第1章 研究Nginx前的准备工作

1.1 Nginx是什么

2012年，Nginx荣获年度云计算开发奖，并成长为世界第二大**Web**服务器。全世界流量最高的前1000名网站中，超过25% 都使用Nginx来处理海量的互联网请求。Nginx已经成为业界**高性能Web服务器**的代名词。

在学习Nginx时，先来看看Nginx的竞争对手——Apache、Lighttpd、 Tomcat、Jetty、IIS，它们都是Web服务器，或者叫做WWW（World Wide Web）服务器， 相应地也都具备Web服务器的基本功能：基于REST架构风格，以统一资源描述符 （Uniform Resource Identifier，URI）或者统一资源定位符（Uniform Resource Locator， URL）作为沟通依据，通过HTTP为浏览器等客户端程序提供各种网络服务。然而，由于这些Web服务器在设计阶段就受到许多局限，例如当时的互联网用户规模、网络带宽、产品特点等局限，并且各自的定位与发展方向都不尽相同，使得每一款Web服务器的特点与应用场合都很鲜明。

Nginx 作为一个基于 C 实现的高性能 Web 服务器，可以通过系列算法解决上述的负载均衡问题。并且由于它具有高并发、高可靠性、高扩展性、开源等特点，成为开发人员常用的**反向代理工具**。

1.2 为什么选择Nginx

Nginx具有以下特点：

（1）更快

第一，在正常情况下，单次请求会得到更快的响应。第二，在同时有数以百万的请求时的高峰期，Nginx会比其他web服务器更快的响应。

（2）高扩展性

Nginx的设计极具扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极低的模块组成。可以处理某一个模块而无需在意其他模块。这种低耦合度的优秀设计，造就了Nginx庞大的第三方模块。

（3）高可靠性

高可靠是我们选择Nginx的最基本条件，Nginx的高可靠性来自于其核心框架代码的优秀设计、模块设计的简单。每个worker进程相对独立，master进程在1个worker进程出错时可以快速拉起新的worker子进程提供服务。

（4）低内存消耗

一般情况下，10000个非活跃的HTTP Keep-Alive连接在Nginx中消耗2.5MB的内存。

（5）单机支持10万以上的并发连接

随着互联网的迅猛发展和互联网用户数量的成倍增长，各大公司、网站都需要应付海量并发请求，一个能够在峰值期顶住10万以上并发请求的 Server，无疑会得到大家的青睐。理论上，Nginx支持的并发连接上限取决于内存，10万远未封顶。

（6）热部署

master管理进程与worker工作进程的分离设计，使得Nginx能够提供热部署功能，即可 以在7×24小时不间断服务的前提下，升级Nginx的可执行文件。当然，它也支持不停止服 务就更新配置项、更换日志文件等功能。

（7）最自由的BSD许可协议

BSD许可协议（*BSD开源协议是一个给予使用者很大自由的协议。基本上使用者可以"为所欲为",可以自由的使用，修改源代码*）不只是允许用户免费使用Nginx， 它还允许用户在自己的项目中直接使用或修改Nginx源码，然后发布。这吸引了无数开发者继续为Nginx贡献自己的智慧。

当然，选择Nginx的核心理由还是它能在支持高并发请求的同时保持高效的服务。

1.3 准备工作

1.3.1 Linux操作系统

我们需要一个内核在Linux 2.6及以上版本的操作系统，因为Linux 2.6及以上内核才支持epoll。

我们使用uname-a命令查询Linux内核版本：

```
root@wudawei-virtual-machine:~# uname -a
Linux wudawei-virtual-machine 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

本机内核版本是4.15.0，符合要求。

1.3.2 使用Nginx的必备软件

如果要使用Nginx的常用功能，那么首先需要确保该操作系统上至少安装了如下软件：

（1）GCC编译器

GCC编译器可以用来编译C语言程序，Ubuntu下安装GCC：

```
apt-get install gcc
```

（2）PCRE库

如果我们在配置文件nginx.conf里使用了正则表达式，那么在编译Nginx时就必须把 PCRE库编译进Nginx，因为Nginx的HTTP模块要靠它来解析正则表达式。

```
apt-get install libpcre3 libpcre3-dev
```

（3）zlib库

zlib库用于对HTTP包的内容做gzip格式的压缩，如果我们在nginx.conf里配置了gzip on，并指定对于某些类型（content-type）的HTTP响应使用gzip来进行压缩以减少网络传输量，那么，在编译时就必须把zlib编译进Nginx。

```
apt-get install zlib1g zlib1g-dev
```

（4）OpenSSL开发库

如果我们的服务器不只是要支持HTTP，还需要在更安全的SSL协议上传输HTTP，那么就需要拥有OpenSSL了。

```
apt-get install openssl openssl-dev
```

1.3.3 磁盘目录

要是用Nginx，还需要在Linux文件系统上准备以下目录

（1）Nginx源代码存放目录

该目录用于放置从官网上下载的源代码文件。

（2）Nginx编译阶段产生的中间文件存放目录

该目录用于放置在configure命令执行后所生成的源文件及目录，以及make命令执行后 生成的目标文件和最终连接成功的二进制文件。

（3）部署目录

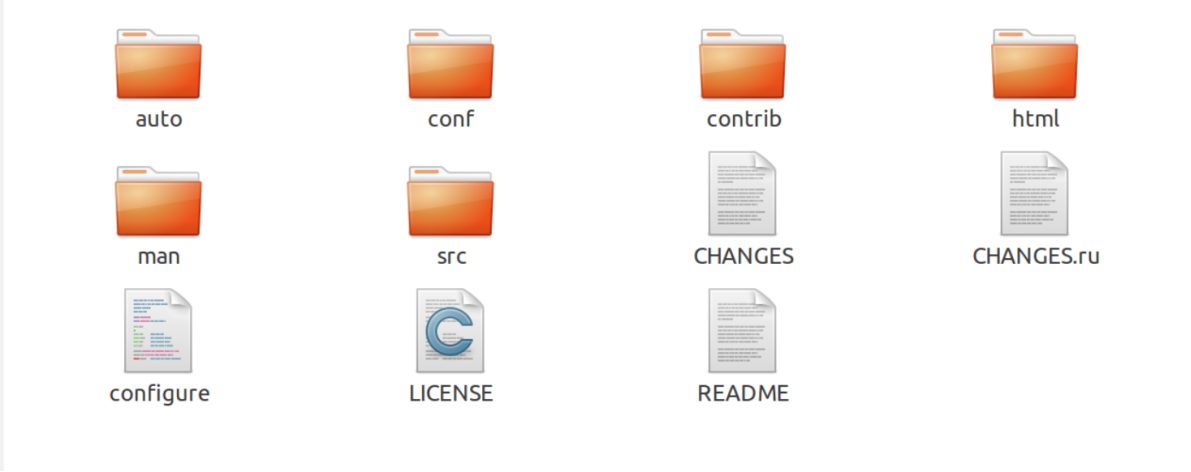
该目录存放实际Nginx服务运行期间所需要的二进制文件。

（4）日志文件存放目录

日志文件通常会比较大，当研究Nginx的底层架构时，需要打开debug级别的日志，这 个级别的日志非常详细，会导致日志文件的大小增长得极快，需要预先分配一个拥有更大磁盘空间的目录。

1.3.4 获取Nginx源码

可以在Nginx官方获取源码包，将下载的nginx源码压缩包放置到准备好的Nginx源代码目录中，然后解压。



1.4 编译安装Nginx

安装Nginx最简单的方式就是，进入nginx目录后执行以下三行命令

```
./configure
make
make install
```

configure命令做了大量的“幕后”工作，包括检测操作系统内核和已经安装的软件，参数的解析，中间目录的生成以及根据各种参数生成一些C源码文件、Makefile文件等。make命令根据configure命令生成的Makefile文件编译Nginx工程，并生成目标文件、最终的二进制文件。make install命令根据configure执行时的参数将Nginx部署到指定的安装目录，包括相关目录的建立和二进制文件、配置文件的复制。

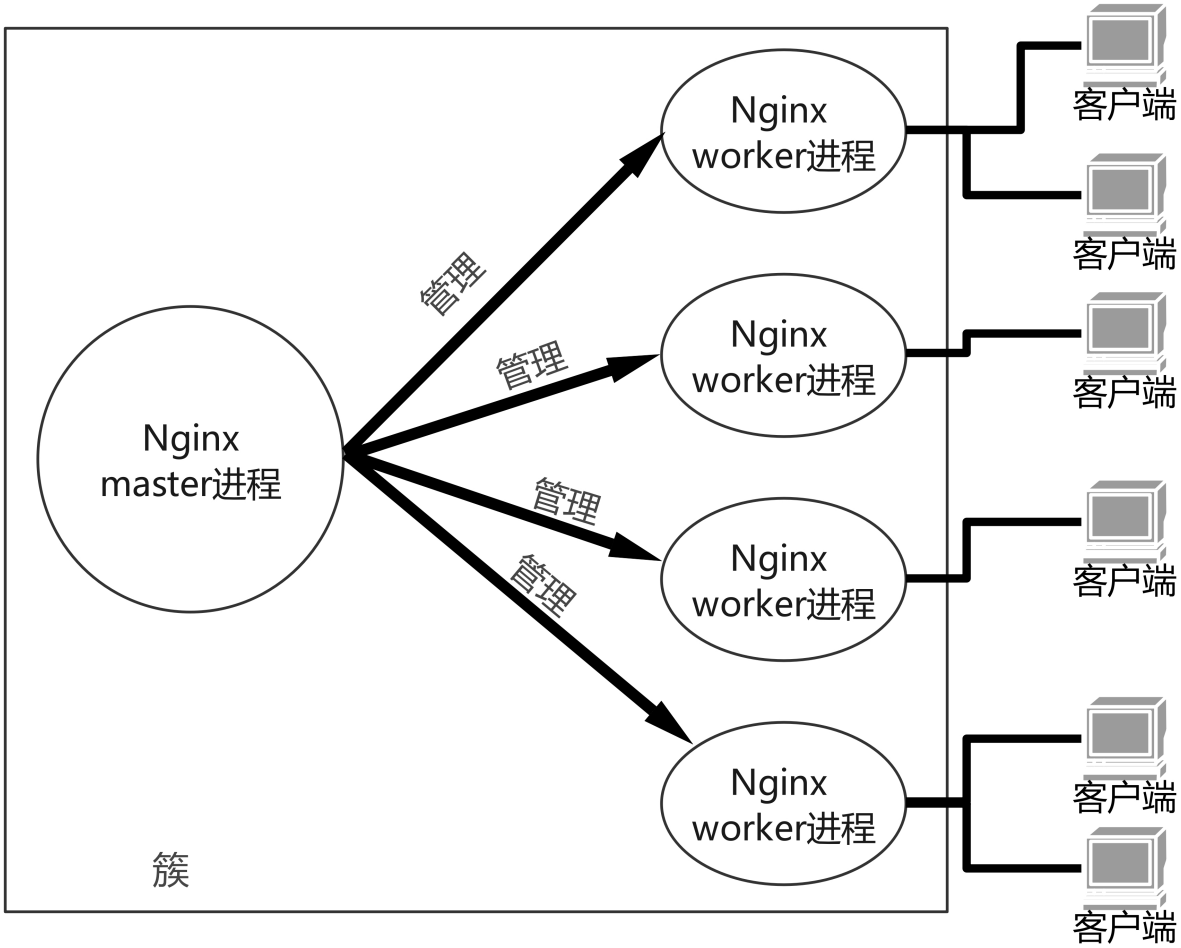
安装成功后，打开网页，输入ip地址，就可以看到：

第2章 Nginx的配置

Nginx有着许多的官方模块和第三方模块，这些已有的模块可以帮我们实现很多的Web服务器的功能。而在使用这些模块时，只需要修改一些配置即可。因此，我们需要学习一下Nginx的配置文件，包括配置文件的语法格式、运行所以有Nginx服务必须具备的基础配置以及使用HTTP核心模块配置静态Web服务器的方法。

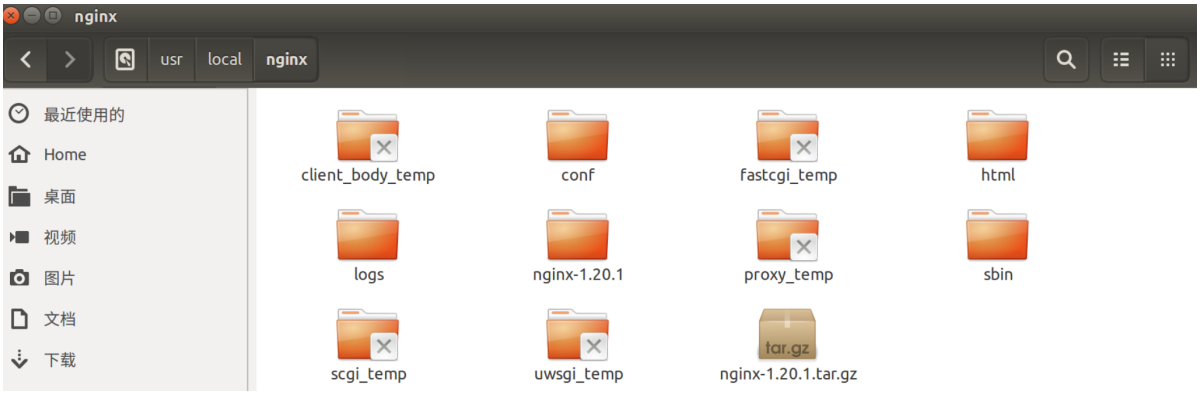
2.1 运行中的Nginx进程间的关系

在正式提供服务的产品环境下，部署Nginx时都是使用一个master进程来管理多个 worker进程，一般情况下，worker进程的数量与服务器上的CPU核心数相等。worker进程之间通过共享内存、原子操作等一些进程间通信机制来 实现负载均衡等功能。



为什么要把worker进程数量设置得与CPU核心数量一致呢？这正是Nginx与Apache服务器的不同之处。在Apache上每个进程在一个时刻只处理一个请求，因此，如果希望Web服务器拥有并发处理的请求数更多，就要把Apache的进程或线程数设置得更多，通常会达到一台服务器拥有几百个工作进程，这样大量的进程间切换将带来无谓的系统资源消耗。而Nginx则不然，一个worker进程可以同时处理的请求数只受限于内存大小，而且在架构设计上，不同的worker进程之间处理并发请求时几乎没有同步锁的限制，worker进程通常不会进入睡眠状态，因此，当Nginx上的进程数与CPU核心数相等时（最好每一个worker进程都绑定特定的CPU核心），进程间切换的代价是最小的。

默认设置情况下，Nginx的运行目录为/usr/local/nginx，相关目录如下：



```
|---sbin
|    |---nginx
|---conf
|    |---koi-win
|    |---koi-utf
|    |---win-utf
|    |---mime.types
|    |---mime.types.default
|    |---fastcgi_params
|    |---fastcgi_params.default
|    |---fastcgi.conf
|    |---fastcgi.conf.default
|    |---uwsgi_params
|    |---uwsgi_params.default
|    |---scgi_params
|    |---scgi_params.default
|    |---nginx.conf
|    |---nginx.conf.default
|---logs
|    |---error.log
|    |---access.log
|    |---nginx.pid
|---html
|    |---50x.html
|    |---index.html
|---client_body_temp
|---proxy_temp
|---fastcgi_temp
|---uwsgi_temp
|---scgi_temp
```

2.2 Nginx配置的通用语法

2.2.1 配置项的语法格式

Nginx的配置文件其实是一个普通的文本文件。最基本的配置项语法格式如下：

配置项名

配置项值

1 配置项值

2 ...

;

首先，在行首的是配置项名，这些配置项名必须是Nginx的某一个模块想要处理的，否则Nginx会认为配置文件出现了非法的配置项名。配置项名输入结束后，将以空格作为分隔符。

其次是配置项值，它可以是数字或字符串（当然也包括正则表达式）。针对一个配置项，既可以只有一个值，也可以包含多个值，配置项值之间仍然由空格符来分隔。当然，一个配置项对应的值究竟有多少个，取决于解析这个配置项的模块。我们必须根据某个Nginx模块对一个配置项的约定来更改配置项。

最后，每行配置的结尾要加上分号。

- 注意，如果配置项值中包括语法符号，比如空格符，那么需要使用单引号或双引号括住配置项值，否则Nginx会报语法错误。例如：

```
log_format main '$remote_addr - $remote_user [$time_local] "$request" ';
```

2.2.2 配置项的注释

如果有一个配置项暂时需要注释掉，可以用“#”注释掉这一行。例如：

```
#pid logs/nginx.pid;
```

2.2.3 配置项的单位

大部分模块遵循一些通用的规定，如指定空间大小时不用每次都定义到字节、指定时间时不用精确到毫秒。

当指定空间大小时，可以使用的单位包括：

- K或者k千字节（KiloByte，KB）
- M或者m兆字节（MegaByte，MB）

例如：

```
gzip_buffers 4 8k;
client_max_body_size 64M;
```

当指定时间时，可以适用的单位包括：

- ms（毫秒）
- s（秒）
- m（分钟）
- h（小时）
- d（天），
- w（周，包含7天）
- M（月，包含30天）
- y（年，包含365天）

例如：

```
expires 10y;
proxy_read_timeout 600;
client_body_timeout 2m;
```

2.2.4 在配置中使用变量

有些模块允许在配置项中使用变量，例如：

```
log_format main '$remote_addr - $remote_user [$time_local] "$request" '
'$status $bytes_sent "$http_referer" '
'"$http_user_agent" "$http_x_forwarded_for"';
```

其中，remote_addr是一个变量，使用它的时候前面要加$符号。

- 需要注意，这种变量只有少数模块支持，不是通用的。
- 事实上，在执行configure命令时（安装Nginx时），已经将许多模块编译进Nginx中了，但是否启用一般取决于配置文件中相应的配置项。每个Nginx模块都有自己感兴趣的配置项，大部分模块都必须在nginx.conf中读取某个配置项后才会在运行时启用。

例如，当配置http..这个配置项时，ngx_http_module模块才会启动，其他依赖这个模块的模块也会正常使用。

2.3 Nginx服务的基本配置

Nginx在运行时，必须加载几个核心模块和一个事件类模块。这些模块运行时所支持的配置项称为基本配置——所有其他模块执行时都依赖的配置项。

下面详述基本配置项的用法。由于配置项较多，所以把它们按照用户使用时的预期功能分成了以下4类：

- 用于调试、定位问题的配置项。
- 正常运行必备的配置项。
- 优化性能的配置项。
- 事件类的配置项（有些事件类配置项归纳到优化性能类，这是因为它们虽然也属于 events{} 块，但作用是优化性能）。

2.3.1 用于调试进程和定位问题的配置项

先来看一下用于调试进程、定位问题的配置项，如下所示：

（1）是否以守护进程方式运行Nginx

```
语法：daemon on|off;
默认：daemon on;
```

守护进程（daemon）是脱离终端并且在后台运行的进程。它脱离终端是为了避免进程执行过程中的信息在任何终端上显示，这样一来，进程也不会被任何终端所产生的信息所打断。Nginx毫无疑问是一个需要以守护进程方式运行的服务，因此，默认都是以这种方式运行的。

（2）是否以master/worker方式工作

```
语法：master_process on|off;
默认：master_process on;
```

（3）error日志的设置

```
语法：error_log/path/file level;
默认：error_log logs/error.log error;
```

error日志是定位Nginx问题的最佳工具，我们可以根据自己的需求妥善设置error日志的路径和级别。

- /path/file参数可以是一个具体的文件，例如，默认情况下是logs/error.log文件；/path/file也可以是/dev/null，这样就不会输出任何日志了，这也是关闭error日志的唯一手段；/path/file也可以是stderr，这样日志会输出到标准错误文件中。
- level是日志的输出级别，取值范围是debug、info、notice、warn、error、crit、alert、emerg，从左至右级别依次增大。当设定为一个级别时，大于或等于该级别的日志都会被输出到/path/file文件中，小于该级别的日志则不会输出。（如果设定的日志级别是debug，则会输出所有的日志，这样数据量会很大）

（4）是否处理几个特殊的调试点

```
语法：debug_points[stop|abort];
```

如果设置了debug_points为stop，那么Nginx的代码执行到这些调试点时就会发出SIGSTOP信号以用于调试。如果debug_points设置为abort，则会产生一个coredump文件，可以使用gdb来查看Nginx当时的各种信息。

通常不会使用这个配置项。

（5）仅对指定的客户端输出debug级别的日志

```
语法：debug_connection[IP|CIDR];
```

这个配置项实际上属于事件类配置，因此，它必须放在events{...}中才有效。它的值可以是IP地址或CIDR地址，例如：

```
events {
debug_connection 10.224.66.14;
debug_connection 10.224.57.0/24;
}
```

这样，仅仅来自以上IP地址的请求才会输出debug级别的日志，其他请求仍然沿用error_log中配置的日志级别。

（6）限制coredump核心转储文件的大小

```
语法：worker_rlimit_core size;
```

在Linux系统中，当进程发生错误或收到信号而终止时，系统会将进程执行时的内存内容（核心映像）写入一个文件（core文件），以作为调试之用，这就是所谓的核心转储（core dumps）。

当Nginx进程出现一些非法操作导致进程直接被操作系统强制结束时，会生成核心转储core文件，可以从core文件获取当时的堆栈、寄存器等信息，从而帮助我们定位问题。但这种文件可能会很大，需要限制内存。通过worker_rlimit_core可以限制core文件的大小。

我们还可以通过working_directory path来指定coredump文件放置的目录。

2.3.2 正常运行的配置项

正常运行的配置项如下：

（1）定义环境变量

```
语法：env VAR|VAR=VALUE;
```

这个配置项可以让用户直接设置操作系统上的环境变量。例如：

```
env TESTPATH=/tmp/;
```

（2）嵌入其他配置文件

```
语法：include/path/file;
```

include配置项可以将其他配置文件嵌入到当前的nginx.conf文件中，

（3）pid文件的路径

```
语法：pid path/file;
默认：pid logs/nginx.pid;
```

保存master进程ID的pid文件存放路径。默认与configure执行时的参数“--pid-path”所指定的路径是相同的，也可以随时修改，但应确保Nginx有权在相应的目标中创建pid文件， 该文件直接影响Nginx是否可以运行。

（4）Nginx worker进程运行的用户及用户组

```
语法：user username[groupname];
默认：user nobody nody;
```

user用于设置master进程启动后，fork出的worker进程运行在哪个用户和用户组下。当按照“user username;”设置时，用户组名与用户名相同。

（5）指定Nginx worker进程可以打开的最大句柄描述符个数

```
语法：worker_rlimit_nofile limit;
```

设置一个worker进程可以打开的最大文件句柄数。

（6）限制信号队列

```
语法：worker_rlimit_sigpending limit;
```

设置每个用户发往Nginx的信号队列的大小，超出的信息会丢掉。

2.3.3 优化性能的配置项

有关于优化性能的配置项如下：

（1）Nginx worker进程个数

```
语法：worker_processes number;
默认：worker_processes 1;
```

在master/worker运行方式下，定义worker进程的个数。

每个worker进程都是单线程的进程，它们会调用各个模块以实现多种多样的功能。如果这些模块确认不会出现阻塞式的调用，那么，有多少CPU内核就应该配置多少个进程。

如果worker进程的数量多于CPU内核数，会增大进程间切换带来的消耗（Linux是抢占式内核）。一般情况下，用户要配置与CPU内核数相等的worker进程，并且使用下面的worker_cpu_affinity配置来绑定CPU内核。

（2）绑定Nginx worker进程到指定的CPU内核

```
语法：worker_cpu_affinity cpumaskp[cpumask...];
```

为什么要绑定worker进程到指定的CPU内核呢？假定每一个worker进程都是非常繁忙的，如果多个worker进程都在抢同一个CPU，那么这就会出现同步问题。反之，如果每一 个worker进程都独享一个CPU，就在内核的调度策略上实现了完全的并发。

（3）SSL硬件加速

```
语法：ssl_engine device;
```

如果服务器上有SSL硬件加速设备，那么就可以进行配置以加快SSL协议的处理速度。

（4）系统调用gettimeofday的执行频率

```
语法：timer_resolution t;
```

每次内核的事件调用，如epoll、select、poll、kqueue等返回时，都会执行一次gettimeofday，实现用内核的时钟来更新Nginx中的缓存时钟。当需要降低gettimeofday的调用频率时，可以使用这个命令

```
例如：timer_resolution 100ms;
```

表示每100ms才调用一次gettimeofday。

但在目前的大多数内核中，仅仅对共享内存页中的数据做访问，并不是通常的系统调用，代价并不大，一般不必使用这个配置。

（5）Nginx worker进程优先级设置

```
语法：worker_priority nice;
默认：worker_priority 0;
```

可以通过这个配置项设置Nginx worker进程的nice优先级。

在Linux操作系统中，当许多进程都处于可执行状态时，将按照所有进程的优先级来决定本次内核选择哪一个进程执行，优先级越高，进程分配到的时间片也就越大。所以，优先级高的进程会占有更多的系统资源。

nice值是进程的静态优先级，它的取值范围是–20~+19，–20是最高优先 级，+19是最低优先级。因此，如果用户希望Nginx占有更多的系统资源，那么可以把nice值配置得更小一些，但不建议比内核进程的nice值（通常为–5）还要小。

2.3.4 事件类配置项

下面就是事件类配置项的相关介绍：

（1）是否打开accept锁

```
语法：accept_mutex[on|off];
默认：accept_mutex on;;
```

accept_mutex是Nginx的负载均衡锁，这个锁可以让多个worker进程轮流的与新的客户端建立TCP连接。当一个worker进程建立的连接数达到worker_connections配置的最大连接数的7/8时，会大大减小该进程建立TCP连接的机会，从而达到所有worker进程上处理客户端请求尽量接近。

accept锁是默认打开的，如果关闭它，建立TCP连接耗时会短，但负载会不均衡。

（2）lock文件的路径

```
语法：lock_file path/file;
默认：lock_file logs/nginx.lock;
```

accept锁可能需要这个lock文件，如果accept锁关闭，lock_file配置完全不生效。

（3）使用accept锁后到真正建立连接之间的延迟时间

```
语法：accept_mutex_delay Nms;
默认：accept_mutex_delay 500ms;
```

在使用accept锁后，同一时间只有一个worker进程能够取到accept锁。这个accept锁不是阻塞锁，如果取不到会立刻返回。如果有一个worker进程试图取accept锁而没有取到，它至少要等accept_mutex_delay定义的时间间隔后才能再次取锁。

（4）批量建立新连接

```
语法：multi_accept[on|off];
默认：multi_accept off;
```

当事件模型通知有新连接时，尽可能地对本次调度中客户端发起的所有TCP请求都建立连接。

（5）选择事件模型

```
语法：use[kqueue|rtsig|epoll|dev|poll|select|poll|eventport];
默认：Nginx会自动使用最适合的事件模型。
```

对于Linux操作系统来说，可供选择的事件驱动模型有poll、select、epoll三种。epoll当然是性能最高的一种。

（6）每个worker的最大连接数

```
语法：worker_connectioons number;
```

定义每个worker进程可以同时处理的最大连接数。

2.4 用HTTP核心模块配置一个静态web服务器

静态Web服务器的主要功能由**ngx_http_core_module**模块（HTTP框架的主要成员）实现，当然，一个完整的静态Web服务器还有许多功能是由其他的HTTP模块实现的。

除了2.3节提到的基本配置项外，一个典型的静态Web服务器还会包含多个server块和location块。

所有的HTTP配置项都必须直属于**http块、server块、location块、upstream块或if块**等。同时，在描述每个配置项的功能时，会说明它可以在上述的哪个块中存在，因为有些配置项可以任意地出现在某一个块中，而有些配置项只能出现在特定的块中。

Nginx为配置一个完整的静态Web服务器提供了非常多的功能，下面会把这些配置项分为以下8类进行详述：虚拟主机与请求的分发、文件路径的定义、内存及磁盘资源的分配、网络连接的设置、MIME类型的设置、对客户端请求的限制、文件操作的优化、对客户端请求的特殊处理。

2.4.1 虚拟主机与请求的分发

因为IP地址的数量限制，会存在许多个主机域名对应的一个IP地址的情况。这时在nginx.conf中就可以按照server_name（对应用户请求中的主机域名）并通过server块来定义虚拟主机，每个server块就是一个虚拟主机，它只处理与之相对应的主机域名请求。 这样，一台服务器上的Nginx就能以不同的方式处理访问不同主机域名的HTTP请求了。

（1）监听端口

```
语法：listen address:port[default|default_server|
[backlog=num|rcvbuf=size|sndbuf=size|accept_filter=filter|deferred|bind|ipv6only=
[on|off]|ssl]];
默认：listen 80;
配置块：server
```

listen参数决定Nginx服务如何监听端口。在listen后可以只加IP地址、端口或主机名，非常灵活，例如：

```
listen 127.0.0.1:8000;
```

在地址和端口后，还可以加上其他参数，例如：

```
listen 127.0.0.1 default_server accept_filter=dataready backlog=1024;
```

下面说明listen可用参数的意义：

- default、default_server：将所在的server块作为整个Web服务的默认server块，如果没有设置这个参数，那么将会以在nginx.conf中找到的第一个server块作为默认server块。
- backlog=num：表示TCP中backlog队列的大小。默认为-1，表示不予设置。在TCP 建立三次握手过程中，进程还没有开始处理监听句柄，这时backlog队列将会放置这些新连接。可如果backlog队列已满，还有新的客户端试图通过三次握手建立TCP连接，这时客户端将会建立连接失败。
- rcvbuf=size：设置监听句柄的SO_RCVBUF参数。
- sndbuf=size：设置监听句柄的SO_SNDBUF参数。
- accept_filter：设置accept过滤器，只对FreeBSD操作系统有用。
- deferred：在设置该参数后，若用户发起建立连接请求，并且完成了TCP的三次握手，内核也不会为了这次的连接调度worker进程来处理，只有用户真的发送请求数据时（内核已经在网卡中收到请求数据包），内核才会唤醒worker进程处理这个连接。
- bind：绑定当前端口/地址对，如127.0.0.1:8000。只有同时对一个端口监听多个地址时才会生效。
- ·ssl：在当前监听的端口上建立的连接必须基于SSL协议。

（2）主机名称

```
语法：server_name name[...];
默认：server_name"";
配置块：server
```

在开始处理一个HTTP请求时，Nginx会取出header头中的Host，与每个server中的server_name进行匹配，以此决定到底由哪一个server块来处理这个请求。

（3）server_names_hash_bucket_size

```
语法：server_names_hash_bucket_size size;
默认：server_names_hash_bucket_size 32|64|128;
配置块：http、server、location
```

为了提高快速寻找到相应server name的能力，Nginx使用散列表来存储server name。server_names_hash_bucket_size设置了每个散列桶占用的内存大小。

（4）重定向主机名称的处理

```
语法：server_name_in_redirect on|off;
默认：server_name_in_redirect on;
配置块：http、server、location
```

该配置需要配合server_name使用。在使用on打开时，表示在重定向请求时会使用 server_name里配置的第一个主机名代替原先请求中的Host头部，而使用off关闭时，表示在重定向请求时使用请求本身的Host头部。

（5）location

```
语法：location[=|~|~*|^~|@]/uri/{...};
配置块：server
```

location会尝试根据用户请求中的URI来匹配上面的/uri表达式，如果可以匹配，就选择location{}块中的配置来处理用户请求。

- =表示把URI作为字符串，以便与参数中的uri做完全匹配。
- ~表示匹配URI时是字母大小写敏感的。
- ~*表示匹配URI时忽略字母大小写问题。
- ^~表示匹配URI时只需要其前半部分与uri参数匹配即可。
- @表示仅用于Nginx服务内部请求之间的重定向，带有@的location不直接处理用户请求。

2.4.2 文件路径的定义

文件路径的定义配置项：

（1）以root方式设置资源路径

```
语法：root path;
默认：root html;
配置块：http、server、location、if
```

例如：

```
location /download/ {
root /opt/web/html/;
}
```

在上面的配置中，如果有一个请求的URI是/download/index/test.html，那么Web服务器将会返回服务器上/opt/web/html/download/index/test.html文件的内容。

（2）以alias方式设置资源路径

```
语法：alias path;
配置块：location
```

alias也是用来设置文件资源路径的，它与root的不同点主要在于如何解读紧跟location后面的uri参数，

例如，如果有一个请求的URI是/conf/nginx.conf，而用户实际想访问的文件在/usr/local/nginx/conf/nginx.conf，那么想要使用alias来进行设置的话，可以采用如下方 式：

```
location /conf {
alias /usr/local/nginx/conf/;
}
```

如果用root设置，那么语句如下所示：

```
location /conf {
root /usr/local/nginx/;
}
```

使用alias时，在URI向实际文件路径的映射过程中，已经把location后配置的/conf这部分字符串丢弃掉，因此，/conf/nginx.conf请求将根据alias path映射为path/nginx.conf。

（3）访问首页

```
语法：index file...;
默认：index index.html;
配置块：http、server、location
```

有时，访问站点时的URI是/，这时一般是返回网站的首页，而这与root和alias都不同。这里用ngx_http_index_module模块提供的index配置实现。

index后可以跟多个文件参数，Nginx将会按照顺序来访问这些文件，例如：

```
location / {
root path;
index /index.html /html/index.php /index.php;
}
```

接收到请求后，Nginx首先会尝试访问path/index.php文件，如果可以访问，就直接返回文件内容结束请求，否则再试图返回path/html/index.php文件的内容，依此类推。

（4）根据HTTP返回码重定向页面

```
语法：error_page code[code...][=|=answer-code]uri|@named_location
配置块：http、server、location、if
```

当对于某个请求返回错误码时，如果匹配上了error_page中设置的code，则重定向到新的URI中。例如：

```
error_page 403 http://example.com/forbidden.html
```

如果返回的错误码是403，就可以重定向到http://example.com/forbidden.html。

如果不想修改URI，只是想让这样的请求重定向到另一个location中进行处理，那么可以这样设置：

```
location / (
error_page 404 @fallback;
)
location @fallback (
proxy_pass http://backend
;
)
```

这样，返回404的请求会被反向代理到http://backend中处理。

（5）是否允许递归使用error_page

```
语法：recursive_error_pages[on|off];
默认：recursive_error_pages off;
配置块：http、server、location
```

确定是否允许递归地定义error_page。

（6）try_files

```
语法：try_files path1[path2]uri;
配置块：server、location
```

尝试按照顺序访问每一个path，如果可以有效地读取，就直接向用户返回这个path对应的文件结束请求，否则继续向下访问。如果所有的path都找不到有效的文件，就重定向到最后的参数uri上。

```
try_files /system/maintenance.html $uri $uri/index.html $uri.html @other;
location @other {
proxy_pass http://backend
;
}
```

这段代码表示如果前面的路径，如/system/maintenance.html等，都找不到，就会反向代理到http://backend服务上。

2.4.3 内存及磁盘资源的分配

下面介绍处理请求时内存、磁盘资源分配的配置项：

（1）HTTP包体只存储到磁盘文件中

```
语法：client_body_in_file_only on|clean|off;
默认：client_body_in_file_only off;
配置块：http、server、location
```

当值为非off时，用户请求中的HTTP包体一律存储到磁盘文件中，即使只有0字节也会存储为文件。当请求结束时，如果配置为on，则这个文件不会被删除（该配置一般用于调试、定位问题），但如果配置为clean，则会删除该文件。

（2）HTTP包体写入到一个内存buffer中

```
语法：client_body_in_single_buffer on|off;
默认：client_body_in_single_buffer off;
配置块：http、server、location
```

用户请求中的HTTP包体一律存储到内存buffer中。如果包体大小超过了client_body_buffer_size设置的值，还是会写入磁盘文件。

（3）存储HTTP头部的内存buffer大小

```
语法：client_header_buffer_size size;
默认：client_header_buffer_size 1k;
配置块：http、server
```

上面配置项定义了正常情况下Nginx接收用户请求中HTTP header部分（包括HTTP行和HTTP头部）时分配的内存buffer大小。

有时，请求中的HTTP header部分可能会超过这个大小，这时large_client_header_buffers定义的buffer将会生效。

（4）存储超大HTTP头部的内存buffer大小

```
语法：large_client_header_buffers number size;
默认：large_client_header_buffers 48k;
配置块：http、server
```

large_client_header_buffers定义了Nginx接收一个超大HTTP头部请求的buffer个数和每个buffer的大小。如果HTTP请求行的大小超过上面的单个 buffer，则返回"Request URI too large"(414)。

（5）存储HTTP包体的内存buffer大小

```
语法：client_body_buffer_size size;
默认：client_body_buffer_size 8k/16k;
配置块：http、server、location
```

定义了Nginx接收HTTP包体的内存缓冲区大小。HTTP包体会先接收到指定的这块缓存中，之后才决定是否写入磁盘。

（6）HTTP包体的临时存放目录

```
语法：client_body_temp_path dir-path[level1[level2[level3]]]
默认：client_body_temp_path client_body_temp;
配置块：http、server、location
```

在接收HTTP包体时，如果包体的大小大于client_body_buffer_size，则会以一个递增的整数命名并存放到client_body_temp_path指定的目录中。

（7）connection_pool_size

```
语法：connection_pool_size size;
默认：connection_pool_size 256;
配置块：http、server
```

Nginx对于每个建立成功的TCP连接会预先分配一个内存池，上面的size配置项将指定这个内存池的初始大小。

（8）request_pool_size

```
语法：request_pool_size size;
默认：request_pool_size 4k;
配置块：http、server
```

Nginx开始处理HTTP请求时，将会为每个请求都分配一个内存池，size配置项将指定这个内存池的初始大小。

TCP连接关闭时会销毁connection_pool_size指定的连接内存池，HTTP请求结束时会销毁request_pool_size指定的HTTP请求内存池，但它们的创建、销毁时间并不一致，因为一个TCP连接可能被复用于多个HTTP请求。

2.4.4 网络连接的设置

网络连接的设置配置项：

（1）读取HTTP头部的超时时间

```
语法：client_header_timeout time（默认单位：秒）；
默认：client_header_timeout 60;
配置块：http、server、location
```

客户端与服务器建立连接后将开始接收HTTP头部，在这个过程中，如果在一个时间间隔（超时时间）内没有读取到客户端发来的字节，则认为超时，并向客户端返回 408("Request timed out")响应。

（2）读取HTTP包体的超时时间

```
语法：client_body_timeout time（默认单位：秒）；
默认：client_body_timeout 60;
配置块：http、server、location
```

与client_header_timeout类似。

（3）发送响应的超时时间

```
语法：send_timeout time;
默认：send_timeout 60;
配置块：http、server、location
```

这个超时时间是发送响应的超时时间，即Nginx服务器向客户端发送了数据包，但客户端一直没有去接收这个数据包。如果某个连接超过send_timeout定义的超时时间，那么Nginx将会关闭这个连接。

（4）reset_timeout_connection

```
语法：reset_timeout_connection on|off;
默认：reset_timeout_connection off;
配置块：http、server、location
```

连接超时后将通过向客户端发送RST包来直接重置连接。不是使用正常情形下的四次握手关闭TCP连接，而是直接向用户发送RST重置包。

使用RST重置包关闭连接会带来一些问题，默认情况下不会开启。

（5）lingering_close

```
语法：lingering_close off|on|always;
默认：lingering_close on;
配置块：http、server、location
```

该配置控制Nginx关闭用户连接的方式。always表示关闭用户连接前必须无条件地处理连接上所有用户发送的数据。off表示关闭连接时完全不管连接上是否已经有准备就绪的来自用户的数据。on是中间值，一般情况下在关闭连接前都会处理连接上的用户发送的数据，除了有些情况下在业务上认定这之后的数据是不必要的。

（6）lingering_time

```
语法：lingering_time time;
默认：lingering_time 30s;
配置块：http、server、location
```

lingering_close启用后，这个配置项对于上传大文件很有用。

当用户请求的Content-Length大于max_client_body_size配置时，Nginx服务会立刻向用户发送413（Request entity too large）响应。但是，很多客户端不管413返回值，仍然上传HTTP body，这时，经过了lingering_time设置的时间后，Nginx将不管用户是否仍在上传，都会把连接关闭掉。

（7）lingering_timeout

```
语法：lingering_timeout time;
默认：lingering_timeout 5s;
配置块：http、server、location
```

（8）对某些浏览器禁用keepalive功能

```
语法：keepalive_disable[msie6|safari|none]...
默认：keepalive_disablemsie6 safari
配置块：http、server、location
```

HTTP请求中的keepalive功能是为了让多个请求复用一个HTTP长连接，这个功能对服务器的性能提高是很有帮助的。但有些浏览器，如IE 6和Safari，它们对于使用keepalive功能的POST请求处理有功能性问题。

（9）keepalive超时时间

```
语法：keepalive_timeout time（默认单位：秒）;
默认：keepalive_timeout 75;
配置块：http、server、location
```

一个keepalive连接在闲置超过一定时间后，服务器和浏览器都会去关闭这个连接。

（10）一个keepalive长连接上允许承载的请求最大数

```
语法：keepalive_requests n;
默认：keepalive_requests 100;
配置块：http、server、location
```

一个keepalive连接上默认最多只能发送100个请求。

（11）tcp_nodelay

```
语法：tcp_nodelay on|off;
默认：tcp_nodelay on;
配置块：http、server、location
```

确定对keepalive连接是否使用TCP_NODELAY选项。

2.4.5 对客户端请求的限制

对客户端请求的限制的配置项：

（1）按HTTP方法名限制用户请求

```
语法：limit_except method...{...}
配置块：location
```

Nginx通过limit_except后面指定的方法名来限制用户请求。方法名可取值包括： GET、HEAD、POST、PUT、DELETE、MKCOL、COPY、MOVE、OPTIONS、 PROPFIND、PROPPATCH、LOCK、UNLOCK或者PATCH。例如：

```
limit_except GET {
allow 192.168.1.0/32;
deny all;
}
```

上面这段代码表示的是禁止GET方法和HEAD方法，但其他HTTP方法是允许的。

（2）HTTP请求包体的最大值

```
语法：client_max_body_size size;
默认：client_max_body_size 1m;
配置块：http、server、location
```

浏览器在发送含有较大HTTP包体的请求时，其头部会有一个Content-Length字段，client_max_body_size是用来限制Content-Length所示值的大小的。这个限制包体的配置非常有用处，因为不用等Nginx接收完所有的HTTP包体就可以告诉用户请求过大，就直接发送 413("Request Entity Too Large")响应给客户端。

（3）对请求的限速

```
语法：limit_rate speed;
默认：limit_rate 0;
配置块：http、server、location、if
```

此配置是对客户端请求限制每秒传输的字节数。默认参数为0，表示不限速。

（4）limit_rate_after

```
语法：limit_rate_after time;
默认：limit_rate_after 1m;
配置块：http、server、location、if
```

此配置表示Nginx向客户端发送的响应长度超过limit_rate_after后才开始限速。

```
limit_rate_after 1m;
limit_rate 100k;//传输长度超过1m后开始限速100k。
```

2.4.6 文件操作的优化

文件操作的优化配置项：

（1）sendfile系统调用

```
语法：sendfile on|off;
默认：sendfile off;
配置块：http、server、location
```

可以启用Linux上的sendfile系统调用来发送文件，它减少了内核态与用户态之间的两次内存复制，这样就会从磁盘中读取文件后直接在内核态发送到网卡设备，提高了发送文件的效率。

（2）打开文件缓存

```
语法：open_file_cache max=N[inactive=time]|off;
默认：open_file_cache off;
配置块：http、server、location
```

文件缓存会在内存中存储以下3种信息：

- 文件句柄、文件大小和上次修改时间。
- 已经打开过的目录结构。
- 没有找到的或者没有权限操作的文件信息。

这样，通过读取缓存就减少了对磁盘的操作。该配置项后面跟3种参数：

- max：表示在内存中存储元素的最大个数。当达到最大限制数量后，将采用 LRU（Least Recently Used）算法从缓存中淘汰最近最少使用的元素。
- ·inactive：表示在inactive指定的时间段内没有被访问过的元素将会被淘汰。默认时间为60秒。
- off：关闭缓存功能。

例如：

```
open_file_cache max=1000 inactive=20s;
```

（3）是否缓存打开文件错误的信息

```
语法：open_file_cache_errors on|off;
默认：open_file_cache_errors off;
配置块：http、server、location
```

此配置项表示是否在文件缓存中缓存打开文件时出现的找不到路径、没有权限等错误信息。

（4）不被淘汰的最小访问次数

```
语法：open_file_cache_min_uses number;
默认：open_file_cache_min_uses 1;
配置块：http、server、location
```

它与open_file_cache中的inactive参数配合使用。如果在inactive指定的时间段内，访问次数超过了open_file_cache_min_uses指定的最小次数，那么将不会被淘汰出缓存。

2.4.7 对客户端请求的特殊处理

客户端请求的特殊处理的配置项：

（1）忽略不合法的HTTP头部

```
语法：ignore_invalid_headers on|off;
默认：ignore_invalid_headers on;
配置块：http、server
```

设置为off，当出现不合法的HTTP头部时，Nginx会拒绝服务，并直接向用户发送400（Bad Request）错误。如果将其设置为on，则会忽略此HTTP头部。

（2）HTTP头部是否允许下划线

```
语法：underscores_in_headers on|off;
默认：underscores_in_headers off;
配置块：http、server
```

默认为off，表示HTTP头部的名称中不允许带（下划线）。

（3）对If-Modified-Since头部的处理策略

```
语法：if_modified_since[off|exact|before];
默认：if_modified_since exact;
配置块：http、server、location
```

出于性能考虑，Web浏览器一般会在客户端本地缓存一些文件，并存储当时获取的时间。这样，下次向Web服务器获取缓存过的资源时，就可以用If-Modified-Since头部把上次获取的时间捎带上，而if_modified_since将根据后面的参数决定如何处理If-Modified-Since头部。

相关参数说明如下：

- off：表示忽略用户请求中的If-Modified-Since头部。这时，如果获取一个文件，那么会正常地返回文件内容。HTTP响应码通常是200。
- exact：将If-Modified-Since头部包含的时间与将要返回的文件上次修改的时间做精确比较，如果没有匹配上，则返回200和文件的实际内容，如果匹配上，则表示浏览器缓存的文件内容已经是最新的了，没有必要再返回文件从而浪费时间与带宽了，这时会返回 304 Not Modified，浏览器收到后会直接读取自己的本地缓存。
- before：是比exact更宽松的比较。只要文件的上次修改时间等于或者早于用户请求中的If-Modified-Since头部的时间，就会向客户端返回304 Not Modified。

（4）文件未找到时是否记录到error日志

```
语法：log_not_found on|off;
默认：log_not_found on;
配置块：http、server、location
```

此配置项表示当处理用户请求且需要访问文件时，如果没有找到文件，是否将错误日志记录到error.log文件中。这仅用于定位问题。

（5）merge_slashes

```
语法：merge_slashes on|off;
默认：merge_slashes on;
配置块：http、server、location
```

此配置项表示是否合并相邻的“/”，例如，//test///a.txt，在配置为on时，会将其匹配为location/test/a.txt；如果配置为off，则不会匹配，URI将仍然是//test///a.txt。

（6）DNS解析地址

```
语法：resolver address...;
配置块：http、server、location
```

第3章 负载均衡模块

3.1 负载均衡原理

当Web服务器直接面向用户，往往要承载大量并发请求，单台服务器难以负荷，那么就需要使用多台Web服务器组成集群，前端使用Nginx负载均衡，将请求分散的打到后端服务器集群中，实现负载的分发。那么会大大提升系统的吞吐率、请求性能、高容灾。

集群的分类：

- 负载均衡集群（Load Balancing clusters），简称LBC或者LB。
- 高可用性集群（High-availability（HA）clusters），简称HAC。
- 高性能计算集群（High-performance（HP）clusters），简称HPC。
- 网格计算（Grid computing）集群。

负载均衡集群的作用：

- 分担用户访问请求及数据流量（负载均衡）。
- 保持业务连续性，即7×24小时服务（高可用性）。
- 应用于Web业务及数据库从库等服务器的业务。
- 负载均衡集群典型的开源软件包括LVS、Nginx、Haproxy等。

负载均衡可以分为硬件负载均衡和软件负载均衡，前者一般是专用的软件和硬件相结合的设施，设施商会提供完整成熟的处理方案，通常也会更加昂贵。软件的复杂均衡以Nginx占据绝大多数。

客户端向反向代理发送请求，接着反向代理根据某种负载机制转发请求至目标服务器(这些服务器都运行着相同的应用)，并把获得的内容返回给客户端，代理请求可能根据配置被发往不同的目标服务器。

负载均衡能实现的应用场景一：四层负载均衡：

所谓四层负载均衡指的是OSI七层模型中的传输层，那么传输层Nginx已经能支持TCP/IP的控制，所以只需要对客户端的请求进行TCP/IP协议的包转发就可以实现负载均衡，那么它的好处是性能非常快、只需要底层进行应用处理，而不需要进行一些复杂的逻辑。

负载均衡能实现的应用场景二：七层负载均衡：

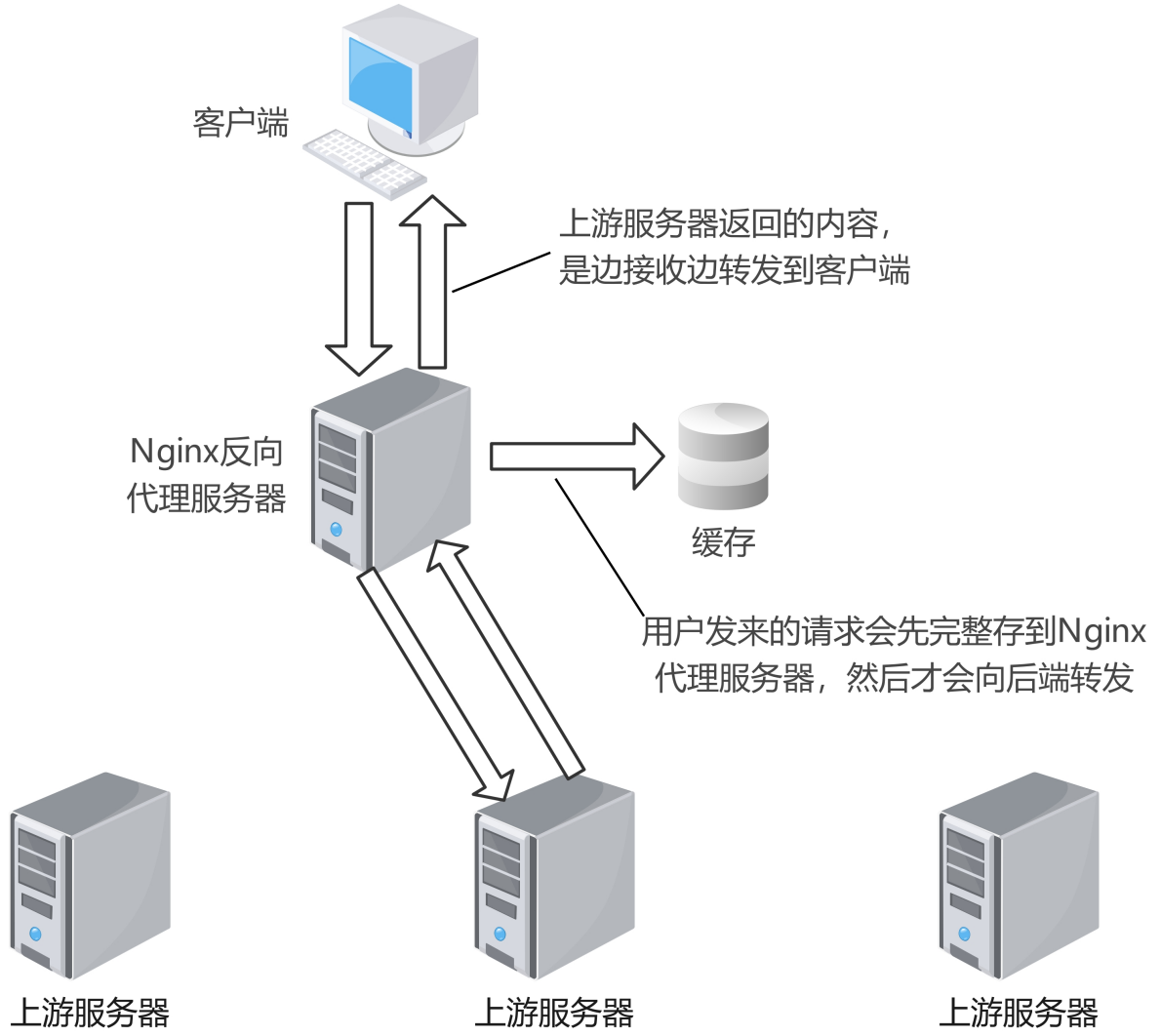
七层负载均衡它是在应用层，那么它可以完成很多应用方面的协议请求，比如常说的http应用的负载均衡，它可以实现http信息的改写、头信息的改写、安全应用规则控制、URL匹配规则控制、以及转发、rewrite等等的规则，所以在应用层的服务里面，可以做的内容就更多，那么Nginx则是一个典型的七层负载均衡SLB。

四层负载均衡与七层负载均衡区别

四层负载均衡数据包在底层就进行了分发，而七层负载均衡数据包则是在最顶层进行分发、由此可以看出，七层负载均衡效率没有四负载均衡高。

但七层负载均衡更贴近于服务，如：http协议就是七层协议，可以用Nginx可以作会话保持，URL路径规则匹配、head头改写等等，这些是四层负载均衡无法实现的。

Nginx的这种工作方式有什么优缺点呢？很明显，缺点是延长了一个请求的处理时间，并增加了用于缓存请求内容的内存和磁盘空间。而优点则是降低了上游服务器的负载，尽量把压力放在Nginx服务器上。



通常，客户端与代理服务器之间的网络环境会比较复杂，多半是“走”公网，网速平均下来可能较慢，因此，一个请求可能要持续很久才能完成。而代理服务器与上游服务器之间一般是“走”内网，或者有专线连接，传输速度较快。

Squid等反向代理服务器在与客户端建立连接且还没有开始接收HTTP包体时，就已经向上游服务器建立了连接。

Nginx则不然，它在接收到完整的客户端请求（如1GB的文件）后，才会与上游服务器建立连接转发请求，由于是内网，所以这个转发过程会执行得很快。这样，一个客户端请求占用上游服务器的连接时间就会非常短，也就是说，Nginx的这种反向代理方案主要是为了降低上游服务器的并发压力。

3.2 Nginx负载均衡模块ngx_http_upstream_module

ngx_http_upstream_module模块用于定义可以被 proxy_pass、fastcgi_pass、uwsgi_pass、scgi_pass 以及memcached_pass 等指令引用的服务器群。

对于常用的HTTP负载均衡，主要先定义一个upstream作为backend group，而后通过proxy_pass/fastcgi_pass等方式进行转发操作，其中fastcgi_pass几乎算是Nginx+PHP站点的标配。

Nginx负载均衡也是一种代理，与Nginx代理不同地方在于，Nginx的一个location仅能代理一台服务器，而Nginx负载均衡则是将客户端请求代理转发至一组upstream虚拟服务池（对应于后端多个服务器，逻辑上的一组），即负载均衡能对物理主机进行逻辑上的捆绑

ngx_http_upstream_module提供的常用配置指令：

（1）upstream块

```
语法：upstream name{...}
配置块：http
```

upstream块定义了一个上游服务器的**集群**，便于反向代理中的proxy_pass使用。例如：

```
upstream backend {
server backend1.example.com;
server backend2.example.com;
server backend3.example.com;
}
server {
location / {
proxy_pass http://backend;
}
}
```

（2）server

```
语法：server name[parameters];
配置块：upstream
```

server配置项指定了一台上游服务器的名字，这个名字可以是域名、IP地址端口、 UNIX句柄等，在其后还可以跟下列参数：

- weight=number：设置向这台上游服务器转发的权重，默认为1。
- max_fails=number：该选项与fail_timeout配合使用，指在fail_timeout时间段内，如果向当前的上游服务器转发失败次数超过number，则认为在当前的fail_timeout时间段内这台上游服务器不可用。max_fails默认为1，如果设置为0，则表示不检查失败次数。
- fail_timeout=time：fail_timeout表示该时间段内转发失败多少次后就认为上游服务器暂时不可用，用于优化反向代理功能。fail_timeout默认为10秒。
- down：表示所在的上游服务器永久下线，只在使用ip_hash配置项时才有用。
- backup：在使用ip_hash配置项时它是无效的。它表示所在的上游服务器只是备份服务器，只有在所有的非备份上游服务器都失效后，才会向所在的上游服务器转发请求。

例如：

```
upstream backend {
server backend1.example.com weight=5;
server 127.0.0.1:8080 max_fails=3 fail_timeout=30s;
server unix:/tmp/backend3;
}
```

（3）ip_hash

```
语法：ip_hash;
配置块：upstream
```

某些时候，希望来自某一个用户的请求始终落到固定的一台上游服务器中。例如，上游服务器会缓存一些信息，如果同一个用户的请求任意地转发到集群中的任一台上游服务器中，那么每一台上游服务器都有可能会缓存同一份信息，这既会造成资源的浪费，也会难以有效地管理缓存信息。

ip_hash就是用以解决上述问题的，它首先根据客户端的IP地址计算出一个key，将key按照upstream集群里的上游服务器数量进行取模，然后以取模后的结果把请求转发到相应的上游服务器中。这样就确保了同一个客户端的请求只会转发到指定的上游服务器中。

ip_hash与weight（权重）配置不可同时使用。如果upstream集群中有一台上游服务器暂时不可用，不能直接删除该配置，而是要down参数标识，确保转发策略的一贯性。例如：

```
upstream backend {
ip_hash;
server backend1.example.com;
server backend2.example.com;
server backend3.example.com down;
server backend4.example.com;
}
```

（4）记录日志时支持的变量

如果需要将负载均衡时的一些信息记录到access_log日志中，那么在定义日志格式时可以使用负载均衡功能提供的变量：

```
$upstream_addr              //处理请求的上游服务器地址
$upstream_cache_status      //表示是否命中缓存，取值：MISS、EXPIRED、UODATING、STALE、
HIT
$upstream_status            //上游服务器返回的响应中的HTTP响应码
$upstream_response_time     //上游服务器的响应时间
$upstream_http_$HEADER      //HTTP的头部
```

例如：

```
log_format timing '$remote_addr - $remote_user [$time_local] $request '
'upstream_response_time $upstream_response_time '
'msec $msec request_time $request_time';
log_format up_head '$remote_addr - $remote_user [$time_local] $request '
'upstream_http_content_type $upstream_http_content_type';
```

3.3 反向代理的基本配置

反向代理的基本配置项：

（1）proxy_pass

```
语法：proxy_pass URL;
配置块：location、if
```

此配置项将当前请求反向代理到URL参数指定的服务器上，URL可以是主机名或IP地址加端口的形式，例如：

```
proxy_pass http://localhost:8000/uri/
proxy_pass https://192.168.0.1
```

默认情况下反向代理是不会转发请求中的Host头部的。如果需要转发，那么必须加上配置：

```
proxy_set_header Host $host;
```

（2）proxy_method

```
语法：proxy_method method;
配置块：http、server、location
```

此配置项表示转发时的协议方法名。例如设置为：

```
proxy_method POST;
```

那么客户端发来的GET请求在转发时方法名也会改为POST。

（3）proxy_hide_header

```
语法：proxy_hide_header the_header;
配置块：http、server、location
```

Nginx会将上游服务器的响应转发给客户端，但默认不会转发以下HTTP头部字段：Date、Server、X-Pad和X-Accel-*。

使用proxy_hide_header后可以任意地指定哪些HTTP头部字段不能被转发。

（4）proxy_pass_header

```
语法：proxy_pass_header the_header;
配置块：http、server、location
```

与proxy_hide_header功能相反，proxy_pass_header会将原来禁止转发的header设置为允许转发。

（5）proxy_pass_request_body

```
语法：proxy_pass_request_body on|off;
默认：proxy_pass_request_body on;
配置块：http、server、location
```

作用为确定是否向上游服务器发送HTTP包体部分。

（6）proxy_pass_request_headers

```
语法：proxy_pass_request_headers on|off;
默认：proxy_pass_request_headers on;
配置块：http、server、location
```

作用为确定是否转发HTTP头部。

（7）proxy_redirect

```
语法：proxy_redirect[default|off|redirect replacement];
默认：proxy_redirect default;
配置块：http、server、location
```

当上游服务器返回的响应是重定向或刷新请求（如HTTP响应码是301或者302）时，proxy_redirect可以重设HTTP头部的location或refresh字段。

例如，如果上游服务器发出的响应是302重定向请求，location字段的URI是http://localhost:8000/two/some/uri/，那么在下面的配置情况下，实际转发给客户端的location是http://frontend/one/some/uri/。

```
proxy_redirect http://localhost:8000/two/
http://frontend/one/;
```

使用off参数时，将使location或者refresh字段维持不变。

（8）proxy_next_upstream

```
语法：
proxy_next_upstream[error|timeout|invalid_header|http_500|http_502|http_503|http_
504|http_404|off];
默认： proxy_next_upstream error timeout;
配置块： http、server、location
```

此配置项表示当向一台上游服务器转发请求出现错误时，继续换一台上游服务器处理这个请求。

上游服务器一旦开始发送应答，Nginx反向代理服务器会立刻把应答包转发给客户端。因此，一旦Nginx开始向客户端发送响应包，之后的过程中若出现错误也是不允许换下一台上游服务器继续处理的。

proxy_next_upstream的参数用来说明在哪些情况下会继续选择下一台上游服务器转发请求：

- error：当向上游服务器发起连接、发送请求、读取响应时出错。
- timeout：发送请求或读取响应时发生超时。
- invalid_header：上游服务器发送的响应是不合法的。
- http_500：上游服务器返回的HTTP响应码是500。
- http_502：上游服务器返回的HTTP响应码是502。

- http_503：上游服务器返回的HTTP响应码是503。
- http_504：上游服务器返回的HTTP响应码是504。
- http_404：上游服务器返回的HTTP响应码是404。
- off：关闭proxy_next_upstream功能，出错就选择另一台上游服务器再次转发。

第四章 进程管理模块

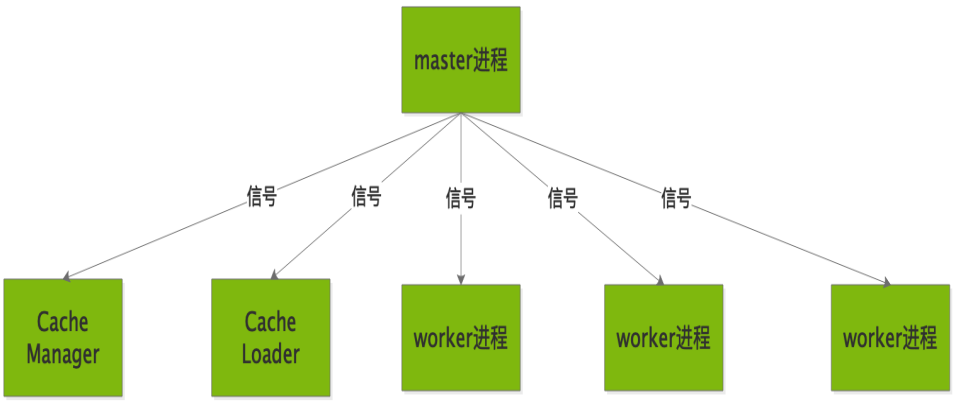
4.1 nginx 进程简介

为了保证 nginx 的高可用高可靠，nginx 被设计成多进程结构。多进程相对于多线程之所以能够保证高可用与高可靠是因为进程间地址空间是独立的，进程间的任务不会相互影响，相对多线程更加耗费 CPU 资源。而多线程共享一个进程的地址空间，其中一个线程任务失败会影响到其它线程任务。

4.1.1 nginx 进程类型

nginx 中有数种类型的进程。进程的类型保存在 **ngx_process** 全局变量中，介绍如下：

- NGX_PROCESS_MASTER —— 主进程，读取 NGINX 配置，创建 cycles，启动并控制子进程。不执行任何 I/O ，只对信号做出响应。它的 cycle 函数是 ngx_process_cycle() .
- NGX_PROCESS_WORKER —— 辅助进程，处理客户端连接，由主进程启动，并对其信号和通道命令做出响应。它的 cycle 函数是 ngx_worker_process_cycle() . 可以由 worker_processes 指令配置多个 worker 进程。
- NGX_PROCESS_SINGLE —— 只存在于 master_process off 模式下的单个进程，并且是该模式下运行的的唯一进程。它会像主进程一样创建循环cycles并像辅助进程一样处理客户端连接。它的循环函数是 ngx_single_process_cycle() 。
- NGX_PROCESS_HELPER —— 帮助进程，目前有两种类型：缓存管理器和缓存加载器。这两者的 cycle 函数都是 ngx_cache_manager_process_cycle() 。



4.1.2 nginx 进程中的信号量

nginx 进程处理以下信号：

- NGX_SHUTDOWN_SIGNAL （在大多数系统上为 SIGQUIT） —— 平滑地关闭。在接收到这个信号之后，master 进程向所有子进程发送一个关闭信号。当没有子进程留下时，主进程销毁循环池并退出。当 worker 进程接收到这个信号，关闭所有监听套接字并等待，直到没有安排不可取消的事件，然后销毁循环池并退出。当缓存管理器或缓存加载器程序进程接收到这个信号之后，将立即退出。当一个进程收到这个信号时，变量 ngx_exit 将置为 1 ，并且在被处理之后立即重置。当 worker 进程处于关闭状态时，ngx_exiting 变量设置为 1 .
- NGX_TERMINATE_SIGNAL（在大多数系统上为 SIGTERM） —— 终止。接收到此信号之后，master 进程向所有子进程发送终止信号。如果一个子进程没有在 1 s 内退出，master 进程将发送 SIGKILL 信号来终止它。当没有子进程时，master 进程销毁循环池并退出。当 worker 进程、缓存管理器进程或缓存加载器程序进程接收到此信号时，它将销毁循环池并退出。所有进程接收到这个信号时，都会将 ngx_terminate 设置为 1 .
- NGX_NOACCEPT_SIGNAL（在大多数系统中为 SIGWINCH） —— 关闭所有的 worker 进程和 helper 进程。接收到此信号之后，master 进程关闭其子进程。如果先前启动的新的 nginx binary 退出，则再次启动旧 master 进程的子进程。当 worker 进程接收到该信号时，它将在 debug_points 指令设置的调试模式下关闭。
- NGX_RECONFIGURE_SIGNAL（在大多数系统中为 SIGHUP） —— 重新配置。接收到这个信号之后，master 进程重新读取配置文件并根据它创建一个新的循环。如果成功创建了新的循环，则删除旧的循环并启动新的子进程。与此同时，旧的子进程接收 NGX_SHUTDOWN_SIGNAL 。在单进程模式下，nginx 创建了一个新的循环，但是保留了旧的循环直到不再有存活的连接绑定到它的客户端为止。worker 进程和 helper 进程忽略这个信号。
- NGX_REOPEN_SIGNAL（在大多数系统中为 SIGUSR1） —— 重新打开文件。master 进程将这个信号发送给 workers，worker 进程重新打开所有与循环相关的 open_files 。
- NGX_CHANGEBIN_SIGNAL（在大多数系统中为 SIGUSR2） —— 更改 nginx 二进制文件。master 进程启动一个新的 nginx 二进制文件，并传入所有监听套接字的列表。在“NGINX”环境变量中传递的文本格式列表由用分号分隔的描述符号组成。新的 nginx 二进制文件读取“NGINX”变量并将套接字添加到其初始周期中。其他进程忽略该信号。

4.2 nginx 进程管理

通过以上的信息我们可以得知，可以通过 master 进程、worker 进程以及命令行来管理 nginx 进程。一般情况下，我们使用信号量管理 master 进程，进而来管理和维护 worker 进程，而不直接使用发送信号量来管理 worker 进程。

- master 进程主要完成以下工作：

```
1. 读取并验证配置信息
2. 创建、绑定及关闭套接字
3. 启动、终止及维护 worker 进程的个数
4. 无需中止服务而重新配置工作特性
5. 控制非中断式程序升级，启用新的二进制程序并在需要时回退至老版本
6. 重新打开日志文件，实现日志滚动
7. 编译嵌入式 perl 脚本
```

- worker 进程主要完成的任务：

```
1. 接收、传入并处理来自客户端的连接
2. 提供反向代理及过滤功能
3. nginx 任何能完成的其他任务
```

- cache loader 进程主要完成的任务：

```
1. 检查缓存存储中的缓存对象
2. 使用缓存元数据建立内存数据库
```

- cache manager 进程的主要任务：

```
缓存的失效及过期检验
```

使用命令行管理 nginx 进程

启动 nginx：

```
nginx
```

强制退出：

```
nginx -s stop
```

平滑退出：

```
nginx -s quit
```

重新打开日志文件：

```
nginx -s reopen
```

重新加载配置：

```
nginx -s reload
```

指定安装路径：

```
nginx -p prefix
```

指明配置文件路径：

```
nginx -c filename
```

结束所有 nginx 进程：

```
killall nginx
```

虽然 nginx 所有 worker 进程都能够接收并正确处理 POSIX 信号，但主进程不使用标准 kill() 系统调用将信号传递给 worker 进程和 helper 进程。相反，nginx 使用进程间套接字对，它允许在所有 nginx 进程之间发送消息。但是，目前消息仅从主服务器发送给其子服务器。信息传递的是标准信号。

4.3 nginx 多进程具体实现

nginx 多进程实现包括以下几个步骤：

1. 启动 nginx 的多进程模式。主进程信号监听和启动工作进程。
2. 创建和启动工作进程。工作进程数一般为 CPU 个数的 1 ~ 2 倍。
3. fork 工作进程的子进程。启动成功，回调 ngx_worker_process_cycle
4. 子进程回调函数。每个进程的逻辑处理就从这个方法开始。
5. 工作进程初始化。
6. 事件驱动核心函数，进入事件循环驱动。

4.3.1 ngx_master_process_cycle 进入多进程模式

ngx_master_process_cycle 方法主要做了两个工作：

- 主进程进行信号的监听和处理
- 开启子进程

```
/**
 * Nginx的多进程运行模式
 */
void ngx_master_process_cycle(ngx_cycle_t *cycle) {
    char *title;
    u_char *p;
    size_t size;
    ngx_int_t i;
    ngx_uint_t n, sigio;
    sigset_t set;
    struct itimerval itv;
    ngx_uint_t live;
    ngx_msec_t delay;
    ngx_listening_t *ls;
    ngx_core_conf_t *ccf;

    /* 设置能接收到的信号 */
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigaddset(&set, SIGALRM);
    sigaddset(&set, SIGIO);
    sigaddset(&set, SIGINT);
    sigaddset(&set, ngx_signal_value(NGX_RECONFIGURE_SIGNAL));
    sigaddset(&set, ngx_signal_value(NGX_REOPEN_SIGNAL));
    sigaddset(&set, ngx_signal_value(NGX_NOACCEPT_SIGNAL));
    sigaddset(&set, ngx_signal_value(NGX_TERMINATE_SIGNAL));
    sigaddset(&set, ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
    sigaddset(&set, ngx_signal_value(NGX_CHANGEBIN_SIGNAL));

    if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                "sigprocmask() failed");
    }
```

```
    sigemptyset(&set);

    size = sizeof(master_process);

    for (i = 0; i < ngx_argc; i++) {
        size += ngx_strlen(ngx_argv[i]) + 1;
    }

    /* 保存进程标题 */
    title = ngx_pnalloc(cycle->pool, size);
    if (title == NULL) {
        /* fatal */
        exit(2);
    }

    p = ngx_cpymem(title, master_process, sizeof(master_process) - 1);
    for (i = 0; i < ngx_argc; i++) {
        *p++ = ' ';
        p = ngx_cpystrn(p, (u_char *) ngx_argv[i], size);
    }

    ngx_setproctitle(title);

    /* 获取核心配置 ngx_core_conf_t */
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

    /* 启动工作进程 - 多进程启动的核心函数 */
    ngx_start_worker_processes(cycle, ccf->worker_processes,
            NGX_PROCESS_RESPAWN);
    ngx_start_cache_manager_processes(cycle, 0);

    ngx_new_binary = 0;
    delay = 0;
    sigio = 0;
    live = 1;

    /* 主线程循环 */
    for (;;) {

        /* delay用来设置等待worker推出的时间，master接受了退出信号后，
         * 首先发送退出信号给worker，而worker退出需要一些时间*/
        if (delay) {
            if (ngx_sigalrm) {
                sigio = 0;
                delay *= 2;
                ngx_sigalrm = 0;
            }

            ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                    "termination cycle: %M", delay);

            itv.it_interval.tv_sec = 0;
            itv.it_interval.tv_usec = 0;
            itv.it_value.tv_sec = delay / 1000;
            itv.it_value.tv_usec = (delay % 1000) * 1000;
```

```
        if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "setitimer() failed");
        }
    }

    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "sigsuspend");

    /* 等待信号的到来，阻塞函数 */
    sigsuspend(&set);

    ngx_time_update();

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
            "wake up, sigio %i", sigio);

    /* 收到了SIGCHLD信号，有worker退出(ngx_reap == 1) */
    if (ngx_reap) {
        ngx_reap = 0;
        ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "reap children");

        live = ngx_reap_children(cycle);
    }

    if (!live && (ngx_terminate || ngx_quit)) {
        ngx_master_process_exit(cycle);
    }

    /* 中止进程  */
    if (ngx_terminate) {
        if (delay == 0) {
            delay = 50;
        }

        if (sigio) {
            sigio--;
            continue;
        }

        sigio = ccf->worker_processes + 2 /* cache processes */;

        if (delay > 1000) {
            ngx_signal_worker_processes(cycle, SIGKILL);
        } else {
            ngx_signal_worker_processes(cycle,
                    ngx_signal_value(NGX_TERMINATE_SIGNAL));
        }

        continue;
    }

    /* 退出进程 */
    if (ngx_quit) {
        ngx_signal_worker_processes(cycle,
                ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
```

```
            ls = cycle->listening.elts;
            for (n = 0; n < cycle->listening.nelts; n++) {
                if (ngx_close_socket(ls[n].fd) == -1) {
                    ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
                                  ngx_close_socket_n " %V failed", &ls[n].addr_text);
                }
            }
            cycle->listening.nelts = 0;

            continue;
        }

        /* 收到SIGHUP信号 重新初始化配置 */
        if (ngx_reconfigure) {
            ngx_reconfigure = 0;

            if (ngx_new_binary) {
                ngx_start_worker_processes(cycle, ccf->worker_processes,
                                           NGX_PROCESS_RESPAWN);
                ngx_start_cache_manager_processes(cycle, 0);
                ngx_noaccepting = 0;

                continue;
            }

            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");

            cycle = ngx_init_cycle(cycle);
            if (cycle == NULL) {
                cycle = (ngx_cycle_t *) ngx_cycle;
                continue;
            }

            ngx_cycle = cycle;
            ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,
                                                   ngx_core_module);
            ngx_start_worker_processes(cycle, ccf->worker_processes,
                                       NGX_PROCESS_JUST_RESPAWN);
            ngx_start_cache_manager_processes(cycle, 1);

            /* allow new processes to start */
            ngx_msleep(100);

            live = 1;
            ngx_signal_worker_processes(cycle,
                                        ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
        }

        /* 当ngx_noaccepting==1时，会把ngx_restart设为1，重启worker  */
        if (ngx_restart) {
            ngx_restart = 0;
            ngx_start_worker_processes(cycle, ccf->worker_processes,
                                       NGX_PROCESS_RESPAWN);
            ngx_start_cache_manager_processes(cycle, 0);
            live = 1;
```

```
        }

        /* 收到SIGUSR1信号，重新打开log文件 */
        if (ngx_reopen) {
            ngx_reopen = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
            ngx_reopen_files(cycle, ccf->user);
            ngx_signal_worker_processes(cycle,
                    ngx_signal_value(NGX_REOPEN_SIGNAL));
        }

        /* SIGUSER2，热代码替换 */
        if (ngx_change_binary) {
            ngx_change_binary = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "changing binary");
            ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
        }

        /* 收到SIGWINCH信号不在接受请求，worker退出，master不退出 */
        if (ngx_noaccept) {
            ngx_noaccept = 0;
            ngx_noaccepting = 1;
            ngx_signal_worker_processes(cycle,
                    ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
        }
    }
}
```

4.3.2 ngx_start_worker_process 创建工作进程

- 通过循环创建 N 个子进程，每个子进程都有独立的内存空间。
- 子进程的个数由 nginx 的配置：ccf -> worker_processes 决定

```
/**
 * 创建工作进程
 */
static void ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n,
        ngx_int_t type) {
    ngx_int_t i;
    ngx_channel_t ch;

    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start worker processes");

    ngx_memzero(&ch, sizeof(ngx_channel_t));

    ch.command = NGX_CMD_OPEN_CHANNEL;

    /* 循环创建工作进程  默认ccf->worker_processes=8个进程，根据CPU个数决定    */
    for (i = 0; i < n; i++) {

        /* 打开工作进程  （ngx_worker_process_cycle 回调函数，主要用于处理每个工作线程）
*/
        ngx_spawn_process(cycle, ngx_worker_process_cycle,
```

```
                (void *) (intptr_t) i, "worker process", type);

        ch.pid = ngx_processes[ngx_process_slot].pid;
        ch.slot = ngx_process_slot;
        ch.fd = ngx_processes[ngx_process_slot].channel[0];

        ngx_pass_open_channel(cycle, &ch);
    }
}
```

4.3.3 ngx_spawn_process fork 工作进程

ngx_spawn_process 方法主要用于 fork 出各个工作进程。代码如下：

```
    /* fork 一个子进程 */
    pid = fork();

    switch (pid) {

    case -1:
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "fork() failed while spawning \"%s\"", name);
        ngx_close_channel(ngx_processes[s].channel, cycle->log);
        return NGX_INVALID_PID;

    case 0:
        /* 如果pid fork成功，则调用 ngx_worker_process_cycle方法 */
        ngx_pid = ngx_getpid();
        proc(cycle, data);
        break;

    default:
        break;
    }
```

4.3.4 ngx_worker_process_cycle 子进程的回调函数

- ngx_worker_process_cycle 是子进程的回调函数，所有子进程的工作从这个方法开始。
- nginx 的进程最终也是有事件驱动的，所以这个方法中，最终会调用 ngx_process_events_and_timers 事件驱动的核心函数。

```
/**
 * 子进程 回调函数
 * 每个进程的逻辑处理就从这个方法开始
 */
static void ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data) {
    ngx_int_t worker = (intptr_t) data;

    ngx_process = NGX_PROCESS_WORKER;
    ngx_worker = worker;
```

```
    /* 工作进程初始化 */
    ngx_worker_process_init(cycle, worker);

    ngx_setproctitle("worker process");

    /* 进程循环 */
    for (;;) {

        /* 判断是否是退出的状态，如果退出，则需要清空socket连接句柄 */
        if (ngx_exiting) {
            ngx_event_cancel_timers();

            if (ngx_event_timer_rbtree.root
                    == ngx_event_timer_rbtree.sentinel) {
                ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exiting");

                ngx_worker_process_exit(cycle);
            }
        }

        ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "worker cycle");

        /* 事件驱动核心函数 */
        ngx_process_events_and_timers(cycle);

        if (ngx_terminate) {
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exiting");

            ngx_worker_process_exit(cycle);
        }

        /* 如果是退出 */
        if (ngx_quit) {
            ngx_quit = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
                    "gracefully shutting down");
            ngx_setproctitle("worker process is shutting down");

            if (!ngx_exiting) {
                ngx_exiting = 1;
                ngx_close_listening_sockets(cycle);
                ngx_close_idle_connections(cycle);
            }
        }

        /* 如果是重启 */
        if (ngx_reopen) {
            ngx_reopen = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
            ngx_reopen_files(cycle, -1);
        }
    }
}
```

4.3.5 ngx_worker_process_init 工作进程初始化

```
/**
 * 工作进程初始化
 */
static void ngx_worker_process_init(ngx_cycle_t *cycle, ngx_int_t worker) {
    sigset_t set;
    ngx_int_t n;
    ngx_uint_t i;
    ngx_cpuset_t *cpu_affinity;
    struct rlimit rlmt;
    ngx_core_conf_t *ccf;
    ngx_listening_t *ls;

    /* 配置环境变量 */
    if (ngx_set_environment(cycle, NULL) == NULL) {
        /* fatal */
        exit(2);
    }

    /* 获取核心配置 */
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

    if (worker >= 0 && ccf->priority != 0) {
        if (setpriority(PRIO_PROCESS, 0, ccf->priority) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "setpriority(%d) failed", ccf->priority);
        }
    }

    if (ccf->rlimit_nofile != NGX_CONF_UNSET) {
        rlmt.rlim_cur = (rlim_t) ccf->rlimit_nofile;
        rlmt.rlim_max = (rlim_t) ccf->rlimit_nofile;

        if (setrlimit(RLIMIT_NOFILE, &rlmt) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "setrlimit(RLIMIT_NOFILE, %i) failed", ccf->rlimit_nofile);
        }
    }

    if (ccf->rlimit_core != NGX_CONF_UNSET) {
        rlmt.rlim_cur = (rlim_t) ccf->rlimit_core;
        rlmt.rlim_max = (rlim_t) ccf->rlimit_core;

        if (setrlimit(RLIMIT_CORE, &rlmt) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "setrlimit(RLIMIT_CORE, %O) failed", ccf->rlimit_core);
        }
    }

    /* 设置UID GROUPUID */
    if (geteuid() == 0) {
        if (setgid(ccf->group) == -1) {
```

```
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                    "setgid(%d) failed", ccf->group);
            /* fatal */
            exit(2);
        }

        if (initgroups(ccf->username, ccf->group) == -1) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                    "initgroups(%s, %d) failed", ccf->username, ccf->group);
        }

        if (setuid(ccf->user) == -1) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                    "setuid(%d) failed", ccf->user);
            /* fatal */
            exit(2);
        }
    }

    /* 设置CPU亲和性 */
    if (worker >= 0) {
        cpu_affinity = ngx_get_cpu_affinity(worker);

        if (cpu_affinity) {
            ngx_setaffinity(cpu_affinity, cycle->log);
        }
    }

#if (NGX_HAVE_PR_SET_DUMPABLE)

    /* allow coredump after setuid() in Linux 2.4.x */

    if (prctl(PR_SET_DUMPABLE, 1, 0, 0, 0) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                "prctl(PR_SET_DUMPABLE) failed");
    }

#endif

    /* 切换工作目录 */
    if (ccf->working_directory.len) {
        if (chdir((char *) ccf->working_directory.data) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "chdir(\"%s\") failed", ccf->working_directory.data);
            /* fatal */
            exit(2);
        }
    }

    sigemptyset(&set);

    /* 清除所有信号 */
    if (sigprocmask(SIG_SETMASK, &set, NULL) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                "sigprocmask() failed");
    }
```

```
    srandom((ngx_pid << 16) ^ ngx_time());

    /*
     * disable deleting previous events for the listening sockets because
     * in the worker processes there are no events at all at this point
     */
    /* 清除sokcet的监听 */
    ls = cycle->listening.elts;
    for (i = 0; i < cycle->listening.nelts; i++) {
        ls[i].previous = NULL;
    }

    /* 对模块初始化  */
    for (i = 0; cycle->modules[i]; i++) {
        if (cycle->modules[i]->init_process) {
            if (cycle->modules[i]->init_process(cycle) == NGX_ERROR) {
                /* fatal */
                exit(2);
            }
        }
    }

    /**
     *将其他进程的channel[1]关闭，自己的channel[0]关闭
     */
    for (n = 0; n < ngx_last_process; n++) {

        if (ngx_processes[n].pid == -1) {
            continue;
        }

        if (n == ngx_process_slot) {
            continue;
        }

        if (ngx_processes[n].channel[1] == -1) {
            continue;
        }

        if (close(ngx_processes[n].channel[1]) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                    "close() channel failed");
        }
    }

    if (close(ngx_processes[ngx_process_slot].channel[0]) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                "close() channel failed");
    }

#if 0
    ngx_last_process = 0;
#endif

    /**
```

```
     * 给ngx_channel注册一个读事件处理函数
     */
    if (ngx_add_channel_event(cycle, ngx_channel, NGX_READ_EVENT,
            ngx_channel_handler) == NGX_ERROR) {
        /* fatal */
        exit(2);
    }
}
```

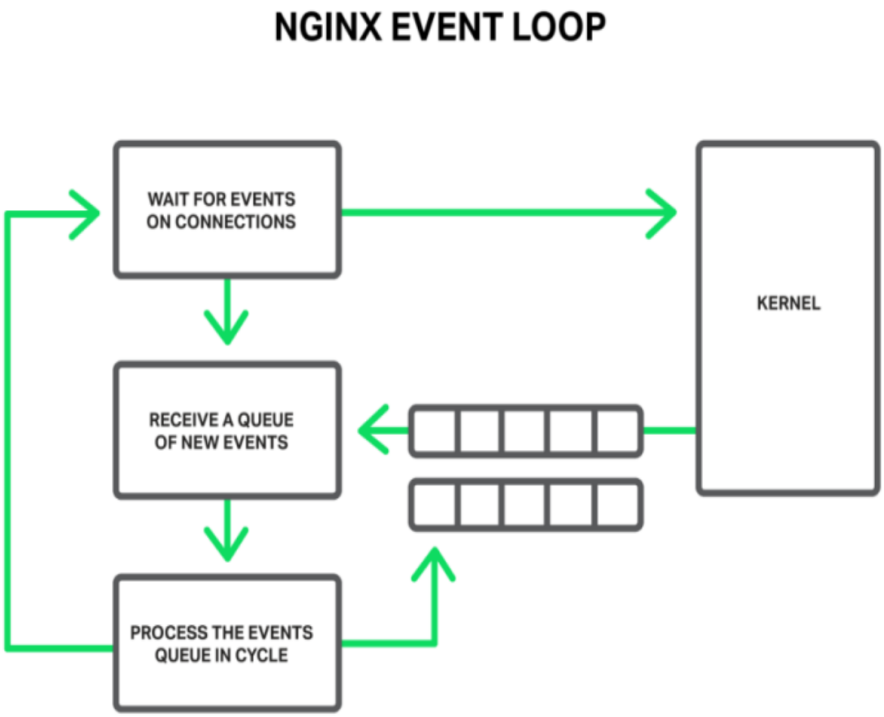
第五章 事件循环模块

nginx中的事件对象 **ngx_event_t** 提供了一种通知特定事件已经发生的机制。

ngx_event_t 中的字段包括：

- data —— 事件处理程序中使用的任意事件上下文，通常作为指向与该事件相关的连接的指针。
- handler —— 事件发生时调用的回调函数。
- write —— 指示写入事件的标志。（没有表示读取事件的标志）
- active —— 指示事件注册为接收I/O通知的标志，通常来自epoll、kqueue、poll等通知机制。
- ready —— 指示事件已经接收到I/O通知的标志。
- delayed —— 指示由于速率限制I/O延迟的标志。
- timer —— 红黑树节点，用于将事件插入到计时器树中。
- timer_set —— 指示计时器已设置且未过期的标志。
- timeout —— 指示计时器已过期的标志。
- eof —— 指示读取数据时发生EOF的标志。
- pending_eof —— 指示EOF在套接字上挂起的标志，即使在此之前可能有一些数据可用。通过EPOLLRDHUP epoll事件或EV_EOF kqueue标志传递。
- error —— 指示在读事件或写事件期间发生错误的标志。
- cancelable —— 定时器事件标志，指示在关闭辅助器时应忽略该事件。graceful worker shutdown 被延迟，直到没有不可取消的计时器事件被安排。
- posted —— 指示将事件发送到队列的标志。
- queue —— 用于将时间发送到队列的队列节点。

nginx 刚启动时，在 wait for events connections 处打开80或443端口，等待新的事件进来，比如新的客户端连接请求（epoll wait），这时 nginx 处于 sleep 状态。当操作系统接收到了一个建立TCP连接的握手报文并且处理完握手流程之后，操作系统就会通知 epoll wait，告诉他现在可以往下走了，同时唤醒worker进程。处理完一个事件之后，操作系统会把他准备好的事件放到事件队列中，从这个事件队列可以获取到一个要处理的事件，从队列中取出来，然后再开始处理事件。



5.1 I/O events

通过调用 **ngx_get_connection()** 函数获得的每个连接都有两个附加事件：**c->read** 和 **c->write**，它们用于接收套接字已经准备好进行读写的通知。所有这些事件都在 Edge-Triggered 模式下运行，这意味着它们只在套接字状态变更时触发通知。例如，在套接字上执行部分读操作并不会使 nginx 在更多数据到达套接字之前传递重复的读通知。即使底层的 I/O 通知机制本质上是 Level-Triggered （poll、select等），nginx 也会将通知转换为 Edge-Triggered . 要使 nginx 事件通知再不同平台上的所有通知系统中保持一致，必须在处理 I/O 套接字通知或调用该套接字上的任何 I/O 函数之后调用 **ngx_handle_read_event(rev, flags)** 和 **ngx_handle_write_event(wev, lowat)** 函数。通常，这些函数在每个读事件或写事件处理程序结束时调用一次。

5.2 Timer events

可以将事件设置为在超时过期时发送通知。事件使用的计时器从过去截断为 **ngx_msec_t** 类型的某个未指定点开始计算毫秒。它当前的值可以从 **ngx_current_msec** 变量获得。

函数 **ngx_add_timer(ev, timer)** 为事件设置超时，**ngx_del_timer(ev)** 删除先前设置的超时。全局超时红黑树 **ngx_event_timer_rbtree()** 储存当前设置的所有超时。树中的关键类型为 **ngx_msec_t**，表示事件发生的时间。红黑树的结构支持快速插入和删除操作，以及访问最近的超时，nginx 使用超时来确定等待 I/O 事件和过期超时事件的时间。

5.3 Posted events

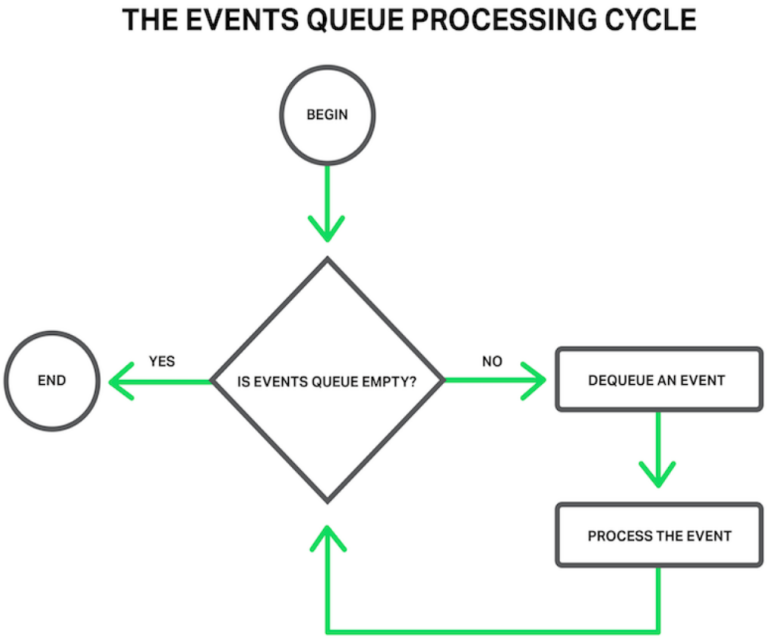
事件被发布意味着它的处理程序将在当前事件循环迭代的某个时刻被调用。发布事件可以简化代码和避免堆栈溢出。已发布的事件保存在发布队列（post queue）中。**ngx_post_event(ev, q)** 将事件 ev 发送到 post 队列 q 。**ngx_delete_posted_event(ev)** 从当前发送到的队列中删除事件 ev 。通常，事件被发送到 **ngx_posted_events** 队列中，该队列会在事件循环（event loop）的后期处理，在处理了所有的 I/O 事件和计时器事件之后。函数 **ngx_event_process_posted()** 被调用来处理事件队列。它调用事件处理程序，直到队列不为空。这意味着发布的事件处理程序可以发布更多要在当前事件循环迭代中处理的事件。

一个例子：

```
void
ngx_my_connection_read(ngx_connection_t *c)
{
    ngx_event_t  *rev;

    rev = c->read;

    ngx_add_timer(rev, 1000);

    rev->handler = ngx_my_read_handler;

    ngx_my_read(rev);
}


void
ngx_my_read_handler(ngx_event_t *rev)
{
    ssize_t            n;
    ngx_connection_t  *c;
    u_char             buf[256];

    if (rev->timedout) { /* timeout expired */ }

    c = rev->data;

    while (rev->ready) {
        n = c->recv(c, buf, sizeof(buf));

        if (n == NGX_AGAIN) {
            break;
        }

        if (n == NGX_ERROR) { /* error */ }

        /* process buf */
    }

    if (ngx_handle_read_event(rev, 0) != NGX_OK) { /* error */ }
}
```

5.4 Event loop

除了 nginx 主进程，所有 nginx 进程都执行 I/O ，因此都有一个event loop事件循环。nginx 主进程花费大部分时间在 **sigsuspend()** 调用中，等待信号到达。nginx 事件循环在 **ngx_process_events_and_timers()** 函数中实现，这个函数反复被调用，直到进程退出。



事件循环有以下几个阶段：

1. 调用 **ngx_event_find_timer()** 查找最接近到期的超时。这个函数找到计时器树中最左边的节点，并返回该节点到期的毫秒数。
2. 使用处理程序来处理 I/O 事件。这个处理程序是由 nginx 配置选择的特定的事件通知机制。这个处理程序至少等待一个 I/O 事件发生，但是只等到下一次超时过期。当发生一个读或写事件时，将设置 **ready** 标志，并且调用事件的处理程序。在 Linux 中，通常使用 **ngx_epoll_process_events()** 处理程序，该程序调用 **epoll_wait()** 来等待 I/O 事件。
3. 调用 **ngx_event_expire_timers()** 来作废计时器。从最左边的元素到最右边的元素迭代计时器树，直到找到未过期的超时。对每个过期的节点，设置 **timeout** 事件标志，重置 **timer_set** 标志，调用事件处理程序。
4. 调用 **ngx_event_process_posted()** 处理发布的事件。该函数重复地从发布的事件队列中删除第一个元素，并调用该元素的处理程序，直到队列为空。

所有的 nginx 进程也都处理信号。信号处理程序只设置调用 **ngx_process_events_and_timers()** 之后已被检查的全局变量。