性能设计篇之"缓存"

2018-05-10 陈皓

前面分享了《分布式系统设计模式》系列文章的前两部分——弹力设计篇和管理设计篇。今天开始这一系列的最后一部分内容——性能设计篇，主题为《性能设计篇之"缓存"》。

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不太会出现性能问题（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，所以加剧了慢查询的问题。

分布式系统中远程调用也会耗很多性能，因为有网络开销，会导致整体的响应时间下降。为了挽救这样的性能开销，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。

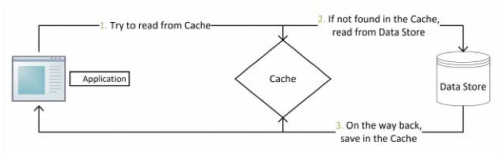
从另一个方面说，缓存在今天的移动互联网中是必不可少的一部分，因为网络质量不总是最好的，所以前端也会为所有的API加上缓存。不然，网络不通畅的时候，没有数据，前端都不知道怎么展示UI了。既然因为移动互联网的网络质量而导致我们必需容忍数据的不实时性，那么，从业务上来说，在大多数情况下是可以使用缓存的。

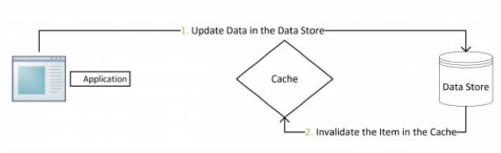
缓存是提高性能最好的方式，一般来说，缓存有以下三种模式。

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，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》中也使用了这个策略。为什么不是写完数据库后更新缓存？你可以看一下Quora上的这个问答《Why does Facebook use delete to remove the key-value pair in Memcached instead of updating the Memcached during write request to the backend?》，主要是怕两个并发的写操作导致脏数据。

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，而读操作必需在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。

所以，这也就是Quora上的那个答案里说的，要么通过2PC或是Paxos协议保证一致性，要么就是拼命地降低并发时脏数据的概率。而Facebook使用了这个降低概率的玩法，因为2PC太慢，而Paxos太复杂。当然，最好还是为缓存设置好过期时间。

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（cache），一个是数据库（repository）。所以，应用程序比较啰嗦。
而Read/Write Through套路是把更新数据库（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。可以理解为，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。

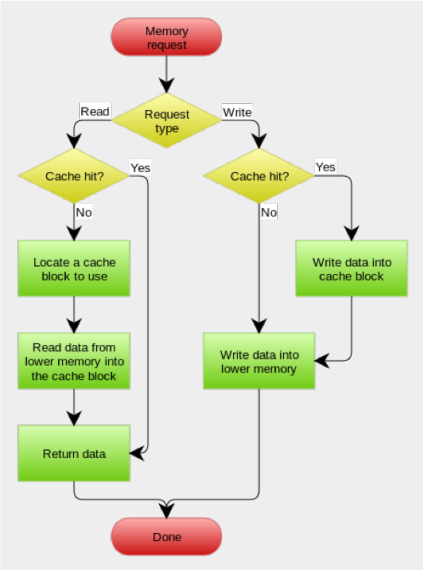
Read Through

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，从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，如果没有命中缓存，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自己更新数据库（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

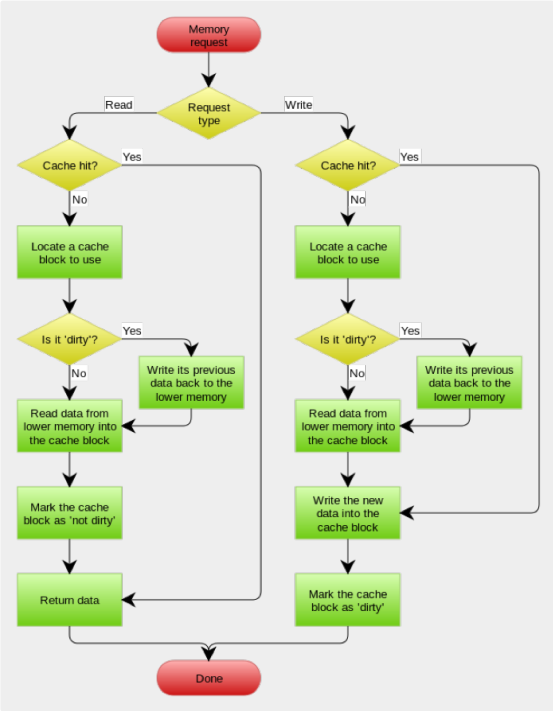
Write Behind又叫Write Back。一些了解Linux操作系统内核的同学对write back应该非常熟悉，这不就是Linux文件系统的page cache算法吗？是的，你看基础知识全都是相通的。所以，基础很重要，我已经说过不止一次了。

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，Write 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，就像算法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，需要刷到持久层上。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有多个实例，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，这让我们的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要足够大，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，一般来说命中率到80%以上就算很高了。当然，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务都适合用缓存，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不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，当内存不够需要有数据被清出内存时，会找最不活跃的数据清除。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策略会让缓存在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在读写时都需要加锁（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到一些很古老的数据，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去（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去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，把用户和第三方开发者的缓存系统分离开来）。

小结

好了，我们来总结一下今天分享的主要内容。首先，缓存是为了加速数据访问，在数据库之上添加的一层机制。然后，我讲了几种典型的缓存模式，包括Cache Aside、Read/Write Through和Write Behind Caching以及它们各自的优缺点。

最后，我介绍了缓存设计的重点，除了性能之外，在分布式架构下和公网环境下，对缓存集群、一致性、LRU的锁竞争、爬虫等多方面都需要考虑。下篇文章中，我们讲述异步处理。希望对你有帮助。

也欢迎你分享一下你接触到的缓存方式有哪些？怎样权衡一致性和缓存的效率？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
  - 认识故障和弹力设计
  - 隔离设计Bulkheads
  - 异步通讯设计Asynchronous

- 幂等性设计Idempotency
- 服务的状态State
- 补偿事务Compensating Transaction
- 重试设计Retry
- 熔断设计Circuit Breaker
- 限流设计Throttle
- 降级设计degradation
- 弹力设计总结

- 管理设计篇
  - 分布式锁Distributed Lock
  - 配置中心Configuration Management
  - 边车模式Sidecar
  - 服务网格Service Mesh
  - 网关模式Gateway
  - 部署升级策略
- 性能设计篇
  - 缓存Cache
  - 异步处理Asynchronous
  - 数据库扩展
  - 秒杀Flash Sales
  - 边缘计算Edge Computing

翎逸

2018-05-10

感觉更多的是不是应该说下缓存的监控，雪崩，缓存和数据库的一致性，以及热点缓存处理等一些场景的处理，这样会觉得更深入一些

Black

2018-05-10

这篇的内容有大部分是跟之前博客上的一篇 缓存更新的套路 重复了

作者回复

2018-05-10

是的。这是为了整个系列的完整。

W_T

2018-05-10

Read/Write Through 模式中对数据库的操作一定要交给交给缓存代理么，如果是这样就会带来两个问题：
1. 需要在缓存服务中实现数据库操作的代码，我从来没有这么做过，也不清楚目前主流缓存是否支持这样的操作。
2. 缓存服务与数据库之间建立了依赖。
我在工作中更常见的做法是由应用服务操作缓存以及数据库，这样的话感觉就跟前面的cache aside模式很像了。
可能是我对Read/Write Through模式理解不深，说错的地方还请老师指正

MarksGui

2018-06-01

皓哥，对于很多需要统计的数据或者筛选条件复杂的怎么利用缓存了？

A圆规

2018-06-01

Cache aside 需要处理并发读问题，缓存失效时多个读会打到数据库

李书德

2018-05-15

这篇和博客的很相似

---

river

2018-05-14

Cache aside 更新数据库 然后失效缓存。失效时读不到缓存，不是会打到数据库的流量很高？

---

river

2018-05-13

Cache aside 更新数据库 然后失效缓存，在读很高的情况下，会不会相当于缓存被击穿？

作者回复

2018-05-14

怎么会呢?

FF

2018-05-11

write through 这种模式，如果没有命中缓存更新数据库后返回，后面缓存谁来更新？这种更新想想好像很复杂

坤

2018-05-10

陈皓老师好，redis 分片热点问题，有没有什么好的解决方案?

作者回复

2018-05-10

建数据索引服务