26-负载均衡：怎样提升系统的横向扩展能力？

你好，我是唐扬。

在基础篇中，我提到了高并发系统设计的三个通用方法：缓存、异步和横向扩展，到目前为止，你接触到了缓存的使用姿势，也了解了，如何使用消息队列异步处理业务逻辑，那么本节课，我将带你了解一下，如何提升系统的横向扩展能力。

在之前的课程中，我也提到过提升系统横向扩展能力的一些案例。比如，08讲提到，可以通过部署多个从库的方式，来提升数据库的扩展能力，从而提升数据库的查询性能，那么就需要借助组件，将查询数据库的请求，按照一些既定的策略分配到多个从库上，这是负载均衡服务器所起的作用，而我们一般使用DNS服务器来承担这个角色。

不过在实际的工作中，你经常使用的负载均衡的组件应该算是Nginx，它的作用是承接前端的HTTP请求，然后将它们按照多种策略，分发给后端的多个业务服务器上。这样，我们可以随时通过扩容业务服务器的方式，来抵挡突发的流量高峰。与DNS不同的是，Nginx可以在域名和请求URL地址的层面做更细致的流量分配，也提供更复杂的负载均衡策略。

你可能会想到，在微服务架构中，我们也会启动多个服务节点，来承接从用户端到应用服务器的请求，自然会需要一个负载均衡服务器，作为流量的入口，实现流量的分发。那么在微服务架构中，如何使用负载均衡服务器呢？

在回答这些问题之前，我先带你了解一下，常见的负载均衡服务器都有哪几类，因为这样，你就可以依据不同类型负载均衡服务器的特点做选择了。

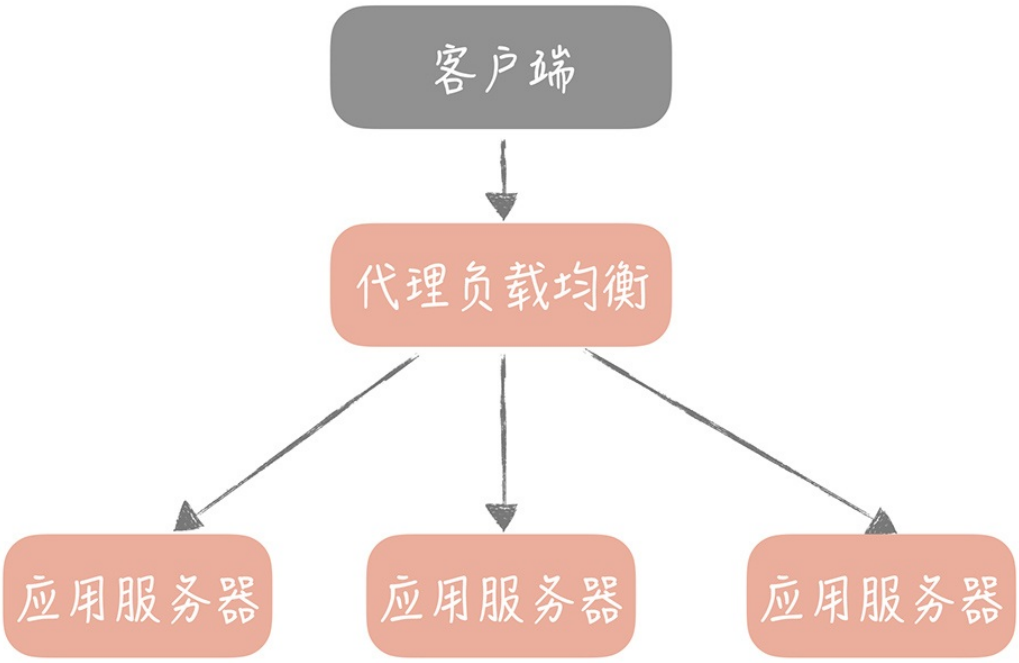
负载均衡服务器的种类

负载均衡的含义是：将负载（访问的请求）“均衡”地分配到多个处理节点上。这样可以减少单个处理节点的请求量，提升整体系统的性能。

同时，负载均衡服务器作为流量入口，可以对请求方屏蔽服务节点的部署细节，实现对于业务方无感知的扩容。它就像交通警察，不断地疏散交通，将汽车引入合适的道路上。

而在我看来，负载均衡服务大体上可以分为两大类：一类是代理类的负载均衡服务；另一类是客户端负载均衡服务。

代理类的负载均衡服务，以单独的服务方式部署，所有的请求都要先经过负载均衡服务，在负载均衡服务中，选出一个合适的服务节点后，再由负载均衡服务，调用这个服务节点来实现流量的分发。



代理负载均衡服务示意图

由于这类服务需要承担全量的请求，所以对于性能的要求极高。代理类的负载均衡服务有很多开源实现，比较著名的有LVS，Nginx等等。LVS在OSI网络模型中的第四层，传输层工作，所以LVS又可以称为四层负载；而Nginx运行在OSI网络模型中的第七层，应用层，所以又可以称它为七层负载（你可以回顾一下02讲的内容）。

在项目的架构中，我们一般会同时部署LVS和Nginx来做HTTP应用服务的负载均衡。也就是说，在入口处部署LVS，将流量分发到多个Nginx服务器上，再由Nginx服务器分发到应用服务器上，**为什么这么做呢?**

主要和LVS和Nginx的特点有关，LVS是在网络栈的四层做请求包的转发，请求包转发之后，由客户端和后端服务直接建立连接，后续的响应包不会再经过LVS服务器，所以相比Nginx，性能会更高，也能够承担更高的并发。

可LVS缺陷是工作在四层，而请求的URL是七层的概念，不能针对URL做更细致地请求分发，而且LVS也没有提供探测后端服务是否存活的机制；而Nginx虽然比LVS的性能差很多，但也可以承担每秒几万次的请求，并且它在配置上更加灵活，还可以感知后端服务是否出现问题。

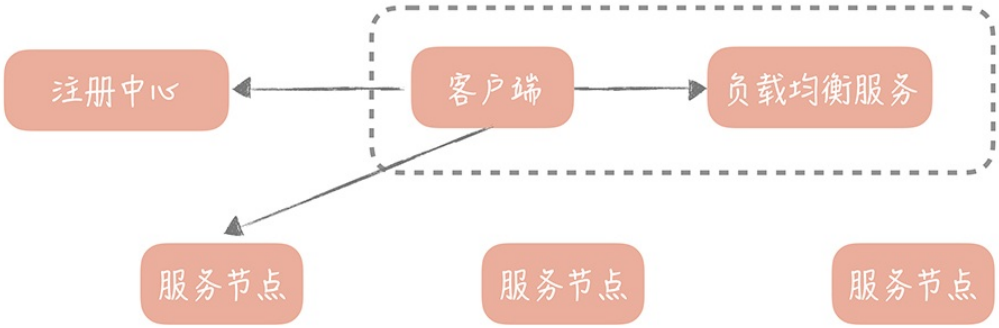
因此，LVS适合在入口处，承担大流量的请求分发，而Nginx要部署在业务服务器之前做更细维度的请求分发。**我给你的建议是，**如果你的QPS在十万以内，那么可以考虑不引入LVS而直接使用Nginx作为唯一的负载均衡服务器，这样少维护一个组件，也会减少系统的维护成本。

不过这两个负载均衡服务适用于普通的Web服务，对于微服务架构来说，它们是不合适的。因为微服务架构中的服务节点存储在注册中心里，使用LVS就很难和注册中心交互，获取全量的服务节点列表。另外，一般微服务架构中，使用的是RPC协议而不是HTTP协议，所以Nginx也不能满足要求。

所以，我们会使用另一类的负载均衡服务，客户端负载均衡服务，也就是把负载均衡的服务内嵌在RPC客户端中。

它一般和客户端应用，部署在一个进程中，提供多种选择节点的策略，最终为客户端应用提供一个最佳的，

可用的服务端节点。这类服务一般会结合注册中心来使用，注册中心提供服务节点的完整列表，客户端拿到列表之后使用负载均衡服务的策略选取一个合适的节点，然后将请求发到这个节点上。



客户端负载均衡服务示意图

了解负载均衡服务的分类，是你学习负载均衡服务的第一步，接下来，你需要掌握负载均衡策略，这样一来，你在实际工作中，配置负载均衡服务的时候，可以对原理有更深刻的了解。

常见的负载均衡策略有哪些

负载均衡策略从大体上来看可以分为两类：

- 一类是静态策略，也就是说负载均衡服务器在选择服务节点时，不会参考后端服务的实际运行的状态。
- 一类是动态策略，也就是说负载均衡服务器会依据后端服务的一些负载特性，来决定要选择哪一个服务节点。

常见的静态策略有几种，其中使用最广泛的是**轮询的策略（RoundRobin，RR），**这种策略会记录上次请求后端服务的地址或者序号，然后在请求时，按照服务列表的顺序，请求下一个后端服务节点。伪代码如下：

```
AtomicInteger lastCounter = getLastCounter();//获取上次请求的服务节点的序号
List<String> serverList = getServerList(); // 获取服务列表
int currentIndex = lastCounter.addAndGet(); //增加序列号
if(currentIndex >= serverList.size()) {
  currentIndex = 0;
}
setLastCounter(currentIndex);
return serverList.get(currentIndex);
```

它其实是一种通用的策略，基本上，大部分的负载均衡服务器都支持。轮询的策略可以做到将请求尽量平均地分配到所有服务节点上，但是，它没有考虑服务节点的具体配置情况。比如，你有三个服务节点，其中一个服务节点的配置是8核8G，另外两个节点的配置是4核4G，那么如果使用轮询的方式来平均分配请求的话，8核8G的节点分到的请求数量和4核4G的一样多，就不能发挥性能上的优势了

所以，我们考虑给节点加上权重值，比如给8核8G的机器配置权重为2，那么就会给它分配双倍的流量，**这种策略就是带有权重的轮询策略。**

除了这两种策略之外，目前开源的负载均衡服务还提供了很多静态策略：

- Nginx提供了ip_hash和url_hash算法；
- LVS提供了按照请求的源地址，和目的地址做hash的策略；
- Dubbo也提供了随机选取策略，以及一致性hash的策略。

但是在我看来，轮询和带有权重的轮询策略，能够将请求尽量平均地分配到后端服务节点上，也就能够做到对于负载的均衡分配，在没有更好的动态策略之前，应该优先使用这两种策略，比如Nginx就会优先使用轮询的策略。

而目前开源的负载均衡服务中，也会提供一些动态策略，我强调一下它们的原理。

在负载均衡服务器上会收集对后端服务的调用信息，比如从负载均衡端到后端服务的活跃连接数，或者是调用的响应时间，然后从中选择连接数最少的服务，或者响应时间最短的后端服务。**我举几个具体的例子：**

- Dubbo提供的LeastAcive策略，就是优先选择活跃连接数最少的服务；
- Spring Cloud全家桶中的Ribbon提供了WeightedResponseTimeRule是使用响应时间，给每个服务节点计算一个权重，然后依据这个权重，来给调用方分配服务节点。

这些策略的思考点是从调用方的角度出发，选择负载最小、资源最空闲的服务来调用，以期望能得到更高的服务调用性能，也就能最大化地使用服务器的空闲资源，请求也会响应地更迅速，**所以，我建议你，**在实际开发中，优先考虑使用动态的策略。

到目前为止，你已经可以根据上面的分析，选择适合自己的负载均衡策略，并选择一个最优的服务节点，**那么问题来了：**你怎么保证选择出来的这个节点，一定是一个可以正常服务的节点呢？如果你采用的是轮询的策略，选择出来的，是一个故障节点又要怎么办呢？所以，为了降低请求被分配到一个故障节点的几率，有些负载均衡服务器，还提供了对服务节点的故障检测功能。

如何检测节点是否故障

24讲中，我带你了解到，在微服务化架构中，服务节点会定期地向注册中心发送心跳包，这样注册中心就能够知晓服务节点是否故障，也就可以确认传递给负载均衡服务的节点，一定是可用的。

但对于Nginx来说，**我们要如何保证配置的服务节点是可用的呢？**

这就要感谢淘宝开源的Nginx模块nginx_upstream_check_module了，这个模块可以让Nginx定期地探测后端服务的一个指定的接口，然后根据返回的状态码，来判断服务是否还存活。当探测不存活的次数达到一定阈值时，就自动将这个后端服务从负载均衡服务器中摘除。**它的配置样例如下：**

```
upstream server {
        server 192.168.1.1:8080;
        server 192.168.1.2:8080;
        check interval=3000 rise=2 fall=5 timeout=1000 type=http default_down=true;//检测间隔为3秒，检测超时时间
        check_http_send "GET /health_check HTTP/1.0\r\n\r\n"; //检测URL
        check_http_expect_alive http_2xx; //检测返回状态码为200时认为检测成功
}
```

Nginx按照上面的方式配置之后，你的业务服务器也要实现一个“/health_check”的接口，在这个接口中返回的HTTP状态码，这个返回的状态码可以存储在配置中心中，这样在变更状态码时，就不需要重启服务了（配置中心在第33节课中会讲到）。

节点检测的功能，还能够帮助我们实现Web服务的优雅关闭。在24讲中介绍注册中心时，我曾经提到，服务的优雅关闭需要先切除流量再关闭服务，使用了注册中心之后，就可以先从注册中心中摘除节点，再重启服务，以便达到优雅关闭的目的。那么Web服务要如何实现优雅关闭呢？接下来，我来给你了解一下，有了节点检测功能之后，服务是如何启动和关闭的。

在服务刚刚启动时，可以初始化默认的HTTP状态码是500，这样Nginx就不会很快将这个服务节点标记为可用，也就可以等待服务中，依赖的资源初始化完成，避免服务初始启动时的波动。

在完全初始化之后，再将HTTP状态码变更为200，Nginx经过两次探测后，就会标记服务为可用。在服务关闭时，也应该先将HTTP状态码变更为500，等待Nginx探测将服务标记为不可用后，前端的流量也就不会继续发往这个服务节点。在等待服务正在处理的请求全部处理完毕之后，再对服务做重启，可以避免直接重启导致正在处理的请求失败的问题。**这是启动和关闭线上Web服务时的标准姿势，你可以在项目中参考使用。**

课程小结

本节课，我带你了解了与负载均衡服务相关的一些知识点，以及在实际工作中的运用技巧。我想强调几个重点：

- 网站负载均衡服务的部署，是以LVS承接入口流量，在应用服务器之前，部署Nginx做细化的流量分发，和故障节点检测。当然，如果你的网站的并发不高，也可以考虑不引入LVS。
- 负载均衡的策略可以优先选择动态策略，保证请求发送到性能最优的节点上；如果没有合适的动态策略，那么可以选择轮询的策略，让请求平均分配到所有的服务节点上。
- Nginx可以引入nginx_upstream_check_module，对后端服务做定期的存活检测，后端的服务节点在重启时，也要秉承着“先切流量后重启”的原则，尽量减少节点重启对于整体系统的影响。

你可能会认为，像Nginx、LVS应该是运维所关心的组件，作为开发人员不用操心维护。**不过通过今天的学习你应该可以看到：**负载均衡服务是提升系统扩展性，和性能的重要组件，在高并发系统设计中，它发挥的作用是无法替代的。理解它的原理，掌握使用它的正确姿势，应该是每一个后端开发同学的必修课。

一课一思

在实际的工作中，你一定也用过很多的负载均衡的服务和组件，那么在使用过程中你遇到过哪些问题呢，有哪些注意的点呢？欢迎在留言区与我分享你的经验。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

精选留言：

- XD 2019-11-22 10:23:10
  一口气读完。干货还是很多的。

- M 2019-11-22 10:15:38

请教下老师，app与服务器之间使用websocket协议连接，如何使用负载均衡呢？

- sdjdd 2019-11-22 10:13:34
  关闭服务之前，用 503 状态码响应健康检查是不是语义更明确一些。

- Demter 2019-11-22 05:40:25
  这里说的客户端是啥，是用户的浏览器吗。。。