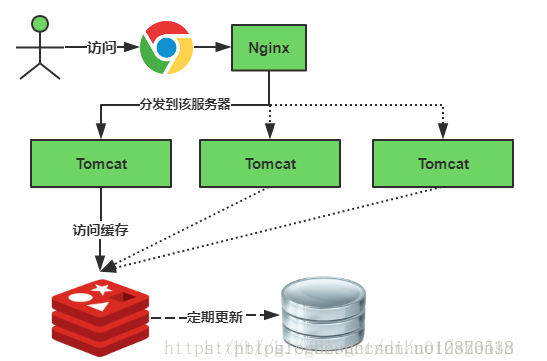
关于Redis经常被问到的问题：缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级、缓存热点 key等概念的入门及简单解决方案。

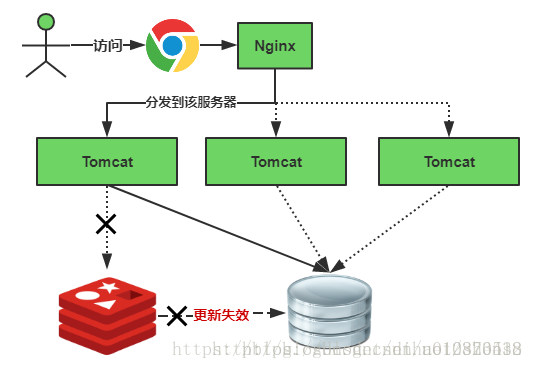
**一、缓存雪崩**

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

缓存正常从Redis中获取，示意图如下：



缓存失效瞬间示意图如下：



缓存失效时的雪崩效应对底层系统的冲击非常可怕！大多数系统设计者考虑用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开，比如我们可以在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。

* 事前：（高可用）尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。也就是搭建的集群。（数据预热）数据加热的含义就是在正式部署之前，我先把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中。在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。
* 事中：（**限流降级**）本地ehcache缓存 + hystrix限流&降级，避免MySQL崩掉，通过加锁或者队列来控制读数据库写缓存的线程数量。
* 事后：利用 redis 持久化机制保存的数据尽快恢复缓存

以下简单介绍两种实现方式的伪代码：

（1）碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队，伪代码如下：

//伪代码

[复制代码](javascript:void(0);)

public object GetProductListNew() { i

nt cacheTime = 30;

String cacheKey = "product\_list";

String lockKey = cacheKey;

String cacheValue = CacheHelper.get(cacheKey);

if (

cacheValue != null) { return cacheValue;

} else {

synchronized(lockKey) {

cacheValue = CacheHelper.get(cacheKey);

if (cacheValue != null) {

return cacheValue;

} else {

//这里一般是sql查询数据

cacheValue = GetProductListFromDB();

CacheHelper.Add(cacheKey, cacheValue, cacheTime);

} }

return cacheValue; } }

[复制代码](javascript:void(0);)

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

（2）还有一个解决办法解决方案是：给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存，实例伪代码如下：

//伪代码 public object GetProductListNew() { int cacheTime = 30; String cacheKey = "product\_list"; //缓存标记 String cacheSign = cacheKey + "\_sign"; String sign = CacheHelper.Get(cacheSign); //获取缓存值 String cacheValue = CacheHelper.Get(cacheKey); if (sign != null) { return cacheValue; //未过期，直接返回 } else { CacheHelper.Add(cacheSign, "1", cacheTime); ThreadPool.QueueUserWorkItem((arg) -> { //这里一般是 sql查询数据 cacheValue = GetProductListFromDB(); //日期设缓存时间的2倍，用于脏读 CacheHelper.Add(cacheKey, cacheValue, cacheTime \* 2); }); return cacheValue; } }

解释说明：

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际key的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长1倍，例：标记缓存时间30分钟，数据缓存设置为60分钟。 这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为key设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

**二、缓存穿透**

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。如果这样的请求短时间内非常多可能会压垮数据库。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

//伪代码 public object GetProductListNew() { int cacheTime = 30; String cacheKey = "product\_list"; String cacheValue = CacheHelper.Get(cacheKey); if (cacheValue != null) { return cacheValue; } cacheValue = CacheHelper.Get(cacheKey); if (cacheValue != null) { return cacheValue; } else { //数据库查询不到，为空 cacheValue = GetProductListFromDB(); if (cacheValue == null) { //如果发现为空，设置个默认值，也缓存起来 cacheValue = string.Empty; } CacheHelper.Add(cacheKey, cacheValue, cacheTime); return cacheValue; } }

把空结果，也给缓存起来，这样下次同样的请求就可以直接返回空了，即可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值，对要查询的key进行预先校验，然后再放行给后面的正常缓存处理逻辑。

**三．缓存击穿**

缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

处理方式：1.设置永不过期。

2. 使用互斥锁。单机，可以用synchronized或者lock来处理，如果是分布式环境可以用分布式锁就可以了（分布式锁，可以用memcache的add, redis的setnx, zookeeper的添加节点操作）。

**三、缓存预热**

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

2、数据量不大，可以在项目启动的时候自动进行加载；

3、定时刷新缓存；

**四、缓存更新**

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

**五、缓存降级**

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

**六、缓存热点 key**

使用缓存 + 过期时间的策略既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。但是有两个问题如果同时出现，可能就会对应用造成致命的危害：

1. 当前 key 是一个热点 key( 可能对应应用的热卖商品、热点新闻、热点评论等），并发量非常大。
2. 重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的 SQL、多次 IO、多个依赖等。

在缓存失效的瞬间，有大量线程来重建缓存 ( 如下图)，造成后端负载加大，甚至可能会让应用崩溃。

热点 key 失效后大量线程重建缓存

要解决这个问题也不是很复杂，但是不能为了解决这个问题给系统带来更多的麻烦，所以需要制定如下目标：

* 减少重建缓存的次数
* 数据尽可能一致
* 较少的潜在危险

**1）互斥锁 (mutex key)**

此方法只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可，整个过程如图 ,使用互斥锁重建缓存

下面代码使用 Redis 的 setnx 命令实现上述功能，伪代码：

String get(String key) { //从redis中获取key String value = redis.get(key); //如果value为空则开始重构缓存 if (value == null) { //只允许一个线程重构缓存，使用nx，并设置过期时间ex String mutexKey = "mutex:key" + key; if (redis.set(mutexKey, "1", "ex 180", "nx")) { //从数据源获取数据 value = db.get(key); //回写redis并设置过期时间 redis.set(key, value, timeout); //删除mutexKey redis.del(mutexKey); } else { //其他线程睡眠50秒再重试 Thread.sleep(50); get(key); } } return value; }

* 从 Redis 获取数据，如果值不为空，则直接返回值。
* 如果 set(nx 和 ex) 结果为 true，说明此时没有其他线程重建缓存，那么当前线程执行缓存构建逻辑。
* 如果 setnx(nx 和 ex) 结果为 false，说明此时已经有其他线程正在执行构建缓存的工作，那么当前线程将休息指定时间 (例如这里是 50 毫秒，取决于构建缓存的速度 ) 后，重新执行函数，直到获取到数据。

**2）永远不过期**

永远不过期”包含两层意思：

从缓存层面来看，确实没有设置过期时间，所以不会出现热点 key 过期后产生的问题，也就是“物理”不过期。

从功能层面来看，为每个 value 设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

” 永远不过期 ” 策略，整个过程如下图所示：

从实战看，此方法有效杜绝了热点 key 产生的问题，但唯一不足的就是重构缓存期间，会出现数据不一致的情况，这取决于应用方是否容忍这种不一致。下面代码使用 Redis 进行模拟：

String get(final String key) { V v = redis.get(key); String value = v.getValue(); //逻辑过期时间 final Long logicTimeout = v.getLogicTimeout(); //如果逻辑时间小于当前时间，开始重建缓存 if (logicTimeout <= System.currentTimeMillis()) { final String mutexKey = "mutex:key" + key; if (redis.set(mutexKey, "1", "ex 180", "nx")) { //重建缓存 threadPool.execute(new Runnable() { @Override public void run() { String dbValue = db.get(key); redis.set(key, (dbValue, newLogicTimeout)); redis.del(mutexKey); } }); } } return value; }

作为一个并发量较大的应用，在使用缓存时有三个目标：第一，加快用户访问速度，提高用户体验。第二，降低后端负载，减少潜在的风险，保证系统平稳。第三，保证数据“尽可能”及时更新。下面将按照这三个维度对上述两种解决方案进行分析。

互斥锁 (mutex key)：这种方案思路比较简单，但是存在一定的隐患，如果构建缓存过程出现问题或者时间较长，可能会存在死锁和线程池阻塞的风险，但是这种方法能够较好的降低后端存储负载并在一致性上做的比较好。

” 永远不过期 “：这种方案由于没有设置真正的过期时间，实际上已经不存在热点 key 产生的一系列危害，但是会存在数据不一致的情况，同时代码复杂度会增大

# 七; redis缓存与数据库一致性问题解决

读数据一般是先读redis，命中返回，未命中查数据库，再插入redis，这种一般没问题

更新删除数据有两种，先删redis，再更新db，第二种先更新db，再删除redis。

###### 更新缓存VS淘汰缓存

     答：更新缓存很直接，但是涉及到本次更新的数据结果需要一堆数据运算（例如更新用户余额，可能需要先看看有没有优惠券等），复杂度就增加了。而淘汰缓存仅仅会增加一次cache miss，代价可以忽略，所以建议淘汰缓存

但是在更新缓存方面，对于更新完数据库，是更新缓存呢，还是删除缓存。又或者是先删除缓存，再更新数据库，其实大家存在很大的争议。目前没有一篇全面的博客，对这几种方案进行解析。于是博主战战兢兢，顶着被大家喷的风险，写了这篇文章。

### 文章结构

本文由以下三个部分组成  
1、讲解缓存更新策略  
2、对每种策略进行缺点分析  
3、针对缺点给出改进方案

## 正文

先做一个说明，从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。这种方案下，我们可以对存入缓存的数据设置过期时间，所有的写操作以数据库为准，对缓存操作只是尽最大努力即可。也就是说如果数据库写成功，缓存更新失败，那么只要到达过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。因此，接下来讨论的思路不依赖于给缓存设置过期时间这个方案。  
在这里，我们讨论**三种**更新策略：

1. 先更新数据库，再更新缓存
2. 先删除缓存，再更新数据库
3. 先更新数据库，再删除缓存

应该没人问我，为什么没有先更新缓存，再更新数据库这种策略。

### (1)先更新数据库，再更新缓存

这套方案，大家是普遍反对的。为什么呢？有如下两点原因。  
**原因一（线程安全角度）**  
同时有请求A和请求B进行更新操作，那么会出现  
（1）线程A更新了数据库  
（2）线程B更新了数据库  
（3）线程B更新了缓存  
（4）线程A更新了缓存  
这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据，因此不考虑。  
**原因二（业务场景角度）**  
有如下两点：  
（1）如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。  
（2）如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

接下来讨论的就是争议最大的，先删缓存，再更新数据库。还是先更新数据库，再删缓存的问题。

### (2)先删缓存，再更新数据库

该方案会导致不一致的原因是。同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么会出现如下情形:  
（1）请求A进行写操作，删除缓存  
（2）请求B查询发现缓存不存在  
（3）请求B去数据库查询得到旧值  
（4）请求B将旧值写入缓存  
（5）请求A将新值写入数据库  
上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。  
那么，**如何解决呢？采用延时双删策略**  
伪代码如下

public void write(String key,Object data){

redis.delKey(key);

db.updateData(data);

Thread.sleep(1000);

redis.delKey(key);

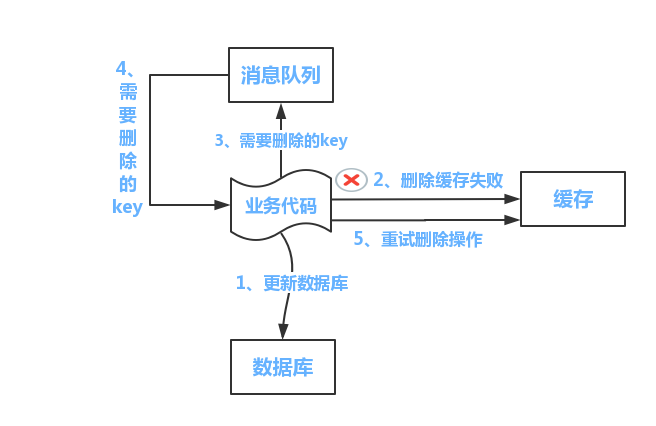
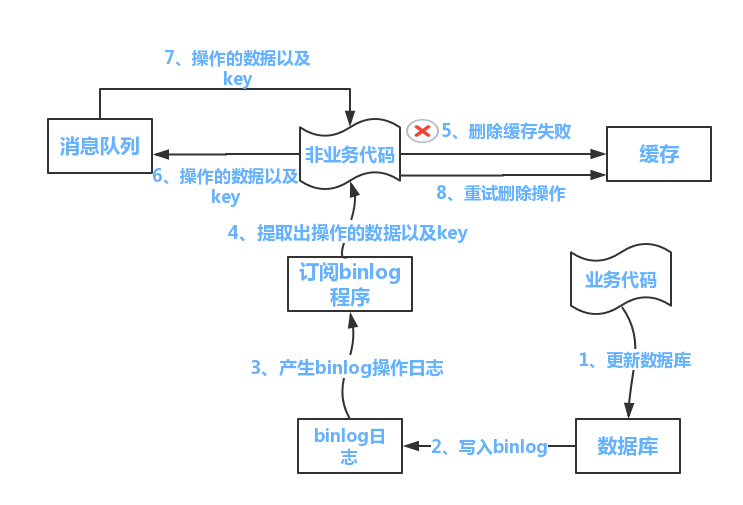
}

转化为中文描述就是  
（1）先淘汰缓存  
（2）再写数据库（这两步和原来一样）  
（3）休眠1秒，再次淘汰缓存  
这么做，可以将1秒内所造成的缓存脏数据，再次删除。  
**那么，这个1秒怎么确定的，具体该休眠多久呢？**  
针对上面的情形，读者应该自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。  
**如果你用了mysql的读写分离架构怎么办？**  
ok，在这种情况下，造成数据不一致的原因如下，还是两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。  
（1）请求A进行写操作，删除缓存  
（2）请求A将数据写入数据库了，  
（3）请求B查询缓存发现，缓存没有值  
（4）请求B去从库查询，这时，还没有完成主从同步，因此查询到的是旧值  
（5）请求B将旧值写入缓存  
（6）数据库完成主从同步，从库变为新值  
上述情形，就是数据不一致的原因。还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。  
**采用这种同步淘汰策略，吞吐量降低怎么办？**  
ok，那就将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，再返回。这么做，加大吞吐量。  
**第二次删除,如果删除失败怎么办？**  
这是个非常好的问题，因为第二次删除失败，就会出现如下情形。还是有两个请求，一个请求A进行更新操作，另一个请求B进行查询操作，为了方便，假设是单库：  
（1）请求A进行写操作，删除缓存  
（2）请求B查询发现缓存不存在  
（3）请求B去数据库查询得到旧值  
（4）请求B将旧值写入缓存  
（5）请求A将新值写入数据库  
（6）请求A试图去删除请求B写入对缓存值，结果失败了。  
ok,这也就是说。如果第二次删除缓存失败，会再次出现缓存和数据库不一致的问题。  
**如何解决呢？**  
具体解决方案，且看博主对第(3)种更新策略的解析。

### (3)先更新数据库，再删缓存

首先，先说一下。老外提出了一个缓存更新套路，名为[《Cache-Aside pattern》](https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside)。其中就指出

* **失效**：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。
* **命中**：应用程序从cache中取数据，取到后返回。
* **更新**：先把数据存到数据库中，成功后，再让缓存失效。

另外，知名社交网站facebook也在论文[《Scaling Memcache at Facebook》](https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf)中提出，他们用的也是先更新数据库，再删缓存的策略。  
**这种情况不存在并发问题么？**  
不是的。假设这会有两个请求，一个请求A做查询操作，一个请求B做更新操作，那么会有如下情形产生  
（1）缓存刚好失效  
（2）请求A查询数据库，得一个旧值  
（3）请求B将新值写入数据库  
（4）请求B删除缓存  
（5）请求A将查到的旧值写入缓存  
ok，如果发生上述情况，确实是会发生脏数据。  
**然而，发生这种情况的概率又有多少呢？**  
发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。  
假设，有人非要抬杠，有强迫症，一定要解决怎么办？  
**如何解决上述并发问题？**  
首先，给缓存设有效时间是一种方案。其次，采用策略（2）里给出的异步延时删除策略，保证读请求完成以后，再进行删除操作。  
**还有其他造成不一致的原因么？**  
有的，这也是缓存更新策略（2）和缓存更新策略（3）都存在的一个问题，如果删缓存失败了怎么办，那不是会有不一致的情况出现么。比如一个写数据请求，然后写入数据库了，删缓存失败了，这会就出现不一致的情况了。这也是缓存更新策略（2）里留下的最后一个疑问。  
**如何解决？**  
提供一个保障的重试机制即可，这里给出两套方案。  
**方案一**：  
如下图所示  
  
流程如下所示  
（1）更新数据库数据；  
（2）缓存因为种种问题删除失败  
（3）将需要删除的key发送至消息队列  
（4）自己消费消息，获得需要删除的key  
（5）继续重试删除操作，直到成功  
然而，该方案有一个缺点，对业务线代码造成大量的侵入。于是有了方案二，在方案二中，启动一个订阅程序去订阅数据库的binlog，获得需要操作的数据。在应用程序中，另起一段程序，获得这个订阅程序传来的信息，进行删除缓存操作。  
**方案二**：  
  
流程如下图所示：  
（1）更新数据库数据  
（2）数据库会将操作信息写入binlog日志当中  
（3）订阅程序提取出所需要的数据以及key  
（4）另起一段非业务代码，获得该信息  
（5）尝试删除缓存操作，发现删除失败  
（6）将这些信息发送至消息队列  
（7）重新从消息队列中获得该数据，重试操作。

**备注说明：**上述的订阅binlog程序在mysql中有现成的中间件叫canal，可以完成订阅binlog日志的功能。至于oracle中，博主目前不知道有没有现成中间件可以使用。另外，重试机制，博主是采用的是消息队列的方式。Kafka或者rabitmq等。如果对一致性要求不是很高，直接在程序中另起一个线程，每隔一段时间去重试即可，这些大家可以灵活自由发挥，只是提供一个思路。