高并发服务端分布式系统设计概要

2015/12/26 · [IT技术](http://blog.jobbole.com/category/it-tech/) · [2 评论](http://blog.jobbole.com/96074/#article-comment)· [分布式](http://blog.jobbole.com/tag/%e5%88%86%e5%b8%83%e5%bc%8f/), [架构](http://blog.jobbole.com/tag/architecture/)

分享到：**14**

原文出处： [张峻崇](http://www.cnblogs.com/ccdev/p/3338412.html)

写这篇文章的目的，主要是把今年以来学习的一些东西积淀下来，同时作为之前文章《[高性能分布式计算与存储系统设计概要](http://blog.jobbole.com/95932/)》的补充与提升，然而本人水平非常有限，回头看之前写的文章也有许多不足，甚至是错误，希望同学们看到了错误多多见谅，更欢迎与我讨论并指正。

我大概是从2010年底起开始进入高并发、高性能服务器和分布式这一块领域的研究，到现在也差不多有三年，但其实很多东西仍然是一知半解，我所提到的许许多多概念，也许任何一个我都不能讲的很清楚，还需要继续钻研。但我们平时在工作和学习中，多半也只能从这种一知半解开始，慢慢琢磨，不断改进。

好了，下面开始说我们今天要设计的系统。

这个系统的目标很明确，针对千万级以上PV的网站，设计一套用于后台的高并发的分布式处理系统。这套系统包含业务逻辑的处理、各种计算、存储、日志、备份等方面内容，可用于类微博，SNS，广告推送，邮件等有大量线上并发请求的场景。

如何抗大流量高并发？（不要告诉我把服务器买的再好一点）说起来很简单，就是“分”，如何“分”，简单的说就是把不同的业务分拆到不同的服务器上去跑（垂直拆分），相同的业务压力分拆到不同的服务器去跑（水平拆分），并时刻不要忘记备份、扩展、意外处理等讨厌的问题。说起来都比较简单，但设计和实现起来，就会比较困难。以前我的文章，都是“从整到零”的方式来设计一个系统，这次咱们就反着顺序来。

那我们首先来看，我们的数据应该如何存储和取用。根据我们之前确定的“分”的方法，先确定以下2点：

（1）我们的分布式系统，按不同的业务，存储不同的数据；（2）同样的业务，同一个数据应存储多份，其中有的存储提供读写，而有的存储只提供读。

好，先解释下这2点。对于（1）应该容易理解，比如说，我这套系统用于微博（就假想我们做一个山寨的推特吧，给他个命名就叫“山推” 好了，以下都叫山推，Stwi），那么，“我关注的人”这一个业务的数据，肯定和“我发了的推文”这个业务的数据是分开存储的，那么我们现在把，每一个业务所负责的数据的存储，称为一个group。即以group的方式，来负责各个业务的数据的存储。接下来说（2），现在我们已经知道，数据按业务拆到group里面去存取，那么一个group里面又应该有哪些角色呢？自然的，应该有一台主要的机器，作为group的核心，我们称它为Group Master，是的，它就是这个group的主要代表。这个group的数据，在Group Master上应该都能找到，进行读写。另外，我们还需要一些辅助角色，我们称它们为Group Slaves，这些slave机器做啥工作呢？它们负责去Group Master处拿数据，并尽量保持和它同步，并提供读服务。请注意我的用词，“尽量”，稍后将会解释。现在我们已经有了一个group的基本轮廓：

一个group提供对外的接口（废话否则怎么存取数据），group的底层可以是实际的File System，甚至是HDFS。Group Master和Group Slave可以共享同一个File System（用于不能丢数据的强一致性系统），也可以分别指向不同的File System（用于弱一致性，允许停写服务和系统宕机时丢数据的系统），但总之应认为这个”File System”是无状态，有状态的是Group Master和各个Group Slave。

下面来说一个group如何工作，同步等核心问题。首先，一个group的Group Master和Group Slave间应保持强一致性还是弱一致性（最终一致性）应取决于具体的业务需求，以我们的“山推”来说，Group Master和Group Slave并不要求保持强一致性，而弱一致性（最终一致性）即能满足要求，为什么？因为对于“山推”来讲，一个Group Master写了一个数据，而另一个Group Slave被读到一个“过期”（因为Group Master已经写，但此Group Slave还未更新此数据）的数据通常并不会带来大问题，比如，我在“山推”上发了一个推文，“关注我的人”并没有即时同步地看到我的最新推文，并没有太大影响，只要“稍后”它们能看到最新的数据即可，这就是所谓的最终一致性。但当Group Master挂掉时，写服务将中断一小段时间由其它Group Slave来顶替，稍后还要再讲这个问题。假如我们要做的系统不是山推，而是淘宝购物车，支付宝一类的，那么弱一致性（最终一致性）则很难满足要求，同时写服务挂掉也是不能忍受的，对于这样的系统，应保证“强一致性”，保证不能丢失任何数据。

接下来还是以我们的“山推“为例，看看一个group如何完成数据同步。假设，现在我有一个请求要写一个数据，由于只有Group Master能写，那么Group Master将接受这个写请求，并加入写的队列，然后Group Master将通知所有Group Slave来更新这个数据，之后这个数据才真正被写入File System。那么现在就有一个问题，是否应等所有Group Slave都更新了这个数据，才算写成功了呢？这里涉及一些NWR的概念，我们作一个取舍，即至少有一个Group Slave同步成功，才能返回写请求的成功。这是为什么呢？因为假如这时候Group Master突然挂掉了，那么我们至少可以找到一台Group Slave保持和Group Master完全同步的数据并顶替它继续工作，剩下的、其它的Group Slave将“异步”地更新这个新数据，很显然，假如现在有多个读请求过来并到达不同的Group Slave节点，它们很可能读到不一样的数据，但最终这些数据会一致，如前所述。我们做的这种取舍，叫“半同步”模式。那之前所说的强一致性系统应如何工作呢？很显然，必须得等所有Group Slave都同步完成才能返回写成功，这样Group Master挂了，没事，其它Group Slave顶上就行，不会丢失数据，但是付出的代价就是，等待同步的时间。假如我们的group是跨机房、跨地区分布的，那么等待所有Group Slave同步完成将是很大的性能挑战。所以综合考虑，除了对某些特别的系统，采用“最终一致性”和“半同步”工作的系统，是符合高并发线上应用需求的。而且，还有一个非常重要的原因，就是通常线上的请求都是读>>写，这也正是“最终一致性”符合的应用场景。

好，继续。刚才我们曾提到，如果Group Master宕机挂掉，至少可以找到一个和它保持同不的Group Slave来顶替它继续工作，其它的Group Slave则“尽量”保持和Group Master同步，如前文所述。那么这是如何做到的呢？这里涉及到“分布式选举”的概念，如Paxos协议，通过分布式选举，总能找到一个最接近Group Master的Group Slave，来顶替它，从而保证系统的可持续工作。当然，在此过程中，对于最终一致性系统，仍然会有一小段时间的写服务中断。现在继续假设，我们的“山推”已经有了一些规模，而负责“山推”推文的这个group也有了五台机器，并跨机房，跨地区分布，按照上述设计，无论哪个机房断电或机器故障，都不会影响这个group的正常工作，只是会有一些小的影响而已。

那么对于这个group，还剩2个问题，一是如何知道Group Master挂掉了呢？二是在图中我们已经看到Group Slave是可扩展的，那么新加入的Group Slave应如何去“偷”数据从而逐渐和其它节点同步呢？对于问题一，我们的方案是这样的，另外提供一个类似“心跳”的服务（由谁提供呢，后面我们将讲到的Global Master将派上用场），group内所有节点无论是Group Master还是Group Slave都不停地向这个“心跳”服务去申请一个证书，或认为是一把锁，并且这个锁是有时间的，会过期。“心跳”服务定期检查Group Master的锁和其有效性，一旦过期，如果Group Master工作正常，它将锁延期并继续工作，否则说明Group Master挂掉，由其它Group Slave竞争得到此锁（分布式选举），从而变成新的Group Master。对于问题二，则很简单，新加入的Group Slave不断地“偷”老数据，而新数据总由于Group Master通知其更新，最终与其它所有结点同步。（当然，“偷”数据所用的时间并不乐观，通常在小时级别）

我们完成了在此分布式系统中，一个group的设计。那么接下来，我们设计系统的其他部分。如前文所述，我们的业务及其数据以group为单位，显然在此系统中将存在many many的groups（别告诉我你的网站总共有一个业务，像我们的“山推”，那业务是一堆一堆地），那么由谁来管理这些groups呢？由Web过来的请求，又将如何到达指定的group，并由该group处理它的请求呢？这就是我们要讨论的问题。

我们引入了一个新的角色——Global Master，顾名思义，它是管理全局的一个节点，它主要完成如下工作：（1）管理系统全局配置，发送全局控制信息；（2）监控各个group的工作状态，提供心跳服务，若发现宕机，通知该group发起分布式选举产生新的Group Master；（3）处理Client端首次到达的请求，找出负责处理该请求的group并将此group的信息（location）返回，则来自同一个前端请求源的该类业务请求自第二次起不需要再向Global Master查询group信息（缓存机制）；（4）保持和Global Slave的强一致性同步，保持自身健康状态并向全局的“心跳”服务验证自身的状态。

现在我们结合图来逐条解释上述工作，显然，这个系统的完整轮廓已经初现。

首先要明确，不管我们的系统如何“分布式”，总之会有至少一个最主要的节点，术语可称为primary node，如图所示，我们的系统中，这个节点叫Global Master，也许读过GFS + Bigtable论文的同学知道，在GFS + Bigtable里，这样的节点叫Config Master，虽然名称不一样，但所做的事情却差不多。这个主要的Global Master可认为是系统状态健康的标志之一，只要它在正常工作，那么基本可以保证整个系统的状态是基本正常的（什么？group或其他结点会不正常不工作？前面已经说过，group内会通过“分布式选举”来保证自己组内的正常工作状态，不要告诉我group内所有机器都挂掉了，那个概率我想要忽略它），假如Global Master不正常了，挂掉了，怎么办？显然，图中的Global Slave就派上用场了，在我们设计的这个“山推”系统中，至少有一个Global Slave，和Global Master保持“强一致性”的完全同步，当然，如果有不止一个Global Slave，它们也都和Global Master保持强一致性完全同步，这样有个好处，假如Global Master挂掉，不用停写服务，不用进行分布式选举，更不会读服务，随便找一个Global Slave顶替Global Master工作即可。这就是强一致性最大的好处。那么有的同学就会问，为什么我们之前的group，不能这么搞，非要搞什么最终一致性，搞什么分布式选举（Paxos协议属于既难理解又难实现的坑爹一族）呢？我告诉你，还是压力，压力。我们的系统是面向日均千万级PV以上的网站（“山推”嘛，推特是亿级PV，我们千万级也不过分吧），但系统的压力主要在哪呢？细心的同学就会发现，系统的压力并不在Global Master，更不会在Global Slave，因为他们根本不提供数据的读写服务！是的，系统的压力正是在各个group，所以group的设计才是最关键的。同时，细心的同学也发现了，由于Global Master存放的是各个group的信息和状态，而不是用户存取的数据，所以它更新较少，也不能认为读>>写，这是不成立的，所以，Global Slave和Global Master保持强一致性完全同步，正是最好的选择。所以我们的系统，一台Global Master和一台Global Slave，暂时可以满足需求了。

好，我们继续。现在已经了解Global Master的大概用途，那么，一个来自Client端的请求，如何到达真正的业务group去呢？在这里，Global Master将提供“首次查询”服务，即，新请求首次请求指定的group时，通过Global Master获得相应的group的信息，以后，Client将使用该信息直接尝试访问对应的group并提交请求，如果group信息已过期或是不正确，group将拒绝处理该请求并让Client重新向Global Master请求新的group信息。显然，我们的系统要求Client端缓存group的信息，避免多次重复地向Global Master查询group信息。这里其实又挖了许多烂坑等着我们去跳，首先，这样的工作模式满足基本的Ddos攻击条件，这得通过其他安全性措施来解决，避免group总是收到不正确的Client请求而拒绝为其服务；其次，当出现大量“首次”访问时，Global Master尽管只提供查询group信息的读服务，仍有可能不堪重负而挂掉，所以，这里仍有很大的优化空间，比较容易想到的就是采用DNS负载均衡，因为Global Master和其Global Slave保持完全同步，所以DNS负载均衡可以有效地解决“首次”查询时Global Master的压力问题；再者，这个工作模式要求Client端缓存由Global Master查询得到的group的信息，万一Client不缓存怎么办？呵呵，不用担心，Client端的API也是由我们设计的，之后才面向Web前端。

之后要说的，就是图中的“Global Heartbeat”，这又是个什么东西呢？可认为这是一个管理Global Master和Global Slave的节点，Global Master和各个Global Slave都不停向Global Heartbeat竞争成为Global Master，如果Global Master正常工作，定期更新其状态并延期其获得的锁，否则由Global Slave替换之，原理和group内的“心跳”一样，但不同的是，此处Global Master和Global Slave是强一致性的完全同步，不需要分布式选举。有同学可能又要问了，假如Global Heartbeat挂掉了呢？我只能告诉你，这个很不常见，因为它没有任何压力，而且挂掉了必须人工干预才能修复。在GFS + Bigtable里，这个Global Heartbeat叫做Lock Service。

现在接着设计我们的“山推”系统。有了前面两篇的铺垫，我们的系统现在已经有了五脏六腑，剩下的工作就是要让其羽翼丰满。那么，是时候，放出我们的“山推”系统全貌了：

前面啰嗦了半天，也许不少同学看的不明不白，好了，现在开始看图说话环节：

（1）整个系统由N台机器组合而成，其中Global Master一台，Global Slave一台到多台，两者之间保持强一致性并完全同步，可由Global Slave随时顶替Global Master工作，它们被Global Heartbeat（一台）来管理，保证有一个Global Master正常工作；Global Heartbeat由于无压力，通常认为其不能挂掉，如果它挂掉了，则必须人工干预才能恢复正常；

（2）整个系统由多个groups合成，每一个group负责相应业务的数据的存取，它们是数据节点，是真正抗压力的地方，每一个group由一个Group Master和一个到多个Group Slave构成，Group Master作为该group的主节点，提供读和写，而Group Slave则只提供读服务且保证这些Group Slave节点中，至少有一个和Group Master保持完全同步，剩余的Group Slave和Group Master能够达到最终一致，它们之间以“半同步”模式工作保证最终一致性；

（3）每一个group的健康状态由Global Master来管理，Global Master向group发送管理信息，并保证有一个Group Master正常工作，若Group Master宕机，在该group内通过分布式选举产生新的Group Master顶替原来宕机的机器继续工作，但仍然有一小段时间需要中断写服务来切换新的Group Master；

（4）每一个group的底层是实际的存储系统，File system，它们是无状态的，即，由分布式选举产生的Group Master可以在原来的File system上继续工作；

（5）Client的上端可认为是Web请求，Client在“首次”进行数据读写时，向Global Master查询相应的group信息，并将其缓存，后续将直接与相应的group进行通信；为避免大量“首次”查询冲垮Global Master，在Client与Global Master之间增加DNS负载均衡，可由Global Slave分担部分查询工作；

（6）当Client已经拥有足够的group信息时，它将直接与group通信进行工作，从而真正的压力和流量由各个group分担，并处理完成需要的工作。

好了，现在我们的“山推”系统设计完成了，但是要将它编码实现，还有很远的路要走，细枝末节的问题也会暴露更多。如果该系统用于线上计算，如有大量的Map-Reduce运行于group中，系统将会更复杂，因为此时不光考虑的数据的存储同步问题，操作也需要同步。现在来检验下我们设计的“山推”系统，主要分布式指标：

**一致性**：如前文所述，Global机器强一致性，Group机器最终一致性；

**可用性**：Global机器保证了HA（高可用性），Group机器则不保证，但满足了分区容错性；

**备份Replication**：Global机器采用完全同步，Group机器则是半同步模式，都可以进行横向扩展；

**故障恢复**：如前文所述，Global机器完全同步，故障可不受中断由slave恢复工作，但Group机器采用分布式选举和最终一致性，故障时有较短时间的写服务需要中断并切换到slave机器，但读服务可不中断。

还有其他一些指标，这里就不再多说了。还有一些细节，需要提一下，比如之前的评论中有同学提到，group中master挂时，由slave去顶替，但这样一来该group内其他所有slave需要分担之前成这新master的这个slave的压力，有可能继续挂掉而造成雪崩。针对此种情况，可采用如下做法：即在一个group内，至少还存在一个真正做“备份”用途的slave，平时不抗压力，只同步数据，这样当出现上述情况时，可由该备份slave来顶替成为新master的那个slave，从而避免雪崩效应。不过这样一来，就有新的问题，由于备份slave平时不抗压力，加入抗压力后必然产生一定的数据迁移，数据迁移也是一个较麻烦的问题。常采用的分摊压力做法如一致性Hash算法（环状Hash），可将新结点加入对整个group的影响降到较小的程度。

另外，还有一个较为棘手的问题，就是系统的日志处理，主要是系统宕机后如何恢复之前的操作日志。比较常见的方法是对日志作快照（Snapshot）和回放点（checkpoint），并采用Copy-on-write方式定期将日志作snapshot存储，当发现宕机后，找出对应的回放点并恢复之后的snapshot，但此时仍可能有新的写操作到达，并产生不一致，这里主要依靠Copy-on-write来同步。

最后再说说图中的Client部分。显然这个模块就是面向Web的接口，后面连接我们的“山推”系统，它可以包含诸多业务逻辑，最重要的，是要缓存group的信息。在Client和Web之间，还可以有诸如Nginx之类的反向代理服务器存在，做进一步性能提升，这已经超出了本文的范畴，但我们必须明白的是，一个高并发高性能的网站，对性能的要求是从起点开始的，何为起点，即用户的浏览器。

现在，让我们来看看GFS的设计：

很明显，这么牛的系统我是设计不出来的，我们的“山推”，就是在学习GFS + Bigtable的主要思想。说到这，也必须提一句，可能我文章中，名词摆的有点多了，如NWR，分布式选举，Paxos包括Copy-on-write等，有兴趣的同学可自行google了解。因为说实在的，这些概念我也没法讲透彻，只是一知半解。另外，大家可参考一些分布式项目的设计，如Cassandra，包括淘宝的Oceanbase等，以加深理解。