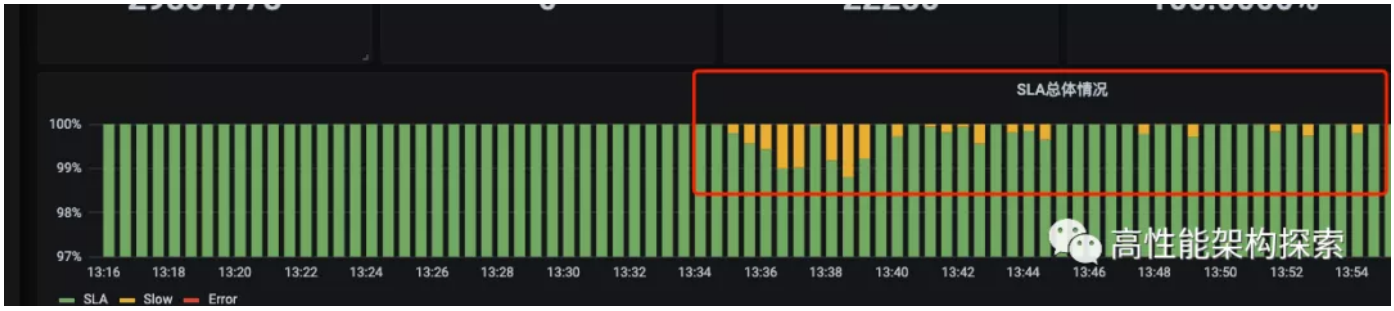
【万字长文】吃透负载均衡

首先告诉大家一件事，在十一国庆期间，引擎的机器又又。。。又扛不住了流量。



经过监控分析，发现某个服务的一个实例所在的虚拟机扛不住了，所以采取临时措施流量控制之后，问题解决了，但还是造成了不小的损失。

经过此次故障，以及分析故障的过程中对负载均衡又有了新的更加深入的认识，所以将这部分写出来，算是做个故障总结吧。

1写在前面

写本文的目的：

- 对负载均衡的理解零零散散，不成体系。

阅读这篇文章需要的条件：

- 对OSI模型有些许了解
- 有耐心。本文涉及大量的知识点，且只能用文字才能讲清楚，所以文字比较多。

收获：

- 读完此篇文章，从宏观的角度理解了负载均衡的原理以及实现机制。加深对分布式架构的了解

主要内容：

- 本文首先从概念开始，讲解什么是负载均衡，以及负载均衡在分布式系统中所承担的角色以及提供的功能。
- 讲解负载均衡的分类。分别从 ***软硬件角度***、***地域范围角度*** 以及 ***OSI模型角度*** 进行分类讲解负载均衡的实现方案。
- 从负载均衡的策略角度来分析目前业界的负载均衡算法以及其优缺点

好了，准备好了么，让我们开始这次愉快之旅。

2引言

首先 ***撇开对线上的影响***，如果线上突发来了流量,后端服务扛不住，我们会怎么做呢？无非两种方式：

- 提升机器配置(CPU、内存、硬盘、带宽等)
- 加机器

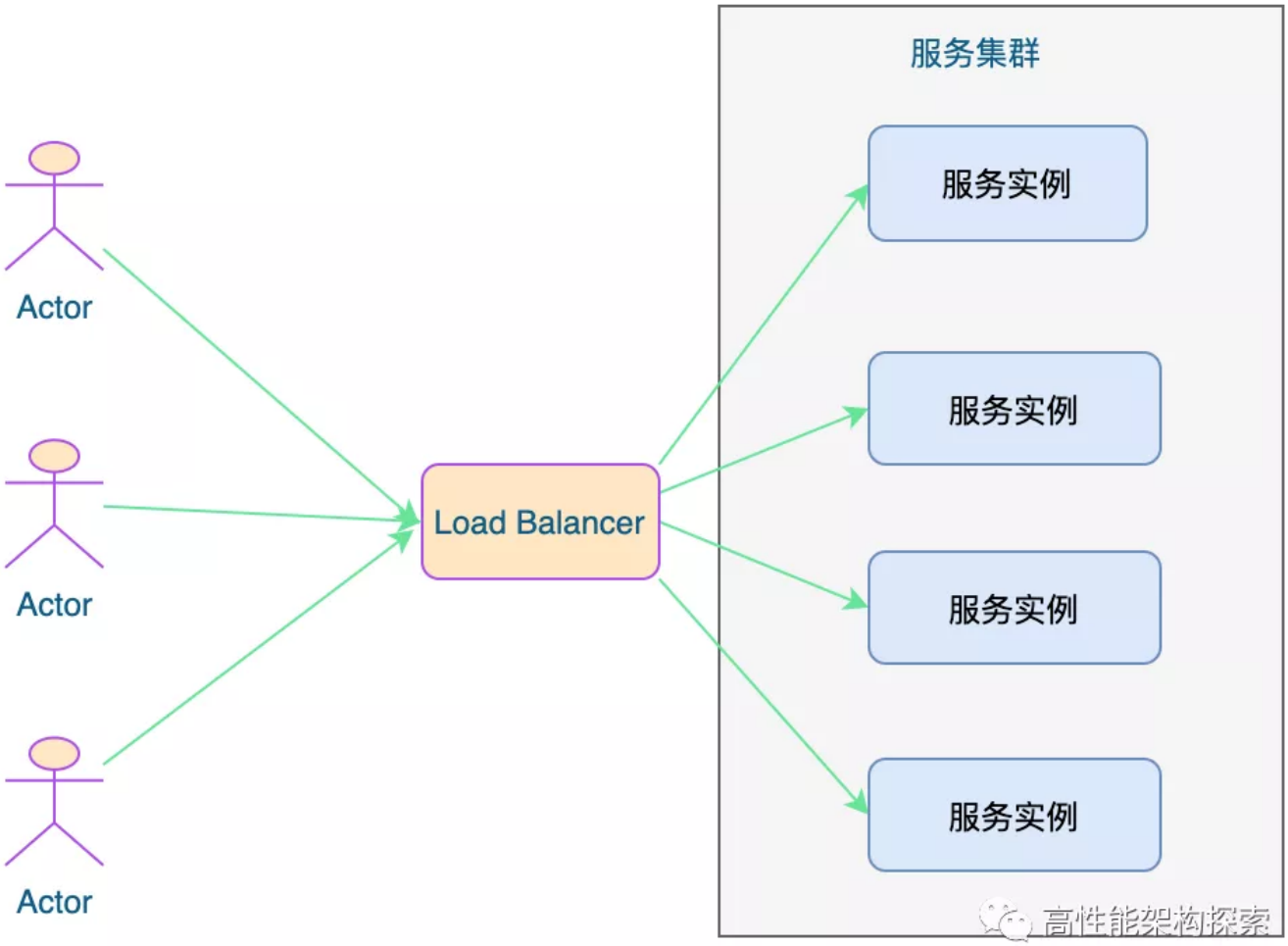
上面两种方式，我们称之为纵向扩展和横向扩展。

纵向扩展，是从单机的角度通过增加硬件处理能力，比如CPU处理能力，内存容量，磁盘等方面，实现服务器处理能力的提升，不能满足大型分布式系统（网站），大流量，高并发，海量数据的问题。

横向扩展，通过添加机器来满足大型网站服务的处理能力。比如：一台机器不能满足，则增加两台或者多台机器，共同承担访问压力。

3概念

> 负载均衡，英文名称为Load Balance，其含义就是指将负载（工作任务或者网络请求）进行平衡，分摊到多个操作单元(服务器或者组件)上进行运行。目的是尽量将网络流量 ***平均*** 发送到多个服务器上，以保证整个业务系统的高可用。



负载均衡

在互联网的早期，网络还不是很发达，流量相对较小，业务也比较简单，单台服务器或者实例就有可能满足访问需要。但如今在互联网发达的今天，流量请求动辄百亿、甚至上千亿，单台服务器或者实例已完全不能满足需求，这就有了集群。不论是为了实现高可用还是高性能，都需要用到多台机器来扩展服务能力，用户的请求不管连接到哪台服务器，都能得到相同的相应处理。

另一方面，如何构建和调度服务集群这事情，又必须对用户一侧保持足够的透明，即使请求背后是由一千台、一万台机器来共同响应的，也绝非用户所关心的事情，用户需记住的只有一个域名地址而已。调度后方的多台机器，以统一的接口对外提供服务，承担此职责的技术组件被称为 ***负载均衡***。

负载均衡主要有以下作用：

- 高并发。通过采取一定的算法策略，将流量尽可能的均匀发送给后端的实例，以此提高集群的并发处理能力。
- 伸缩性。根据网络流量的大小，增加或者减少后端服务器实例，由负载均衡设备进行控制，这样使得集群具有伸缩性。
- 高可用。负载均衡器通过算法或者其他性能数据来监控候选实例，当实例负载过高或者异常时，减少其流量请求或者直接跳过该实例，将请求发送个其他可用实例，这使得集群具有高可用的特性。
- 安全防护。有些负载均衡器提供了安全防护功能。如：黑白名单处理、防火墙等。

4分类

根据载体类型分类

从支持负载均衡的载体来看，可以将负载均衡分为两类：

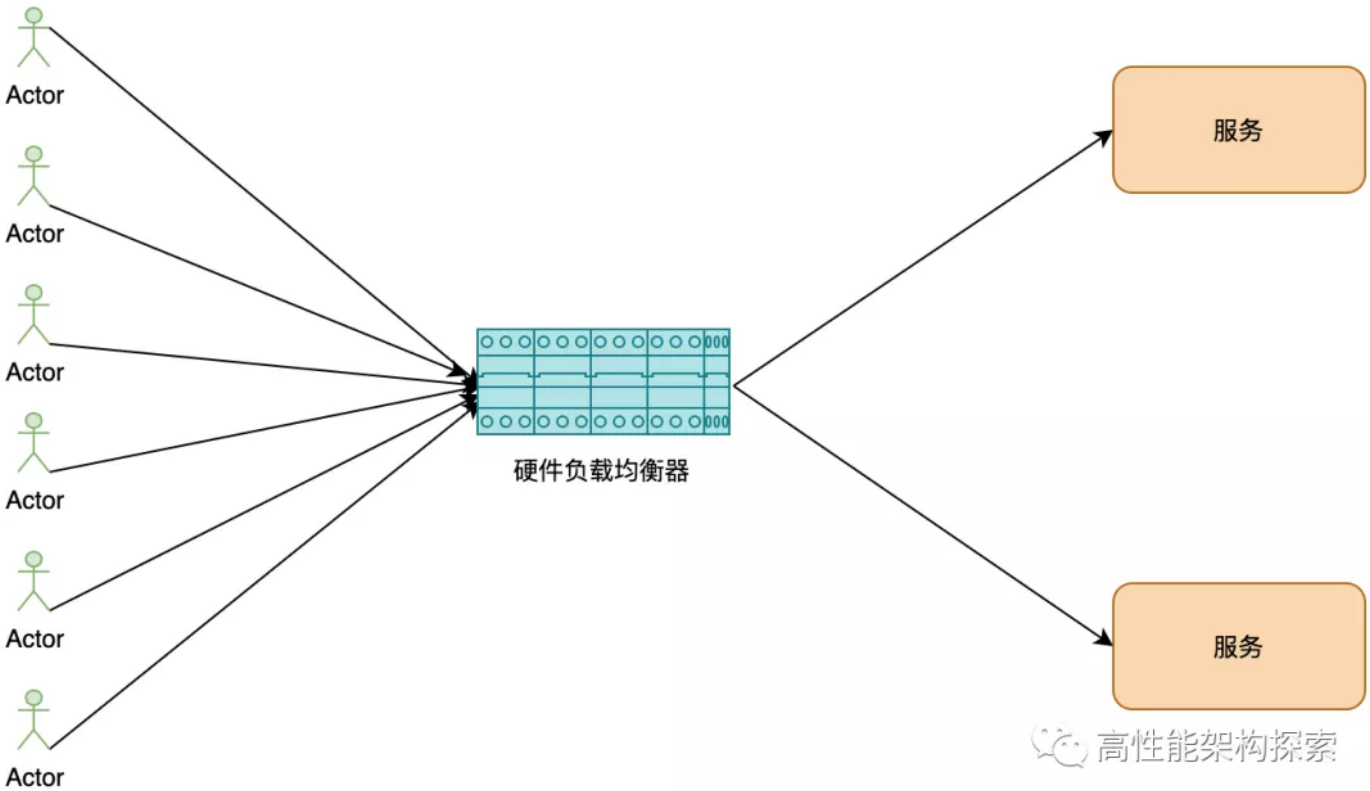
- 硬件负载均衡
- 软件负载均衡

硬件负载均衡

硬件负载平衡器是一种硬件设备，具有专门的操作系统。硬件负载平衡器位于传入流量和内部服务器之间，本质上充当“流量警察”。当用户访问网站或者使用app某个功能时，它们首先被发送到负载均衡器，然后负载均衡器根据一定的策略，将流量转发到后端不同的服务器。为确保最佳性能，硬件负载均衡器根据自定义规则分配流量，以免后端实例不堪重负。

传统上，硬件负载平衡器和应用服务器部署在本地数据中心，负载平衡器的数量取决于预期的峰值流量。负载均衡器通常成对部署，以防其中一个失败。

目前业界领先的两款硬件负载均衡器：F5和A10



硬件负载均衡

优点：

> 功能强大：支持全局负载均衡并提供较全面的、复杂的负载均衡算法。
>
> 性能强悍：硬件负载均衡由于是在专用处理器上运行，因此吞吐量大，可支持单机百万以上的并发。
>
> 安全性高：往往具备防火墙，防 DDos 攻击等安全功能。

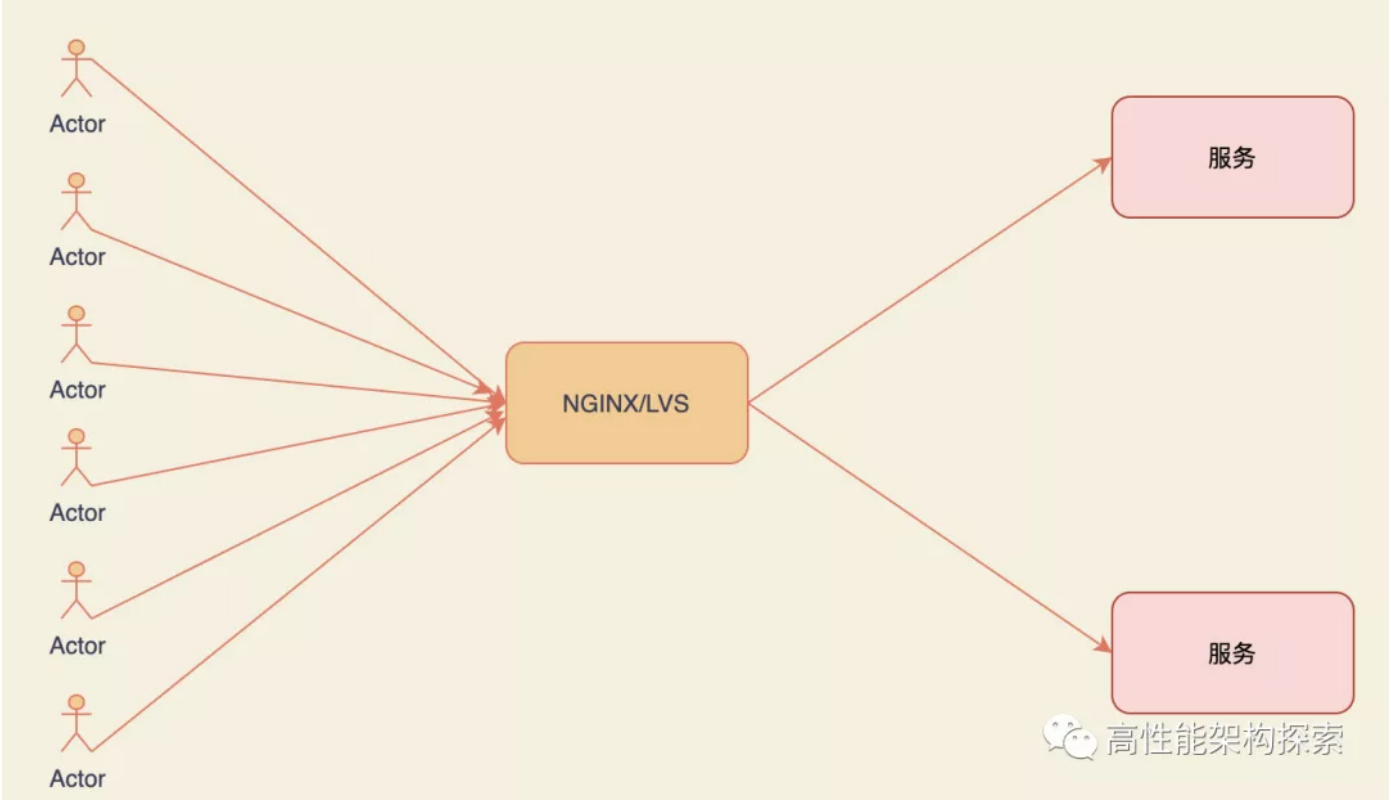
缺点

> 成本昂贵：购买和维护硬件负载均衡的成本都很高(：F5价格在15w~55w不等，A10价格在55w-100w不等)。
>
> 扩展性差：当访问量突增时，超过限度不能动态扩容。

软件负载均衡

软件负载均衡指的是在服务器的操作系统上安装负载均衡软件，从此服务器发出的请求经软件负载均衡算法路由到后端集群的某一台机器上。

常见负载均衡软件有：LVS、Nginx、Haproxy。



软件负载均衡

优点

> 扩展性好：适应动态变化，可以通过添加软件负载均衡实例，动态扩展到超出初始容量的能力。
>
> 成本低廉：软件负载均衡可以在任何标准物理设备上运行，降低了购买和运维的成本。

缺点

> 性能略差：相比于硬件负载均衡，软件负载均衡的性能要略低一些。

软硬件负载均衡器的区别

- 硬件负载平衡器与软件负载平衡器之间最明显的区别在于，硬件负载平衡器需要专有的机架堆叠硬件设备，而

软件负载平衡器只需安装在标准 x86 服务器或虚拟机上。网络负载平衡器硬件通常是过度配置的——换句话说，它们的大小能够处理偶尔的高峰流量负载。此外，每个硬件设备都必须与一个额外的设备配对以获得高可用性，以防其他负载均衡器出现故障。

- 硬件和软件负载平衡器之间的另一个关键区别在于扩展能力。随着网络流量的增长，数据中心必须提供足够的负载均衡器以满足峰值需求。对于许多企业来说，这意味着大多数负载均衡器在高峰流量时间（例如黑色星期五）之前一直处于空闲状态。
- 如果流量意外超出容量，最终用户体验会受到显着影响。另一方面，软件负载平衡器能够弹性扩展以满足需求。无论网络流量是低还是高，软件负载平衡器都可以简单地实时自动扩展，消除过度配置成本和对意外流量激增的担忧。
- 此外，硬件负载平衡器配置可能很复杂。基于软件定义原则构建的软件负载平衡器跨多个数据中心和混合/多云环境。事实上，硬件设备与云环境不兼容，而软件负载均衡器与裸机、虚拟、容器和云平台兼容。

根据地域范围分类

负载均衡从其应用的地理结构上分为本地负载均衡(Local Load Balance)和全局负载均衡(Global Load Balance，也叫地域负载均衡)。

地域负载均衡

本地负载均衡

> 本地负载均衡是指对本地的服务器群做负载均衡。

本地负载均衡针对本地范围的服务器群做负载均衡，本地负载均衡不需要花费高额成本购置高性能服务器，只需利用现有设备资源,就可有效避免服务器单点故障造成数据流量的损失，通常用来解决数据流量过大、网络负荷过重的问题。同时它拥有形式多样的均衡策略把数据流量合理均衡的分配到各台服务器。如果需要在现在服务器上升级扩充，不需改变现有网络结构、停止现有服务，仅需要在服务群中简单地添加一台新服务器。

本地负载均衡能有效地解决数据流量过大、网络负荷过重的问题，并且不需花费昂贵开支购置性能卓越的服务器，充分利用现有设备，避免服务器单点故障造成数据流量的损失。

其有灵活多样的均衡策略把数据流量合理地分配给服务器群内的服务器共同负担。即使是再给现有服务器扩充升级，也只是简单地增加一个新的服务器到服务群中，而不需改变现有网络结构、停止现有的服务。

全局负载均衡

> 全局负载均衡是指对分别放置在不同的地理位置、有不同网络结构的服务器群间作负载均衡。

全局负载均衡主要用于在一个多区域拥有自己服务器的站点，为了使全球用户只以一个IP地址或域名就能访问到离自己最近的服务器，从而获得最快的访问速度，也可用于子公司分散站点分布广的大公司通过Intranet(企业内部互联网)来达到资源统一合理分配的目的。

全局负载均衡，目前实现方式有以下几种：

- 通过运营商线路调度：这个主要是指国内，由于特殊原因国内不同运营商互联互通存在很大问题，比如联通用户访问电信机房服务器延迟很大，甚至有可能无法访问的情况。假如您的业务部署在不同运营商机房，可以通过运营商线路解析来实现调度，联通线路用户域名解析到联通机房IP，电信线路用户域名解析电信机房IP，这样保证不同用户访问最佳的服务器。
- 通过地域线路调度：
    - 我们都知道，网站服务器越近，访问速度越快，比如天津用户访问北京服务器会比广州服务器快很多。假如您的业务部署在华北，华南两个Region，可以通过地域线路解析，设置华北，东北，西北，华中用户访问域名解析到北京服务器IP，华东，华南，西南用户访问域名解析到广州服务器IP，这样用户访问

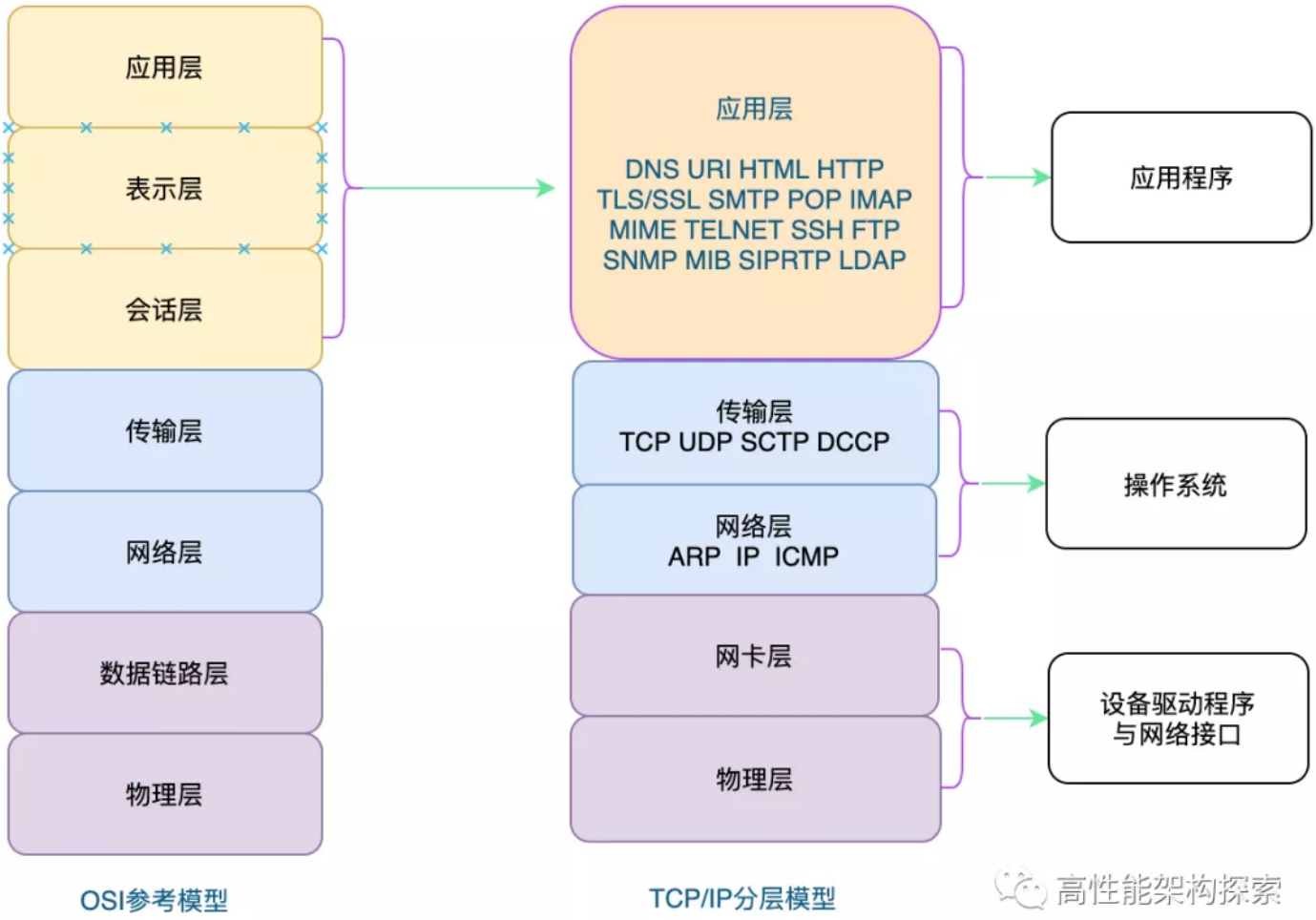
离自己最近的服务器可以提升访问体验。
    - 假如您的业务是面向全球的，国内部署有业务，海外也部署有业务，可以选择中国用户访问域名解析到国内服务器，海外用户访问域名解析到海外服务器。当然海外的还可以细分，比如选择亚太--新加坡的用户等，可以具体到洲，国家。

- 权重轮询：比如一个域名解析到多个IP，可以根据不同IP服务器的配置，业务情况设置解析比重，比如2:1或者1:1等等。
- 健康检查，故障转移：可以创建监控任务实时监控后端服务器IP的健康状态，如果发现后端服务器异常，可以把解析流量切换到其他正常的服务器或者备用服务器，保证业务不会中断。

CDN的全称是Content Delivery Network，即内容分发网络。其就是采用的全局负载均衡。假如我们将图片存储在CDN上，且该CDN所在厂家在北京、杭州均有服务器。那么：

- 当天津的用户需要下载该图片的时候，会自动将流量请求转发至距离其最近的CDN服务器，也就是北京
- 当安徽的用户需要下载图片的时候，就会将流量请求转发至杭州。

根据OSI网络模型分类



OSI是一个开放性的通信系统互连参考模型，如上图所示。在OSI参考模型中，分别有：

- 应用层
- 表示层
- 会话层
- 传输层
- 网络层

- 数据链路层
- 物理层

从上图可以看出：

> TELNET、HTTP、FTP、NFS、SMTP、DNS等属于第七层应用层的概念。
>
> TCP、UDP、SPX等属于第四层传输层的概念。
>
> IP、IPX等属于第三层网络层的概念。
>
> ATM、FDDI等属于第二层数据链路层的概念。

根据负载均衡技术实现在OSI七层模型的不同层次，我们给负载均衡分类：

- 七层负载均衡：工作在应用层的负载均衡称
- 四层负载均衡：工作在传输层的负载均衡称
- 三层负载均衡：工作在网络层的负载均衡，
- 二层负载均衡：工作在数据链路层的负载均衡。

其中最常用的是四层和七层负载均衡。

下面我们将从OSI模型从下往上的顺序，来详细讲解上述几种负载均衡。

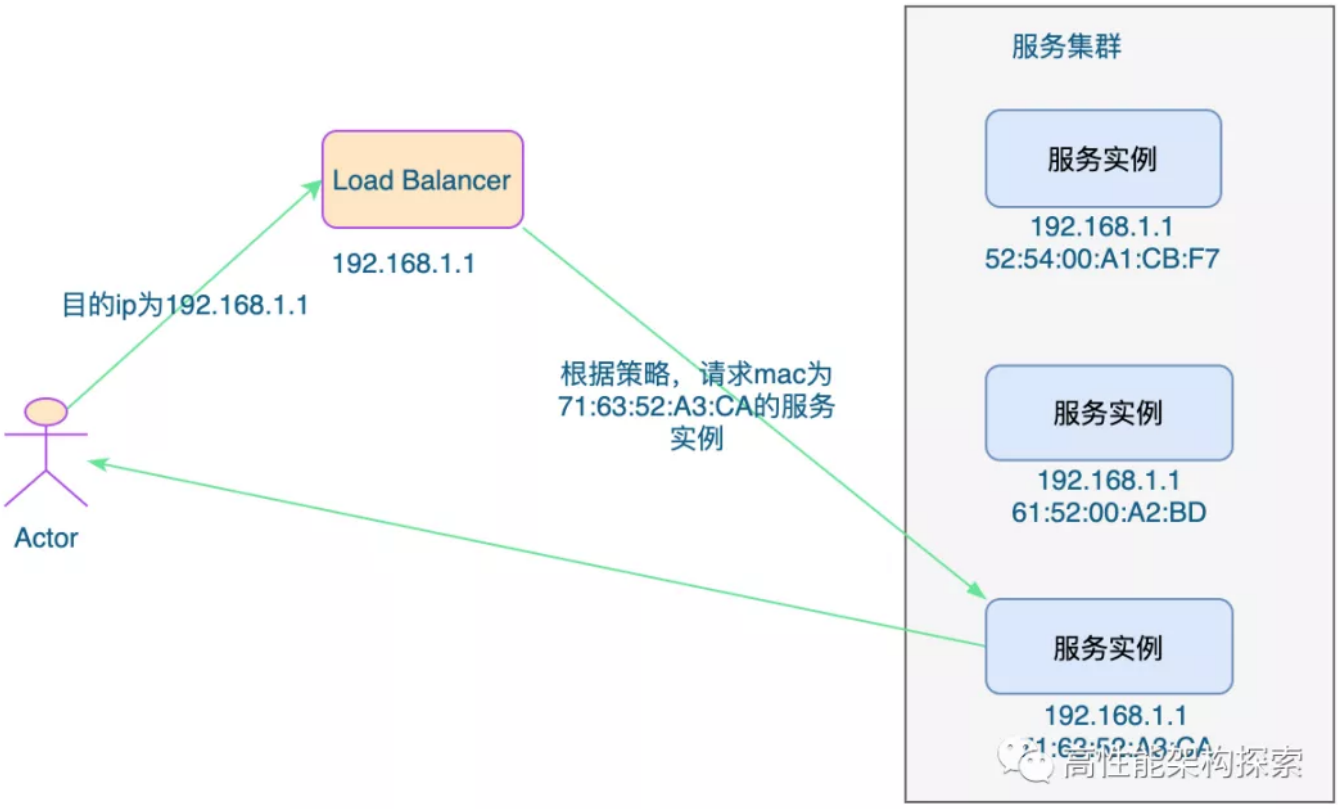
二层负载均衡

> 工作在数据链路层的负载均衡称之为二层负载均衡(又称为数据链路层负载均衡)，通过在通信协议的数据链路层修改mac地址进行负载均衡。

二层负载均衡是基于数据链路层的负载均衡，即让负载均衡服务器和业务服务器绑定同一个虚拟IP（即VIP），客户端直接通过这个VIP进行请求集群。集群中不同的机器采用相同IP地址，但是机器的MAC地址不一样。当负载均衡服务器接受到请求之后，通过改写报文的目标MAC地址的方式将请求转发到目标机器实现负载均衡。

数据链路层负载均衡所做的工作，是修改请求的数据帧中的 MAC 目标地址，让用户原本是发送给负载均衡器的请求的数据帧，被二层交换机根据新的 MAC 目标地址转发到服务器集群中对应的服务器（真实服务器）的网卡上，这样真实服务器就获得了一个原本目标并不是发送给它的数据帧。

为了便于理解，我们假设负载均衡器所在的ip地址为192.168.1.1，后端服务实例的mac地址分别为52:54:00:A1:CB:F7，61:52:00:A2:BD, 71:63:52:A3:CA。如下图所示：



二层负载均衡

在上图中，用户的请求首先到达ip为192.168.1.1的二层负载均衡器，然后二层负载均衡器通过采取一定的策略，选中了mac地址为71:63:52:A3:CA，然后将流量转发至该服务实例。

需要注意的是,上述只有请求经过负载均衡器，而服务的响应无须从负载均衡器原路返回的工作模式，整个请求、转发、响应的链路形成一个“三角关系”，所以这种负载均衡模式也常被很形象地称为“三角传输模式”，也有叫“单臂模式”或者“直接路由”。

二层负载均衡器直接改写目标 MAC 地址的工作原理决定了它与真实的服务器的通信必须是二层可达的，通俗地说就是必须位于同一个子网当中，无法跨 VLAN。优势（效率高）和劣势（不能跨子网）共同决定了数据链路层负载均衡最适合用来做数据中心的第一级均衡设备，用来连接其他的下级负载均衡器。

三层负载均衡

> 三层负载均衡是基于网络层的负载均衡，因此又叫网络层负载均衡。通俗的说就是按照不同机器不同IP地址进行转发请求到不同的机器上。

根据 OSI 七层模型，在第三层网络层传输的单位是分组数据包，这是一种在分组交换网络中传输的结构化数据单位。以IP协议为例，一个IP 数据包由 Headers 和 Payload 两部分组成， Headers 长度最大为60Bytes，其中包括了20Bytes的固定数据和最长不超过40Bytes 的可选的额外设置组成。

三层负载均衡服务器对外依然提供一个VIP（虚IP），但是集群中不同的机器采用不同的IP地址。当负载均衡服务器接受到请求之后，根据不同的负载均衡算法，通过IP将请求转发至不同的真实服务器。

学过计算机网络的都知道，在IP分组的数据报header中有 *源IP* 和 *目标IP*。源IP和目标IP代表分组交换中数据是从哪台机器到哪台机器的，那么，我们可以采用跟修改二层负载均衡中MAC地址的方式一样，直接修改目标IP，以达到数据转发的目的。

修改目标IP的方式有两种：1、原有的数据包保持不变，生成一个新的数据包，原数据包的Header和Payload作为新数据包的Payload，在这个新数据包的 Headers 中写入真实服务器的 IP 作为目标地址，然后把它发送出去。

真实服务器收到数据包后，必须在接收入口处设计一个针对性的拆包机制，把由负载均衡器自动添加的那层 Headers 扔掉，还原出原来的数据包来进行使用。这样，真实服务器就同样拿到了一个原本不是发给它（目标 IP 不是它）的数据包，达到了流量转发的目的。这种数据传输方式叫做 *IP隧道* 传输。

尽管因为要封装新的数据包，IP 隧道的转发模式比起直接路由模式效率会有所下降，但由于并没有修改原有数据包中的任何信息，所以 IP 隧道的转发模式仍然具备三角传输的特性，即负载均衡器转发来的请求，可以由真实服务器去直接应答，无须在经过均衡器原路返回。而且由于 IP 隧道工作在网络层，所以可以跨越 VLAN，因此摆脱了直接路由模式中网络侧的约束。

此模式从请求到响应如下图所示：

IP隧道模式负载均衡

优点：

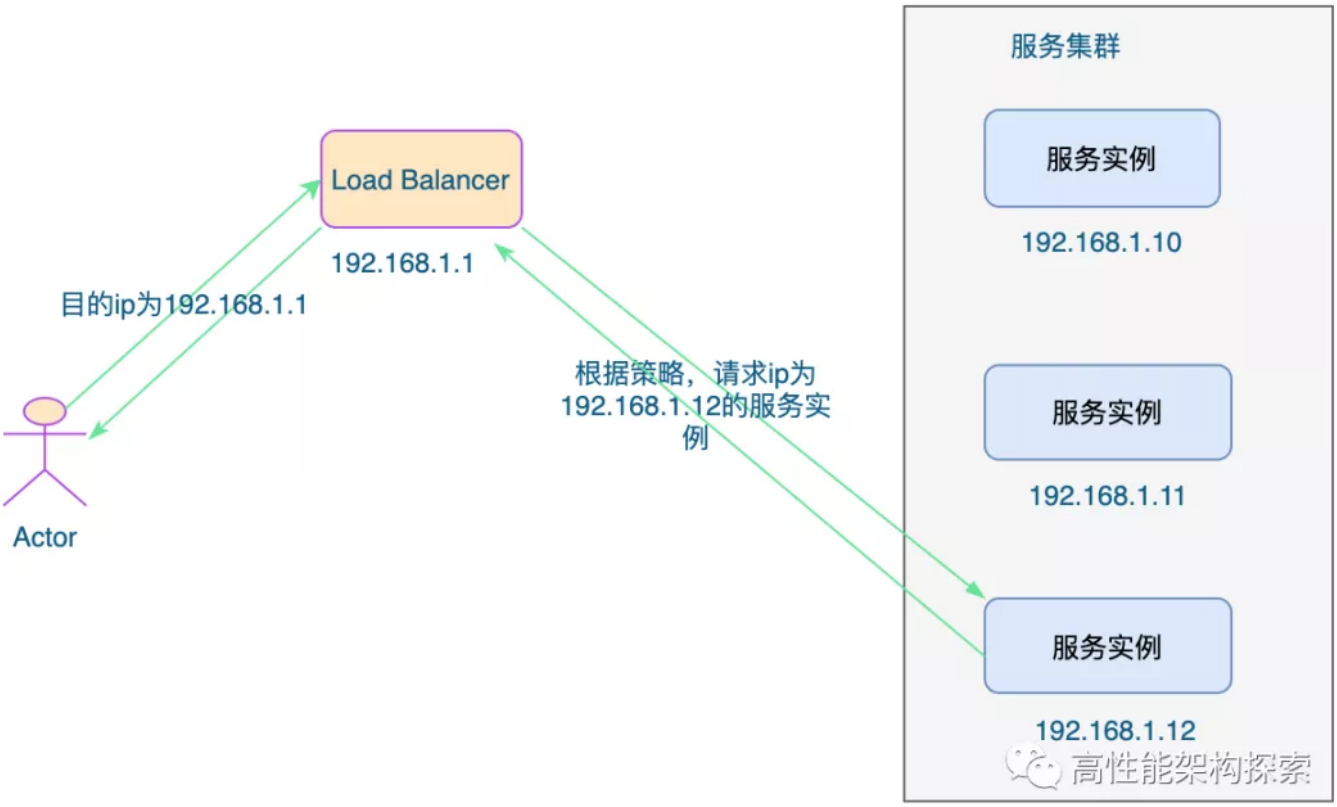
- 可以跨越 VLAN 缺点：
- 要求真实服务器必须支持IP隧道协议，也就是说服务器需要自己会拆包
- 必须通过专门的配置，必须保证所有的真实服务器与均衡器有着相同的虚拟 IP 地址，因为回复该数据包时，需要使用这个虚拟 IP 作为响应数据包的源地址，这样客户端收到这个数据包时才能正确解析。

基于以上原因，就有了第二种修改方式。2、改变目标数据包。

直接把数据包 Headers 中的目标地址改为真实服务器地址，修改后原本由用户发给均衡器的数据包，也会被三层交换机转发送到真实服务器的网卡上，而且因为没有经过 IP 隧道的额外包装，也就无须再拆包了。

因为这种模式是通过修改目标 IP 地址才到达真实服务器的，如果真实服务器直接将应答包返回客户端的话，这个应答数据包的源 IP 是真实服务器的 IP，也即均衡器修改以后的 IP 地址，客户端不可能认识该 IP，自然就无法再正常处理这个应答了。因此，只能让应答流量继续回到负载均衡，由负载均衡把应答包的源 IP 改回自己的 IP，再发给客户端，这样才能保证客户端与真实服务器之间的正常通信。

这种修改目标IP的方式叫NAT模式，这种通过修改目标IP的方式达到负载均衡目的的方式叫做NAT负载均衡。如下图所示：



NAT模式负载均衡

四层负载均衡

> 所谓四层负载均衡，也就是主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

由于四层负载均衡是作用在传输层，因此，我们就以常见的TCP进行举例。

负载均衡设备在接收到第一个来自客户端的SYN 请求时，即通过上述方式选择一个最佳的服务器，并对报文中目标IP地址进行修改(改为后端服务器IP），直接转发给该服务器。TCP的连接建立，即三次握手是客户端和服务器直接建立的，负载均衡设备只是起到一个类似路由器的转发动作。在某些部署情况下，为保证服务器回包可以正确返回给负载均衡设备，在转发报文的同时可能还会对报文原来的源地址进行修改。

四层负载均衡

四层负载均衡主要是基于tcp协议报文，可以做任何基于tcp/ip协议的软件的负载均衡，比如Haproxy、LVS等。

七层负载均衡

> 所谓七层负载均衡，也称为“内容交换”，也就是主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

应用层协议较多，常用http、radius、dns等。七层负载就可以基于这些协议来负载。

我们仍然以TCP为例。负载均衡设备如果要根据真正的应用层内容再选择服务器，只能先代理最终的服务器和客户端建立连接(三次握手)后，才可能接受到客户端发送的真正应用层内容的报文，然后再根据该报文中的特定字段，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。负载均衡设备在这种情况下，更类似于一个代理服务器。负载均衡和前端的客户端以及后端的服务器会分别建立TCP连接。所以从这个技术原理上来看，七层负载均衡明显的对负载均衡设备的要求更高，处理七层的能力也必然会低于四层模式的部署方式。

> 七层负载均衡器会与客户端 以及 后端的服务实例分别建立连接

七层负载均衡

七层负载均衡基本都是基于http协议的，适用于web服务器的负载均衡，比如Nginx等。

对比(四层和七层)

- 智能性
    - 七层负载均衡由于具备OIS七层的所有功能，所以在处理用户需求上能更加灵活，从理论上讲，七层模型能对用户的所有跟服务端的请求进行修改。例如对文件header添加信息，根据不同的文件类型进行分类转发。
    - 四层模型仅支持基于网络层的需求转发，不能修改用户请求的内容。
- 安全性
    - 七层负载均衡由于具有OSI模型的全部功能，能更容易抵御来自网络的攻击
    - 四层模型从原理上讲，会直接将用户的请求转发给后端节点，无法直接抵御网络攻击。
- 复杂度
    - 四层模型一般比较简单的架构，容易管理，容易定位问题
    - 七层模型架构比较复杂，通常也需要考虑结合四层模型的混用情况，出现问题定位比较复杂。
- 效率比
    - 四层模型基于更底层的设置，通常效率更高，但应用范围有限
    - 七层模型需要更多的资源损耗，在理论上讲比四层模型有更强的功能，现在的实现更多是基于http应用。

6算法与实现

常用的负载均衡算法分为以下两类：

- 静态负载均衡
- 动态负载均衡

常见的静态均衡算法：轮询法、随机法、源地址哈希法、一致性哈希法、加权轮询法、加权随机法。

常见的动态负载均衡算法：最小连接数法、最快响应速度法。

随机法(Random)

> 将请求随机分配到各个节点。由概率统计理论得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配，也就是轮询的结果。

随机策略会导致配置较低的机器Down机，从而可能引起雪崩，一般采用随机算法时建议后端集群机器配置最好同等的，随机策略的性能取决于随机算法的性能。

- 优点：简单高效，易于水平扩展，每个节点满足字面意义上的均衡；
- 缺点：没有考虑机器的性能问题，根据木桶最短木板理论，集群性能瓶颈更多的会受性能差的服务器影响。

随机法

实现：

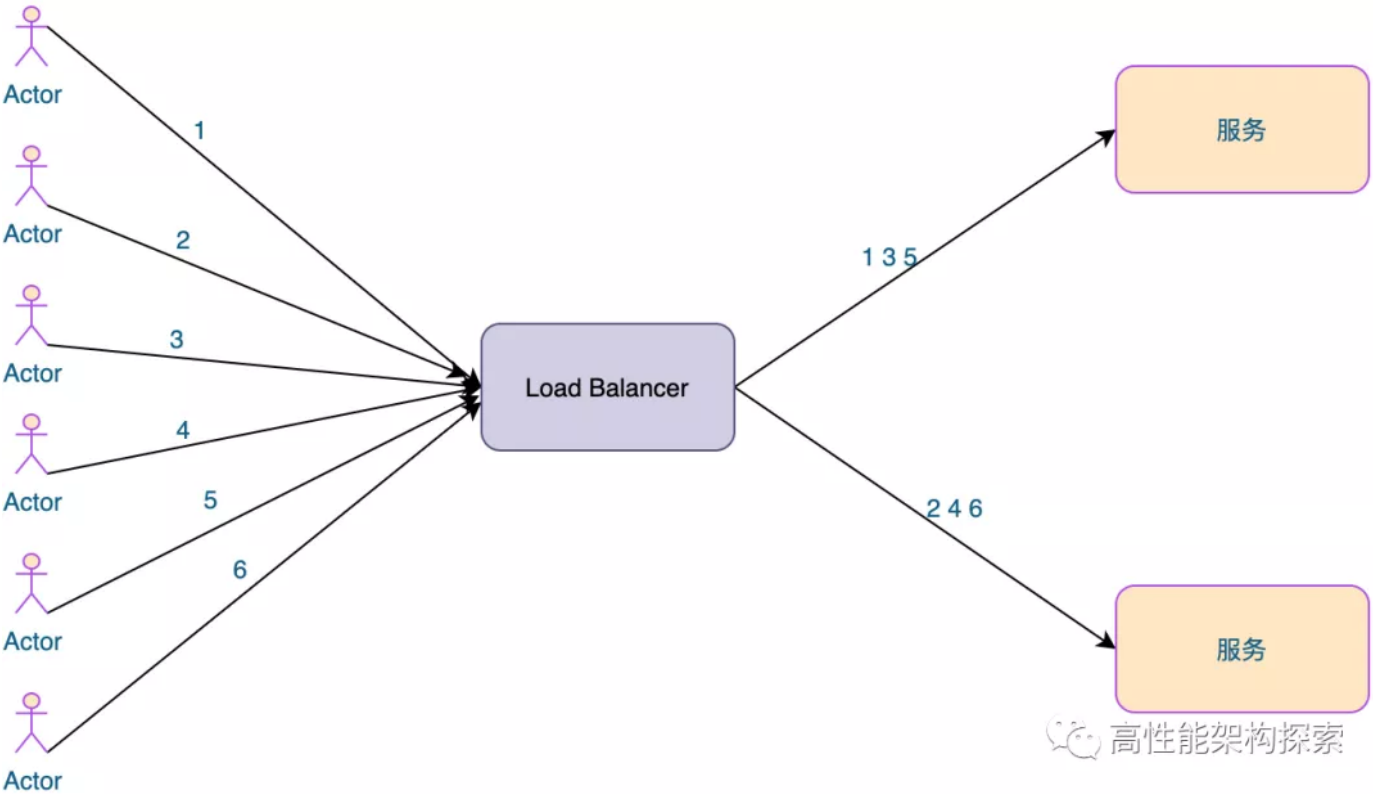
```
std::string Select(const std::vector<int> &ips) {
  size_t size = ips.size();
  if (size == 0) {
    return "";
  }

  return ips[random() % size];
}
```

轮询法(Round Robin)

> 每一次来自网络的请求轮流分配给内部中的服务器，从1至N然后重新开始。此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。

假设10台机器，从0-9，请求来临时从0号机器开始，后续每来一次请求对编号加1，这样一直循环，上面的随机策略其实最后就变成轮询了，这两种策略都不关心机器的负载和运行情况，而且对变量操作会引入锁操作，性能也会下会下降。

- 优点：简单高效，易于水平扩展，每个节点满足字面意义上的均衡；
- 缺点：没有考虑机器的性能问题，根据木桶最短木板理论，集群性能瓶颈更多的会受性能差的服务器影响。



轮询法

代码实现：

```
static int idx = 0;
std::string Select(const std::vector<int> &ips) {
  size_t size = ips.size();
  if (size == 0) {
    return "";
  }

  if (idx == ips.size()) {
    idx = 0;
  }

  return ips[idx++];
}
```

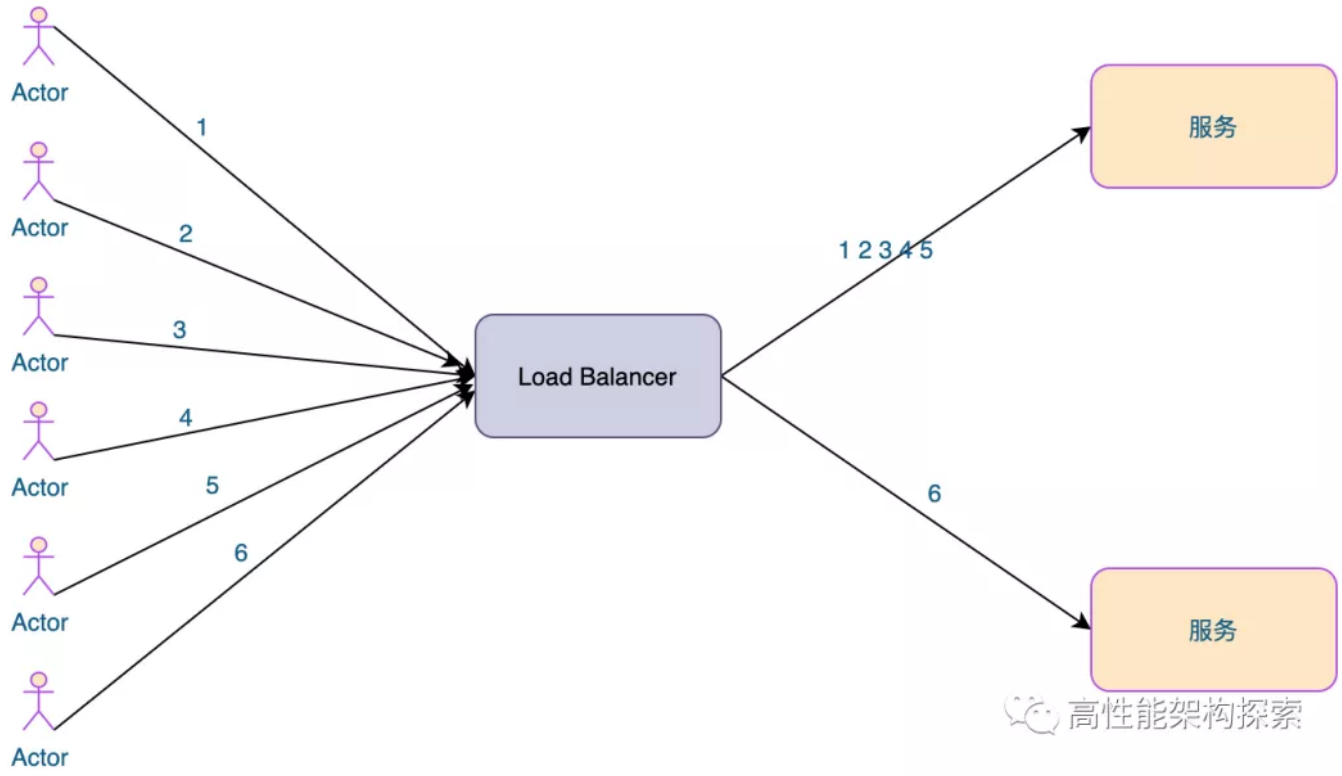
加权轮询法(Weighted Round Robin)

> 不同的后端服务器可能机器的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；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

假设后端有3台服务器，分别为a b c，现在在负载均衡器中配置a服务器的权重为7，b服务的权重为2，c服务的权重为1。当来了10次请求的时候，其中有7次请求a，2次请求b，1次请求c。即最终结果是

```
aaaaaaabbc
```

- 优点：可以将不同机器的性能问题纳入到考量范围，集群性能最优最大化；
- 缺点：生产环境复杂多变，服务器抗压能力也无法精确估算，静态算法导致无法实时动态调整节点权重，只能粗糙优化。



加权轮询

加权随机法(Weighted Random)

> 与加权轮询法一样，加权随机法也根据服务器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

在之前的文章权重随机分配器我们有详细讲过各种实现方案，此处我们不再赘述，从里面摘抄了一种实现方案作为本方案的实现。

加权随机

- 优点：可以将不同机器的性能问题纳入到考量范围，集群性能最优最大化；
- 缺点：生产环境复杂多变，服务器抗压能力也无法精确估算，静态算法导致无法实时动态调整节点权重，只能粗糙优化。

代码实现

```
srtuct Item {
  std::string ip;
  int weight;
};
std::string select(const std::vector<Item> &items) {
  int sum = 0;
  for (auto elem : items) {
    sum += elem.weight;
  }

  int rd = rand() % sum;
```

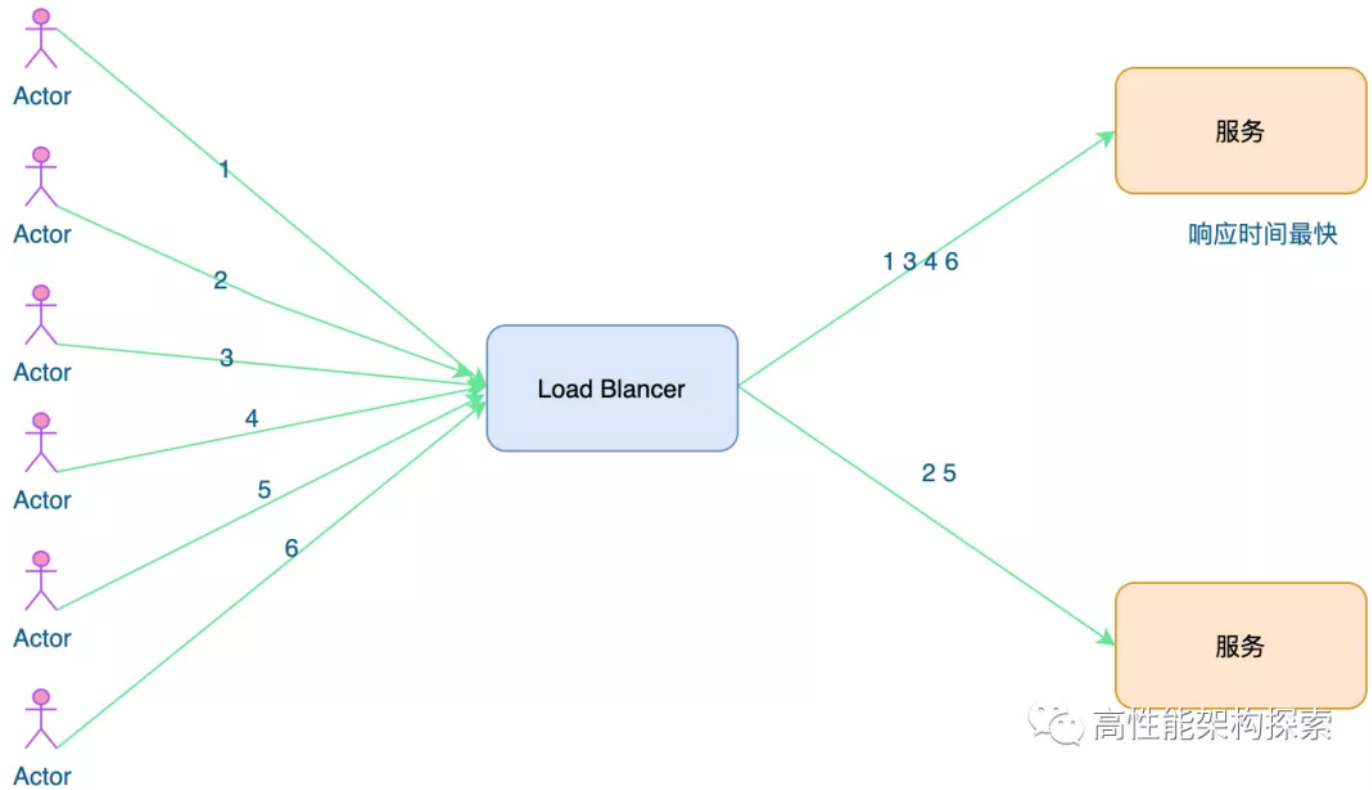
```
  int s = 0;
  std::string res;
  for (auto elem : items) {
    s += elem.weight;
    if (s >= rd) {
      res = elem.ip;
      break;
    }
  }
  return res;
}
```

最快响应速度法(Response Time)

> 根据请求的响应时间，来动态调整每个节点的权重，将响应速度快的服务节点分配更多的请求，响应速度慢的服务节点分配更少的请求

负载均衡设备对内部各服务器发出一个探测请求（例如Ping），然后根据内部中各服务器对探测请求的最快响应时间来决定哪一台服务器来响应客户端的服务请求。此种均衡算法能较好的反映服务器的当前运行状态，但这最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务器间的最快响应时间。

- 优点：动态，实时变化，控制的粒度更细，更灵敏；
- 缺点：复杂度更高，每次需要计算请求的响应速度；



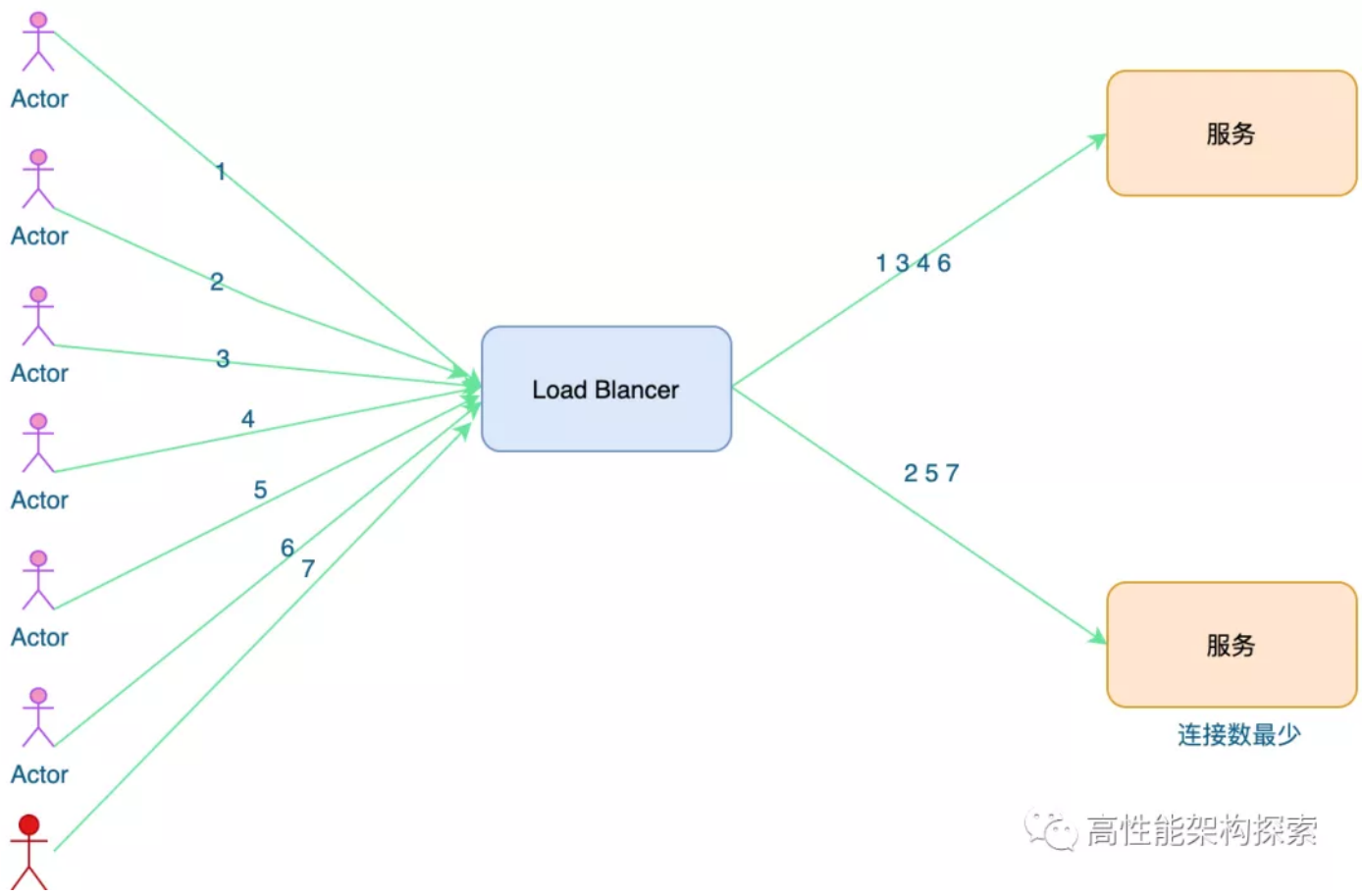
最快响应速度

最少连接数法(Least Connections)

> 将请求分发到连接数/请求数最少的候选服务器，已达到负载均衡的目的

客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间加长，如果采用简单的轮循或随机均衡算法，每一台服务器上的连接进程可能会产生极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如FTP。

- 优点：动态，根据节点状况实时变化
- 缺点：提高了复杂度，每次连接断开需要进行计数



最少连接数

源地址哈希法(Source Hashing)

> 根据请求源 IP，通过哈希计算得到一个数值，用该数值在候选服务器列表的进行取模运算，得到的结果便是选中的服务器。

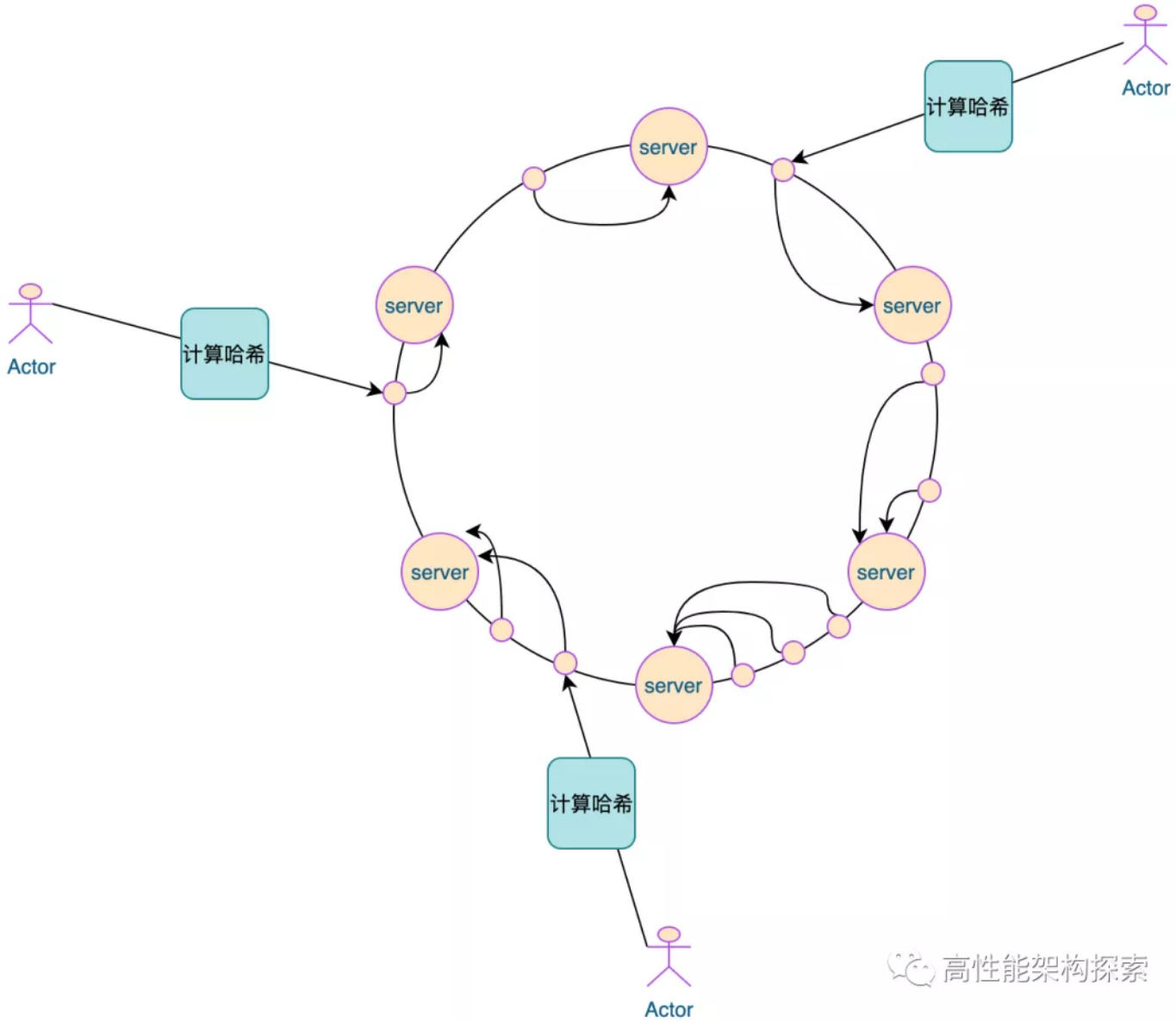
能够让同一客户端的请求或者同一用户的请求总是请求在后端同一台机器上，这种算法根据客户端IP求出Hash值然后对端集群总数求余得到值就是服务器集合的下标，一般这种算法用于缓存命中，或者同一会话请求等，但这种算法也有一定的缺点，某一用户访问量(黑产)非常高时可能造成服务端压力过大或者后端服务Down掉，那么客户端就会无法访问，所以也需要一定的降级策略。

- 优点：将来自同一IP地址的请求，同一会话期内，转发到相同的服务器；实现会话粘滞
- 缺点：目标服务器宕机后，会话会丢失

源地址哈希

一致性哈希(Consistency hash)

一些场景希望同样的请求尽量落到一台机器上，比如访问缓存集群时，我们往往希望同一种请求能落到同一个后端上，以充分利用其上已有的缓存，不同的机器承载不同的稳定请求量（也可以理解为固定批用户的请求）。而不是随机地散落到所有机器上，那样的话会迫使所有机器缓存所有的内容，最终由于存不下形成颠簸而表现糟糕。我们都知道hash能满足这个要求，比如当有n台服务器时，输入x总是会发送到第hash(x) % n台服务器上。但当服务器变为m台时，hash(x) % n和hash(x) % m很可能都不相等，这会使得几乎所有请求的发送目的地都发生变化，如果目的地是缓存服务，所有缓存将失效，继而对原本被缓存遮挡的数据库或计算服务造成请求风暴，触发雪崩。一致性哈希是一种特殊的哈希算法，在增加服务器时，发向每个老节点的请求中只会有一部分转向新节点，从而实现平滑的迁移。



一致性哈希

优点：

- 平衡性: 每个节点被选到的概率是O(1/n)。
- 单调性: 当新节点加入时， 不会有请求在老节点间移动， 只会从老节点移动到新节点。当有节点被删除时，也不会影响落在别的节点上的请求。
- 分散性: 当上游的机器看到不同的下游列表时(在上线时及不稳定的网络中比较常见), 同一个请求尽量映射到少量的节点中。

- 负载: 当上游的机器看到不同的下游列表的时候， 保证每台下游分到的请求数量尽量一致。

缺点：

- 在机器数量较少的时候，区间大小会不平衡。
- 当一台机器故障的时候，它的压力会完全转移到另外一台机器， 可能无法承载。