18 | 如何使用负载均衡算法？

2018-10-02 胡忠想

18 | 如何使用负载均衡算法？
朗读人：胡忠想　09’35’’ | 4.40M

假设你订阅了一个别人的服务，从注册中心查询得到了这个服务的可用节点列表，而这个列表里包含了几十个节点，这个时候你该选择哪个节点发起调用呢？这就是今天我要给你讲解的关于客户端负载均衡算法的问题。

为什么要引入负载均衡算法呢？主要有两个原因：一个是要考虑调用的均匀性，也就是要让每个节点都接收到调用，发挥所有节点的作用；另一个是要考虑调用的性能，也就是哪个节点响应最快，优先调用哪个节点。

不同的负载均衡算法在这两个方面的考虑不同，下面我就来能给介绍常见的负载均衡算法及其应用场景。

常见的负载均衡算法

1. 随机算法

随机算法，顾名思义就是从可用的服务节点中，随机挑选一个节点来访问。

在实现时，随机算法通常是通过生成一个随机数来实现，比如服务有 10 个节点，那么就每一次生成一个 1～10 之间的随机数，假设生成的是 2，那么就访问编号为 2 的节点。

采用随机算法，在节点数量足够多，并且访问量比较大的情况下，各个节点被访问的概率是基本相同的。一个随机算法的代码实现，可以参考这个示例。

2. 轮询算法

轮询算法，顾名思义就是按照固定的顺序，把可用的服务节点，挨个访问一次。

在实现时，轮询算法通常是把所有可用节点放到一个数组里，然后按照数组编号，挨个访问。比如服务有 10 个节点，放到数组里就是一个大小为 10 的数组，这样的话就可以从序号为 0 的节点开始访问，访问后序号自动加 1，下一次就会访问序号为 1 的节点，以此类推。

轮询算法能够保证所有节点被访问到的概率是相同的。一个轮询算法的代码实现，可以参考这个示例。

3. 加权轮询算法

轮询算法能够保证所有节点被访问的概率相同，而加权轮询算法是在此基础上，给每个节点赋予一个权重，从而使每个节点被访问到的概率不同，权重大的节点被访问的概率就高，权重小的节点被访问的概率就小。

在实现时，加权轮询算法是生成一个节点序列，该序列里有 n 个节点，n 是所有节点的权重之和。在这个序列中，每个节点出现的次数，就是它的权重值。比如有三个节点：a、b、c，权重分别是 3、2、1，那么生成的序列就是{a、a、b、c、b、a}，这样的话按照这个序列访问，前 6 次请求就会分别访问节点 a 三次，节点 b 两次，节点 c 一次。从第 7 个请求开始，又重新按照这个序列的顺序来访问节点。

在应用加权轮询算法的时候，根据我的经验，要尽可能保证生产的序列的均匀，如果生成的不均匀会造成节点访问失衡，比如刚才的例子，如果生成的序列是{a、a、a、b、b、c}，就会导致前 3 次访问的节点都是 a。一个加权轮询算法的代码实现，可以参考这个示例。

4. 最少活跃连接算法

最少活跃连接算法，顾名思义就是每一次访问都选择连接数最少的节点。因为不同节点处理请求的速度不同，使得同一个服务消费者同每一个节点的连接数都不相同。连接数大的节点，可以认为是处理请求慢，而连接数小的节点，可以认为是处理请求快。所以在挑选节点时，可以以连接数为依据，选择连接数最少的节点访问。

在实现时，需要记录跟每一个节点的连接数，这样在选择节点时，才能比较出连接数最小的节点。一个最少活跃连接算法的代码实现，可以参考这个示例。

5. 一致性 hash 算法

一致性 hash 算法，是通过某个 hash 函数，把同一个来源的请求都映射到同一个节点上。一致性 hash 算法最大的特点就是同一个来源的请求，只会映射到同一个节点上，可以说是具有记忆功能。只有当这个节点不可用时，请求才会被分配到相邻的可用节点上。

一个一致性 hash 算法的代码实现，可以参考这个示例。

负载均衡算法的使用场景

上面这五种负载均衡算法，具体在业务中该如何选择呢？根据我的经验，它们的各自应用场景如下：

- 随机算法：实现比较简单，在请求量远超可用服务节点数量的情况下，各个服务节点被访问的概率基本相同，主要应用在各个服务节点的性能差异不大的情况下。
- 轮询算法：跟随机算法类似，各个服务节点被访问的概率也基本相同，也主要应用在各个服务节点性能差异不大的情况下。
- 加权轮询算法：在轮询算法基础上的改进，可以通过给每个节点设置不同的权重来控制访问的概率，因此主要被用在服务节点性能差异比较大的情况。比如经常会出现一种情况，因为采购时间的不同，新的服务节点的性能往往要高于旧的节点，这个时候可以给新的节点设置更高的权重，让它承担更多的请求，充分发挥新节点的性能优势。
- 最少活跃连接算法：与加权轮询算法预先定义好每个节点的访问权重不同，采用最少活跃连接算法，客户端同服务端节点的连接数是在时刻变化的，理论上连接数越少代表此时服务端节点越空闲，选择最空闲的节点发起请求，能获取更快的响应速度。尤其在服务端节点性能差异较大，而又不好做到预先定义权重时，采用最少活跃连接算法是比较好的选择。
- 一致性 hash 算法：因为它能够保证同一个客户端的请求始终访问同一个服务节点，所以适合服务端节点处理不同客户端请求差异较大的场景。比如服务端缓存里保存着客户端的请求结果，如果同一客户端一直访问一个服务节点，那么就可以一直从缓存中获取数据。

这五种负载均衡算法是业界最常用的，不光在 RPC 调用中被广泛采用，在一些负载均衡组件比如 Nginx 中也有应用，所以说是一种通用的负载均衡算法，但是不是所有的业务场景都能很好解决呢？

我曾经遇到过这种场景：

- 服务节点数量众多，且性能差异比较大；
- 服务节点列表经常发生变化，增加节点或者减少节点时有发生；

- 客户端和服务节点之间的网络情况比较复杂，有些在一个数据中心，有些不在一个数据中心需要跨网访问，而且网络经常延迟或者抖动。

显然无论是随机算法还是轮询算法，第一个情况就不满足，加权轮询算法需要预先配置服务节点的权重，在节点列表经常变化的情况下不好维护，所以也不适合。而最少活跃连接算法是从客户端自身维度去判断的，在实际应用时，并不能直接反映出服务节点的请求量大小，尤其是在网络情况比较复杂的情况下，并不能做到动态的把请求发送给最合适的服务节点。至于一致性 hash 算法，显然不适合这种场景。

针对上面这种场景，有一种算法更加适合，这种算法就是自适应最优选择算法。

自适应最优选择算法

这种算法的主要思路是在客户端本地维护一份同每一个服务节点的性能统计快照，并且每隔一段时间去更新这个快照。在发起请求时，根据“二八原则”，把服务节点分成两部分，找出 20% 的那部分响应最慢的节点，然后降低权重。这样的话，客户端就能够实时的根据自身访问每个节点性能的快慢，动态调整访问最慢的那些节点的权重，来减少访问量，从而可以优化长尾请求。

由此可见，自适应最优选择算法是对加权轮询算法的改良，可以看作是一种动态加权轮询算法。它的实现关键之处就在于两点：第一点是每隔一段时间获取客户端同每个服务节点之间调用的平均性能统计；第二点是按照这个性能统计对服务节点进行排序，对排在性能倒数 20% 的那部分节点赋予一个较低的权重，其余的节点赋予正常的权重。

在具体实现时，针对第一点，需要在内存中开辟一块空间记录客户端同每一个服务节点之间调用的平均性能，并每隔一段固定时间去更新。这个更新的时间间隔不能太短，太短的话很容易受瞬时的性能抖动影响，导致统计变化太快，没有参考性；同时也不能太长，太长的话时效性就会大打折扣，效果不佳。根据我的经验，1 分钟的更新时间间隔是个比较合适的值。

针对第二点，关键点是权重值的设定，即使服务节点之间的性能差异较大，也不适合把权重设置得差异太大，这样会导致性能较好的节点与性能较差的节点之间调用量相差太大，这样也不是一种合理的状态。在实际设定时，可以设置 20% 性能较差的节点权重为 3，其余节点权重为 5。

总结

今天我给你讲解了最常用的五种客户端负载均衡算法的原理以及适用场景，在业务实践的过程汇总，究竟采用哪种，需要根据实际情况来决定，并不是算法越复杂越好。

比如在一种简单的业务场景下，有 10 个服务节点，并且配置基本相同，位于同一个数据中心，此时客户端选择随机算法或者轮询算法既简单又高效，并没有必要选择加权轮询算法或者最少活跃连接算法。

但在遇到前面提到的那种复杂业务场景下，服务节点数量众多，配置差异比较大，而且位于不同的数据中心，客户端与服务节点之间的网络情况也比较复杂，这个时候简单的负载均衡算法通常都难以应对，需要针对实际情况，选择更有针对性的负载均衡算法，比如自适应最优选择算法。

思考题

今天我给你讲的都属于软件层面的负载均衡算法，它与 F5 这种硬件负载均衡器有什么不同呢？

欢迎你在留言区写下自己的思考，与我一起讨论。

扩展阅读：

一致性 hash 算法是如何做到添加或者删除节点对整体请求的分布影响不大：
https://www.codeproject.com/Articles/56138/Consistent-hashing

写留言

精选留言

阿恺 1

在加权轮询算法中，通过一个随机数去访问生成序列，就不需要考虑生成序列的顺序是否合理的问题。

2018-10-02