[C# Hadoop学习笔记（六）—C#的云计算框架借鉴（上）](http://blog.csdn.net/black0707/article/details/12853003)

2013-10-18 14:20 1727人阅读 [评论](http://blog.csdn.net/black0707/article/details/12853003#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/black0707/article/details/12853003#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C# Hadoop云计算 初探与思考（备忘）（9） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

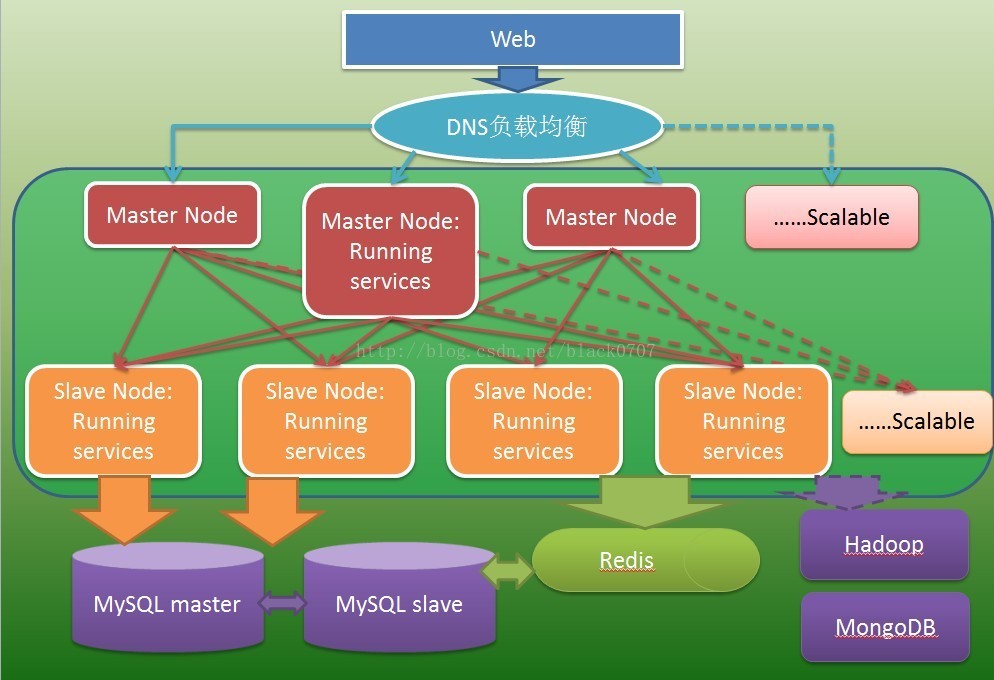
2012年底，末日之后，看到大家都在写年末总结，我也忍不住想一试。工作已经3年半了，头一次写总结。虽然到现在仍是无名小码农一名，但工作这些年，技术着实有不少积累。成长最大的，当然就是这篇文章标题提到的——高性能分布式计算与存储系统的设计和研发过程，这也是我自2010年供职于国内最大的某著名网站之后，和这个系统一起成长，亲眼见证和伴随着它的发展，从一个婴儿一样的"Demo"程序，成长为现在可以处理千万级日PV的强大系统，直到2012年我离开。我也顺势积累了Unix/Linux服务器、多线程、I/O、海量数据处理、注重高性能与效率的C/C++编程等宝贵的码农财富，当然，遗憾和不足，仍然是有许多的。

        2012年，其实是自工作以来，技术积淀最多的一年。因为，在2012年，我终于学会了独立思考，我不再像以前一样，许许多的技术只是需要用到的时候，匆忙的google（有时候还要先匆忙的先翻墙http://www.cnblogs.com/Emoticons/msn/red_smile.gif），我发现，“好记性不如烂笔头”，古训确实毋庸置疑，有大量的、琐碎的技术经验、编程细节、技巧，需要积淀下来，可能单条的细节与技巧，并不会对一个人的职业生涯产生什么影响，但把它们都积聚起来，就会强大许多，很实际的，带来的技术提升，能带来更高的Offer。所以，2012年，我开始到博客园写技术博客，和众多园友分享我对技术的一知半解，共同进步；也终于耐下心，为自己做了一个简单的个人主页，虽然10年前，我就可以做出这样的东西……；我成为了更忠实的苹果粉，所以我尝试去做**[iOS](http://lib.csdn.net/base/ios" \o "iOS知识库" \t "_blank)**创业，虽然这和我的主要工作研究方向并不一致，当我看到自己做的demo在自己iPhone 4s上跑起来，我突然又有一了一种久违的兴奋——那是每一个程序员，都体会过的，小小的成就感；2012年，我开始接触和了解许多以前从来不懂的技术：**[Hadoop](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "_blank)**、GoogleFS、JVM、XCode、ARC……小到如何将vim打造成一个IDE……；2012年，我暂时离开了生活工作了6年的北京，来到了陌生的上海，虽然明年我可能就会回到北京，在上海这个繁华的城市，我又体会到了一种和京城码农感觉一样的，技术氛围和文化；最后必须一提的，2012年，我结婚了，并喜得一龙子，在这篇总结里，衷心的对我老婆说一声：“老婆，谢谢你，我爱你。”

        接下来，该进入这篇文章的正题了， 就是简单地谈谈，我这两年，主要做的东西——高性能分布式计算与存储系统。

        这个系统看名字十分牛比，所涉足的目前互联网最领先的技术领域。具体有什么用途？ 在我之前供职的公司，它主要是作为中间层，给网站页面提供缓存服务的，并且，它对付的难题，是**[大数据](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "_blank)**、海量数据，相信，每一个日PV超过千万级的网站，都必须会有类似的系统存在，如果，你曾经看过，博客园里的《淘宝技术发展》等类似文章，就一定不会对我接来将要提到的许多概念和术语感到陌生。对于这样大流量，需要处理大数据的网站而言，由Web的逻辑直接调用管理数据存储，是非常不科学的，实际上也是不可能的，大数据、高并发的对**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**进行读写，通常数据库都会挂掉，从而使网站也挂掉，必须要在Web和数据库之间，通过技术手段实现一种“转换”或“控制”，或“均衡”或“过渡”，我不知道这样用词是否正确，你只要明白其中的意思就好了。这样的技术手段有许多，所实现的东东也有许多，我们用到的，就是被称为“中间层”的一个逻辑层，在这个层，**将数据库的海量数据抓出来，做成缓存，运行在服务器的内存中，同理，当有新的数据到来，也先做成缓存，再想办法，持久化到数据库中，就是这样简单的思路**，但实现起来，从零到有，可以说难如登天，但是，任何事物，都是在曲折中，不断发展前进的，这是中学我们就学过的哲学理论。这个系统，就被我们称简为“缓存系统”，它最大的好处，就是砍掉了每天上千万次的数据库读写操作，取代而之的，是读取服务器中提供缓存服务的进程所控制的内存，所以你知道，这里面节省了多少的资源申请、竞争、I/O……当然，后面你也会发现，它会带来许多新的问题，最显著的问题，就是数据的同步和一致性，后面我会讲到。

        现在，让我们先看看， 这个系统，发展到我离开它的时候，长什么样子？（由于涉及到商业机密，具体的技术不能提供）



就是这样的一张[**架构**](http://lib.csdn.net/base/architecture)图，代表着可以处理每日上千万PV的系统，涉及到许多的技术，让我们一个部分一个部分解读它。

        首先，从当我有一个web请求到达时，将会发生怎样的事情说起。比如，我是一个用户，我在这个网站登陆，我的“个人”页面上，将会加载许许多多的东西，有许许多多的图片、文字、消息等，我们举其中一个例子，我将要得到我的好友列表——friend list。通过常识可以知道，这个friend list，不是随机的、临时的，而肯定是一个（一组）持久化存储于数据库里的数据，我们就是一个用户请求得到他的friend list说起，来解读这张架构图。如果我的网站流量很小，每天不超过10万PV，峰值可能就几百个上千个用户，同时请求他们的friend list，那么，现今任何一种语言配上任何一种数据库的搭配，只要稍做处理，都可以很好的完成这个工作——从数据库中，读出该用户的friend list，然后访回给web，如果用户对好友列表作了任何修改，web马上将修改内容写入数据库，形成新的friend list。然而，当访问流量持续提升，达到千万级、甚至亿级PV的时候，刚才说的方法就不可行了。因为，同时可能有几十万甚至上百万用户，通过web请求从数据库中读（如果写将会更糟糕）上百条万数据，数据库将不堪重负，形成巨大的延迟甚至挂掉。通过上面的系统，来解决这样的问题。

        现在，我们要设计和研发的上述系统，当一个web页面提交一个获取friend list的请求后，它首先将根据一定的规则，通过负载均衡，然后到达相应的master节点。上面我们提到的是DNS负载均衡，这得众多负载均衡技术中的一种方法。也就是说，我有许许多多的master节点（上图的scalabe表明，我是可扩展的，只要有条件，可随意横向扩展节点，以提高速度、容灾、容量等指标），每个master节点的IP地址（域名）当然不一样，通过DNS负载均衡，合理地把该请求，送到相对“空闲”的master节点服务器。现在解释一下master节点服务器和slave节点服务器的功能：slave节点，主要用于"Running services"，即，实际处理请求的缓存服务进程，通常运行在slave节点上；master节点，主要用于分发通过负载均衡的请求（当然，master节点上也可以运行一些“缓存服务进程”，即并发流量不高、较辅助的一些服务），找到用于处理实际请求的合适的slave节点，将该请求交给它处理，再次实现了一道“负载均衡”，同时，需要分布式计算的内容，将可能同时分发到几个slave节点，之后再对结果进行合并返回（Map-Reduce原理）。

        好了，现在我们已经知道，一个friend list请求已经通过DNS负载均衡、通过master节点进行分配，到达了相应的slave节点上。我们还知道，所说的“缓存” ，正是slave节点中所运行的services进程中所管理的内存，提供同样功能的service可能会有很多份，同时运行在不同slave节点上，以提供高并发和分布式计算的功能。例如，获得friend list就是这样的service，因为这个功能太常用了，所以，在我们的系统中，这样的服务可能同时提供5份、10份甚至更多，那么我这个获取friend list的请求，究竟被分配到哪个slave节点上的service处理呢？这正是刚才提到的master节点来完成这一工作。再比如，我现在需要获取“二度关系”的列表（关于六度人脉理论，可google），所谓“二度关系”，就是好友的好友，那么我要取这样的列表，即friend's every friend list，这样的请求，将会把取每个friend list分配（Map）到不同slave节点上去做（根据一定的规则），然后再进行合并（Reduce）（当然，熟悉**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**的同学可能已经发现，这样去获取请求，非常的笨拙，有没有更好的方法呢？当然有！因为好友的好友，其实就是好友的friend list与我和好友的共同好友common friend list的“差集”，对吗？http://www.cnblogs.com/Emoticons/msn/teeth_smile.gif，所以我不用去取好友的每个好友的friend list，而只用取2次就可以通过计算完成请求，这又节省了多少资源呢？假如我有100个好友，1000个，10000万个？会节省多少次计算呢？这也证明，一个良好的算法，对改善程序性能，有多么大的帮助！）

        好，我们继续。现在，我的获取friend list的请求，已经在被某个slave节点中的负责这一功能的service进程处理，它将根据一定规则，给出两种可能的处理方式：

        1、 我这个用户非常活跃，经常登陆网站（一定的规则，认为缓存未到过期时间），且我这个slave节点自上次“重建缓存”（即重新从数据库中读取数据，建立缓存，后面会谈）后，没有发生过down机重启行为（又一定的规则），我也没有收到过master节点发送过来要求更新缓存（即从数据库中比较数据并更新）的Notification（通知），或是在一定条件下我这个slave节点对它掌握的缓存数据版本（版本管理系统原理，思考一下svn的工作原理）和数据库进行了一次比较（注意，比较数据版本可认为只是一个int值，且是原子操作，这和比较整条数据是否一致在性能上有天壤之别）发现是最新的数据版本，那么，我这个slave节点**将直接返回缓存数据，而没有任何数据库读操作**，也就是说，我这一次获取friend list的请求，得到的是缓存数据，当然，这个缓存数据肯定是最新的、正确的、和数据库中的持久化数据是一致的，后面会提到怎样来尽量保证这一点；

        2、第1点中的“一定规则”不满足时，即我这个slave节点的缓存和数据库中的数据可能存在不一致的没有其它办法，我必须从数据库中读取数据，更新缓存，然后再返回。但同时注意，slave节点中的service服务进程，将认为此用户现在活跃，可能还会请求一些相关、类似的数据（如马上可能进行添加好友、删除好友等操作），所以去数据库读取数据的时候，将不会只读friend list，可能与用户有关的其它一部分数据，会被同时读取并更新缓存，如果负责这一部分数据的缓存服务并不是当前的service进程，或在其它slave节点，或同时还有几份service进程在工作，那么slave节点将提交“更新缓存”请求给master节点，通过master节点发出Notification给相关slave节点的相关service进程，从而，**尽可能使每一次读取数据库的作用最大化，而如果稍后用户果然进行了我们猜测的行为（可认为cache命中），结果将同第1点，直接通过缓存返回数据而且保证了数据的正确和一致性。**

        好了，刚刚提到的都是“读操作”，相比“写操作”， 其数据一致性更容易保证，之后我们将讲述“写操作”的工作原理。现在，让我们先跳过这一部分，继续看架构图。slave节点之后，就是实际的数据存储了，使用了**[MySQL](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**、**[Redis](http://lib.csdn.net/base/redis" \o "Redis知识库" \t "_blank)**，**[mysql](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**主从之间的协同是DBA的工作，不在此篇讨论，**[redis](http://lib.csdn.net/base/redis" \o "Redis知识库" \t "_blank)**主要存储K-V键值对数据，比如用户id和用户昵称，是最常用的K-V对之一，通过Redis进行存储，再结合上述的工作过程，可保证这个系统的高性能。而架构图最右下角的**[hadoop](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "_blank)**与**[MongoDB](http://lib.csdn.net/base/mongodb" \o "MongoDB知识库" \t "_blank)**，是可选的MySQL替代方案，其实，正是未来的主要发展方向。如果slave节点中的service服务进程与Hadoop良好结合，系统的性能将更上一层楼。顺便说一句，master、slave节点都是由C++开发的。[Why C++？可参考酷壳上的一篇文章《C++ Performance per $》](http://coolshell.cn/articles/6548.html" \t "_blank)。

        好了，上篇就讲到这里，余下的问题，我们在下篇进行讨论，谢谢大家。

[C# Hadoop学习笔记（七）—C#的云计算框架借鉴（下）](http://blog.csdn.net/black0707/article/details/12853049)

2013-10-18 14:24 1739人阅读 [评论](http://blog.csdn.net/black0707/article/details/12853049#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/black0707/article/details/12853049#report)

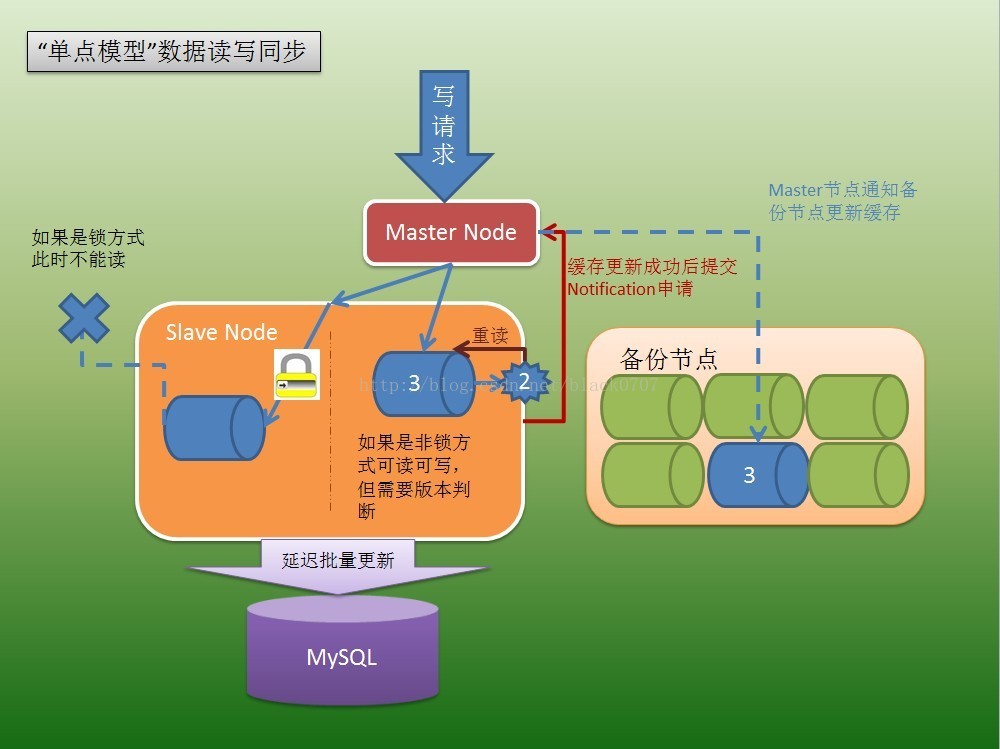
http://static.blog.csdn.net/images/category_icon.jpg 分类：

C# Hadoop云计算 初探与思考（备忘）（9） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

在上篇里，我们主要讨论了，这个系统怎样处理大数据的“读”操作，当然还有一些细节没有讲述。下篇，我们将主要讲述，“写”操作是如何被处理的。我们都知道，如果只有“读”，那几乎是不用做任何数据同步的，也不会有并发安全问题，之所以，会产生这样那样的问题，会导致缓存和数据库的数据不一致，其实根源就在于“写”操作的存在。下面，让我们看一看，当系统需要写一条数据的时候，又会发生怎样的事情？

       同样，我们还是以friend list为例。现在，我登陆了这个网站，获取了friend list之后，我添加了一个好友，那么，我的friend list必定要做修改和更新（当然，添加好友这一个动作肯定不会只有修改更新friend list这一个请求，但我们以此为例，其它请求也是类似处理），那么，这个要求修改和更新friend list的请求，和获取friend list请求类似，在被slave节点中的服务进程处理之前，也是先通过DNS负载均衡，被分配到合适的master节点，再由master节点，分配到合适的相对空闲的负责这一功能的slave点上。现在假设，前面我们已经讲过，获取friend list这样的请求，非常常用，所以，提供这一供能的服务进程将会有多份，比如，有10份，服务进程编号为0~9，同时运行在10个（也可能仅运行在1个~9个slave节点上！）slave节点上，具体分配请求的时候，选择哪一个slave节点和哪一份服务进程呢？这当然有许多种规则去影响分配策略，我们就举一个最简单的例子，采用用户id对10取模，得到0~9的结果，即是所选择的服务进程编号，假设我的用户id尾号为9，那么我这个请求，只会被分配到编号为9的服务进程去处理（当然，所有用户id尾号为9的都是如此），编号为9的服务进程，也只负责为**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**中用户id尾号为9的那些数据做缓存，而用户id尾号为0~8的缓存则由其它服务进程来处理。如果所需的请求是以刚才这种方式工作的，那么现在我要求修改和更新friend list的这个请求，将只会被分配到服务进程编号为9的进程来处理，我们称之为“单点模型”（也就是说，同一条数据只会有一份可用缓存，备份节点上的的不算），你可能已经猜到了，还会有“多点模型”——即同时有好几个服务进程都会负责同样的缓存数据，这是更复杂的情况，我们稍后再讨论。



现在，我们接着说“单点模型” 。这个修改和更新friend list的请求到了编号为9的服务进程中后，如何被处理呢？缓存肯定先要被处理，之后才考虑缓存去和数据库同步一致，这大家都知道（否则还要这个系统干嘛？）大家还知道，只要涉及到并发的读写，就肯定存在并发冲突和安全问题，这又如何解决呢？我们有两种方式，来进行读写同步。

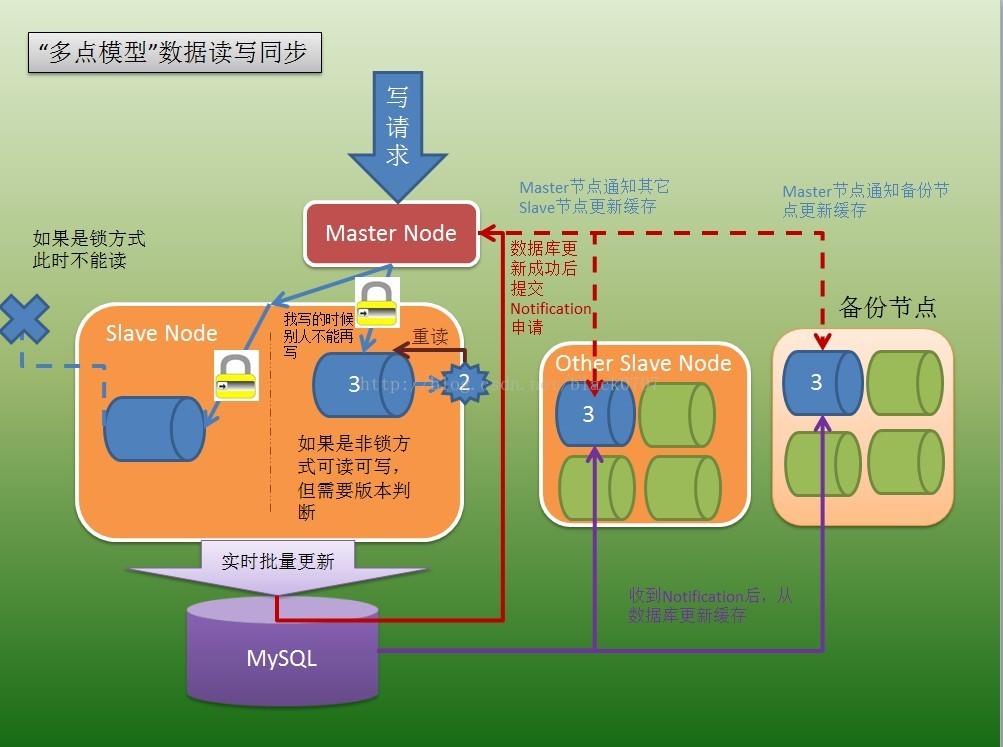
1、 第一种方式，就是传统的，加锁方式——通过加锁，可以有效地保证缓存中数据的同步和正确，但缺点也非常明显，当服务进程中同时存在读写操作的线程时，将会存在严重的锁竞争，进而重新形成性能瓶颈。好在，通常使用这种方式处理的业务需求，都经过上述的一些负载均衡、分流措施之后，锁的粒度不会太大，还是上述例子，我最多也就锁住了所有用户id尾号为9的这部分缓存数据更新，其它90%的用户则不受影响。再具体些，锁住的缓存数据可以更小，甚至仅锁住我这个用户的缓存数据，那么，锁产生的性能瓶颈影响就会更小了（为什么锁的粒度不可能小到总是直接锁住每个用户的缓存数据呢？答案很简单，你不可能有那么多的锁同时在工作，数据库也不可能为每个用户建一张表），即**锁的粒度是需要平衡和调整的**。好，现在继续，我要求修改和更新friend list的请求，已经被服务进程中的写进程在处理，它将会申请获得对这部分缓存数据的锁，然后进行写操作，之后释放锁，传统的锁工作流程。在这期间，读操作将被阻塞等待，可想而知，如果锁的粒度很大，将有多少读操作处于阻塞等待状态，那么该系统的高性能就无从谈起了。

2、有没有更好的方法呢？当然有，这就是无锁的工作方式。首先，我们的网站，是一个读操作远大于写操作的网站（如果需求相反，可能处理的方式也就相反了），也就是说，大多数时候，读操作不应该被写操作阻塞，应优先保证读操作，如果产生了写操作，再想办法使读操作“更新”一次，进而使得读写同步。这样的工作方式，其实很像版本管理工具，如svn的工作原理：即，每个人，都可以读，不会因为有人在进行写，使得读被阻塞；当我读到数据后，由于有人写，可能已经不是最新的数据了，svn在你尝试提交写的时候，进行判断，如果版本不一致，则重新读，合并，再写。我们的系统也是按类似的方式工作的：即每个线程，都可以读，但读之前先比较一下版本号，然后读缓存数据，读完之后准备返回给Web时，再次比较版本号，如果发现版本已经被更新（当然你读的数据顶多是“老”数据，但不至于是错误的数据，Why？还是参考svn，这是"Copy and Write"原理，即我写的那一份数据，是copy出来写的，写完再copy回去，不会在你读出的那一份上写），则必须重新读，直到读到的缓存数据版本号是最新的。前面已经说过，比较和更新版本号，可认为是原子操作（比如，利用CAS操作可以很好的完成这一点，关于CAS操作，可以google到一大堆东西），所以，整个处理流程就实现了无锁化，这样，在**[大数据](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "_blank)**高并发的时候，没有锁瓶颈产生。然而，你可能已经发现其中的一些问题，最显著的问题，就是可能多读不止一次数据，如果读的数据较多较大，又要产生性能瓶颈了（苦！没有办法），并且可能产生延迟，造成差的用户体验。那么，又如何来解决这些问题呢？其实，我们是根据实际的业务需求来做权衡的，如果，所要求的请求，允许一定的延迟存在，实时性要求不是最高，比如，我看我好友发的动态，这样的缓存数据，并不要求实时性非常高，稍稍有延迟是允许的，你可以想象一下，如果你的好友发了一个状态，你完全没有必要，其实也不可能在他点击“发布”之后，你的动态就得到了更新，其实只要在一小段时间内（比如10秒？）你的动态更新了，看到了他新发布了状态，就足够了。假设是这样的请求，且如果我采用第1种加锁的方式所产生的性能瓶颈更大，那么，将**采用这种无锁的工作方式，即当读写有冲突时候，读操作重新读所产生的开销或延迟，是可以忍受的**。比较幸运的是，同时有多个读写线程操作同一条缓存数据导致多次的重读行为，其实并不是总是发生，也就是说，我们系统的大数据并发，主要在多个进程线程同时读不同条的数据这一业务需求上，这也很容易理解，每个用户登陆，都是读他们各自的friend list（不同条数据，且在不同的slave节点上），只不过，这些请求是并发的（如果不进行分布式处理会冲垮服务器或数据库），但是并不总是会，许多用户都要同时读某一条friend list同时我还在更新该条friend list导致多次无效的重读行为。

       我们继续上面的friend list。现在，我的friend list已经在缓存中被修改和更新了。无论是采用方式1还是方式2进行，在这期间，如果恰好有其它线程来读我的friend list，那么总之会受到影响，如果是方式1，该请求将等待写完毕；而如果是方式2，该请求将读2次（也可能更多，但实在不常见）。这样的处理方式，应该不是最好的，但前面已经说过了，我们的系统，主要解决：**大流量高并发地读写多条数据，而不是一条。**接下来，该考虑和数据库同步的事情了。

       恩，刚才说了那么多，你有没有发现，经过我修改和更新friend list后，缓存中的数据和数据库不一致了呢？显然，数据库中的数据，已经过期了，需要对其更新。现在，slave节点中的编号为9的服务进程，更新完了自己的缓存数据后（修改更新我的friend list），将“尝试”向数据库更新。注意，用词“尝试”表明该请求不一定会被马上得到满足。其实，服务进程对数据库的更新，是批量进行的，可认为是一个TaskContainer（任务容器），每间隔一段时间，或得到一定的任务数量，则成批地向数据库进行更新操作，而不是每过来一个请求，更新缓存后就更新一次数据库（你现在知道了这样做又节省了多少次数据库操作！）。那么，为什么可以这样做呢？因为，我们已经有了缓存，缓存就是我们的保障，在“单点模型”下，缓存更新后，任何读缓存的操作，都只会读到该缓存，不需要经过数据库，参看上篇中提到过此问题。所以，数据库的写更新操作，可以“聚集”，可以一定延迟之后，再进行处理。你会发现，既然如此，我就可以对这些操作进行合并、优化，比如，两个写请求都是操作同一张表，那么可以合并成一条，没错，这其实已经涉及到SQL优化的领域了。当然，你也会发现，现在缓存中的新数据还没有进行持久化，如果在这个时间点，slave节点机器down掉了，那么，这部分数据就丢失了！所以，这个延迟时间并不会太长，通常10秒已经足够了。即，每10秒，整理一下我这个服务进程中已经更新缓存未更新DB的请求，然后统一处理，如果更杞人忧天（虽然考虑数据安全性决不能说是杞人忧天，但你要明白，其实任何实时服务器发生down行为总是会有数据丢失的，只是或多或少），则延迟间隔可以更短一些，则DB压力更大一些，再次需要进行实际的考量和权衡。至此，我的friend list修改和更新请求，就全部完成了，虽然，可能在几十秒之前，就已经在页面上看到了变化（通过缓存返回的数据）。

       那么，读和写都已经讲述了，还有其它问题吗？问题还不少。刚才讨论的，都是“单点模型”。



即，每一条数据库中的数据，都只有一份缓存数据与之对应。然而，实际上，“多点模型”是必须存在的，而且是更强大的处理方式，也带来同步和一致性的更多难题，即每一条数据，可能有多份缓存与之对应。即多个slave节点上的服务进程中，都有一份对应DB中相同数据的缓存，这个时候，又将如何同步呢？我们解决的方式，叫做“最终一致性”原则，关于最终一致性模型，又可以google到一大堆，特别要提出的是GoogleFS的多点一致性同步，就是通过“最终一致性”来解决的，通俗的讲，就是**同一条数据，同一时刻，只能被一个节点修改。**假设，我现在的业务，是“多点模型”，比如，我的friend list，是多点模型，有多份缓存（虽然实际并不是这样的），那么，我对friend list的修改和更新，将只会修改我被分配到的slave节点服务进程中的缓存，其它服务进程或slave节点的缓存，以及数据库，将必须被同步更新，这是如何做到的呢？这又要用到上篇曾提到的Notification（通知服务），这个模块虽然没有在**[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "_blank)**图中出现，却是这个系统中最核心的一种服务（当然，它也是多份的，呵呵），即，当一条数据是多点模型时，当某一个服务进程对其进行修改和更新后，将通过向master节点提交Notificaion并通知其它服务进程或其它slave节点，告知他们的缓存已经过期，需要进行更新，这个更新，可能由所进行修改更新的服务进程，发送缓存数据给其它进程或节点，也由可能等待DB更新之后，由其它节点从DB进行更新，从而间接保证多点一致性。等等，刚才不是说，通常10秒才批量更新DB吗？那是因为在单点模型下，这样做是合理的，但在多点模型下，虽然也是批理对数据库进行更新，但这样的延迟通常非常小，可认为即时对数据库进行批量更新，然后，通过Notification通知所有有这一条数据的节点，更新他们的缓存。由此可见，多点模型，所可能产生的问题是不少的。那么，为什么要用多点模型呢？假设我有这样的业务：大数据高并发的读某一条数据，非常非常多的读，但写很少，比如一张XX门的热门图片，有很多很多的请求来自不同的用户都需要这个条数据的缓存，多点模型即是完美的选择。我许多slave节点上都有它的缓存，而很少更新，则可最大限度的享用到多点模型带来的性能提升。

       还有一些问题，不得不说一下。就是down机和定期缓存更新的问题。先说宕机，很显然，缓存是slave节点中的服务进程的内存，一旦节点宕机，缓存就丢失了，这时就需要前面我提到过的“重建缓存”，这通常是由master节点发出的，master节点负责监控各个slave节点（当然也可以是其它master节点）的运行状况，如果发现某个slave节点宕机（没有了“心跳”，如果你了解一些[**Hadoop**](http://lib.csdn.net/base/hadoop)，你会发现它也是这样工作的），则在slave节点重新运行之后（可能进行了重启），master节点将通知该slave节点，重建其所负责的数据的缓存，从哪重建，当然是从数据库了，这需要一定的时间（在我们拥有百万用户之后，重建一个slave节点所负责的数据的缓存通常需要几分钟），那么，从宕机到slave节点重建缓存完毕这一段时间，服务由谁提供呢？显然备份节点就出马了。其实在单点模型下，如果考虑了备份节点，则其实所有的请求都是多点模型。只不过备份节点并不是总是会更新它的缓存，而是定期，或收到Notification时，才会进行更新。master节点在发现某个slave节点宕机后，可以马上指向含有同样数据的备份节点，保证缓存服务不中断。那么，备份节点的缓存数据是否是最新的呢？有可能不是。虽然，通常每次对数据库完成批量更新后，都会通知备份节点，去更新这些缓存，但还是有可能存在不一致的情况。所以，备份节点的工作方式，是特别的，即对于每次请求的缓存都采用Pull（拉）方式，如何Pull？前面提到的版本管理系统再次出马，即每次读之前，先比较版本，再读，写也是一样的。所以，备份节点的性能，并不会很高，而且，通常需要同时负责几个slave节点的数据的备份，所以，存在被冲垮的可能性，还需要slave节点尽快恢复，然后把服务工作重新还给它。

       再说定期缓存更新的问题。通常，所有的slave节点，都会被部署在夜深人静的某个时候（如02:00~06:00），用户很少的时候，定期进行缓存更新，以尽可能保证数据的同步和一致性，且第二天上午，大量请求到达时，基本都能从缓存返回最新数据。而备份节点，则可能每30分钟，就进行一次缓存更新。咦？前面你不是说，备份节点上每次读都要Pull，比较版本并更新缓存，才会返回吗？是的，那为什么还要定期更新呢？答案非常简单，因为如果大部分缓存都是最新的数据，只比较版本而没有实际的更新操作，所消耗的性能很小很小，所以定期更新，在发生slave节点宕机转由备份节点工作的时候，有很大的帮助。

       最后，再说一下Push（推送）方式，即，每次有数据改动，都强制去更新所有缓存。这种方式很消耗性能，但更能保证实时性。而通常我们使用的，都是Pull（拉）方式，即无论是定期更新缓存，还是收到Notification（虽然收通知是被“推”了一把）后更新缓存，其实都是拉，把新的数据拉过来，就好了。在实际的系统中，两种方式都有，还是那句话，看需求，再决定处理方式。

       好了，终于写完了这篇总结，看到上篇发布后，得到了许许多多园友的鼓励和支持，在此一并感谢！相信也有不少园友，已经看到了这个系统的许多不足和瓶颈，确实，它并不是一个完美的系统，还需要不断进化。我写出这篇文章，也是希望和大家多多交流，共同进步。马上就是2013年了，希望自己能有更好的发展，也希望所有的朋友，都能更上一层楼！