# 一致性hash算法(consistent hashing

# )

## 基本场景

比如你有N台cache服务器（后面简称cache），那么如何将一个对象object映射到N个cache上呢，通常我们会采用取模计算 hash(key)%N 这样将object 映射到N台cache上；

一切运行正常，假如出现以下两种情况:

1. 一个cache服务器M down掉了（实际应用中应该考虑这种情况），这样所有映射到cache M的对象都会失效，怎么办，需要把cacheM从cache中移除，这时候服务器剩余N-1台，映射算法变为 hash(kjey)%(N-1);
2. 由于访问加重，需要新加一台cache，这时候映射公式变为hash(key)%(N+1);

上面两种情况意味着什么?这意味着突然之间所有的cache都将会失效，所有的缓存都需重新去存放，所有的访问都将会再次去数据库读取一次，以前的cache中也存在相同的无用信息

考虑第三种情况，由于硬件越来越强，你想要后面添加的cache负担更多的请求，显然上面的算法是无法做到的。

所以有了consistenthashing

## hash算法的和单调性

Hash算法的一个衡量指标是单调性，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经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它缓冲区。

容易看到，上面的简单hash算法hash(key)%N难以满足单调性的要求。

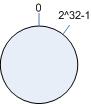
## Consistent hashing算法的原理

Congsistent hashing 是一种hash算法，简单的说，在添加/移除一个cache时，它能够尽可能小的改变已经存在key的映射关系，尽可能的满足单调性的要求。

原理如下:

1.环形的hash空间

考虑通常的hash算法是将value映射到一个32位的key值，也就是0~2^32-1次方的数值空间；我们可以把这个空间想象成一个首尾相连的圆环，如下图所示



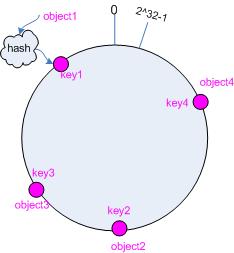
2.把对象 映射到hash空间

假设我们有四个对象 object1~object4，通过hash函数计算出的hash值key在圆环上的分布如下图所示

Hash(object1) = key1

。。。。

Hash(object4)=key4



3.把cache映射到hash空间

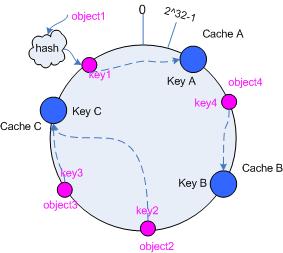
Consistent hashing 的基本思想就是将对象和cache都映射到同一个hash数值空间，并使用相同的hash算法

假设当前有A\B\C三台cache，那么映射的结果如下图所示，他们在hash空间中以对应的hash值排列

Hash（cacheA）=keyA

。。。

Hash(cacheC) = keyC



Cache的hash计算，一般的方法可以使用cache机器的IP地址或者机器名作为hash输入。

4.把对象映射到cache

现在cache和对象都在同一个hash环空间了，接下来需要考虑的就是如何将对象映射到cache上面了。

在这个环形空间中，如果沿着顺时针方向从对象的key值出发，直到遇到一个cache，那么就将该对象存储在这个cache上，因为对象和cache的hash值是固定的因此这个cache必然是唯一和正确的。这样不就找到了对象和cache的映射方法了么?

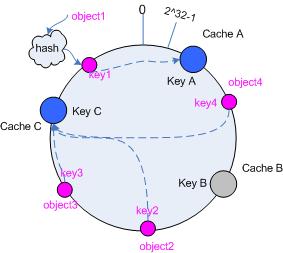
参考上面的图，根据上面的方法，对象object1将被映射到cacheA上；object2和object3对应到cacheC；object4对应到cacheB

5 考察cache的变动

前面说的，通过hash然后取余方法带来的最大问题就在不不能满足单调性，当cache变动时cache会失效，进而对后台服务器造成巨大的冲击，现在来分析下consistent hashing算法

5.1 移除cache

考虑假设cacheB挂掉了，根据上面讲到的映射方法，这时受影响的将仅是cacheA 到cahceB之间的对象，这些本来需要映射到cacheB上的对象，将重新映射到cacheC上。



5.2添加cache

考虑添加一台新的cacheD的情况，假设在这个环形hash空间中，cacheD背影设在对象object2和object3直接，这时候受到影响的是cacheB到cacheD之间的对象，这部分本来映射到cacheC上的对象将会映射到cacheD上即可。

## 虚拟节点

考量hash算法的另一个指标是平衡性，定义如下：

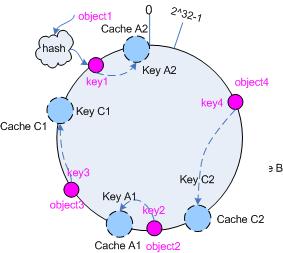
平衡性： 平衡性是指哈希的结果能够尽可能分不到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。

Hash算法并不是保证绝对的平衡，如果cache较少的话，对象不能均匀的映射到cache上，比如在上面的例子中，仅仅部署cacheA和cacheC的情况下，4个对象只有一个缓存到cacheA上，其它的都缓存在cacheC上，这样的分布是很不均衡的。

为了解决这种情况，congsistent hashing 引入了虚拟节点的概念，它的定于如下：

“虚拟节点”是实际节点的复制品，一个实际节点对应了若干个”虚拟节点”，这个对应个数也成为”复制个数”，”虚拟节点”在hash空间中以hash值排列。

仍以仅仅部署cacheA和cacheC的情况为例，在上图中我们已经看到，cache分布并不均匀。现在我们引入虚拟节点，并设置“复制个数”为2，这意味着一共会出现4个”虚拟节点”，cacheA1、cacheA2代表了cacheA；cacheC1、cacheC2代表了cacheC；假设一种比较理想的情况，下图

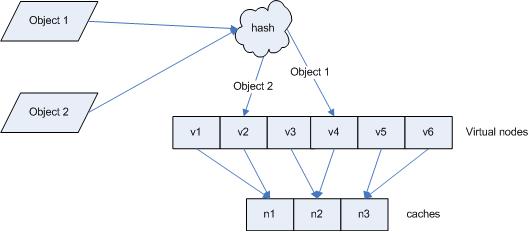


这个时候，对象到虚拟节点的映射情况为：

Object1->cacheA2; object2->cacheA2; object3->cacheC1; object4->cacheC2

因此对象object1和object2都被映射到cacheA上，而object3和object4都被映射到cacheC上，平衡性有了很大的提高。

引入”虚拟节点”后，映射关系就从{对象-节点} 转换到了{对象->虚拟节点}。查询对象所在的cache时的映射关系如下：



虚拟节点的hash计算可以采用对应节点的IP地址加数字后缀的方式。例如cacheA的ip地址为 202.168.12.202.

引入虚拟节点前，计算cacheA的hash值为Hash(“202.168.12.202”)

引入虚拟节点后，计算虚拟机的点A1和A2的hash值：

Hash(“202.168.12.202#1”)

Hash(“202.168.12.202#2”ss)