

漫画：什么是一致性哈希？

2017-07-17 玻璃猫 梦见

小灰，最近在忙什么呢？



哎，最近在搞数据库迁移，可把我给累坏了……



哦？好好的为什么要做数据库迁移呀？



哎，还不是因为数据量超出了
当初的预估.....



一年之前——

产品经理：小灰，这是项目的需求文档，你好好看看吧。咱们未来的订单量会比较大，你要多注意系统的可扩展性哦。



放心吧，系统设计交给我妥妥的！



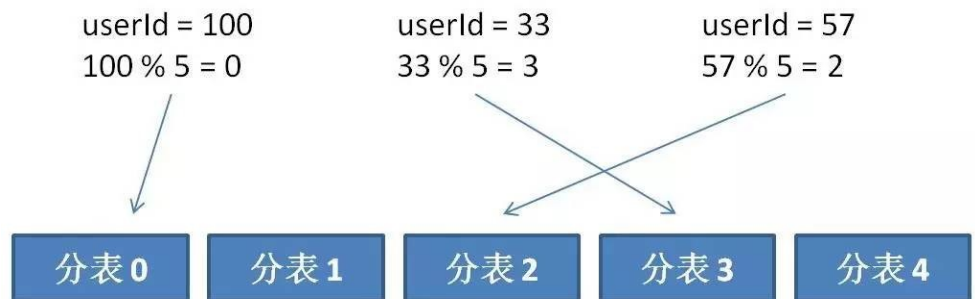
未来两年内，系统预估的总订单数量可达一亿条左右。

按Mysql单表存储500万条记录来算，暂时不必分库，单库30个分表是比较合适的水平分表方案。

于是小灰设计了这样的分表逻辑：

1.
订单表创建单库30个分表
2.
对用户ID和30进行取模，取模结果决定了记录存于第几个分表
3.
查询时需要以用户ID作为条件，根据取模结果确定查询哪一个分表

分表方式如下图（为了便于描述，简化为5个分表）：



过了两个月——

测试一切顺利，项目终于
上线啦！



辛苦啦小灰！



又过了半年多——

小灰小灰，有好消息！咱们项目的流量比预估的更大，上线刚半年多，订单量已经接近一亿了！



哇，真是个好消息！可是数据量这么大，当初设计的 30 个分表就不够用了，每个分表数据太多会影响性能的！



这个就要看你的喽，想一想
解决方案吧。



好吧，我回去想一想……



直接增加分表肯定不行，原先的哈希
规则会被打乱……



做全量数据迁移倒是能解决问题，
但是迁移的代价实在太大了



天呐，这可怎么办



小灰的回忆告一段落——

就这样，我们只好进行数据迁移，用新的 hash 规则 ($\text{userID} \% 80$)，把 30 个分表的数据迁移到 80 个分表当中.....



哈哈，小灰，你知道「一致性哈希」吗？



一致性哈希？那是什么鬼？



一致性哈希简称 DHT，是麻省理工学院提出的一种算法，目前主要应用于分布式缓存当中。



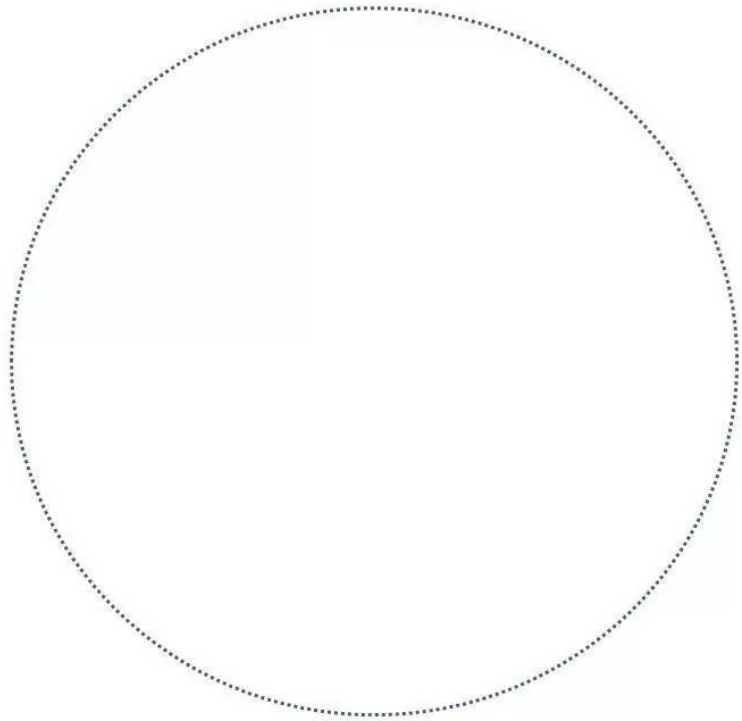
听起来好高大上，这种算法有什么好处呀？



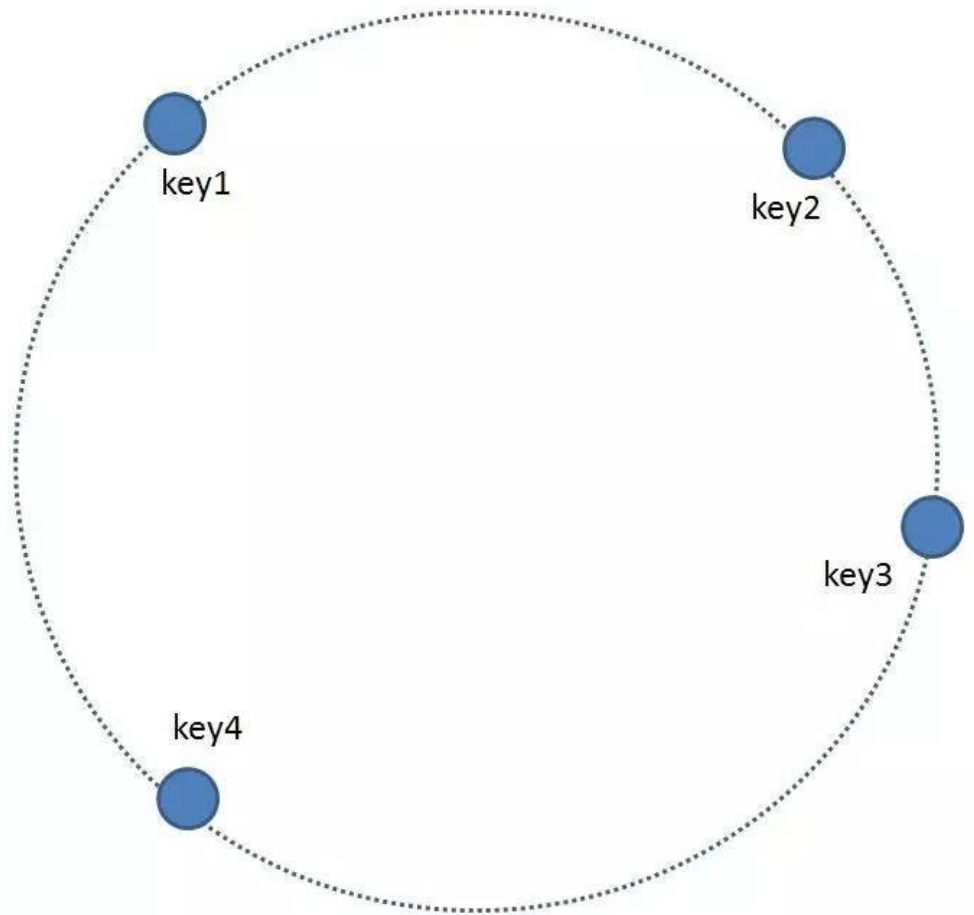
一致性哈希可以有效地解决分布式存储结构下动态增加和删除节点所带来的问题。我们简单举例说明一下：



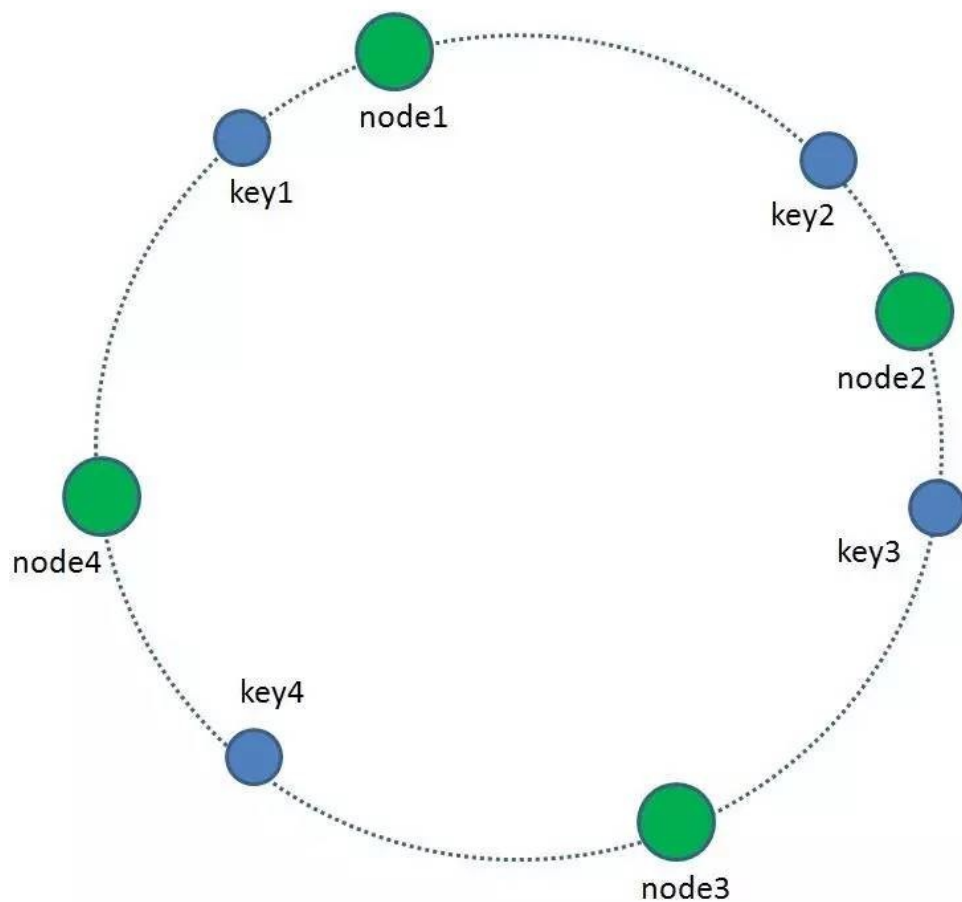
1.首先，我们把全量的缓存空间当做一个环形存储结构。环形空间总共分成 2^{32} 个缓存区，在Redis中则是把缓存key分配到16384个slot。



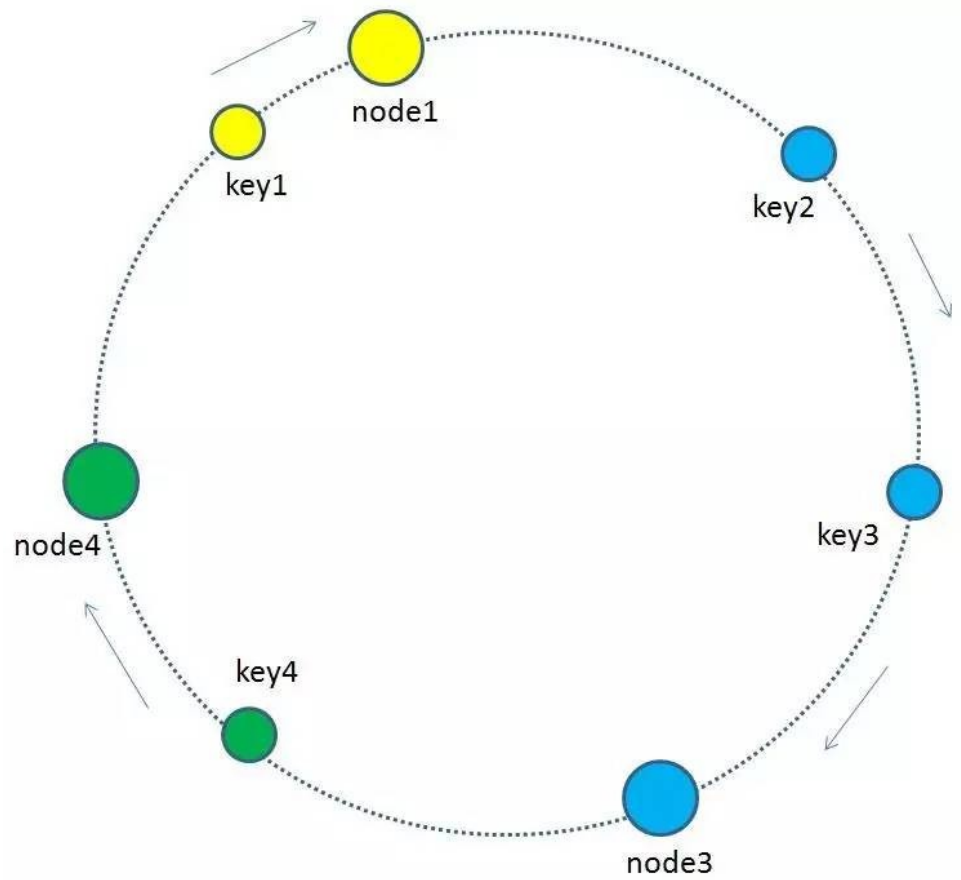
2.每一个缓存key都可以通过Hash算法转化为一个32位的二进制数，也就对应着环形空间的某一个缓存区。我们把所有的缓存key映射到环形空间的不同位置。



3.我们的每一个缓存节点（Shard）也遵循同样的Hash算法，比如利用IP做Hash，映射到环形空间当中。



4.如何让key和节点对应起来呢？很简单，每一个key的顺时针方向最近节点，就是key所归属的存储节点。所以图中key1存储于node1，key2，key3存储于node2，key4存储于node3。



一致性哈希的映射方式好古怪啊，
这样设计究竟有什么优势呢？

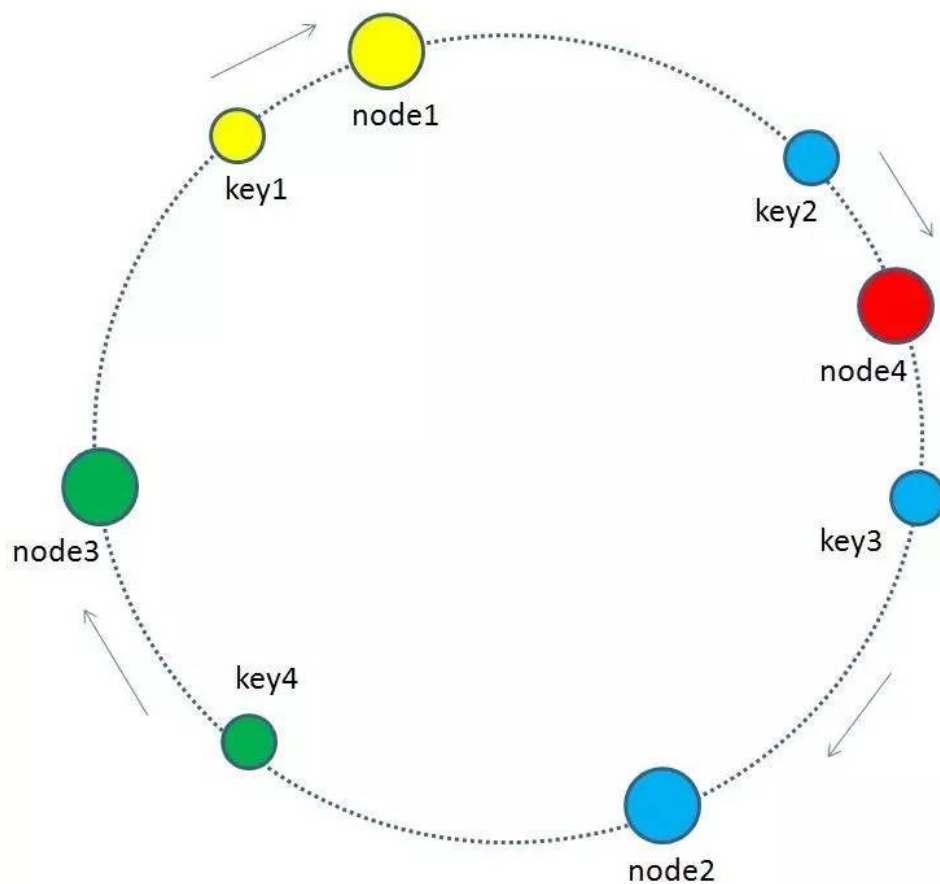


当缓存的节点有增加或删除的时候，一致性哈希的优势就显现出来了。让我们来看看实现细节：

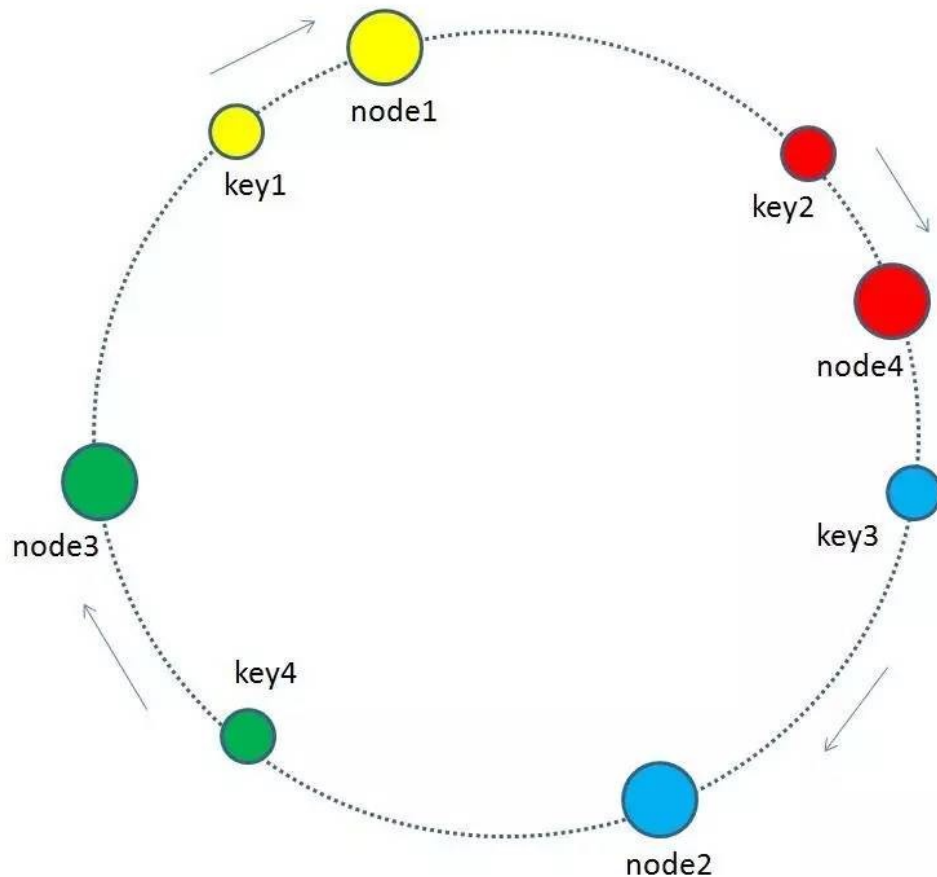


1.增加节点

当缓存集群的节点有所增加的时候，整个环形空间的映射仍然会保持一致性哈希的顺时针规则，所以有一小部分key的归属会受到影响。



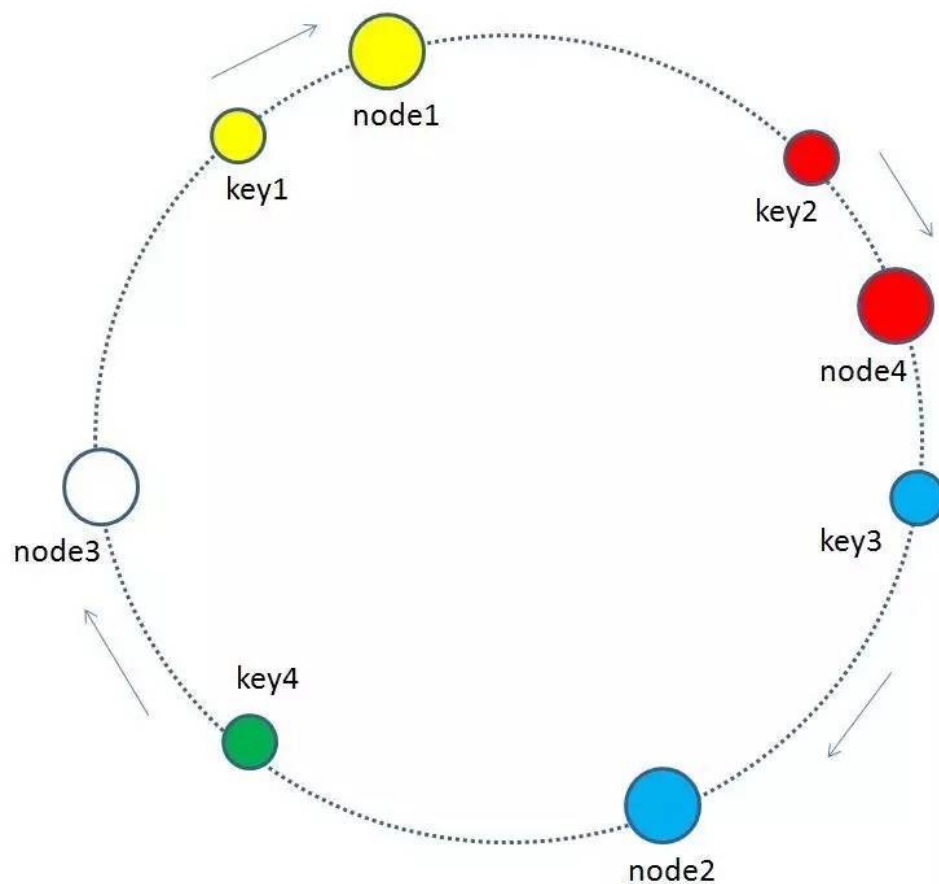
有哪些key会受到影响呢？图中加入了新节点node4，处于node1和node2之间，按照顺时针规则，从node1到node4之间的缓存不再归属于node2，而是归属于新节点node4。因此受影响的key只有key2。



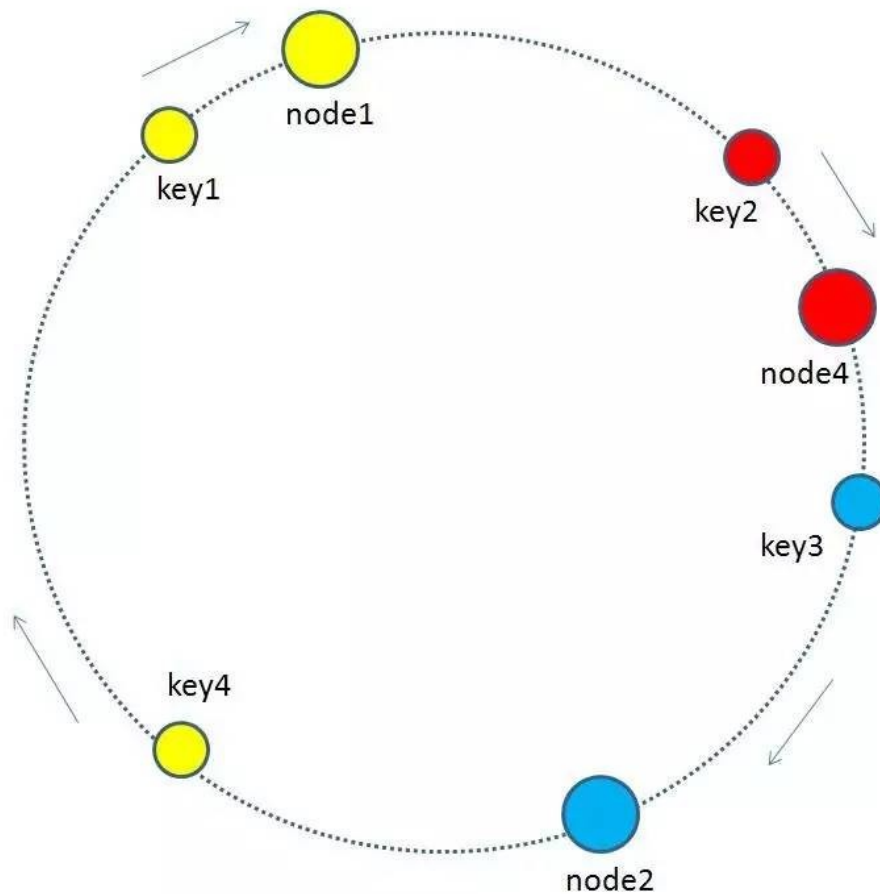
最终把key2的缓存数据从node2迁移到node4，就形成了新的符合一致性哈希规则的缓存结构。

2.删除节点

当缓存集群的节点需要删除的时候（比如节点挂掉），整个环形空间的映射同样会保持一致性哈希的顺时针规则，同样有一小部分key的归属会受到影响。



有哪些key会受到影响呢？图中删除了原节点node3，按照顺时针规则，原本node3所拥有的缓存数据就需要“托付”给node3的顺时针后继节点node1。因此受影响的key只有key4。



最终把key4的缓存数据从node3迁移到node1，就形成了新的符合一致性哈希规则的缓存结构。

大黄，我有一点不明白，既然节点node3已经挂掉了，它还怎么迁移数据到节点node1呀？



傻孩子，这里所说的迁移并不是直接的数据迁移，而是在查询时去找顺时针的后继节点，因缓存未命中而刷新缓存。

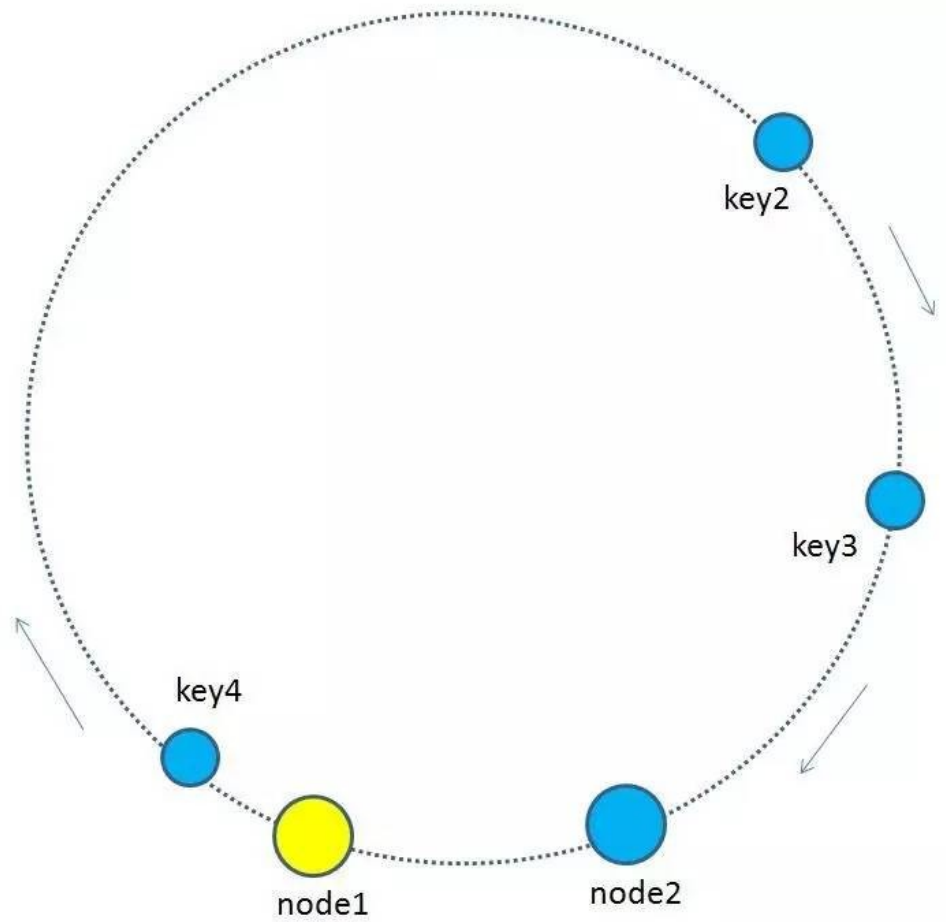


原来如此。还有一个问题，既然缓存节点都是按 ip 来 hash 到环形空间，如果出现分布不均匀的情况怎么办呢？



比如像下图这样，按顺时针规则，所有的 key 都不巧归属于同一个节点。

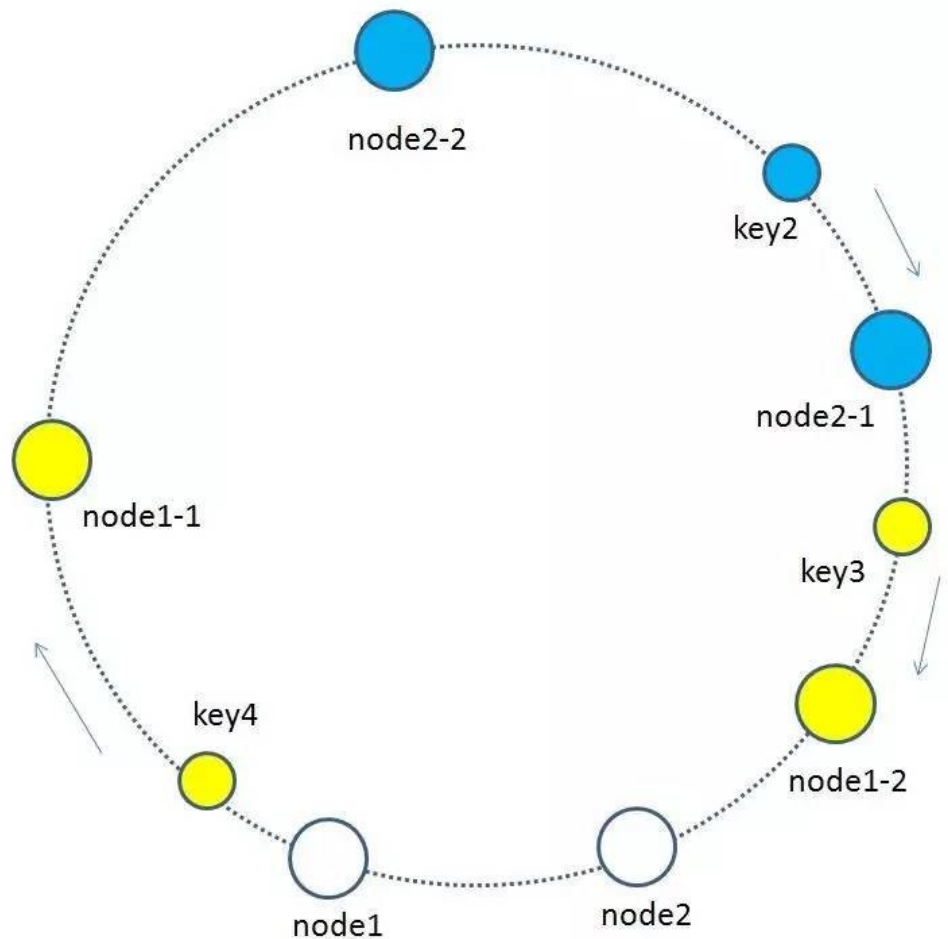




问得好，为了优化这种节点太少而产生的不均衡情况，一致性哈希算法引入了「虚拟节点」的概念。



所谓虚拟节点，就是基于原来的物理节点映射出N个子节点，最后把所有的子节点映射到环形空间上。



如上图所示，假如node1的ip是192.168.1.109，那么原node1节点在环形空间的位置就是hash（“192.168.1.109”）。

我们基于node1构建两个虚拟节点，node1-1 和 node1-2，虚拟节点在环形空间的位置可以利用（IP+后缀）计算，例如：

hash（“192.168.1.109#1”），hash（“192.168.1.109#2”）

此时，环形空间中不再有物理节点node1，node2，只有虚拟节点node1-1，node1-2，node2-1，node2-2。由于虚拟节点数量较多，缓存key与虚拟节点的映射关系也变得相对均衡了。

最后一个问题，为什么一致性哈希算法更多地应用于像 Redis 这样的缓存数据库呢？



关于这一点，我认为这是由于分布式缓存系统的节点部署变化更频繁，而传统关系型数据库的分库分表相对稳定。



不过说回到 mysql，在水平分库分表的过程中，你仍然可以采用一致性哈希的思想。



虽然这样的处理逻辑会复杂一些，
却可以避免动态水平扩展时候的尴尬。小灰，你说是不是？



有道理，如果我一开始也这样
设计水平分表，就不会遇到现
在的全量数据迁移问题了！

