一致性HASH算法解析

1、为什么要用一致性HASH算法？
问题场景：比如有N个cache服务器，那么如何将一个对象object映射到N个cache上呢？传统的通用方法是计算object的hash值，然后均匀的映射到N个cache上：hash(object)%N，这样会出现两种情况：
第一：N个cache服务器中有一个down掉，这时所有映射到m的对象都会失效，需要把cache m从cache中移除，映射公式变成了hash(object)%(N-1);
第二：由于访问加重，需要添加cache，这时cache服务器变成N+1，映射公式变成hash(object)%(N+1)；
这两种情况下意味着什么呢？意味着突然之间所有的cache全都失效了，对于服务器而言，这是一场灾难。于是产生了一致性HASH算法。
2、一致性HASH算法解决了哪些问题？
第一：在移除和添加一个cache时，能够尽可能小的改变已经存在的KEY映射关系，尽可能满足HASH算法的单调性要求。HASH算法的单调性是指如何已经有一些内容通过HASH分配到相应的缓冲区，又有新的缓冲加入到系统中。HASH的结果应该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。
第二：能够保证HASH算法的平衡性要求。HASH算法的平衡性是指HASH的结果能够尽可能分布到所有的缓冲中去，使得所有的缓冲空间都得到利用。
3、一致性HASH是怎样实现去保证这些问题的解决。
一致性HASH的实现原理：
环形HASH空间。

考虑通常的 hash 算法都是将 value 映射到一个 32 为的 key 值，也即是 0~2^32-1 次方的数值空间；我们可以将这个空间想象成一个首（ 0 ）尾（ 2^32-1 ）相接的圆环，如下面图 1 所示的那样。

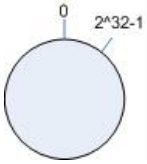

图 1： 环形 HASH空间

把对象映射到HASH空间

接下来考虑 4 个对象 object1~object4 ， 通过 hash 函数计算出的 hash 值 key 在环上的分布如图 2 所示。

hash(object1) = key1;

… …

hash(object4) = key4;

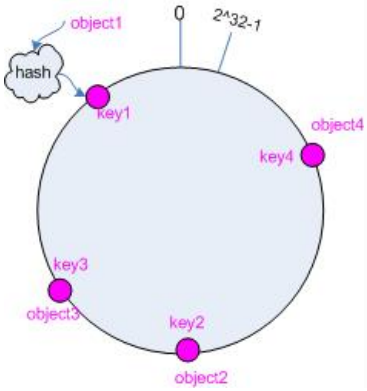

图 2： 4 个对象的 key 值分布

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 数值空间中，并且使用相同的hash 算法。

假设当前有 A,B 和 C 共 3 台 cache ， 那么其映射结果将如图 3 所示，他们在 hash 空间中，以对应的 hash值排列。

hash(cache A) = key A;

… …

hash(cache C) = key C;

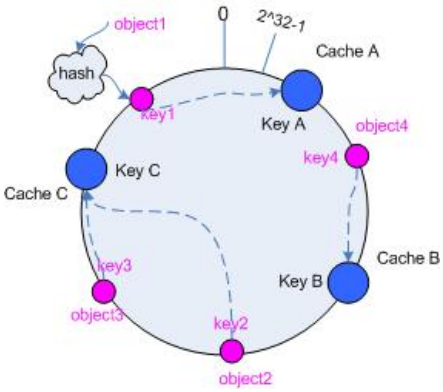

图 3 cache 和对象的 key 值分布

说到这里，顺便提一下 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。

把对象映射到cache

现在 cache 和对象都已经通过同一个 hash 算法映射到 hash 数值空间中了，接下来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，直到遇见一个 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 必然是唯一和确定的。这样不就找到了对象和 cache 的映射方法了吗？！

依然继续上面的例子（参见图 3 ），那么根据上面的方法，对象 object1 将被存储到 cache A 上； object2和 object3 对应到 cache C ； object4 对应到 cache B ；

考察cache 的变动

前面讲过，通过 hash 然后求余的方法带来的最大问题就在于不能满足单调性，当 cache 有所变动时，cache 会失效，进而对后台服务器造成巨大的冲击，现在就来分析分析 consistent hashing 算法。

移除 cache

考虑假设 cache B 挂掉了，根据上面讲到的映射方法，这时受影响的将仅是那些沿 cache B 逆时针遍历直到下一个 cache （ cache C ）之间的对象，也即是本来映射到 cache B 上的那些对象。

因此这里仅需要变动对象 object4 ，将其重新映射到 cache C 上即可；参见图 4 。

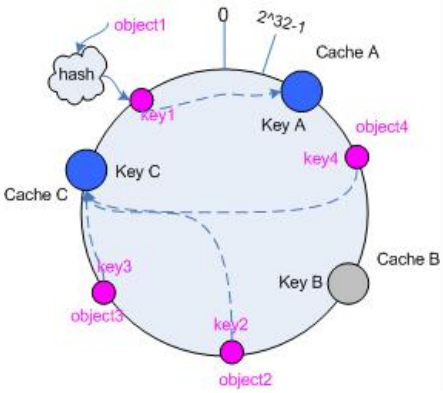

图 4 Cache B 被移除后的 cache 映射

添加 cache

再考虑添加一台新的 cache D 的情况，假设在这个环形 hash 空间中， cache D 被映射在对象 object2 和object3 之间。这时受影响的将仅是那些沿 cache D 逆时针遍历直到下一个 cache （ cache B ）之间的对象（它们是也本来映射到 cache C 上对象的一部分），将这些对象重新映射到 cache D 上即可。

因此这里仅需要变动对象 object2 ，将其重新映射到 cache D 上；参见图 5 。

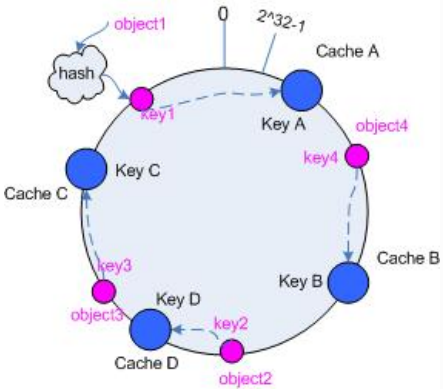

图 5 添加 cache D 后的映射关系

虚拟节点

考量 Hash 算法的另一个指标是平衡性 (Balance)，定义如下：

平衡性

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。

hash 算法并不是保证绝对的平衡，如果 cache 较少的话，对象并不能被均匀的映射到 cache 上，比如在上面的例子中，仅部署 cache A 和 cache C 的情况下，在 4 个对象中， cache A 仅存储了 object1 ，而 cache C 则存储了 object2 、object3 和 object4 ；分布是很不均衡的。

为了解决这种情况， consistent hashing 引入了“虚拟节点”的概念，它可以如下定义：

“虚拟节点”（ virtual node ）是实际节点在 hash 空间的复制品（ replica ），一实际个节点对应了若干个“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。

仍以仅部署 cache A 和 cache C 的情况为例，在图 4 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 ，这就意味着一共会存在 4 个“虚拟节点”， cache A1, cache A2 代表了 cache A ； cache C1, cache C2 代表了 cache C ；假设一种比较理想的情况，参见图 6 。

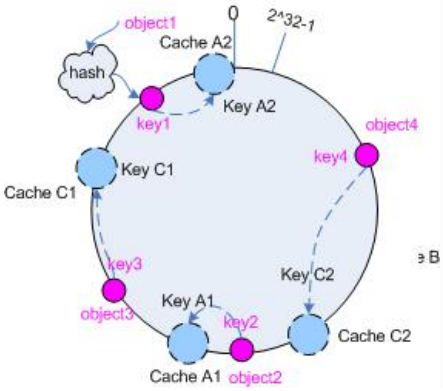

图 6 引入“虚拟节点”后的映射关系

此时，对象到“虚拟节点”的映射关系为：

objec1->cache A2 ； objec2->cache A1 ； objec3->cache C1 ； objec4->cache C2 ；

因此对象 object1 和 object2 都被映射到了 cache A 上，而 object3 和 object4 映射到了 cache C 上；平衡性有了很大提高。

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 } 。查询物体所在 cache时的映射关系如图 7 所示。

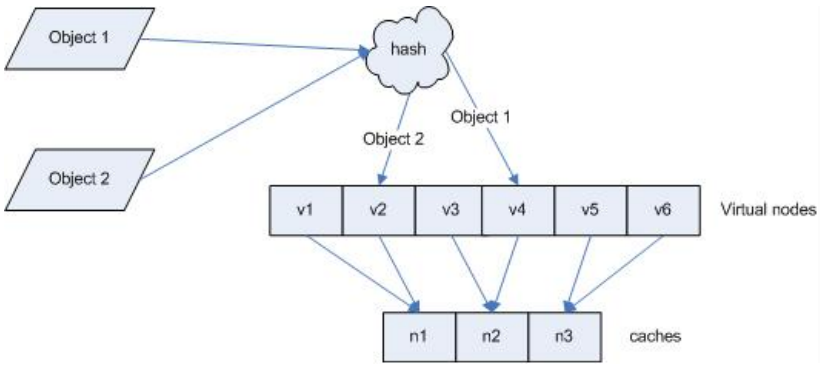

图 7 查询对象所在 cache

“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。例如假设 cache A 的 IP 地址为202.168.14.241。

引入“虚拟节点”前，计算 cache A 的 hash 值：

Hash(“202.168.14.241”);

引入“虚拟节点”后，计算“虚拟节”点 cache A1 和 cache A2 的 hash 值：

Hash(“202.168.14.241#1”); // cache A1

Hash(“202.168.14.241#2”); // cache A2