高性能服务器架构思路 - 码农网

分享到：
更多2
在服务器端程序开发领域，性能问题一直是备受关注的重点。业界有大量的框架、组件、类库都是以性能为卖点而广为人知。然而，服务器端程序在性能问题上应该有何种基本思路，这个却很少被这些项目的文档提及。本文正式希望介绍服务器端解决性能问题的基本策略和经典实践，并分为几个部分来说明：

1. 缓存策略的概念和实例

2.缓存策略的难点：不同特点的缓存数据的清理机制

3.分布策略的概念和实例

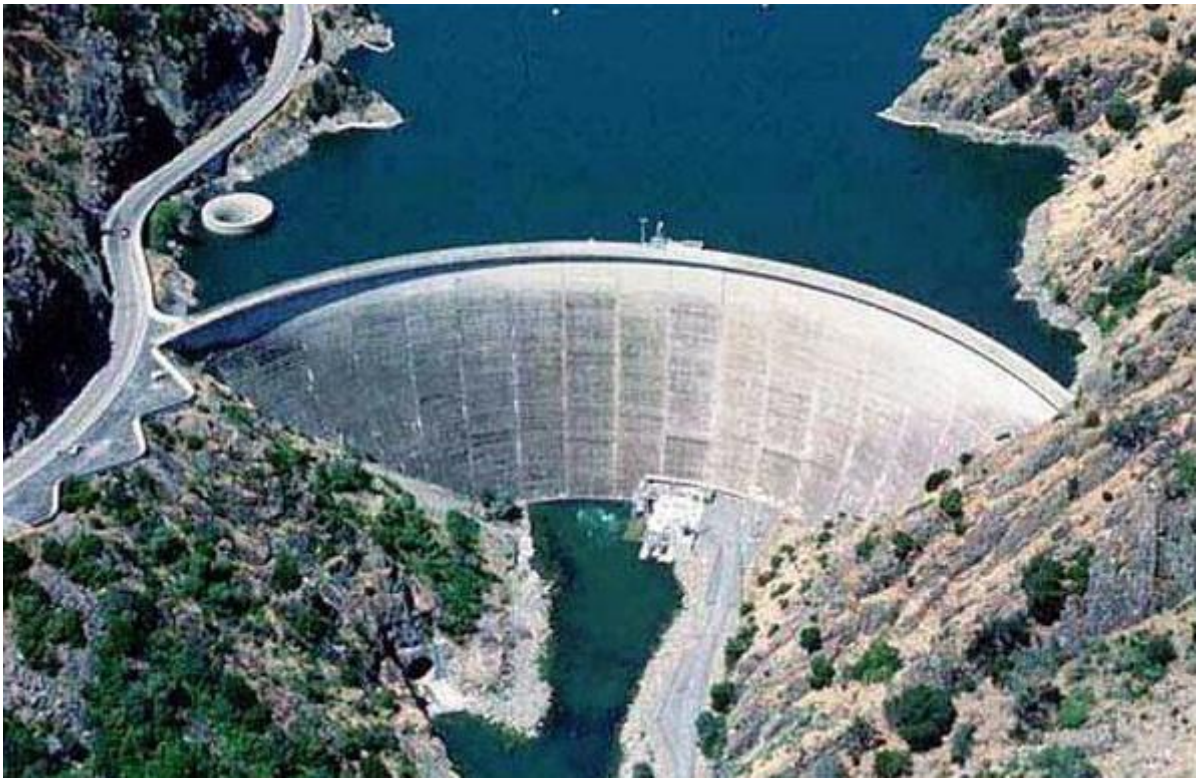
4.分布策略的难点：共享数据安全性与代码复杂度的平衡

缓存

缓存策略的概念

我们提到服务器端性能问题的时候，往往会混淆不清。因为当我们访问一个服务器时，出现服务卡住不能得到数据，就会认为是“性能问题”。但是实际上这个性能问题可能是有不同的原因，表现出来都是针对客户请求的延迟很长甚至中断。我们来看看这些原因有哪些：第一个是所谓并发数不足，也就是同时请求的客户过多，导致超过容纳能力的客户被拒绝服务，这种情况往往会因为服务器内存耗尽而导致的；第二个是处理延迟过长，也就是有一些客户的请求处理时间已经超过用户可以忍受的长度，这种情况常常表现为CPU占用满额100%。

我们在服务器开发的时候，最常用到的有下面这几种硬件：CPU、内存、磁盘、网卡。其中CPU是代表计算机处理时间的，硬盘的空间一般很大，主要是读写磁盘会带来比较大的处理延迟，而内存、网卡则是受存储、带宽的容量限制的。所以当我们的服务器出现性能问题的时候，就是这几个硬件某一个甚至几个都出现负荷占满的情况。这四个硬件的资源一般可以抽象成两类：一类是时间资源，比如CPU和磁盘读写；一类是空间资源，比如内存和网卡带宽。所以当我们的服务器出现性能问题，有一个最基本的思路，就是——时间空间转换。我们可以举几个例子来说明这个问题。



[水坝就是用水库空间来换流量时间的例子]

当我们访问一个WEB的网站的时候，输入的URL地址会被服务器变成对磁盘上某个文件的读取。如果有大量的用户访问这个网站，每次的请求都会造成对磁盘的读操作，可能会让磁盘不堪重负，导致无法即时读取到文件内容。但是如果我们写的程序，会把读取过一次的文件内容，长时间的保存在内存中，当有另外一个对同样文件的读取时，就直接从内存中把数据返回给客户端，就无需去让磁盘读取了。由于用户访问的文件往往很集中，所以大量的请求可能都能从内存中找到保存的副本，这样就能大大提高服务器能承载的访问量了。这种做法，就是用内存的空间，换取了磁盘的读写时间，属于用空间换时间的策略。

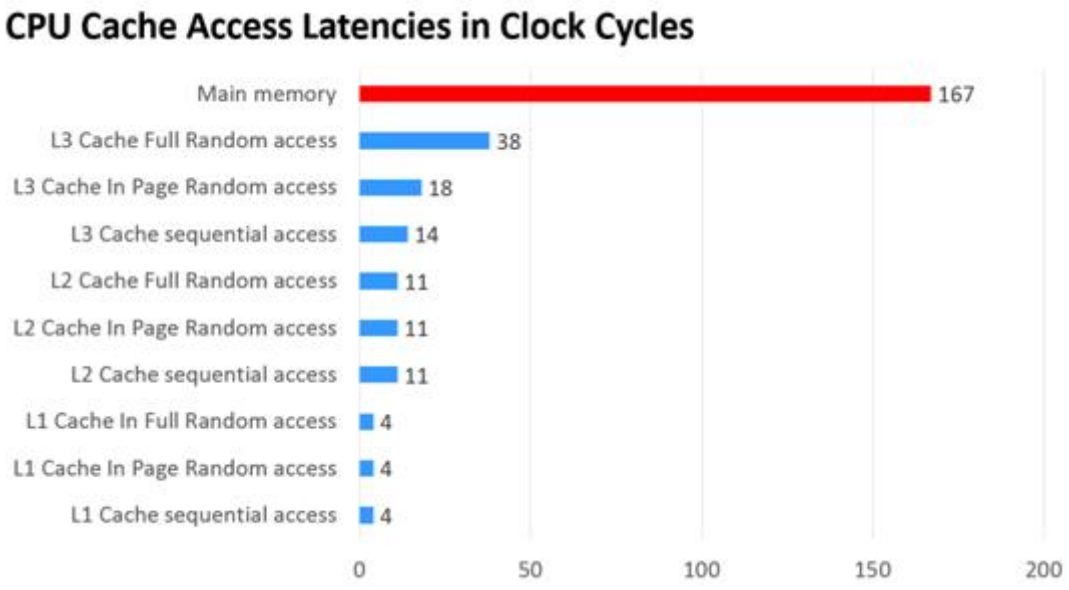
[方便面预先缓存了大量的烹饪操作]

举另外一个例子：我们写一个网络游戏的服务器端程序，通过读写数据库来提供玩家资料存档。如果有大量玩家进入这个服务器，必定有很多玩家的数据资料变化，比如升级、获得武器等等，这些通过读写数据库来实现的操作，可能会让数据库进程负荷过重，导致玩家无法即时完成游戏操作。我们会发现游戏中的读操作，大部分都是针是对一些静态数据的，比如游戏中的关卡数据、武器道具的具体信息；而很多写操作，实际上是会覆盖的，比如我的经验值，可能每打一个怪都会增加几十点，但是最后记录的只是最终的一个经验值，而不会记录下打怪的每个过程。所以我们也可以使用时空转换的策略来提供性能：我们可以用内存，把那些游戏中的静态数据，都一次性读取并保存起来，这样每次读这些数据，都和数据库无关了；而玩家的资料数据，则不是每次变化都去写数据库，而是先在内存中保持一个玩家数据的副本，所有的写操作都先去写内存中的结构，然后定期再由服务器主动写回到数据库中，这样可以把多次的写数据库操作变成一次写操作，也能节省很多写数据库的消耗。这种做法也是用空间换时间的策略。

[拼装家具很省运输空间，但是安装很费时]

最后说说用时间换空间的例子：假设我们要开发一个企业通讯录的数据存储系统，客户要求我们能保存下通讯录的每次新增、修改、删除操作，也就是这个数据的所有变更历史，以便可以让数据回退到任何一个过去的时间点。那么我们最简单的做法，就是这个数据在任何变化的时候，都拷贝一份副本。但是这样会非常的浪费磁盘空间，因为这个数据本身变化的部分可能只有很小一部分，但是要拷贝的副本可能很大。这种情况下，我们就可以在每次数据变化的时候，都记下一条记录，内容就是数据变化的情况：插入了一条内容是某某的联系方法、删除了一条某某的联系方法……，这样我们记录的数据，仅仅就是变化的部分，而不需要拷贝很多份副本。当我们需要恢复到任何一个时间点的时候，只需要按这些记录依次对数据修改一遍，直到指定的时间点的记录即可。这个恢复的时间可能会有点长，但是却可以大大节省存储空间。这就是用CPU的时间来换磁盘的存储空间的策略。我们现在常见的MySQL InnoDB日志型数据表，以及SVN源代码存储，都是使用这种策略的。

另外，我们的Web服务器，在发送HTML文件内容的时候，往往也会先用ZIP压缩，然后发送给浏览器，浏览器收到后要先解压，然后才能显示，这个也是用服务器和客户端的CPU时间，来换取网络带宽的空间。



在我们的计算机体系中，缓存的思路几乎无处不在，比如我们的CPU里面就有1级缓存、2级缓存，他们就是为了用这些快速的存储空间，换取对内存这种相对比较慢的存储空间的等待时间。我们的显示卡里面也带有大容量的缓存，他们是用来存储显示图形的运算结果的。

[通往大空间的郊区路上容易交通堵塞]

缓存的本质，除了让“已经处理过的数据，不需要重复处理”以外，还有“以快速的数据存储读写，代替较慢速的存储读写”的策略。我们在选择缓存策略进行时空转换的时候，必须明确我们要转换的时间和空间是否合理，是否能达到效果。比如早期有一些人会把WEB文件缓存在分布式磁盘上（例如NFS），但是由于通过网络访问磁盘本身就是一个比较慢的操作，而且还会占用可能就不充裕的网络带宽空间，导致性能可能变得更慢。

在设计缓存机制的时候，我们还容易碰到另外一个风险，就是对缓存数据的编程处理问题。如果我们要缓存的数据，并不是完全无需处理直接读写的，而是需要读入内存后，以某种语言的结构体或者对象来处理的，这就需要涉及到“序列化”和“反序列化”的问题。如果我们采用直接拷贝内存的方式来缓存数据，当我们的这些数据需要跨进程、甚至跨语言访问的时候，会出现那些指针、ID、句柄数据的失效。因为在另外一个进程空间里，这些“标记型”的数据都是不存在的。因此我们需要更深入的对数据缓存的方法，我们可能会使用所谓深拷贝的方案，也就是跟着那些指针去找出目标内存的数据，一并拷贝。一些更现代的做法，则是使用所谓序列化方案来解决这个问题，也就是用一些明确定义了的“拷贝方法”来定义一个结构体，然后用户就能明确的知道这个数据会被拷贝，直接取消了指针之类的内存地址数据的存在。比如著名的Protocol Buffer就能很方便的进行内存、磁盘、网络位置的缓存；现在我们常见的JSON，也被一些系统用来作为缓存的数据格式。

但是我们需要注意的是，缓存的数据和我们程序真正要操作的数据，往往是需要进行一些拷贝和运算的，这就是序列化和反序列化的过程，这个过程很快，也有可能很慢。所以我们在选择数据缓存结构的时候，必须要注意其转换时间，否则你缓存的效果可能被这些数据拷贝、转换消耗去很多，严重的甚至比不缓存更差。一般来说，缓存的数据越解决使用时的内存结构，其转换速度就越快，在这点上，Protocol Buffer采用TLV编码，就比不上直接memcpy的一个C结构体，但是比编码成纯文本的XML或者JSON要来的更快。因为编解码的过程往往要进行复杂的查表映射，列表结构等操作。

缓存策略的难点

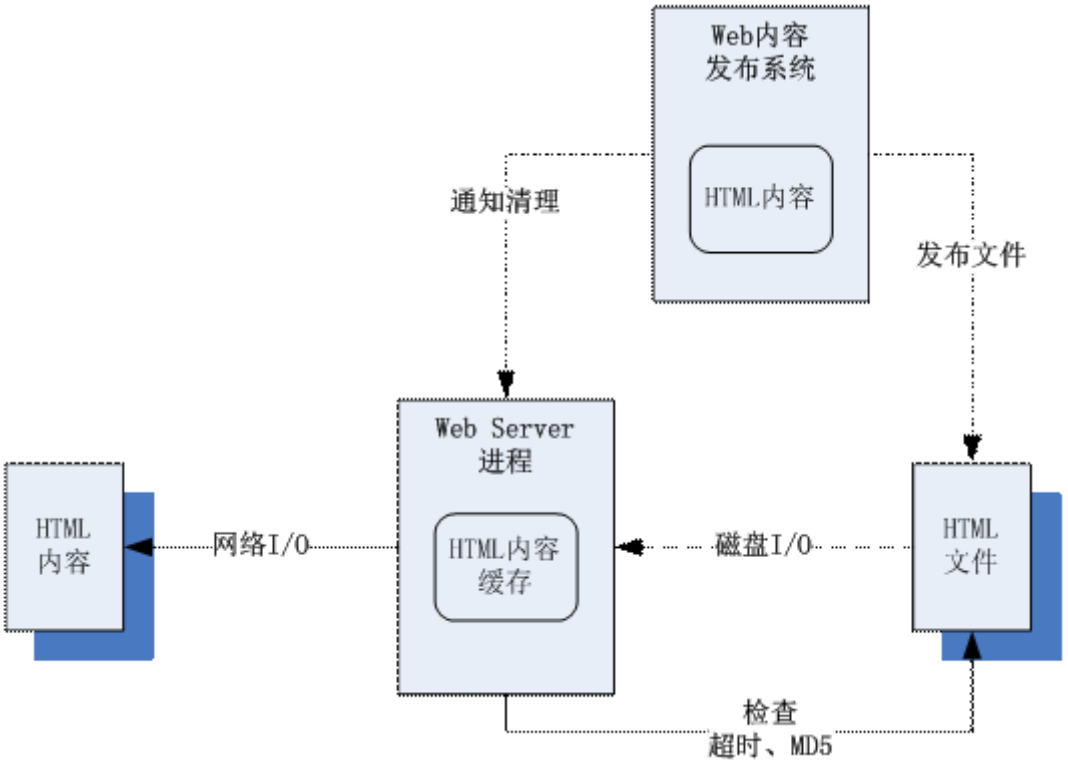
虽然使用缓存思想似乎是一个很简单的事情，但是缓存机制却有一个核心的难点，就是——缓存清理。我们所说的缓存，都是保存一些数据，但是这些数据往往是会变化的，我们要针对这些变化，清理掉保存的“脏”数据，却可能不是那么容易。

首先我们来看看最简单的缓存数据——静态数据。这种数据往往在程序的运行时是不会变化的，比如Web服务器内存中缓存的HTML文件数据，就是这种。事实上，所有的不是由外部用户上传的数据，都属于这种“运行时静态数据”。一般来说，我们对这种数据，可以采用两种建立缓存的方法：一是程序一启动，就一股脑把所有的静态数据从文件或者数据库读入内存；二就是程序启动的时候并不加载静态数据，而是等有用户访问相关数据的时候，才去加载，这也就是所谓lazy load的做法。第一种方法编程比较简单，程序的内存启动后就稳定了，不太容易出现内存漏洞（如果加载的缓存太多，程序在启动后立刻会因内存不足而退出，比较容易发现问题）；第二种方法程序启动很快，但要对缓存占用的空间有所限制或者规划，否则如果要缓存的数据太多，可能会耗尽内存，导致在线服务中断。

一般来说，静态数据是不会“脏”的，因为没有用户会去写缓存中的数据。但是在实际工作中，我们的在线服务往往会需要“立刻”变更一些缓存数据。比如在门户网站上发布了一条新闻，我们会希望立刻让所有访问的用户都看到。按最简单的做法，我们一般只要重启一下服务器进程，内存中的缓存就会消失了。对于静态缓存的变化频率非常低的业务，这样是可以的，但是如果是新闻网站，就不能每隔几分钟就重启一下WEB服务器进程，这样会影响大量在线用户的访问。常见的解决这类问题有两种处理策略：

第一种是使用控制命令。简单来说，就是在服务器进程上，开通一个实时的命令端口，我们可以通过网络数据包（如UDP包），或者Linux系统信号（如kill SIGUSR2进程号）之类的手段，发送一个命令消息给服务器进程，让进程开始清理缓存。这种清理可能执行的是最简单的“全部清理”，也有的可以细致一点的，让命令消息中带有“想清理的数据ID”这样的信息，比如我们发送给WEB服务器的清理消息网络包中会带一个字符串URL，表示要清理哪一个HTML文件的缓存。这种做法的好处是清理的操作很精准，可以明确的控制清理的时间和数据。但是缺点就是比较繁琐，手工去编写发送这种命令很烦人，所以一般我们会把清理缓存命令的工作，编写到上传静态数据的工具当中，比如结合到网站的内容发布系统中，一旦编辑提交了一篇新的新闻，发布系统的程序就自动的发送一个清理消息给WEB服务器。

第二种是使用字段判断逻辑。也就是服务器进程，会在每次读取缓存前，根据一些特征数据，快速的判断内存中的缓存和源数据内容，是否有不一致（是否脏）的地方，如果有不一致的地方，就自动清理这条数据的缓存。这种做法会消耗一部分CPU，但是就不需要人工去处理清理缓存的事情，自动化程度很高。现在我们的浏览器和WEB服务器之间，就有用这种机制：检查文件MD5；或者检查文件最后更新时间。具体的做法，就是每次浏览器发起对WEB服务器的请求时，除了发送URL给服务器外，还会发送一个缓存了此URL对应的文件内容的MD5校验串、或者是此文件在服务器上的“最后更新时间”（这个校验串和“最后更新时间”是第一次获的文件时一并从服务器获得的）；服务器收到之后，就会把MD5校验串或者最后更新时间，和磁盘上的目标文件进行对比，如果是一致的，说明这个文件没有被修改过（缓存不是“脏”的），可以直接使用缓存。否则就会读取目标文件返回新的内容给浏览器。这种做法对于服务器性能是有一定消耗的，所以如果往往我们还会搭配其他的缓存清理机制来用，比如我们会在设置一个“超时检查”的机制：就是对于所有的缓存清理检查，我们都简单的看看缓存存在的时间是否“超时”了，如果超过了，才进行下一步的检查，这样就不用每次请求都去算MD5或者看最后更新时间了。但是这样就存在“超时”时间内缓存变脏的可能性。

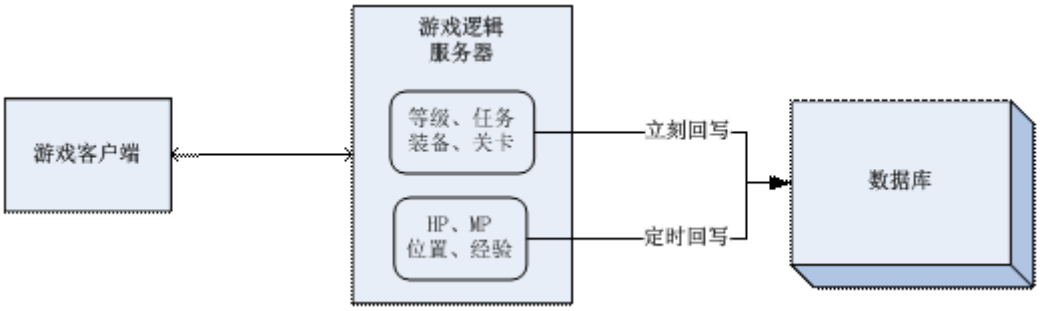


[WEB服务器静态缓存例子]

上面说了运行时静态的缓存清理，现在说说运行时变化的缓存数据。在服务器程序运行期间，如果用户和服务器之间的交互，导致了缓存的数据产生了变化，就是所谓“运行时变化缓存”。比如我们玩网络游戏，登录之后的角色数据就会从数据库里读出来，进入服务器的缓存（可能是堆内存或者memcached、共享内存），在我们不断进行游戏操作的时候，对应的角色数据就会产生修改的操作，这种缓存数据就是“运行时变化的缓存”。这种运行时变化的数据，有读和写两个方面的清理问题：由于缓存的数据会变化，如果另外一个进程从数据库读你的角色数据，就会发现和当前游戏里的数据不一致；如果服务器进程突然结束了，你在游戏里升级，或者捡道具的数据可能会从内存缓存中消失，导致你白忙活了半天，这就是没有回写（缓存写操作的清理）导致的问题。这种情况在电子商务领域也很常见，最典型的就是火车票网上购买的系统，火车票数据缓存在内存必须有合适的清理机制，否则让两个买了同一张票就麻烦了，但如果不缓存，大量用户同时抢票，服务器也应对不过来。因此在运行时变化的数据缓存，应该有一些特别的缓存清理策略。

在实际运行业务中，运行变化的数据往往是根据使用用户的增多而增多的，因此首先要考虑的问题，就是缓存空间不够的可能性。我们不太可能把全部数据都放到缓存的空间里，也不可能清理缓存的时候就全部数据一起清理，所以我们一般要对数据进行分割，这种分割的策略常见的有两种：一种是按重要级来分割，一种是按使用部分分割。

先举例说说“按重要级分割”，在网络游戏中，同样是角色的数据，有些数据的变化可能会每次修改都立刻回写到数据库（清理写缓存），其他一些数据的变化会延迟一段时间，甚至有些数据直到角色退出游戏才回写，如玩家的等级变化（升级了），武器装备的获得和消耗，这些玩家非常看重的数据，基本上会立刻回写，这些就是所谓最重要的缓存数据。而玩家的经验值变化、当前HP、MP的变化，就会延迟一段时间才写，因为就算丢失了缓存，玩家也不会太过关注。最后有些比如玩家在房间（地区）里的X/Y坐标，对话聊天的记录，可能会退出时回写，甚至不回写。这个例子说的是“写缓存”的清理，下面说说“读缓存”的按重要级分割清理。



假如我们写一个网店系统，里面容纳了很多产品，这些产品有一些会被用户频繁检索到，比较热销，而另外一些商品则没那么热销。热销的商品的余额、销量、评价都会比较频繁的变化，而滞销的商品则变化很少。所以我们在设计的时候，就应该按照不同商品的访问频繁程度，来决定缓存哪些商品的数据。我们在设计缓存的结构时，就应该构建一个可以统计缓存读写次数的指标，如果有些数据的读写频率过低，或者空闲（没有人读、写缓存）时间超长，缓存应该主动清理掉这些数据，以便其他新的数据能进入缓存。这种策略也叫做“冷热交换”策略。实现“冷热交换”的策略时，关键是要定义一个合理的冷热统计算法。一些固定的指标和算法，往往并不能很好的应对不同硬件、不同网络情况下的变化，所以现在人们普遍会用一些动态的算法，如Redis就采用了5种，他们是：

1.根据过期时间，清理最长时间没用过的

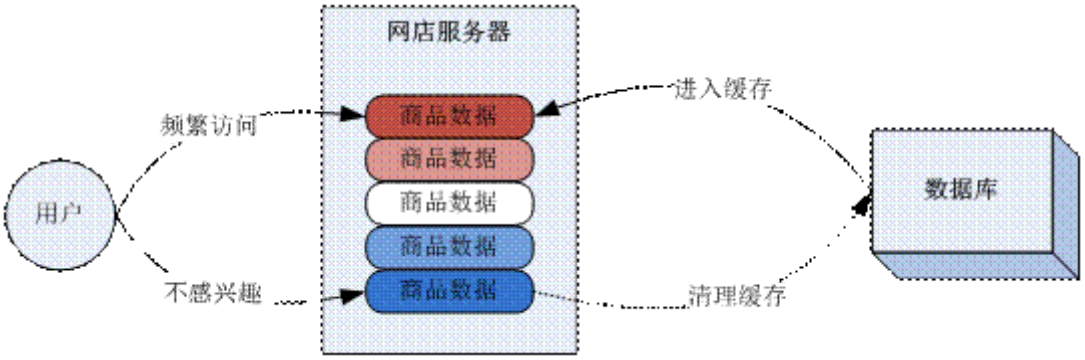
2.根据过期时间，清理即将过期的

3.根据过期时间，任意清理一个

4.无论是否过期，随机清理

5.无论是否过期，根据LRU原则清理：所谓LRU，就是Least Recently Used，最近最久未使用过。这个原则的思想是：如果一个数据在最近一段时间没有被访问到，那么在将来他被访问的可能性也很小。LRU是在操作系统中很常见的一种原则，比如内存的页面置换算法（也包括FIFO,LFU等），对于LRU的实现，还是非常有技巧的，但是本文就不详细去说明如何实现，留待大家上网搜索“LRU”关键字学习。

数据缓存的清理策略其实远不止上面所说的这些，要用好缓存这个武器，就要仔细研究需要缓存的数据特征，他们的读写分布，数据之中的差别。然后最大化的利用业务领域的知识，来设计最合理的缓存清理策略。这个世界上不存在万能的优化缓存清理策略，只存在针对业务领域最优化的策略，这需要我们程序员深入理解业务领域，去发现数据背后的规律。

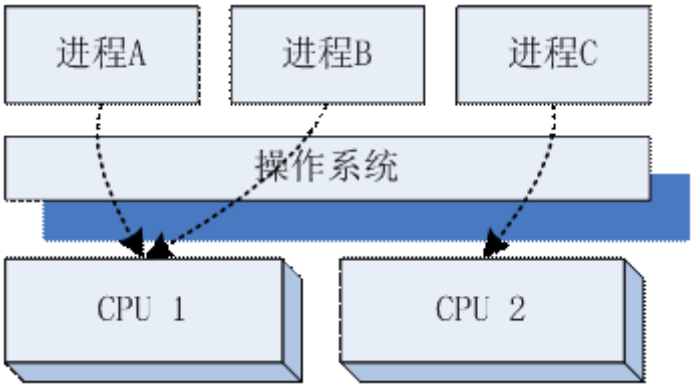


分布

分布策略的概念

任何的服务器的性能都是有极限的，面对海量的互联网访问需求，是不可能单靠一台服务器或者一个CPU来承担的。所以我们一般都会在运行时架构设计之初，就考虑如何能利用多个CPU、多台服务器来分担负载，这就是所谓分布的策略。分布式的服务器概念很简单，但是实现起来却比较复杂。因为我们写的程序，往往都是以一个CPU，一块内存为基础来设计的，所以要让多个程序同时运行，并且协调运作，这需要更多的底层工作。

首先出现能支持分布式概念的技术是多进程。在DOS时代，计算机在一个时间内只能运行一个程序，如果你想一边写程序，同时一边听mp3，都是不可能的。但是，在WIN95操作系统下，你就可以同时开多个窗口，背后就是同时在运行多个程序。在Unix和后来的Linux操作系统里面，都普遍支持了多进程的技术。所谓的多进程，就是操作系统可以同时运行我们编写的多个程序，每个程序运行的时候，都好像自己独占着CPU和内存一样。在计算机只有一个CPU的时候，实际上计算机会分时复用的运行多个进程，CPU在多个进程之间切换。但是如果这个计算机有多个CPU或者多个CPU核，则会真正的有几个进程同时运行。所以进程就好像一个操作系统提供的运行时“程序盒子”，可以用来在运行时，容纳任何我们想运行的程序。当我们掌握了操作系统的多进程技术后，我们就可以把服务器上的运行任务，分为多个部分，然后分别写到不同的程序里，利用上多CPU或者多核，甚至是多个服务器的CPU一起来承担负载。

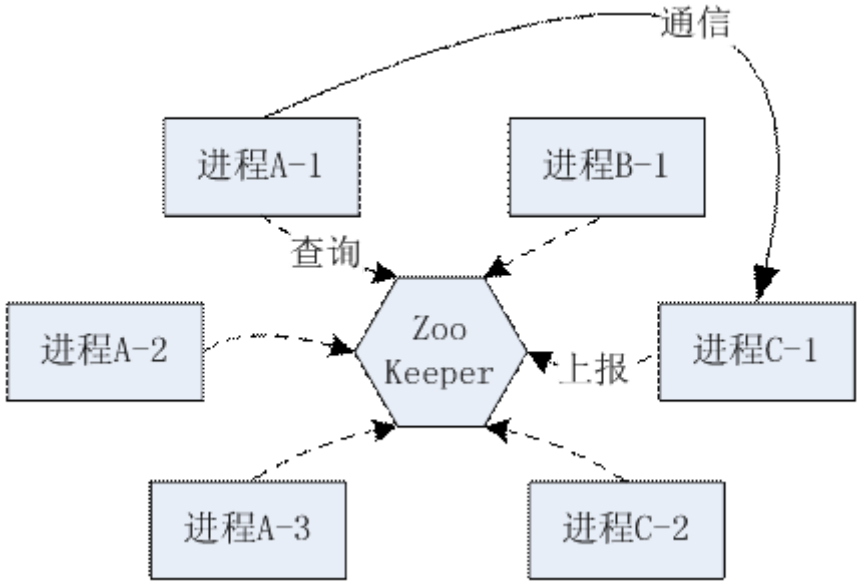


[多进程利用多CPU]

这种划分多个进程的架构，一般会有两种策略：一种是按功能来划分，比如负责网络处理的一个进程，负责数据库处理的一个进程，负责计算某个业务逻辑的一个进程。另外一种策略是每个进程都是同样的功能，只是分担不同的运算任务而已。使用第一种策略的系统，运行的时候，直接根据操作系统提供的诊断工具，就能直观的监测到每个功能模块的性能消耗，因为操作系统提供进程盒子的同时，也能提供对进程的全方位的监测，比如CPU占用、内存消耗、磁盘和网络I/O等等。但是这种策略的运维部署会稍微复杂一点，因为任何一个进程没有启动，或者和其他进程的通信地址没配置好，都可能导致整个系统无法运作；而第二种分布策略，由于每个进程都是一样的，这样的安装部署就非常简单，性能不够就多找几个机器，多启动几个进程就完成了，这就是所谓的平行扩展。

现在比较复杂的分布式系统，会结合这两种策略，也就是说系统既按一些功能划分出不同的具体功能进程，而这些进程又是可以平行扩展的。当然这样的系统在开发和运维上的复杂度，都是比单独使用“按功能划分”和“平行划分”要更高的。由于要管理大量的进程，传统的依靠配置文件来配置整个集群的做法，会显得越来越不实用：这些运行中的进程，可能和其他很多进程产生通信关系，当其中一个进程变更通信地址时，势必影响所有其他进程的配置。所以我们需要集中的管理所有进程的通信地址，当有变化的时候，只需要修改一个地方。在大量进程构建的集群中，我们还会碰到容灾和扩容的问题：当集群中某个服务器出现故障，可能会有一些进程消失；而当我们需要增加集群的承载能力时，我们又需要增加新的服务器以及进程。这些工作在长期运行的服务器系统中，会是比较常见的任务，如果整个分布系统有一个运行中的中心进程，能自动化的监测所有的进程状态，一旦有进程加入或者退出集群，都能即时的修改所有其他进程的配置，这就形成了一套动态的多进程管理系统。开源的ZooKeeper给我们提供了一个可以充当这种动态集群中心的实现方案。由于ZooKeeper本身是可以平行扩展的，所以它自己也是具备一定容灾能力的。现在越来越多的分布式系统都开始使用以ZooKeeper为集群中心的动态进

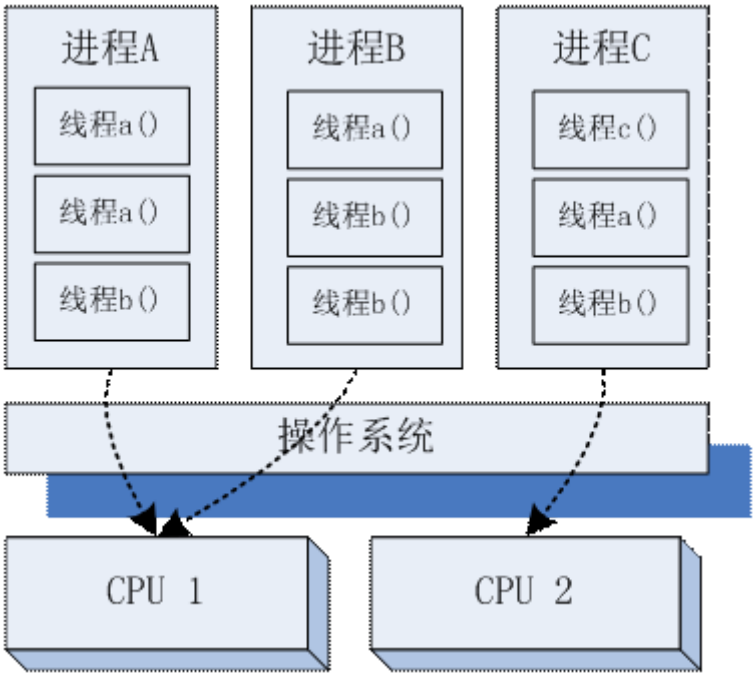
程管理策略了。



[动态进程集群]

在调用多进程服务的策略上，我们也会有一定的策略选择，其中最著名的策略有三个：一个是动态负载均衡策略；一个是读写分离策略；一个是一致性哈希策略。动态负载均衡策略，一般会搜集多个进程的服务状态，然后挑选一个负载最轻的进程来分发服务，这种策略对于比较同质化的进程是比较合适的。读写分离策略则是关注对持久化数据的性能，比如对数据库的操作，我们会提供一批进程专门用于提供读数据的服务，而另外一个（或多个）进程用于写数据的服务，这些写数据的进程都会每次写多份拷贝到“读服务进程”的数据区（可能就是单独的数据库），这样在对外提供服务的时候，就可以提供更多的硬件资源。一致性哈希策略是针对任何一个任务，看看这个任务所涉及读写的数据，是属于哪一片的，是否有某种可以缓存的特征，然后按这个数据的ID或者特征值，进行“一致性哈希”的计算，分担给对应的处理进程。这种进程调用策略，能非常的利用上进程内的缓存（如果存在），比如我们的一个在线游戏，由100个进程承担服务，那么我们就可以把游戏玩家的ID，作为一致性哈希的数据ID，作为进程调用的KEY，如果目标服务进程有缓存游戏玩家的数据，那么所有这个玩家的操作请求，都会被转到这个目标服务进程上，缓存的命中率大大提高。而使用“一致性哈希”，而不是其他哈希算法，或者取模算法，主要是考虑到，如果服务进程有一部分因故障消失，剩下的服务进程的缓存依然可以有效，而不会整个集群所有进程的缓存都失效。具体有兴趣的读者可以搜索“一致性哈希”一探究竟。

以多进程利用大量的服务器，以及服务器上的多个CPU核心，是一个非常有效的手段。但是使用多进程带来的额外的编程复杂度的问题。一般来说我们认为最好是每个CPU核心一个进程，这样能最好的利用硬件。如果同时运行的进程过多，操作系统会消耗很多CPU时间在不同进程的切换过程上。但是，我们早期所获得的很多API都是阻塞的，比如文件I/O，网络读写，数据库操作等。如果我们只用有限的进程来执行带这些阻塞操作的程序，那么CPU会大量被浪费，因为阻塞的API会让有限的这些进程停着等待结果。那么，如果我们希望能处理更多的任务，就必须要启动更多的进程，以便充分利用那些阻塞的时间，但是由于进程是操作系统提供的“盒子”，这个盒子比较大，切换耗费的时间也比较多，所以大量并行的进程反而会无谓的消耗服务器资源。加上进程之间的内存一般是隔离的，进程间如果要交换一些数据，往往需要使用一些操作系统提供的工具，比如网络socket，这些都会额外消耗服务器性能。因此，我们需要一种切换代价更少，通信方式更便捷，编程方法更简单的并行技术，这个时候，多线程技术出现了。



[在进程盒子里面的线程盒子]

多线程的特点是切换代价少，可以同时访问内存。我们可以在编程的时候，任意让某个函数放入新的线程去执行，这个函数的参数可以是任何的变量或指针。如果我们希望和这些运行时的线程通信，只要读、写这些指针指向的变量即可。在需要大量阻塞操作的时候，我们可以启动大量的线程，这样就能较好的利用CPU的空闲时间；线程的切换代价比进程低得多，所以我们能利用的CPU也会多很多。线程是一个比进程更小的“程序盒子”，他可以放入某一个函数调用，而不是一个完整的程序。一般来说，如果多个线程只是在一个进程里面运行，那其实是没有利用到多核CPU的并行好处的，仅仅是利用了单个空闲的CPU核心。但是，在JAVA和C#这类带虚拟机的语言中，多线程的实现底层，会根据具体的操作系统的任务调度单位（比如进程），尽量让线程也成为操作系统可以调度的单位，从而利用上多个CPU核心。比如Linux2.6之后，提供了NPTL的内核线程模型，JVM就提供了JAVA线程到NPTL内核线程的映射，从而利用上多核CPU。而Windows系统中，据说本身线程就是系统的最小调度单位，所以多线程也是利用上多核CPU的。所以我们在使用JAVA\C#编程的时候，多线程往往已经同时具备了多进程利用多核CPU、以及切换开销低的两个好处。

早期的一些网络聊天室服务，结合了多线程和多进程使用的例子。一开始程序会启动多个广播聊天的进程，每个进程都代表一个房间；每个用户连接到聊天室，就为他启动一个线程，这个线程会阻塞的读取用户的输入流。这种模型在使用阻塞API的环境下，非常简单，但也非常有效。

当我们在广泛使用多线程的时候，我们发现，尽管多线程有很多优点，但是依然会有明显的两个缺点：一个内存占用比较大且不太可控；第二个是多个线程对于用一个数据使用时，需要考虑复杂的“锁”问题。由于多线程是基于对一个函数调用的并行运行，这个函数里面可能会调用很多个子函数，每调用一层子函数，就会要在栈上占用新的内存，大量线程同时在运行的时候，就会同时存在大量的栈，这些栈加在一起，可能会形成很大的内存占用。并且，我们编写服务器端程序，往往希望资源占用尽量可控，而不是动态变化太大，因为你不知道什么时候会因为内存用完而当机，在多线程的程序中，由于程序运行的内容导致栈的伸缩幅度可能很大，有可能超出我们预期的内存占用，导致服务的故障。而对于内存的“锁”问题，一直是多线程中复杂的课题，很多多线程工具库，都推出了大量的“无锁”容器，或者“线程安全”的容器，并且还大量设计了很多协调线程运作的类库。但是这些复杂的工具，无疑都是证明了多线程对于内存使用上的问题。

[同时排多条队就是并行]

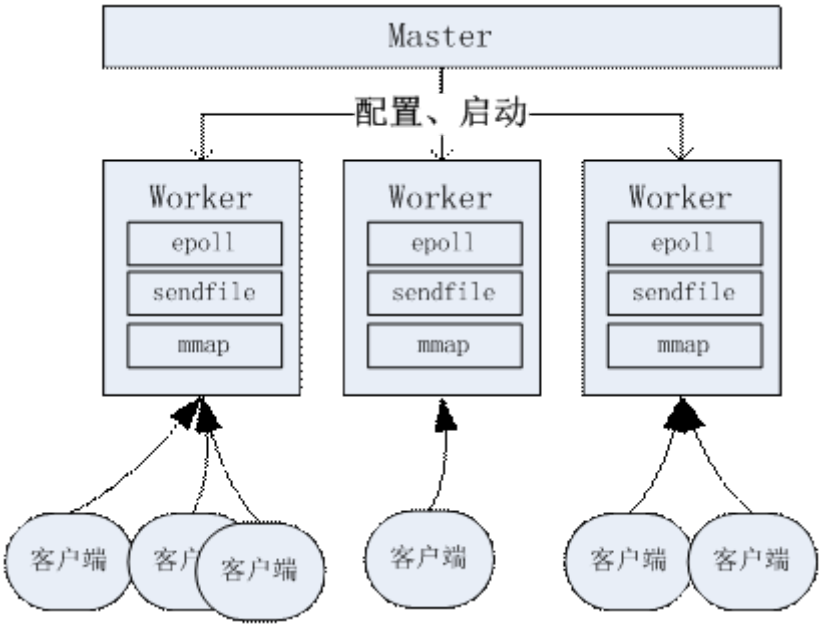
由于多线程还是有一定的缺点，所以很多程序员想到了一个釜底抽薪的方法：使用多线程往往是因为阻塞式API的存在，比如一个read()操作会一直停止当前线程，那么我们能不能让这些操作变成不阻塞呢？——selector/epoll就是Linux退出的非阻塞式API。如果我们使用了非阻塞的操作函数，那么我们也无需用多线程来并发的等待阻塞结果。我们只需要用一个线程，循环的检查操作的状态，如果有结果就处理，无结果就继续循环。这种程序的结果往往会有一个大的死循环，称为主循环。在主循环体内，程序员可以安排每个操作事件、每个逻辑状态的处理逻辑。这样CPU既无需在多线程间切换，也无需处理复杂的并行数据锁的问题——因为只有一个线程在运行。这种就是被称为“并发”的方案。

[服务员兼了点菜、上菜就是并发]

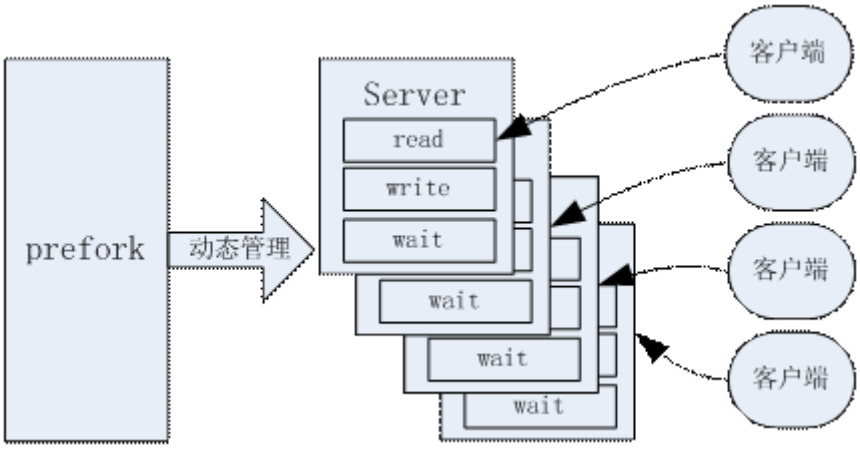
实际上计算机底层早就有使用并发的策略，我们知道计算机对于外部设备（比如磁盘、网卡、显卡、声卡、键盘、鼠标），都使用了一种叫“中断”的技术，早期的电脑使用者可能还被要求配置IRQ号。这个中断技术的特点，就是CPU不会阻塞的一直停在等待外部设备数据的状态，而是外部数据准备好后，给CPU发一个“中断信号”，让CPU转去处理这些数据。非阻塞的编程实际上也是类似这种行为，CPU不会一直阻塞的等待某些I/O的API调用，而是先处理其他逻辑，然后每次主循环去主动检查一下这些I/O操作的状态。

多线程和异步的例子，最著名就是Web服务器领域的Apache和Nginx的模型。Apache是多进程/多线程模型的，它会在启动的时候启动一批进程，作为进程池，当用户请求到来的时候，从进程池中分配处理进程给具体的用户请

求，这样可以节省多进程/线程的创建和销毁开销，但是如果同时有大量的请求过来，还是需要消耗比较高的进程/线程切换。而Nginx则是采用epoll技术，这种非阻塞的做法，可以让一个进程同时处理大量的并发请求，而无需反复切换。对于大量的用户访问场景下，apache会存在大量的进程，而nginx则可以仅用有限的进程（比如按CPU核心数来启动），这样就会比apache节省了不少“进程切换”的消耗，所以其并发性能会更好。



[Nginx的固定多进程，一个进程异步处理多个客户端]



[Apache的多态多进程，一个进程处理一个客户]

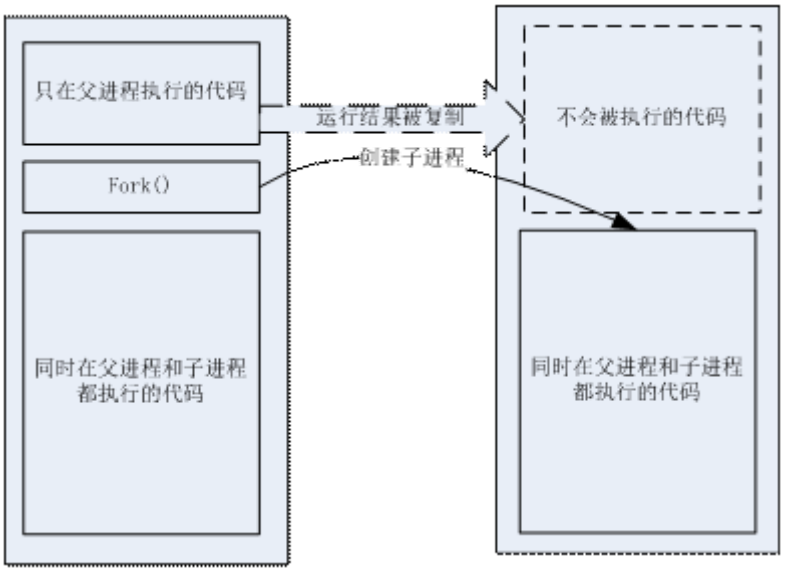
在现代服务器端软件中，nginx这种模型的运维管理会更简单，性能消耗也会稍微更小一点，所以成为最流行的进程架构。但是这种好处，会付出一些另外的代价：非阻塞代码在编程的复杂度变大。

分布式编程复杂度

以前我们的代码，从上往下执行，每一行都会占用一定的CPU时间，这些代码的直接顺序，也是和编写的顺序基本一致，任何一行代码，都是唯一时刻的执行任务。当我们在编写分布式程序的时候，我们的代码将不再好像那些单进程、单线程的程序一样简单。我们要把同时运行的不同代码，在同一段代码中编写。就好像我们要把整个交响乐团的每个乐器的乐谱，全部写到一张纸上。为了解决这种编程的复杂度，业界发展出了多种编码形式。

在多进程的编码模型上，fork()函数可以说一个非常典型的代表。在一段代码中，fork()调用之后的部分，可能会被新的进程中执行。要区分当前代码的所在进程，要靠fork()的返回值变量。这种做法，等于把多个进程的代码都合并到一块，然后通过某些变量作为标志来划分。这样的写法，对于不同进程代码大部份相同的“同质进

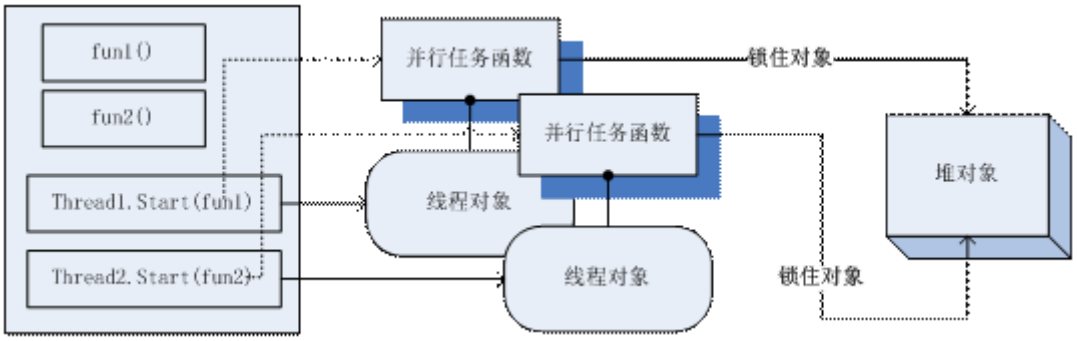
程”来说，还是比较方便的，最怕就是有大量的不同逻辑要用不同的进程来处理，这种情况下，我们就只能自己通过规范fork()附近的代码，来控制混乱的局面。比较典型的是把fork()附近的代码弄成一个类似分发器（dispatcher）的形式，把不同功能的代码放到不同的函数中，以fork之前的标记变量来决定如何调用。



[动态多进程的代码模式]

在我们使用多线程的API时，情况就会好很多，我们可以用一个函数指针，或者一个带回调方法的对象，作为线程执行的主体，并且以句柄或者对象的形式来控制这些线程。作为开发人员，我们只要掌握了对线程的启动、停止等有限的几个API，就能很好的对并行的多线程进行控制。这对比多进程的fork()来说，从代码上看会更直观，只是我们必须要分清楚调用一个函数，和新建一个线程去调用一个函数，之间的差别：新建线程去调用函数，这个操作会很快的结束，并不会依序去执行那个函数，而是代表着，那个函数中的代码，可能和线程调用之后的代码，交替的执行。

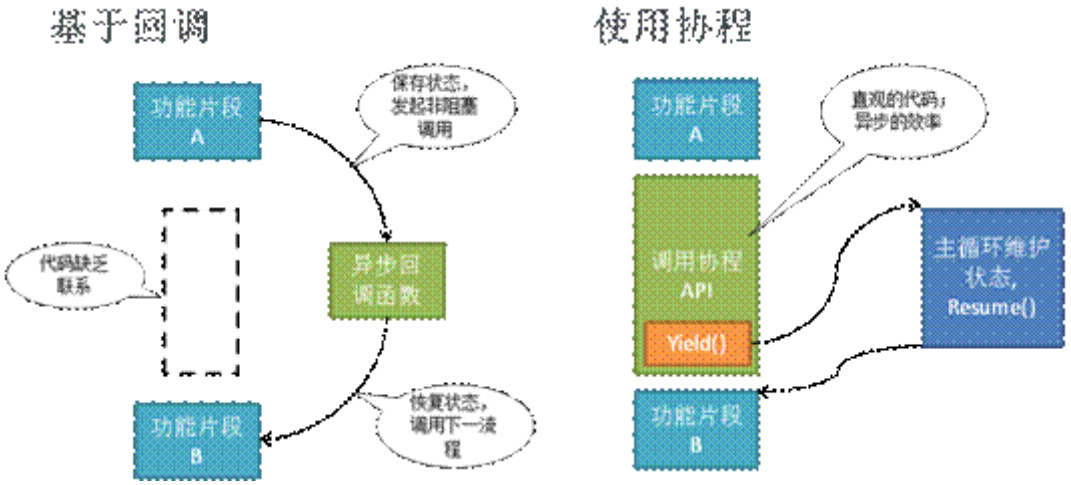
由于多线程把“并行的任务”作为一个明确的编程概念定义了出来，以句柄、对象的形式封装好，那么我们自然会希望对多线程能更多复杂而细致的控制。因此出现了很多多线程相关的工具。比较典型的编程工具有线程池、线程安全容器、锁这三类。线程池提供给我们以“池”的形态，自动管理线程的能力：我们不需要自己去考虑怎么建立线程、回收线程，而是给线程池一个策略，然后输入需要执行的任务函数，线程池就会自动操作，比如它会维持一个同时运行线程数量，或者保持一定的空闲线程以节省创建、销毁线程的消耗。在多线程操作中，不像多进程在内存上完全是区分开的，所以可以访问同一份内存，也就是对堆里面的同一个变量进行读写，这就可能产生程序员所预计不到的情况（因为我们写程序只考虑代码是顺序执行的）。还有一些对象容器，比如哈希表和队列，如果被多个线程同时操作，可能还会因为内部数据对不上，造成严重的错误，所以很多人开发了一些可以被多个线程同时操作的容器，以及所谓“原子”操作的工具，以解决这样的问题。有些语言如Java，在语法层面，就提供了关键字来对某个变量进行“上锁”，以保障只有一个线程能操作它。多线程的编程中，很多并行任务，是有一定的阻塞顺序的，所以有各种各样的锁被发明出来，比如倒数锁、排队锁等等。java.concurrent库就是多线程工具的一个大集合，非常值得学习。然而，多线程的这些五花八门的武器，其实也是证明了多线程本身，是一种不太容易使用的顺手的技术，但是我们一下子还没有更好的替代方案罢了。



[多线程的对象模型]

在多线程的代码下，除了启动线程的地方，是和正常的执行顺序不同以外，其他的基本都还是比较近似单线程代码的。但是如果在异步并发的代码下，你会发现，代码一定要装入一个个“回调函数”里。这些回调函数，从代码的组织形态上，几乎完全无法看出来其预期的执行顺序，一般只能在运行的时候通过断点或者日志来分析。这就对代码阅读带来了极大的障碍。因此现在有越来越多的程序员关注“协程”这种技术：可以用类似同步的方法来写异步程序，而无需把代码塞到不同的回调函数里面。协程技术最大的特点，就是加入了一个叫yield的概念，这个关键字所在的代码行，是一个类似return的作用，但是又代表着后续某个时刻，程序会从yield的地方继续往下执行。这样就把那些需要回调的代码，从函数中得以解放出来，放到yield的后面了。在很多客户端游戏引擎中，我们写的代码都是由一个框架，以每秒30帧的速度在反复执行，为了让一些任务，可以分别放在各帧中运行，而不是一直阻塞导致“卡帧”，使用协程就是最自然和方便的了——Unity3D就自带了协程的支持。

在多线程同步程序中，我们的函数调用栈就代表了一系列同属一个线程的处理。但是在单线程的异步回调的编程模式下，我们的一个回调函数是无法简单的知道，是在处理哪一个请求的序列中。所以我们往往需要自己写代码去维持这样的状态，最常见的做法是，每个并发任务启动的时候，就产生一个序列号（seqid），然后在所有的对这个并发任务处理的回调函数中，都传入这个seqid参数，这样每个回调函数，都可以通过这个参数，知道自己在处理哪个任务。如果有些不同的回调函数，希望交换数据，比如A函数的处理结果希望B函数能得到，还可以用seqid作为key把结果存放到一个公共的哈希表容器中，这样B函数根据传入的seqid就能去哈希表中获得A函数存入的结果了，这样的一份数据我们往往叫做“会话”。如果我们使用协程，那么这些会话可能都不需要自己来维持了，因为协程中的栈代表了会话容器，当执行序列切换到某个协程中的时候，栈上的局部变量正是之前的处理过程的内容结果。



[协程的代码特征]

为了解决异步编程的回调这种复杂的操作，业界还发明了很多其他的手段，比如lamda表达式、闭包、promise模型等等，这些都是希望我们，能从代码的表面组织上，把在多个不同时间段上运行的代码，以业务逻辑的形式组织到一起。

最后我想说说函数式编程，在多线程的模型下，并行代码带来最大的复杂性，就是对堆内存的同时操作。所以我们才弄出来锁的机制，以及一大批对付死锁的策略。而函数式编程，由于根本不使用堆内存，所以就无需处理什么锁，反而让整个事情变得非常简单。唯一需要改变的，就是我们习惯于把状态放到堆里面的编程思路。函数式编程的语言，比如LISP或者Erlang，其核心数据结果是链表——一种可以表示任何数据结构的结构。我们可以把所有的状态，都放到链表这个数据列车中，然后让一个个函数去处理这串数据，这样同样也可以传递程序的状态。这是一种用栈来代替堆的编程思路，在多线程并发的环境下，非常的有价值。

分布式程序的编写，一直都伴随着大量的复杂性，影响我们对代码的阅读和维护，所以我们才有各种各样的技术和概念，试图简化这种复杂性。也许我们无法找到任何一个通用的解决方案，但是我们可以通过理解各种方案的目标，来选择最适合我们的场景：

1 动态多进程fork——同质的并行任务

1 多线程——能明确划的逻辑复杂的并行任务

1 异步并发回调——对性能要求高，但中间会被阻塞的处理较少的并行任务

1 协程——以同步的写法编写并发的任务，但是不合适发起复杂的动态并行操作。

1 函数式编程——以数据流为模型的并行处理任务

分布式数据通信

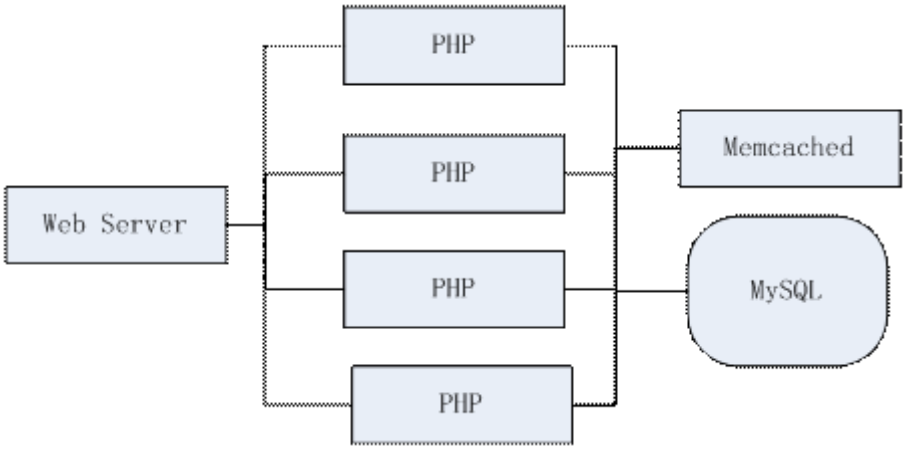
分布式的编程中，对于CPU时间片的切分本身不是难点，最困难的地方在于并行的多个代码片段，如何进行通信。因为任何一个代码段，都不可能完全单独的运作，都需要和其他代码产生一定的依赖。在动态多进程中，我们往往只能通过父进程的内存提供共享的初始数据，运行中则只能通过操作系统间的通讯方式了：Socket、信号、共享内存、管道等等。无论那种做法，这些都带来了一堆复杂的编码。这些方式大部分都类似于文件操作：一个进程写入、另外一个进程读出。所以很多人设计了一种叫“消息队列”的模型，提供“放入”消息和“取出”消息的接口，底层则是可以用Socket、共享内存、甚至是文件来实现。这种做法几乎能够处理任何状况下的数据通讯，而且有些还能保存消息。但是缺点是每个通信消息，都必须经过编码、解码、收包、发包这些过程，对处理延迟有一定的消耗。

如果我们在多线程中进行通信，那么我们可以直接对某个堆里面的变量直接进行读写，这样的性能是最高的，使用也非常方便。但是缺点是可能出现几个线程同时使用变量，产生了不可预期的结果，为了对付这个问题，我们设计了对变量的“锁”机制，而如何使用锁又成为另外一个问题，因为可能出现所谓的“死锁”问题。所以我们一般会用一些“线程安全”的容器，用来作为多线程间通讯的方案。为了协调多个线程之间的执行顺序，还可以使用很多种类型的“工具锁”。

在单线程异步并发的情况下，多个会话间的通信，也是可以通过直接对变量进行读写操作，而且不会出现“锁”的问题，因为本质上每个时刻都只有一个段代码会操作这个变量。然而，我们还是需要对这些变量进行一定规划和整理，否则各种指针或全局变量在代码中散布，也是很出现BUG的。所以我们一般会把“会话”的概念变成一个数据容器，每段代码都可以把这个会话容器作为一个“收件箱”，其他的并发任务如果需要在这个任务中通讯，就把数据放入这个“收件箱”即可。在WEB开发领域，和cookie对应的服务器端Session机制，就是这种概念的典型实现。

分布式缓存策略

在分布式程序架构中，如果我们需要整个体系有更高的稳定性，能够对进程容灾或者动态扩容提供支持，那么最难解决的问题，就是每个进程中的内存状态。因为进程一旦毁灭，内存中的状态会消失，这就很难不影响提供的服务。所以我们需要一种方法，让进程的内存状态，不太影响整体服务，甚至最好能变成“无状态”的服务。当然“状态”如果不写入磁盘，始终还是需要某些进程来承载的。在现在流行的WEB开发模式中，很多人会使用PHP+Memcached+MySQL这种模型，在这里，PHP就是无状态的，因为状态都是放在Memcached里面。这种做法对于PHP来说，是可以随时动态的毁灭或者新建，但是Memcached进程就要保证稳定才行；而且Memcached作为一个额外的进程，和它通信本身也会消耗更多的延迟时间。因此我们需要一种更灵活和通用的进程状态保存方案，我们把这种任务叫做“分布式缓存”的策略。我们希望进程在读取数据的时候，能有最高的性能，最好能和在堆内存中读写类似，又希望这些缓存数据，能被放在多个进程内，以分布式的形态提供高吞吐的服务，其中最关键的问题，就是缓存数据的同步。



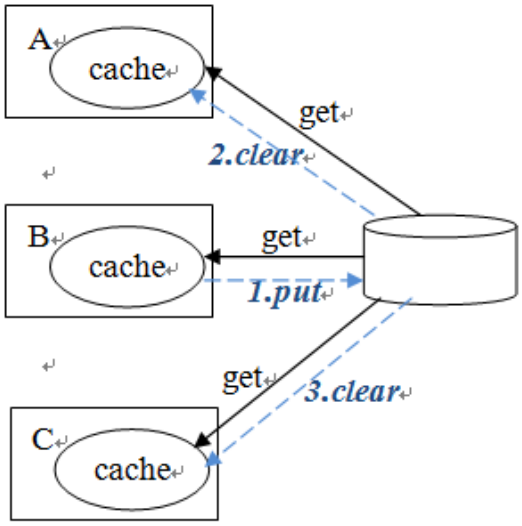
[PHP常用Memached做缓存]

为了解决这个问题，我们需要先一步步来分解这个问题：

首先，我们的缓存应该是某种特定形式的对象，而不应该是任意类型的变量。因为我们需要对这些缓存进行标准化的管理，尽管C++语言提供了运算重载，我们可以对“=”号的写变量操作进行重新定义，但是现在基本已经没有人推荐去做这样的事。而我们手头就有最常见的一种模型，适合缓存这种概念的使用，它就是——哈希表。所有的哈希表（或者是Map接口），都是把数据的存放，分为key和value两个部分，我们可以把想要缓存的数据，作为value存放到“表”当中，同时我们也可以用key把对应的数据取出来，而“表”对象就代表了缓存。

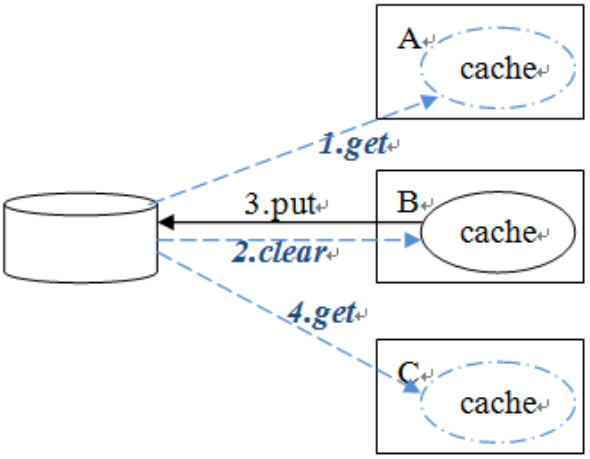
其次我们需要让这个“表”能在多个进程中都存在。如果每个进程中的数据都毫无关联，那问题其实就非常简单，但是如果我们可能从A进程把数据写入缓存，然后在B进程把数据读取出来，那么就比较复杂了。我们的“表”要有能把数据在A、B两个进程间同步的能力。因此我们一般会用三种策略：租约清理、租约转发、修改广播

1　　租约清理，一般是指，我们把存放某个key的缓存的进程，称为持有这个key的数据的“租约”，这个租约要登记到一个所有进程都能访问到的地方，比如是ZooKeeper集群进程。那么在读、写发生的时候，如果本进程没有对应的缓存，就先去查询一下对应的租约，如果被其他进程持有，则通知对方“清理”，所谓“清理”，往往是指删除用来读的数据，回写用来写的数据到数据库等持久化设备，等清理完成后，在进行正常的读写操作，这些操作可能会重新在新的进程上建立缓存。这种策略在缓存命中率比较高的情况下，性能是最好的，因为一般无需查询租约情况，就可以直接操作；但如果缓存命中率低，那么就会出现缓存反复在不同进程间“移动”，会严重降低系统的处理性能。



读缓存同步

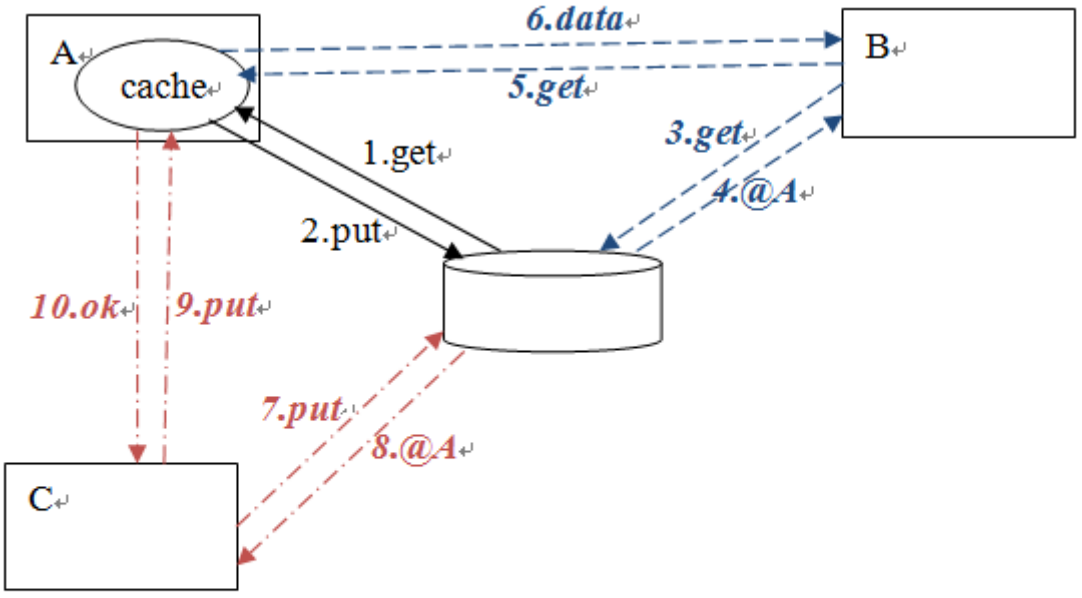
- 多点缓存性能好
- 写操作同步时间不保证



写缓存同步

- 单点缓存保持一致
- 频繁切换易损失性能

1　　租约转发。同样，我们把存放某个KEY的缓存的进程，称为持有这个KEY数据的“租约”，同时也要登记到集群的共享数据进程中。和上面租约清理不同的地方在于，如果发现持有租约的进程不是本次操作的进程，就会把整个数据的读、写请求，都通过网络“转发”个持有租约的进程，然后等待他的操作结果返回。这种做法由于每次操作都需要查询租约，所以性能会稍微低一些；但如果缓存命中率不高，这种做法能把缓存的操作分担到多个进程上，而且也无需清理缓存，这比租约清理的策略适应性更好。

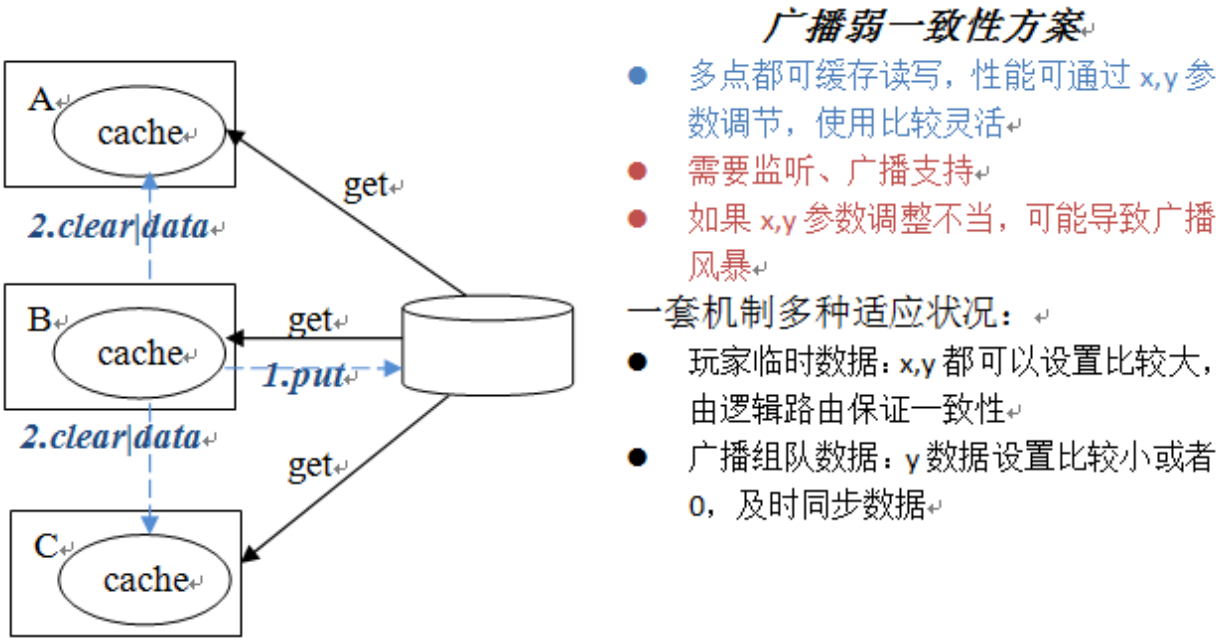


根据租约操作

- 无需监听和通知
- 多点访问性能差

1　　修改广播。上面两种策略，都需要维护一份缓存数据的租约，但是本身对于租约的操作，就是一种比较耗费性能的事情。所以有时候可以采用一些更简单，但可能承受一些不一致性的策略：对于读操作，每个节点的读都建立缓存，每次读都判断是否超过预设的读冷却时间x，超过则清理缓存从持久化重建；对于写操作，么个节点上都判断是否超过预设的写冷却时间y，超过则展开清理操作。清理操作也分两种，如果数据量小就广播修改数据；如果数据量大就广播清理通知回写到持久化中。这样虽然可能会有一定的不一致风险，但是如果数据不是那种要求太高的，而且缓存命中率又能比较有保障的话（比如根据KEY来进行一致性哈希访问缓存进程），那么

真正因为写操作广播不及时，导致数据不一致的情况还是会比较少的。这种策略实现起来非常简单，无需一个中心节点进程维护数据租约，也无需复杂的判断逻辑进行同步，只要有广播的能力，加上对于写操作的一些配置，就能实现高效的缓存服务。所以“修改广播”策略是在大多数需要实时同步，但数据一致性要求不高的领域最常见的手段。著名的DNS系统的缓存就是接近这种策略：我们要修改某个域名对应的IP，并不是立刻在全球所有的DNS服务器上生效，而是需要一定时间广播修改给其他服务区。而我们每个DSN服务器，都具备了大量的其他域名的缓存数据。



总结

在高性能的服务器架构中，常用的缓存和分布两种策略，往往是结合到一起使用的。虽然这两种策略，都有无数种不同的表现形式，成为各种各样的技术流派，但是只有清楚的理解这些技术的原理，并且和实际的业务场景结合起来，才能真正的做出满足应用要求的高性能架构。

分享到：
更多2