尚品汇商城

版本：V 2.0

# 一、 缓存

## 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。

提高数据库本身的性能首先是优化sql，包括：使用索引，减少不必要的大表关联次数，控制查询字段的行数和列数。另外当数据量巨大是可以考虑分库分表，以减轻单点压力。

这部分知识在mysql高级已有讲解，这里大家可以以详情页中的sql作为练习，尝试进行优化，这里不做赘述。

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

缓存可以理解是数据库的一道保护伞，任何请求只要能在缓存中命中，都不会直接访问数据库。而缓存的处理性能是数据库10-100倍。

咱们就用Redis作为缓存系统进行优化。

安装Redis ：

1. 安装依赖
2. 导入压缩包
3. 创建安装目录
   1. mkdir –p /usr/local/redis
4. make
5. make install PREFIX=/usr/local/redis
6. 修改配置文件

## 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。

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**> |

添加redis配置类

|  |
| --- |
| **package** com.atguigu.gmall.common.config  @Configuration @EnableCaching **public class** RedisConfig {   @Bean  **public** KeyGenerator wiselyKeyGenerator() {  **return new** KeyGenerator() {  @Override  **public** Object generate(Object target, Method method, Object... params) {  StringBuilder sb = **new** StringBuilder();  sb.append(target.getClass().getName());  sb.append(method.getName());  **for** (Object obj : params) {  sb.append(obj.toString());  }  **return** sb.toString();  }  };  }   @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;  }   @Bean  **public** CacheManager cacheManager(RedisConnectionFactory factory) {  RedisSerializer<String> redisSerializer = **new** StringRedisSerializer();  Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = **new** Jackson2JsonRedisSerializer(Object.**class**);   *//解决查询缓存转换异常的问题* ObjectMapper om = **new** ObjectMapper();  om.setVisibility(PropertyAccessor.***ALL***, JsonAutoDetect.Visibility.***ANY***);  om.enableDefaultTyping(ObjectMapper.DefaultTyping.***NON\_FINAL***);  jackson2JsonRedisSerializer.setObjectMapper(om);   *// 配置序列化（解决乱码的问题）,过期时间600秒* RedisCacheConfiguration config = RedisCacheConfiguration.*defaultCacheConfig*()  .entryTtl(Duration.*ofSeconds*(600))  .serializeKeysWith(RedisSerializationContext.SerializationPair.*fromSerializer*(redisSerializer))  .serializeValuesWith(RedisSerializationContext.SerializationPair.*fromSerializer*(jackson2JsonRedisSerializer))  .disableCachingNullValues();   RedisCacheManager cacheManager = RedisCacheManager.*builder*(factory)  .cacheDefaults(config)  .build();  **return** cacheManager;  } } |

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

## 3 使用redis进行业务开发

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  \*  \** ***@author*** *qy  \*/* **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;  } |

|  |
| --- |
| @Autowired **private** RedisTemplate **redisTemplate**;  @Override **public** SkuInfo getSkuInfo(Long skuId) {  *// 定义key sku:skuId:info* String skuKey = RedisConst.***SKUKEY\_PREFIX*** + skuId + RedisConst.***SKUKEY\_SUFFIX***;   *// 分布式锁* SkuInfo skuInfo = (SkuInfo)**redisTemplate**.opsForValue().get(skuKey);  **if** (**null** == skuInfo){  skuInfo = getSkuInfoDB(skuId);  **this**.**redisTemplate**.opsForValue().set(skuKey, skuInfo);   }  **return** skuInfo; }  **public** SkuInfo getSkuInfoDB(Long skuId){  SkuInfo skuInfo = **skuInfoMapper**.selectById(skuId);  *// 根据skuId 查询图片列表集合* QueryWrapper<SkuImage> queryWrapper = **new** QueryWrapper<>();  queryWrapper.eq(**"sku\_id"**, skuId);  List<SkuImage> skuImageList = **skuImageMapper**.selectList(queryWrapper);   skuInfo.setSkuImageList(skuImageList);  **return** skuInfo; } |

以上基本实现使用缓存的方案。

## 4 缓存常见问题

缓存最常见的3个问题：

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。

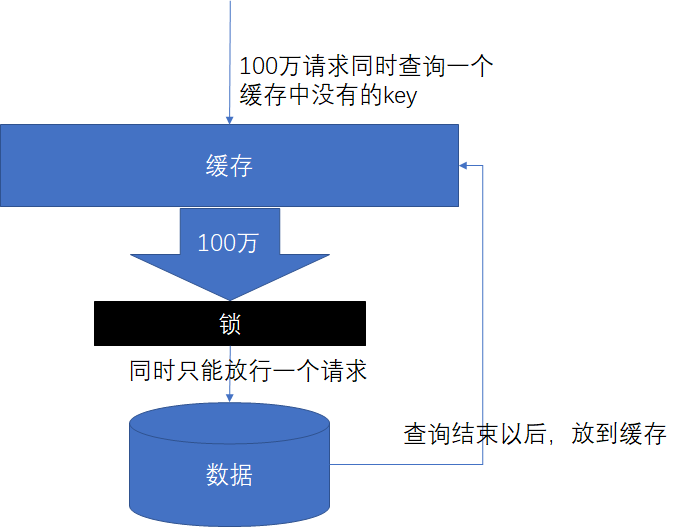
缓存击穿是指对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个key在大量请求同时进来之前正好失效，那么所有对这个key的数据查询都落到db，我们称为缓存击穿。

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



# 分布式锁

## 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

### 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  @Service **public class** TestServiceImpl **implements** TestService {   @Autowired  **private** RedisTemplate **redisTemplate**;   @Override  **public void** testLockMethod() {  Integer num = (Integer) **redisTemplate**.opsForValue().get(**"num"**);  **if**(!StringUtils.*isEmpty*(num)){  num++;  **redisTemplate**.opsForValue().set(**"num"**,num);  }  } } |

说明：通过reids客户端设置**num=0**

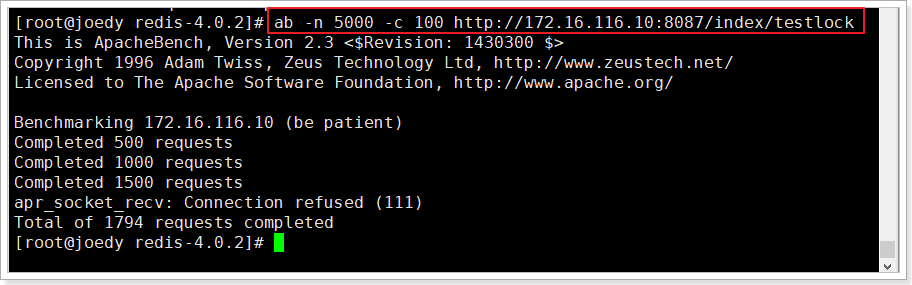
### 1.2 使用ab工具测试

之前在redis中，玩过ab测试工具：httpd-tools（yum install -y httpd-tools）

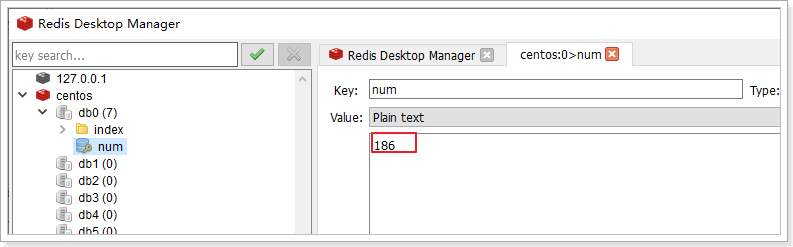
ab -n（一次发送的请求数） -c（请求的并发数） 访问路径

测试如下：5000请求，100并发

ab -n 5000 -c 100 http://api.gmall.com/admin/product/test/testLock



只完成了1794个请求。查看redis中的值：



只有186。这代码问题很大！

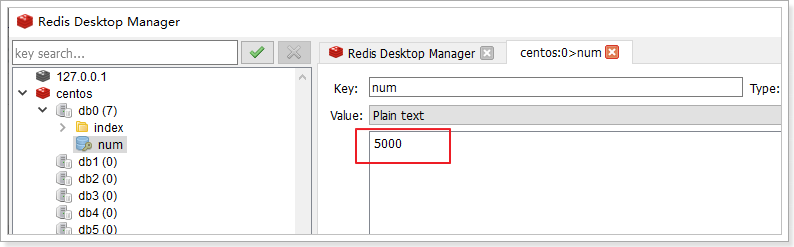
### 1.3使用本地锁

|  |
| --- |
| @Override **public synchronized void** testLockMethod() {  Integer num = (Integer) **redisTemplate**.opsForValue().get(**"num"**);  **if**(!StringUtils.*isEmpty*(num)){  num++;  **redisTemplate**.opsForValue().set(**"num"**,num);  } } |

使用ab工具压力测试：5000次请求，并发100



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？

### 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：

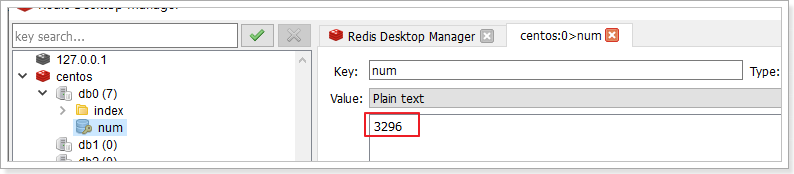
通过网关压力测试（通过网关自动会负载均衡）：

测试如下：5000请求，100并发

ab -n 5000 -c 100 http://api.gmall.com/admin/product/test/testLock



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 分布式锁实现

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。

## 3 使用redis实现分布式锁

Redis:命令

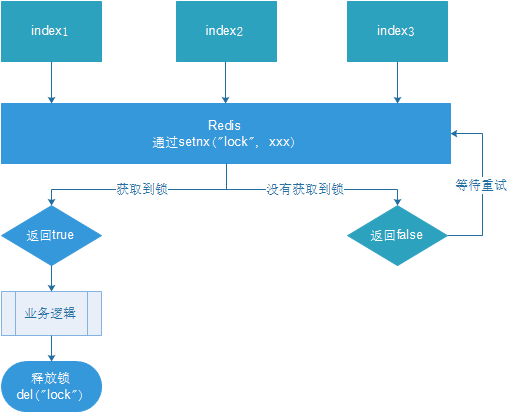
# set sku:1:info “OK” NX PX 10000

EX second ：设置键的过期时间为 second 秒。 SET key value EX second 效果等同于 SETEX key second value 。

PX millisecond ：设置键的过期时间为 millisecond 毫秒。 SET key value PX millisecond 效果等同于 PSETEX key millisecond value 。

NX ：只在键不存在时，才对键进行设置操作。 SET key value NX 效果等同于 SETNX key value 。

XX ：只在键已经存在时，才对键进行设置操作。



1. 多个客户端同时获取锁（setnx）

2. 获取成功，执行业务逻辑，执行完成释放锁（del）

3. 其他客户端等待重试

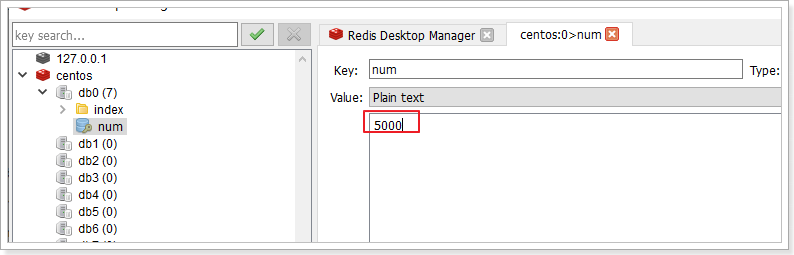
### 3.1 编写代码

|  |
| --- |
| @Override **public void** testLockMethod() {   *//使用Redis 的 setnx 上锁* Boolean lock = **redisTemplate**.opsForValue().setIfAbsent(**"lock"**, **"111"**);  **if**(lock){  *//上锁了* Integer num = (Integer) **redisTemplate**.opsForValue().get(**"num"**);  **if**(!StringUtils.*isEmpty*(num)){  num++;  **redisTemplate**.opsForValue().set(**"num"**,num);  }  *//取消锁* **redisTemplate**.delete(**"lock"**);  }**else**{  *//未上锁* **try** {  Thread.*sleep*(1000);  testLockMethod();  } **catch** (InterruptedException e) {  e.printStackTrace();  }   } } |

重启，服务集群，通过网关压力测试：



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

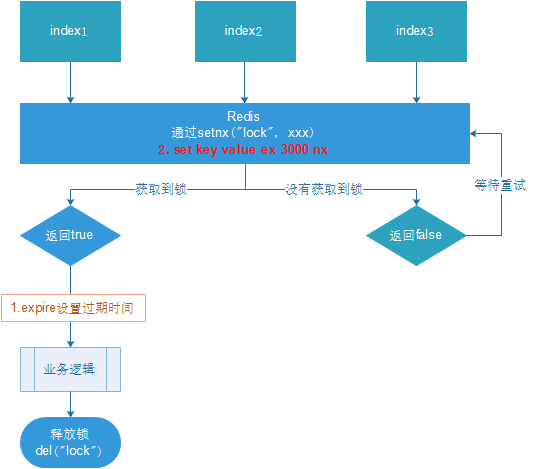
解决：设置过期时间，自动释放锁。

### 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：



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。

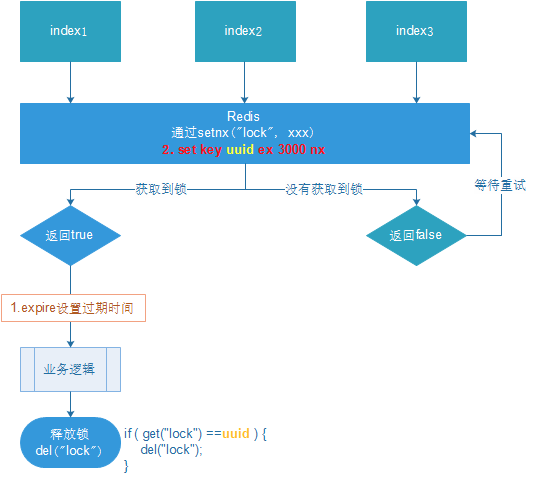
3. index3获取到锁，执行业务逻辑

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

### 3.3 优化之UUID防误删





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

2. index1执行删除前，lock刚好过期时间已到，被redis自动释放

3. index2获取了lock

4. index1执行删除，此时会把index2的lock删除

### 3.4 优化之LUA脚本保证删除的原子性

|  |
| --- |
| @Override  **public void** testLock() {  *// 1. 从redis中获取锁,setnx* String uuid = UUID.*randomUUID*().toString();  Boolean lock = **this**.**redisTemplate**.opsForValue()  .setIfAbsent(**"lock"**, uuid, 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* String script = **"if redis.call('get', KEYS[1]) == ARGV[1] then return tostring(redis.call('del',KEYS[1])) else return 0 end"**; **this**.**redisTemplate**.execute(**new** DefaultRedisScript<>(script), Collections.*singletonList*(**"lock"**), uuid); *// if(uuid.equals((String)redisTemplate.opsForValue().get("lock"))) { // 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*(500);  testLock();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

### 3.5 总结

1、加锁

|  |
| --- |
| *// 1. 从redis中获取锁,setnx* String uuid = UUID.*randomUUID*().toString(); Boolean lock = **this**.**redisTemplate**.opsForValue()  .setIfAbsent(**"lock"**, uuid, 2, TimeUnit.***SECONDS***); |

1. 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"**; **this**.**redisTemplate**.execute(**new** DefaultRedisScript<>(script), Arrays.*asList*(**"lock"**), Arrays.*asList*(uuid)); |

1. 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。

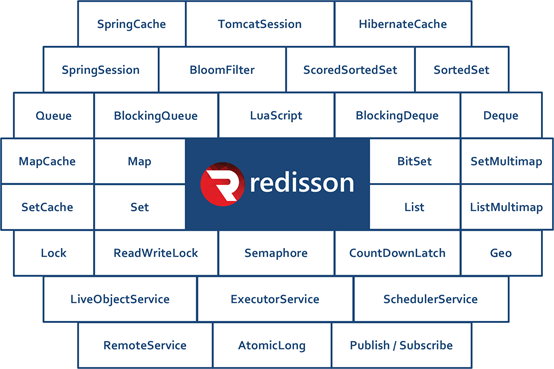
3. slave节点被晋级为master节点

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！

## 4 使用redisson 解决分布式锁

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务。其中包括(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>

连接文档：<https://github.com/redisson/redisson>

### 4.1 实现代码

|  |
| --- |
| 1. 导入依赖 service-util   *<!-- redisson -->* <**dependency**>  <**groupId**>org.redisson</**groupId**>  <**artifactId**>redisson</**artifactId**>  <**version**>3.11.2</**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);  } } |
| 1. 修改实现类   @Autowired **private** RedissonClient **redissonClient**;  @Override **public void** testLockMethod() {  RLock lock = **redissonClient**.getLock(**"lock"**);*//只要锁名称相同就是同一把锁* lock.lock(3,TimeUnit.***SECONDS***);*//上锁了 //默认是30秒 可以自定义修改* Integer num = (Integer) **redisTemplate**.opsForValue().get(**"num"**);  **if**(!StringUtils.*isEmpty*(num)){  num++;  **redisTemplate**.opsForValue().set(**"num"**,num);  }  lock.unlock();*//解锁* } |

### 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。

大家都知道，如果负责储存这个分布式锁的Redisson节点宕机以后，而且这个锁正好处于锁住的状态时，这个锁会出现锁死的状态。为了避免这种情况的发生，Redisson内部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。默认情况下，看门狗的检查锁的超时时间是30秒钟，也可以通过修改Config.lockWatchdogTimeout来另行指定。

另外Redisson还通过加锁的方法提供了leaseTime的参数来指定加锁的时间。超过这个时间后锁便自动解开了。

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：



注意事项：设置的10秒必须能完成下面的代码、否则锁就失效了。

### 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  @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。。。。。。"**; } |

打开开两个浏览器窗口测试：

- 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

# 分布式锁改造获取sku信息

### 4.1 使用redis

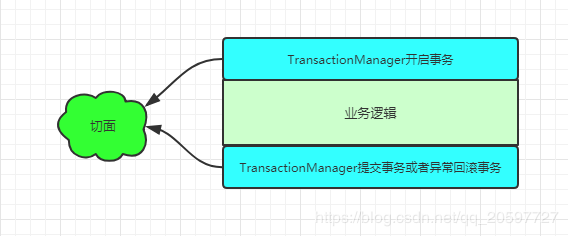
|  |
| --- |
| *//根据SkuID查询SkuInfo信息*   **public** SkuInfo getSkuInfo(Long skuId) {  String cacheKey = RedisConst.***SKUKEY\_PREFIX*** + skuId + RedisConst.***SKUKEY\_SUFFIX***;  String lockKey = RedisConst.***SKUKEY\_PREFIX*** + skuId + RedisConst.***SKULOCK\_SUFFIX***;  *//1:优先从缓存中获取数据* SkuInfo skuInfo = (SkuInfo) **redisTemplate**.opsForValue().get(cacheKey);  **if** (**null** != skuInfo) {  *//2:有 直接返回* **return** skuInfo;  } **else** {  String uuid = UUID.*randomUUID*().toString();  *//100 万请求 上锁 setnx 返回值 1 true 返回值 0 false* Boolean isLock = **redisTemplate**.opsForValue().setIfAbsent(lockKey, uuid, 1, TimeUnit.***SECONDS***);  **if** (isLock) {   *//上锁 100万人当中有一个已经来了 当前就是这个人  //3:没有 查询Mysql* skuInfo = **skuInfoMapper**.selectById(skuId);  *//如果是缓存穿透的话 Mysql中数据也是不存在的* **if** (**null** == skuInfo) {  skuInfo = **new** SkuInfo();*// 空结果  //解决： 将空结果保存在缓存中* **redisTemplate**.opsForValue().set(cacheKey, skuInfo, 5, TimeUnit.***MINUTES***);  } **else** {  *//查询图片* List<SkuImage> skuImageList = **skuImageMapper**.selectList(**new** QueryWrapper<SkuImage>().eq(**"sku\_id"**, skuId));  skuInfo.setSkuImageList(skuImageList);  *//保存缓存一份 并且设置时间为 1天 或 24小时  //缓存的雪崩问题及解决方案 // Random random = new Random(); // int i = random.nextInt(3000);* **redisTemplate**.opsForValue().set(cacheKey, skuInfo, RedisConst.***SKUKEY\_TIMEOUT***, TimeUnit.***SECONDS***);  }  *//防误删 // String u = (String) redisTemplate.opsForValue().get(lockKey); // if(!StringUtils.isEmpty(u) && uuid.equals(u)){ // //自己的锁 // redisTemplate.delete(lockKey); // }  //删除之原子性操作* String script = **"if redis.call('get', KEYS[1]) == ARGV[1] then return tostring(redis.call('del',KEYS[1])) else return 0 end"**;  **this**.**redisTemplate**.execute(**new** DefaultRedisScript<>(script), Collections.*singletonList*(lockKey), uuid);  } **else** {  *//别人已经上锁了 我们不能上锁了* **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *//重新加载当前方法* **return** getSkuInfo(skuId);  }  }  *//直接返回* **return** skuInfo;  } |

### 4.2 使用redisson

|  |
| --- |
| *//根据SKUID查询SKUINFO信息 本次将使用Redisson来完成分布式锁的实现* **public** SkuInfo getSkuInfoRedisson(Long skuId) {  String cacheKey = RedisConst.***SKUKEY\_PREFIX*** + skuId + RedisConst.***SKUKEY\_SUFFIX***;  String lockKey = RedisConst.***SKUKEY\_PREFIX*** + skuId + RedisConst.***SKULOCK\_SUFFIX***;  *//1:优先从缓存中获取数据* SkuInfo skuInfo = (SkuInfo) **redisTemplate**.opsForValue().get(cacheKey);  **if** (**null** != skuInfo) {  *//2:有 直接返回* **return** skuInfo;  } **else** {  *//3: 获取上锁对象* RLock lock = **redissonClient**.getLock(lockKey);  *//4:上锁 缓存击穿问题  //lock.lock(10, TimeUnit.SECONDS);//可重入锁 如果未获取锁 处于阻塞状态 一直等待 直到获取到锁* **try** {  *//参数1：尝试获取锁的时间 1s  //参数2：设置获取到锁的过期时间* **boolean** res = lock.tryLock(1, 3, TimeUnit.***SECONDS***);  **if**(res){  *//我是第一人  //5:查询DB* skuInfo = **skuInfoMapper**.selectById(skuId);  *//6:缓存穿透* **if** (**null** == skuInfo) {  skuInfo = **new** SkuInfo();  **redisTemplate**.opsForValue().set(cacheKey, skuInfo, 5, TimeUnit.***MINUTES***);  } **else** {  *//查询图片* List<SkuImage> skuImageList = **skuImageMapper**.selectList(**new** QueryWrapper<SkuImage>().eq(**"sku\_id"**, skuId));  skuInfo.setSkuImageList(skuImageList);  *//随机数 解决缓存雪崩问题  //缓存一天* **redisTemplate**.opsForValue().set(cacheKey, skuInfo,  RedisConst.***SKUKEY\_TIMEOUT***, TimeUnit.***SECONDS***);  }  }**else**{  *//我不是第一人 获取缓存中数据* Thread.*sleep*(1000);  **return** (SkuInfo) **redisTemplate**.opsForValue().get(cacheKey);  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }**finally** {  *//手动解锁* lock.unlock();  }  }  *//直接返回* **return** skuInfo; } |

# 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。



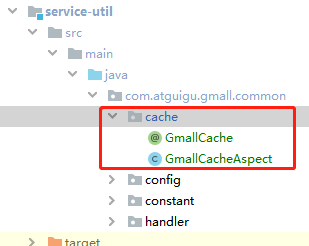
1. 以 @Transactional 注解为植入点的切点，这样才能知道@Transactional注解标注的方法需要被代理。

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



### 1 定义一个注解

|  |
| --- |
| **package** com.atguigu.gmall.common.cache;   **import** java.lang.annotation.\*;  */\*\*  \*****@Target：*** *\*  \** ***@Target说明了Annotation所修饰的对象范围：Annotation可被用于*** *\* packages、types（类、接口、枚举、Annotation类型）、  \* 类型成员（方法、构造方法、成员变量、枚举值）、  \* 方法参数和本地变量（如循环变量、catch参数）。  \* 在Annotation类型的声明中使用了target可更加明晰其修饰的目标。  \*  \* 作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方  \*  \* CLASS RUNTIME SOURCE  \* 按生命周期来划分可分为3类：  \* 1、RetentionPolicy.SOURCE：注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗弃；  \* 2、RetentionPolicy.CLASS：注解被保留到class文件，但jvm加载class文件时候被遗弃，这是默认的生命周期；  \* 3、RetentionPolicy.RUNTIME：注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在；  \*  \*  \** ***@Documented注解标记的元素，Javadoc工具会将此注解标记元素的注解信息包含在javadoc中*** *\*/* @Target({ElementType.***METHOD***}) *//此注解的使用范围 方法级别的注解 此注解打在方法上* @Retention(RetentionPolicy.***RUNTIME***) @Documented **public** @**interface** GmallCache {   */\*\*  \* 缓存key的前缀  \** ***@return*** *\*/* String prefix() **default "cache"**; } |

### 2定义一个切面类加强注解

|  |
| --- |
| **package com.atguigu.gmall.common.cache;**  **import com.atguigu.gmall.common.constant.RedisConst;**  **import lombok.extern.slf4j.Slf4j;**  **import org.aspectj.lang.ProceedingJoinPoint;**  **import org.aspectj.lang.Signature;**  **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.redisson.client.RedisTimeoutException;**  **import org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint;**  **import org.springframework.beans.factory.annotation.Autowired;**  **import org.springframework.data.redis.core.RedisTemplate;**  **import org.springframework.stereotype.Component;**  **import java.util.Arrays;**  **import java.util.concurrent.TimeUnit;**  **/\*\***  **\* 分布式锁 + Aop 切面实现类**  **\* 1、创建自定义注解**  **\* 2、编写切面类 实现类自定义注解的逻辑缓存**  **\*/**  **@Aspect**  **@Component**  **@Slf4j**  **public class GmallCacheAspect {**  **@Autowired**  **private RedissonClient redissonClient;//分布式锁**  **@Autowired**  **private RedisTemplate redisTemplate;**  **//环绕通知 实现类逻辑缓存**  **// 找注解 切入点表达式 而不是找类 找方法**  **@Around(value = "@annotation(com.atguigu.gmall.common.cache.GmallCache)")**  **public Object cacheData(ProceedingJoinPoint pjp) {**  **//方法的标签**  **MethodSignature signature = (MethodSignature) pjp.getSignature();**  **//方法上的注解**  **GmallCache gmallCache = signature.getMethod().getAnnotation(GmallCache.class);**  **//缓存的Key**  **String cacheKey = gmallCache.prefix() +**  **Arrays.asList(pjp.getArgs()) + RedisConst.CACHE\_SUFFIX;**  **//分布式锁的Key**  **String cacheLock = gmallCache.prefix() + Arrays.asList(pjp.getArgs()) +**  **RedisConst.LOCK\_SUFFIX;**  **//方法的返回值**  **Class returnType = signature.getReturnType();**  **// 获取缓存中的数据**  **Object o = redisTemplate.opsForValue().get(cacheKey);**  **if (null != o) {**  **return o;**  **}**  **RLock lock = redissonClient.getLock(cacheLock);**  **try {**  **boolean res = lock.tryLock(RedisConst.SKULOCK\_EXPIRE\_PX1,**  **RedisConst.SKULOCK\_EXPIRE\_PX2, TimeUnit.SECONDS);**  **if (res) {**  **//上锁成功 去方法中查询结果**  **Object result = pjp.proceed(pjp.getArgs());**  **//防止缓存穿透**  **if (null == result) {**  **result = returnType.newInstance();**  **redisTemplate.opsForValue().set(cacheKey**  **, result, 3, TimeUnit.MINUTES);**  **}else{**  **redisTemplate.opsForValue().set(cacheKey**  **, result, RedisConst.SKUKEY\_TIMEOUT, TimeUnit.SECONDS);**  **}**  **return result;**  **} else {**  **//上锁失败 睡一会 直接获取缓存中的数据**  **Thread.sleep(2000);**  **return redisTemplate.opsForValue().get(cacheKey);**  **}**  **} catch (Throwable e) {**  **//e.printStackTrace();**  **}finally {**  **lock.unlock();**  **}**  **return null;**  **}**  **}** |

### 2使用注解完成缓存

|  |
| --- |
| @GmallCache(prefix = RedisConst.***SKUKEY\_PREFIX***) @Override **public** SkuInfo getSkuInfo(Long skuId) {   **return** getSkuInfoDB(skuId); } |