参考：<http://www.cnblogs.com/xrq730/p/5154340.html>

**什么是负载均衡**

负载均衡，英文名称为Load Balance，指由多台服务器以对称的方式组成一个服务器集合，每台服务器都具有等价的地位，都可以单独对外提供服务而无须其他服务器的辅助。通过某种负载分担技术，将外部发送来的请求均匀分配到对称结构中的某一台服务器上，而接收到请求的服务器独立地回应客户的请求。负载均衡能够平均分配客户请求到服务器阵列，借此提供快速获取重要数据，解决大量并发访问服务问题，这种集群技术可以用最少的投资获得接近于大型主机的性能。

负载均衡分为软件负载均衡和硬件负载均衡，前者的代表是阿里章文嵩博士研发的LVS，后者则是均衡服务器比如F5，当然这只是提一下，不是重点。

本文讲述的是"**将外部发送来的请求均匀分配到对称结构中的某一台服务器上**"的各种算法，并以Java代码演示每种算法的具体实现，OK，下面进入正题，在进入正题前，先写一个类来模拟Ip列表：

1 public class IpMap

2 {

3 // 待路由的Ip列表，Key代表Ip，Value代表该Ip的权重

4 public static HashMap<String, Integer> serverWeightMap =

5 new HashMap<String, Integer>();

6

7 static

8 {

9 serverWeightMap.put("192.168.1.100", 1);

10 serverWeightMap.put("192.168.1.101", 1);

11 // 权重为4

12 serverWeightMap.put("192.168.1.102", 4);

13 serverWeightMap.put("192.168.1.103", 1);

14 serverWeightMap.put("192.168.1.104", 1);

15 // 权重为3

16 serverWeightMap.put("192.168.1.105", 3);

17 serverWeightMap.put("192.168.1.106", 1);

18 // 权重为2

19 serverWeightMap.put("192.168.1.107", 2);

20 serverWeightMap.put("192.168.1.108", 1);

21 serverWeightMap.put("192.168.1.109", 1);

22 serverWeightMap.put("192.168.1.110", 1);

23 }

24 }

**轮询（Round Robin）法**

轮询法即Round Robin法，其代码实现大致如下：

1 public class RoundRobin

2 {

3 private static Integer pos = 0;

4

5 public static String getServer()

6 {

7 // 重建一个Map，避免服务器的上下线导致的并发问题

8 Map<String, Integer> serverMap =

9 new HashMap<String, Integer>();

10 serverMap.putAll(IpMap.serverWeightMap);

11

12 // 取得Ip地址List

13 Set<String> keySet = serverMap.keySet();

14 ArrayList<String> keyList = new ArrayList<String>();

15 keyList.addAll(keySet);

16

17 String server = null;

18 synchronized (pos)

19 {

20 if (pos > keySet.size())

21 pos = 0;

22 server = keyList.get(pos);

23 pos ++;

24 }

25

26 return server;

27 }

28 }

由于serverWeightMap中的地址列表是动态的，随时可能有机器上线、下线或者宕机，因此为了避免可能出现的并发问题，方法内部要新建局部变量serverMap，先将serverMap中的内容复制到线程本地，以避免被多个线程修改。这样可能会引入新的问题，复制以后serverWeightMap的修改无法反映给serverMap，也就是说这一轮选择服务器的过程中，新增服务器或者下线服务器，负载均衡算法将无法获知。新增无所谓，如果有服务器下线或者宕机，那么可能会访问到不存在的地址。因此，**服务调用端需要有相应的容错处理，比如重新发起一次server选择并调用**。

对于当前轮询的位置变量pos，为了保证服务器选择的顺序性，需要在操作时对其加锁，**使得同一时刻只能有一个线程可以修改pos的值**，否则当pos变量被并发修改，则无法保证服务器选择的顺序性，甚至有可能导致keyList数组越界。

**轮询法的优点在于：试图做到请求转移的绝对均衡**。

**轮询法的缺点在于：为了做到请求转移的绝对均衡，必须付出相当大的代价，因为为了保证pos变量修改的互斥性，需要引入重量级的悲观锁synchronized，这将会导致该段轮询代码的并发吞吐量发生明显的下降**。

**随机（Random）法**

通过系统随机函数，根据后端服务器列表的大小值来随机选择其中一台进行访问。由概率统计理论可以得知，随着调用量的增大，其实际效果越来越接近于平均分配流量到每一台后端服务器，也就是轮询的效果。

随机法的代码实现大致如下：

1 public class Random

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 ArrayList<String> keyList = new ArrayList<String>();

13 keyList.addAll(keySet);

14

15 java.util.Random random = new java.util.Random();

16 int randomPos = random.nextInt(keyList.size());

17

18 return keyList.get(randomPos);

19 }

20 }

整体代码思路和轮询法一致，先重建serverMap，再获取到server列表。在选取server的时候，通过Random的nextInt方法取0~keyList.size()区间的一个随机值，从而从服务器列表中随机获取到一台服务器地址进行返回。**基于概率统计的理论，吞吐量越大，随机算法的效果越接近于轮询算法的效果**。

**源地址哈希（Hash）法**

源地址哈希的思想是获取客户端访问的IP地址值，通过哈希函数计算得到一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是要访问的服务器的序号。源地址哈希算法的代码实现大致如下：

1 public class Hash

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 ArrayList<String> keyList = new ArrayList<String>();

13 keyList.addAll(keySet);

14

15 // 在Web应用中可通过HttpServlet的getRemoteIp方法获取

16 String remoteIp = "127.0.0.1";

17 int hashCode = remoteIp.hashCode();

18 int serverListSize = keyList.size();

19 int serverPos = hashCode % serverListSize;

20

21 return keyList.get(serverPos);

22 }

23 }

前两部分和轮询法、随机法一样就不说了，差别在于路由选择部分。通过客户端的ip也就是remoteIp，取得它的Hash值，对服务器列表的大小取模，结果便是选用的服务器在服务器列表中的索引值。

**源地址哈希法的优点在于：保证了相同客户端IP地址将会被哈希到同一台后端服务器，直到后端服务器列表变更。根据此特性可以在服务消费者与服务提供者之间建立有状态的session会话**。

**源地址哈希算法的缺点在于：除非集群中服务器的非常稳定，基本不会上下线，否则一旦有服务器上线、下线，那么通过源地址哈希算法路由到的服务器是服务器上线、下线前路由到的服务器的概率非常低，如果是session则取不到session，如果是缓存则可能引发"雪崩"**。如果这么解释不是很明白，可以看我之前的一篇文章[MemCache超详细解读](http://www.cnblogs.com/xrq730/p/4948707.html)，一致性Hash算法部分。

**加权轮询（Weight Round Robin）法**

不同的服务器可能机器配置和当前系统的负载并不相同，因此它们的抗压能力也不尽相同，给配置高、负载低的机器配置更高的权重，让其处理更多的请求，而低配置、高负载的机器，则给其分配较低的权重，降低其系统负载。加权轮询法可以很好地处理这一问题，并将请求顺序按照权重分配到后端。加权轮询法的代码实现大致如下：

1 public class WeightRoundRobin

2 {

3 private static Integer pos;

4

5 public static String getServer()

6 {

7 // 重建一个Map，避免服务器的上下线导致的并发问题

8 Map<String, Integer> serverMap =

9 new HashMap<String, Integer>();

10 serverMap.putAll(IpMap.serverWeightMap);

11

12 // 取得Ip地址List

13 Set<String> keySet = serverMap.keySet();

14 Iterator<String> iterator = keySet.iterator();

15

16 List<String> serverList = new ArrayList<String>();

17 while (iterator.hasNext())

18 {

19 String server = iterator.next();

20 int weight = serverMap.get(server);

21 for (int i = 0; i < weight; i++)

22 serverList.add(server);//已经按照权重重复添加同一服务器

23 }

24

25 String server = null;

26 synchronized (pos)

27 {

28 if (pos > keySet.size())

29 pos = 0;

30 server = serverList.get(pos);

31 pos ++;

32 }

33

34 return server;

35 }

36 }

与轮询法类似，只是在获取服务器地址之前增加了一段权重计算的代码，根据权重的大小，将地址重复地增加到服务器地址列表中，权重越大，该服务器每轮所获得的请求数量越多。

**加权随机（Weight Random）法**

与加权轮询法类似，加权随机法也是根据后端服务器不同的配置和负载情况来配置不同的权重。不同的是，它是按照权重来随机选择服务器的，而不是顺序。加权随机法的代码实现如下：

1 public class WeightRandom

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 Iterator<String> iterator = keySet.iterator();

13

14 List<String> serverList = new ArrayList<String>();

15 while (iterator.hasNext())

16 {

17 String server = iterator.next();

18 int weight = serverMap.get(server);

19 for (int i = 0; i < weight; i++)

20 serverList.add(server);

21 }

22

23 java.util.Random random = new java.util.Random();

24 int randomPos = random.nextInt(serverList.size());

25

26 return serverList.get(randomPos);//权重大的重复添加多，概率就大

27 }

28 }

这段代码相当于是随机法和加权轮询法的结合，比较好理解，就不解释了。

**最小连接数（Least Connections）法**

前面几种方法费尽心思来实现服务消费者请求次数分配的均衡，当然这么做是没错的，可以为后端的多台服务器平均分配工作量，最大程度地提高服务器的利用率，但是实际情况是否真的如此？实际情况中，请求次数的均衡真的能代表负载的均衡吗？这是一个值得思考的问题。

上面的问题，再换一个角度来说就是：**以后端服务器的视角来观察系统的负载，而非请求发起方来观察**。最小连接数法便属于此类。

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它正是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前请求，尽可能地提高后端服务器的利用效率，将负载合理地分流到每一台机器。由于最小连接数设计服务器连接数的汇总和感知，设计与实现较为繁琐，此处就不说它的实现了。

**==================================================================================   
  
我不能保证写的每个地方都是对的，但是至少能保证不复制、不黏贴，保证每一句话、每一行代码都经过了认真的推敲、仔细的斟酌。每一篇文章的背后，希望都能看到自己对于技术、对于生活的态度。  
  
我相信乔布斯说的，只有那些疯狂到认为自己可以改变世界的人才能真正地改变世界。面对压力，我可以挑灯夜战、不眠不休；面对困难，我愿意迎难而上、永不退缩。  
  
其实我想说的是，我只是一个程序员，这就是我现在纯粹人生的全部。**