

---

# 反向代理服务器 Nginx

## 课程讲义

主讲: Reythor 雷

2019

# 反向代理服务器 Nginx

## 第1章 Nginx 概述

### 1.1 Nginx 简介

Nginx (engine x) 是一个轻量级的、高性能的、基于 Http 的、反向代理服务器，静态 web 服务器。

Nginx 最初是由俄罗斯人 Igor Sysoev（伊戈尔·赛索耶夫）使用 C 语言为俄罗斯访问量第二的 Rambler.ru 站点开发的一款服务器。2004 年 10 月发布第一个版本。

Nginx 的官网：<http://nginx.org>

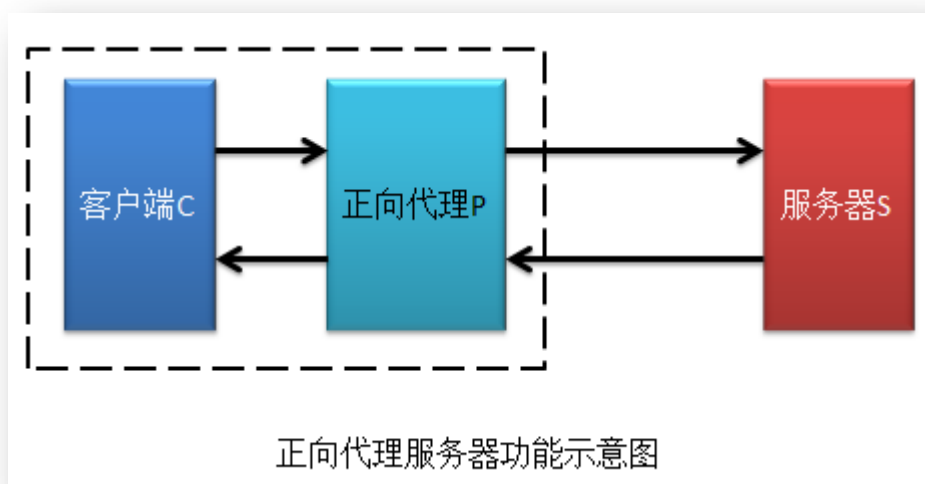
国内大型的站点，例如百度、京东、新浪、网易、腾讯、淘宝等，都使用了 Nginx。

<https://www.netcraft.com/>

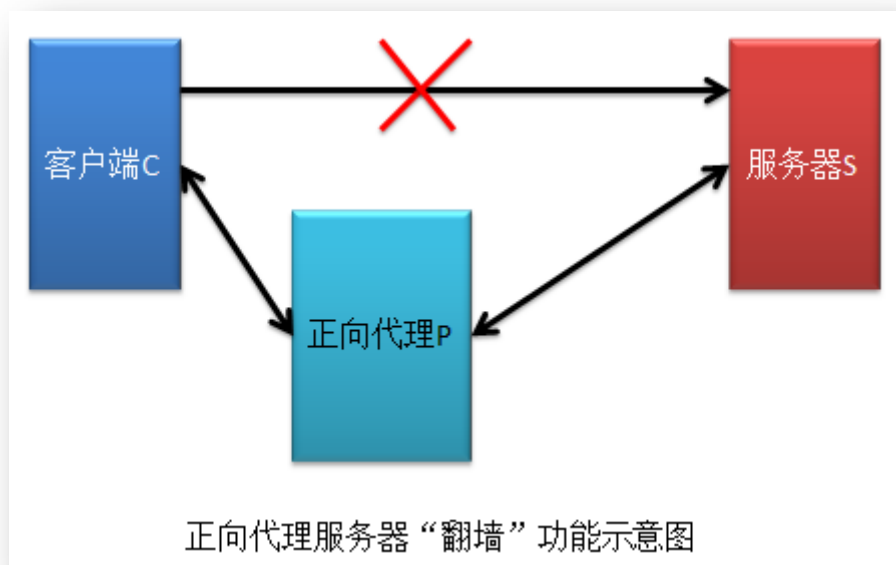
### 1.2 代理服务器

#### 1.2.1 正向代理

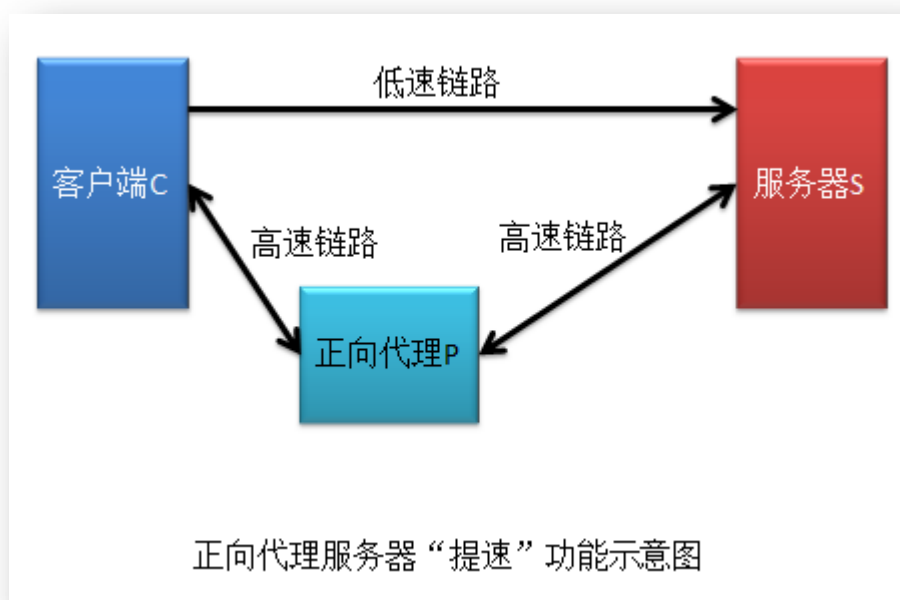
##### （1）隐藏



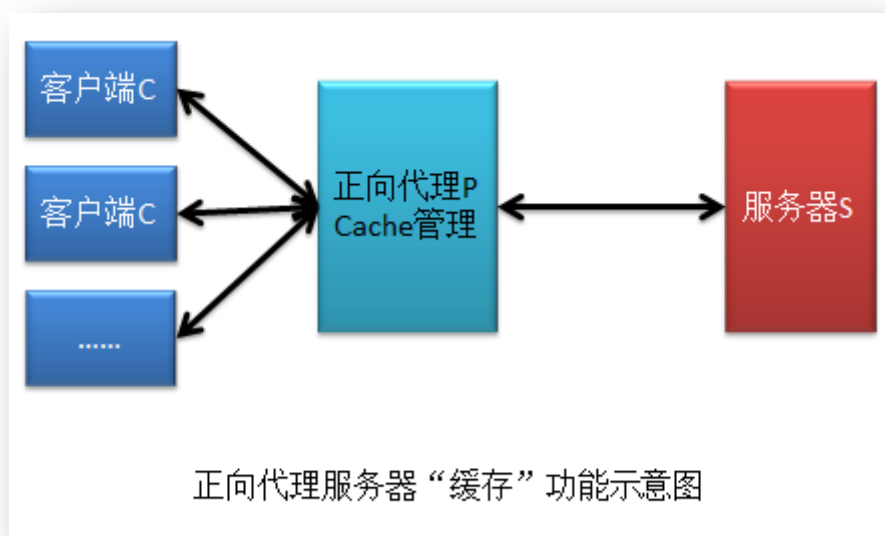
## (2) 翻墙



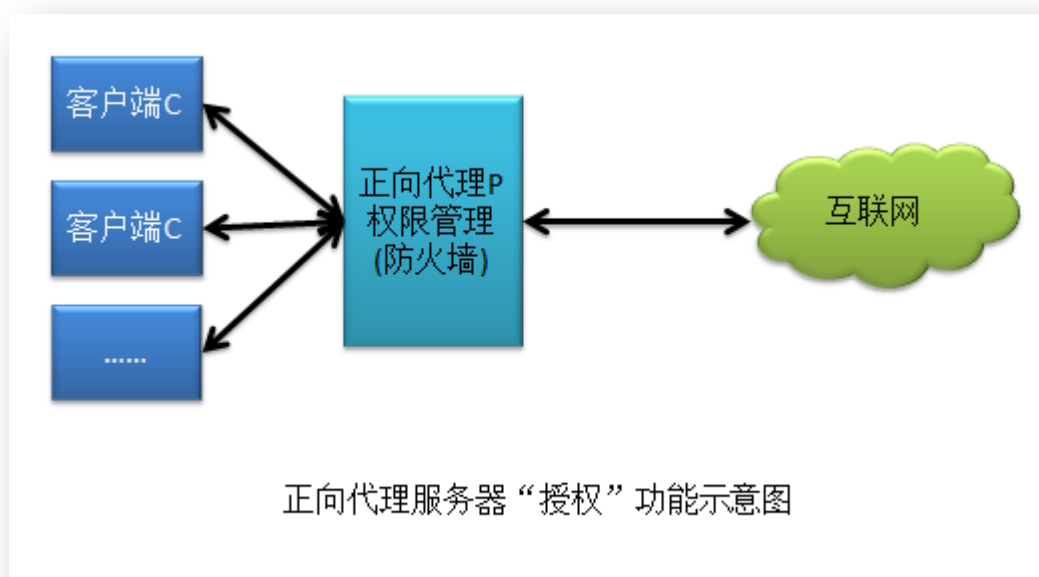
## (3) 提速



#### (4) 缓存

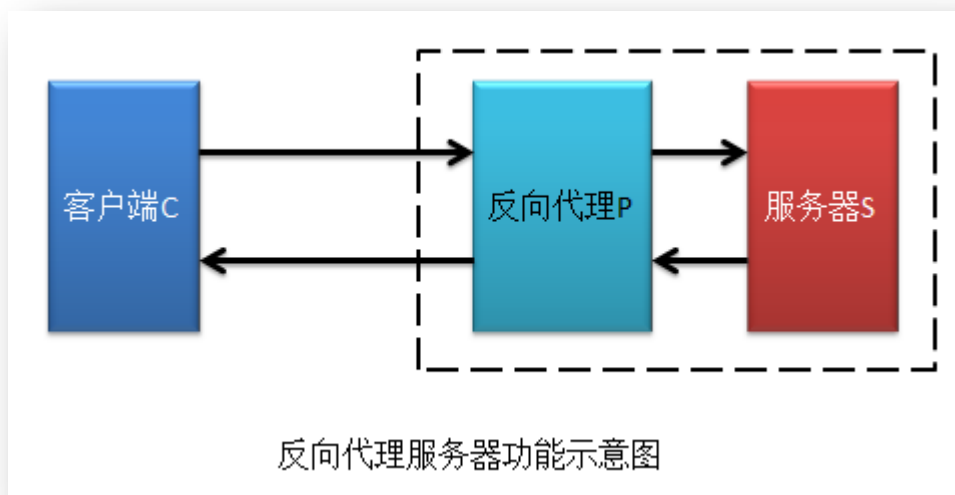


#### (5) 授权

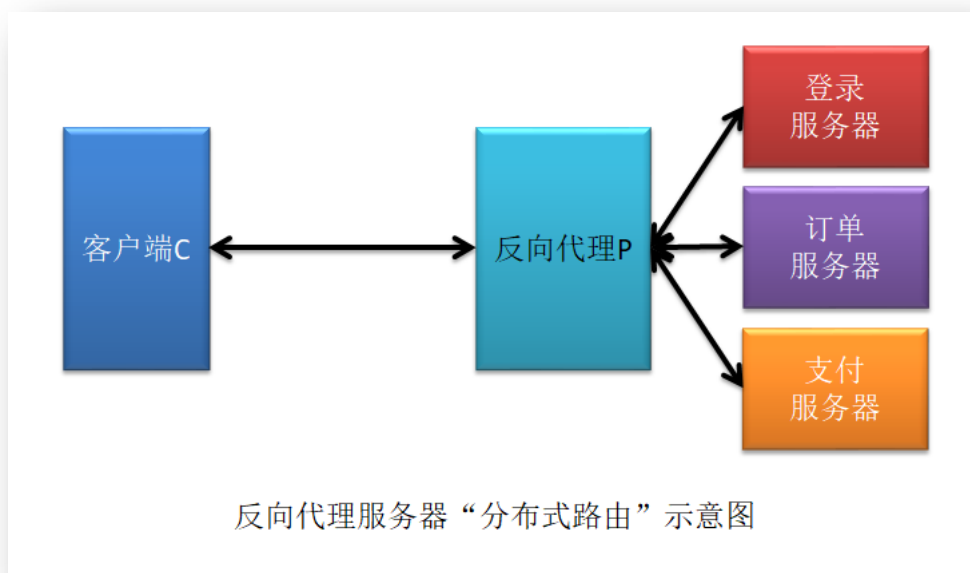


## 1.2.2 反向代理

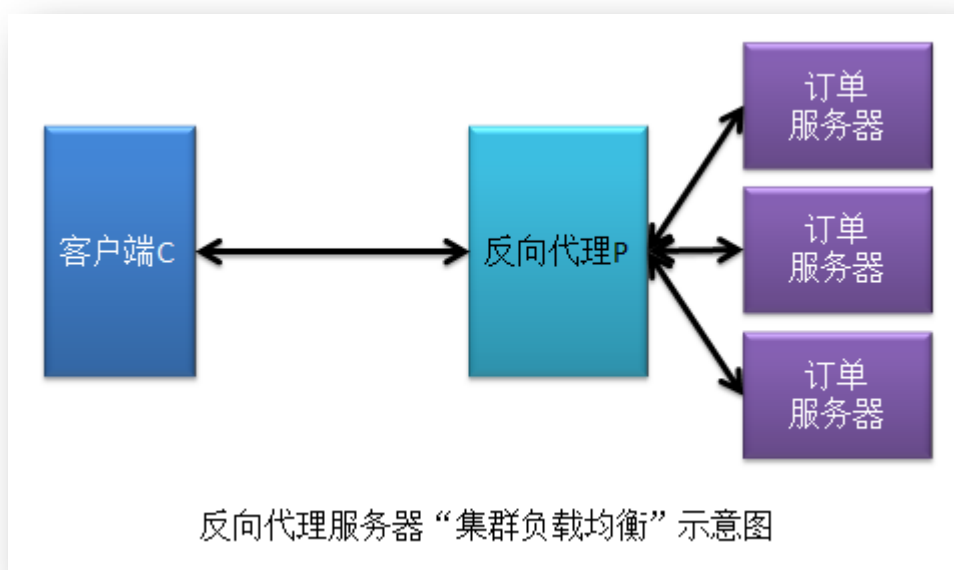
### (1) 保护隐藏



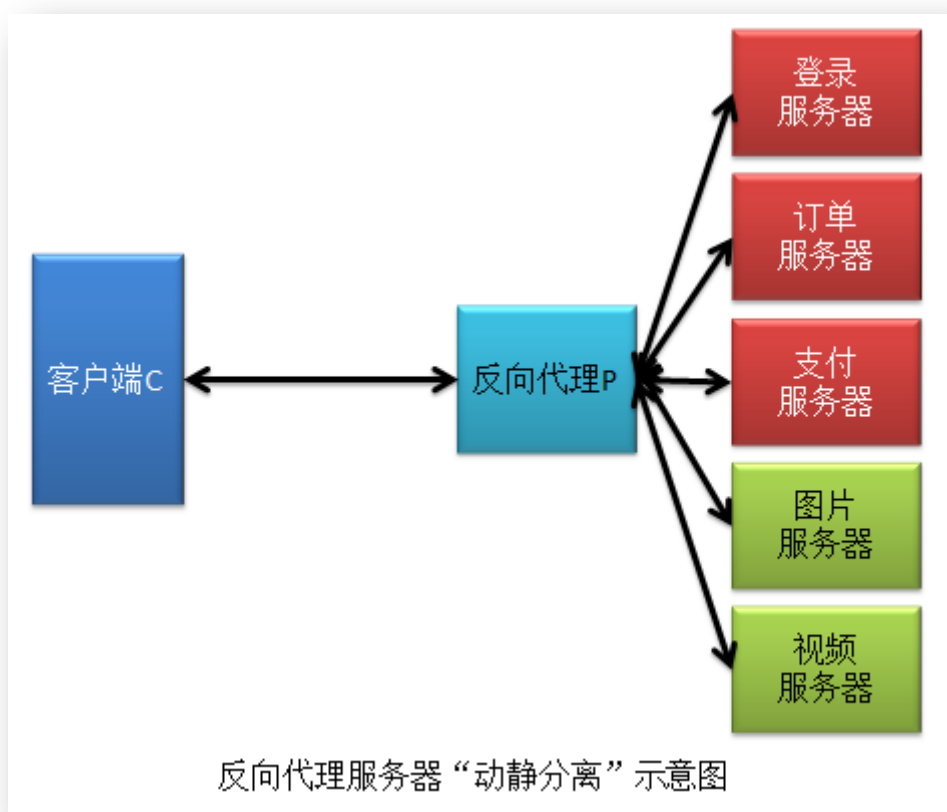
### (2) 分布式路由



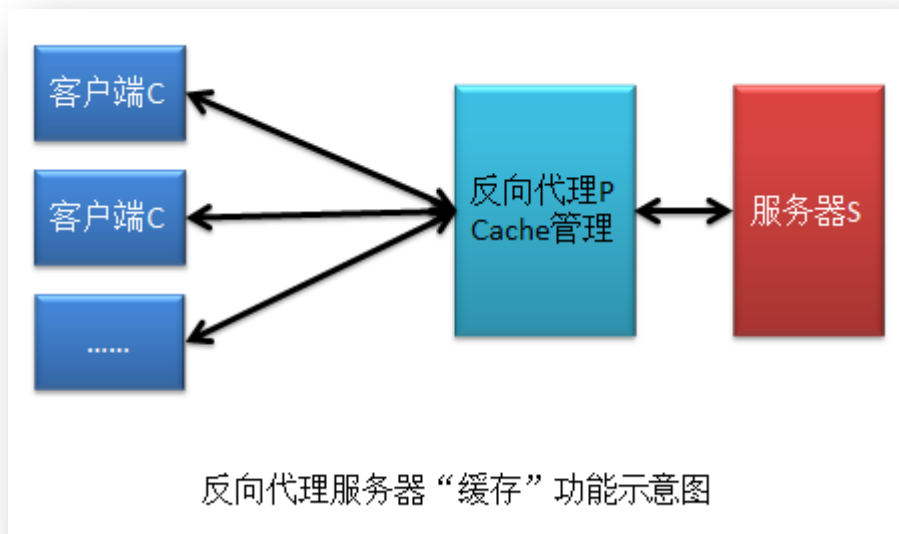
### (3) 负载均衡



#### (4) 动静分离



## （5）数据缓存



### 1.2.3 正向代理与反向代理的区别

客户端是否清楚自己所要访问的服务器是谁？

架设的位置不同

## 1.3 Nginx 的特点

### 1.3.1 高并发

一个 Nginx 服务器在不做任何配置的情况下并发量可达 1000 左右。在硬件条件允许的前提下，Nginx 可以支持高达 5-10 万的并发量（除了 Nginx 的设置外，Linux 主机需要做大量的设置来配合 Nginx）。

对比一下 Tomcat。Tomcat 服务器默认的并发量为 150（不做任何配置）。即，当有超过 150 个用户同时访问某 Servlet 时，Tomcat 的响应就会变得非常慢。

### 1.3.2 低消耗

官方给出的测试结果，10000 个非活跃连接，在 Nginx 中仅消耗 2.5M 内存。对于一般性的 DoS 攻击来说就不是事儿，但对于 DDos 也会是问题。



### 1.3.3 热部署

可以在 7\*24 小时不间断服务的前提下，进行 Nginx 版本的平滑升级，Nginx 配置文件的平滑修改。即在不停机的情况下升级 Nginx，修改替换 Nginx 配置文件。

### 1.3.4 高可用

Nginx 之所以可以实现高并发，是因为其具有很多工作进程 `worker`。当这些工作进程中的某些出现问题停止工作时，并不会影响整个系统的整体运行。因为其它 `worker` 会接替那些出问题的线程。

### 1.3.5 高扩展

Nginx 之所以现在的用户很多，是因为很多功能都已经开发好并模块化。若需要哪些功能，只需要安装相应功能的扩展模块即可。根据编写扩展模块所使用的语言的不同，可以划分为两类：C 语言扩展模块与 LUA 脚本扩展模块。 <http://openresty.org/cn/>

## 1.4 Nginx 的下载与安装

### 1.4.1 Nginx 的下载

nginx 的官网为： <http://nginx.org>。

### 1.4.2 Nginx 的源码安装

#### （1）安装 Nginx

##### A、上传 Nginx

将下载好的 Nginx 上传到新复制的主机的 `/usr/tools` 目录。

##### B、安装 gcc

由于 Nginx 是由 C/C++ 语言编写的，所以对其进行编译就必须使用相关编译器。对于 C/C++ 语言的编译器，使用最多的是 `gcc` 与 `gcc-c++`，而这两款编译器在 CentOS7 中是没有安装的，所以首先要安装这两款编译器。

```
nginxos x
[root@nginxos ~]# yum -y install gcc gcc-c++
```

### C、安装依赖库

基本的 Nginx 功能依赖于一些基本的库，在安装 Nginx 之前需要提前安装这些库。

```
nginxos x
[root@nginxos ~]# yum -y install pcre-devel openssl-devel
```

### D、创建解压目录

在/usr 下创建 apps 目录，用于存放解压后的安装包程序。

```
01nginx x
[root@01nginx ~]# mkdir /usr/apps
[root@01nginx ~]#
```

### E、解压 Nginx

将 Nginx 解压到/usr/apps 目录中。

```
nginxos x
[root@nginxos tools]# tar -zxvf nginx-1.12.2.tar.gz -C /usr/apps
```

进入到/usr/apps 目录中的 Nginx 解压包目录，查看 Nginx 的目录。

```

nginxos x
[root@nginxOS nginx-1.12.2]# ll
total 724
drwxr-xr-x. 6 1001 1001 4096 Feb  1 14:24 auto
-rw-r--r--. 1 1001 1001 278202 Oct 17 21:16 CHANGES
-rw-r--r--. 1 1001 1001 423948 Oct 17 21:16 CHANGES.ru
drwxr-xr-x. 2 1001 1001 4096 Feb  1 14:24 conf
-rwxr-xr-x. 1 1001 1001 2481 Oct 17 21:16 configure
drwxr-xr-x. 4 1001 1001 4096 Feb  1 14:24 contrib
drwxr-xr-x. 2 1001 1001 4096 Feb  1 14:24 html
-rw-r--r--. 1 1001 1001 1397 Oct 17 21:16 LICENSE
drwxr-xr-x. 2 1001 1001 4096 Feb  1 14:24 man
-rw-r--r--. 1 1001 1001 49 Oct 17 21:16 README
drwxr-xr-x. 9 1001 1001 4096 Feb  1 14:24 src
[root@nginxOS nginx-1.12.2]#

```

## F、生成 makefile

在 Nginx 解压目录下运行 `make` 命令，用于完成编译。但此时会给出提示：没有指定目标，并且没有发现编译文件 `makefile`。

```

nginxos x
[root@nginxOS nginx-1.12.2]# make
make: *** No targets specified and no makefile found.  stop.
[root@nginxOS nginx-1.12.2]#

```

编译命令 `make` 需要根据编译文件 `makefile` 进行编译，所以在编译之前需要先生成编译文件 `makefile`。使用 `configure` 命令可以生成该文件。

## G、编译安装

```

01nginx x
[root@01nginx nginx-1.12.2]# make && make install

```

## (2) 使 nginx 命令随处可用

在 Nginx 的安装目录 `/usr/local/nginx` 中有一个 `sbin` 目录，其中存放着 nginx 的命令程序 `nginx`。

```
nginxos x
[root@nginx05 ~]# ll /usr/local/nginx/sbin/
total 5672
-rwxr-xr-x. 1 root root 5805211 Feb  1 14:56 nginx
[root@nginx05 ~]#
```

```
S ~]# ln -s /usr/local/nginx/sbin/nginx /usr/local/sbin/
S ~]#
S ~]# ll /usr/local/sbin
1 root root 27 Feb  1 15:37 nginx -> /usr/local/nginx/sbin/nginx
S ~]#
```

### 1.4.3 Nginx 命令

#### (1) 查看命令选项 nginx -h

使用 `nginx -h` 可以查看 Nginx 命令的选项。

```
01nginx x
[root@01nginx sbin]# ./nginx -h
nginx version: nginx/1.12.2
usage: nginx [-?hvvtTq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
  -?, -h      : this help
  -v          : show version and exit
  -V          : show version and configure options then exit
  -t          : test configuration and exit
  -T          : test configuration, dump it and exit
  -q          : suppress non-error messages during configuration testing
  -s signal   : send signal to a master process: stop, quit, reopen, reload
  -p prefix   : set prefix path (default: /usr/local/nginx/)
  -c filename : set configuration file (default: conf/nginx.conf)
  -g directives : set global directives out of configuration file

[root@01nginx sbin]#
```

#### (2) 相看 Nginx 版本信息 nginx -v 或 -V

`nginx -v`: 显示 Nginx 版本信息。

```

✓ nginxos x
[root@nginxOS ~]# nginx -v
nginx version: nginx/1.14.1
[root@nginxOS ~]#
    
```

nginx -V: 显示更多的版本相关信息, 例如 gcc 的版本, OpenSSL 的版本等。

```

✓ nginxos x
[root@nginxOS ~]# nginx -V
nginx version: nginx/1.14.1
built by gcc 4.8.5 20150623 (Red Hat 4.8.5-28) (GCC)
built with openssl 1.0.2k-fips 26 Jan 2017
TLS SNI support enabled
configure arguments: --prefix=/usr/local/nginx --with
[root@nginxOS ~]#
    
```

### (3) 测试配置文件命令 nginx -tq

nginx -t: 测试配置文件是否正确, 默认只测试默认的配置文​​件 conf/nginx.conf。

nginx -T: 测试配置文件是否正确, 并显示配置文件内容。

nginx -tq: 在配置文件测试过程中, 禁止显示非错误信息, 即只显示错误信息。

```

✓ nginxos x
[root@nginxOS ~]# nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
[root@nginxOS ~]#
    
```

可以结合 -c 选项指定要测试的配置文件。注意, 其不会启动 nginx。

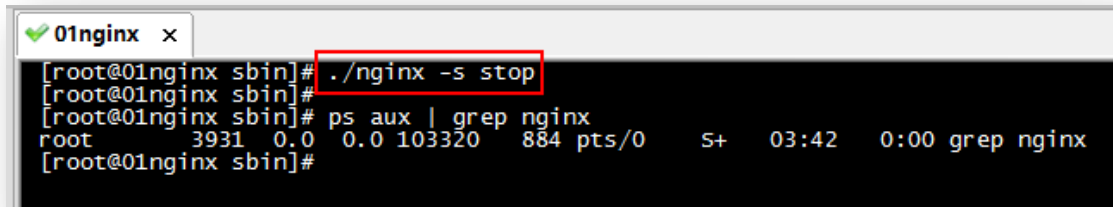
```

✓ nginxos x
[root@nginxOS ~]# nginx -c /usr/local/nginx/conf/nginx2.conf -t
nginx: the configuration file /usr/local/nginx/conf/nginx2.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx2.conf test is successful
[root@nginxOS ~]#
    
```

## (4) 停止命令 `nginx -s stop/quit`

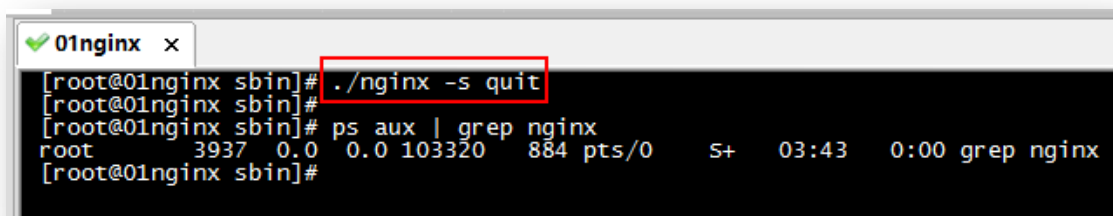
在 `nginx` 命令后通过 `-s` 选项，可以指定不同的信号完成不同的功能。

- `nginx -s stop`: 强制停止 Nginx，无论当前工作进程是否正在处理工作。
- `nginx -s quit`: 优雅停止 Nginx，使当前的工作进程完成当前工作后停止。



```

01nginx x
[root@01nginx sbin]# ./nginx -s stop
[root@01nginx sbin]#
[root@01nginx sbin]# ps aux | grep nginx
root      3931  0.0  0.0 103320  884 pts/0    S+   03:42   0:00 grep nginx
[root@01nginx sbin]#
  
```

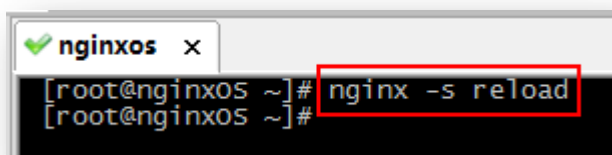


```

01nginx x
[root@01nginx sbin]# ./nginx -s quit
[root@01nginx sbin]#
[root@01nginx sbin]# ps aux | grep nginx
root      3937  0.0  0.0 103320  884 pts/0    S+   03:43   0:00 grep nginx
[root@01nginx sbin]#
  
```

## (5) 平滑重启命令 `nginx -s reload`

在不重启 Nginx 的前提下重新加载 Nginx 配置文件，称为平滑重启。



```

nginxos x
[root@nginxos ~]# nginx -s reload
[root@nginxos ~]#
  
```

## (6) `nginx -s reopen`

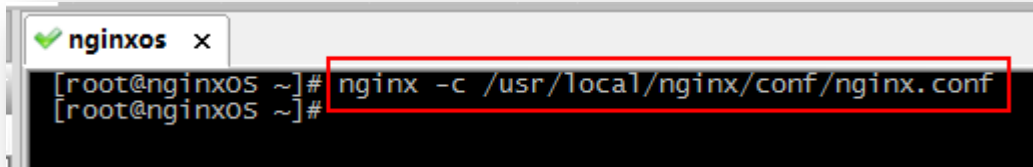
重新打开日志文件。

## (7) `nginx -p`

指定 Nginx 配置文件的存放路径。

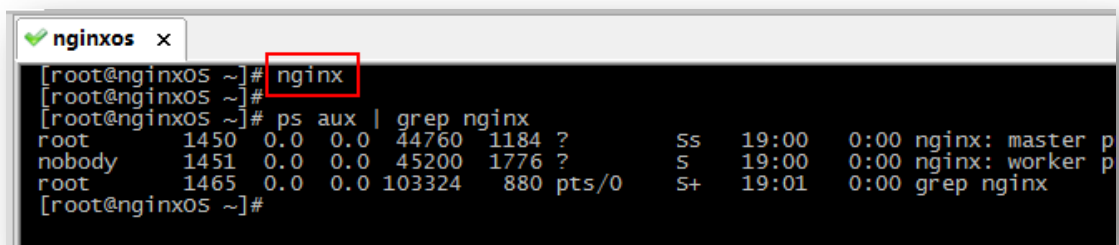
## (8) 启动命令 `nginx -c file`

`nginx -c`（小写字母）可启动 Nginx，启动成功后无任何提示。



```
[root@nginx05 ~]# nginx -c /usr/local/nginx/conf/nginx.conf
[root@nginx05 ~]#
```

若不指定配置文件，则默认加载的是 Nginx 安装目录下的 `conf/nginx.conf`。



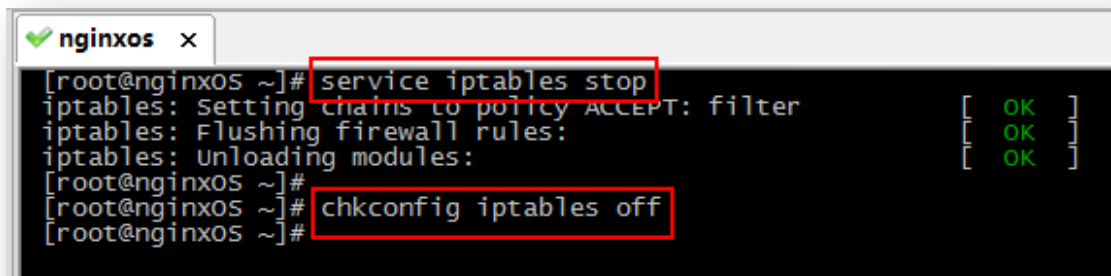
```
[root@nginx05 ~]# nginx
[root@nginx05 ~]#
[root@nginx05 ~]# ps aux | grep nginx
root      1450  0.0  0.0  44760  1184 ?        Ss   19:00   0:00 nginx: master p
nobody    1451  0.0  0.0  45200  1776 ?        S    19:00   0:00 nginx: worker p
root      1465  0.0  0.0 103324   880 pts/0    S+   19:01   0:00 grep nginx
[root@nginx05 ~]#
```

## (9) `nginx -g`

设置配置文件以外的全局指令。

### 1.4.4 页面访问测试

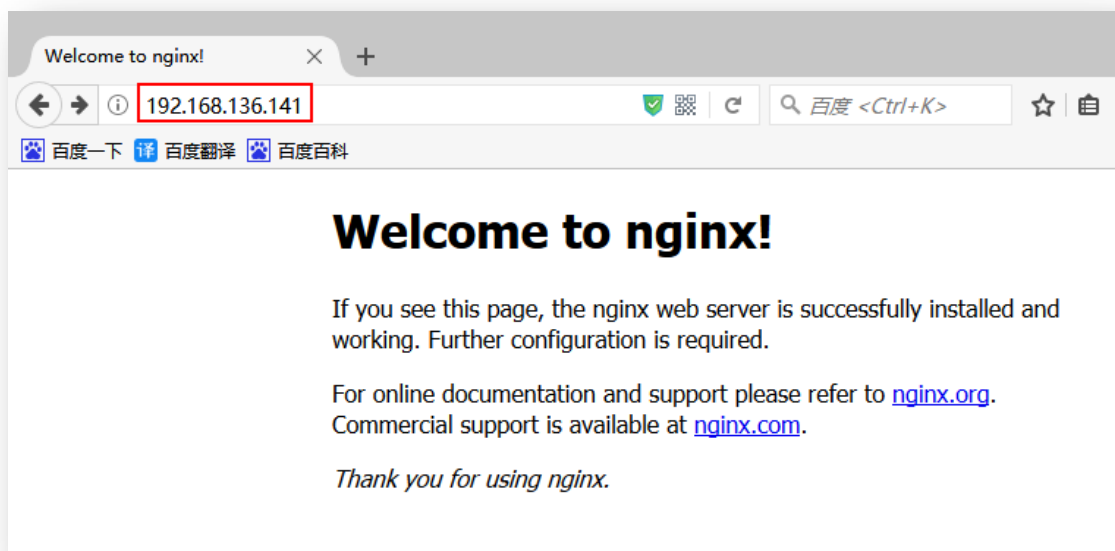
#### (1) 关闭防火墙



```
[root@nginx05 ~]# service iptables stop
iptables: Setting chains to policy ACCEPT: filter      [ OK ]
iptables: Flushing firewall rules:                    [ OK ]
iptables: Unloading modules:                          [ OK ]
[root@nginx05 ~]#
[root@nginx05 ~]# chkconfig iptables off
[root@nginx05 ~]#
```

## (2) 浏览器访问

由于 Nginx 服务器默认的端口号为 80，所以在浏览器中直接输入 Nginx 的主机名或 IP，就可以看到 Nginx 欢迎页面。只要可以看到以下页面信息，则说明 Nginx 安装运行成功。





## 第2章 Nginx 核心配置

### 2.1 Nginx 性能调优

在 Nginx 性能调优中，有两个非常重要的理论点（面试点）需要掌握。所以，下面首先讲解这两个知识点，再进行性能调优的配置。

#### 2.1.1 零拷贝（Zero Copy）

##### （1）零拷贝基础

零拷贝指的是，从一个存储区域到另一个存储区域的 copy 任务没有 CPU 参与。零拷贝通常用于网络文件传输，以减少 CPU 消耗和内存带宽占用，减少用户空间与 CPU 内核空间的拷贝过程，减少用户上下文与 CPU 内核上下文间的切换，提高系统效率。

用户空间指的是用户可操作的内存缓存区域，CPU 内核空间是指仅 CPU 可以操作的寄存器缓存及内存缓存区域。

用户上下文指的是用户状态环境，CPU 内核上下文指的是 CPU 内核状态环境。

零拷贝需要 DMA 控制器的协助。DMA，Direct Memory Access，直接内存存取，是 CPU 的组成部分，其可以在 CPU 内核（算术逻辑运算器 ALU 等）不参与运算的情况下将数据从

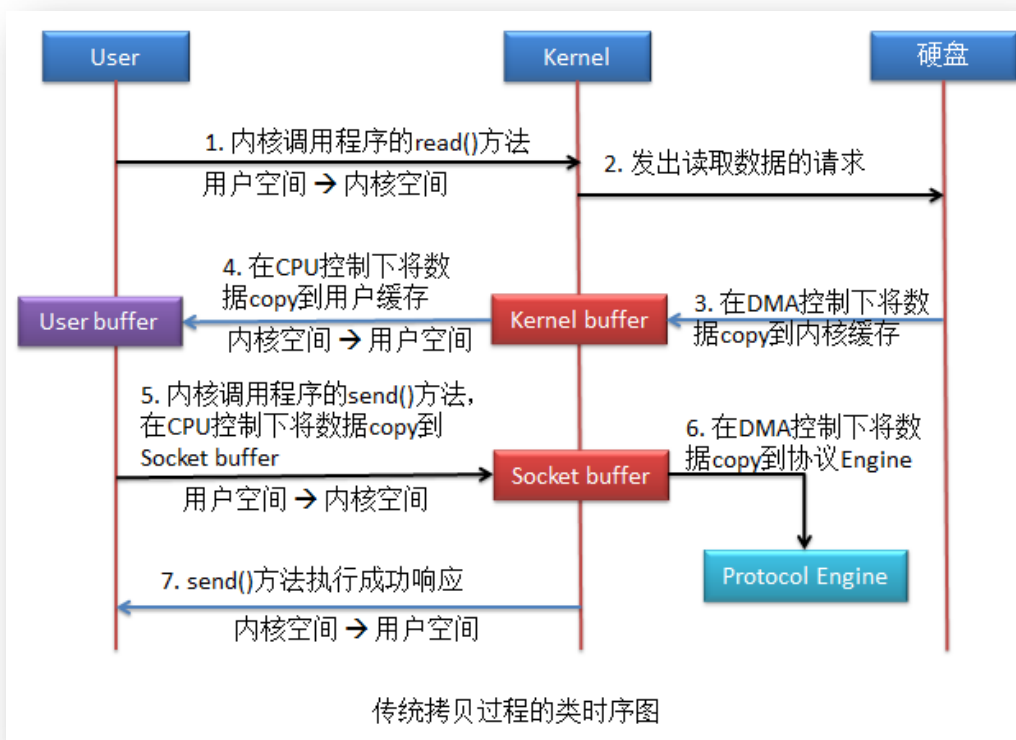
一个地址空间拷贝到另一个地址空间。

## (2) 传统拷贝方式

下面均以“将一个硬盘中的文件通过网络发送出去”的过程为例，来详细详细分析不同拷贝方式的实现细节。

### A、实现细节

首先通过应用程序的 `read()` 方法将文件从硬盘读取出来，然后再调用 `send()` 方法将文件发送出去。



### B、总结

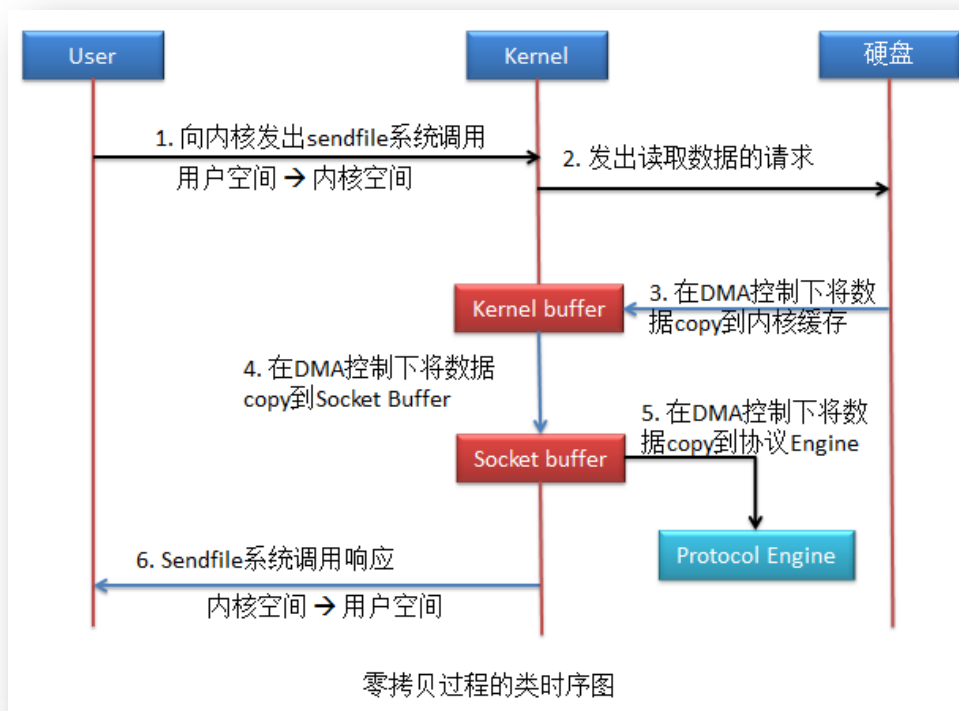
该拷贝方式共进行了 4 次用户空间与内核空间的上下文切换，以及 4 次数据拷贝，其中两次拷贝存在 CPU 参与。

我们发现一个很明显的问题：应用程序的作用仅仅就是一个数据传输的中介，最后将 kernel buffer 中的数据传递到了 socket buffer。显然这是没有必要的。所以就引入了零拷贝。

### (3) 零拷贝方式

#### A、实现细节

Linux 系统（CentOS6 及其以上版本）对于零拷贝是通过 `sendfile` 系统调用实现的。



#### B、总结

该拷贝方式共进行了 2 次用户空间与内核空间的上下文切换，以及 3 次数据拷贝，但整个拷贝过程均没有 CPU 的参与，这就是零拷贝。

我们发现这里还存在一个问题：kernel buffer 到 socket buffer 的拷贝需要吗？kernel buffer 与 socket buffer 有什么区别呢？DMA 控制器所控制的拷贝过程有一个要求，数据在源头的存放地址空间必须是连续的。kernel buffer 中的数据无法保证其连续性，所以需要将数据再拷贝到 socket buffer，socket buffer 可以保证数据的连续性。

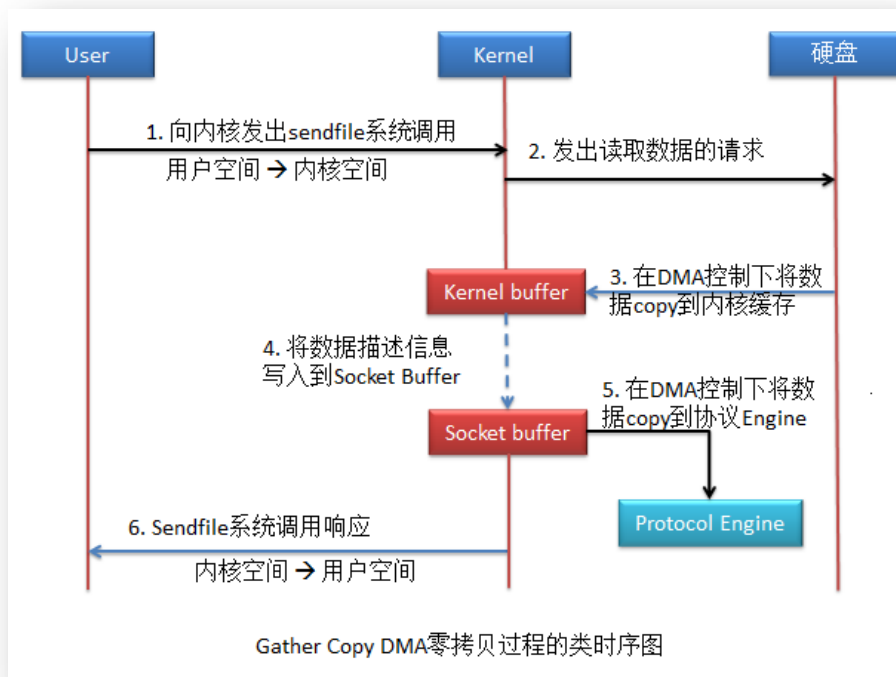
这个拷贝过程能否避免呢？可以，只要主机的 DMA 支持 Gather Copy 功能，就可以避免由 kernel buffer 到 socket buffer 的拷贝。

### (4) Gather Copy DMA 零拷贝方式

由于该拷贝方式是由 DMA 完成，与系统无关，所以只要保证系统支持 `sendfile` 系统调用功能即可。

## A、实现细节

该方式中没有数据拷贝到 `socket buffer`。取而代之的是只是将 `kernel buffer` 中的数据描述信息写到了 `socket buffer` 中。数据描述信息包含了两方面的信息：`kernel buffer` 中数据的地址及偏移量。



## B、总结

该拷贝方式共进行了 2 次用户空间与内核空间的上下文切换，以及 2 次数据拷贝，并且整个拷贝过程均没有 CPU 的参与。

该拷贝方式的系统效率是高了，但与传统相比，也存在有不足。传统拷贝中 `user buffer` 中存有数据，因此应用程序能够对数据进行修改等操作；零拷贝中的 `user buffer` 中没有了数据，所以应用程序无法对数据进行操作了。Linux 的 `mmap` 零拷贝解决了这个问题。

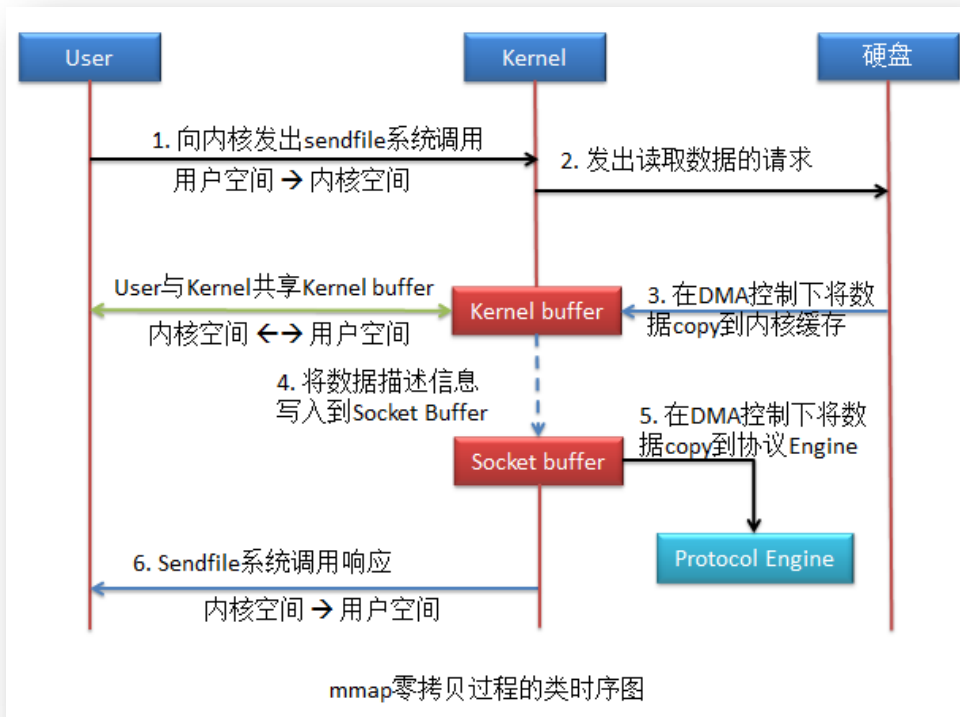
## （5）mmap 零拷贝

`mmap` 零拷贝是对零拷贝的改进。当然，若当前主机的 DMA 支持 Gather Copy，`mmap` 同样可以实现 Gather Copy DMA 的零拷贝。

## A、实现细节

该方式与零拷贝的唯一区别是，应用程序与内核共享了 `Kernel buffer`。由于是共享，所以应用程序也就可以操作该 `buffer` 了。当然，应用程序对于 `Kernel buffer` 的操作，就会引发

用户空间与内核空间的相互切换。



## B、总结

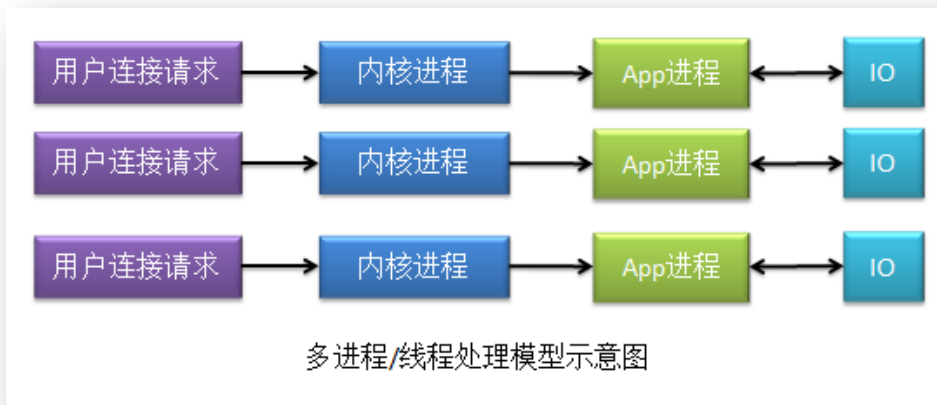
该拷贝方式共进行了 4 次用户空间与内核空间的上下文切换，以及 2 次数据拷贝，并且整个拷贝过程均没有 CPU 的参与。虽然较之前面的零拷贝增加了两次上下文切换，但应用程序可以对数据进行修改了。

### 2.1.2 多路复用器 select|poll|epoll

#### （1）基本知识

若要理解 select、poll 与 epoll 多路复用器的工作原理，就需要首先了解什么是多路复用器。而要了解什么是多路复用器，就需要先了解什么是“多进程/多线程连接处理模型”。

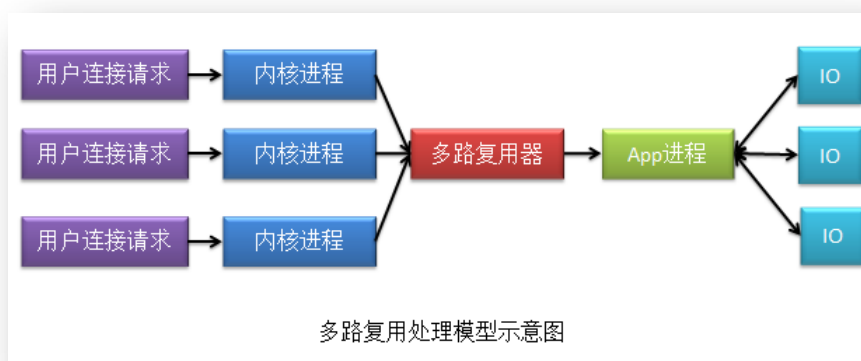
## A、多进程/多线程连接处理模型



在该模型下，一个用户连接请求会由一个内核进程处理，而一个内核进程会创建一个应用程序进程，即 **app** 进程来处理该连接请求。应用程序进程在调用 **IO** 时，采用的是 **BIO** 通讯方式，即应用程序进程在未获取到 **IO** 响应之前是处于阻塞态的。

该模型的优点是，内核进程不存在对 **app** 进程的竞争，一个内核进程对应一个 **app** 进程。但，也正因为如此，所以其弊端也就显而易见了。需要创建过多的 **app** 进程，而该创建过程十分的消耗系统资源。且一个系统的进程数量是有上限的，所以该模型不能处理高并发的情况。

## B、多路复用连接处理模型



在该模型下，只有一个 **app** 进程来处理内核进程事务，且 **app** 进程一次只能处理一个内核进程事务。故这种模型对于内核进程来说，存在对 **app** 进程的竞争。

在前面的“多进程/多线程连接处理模型”中我们说过，**app** 进程只要被创建了就会执行内核进程事务。那么在该模型下，应用程序进程应该执行哪个内核进程事务呢？谁的准备就绪了，**app** 进程就执行哪个。但 **app** 进程怎么知道哪个内核进程就绪了呢？需要通过“多路复用器”来获取各个内核进程的状态信息。那么多路复用器又是怎么获取到内核进程的状

态信息的呢？不同的多路复用器，其算法不同。常见的有三种：`select`、`poll` 与 `epoll`。

`app` 进程在进行 IO 时，其采用的是 NIO 通讯方式，即该 `app` 进程不会阻塞。当一个 IO 结果返回时，`app` 进程会暂停当前事务，将 IO 结果返回给对应的内核进程。然后再继续执行暂停的线程。

该模型的优点很明显，无需再创建很多的应用程序进程去处理内核进程事务了，仅需一个即可。

## （2）多路复用器工作原理

### A、select

`select` 多路复用器是采用轮询的方式，一直在轮询所有的相关内核进程，查看它们的进程状态。若已经就绪，则马上将该内核进程放入到就绪队列。否则，继续查看下一个内核进程状态。在处理内核进程事务之前，`app` 进程首先会从内核空间中用户连接请求相关数据复制到用户空间。

该多路复用器的缺陷有以下几点：

- 对所有内核进程采用轮询方式效率会很低。因为对于大多数情况下，内核进程都不属于就绪状态，只有少部分才会是就绪态。所以这种轮询结果大多数都是无意义的。
- 由于就绪队列底层由数组实现，所以其所能处理的内核进程数量是有限制的，即其能够处理的最大并发连接数量是有限制的。
- 从内核空间到用户空间的复制，系统开销大。

### B、poll

`poll` 多路复用器的工作原理与 `select` 几乎相同，不同的是，由于其就绪队列由链表实现，所以，其对于要处理的内核进程数量理论上是没有限制的，即其能够处理的最大并发连接数量是没有限制的（当然，要受限于当前系统中进程可以打开的最大文件描述符数 `ulimit`，后面会讲到）。

### C、epoll

`epoll` 多路复用是对 `select` 与 `poll` 的增强与改进。其不再采用轮询方式了，而是采用回调方式实现对内核进程状态的获取：一旦内核进程就绪，其就会回调 `epoll` 多路复用器，进入到多路复用器的就绪队列（由链表实现）。所以 `epoll` 多路复用模型也称为 `epoll 事件驱动模型`。

另外，应用程序所使用的数据，也不再从内核空间复制到用户空间了，而是使用 `mmap` 零拷贝机制，大大降低了系统开销。

当内核进程就绪信息通知了 `epoll` 多路复用器后，多路复用器就会马上对其进行处理，将其马上存放到就绪队列吗？不是的。根据处理方式的不同，可以分为两种处理模式：`LT` 模式与 `ET` 模式。

## a、LT 模式

LT, Level Triggered, 水平触发模式。即只要内核进程的就绪通知由于某种原因暂时没有被 `epoll` 处理, 则该内核进程就会定时将其就绪信息通知 `epoll`。直到 `epoll` 将其写入到就绪队列, 或由于某种原因该内核进程又不再就绪而不再通知。其支持两种通讯方式: `BIO` 与 `NIO`。

## b、ET 模式

ET, Edge Triggered, 边缘触发模式。其仅支持 `NIO` 的通讯方式。

当内核进程的就绪信息仅会通知一次 `epoll`, 无论 `epoll` 是否处理该通知。明显该方式的效率要高于 LT 模式, 但其有可能会出就绪通知被忽视的情况, 即连接请求丢失的情况。

## 2.1.3 Nginx 的并发处理机制

一般情况下并发处理机制有三种: 多进程、多线程, 与异步机制。Nginx 对于并发的处理同时采用了三种机制。当然, 其异步机制使用的是异步非阻塞方式。

我们知道 Nginx 的进程分为两类: `master` 进程与 `worker` 进程。每个 `master` 进程可以生成多个 `worker` 进程, 所以其是多进程的。每个 `worker` 进程可以同时处理多个用户请求, 每个用户请求会由一个线程来处理, 所以其是多线程的。

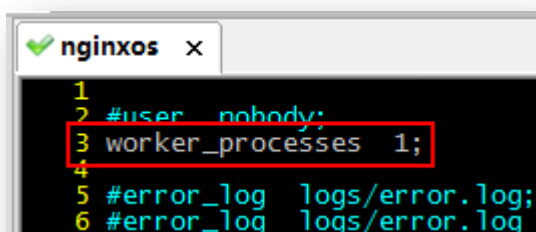
那么, 如何解释其“异步非阻塞”并发处理机制呢?

`worker` 进程采用的就是 `epoll` 多路复用机制来对后端服务器进行处理的。当后端服务器返回结果后, 后端服务器就会回调 `epoll` 多路复用器, 由多路复用器对相应的 `worker` 进程进行通知。此时, `worker` 进程就会挂起当前正在处理的事务, 拿 IO 返回结果去响应客户端请求。响应完毕后, 会再继续执行挂起的事务。这个过程就是“异步非阻塞”的。

## 2.1.4 全局模块下的调优

### (1) worker\_processes 2

打开 `nginx.conf` 配置文件, 可以看到 `worker_processes` 的默认值为 1。



`worker_processes`, 工作进程, 用于指定 Nginx 的工作进程数量。其数值一般设置为 CPU 内核数量, 或内核数量的整数倍。

不过需要注意, 该值不仅仅取决于 CPU 内核数量, 还与硬盘数量及负载均衡模式相关。在不确定时可以指定其值为 `auto`。



```

1 #user  nobody;
2
3
4 worker_processes  auto;
5
6 #error_log  logs/error.log;
  
```

## (2) worker\_cpu\_affinity 01 10

将 worker 进程与具体的内核进行绑定。不过，若指定 worker\_processes 的值为 auto，则无法设置 worker\_cpu\_affinity。

```

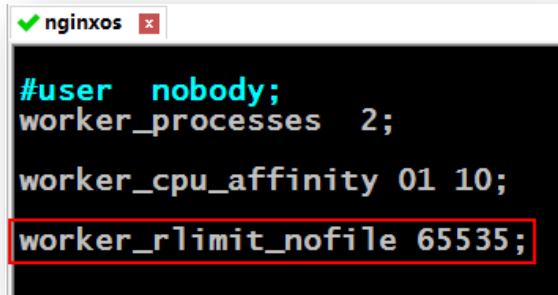
#user  nobody;
worker_processes  2;
worker_cpu_affinity 01 10;
  
```

该设置是通过二进制进行的。每个内核使用一个二进制位表示，0 代表内核关闭，1 代表内核开启。也就是说，有几个内核，就需要使用几个二进制位。

内核数量	worker_processes 工作进程数量	Worker_cpu_affinity取值	说明
2	2	01 10	每个进程各使用一个内核
2	4	01 10 01 10	每个进程交替使用各个内核
4	4	0001 0010 0100 1000	每个进程各使用一个内核
4	2	0101 1010	每个进程使用两个内核。对于需要CPU进行大量运算的应用，可以让每个进程使用多个CPU内核

### (3) worker\_rlimit\_nofile 65535

用于设置一个 worker 进程所能打开的最多文件数量。其默认值与当前 Linux 系统可以打开的最大文件描述符数量相同。



## 2.1.5 events 模块下的调优

### (1) worker\_connections 1024

设置每一个 worker 进程可以并发处理的最大连接数。该值不能超过 worker\_rlimit\_nofile 的值。

### (2) accept\_mutex on

- on: 默认值，表示当一个新连接到达时，那些没有处于工作状态的 worker 将以串行方式来处理；
- off: 表示当一个新连接到达时，所有的 worker 都会被唤醒，不过只有一个 worker 能获取新连接，其它的 worker 会重新进入阻塞状态，这就是“惊群”现象。

### (3) accept\_mutex\_delay 500ms

设置队首 worker 会尝试获取互斥锁的时间间隔。默认值为 500 毫秒。

### (4) multi\_accept on

- off: 系统会逐个拿出新连接按照负载均衡策略，将其分配给当前处理连接个数最少的 worker。
- on: 系统会实时的统计出各个 worker 当前正在处理的连接个数，然后会按照“缺编”最多的 worker 的“缺编”数量，一次性将这么多的新连接分配给该 worker。

## （5） use epoll

设置 worker 与客户端连接的处理方式。Nginx 会自动选择适合当前系统的最高效的方式。当然，也可以使用 use 指令明确指定所要使用的连接处理方式。user 的取值有以下几种：  
select | poll | epoll | rtsig | kqueue | /dev/poll 。

### A、 select | poll | epoll

这是三种多路复用机制。select 与 poll 工作原理几乎相同，而 epoll 的效率最高，是现在使用最多的一种多路复用机制。

### B、 rtsig

realtime signal，实时信号，Linux 2.2.19+的高效连接处理方式。但在 Linux 2.6 版本后，不再支持该方式。

### C、 kqueue

应用在 BSD 系统上的 epoll。

### D、 /dev/poll

UNIX 系统上使用的 poll。

## 2.1.6 http 模块下的调优

### （1） 非调优属性简介

```
http {
    include      mime.types;
    default_type application/octet-stream;
    charset      utf-8;
```

- include mime.types;  
将当前目录(conf 目录)中的 mime.types 文件包含进来。
- default\_type application/octet-stream;  
对于无扩展名的文件，默认其为 application/octet-stream 类型，即 Nginx 会将其作为一个八进制流文件来处理。

- `charset utf-8;`  
设置请求与响应的字符编码。

## (2) `sendfile on`

设置为 `on` 则开启 Linux 系统的零拷贝机制，否则不启用零拷贝。当然，开启后是否起作用，要看所使用的系统版本。CentOS6 及其以上版本支持 `sendfile` 零拷贝。

## (3) `tcp_nopush on`

- `on`: 以单独的数据包形式发送 Nginx 的响应头信息，而真正的响应体数据会再以数据包的形式发送，这个数据包中就不再包含响应头信息了。
- `off`: 默认值，响应头信息包含在每一个响应体数据包中。

## (4) `tcp_nodelay on`

- `on`: 不设置数据发送缓存，即不推迟发送，适合于传输小数据，无需缓存。
- `off`: 开启发送缓存。若传输的数据是图片等大数据量文件，则建议设置为 `off`。

## (5) `keepalive_timeout 60`

设置客户端与 Nginx 间所建立的长连接的生命超时时间，时间到达，则连接将自动关闭。单位秒。

## (6) `keepalive_requests 10000`

设置一个长连接最多可以发送的请求数。该值需要在真实环境下测试。

## (7) `client_body_timeout 10`

设置客户端获取 Nginx 响应的超时时限，即一个请求从客户端发出到接收到 Nginx 的响应的最长时间间隔。若超时，则认为本次请求失败。

## 2.2 请求定位

### 2.2.1 资源访问

#### (1) 修改配置文件

```

40
41     #access_log  logs/host.access.log  main;
42
43     location / {
44         root      html;
45         index     index.html index.htm;
46     }
47
48     location /xxx/ooo {
49         root      /opt/aaa;
50         index     myfile.txt;
51     }
52
53     #error_page   404             /404.html;
54

```

#### (2) 创建目录

在真实目录中，必须要在 root 属性指定的目录下存在 location 指定的 URI 路径目录。所以需要在/opt/aaa 下创建 xxx/ooo 目录。

```

nginxos x
[root@nginx05 ~]# mkdir -p /opt/aaa/xxx/ooo
[root@nginx05 ~]#

```

#### (3) 创建文件

在/opt/aaa/xxx/ooo 目录下新建一个 myfile.txt 文件，文件内容为：this default page。

```
✓ nginxos x
[root@nginx05 nginx]# echo "this default page" > /opt/aaa/xxx/ooo/myfile.txt
[root@nginx05 nginx]#
```

再新建一个 hello.txt 文件，文件内容为：hello nginx world

```
✓ nginxos x
[root@nginx05 nginx]# echo "hello nginx world" > /opt/aaa/xxx/ooo/hello.txt
[root@nginx05 nginx]#
```

## 2.2.2 路径匹配优先级

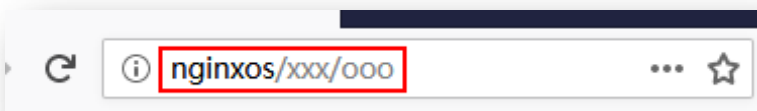
### (1) 优先级规则

优先级由低到高依次是：

普通匹配 < 长路径匹配 < 正则匹配 < 短路匹配 < 精确匹配

### (2) 普通匹配

浏览器地址栏中的访问路径均为如下形式，不变。



下面的匹配规则是：只要请求是以/xxx 开头的路径就可命中。

```
location /xxx {
    return 400;
}
```

## 400 Bad Request

nginx/1.14.1

### (3) 长路径匹配

当一个请求路径既可以与一个长路径相匹配，又可以与一个短路径相匹配时，长路径优先级高。

```
location /xxx {
    return 400;
}

location /xxx/ooo {
    return 402;
}
```

## 402 Payment Required

nginx/1.14.1

### (4) 正则匹配

在正则匹配与普通匹配（长路径匹配也属于普通匹配）均可匹配上时，正则匹配的优先级高。

#### A、区分大小写的正则匹配

~表示这里是正则表达式，默认匹配是区分大小写的。

在长路径匹配与正则匹配间，仍然是正则匹配的优先级要高于长路径匹配的，即使正则匹配的要短于长路径匹配的。

```
location /xxx {
    return 400;
}

location /xxx/ooo {
    return 402;
}

location ~xxx {
    return 401;
}
```

## 401 Authorization Required

nginx/1.14.1

当请求中的 xxx 写为大写字母，会报 404 找到资源。

## 404 Not Found

nginx/1.14.1

### B、不区分大小写的正则匹配

~后跟上\*号，表示这是不区分大小写的正则表达式。

```
location /xxx {
    return 400;
}

location ~*/xxx {
    return 401;
}
```



## 401 Authorization Required

nginx/1.14.1

当请求中的 xxx 写为大写字母时，依然可以访问。

## 401 Authorization Required

nginx/1.14.1

### (5) 短路匹配

以`^~`开头的匹配路径称为短路匹配，表示只要匹配上，就不再匹配其它的了，即使是正则匹配也不再匹配了。即其优先级要高于正则匹配的。

```
location /xxx {
    return 400;
}

#location /xxx/ooo {
#    return 402;
#}

location ~ /xxx {
    return 401;
}

location ^~ /xxx/ooo {
    return 403;
}
```

## 403 Forbidden

nginx/1.14.1

### (6) 精确匹配

以等号(=)开头的匹配称为精确匹配，其是优先级最高的匹配。

```
location /xxx {
    return 400;
}

#location /xxx/ooo {
#    return 402;
#}

location ~/xxx {
    return 401;
}

location ^~/xxx/ooo {
    return 403;
}

location =/xxx/ooo {
    return 405;
}
```

## 405 Not Allowed

nginx/1.14.1

### 2.2.3 缓存配置

Nginx 具有很强大的缓存功能，可以对请求的 response 进行缓存，起到类似 CDN 的作用，甚至有比 CDN 更强大的功能。同时，Nginx 缓存还可以用来“数据托底”，即当后台 web 服务

器挂掉的时候，Nginx 可以直接将缓存中的托底数据返回给用户。此功能就是 Nginx 实现“服务降级”的体现。

Nginx 缓存功能的配置由两部分构成：全局定义与局部定义。在 `http{}` 模块的全局部分中进行缓存全局定义，在 `server{}` 模块的各个 `location{}` 模块中根据业务需求进行缓存局部定义。

## （1）http{} 模块的缓存全局定义

```
http {
    include      mime.types;
    default_type application/octet-stream;
    charset      utf-8;

    proxy_cache_path /usr/local/nginx/cache levels=1:2
                    keys_zone=mycache:10m max_size=5g
                    inactive=2h use_temp_path=off;
    #proxy_temp_path proxy/cache;
```

### A、proxy\_cache\_path

用于指定 Nginx 缓存的存放路径及相关配置。

### B、proxy\_temp\_path

指定 Nginx 缓存的临时存放目录。若 `proxy_cache_path` 中的 `use_temp_path` 设置为了 `off`，则该属性可以不指定。

## （2）location{} 模块的缓存局部定义

### A、proxy\_cache mycache

指定用于存放缓存 key 内存区域名称。其值为 `http{}` 模块中 `proxy_cache_path` 中的 `keys_zone` 的值。

### B、proxy\_cache\_key \$host\$request\_uri\$args\_age

指定 Nginx 生成的缓存的 key 的组成。

### C、`proxy_cache_bypass $arg_age`

指定是否越过缓存。

### D、`proxy_cache_methods GET HEAD`

指定客户端请求的哪些提交方法将被缓存，默认为 GET 与 HEAD，但不缓存 POST。

### E、`proxy_no_cache $aaa $bbb $ccc`

指定对本次请求是否不做缓存。只要有一个不为 0，就不对该请求结果缓存。

### F、`proxy_cache_purge $ddd $eee $fff`

指定是否清除缓存 key。

### G、`proxy_cache_lock on`

指定是否采用互斥方式回源。

### H、`proxy_cache_lock_timeout 5s`

指定再次生成回源互斥锁的时限。

### I、`proxy_cache_valid 5s`

对指定的 HTTP 状态码的响应数据进行缓存，并指定缓存时间。默认指定的状态码为 200, 301, 302。

```
proxy_cache_valid 5s;  
proxy_cache_valid 403 24h;  
proxy_cache_valid 404 2h;  
proxy_cache_valid 500 502 2h;
```

### J、`proxy_cache_use_stale http_404 http_500`

设置启用托底缓存的条件。而一旦这里指定了相应的状态码，则前面 `proxy_cache_valid` 中指定的相应状态码所生成的缓存就变为了“托底缓存”。

## K、expires 3m

为请求的静态资源开启浏览器端的缓存。

## (3) Nginx 变量

### A、自定义变量

由于 Nginx 配置文件是 perl 脚本，所以其是可以使用如下方式自定义变量的。

```
set $aaa "hello";
set $bbb 0;

location /xxx/ooo {
    root /opt/aaa;
    index myfile.txt;
```

### B、内置变量

Nginx 中已经内置定义了很多变量，这些变量的意义如下：

\$args	请求中的参数;
\$binary_remote_addr	远程地址的二进制表示
\$body_bytes_sent	已发送的消息体字节数
\$content_length	HTTP 请求信息里的"Content-Length"
\$content_type	请求信息里的"Content-Type"
\$document_root	针对当前请求的根路径设置值
\$document_uri	与\$uri 相同
\$host	请求信息中的"Host"，如果请求中没有 Host 行，则等于设置的服务器名;
\$http_cookie	cookie 信息
\$http_referer	来源地址
\$http_user_agent	客户端代理信息
\$http_via	最后一个访问服务器的 ip 地址
\$http_x_forwarded_for	相当于网络访问路径。
\$limit_rate	对连接速率的限制
\$remote_addr	客户端地址
\$remote_port	客户端端口号
\$remote_user	客户端用户名，认证用
\$request	用户请求信息

\$request_body	用户请求主体
\$request_body_file	发往后端的本地文件名称
\$request_filename	当前请求的文件路径名
\$request_method	请求的方法，比如"GET"、"POST"等
\$request_uri	请求的 URI，带参数
\$server_addr	服务器地址，如果没有用 <code>listen</code> 指明服务器地址，使用这个变量将发起一次系统调用以取得地址(造成资源浪费)
\$server_name	请求到达的服务器名
\$server_port	请求到达的服务器端口号
\$server_protocol	请求的协议版本，"HTTP/1.0"或"HTTP/1.1"
\$uri	请求的 URI，可能和最初的值有不同，比如经过重定向之类的

## 2.3 Nginx 日志管理及自动切割

对于程序员、运维来说，日志非常得重要。通过日志可以查看到很多请求访问信息，及异常信息。Nginx 也提供了对日志的强大支持。

### 2.3.1 日志管理范围

首先，下面要讲的这些日志相关属性可以配置在任意模块。在不同的模块，记录的是不同请求的日志信息。即，日志记录的请求范围是不同的。Nginx 日志一般可以指定三个范围：`http{}`模块范围、`server{}`模块范围，与 `location{}`模块范围。

#### (1) `http{}`模块范围

只要有请求通过 `http` 协议访问该 Nginx，就会有日志信息写入到这里的日志文件。

```

http {
    include      mime.types;
    default_type application/octet-stream;

    log_format  main  '[${remote_addr}=${remote_user}=${time_local}]=${request}'
                      '[${status}=${body_bytes_sent}=${http_referer}]'
                      '[${http_user_agent}=${http_x_forwarded_for}]';

    access_log  logs/access.log  main buffer=64k;
    error_log   logs/myerror.log;
    open_log_file_cache max=1000 inactive=10s min_uses=2 valid=60s;

    sendfile    on;

```

## (2) server{}模块范围

只要有请求访问当前 Server，就会有日志信息写入到这里的日志文件。

```
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

    access_log  logs/host.access.log  main;
    error_log   logs/host.error.log;

    location / {
        root    html;
        index   index.html index.htm;
    }
}
```

## (3) location{}模拟范围

只要有请求访问当前 location，就会有日志信息写入到这里的日志文件。

```
location / {
    root    html;
    index   index.html index.htm;

    access_log logs/location.access.log main;
    error_log  logs/location.error.log;
}
```

### 2.3.2 日志管理指令

下面以 http{}模块下的日志为例来学习 Nginx 日志管理指令。

```

http {
    include      mime.types;
    default_type application/octet-stream;

    log_format   main '[$remote_addr]=$[remote_user]=$[time_local]=$[request]'
                     '[$status]=$[body_bytes_sent]=$[http_referer]'
                     '[$http_user_agent]=$[http_x_forwarded_for]';

    access_log   logs/access.log main buffer=64k;
    error_log    logs/myerror.log;
    open_log_file_cache max=1000 inactive=10s min_uses=2 valid=60s;

    sendfile     on;
    
```

Nginx 的日志分为两类：访问日志与错误日志。Nginx 整个系统的默认日志在生成预编译文件 makefile 时就已经默认给配置好了。当然，无论是访问日志还是错误日志，其默认路径与名称在 nginx.conf 中均是可以修改的。在配置文件中不仅定义了日志文件的路径及名称，还定义了日志格式。

```

Configuration summary
+ using system PCRE library
+ using system OpenSSL library
+ using system zlib library

nginx path prefix: "/usr/local/nginx"
nginx binary file: "/usr/local/nginx/sbin/nginx"
nginx modules path: "/usr/local/nginx/modules"
nginx configuration prefix: "/usr/local/nginx/conf"
nginx configuration file: "/usr/local/nginx/conf/nginx.conf"
nginx pid file: "/usr/local/nginx/logs/nginx.pid"
nginx error log file: "/usr/local/nginx/logs/error.log"
nginx http access log file: "/usr/local/nginx/logs/access.log"
nginx http client request body temporary files: "client_body_temp"
nginx http proxy temporary files: "proxy_temp"
nginx http fastcgi temporary files: "fastcgi_temp"
nginx http uwsgi temporary files: "uwsgi_temp"
nginx http scgi temporary files: "scgi_temp"

[root@01nginx nginx-1.12.2]#
    
```

## (1) log\_format

```

log_format   main '[$remote_addr]=$[remote_user]=$[time_local]=$[request]'
                 '[$status]=$[body_bytes_sent]=$[http_referer]'
                 '[$http_user_agent]=$[http_x_forwarded_for]';
    
```

用于设置访问日志的格式，其后的 main 是为该格式所起的名称，可以任意，而其后面的内容则为具体格式，通过 Nginx 内置变量定义。



- **\$remote\_addr**: 获取访问者的 IP 地址。若当前 Nginx 是反代服务器，则此变量获取到的就是客户端的 IP 地址；若当前 Nginx 是静态代理服务器，则此变量获取到的是反代服务器的 IP 地址。
- **\$http\_x\_forwarded\_for**: 获取客户端浏览器的 IP。若当前 Nginx 是反代服务器，则此变量获取到的值为杠(-)。若当前 Nginx 是静态代理服务器，则此变量获取到的是客户端的 IP 地址。
- **\$remote\_user**: 获取访问者的用户名。
- **\$time\_local**: 获取请求访问的时间与时区。
- **\$request**: 获取请求的相关信息，包含请求方式、请求的 URI，及访问协议。
- **\$status**: 后端服务器向其返回的状态码，例如 200。
- **\$body\_bytes\_sent**: 后端服务器向客户端发送的响应体内容字节数。
- **\$http\_referer**: 获取当前请求是从哪个页面过来的。其值在这里显示为杠(-)。
- **\$http\_user\_agent**: 用户所使用的代理，一般为浏览器。

## (2) access\_log

```
access_log logs/access.log main buffer=64k;
```

该指令用于设置访问日志。上面的格式包含三个参数：

- 第一个参数是日志的存放路径与日志文件名；
- 第二个参数是日志格式名；
- 第三个参数是日志文件所使用的缓存。不过，即使不指定 **buffer**，其也会存在默认日志缓存的。
- **access\_log** 还可以跟一个参数 **off**，用于关闭访问日志，即直接写 **access\_log off** 即可关闭访问日志。

## (3) error\_log

```
#error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;
```

该指令用于指定错误日志的路径与文件名。需要注意以下几点：

- 其不能指定格式，因为其有默认格式。
- 可以使用自己指定的错误日志文件，不过，将来的访问异常日志就不会再写入到默认的 **logs/error.log** 文件中了。所以关于错误日志，一般使用默认的即可。
- 错误日志级别由低到高有：**[debug | info | notice | warn | error | crit | alert | emerg]**，默

认为 error，级别越高记录的信息越少。

- 错误日志默认是开启的。关闭错误日志的写法为 `error_log /dev/null;`

#### (4) open\_log\_file\_cache

```
open_log_file_cache max=1000 inactive=10s min_uses=2 valid=60s;
```

该指令用于打开日志文件读缓存，将日志信息读取到缓存中，以加快对日志的访问。该功能默认为 off，即 `open_log_file_cache off;`

### 2.3.3 默认的/favicon.ico 请求

客户端对于服务端页面会自动提交一个/favicon.ico 请求，若没有 favicon.ico 文件则会在日志文件中报出 404。

从网上任意下载一个 ico 图标，将其重命名为 favicon.ico，然后放到 Linux 中的任意目录。然后再修改 nginx.conf 文件，在其中添加如下的 location{} 模块。注意，若将其添加到的位置与页面在同一个目录，下面的 location{} 模块不用设置。

```
location / {
    root    html;
    index  index.html index.htm;
}

location /favicon.ico {
    root html;
}
```

### 2.3.4 日志自动切割

这里仅仅简单介绍一下日志切割的实现步骤，具体实现，网上非常多，用时再查即可。

#### (1) 创建切割 shell 脚本文件

在 Linux 下创建一个实现日志切割的 shell 脚本文件，脚本文件的具体内容可以从网上查找，资源很多。例如，将该 shell 文件创建在 Nginx 安装目录下的 logs 目录中，并命名为 cut\_nginx\_log.sh。

## (2) 为该文件添加可执行权限

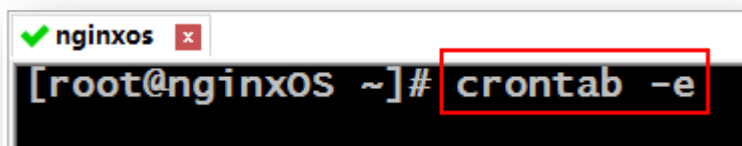
该文件是要作为定时任务被执行的，所以该文件需要具有可执行权限。

## (3) 向 crontab 中添加一个定时任务

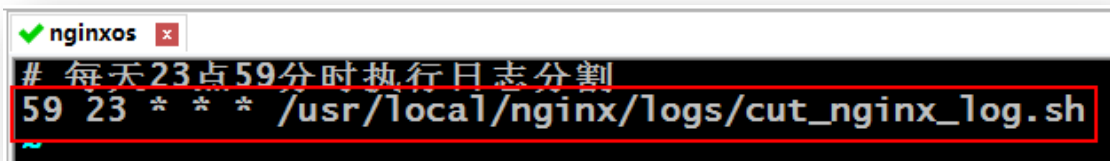
crontab 是 Linux 中的一个定义任务文件，每一行都代表一项定义任务。每行由 6 个字段组成，前 5 段是时间设定段，第 6 段是任务段。具体格式如下：

minute(0-59) hour(0-23) day(1-31) month(1-12) week(0-6) command

本例执行如下命令后会打开文本编辑器，然后再输入如下文本内容即可。



```
[root@nginxos ~]# crontab -e
```



```
# 每天23点59分时执行日志分割
59 23 * * * /usr/local/nginx/logs/cut_nginx_log.sh
```

## 2.4 静态代理

Nginx 静态代理是指，将所有的静态资源，例如，css、js、html、jpg 等资源存放到 Nginx 服务器，而不存放在应用服务器 Tomcat 中。当客户端发出的请求是对这些静态资源的请求时，Nginx 直接将静态资源响应给客户端，而无需提交给应用服务器处理。这样就减轻了应用服务器的压力。

## 2.4.1 扩展名拦截

### (1) 修改配置文件

```

40
41     #access_log  logs/host.access.log  main;
42
43     location / {
44         root      html;
45         index     index.html index.htm;
46     }
47
48     location /xxx/ooo {
49         root      /opt/aaa;
50         index     myfile.txt;
51     }
52
53     location /kkk/jjj {
54         root      /opt/bbb;
55     }
56
57     location ~ .*\. (css|js|html|jpg|png)$ {
58         root      /opt/statics;
59     }
60
61     #error_page  404              /404.html;
62

```

### (2) 创建目录

在/opt 目录中创建 statics 目录。

```

nginxos x
[root@nginxos ~]# mkdir /opt/statics
[root@nginxos ~]#

```

在/opt/statics 目录中创建 css、js、images 目录。

```

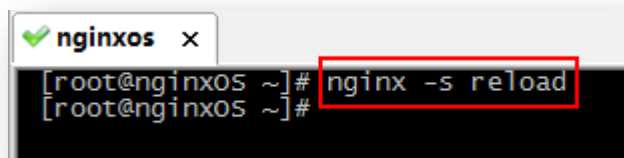
nginxos x
[root@nginxos statics]# mkdir css js images
[root@nginxos statics]#

```

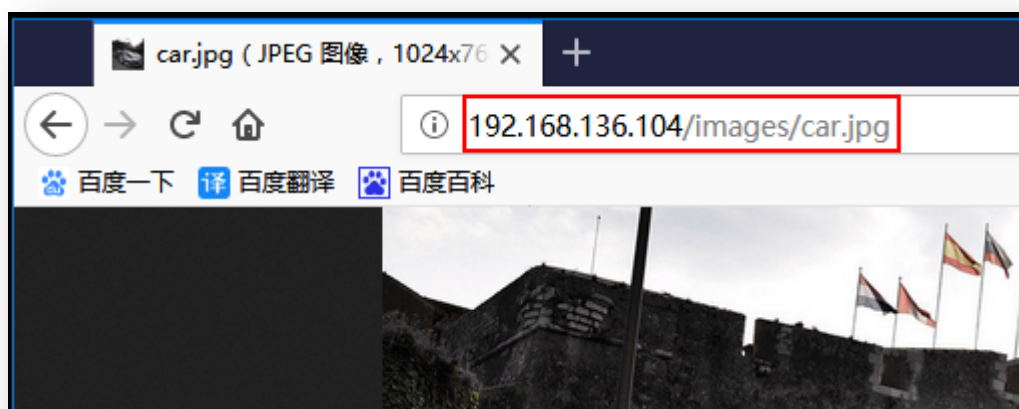
### (3) 上传图片

向/opt/statics/images 目录中上传一个图片 car.jpg。

### (4) 重启 Nginx



### (5) 浏览器访问



## 2.4.2 目录名拦截

### (1) 修改配置文件

```

52
53     #location ~.*\.(css|js|html|jpg|png)$ {
54     #     root /opt/statics;
55     #}
56
57     location ~.*(css|js|images|html).+ {
58         root /opt/statics;
59     }
60

```

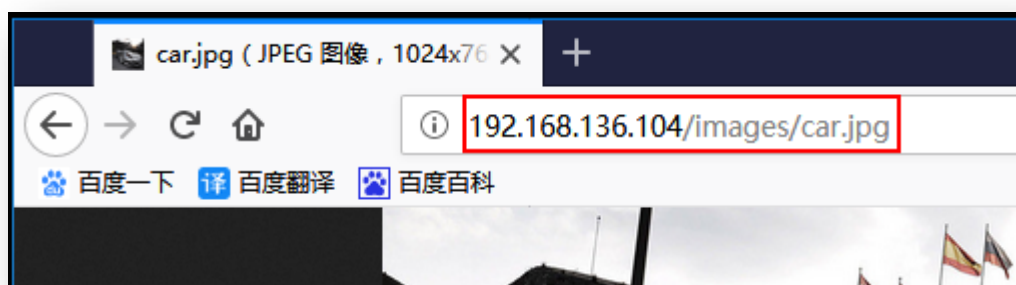
### (2) 重启 Nginx

```

nginxs x
[root@nginxs ~]# nginx -s reload
[root@nginxs ~]#

```

### (3) 浏览器访问



### 2.4.3 页面压缩

#### (1) 浏览器常见的压缩协议

浏览器中最常见的压缩算法有：

- **deflate**：是一种过时的压缩算法，是 huffman 编码的一种加强。
- **gzip**：是目前大多数浏览器都支持的一种压缩算法，是对 deflate 的改进。
- **sdch**：谷歌开发的一种压缩算法，一种全新的压缩思路。deflate 与 gzip 的的压缩思想是，修改传输数据的编码格式以达到减少体量的目的，其最终传输的数据并没有减少。而 sdch 压缩算法的思想是，让冗余的数据仅出现一次，其最终传输的数据减少了。
- **Zopfli**：谷歌开发的一种压缩算法，Deflate 压缩算法的改进。比标准的 gzip -9 要小 3%-8%，但压缩用时是 gzip -9 的 80 多倍。
- **br**：即 Brotli，谷歌开发的一种压缩算法，是一种全新的数据格式。与 Zopfli 相比，压缩率能够降低 20%-26%。Brotli -1 有着与 Gzip -9 相近的压缩比和更快的压缩解压速度。

#### (2) 常用设置

```
#keepalive_timeout 0;
keepalive_timeout 65;

gzip on;
gzip_min_length 5k;
gzip_comp_level 4;
gzip_buffers 4 16k;
gzip_vary on;
gzip_types text/html text/css text/xml application/x-javascript;

server {
    listen 80;
    server_name localhost;
```

##### A、gzip on;

开启 gzip 压缩，默认为 off。

##### B、gzip\_min\_length 5k;

指定最小启用压缩的文件大小。

### C、gzip\_comp\_level 4;

指定压缩级别，取值为 1-9，数字越大，压缩比越高，但压缩所用时间会越长。默认为 1，建议使用 4。

### D、gzip\_buffers 4 16k;

“4”表示的是缓存颗粒数量，而“16k”表示的是缓存颗粒大小。

### E、gzip\_vary on;

开启动态压缩。默认值 off。

### F、gzip\_types mimeType;

```
gzip_vary on;
gzip_types text/html text/css text/xml application/x-javascript;
```

通过 MIME 类型来指定要压缩的文件类型。默认值 text/html。

## 2.5 反向代理

通过在 location{} 中添加通行代理 proxy\_pass 可以指定当前 Nginx 所要代理的真正服务器。

### 2.5.1 反向代理 tomcat 服务器

#### (1) 定义一个 web 工程

定义一个 Maven Web 工程，并命名为 webdemo。其包含一个 JSP 页面，及一个 Servlet。

#### A、修改 pom.xml

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
```



```
<!-- Servlet 依赖 -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>

<!-- JSP 依赖 -->
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>javax.servlet.jsp-api</artifactId>
  <version>2.2.1</version>
  <scope>provided</scope>
</dependency>
</dependencies>
```

## B、定义 index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
  <title>webdemo</title>
</head>
<body>
  Nginx World Welcome You!<br>
  Nginx Addr = ${pageContext.request.remoteAddr} <br>
  Tomcat Addr = ${pageContext.request.localAddr} <br>
</body>
</html>
```

## C、定义 SomServlet

```
@WebServlet("/some")
public class SomeServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    PrintWriter writer = response.getWriter();
    writer.println("NginxIp = " + request.getRemoteAddr());
    writer.println("TomcatIp = " + request.getLocalAddr());
  }
}
```

```
}  
}
```

## (2) 克隆一台 Tomcat 并部署工程

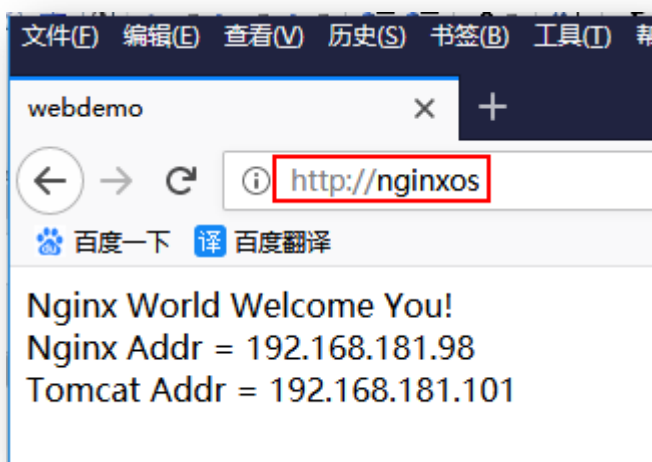
克隆一台 Tomcat 主机，将 webdemo 工程打包后部署到 Tomcat 的 webapps/ROOT 目录中。当然，需要首先将 ROOT 目录中的文件全部删除。

## (3) 修改 Nginx 配置文件

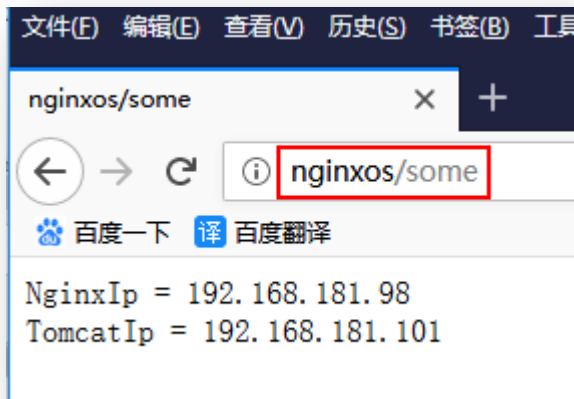
```
#location / {  
#    root    html;  
#    index  index.html index.htm;  
#}  
  
location ~.*(//some) {  
    proxy_pass http://192.168.181.101:8080;  
}
```

## (4) 访问

通过 Nginx 反向代理访问 Tomcat 主机的 index.jsp 页面。



通过 Nginx 反向代理访问 Tomcat 主机的 SomeServlet。



## 2.5.2 反向代理的属性设置

### (1) `client_max_body_size 100k;`

Nginx 允许客户端请求的单文件最大大小，单位字节。

### (2) `client_body_buffer_size 80k;`

Nginx 为客户端请求设置的缓存大小。

### (3) `proxy_buffering on`

开启从后端被代理服务器的响应内容缓冲区。默认值 `on`。

### (4) `proxy_buffers 4 8k;`

该指令用于设置缓冲区的数量与大小。从被代理的后端服务器取得的响应内容，会缓存到这里。

### (5) `proxy_busy_buffers_size 16k;`

高负荷下缓存大小，其默认值为一般为单个 `proxy_buffers` 的 2 倍。

## (6) proxy\_connect\_timeout 60s;

Nginx 跟后端服务器连接超时时间。默认 60 秒。

## (7) proxy\_read\_timeout 60s;

Nginx 发出请求后等待后端服务器响应的最长时限。默认 60 秒。

## 2.6 负载均衡

### 2.6.1 负载均衡简介

负载均衡，Load Balancing，就是将对请求的处理分摊到多个操作单元上进行。

### 2.6.2 负载均衡分类

#### (1) 软硬件分类

##### A、硬件负载均衡

硬件负载均衡器的性能稳定，且有生产厂商作为专业的服务团队。但其成本很高，一台硬件负载均衡器的价格一般都在十几万到几十万，甚至上百万。知名的负载均衡器有 F5、Array、深信服、梭子鱼等。



##### B、软件负载均衡

软件负载均衡成本几乎为零，基本都是开源软件。例如，LVS、HAProxy、Nginx 等。

## （2）负载均衡工作层分类

负载均衡就其所工作的 OSI 层次，在生产应用层面分为两类：七层负载均衡与四层负载均衡。当然，为四层负载均衡提供更为底层实现的，还有三层负载均衡与二层负载均衡。

- **七层负载均衡**：应用层，基于 HTTP 协议，通过虚拟 URL 将请求分配到真实服务器。一般应用于 B/S 架构系统。Nginx 就是七层负载均衡。
- **四层负载均衡**：传输层，基于 TCP 协议，通过“虚拟 IP + 端口号”将请求分配到真实服务器。一般应用于 C/S 架构系统。例如，LVS、F5、Nginx Plus 都属于四层负载均衡。
- 三层负载均衡：网络层，基于 IP 协议，通过虚拟 IP 将请求分配到真实服务器。
- 二层负载均衡：链路层，基于虚拟 MAC 地址将请求分配到真实服务器。

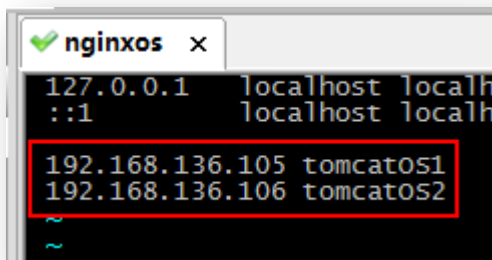
### 2.6.3 负载均衡的实现

#### （1）总体规划

该机群包含一台 Nginx 服务器，两台 Tomcat 服务器。将前面打过包的 web 工程直接部署到两台 Tomcat 主机上。然后，在 Nginx 服务器上设置对这两台 Tomcat 主机的负载均衡。

#### （2）配置 Nginx 主机

##### A、修改 Nginx 主机的 hosts 文件



## B、修改 Nginx 配置文件

```

32
33 #gzip on;
34
35 upstream www.xxx.com {
36     server oStomcat1:8080 weight=1;
37     server oStomcat2:8080 weight=1;
38 }
39
40
41 server {
42     listen      80;
43     server_name localhost;
44
45     #charset koi8-r;
46
47     #access_log logs/host.access.log main;
48
49     #location / {
50     #     root   html;
51     #     index  index.html index.htm;
52     #}
53
54     location ~.*(/some!/) {
55         proxy_pass http://www.xxx.com;
56     }
57
58     #error_page 404              /404.html;
59

```

### 2.6.4 Nginx 负载均衡策略

Nginx 内置了三种负载均衡策略，另外，其还支持第三方的负载均衡。而每种负载均衡主机根据负载均衡策略的不同，又可设置很多性能相关的属性。

#### (1) 轮询

默认的负载均衡策略，其是按照各个主机的权重比例依次进行请求分配的。

```

upstream tomcat.abc.com {
    server tomcat1:8080 weight=2 fail_timeout=20 max_fail=3 ;
    server tomcat1:8080 weight=1 fail_timeout=20 max_fail=3 ;
    server tomcat2:8080 backup;
    server tomcat2:8080 down;
}

```

- backup: 表示当前服务器为备用服务器。
- down: 表示当前服务器永久停机。

- fail\_timeout: 表示当前主机被 Nginx 认定为停机的最长失联时间，默认为 10 秒。常与 max\_fails 联合使用。
- max\_fails: 表示在 fail\_timeout 时间内最多允许的失败次数。

## (2) ip\_hash

指定负载均衡器按照基于客户端 IP 的分配方式。

```
upstream tomcat.abc.com { 指定使用ip_hash负载均衡策略
    ip_hash;
    server tomcat1:8080 weight=2 fail_timeout=20 max_fail=3 ;
    server tomcat1:8080 weight=1 fail_timeout=20 max_fail=3 ;
    server tomcat2:8080 weight=2 fail_timeout=20 max_fail=3 ;
    server tomcat2:8080 weight=1 fail_timeout=20 max_fail=3 ;
}
```

对于该策略需要注意以下几点：

- 在 nginx1.3.1 版本之前，该策略中不能指定 weight 属性。
- 该策略不能与 backup 同时使用。
- 此策略适合有状态服务，比如 session。
- 当有服务器宕机，必须手动指定 down 属性，否则请求仍是会落到该服务器。

## (3) least\_conn

把请求转发给连接数最少的服务器。

```
upstream tomcat.abc.com { 指定使用least_conn负载均衡策略
    least_conn;
    server tomcat1:8080 weight=2 fail_timeout=20 max_fail=3 ;
    server tomcat1:8080 weight=1 fail_timeout=20 max_fail=3 ;
    server tomcat2:8080 backup;
    server tomcat2:8080 down;
}
```

### 2.6.5 Nginx Plux 的四层负载均衡实现

同样是修改 nginx.conf 文件，添加一个 stream 模块，其与 events、http 等模块同级。在其中配置 upstream{} 与 server{} 模块。此时需要注意，通行代理配置在 server{} 中（前面的是配置在 server 模块的 location 模块中），且不能再是 http://开头的了，因为其负载均衡协议不再是 HTTP 协议了。

```

✓ nginxos x
// ....

events {
    // .....
}

stream {
    upstream app {
        server 192.168.188.53:1345;
        server 192.168.188.54:1345;
        server 192.168.188.55:1345;
    }

    server {
        listen 1345;
        proxy_pass app;
    }
}

http {
    // ....
}

```

## 2.7 动静分离

### 2.7.1 Nginx 动静分离的实现

下面要搭建的 Nginx 环境中有三台 Nginx 主机：一台用于完成负载均衡，两台用于存放项目中的静态资源。另外，还包含前面的两台 Tomcat 主机。

#### (1) 修改前面的 web 工程

在前面的 web 工程的 jsp 页面中添加一个背景图片，无需在工程中添加 images 目录及 bj.jpg 图片。因为这些都是要存放在 Nginx 静态代理服务器的。



```
<body background="images/bj.jpg">
  Nginx World Welcome You!<br>
  Nginx Addr = ${pageContext.request.remoteAddr} <br>
  Tomcat Addr = ${pageContext.request.localAddr} <br>
</body>
```

## (2) 复制并配置一台 Nginx 服务器

### A、修改 nginx.conf

```
40
41     #access_log logs/host.access.log main;
42
43     location / {
44         root    html;
45         index  index.html index.htm;
46     }
47
48     location ~.*\/(css|js|images) {
49         root /opt/statics;
50     }
51
52     #error_page 404              /404.html;
53
```

### B、添加静态资源

在/opt/statics 下创建 images 目录，并将 bj.jpg 存放到该目录。

```
✓ nginxos  ✓ nginxos1  ✕
[root@nginxOS1 images]# ll
总用量 80
-rw-r--r-- 1 root root 80116 12月 25 18:19 bj.jpg
[root@nginxOS1 images]#
```

### (3) 再复制一台 Nginx 服务器

以 nginxOS1 为母机，再复制一台 Nginx 主机，并命名为 nginxOS2。完成以下配置：

- 修改主机名：/etc/hostname
- 修改网络配置：/etc/sysconfig/network-scripts/ifcfg-ens33

### (4) 修改负载均衡 Nginx 主机

在 nginxOS 主机的 nginx.conf 文件中添加静态代理的负载均衡配置。

```

31 keepalive_timeout 65;
32
33 #gzip on;
34
35 upstream www.xxx.com {
36     server tomcatOS1:8080 weight=1;
37     server tomcatOS2:8080 weight=1;
38 }
39
40 upstream static.xxx.com {
41     server nginxOS1:80 weight=1;
42     server nginxOS2:80 weight=1;
43 }
44
45 server {
46     listen 80;
47     server_name localhost;
48
49     #charset koi8-r;
50
51     #access_log logs/host.access.log main;
52
53     #location / {
54     #    # root html;
55     #    #index index.html index.htm;
56     #}
57
58     location / {
59         proxy_pass http://www.xxx.com;
60     }
61
62     location ~.*/(css|js|images) {
63         proxy_pass http://static.xxx.com;
64     }
65
66     #error_page 404 /404.html;
67

```

## 2.7.2 项目的启动

### (1) 启动两台 Tomcat 服务器

```

nginxos  tomcatos1 x tomcatos2  nginxos1  n
[root@tomcatos1 ROOT]# startup.sh
Using CATALINA_BASE:   /usr/apps/tomcat
Using CATALINA_HOME:   /usr/apps/tomcat
Using CATALINA_TMPDIR: /usr/apps/tomcat/temp
Using JRE_HOME:        /usr
Using CLASSPATH:        /usr/apps/tomcat/bin/boot
Tomcat started.
[root@tomcatos1 ROOT]#

```

```

nginxos  tomcatos1  tomcatos2 x  nginxos1  ✓
[root@tomcatos2 ROOT]# startup.sh
Using CATALINA_BASE:   /usr/apps/tomcat
Using CATALINA_HOME:   /usr/apps/tomcat
Using CATALINA_TMPDIR: /usr/apps/tomcat/temp
Using JRE_HOME:        /usr
Using CLASSPATH:        /usr/apps/tomcat/bin/boc
Tomcat started.
[root@tomcatos2 ROOT]#

```

### (2) 启动两台 Nginx 静态代理服务器

```

nginxos  tomcatos1  tomcatos2  nginxos1 x  nginxos2
[root@nginxos1 ~]# nginx
[root@nginxos1 ~]#

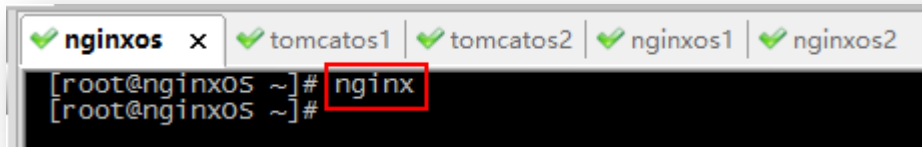
```

```

nginxos  tomcatos1  tomcatos2  nginxos1  nginxos2 x
[root@nginxos2 ~]# nginx
[root@nginxos2 ~]#

```

### (3) 启动 Nginx 负载均衡服务器



## 2.8 虚拟主机

### 2.8.1 总体规划

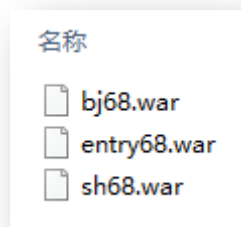
现在很多生活服务类网络平台都具有这样的功能：不同城市的用户可以打开不同城市专属的站点。用户首先打开的是平台总的站点，然后允许用户切换到不同的城市。其实，不同的城市都是一个不同的站点。

这里我们要实现的功能是为平台总站点，北京、上海两个城市站点分别创建一个虚拟主机。每一个虚拟主机都具有两台 Tomcat 的负载均衡主机。由于有三个站点，所以共需六台 Tomcat 主机，克隆 Tomcat 主机太过麻烦，所以这六台 Tomcat 我们使用一台主机实现。在一台主机中安装六个 Tomcat，它们分别使用六个不同的端口号。

首先要创建一个 web 工程，其中就一个 index.jsp 页面，页面除了显示当前城市外，还要显示城市切换的超链接。为了能够再明显的区分出当前访问的 Tomcat，再在页面中显示出当前工程所在的主机名与端口号。

### 2.8.2 创建三个 web 工程

直接复制前面的 web 工程，每个工程都仅需一个 JSP 即可。最终打为三个 war 包。三个工程的 JSP 页面内容各不相同。



## (1) 总站

注意，JSP 文件的 EL 中通过 request 获取的是 localAddr 与 localPort。

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
  <title>entry68</title>
</head>
<body>
  这是 68 平台总站<br>
  站点切换:
  <a href="http://bj.68.com">北京</a>
  <a href="http://sh.68.com">上海</a> <br>
  <hr>
  Tomcat Addr = ${pageContext.request.localAddr} <br>
  Tomcat Port = ${pageContext.request.localPort} <br>
</body>
</html>
```

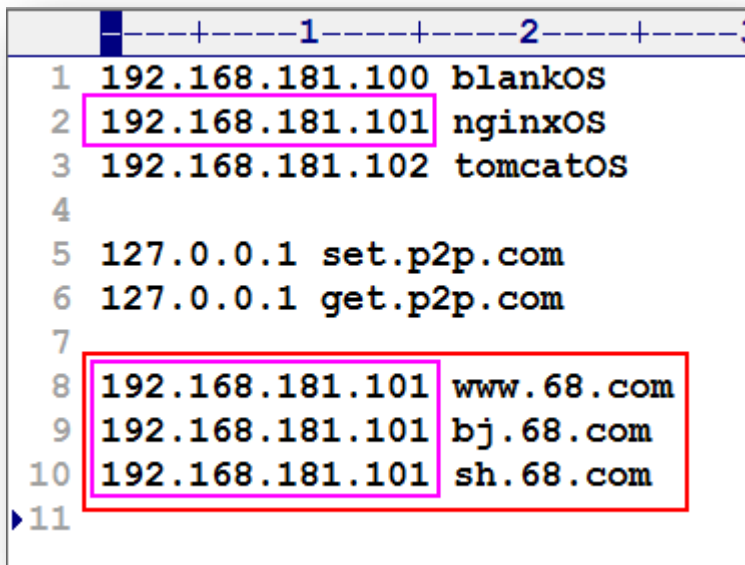
## (2) 北京站

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
  <title>bj68</title>
</head>
<body>
  这是 68 平台【北京】站<br>
  站点切换:
  <a href="http://www.68.com">总站</a>
  <a href="http://sh.68.com">上海</a> <br>
  <hr>
  Tomcat Addr = ${pageContext.request.localAddr} <br>
  Tomcat Port = ${pageContext.request.localPort} <br>
</body>
</html>
```

### (3) 上海站

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
  <title>sh68</title>
</head>
<body>
  这是 68 平台【上海】站<br>
  站点切换:
  <a href="http://www.68.com">总站</a>
  <a href="http://bj.68.com">北京</a> <br>
  <hr>
  Tomcat Addr = ${pageContext.request.localAddr} <br>
  Tomcat Port = ${pageContext.request.localPort} <br>
</body>
</html>
```

### 2.8.3 修改 hosts 文件



```

1 192.168.181.100 blankOS
2 192.168.181.101 nginxOS
3 192.168.181.102 tomcatOS
4
5 127.0.0.1 set.p2p.com
6 127.0.0.1 get.p2p.com
7
8 192.168.181.101 www.68.com
9 192.168.181.101 bj.68.com
10 192.168.181.101 sh.68.com
11
```

## 2.8.4 配置 Tomcat 主机

### (1) 项目部署规划

```

tomcatos6 x
[root@tomcat0S6 apps]# ll
总用量 0
drwxr-xr-x 9 root root 220 11月 10 16:45 tomcat8081
drwxr-xr-x 9 root root 220 11月 10 16:48 tomcat8082
drwxr-xr-x 9 root root 220 11月 10 16:49 tomcat8083
drwxr-xr-x 9 root root 220 11月 10 16:51 tomcat8084
drwxr-xr-x 9 root root 220 11月 10 16:52 tomcat8085
drwxr-xr-x 9 root root 220 11月 10 16:54 tomcat8086
[root@tomcat0S6 apps]#

```

第一、二台 Tomcat 中部署着 entry68.war 包；

第三、四台 Tomcat 中部署着 bj68.war 包；

第五、六台 Tomcat 中部署着 sh68.war 包。

### (2) 配置 Tomcat 主机

打开 Tomcat 安装目录下的 conf/server.xml 文件，修改三处端口号。

```

18 <!-- Note: A "Server" is not itself a "Container", so you may not
19      define subcomponents such as "Valves" at this level.
20      Documentation at /docs/config/server.html
21 -->
22 <Server port="8006" shutdown="SHUTDOWN">
23   <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
24   <!-- Security listener. Documentation at /docs/config/listeners.html
25   <Listener className="org.apache.catalina.security.SecurityListener" />
26 -->

```

```

66      APR (HTTP/AJP) Connector: /docs/apr.html
67      Define a non-SSL/TLS HTTP/1.1 Connector on port 80
68 -->
69 <Connector port="8081" protocol="HTTP/1.1"
70      connectionTimeout="20000"
71      redirectPort="8443" />
72 <!-- A "Connector" using the shared thread pool-->
73 <!--
74 <Connector executor="tomcatThreadPool"
75      port="8080" protocol="HTTP/1.1"

```

```

112 </Connector>
113 -->
114
115 <!-- Define an AJP/1.3 Connector on port 8009 -->
116 <Connector port="8019" protocol="AJP/1.3" redirectPort="8443" />
117
118 <!-- An Engine represents the entry point (within Catalina) that
119      handles requests. The Engine implementation for Tomcat stand
120

```

## 2.8.5 配置虚拟主机

### (1) 直接配置到 `nginx.conf` 中

修改 `nginx` 的核心配置文件，在其中添加如下内容：

```

27     sendfile        on;
28     #tcp_nopush     on;
29
30     #keepalive_timeout 0;
31     keepalive_timeout 65;
32
33     #gzip on;
34
35
36     upstream www.68.com {
37         server tomcat05:8080 weight=1;
38         server tomcat05:8081 weight=1;
39     }
40
41     upstream bj.68.com {
42         server tomcat05:8082 weight=1;
43         server tomcat05:8083 weight=1;
44     }
45
46     upstream sh.68.com {
47         server tomcat05:8084 weight=1;
48         server tomcat05:8085 weight=1;
49     }
50

```



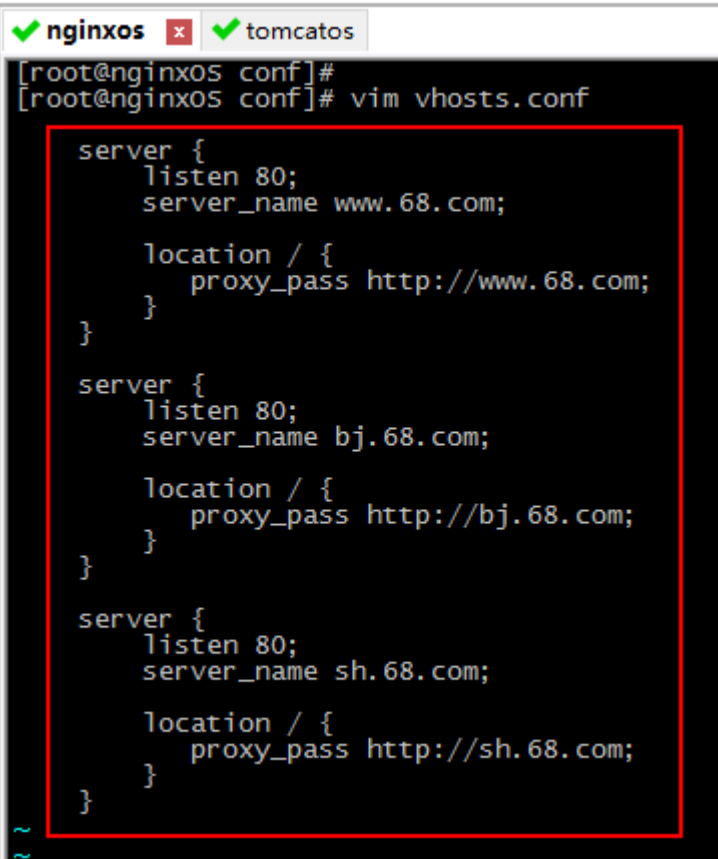
```

50
51     server {
52         listen 80;
53         server_name www.68.com;
54
55         location / {
56             proxy_pass http://www.68.com;
57         }
58     }
59
60     server {
61         listen 80;
62         server_name bj.68.com;
63
64         location / {
65             proxy_pass http://bj.68.com;
66         }
67     }
68
69     server {
70         listen 80;
71         server_name sh.68.com;
72
73         location / {
74             proxy_pass http://sh.68.com;
75         }
76     }
77
78     server {
79         listen      80;
80         server_name localhost;
81
82         #charset koi8-r;
83

```

## (2) 单独配置到一个文件

### A、定义 vhosts.conf 文件



```
[root@nginx05 conf]#
[root@nginx05 conf]# vim vhosts.conf

server {
    listen 80;
    server_name www.68.com;

    location / {
        proxy_pass http://www.68.com;
    }
}

server {
    listen 80;
    server_name bj.68.com;

    location / {
        proxy_pass http://bj.68.com;
    }
}

server {
    listen 80;
    server_name sh.68.com;

    location / {
        proxy_pass http://sh.68.com;
    }
}

~
```

## B、修改 nginx.conf 文件

```
#keepalive_timeout 0;
keepalive_timeout 65;

#gzip on;

upstream www.68.com {
    server tomcat05:8080 weight=1;
    server tomcat05:8081 weight=1;
}

upstream bj.68.com {
    server tomcat05:8082 weight=1;
    server tomcat05:8083 weight=1;
}

upstream sh.68.com {
    server tomcat05:8084 weight=1;
    server tomcat05:8085 weight=1;
}

include /usr/local/nginx/conf/vhosts.conf;

server {
    listen 80;
    server_name localhost;
```