Consistent Hashing算法入门及PHP代码实现

在设计一个分布式系统的架构时，为了提高系统的负载能力，需要把不同的数据分发到不同的服务节点上。因此这里就需要一种分发的机制，其实就是一种算法，来实现这种功能。这里我们就用到了Consistent Hashing算法。

在正式介绍Consistent Hashing算法之前我们先来看一个简单的hash算法，就是用取余数的方式来选择节点。具体的步骤如下：

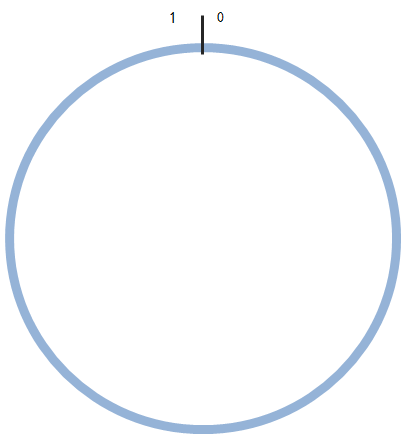
1. 根据集群服务的节点数创建一个哈希表
2. 然后根据键名计算出键名的整数哈希值，用该哈希值对节点数取余。
3. 最后根据余数在哈希表中取出节点。

假设在一个集群中有n个服务器节点，对这些节点编号为0,1,2,…,n-1 。然后，将一条数据(key,value)存储到服务器中。这时我们该如何来选择服务器节点呢？根据上面的步骤我们需要对key计算hash值，然后再对n（节点个数）取余数。最后得到的值就是我们所要的节点。用一个公式来表示：num = hash(key) % n。hash()是一个计算hash值的函数，这里对hash()函数还是有一定的要求的，如果我们使用的hash()函数很优化的话，那计算出的num是均匀分布在0,1,2,…,n-1之间的，从而使尽可能多的服务器节点都能被使用。而不是所有的数据都集中在一个或者几个服务器节点上面。具体的hash()实现不是本章讨论的重点。

这种单纯的取余数的方式虽然简单，但是如果将其应用到实际生产系统中会出现很大的问题。假设我们有23个服务节点。那么根据上面的方式，一个key映射到每个节点的概率都是1/23。假设增加了一个服务节点的话，之前的hash(key) % n 就会变成hash(key) % (n+1) 。也就是说对于key来说有23/24的概率会被重新分配到新的节点。相反只会有1/24的概率会被分配到原节点。同样，当你减少一个节点的时候，有22/23的概率会被重新分配到新的节点上去。

鉴于这种情况，就需要有一种方式来避免或者减少在横向扩展的时候命中率降低的情况的发生。这种方法就是我们将要介绍的Consistent Hashing算法，我们称其为一致性hash算法。

为了了解Consistent Hashing算法是如何工作的，我们假设单位区间 [ 0 , 1 ) 依顺时针的方向均匀的分布在圆上

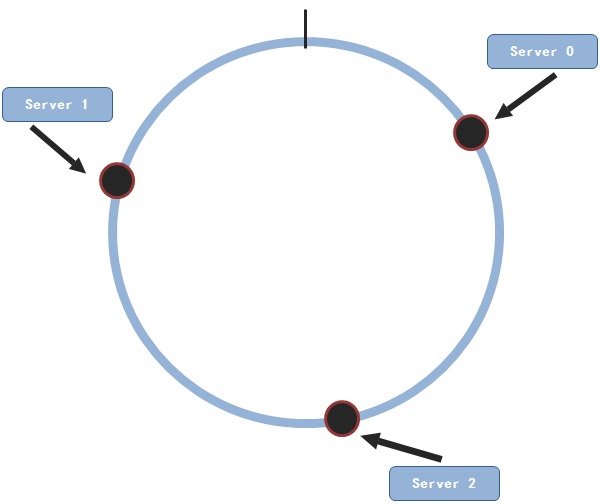


假使有n个服务节点，为每个服务节点编号为0, 1, 2, …, n-1。然后我们需要有一个hash()函数来对服务节点计算hash值。如果选用的hash()函数返回值的取值范围为[ 0, R )，那么使用公式 v = hash(n) / R。这样得到的v会分布在单位区间[ 0, 1 )内。所以，通过这个方式就可以使我们的服务节点分布在圆上面。

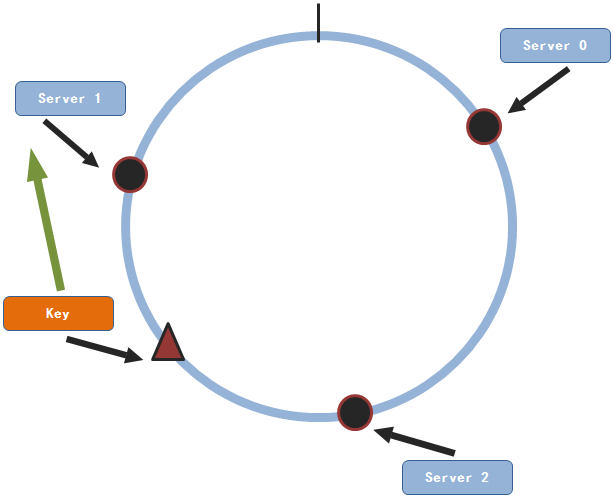
当然，以单位区间[ 0, 1 ) 画圆只是一种方式，还有很多其他的画圆方式，比如说：以区间[ 0, 2^32-1 ) 为圆，然后使用hash()函数对服务节点计算hash()值。选用的hash()函数产生的值当然也必须在0 – (2^32-1) 范围之内了。

这里我们还是以[ 0, 1 )为例来介绍。

我们以3个服务节点为例来进行说明

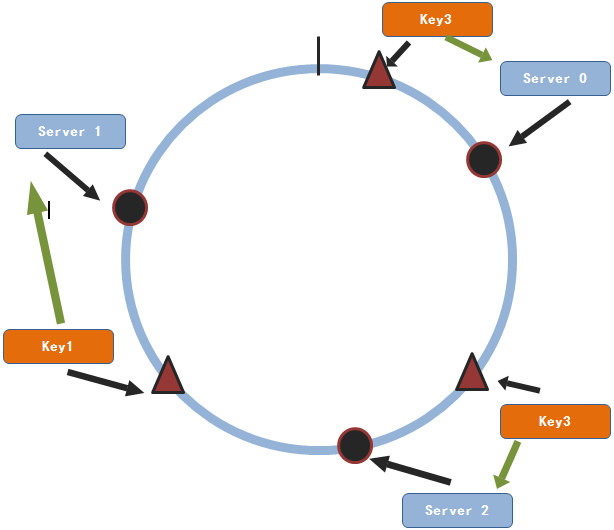


这三个节点随机的分布在这个圆上面。现在假设我们有一条数据(key,value)需要存储，接下来要做的就是将这条数据通过同样的方法映射到圆上面。



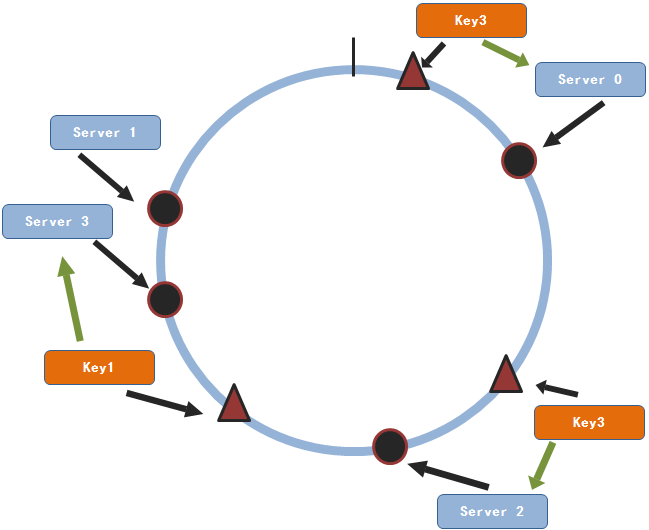
然后从key所坐落在圆上的位置开始顺时针查找服务节点所在的位置，找到的第一个服务节点即是要存储的节点。所以说这条数据将要存储在服务节点1上。

同理，当有其它的(key,value)对需要存储的时候，也是按照上面的方式进行服务节点的选择。



现在我们来看该方法对于我们刚开始提到的横向扩展的问题是否能够很好的解决呢？

假设我们需要增加一个服务节点3



通过上图，我们可以看出，只有key1会改变其存储服务节点。对于大部分的数据来说依然会找到原先的节点。因此，对于n个服务节点的集群来说，当有服务节点增加的时候一条数据只有1/(n+1)的概率会改变其存储的服务节点。这个概率远比取余数法所得的概率要小的多。同样，减少一个服务节点和增加服务节点的原理是相同的，其每条数据重新选择服务节点的概率为1/(n-1)。同样这个概率也是很小的。

下面就用一段php代码来简单的实现这个过程

$nodes = array('192.168.5.201','192.168.5.102','192.168.5.111');

$keys = array('onmpw', 'jiyi', 'onmpw\_key', 'jiyi\_key', 'www','www\_key','key1');

$buckets = array(); //节点的hash字典

$maps = array(); //存储key和节点之间的映射关系

/\*\*

\* 生成节点字典 —— 使节点分布在单位区间[0,1)的圆上

\*/

foreach( $nodes as $key) {

$crc = crc32($key)/pow(2,32); // CRC値

$buckets[] = array('index'=>$crc,'node'=>$key);

}

/\*

\* 根据索引进行排序

\*/

sort($buckets);

/\*

\* 对每个key进行hash计算，找到其在圆上的位置

\* 然后在该位置开始依顺时针方向找到第一个服务节点

\*/

foreach($keys as $key){

$flag = false; //表示是否有找到服务节点

$crc = crc32($key)/pow(2,32);//计算key的hash值

for($i = 0; $i < count($buckets); $i++){

if($buckets[$i]['index'] > $crc){

/\*

\* 因为已经对buckets进行了排序

\* 所以第一个index大于key的hash值的节点即是要找的节点

\*/

$maps[$key] = $buckets[$i]['node'];

$flag = true;

break;

}

}

if(!$flag){

//没有找到，则使用buckets中的第一个服务节点

$maps[$key] = $buckets[0]['node'];

}

}

foreach($maps as $key=>$val){

echo $key.'=>'.$val,"<br />";

}

这段代码运行的结果如下

onmpw=>192.168.5.102  
jiyi=>192.168.5.201  
onmpw\_key=>192.168.5.201  
jiyi\_key=>192.168.5.102  
www=>192.168.5.201  
www\_key=>192.168.5.201  
key1=>192.168.5.111

然后我们添加一个服务节点，修改代码如下

$nodes = array('192.168.5.201','192.168.5.102','192.168.5.111','192.168.5.11');

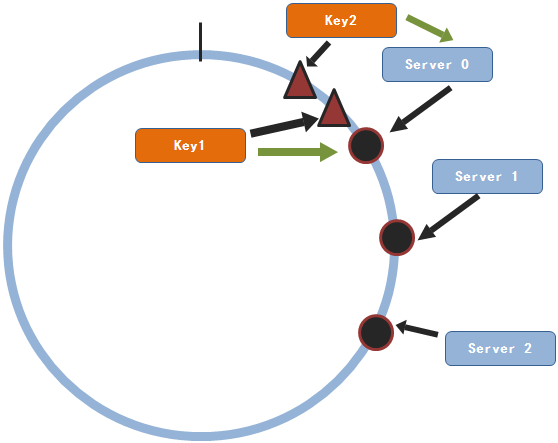
其它代码不变，继续运行结果如下

onmpw=>192.168.5.102  
jiyi=>192.168.5.201  
onmpw\_key=>192.168.5.11  
jiyi\_key=>192.168.5.102  
www=>192.168.5.201  
www\_key=>192.168.5.201  
key1=>192.168.5.111

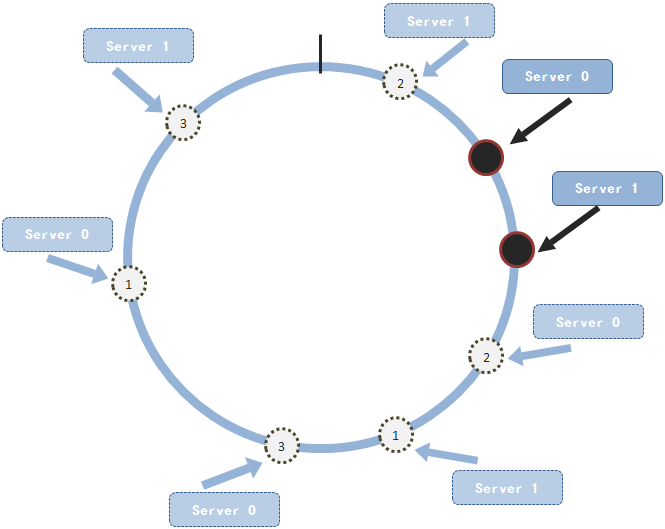
我们看到，只有onmpw\_key重新选择了服务节点。其它的都是原先的节点。

到这里我们看到，较之于取余数法命中的概率提高了相当多了。那这里是不是就解决了我们前面遇到的问题了呢？

其实，还没有。因为这些值的分布毕竟不是那么的均匀。在系统中有可能这些服务节点分布非常的集中，这可能导致的情况就是所有的key都映射到其中的一个或者几个节点上面，剩下的服务节点都没有被用到。虽然这并不是什么很严重的问题，那为什么我们要浪费哪怕只是一台服务器呢。



我们看，这种情况就造成了数据集中在一个服务节点上面，造成了其它服务节点的浪费。那如何解决这个问题呢？人们就又想出了一种新的方式：就是为每个节点建立虚拟的节点。什么意思呢？就是说对于节点j，为其建立m个复制品。这m个复制出来的节点都通过hash()函数得出不同的hash值，但是每个虚拟节点保存的节点信息都是节点j的。然后这些虚拟节点都会随机的分布在圆上面。举例子来说，我们有两个服务节点。并且为每个节点都复制出三个虚拟节点。这些节点（包括虚拟节点都随机的分布在圆上面）



这样看起来服务节点在圆上分布还是比较均匀的了。其实，总结起来就是在上面的那种方式上稍微做了一下改进——给每个节点复制一些虚拟节点。

因此，我们的代码也不需要做过多的修改。为了看代码比较直观，我在这里还是将整段代码罗列在这。

$nodes = array('192.168.5.201','192.168.5.102','192.168.5.111');

$keys = array('onmpw', 'jiyi', 'onmpw\_key', 'jiyi\_key', 'www','www\_key','key1');

//添加的变量 修改的地方

$replicas = 160; //每个节点的复制的个数

$buckets = array(); //节点的hash字典

$maps = array(); //存储key和节点之间的映射关系

/\*\*

\* 生成节点字典 —— 使节点分布在单位区间[0,1)的圆上

\*/

foreach( $nodes as $key) {

//修改的地方

for($i=1;$i<=$replicas;$i++){

$crc = crc32($key.'.'.$i)/pow(2,32); // CRC値

$buckets[] = array('index'=>$crc,'node'=>$key);

}

}

/\*

\* 根据索引进行排序

\*/

sort($buckets);

/\*

\* 对每个key进行hash计算，找到其在圆上的位置

\* 然后在该位置开始依顺时针方向找到第一个服务节点

\*/

foreach($keys as $key){

$flag = false; //表示是否有找到服务节点

$crc = crc32($key)/pow(2,32);//计算key的hash值

for($i = 0; $i < count($buckets); $i++){

if($buckets[$i]['index'] > $crc){

/\*

\* 因为已经对buckets进行了排序

\* 所以第一个index大于key的hash值的节点即是要找的节点

\*/

$maps[$key] = $buckets[$i]['node'];

$flag = true;

break;

}

}

if(!$flag){

//没有找到，则使用buckets中的第一个服务节点

$maps[$key] = $buckets[0]['node'];

}

}

foreach($maps as $key=>$val){

echo $key.'=>'.$val,"<br />";

}

有改动的地方在代码里已经标注出来了。可以看到，修改的地方还是比较少的。

至此，相信大家对Consistent Hashing应该有了一个比较清晰的认识。hash算法的用处还是很广泛的，比如在memcache集群，nginx负载等方面都有用到。所以，了解hash算法对于我们是有很大的帮助的。

上述算法过程的表述有不清楚或者不合适的地方，欢迎大家不吝赐教。