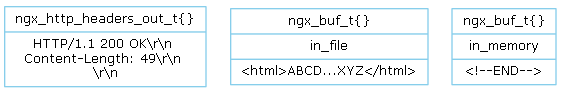
Nginx HTTP Output

# 概述

# 情境分析

## 内容生成

在Content阶段处理中，已生成要发送的Response的Headers和二个内容ngx\_buf\_t：

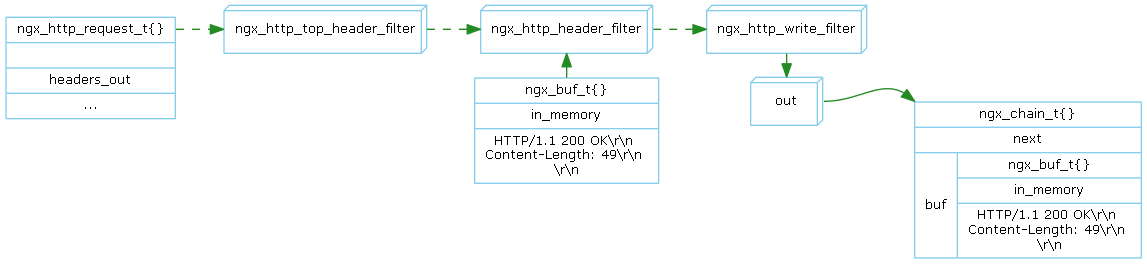


## 发送Header

将ngx\_http\_request\_t经过Header过滤器串，在ngx\_http\_header\_filter过滤器，将Status Line和Headers组成字符串，保存在ngx\_buf\_t。

ngx\_http\_write\_filter过滤器将ngx\_buf\_t挂入ngx\_http\_request::out所管理的ngx\_chain\_t。out所管理的ngx\_chain\_t是正在发送的队列。

考虑到发送效率，尽量减少系统调用，这里采用的做法是延迟并合并发送，因此数据会被缓存在out之中。



## 发送Body

将产生的数据经过Body过滤器串，最终将数据挂入out所管理的ngx\_chain\_t。

在这个例子中需要发送一个静态文件的ngx\_buf\_t。ngx\_http\_copy\_filter过滤器负责将文件ngx\_buf\_t转化为若干个内存ngx\_buf\_t。

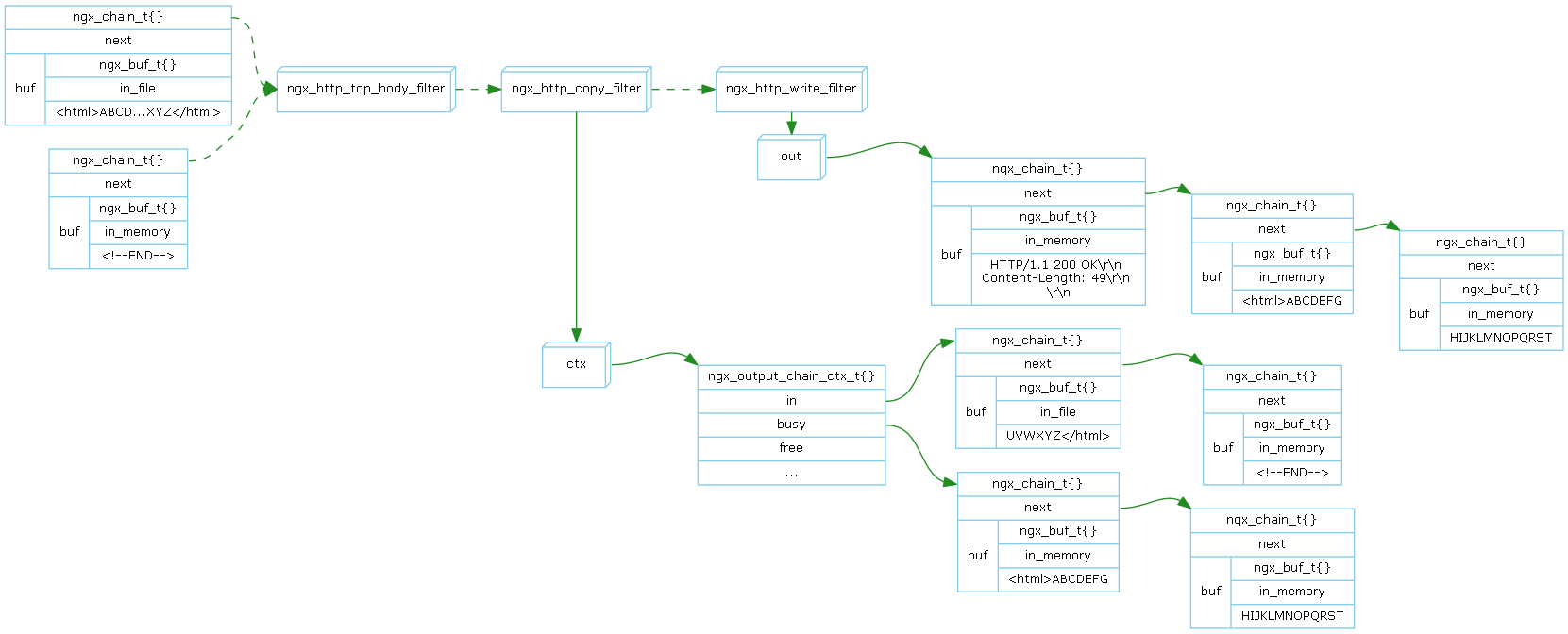
这个转化过程实际就是读文件内容到内存ngx\_buf\_t的过程。

假设这个静态文件很大（1GBytes），如果全部读入内存ngx\_buf\_t，则处理单个请求就要占用1GB的内存，显然是不合理的。

Nginx采用循环重用的策略来解决这个问题：

分配N个大小为S的ngx\_buf\_t，读文件填满N个ngx\_buf\_t，发送完一个ngx\_buf\_t，重用这个ngx\_buf\_t读取文件并发送。

N、S这二个参数是可配置的。

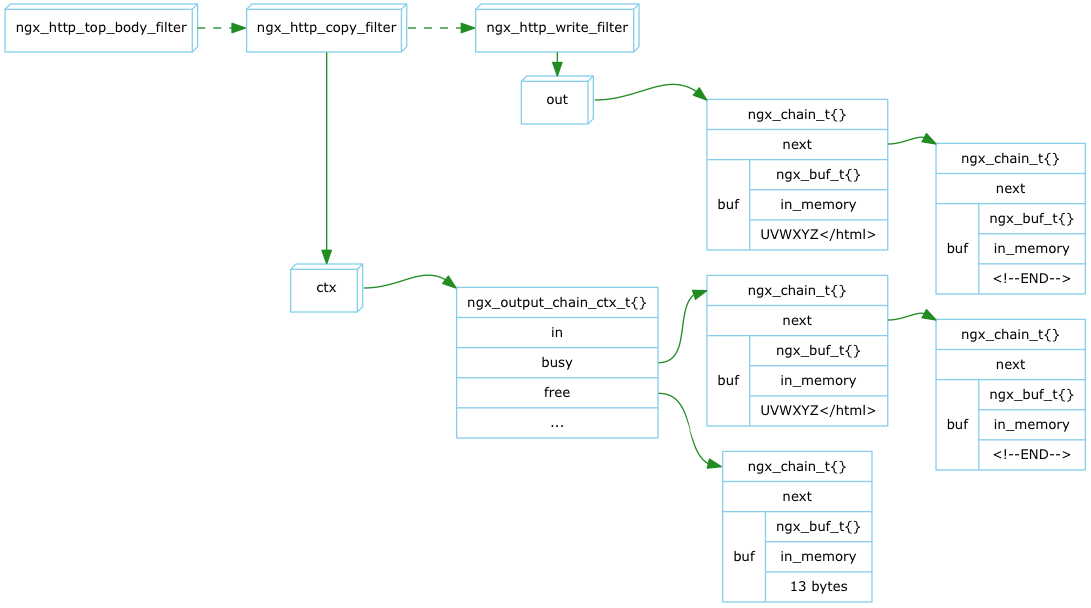


由ngx\_http\_request::ctx所管理的ngx\_output\_chain\_ctx\_t用于管理N个S大少的ngx\_buf\_t的数据结构。

* in：传入过滤器串并且没有处理的队列
* busy：正在发送的队列（ngx\_buf\_t也是out队列中）
* free：已经发送的队列，通过这个队列实现ngx\_buf\_t的重用。ngx\_chain\_t在pool中实现重用。

在这个例子中，静态文件大少为39字节，并且N配置为2，S配置为13。

发送3个ngx\_buf\_t后：



保存Headers的ngx\_buf\_t， 因为不是由ngx\_output\_chain\_ctx\_t分配和管理的，所以不会被重用（通过tag对比实现）。

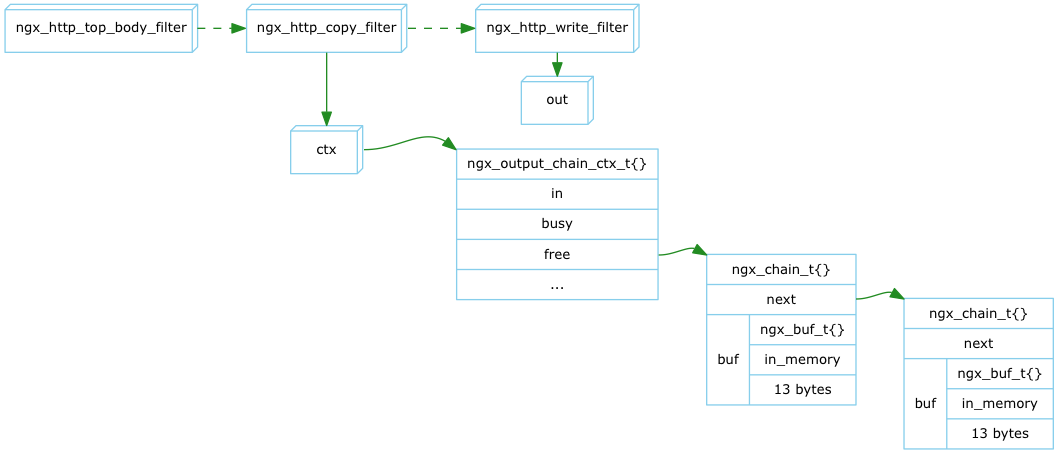
保存“<html>ABCDEFG”的ngx\_buf\_t（BUF1），发送完成，挂入free队列，等待被重用。

保存“HIJKLMNOPQRST”的ngx\_buf\_t（BUF2）和BUF1一样，也挂入free队列。

因为文件还有内容没有发送，BUF1被重用。读入文件内容“UVWXYZ</html>”，并挂入busy队列，同时挂入out所管理的ngx\_chain\_t。

因为文件已没有内容，所以BUF2没有被重用。

发送完成后：



BUF1和BUF2都挂入free队列。

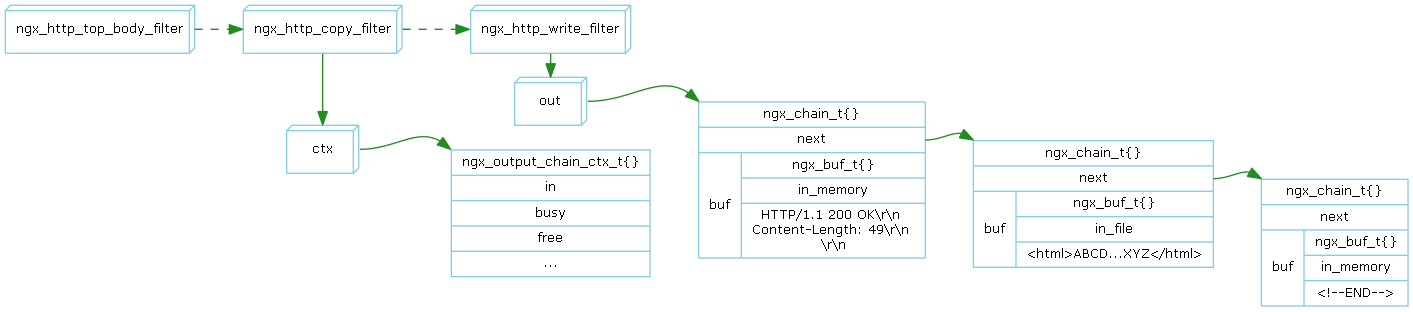
# Sendfile

在上述例子中，需要读取文件内容到内存ngx\_buf\_t，然后才能发送。这个过程是从内核空间复制数据到用户空间，很耗资源。

可以利用sendfile技术避免内核空间和用户空间的拷贝。

sendfile()在内核内从一个文件描述符拷贝内容到另一文件描述符，避免了内核空间和用户空间的拷贝，它比read/write组合效率高。

但是它也有一个缺点，就是阻塞的，阻塞在是读磁盘的IO操作。

利用sendfile()，就不再需要ngx\_http\_copy\_filter过滤器将文件ngx\_buf\_t转化为若干个内存ngx\_buf\_t。

这里讲述如何把out队列的数据通过系统调用发送到网络上。由ngx\_connection\_t的send\_chain回调函数完成。

在Linux下，send\_chain指向ngx\_linux\_sendfile\_chain()。

内存数据调用writev发送，文件数据调用sendfile发送。

合并连续的内存ngx\_buf\_t或文件ngx\_buf\_t。

成功发送的数据作为ngx\_buf\_t已处理的数据，推进pos或file\_pos指针。

# File AIO

传统的文件读写接口read/write是同步阻塞的，处理访问静态文件请求的Worker会被阻塞，从而导致后来请求积压，影响并发能力。

File AIO即文件异步读写，可以解决这个问题。

File AIO基本思路是进程发起一个读操作，不等待完成就返回，内核在正读的时候进程可以继续处理其它事务，

内核读完成后，通知进程作后续处理。

因为写有缓冲的存在，通常不会阻塞，所以这里只考虑读的情况。实际上Nginx也只是用了File Read AIO。

Linux系统上二套File AIO的实现，Nginx采用Kernel AIO。

## Glibc AIO

由glibc实现的File AIO，它通知进程作后续处理的方式有二种：

* Thread：发起读操作时，指定一个线程处理函数，内核读完成后，创建一个线程（在线程池中分配）在该线程中调用线程处理函数完成后续处理。
* Signal：发起读操作时，指定一个信号处理函数，内核读完成后，发送一个SIGIO信号给进程，调用信号处理函数完成后续处理。

## Kernel AIO

Linux在2.6.22+提供了Kernel AIO支持。它与Glibc AIO最大的差别于通知进程作后续处理的方式上。

异步的读操作和一个特殊类型(eventfd)的文件描述符绑定，epoll\_wait监听这个fd，内核读完成后，向ngx\_eventfd写数据，

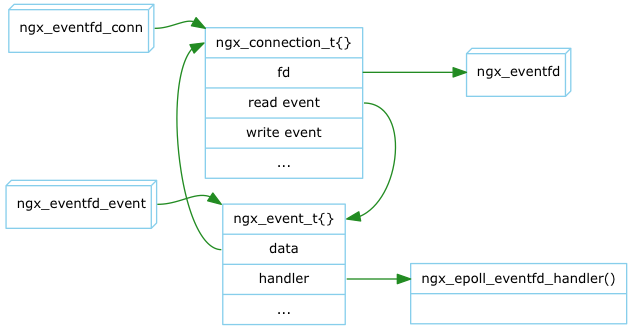
epoll\_wait便能获得通知。

由于eventfd通知了文件读写事件，相当于Socket（网络套接字）和Fd（磁盘文件描述符）可以统一由epoll\_wait监听，

在epoll\_wait处实现多路复用。

### 创建eventfd

由epoll模块的ngx\_epoll\_init()🡪ngx\_epoll\_aio\_init()创建完成。



* ngx\_eventfd

由系统调用SYS\_eventfd()创建的文件描述符全局变量。

* ngx\_eventfd\_conn

管理ngx\_eventfd的ngx\_connection\_t。

* ngx\_eventfd\_event

ngx\_eventfd上的读事件。

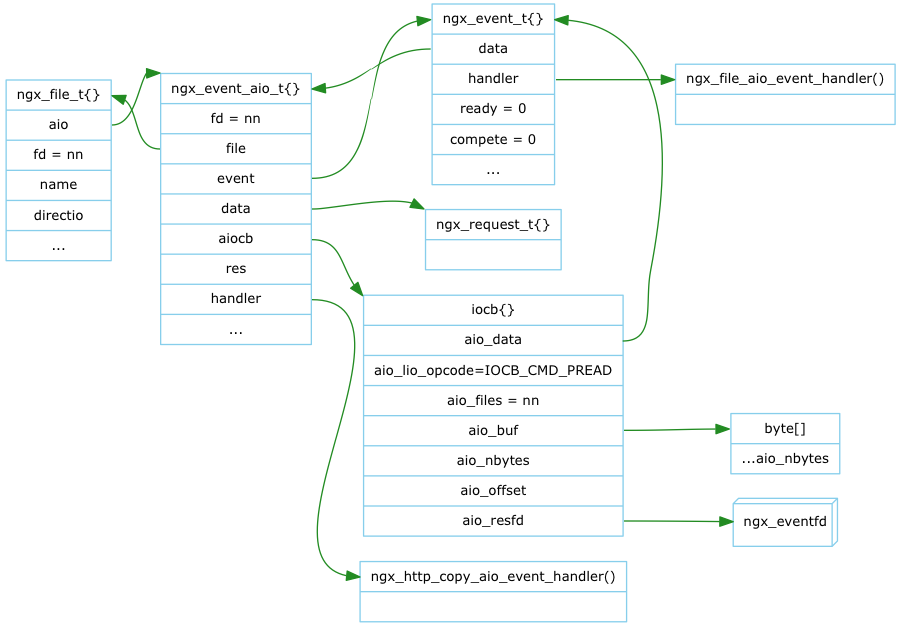
在epoll\_wait监听ngx\_eventfd\_event事件。

### 发起读操作

在发送Response过程中，ngx\_http\_copy\_filter过滤器负责将文件ngx\_buf\_t转化为若干个内存ngx\_buf\_t，

实际上是读文件内容到内存ngx\_buf\_t。由ngx\_linux\_aio\_read.c中的ngx\_file\_aio\_read()调用系统调用SYS\_submit发起AIO读操作，发送Response中断退出。

发起AIO读操作后：

* ngx\_file\_t

所读文件。

* ngx\_event\_aio\_t

文件异步读写数据结构。

* ngx\_event\_t

文件AIO事件，为区别ngx\_eventfd\_event事件，称它为file\_aio\_event。

状态设成未就绪 （ready = 0），和未完成 （complete = 0）。

* iocb

Kernel AIO实现数据结构。

* aio\_data：用户数据，可以是回调函数，Nginx中指向file\_aio\_event事件。
* aio\_lio\_opcode：是读还是写的标枳。
* aio\_files：所读文件描述符。
* aio\_buf：读入缓冲区。
* aio\_nbytes：读取多少字节。
* aio\_offset：从文件偏移开始读。
* aio\_resfd：用于通知的eventfd文件描述符，Nginx中是ngx\_eventfd。

### 读完成通知

1. 内核读完成后，向ngx\_eventfd写数据。
2. epoll\_wait监听到ngx\_eventfd可读。
3. ngx\_eventfd的读事件ngx\_eventfd\_event触发，调用回调函数ngx\_epoll\_eventfd\_handler()。
4. 调用系统调用SYS\_getevents()得到iocb中的aio\_data即file\_aio\_event事件和读写结果。
5. 将读写结果存入ngx\_event\_aio\_t的res中。
6. 将file\_aio\_event事件设为就绪状态 （ready = 1），和完成状态 （complete = 1）。
7. 触发file\_aio\_event事件，并放入ngx\_posted\_events队列，延迟处理。
8. ngx\_eventfd\_event事件处理结束。
9. 处理file\_aio\_event事件，调用回调函数ngx\_file\_aio\_event\_handler()。
10. 调用ngx\_event\_aio\_t中的handler即ngx\_http\_copy\_aio\_event\_handler()。
11. 获得TCP Connection的ngx\_connection\_t下的写事件，并调用写事件handler开始后续处理。

### 后续处理

1. ngx\_http\_writer()🡪ngx\_http\_top\_body\_filter()🡪ngx\_http\_copy\_filter()
2. 再次调用ngx\_file\_aio\_read()发起AIO读操作读上次相同的内容，因为已经读完成，所以返回成功。
3. 和普通的同样处理。

# Upstream Output

Ctx(ngx\_output\_chain\_ctx\_t)由ngx\_http\_upstream\_t::output所管理。

Out(ngx\_chain\_t)由ngx\_http\_upstream\_t::writer::out所管理。

# Prototype

暂无

# Latest revision

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