一致性哈希 [1]

leolinuxer

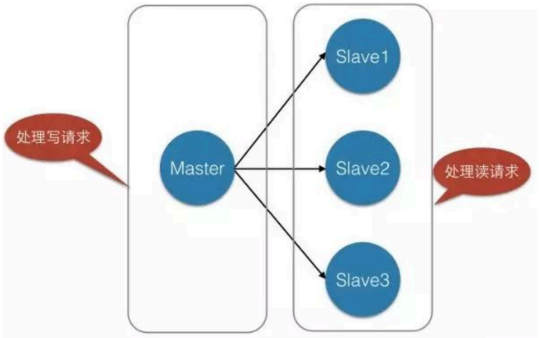
August 12, 2020

Contents

1 一致性哈希的直观理解

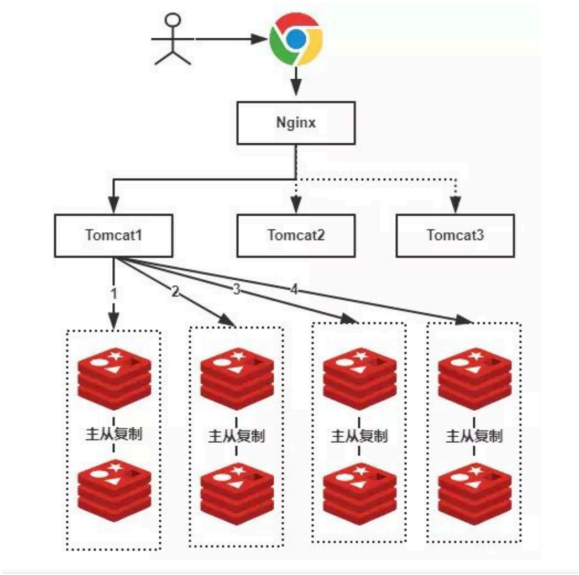
1.1 Redis 集群的使用

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（标准可能不一样，要看 Redis 服务器容量）我们同样可以对 Redis 进行类似的操作，就是分库分表。

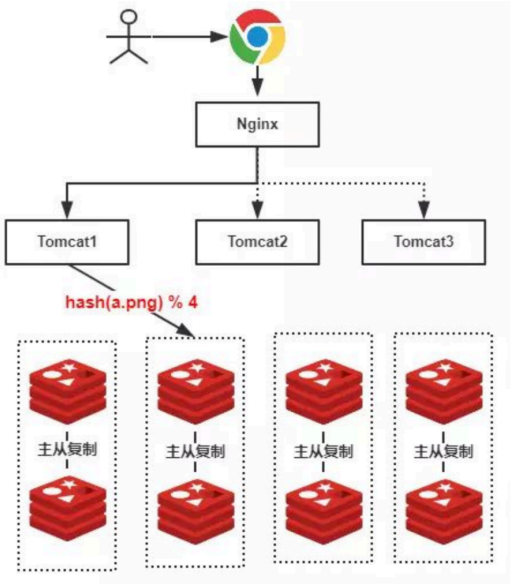
假设，我们有一个社交网站，需要使用 Redis 存储图片资源，存储的格式为键值对，key 值为图片名称，value 为该图片所在文件服务器的路径，我们需要根据文件名查找该文件所在文件服务器上的路径，数据量大概有 2000W 左右，按照我们约定的规则进行分库，规则就是随机分配，我们可以部署 8 台缓存服务器，每台服务器大概含有 500W 条数据，并且进行主从复制，示意图如下：



由于规则是随机的，所有我们的一条数据都有可能存储在任何一组 Redis 中，例如上图我们用户查找一张名称为”a.png”的图片，由于规则是随机的，我们不确定具体是在哪一个 Redis 服务器上的，因此我们需要进行 1、2、3、4，4 次查询才能够查询到（也就是遍历了所有的 Redis 服务器），这显然不是我们想要的结果，有了解过的小伙伴可能会想到，随机的规则不行，可以使用类似于数据库中的分库分表规则：按照 Hash 值、取模、按照类别、按照某一个字段值等等常见的规则就可以出来了！好，按照我们的主题，我们就使用 Hash 的方式。

1.2 为 Redis 集群使用 Hash

可想而知，如果我们使用 Hash 的方式，每一张图片在进行分库的时候都可以定位到特定的服务器，示意图如下：



上图中，假设我们查找的是” a.png”，由于有 4 台服务器（排除从库），因此公式为 hash(a.png) % 4 = 2 ，可知定位到了第 2 号服务器，这样的话就不会遍历所有的服务器，大大提升了性能！

1.3 使用 Hash 的问题

上述的方式虽然提升了性能，我们不再需要对整个 Redis 服务器进行遍历！但是，使用上述 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的时候，所有缓存的位置都要发生改变！

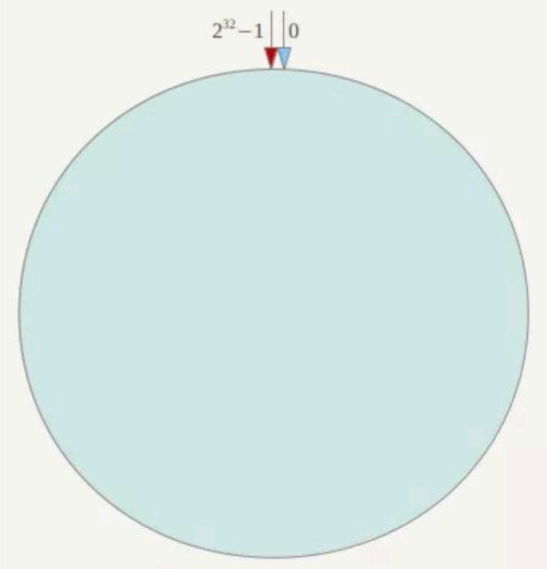
试想一下，如果 4 台缓存服务器已经不能满足我们的缓存需求，那么我们应该怎么做呢？很简单，多增加几台缓存服务器不就行了！假设：我们增加了一台缓存服务器，那么缓存服务器的数量就由 4 台变成了 5 台。那么原本 hash(a.png) % 4 = 2 的公式就变成了 hash(a.png) % 5 = ?，可想而知这个结果肯定不是 2 的，这种情况带来的结果就是当服务器数量变动时，所有缓存的位置都要发生改变！换句话说，当服务器数量发生改变时，所有缓存在一定时间内是失效的，当应用无法从缓存中获取数据时，则会向后端数据库请求数据！

同样的，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，那么缓存服务器数量从 4 台变为 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！

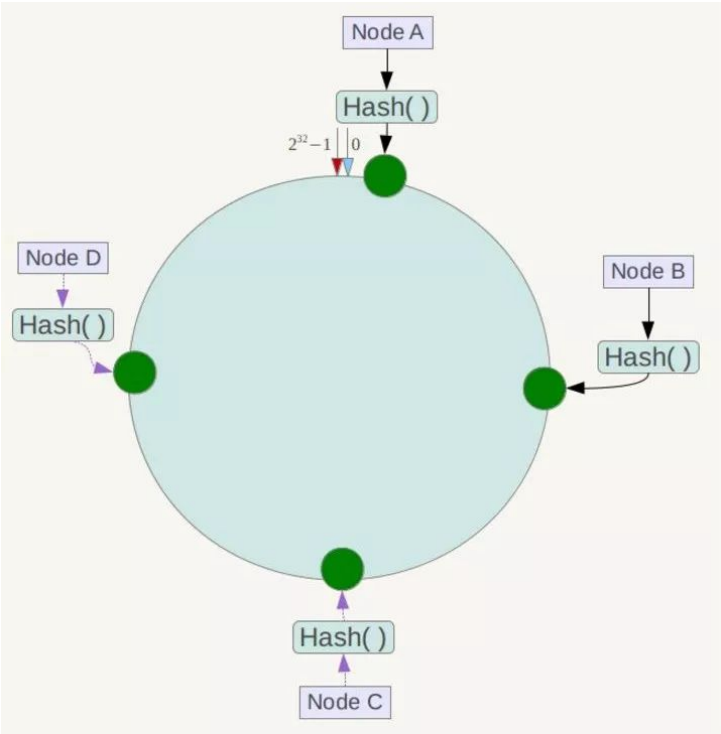
1.4 一致性 Hash 算法的神秘面纱

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，什么意思呢？简单来说，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：



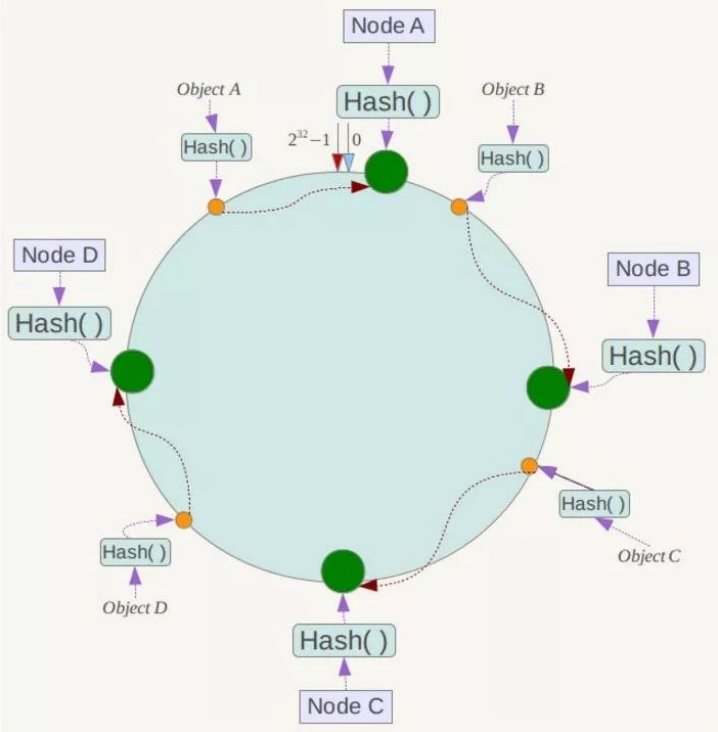
整个空间按顺时针方向组织，圆环的正上方的点代表 0，0 点右侧的第一个点代表 1，以此类推，2、3、4、5、6……直到 2^32-1，也就是说 0 点左侧的第一个点代表 2^32-1，0 和 2^32-1 在零点中方向重合，我们把这个由 2^32 个点组成的圆环称为 Hash 环。

下一步将各个服务器使用 Hash 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，这里假设将上文中四台服务器使用 IP 地址哈希后在环空间的位置如下：



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！

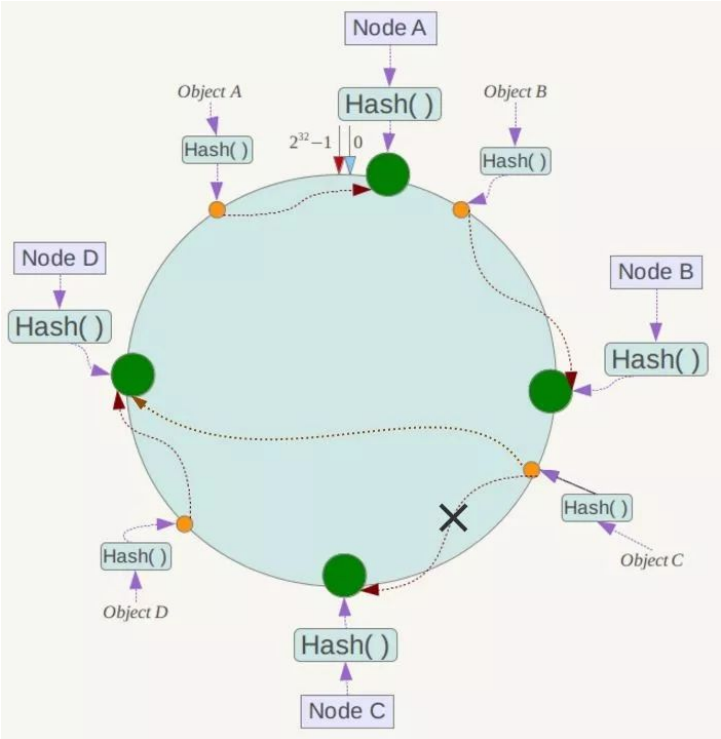
例如我们有 Object A、Object B、Object C、Object D 四个数据对象，经过哈希计算后，在环空间上的位置如下：



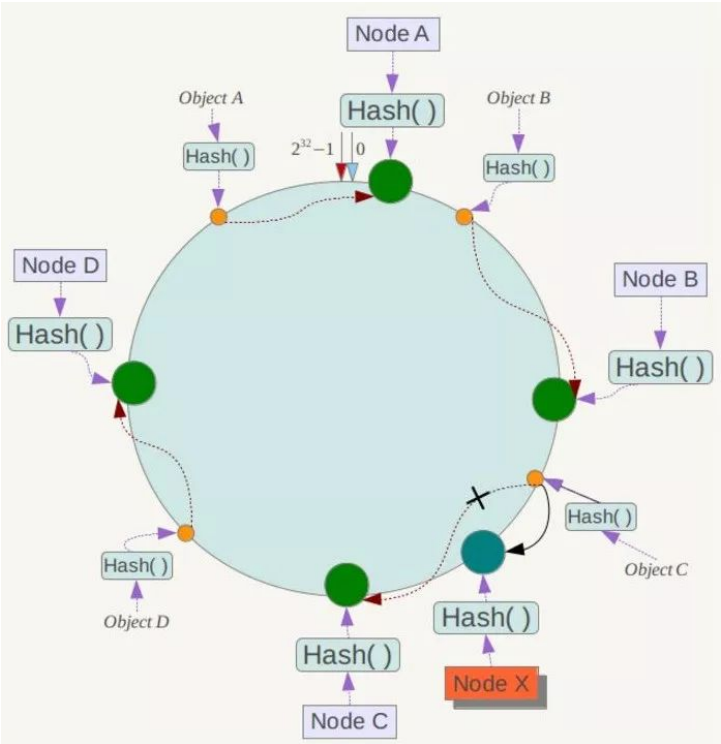
根据一致性 Hash 算法，数据 A 会被定为到 Node A 上，B 被定为到 Node B 上，C 被定为到 Node C 上，D 被定为到 Node D 上。

1.5 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：



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

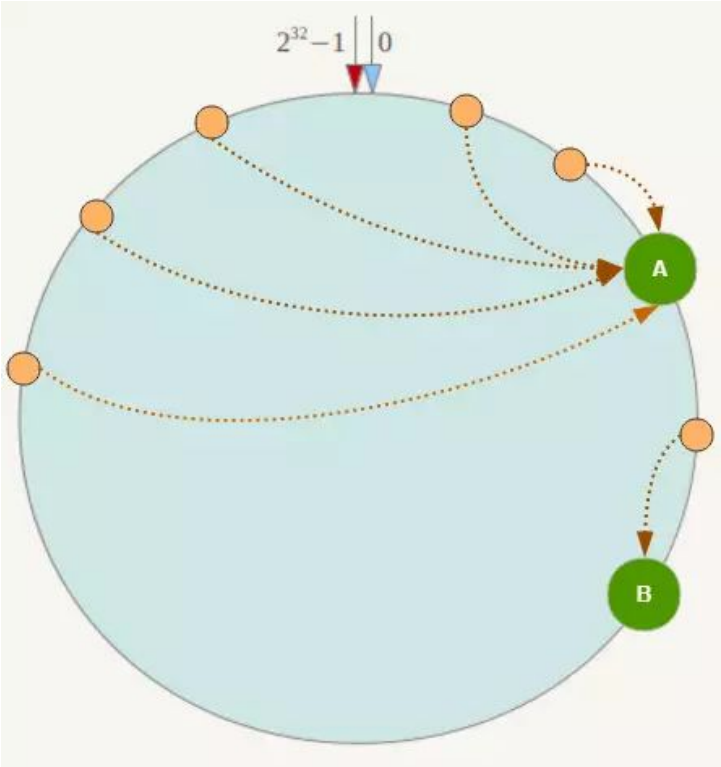


此时对象 Object A、B、D 不受影响，只有对象 C 需要重定位到新的 Node X！一般的，在一致性 Hash 算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

综上所述，一致性 Hash 算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

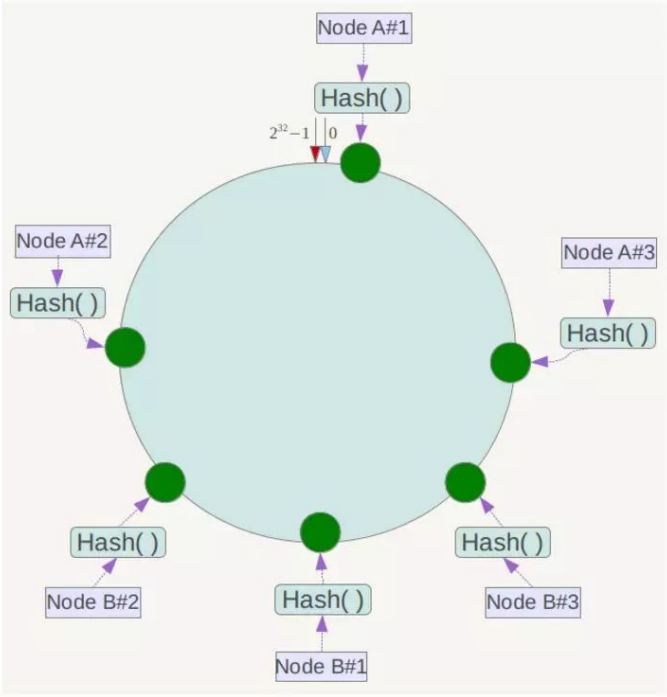
1.6 Hash 环的数据倾斜问题

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。

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。

References

[1] “面试必备：什么是一致性 hash 算法？.” [Online]. Available: https://zhuanlan.zhihu.com/p/34985026