--redis实战学习笔记--

- 注意：此文件里的部分代码有问题，后面我更新了项目，但是没有更新此文件，应该以项目为准
- 项目地址：点击进入

登录

基于redis实现登录功能

发送验证码

```
@Override
    public Result sendCode(String phone, HttpSession session)
    {
        //验证手机号
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //验证不通过，返回错误提示
            log.debug("验证码错误.....");
            return Result.fail("手机号错误，请重新填写");
        }
        //验证通过，生成验证码
        //6位数
        String code = RandomUtil.randomNumbers(6);
        //保存验证码到redis
        stringRedisTemplate.opsForValue().set(RedisConstants.LOGIN_CODE_KEY +
phone,
                code, RedisConstants.LOGIN_CODE_TTL, TimeUnit.MINUTES);
        //发送验证码
        log.debug("验证码发送成功," + code);
        //返回响应
        return Result.ok();
    }
```

登录

```
 @Override
    public Result login(LoginFormDTO loginForm, HttpSession session)
    {
        //判断手机号格式是否正确
        String phone = loginForm.getPhone();
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //如果不正确则直接返回错误
            log.debug("手机号:" + phone + "错误");
            return Result.fail("手机号格式错误");
```

```
        }
        //判断验证码是否一致，redis中对比
        //String cacheCode = session.getAttribute("code").toString();
        String cacheCode =
stringRedisTemplate.opsForValue().get(RedisConstants.LOGIN_CODE_KEY + phone);
        String code = loginForm.getCode();
        //如果验证码为空，或者不一致，则返回验证码错误
        if (code == null || code.length() == 0)
        {
            return Result.fail("验证码不能为空");
        }
        //判断验证码是否为6位数
        if (code.length() != 6)
        {
            return Result.fail("验证码长度不正确");
        }
        //判断验证码是否正确
        if (!code.equals(cacheCode))
        {
            //验证码错误
            return Result.fail("验证码错误");
        }
        //验证码输入正确
        //判断用户是否存在
        User user = query().eq("phone", phone).one();
        //如果用户不存在则创建用户，保存到数据库
        if (user == null)
        {
            //创建用户，保存到数据库
            user = createUser(phone);
        }
        //如果用户存在，保存到redis
        //session.setAttribute("user", user);
        //生成token，作为登录令牌
        String token = UUID.randomUUID().toString(true);
        //将User对象转为Hash存储. UserDTO是用户的部分信息
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        //转map
        Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new HashMap<>
(), CopyOptions.create()
                .setIgnoreNullValue(true) // 忽略空的值
                .setFieldValueEditor((fieldName, fieldVaule) ->
fieldVaule.toString()));

        //保存到redis中
        //保存的key
        String tokenKey = RedisConstants.LOGIN_USER_KEY + token;
        //保存
        stringRedisTemplate.opsForHash().putAll(tokenKey, userMap);
        //设置有效期
        stringRedisTemplate.expire(tokenKey, RedisConstants.LOGIN_USER_TTL,
TimeUnit.MINUTES);
        //返回响应，返回token
        return Result.ok(token);
    }

    /**
     * 创建用户，添加到数据库中
```

```
     *
     * @param phone 手机号码
     * @return user
     */
    private User createUser(String phone)
    {
        User user = new User();
        user.setPhone(phone);
        user.setNickName(SystemConstants.USER_NICK_NAME_PREFIX +
RandomUtil.randomString(10));
        //将用户信息插入到 t_user表中
        this.save(user);
        //返回数据
        return user;
    }
```

登录校验

登录拦截器

```
public class LoginInterceptor implements HandlerInterceptor
{

    /**
     * 拦截器校验用户
     *
     * @param request  HttpServletRequest
     * @param response HttpServletResponse
     * @param handler  Object
     * @return boolean
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception
    {
        // 判断是否拦截,查看ThreadLoad是否有用户
        if (UserHolder.getUser() == null)
        {
            //不存在，拦截，响应401
            response.setStatus(401);
            return false;
        }
        //放行
        return true;

    }
}
```

刷新token的拦截器

```
public class RefreshTokenInterceptor implements HandlerInterceptor
{
    private StringRedisTemplate stringRedisTemplate;

    /**
     * 构造函数
     *
     * @param stringRedisTemplate StringRedisTemplate
     */
    public RefreshTokenInterceptor(StringRedisTemplate stringRedisTemplate)
    {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    /**
     * 拦截器刷新token的过期时间
     *
     * @param request  HttpServletRequest
     * @param response HttpServletResponse
     * @param handler  Object
     * @return boolean
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception
    {
        //从请求头里获取token
        String token = request.getHeader("authorization");
        //判断token是否存在
        if (token == null || token.equals(""))
        {
            //不存在，拦截
            return false;
        }
        //token存在，根据token获取redisKey
        String redisKey = RedisConstants.LOGIN_USER_KEY + token;
        Map<Object, Object> userMap =
stringRedisTemplate.opsForHash().entries(redisKey);
        //
        //判断用户是否存在，userMap为空
        if (userMap.isEmpty())
        {
            //不存在，拦截，响应401
            response.setStatus(401);
            return false;
        }
        //存在
        //转实体类
        UserDTO userDTO = BeanUtil.fillBeanWithMap(userMap, new UserDTO(),
false);
        //保存到ThreadLoad
        UserHolder.saveUser(BeanUtil.copyProperties(userDTO, UserDTO.class));
        //刷新过期时间，类似于session机制
```

```
        stringRedisTemplate.expire(redisKey, RedisConstants.LOGIN_USER_TTL,
TimeUnit.MINUTES);
        //放行
        return true;
    }

    /**
     * 渲染之后，返回用户之前。 用户执行完毕后，销毁对应的用户信息
     *
     * @param request  HttpServletRequest
     * @param response HttpServletResponse
     * @param handler  Object
     * @param ex       Exception
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex)
            throws Exception
    {
        UserHolder.removeUser();
    }
}
```

spring mvc拦截器配置

```
@Configuration
public class SpringMvcConfiguration implements WebMvcConfigurer
{
    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public void addInterceptors(InterceptorRegistry registry)
    {
        //添加拦截器，登录拦截器
        InterceptorRegistration interceptorRegistration =
registry.addInterceptor(new LoginInterceptor());
        // 添加配置可以放行哪些路径
        interceptorRegistration.excludePathPatterns(
                "/shop/**",
                "/voucher/**",
                "/shop-type/**",
                "/upload/**",
                "/blog/hot",
                "/user/code",
                "/user/login"
        ).order(1);
        //刷新token过期时间拦截器
        InterceptorRegistration refreshTokenInterceptorRegistration =
                registry.addInterceptor(new
RefreshTokenInterceptor(stringRedisTemplate));
        refreshTokenInterceptorRegistration.addPathPatterns("/**").order(0);
    }
```

```
}
```

商户查询缓存

缓存是什么?

缓存就是数据交换的缓冲区（称作Cache [kæʃ] ），是存贮数据的临时地方，一般读写性能较高

缓存的作用

- 降低后端负载：当用户进行请求时，先去查询缓存，查询到之后直接返回给用户，而不必查询数据库，大大降低了后端的压力
- 提高读写效率，降低响应时间：数据库的读写是磁盘读写，其响应时间一般比较长，而 Redis 是基于内存的，读写时间快

缓存的成本

- 数据一致性成本：当数据库中的数据发生了改变，而缓存中的数据还是旧的数据，当用户从缓存中读取数据时，获取到的依旧是旧的数据
- 代码维护成本：为了解决一致性成本，以及缓存雪崩，缓存击穿等问题，就需要非常复杂的业务编码
- 运维成本：缓存集群的部署维护需要额外的人力成本、硬件成本

查询实现

```
    @Override
    public Result queryById(Long id)
    {
        // 1、根据 Id 查询 Redis
        String shopJson =
stringRedisTemplate.opsForValue().get(RedisConstants.CACHE_SHOP_KEY + id);
        // 2、判断 shopJSON 是否为空
        if (StrUtil.isNotBlank(shopJson)) {
            // 3、存在，直接返回
            Shop shop = JSONUtil.toBean(shopJson, Shop.class);
            return Result.ok(shop);
        }
        // 4、不存在，查询数据库
        Shop shop = getById(id);

        // 5、不存在，返回错误
        if(shop == null){
            return Result.fail("店铺不存在！");
```

```
        }

        // 6、存在，写入 Redis
        stringRedisTemplate.opsForValue().set(RedisConstants.CACHE_SHOP_KEY +
id, JSONUtil.toJsonStr(shop));
        // 7、返回
        return Result.ok(shop);
    }
```

缓存更新策略

- 内存淘汰
- 超时剔除
- 主动更新

- 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

主动更新策略

- Cache Aside Pattern：由缓存的调用者，在更新数据库的同时更新缓存，推荐
- Read/Writer Through Pattern：缓存与数据库整合为一个服务，由服务来维护一致性。调用者调用该服务，无需关心缓存一致性问题。
- Writer Behind Caching Pattern：调用者只操作缓存，由其他线程异步的将缓存数据持久化到数据库，保证最终一致。

更新缓存的三个问题

1.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。
  如果我们对数据库做了上百次操作，那么就需要对缓存进行上百次操作，在进行这上百次的操作过程中，如果没有任何的查询操作，也就是写多读少，那么对于缓存的上百次操作都可以看作成是无效的操作。
- 删除操作：更新数据库时让缓存失效，查询时再更新缓存。（一般选择此种方案）

2.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
- 分布式事务：利用 TCC 等分布式事务方案

3.先操作缓存还是先操作数据库?

- 先删除缓存，再操作数据库：假设有两个线程：线程1 和 线程2，线程1执行更新操作，先将缓存中的数据删除，然后执行更新数据库操作，由于更新逻辑复杂，执行时间较长，此时线程2 也开始执行，线程2 执行查询操作，由于缓存中的数据被线程 1 删除了，导致查询缓存未命中，于是线程2 转而去查询数据库，此时数据库并未完成更新操作，查询出的数据依旧为旧数据，接着程序就将旧数据重新写入到了缓存。这就会导致后续的所有查询操作查询到的数据依旧是旧数据。
- 先操作数据库，再删除缓存：
  情况一：假设有两个线程，线程 1 和线程 2，线程 1 执行更新操作，线程 1 先去更新数据库，然后再删除缓存，由于更新逻辑复杂，执行时间较长，此时线程2 也开始执行，线程 2 执行查询操作，由于此时数据库尚未更新完成，且缓存未被删除，线程 2 依然能从缓存中查询到旧的数据，一旦线程 1 更新数据库完成，且删除了缓存中的数据，那么其他线程再查询时就会无法命中缓存，从而去查询数据库同步缓存数据。这种情况的一个好处就是，即使线程1 未完成数据库的更新，其他线程在查询时依然能够命中缓存，哪怕是旧的缓存，也不会额外浪费时间去查询数据库。而且一旦数据库更新完成，后续的查询便都是最新的数据。情况二：还有一种情况就是当线程 2 执行查询操作时，此时缓存中的数据恰好过期，然后线程 2 便会去数据库中查询，但是此时线程 1 未完成更新操作，所以数据库中还是原先的数据，线程 2 在将旧数据重新写入缓存的同时，恰巧线程 1 完成了数据库更新操作，并将缓存删除，这就导致缓存中的数据一直是旧数据。但实际上这种情况发生的概率极低，为了避免这种情况的发生，可以在写入缓存的时候设置过期时间。

缓存更新策略的最佳实践方案

- 低一致性需求：使用Redis自带的内存淘汰机制
- 高一致性需求：主动更新，并以超时剔除作为兜底方案

  读操作：·缓存命中则直接返回·缓存未命中则查询数据库，并写入缓存，设定超时时间

  写操作：·先写数据库，然后再删除缓存·要确保数据库与缓存操作的原子性

更新实现

```
@Override
public Result updateShop(Shop shop)
{
    //获得id
    Long id = shop.getId();
    //判断是否为空
    if (id == null)
    {
        return Result.fail("商户id不能为空");
    }
    //不为空
    //先更新数据库
    boolean b = this.updateById(shop);
    //更新失败，返回
    if (!b)
    {
        return Result.fail("更新失败");
    }
    //更新没有失败
    //删除redis里的数据，下一次查询时自动添加进redis
    //redisKey
    String redisKey = RedisConstants.CACHE_SHOP_KEY + id;
    stringRedisTemplate.delete(redisKey);
```

```
        //返回响应
        return Result.ok();
    }
```

或者

```
     /**
     * 更新数据
     *
     * @param id          要更新的主键
     * @param data        要更新的对象
     * @param keyPrefix   redis的key前缀
     * @param dbFallback 更新数据库的函数，返回值要为Boolean类型
     * @param <T>         要更新的对象的泛型
     * @param <ID>        主键的类型
     * @return boolean
     */
    public <T, ID> boolean update(ID id, T data, String keyPrefix, Function<T,
Boolean> dbFallback)
    {
        //判断是否为空
        if (id == null)
        {
            return false;
        }
        //不为空
        //先更新数据库
        boolean b = dbFallback.apply(data);
        //更新失败，返回
        if (!b)
        {
            return false;
        }
        //更新没有失败
        //删除redis里的数据，下一次查询时自动添加进redis
        //redisKey
        String redisKey = keyPrefix + id;
        stringRedisTemplate.delete(redisKey);
        //返回响应
        return true;
    }
```

缓存问题

缓存穿透

是什么?

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，这些请求都会到达数据库。
如果有恶意用户，使用大量线程并发访问这些不存在的数据，这样所有的请求都会到达数据库，数据库顶不住访问压力，就会崩掉

解决方案

缓存空对象

将数据库中不存在的数据以 null 的形式存储到缓存中。但是这种方式会增加额外的内存消耗，我们可以在缓存 null 的时候，设置过期时间。

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

在客户端与 Redis 之间增加一层过滤，当用户请求来的时候，先去访问布隆过滤器，判断请求的数据是否存在，如果不存在则拒绝请求，如果存在，则放行。

优点:

- 内存占用较少，没有多余的 key

缺点:

- 实现复杂
- 存在误判的可能

布隆过滤器判断时，如果数据不存在，就是真的不存在，如果判断数据存在，那么有可能不存在。存在一定的穿透风险

解决缓存穿透

```
    /**
     * 查询数据，有缓存，解决缓存穿透问题，未解决缓存雪崩问题
     *
     * @param keyPrefix  redisKey的前缀
     * @param id         id
     * @param type       返回值的类型
     * @param dbFallback 查询数据库的函数
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     * @param <R>        返回值的类型
     * @param <ID>       id的类型
     * @return 泛型R
     */
```

```
    public <R, ID> R queryWithPassThrough(String keyPrefix, ID id, Class<R>
type,
                                          Function<ID, R> dbFallback, Long
expireTime, TimeUnit timeUnit)
    {
        //获得前缀
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断是否为空
        if (StrUtil.isNotBlank(json))
        {
            //不为空，返回
            return JSONUtil.toBean(json, type);
        }
        //判断是否为空串
        if (json != null)
        {
            //空串
            return null;
        }
        //null
        //查数据库
        R r = dbFallback.apply(id);
        //判断
        if (r == null)
        {
            //数据库也为空，缓存空值
            this.set(redisKey, "", expireTime, timeUnit);
            return null;
        }
        //数据库存在，写入redis
        this.set(redisKey, r, expireTime, timeUnit);
        //返回
        return r;
    }
```

缓存雪崩

是什么?

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，给数据库带来巨大压力。

解决方案

给不同的 Key 的 TTL（过期时间）添加随机值

一般在做缓存预热时，可能会提前将数据库中的数据批量导入到缓存中，由于是批量导入的，所以这些 key 的 TTL 是一样的，这就很有可能导致这些 key 在未来的某一时刻一起过期，从而引发缓存雪崩问题。为了解决这个问题，我们可以在做缓存预热时，可以在设置 TTL 时，在 TTL 后面追加一个随机数，比如 TTL 设置的 30 分钟，我们在30 的基础上加上一个 1~5之间的随机数，那么这些 key 的过期时间就会在 30 ~ 35 之间，这样就可以将 key 的过期时间分散开来，而不是一起失效。

利用 Redis 集群提高服务的可用性

利用 Redis 的哨兵机制，Redis 哨兵机制可以实现服务的监控，比如在一个主从模式下的 Redis 集群，当主机宕机时，哨兵就会从从机中选出一个来替代主机，这样就可以确保 Redis 一直对外提供服务。另外，主从模式还可以实现数据的同步，当主机宕机，从机上的数据也不会丢失。

给缓存业务添加降级限流策略

给业务添加多级缓存

可以先在反向代理服务器 Nginx 中做缓存，在 Nginx 中未命中缓存时，再去 Redis 中查询

解决缓存雪崩

```
    /**
     * 查询数据，有缓存，解决缓存穿透问题，解决缓存雪崩问题
     *
     * @param keyPrefix                         redisKey的前缀
     * @param id                                id
     * @param type                              返回值的类型
     * @param dbFallback                        查询数据库的函数
     * @param expireTime                        过期时间
     * @param timeUnit                          时间单位
     * @param <R>                               返回值的类型
     * @param <ID>                              id的类型
     * @param maxTimeSecondsByCacheAvalanche this.set(redisKey, r,
     *
timeUnit.toSeconds(expireTime)+getIntRandom(0,maxTimeSecondsByCacheAvalanche),
TimeUnit.SECONDS);
     * @return 泛型R
     */
    public <R, ID> R queryWithPassThroughAndCacheAvalanche(String keyPrefix, ID
id, Class<R> type,
                                                           Function<ID, R>
dbFallback, Long expireTime, TimeUnit timeUnit,
                                                           Integer
maxTimeSecondsByCacheAvalanche)
    {
        //获得前缀
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断是否为空
        if (StrUtil.isNotBlank(json))
        {
            //不为空，返回
            return JSONUtil.toBean(json, type);
        }
        //判断是否为空串
        if (json != null)
        {
            //空串
            return null;
        }
        //null
```

```
        //查数据库
        R r = dbFallback.apply(id);
        //判断
        if (r == null)
        {
            //数据库也为空，缓存空值
            this.set(redisKey, "",
                    timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                    TimeUnit.SECONDS);
            return null;
        }
        //数据库存在，写入redis
        this.set(redisKey, r,
                timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                TimeUnit.SECONDS);
        //返回
        return r;
    }

    /**
     * 获取一个随机数，区间包含min和max
     *
     * @param min 最小值
     * @param max 最大值
     * @return int 型的随机数
     */
    @SuppressWarnings("all")
    private int getIntRandom(int min, int max)
    {
        if (min > max)
        {
            min = max;
        }
        return min + (int) (Math.random() * (max - min + 1));
    }
```

缓存击穿

是什么?

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，无效的请求访问会在瞬间给数据库带来巨大压力。

解决方案

互斥锁

假设线程 1 查询缓存未命中，那么线程 1 就需要进行缓存重建工作，为了避免其他线程重复线程1 的工作，那么线程 1 就必须要先获取互斥锁，只有获取锁成功的线程才能够重建缓存数据。重建完成后，线程 1 就会将数据写入到缓存中，并将锁释放。如果在线程 1 将数据写入缓存之前，其他线程涌入，这个时候，其他线程查询缓存依然是未命中的，那么这些线程为了重建缓存，也必须先获取到互斥锁，但是，由于此时线程 1 未释放锁，所以其他线程就会获取锁失败，一旦获取锁失败，一般程序处理是让线程休眠一会儿，然后再重试（包括查询缓存以及获取互斥锁），如果线程 1 执行缓存重建时间过长，就会导致其他线程一直处于阻塞等待重试的状态，效率过低。

逻辑过期

当我们在向 Redis 中存储数据时，不再为 key 设置过期时间（TTL），但是，需要在 value 中额外添加一个逻辑时间（以当前时间为基础，加上需要设置的过期时间），也就是说，这个 key 一旦存入到 Redis 中，就会永不过期。假设线程 1 在查询缓存时发现逻辑时间已经过期，为了避免出现多个线程重建缓存，线程 1 就会去获取互斥锁，一旦线程 1 获取互斥锁成功，线程 1 就会开启一个独立线程，由独立线程去查询数据库重建缓存数据，以及写入缓存重置逻辑过期时间等操作，一旦完成操作，独立线程就会将互斥锁释放掉。线程 1 在开启独立线程后，会直接将过期数据返回。而在独立线程释放锁之前，缓存中的数据都是过期数据。当其他线程在此之前涌入程序时，去查询缓存获取到依旧是逻辑时间过期的数据，那么这些线程就会试图获取互斥锁，此时由于独立线程还未释放锁，所以会获取锁失败，一旦失败，这些线程就会将查询到的旧数据返回。只有当独立线程执行结束，其他线程才会从缓存中获取到新数据。

两种方案的优点和缺点

互斥锁

优点：

- 没有额外的内存消耗
- 保证一致性
- 实现简单

缺点：

- 线程需要等待，性能受影响
- 可能有死锁风险

逻辑过期

优点：

- 线程无需等待，性能较好

缺点：

- 不保证一致性
- 有额外内存消耗
- 实现复杂

解决缓存击穿

互斥锁

使用 SETNX 命令来实现互斥锁

```
    @Resource
    StringRedisTemplate stringRedisTemplate;

    //线程池
    private static final ExecutorService CACHE_REBUILD_EXECUTOR =
Executors.newFixedThreadPool(10);

    /**
     * 向redis里添加数据
     *
     * @param redisKey   redis的key
     * @param value      数据
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     */
    public void set(String redisKey, Object value, Long expireTime, TimeUnit
timeUnit)
    {
        stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(value), expireTime, timeUnit);
    }

    /**
     * 获取锁
     *
     * @param key redisKey
     * @return 获取锁成功，返回true，否则返回false
     */
    private boolean tryLock(String key)
    {
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
                RedisConstants.LOCK_SHOP_TTL, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(result);
    }

    /**
     * 释放锁
     *
     * @param key redisKey
     */
    private void unlock(String key)
    {
        stringRedisTemplate.delete(key);
    }



    /**
     * 查询数据，解决缓存穿透，互斥锁方法解决缓存击穿，解决缓存雪崩
     *
     * @param keyPrefix                         redisKey的前缀
```

```
     * @param lockKeyPrefix                    锁的前缀
     * @param id                               id
     * @param type                             返回值的类型
     * @param dbFallback                       查询数据库的函数
     * @param expireTime                       过期时间
     * @param timeUnit                         时间单位
     * @param <R>                              返回值的类型
     * @param <ID>                             id的类型
     * @param maxTimeSecondsByCacheAvalanche this.set(redisKey, r,
     *
timeUnit.toSeconds(expireTime)+getIntRandom(0,maxTimeSecondsByCacheAvalanche),
TimeUnit.SECONDS);
     * @return 泛型R
     */
    public <R, ID> R query(String keyPrefix, String lockKeyPrefix, ID id,
Class<R> type,
                           Function<ID, R> dbFallback, Long expireTime, TimeUnit
timeUnit,
                           Integer maxTimeSecondsByCacheAvalanche)
    {
        //获取redisKey
        String redisKey = keyPrefix + id;
        //从redis中查询信息，根据id
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断取出的数据是否为空
        if (StrUtil.isNotBlank(json))
        {
            //不是空，redis里有，返回
            return JSONUtil.toBean(json, type);
        }
        //是空串，不是null，返回
        if (json != null)
        {
            return null;
        }
        //锁的key
        String lockKey = lockKeyPrefix + id;

        R r = null;
        try
        {
            //获取互斥锁
            boolean lock = tryLock(lockKey);
            //判断锁是否获取成功
            if (!lock)
            {
                //没有获取到锁
                //200毫秒后再次获取
                Thread.sleep(200);
                //递归调用
                return query(keyPrefix, lockKeyPrefix, id, type, dbFallback,
                        expireTime, timeUnit, maxTimeSecondsByCacheAvalanche);
            }
            //得到了锁
            //null，查数据库
            r = dbFallback.apply(id);
            //判断数据库里的信息是否为空
            if (r == null)
```

```
            {
                //数据库也为空，缓存空值
                this.set(redisKey, "",
                        timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                        TimeUnit.SECONDS);
                return null;
            }
            //存在，回写到redis里，设置随机的过期时间
            this.set(redisKey, r,
                    timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                    TimeUnit.SECONDS);
        }
        catch (InterruptedException e)
        {
            throw new RuntimeException(e);
        }
        finally
        {
            //释放锁
            this.unlock(lockKey);
        }
        //返回数据
        return r;
    }

    /**
     * 获取一个随机数，区间包含min和max
     *
     * @param min 最小值
     * @param max 最大值
     * @return int 型的随机数
     */
    @SuppressWarnings("all")
    private int getIntRandom(int min, int max)
    {
        if (min > max)
        {
            min = max;
        }
        return min + (int) (Math.random() * (max - min + 1));
    }
```

逻辑过期

```
     @Resource
    StringRedisTemplate stringRedisTemplate;

    //线程池
    private static final ExecutorService CACHE_REBUILD_EXECUTOR =
Executors.newFixedThreadPool(10);
```

```
    /**
     * 向redis里添加数据
     *
     * @param redisKey   redis的key
     * @param value      数据
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     */
    public void set(String redisKey, Object value, Long expireTime, TimeUnit
timeUnit)
    {
        stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(value), expireTime, timeUnit);
    }


    /**
     * 向redis里添加数据 设置逻辑过期
     *
     * @param redisKey   redis的key
     * @param value      数据
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     */
    public void setWithLogicalExpire(String redisKey, Object value, Long
expireTime, TimeUnit timeUnit)
    {
        RedisData redisData = new RedisData();
        //添加数据
        redisData.setData(value);
        //设置过期时间

 redisData.setExpireTime(LocalDateTime.now().plusSeconds(timeUnit.toSeconds(expi
reTime)));
        //放入redis
        stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(redisData));
    }


    /**
     * @param keyPrefix     redisKey的前缀
     * @param lockKeyPrefix 锁的前缀
     * @param id            id
     * @param type          返回值的类型
     * @param dbFallback    查询数据库的函数
     * @param time          过期时间
     * @param timeUnit      时间单位
     * @param <R>           返回值的类型
     * @param <ID>          id的类型
     * @return 泛型R
     */
    public <R, ID> R queryWithLogicalExpire(String keyPrefix, String
lockKeyPrefix, ID id, Class<R> type,
                                            Function<ID, R> dbFallback, Long
time, TimeUnit timeUnit)
    {
        //获得前缀
```

```
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断是否为空
        if (StrUtil.isBlank(json))
        {
            //空，返回
            return null;
        }
        //不为空
        //json 反序列化为对象
        RedisData redisData = JSONUtil.toBean(json, RedisData.class);
        //获得过期时间
        LocalDateTime expireTime = redisData.getExpireTime();
        //获取数据
        R r = JSONUtil.toBean((JSONObject) redisData.getData(), type);
        //判断是否过期
        if (expireTime.isAfter(LocalDateTime.now()))
        {
            //未过期，返回
            return r;
        }
        //过期，缓存重建
        //获取互斥锁
        String lockKey = lockKeyPrefix + id;
        boolean isLock = tryLock(lockKey);
        if (isLock)
        {
            //获取锁成功
            // 开辟独立线程
            CACHE_REBUILD_EXECUTOR.submit(new Runnable()
            {
                @Override
                public void run()
                {
                    try
                    {
                        R r1 = dbFallback.apply(id);
                        setWithLogicalExpire(redisKey, r1, time, timeUnit);
                    }
                    catch (Exception e)
                    {
                        throw new RuntimeException(e);
                    }
                    finally
                    {
                        //释放锁
                        unlock(lockKey);
                    }
                }
            });
        }
        //没有获取到锁，使用旧数据返回
        return r;
    }


    /**
```

```
     * 获取锁
     *
     * @param key redisKey
     * @return 获取锁成功，返回true，否则返回false
     */
    private boolean tryLock(String key)
    {
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
                RedisConstants.LOCK_SHOP_TTL, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(result);
    }

    /**
     * 释放锁
     *
     * @param key redisKey
     */
    private void unlock(String key)
    {
        stringRedisTemplate.delete(key);
    }


    /**
     * 获取一个随机数，区间包含min和max
     *
     * @param min 最小值
     * @param max 最大值
     * @return int 型的随机数
     */
    @SuppressWarnings("all")
    private int getIntRandom(int min, int max)
    {
        if (min > max)
        {
            min = max;
        }
        return min + (int) (Math.random() * (max - min + 1));
    }
```

redisData:

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

redis工具类

```
package mao.spring_boot_redis_hmdp.utils;

import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import cn.hutool.json.JSONObject;
import cn.hutool.json.JSONUtil;
import lombok.extern.slf4j.Slf4j;
import mao.spring_boot_redis_hmdp.dto.RedisData;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

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

/**
 * Project name(项目名称)：spring_boot_redis_hmdp_redis_utils
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Class(类名): RedisUtils
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/14
 * Time(创建时间)： 20:48
 * Version(版本): 1.0
 * Description(描述)： redis工具类
 */

@Slf4j
@Component
public class RedisUtils
{

    @Resource
    StringRedisTemplate stringRedisTemplate;

    //线程池
    private static final ExecutorService CACHE_REBUILD_EXECUTOR =
Executors.newFixedThreadPool(10);

    /**
     * 向redis里添加数据
     *
     * @param redisKey   redis的key
     * @param value      数据
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     */
```

```
    public void set(String redisKey, Object value, Long expireTime, TimeUnit
timeUnit)
    {
        stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(value), expireTime, timeUnit);
    }


    /**
     * 向redis里添加数据 设置逻辑过期
     *
     * @param redisKey   redis的key
     * @param value      数据
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     */
    public void setWithLogicalExpire(String redisKey, Object value, Long
expireTime, TimeUnit timeUnit)
    {
        RedisData redisData = new RedisData();
        //添加数据
        redisData.setData(value);
        //设置过期时间

 redisData.setExpireTime(LocalDateTime.now().plusSeconds(timeUnit.toSeconds(expi
reTime)));
        //放入redis
        stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(redisData));
    }


    /**
     * 查询数据，有缓存，解决缓存穿透问题，未解决缓存雪崩问题
     *
     * @param keyPrefix  redisKey的前缀
     * @param id         id
     * @param type       返回值的类型
     * @param dbFallback 查询数据库的函数
     * @param expireTime 过期时间
     * @param timeUnit   时间单位
     * @param <R>        返回值的类型
     * @param <ID>       id的类型
     * @return 泛型R
     */
    public <R, ID> R queryWithPassThrough(String keyPrefix, ID id, Class<R>
type,
                                          Function<ID, R> dbFallback, Long
expireTime, TimeUnit timeUnit)
    {
        //获得前缀
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断是否为空
        if (StrUtil.isNotBlank(json))
        {
            //不为空，返回
```

```
            return JSONUtil.toBean(json, type);
        }
        //判断是否为空串
        if (json != null)
        {
            //空串
            return null;
        }
        //null
        //查数据库
        R r = dbFallback.apply(id);
        //判断
        if (r == null)
        {
            //数据库也为空，缓存空值
            this.set(redisKey, "", expireTime, timeUnit);
            return null;
        }
        //数据库存在，写入redis
        this.set(redisKey, r, expireTime, timeUnit);
        //返回
        return r;
    }

    /**
     * 查询数据，有缓存，解决缓存穿透问题，解决缓存雪崩问题
     *
     * @param keyPrefix                      redisKey的前缀
     * @param id                             id
     * @param type                           返回值的类型
     * @param dbFallback                     查询数据库的函数
     * @param expireTime                     过期时间
     * @param timeUnit                       时间单位
     * @param <R>                            返回值的类型
     * @param <ID>                           id的类型
     * @param maxTimeSecondsByCacheAvalanche this.set(redisKey, r,
     *
timeUnit.toSeconds(expireTime)+getIntRandom(0,maxTimeSecondsByCacheAvalanche),
TimeUnit.SECONDS);
     * @return 泛型R
     */
    public <R, ID> R queryWithPassThroughAndCacheAvalanche(String keyPrefix, ID
id, Class<R> type,
                                                           Function<ID, R>
dbFallback, Long expireTime, TimeUnit timeUnit,
                                                           Integer
maxTimeSecondsByCacheAvalanche)
    {
        //获得前缀
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断是否为空
        if (StrUtil.isNotBlank(json))
        {
            //不为空，返回
            return JSONUtil.toBean(json, type);
        }
```

```
        //判断是否为空串
        if (json != null)
        {
            //空串
            return null;
        }
        //null
        //查数据库
        R r = dbFallback.apply(id);
        //判断
        if (r == null)
        {
            //数据库也为空，缓存空值
            this.set(redisKey, "",
                    timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                    TimeUnit.SECONDS);
            return null;
        }
        //数据库存在，写入redis
        this.set(redisKey, r,
                timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                TimeUnit.SECONDS);
        //返回
        return r;
    }

    /**
     * 查询数据，解决缓存穿透，互斥锁方法解决缓存击穿，解决缓存雪崩
     *
     * @param keyPrefix                      redisKey的前缀
     * @param lockKeyPrefix                  锁的前缀
     * @param id                             id
     * @param type                           返回值的类型
     * @param dbFallback                     查询数据库的函数
     * @param expireTime                     过期时间
     * @param timeUnit                       时间单位
     * @param <R>                            返回值的类型
     * @param <ID>                           id的类型
     * @param maxTimeSecondsByCacheAvalanche this.set(redisKey, r,
     *
timeUnit.toSeconds(expireTime)+getIntRandom(0,maxTimeSecondsByCacheAvalanche),
TimeUnit.SECONDS);
     * @return 泛型R
     */
    public <R, ID> R query(String keyPrefix, String lockKeyPrefix, ID id,
Class<R> type,
                           Function<ID, R> dbFallback, Long expireTime, TimeUnit
timeUnit,
                           Integer maxTimeSecondsByCacheAvalanche)
    {
        //获取redisKey
        String redisKey = keyPrefix + id;
        //从redis中查询信息，根据id
        String json = stringRedisTemplate.opsForValue().get(redisKey);
        //判断取出的数据是否为空
        if (StrUtil.isNotBlank(json))
```

```
        {
            //不是空，redis里有，返回
            return JSONUtil.toBean(json, type);
        }
        //是空串，不是null，返回
        if (json != null)
        {
            return null;
        }
        //锁的key
        String lockKey = lockKeyPrefix + id;

        R r = null;
        try
        {
            //获取互斥锁
            boolean lock = tryLock(lockKey);
            //判断锁是否获取成功
            if (!lock)
            {
                //没有获取到锁
                //200毫秒后再次获取
                Thread.sleep(200);
                //递归调用
                return query(keyPrefix, lockKeyPrefix, id, type, dbFallback,
                        expireTime, timeUnit, maxTimeSecondsByCacheAvalanche);
            }
            //得到了锁
            //null，查数据库
            r = dbFallback.apply(id);
            //判断数据库里的信息是否为空
            if (r == null)
            {
                //数据库也为空，缓存空值
                this.set(redisKey, "",
                        timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                        TimeUnit.SECONDS);
                return null;
            }
            //存在，回写到redis里，设置随机的过期时间
            this.set(redisKey, r,
                    timeUnit.toSeconds(expireTime) + getIntRandom(0,
maxTimeSecondsByCacheAvalanche),
                    TimeUnit.SECONDS);
        }
        catch (InterruptedException e)
        {
            throw new RuntimeException(e);
        }
        finally
        {
            //释放锁
            this.unlock(lockKey);
        }
        //返回数据
        return r;
    }
```

```
    /**
     * 更新数据
     *
     * @param id          要更新的主键
     * @param data        要更新的对象
     * @param keyPrefix   redis的key前缀
     * @param dbFallback 更新数据库的函数，返回值要为Boolean类型
     * @param <T>         要更新的对象的泛型
     * @param <ID>        主键的类型
     * @return boolean
     */
    public <T, ID> boolean update(ID id, T data, String keyPrefix, Function<T,
Boolean> dbFallback)
    {
        //判断是否为空
        if (id == null)
        {
            return false;
        }
        //不为空
        //先更新数据库
        boolean b = dbFallback.apply(data);
        //更新失败，返回
        if (!b)
        {
            return false;
        }
        //更新没有失败
        //删除redis里的数据，下一次查询时自动添加进redis
        //redisKey
        String redisKey = keyPrefix + id;
        stringRedisTemplate.delete(redisKey);
        //返回响应
        return true;
    }

    /**
     * @param keyPrefix     redisKey的前缀
     * @param lockKeyPrefix 锁的前缀
     * @param id            id
     * @param type          返回值的类型
     * @param dbFallback    查询数据库的函数
     * @param time          过期时间
     * @param timeUnit      时间单位
     * @param <R>           返回值的类型
     * @param <ID>          id的类型
     * @return 泛型R
     */
    public <R, ID> R queryWithLogicalExpire(String keyPrefix, String
lockKeyPrefix, ID id, Class<R> type,
                                            Function<ID, R> dbFallback, Long
time, TimeUnit timeUnit)
    {
        //获得前缀
        String redisKey = keyPrefix + id;
        //查询redis
        String json = stringRedisTemplate.opsForValue().get(redisKey);
```

```
        //判断是否为空
        if (StrUtil.isBlank(json))
        {
            //空，返回
            return null;
        }
        //不为空
        //json 反序列化为对象
        RedisData redisData = JSONUtil.toBean(json, RedisData.class);
        //获得过期时间
        LocalDateTime expireTime = redisData.getExpireTime();
        //获取数据
        R r = JSONUtil.toBean((JSONObject) redisData.getData(), type);
        //判断是否过期
        if (expireTime.isAfter(LocalDateTime.now()))
        {
            //未过期，返回
            return r;
        }
        //过期，缓存重建
        //获取互斥锁
        String lockKey = lockKeyPrefix + id;
        boolean isLock = tryLock(lockKey);
        if (isLock)
        {
            //获取锁成功
            // 开辟独立线程
            CACHE_REBUILD_EXECUTOR.submit(new Runnable()
            {
                @Override
                public void run()
                {
                    try
                    {
                        R r1 = dbFallback.apply(id);
                        setWithLogicalExpire(redisKey, r1, time, timeUnit);
                    }
                    catch (Exception e)
                    {
                        throw new RuntimeException(e);
                    }
                    finally
                    {
                        //释放锁
                        unlock(lockKey);
                    }
                }
            });
        }
        //没有获取到锁，使用旧数据返回
        return r;
    }


    /**
     * 获取锁
     *
     * @param key redisKey
```

```
     * @return 获取锁成功，返回true，否则返回false
     */
    private boolean tryLock(String key)
    {
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
                RedisConstants.LOCK_SHOP_TTL, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(result);
    }

    /**
     * 释放锁
     *
     * @param key redisKey
     */
    private void unlock(String key)
    {
        stringRedisTemplate.delete(key);
    }


    /**
     * 获取一个随机数，区间包含min和max
     *
     * @param min 最小值
     * @param max 最大值
     * @return int 型的随机数
     */
    @SuppressWarnings("all")
    private int getIntRandom(int min, int max)
    {
        if (min > max)
        {
            min = max;
        }
        return min + (int) (Math.random() * (max - min + 1));
    }
}
```

redisData:

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

优惠券秒杀

全局唯一ID

- 唯一性
- 高可用
- 高性能
- 递增型
- 安全性

组成：

- 符号位：1 bit，永远为 0
- 时间戳：31 bit，以秒为单位，可以使用 69 年
- 序列号：32 bit，秒内的计数器，支持每秒产生 2^32 个不同的 ID

实现

```
package mao.spring_boot_redis_hmdp.utils;

import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;

/**
 * Project name(项目名称)：spring_boot_redis_hmdp_global_id_generator
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Class(类名): RedisIDGenerator
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/15
 * Time(创建时间)： 12:24
 * Version(版本): 1.0
 * Description(描述)： id生成器
 */

@Component
public class RedisIDGenerator
{
    @Resource
    private StringRedisTemplate stringRedisTemplate;

    /**
     * 2022年1月1日的时间
     */
    private static final long BEGIN_TIMESTAMP = 1640995200L;

    /**
     * 序列号位数
     */
```

```
    private static int COUNT_BITS = 32;

    /**
     * 获取一个id
     *
     * @param prefix 前缀
     * @return id
     */
    public Long nextID(String prefix)
    {
        //获取当前时间
        LocalDateTime now = LocalDateTime.now();
        //转换成秒数
        long nowSecond = now.toEpochSecond(ZoneOffset.UTC);
        //获得时间差
        long time = nowSecond - BEGIN_TIMESTAMP;
        //格式化成字符串
        String format = now.format(DateTimeFormatter.ofPattern("yyyy:MM:dd"));
        //将key下存储为字符串值的整数值加一
        Long count = stringRedisTemplate.opsForValue().increment("id:" + prefix
+ ":" + format);
        return time << COUNT_BITS | count;
    }

}
```

实现优惠券秒杀下单

实现

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;
import java.time.LocalDateTime;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
```

```
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        //查询优惠券
        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
        //判断是否存在
        if (seckillVoucher == null)
        {
            return Result.fail("活动不存在");
        }
        //判断是否开始
        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
        {
            //未开始
            return Result.fail("秒杀活动未开始");
        }
        //判断是否结束
        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
        {
            //结束
            return Result.fail("秒杀活动已经结束");
        }
        //判断库存是否充足
        if (seckillVoucher.getStock() <= 0)
        {
            //库存不足
            return Result.fail("库存不足");
        }
        //创建订单
        return this.createVoucherOrder(voucherId);
    }


    /**
     * 创建订单
     *
     * @param voucherId voucherId
     * @return Result
     */
    @Transactional
    public Result createVoucherOrder(Long voucherId)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        synchronized (userID.toString().intern())
        {
            //查询数据库
```

```
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return Result.fail("不能重复下单");
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return Result.fail("库存扣减失败");
            }
            //扣减成功
            //生成订单
            VoucherOrder voucherOrder = new VoucherOrder();
            //生成id
            Long orderID = redisIDGenerator.nextID("order");
            voucherOrder.setVoucherId(voucherId);
            voucherOrder.setId(orderID);
            //设置用户
            //Long userID = UserHolder.getUser().getId();
            voucherOrder.setUserId(userID);
            //保存订单
            this.save(voucherOrder);
            //返回
            return Result.ok(orderID);
        }
    }
}
```

超卖问题

原因

在高并发情况下，假设线程 1 查询库存，查询结果为 1 ，当线程 1 准备要去扣减库存时，其他线程也去查询库存，结果查询出来的库存数也是 1，那么这时所有的线程查询到的库存数都是大于 0 的，所有的线程都会去执行扣减操作，就会导致超卖问题。

常见解决方案

悲观锁

- 认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行
- 例如Synchronized、Lock都属于 悲观锁

乐观锁

- 认为线程安全问题不一定会发生，因 此不加锁，只是在更新数据时去判断 有没有其它线程对数据做了修改。
- 如果没有修改则认为是安全的，自 己才更新数据。
- 如果已经被其它线程修改说明发生 了安全问题，此时可以重试或异常。

乐观锁实现

- 版本号法：给查询得到的数据加一个版本号，在多线程并发的时候，基于版本号来判断数据有没有被修改过，每当数据被修改，版本号就会加1。
- CAS（Compare And Swap）法：即比较和替换法，是在版本号法的基础上改进而来。CAS 法去除了版本号法中的版本号信息，以库存信息本身有没有变化为判断依据，当线程修改库存时，判断当前数据库中的库存与之前查询得到的库存数据是否一致，如果一致，则说明线程安全，可以执行扣减操作，如果不一致，则说明线程不安全，扣减失败。

优点和缺点

乐观锁：

- 优点：性能好
- 缺点：存在成功率低的问题。

悲观锁：

- 优点：简单粗暴
- 缺点：性能一般

代码改进

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;
import java.time.LocalDateTime;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
```

```
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        //查询优惠券
        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
        //判断是否存在
        if (seckillVoucher == null)
        {
            return Result.fail("活动不存在");
        }
        //判断是否开始
        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
        {
            //未开始
            return Result.fail("秒杀活动未开始");
        }
        //判断是否结束
        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
        {
            //结束
            return Result.fail("秒杀活动已经结束");
        }
        //判断库存是否充足
        if (seckillVoucher.getStock() <= 0)
        {
            //库存不足
            return Result.fail("库存不足");
        }
        //创建订单
        return this.createVoucherOrder(voucherId);
    }


    /**
     * 创建订单
     *
     * @param voucherId voucherId
     * @return Result
     */
    @Transactional
    public Result createVoucherOrder(Long voucherId)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        synchronized (userID.toString().intern())
        {
            //查询数据库
```

```
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return Result.fail("不能重复下单");
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return Result.fail("库存扣减失败");
            }
            //扣减成功
            //生成订单
            VoucherOrder voucherOrder = new VoucherOrder();
            //生成id
            Long orderID = redisIDGenerator.nextID("order");
            voucherOrder.setVoucherId(voucherId);
            voucherOrder.setId(orderID);
            //设置用户
            //Long userID = UserHolder.getUser().getId();
            voucherOrder.setUserId(userID);
            //保存订单
            this.save(voucherOrder);
            //返回
            return Result.ok(orderID);
        }
    }
}
```

实现一人一单功能

要求同一个优惠券，一个用户只能下一单

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
import javax.annotation.Resource;
import java.time.LocalDateTime;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        //查询优惠券
        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
        //判断是否存在
        if (seckillVoucher == null)
        {
            return Result.fail("活动不存在");
        }
        //判断是否开始
        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
        {
            //未开始
            return Result.fail("秒杀活动未开始");
        }
        //判断是否结束
        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
        {
            //结束
            return Result.fail("秒杀活动已经结束");
        }
        //判断库存是否充足
        if (seckillVoucher.getStock() <= 0)
        {
            //库存不足
            return Result.fail("库存不足");
        }
        //创建订单
        return this.createVoucherOrder(voucherId);
    }


    /**
     * 创建订单
     *
     * @param voucherId voucherId
     * @return Result
     */
    @Transactional
    public Result createVoucherOrder(Long voucherId)
    {
```

```
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        synchronized (userID.toString().intern())
        {
            //查询数据库
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return Result.fail("不能重复下单");
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return Result.fail("库存扣减失败");
            }
            //扣减成功
            //生成订单
            VoucherOrder voucherOrder = new VoucherOrder();
            //生成id
            Long orderID = redisIDGenerator.nextID("order");
            voucherOrder.setVoucherId(voucherId);
            voucherOrder.setId(orderID);
            //设置用户
            //Long userID = UserHolder.getUser().getId();
            voucherOrder.setUserId(userID);
            //保存订单
            this.save(voucherOrder);
            //返回
            return Result.ok(orderID);
        }
    }
}
```

分布式锁

是什么?

满足分布式系统或集群模式下多进程可见并且互斥的锁。

特点

- 多进程可见
- 互斥：必须确保只能有一个线程拿到互斥锁。
- 高可用：必须保证大多数情况下获取锁都是成功的
- 高性能：加锁以后就会影响业务的性能，加锁后，业务的执行变成串行执行，如果获取锁的动作又很慢，就会导致执行效率更低，雪上加霜。
- 安全性：在获取锁的时候应当考虑一些异常情况，比如获取后还未释放，服务器宕机，锁需要怎么处理，又或者会不会产生死锁问题。

分布式锁的实现方式

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

基于Redis的分布式锁

获取锁：

```
SETNX lock thread1
EXPIRE lock 10
```

释放锁：

DEL key

实现

基本锁：

```
    /**
     * 获取锁
     *
     * @param key redisKey
     * @return 获取锁成功，返回true，否则返回false
     */
    private boolean tryLock(String key)
    {
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
                RedisConstants.LOCK_SHOP_TTL, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(result);
    }

    /**
     * 释放锁
     *
```

```
     * @param key redisKey
     */
    private void unlock(String key)
    {
        stringRedisTemplate.delete(key);
    }
```

改进：

锁接口：

```
package mao.spring_boot_redis_hmdp.utils;

/**
 * Project name(项目名称)：
spring_boot_redis_hmdp_distributed_lock_realize_the_function_of_one_person_and_o
ne_order
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Interface(接口名): RedisLock
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 10:51
 * Version(版本): 1.0
 * Description(描述)： 无
 */

public interface RedisLock
{
    /**
     * 尝试获取锁
     *
     * @param timeoutSec 超时时间
     * @return boolean，成功返回true，否则返回false
     */
    boolean tryLock(long timeoutSec);

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

锁实现：

```
package mao.spring_boot_redis_hmdp.utils;

import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

import java.util.concurrent.TimeUnit;

/**
```

```
 * Project name(项目名称)：
spring_boot_redis_hmdp_distributed_lock_realize_the_function_of_one_person_and_o
ne_order
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Class(类名): RedisLockImpl
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 10:53
 * Version(版本): 1.0
 * Description(描述)： 简单分布式锁，非单例
 */


public class RedisLockImpl implements RedisLock
{
    /**
     * 锁的名称
     */
    private String name;
    /**
     * StringRedisTemplate
     */
    private StringRedisTemplate stringRedisTemplate;

    /**
     * 锁前缀
     */
    private static final String KEY_PREFIX = "lock:";

    /**
     * 构造函数
     *
     * @param name                锁的名称
     * @param stringRedisTemplate StringRedisTemplate
     */
    public RedisLockImpl(String name, StringRedisTemplate stringRedisTemplate)
    {
        this.name = name;
        this.stringRedisTemplate = stringRedisTemplate;
    }

    @Override
    public boolean tryLock(long timeoutSec)
    {
        //获得线程标识
        long threadID = Thread.currentThread().getId();
        //锁key
        String lockKey = KEY_PREFIX + name;
        //获取锁
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(lockKey,
String.valueOf(threadID),
                timeoutSec, TimeUnit.SECONDS);
        //返回
        return Boolean.TRUE.equals(result);
    }
```

```
    @Override
    public void unlock()
    {
        //锁key
        String lockKey = KEY_PREFIX + name;
        //释放
        stringRedisTemplate.delete(lockKey);
    }
}
```

业务：

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import mao.spring_boot_redis_hmdp.utils.RedisLock;
import mao.spring_boot_redis_hmdp.utils.RedisLockImpl;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;
import java.time.LocalDateTime;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        //查询优惠券
        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
        //判断是否存在
```

```
        if (seckillVoucher == null)
        {
            return Result.fail("活动不存在");
        }
        //判断是否开始
        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
        {
            //未开始
            return Result.fail("秒杀活动未开始");
        }
        //判断是否结束
        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
        {
            //结束
            return Result.fail("秒杀活动已经结束");
        }
        //判断库存是否充足
        if (seckillVoucher.getStock() <= 0)
        {
            //库存不足
            return Result.fail("库存不足");
        }
        //创建订单
        return this.createVoucherOrder(voucherId);
    }


    /**
     * 创建订单
     *
     * @param voucherId voucherId
     * @return Result
     */
    @Transactional
    public Result createVoucherOrder(Long voucherId)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        //创建锁对象
        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
        //取得锁
        boolean isLock = redisLock.tryLock(500);
        //判断
        if (!isLock)
        {
            return Result.fail("不允许重复下单! ");
        }

        //获取锁成功
        //synchronized (userID.toString().intern())
        try
        {
            //查询数据库
```

```
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return Result.fail("不能重复下单");
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return Result.fail("库存扣减失败");
            }
            //扣减成功
            //生成订单
            VoucherOrder voucherOrder = new VoucherOrder();
            //生成id
            Long orderID = redisIDGenerator.nextID("order");
            voucherOrder.setVoucherId(voucherId);
            voucherOrder.setId(orderID);
            //设置用户
            //Long userID = UserHolder.getUser().getId();
            voucherOrder.setUserId(userID);
            //保存订单
            this.save(voucherOrder);
            //返回
            return Result.ok(orderID);
        }
        finally
        {
            //释放锁
            redisLock.unlock();
        }
    }
}
```

再次改进：

```
package mao.spring_boot_redis_hmdp.utils;

import cn.hutool.core.lang.UUID;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

import java.util.concurrent.TimeUnit;

/**
```

```
 * Project name(项目名称)：
spring_boot_redis_hmdp_distributed_lock_realize_the_function_of_one_person_and_o
ne_order
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Class(类名): RedisLockImpl
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 10:53
 * Version(版本): 1.0
 * Description(描述)： 简单分布式锁，非单例
 */


public class RedisLockImpl implements RedisLock
{
    /**
     * 锁的名称
     */
    private String name;
    /**
     * StringRedisTemplate
     */
    private StringRedisTemplate stringRedisTemplate;

    private static final String ID_PREFIX = UUID.randomUUID().toString(true) +
"_";


    /**
     * 锁前缀
     */
    private static final String KEY_PREFIX = "lock:";

    /**
     * 构造函数
     *
     * @param name                锁的名称
     * @param stringRedisTemplate StringRedisTemplate
     */
    public RedisLockImpl(String name, StringRedisTemplate stringRedisTemplate)
    {
        this.name = name;
        this.stringRedisTemplate = stringRedisTemplate;
    }

    @Override
    public boolean tryLock(long timeoutSec)
    {
        //获得线程标识
        long threadID = Thread.currentThread().getId();
        //锁key
        String lockKey = KEY_PREFIX + name;
        //获取锁
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(lockKey,
String.valueOf(threadID),
                timeoutSec, TimeUnit.SECONDS);
```

```
        //返回
        return Boolean.TRUE.equals(result);
    }

    @Override
    public void unlock()
    {
        // 获取线程标识
        String threadID = ID_PREFIX + Thread.currentThread().getId();
        //锁key
        String lockKey = KEY_PREFIX + name;
        // 获取锁中的标识
        String id = stringRedisTemplate.opsForValue().get(lockKey);
        //判断锁是否是自己的，通过线程id来判断
        if (threadID.equals(id))
        {
            //释放
            stringRedisTemplate.delete(lockKey);
        }
    }
}
```

再次改进：

lua脚本：

```
-- 这里的 KEYS[1] 就是锁的 key，这里的 ARGV[1] 就是当前线程标识
-- 获取锁中的标识，判断是否与当前线程标识一致
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
```

业务：

```
package mao.spring_boot_redis_hmdp.utils;

import cn.hutool.core.lang.UUID;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.stereotype.Component;

import java.util.Collections;
import java.util.concurrent.TimeUnit;

/**
```

```
 * Project name(项目名称)：
spring_boot_redis_hmdp_distributed_lock_realize_the_function_of_one_person_and_o
ne_order
 * Package(包名): mao.spring_boot_redis_hmdp.utils
 * Class(类名): RedisLockImpl
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 10:53
 * Version(版本): 1.0
 * Description(描述)： 简单分布式锁，非单例
 */

@Slf4j
public class RedisLockImpl implements RedisLock
{
    /**
     * 锁的名称
     */
    private final String name;
    /**
     * StringRedisTemplate
     */
    private final StringRedisTemplate stringRedisTemplate;

    private static final String ID_PREFIX = UUID.randomUUID().toString(true) +
"-";

    private static final DefaultRedisScript<Long> UNLOCK_SCRIPT;

    static
    {
        //创建对象
        UNLOCK_SCRIPT = new DefaultRedisScript<>();
        //加载类路径下的资源
        UNLOCK_SCRIPT.setLocation(new ClassPathResource("unlock.lua"));
        //设置返回值的类型
        UNLOCK_SCRIPT.setResultType(Long.class);
    }

    /**
     * 锁前缀
     */
    private static final String KEY_PREFIX = "lock:";

    /**
     * 构造函数
     *
     * @param name                锁的名称
     * @param stringRedisTemplate StringRedisTemplate
     */
    public RedisLockImpl(String name, StringRedisTemplate stringRedisTemplate)
    {
        this.name = name;
        this.stringRedisTemplate = stringRedisTemplate;
    }
```

```
    @Override
    public boolean tryLock(long timeoutSec)
    {
        //获得线程标识
        long threadID = Thread.currentThread().getId();
        //锁key
        String lockKey = KEY_PREFIX + name;
        //获取锁
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(lockKey,
String.valueOf(threadID),
                timeoutSec, TimeUnit.SECONDS);
        //返回
        return Boolean.TRUE.equals(result);
    }

    @Override
    public void unlock()
    {
        // 获取线程标识
        String threadID = ID_PREFIX + Thread.currentThread().getId();
        //锁key
        String lockKey = KEY_PREFIX + name;

        /*// 获取锁中的标识
        String id = stringRedisTemplate.opsForValue().get(lockKey);
        //判断锁是否是自己的，通过线程id来判断
        if (threadID.equals(id))
        {
            //释放
            stringRedisTemplate.delete(lockKey);
        }*/

        //执行lua脚本
        Long result = stringRedisTemplate.execute(UNLOCK_SCRIPT,
Collections.singletonList(lockKey), threadID);
        if (result == null || result == 0)
        {
            //释放到了别人的锁
            log.debug("释放锁异常");
        }
    }
}
```

Redisson

依赖:

```
<!--spring boot redisson 依赖-->
        <dependency>
            <groupId>org.redisson</groupId>
            <artifactId>redisson-spring-boot-starter</artifactId>
            <version>3.17.0</version>
        </dependency>
```

配置:

```
package mao.spring_boot_redis_hmdp.config;

import org.redisson.Redisson;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Project name(项目名称)：
spring_boot_redis_hmdp_distributed_lock_based_on_redisson
 * Package(包名): mao.spring_boot_redis_hmdp.config
 * Class(类名): RedissonConfig
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 13:40
 * Version(版本): 1.0
 * Description(描述)： Redisson的配置
 */

@Configuration
public class RedissonConfig
{
    /**
     * Redisson配置
     *
     * @return RedissonClient
     */
    @Bean
    public RedissonClient redissonClient()
    {
        //配置类
        Config config = new Config();
        //添加redis地址，用config.useClusterServers()添加集群地址

 config.useSingleServer().setAddress("redis://127.0.0.1:6379").setPassword("1234
56");
        //创建客户端
        return Redisson.create(config);
    }
}
```

使用：

```
@Resource
private RedissonClient redissonClient;
@Test
void testRedisson() throws InterruptedException {
// 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);
// 判断释放获取成功
if(isLock){
try {
System.out.println("执行业务");
}finally {
// 释放锁
lock.unlock();
}
}
}
```

相关概念

- 可重入：利用 hash 结构记录线程 id 和重入次数。每次获取锁时，先判断锁是否存在，如果不存在，则直接获取，如果已经存在，且线程标识为当前线程，则可以再次获取，并将重入次数加 1。释放锁时，每释放一次，重入次数减 1，直至重入次数减为 0，则证明所有的业务已经执行结束，则可以直接释放锁。
- 可重试：在第一次尝试获取锁失败后，并不是立即失败，而是去等待释放锁的信号（利用了 Redis 中 PubSub 机制）。而获取锁成功的线程在释放锁的时候，就会向等待中的线程发送一条消息，等待中的线程捕获到消息后，就可以重新尝试获取锁。如果重试失败，则会继续等待释放锁的信号，然后再去重试。当然，重试并不是无限次的，会有一个等待时间，如果超过等待时间，就结束重试。
- 超时续约：利用watchDog，每隔一段时间（releaseTime / 3），重置超时时间

实现

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import mao.spring_boot_redis_hmdp.utils.RedisLock;
```

```
import mao.spring_boot_redis_hmdp.utils.RedisLockImpl;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;
import java.time.LocalDateTime;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Resource
    private RedissonClient redissonClient;

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        //查询优惠券
        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
        //判断是否存在
        if (seckillVoucher == null)
        {
            return Result.fail("活动不存在");
        }
        //判断是否开始
        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
        {
            //未开始
            return Result.fail("秒杀活动未开始");
        }
        //判断是否结束
        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
        {
            //结束
            return Result.fail("秒杀活动已经结束");
        }
        //判断库存是否充足
        if (seckillVoucher.getStock() <= 0)
        {
            //库存不足
            return Result.fail("库存不足");
        }
        //创建订单
        return this.createVoucherOrder(voucherId);
```

```
    }


    /**
     * 创建订单
     *
     * @param voucherId voucherId
     * @return Result
     */
    @Transactional
    public Result createVoucherOrder(Long voucherId)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        /*//创建锁对象
        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
        //取得锁
        boolean isLock = redisLock.tryLock(1500);
        //判断
        if (!isLock)
        {
            return Result.fail("不允许重复下单！");
        }*/


        //redisson锁
        RLock lock = redissonClient.getLock("lock:order:" + userID);
        //尝试获取锁
        //仅当调用时它是空闲的时才获取锁。
        //如果锁可用，则获取锁并立即返回值为true
        //如果锁不可用，则此方法将立即返回值false
        boolean tryLock = lock.tryLock();
        //判断是否获取成功
        if (!tryLock)
        {
            return Result.fail("不允许重复下单！");
        }
        //获取锁成功
        //synchronized (userID.toString().intern())
        try
        {
            //查询数据库
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return Result.fail("不能重复下单");
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
```

```
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return Result.fail("库存扣减失败");
            }
            //扣减成功
            //生成订单
            VoucherOrder voucherOrder = new VoucherOrder();
            //生成id
            Long orderID = redisIDGenerator.nextID("order");
            voucherOrder.setVoucherId(voucherId);
            voucherOrder.setId(orderID);
            //设置用户
            //Long userID = UserHolder.getUser().getId();
            voucherOrder.setUserId(userID);
            //保存订单
            this.save(voucherOrder);
            //返回
            return Result.ok(orderID);
        }
        finally
        {
            //释放锁
            //redisLock.unlock();
            lock.unlock();
        }
    }
}
```

Redisson 可重入锁

我们可以在获取锁的时候，首先判断锁是否已经被占用，如果已经被占用，判断是否是当前线程所占用的，如果是同一个线程占用，也会让其获取到锁，但是会额外增加一个计数器，用于记录重入的次数，即当前线程总共获取了几次锁。当前线程每获取一次锁，计数器便加1，而在释放锁时，每释放一次锁，计数器就会减1，直至计数器为 0 时，将当前锁直接删除。那么现在，我们不仅要在锁中记录获取锁的线程，还要记录当前线程重入次数，即获取了几次锁，显然，String 类型的数据结构并不满足这个业务需求，这里可以使用 Hash 类型进行存储，这样就可以在 key 的位置，记录锁的名称，在 field 的位置记录线程标识，在 value 的位置记录锁重入次数。

获取锁：

```
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
    -- 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
```

释放锁：

```
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);
-- 判断重入次数是否已经为0
if(count > 0) then
    -- 大于0说明不能是释放锁，重置有效期然后返回
    redis.call('expire', threadId, releaseTime);
    return nil;
else
    redis.call('del', key); -- 等于 0 说明可以释放锁，直接删除
    return nil;
end
```

联锁

配置：

```
@Configuration
public class RedissonConfig {

    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();
        config.useSingleServer().setAddress("redis://虚拟机
ip:6379").setPassword("密码");
        return Redisson.create(config);
    }

    @Bean
    public RedissonClient redissonClient2(){
        Config config = new Config();
        config.useSingleServer().setAddress("redis://虚拟机
ip:6380").setPassword("密码");
```

```
        return Redisson.create(config);
    }


    @Bean
    public RedissonClient redissonClient3(){
        Config config = new Config();
        config.useSingleServer().setAddress("redis://虚拟机
ip:6381").setPassword("密码");
        return Redisson.create(config);
    }
}
```

测试：

```
@Slf4j
@SpringBootTest
class RedissonTest {

    @Resource
    private RedissonClient redissonClient;

    @Resource
    private RedissonClient redissonClient2;

    @Resource
    private RedissonClient redissonClient3;

    private RLock lock;

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

        lock = redissonClient.getMultiLock(lock1, lock2, lock3);
    }

    @Test
    void method1() throws InterruptedException {
        // 尝试获取锁
        boolean isLock = lock.tryLock(1L, TimeUnit.SECONDS);
        if (!isLock) {
            log.error("获取锁失败 .... 1");
            return;
        }
        try {
            log.info("获取锁成功 .... 1");
            method2();
            log.info("开始执行业务 ... 1");
        } finally {
            log.warn("准备释放锁 .... 1");
            lock.unlock();
        }
    }
    void method2() {
```

```
        // 尝试获取锁
        boolean isLock = lock.tryLock();
        if (!isLock) {
            log.error("获取锁失败 .... 2");
            return;
        }
        try {
            log.info("获取锁成功 .... 2");
            log.info("开始执行业务 ... 2");
        } finally {
            log.warn("准备释放锁 .... 2");
            lock.unlock();
        }
    }
}
```

秒杀优化

业务:

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Resource
    private RedissonClient redissonClient;

    private static final DefaultRedisScript<Long> SECKILL_SCRIPT;

    /**
     * 阻塞队列
     */
    private final BlockingQueue<VoucherOrder> blockingQueue = new
ArrayBlockingQueue<>(1024 * 1024);

    /**
     * 线程池
     */
    private static final ExecutorService SECKILL_ORDER_EXECUTOR =
Executors.newSingleThreadExecutor();

    static
    {
        //创建对象
        SECKILL_SCRIPT = new DefaultRedisScript<>();
        //加载
        SECKILL_SCRIPT.setLocation(new ClassPathResource("seckill.lua"));
        //设置返回类型
        SECKILL_SCRIPT.setResultType(Long.class);
    }

    @PostConstruct
    private void init()
    {
        SECKILL_ORDER_EXECUTOR.submit((Runnable) VoucherOrderHandler::new);
    }

    private class VoucherOrderHandler implements Runnable
    {
        @Override
        public void run()
        {
            //系统启动开始，便不断从阻塞队列中获取优惠券订单信息
            while (true)
            {
                try
```

```
                {
                    //获取订单信息
                    VoucherOrder voucherOrder = blockingQueue.take();
                    //创建订单
                    createVoucherOrder(voucherOrder);
                    log.debug("异步订单创建任务执行成功！订单id：" +
voucherOrder.getVoucherId());
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        }
    }


//    @Override
//    public Result seckillVoucher(Long voucherId)
//    {
//        //查询优惠券
//        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
//        //判断是否存在
//        if (seckillVoucher == null)
//        {
//            return Result.fail("活动不存在");
//        }
//        //判断是否开始
//        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
//        {
//            //未开始
//            return Result.fail("秒杀活动未开始");
//        }
//        //判断是否结束
//        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
//        {
//            //结束
//            return Result.fail("秒杀活动已经结束");
//        }
//        //判断库存是否充足
//        if (seckillVoucher.getStock() <= 0)
//        {
//            //库存不足
//            return Result.fail("库存不足");
//        }
//        //创建订单
//        return this.createVoucherOrder(voucherId);
//    }
//
//
//    /**
//     * 创建订单
//     *
//     * @param voucherId voucherId
//     * @return Result
//     */
//    @Transactional
```

```
//    public Result createVoucherOrder(Long voucherId)
//    {
//        //判断当前优惠券用户是否已经下过单
//        //获得用户id
//        //Long userID = UserHolder.getUser().getId();
//        Long userID = 5L;
//        //todo:记得更改回来
//        /*//创建锁对象
//        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
//        //取得锁
//        boolean isLock = redisLock.tryLock(1500);
//        //判断
//        if (!isLock)
//        {
//            return Result.fail("不允许重复下单! ");
//        }*/
//
//
//        //redisson锁
//        RLock lock = redissonClient.getLock("lock:order:" + userID);
//        //尝试获取锁
//        //仅当调用时它是空闲的时才获取锁。
//        //如果锁可用，则获取锁并立即返回值为true
//        //如果锁不可用，则此方法将立即返回值false
//        boolean tryLock = lock.tryLock();
//        //判断是否获取成功
//        if (!tryLock)
//        {
//            return Result.fail("不允许重复下单! ");
//        }
//        //获取锁成功
//        //synchronized (userID.toString().intern())
//        try
//        {
//            //查询数据库
//            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
//            //判断长度
//            if (count > 0)
//            {
//                //长度大于0，用户购买过
//                return Result.fail("不能重复下单");
//            }
//            //扣减库存
//            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>
();
//            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
//            boolean update = seckillVoucherService.update(updateWrapper);
//            if (!update)
//            {
//                //失败
//                return Result.fail("库存扣减失败");
//            }
//            //扣减成功
//            //生成订单
//            VoucherOrder voucherOrder = new VoucherOrder();
```

```
//             //生成id
//             Long orderID = redisIDGenerator.nextID("order");
//             voucherOrder.setVoucherId(voucherId);
//             voucherOrder.setId(orderID);
//             //设置用户
//             //Long userID = UserHolder.getUser().getId();
//             voucherOrder.setUserId(userID);
//             //保存订单
//             this.save(voucherOrder);
//             //返回
//             return Result.ok(orderID);
//         }
//         finally
//         {
//             //释放锁
//             //redisLock.unlock();
//             lock.unlock();
//         }
//     }


    @Override
    public Result seckillVoucher(Long voucherId)
    {
        String s =
stringRedisTemplate.opsForValue().get(RedisConstants.SECKILL_STOCK_KEY +
voucherId);
        //判断此优惠券是否存在
        if (s == null)
        {
            return Result.fail("此优惠券不存在");
        }
        UserDTO user = UserHolder.getUser();
        //获得用户ID
        //Long userID = user.getId();
        Long userID = 5L;
        //todo:记得更改
        //执行lua脚本
        Long result = stringRedisTemplate.execute(SECKILL_SCRIPT,
                Collections.emptyList(),
                voucherId.toString(), userID.toString());
        if (result == null)
        {
            return Result.fail("订单异常");
        }
        // 判断结果是否为 0
        if (result != 0)
        {
            // 不为 0 ，代表没有购买资格
            Result.fail(result == 1 ? "库存不足！" : "不能重复下单!");
        }
        //为0，创建订单
        Long orderId = redisIDGenerator.nextID("oder");
        VoucherOrder voucherOrder = new VoucherOrder();
        voucherOrder.setVoucherId(voucherId);
        voucherOrder.setUserId(userID);
        voucherOrder.setId(orderId);
        //加入到阻塞队列
```

```
        blockingQueue.add(voucherOrder);
        //返回订单
        return Result.ok(voucherOrder);
    }

    /**
     * 创建订单
     *
     * @param voucherOrder VoucherOrder
     */
    @Transactional
    protected void createVoucherOrder(VoucherOrder voucherOrder)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        /*//创建锁对象
        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
        //取得锁
        boolean isLock = redisLock.tryLock(1500);
        //判断
        if (!isLock)
        {
            return Result.fail("不允许重复下单！");
        }*/


        //redisson锁
        RLock lock = redissonClient.getLock("lock:order:" + userID);
        //尝试获取锁
        //仅当调用时它是空闲的时才获取锁。
        //如果锁可用，则获取锁并立即返回值为true
        //如果锁不可用，则此方法将立即返回值false
        boolean tryLock = lock.tryLock();
        //判断是否获取成功
        if (!tryLock)
        {
            return;
        }
        //获取锁成功
        //synchronized (userID.toString().intern())
        Long voucherId = voucherOrder.getVoucherId();
        try
        {
            //查询数据库
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return;
            }
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
```

```
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return;
            }
            //扣减成功
            //保存订单
            this.save(voucherOrder);
        }
        finally
        {
            //释放锁
            //redisLock.unlock();
            lock.unlock();
        }
    }
}
```

lua脚本:

```
-- 优惠券id
local voucherId = ARGV[1]
-- 用户id
local userId = ARGV[2]

-- 库存key
local stockKey = "seckill:stock:"..voucherId
-- 订单key
local orderKey = "seckill:order:"..voucherId

-- 判断库存是否充足
if(tonumber(redis.call('get', stockKey)) <= 0) then
    return 1
end

-- 判断用户是否已经下过单
if(redis.call('sismember', orderKey, userId) == 1) then
    return 2
end

-- 扣减库存
redis.call('incrby', stockKey, -1)

-- 将 userId 存入当前优惠券的 set 集合
redis.call('sadd', orderKey, userId)

return 0
```

思路

- 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

存在问题

- 内存限制问题
- 数据安全问题。我们是基于内存保存的订单信息，如果服务突然宕机，那么内存中的订单信息也就丢失了

消息队列实现秒杀

- 消息队列是在 JVM 以外的独立服务，所以不受 JVM 内存的限制
- 消息队列不仅仅做数据存储，还需要确保数据安全，存入到消息队列中的所有消息都需要做持久化，这样不管是服务宕机还是重启，数据都不会丢失。而且消息队列还会在消息投递给消费者后，要求消费者做消息确认，如果消费者没有确认，那么这条消息就会一直存在于消息队列中，下一次会继续投递给消费者，让消费者继续处理，直到消息被成功处理。

Redis 提供的消息队列

- list 结构：基于 List 结构模拟消息队列
- PubSub：基本的点对点消息队列
- Stream：比较完善的消息队列模型
-

list 结构消息队列

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。假设某个消费者从消息队列（List 结构）中获取到一条消息，但还未来得及处理，该消费者出现故障，那么这条消息就会丢失，这是因为 POP 命令是 remove and get，会将消息直接从消息队列中直接移除，这样其他消费者就获取不到。
- 只支持单消费者。消息队列（List 结构）中的消息，一旦被某个消费者取走，就会从队列中移除，其他消费者就获取不到了，无法实现一条消息被很多消费者消费的需求。

PubSub消息队列

优点：

- 采用发布订阅模型，支持多生产、多消费。一条消息可以发给多个消费者，也可以发给一个消费者，而且也支持不同生产者往相同频道发。

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。当发送一条消息时，如果有消费者监听，消费者会将发送过来的消息缓存至消息缓存区，由消费者进行处理。而消费者的缓存空间是有上限的，如果超出了就会丢失。

Stream 消息队列

优点：

- 消息可回溯。消息读完后不消失，永久保存在队列中
- 一个消息可以被多个消费者读取
- 可以阻塞读取

rabbitMQ实现

配置：

```
package mao.spring_boot_redis_hmdp.config;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Project name(项目名称)：
spring_boot_redis_hmdp_message_queue_realizes_asynchronous_spike
 * Package(包名): mao.spring_boot_redis_hmdp.config
 * Class(类名): RabbitMQConfig
 * Author(作者）: mao
 * Author QQ：1296193245
 * GitHub：https://github.com/maomao124/
 * Date(创建日期)： 2022/5/17
 * Time(创建时间)： 22:23
 * Version(版本): 1.0
 * Description(描述)： RabbitMQ基本配置，暂时不考虑消息丢失问题
 */

@Configuration
public class RabbitMQConfig
```

```
{
    /**
     * 交换机名称
     */
    public static final String EXCHANGE_NAME = "hmdp_exchange";
    /**
     * 队列名称
     */
    public static final String QUEUE_NAME = "hmdp_queue";
    /**
     * routingKey
     */
    public static final String ROUTING_KEY = "VoucherOrder";

    /**
     * 声明直接交换机
     *
     * @return DirectExchange
     */
    @Bean
    public DirectExchange directExchange()
    {
        return new DirectExchange(EXCHANGE_NAME, false, false, null);
    }

    /**
     * 声明队列
     *
     * @return Queue
     */
    @Bean
    public Queue queue()
    {
        return new Queue(QUEUE_NAME, false, false, false, null);
    }

    /**
     * 绑定
     *
     * @return Binding
     */
    @Bean
    public Binding exchange_binding_queue()
    {
        return
BindingBuilder.bind(queue()).to(directExchange()).with(ROUTING_KEY);
    }

    /**
     * 使用json传递消息
     *
     * @return Jackson2JsonMessageConverter
     */
    @Bean
    public Jackson2JsonMessageConverter jackson2JsonMessageConverter()
    {
        return new Jackson2JsonMessageConverter();
    }
```

```
}
```

业务：

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.config.RabbitMQConfig;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.SeckillVoucher;
import mao.spring_boot_redis_hmdp.entity.VoucherOrder;
import mao.spring_boot_redis_hmdp.mapper.VoucherOrderMapper;
import mao.spring_boot_redis_hmdp.service.ISeckillVoucherService;
import mao.spring_boot_redis_hmdp.service.IVoucherOrderService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisIDGenerator;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


@Service
public class VoucherOrderServiceImpl extends ServiceImpl<VoucherOrderMapper,
VoucherOrder> implements IVoucherOrderService
{
    @Resource
    private ISeckillVoucherService seckillVoucherService;

    @Resource
    private RedisIDGenerator redisIDGenerator;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Resource
```

```
    private RedissonClient redissonClient;

    @Resource
    private RabbitTemplate rabbitTemplate;

    private static final DefaultRedisScript<Long> SECKILL_SCRIPT;

    /**
     * 阻塞队列
     */
    private final BlockingQueue<VoucherOrder> blockingQueue = new
ArrayBlockingQueue<>(1024 * 1024);

    /**
     * 线程池
     */
    private static final ExecutorService SECKILL_ORDER_EXECUTOR =
Executors.newSingleThreadExecutor();

    static
    {
        //创建对象
        SECKILL_SCRIPT = new DefaultRedisScript<>();
        //加载
        SECKILL_SCRIPT.setLocation(new ClassPathResource("seckill.lua"));
        //设置返回类型
        SECKILL_SCRIPT.setResultType(Long.class);
    }

    @PostConstruct
    private void init()
    {
        SECKILL_ORDER_EXECUTOR.submit((Runnable) VoucherOrderHandler::new);
    }

    private class VoucherOrderHandler implements Runnable
    {
        @Override
        public void run()
        {
            //系统启动开始，便不断从阻塞队列中获取优惠券订单信息
            while (true)
            {
                try
                {
                    //获取订单信息
                    VoucherOrder voucherOrder = blockingQueue.take();
                    //创建订单
                    createVoucherOrder(voucherOrder);
                    log.debug("异步订单创建任务执行成功！订单id：" +
voucherOrder.getVoucherId());
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        }
```

```
    }


//    @Override
//    public Result seckillVoucher(Long voucherId)
//    {
//        //查询优惠券
//        SeckillVoucher seckillVoucher =
seckillVoucherService.getById(voucherId);
//        //判断是否存在
//        if (seckillVoucher == null)
//        {
//            return Result.fail("活动不存在");
//        }
//        //判断是否开始
//        if (seckillVoucher.getBeginTime().isAfter(LocalDateTime.now()))
//        {
//            //未开始
//            return Result.fail("秒杀活动未开始");
//        }
//        //判断是否结束
//        if (seckillVoucher.getEndTime().isBefore(LocalDateTime.now()))
//        {
//            //结束
//            return Result.fail("秒杀活动已经结束");
//        }
//        //判断库存是否充足
//        if (seckillVoucher.getStock() <= 0)
//        {
//            //库存不足
//            return Result.fail("库存不足");
//        }
//        //创建订单
//        return this.createVoucherOrder(voucherId);
//    }
//
//
//    /**
//     * 创建订单
//     *
//     * @param voucherId voucherId
//     * @return Result
//     */
//    @Transactional
//    public Result createVoucherOrder(Long voucherId)
//    {
//        //判断当前优惠券用户是否已经下过单
//        //获得用户id
//        //Long userID = UserHolder.getUser().getId();
//        Long userID = 5L;
//        //todo:记得更改回来
//        /*//创建锁对象
//        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
//        //取得锁
//        boolean isLock = redisLock.tryLock(1500);
//        //判断
//        if (!isLock)
```

```
//         {
//             return Result.fail("不允许重复下单！");
//         }*/
//
//
//         //redisson锁
//         RLock lock = redissonClient.getLock("lock:order:" + userID);
//         //尝试获取锁
//         //仅当调用时它是空闲的时才获取锁。
//         //如果锁可用，则获取锁并立即返回值为true
//         //如果锁不可用，则此方法将立即返回值false
//         boolean tryLock = lock.tryLock();
//         //判断是否获取成功
//         if (!tryLock)
//         {
//             return Result.fail("不允许重复下单！");
//         }
//         //获取锁成功
//         //synchronized (userID.toString().intern())
//         try
//         {
//             //查询数据库
//             Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
//             //判断长度
//             if (count > 0)
//             {
//                 //长度大于0，用户购买过
//                 return Result.fail("不能重复下单");
//             }
//             //扣减库存
//             UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>
();
//             updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
//             boolean update = seckillVoucherService.update(updateWrapper);
//             if (!update)
//             {
//                 //失败
//                 return Result.fail("库存扣减失败");
//             }
//             //扣减成功
//             //生成订单
//             VoucherOrder voucherOrder = new VoucherOrder();
//             //生成id
//             Long orderID = redisIDGenerator.nextID("order");
//             voucherOrder.setVoucherId(voucherId);
//             voucherOrder.setId(orderID);
//             //设置用户
//             //Long userID = UserHolder.getUser().getId();
//             voucherOrder.setUserId(userID);
//             //保存订单
//             this.save(voucherOrder);
//             //返回
//             return Result.ok(orderID);
//         }
//         finally
//         {
```

```
//             //释放锁
//             //redisLock.unlock();
//             lock.unlock();
//         }
//     }


    /**
     * 处理消息的方法，创建订单
     *
     * @param voucherOrder voucherOrder
     */
    @RabbitListener(queues = {RabbitMQConfig.QUEUE_NAME})
    public void handleMessage(VoucherOrder voucherOrder)
    {
        //创建订单
        createVoucherOrder(voucherOrder);
        log.debug("异步订单创建任务执行成功！订单id：" + voucherOrder.getVoucherId());
    }

    @Override
    public Result seckillVoucher(Long voucherId)
    {
        String s =
stringRedisTemplate.opsForValue().get(RedisConstants.SECKILL_STOCK_KEY +
voucherId);
        //判断此优惠券是否存在
        if (s == null)
        {
            return Result.fail("此优惠券不存在");
        }
        UserDTO user = UserHolder.getUser();
        //获得用户ID
        //Long userID = user.getId();
        Long userID = 5L;
        //todo:记得更改
        //执行lua脚本
        Long result = stringRedisTemplate.execute(SECKILL_SCRIPT,
                Collections.emptyList(),
                voucherId.toString(), userID.toString());
        if (result == null)
        {
            return Result.fail("订单异常");
        }
        // 判断结果是否为 0
        if (result != 0)
        {
            // 不为 0 ，代表没有购买资格
            Result.fail(result == 1 ? "库存不足！" : "不能重复下单!");
        }
        //为0，创建订单
        Long orderId = redisIDGenerator.nextID("oder");
        VoucherOrder voucherOrder = new VoucherOrder();
        voucherOrder.setVoucherId(voucherId);
        voucherOrder.setUserId(userID);
        voucherOrder.setId(orderId);
        //加入到阻塞队列
        //blockingQueue.add(voucherOrder);
```

```
        //加入到消息队列
        rabbitTemplate.convertAndSend(RabbitMQConfig.EXCHANGE_NAME,
RabbitMQConfig.ROUTING_KEY, voucherOrder);
        //返回订单
        return Result.ok(voucherOrder);
    }

    /**
     * 创建订单
     *
     * @param voucherOrder VoucherOrder
     */
    @Transactional
    protected void createVoucherOrder(VoucherOrder voucherOrder)
    {
        //判断当前优惠券用户是否已经下过单
        //获得用户id
        //Long userID = UserHolder.getUser().getId();
        Long userID = 5L;
        //todo:记得更改回来
        /*//创建锁对象
        RedisLock redisLock = new RedisLockImpl("order:" + userID,
stringRedisTemplate);
        //取得锁
        boolean isLock = redisLock.tryLock(1500);
        //判断
        if (!isLock)
        {
            return Result.fail("不允许重复下单！");
        }*/


        //redisson锁
        RLock lock = redissonClient.getLock("lock:order:" + userID);
        //尝试获取锁
        //仅当调用时它是空闲的时才获取锁。
        //如果锁可用，则获取锁并立即返回值为true
        //如果锁不可用，则此方法将立即返回值false
        boolean tryLock = lock.tryLock();
        //判断是否获取成功
        if (!tryLock)
        {
            return;
        }
        //获取锁成功
        //synchronized (userID.toString().intern())
        Long voucherId = voucherOrder.getVoucherId();
        try
        {
            //查询数据库
            Long count = this.query().eq("user_id", userID).eq("voucher_id",
voucherId).count();
            //判断长度
            if (count > 0)
            {
                //长度大于0，用户购买过
                return;
            }
```

```
            //扣减库存
            UpdateWrapper<SeckillVoucher> updateWrapper = new UpdateWrapper<>();
            updateWrapper.setSql("stock = stock - 1").eq("voucher_id",
voucherId).gt("stock", 0);
            boolean update = seckillVoucherService.update(updateWrapper);
            if (!update)
            {
                //失败
                return;
            }
            //扣减成功
            //保存订单
            this.save(voucherOrder);
        }
        finally
        {
            //释放锁
            //redisLock.unlock();
            lock.unlock();
        }
    }
}
```

达人探店

查询探店笔记

controller:

```
    @GetMapping("/hot")
    public Result queryHotBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        return blogService.queryHotBlog(current);
    }

    @GetMapping("/{id}")
    public Result queryBlogById(@PathVariable("id") String id)
    {
        return blogService.queryBlogById(id);
    }
```

接口:

```
package mao.spring_boot_redis_hmdp.service;
```

```
import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;


public interface IBlogService extends IService<Blog>
{
    /**
     * 查询热门的探店笔记
     *
     * @param current 当前页
     * @return Result
     */
    Result queryHotBlog(Integer current);

    /**
     * 根据id进行查询
     *
     * @param id id
     * @return Result
     */
    Result queryBlogById(String id);
}
```

实现：

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.BlogMapper;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.List;
import java.util.concurrent.TimeUnit;


@Service("blogService")
public class BlogServiceImpl extends ServiceImpl<BlogMapper, Blog> implements
IBlogService
{

    @Resource
    private IUserService userService;
```

```
    @Resource
    private RedisUtils redisUtils;

    @Override
    public Result queryHotBlog(Integer current)
    {
        // 根据用户查询
        Page<Blog> page = query()
                .orderByDesc("liked")
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        // 查询用户
        records.forEach(blog ->
        {
            Long userId = blog.getUserId();
            User user = userService.getById(userId);
            blog.setName(user.getNickName());
            blog.setIcon(user.getIcon());
        });
        return Result.ok(records);
    }

    @Override
    public Result queryBlogById(String id)
    {
        //查询
        //Blog blog = this.getById(id);
        Blog blog = redisUtils.query(RedisConstants.BLOG_KEY,
                RedisConstants.LOCK_BLOG_KEY, id,
                Blog.class, this::getById,
                RedisConstants.CACHE_BLOG_TTL,
                TimeUnit.MINUTES, 120);
        //判断是否存在
        if (blog == null)
        {
            //不存在，返回
            return Result.fail("该笔记信息不存在");
        }
        //存在
        //填充用户信息
        //获得用户id
        Long userId = blog.getUserId();
        //查询
        User user = userService.getById(userId);
        //填充
        blog.setIcon(user.getIcon());
        blog.setName(user.getNickName());
        //返回
        return Result.ok(blog);
    }
}
```

实现点赞功能

controller：

```
    @PutMapping("/like/{id}")
    public Result likeBlog(@PathVariable("id") Long id)
    {
        return blogService.likeBlog(id);
    }
```

接口：

```
package mao.spring_boot_redis_hmdp.service;


import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;


public interface IBlogService extends IService<Blog>
{
    /**
     * 查询热门的探店笔记
     *
     * @param current 当前页
     * @return Result
     */
    Result queryHotBlog(Integer current);

    /**
     * 根据id进行查询
     *
     * @param id id
     * @return Result
     */
    Result queryBlogById(String id);

    /**
     * 点赞功能
     * @param id id
     * @return result
     */
    Result likeBlog(Long id);
}
```

实现：

```java
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.util.BooleanUtil;
import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.BlogMapper;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.List;
import java.util.concurrent.TimeUnit;


@Service("blogService")
public class BlogServiceImpl extends ServiceImpl<BlogMapper, Blog> implements
IBlogService
{

    @Resource
    private IUserService userService;

    @Resource
    private RedisUtils redisUtils;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result queryHotBlog(Integer current)
    {
        // 根据用户查询
        Page<Blog> page = query()
                .orderByDesc("liked")
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        // 查询用户
        records.forEach(blog ->
        {
            Long userId = blog.getUserId();
            User user = userService.getById(userId);
            blog.setName(user.getNickName());
            blog.setIcon(user.getIcon());
        });
        return Result.ok(records);
```

```
    }

    @Override
    public Result queryBlogById(String id)
    {
        //查询
        //Blog blog = this.getById(id);
        Blog blog = redisUtils.query(RedisConstants.BLOG_KEY,
                RedisConstants.LOCK_BLOG_KEY, id,
                Blog.class, this::getById,
                RedisConstants.CACHE_BLOG_TTL,
                TimeUnit.MINUTES, 120);
        //判断是否存在
        if (blog == null)
        {
            //不存在，返回
            return Result.fail("该笔记信息不存在");
        }
        //存在
        //填充用户信息
        //获得用户id
        Long userId = blog.getUserId();
        //查询
        User user = userService.getById(userId);
        //填充
        blog.setIcon(user.getIcon());
        blog.setName(user.getNickName());
        //返回
        return Result.ok(blog);
    }

    @Override
    public Result likeBlog(Long id)
    {
        //获取用户信息
        UserDTO user = UserHolder.getUser();
        //判断用户是否已经点赞(检查设置在key是否包含value)
        Boolean member =
stringRedisTemplate.opsForSet().isMember(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
        if (BooleanUtil.isFalse(member))
        {
            //未点赞
            //数据库点赞数量+1
            boolean update = update().setSql("liked = liked + 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //保存用户到Redis的set集合

 stringRedisTemplate.opsForSet().add(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
            }
```

```
        }
        else
        {
            //已点赞，取消点赞
            //数据库点赞数量-1
            boolean update = update().setSql("liked = liked - 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //移除用户
                stringRedisTemplate.delete(RedisConstants.BLOG_LIKED_KEY + id);
            }
        }

        return Result.ok();
    }
}
```

实现点赞排行榜

controller：

```
    @GetMapping("/likes/{id}")
    public Result queryBlogLikes(@PathVariable("id") String id)
    {
        return blogService.queryBlogLikes(id);
    }
```

接口：

```
package mao.spring_boot_redis_hmdp.service;


import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;


public interface IBlogService extends IService<Blog>
{
    /**
     * 查询热门的探店笔记
     *
     * @param current 当前页
     * @return Result
     */
    Result queryHotBlog(Integer current);
```

```
    /**
     * 根据id进行查询
     *
     * @param id id
     * @return Result
     */
    Result queryBlogById(String id);

    /**
     * 点赞功能
     *
     * @param id id
     * @return result
     */
    Result likeBlog(Long id);

    /**
     * 点赞排行榜
     *
     * @param id id
     * @return Result
     */
    Result queryBlogLikes(String id);
}
```

实现类:

```
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.bean.BeanUtil;
import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.BlogMapper;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.Collections;
import java.util.List;
```

```java
import java.util.Set;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;


@Service("blogService")
public class BlogServiceImpl extends ServiceImpl<BlogMapper, Blog> implements
IBlogService
{

    @Resource
    private IUserService userService;

    @Resource
    private RedisUtils redisUtils;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result queryHotBlog(Integer current)
    {
        // 根据用户查询
        Page<Blog> page = query()
                .orderByDesc("liked")
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        // 查询用户
        records.forEach(blog ->
        {
            Long userId = blog.getUserId();
            User user = userService.getById(userId);
            blog.setName(user.getNickName());
            blog.setIcon(user.getIcon());
        });
        return Result.ok(records);
    }

    @Override
    public Result queryBlogById(String id)
    {
        //查询
        //Blog blog = this.getById(id);
        Blog blog = redisUtils.query(RedisConstants.BLOG_KEY,
                RedisConstants.LOCK_BLOG_KEY, id,
                Blog.class, this::getById,
                RedisConstants.CACHE_BLOG_TTL,
                TimeUnit.MINUTES, 120);
        //判断是否存在
        if (blog == null)
        {
            //不存在，返回
            return Result.fail("该笔记信息不存在");
        }
        //存在
        //填充用户信息
        //获得用户id
```

```
        Long userId = blog.getUserId();
        //查询
        User user = userService.getById(userId);
        //填充
        blog.setIcon(user.getIcon());
        blog.setName(user.getNickName());
        //返回
        return Result.ok(blog);
    }

    @Override
    public Result likeBlog(Long id)
    {
        //获取用户信息
        UserDTO user = UserHolder.getUser();
        //判断用户是否已经点赞(检查设置在key是否包含value)
        Boolean member =
stringRedisTemplate.opsForSet().isMember(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
        if (BooleanUtil.isFalse(member))
        {
            //未点赞
            //数据库点赞数量+1
            boolean update = update().setSql("liked = liked + 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //保存用户到Redis的set集合

 stringRedisTemplate.opsForSet().add(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
            }

        }
        else
        {
            //已点赞，取消点赞
            //数据库点赞数量-1
            boolean update = update().setSql("liked = liked - 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //移除用户
                stringRedisTemplate.delete(RedisConstants.BLOG_LIKED_KEY + id);
            }
        }

        return Result.ok();
    }
```

```
    @Override
    public Result queryBlogLikes(String id)
    {
        //获得key
        String redisKey = RedisConstants.BLOG_LIKED_KEY + id;
        //查询前5名的点赞的用户(从排序集中获取start和end之间的元素)
        Set<String> range = stringRedisTemplate.opsForZSet().range(redisKey, 0,
4);
        //判断
        if (range == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //非空
        //解析出用户的id
        List<Long> ids =
range.stream().map(Long::valueOf).collect(Collectors.toList());
        //拼接
        String join = StrUtil.join(",", ids);
        //查询数据库
        List<User> users = userService.query().in("id", ids).last("order by
filed(id, " + join + ")").list();
        //转换成dto
        List<UserDTO> dtoList = users.stream().map(user ->
BeanUtil.copyProperties(user, UserDTO.class)).
                collect(Collectors.toList());
        //返回数据
        return Result.ok(dtoList);
    }
}
```

关注

关注和取消关注

controller:

```
package mao.spring_boot_redis_hmdp.controller;


import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.service.IFollowService;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;
```

```
@RestController
@RequestMapping("/follow")
public class FollowController
{

    @Resource
    private IFollowService followService;

    @PutMapping("/{id}/{isFollow}")
    public Result follow(@PathVariable("id") Long followUserId,
@PathVariable("isFollow") Boolean isFollow)
    {
        return followService.follow(followUserId, isFollow);
    }

    @PutMapping("/or/not/{id}")
    public Result isFollow(@PathVariable("id") Long followUserId)
    {
        return followService.isFollow(followUserId);
    }

}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;


import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Follow;


public interface IFollowService extends IService<Follow>
{

    /**
     * 关注或者取消关注，这取决于isFollow的值
     *
     * @param followUserId 被关注的用户的id
     * @param isFollow     如果是关注，则为true，否则为false
     * @return Result
     */
    Result follow(Long followUserId, Boolean isFollow);

    /**
     * 判断当前用户是否关注了 用户id为followUserId的人
     *
     * @param followUserId 被关注的用户的id
     * @return Result
     */
    Result isFollow(Long followUserId);
}
```

实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;


import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Follow;
import mao.spring_boot_redis_hmdp.mapper.FollowMapper;
import mao.spring_boot_redis_hmdp.service.IFollowService;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.stereotype.Service;


@Service
public class FollowServiceImpl extends ServiceImpl<FollowMapper, Follow>
implements IFollowService
{

    @Override
    public Result follow(Long followUserId, Boolean isFollow)
    {
        //空值判断
        if (followUserId == null)
        {
            return Result.fail("关注的用户id不能为空");
        }
        if (isFollow == null)
        {
            return Result.fail("参数异常");
        }
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //判断是关注还是取消关注
        if (isFollow)
        {
            //是关注
            //加关注
            Follow follow = new Follow();
            //设置关注的用户id
            follow.setFollowUserId(followUserId);
            //设置当前用户的id
            follow.setUserId(userID);
            //保存到数据库
            boolean save = this.save(follow);
            //判断是否关注失败
            if (!save)
            {
                return Result.fail("关注失败");
            }
        }
        else
```

```
        {
            //不是关注，取消关注
            //删除数据库里的相关信息
            QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
            queryWrapper.eq("follow_user_id", followUserId).eq("user_id",
userID);
            //删除
            boolean remove = this.remove(queryWrapper);
            if (!remove)
            {
                return Result.fail("取消关注失败");
            }
        }
        //返回
        return Result.ok();

    }

    @Override
    public Result isFollow(Long followUserId)
    {
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //查数据库
        QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("follow_user_id", followUserId).eq("user_id", userID);
        long count = this.count(queryWrapper);
        //返回
        return Result.ok(count > 0);
    }
}
```

共同关注

修改userController：

```
package mao.spring_boot_redis_hmdp.controller;


import cn.hutool.core.bean.BeanUtil;
import lombok.extern.slf4j.Slf4j;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.entity.UserInfo;
import mao.spring_boot_redis_hmdp.service.IUserInfoService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import javax.servlet.http.HttpSession;
```

```
@Slf4j
@RestController
@RequestMapping("/user")
public class UserController
{

    @Resource
    private IUserService userService;

    @Resource
    private IUserInfoService userInfoService;

    /**
     * 发送手机验证码
     */
    @PostMapping("code")
    public Result sendCode(@RequestParam("phone") String phone, HttpSession
session)
    {
        return userService.sendCode(phone, session);
    }

    /**
     * 登录功能
     *
     * @param loginForm 登录参数，包含手机号、验证码；或者手机号、密码
     */
    @PostMapping("/login")
    public Result login(@RequestBody LoginFormDTO loginForm, HttpSession
session)
    {
        return userService.login(loginForm, session);
    }

    /**
     * 登出功能
     *
     * @return 无
     */
    @PostMapping("/logout")
    public Result logout()
    {
        // TODO 实现登出功能
        return Result.fail("功能未完成");
    }

    @GetMapping("/me")
    public Result me()
    {
        return Result.ok(UserHolder.getUser());
    }

    @GetMapping("/info/{id}")
    public Result info(@PathVariable("id") Long userId)
    {
        // 查询详情
        UserInfo info = userInfoService.getById(userId);
```

```
        if (info == null)
        {
            // 没有详情，应该是第一次查看详情
            return Result.ok();
        }
        info.setCreateTime(null);
        info.setUpdateTime(null);
        // 返回
        return Result.ok(info);
    }

    /**
     * 根据查询用户信息
     *
     * @param userId 用户的id
     * @return Result
     */
    @GetMapping("/{id}")
    public Result queryUserById(@PathVariable("id") Long userId)
    {
        //查询用户信息
        User user = userService.getById(userId);
        if (user == null)
        {
            return Result.ok();
        }
        //转换
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        return Result.ok(userDTO);
    }
}
```

修改BlogController：

```
package mao.spring_boot_redis_hmdp.controller;


import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import java.util.List;


@RestController
@RequestMapping("/blog")
public class BlogController
{

    @Resource
```

```
    private IBlogService blogService;


    @PostMapping
    public Result saveBlog(@RequestBody Blog blog)
    {
        // 获取登录用户
        UserDTO user = UserHolder.getUser();
        blog.setUserId(user.getId());
        // 保存探店博文
        blogService.save(blog);
        // 返回id
        return Result.ok(blog.getId());
    }

    @PutMapping("/like/{id}")
    public Result likeBlog(@PathVariable("id") Long id)
    {
        return blogService.likeBlog(id);
    }

    @GetMapping("/of/me")
    public Result queryMyBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        // 获取登录用户
        UserDTO user = UserHolder.getUser();
        // 根据用户查询
        Page<Blog> page = blogService.query()
                .eq("user_id", user.getId()).page(new Page<>(current,
SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        return Result.ok(records);
    }

    @GetMapping("/hot")
    public Result queryHotBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        return blogService.queryHotBlog(current);
    }

    @GetMapping("/{id}")
    public Result queryBlogById(@PathVariable("id") String id)
    {
        return blogService.queryBlogById(id);
    }

    @GetMapping("/likes/{id}")
    public Result queryBlogLikes(@PathVariable("id") String id)
    {
        return blogService.queryBlogLikes(id);
    }

    /**
     * 查询用户的笔记信息
     *
```

```
     * @param current 当前页，如果不指定，则为第一页
     * @param id      博主的id
     * @return Result
     */
    @GetMapping("/of/user")
    public Result queryBlogByUserId(@RequestParam(value = "current",
defaultValue = "1") Integer current,
                                    @RequestParam("id") Long id)
    {
        //根据用户查询
        Page<Blog> page = blogService.query().
                eq("user_id", id).
                page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        //获取当前页数据
        List<Blog> records = page.getRecords();
        //返回
        return Result.ok(records);
    }


}
```

修改接口：

```
package mao.spring_boot_redis_hmdp.service;


import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Follow;


public interface IFollowService extends IService<Follow>
{

    /**
     * 关注或者取消关注，这取决于isFollow的值
     *
     * @param followUserId 被关注的用户的id
     * @param isFollow     如果是关注，则为true，否则为false
     * @return Result
     */
    Result follow(Long followUserId, Boolean isFollow);

    /**
     * 判断当前用户是否关注了 用户id为followUserId的人
     *
     * @param followUserId 被关注的用户的id
     * @return Result
     */
    Result isFollow(Long followUserId);

    /**
     * 获取共同关注的人
```

```
     *
     * @param id 博主的id
     * @return Result
     */
    Result followCommons(Long id);
}
```

修改实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.bean.BeanUtil;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Follow;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.FollowMapper;
import mao.spring_boot_redis_hmdp.service.IFollowService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.redisson.mapreduce.Collector;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;


@Service
public class FollowServiceImpl extends ServiceImpl<FollowMapper, Follow>
implements IFollowService
{

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Resource
    private IUserService userService;

/*    @Override
    public Result follow(Long followUserId, Boolean isFollow)
    {
        //空值判断
        if (followUserId == null)
        {
            return Result.fail("关注的用户id不能为空");
        }
```

```
        if (isFollow == null)
        {
            return Result.fail("参数异常");
        }
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //判断是关注还是取消关注
        if (isFollow)
        {
            //是关注
            //加关注
            Follow follow = new Follow();
            //设置关注的用户id
            follow.setFollowUserId(followUserId);
            //设置当前用户的id
            follow.setUserId(userID);
            //保存到数据库
            boolean save = this.save(follow);
            //判断是否关注失败
            if (!save)
            {
                return Result.fail("关注失败");
            }
        }
        else
        {
            //不是关注，取消关注
            //删除数据库里的相关信息
            QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
            queryWrapper.eq("follow_user_id", followUserId).eq("user_id",
userID);
            //删除
            boolean remove = this.remove(queryWrapper);
            if (!remove)
            {
                return Result.fail("取消关注失败");
            }
        }
        //返回
        return Result.ok();
    }*/

    @Override
    public Result follow(Long followUserId, Boolean isFollow)
    {
        //空值判断
        if (followUserId == null)
        {
            return Result.fail("关注的用户id不能为空");
        }
        if (isFollow == null)
        {
            return Result.fail("参数异常");
        }
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //判断是关注还是取消关注
        if (isFollow)
```

```
        {
            //是关注
            //加关注
            Follow follow = new Follow();
            //设置关注的用户id
            follow.setFollowUserId(followUserId);
            //设置当前用户的id
            follow.setUserId(userID);
            //保存到数据库
            boolean save = this.save(follow);
            //判断是否关注失败
            if (!save)
            {
                return Result.fail("关注失败");
            }
            //关注成功，保存到redis
            //登录用户的key
            String redisUserKey = RedisConstants.FOLLOW_KEY + userID;
            //保存
            stringRedisTemplate.opsForSet().add(redisUserKey,
followUserId.toString());
        }
        else
        {
            //不是关注，取消关注
            //删除数据库里的相关信息
            QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
            queryWrapper.eq("follow_user_id", followUserId).eq("user_id",
userID);
            //删除
            boolean remove = this.remove(queryWrapper);
            if (!remove)
            {
                return Result.fail("取消关注失败");
            }
            //删除成功，移除redis相关数据
            //登录用户的key
            String redisUserKey = RedisConstants.FOLLOW_KEY + userID;
            //移除
            stringRedisTemplate.opsForSet().remove(redisUserKey, followUserId);
        }
        //返回
        return Result.ok();
    }

    @Override
    public Result isFollow(Long followUserId)
    {
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //查数据库
        QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("follow_user_id", followUserId).eq("user_id", userID);
        long count = this.count(queryWrapper);
        //返回
        return Result.ok(count > 0);
    }
```

```
    @Override
    public Result followCommons(Long id)
    {
        //获取当前用户的id
        Long userID = UserHolder.getUser().getId();
        //获得redisKey
        //登录用户的key
        String redisUserKey = RedisConstants.FOLLOW_KEY + userID;
        //博主的key
        String redisBlogKey = RedisConstants.FOLLOW_KEY + id;
        //获得交集
        Set<String> intersect =
stringRedisTemplate.opsForSet().intersect(redisUserKey, redisBlogKey);
        //判断是否为空
        if (intersect == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        if (intersect.size() == 0)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //过滤获得id
        List<Long> ids =
intersect.stream().map(Long::valueOf).collect(Collectors.toList());
        //查询
        List<User> users = userService.listByIds(ids);
        //转换
        List<UserDTO> collect = users.stream()
                .map(user -> (BeanUtil.copyProperties(user, UserDTO.class)))
                .collect(Collectors.toList());
        //返回
        return Result.ok(collect);
    }
}
```

关注推送

是什么?

关注推送也叫做Feed流，直译为投喂。为用户持续的提供“沉浸式”的体验，通过无限下拉刷新获取新的信息

Feed流的模式

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

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，推送用户感兴趣的信息来吸引用户

- 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

Timeline的三种实现模式

拉模式

拉模式也叫做读扩散。假设有三个人，分别是张三、李四、王五，这三个人分别会在自己的账号发布自己的笔记或者视频，在这里我们统一称之为消息，这三个人发布的所有的消息都会被发送到发件箱中，发送到发件箱中的消息除了消息本身之外，还需要带有时间戳。粉丝赵六会有一个收件箱，平时收件箱是空的，只有他在读取消息时，才会把赵六关注的所有人的发件箱中的消息拉取到他的收件箱中，拉取到收件箱后，消息会按照携带的时间戳进行排序，然后赵六就可以读取消息了。

优点：

- 节省内存空间。收件箱中的消息在读取完后就会被清空，下次需要读取的时候会重新从所有关注人的发件箱中拉取。消息只保存了一份。

缺点：

- 每次读取消息都要重新拉取发件箱中的消息，然后再做排序，比较耗时。

推模式

推模式也叫做写扩散。假设现在有两个 up 主：张三、李四，有三个粉丝：粉丝1、粉丝2、粉丝3，粉丝1关注了张三，粉丝2和3都关注了张三和李四，如果张三此时想要发送消息，那么这条消息会直接推送到张三的所有粉丝的收件箱中，而李四发送的消息也会被推送到粉丝2和3的收件箱中，收件箱收到消息后会对所有的消息进行排序，粉丝在读取消息时，就已经是排序好的消息了。这样的一个好处就是延时低，不必每次读取时都需要重新拉取消息。但这种方式内存占用会比较高，up 主每次发消息都要同步所有的粉丝，如果粉丝数过多，超过几百万，就需要复制几百万份。

优点：

- 延时低，不必每次读取时都需要重新拉取消息

缺点：

- 内存占用会比较高

推拉结合

推拉结合也叫读写混合，兼具推和拉两种模式的优点。

普通粉丝人数众多，但是活跃度较低，读取消息的频率也就低，可采用拉模式；
而活跃粉丝人数少，但是活跃度高，读取消息的频率高，可采用推模式。

大 V 发送消息时，会直接将消息推送到活跃粉丝的发件箱中，而针对于普通粉丝，消息会先发送到发件箱中，当普通粉丝读取消息时，会直接从发件箱中拉取。

基于推模式实现关注推送功能

发送

BlogController:

```
package mao.spring_boot_redis_hmdp.controller;


import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import java.util.List;


@RestController
@RequestMapping("/blog")
public class BlogController
{

    @Resource
    private IBlogService blogService;


    /**
     * 保存（发布）博客信息
     *
     * @param blog Blog
     * @return Result
     */
    @PostMapping
    public Result saveBlog(@RequestBody Blog blog)
    {
        return blogService.saveBlog(blog);
    }

    @PutMapping("/like/{id}")
```

```
    public Result likeBlog(@PathVariable("id") Long id)
    {
        return blogService.likeBlog(id);
    }

    @GetMapping("/of/me")
    public Result queryMyBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        // 获取登录用户
        UserDTO user = UserHolder.getUser();
        // 根据用户查询
        Page<Blog> page = blogService.query()
                .eq("user_id", user.getId()).page(new Page<>(current,
SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        return Result.ok(records);
    }

    @GetMapping("/hot")
    public Result queryHotBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        return blogService.queryHotBlog(current);
    }

    @GetMapping("/{id}")
    public Result queryBlogById(@PathVariable("id") String id)
    {
        return blogService.queryBlogById(id);
    }

    @GetMapping("/likes/{id}")
    public Result queryBlogLikes(@PathVariable("id") String id)
    {
        return blogService.queryBlogLikes(id);
    }

    /**
     * 查询用户的笔记信息
     *
     * @param current 当前页，如果不指定，则为第一页
     * @param id      博主的id
     * @return Result
     */
    @GetMapping("/of/user")
    public Result queryBlogByUserId(@RequestParam(value = "current",
defaultValue = "1") Integer current,
                                    @RequestParam("id") Long id)
    {
        //根据用户查询
        Page<Blog> page = blogService.query().
                eq("user_id", id).
                page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        //获取当前页数据
        List<Blog> records = page.getRecords();
        //返回
```

```
        return Result.ok(records);
    }


}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;


import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;


public interface IBlogService extends IService<Blog>
{
    /**
     * 查询热门的探店笔记
     *
     * @param current 当前页
     * @return Result
     */
    Result queryHotBlog(Integer current);

    /**
     * 根据id进行查询
     *
     * @param id id
     * @return Result
     */
    Result queryBlogById(String id);

    /**
     * 点赞功能
     *
     * @param id id
     * @return result
     */
    Result likeBlog(Long id);

    /**
     * 点赞排行榜
     *
     * @param id id
     * @return Result
     */
    Result queryBlogLikes(String id);

    /**
     * 保存（发布）博客信息
     *
     * @param blog Blog
```

```
     * @return Result
     */
    Result saveBlog(Blog blog);
}
```

实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.bean.BeanUtil;
import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.entity.Follow;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.BlogMapper;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.service.IFollowService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;


@Service("blogService")
public class BlogServiceImpl extends ServiceImpl<BlogMapper, Blog> implements
IBlogService
{

    @Resource
    private IUserService userService;

    @Resource
    private RedisUtils redisUtils;

    @Resource
    private IFollowService followService;
```

```
    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result queryHotBlog(Integer current)
    {
        // 根据用户查询
        Page<Blog> page = query()
                .orderByDesc("liked")
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        // 查询用户
        records.forEach(blog ->
        {
            Long userId = blog.getUserId();
            User user = userService.getById(userId);
            blog.setName(user.getNickName());
            blog.setIcon(user.getIcon());
        });
        return Result.ok(records);
    }

    @Override
    public Result queryBlogById(String id)
    {
        //查询
        //Blog blog = this.getById(id);
        Blog blog = redisUtils.query(RedisConstants.BLOG_KEY,
                RedisConstants.LOCK_BLOG_KEY, id,
                Blog.class, this::getById,
                RedisConstants.CACHE_BLOG_TTL,
                TimeUnit.MINUTES, 120);
        //判断是否存在
        if (blog == null)
        {
            //不存在，返回
            return Result.fail("该笔记信息不存在");
        }
        //存在
        //填充用户信息
        //获得用户id
        Long userId = blog.getUserId();
        //查询
        User user = userService.getById(userId);
        //填充
        blog.setIcon(user.getIcon());
        blog.setName(user.getNickName());
        //返回
        return Result.ok(blog);
    }

    @Override
    public Result likeBlog(Long id)
    {
        //获取用户信息
        UserDTO user = UserHolder.getUser();
        //判断用户是否已经点赞(检查设置在key是否包含value)
```

```
        Boolean member =
stringRedisTemplate.opsForSet().isMember(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
        if (BooleanUtil.isFalse(member))
        {
            //未点赞
            //数据库点赞数量+1
            boolean update = update().setSql("liked = liked + 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //保存用户到Redis的set集合

 stringRedisTemplate.opsForSet().add(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
            }

        }
        else
        {
            //已点赞，取消点赞
            //数据库点赞数量-1
            boolean update = update().setSql("liked = liked - 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //移除用户
                stringRedisTemplate.delete(RedisConstants.BLOG_LIKED_KEY + id);
            }
        }

        return Result.ok();
    }

    @Override
    public Result queryBlogLikes(String id)
    {
        //获得key
        String redisKey = RedisConstants.BLOG_LIKED_KEY + id;
        //查询前5名的点赞的用户(从排序集中获取start和end之间的元素)
        Set<String> range = stringRedisTemplate.opsForZSet().range(redisKey, 0,
4);
        //判断
        if (range == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //非空
        //解析出用户的id
```

```
        List<Long> ids =
range.stream().map(Long::valueOf).collect(Collectors.toList());
        //拼接
        String join = StrUtil.join(",", ids);
        //查询数据库
        List<User> users = userService.query().in("id", ids).last("order by
filed(id, " + join + ")").list();
        //转换成dto
        List<UserDTO> dtoList = users.stream().map(user ->
BeanUtil.copyProperties(user, UserDTO.class)).
                collect(Collectors.toList());
        //返回数据
        return Result.ok(dtoList);
    }

    @Override
    public Result saveBlog(Blog blog)
    {
        //获取登录用户
        UserDTO user = UserHolder.getUser();
        blog.setUserId(user.getId());
        //保存探店博文
        boolean save = this.save(blog);
        //判断是否保存成功
        if (!save)
        {
            //保存失败
            return Result.ok();
        }
        //保存成功
        //先查询笔记作者的所有粉丝
        QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("follow_user_id", user.getId());
        List<Follow> followList = followService.list(queryWrapper);
        //判断是否为空
        if (followList == null || followList.isEmpty())
        {
            //为空，无粉丝或者为null
            return Result.ok(blog.getId());
        }
        //不为空
        //推送给所有粉丝
        for (Follow follow : followList)
        {
            //获得用户id
            Long userId = follow.getUserId();
            //放入redis的zset集合里
            stringRedisTemplate.opsForZSet().add(RedisConstants.FEED_KEY +
userId,
                    blog.getId().toString(),
                    System.currentTimeMillis());
        }
        //返回
        return Result.ok(blog.getId());
    }
}
```

用户读取

实现滚动分页的命令：

```
ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]
```

返回有序集 key 中， score 值介于 max 和 min 之间(默认包括等于 max 或 min)的所有的成员。有序集成员按 score 值递减(从大到小)的次序排列。WITHSCORES 表示是否返回成员分数。LIMIT 表示是否分页，如果带有 LIMIT，则后面必须带有offset、count，offset表示偏移量（相对于max值而言），count表示结果数量。max 值是上一次查询结果中的最小分数（即时间戳）。而 offset 的取值也与上一次查询结果中的最小分数有关，如果上一次查询结果中的最小分数值重复多次出现，offset的值就应该为最小分数重复出现的次数。

结果类：

```
package mao.spring_boot_redis_hmdp.dto;

import lombok.Data;

import java.util.List;

@Data
public class ScrollResult
{
    /**
     * list集合，用于放分页数据
     */
    private List<?> list;
    /**
     * 最小的时间戳
     */
    private Long minTime;
    /**
     * 偏移量
     */
    private Integer offset;
}
```

BlogController:

```
package mao.spring_boot_redis_hmdp.controller;


import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
```

```
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import java.util.List;


@RestController
@RequestMapping("/blog")
public class BlogController
{

    @Resource
    private IBlogService blogService;


    /**
     * 保存（发布）博客信息
     *
     * @param blog Blog
     * @return Result
     */
    @PostMapping
    public Result saveBlog(@RequestBody Blog blog)
    {
        return blogService.saveBlog(blog);
    }

    @PutMapping("/like/{id}")
    public Result likeBlog(@PathVariable("id") Long id)
    {
        return blogService.likeBlog(id);
    }

    @GetMapping("/of/me")
    public Result queryMyBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        // 获取登录用户
        UserDTO user = UserHolder.getUser();
        // 根据用户查询
        Page<Blog> page = blogService.query()
                .eq("user_id", user.getId()).page(new Page<>(current,
SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        return Result.ok(records);
    }

    @GetMapping("/hot")
    public Result queryHotBlog(@RequestParam(value = "current", defaultValue =
"1") Integer current)
    {
        return blogService.queryHotBlog(current);
    }

    @GetMapping("/{id}")
    public Result queryBlogById(@PathVariable("id") String id)
```

```
    {
        return blogService.queryBlogById(id);
    }

    @GetMapping("/likes/{id}")
    public Result queryBlogLikes(@PathVariable("id") String id)
    {
        return blogService.queryBlogLikes(id);
    }

    /**
     * 查询用户的笔记信息
     *
     * @param current 当前页，如果不指定，则为第一页
     * @param id      博主的id
     * @return Result
     */
    @GetMapping("/of/user")
    public Result queryBlogByUserId(@RequestParam(value = "current",
defaultValue = "1") Integer current,
                                    @RequestParam("id") Long id)
    {
        //根据用户查询
        Page<Blog> page = blogService.query().
                eq("user_id", id).
                page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        //获取当前页数据
        List<Blog> records = page.getRecords();
        //返回
        return Result.ok(records);
    }

    /**
     * 从收件箱里取关注的人发的信息
     *
     * @param max    时间戳，第一页为当前时间，第n页为第n-1页最后一条数据的时间戳
     * @param offset 偏移量，第一页为0，不是第一页，取决于上一页最后一个时间戳的条数
     * @return Result
     */
    @GetMapping("/of/follow")
    public Result queryBlogOfFollow(@RequestParam("lastId") Long max,
                                    @RequestParam(value = "offset", defaultValue
= "0") Integer offset)
    {
        return blogService.queryBlogOfFollow(max, offset);
    }

}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;
```

```java
import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Blog;


public interface IBlogService extends IService<Blog>
{
    /**
     * 查询热门的探店笔记
     *
     * @param current 当前页
     * @return Result
     */
    Result queryHotBlog(Integer current);

    /**
     * 根据id进行查询
     *
     * @param id id
     * @return Result
     */
    Result queryBlogById(String id);

    /**
     * 点赞功能
     *
     * @param id id
     * @return result
     */
    Result likeBlog(Long id);

    /**
     * 点赞排行榜
     *
     * @param id id
     * @return Result
     */
    Result queryBlogLikes(String id);

    /**
     * 保存（发布）博客信息
     *
     * @param blog Blog
     * @return Result
     */
    Result saveBlog(Blog blog);

    /**
     * 从收件箱里取关注的人发的信息
     *
     * @param max    时间戳，第一页为当前时间，第n页为第n-1页最后一条数据的时间戳
     * @param offset 偏移量，第一页为0，不是第一页，取决于上一页最后一个时间戳的条数
     * @return Result
     */
    Result queryBlogOfFollow(Long max, Integer offset);
}
```

实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.bean.BeanUtil;
import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.ScrollResult;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.Blog;
import mao.spring_boot_redis_hmdp.entity.Follow;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.BlogMapper;
import mao.spring_boot_redis_hmdp.service.IBlogService;
import mao.spring_boot_redis_hmdp.service.IFollowService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.ZSetOperations;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;


@Service("blogService")
public class BlogServiceImpl extends ServiceImpl<BlogMapper, Blog> implements
IBlogService
{

    @Resource
    private IUserService userService;

    @Resource
    private RedisUtils redisUtils;

    @Resource
    private IFollowService followService;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result queryHotBlog(Integer current)
    {
```

```
        // 根据用户查询
        Page<Blog> page = query()
                .orderByDesc("liked")
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 获取当前页数据
        List<Blog> records = page.getRecords();
        // 查询用户
        records.forEach(blog ->
        {
            Long userId = blog.getUserId();
            User user = userService.getById(userId);
            blog.setName(user.getNickName());
            blog.setIcon(user.getIcon());
        });
        return Result.ok(records);
    }

    @Override
    public Result queryBlogById(String id)
    {
        //查询
        //Blog blog = this.getById(id);
        Blog blog = redisUtils.query(RedisConstants.BLOG_KEY,
                RedisConstants.LOCK_BLOG_KEY, id,
                Blog.class, this::getById,
                RedisConstants.CACHE_BLOG_TTL,
                TimeUnit.MINUTES, 120);
        //判断是否存在
        if (blog == null)
        {
            //不存在，返回
            return Result.fail("该笔记信息不存在");
        }
        //存在
        //填充用户信息
        //获得用户id
        Long userId = blog.getUserId();
        //查询
        User user = userService.getById(userId);
        //填充
        blog.setIcon(user.getIcon());
        blog.setName(user.getNickName());
        //返回
        return Result.ok(blog);
    }

    @Override
    public Result likeBlog(Long id)
    {
        //获取用户信息
        UserDTO user = UserHolder.getUser();
        //判断用户是否已经点赞(检查设置在key是否包含value)
        Boolean member =
stringRedisTemplate.opsForSet().isMember(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
        if (BooleanUtil.isFalse(member))
        {
            //未点赞
```

```
            //数据库点赞数量+1
            boolean update = update().setSql("liked = liked + 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //保存用户到Redis的set集合

 stringRedisTemplate.opsForSet().add(RedisConstants.BLOG_LIKED_KEY + id,
user.getId().toString());
            }

        }
        else
        {
            //已点赞，取消点赞
            //数据库点赞数量-1
            boolean update = update().setSql("liked = liked - 1").eq("id",
id).update();
            //判断是否成功
            if (update)
            {
                //成功
                //让redis数据过期
                stringRedisTemplate.delete(RedisConstants.BLOG_KEY);
                //移除用户
                stringRedisTemplate.delete(RedisConstants.BLOG_LIKED_KEY + id);
            }
        }

        return Result.ok();
    }

    @Override
    public Result queryBlogLikes(String id)
    {
        //获得key
        String redisKey = RedisConstants.BLOG_LIKED_KEY + id;
        //查询前5名的点赞的用户(从排序集中获取start和end之间的元素)
        Set<String> range = stringRedisTemplate.opsForZSet().range(redisKey, 0,
4);
        //判断
        if (range == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //非空
        //解析出用户的id
        List<Long> ids =
range.stream().map(Long::valueOf).collect(Collectors.toList());
        //拼接
        String join = StrUtil.join(",", ids);
        //查询数据库
```

```
        List<User> users = userService.query().in("id", ids).last("order by
filed(id, " + join + ")").list();
        //转换成dto
        List<UserDTO> dtoList = users.stream().map(user ->
BeanUtil.copyProperties(user, UserDTO.class)).
                collect(Collectors.toList());
        //返回数据
        return Result.ok(dtoList);
    }

    @Override
    public Result saveBlog(Blog blog)
    {
        //获取登录用户
        UserDTO user = UserHolder.getUser();
        blog.setUserId(user.getId());
        //保存探店博文
        boolean save = this.save(blog);
        //判断是否保存成功
        if (!save)
        {
            //保存失败
            return Result.ok();
        }
        //保存成功
        //先查询笔记作者的所有粉丝
        QueryWrapper<Follow> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("follow_user_id", user.getId());
        List<Follow> followList = followService.list(queryWrapper);
        //判断是否为空
        if (followList == null || followList.isEmpty())
        {
            //为空，无粉丝或者为null
            return Result.ok(blog.getId());
        }
        //不为空
        //推送给所有粉丝
        for (Follow follow : followList)
        {
            //获得用户id
            Long userId = follow.getUserId();
            //放入redis的zset集合里
            stringRedisTemplate.opsForZSet().add(RedisConstants.FEED_KEY +
userId,
                    blog.getId().toString(),
                    System.currentTimeMillis());
        }
        //返回
        return Result.ok(blog.getId());
    }

    @Override
    public Result queryBlogOfFollow(Long max, Integer offset)
    {
        //获得当前登录用户
        UserDTO user = UserHolder.getUser();
        //key
        String redisKey = RedisConstants.FEED_KEY + user.getId();
```

```
        //从redis收件箱里取数据
        //参数2：最小分数 参数3：最大分数 参数4：偏移量 参数5：每次取几条
        Set<ZSetOperations.TypedTuple<String>> typedTuples =
stringRedisTemplate.opsForZSet().
                reverseRangeByScoreWithScores(redisKey, 0, max, offset, 3);
        //TypedTuple里有V getValue();  和Double getScore(); 方法
        //判断是否为空
        if (typedTuples == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //不为空
        //最后一个时间戳重复的数量
        int count = 1;
        //最小时间戳
        long minTime = 0;
        List<Long> ids = new ArrayList<>(typedTuples.size());
        //遍历
        for (ZSetOperations.TypedTuple<String> typedTuple : typedTuples)
        {
            //加入到list集合里

 ids.add(Long.valueOf(Objects.requireNonNull(typedTuple.getValue())));
            //获得时间戳
            long time =
Objects.requireNonNull(typedTuple.getScore()).longValue();
            if (time == minTime)
            {
                //时间是最小时间，计数器+1
                count++;
            }
            else
            {
                //不是最小时间，刷新最小时间，计数器清成1（包含自己）
                minTime = time;
                count = 1;
            }
        }
        String join = StrUtil.join(",", ids);
        //查数据库
        List<Blog> blogs = this.query().in("id", ids).last("order by field (id,"
+ join + ")").list();
        //封装结果
        ScrollResult scrollResult = new ScrollResult();
        scrollResult.setList(blogs);
        scrollResult.setMinTime(minTime);
        scrollResult.setOffset(count);
        //返回
        return Result.ok(scrollResult);
    }
}
```

实现查看附近商铺功能

GEO数据结构

是什么?

GEO就是Geolocation的简写形式，代表地理坐标。Redis在3.2版本中加入了对GEO的支持，允许存储地理坐标信息， 帮助我们根据经纬度来检索数据

常用命令

- GEOADD：添加一个地理空间信息，包含：经度（longitude）、纬度（latitude）、值（member）
- GEODIST：计算指定的两个点之间的距离并返回
- GEOHASH：将指定 member 的坐标转为 hash 字符串形式并返回
- GEOPOS：返回指定 member 的坐标
- GEORADIUS：指定圆心、半径，找到该圆内包含的所有 member，并按照与圆心之间的距离排序后返回。6.2 以后已废弃
- GEOSEARCH：在指定范围内搜索 member，并按照与指定点之间的距离排序后返回。范围可以是圆形或矩形。6.2 新功能
- GEOSEARCHSTORE：与 GEOSEARCH 功能一致，不过可以把结果存储到一个指定的 key。6.2 新功能

实现

将店铺信息加载到redis:

```
package mao.spring_boot_redis_hmdp;

import mao.spring_boot_redis_hmdp.entity.Shop;
import mao.spring_boot_redis_hmdp.service.IShopService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.geo.Point;
import org.springframework.data.redis.connection.RedisGeoCommands;
import org.springframework.data.redis.core.StringRedisTemplate;

import javax.annotation.Resource;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@SpringBootTest
class SpringBootRedisHmdp
{

    @Resource
```

```
    private IShopService shopService;

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Test
    void contextLoads()
    {
    }

    @Test
    void load()
    {
        //查询店铺信息
        List<Shop> list = shopService.list();
        //店铺分组，放入到一个集合中
        Map<Long, List<Shop>> collect =
list.stream().collect(Collectors.groupingBy(Shop::getTypeId));
        //分批写入redis
        for (Long typeId : collect.keySet())
        {
            //值
            List<Shop> shops = collect.get(typeId);
            List<RedisGeoCommands.GeoLocation<String>> locations = new
ArrayList<>(shops.size());
            for (Shop shop : shops)
            {
                locations.add(new RedisGeoCommands.GeoLocation<>
                        (shop.getId().toString(), new Point(shop.getX(),
shop.getY())));
            }
            //写入redis
            stringRedisTemplate.opsForGeo().add(RedisConstants.SHOP_GEO_KEY +
typeId, locations);
        }
    }
}
```

ShopController:

```
package mao.spring_boot_redis_hmdp.controller;


import cn.hutool.core.util.StrUtil;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Shop;
import mao.spring_boot_redis_hmdp.service.IShopService;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
```

```java
@RestController
@RequestMapping("/shop")
public class ShopController
{

    @Resource
    public IShopService shopService;

    /**
     * 根据id查询商铺信息
     *
     * @param id 商铺id
     * @return 商铺详情数据
     */
    @GetMapping("/{id}")
    public Result queryShopById(@PathVariable("id") Long id)
    {
        return Result.ok(shopService.queryShopById(id));
    }

    /**
     * 新增商铺信息
     *
     * @param shop 商铺数据
     * @return 商铺id
     */
    @PostMapping
    public Result saveShop(@RequestBody Shop shop)
    {
        // 写入数据库
        shopService.save(shop);
        // 返回店铺id
        return Result.ok(shop.getId());
    }

    /**
     * 更新商铺信息
     *
     * @param shop 商铺数据
     * @return 无
     */
    @PutMapping
    public Result updateShop(@RequestBody Shop shop)
    {
        return shopService.updateShop(shop);
    }


    /**
     * 根据商铺类型分页查询商铺信息
     *
     * @param typeId  商铺类型
     * @param current 页码
     * @param x       坐标轴x
     * @param y       坐标轴y
     * @return 商铺列表
     */
    @GetMapping("/of/type")
```

```
    public Result queryShopByType(
            @RequestParam("typeId") Integer typeId,
            @RequestParam(value = "current", defaultValue = "1") Integer
current,
            @RequestParam(value = "x", required = false) Double x,
            @RequestParam(value = "y", required = false) Double y
    )
    {
        return shopService.queryShopByType(typeId, current, x, y);
    }

    /**
     * 根据商铺名称关键字分页查询商铺信息
     *
     * @param name     商铺名称关键字
     * @param current 页码
     * @return 商铺列表
     */
    @GetMapping("/of/name")
    public Result queryShopByName(
            @RequestParam(value = "name", required = false) String name,
            @RequestParam(value = "current", defaultValue = "1") Integer current
    )
    {
        // 根据类型分页查询
        Page<Shop> page = shopService.query()
                .like(StrUtil.isNotBlank(name), "name", name)
                .page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
        // 返回数据
        return Result.ok(page.getRecords());
    }
}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;

import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Shop;


public interface IShopService extends IService<Shop>
{
    /**
     * 根据id查询商户信息，有缓存
     *
     * @param id 商户的id
     * @return Result
     */
    Result queryShopById(Long id);

    /**
     * 更新商户信息，包含更新缓存
```

```
     *
     * @param shop 商户信息
     * @return Result
     */
    Result updateShop(Shop shop);

    /**
     * 根据商铺类型分页查询商铺信息
     *
     * @param typeId  商铺类型
     * @param current 页码
     * @param x       坐标轴x
     * @param y       坐标轴y
     * @return 商铺列表
     */
    Result queryShopByType(Integer typeId, Integer current, Double x, Double y);
}
```

实现类:

```
package mao.spring_boot_redis_hmdp.service.impl;


import cn.hutool.core.util.BooleanUtil;
import cn.hutool.core.util.StrUtil;
import cn.hutool.json.JSONObject;
import cn.hutool.json.JSONUtil;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.RedisData;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.Shop;
import mao.spring_boot_redis_hmdp.mapper.ShopMapper;
import mao.spring_boot_redis_hmdp.service.IShopService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RedisUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import org.springframework.data.geo.*;
import org.springframework.data.redis.connection.RedisGeoCommands;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.time.LocalDateTime;
import java.util.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

@Service
public class ShopServiceImpl extends ServiceImpl<ShopMapper, Shop> implements
IShopService
{
```

```
    @Resource
    StringRedisTemplate stringRedisTemplate;

    @Resource
    private RedisUtils redisUtils;


    @Override
    public Result queryShopById(Long id)
    {
        //查询
        //Shop shop = this.queryWithMutex(id);
        //Shop shop = this.queryWithLogicalExpire(id);
        Shop shop = redisUtils.query(RedisConstants.CACHE_SHOP_KEY,
RedisConstants.LOCK_SHOP_KEY, id, Shop.class, this::getById,
                RedisConstants.CACHE_SHOP_TTL, TimeUnit.MINUTES, 300);
        //判断
        if (shop == null)
        {
            //不存在
            return Result.fail("店铺信息不存在");
        }
        //返回
        return Result.ok(shop);
    }

    /**
     * 互斥锁解决缓存击穿问题
     *
     * @param id 商铺id
     * @return Shop
     */
    private Shop queryWithMutex(Long id)
    {
        //获取redisKey
        String redisKey = RedisConstants.CACHE_SHOP_KEY + id;
        //从redis中查询商户信息，根据id
        String shopJson = stringRedisTemplate.opsForValue().get(redisKey);
        //判断取出的数据是否为空
        if (StrUtil.isNotBlank(shopJson))
        {
            //不是空，redis里有，返回
            return JSONUtil.toBean(shopJson, Shop.class);
        }
        //是空串，不是null，返回
        if (shopJson != null)
        {
            return null;
        }
        //锁的key
        String lockKey = RedisConstants.LOCK_SHOP_KEY + id;

        Shop shop = null;
        try
        {
            //获取互斥锁
            boolean lock = tryLock(lockKey);
            //判断锁是否获取成功
```

```
            if (!lock)
            {
                //没有获取到锁
                //200毫秒后再次获取
                Thread.sleep(200);
                //递归调用
                return queryWithMutex(id);
            }
            //得到了锁
            //null，查数据库
            shop = this.getById(id);
            //判断数据库里的信息是否为空
            if (shop == null)
            {
                //空，将空值写入redis，返回错误
                stringRedisTemplate.opsForValue().set(redisKey, "",
RedisConstants.CACHE_NULL_TTL, TimeUnit.MINUTES);
                return null;
            }
            //存在，回写到redis里，设置随机的过期时间
            stringRedisTemplate.opsForValue().set(redisKey,
JSONUtil.toJsonStr(shop),
                    RedisConstants.CACHE_SHOP_TTL * 60 + getIntRandom(0, 300),
TimeUnit.SECONDS);
        }
        catch (InterruptedException e)
        {
            throw new RuntimeException(e);
        }
        finally
        {
            //释放锁
            //System.out.println("释放锁");
            this.unlock(lockKey);
        }
        //返回数据
        return shop;
    }

    //线程池
    private static final ExecutorService CACHE_REBUILD_EXECUTOR =
Executors.newFixedThreadPool(10);

    /**
     * 使用逻辑过期解决缓存击穿问题
     *
     * @param id 商铺id
     * @return Shop
     */
    private Shop queryWithLogicalExpire(Long id)
    {
        //获取redisKey
        String redisKey = RedisConstants.CACHE_SHOP_KEY + id;
        //从redis中查询商户信息，根据id
        String shopJson = stringRedisTemplate.opsForValue().get(redisKey);
        //判断取出的数据是否为空
        if (StrUtil.isBlank(shopJson))
        {
```

```
            //是空，redis里没有，返回
            return null;
        }

        //json转类
        RedisData redisData = JSONUtil.toBean(shopJson, RedisData.class);
        //获取过期时间
        LocalDateTime expireTime = redisData.getExpireTime();
        //获取商铺信息
        Shop shop = JSONUtil.toBean((JSONObject) redisData.getData(),
Shop.class);
        //判断是否过期
        if (expireTime.isAfter(LocalDateTime.now()))
        {
            //没有过期，返回
            return shop;
        }
        //已经过期，缓存重建
        //获取互斥锁
        String lockKey = RedisConstants.LOCK_SHOP_KEY + id;
        boolean isLock = tryLock(lockKey);
        if (isLock)
        {
            //获取锁成功
            // 开辟独立线程
            CACHE_REBUILD_EXECUTOR.submit(new Runnable()
            {
                @Override
                public void run()
                {
                    try
                    {
                        saveShop2Redis(id, 20L);
                    }
                    catch (InterruptedException e)
                    {
                        throw new RuntimeException(e);
                    }
                    finally
                    {
                        //释放锁
                        unlock(lockKey);
                    }
                }
            });
        }
        //没有获取到锁，使用旧数据返回
        return shop;

    }

    /**
     * 保存商铺信息到redis
     *
     * @param id            商铺的id
     * @param expireSeconds 过期的时间，单位是秒
     * @throws InterruptedException 异常
     */
```

```
    public void saveShop2Redis(Long id, Long expireSeconds) throws
InterruptedException
    {
        // 查询数据库
        Shop shop = getById(id);
        // 封装缓存过期时间
        RedisData redisData = new RedisData();
        redisData.setData(shop);
        redisData.setExpireTime(LocalDateTime.now().plusSeconds(expireSeconds));
        //保存到redis
        stringRedisTemplate.opsForValue().set(RedisConstants.CACHE_SHOP_KEY,
JSONUtil.toJsonStr(redisData));
    }


    @Override
    public Result updateShop(Shop shop)
    {
        //获得id
        Long id = shop.getId();
        //判断是否为空
        if (id == null)
        {
            return Result.fail("商户id不能为空");
        }
        //不为空
        //先更新数据库
        boolean b = this.updateById(shop);
        //更新失败，返回
        if (!b)
        {
            return Result.fail("更新失败");
        }
        //更新没有失败
        //删除redis里的数据，下一次查询时自动添加进redis
        //redisKey
        String redisKey = RedisConstants.CACHE_SHOP_KEY + id;
        stringRedisTemplate.delete(redisKey);
        //返回响应
        return Result.ok();
    }

    @Override
    public Result queryShopByType(Integer typeId, Integer current, Double x,
Double y)
    {
        //判断是否传递了坐标轴信息，如果没有传递，基本分页
        if (x == null || y == null)
        {
            // 根据类型分页查询
            Page<Shop> page = this.query()
                    .eq("type_id", typeId)
                    .page(new Page<>(current,
SystemConstants.DEFAULT_PAGE_SIZE));
            // 返回数据
            return Result.ok(page.getRecords());
        }
        //传递了坐标信息
```

```
        int from = (current - 1) * SystemConstants.DEFAULT_PAGE_SIZE;
        int end = current * SystemConstants.DEFAULT_PAGE_SIZE;
        //按距离排序且分页
GeoSearchCommandArgs.newGeoSearchArgs().includeDistance().limit(end))
        GeoResults<RedisGeoCommands.GeoLocation<String>> geoResults =
stringRedisTemplate.
                opsForGeo().geoRadius(RedisConstants.SHOP_GEO_KEY + typeId,
                        new Circle(new Point(x, y), 5000),

 RedisGeoCommands.GeoRadiusCommandArgs.newGeoRadiusArgs().includeDistance().limi
t(end));
        //判断是否为空
        if (geoResults == null)
        {
            //返回空集合
            return Result.ok(Collections.emptyList());
        }
        //不为空
        //获取内容
        List<GeoResult<RedisGeoCommands.GeoLocation<String>>> content =
geoResults.getContent();
        //判断是否到底
        if (from >= content.size())
        {
            return Result.ok(Collections.emptyList());
        }
        List<Long> ids = new ArrayList<>(content.size());
        Map<String, Distance> distanceMap = new HashMap<>(content.size());
        //截取from到end的部分
        content.stream().skip(from).forEach(result ->
        {
            //获取店铺的id
            String id = result.getContent().getName();
            //加入到集合中
            ids.add(Long.valueOf(id));
            //获得距离信息
            Distance distance = result.getDistance();
            //加入到map集合里
            distanceMap.put(id, distance);
            //System.out.println(id+"-----"+distance.getValue());
        });
        //拼接
        String join = StrUtil.join(",", ids);
        //查询数据库
        List<Shop> shops = this.query().in("id", ids).last("order by field(id,"
+ join + ")").list();
        //填充距离信息
        for (Shop shop : shops)
        {

 shop.setDistance(distanceMap.get(shop.getId().toString()).getValue());
        }
        //返回
        return Result.ok(shops);
    }

    /**
     * 获取一个随机数，区间包含min和max
```

```
     *
     * @param min 最小值
     * @param max 最大值
     * @return int 型的随机数
     */
    @SuppressWarnings("all")
    private int getIntRandom(int min, int max)
    {
        if (min > max)
        {
            min = max;
        }
        return min + (int) (Math.random() * (max - min + 1));
    }

    /**
     * 获取锁
     *
     * @param key redisKey
     * @return 获取锁成功，返回true，否则返回false
     */
    private boolean tryLock(String key)
    {
        Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(key, "1",
                RedisConstants.LOCK_SHOP_TTL, TimeUnit.SECONDS);
        return BooleanUtil.isTrue(result);
    }

    /**
     * 释放锁
     *
     * @param key redisKey
     */
    private void unlock(String key)
    {
        stringRedisTemplate.delete(key);
    }

}
```

用户签到

按月来统计用户签到信息，签到记录为 1，未签到则记录 0

把每一个bit位对应当月的每一天，形成了映射关系。用0和1标示业务状态，这种思路就称为位图

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用法

- 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 中的 bit 数组中的指定位置（offset）的值。
- BITFIELD_RO：获取 BitMap 中的 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

```
127.0.0.1:6379> setbit bit 0 1
(integer) 0
127.0.0.1:6379> setbit bit 1 1
(integer) 0
127.0.0.1:6379> setbit bit 2 1
(integer) 0
127.0.0.1:6379> setbit bit 5 1
(integer) 0
127.0.0.1:6379> getbit bit 0
(integer) 1
127.0.0.1:6379> getbit bit 1
(integer) 1
127.0.0.1:6379> getbit bit 2
(integer) 1
127.0.0.1:6379> getbit bit 3
(integer) 0
127.0.0.1:6379> getbit bit 4
(integer) 0
127.0.0.1:6379> getbit bit 5
(integer) 1
127.0.0.1:6379> getbit bit 6
(integer) 0
127.0.0.1:6379> BITFIELD bit get u1 0
1) (integer) 1
127.0.0.1:6379> BITFIELD bit get u2 0
1) (integer) 3
127.0.0.1:6379> BITFIELD bit get u5 0
1) (integer) 28
127.0.0.1:6379> BITCOUNT bit 0 5
(integer) 4
```

实现签到

UserController:

```
package mao.spring_boot_redis_hmdp.controller;


import cn.hutool.core.bean.BeanUtil;
import lombok.extern.slf4j.Slf4j;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.entity.UserInfo;
```

```java
import mao.spring_boot_redis_hmdp.service.IUserInfoService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import javax.servlet.http.HttpSession;


@Slf4j
@RestController
@RequestMapping("/user")
public class UserController
{

    @Resource
    private IUserService userService;

    @Resource
    private IUserInfoService userInfoService;

    /**
     * 发送手机验证码
     */
    @PostMapping("code")
    public Result sendCode(@RequestParam("phone") String phone, HttpSession
session)
    {
        return userService.sendCode(phone, session);
    }

    /**
     * 登录功能
     *
     * @param loginForm 登录参数，包含手机号、验证码；或者手机号、密码
     */
    @PostMapping("/login")
    public Result login(@RequestBody LoginFormDTO loginForm, HttpSession
session)
    {
        return userService.login(loginForm, session);
    }

    /**
     * 登出功能
     *
     * @return 无
     */
    @PostMapping("/logout")
    public Result logout()
    {
        // TODO 实现登出功能
        return Result.fail("功能未完成");
    }

    @GetMapping("/me")
    public Result me()
    {
```

```
        return Result.ok(UserHolder.getUser());
    }

    @GetMapping("/info/{id}")
    public Result info(@PathVariable("id") Long userId)
    {
        // 查询详情
        UserInfo info = userInfoService.getById(userId);
        if (info == null)
        {
            // 没有详情，应该是第一次查看详情
            return Result.ok();
        }
        info.setCreateTime(null);
        info.setUpdateTime(null);
        // 返回
        return Result.ok(info);
    }

    /**
     * 根据查询用户信息
     *
     * @param userId 用户的id
     * @return Result
     */
    @GetMapping("/{id}")
    public Result queryUserById(@PathVariable("id") Long userId)
    {
        //查询用户信息
        User user = userService.getById(userId);
        if (user == null)
        {
            return Result.ok();
        }
        //转换
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        return Result.ok(userDTO);
    }

    /**
     * 实现用户签到功能
     *
     * @return Result
     */
    @PostMapping("/sign")
    public Result sign()
    {
        return userService.sign();
    }
}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;
```

```
import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.User;

import javax.servlet.http.HttpSession;


public interface IUserService extends IService<User>
{
    /**
     * 发送短信验证码
     *
     * @param phone   手机号码
     * @param session HttpSession
     * @return Result
     */
    Result sendCode(String phone, HttpSession session);

    /**
     * 登录
     *
     * @param loginForm 登录信息，包含手机号和验证码（密码），一个实体类
     * @param session   HttpSession
     * @return Result
     */
    Result login(LoginFormDTO loginForm, HttpSession session);

    /**
     * 实现用户签到功能
     *
     * @return Result
     */
    Result sign();
}
```

实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;

import cn.hutool.core.bean.BeanUtil;
import cn.hutool.core.bean.copier.CopyOptions;
import cn.hutool.core.lang.UUID;
import cn.hutool.core.util.RandomUtil;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.UserMapper;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RegexUtils;
```

```
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import javax.servlet.http.HttpSession;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;


@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
IUserService
{

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result sendCode(String phone, HttpSession session)
    {
        //验证手机号
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //验证不通过，返回错误提示
            log.debug("验证码错误.....");
            return Result.fail("手机号错误，请重新填写");
        }
        //验证通过，生成验证码
        //6位数
        String code = RandomUtil.randomNumbers(6);
        //保存验证码到redis
        stringRedisTemplate.opsForValue().set(RedisConstants.LOGIN_CODE_KEY +
phone,
                code, RedisConstants.LOGIN_CODE_TTL, TimeUnit.MINUTES);
        //发送验证码
        log.debug("验证码发送成功," + code);
        //返回响应
        return Result.ok();
    }

    @Override
    public Result login(LoginFormDTO loginForm, HttpSession session)
    {
        //判断手机号格式是否正确
        String phone = loginForm.getPhone();
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //如果不正确则直接返回错误
            log.debug("手机号:" + phone + "错误");
            return Result.fail("手机号格式错误");
        }
        //判断验证码是否一致，redis中对比
```

```
        //String cacheCode = session.getAttribute("code").toString();
        String cacheCode =
stringRedisTemplate.opsForValue().get(RedisConstants.LOGIN_CODE_KEY + phone);
        String code = loginForm.getCode();
        //如果验证码为空，或者不一致，则返回验证码错误
        if (code == null || code.length() == 0)
        {
            return Result.fail("验证码不能为空");
        }
        //判断验证码是否为6位数
        if (code.length() != 6)
        {
            return Result.fail("验证码长度不正确");
        }
        //判断验证码是否正确
        if (!code.equals(cacheCode))
        {
            //验证码错误
            return Result.fail("验证码错误");
        }
        //验证码输入正确
        //判断用户是否存在
        User user = query().eq("phone", phone).one();
        //如果用户不存在则创建用户，保存到数据库
        if (user == null)
        {
            //创建用户，保存到数据库
            user = createUser(phone);
        }
        //如果用户存在，保存到redis
        //session.setAttribute("user", user);
        //生成token，作为登录令牌
        String token = UUID.randomUUID().toString(true);
        //将User对象转为Hash存储. UserDTO是用户的部分信息
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        //转map
        Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new HashMap<>
(), CopyOptions.create()
                .setIgnoreNullValue(true) // 忽略空的值
                .setFieldValueEditor((fieldName, fieldVaule) ->
fieldVaule.toString()));

        //保存到redis中
        //保存的key
        String tokenKey = RedisConstants.LOGIN_USER_KEY + token;
        //保存
        stringRedisTemplate.opsForHash().putAll(tokenKey, userMap);
        //设置有效期
        stringRedisTemplate.expire(tokenKey, RedisConstants.LOGIN_USER_TTL,
TimeUnit.MINUTES);
        //返回响应，返回token
        return Result.ok(token);
    }

    /**
     * 创建用户，添加到数据库中
     *
     * @param phone 手机号码
```

```
     * @return user
     */
    private User createUser(String phone)
    {
        User user = new User();
        user.setPhone(phone);
        user.setNickName(SystemConstants.USER_NICK_NAME_PREFIX +
RandomUtil.randomString(10));
        //将用户信息插入到 t_user表中
        this.save(user);
        //返回数据
        return user;
    }

    @Override
    public Result sign()
    {
        //获得当前登录的用户
        UserDTO user = UserHolder.getUser();
        //获得用户的id
        Long userId = user.getId();
        //获得当前的日期
        LocalDateTime now = LocalDateTime.now();
        //格式化，：年月
        String keySuffix = now.format(DateTimeFormatter.ofPattern(":yyyyMM"));
        //redis key
        String redisKey = RedisConstants.USER_SIGN_KEY + userId + keySuffix;
        //获得今天是本月的第几天
        int dayOfMonth = now.getDayOfMonth();
        //写入到redis
        stringRedisTemplate.opsForValue().setBit(redisKey, dayOfMonth - 1,
true);
        //返回
        return Result.ok();
    }
}
```

签到统计

- 什么叫做连续签到天数?

  从最后一次签到开始向前统计，直到遇到第一次未签到为止，计算总的签到次数，就是连续签到天数

- 如何得到本月到今天为止的所有签到数据?

命令：BITFIELD key GET u[dayOfMonth] 0

- 如何从后向前遍历每个bit位?

与 1 做与运算，就能得到最后一个bit位。 随后右移1位，下一个bit位就成为了最后一个bit位。

实现签到统计

UserController:

```
package mao.spring_boot_redis_hmdp.controller;


import cn.hutool.core.bean.BeanUtil;
import lombok.extern.slf4j.Slf4j;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.entity.UserInfo;
import mao.spring_boot_redis_hmdp.service.IUserInfoService;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.web.bind.annotation.*;

import javax.annotation.Resource;
import javax.servlet.http.HttpSession;


@Slf4j
@RestController
@RequestMapping("/user")
public class UserController
{

    @Resource
    private IUserService userService;

    @Resource
    private IUserInfoService userInfoService;

    /**
     * 发送手机验证码
     */
    @PostMapping("code")
    public Result sendCode(@RequestParam("phone") String phone, HttpSession
session)
    {
        return userService.sendCode(phone, session);
    }

    /**
     * 登录功能
     *
     * @param loginForm 登录参数，包含手机号、验证码；或者手机号、密码
     */
    @PostMapping("/login")
    public Result login(@RequestBody LoginFormDTO loginForm, HttpSession
session)
    {
        return userService.login(loginForm, session);
    }
```

```java
    /**
     * 登出功能
     *
     * @return 无
     */
    @PostMapping("/logout")
    public Result logout()
    {
        // TODO 实现登出功能
        return Result.fail("功能未完成");
    }

    @GetMapping("/me")
    public Result me()
    {
        return Result.ok(UserHolder.getUser());
    }

    @GetMapping("/info/{id}")
    public Result info(@PathVariable("id") Long userId)
    {
        // 查询详情
        UserInfo info = userInfoService.getById(userId);
        if (info == null)
        {
            // 没有详情，应该是第一次查看详情
            return Result.ok();
        }
        info.setCreateTime(null);
        info.setUpdateTime(null);
        // 返回
        return Result.ok(info);
    }

    /**
     * 根据查询用户信息
     *
     * @param userId 用户的id
     * @return Result
     */
    @GetMapping("/{id}")
    public Result queryUserById(@PathVariable("id") Long userId)
    {
        //查询用户信息
        User user = userService.getById(userId);
        if (user == null)
        {
            return Result.ok();
        }
        //转换
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        return Result.ok(userDTO);
    }

    /**
     * 实现用户签到功能
     *
     * @return Result
```

```
     */
    @PostMapping("/sign")
    public Result sign()
    {
        return userService.sign();
    }

    /**
     * 实现签到统计功能
     * 连续签到次数：从最后一次签到开始向前统计，直到遇到第一次未签到为止，计算总的签到次数
     *
     * @return Result
     */
    @GetMapping("/signCount")
    public Result signCount()
    {
        return userService.signCount();
    }
}
```

接口：

```
package mao.spring_boot_redis_hmdp.service;

import com.baomidou.mybatisplus.extension.service.IService;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.entity.User;

import javax.servlet.http.HttpSession;


public interface IUserService extends IService<User>
{
    /**
     * 发送短信验证码
     *
     * @param phone   手机号码
     * @param session HttpSession
     * @return Result
     */
    Result sendCode(String phone, HttpSession session);

    /**
     * 登录
     *
     * @param loginForm 登录信息，包含手机号和验证码（密码），一个实体类
     * @param session   HttpSession
     * @return Result
     */
    Result login(LoginFormDTO loginForm, HttpSession session);

    /**
```

```
     * 实现用户签到功能
     *
     * @return Result
     */
    Result sign();

    /**
     * 实现签到统计功能
     * 连续签到次数：从最后一次签到开始向前统计，直到遇到第一次未签到为止，计算总的签到次数
     *
     * @return Result
     */
    Result signCount();
}
```

实现类：

```
package mao.spring_boot_redis_hmdp.service.impl;

import cn.hutool.core.bean.BeanUtil;
import cn.hutool.core.bean.copier.CopyOptions;
import cn.hutool.core.lang.UUID;
import cn.hutool.core.util.RandomUtil;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import mao.spring_boot_redis_hmdp.dto.LoginFormDTO;
import mao.spring_boot_redis_hmdp.dto.Result;
import mao.spring_boot_redis_hmdp.dto.UserDTO;
import mao.spring_boot_redis_hmdp.entity.User;
import mao.spring_boot_redis_hmdp.mapper.UserMapper;
import mao.spring_boot_redis_hmdp.service.IUserService;
import mao.spring_boot_redis_hmdp.utils.RedisConstants;
import mao.spring_boot_redis_hmdp.utils.RegexUtils;
import mao.spring_boot_redis_hmdp.utils.SystemConstants;
import mao.spring_boot_redis_hmdp.utils.UserHolder;
import org.springframework.data.redis.connection.BitFieldSubCommands;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import javax.servlet.http.HttpSession;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.TimeUnit;


@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
IUserService
{
```

```
    @Resource
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Result sendCode(String phone, HttpSession session)
    {
        //验证手机号
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //验证不通过，返回错误提示
            log.debug("验证码错误.....");
            return Result.fail("手机号错误，请重新填写");
        }
        //验证通过，生成验证码
        //6位数
        String code = RandomUtil.randomNumbers(6);
        //保存验证码到redis
        stringRedisTemplate.opsForValue().set(RedisConstants.LOGIN_CODE_KEY +
phone,
                code, RedisConstants.LOGIN_CODE_TTL, TimeUnit.MINUTES);
        //发送验证码
        log.debug("验证码发送成功," + code);
        //返回响应
        return Result.ok();
    }

    @Override
    public Result login(LoginFormDTO loginForm, HttpSession session)
    {
        //判断手机号格式是否正确
        String phone = loginForm.getPhone();
        if (RegexUtils.isPhoneInvalid(phone))
        {
            //如果不正确则直接返回错误
            log.debug("手机号:" + phone + "错误");
            return Result.fail("手机号格式错误");
        }
        //判断验证码是否一致，redis中对比
        //String cacheCode = session.getAttribute("code").toString();
        String cacheCode =
stringRedisTemplate.opsForValue().get(RedisConstants.LOGIN_CODE_KEY + phone);
        String code = loginForm.getCode();
        //如果验证码为空，或者不一致，则返回验证码错误
        if (code == null || code.length() == 0)
        {
            return Result.fail("验证码不能为空");
        }
        //判断验证码是否为6位数
        if (code.length() != 6)
        {
            return Result.fail("验证码长度不正确");
        }
        //判断验证码是否正确
        if (!code.equals(cacheCode))
        {
            //验证码错误
            return Result.fail("验证码错误");
        }
```

```
        //验证码输入正确
        //判断用户是否存在
        User user = query().eq("phone", phone).one();
        //如果用户不存在则创建用户，保存到数据库
        if (user == null)
        {
            //创建用户，保存到数据库
            user = createUser(phone);
        }
        //如果用户存在，保存到redis
        //session.setAttribute("user", user);
        //生成token，作为登录令牌
        String token = UUID.randomUUID().toString(true);
        //将User对象转为Hash存储. UserDTO是用户的部分信息
        UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
        //转map
        Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new HashMap<>
(), CopyOptions.create()
                .setIgnoreNullValue(true) // 忽略空的值
                .setFieldValueEditor((fieldName, fieldVaule) ->
fieldVaule.toString()));

        //保存到redis中
        //保存的key
        String tokenKey = RedisConstants.LOGIN_USER_KEY + token;
        //保存
        stringRedisTemplate.opsForHash().putAll(tokenKey, userMap);
        //设置有效期
        stringRedisTemplate.expire(tokenKey, RedisConstants.LOGIN_USER_TTL,
TimeUnit.MINUTES);
        //返回响应，返回token
        return Result.ok(token);
    }

    /**
     * 创建用户，添加到数据库中
     *
     * @param phone 手机号码
     * @return user
     */
    private User createUser(String phone)
    {
        User user = new User();
        user.setPhone(phone);
        user.setNickName(SystemConstants.USER_NICK_NAME_PREFIX +
RandomUtil.randomString(10));
        //将用户信息插入到 t_user表中
        this.save(user);
        //返回数据
        return user;
    }

    @Override
    public Result sign()
    {
        //获得当前登录的用户
        UserDTO user = UserHolder.getUser();
        //获得用户的id
```

```
        Long userId = user.getId();
        //获得当前的日期
        LocalDateTime now = LocalDateTime.now();
        //格式化，：年月
        String keySuffix = now.format(DateTimeFormatter.ofPattern(":yyyyMM"));
        //redis key
        String redisKey = RedisConstants.USER_SIGN_KEY + userId + keySuffix;
        //获得今天是本月的第几天
        int dayOfMonth = now.getDayOfMonth();
        //写入到redis
        stringRedisTemplate.opsForValue().setBit(redisKey, dayOfMonth - 1,
true);
        //返回
        return Result.ok();
    }

    @Override
    public Result signCount()
    {
        //获得当前登录的用户
        UserDTO user = UserHolder.getUser();
        //获得用户的id
        Long userId = user.getId();
        //获得当前的日期
        LocalDateTime now = LocalDateTime.now();
        //格式化，：年月
        String keySuffix = now.format(DateTimeFormatter.ofPattern(":yyyyMM"));
        //redis key
        String redisKey = RedisConstants.USER_SIGN_KEY + userId + keySuffix;
        //获得今天是本月的第几天，日期：从 1 到 31
        int dayOfMonth = now.getDayOfMonth();
        //从redis里取签到结果
        List<Long> list = stringRedisTemplate.opsForValue()
                .bitField(redisKey,
                        BitFieldSubCommands.create()
                                .get(BitFieldSubCommands
                                        .BitFieldType
                                        .unsigned(dayOfMonth)).valueAt(0));
        //判断是否为空
        if (list == null || list.size() == 0)
        {
            //没有，返回0
            return Result.ok(0);
        }
        //取第一个，因为一个月最多有31天，小于32位，所以只有一个
        Long num = list.get(0);
        //判断第一个是否为空
        if (num == null || num == 0)
        {
            //第一个为0，返回直接0
            return Result.ok(0);
        }
        //计数器
        int count = 0;
        //循环遍历数据
        while (true)
        {
            //无符号，和1做与运算
```

```
            long result = num & 1;
            //判断是否为未签到
            if (result == 0)
            {
                //为签到，跳出循环
                break;
            }
            //不是0
            //计数器+1
            count++;
            //右移一位，左边会补0，所以不用担心会死循环
            num = num >> 1;
        }
        //返回
        return Result.ok(count);
    }
}
```

UV统计

HyperLogLog用法

- 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。

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

作用

做海量数据的统计工作

优点

- 内存占用极低
- 性能非常好

缺点

- 有一定的误差

```
127.0.0.1:6379> help pfadd
```

```
  PFADD key element [element ...]
  summary: Adds the specified elements to the specified HyperLogLog.
  since: 2.8.9
  group: hyperloglog

127.0.0.1:6379> help PFCOUNT

  PFCOUNT key [key ...]
  summary: Return the approximated cardinality of the set(s) observed by the
HyperLogLog at key(s).
  since: 2.8.9
  group: hyperloglog

127.0.0.1:6379> help PFMERGE

  PFMERGE destkey sourcekey [sourcekey ...]
  summary: Merge N different HyperLogLogs into a single one.
  since: 2.8.9
  group: hyperloglog
```

实现

```
    /**
     * 测试redis的uv统计功能
     */
    @Test
    void UV_statistics()
    {
        //发送单位,当前为1000条发一次，如果每次都发送会大大增加网络io
        int length = 1000;
        //发送的总数，当前为一百万条数据
        int total = 1000000;
        int j = 0;
        String[] values = new String[length];
        for (int i = 0; i < total; i++)
        {
            j = i % length;
            //赋值
            values[j] = "user_" + i;
            if (j == length - 1)
            {
                //发送到redis
                stringRedisTemplate.opsForHyperLogLog().add("UV", values);
            }
        }
        //发送完成，获得数据
        Long size = stringRedisTemplate.opsForHyperLogLog().size("UV");
        log.info("统计结果：" + size);
        //统计结果：997593
        //统计结果：1998502(两百万)
    }
```