



链滴

性能设计 - 缓存

作者: [jianzh5](#)

原文链接: <https://ld246.com/article/1587601813950>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

基本上来说，在分布式系统中最耗性能的地方就是最后端的数据库了。一般来说，只要小心维护好，数据库四种操作（select、update、insert 和 delete）中的三个写操作 insert、update 和 delete 不会出现性能问题（insert 一般不会有性能问题，update 和 delete 一般会有主键，所以也不会太慢）除非索引建得太多，而数据库里的数据又太多，这三个操作才会变慢。

绝大多数情况下，select 是出现性能问题最大的地方。一方面，select 会有很多像 join、group、order、like 等这样丰富的语义，而这些语义是非常耗性能的；另一方面，大多数应用都是读多写少，所以刷了慢查询的问题。

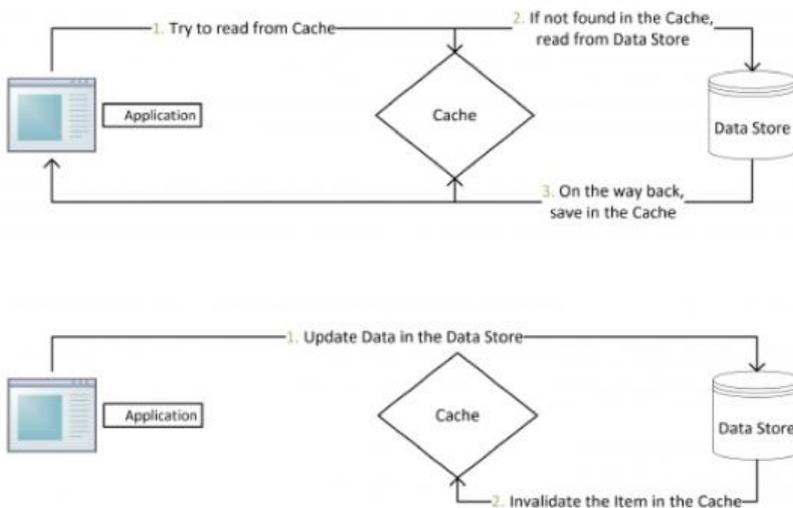
分布式系统中远程调用也会消耗很多资源，因为网络开销会导致整体的响应时间下降。为了挽救这样性能开销，在业务允许的情况（不需要太实时的数据）下，使用缓存是非常必要的事情。从另一个方面说，缓存在今天的移动互联网中是必不可少的一部分，因为网络质量不一定永远是最好的，所以前端会为所有的 API 加上缓存。不然，网络不通畅的时候，没有数据，前端都不知道怎么展示 UI 了。既因为移动互联网的网络质量而导致我们必须容忍数据的不实时性，那么，从业务上来说，在大多数情况下是可以使用缓存的。

缓存是提高性能最好的方式，一般来说，缓存有以下三种模式。

Cache Aside 更新模式

这是最常用的设计模式了，其具体逻辑如下。

- **失效**：应用程序先从 Cache 取数据，如果没有得到，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。



这是标准的设计模式，包括 Facebook 的论文《Scaling Memcache at Facebook》中也使用了这个略。

那么，是不是这个 Cache Aside 就不会有并发问题了？不是的。比如，一个是读操作，但是没有命中缓存，就会到数据库中取数据。而此时来了一个写操作，写完数据库后，让缓存失效，然后之前的那个读操作再把老的数据放进去，所以会造成脏数据。

这个案例理论上会出现，但实际上出现的概率可能非常低，因为这个条件需要发生在读缓存时缓存失效，而且有一个并发的写操作。实际上数据库的写操作会比读操作慢得多，而且还要锁表，读操作必须写操作前进入数据库操作，又要晚于写操作更新缓存，所有这些条件都具备的概率并不大。

Read/Write Through 更新模式

我们可以看到，在上面的 Cache Aside 套路中，应用代码需要维护两个数据存储，一个是缓存 (cach)，一个是数据库 (repository)。所以，应用程序比较啰嗦。而 Read/Write Through 套路是把更数据库 (repository) 的操作由缓存自己代理了，所以，对于应用层来说，就简单很多了。可以理解，应用认为后端就是一个单一的存储，而存储自己维护自己的 Cache。

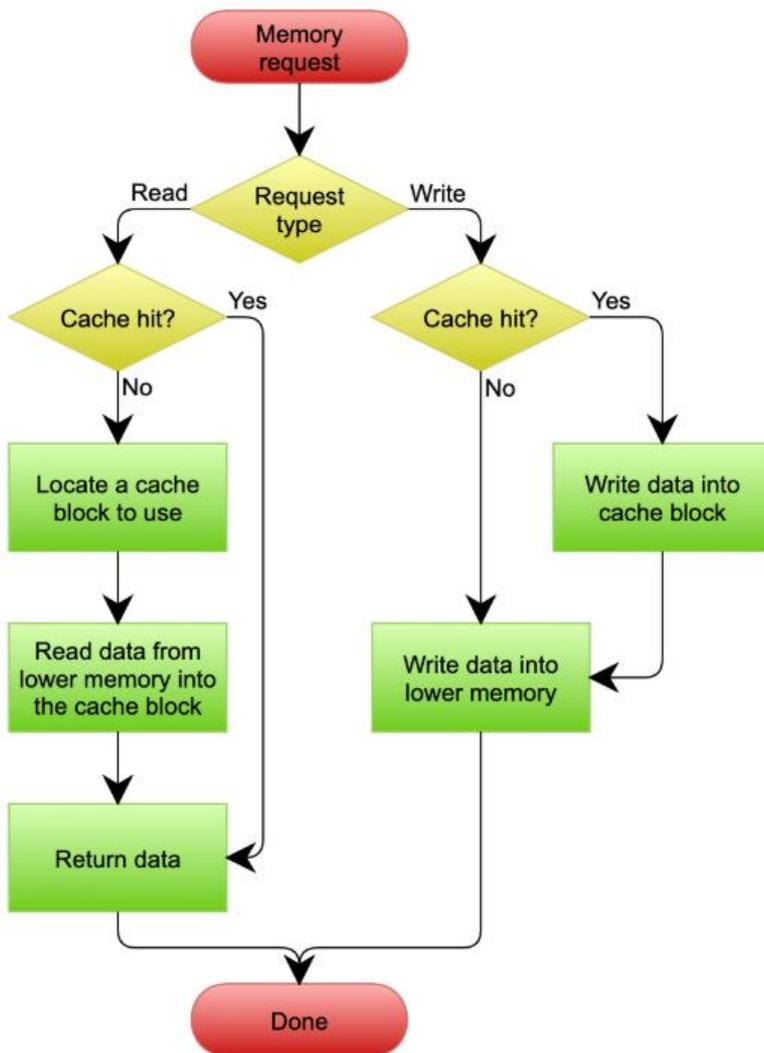
Read Through

Read Through 套路就是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或 LRU 换出，Cache Aside 是由调用方负责把数据加载入缓存，而 Read Through 则用缓存服务自己来加载，而对应用方是透明的。

Write Through

Write Through 套路和 Read Through 相仿，不过是在更新数据时发生。当有数据更新的时候，如没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后由 Cache 自己更数据库（这是一个同步操作）。

下图自来 Wikipedia 的 Cache 词条。其中的 Memory，你可以理解为就是我们例子里的数据库。



Write Behind Caching 更新模式

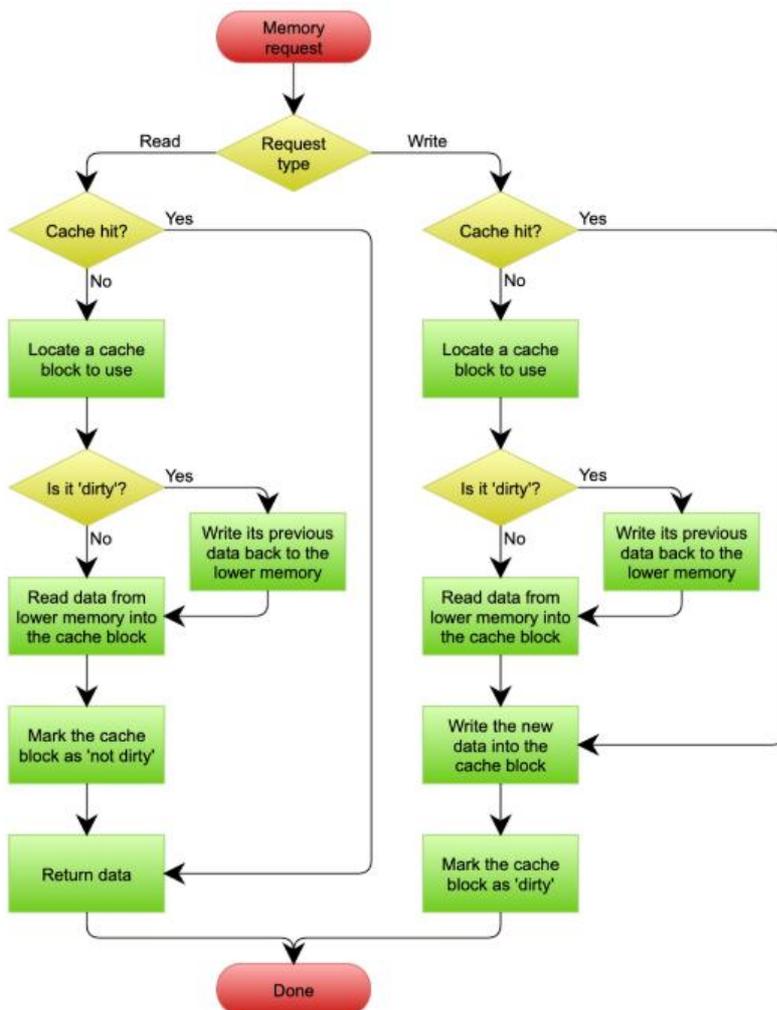
Write Behind 又叫 Write Back。一些了解 Linux 操作系统内核的同学对 write back 应该非常熟悉，这不就是 Linux 文件系统的 page cache 算法吗？是的，你看基础知识全都是相通的。所以，基础很重要，我已经说过不止一次了。

Write Back 套路就是，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的 I/O 操作飞快无比（因为直接操作内存嘛）。因为异步，rite Back 还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。

但其带来的问题是，数据不是强一致性的，而且可能会丢失（我们知道 Unix/Linux 非正常关机会导致数据丢失，就是因为这个事）。在软件设计上，我们基本上不可能做出一个没有缺陷的设计，就像设计中的时间换空间、空间换时间一个道理。有时候，强一致性和高性能，高可用和高性能是有冲突的。软件设计从来都是 trade-off（取舍）。

另外，Write Back 实现逻辑比较复杂，因为它需要 track 有哪些数据是被更新了的，需要刷到持久层。操作系统的 Write Back 会在仅当这个 Cache 需要失效的时候，才会把它真正持久起来。比如，内存不够了，或是进程退出了等情况，这又叫 lazy write。

在 Wikipedia 上有一张 Write Back 的流程图，基本逻辑可以在下图中看到。



缓存设计的重点

缓存更新的模式基本如前面所说，不过这还没完，缓存已经成为高并发高性能架构的一个关键组件了。现在，很多公司都在用 Redis 来搭建他们的缓存系统。一方面是因为 Redis 的数据结构比较丰富。另一方面，我们不能在 Service 内放 Local Cache，一是每台机器的内存不够大，二是我们的 Service 有个实例，负载均衡器会把请求随机分布到不同的实例。缓存需要在所有的 Service 实例上都建好，这我们的 Service 有了状态，更难管理了。

所以，在分布式架构下，一般都需要一个外部的缓存集群。关于这个缓存集群，你需要保证的是内存足够大，网络带宽也要好，因为缓存本质上是个内存和 IO 密集型的应用。

另外，如果需要内存很大，那么你还要动用数据分片技术来把不同的缓存分布到不同的机器上。这样可以保证我们的缓存集群可以不断地 scale 下去。关于数据分片的事，我会在后面讲述。

缓存的好坏要看命中率。缓存的命中率高说明缓存有效，一般来说命中率达到 80% 以上就算很高了。然，有的网络为了追求更高的性能，要做到 95% 以上，甚至可能会把数据库里的数据几乎全部装进缓存中。这当然是不必要的，也是没有效率的，因为通常来说，热点数据只会是少数。

另外，缓存是通过牺牲强一致性来提高性能的，这世上任何事情都不是免费的，所以并不是所有的业都适合用缓存，这需要在设计的时候仔细调研好需求。使用缓存提高性能，就是会有数据更新的延迟。

缓存数据的时间周期也需要好好设计，太长太短都不好，过期期限不宜太短，因为可能导致应用程序断从数据存储检索数据并将其添加到缓存。同样，过期期限不宜太长，因为这会导致一些没人访问的数据还在内存中不过期，而浪费内存。

使用缓存的时候，一般会使用 LRU 策略。也就是说，当内存不够需要数据被清出内存时，会找最活跃的数据清除。所谓最不活跃的意思是最长时间没有被访问过了。所以，开启 LRU 策略会让缓存每个数据访问的时候把其调到前面，而要淘汰数据时，就从最后面开始淘汰。

于是，对于 LRU 的缓存系统来说，其需要在 key-value 这样的非顺序的数据结构中维护一个顺序的数据结构，并在读缓存时，需要改变被访问数据在顺序结构中的排位。于是，我们的 LRU 在读写时都要加锁（除非是单线程无并发），因此 LRU 可能会导致更慢的缓存存取的时间。这点要小心。

最后，我们的世界是比较复杂的，很多网站都会被爬虫爬，要小心这些爬虫。因为这些爬虫可能会爬一些很古老的数据，而程序会把这些数据加入到缓存中去，而导致缓存中那些真实的热点数据被挤出（因为机器的速度足够快）。对此，一般来说，我们需要有一个爬虫保护机制，或是我们引导这些人使用我们提供的外部 API。在那边，我们可以有针对性地做多租户的缓存系统（也就是说，把用户和三方开发者的缓存系统分离开来）。

文章来源：左耳听风

文章地址：<https://time.geekbang.org/column/article/6282>