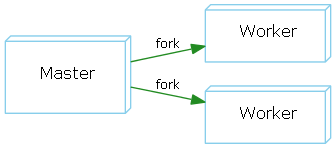
Nginx Process Model

# 概述

Nginx采用Master/Worker进程模型。Master管理Worker(s)，推荐采用一个CPU核对应一个Worker，以减少进程或线程切换开销，对于读磁盘文件多的应用，则推荐一个CPU核对应2个Worker，以便利用IO等待的时间。相对应于Apache的 “[hybrid multi-threaded multi-process](http://httpd.apache.org/docs/2.2/mod/worker.html)”，Nginx的进程模型是“single threaded multi-process”。进程模型的差异是Nginx比Apache快的一个原因。得到高性能的同时，牺牲了模块开发的简单性，增加了业务状态机的复杂性，因为单线程的模型中不能有堵塞的操作(读磁盘文件操作是堵塞的，但可以配置AIO成异步非阻塞的)。

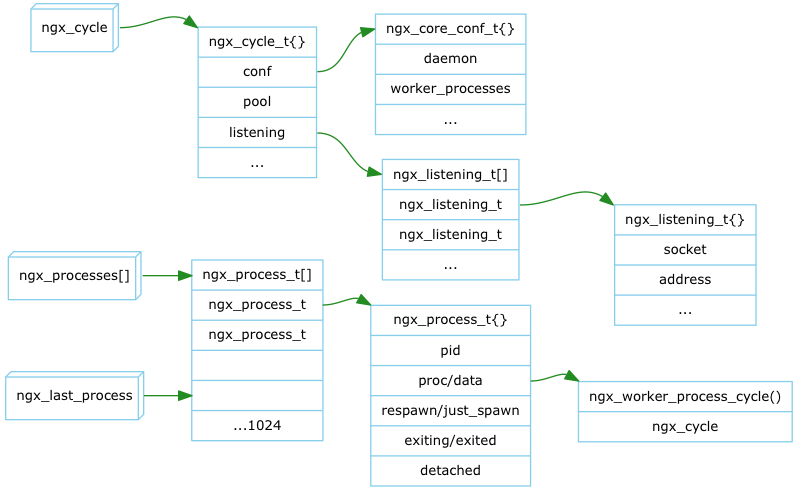
# Nginx Master/Worker



在启动时，Master进程打开Listen Socket(s)并创建若干个子进程Worker。Master此后负责监控Worker，而不处理具体的业务。

Worker同时监听Listen Socket(s)和Accepted Socket(s)，并且以异步非阻塞的方式接受新连接和处理请求。

# 数据结构



## Cycle

全局变量ngx\_cycle主要是保存了配置数据、内存管理池、Listen Socket(s)等数据。

### ngx\_core\_conf\_t

配置参数

### ngx\_pool\_t

Cycle层次的内存池，即整个生命周期都要用的数据从这个池分配

### ngx\_listening\_t

Listen Socket数据结构，关键信息是Listen Socket句柄

## Process

### ngx\_processes

全局变量ngx\_processes是一个ngx\_process\_t的数组（1024个），用以管理Wokers。

### ngx\_last\_process

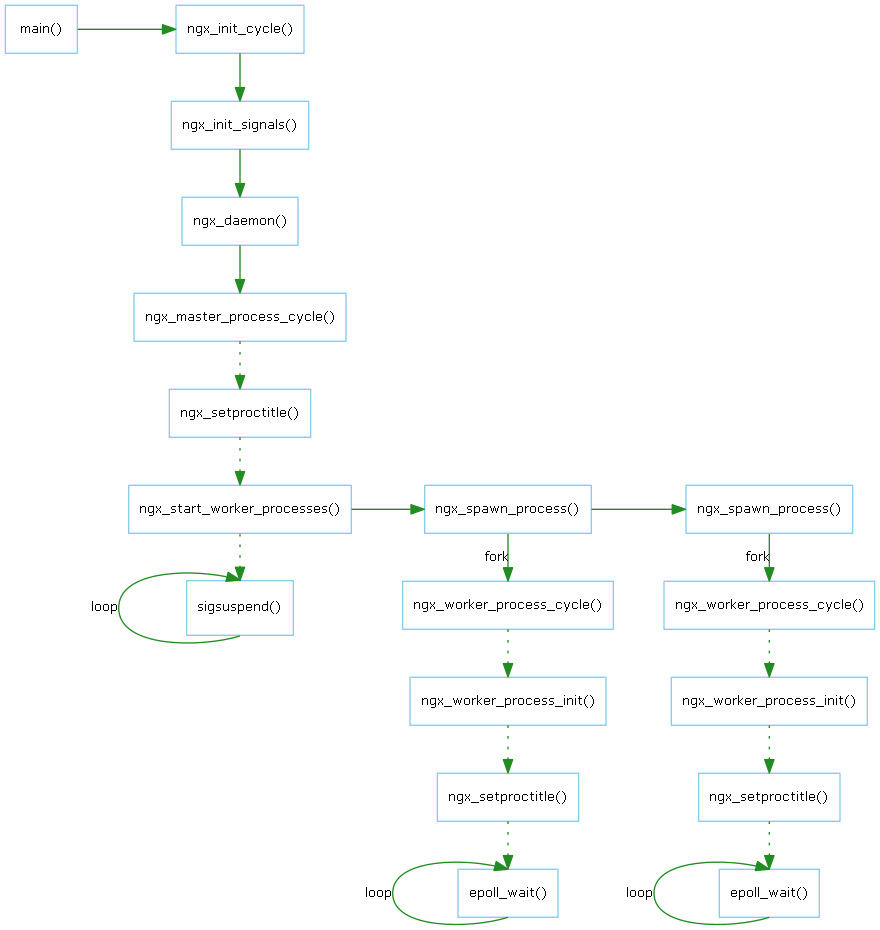
全局变量ngx\_last\_process记录ngx\_processes数组已用的位置。

### ngx\_process\_t

ngx\_process\_t 是一个关键的结构体，它抽象了一个Worker进程:

* pid : Worker的进程号， -1表示在ngx\_processes数组中空闲。
* proc/data : Worker 运行的函数和参数，它通常是ngx\_worker\_process\_cycle(ngx\_cycle)。
* respawn : 表示如果Worker退出，需要重新创建它。
* just\_spawn : 表示 Worker刚被创建，不接收Master发的信号。
* exited : 表示 Worker已经退出。
* exiting : 表示 Master命令Worker退出。
* detached : 用于upgrade binary。

# 启动过程



## ngx\_init\_cycle()

读取配置数据到ngx\_cycle中，并打开Listen Sockets

## ngx\_init\_signals()

安装信号处理函数

## ngx\_daemon()

fork后台进程，标准输入输出定向到/dev/null

## ngx\_master\_process\_cycle()

Master进程的主循环，先创建Wokers，再进入等待信号/处理信号的循环中

## ngx\_setproctitle()

设置进程的title，ps 命令显示

## ngx\_start\_worker\_processes()

创建若干个Worker，Worker个数可通过worker\_processes项配置

## ngx\_spawn\_process()

从全局变量数组ngx\_processes中找一个空位，通过系统调用fork出一个Worker，它继承了Master的信号处理、Cycle、Listen Sockets、文件句柄等。

## ngx\_worker\_process\_cycle()

Worker进程的主循环（等待事件/处理事件）

## ngx\_worker\_process\_init()

初始化Worker进程，设置资源限制、用户和组，绑定CPU

# 情境分析

## Startup

lingjf@dell:/usr/local/nginx$sudo ./nginx

PID PPID USER COMMAND

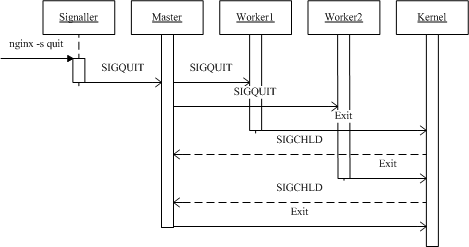
33126 1 root nginx: master process /usr/local/nginx

33127 33126 nobody nginx: worker process

33128 33126 nobody nginx: worker process

## Graceful Shutdown

lingjf@dell:/usr/local/nginx$sudo ./nginx -s quit



1. 新创建的nginx进程（Signaller）从pid文件(nginx.pid)中获得Master进程号(pid)。
2. Signaller通过系统调用kill向Master发送SIGQUIT（NGX\_SHUTDOWN\_SIGNAL）信号，并退出。
3. Master收到SIGQUIT后向所有的Worker通过kill发SIGQUIT。
4. Worker收到信号后关闭资源（Sockets、Pool等）退出(exit)。
5. OS向Master发送SIGCHLD。
6. Master收到SIGCHLD后调用系统调用waitpid()获得Worker的pid和状态并更新ngx\_processes数组(信号处理函数中)。
7. Master检查是否所有的Worker都退出，如果还有没退出，则在sigsuspend()继续等待。
8. Master删除pid文件，清理资源并退出。

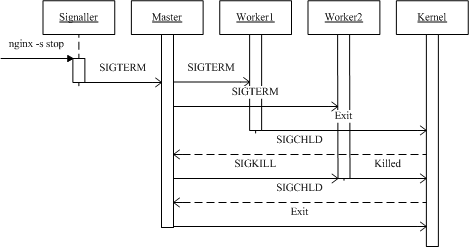
main()🡪ngx\_signal\_process()🡪ngx\_os\_signal\_process()🡪kill()🡪ngx\_signal\_handler()🡪ngx\_master\_process\_cycle()

🡪ngx\_signal\_worker\_processes()🡪kill()🡪ngx\_signal\_handler()🡪ngx\_worker\_process\_cycle()🡪ngx\_worker\_process\_exit()

🡪ngx\_signal\_handler()🡪ngx\_process\_get\_status()🡪waitpid()🡪ngx\_master\_process\_cycle()🡪ngx\_master\_process\_exit()

## Fast Shutdown

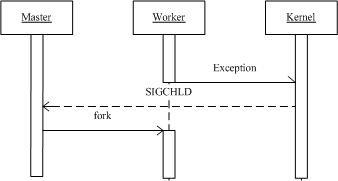
lingjf@dell:/usr/local/nginx$sudo ./nginx -s stop



过程基本和Graceful Shutdown相同，不同点有：

* 发送的信号是SIGTERM (NGX\_TERMINATE\_SIGNAL)。
* Master等待超时后向Worker发SIGKILL，强制Worker退出。

## Worker Exception

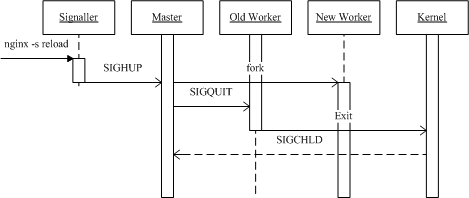


1. Worker异常退出后，OS会向Master发送SIGCHLD信号。
2. Master收到信号后会调用waitpid()获得Worker的pid和状态并更新ngx\_processes数组。
3. Master再根据ngx\_processes重新创建Worker。

ngx\_signal\_handler()🡪ngx\_worker\_process\_cycle()🡪ngx\_reap\_children()🡪ngx\_spawn\_process()

## Change configuration

lingjf@dell:/usr/local/nginx$sudo ./nginx -s reload



1. 新创建的nginx进程（Signaller）从pid文件(nginx.pid)中获得Master进程号。
2. Signaller通过系统调用kill向Master发送SIGHUP（NGX\_RECONFIGURE\_SIGNAL）信号，并退出。
3. Master收到信号后根据新的配置生成新的Cycle。
4. 创建若干个新的Worker，此时新老Worker并存。
5. Master向老Workers发送SIGQUIT(NGX\_SHUTDOWN\_SIGNAL)，并更新ngx\_processes中的exiting为1，以此来确保老Worker不会被重新创建。
6. 老Workers退出。
7. OS向Master发送SIGCHLD信号。
8. Master收到信号后，将ngx\_processes中老Worker项设为空闲（pid设为-1）。

## Rotate log files

lingjf@dell:/usr/local/nginx$sudo ./nginx -s reopen

略

## No accept

lingjf@dell:/usr/local/nginx$sudo kill -WATCH `cat logs/nginx.pid`

略

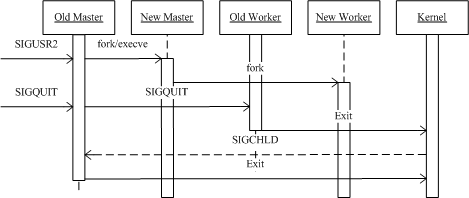
## Upgrade binary on fly

lingjf@dell:/usr/local/nginx$mv nginx nginx\_old

lingjf@dell:/usr/local/nginx$cp nginx\_new nginx

lingjf@dell:/usr/local/nginx$sudo kill -USR2 `cat logs/nginx.pid`

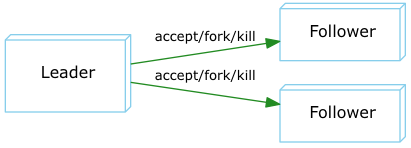
lingjf@dell:/usr/local/nginx$sudo kill -QUIT `cat logs/nginx.pid.oldbin`



1. 使用新的二进制文件替换老的文件。
2. 向老Master发送SIGUSR2(NGX\_CHANGEBIN\_SIGNAL)信号。
3. 老Master收到信号后，把Listening Sockets fd 存入环境变量NGINX。
4. 老Master把pid文件更名为nginx.pid.oldbin。
5. 老Master通过系统调用fork()、execve() 运行新的二进制文件，同时继承文件描述表。
6. 新Master解析环境变量NGINX，并继承Listening Sockets。
7. 新二进制文件启动完成。此时有新老Master和新老Workers以及新老pid文件。
8. 向老Master发送SIGQUIT信号。
9. 老Master/Workers退出。

# 其它进程模型

## Leader/Follower

Leader进程负责监听Listen Socket(s)，当有新连接到达时，Accept该新连接，创建(fork)一个子进程Follower，

该Follower专职监听Accepted Socket，并处理此连接中的请求。

请求处理完毕，Follower退出。一个请求对应一个进程实例，并且是阻塞同步的。

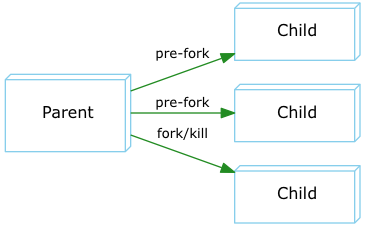
* 优点

因为各个请求处理在独立的进程，互相隔离，所以有高稳定性。

* 缺点

进程的创建和销毁资源消耗大、并发性差。

## Apache Prefork

针对Leader/Follower模型的不足，Apache Prefork模型作了部分优化。

总管理进程Parent预先创建若干个子进程(Child)，它们首先处于空闲状态，此时监听Listen Socket(s)，

一旦接受到新连接后，便不再监听Listen Socket(s)，而只监听Accepted Socket，并处理此连接中的请求，

处理完后，继续监听Listen Socket(s)。

当预先创建的Child不够时，Parent新创建Child，处理新的连接与请求。

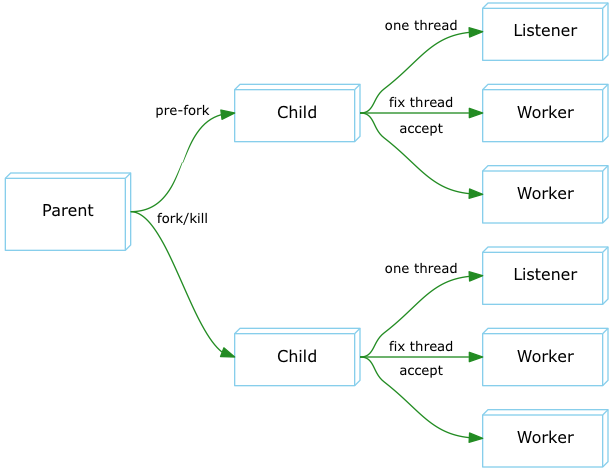
Parent根据配置，调整空闲Child的数量。

当Child处理一定数量的请求后，将退出，以释放可能泄漏的内存。

一个请求独占一个进程，并且是阻塞同步的。

优点是稳定，缺点是并发能力差。

## Apache Worker



总管理进程Parent预先创建若干个子进程(Child)，各个Child创建一个只用于监听Listen Socket(s)的线程Listener，

和若干专用于处理连接请求的线程Worker。

当Listener接收到一个Accepted Socket后，将其交给一个空闲的Worker处理，如果没有空闲的，则创建一个新的Worker。

Worker只监听Accepted Socket，并处理此连接中的请求，处理完后，又处理空闲状等待Listener给的Accepted Socket。

Child根据配置，调整空闲Worker的数量。

Parent根据配置，调整Child的数量。

当Child处理一定数量的请求后，将退出，以释放可能泄漏的内存。

一个请求独占一个线程，并且是阻塞同步的。

这是Apache主流的进程模型。稳定性和Apache Prefork模型相比，略差，因为同一进程下的线程之间是互相影响的。

并发能力比Apache Prefork模型好，因为线程比进程开销要小。但并发能力也受限于线程数。

测试benchmark显示Apache Worker模型比Nginx Master/Worker模型要差。

另一个严重的缺点是对KeepAlive的支持很差，因为要支持KeepAlive，每一个连接就要有一个线程Worker来维护，

导致线程数量过多，线程切换开销过大。

# 其它问题

## Master循环等待信号

Master循环的工作只是等待信号并处理。有二种OS API可以选择pause()和sigsuspend()。

pause() 使调用进程（线程）进入睡眠，直到有信号来而且处理函数已经返回。

sigsuspend() 临时替换信号阻塞集合并进入睡眠，直到有信号来而且处理函数已经返回。在sigsuspend()返回之前，信号阻塞集合恢复调用它之前的。

Nginx选用sigsuspend()方案，以实现信号处理过程当中不会被其它信号干拢。

首先将所有关心的信号加入信号阻塞集合。等待信号时用sigsuspend()特性解除阻塞。

有信号来，sigsuspend()返回前又阻塞信号。在真正处理信号时，因为所有信号是被阻塞的，也就不会被干拢。

## Replace a running executable in linux

Windows不可以替换正在运行的文件，但Linux是可以的。

参见：http://stackoverflow.com/questions/1712033/replacing-a-running-executable-in-linux

Linux无法直接使用cp命令覆盖原文件，而需要先删除(rm)或重命令(mv)原文件，再拷贝(cp)过去。

但是在Ubuntu12.4上无法用上述方法？

# Prototype

<https://github.com/lingjf/nginx_analyse/tree/master/prototype/nginx_process>

Prototype删除了和进程模型不相关的功能，以及出错处理等代码。

# Latest revision

https://github.com/lingjf/nginx\_analyse/tree/master/doc/

Ling Jianfa [lingjf@gmail.com](mailto:lingjf@gmail.com) <https://github.com/lingjf>