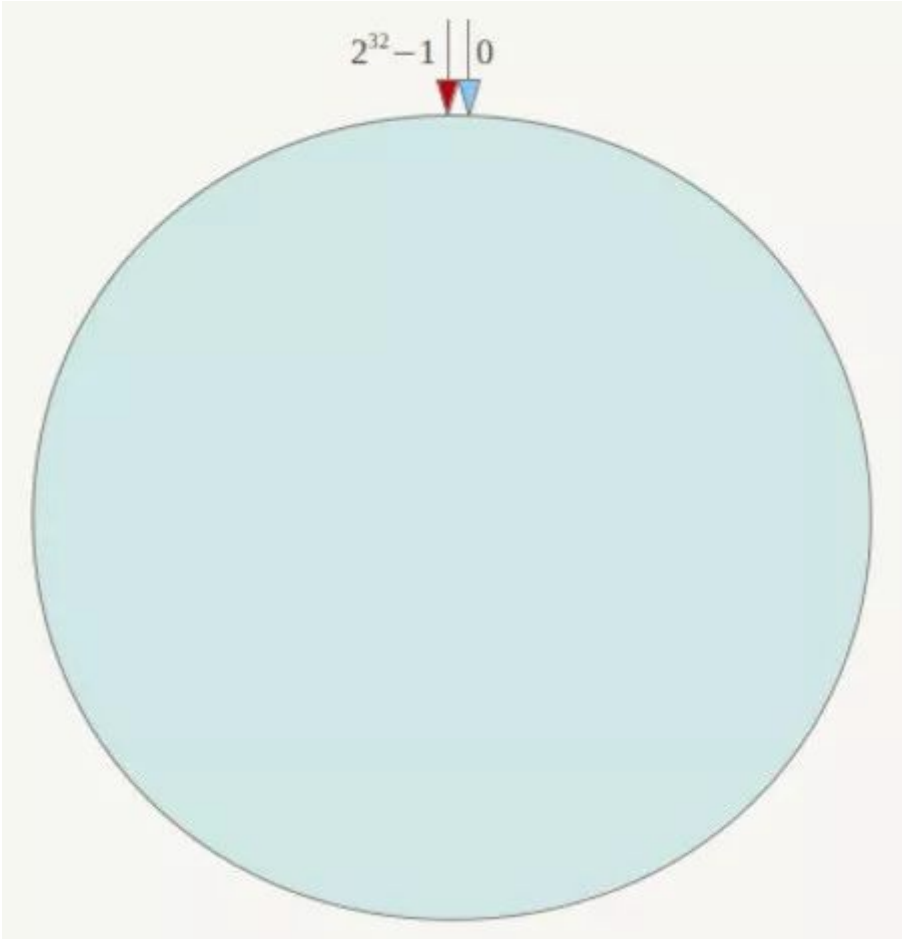
一致性Hash算法

一致性哈希最早由 MIT的 Karger 提出，在发表于1997年的论文Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web， Karger et al 和合作者们提出了一致性哈希的概念(consistent hash)，用来解决分布式Cache的问题。这篇论文中提出在动态变化的Cache环境中，哈希算法应该满足的4个适应条件：：Balance(均衡)、Monotonicity(单调性)、Spread(分散性)、Load(负载)。
在分布式缓存系统中使用一致性哈希算法时，某个节点的添加和移除不会重新分配全部的缓存，而只会影响小部分的缓存系统，如果均衡性做的好的话，当添加一个节点时，会均匀地从其它节点移一部分缓存到新的节点上；当删除一个节点的时候，这个节点上的缓存会均匀地分配到其它活着的节点上。

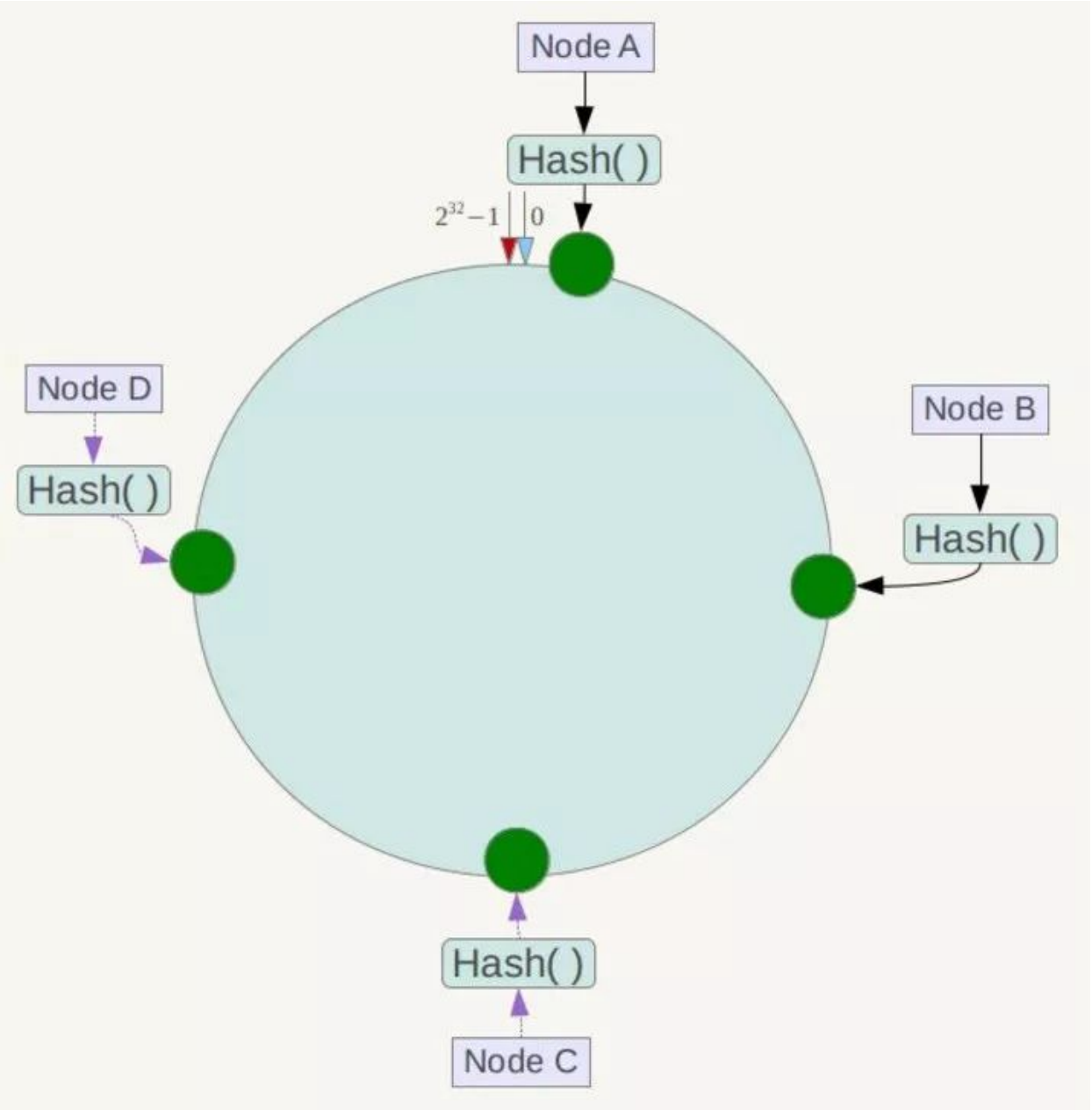
Karger 一致性哈希算法将每个节点(bucket)关联一个圆环上的一些随机点，对于一个键值，将其映射到圆环中的一个点上，然后按照顺时针方向找到第一个关联bucket的点，将值放入到这个bucke中。

对2^32取模，一致性Hash算法将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^32-1（即哈希值是一个32位无符号整形），整个哈希环如下：



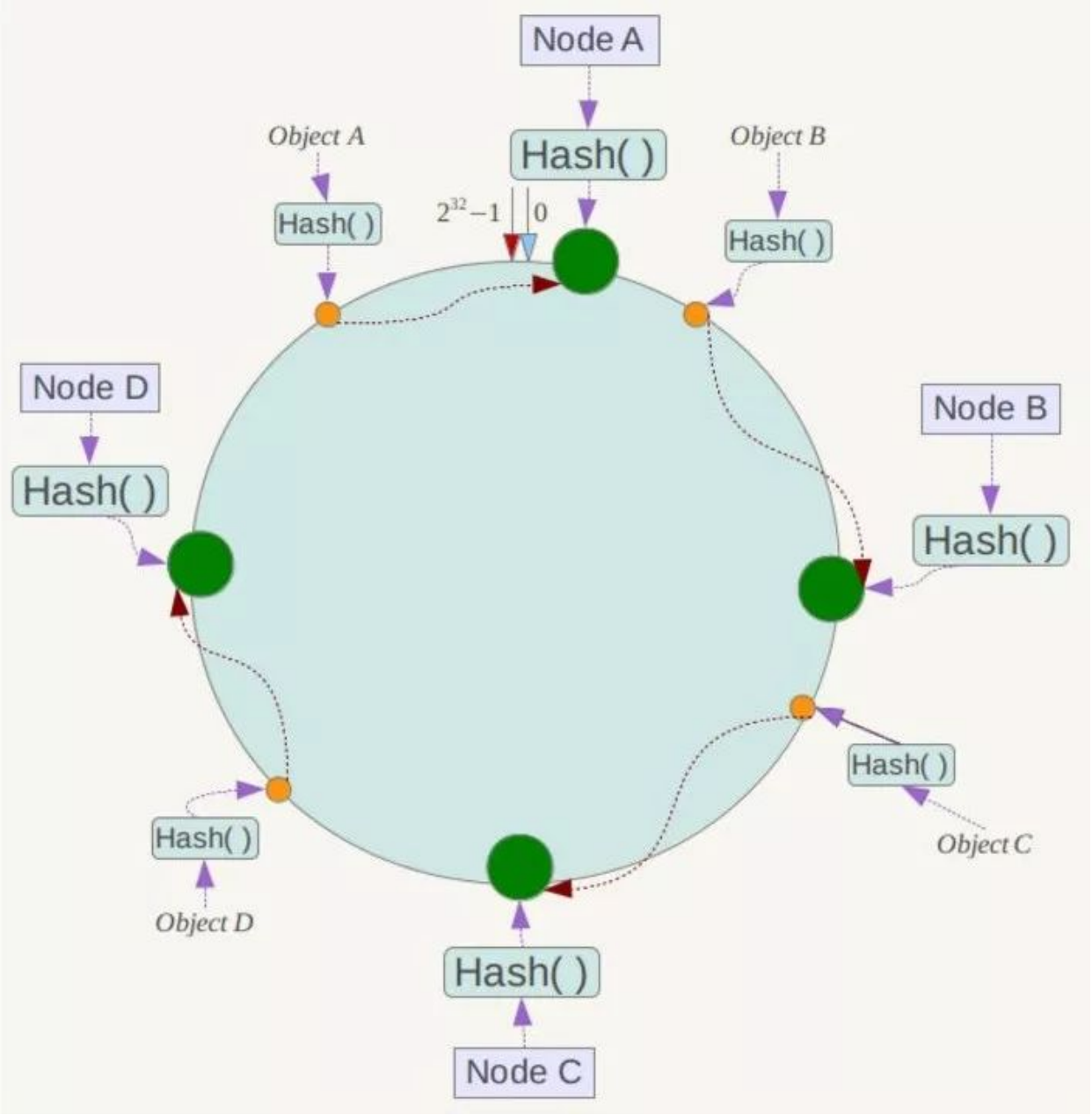
整个空间按**顺时针方向组织**，圆环的正上方的点代表0，0点右侧的第一个点代表1，以此类推，2、3、4、5、6……直到2^32-1，也就是说0点左侧的第一个点代表2^32-1， 0和2^32-1在零点中方向重合，这个由2^32个点组成的圆环称为**Hash环**。

将各个服务器使用Hash进行一个哈希，具体可以选择**服务器的IP或主机名**作为**关键字**进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将四台服务器使用IP地址哈希后在环空间的位置如下：



将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器！

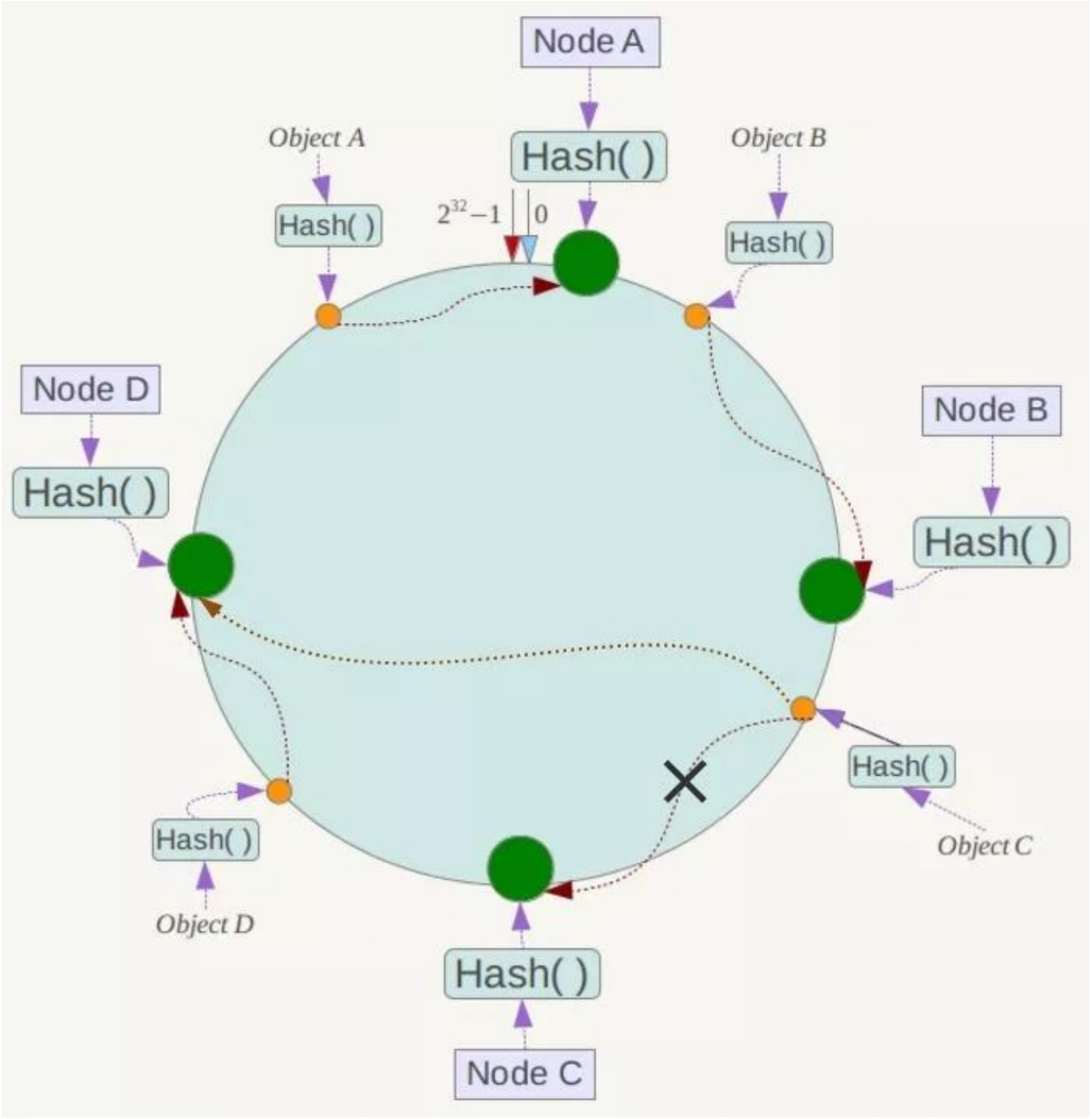
设有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



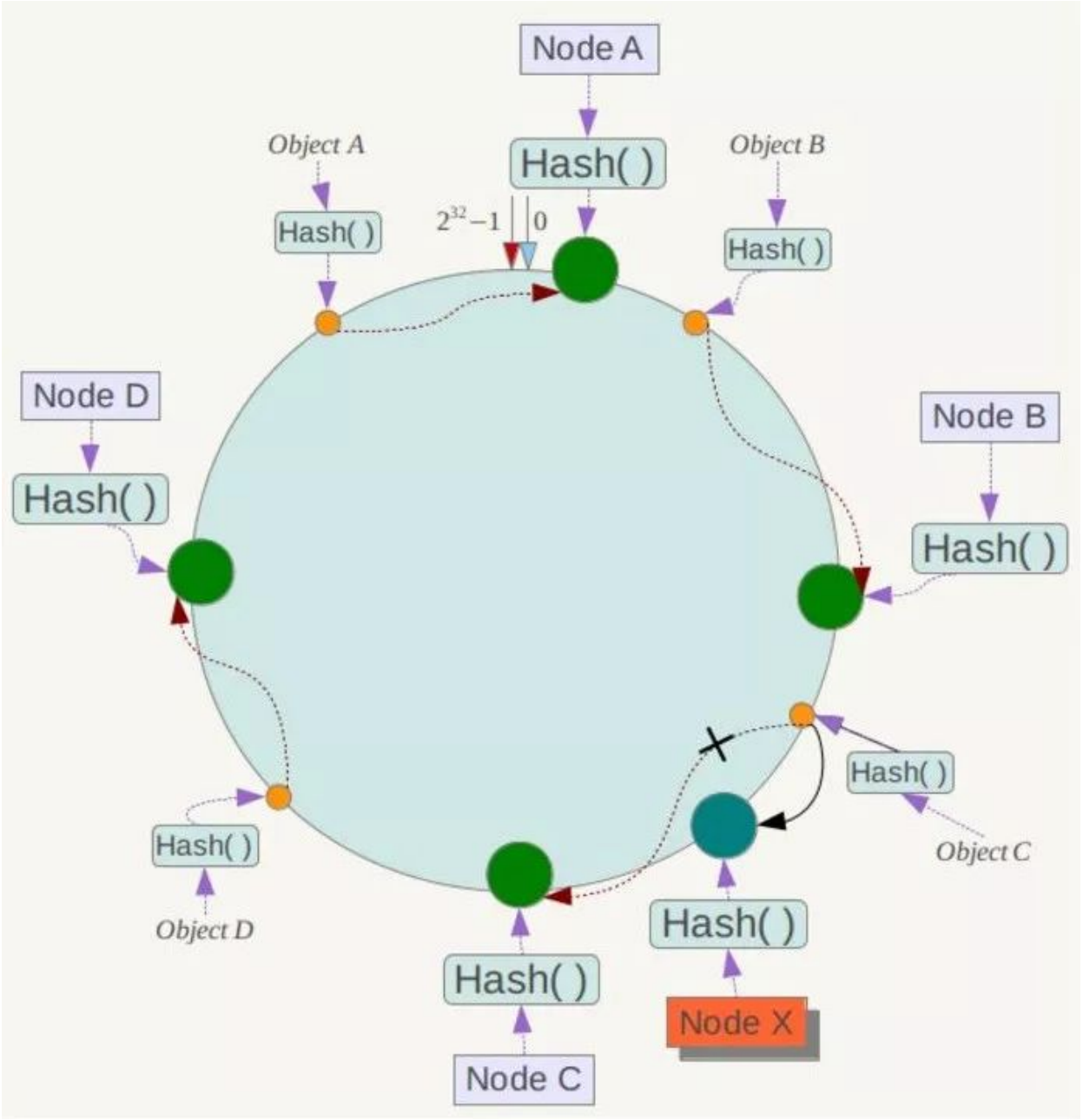
根据一致性Hash算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

一致性Hash算法的容错性和可扩展性

假设Node C宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性Hash算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响，如下所示：



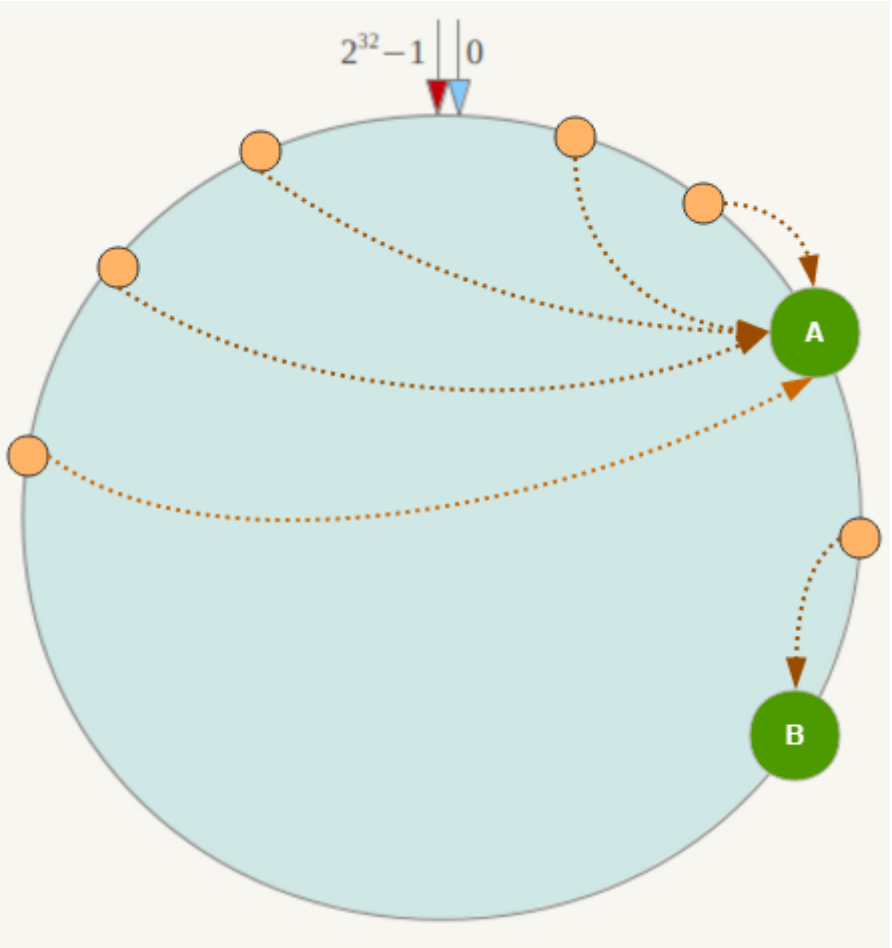
如果在系统中增加一台服务器Node X，如下图所示：



对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X ！一般的，在一致性Hash算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

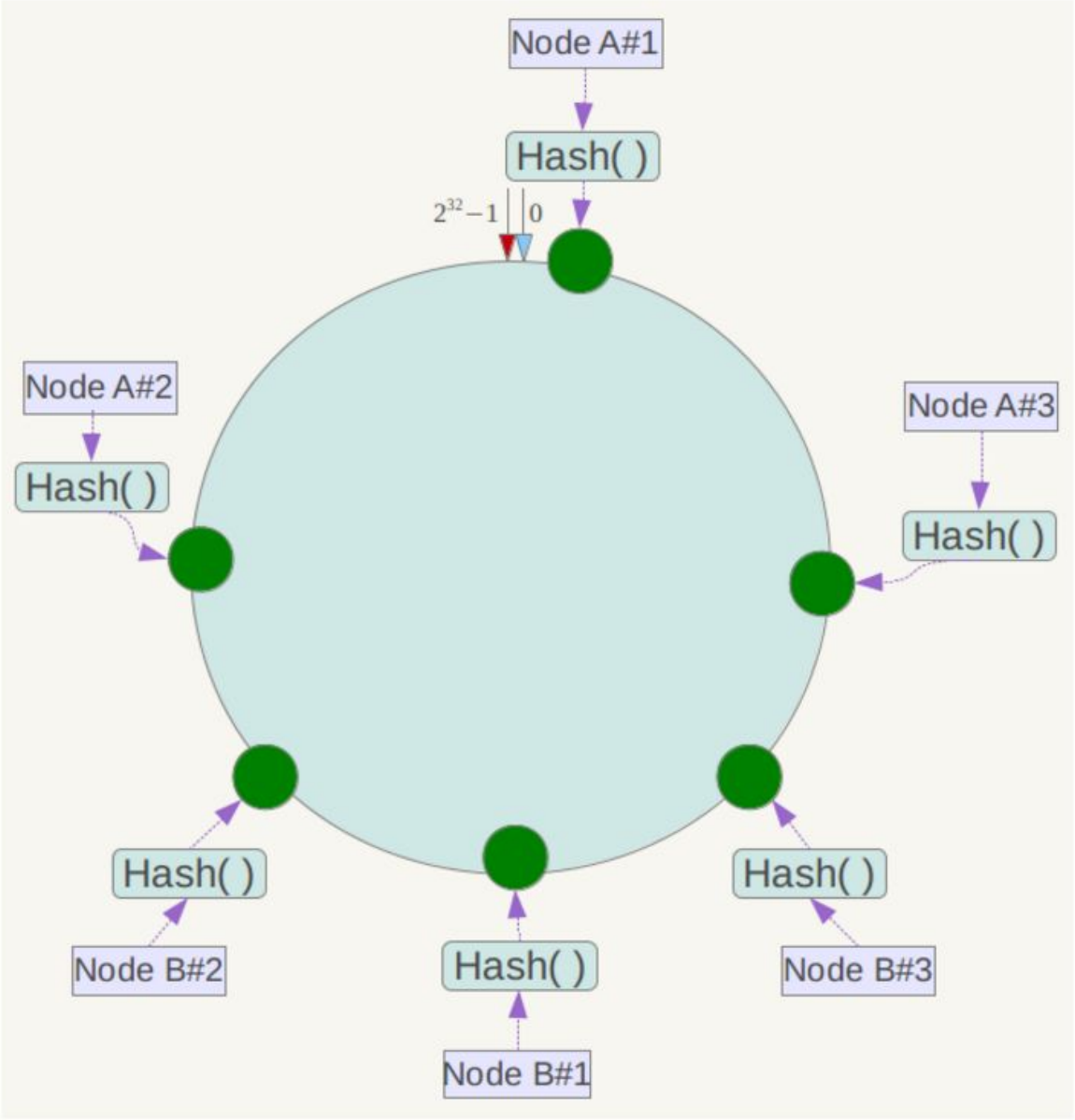
Hash环的数据倾斜问题

一致性Hash算法在**服务节点太少时**，容易因为节点分部不均匀而造成**数据倾斜**（被缓存的对象大部分集中缓存在某一台服务器上）问题，例如系统中只有两台服务器，其环分布如下：



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性Hash算法引入了**虚拟节点机制**，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为**虚拟节点**。具体做法可以在服务器IP或主机名的后面增加编号来实现。

可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点



数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

其他一致性哈希算法

Rendezvous hashing, 计算一个key应该放入到哪个bucket(节点)时，它使用哈希函数h(key,bucket)计算每个候选bucket的值，然后返回值最大的bucket。buckets比较多的时候耗时也较长，有人也提出了一些改进的方法，比如将bucket组织成tree的结构，但是在reblance的时候花费时间又长了。

Ketama算法。其实Ketama算法最早于2007年用c 实现(libketama)，很多其它语言也实现了相同的算法，它是基于Karger 一致性哈希算法实现：

1.建立一组服务器的列表 (如: 1.2.3.4:11211, 5.6.7.8:11211, 9.8.7.6:11211)
2.为每个服务器节点计算一二百个哈希值
3.从概念上讲，这些数值被放入一个环上(continuum). (想象一个刻度为 0 到 2^32的时钟，这个时钟上就会散落着一些数字)
4.每一个数字关联一个服务器，所以服务器出现在这个环上的一些点上，它们是哈希分布的
5.为了找个一个Key应该放入哪个服务器，先哈希key，得到一个无符号整数, 沿着圆环找到和它相邻的最大的数，这个数对应的服务器就是被选择的服务器
6.对于靠近 2^32的 key, 因为没有超过它的数字点，按照圆环的原理，选择圆环中的第一个服务器。

上两种算法可以处理节点增加和移除的情况。对于分布式存储系统，当一个节点失效时，并不期望它被移除，而是使用备份节点替换它，或者将它恢复起来，因为不期望丢掉它上面的数据。对于这种情况(节点可以扩容，但是不会移除节点)，Google的John Lamping, Eric Veach提供一个高效的几乎不占用持久内存的算法：Jump Consistent Hash。

```
int32_t JumpConsistentHash(uint64_t key, int32_t num_buckets)
{
    int64_t b = -1, j = 0;

    while (j < num_buckets) {
        b = j;
        key = key * 2862933555777941757ULL + 1;
        j = (b + 1) * (double(1LL << 31)/double((key >> 33) + 1));
    }
    return b;
}
```

因为Jump consistent hash算法不使用节点挂掉，如果有这种需求，比如要做一个缓存系统，可以考虑使用ketama算法，或者对Jump consistent hash算法改造一下：节点挂掉时不移除节点，只是标记这个节点不可用。当选择节点时，如果选择的节点不可用，则再一次Hash，尝试选择另外一个节点，比如将key加1再进行选择。(此时没

有设定重试的测试，如果所有的节点都挂掉，则会进入死循环，所以最好设置重试次数(递归次数)，超过n次没有选择到则返回失败。)