**我的探索历程**

　　这一部分，与分布式不大相关，记录的是我是如何在分布式学习这条道路上摸索的，不感兴趣的读者请直接跳到下一章。

　　过去的一年，我在分布式学习这条道路上苦苦徘徊，始终没有找到一个好的学习方法，一个能激励我持续学习的方法。

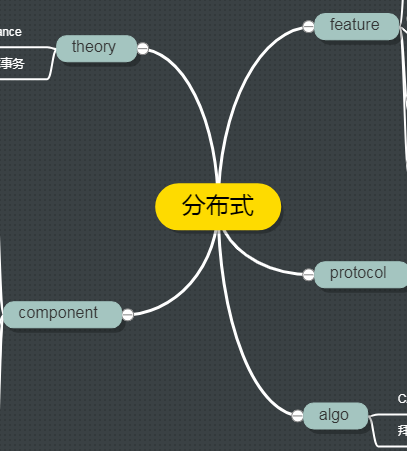
　　当然，我也在不停的反思，最先发现的问题是学习不系统，东看看，西看看，看过就忘。于是写了一篇文章《[带着问题学习分布式系统](http://www.cnblogs.com/xybaby/p/6930977.html)》提出了一些问题，然后写了两篇文章来回带自己提出的问题。回过头看，其实这两篇文章都是基于分布式存储的。

　　然后接下来，又是一段时间的迷茫，不知道要学习啥（也是在学习Paxos的时候遇到了阻碍），偶然看到了关于SMART的一些文章，发现原来自己的学习一点都不SAMRT，所以坚持不下去。于是又写了一篇文章《[带着SMART原则重新出发](http://www.cnblogs.com/xybaby/p/7295555.html)》，希望给自己设定SMART的目标，结果被博友打脸“一点都不smart”，而且也被结果打脸了，确实没有让我“重新出发”。

　　我也曾经在网上搜索，“如何学习分布式系统“，但并没有找到理想的、或者说适合我的学习方法。网上的答案，都是说，看这些论文，看这些系统的实现。。。或者很多大学的课程，都是罗列出一堆术语：概念、协议、组件。而我觉得，逐个去了解这些术语似乎很无聊，因为不知道为什么要这个协议，或者说这些协议、组件是如何组合起来的。

　　在没有得到适合自己的方案之后，我自己思考，希望更系统、全面的掌握，于是写了这篇文章《[什么是分布式系统，如何学习分布式系统](http://www.cnblogs.com/xybaby/p/7787034.html)》。主要内容是，从一个实际的大型网络出发，思考会遇到哪些问题，需要哪些分布式协议与组件。写完这篇文章，我的想法就是，逐个去学习这些组件。

　　春节的时候，看完了《[小强升职记](http://www.cnblogs.com/xybaby/p/8464430.html)》，里面写到要用六个高度与大树分解法来设定自己的目标，制定自己的执行计划。于是，我尝试用这两个方法来指导我对分布式的学习。遗憾的是，在大数据分解法 树干（月计划）这一层，我就无法计划下去了，不知道每个月要学啥。具体原因，我想，还是没有一个对分布式系统的整理的认识。怎么整体把握呢，我想起了之前看到过的思维导图，于是就想画一张分布式系统的思维导图，然后逐个击破。当然，最开始也是瞎画，当时划出了5个树干，是这样子的：



　　在延伸feature（分布式系统需要考虑的特性）的时候，我逐渐明白，这是因为要满足这些feature，才设计了很多协议与算法，也提出了一些理论。比如说，这是因为要解决去中心化副本的一致性问题，才引入了Paxos（raft）协议。而每一个分布式系统，如分布式存储、分布式计算、分布式消息队列、分布式RPC框架，根据业务的不同，会使用不同的方法来满足这些feature，对这些feature的支持也可能会有权衡，比如一致性与可用性的权衡。

　　所有，我觉得从分布式的特性出发，来一步步学习分布式是一种可行的方式。

**从分布式系统的特征出发**

　　分布式的世界中涉及到大量的协议（raft、2pc、lease、quorum等）、大量的理论（FLP， CAP等）、大量的系统（GFS、MongoDB、MapReduce、Spark、RabbitMQ等）。这些大量的知识总是让我们无从下手，任何一个东西都需要花费大量的时间，特别是在没有项目、任务驱动的时候，没有一个明确的目标，真的很难坚持下去。

　　所以，我一直在思考，能有什么办法能把这些东西串起来？当我掌握了知识点A的时候，能够自然地想到接下来要学习B知识，A和B的关系，也许是递进的，也许是并列的。我也这样尝试了，那就是《[什么是分布式系统，如何学习分布式系统](http://www.cnblogs.com/xybaby/p/7787034.html)》一文中我提到的，思考一个大型网站的架构，然后把这些协议、理论串起来。按照这个想法，我的计划就是去逐个学习这些组件。

　　但是，其实在这里有一个误区，我认为一个大型网站就是一个分布式系统，包含诸多组件，这些组件是分布式系统的组成部分；而我现在认为，一个大型网站包含诸多组件，每一个组件都是一个分布式系统，比如分布式存储就是一个分布式系统，消息队列就是一个分布式系统。

　　**为什么说从思考分布式的特征出发，是一个可行的、系统的、循序渐进的学习方式呢**，因为：

　　（1）先有问题，才会去思考解决问题的办法

　　由于我们要提高可用性，所以我们才需要冗余；由于需要扩展性，所以我们才需要分片

　　（2）解决一个问题，常常会引入新的问题

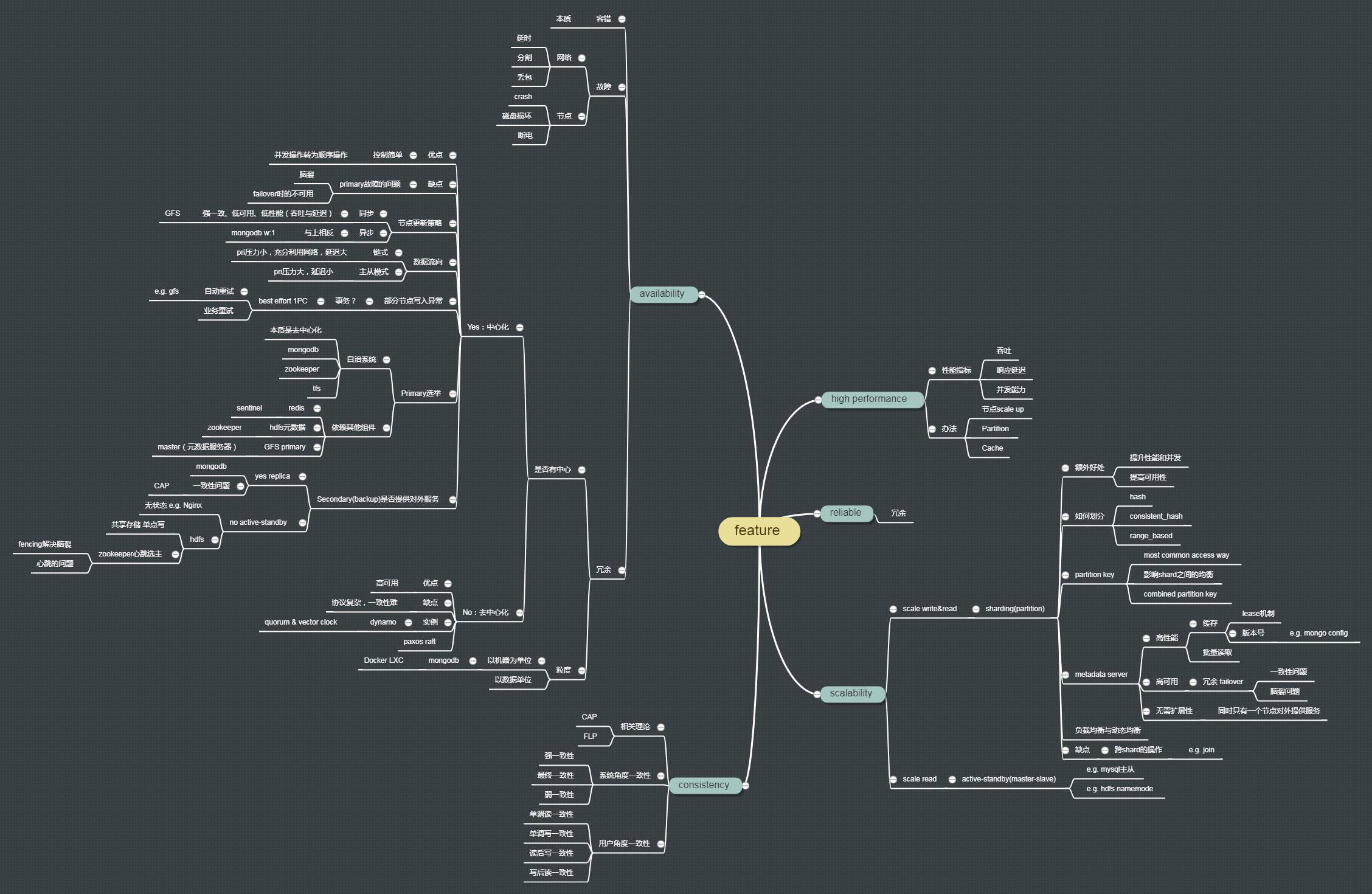
　　比如，为了提高可用性，引入了冗余；而冗余又带来了副本之间的一致性问题，所以引入了中心化副本协议（primary/secondary)；那么接下来就要考虑primary（节点）故障时候的选举问题。。。

　　（3）这是一个金字塔结构，或者说，也是一个深度优先遍历的过程。

　　在这个过程中，我们始终知道自己已经掌握了哪些知识；还有哪些是已经知道，但未了解的知识；也能知道，哪些是空白，即我们知道这里可能有很多问题，但是具体是什么，还不知道。

　　那么，各个分布式系统如何与这些特征相关联呢？不难发现，每个分布式系统都会或多或少的体现出这些特征，只是使用的方法、算法可能不大一样。所以，我们应该思考，某一个问题，在某个特定系统中是如何解决的。比如元数据管理的强一致性，在MongoDB中是如何实现的，在HDFS中是如何实现的。这也指导了我们如何去学习一个具体的分布式系统：带着问题，只关注关心的部分，而不是从头到尾看一遍。

　　下面是，到目前为止，我对分布式特征的思维导图



　　对于上图，需要声明的是，第一：不一定完全正确，第二：不完整。这是因为，我自己也在学习中，可以看到，很多分支很短（比如去中心化副本协议），不是因为这一块没有内容，而是我压根儿还没去了解，还没去学习。

　　**我会持续跟新这幅脑图的**！

　　下一章，介绍一下分布式系统的各个特征。

**分布式系统的一般特征**

 　　任何介绍分布式系统的文章或者书籍都会提到分布式系统的几个特性：**可扩展性、高性能、高可用、一致性**。这几个特性也是分布式系统的衡量指标，正是为了在不同的程度上满足这些特性（或者说达到这些指标），才会设计出各种各样的算法、协议，然后根据业务的需求在这些特性间平衡。

　　那么本章节简单说明，为什么要满足这些特性，要满足这些特性需要解决什么问题，有什么好的解决方案。

**可扩展性**



Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

　　可扩展性是指当系统的任务（work）增加的时候，通过增加资源来应对任务增长的能力。可扩展性是任何分布式系统必备的特性，这是由分布式系统的概念决定的：

分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统

　　分布式系统的出现是为了解决单个计算机无法完成的计算、存储任务。那么当任务规模增加的时候，必然就需要添加更多的节点，这就是可扩展性。

　　扩展性的目标是使得系统中的节点都在一个较为稳定的负载下工作，这就是负载均衡，当然，在动态增加节点的时候，需要进行任务（可能是计算，可能是数据存储）的迁移，以达到动态均衡。

　　那么首先要考虑的问题就是，如何对任务进行拆分，将任务的子集分配到每一个节点，我们称这个过程问题Partition（Sharding）。关于Partition，其实我在《[带着问题学习分布式系统之数据分片](http://www.cnblogs.com/xybaby/p/7076731.html) 》一文中有详细介绍，这里进行归纳总结。

**第一：分片分式，即按照什么算法对任务进行拆分**

　　常见的算法包括：哈希（hash），一致性哈希（consistency hash），基于数据范围（range based）。每一种算法有各自的优缺点，也就有各自的适用场景。

**第二：分片的键，partition key**

　　partition key是数据的特征值，上面提到的任何分片方式都依赖于这个partition key，那么该如何选择呢

 based on what you think the primary access pattern will be

　　partition key会影响到任务在分片之间的均衡，而且一些系统中（mongodb）几乎是不能重新选择partition key的，因此在设计的时候就得想清楚

**第三：分片的额外好处**

　　提升性能和并发：不同的请求分发到不同的分片

　　提高可用性：一个分片挂了不影响其他的分片

**第四：分片带来的问题**

　　如果一个操作需要跨越多个分片，那么效率就会很低下，比如数据中的join操作

**第五：元数据管理**

　　元数据记录了分片与节点的映射关系、节点状态等核心信息，分布式系统中，有专门的节点（节点集群）来管理元数据，我们称之为元数据服务器。元数据服务器有以下特点：

　　高性能：cache

　　高可用：冗余 加 快速failover

　　强一致性（同时只有一个节点对外提供服务）

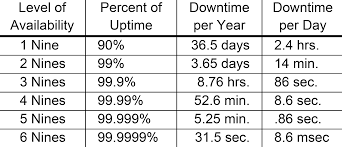
**第六：任务的动态均衡**

　　为了达到动态均衡，需要进行数据的迁移，如何保证在迁移的过程中保持对外提供服务，这也是一个需要精心设计的复杂问题。

**可用性**



　　可用性（Availability）是系统不间断对外提供服务的能力，可用性是一个度的问题，最高目标就是7 \* 24，即永远在线。但事实上做不到的，一般是用几个9来衡量系统的可用性，如下如所示：



　　也就是如果要达到4个9的可用度（99.99%），那么一年之中只能有52.6分钟不可用，这是个巨大的挑战

　　为什么分布式系统中必须要考虑可用性呢，这是因为分布式系统中故障的概率很高。分布式系统由大量异构的节点和网络组成，节点可能会crash、断电、磁盘损坏，网络可能丢包、延迟、网络分割。系统的规模放大了出故障的概率，因此分布式系统中，故障是常态。那么分布式系统的其中一个设计目标就是容错，在部分故障的情况下仍然对外提供服务，这就是可用性。

　　**冗余是提高可用性、可靠性的法宝**。

　　冗余就是说多个节点负责相同的任务，在需要状态维护的场景，比如分布式存储中使用非常广泛。在分布式计算，如MapReduce中，当一个worker运行异常缓慢时，master会将这个worker上的任务重新调度到其它worker，以提高系统的吞吐，这也算一种冗余。但存储的冗余相比计算而言要复杂许多，因此主要考虑存储的冗余。

　　维护同一份数据的多个节点称之为多个副本。我们考虑一个问题，当向这个副本集写入数据的时候，怎么保证并发情况下数据的一致性，是否有一个节点有决定更新的顺序，这就是中心化、去中心话副本协议的区别。

**中心化与去中心化**

　　中心化就是有一个主节点（primary master）负责调度数据的更新，其优点是协议简单，将并发操作转变为顺序操作，缺点是primar可能成为瓶颈，且在primary故障的时候重新选举会有一段时间的不可用。

　　去中心化就是所有节点地位平等，都能够发起数据的更新，优点是高可用，缺点是协议复杂，要保证一致性很难。

　　提到去中心化，比较有名的是dynamo，cassandra，使用了quorum、vector clock等算法来尽量保证去中心化环境下的一致性。对于去中心化这一块，目前还没怎么学习，所以下面主要讨论中心化副本集。对于中心化副本协议，我在《[带着问题学习分布式之中心化复制集](http://www.cnblogs.com/xybaby/p/7153755.html)》一文中也有详细介绍，这里简单归纳总结。

**节点更新策略**

　　primary节点到secondary节点的数据时同步还是异步，即客户端是否需要等待数据落地到副本集中的所有节点。

　　同步的优点在于强一致性，但是可用性和性能（响应延迟）比较差；异步则相反。

**数据流向**

　　即数据是如何从Primary节点到secondary节点的，有链式和主从模式。

　　链式的优点时充分利用网络带宽，减轻primary压力，但缺点是写入延迟会大一些。GFS，MongoDB（默认情况下）都是链式。

**部分节点写入异常**

　　理论上，副本集中的多个节点的数据应该保持一致，因此多个数据的写入理论上应该是一个事务：要么都发生，要么都不发生。但是分布式事务（如2pc）是一个复杂的、低效的过程，因此副本集的更新一般都是best effort 1pc，如果失败，则重试，或者告诉应用自行处理。

**primary的选举**

　　在中心化副本协议中，primary节点是如何选举出来的，当primary节点挂掉之后，又是如何选择出新的primary节点呢，有两种方式：自治系统，依赖其他组件的系统。（ps，这两个名字是我杜撰的 。。。）

　　所谓的自治系统，就是节点内部自行投票选择，比如mongodb，tfs，zookeeper

　　依赖其他组件的系统，是指primary由副本集之后的组件来任命，比如GFS中的primary由master（GFS的元数据服务器）任命，hdfs的元数据namenode由zookeeper心跳选出。

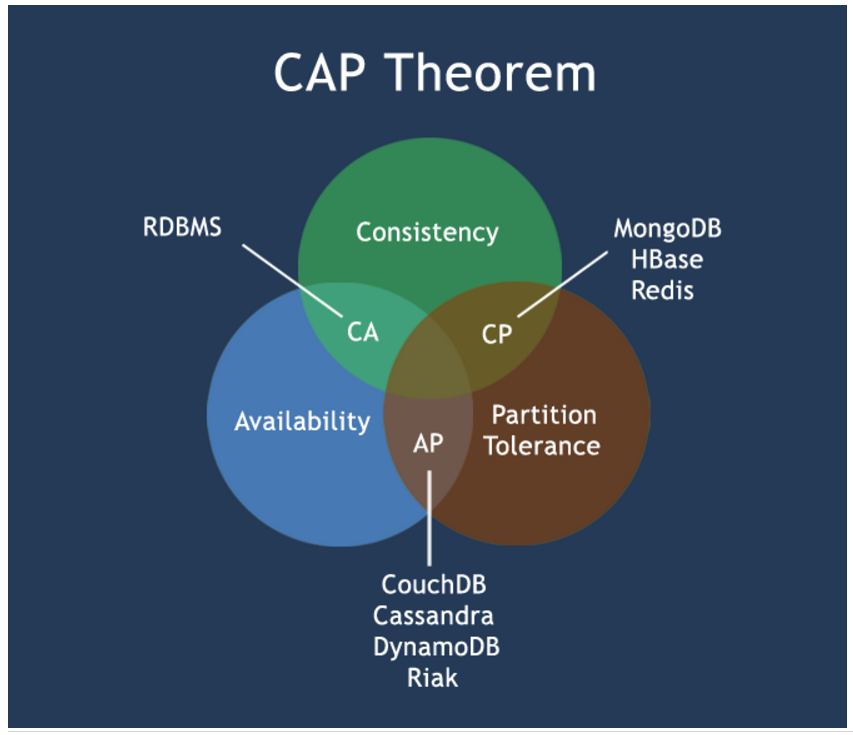
**secondary是否对外提供服务（读服务）**

 　　中心化复制集中，secondary是否对外提供读服务，取决于系统对一致性的要求。

　　比如前面介绍到节点更新策略时，可能是异步的，那么secondary上的数据相比primary会有一定延迟，从secondary上读数据的话无法满足强一致性要求。

　　比如元数据，需要强一致性保证，所以一般都只会从primary读数据。而且，一般称主节点为active（master），从节点为standby（slave）。在这种情况下，是通过冗余 加上 快速的failover来保证可用性。

**一致性**



　　从上面可以看到，为了高可用性，引入了冗余（副本）机制，而副本机制就带来了一致性问题。当然，如果没有冗余机制，或者不是数据（状态）的冗余，那么不会出现一致性问题，比如MapReduce。

　　一致性与可用性在分布式系统中的关系，已经有足够的研究，形成了CAP理论。CAP理论就是说分布式数据存储，最多只能同时满足一致性（C，Consistency）、可用性（A， Availability）、分区容错性（P，Partition Tolerance）中的两者。但一致性和可用性都是一个度的问题，是0到1，而不是只有0和1两个极端。详细可以参考之前的文章《[CAP理论与MongoDB一致性、可用性的一些思考](http://www.cnblogs.com/xybaby/p/6871764.html)》

　　一致性从系统的角度和用户的角度有不同的等级。

**系统角度的一致性**

 　　强一致性、若一致性、最终一致性

**用户角度的一致性**

 　　单调读一致性，单调写一致性，读后写一致性，写后读一致性

**高性能**



 　　正式因为单个节点的scale up不能完成任务，因此我们才需要scale out，用大量的节点来完成任务，分布式系统的理想目标是任务与节点按一定的比例线性增长。

**衡量指标**

　　高并发

　　高吞吐

　　低延迟

　　不同的系统关注的核心指标不一样，比如MapReduce，本身就是离线计算，无需低延迟

**可行的办法**

　　单个节点的scaleup

　　分片（partition）

　　缓存：比如元数据

 　   短事务

**references**

 [Distributed systems for fun and profit](http://book.mixu.net/distsys/)

刘杰:《[分布式系统原理介绍](http://blog.sciencenet.cn/home.php?mod=attachment&id=31413)》