负载均衡算法

在集群负载均衡时，Dubbo提供了4种均衡策略，如：Random LoadBalance(随机均衡算法)、；RoundRobin LoadBalance(权重轮循均衡算法)、LeastAction LoadBalance(最少活跃调用数均衡算法)、ConsistentHash LoadBalance(一致性Hash均衡算法)。缺省时为Random随机调用。这四种算法的原理简要介绍如下：

## 1、RoundRobin LoadBalance

Round-Robin既是轮询算法，是按照公约后的权重设置轮询比率，即权重轮询算法(Weighted Round-Robin) ，它是基于轮询算法改进而来的。这里之所以写RoundRobin是为了跟Dubbo中的内容保持一致。

轮询调度算法的原理是：每一次把来自用户的请求轮流分配给内部中的服务器。如：从1开始，一直到N(其中，N是内部服务器的总个数)，然后重新开始循环。

该算法的优点：

其简洁性，它无需记录当前所有连接的状态，所以它是一种无状态调度。

该算法的缺点：

轮询调度算法假设所有服务器的处理性能都相同，不关心每台服务器的当前连接数和响应速度。当请求服务间隔时间变化比较大时，轮询调度算法容易导致服务器间的负载不平衡。

所以此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。但是，在实际情况中，可能并不是这种情况。由于每台服务器的配置、安装的业务应用等不同，其处理能力会不一样。所以，我们根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。

**权重轮询调度算法流程**

假设有一组服务器S = {S0, S1, …, Sn-1}，W(Si)表示服务器Si的权值，一个指示变量i表示上一次选择的服务器，指示变量cw表示当前调度的权值，max(S)表示集合S中所有服务器的最大权值，gcd(S)表示集合S中所有服务器权值的最大公约数。变量i初始化为-1，cw初始化为零。其算法如下：

|  |
| --- |
| while (true) {    i = (i + 1) mod n;    if (i == 0) {       cw = cw - gcd(S);       if (cw <= 0) {         cw = max(S);         if (cw == 0)           return NULL;       }    }    if (W(Si) >= cw)      return Si;  } |

这种算法的逻辑实现如图2所示，图中我们假定四台服务器的处理能力为3:1:1:1。

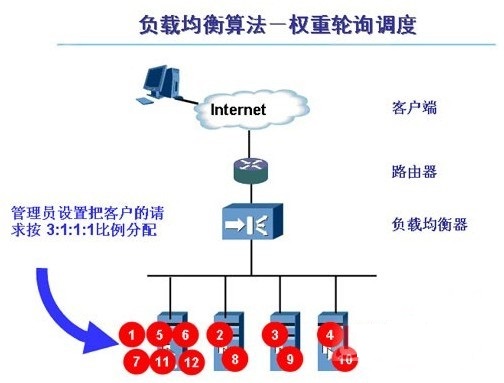


                                      图1 权重轮询调度实现逻辑图示

由于权重轮询调度算法考虑到了不同服务器的处理能力，所以这种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。所以，在实际应用中比较常见。

## 2、ConsistentHash LoadBalance

一致性Hash，相同参数的请求总是发到同一个提供者。一：一致性Hash算法可以解决服务提供者的增加、移除及挂掉时的情况，能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。二：一致性Hash通过构建虚拟节点，能尽可能避免分配失衡，具有很好的平衡性。

一致性Hash下面就来按照 5 个步骤简单讲讲 consistent hash算法的基本原理。因为以下资料来自于互联网，现说明几点：一、下面例子中的对象就相当于Client发的请求，cache相当于服务提供者。

**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 2 所示的那样。

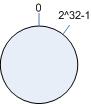


图 2 环形 hash 空间

**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 3所示。

hash(object1) = key1;

… …

hash(object4) = key4;

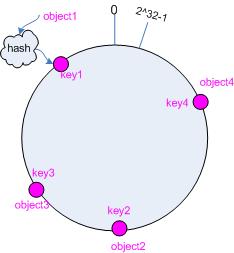


图 3 4个对象的 key 值分布

**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 4 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;

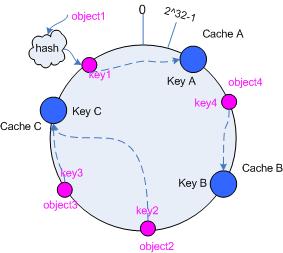


图 4 cache 和对象的 key 值分布

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 4 ），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，一致性Hash算法可以解决服务提供者的增加、移除及挂掉时的情况，能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。

**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 5 。

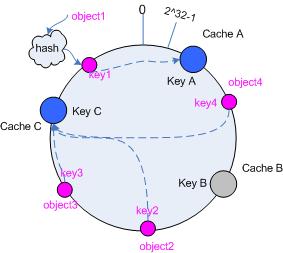


图 5 Cache B 被移除后的 cache 映射

**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 6 。

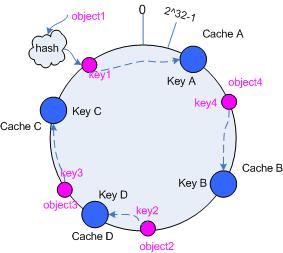


图 6 添加 cache D 后的映射关系

**虚拟节点**

考虑Hash 算法的另一个指标是平衡性 (Balance) ，定义如下：

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 5 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 7 。

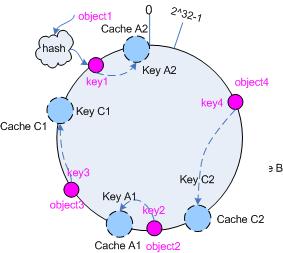


图 7 引入“虚拟节点”后的映射关系

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 8 所示。

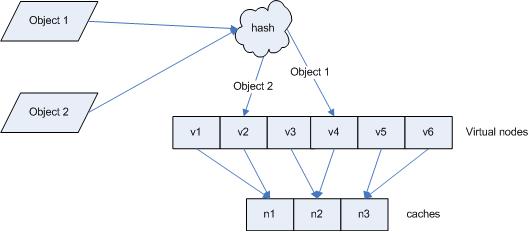


图 8 查询对象所在 cache

“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

## 3、Random LoadBalance与LeastAction LoadBalance

Random LoadBalance与LeastAction LoadBalance算法比较简单，可以参照Dubbo文档中的给的描述及后面代码附录。Dubbo文档截图如下图9所示：



图9 负载均衡算法

## 4、附录

### 1、RandomLoadBalance算法

**public** **class** RandomLoadBalance **extends** AbstractLoadBalance {

**public** **static** **final** String *NAME* = "random";

**private** **final** Random random = **new** Random();

**protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {

**int** length = invokers.size(); // 总个数

**int** totalWeight = 0; // 总权重

**boolean** sameWeight = **true**; // 权重是否都一样

**for** (**int** i = 0; i < length; i++) {

**int** weight = getWeight(invokers.get(i), invocation);

totalWeight += weight; // 累计总权重

**if** (sameWeight && i > 0

&& weight != getWeight(invokers.get(i - 1), invocation)) {

sameWeight = **false**; // 计算所有权重是否一样

}

}

**if** (totalWeight > 0 && ! sameWeight) {

// 如果权重不相同且权重大于0则按总权重数随机

**int** offset = random.nextInt(totalWeight);

// 并确定随机值落在哪个片断上

**for** (**int** i = 0; i < length; i++) {

offset -= getWeight(invokers.get(i), invocation);

**if** (offset < 0) {

**return** invokers.get(i);

}

}

}

// 如果权重相同或权重为0则均等随机

**return** invokers.get(random.nextInt(length));

}

}

### 2、RoundRobinLoadBalance算法

**public** **class** RoundRobinLoadBalance **extends** AbstractLoadBalance {

**public** **static** **final** String *NAME* = "roundrobin";

**private** **final** ConcurrentMap<String, AtomicPositiveInteger> sequences = **new** ConcurrentHashMap<String, AtomicPositiveInteger>();

**private** **final** ConcurrentMap<String, AtomicPositiveInteger> weightSequences = **new** ConcurrentHashMap<String, AtomicPositiveInteger>();

**protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {

String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();

**int** length = invokers.size(); // 总个数

**int** maxWeight = 0; // 最大权重

**int** minWeight = Integer.*MAX\_VALUE*; // 最小权重

**for** (**int** i = 0; i < length; i++) {

**int** weight = getWeight(invokers.get(i), invocation);

maxWeight = Math.*max*(maxWeight, weight); // 累计最大权重

minWeight = Math.*min*(minWeight, weight); // 累计最小权重

}

**if** (maxWeight > 0 && minWeight < maxWeight) { // 权重不一样

AtomicPositiveInteger weightSequence = weightSequences.get(key);

**if** (weightSequence == **null**) {

weightSequences.putIfAbsent(key, **new** AtomicPositiveInteger());

weightSequence = weightSequences.get(key);

}

**int** currentWeight = weightSequence.getAndIncrement() % maxWeight;

List<Invoker<T>> weightInvokers = **new** ArrayList<Invoker<T>>();

**for** (Invoker<T> invoker : invokers) { // 筛选权重大于当前权重基数的Invoker

**if** (getWeight(invoker, invocation) > currentWeight) {

weightInvokers.add(invoker);

}

}

**int** weightLength = weightInvokers.size();

**if** (weightLength == 1) {

**return** weightInvokers.get(0);

} **else** **if** (weightLength > 1) {

invokers = weightInvokers;

length = invokers.size();

}

}

AtomicPositiveInteger sequence = sequences.get(key);

**if** (sequence == **null**) {

sequences.putIfAbsent(key, **new** AtomicPositiveInteger());

sequence = sequences.get(key);

}

// 取模轮循

**return** invokers.get(sequence.getAndIncrement() % length);

}

}

### 3、LeastActionLoadBalance算法

**public** **class** LeastActiveLoadBalance **extends** AbstractLoadBalance {

**public** **static** **final** String *NAME* = "leastactive";

**private** **final** Random random = **new** Random();

**protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {

**int** length = invokers.size(); // 总个数

**int** leastActive = -1; // 最小的活跃数

**int** leastCount = 0; // 相同最小活跃数的个数

**int**[] leastIndexs = **new** **int**[length]; // 相同最小活跃数的下标

**int** totalWeight = 0; // 总权重

**int** firstWeight = 0; // 第一个权重，用于于计算是否相同

**boolean** sameWeight = **true**; // 是否所有权重相同

**for** (**int** i = 0; i < length; i++) {

Invoker<T> invoker = invokers.get(i);

**int** active = RpcStatus.*getStatus*(invoker.getUrl(), invocation.getMethodName()).getActive(); // 活跃数

**int** weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.*WEIGHT\_KEY*, Constants.*DEFAULT\_WEIGHT*); // 权重

**if** (leastActive == -1 || active < leastActive) { // 发现更小的活跃数，重新开始

leastActive = active; // 记录最小活跃数

leastCount = 1; // 重新统计相同最小活跃数的个数

leastIndexs[0] = i; // 重新记录最小活跃数下标

totalWeight = weight; // 重新累计总权重

firstWeight = weight; // 记录第一个权重

sameWeight = **true**; // 还原权重相同标识

} **else** **if** (active == leastActive) { // 累计相同最小的活跃数

leastIndexs[leastCount ++] = i; // 累计相同最小活跃数下标

totalWeight += weight; // 累计总权重

// 判断所有权重是否一样

**if** (sameWeight && i > 0

&& weight != firstWeight) {

sameWeight = **false**;

}

}

}

// assert(leastCount > 0)

**if** (leastCount == 1) {

// 如果只有一个最小则直接返回

**return** invokers.get(leastIndexs[0]);

}

**if** (! sameWeight && totalWeight > 0) {

// 如果权重不相同且权重大于0则按总权重数随机

**int** offsetWeight = random.nextInt(totalWeight);

// 并确定随机值落在哪个片断上

**for** (**int** i = 0; i < leastCount; i++) {

**int** leastIndex = leastIndexs[i];

offsetWeight -= getWeight(invokers.get(leastIndex), invocation);

**if** (offsetWeight <= 0)

**return** invokers.get(leastIndex);

}

}

// 如果权重相同或权重为0则均等随机

**return** invokers.get(leastIndexs[random.nextInt(leastCount)]);

}

}

### 4、ConsistentHashLoadBalance算法

**public** **class** ConsistentHashLoadBalance **extends** AbstractLoadBalance {

**private** **final** ConcurrentMap<String, ConsistentHashSelector<?>> selectors = **new** ConcurrentHashMap<String, ConsistentHashSelector<?>>();

@SuppressWarnings("unchecked")

@Override

**protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {

String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();

**int** identityHashCode = System.*identityHashCode*(invokers);

ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key);

**if** (selector == **null** || selector.getIdentityHashCode() != identityHashCode) {

selectors.put(key, **new** ConsistentHashSelector<T>(invokers, invocation.getMethodName(), identityHashCode));

selector = (ConsistentHashSelector<T>) selectors.get(key);

}

**return** selector.select(invocation);

}

**private** **static** **final** **class** ConsistentHashSelector<T> {

**private** **final** TreeMap<Long, Invoker<T>> virtualInvokers;

**private** **final** **int** replicaNumber;

**private** **final** **int** identityHashCode;

**private** **final** **int**[] argumentIndex;

**public** ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, **int** identityHashCode) {

**this**.virtualInvokers = **new** TreeMap<Long, Invoker<T>>();

**this**.identityHashCode = System.*identityHashCode*(invokers);

URL url = invokers.get(0).getUrl();

**this**.replicaNumber = url.getMethodParameter(methodName, "hash.nodes", 160);

String[] index = Constants.*COMMA\_SPLIT\_PATTERN*.split(url.getMethodParameter(methodName, "hash.arguments", "0"));

argumentIndex = **new** **int**[index.length];

**for** (**int** i = 0; i < index.length; i ++) {

argumentIndex[i] = Integer.*parseInt*(index[i]);

}

**for** (Invoker<T> invoker : invokers) {

**for** (**int** i = 0; i < replicaNumber / 4; i++) {

**byte**[] digest = md5(invoker.getUrl().toFullString() + i);

**for** (**int** h = 0; h < 4; h++) {

**long** m = hash(digest, h);

virtualInvokers.put(m, invoker);

}

}

}

}

**public** **int** getIdentityHashCode() {

**return** identityHashCode;

}

**public** Invoker<T> select(Invocation invocation) {

String key = toKey(invocation.getArguments());

**byte**[] digest = md5(key);

Invoker<T> invoker = sekectForKey(hash(digest, 0));

**return** invoker;

}

**private** String toKey(Object[] args) {

StringBuilder buf = **new** StringBuilder();

**for** (**int** i : argumentIndex) {

**if** (i >= 0 && i < args.length) {

buf.append(args[i]);

}

}

**return** buf.toString();

}

**private** Invoker<T> sekectForKey(**long** hash) {

Invoker<T> invoker;

Long key = hash;

**if** (!virtualInvokers.containsKey(key)) {

SortedMap<Long, Invoker<T>> tailMap = virtualInvokers.tailMap(key);

**if** (tailMap.isEmpty()) {

key = virtualInvokers.firstKey();

} **else** {

key = tailMap.firstKey();

}

}

invoker = virtualInvokers.get(key);

**return** invoker;

}

**private** **long** hash(**byte**[] digest, **int** number) {

**return** (((**long**) (digest[3 + number \* 4] & 0xFF) << 24)

| ((**long**) (digest[2 + number \* 4] & 0xFF) << 16)

| ((**long**) (digest[1 + number \* 4] & 0xFF) << 8)

| (digest[0 + number \* 4] & 0xFF))

& 0xFFFFFFFFL;

}

**private** **byte**[] md5(String value) {

MessageDigest md5;

**try** {

md5 = MessageDigest.*getInstance*("MD5");

} **catch** (NoSuchAlgorithmException e) {

**throw** **new** IllegalStateException(e.getMessage(), e);

}

md5.reset();

**byte**[] bytes = **null**;

**try** {

bytes = value.getBytes("UTF-8");

} **catch** (UnsupportedEncodingException e) {

**throw** **new** IllegalStateException(e.getMessage(), e);

}

md5.update(bytes);

**return** md5.digest();

}

}

}