数据结构与算法（16）：一致性哈希

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。

一、Hash算法

一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：

1. 平衡性(Balance)：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。很多哈希算法都能够满足这一条件。
2. 单调性(Monotonicity)：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原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。
3. 分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。
4. 负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

假设一个简单的场景：有4个cache服务器（后简称cache）组成的集群，当一个对象object传入集群时，这个对象应该存储在哪一个cache里呢？一种简单的方法是使用映射公式：

```
Hash(object) % 4
```

这个算法就可以保证任何object都会尽可能随机落在其中一个cache中。一切运行正常。

然后考虑以下情况：

由于流量增大，需要增加一台cache，共5个cache。这时，映射公式就变成Hash(object) % 5。
有一个cache服务器down掉，变成3个cache。这时，映射公式就变成Hash(object) % 3。

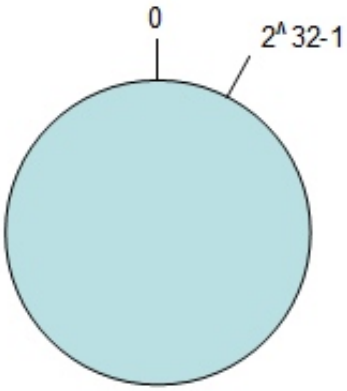
可见，无论新增还是减少节点，都会改变映射公式，而由于映射公式改变，几乎所有的object都会被映射到新的cache中，这意味着一时间所有的缓存全部失效。 大量的数据请求落在app层甚至是db层上，这样严重的违反了单调性原则,这对服务器的影响当然是灾难性的。

接下来主要讲解一下一致性哈希算法是如何设计的：

二、一致性Hash算法

2.1 环形Hash空间

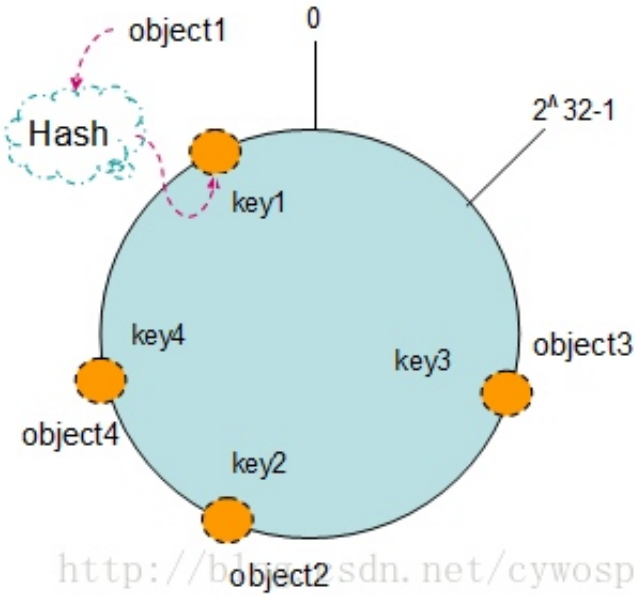
按照常用的hash算法来将对应的key哈希到一个具有2^{32}次方个桶的空间中，即$0至 $(2^{32})-1$的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形。如下图



2.2 数据映射

现在我们将object1、object2、object3、object4四个对象通过特定的Hash函数计算出对应的key值，然后散列到Hash环上。如下图：

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

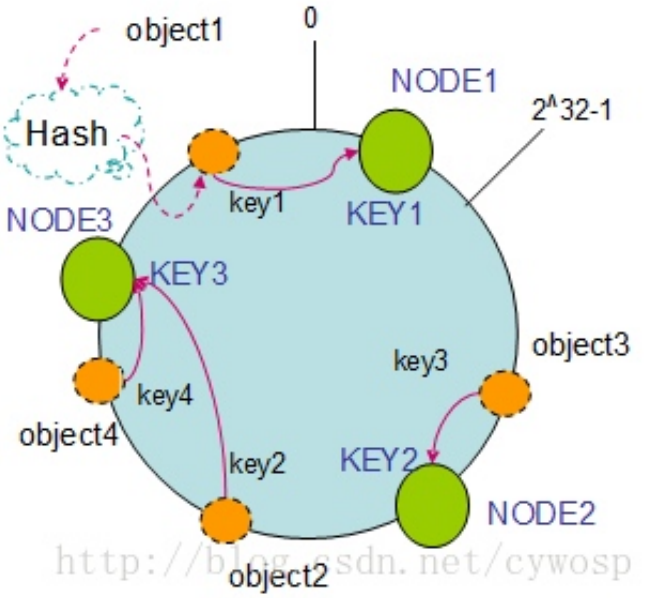


2.3 机器映射

在采用一致性哈希算法的分布式集群中将新的机器加入，其原理是通过使用与对象存储一样的Hash算法将机器也映射到环中（一般情况下对机器的hash计算是采用机器的IP或者机器唯一的别名作为输入值），然后以顺时针的方向计算，将所有对象存储到离自己最近的机器中。

假设现在有NODE1，NODE2，NODE3三台机器，通过Hash算法得到对应的KEY值，映射到环中，其示意图如下：

```
Hash(NODE1) = KEY1;
Hash(NODE2) = KEY2;
Hash(NODE3) = KEY3;
```



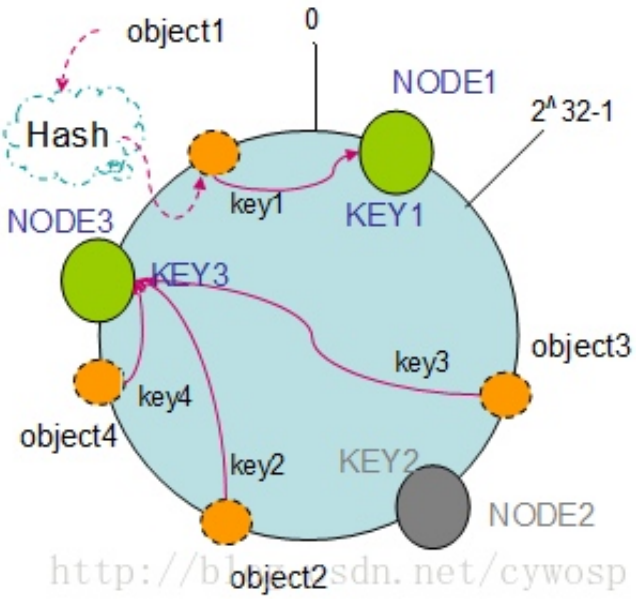
通过上图可以看出对象与机器处于同一哈希空间中，这样按顺时针转动object1存储到了NODE1中，object3存储到了NODE2中，object2、object4存储到了NODE3中。在这样的部署环境中，hash环是不会变更的，因此，通过算出对象的hash值就能快速的定位到对应的机器中，这样就能找到对象真正的存储位置了。

2.4 机器的删除与添加

普通hash求余算法最为不妥的地方就是在有机器的添加或者删除之后会照成大量的对象存储位置失效，这样就大大的不满足单调性了。下面来分析一下一致性哈希算法是如何处理的。

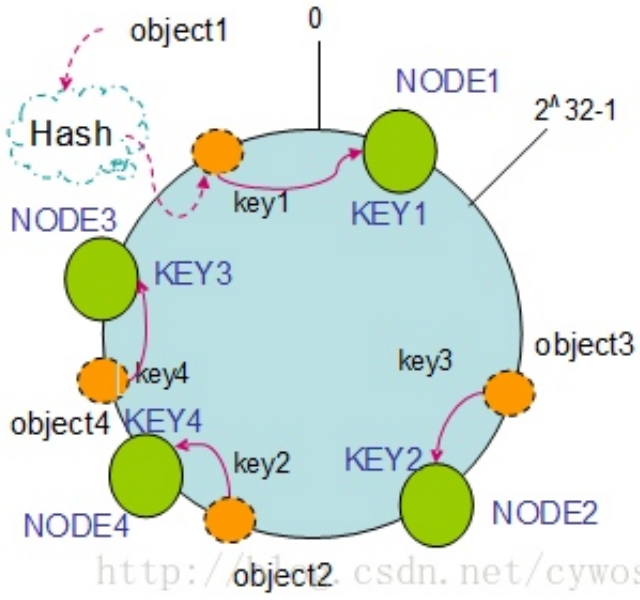
2.4.1 节点（机器）的删除

以上面的分布为例，如果NODE2出现故障被删除了，那么按照顺时针迁移的方法，object3将会被迁移到NODE3中，这样仅仅是object3的映射位置发生了变化，其它的对象没有任何的改动。如下图：



2.4.2 节点（机器）的添加

如果往集群中添加一个新的节点NODE4，通过对应的哈希算法得到KEY4，并映射到环中，如下图：



通过按顺时针迁移的规则，那么object2被迁移到了NODE4中，其它对象还保持这原有的存储位置。通过对节点的添加和删除的分析，一致性哈希算法在保持了单调性的同时，还是数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的的压力。

2.5 平衡性

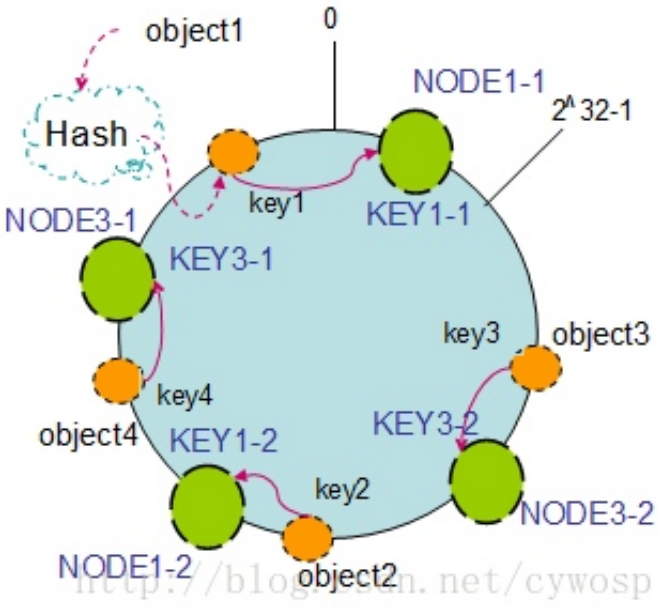
根据上面的图解分析，一致性哈希算法满足了单调性和负载均衡的特性以及一般hash算法的分散性，但这还并不能当做其被广泛应用的原由，因为还缺少了平衡性。下面将分析一致性哈希算法是如何满足平衡性的。hash算法是不保证平衡的，如上面只部署了NODE1和NODE3的情况（NODE2被删除的图），object1存储到了NODE1中，而object2、object3、object4都存储到了NODE3中，这样就照成了非常不平衡的状态。

2.6 虚拟节点

其实，理论上，只要cache足够多，每个cache在圆环上就会足够分散。但是在真实场景里，cache服务器只会有很少，所以，在一致性哈希算法中，为了尽可能的满足平衡性，其引入了虚拟节点的概念。

“虚拟节点”（ virtual node ）是实际节点（机器）在 hash 空间的复制品（ replica ），一实际个节点（机器）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。

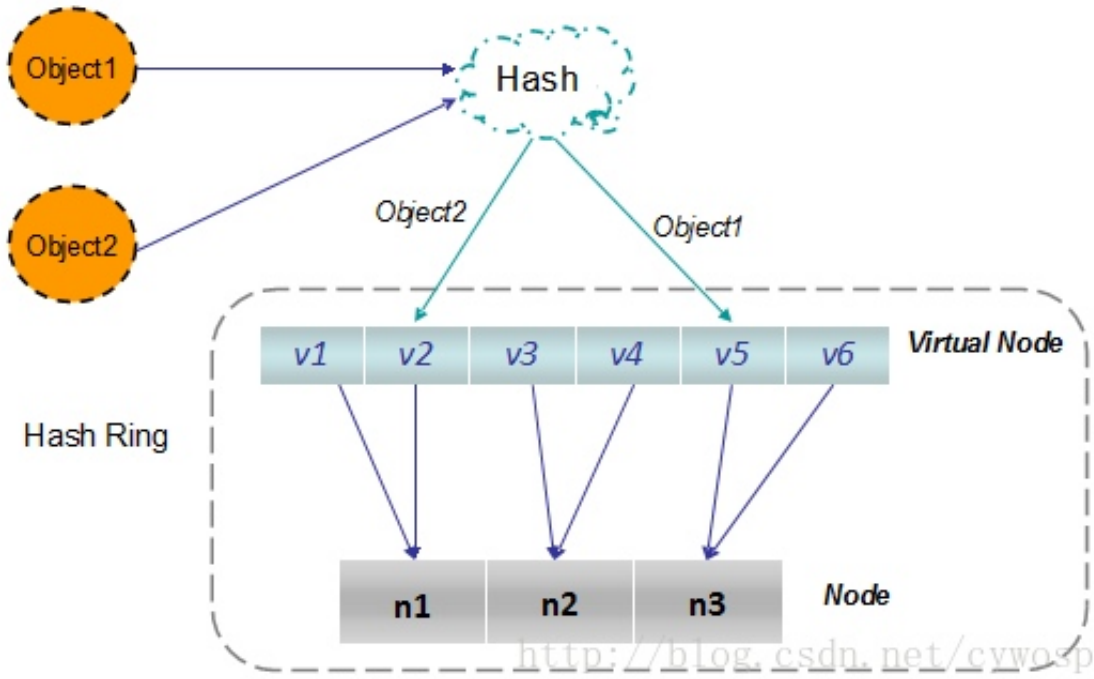
以上面只部署了NODE1和NODE3的情况（NODE2被删除的图）为例，之前的对象在机器上的分布很不均衡，现在我们以2个副本（复制个数）为例，这样整个hash环中就存在了4个虚拟节点，最后对象映射的关系图如下：



根据上图可知对象的映射关系：

```
object1->NODE1-1, object2->NODE1-2, object3->NODE3-2, object4->NODE3-1
```

通过虚拟节点的引入，对象的分布就比较均衡了。那么在实际操作中，正真的对象查询是如何工作的呢？对象从hash到虚拟节点到实际节点的转换如下图：



“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NODE1的IP地址为192.168.1.100。引入“虚拟节点”前，计算 cache A 的 hash 值：

```
Hash(“192.168.1.100”);
```

引入“虚拟节点”后，计算“虚拟节”点NODE1-1和NODE1-2的hash值：

```
Hash(“192.168.1.100#1”); // NODE1-1
Hash(“192.168.1.100#2”); // NODE1-2
```