**缓存穿透，缓存击穿，缓存雪崩解决方案分析**

前言

设计一个缓存系统，不得不要考虑的问题就是：缓存穿透、缓存击穿与失效时的雪崩效应。

缓存穿透

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，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。

解决方案

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（我们采用的就是这种），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。

缓存雪崩

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。

解决方案

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缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。这里分享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。

缓存击穿

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：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

# 一、引出热点key问题

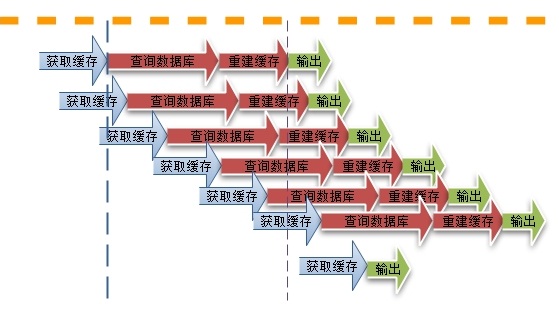
       我们通常使用 缓存 + 过期时间的策略来帮助我们加速接口的访问速度，减少了后端负载，同时保证功能的更新，一般情况下这种模式已经基本满足要求了。

       但是有两个问题如果同时出现，可能就会对系统造成致命的危害：

      (1) 这个key是一个热点key（例如一个重要的新闻，一个热门的八卦新闻等等），所以这种key访问量可能非常大。

      (2) 缓存的构建是需要一定时间的。（可能是一个复杂计算，例如复杂的sql、多次IO、多个依赖(各种接口)等等）

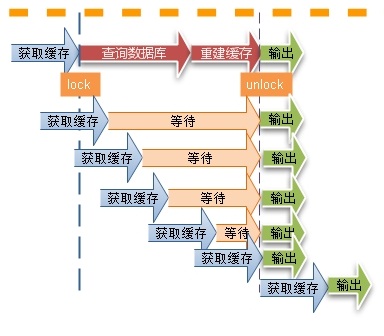
       于是就会出现一个致命问题：在缓存失效的瞬间，有大量线程来构建缓存（见下图），造成后端负载加大，甚至可能会让系统崩溃 。



二、四种解决方案（注释：第1,2种方法来自[Tim Yang博客](http://timyang.net/tag/mutex/)）

我们的目标是：尽量少的线程构建缓存(甚至是一个) + 数据一致性 + 较少的潜在危险，下面会介绍四种方法来解决这个问题：

1. 使用互斥锁(mutex key): 这种解决方案思路比较简单，就是只让一个线程构建缓存，其他线程等待构建缓存的线程执行完，重新从缓存获取数据就可以了（如下图）



     如果是单机，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）。

     下面是Tim yang博客的代码，是memcache的伪代码实现

**Java代码  [收藏代码](javascript:void())**

1. **if** (memcache.get(key) == **null**) {
2. // 3 min timeout to avoid mutex holder crash
3. **if** (memcache.add(key\_mutex, 3 \* 60 \* 1000) == **true**) {
4. value = db.get(key);
5. memcache.set(key, value);
6. memcache.delete(key\_mutex);
7. } **else** {
8. sleep(50);
9. retry();
10. }
11. }

      如果换成redis，就是：

**Java代码  [收藏代码](javascript:void())**

1. String get(String key) {
2. String value = redis.get(key);
3. **if** (value  == **null**) {
4. **if** (redis.setnx(key\_mutex, "1")) {
5. // 3 min timeout to avoid mutex holder crash
6. redis.expire(key\_mutex, 3 \* 60)
7. value = db.get(key);
8. redis.set(key, value);
9. redis.delete(key\_mutex);
10. } **else** {
11. //其他线程休息50毫秒后重试
12. Thread.sleep(50);
13. get(key);
14. }
15. }
16. }

2. "提前"使用互斥锁(mutex key)：

   在value内部设置1个超时值(timeout1), timeout1比实际的memcache timeout(timeout2)小。当从cache读取到timeout1发现它已经过期时候，马上延长timeout1并重新设置到cache。然后再从数据库加载数据并设置到cache中。伪代码如下：

**Java代码  [收藏代码](javascript:void())**

1. v = memcache.get(key);
2. **if** (v == **null**) {
3. **if** (memcache.add(key\_mutex, 3 \* 60 \* 1000) == **true**) {
4. value = db.get(key);
5. memcache.set(key, value);
6. memcache.delete(key\_mutex);
7. } **else** {
8. sleep(50);
9. retry();
10. }
11. } **else** {
12. **if** (v.timeout <= now()) {
13. **if** (memcache.add(key\_mutex, 3 \* 60 \* 1000) == **true**) {
14. // extend the timeout for other threads
15. v.timeout += 3 \* 60 \* 1000;
16. memcache.set(key, v, KEY\_TIMEOUT \* 2);
18. // load the latest value from db
19. v = db.get(key);
20. v.timeout = KEY\_TIMEOUT;
21. memcache.set(key, value, KEY\_TIMEOUT \* 2);
22. memcache.delete(key\_mutex);
23. } **else** {
24. sleep(50);
25. retry();
26. }
27. }
28. }

3. "永远不过期"：

    这里的“永远不过期”包含两层意思：

    (1) 从redis上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“物理”不过期。

    (2) 从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期

# http://dl2.iteye.com/upload/attachment/0114/5825/1647a0f0-0df5-3842-b15a-d60bed5379ae.png

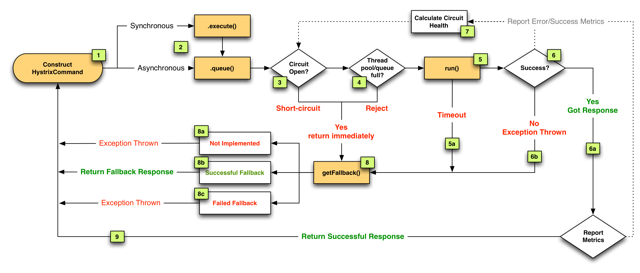
    从实战看，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受。

**Java代码  [收藏代码](javascript:void())**

1. String get(**final** String key) {
2. V v = redis.get(key);
3. String value = v.getValue();
4. **long** timeout = v.getTimeout();
5. **if** (v.timeout <= System.currentTimeMillis()) {
6. // 异步更新后台异常执行
7. threadPool.execute(**new** Runnable() {
8. **public** **void** run() {
9. String keyMutex = "mutex:" + key;
10. **if** (redis.setnx(keyMutex, "1")) {
11. // 3 min timeout to avoid mutex holder crash
12. redis.expire(keyMutex, 3 \* 60);
13. String dbValue = db.get(key);
14. redis.set(key, dbValue);
15. redis.delete(keyMutex);
16. }
17. }
18. });
19. }
20. **return** value;
21. }

4. 资源保护：

       之前在[缓存雪崩](https://www.iteye.com/blog/2249316)那篇文章提到了netflix的hystrix，可以做资源的隔离保护主线程池，如果把这个应用到缓存的构建也未尝不可。



三、四种方案对比：

      作为一个并发量较大的互联网应用，我们的目标有3个:

      1. 加快用户访问速度，提高用户体验。

      2. 降低后端负载，保证系统平稳。

      3. 保证数据“尽可能”及时更新(要不要完全一致，取决于业务，而不是技术。)

      所以第二节中提到的四种方法，可以做如下比较，还是那就话：没有最好，只有最合适。

|  |  |  |
| --- | --- | --- |
| 解决方案 | 优点 | 缺点 |
| 简单分布式锁(Tim yang) | 1. 思路简单  2. 保证一致性 | 1. 代码复杂度增大  2. 存在死锁的风险  3. 存在线程池阻塞的风险 |
| 加另外一个过期时间(Tim yang) | 1. 保证一致性 | 同上 |
| 不过期(本文) | 1. 异步构建缓存，不会阻塞线程池 | 1. 不保证一致性。  2. 代码复杂度增大(每个value都要维护一个timekey)。  3. 占用一定的内存空间(每个value都要维护一个timekey)。 |
| 资源隔离组件hystrix(本文) | 1. hystrix技术成熟，有效保证后端。  2. hystrix监控强大。 | 1. 部分访问存在降级策略。 |

四、总结

   1.  热点key + 过期时间 + 复杂的构建缓存过程 => mutex key问题

   2. 构建缓存一个线程做就可以了。

   3. 四种解决方案：没有最佳只有最合适。