

尚品汇商城

一、商品详情页面优化

1.1 思路

虽然咱们实现了页面需要的功能，但是考虑到该页面是被用户高频访问的，所以性能需要优化。

一般一个系统最大的性能瓶颈，就是数据库的 io 操作。从数据库入手也是调优性价比最高的切入点。

一般分为两个层面，一是提高数据库 sql 本身的性能，二是尽量避免直接查询数据库。

重点要讲的是另外一个层面：尽量避免直接查询数据库。

解决办法就是：**缓存**

1.2 整合 redis 到工程

由于 redis 作为缓存数据库，要被多个项目使用，所以要制作一个通用的工具类，方便工程中的各个模块使用。

而主要使用 redis 的模块，都是后台服务的模块，service 工程。所以咱们把 redis 的工具类放到 service-util 模块中，这样所有的后台服务模块都可以使用 redis。

1.2.1 首先在 service-util 引入依赖包

```
<!-- redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!-- spring2.X 集成 redis 所需 common-pool2 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
    <version>2.6.0</version>
</dependency>
```

1.2.2 添加 redis 配置类

数据结构	KEY或Value序列化设置	具体的序列化类	序列化前	序列化后
Key/Value	ValueSerializer	StringRedisSerializer	test_value111	test_value111
	ValueSerializer	Jackson2JsonRedisSerializer	test_value111	"test_value111"
	ValueSerializer	JdkSerializationRedisSerializer	test_value111	◆◆
Hash	HashValueSerializer	StringRedisSerializer	2016-08-18	2016-08-18
	HashValueSerializer	StringRedisSerializer	1	1
	HashValueSerializer	Jackson2JsonRedisSerializer	2016-08-18	"2016-08-18"
	HashValueSerializer	Jackson2JsonRedisSerializer	1	"1"
	HashValueSerializer	JdkSerializationRedisSerializer	2016-08-18	\xAC\xED\x00\x05t\x00\x04date
	HashValueSerializer	JdkSerializationRedisSerializer	1	\xAC\xED\x00\x05t\x00\x01

```
package com.atguigu.gmall.common.config

@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public RedisTemplate<Object, Object>
redisTemplate(RedisConnectionFactory redisConnectionFactory) {
        RedisTemplate<Object, Object> redisTemplate = new
RedisTemplate<>();
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer =
new Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);

        objectMapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);

        jackson2JsonRedisSerializer.setObjectMapper(objectMapper);
    }
}
```

```
// 序列号 key value
redisTemplate.setKeySerializer(new StringRedisSerializer());

redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);
redisTemplate.setHashKeySerializer(new
StringRedisSerializer());

redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);

redisTemplate.afterPropertiesSet();
return redisTemplate;
}
}
```

说明：由于 service-util 属于公共模块，所以我们把它引入到 service 父模块，其他 service 子模块都自动引入了

1.3 使用 redis 进行业务开发相关规则

开始开发先说明 redis key 的命名规范，由于 Redis 不像数据库表那样有结构，其所有的数据全靠 key 进行索引，所以 redis 数据的可读性，全依靠 key。

企业中最常用的方式就是：object:id:field

比如：sku:1314:info

user:1092:info

:表示根据 windows 的 / 一个意思

重构 getSkuInfo 方法

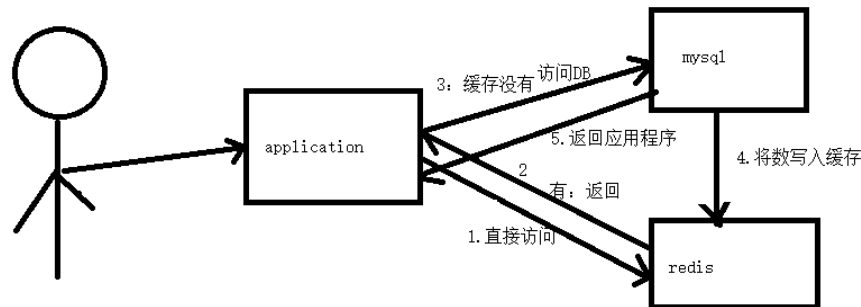
在 RedisConst 中定义 redis 的常量，RedisConst 类在 service-util 模块中，所有的 redis 常量我们都配置在这里

```
package com.atguigu.gmall.common.constant;

/**
 * Redis 常量配置类
 */
public class RedisConst {
```

```
public static final String SKUKEY_PREFIX = "sku:";
public static final String SKUKEY_SUFFIX = ":info";
//单位: 秒
public static final long SKUKEY_TIMEOUT = 24 * 60 * 60;
}
```

如何使用缓存:



1. 用户访问数据的时候, 如果缓存有数据, 则先访问缓存。
2. 如果缓存没有数据, 查询数据库, 并将数据访问缓存。
3. 应用程序再次访问的时候, 则直接走缓存, 不会走数据库了。

以上基本实现使用缓存的方案。

1.4 缓存常见问题

缓存最常见的 3 个问题: 面试

1. 缓存穿透
2. 缓存雪崩
3. 缓存击穿

缓存穿透: 是指查询一个不存在的数据, 由于缓存无法命中, 将去查询数据库, 但是数据库也无此记录, 并且出于容错考虑, 我们没有将这次查询的 null 写入缓存, 这将导致这个不存在的数据每次请求都要到存储层去查询, 失去了缓存的意义。在流量大时, 可能 DB 就挂掉了, 要是有人利用不存在的 key 频繁攻击我们的应用, 这就是漏洞。

解决: 空结果也进行缓存, 但它的过期时间会很短, 最长不超过五分钟。

缓存雪崩:是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。

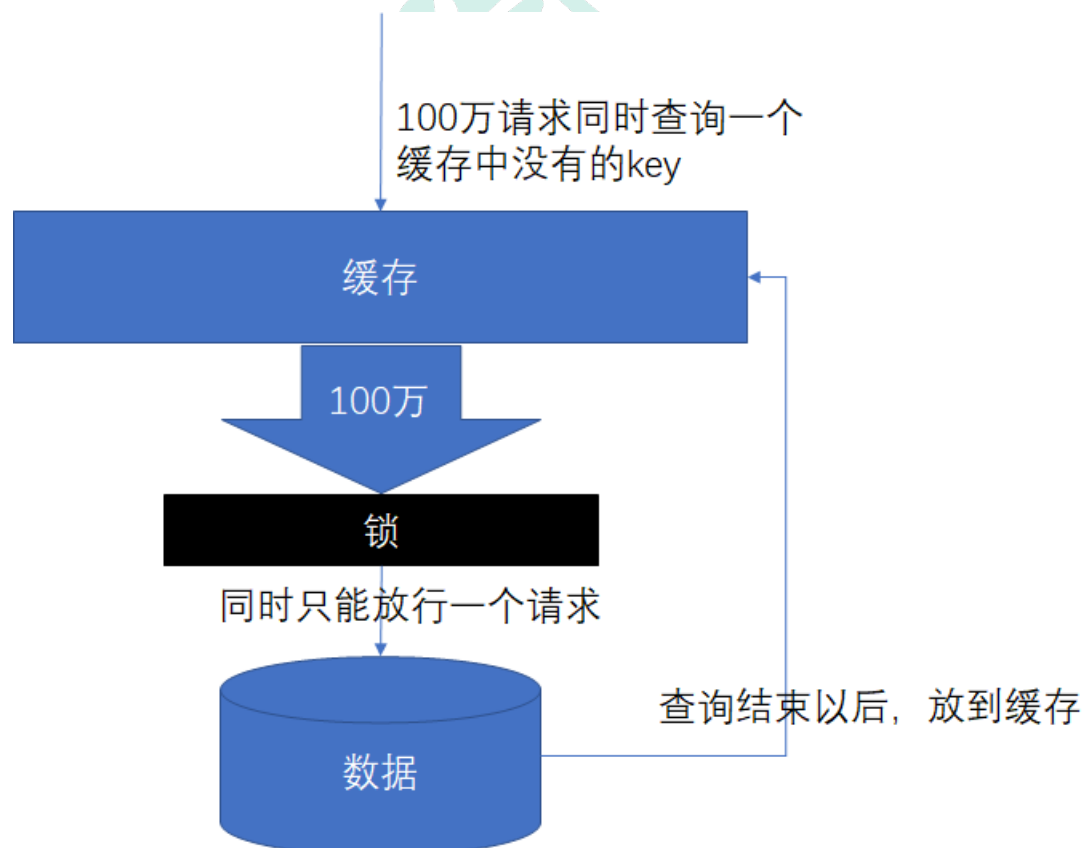
解决: 原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

缓存击穿: 是指对于一些设置了过期时间的 key，如果这些 key 可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个 key 在大量请求同时进来之前正好失效，那么所有对这个 key 的数据查询都落到 db，我们称为缓存击穿。

与缓存雪崩的区别：

1. 击穿是一个热点 key 失效
2. 雪崩是很多 key 集体失效

解决: 锁



二、分布式锁

2.1 本地锁的局限性

之前，我们学习过 `synchronized` 及 `lock` 锁，这些锁都是本地锁。接下来写一个案例，演示本地锁的问题

2.1.1 编写测试代码

在 `service-product` 中的 `TestController` 中添加测试方法

```
package com.atguigu.gmall.product.controller;

@Api(tags = "测试接口")
@RestController
@RequestMapping("admin/product/test")
public class TestController {

    @Autowired
    private TestService testService;

    @GetMapping("testLock")
    public Result testLock() {
        testService.testLock();
        return Result.ok();
    }
}
```

接口

```
package com.atguigu.gmall.product.service;

public interface TestService {

    void testLock();
}
```

```
}
```

实现类

```
package com.atguigu.gmall.product.service.impl;
@Service
public class TestServiceImpl implements TestService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    @Override
    public void testLock() {
        // 查询redis 中的num 值
        String value =
        (String)this.redisTemplate.opsForValue().get("num");
        // 没有该值 return
        if (StringUtils.isBlank(value)){
            return ;
        }
        // 有值就转成 int
        int num = Integer.parseInt(value);
        // 把redis 中的num 值+1
        this.redisTemplate.opsForValue().set("num",
        String.valueOf(++num));
    }
}
```

说明：通过 reids 客户端设置 num=0

2.1.2 使用 ab 工具测试

使用 ab 测试工具：httpd-tools (yum install -y httpd-tools)

```
ab -n (一次发送的请求数) -c (请求的并发数) 访问路径
```

测试如下：5000 请求，100 并发

```
ab -n 5000 -c 100 http://192.168.254.1:8206/admin/product/test/testLock
```

```
[root@localhost ~]# ab -n 5000 -c 100 http://192.168.200.1:8206/admin/product/test/testLock
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.200.1 (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests
```

查看 redis 中的值:



2.1.3 使用本地锁

```
@Override
public synchronized void testLock() {
    // 查询redis 中的num 值
    String value = (String)this.redisTemplate.opsForValue().get("num");
    // 没有该值return
    if (StringUtils.isBlank(value)){
        return ;
    }
    // 有值就转成int
    int num = Integer.parseInt(value);
    // 把redis 中的num 值+1
    this.redisTemplate.opsForValue().set("num",
    String.valueOf(++num));
}
```



```
}
```

使用 ab 工具压力测试：5000 次请求，并发 100

```
[root@localhost ~]# ab -n 5000 -c 100 http://192.168.200.1:8206/admin/product/test/testLock
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.200.1 (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests

Server Software:
Server Hostname:      192.168.200.1
```

查看 redis 中的结果：



完美！与预期一致，是否真的完美？

接下来再看集群情况下，会怎样？

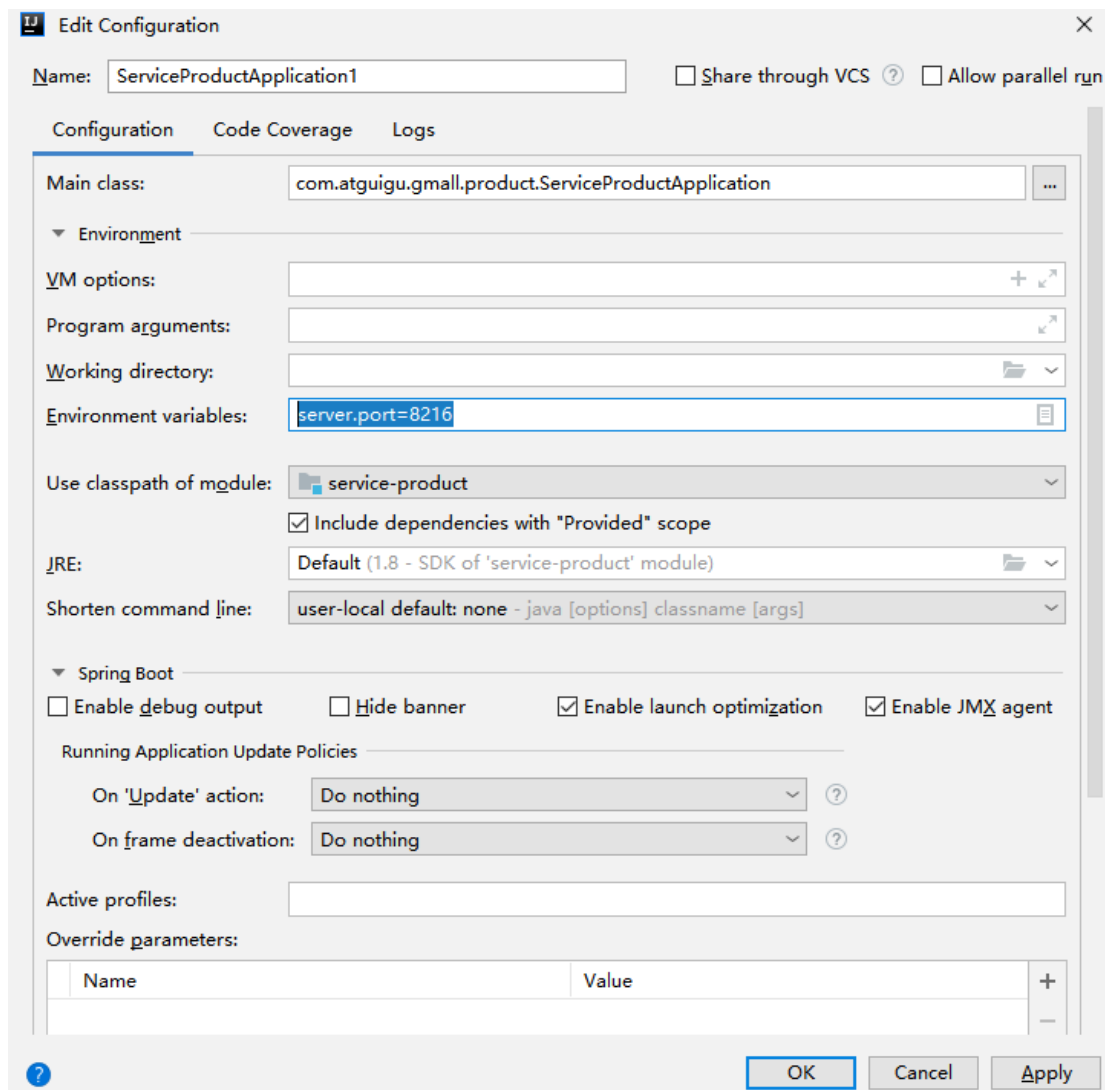
2.1.4 本地锁问题演示锁

接下来启动 8206 8216 8226 三个运行实例。

运行多个 service-product 实例：

server.port=8216

server.port=8226



注意：bootstrap.properties 添加一个 server.port = 8206; 将 nacos 的配置注释掉！

通过网关压力测试：

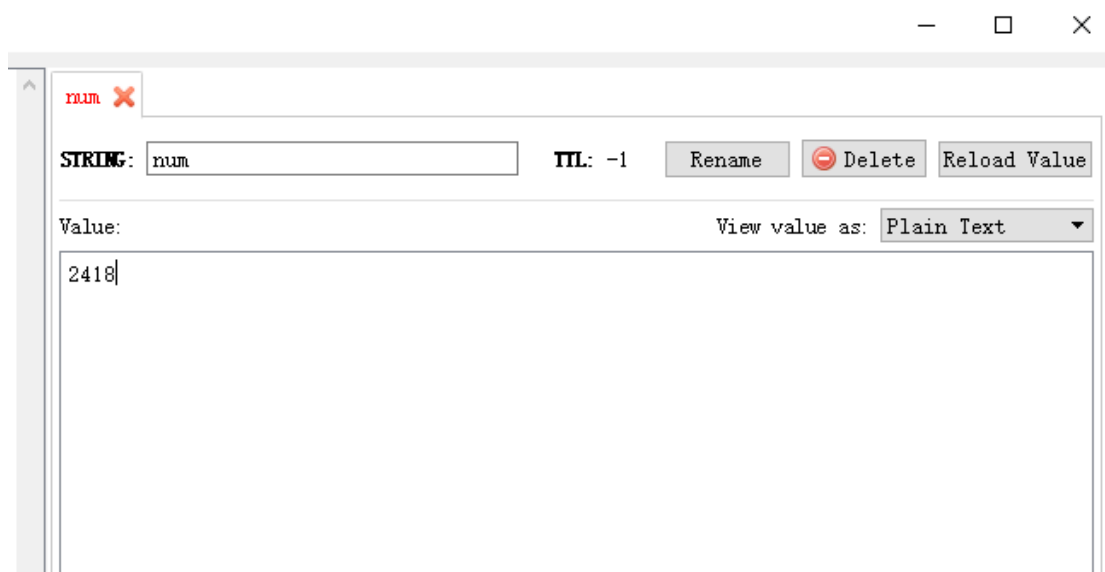
启动网关：

```
ab -n 5000 -c 100 http://192.168.200.1/admin/product/test/testLock
```

```
[root@localhost ~]# ab -n 5000 -c 100 http://192.168.200.1/admin/product/test/testLock
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.200.1 (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests
```

查看 redis 中的值:



集群情况下又出问题了!!!

以上测试, 可以发现:

本地锁只能锁住同一工程内的资源, 在分布式系统里面都存在局限性。

此时需要分布式锁。。

2.2 分布式锁实现的解决方案

随着业务发展的需要, 原单体单机部署的系统被演化成分布式集群系统后, 由于分布式系统多线程、多进程并且分布在不同机器上, 这将使原单机部署情况下的并发控制锁策略失效, 单纯的 Java API 并不能提供分布式锁的能力。为了解决这个问题就需要一种跨 JVM 的互斥机制来控制共享资源的访问, 这就是分布式锁要解决的问题!

分布式锁主流的实现方案：

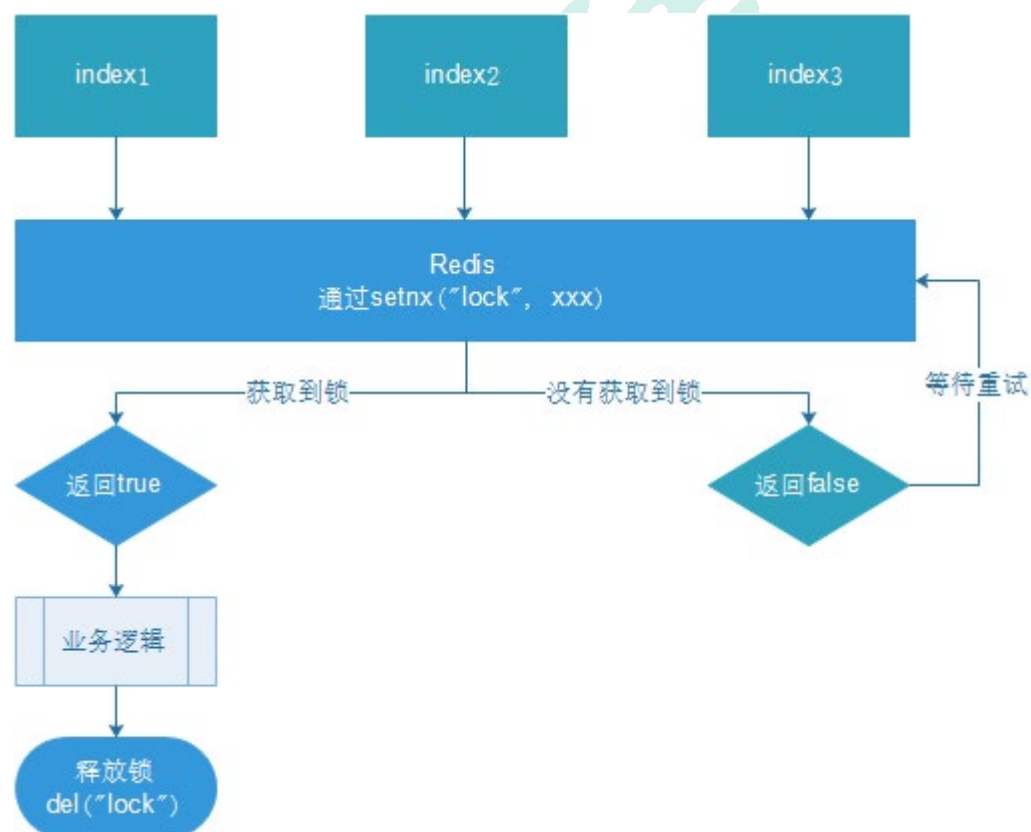
1. 基于数据库实现分布式锁
2. 基于缓存（Redis 等）
3. 基于 Zookeeper

每一种分布式锁解决方案都有各自的优缺点：

1. 性能：redis 最高
2. 可靠性：zookeeper 最高

这里，我们就基于 redis 实现分布式锁。

2.3 使用 redis 实现分布式锁



1. 多个客户端同时获取锁（setnx）
2. 获取成功，执行业务逻辑{从 db 获取数据，放入缓存}，执行完成释放锁（del）

3. 其他客户端等待重试

2.3.1 编写代码

```
@Override
public void testLock() {
    // 1. 从redis 中获取锁, setnx
    Boolean lock =
    this.redisTemplate.opsForValue().setIfAbsent("lock", "111");
    if (lock) {
        // 查询redis 中的num 值
        String value =
        (String) this.redisTemplate.opsForValue().get("num");
        // 没有该值 return
        if (StringUtils.isBlank(value)){
            return ;
        }
        // 有值就转成 int
        int num = Integer.parseInt(value);
        // 把redis 中的num 值+1
        this.redisTemplate.opsForValue().set("num",
        String.valueOf(++num));

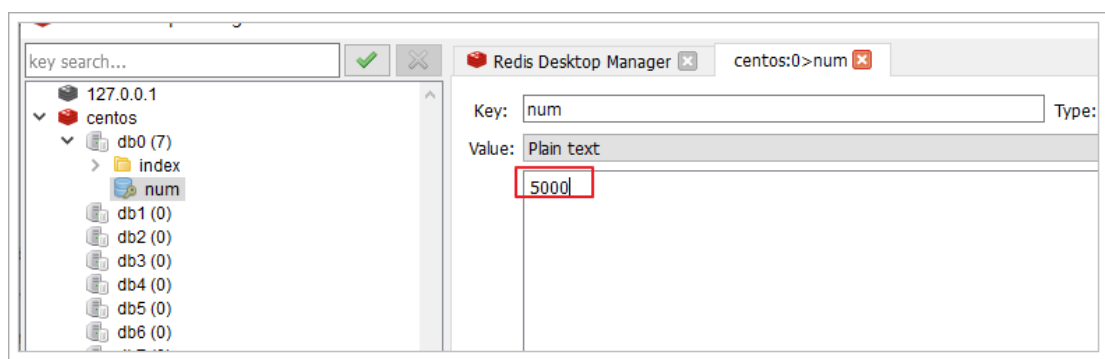
        // 2. 释放锁 del
        this.redisTemplate.delete("lock");
    } else {
        // 3. 每隔1 秒钟回调一次，再次尝试获取锁
        try {
            Thread.sleep(100);
            testLock();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

重启，服务集群，通过网关压力测试：

```
[root@localhost ~]# [root@localhost ~]# ab -n 5000 -c 100 http://192.168.200.1/admin/product/test/testLock
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.200.1 (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests
```

查看 redis 中 num 的值：



基本实现。

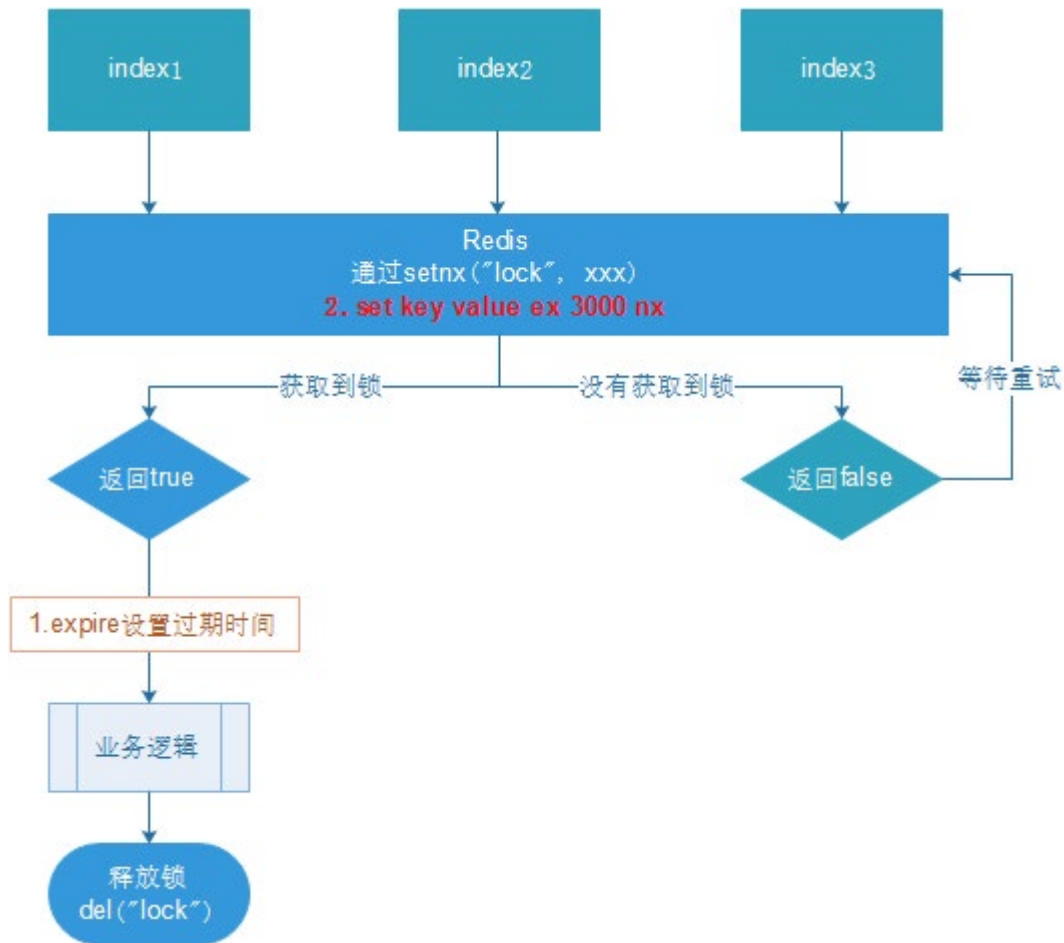
问题：setnx 刚好获取到锁，业务逻辑出现异常，导致锁无法释放

解决：设置过期时间，自动释放锁。

2.3.2 优化之设置锁的过期时间

设置过期时间有两种方式：

1. 首先想到通过 expire 设置过期时间（缺乏原子性：如果在 setnx 和 expire 之间出现异常，锁也无法释放）
2. 在 set 时指定过期时间（推荐）



设置过期时间：

```

54      @Override
55      public void testLock() {
56          // 1. 从redis中获取锁, setnx
57          Boolean lock = this.redisTemplate.opsForValue()
58              .setIfAbsent(key: "lock", value: "111", timeout: 3, TimeUnit.SECONDS);
59          if (lock) {
60              // 查询redis中的num值
61              String value = this.redisTemplate.opsForValue().get("num");
62              // 没有该值return
63              if (StringUtils.isBlank(value)) {
64                  return ;
65              }
66              // 有值就转成int
    
```

压力测试肯定也没有问题。自行测试

问题：可能会释放其他服务器的锁。

场景：如果业务逻辑的执行时间是 7s。执行流程如下

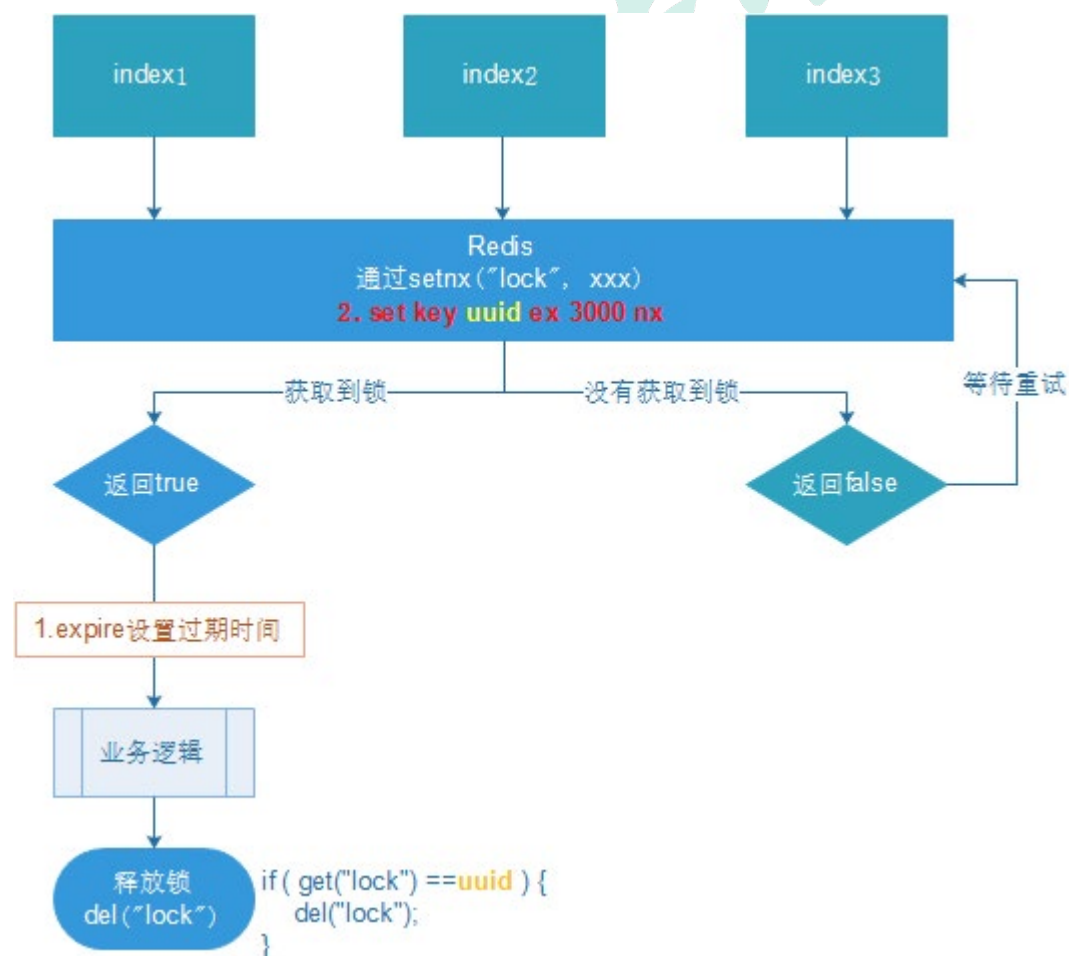
1. index1 业务逻辑没执行完，3 秒后锁被自动释放。

2. index2 获取到锁，执行业务逻辑，3 秒后锁被自动释放。
3. index3 获取到锁，执行业务逻辑
4. index1 业务逻辑执行完成，开始调用 del 释放锁，这时释放的是 index3 的锁，导致 index3 的业务只执行 1s 就被别人释放。

最终等于没锁的情况。

解决：setnx 获取锁时，设置一个指定的唯一值（例如：uuid）；释放前获取这个值，判断是否自己的锁

2.3.3 优化之 UUID 防误删




```
public void testLock() {
    // 1. 从redis中获取锁.setnx
    String uuid = UUID.randomUUID().toString();
    Boolean lock = this.redisTemplate.opsForValue()
        .setIfAbsent(k: "lock", uuid, l: 2, TimeUnit.SECONDS);
    if (lock) {
        // 查询redis中的num值
        String value = this.redisTemplate.opsForValue().get("num");
        // 没有该值return
        if (StringUtils.isBlank(value)){
            return ;
        }
        // 有值就转成int
        int num = Integer.parseInt(value);
        // 把redis中的num值+1
        this.redisTemplate.opsForValue().set("num", String.valueOf(++num));

        if(uuid.equals((String)redisTemplate.opsForValue().get("lock"))){
            this.redisTemplate.delete(key: "lock");
        }
    }
}
```

问题：删除操作缺乏原子性。

场景：

1. index1 执行删除时，查询到的 lock 值确实和 uuid 相等

```
if(uuid.equals((String)redisTemplate.opsForValue().get("lock")))
```

2. index1 执行删除前，lock 刚好过期时间已到，被 redis 自动释放
在 redis 中没有了锁。

```
this.redisTemplate.delete(key: "lock");
```

3. index2 获取了 lock,index2 线程获取到了 cpu 的资源，开始执行方法
4. index1 执行删除，此时会把 index2 的 lock 删除

index1 因为已经在方法中了，所以不需要重新上锁。index1 有执行的权限。index1 已经比较完成了，这个时候，开始执行

```
this.redisTemplate.delete(key: "lock");
```

删除的 index2 的锁!

2.3.4 优化之 LUA 脚本保证删除的原子性

```
@Override
public void testLock() {
    // 设置 uuid
    String uuid = UUID.randomUUID().toString();
    // 缓存的 Lock 对应的值，应该是 index2 的 uuid
    Boolean flag = redisTemplate.opsForValue().setIfAbsent("lock",
    uuid, 1, TimeUnit.SECONDS);
    // 判断 flag index=1
    if (flag){
        // 说明上锁成功！执行业务逻辑
        String value = redisTemplate.opsForValue().get("num");
        // 判断
        if(StringUtils.isEmpty(value)){
            return;
        }
        // 进行数据转换
        int num = Integer.parseInt(value);
        // 放入缓存
        redisTemplate.opsForValue().set("num",String.valueOf(++num));

        // 定义一个 Lua 脚本
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then
        return redis.call('del', KEYS[1]) else return 0 end";

        // 准备执行 Lua 脚本
        DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
        // 将 Lua 脚本放入 DefaultRedisScript 对象中
        redisScript.setScriptText(script);
        // 设置 DefaultRedisScript 这个对象的泛型
        redisScript.setResultType(Long.class);
        // 执行删除
        redisTemplate.execute(redisScript, Arrays.asList("lock"),uuid);

    }else {
        // 没有获取到锁！
        try {
            Thread.sleep(1000);
            // 睡醒了之后，重试
            testLock();
        } catch (InterruptedException e) {
```

```
        e.printStackTrace();
    }
}
}
```

Lua 脚本详解: <http://doc.redisfans.com/string/set.html>

客户端执行以上的命令:

- 如果服务器返回 OK, 那么这个客户端获得锁。
- 如果服务器返回 NIL, 那么客户端获取锁失败, 可以在稍后再重试。

设置的过期时间到达之后, 锁将自动释放。

可以通过以下修改, 让这个锁实现更健壮:

- 不使用固定的字符串作为键的值, 而是设置一个不可猜测 (non-guessable) 的长随机字符串, 作为口令串 (token)。
- 不使用 DEL 命令来释放锁, 而是发送一个 Lua 脚本, 这个脚本只在客户端传入的值和键的口令串相匹配时, 才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例:

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

2.3.5 总结

1、加锁

```
// 1. 从redis 中获取锁,set k1 v1 px 20000 nx
String uuid = UUID.randomUUID().toString();
Boolean lock = this.redisTemplate.opsForValue()
    .setIfAbsent("lock", uuid, 2, TimeUnit.SECONDS);
```

2、使用 lua 释放锁

```
// 2. 释放锁 del
String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
// 设置 Lua 脚本返回的数据类型
DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
// 设置 Lua 脚本返回类型为 Long
redisScript.setResultType(Long.class);
redisScript.setScriptText(script);
redisTemplate.execute(redisScript, Arrays.asList("lock"),uuid);
```

3、重试

```
Thread.sleep(500);  
testLock();
```

为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁。
- 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。
- 加锁和解锁必须具有原子性

redis 集群状态下的问题：

1. 客户端 A 从 master 获取到锁
2. 在 master 将锁同步到 slave 之前，master 宕掉了。
3. slave 节点被晋级为 master 节点
4. 客户端 B 取得了同一个资源被客户端 A 已经获取到的另外一个锁。

安全失效！

解决方案：了解即可！

redlock

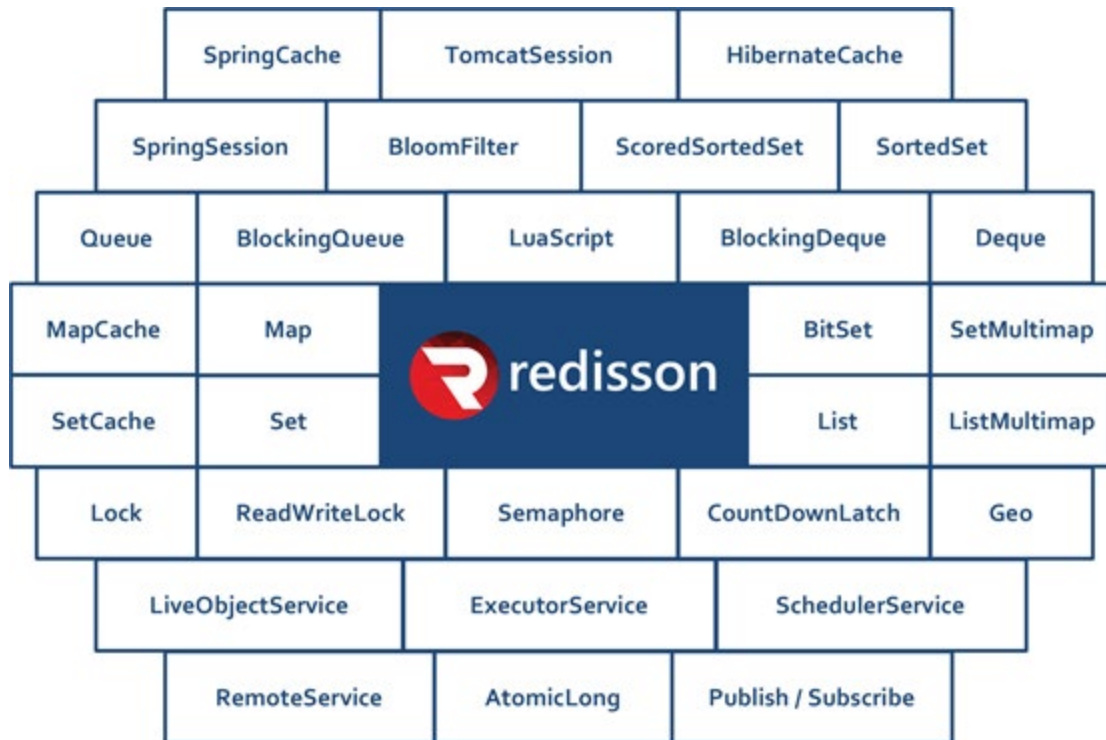
为了解决故障转移情况下的缺陷，Antirez 发明了 Redlock 算法，使用 redlock 算法，需要多个 redis 实例，加锁的时候，它会想多半节点发送 `setex mykey myvalue` 命令，只要过半节点成功了，那么就算加锁成功了。释放锁的时候需要向所有节点发送 del 命令。这是一种基于【大多数都同意】的一种机制。感兴趣的可以查询相关资料。在实际工作中使用的时候，我们可以选择已有的开源实现，python 有 redlock-py，java 中有 Redisson redlock。

redlock 确实解决了上面所说的“不靠谱的情况”。但是，它解决问题的同时，也带来了代价。你需要多个 redis 实例，你需要引入新的库 代码也得调整，性能上也会有损。所以，果然是不存在“完美的解决方案”，我们更需要的是能够根据实际情况和条件把问题解决了就好。

2.4 使用 redisson 解决分布式锁

Github 地址：<https://github.com/redisson/redisson>

Redisson 是一个在 **Redis** 的基础上实现的 **Java 驻内存数据网格** (In-Memory Data Grid)。它不仅提供了一系列的分布式的 Java 常用对象，还提供了许多分布式服务。其中包括(BitSet, Set, Multimap, SortedSet, Map, List, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, AtomicLong, CountDownLatch, Publish / Subscribe, Bloom filter, Remote service, Spring cache, Executor service, Live Object service, Scheduler service) Redisson 提供了使用 Redis 的最简单和最便捷的方法。Redisson 的宗旨是促进使用者对 Redis 的关注分离 (Separation of Concern)，从而让使用者能够将精力更集中地放在处理业务逻辑上。



官方文档地址: <https://github.com/redisson/redisson/wiki>

2.4.1 实现代码

1. 导入依赖 service-util

```
<!-- redisson -->
<dependency>
  <groupId>org.redisson</groupId>
  <artifactId>redisson</artifactId>
  <version>3.15.3</version>
</dependency>
```

配置 redisson

```
package com.atguigu.gmall.common.config;
```

```
@Data
@Configuration
@ConfigurationProperties("spring.redis")
public class RedissonConfig {

    private String host;

    private String password;

    private String port;

    private int timeout = 3000;
    private static String ADDRESS_PREFIX = "redis://";

    /**
     * 自动装配
     */
    @Bean
    RedissonClient redissonSingle() {
        Config config = new Config();

        if(StringUtils.isEmpty(host)){
            throw new RuntimeException("host is empty");
        }
        SingleServerConfig serverConfig = config.useSingleServer()
            .setAddress(ADDRESS_PREFIX + this.host + ":" + port)
            .setTimeout(this.timeout);
        if(!StringUtils.isEmpty(this.password)) {
            serverConfig.setPassword(this.password);
        }
        return Redisson.create(config);
    }
}
```

2. 修改实现类

```
@Autowired
private RedissonClient redissonClient;

@Override
public void testLock() {
    // 创建锁:
    String skuId="25";
    String lockKey ="lock:"+skuId;
    // 锁的是每个商品
    RLock lock = redissonClient.getLock(lockKey);
    // 开始加锁
    lock.lock();
    // 业务逻辑代码
    // 获取数据
    String value = redisTemplate.opsForValue().get("num");
    if (StringUtils.isBlank(value)){
```

```
        return;
    }
    // 将value 变为int
    int num = Integer.parseInt(value);
    // 将num +1 放入缓存
    redisTemplate.opsForValue().set("num",String.valueOf(++num));
    // 解锁:
    lock.unlock();
}
```

2.4.2 可重入锁 (Reentrant Lock)

基于 Redis 的 Redisson 分布式可重入锁 RLock Java 对象实现了 `java.util.concurrent.locks.Lock` 接口。

如果拿到分布式锁的节点宕机，且这个锁正好处于锁住的状态时，会出现锁死的状态，为了避免这种情况的发生，锁都会设置一个过期时间。这样也存在一个问题，假如一个线程拿到了锁设置了 30s 超时，在 30s 后这个线程还没有执行完毕，锁超时释放了，就会导致问题，Redisson 给出了自己的答案，就是 watch dog 自动延期机制。

Redisson 提供了一个监控锁的看门狗，它的作用是在 Redisson 实例被关闭前，不断的延长锁的有效期，也就是说，如果一个拿到锁的线程一直没有完成逻辑，那么看门狗会帮助线程不断的延长锁超时时间，锁不会因为超时而被释放。

默认情况下，看门狗的续期时间是 30s，也可以通过修改 `Config.lockWatchdogTimeout` 来另行指定。另外 Redisson 还提供了可以指定 `leaseTime` 参数的加锁方法来指定加锁的时间。超过这个时间后锁便自动解开了，不会延长锁的有效期

快速入门使用的就是可重入锁。也是最常使用的锁。

最常见的使用：

```
RLock lock = redisson.getLock("anyLock");

// 最常使用
lock.lock();

// 加锁以后 10 秒钟自动解锁
// 无需调用 unlock 方法手动解锁
lock.lock(10, TimeUnit.SECONDS);

// 尝试加锁，最多等待 100 秒，上锁以后 10 秒自动解锁
boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
if (res) {
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

改造程序：

```
63      @Override
64      public void testLock() {
65
66          RLock lock = this.redissonClient.getLock(s: "lock"); // 只要锁的名称相同就是同一把锁
67          lock.lock(l: 10, TimeUnit.SECONDS); // 加锁
68
69          // 查询redis中的num值
70          String value = this.redisTemplate.opsForValue().get("num");
71          // 没有该值return
72          if (StringUtils.isBlank(value)) {...}
73          // 有值就转成int
74          int num = Integer.parseInt(value);
75          // 把redis中的num值+1
76          this.redisTemplate.opsForValue().set("num", String.valueOf(++num));
77
78          //lock.unlock(); // 解锁      10秒之后自动释放，也可以手动释放
79
80      }
```

重启后在浏览器测试：

2.4.3 读写锁 (ReadWriteLock)

基于 Redis 的 Redisson 分布式可重入读写锁 RReadWriteLock Java 对象实现了 `java.util.concurrent.locks.ReadWriteLock` 接口。其中读锁和写锁都继承了 `RLock` 接口。

分布式可重入读写锁允许同时有多个读锁和一个写锁处于加锁状态。

```
RReadWriteLock rwlock = redisson.getReadWriteLock("anyRWLock");

// 最常见的使用方法
rwlock.readLock().lock();

// 或
rwlock.writeLock().lock();

// 10 秒钟以后自动解锁
// 无需调用 unlock 方法手动解锁
rwlock.readLock().lock(10, TimeUnit.SECONDS);

// 或
rwlock.writeLock().lock(10, TimeUnit.SECONDS);

// 尝试加锁，最多等待 100 秒，上锁以后 10 秒自动解锁
boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);

// 或
boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);

...

lock.unlock();
```

代码实现

```
TestController
```

```
@GetMapping("read")
public Result<String> read(){
    String msg = testService.readLock();

    return Result.ok(msg);
}

@GetMapping("write")
public Result<String> write(){
    String msg = testService.writeLock();

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

TestService 接口

```
String readLock();

String writeLock();
```

实现类

读锁，写锁要想达到互斥效果，那么锁的 key，必须是同一把 **readwriteLock**

```
@Override
public String readLock() {
    // 初始化读写锁
    RReadWriteLock readWriteLock =
    redissonClient.getReadWriteLock("readwriteLock");
    RLock rLock = readWriteLock.readLock(); // 获取读锁

    rLock.lock(10, TimeUnit.SECONDS); // 加10s 锁

    String msg = this.redisTemplate.opsForValue().get("msg");

    //rLock.unlock(); // 解锁
    return msg;
}

@Override
public String writeLock() {
    // 初始化读写锁
    RReadWriteLock readWriteLock =
    redissonClient.getReadWriteLock("readwriteLock");
    RLock rLock = readWriteLock.writeLock(); // 获取写锁
```

```
rLock.lock(10, TimeUnit.SECONDS); // 加10s 锁

this.redisTemplate.opsForValue().set("msg",
UUID.randomUUID().toString());

//rLock.unlock(); // 解锁
return "成功写入了内容。。。。。。";
}
```

打开两个浏览器窗口测试：

http://localhost:8206/admin/product/test/read

http://localhost:8206/admin/product/test/write

- 同时访问写：一个写完之后，等待一会儿（约 10s），另一个写开始
- 同时访问读：不用等待
- 先写后读：读要等待（约 10s）写完成
- 先读后写：写要等待（约 10s）读完成

三、分布式锁改造获取 sku 信息

3.1 使用 redis

RedisConst 类中追加一个变量

```
// 商品如果在数据库中不存在那么会缓存一个空对象进去，但是这个对象是没有用的，
// 所以这个对象的过期时间应该不能太长，
// 如果太长会占用内存。
// 定义变量，记录空对象的缓存过期时间
public static final long SKUKEY_TEMPORARY_TIMEOUT = 10 * 60;
```

在实现类中引入

@Autowired

private RedisTemplate redisTemplate;

// 使用redis' 做分布式锁

private SkuInfo getSkuInfoRedis(Long skuId) {

SkuInfo skuInfo = null;

try {

// 缓存存储数据: key-value

// 定义key sku:skuId:info

String

skuKey

=

RedisConst.SKUKEY_PREFIX+skuId+RedisConst.SKUKEY_SUFFIX;

// 获取里面的数据? redis 有五种数据类型 那么我们存储商品详情 使用哪种数据类型?

// 获取缓存数据

skuInfo = (SkuInfo) redisTemplate.opsForValue().get(skuKey);

// 如果从缓存中获取的数据是空

if (skuInfo==null){

// 直接获取数据库中的数据, 可能会造成缓存击穿。所以在这个位置, 应该添加锁。

// 第一种: redis , 第二种: redisson

// 定义锁的key sku:skuId:Lock set k1 v1 px 10000 nx

String

lockKey

=

RedisConst.SKUKEY_PREFIX+skuId+RedisConst.SKULOCK_SUFFIX;

// 定义锁的值

String uuid = UUID.randomUUID().toString().replace("-", "");

// 上锁

Boolean isExist = redisTemplate.opsForValue().setIfAbsent(lockKey, uuid, RedisConst.SKULOCK_EXPIRE_PX2, TimeUnit.SECONDS);

if (isExist){

// 执行成功的话, 则上锁。

System.out.println("获取到分布式锁!");

// 真正获取数据库中的数据 { 数据库中到底有没有这个数据 = 防止缓存穿透}

skuInfo = getSkuInfoDB(skuId);

// 从数据库中获取的数据就是空

if (skuInfo==null){

// 为了避免缓存穿透 应该给空的对象放入缓存

SkuInfo skuInfo1 = new SkuInfo(); //对象的地址

redisTemplate.opsForValue().set(skuKey, skuInfo1, RedisConst.SKUKEY_TEMPORARY_TIMEOUT, TimeUnit.SECONDS);

return skuInfo1;

}

// 查询数据库的时候, 有值

redisTemplate.opsForValue().set(skuKey, skuInfo, RedisConst.SKUKEY_TIMEOUT, TimeUnit.SECONDS);

// 解锁: 使用 Lua 脚本解锁

String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

// 设置 Lua 脚本返回的数据类型

DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();

```
// 设置 Lua 脚本返回类型为 Long
redisScript.setResultType(Long.class);
redisScript.setScriptText(script);
// 删除 key 所对应的 value
redisTemplate.execute(redisScript,
Arrays.asList(lockKey), uuid);

        return skuInfo;
    }else {
        // 其他线程等待
        Thread.sleep(1000);
        return getSkuInfo(skuId);
    }
}
}
}
return skuInfo;
}
} catch (InterruptedException e) {
    e.printStackTrace();
}
// 为了防止缓存宕机：从数据库中获取数据
return getSkuInfoDB(skuId);
}
```

```
@Override
public SkuInfo getSkuInfoDB(Long skuId) {

    SkuInfo skuInfo = skuInfoMapper.selectById(skuId);
    if (skuInfo!=null){
        QueryWrapper<SkuImage> skuImageQueryWrapper = new
        QueryWrapper<>();
        skuImageQueryWrapper.eq("sku_id", skuId);
        List<SkuImage> skuImageList =
        skuImageMapper.selectList(skuImageQueryWrapper);
        skuInfo.setSkuImageList(skuImageList);
    }

    return skuInfo;
}
```

3.2 使用 redisson

在实现类添加

```
@Autowired
private RedissonClient redissonClient;

private SkuInfo getSkuInfoRedisson(Long skuId) {
```

```
SkuInfo skuInfo = null;
try {
    // 缓存存储数据: key-value
    // 定义key sku:skuId:info
    String skuKey =
RedisConst.SKUKEY_PREFIX+skuId+RedisConst.SKUKEY_SUFFIX;
    // 获取里面的数据? redis 有五种数据类型 那么我们存储商品详情 使用哪种
    // 数据类型?
    // 获取缓存数据
    skuInfo = (SkuInfo) redisTemplate.opsForValue().get(skuKey);
    // 如果从缓存中获取的数据是空
    if (skuInfo==null){
        // 直接获取数据库中的数据, 可能会造成缓存击穿。所以在这个位置, 应
        // 该添加锁。
        // 第二种: redisson
        // 定义锁的key sku:skuId:lock set k1 v1 px 10000 nx
        String lockKey =
RedisConst.SKUKEY_PREFIX+skuId+RedisConst.SKULOCK_SUFFIX;
        RLock lock = redissonClient.getLock(lockKey);
        /*
        第一种: lock.lock();
        第二种: lock.lock(10,TimeUnit.SECONDS);
        第三种: lock.tryLock(100,10,TimeUnit.SECONDS);
        */
        // 尝试加锁
        boolean res = lock.tryLock(RedisConst.SKULOCK_EXPIRE_PX1,
RedisConst.SKULOCK_EXPIRE_PX2, TimeUnit.SECONDS);
        if (res){
            try {
                // 处理业务逻辑 获取数据库中的数据
                // 真正获取数据库中的数据 {数据库中到底有没有这个数据 = 防止缓存穿
                // 透}
                skuInfo = getSkuInfoDB(skuId);
                // 从数据库中获取的数据就是空
                if (skuInfo==null){
                    // 为了避免缓存穿透 应该给空的对象放入缓存
                    SkuInfo skuInfo1 = new SkuInfo(); //对象的地址
                    redisTemplate.opsForValue().set(skuKey,skuInfo1,RedisConst.SKUKEY_TEMPO
RARY_TIMEOUT,TimeUnit.SECONDS);
                    return skuInfo1;
                }
                // 查询数据库的时候, 有值
                redisTemplate.opsForValue().set(skuKey,skuInfo,RedisConst.SKUKEY_TIMEOU
T,TimeUnit.SECONDS);

                // 使用 redis 用的是 lua 脚本删除 , 但是现在用么?
                lock.unlock();

                return skuInfo;
            }catch (Exception e){
                e.printStackTrace();
            }finally {
```

```
        // 解锁:
        lock.unlock();
    }
    }else {
        // 其他线程等待
        Thread.sleep(1000);
        return getSkuInfo(skuId);
    }
    }else {

        return skuInfo;
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
// 为了防止缓存宕机: 从数据库中获取数据
return getSkuInfoDB(skuId);
}
```

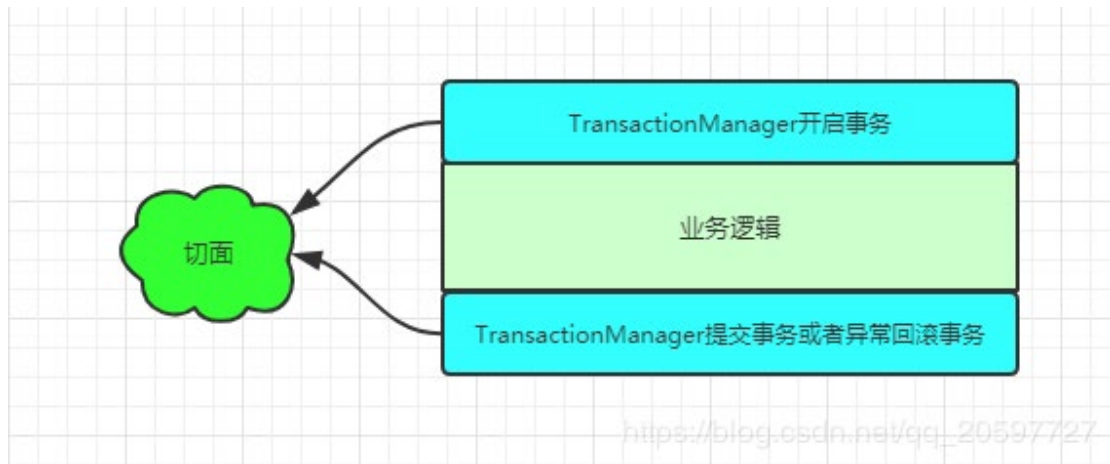
3.3 在 getSkuInfo 中调用上述两个方法进行测试

```
@Override
public SkuInfo getSkuInfo(Long skuId) {
    // 使用框架redisson 解决分布式锁!
    return getSkuInfoRedisson(skuId);

    // return getSkuInfoRedis(skuId);
}
```

四、分布式锁 + AOP 实现缓存

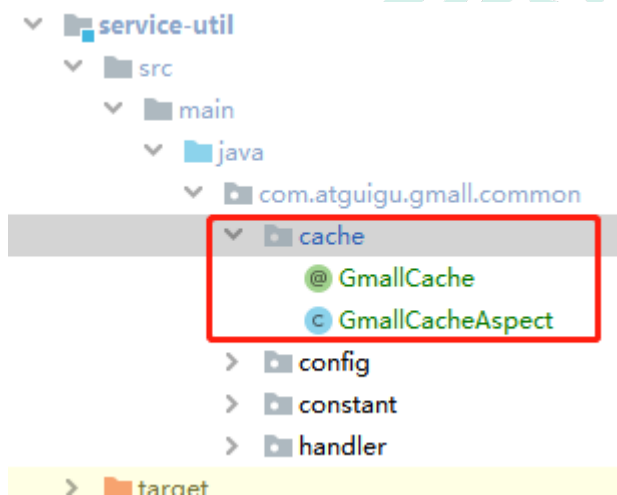
随着业务中缓存及分布式锁的加入，业务代码变的复杂起来，除了需要考虑业务逻辑本身，还要考虑缓存及分布式锁的问题，增加了程序员的工作量及开发难度。而缓存的玩法套路特别类似于事务，而声明式事务就是用了 aop 的思想实现的。



1. 以 `@Transactional` 注解为植入点的切点，这样才能知道 `@Transactional` 注解标注的方法需要被代理。
2. `@Transactional` 注解的切面逻辑类似于 `@Around`

模拟事务，缓存可以这样实现：

1. 自定义缓存注解 `@GmallCache`（类似于事务 `@Transactional`）
2. 编写切面类，使用环绕通知实现缓存的逻辑封装



4.1 定义一个注解

```
package com.atguigu.gmall.common.cache;

import java.lang.annotation.*;
```

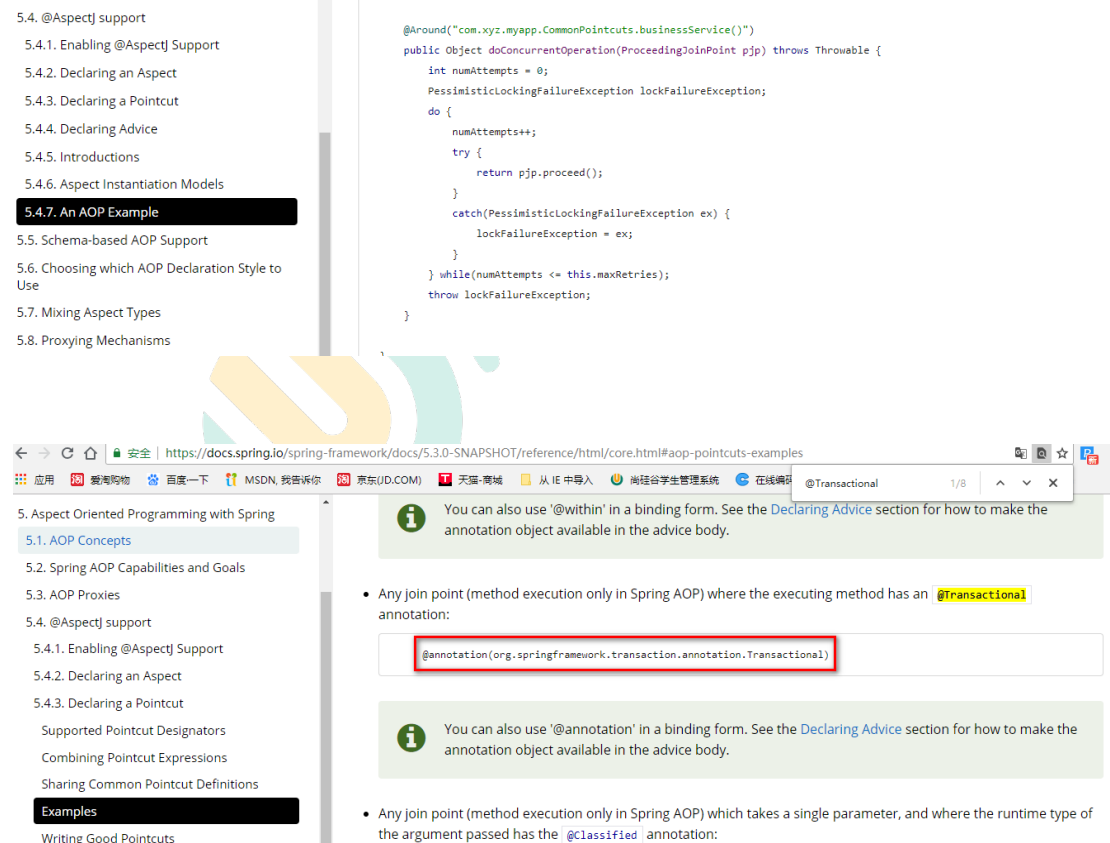


```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface GmallCache {
    /**
     * 缓存key的前缀
     * @return
     */
    String prefix() default "cache";
}
```

4.2 定义一个切面类加上注解

Spring aop 参考文档:

<https://docs.spring.io/spring-framework/docs/5.3.9-APSHOT/reference/html/core.html#aop>



The screenshot shows the Spring Framework documentation for AOP Pointcuts. The left sidebar lists the table of contents, with '5.4.7. An AOP Example' highlighted. The main content area shows a code example for a custom annotation. The code is as follows:

```
@Around("com.xyz.myapp.CommonPointcuts.businessService()")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    int numAttempts = 0;
    PessimisticLockingFailureException lockFailureException;
    do {
        numAttempts++;
        try {
            return pjp.proceed();
        }
        catch (PessimisticLockingFailureException ex) {
            lockFailureException = ex;
        }
    } while (numAttempts <= this.maxRetries);
    throw lockFailureException;
}
```

Below the code, there are two informational boxes. The first box states: 'You can also use '@within' in a binding form. See the Declaring Advice section for how to make the annotation object available in the advice body.' The second box states: 'You can also use '@annotation' in a binding form. See the Declaring Advice section for how to make the annotation object available in the advice body.'

Below these boxes, there are two bullet points. The first bullet point states: 'Any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:'. The second bullet point states: 'Any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:'.

```
package com.atguigu.gmall.common.cache;

import com.alibaba.fastjson.JSON;
import com.atguigu.gmall.common.constant.RedisConst;
import lombok.SneakyThrows;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import java.util.Arrays;
import java.util.concurrent.TimeUnit;

/**
 * @author atguigu-mqx
 * 处理环绕通知
 */
@Component
@Aspect
public class GmallCacheAspect {

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private RedissonClient redissonClient;

    // 切GmallCache 注解
    @SneakyThrows
    @Around("@annotation(com.atguigu.gmall.common.cache.GmallCache)")
    public Object cacheAroundAdvice(ProceedingJoinPoint joinPoint){
        // 声明一个对象
        Object object = new Object();
        // 在环绕通知中处理业务逻辑 {实现分布式锁}
        // 获取到注解，注解使用在方法上！
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        GmallCache gmallCache = signature.getMethod().getAnnotation(GmallCache.class);

        // 获取到注解上的前缀
        String prefix = gmallCache.prefix(); // sku

        // 方法传入的参数
        Object[] args = joinPoint.getArgs();

        // 组成缓存的key 需要前缀+方法传入的参数
        String key = prefix+ Arrays.asList(args).toString();

        // 防止 redis , redisson 出现问题！
        try {
            // 从缓存中获取数据
            // 类似于 skuInfo = (SkuInfo)
            redisTemplate.opsForValue().get(skuKey);
            object = cacheHit(key,signature);
            // 判断缓存中的数据是否为空！
            if (object==null){
                // 从数据库中获取数据，并放入缓存，防止缓存击穿必须上锁
                // prefix = sku index1 skuId = 32 , index2 skuId = 33
            }
        }
    }
}
```

```
// public SkuInfo getSkuInfo(Long skuId)
// key+":lock"
String lockKey = prefix + ":lock";
// 准备上锁
RLock lock = redissonClient.getLock(lockKey);
boolean result =
lock.tryLock(RedisConst.SKULOCK_EXPIRE_PX1, RedisConst.SKULOCK_EXPIRE_PX2,
TimeUnit.SECONDS);
// 上锁成功
if (result){
    try {
        // 表示执行方法体 getSkuInfoDB(skuId);
        object = joinPoint.proceed(joinPoint.getArgs());
        // 判断object 是否为空
        if (object==null){
            // 防止缓存穿透
            Object object1 = new Object();
            redisTemplate.opsForValue().set(key,
JSON.toJSONString(object1),RedisConst.SKUKEY_TEMPORARY_TIMEOUT,TimeUnit.SEC
ONDS);

            // 返回数据
            return object1;
        }
        // 放入缓存
        redisTemplate.opsForValue().set(key,
JSON.toJSONString(object),RedisConst.SKUKEY_TIMEOUT,TimeUnit.SECONDS);

        // 返回数据
        return object;
    } finally {
        lock.unlock();
    }
}else{
    // 上锁失败,睡眠自旋
    Thread.sleep(1000);
    return cacheAroundAdvice(joinPoint);
    // 理想状态
    // return object;
}return cacheHit(key, signature);
}
}else {

} catch (Throwable throwable) {
    throwable.printStackTrace();
}
// 如果出现问题数据库兜底
return joinPoint.proceed(joinPoint.getArgs());
}
/**
 * 表示从缓存中获取数据
 * @param key 缓存的key
 * @param signature 获取方法的返回值类型
 * @return
 */
private Object cacheHit(String key, MethodSignature signature) {
    // 通过key 来获取缓存的数据
    String strJson = (String) redisTemplate.opsForValue().get(key);
```

```
// 表示从缓存中获取到了数据
if (!StringUtil.isEmpty(strJson)){
    // 字符串存储的数据是什么? 就是方法的返回值类型
    Class returnType = signature.getReturnType();
    // 将字符串变为当前的返回值类型
    return JSON.parseObject(strJson, returnType);
}
return null;
}
```

4.3 使用注解完成缓存

```
@GmallCache(prefix = RedisConst.SKUKEY_PREFIX)
@Override
public SkuInfo getSkuInfo(Long skuId) {

    return getSkuInfoDB(skuId);
}
```

```
@GmallCache(prefix = "saleAttrValuesBySpu:")
public Map getSaleAttrValuesBySpu(Long spuId) {

    ....
}
```

```
@GmallCache(prefix = "spuSaleAttrListCheckBySku:")
public List<SpuSaleAttr> getSpuSaleAttrListCheckBySku(Long skuId, Long spuId) {

    ....
}
```

```
@Override
@GmallCache(prefix = "SpuPosterList:")
public List<SpuPoster> getSpuPosterList(Long spuId) {
    // select * from spu_poster where spu_id = spuId;
    return spuPosterMapper.selectList(new
    QueryWrapper<SpuPoster>().eq("spu_id", spuId));
}
```

```
@GmallCache(prefix = "categoryViewByCategory3Id:")
public BaseCategoryView getCategoryViewByCategory3Id(Long category3Id)
{

    ....
}
```

```
@Override
@GmallCache(prefix = "BaseAttrInfoList:")
public List<BaseAttrInfo> getBaseAttrInfoList(Long skuId) {
    // 根据 skuId 获取数据
    return baseAttrInfoMapper.selectBaseAttrInfoList(skuId);
}

@Override
public BigDecimal getPrice(Long skuId) {
    // select price from sku_info where id = skuId;
    // select * from sku_info where id = skuId;
    // SkuInfo skuInfo = skuInfoMapper.selectById(skuId);
    // 不需要将数据放入缓存!
    RLock lock = redissonClient.getLock(skuId + ":lock");
    // 上锁
    lock.lock();
    SkuInfo skuInfo = null;
    BigDecimal price = new BigDecimal(0);
    try {
        QueryWrapper<SkuInfo> skuInfoQueryWrapper = new
QueryWrapper<>();
        skuInfoQueryWrapper.eq("id", skuId);
        skuInfoQueryWrapper.select("price");
        skuInfo = skuInfoMapper.selectOne(skuInfoQueryWrapper);
        if (skuInfo!=null){
            price = skuInfo.getPrice();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }finally {
        // 解锁!
        lock.unlock();
    }
    // 返回价格
    return price;
}
```