Comments项目

概述

做一个类似于大众点评的项目,可以实现用户的登录,注册,商家的注册以及发布用户对商家的点评等的一个点评平台

用户部分

登录

登录使用session来实现,获取验证码将用户保存在session中

登录校验

使用拦截器对用户的登录状态统一进行校验

```
// 拦截器的书写,学习一下
public class LoginInterceptor implements HandlerInterceptor {
    /**
     * 前置拦截器,对用户的登录状态进行校验
     * @param request
     * @param response
     * @param handler
     * @return 校验成功与否的布尔值
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        HttpSession session = request.getSession();
        User user = (User) session.getAttribute("user");
        // 校验失败,session中没有user信息
        if (user == null){
            response.setStatus(401);
            return false;
        }
        // 校验成功,将信息保存到ThreadLocal中方便后续的取出
        UserHolder.saveUser(user);
        return true;
    }

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

拦截器做完之后还需要写一个配置类来开启拦截器

```
@Configuration
public class MvcConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())
                .excludePathPatterns(
                        "/user/code",
                        "/user/login",
                        "/blog/hot",
                        "/shop/**",
                        "/shop-type/**",
                        "/upload/**",
                        "/voucher/**"
                );

    }
}
```

后面发现使用session不利于后面的横向发展微服务业务,所以将保存的session中的验证码以及用户user改为保存在redis中,更好的实现了可扩展性.

登录成功后将创建一个随机token,这个token需要发送回给前端保存起来,并且对于其他需要校验登录的请求都需要携带这个token才能进行身份校验,相当于一个身份令牌,实现过程就是以token为key,user为值保存在redis中,每次登录校验就去redis根据key获取值,如果获取不到则相当于没有登录或者是身份过期了,需要重新登录.

为了实现这个,我们需要设置一个拦截器来进行身份校验,以及进行token的过期刷新(30分钟没有任何请求才过期,如果其中有任何请求过来则刷新过期时间为30分钟)

```
// 这个拦截器进行过期时间的刷新
@Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

        // 获取请求头中的token
        String token = request.getParameter("authorization");
        // 没有token直接返回错误
        if (StrUtil.isBlankIfStr(token)) {
            return true;
        }
        // 从redis获取这个token对应的map
        Map<Object, Object> userMap =
stringRedisTemplate.opsForHash().entries(LOGIN_USER_KEY + token);
        // 找不到或为空直接返回错误
        if (userMap.isEmpty()){
            return true;
        }
        // 将map转换为对象
        UserDTO user = BeanUtil.fillBeanWithMap(userMap,new UserDTO(),false);
        // 刷新token过期时间
        stringRedisTemplate.expire(LOGIN_USER_KEY + token,LOGIN_USER_TTL,
TimeUnit.MINUTES);
        // 校验成功,将信息保存到ThreadLocal中方便后续的取出
        UserHolder.saveUser(user);
        return true;
```

```
    }
```

```
// 第二个拦截器进行身份的校验
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        // 派单人是否需要拦截(ThreadLocal中是否有用户)
        if (UserHolder.getUser() == null) {
            response.setStatus(401);
            return false;
        }
        return true;
    }
```

使用redis进行数据缓存

在我们进行数据查询的时候,可以先去查询redis中是否存在该数据,如果不存在再去查询数据库,如果数据库存在的话将该数据保存在redis中,这样可以大大减少进入数据库查询的次数,加快访问时间.比如没加redis查询是1s,加了redis是34ms

```
@Override
public Result queryById(Long id) {
    // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
    String shopCache = stringRedisTemplate.opsForValue().get(CACHE_SHOP_KEY +
id);
    //redis中存在,直接返回
    if (StrUtil.isNotBlank(shopCache)) {
        // 存在后将json数据转成shop对象
        Shop shop = JSONUtil.toBean(shopCache, Shop.class);
        return Result.ok(shop);
    }
    //redis中不存在,去查询数据库
    Shop shop = shopMapper.getShopById(id);
    //数据库中不存在,返回错误店铺不存在
    if (shop == null){
        return Result.fail("店铺不存在!!!");
    }
    //数据库中存在

    //店铺添加到redis中
    String shopJson = JSONUtil.toJsonStr(shop);
    stringRedisTemplate.opsForValue().set(CACHE_SHOP_KEY + id,shopJson);
    //返回店铺数据
    return Result.ok(shop);
}
```

使用redis缓存在的问题

数据一致性问题

即数据库中的数据和缓存中的数据不一致的问题

在高并发的场景下有可能出现

解决该问题的最佳实践方案:

1. 低一致性需求(即对更新操作不频繁的): 使用redis自带的缓存淘汰机制
2. 高一致性需求(更新需要及时的): 主动更新(即手动写代码更新数据库和redis),并以超时剔除作为兜底的方案

在进行更新操作时先进行写数据库操作,再删除缓存,这样的话在高并发场景下出错的概率比删除缓存再写数据库的出错概率低.

然后进行更新后不需要自己手动在将新数据写进到redis缓存中,等待有查询操作进来查询到redis中没有该数据再然其自动写入就好,这样可以避免在更新操作多读操作少的时候对redis的频繁更新

```
// 手动进行更新动作,同时添加一个@Transactional注解添加方法的事务保证方法的原子性
// 如果是分布式项目的话可能需要使用seata来实现事务
@Override
    @Transactional(rollbackFor = Exception.class)
    public Result updateShop(Shop shop) {
        if (shop.getId() == null) {
            return Result.fail("店铺信息更新失败,没有店铺ID!!");
        }
        updateById(shop);
        // 删除缓存
        stringRedisTemplate.delete(CACHE_SHOP_KEY + shop.getId());

        return Result.ok("更新店铺数据成功!!");
    }
```

缓存穿透问题

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时,这样缓存永远不会生效,这些请求通通都会打到数据库中,造成数据库的负担大大增加

常见解决方法:

1. 缓存空对象

即在第一次请求的时候查询不到数据就将该请求的id放置的redis中设置一个null值,这个请求就在redis中解决了不会到达数据库中.

这种实现方法简单,维护方法

缺点是: 会造成额外的内存消耗,而且可能造成短期内数据的不一致(可以通过设置空值缓存的ttl来降低这个缺点)

2. 布隆过滤

即在客户端请求和redis中添加一层布隆过滤器,请求来了之后先过过滤器,如果过滤器中没有的就直接返回.

布隆过滤器的具体实现比较难,redis有现成的布隆过滤器可以使用

优点:内存占用少,没有多余的key

缺点: 实现复杂,存在误判的可能(拒绝就是真没有,通过是不一定有)

```
// 解决缓存穿透后的方法
@Override
public Result queryById(Long id) {
    String key = CACHE_SHOP_KEY + id;
    // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
    String shopCache = stringRedisTemplate.opsForValue().get(key);
    //redis中存在,直接返回
    if (StrUtil.isNotBlank(shopCache)) {
        // 存在后将json数据转成shop对象
        Shop shop = JSONUtil.toBean(shopCache, Shop.class);
        return Result.ok(shop);
    }
    // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
    if (shopCache != null) {
        return Result.fail("该店铺不存在!!");
    }
    //redis中不存在,去查询数据库
    Shop shop = shopMapper.getShopById(id);
    //数据库中不存在,返回错误店铺不存在
    if (shop == null) {
        // 解决缓存穿透问题,向redis添加空值信息
        stringRedisTemplate.opsForValue().set(key, "", CACHE_NULL_TTL,
TimeUnit.MINUTES);
        return Result.fail("店铺不存在!!!");
    }
    //数据库中存在

    //店铺添加到redis中
    String shopJson = JSONUtil.toJsonStr(shop);
    stringRedisTemplate.opsForValue().set(key, shopJson, CACHE_SHOP_TTL,
TimeUnit.MINUTES);
    //返回店铺数据
    return Result.ok(shop);
}
```

缓存雪崩问题

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,导致大量的请求到达数据库,带来巨大的压力

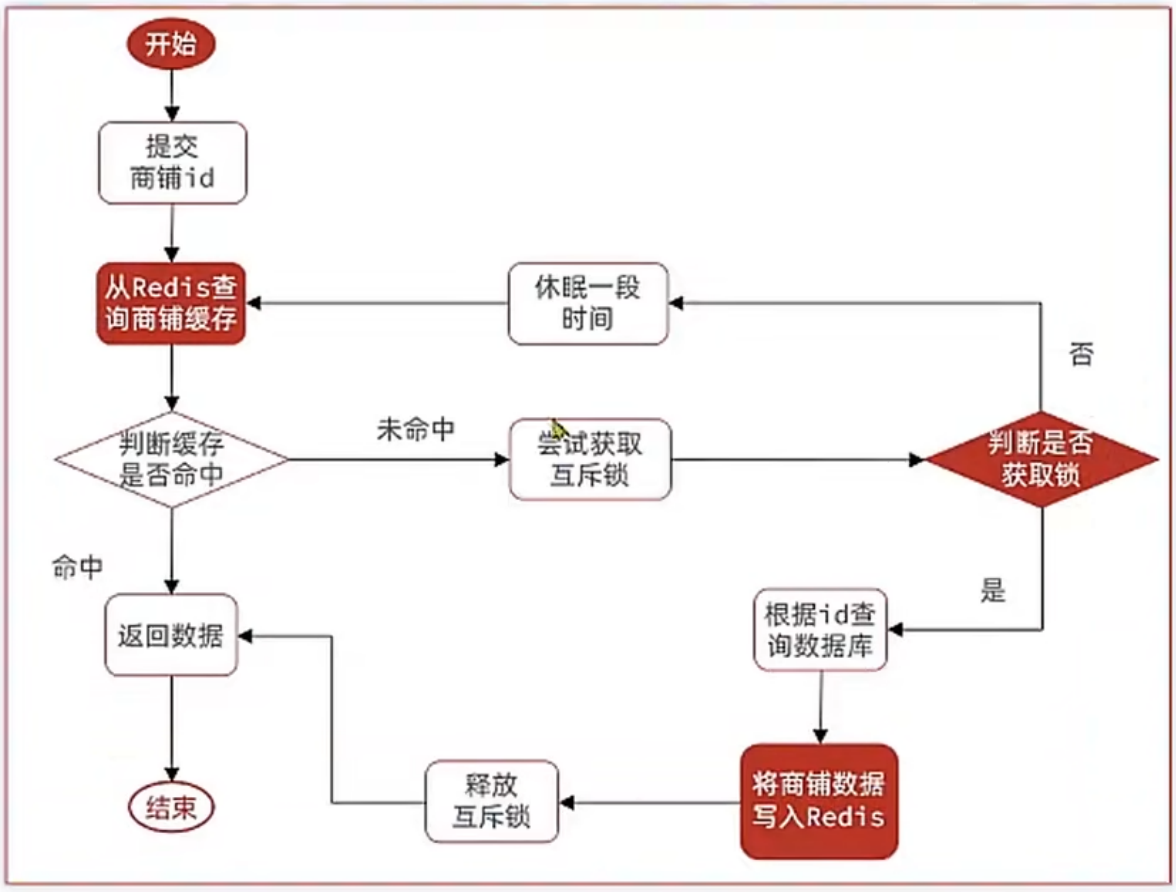
解决方案:

1. 给不同的key的ttl添加随机值
2. 利用redis集群提高服务的可用性
3. 给缓存业务添加将极限流策略
4. 给业务添加多级缓存

缓存击穿问题

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重建业务比较复杂的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

基于互斥锁解决缓存击穿问题



首先编写两个方法获取锁和释放锁

这里的锁是使用redis来保存的,可以做到分布式也能用的锁.

其中获取锁的函数使用BooleanUtil来进行拆包是因为如果直接返回Boolean类型数据让java自动拆包的话有可能有空指针问题

Boolean.true.equal...

```
/**
 * 获取锁,使用redis中的setnx功能,如果redis中已经存在该键则返回false,不存在则创建并返回true
 * @param key 保存在redis中的key
 * @return 返回boolean表示是否获取到了锁
 */
private boolean tryLock(String key){
    Boolean flag = stringRedisTemplate.opsForValue().setIfAbsent(key, "1", 10,
TimeUnit.SECONDS);
    return BooleanUtil.isTrue(flag);
}

/**
 * 释放锁
 * @param key 释放的锁的key
 */
```

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

解决了缓存击穿问题的query

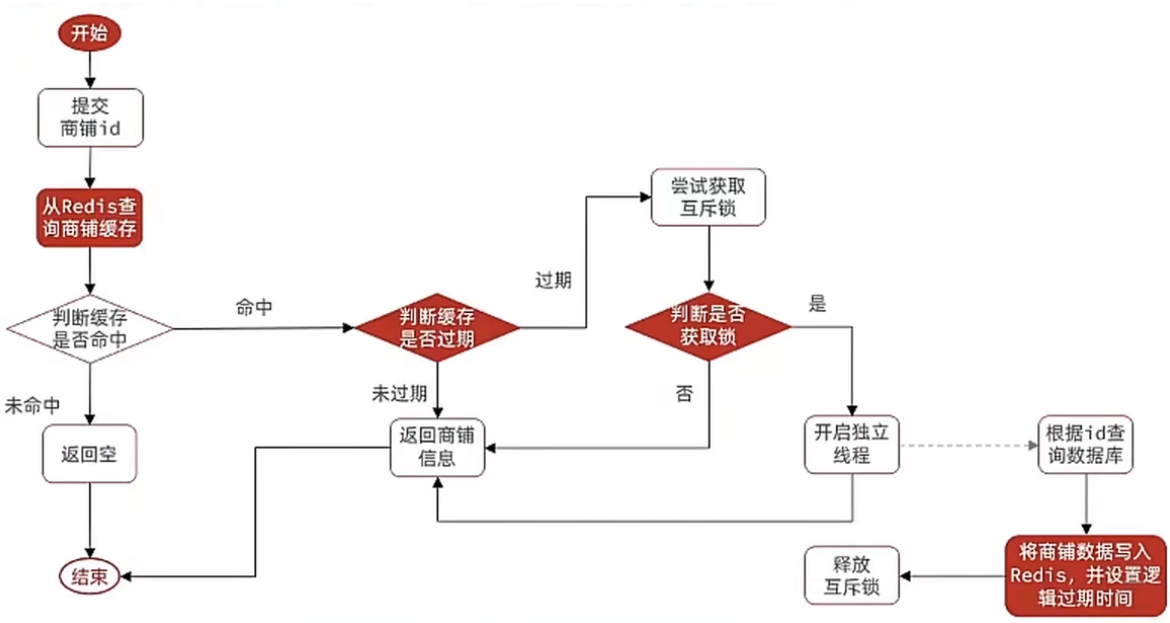
其中值得注意的是锁的获取以及等待的过程和其中的一个DoubleCheck,最后记得释放锁

```
/**
 * 解决了缓存击穿和缓存穿透问题的获取店铺信息的方法的备份
 * @param id 获取店铺的id
 * @return 返回店铺的详细信息
 */
private Shop queryWithMutex(Long id) {
    String key = CACHE_SHOP_KEY + id;
    String lock = LOCK_SHOP_KEY + id;
    Shop shop = null;
    try {
        // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
        String shopCache = stringRedisTemplate.opsForValue().get(key);
        //redis中存在,直接返回
        if (StrUtil.isNotBlank(shopCache)) {
            // 存在后将json数据转成shop对象
            Shop shop = JSONUtil.toBean(shopCache, Shop.class);
            return shop;
        }
        // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
        if (shopCache != null) {
            return null;
        }
        // redis中不存在
        // 获取锁
        // 获取不到锁,则递归获取数据,在这里想当与等待
        if (!tryLock(lock)) {
            Thread.sleep(50);
            return queryWithMutex(id);
        }
        // 获取到锁
        // 进行一个DoubleCheck,以防其他的线程修改完了我们再获取
        shopCache = stringRedisTemplate.opsForValue().get(key);
        //redis中存在,直接返回
        if (StrUtil.isNotBlank(shopCache)) {
            // 存在后将json数据转成shop对象
            Shop shop = JSONUtil.toBean(shopCache, Shop.class);
            return shop;
        }
        // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
        if (shopCache != null) {
            return null;
        }
        // redis中的确不存在该数据,则查询数据库
        shop = shopMapper.getShopById(id);
        //数据库中不存在,返回错误店铺不存在
        if (shop == null) {
            // 解决缓存穿透问题,向redis添加空值信息
```

```
            stringRedisTemplate.opsForValue().set(key, "", CACHE_NULL_TTL,
TimeUnit.MINUTES);
            return null;
        }
        //数据库中存在

        //店铺添加到redis中
        String shopJson = JSONUtil.toJsonStr(shop);
        stringRedisTemplate.opsForValue().set(key, shopJson, CACHE_SHOP_TTL,
TimeUnit.MINUTES);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        // 释放锁
        unlock(lock);
    }
    return shop;
}
```

基于逻辑过期解决缓存击穿问题



需要写一个工具类来包装我们的数据对象,因为我们本来的对象里面是没有过期时间这个属性的

在多写一个封装类可以在不修改源代码的情况下来进行逻辑过期的实现

```
@Data
public class RedisData {
    private LocalDateTime expireTime;
    private Object data;
}
```

一般我们使用逻辑过期来处理的缓存都是一些热点缓存,即我们在开启活动之前会先将数据都存入到redis中,又由于逻辑过期的数据不会真正的过期,所以如果在缓存查询不到数据则说明数据库中也没有这个数据,就直接返回null了

这里进行缓存重建是使用ThreadUtil来获取一个线程进行缓存重建,原线程直接返回旧数据

注意,在我们封装的redisData中,获取data的时候由于redis不知道我们的data是什么数据类型的(使用泛型我无法解决),所以他统统转换成JsonObject类型,如果需要转换成对应的实体类对象,需要我们继续使用JsonUtil.toBean()来进行转换

```
private Shop queryWithLogicalExpire(Long id) {
    String key = CACHE_SHOP_KEY + id;
    String lockKey = LOCK_SHOP_KEY + id;
    // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
    String shopCache = stringRedisTemplate.opsForValue().get(key);
    // redis中不存在,直接返回null值,不需要去数据库查询的,
    // 因为一般使用这种方式的都是提前预热将数据加载到redis中的,如果没有查到说明不在本活动中
    if (StrUtil.isBlank(shopCache)) {
        return null;
    }
    // redis中存在,将redis中json转成实体对象
    RedisData redisData = JSONUtil.toBean(shopCache, RedisData.class);
    // 查看过期时间
    LocalDateTime cacheExpireTime = redisData.getExpireTime();
    JSONObject data = (JSONObject) redisData.getData();
    Shop shop = JSONUtil.toBean(data, Shop.class);
    // 判断是否过期
    if (LocalDateTime.now().isBefore(cacheExpireTime)) {
        // 未过期,直接返回店铺信息
        return shop;
    }
    // 过期,需要缓存重建
    if (tryLock(lockKey)) {
        try {
            // 获取锁之后进行一个DoubleCheck
            shopCache = stringRedisTemplate.opsForValue().get(key);
            redisData = JSONUtil.toBean(shopCache, RedisData.class);
            if (LocalDateTime.now().isBefore(redisData.getExpireTime())) {
                data = (JSONObject) redisData.getData();
                shop = JSONUtil.toBean(data, Shop.class);
                return shop;
            }
            // doubleCheck失败,redis中确实过期了,开启独立线程,需要这个这进行缓存重建
            ThreadUtil.execAsync(() -> {
                this.saveShop2Redis(1L,20L);
            });
        } finally {
            // 释放锁
            unlock(lockKey);
        }
    }
    //没获取到锁,直接返回旧的店铺信息
    return shop;
}
```

方法封装

最后我们将上面的几个缓存问题的解决方法进行了封装,得到通用的api来进行缓存的查询.便于以后的扩展

```
package com.xavier.utils;

import cn.hutool.core.thread.ThreadUtil;
import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import cn.hutool.json.JSONObject;
import cn.hutool.json.JSONUtil;
import com.xavier.entity.Shop;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
import java.time.LocalDateTime;
import java.util.concurrent.TimeUnit;
import java.util.function.Function;

import static com.xavier.utils.RedisConstants.*;
import static com.xavier.utils.RedisConstants.CACHE_SHOP_TTL;

@Component
@Slf4j
public class CacheClient {
    @Resource
    private StringRedisTemplate stringRedisTemplate;

    /**
     * 将数据设置到redis缓存中
     * @param key 设置到redis中的key
     * @param value 设置到redis中的value
     * @param time  设置到redis中的过期时间
     * @param unit  过期时间的单位
     */
    public void set(String key, Object value, Long time, TimeUnit unit) {
        stringRedisTemplate.opsForValue().set(key, JSONUtil.toJsonStr(value),
time, unit);
    }

    /**
     * 将数据以逻辑过期的方式设置到redis缓存中
     * @param key 设置到redis中的key
     * @param value 设置到redis中的value
     * @param time  设置到redis中的逻辑过期时间
     * @param unit  过期时间的单位
     */
    public void setWithLogicalExpire(String key, Object value, Long time,
TimeUnit unit) {
        RedisData data = new RedisData();
        data.setData(value);
```

```
 data.setExpireTime(LocalDateTime.now().plusSeconds(unit.toSeconds(time)));
        stringRedisTemplate.opsForValue().set(key, JSONUtil.toJsonStr(data));
    }

    /**
     * 解决了缓存穿透的redis查询
     * @param keyPrefix 查询的key的前缀
     * @param id 查询的key的id
     * @param type  查询到的数据的类型
     * @param dbFallback redis中查询不到数据到数据库中进行查询的方法
     * @param time  过期数据重新写入redis的过期时间
     * @param unit 过期时间的单位
     * @param <R> 返回值的类型
     * @param <ID> ID的类型
     * @return 返回数据
     */
    public <R, ID> R queryWithPassThrough(String keyPrefix, ID id, Class<R>
type, Function<ID, R> dbFallback, Long time, TimeUnit unit) {
        String key = keyPrefix + id;
        // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
        String valueCache = stringRedisTemplate.opsForValue().get(key);
        //redis中存在,直接返回
        if (StrUtil.isNotBlank(valueCache)) {
            // 存在后将json数据转成shop对象
            return JSONUtil.toBean(valueCache, type);
        }
        // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
        if (valueCache != null) {
            return null;
        }
        //redis中不存在,去查询数据库
        R r = dbFallback.apply(id);
        //数据库中不存在,返回错误店铺不存在
        if (r == null) {
            // 解决缓存穿透问题,向redis添加空值信息
            this.set(key, "", CACHE_NULL_TTL, TimeUnit.MINUTES);
            return null;
        }
        //数据库中存在
        //店铺添加到redis中
        this.set(key, r, time, unit);
        return r;
    }

    /**
     * 使用互斥锁的方式解决缓存击穿和缓存穿透的redis查询
     * @param keyPrefix 查询的key的前缀
     * @param id 查询的key的id
     * @param type  查询到的数据的类型
     * @param dbFallback redis中查询不到数据到数据库中进行查询的方法
     * @param time  过期数据重新写入redis的过期时间
     * @param unit 过期时间的单位
     * @param <R> 返回值的类型
     * @param <ID> ID的类型
```

```
     * @return 返回数据
     */
    public <R,ID> R queryWithMutex(String keyPrefix,ID id,Class<R>
type,Function<ID,R> dbFallback,Long time,TimeUnit unit) {
        String key = keyPrefix + id;
        String lock = LOCK_SHOP_KEY + id;
        R r = null;
        try {
            // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
            String valueCache = stringRedisTemplate.opsForValue().get(key);
            //redis中存在,直接返回
            if (StrUtil.isNotBlank(valueCache)) {
                // 存在后将json数据转成shop对象
                r = JSONUtil.toBean(valueCache,type);
                return r;
            }
            // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
            if (valueCache != null) {
                return null;
            }
            // redis中不存在
            // 获取锁
            // 获取不到锁,则递归获取数据,在这里想当与等待
            if (!tryLock(lock)) {
                Thread.sleep(50);
                return queryWithMutex(keyPrefix,id,type,dbFallback,time,unit);
            }
            // 获取到锁
            // 进行一个DoubleCheck,以防其他的线程修改完了我们再获取
            valueCache = stringRedisTemplate.opsForValue().get(key);
            //redis中存在,直接返回
            if (StrUtil.isNotBlank(valueCache)) {
                // 存在后将json数据转成shop对象
                r = JSONUtil.toBean(valueCache, type);
                return r;
            }
            // 判断命中的是否为空值,空值则直接返回fail了(即为空值缓存)
            if (valueCache != null) {
                return null;
            }
            // redis中的确不存在该数据,则查询数据库
            R value = dbFallback.apply(id);
            //数据库中不存在,返回错误店铺不存在
            if (value == null) {
                // 解决缓存穿透问题,向redis添加空值信息
                stringRedisTemplate.opsForValue().set(key, "", CACHE_NULL_TTL,
TimeUnit.MINUTES);
                return null;
            }
            //数据库中存在

            //店铺添加到redis中
            this.set(key,value,time,unit);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
```

```
        } finally {
            // 释放锁
            unlock(lock);
        }
        return r;
    }

    /**
     * 使用逻辑过期的方式解决缓存击穿的redis查询
     * @param keyPrefix 查询的key的前缀
     * @param id 查询的key的ID
     * @param type 查询数据的类型
     * @param dbFallback 过期数据到数据库中进行查询的方法
     * @param time  新数据的过期时间
     * @param unit 过期时间的单位
     * @param <R> 返回值类型
     * @param <ID> ID的类型
     * @return 返回数据
     */
    public <R, ID> R queryWithLogicalExpire(String keyPrefix, ID id, Class<R>
type, Function<ID, R> dbFallback, Long time, TimeUnit unit) {
        String key = keyPrefix + id;
        String lockKey = LOCK_SHOP_KEY + id;
        // 请求发送过来,需要获取店铺信息,我们先去查询redis缓存
        String valueCache = stringRedisTemplate.opsForValue().get(key);
        // redis中不存在,直接返回null值,不需要去数据库查询的,
        // 因为一般使用这种方式的都是提前预热将数据加载到redis中的,如果没有查到说明不在本活动
中
        if (StrUtil.isBlank(valueCache)) {
            return null;
        }
        // redis中存在,将redis中json转成实体对象
        RedisData redisData = JSONUtil.toBean(valueCache, RedisData.class);
        // 查看过期时间
        LocalDateTime cacheExpireTime = redisData.getExpireTime();
        JSONObject data = (JSONObject) redisData.getData();
        R r = JSONUtil.toBean(data, type);
        // 判断是否过期
        if (LocalDateTime.now().isBefore(cacheExpireTime)) {
            // 未过期,直接返回店铺信息
            return r;
        }
        // 过期,需要缓存重建
        if (tryLock(lockKey)) {
            try {
                // 获取锁之后进行一个DoubleCheck
                valueCache = stringRedisTemplate.opsForValue().get(key);
                redisData = JSONUtil.toBean(valueCache, RedisData.class);
                if (LocalDateTime.now().isBefore(redisData.getExpireTime())) {
                    data = (JSONObject) redisData.getData();
                    r = JSONUtil.toBean(data, type);
                    return r;
                }
                // doubleCheck失败,redis中确实过期了,开启独立线程,需要这个这进行缓存重建
                ThreadUtil.execAsync(() -> {
```

```
                    R value = dbFallback.apply(id);
                    this.setWithLogicalExpire(key,value, time, unit);
                });
            } finally {
                // 释放锁
                unlock(lockKey);
            }
        }
        //没获取到锁,直接返回旧的店铺信息
        return r;
    }

    /**
     * 获取锁,使用redis中的setnx功能,如果redis中已经存在该键则返回false,不存在则创建并返回
true
     *
     * @param key 保存在redis中的key
     * @return 返回boolean表示是否获取到了锁
     */
    private boolean tryLock(String key) {
        Boolean flag = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
10, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(flag);
    }

    /**
     * 释放锁
     *
     * @param key 释放的锁的key
     */
    private void unlock(String key) {
        stringRedisTemplate.delete(key);
    }
}
```

全局唯一ID

生成全局唯一ID可以使用在订单部分中,每下一个订单就需要生成一个全局唯一的订单号,这个订单号需要有多个要求,也不能说非常的有规律,所以我们使用一个 时间戳 + 递增ID 来生成一个64位的long类型的全局唯一ID

前32位由时间戳组成,大概可以记录几十年的数据

后32为由redis生成一个递增的数据,即使用redis中的increase函数,输入相同的key便会得到一个一直递增的数字.

最后的实现如下:

```
public class RedisIWorker {
    @Resource
    private StringRedisTemplate stringRedisTemplate;
    /**
     * 指定时间戳的开始时间,即现在距离2022-1-1:0:0:0的秒数
     */
    private static final long BEGIN_TIMESTAMP = 1640995200L;
```

```
    /**
     * 序列号的位数
     */
    private static final int COUNT_BITS = 32;

    public long nextID(String keyPrefix){
        // 生成时间戳: 即现在距离2022-1-1的秒数
        long nowSecond = LocalDateTime.now().toEpochSecond(ZoneOffset.UTC);
        long timestamp = nowSecond - BEGIN_TIMESTAMP;
        // 生成序列号: 有redis生成唯一的序列号,使用redis的好处是以后分布式全局唯一ID也可以使
用这个
        String nowDateFormat =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy:MM:dd"));
        // 这个的key使用一个前缀的方式加上一个天数,目的的为了使用同一个key的数据不要超过
redis的限制.而且方便以后进行数据统计
        long count = stringRedisTemplate.opsForValue().increment("icr:" +
keyPrefix + ":" + nowDateFormat);
        // 拼接并返回:使用位运算进行这个全局唯一ID的返回
        return timestamp << COUNT_BITS | count;
    }
}
```

然后是对这个全局唯一ID的测试,我觉得也可以学到一点东西

```
@SpringBootTest
public class RedisWorkerTest {
    @Resource
    private RedisIWorker redisWorker;
    // 使用ThreadUtil来创建一个线程池,里面包含300个线程
    private ExecutorService executorService = ThreadUtil.newExecutor(300);
    // 使用ThreadUtil创建一个CountDownLatch,这个是用于在异步的时候进行计时的,
    // 即生成一个300位的计数器,每一个线程完成任务之后主动对这个计数器进行减操作,在计数器为0时
    // 这个计数器的await方法才会被通行,否则一直等待到0才向下执行.
    private CountDownLatch countDownLatch = ThreadUtil.newCountDownLatch(300);

    @Test
    public void testNextID() throws InterruptedException {
        long start = System.currentTimeMillis();
        // 一个任务,对应一个线程,生成100个唯一ID
        Runnable task = () -> {
            for (int i = 0; i < 100; i++) {
                long id = redisWorker.nextID("order");
                System.out.println(id);
            }
            // 每次任务完成countDown一下,300个任务countDown300下
            countDownLatch.countDown();
        };
        // 启动300个线程
        for (int i = 0; i < 300; i++) {
            executorService.submit(task);
        }
        // 等待直到计数器为0
        countDownLatch.await();
        long end = System.currentTimeMillis();
```

```
        System.out.println(end - start);

    }
}
```

优惠券秒杀

优惠券下单功能

这个只是完成了优惠券的基本下单功能,没有其他特别的东西,当在高并发的场景下,会有超卖的问题出现

这个需要注意的是需要加上事务的功能

```
@Override
@Transactional
public Result seckillVoucher(Long voucherId) {
    // 1.查询优惠券
    SeckillVoucher seckillVoucher =
seckillVoucherService.getSeckillVoucherById(voucherId);
    if (seckillVoucher == null){
        return Result.fail("查询不到此优惠券!!!");
    }
    // 2.判断秒杀是否开始
    if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now())) {
        return Result.fail("秒杀活动还未开始!");
    }
    // 3.判断秒杀是否结束
    if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now())) {
        return Result.fail("秒杀活动已经结束!");
    }
    // 4.判断库存是否充足
    if (seckillVoucher.getStock() < 1){
        return Result.fail("库存不足!!!");
    }
    // 5.扣减库存
    boolean success = seckillVoucherService.updateStockById(voucherId);
    if (!success) {
        return Result.fail("更新失败!!");
    }
    // 6.创建订单
    VoucherOrder voucherOrder = new VoucherOrder();
    // 订单id,通过之前自己编写的全局唯一ID获取
    voucherOrder.setId(redisIWorker.nextID("order"));
    // 用户id,通过保存在ThreadLocal中的User获取
    voucherOrder.setUserId(UserHolder.getUser().getId());
    // 优惠券的id
    voucherOrder.setVoucherId(voucherId);
    // 订单信息写入数据库
    voucherOrderMapper.addVoucherOrder(voucherOrder);
    // 7.返回订单
    return Result.ok(voucherOrder.getId());
}
```

乐观锁解决超卖问题

CAS法: 即在更新的时候需要更新我们的数据以及版本号,所以我们直接不设置版本号以数据为版本做乐观锁就可以了

加乐观锁,可以在我们执行扣减库存的那个时候去查验一下我们的库存和当时我们查询到的库存一不一致,如果一致的话可以直接进行库存扣减,如果不一致则不执行sql语句

即将原来的sql语句改为

```
<update id="updateStock">
    update tb_seckill_voucher set stock = stock - 1 where voucher_id = #{voucherId}
</update>
```

```
<update id="updateStock">
    update tb_seckill_voucher set stock = stock - 1 where voucher_id = #{voucherId} and stock = #{stock}
</update>
```

这样进行修改之后就不会有超卖的问题了

但是出现了另一个问题,就失败率大大增加了,所有查询到的不一致的线程统统不通过,这样有可能200个线程秒杀100张优惠券卖不完就提示结束了.这是一个大问题

解决方案:我们查询到库存不一致的时候也许可以不要直接不执行sql,而已不一致的时候查询一下库存是否大于0,如果这时候还是大于0的就可以直接购买,也不会出现超买问题.

最终sql

```
<update id="updateStock">
    update tb_seckill_voucher set stock = stock - 1 where voucher_id = #{voucherId} and stock > 0
</update>
```

一人一单

在优惠券秒杀中,上面的实现不能够保证一个用户只能购买一个优惠券的功能,一个用户可以购买所有的优惠券,这样是不行的.

我们要实现一个用户只能购买一个优惠券,就需要在用户进行优惠券下单的时候去查询一下订单表看有没有与这个用户id和优惠券id重复的订单,如果有就不能下单了

在扣减库存之前需要进行下面的查询

```
// 一人一单功能实现,查看该用户是否已经购买过该优惠券了
int count = voucherOrderMapper.countOrderByUserIdAndVoucherId(userId,voucherId);
if (count > 0) {
    return Result.fail("该用户已经拥有该优惠券了!不能再购买");
}
```

但是这样在高并发的场景下也会出现一个用户购买多个订单的问题,因为查询和扣减并不是在一个同步线程内的.

这时候我们就需要对这部分的逻辑加锁了,这里只能加悲观锁即sygnization

这里我们将下单的逻辑抽取出来封装成一个方法,在这个方法里面实现悲观锁.

悲观锁可以加在方法上,这时候使用的锁就是调用这个方法的对象,但是这样会有一个效率问题,即所有用户下单都需要在这个等待,只能一个用户一个用户地下单,效率极低,所以这个加锁方式不推荐.

我们可以将锁加在方法的内部,使用userId或token来进行加锁,这样的话只有同一个用户在进行并发操作的时候需要等待,其他用户可以照常通过,非常的nice,实现如下

```
synchronized (userId.toString()) {
    // 下单逻辑
}
```

注意细节: 这里我们的userId是Long类型的,在使用toString方法的时候底层实现是每次都去new一个字符串,所以会导致我们不同线程来获取userId.toString()的时候使用的锁是不一样的,会导致锁失效的问题

所以我们需要使用userId.toString().intern()来作为锁.intern()方法对字符串去常量池中找同一个字符串并返回他的引用,这样就保证了我们同一个userId使用的是同一个字符串的引用,就不会出现上面的锁失效问题

现在整个下单方法如下

```
@Transactional
public Result createVoucherOrder(SeckillVoucher seckillVoucher){
    Long userId = UserHolder.getUser().getId();
    Long voucherId = seckillVoucher.getVoucherId();
    synchronized (userId.toString().intern()) {
        // 一人一单功能实现,查看该用户是否已经购买过该优惠券了
        int count = voucherOrderMapper.countOrderByUserIdAndVoucherId(userId,
voucherId);
        if (count > 0) {
            return Result.fail("该用户已经拥有该优惠券了!不能再购买");
        }
        // 5.扣减库存
        boolean success = seckillVoucherService.updateStock(seckillVoucher);
        if (!success) {
            return Result.fail("更新失败!!");
        }
        // 6.创建订单
        VoucherOrder voucherOrder = new VoucherOrder();
        // 订单id,通过之前自己编写的全局唯一ID获取
        voucherOrder.setId(redisIWorker.nextID("order"));
        // 用户id,通过保存在ThreadLocal中的User获取
        voucherOrder.setUserId(userId);
        // 优惠券的id
        voucherOrder.setVoucherId(voucherId);
        // 订单信息写入数据库
        voucherOrderMapper.addVoucherOrder(voucherOrder);
        return Result.ok(voucherOrder.getId());
    }
}
```

这里的实现还有点问题: 锁和事务的问题

在我们上面的实现中,事务在锁的外面,即我们释放锁的时候事务还可能未提交,这就导致了高并发下还是有可能出现多次下单的问题,即释放锁有立即查询订单数还是零的情况出现.出现这种情况的原因是我们锁的范围太小了,应该将锁的范围扩大到整个函数中.像下面这样实现:

在调用下单方法的时候加锁,这样保证了同一个用户只能同步地去调用这个下单方法,不会出现事务的问题了

```
synchronized (userId.toString().intern()) {
    // 7.优惠券下单
    return createVoucherOrder(seckillVoucher);
}
```

事务失效问题

上面的这样调用会出现一个问题: @Transactional 事务失效,原因是这个发生了方法的自身调用问题,调该类自己的方法，而没有经过 Spring 的代理类，默认只有在外部调用事务才会生效.

具体原因看博客: 事务失效八大原因

解决方法: AspectJ 暴露事务

使用AopContext.currentProxy()获取当前对象在spring中的代理对象,通过代理对象来进行方法的调用才能使事务生效.

```
synchronized (userId.toString().intern()) {
    // 获取代理对象
    IVoucherOrderService proxy = (IVoucherOrderService)
AopContext.currentProxy();
    // 7.优惠券下单
    return proxy.createVoucherOrder(seckillVoucher);
}
```

这样还需要有下面两个步骤才能生效

1. 引入AspectJ依赖

```
<!--AspectJ-->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

2. 在启动类中暴露AspectJ代理对象: @EnableAspectJAutoProxy(exposeProxy = true)

```
@MapperScan("com.xavier.mapper")
@SpringBootApplication
@EnableAspectJAutoProxy(exposeProxy = true)
public class CommentsApplication {

    public static void main(String[] args) {
        SpringApplication.run(CommentsApplication.class, args);
    }

}
```

使用Jmeter测试之后发现已经解决单机下的一人一单问题

基于redis的分布式锁

在上面的一人一单实现中,使用加锁来解决高平发中超买的问题,但是在集群分布式的情况下这个java原生的加锁机制就会失效了,需要使用到分布式锁.

实现如下:

```
@Override
public boolean tryLock(Long timeoutSec) {
    long threadId = Thread.currentThread().getId();
    Boolean success = stringRedisTemplate.opsForValue()
            .setIfAbsent(KEY_PREFIX + name, threadId + "", timeoutSec,
TimeUnit.SECONDS);
    return BooleanUtil.isTrue(success);
}

@Override
public void unlock() {
    stringRedisTemplate.delete(KEY_PREFIX + name);
}
```

所以上面的单机锁可以换成这个分布式锁的实现

但是这样还有一个问题,如果当线程1业务逻辑时间过长导致锁过期自动释放后,其他的线程如线程2就会进来,那么如果这时候线程1结束了,就会去释放锁,但是这时候的锁已经不是他的锁了,那么他就会将别人的锁释放,这会导致一系列的问题,所以在我们释放锁之前要进行一个校验,看看当前的锁是不是自己的锁,即校验锁的值.如果不是自己的就不用释放了,防止误删.

修改完成的锁: 在释放之前进行一次校验

```
@Override
public boolean tryLock(Long timeoutSec) {
    long threadId = Thread.currentThread().getId();
    String value = VALUE_PREFIX + threadId;
    Boolean success = stringRedisTemplate.opsForValue()
            .setIfAbsent(KEY_PREFIX + name, value, timeoutSec,
TimeUnit.SECONDS);
    return BooleanUtil.isTrue(success);
}

@Override
public void unlock() {
    // 获取线程标识
    String value = stringRedisTemplate.opsForValue().get(KEY_PREFIX + name);
    // 判断与当前线程是否一致,防止误删
    if (value != null && value.equals(VALUE_PREFIX +
Thread.currentThread().getId())) {
        stringRedisTemplate.delete(KEY_PREFIX + name);
    }
}
```

这个还有问题(!!!!????我服了太难了),即判断是否是自己的锁和释放锁并不是原子性的操作,即在判断完后没释放还是有别的线程可以插入进来,同样造成了误删别人的锁的问题.需要将这个判断和释放锁的操作变成一个原子性操作才能真正解决问题.

Lua脚本实现redis操作的原子性

Redis提供了Lua脚本共嫩,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调用redis:

```
redis.call("命令名称","key","其他参数")
-- 如要执行 set name jack
redis.call("set","name","jack")
```

redis执行脚本:

```
EVAL [脚本] 参数列表
# 不会使用EVAL的可以使用help @scripting学习
```

Lua脚本取参数:

redis会将key参数保存在一个KEYS数组中,将其他参数保存在一个ARGV数组中,我们在Lua脚本中想要使用外面传进来的参数值只需要通过这个数组来取相应的值便可以了.如 KEYS[1]...

注意: 这两个数的索引是从1开始的

释放锁的Lua代码如下

```
-- 获取当前线程的key
local key = KEYS[1]
-- 获取当前线程的value
local value = ARGV[1]

-- 获取redis中的当前线程的value
local now = redis.call("get",key)

-- 如果两者相等的话,则释放锁
if now == value then
    return redis.call("delete",key)
end
return 0
```

然后只需要在java中释放锁的方法中调用执行这个lua脚本就可以实现原子操作了

lua脚本的引入: 使用一个静态常量进行初始化,避免的释放锁的时候执行时间过长.

```
private static final DefaultRedisScript<Long> unlockScript;

static {
    unlockScript = new DefaultRedisScript<>();
    unlockScript.setLocation(new ClassPathResource("unlock"));
    unlockScript.setResultType(Long.class);
}
```

释放锁:

```
@Override
public void unlock() {
    // 本线程的key
    String key = KEY_PREFIX + name;
    // 本线程的value
    String value = VALUE_PREFIX + Thread.currentThread().getId();
    // 与当前线程的进行比较,一致则删除key,释放锁
    stringRedisTemplate.execute(unlockScript,
Collections.singletonList(key),value);
}
```

到这里的分布式锁就是一个可以在小项目使用的分布式锁的,但是还有其他的小问题,在大型项目中可能会有其他问题:

1. 不可重入问题
2. 不可重试问题
3. 超时释放问题
4. 主从一致性问题

这些问题出现的概率比较低,而且实现上比较难,我们自己实现的话很难,但是有现成的框架redisson

Redissson

依赖

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

配置

使用一个配置类进行配置注入

```
@Configuration
public class RedissonConfig {

    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();

 config.useSingleServer().setAddress("redis://120.25.206.171:6699").setPassword(
"123456");
        return Redisson.create(config);
    }
}
```

使用

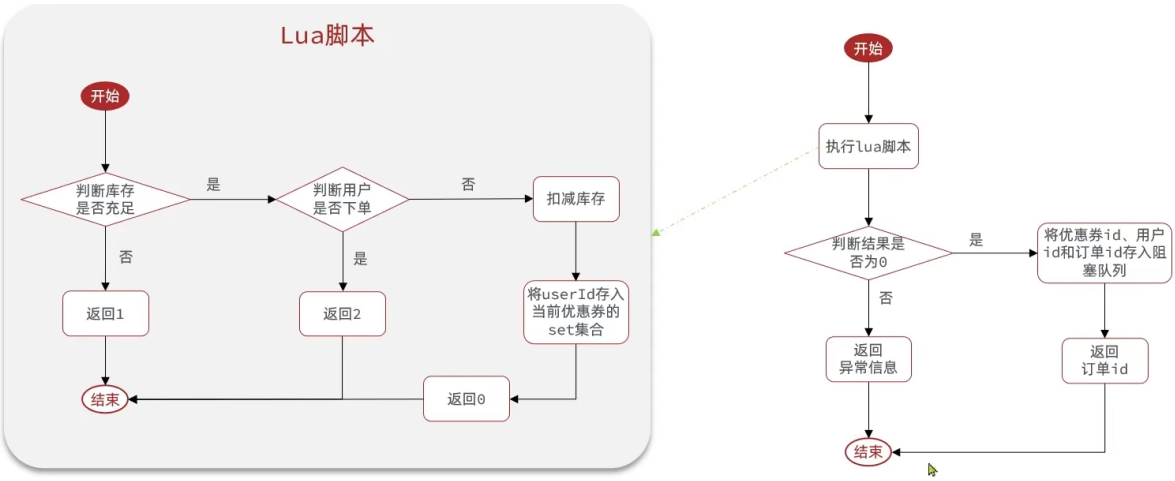
```
RLock lock = redissonClient.getLock("lock:order:" + userId);
boolean isLock = lock.tryLock();
if (!isLock){
    return Result.fail("不能重复下单!!");
}
lock.unlock();
```

秒杀优化

在前面我们的秒杀业务中,所有的秒杀操作都是串行化执行的,即只有一个线程执行到底,这样的话效率会比较低,我们可以将不同的业务,分离开来,比如秒杀资格的判定和对数据库的操作分离开来,使用不同的线程来执行这两个操作,这样的话可以提高一点效率

还有就是我们在秒杀中对数据库的操作太多,可以思考一下可不可以将对数据库的操作转移到redis中,这样就可以大大的提高效率了.

下面是我们对优惠券秒杀的优化思路,将一些对数据库的操作提前到redis中,只在redis进行下单以及购买资格和库存的查询,这样可以大大提高效率,然后我们再异步地将订单信息写入到数据库中,写数据库的操作可以慢点执行,对性能的要求不高,因为用户的请求都已经完成了.



lua脚本实现

```
-- 获取优惠券id
local voucherId = ARGV[1]
-- 获取用户id
local userId = ARGV[2]
-- 秒杀库存的前缀
local SECKILL_STOCK_KEY = ARGV[3]
-- 秒杀订单的前缀
local SECKILL_ORDER_KEY = ARGV[4]
-- 库存的key
local stockKey = SECKILL_STOCK_KEY .. voucherId
-- 订单的key
local orderKey = SECKILL_ORDER_KEY .. voucherId
-- 判断库存是否足够
local stock = redis.call("get",stockKey)
if tonumber(stock) <= 0 then
    -- 库存不足
    return 1
```

```
end
-- 判断用户是否已经购买过该优惠券
local hadBuy = redis.call("sismember",orderKey,userId)
if hadBuy == 1 then
    -- 重复下单
    return 2
end
-- 扣减库存
redis.call("incrby",stockKey,-1)
-- 添加订单
redis.call("sadd",orderKey,userId)
-- 可以下单了
return 0
```

秒杀实现: 执行lua脚本后判断返回值

使用阻塞队列(阻塞队列的特点:当队列中没有元素的时候,如果一个线程想去队列中获取元素,则线程会在这里阻塞直到有元素为止)来存放订单,然后创建线程来去将订单信息存放到数据库当中

最终秒杀业务

```
@Slf4j
@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService {
    @Resource
    private VoucherOrderMapper voucherOrderMapper;

    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIWorker redisIWorker;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    private static final DefaultRedisScript<Long> seckillScript;

    // 静态代码块初始化lua脚本
    static {
        seckillScript = new DefaultRedisScript<>();
        seckillScript.setLocation(new ClassPathResource("seckill.lua"));
        seckillScript.setResultType(Long.class);
    }

    // 阻塞队列的特点:当队列中没有元素的时候,如果一个线程想去队列中获取元素,则线程会在这里阻塞
直到有元素为止
    private BlockingQueue<VoucherOrder> orderTasks = new ArrayBlockingQueue<>
(1024 * 1024);
    // 创建线程池
    private static final ExecutorService SECKILL_ORDER_EXECUTOR =
ThreadUtil.newExecutor(1);

    /**
```

```
     * 实现Runnable的异步类
     * run方法中循环执行阻塞队列中的数据添加到数据库中
     */
    private class VoucherOrderHandler implements Runnable{
        @Override
        public void run() {
            try {
                while (true){
                    VoucherOrder voucherOrder = orderTasks.take();
                    // 更新数据库数据
                    updateDataBase(voucherOrder);
                }
            } catch (InterruptedException e) {
                log.error("处理订单异常!!",e);
            }
        }
    }

    /**
     * 使用@PostConstruct注解,在类加载的时候便会执行这个函数
     * 函数中使用线程池来去提交我们的任务
     */
    @PostConstruct
    private void init(){
        SECKILL_ORDER_EXECUTOR.submit(new VoucherOrderHandler());
    }
    /**
     * 秒杀优惠券优化后的业务
     *
     * @param voucherId 优惠券的id
     * @return 返回购买结果
     */
    @Override
    public Result seckillVoucher(Long voucherId) {
        Long userId = UserHolder.getUser().getId();
        // 执行脚本
        Long result = stringRedisTemplate.execute(
                seckillScript,
                Collections.emptyList(),
                voucherId.toString(), userId.toString(), SECKILL_STOCK_KEY,
SECKILL_ORDER_KEY
        );
        int r = result.intValue();
        // 购买失败
        if (r != 0) {
            return Result.fail(r == 1 ? "库存不足,请下次再来!" : "你已经购买过了!!");
        }
        // 购买成功
        long orderId = redisIWorker.nextID("order");
        // 新建订单对象
        VoucherOrder voucherOrder = new VoucherOrder();
        voucherOrder.setId(orderId);
        voucherOrder.setVoucherId(voucherId);
        voucherOrder.setUserId(userId);
        // 使用阻塞队列进行数据库下单
```

```
        orderTasks.add(voucherOrder);
        return Result.ok(orderId);
    }

    /**
     * 更新数据库的操作
     * 包括扣减库存以及插入订单
     * @param voucherOrder 订单信息
     */
    private void updateDataBase(VoucherOrder voucherOrder){
        // 扣减库存
        seckillVoucherService.updateStock(voucherOrder.getVoucherId());
        // 这里直接将voucherOrder写入数据库了
        // 其他也可以像我们前面做的一样加一个分布式锁来实现,
        // 但是已经没必要了,因为我们已经的redis将重复下单的用户过滤掉了,
        // 理论上不会出现同一个用户进入这个函数的情况,但是自己也可以实现做一个兜底
        voucherOrderMapper.addVoucherOrder(voucherOrder);
    }
}
```

博客部分

博客部分主要是在主页以及对应用户界面中查询用户发过的博客或者是主页中获取比较火热的博客列表功能.这部分的功能相对来说比较的简单

这部分主要是使用了redis中的set类型数据来进行数据的保存

如点赞列表中,使用一个当前用户id为key,以点赞过的博客id为value就可以轻松查出点赞过的博客

点赞的功能如下: 这里使用zset数据类型主要是为了之后在博客页面中按点赞顺序显示点赞过的用户的头像来做的,其实也可以使用set实现

```
@Override
public Result likeBlog(Long id) {
    // 获取当前用户id
    UserDTO user = UserHolder.getUser();

    // 判断是否已经点赞
    Double score = stringRedisTemplate.opsForZSet().score(BLOG_LIKED_KEY + id,
user.getId().toString());
    // 没点赞过
    if (score == null){
        // 数据库点赞数+1
        blogMapper.updateLiked(id,1);
        // 将点赞信息保存在redis中
        stringRedisTemplate.opsForZSet().add(BLOG_LIKED_KEY +
id,user.getId().toString(), System.currentTimeMillis());
    }else {
        // 点赞过,取消
        blogMapper.updateLiked(id,-1);
        stringRedisTemplate.opsForZSet().remove(BLOG_LIKED_KEY + id,
user.getId().toString());
    }

    return Result.ok();
```

```
}
```

这部分没什么太难的内容,主要实现可以自己去看BlogServiceImpl中的具体实现

关注部分

好友关注

关注功能实现也是比较的简单,只是需要将关注的主体以及被关注的用户保存到数据库中便可以了

共同关注

共同关注就需要使用redis中的set数据类型,因为set数据类型有一个取交集的功能,可以取出不同用户关注列表中的交集部分也就是共同关注部分了

要实现这个就需要我们在关注用户的时候就将关注的用户保存在一个set中,这个set可以以当前用户id为key,以被关注的用户id的set集合为值,这样在找共同关注的时候直接使用两个key来进行取交集就可以了.

其中获取到的交集是一个set<String>集合,这里使用到了stream流来就set集合转化为一个List<Long>列表,可以学习一下午这个转化

```
// 获取共同关注的userId列表
List<Long> ids =
intersect.stream().map(Long::valueOf).collect(Collectors.toList());
```

具体共同关注的实现

```
@Override
public Result findCommon(Long id) {
    // 获取当前登录的用户id
    Long userId = UserHolder.getUser().getId();
    String key1 = FOLLOWS_KEY + userId;
    String key2 = FOLLOWS_KEY + id;

    // redis中查找两个用户的关注的交集
    Set<String> intersect =
stringRedisTemplate.opsForSet().intersect(key1,key2);
    // 没有共同关注返回空列表
    if (intersect == null || intersect.isEmpty()) return
Result.ok(Collections.emptyList());
    // 获取共同关注的userId列表
    List<Long> ids =
intersect.stream().map(Long::valueOf).collect(Collectors.toList());
    // 查询共同关注用户信息
    List<UserDTO> users = new LinkedList<>();
    for (Long id2: ids){
        UserDTO userDTO = new UserDTO();
        User user = userService.getUserById(id2);
        BeanUtil.copyProperties(user,userDTO);
        users.add(userDTO);
    }
    return Result.ok(users);
}
```

消息推送

消息推送也叫Feed流,有两种常见的模式:

1. Timeline: 补做内容筛选,简单的按照内容发布时间排序,常用于好友或关注.
2. 智能排序: 利用智能算法屏蔽违规,不感兴趣的内容,推送用户感兴趣信息来吸引用户

在我们的这个应用中,是在个人关注页面做Feed流,因此采用Timeline模式

这种模式也有三种实现方案

1. 拉模式
2. 推模式
3. 推拉结合

	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

推送

在用户上传blog的时候直接将blog保存在redis中用户的粉丝的收件箱中,到时候粉丝一登录就可以直接去收件箱中查询对应的blog并展示了

```
@Override
public Result uploadBlog(Blog blog) {
    // 获取登录用户
    UserDTO user = UserHolder.getUser();
    blog.setUserId(user.getId());
    // 保存探店博文
    blogMapper.saveBlog(blog);
    // 将blog推送到所有粉丝的redis收件箱中
    // 查询粉丝
    List<Long> fansId = followService.getUserIdByFollowUserId(blog.getUserId());
    for (Long id : fansId){
        stringRedisTemplate.opsForZSet().add(FANS_FEED_KEY +
id,blog.getId().toString(),System.currentTimeMillis());
    }
    // 返回id
    return Result.ok(blog.getId());
}
```

接收(滚动查询)

这里接收有一个难点,就是要实现滚动查询.

滚动查询有两者实现,一是使用下标的方式来进行滚动查询,但是这样有一个问题,就是当有新增的blog的时候就有可能会发生下一页的blog和上一页的blog重复了,因为下标变化了,所以这个方式不可取

第二种是使用zset的方式,通过score排序来进行查询,score中填写的是发送blog的时候的时间戳,在我们进行分页查询的时候,发送给后端我们最后一个blog的score值,我们就可以根据这个score值去redis查询这个对应score下面的数据blog,这样不会出现blog重复的问题.

这个的实现比较复杂,offset为重复score的个数,我们的在进行查询返回blog的时候也要返回给前端我们本次查询的offset个数,以便他下一次查询的时候将这个offset发送给我们,我们好根据这个来进行重复数据的跳过.

```
@Override
public Result queryBlogOfFollow(Long lastId, Integer offset) {
    Long userId = UserHolder.getUser().getId();
    String key = FANS_FEED_KEY + userId;
    // 获取收件箱中的数据
    Set<ZSetOperations.TypedTuple<String>> typedTuples =
stringRedisTemplate.opsForZSet().
            reverseRangeByScoreWithScores(key, 0, lastId, offset,
MAX_FOLLOW_PAGE_SIZE);
    // 非空判断
    if (typedTuples == null ||typedTuples.isEmpty()) return
Result.ok(Collections.emptyList());
    // 创建一个集合用于保存所有的blog的id
    List<Long> ids = new ArrayList<>(typedTuples.size());
    // 用户保存最后一个数据的分数
    long minTime = 0;
    // 记录最后一个元素有多少个是同一个分数的,用户以后的offset赋值
    int os = 1;
    for (ZSetOperations.TypedTuple<String> typedTuple : typedTuples) {
        // 获取id
        ids.add(Long.valueOf(typedTuple.getValue()));
        long time = typedTuple.getScore().longValue();
        if (time == minTime){
            os ++;
        }else{
            // 获取分数
            minTime = time;
            os = 1;
        }
    }
    // 用于保存收件箱中的所有blog
    List<Blog> blogs = new ArrayList<>(ids.size());

    // 根据id查询blog并保存中blogs中
    for (Long id : ids) {
        Blog blog = blogMapper.getBlogById(id);
        queryUserByBlog(blog);
        isBlogLiked(blog);
        blogs.add(blog);
    }
    ScrollResult result = new ScrollResult();
```

```
    result.setList(blogs);
    result.setMinTime(minTime);
    result.setOffset(os);
    return Result.ok(result);
}
```

附近商铺

实现附近商铺功能,需要使用到redis中的GEO数据类型,这个数据类型允许存储地理坐标信息,帮助我们根据经纬度来检索数据.

添加店铺坐标信息

将店铺的坐标信息添加到redis中

```
@Test
public void preCacheShopGEO() {
    List<Shop> shopList = shopService.list();
    Map<Long, List<Shop>> collect =
shopList.stream().collect(Collectors.groupingBy(Shop::getTypeId));
    for (Map.Entry<Long, List<Shop>> entry : collect.entrySet()) {
        String key = SHOP_GEO_KEY + entry.getKey();
        List<RedisGeoCommands.GeoLocation<String>> locations = new ArrayList<>
();
        for (Shop shop : entry.getValue()){
            RedisGeoCommands.GeoLocation<String> stringGeoLocation =
                    new RedisGeoCommands.GeoLocation<>(
                            shop.getId().toString(),
                            new Point(shop.getX(), shop.getY())
                    );
            locations.add(stringGeoLocation);
        }
        stringRedisTemplate.opsForGeo().add(key,locations);
    }
}
```

根据距离获取店铺信息的业务,实现还是比较难的,里面有不认识的对象,可以学一学,以及中间有使用了stream流的skip功能

```
@Override
public Result getShopByTypeOrderByDist(Integer typeId, Integer current, Double
x, Double y) {
    // 计算分页查询的开始坐标
    int startIndex = (current - 1) * MAX_PAGE_SIZE;
    // 不需要根据经纬度排序
    if(x == null || y == null){
        List<Shop> shops =
 shopMapper.getShopByTypeId(typeId,startIndex,MAX_PAGE_SIZE);
        return Result.ok(shops);
    }
    // 计算分页查询结束坐标
    int endIndex = current * MAX_PAGE_SIZE;
    // redis中进行查询的key
    String key = SHOP_GEO_KEY + typeId;
```

```
    // 查询方圆5公里的店铺信息以及对应的距离distance
    GeoResults<RedisGeoCommands.GeoLocation<String>> results =
stringRedisTemplate.opsForGeo().search(
            key,
            GeoReference.fromCoordinate(x, y),
            new Distance(MAX_GEO_DISTANCE),

 RedisGeoCommands.GeoSearchCommandArgs.newGeoSearchArgs().includeDistance().limi
t(endIndex));
    // 空值判断
    if (results == null) return Result.ok(Collections.emptyList());
    // 获取信息内容
    List<GeoResult<RedisGeoCommands.GeoLocation<String>>> content =
results.getContent();
    // 用于保存店铺信息
    List<Shop> shops = new ArrayList<>();
    // 遍历根据获取到的店铺id查询店铺信息,以及将distance填入shop中
    content.stream().skip(startIndex).forEach(result -> {
        Long id = Long.valueOf(result.getContent().getName());
        Double distance = result.getDistance().getValue();
        Shop shop = shopMapper.getShopById(id);
        shop.setDistance(distance);
        shops.add(shop);
    });
    // 返回
    return Result.ok(shops);
}
```

签到功能

签到功能使用redis的bitMap数据类型来记录,bitmap是一个二进制字符串,我们可以将一个用户一个月的签到情况存储到这里面,然后就可以进行签到的记录以及签到情况的统计

签到实现

bitMap底层是使用String实现的,所以bitmap的相关操作都保存在opsForValue中了

```
@Override
public Result userSign() {
    // 获取当前用户
    Long userId = UserHolder.getUser().getId();
    // 获取当前时间
    LocalDateTime now = LocalDateTime.now();
    String format = now.format(DateTimeFormatter.ofPattern("yyyy:MM"));
    // 保存在redis中的key
    String key = USER_SIGN_KEY + userId + ":" + format;
    // 获取当前在天在这个月的天数
    int dayOfMonth = now.getDayOfMonth();
    // 保存到redis中
    stringRedisTemplate.opsForValue().setBit(key,dayOfMonth - 1,true);
    return Result.ok();
}
```

签到统计

用于计算用户本月到今天为止连续签到的天数,使用位移运算

```
public Result userSignCount() {
    // 获取当前用户
    Long userId = UserHolder.getUser().getId();
    // 获取当前时间
    LocalDateTime now = LocalDateTime.now();
    String format = now.format(DateTimeFormatter.ofPattern("yyyy:MM"));
    // 保存在redis中的key
    String key = USER_SIGN_KEY + userId + ":" + format;
    // 获取当前在天在这个月的天数
    int dayOfMonth = now.getDayOfMonth();
    // 获取redis中存储的bitMap的这个串
    List<Long> longs = stringRedisTemplate.opsForValue().bitField(key,
            BitFieldSubCommands.create().

 get(BitFieldSubCommands.BitFieldType.unsigned(dayOfMonth)).valueAt(0));
    // 空值判断
    if (longs == null || longs.isEmpty()) return Result.ok(0);
    Long num = longs.get(0);
    if (num == null || num == 0) return Result.ok(0);
    // 计数器
    int count = 0;
    // 循环遍历并右移,判断是否为1,为1则是签到
    while (true){
        if ((num & 1) == 0){
            break;
        }else {
            count++;
        }
        // 右移并赋值
        num >>>= 1;
    }
    // 返回
    return Result.ok(count);
}
```