28 | 分布式高可靠之负载均衡：不患寡，而患不均

2019-12-02 聂鹏程

分布式技术原理与算法解析 进入课程 >

讲述：聂鹏程
时长 16:38 大小 15.25M

你好！我是聂鹏程。今天，我来继续带你打卡分布式核心技术。

到目前为止，我已经为你介绍了分布式起源、分布式协调与同步、分布式资源管理与负载调度、分布式计算技术、分布式通信技术和分布式数据存储。可以说，掌握了这些内容，基本上就掌握了分布式的关键技术。

然而，只有可靠的分布式系统才能真正应用起来。那么，分布式系统的可靠性又是如何实现的呢？

不要着急，接下来几篇文章，我会和你一起学习分布式可靠性相关的知识，包括负载均衡、流量控制、故障隔离和故障恢复。

在这其中，负载均衡是分布式可靠性中非常关键的一个问题或技术，在一定程度上反映了分布式系统对业务处理的能力。比如，早期的电商抢购活动，当流量过大时，你可能就会发现有些地区可以购买，而有些地区因为服务崩溃而不能抢购。这，其实就是系统的负载均衡出现了问题。

接下来，我们就一起来打卡分布式高可靠之负载均衡。

什么是负载均衡?

先举个例子吧。以超市收银为例，假设现在只有一个窗口、一个收银员：

一般情况下，收银员平均 2 分钟服务一位顾客，10 分钟可以服务 5 位顾客；

到周末高峰期时，收银员加快收银，平均 1 分钟服务一位顾客，10 分钟最多服务 10 位顾客，也就是说一个顾客最多等待 10 分钟；

逢年过节，顾客数量激增，一下增加到 30 位顾客，如果仍然只有一个窗口和一个收银员，那么所有顾客就只能排队等候了，一个顾客最多需要等待 30 分钟。这样购物体验，就非常差了。

那有没有解决办法呢?

当然有。那就是新开一个收银窗口，每个收银窗口服务 15 个顾客，这样最长等待时间从 30 分钟缩短到 15 分钟。但如果，这两个窗口的排队顾客数严重不均衡，比如一个窗口有 5 个顾客排队，另一个窗口却有 25 个顾客排队，就不能最大化地提升顾客的购物体验。

所以，尽可能使得每个收银窗口排队的顾客一样多，才能最大程度地减少顾客的最长排队时间，提高用户体验。

看完这个例子，你是不是想到了一句话“不患寡，而患不均”？这，其实就是负载均衡的基本原理。

通常情况下，**负载均衡可以分为两种**：

一种是请求负载均衡，即将用户的请求均衡地分发到不同的服务器进行处理；

另一种是数据负载均衡，即将用户更新的数据分发到不同的存储服务器。

我在第 25 篇文章分享数据分布方法时，提到：数据分布算法很重要的一个衡量标准，就是均匀分布。可见，哈希和一致性哈希等，其实就是数据负载均衡的常用方法。那么今天，**我就与你着重说说服务请求的负载均衡技术吧。**

分布式系统中，服务请求的负载均衡是指，当处理大量用户请求时，请求应尽量均衡地分配到多台服务器进行处理，每台服务器处理其中一部分而不是所有的用户请求，以完成高并发的请求处理，避免因单机处理能力的上限，导致系统崩溃而无法提供服务的问题。

比如，有 N 个请求、M 个节点，负载均衡就是将 N 个请求，均衡地转发到这 M 个节点进行处理。

服务请求的负载均衡方法

通常情况下，计算机领域中，在不同层有不同的负载均衡方法。比如，从网络层的角度，通常有基于 DNS、IP 报文等的负载均衡方法；在中间件层（也就是我们专栏主要讲的分布式系统层），常见的负载均衡策略主要包括轮询策略、随机策略、哈希和一致性哈希等策略。

今天，我着重与你分析的就是，中间件层所涉及的负载均衡策略。接下来，我们就具体看看吧。

轮询策略

轮询策略是一种实现简单，却很常用的负载均衡策略，核心思想是服务器轮流处理用户请求，以尽可能使每个服务器处理的请求数相同。生活中也有很多类似的场景，比如，学校宿舍里，学生每周轮流打扫卫生，就是一个典型的轮询策略。

在负载均衡领域中，轮询策略主要包括顺序轮询和加权轮询两种方式。

首先，我们一起看看**顺序轮询**。假设有 6 个请求，编号为请求 1~6，有 3 台服务器可以处理请求，编号为服务器 1~3，如果采用顺序轮询策略，则会按照服务器 1、2、3 的顺序轮流进行请求。

如表所示，将 6 个请求当成 6 个步骤：

1. 请求 1 由服务器 1 处理；

2. 请求 2 由服务器 2 进行处理。

3. 以此类推，直到处理完这 6 个请求。

步骤	请求编号	选择的服务器
1	1	1
2	2	2
3	3	3
4	4	1
5	5	2
6	6	3

最终的处理结果是，服务器 1 处理请求 1 和请求 4，服务器 2 处理请求 2 和请求 5，服务器 3 处理请求 3 和请求 6。

接下来，我们看一下**加权轮询**。

加权轮询为每个服务器设置了优先级，每次请求过来时会挑选优先级最高的服务器进行处理。比如服务器 1~3 分配了优先级{4，1，1}，这 6 个请求到来时，还当成 6 个步骤，如表所示。

1. 请求 1 由优先级最高的服务器 1 处理，服务器 1 的优先级相应减 1，此时各服务器优先级为{3，1，1}；

2. 请求 2 由目前优先级最高的服务器 1 进行处理，服务器 1 优先级相应减 1，此时各服务器优先级为{2，1，1}。
3. 以此类推，直到处理完这 6 个请求。每个请求处理完后，相应服务器的优先级会减 1。

步骤	请求编号	各服务器优先级	选择的服务器
1	1	{4,1,1}	1
2	2	{3,1,1}	1
3	3	{2,1,1}	1
4	4	{1,1,1}	1
5	5	{0,1,1}	2
6	6	{0,0,1}	3

最终的处理结果是，服务器 1 处理请求 1~4，服务器 2 处理请求 5，服务器 3 会处理请求 6。

以上就是顺序轮询和加权轮询的核心原理了。轮询策略的应用比较广泛，比如 **Nginx 默认的负载均衡策略就是一种改进的加权轮询策略。**我们具体看看它的核心原理吧。

首先，我来解释下 Nginx 轮询策略需要用到的变量吧。

weight：配置文件中为每个服务节点设置的服务节点权重，固定不变。

effective_weight：服务节点的有效权重，初始值为 weight。 在 Nginx 的源码中有一个最大失败数的变量 max_fails，当服务发生异常时，则减少相应服务节点的有效权重，公式为 effective_weight = effective_weight - weight / max_fails；之后再次选取本节点，若服务调用成功，则增加有效权重，effective_weight ++ ，直至恢复到 weight。

current_weight：服务节点当前权重，初始值均为 0，之后会根据系统运行情况动态变化。

假设，各服务器的优先级是{4，1，1}，我还是将 6 个请求分为 6 步来进行讲解，如表所示：

1. 遍历集群中所有服务节点，使用 current_weight = current_weight + effective_weight，计算此时每个服务节点的 current_weight，得到 current_weight 为{4，1，1}，total 为 4+1+1=6。选出 current_weight 值最大的服务节点即服务器 1 来处理请求，随后服务器 1 对应的 current_weight 减去此时的 total 值，即 4 - 6，变为了 -2 。
2. 按照上述步骤执行，首先遍历，按照 current_weight = current_weight + effective_weight 计算每个服务节点 current_weight 的值，结果为{2，2，2}，total 为 6，选出 current_weight 值最大的服务节点。current_weight 最大值有多个服务节点时，直接选择第一个节点即可，在这里选择服务器 1 来处理请求，随后服务器 1 对应的 current_weight 值减去此时的 total，即 2 - 6，结果为 -4。
3. 以此类推，直到处理完这 6 个请求。

步骤	请求编号	选择前各服务器的current_weight值	选择前各服务器current_weight累加和total值	选择的服务器	选择后各服务器的current_weight值
1	1	{4,1,1}	6	1	{-2,1,1}
2	2	{2,2,2}	6	1	{-4,2,2}
3	3	{0,3,3}	6	2	{0,-3,3}
4	4	{4,-2,4}	6	1	{-2,-2,4}
5	5	{2,-1,5}	6	3	{2,-1,-1}
6	6	{6,0,0}	6	1	{0,0,0}

最终的处理结果为，服务器 1 处理请求 1、2、4、6，服务器 2 处理请求 3，服务器 3 会处理请求 5。

可以看到，与普通的加权轮询策略相比，这种轮询策略的优势在于，**当部分请求到来时，不会集中落在优先级较高的那个服务节点。**

还是上面的例子，假设只有 4 个请求，按照普通的加权轮询策略，会全部由服务器 1 进行处理，即{1,1,1,1}；而按照这种平滑的加权轮询策略的话，会由服务器 1 和 2 共同进行处理，即{1,1,2,1}。

轮询策略的优点就是，实现简单，且对于请求所需开销差不多时，负载均衡效果比较明显，同时加权轮询策略还考虑了服务器节点的异构性，即可以让性能更好的服务器具有更高的优先级，从而可以处理更多的请求，使得分布更加均衡。

但**轮询策略的缺点**是，每次请求到达的目的节点不确定，不适用于有状态请求的场景。并且，轮询策略主要强调请求数的均衡性，所以不适用于处理请求所需开销不同的场景。

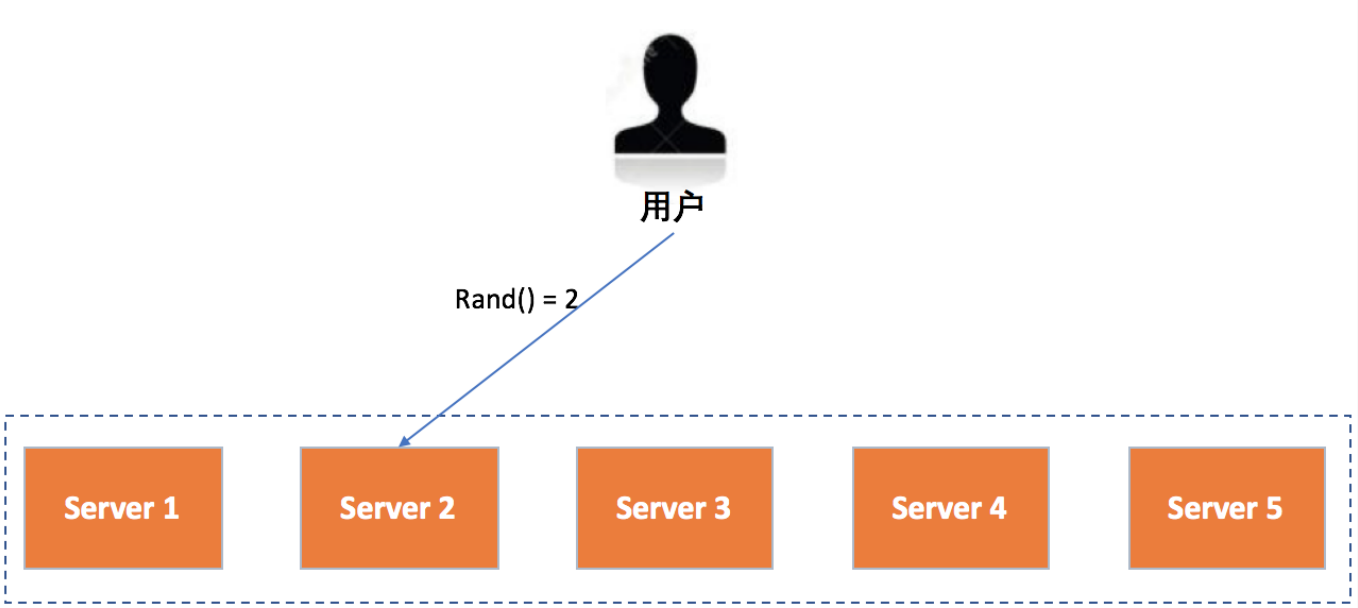
比如，有两个服务器（节点 A 和节点 B）性能相同，CPU 个数和内存均相等，有 4 个请求需要处理，其中请求 1 和请求 3 需要 1 个 CPU，请求 2 和请求 4 需要 2 个 CPU。根据轮询策略，请求 1 和请求 3 由节点 A、请求 2 和请求 4 由节点 B 处理。由此可见，节点 A 和节点 B 关于 CPU 的负载分别是 2 和 4，从这个角度来看，两个节点的负载并不均衡。

综上所述，**轮询策略适用于用户请求所需资源比较接近的场景**。

随机策略

随机策略也比较容易理解，指的就是当用户请求到来时，会随机发到某个服务节点进行处理，可以采用随机函数实现。这里，随机函数的作用就是，让请求尽可能分散到不同节点，防止所有请求放到同一节点或少量几个节点上。

如图所示，假设有 5 台服务器 Server 1~5 可以处理用户请求，每次请求到来时，都会先调用一个随机函数来计算出处理节点。这里，随机函数的结果只能是{1,2,3,4,5}这五个值，然后再根据计算结果分发到相应的服务器进行处理。比如，图中随机函数计算结果为 2，因此该请求会由 Server2 处理。



这种方式的优点是，实现简单，但缺点也很明显，与轮询策略一样，每次请求到达的目的节点不确定，不适用于有状态的场景，而且没有考虑到处理请求所需开销。除此之外，随机策略也没有考虑服务器节点的异构性，即性能差距较大的服务器可能处理的请求差不多。

因此，随机策略适用于，集群中服务器节点处理能力相差不大，用户请求所需资源比较接近的场景。

比如，我在第 19 篇文章中提到的 RPC 框架 Dubbo，当注册中心将服务提供方地址列表返回给调用方时，调用方会通过负载均衡算法选择其中一个服务提供方进行远程调用。关于负载均衡算法，Dubbo 提供了随机策略、轮询策略等。

哈希和一致性哈希策略

无论是轮询还是随机策略，对于一个客户端的多次请求，每次落到的服务器很大可能是不同的，如果这是一台缓存服务器，就会对缓存同步带来很大挑战。尤其是系统繁忙时，主从延迟带来的同步缓慢，可能会造成同一客户端两次访问得到不同的结果。解决方案就是，利用哈希算法定位到对应的服务器。

哈希和一致性哈希，是数据负载均衡的常用算法。 我在第 25 篇文章介绍哈希与一致性哈希时，提到过：数据分布算法的均匀性，一方面指数据的存储均匀，另一方面也指数据请求的均匀。

数据请求就是用户请求的一种，哈希、一致性哈希、带有限负载的一致性哈希和带虚拟节点的一致性哈希算法，同样适用于请求负载均衡。

所以，**哈希与一致性策略的优点**是，哈希函数设置合理的话，负载会比较均衡。而且，相同 key 的请求会落在同一个服务节点上，可以用于有状态请求的场景。除此之外，带虚拟节点的一致性哈希策略还可以解决服务器节点异构的问题。

但其**缺点是**，当某个节点出现故障时，采用哈希策略会出现数据大规模迁移的情况，采用一致性哈希策略可能会造成一定的数据倾斜问题。同样的，这两种策略也没考虑请求开销不同造成的不均衡问题。

应用哈希和一致性哈希策略的框架有很多，比如 Redis、Memcached、Cassandra 等，你可以再回顾下第 25 篇文章中的相关内容。

除了以上这些策略，还有一些负载均衡策略比较常用。比如，根据服务节点中的资源信息（CPU，内存等）进行判断，服务节点资源越多，就越有可能处理下一个请求；再比如，根据请求的特定需求，如请求需要使用 GPU 资源，那就需要由具有 GPU 资源的节点进行处理等。

对比分析

以上，就是轮询策略、随机策略、哈希和一致性哈希策略的主要内容了。接下来，我再通过一个表格对比下这三种方法，以便于你学习和查阅。

	优点	缺点	适用场景	典型应用/系统
轮询策略	实现简单，服务器负载均衡，可解决服务器节点异构的问题	1. 处理每次请求的服务器不确定，不适合有状态请求的场景 2. 没考虑请求开销不同造成的不均衡问题	适用于用户请求所需资源比较接近的场景，以及无状态请求场景	Nginx
随机策略	实现简单，服务器负载基本均衡	1. 处理每次请求的服务器不确定，不适合有状态请求的场景 2. 没考虑服务器节点异构性和请求开销不同造成的不均衡问题	适用于集群中服务器节点处理能力相差不大，用户请求所需资源比较接近的无状态请求的场景	Dubbo
哈希和一致性哈希策略	1. 哈希函数设置合理，服务器负载均衡，且相同key的请求可落在同一个服务器节点，适合有状态请求的场景 2. 带虚拟节点的一致性哈希策略，可以解决服务器节点异构的问题	1. 实现相对复杂 2. 没考虑请求开销不同造成的不均衡问题	适用于用户请求所需资源比较接近的场景，也适用于有状态请求的场景	Redis、Memcache、Cassandra

知识扩展：如果要考虑请求所需资源不同的话，应该如何设计负载均衡策略呢？

上面提到的轮询策略、随机策略，以及哈希和一致性哈希策略，主要考虑的是请求数的均衡，并未考虑请求所需资源不同造成的不均衡问题。那么，如何设计负载均衡策略，才能解决这个问题呢？

其实，这个问题的解决方案有很多，常见的思路主要是对请求所需资源与服务器空闲资源进行匹配，也称调度。

关于调度，不知你是否还记得第 11 篇文章所讲的单体调度？我们可以**使用单体调度的思路**，让集群选举一个主节点，每个从节点会向主节点汇报自己的空闲资源；当请求到来时，主节点通过资源调度算法选择一个合适的从节点来处理该请求。

在这篇文章中，我提到了最差匹配和最佳匹配算法。这两种算法各有利弊，最差匹配算法可以尽量将请求分配到不同机器，但可能会造成资源碎片问题；而最佳匹配算法，虽然可以留出一些“空”机器来处理开销很大的请求，但会造成负载不均的问题。因此，它们适用于不同的场景，你可以再回顾下第 11 篇文章中的相关内容。

除此之外，**一致性哈希策略**也可以解决这个问题：让请求所需的资源和服务器节点的空闲资源，与哈希函数挂钩，即通过将资源作为自变量，带入哈希函数进行计算，从而映射到哈希环中。

比如，我们设置的哈希函数结果与资源正相关，这样就可以让资源开销大的请求由空闲资源多的服务器进行处理，以实现负载均衡。但这种方式也有个缺点，即哈希环上的节点资源变化后，需要进行哈希环的更新。

总结

今天，我主要带你学习了分布式高可靠技术中的负载均衡。

首先，我以超市收银为例，与你介绍了什么是负载均衡。负载均衡包括数据负载均衡和请求负载均衡，我在第 25 篇文章中介绍的数据分布其实就是数据的负载均衡，所以我今天重点与你分享的是请求的负载均衡。

然后，我与你介绍了常见的负载均衡策略，包括轮询策略、随机策略、哈希和一致性哈希策略。其中，轮询策略和随机策略，因为每次请求到达的目的节点不确定，只适用于无状态请求的场景；而哈希和一致性哈希策略，因为相同 key 的请求会落在同一个服务节点上，所以可以用于有状态请求的场景。

最后，我再通过一张思维导图来归纳一下今天的核心知识点吧。

加油，相信通过本讲的学习，你对分布式系统中的负载均衡有了一定的理解，也可以进一步对电商系统、火车票系统等涉及的请求负载均衡的问题进行分析了。加油，行动起来吧！

思考题

在分布式系统中，负载均衡技术除了各节点共同分担请求外，还有什么好处呢？

我是聂鹏程，感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再会！

上一篇　27 | 分布式数据之缓存技术：“身手钥钱”随身带

下一篇　特别放送 | 徐志强：学习这件事儿，不到长城非好汉

精选留言 (4)

写留言

随心而至
2019-12-02

提高可用性。
假如一个实例挂了，可以自动切换到其他实例，对系统整个影响不会太大。

展开

1

阿西吧
2019-12-02

可以处理由于单点故障引起的系统不可用问题，提高系统的可用性

Jackey
2019-12-02

对一致性哈希+资源那有点不理解。想请问老师和各位大佬，如果空闲资源相同的话，是不是还要加入其他影响因素，否则多个节点在环中就等同于一个节点了。另外空闲资源一直在动态变化，这样还能保证相同的key的请求落在同一个节点吗？

展开

xingoo

2019-12-02

还可以实现高可用，一个节点坏掉，其他的也可以提供请求。