# 负载均衡

面对大量用户访问、高并发请求，海量数据，可以使用高性能的服务器、大型数据库，存储设备，高性能Web服务器，采用高效率的编程语言比如(Go,Scala)等，当单机容量达到极限时，我们需要考虑业务拆分和分布式部署，来解决大型网站访问量大，并发量高，海量数据的问题。

从单机网站到分布式网站，很重要的区别是业务拆分和分布式部署，将应用拆分后，部署到不同的机器上，实现大规模分布式系统。分布式和业务拆分解决了，从集中到分布的问题，但是每个部署的独立业务还存在单点的问题和访问统一入口问题，为解决单点故障，我们可以采取冗余的方式。将相同的应用部署到多台机器上。解决访问统一入口问题，我们可以在集群前面增加负载均衡设备，实现流量分发。

负载均衡（Load Balance），意思是将负载（工作任务，访问请求）进行平衡、分摊到多个操作单元（服务器，组件）上进行执行。是解决高性能，单点故障（高可用），扩展性（水平伸缩）的终极解决方案。

## 负载均衡原理

系统的扩展可分为纵向（垂直）扩展和横向（水平）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。因此需要采用横向扩展的方式，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。这就是典型的集群和负载均衡架构

* 应用集群：将同一应用部署到多台机器上，组成处理集群，接收负载均衡设备分发的请求，进行处理，并返回相应数据。
* 负载均衡设备：将用户访问的请求，根据负载均衡算法，分发到集群中的一台处理服务器。

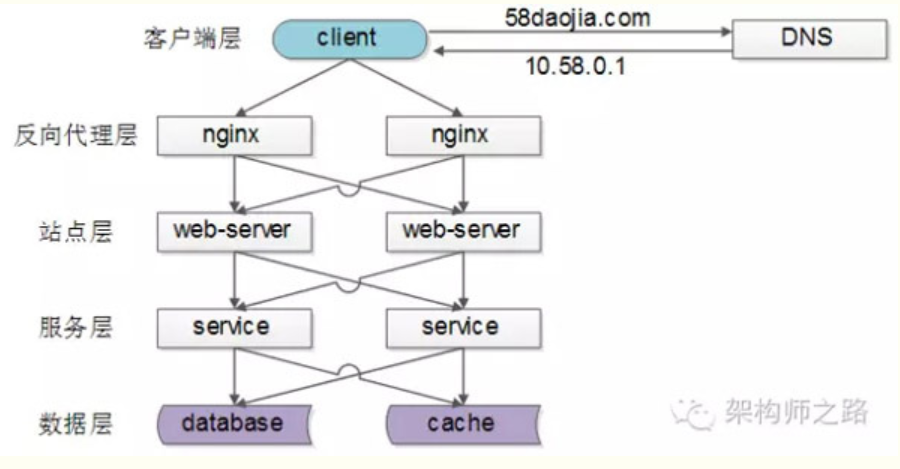
 负载均衡的作用（解决的问题）：

1.解决并发压力，提高应用处理性能（增加吞吐量，加强网络处理能力）；

2.提供故障转移，实现高可用；

3.通过添加或减少服务器数量，提供网站伸缩性（扩展性）；

4.安全防护；（负载均衡设备上做一些过滤，黑白名单等处理）



常见互联网分布式架构如上，分为客户端层、反向代理nginx层、站点层、服务层、数据层。可以看到，每一个下游都有多个上游调用，只需要做到，**每一个上游都均匀访问每一个下游**，就能实现“将请求/数据【均匀】分摊到多个操作单元上执行”。

　　(1)【客户端层】到【反向代理层】的负载均衡，是通过“DNS轮询”实现的  
　　(2)【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的  
　　(3)【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的  
　　(4)【数据层】的负载均衡，要考虑“数据的均衡”与“请求的均衡”两个点，常见的方式有“按照范围水平切分”与“hash水平切分”。

## 负载均衡算法

　 当我们提到一个负载均衡算法，或者具体的应用场景时，应该考虑以下问题

　　第一，是否意识到不同节点的服务能力是不一样的，比如CPU、内存、网络、地理位置

　　第二，是否意识到节点的服务能力是动态变化的，高配的机器也有可能由于一些突发原因导致处理速度变得很慢

　　第三，是否考虑将同一个客户端，或者说同样的请求分发到同一个处理节点，这对于“有状态”的服务非常重要，比如session，比如分布式存储

第四，谁来负责负载均衡，即谁充当负载均衡器（load balancer），balancer本身是否会成为瓶颈

### 轮询算法（round-robin）

轮询算法按顺序把每个新的连接请求分配给下一个服务器，最终把所有请求平分给所有的服务器。

   优点：绝对公平

   缺点：无法根据服务器性能去分配，无法合理利用服务器资源。

### 加权轮询算法（weight round-robin）

权轮训算法就是在轮训算法的基础上，考虑到机器的差异性，分配给机器不同的权重，能者多劳。 该算法中，每个机器接受的连接数量是按权重比例分配的。这是对普通轮询算法的改进，比如你可以设定：第三台机器的处理能力是第一台机器的两倍，那么负载均衡器会把两倍的连接数量分配给第3台机器。加权轮询分为：简单的轮询、平滑的轮询。

### 随机算法（random）

通过系统随机函数，根据后端服务器列表的大小值来随机选择其中一台进行访问。由概率统计理论可以得知，随着调用量的增大，其实际效果越来越接近于平均分配流量到每一台后端服务器，也就是轮询的效果。**基于概率统计的理论，吞吐量越大，随机算法的效果越接近于轮询算法的效果**。

### 加权随机算法

与加权轮询法类似，加权随机法也是根据后端服务器不同的配置和负载情况来配置不同的权重。不同的是，它是按照权重来随机选择服务器的，而不是顺序。

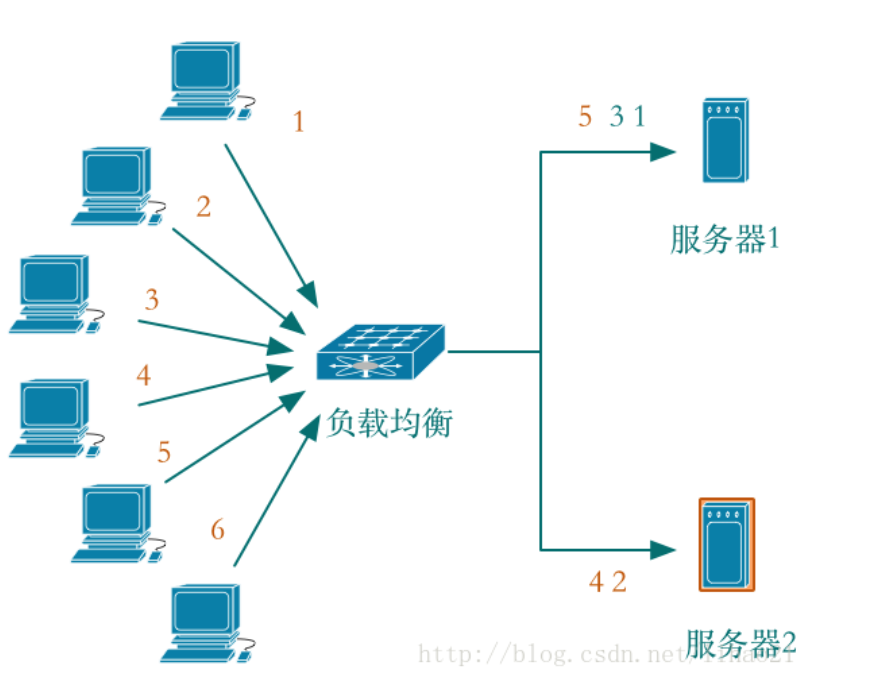
### 哈希法

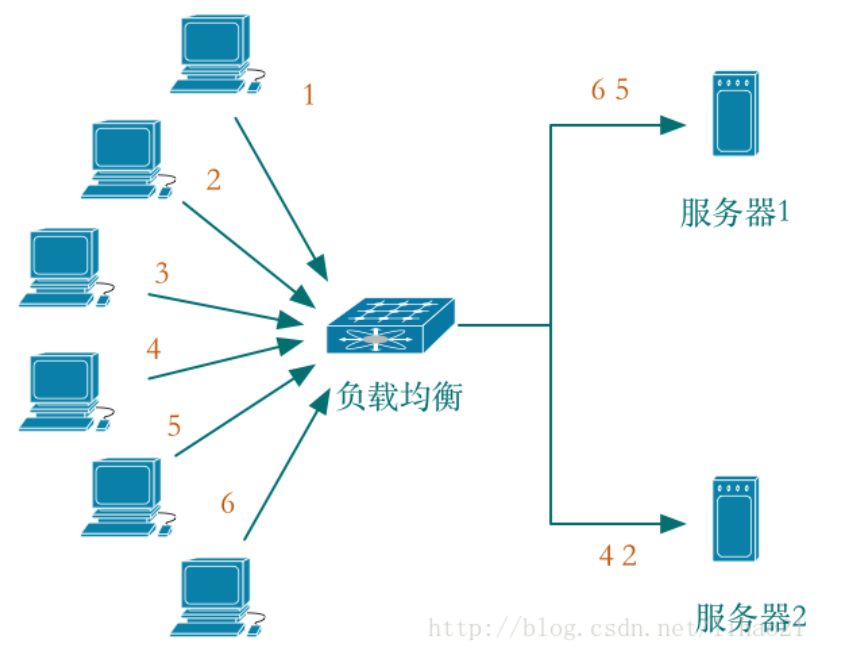
通过客户端的ip也就是remoteIp，取得它的Hash值，对服务器列表的大小取模，结果便是选用的服务器在服务器列表中的索引值。

源地址哈希法的优点在于：保证了相同客户端IP地址将会被哈希到同一台后端服务器，直到后端服务器列表变更。根据此特性可以在服务消费者与服务提供者之间建立有状态的session会话。

源地址哈希算法的缺点在于：除非集群中服务器的非常稳定，基本不会上下线，否则一旦有服务器上线、下线，那么通过源地址哈希算法路由到的服务器是服务器上线、下线前路由到的服务器的概率非常低，如果是session则取不到session，如果是缓存则可能引发"雪崩"。

### 最小连接数法





以上代码均在src中

## 有状态的请求

同一个请求是否分发到同样的服务节点，同一个请求指的是同一个用户或者同样的唯一标示。什么时候同一请求最好（必须）分发到同样的服务节点呢？那就是有状态 -- 请求依赖某些存在于内存或者磁盘的数据，比如web请求的session，比如分布式存储。怎么实现呢，有以下几种办法：

（1）请求分发的时候，保证同一个请求分发到同样的服务节点。

　　这个依赖于负载均衡算法，比如简单的轮训，随机肯定是不行的，哈希法在节点增删的时候也会失效。可行的是一致性hash，以及分布式存储中的按范围分段（即记录哪些请求由哪个服务节点提供服务），代价是需要在load balancer中维护额外的数据。

　　（2）状态数据在backend servers之间共享

　　保证同一个请求分发到同样的服务节点，这个只是手段，目的是请求能使用到对应的状态数据。如果状态数据能够在服务节点之间共享，那么也能达到这个目的。比如服务节点连接到共享数据库，或者内存数据库如memcached（分布式的一致性算法）

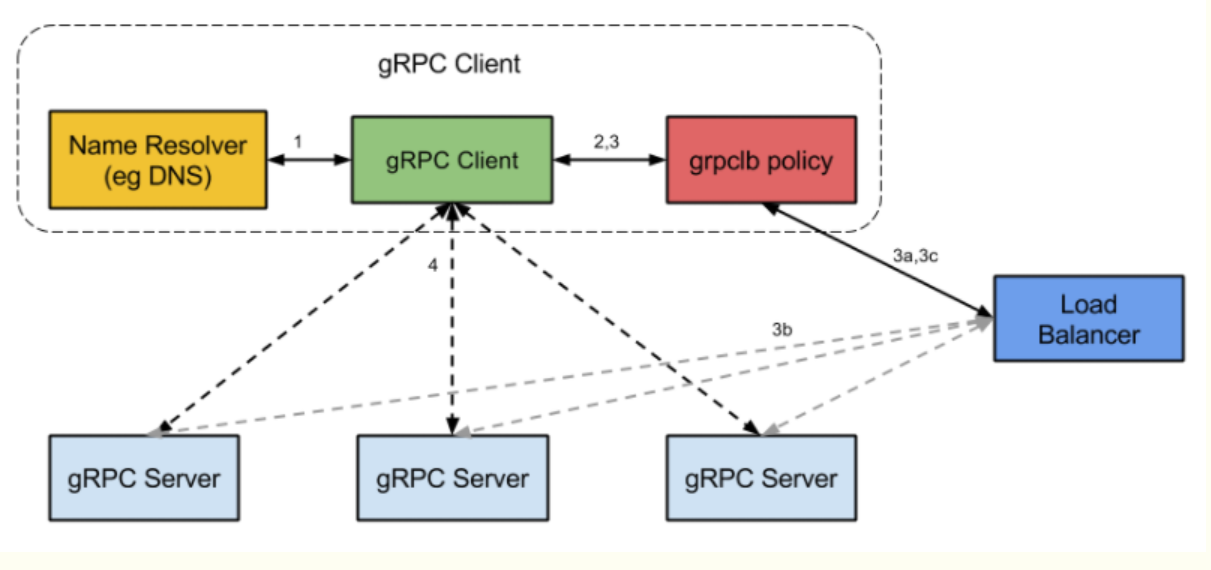
　　（3）状态数据维护在客户端

　　这个在web请求中也有使用，即cookie，不过要考虑安全性，需要加密。

## 关于load balancer

关于load balancer，其实就是说，在哪里做负载均衡，是客户端还是服务端，是请求的发起者还是请求的3。具体而言，要么是在客户端，根据服务节点的信息自行选择，然后将请求直接发送到选中的服务节点；要么是在服务节点集群之前放一个集中式代理（proxy），由代理负责请求求分发。不管哪一种，至少都需要知道当前的服务节点列表这一基础信息。

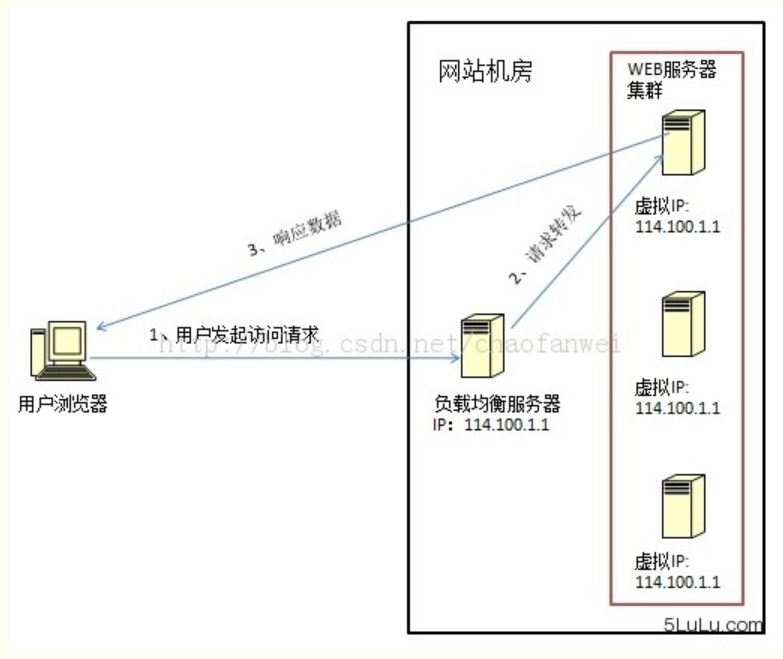
　　如果在客户端实现负载均衡，客户端首先得知道服务器列表，要么是静态配置，要么有简单接口查询，但backend server的详细负载信息，就不适用通过客户端来查询。因此，客户端的负载均衡算法要么是比较简单的，比如轮训（加权轮训）、随机（加权随机）、哈希这几种算法，只要每个客户端足够随机，按照大数定理，服务节点的负载也是均衡的。要在客户端使用较为复杂的算法，比如根据backend的实际负载，那么就需要去额外的负载均衡服务（external load balancing service）查询到这些信息，在[grpc](https://github.com/grpc/grpc/blob/master/doc/load-balancing.md" \t "_blank)中，就是使用的这种办法



可以看到，load balancer与grpc server通信，获得grpc server的负载等具体详细，然后grpc client从load balancer获取这些信息，最终grpc client直连到被选择的grpc server。

而基于Proxy的方式是更为常见的，比如7层的Nginx，四层的F5、LVS，既有硬件路由，也有软件分发。集中式的特点在于方便控制，而且能容易实现一些更精密，更复杂的算法。但缺点也很明显，一来负载均衡器本身可能成为性能瓶颈；二来可能引入额外的延迟，请求一定先发到达负载均衡器，然后到达真正的服务节点。

　　load balance proxy对于请求的响应（response），要么不经过proxy(三角传输模式)，如LVS；要么经过Proxy，如Nginx。下图是LVS示意图（来源见水印）



而如果response也是走load balancer proxy的话，那么整个服务过程对客户端而言就是完全透明的，也防止了客户端去尝试连接后台服务器，提供了一层安全保障！

　　值得注意的是，**load balancer proxy不能成为单点故障**（single point of failure），因此一般会设计为高可用的主从结构

## 尾声

负载均衡是一种推模型，一定会选出一个服务节点，然后把请求推送过来。而换一种思路，使用消息队列，就变成了拉模型：空闲的服务节点主动去拉取请求进行处理，各个节点的负载自然也是均衡的。消息队列相比负载均衡好处在于，服务节点不会被大量请求冲垮，同时增加服务节点更加容易；缺点也很明显，请求不是事实处理的。