一致哈希算法

jask

2024-08-23

一致哈希算法

如果我们通过 Raft 算法实现了 KV 存储，虽然领导者模型简化了算法实现和共识协商，但写请求只能限制在领导者节点上处理，导致了集群的接入性能约等于单机，那么随着业务发展，集群的性能可能就扛不住了，会造成系统过载和服务不可用，这时该怎么办呢？

有同学可能会说了，分集群还不简单吗？加个 Proxy 层，由 Proxy 层处理来自客户端的读写请求，接收到读写请求后，通过对 Key 做哈希找到对应的集群就可以了啊。

是的，哈希算法的确是个办法，但它有个明显的缺点：当需要变更集群数时（比如从 2 个集群扩展为 3 个集群），这时大部分的数据都需要迁移，重新映射，数据的迁移成本是非常高的。那么如何解决哈希算法，数据迁移成本高的痛点呢？答案就是一致哈希（Consistent Hashing）。

如何使用一致哈希实现哈希寻址？

一致哈希算法也用了取模运算，但与哈希算法不同的是，哈希算法是对节点的数量进行取模运算，而一致哈希算法是对 2^32 进行取模运算。你可以想象下，一致哈希算法，将整个哈希值空间组织成一个虚拟的圆环，也就是哈希环。

在一致哈希中，你可以通过执行哈希算法（为了演示方便，假设哈希算法函数为“c-hash()”），将节点映射到哈希环上，比如选择节点的主机名作为参数执行 c-hash()，那么每个节点就能确定其在哈希环上的位置了：

当需要对指定 key 的值进行读写的时候，你可以通过下面 2 步进行寻址：

首先，将 key 作为参数执行 c-hash() 计算哈希值，并确定此 key 在环上的位置；然后，从这个位置沿着哈希环顺时针“行走”，遇到的第一节点就是 key 对应的节点。

假设 key-01、key-02、key-03 三个 key，经过哈希算法 c-hash() 计算后，在哈希环上的位置就像图 6 的样子：

那么根据一致哈希算法，key-01 将寻址到节点 A，key-02 将寻址到节点 B，key-03 将寻址到节点 C。

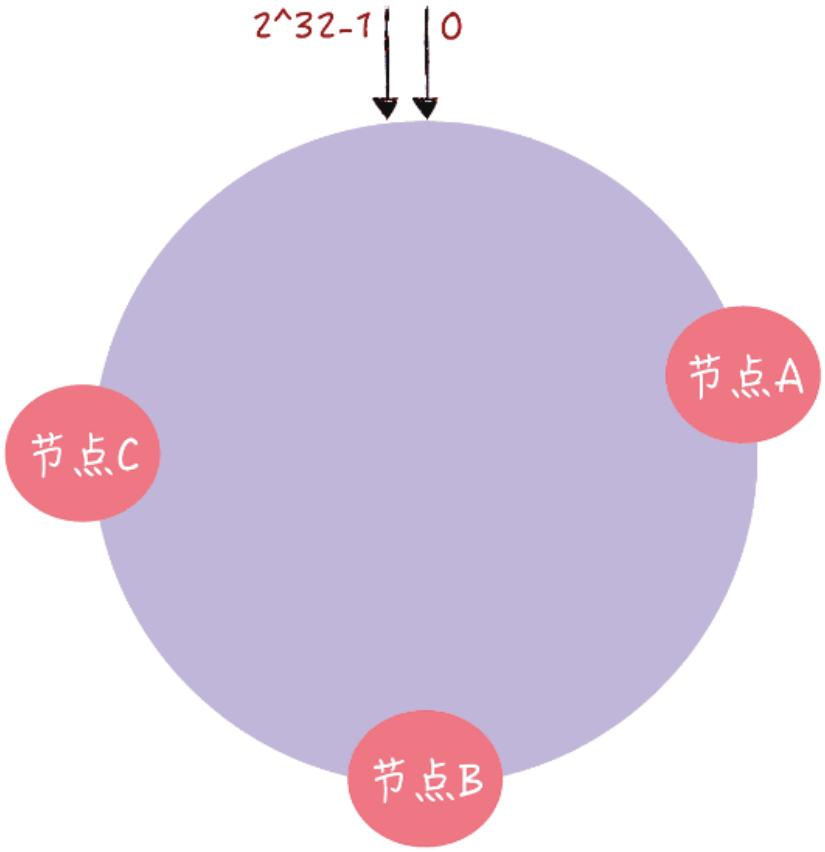

Figure 1: 如图

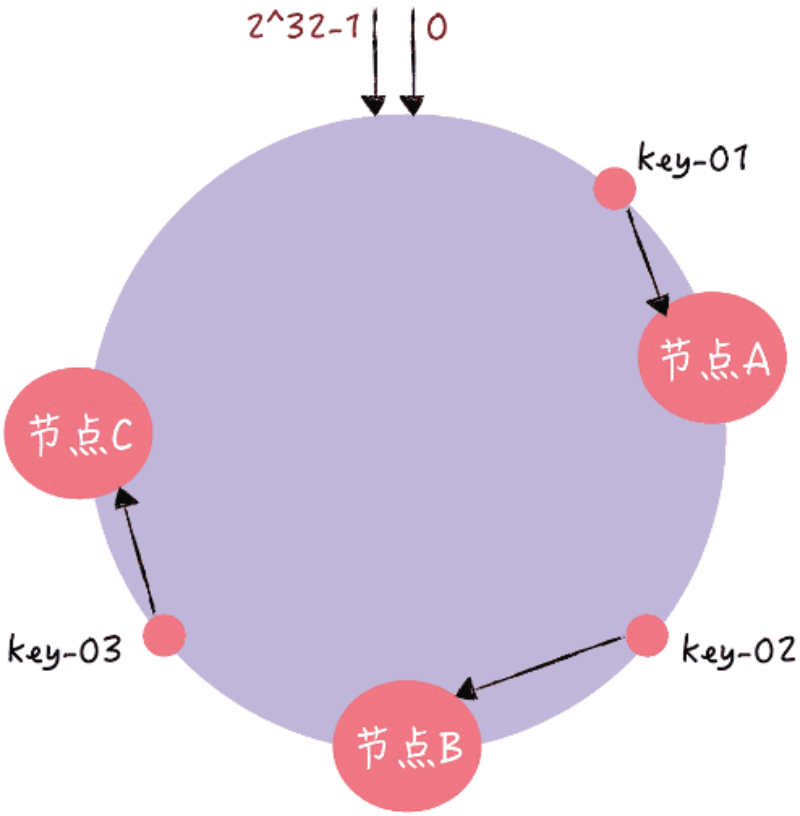

Figure 2: 图 6

移除节点

假设，现在有一个节点故障了（比如节点 *C*）：

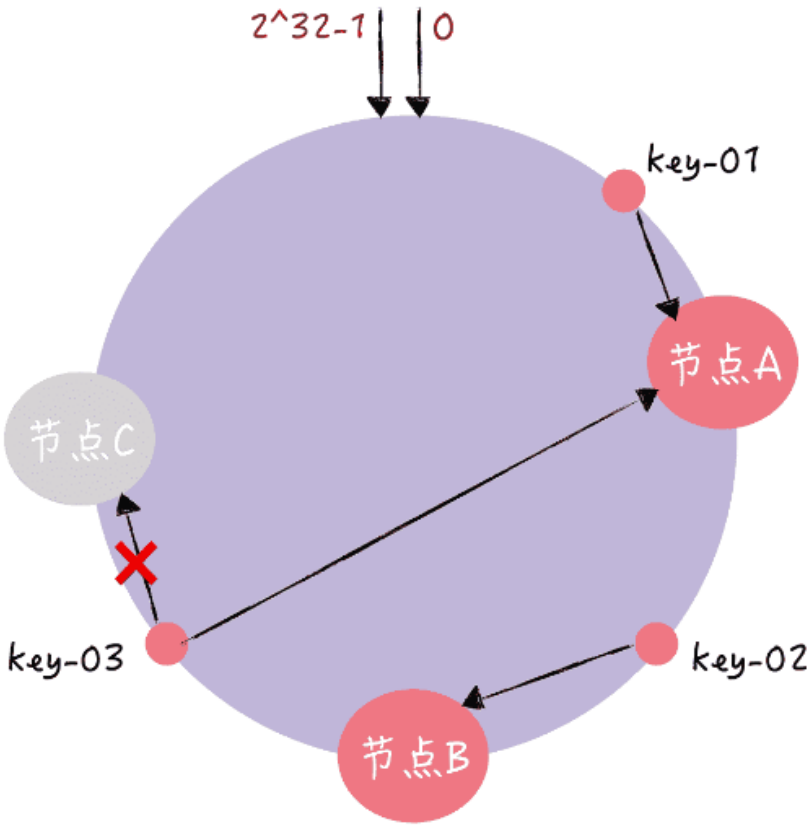

Figure 3: 如图

key-01 和 key-02 不会受到影响，只有 key-03 的寻址被重定位到 A。一般来说，在一致哈希算法中，如果某个节点宕机不可用了，那么受影响的数据仅仅是，会寻址到此节点和前一节点之间的数据。比如当节点 *C* 宕机了，受影响的数据是会寻址到节点 B 和节点 *C* 之间的数据（例如 key-03），寻址到其他哈希环空间的数据（例如 key-01），不会受到影响。

那如果此时集群不能满足业务的需求，需要扩容一个节点（也就是增加一个节点，比如 D）：

你可以看到，key-01、key-02 不受影响，只有 key-03 的寻址被重定位到新节点 D。一般而言，在一致哈希算法中，如果增加一个节点，受影响的数据仅仅是，会寻址到新节点和前一节点之间的数据，其它数据也不会受到影响。

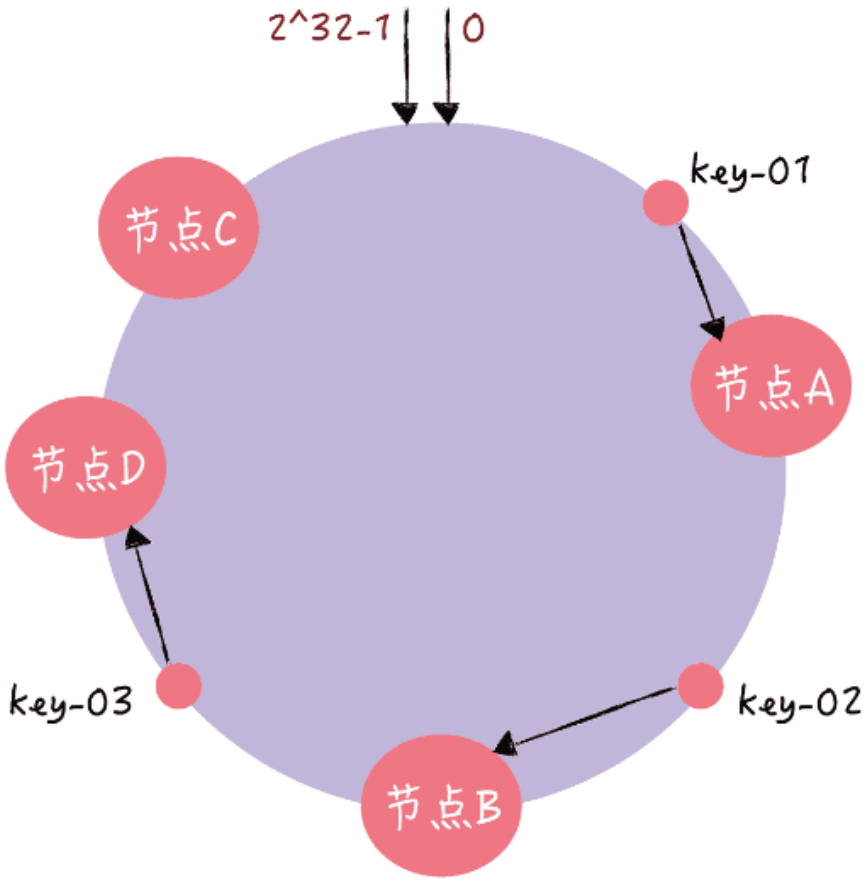

Figure 4: 如图

在哈希寻址中常出现这样的问题

客户端访问请求集中在少数的节点上，出现了有些机器高负载，有些机器低负载的情况，那么在一致哈希中，有什么办法能让数据访问分布的比较均匀呢？答案就是虚拟节点。

如何解决? 其实，就是对每一个服务器节点计算多个哈希值，在每个计算结果位置上，都放置一个虚拟节点，并将虚拟节点映射到实际节点。比如，可以在主机名的后面增加编号，分别计算“Node-A-01”“Node-A-02”“Node-B-01”“Node-B-02”“Node-C-01”“Node-C-02”的哈希值，于是形成 6 个虚拟节点：

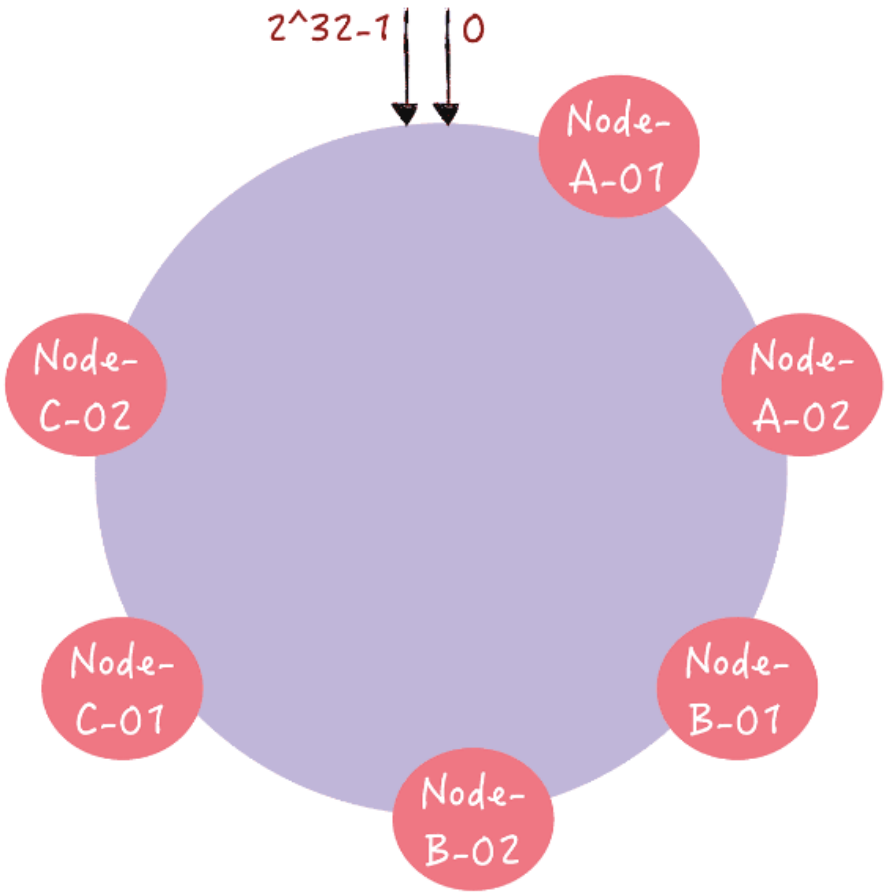

Figure 5: 如图

从图中看到，增加了节点后，节点在哈希环上的分布就相对均匀了。这时，如果有访问请求寻址到“Node-A-01”这个虚拟节点，将被重定位到节点 A。你看，这样我们就解决了冷热不均的问题。

总结

一致哈希是一种特殊的哈希算法，在使用一致哈希算法后，节点增减变化时只影响到部分数据的路由寻址，也就是说我们只要迁移部分数据，就能实现集群的稳定了。

当节点数较少时，可能会出现节点在哈希环上分布不均匀的情况。这样每个节点实际占据环上的区间大小不一，最终导致业务对节点的访问冷热不均。需要你注意的是，这个问题可以通过引入更多的虚拟节点来解决。

一致哈希本质上是一种路由寻址算法，适合简单的路由寻址场景。比如在 KV 存储系统内部，它的特点是简单，不需要维护路由信息。