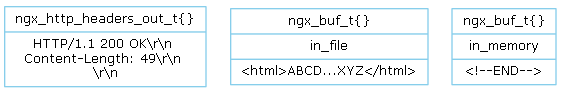
Nginx HTTP Output

# 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：

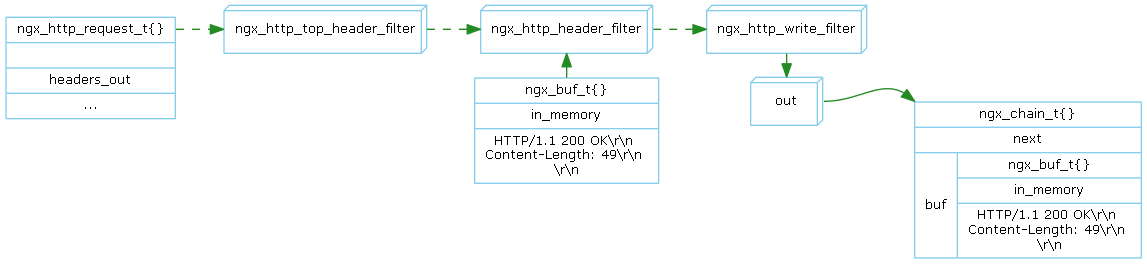


## 发送Header

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。

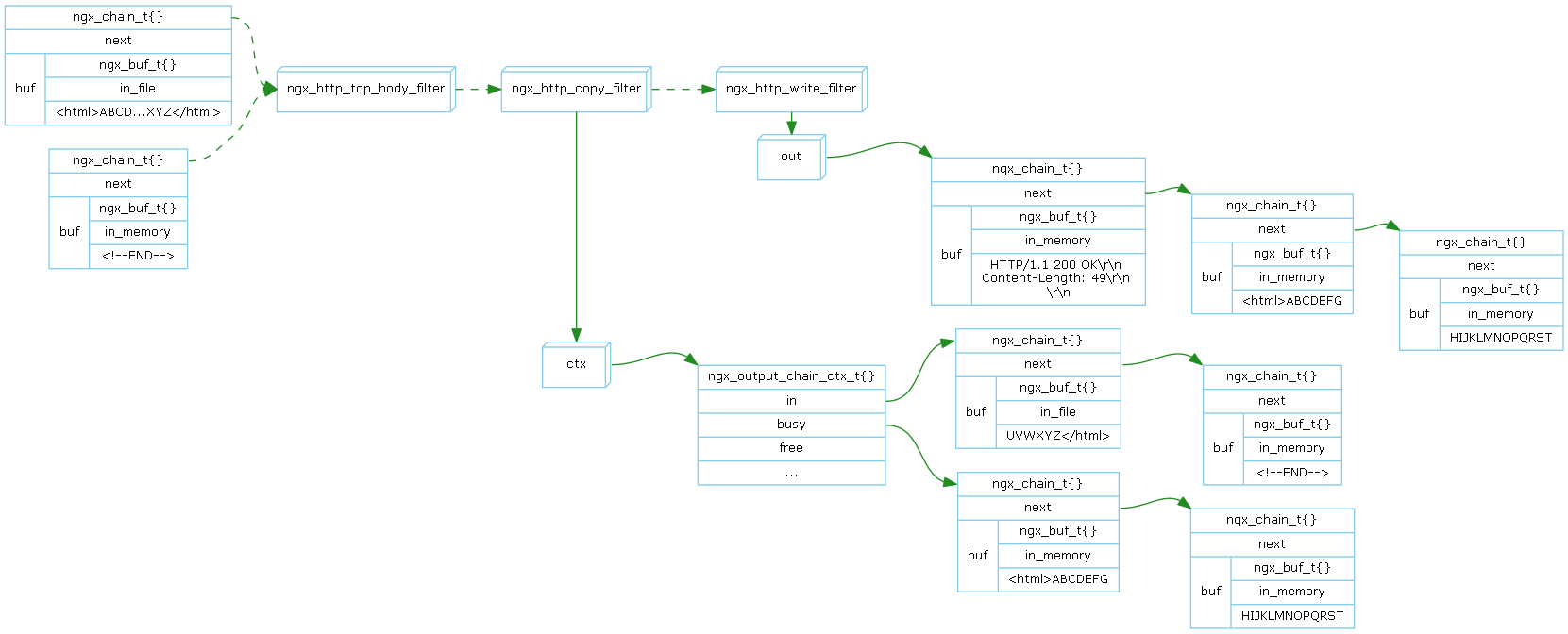
这个转化过程实际就是读文件内容到内存ngx\_buf\_t的过程。

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。

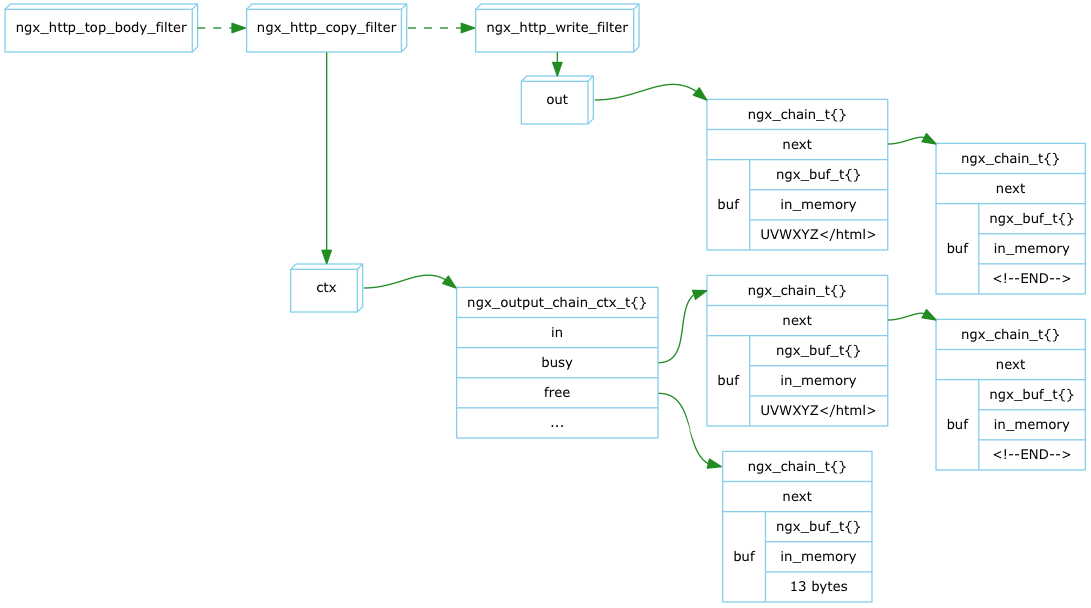


由ngx\_http\_request::ctx所管理的ngx\_output\_chain\_ctx\_t用于管理N个S大少的ngx\_buf\_t的数据结构。

* 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）。

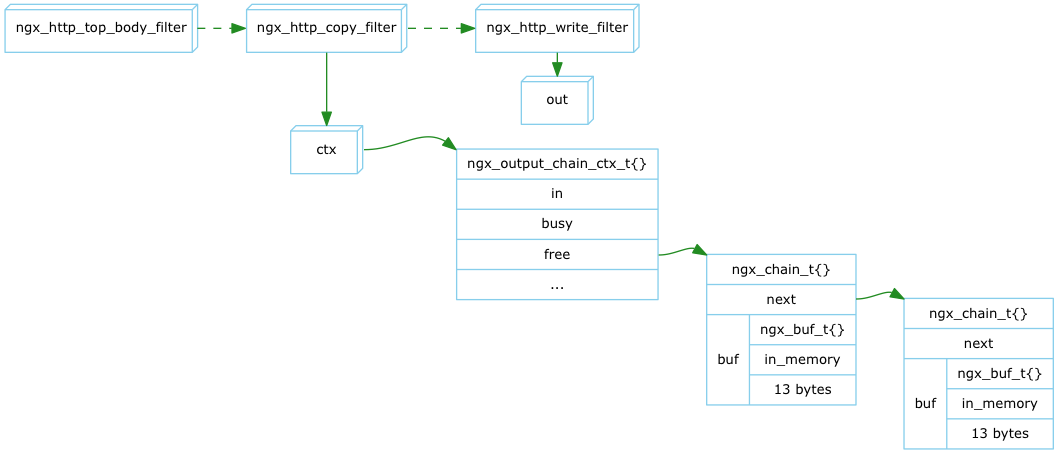
保存“<html>ABCDEFG”的ngx\_buf\_t（BUF1），发送完成，挂入free队列，等待被重用。

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：



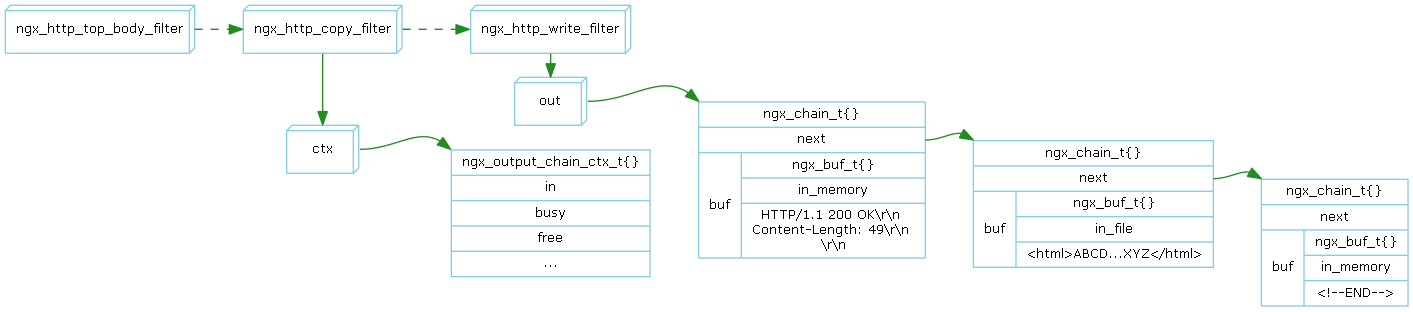
BUF1和BUF2都挂入free队列。

# 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。

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

# AIO

TODO

# 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/blob/master/doc/