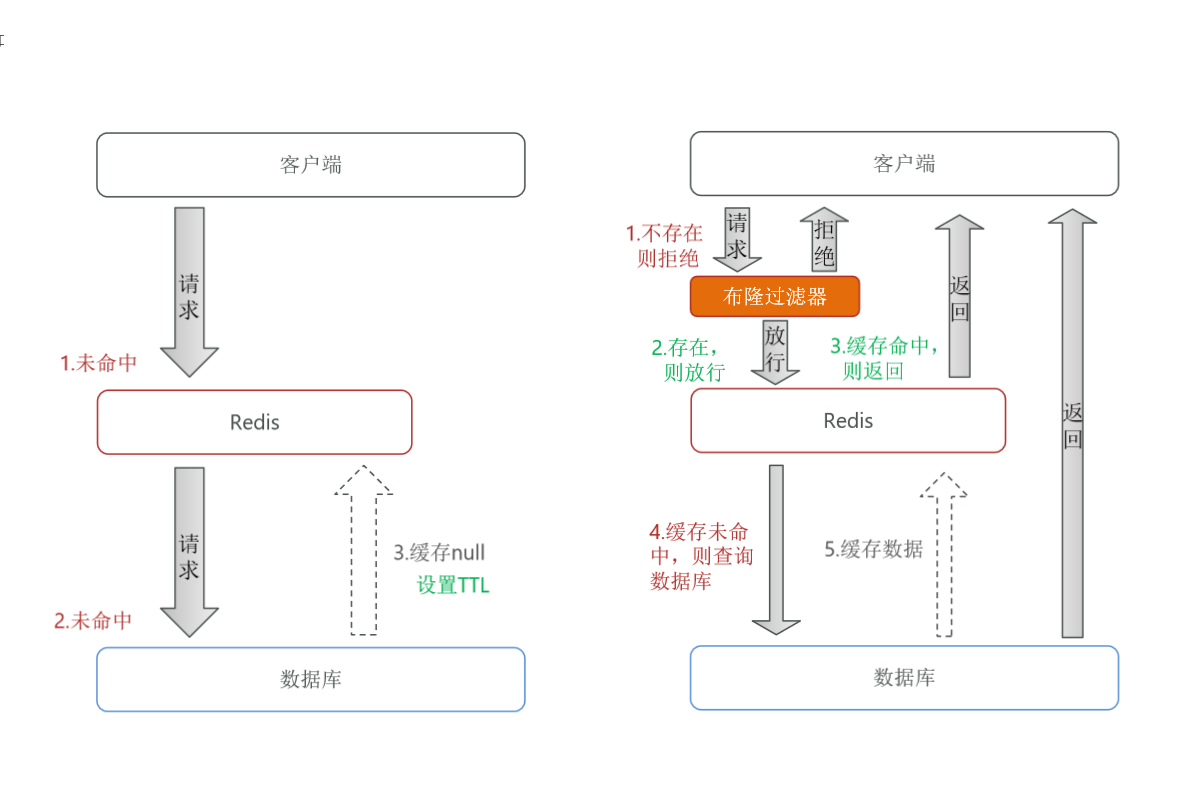
# 缓存穿透，缓存雪崩、缓存击穿

## 缓存穿透

### 什么是缓存穿透

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，永远不会建立缓存，这些请求都会打到数据库，很有可鞥就把数据库搞垮！



### 缓存穿透常见的解决方案有两种：

#### 一：缓存空对像

把空值缓存到Redis中，下次再来查询时，就能在Redis中命中了，尽管命中的是空值，那也是命中。

优点：实现简单，维护方便

缺点1：额外的内存消耗，随便请求不存在的数据，如果没有，都缓存起来了，极可能会缓存很多不存在的垃圾，但是也有解决方法，就是可以缓存的时候加个比较短的TTL；

缺点2：可能造成短期的数据不一致，如果不存在的数据真的加进数据库了，那么缓存中的空值在有效期内还存在，会造成短期的数据不一致，解决方法：TTL设置短些；加入数据库，主动更新缓存。

#### 二：布隆过滤

是在客户端与Redis之间加多了一层布隆过滤器，用户情求来的时候，不是上来就去找Redis，是先在布隆过滤器中，看要查的数据是否存在，如果不存在会直接拒绝，存在才会去Redis查询，去Redis查询，剩下的逻辑就跟以前一样，也就是说后续的逻辑不变，仅仅实在客户端和Redis之间加了前置的过滤。

你可能问布隆过滤器怎么知道我请求的数据有还是没有呢？难道说布隆过滤器中存储了数据库中所有的数据吗？肯定不是，布隆过滤器的原理你可以简单地理解成是一个byte数组，里面存的就是二进制位，当我们要去判断数据库中的数据是否存在的时候，其实并不是真的把数据存储到布隆过滤器，而是把数据基于某一种hash算法计算出哈希值，然后再把这些哈希值转化成二进制位保存到布隆过滤器里，即对应01的形式去保存，而后我们去判断数据是否存在的时候，其实就是判断对应的位置是0还是1，以此来判断数据是否存在。

注意，这种存在与否是一种概率上的统计，并不是百分之百地准确，当布隆过滤器判断数据不存在的时候，那是真的不存在；当布隆过滤器告诉你存在的时候，其实不一定存在，这样就又出现穿透了，即有一定的穿透风险。即没有的一定没有，有的不一定有！

优点：内存占用较少，没有多余的key

缺点一：实现复杂，好在Redis中提供了Bitmap自带的布隆过滤器实现。可以帮助我们简化开发。

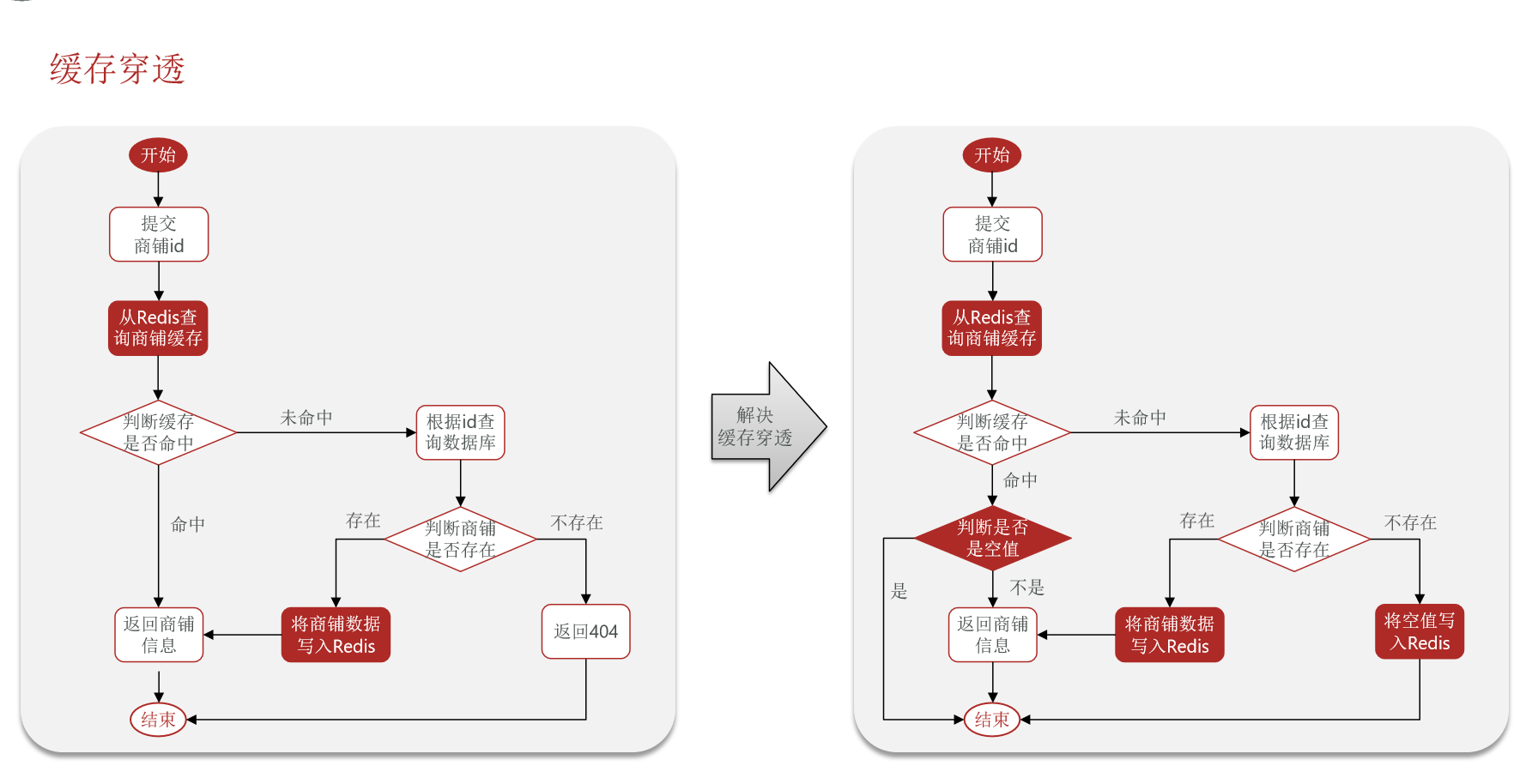
缺点二：存在误判可能，当布隆过滤器说不存在时是一定不存在，存在就不一定真的存在，有一定的穿透风险。

### 解决查询商铺业务的缓存穿透问题

以下是业务流程图的两个变化：

1：是原来流程中的404就是导致缓存穿透的原因，我们要改就改404！即数据库中不存在就缓存空对象到Redis，而不是返回404。

2：既然把空值写到Redis了，当我们从Redis命中时，你命中的就不一定是商铺信息了，还有可能是空值，所以说你从Redis命中了以后还需要对命中结果做一个判断，判断一下命中的是不是空，不是空返回商铺信息，是空直接返回空信息。



### 总结：

#### 缓存穿透产生的的原因是什么？

用户请求的数据再缓存和数据库中都不存在，不断发起这样的请求，给数据库带来巨大的压力

#### 缓存穿透的解决方案有哪些？

##### 缓存空对象；

##### 布隆过滤；

##### 增加id的复杂度，避免被猜测id规律

缓存穿透不是仅仅只有缓存空对象和布隆过滤这两种方案，这两种属于被动的方案，就是请求已经来穿透你了，你被动想办法去弥补。事实上，我们可以主动地采取一些措施去解决缓存穿透，比如说我们可以增加id的复杂度，避免被猜测id的规律，这样一来别人就不太可能去输入一些自己编的id来请求。

##### 做好数据的基础格式校验

你的id复杂度增加了，你就可以去加强基础参数格式的校验，比如不符合id规律的请求直接再controller层拒绝！

##### 加强用户权限校验

一些业务不是任何人都可以去访问，比如说要求必须要先登陆，然后就算是登陆用户，我们还可以对用户做限流，什么用户可以访问，用户访问的频率限制等等！

##### 做好热点参数的限流

比如优先商品访问的就是多，那么就可以做限流

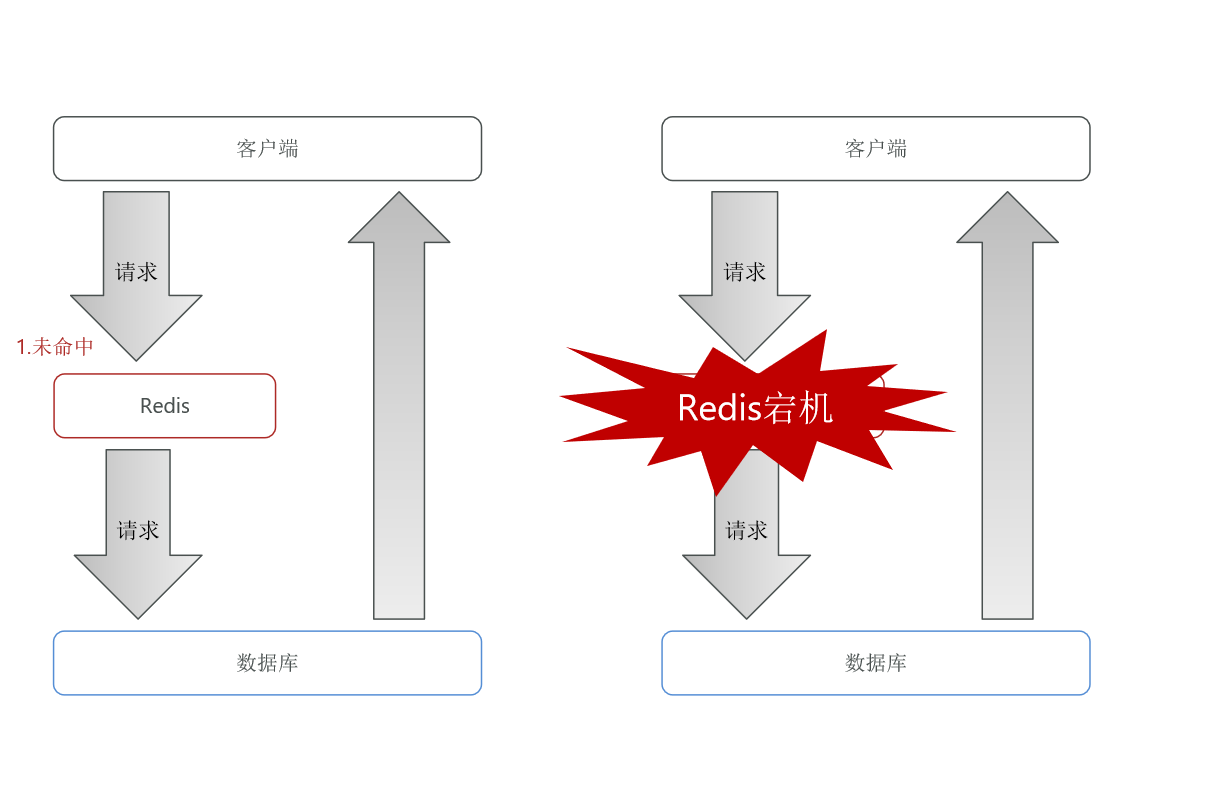
## 缓存雪崩

### 缓存雪崩介绍

是指在同一时段大量的缓存key同时实现奥或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

如下图，正常请求应该是大部分到达Redis直接返回，少数请求到达数据库。可是如果突然之间大量的缓存key都失效了，比如一瞬间几万、几十万的缓存key都过期了，那么这些请求都会直接打到数据库，这个时候就可能导致数据库压力过大而出现服务宕机！

第二种情况更加可怕，刚才说的仅仅是部分key过期了，第二种情况是Redis直接宕机了，这个时候客户端的所有访问都会访问到数据库，这个时候数据库肯定完蛋了！



### 解决方案：

#### 1：给不同的Key的ttl添加随机值

这个解决方案是针对key同时失效情况的，比如我们在做缓存的时候，为了做缓存预热，我们可能提前把数据库中的数据导入Redis，即批量导入的，因为是批量同一时间导入Redis，TTL设置的是一样的话，很有可能超时时间一到缓存数据就同时过期了，那么就出现缓存雪崩，为了解决这个问题，我们在做缓存预热、批量导入数据的时候，我们可以给TTL的后边跟上一个随机数，就可以让key的过期时间分散在一个时间段内了，而不是一起失效。

#### 2：利用Redis集群提高服务的可用性

这个方案针对Redis宕机，就是提高Redis的高可用性，要想提高Redis高可用性，就必须借助Redis集群、Redis哨兵机制，Redis哨兵可以实现服务的监控，比如说Redis集群中，有一台主Redis宕机，我们的哨兵可以自动地在从机中选举出一台替代宕机的主机，这样就可以确保Redis一直正常地对外提供服务。而且主从还能实现数据的同步，如果说主机宕机了，从机还有主机的完整数据不会导致数据的丢失，这样就可以很大程度上保证Redis的高可用。后面Redis高级部分带你深入Redis集群各种骚操作！

#### 3：给缓存业务添加降级限流策略

比如说出现了人们无法抗拒的超级严重的事故，比方说整个机房挂了，结果整个Redis集群都完蛋了，那么Redis的可用性也就没有了，此时我们就可以给服务添加降级限流的策略。就是说提前做好一些容错处理，当我们发现Redis出现故障时，我们应该及时地去做服务降级，比如说快速失败，拒绝服务，而不是把请求继续压到数据库上去。即牺牲部分服务，保护数据库。

#### 4：给业务添加多级缓存

给业务添加多级缓存，什么是多级缓存呢？举个例子，我们开始讲的，我们的缓存使用场景是多种多样的，不仅仅可以在应用层添加缓存，请求从浏览器发出，浏览器是可以添加缓存的，但是浏览器缓存一般缓存的是静态数据，对于一些需要从数据库查询的动态数据是无法做缓存的，对于这部分动态缓存我们还可以在反向代理服务器Nginx层面做缓存，Nginx未命中再去找Redis，Redis再未命中再到达JVM，我们可以再JVM内部建立本地缓存，最后是数据库，就是在多个层面建立缓存。这样Redis这一环崩了，还有很多别的缓存可以去弥补，相当防弹衣穿了多件！所以通过多级缓存的方式就可以有效避免这种Redis雪崩导致的问题了！

多级缓存方案往往在大型互联网项目中会用到，从而应对亿级的并发场景。

## 缓存击穿

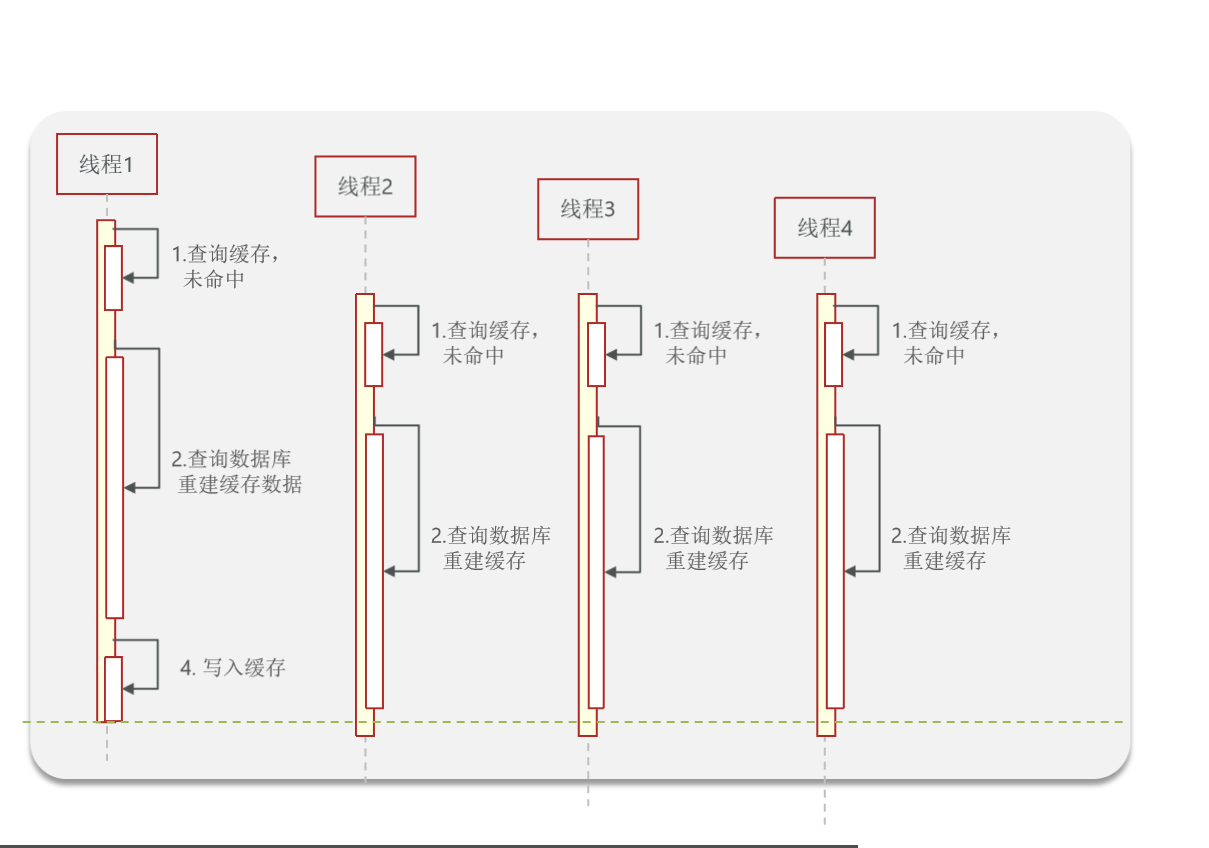
### 缓存击穿介绍

如果说缓存击穿是无数的key同时失效而引发的问题，缓存击穿就是部分key过期而引发的严重后果。为什么部分key或者说少数Key过期也会引起问题呢？这是因为这部分key不是普通的key。

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**缓存重现业务较复杂**的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击，可能就把数据库整垮了。

高并发访问：很好理解，即这个Key访问地非常非常多，可能是正在做活动的某一件商品的缓存，同一时刻就可能有无数的请求来访问这一个key。

缓存重建比较复杂：何谓缓存重建，我们的缓存在Redis中存储着，到达一定时间就会被清除，缓存就会失效，失效以后就需要重新从数据库中查询写入Redis。我们在实际开发中，从数据库中查询并且构建这个缓存数据并不一定就是查到什么就直接往Redis里存什么。有的时候有的业务比较复杂，我们需要从多个数据库中的表进行查询，甚至要做各种各样的表关联运算，最终得到结果给他缓存起来，这样一来这一个业务耗时就比较长，比如达到几十上百，甚至数百毫秒，这样一来在这么长的一段时间段内，我们的Redis等于一直都没有缓存，那在这一段时间内，无数请求来都无法命中，都会访问数据库。

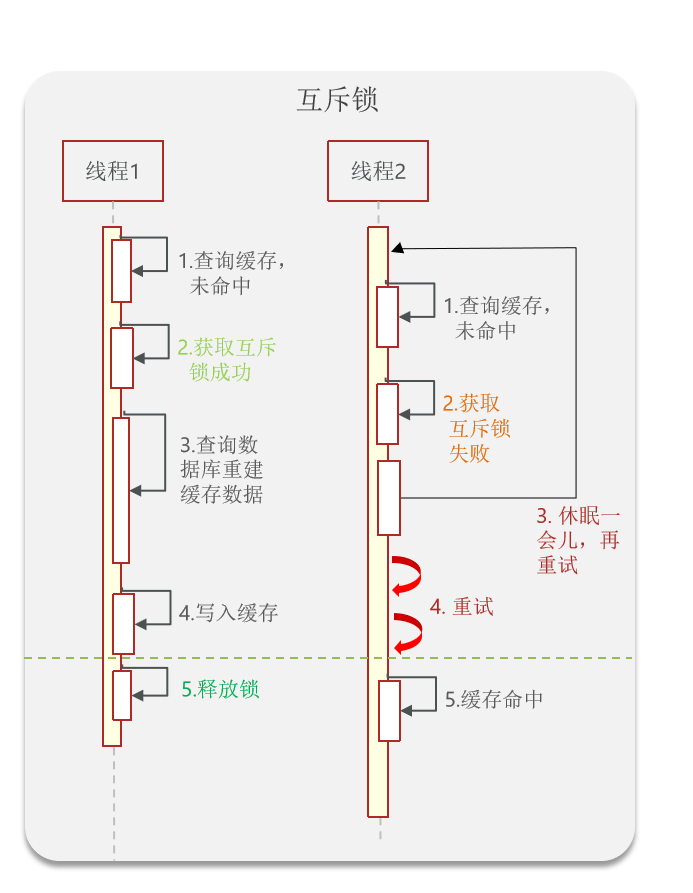


### 解决方法：

#### 1：互斥锁

因为有无数的请求进来做缓存重建，那么就可以用锁的方式，不让那么多请求进来做缓存重建，只让一个请求去做缓存重建就行。

缺点：就是等待，缓存重建时间内涌入的请求，需要等待缓存重建完成，性能比较差。

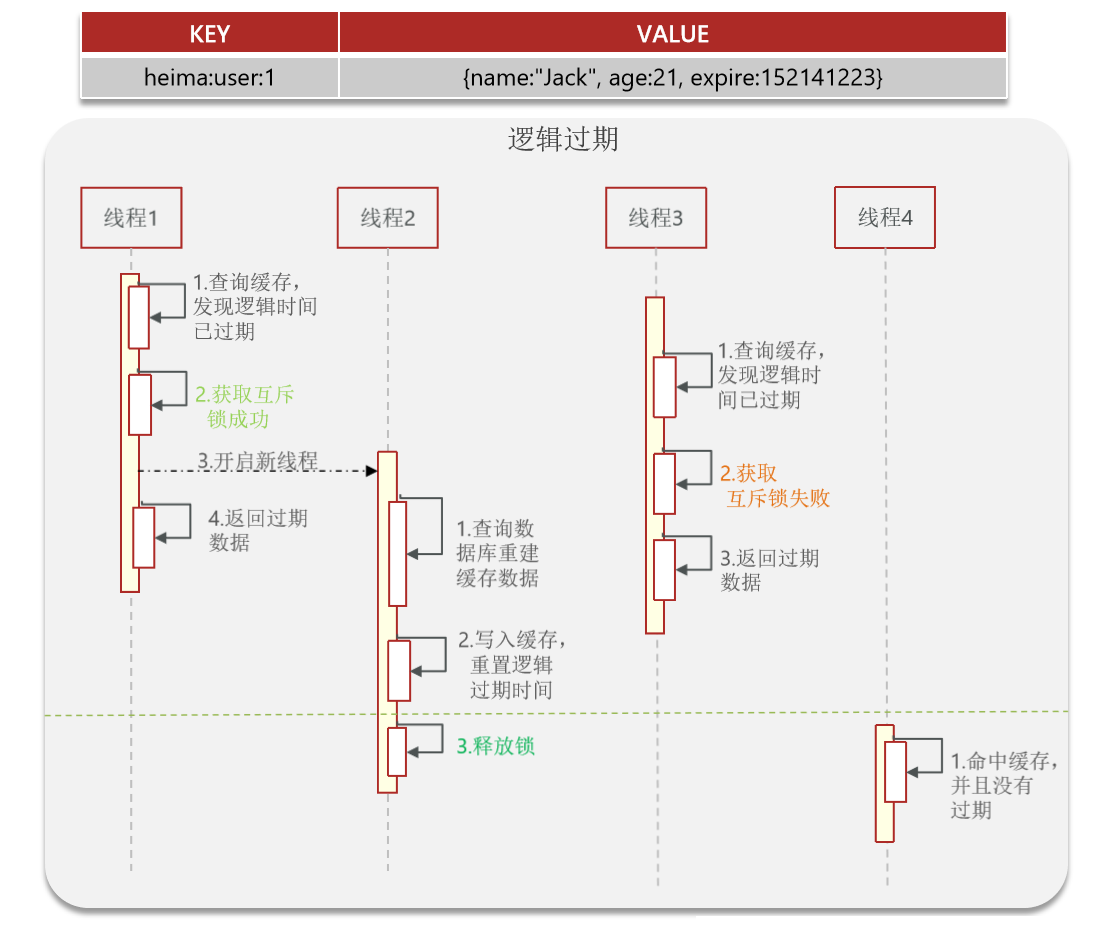


#### 2：逻辑过期：

之前产生缓存击穿的原因是什么？不就是因为缓存数据设置了TTL，突缓存突然间失效了，因为失效了，所以未命中，所以要重建，所以出现问题了。

逻辑过期顾名思义不是真的过期，不给缓存数据设置TTL，而是在添加缓存数据中加入逻辑过期时间，既然这个key没有设置TTL，没有过期时间，因此任何请求进来，理论上都是可以命中的，唯一需要判断的就是逻辑上有没有过期。

这些热点key往往都是在做活动的时候添加进Redis，给数据设置逻辑过期时间，活动结束再移除就ok了。



### 互斥锁 与 逻辑过期 比较

可以发现，经过对比，你可以发现，这两种方案都在解决我们缓存重建这一段时间内产生的并发问题。这是分布式系统中经常面临的问题，你是要保证可用性呢？还是要保证一致性?



#### 互斥锁：

优点：没有额外的内存消耗（因为不需要维护过期字段）；保证一致性；代码实现简单

缺点：线程需要等待，性能受影响；牺牲可用性，有死锁的风险

死锁风险：假设说在业务里有对多个缓存的查询需求，同样另外一个业务里也有对多个缓存的查询需求，这个时候着这个业务你拿到一把锁，你要获取另外一个缓存的锁的时候，结果另外一个缓存的锁在另外一个业务里，于是就出现互相等待死锁了。

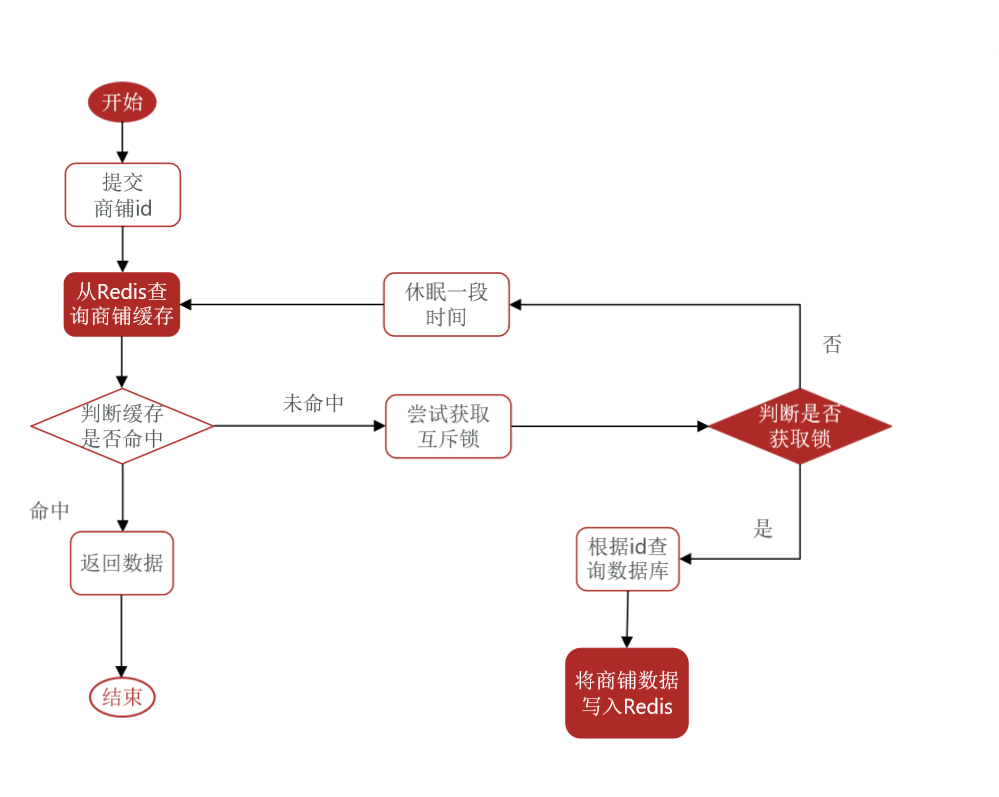
#### 逻辑过期：

优点：线程无需等待，性能较好；保证可用性

缺点：不保证一致性；返回的是过期旧数据，有额外的内存消耗（因为需要维护过期字段）；代码实现复杂

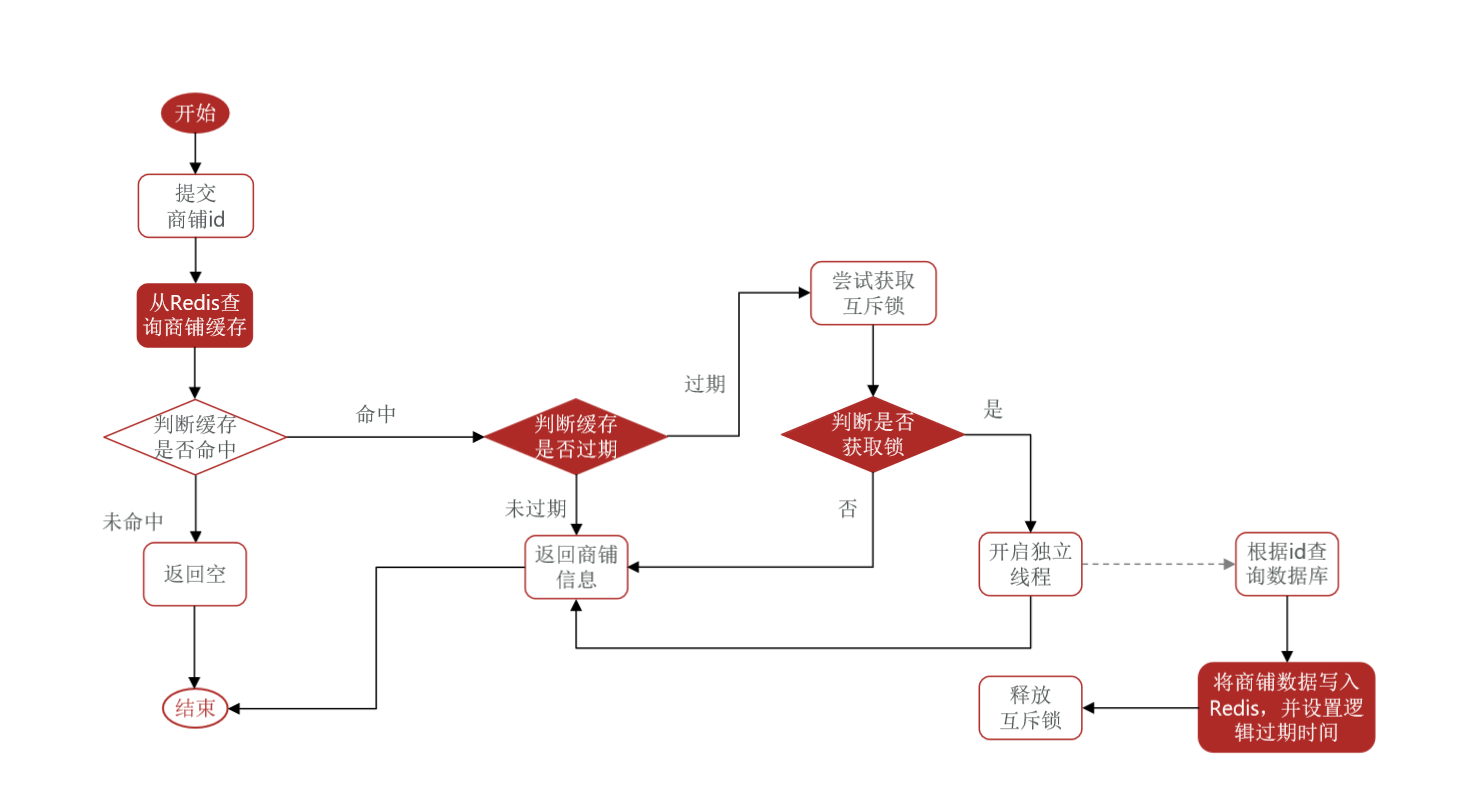
### 基于互斥锁解决缓存击穿问题

需求：修改根据id查询商铺的业务，基于互斥锁方式来解决缓存击穿问题。流程图如下：



### 基于逻辑过期解决缓存击穿问题

需求：修改根据id查询商铺的业务，基于逻辑过期方式来解决缓存击穿问题，流程图如下：



## 缓存工具类封装

基于StringRedisTemplate封装一个缓存工具类，满足下列需求：

方法1：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间

方法2：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，用于处理缓存击穿问题

方法3：根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题

方法4：根据指定的key查询缓存，并反序列化为指定类型，需要利用逻辑过期解决缓存击穿问题