

22 | 赫赫有名的双刃剑：缓存（下）

2019-10-30 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 22:18 大小 15.32M



你好，我是四火。

在上一讲中，我们介绍了缓存的本质和应用模式。今天我们继续讨论缓存，这一讲会结合一些实际项目，谈一谈缓存的使用会有哪些问题，以及缓存框架的一些通用性的东西。

缓存使用的问题

既然说缓存是“双刃剑”，那我们就必须要谈论它的另一刃——缓存使用可能带来的问题。

1. 缓存穿透

缓存穿透，指的是在某些情况下，大量对于同一个数据的访问，经过了缓存屏障，但是缓存却未能起到应有的保护作用。举例来说，对某一个 key 的查询，如果数据库里没有这个数据，那么缓存中也没有数据的存放，每次请求到来都会去查询数据库，缓存根本起不到应有的作用。

当然，这个问题也不难解决，比方说我们可以在缓存中对这个 key 存放一个空结果，毕竟“没有结果”也是结果，也是需要缓存起来的。还有一种缓解方法是使用 [布隆过滤器](#) 等数据结构，在数据库查询之前，预先过滤掉某些不存在的结果。

还有一种特殊情况也会造成缓存穿透的严重后果。一般的缓存策略下，往往需要先发生一次缓存命中失败，接着从实际存储（比如数据库）中得到结果，再回填到内存缓存中。但是，如果这个数据库查询过程比较慢，大量同一数据的请求像雨点一样几乎同时到来，就会全部穿透缓存，一并落到了数据库上，而那个时候最早的那个请求引发的缓存回填甚至都还没有发生，在这种情况下数据库直接就挂掉了，虽然缓存的机制本身看起来并没有任何问题。

这种问题在某些时间窗口敏感的高并发系统中可能出现，解决方法有这样两种。

一种是以流量控制的方式，限制对于同一数据的访问，必须等到前一个完成以后，下一个才能进行，即如果缓存失效而引发的数据库查询正在进行，其它请求就得老老实实地等着。这种方法通用性好，但这个等待机制可能较为复杂，且有可能影响用户体验。

另一种方法是缓存预热，在大批量请求到来以前，先主动将该缓存填充好。这种方法操作简单高效，但局限性是需要提前知道哪些数据可能引发缓存穿透的问题。

2. 缓存雪崩

原本起屏障作用的缓存，如果在一定的时间段内，对于大量的请求访问失效，即失去了屏障作用，造成它后方的系统压力过大，引起系统过载、宕机等问题，就叫做缓存雪崩。

我以前在 Amazon 工作的时候，有个著名的内部分享，介绍了 Amazon 曾经发生的“六大灾难”，其中一大就是缓存雪崩。这个问题发生的时间已经是好多年前了，具体是这样的：有一次 Amazon 机房突然断电，在恢复的时候把网页服务器都通上了电，这时候缓存服务几乎还没有缓存数据，缓存命中率几乎为零，于是大量的请求冲向数据库，直接把数据库冲垮了。外在的表现就是，断电导致网站无法提供服务，短期内访问恢复，随后又丧失服务能力。

事实上，我们也总能看到很多技术报告里面写：平均的缓存命中率能够达到百分之九十多，可以飙到多少多少的 TPS，为此可以节约多少多少硬件成本。初看这样的设计真不错，但是很容易忽视的一点是：这样的数据是建立在足够长的时间以及足够多的统计数据的基础之上的，但是在单个时间段内，由于缓存雪崩效应，缓存命中率可以低到难以承受的地步，导致底层的数据服务直接被冲垮。

对于**这种类型的雪崩，最常见的解决方法无非还是限流、预热两种**：前者保证了请求大量落到数据库的时候，系统只接纳能够承载的数量；而后者则在请求访问前，先主动地往内存中加载一定的热点数据，这样请求到来的时候，缓存不是空的，已经具有一定的保护能力了。

好，我们再回到 Amazon 那个问题，当时的解决方法就是我们刚刚讲的第一种——限流。当时整个系统对于单台机器的限流已经做得比较好了，后来工程师一台一台逐步启动，每启动一台机器，就等一会，等到缓存数据填充并稳定以后，再启动下一台，这样最多也就是单台机器的所有请求全部发生了穿透，这个数量就小得多了，数据库也是可以正常负载的。

另外一个常见的缓存雪崩场景是：缓存数据通常都有过期时间的，如果缓存加载的时间比较集中，那么很可能到了某一时间点，大量的缓存就会同时过期，于是对应这些数据的请求全部落到了后面的数据库上，从而造成系统崩溃。这个问题解决起来也不难，那就是避免缓存集中写入的时间，如果无法避免，就使用一个范围随机数来均匀地分散过期时间，从而打散缓存过期对系统造成的压力。

3. 缓存容量失控

刚工作不久的时候，我参与做过这样一个系统，用户的行为需要被记录到数据库里，但是每条记录发生的时候都写一次数据库的话开销就太大了，于是有同事设计了一个链表：

用户的行为首先会被即时记录到内存链表里面去；

每 10 分钟从链表往数据库里面集中写一次数据，然后清空链表内的数据。

看起来这就像是我们在 [🔗\[第 21 讲\]](#) 中讲到的 Write-Back 模式，看起来也确实可以实现需求。可是，上线没多久系统就挂掉了。那么，这样的设计有什么问题呢？

清空链表数据是使用时间条件触发的任务来完成，**通过时间因素来限制空间大小，远不如通过队列长度来限制空间大小来得可靠**。换句话说，如果这 10 分钟内事件暴增，链表

就很容易变得非常大。**这个变化范围取决于请求的上限，而不是在缓存系统自己的掌控中。**

清空链表的任务，如果在执行的过程中出现了异常，甚至仅仅是处理速度受到阻塞，那就会直接导致链表数据无法得到清空，甚至越积越多。实际上，**链表清空数据并写入数据库是一个耗时的异步行为，这是另一个受控性较差的点。**我们在使用异步系统批量写入数据的时候，一定要考虑这个潜在的危险。

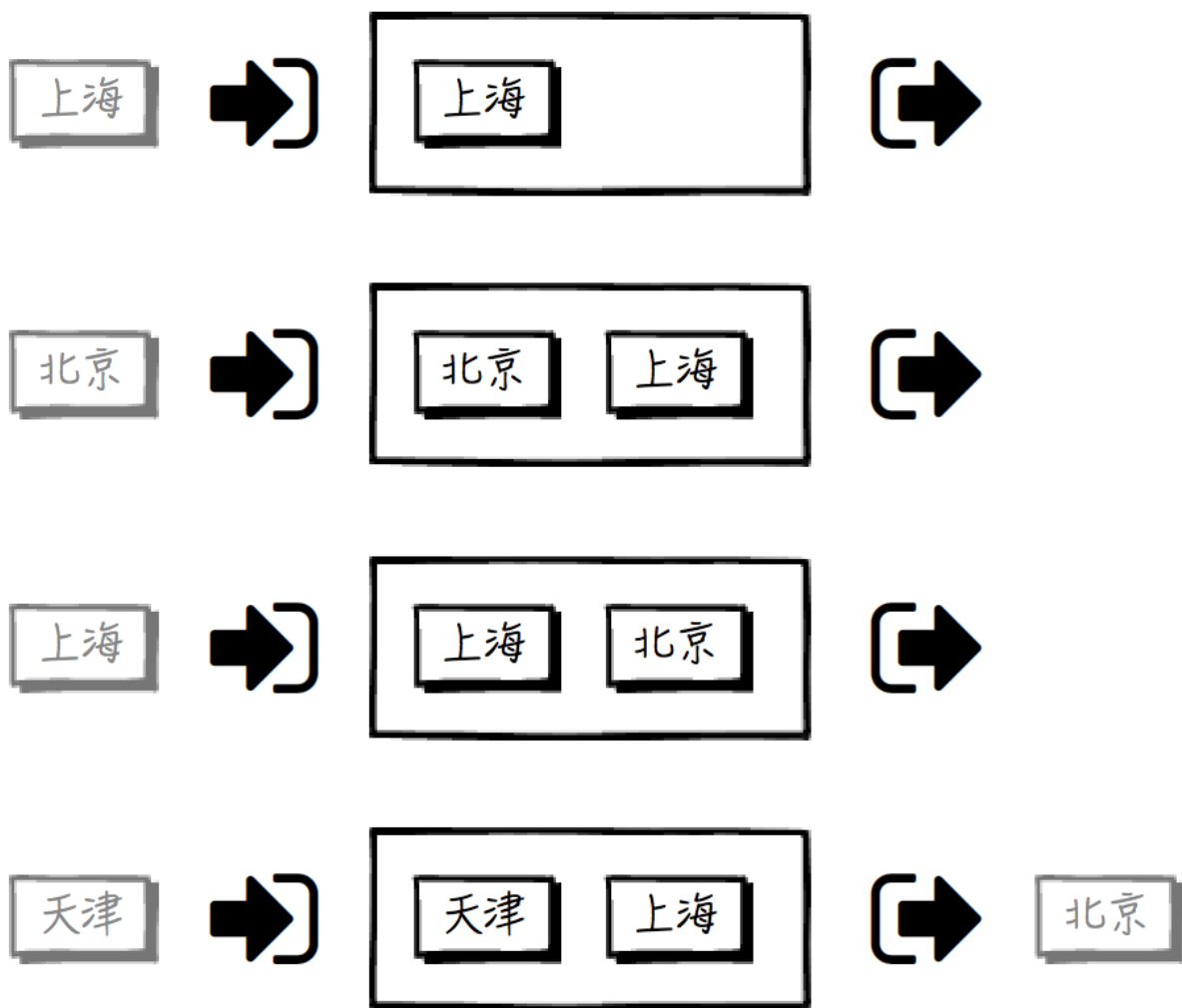
这些问题当然在明确的情况下可以得到规避，但是毫无疑问，这样的设计充满了潜在的危险。事实上，最终这样的问题也确实发生了，二者相加导致的结果是链表巨大，撑死了整个系统，OOM，系统失去响应。

因此，我们对于缓存容量的控制，最好是基于缓存容量本身来直接控制，但是考虑到某些编程语言的自身限制，比如 Java，从内存消耗的角度来实现不方便，那么就可以通过基于队列的长度来替代实现。

4. LRU 的致命缺陷

LRU 指的是 Least Recently Used，最少最近使用算法。这是缓存队列维护的最常见算法，原理是：维护一个限定最大容量的队列，队列头部总是放置最近访问的元素（包括新加入的元素），而在超过容量限制时总是从队尾淘汰元素。

我们可以用这样一张图，来解释 LRU 的工作原理：



假设用这个缓存的 LRU 队列来存储城市信息，且队列容量只有 2。

第一步，用户访问上海信息，上海节点被加入队列；

第二步，用户访问北京信息，北京节点从队列头部加入，上海相应地被往尾部推；

第三步，用户又访问上海信息，上海被挪到头部；

第四步，用户访问天津信息，从头部加入队列后，队列长度超出容量 2，因此从尾部将北京挤出缓存队列。

这看起来是个很完美的缓存淘汰算法，在队列较长时，总是能保证最近访问的数据位于队列的头部，而在需要从缓存中淘汰数据时，总是能从尾部淘汰最不常用的那一个。但是，如果用户有意无意地访问一些错误信息，就会破坏掉这个 LRU 队列中最近访问数据的真实性。

我曾经在实际项目中遇到过这样一个问题，由于搜索引擎的多个并行爬虫在短时间内访问网站并抓取一些冷门页面，这时候这个 LRU 队列中就存储了相关的冷门数据信息。接着网站活动开启的时间到了，用户量很快就上来了，这时候大量的数据访问全部穿透缓存，导致数据库压力剧增，网站响应时间一下就飙升到了告警线之上。

既然这个问题已经很明确了，那么解决就不是难事了。有多种算法可以作为 LRU 的改进方案，比如 LRU-K。就是主缓存队列排的是“第 K 次访问的元素”，也就是说，如果访问次数小于 K，则在另外的一个“低级”队列中维护，这样就保证了只有到达一定的访问下限才会被送到主 LRU 队列中。

这种方法保证了偶然的页面访问不会影响网站在 LRU 队列中应有的数据分布。进一步优化，可以将两级队列变成更多级，或者是将低级队列的策略变成 FIFO（2Q 算法）等等，但原理是不变的。

缓存框架

鉴于缓存的普遍性，缓存框架也可以说是百花齐放。如果你在大型 Web 项目中工作过，你很可能已经用过某一个缓存框架了。下面我就针对缓存框架的两个方面进行讲解，一方面是集成方式，另一方面是核心要素。希望这部分内容，可以帮助你在考察新的缓存框架的时候，心里能有个大致可以参照的谱。

集成方式

在上一讲我介绍了 Web 应用 MVC 的三层都可以集成缓存能力，下面我们来进一步思考这部分内容。缓存功能具体怎样整合集成到 Web 应用中，每一种方式都意味着一个切入点。我认为归纳一下，通常包含了下面这样几种方式。

方式 1：编程方式

这种是最常见的方式，使用编程的方式来获取缓存数据。这种方式比较灵活，对于代码往往以 Cache-Aside 模式应用。我们以 Java 世界应用最广泛的缓存框架 [Ehcache](#) 为例，示例代码片段如下：

 复制代码

```
1 Cache<String, City> cityCache = cacheManager.createCache("cityCache", CacheCon  
2 cityCache.put("Beijing", beijingInfo);
```



```
3 City beijing = cityCache.get("Beijing");
```

这里建立了一个城市的缓存，key 为城市名称，value 为城市对象，存取操作和对普通 Map 的操作相比，没有区别。

方式 2：方法注解

这种方式的好处在于，可以对方法的调用保持透明，不需要使用单独的缓存代码去分散对业务逻辑的专注。且看下面的例子：

```
1 @Cacheable(value="getCity", key="#name")
2 public City getCity(String name) { ... }
```


 复制代码

这种方式下，同名、同参数方法的再次调用，就可以命中缓存而直接返回。

方式 3：配置文件的注入

这种也比较常见，比如 MyBatis 在 mapper 标签中可以指定 cache 标签，通过这种方式就可以把选定的缓存框架注入到这个持久层框架中。对于指定映射的数据，再次访问时会优先从缓存中查找，这种应用方式就是前一讲我们提到的缓存应用模式中的 Read/Write-Through 模式。

```
1 <mapper namespace="..." >
2   <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
3   ...
4 </mapper>
```

 复制代码

方式 4：Web 容器的 Filter

在 Ehcache 2 中，可以配置 net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter 这样一个 filter 到

Tomcat 的 web.xml 中，再配合 filter 的映射匹配参数和初始化参数，就可以实现整个请求的过滤功能。

在 Ehcache 3 中，这个类被取消了，因为它的业务性过于具体，不符合 Ehcache 的设计原则。但是，你依然可以在 filter 里面，以前面提到的编程方式很容易地实现对于完整请求的缓存。如果你对这里提到的 filter 感到陌生，可以回看 [\[第 12 讲\]](#) 中的“Tomcat 中配置过滤器”这部分内容。

方式 5：页面模板中的 Cache 标签

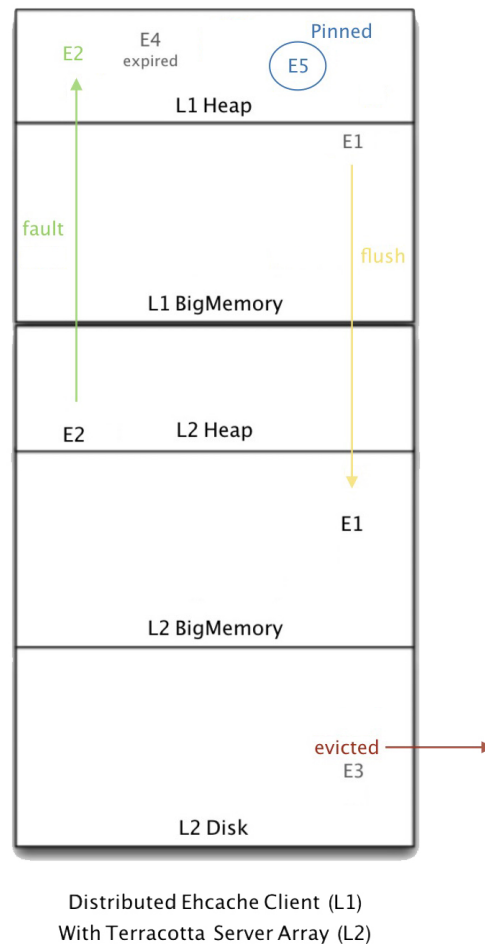
这种方式相对比较少见，有一些页面模板支持 Cache 标签或表达式语法（例如 Django 中，它被称为 Template Fragment Caching），在标签属性或语法参数中可以指定缓存的时间和条件，标签内部的 HTML 将被缓存起来，以避免在每次模板渲染时都去执行其中的逻辑。

核心要素

一个缓存框架，拥有的特性和要素可以说五花八门，可是，有一些是真正的“核心”，在缺少了以后，就很难再称之为一个“缓存框架”了。那么，有哪些要素可以称之为缓存框架的核心呢？我认为，它至少包括这样几点。

要素 1：缓存数据的生命周期管理

缓存框架不只提供了一个简单的容器，还提供了使容器中的数据进行变动的能力，比如数据可以创建、更新、移动以及淘汰。且看 [Ehcache 官网](#)上的这张示意图：



整个容器是分层的，从上到下分别为 L1 Heap、L1 BigMemory、L2 Heap、L2 BigMemory 和 L2 Disk，级别依次降低。这里面定义了几种不同的行为，来反映数据的流动：

Flush，右侧黄色的箭头，数据从高层向低层移动；

Fault，左侧绿色箭头，数据从低层拷贝到高层，但不删除；

Eviction，下方红色箭头，数据永久淘汰出缓存数据容器；

Expiration，上方烟灰色图案，数据过期了，意味着可以被 flushed 或者 evicted，但是考虑到性能，不一定立即执行这个操作；

Pinning，右上角蓝色图案，数据被强制钉在某一层，不受流动规则控制。

要素 2：数据变动规则

上面这些基本数据变动的“行为”，是属于系统侧的定义，只有它们，缓存系统是无法工作的。我们必须有规则，执行规则，才会触发上面的不同行为，引起数据真正的变动。

比如说，当一个热点数据因为最近没有访问而从 L1 Heap 挤出去的时候，Flush 行为发生了；在 L2 Disk 上的数据一直没有被访问，超过了期限，淘汰出容器。这样，这些缓存数据变动的具体行为就得到了解释，而这正是由我们预先定义好的“规则”所决定的（这里的算法不一定只是缓存队列的淘汰算法，正如你所见，淘汰可以只是多个数据变动行为中的一个而已）。

要素 3：核心 API

这里本质上反映的是缓存框架实现的时候，核心代码结构的设计。当我们把这类的代码结构设计进一步上升到规范层面，它们就可以被定义成接口，即允许不同的缓存框架可以实现同样的设计，在 Java 中，这个东西有一个官方 JSR 的版本 [JSR-107](#)。它定义了 CachingProvider、CacheManager、Cache、Cache.Entry 等几个接口。

要素 4：用户侧 API

这是指暴露给用户访问缓存的接口，比如常见的向缓存内放置一条数据的接口，或者从缓存内取出一条数据的接口。值得一提的是，我们通常见到的用户 API 都是 Map-like 的结构，即众所周知的 key-value 形式，但其实缓存框架完全可以支持其它的形式，这取决于数据访问的方式，因此这并不是一个绝对的限制。

总结思考

今天我们结合实际案例学习了缓存使用中的一些常见的“坑”，并了解了千变万化的缓存框架中一些共性的东西。希望你能够重点体会和理解缓存使用中的问题，即这把双刃剑中向着程序员和系统自己的那一刃，绕开那些已经有人踩过的坑。毕竟，失败的故事总是比成功的故事更有总结的价值。

现在，我来提两个问题，请你思考：

在你的项目中，是否使用到了缓存，在使用的过程中，是否遇到过什么问题，能否跟我们大家分享一下呢？

缓存框架我介绍了几个核心要素，但是，一个缓存框架还存在着许多的“重要特性”。那么，根据你的经验和理解，你觉得它们还有哪些呢？

好，今天的内容就到这里，对于缓存，你还有什么感悟，欢迎在留言区和我聊一聊。

扩展阅读

文中提到了布隆过滤器 (Bloom Filter)，它基于概率，用来判断存在性的数据结构，它的时间和空间复杂度往往远远低于一般的存在性判别算法，它对于“不存在”判断的正确率是 100%，但对于“存在”的判断存在错误的可能。想了解其具体设计原理，[🔗 Bloom Filters by Example](#) 这篇文章是一个很好的开始。

文中提到了 Ehcache 和 Spring 整合后，就可以使用注解的方式来建立方法缓存，如果你想进一步了解具体的配置方式，可以参见 [🔗 Spring Caching and Ehcache example](#) 这篇文章。

对于 JSR-107，如果你对文中介绍的核心 API 感兴趣的话，请移步 GitHub 上的 [🔗 jsr107spec](#) 项目。

 极客时间

全栈工程师修炼指南

从全栈入门到技能实战

熊焱

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 21 | 赫赫有名的双刃剑：缓存（上）

下一篇 23 | 知其然，知其所以然：数据的持久化和一致性

精选留言 (2)

写留言



leslie

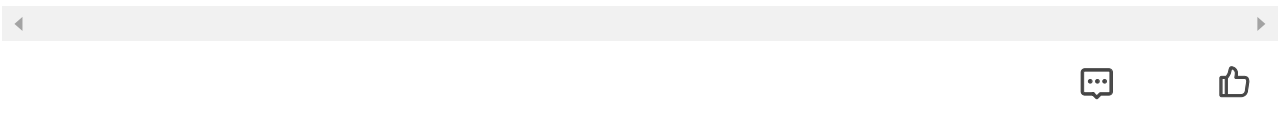
2019-10-30

其实缓存的使用各种中间件存储使用：它已经比过去廉价多了。整改中间件存储都离不开缓存的使用。

简单的说，过去数据库的缓存设置的偏小，现在的设置值完全不同了；毕竟直接重新查询结果集的代价比从缓存池要大许多。数据库之上的各种库几乎都没有离开cache的使用：框架既是优化同样是束缚；用了框架你的创造力就局限了，在此之上想提升性能的空间...

展开 ∨

作者回复: 🙏



_CountingStars

2019-10-30

布隆过滤器老师说反了吧！不存在准确率100 存在准确率不是100 它说没有见过你 就是一定没有见过你 它说见过你 其实可能没有见过你

展开 ∨

作者回复: 感谢提醒。对，说不存在是一定的；说存在是不一定的。已经知会编辑修正。

