## **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。

### **解决方案**

#### **1.使用互斥锁(mutex key)`**

业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

这个SETNX就是一个锁机制,当这个key不存在的时候,你setnx返回值就是1,如果不存在的时候就是0,就可以根据这个实现互斥锁.

SETNX，是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现锁的效果。在redis2.6.1之前版本未实现setnx的过期时间，所以这里给出两种版本代码参考：

*//2.6.1前单机版本锁*

String get(String key) {

String value = redis.get(key);

if (value == null) {

if (redis.setnx(key\_mutex, "1")) {

*// 3 min timeout to avoid mutex holder crash*

redis.expire(key\_mutex, 3 \* 60)

value = db.get(key);

redis.set(key, value);

redis.delete(key\_mutex);

} else {

*//其他线程休息50毫秒后重试*

Thread.sleep(50);

get(key);

}

}

}

最新版本代码：

public String get(key) {

String value = redis.get(key);

if (value == null) { *//代表缓存值过期*

*//设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db*

if (redis.setnx(key\_mutex, 1, 3 \* 60) == 1) { *//代表设置成功*

value = db.get(key);

redis.set(key, value, expire\_secs);

redis.del(key\_mutex);

} else { *//这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可*

sleep(50);

get(key); *//重试*

}

} else {

return value;

}

}

memcache代码：

if (memcache.get(key) == null) {

*// 3 min timeout to avoid mutex holder crash*

if (memcache.add(key\_mutex, 3 \* 60 \* 1000) == true) {

value = db.get(key);

memcache.set(key, value);

memcache.delete(key\_mutex);

} else {

sleep(50);

retry();

}

}

#### **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：

v = memcache.get(key);

if (v == null) {

if (memcache.add(key\_mutex, 3 \* 60 \* 1000) == true) {

value = db.get(key);

memcache.set(key, value);

memcache.delete(key\_mutex);

} else {

sleep(50);

retry();

}

} else {

if (v.timeout <= now()) {

if (memcache.add(key\_mutex, 3 \* 60 \* 1000) == true) {

*// extend the timeout for other threads*

v.timeout += 3 \* 60 \* 1000;

memcache.set(key, v, KEY\_TIMEOUT \* 2);

*// load the latest value from db*

v = db.get(key);

v.timeout = KEY\_TIMEOUT;

memcache.set(key, value, KEY\_TIMEOUT \* 2);

memcache.delete(key\_mutex);

} else {

sleep(50);

retry();

}

}

}

#### **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

        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。

String get(final String key) {

V v = redis.get(key);

String value = v.getValue();

long timeout = v.getTimeout();

if (v.timeout <= System.currentTimeMillis()) {

*// 异步更新后台异常执行*

threadPool.execute(new Runnable() {

public void run() {

String keyMutex = "mutex:" + key;

if (redis.setnx(keyMutex, "1")) {

*// 3 min timeout to avoid mutex holder crash*

redis.expire(keyMutex, 3 \* 60);

String dbValue = db.get(key);

redis.set(key, dbValue);

redis.delete(keyMutex);

}

}

});

}

return value;

}

#### **4. 资源保护：**

采用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。

四种解决方案：没有最佳只有最合适

|  |  |  |
| --- | --- | --- |
| 解决方案 | 优点 | 缺点 |
| 简单分布式互斥锁（mutex key） | 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 |
| “提前”使用互斥锁 | 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。 |