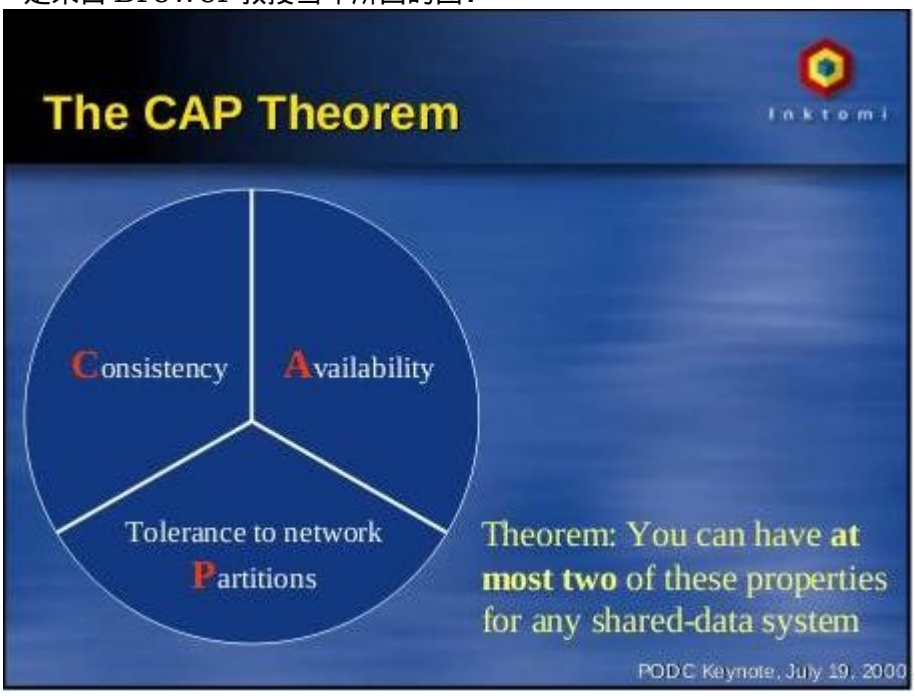
Dynamo 简介

1. NoSQL
NoSQL 的真谛其实应该是 Not Only SQL，其产生背景是在数据量和访问量逐渐增大的情况下下，人为地去添加机器或者切分数据到不同的机器，变得越来越困难，人力成本越来越高，于是便开始有了这样的项目，它们的本意是提高数据存储的自动化程度，减少人为干预的时间，让负载更加均匀等。
Amazon 的 dynamo 是其代表作，Amazon 于 2006 年推出了自己的云存储服务 S3，2007 年其 CTO 公布了 S3 的设计方案。

Dynamo 的意思是发电机，顾名思义，这一整套的方案都像发电机一样，源源不断地提供服务，永不间断。

2. CAP 原则
先来看历史，Eric A. Brewer 教授，Inktomi 公司的创始人，也是 berkeley 大学的计算机教授，Inktomi 是雅虎搜索现在的台端技术核心支持。最主要的是，他们（Inktomi 公司）在最早的时间里，开始研究分布计算。CAP 原则的提出，可以追溯到 2000 年的时候（可以想象有多么早！），
Brewer 教授在一次谈话中，基于他运作 Inktomi 以及在伯克利大学里的经验，总结出了 CAP 原则
图一是来自 Brewer 教授当年所画的图：



图一：CAP 原则当年的 PPT

Consistency（一致性）：即数据一致性，每次信息的读取都需要反映最新更新后的数据，简单的说，就是数据复制到了 N 台机器，如果有更新，要 N 机器的数据是一起更新的。
Availability（可用性）：高可用性意味着每一次请求都可以成功完成并受到响应数据，也可以说是体现好的响应性能，此项意思主要就是速度。
Partition tolerance（分区宽容度）：这个是容错机制的要求。一个服务需要在局部出错的情况下，没有出错的那部分被复制的数据分区仍然可以支持部分服务的操作，可以简单的理解为可以很容易的在线增减机器以达到更高的扩展性，即所谓的横向扩展能力。简单地可理解为是节点的可扩展性。

定理：任何分布式系统只可同时满足二点，没法三者兼顾。
忠告：架构师不要将精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。

3. DHT——分布式哈希表

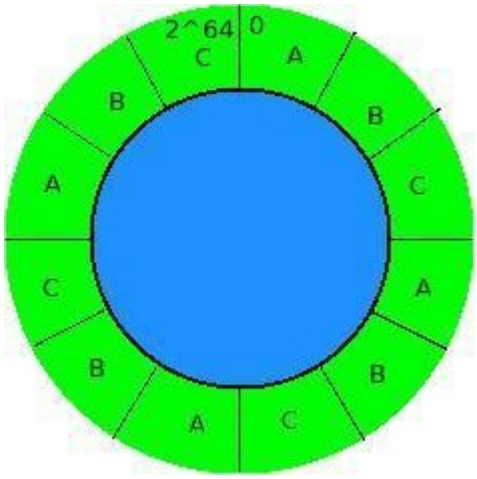
DHT（Distributed Hash Table，分布式哈希表），它是一种分布式存储寻址方法的统称。就像普通的哈希表，里面保存了 key 与 value 的对应关系，一般都能根据一个 key 去对应到相应的节点，从而得到相对应的 value。

这里随带一提，在 DHT 算法中，一致性哈希作为第一个实用的算法，在大多数系统中都使用了它。一致性哈希基本解决了在 P2P 环境中最为关键的问题——如何在动态的网络拓扑中分布存储和路由。每个节点仅需维护少量相邻节点的信息，并且在节点加入/退出系统时，仅有相关的少量节点参与到拓扑的维护中。至于一致性哈希的细节就不在这里详细说了，要指明的一点是，在 Dynamo 的数据分区方式之后，其实内部已然是一个对一致性哈希的改造了。

4. 进入 Dynamo 的世界

Dynamo 的数据分区与作用
在 Dynamo 的实现中提到一个关键的东西，就是数据分区。 假设我们的数据的 key 的范围是 0 到 2 的 64 次方（不用怀疑你的数据量会超过它，正常甚至变态情况下你都是超不过的，甚至像伏地魔等其他类 Dynamo 系统是使用的 2 的 32 次方），然后设置一个常数，比如说 1000，将我们的 key 的范围分成 1000 份。然后再将这 1000 份 key 的范围均匀分配到所有的节点（s 个节点），这样每个节点负责的分区数就是 1000/s 份分区。

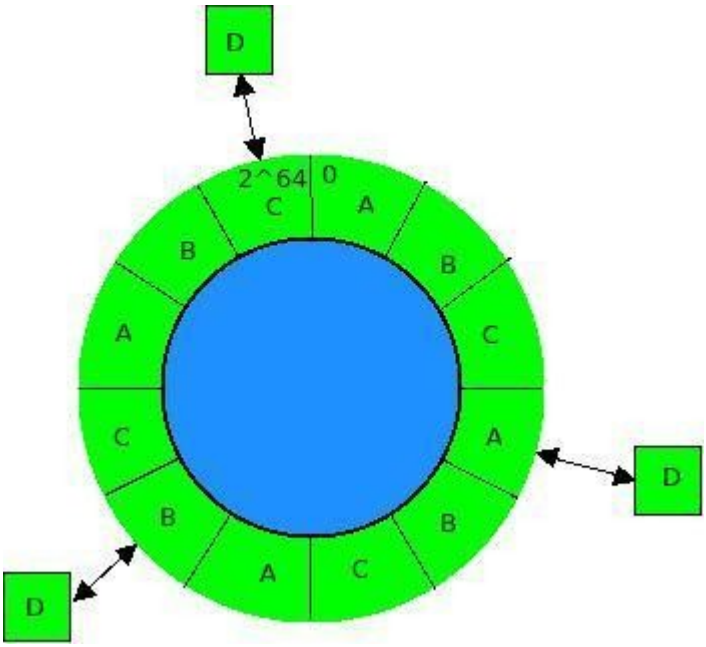
如图二，假设我们有 A、B、C 三台机器，然后将我们的分区定义了 12 个。



图二：三个节点分 12 个区的数据的情况

因为数据是均匀离散到这个环上的（有人开始会认为数据的 key 是从 1、2、3、4……这样子一直下去的，其实不是的，哈希计算出来的值，都是一个离散的结果），所以我们每个分区的数据量是大致相等的。从图上我们可以得出，每台机器都分到了四个分区里的数据，并且因为分区是均匀的，在分区数量是相当大的时候，数据的分布会更加的均匀，与此同时，负载也被均匀地分开了（当然了，如果硬要说你的负载还是只集中在一个分区里，那就不是在这里要讨论的问题了，有可能是你的哈希函数是不是有什么样的问题了）。

为什么要进行这样的分布呢，分布的好处在于，在有新机器加入的时候，只需要替换原有分区即可，如图三所示：



图三：加入一个新的节点 D 的情况

同样是图二里的情况，12 个分区分到 ABC 三个节点，图三中就是再进入了一个新的节点 D，从图上的重新分布情况可以得出，所有节点里只需要转移四分之一的数据到新来的节点即可，同时，新节点的负载也伴随分区的转移而转移了（这里的 12 个分区太少了，如果是 1200 个分区甚至是 12000 个分区的话，这个结论就是正确的了，12 个分区只为演示用）。

5. 从 Dynamo 的 NRW 看 CAP 法则

在 Dynamo 系统中，第一次提出来了 NRW 的方法。
N：复制的次数；
R：读数据的最小节点数；
W：写成功的最小分区数。
这三个数的具体作用是用来灵活地调整 Dynamo 系统的可用性与一致性。

举个例子来说，如果 R=1 的话，表示最少只需要去一个节点读数据即可，读到即返回，这时是可用性是很高的，但并不能保证数据的一致性，如果说 W 同时为 1 的 话，那可用性更新是最高的一种情况，但这时完全不能保障数据的一致性，因为在可供复制的 N 个节点里，只需要写成功一次就返回了，也就意味着，有可能在读的这一次并没有真正读到需要的数据（一致性相当的不好）。如果 W=R=N=3 的话，也就是说，每次写的时候，都保证所有要复制的点都写成功，读的时候也是都读到，这样子读出来的数据一定是正确的，但是其性能大打折扣，也就是说，数据的一致性非常的高，但系统的可用性却非常低了。如果 R + W > N 能够保证我们“读我们所写”，Dynamo 推荐使用 322 的组合。

Dynamo 系统的数据分区让整个网络的可扩展性其实是一个固定值（你分了多少区，实际上网络里扩展节点的上限就是这个数），通过 NRW 来达到另外两个方 向上的调整。

6. Dynamo 的一些增加可用性的补救

针对一些经常可能出现的问题，Dynamo 还提供了一些解决的方法。

第一个是 hinted handoff 数据的加入：在一个节点出现临时性故障时，数据会自动进入列表中的下一个节点进行写操作，并标记为 handoff 数据，在收到通知需要原节点恢复时重新把数据推回去。这能

使系统的写入成功大大提升。

第二个是向量时钟来做版本控制：用一个向量（比如说[a,1]表示这个数据在 a 节点第一次写入）来标记数据的版本，这样在有版本冲突的时候，可以追溯到出现问题的地方。这可以使数据的最终一致成为可能。（Cassandra 未用 vector clock，而只用 client timestamps 也达到了同样效果。）

第三个是 Merkle tree 来提速数据变动时的查找：使用 Merkle tree 为数据建立索引，只要任意数据有变动，都将快速反馈出来。

第四个是 Gossip 协议：一种通讯协议，目标是让节点与节点之间通信，省略中心节点的存在，使网络达到去中心化。提高系统的可用性。

后记
Dynamo 的理论对 CAP 原则里的可扩展性做到了很方便的实现，通过创造性的 NRW 来平衡系统的可用性和一致性，增加了系统在实际情况下遇到问题的可选择方案。