参考资料：

https://www.cnblogs.com/dhcn/p/10264208.html

数据分布是分布式存储系统的一个重要部分，数据分布算法至少要考虑以下三个因素：

1）故障域隔离。同份数据的不同副本分布在不同的故障域，降低数据损坏的风险；

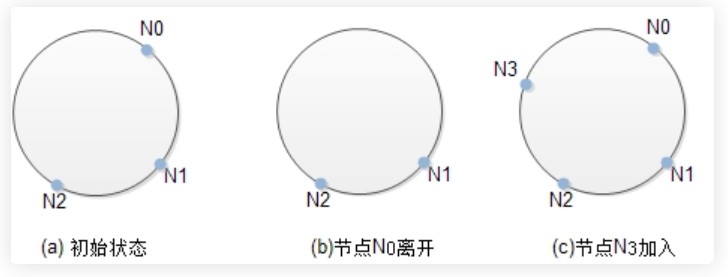
2）负载均衡。数据能够均匀地分布在磁盘容量不等的存储节点，避免部分节点空闲部分节点超载，从而影响系统性能；

3）控制节点加入离开时引起的数据迁移量。当节点离开时，最优的数据迁移是只有离线节点上的数据被迁移到其它节点，而正常工作的节点的数据不会发生迁移。

对象存储中一致性Hash和Ceph的CRUSH算法是使用地比较多的数据分布算法。在Aamzon的Dyanmo键值存储系统中采用一致性Hash算法，并且对它做了很多优化。OpenStack的Swift对象存储系统也使用了一致性Hash算法

1、一致性Hash算法

假设数据为x，存储节点数目为N。将数据分布到存储节点的最直接做法是，计算数据x的Hash值，并将结果同节点数目N取余数，余数就是数据x的目的存储节点。即目的存储节点为 Hash(x) % N。对数据计算Hash值的目的为了可以让数据均匀分布在N个节点中。这种做法的一个严重问题是，当加入新节点或则节点离开时，几乎所有数据都会受到影响，需要重新分布。因此，数据迁移量非常大。

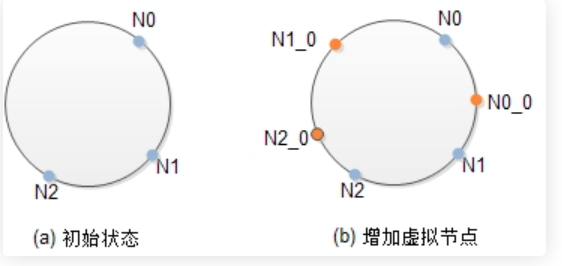


一致性Hash算法将数据和存储节点映射到同个Hash空间，如上图所示。Hash环中的3存储节点把Hash空间划分成3个分区，每个存储节点负责一个分区上的数据。例如，落在分区[N2,N0]上的数据存储在节点N0。

一致性Hash算法能够很好地控制节点加入离开导致的迁移数据的数量。如图(b)所示，当节点N0离开时，原来由它负责的[N2, N0]分区将同[N0, N1]分区合并成[N2, N1]分区，并且都由节点N1负责。也就是说，本来存储在节点N0上的数据都迁移到节点N1，而原来存储在N1和N2节点的数据不受影响。图(c)给出了当节点N3加入时，原来[N2, N0]分区分裂成[N3, N0]和[N2, N3]两个分区，其中[N3, N0]分区上是数据迁移到新加入的N3节点。

2、虚拟节点

一致性Hash的一个问题是，存储节点不能将Hash空间划分地足够均匀。如上图(a)所示，分区[N2, N0]的大小几乎是其它两个分区大小之和。这容易让负责该分区的节点N0负载过重。假设3个节点的磁盘容量相等，那么当节点N0的磁盘已经写满数据时其它两个节点上的磁盘还有很大的空闲空间，但此时系统已经无法继续向分区[N2, N0]写入数据，从而造成资源浪费。

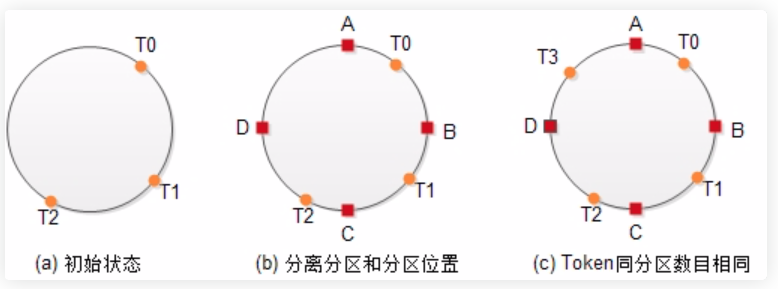


虚拟节点是相对于物理存储节点而言的，虚拟节点负责的分区上的数据最终存储到其对应的物理节点。在一致性Hash中引入虚拟节点可以把Hash空间划分成更多的分区，从而让数据在存储节点上的分布更加均匀。如上图(b)所示，黄颜色的节点代表虚拟节点，Ni\_0代表该虚拟节点对应于物理节点i的第0个虚拟节点。增加虚拟节点后，物理节点N0负责[N1\_0, N0]和[N0, N0\_0]两个分区，物理节点N1负责[N0\_0, N1]和[N2\_0, N1\_0]两个分区，物理节点N2负责[N2, N1]和[N2\_0, N2]两个分区，三个物理节点负责的总的数据量趋于平衡。

实际应用中，可以根据物理节点的磁盘容量的大小来确定其对应的虚拟节点数目。虚拟节点数目越多，节点负责的数据区间也越大。

3、分区与分区位置

前文提到，当节点加入或者离开时，分区会相应地进行分裂或合并。这不对新写入的数据构成影响，但对已经写入到磁盘的数据需要重新计算Hash值以确定它是否需要迁移到其它节点。因为需要遍历磁盘中的所有数据，这个计算过程非常耗时。如下图(a)所示，分区是由落在Hash环上的虚拟节点Ti来划分的，并且分区位置(存储分区数据的节点)也同虚拟节点相关，即存储到其顺时针方向的第1个虚拟节点。

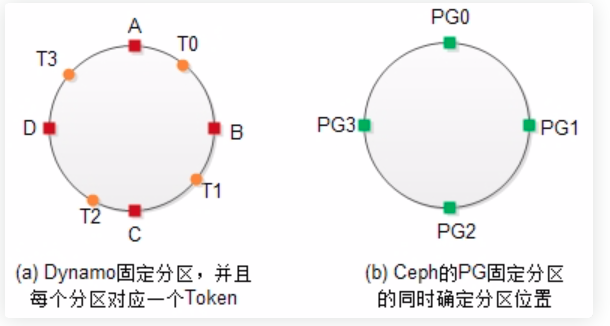


在Dynamo的论文中提出了分离分区和分区位置的方法来解决这个问题。该方法将Hash空间划分成固定的若干个分区，虚拟节点不再用于划分分区而用来确定分区的存储位置。如上图(b)所示，将Hash空间划分成[A,B]，[B,C], [C,D]和[D,A]四个固定的分区。虚拟节点用于确定分区位置，例如T1负责分区[B,C]，T2负责分区[C,D]，T0负责[D,A]和[A,B]两个分区。由于分区固定，因此迁移数据时可以很容易知道哪些数据需要迁移哪些数据不需要迁移。

上图(b)中虚拟节点T0负责了[D,A]和[A,B]两个分区的数据，这是由分区数目和虚拟节点数目不相同导致的。为让分区分布地更加均匀，Dyanmo提出了维持分区数目和虚拟节点数目相等的方法。这样每个虚拟节点负责一个分区，在物理节点的磁盘容量都相同并且虚拟节点数目都相同的情况下，每个物理节点负责的分区大小是完全相同的，从而可以达到最佳的数据分布。

4、CRUSH算法

Ceph分布数据的过程：首先计算数据x的Hash值并将结果和PG数目取余，以得到数据x对应的PG编号。然后，通过CRUSH算法将PG映射到一组OSD中。最后把数据x存放到PG对应的OSD中。这个过程中包含了两次映射，第一次是数据x到PG的映射。如果把PG当作存储节点，那么这和文章开头提到的普通Hash算法一样。不同的是，PG是抽象的存储节点，它不会随着物理节点的加入或则离开而增加或减少，因此数据到PG的映射是稳定的。



在这个过程中，PG起到了两个作用：第一个作用是划分数据分区。每个PG管理的数据区间相同，因而数据能够均匀地分布到PG上；第二个作用是充当Dyanmo中Token的角色，即决定分区位置。实际上，这和Dynamo中固定分区数目，以及维持分区数目和虚拟节点数目相等的原则是同一回事。

在没有多副本的情况下，Dynamo中分区的数据直接存储到Token，而每个Token对应唯一的一个物理存储节点。在多副本(假设副本数目为N)的情况下，分区的数据会存储到连续的N个Token中。但这会引入一个新问题：因为副本必须保持在不同的物理节点，但是如果这组Token中存在两个或多个Token对应到同个物理存储节点，那么就必须要跳过这样的节点。Dynamo采用Preference列表来记录每个分区对应的物理节点。然而，Dynmao论文中没有详述分区的Preference列表如何选取物理节点，以及选取物理节点时该如何隔离故障域等问题。

*(osd0, osd1, osd2 … osdn) = CRUSH(x)*

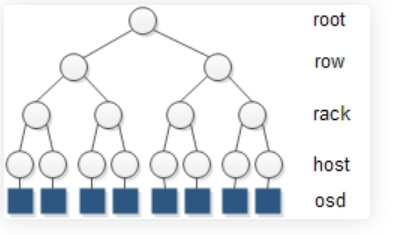
Ceph的PG担当起Dynamo中Token、固定分区以及Preference列表的角色，解决的是同样的问题。PG的Acting集合对应于Dynamo的Preference列表。CRUSH算法解决了Dynamo论文中未提及的问题。

5、OSD层级结构和权重大小

CRUSH算法的目的是，为给定的PG(即分区)分配一组存储数据的OSD节点。选择OSD节点的过程，要考虑以下几个因素：

1）PG在OSD间均匀分布。假设每个OSD的磁盘容量都相同，那么我们希望PG在每个OSD节点上是均匀分布的，也就是说每个OSD节点包含相同数目的PG。假如节点的磁盘容量不等，那么容量大的磁盘的节点能够处理更多数量的PG。

2）PG的OSD分布在不同的故障域。因为PG的OSD列表用于保存数据的不同副本，副本分布在不同的OSD中可以降低数据损坏的风险。

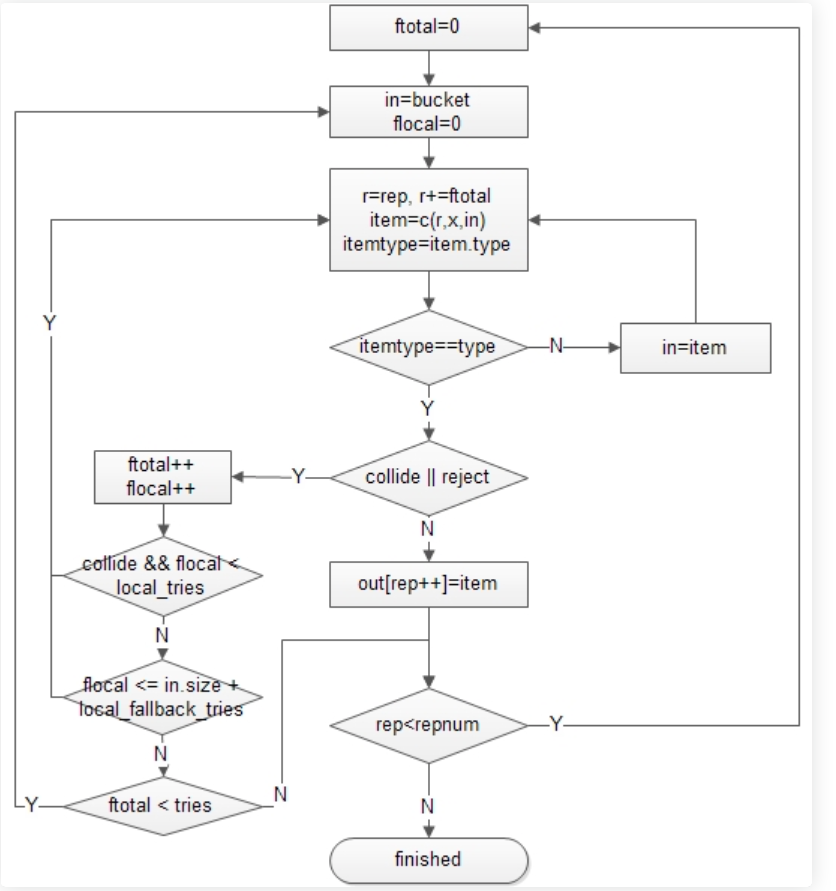


Ceph使用树型层级结构描述OSD的空间位置以及权重(同磁盘容量相关)大小。如上图所示，层级结构描述了OSD所在主机、主机所在机架以及机架所在机房等空间位置。这些空间位置隐含了故障区域，例如使用不同电源的不同的机架属于不同的故障域。CRUSH能够依据一定的规则将副本放置在不同的故障域。

OSD节点在层级结构中也被称为Device，它位于层级结构的叶子节点，所有非叶子节点称为Bucket。Bucket拥有不同的类型，如上图所示，所有机架的类型为Rack，所有主机的类型为Host。使用者还可以自己定义Bucket的类型。Device节点的权重代表存储节点的性能，磁盘容量是影响权重大小的重要参数。Bucket节点的权重是其子节点的权重之和。

CRUSH通过重复执行Take(bucketID)和Select(n, bucketType)两个操作选取副本位置。Take(bucketID)指定从给定的bucketID中选取副本位置，例如可以指定从某台机架上选取副本位置，以实现将不同的副本隔离在不同的故障域; Select(n, bucketType)则在给定的Bucket下选取n个类型为bucketType的Bucket，它选取Bucket主要考虑层级结构中节点的容量，以及当节点离线或者加入时的数据迁移量。

crush算法流程：



bucket: Take操作指定的bucket；

type: Select操作指定的Bucket的类型；

repnum: Select操作指定的副本数目；

rep：当前选择的副本编号；

x: 当前选择的PG编号；

item: 代表当前被选中的Bucket；

c(r, x, in): 代表从Bucket in中为PG x选取第r个副本；

collide: 代表当前选中的副本位置item已经被选中，即出现了冲突；

reject: 代表当前选中的副本位置item被拒绝，例如，在item已经处于out状态的情况下；

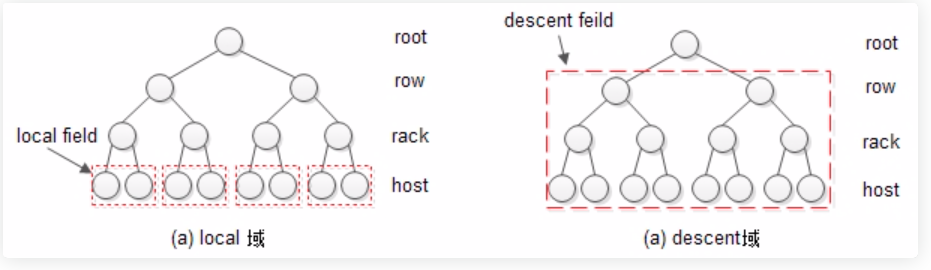
ftotal: 在Descent域中选择的失败次数，即选择一个副本位置的总共的失败次数；

flocal: 在Local域中选择的失败次数；

local\_retries: 在Local域选择冲突时的尝试次数；

local\_fallback\_retries: 允许在Local域的总共尝试次数为bucket.size + local\_fallback\_retires次，以保证遍历完Buckt的所有子节点；

tries: 在Descent的最大尝试次数，超过这个次数则放弃这个副本。



当Take操作指定的Bucket和Select操作指定的Bucket类型之间隔着几层Bucket时，算法直接深度优先地进入到目的Bucket的直接父母节点。例如，从根节点开始选择N个Host时，它会深度优先地查找到Rack类型的节点，并在这个节点下选取Host节点。为了方便表述，将Rack的所有子节点标记为Local域，将Take指定的Bucket的子节点标记为Descent域，如上图所示。

选取过程中出现冲突、过载或者故障时，算法先在Local域内重新选择，尝试有限次数后，如果仍然找不到满足条件的Bucket，那就回到Descent域重新选择。每次重新选择时，修改副本数目为r += ftotal。因此每次选择失败都会递增ftotal，所以可以尽量避免选择时再次选到冲突的节点。

6、Bucket选取Item算法

流程图中的item=c(r,x,in)从给定的Bucket in中选取一个子节点。