**一致性hash算法简介**

Posted on [June 10, 2011](http://blog.huanghao.me/?p=14) by [huanghao](http://blog.huanghao.me/?author=1)

[一致性哈希算法](http://portal.acm.org/citation.cfm?id=258660)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](http://baike.baidu.com/view/924039.htm)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。

普通的哈希算法（也称硬哈希）采用简单取模的方式，将机器进行散列，这在cache环境不变的情况下能取得让人满意的结果，但是当cache环境动态变化时，这种静态取模的方式显然就不满足单调性的要求（当增加或减少一台机子时，几乎所有的存储内容都要被重新散列到别的缓冲区中）。

一致性哈希算法有多种具体的实现，包括[Chord算法](http://en.wikipedia.org/wiki/Chord_(peer-to-peer))，[KAD算法](http://wenku.baidu.com/view/ee91580216fc700abb68fcae.html)等实现，以上的算法的实现都比较复杂，这里介绍一种网上广为流传的一致性哈希算法的基本实现原理，感兴趣的同学可以根据上面的链接或者去网上查询更详细的资料。

一致性哈希算法的基本实现原理是将机器节点和key值都按照一样的hash算法映射到一个0~2^32的圆环上。当有一个写入缓存的请求到来时，计算Key值k对应的哈希值Hash(k)，如果该值正好对应之前某个机器节点的Hash值，则直接写入该机器节点，如果没有对应的机器节点，则顺时针查找下一个节点，进行写入，如果超过2^32还没找到对应节点，则从0开始查找(因为是环状结构)。如图1所示

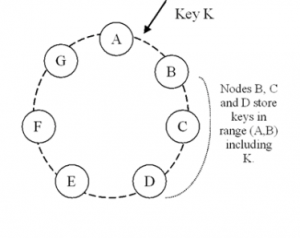
﻿[](http://blog.huanghao.me/wp-content/uploads/2011/06/%E5%9B%BE1.png)

图 1

图1中Key K的哈希值在A与B之间，于是K就由节点B来处理。

另外具体机器映射时，还可以根据处理能力不同，将一个实体节点映射到多个虚拟节点。

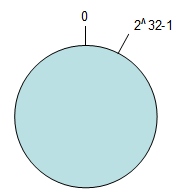
经过一致性哈希算法散列之后，当有新的机器加入时，将只影响一台机器的存储情况，例如新加入的节点H的散列在B与C之间，则原先由C处理的一些数据可能将移至H处理，而其他所有节点的处理情况都将保持不变，因此表现出很好的单调性。而如果删除一台机器，例如删除C节点，此时原来由C处理的数据将移至D节点，而其它节点的处理情况仍然不变。而由于在机器节点散列和缓冲内容散列时都采用了同一种散列算法，因此也很好得降低了分散性和负载。而通过引入虚拟节点的方式，也大大提高了平衡性。

最后附一个c语言一致性hash的[代码实现](http://www.codeproject.com/KB/recipes/lib-conhash.aspx)。

 在分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的hash(object)%N算法，那么在有机器添加或者删除后，很多原有的数据就无法找到了，这样严重的违反了单调性原则。接下来主要讲解一下一致性哈希算法是如何设计的：

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~(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



把数据通过一定的hash算法处理后映射到环上

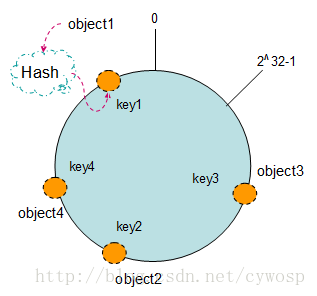
现在我们将object1、object2、object3、object4四个对象通过特定的Hash函数计算出对应的key值，然后散列到Hash环上。如下图：

    Hash(object1) = key1；

    Hash(object2) = key2；

    Hash(object3) = key3；

    Hash(object4) = key4；



将机器通过hash算法映射到环上

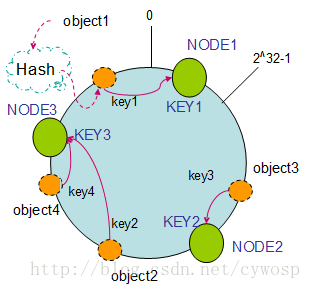
在采用一致性哈希算法的分布式集群中将新的机器加入，其原理是通过使用与对象存储一样的Hash算法将机器也映射到环中（一般情况下对机器的hash计算是采用机器的IP或者机器唯一的别名作为输入值），然后以顺时针的方向计算，将所有对象存储到离自己最近的机器中。

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;



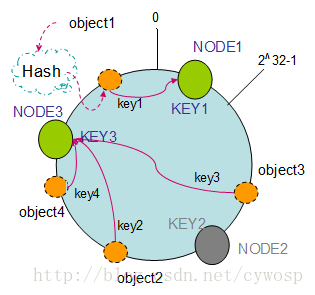
通过上图可以看出对象与机器处于同一哈希空间中，这样按顺时针转动object1存储到了NODE1中，object3存储到了NODE2中，object2、object4存储到了NODE3中。在这样的部署环境中，hash环是不会变更的，因此，通过算出对象的hash值就能快速的定位到对应的机器中，这样就能找到对象真正的存储位置了。

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。

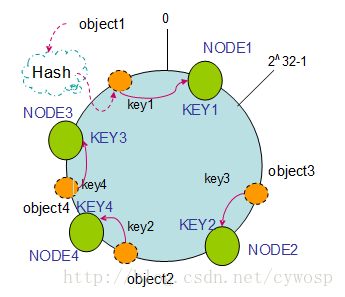
1. 节点（机器）的删除

    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. 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：



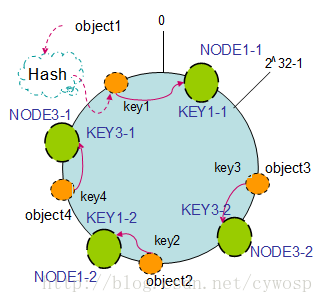
    通过按顺时针迁移的规则，那么object2被迁移到了NODE4中，其它对象还保持这原有的存储位置。通过对节点的添加和删除的分析，一致性哈希算法在保持了单调性的同时，还是数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的的压力。

平衡性

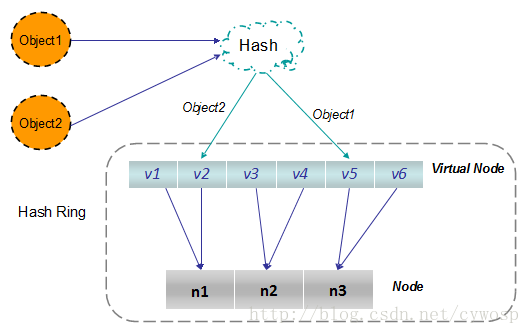
根据上面的图解分析，一致性哈希算法满足了单调性和负载均衡的特性以及一般hash算法的分散性，但这还并不能当做其被广泛应用的原由，因为还缺少了平衡性。下面将分析一致性哈希算法是如何满足平衡性的。hash算法是不保证平衡的，如上面只部署了NODE1和NODE3的情况（NODE2被删除的图），object1存储到了NODE1中，而object2、object3、object4都存储到了NODE3中，这样就照成了非常不平衡的状态。在一致性哈希算法中，为了尽可能的满足平衡性，其引入了虚拟节点。

    ——“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

**tencent2012 笔试题附加题**

     问题描述：  例如手机朋友网有n个服务器，为了方便用户的访问会在服务器上缓存数据，因此用户每次访问的时候最好能保持同一台服务器。已有的做法是根据ServerIPIndex[QQNUM%n]得到请求的服务器，这种方法很方便将用户分到不同的服务器上去。但是如果一台服务器死掉了，那么n就变为了n-1，那么ServerIPIndex[QQNUM%n]与ServerIPIndex[QQNUM%（n-1）]基本上都不一样了，所以大多数用户的请求都会转到其他服务器，这样会发生大量访问错误。

 问：  如何改进或者换一种方法，使得：

（1）一台服务器死掉后，不会造成大面积的访问错误，

（2）原有的访问基本还是停留在同一台服务器上；

（3）尽量考虑负载均衡。

解决思路，采用一致性哈希方法可以解决此问题：见下文：

## ****转自****<http://blog.csdn.net/sparkliang/article/details/5279393>

# 一致性 hash 算法（ consistent hashing ）

张亮

consistent hashing 算法早在 1997 年就在论文 [Consistent hashing and random trees](http://portal.acm.org/citation.cfm?id=258660)中被提出，目前在 cache 系统中应用越来越广泛；

## 1 基本场景

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

一切都运行正常，再考虑如下的两种情况；

1 一个 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；

再来考虑第三个问题，由于硬件能力越来越强，你可能想让后面添加的节点多做点活，显然上面的 hash 算法也做不到。

  有什么方法可以改变这个状况呢，这就是 consistent hashing...

## 2 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。

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

## 3 consistent hashing 算法的原理

consistent hashing 是一种 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 5 个步骤简单讲讲 consistent hashing 算法的基本原理。

### 3.1 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。

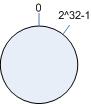


图 1 环形 hash 空间

### 3.2 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;

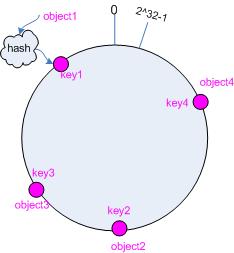


图 2 4 个对象的 key 值分布

### 3.3 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;

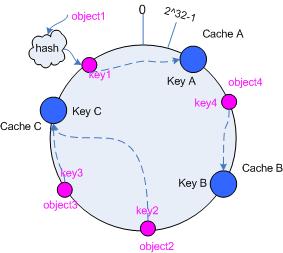


图 3 cache 和对象的 key 值分布

**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。**

### 3.4 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 ；

### 3.5 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。

**3.5.1 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 A ）（原文笔误写为cacheC）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

**3.5.2 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 。

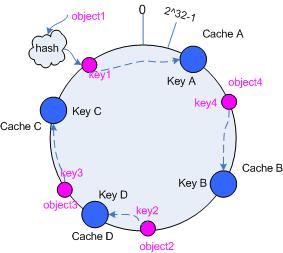


图 5 添加 cache D 后的映射关系

## 4 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 。

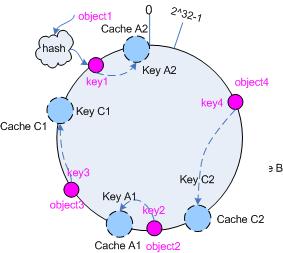


图 6 引入“虚拟节点”后的映射关系

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。

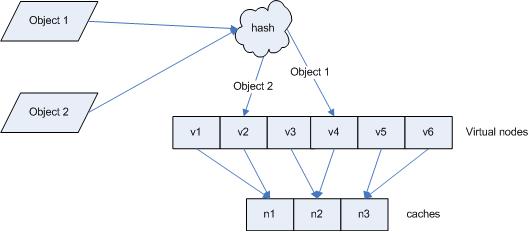


图 7 查询对象所在 cache

**“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

问题：1.请问虚拟节点，怎么保证均匀分布在，那个环上呢？

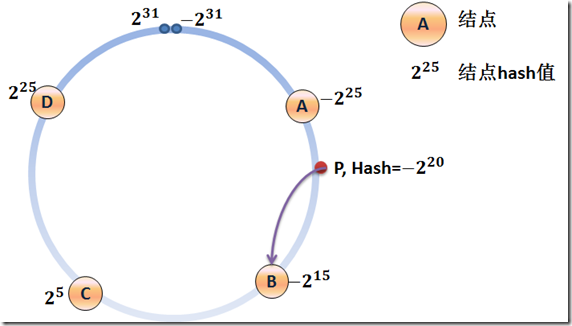
这就要由hash算法来保证了，均匀分布是概率上的均匀，当虚拟节点足够时，就能保证大概均匀了。

         2.假如cache通过hash函数计算出的值和 object通过hash函数计算出来的值是同一个hash值怎么办？  
  
那object应该指向哪个cache?

**C++实现方法：转自：**<http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html>

**一致性hash算法实现有两个关键问题需要解决，一个是用于结点存储和查找的数据结构的选择，另一个是结点hash算法的选择。**

    首先来谈一下一致性hash算法中用于存储结点的数据结构。通过了解一致性hash的原理，我们知道结点可以想象为是存储在一个环形的数据结构上（如下图），结点A、B、C、D按hash值在环形分布上是有序的，也就是说结点可以按hash值存储在一个有序的队列里。如下图所示，当一个hash值为-2^20的请求点P查找路由结点时，一致性hash算法会按hash值的顺时针方向路由到第一个结点上（B），也就是相当于要在存储结点的有序结构中，**按查询的key值找到大于key值中的最小的那个结点。因此，我们应该选择一种数据结构，它应该高效地支持结点频繁地增删，也必须具有理想的查询效率。那么，红黑树可以满足这些要求。**红黑树是一颗近似平衡的一颗二叉查找树，因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。**因此，我们选择使用红黑树作为结点的存储结构**，除了需要实现红黑树基本的插入、删除、查找的基本功能，**我们还应该增加另一个查询lookup函数，用于查找大于key中最小的结点**。

[](http://images.cnblogs.com/cnblogs_com/coser/201111/201111271652114531.png)

   接下来，我们来说hash算法的选择。一致性hash算法最初提出来，就是为了解决负载均衡的问题。每个实体结点会包含很多虚拟结点，**虚拟结点是平衡负载的关键。**我们希望虚拟结点可以均衡的散列在整个“环”上，这样不仅可以负载到不同hash值的路由请求，还可以当某个结点down掉，原来路由到down掉结点的请求也可以较均衡的路由到其他结点而不会对某个结点造成大量的负载请求。这里，**我们选择使用MD5算法。通过MD5算法，可以将一个标示串（用于标示虚拟结点）转化得到一个16字节的字符数组，再对该数组进行处理，得到一个整形的hash值。由于MD5具有高度的离散性，所以生成的hash值也会具有很大的离散性，会均衡的散列到“环”上。**

   笔者用C++语言对一致性hash算法进行了实现，下面我将会描述下一些关键细节。

1、首先定义实体结点类、虚拟结点类。一个实体结点对应多个虚拟结点。

  实体结点 CNode\_s：

[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*实体结点\*/  class CNode\_s  {  public:      /\*构造函数\*/      CNode\_s();      CNode\_s(char \* pIden , int pVNodeCount , void \* pData);        /\*获取结点标示\*/      const char \* getIden();        /\*获取实体结点的虚拟结点数量\*/      int getVNodeCount();        /\*设置实体结点数据值\*/      void setData(void \* data);        /\*获取实体结点数据值\*/      void \* getData();  private:      void setCNode\_s(char \* pIden, int pVNodeCount , void \* pData);      char iden[100];/\*结点标示串\*/      int vNodeCount;/\*虚拟结点数目\*/      void \* data;/\*数据结点\*/  }; |

   虚拟结点 CVirtualNode\_s：虚拟结点有一指针指向实体结点

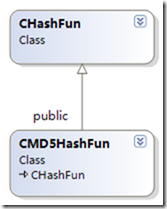
[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | /\*虚拟结点\*/  class CVirtualNode\_s  {  public:      /\*构造函数\*/      CVirtualNode\_s();      CVirtualNode\_s(CNode\_s \* pNode);        /\*设置虚拟结点所指向的实体结点\*/      void setNode\_s(CNode\_s \* pNode);        /\*获取虚拟结点所指向的实体结点\*/      CNode\_s \* getNode\_s();        /\*设置虚拟结点hash值\*/      void setHash(long pHash);        /\*获取虚拟结点hash值\*/      long getHash();  private:      long hash;/\*hash值\*/      CNode\_s \* node; /\*虚拟结点所指向的实体结点\*/  }; |

2、hash算法具有可选择性，定义一个hash算法接口，方便以后进行其他算法的扩展。

      这里创建MD5hash类，并继承该接口，通过MD5算法求hash值。

类图：

[](http://images.cnblogs.com/cnblogs_com/coser/201111/201111271652127530.png)

CHashFun接口：

[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | /\*定义Hash函数类接口，用于计算结点的hash值\*/    class CHashFun  {  public:      virtual long getHashVal(const char \*) = 0;  }; |

CMD5HashFun 类继承CHashFun接口，实现获取hash值的getHashVal函数：

[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | /\*用MD5算法计算结点的hash值，继承CHashFun父类\*/  class CMD5HashFun : public CHashFun  {  public:      virtual long getHashVal (const char \* );  };    long CMD5HashFun::getHashVal(const char \* instr)  {      int i;      long hash = 0;      unsignedchar digest[16];        /\*调用MD5相关函数，生成instr的MD5码，存入digest\*/      md5\_state\_t md5state;      md5\_init(&md5state);      md5\_append(&md5state, (const unsignedchar \*)instr,strlen(instr));      md5\_finish(&md5state, digest);        /\* 每四个字节构成一个32位整数，      将四个32位整数相加得到instr的hash值（可能溢出） \*/      for(i = 0; i < 4; i++)      {          hash += ((long)(digest[i\*4 + 3]&0xFF) << 24)              | ((long)(digest[i\*4 + 2]&0xFF) << 16)              | ((long)(digest[i\*4 + 1]&0xFF) <<  8)              | ((long)(digest[i\*4 + 0]&0xFF));      }      return hash;  } |

3、扩展红黑树结构中的查找函数，用于查找红黑树中大于key值中最小的结点。

[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | util\_rbtree\_node\_t\* util\_rbtree\_lookup(util\_rbtree\_t \*rbtree, long key)  {      if((rbtree != NULL) && !util\_rbtree\_isempty(rbtree))      {          util\_rbtree\_node\_t \*node = NULL;          util\_rbtree\_node\_t \*temp = rbtree->root;          util\_rbtree\_node\_t \*null = \_NULL(rbtree);          while(temp != null)          {              if(key <= temp->key)              {                  node = temp; /\* update node \*/                  temp = temp->left;              }              else if(key > temp->key)              {                  temp = temp->right;              }          }          /\* if node==NULL return the minimum node \*/          return ((node != NULL) ? node : util\_rbtree\_min(rbtree));      }      return NULL;  } |

4、创建一致性hash类。使其具有插入、删除、查找实体结点的功能。

具体算法和操作过程已经在代码注释中说明。

[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139 | class CConHash  {  public:      /\*构造函数\*/      CConHash(CHashFun \* pFunc);        /\*设置hash函数\*/      void setFunc(CHashFun \* pFunc);        /\*增加实体结点 , 0代表成功 , -1代表失败\*/      int addNode\_s(CNode\_s \* pNode);        /\*删除实体结点 , 0代表成功 , -1代表失败\*/      int delNode\_s(CNode\_s \* pNode);        /\*查找实体结点\*/      CNode\_s \* lookupNode\_s(const char \* object);        /\*获取一致性hash结构的所有虚拟结点数量\*/      int getVNodes();  private:      /\*Hash函数\*/      CHashFun \* func;      /\*虚拟结点总个数\*/      int vNodes;      /\*存储虚拟结点的红黑树\*/      util\_rbtree\_t \* vnode\_tree;  };  /\*辅助函数，虚拟结点转化为红黑树结点\*/  util\_rbtree\_node\_t \* vNode2RBNode(CVirtualNode\_s \* vnode);      CConHash::CConHash(CHashFun \* pFunc)  {      /\*设置hash函数\*/      assert(pFunc!=NULL);      this->func = pFunc;      this->vNodes = 0;      /\*初始化红黑树\*/      vnode\_tree = new util\_rbtree\_s();      util\_rbtree\_init(vnode\_tree);  }    int CConHash::addNode\_s(CNode\_s \* pNode)  {      if(pNode==NULL)return -1;      int vCount = pNode->getVNodeCount();      if(vCount<=0)return -1;      CVirtualNode\_s \* virtualNode ;      util\_rbtree\_node\_t \* rbNode;      char str [100];      char num[10];      strcpy(str,pNode->getIden());      long hash = 0;      /\*生成虚拟结点并插入到红黑树中\*/      for(int i=0;i<vCount;i++)      {          virtualNode = new CVirtualNode\_s(pNode);          /\*采用str+“i”的方法产生不同的iden串，用于后面的hash值计算\*/          itoa(i,num,10);          strcat(str,num);          hash = func->getHashVal(str);          virtualNode->setHash(hash);          if(!util\_rbtree\_search(vnode\_tree,hash))          {              /\*生成红黑树结点\*/              rbNode = vNode2RBNode(virtualNode);              if(rbNode!=NULL)              {                  /\*将该结点插入到红黑树中\*/                  util\_rbtree\_insert(vnode\_tree,rbNode);                  this->vNodes++;              }          }      }      return 0;  }    int CConHash::delNode\_s(CNode\_s \* pNode)  {      if(pNode==NULL)return -1;      util\_rbtree\_node\_t \* rbNode;      char str [100];      char num [10];      strcpy(str,pNode->getIden());      int vCount = pNode->getVNodeCount();      long hash = 0;      CVirtualNode\_s \* node = NULL;      /\*将该实体结点产生的所有虚拟结点进行删除\*/      for(int i=0;i<vCount;i++)      {          itoa(i,num,10);          strcat(str,num);/\*采用该方法产生不同的iden串\*/          hash = func->getHashVal(str);          rbNode = util\_rbtree\_search(vnode\_tree,hash);          if(rbNode!=NULL)          {              node = (CVirtualNode\_s \*) rbNode->data;              if(node->getNode\_s()==pNode && node->getHash()==hash)              {                  this->vNodes--;                  /\*将该结点从红黑树中删除\*/                  util\_rbtree\_delete(vnode\_tree,rbNode);                  delete rbNode;                  delete node;              }          }      }      return 0;  }    CNode\_s \* CConHash::lookupNode\_s(const char \* object)  {      if(object==NULL||this->vNodes==0)return NULL;      util\_rbtree\_node\_t \* rbNode;      int key = this->func->getHashVal(object);      /\*在红黑树中查找key值比key大的最小的结点\*/      rbNode = util\_rbtree\_lookup(vnode\_tree,key);      if(rbNode!=NULL)      {          return ((CVirtualNode\_s \*) rbNode->data)->getNode\_s();      }      return NULL;  }    int CConHash::getVNodes()  {      return this->vNodes;  }      util\_rbtree\_node\_t \* vNode2RBNode(CVirtualNode\_s \* vnode)  {      if(vnode==NULL)return NULL;      util\_rbtree\_node\_t \*rbNode = new util\_rbtree\_node\_t();      rbNode->key = vnode->getHash();      rbNode->data = vnode;      return rbNode;  } |

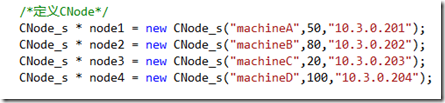
5、创建一个客户端类，对一致性hash算法进行测试。

      写了一个getIP的函数，模拟随机产生的IP字符串。

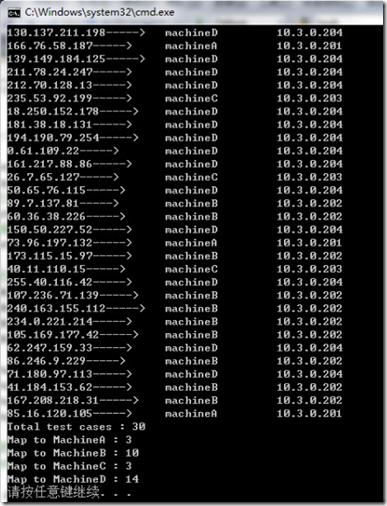
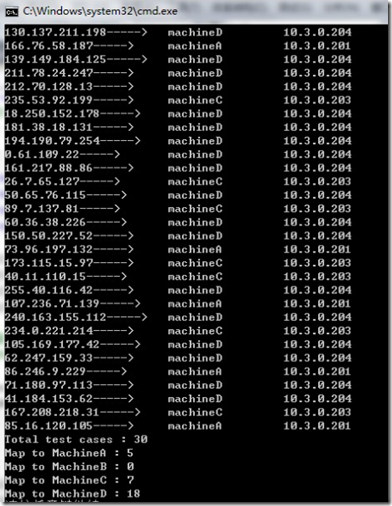
[?](http://www.cnblogs.com/coser/archive/2011/11/27/2265134.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | #include<iostream>  #include"CNode\_s.h"  #include"CVirtualNode\_s.h"  #include"CHashFun.h"  #include"CMD5HashFun.h"  #include"CConHash.h"  #include<string.h>  #include<time.h>    using namespace std;    void getIP(char \* IP)  {      int a=0, b=0 , c=0 , d=0;      a = rand()%256;      b = rand()%256;      c = rand()%256;      d = rand()%256;      char aa[4],bb[4],cc[4],dd[4];      itoa(a, aa, 10);      itoa(b, bb, 10);      itoa(c, cc, 10);      itoa(d, dd, 10);      strcpy(IP,aa);      strcat(IP,".");      strcat(IP,bb);      strcat(IP,".");      strcat(IP,cc);      strcat(IP,".");      strcat(IP,dd);  }    int main()  {      srand(time(0));      freopen("out.txt","r",stdin);      /\*定义hash函数\*/      CHashFun \* func = new CMD5HashFun();      /\*创建一致性hash对象\*/      CConHash \* conhash = new CConHash(func);        /\*定义CNode\*/      CNode\_s \* node1 = new CNode\_s("machineA",50,"10.3.0.201");      CNode\_s \* node2 = new CNode\_s("machineB",80,"10.3.0.202");      CNode\_s \* node3 = new CNode\_s("machineC",20,"10.3.0.203");      CNode\_s \* node4 = new CNode\_s("machineD",100,"10.3.0.204");        conhash->addNode\_s(node1);      conhash->addNode\_s(node2);      conhash->addNode\_s(node3);      conhash->addNode\_s(node4);        /\*动态更改结点数据值\*/  //  node1->setData("99999999");        int ans1 ,ans2 ,ans3 ,ans4;      ans1=ans2=ans3=ans4=0;        char object[100];      CNode\_s \* node ;      /\*动态删除结点\*/      //conhash->delNode\_s(node2);      for(int i =0;i<30;i++)      {      //  getIP(object);      //  cout<<object<<endl;          cin>>object;          node = conhash->lookupNode\_s(object);          if(node!=NULL)          {              cout<<object<<"----->\t"<<node->getIden()<<" \t "<<(char \*)node->getData()<<endl;              if(strcmp(node->getIden(),"machineA")==0) ans1++;              if(strcmp(node->getIden(),"machineB")==0) ans2++;              if(strcmp(node->getIden(),"machineC")==0) ans3++;              if(strcmp(node->getIden(),"machineD")==0) ans4++;          }      }        cout<<"Total test cases : "<<ans1+ans2+ans3+ans4<<endl;      cout<<"Map to MachineA : "<<ans1<<endl;      cout<<"Map to MachineB : "<<ans2<<endl;      cout<<"Map to MachineC : "<<ans3<<endl;      cout<<"Map to MachineD : "<<ans4<<endl;      fclose(stdin);      return 0;  } |

6、删除结点对hash路由的影响测试

[](http://images.cnblogs.com/cnblogs_com/coser/201111/201111271652121401.png)

测试结果截图：

[](http://images.cnblogs.com/cnblogs_com/coser/201111/201111271652137398.png)[](http://images.cnblogs.com/cnblogs_com/coser/201111/2011112716521457.png)

分析：上面两幅图，左边为原始四个实体结点的路由情况，后面为删除结点2（Node2）之后的路由情况。不难发现，MachineB down之后，原先的路由请求，较均衡地负载到了其他机器结点，而且对原先路由到其他结点的请求没有影响。比如139.149.184.125这个请求仍会路由到MachineD，并不会因为结点的减少而造成影响。但是，如果是增加实体结点，可能会造成增加前后路由情况不一致的现象，因为路由区间的更加狭小，但是不会有特别大的影响。 另一方面，可以发现实体结点的虚拟结点个数比例分配情况很大程度影响了结点的负载路由情况，比例大致与虚拟结点个数相一致。

总结：

   本文首先通过介绍实现一致性hash算法的关键算法和数据结构的选择分析，选择了红黑树作为虚拟结点的存储结构，以及MD5算法作为Hash函数用于计算结点的hash值。并使用C++语言，对一致性hash算法进行了实现，实现了一致性hash实体结点的增加、删除、查找等基本功能，并进行了测试分析。由于笔者水平有限，存在很多有待改进的地方，因此本文仅供大家参考、讨论学习。