分布式系统常用思想和技术总结

February 10, 2014

一、分布式系统的难点

分布式系统比起单机系统存在哪些难点呢？

1. 网络因素

由于服务和数据分布在不同的机器上，每次交互都需要跨机器运行，这带来如下几个问题：

1. 网络延迟：性能、超时

同机房的网络IO还是比较块的，但是跨机房，尤其是跨IDC，网络IO就成为不可忽视的性能瓶颈了。并且，延迟不是带宽，带宽可以随便增加，千兆网卡换成万兆，只是成本的问题，但延迟是物理限制，基本不可能降低。

这带来的问题就是系统整体性能的降低，会带来一系列的问题，比如资源的锁住，所以系统调用一般都要设置一个超时时间进行自我保护，但是过度的延迟就会带来系统的RPC调用超时，引发一个令人头疼的问题：分布式系统调用的三态结果：成功、失败、超时。不要小看这个第三态，这几乎是所有分布式系统复杂性的根源。

针对这个问题有一些相应的解决方案：异步化，失败重试。 而对于跨IDC数据分布带来的巨大网络因素影响，则一般会采用数据同步，代理专线等处理方式。

2. 网络故障：丢包、乱序、抖动。

这个可以通过将服务建立在可靠的传输协议上来解决，比如TCP协议。不过带来的是更多的网络交互。因此是性能和流量的一个trade off。这个在移动互联网中更需要考虑。

2. 鱼与熊掌不可兼得——CAP定律

CAP理论是由Eric Brewer提出的分布式系统中最为重要的理论之一：

1. Consistency：[强]一致性，事务保障，ACID模型。
2. Availiablity：[高]可用性，冗余以避免单点，至少做到柔性可用（服务降级）。
3. Partition tolerance：[高]可扩展性（分区容忍性）：一般要求系统能够自动按需扩展，比如HBase。

CAP原理告诉我们，这三个因素最多只能满足两个，不可能三者兼顾。对于分布式系统来说，分区容错是基本要求，所以必然要放弃一致性。对于大型网站来说，分区容错和可用性的要求更高，所以一般都会选择适当放弃一致性。对应CAP理论，NoSQL追求的是AP，而传统数据库追求的是CA，这也可以解释为什么传统数据库的扩展能力有限的原因。

在CAP三者中，“可扩展性”是分布式系统的特有性质。分布式系统的设计初衷就是利用集群多机的能力处理单机无法解决的问题。当需要扩展系统性能时，一种做法是优化系统的性能或者升级硬件(scale up)，一种做法就是“简单”的增加机器来扩展系统的规模(scale out)。好的分布式系统总在追求”线性扩展性”，即性能可以随集群数量增长而线性增长。

可用性和可扩展性一般是相关联的，可扩展行好的系统，其可用性一般会比较高，因为有多个服务(数据)节点，不是整体的单点。所以分布式系统的所有问题，基本都是在一致性与可用性和可扩展性这两者之间的一个协调和平衡。对于没有状态的系统，不存在一致性问题，根据CAP原理，它们的可用性和分区容忍性都是很高，简单的添加机器就可以实现线性扩展。而对于有状态的系统，则需要根据业务需求和特性在CAP三者中牺牲其中的一者。一般来说，交易系统类的业务对一致性的要求比较高，一般会采用ACID模型来保证数据的强一致性，所以其可用性和扩展性就比较差。而其他大多数业务系统一般不需要保证强一致性，只要最终一致就可以了，它们一般采用BASE模型，用最终一致性的思想来设计分布式系统，从而使得系统可以达到很高的可用性和扩展性。

CAP定律其实也是衡量分布式系统的重要指标，另一个重要的指标是性能。

一致性模型

主要有三种：

1. Strong Consistency（强一致性）：新的数据一旦写入，在任意副本任意时刻都能读到新值。比如：文件系统，RDBMS，Azure Table都是强一致性的。
2. Week Consistency（弱一致性）：不同副本上的值有新有旧，需要应用方做更多的工作获取最新值。比如Dynamo。
3. Evantual Consistency（最终一致性）：一旦更新成功，各副本的数据最终将达到一致。

从这三种一致型的模型上来说，我们可以看到，Weak和Eventually一般来说是异步冗余的，而Strong一般来说是同步冗余的(多写)，异步的通常意味着更好的性能，但也意味着更复杂的状态控制。同步意味着简单，但也意味着性能下降。

以及其他变体：

1. Causal Consistency（因果一致性）：如果Process A通知Process B它已经更新了数据，那么Process B的后续读取操作则读取A写入的最新值，而与A没有因果关系的C则可以最终一致性。
2. Read-your-writes Consistency（读你所写一致性）：如果Process A写入了最新的值，那么 Process A的后续操作都会读取到最新值。但是其它用户可能要过一会才可以看到。
3. Session Consistency（会话一致性）：一次会话内一旦读到某个值，不会读到更旧的值。
4. Monotonic Read Consistency（单调一致性）：一个用户一旦读到某个值，不会读到比这个值更旧的值，其他用户不一定。

等等。

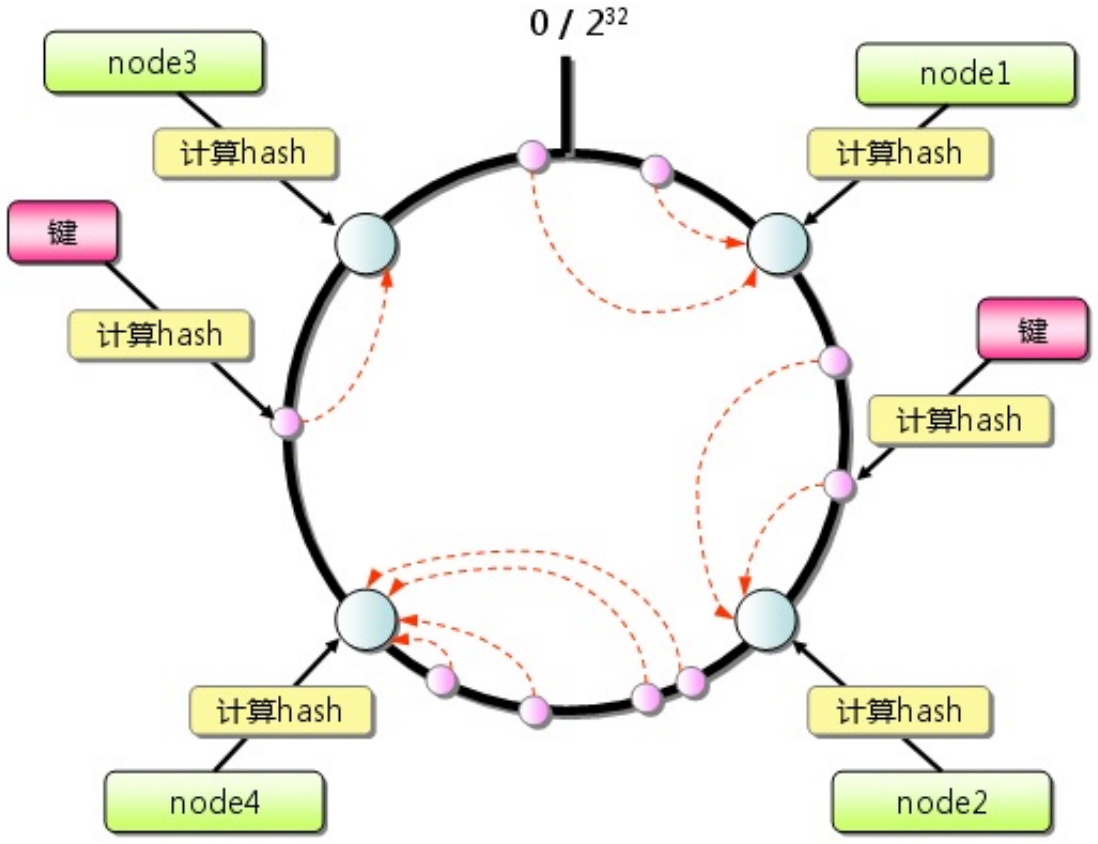
其中最重要的变体是第二条：Read-your-Writes Consistency。特别适用于数据的更新同步，用户的修改马上对自己可见，但是其他用户可以看到他老的版本。Facebook的数据同步就是采用这种原则。

二、分布式系统常用技术和应用场景

- consistent hashing [with virtual node]：一致性哈希，数据分布
- vector clock：时钟向量，多版本数据修改
- Quorum W+R>N [with vector clock]：抽屉原理，数据一致性的另一种解决方案。时钟向量，多版本数据修改。
- Merkle tree [with anti-entropy]：数据复制
- MVCC：copy-on-write与snapshot
- 2PC/3PC：分布式事务
- Paxos：强一致性协议
- Symmetry and Decentralization：对称性和去中心化。对称性(symmetry)简化了系统的配置和维护。去中心化是对对称性的延伸，可以避免master单点，同时方便集群scale out。
- Map-Reduce：分而治之；移动数据不如移动计算。将计算尽量调度到与存储节点在同一台物理机器上的计算节点上进行，这称之为本地化计算。本地化计算是计算调度的一种重要优化。
- Gossip协议：节点管理
- Lease机制：

consistent hashing：一致性哈希，解决数据均衡分布问题

我们通常使用的hash算法是hash() mod n，但是如果发生某个节点失效时，无法快速切换到其他节点。为了解决单点故障的问题，我们为每个节点都增加一个备用节点，当某个节点失效时，就自动切换到备用节点上，类似于数据库的master和slave。但是依然无法解决增加或删除节点后，需要做hash重分布的问题，也就是无法动态增删节点。这时就引入了一致性hash的概念，将所有的节点分布到一个hash环上，每个请求都落在这个hash环上的某个位置，只需要按照顺时针方向找到的第一个节点，就是自己需要的服务节点。当某个节点发生故障时，只需要在环上找到下一个可用节点即可。



一致性hash算法最常用于分布式cache中，比如注意的memcached。Dynamo也用其作为数据

分布算法，并且对一致性算法进行了改进，提出了基于虚拟节点的改进算法，其核心思路是引入虚拟节点，每个虚拟节点都有一个对应的物理节点，而每个物理节点可以对应若干个虚拟节点。

关于一致性hash的更多内容，可以参考笔者另一篇博文：Memcached的分布式算法学习。

这篇文章也可以看看：某分布式应用实践一致性哈希的一些问题

virtual node

前面说过，有的Consistent Hashing的实现方法采用了虚拟节点的思想。使用一般的hash函数的话，服务器的映射地点的分布非常不均匀。因此，使用虚拟节点的思想，为每个物理节点（服务器）在continuum上分配100～200个点。这样就能抑制分布不均匀，最大限度地减小服务器增减时的缓存重新分布。

Quorum W+R>N：抽屉原理，数据一致性的另一种解决方案

N: 复制的节点数，即一份数据被保存的份数。 R: 成功读操作的最小节点数，即每次读取成功需要的份数。 W: 成功写操作的最小节点数 ，即每次写成功需要的份数。

所以 W+R>N的意思是：对于有N份拷贝的分布式系统，写到W(W<=N)份成功算写成功，读R(R<=N)份数据算读成功。

这三个因素决定了可用性，一致性和分区容错性。W+R>N可以保证数据的一致性(C)，W越大数据一致性越高。这个NWR模型把CAP的选择权交给了用户，让用户自己在功能，性能和成本效益之间进行权衡。

对于一个分布式系统来说，N通常都大于3，也就说同一份数据需要保存在三个以上不同的节点上，以防止单点故障。W是成功写操作的最小节点数，这里的写成功可以理解为“同步”写，比如N=3，W=1，那么只要写成功一个节点就可以了，另外的两份数据是通过异步的方式复制的。R是成功读操作的最小节点数，读操作为什么要读多份数据呢？在分布式系统中，数据在不同的节点上可能存在着不一致的情况，我们可以选择读取多个节点上的不同版本，来达到增强一致性的目的。

NWR模型的一些设置会造成脏数据和版本冲突问题，所以一般要引入vector clock算法来解决这个问题。

需要保证系统中有max(N-W+1,N-R+1)个节点可用。

关于NWR模型，建议阅读 分布式系统的事务处理，写的很通俗易懂。

vector clock：时钟向量，多版本数据修改

参见 分布式系统的事务处理，写的很通俗易懂。

lease机制

chubby、zookeeper 获得lease（租约）的节点得到系统的承诺：在有效期内数据/节点角色等是有效的，不会变化的。

lease机制的特点：

- lease颁发过程只需要网络可以单向通信，同一个lease可以被颁发者不断重复向接受方

发送。即使颁发者偶尔发送lease失败，颁发者也可以简单的通过重发的办法解决。
- 机器宕机对lease机制的影响不大。如果颁发者宕机，则宕机的颁发者通常无法改变之前的承诺，不会影响lease的正确性。在颁发者机恢复后，如果颁发者恢复出了之前的lease 信息，颁发者可以继续遵守lease的承诺。如果颁发者无法恢复lease信息，则只需等待一个最大的lease超时时间就可以使得所有的lease都失效，从而不破坏lease机制。
- lease机制依赖于有效期，这就要求颁发者和接收者的时钟是同步的。
    - 如果颁发者的时钟比接收者的时钟慢，则当接收者认为lease已经过期的时候，颁发者依旧认为lease有效。接收者可以用在lease到期前申请新的lease的方式解决这个问题。
    - 如果颁发者的时钟比接收者的时钟快，则当颁发者认为lease已经过期的时候，可能将lease颁发给其他节点，造成承诺失效，影响系统的正确性。对于这种时钟不同步，实践中的通常做法是将颁发者的有效期设置得比接收者的略大，只需大过时钟误差就可以避免对lease的有效性的影响。

工程中，常选择的lease时长是10秒级别，这是一个经过验证的经验值，实践中可以作为参考并综合选择合适的时长。

双主问题（脑裂问题）

lease机制可以解决网络分区问题造成的“双主”问题，即所谓的“脑裂”现象。配置中心为一个节点发放lease，表示该节点可以作为primary节点工作。当配置中心发现primary有问题时，只需要等到前一个primary的lease过期，就可以安全地颁发新的lease给新的primary节点，而不会出现“双主”问题。 在实际系统中，若用一个中心节点作为配置中心发送lease也有很大的风险。实际系统总是使用多个中心节点互为副本，成为一个小的集群，该小集群具有高可用性，对外提供颁发lease的功能。chubby和zookeeper都是基于这样的设计。

chubby一般有五台机器组成一个集群，可以部署成两地三机房。chubby内部的五台机器需要通过Paxos协议选取一个chubby master机器，其它机器是chubby slave，同一时刻只有一个chubby master。chubby相关的数据，比如锁信息，客户端的session信息等都需要同步到整个集群，采用半同步的做法，超过一半的机器成功就可以回复客户端。最后可以确保只有一个和原有的chubby master保持完全同步的chubby slave被选取为新的chubby master。

Gossip协议

Gossip用于P2P系统中自治节点获悉对集群认识（如集群的节点状态，负载情况等）。 系统中的节点定期互相八卦，很快八卦就在整个系统传开了。 A、B两个节点八卦的方式主要是：A告诉B知道哪些人的什么八卦；B告诉A这些八卦里B知道哪些更新了；B更新A告诉他的八卦..... 说是自治系统，其实节点中还有一些种子节点。种子节点的作用主要是在有新节点加入系统时体现。新节点加入系统中，先与种子节点八卦，新节点获得系统信息，种子节点知道系统中多了新节点。其他节点定期与种子节点八卦的时候就知道有新节点加入了。 各个节点互相八卦的过程中，如果发现某个节点的状态很长时间都没更新，就认为该节点已经宕机了。

Dynamo使用了Gossip协议来做会员和故障检测。

2PC、3PC、Paxos协议: 分布式事务的解决方案

分布式事务很难做，所以除非必要，一般来说都是采用最终一致性来规避分布式事务。

目前底层NoSQL存储系统实现分布式事务的只有Google的系统，它在Bigtable之上用Java语言开发了一个系统 Megastore，实现了两阶段锁，并通过Chubby来避免两阶段锁协调者宕机带来的问题。Megastore实现目前只有简单介绍，还没有相关论文。

2PC

实现简单，但是效率低，所有参与者需要block，throughput低；无容错，一个节点失败整个事务失败。如果第一阶段完成后，参与者在第二阶没有收到决策，那么数据结点会进入“不知所措”的状态，这个状态会block住整个事务。

3PC

改进版的2PC，把2PC的第一个段break成了两段: 询问，然后再锁资源，最后真正提交。3PC的核心理念是：在询问的时候并不锁定资源，除非所有人都同意了，才开始锁资源。

3PC比2PC的好处是，如果结点处在P状态（PreCommit）的时候发生了Fail/Timeout的问题，3PC可以继续直接把状态变成C状态（Commit），而2PC则不知所措。

不过3PC实现比较困难，而且无法处理网络分离问题。如果preCommit消息发送后两个机房断开，这时候coordinator所在的机房会abort，剩余的participant会commit。

Paxos

Paxos的目的是让整个集群的结点对某个值的变更达成一致。Paxos算法是一种基于消息传递的一致性算法。Paxos算法基本上来说是个民主选举的算法——大多数的决定会成个整个集群的统一决定。

任何一个点都可以提出要修改某个数据的提案，是否通过这个提案取决于这个集群中是否有超过半数的结点同意（所以Paxos算法需要集群中的结点是单数）。这个是Paxos相对于2PC和3PC最大的区别，在2f+1个节点的集群中，允许有f个节点不可用。

Paxos的分布式民主选举方式，除了保证数据变更的一致性之外，还常用于单点切换，比如Master选举。

Paxos协议的特点就是难，both 理解 and 实现 :(

关于2PC，3PC和Paxos，强烈推荐阅读 分布式系统的事务处理。

目前大部分支付系统其实还是在2PC的基础上进行自我改进的。一般是引入一个差错处理器，进行差错协调（回滚或者失败处理）。

MVCC：多版本并发控制

这个是很多RDMS存储引擎实现高并发修改的一个重要实现机制。具体可以参考：

1. 多版本并发控制(MVCC)在分布式系统中的应用
2. MVCC (Oracle, Innodb, Postgres).pdf

Map-Reduce思想

1. 分而治之

2. 移动数据不如移动计算

如果计算节点和存储节点位于不同的物理机器则计算的数据需要通过网络传输，此种方式的开

销很大。另一种思路是，将计算尽量调度到与存储节点在同一台物理机器上的计算节点上进行，这称之为本地化计算。本地化计算是计算调度的一种重要优化。

经典论文和分布式系统学习

Dynamo

HBase

LSM Tree

- LSM（Log Structured Merge Trees）是B+ Tree一种改进
- 牺牲了部分读性能，用来大幅提高写性能
- 思路：拆分树
    - 首先写WAL，然后记录数据到入到内存中，构建一颗有序子树(memstore)
    - 随着子树越来越大，内存的子树会flush到磁盘上(storefile)
    - 读取数据：必须遍历所有的有序子树（不知数据在哪棵子树）
    - Compact：后台线程对磁盘中的子树进行归并，变成大树（子树多了读得慢）

事实上，lucene的索引机制也类似HBase的LSM树。也是写的时候分别写在单独的segment，后台进行segement合并。

参考文档

1. NoSQL漫谈
2. 多IDC的数据分布设计(一)
3. 分布式系统的事务处理
4. 海量存储系列之四-单机事务处理
5. 本人的一些技术方面的分享集合
6. Learning from google megastore (Part-1)

Related Posts

- 24 Jul 2015 » Tomcat调优
- 22 Jul 2015 » 记一次MySQL主从同步错误处理
- 03 Jul 2015 » Metric监控系统