分布式哈希算法

一，普通的 Hash 方式

在介绍分布式哈希算法之前，先了解下普通的 Hash 是如何实现的。JDK 中的 java.util.HashMap 类就实现了一个哈希表，它的特点有：

①创建哈希表(HashMap)**需要先指定大小**，即默认创建一个能够存储多少个元素的哈希表，它的默认大小为 16。

②当不断地向 HashMap 中添加元素时，HashMap 越来越满，当添加的元素达到了装载因子乘以表长时，就需要扩容了。扩容时，原来已经映射到哈希表中的某个位置(桶)的元素需要重新再哈希，然后再把原来的数据复制到新的哈希表中。

对于普通的哈希表而言，扩容的代价是很大的。因为普通的 Hash 计算地址方式如下：Hash(Key)%M，为了演示方便，举个极端的例子如下：

假设哈希函数为 hash(x)=x ，哈希表的长度为 5(有 5 个桶)

key=6 时，hash(6)%5 = 1，即 Key 为 6 的元素存储在第一个桶中

key=7 时，hash(7)%5 = 2，即 Key 为 7 的元素存储在第二个桶中

Key=13 时，hash(13)%5=3，即 Key 为 13 的元素存储在第三个桶中....

假设现在 hash 表长度扩容成 8，那么 Key 为 6，7，13 的数据全都需要重新哈希。因为，

key=6 时，hash(6)%8 = 6，即 Key 为 6 的元素存储在第六个桶中

key=7 时，hash(7)%8 = 7，即 Key 为 7 的元素存储在第七个桶中

Key=13 时，hash(13)%8=5，即 key 为 13 的元素存储在第五个桶中....

从上可以看出：扩容之后，元素的位置全变了。比如：Key 为 6 的元素原来存储在第一个桶中，扩容之后需要存储到第 6 个桶中。

因此，这是普通哈希的一个不足：扩容可能会影响到所有元素的移动。这也是为什么：为了减少扩容时元素的移动，总是将哈希表扩容成原来大小的两倍的原因。因为，有数学证明，扩容成两倍大小，使得再哈希的元素个数最少。

二，一致性哈希方式

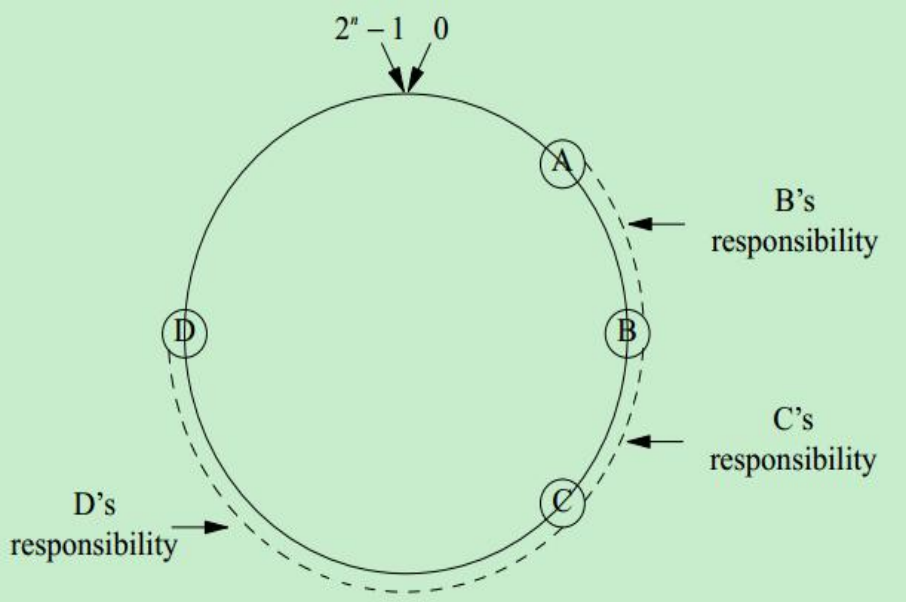
在分布式系统中，节点的宕机、某个节点加入或者移出集群是常事。对于分布式存储而言，假设存储集群中有 10 台机子，如果采用 Hash 方式对数据分片(即将数据根据哈希函数映射到某台机器上存储)，哈希函数应该是这样的：hash(file) % 10。根据上面的介绍，扩容是非常不利的,如果要添加一台机器,很多已经存储到机器上的文件都需要重新分配移动，如果文件很大，这种移动就造成网络的负载。

因此，就出现了一种一致性哈希的方式。关于一致性哈希，可参考：一致性哈希算法学习及 JAVA 代码实现分析

一致性哈希，其实就是把哈希函数可映射的空间（相当于普通哈希中桶的数目是固定的）**固定**下来了，比如固定为：2^n-1，并组织成环的形状

每个机器对应着一个 n 位的 ID，并且映射到环中。每个查询键，也是一个 n 位的 ID，节点的 ID 和查询键对应着相同的映射空间。

如下图：有四台机器映射到**固定大小**为 2^n-1 的哈希环中。四台机器一共将整个环分成了四部分：（A，B）、（B，C）、（C，D）、（D，A）



机器 A 负责存储落在（D，A）范围内的数据，机器 B 负责存储落在（A，B）范围内的数据.....

也就是说，对数据进行 Hash 时，数据的地址会落在环上的某个点上，数据就存储 该点的 顺时针方向上的那台机器上。

相比于普通哈希方式，**一致性哈希的好处就是：当添加新机器或者删除机器时，不会影响到全部数据的存储，而只是影响到这台机器上所存储的数据（落在这台机器所负责的环上的数据）。**

比如，B 机器被移除了，那落在（A，B）范围内的数据 全部需要由 C 机器来存储，也就只影响到落在（A，B）这个范围内的数据。

同时，扩容也很方便，比如在（C，D）这段环上再添加一台机器 E，只需要将 D 上的一部分数据拷贝到机器 E 上即可。

那一致性哈希有没有缺点呢？当然是有的。总结一下就是：没有考虑到每台机器的异构性质，不能实现很好的负载均衡。

举个例子：机器 C 的配置很高，性能很好，而机器 D 的配置很低。但是，可能 现实中 大部分数据由于某些特征 都哈希到（C，D）这段环上，直接就导致了机器 D 的存储压力很大。

另外，一致性哈希还存在“热点”问题（hotspot）。

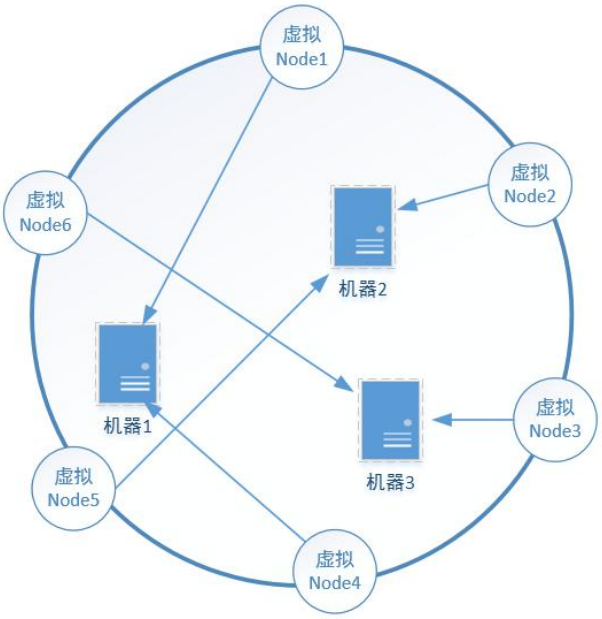
比如说，由于机器 D 上存储了大量的数据，为了缓解下机器 D 的压力，在 环（C,D）段再添加一台机器 E，就需要将机器 D 上的一部分数据拷贝到机器 E 上。而且**只有一台机器：即机器 D 参与拷贝**，这会引起机器 D 与 机器 E 之间的网络负载突然增大。机器 D，就是我们所说的 hotspot。

三，引入虚拟节点的一致性哈希方式

为了解决一致性哈希的一些不足，又引入了虚拟节点的概念。

引入虚拟节点，可以有效地防止物理节点(机器)映射到哈希环中出现不均匀的情况。比如上图中的机器 A、B、C 都映射在环的右半边上。

一般，虚拟节点会比物理节点多很多，并可均衡分布在环上，从而可以提高负载均衡的能力。如下图：



①如果虚拟机器与物理机器映射得好，某一台物理机器宕机后，其上的数据可由其他若干台物理机器共同分担。

②如果新添加一台机器，它可以对应多个**不相邻** 环段 上的虚拟节点，从而使得 hash 的数据存储得更分散。

DHT 的路由机制

基本路由（简单遍历）

当收到请求（key），先看 key 是否在自己这里。如果在自己这里，就直接返回信息；否则就把 key 转发给自己的继任者。以此类推。

这种玩法的时间复杂度是：O(N)。对于一个节点数很多的 DHT 网络，这种做法显然【非常低效】。

高级路由（Finger Table）

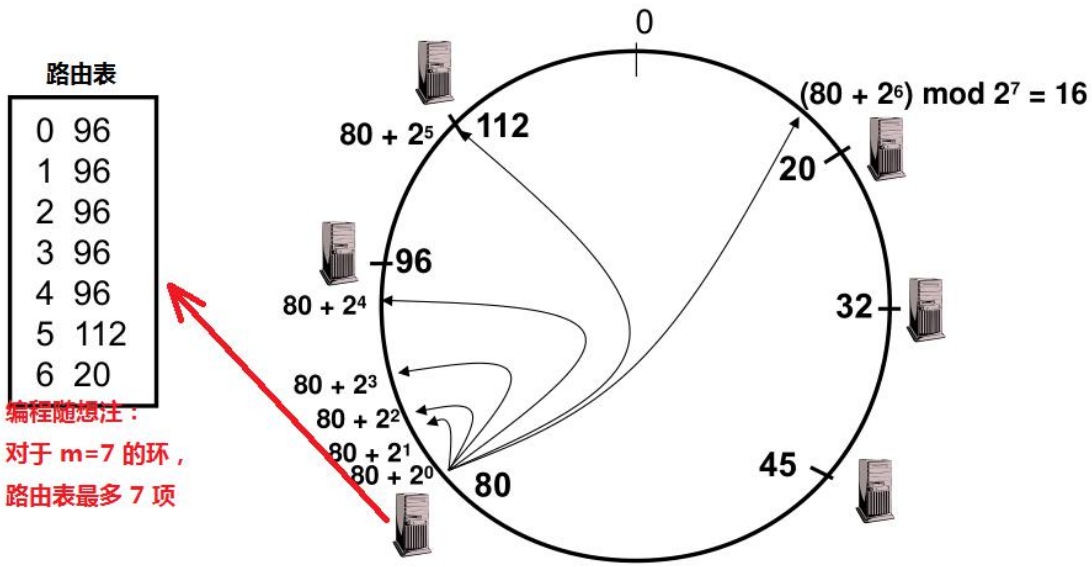
由于“基本路由”非常低效，自然就引入更高级的玩法——基于“Finger Table”的路由。

“Finger Table”是一个列表，最多包含 m 项（m 就是散列值的比特数），每一项都是节点 ID。

假设当前节点的 ID 是 n，那么表中第 i 项的值是：successor（（n + 2i）mod 2m ）

当收到请求（key），就到“Finger Table”中找到【最大的且不超过 key】的那一项，然后把 key 转发给这一项对应的节点。

有了“Finger Table”之后，时间复杂度可以优化为：O(log N)。



◇节点的加入

任何一个新来的节点（假设叫 A），需要先跟 DHT 中已有的任一节点（假设叫 B）建立连接。

A 随机生成一个散列值作为自己的 ID（对于足够大的散列值空间，ID 相同的概率忽略不计）

A 通过跟 B 进行查询，找到自己这个 ID 在环上的接头人。也就是——找到自己这个 ID 对应的“继任”（假设叫 C）与“前任”（假设叫 D）

接下来，A 需要跟 C 和 D 进行一系列互动，使得自己成为 C 的前任，以及 D 的继任。

这个互动过程，大致类似于在双向链表当中插入元素（考虑到篇幅，此处省略 XXX 字）。

◇节点的【正常】退出

如果某个节点想要主动离开这个 DHT 网络,按照约定需要作一些善后的处理工作。比如说，通知自己的前任去更新其继任者.....

这些善后处理，大致类似于：在双向链表中删除元素（考虑到篇幅，此处省略 XXX 字）。

◇节点的【异常】退出

作为一个分布式系统，任何节点都有可能意外下线（也就是说，来不及进行善后就挂掉了）

假设 节点 A 的继任者【异常】下线了，那么 节点 A 就抓瞎了。咋办捏？

为了保险起见，Chord 引入了一个“继任者候选列表”的概念。每个节点都用这个列表来包含：距离自己最近的 N 个节点的信息，顺序是【由近到远】。一旦自己的继任者下线了，就在列表中找到一个【距离最近且在线】的节点，作为新的继任者。然后 节点 A 更新该列表，确保依然有 N 个候选。更新完“继任者候选列表”后，节点 A 也会通知自己的前任，那么 A 的前任也就能更新自己的“继任者候选列表”。