商品详情

一、商品详情业务介绍

商品详情，简单说就是以购物者的角度展现一个商品的详情信息。

但是这个页面不同于传统的详情页面，与传统的详情页面主要有两点不同之处。

1）角色不同---商品详情页面使用者并不是管理员对该页面做一些操作，取而代之的是用户点击购买、放入购物车、切换颜色等等。

2）访问量不同---商品详情页面拥有较高的访问量，虽然只是一个查询操作，但是由于频繁的访问所以我们必须对其性能进行最大程度的优化，从而来提高系统的可用性和用户的体验度。

在商品详情中所需构建的数据信息如下

- Spu基本信息
- Sku图片信息
- Sku价格信息
- Sku分类信息
- Spu销售属性相关信息

图1-1商品详情业务图

二、商品详情核心代码片段

处理业务代码位置位于com.atguigu.gmall.item.service.impl.ItemServiceImpl.java

```
  private Map<String, Object> getFromServiceItemFeign(Long skuId) {
        log.info("开始远程查询，远程会操作数据库.....");

        Map<String, Object> result = new HashMap<>();
        //skuInfo信息

        //RPC 查询  skuDetail
        //1、Sku基本信息（名字，id，xxx，价格，sku_描述）sku_info
        //2，Sku图片信息（sku的默认图片[sku_info]，sku_image[一组图片])
```

```
        SkuInfo skuInfo =
skuInfoFeignClient.getSkuInfo(skuId);
        if (skuInfo != null) {
            result.put("skuInfo", skuInfo);
            //3，Sku分类信息（sku_info[只有三级分类]，根据这个三
级分类查出所在的一级，二级分类内容，连上三张分类表继续查）
            BaseCategoryView skuCategorys =
skuInfoFeignClient.getSkuCategorys(skuInfo.getCategory3Id());
            result.put("categoryView", skuCategorys);
            //4，Sku销售属性相关信息（查出自己的sku组合，还要查出
这个sku所在的spu定义了的所有销售属性和属性值）
            List<SpuSaleAttr> spuSaleAttrListCheckBySku =
skuInfoFeignClient.getSpuSaleAttrListCheckBySku(skuId,
skuInfo.getSpuId());
            result.put("spuSaleAttrList",
spuSaleAttrListCheckBySku);

            //5，Sku价格信息（平台可以单独修改价格，sku后续会放入
缓存，为了回显最新价格，所以单独获取）
            BigDecimal skuPrice =
skuInfoFeignClient.getSkuPrice(skuId);
            result.put("price", skuPrice);
            //6、SPU下面的所有存在的sku组合信息
{"121|123|156":65,"122|123|111":67}
            //前端这里还需要把map转成json字符串，
JSON.parse(.valuesSkuJson)
            Map map =
skuInfoFeignClient.getSkuValueIdsMap(skuInfo.getSpuId());
            ObjectMapper mapper = new ObjectMapper();
            try {
                String jsonStr =
mapper.writeValueAsString(map);
                log.info("valuesSkuJson 内容：{}", jsonStr);
                result.put("valuesSkuJson", jsonStr);
            } catch (JsonProcessingException e) {
```

```
                log.error("商品sku组合数据转换异常：{}", e);
            }

        }
        return result;
    }
```

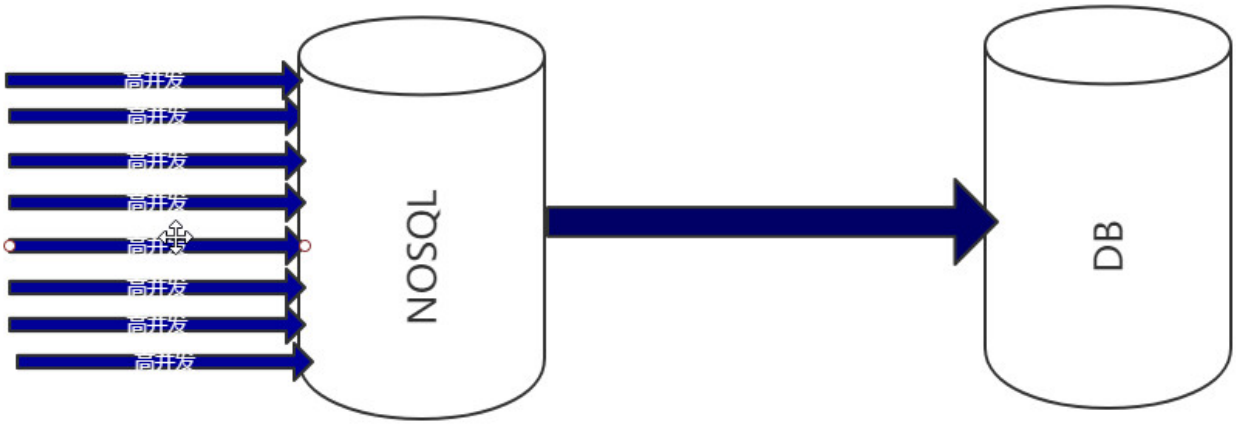
三、商品详情存在问题

作为开发人员，从功能业务角度来说，这已经实现了产品分配的需求【查询商品详情单】功能。但是由于详情页面不像传统的详情页面没有大流量，相反在该详情页面会涉及到高并发的压力，因此，我们必须要解决并发情况下保证商品详情页面快速的响应以及高可用这个问题。

四、商品详情解决并发方案

在查商品详情的数据时候采用传统的同步渲染，虽然服务器压力与异步渲染相比减轻不少。但是当并发过多的时候，服务器仍然面临宕机的情况。因此我们才用缓存在数据库的前面做一层抵挡，从而减轻对DB的压力。

虽然nosql数据库比较多，但在我们项目中 我们选择redis作为缓存解决并发问题。



从宏观角度上看 我们已经是解决了高并发去请求数据库并且性能也能提高不少，但是一旦我们使用redis作为缓存数据库的时候，随之而来的缓存穿透、缓存击穿、缓存雪崩问题也都无法规避，所以我们针对每种情况都采取具体的解决方案。

1、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：

1）空对象结果也进行缓存，但它的过期时间会很短，最长不超过五分钟。

2）使用布隆过滤器

使用布隆过滤器的步骤：

> 1）使用的是Redison自带的布隆过滤器

2）针对后台管理系统中的商品数据在添加的时候，向布隆过滤器中添加

3）在Spring容器中注入布隆对象并初始化布隆过滤器bit会大小和误判率

4）在查询缓存完之后 查询数据库之前 去查询布隆过滤器，如果布隆中有，则说明一定数据库中一定有，进一步执行查询数据库。反之 如果没有，则直接结束

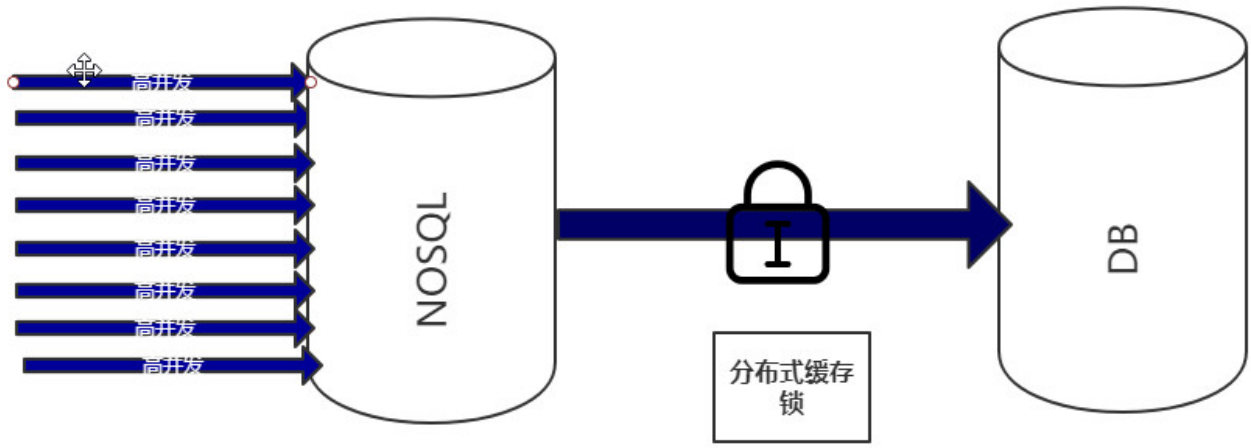
布隆说又 不一定有 布隆说没有 一定没有

布隆过滤器要在一定时间进行重新构建

2、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。

解决：加锁



问题：加什么锁，本地锁还是分布式锁？若是分布式锁是基于数据库的还是zk的还基于缓存的呢？这些选型都是一个问题。

1、本地锁和分布式锁的选择

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！ 因此我们选用的是分布式锁。

添加本地锁（Synchronized）：

针对商品中的商品skuId一个字符串常量来细粒度加锁，从而控制在单个节点下保证DB的安全性

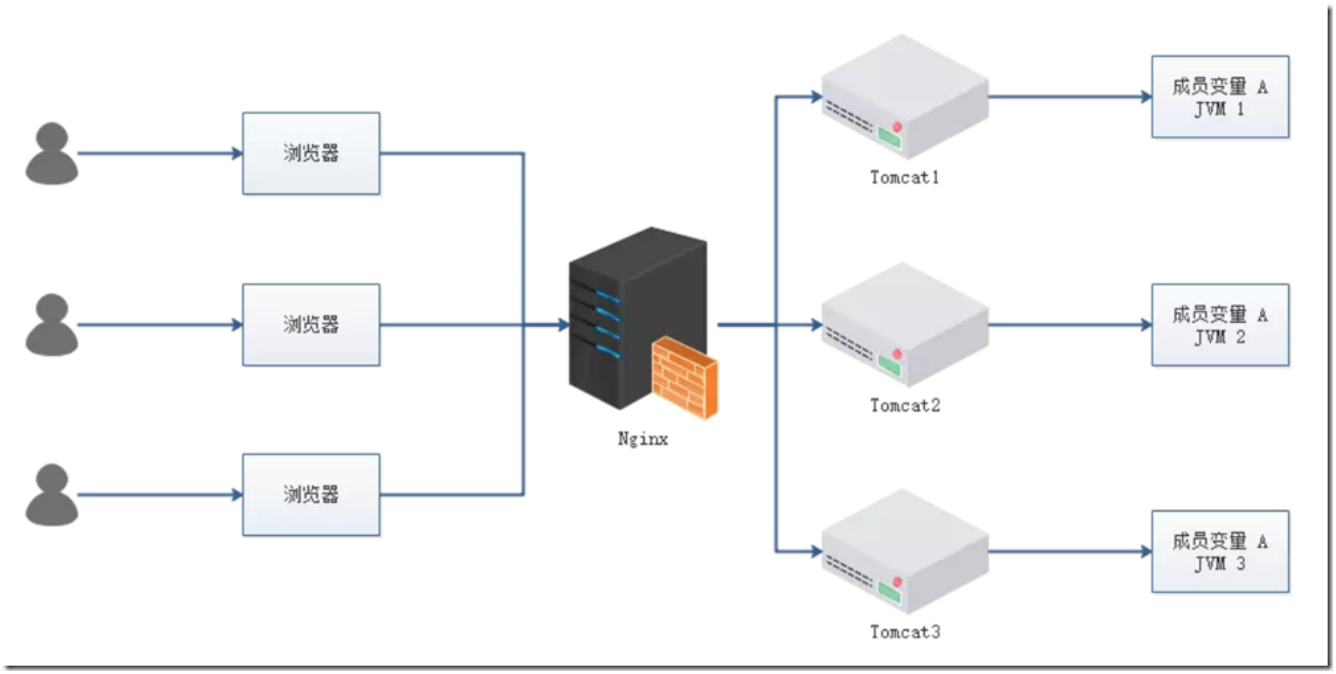
添加分布式锁:

1、使用redis的天生命令Setnx添加锁和Lua脚本解锁

针对某一个key进行加锁，从而保证在多个节点下保证同一时刻只有一个请求进来，从而让大量的请求处于等待状态，来保护我们的DB

2、什么是分布式锁

为了防止分布式系统中的多个进程之间相互干扰，我们需要一种分布式协调技术来对这些进程进行调度。而这个分布式协调技术的核心就是来实现这个分布式锁。



- 成员变量 A 存在 JVM1、JVM2、JVM3 三个 JVM 内存中
- 成员变量 A 同时都会在 JVM 分配一块内存，三个请求发过来同时对这个变量操作，显然结果是不对的
- 不是同时发过来，三个请求分别操作三个不同 JVM 内存区域的数据，变量 A 之间不存在共享，也不具有可见性，处理的结果也是不对的

注：该成员变量 A 是一个有状态的对象

而我们业务中确实存在这个场景的话，因此就需要一种方法解决这个问题，这就是分布式锁要解决的问题

2、基于什么的分布式锁

从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库

从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）

Zookeeper > 缓存 > 数据库

从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

因此我们项目最总从性能角度使用的是基于缓存的分布式锁。

基于缓存的redis做为分布式锁，天生自带一个命令setnx，其特点作用只在键不存在时，才对键进行设置操作。如果键存在，则该方法执行不成功。

流程：

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、添加分布式加锁的位置

由于添加分布式锁目的是为了：让某一个线程去请求DB，因此添加锁的位置在访问DB之前。

4、添加分布式锁出现的问题

1） 锁无法释放

场景：由于是在客户端通过delete手动释放锁，那么如果setnx刚好获取到锁，业务逻辑出现异常，导致锁无法释放

```
  Boolean ack = redisTemplate.opsForValue().setIfAbsent("sku:"
+ skuId + ":lock",key);
```

> 解决：服务端设置锁的过期时间，自动释放锁
>
> 那么释放锁的的过期时间有两种
>
> 其一：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）】
>
> expire sku:" + skuId + ":lock 3
>
> 其二：在set时 指定过期时间
>
> Boolean OK = redisTemplate.opsForValue().setIfAbsent("sku:" + skuId + ":lock", 1, 2, TimeUnit.SECONDS);

2） 锁的误删操作

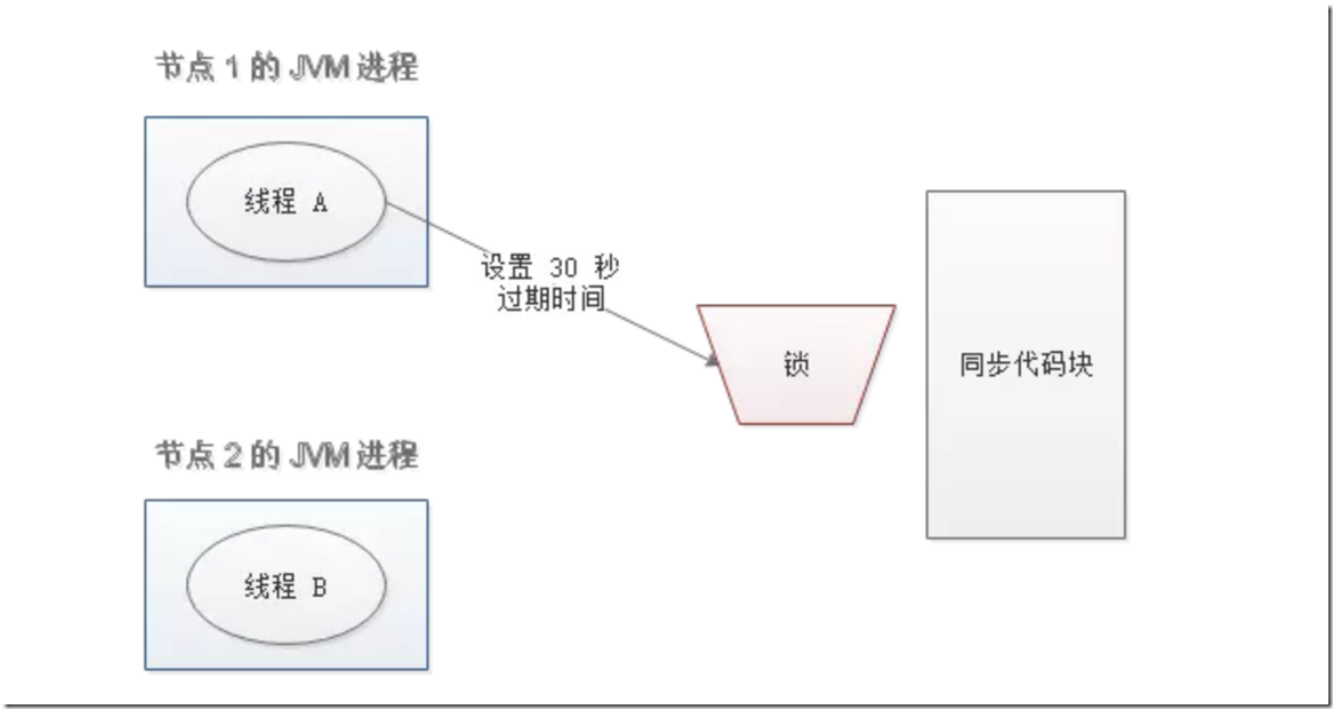
场景：

如果业务逻辑的执行时间是4s。执行流程如下

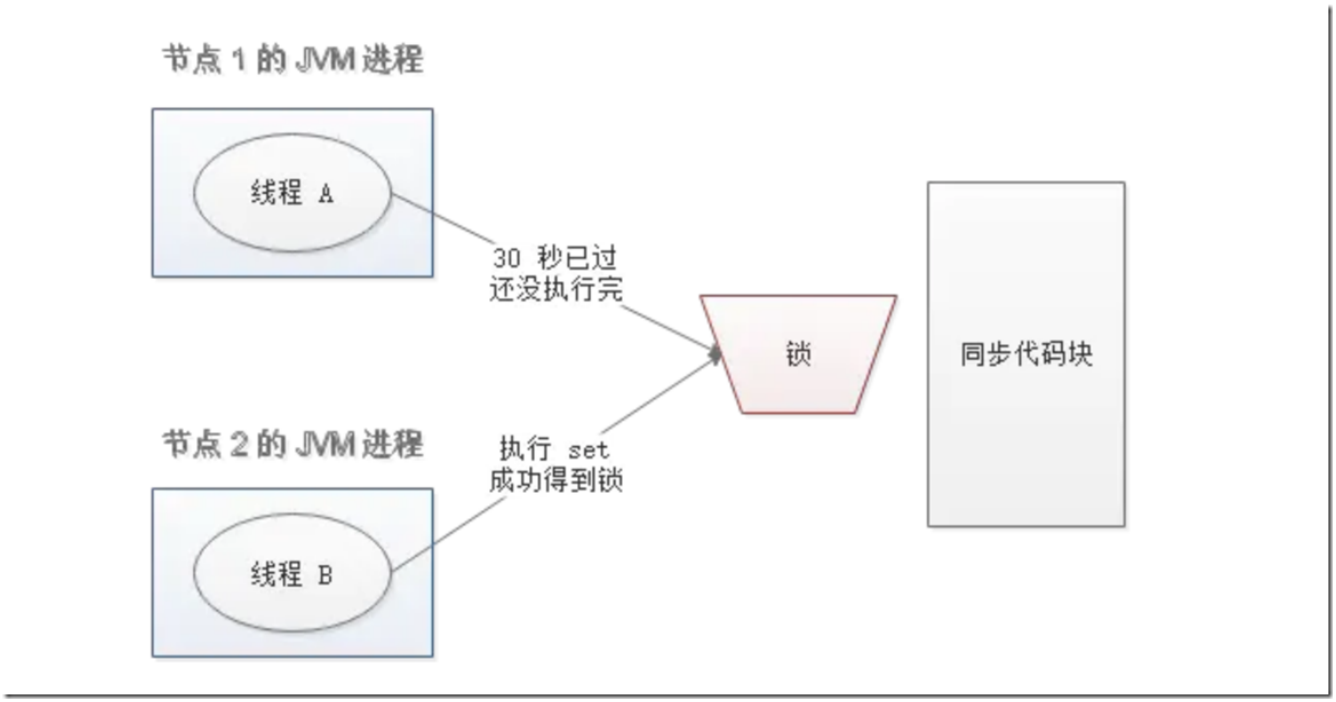
- A线程开始添加锁并设置过期时间4s，开始查询数据库，将数据库数据同步到缓存
- A线程该释放锁的时候，由于网络故障4s，没有执行到此步，但是这时锁已经过期了。
- B线程此时进来又重新添加一个新锁，然后此时A线程刚好恢复网络，开始执行释放锁。

- 那么就出现了一个问题，此时A线程却释放了B线程的锁。
- 锁没用，高并发并没有挡住。

    - 又是一个极端场景，假如某线程成功得到了锁，并且设置的超时时间是 30 秒。



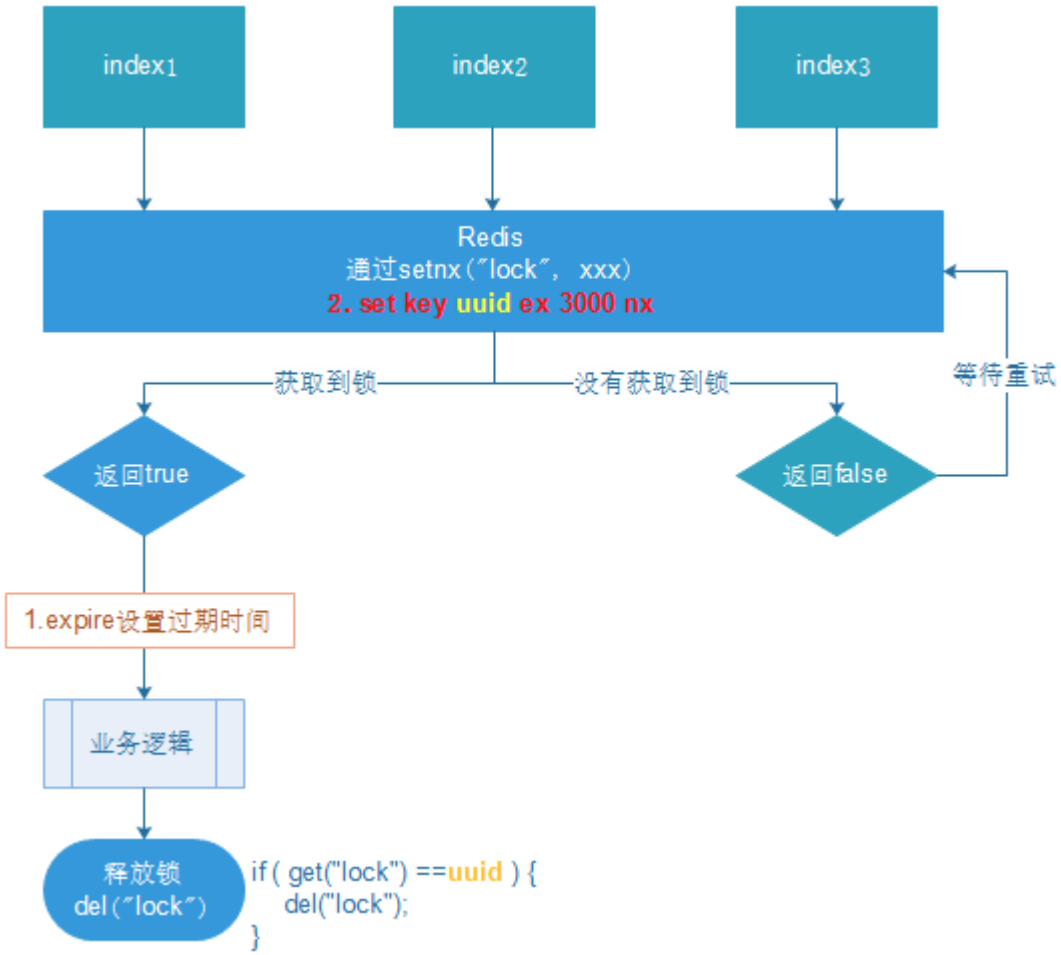
如果某些原因导致线程 A 执行的很慢很慢，过了 30 秒都没执行完，这时候锁过期自动释放，线程 B 得到了锁。



随后，线程 A 执行完了任务，线程 A 接着执行 del 指令来释放锁。但这时候线程 B 还没执行完，线程A实际上 删除的是线程 B 加的锁。

> 解决： setnx获取锁时，给这把锁设置一个指定的唯一值（例如：uuid）；手动释放前在根据key获取这个值，判断是否自己的锁，在做删除

```
String openKey = (String)
redisTemplate.opsForValue().get("sku:" + skuId + ":lock");
if (key.equals(openKey)) {
    // 如果发现二者相等
    redisTemplate.delete("sku:" + skuId + ":lock");
}
```



看上去似乎没有什么问题，但是还是不能够完全保证判断和删除能够一起完成。如果我们能将判断和删除做成一个操作，就完美解决了。

场景：

- index1执行删除时，查询到的lock值确实和uuid相等
- index1执行删除前，lock刚好过期时间已到，被redis自动释放
- index2获取了lock
- index1执行删除，此时会把index2的lock删除

> 解决：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("sku:" + skuId + ":lock"),key);
```

3）自旋锁的升级

对于加锁成功的线程，可以顺利的去执行我们的业务逻辑，但是对于加锁失败的其它线程来说，该怎么办呢？通常会有两种方案来解决

①：递归使用该方法。

②：直接查询缓存。

针对方法一和方法二，如果在一切正常情况下，业务逻辑是完全能够得到执行的，但是如果执行过程中出现问题，业务将会出现问题。

代码逻辑大致分为：

```
public Map<String, Object> getSkuDetailsWithRedisLock(Long
skuId){

        // 先查redis
       Map cache =
cacheService.getFromCache(skuId.toString(), Map.class);
        if (cache == null) {
          // 加锁
```

```
            absent =
redisTemplate.opsForValue().setIfAbsent(lockKey, token, 20,
TimeUnit.SECONDS);
           //查询DB
            if (absent) {
            Map<String, Object> data = getDataFromDB(skuId);
            cacheService.putCache(skuId.toString(), data);
            // 手动去释放锁
                redisTemplate.delete(lockKey)

    }
    else{
        // 递归
        return getSkuDetailsWithRedisLock (skuId)
    }
 }

}
```

分析：由代码业务可知，在执行查询方法中，大致分为

> 查询缓存
>
> 缓存为空
>
> 加锁
>
> 加锁成功
>
> 查询DB
>
> 同步缓存
>
> 加锁失败
>
> **递归、直接查询缓存**

> 缓存不为空
>
> 返回数据

③：递归、缓存问题模拟：

单个节点情况：

假设查询业务这段代码需要执行5s完成，且只有A、B 2个线程进入，同时执行查 询缓存，但A线程顺利抢到了锁，则B线程就会执行递归操作，而与此同时A线程在刚执行查询数据库的时候，出现了网络抖动，【预估30分钟内网络不能恢复】。那么B线程接着去执行递归方法，也是先去查询缓存，此时缓存数据并没有被A线程从数据库同步到缓存，因此B线程继续执行加锁操作。如果此时A线程锁在20s被释放了 那么B线程就会加锁成功，执行后面保存数据到缓存，那么在该节点下的后续线程都会直接从缓存中查询到数据。那么也就是说只有在A线程没有释放锁的20s期间，后面的所有线程都会去执行递归方法，若并发量小，可能只是在20s时间用户查询响应得不到数据，若大并发情况下，可能会因为大量的请求来执行递归，造成栈空间溢出，服务直接挂掉。

多个节点情况下：

如果商品详情服务部署了10个节点，机器序号分别为1-10，并且假设在大并发情况下，1号机器分担1w并发，2号机器分担3w并发,...10号机器分担2w并发。百万并发进来，假设只有1号机器的010号线程成功获取到了锁，那么100w-1的线程就都加锁失败，来执行我们的递归方法。但是，好巧不巧，刚好抢到锁的这个线程所在的机器A 出现了断电，那么会在短时间内，造成10台机器的栈空间溢出，整个集群服务都不可用，服务直接崩掉。

> 同理直接查缓存问题也是一样，我就不再这里赘述

④： 解决：使用自旋锁

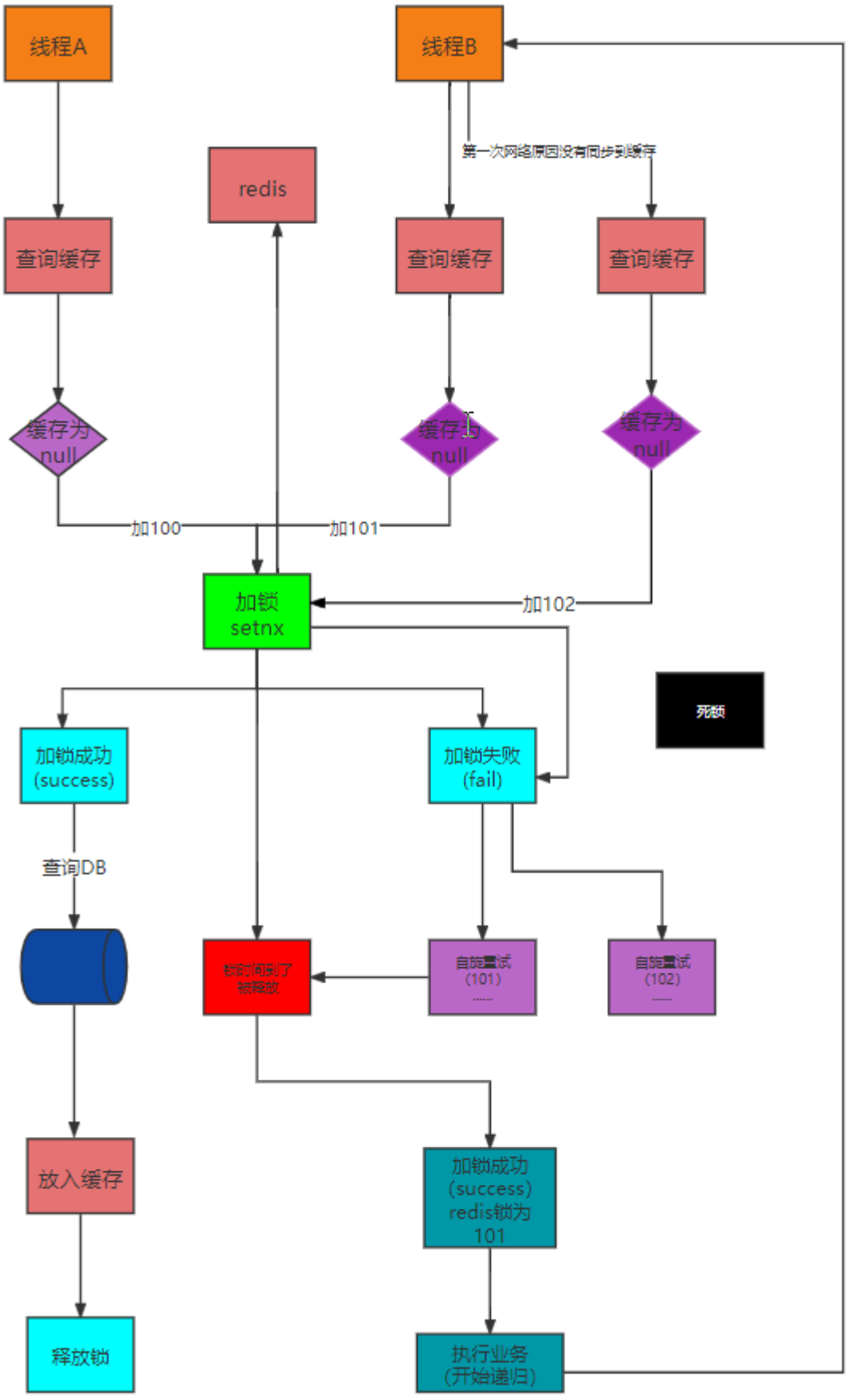
因为项目中我们针对加锁失败的线程都去使用递归调用，大并发情况下很快就出现了栈溢出，从而导致服务不可用，因此想设计一种能不能轻量级的抢锁机制，只从抢占锁的这一个角度去把控，而不是从整个方法层面去做控制，同时只让加锁成功的线程去递归调用，大量没有加锁成功的线程去循环，那么显然用一个for(；；)或者while(true)的死循环来做，就可以达到一种轻量级的控制。而这种死循环，也就是我们的自旋思想。

```
while (true) { //CPU一直起飞
    TimeUnit.SECONDS.sleep(1);
    System.out.println("线程: " +
Thread.currentThread().getId() + "; 【" + token + "】自旋重
试....");
    Boolean locked =
redisTemplate.opsForValue().setIfAbsent(lockKey, token, 20,
TimeUnit.SECONDS);
    if (locked) {
        //锁住了
        System.out.println("线程: " +
Thread.currentThread().getId() + "; 【" + token + "】自旋加锁成
功....");
        threadLocal.set(token);
        break;
    }
     //执行业务
   return getSkuDetails(skuId);
}
```

⑤：死锁现象的模拟

通过使用了自旋锁解决了栈溢出，但是当有线程加锁成功之后，开始执行递归调用的时候，可能会出现死锁现象。

场景模拟：


线程A
线程B
redis
第一次网络原因没有同步到缓存
查询缓存
查询缓存
查询缓存
缓存为null
缓存为null
缓存为null
加100
加101
加102
加锁
setnx
死锁
加锁成功
(success)
加锁失败
(fail)
查询DB
设时间到了
被释放
自旋重试
(101)
.....
自旋重试
(102)
.....
放入缓存
加锁成功
(success)
redis锁为
101
释放锁
执行业务
(开始递归)

⑥：死锁的解决：

①：使用一个Map==（key：为当前线程 value：为Token）

当自旋加锁成功之后，第二次在进入到该方法后，又会重新执行加锁操作，因此将会出现死锁现象，所以第一次加锁成功之后，在第二次在进入该方法，就不要对当前线程执行加锁操作，继续使用上一次的锁即可解决死锁现象。因此这就是可重入锁的思想。

在进入自旋加锁以后，往以当前线程为key,token为value的map中添加一个entity,然后再递归方法的业务逻辑中，先根据当前的线程为key 去map中获取value,如果获取不到，如果value==null,则表明该线程没有加入过锁，那么就让改线程来执行加锁操作。反之，value中获取到值，则表明该线程已经加过锁，那么就不让改线程在次加锁，执行加锁成功后的业务代码。

②：ThreadLocal（因为ThreadLoca中底层就是一个以当前线程为key的map）

可能出现问题：

⑦：内存泄漏

解决：

①：通过线上日志以及抓取到的当前微服务内存模型

②：jvisualvm连上线上应用进行观测，发现有一个Map内存不停的再往上涨

③: 定位该Map是我们商品详情中用来解决死锁而使用的一个本地Map或者ThreadLocal。

④：发现业务代码中，一直不停的在往这个Map存放数据，但是没有去做移除操作

⑤：在业务执行完，移除每个线程往Map中不停存放的数据

4）添加分布式锁注意事项

- 为了保证分布式锁可用 使用分布式锁要保证这四点
- 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
- 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

5）集群状态下安全失效

场景：

- 客户端A从master获取到锁
- 在master将锁同步到slave之前，master宕掉了。
- slave节点被晋级为master节点
- 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。

> 解决：使用redission解决分布式锁
>
> ```
> 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();
> }
> }
> ```

了解

红锁---RedLock

基于多个 Redis 节点实现高可靠的分布式锁

当我们要实现高可靠的分布式锁时，就不能只依赖单个的命令操作了，我们需要按照一定的步骤和规则进行加解锁操作，否则，就可能会出现锁无法工作的情况。“一定的步骤和规则”是指啥呢？其实就是分布式锁的算法。

为了避免 Redis 实例故障而导致的锁无法工作的问题，Redis 的开发者提出了分布式锁算法 Redlock

Redlock 算法的基本思路，是让客户端和多个独立的 Redis 实例依次请求加锁，如果客户端能够和半数以上的实例成功地完成加锁操作，那么我们就认为，客户端成功地获得分布式锁了，否则加锁失败。这样一来，即使有单个 Redis 实例发生故障，因为锁变量在其它实例上也有保存，所以，客户端仍然可以正常地进行锁操作，锁变量并不会丢失。

Redlock 算法的实现需要有 N 个独立的 Redis 实例。接下来，分成 3 步来完成加锁操作。

第一步是，客户端获取当前时间。

第二步是，客户端按顺序依次向 N 个 Redis 实例执行加锁操作。

这里的加锁操作和在单实例上执行的加锁操作一样，使用 SET 命令，带上 NX，EX/PX 选项，以及带上客户端的唯一标识。当然，如果某个 Redis 实例发生故障了，为了保证在这种情况下，Redlock 算法能够继续运行，我们需要给加锁操作设置一个超时时间。如果客户端在和一个 Redis 实例请求加锁时，一直到超时都没有成功，那么此时，客户端会和下一个 Redis 实例继续请求加锁。加锁操作的超时时间需要远远地小于锁的有效时间，一般也就是设置为几十毫秒

第三步是，一旦客户端完成了和所有 Redis 实例的加锁操作，客户端就要计算整个加锁过程的总耗时。

客户端只有在满足下面的这两个条件时，才能认为是加锁成功。

- 条件一：客户端从超过半数（大于等于 N/2+1）的 Redis 实例上成功获取到了锁；
- 条件二：客户端获取锁的总耗时没有超过锁的有效时间。

> 在满足了这两个条件后，我们需要重新计算这把锁的有效时间，计算的结果是锁的最初有效时间减去客户端为获取锁的总耗时。如果锁的有效时间已经来不及完成共享数据的操作了，我们可以释放锁，以免出现还没完成数据操作，锁就过期了的情况。
>
> 在 Redlock 算法中，释放锁的操作和在单实例上释放锁的操作一样，只要执行释放锁的 Lua 脚本就可以了。这样一来，只要 N 个 Redis 实例中的半数以上实例能正常工作，就能保证分布式锁的正常工作了

3、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。

> 小总结：
>
> 你们这个模块都遇到哪些问题？
>
> 1、接口响应速度慢
>
> 2、缓存穿透

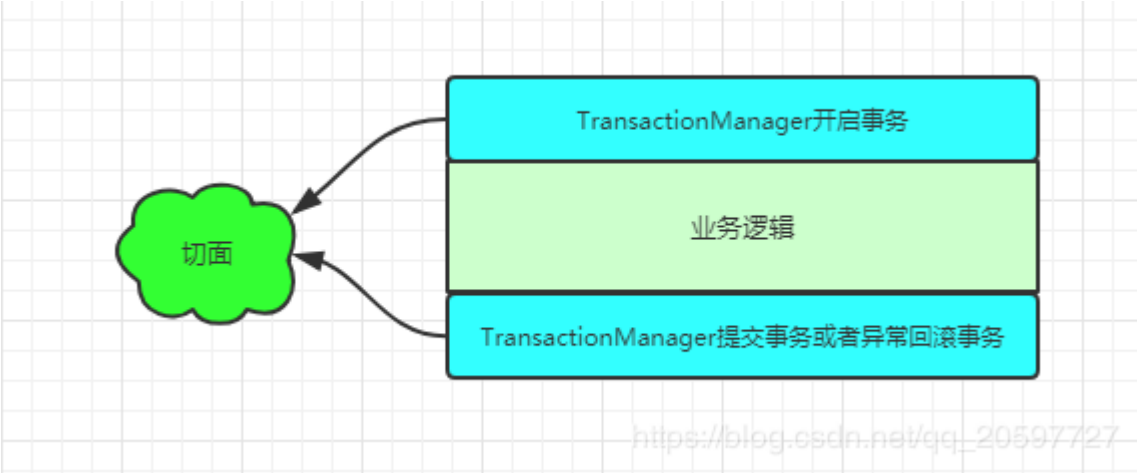
> 3、缓存击穿
>
> 4、死锁现象
>
> 5、内存泄漏

> 小总结：
>
> 如何解决的？
>
> 根据文档中的内容以及总结课上所讲 都分被对应有解决方案。千万不要说你做的这个模块没有遇到任何问题。。。。。。更不能说，你这个项目是跟着老师做的

4、缓存优化

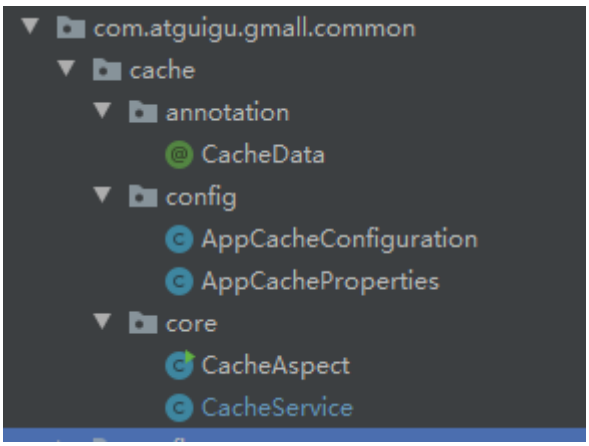
本地锁 + AOP实现缓存

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。



模拟事务，缓存可以这样实现：

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



- 定义一个注解

```
/**
 * 需要加入缓存逻辑的数据，只需要标注这个注解即可
 */
@Target({ElementType.METHOD})  //代表当前注解只能标注在方法位置
@Retention(RetentionPolicy.RUNTIME)  //这个注解运行期有效
public @interface CacheData {
    /**
     * 允许手动指定key，这里需要传入表达式
     * SpEL：Spring表达式
```

```
     * #{xxxx}
     */
    String keyExpression() default "";
    /**
     * 是否开启布隆过滤器功能
     * @return
     */
    boolean bloomEnable() default true;
}
```

- 定义一个切面类

```
@Slf4j
@Component
@Aspect   //表示这个是一个切面
public class CacheAspect {
 @Autowired
    CacheService cacheService;


    @Autowired
    RedissonClient redissonClient;


    @Autowired
    RBloomFilter<Long> skuBloomFilter;


    //spu, sku, image, order


    @Autowired
    private StringRedisTemplate redisTemplate;
```

```
 /**
     * 本地锁版
     * 基于字符串本地锁高性能动态缓存表达式可选择性布隆泛型缓存切面
     *
     * @param pjp
     * @return
     * @throws Throwable
     */

 @Around("@annotation(com.atguigu.gmall.common.cache.annotation.CacheData)") //切入点表达式
    public Object aroundWithLocalLock(ProceedingJoinPoint pjp) throws Throwable {
        //1、分析缓存注解,预准备信息
        CacheData cacheData = getMethodAnnotation(pjp, CacheData.class);
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        Method method = signature.getMethod();
        Type type = method.getGenericReturnType(); //精确，带泛型
        Object[] args = pjp.getArgs();

        log.info("缓存切面，开始执行.....");
        //2、先查缓存    //key可以通过一个动态的表达式确定值，不应该是写死的逻辑
        String cacheKey = parseCacheKey(pjp); //当前就是skuId
        //目标方法需要map类型的数据  //返回期望的数据类型
        //应该目标方法的返回值类型
        Object cache = cacheService.getFromCache(cacheKey, type);
        if (cache == null) {
            //缓存中没有，执行目标方法
            //3、防止缓存击穿，先加锁
```

```
            String lockKey = "lock-" + cacheKey; //锁的粒度
//  lock-49   lock-50
            //缓存本地锁
            //TODO 字符串锁,性能强，极快的执行
            synchronized (lockKey.intern()){ //
                //1、(this) 代表当前对象作为锁。this代表当前切面？切面对象有几个？ (Spring容器中的组件默认是单实例的)
                //2、(this) 代表当前切面,当前机器只有一个?
                //3、百万并发  10w-49  10w-50  10w-51  this锁粒度太大?   jvm字符串不可变性
                //3、再查缓存
                cache = cacheService.getFromCache(cacheKey, type);
                if (cache == null) {
                    //缓存中没有就查数据库
                    //1、先看是否需要开启布隆过滤器
                    Object proceed = null;
                    if(cacheData.bloomEnable()){
                        boolean contains = skuBloomFilter.contains((Long) args[0]);
                        if(contains){
                            //2、放行目标方法
                            proceed = pjp.proceed(args);
                        }
                    }else {
                        //2、放行目标方法
                        proceed = pjp.proceed(args);
                    }
                    cacheService.putCache(cacheKey,proceed);
                    return proceed;
                }else {
                    return cache;
                }

            }
```

```
        } else {
            //返回缓存中，把json转成对象返回出去
            return cache;
        }
    }

    /**
     * 指定了key，就用key，没有指定就用参数
     *
     * @return
     */
    private String parseCacheKey(ProceedingJoinPoint pjp) {

        //#{#params[0]}   #{ #param  }
        //1、获取表达式
        CacheData cacheData =
getMethodAnnotation(pjp,CacheData.class);

        //获取到注解写的表达式内容
        String expression = cacheData.keyExpression();
        //计算表达式的结果
        String result = evaluteExpression(expression,pjp);


        //表达式结果就是缓存用的key
        return result;
    }

    private <A extends Annotation> A
getMethodAnnotation(ProceedingJoinPoint pjp, Class<A>
annotationType) {
        MethodSignature signature = (MethodSignature)
pjp.getSignature();
        //先获取到当前方法
```

```
        Method method = signature.getMethod();

        return AnnotationUtils.findAnnotation(method,
annotationType);
    }


    /**
     * 计算表达式
     * @param expression
     * @param pjp
     * @return
     */
    private String evaluteExpression(String expression,
ProceedingJoinPoint pjp) {
        //1、先拿到一个表达式解析器
        SpelExpressionParser parser = new
SpelExpressionParser();


        //2、解析出表达式对象
        Expression expObj = parser.parseExpression(expression,
new TemplateParserContext());


        //3、计算表达式的值
        //准备一个计算环境
        EvaluationContext context = new
StandardEvaluationContext();
        //根据上面的环境算出表达式的最终结果
        Object[] args = pjp.getArgs();

        context.setVariable("params",args);
        context.setVariable("date",new SimpleDateFormat("yyyy-
MM-dd").format(new Date()));
```

```
            context.setVariable("random",
UUID.randomUUID().toString());
            //#{#params[0]}  #{#random}
            //categorys
            String value = expObj.getValue(context, String.class);

            return value;

        }

}
```

- 使用注解完成缓存

```
    @CacheData(keyExpression = "#{#params[0]}") //这个表达式自己
算出的是 skuId（第一个参数的值）
    @Override  //直接查库
    public Map<String, Object> getSkuDetails(Long skuId) throws
Exception {
        log.info("sku信息开始查询数据库.....");
        Map<String, Object> dataFromDB = getDataFromDB(skuId);
        return dataFromDB;
    }
```

布隆过滤器+分布式锁+AOP实现缓存

@Component @Aspect public class GmallCacheAspect { @Autowired private RedisTemplate redisTemplate;

```
@Autowired
private RedissonClient redissonClient;
```

```
@Slf4j
@Component
@Aspect
public class GmallCacheAspect {
    @Qualifier(BloomName.SKU)
    @Autowired
    RBloomFilter<Object> skufilter;
    /**
     * Spring会自动从容器中拿到所有 RBloomFilter 类型的组件，
     * 给我们封装好map，map的key用的就是组件的名，值就是组件对象
     */
    @Autowired
    Map<String, RBloomFilter<Object>> blooms; //要把全系统的布隆过滤器都加入进来，进行选择
    @Autowired
    StringRedisTemplate stringRedisTemplate;
    @Autowired
    RedissonClient redissonClient;
    /**
     * 所有能标了 @GmallCache 的一般都是id查
     * 环绕通知= 前置+返回+后置+异常
     *
     * @param pjp 连接点
     * @return
     */
    //哪种通知都能阻拦目标方法？？？ 环绕通知
    //切入点表达；看谁标注了我们自己定义的注解

@Around("@annotation(com.atguigu.gmall.common.cache.GmallCache)")
```

```
    public Object cacheAround(ProceedingJoinPoint pjp) throws
Throwable {
        Object[] args = pjp.getArgs();
        MethodSignature signature = (MethodSignature)
pjp.getSignature();
        Method method = signature.getMethod();
        GmallCache gmallCache =
signature.getMethod().getAnnotation(GmallCache.class);
        String prefix = gmallCache.bloomPrefix();
        String bloomSuffix = gmallCache.bloomSuffix();
        String bloomName = gmallCache.bloomName();
        long missDataTtl = gmallCache.missDataTtl();
        long ttl = gmallCache.ttl();
        String redisCacheKey = prefix + args[0] + bloomSuffix;
        String intern = redisCacheKey.intern();
        log.info("redisCacheKey：对象地址：", redisCacheKey);
        RBloomFilter<Object> bloomFilter =
blooms.get(bloomName);
        boolean contains = bloomFilter.contains(redisCacheKey);
        RLock lock = null;
        if (!contains) {
            //布隆过滤器没有此值
            return null;
        }
        try {
            //1、先看缓存有没有
            Object cache = getFromCache(redisCacheKey,method);
//要json转成方法返回值类型，但是用Object，毕竟切面返回都是Object
            if (cache == null) {
                //如果缓存中没有，字符串是常量池的  "sku:51:info"
在常量池就一个地址
                log.info("分布式缓存切面，准备加锁执行目标方
法.....");
                String lockKey = prefix + ":lock";
                //  准备上锁
```

```
                lock = redissonClient.getLock(lockKey);
                boolean result =
lock.tryLock(RedisConst.SKULOCK_EXPIRE_PX1,
RedisConst.SKULOCK_EXPIRE_PX2, TimeUnit.SECONDS);
                if (result){
                    //如果加锁成功
                    Object cacheTemp =
getFromCache(redisCacheKey,method);
                    if(cacheTemp==null){
                        Object  object =
pjp.proceed(pjp.getArgs());
                        saveToCache(redisCacheKey, object,
ttl,missDataTtl);
                        return  object;
                    }
                    return cacheTemp;
                }
            }
            return cache;
        } catch (Exception e){
            log.error("缓存切面失败：{}",e);
        }finally {
            //后置通知
            lock.unlock();
        }
        return null;
    }
    /**
     * 把数据保存到缓存
     *
     * @param redisCacheKey
     * @param proceed
     * @param ttl
     */
    private void saveToCache(String redisCacheKey, Object
```

```
proceed, long ttl,long missTtl) throws JsonProcessingException
{
    }
    /**
     * 把数据从缓存加载
     *
     * 别把这个写死  Map<String, Object>
     * @param cacheKey
     * @return
     */
    private Object getFromCache(String cacheKey,Method method)
throws JsonProcessingException {

    }
```

布隆过滤器+本地锁+AOP(自定义表达式并动态获取注解参数值以及泛型返回值)实现缓存

> 通过使用Aop代理的设计模式，自定义注解的解耦合思想，最终我们在商品详情模块实现了缓存的极致优化 在自定义注解中定义了动态可插拔的布隆过滤器，实现不同的缓存业务不同的处理逻辑，并且采用了Spring5中强大的的Spel表达式语言动态的从不同的入参实现对redis中key值的精准的使用，以及针对不同的返回数据类型的统一接收和处理。并在最后结合我们业务的模式采用了本地sychronized（商品id==字符串锁的方式）代替分布式锁的优化，从而大大提高了接口的响应速度。

六、redis有关面试题

1、简单介绍一下redis

（1）redis是一个key-value类型的非关系型数据库，基于内存也可持久化的数据库，相对于关系型数据库（数据主要存在硬盘中），性能高，因此我们一般用redis来做缓存使用；并且redis支持丰富的数据类型，比较容易解决各种问题

（2）Redis的Value支持5种基本数据类型，string、hash、list、set、zset（sorted set）；

String类型是最简单的类型，一个key对应一个value，项目中我们主要利用单点登录中的用户信息用string类型来存储；

Hash类型中的key是string类型，value又相当于一个map（key-value），针对这种数据特性，比较适合存储对象；

List类型是按照插入顺序的字符串链表（双向链表），主要命令是LPUSH和RPUSH，能够支持反向查找和遍历；

Set类型是用哈希表类型的字符串序列，没有顺序，集合成员是唯一的，没有重复数据。

zset（sorted set）类型和set类型基本是一致的，不同的是zset这种类型会给每个元素关联一个**double**类型的分数（score），这样就可以为成员排序，并且插入是有序的。

2、redis在项目中怎么用的?

（1）商品详情中的数据放入缓存，分布式锁；

（2）单点登录系统中也用到了redis。因为我们是微服务系统，把用户信息存到redis中便于多系统之间获取共用数据；

（3）我们项目中同时也将购物车的信息设计存储在redis中，用户未登录采用UUID作为Key，value是购物车对象；用户登录之后将商品添加到购物车后存储到redis中，key是用户id，value是购物车对象；

（4）订单模块的，结算页中订单流水号String类型；

（5）秒杀中使用list类型控制库存；

（6）搜索中商品热度统计，使用sorted set；

3、对redis的持久化了解不?

Redis是内存型数据库，同时它也可以持久化到硬盘中，redis的持久化方式有两种：

（1）RDB（半持久化方式）：

按照配置不定期的通过异步的方式、快照的形式直接把内存中的数据持久化到磁盘的一个dump.rdb文件（二进制文件）中；

这种方式是redis默认的持久化方式，它在配置文件（redis.conf）中的格式是：save N M，表示的是在N秒之内发生M次修改，则redis抓快照到磁盘中；

优点：只包含一个文件，对于文件备份、灾难恢复而言，比较实用。因为我们可以轻松的将一个单独的文件转移到其他存储媒介上；性能最大化，因为对于这种半持久化方式，使用的是写时拷贝技术，可以极大的避免服务进程执行IO操作；相对于AOF来说，如果数据集很大，RDB的启动效率就会很高

缺点：如果想保证数据的高可用（最大限度的包装数据丢失），那么RDB这种半持久化方式不是一个很好的选择，因为系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产生丢失；rdb是通过子进程来协助完成持久化的，因此当数据集较大的时候，我们就需要等待服务器停止几百毫秒甚至一秒；

（2）AOF（全持久化的方式）

把每一次数据变化都通过write()函数将你所执行的命令追加到一个appendonly.aof文件里面；

Redis默认是不支持这种全持久化方式的，需要将no改成yes

实现文件刷新的三种方式：

no:不会自动同步到磁盘上，需要依靠OS（操作系统）进行刷新，效率快，但是安全性就比较差；

always:每提交一个命令都调用异步刷新到aof文件，非常慢，但是安全；

everysec:每秒钟都调用fsync刷新到aof文件中，很快，但是可能丢失一秒内的数据，推荐使用，兼顾了速度和安全；

优点：

数据安全性高

该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机问题，也不会破坏日志文件中已经存在的内容；

缺点：

对于数量相同的数据集来说，aof文件通常要比rdb文件大，因此rdb在恢复大数据集时的速度大于AOF；

根据同步策略的不同，AOF在运行效率上往往慢于RDB，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效；

针对以上两种不同的持久化方式，如果缓存数据安全性要求比较高的话，用aof这种持久化方式（比如项目中的购物车）；如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

4、redis的淘汰策略你了解多少?

volatile-lru：设置了过期时间的key使用LRU算法淘汰； allkeys-lru：所有key使用LRU算法淘汰； volatile-lfu：设置了过期时间的key使用LFU算法淘汰；allkeys-lfu：所有key使用LFU算法淘汰； volatile-random：设置了过期时间的key使用随机淘汰； allkeys-random：所有key使用随机淘汰； volatile-ttl：设置了过期时间的key根据过期时间淘汰，越早过期越早淘汰； noeviction：默认策略，当内存达到设置的最大值时，所有申请内存的操作都会报错(如set,lpush等)，只读操作如get命令可以正常执行；

- LRU算法底层你知道吗?

Redis中的LRU与常规的LRU实现并不相同，常规LRU会准确的淘汰掉队头的元素，但是Redis的LRU并不维护队列，只是根据配置的策略要么从所有的key中随机选择N个（N可以配置）要么从所有的设置了过期时间的key中选出N个键，然后再从这N个键中选出最久没有使用的一个key进行淘汰。、

常规LRU:

LRU的实现方式是使用HashMap结合双向链表，HashMap的值是双向链表的节点，双向链表的节点也保存一份key value。

- 新增key value的时候首先在链表结尾添加Node节点，如果超过LRU设置的阈值就淘汰队头的节点并删除掉HashMap中对应的节点。
- 修改key对应的值的时候先修改对应的Node中的值，然后把Node节点移动队尾。
- 访问key对应的值的时候把访问的Node节点移动到队尾即可。

RedisLRU:

Redis维护了一个24位时钟，可以简单理解为当前系统的时间戳，每隔一定时间会更新这个时钟。每个key对象内部同样维护了一个24位的时钟，当新增key对象的时候会把系统的时钟赋值到这个内部对象时钟。比如我现在要进行LRU，那么首先拿到当前的全局时钟，然后再找到内部时钟与全局时钟距离时间最久的（差最大）进行淘汰，这里值得注意的是全局时钟只有24位，按秒为单位来表示才能存储194天，所以可能会出现key的时钟大于全局时钟的情况，如果这种情况出现那么就两个相加而不是相减来求最久的key。

5、使用redis如何保证和数据库的一致性？

什么情况下缓存和数据库会不一致？

在高并发的情况下，如果所有的数据都从数据库中去读取，那再强大的数据库系统都承受不了这个压力，因此我们会将部分数据放入缓存中，比如放入redis中。这是典型的用空间换时间的方式。

但是这个redis相当于是真实数据的一个副本，这就意味着如果数据库中数据发生变化的时候，就会导致缓存数据不一致的问题。

归根结底，只要有两份数据存在，数据一致性问题就是不可避免的。

解决方案1：数据实时更新

当更新数据库的时候，同步更新缓存。

优点：数据一致性强，不会出现缓存雪崩的问题。

缺点：代码耦合度高，影响正常业务，增加一次网络开销。

适用环境：适用于数据一致性要求高的场景，比如银行业务，证券交易

解决方案2：数据准实时更新

当更新数据库的同时，异步去更新缓存，比如更新数据库后把一条消息发送到mq中去实现。

优点：与业务解耦，不影响正常业务，不会出现缓存雪崩。

缺点：有较短的延迟，并且无法保证最终的一致性，需要补偿机制。

适用环境：写操作不频繁并且实时性要求不严格的场景。

双写模式：

- 在写数据库的同时 去将数据写入redis
- 在并发写的情况下可能会数据不一致
- **慢的线程旧数据居然把新数据覆盖** 这是暂时性的脏数据问题，但是在数据稳定，缓存过期以后，又能得到最新的正确数据。

失效模式：

- 在写数据库的同时，去删除缓存。
- 在并发的情况下也可能出现不一致

解决方案3：缓存失效机制

基于缓存本身的失效机制，具体实现方式为设置缓存失效时间，如果有缓存就从缓存中取数据，如果没缓存就从数据库中取数据，并且重新设置缓存。

优点：实现方式简单，与业务完美解耦，不影响正常业务。

缺点：有一定延迟，并且存在缓存雪崩的情况。

适用环境：适合读多写少的互联网环境，能接受一定的数据延时。

解决方案4：延时双删

情况一：先删除缓存，再更新数据库。

在线程A更新完数据库值以后，我们可以让它先sleep一小段时间，再进行一次缓存删除操作。

之所以要加上sleep的这段时间，就是为了让线程B能够先从数据库读取数据，再把缺失的数据写入缓存，然后，线程A再进行删除。所以，线程A sleep的时间，就需要大于线程B读取数据再写入缓存的时间。这个时间怎么确定呢？建议你在业务程序运行的时候，统计下线程读数据和写缓存的操作时间，以此为基础来进行估算。 redis.delKey(X) db.update(X) Thread.sleep(N) redis.delKey(X)

情况二：先更新数据库值，再删除缓存值。

如果线程A删除了数据库中的值，但还没来得及删除缓存值，线程B就开始读取数据了，那么此时，线程B查询缓存时，发现缓存命中，就会直接从缓存中读取旧值。不过，在这种情况下，如果其他线程并发读缓存的请求不多，那么，就不会有很多请求读取到旧值。而且，线程A一般也会很快删除缓存值，这样一来，其他线程再次读取时，就会发生缓存缺失，进而从数据库中读取最新值。所以，这种情况对业务的影响较小 db.update(X)

redis.delKey(X)

Thread.sleep(N) redis.delKey(X)

==========

不管哪种方式，都可能导致数据库数据和缓存数据不一致问题

解决：

1、牺牲性能，进行加锁

双写模式下进行加锁，写数据库和写缓存应该保持一个操作，中间不能被打断。

锁的选择：

1）分布式锁

2）粒度，锁当前数据

3）锁的形式，使用读写锁

2、引入中间件Canal，数据同步指的mysql和mysql的同步

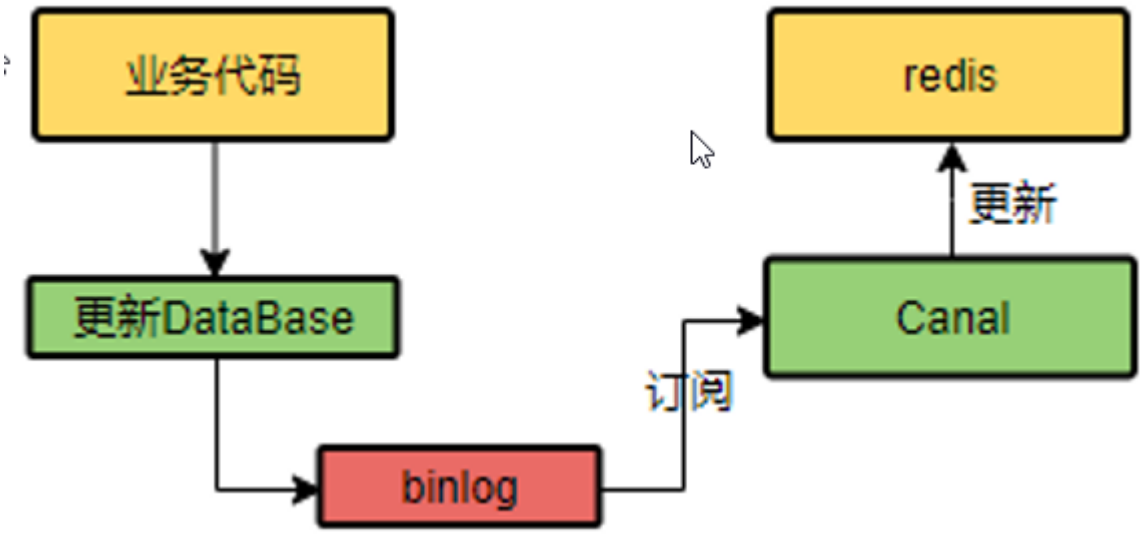
1）读Redis：热数据基本都在Redis

2）写MySQL:增删改都是操作MySQL

3）更新Redis数据：MySQ的数据操作binlog，来更新到Redis

第一次将数据全部写入到redis中做一个全量备份，接这mysql中的insert update delet

作为增量进行实时更新。当读取到binglog后分析，利用消息对列，推送到各个redis实例中去，对redis进行更新。



优点：类似于mysql的主备模式，因为MySQL的主备也是通过binlog来实现的数据一致性

缺点： 引入中间件，代码开发难度较大，成本较高。

> 你们项目中的商品详情模块如何解决数据缓存不一致的问题？
>
> 我们项目中解决一致性问题，即不用加锁也没有引入canel，只用一种失效模式或者延时双删。
>
> 因为我们系统中在详情模块缓存中存放的数据，是我们商品数据信息，而这种数据是一种读多写少并且一旦录入到后台基本不修改。即使修改也没事，因为我们不是并发修改。初次之外我们每个数据都有过期时间，因此用失效模式或者双写模式足够了。
>
> 因为失效模式或者双删模式+过期时间。可以保证数据的最终一致，只会让过期时间以内的数据出现短时间的不一致。

6、redis单线程为什么这么快？

通常来说，单线程的处理能力要比多线程差很多，但是 Redis 却能使用单线程模型达到每秒数十万级别的处理能力，这是为什么呢？其实，这是 Redis 多方面设计选择的一个综合结果。

一方面，Redis 的大部分操作在内存上完成，再加上它采用了高效的数据结构，例如哈希表和跳表，这是它实现高性能的一个重要原因。另一方面，就是 Redis 采用了多路复用机制，使其在网络 IO 操作中能并发处理大量的客户端请求，实现高吞吐率

Redis 单线程是指它对网络 IO 和数据读写的操作采用了一个线程，而采用单线程的一个核心原因是避免多线程开发的并发控制问题。单线程的 Redis 也能获得高性能，跟多路复用的 IO 模型密切相关，因为这避免了 accept() 和 send()/recv() 潜在的网络 IO 操作阻塞点。