链滴

我的高并发之道

作者：michael

原文链接：https://ld246.com/article/1537792889074

来源网站：链滴

<h4 id="什么是高并发-">什么是高并发？</h4>
　　“什么是高并发？”这是一个经常出现在面试过程中的问题，每个人理解不一样，有一个理大家应该都认同，那就是“短时间内处理大量请求”，到底时间多短算短？请求量多大算大？这个得据业务自身而定，通常会有一些压测指标来量化它们，例如 TPS、QPS、响应时间等。那么下面就来聊我对高并发的理解。
<h3 id="时空理论">时空理论</h3>
　　一个系统的高并发能力，我觉得与两个主要因素有关，那就是这个系统对“时间”和“空间的处理能力，“时间”指的是系统的处理时间或者响应时间，“空间”指的是这个系统的并行处理能。这两个指标决定了一个系统的高并发处理能力。总体上讲，并发能力的公式大概就是“并发能力 ≈ 间 ÷ 时间”，跟空间成正比，跟时间成反比，空间越大并发能力越强，时间越小并发能力越强。影响两个因素的因子有很多，涉及到的技术层面也很广，接下来慢慢聊。
<h4 id="时间">时间</h4>
　　时间指系统响应时间，如果你的系统是实时系统，那么这个因素对于你的系统来说非常重要是需要优先考虑的因素，也是影响系统并发能力的主要因素，对系统优化而言，这个是最容易去优化。跟响应时间有关的影响因子非常多，我们可以从底层的服务器运行的环境着手看这个问题，从服务硬件资源的角度讲，与系统性能紧紧相关的有：CPU、内存、网络、磁盘。</p

<h5 id="磁盘">磁盘</h5>
　　首先说说磁盘，磁盘 IO 是最昂贵的资源，对系统响应时间影响最大网络次之，内存和 CPU 相对较好。所以要提高系统的响应时间，我们应该避免直接使用磁盘这个系资源，通过资源转换的方式，将磁盘资源转换成更廉价，性能更好的资源，例如内存。像一些热点数，放在集中缓存里面，对系统的“时间”因素影响特别大，这就是用内存换磁盘的一种方式，用更廉效率更高的资源去优化系统瓶颈。磁盘往往是一个系统最主要的性能瓶颈，这里说的磁盘不是说磁盘存储能力，而是磁盘的 IO 能力，磁盘 IO 要优化的话，代价很高，做 raid、SAS 升级成 SSD、磁盘列等等，要把磁盘 IO 提升一个等级的话，需要的资金也不少。所以如果遇到磁盘 IO 为瓶颈的时候首先可以考虑使用其他系统资源来替代磁盘 IO，从而达到优化的目的，如果实在不行，只能考虑烧了。
<h5 id="网络">网络</h5>
　　其次的系统资源是网络，网络指的是系统之间交互时候的网络资源，与这个资源有关的例如络带宽、网卡数、网络收发包数等，这个资源也是很昂贵的，一个公司的网络环境始终是有个上限的要优化网络这块的处理能力，方式也很多，节约带宽，可以考虑数据压缩，目很多数据压缩方式效率很高，压缩比很大，这个是通过消耗 CPU 资源去换网络资源</strog>，如果一个系统的 CPU 不吃紧，可以考虑这种方式。如果说节约网络请求次数，可以考虑<stron>批处理的方式来进行网络请求发送，例如：数据库请求的 batch，一次处理一批数据，少网络 IO 次数；redis 通过管道的方式，一次处理多条数据；数据聚合，将原本分散而又关联的数据聚合到一起存储，减少请求次数。另外一种节约网络资源的做法就是，就是用内存资源去网络资源，在应用实体节点内，启用本地缓存的方式（例如 EHCache），来存储一些不变化的数据，请求的时候先到本地缓存找，再通过网络到其他服务器上找，这种方式可以带来很大的能提升，当然要实现这种方式，得考虑很多问题，例如每个实例内部数据一致性的问题，数据更新问，系统会变得更复杂。
<h5 id="CPU">CPU</h5>
　　CPU 资源也是非常昂贵的，如果是计算密集型的系统，那么服务器的 CPU 计算能力直接关到这个系统的处理能力，要优化 CPU 的处理能力，可以从多个角度入手，优化的方式也很多，一般们可以考虑从应用本身来进行优化，这是一种至顶向下的优化方式。从应用本身的角度来讲，优化 CP 的处理能力，那么就是考虑如何使用 CPU 资源来完成我们的任务，也就是说如何设计系统内部线程执行方式、锁的使用等。应用程序内部的线程结构设计，目前有很多种比较流行的，例如基于线程、件驱动、SEDA 等，我本身比较偏向第三种 SEDA（不懂的自行百度）。我觉得 SEDA 更适合将系统块化，让系统的复用性、扩展性更强，让系统处理更加灵活（我写过一个开源项目，https://blog.junxworks.cn/articles/2018/08/22/1534921023342.html，其中 junx-event 就是用来辅助开发 SEDA 的包）。br>
　　当然应用内部的线程结构只是其一，还有就是锁的使用，这个锁指的是应用进程内的锁，这个也普遍比较关心的问题。锁不能乱用、不能滥用，执行线程被 block 住过后，会进行线程切换，都知道

PU 计算是基于时间片轮转的，在线程切换会浪费 CPU 时钟，浪费 CPU 资源，如果系统内部经常进线程切换，那并发能力会大打折扣。不管其他人怎么说，我个人倡导，能不用锁就不用锁。首先，我要明白锁的作用，为什么要使用锁？这个得从底层的技术说起，我目前所用的开发语言是 java，所以说的都是基于 java 语言的。java 语言的内存模型中，有堆和栈两个区域（JVM 运行时数据区还有其区域，例如方法区、本地方法区、程序计数区），堆是常说的主内存，这个区域是线程共享的，栈呢是线程间隔离的，也就是线程独享的。我们有一个对象的属性，是存放在堆中的，线程要使用的时候需要进行数据交换，把数据从堆中读取到栈中，计算完成后，再进行赋值，回写到堆中，整个过程是ead-Load,Use-Assign,Store-Write，具体的过程可以参考 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fblog.csdn.net%2Fu011080472%2Farticle%2Fdetails%2F51337422" targe="_blank" rel="nofollow ugc">https://blog.csdn.net/u011080472/article/details/51337422</>。整个过程，栈上计算是独立的，所以当多个线程同时对一个变量进行赋值的时候，会导致变量值覆盖。这时候，就需要对这个处理过程加锁，锁的作用就是控制资源竞争，加锁过后，多个线程对这资源的操作就只能串行处理，等第一个线程执行完后，才能执行第二个线程。明白锁的作用后，就可考虑从程序设计的角度，来合理的规避这个问题，例如通过引入队列的方式将并行处理的业务串行化让单线程去处理。这里引入一个题外话，Disruptr 在这方面做得非常好，整个高并发架构是无锁化的，缓存行填充、伪共享、内存屏障用的恰到处，有兴趣的可以去观摩一下。
　　如果说在某些场景下，必须要使用锁，那么使用的时候一定要谨慎，我大概总结了一下：
　　1、最好不要在 method 上面加锁，如果当前整个 method 内部都是需要同步的，那你能保证以这个 method 内部也都是需要同步的吗？
　　2、synchronized 和 lock 的选择，synchronized 关键字是由 JVM 来控制内部执行的，目前也行了大量优化，个人感觉效率比 lock 要高。Lock 是 Concurrent 包里面提供的，由 JDK 提供的锁，个使用非常灵活，适合复杂的业务场景，但是这个 lock 一定要在 try-finally 中关闭，防止锁死。所一些简单的业务场景，可以使用 synchronized 关键字，复杂的场景可以考虑使用 lock。
　　3、CAS 的使用，CAS 效率比锁要高很多，CAS 全称是 compare and swap，什么是 CAS 自行度。
　　4、优化锁的使用，如果 Concurrent 包中提供了一些支持高并发的容器，例如 ConcurrentHasMap，适合多线程并发读写的场景，能用还是尽量用（如果基本都是读，很少有改动，那么可以考虑OW+HashMap 的方式）。
　　5、可以考虑使用 COW 的方式来减少锁的使用，COW 即 copy on write，这个适合读多写少的景，具体实现自行百度。google 的 guava 包中也提供了类似的容器，例如 Lists.newCopyOnWriteArrayList，就提供了一个 COW 的 list 容器。另外 Concurrent 包中提供的读写锁，在读多写少的场景效率很高。
当然还有其他的一些优化，例如 volatile 关键字的使用（内存屏障这个比较难理解，不太适合一般开人员，如果不懂，就不要用）、代码中的属性定义排序问题，相同使用的属性应当写在一起（跟 CPU 缓存加载机制有关，可以参考 disruptor 的缓存行填充）等等，优化机制非常多，推荐一本书《Java 能优化权威指南》。
　　上面从线程并发结构到锁的使用，简单的介绍了一下应用程序内 CPU 资源的优化，CPU 这其实也可以优化底层，不过一般公司都没有这么去做，我们现在用的服务器，CPU 一个核一般适合 1- 个线程的并行执行，所以我们在设置计算型模块的线程数的时候，一般为 cpu 的核数 ×2，但是甲骨提供了 SPARC T 系列的 CPU，支持一个核 4-8 个线程并行执行，大大增强了 CPU 的并行处理能力适合高并发的应用。不过 SPARC T 系列的 CPU 核支持的线程再多，性价比也不如基于显卡 GPU 加来得高，一个好一点的显卡，都是成千上万个核，并发能力是 CPU 没法比的，个人感觉基于 GPU 的算是一个趋势，目前有很多数据库已经使用 GPU 进行计算加速了，例如 MapD。
<h5 id="内存">内存</h5>
　　内存对于服务器资源来说，是最廉价的，通常用来换取其他系统资源，提升系统处理能力，如内存换磁盘，内存换网络。总体来说，内存很少成为系统的瓶颈。
上面说了一些关于时间因素的优化方式，只是个人的理解和总结，本人才疏学浅，不能面俱到（例如分布式这块，将一个大任务分布到 N 个节点上执行，利用集群处理加快响应时间），如有地方写得不对，还请斧正。
<h4 id="空间">空间</h4>
　　空间，指的是一个系统请求的并行执行能力，相当于网络的带宽一样，一次能同时处理多少请求。目前空间的优化机制非常多，最常用的肯定是做集群，通过负载均衡 + 集群进行横向扩展，支

集群的应用系统最好是无状态的。当然纵向扩展也行，通过升级服务器的硬件设备，也能达到增加并处理的能力。增加服务器数量、升级服务器配置的方式，是扩展整个系统应用并发能力最直接的方式也是最普遍的方式。如果一个应用程序有状态，例如我之前做过一个应用程序，应用的请求是有状态，每个用户的请求必须要按顺序依次处理，这时候就需要做进程间的并发控制了，让请求在多个进程有序的处理。如果是这样的系统架构，那么服务器的横向扩展或者纵向扩展对于整个系统的并发能力影响程度有多大，还得看系统架构是怎么设计的了，具体的设计不在此讨论，另外写一篇文章专门讨跨进程的并发控制。

通过服务器节点横向扩展，服务器硬件纵向扩展，可以直接扩展整个集群的并行处理能力，这个最简单的方式，当然也可以通过优化应用程序本身来提高应用程序的并行处理能力，这个涉及到应用序内部的线程结构设计，在上面 CPU 部分谈了一下，如果是 IO 型的应用，那么适合基于线程的并发构，如果是计算密集型的，那么适合 SEDA 这种线程并发结构。例如 Springboot 提供了对应的两种理框架，Springboot1.x，提供的 springmvc，就是基于线程的并发结构，适合常规的基于请求-IO 作-应答的 IO 型应用程序，Springboot2.x 提供的 webflux，是基于 reactor 设计的高并发结构，内控制的线程的使用，提供了多种场景的 scheduler，适合对 cpu 敏感的计算密集型应用。注意，并不说一个应用程序内部，线程越多越好，线程切换会消耗 CPU 时钟，反而浪费 CPU 资源，这个得结合身应用合理的设计。

<h4 id="总结">总结</h4>

说一千道一万，影响整个系统的并发能力有很多，以上都是个人见解，不代表权威，如果有方写得不合理或者有问题，请直接留言挽尊。就算一个系统，在时间和空间上都处理的很好，一个好高并发架构应用，还必须同时具备其他的能力，比如稳定性、可靠性、鲁棒性、扩展性、自我恢复能等等。同时现在还流行虚拟化、容器化，应用程序的设计能否适应潮流的发展？架构师的道路，坎坷岖，任重而道远。