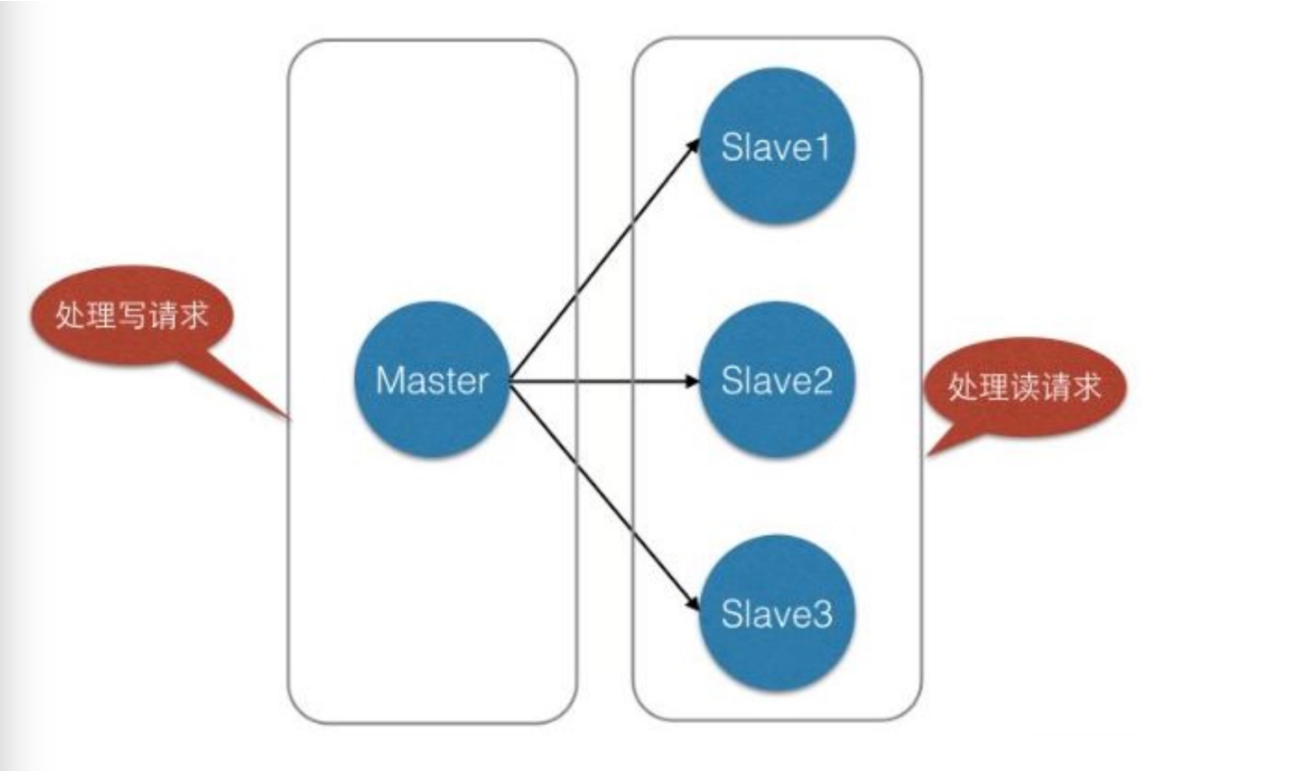
一致性hash算法的来源：

我们在使用redis的时候，为了保证redis的高可用，提高redis的读写性能，最简单的方式我们会做主从复制，组成Master-Master或者Master-Slave的形式，或者搭建redis集群，进行数据的读写分离，类似于数据库的主从复制和读写分离。如下所示：



同样类似于数据库，当单表数据大于500w的时候需要对其进行分库分表，当数据量很大的时候，我们同样可以对redis进行类似的操作，就是分库和分表。

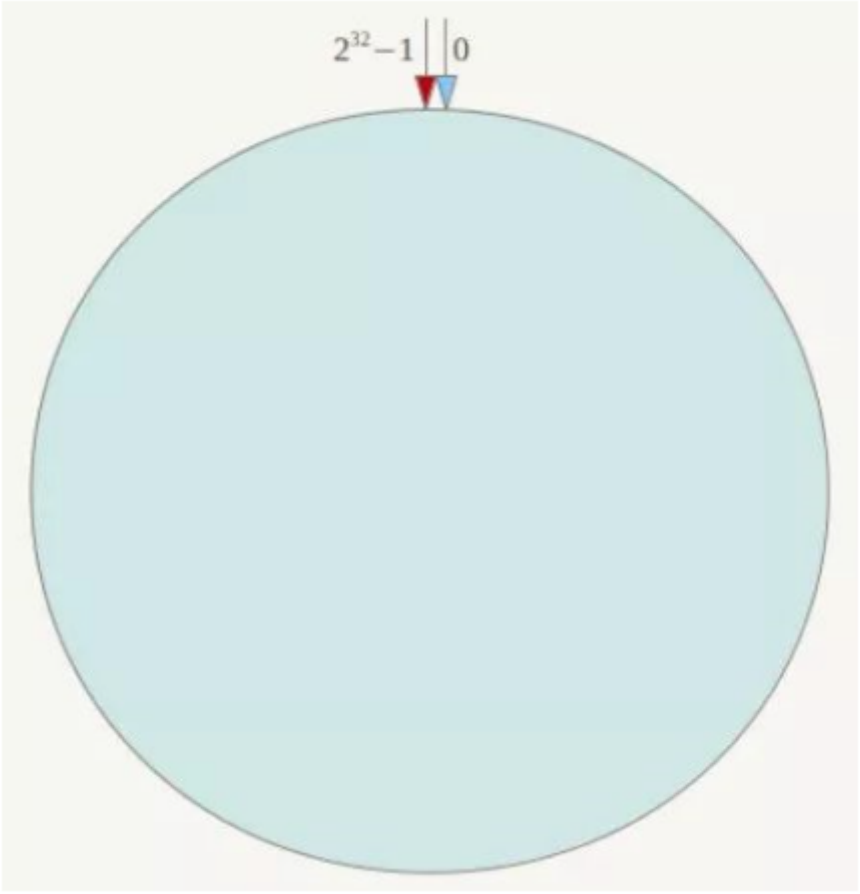
假如我们现在有4台机器，有个照片a.png位于第2台机器上，现在定的hash公式为hash(a.png)%4=2，可以通过hash算法直接将位置定在第二台机器上，但是上述hash算法进行缓存时，会出现一些缺陷，主要体现在服务器数量变动时，所有的缓存位置都发生了变化。

试想一下，如果4台缓存服务器已经不能满足我们的缓存需求，最简单我们增加了一台缓存服务器，那么缓存服务器的数量由4台变成了5台。那么原本hash(a.png)%4=2变成了hash(a.png) % 5 = ?，可想而知这个结果肯定不是2，这种情况带来的结果就是当服务器数量变动时，所有缓存的位置都要发生改变。假设4台缓存中突然有一台缓存服务器出现了故障，无法进行缓存，那么我们则需要将故障机器移除，但是如果移除了一台缓存服务器，那么服务器数量从4台变为了3台，也是会出现上述的问题。

所以，我们应该想办法不让这种情况发生，但是由于上述Hash算法本身的缘故，使用取模法进行缓存时，这种情况是无法避免的，为了解决这些问题，Hash一致性算法(一致性Hash算法)诞生了。

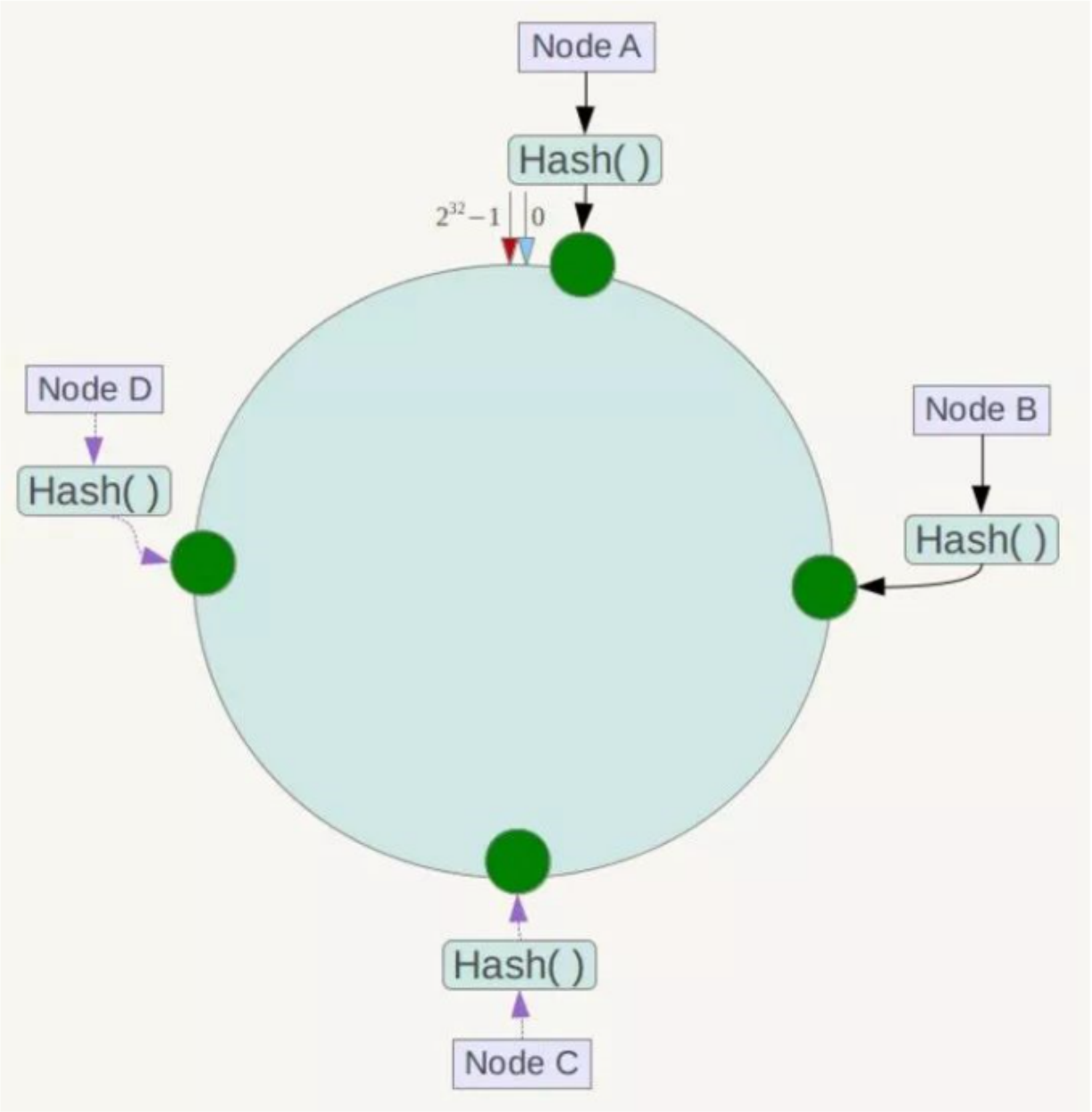
一致性Hash算法：

一致性Hash算法也是使用取模的方法，只是，刚才描述的取模法是对服务器的数量进行取模，而一致性Hash算法是对2^32取模。简单来说，一致性Hash算法将整个hash哈希值空间组织成一个虚拟的圆环，如假设谋 hash函数H的值空间为0～2^32-1，（即哈希值是一个32位无符号整数），整个哈希环如下：



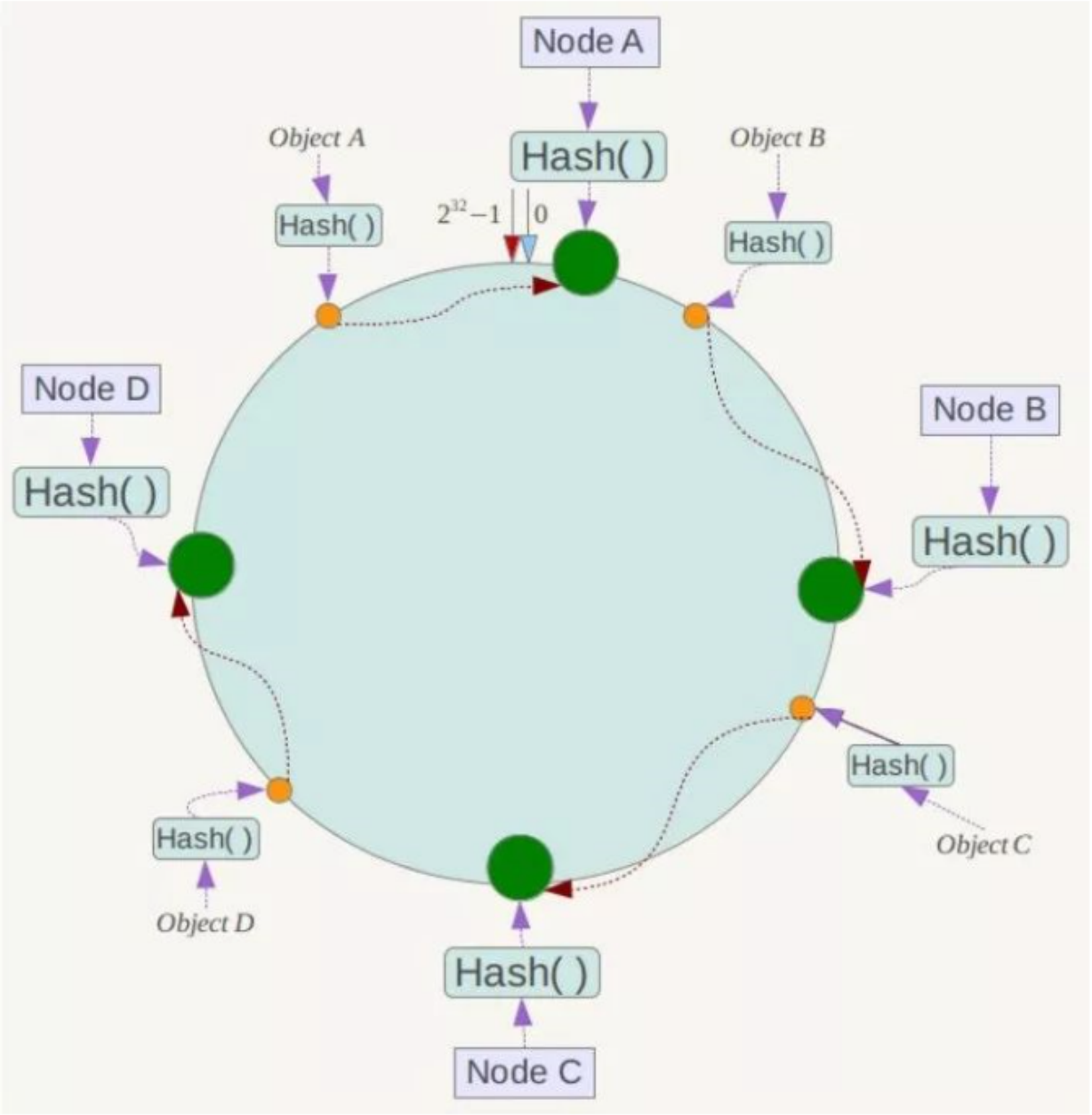
整个空间按顺时针方向组织，圆环的正上方的点代表0，0点右侧的第一个点代表1，以此类推，2、3、4、5、6....直到2^32-1，也就是说0点左侧的第一个点代表2^32-1，0和2^32-1在零点中方向重合，我们把这个由2^32个点组成的圆环称为Hash环。

下一步将各个服务器使用hash算法进行一个hash计算，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，这里假设将上文中四台服务器使用IP地址哈希后空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在换上的位置，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。

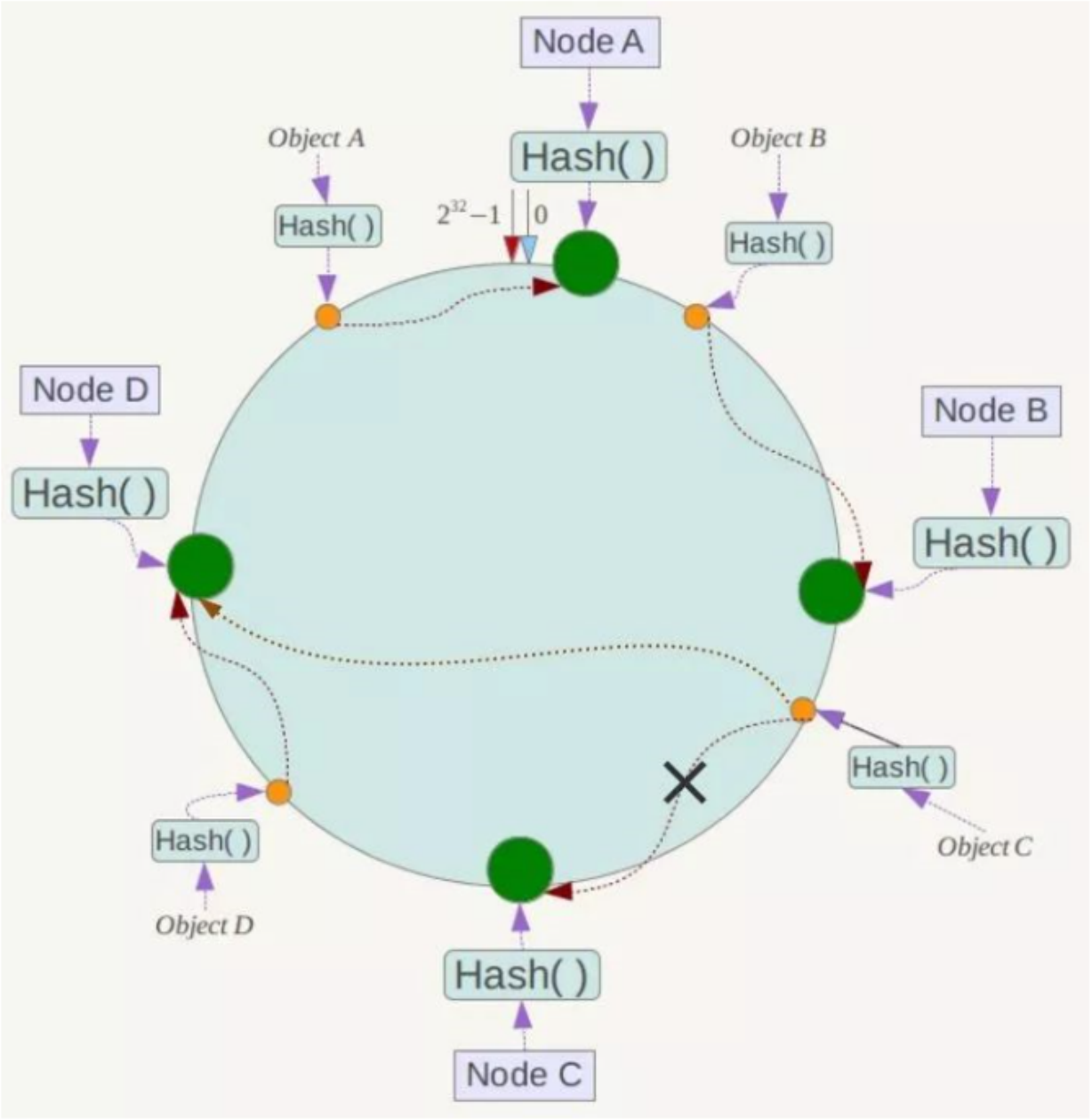
例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



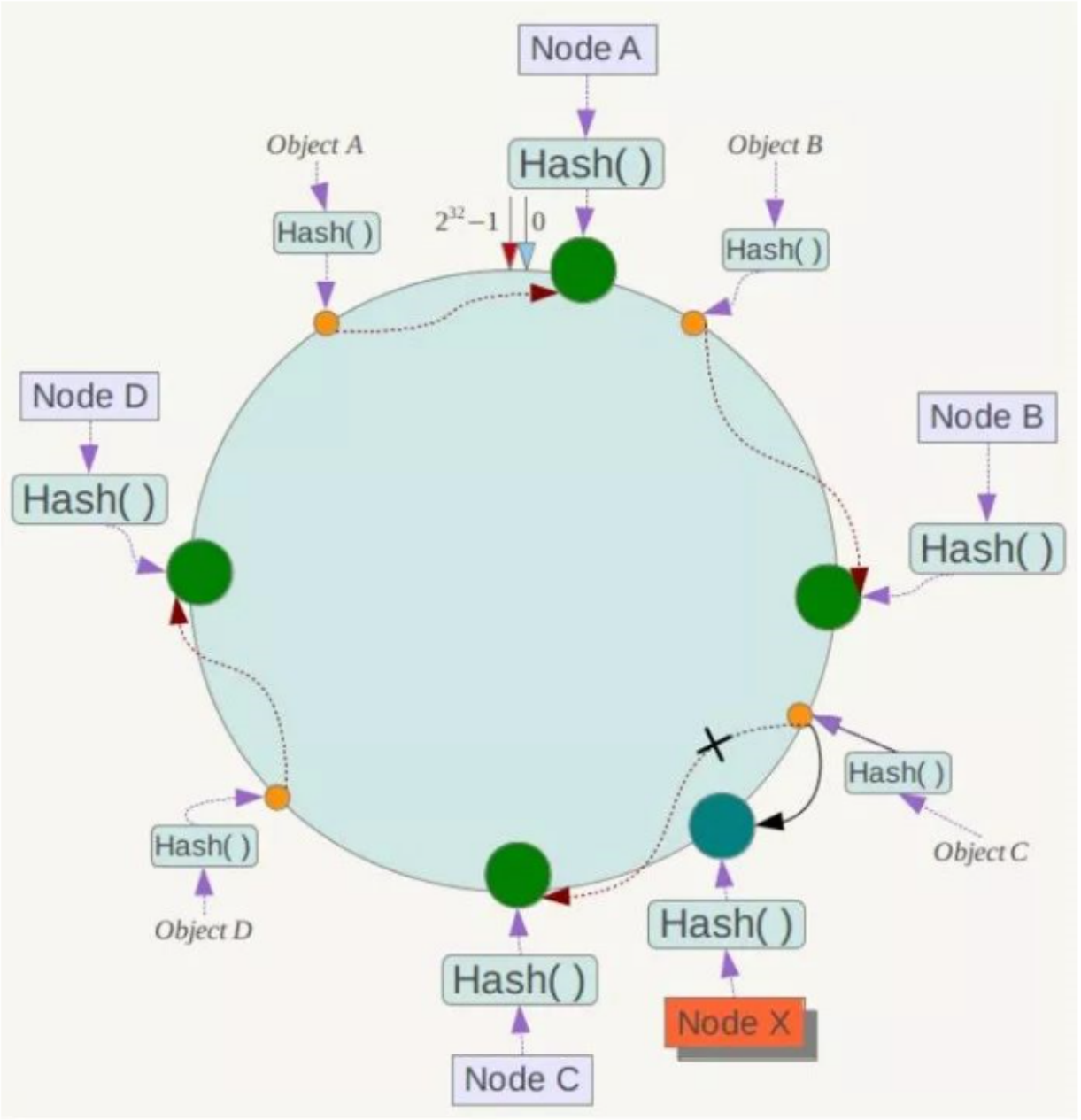
根据一致性Hash算法，数据A会被定位Node A上，B被定位Node B上，C被定位到Node C上，D被定位到Node D上。

一致性Hash算法的容错性和可扩展性

现假设Node C不幸宕机，可以看到此时对象A、B、D不会受影响， 只有C对象被重定位到Node D。一般的，在一致性Hash算法中，如果有一台服务器不可用，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，其它不会受影响，如下所示：



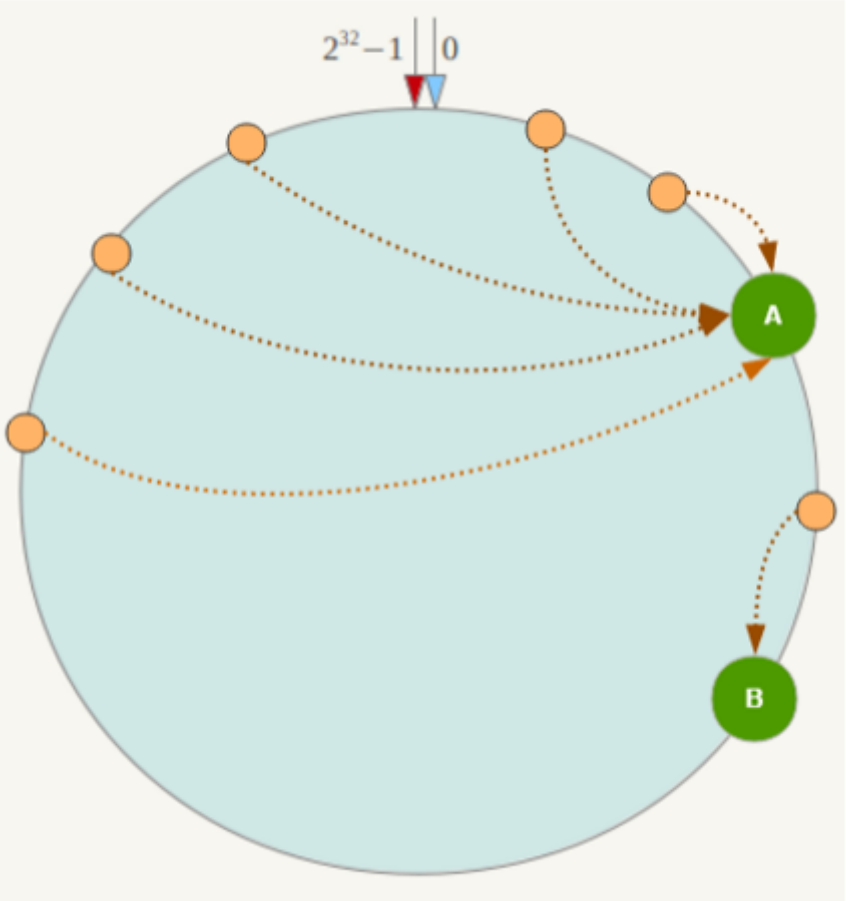
下面需要考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



此时对象Object A、B、D不受影响，只有对象C需要定位到新的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(即沿着逆时针方向走遇到的第一台服务器)之间数据，其他数据不会受到影响。
综上所述，一致性Hash算法对于节点的增减都只需要重定位环空间中的一小部分数据，具有较好的容错行和可扩展性。

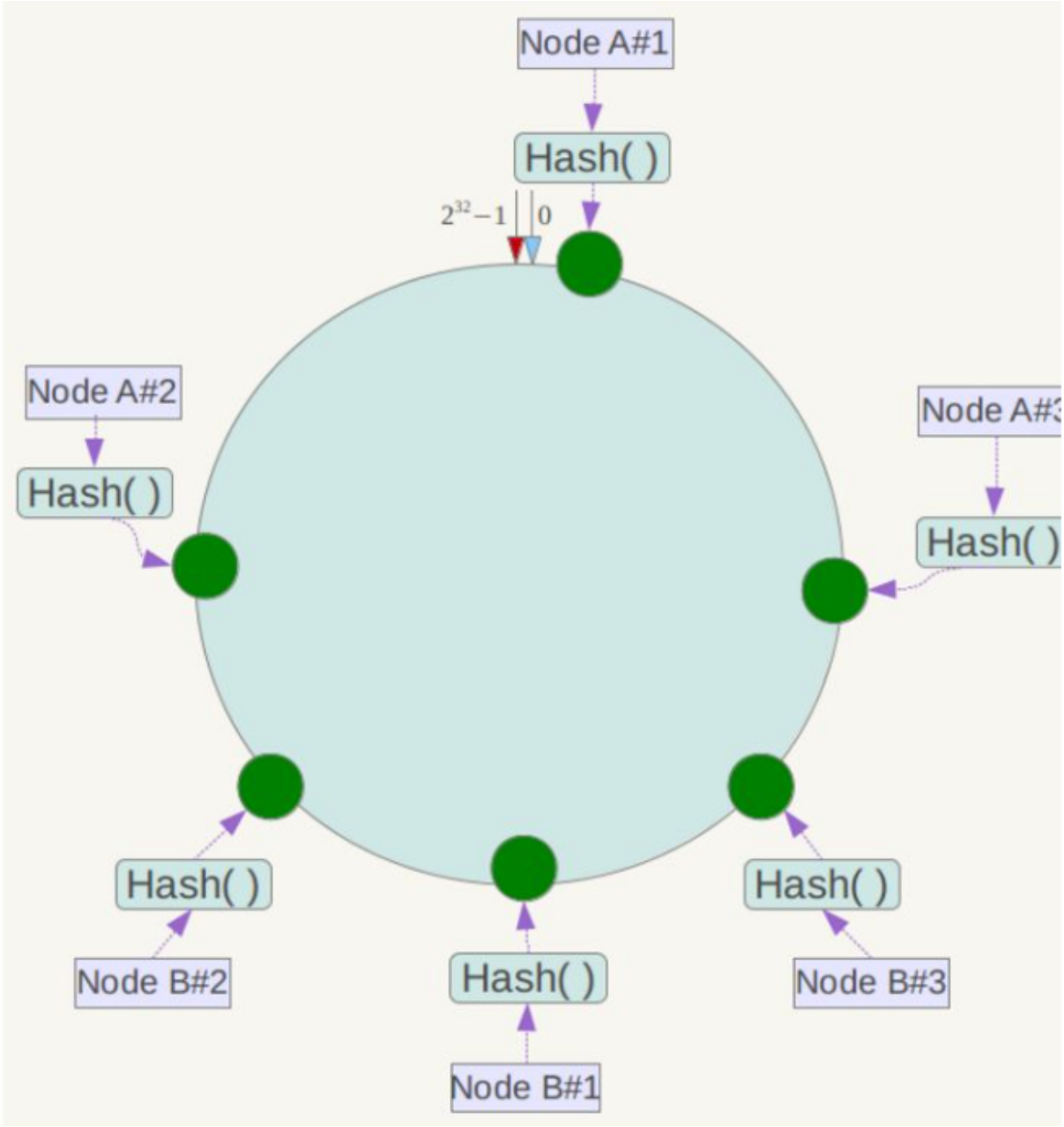
Hash环的数据倾斜问题

一致性Hash算法在服务节点太少时，容易因为节点分布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)的问题，例如系统中只有两台服务器，其环分布如下：



此时必然会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Hash，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。

例如上面的情况，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



同时数据定位算法不变，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。

七、总结

上文中，我们一步步分析了什么是一致性Hash算法，主要是考虑到分布式系统每个节点都有可能失效，并且新的节点很可能动态的增加进来的情况，如何保证当系统的节

点数目发生变化的时候，我们的系统仍然能够对外提供良好的服务，这是值得考虑的！

手写一下java实现代码（了解一下）