Ketama一致性Hash算法(含Java代码)

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)，详细的介绍在这篇帖子中http://www.iteye.com/topic/611976(后文指代这篇文章的地方均称为引文)。

［下面以Memcached的分布式问题为讨论点，但将Memcached server抽象为节点(Node)］
引文中描述的一致性Hash算法有个潜在的问题是:

将节点hash后会不均匀地分布在环上，这样大量key在寻找节点时，会存在key命中各个节点的概率差别较大，无法实现有效的负载均衡。

如有三个节点Node1,Node2,Node3，分布在环上时三个节点挨的很近，落在环上的key寻找节点时，大量key顺时针总是分配给Node2，而其它两个节点被找到的概率都会很小。

这种问题的解决方案可以有:

改善Hash算法，均匀分配各节点到环上；［引文］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。

在查看Spy Memcached client时，发现它采用一种称为Ketama的Hash算法，以虚拟节点的思想，解决Memcached的分布式问题。

对Ketama的介绍

引用

Ketama is an implementation of a consistent hashing algorithm, meaning you

can add or remove servers from the memcached pool without causing a complete remap of all keys.
Here's how it works:
* Take your list of servers (eg: 1.2.3.4:11211, 5.6.7.8:11211, 9.8.7.6:11211)
* Hash each server string to several (100-200) unsigned ints
* Conceptually, these numbers are placed on a circle called the continuum. (imagine a clock face that goes from 0 to 2^32)
* Each number links to the server it was hashed from, so servers appear at several points on the continuum, by each of the numbers they hashed to.
* To map a key->server, hash your key to a single unsigned int, and find the next biggest number on the continuum. The server linked to that number is the correct server for that key.
* If you hash your key to a value near 2^32 and there are no points on the continuum greater than your hash, return the first server in the continuum.
If you then add or remove a server from the list, only a small proportion of keys end up mapping to different servers.

下面以Spy Memcached中的代码为例来说明这种算法的使用

该client采用TreeMap存储所有节点，模拟一个环形的逻辑关系。在这个环中，节点之前是存在顺序关系的，所以TreeMap的key必须实现Comparator接口。
那节点是怎样放入这个环中的呢?

上面的流程大概可以这样归纳:四个虚拟结点为一组，以getKeyForNode方法得到这组虚拟节点的name，Md5编码后，每个虚拟结点对应Md5码16个字节中的4个，组成一个long型数值，做为这个虚拟结点在环中的惟一key。第12行k为什么是Long型的呢？呵呵，就是因为Long型实现了Comparator接口。

处理完正式结点在环上的分布后，可以开始key在环上寻找节点的游戏了。对于每个key还是得完成上面的步骤:计算出Md5，根据Md5的字节数组，通过Kemata Hash算法得到key在这个环中的位置。

```
1.        final Node rv;
```

```
2.          byte[] digest = hashAlg.computeMd5(keyValue);
3.          Long key = hashAlg.hash(digest, 0);
4.
5.          if(!ketamaNodes.containsKey(key)) {
6.
7.              SortedMap<Long, Node> tailMap=ketamaNodes.tailMap(key);
8.              if(tailMap.isEmpty()) {
9.                  key=ketamaNodes.firstKey();
10.             } else {
11.                 key=tailMap.firstKey();
12.             }
13.
14.
15.
16.
17.
18.
19.         }
20.
21.
22.         rv=allNodes.get(key);
```

引文中已详细描述过这种取节点逻辑:在环上顺时针查找，如果找到某个节点，就返回那个节点;如果没有找到，则取整个环的第一个节点。

测试结果
测试代码是自己整理的，主体方法没有变

分布平均性测试:测试随机生成的众多key是否会平均分布到各个结点上
测试结果如下:

1. Nodes count : 5, Keys count : 100000, Normal percent : 20.0%
2. ---------- boundary ----------
3. Node name :node1 - Times : 20821 - Percent : 20.821001%

4. Node name :node3 - Times : 19018 - Percent : 19.018%
5. Node name :node5 - Times : 19726 - Percent : 19.726%
6. Node name :node2 - Times : 19919 - Percent : 19.919%
7. Node name :node4 - Times : 20516 - Percent : 20.516%

最上面一行是参数说明，节点数目，总共有多少key，每个节点应该分配key的比例是多少。下面是每个结点分配到key的数目和比例。
多次测试后发现，这个Hash算法的节点分布还是不错的，都在标准比例左右徘徊，是个合适的负载均衡算法。

节点增删测试:在环上插入N个结点，每个节点nCopies个虚拟结点。随机生成众多key，在增删节点时，测试同一个key选择相同节点的概率
测试如果如下:

1. Normal **case** : nodes count : 50
2. Added **case** : nodes count : 51
3. Reduced **case** : nodes count : 49
4. ---------- boundary ----------
5. Same percent in added **case** : 93.765%
6. Same percent in reduced **case** : 93.845%

上面三行分别是正常情况，节点增加，节点删除情况下的节点数目。下面两行表示在节点增加和删除情况下，同一个key分配在相同节点上的比例(命中率)。
多次测试后发现，命中率与结点数目和增减的节点数量有关。同样增删结点数目情况下，结点多时命中率高。同样节点数目，增删结点越少，命中率越高。这些都与实际情况相符。

附件为Ketama算法的Java代码及测试代码