Nginx Event Mechanism

# 概述

Nginx进程模型是“single threaded multi-process”，而又需要实现高并发，则必须依赖异步事件机制。

一个HTTP请求处理过程当中总会有一些资源没有就绪，比如网络数据没有到达，数据库操作没有返回等，只有当它们就绪才能继续处理。

显然不能通过轮询的方式等待资源就绪。

在传统的线程池模型中，一个处理过程独占一个线程，等待资源时，线程被OS调度器切换，CPU运行其它线程处理其它的请求。

在Nginx进程模型中，一个进程同时处理N个HTTP请求，于是不允许堵塞在某一个资源上。

设计事件机制的目标是消除堵塞。

# 模块

* ngx\_event\_core\_module
* ngx\_select\_module
* ngx\_epoll\_module

# 文件句柄

## event_descriptorngx\_listening\_t

用于管理Listen Socket，它由配置所创建。

* socket

文件描述符，等同于fd。

* address

略

* connection

指向ngx\_connection\_t，管理该Listen Socket的Connection对象。

* handler

当该Listen Socket收到连接请求并接受后的处理函数(回调)。

## ngx\_connection\_t

用于管理Socket的Connection对象，包括Listen Socket和Monitor Socket。这里关注Connection对事件和回调函数的管理。

* socket

文件描述符，等同于fd。

* recv send

收发函数指针，间接调用系统API recv和send。

* listening

反指向ngx\_listening\_t。如果socket不是Listen Socket，则listening指向相应的ngx\_listening\_t。

* read

指向读事件 ngx\_event\_t。

* write

指向写事件 ngx\_event\_t。

## ngx\_event\_t

抽象一个事件。在Nginx中事件是发生在文件描述符(file descriptor, socket)上的事件。比如socket有数据收到。

* data

指向ngx\_connection\_t对象。

* handler

事件发生后的处理函数(回调)。

* write accept

write表示是可写事件。

accept表示是可接受事件。可接受事件实际是Listen Socket上的可读事件。

write和accept都为0时，表示是可读事件。

* active

表示文件描述符已经送入Monitor。

* ready

表示事件已经发生。

## ngx\_cycle\_t

全局变量ngx\_cycle。生命周期内的数据管理。

* listening

ngx\_listening\_t数组，管理所有的Listen Sockets。

* connectons

预分配的ngx\_connection\_t数组资源。

* read\_events

预分配的可读事件ngx\_event\_t数组资源。

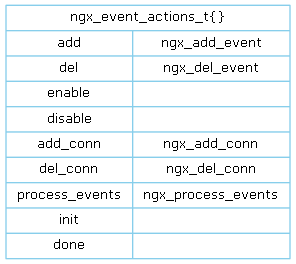
* write\_events

预分配的可写事件ngx\_event\_t数组资源。

# I/O多路复用模型

Nginx支持多种IO多路复用模型select、epoll等。 select是传统的模型，epoll是linux 2.6开始支持的更好的模型。

## 统一接口

* add

把ngx\_event\_t送入Monitor，ngx\_add\_event是一个宏定义的别名。

* del

把ngx\_event\_t从Monitor删除，ngx\_del\_event是一个宏定义的别名。

* enable

略

* disable

略

* add\_conn

把ngx\_connection\_t的read event和write event送入Monitor，ngx\_add\_conn是一个宏定义的别名。

* del\_conn

把ngx\_connection\_t的read event和write event从Monitor删除，ngx\_del\_conn是一个宏定义的别名。

* process\_events

等待事件发生，并调用ngx\_event\_t的handler处理回调函数或把ngx\_event\_t加入postpone队列，延迟处理。

ngx\_process\_events是一个宏定义的别名。

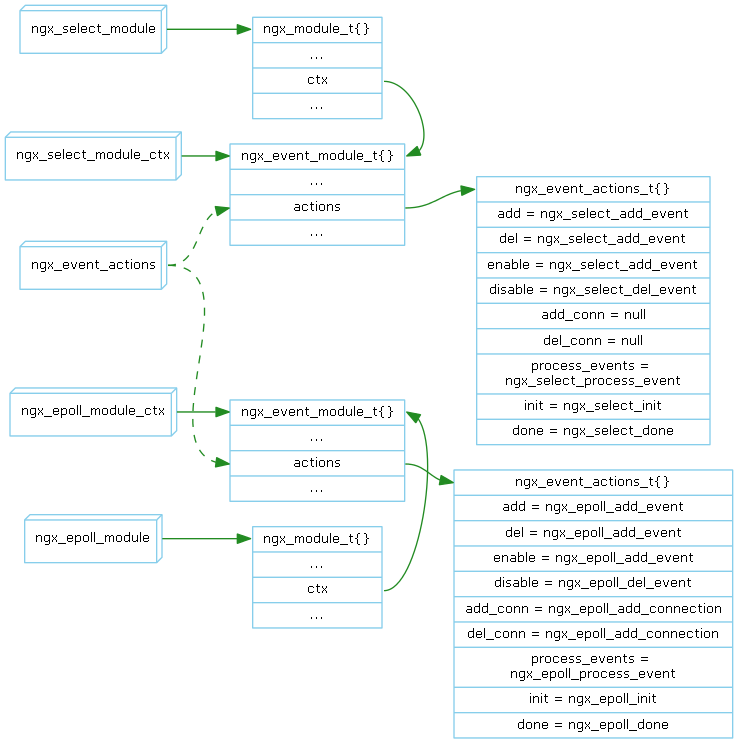
* init

初始化I/O多路复用模型。

* done

略

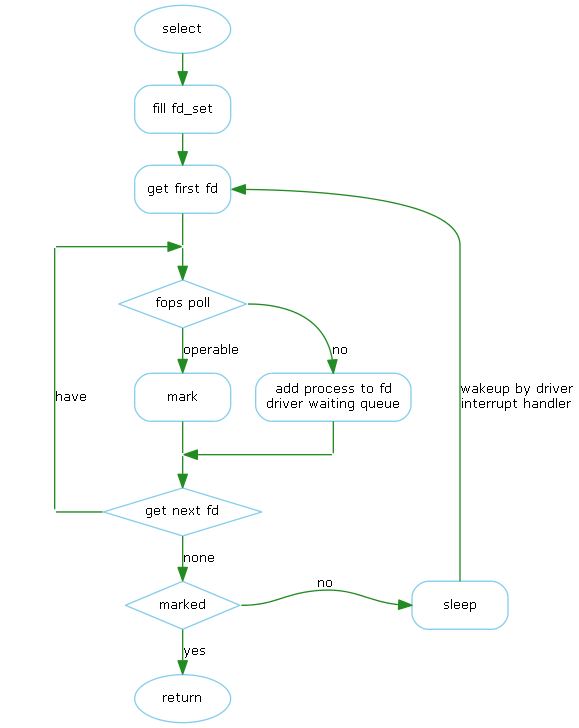
## 数据组织

各个复用模型的接口实现由ngx\_module\_t和ngx\_event\_module\_t.actions管理。

全局变更ngx\_event\_actions指向(复制)了选用的接口实现(init函数)。如图。也就是说：

不管选用的是select模型还是epoll模型，调用ngx\_event\_actions接口即可。

## select机制



1. 从用户空间复制文件描述符集合到内核空间。
2. 遍历集合，调用poll（虚拟文件系统的一个接口）查看是否可操作（可读、可写、异常）。

2.1) 文件描述符可操作，加入可读、可写、异常集合。

2.2) 文件描述符不可操作，将进程挂入文件描述符对应的文件（设备）的等待队列。

1. 如果有任何一个文件描述符上可操作，则可操作的文件描述符集合从select返回。
2. 如果没有任何文件描述符上可操作，则进入sleep状态等待。
3. 任何一个文件描述符对应的文件（设备）中断发生，中断处理函数唤醒进程。
4. 进程从sleep返回，并进入2)继续运行。

上述流程图和过程是select的内部工作原理流程。调用者过程如下：

1. 复制文件描述符集合
2. 调用select
3. 遍历可操作集合和事件集合

可以看出它有三个方面的不足：

1. 每次调用时要重复地从用户空间复制文件描述符集合到内核空间。
2. 每次调用要把进程放入各个文件描述符的等待队列，返回前又从各个等待队列中删除。
3. 每次调用时要重复地遍历文件描述符(轮询)。

## epoll机制

针对select的三个不足，epoll机制作了改进，提供了三个接口函数：epoll\_create、epoll\_ctl、epoll\_wait。

1. 调用epoll\_create在内核创建一个对象EP。
2. 调用epoll\_ctl加入文件描述符。

2.1) 将文件描述符和用户参数加入EP的容器，以便删除。

2.2) 调用poll查看是否可操作，如果可操作则将文件描述符放入EP的一个数组RD。

2.3) 并将进程挂入文件描述符对应的文件（设备）的等待队列。

2.4) 向文件描述符对应的文件（设备）Driver注册回调函数挂入的等待队列。

1. 调用epoll\_wait

3.1) 如果RD不为空，则将RD返回。

3.2) 如果RD为空，则进入sleep状态等待。

1. （数据到达网卡后）文件描述符对应的文件（设备）中断发生，调用中断处理函数。
2. 中断处理函数调用回调函数，将文件描述符放入RD，并唤醒进程
3. 进程从sleep返回，并将RD从epoll\_wait返回。
4. 调用者从RD中得到文件描述符和用户参数，从用户参数可又得到ngx\_event\_t以及ngx\_connection\_t。

由于EP的存在，所以不需重复地从用户空间复制文件描述符集合到内核空间（解决了select第一个不足），

也不需要重复地将进程放入各个文件描述符的等待队列然后删除（解决了select第二个不足）。

epoll\_wait内部以及调用者不再需要遍历集合（解决了select第三个不足）。

# 超时管理

## 时间管理

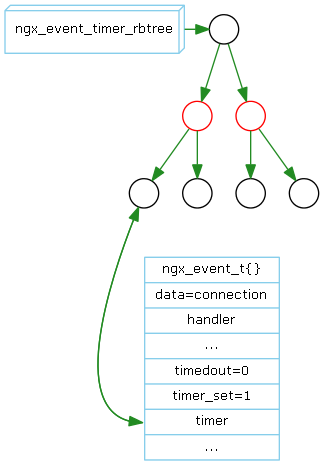
通常调用系统API gettimeofday来获得当前的时间值。由于系统调用开销问题，Nginx努力避免频繁调用gettimeofday。

Nginx以一定策略在select、epoll\_wait等阻塞函数返回之后调用gettimeofday，

并将获得的时间值保存到全局变量ngx\_cached\_time。此后读ngx\_cached\_time来获得当前时间。

这种方法会带来时间误差，但Nginx对时间精度要求并不高，这种方法是可接受的、高效的。

## 事件管理



Nginx使用红黑树来管理超时事件，并以超时时刻为KEY从早到晚排列。即最早要超时的事件在树的最左结点。

全局变量ngx\_event\_timer\_rbtree用于管理这棵红黑树。

事件ngx\_event\_t中有三个字段用于超时：

* timedout

事件是否已经超时。

* timer\_set

事件是否已经加入红黑树。

* timer

红黑树结点，将ngx\_event\_t挂进红黑树。已知它的指针可以用offsetof来算出ngx\_event\_t的地址。

## 扫描时机

### 事件触发扫描

超时事件是由排序的红黑树管理的，很容易（快）得到最早要超时的事件，即可算出第一个要超时事件距离当前的时间差。

以上述的时间差作为函数超时值调用select或epoll\_wait阻塞函数。select或epoll\_wait因事件发生或等待超时而返回后，

调用gettimeofday更新ngx\_cached\_time。如果当前时间ngx\_cached\_time大于上一次扫描红黑树的时间，则扫描红黑树，处理超时事件。

当事件很多时，select或epoll\_wait因为等到事件而频繁返回，导致频繁调用gettimeofday，和频繁扫描红黑树。

（TODO：定量分析）

于是就开发了定时触发扫描。

### 定时触发扫描

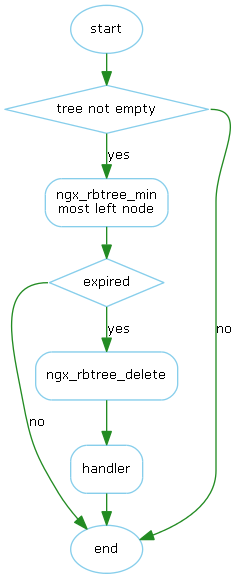
设置一个系统定时器(默认为100ms)。调用无超时的select或epoll\_wait阻塞函数。

如果select或epoll\_wait因事件发生而返回后，不调用gettimeofday更新ngx\_cached\_time，也就不会扫描红黑树。

如果select或epoll\_wait因定时器信号(SIGALRM)触发而中断返回后(SIGALRM信号处理函数中置标识)，

调用gettimeofday更新ngx\_cached\_time，当前时间ngx\_cached\_time一定大于上一次扫描红黑树的时间，扫描红黑树。

## 扫描过程



1. 红黑树没有结点，则结束扫描。
2. 如果最左结点没有超时，则结束扫描。
3. 如果最左结点超时，则把它从红黑树中删除，并调用回调处理函数。
4. 返回1)

扫描时，只需判断最左结点是否超时，而不需要扫描整棵树。

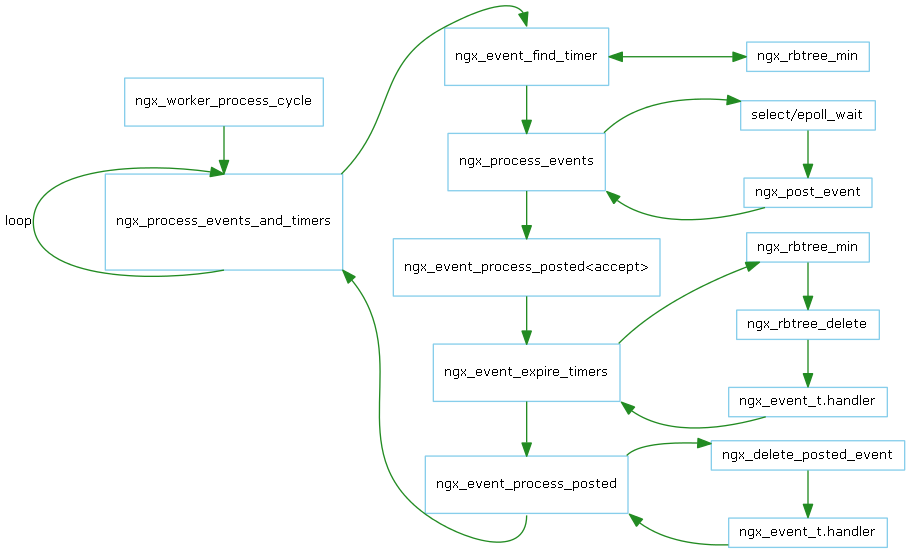
# 延迟处理

通过select或epoll\_wait收到事件后，可以把事件ngx\_event\_t先挂入ngx\_posted\_accept\_events和ngx\_posted\_events队列。

顾名思义，ngx\_posted\_accept\_events挂可接受事件，ngx\_posted\_events挂可读和可写事件。

事件全收下后，先处理ngx\_posted\_accept\_events队列中的事件，再处理超时，最后处理ngx\_posted\_events队列中的事件。

# 事件引擎



# 接受处理

1. 解析配置文件listen指令，创建ngx\_listening\_t。
2. 创建ngx\_listening\_t对应的ngx\_connection\_t和读（接受）写事件ngx\_event\_t。
3. 设置ngx\_listening\_t的回调处理函数为ngx\_http\_init\_connection。
4. 设置读（接受）事件的回调处理函数为ngx\_event\_accept。
5. 监视(Monitor)事件（select 或 epoll\_wait）。
6. Client访问（如：wget <http://127.0.0.1>）。
7. 读（接受）事件发生（Client访问的连接请求报文到达），调用回调处理函数ngx\_event\_accept。
8. ngx\_event\_accept创建新的ngx\_connection\_t和读写事件ngx\_event\_t。
9. ngx\_event\_accept调用ngx\_listening\_t的回调处理函数ngx\_http\_init\_connection。
10. ngx\_http\_init\_connection初始化读事件的回调处理函数为ngx\_init\_request，并设置读超时事件。
11. 监视(Monitor)事件（select 或 epoll\_wait）。

# 读处理

Client发送如下的HTTP Request：

GET /index.html HTTP/1.1\r\n

Content-Length: 0\r\n

\r\n

并假设上述数据被分成3个IP包在较大延时的网络上发出。

* 包1： “GET /index.html HT”
* 包2： “TP/1.1\r\nContent-Length”
* 包3： “:0\r\n\r\n”

下面讨论读处理的过程：

1. 读事件的回调函数已经设为ngx\_http\_process\_request\_line()，进程在epoll\_wait()等待网络数据。
2. 包1收到后，epoll\_wait()返回，触发读事件，并调用读事件的回调函数ngx\_http\_process\_request\_line()。
3. 读取（recv系统调用）“GET /index.html HT”到缓冲区ngx\_buf\_t。
4. 解析Request-Line = <Method> SP <URL> SP <Version>
5. “GET ”被成功地解析为Method。
6. “/index.html ”被成功地解析为URI。
7. “HT”信息不全，无法继续解析。记录解析位置后，返回。
8. 进程又在epoll\_wait()等待网络数据。这时其它的事件就有机会得到处理。
9. 包2收到后，epoll\_wait()返回，触发读事件，并调用读事件的回调函数ngx\_http\_process\_request\_line()。
10. 读取（recv系统调用）“TP/1.1\r\nContent-Length”追加到缓冲区ngx\_buf\_t。
11. 使用新读入的数据，继续解析Request-Line。
12. “HTTP/1.1”被解析为Version。
13. “\r\n”被解析为Request-Line结束，到此Request-Line解析完毕。接下来开始解析Request-Headers。
14. 替换读事件回调函数为ngx\_http\_process\_request\_headers()。
15. 调用ngx\_http\_process\_request\_headers()解析Request-Headers。
16. “Content-Length”信息不全，无法继续解析。返回。
17. 进程又在epoll\_wait()等待网络数据。这时其它的事件就有机会得到处理。
18. 包3收到后，epoll\_wait()返回，触发读事件，并调用读事件的回调函数ngx\_http\_process\_request\_headers ()。
19. 读取（recv系统调用）“:0\r\n\r\n”到缓冲区ngx\_buf\_t。
20. 使用新读入的数据，继续解析Request-Headers。
21. “Content-Length:0\r\n”被解析为一个Header。
22. “\r\n”被解析为Request-Headers结束。到此Request-Headers解析完毕。
23. 替换读事件回调函数为空操作函数。

读事件的处理是通过更换回调函数，以及记录一些处理状态的方法来实现无堵塞异步读的。

# 写处理

Server生成如下的HTTP Response：

HTTP/1.1 200 OK\r\n

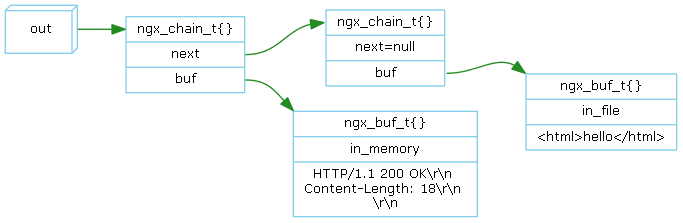
Content-Length: 18\r\n

\r\n

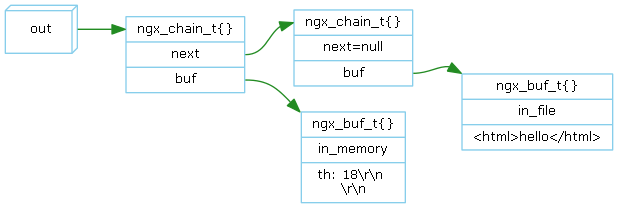
<html>hello</html>

并假设系统调用API一次最多只能发送29字节。

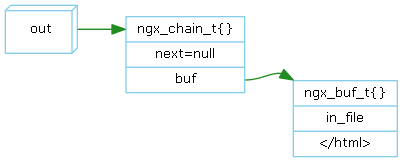
1. 读事件被初始化为Ready。待发送数据已被组织在out。



1. 调用ngx\_connection\_t::send\_chain回调函数ngx\_linux\_send\_chain()
2. 调用系统调用writev发送第一个ngx\_buf\_t “HTTP/1.1 200 OK\r\nContent-Length: 18\r\n\r\n”共计39个字节。
3. 发送出“HTTP/1.1 200 OK\r\nContent-Leng”共计29个字节。还有“th: 18\r\n\r\n”共计10字节未发送。



1. writev只发送成功部分数据，说明此时拥堵，不能继续发送，而需等待。设置buffered标枳。
2. 在ngx\_http\_finalize\_request中根据buffered标枳调用ngx\_http\_set\_write\_handler设置写事件回调函数为ngx\_http\_writer()。
3. 在epoll\_wait()等待。
4. 拥堵缓解后，触发写事件，epoll\_wait()返回，调用写事件回调函数ngx\_http\_writer()。
5. 由于带宽的限制，到此刻只能再发21字节。设置写事件定时器，在定时器到时后继续发送。
6. 调用ngx\_http\_writer\_filter()->ngx\_linux\_sendchain()。
7. 调用writev成功发送“th: 18\r\n\r\n”共计10字节。
8. 调用sendfile发送第二个文件类型的ngx\_buf\_t “<html>hello</html>”，发送成功“<html>hello”共计11字节。



1. 在epoll\_wait()等待。
2. 超时，epoll\_wait()返回，调用写事件回调函数ngx\_http\_writer()->ngx\_http\_writer\_filter()->ngx\_linux\_sendchain()。
3. 调用sendfile成功发送“</html>”共计7字节。
4. HTTP Response全部发送完毕。清除buffered标枳。

# 情境分析

处理HTTP请求（简化）情境：

1. 解析配置文件listen指令，创建ngx\_listening\_t。
2. 创建ngx\_listening\_t对应的ngx\_connection\_t和读（接受）写事件ngx\_event\_t。
3. 设置ngx\_listening\_t的回调处理函数为ngx\_http\_init\_connection。
4. 设置读（接受）事件的回调处理函数为ngx\_event\_accept。
5. 监视(Monitor)事件（select 或 epoll\_wait）。
6. Client访问（如：wget <http://127.0.0.1>）。
7. 读（接受）事件发生（Client访问的连接请求报文到达），调用回调处理函数ngx\_event\_accept。
8. ngx\_event\_accept创建新的ngx\_connection\_t和读写事件ngx\_event\_t。
9. ngx\_event\_accept调用ngx\_listening\_t的回调处理函数ngx\_http\_init\_connection。
10. ngx\_http\_init\_connection初始化读事件的回调处理函数为ngx\_init\_request，并设置读超时事件。
11. 监视(Monitor)事件（select 或 epoll\_wait）。
12. 如果读超时事件发生，则处理结束。
13. 读事件发生（Client访问的数据到达），调用回调处理函数ngx\_init\_request。
14. ngx\_init\_request创建ngx\_http\_request\_t，重设读事件的回调处理函数为ngx\_http\_process\_request\_line。
15. ngx\_init\_request直接调用ngx\_http\_process\_request\_line。
16. ngx\_http\_process\_request\_line调用系统函数recv读取数据。
17. 如果没有新数据，则读事件的回调处理函数彻底返回，重新监视。
18. 如果有新数据，则调用ngx\_http\_parse\_request\_line。
19. 如果Request-Line的数据未收全，ngx\_http\_parse\_request\_line未成功，则读事件的回调处理函数彻底返回，重新监视。
20. 如果Request-Line的数据收全，ngx\_http\_parse\_request\_line成功，重设读事件的回调处理函数为ngx\_http\_process\_request\_headers。
21. ngx\_http\_process\_request\_line直接调用ngx\_http\_process\_request\_headers。
22. ngx\_http\_process\_request\_headers调用系统函数recv读取数据。
23. 如果没有新数据，则读事件的回调处理函数彻底返回，重新监视。
24. 如果有新数据，则调用ngx\_http\_parse\_header\_line。
25. 如果Request-Headers的数据未收全，则ngx\_http\_parse\_header\_line未成功，则读事件的回调处理函数彻底返回，重新监视。
26. 如果Request-Headers的数据收全，则ngx\_http\_parse\_header\_line成功，重设读事件的回调处理函数为ngx\_http\_reader。
27. 调用系统函数send发送HTTP Response数据。
28. 如果Response数据全部发送完，则处理结束。
29. 如果Response数据未全发完，则重设写事件的回调处理函数为ngx\_http\_writer。回调处理函数彻底返回，重新监视。
30. 写事件发生，调用回调处理函数ngx\_http\_writer发送剩余数据。
31. 若干次回调ngx\_http\_writer将Response数据全部发送完。
32. 读事件发生，调用回调处理函数ngx\_http\_reader但收不到数据，说明Client关闭了连接。
33. 调用ngx\_http\_close\_connection，清理资源。

# Prototype

<https://github.com/lingjf/nginx_event.git>

# Latest revision

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