

大型网站高并发的读、写实践

问题分类

不管什么系统，要让各式各样的业务功能与逻辑最终在计算机系统里实现，只能通过两种操作：读和写。本章节我们学习高并发问题的方法论也从这两个方面展开分析。

任何一个大型网站，不可能只有读，或者只有写，肯定是读写混合在一起的。但具体到某种业务场景，其高并发的问題，站在 C 端用户角度来看，往往侧重于读或写，或读写同时有。

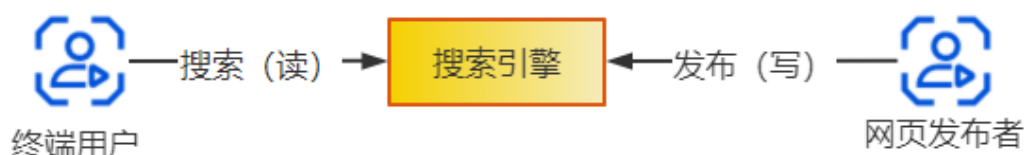
我们来看看大厂有哪些比较具体的业务场景，并了解他们的读写特点是什么，会更容易理解。

侧重于“高并发读”的系统

场景 1：搜索引擎

搜索引擎大家再熟悉不过，用户在百度里输入关键词，百度输出网页列表，然后用户可以一页页地往下翻。在这个过程中，用户只是“浏览”，并没有编辑和修改网页内容。

很明显对于搜索引擎来说，C 端用户是“读”，网页发布者（可能是组织或者个人）是“写”。当然搜索引擎的网页来源大都是网络爬虫主动获得的，不过不影响我们的分析。



为什么说它是一个侧重于“读”的系统呢？让我们来对比读写两端的差异：

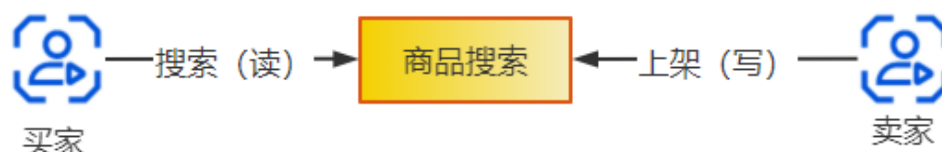
（1）数量级。读的一端，C 端用户，是亿或数十亿数量级；写的一端，网页发布者，可能是“百万或千万”数量级。毕竟读网页的人比发布网页的人要多得多。

（2）响应时间。读的一端通常要求在毫秒级，最差情况为 1~2s 内返回结果；写的一端可能是几分钟或几天。比如发布一篇博客，可能 5 分钟之后才会被搜索引擎检索到；再差一点，可能几个小时后；再差一点，可能永远检索不到，搜索引擎并不保证发布的文章一定被检索到。

（3）频率。读的频率远比写的频率高。

场景2：电商的商品搜索

电商的商品搜索和搜索引擎的网页搜索类似，一个商品类比一篇网页。卖家发布商品，买家搜索商品。

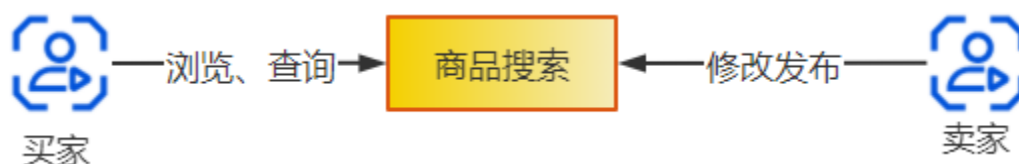


同样，读和写的差异，在其用户规模的数量级、响应时间、频率几个维度，和搜索引擎类似。

场景3:电商系统的商品描述、图片和价格

商品的文字描述、图片和价格有一个显著特点：对于这些信息，C端的买家只会看，不会编辑；B端的卖家会修改这些信息，但其修改频率远低于C端买家的查询流频率。

这个场景里，读和写两端的用户在数量级、响应时间、频率方面，同样有上面类似特征。

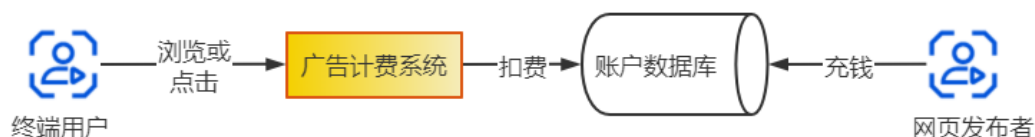


侧重于“高并发写”的系统

以广告扣费系统为例，广告是互联网的三大变现模式之一（另外两个是游戏和电商），对于大家来说已经不陌生。百度的搜索结果中有广告，微博的 Feeds 流中有广告，淘宝的商品列表页中有广告，微信的朋友圈里也有广告。

这些广告通常要么按浏览付费，要么按点击付费（业界叫作 CPC 或 CPM）。具体来说，就是广告主在广告平台开通一个账号，充一笔钱进去，然后投放自己的广告。C端用户看到了这个广告后，可能点击一次扣一块钱(CPC)；或者浏览这个广告，浏览 1000 次扣 10 块钱(CPM)。这里只是打个比方，实际操作当然不是这个价格。

这样的广告计费系统（即扣费系统）就是一个典型的“高并发写”的系统，



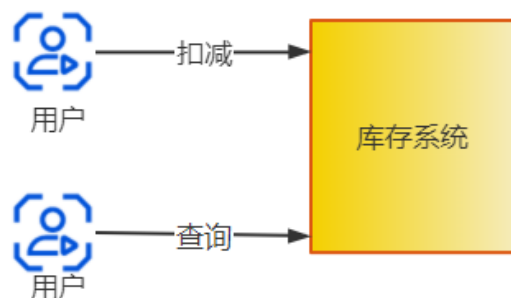
1) C端用户的每一次浏览或点击，都会对广告主的账号余额进行一次扣减。

2) 这种扣减要尽可能实时。如果扣慢了，广告主的账户里明明没有钱了，但广告仍然在线上播放，会造成平台流量的损失。

同时有“高并发读”和“高并发写”的系统

场景 1：电商的库存系统和秒杀系统

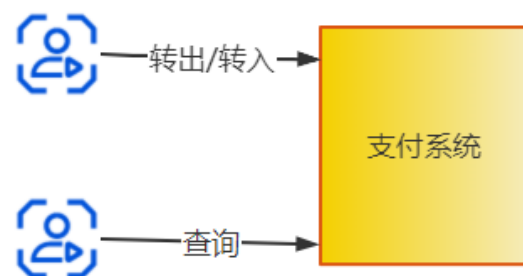
库存系统和秒杀系统的一个典型特征是：C 端用户要对数据同时进行高并发的读和写，这是它不同于商品的文字描述、图片和价格这种系统的重要之处。



一个商品有 100 个，用户 A 买了 1 个，用户 B 买了 1 个……大家在实时地并发扣减，这个信息要近乎实时地更新，才能保证其他用户及时地看到信息。12306 网站的火车票售卖系统也是一个典型例子，当然，它比库存的扣减更为复杂。因为一条线路，可能某个用户买了中间的某一段，剩下的部分还要分成两段继续卖。

场景 2：支付系统和微信红包

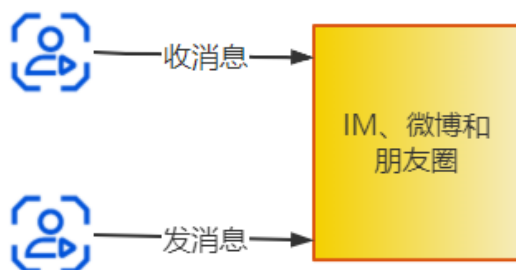
支付系统也是读和写的高并发都发生在 C 端用户的场景，一方面用户要实时地查看自己的账号余额(这个值需要实时并且很准确)，另一方面用户 A 向用户 B 转账的时候，A 账户的扣钱、B 账户的加钱也要尽可能地快。钱一类的信息很敏感，其对数据一致性的要求要比商品信息、网页信息高很多。



从支付扩展到红包系统，业务场景会更复杂，一个用户发红包，多个人抢。一个人的账号发生扣减，多个人的账号加钱，开且在这个过程中还要查看哪些红包已经被抢了，哪些还没有。

场景 3：IM、微博和朋友圈

对于 QQ、微信类的即时通信系统，C 端用户要进行消息的发送和接收；对于微博，C 端用户发微博、查看微博；对于朋友圈，C 端用户发朋友圈、查看朋友圈。



所以无论在读的端，还是写的端，都面临着高并发的压力，用户规模在亿级别，同时要求读和写的处理要非常及时。

通过上面一系列业务场景的说明，会发现针对不同的业务系统，有的可能在“读”的一端面临的高并发压力多一些，有的可能在“写”的一端面临的压力大一些，还有些是同时面临读和写的压力。

之所以要这样区分，是因为外理“高并发读”和“高并发写”的策略很不一样。下面分别展开应对“高并发读”和“高并发写”的不同策略。

高并发读

策略 1：加缓存/读副本

如果流量扛不住了，相信大家首先想到的策略就是“加缓存”。缓存几乎无处不在，它的本质是以空间换时间。下面列举几个缓存的典型用例：

方案 1：本地缓存或集中式缓存

当数据库支持不住的时候，首先想到的就是为其加一层缓存。缓存通常有两种思路：一种是本地缓存，另一种是 Memcached/Redis 类的集中式缓存。

缓存的数据结构通常都是 $\langle k, v \rangle$ 结构， v 是一个普通的对象。再复杂一点，有 $\langle k, list \rangle$ 或 $\langle k, hash \rangle$ 结构。

$\langle k, v \rangle$ 结构和关系型数据库中的单表的一行记录刚好对应，很容易缓存。

缓存的更新有两种：一种是主动更新，当数据库中的数据发生变更时，主动地删除或更新缓存中的数据；另一种是被动更新，当用户的查询请求到来时，如果缓存过期，再更新缓存。

对于缓存,需要考虑几个问题：

(1)缓存的高可用问题。如果缓存宕机，是否会导致所有请求全部写入并压垮数据库呢？

(2)缓存穿透。虽然缓存没有宕机，但是某些 Key 发生了大量查询，并且这些 Key 都不在缓存里，导致短时间内大量请求压垮数据库。

(3)缓存击穿。指一个热点 Key，大并发集中对这一个点进行访问，当这个 Key 在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库。

(4)大量的热 Key 过期。和第二个问题类似，也是因为某些 Key 失效，大量请求在短时间内写入并压垮数据库，也就是缓存雪崩。其实第一个问题也可以视为缓存雪崩。

这些问题和缓存的回源策略有关：一种是不回源，只查询缓存，缓存没有，直接返回给客户端为空，这种方式肯定是主动更新缓存，并且不设置缓存的过期时间，不会有缓存穿透、大量热 Key 过期问题；另一种是回源，缓存没有，要再查询数据库更新缓存，这种需要考虑应对上面的问题。

方案2: MySQL 的 Master/Slave

上述的缓存策略很容易用来缓存各种结构相对简单的<k,v>数据。但对于有的场景，操作复杂的业务数据，如果直接查业务系统的数据库，会影响 C 端用户的高并发访问。

对于这种查询，往往会为 MySQL 加一个或多个 Slave，来分担主库的读压力，是一个简单而又很有效的办法。

方案3: CDN/静态文件加速/动静分离

在网站的展示数据中，并不是所有的内容每次查询都需要重新计算的，可以分为静态内容和动态内容两部分。

(1)静态内容。数据不变，并且对于不同的用户来说，数据基本是一样的，比如图片、HTML、JS、CSS 文件；再比如各种直播系统，内容生成端产生的视频内容，对于消费端来说，看到的都是一样的内容。

(2)动态内容。需要根据用户的信息或其他信息（比如当前时间）实时地生成并返回给用户。

对于静态内容，一个最常用的处理策略就是 CDN。一个静态文件缓存到了全网的各个节点，当第一个用户访问的时候，离用户就近的节点还没有缓存数据，CDN 就去源系统抓取文件缓存到该节点；等第二个用户访问的时候，只需要从这个节点访问即可，而不再需要去源系统取。

注意：对于 Redis、MySQL 的 Slave、CDN，虽然从技术上看完全不一样，但从策略上看都是一种“缓存/加副本”的形式。都是通过对数据进行冗余，达到空间换时间的效果。

策略 2:并发读

无论“读”还是“写”，串行改并行都是一个常用策略。如何把串行改成并行？

方案1: 异步 RPC

现在的 RPC 框架基本都支持了异步 RPC，对于用户的一个请求，如果需要调用 3 个 RPC 接口，则耗时分别是 T1、T2、T3。

如果是同步调用,则所消耗的总时间 $T=T1+T2+T3$ ；如果是异步调用，则所消耗的总时间 $T=\text{Max}(T1, T2, T3)$ 。

当然，这有个前提条件：3 个调用之间没有耦合关系，可以并行。如果必须在拿到第 1 个调用的结果之后，根据结果再去调用第 2、第 3 个接口，就不能做异步调用了，而且异步调用后，如何获得异步调用的结果也需要仔细考虑。

方案 2：冗余请求

2013 年 Google 公司的 Jeaf Dean 在论文《The Tail at Scale》（里面讲述的是 google 内部的一些长尾耗时优化相关的经验）一文中讲过这样一个案例：假设一个用户的请求需要 100 台服务器同时联合处理，每台服务器有 1% 的概率发生调用延迟（假设定义响应时间大于 1s 为延迟），那么对于 C 端用户来说，响应时间大于 1s 的概率是 63%。

这个数字是怎么计算出来的呢？如果用户的请求响应时间小于 1s，意味着 100 台机器的响应时间都小于 1s，这个概念是 100 个 99% 相乘，即 $99\%^{100}$ 。

反过来，只要任意一台机器的响应时间大于 1s，用户的请求就会延迟，这个概率是

$$1 - 99\%^{100} = 63\%$$

这意味着：虽然每一台机器的延迟率只有 1%，但对于 C 端用户来说，延迟率却是 63%。机器数越多，问题越严重。

而越是大规模的分布式系统，服务越多，机器越多，一个用户请求调动的机器也就越多，问题就越严重。

论文中给出了问题的解决方法：冗余请求。客户端同时向多台服务器发送请求，哪个返回得快就用哪个，其他的丢弃，但这会让整个系统的调用量翻倍。

把这个方法调整一下就变成了：客户端首先给服务端发送一个请求，并等待服务端返回的响应；如果客户端在一定的时间内没有收到服务端的响应，则马上给另一台(或多台)服务器发送同样的请求；客户端等待第一个响应到达之后，终止其他请求的处理。上面“一定的时间”定义为：内部服务 95% 请求的响应时间。这种方法在论文中称为“对冲请求”。

论文中提到了 Google 公司的一个测试数据：采用这种方法，可以仅用 2% 的额外请求将系统 99.9% 的请求响应时间从 1800ms 降低到 74ms。

ps：《The Tail at Scale》概述

文章，主要是为分布式系统中一个复杂而重要的问题提供了一整套的解决方案，这个问题就是：

当用户的请求依赖大规模分布式系统来协作完成时，如何保证请求的响应时间？

首先，**为什么**要保证用户请求的响应时间？因为对于某些使用场景来说，只有当响应时间控制在百毫秒以内，用户才能感觉到系统是流畅的，例如 Google 的搜索框会随着用户的输入不断地提示用户可能之后输入的内容，试想如果这个提示需要一秒的时间才能出现？那用户体验是会大打折扣的。

其次，什么是大规模分布式系统？

举例来说，当用户打开亚马逊购物网站的一个商品信息页时，可能需要从底层数百个子服务来获取信息，而这些子服务还有它们依赖的更基础的服务，这意味着一个用户的请求可能需要数千台服务器协同工作才能完成。

在这样的集群规模下，即使有极少数机器请求响应时间较长，也有很大的概率导致用户请求的响应时间变长

比如：当用户请求需要 100 台服务器来协同完成时，假设只有 1% 的调用延迟较大，那用户请求受影响导致响应时间较长的概率为 0.63 ($1 - 0.99^{100}$)

当用户的请求需要 2000 台服务器来协同完成时，假设只有 0.01% 的调用延迟较大，那用户请求受影响导致响应时间较长的概率为 0.18 ($1 - 0.9999^{2000}$)，相当于每 5 个用户就有一个体验不佳

在实际生产环境中，有多种原因导致服务调用的响应时间变长，如：

在一台机器上可能同时部署了多个服务，这些服务可能会对于如 CPU，内存，网络等多种资源进行竞争，导致某个服务的响应时间变长；

某些后台运行的线程（假设每小时运行一次），可能会在运行时占用一些系统资源（如磁盘 IO）导致这台机器上的服务响应时间变长；

在分布式环境下，当多个服务对共享资源（如分布式文件系统等）产生竞争时，也会导致服务的响应时间变长；

其他的原因还包括请求排队、垃圾回收、日志压缩等；

怎么做

首先要保证一个模块对外的每一次响应的时间是快速一致的，文中提到的方法有：

不要在底层设置太长的任务队列，而由上层基于任务的优先级进行动态的下发。比如自己维护一个队列允许服务器跳过那些更早到来的非延迟敏感的批处理操作，优先往下传那些高优先级的交互请求。；

将需要长时间运行的任务拆成多个短时间运行的任务。这个跟上面做法的逻辑一样，就是为了防止一个任务长时间占用系统资源而导致其他任务响应时间变长；

控制好定时任务和后台运行的任务。具体做法有，将后台的大任务拆分成一系列的小任务，在后台任务运行的时候首先确认系统的负载，如果负载太高，则延迟运行后台任务等。

然而由于大规模分布式系统的复杂性，实际上根本无法做到完全消除单个模块的响应时间出现突刺，所以论文中在更大的层面说明如何解决这个问题，也就是一个服务调用另一个服务时，可以通过什么技巧来解决响应时间出现突刺的问题。

分布式系统为了解决数据可靠性的问题，通常将数据复制到多台服务器上，同时为了实现服务的高可用，又会有多个服务器提供同样的功能，防止部分服务器出现故障。所以论文中提出了前面所说的“对冲请求”办法，可以大大降低某个调用出现突刺的情况，因为所有的服务器都出现突刺的概率会大大降低，但是同也会让整个系统的请求数量翻两倍或三倍。

除此之外，论文中还提出了叫 Tied Request “捆绑请求”的解决办法，当一个上游系统向下游系统分发请求时，一个比较合理的做法是将请求分发给任务队列较短，负载较轻的服务器，因为服务抖动一个常见的来源是请求被执行前在服务端的排队延迟。

但是上游如何对下游进行探测呢？因为探测和发送请求之间有一段时间间隔，即在探测之后再发送请求时，各个服务器的负载已经发生了变化；另外如果很多

客户端同时发现某个服务器负载较低，则可能同时将请求发给该服务器，导致该服务器瞬间过热；其次仅通过队列长度来估计处理时间未必准确，即任务队列端不一定真的负载低。

Google 推荐的做法是，上游系统同时发送两个一样的请求给下游多个服务器，当下游服务器处理完成后，通知其他服务器不用再处理该请求。为了防止由于任务队列为空而导致的所有服务器同时处理任务的情况，上游系统需要在发给下游多个系统的两个请求之间引入一个延迟，该延迟时间要足够第一个系统处理完任务并通知其他系统。

策略 3：重写轻读

方案 1：微博 Feeds 流的实现

微博首页或微信朋友圈都存在类似的查询场景：用户关注了 n 个人(或者有 n 个好友)，每个人都在不断地发微博，然后系统需要把这 n 个人的微博按时间排序成一个列表，也就是 Feeds 流并展示给用户。同时，用户也需要查看自己发布的微博列表。

所以对于用户来说，最基本的需求有两个：查看关注的人的微博列表(Feeds 流)和查看自己发布的微博列表。

先考虑最原始的方案，如果这个数据存在数据库里面可以这么设计：

关注关系表 (Following)		
id 自增主键	user_id (关注者)	followings (被关注的人)

微博发布者 (Msg)		
id 自增主键	user_id (发布者)	msg_id (发布的微博ID)

假设这里只存储微博 ID，而微博的内容、发布时间等信息存在另外一个专门的 NoSQL 数据库中。

针对上面的数据模型，假设要查询 user id= 1 发布的微博列表（分页显示），
`select msg_id from msg where user_id= 1 limit offset,count`

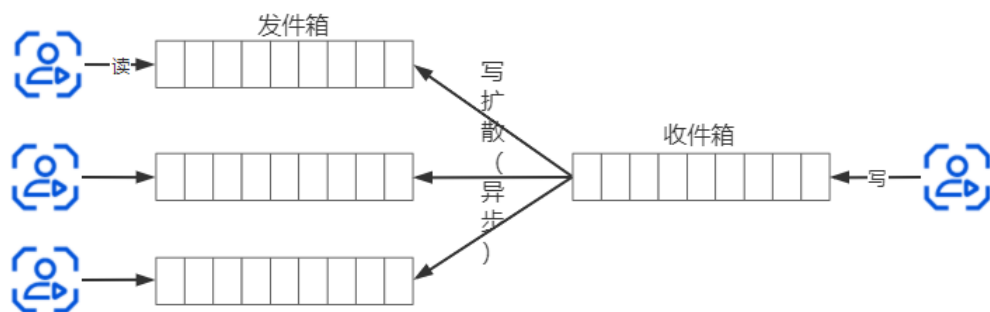
假设要查询 user_id= 1 用户的 Feeds 流，并且按时间排序、分页显示，需要两条 SQL 语句：

`select followings from Following where user_id = 1` //查询 user_id = 1 的用户的关注的用户列表

`select msg_id from msg where user_id in (followings) limit offset,count` //查询关注的所有用户的微博列表

很显然这种模型无法满足高并发的查询请求，那怎么处理呢？

改成重写轻读，不是查询的时候再去聚合，而是提前为每个 user_id 准备一个 Feeds 流，或者叫收件箱。



每个用户都有一个发件箱和收件箱。假设某个用户有 1000 个粉丝，发布 1 条微博后，只写入自己的发件箱就返回成功。然后后台异步地把这条微博推送到 1000 个粉丝的收件箱，也就是“写扩散”。这样，每个用户读取 Feeds 流的时候不需要再实时地聚合了，直接读取各自的收件箱就可以。这也就是“重写轻读”，把计算逻辑从“读”的一端移到了“写”的一端。

这里的关键问题是收件箱是如何实现的？因为从理论上来说，这是个无限长的列表。很显然，这个列表必须在内存里。假设用 Redis 的 `<key,list>` 来实现，key 是 `user_id`，list 是 `msg_id` 的列表。但这个 list 不能无限制地增长，假设设置一个上限为 2000。

那么用户在屏幕上一一直往下翻，当翻到 2000 个以外时，怎么分页查询呢？

最简单的方法：限制数量！最多保存 2000 条，2000 条以外的丢弃。因为按常识，手机屏幕一屏通常显示 4~6，2000 条意味着用户可以翻 500 次，一般的用户根本翻不到这么多，即使是 PC 也要翻百次。

而这实际上就是 Twitter 的做法，据公开的资料显示，Twitter 实际限制为 800 条。

但对于用户发布的微博，不希望过了一段时间之后，系统把历史数据删掉了，而是希望系统可以全量地保存数据。但 Redis 只能保存最近的 2000 个，2000 个以前的数据肯定要持久化存储并且支持分页查询。

这里自然还是考虑用 MySQL 来存储表中的数据，很显然这个数据会一直增长，不可能放在一个数据库里面。那就涉及按什么维度进行数据库的分片。

一种是按 `user id` 进行分片，一种是按时间范围进行分片（比如每个月存储一张表）。

如果只按 `user id` 分片，显然不能完全满足需求。因为数据会随着时间一直增长，并且增长得还很快，用户在频繁地发布微博。

如果只按时间范围分片，会冷热不均。假设每个月存储一张表，则绝大部分读和写的请求都发生在当前月份里，历史月份的读请求很少，写请求则没有。

所以需要同时按 `user_id` 和时间范围进行分片。但分完之后，如何快速地查看某个 `user id` 从某个 `offset` 开始的微博呢？比如一页有 100 个，现在要显示第 50 页，也就是 `offset = 5000` 的位置开始之后的微博。如何快速地定位到 5000 所属的库呢？

这就需要有一个二级索引：另外要有一张表，记录 `<user_id,月份,count>`。也就是每个 `user_id` 在每个月份发表的微博总数。基于这个索引表才能快速地定位到 `offset = 5000` 的微博发生在哪个月份，也就是哪个数据库的分片。

解决了读的高并发问题，但又带来一个新问题：假设一个用户的粉丝很多，给每个粉丝的收件箱都复制一份，计算量和延迟都很大。比如某个明星的粉丝有 8000 万，如果复制 8000 万份，对系统来说是一个沉重负担，也没有办法保证微博及时地传播给所有粉丝。

这就又回到了最初的思路，也就是读的时候实时聚合，或者叫作“拉”。具体怎么做呢？

在写的一端，对于粉丝数量少的用户（假设定个阈值为 5000，小于 5000 的用户），发布一条微博之后推送给 5000 个粉丝；

对于粉丝数多的用户，只推送给在线的粉丝们（系统要维护一个全局的、在线的用户列表）。

对于读的一端，一个用户的关注的人当中，有的人是推给他的（粉丝数小于 5000），有的人是需要他去拉的（粉丝数大于 5000），需要把两者聚合起来，再按时间排序，然后分页显示，这就是“推拉结合”。

方案2：多表的关联查询：宽表与搜索引擎

后端经常需要对业务数据做多表关联查询，可以通过加 Slave 解决，但这种方法只适合没有分库的场景。

如果数据库已经分了库，那么需要从多个库查询数据来聚合，无法使用数据的原生 Join 功能，则只能在程序中分别从两个库读取数据，再做聚合。

但存在一个问题：如果需要把聚合出来的数据按某个维度排序并分页显示，而且这个维度是一个临时计算出来的维度，而不是数据库本来就有的维度。

由于无法使用数据库的排序和分页功能，也无法在内存中通过实时计算来实现排序、分页（数据量太大），这时如何处理呢？

还是采用类似微博的重写轻读的思路：提前把关联数据计算好，存在一个地方，读的时候直接去读聚合好的数据，而不是读取的时候再去做 Join。

具体实现来说，可以另外准备一张宽表：把要关联的表的数据算好后保存在宽表里。依据实际情况，可以定时算，也可能任何一张原始表发生变化之后就触发一次宽表数据的计算。

也可以用 ES 类的搜索引擎来实现：把多张表的 Join 结果做成一个个的文档，放在搜索引擎里面，也可以灵活地实现排序和分页查询功能。

总结：读写分离（CQRS 架构）

无论加缓存、动静分离，还是重写轻读，其实本质上都是读写分离，这也就是微服务架构里经常提到的 CQRS (Command Query Responsibility Separation)。

读写分离架构的典型模型有几个典型特征：

(1) 分别为读和写设计不同的数据结构。在 C 端，当同时面临读和写的高并发压力时，把系统分成读和写两个视角来设计，各自设计适合高并发读和写的数据结构或数据模型。

某种程度来说，缓存其实是读写分离的一个简化，或者说是特例，写业务 DB 和读缓存用了基本一样的数据结构。

(2) 写的这一端，通常也就是在线的业务 DB，通过分库分表抵抗写的压力。读的这一端为了抵抗高并发压力，针对业务场景，可能是<K,V>缓存，也可能是提前做好 Join 的宽表，又或者是 ES 搜索引擎。

(3) 读和写的串联。定时任务定期把业务数据库中的数据转换成适合高并发读的数据结构；或者是写的一端把数据的变更发送到消息中间件，然后读的一端消费消息；或者直接监听业务数据库中的 Binlog，监听数据库的变化来更新读的一端的数据。

(4) 读比写有延迟。因为左边写的数据是在实时变化的，右边读的数据肯定会有延迟，读和写之间是最终一致性，而不是强一致性，但这并不影响业务的正常运行。

拿库存系统举例，假设用户读到某个商品的库存是 9 件，实际可能是 8 件(某个用户刚买走了 1 件)，也可能是 10 件(某个用户刚刚取消了一个订单)，但等用户下单的一刻，会去实时地扣减数据库里面的库存，也就是左边的写是“实时、完全准确”的，即使右边的读有一定时间延迟也没有影响。

同样，拿微博系统举例，一个用户发了微博后，并不要求其粉丝立即能看到。延迟几秒钟才看到微博也可以接受，因为粉丝并不会感知到自己看到的微博是几秒钟之前的。

但是对于用户自己的数据，自己写自己读(比如账号里面的钱、用户下的订单)，在用户体验上肯定要保证自己修改的数据马上能看到。

这种在实现上读和写可能是完全同步的(对一致性要求非常高，比如涉及钱的场景)；也可能是异步的，但要控制读比写的延迟非常小，用户感知不到。

虽然读的数据可以比写的数据有延迟(最终一致性)，但还是要保证数据不能丢失、不能乱序，这就要求读和写之间的数据传输通道要非常可靠。

高并发写

在解决了高并发读的问题后，下面讨论高并发写的各种应对策略。

策略 1：数据分片

数据分片也就是对要处理的数据或请求分成多份并行处理。

比如，数据库的分库分表。数据库为了应对高并发读的压力，可以加缓存、Slave。应对高并发写的压力，就需要分库分表了。分表后，还是在一个数据库、一台机器上，但可以更充分地利用 CPU、内存等资源；分库后，可以利用多台机器的资源。Redis Cluster 集群也是一样的道理。

ES 的分布式索引，在搜索引擎里有一个基本策略是分布式索引。比如有 10 亿个网页或商品，如果建在一个倒排索引里面，则索引很大，也不能并发地查询。

可以把这 10 亿个网页或商品分成 n 份，建成 n 个小的索引。一个查询请求来了以后，并行地在 n 个索引上查询，再把查询结果进行合并。

。

策略 2：异步化

“异步”一词在计算机的世界里几乎无处不在，

对于“异步”而言，站在客户端的角度来讲，是请求服务器做一个事情，客户端不等结果返回，就去做其他的事情，回头再去轮询，或者让服务器回调通知。

站在服务器角度来看，是接收到一个客户的请求之后不立即处理，也不立马返回结果，而是在“后台慢慢地处理”，稍后返回结果。因为客户端不等上一个请求返回结果就可以发送一个请求，可以源源不断地发送请求，从而就形成了异步化。

HTTP 1.1 的异步化、HTTP/2 的二进制分帧都是“异步”的例子，客户端发送了一个 HTTP 请求后不等结果返回，立即发送第 2 个、第 3 个；数据库的内存事务提交与 Write-ahead Log 也是“异步”的例子。

案例 1：短信验证码注册或登录

通常在注册或登录 App 或小程序时，采用的方式为短信验证码。短信的发送通常需要依赖第三方的短信发送平台。客户端请求发送验证码，应用服务器收到请求后调用第三方的短信平台。

公网的 HTTP 调用可能需要 1~2 秒，如果是同步调用，则应用服务器会被阻塞。假设应用服务器是 Tomcat，一台机器最多可以同时处理几百个请求，如果同时来几百个请求，Tomcat 就会被卡死了。

改成异步调用就可以避免这个问题，应用服务器收到客户端的请求后，放入消息队列，立即返回。然后有个后台消费者，从消息队列读取消息，去调用第三方短信平台发送验证码。

应用服务器和消息队列之间是内网通信，不会被阻塞，即使客户端并发量很大，最多是消息堆积在消息队列里面。

对用户来说，并不会感知到同步或者异步的差别，反正都是按了“获取验证码”的按钮后等待接收短信。可能过于 60s 之后没有收到短信，用户又会再次按按钮。

案例 2：电商的订单系统

有电商购物经验的人可能会发现，假设我们在淘宝或者天猫上买了 3 个商品，且来自 3 个卖家，虽然只下了一个订单、付了一次款，但在“我的订单”里去查看，却发现变成了 3 个订单，3 个卖家分别发货，对应 3 个包裹。

从 1 个变为 3 个的过程，是电商系统的一个典型处理环节，叫作“拆单”。而这个环节就是通过异步实现的。

对于客户端来说，首先是创建了一个订单，写入订单系统的数据库，此时未支付。

然后去支付，支付完成后，服务器会立即返回成功，而不是等 1 个拆成 3 个之后再返回成功。

总之，凡是不阻碍主流程的业务逻辑，都可以异步化，放到后台去做。

案例3：广告计费系统

前面我们提到过广告计费系统是一个侧重于“高并发写”的系统，用户每点击 1 次，就需要对广告主的账号扣 1 次钱。

如果用户每点击 1 次或浏览 1 次，都同步地调用账户数据库进行扣钱，账户数据库肯定支撑不住。

同时，对于用户的点击来说，在扣费之前其实还有一系列的业务逻辑要处理，比如判断是否为机器人在刷单，这种点击要排除在外。

所以，实际上用户的点击请求或浏览请求首先会以日志的形式进行落盘。一般是支持持久化的消息队列。落盘之后，立即给客户端返回数据。后续的所有处理，当然也包括扣费，全部是异步化的，而且会使用流式计算模型完成后续的所有工作。

案例4：写内存 + Write-Ahead 日志

MySQL 中为了提高磁盘 IO 的写性能，使用了 Write-Ahead 日志，也就是 Redo Log。这种思路不仅在数据库和 KV 存储领域使用，在上层业务领域中同样可以使用。比如高并发地扣减 MySQL 中的账户余额，或者电商系统中扣库存，如果直接在数据库中扣，数据库会扛不住，则可以在 Redis 中扣，同时落一条日志（日志可以在一个高可靠的消息中间件或数据库中插入一条条的日志）。当 Redis 宕机，把所有的日志重放完毕，再用数据库中的数据初始化 Redis 中的数据。

策略 3：批量写

案例1：广告计费系统的合并扣费

在异步化里提到广告计费系统使用了异步化的策略。在异步化的基础上，可以实现合并扣费。

假设有 10 个用户，对于同 1 个广告，每个用户都点击了 1 次，也就意味着同 1 个广告主的账号要扣 10 次钱，每次扣 1 块（假设点击 1 次扣 1 块钱）。如果改成合并扣费，就是 1 次扣 10 块钱。

扣费模块一次性地从持久化消息队列中取多条消息，对多条消息按广告主的账号 ID 进行分组，同一个组内的消息的扣费金额累加合并，然后从数据库里扣除。

2. 案例2：MySQL 的小事务合并机制

MySQL 的事务的实现上，存在着小事务合并机制。

比如扣库存，对同一个 SKU，本来是扣 10 次、每次扣 1 个，也就是 10 个事务；在 MySQL 内核里面合并成 1 次扣 10 个，也就是 10 个事务变成了 1 个事务。

我们同样可以借鉴这一点，比如我们的 Canal 同步代码中，首页广告内容的更新就采用同样的机制


```

/*一个表在一次周期内可能会被修改多次，而对Redis缓存的处理只需要处理一次即可*/
Set<String> factKeys = new HashSet<>();
for(CanalEntry.Entry entry : message.getEntries()){
    if (entry.getEntryType() == CanalEntry.EntryType.TRANSACTIONBEGIN
        || entry.getEntryType() == CanalEntry.EntryType.TRANSACTIONEND) {
        continue;
    }
    CanalEntry.RowChange rowChange = CanalEntry.RowChange.parseFrom(entry.getStoreValue());
    String tableName = entry.getHeader().getTableName();
    if(log.isDebugEnabled()){
        CanalEntry.EventType eventType = rowChange.getEventType();
        log.debug("数据变更详情：来自binlog[{}.{}], 数据源{}.{}, 变更类型{}",
            entry.getHeader().getLogfileName(),entry.getHeader().getLogfileOffset(),
            entry.getHeader().getSchemaName(),tableName,eventType);
    }
    factKeys.add(tableMapKey.get(tableName));
}
for(String key : factKeys){
    /*此处未做删除缓存失败的补偿操作，可以自行加入*/
    if(StringUtils.isEmpty(key)) redisOpsExtUtil.delete(key);
}

```

同样，在多机房的数据库多活（跨数据中心的数据库复制）场景中，事务合并也是加速数据库复制的一个重要策略。

RockDB 详解

在秒杀库存扣减中，我们使用了写内存 + Write-Ahead 日志的方式。，但是如何写 Write-Ahead 日志呢？

我们知道，在事务的实现机制上，MySQL 采用的就是 WAL（Write-ahead logging，预写式日志）机制来实现的，所有的修改都先被写入到日志中，然后再被应用到系统中，通常包含 redo 和 undo 两部分信息。就是我们熟知的 Redo 日志，文件名一般是 ib_logfile0,1,2.....，MySQL 使用了非常复杂的机制来实现 WAL，单独的日志格式，种类繁多的日志类型，分组写入等等。

难道我们也需要自己来实现一个类似于 Redo 日志的功能吗？并不需要，我们直接使用了 RocksDB。为什么选择它？

RocksDB 是 Facebook 开源的一个高性能、持久化的 KV 存储引擎，最初是 Facebook 的数据库工程师团队基于 Google LevelDB 开发。一般来说我们很少见到过哪个项目直接使用 RocksDB 来保存数据，即使未来大概也不会像 Redis 那样被业务系统直接使用。

但是越来越多的新生代数据库都选择 RocksDB 作为它们的存储引擎。比如：

CockroachDB（中文名蟑螂，一个开源、可伸缩、跨地域复制且兼容事务的 ACID 特性的分布式数据库，思路源自 Google 的全球性分布式数据库 Spanner。其理念是将数据分布在多数据中心的多台服务器上）

YugabyteDB，Tidb 作为 CockroachDB 的竞争产品，底层也是 RocksDB

开源项目 MyRocks 使用 RocksDB 给 MySQL 做存储引擎，目的是取代现有的 InnoDB 存储引擎。MySQL 的亲兄弟 MariaDB 也已经接纳了 MyRocks，作为它的存储引擎。

实时计算引擎 Flink，其 State 就是一个 KV 存储,它使用的也是 RocksDB。

此外包括 MongoDB、Cassandra、Hbase 等在内的很多数据库,都在开发基于 RocksDB 的存储引擎。

这其中的主要原因就是 RocksDB 高性能、支持事务。按照 RocksDB 主页 <https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks> 的说法:

随机读最高可以达到 19W/S, 平均水平 13W/S, 覆盖操作可以达到 9W/S, 多读单写的情况下在 10W/S 左右。

根据 InfluxDB 的测试:

<https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/>

批量写入 5000 万数据,RocksDB 只花了 1m26.9s,在几个对照引擎中(LevelDB 与 RocksDB 与 HyperLevelDB 与 LMDB), RocksDB 读取和删除方面表现最好, InfluxDB 也认为根据测试用例, RocksDB 作为存储引擎是个非常好的选择

Overall it looks like RocksDB might be the best choice for our use case. However, there are lies, damn lies, and benchmarks. Things can change drastically based on hardware configuration and settings on the storage engines.

回到我们的秒杀,现在我们需要实现写内存 + Write-Ahead 日志来应对写热点,自然 Write-Ahead 日志部分性能一定要强,这就是为什么我们选择了 RocksDB 的原因。

RocksDB VS Redis

说到 KV 存储,我们最熟悉的的就是 Redis 了。RocksDB 和 Redis 都是 KV 存储。其实 Redis 和 RocksDB 之间并没有可比性,一个是缓存,一个是数据库存储引擎。

Redis 是一个内存数据库,它的性能非常好,主要原因之一是它的数据全都保存在内存中。按照 Redis 官方网站提供的测试数据来看,它的随机读写性能大约为 50 万次/秒,当然我们普遍的认可的说是 10W/S。从我们上面的文字可以看到 RocksDB 相应的随机读写性能大约为 20 万次/秒,虽然其性能还不如 Redis,但是已经可以算是在同一个量级水平了,毕竟一个是内存操作,另一个是磁盘 IO 操作。

但是我们也知道,Redis 只是一个内存数据库,并不是一个可靠的存储引擎。在 Redis 中,数据写到内存中就算成功了,其并不能保证将数据安全地保存到磁盘上。而 RocksDB 则是一个持久化的 KV 存储引擎,它需要保证每条数据都已安全地写到磁盘上。这种情况下,RocksDB 的优势就很明显,磁盘的读写性能与内存的读写性能本就相差了一两个数量级,读写磁盘的 RocksDB 能达到与读写内存的 Redis 相近的性能这就是 RocksDB 的价值所在了。

一个存储系统的读写性能主要取决于它的存储结构,也就是数据是如何组织的。RocksDB 采用了一个非常复杂的数据存储结构,并且这个存储结构采用了内存和磁盘混合存储的方式,它使用磁盘来保证数据的可靠存储的,并且会利用速度更快的内存来提升读写性能。

RocksDB 为什么能实现这么高的写入性能呢?大多数存储系统为了能够实现快速查找都会采用树或哈希表之类的存储结构。数据在写入的时候必须写到特定的位置上。比如,我们在向 B+树中写入一条数据时,必须按照 B+树的排序方式,写到某个固定的节点下面。哈希表也与之类似,必须要写到特定的哈希槽中。

这样的数据结构会导致在写入数据的时候,不得不先在磁盘的这里写一部分,再到那里写一部分这样跳来跳去地写,即我们所说的“随机写”。MySQL 为了减少随机读写,下了不少功夫。而 RocksDB 的数据结构,可以保证写入磁盘的绝大多数操作都是顺序写入的。

Kafka 所采用的也是顺序读写的方式,所以读写性能也非常好。凡事有利也有弊,这种数据基本上是无法查询的因为数据没有结构,只能采用遍历的方式。

RocksDB 究竟如何在保证数据顺序写入的前提下,还能兼顾很好的查询性能呢?? 其实使用了数据结构 LSM-Tree。

LSM-Tree 如何兼顾读写性能

LSM-Tree 的全称是 The Log-Structured Merge-Tree, 是一种非常复杂的复合数据结构。它包含了 WAL (Write Ahead Log)、跳表(SkipList)和一个分层的有序表 (Sorted String Table, SSTable), LSM-tree 专门为 key-value 存储系统设计的,以牺牲部分读取性能为代价提高写入性能,通常适合于写多读少的场景。

我们借用下论文 “An efficient design and implementation of LSM-tree based key-value store on open-channel SSD” (pdf 链接 https://cadlab.cs.ucla.edu/beta/cadlab/sites/default/files/publications/eurosys2014_final44.pdf, 本论文主要讲述在 SSD 的高并行性下,如何扩展 LevelDB 以显式利用 SSD 的多个通道,优化并发 I/O 请求的调度和调度策略,将常规 SSD 上运行 LevelDB 的吞吐量再提高 4 倍以上) 的图

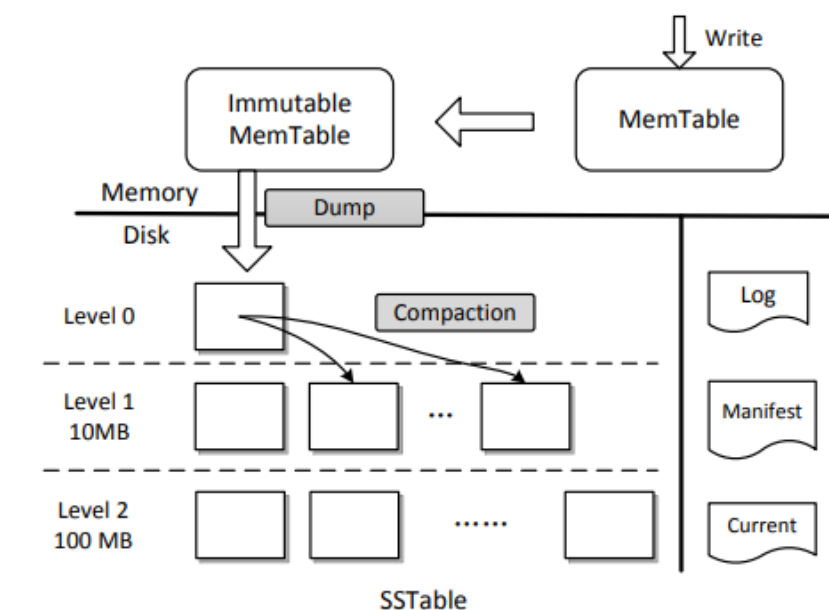


Figure 1. Illustration of the LevelDB architecture.

当然 LSM-Tree 的结构比上面这个图其实复杂。我们只需要关心核心原理即可。

图中有一个横向的实线,这是内存和磁盘的分界线,实线上面的部分是内存,下面的部分是磁盘。

数据的写入过程。当 LSM-Tree 收到一个写请求时，比如 “PUT foo bar”，即把 Key foo 的值设置为 bar，这条操作命令会被写入磁盘的 WAL 日志中，图中右侧的 Log，这是一个顺序写磁盘的操作，性能很好。这个日志的唯一作用就是从故障中恢复系统数据，一旦系统宕机，就可以根据日志把内存中还没有来得及写入磁盘的数据恢复出来。

写完日志之后，数据可靠性的问题就解决了。然后数据会被写入内存的 MemTable 中，这个 MemTable 就是一个按照 Key 组织的跳表(SkipList)，跳表的查找性能与平衡树类似，但实现起来更简单一些。写 MemTable 是一项内存操作，速度也非常快。

数据写入 MemTable 之后就可以返回写入成功的信息了。不过 LSM-Tree 在处理写入数据的过程中会直接将 Key 写入 MemTable，而不会预先查看 MemTable 中是否已经存在该 Key。

很明显 MemTable 不能无限制地写入内存，一是内存的容量毕竟有限，另外，MemTable 太大会导致读写性能下降。所以 MemTable 有一个固定的上限大小，一般是 32MB。

MemTable 写满之后，就会转换成 Immutable(不可变的)MemTable，然后再创建一个空的 MemTable 继续写。这个 Immutable MemTable 是只读的 MemTable，它与 MemTable 的数据结构完全一样唯一的区别就是不允许再写入了。

Immutable MemTable 也不能在内存中无限地占地方，而是会有一个后台线程，不停地把 Immutable MemTable 复制到磁盘文件中，然后释放内存空间。每个 Immutable MemTable 对应于一个磁盘文件，MemTable 的数据结构跳表本身就是一个有序表，写入文件的数据结构也是按照 Key 来排序的，这些文件就是 SSTable。由于把 MemTable 写入 SSTable 的这个写操作，是把整块内存写入整个文件中，因此该操作同样也是一个顺序写操作。

虽然数据已经保存到磁盘上了，而且这些 SSTable 文件中的 Key 是有序的，但是文件之间却是完全无序的，所以还是无法查找。SSTable 采用了一个分层合并机制来解决这个问题。SSTable 被分为很多层，每一层的容量都有一个固定的上限。

一般来说，下一层的容量是上一层的 10 倍。当某一层写满时，就会触发后台线程往下一层合并，数据合并到下一层之后，本层的 SSTable 文件就可以删除了。合并的过程也是排序的过程，除了 Level 0 以外，每层中的文件都是有序的，文件内的 KV 也是有序的，这样就比较便于查找了。

LSM-Tree 查找的过程也是分层查找，先在内存中的 MemTable 和 Immutable MemTable 中查找，然后再按照顺序依次在磁盘的每层 SSTable 文件中查找，一旦找到了就直接返回。看起来这样的查找方式很低效的，可能需要多次查找内存和多个文件才能找到一个 Key，但实际上这样一个分层的结构，它会天然形成一个非常有利于查找的状况，即越是经常被读写到的热数据，它在这个分层结构中就越靠上，对这样的 Key 查找就越快。

比如最经常读写的 Key 很大概率会在内存中，这样不用读写磁盘就能完成查找。即使在内存中查不到真正需要穿透很多层 SSTable，一直查到最底层的请求，实际也还是很少的。另外在工程实现上，还会做很多优化。

比如 RocksDB 里，为存储的数据逻辑分族，独特的 **filter** 对读优化，使用多个 **memtable** 并在 **immutable memtable** 提前进行数据合并的优化，在内存中缓存 **SSTable** 文件的 **Key** 用布隆过滤器避免无谓的查找等，以加速查找过程；不同的合并算法，**SSTFile-Indexing** 机制，针对 **sstfile** 有自己的 **block cache** 和 **table cache**。这样综合优化下来最终的性能，尤其是查找会相对较好。

本文分享地址：

<http://note.youdao.com/noteshare?id=908f68e12dd09be0824125c32851c863&sub=9B16C876A41B4045B272099E50E1EC5E>