分布式一致性hash算法

2018.11.13 13:18

# 第一篇

[https://www.cnblogs.com/moonandstar08/p/5405991.html]

**一、分布式算法**

    在做服务器负载均衡时候可供选择的负载均衡的算法有很多，包括： 轮循算法(Round Robin)、哈希算法(HASH)、最少连接算法(Least Connection)、响应速度算法(Response Time)、加权法(Weighted )等。其中哈希算法是最为常用的算法.

  典型的应用场景是： 有N台服务器提供缓存服务，需要对服务器进行负载均衡，将请求平均分发到每台服务器上，每台机器负责1/N的服务。

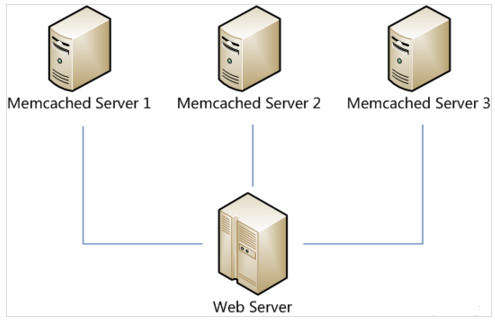
  常用的算法是对hash结果取余数 (hash() mod N )：对机器编号从0到N-1，按照自定义的 hash()算法，对每个请求的hash()值按N取模，得到余数i，然后将请求分发到编号为i的机器。但这样的算法方法存在致命问题，如果某一台机器宕机，那么应该落在该机器的请求就无法得到正确的处理，这时需要将当掉的服务器从算法从去除，此时候会有(N-1)/N的服务器的缓存数据需要重新进行计算;如果新增一台机器，会有N /(N+1)的服务器的缓存数据需要进行重新计算。对于系统而言，这通常是不可接受的颠簸(因为这意味着大量缓存的失效或者数据需要转移)。那么，如何设计一个负载均衡策略，使得受到影响的请求尽可能的少呢?

  在Memcached、Key-Value Store 、Bittorrent DHT、LVS中都采用了Consistent Hashing算法，可以说Consistent Hashing 是分布式系统负载均衡的首选算法。

**二、分布式缓存问题**

   在大型web应用中，缓存可算是当今的一个标准开发配置了。在大规模的缓存应用中，应运而生了分布式缓存系统。分布式缓存系统的基本原理，大家也有所耳闻。key-value如何均匀的分散到集群中？说到此，最常规的方式莫过于hash取模的方式。比如集群中可用机器适量为N，那么key值为K的的数据请求很简单的应该路由到hash(K) mod N对应的机器。的确，这种结构是简单的，也是实用的。但是在一些高速发展的web系统中，这样的解决方案仍有些缺陷。随着系统访问压力的增长，缓存系统不得不通过增加机器节点的方式提高集群的相应速度和数据承载量。增加机器意味着按照hash取模的方式，在增加机器节点的这一时刻，大量的缓存命不中，缓存数据需要重新建立，甚至是进行整体的缓存数据迁移，瞬间会给DB带来极高的系统负载，设置导致DB服务器宕机。 那么就没有办法解决hash取模的方式带来的诟病吗？

 假设我们有一个网站，最近发现随着流量增加，服务器压力越来越大，之前直接读写数据库的方式不太给力了，于是我们想引入Memcached作为缓存机制。现在我们一共有三台机器可以作为Memcached服务器，如下图所示：



    很显然，最简单的策略是将每一次Memcached请求随机发送到一台Memcached服务器，但是这种策略可能会带来两个问题：一是同一份数据可能被存在不同的机器上而造成数据冗余，二是有可能某数据已经被缓存但是访问却没有命中，因为无法保证对相同key的所有访问都被发送到相同的服务器。因此，随机策略无论是时间效率还是空间效率都非常不好。

    要解决上述问题只需做到如下一点：保证对相同key的访问会被发送到相同的服务器。很多方法可以实现这一点，最常用的方法是计算哈希。例如对于每次访问，可以按如下算法计算其哈希值：

h = Hash(key) % 3

    其中Hash是一个从字符串到正整数的哈希映射函数。这样，如果我们将Memcached Server分别编号为0、1、2，那么就可以根据上式和key计算出服务器编号h，然后去访问。

这个方法虽然解决了上面提到的两个问题，但是存在一些其它的问题。如果将上述方法抽象，可以认为通过：

h = Hash(key) % N

    这个算式计算每个key的请求应该被发送到哪台服务器，其中N为服务器的台数，并且服务器按照0 – (N-1)编号。

    这个算法的问题在于容错性和扩展性不好。所谓容错性是指当系统中某一个或几个服务器变得不可用时，整个系统是否可以正确高效运行；而扩展性是指当加入新的服务器后，整个系统是否可以正确高效运行。

   现假设有一台服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个key就要按h = Hash(key) % (N-1)重新计算；同样，如果新增了一台服务器，虽然原有服务器编号不用改变，但是要按h = Hash(key) % (N+1)重新计算哈希值。因此系统中一旦有服务器变更，大量的key会被重定位到不同的服务器从而造成大量的缓存不命中。而这种情况在分布式系统中是非常糟糕的。

一个设计良好的分布式哈希方案应该具有良好的单调性，即服务节点的增减不会造成大量哈希重定位。一致性哈希算法就是这样一种哈希方案。

Hash 算法的一个衡量指标是单调性（ Monotonicity ），定义如下：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

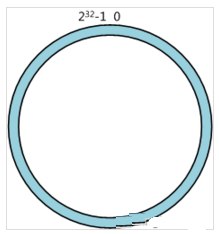
容易看到，上面的简单 hash 算法 hash(object)%N 难以满足单调性要求

**三、一致性哈希算法的理解**

1、算法简述

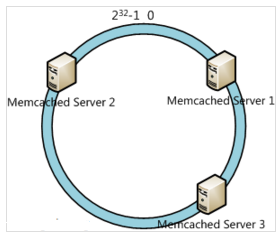
     一致性哈希算法(Consistent Hashing Algorithm)是一种分布式算法，常用于负载均衡。Memcached client也选择这种算法，解决将key-value均匀分配到众多Memcached server上的问题。它可以取代传统的取模操作，解决了取模操作无法应对增删Memcached Server的问题(增删server会导致同一个key,在get操作时分配不到数据真正存储的server，命中率会急剧下降)。

  简单来说，一致性哈希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），整个哈希空间环如下：



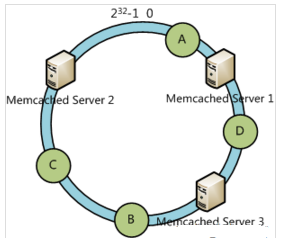
    整个空间按顺时针方向组织。0和(2^32)-1在零点中方向重合。

    下一步将各个服务器使用H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，这里假设将上文中三台服务器使用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H计算出哈希值h，通根据h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。

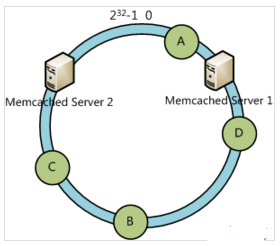
    例如我们有A、B、C、D四个数据对象，经过哈希计算后，在环空间上的位置如下：



 根据一致性哈希算法，数据A会被定为到Server 1上，D被定为到Server 3上，而B、C分别被定为到Server 2上。

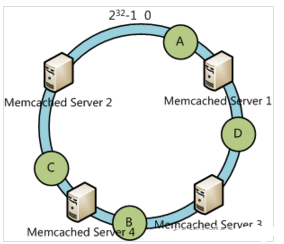
2、容错性与可扩展性分析

   下面分析一致性哈希算法的容错性和可扩展性。现假设Server 3宕机了：



  可以看到此时A、C、B不会受到影响，只有D节点被重定位到Server 2。一般的，在一致性哈希算法中，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（即顺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。

  下面考虑另外一种情况，如果我们在系统中增加一台服务器Memcached Server 4：

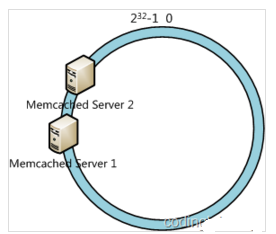


     此时A、D、C不受影响，只有B需要重定位到新的Server 4。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

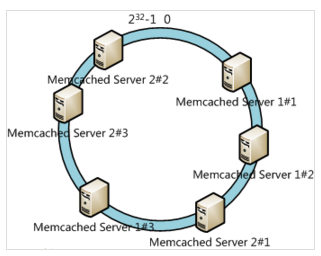
    综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

3、虚拟节点

    一致性哈希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问题。例如我们的系统中有两台服务器，其环分布如下：



   此时必然造成大量数据集中到Server 1上，而只有极少量会定位到Server 2上。为了解决这种数据倾斜问题，一致性哈希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，我们决定为每台服务器计算三个虚拟节点，于是可以分别计算“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”、“Memcached Server 2#1”、“Memcached Server 2#2”、“Memcached Server 2#3”的哈希值，于是形成六个虚拟节点：



 4、JAVA实现

|  |
| --- |
| import java.security.MessageDigest;  import java.security.NoSuchAlgorithmException;  import java.util.Collection;  import java.util.SortedMap;  import java.util.TreeMap;  /\*\*  \* 一致性Hash算法  \*  \* @param <T> 节点类型  \*/  public class ConsistentHash<T> {  /\*\*  \* Hash计算对象，用于自定义hash算法  \*/  HashFunc hashFunc;  /\*\*  \* 复制的节点个数  \*/  private final int numberOfReplicas;  /\*\*  \* 一致性Hash环  \*/  private final SortedMap<Long, T> circle = new TreeMap<>();  /\*\*  \* 构造，使用Java默认的Hash算法  \* @param numberOfReplicas 复制的节点个数，增加每个节点的复制节点有利于负载均衡  \* @param nodes 节点对象  \*/  public ConsistentHash(int numberOfReplicas, Collection<T> nodes) {  this.numberOfReplicas = numberOfReplicas;  this.hashFunc = new HashFunc() {  @Override  public Long hash(Object key) {  // return fnv1HashingAlg(key.toString());  return md5HashingAlg(key.toString());  }  };  //初始化节点  for (T node : nodes) {  add(node);  }  }  /\*\*  \* 构造  \* @param hashFunc hash算法对象  \* @param numberOfReplicas 复制的节点个数，增加每个节点的复制节点有利于负载均衡  \* @param nodes 节点对象  \*/  public ConsistentHash(HashFunc hashFunc, int numberOfReplicas, Collection<T> nodes) {  this.numberOfReplicas = numberOfReplicas;  this.hashFunc = hashFunc;  //初始化节点  for (T node : nodes) {  add(node);  }  }  /\*\*  \* 增加节点<br>  \* 每增加一个节点，就会在闭环上增加给定复制节点数<br>  \* 例如复制节点数是2，则每调用此方法一次，增加两个虚拟节点，这两个节点指向同一Node  \* 由于hash算法会调用node的toString方法，故按照toString去重  \*  \* @param node 节点对象  \*/  public void add(T node) {  for (int i = 0; i < numberOfReplicas; i++) {  circle.put(hashFunc.hash(node.toString() + i), node);  }  }  /\*\*  \* 移除节点的同时移除相应的虚拟节点  \*  \* @param node 节点对象  \*/  public void remove(T node) {  for (int i = 0; i < numberOfReplicas; i++) {  circle.remove(hashFunc.hash(node.toString() + i));  }  }  /\*\*  \* 获得一个最近的顺时针节点  \*  \* @param key 为给定键取Hash，取得顺时针方向上最近的一个虚拟节点对应的实际节点  \* @return 节点对象  \*/  public T get(Object key) {  if (circle.isEmpty()) {  return null;  }  long hash = hashFunc.hash(key);  if (!circle.containsKey(hash)) {  SortedMap<Long, T> tailMap = circle.tailMap(hash); //返回此映射的部分视图，其键大于等于 hash  hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();  }  //正好命中  return circle.get(hash);  }  /\*\*  \* 使用MD5算法  \* @param key  \* @return  \*/  private static long md5HashingAlg(String key) {  MessageDigest md5 = null;  try {  md5 = MessageDigest.getInstance("MD5");  md5.reset();  md5.update(key.getBytes());  byte[] bKey = md5.digest();  long res = ((long) (bKey[3] & 0xFF) << 24) | ((long) (bKey[2] & 0xFF) << 16) | ((long) (bKey[1] & 0xFF) << 8)| (long) (bKey[0] & 0xFF);  return res;  } catch (NoSuchAlgorithmException e) {  e.printStackTrace();  }  return 0l;  }  /\*\*  \* 使用FNV1hash算法  \* @param key  \* @return  \*/  private static long fnv1HashingAlg(String key) {  final int p = 16777619;  int hash = (int) 2166136261L;  for (int i = 0; i < key.length(); i++)  hash = (hash ^ key.charAt(i)) \* p;  hash += hash << 13;  hash ^= hash >> 7;  hash += hash << 3;  hash ^= hash >> 17;  hash += hash << 5;  return hash;  }  /\*\*  \* Hash算法对象，用于自定义hash算法  \*/  public interface HashFunc {  public Long hash(Object key);  }  } |

Consistent Hashing最大限度地抑制了hash键的重新分布。另外要取得比较好的负载均衡的效果，往往在服务器数量比较少的时候需要增加虚拟节点来保证服务器能均匀的分布在圆环上。因为使用一般的hash方法，服务器的映射地点的分布非常不均匀。使用虚拟节点的思想，为每个物理节点（服务器）在圆上分配100～200个点。这样就能抑制分布不均匀，最大限度地减小服务器增减时的缓存重新分布。用户数据映射在虚拟节点上，就表示用户数据真正存储位置是在该虚拟节点代表的实际物理服务器上。

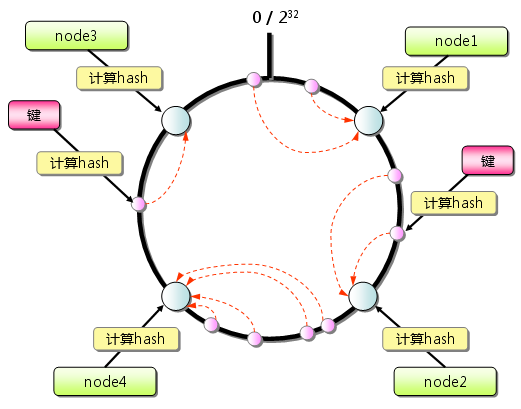
# 第二篇

## 一致性Hash算法背景

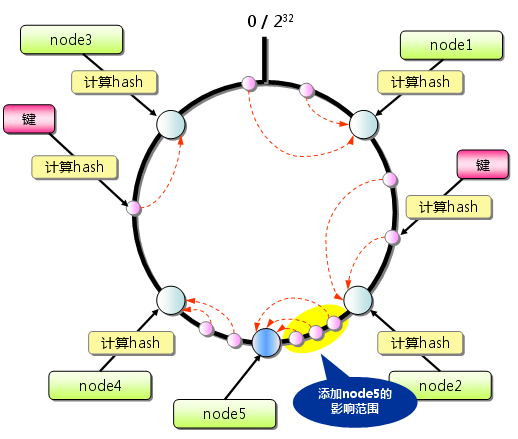
　　一致性哈希算法在1997年由麻省理工学院的Karger等人在解决分布式Cache中提出的，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP使用的简单哈希算法带来的问题，使得DHT可以在P2P环境中真正得到应用。

　　但现在一致性hash算法在分布式系统中也得到了广泛应用，研究过memcached缓存数据库的人都知道，memcached服务器端本身不提供分布式cache的一致性，而是由客户端来提供，具体在计算一致性hash时采用如下步骤：

1. 首先求出memcached服务器（节点）的哈希值，并将其配置到0～232的圆（continuum）上。
2. 然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
3. 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过232仍然找不到服务器，就会保存到第一台memcached服务器上。



　　从上图的状态中添加一台memcached服务器。余数分布式算法由于保存键的服务器会发生巨大变化而影响缓存的命中率，但Consistent Hashing中，只有在园（continuum）上增加服务器的地点逆时针方向的第一台服务器上的键会受到影响，如下图所示：



## 

## 一致性Hash性质

　　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，如何保证当系统的节点数目发生变化时仍然能够对外提供良好的服务，这是值得考虑的，尤其实在设计分布式缓存系统时，如果某台服务器失效，对于整个系统来说如果不采用合适的算法来保证一致性，那么缓存于系统中的所有数据都可能会失效（即由于系统节点数目变少，客户端在请求某一对象时需要重新计算其hash值（通常与系统中的节点数目有关），由于hash值已经改变，所以很可能找不到保存该对象的服务器节点），因此一致性hash就显得至关重要，良好的分布式cahce系统中的一致性hash算法应该满足以下几个方面：

* **平衡性(Balance)**

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

* **单调性(Monotonicity)**

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲区加入到系统中，那么哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲区中去，而不会被映射到旧的缓冲集合中的其他缓冲区。简单的哈希算法往往不能满足单调性的要求，如最简单的线性哈希：x = (ax + b) mod (P)，在上式中，P表示全部缓冲的大小。不难看出，当缓冲大小发生变化时(从P1到P2)，原来所有的哈希结果均会发生变化，从而不满足单调性的要求。哈希结果的变化意味着当缓冲空间发生变化时，所有的映射关系需要在系统内全部更新。而在P2P系统内，缓冲的变化等价于Peer加入或退出系统，这一情况在P2P系统中会频繁发生，因此会带来极大计算和传输负荷。单调性就是要求哈希算法能够应对这种情况。

* **分散性(Spread)**

在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

* **负载(Load)**

负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

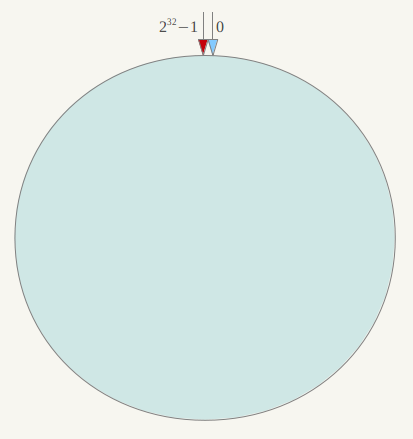
* **平滑性(Smoothness)**

平滑性是指缓存服务器的数目平滑改变和缓存对象的平滑改变是一致的。

## 原理

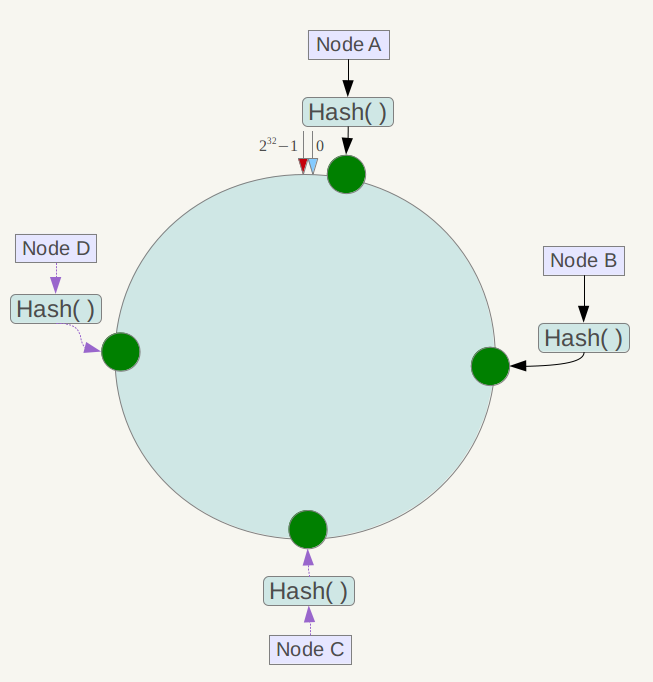
### 基本概念

　　一致性哈希算法（Consistent Hashing）最早在论文《[Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web](http://dl.acm.org/citation.cfm?id=258660)》中被提出。简单来说，一致性哈希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），整个哈希空间环如下：



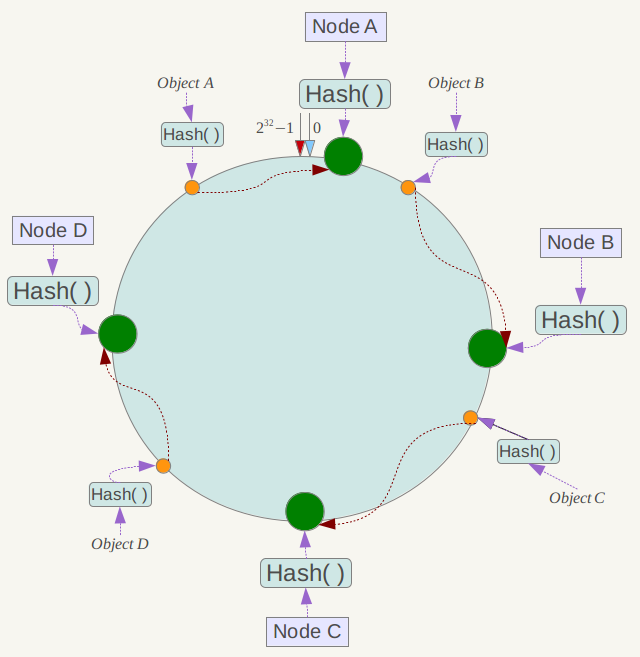
　　整个空间按顺时针方向组织。0和232-1在零点中方向重合。

　　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。

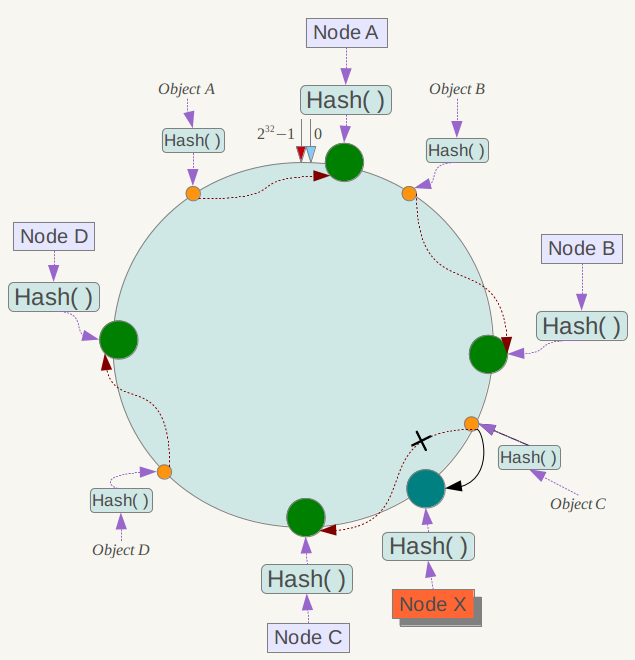
　　例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法，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。

下面分析一致性哈希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，在一致性哈希算法中，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。

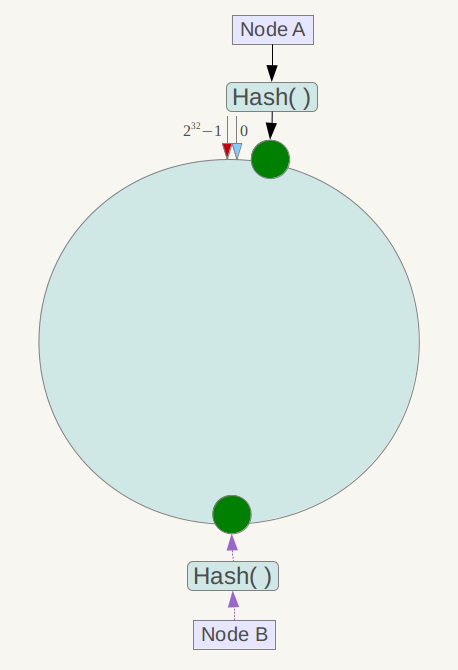
下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



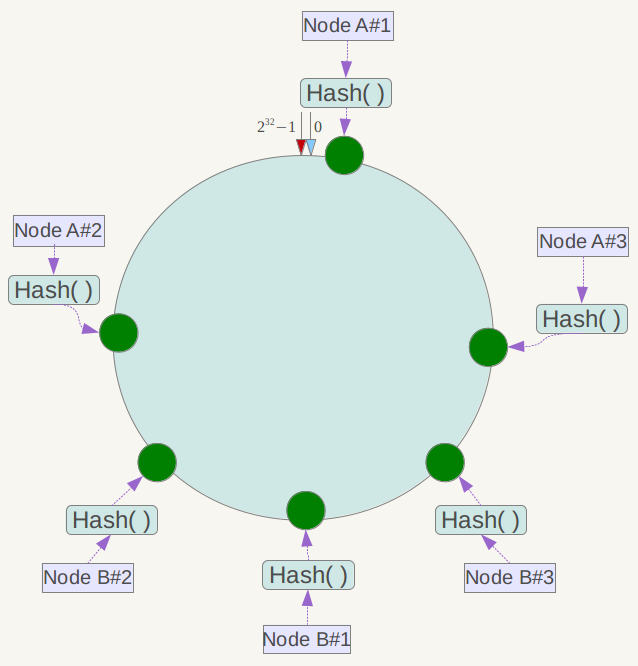
此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

综上所述，一致性哈希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。

另外，一致性哈希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，一致性哈希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。