高并发缓存一致性方案梳理

大部分面向公众的互联网系统，其并发请求数量与在线用户数量都是正相关的，而 MySQL能够承担的并发读写量是有一定上限的，当系统的访问量超过一定程度的时候，纯MySQL就很难应付了。

绝大多数互联网系统都是采用MySQL+Redis这对经典组合来解决高并发问题的。Redis作为MySQL 的前置缓存，可以应对绝大部分查询请求，从而在很大程度上缓解 MySQL并发请求的压力，但是不能一说到缓存脑海中就只有Redis，这无论在工作还是面试中都不合适，所以我们先全面了解缓存。

缓存不仅仅是Redis

缓存的意义

从浏览器到网络，再到应用服务器，甚至到数据库，通过在各个层面应用缓存技术，整个系统的性能将大幅提高。

例如，缓存离客户端更近（缓存一定是离用户越近越好），从缓存请求内容比从源服务器所用时间更少，呈现速度更快，系统就显得更灵敏。缓存数据的重复使用，大大降低了用户的带宽使用，其实也是一种变相的省钱，同时保证了带宽请求在一个低水平上，更容易维护。

所以，使用缓存技术，可以降低系统的响应时间，减少网络传输时间和应用延迟时间，进而提高了系统的吞吐量，增加了系统的并发用户数。利用缓存还可以最小化系统的工作量，使用了缓存就可以不必反复从数据源中查找，缓存所创建或提供的同一条数据更好地利用了系统的资源，并且能够更好的保护后端的系统和服务。

因此，缓存是系统调优时常用且行之有效的手段，无论是操作系统还是应用系统，缓存策略无处不在。

缓存的分类

客户端缓存

客户端缓存相对于其他端的缓存而言，要简单一些，而且通常是和服务端以及网络侧的应用或缓存配合使用的。对于互联网应用而言，也就是通常所说的BS架构应用，可以分为页面缓存和浏览器缓存。对于移动互联网应用而言，是指APP自身所使用的缓存。

页面缓存

页面缓存有两层含义：一个是页面自身对某些元素或全部元素进行缓存;另一层意思是服务端将静态页面或动态页面的元素进行缓存，然后给客户端使用。这里的页面缓存指的是页面自身的缓存或者离线应用缓存。

页面缓存是将之前渲染的页面保存为文件，当用户再次访问时可以避开网络连接，从而减少负载，提升性能和用户体验。

HTML5提供的离线应用缓存机制，使得网页应用可以离线使用，这种机制在浏览器上支持度非常广，可以放心地使用该特性来加速页面的访问，比如我们商城系统的前端页面中就使用了localstorage。

浏览器缓存

浏览器缓存是根据一套与服务器约定的规则进行工作的，工作规则很简单:检查以确保副本是最新的，通常只要一次会话。浏览器会在硬盘上专门开辟一个空间来存储资源副本作为缓存。在用户触发“后退”操作或点击一个之前看过的链接的时候，浏览器缓存会很管用。同样，如果访问系统中的同一张图片，该图片可以从浏览器缓存中调出并几乎立即显现出来。

对浏览器而言，HTTP1.0提供了一些很基本的缓存特性，例如在服务器侧设置 Expires的HTTP头来告诉客户端在重新请求文件之前缓存多久是安全的，可以通过if-modified-since 的条件请求来使用缓存，还有Cache-Control等等。HTTP 1.1有了较大的增强，缓存系统被形式化了，引入了实体标签e-tag等。

通过在HTML页面的节点中加入meta标签，可以告诉浏览器当前页面不被缓存，每次访问都需要去服务器拉取。

APP上的缓存

尽管混合编程(hybrid programming)成为时尚，但整个移动互联网目前还是原生应用(以下简称APP)的天下。无论大型或小型APP，灵活的缓存不仅大大减轻了服务器的压力，

而且因为更快速的用户体验而方便了用户。如何把APP缓存对于业务组件透明，以及APP缓存数据的及时更新，是APP缓存能否成功应用起来的关键。APP可以将内容缓存在内存、文件或本地数据库（例如SQLite)中

网络缓存

网络中的缓存位于客户端和服务端之间，代理或响应客户端的网络请求，从而对重复的请求返回缓存中的数据资源。同时，接受服务端的请求，更新缓存中的内容。

Web代理缓存

Web代理几乎是伴随着互联网诞生的，常用的 Web代理分为正向代理、反向代理和透明代理。Web代理缓存是将Web代理作为缓存的一种技术。

为了从源服务器取得内容，用户向代理服务器发送一个请求并指定目标服务器，然后代理服务向源服务器转交请求并将获得的内容返回给客户端。一般地，客户端要进行一些特别的设置才能使用正向代理。

反向代理与正向代理相反，对于客户端而言代理服务器就像是源服务器，并且客户端不需要进行设置。客户端向反向代理发送普通请求，接着反向代理将判断向何处转发请求,并将从源服务器获得的内容返回给客户端。

透明代理的意思是客户端根本不需要知道有代理服务器的存在，由代理服务器改变客户端请求的报文字段，并会传送真实的IP地址。加密的透明代理属于匿名代理，不用设置就可以使用代理了。透明代理的例子就是时下很多公司使用的行为管理软件。

Web正向代理代理缓存是指使用正向代理的缓存技术。Web代理缓存的作用跟浏览器的内置缓存类似,只是介于浏览器和互联网之间。

当通过代理服务器进行网络访问时，浏览器不是直接到Web服务器去取回网页而是向Web代理发出请求，由代理服务器来取回浏览器所需要的信息并传送给浏览器。而且，Web代理缓存有很大的存储空间，不断将新获取的数据储存到本地的存储器上，如果浏览器所请求的数据在Web代理的缓存上已经存在而且是最新的，那么就不重新从Web服务器取数据，而是直接将缓存的数据传送给用户的浏览器，这样就能显著提高浏览速度和效率。对于企业而言，使用Web代理既可以节省成本，又能提高性能。

对于Web代理缓存而言，较流行的是Squid，它支持建立复杂的缓存层级结构，拥有详细的日志、高性能缓存以及用户认证支持。Squid同时支持各种插件。

使用web反向代理服务器和使用正向代理服务器一样，可以拥有缓存的作用，反向代理缓存可以缓存原始资源服务器的资源，而不是每次都要向原始资源服务器请求数据，特别是一些静态的数据，比如图片和文件，很多Web服务器就具备反向代理的功能，比如大名鼎鼎的Nginx，我们的商城系统的秒杀部分就使用了Nginx的缓存机制。

边缘缓存

如果反向代理服务器能够做到和用户来自同一个网络，那么用户访问反向代理服务器，就会得到很高质量的响应速度，所以可以将这样的反向代理缓存称为边缘缓存。边缘缓存在网络上位于靠近用户的一侧，可以处理来自不同用户的请求，主要用于向用户提供静态的内容，以减少应用服务器的介入。边缘缓存的一个有名的开源工具就是Varnish。

边缘缓存中典型的商业化服务就是CDN了，例如AWS 的 Cloud Front，我国的ChinaCache等，现在一般的公有云服务商都提供了CDN服务。CDN是Content DeliveryNetwork 的简称，即“内容分发网络”的意思。。

服务端缓存

服务端缓存是整个缓存体系中的重头戏，网站的架构演进中服务端缓存是系统性能的重中之重了。

数据库缓存

数据库属于IO密集型的应用，主要负责数据的管理及存储。数据库缓存是一类特殊的缓存，是数据库自身的缓存机制。大多数数据库不需要配置就可以快速运行，但并没有为特定的需求进行优化。在数据库调优的时候，缓存优化是一项很重要的工作。

当使用InnoDB存储引擎的时候，innodb_buffer_pool_size参数可能是影响性能的最为关键的一个参数了，用来设置用于缓存InnoDB索引及数据块的内存区域大小，简单来说，当操作一个InnoDB表的时候，返回的所有数据或者查询过程中用到的任何一个索引块，都会在这个内存区域中去查询一遍。

innodb_buffer_pool_size 设置了InnoDB存储引擎需求最大的一块内存区域的大小，直接关系到InnoDB存储引擎的性能，所以如果有足够的内存，尽可将该参数设置到足够大，将尽可能多的InnoDB的索引及数据都放入到该缓存区域中，直至全部。

比如在商城系统的MySQL中innodb_buffer_pool_size被设置为物理内存的一半：

```
[root@192-168-65-184 conf]# free -m
              total        used        free      shared  buff/cache   available
Mem:           7820        1619        4819         396        1381        5539
Swap:          8063           0        8063
```

```
[mysqld]
innodb_buffer_pool_size = 4294967296
innodb_buffer_pool_instances = 4
join_buffer_size = 33554432
sort_buffer_size = 2097152
read_rnd_buffer_size = 2097152
```

当然，数据库还有很多值得深入研究的地方，需要专业的技能，这就是很多公司专门设有DBA角色的原因。

应用级缓存

在系统开发的时候，适当地使用应用级缓存，往往可以取得事半功倍的效果。应用级缓存在这里指的是用来写带有缓存特性的应用框架，或者可用于缓存功能的专用库。

在Java语言中，缓存框架更多，例如 Ehcache、Voldemort（Voldemort是一款基于Java开发的分布式键-值缓存系统，像JBoss的缓存一样，Voldemort同样支持多台服务器之间的缓存同步，以增强系统的可靠性和读取性能。Voldemort 相当于是Amazon Dynamo的一个开源实现，LinkedIn用它解决了网站的高扩展性存储问题） 等等。

平台级缓存

在很多的时候，我们还需要考虑使用平台级缓存。

不论是Redis还是 MongoDB，以及 Memcached都可以作为平台级缓存的重要技术，当然Redis/Memcached用的更多一些，MongoDB更多的时候是做为持久化的NoSQL数据库来说使用的。

如何保证缓存的数据一致性

工程实践

只要使用到缓存，无论是本地内存做缓存还是使用 redis 做缓存，那么就会存在数据同步的问题。

先读缓存数据，缓存数据有，则立即返回结果；如果没有数据，则从数据库读数据，并且把读到的数据同步到缓存里，提供下次读请求返回数据。

这样能有效减轻数据库压力，但是如果修改删除数据，因为缓存无法感知到数据在数据库的修改。这样就会造成数据库中的数据与缓存中数据不一致的问题，那该如何解决呢?

有好几种解决方案，

1、先更新缓存，再更新数据库

2、先更新数据库，再更新缓存

3、先删除缓存，后更新数据库

4、先更新数据库，后删除缓存

我们一一来看看：

更新缓存类

1、先更新缓存，再更新DB

这个方案我们一般不考虑。原因是更新缓存成功，更新数据库出现异常了，导致缓存数据与数据库数据完全不一致，而且很难察觉，因为缓存中的数据一直都存在。

2. 先更新DB，再更新缓存

这个方案也我们一般不考虑，原因跟第一个一样，数据库更新成功了，缓存更新失败，同样会出现数据不一致问题。

删除缓存类

3、先删除缓存，后更新DB

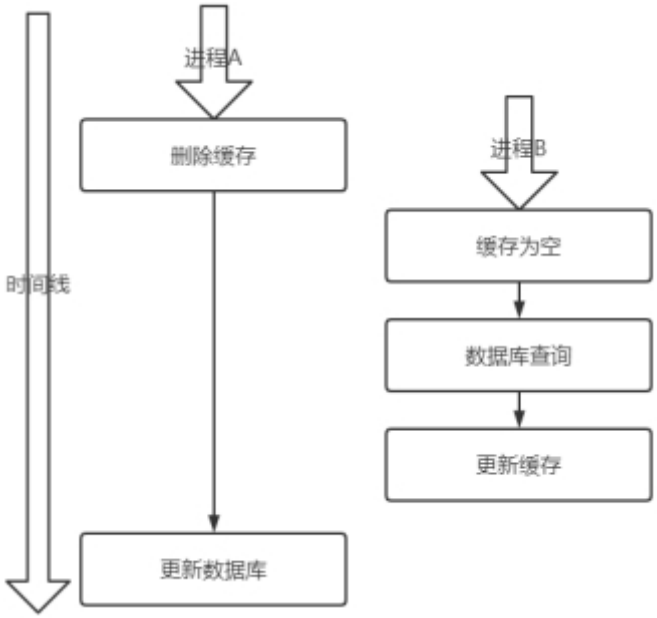
该方案也会出问题，具体出现的原因如下。

此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

请求 A 会先删除 Redis 中的数据，然后去数据库进行更新操作;

此时请求 B 看到 Redis 中的数据时空的，会去数据库中查询该值，补录到 Redis 中;

但是此时请求 A 并没有更新成功，或者事务还未提交，请求B去数据库查询得到旧值;



那么这时候就会产生数据库和 Redis 数据不一致的问题。因此一般不建议使用这种方式。

如何解决呢？其实最简单的解决办法就是延时双删的策略。就是

（1）先淘汰缓存

（2）再写数据库

（3）休眠1秒，再次淘汰缓存

这么做，可以将1秒内所造成的缓存脏数据，再次删除。

那么，这个1秒怎么确定的，具体该休眠多久呢？

针对上面的情形，自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

但是上述的保证事务提交完以后再进行删除缓存还有一个问题，就是如果你使用的是 Mysql 的读写分离的架构\的话，那么其实主从同步之间也会有时间差。

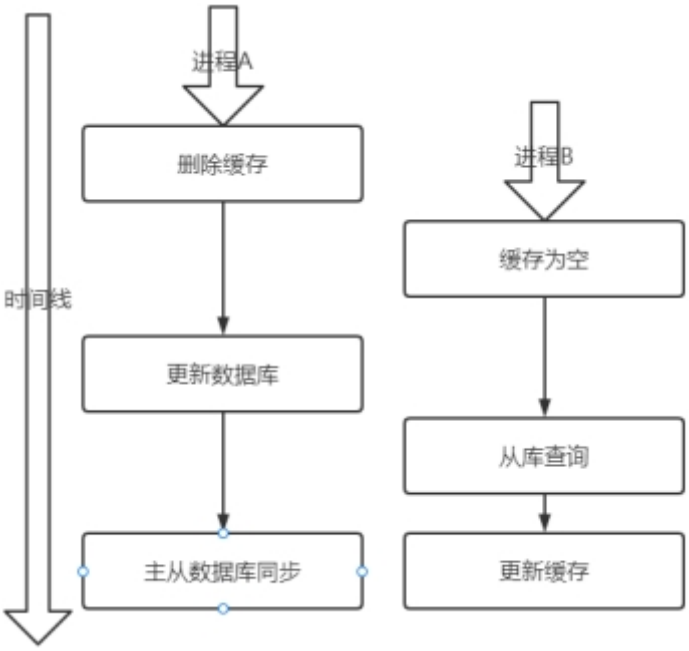
此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

请求 A 更新操作，删除了 Redis，

请求主库进行更新操作，主库与从库进行同步数据的操作，

请 B 查询操作，发现 Redis 中没有数据，

去从库中拿去数据，此时同步数据还未完成，拿到的数据是旧数据。



此时的解决办法有两个：

1、还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。

2、就是如果是对 Redis 进行填充数据的查询数据库操作，那么就强制将其指向主库进行查询。

继续深入，采用这种同步淘汰策略，吞吐量降低怎么办？

那就将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，再返回。这么做，加大吞吐量。

不过总的来说，先删除缓存值再更新数据库有可能导致请求因缓存缺失而访问数据库，给数据库带来压力； 2. 业务应用中读取数据库和写缓存的时间有时不好估算，导致延迟双删中的sleep时间不好设置。

4、先更新DB，后删除缓存

读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。更新的时候，先更新数据库，然后再删除缓存。

这种情况不存在并发问题么？

依然存在。假设这会有两个请求，一个请求A做查询操作，一个请求B做更新操作，那么会有如下情形产生

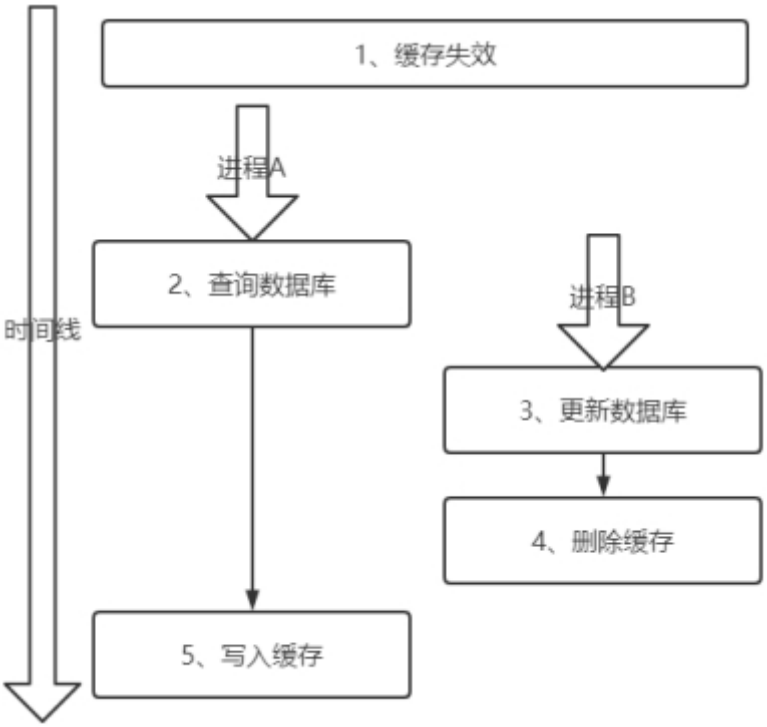
（1）缓存刚好失效

（2）请求A查询数据库，得一个旧值

（3）请求B将新值写入数据库

（4）请求B删除缓存

（5）请求A将查到的旧值写入缓存



如果发生上述情况，确实是会发生脏数据。然而，发生这种情况的概率又有多少呢？

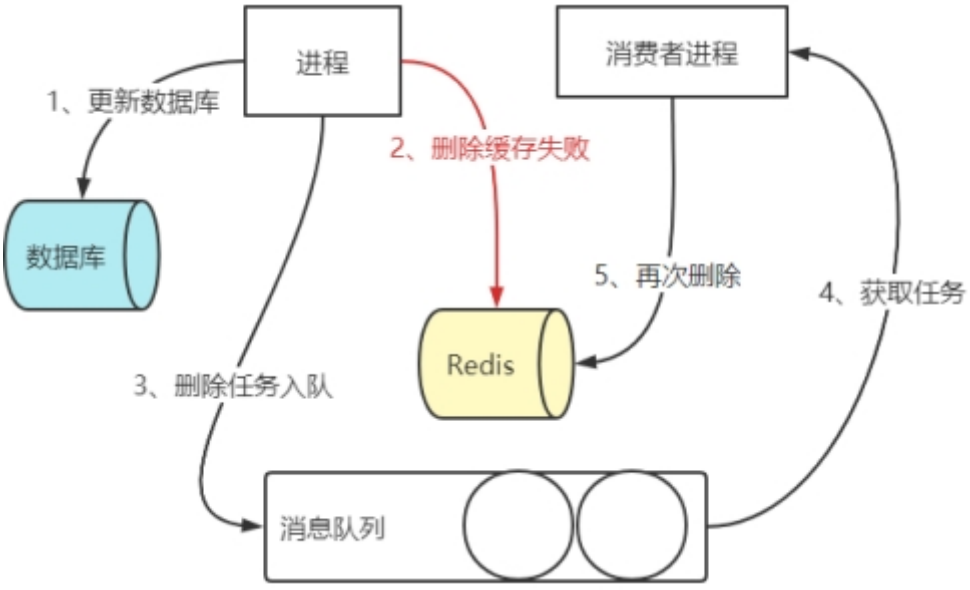
发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。一定要解决怎么办？如何解决上述并发问题？

首先，给缓存设有效时间是一种方案。

其次，采用异步延时删除策略。

但是，更新数据库成功了，但是在删除缓存的阶段出错了没有删除成功怎么办？这个问题，在删除缓存类的方案都是存在的，那么此时再读取缓存的时候每次都是错误的数据了。

此时解决方案有两个，一是就是利用消息队列进行删除的补偿。具体的业务逻辑用语言描述如下：



1、请求 A 先对数据库进行更新操作

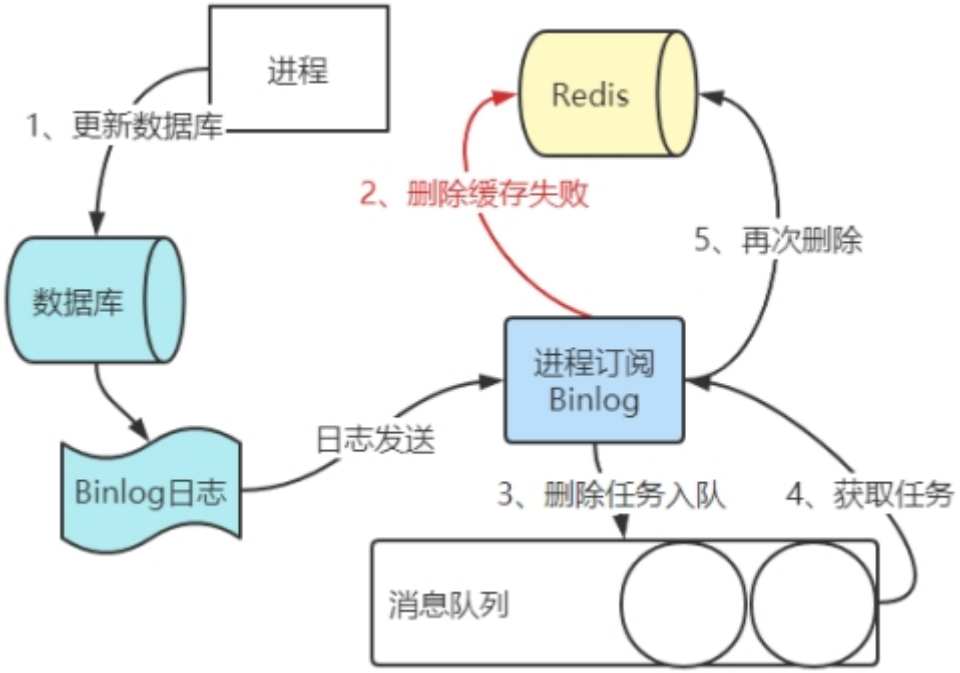
2、在对 Redis 进行删除操作的时候发现报错，删除失败

3、此时将Redis 的 key 作为消息体发送到消息队列中

4、系统接收到消息队列发送的消息后

5、再次对 Redis 进行删除操作

但是这个方案会有一个缺点就是会对业务代码造成大量的侵入，深深的耦合在一起，所以这时会有一个优化的方案，我们知道对 Mysql 数据库更新操作后再 binlog 日志中我们都能够找到相应的操作，那么我们可以订阅 Mysql 数据库的 binlog 日志对缓存进行操作。



说到底就是通过数据库的binlog来异步淘汰key，利用工具(canal)将binlog日志采集发送到MQ中，然后通过ACK机制确认处理删除缓存。

先更新DB，后删除缓存，这种方式，被称为Cache Aside Pattern，属于缓存更新的设计模式之一。

缓存更新的设计模式

更新缓存的的设计模式主要有四种：Cache aside, Read through, Write through, Write behind caching

Cache Aside

这是最常用最常用的pattern了。上面已经讲过了，这是标准模式，包括Facebook的论文《Scaling Memcache at Facebook》也使用了这个策略。

缓存一致性如果追求强一致，要么串行化，要么使用分布式读写锁，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，而一般大厂包括Facebook都选择了使用这个降低概率的做法，因为强一致的实现性能往往比较差，而且比较复杂，还要考虑各种容错问题。

Read/Write Through

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。

Write Behind Caching

Write Behind 又叫 Write Back。Linux文件系统的Page Cache也是同样算法。

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）。

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。

> 有道云笔记链接：https://note.youdao.com/s/Oy1T9mxA