High一下!

酷 壳 – CoolShell
享受编程和技术所带来的快乐 – Coding Your Ambition
(https://coolshell.cn/)

Search …

缓存更新的套路

2016年07月27日 (Https://Coolshell.Cn/Articles/17416.Html) 陈皓 (Https://Coolshell.Cn/Articles/Author/Haoel) 123,710 人阅读

看到好些人在写更新缓存数据代码时，**先删除缓存，然后再更新数据库**，而后续的操作会把数据再装载的缓存中。**然而，这个是逻辑是错误的。**试想，两个并发操作，一个是更新操作，另一个是查询操作，更新操作删除缓存后，查询操作没有命中缓存，先把老数据读出来后放到缓存中，然后更新操作更新了数据库。于是，在缓存中的数据还是老的数据，导致缓存中的数据是脏的，而且还一直这样脏下去了。

我不知道为什么这么多人用的都是这个逻辑，当我在微博上发了这个贴以后，我发现好些人给了好多非常复杂和诡异的方案，所以，我想写这篇文章说一下几个缓存更新的Design Pattern（让我们多一些套路吧）。

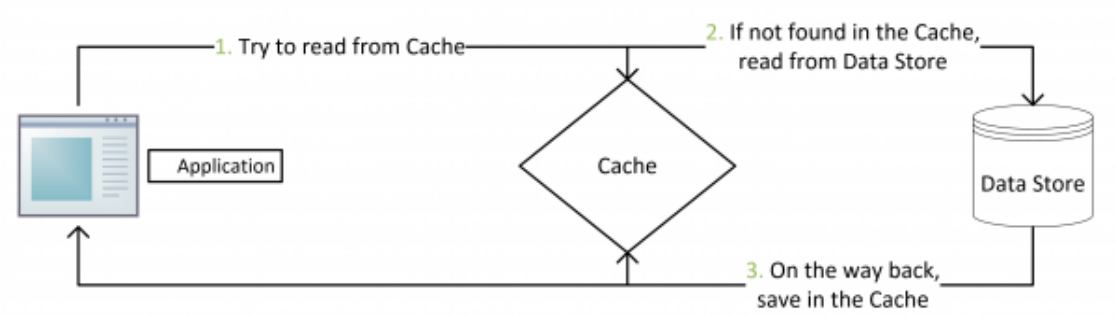
这里，我们先不讨论更新缓存和更新数据这两个事是一个事务的事，或是会有失败的可能，我们先假设更新数据库和更新缓存都可以成功的情况（我们先把成功的代码逻辑先写对）。

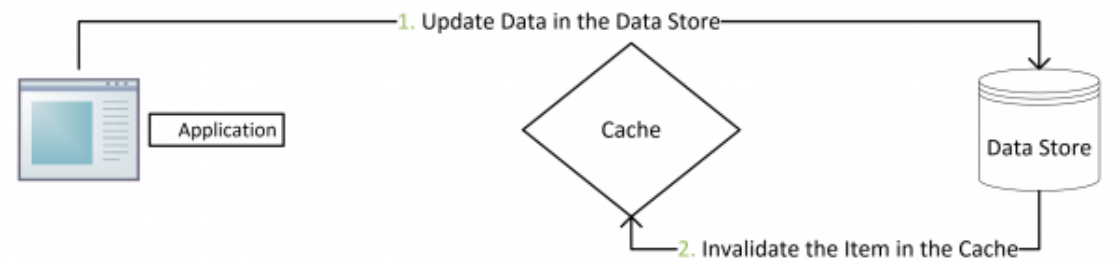
更新缓存的的Design Pattern有四种：Cache aside, Read through, Write through, Write behind caching，我们下面一一来看一下这四种Pattern。

Cache Aside Pattern

这是最常用最常用的pattern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。





注意，我们的更新是先更新数据库，成功后，让缓存失效。那么，这种方式是否可以没有文章前面提到过的那个问题呢？我们可以脑补一下。

一个是查询操作，一个是更新操作的并发，首先，没有了删除cache数据的操作了，而是先更新了数据库中的数据，此时，缓存依然有效，所以，并发的查询操作拿的是没有更新的数据，但是，更新操作马上让缓存的失效了，后续的查询操作再把数据从数据库中拉出来。而不会像文章开头的那个逻辑产生的问题，后续的查询操作一直都在取老的数据。

这是标准的design pattern，包括Facebook的论文《Scaling Memcache at Facebook (https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf)》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? (https://www.quora.com/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Cache Aside这个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，然后就到数据库中取数据，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。

但，这个case理论上会出现，不过，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，而且并发着有一个写操作。而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，而又要晚于写操作更新缓存，所有的这些条件都具备的概率基本并不大。

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，要么就是拼命的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上过期时间。

Read/Write Through Pattern

我们可以看到，在上面的Cache Aside套路中，我们的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。

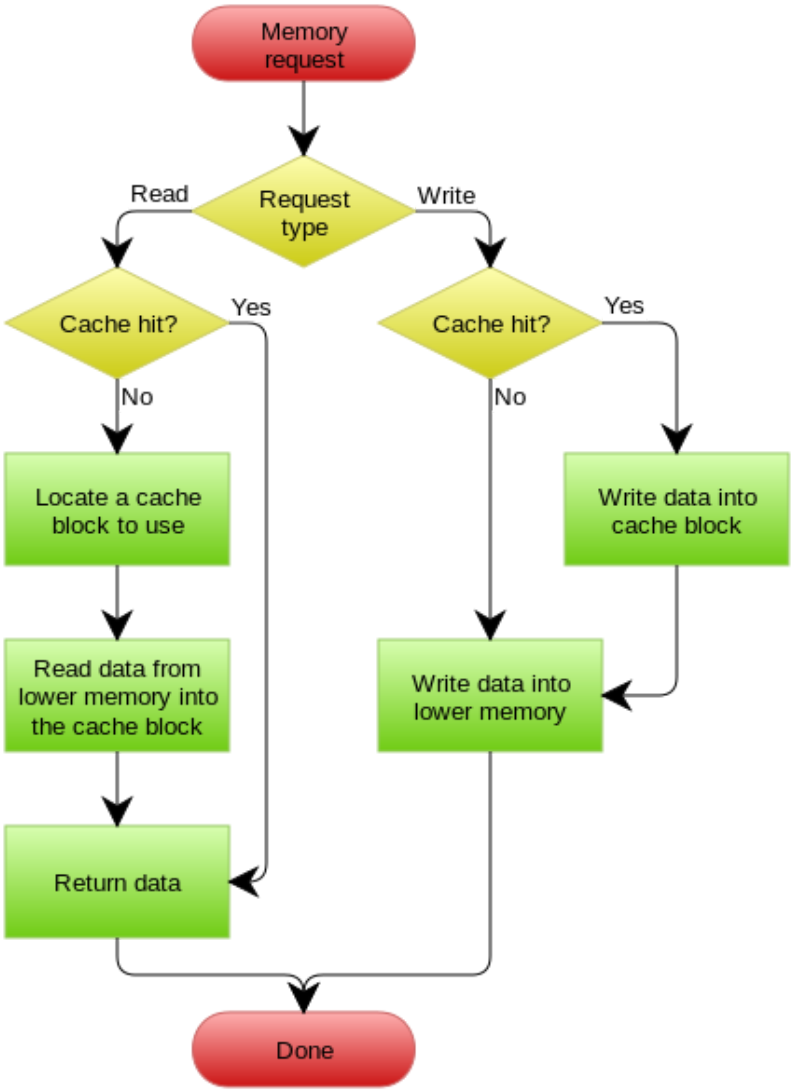
Read Through

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，然后再由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 (https://en.wikipedia.org/wiki/Cache_(computing))。其中的Memory你可以理解为就是我们例子里的数据库。



Write Behind Caching Pattern

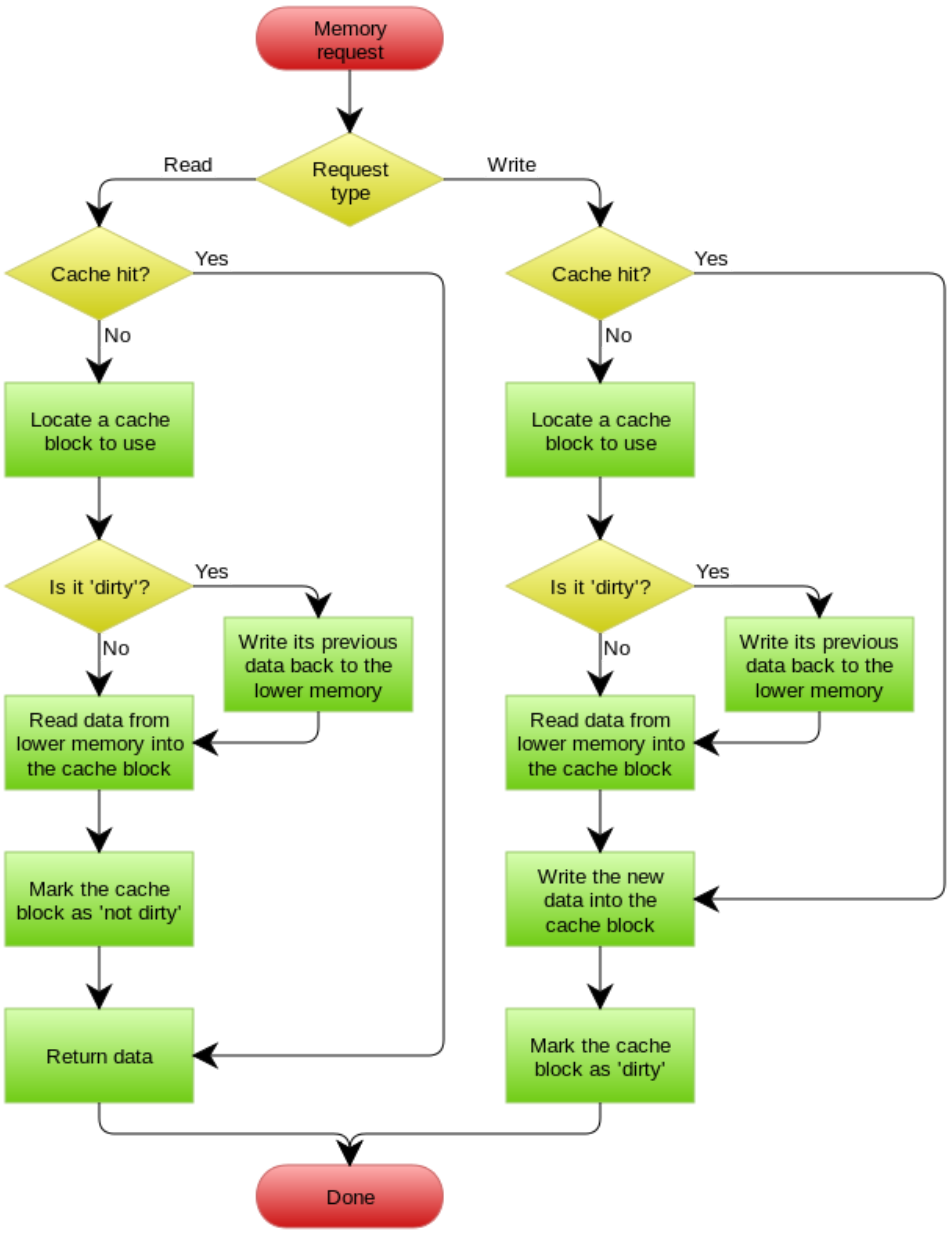
Write Behind 又叫 Write Back。**一些了解Linux操作系统内核的同学对write back应该非常熟悉，这不就是Linux文件系统的Page Cache的算法吗？是的，你看基础这玩意全都是相通的。**所以，基础很重要，我已经不是一次说过基础很重要这事了。

Write Back套路，一句说就是，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 backg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。

但是，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，高可用和高性性是有冲突的。软件设计从来都是取舍Trade-Off。

另外，Write Back实现逻辑比较复杂，因为他需要track有哪数据是被更新了的，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，才会被真正持久起来，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，基本逻辑如下：



再多唠叨一些

1）上面讲的这些Design Pattern，其实并不是软件架构里的mysql数据库和memcache/redis的更新策略，这些东西都是计算机体系结构里的设计，比如CPU的缓存，硬盘文件系统中的缓存，硬盘上的缓存，数据库中的缓存。**基本上来说，这些缓存更新的设计模式都是非常老古董的，而且历经长时间考验的策略**，所以这也就是，工程学上所谓的Best Practice，遵从就好了。

2）有时候，我们觉得能做宏观的系统架构的人一定是很有经验的，其实，宏观系统架构中的很多设计都来源于这些微观的东西。比如，云计算中的很多虚拟化技术的原理，和传统的虚拟内存不是很像么？Unix下的那些I/O模型，也放大到了架构里的同步异步的模型，还有Unix发明的管道不就是数据流式计算架构吗？TCP的好些设计也用在不同系统间的通讯中，仔细看看这些微观层面，你会发现有很多设计都非常精妙……所以，**请允许我在这里放句观点鲜明的话——如果你要做好架构，首先你得把计算机体系结构以及很多老古董的基础技术吃透了。**

3）在软件开发或设计中，我非常建议在之前先去参考一下已有的设计和思路，**看看相应的guideline，best practice或design pattern，吃透了已有的这些东西，再决定是否要重新发明轮子。**千万不要似是而非地，想当然的做软件设计。

4）上面，我们没有考虑缓存（Cache）和持久层（Repository）的整体事务的问题。比如，更新Cache成功，更新数据库失败了怎么吗？或是反过来。关于这个事，如果你需要强一致性，你需要使用“两阶段提交协议”——prepare, commit/rollback，比如Java 7 的 XAResource (http://docs.oracle.com/javaee/7/api/javax/transaction/xa/XAResource.html)，还有MySQL 5.7 的 XA Transaction (http://dev.mysql.com/doc/refman/5.7/en/xa.html)，有些cache也支持XA，比如EhCache (http://www.ehcache.org/documentation/3.0/xa.html)。当然，XA这样的强一致性的玩法会导致性能下降，关于分布式的事务的相关话题，你可以看看《分布式系统的事务处理 (https://coolshell.cn/articles/10910.html)》一文。

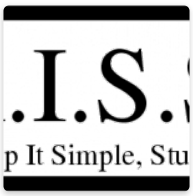
（全文完）

关注CoolShell微信公众账号和微信小程序

相关文章

(https://coolshell.cn/articles/7236.html)

用Unix的设计思想来应对多变的需求 (https://coolshell.cn/articles/7236.html)



(https://coolshell.cn/articles/8961.html)

从面向对象的设计模式看软件设计 (https://coolshell.cn/articles/8961.html)

(https://coolshell.cn/articles/17680.html)

从Gitlab误删除数据库想到的 (https://coolshell.cn/articles/17680.html)

(https://coolshell.cn/articles/17459.html)

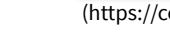
关于高可用的系统 (https://coolshell.cn/articles/17459.html)


(https://coolshell.cn/articles/9949.html)

IoC/DIP其实是一种管理思想 (https://coolshell.cn/articles/9949.html)


(https://coolshell.cn/articles/6950.html)

需求变化与IoC (https://coolshell.cn/articles/6950.html)

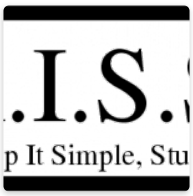

(66 人打了分，平均分： **4.79**)

Unix/Linux (Https://Coolshell.Cn/Category/Operatingsystem/Unixlinux), 程序设计 (Https://Coolshell.Cn/Category/Progdesign)
Cache (Https://Coolshell.Cn/Tag/Cache), Design (Https://Coolshell.Cn/Tag/Design), Design Pattern (Https://Coolshell.Cn/Tag/Design-Pattern), Linux (Https://Coolshell.Cn/Tag/Linux)

相关文章

(https://coolshell.cn/articles/7236.html)

用Unix的设计思想来应对多变的需求 (https://coolshell.cn/articles/7236.html)



(https://coolshell.cn/articles/8961.html)

从面向对象的设计模式看软件设计 (https://coolshell.cn/articles/8961.html)

(https://coolshell.cn/articles/17680.html)

从Gitlab误删除数据库想到的 (https://coolshell.cn/articles/17680.html)

(https://coolshell.cn/articles/17459.html)

关于高可用的系统 (https://coolshell.cn/articles/17459.html)


(https://coolshell.cn/articles/9949.html)

IoC/DIP其实是一种管理思想 (https://coolshell.cn/articles/9949.html)


(https://coolshell.cn/articles/6950.html)

需求变化与IoC (https://coolshell.cn/articles/6950.html)

《缓存更新的套路》的相关评论

Pingback： 面试前必须要知道的Redis面试题 – 技术成就梦想 (http://sparkgis.com/2019/01/14/%e9%9d%a2%e8%af%95%e5%89%8d%e5%bf%85%e9%a1%bb%e8%a6%81%e7%9f%a5

Pingback： 分布式之数据库和缓存双写一致性方案解析 – CSharp (http://www.csharp.me/158.html)

Pingback： 面试前必须要知道的Redis面试题 | 天英星网 (http://www.wxn.fun/?p=3687)

Pingback： Redis cache policy – DDCODE (https://ddcode.net/2019/04/14/redis-cache-policy/)

Pingback： 分布式之数据库和缓存双写一致性方案解析 – 前端开发，JQUERY特效，全栈开发，vue开发 (https://www.jqhtml.com/39177.html)