# 最通俗易懂的一致性哈希算法原理

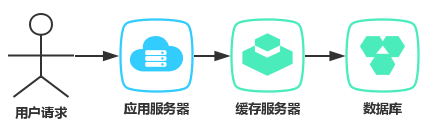
2019年01月08日 01:05:52 [活在梦里丶](https://me.csdn.net/qq_25448409" \t "https://blog.csdn.net/qq_25448409/article/details/_blank) 阅读数：51 标签： [redis](http://so.csdn.net/so/search/s.do?q=redis&t=blog" \t "https://blog.csdn.net/qq_25448409/article/details/_blank)[一致性哈希算法](http://so.csdn.net/so/search/s.do?q=%E4%B8%80%E8%87%B4%E6%80%A7%E5%93%88%E5%B8%8C%E7%AE%97%E6%B3%95&t=blog" \t "https://blog.csdn.net/qq_25448409/article/details/_blank)[哈希倾斜](http://so.csdn.net/so/search/s.do?q=%E5%93%88%E5%B8%8C%E5%80%BE%E6%96%9C&t=blog" \t "https://blog.csdn.net/qq_25448409/article/details/_blank) 更多

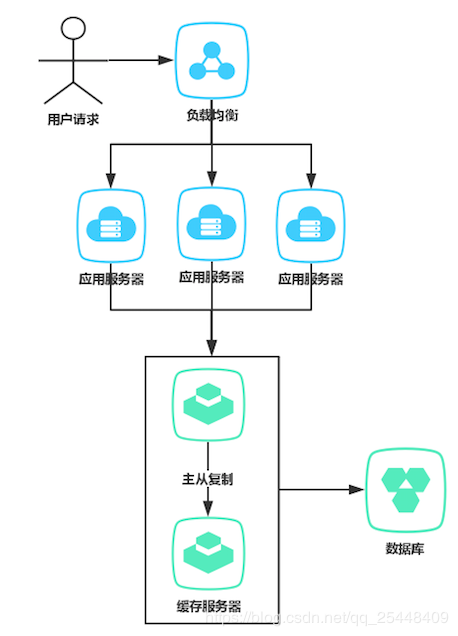
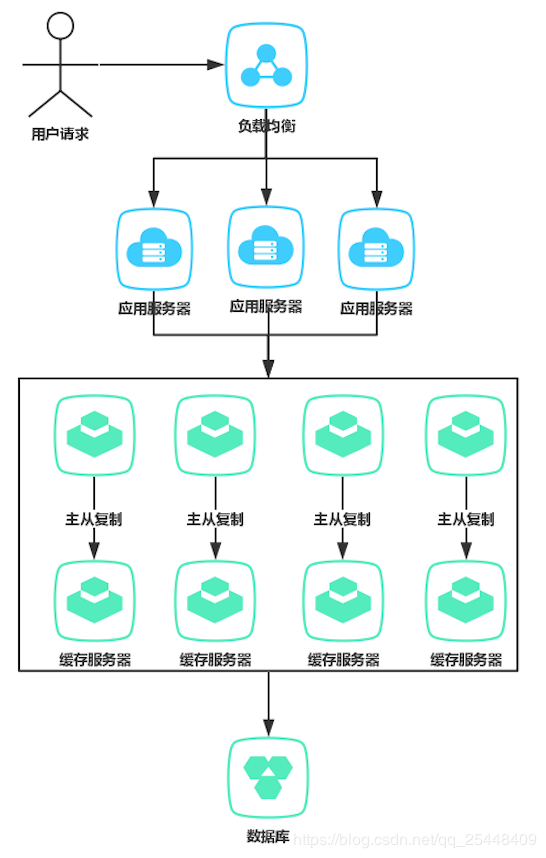
个人分类： [缓存](https://blog.csdn.net/qq_25448409/article/category/8593472" \t "https://blog.csdn.net/qq_25448409/article/details/_blank)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/qq\_25448409/article/details/86013885

# **前言**

再讨论一致性哈希之前，我们先来回顾一下缓存的演化历史：

当我们的系统还是一个非常小的时候，对于用户的请求我们直接访问数据库就好了。当访问量到了一定数量的时候，我们使用缓存服务器，缓存一部分热数据来减轻数据库的压力，这便是最原始的阶段。  


网站发展到一定规模后，并发量上来了，我们通过使用nginx的反向代理对我们的应用服务器做负载均衡，同时需要缓存服务的高可用，以免防止出现缓存雪崩等问题，造成数据库宕机，从而影响整个系统的可用性。这时候我们可以采用主从复制，或者是读写分离来提升缓存服务的可用性。  
  
但是，当我们的系统越来越大，需要缓存的数据越来越多的时候，就会遇到缓存不够用，被撑爆的情况。同时如果我们无脑的通过扩充内存容量（甚至TB级别以上）来解决缓存不够用的情况，可能会出现性能越来越慢的情况。此时我们就需要采用分库分表策略，把我们的缓存数据，均分到不同的缓存服务器，从而避免单节点服务器内存过大的情况。  


# **Redis集群**

## **使用Hash的集群**

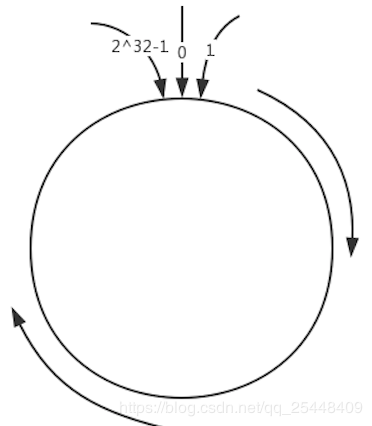
为了系统高可用，我们构建了redis主从复制；为了提升响应速率，我们对redis进行了分库分表。

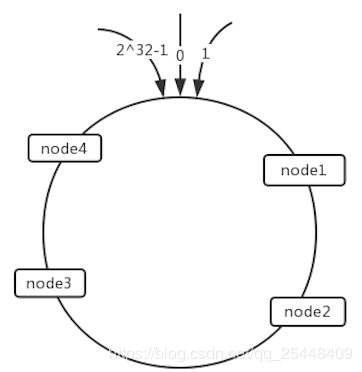
假设现在有8台redis服务，4台master，4台slave，构成的redis集群（如上图）。我们现在我们要缓存一个用户，key是用户id，value是用户的信息。假设用户id=1234，1234%4=2我们就把这条用户的信息存放到第二台服务器上面。当我们需要读取缓存的时候，同样根据用户id=1232求一次哈希值，得到结果2，从第二台服务器上面获取数据。

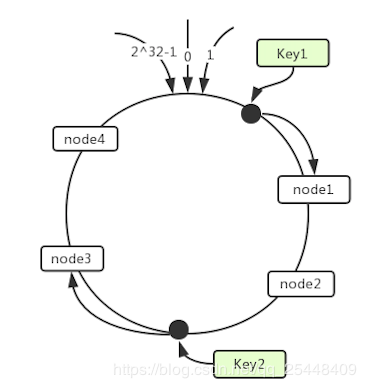
这样似乎解决问题了，但是又有新的问题产生了。当我们需要扩容我们的集群，或者缩容的时候，我们想一下，我们以前id=1234的用户信息在第二台服务器上存着，当服务器增加到5台的时候，我们获取用户信息求哈希值1234%5=4，这时候就出大问题了，因为哈希因子的改变，我们id=1234的用户信息明明存放在第二台服务器，但是现在变成第四台了，我们所有节点存放的数据都没用了，自己手写过HashMap的人肯定明白，这时候需要对所有的数据重新求一次哈希值。但是我们线上缓存系统可不能这么玩，让缓存服务重新计算哈希值，肯定需要花费一定时间，少了缓存系统，大量的请求到了数据库，就会发生缓存雪崩。

所以为了不让这种事情发生，一致性哈希算法就诞生啦！

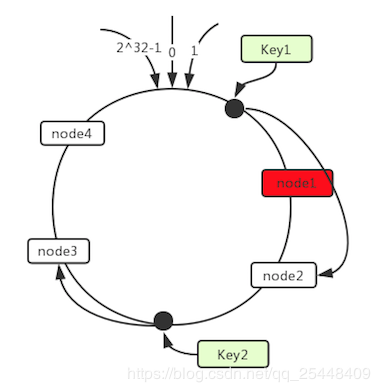
# **一致性Hash**

一致性Hash也是基于模运算，只是和刚才的模运算有所不同。不是针对节点数量进行取模，而是针对232-1 进行取模，什么意思？就是将整个哈希空间想象成一个虚拟圆环，圆环顺时针从0开始到232-1结束，我们称这个圆环为哈希环，如下图：  


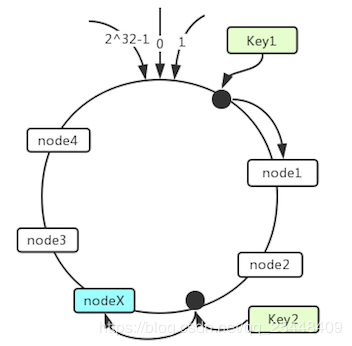
接下来对服务器IP或者主机名作为关键字进行哈希取模，这样就能在哈希环上定位每台机器的位置  


现在我们要插入一个Key-Value键值对，首先求出key的hash code然后进行模运算，这样就能定位这个key在哈希环上的位置，如果当前位置没有node服务器，那么我们就顺时针寻找，遇到的第一个节点，就是我们数据所对应的服务器。  


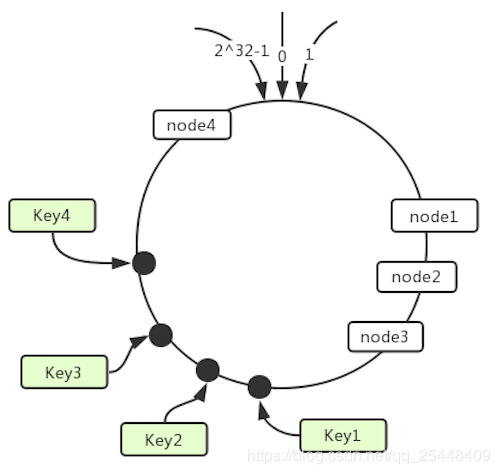
## **一致性哈希的容错性**

现在我们假设我们的node1挂掉了，但是我们的node2，node3，node4并不会收到影响，而key1只会被重新定位到node2上面。所以对于一致性哈希算法来说，如果其中一个节点出现故障，受影响的只是该节点的后一个节点node2。  


## **一致性哈希的扩展性**

如果我们要新添加一台服务器nodeX，计算出哈希值后，发现它的位置在node2和node3之间。那么受影响的只有新增节点的前一个节点node3（我们需要重新增加缓存到新的nodeX）而其他的节点都不会受影响。  


## **哈希环数据倾斜**

上面的4个节点都是相对完美的情况下，把哈希节点较为均等的分成了4等分。但是如果求出的哈希值不均等，某些节点的位置隔得很近，大量的数据就会集中到某一台服务器，如下图  
  
这种情况下，大量的节点定位到了node4这台服务器上面，极少的数据定位到node2或node3，造成了数据数据像node4上倾斜。一致性哈希算法为了数据倾斜，引入了虚拟节点，针对每个节点计算出多个哈希节点，均匀的分布到哈希环上面

比如现在的node1，node2，node3，node4，我们可以将关键字（服务器IP或者主机名）后面加上版本号。例如nodeA#1、nodeA#2、nodeB#1、nodeB#2意思类推。定位算法不变，只是多了异步虚拟节点映射，虽然定位到nodeA#1或nodeA#2节点，但是最终会映射到node1。  
