负载均衡的那些算法们

原创 2016-05-29 简单的老王 [SimpleMain](http://mp.weixin.qq.com/s?__biz=MzA3MDExNzcyNA==&mid=2650392075&idx=1&sn=fca2ebeca258e15f78a43c44bbb6153d&scene=19##)

上周发了问卷，想了解一下大家对老王有没有什么建议，然后好多朋友都投了票，想了解编程技术和服务器架构的干货，所以接下来会先聊聊编程和架构相关的算法，然后大概在6月下旬会跟大家聊聊面试那些事儿（老王到目前大约参加了几百次的面试，可以从面试官的角度来聊聊不一样的面试）。老王聊技术有个特点，就是绝不假大空，只求贴地飞行。所以，聊的东西一定会跟实际有关联，大家在平时也有可能用得着。

今天跟大伙儿聊的是负载均衡相关的一些算法。老王在百度的时候（估计是5-6年前），写过一个通用的基础库（不知道现在还有没有部门在用），用来做不同系统间负载均衡。太细节的东东估计想不起来了，不过基本的算法可以跟大家做做分享。

那第一个问题：**what's load-balance?**

假设我有两个模块（或者两个系统）：module-A和module-B，A依赖B提供服务。当用户请求过来的时候，A就会去请求B，让B根据请求进行某些处理（比如：根据单词id查对应的单词），完成后把结果返回给A，A再对这个结果进行处理。然而，为了保证服务稳定，有可能B服务有很多台机器，A遇到这个时候就犯难了：我该去找B的哪台机器取数据呢？

最常见的一个case就是nginx：比如我们的web逻辑服务器是jetty或者tomcat，一般会有多台，nginx就需要配置这多台机器：

upstream simplemain.com {

     server  192.168.1.100:8080;

     server  192.168.1.101:8080;

}

那这些机器是怎么样选择的呢？实际就是负载均衡算法。

老王对负载均衡的理解，他应该包含两个层面：

1、负载：就是后端系统的承载能力。比如同等条件下，一个1核cpu-1G内存的机器的承载能力一般会比8核cpu-8G内存的机器要差；相同配置下，一个cpu利用率为80%的机器比30%的承载能力一般要差等等。

2、均衡：保证后端请求的平衡。比如：在同等情况下，分配到多台机器的请求要相当；有些情况下，同一用户尽可能分配到同一台机器等等。

所以，负载均衡的算法实际上就是解决跨系统调用的时候，在考虑后端机器承载情况的前提下，保证请求分配的平衡和合理。

那第二个问题随之而来：why？

为什么要有负载均衡呢？

1、很明显，如果我们不去考虑后端的承载情况，有可能直接就把某台机器压垮了（比如cpu利用率已经80%了，再给大量的请求直接就干死了），更严重的会直接造成雪崩（一台压死了，对应的请求又压倒其他某台机器上，又干死一台……），从而致使服务瘫痪。

2、如果我们均衡算法选的不好，就会导致后端资源浪费。比如：如果选择一致Hash算法，可以很好利用cache的容量。而如果用随机，有可能就会让cache效果大打折扣（每台机器上都要缓存几乎相同的内容）。

所以，用负载均衡应该是一个比较好的选择。

那就解决第三个问题吧：**how？**

按照之前的思路，我们还是分成两个部分来讲：负载& 均衡。

1、先来看**负载算法**：

既然要解决后端系统的承载能力，那我们就有很多方式，常见的有以下几种：

A、简单粗暴有效的：手工配置！

大家是不是觉得这个听起来很山寨呢？其实不是。这种方式对于中小系统来讲是最有效最稳定的。因为后端机器的性能配置、上面部署了哪些服务、还能有多大的承载能力等等，我们是最清楚的。那我们在配置的时候，就可以明确的告诉调用者，你只能分配多大的压力到某台服务器上，多了不行！

比如，我们经常看到nginx的配置：

upstream simplemain.com {

     server  192.168.1.100:8080 weight=30;

     server  192.168.1.101:8080 weight=70;

}

就是说，虽然有两台后端的服务器，但是他们承载能力是不一样的，有一个能力更强，我们就给他70%的压力；有一个更弱，我们就给他30%的压力。这样，nginx就会把更多的压力分配给第二台。

这种方式配置简单，而且很稳定，基本不会产生分配的抖动。不过，带来的问题就是分配很固定，不能动态调整。如果你的后端服务器有一段时间出现性能抖动（比如有其他服务扰动了机器的稳定运行，造成cpu利用率一段时间升高），前端调用者就很难根据实际的情况重新分配请求压力。所以，引入了第二种方法。

B、动态调整。

这种方案会根据机器当前运行的状态和历史平均值进行对比，发现如果当前状态比历史的要糟糕，那么就动态减少请求的数量。如果比历史的要好，那么就可以继续增加请求的压力，直到达到一个平衡。

具体怎么做呢？

首先，刚开始接入的时候，我们可以计算所有机器对于请求的响应时间，算一个平均值。对于响应较快的机器，我们可以多分配一些请求。如果请求多了导致响应减慢，这个时候就会逐步和其他机器持平，说明这台机器达到了相应的平衡。

接着，当接入达到平衡以后，就可以统计这台机器平均的响应时间。如果某一段响应请求变慢了（同时比其他机器都要慢），就可以减少对他请求的分配，将压力转移一部分到其他机器，直到所有机器达到一个整体的平衡。

这种方案是不是看起来很高级呢？他的好处在于可以动态的来平衡后面服务器的处理能力。不过，任何事物都有两面性。这种方案如果遇到极端情况，可能会造成系统雪崩！当某台机器出现短暂网络抖动的时候，他的响应就可能变慢，这个时候，前端服务就会将他的请求分配给其他的机器。如果分配的很多，就有可能造成某些机器响应也变慢。然后又将这些机器的请求分配给另外的……如此这般，那些勤勤恳恳的机器终将被这些请求压死。

所以，更好的方案，将两者结合。一方面静态配置好承载负荷的一个范围，超过最大的就扔掉；另一方面动态的监控后端机器的响应情况，做小范围的请求调整。

2、**均衡算法**

均衡算法主要解决将请求如何发送给后端服务。经常会用到以下四种算法：随机（random）、轮训（round-robin）、一致哈希（consistent-hash）和主备（master-slave）。

比如：我们配置nginx的时候，经常会用到这样的配置：

upstream simplemain.com {

     ip\_hash;

     server  192.168.1.100:8080;

     server  192.168.1.101:8080;

}

这个配置就是按ip做hash算法，然后分配给对应的机器。

接下来我们详细的看看这几个算法是如何来工作的。

**A、随机算法**。

顾名思义，就是在选取后端服务器的时候，采用随机的一个方法。在具体讲这个算法之前，我们先来看看一个例子，我们写如下C语言的代码:

#include<stdlib.h>

#include<stdio.h>

int main()

{

        srand(1234);

        printf("%d\n", rand());

        return 0;

}

我们用srand函数给随机算法播了一个1234的种子，然后再去随机数，接着我们编译和链接gcc rand.c -o rand

按理想中说，我们每次运行rand这个程序，都应该得到不一样的结果，对吧。可是……

可以看到，我们每次运行的结果都是一样的！！出了什么问题呢？

我们说的随机，在计算机算法中通常采用的是一种伪随机的算法。我们会先给算法放一个种子，然后根据一定的算法将种子拿来运算，最后得到一个所谓的随机值。我们将上面的算法做一个小小的改动，将1234改为time(NULL)，效果就不一样了：

#include<stdlib.h>

#include<stdio.h>

#include<time.h>

int main()

{

        srand((int)time(NULL));

        printf("%d\n", rand());

        return 0;

}

time这个函数会获取当前秒数，然后将这个值作为种子放入到伪随机函数，从而计算出的伪随机值会因为秒数不一样而不同。

具体来看一下java源代码里如何来实现的。我们常用的java随机类是java.util.Random这个类。他提供了两个构造函数：

**public** Random() {

**this**(*seedUniquifier*() ^ System.*nanoTime*());

}

**public** Random(**long** seed) {

**if** (getClass() == Random.**class**)

**this**.seed = **new** AtomicLong(*initialScramble*(seed));

**else** {

        //subclass might have overriden setSeed

**this**.seed = **new** AtomicLong();

        setSeed(seed);

    }

}

我们可以看到，这个类也是需要一个种子。然后我们获取随机值的时候，会调用next函数：

**protectedint** next(**int** bits) {

**long** oldseed, nextseed;

    AtomicLong seed = **this**.seed;

**do** {

        oldseed = seed.get();

        nextseed = (oldseed \* *multiplier* + *addend*) & *mask*;

    } **while** (!seed.compareAndSet(oldseed, nextseed));

**return** (**int**)(nextseed>>> (48 - bits));

}

这个函数会利用种子进行一个运算，然后得到随机值。所以，我们看起来随机的一个算法，实际上跟时间是相关的，跟算法的运算是相关的。并不是真正的随机。

好了，话归正题，我们用随机算法怎么样做请求均衡呢？比如，还是我们之前那个nginx配置：

upstream simplemain.com {

     server  192.168.1.100:8080 weight=30;

     server  192.168.1.101:8080 weight=70;

}

我们有两台机器，分别需要承载30%和70%的压力，那么我们算法就可以这样来写（伪代码）：

bool res = abs(rand()) % 100 < 30

这句话是什么意思呢？

1、我们先产生一个伪随机数：rand()

2、将这个伪随机数的转化为非负数: abs(rand())

3、将这个数取模100，将值转化到[0,100)的半开半闭区间：abs(rand()) % 100

4、看这个数是否落入了前30个数的区间[0,30)：abs(rand()) % 100 < 30

如果随机是均匀的话，他们落到[0,100)这个区间里一定是均匀的，所以只要在[0,30)这个区间里，我们就分给第一台机器，否则就分给第二台机器。

其实这里讲述的只是一种方法，还有很多其他的方法，大家都可以去想想。

随机算法是我们最最最最最最常用的算法，绝大多数情况都使用他。首先，从概率上讲，它能保证我们的请求基本是分散的，从而达到我们想要的均衡效果；其次，他又是无状态的，不需要维持上一次的选择状态，也不需要均衡因子等等。总体上，方便实惠又好用，我们一直用他！

**B、轮训算法**。

轮训算法就像是挨个数数一样（123-123-123……），一个个的轮着来。

upstream simplemain.com {

     server  192.168.1.100:8080 weight=30;

     server  192.168.1.101:8080 weight=70;

}

还是这个配置，我们就可以这样来做（为了方便，我们把第一台机器叫做A，第二台叫做B）：

1、我们先给两台机器做个排序的数组：array = [ABBABBABBB]

2、我们用一个计数指针来标明现在数组的位置：idx = 3

3、当一个请求来的时候，我们就把指针对应的机器选取出来，并且指针加一，挪到下一个位置。

这样，十个请求，我们就可以保证有3个一定是A，7个一定是B。

轮训算法在实际中也有使用，但是因为要维护idx指针，所以是有状态的。我们经常会用随机算法取代。

**C、一致哈希算法**。

这个算法是大家讨论最对，研究最多，神秘感最强的一个算法。老王当年刚了解这个算法的时候，也是花了很多心思去研究他。在百度上搜：“一致hash”，大概有321万篇相关文章。

大家到网上搜这个算法，一般都会讲将[0,232)所有的整数投射到一个圆上，然后再将你的机器的唯一编码（比如：IP）通过hash运算得到的整数也投射到这个圆上（Node-A、Node-B）。如果一个请求来了，就将这个请求的唯一编码（比如：用户id）通过hash算法运算得到的整数也投射到这个圆上（request-1、request-2），通过顺时针方向，找到第一个对应的机器。如下图：

当时老王看了这些文章也觉得很有道理，但是过了一段时间就忘了……自己琢磨了一段时间，不断的问自己，为什么要这样做呢？

过了很久，老王有了一些体会。实际上，一致Hash要解决的是两个问题：

1、散列的不变性：就是同一个请求（比如：同一个用户id）尽量的落入到一台机器，不要因为时间等其他原因，落入到不同的机器上了；

2、异常以后的分散性：当某些机器坏掉（或者增加机器），原来落到同一台机器的请求（比如：用户id为1，101，201），尽量分散到其他机器，不要都落入其他某一台机器。这样对于系统的冲击和影响最小。

有了以上两个原则，这个代码写起来就很好写了。比如我们可以这样做(假定请求的用户id=100）：

1、我们将这个id和所有的服务的IP和端口拼接成一个字符串：

str1 = "192.168.1.100:8080-100"

str2 = "192.168.1.101:8080-100"

2、对这些字符串做hash，然后得到对应的一些整数：

iv1 = hash(str1)

iv2 = hash(str2)

3、对这些整数做从大到小的排序，选出第一个。

好，现在来看看我们的这个算法是否符合之前说的两个原则。

1、散列的不变性：很明显，这个算法是可重入的，只要输入一样，结果肯定一样；

2、异常以后的分散性：当某台机器坏掉以后，原本排到第一的这些机器就被第二位的取代掉了。只要我们的hash算法是分散的，那么得到排到第二位的机器就是分散的。

所以，这种算法其实也能达到同样的目的。当然，可以写出同样效果的算法很多很多，大家也可以自己琢磨琢磨。最根本的，就是要满足以上说的原则。

一致Hash算法用的最多的场景，就是分配cache服务。将某一个用户的数据缓存在固定的某台服务器上，那么我们基本上就不用多台机器都缓存同样的数据，这样对我们提高缓存利用率有极大的帮助。

不过硬币都是有两面的，一致Hash也不例外。当某台机器出问题以后，这台机器上的cache失效，原先压倒这台机器上的请求，就会压到其他机器上。由于其他机器原先没有这些请求的缓存，就有可能直接将请求压到数据库上，造成数据库瞬间压力增大。如果压力很大的话，有可能直接把数据库压垮。

所以，在考虑用一致Hash算法的时候，一定要估计一下如果有机器宕掉后，后端系统是否能承受对应的压力。如果不能，则建议浪费一点内存利用率，使用随机算法。

**D、主备算法**。

这个算法核心的思想是将请求尽量的放到某个固定机器的服务上（注意这里是尽量），而其他机器的服务则用来做备份，如果出现问题就切换到另外的某台机器的服务上。

这个算法用的相对不是很多，只是在一些特殊情况下会使用这个算法。比如，我有多台Message Queue的服务，为了保证提交数据的时序性，我就想把所有的请求都尽量放到某台固定的服务上，当这台服务出现问题，再用其他的服务。

那怎么做呢？最简单的做法，我们就对每台机器的IP：Port做一个hash，然后按从大到小的顺序排序，第一个就是我们想要的结果。如果第一个出现问题，那我们再取第二个：head(sort(hash("IP:Port1"), hash("IP:Port2"), ……))

当然，还有其他做法。比如：老王做的Naming Service就用一个集中式的锁服务来判定当前的主服务器，并对他进行锁定。

好了，关于负载均衡相关的算法就大体上说这么多。其实还有一个相关话题没有说，就是健康检查。他的作用就是对所有的服务进行存活和健康检测，看是否需要提供给负载均衡做选择。如果一台机器的服务出现了问题，健康检查就会将这台机器从服务列表中去掉，让负载均衡算法看不到这台机器的存在。这个是给负载均衡做保障的，但是可以不划在他的体系内。不过也有看法是可以将这个也算在负载均衡算法中。因为这个算法的实现其实也比较复杂，老王这次就不讲这个算法了，可以放到接下来的文章中来分析。