**简介：**

一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，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，使得分布式哈希（DHT）可以在P2P环境中真正得到应用。

**场景引入：**

比如你有 N 个 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，你很可能会采用类似下面的通用方法计算 object 的 hash 值，然后均匀的映射到到 N 个 cache ：

hash(object)%N

* 1

上面取模的方法一般称为简单 hash 算法 。通过简单 hash 算法确实能够比较均匀地实现分布式布置（映射），但是我们来考虑下面两种情形：

1） 某个 cache 服务器 m down 掉了（在实际应用中必须要考虑这种情况），这样所有映射到 cache m 的对象都会失效，怎么办，需要把 cache m 从 cache 中移除，这时候 cache 是 N-1 台，映射公式变成了 hash(object)%(N-1) ；

2） 由于访问加重，需要添加 cache ，这时候 cache 是 N+1 台，映射公式变成了 hash(object)%(N+1) ；

1） 和 2） 意味着什么？这意味着无论是添加或移除 cache 服务器，突然之间几乎所有的 cache 都失效了。对于服务器而言，这是一场灾难，洪水般的访问都会直接冲向后台服务器；

为了解决上面的问题，我们引入一致性 Hash 算法（consistent hashing）。

**hash 算法和单调性**

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。

简单的说，单调性要求在移除 / 添加一个 cache（机器，ip）时，它能够尽可能小的改变已存在 key 映射关系。

容易看到，上面的简单 hash 算法 hash(object)%N 难以满足单调性要求。因为 N 的变化会使取模结果发生变化。

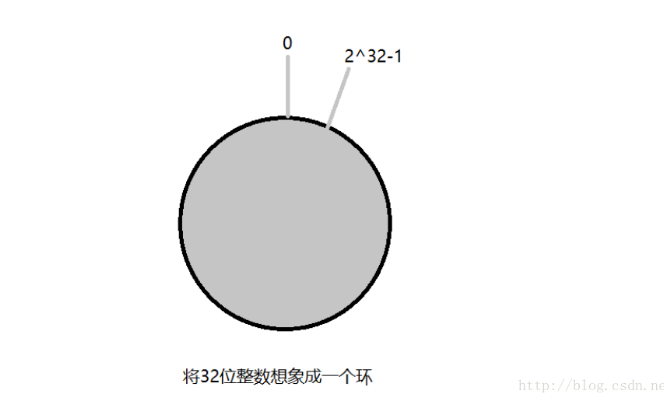
**一致性 Hash 算法原理：**

一致性 Hash 算法简单的说，在移除 / 添加一个 cache 时，它能够尽可能小的改变已存在 key 映射关系，尽可能的满足单调性的要求。

下面就来按照 6 个步骤简单讲讲一致性 Hash 算法的基本原理。

**步骤一：环形 hash 空间**

考虑通常的 hash 算法都是将 value 映射到一个 32 位的 key 值（然后取模），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。如下图所示：

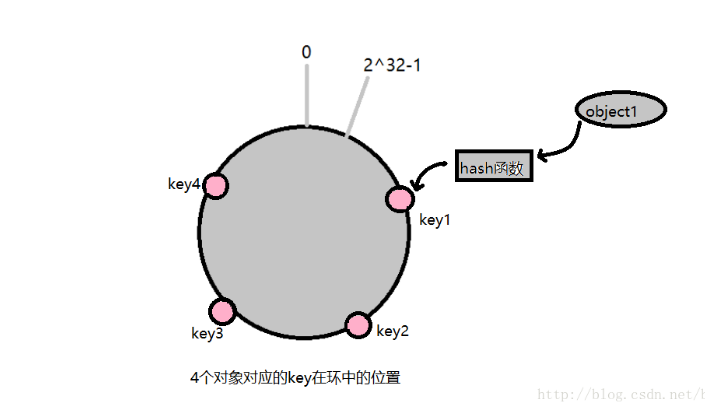


**步骤二：把对象处理成整数并映射到环形 hash 空间**

例如现在我们有四个对象 object1~object4，通过 hash 函数将四个对象处理成整数 key：

key1 = hash(object1);   
key2 = hash(object2);   
key3 = hash(object3);   
key4 = hash(object4);

然后将这些对象按照 key 的值映射到环形 hash 空间上：



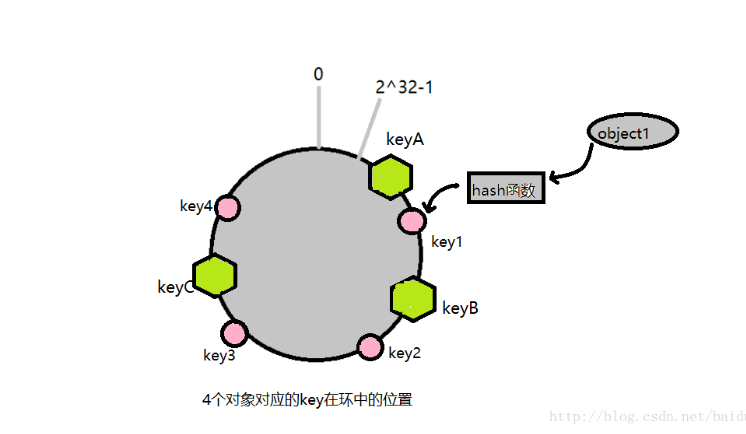
**步骤三：把 cache 映射到环形 hash 空间**

一致性 Hash 算法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。

假设现在有三台 cache 服务器：cacheA、cacheB、cacheC，通过 hash 函数处理获取对应的key值：

keyA = hash(cacheA);   
keyB = hash(cacheB);   
keyC = hash(cacheC);

将三个 cache 服务器按照 key 的值映射到环形 hash 空间上：

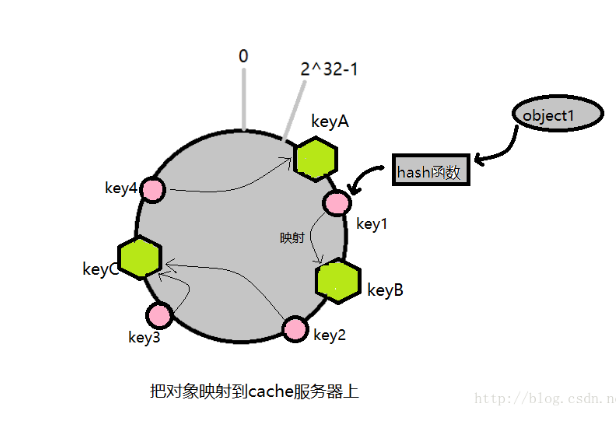


说到这里，顺便提一下 cache 的 hash 计算，一般的方法可以使用 cache 机器的 IP 地址或者机器名作为 hash 输入。

经过上面的步骤，我们把对象和cache 服务器都映射到同一个环形 hash 空间上。下面考虑的是如何将对象映射到 cache 服务器上。

**步骤四：将对象映射到 cache 服务器**

我们沿着圆环顺时针方向的对象 key （图中的 key1）出发，直到遇到一个cache服务器为止（cacheB），把对象key对应的对象映射到这个服务器上。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。按照这样的方法，可以得出：对象 object 1 映射到 cacheB，object2、object3 映射到 cacheC，object4 映射到 cacheA。如图：

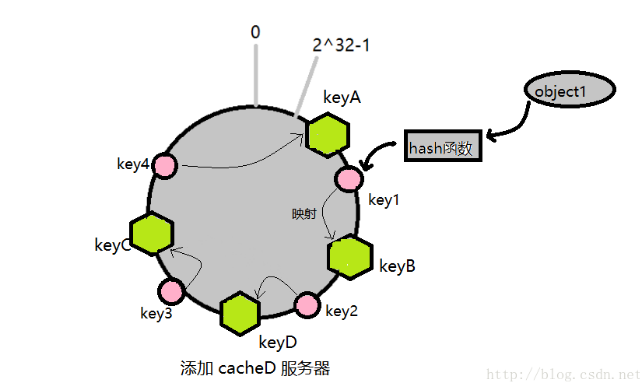


前面讲过，普通 hash 算法（通过 hash 然后求余的方法）带来的最大问题就在于不能满足单调性，当 cache 数量有所变动时（添加/移除）， 几乎所有的 cache 会失效，进而对后台服务器造成巨大的冲击，接下来分析一致 hash 算法。

**步骤五：添加 cache 服务器**

现在假如访问加重，需要增加 cacheD 服务器，经过 hash 函数计算（keyD = hash(cacheD)）发现数值介于 key3 和 key2 之间，即在圆环上的位置也是介于它们之间。这时候受到影响的是沿着 KeyD 逆时针出发直到遇到下一个 cache 服务器（keyB）之间的对象（这些对象原本是映射到 cacheC 上的），将这些对象重新映射到 cacheD 即可。

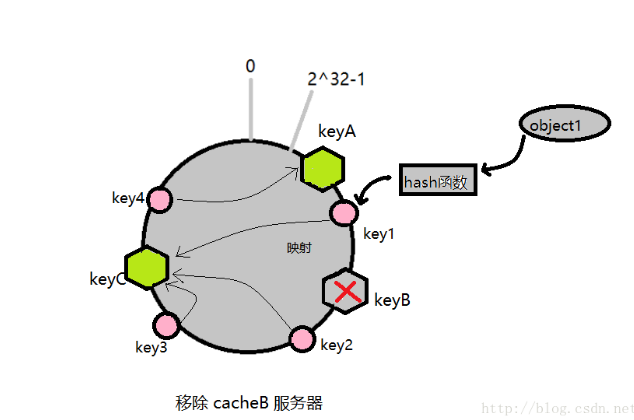
在我们的例子中仅仅是 object2（key2）需要变动，将其重新映射到 cacheD 即可：



**步骤六：移除 cache 服务器**

还是按照原来的图（步骤五之前）分析，假如现在 cacheB 服务器 down 掉了，需要把 cacheB 服务器移除，这时候受到影响的仅是那些沿着 keyB 逆时针出发知道遇到下一个服务器（cacheA）之间的对象，也就是本来映射到 cacheB 上的那些对象。

我们的例子中仅仅是 object1（key1）需要变动，将其重新映射到 cacheC 即可：



**平衡性与虚拟节点：**

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。

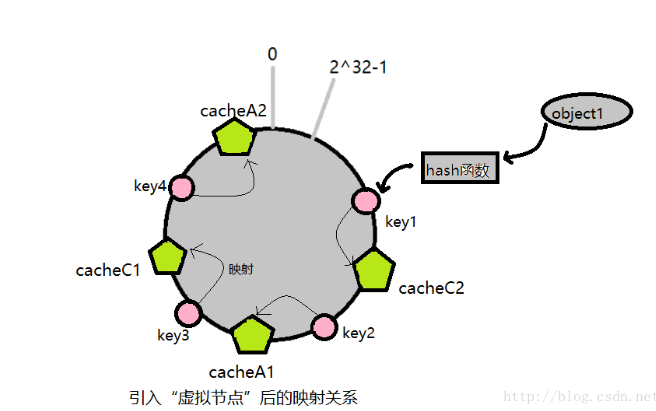
hash 算法并不是保证绝对的平衡，如果 cache 较少的话，对象并不能被均匀的映射到 cache 上，比如在上面的例子中，仅部署 cache A 和 cache C 的情况下，在 4 个对象中， cache A 仅存储了 object1 ，而 cache C 则存储了 object2 、 object3 和 object4 ；分布是很不均衡的。

为了解决这种情况， 一致性 Hash 算法引入了“虚拟节点”的概念，它可以如下定义：

**虚拟节点**

“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，在移除 cacheB 服务器图中我们已经看到， 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，如图：

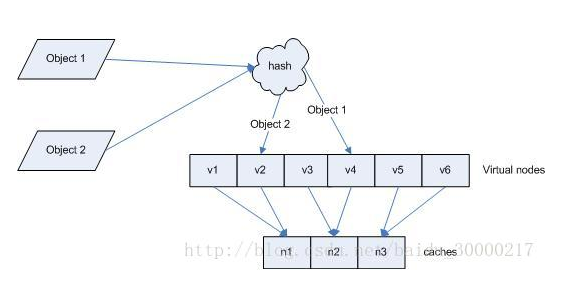


此时，对象到“虚拟节点”的映射关系为：

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

因此对象 object4 和 object2 都被映射到了 cache A 上，而 object3 和 object1 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 时的映射关系如图所示。



“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