**Nginx学习笔记**

# Nginx作用

Nginx (engine x) 是一款轻量级的Web 服务器 、反向代理服务器及电子邮件（IMAP/POP3）代理服务器。

nginx还可以按照调度规则实现动态、静态页面的分离，可以按照轮询、ip哈希、URL哈希、权重等多种方式对后端服务器做负载均衡，同时还支持后端服务器的健康检查。

# 反向代理服务器

反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器，如图2.1

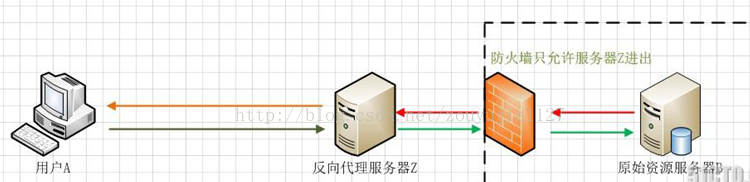


图2.1

用户A始终认为它访问的是原始服务器B而不是代理服务器Z，但实用际上反向代理服务器接受用户A的应答，从原始资源服务器B中取得用户A的需求资源，然后发送给用户A。由于防火墙的作用，只允许代理服务器Z访问原始资源服务器B。尽管在这个虚拟的环境下，防火墙和反向代理的共同作用保护了原始资源服务器B，但用户A并不知情。

# 3、nginx的特点

\* 跨平台

Nginx 可以在大多数 Unix like OS编译运行，而且也有Windows的移植版本。

\* 配置异常简单

非常容易上手。配置风格跟程序开发一样。

\* 非阻塞、高并发连接

数据复制时，磁盘I/O的第一阶段是非阻塞的。官方测试能够支撑5万并发连接，在实际生产环境中跑到2～3万并发连接数(这得益于Nginx使用了最新的epoll模型)。

\* 事件驱动

通信机制采用epoll模型，支持更大的并发连接。

\* nginx代理和后端web服务器间无需长连接；

\* 接收用户请求是异步的

先将用户请求全部接收下来，再一次性发送后后端web服务器，极大的减轻后端web服务器的压力

\* 发送响应报文时，是边接收来自后端web服务器的数据，边发送给客户端的。

\* 网络依赖型低

理论上讲，只要能够ping通就可以实施负载均衡，而且可以有效区分内网和外网流量

\* 支持服务器检测

Nginx能够根据应用服务器处理页面返回的状态码、超时信息等检测服务器是否出现故障，并及时返回错误的请求重新提交到其它节点上。

# nginx的工作模式

Nginx默认是以多进程的方式来工作的，当然nginx也是支持单线程的方式的，如图4.1。

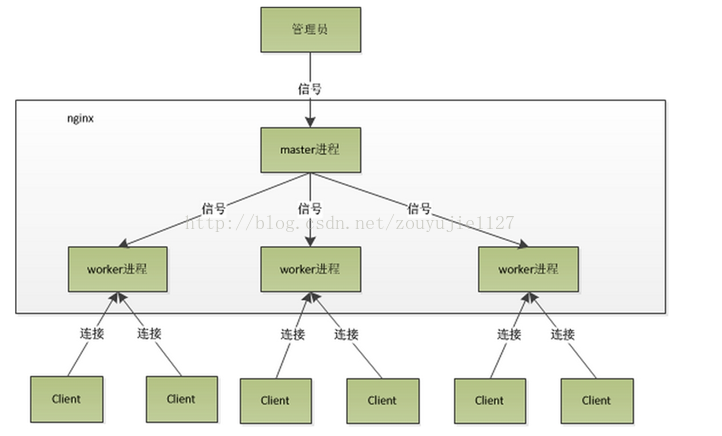


图4.1

nginx在启动后，会有一个master进程和多个worker进程。master进程主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控 worker进程的运行状态,当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

而基本的网络事件，则是放在worker进程中来处理了 。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的 。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。 worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的。

**Master接收到信号以后怎样进行处理（./nginx -s reload ）?**

首先master进程在接到信号后，会先重新加载配置文件，然后再启动新的进程，并向所有老的进程发送信号，告诉他们可以光荣退休了。新的进程在启动后，就开始接收新的请求，而老的进程在收到来自master的信号后，就不再接收新的请求，并且在当前进程中的所有未处理完的请求处理完成后，再退出。

**worker进程又是如何处理请求的呢？**

当我们提供80端口的http服务时，一个连接请求过来，每个worker进程都有可能处理这个连接，怎么做到的呢？首先，每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket之后，然后再fork出多个worker进程，这样每个worker进程都可以去accept这个socket(当然不是同一个socket，只是每个进程的这个socket会监控在同一个ip地址与端口，这个在网络协议里面是允许的)。一般来说，当一个连接进来后，所有accept在这个socket上面的进程，都会收到通知，而只有一个进程可以accept这个连接，其它的则accept失败，这是所谓的**惊群现象**。当然，nginx也不会视而不见，所以nginx提供了一个accept\_mutex这个东西，从名字上，我们可以看这是一个加在accept上的一把共享锁。有了这把锁之后，同一时刻，就只会有一个进程在accpet连接，这样就不会有惊群问题了。accept\_mutex是一个可控选项，我们可以显示地关掉，默认是打开的。当一个worker进程在accept这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由worker进程来处理，而且只在一个worker进程中处理。

# 5、nginx的upstream

nginx 的 upstream目前支持 4 种方式的分配

1)、轮询（默认）

每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。

2)、weight （权重）

指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。

3)、ip\_hash

每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。

4)、fair（第三方）

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

1. 、url\_hash（第三方）

# 6、实现多台服务器之间session的共享

**1) 不使用session，换作cookie**

能把session改成cookie，就能避开session的一些弊端。

**2) 应用服务器自行实现共享**

asp.net可以用数据库或memcached来保存session，从而在asp.net本身建立了一个session集群，用这样的方式可以令 session保证稳定，即使某个节点有故障，session也不会丢失，适用于较为严格但请求量不高的场合。但是它的效率是不会很高的，不适用于对效率 要求高的场合。

**3) ip\_hash**

nginx中的ip\_hash技术能够将某个ip的请求定向到同一台后端，这样一来这个ip下的某个客户端和某个后端就能建立起稳固的session，ip\_hash是在upstream配置中定义的：

upstream backend {  
  server 127.0.0.1:8080 ;  
  server 127.0.0.1:9090 ;  
   ip\_hash;  
 }

ip\_hash是容易理解的，但是因为仅仅能用ip这个因子来分配后端，因此ip\_hash是有缺陷的，不能在一些情况下使用：

1、 nginx不是最前端的服务器。ip\_hash要求nginx一定是最前端的服务器，否则nginx得不到正确ip，就不能根据ip作hash。譬如使用的是squid为最前端，那么nginx取ip时只能得到squid的服务器ip地址，用这个地址来作分流是肯定错乱的。

2、 nginx的后端还有其它方式的负载均衡。假如nginx后端又有其它负载均衡，将请求又通过另外的方式分流了，那么某个客户端的请求肯定不能定位到同一台session应用服务器上。这么算起来，nginx后端只能直接指向应用服务器，或者再搭一个squid，然后指向应用服务器。最好的办法是用location作一次分流，将需要session的部分请求通过ip\_hash分流，剩下的走其它后端去。

**4）upstream\_hash**

为了解决ip\_hash的一些问题，可以使用upstream\_hash这个第三方模块，这个模块多数情况下是用作url\_hash的，但是并不妨碍将它用来做session共享：

假如前端是squid，他会将ip加入x\_forwarded\_for这个http\_header里，用upstream\_hash可以用这个头做因子，将请求定向到指定的后端：

可见这篇文档：<http://www.sudone.com/nginx/nginx_url_hash.html>

在文档中是使用$request\_uri做因子，稍微改一下：

hash   $http\_x\_forwarded\_for;

这样就改成了利用x\_forwarded\_for这个头作因子，在nginx新版本中可支持读取cookie值，所以也可以改成：

hash   $cookie\_jsessionid;

# 应用场景

*负载均衡* 技术在现有网络结构之上提供了一种廉价、有效、透明的方法，来扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的 灵活性和可用性。

它有两方面的含义：

首先，大量的并发访问或数据流量分担到多台节点设备上分别处理，减少用户等待响应的时间；

其次，单个重负载的运算分担 到多台节点设备上做并行处理，每个节点设备处理结束后，将结果汇总，返回给用户，系统处理能力得到大幅度提高

