("liked = liked + 1").eq("id",
        id).update();
        //3.2 保存用户到Redis的set集合
        if(isSuccess){
            stringRedisTemplate.opsForSet().add(key,userId.toString());
        }
    }
}

```

```

        }
    }else{
        //4.如果已点赞，取消点赞
        //4.1 数据库点赞数-1
        boolean isSuccess = update().setSql("liked = liked - 1").eq("id",
id).update();
        //4.2 把用户从Redis的set集合移除
        if(isSuccess){
            stringRedisTemplate.opsForSet().remove(key,userIds.toString());
        }
    }

private void isBlogLike(Blog blog) {
    // 1.获取登录用户
    UserDTO user = UserHolder.getUser();
    if (user == null) {
        // 用户未登录，无需查询是否点赞
        // 不然一点进去首页就会报错空指针，因为调用了isBlogLike方法，没登陆的话是没有userId的
        return;
    }
    String key = BLOG_LIKED_KEY + blog.getId();
    Boolean isMember = stringRedisTemplate.opsForSet().isMember(key,
user.getId().toString());
    blog.setIsLike(isMember = true);
}

```

3、点赞排行榜

redis使用：

1. 所有点赞的人，需要是唯一——>set或者是sortedSet
2. 需要排序，——>sortedSet

	List	Set	SortedSet
排序方式	按添加顺序排序	无法排序	根据score值排序
唯一性	不唯一	唯一	唯一
查找方式	按索引查找或首尾查找	根据元素查找	根据元素查找

业务逻辑：没有什么特别的捏

代码实现：

1、点赞逻辑代码修改：set修改为zset

```
@Override
public Result likeBlog(Long id) {
    // 修改点赞数量,直接update数据库,但是这样可以无限刷赞
    // blogService.update().setSql("liked = liked + 1").eq("id", id).update();

    // 1.获取登录用户
    Long userId = UserHolder.getUser().getId();
    // 2.判断当前登录用户是否已经点赞
    String key = BLOG_LIKED_KEY + id;

    // Boolean isMember = stringRedisTemplate.opsForSet().isMember(key,
    user.getId().toString());
    Double score = stringRedisTemplate.opsForZSet().score(key,
    userId.toString());
    if (score == null) {
        //3.如果未点赞,可以点赞
        //3.1 数据库点赞数+1
        boolean isSuccess = blogService.update().setSql("liked = liked +
1").eq("id", id).update();
        //3.2 保存用户到Redis的set集合
        if (isSuccess) {
            stringRedisTemplate.opsForZSet().add(key, userId.toString(),
System.currentTimeMillis());
        }
    } else {
        //4.如果已点赞,取消点赞
        //4.1 数据库点赞数-1
        boolean isSuccess = blogService.update().setSql("liked = liked -
1").eq("id", id).update();
        //4.2 把用户从Redis的set集合移除
        if (isSuccess) {
            stringRedisTemplate.opsForZSet().remove(key, userId.toString());
        }
    }
    return Result.ok();
}

private void isBlogLike(Blog blog) {
    // 1.获取登录用户
    UserDTO user = UserHolder.getUser();
    if (user == null) {
        // 用户未登录,无需查询是否点赞
        // 不然一点进去首页就会报错空指针,因为调用了isBlogLike方法,没登陆的话是没有userId的
        return;
    }
    String key = BLOG_LIKED_KEY + blog.getId();
    // Boolean isMember = stringRedisTemplate.opsForSet().isMember(key,
    user.getId().toString());
    Double score = stringRedisTemplate.opsForZSet().score(key,
    user.getId().toString());
```

```

        blog.setIsLike(score != null);
    }
}

```

2、按时间查询点赞用户列表

```

@Override
public Result queryBlogLikes(Long id) {
    String key = BLOG_LIKED_KEY + id;
    // 1.查询top5的点赞用户 zrange key 0 4
    Set<String> top5 = stringRedisTemplate.opsForZSet().range(key, 0, 4);
    if (top5 == null || top5.isEmpty()) {
        return Result.ok(Collections.emptyList());
    }
    // 2.解析出其中的用户id
    List<Long> ids =
    top5.stream().map(Long::valueOf).collect(Collectors.toList());
    String idStr = StrUtil.join(",", ids);

    // 3.根据用户id查询用户 WHERE id IN ( 5 , 1 ) ORDER BY FIELD(id, 5, 1)
    Stream<UserDTO> userDTOS = userService.query().in("id", ids)
        // 注意这里的ORDER BY FIELD, 这样才能实现数据库查询也按照自己想要的顺序排序
        .last("ORDER BY FIELD(id," + idStr + ")")
        .list().stream().map(user -> BeanUtil.copyProperties(user,
    UserDTO.class));
    // 4.返回
    return Result.ok(userDTOS);
}

```

Tips: 注意controller的return不用再加Result.ok(BlogServiceImpl.xxx(xxx))了，直接return BlogServiceImpl.xxx(xxx)

前面一种返回值如下，前端是无法读取到的

```

▼ {success: true, data: {success: true,...}}
  ▼ data: {success: true,...}
    ▼ data: [{id: 4, shopId: 4, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不
      ▶ 0: {id: 4, shopId: 4, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不
      ▶ 1: {id: 5, shopId: 1, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不
      ▶ 2: {id: 6, shopId: 10, userId: 1, icon: "/imgs/blogs/blog1.jpg", name: "小鱼同学",...
      ▶ 3: {id: 7, shopId: 10, userId: 1, icon: "/imgs/blogs/blog1.jpg", name: "小鱼同学",...
      ▶ 4: {id: 23, shopId: 1, userId: 1011, icon: "", name: "炫仔捏", title: "好好好",...}
      ▶ 5: {id: 24, shopId: 1, userId: 1011, icon: "", name: "炫仔捏", title: "1", images:
        success: true
      success: true
    
```

后面一种的返回值如下，才能正常读取

```
▼ {success: true,...}
  ▼ data: [{id: 4, shopId: 4, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不吃|",...}
    ▶ 0: {id: 4, shopId: 4, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不吃|",...}
    ▶ 1: {id: 5, shopId: 1, userId: 2, icon: "/imgs/icons/kkjtbcr.jpg", name: "可可今天不吃|",...}
    ▶ 2: {id: 6, shopId: 10, userId: 1, icon: "/imgs/blogs/blog1.jpg", name: "小鱼同学",...}
    ▶ 3: {id: 7, shopId: 10, userId: 1, icon: "/imgs/blogs/blog1.jpg", name: "小鱼同学",...}
    ▶ 4: {id: 23, shopId: 1, userId: 1011, icon: "", name: "炫仔捏", title: "好好好",...}
    ▶ 5: {id: 24, shopId: 1, userId: 1011, icon: "", name: "炫仔捏", title: "1", images: ""}

  success: true
```

set改成zset的时候一直报错: WRONGTYPE Operation against a key holding the wrong kind of value

```
2023-09-24 13:00:34.394 ERROR 27012 --- [nio-8081-exec-1] com.hmdp.config.WebExceptionAdvice : org.springframework.data.redis.RedisSystemException: Error in execution; nested exception is io.lettuce.core.RedisCommandExecutionException: WRONGTYPE Operation against a key holding the wrong kind of value
```

一脸懵，就是DOUBLE类型啊，gpt也笨笨的找不到原因

csdn了一下，是因为存取的类型不一致导致，之前只存了一个member，现在还存了score，直接把之前的key删掉重新加载一下就行

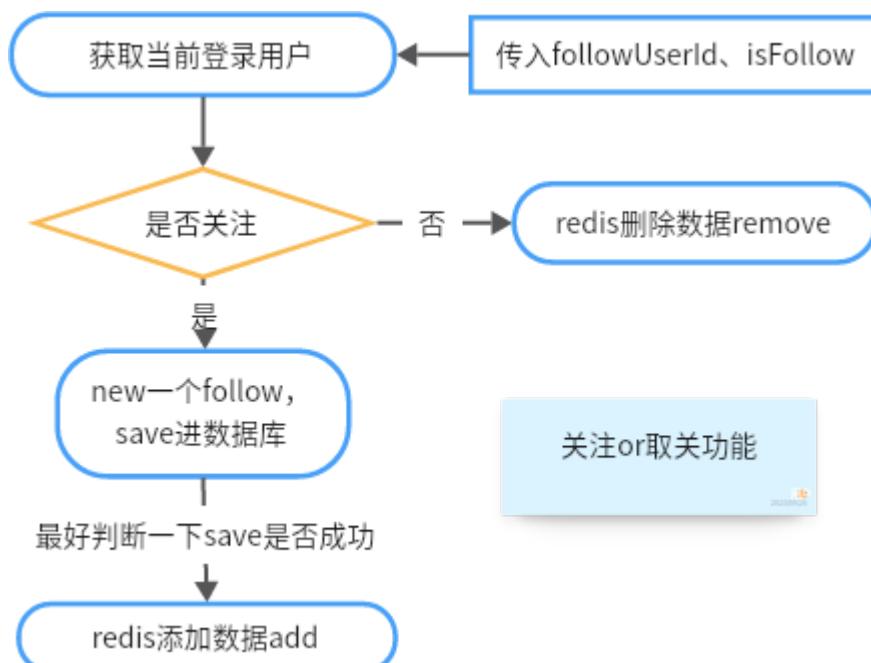
好友关注

1、实现关注/取关

前端请求: /follow/2/true --> 后端: @PutMapping("/{id}/{isFollow}")

redis使用：使用set的add方法，key为follow:user:userId，value为followerId

业务逻辑：



代码实现：FollowServiceImpl

```

@Override
public Result follow(Long followUserId, Boolean isFollow) {
    // 1. 获取当前登录用户
    long userId = UserHolder.getUser().getId();
    String key = FOLLOW_USER_KEY + userId;

    // 2. 判断是否关注
    if (isFollow) {
        // 未关注, 进行关注操作
        Follow follow = new
Follow().setUserId(userId).setFollowUserId(followUserId).setCreateTime(LocalDateTi
me.now());
        boolean isSuccess = save(follow);
        if (isSuccess) {
            // 成功的话就放在redis里面
            stringRedisTemplate.opsForSet().add(key, followUserId.toString());
        } else {
            return Result.fail("关注失败, 请稍后再试! ");
        }
    } else {
        // 关注, 进行取关操作 :数据库--redis
        // remove(new QueryWrapper<Follow>().eq("user_id",
userId).eq("follow_user_id", followUserId));
        stringRedisTemplate.opsForSet().remove(key, followUserId.toString());
    }
    return Result.ok();
}

```

2、查看是否关注

前端请求: `follow/or/not/1` --> 后端: `@GetMapping("or/not/{id}")`

redis使用: set的`isMember`方法, 判断是否存在某个value

代码实现: `FollowServiceImpl`

```

@Override
public Result isFollow(Long followUserId) {
    // 1. 获取当前登录用户
    long userId = UserHolder.getUser().getId();
    // 2. 从数据库中查是否有关注数据 --> 从redis中查
    // Integer count = query().eq("user_id", userId).eq("follow_user_id",
followUserId).count();
    Boolean member = stringRedisTemplate.opsForSet().isMember(FOLLOW_USER_KEY
+ userId, followUserId.toString());
    return Result.ok(member);
}

```

3、查看共同关注

前端请求: `follow/common/1` --> 后端: `@GetMapping("/common/{id}")`

redis使用: set的`intersect`方法, 查询交集

难点:

1. 获取的交集是Set类型的, 如何转换成List类型方便后续查询具体用户信息时调用

```
// 解析id集合
// .stream() 方法将集合 intersect 转换为一个 Java 8 流 (Stream)
// .map(Long::valueOf) 是一个映射操作, 它将集合中的每个元素从字符串转换为 Long 类型。
// .collect(Collectors.toList()) 是一个收集操作, 它将流中的元素收集到一个新的 List 集合中。这个新的 List 中包含了经过映射后的 Long 类型的元素。
List<Long> ids = intersect
    .stream()
    .map(Long::valueOf)
    .collect(Collectors.toList());
```

2. 获取的List怎么转换成List

```
// 查询用户
// listByIds返回的是一个List<User>, 所以需要进行转换
// .stream() 方法将集合 ids 转换为一个 Java 8 流 (Stream)
// .map(user -> BeanUtil.copyProperties(user, UserDTO.class)) 是一个映射操作, 它将集合中的每个元素从 User 类型转换为 UserDTO 类型。
// .collect(Collectors.toList()) 将流中的映射后的 UserDTO 对象收集到一个新的 List<UserDTO> 中。这个操作将流转换为一个列表, 其中包含了经过映射后的 UserDTO 对象。
List<UserDTO> users = userService
    .listByIds(ids)
    .stream()
    .map(user -> BeanUtil.copyProperties(user, UserDTO.class))
    .collect(Collectors.toList());
```

代码实现: FollowServiceImpl

```
@Override
public Result followCommons(Long otherUserId) {
    // 1. 获取当前登录用户
    long userId = UserHolder.getUser().getId();
    String userKey = FOLLOW_USER_KEY + userId;
    String otherKey = FOLLOW_USER_KEY + otherUserId;

    //求交集
    Set<String> intersect = stringRedisTemplate.opsForSet().intersect(userKey,
otherKey);
    // 没有的话直接返回空列表
    if (intersect == null || intersect.isEmpty()) {
        return Result.ok(Collections.emptyList());
    }
}
```

```
// 解析id集合
List<Long> ids =
intersect.stream().map(Long::valueOf).collect(Collectors.toList());

// 查询用户
List<UserDTO> users = userService.listByIds(ids).stream().map(user ->
BeanUtil.copyProperties(user, UserDTO.class)).collect(Collectors.toList());
return Result.ok(users);
}
```

4、关注推送

Feed流实现方案

Feed流产品有两种常见模式：

1. Timeline：不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注。例如朋友圈

优点 信息全面，不会有缺失。并且实现也相对简单

缺点 信息噪音较多，用户不一定感兴趣，内容获取效率低

2. 智能排序：利用智能算法屏蔽掉违规的、用户不感兴趣的内容。推送用户感兴趣信息来吸引用户

优点 投喂用户感兴趣信息，用户粘度很高，容易沉迷

缺点 如果算法不精准，可能起到反作用

采用Timeline的模式。该模式的实现方案有三种：

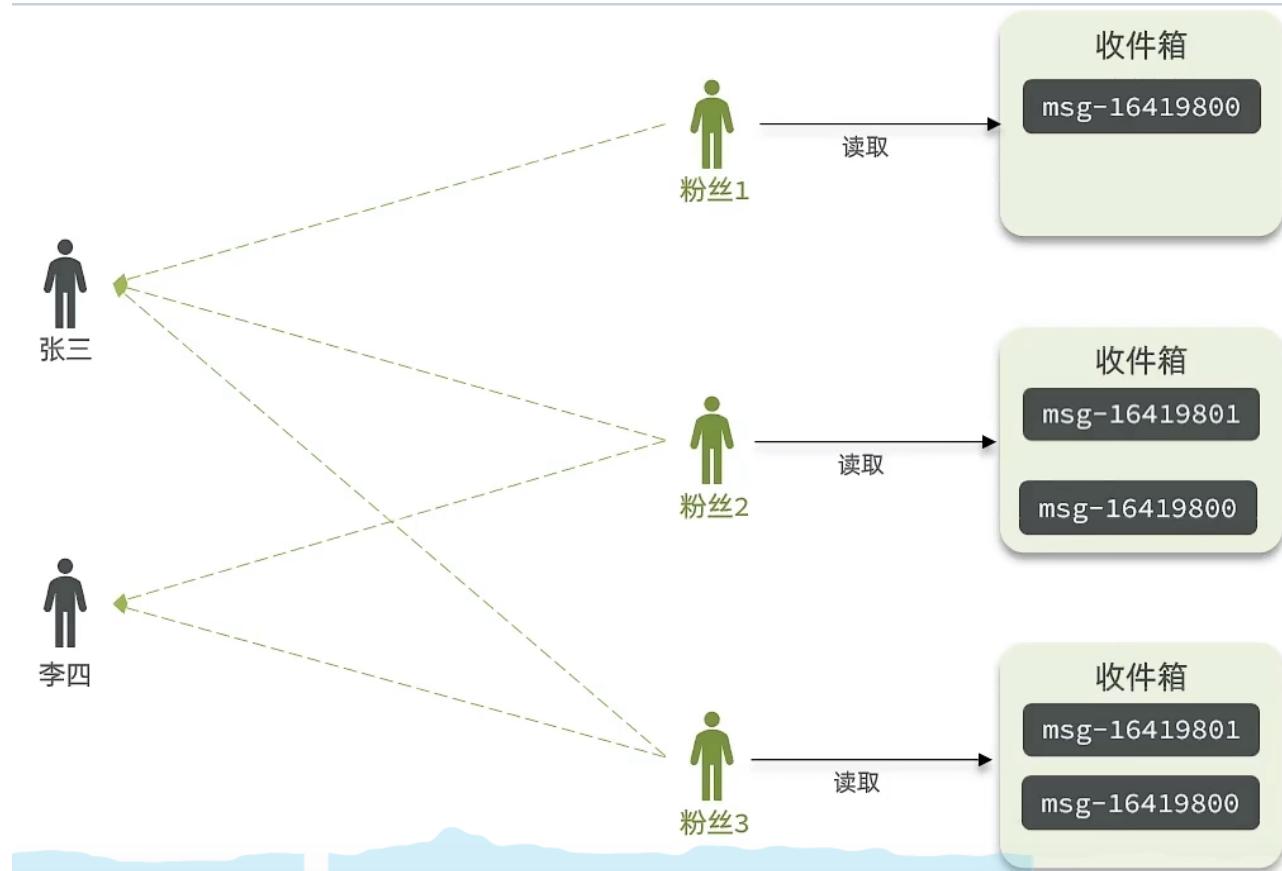
1. 拉模式：只有粉丝用户在读取收件箱的时候，才会根据其关注的用户进行拉取，把博主发件箱里的消息拉取到粉丝用户的收件箱里，然后对收件箱里的消息按时间戳进行排序。

优点：节约空间

缺点：比较延迟，假设用户关注了大用户，此时就会拉取海量的内容，对服务器压力巨大。

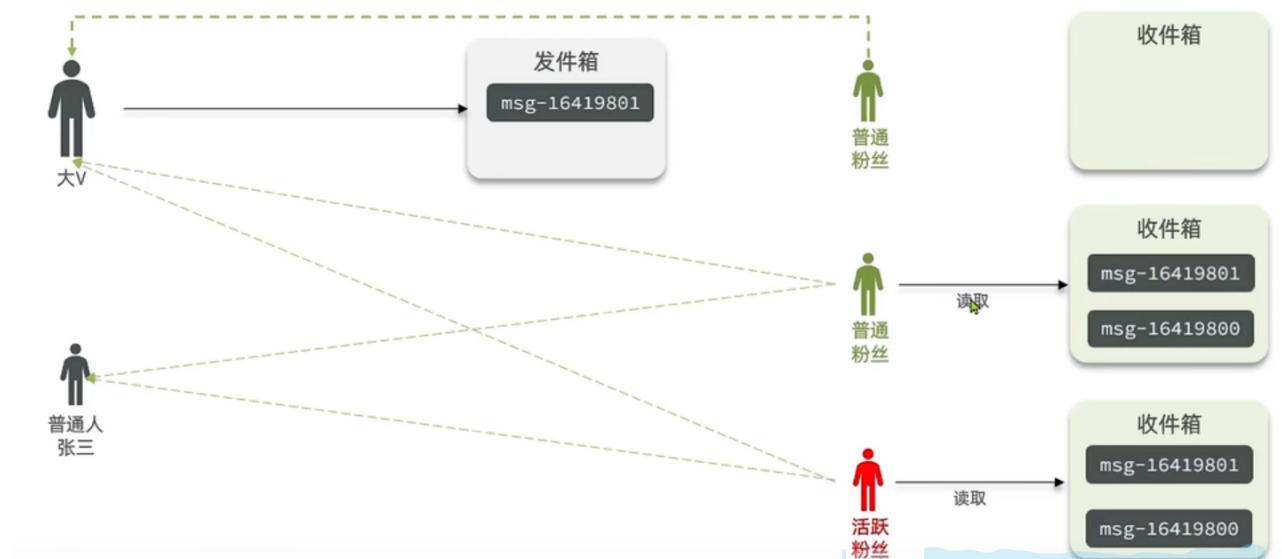


2. 推模式：当用户（博主）发送消息时，会把消息+时间戳直接发送到所有粉丝用户的收件箱中，并按时间戳进行排序。当粉丝用户在读取收件箱的消息时，直接读取。优点：延迟低 缺点：发消息时，内容占用较高。因为每个粉丝都会保留一份消息。



3. 推拉模式：对于粉丝少的博主用户，采用推模式。对于粉丝多的博主用户，根据粉丝用户类型进行判断：活跃度高的粉丝用户，采用推模式 活跃度低的粉丝用户，采用拉模式

兼具推和拉两种模式的优点



4. 总结

	拉模式	推模式	推拉结合
写比例	低	高	中
读比例	高	低	中
用户读取延迟	高	低	低
实现难度	复杂	简单	很复杂
使用场景	很少使用	用户量少、没有大V	过千万的用户量，有大V

最后俺们选择推模式（因为实现起来比较简单）

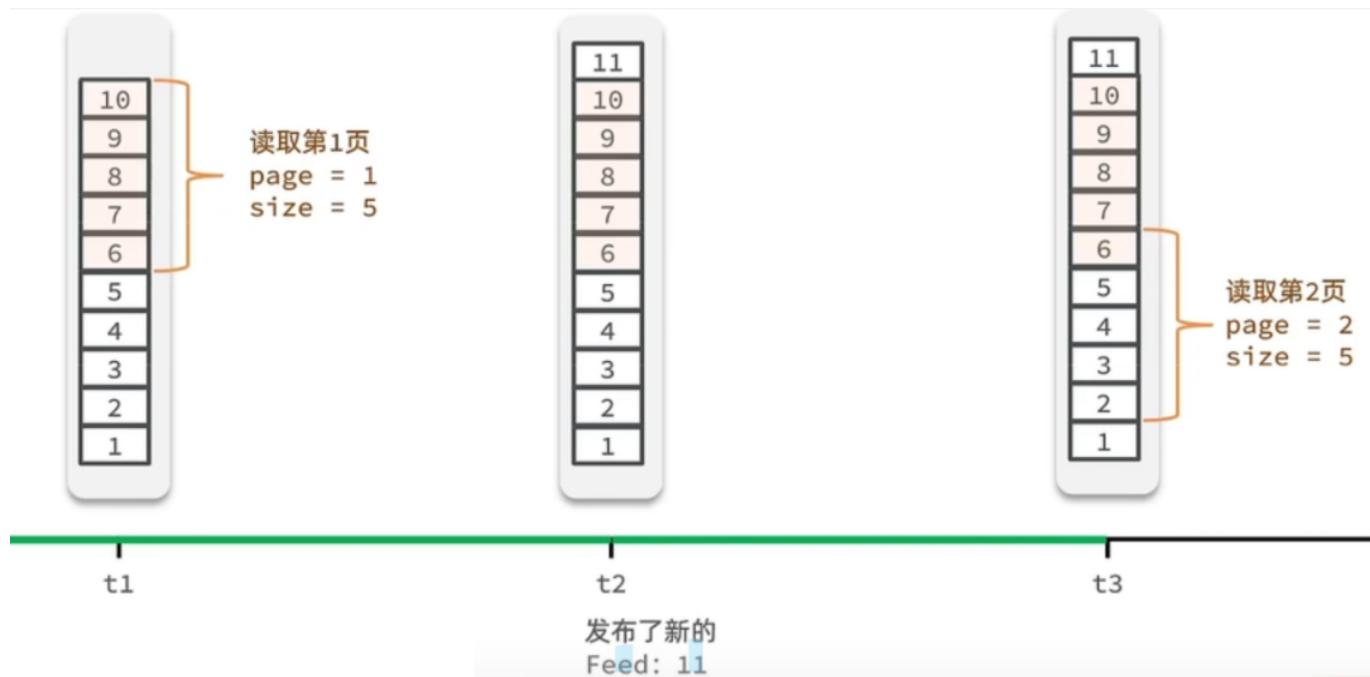
推送到粉丝收件箱

需求：

- 修改新增探店笔记的业务，在保存blog到数据库的同时，推送到粉丝的收件箱
 - 收件箱满足可以根据时间戳排序，必须用Redis的数据结构实现
 - 查询收件箱数据时，可以实现分页查询

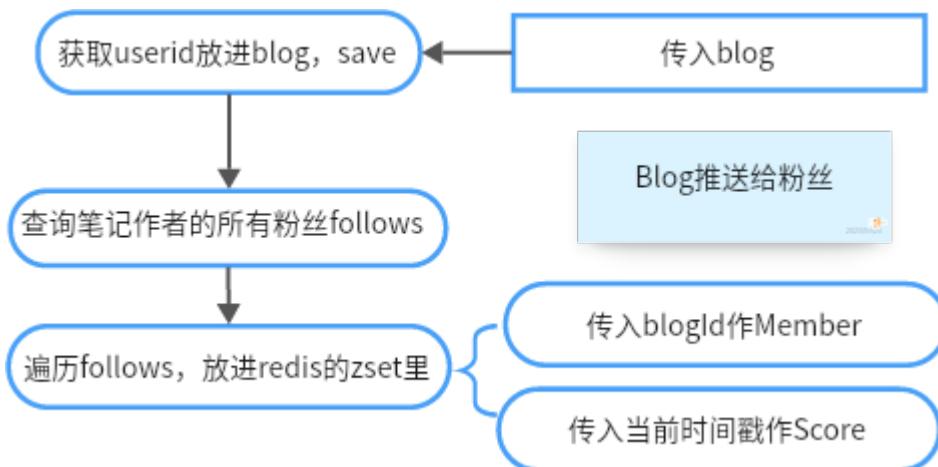
难点：

当粉丝用户需要按分页模式来读取收件箱的信息时，不能采用传统的分页模式（按数据的角标开始查）。因为 Feed 流中的数据会不断更新，所以数据的角标也在不断变化。传统的分页模式，会出现消息重复读的问题。



业务逻辑：

因为是推模式，所以我们是按照接受Blog的用户作为key



代码实现：BlogServiceImpl

```

@Override
public Result saveBlog(Blog blog) {
    if (blog.getShopId() == null || blog.getTitle() == null ||
    blog.getContent() == null) {
        return Result.fail("提交前请把Blog全部信息填写完整(●'◡'●)");
    }
    // 1. 获取登录用户
    UserDTO user = UserHolder.getUser();
    blog.setUserId(user.getId());
    // 2.保存探店笔记
    boolean isSuccess = save(blog);
    if (!isSuccess) {
        return Result.fail("新增笔记失败!");
    }
    // 3.查询笔记作者的所有粉丝
    List<Follow> follows = followService.query().eq("follow_user_id",
user.getId()).list();
    // 4.推送笔记id给所有粉丝
    for (Follow follow : follows) {
        // 4.1.获取粉丝id
        Long userId = follow.getUserId();
        // 4.2.推送 (思路就是只把blog的id传到redis里面, 到时候再调用blog的query方法获取详情)
        String key = FEED_KEY + userId;
        // 还是要按时间戳当作value, 因为要进行排序
        stringRedisTemplate.opsForZSet().add(key, blog.getId().toString(),
System.currentTimeMillis());
    }
    return Result.ok(blog.getId());
}
  
```

redis数据

The screenshot shows the Redis desktop manager interface. On the left, under the database 'DB0', there's a tree view with nodes: 'feed:1' (1 item), 'feed:2' (2 items), 'follow' (1 item), 'user' (1 item), 'follow:user:1011' (1 item), and 'shop' (2 items). A red box highlights 'feed:2' and a red arrow points from it to the main table view on the right. The table view shows a Zset named 'feed:2' with TTL set to -1. It contains one item with ID 1, score 1695742425001, and member 'blogId'. There are buttons for '向' (left), '向' (right), and '</>'.

实现分页查询收邮箱

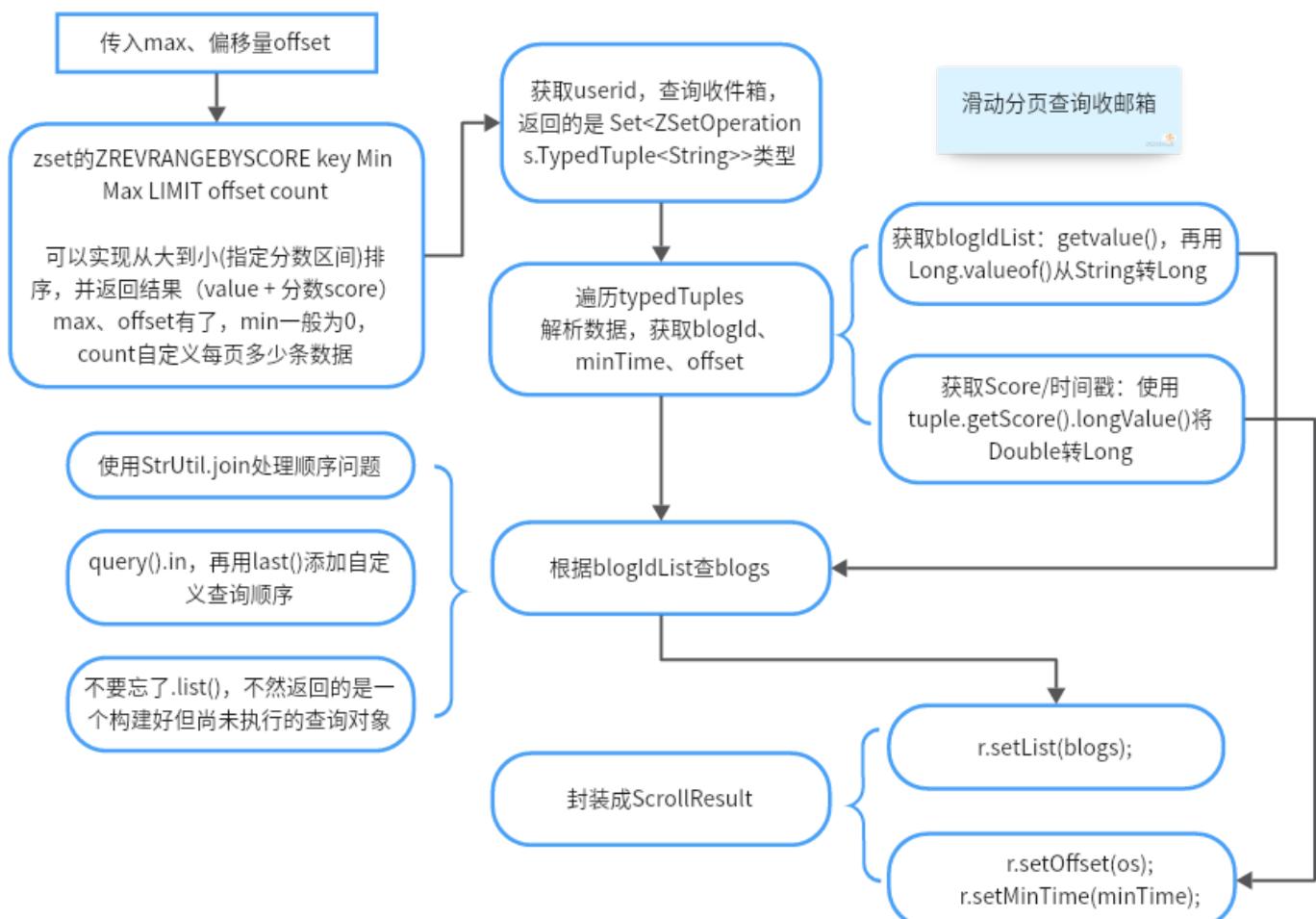
具体操作如下：

- 1、每次查询完成后，我们要分析出查询出数据的最小时间戳，这个值会作为下一次查询的条件
- 2、我们需要找到与上一次查询相同的查询个数作为偏移量，下次查询时，跳过这些查询过的数据，拿到我们需要的数据

综上：我们的请求参数中就需要携带 lastId：上一次查询的最小时间戳 和偏移量这两个参数。

这两个参数第一次会由前端来指定，以后的查询就根据后台结果作为条件，再次传递到后台。

业务逻辑：



代码实现:

ScrollResult

```
@Data  
public class ScrollResult {  
    private List<?> list;  
    //  
    private Long minTime;  
    private Integer offset;  
}
```

BlogServiceimpl

```
@Override  
public Result queryBlogOfFollow(Long max, Integer offset) {  
    // 1. 获取登录用户  
    Long userId = UserHolder.getUser().getId();  
  
    // 2. 查询自己的收件箱  
    String key = FEED_KEY + userId;  
    Set<ZSetOperations.TypedTuple<String>> typedTuples =  
    stringRedisTemplate.opsForZSet().reverseRangeByScoreWithScores(key, 0, max,  
    offset, 2);  
    if (typedTuples == null || typedTuples.isEmpty()) {  
        return Result.ok(Collections.emptyList());  
    }  
  
    // 解析数据: blogId、minTime (时间戳)、offset  
    ArrayList<Long> ids = new ArrayList<>(typedTuples.size());  
    long minTime = 2;  
    int os = 1;  
    for (ZSetOperations.TypedTuple<String> tuple : typedTuples) {  
        ids.add(Long.valueOf(tuple.getValue()));  
        long time = tuple.getScore().longValue();  
        if (time == minTime) {  
            os++;  
        } else {  
            minTime = time;  
            os = 1;  
        }  
    }  
    os = minTime == max ? os : os + offset;  
    // 根据id查blog  
    String idStr = StrUtil.join(",", ids);  
    List<Blog> blogs = query().in("id", ids).last("ORDER BY FIELD(id," + idStr  
+ ")").list();  
    // 查询blog相关信息  
    for (Blog blog : blogs) {  
        queryBlogUser(blog);  
    }  
}
```

```

        isBlogLike(blog);
    }
    // 封装并返回
    ScrollResult r = new ScrollResult();
    r.setList(blogs);
    r.setOffset(os);
    r.setMinTime(minTime);

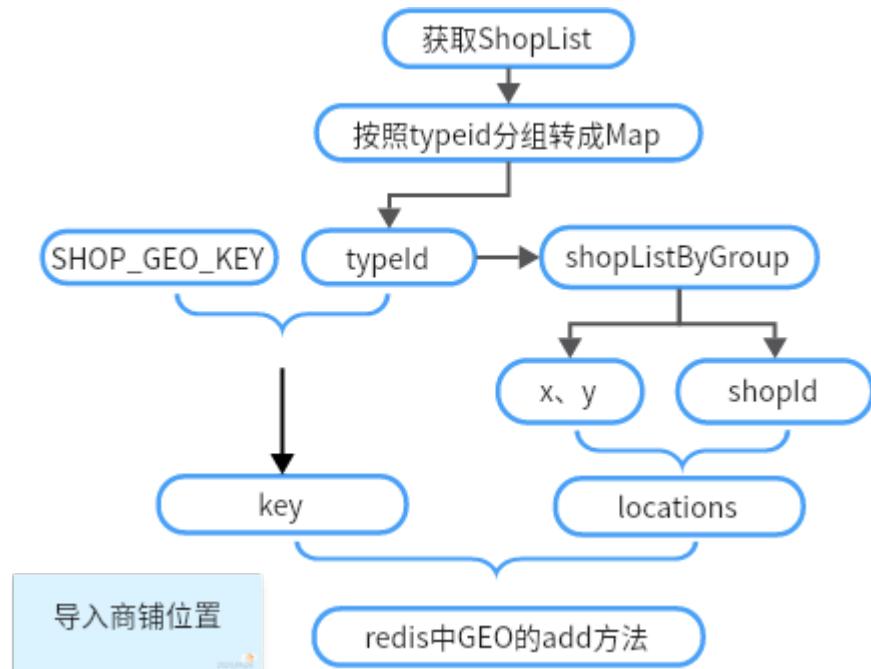
    return Result.ok(r);
}

```

附近商户

1、导入商铺位置

业务逻辑



代码实现：

```

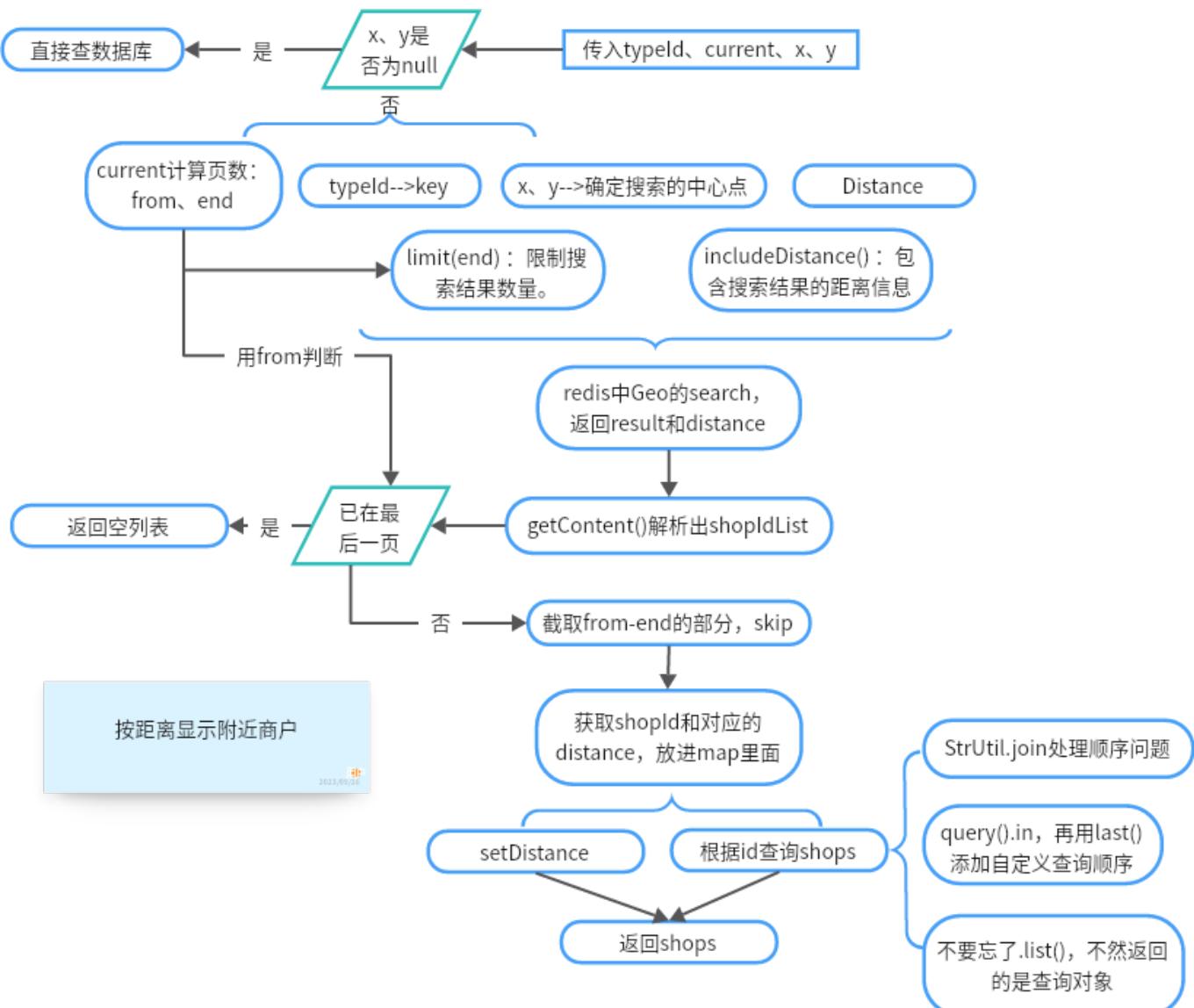
@Test
public void loadShopData() {
    // 1.查询店铺信息
    List<Shop> list = shopService.list();
    // 2.把店铺分组，按照typeId分组，typeId一致的放到一个集合
    Map<Long, List<Shop>> map =
list.stream().collect(Collectors.groupingBy(Shop::getTypeId));
    // 3.分批完成写入Redis
    for (Map.Entry<Long, List<Shop>> entry : map.entrySet()) {
        // 3.1.获取类型id
        Long typeId = entry.getKey();
        String key = SHOP_GEO_KEY + typeId;
        // 3.2.获取同类型的店铺的集合
        List<Shop> value = entry.getValue();

```

```
        List<RedisGeoCommands.GeoLocation<String>> locations = new ArrayList<>(value.size());
        // 3.3.写入redis GEOADD key 经度 纬度 member
        for (Shop shop : value) {
            // stringRedisTemplate.opsForGeo().add(key, new Point(shop.getX(),
            shop.getY(), shop.getId().toString());
            locations.add(new RedisGeoCommands.GeoLocation<>(
                shop.getId().toString(),
                new Point(shop.getX(), shop.getY())
            ));
        }
        stringRedisTemplate.opsForGeo().add(key, locations);
    }
}
```

2、附近商户

业务逻辑：



代码实现：

```
@Override
public Result queryShopByType(Integer typeId, Integer current, Double x,
Double y) {
    // 1. 判断是否需要根据坐标查询
    if (x == null || y == null) {
        // 不需要坐标查询, 按数据库查询
        Page<Shop> page = query()
            .eq("type_id", typeId)
            .page(new Page<>(current, SystemConstants.DEFAULT_PAGE_SIZE));
        // 返回数据
        return Result.ok(page.getRecords());
    }

    // 2. 计算分页参数
    int from = (current - 1) * SystemConstants.DEFAULT_PAGE_SIZE;
    int end = current * SystemConstants.DEFAULT_PAGE_SIZE;

    // 3. 查询redis、按照距离排序、分页。结果: shopId、distance
    String key = SHOP_GEO_KEY + typeId;
    GeoResults<RedisGeoCommands.GeoLocation<String>> results =
stringRedisTemplate.opsForGeo() // GEOSEARCH key BYLONLAT x y BYRADIUS 10
WITHDISTANCE
        .search(
            key,
            GeoReference.fromCoordinate(x, y),
            new Distance(5000),

RedisGeoCommands.GeoSearchCommandArgs.newGeoSearchArgs().includeDistance().limit(e
nd)
        );
    // 4. 解析出id
    if (results == null) {
        return Result.ok(Collections.emptyList());
    }
    List<GeoResult<RedisGeoCommands.GeoLocation<String>>> list =
results.getContent();
    if (list.size() <= from) {
        // 没有下一页了, 结束
        return Result.ok(Collections.emptyList());
    }
    // 4.1. 截取 from ~ end的部分
    List<Long> ids = new ArrayList<>(list.size());
    Map<String, Distance> distanceMap = new HashMap<>(list.size());
    list.stream().skip(from).forEach(result -> {
        // 4.2. 获取店铺id
        String shopIdStr = result.getContent().getName();
        ids.add(Long.valueOf(shopIdStr));
        // 4.3. 获取距离
        Distance distance = result.getDistance();
        distanceMap.put(shopIdStr, distance);
    });
    // 5. 根据id查询Shop
    String idStr = StrUtil.join(",", ids);
```

```

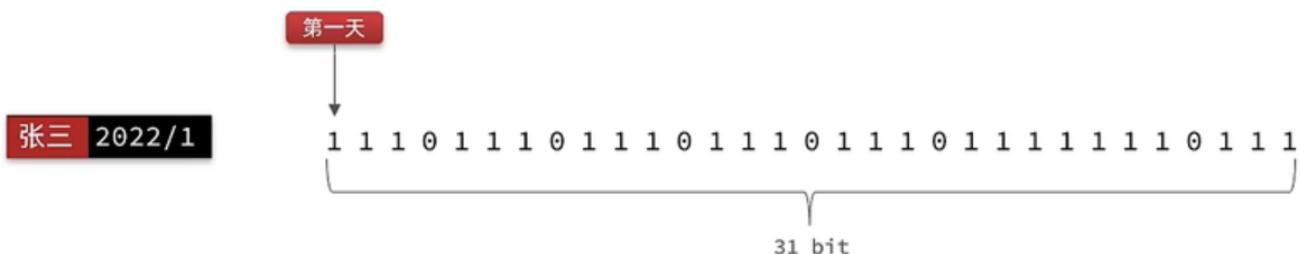
        List<Shop> shops = query().in("id", ids).last("ORDER BY FIELD(id," + idStr
+ ")").list();
        for (Shop shop : shops) {
            shop.setDistance(distanceMap.get(shop.getId().toString()).getValue());
        }
        // 6. 返回
        return Result.ok(shops);
    }
}

```

用户签到

1、实现签到功能

Redis中是利用string类型数据结构实现BitMap，因此最大上限是512M，转换为bit则是 2^{32} 个bit位。

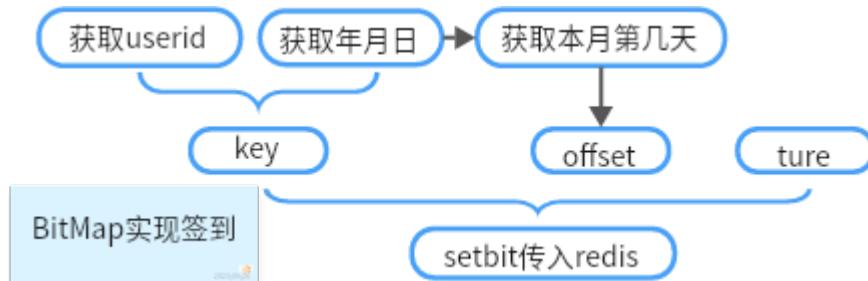


BitMap的操作命令有：

- SETBIT：向指定位置 (offset) 存入一个0或1
- GETBIT：获取指定位置 (offset) 的bit值
- BITCOUNT：统计BitMap中值为1的bit位的数量
- BITFIELD：操作（查询、修改、自增）BitMap中bit数组中的指定位置 (offset) 的值
- BITFIELD_RO：获取BitMap中bit数组，并以十进制形式返回
- BITOP：将多个BitMap的结果做位运算（与、或、异或）
- BITPOS：查找bit数组中指定范围内第一个0或1出现的位置

redis使用：使用BitMap的setbit方法

业务逻辑：



代码实现：

```

@Override
public Result sign() {
}

```

```

// 1.获取当前登录用户
Long userId = UserHolder.getUser().getId();
// 2.获取日期
LocalDateTime now = LocalDateTime.now();
// 3.拼接key
String keySuffix = now.format(DateTimeFormatter.ofPattern(":yyyyMM"));
String key = USER_SIGN_KEY + userId;
key += keySuffix;
// 4.获取今天是本月的第几天
int dayOfMonth = now.getDayOfMonth();
// 5.写入Redis SETBIT key offset 1
stringRedisTemplate.opsForValue().setBit(key, dayOfMonth - 1, true);
return Result.ok();
}

```

2、签到统计

redis使用: bitField方法 --> 操作 (查询、修改、自增) BitMap中bit数组中的指定位置 (offset) 的值

业务逻辑:



代码实现:

```

@Override
public Result signCount() {
    // 1.获取当前登录用户
    Long userId = UserHolder.getUser().getId();
    // 2.获取日期
    LocalDateTime now = LocalDateTime.now();
    // 3.拼接key
    String keySuffix = now.format(DateTimeFormatter.ofPattern(":yyyyMM"));
    String key = USER_SIGN_KEY + userId + keySuffix;
    // 4.获取今天是本月的第几天
}

```

```
int dayOfMonth = now.getDayOfMonth();

// 5. 获取本月截止今天为止的所有签到记录，返回的是一个十进制的数字 BITFIELD
sign:5:202203 GET u14 0
// bitField(key, BitFieldSubCommands.create()): 这是使用 Redis 的
BITFIELD 命令来进行位域操作的部分。它接受一个键 (key) 以及一个位域子命令
(BitFieldSubCommands.create())。
// BitFieldSubCommands.create(): 这是一个用于创建位域子命令的工厂方法。
// .get(BitFieldSubCommands.BitFieldType.unsigned(dayOfMonth)): 这一部分定义了要获取的位域。
// BitFieldSubCommands.BitFieldType.unsigned(dayOfMonth) 指示要获取的位域
类型是无符号整数，dayOfMonth 是一个表示具体位域位置的变量或常量。
// .valueAt(0): 这一部分指示要获取位域的位置，这里是位域中的第一个位（索引为
0）。
// 作用是从指定的 Redis 键 (key) 中获取位域中的某个位的值，位域类型为无符号整数
(unsigned)，位域的位置是位域中的第一个位（索引为0）。获取的结果将被存储在一个
List<Long> 中，并且该 List 中的每个元素对应于位域中的一个位的值。
// 涉及到多个位，需要返回一个List<Long>，其中每个Long表示一个位的状态
List<Long> result = stringRedisTemplate
    .opsForValue()
    .bitField(key, BitFieldSubCommands.create()
        // 获取多少位

.get(BitFieldSubCommands.BitFieldType.unsigned(dayOfMonth))
    // 从左到右取
    .valueAt(0));
System.out.println("result ===== " + result);

if (result == null || result.isEmpty()) {
    // 没有任何签到结果
    return Result.ok(0);
}
Long num = result.get(0);
if (num == null || num == 0) {
    return Result.ok(0);
}
// 6. 循环遍历
int count = 0;
while (num > 0) {
    // 6.1. 让这个数字与1做与运算，得到数字的最后一个bit位 // 判断这个bit位是否
    为0
    if ((num & 1) == 0) {
        break;
    } else {
        count++;
    }
    // 把数字右移一位，抛弃最后一个bit位，继续下一个bit位
    num >>>= 1;
    System.out.println("num ===== " + num);
}
return Result.ok(count);
}
```

UV统计

- UV：全称Unique Visitor，也叫独立访客量，是指通过互联网访问、浏览这个网页的自然人。1天内同一个用户多次访问该网站，只记录1次。
- PV：全称Page View，也叫页面访问量或点击量，用户每访问网站的一个页面，记录1次PV，用户多次打开页面，则记录多次PV。往往用来衡量网站的流量。

Hyperloglog(HLL)是从Loglog算法派生的概率算法，用于确定非常大的集合的基数，而不需要存储其所有值。
相关算法原理大家可以参考：<https://juejin.cn/post/6844903785744056333#heading-0>

Redis中的HLL是基于string结构实现的，单个HLL的内存永远小于16kb，内存占用低的令人发指！作为代价，其测量结果是概率性的，有小于0.81%的误差。不过对于UV统计来说，这完全可以忽略。

1、测试百万数据的统计

代码实现：HmDianPingApplicationTests

```

    @Test
    void testHyperLogLog() {
        String[] values = new String[1000];
        int count = 0;
        for (int i = 0; i < 1000000; i++) {
            count = i % 1000;
            values[count] = "user_" + i;
            if (count == 999) {
                //发送到redis
                stringRedisTemplate.opsForHyperLogLog().add("testHyperLogLog",
values);
            }
        }
        //统计数量
        Long res =
        stringRedisTemplate.opsForHyperLogLog().size("testHyperLogLog");
        log.debug("数量为: {}", res);
    }
}

```

结果：百万数据才十几kb捏o(—▽—)d

The screenshot displays two panels. On the left, a memory analysis tool shows a table of keys and their sizes. The key 'testHyperLogLog' is highlighted with a red box and has a size of 14.06KB. On the right, a terminal window shows application logs. It includes the memory analysis output and the application's startup logs, which mention the use of a concurrent ExecutorService and the execution of the test. The log also shows the debug message '数量为: 997593' (Quantity is: 997593), which corresponds to the size of the HLL set.

Key	Size
1. testHyperLogLog	14.06KB
2. shop:geo:1	175B
3. login_user_846792c5d...	136B
4. shop:geo:2	129B
5. follow:user:1011	76B
6. feed:1	72B
7. feed:2	72B

tips

alt+j 多选快捷键 Ctrl+Alt+V 自动补全快捷键 ctrl+alt+L 格式化代码 ctrl+O 重写方法 ctrl+shift+U 大小写转换