1、缓存模型和思路

标准的操作方式就是查询数据库之前先查询缓存，如果缓存数据存在，则直接从缓存中返回，如果缓存数据不存在，再查询数据库，然后将数据存入redis。

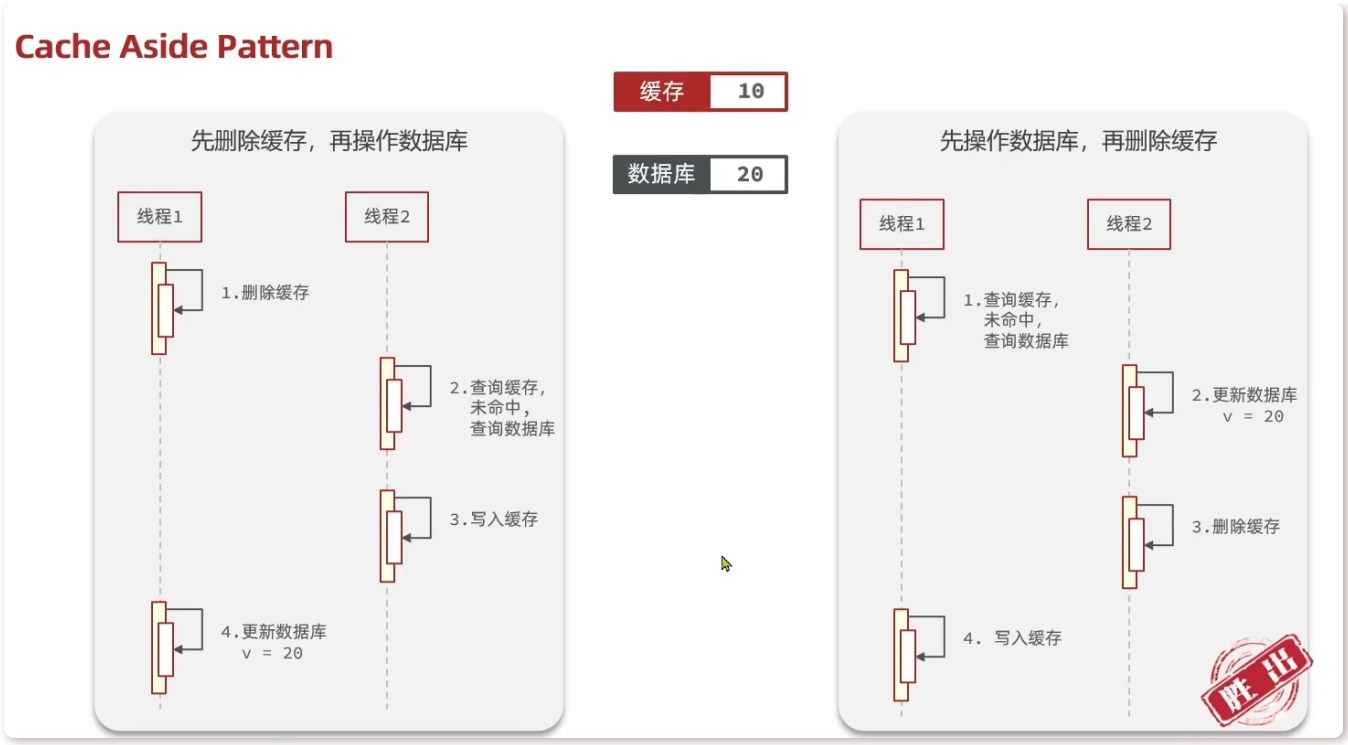
1.1、缓存更新策略

假设我们每次操作数据库后，都操作缓存，但是中间如果没有人查询，那么这个更新动作实际上只有最后一次生效，中间的更新动作意义并不大，我们可以把缓存删除，等待再次查询时，将缓存中的数据加载出来

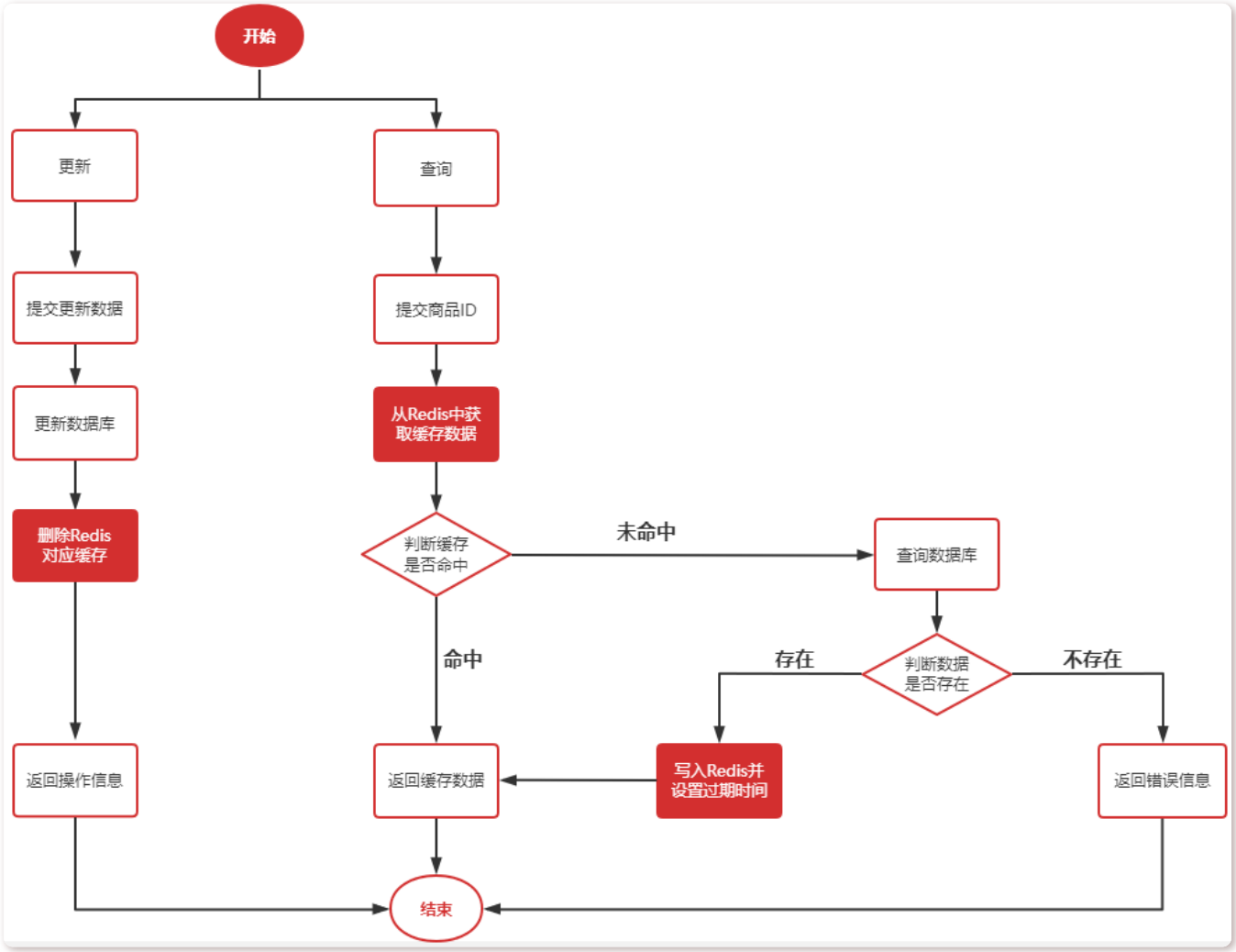
- 删除缓存还是更新缓存?
    - 更新缓存：每次更新数据库都更新缓存，无效写操作较多
    - 删除缓存：更新数据库时让缓存失效，查询时再更新缓存
- 如何保证缓存与数据库的操作的同时成功或失败?
    - 单体系统，将缓存与数据库操作放在一个事务
    - 分布式系统，利用TCC等分布式事务方案

应该具体操作缓存还是操作数据库，我们应当是**先操作数据库，再删除缓存**，原因在于，如果你选择第一种方案，在两个线程并发来访问时，假设线程1先来，他先把缓存删了，此时线程2过来，他查询缓存数据并不存在，此时他写入缓存，当他写入缓存后，线程1再执行更新动作时，实际上写入的就是旧的数据，新的数据被旧数据覆盖了。

- 先操作缓存还是先操作数据库?
    - 先删除缓存，再操作数据库
    - 先操作数据库，再删除缓存



1.2、具体实现思路



目前存在**缓存穿透、缓存雪崩和缓存击穿**问题

2、缓存穿透问题

2.1、方案分析

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**，这些请求都会打到数据库。

常见的解决方案有两种：

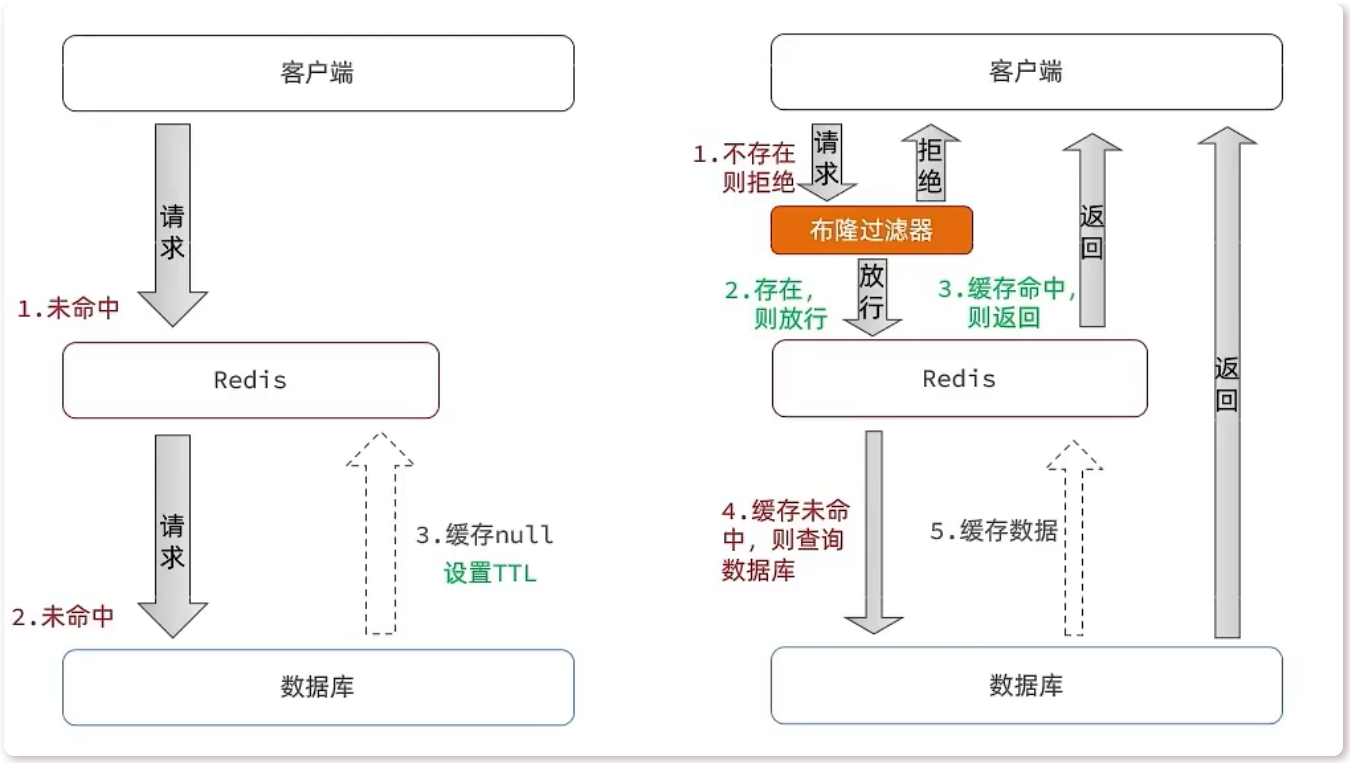
- **缓存空对象**
    - 优点：实现简单，维护方便
    - 缺点：
        - 额外的内存消耗
        - 可能造成短期的不一致
- **布隆过滤**
    - 优点：内存占用较少，没有多余key
    - 缺点：
        - 实现复杂
        - 存在误判可能

缓存空对象思路分析：当我们客户端访问不存在的数据时，先请求redis，但是此时redis中没有数据，此时会访问到数据库，但是数据库中也没有数据，这个数据穿透了缓存，直击数据库，我们都知道数据库能够承载的并发不如redis这么高，如果大量的请求同时过来访问这种不存在的数据，这些请求就都会访问到数据库，简单的解决方案就是哪怕这个数据在数据库中也不存在，我们也把这个数据存入到redis中去，这样，下次用户过来访问这个不存在的数据，那么在redis中也能找到这个数据就不会进入到缓存了

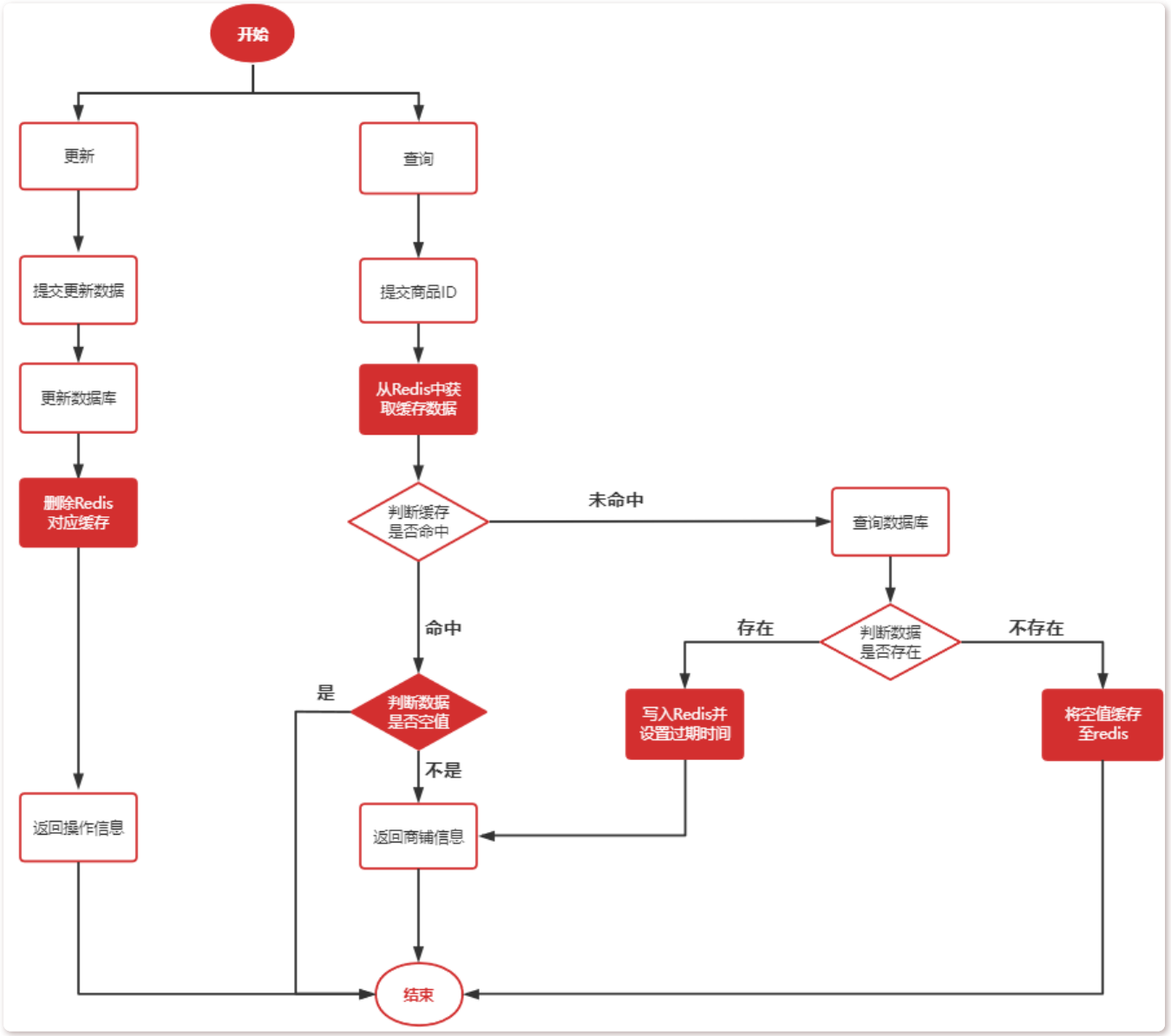
布隆过滤：布隆过滤器其实采用的是**哈希思想**来解决这个问题，通过一个**庞大的二进制数组**，走哈希思想去判断当前这个要查询的这个数据是否存在，如果布隆过滤器判断存在，则放行，这个请求会去访问redis，哪怕此时redis中的数据过期了，但是数据库中一定存在这个数据，在数据库中查询出来这个数据后，再将其放入到redis中，

假设布隆过滤器判断这个数据不存在，则直接返回

这种方式优点在于**节约内存空间**，**存在误判**，误判原因在于：布隆过滤器走的是哈希思想，只要哈希思想，就可能存在**哈希冲突**



2.2、缓存空对象实现思路



2.3、小总结

缓存穿透产生的原因是什么?

- 用户请求的数据在缓存中和数据库中都不存在，不断发起这样的请求，给数据库带来巨大压力

缓存穿透的解决方案有哪些?

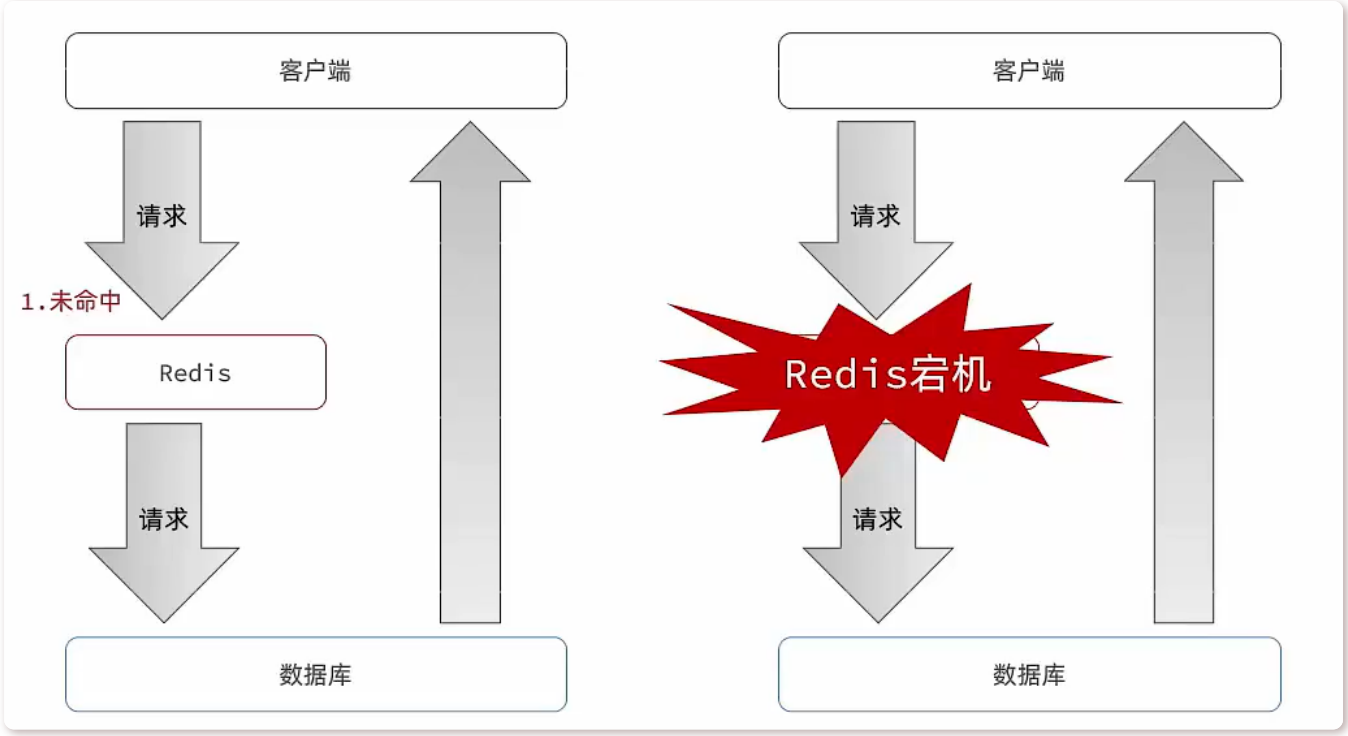
- 缓存null值
- 布隆过滤
- 增强id的复杂度，避免被猜测id规律
- 做好数据的基础格式校验
- 加强用户权限校验
- 做好热点参数的限流

3、缓存雪崩

缓存雪崩是指在**同一时段大量的缓存key同时失效或者Redis服务宕机**，导致大量请求到达数据库，带来巨大压力。

解决方案:

- 给不同的Key的TTL添加随机值
- 利用Redis集群提高服务的可用性
- 给缓存业务添加降级限流策略
- 给业务添加多级缓存



4、缓存击穿

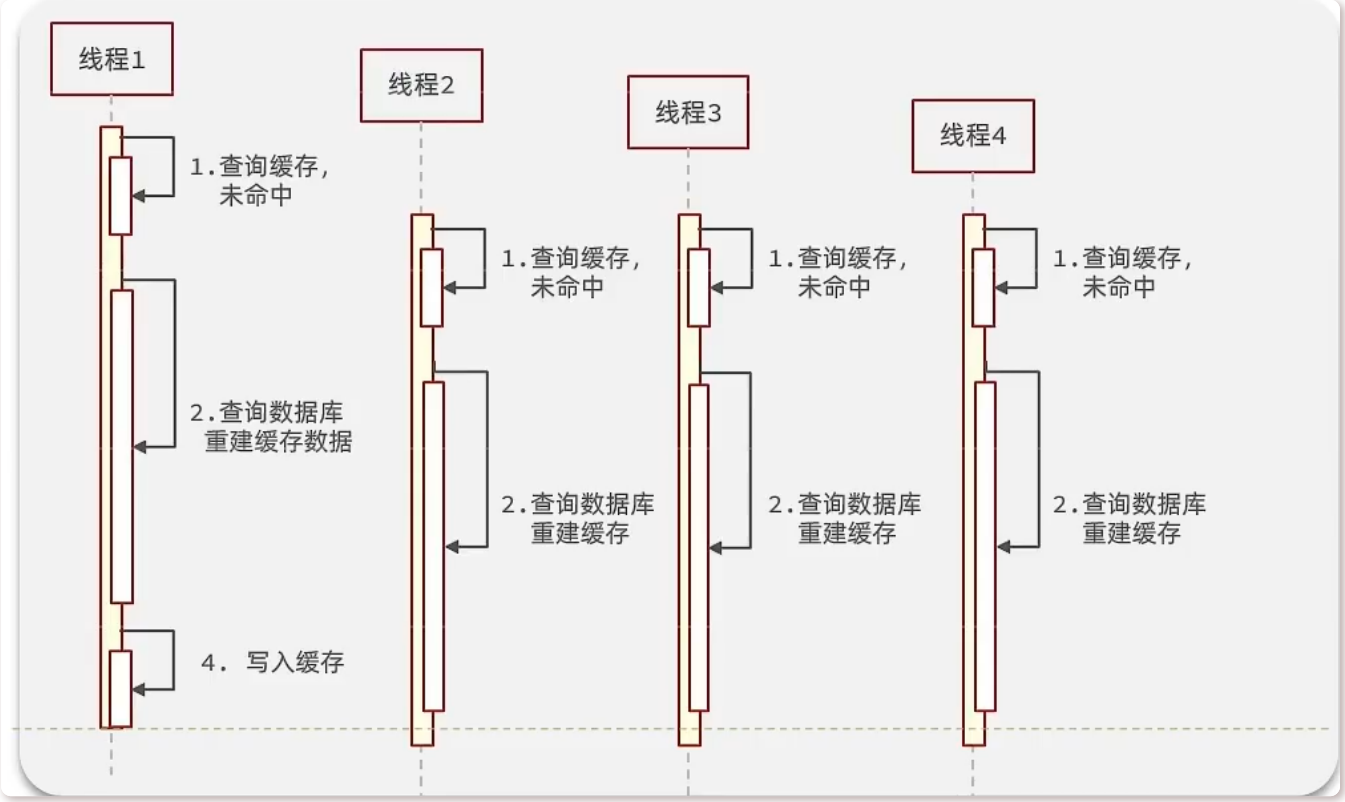
4.1、方案分析

缓存击穿问题也叫热点Key问题，就是一个**被高并发访问并且缓存重建业务较复杂的key**突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

常见的解决方案有两种：

- 互斥锁
- 逻辑过期

逻辑分析：假设线程1在查询缓存之后，本来应该去查询数据库，然后把这个数据重新加载到缓存的，此时只要线程1走完这个逻辑，其他线程就都能从缓存中加载这些数据了，但是假设**在线程1没有走完的时候**，后续的线程2，线程3，线程4同时过来访问当前这个方法， 那么这些线程都不能从缓存中查询到数据，那么他们就会同一时刻来访问查询缓存，都没查到，接着**同一时间去访问数据库**，同时的去执行数据库代码，对数据库访问压力过大

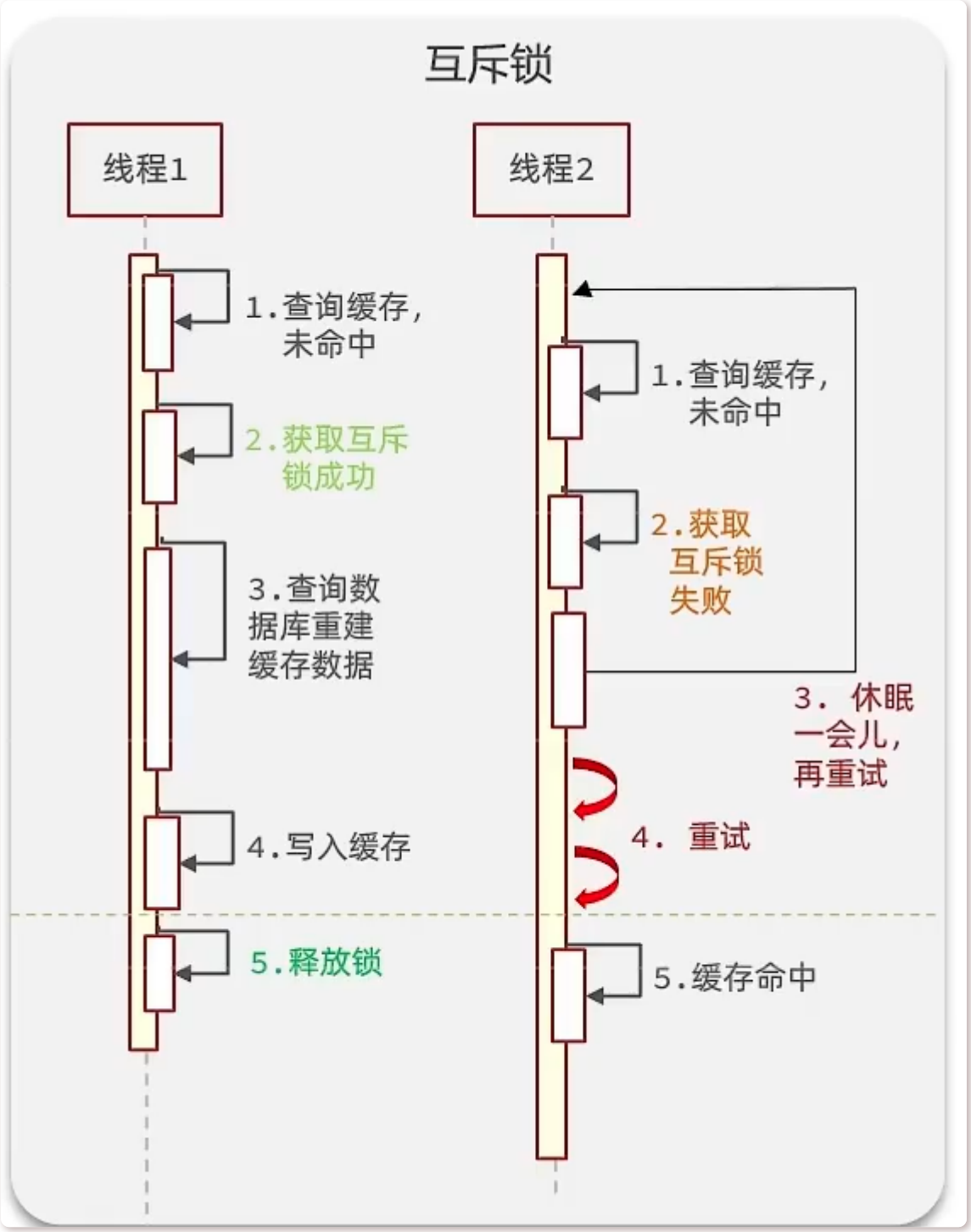


4.1.1、互斥锁

因为锁能实现互斥性。假设线程过来，只能一个人一个人的来访问数据库，从而避免对于数据库访问压力过大，但这也会影响查询的性能，因为此时会让查询的性能从并行变成了串行，我们可以采用`tryLock方法 + double check`来解决这样的问题。

假设现在线程1过来访问，他查询缓存没有命中，但是此时他获得到了锁的资源，那么线程1就会一个人去执行逻辑，假设现在线程2过来，线程2在执行过程中，并没有获得到锁，那么线程2就可以进行到休眠，直到线程1把锁释放后，线程2获得到锁，然后再来执行逻辑，此时就能够从缓存中拿到数据了。

> 存在阻塞问题，存在死锁的隐患。



4.1.2、逻辑过期

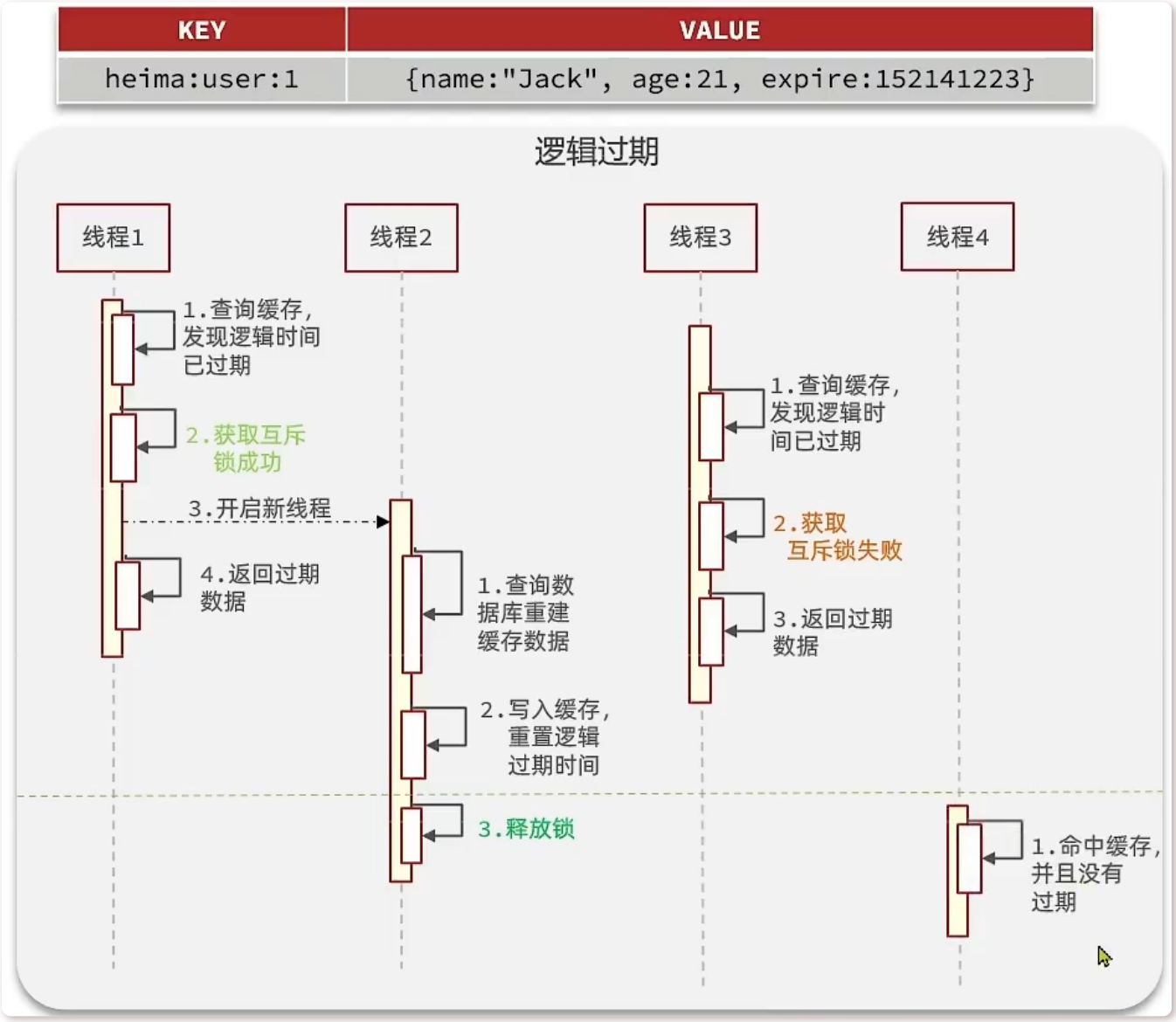
我们之所以会出现这个缓存击穿问题，**主要原因**是在于我们**对key设置了过期时间**，假设我们不设置过期时间，其实就不会有缓存击穿的问题，但是不设置过期时间，这样数据不就一直占用我们内存了吗，我们可以采用逻辑过期方案。

我们把**过期时间设置在 redis的value**中，注意：这个过期时间**并不会直接作用于redis**，而是我们后续通过逻辑去处理。假设：

1. 线程1去查询缓存；
2. 然后从value中判断出来当前的数据已经过期了；
3. 线程1去获得互斥锁，那么其他线程会进行阻塞；
4. 获得了锁的线程会开启一个线程去进行以前的重构数据的逻辑，直到新开的线程完成这个逻辑后，才释放锁；

5．而线程1直接进行返回；
6．假设现在线程3过来访问，由于线程线程2持有着锁，所以线程3无法获得锁，线程3也直接返回数据；
7．只有等到新开的线程2把重建数据构建完后，其他线程才能走返回正确的数据。

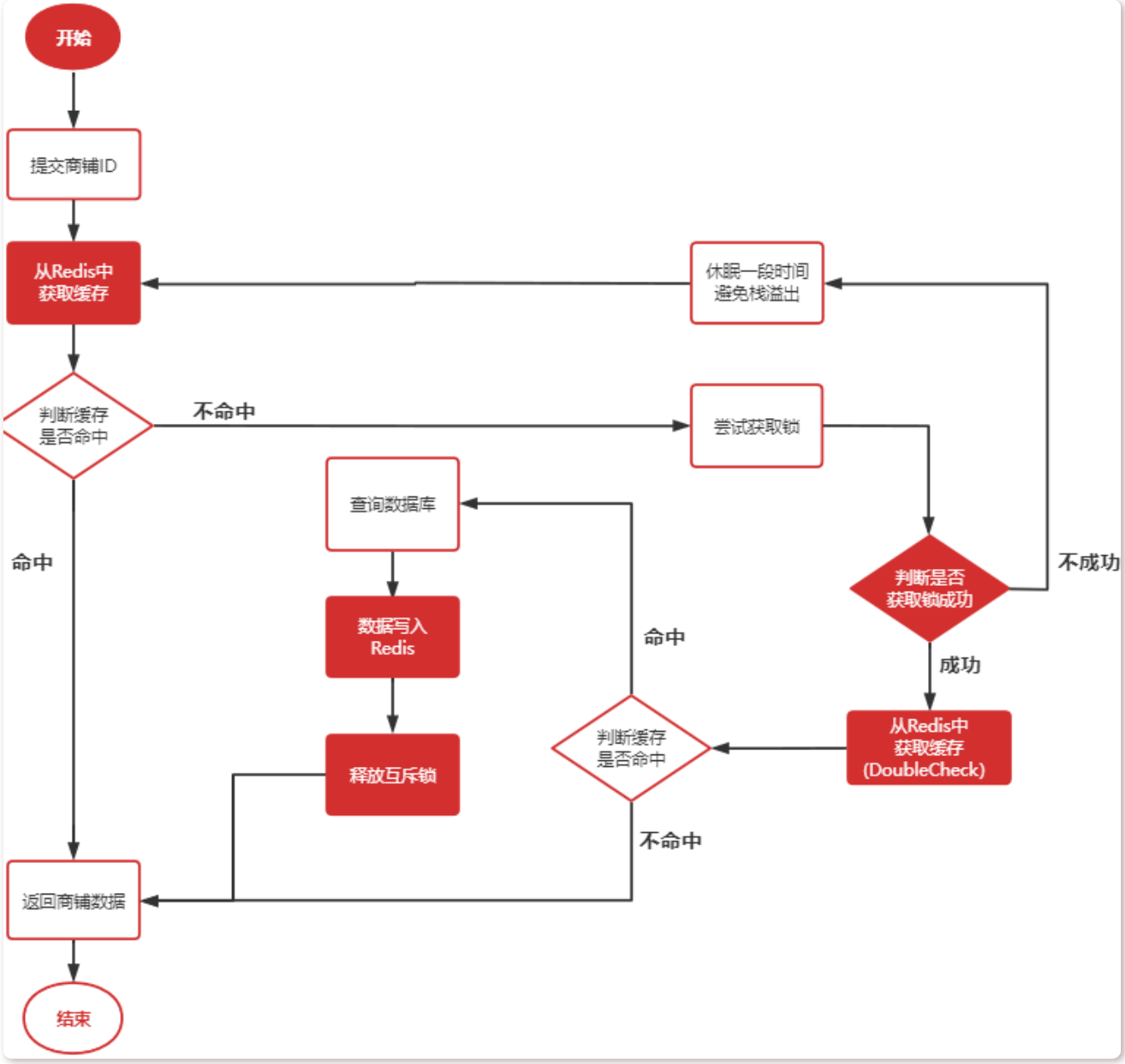
这种方案巧妙在于，异步的构建缓存，缺点在于在**构建完缓存之前**，**返回的都是脏数据**。



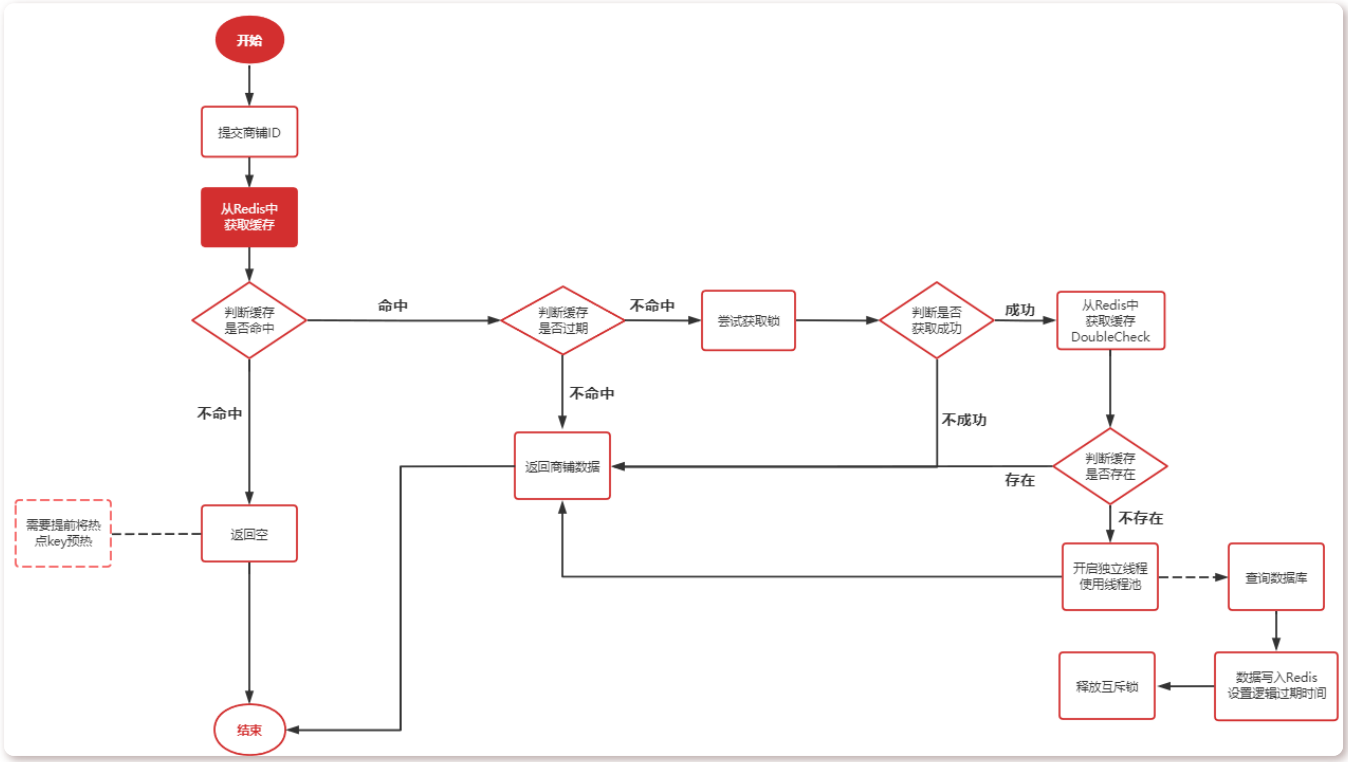
4.1.3、方案对比

解决方案	优点	缺点
互斥锁	• 没有额外的内存消耗 • 保证一致性 • 实现简单	• 线程需要等待，性能受影响 • 可能有死锁风险
逻辑过期	• 线程无需等待，性能较好	• 不保证一致性 • 有额外内存消耗 • 实现复杂

4.2、互斥锁实现思路



4.3、逻辑过期实现思路


开始
提交商铺ID
从Redis中
获取缓存
判断缓存
是否命中
命中
判断缓存
是否过期
不命中
尝试获取锁
判断是否
获取成功
成功
从Redis中
获取缓存
DoubleCheck
不命中
不命中
不成功
返回商铺数据
判断缓存
是否存在
存在
不存在
需要提前将热
点key预热
返回空
开启独立线程
使用线程池
查询数据库
释放互斥锁
数据写入Redis
设置逻辑过期时间
结束