

23 | 缓存设计：缓存可以锦上添花也可以落井下石

2020-05-07 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽

时长 22:03 大小 20.19M



你好，我是朱晔。今天，我从设计的角度，与你聊聊缓存。

通常我们会使用更快的介质（比如内存）作为缓存，来解决较慢介质（比如磁盘）读取数据慢的问题，缓存是用空间换时间，来解决性能问题的一种架构设计模式。更重要的是，磁盘上存储的往往是原始数据，而缓存中保存的可以是面向呈现的数据。这样一来，缓存不仅仅是加快了 IO，还可以减少原始数据的计算工作。

此外，缓存系统一般设计简单，功能相对单一，所以诸如 Redis 这种缓存系统的整体量，能达到关系型数据库的几倍甚至几十倍，因此缓存特别适用于互联网应用的高并发场景。

使用 Redis 做缓存虽然简单好用，但使用和设计缓存并不是 set 一下这么简单，需要注意缓存的同步、雪崩、并发、穿透等问题。今天，我们就来详细聊聊。

不要把 Redis 当作数据库

通常，我们会使用 Redis 等分布式缓存数据库来缓存数据，但是**千万别把 Redis 当做数据库来使用**。我就见过许多案例，因为 Redis 中数据消失导致业务逻辑错误，并且因为没有保留原始数据，业务都无法恢复。

Redis 的确具有数据持久化功能，可以实现服务重启后数据不丢失。这一点，很容易让我们误认为 Redis 可以作为高性能的 KV 数据库。

其实，从本质上来看，Redis（免费版）是一个内存数据库，所有数据保存在内存中，并且直接从内存读写数据响应操作，只不过具有数据持久化能力。所以，Redis 的特点是，处理请求很快，但无法保存超过内存大小的数据。

备注：VM 模式虽然可以保存超过内存大小的数据，但是因为性能原因从 2.6 开始已经被废弃。此外，Redis 企业版提供了 Redis on Flash 可以实现 Key+ 字典 + 热数据保存在内存中，冷数据保存在 SSD 中。

因此，把 Redis 用作缓存，我们需要注意两点。

第一，从客户端的角度来说，缓存数据的特点一定是有原始数据来源，且允许丢失，即使设置的缓存时间是 1 分钟，在 30 秒时缓存数据因为某种原因消失了，我们也要能接受。当数据丢失后，我们需要从原始数据重新加载数据，不能认为缓存系统是绝对可靠的，更不能认为缓存系统不会删除没有过期的数据。

第二，从 Redis 服务端的角度来说，缓存系统可以保存的数据量一定是小于原始数据的。首先，我们应该限制 Redis 对内存的使用量，也就是设置 maxmemory 参数；其次，我们应该根据数据特点，明确 Redis 应该以怎样的算法来驱逐数据。

从 [Redis 的文档](#) 可以看到，常用的数据淘汰策略有：

allkeys-lru，针对所有 Key，优先删除最近最少使用的 Key；

volatile-lru，针对带有过期时间的 Key，优先删除最近最少使用的 Key；

volatile-ttl，针对带有过期时间的 Key，优先删除即将过期的 Key（根据 TTL 的值）；

allkeys-lfu（Redis 4.0 以上），针对所有 Key，优先删除最少使用的 Key；

volatile-lfu（Redis 4.0 以上），针对带有过期时间的 Key，优先删除最少使用的 Key。

其实，这些算法是 Key 范围 + Key 选择算法的搭配组合，其中范围有 allkeys 和 volatile 两种，算法有 LRU、TTL 和 LFU 三种。接下来，我就从 Key 范围和算法角度，和你说说如何选择合适的驱逐算法。

首先，从算法角度来说，Redis 4.0 以后推出的 LFU 比 LRU 更“实用”。试想一下，如果一个 Key 访问频率是 1 天一次，但正好在 1 秒前刚访问过，那么 LRU 可能不会选择优先淘汰这个 Key，反而可能会淘汰一个 5 秒访问一次但最近 2 秒没有访问过的 Key，而 LFU 算法不会有这个问题。而 TTL 会比较“头脑简单”一点，优先删除即将过期的 Key，但有可能这个 Key 正在被大量访问。

然后，从 Key 范围角度来说，allkeys 可以确保即使 Key 没有 TTL 也能回收，如果使用的时候客户端总是“忘记”设置缓存的过期时间，那么可以考虑使用这个系列的算法。而 volatile 会更稳妥一些，万一客户端把 Redis 当做了长效缓存使用，只是启动时候初始化一次缓存，那么一旦删除了此类没有 TTL 的数据，可能会导致客户端出错。

所以，不管是使用者还是管理者都要考虑 Redis 的使用方式，使用者需要考虑应该以缓存的姿势来使用 Redis，管理者应该为 Redis 设置内存限制和合适的驱逐策略，避免出现 OOM。

注意缓存雪崩问题

由于缓存系统的 IOPS 比数据库高很多，因此要特别小心短时间内大量缓存失效的情况。这种情况一旦发生，可能就会在瞬间有大量的数据需要回源到数据库查询，对数据库造成极大的压力，极限情况下甚至导致后端数据库直接崩溃。**这就是我们常说的缓存失效，也叫作缓存雪崩。**

从广义上说，产生缓存雪崩的原因有两种：

第一种是，缓存系统本身不可用，导致大量请求直接回源到数据库；

第二种是，应用设计层面大量的 Key 在同一时间过期，导致大量的数据回源。

第一种原因，主要涉及缓存系统本身高可用的配置，不属于缓存设计层面的问题，所以今天我主要和你说说如何确保大量 Key 不在同一时间被动过期。

程序初始化的时候放入 1000 条城市数据到 Redis 缓存中，过期时间是 30 秒；数据过期后从数据库获取数据然后写入缓存，每次从数据库获取数据后计数器 +1；在程序启动的同时，启动一个定时任务线程每隔一秒输出计数器的值，并把计数器归零。

压测一个随机查询某城市信息的接口，观察一下数据库的 QPS：


 复制代码

```
1 @Autowired
2 private StringRedisTemplate stringRedisTemplate;
3 private AtomicInteger atomicInteger = new AtomicInteger();
4
5 @PostConstruct
6 public void wrongInit() {
7     //初始化1000个城市数据到Redis, 所有缓存数据有效期30秒
8     IntStream.rangeClosed(1, 1000).forEach(i -> stringRedisTemplate.opsForValue().set("city:" + i, "city:" + i, 30, TimeUnit.SECONDS));
9     log.info("Cache init finished");
10
11     //每秒一次，输出数据库访问的QPS
12
13     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
14         log.info("DB QPS : {}", atomicInteger.getAndSet(0));
15     }, 0, 1, TimeUnit.SECONDS);
16 }
17
18 @GetMapping("city")
19 public String city() {
20     //随机查询一个城市
21     int id = ThreadLocalRandom.current().nextInt(1000) + 1;
22     String key = "city:" + id;
23     String data = stringRedisTemplate.opsForValue().get(key);
24     if (data == null) {
25         //回源到数据库查询
26         data = getCityFromDb(id);
27         if (!StringUtils.isEmpty(data))
28             //缓存30秒过期
29             stringRedisTemplate.opsForValue().set(key, data, 30, TimeUnit.SECONDS);
30     }
31     return data;
32 }
33
34 private String getCityFromDb(int cityId) {
```

```
35 //模拟查询数据库，查一次增加计数器加一
36 atomicInteger.incrementAndGet();
37 return "citydata" + System.currentTimeMillis();
38 }
```

使用 wrk 工具，设置 10 线程 10 连接压测 city 接口：

```
1 wrk -c10 -t10 -d 100s http://localhost:45678/cacheinvalid/city
```

 复制代码


启动程序 30 秒后缓存过期，回源的数据库 QPS 最高达到了 700 多：

```
[16:51:35.362] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :33 ] - DB QPS : 0
[16:51:36.363] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :33 ] - DB QPS : 789
[16:51:37.364] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :33 ] - DB QPS : 213
[16:51:38.364] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :33 ] - DB QPS : 0
```

解决缓存 Key 同时大规模失效需要回源，导致数据库压力激增问题的方式有两种。

方案一，差异化缓存过期时间，不要让大量的 Key 在同一时间过期。比如，在初始化缓存的时候，设置缓存的过期时间是 30 秒 + 10 秒以内的随机延迟（扰动值）。这样，这些 Key 不会集中在 30 秒这个时刻过期，而是会分散在 30~40 秒之间过期：

```
1 @PostConstruct
2 public void rightInit1() {
3     //这次缓存的过期时间是30秒+10秒内的随机延迟
4     IntStream.rangeClosed(1, 1000).forEach(i -> stringRedisTemplate.opsForValue
5         log.info("Cache init finished");
6         //同样1秒一次输出数据库QPS:
7         Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
8             log.info("DB QPS : {}", atomicInteger.getAndSet(0));
9             }, 0, 1, TimeUnit.SECONDS);
10 }
```

 复制代码

修改后，缓存过期时的回源不会集中在同一秒，数据库的 QPS 从 700 多降到了最高 100 左右：


```

[16:56:42.374] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 0
[16:56:43.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 83
[16:56:44.382] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 89
[16:56:45.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 114
[16:56:46.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 100
[16:56:47.374] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 115
[16:56:48.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 90
[16:56:49.376] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 98
[16:56:50.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 91
[16:56:51.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 106
[16:56:52.374] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 110
[16:56:53.375] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 11
[16:56:54.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 0
[16:56:55.373] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController :42 ] - DB QPS : 0

```

方案二，让缓存不主动过期。初始化缓存数据的时候设置缓存永不过期，然后启动一个后台线程 30 秒一次定时把所有数据更新到缓存，而且通过适当的休眠，控制从数据库更新数据的频率，降低数据库压力：

 复制代码

```

1  @PostConstruct
2  public void rightInit2() throws InterruptedException {
3      CountDownLatch countDownLatch = new CountDownLatch(1);
4      //每隔30秒全量更新一次缓存
5      Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
6          IntStream.rangeClosed(1, 1000).forEach(i -> {
7              String data = getCityFromDb(i);
8              //模拟更新缓存需要一定的时间
9              try {
10                 TimeUnit.MILLISECONDS.sleep(20);
11             } catch (InterruptedException e) { }
12             if (!StringUtils.isEmpty(data)) {
13                 //缓存永不过期，被动更新
14                 stringRedisTemplate.opsForValue().set("city" + i, data);
15             }
16         });
17         log.info("Cache update finished");
18         //启动程序的时候需要等待首次更新缓存完成
19         countDownLatch.countDown();
20     }, 0, 30, TimeUnit.SECONDS);
21
22     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
23         log.info("DB QPS : {}", atomicInteger.getAndSet(0));
24     }, 0, 1, TimeUnit.SECONDS);
25
26     countDownLatch.await();
27 }

```

这样修改后，虽然缓存整体更新的耗时在 21 秒左右，但数据库的压力会比较稳定：

```
[17:09:03.060] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 0
[17:09:04.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 43
[17:09:05.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 45
[17:09:06.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:07.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 42
[17:09:08.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 44
[17:09:09.059] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:10.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 47
[17:09:11.059] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 45
[17:09:12.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:13.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 44
[17:09:14.061] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 47
[17:09:15.059] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 42
[17:09:16.059] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 43
[17:09:17.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:18.060] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 44
[17:09:19.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:20.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 44
[17:09:21.061] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 44
[17:09:22.062] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:23.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 45
[17:09:24.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 45
[17:09:25.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 46
[17:09:25.355] [pool-2-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :59 ] - Cache update finished
[17:09:26.058] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 14
[17:09:27.059] [pool-3-thread-1] [INFO ] [o.g.t.c.c.c.CacheInvalidController] :64 ] - DB QPS : 0
```

关于这两种解决方案，**我们需要特别注意以下三点：**

方案一和方案二是截然不同的两种缓存方式，如果无法全量缓存所有数据，那么只能使用方案一；

即使使用了方案二，缓存永不过期，同样需要在查询的时候，确保有回源的逻辑。正如之前所说，我们无法确保缓存系统中的数据永不丢失。

不管是方案一还是方案二，在把数据从数据库加入缓存的时候，都需要判断来自数据库的数据是否合法，比如进行最基本的判空检查。

之前我就遇到过这样一个重大事故，某系统会在缓存中对基础数据进行长达半年的缓存，在某个时间点 DBA 把数据库中的原始数据进行了归档（可以认为是删除）操作。因为缓存中的数据一直在所以一开始没什么问题，但半年后的一天缓存中数据过期了，就从数据库中查询到了空数据加入缓存，爆发了大面积的事故。

这个案例说明，缓存会让我们更不容易发现原始数据的问题，所以在把数据加入缓存之前一定要校验数据，如果有明显异常要及时报警。

说到这里，我们再仔细看一下回源 QPS 超过 700 的截图，可以看到在并发情况下，总共 1000 条数据回源达到了 1002 次，说明有一些条目出现了并发回源。这，就是我后面要讲到的缓存并发问题。

注意缓存击穿问题

在某些 Key 属于极端热点数据，且并发量很大的情况下，如果这个 Key 过期，可能会在某个瞬间出现大量的并发请求同时回源，相当于大量的并发请求直接打到了数据库。**这种情况，就是我们常说的缓存击穿或缓存并发问题。**

我们来重现下这个问题。在程序启动的时候，初始化一个热点数据到 Redis 中，过期时间设置为 5 秒，每隔 1 秒输出一下回源的 QPS：


 复制代码

```
1  @PostConstruct
2  public void init() {
3      //初始化一个热点数据到Redis中，过期时间设置为5秒
4      stringRedisTemplate.opsForValue().set("hotsopt", getExpensiveData(), 5, TimeUnit.SECONDS);
5      //每隔1秒输出一下回源的QPS
6
7      Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
8          log.info("DB QPS : {}", atomicInteger.getAndSet(0));
9      }, 0, 1, TimeUnit.SECONDS);
10 }
11
12 @GetMapping("wrong")
13 public String wrong() {
14     String data = stringRedisTemplate.opsForValue().get("hotsopt");
15     if (StringUtils.isEmpty(data)) {
16         data = getExpensiveData();
17         //重新加入缓存，过期时间还是5秒
18         stringRedisTemplate.opsForValue().set("hotsopt", data, 5, TimeUnit.SECONDS);
19     }
20     return data;
21 }
```

可以看到，每隔 5 秒数据库都有 20 左右的 QPS：

```
[17:41:36.096] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:37.097] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 19
[17:41:38.096] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:39.096] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:40.098] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:41.097] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:42.098] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 19
[17:41:43.097] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:44.100] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:45.101] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:46.101] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:47.100] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 20
[17:41:48.100] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
[17:41:49.100] [pool-2-thread-1] [INFO] [g.t.c.c.c.CacheConcurrentController:33] - DB QPS : 0
```


如果回源操作特别昂贵，那么这种并发就不能忽略不计。这时，我们可以考虑使用锁机制来限制回源的并发。比如如下代码示例，使用 Redisson 来获取一个基于 Redis 的分布式锁，在查询数据库之前先尝试获取锁：

 复制代码

```
1 @Autowired
2 private RedissonClient redissonClient;
3 @GetMapping("right")
4 public String right() {
5     String data = stringRedisTemplate.opsForValue().get("hotsopt");
6     if (StringUtils.isEmpty(data)) {
7         RLock locker = redissonClient.getLock("locker");
8         //获取分布式锁
9         if (locker.tryLock()) {
10             try {
11                 data = stringRedisTemplate.opsForValue().get("hotsopt");
12                 //双重检查，因为可能已经有一个B线程过了第一次判断，在等锁，然后A线程已经把
13                 if (StringUtils.isEmpty(data)) {
14                     //回源到数据库查询
15                     data = getExpensiveData();
16                     stringRedisTemplate.opsForValue().set("hotsopt", data, 5, TimeUnit.SECONDS);
17                 }
18             } finally {
19                 //别忘记释放，另外注意写法，获取锁后整段代码try+finally，确保unlock万无
20                 locker.unlock();
21             }
22         }
23     }
24     return data;
25 }
```

这样，可以把回源到数据库的并发限制在 1：

```
[18:47:50.298] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:51.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:52.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 1
[18:47:53.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:54.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:55.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:56.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:57.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 1
[18:47:58.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:47:59.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:48:00.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:48:01.301] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:48:02.297] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 1
[18:48:03.298] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
[18:48:04.299] [pool-2-thread-1] [INFO ] [.g.t.c.c.c.CacheConcurrentController:32 ] - DB QPS : 0
```

在真实的业务场景下，**不一定要**这么严格地使用双重检查分布式锁进行全局的并发限制，因为这样虽然可以把数据库回源并发降到最低，但也限制了缓存失效时的并发。可以考虑的方式是：


方案一，使用进程内的锁进行限制，这样每一个节点都可以以一个并发回源数据库；

方案二，不使用锁进行限制，而是使用类似 Semaphore 的工具限制并发数，比如限制为 10，这样既限制了回源并发数不至于太大，又能使得一定量的线程可以同时回源。

注意缓存穿透问题

在之前的例子中，缓存回源的逻辑都是当缓存中查不到需要的数据时，回源到数据库查询。这里容易出现的一个漏洞是，缓存中没有数据不一定代表数据没有缓存，还有一种可能是原始数据压根就不存在。

比如下面的例子。数据库中只保存有 ID 介于 0（不含）和 10000（包含）之间的用户，如果从数据库查询 ID 不在这个区间的用户，会得到空字符串，所以缓存中缓存的也是空字符串。如果使用 ID=0 去压接口的话，从缓存中查出了空字符串，认为是缓存中没有数据回源查询，其实相当于每次都回源：

 复制代码

```
1 @GetMapping("wrong")
2 public String wrong(@RequestParam("id") int id) {
3     String key = "user" + id;
4     String data = stringRedisTemplate.opsForValue().get(key);
5     //无法区分是无效用户还是缓存失效
6     if (StringUtils.isEmpty(data)) {
7         data = getCityFromDb(id);
8         stringRedisTemplate.opsForValue().set(key, data, 30, TimeUnit.SECONDS)
9     }
10    return data;
11 }
12
13 private String getCityFromDb(int id) {
14     AtomicInteger.incrementAndGet();
15     //注意，只有ID介于0（不含）和10000（包含）之间的用户才是有效用户，可以查询到用户信息
16     if (id > 0 && id <= 10000) return "userdata";
17     //否则返回空字符串
18     return "";
19 }
```

压测后数据库的 QPS 达到了几千：

```
[18:01:56.497] [pool-2-thread-1] [INFO] [g.t.c.c.c.CachePenetrationController:33] - DB QPS : 4106
[18:01:57.497] [pool-2-thread-1] [INFO] [g.t.c.c.c.CachePenetrationController:33] - DB QPS : 5268
[18:01:58.497] [pool-2-thread-1] [INFO] [g.t.c.c.c.CachePenetrationController:33] - DB QPS : 4908
[18:01:59.497] [pool-2-thread-1] [INFO] [g.t.c.c.c.CachePenetrationController:33] - DB QPS : 6198
```

如果这种漏洞被恶意利用的话，就会对数据库造成很大的性能压力。**这就是缓存穿透。**


这里需要注意，缓存穿透和缓存击穿的区别：

缓存穿透是指，缓存没有起到压力缓冲的作用；

而缓存击穿是指，缓存失效时瞬时的并发打到数据库。

解决缓存穿透有以下两种方案。

方案一，对于不存在的数据，同样设置一个特殊的 Value 到缓存中，比如当数据库中查出的用户信息为空的时候，设置 NODATA 这样具有特殊含义的字符串到缓存中。这样下次请求缓存的时候还是可以命中缓存，即直接从缓存返回结果，不查询数据库：

 复制代码

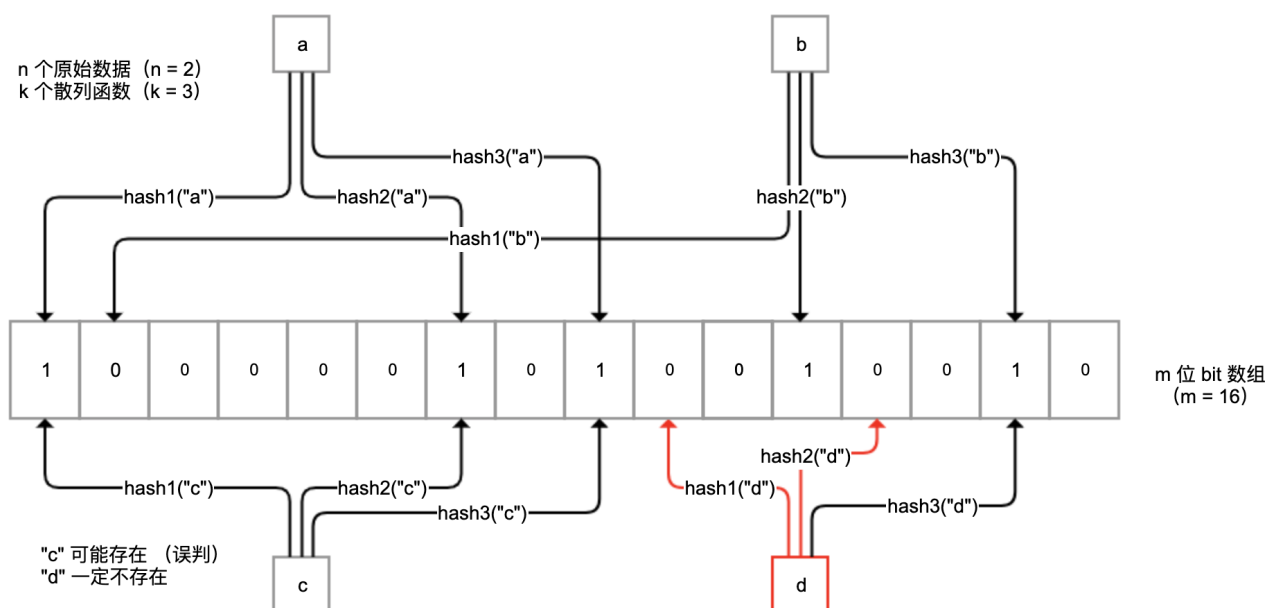
```
1 @GetMapping("right")
2 public String right(@RequestParam("id") int id) {
3     String key = "user" + id;
4     String data = stringRedisTemplate.opsForValue().get(key);
5     if (StringUtils.isEmpty(data)) {
6         data = getCityFromDb(id);
7         //校验从数据库返回的数据是否有效
8         if (!StringUtils.isEmpty(data)) {
9             stringRedisTemplate.opsForValue().set(key, data, 30, TimeUnit.SECONDS);
10        }
11        else {
12            //如果无效，直接在缓存中设置一个NODATA，这样下次查询时即使是无效用户还是可以命中缓存
13            stringRedisTemplate.opsForValue().set(key, "NODATA", 30, TimeUnit.SECONDS);
14        }
15    }
16    return data;
17 }
```

但，这种方式可能会把大量无效的数据加入缓存中，如果担心大量无效数据占满缓存的话还可以考虑方案二，即使用布隆过滤器做前置过滤。

布隆过滤器是一种概率型数据库结构，由一个很长的二进制向量和一系列随机映射函数组成。它的原理是，当一个元素被加入集合时，通过 k 个散列函数将这个元素映射成一个 m 位 bit 数组中的 k 个点，并置为 1。

检索时，我们只要看看这些点是不是都是 1 就（大概）知道集合中有没有它了。如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。

原理如下图所示：



布隆过滤器不保存原始值，空间效率很高，平均每一个元素占用 2.4 字节就可以达到万分之一的误判率。这里的误判率是指，过滤器判断值存在而实际并不存在的概率。我们可以设置布隆过滤器使用更大的存储空间，来得到更小的误判率。


你可以把所有可能的值保存在布隆过滤器中，从缓存读取数据前先过滤一次：

如果布隆过滤器认为值不存在，那么值一定是不存在的，无需查询缓存也无需查询数据库；

对于极小概率的误判请求，才会最终让非法 Key 的请求走到缓存或数据库。

要用上布隆过滤器，我们可以使用 Google 的 Guava 工具包提供的 BloomFilter 类改造一下程序：启动时，初始化一个具有所有有效用户 ID 的、10000 个元素的 BloomFilter，在

从缓存查询数据之前调用其 `mightContain` 方法，来检测用户 ID 是否可能存在；如果布隆过滤器说值不存在，那么一定是不存在的，直接返回：

 复制代码

```
1 private BloomFilter<Integer> bloomFilter;
2
3 @PostConstruct
4 public void init() {
5     //创建布隆过滤器，元素数量10000，期望误判率1%
6     bloomFilter = BloomFilter.create(Funnels.integerFunnel(), 10000, 0.01);
7     //填充布隆过滤器
8     IntStream.rangeClosed(1, 10000).forEach(bloomFilter::put);
9 }
10
11 @GetMapping("right2")
12 public String right2(@RequestParam("id") int id) {
13     String data = "";
14     //通过布隆过滤器先判断
15     if (bloomFilter.mightContain(id)) {
16         String key = "user" + id;
17         //走缓存查询
18         data = stringRedisTemplate.opsForValue().get(key);
19         if (StringUtils.isEmpty(data)) {
20             //走数据库查询
21             data = getCityFromDb(id);
22             stringRedisTemplate.opsForValue().set(key, data, 30, TimeUnit.SECONDS);
23         }
24     }
25     return data;
26 }
```

对于方案二，我们需要同步所有可能存在的值并加入布隆过滤器，这是比较麻烦的地方。如果业务规则明确的话，你也可以考虑直接根据业务规则判断值是否存在。

其实，方案二可以和方案一同时使用，即将布隆过滤器前置，对于误判的情况再保存特殊值到缓存，双重保险避免无效数据查询请求打到数据库。

注意缓存数据同步策略

前面提到的 3 个案例，其实都属于缓存数据过期后的被动删除。在实际情况下，修改了原始数据后，考虑到缓存数据更新的及时性，我们可能会采用主动更新缓存的策略。这些策略可能是：

先更新缓存，再更新数据库；

先更新数据库，再更新缓存；

先删除缓存，再更新数据库，访问的时候按需加载数据到缓存；

先更新数据库，再删除缓存，访问的时候按需加载数据到缓存。

那么，我们应该选择哪种更新策略呢？我来和你逐一分析下这 4 种策略：

“先更新缓存再更新数据库”策略不可行。数据库设计复杂，压力集中，数据库因为超时等原因更新操作失败的可能性较大，此外还会涉及事务，很可能因为数据库更新失败，导致缓存和数据库的数据不一致。

“先更新数据库再更新缓存”策略不可行。一是，如果线程 A 和 B 先后完成数据库更新，但更新缓存时却是 B 和 A 的顺序，那很可能把旧数据更新到缓存中引起数据不一致；二是，我们不确定缓存中的数据是否会被访问，不一定要把所有数据都更新到缓存中去。

“先删除缓存再更新数据库，访问的时候按需加载数据到缓存”策略也不可行。在并发的情况下，很可能删除缓存后还没来得及更新数据库，就有另一个线程先读取了旧值到缓存中，如果并发量很大的话这个概率也会很大。

“先更新数据库再删除缓存，访问的时候按需加载数据到缓存”策略是最好的。虽然在极端情况下，这种策略也可能出现数据不一致的问题，但概率非常低，基本可以忽略。举一个“极端情况”的例子，比如更新数据的时间节点恰好是缓存失效的瞬间，这时 A 先读取到了旧值，随后在 B 操作数据库完成更新并且删除了缓存之后，A 再把旧值加入缓存。

需要注意的是，更新数据库后删除缓存的操作可能失败，如果失败则考虑把任务加入延迟队列进行延迟重试，确保数据可以删除，缓存可以及时更新。因为删除操作是幂等的，所以即使重复删问题也不是太大，这又是删除比更新好的一个原因。

因此，针对缓存更新更推荐的方式是，缓存中的数据不由数据更新操作主动触发，统一在需要使用的时候按需加载，数据更新后及时删除缓存中的数据即可。

重点回顾

今天，我主要是从设计的角度，和你分享了数据缓存的三大问题。

第一，我们不能把诸如 Redis 的缓存数据库完全当作数据库来使用。我们不能假设缓存始终可靠，也不能假设没有过期的数据必然可以被读取到，需要处理好缓存的回源逻辑；而且要显式设置 Redis 的最大内存使用和数据淘汰策略，避免出现 OOM 的问题。

第二，缓存的性能比数据库好很多，我们需要考虑大量请求绕过缓存直击数据库造成数据库瘫痪的各种情况。对于缓存瞬时大面积失效的缓存雪崩问题，可以通过差异化缓存过期时间解决；对于高并发的缓存 Key 回源问题，可以使用锁来限制回源并发数；对于不存在的数据穿透缓存的问题，可以通过布隆过滤器进行数据存在性的预判，或在缓存中也设置一个值来解决。

第三，当数据库中的数据有更新的时候，需要考虑如何确保缓存中数据的一致性。我们看到，“先更新数据库再删除缓存，访问的时候按需加载数据到缓存”的策略是最为妥当的，并且要尽量设置合适的缓存过期时间，这样即便真的发生不一致，也可以在缓存过期后数据得到及时同步。

最后，我要提醒你的是，在使用缓存系统的时候，要监控缓存系统的内存使用量、命中率、对象平均过期时间等重要指标，以便评估系统的有效性，并及时发现问题。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

思考与讨论

1. 在聊到缓存并发问题时，我们说到热点 Key 回源会对数据库产生的压力问题，如果 Key 特别热的话，可能缓存系统也无法承受，毕竟所有的访问都集中打到了一台缓存服务器。如果我们使用 Redis 来做缓存，那可以把一个热点 Key 的缓存查询压力，分散到多个 Redis 节点上吗？
2. 大 Key 也是数据缓存容易出现的一个问题。如果一个 Key 的 Value 特别大，那么可能会对 Redis 产生巨大的性能影响，因为 Redis 是单线程模型，对大 Key 进行查询或删除等操作，可能会引起 Redis 阻塞甚至是高可用切换。你知道怎么查询 Redis 中的大 Key，以及如何在设计上实现大 Key 的拆分吗？

关于缓存设计，你还遇到过哪些坑呢？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

5月-6月课表抢先看

充 ¥500 得 ¥580

赠 「¥ 99 运动水杯+ ¥129 防紫外线伞」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 22 | 接口设计: 系统间对话的语言, 一定要统一

下一篇 24 | 业务代码写完, 就意味着生产就绪了?

精选留言 (10)

写留言



Darren

2020-05-07

第一个问题:

vivi童鞋回复的很棒, 我的第一想法也是加随机后缀。

分型一个场景: 假如在一个非常热点的数据, 数据更新不是很频繁, 但是查询非常的频繁, 要保证基本保证100%的缓存命中率, 该怎么处理?

我们的做法是, 空间换效率, 同一个key保留2份, 1个不带后缀, 1个带后缀, ...

展开 ∨

作者回复: 厉害



7

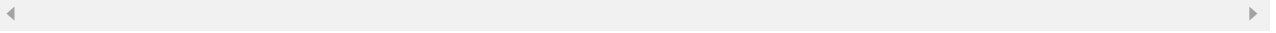


vivi

2020-05-07

第一个问题，我认为可以给hotkey加上后缀，让这些hotkey打散到不同的redis实例上。

作者回复: 是的，加随机前缀后缀是一个办法



2



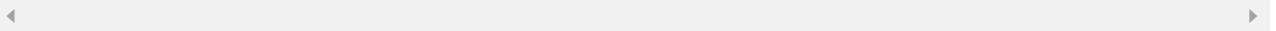
赵宇浩

2020-05-07

跟用户信息相关的缓存怎么处理穿透，因为用户的量很大，很难全放进布隆过滤器。

作者回复: 测试一下 <https://krisives.github.io/bloom-calculator/>

10亿的数据量，期望千分之一的错误率，推荐10个Hash函数，占用内存空间不到1.8GB



2



汝林外史

2020-05-08

先更新数据库再删缓存，如果并发查询发生在删缓存之前更新数据库之后，查到的不都是旧数据吗？

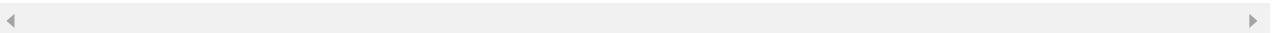
不是应该先删除缓存，向队列中插入一个数据的修改标识，并发查询发现缓存为空把查询数据库的标识也放入队列中，等修改的处理完了再处理查询的请求。

展开 ∨

作者回复: 我们的目标是避免长期出现不一致（读取到了旧值进入缓存属于长期不一致，因为又需要等一个缓存周期了）。先更新数据库再删缓存，如果并发查询发生在删缓存之前更新数据库之后，查到的不都是旧数据吗？是旧数据但是这是非常短暂的，下次查询就是新数据了。

你说的如何实现绝度一致。先删除缓存，向队列中插入一个数据的修改标识，我们如何确保删除缓存后向队列插入数据修改标识之前又有请求过来读取数据了呢？绝对一致或许考虑锁的方案。

不过反过来思考，既然已经缓存了，真的需要绝对一致吗？



1



hellojd

2020-05-11

文稿中提到的布隆过滤器，怎么保证加载用户全部数据。用户量太大会不会oom?,如果新

用户注册过来，怎么同步更新所有实例的布隆过滤器数据

展开 ▾

作者回复: 布隆过滤器容量预估可以参考我之前的回答，如果数据保存节点内可以定时任务来加载新增数据，不同节点数据略有不同问题不大，或者也可以为redis安装布隆模块，集中保存



hellojd

2020-05-11

Mongodb算缓存吗？还是数据库？我们现正在开发的一个功能，将用户会员状态放到缓存中，缓存的过期时间是会员的过期时间，如果缓存没到缓存过期时间就被驱除，那就死翘翘了。

作者回复: mongodb是数据库，会员状态放缓存显然不合适



wang

2020-05-09

对于一个频繁更新的热点key，有什么好的方案，先更新redis在定时同步到数据库，可能会丢数据

作者回复: 可以考虑一下Event Sourcing的思想，所有数据都从缓存读取，修改直接记录Append Log+刷缓存，异步Apply到数据库，如果丢数据重放日志即可



那一刻

2020-05-08

关于redis热key的处理，vivi和Darren给出很好的方案。如果hot key数据量不大的话，服务器本地localcache也是个方法吧？另外想问老师，在工程上如何发现热key呢？使用redis4提供的object freq？不知是否有其他方式？

展开 ▾

作者回复: <https://www.infoq.cn/article/3L3zAQ4H8xpNoM2glSyi>





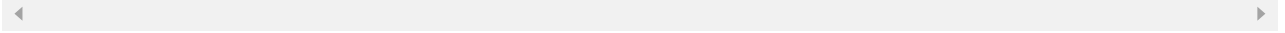
Geek_2b3614

2020-05-08

山穷水尽疑无路，柳暗花明又一村。啊。

展开 ∨

作者回复: :)



花轮君

2020-05-08

热key为什么不落到localcache呢

展开 ∨

