理解 Nginx 源码

博客选编

前言

> 原文出处：理解 Nginx 源码 作者：chenhanzhun

理解 Nginx 源码

> 本专栏是对 Nginx 高性能服务器源码的学习与理解。根据对其源码的解读，了解Nginx服务器的基本框架。

Nginx 配置文件

概述

Nginx 是使用一个 master 进程来管理多个 worker 进程提供服务。master 负责管理 worker 进程，而 worker 进程则提供真正的客户服务，worker 进程的数量一般跟服务器上 CPU 的核心数相同，worker 之间通过一些进程间通信机制实现负载均衡等功能。Nginx 进程之间的关系可由下图表示：

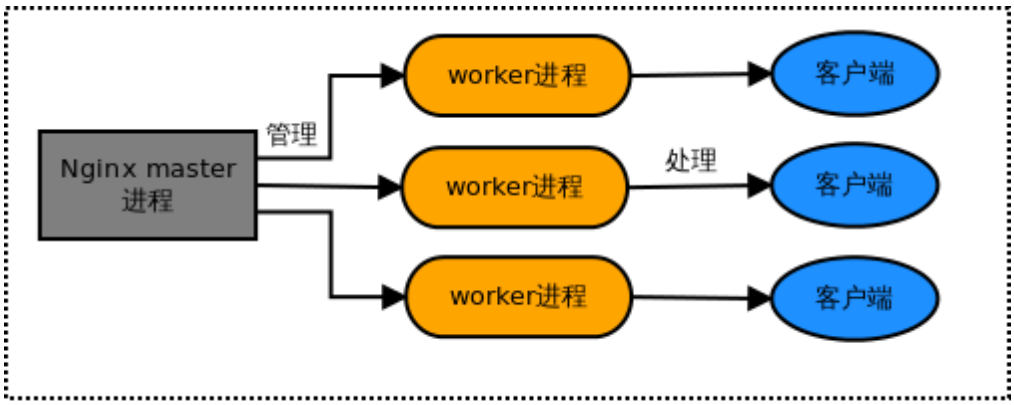


图 1 Nginx 进程间的关系

Nginx 服务启动时会读入配置文件，后续的行为则按照配置文件中的指令进行。Nginx 的配置文件是纯文本文件，默认安装 Nginx 后，其配置文件均在 /usr/local/nginx/conf/ 目录下。其中，nginx.conf 为主配置文件。配置文件中以 # 开始的行，或者是前面有若干空格或者 TAB 键，然后再跟 # 的行，都被认为是注释。这里只是了解主配置文件的结构。

Nginx 配置文件是以 block（块）形式组织，每个 block 都是以一个块名字和一对大括号 “{}” 表示组成，block 分为几个层级，整个配置文件为 main 层级，即最大的层级；在 main 层级下可以有 event、http 、mail 等层级，而 http 中又会有 server block，server block中可以包含 location block。即块之间是可以嵌套的，内层块继承外层块。最基本的配置项语法格式是“配置项名 配置项值1 配置项值2 配置项值3 ... ” ；

每个层级可以有自己的指令（Directive），例如 worker_processes 是一个main层级指令，它指定 Nginx 服务的 Worker 进程数量。有的指令只能在一个层级中配置，如worker_processes 只能存在于 main 中，而有的指令可以存在于多个层级，在这种情况下，子 block 会继承 父 block 的配置，同时如果子block配置了与父block不同的指令，则会覆盖掉父 block 的配置。指令的格式是“指令名 参数1 参数2 ... 参数N;” ，注意参数间可用任意数量空格分隔，最后要加分号。

下图是 Nginx 配置文件通常结构图示。

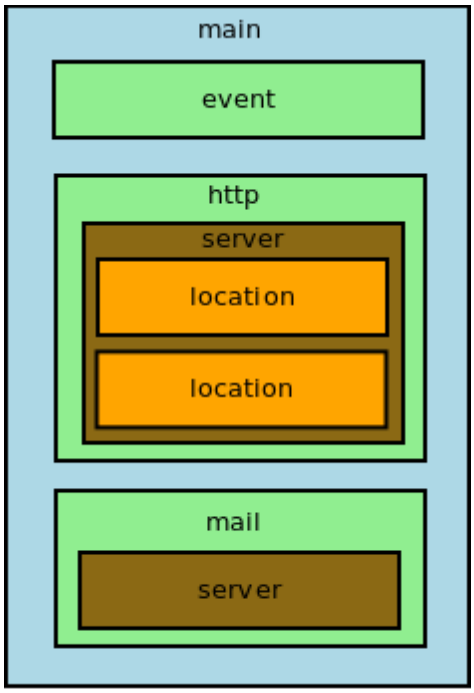


图 1 Nginx配置文件的层次结构

Nginx 服务的基本配置项

Nginx 服务运行时，需要加载几个核心模块和一个事件模块，这些模块运行时所支持的配置项称为基本配置；基本配置项大概可分为以下四类：

- 用于调试、定位的配置项；
- 正常运行的必备配置项；
- 优化性能的配置项；
- 事件类配置项；

各个配置项的具体实现如下：

```
/* Nginx 服务基本配置项 */

/* 用于调试、定位的配置项 */

#以守护进程 Nginx 运行方式
#语法：daemon off | on;
#默认：daemon on;

#master / worker 工作方式
#语法：master_process on | off;
#默认：master_process on;

#error 日志设置
#                路径        错误级别
#语法：error_log    /path/file  level;
#默认：error_log    logs/error.log  error;
#其中/path/file是一个具体文件；level是日志的输出级别，其取值如下：
#   debug info notice warn error crit alert emerg
#从左至右级别增大；若设定一个级别后，则在输出的日志文件中只输出级别大于或等于已设定的级别；
```

```
#处理特殊调试点
#语法：debug_points [stop | abort]
#这个设置是来跟踪调试 Nginx 的；

#仅对指定的客户端输出 debug 级别的日志
#语法：debug_connection [IP | DIR]

#限制 coredump 核心转储文件的大小
#语法：worker_rlimit_core   size;

#指定 coredump 文件的生成目录
#语法：working_directory    path;

/* 正常运行的配置项 */

#定义环境变量
#语法：env  VAR | VAR=VALUE;
#VAR 是变量名，VALUE 是目录；

#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 Nginx 的 nginx.conf 文件中；

#pid 的文件路径
#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；

#Nginx worker 运行的用户及用户组
#语法：user username    [groupname];
#默认：user nobody nobody;

#指定 Nginx worker进程可打开最大句柄个数
#语法：worker_rlimit_nofile limit;

#限制信号队列
#语法：worker_rlimit_sigpending limit;
#设置每个用户发给 Nginx 的信号队列大小，超出则丢弃；

/* 优化性能配置项 */

#Nginx worker 进程的个数
#语法：worker_process   number;
#默认：worker_process   1;

#绑定 Nginx worker 进程到指定的 CPU 内核
#语法：worker_cpu_affinity  cpumask [cpumask...]

#SSL 硬件加速
#语法：ssl_engine   device;

#系统调用 gettimeofday 的执行频率
#语法：timer_resolution t;

#Nginx worker 进程优先级设置
```

```
#Nginx worker 进程优先级设置
#语法：worker_priority  nice;
#默认：worker_priority  0;

/* 事件类配置项  */
#一般有以下几种配置：
#1、是否打开accept锁
#   语法格式：accept_mutex [on | off];

#2、lock文件的路径
#   语法格式：lock_file  path/file;

#3、使用accept锁后到真正建立连接之间的延迟时间
#   语法格式：accept_mutex_delay Nms;

#4、批量建立新连接
#   语法格式：multi_accept [on | off];
#
#5、选择事件模型
#   语法格式：use [kqueue | rtisg | epoll | /dev/poll | select | poll | eventport];

#6、每个worker进行的最大连接数
#   语法格式：worker_connections number;
```

HTTP 核心模块的配置

具体可以参看《Nginx 中 HTTP 核心模块配置》

```
/* HTTP 核心模块配置的功能 */

/* 虚拟主机与请求分发 */

#监听端口
#语法：listen   address:port[default | default_server | [backlong=num | rcvbuf=size | sndbuf=size |
# accept_filter | deferred | bind | ipv6only=[on | off] | ssl]];
# 默认：listen:80;
# 说明：
#   default或default_server：将所在的server块作为web服务的默认server块；当请求无法匹配配置文件中的所有主机名时，就会选择默认的虚拟主机；
#   backlog=num：表示 TCP 中backlog队列存放TCP新连接请求的大小，默认是-1，表示不予设置；
#   rcvbuf=size：设置监听句柄SO_RCVBUF的参数；
#   sndbuf=size：设置监听句柄SO_SNDBUF的参数；
#   accept_filter：设置accept过滤器，只对FreeBSD操作系统有用；
#   deferred：设置该参数后，若用户发起TCP连接请求，并且完成TCP三次握手，但是若用户没有发送数据，则不会唤醒worker进程，直到发送数据；
#   bind：绑定当前端口 / 地址对，只有同时对一个端口监听多个地址时才会生效；
#   ssl：在当前端口建立的连接必须基于ssl协议；
#配置块范围：server

#主机名称
#语法：server_name  name[...];
#默认：server_name  "";
#配置块范围：server
```

```
#server name 是使用散列表存储的
#每个散列桶占用内存大小
#语法：server_names_hash_bucket_size    size;
#默认：server_names_hash_bucker_size    32|64|128;
#
#散列表最大bucket数量
#语法：server_names_hash_max_size   size;
#默认：server_names_hash_max_size   512;
#默认：server_name_in_redirect  on;
#配置块范围：server、http、location

#处理重定向主机名
#语法：server_name_in_redirect  on | off;
#默认：server_name_in_redirect  on;
#配置块范围：server、http、location

#location语法：location[= | ~ | ~* | ^~ | @] /uri/ {}
#配置块范围：server
      #location尝试根据用户请求中的URI来匹配 /uri表达式，若匹配成功，则执行{}里面的配置来处理用户请求
#以下是location的一般配置项
#1、以root方式设置资源路径
#   语法格式：root path;
#2、以alias方式设置资源路径
#   语法格式：alias path;
#3、访问首页
#   语法格式：index file...;
#4、根据HTTP返回码重定向页面
#   语法格式：error_page code [code...] [= | =answer-code] uri | @named_location;
#5、是否允许递归使用error_page
#   语法格式：recursive_error_pages [on | off];
#6、try_files
#   语法格式：try_files path1 [path2] uri;

/* 文件路径的定义 */

#root方式设置资源路径
#语法：root path;
#默认：root html;
#配置块范围：server、http、location、if

#以alias方式设置资源路径
#语法：alias path;
#配置块范围：location

#访问主页
#语法：index    file...;
#默认：index    index.html;
#配置块范围：http、server、location

#根据HTTP返回码重定向页面
#   语法：error_page code [code...] [= | =answer-code] uri | @named_location;
#配置块范围：server、http、location、if
```

```
#是否允许递归使用error_page
#   语法：recursive_error_pages [on | off];
#配置块范围：http、server、location

#try_files
#   语法：try_files path1 [path2] uri;
#配置块范围：server、location

/* 内存及磁盘资源分配 */

# HTTP 包体只存储在磁盘文件中
# 语法：client_body_in_file_only    on | clean | off;
# 默认：client_body_in_file_only  off;
# 配置块范围：http、server、location

# HTTP 包体尽量写入到一个内存buffer中
# 语法：client_body_single_buffer   on | off;
# 默认：client_body_single_buffer   off;
# 配置块范围：http、server、location

# 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;
# 配置块范围：http、server

# 存储超大 HTTP 头部的内存buffer大小
# 语法：large_client_header_buffer_size   number    size;
# 默认：large_client_header_buffer_size   4   8k;
# 配置块范围：http、server

# 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;
# 配置块范围：http、server、location

# 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;
# 配置块范围：http、server、location

# 存储 TCP 成功建立连接的内存池大小
# 语法：connection_pool_size    size;
# 默认：connection_pool_size    256;
# 配置块范围：http、server

# 存储 TCP 请求连接的内存池大小
# 语法：request_pool_size    size;
# 默认：request_pool_size    4k;
# 配置块范围：http、server

/* 网络连接设置 */

# 读取 HTTP 头部的超时时间
# 语法：client_header_timeout   time;
# 默认：client_header_timeout   60;
```

```
# 默认：client_header_timeout  60;
# 配置块范围：http、server、location

# 读取 HTTP 包体的超时时间
# 语法：client_body_timeout   time;
# 默认：client_body_timeout   60;
# 配置块范围：http、server、location

# 发送响应的超时时间
# 语法：send_timeout   time;
# 默认：send_timeout   60;
# 配置块范围：http、server、location

# TCP 连接的超时重置
# 语法：reset_timeout_connection   on | off;
# 默认：reset_timeout_connection   off;
# 配置块范围：http、server、location

# 控制关闭 TCP 连接的方式
# 语法：lingering_close off | on | always;
# 默认：lingering_close on;
# 配置块范围：http、server、location
# always 表示关闭连接之前无条件处理连接上所有用户数据；
# off 表示不处理；on 一般会处理；

# lingering_time
# 语法：lingering_time   time;
# 默认：lingering_time   30s;
# 配置块范围：http、server、location

# lingering_timeout
# 语法：lingering_timeout   time;
# 默认：lingering_time   5s;
# 配置块范围：http、server、location

# 对某些浏览器禁止keepalive功能
# 语法：keepalive_disable   [mise6 | safari | none]...
# 默认：keepalive_disable   mise6  safari;
# 配置块范围：http、server、location

# keepalive超时时间
# 语法：keepalive_timeout   time;
# 默认：keepalive_timeout   75;
# 配置块范围：http、server、location

# keepalive长连接上允许最大请求数
# 语法：keepalive_requests  n;
# 默认：keepalive_requests  100;
# 配置块范围：http、server、location

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

```
# tcp_nopush
# 语法：tcp_nopush on | off;
# 默认：tcp_nopush off;
# 配置块范围：http、server、location

/* MIME 类型设置 */

# MIME type 与文件扩展的映射
# 语法：type{...}
# 配置块范围：http、server、location
# 多个扩展名可映射到同一个 MIME type

# 默认 MIME type
# 语法：default_type    MIME-type;
# 默认：default_type    text/plain;
# 配置块范围：http、server、location

# type_hash_bucket_size
# 语法：type_hash_bucket_size   size;
# 默认：type_hash_bucket_size   32 | 64 | 128;
# 配置块范围：http、server、location

# type_hash_max_size
# 语法：type_hash_max_size   size;
# 默认：type_hash_max_size   1024;
# 配置块范围：http、server、location

/* 限制客户端请求 */

# 按 HTTP 方法名限制用户请求
# 语法：limit_except    method...{...}
# 配置块：location
# method 的取值如下：
# GET、HEAD、POST、PUT、DELETE、MKCOL、COPY、MOVE、OPTIONS、
# PROPFIND、PROPPATCH、LOCK、UNLOCK、PATCH

# HTTP 请求包体的最大值
# 语法：client_max_body_size    size;
# 默认：client_max_body_size    1m;
# 配置块范围：http、server、location

# 对请求限制速度
# 语法：limit_rate  speed;
# 默认：limit_rate  0;
# 配置块范围：http、server、location、if
# 0 表示不限速

# limit_rate_after规定时间后限速
# 语法：limit_rate_after  time;
# 默认：limit_rate_after    1m;
# 配置块范围：http、server、location、if

/* 文件操作的优化 */

# sendfile系统调用
```

```
# sendfile 系统调用
# 语法：sendfile    on | off;
# 默认：sendfile    off;
# 配置块：http、server、location

# AIO 系统调用
# 语法：aio on | off;
# 默认：aio off;
# 配置块：http、server、location

# directio
# 语法：directio    size | off;
# 默认：directio    off;
# 配置块：http、server、location

# directio_alignment
# 语法：directio_alignment    size;
# 默认：directio_alignment    512;
# 配置块：http、server、location

# 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

# 是否缓存打开文件的错误信息
# 语法：open_file_cache_errors  on | off;
# 默认：open_file_cache_errors  off;
# 配置块：http、server、location

# 不被淘汰的最小访问次数
# 语法：open_file_cache_min_user  number;
# 默认：open_file_cache_min_user  1;
# 配置块：http、server、location

# 检验缓存中元素有效性的频率
# 语法：open_file_cache_valid  time;
# 默认：open_file_cache_valid  60s;
# 配置块：http、server、location

/* 客户请求的特殊处理 */

# 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

# 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

# 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

# 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

# 是否合并相邻的“/”
# 语法：merge_slashes    on | off;
# 默认：merge_slashes    on;
# 配置块：http、server、location

# DNS解析地址
# 语法：resolver     address...;
# 配置块：http、server、location

# DNS解析的超时时间
# 语法：resolver_timeout     time;
# 默认：resolver_timeout     30s;
# 配置块：http、server、location

# 返回错误页面是否在server中注明Nginx版本
# 语法：server_tokens    on | off;
# 默认：server_tokens    on;
# 配置块：http、server、location
```

以下是在 Ubuntu 12.04 系统成功安装 Nginx 之后的主配置文件：

```
#Nginx服务器正常启动时会读取该配置文件，以下的值都是默认的，若需要可自行修改；
#以下是配置选项

#Nginx worker进程运行的用户以及用户组
#语法格式：user  username[groupname]
#user  nobody;

#Nginx worker 进程个数
worker_processes  1;

#error 日志设置
#语法格式：error /path/file level
#其中/path/file是一个具体文件；level是日志的输出级别，其取值如下：
#debug info notice warn error crit alert emerg,从左至右级别增大；
#若设定一个级别后，则在输出的日志文件中只输出级别大于或等于已设定的级别；
#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#保存master进程ID的pid文件存放路径
#语法格式：pid path/file
#pid        logs/nginx.pid;

#事件类配置项
#一般有以下几种配置：
#1.是否打开                    能
```

```
#1、是否打开accept锁
#   语法格式：accept_mutex [on | off];
#2、lock文件的路径
#   语法格式：lock_file  path/file;
#3、使用accept锁后到真正建立连接之间的延迟时间
#   语法格式：accept_mutex_delay Nms;
#4、批量建立新连接
#   语法格式：multi_accept [on | off];
#5、选择事件模型
#   语法格式：use [kqueue | rtisg | epoll | /dev/poll | select | poll | eventport];
#6、每个worker进行的最大连接数
#   语法格式：worker_connections number;
events {
    worker_connections  1024;
}

#以下是http模块
http {
    include       mime.types;
    default_type  application/octet-stream;

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

    #access_log  logs/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    #keepalive_timeout  0;
    keepalive_timeout  65;

    #gzip  on;

#server块
#   每个server块就是一个虚拟主机，按照server_name来区分
    server {
#监听端口
        listen       80;
#主机名称
        server_name  localhost;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;
#location语法：location[= | ~ | ~* | ^~ | @] /uri/ {}
        #location尝试根据用户请求中的URI来匹配 /uri表达式，若匹配成功，则执行{}里面的配置来处理用户请
求
#以下是location的一般配置项
#1、以root方式设置资源路径
#   语法格式：root path;
#2、以alias方式设置资源路径
#   语法格式：alias path;
#3、访问首页
```

```
#   语法格式：index file...;
#4、根据HTTP返回码重定向页面
#   语法格式：error_page code [code...] [= | =answer-code] uri | @named_location;
#5、是否允许递归使用error_page
#   语法格式：recursive_error_pages [on | off];
#6、try_files
#   语法格式：try_files path1 [path2] uri;
        location / {
            root   html;
            index  index.html index.htm;
        }

        #error_page  404              /404.html;

        # redirect server error pages to the static page /50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }

        # proxy the PHP scripts to Apache listening on 127.0.0.1:80
        #
        #location ~ \.php$ {
        #    proxy_pass   http://127.0.0.1;
        #}

        # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
        #
        #location ~ \.php$ {
        #    root           html;
        #    fastcgi_pass   127.0.0.1:9000;
        #    fastcgi_index  index.php;
        #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
        #    include        fastcgi_params;
        #}

        # deny access to .htaccess files, if Apache's document root
        # concurs with nginx's one
        #
        #location ~ /\.ht {
        #    deny  all;
        #}
    }


    # another virtual host using mix of IP-, name-, and port-based configuration
    #
    #server {
    #    listen       8000;
    #    listen       somename:8080;
    #    server_name  somename  alias  another.alias;

    #    location / {
    #        root   html;
    #        index  index.html index.htm;
```

```
    #        index  index.html index.htm;
    #    }
    #}

    # HTTPS server
    #
    #server {
    #    listen       443 ssl;
    #    server_name  localhost;

    #    ssl_certificate      cert.pem;
    #    ssl_certificate_key  cert.key;

    #    ssl_session_cache    shared:SSL:1m;
    #    ssl_session_timeout  5m;

    #    ssl_ciphers  HIGH:!aNULL:!MD5;
    #    ssl_prefer_server_ciphers  on;

    #    location / {
    #        root   html;
    #        index  index.html index.htm;
    #    }
    #}

}
```

参考资料：

《深入理解Nginx》

《Nginx模块开发入门》

《Nginx开发从入门到精通》

Nginx 内存池管理

概述

Nginx 使用内存池对内存进行管理，内存管理的实现类似于前面文章介绍的《 STL源码剖析——空间配置器》，把内存分配归结为大内存分配和小内存分配。若申请的内存大小比同页的内存池最大值 max 还大，则是大内存分配，否则为小内存分配。

1. 大块内存的分配请求不会直接在内存池上分配内存来满足请求，而是直接向系统申请一块内存（就像直接使用 malloc 分配内存一样），然后将这块内存挂到内存池头部的 large 字段下。
2. 小块内存分配，则是从已有的内存池数据区中分配出一部分内存。

Nginx 内存管理相关文件：

1. src/os/unix/ngx_alloc.h/.c

- 内存相关的操作，封装了最基本的内存分配函数。
- 如 free / malloc / memalign / posix_memalign，分别被封装为 ngx_free，ngx_alloc / ngx_calloc, ngx_memalign
- ngx_alloc：封装malloc分配内存
- ngx_calloc：封装malloc分配内存，并初始化空间内容为0
- ngx_memalign：返回基于一个指定 alignment 的大小为 size 的内存空间，且其地址为 alignment 的整数倍，alignment 为2的幂。

1. src/core/ngx_palloc.h/.c

- 封装创建/销毁内存池，从内存池分配空间等函数。

Nginx 内存分配总流图如下：其中 size 是用户请求分配内存的大小，pool是现有内存池。

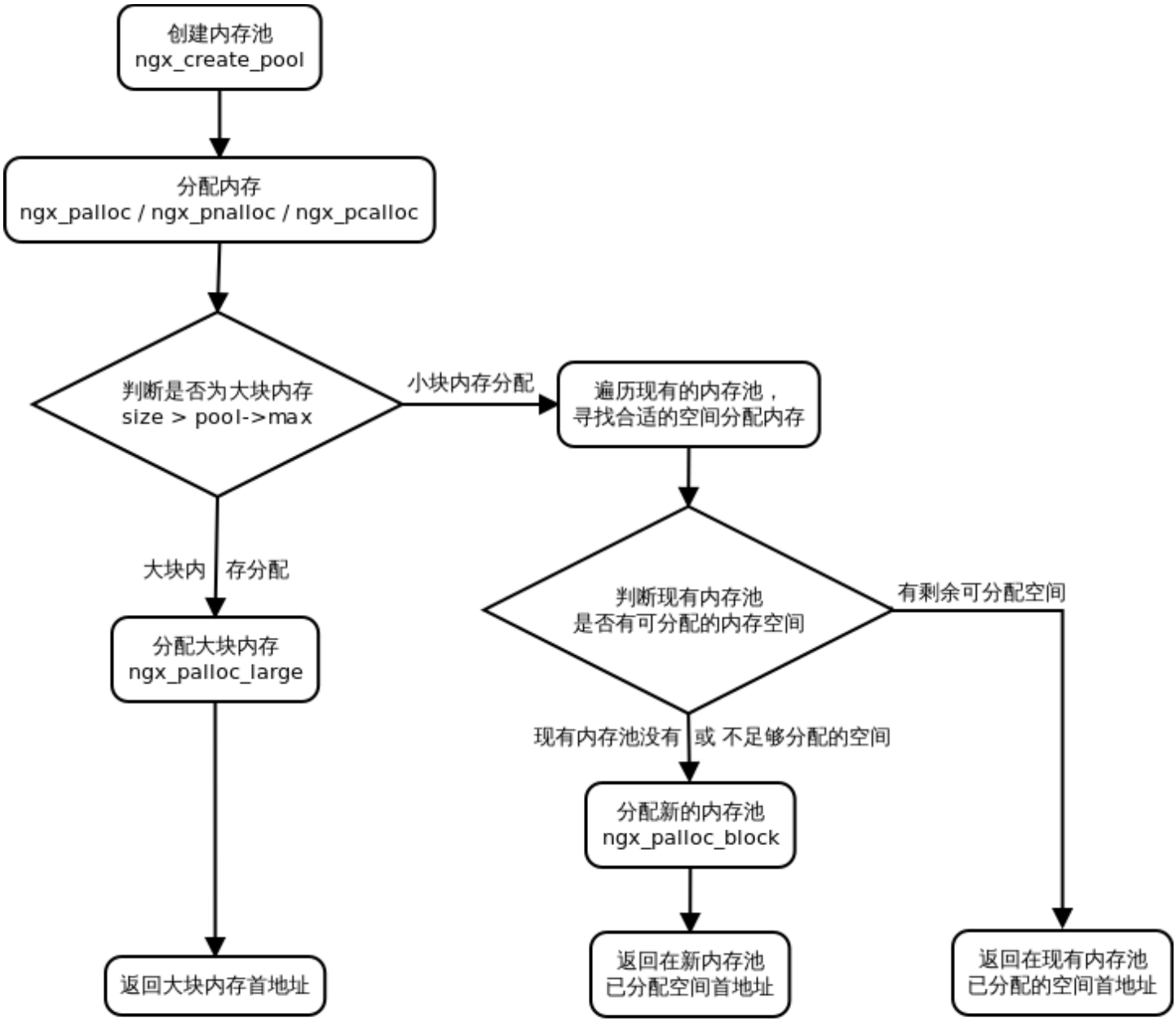


图 1 Nginx内存分配

内存池基本结构

Nginx 内存池基本结构定义如下：

```
/* 内存池结构 */
/* 文件 core/ngx_palloc.h */
typedef struct {/* 内存池数据结构模块 */
    u_char               *last; /* 当前内存分配的结束位置，即下一段可分配内存的起始位置 */
    u_char               *end;  /* 内存池的结束位置 */
    ngx_pool_t           *next; /* 指向下一个内存池 */
    ngx_uint_t           failed;/* 记录内存池内存分配失败的次数 */
} ngx_pool_data_t;  /* 维护内存池的数据块 */

struct ngx_pool_s {/* 内存池的管理模块，即内存池头部结构 */
    ngx_pool_data_t       d;    /* 内存池的数据块 */
    size_t                max;  /* 内存池数据块的最大值 */
    ngx_pool_t           *current;/* 指向当前内存池 */
    ngx_chain_t          *chain;/* 指向一个 ngx_chain_t 结构 */
    ngx_pool_large_t     *large;/* 大块内存链表，即分配空间超过 max 的内存 */
    ngx_pool_cleanup_t   *cleanup;/* 析构函数，释放内存池 */
    ngx_log_t            *log;/* 内存分配相关的日志信息 */
};
/* 文件 core/ngx_core.h */
typedef struct ngx_pool_s    ngx_pool_t;
typedef struct ngx_chain_s  ngx_chain_t;
```

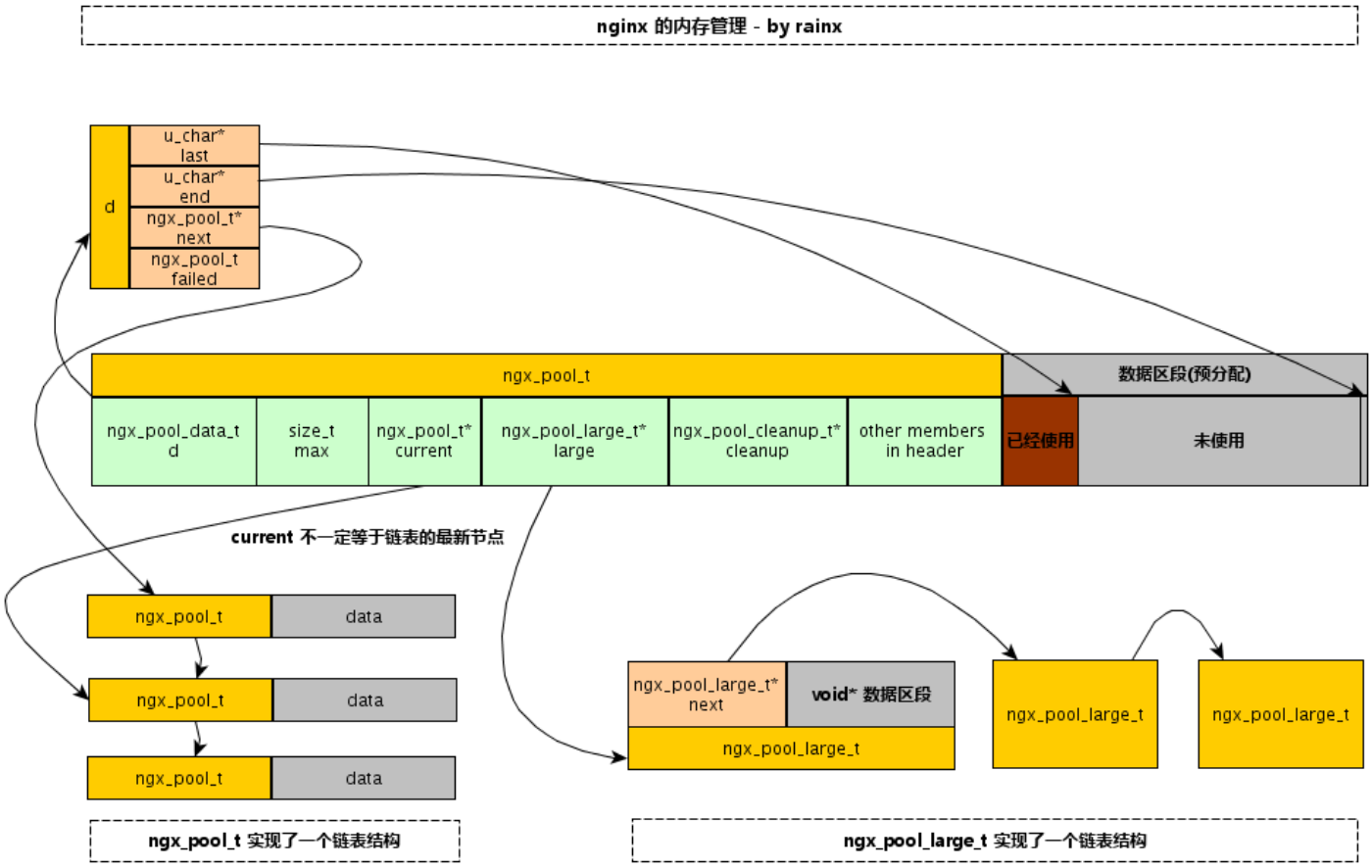
大块内存分配的数据结构如下：

```
typedef struct ngx_pool_large_s ngx_pool_large_t;

struct ngx_pool_large_s{
        ngx_pool_large_t  *next;    //指向下一块大块内存
        void    *alloc;             //指向分配的大块内存
};
```

其他数据结构如下：

```
typedef void (*ngx_pool_cleanup_pt)(void *data);    //cleanup的callback类型

typedef struct ngx_pool_cleanup_s ngx_pool_cleanup_t;

struct ngx_pool_cleanup_s{
    ngx_pool_cleanup_pt handler;
    void    *data;              //指向要清除的数据
    ngx_pool_cleanup_t *next;   //下一个cleanup callback
};


typedef struct {
    ngx_fd_t    fd;
    u_char    *name;
    ngx_log_t *log;
} ngx_pool_cleanup_file_t;
```

内存池基本机构之间的关系如下图所示：



ngx_pool_t 的逻辑结构

上面数据结构之间逻辑结构图如下：该图是采用 UML 画的，第一行黑色粗体表示对应数据结构，第二行是结构内的成员，冒号左边是变量，冒号右边是变量的类型；

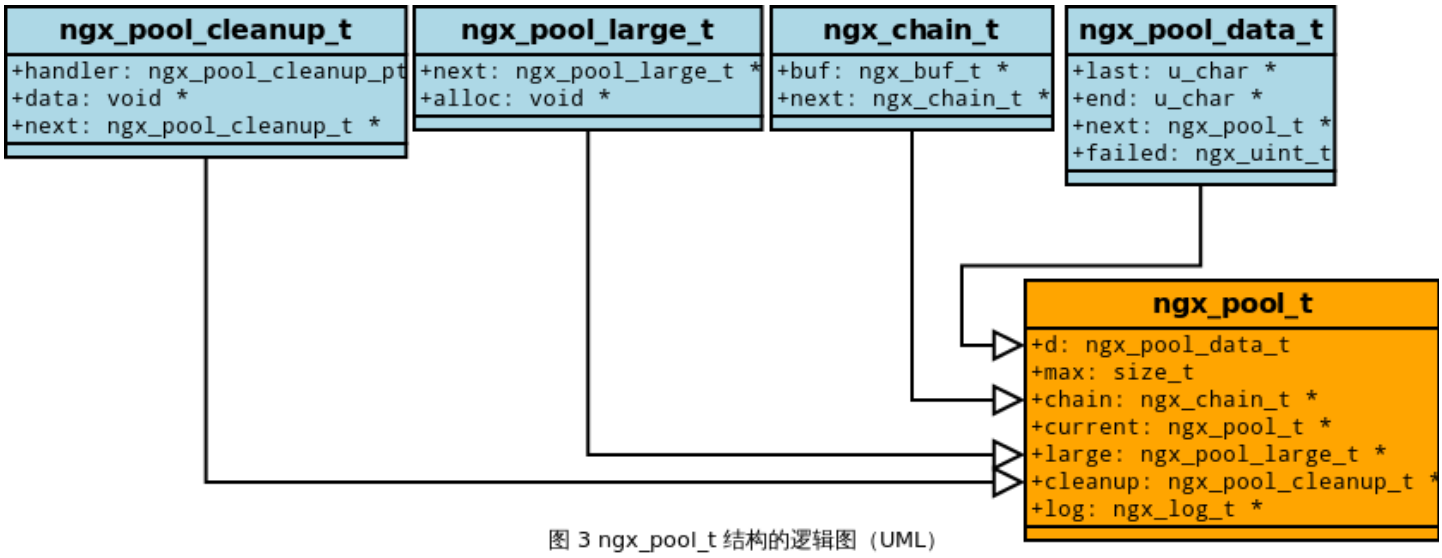


图 3 ngx_pool_t 结构的逻辑图（UML）

内存池的操作

创建内存池

```
/* 创建内存池，该函数定义于 src/core/ngx_palloc.c 文件中 */
ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ngx_pool_t  *p; /* 执行内存池头部 */

    /* 分配大小为 size 的内存 */
    /* ngx_memalign 函数实现于 src/os/unix/ngx_alloc.c 文件中 */
    p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
    if (p == NULL) {
        return NULL;
    }

    /* 以下是初始化 ngx_pool_t 结构信息 */

    p->d.last = (u_char *) p + sizeof(ngx_pool_t);
    p->d.end = (u_char *) p + size;
    p->d.next = NULL;
    p->d.failed = 0;

    size = size - sizeof(ngx_pool_t);   /* 可供分配的空间大小 */
    /* 不能超过最大的限定值 4096B */
    p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_POOL;

    p->current = p; /* 指向当前的内存池 */
    p->chain = NULL;
    p->large = NULL;
    p->cleanup = NULL;
    p->log = log;

    return p;
}
```

其中内存分配函数 ngx_memalign 定义如下：

```
void *
ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
{
    void  *p;
    int    err;

    err = posix_memalign(&p, alignment, size);
    //该函数分配以alignment为对齐的size字节的内存大小，其中p指向分配的内存块。

    if (err) {
        ngx_log_error(NGX_LOG_EMERG, log, err,
            "posix_memalign(%uz, %uz) failed", alignment, size);
        p = NULL;
    }

    ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
        "posix_memalign: %p:%uz @%uz", p, size, alignment);

    return p;
}
//函数分配以NGX_POOL_ALIGNMENT字节对齐的size字节的内存，在src/core/ngx_palloc.h文件中：
#define NGX_POOL_ALIGNMENT       16
```

销毁内存池

销毁内存池由 void ngx_destroy_pool(ngx_pool_t *pool) 函数完成。该函数将遍历内存池链表，释放所有内存，如果注册了clenup (也是一个链表结构)，亦将遍历该 cleanup 链表结构依次调用 clenup 的 handler 清理。同时，还将遍历 large 链表，释放大块内存。

```
/* 销毁内存池 */

void
ngx_destroy_pool(ngx_pool_t *pool)
{
    ngx_pool_t          *p, *n;
    ngx_pool_large_t    *l;
    ngx_pool_cleanup_t  *c;

    /* 若注册了cleanup，则遍历该链表结构，依次调用handler函数清理数据 */
    for (c = pool->cleanup; c; c = c->next) {
        if (c->handler) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                           "run cleanup: %p", c);
            c->handler(c->data);
        }
    }

    /* 遍历 large 链表，释放大块内存 */
    for (l = pool->large; l; l = l->next) {

        ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free: %p", l->alloc);
```

```
        if (l->alloc) {
            ngx_free(l->alloc); /* 释放内存 */
        }
    }

    /* 在debug模式下执行 if 和 endif 之间的代码；
     * 主要是用于log记录，跟踪函数销毁时日志信息
     */
#if (NGX_DEBUG)

    /*
     * we could allocate the pool->log from this pool
     * so we cannot use this log while free()ing the pool
     */

    for (p = pool, n = pool->d.next; /* void */; p = n, n = n->d.next) {
        ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                       "free: %p, unused: %uz", p, p->d.end - p->d.last);

        if (n == NULL) {
            break;
        }
    }

#endif

    /* 遍历所有分配的内存池，释放内存池结构 */
    for (p = pool, n = pool->d.next; /* void */; p = n, n = n->d.next) {
        ngx_free(p);

        if (n == NULL) {
            break;
        }
    }
}
```

重置内存池

重置内存池由 void ngx_reset_pool(ngx_pool_t *pool) 函数完成。该函数将释放所有 large 内存，并且将 d->last 指针重新指向 ngx_pool_t 结构之后数据区的开始位置，使内存池恢复到刚创建时的位置。由于内存池刚被创建初始化时是不包含大块内存的，所以必须释放大块内存。

```
/* 重置内存池
 * 定义于 src/core/ngx_palloc.c 文件中
 */
void
ngx_reset_pool(ngx_pool_t *pool)
{
    ngx_pool_t        *p;
    ngx_pool_large_t  *l;

    /* 遍历大块内存链表，释放大块内存 */
    for (l = pool->large; l; l = l->next) {
        if (l->alloc) {
            ngx_free(l->alloc);
        }
    }

    for (p = pool; p; p = p->d.next) {
        p->d.last = (u_char *) p + sizeof(ngx_pool_t);
        p->d.failed = 0;
    }

    pool->current = pool;
    pool->chain = NULL;
    pool->large = NULL;
}
```

内存分配

小块内存分配

小块内存分配，即请求分配空间 size 小于内存池最大内存值 max。小内存分配的接口函数如下所示：

```
void *ngx_palloc(ngx_pool_t *pool, size_t size);
void *ngx_pnalloc(ngx_pool_t *pool, size_t size);
void *ngx_pcalloc(ngx_pool_t *pool, size_t size);
void *ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment);
```

ngx_palloc 和 ngx_pnalloc 都是从内存池里分配 size 大小内存。他们的不同之处在于，palloc 取得的内存是对齐的，pnalloc 则不考虑内存对齐问题。ngx_pcalloc 是直接调用 palloc 分配内存，然后进行一次 0 初始化操作。ngx_pmemalign 将在分配 size 大小的内存并按 alignment 对齐，然后挂到 large 字段下，当做大块内存处理。

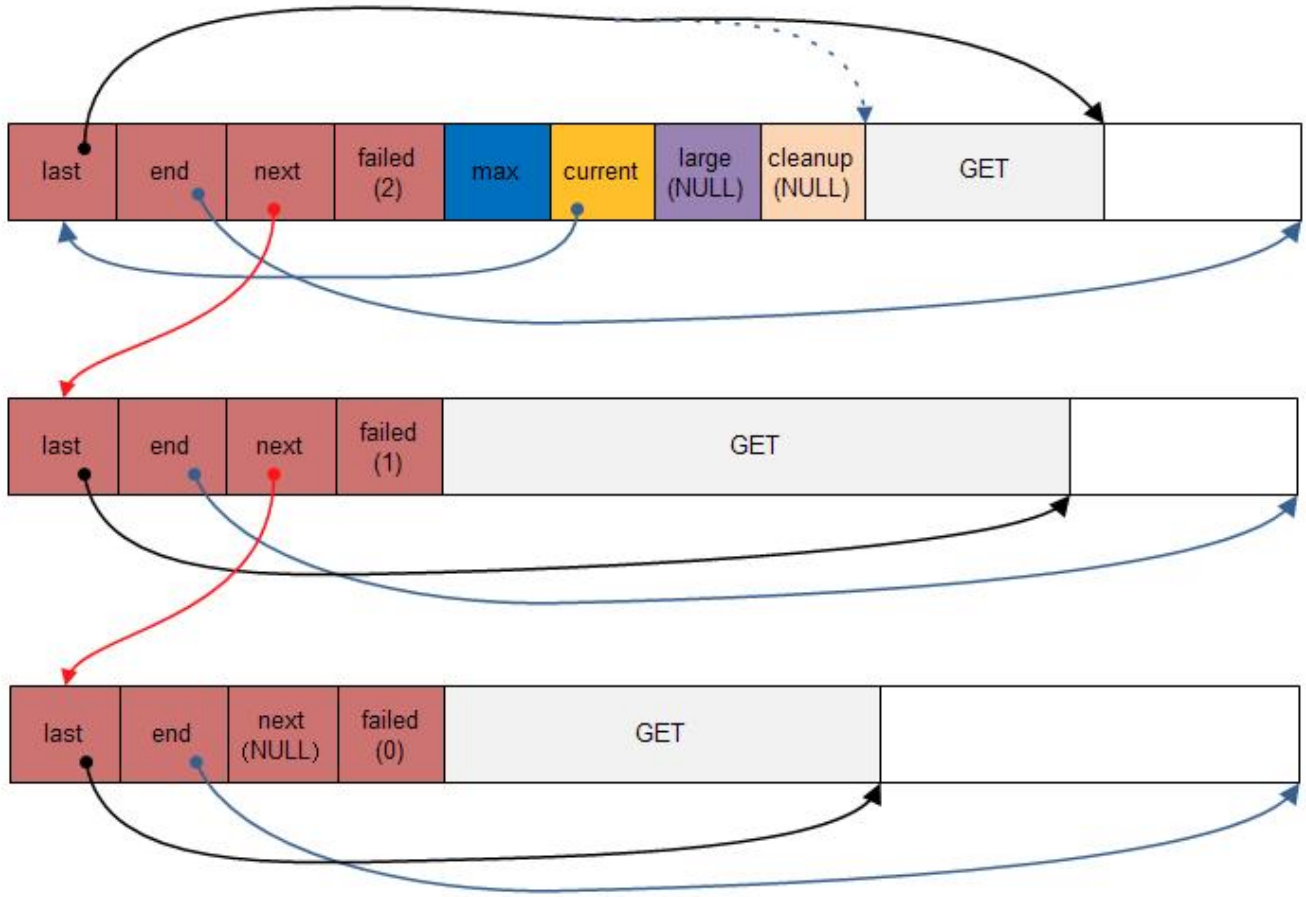
ngx_palloc的过程一般为，首先判断待分配的内存是否大于 pool->max，如果大于则使用 ngx_palloc_large 在 large 链表里分配一段内存并返回， 如果小于测尝试从链表的 pool->current 开始遍历链表，尝试找出一个可以分配的内存，当链表里的任何一个节点都无法分配内存的时候，就调用 ngx_palloc_block 生成链表里一个新的节点， 并在新的节点里分配内存并返回， 同时， 还会将pool->current 指针指向新的位置（从链表里面pool->d.failed小于等于4的节点里找出）。

```
/* 分配内存 */

void *
ngx_palloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t  *p;

    /* 若请求的内存大小size小于内存池最大内存值max，
     * 则进行小内存分配，从current开始遍历pool链表
     */
    if (size <= pool->max) {

        p = pool->current;

        do {
            /* 执行对齐操作 */
            m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);

            /* 检查现有内存池是否有足够的内存空间，
             * 若有足够的内存空间，则移动last指针位置，
             * 并返回所分配的内存地址的起始地址
             */
            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size;   /* 在该节点指向的内存块中分配size大小的内存 */

                return m;
            }

            /* 若不满足，则查找下一个内存池 */
            p = p->d.next;

        } while (p);

        /* 若遍历所有现有内存池链表都没有可用的内存空间，
         * 则分配一个新的内存池，并将该内存池连接到现有内存池链表中
         * 同时，返回分配内存的起始地址
         */
        return ngx_palloc_block(pool, size);
    }

    /* 若所请求的内存大小size大于max则调用大块内存分配函数 */
    return ngx_palloc_large(pool, size);
}

static void *
ngx_palloc_block(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    size_t       psize;
    ngx_pool_t  *p, *new, *current;

    /* 计算pool的大小，即需要分配新的block的大小 */
    psize = (size_t) (pool->d.end - (u_char *) pool);
```

```
    /* NGX_POOL_ALIGNMENT对齐操作 */
    m = ngx_memalign(NGX_POOL_ALIGNMENT, psize, pool->log);
    if (m == NULL) {
        return NULL;
    }
    /* 计算需要分配的block的大小 */
    new = (ngx_pool_t *) m;
    new->d.end = m + psize;
    new->d.next = NULL;
    new->d.failed = 0;
    /* 初始化新的内存池 */
    /* 让m指向该块内存ngx_pool_data_t结构体之后数据区起始位置 */
    m += sizeof(ngx_pool_data_t);
    /* 在数据区分配size大小的内存并设置last指针 */
    m = ngx_align_ptr(m, NGX_ALIGNMENT);
    new->d.last = m + size;

    current = pool->current;
    for (p = current; p->d.next; p = p->d.next) {
        if (p->d.failed++ > 4) {
            /* 失败4次以上移动current指针 */
            current = p->d.next;
        }
    }

    /* 将分配的block连接到现有的内存池  */
    p->d.next = new;

    /* 如果是第一次为内存池分配block，这current将指向新分配的block */
    pool->current = current ? current : new;

    return m;
}
```

```
/* 直接调用palloc函数，再进行一次0初始化操作 */
void *
ngx_pcalloc(ngx_pool_t *pool, size_t size)
{
    void *p;

    p = ngx_palloc(pool, size);
    if (p) {
        ngx_memzero(p, size);
    }

    return p;
}

/* 按照alignment对齐分配size内存，然后将其挂到large字段，当做大块内存处理 */
void *
ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment)
{
    void              *p;
    ngx_pool_large_t  *large;

    p = ngx_memalign(alignment, size, pool->log);
    if (p == NULL) {
        return NULL;
    }

    large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
    if (large == NULL) {
        ngx_free(p);
        return NULL;
    }

    large->alloc = p;
    large->next = pool->large;
    pool->large = large;

    return p;
}
```

小内存分配之后如下图所示：



上图是由3个小内存池组成的内存池模型，由于第一个内存池上剩余的内存不够分配了，于是就创建了第二个新的内存池，第三个内存池是由于前面两个内存池的剩余部分都不够分配，所以创建了第三个内存池来满足用户的需求。由图可见：所有的小内存池是由一个单向链表维护在一起的。这里还有两个字段需要关注，failed和current字段。failed表示的是当前这个内存池的剩余可用内存不能满足用户分配请求的次数，即是说：一个分配请求到来后，在这个内存池上分配不到想要的内存，那么就failed就会增加1；这个分配请求将会递交给下一个内存池去处理，如果下一个内存池也不能满足，那么它的failed也会加1，然后将请求继续往下传递，直到满足请求为止（如果没有现成的内存池来满足，会再创建一个新的内存池）。current字段会随着failed的增加而发生改变，如果current指向的内存池的failed达到了4的话，current就指向下一个内存池了。

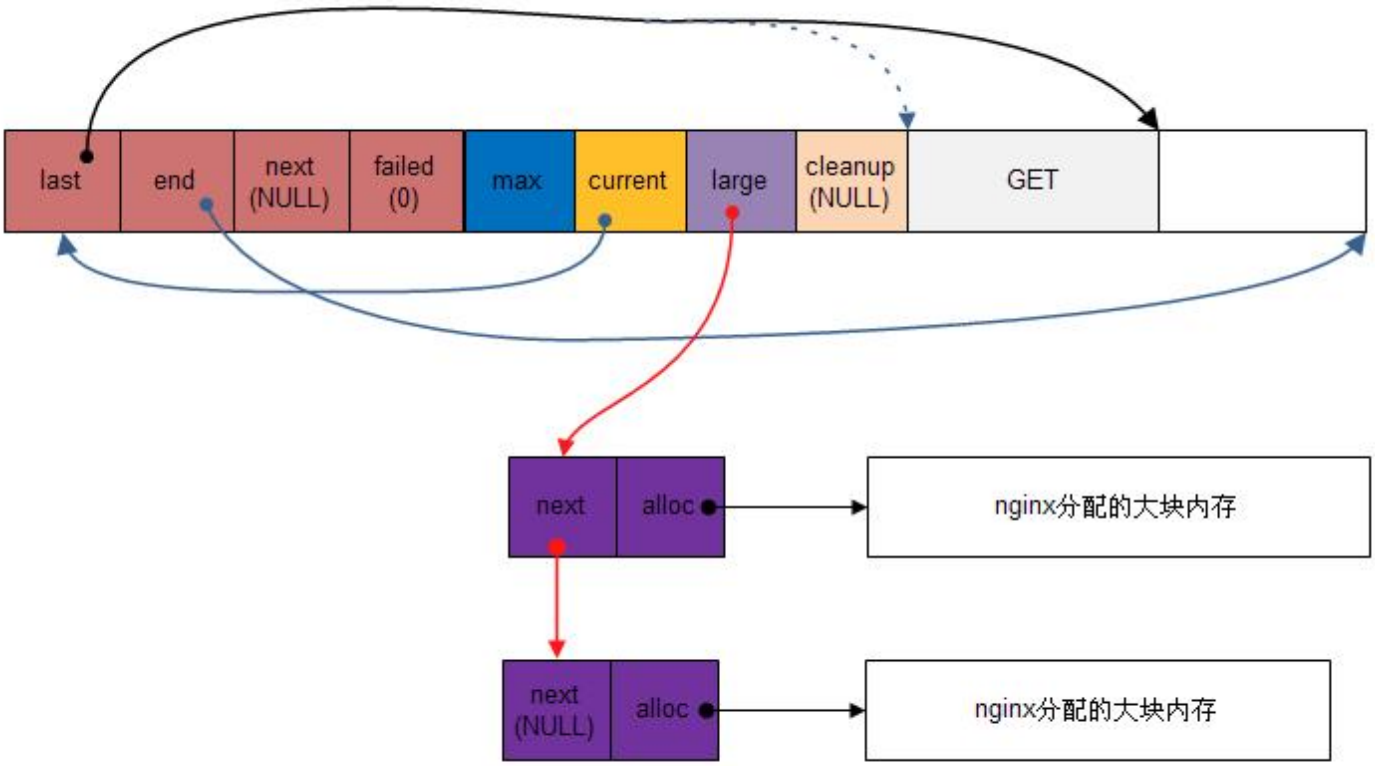
大块内存分配

```
/* 分配大块内存 */
static void *
ngx_palloc_large(ngx_pool_t *pool, size_t size)
{
    void              *p;
    ngx_uint_t         n;
    ngx_pool_large_t  *large;

    /* 分配内存 */
    p = ngx_alloc(size, pool->log);
    if (p == NULL) {
        return NULL;
    }
```

```
    n = 0;

    /* 若在该pool之前已经分配了large字段，
     * 则将所分配的大块内存挂载到内存池的large字段中
     */
    for (large = pool->large; large; large = large->next) {
        if (large->alloc == NULL) {
            large->alloc = p;
            return p;
        }

        if (n++ > 3) {
            break;
        }
    }

    /* 若在该pool之前并未分配large字段，
     * 则执行分配ngx_pool_large_t 结构体，分配large字段内存，
     * 再将大块内存挂载到pool的large字段中
     */
    large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
    if (large == NULL) {
        ngx_free(p);
        return NULL;
    }

    large->alloc = p;
    large->next = pool->large;
    pool->large = large;

    return p;
}

void *
ngx_alloc(size_t size, ngx_log_t *log)
{
    void  *p;

    p = malloc(size);

    if (p == NULL) {
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
            "malloc() %uz bytes failed", size);
    }

    ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, log, 0, "malloc: %p:%uz", p, size);
    return p;
}

/* 释放大块内存 */
ngx_int_t
ngx_pfree(ngx_pool_t *pool, void *p)
{
```

```
    ngx_pool_large_t  *l;

    for (l = pool->large; l; l = l->next) {
        if (p == l->alloc) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                           "free: %p", l->alloc);
            ngx_free(l->alloc);
            l->alloc = NULL;

            return NGX_OK;
        }
    }

    return NGX_DECLINED;
}
```

大块内存申请之后如下所示：



cleanup 资源

```
/* 注册cleanup；
 * size 是 data 字段所指向的资源的大小；
 */
ngx_pool_cleanup_t * ngx_pool_cleanup_add(ngx_pool_t *p, size_t size);

/* 对内存池进行文件清理操作,即执行handler,此时handler==ngx_pool_cleanup_file */
 void ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd);

/* 关闭data指定的文件句柄 */
 void ngx_pool_cleanup_file(void *data);
```

```
 void ngx_pool_cleanup_file(void *data);

/* 删除data指定的文件 */
 void ngx_pool_delete_file(void *data);

 /* 注册cleanup */
ngx_pool_cleanup_t *
ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)
{
    ngx_pool_cleanup_t  *c;

    c = ngx_palloc(p, sizeof(ngx_pool_cleanup_t));
    if (c == NULL) {
        return NULL;
    }

    if (size) {
        c->data = ngx_palloc(p, size);
        if (c->data == NULL) {
            return NULL;
        }

    } else {
        c->data = NULL;
    }

    c->handler = NULL;
    c->next = p->cleanup;

    p->cleanup = c;

    ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, p->log, 0, "add cleanup: %p", c);

    return c;
}

/* 清理内存池的文件 */
void
ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd)
{
    ngx_pool_cleanup_t       *c;
    ngx_pool_cleanup_file_t  *cf;

    /* 遍历cleanup结构链表，并执行handler */
    for (c = p->cleanup; c; c = c->next) {
        if (c->handler == ngx_pool_cleanup_file) {

            cf = c->data;

            if (cf->fd == fd) {
                c->handler(cf);
                c->handler = NULL;
                return;
            }
        }
```

```
    }
}

/* 关闭data指定的文件句柄 */
void
ngx_pool_cleanup_file(void *data)
{
    ngx_pool_cleanup_file_t  *c = data; /* 指向data所指向的文件句柄 */

    ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, c->log, 0, "file cleanup: fd:%d",
                   c->fd);

    /* 关闭指定文件 */
    if (ngx_close_file(c->fd) == NGX_FILE_ERROR) {
        ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
                      ngx_close_file_n " \"%s\" failed", c->name);
    }
}

/* 删除data所指向的文件 */
void
ngx_pool_delete_file(void *data)
{
    ngx_pool_cleanup_file_t  *c = data;

    ngx_err_t  err;

    ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, c->log, 0, "file cleanup: fd:%d %s",
                   c->fd, c->name);

    /* 删除data所指向的文件 */
    if (ngx_delete_file(c->name) == NGX_FILE_ERROR) {
        err = ngx_errno;

        if (err != NGX_ENOENT) {
            ngx_log_error(NGX_LOG_CRIT, c->log, err,
                          ngx_delete_file_n " \"%s\" failed", c->name);
        }
    }

    /* 关闭文件句柄 */
    if (ngx_close_file(c->fd) == NGX_FILE_ERROR) {
        ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
                      ngx_close_file_n " \"%s\" failed", c->name);
    }
}
```

参考资料：

《 Nginx源码剖析之内存池，与内存管理》

《nginx源码分析—内存池结构ngx_pool_t及内存管理》

《Nginx内存池实现源码分析 》

《Nginx源码分析-内存池》

《Ningx代码研究》

Nginx 基本数据结构

概述

在学习 Nginx 之前首先了解其基本的数据结构是非常重要的，这是入门必须了解的一个步骤。本节只是简单介绍了 Nginx 对基本数据的一种封装，包括 基本整型数据类型、字符串数据类型、缓冲区类型以及 chain 数据类型。

基本数据类型

整型数据

```
/* 基本数据结构 */

/* Nginx 简单数据类型 */
/* 在文件 src/core/ngx_config.h 定义了基本的数据映射 */

typedef intptr_t        ngx_int_t;
typedef uintptr_t       ngx_uint_t;
typedef intptr_t        ngx_flag_t;
/* 其中 intptr_t uintptr_t 定义在文件 /usr/include/stdint.h 文件中*/

/* Types for `void *' pointers.  */
#if __WORDSIZE == 64
# ifndef __intptr_t_defined
typedef long int        intptr_t;
#  define __intptr_t_defined
# endif
typedef unsigned long int    uintptr_t;
#else
# ifndef __intptr_t_defined
typedef int             intptr_t;
#  define __intptr_t_defined
# endif
typedef unsigned int        uintptr_t;

/* 因此，Nginx 的简单数据类型的操作和整型或指针类型类似 */
```

字符串类型

```
/* Nginx 字符串数据类型 */
/* Nginx 字符串类型是对 C 语言字符串类型的简单封装，
 * 其定义在 core/ngx_string.h 或 core/ngx_string.c 中
 * 定义了 ngx_str_t, ngx_keyval_t, ngx_variable_value_t
 */

/* ngx_str_t 在 u_char 的基础上增加了字符串长度的信息，即len变量 */
typedef struct {
```

```
    size_t      len;    /* 字符串的长度 */
    u_char     *data;   /* 指向字符串的第一个字符 */
} ngx_str_t;

typedef struct {
    ngx_str_t   key;
    ngx_str_t   value;
} ngx_keyval_t;

typedef struct {
    unsigned    len:28;

    unsigned    valid:1;
    unsigned    no_cacheable:1;
    unsigned    not_found:1;
    unsigned    escape:1;

    u_char     *data;
} ngx_variable_value_t;
/* Nginx 字符串的初始化使用 ngx_string 或 ngx_null_string ，这两个宏定义如下 */

#define ngx_string(str) {sizeof(str)-1, (u_char *) str}
#define ngx_null_string {0, NULL}

/* 若已经定义了 Nginx 字符串变量之后再赋值，则必须使用 ngx_str_set, ngx_str_null 宏定义*/

#define ngx_str_set(str, text)
    (str)->len = sizeof(text)-1; (str)->data = (u_char *)text

#define ngx_str_null(str)   (str)->len = 0; (str)->data = NULL

/* 例如：*/
/* 正确写法*/
ngx_str_t str1 = ngx_string("hello nginx");
ngx_str_t str2 = ngx_null_string;

/* 错误写法*/
ngx_str_t str1, str2;
str1 = ngx_string("hello nginx");   /* 编译出错 */
str2 = ngx_null_string;             /* 编译出错 */

/* 正确写法*/
ngx_str_t str1, str2;
ngx_str_set(&str1, "hello nginx");
ngx_str_null(&str2);
/* 注意：ngx_string 和 ngx_str_set 字符串参数必须是常量字符串，不能是变量字符串 */
```

内存池类型

内存池类型即是 ngx_pool_t ，有关内存池的讲解可参考前文《Nginx 内存池管理》

```
/* 内存池结构 */
/* 文件 core/ngx_palloc.h */
typedef struct {/* 内存池数据结构模块 */
    u_char               *last; /* 当前内存分配的结束位置，即下一段可分配内存的起始位置 */
    u_char               *end;  /* 内存池的结束位置 */
    ngx_pool_t           *next; /* 指向下一个内存池 */
    ngx_uint_t           failed;/* 记录内存池内存分配失败的次数 */
} ngx_pool_data_t;  /* 维护内存池的数据块 */

struct ngx_pool_s {/* 内存池的管理模块，即内存池头部结构 */
    ngx_pool_data_t       d;    /* 内存池的数据块 */
    size_t                max;  /* 内存池数据块的最大值 */
    ngx_pool_t           *current;/* 指向当前内存池 */
    ngx_chain_t          *chain;/* 指向一个 ngx_chain_t 结构 */
    ngx_pool_large_t     *large;/* 大块内存链表，即分配空间超过 max 的内存 */
    ngx_pool_cleanup_t   *cleanup;/* 析构函数，释放内存池 */
    ngx_log_t            *log;/* 内存分配相关的日志信息 */
};
/* 文件 core/ngx_core.h */
typedef struct ngx_pool_s   ngx_pool_t;
typedef struct ngx_chain_s  ngx_chain_t;
```

缓冲区数据类型

缓冲区 ngx_buf_t 的定义如下：

```
/* 缓冲区结构 */
typedef void *            ngx_buf_tag_t;

typedef struct ngx_buf_s  ngx_buf_t;

struct ngx_buf_s {
    u_char          *pos;   /* 缓冲区数据在内存的起始位置 */
    u_char          *last;  /* 缓冲区数据在内存的结束位置 */
    /* 这两个参数是处理文件时使用，类似于缓冲区的pos, last */
    off_t           file_pos;
    off_t           file_last;

    /* 由于实际数据可能被包含在多个缓冲区中，则缓冲区的start和end指向
     * 这块内存的开始地址和结束地址，
     * 而pos和last是指向本缓冲区实际包含的数据的开始和结尾
     */
    u_char          *start;         /* start of buffer */
    u_char          *end;           /* end of buffer */
    ngx_buf_tag_t    tag;
    ngx_file_t      *file;          /* 指向buffer对应的文件对象 */
    /* 当前缓冲区的一个影子缓冲区，即当一个缓冲区复制另一个缓冲区的数据，
     * 就会发生相互指向对方的shadow指针
     */
    ngx_buf_t       *shadow;

    /* 为1时，表示该buf所包含的内容在用户创建的内存块中
```

```
     * 可以被filter处理变更
     */
    /* the buf's content could be changed */
    unsigned         temporary:1;

    /* 为1时，表示该buf所包含的内容在内存中，不能被filter处理变更 */
    /*
     * the buf's content is in a memory cache or in a read only memory
     * and must not be changed
     */
    unsigned         memory:1;

    /* 为1时，表示该buf所包含的内容在内存中，
     * 可通过mmap把文件映射到内存中，不能被filter处理变更 */
    /* the buf's content is mmap()ed and must not be changed */
    unsigned         mmap:1;

    /* 可回收，即这些buf可被释放 */
    unsigned         recycled:1;
    unsigned         in_file:1; /* 表示buf所包含的内容在文件中 */
    unsigned         flush:1;   /* 刷新缓冲区 */
    unsigned         sync:1;    /* 同步方式 */
    unsigned         last_buf:1;/* 当前待处理的是最后一块缓冲区 */
    unsigned         last_in_chain:1;/* 在当前的chain里面，该buf是最后一个，但不一定是last_buf */

    unsigned         last_shadow:1;
    unsigned         temp_file:1;

    /* STUB */ int   num;
};
```

chain 数据类型

ngx_chain_t 数据类型是与缓冲区类型 ngx_buf_t 相关的链表结构，定义如下：

```
struct ngx_chain_s {
    ngx_buf_t    *buf;  /* 指向当前缓冲区 */
    ngx_chain_t  *next; /* 指向下一个chain，形成chain链表 */
};
typedef struct {
```

链表图如下：

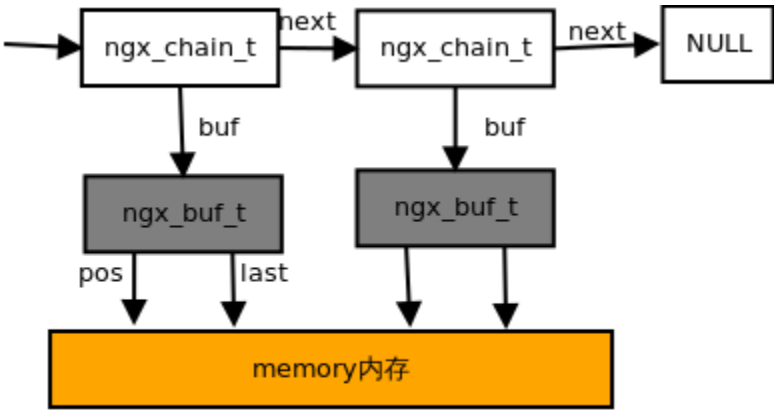


图 1 chain链表

参考资料：

《深入理解 Nginx 》

《Nginx 从入门到精通》

《Nginx 代码研究》

Nginx 数组结构 ngx_array_t

概述

本节源码来自 src/core/ngx_array.h/.c。Nginx 源码的数组类似于前面介绍的《 STL源码剖析——序列容器之 vector》，在 Nginx 数组中，内存分配是基于内存池的，并不是固定不变的，也不是需要多少内存就申请多少，若当前内存不足以存储所需元素时，按照当前数组的两倍内存大小进行申请，这样做减少内存分配的次数，提高效率。

数组数据结构

动态数组的数据结构定义如下：

```
typedef struct {
    void        *elts;  /* 指向数组数据区域的首地址 */
    ngx_uint_t   nelts; /* 数组实际数据的个数 */
    size_t       size;  /* 单个元素所占据的字节大小 */
    ngx_uint_t   nalloc;/* 数组容量 */
    ngx_pool_t  *pool;  /* 数组对象所在的内存池 */
} ngx_array_t;
```

数组结构图如下：

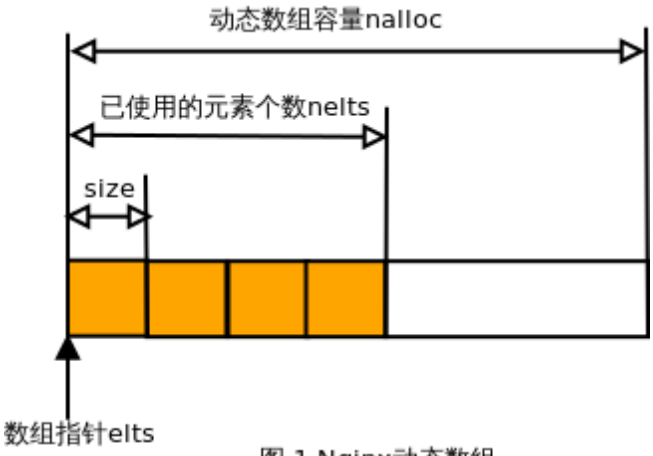


图 1 Nginx动态数组

数组的基本操作

```
/* 创建新的动态数组 */
ngx_array_t *ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size);
/* 销毁数组对象，内存被内存池回收 */
void ngx_array_destroy(ngx_array_t *a);
/* 在现有数组中增加一个新的元素 */
void *ngx_array_push(ngx_array_t *a);
/* 在现有数组中增加 n 个新的元素 */
void *ngx_array_push_n(ngx_array_t *a, ngx_uint_t n);
```

创建新的动态数组：

创建数组的操作实现如下，首先分配数组头，然后分配数组数据区，两次分配均在传入的内存池(pool指向的内存池)中进行。然后简单初始化数组头并返回数组头的起始位置。

```
/* 创建动态数组对象 */
ngx_array_t *
ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size)
{
    ngx_array_t *a;

    /* 分配动态数组头部 */
    a = ngx_palloc(p, sizeof(ngx_array_t));
    if (a == NULL) {
        return NULL;
    }

    /* 分配容量为 n 的动态数组数据区，并将其初始化 */
    if (ngx_array_init(a, p, n, size) != NGX_OK) {
        return NULL;
    }

    return a;
}

/* 当一个数组对象被分配在堆上，且调用ngx_array_destroy之后，若想重新使用，则需调用该函数 */
/* 若数组对象被分配在栈上，则需调用此函数 */
static ngx_inline ngx_int_t
ngx_array_init(ngx_array_t *array, ngx_pool_t *pool, ngx_uint_t n, size_t size)
{
    /*
     * set "array->nelts" before "array->elts", otherwise MSVC thinks
     * that "array->nelts" may be used without having been initialized
     */

    /* 初始化数组成员，注意：nelts必须比elts先初始化 */
    array->nelts = 0;
    array->size = size;
    array->nalloc = n;
    array->pool = pool;

    /* 分配数组数据域所需要的内存 */
    array->elts = ngx_palloc(pool, n * size);
    if (array->elts == NULL) {
        return NGX_ERROR;
    }

    return NGX_OK;
}
```

销毁动态数组

销毁数组的操作实现如下，包括销毁数组数据区和数组头。销毁动作实际上就是修改内存池的 last 指针，即数组的内存被内存池回收，并没有调用 free 等释放内存的操作。

```
/* 销毁数组对象，即数组所占据的内存被内存池回收 */
void
ngx_array_destroy(ngx_array_t *a)
{
    ngx_pool_t  *p;

    p = a->pool;

    /* 移动内存池的last指针，释放数组所有元素所占据的内存 */
    if ((u_char *) a->elts + a->size * a->nalloc == p->d.last) {
        p->d.last -= a->size * a->nalloc;
    }

    /* 释放数组首指针所占据的内存 */
    if ((u_char *) a + sizeof(ngx_array_t) == p->d.last) {
        p->d.last = (u_char *) a;
    }
}
```

添加元素操作

数组添加元素的操作有两个，ngx_array_push 和ngx_array_push_n，分别添加一个和多个元素。实际的添加操作并不在这两个函数中完成，只是在这两个函数中申请元素所需的内存空间，并返回指向该内存空间的首地址，在利用指针赋值的形式添加元素。

```
/* 数组增加一个元素 */
void *
ngx_array_push(ngx_array_t *a)
{
    void        *elt, *new;
    size_t       size;
    ngx_pool_t  *p;

    /* 判断数组是否已满 */
    if (a->nelts == a->nalloc) {

        /* 若现有数组所容纳的元素个数已满 */
        /* the array is full */

        /* 计算数组所有元素占据的内存大小 */
        size = a->size * a->nalloc;

        p = a->pool;

        if ((u_char *) a->elts + size == p->d.last
            && p->d.last + a->size <= p->d.end)
            /* 若当前内存池的内存空间至少可容纳一个元素大小 */
        {
```

```
            /*
             * the array allocation is the last in the pool
             * and there is space for new allocation
             */

            p->d.last += a->size;
            a->nalloc++;

        } else {
            /* 若当前内存池不足以容纳一个元素，则分配新的数组内存 */
            /* allocate a new array */

            /* 新的数组内存为当前数组大小的 2 倍 */
            new = ngx_palloc(p, 2 * size);
            if (new == NULL) {
                return NULL;
            }

            /* 首先把现有数组的所有元素复制到新的数组中 */
            ngx_memcpy(new, a->elts, size);
            a->elts = new;
            a->nalloc *= 2;
        }
    }

    elt = (u_char *) a->elts + a->size * a->nelts;
    a->nelts++;

    /* 返回指向新增加元素的指针 */
    return elt;
}

/* 数组增加 n 个元素 */
void *
ngx_array_push_n(ngx_array_t *a, ngx_uint_t n)
{
    void        *elt, *new;
    size_t       size;
    ngx_uint_t   nalloc;
    ngx_pool_t  *p;

    size = n * a->size;

    if (a->nelts + n > a->nalloc) {

        /* the array is full */

        p = a->pool;

        if ((u_char *) a->elts + a->size * a->nalloc == p->d.last
            && p->d.last + size <= p->d.end)
        {
            /*
             * the array allocation is the last in the pool
             * and there is space for new allocation
```

```
            and there is space for new allocation
             */

            p->d.last += size;
            a->nalloc += n;

        } else {
            /* allocate a new array */

            nalloc = 2 * ((n >= a->nalloc) ? n : a->nalloc);

            new = ngx_palloc(p, nalloc * a->size);
            if (new == NULL) {
                return NULL;
            }

            ngx_memcpy(new, a->elts, a->nelts * a->size);
            a->elts = new;
            a->nalloc = nalloc;
        }
    }

    elt = (u_char *) a->elts + a->size * a->nelts;
    a->nelts += n;
```

测试程序：

```
#include "ngx_config.h"
#include <stdio.h>
#include "ngx_conf_file.h"
#include "nginx.h"
#include "ngx_core.h"
#include "ngx_string.h"
#include "ngx_palloc.h"
#include "ngx_array.h"

volatile ngx_cycle_t  *ngx_cycle;

void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
            const char *fmt, ...)
{
}

void dump_array(ngx_array_t* a)
{
    if (a)
    {
        printf("array = 0x%x\n", a);
        printf("  .elts = 0x%x\n", a->elts);
        printf("  .nelts = %d\n", a->nelts);
        printf("  .size = %d\n", a->size);
        printf("  .nalloc = %d\n", a->nalloc);
        printf("  .pool = 0x%x\n", a->pool);
```

```
        printf("elements: ");
        int *ptr = (int*)(a->elts);
        for (; ptr < (int*)(a->elts + a->nalloc * a->size); )
        {
            printf("%d  ", *ptr++);
        }
        printf("\n");
    }
}

int main()
{
    ngx_pool_t *pool;
    int i;

    printf("-------------------------------\n");
    printf("create a new pool:\n");
    printf("-------------------------------\n");
    pool = ngx_create_pool(1024, NULL);

    printf("-------------------------------\n");
    printf("alloc an array from the pool:\n");
    printf("-------------------------------\n");
    ngx_array_t *a = ngx_array_create(pool, 5, sizeof(int));

    for (i = 0; i < 5; i++)
    {
        int *ptr = ngx_array_push(a);
        *ptr = 2*i;
    }

    dump_array(a);

    ngx_array_destroy(a);
    ngx_destroy_pool(pool);
    return 0;
}
```

输出结果：

```
$ ./test
-------------------------------
create a new pool:
-------------------------------
-------------------------------
alloc an array from the pool:
-------------------------------
array = 0x9fe2048
  .elts = 0x9fe205c
  .nelts = 5
  .size = 4
  .nalloc = 5
  .pool = 0x9fe2020
elements: 0  2  4  6  8
```

参考资料：

《深入理解 Nginx》

《Nginx源码分析—数组结构ngx_array_t》

Nginx 链表结构 ngx_list_t

链表结构

ngx_list_t 是 Nginx 封装的链表容器，链表容器内存分配是基于内存池进行的，操作方便，效率高。Nginx 链表容器和普通链表类似，均有链表表头和链表节点，通过节点指针组成链表。其结构定义如下：

```
/* 链表结构 */
typedef struct ngx_list_part_s  ngx_list_part_t;

/* 链表中的节点结构 */
struct ngx_list_part_s {
    void             *elts; /* 指向该节点数据区的首地址 */
    ngx_uint_t        nelts;/* 该节点数据区实际存放的元素个数 */
    ngx_list_part_t  *next; /* 指向链表的下一个节点 */
};

/* 链表表头结构 */
typedef struct {
    ngx_list_part_t  *last; /* 指向链表中最后一个节点 */
    ngx_list_part_t   part; /* 链表中表头包含的第一个节点 */
    size_t            size; /* 元素的字节大小 */
    ngx_uint_t        nalloc;/* 链表中每个节点所能容纳元素的个数 */
    ngx_pool_t       *pool; /* 该链表节点空间的内存池对象 */
} ngx_list_t;
```

链表数据结构如下图所示：

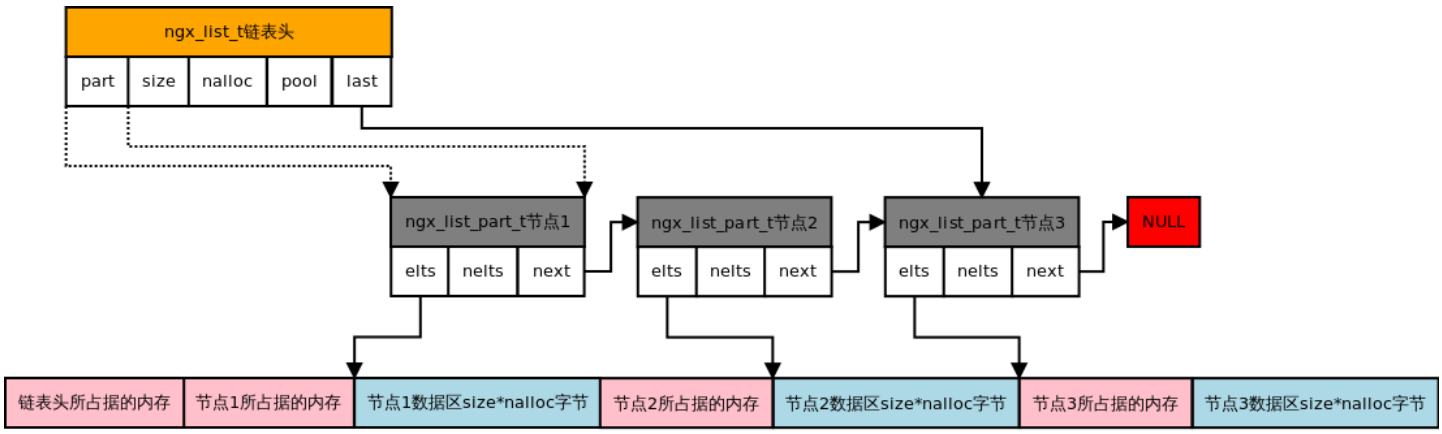


图 1 Nginx链表结构

链表操作

Nginx 链表的操作只有两个：创建链表 和 添加元素。由于链表的内存分配是基于内存池，所有内存的销毁由内存池进行，即链表没有销毁操作。

创建链表

创建新的链表时，首先分配链表表头，再对该链表进行初始化，在初始化过程中分配头节点数据区内存。

```
/* 创建链表 */
ngx_list_t *
ngx_list_create(ngx_pool_t *pool, ngx_uint_t n, size_t size)
{
    ngx_list_t  *list;

    /* 分配链表表头的内存 */
    list = ngx_palloc(pool, sizeof(ngx_list_t));
    if (list == NULL) {
        return NULL;
    }

    /* 初始化链表 */
    if (ngx_list_init(list, pool, n, size) != NGX_OK) {
        return NULL;
    }

    return list;
}

/* 初始化链表 */
static ngx_inline ngx_int_t
ngx_list_init(ngx_list_t *list, ngx_pool_t *pool, ngx_uint_t n, size_t size)
{
    /* 分配节点数据区内存，并返回该节点数据区的首地址 */
    list->part.elts = ngx_palloc(pool, n * size);
    if (list->part.elts == NULL) {
        return NGX_ERROR;
    }

    /* 初始化节点成员 */
    list->part.nelts = 0;
    list->part.next = NULL;
    list->last = &list->part;
    list->size = size;
    list->nalloc = n;
    list->pool = pool;

    return NGX_OK;
}
```

添加元素

添加元素到链表时，都是从最后一个节点开始，首先判断最后一个节点的数据区是否由内存存放新增加的元素，若足以存储该新元素，则返回存储新元素内存的位置，若没有足够的内存存储新增加的元素，则分配一个新的节点，再把该新的节点连接到现有链表中，并返回存储新元素内存的位置。注意：添加的元素可以是整数，也可以是一个结构。

```
/* 添加一个元素 */
void *
ngx_list_push(ngx_list_t *l)
{
    void             *elt;
    ngx_list_part_t  *last;

    /* last节点指针指向链表最后一个节点 */
    last = l->last;

    /* 若最后一个节点的数据区已满 */
    if (last->nelts == l->nalloc) {

        /* the last part is full, allocate a new list part */

        /* 则分配一个新的节点 */
        last = ngx_palloc(l->pool, sizeof(ngx_list_part_t));
        if (last == NULL) {
            return NULL;
        }

        /* 分配新节点数据区内存，并使节点结构指向该数据区的首地址 */
        last->elts = ngx_palloc(l->pool, l->nalloc * l->size);
        if (last->elts == NULL) {
            return NULL;
        }

        /* 初始化新节点结构 */
        last->nelts = 0;
        last->next = NULL;

        /* 把新节点连接到现有链表中 */
        l->last->next = last;
        l->last = last;
    }

    /* 计算存储新元素的位置 */
    elt = (char *) last->elts + l->size * last->nelts;
    last->nelts++;  /* 实际存放元素加1 */

    /* 返回新元素所在位置 */
    return elt;
}
```

测试程序：

```
#include "ngx_config.h"
#include <stdio.h>
#include "ngx_conf_file.h"
#include "nginx.h"
#include "ngx_core.h"
#include "ngx_string.h"
```

```
#include "ngx_palloc.h"
#include "ngx_list.h"

volatile ngx_cycle_t  *ngx_cycle;

void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
            const char *fmt, ...)
{
}
void dump_list_part(ngx_list_t* list, ngx_list_part_t* part)
{
    int *ptr = (int*)(part->elts);
    int loop = 0;

    printf("  .part = 0x%x\n", &(list->part));
    printf("    .elts = 0x%x  ", part->elts);
    printf("(");
    for (; loop < list->nalloc - 1; loop++)
    {
        printf("%d, ", ptr[loop]);
    }
    printf("%d)\n", ptr[loop]);
    printf("    .nelts = %d\n", part->nelts);
    printf("    .next = 0x%x", part->next);
    if (part->next)
        printf(" -->\n");
    printf(" \n");
}
void dump_list(ngx_list_t* list)
{
    if (list)
    {
        printf("list = 0x%x\n", list);
        printf("  .last = 0x%x\n", list->last);
        printf("  .part = %d\n", &(list->part));
        printf("  .size = %d\n", list->size);
        printf("  .nalloc = %d\n", list->nalloc);
        printf("  .pool = 0x%x\n", list->pool);

        printf("elements: \n");
        ngx_list_part_t *part = &(list->part);
        while(part)
        {
            dump_list_part(list, part);
            part = part->next;
        }
        printf("\n");
    }
}

int main()
{
    ngx_pool_t *pool;
    int i;
```

```
    printf("-------------------------------\n");
    printf("create a new pool:\n");
    printf("-------------------------------\n");
    pool = ngx_create_pool(1024, NULL);

    printf("-------------------------------\n");
    printf("alloc an list from the pool:\n");
    printf("-------------------------------\n");
    ngx_list_t *list = ngx_list_create(pool, 5, sizeof(int));

    if(NULL == list)
    {
        return -1;
    }
    for (i = 0; i < 5; i++)
    {
        int *ptr = ngx_list_push(list);
        *ptr = 2*i;
    }

    dump_list(list);

    ngx_destroy_pool(pool);
    return 0;
}
```

输出结果：

```
$ ./list_test
-------------------------------
create a new pool:
-------------------------------
-------------------------------
alloc an list from the pool:
-------------------------------
list = 0x98ce048
  .last = 0x98ce04c
  .part = 160227404
  .size = 4
  .nalloc = 5
  .pool = 0x98ce020
elements:
  .part = 0x98ce04c
    .elts = 0x98ce064  (0, 2, 4, 6, 8)
    .nelts = 5
    .next = 0x0
```

参考资料：

《深入理解 Nginx 》

《nginx源码分析—链表结构ngx_list_t》

Nginx 队列双向链表结构 ngx_queue_t

队列链表结构

队列双向循环链表实现文件：文件：src/core/ngx_queue.h/.c。在 Nginx 的队列实现中，实质就是具有头节点的双向循环链表，这里的双向链表中的节点是没有数据区的，只有两个指向节点的指针。需注意的是队列链表的内存分配不是直接从内存池分配的，即没有进行内存池管理，而是需要我们自己管理内存，所有我们可以指定它在内存池管理或者直接在堆里面进行管理，最好使用内存池进行管理。节点结构定义如下：

```
/* 队列结构，其实质是具有有头节点的双向循环链表 */
typedef struct ngx_queue_s  ngx_queue_t;

/* 队列中每个节点结构，只有两个指针，并没有数据区 */
struct ngx_queue_s {
    ngx_queue_t  *prev;
    ngx_queue_t  *next;
};
```

队列链表操作

其基本操作如下：

```
/* h 为链表结构体 ngx_queue_t 的指针；初始化双链表 */
ngx_queue_int(h)

/* h 为链表容器结构体 ngx_queue_t 的指针； 判断链表是否为空 */
ngx_queue_empty(h)

/* h 为链表容器结构体 ngx_queue_t 的指针，x 为插入元素结构体中 ngx_queue_t 成员的指针；将 x 插入到链表头部 */
ngx_queue_insert_head(h, x)

/* h 为链表容器结构体 ngx_queue_t 的指针，x 为插入元素结构体中 ngx_queue_t 成员的指针。将 x 插入到链表尾部 */
ngx_queue_insert_tail(h, x)

/* h 为链表容器结构体 ngx_queue_t 的指针。返回链表容器 h 中的第一个元素的 ngx_queue_t 结构体指针 */
ngx_queue_head(h)

/* h 为链表容器结构体 ngx_queue_t 的指针。返回链表容器 h 中的最后一个元素的 ngx_queue_t 结构体指针 */
ngx_queue_last(h)

/* h 为链表容器结构体 ngx_queue_t 的指针。返回链表结构体的指针 */
ngx_queue_sentinel(h)
```

```
/* x 为链表容器结构体 ngx_queue_t 的指针。从容器中移除 x 元素 */
ngx_queue_remove(x)

/* h 为链表容器结构体 ngx_queue_t 的指针。该函数用于拆分链表，
 * h 是链表容器，而 q 是链表 h 中的一个元素。
 * 将链表 h 以元素 q 为界拆分成两个链表 h 和 n
 */
ngx_queue_split(h, q, n)

/* h 为链表容器结构体 ngx_queue_t 的指针， n为另一个链表容器结构体 ngx_queue_t 的指针
 * 合并链表，将 n 链表添加到 h 链表的末尾
 */
ngx_queue_add(h, n)

/* h 为链表容器结构体 ngx_queue_t 的指针。返回链表中心元素，即第 N/2 + 1 个 */
ngx_queue_middle(h)

/* h 为链表容器结构体 ngx_queue_t 的指针，cmpfunc 是比较回调函数。使用插入排序对链表进行排序 */
ngx_queue_sort(h, cmpfunc)

/* q 为链表中某一个元素结构体的 ngx_queue_t 成员的指针。返回 q 元素的下一个元素。*/
ngx_queue_next(q)

/* q 为链表中某一个元素结构体的 ngx_queue_t 成员的指针。返回 q 元素的上一个元素。*/
ngx_queue_prev(q)

/* q 为链表中某一个元素结构体的 ngx_queue_t 成员的指针，type 是链表元素的结构体类型名称，
 * link 是上面这个结构体中 ngx_queue_t 类型的成员名字。返回 q 元素所属结构体的地址
 */
ngx_queue_data(q, type, link)

/* q 为链表中某一个元素结构体的 ngx_queue_t 成员的指针，x 为插入元素结构体中 ngx_queue_t 成员的指
针 */
ngx_queue_insert_after(q, x)
```

下面是队列链表操作源码的实现：

初始化链表

```
/* 初始化队列，即节点指针都指向自己，表示为空队列  */
#define ngx_queue_init(q)                                     \
    (q)->prev = q;                                        \
    (q)->next = q

/* 判断队列是否为空 */
#define ngx_queue_empty(h)                                        \
    (h == (h)->prev)
```

获取指定的队列链表中的节点

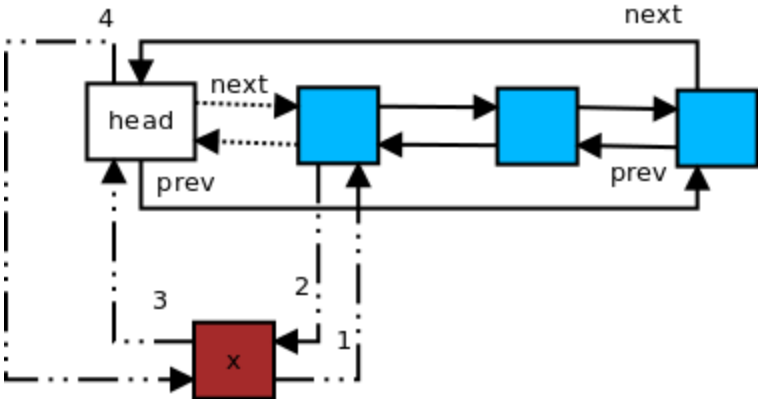
```
/* 获取队列头节点 */
#define ngx_queue_head(h)                                    \
    (h)->next

/* 获取队列尾节点 */
#define ngx_queue_last(h)                                    \
    (h)->prev

#define ngx_queue_sentinel(h)                                \
    (h)

/* 获取队列指定节点的下一个节点 */
#define ngx_queue_next(q)                                    \
    (q)->next

/* 获取队列指定节点的前一个节点 */
#define ngx_queue_prev(q)                                    \
    (q)->prev
```

插入节点

在头节点之后插入新节点：

```
/* 在队列头节点的下一节点插入新节点，其中h为头节点，x为新节点 */
#define ngx_queue_insert_head(h, x)                          \
    (x)->next = (h)->next;                                   \
    (x)->next->prev = x;                                     \
    (x)->prev = h;                                           \
    (h)->next = x
```

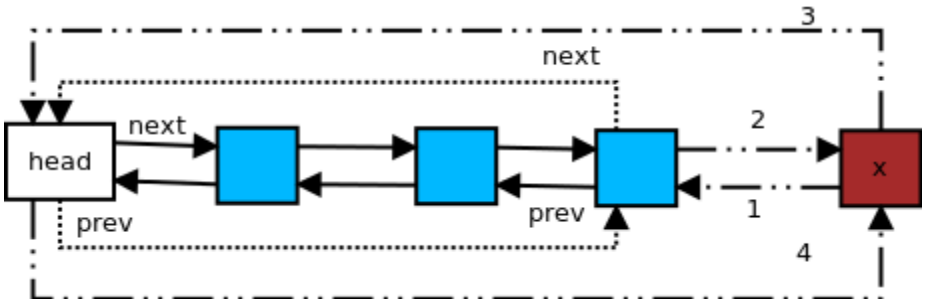
插入节点比较简单，只是修改指针的指向即可。下图是插入节点的过程：注意：虚线表示断开连接，实线表示原始连接，破折线表示重新连接，图中的数字与源码步骤相对应。



在尾节点之后插入节点

```
/* 在队列尾节点之后插入新节点，其中h为尾节点，x为新节点 */
#define ngx_queue_insert_tail(h, x)                                          \
    (x)->prev = (h)->prev;                                                    \
    (x)->prev->next = x;                                                      \
    (x)->next = h;                                                           \
    (h)->prev = x
```

下图是插入节点的过程：

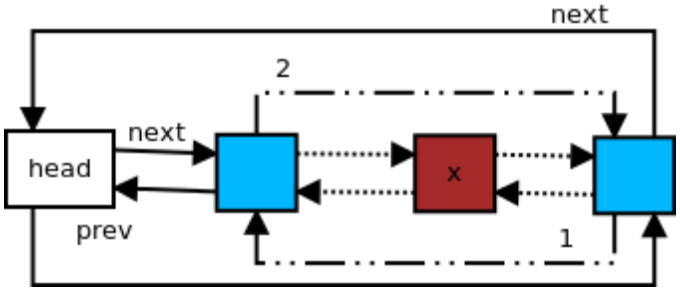


删除节点

删除指定的节点，删除节点只是修改相邻节点指针的指向，并没有实际将该节点的内存释放，内存释放必须由我们进行处理。

```
#if (NGX_DEBUG)

#define ngx_queue_remove(x)                                                  \
    (x)->next->prev = (x)->prev;                                             \
    (x)->prev->next = (x)->next;                                             \
    (x)->prev = NULL;                                                       \
    (x)->next = NULL

#else
/* 删除队列指定的节点 */
#define ngx_queue_remove(x)                                                  \
    (x)->next->prev = (x)->prev;                                             \
    (x)->prev->next = (x)->next

#endif
```

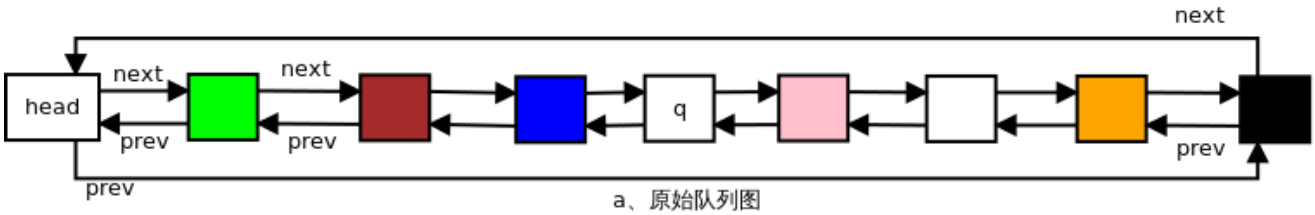
删除节点过程如下图所示：



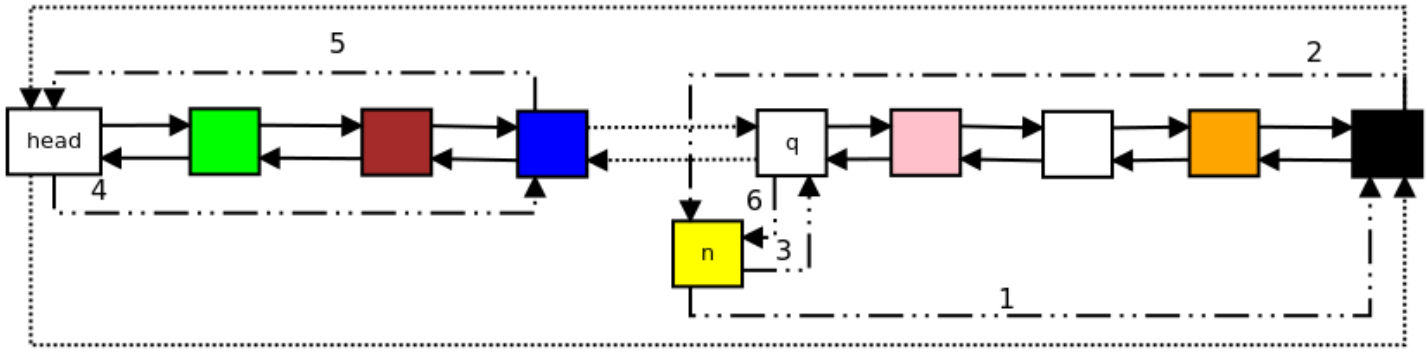
拆分链表

```
/* 拆分队列链表，使其称为两个独立的队列链表；
 * 其中h是原始队列的头节点，q是原始队列中的一个元素节点，n是新的节点，
 * 拆分后，原始队列以q为分界，头节点h到q之前的节点作为一个队列（不包括q节点），
 * 另一个队列是以n为头节点，以节点q及其之后的节点作为新的队列链表；
 */
#define ngx_queue_split(h, q, n)                                \
    (n)->prev = (h)->prev;                                      \
    (n)->prev->next = n;                                        \
    (n)->next = q;                                              \
    (h)->prev = (q)->prev;                                      \
    (h)->prev->next = h;                                        \
    (q)->prev = n;
```

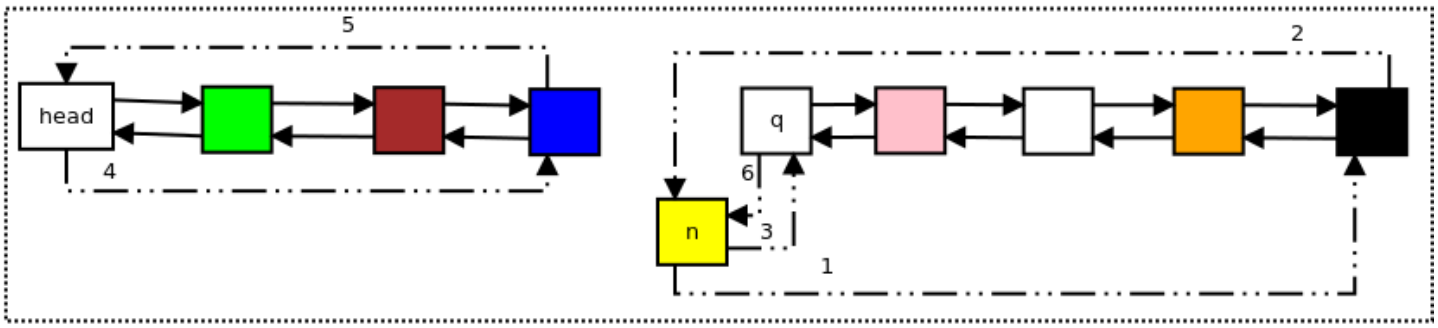
该宏有 3 个参数，h 为队列头(即链表头指针)，将该队列从 q 节点将队列(链表)拆分为两个队列(链表)，q 之后的节点组成的新队列的头节点为 n 。链表拆分过程如下图所示：



a、原始队列图



b、队列拆分过程

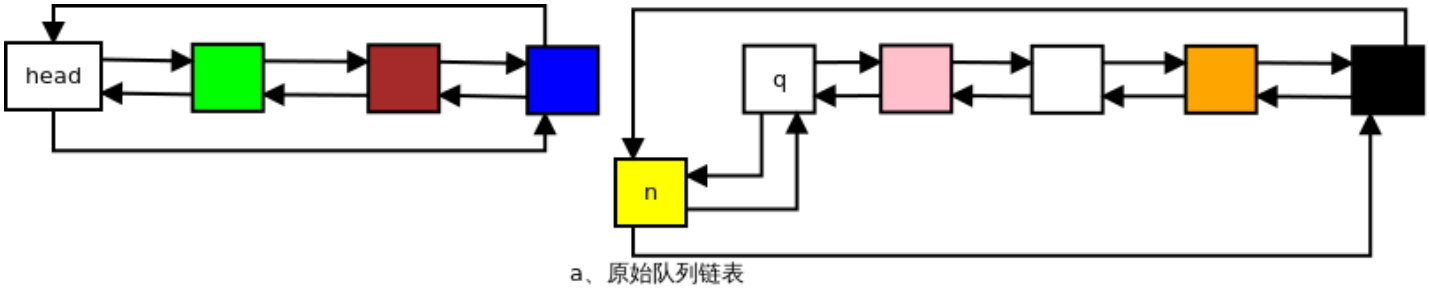


c、拆分后的队列图

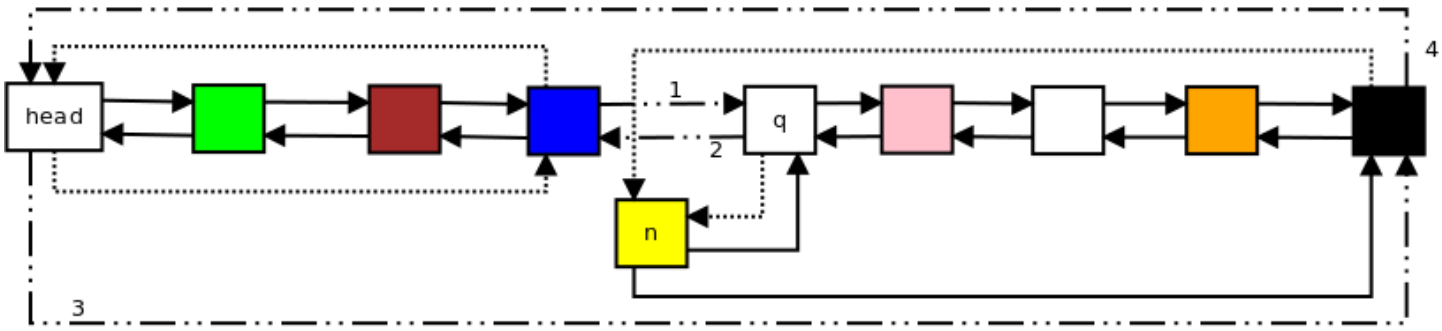
合并链表

```
/* 合并两个队列链表，把n队列链表连接到h队列链表的尾部 */
#define ngx_queue_add(h, n)                                                  \
    (h)->prev->next = (n)->next;                                             \
    (n)->next->prev = (h)->prev;                                             \
    (h)->prev = (n)->prev;                                                  \
    (h)->prev->next = h;                                                    \
    (n)->prev = (n)->next = n;/* 这是我个人增加的语句，若加上该语句，就不会出现头节点n会指向队列链表的节点 */
```

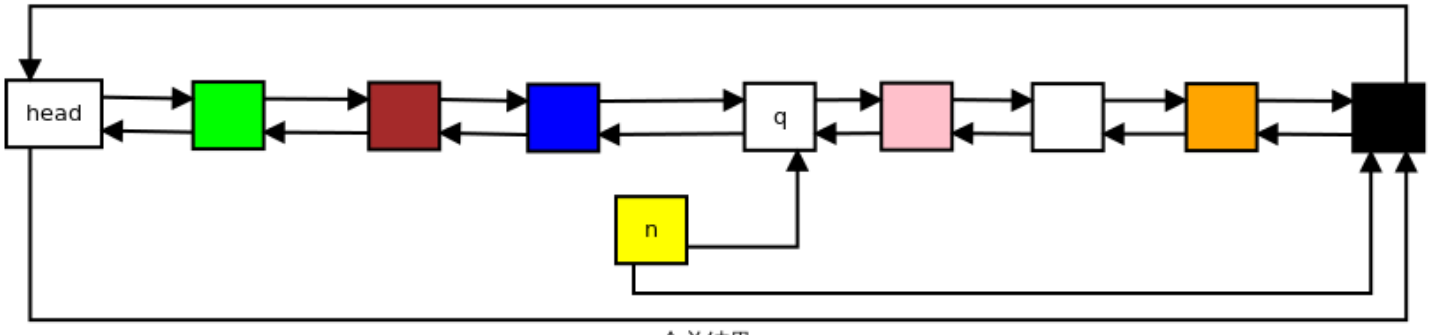
其中，h、n分别为两个队列的指针，即头节点指针，该操作将n队列链接在h队列之后。具体操作如下图所示：



a、原始队列链表



b、队列链表合并过程



c、合并结果

获取中间节点

```
/* 返回队列链表中心元素 */
ngx_queue_t *
ngx_queue_middle(ngx_queue_t *queue)
{
    ngx_queue_t  *middle, *next;

    /* 获取队列链表头节点 */
    middle = ngx_queue_head(queue);

    /* 若队列链表的头节点就是尾节点，表示该队列链表只有一个元素 */
    if (middle == ngx_queue_last(queue)) {
        return middle;
    }

    /* next作为临时指针，首先指向队列链表的头节点 */
    next = ngx_queue_head(queue);

    for ( ;; ) {
        /* 若队列链表不止一个元素，则等价于middle = middle->next */
        middle = ngx_queue_next(middle);

        next = ngx_queue_next(next);

        /* 队列链表有偶数个元素 */
        if (next == ngx_queue_last(queue)) {
            return middle;
        }

        next = ngx_queue_next(next);

        /* 队列链表有奇数个元素 */
        if (next == ngx_queue_last(queue)) {
            return middle;
        }
    }
}
```

链表排序

队列链表排序采用的是稳定的简单插入排序方法，即从第一个节点开始遍历，依次将当前节点(q)插入前面已经排好序的队列(链表)中，下面程序中，前面已经排好序的队列的尾节点为prev。操作如下：

```
/* the stable insertion sort */

/* 队列链表排序 */
void
ngx_queue_sort(ngx_queue_t *queue,
    ngx_int_t (*cmp)(const ngx_queue_t *, const ngx_queue_t *))
{
    ngx_queue_t  *q, *prev, *next;

    q = ngx_queue_head(queue);

    /* 若队列链表只有一个元素，则直接返回 */
    if (q == ngx_queue_last(queue)) {
        return;
    }

    /* 遍历整个队列链表 */
    for (q = ngx_queue_next(q); q != ngx_queue_sentinel(queue); q = next) {

        prev = ngx_queue_prev(q);
        next = ngx_queue_next(q);

        /* 首先把元素节点q独立出来 */
        ngx_queue_remove(q);

        /* 找到适合q插入的位置 */
        do {
            if (cmp(prev, q) <= 0) {
                break;
            }

            prev = ngx_queue_prev(prev);

        } while (prev != ngx_queue_sentinel(queue));

        /* 插入元素节点q */
        ngx_queue_insert_after(prev, q);
    }
}
```

获取队列中节点数据地址

由队列基本结构和以上操作可知，nginx 的队列操作只对链表指针进行简单的修改指向操作，并不负责节点数据空间的分配。因此，用户在使用nginx队列时，要自己定义数据结构并分配空间，且在其中包含一个 ngx_queue_t 的指针或者对象，当需要获取队列节点数据时，使用ngx_queue_data宏，其定义如下：

```
/* 返回q在所属结构类型的地址，type是链表元素的结构类型 */
#define ngx_queue_data(q, type, link)                                        \
    (type *) ((u_char *) q - offsetof(type, link))
/*
```

测试程序：

```
#include <stdio.h>
#include "ngx_queue.h"
#include "ngx_conf_file.h"
#include "ngx_config.h"
#include "ngx_palloc.h"
#include "nginx.h"
#include "ngx_core.h"

#define MAX     10
typedef struct Score
{
    unsigned int score;
    ngx_queue_t Que;
}ngx_queue_score;
volatile ngx_cycle_t  *ngx_cycle;

void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
            const char *fmt, ...)
{
}

ngx_int_t CMP(const ngx_queue_t *x, const ngx_queue_t *y)
{
    ngx_queue_score *xinfo = ngx_queue_data(x, ngx_queue_score, Que);
    ngx_queue_score *yinfo = ngx_queue_data(y, ngx_queue_score, Que);

    return(xinfo->score > yinfo->score);
}

void print_ngx_queue(ngx_queue_t *queue)
{
    ngx_queue_t *q = ngx_queue_head(queue);

    printf("score: ");
    for( ; q != ngx_queue_sentinel(queue); q = ngx_queue_next(q))
    {
        ngx_queue_score *ptr = ngx_queue_data(q, ngx_queue_score, Que);
        if(ptr != NULL)
            printf(" %d\t", ptr->score);
    }
    printf("\n");
}

int main()
{
```

```
    ngx_pool_t *pool;
    ngx_queue_t *queue;
    ngx_queue_score *Qscore;

    pool = ngx_create_pool(1024, NULL);

    queue = ngx_palloc(pool, sizeof(ngx_queue_t));
    ngx_queue_init(queue);

    int i;
    for(i = 1; i < MAX; i++)
    {
        Qscore = (ngx_queue_score*)ngx_palloc(pool, sizeof(ngx_queue_score));
        Qscore->score = i;
        ngx_queue_init(&Qscore->Que);

        if(i%2)
        {
            ngx_queue_insert_tail(queue, &Qscore->Que);
        }
        else
        {
            ngx_queue_insert_head(queue, &Qscore->Que);
        }
    }

    printf("Before sort: ");
    print_ngx_queue(queue);

    ngx_queue_sort(queue, CMP);

    printf("After sort: ");
    print_ngx_queue(queue);

    ngx_destroy_pool(pool);
    return 0;

}
```

输出结果：

```
$./queue_test
Before sort: score:  8    6   4   2   1   3   5   7   9
After sort: score:  1    2   3   4   5   6   7   8   9
```

总结

在 Nginx 的队列链表中，其维护的是指向链表节点的指针，并没有实际的数据区，所有对实际数据的操作需要我们自行操作，队列链表实质是双向循环链表，其操作是双向链表的基本操作。

Nginx 哈希表结构 ngx_hash_t

概述

关于哈希表的基本知识在前面的文章《 数据结构-哈希表》已作介绍。哈希表结合了数组和链表的特点，使其寻址、插入以及删除操作更加方便。哈希表的过程是将关键字通过某种哈希函数映射到相应的哈希表位置，即对应的哈希值所在哈希表的位置。但是会出现多个关键字映射相同位置的情况导致冲突问题，为了解决这种情况，哈希表使用两个可选择的方法：拉链法和 开放寻址法。

Nginx 的哈希表中使用开放寻址来解决冲突问题，为了处理字符串，Nginx 还实现了支持通配符操作的相关函数，下面对 Nginx 中哈希表的源码进行分析。源码文件：src/core/ngx_hash.h/.c。

哈希表结构

**ngx_hash_elt_t 结构 **

哈希表中关键字元素的结构 ngx_hash_elt_t，哈希表元素结构采用 键-值 形式，即 。其定义如下：

```
/* hash散列表中元素的结构，采用键值及其所以应的值<key，value>*/
typedef struct {
    void             *value;    /* 指向用户自定义的数据 */
    u_short           len;      /* 键值key的长度 */
    u_char            name[1];  /* 键值key的第一个字符，数组名name表示指向键值key首地址 */
} ngx_hash_elt_t;
```

ngx_hash_t 结构

哈希表基本结构 ngx_hash_t，其结构定义如下：

```
/* 基本hash散列表结构 */
typedef struct {
    ngx_hash_elt_t  **buckets;  /* 指向hash散列表第一个存储元素的桶 */
    ngx_uint_t        size;     /* hash散列表的桶个数 */
} ngx_hash_t;
```

元素结构图以及基本哈希结构图如下所示：

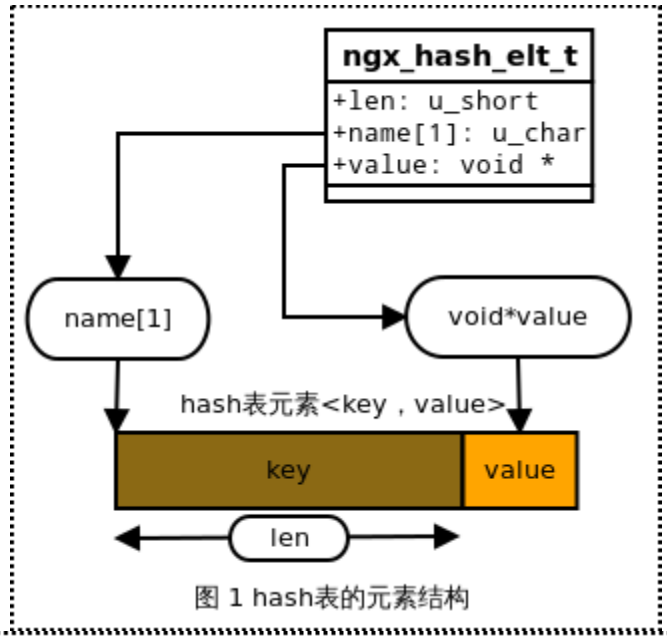


图 1 hash表的元素结构

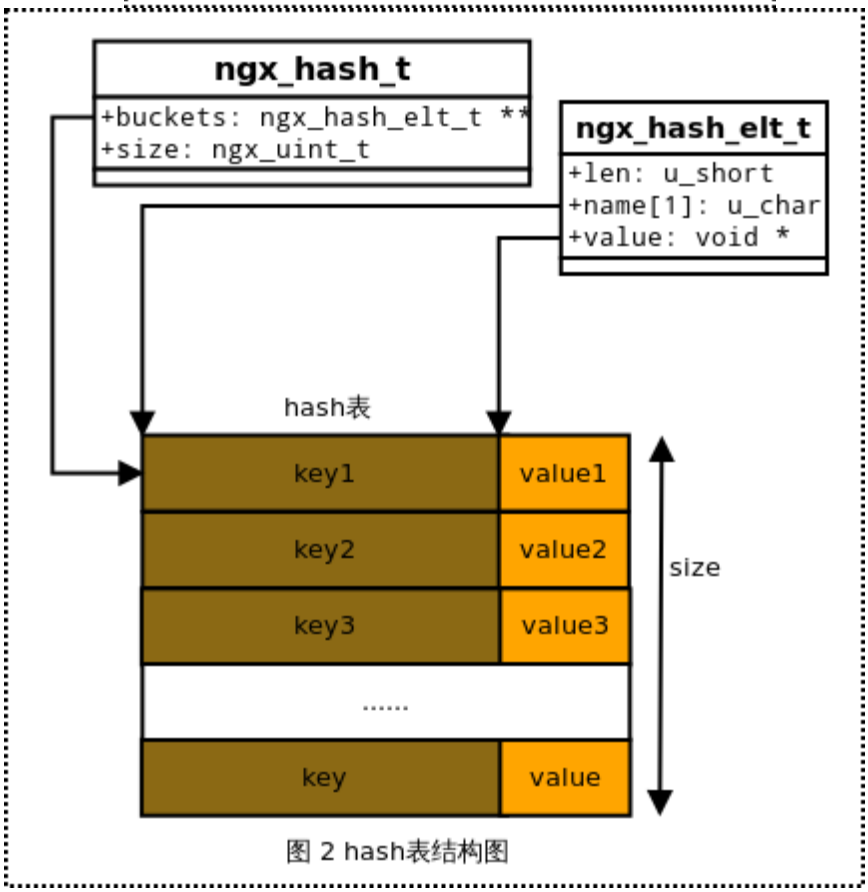


图 2 hash表结构图

**ngx_hash_init_t 初始化结构 **

哈希初始化结构 ngx_hash_init_t，Nginx 的 hash 初始化结构是 ngx_hash_init_t，用来将其相关数据封装起来作为参数传递给ngx_hash_init()，其定义如下：

```
typedef ngx_uint_t (*ngx_hash_key_pt) (u_char *data, size_t len);

/* 初始化hash结构 */
typedef struct {
    ngx_hash_t       *hash;         /* 指向待初始化的基本hash结构 */
    ngx_hash_key_pt   key;          /* hash 函数指针 */

    ngx_uint_t        max_size;     /* hash表中桶bucket的最大个数 */
    ngx_uint_t        bucket_size;  /* 每个桶bucket的存储空间 */

    char             *name;         /* hash结构的名称(仅在错误日志中使用) */
    ngx_pool_t       *pool;         /* 分配hash结构的内存池 */
    /* 分配临时数据空间的内存池，仅在初始化hash表前，用于分配一些临时数组 */
    ngx_pool_t       *temp_pool;
} ngx_hash_init_t;
```

哈希元素数据 ngx_hash_key_t，该结构也主要用来保存要 hash 的数据，即键-值对 ，在实际使用中，一般将多个键-值对保存在 ngx_hash_key_t 结构的数组中，作为参数传给ngx_hash_init()。其定义如下：

```
/* 计算待添加元素的hash元素结构 */
typedef struct {
    ngx_str_t         key;      /* 元素关键字 */
    ngx_uint_t        key_hash; /* 元素关键字key计算出的hash值 */
    void             *value;    /* 指向关键字key对应的值，组成hash表元素：键-值<key，value> */
} ngx_hash_key_t;
```

哈希操作

哈希操作包括初始化函数、查找函数；其中初始化函数是 Nginx 中哈希表比较重要的函数，由于 Nginx 的 hash 表是静态只读的，即不能在运行时动态添加新元素的，一切的结构和数据都在配置初始化的时候就已经规划完毕。

哈希函数

哈希表中使用哈希函数把用户数据映射到哈希表对应的位置中，下面是 Nginx 哈希函数的定义：

```
/* hash函数 */
#define ngx_hash(key, c)   ((ngx_uint_t) key * 31 + c)
ngx_uint_t ngx_hash_key(u_char *data, size_t len);
ngx_uint_t ngx_hash_key_lc(u_char *data, size_t len);
ngx_uint_t ngx_hash_strlow(u_char *dst, u_char *src, size_t n);
#define ngx_hash(key, c)   ((ngx_uint_t) key * 31 + c)
/* hash函数 */
ngx_uint_t
ngx_hash_key(u_char *data, size_t len)
{
    ngx_uint_t  i, key;
```

```
    key = 0;

    for (i = 0; i < len; i++) {
        /* 调用宏定义的hash 函数 */
        key = ngx_hash(key, data[i]);
    }

    return key;
}

/* 这里只是将字符串data中的所有字符转换为小写字母再进行hash值计算 */
ngx_uint_t
ngx_hash_key_lc(u_char *data, size_t len)
{
    ngx_uint_t  i, key;

    key = 0;

    for (i = 0; i < len; i++) {
        /* 把字符串转换为小写字符，并计算每个字符的hash值 */
        key = ngx_hash(key, ngx_tolower(data[i]));
    }

    return key;
}

/* 把原始关键字符串的前n个字符转换为小写字母再计算hash值
 * 注意：这里只计算前n个字符的hash值
 */
ngx_uint_t
ngx_hash_strlow(u_char *dst, u_char *src, size_t n)
{
    ngx_uint_t  key;

    key = 0;

    while (n--) {/* 把src字符串的前n个字符转换为小写字母 */
        *dst = ngx_tolower(*src);
        key = ngx_hash(key, *dst);/* 计算所转换小写字符的hash值 */
        dst++;
        src++;
    }

    return key;/* 返回整型的hash值 */
}
```

哈希初始化函数

hash 初始化由 ngx_hash_init() 函数完成，其 names 参数是 ngx_hash_key_t 结构的数组，即键-值对 数组，nelts 表示该数组元素的个数。该函数初始化的结果就是将 names 数组保存的键-值对，通过 hash 的方式将其存入相应的一个或多个 hash 桶(即代码中的 buckets)中。hash 桶里面存放的是 ngx_hash_elt_t 结构的指针(hash元素指针)，该指针指向一个基本连续的数据区。该数据区中存放的是经

hash 之后的键-值对，即 ngx_hash_elt_t 结构中的字段 。每一个这样的数据区存放的键-值对可以是一个或多个。其定义如下：

```
#define NGX_HASH_ELT_SIZE(name)                                               \
    (sizeof(void *) + ngx_align((name)->key.len + 2, sizeof(void *)))

/* 初始化hash结构函数 */
/* 参数hinit是hash表初始化结构指针；
 * name是指向待添加在hash表结构的元素数组；
 * nelts是待添加元素数组中元素的个数；
 */
ngx_int_t
ngx_hash_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names, ngx_uint_t nelts)
{
    u_char          *elts;
    size_t           len;
    u_short         *test;
    ngx_uint_t       i, n, key, size, start, bucket_size;
    ngx_hash_elt_t  *elt, **buckets;

    for (n = 0; n < nelts; n++) {
        /* 若每个桶bucket的内存空间不足以存储一个关键字元素，则出错返回
         * 这里考虑到了每个bucket桶最后的null指针所需的空间，即该语句中的sizeof(void *)，
         * 该指针可作为查找过程中的结束标记
         */
        if (hinit->bucket_size < NGX_HASH_ELT_SIZE(&names[n]) + sizeof(void *))
        {
            ngx_log_error(NGX_LOG_EMERG, hinit->pool->log, 0,
                          "could not build the %s, you should "
                          "increase %s_bucket_size: %i",
                          hinit->name, hinit->name, hinit->bucket_size);
            return NGX_ERROR;
        }
    }

    /* 临时分配sizeof(u_short)*max_size的test空间，即test数组总共有max_size个元素，即最大bucket的数量，
     * 每个元素会累计落到相应hash表位置的关键字长度，
     * 当大于256字节，即u_short所表示的字节大小，
     * 则表示bucket较少
     */
    test = ngx_alloc(hinit->max_size * sizeof(u_short), hinit->pool->log);
    if (test == NULL) {
        return NGX_ERROR;
    }

    /* 每个bucket桶实际容纳的数据大小，
     * 由于每个bucket的末尾结束标志是null，
     * 所以bucket实际容纳的数据大小必须减去一个指针所占的内存大小
     */
    bucket_size = hinit->bucket_size - sizeof(void *);

    /* 估计hash表最少bucket数量；
     * 每个关键字元素需要的内存空间是 NGX_HASH_ELT_SIZE(&names[n]) 至少需要占用两个指针的大小即2
```

```
     * 每个关键字元素需要的内存空间是 NGX_HASH_ELT_SIZE(&name[n])，至少需要占用两个指针的大小即2
*sizeof(void *)
     * 这样来估计hash表所需的最小bucket数量
     * 因为关键字元素内存越小，则每个bucket所容纳的关键字元素就越多
     * 那么hash表的bucket所需的数量就越少，但至少需要一个bucket
     */
    start = nelts / (bucket_size / (2 * sizeof(void *)));
    start = start ? start : 1;


    if (hinit->max_size > 10000 && nelts && hinit->max_size / nelts < 100) {
        start = hinit->max_size - 1000;
    }

    /* 以前面估算的最小bucket数量start，通过测试数组test估算hash表容纳 nelts个关键字元素所需的bucket
数量
     * 根据需求适当扩充bucket的数量
     */
    for (size = start; size <= hinit->max_size; size++) {

        ngx_memzero(test, size * sizeof(u_short));

        for (n = 0; n < nelts; n++) {
            if (names[n].key.data == NULL) {
                continue;
            }

            /* 根据关键字元素的hash值计算存在到测试数组test对应的位置中，即计算bucket在hash表中的编号k
ey,key取值为0～size-1 */
            key = names[n].key_hash % size;
            test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));

#if 0
            ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
                          "%ui: %ui %ui \"%V\"",
                          size, key, test[key], &names[n].key);
#endif

            /* test数组中对应的内存大于每个桶bucket最大内存，则需扩充bucket的数量
             * 即在start的基础上继续增加size的值
             */
            if (test[key] > (u_short) bucket_size) {
                goto next;
            }
        }

        /* 若size个bucket桶可以容纳name数组的所有关键字元素，则表示找到合适的bucket数量大小即为size
*/
        goto found;

    next:

        continue;
    }
```

```
    ngx_log_error(NGX_LOG_WARN, hinit->pool->log, 0,
                  "could not build optimal %s, you should increase "
                  "either %s_max_size: %i or %s_bucket_size: %i; "
                  "ignoring %s_bucket_size",
                  hinit->name, hinit->name, hinit->max_size,
                  hinit->name, hinit->bucket_size, hinit->name);

found:

    /* 到此已经找到合适的bucket数量，即为size
     * 重新初始化test数组元素，初始值为一个指针大小
     */
    for (i = 0; i < size; i++) {
        test[i] = sizeof(void *);
    }

    /* 计算每个bucket中关键字所占的空间，即每个bucket实际所容纳数据的大小，
     * 必须注意的是：test[i]中还有一个指针大小
     */
    for (n = 0; n < nelts; n++) {
        if (names[n].key.data == NULL) {
            continue;
        }

        /* 根据hash值计算出关键字放在对应的test[key]中，即test[key]的大小增加一个关键字元素的大小 */
        key = names[n].key_hash % size;
        test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));
    }

    len = 0;

    /* 调整成对齐到cacheline的大小，并记录所有元素的总长度 */
    for (i = 0; i < size; i++) {
        if (test[i] == sizeof(void *)) {
            continue;
        }

        test[i] = (u_short) (ngx_align(test[i], ngx_cacheline_size));

        len += test[i];
    }

    /*
     * 向内存池申请bucket元素所占的内存空间，
     * 注意：若前面没有申请hash表头结构，则在这里将和ngx_hash_wildcard_t一起申请
     */
    if (hinit->hash == NULL) {
        hinit->hash = ngx_pcalloc(hinit->pool, sizeof(ngx_hash_wildcard_t)
                                  + size * sizeof(ngx_hash_elt_t *));
        if (hinit->hash == NULL) {
            ngx_free(test);
            return NGX_ERROR;
        }

        /* 计算bucket的起始位置 */
```

```
        /* 计算buckets的起始位置 */
        buckets = (ngx_hash_elt_t **)
                      ((u_char *) hinit->hash + sizeof(ngx_hash_wildcard_t));

    } else {
        buckets = ngx_pcalloc(hinit->pool, size * sizeof(ngx_hash_elt_t *));
        if (buckets == NULL) {
            ngx_free(test);
            return NGX_ERROR;
        }
    }

    /* 分配elts，对齐到cacheline大小 */
    elts = ngx_palloc(hinit->pool, len + ngx_cacheline_size);
    if (elts == NULL) {
        ngx_free(test);
        return NGX_ERROR;
    }

    elts = ngx_align_ptr(elts, ngx_cacheline_size);

    /* 将buckets数组与相应的elts对应起来，即设置每个bucket对应实际数据的地址 */
    for (i = 0; i < size; i++) {
        if (test[i] == sizeof(void *)) {
            continue;
        }

        buckets[i] = (ngx_hash_elt_t *) elts;
        elts += test[i];

    }

    /* 清空test数组，以便用来累计实际数据的长度，这里不计算结尾指针的长度 */
    for (i = 0; i < size; i++) {
        test[i] = 0;
    }

    /* 依次向各个bucket中填充实际数据 */
    for (n = 0; n < nelts; n++) {
        if (names[n].key.data == NULL) {
            continue;
        }

        key = names[n].key_hash % size;
        elt = (ngx_hash_elt_t *) ((u_char *) buckets[key] + test[key]);

        elt->value = names[n].value;
        elt->len = (u_short) names[n].key.len;

        ngx_strlow(elt->name, names[n].key.data, names[n].key.len);

        /* test[key]记录当前bucket内容的填充位置，即下一次填充的起始位置 */
        test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));
    }
```

```
    /* 设置bucket结束位置的null指针 */
    for (i = 0; i < size; i++) {
        if (buckets[i] == NULL) {
            continue;
        }

        elt = (ngx_hash_elt_t *) ((u_char *) buckets[i] + test[i]);

        elt->value = NULL;
    }

    ngx_free(test);

    hinit->hash->buckets = buckets;
    hinit->hash->size = size;

#if 0

    for (i = 0; i < size; i++) {
        ngx_str_t   val;
        ngx_uint_t  key;

        elt = buckets[i];

        if (elt == NULL) {
            ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
                          "%ui: NULL", i);
            continue;
        }

        while (elt->value) {
            val.len = elt->len;
            val.data = &elt->name[0];

            key = hinit->key(val.data, val.len);

            ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
                          "%ui: %p \"%V\" %ui", i, elt, &val, key);

            elt = (ngx_hash_elt_t *) ngx_align_ptr(&elt->name[0] + elt->len,
                                                   sizeof(void *));
        }
    }

#endif

    return NGX_OK;
}
```

哈希查找函数

hash 查找操作由 ngx_hash_find() 函数完成，查找时由 key 直接计算所在的 bucket，该 bucket 中保存其所在 ngx_hash_elt_t 数据区的起始地址；然后根据长度判断并用 name 内容匹配，匹配成功，其

ngx_hash_elt_t 结构的 value 字段即是所求。其定义如下：

```
/* 查找hash元素 */
void *
ngx_hash_find(ngx_hash_t *hash, ngx_uint_t key, u_char *name, size_t len)
{
    ngx_uint_t       i;
    ngx_hash_elt_t  *elt;

#if 0
    ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "hf:\"%*s\"", len, name);
#endif

    /* 由key找到元素在hash表中所在bucket的位置 */
    elt = hash->buckets[key % hash->size];

    if (elt == NULL) {
        return NULL;
    }

    while (elt->value) {
        if (len != (size_t) elt->len) {/* 判断长度是否相等 */
            goto next;
        }

        for (i = 0; i < len; i++) {
            if (name[i] != elt->name[i]) {/* 若长度相等，则比较name的内容 */
                goto next;
            }
        }

        /* 匹配成功，则返回value字段 */
        return elt->value;

    next:

        elt = (ngx_hash_elt_t *) ngx_align_ptr(&elt->name[0] + elt->len,
                                               sizeof(void *));
        continue;
    }

    return NULL;
}
```

测试程序：

```
#include <stdio.h>
#include "ngx_config.h"
#include "ngx_conf_file.h"
#include "nginx.h"
#include "ngx_core.h"
#include "ngx_string.h"
#include "ngx_palloc.h"
```

```
#include "ngx_palloc.h"
#include "ngx_array.h"
#include "ngx_hash.h"
volatile ngx_cycle_t  *ngx_cycle;
void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err, const char *fmt, ...) { }

static ngx_str_t names[] = {ngx_string("www.baidu.com"),
                            ngx_string("www.google.com.hk"),
                            ngx_string("www.github.com")};
static char* descs[] = {"baidu: 1","google: 2", "github: 3"};

// hash table的一些基本操作
int main()
{
    ngx_uint_t          k;
    ngx_pool_t*         pool;
    ngx_hash_init_t     hash_init;
    ngx_hash_t*         hash;
    ngx_array_t*        elements;
    ngx_hash_key_t*     arr_node;
    char*               find;
    int                 i;
    ngx_cacheline_size = 32;

    pool = ngx_create_pool(1024*10, NULL);

    /* 分配hash表基本结构内存 */
    hash = (ngx_hash_t*) ngx_pcalloc(pool, sizeof(hash));
    /* 初始化hash结构 */
    hash_init.hash      = hash;                     // hash结构
    hash_init.key       = &ngx_hash_key_lc;         // hash函数
    hash_init.max_size  = 1024*10;                  // max_size
    hash_init.bucket_size = 64;
    hash_init.name      = "test_hash_error";
    hash_init.pool          = pool;
    hash_init.temp_pool     = NULL;

    /* 创建数组，把关键字压入到数组中 */

    elements = ngx_array_create(pool, 32, sizeof(ngx_hash_key_t));
    for(i = 0; i < 3; i++) {
        arr_node            = (ngx_hash_key_t*) ngx_array_push(elements);
        arr_node->key       = (names[i]);
        arr_node->key_hash  = ngx_hash_key_lc(arr_node->key.data, arr_node->key.len);
        arr_node->value     = (void*) descs[i];

        printf("key: %s , key_hash: %u\n", arr_node->key.data, arr_node->key_hash);
    }

    /* hash初始化函数 */
    if (ngx_hash_init(&hash_init, (ngx_hash_key_t*) elements->elts, elements->nelts) != NGX_OK){
        return 1;
    }

    /* 查找hash 元素 */
```

```
    k    = ngx_hash_key_lc(names[0].data, names[0].len);
    printf("%s key is %d\n", names[0].data, k);
    find = (char*)
        ngx_hash_find(hash, k, (u_char*) names[0].data, names[0].len);

    if (find) {
        printf("get desc: %s\n",(char*) find);
    }

    ngx_array_destroy(elements);
    ngx_destroy_pool(pool);

    return 0;
}
```

输出结果：

```
 ./hash_test
key: www.baidu.com , key_hash: 270263191
key: www.google.com.hk , key_hash: 2472785358
key: www.github.com , key_hash: 2818415021
www.baidu.com key is 270263191
get desc: baidu: 1
```

参考资料：

《深入理解 Nginx 》

《 nginx中hash表的设计与实现》

《Nginx 代码研究》

Nginx 红黑树结构 ngx_rbtree_t

概述

有关红黑树的基础知识在前面文章中已经做了介绍，想要更详细的了解红黑树可以参考文章《数据结构-红黑树》，在这里只是单纯对 Nginx 中红黑树源码的解析，Nginx 红黑树源码是实现跟算法导论中的讲解是一样的。

红黑树结构

```
typedef ngx_uint_t  ngx_rbtree_key_t;
typedef ngx_int_t   ngx_rbtree_key_int_t;

/* 红黑树节点结构 */
typedef struct ngx_rbtree_node_s  ngx_rbtree_node_t;

struct ngx_rbtree_node_s {
    ngx_rbtree_key_t       key;     /* 节点的键值 */
    ngx_rbtree_node_t     *left;    /* 节点的左孩子 */
    ngx_rbtree_node_t     *right;   /* 节点的右孩子 */
    ngx_rbtree_node_t     *parent;  /* 节点的父亲 */
    u_char                 color;   /* 节点的颜色 */
    u_char                 data;    /* */
};

typedef struct ngx_rbtree_s  ngx_rbtree_t;

typedef void (*ngx_rbtree_insert_pt) (ngx_rbtree_node_t *root,
    ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);

/* 红黑树结构 */
struct ngx_rbtree_s {
    ngx_rbtree_node_t     *root;    /* 指向树的根节点 */
    ngx_rbtree_node_t     *sentinel;/* 指向树的叶子节点NIL */
    ngx_rbtree_insert_pt   insert;  /* 添加元素节点的函数指针，解决具有相同键值，但不同颜色节点的冲突问
题；
                                   * 该函数指针决定新节点的行为是新增还是替换原始某个节点*/
};
```

红黑树的操作

初始化操作

```
/* 给节点着色，1表示红色，0表示黑色  */
#define ngx_rbt_red(node)               ((node)->color = 1)
#define ngx_rbt_black(node)             ((node)->color = 0)
/* 判断节点的颜色 */
#define ngx_rbt_is_red(node)            ((node)->color)
#define ngx_rbt_is_black(node)          (!ngx_rbt_is_red(node))
/* 复制某个节点的颜色 */
#define ngx_rbt_copy_color(n1, n2)      (n1->color = n2->color)

/* 节点着黑色的宏定义 */
/* a sentinel must be black */

#define ngx_rbtree_sentinel_init(node)  ngx_rbt_black(node)

/* 初始化红黑树，即为空的红黑树 */
/* tree 是指向红黑树的指针，
 * s 是红黑树的一个NIL节点，
 * i 表示函数指针，决定节点是新增还是替换
 */
#define ngx_rbtree_init(tree, s, i)                                           \
    ngx_rbtree_sentinel_init(s);                                              \
    (tree)->root = s;                                                         \
    (tree)->sentinel = s;                                                     \
    (tree)->insert = i
```

旋转操作

```
/* 左旋转操作 */
static ngx_inline void
ngx_rbtree_left_rotate(ngx_rbtree_node_t **root, ngx_rbtree_node_t *sentinel,
    ngx_rbtree_node_t *node)
{
    ngx_rbtree_node_t  *temp;

    temp = node->right;/* temp为node节点的右孩子 */
    node->right = temp->left;/* 设置node节点的右孩子为temp的左孩子 */

    if (temp->left != sentinel) {
        temp->left->parent = node;
    }

    temp->parent = node->parent;

    if (node == *root) {
        *root = temp;

    } else if (node == node->parent->left) {
        node->parent->left = temp;

    } else {
        node->parent->right = temp;
    }
```

```
    temp->left = node;
    node->parent = temp;
}

static ngx_inline void
ngx_rbtree_right_rotate(ngx_rbtree_node_t **root, ngx_rbtree_node_t *sentinel,
    ngx_rbtree_node_t *node)
{
    ngx_rbtree_node_t  *temp;

    temp = node->left;
    node->left = temp->right;

    if (temp->right != sentinel) {
        temp->right->parent = node;
    }

    temp->parent = node->parent;

    if (node == *root) {
        *root = temp;

    } else if (node == node->parent->right) {
        node->parent->right = temp;

    } else {
        node->parent->left = temp;
    }

    temp->right = node;
    node->parent = temp;
}
```

插入操作

```
/* 获取红黑树键值最小的节点 */
static ngx_inline ngx_rbtree_node_t *
ngx_rbtree_min(ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
{
    while (node->left != sentinel) {
        node = node->left;
    }

    return node;
}

/* 插入节点 */
/* 插入节点的步骤：
 * 1、首先按照二叉查找树的插入操作插入新节点；
 * 2、然后把新节点着色为红色（避免破坏红黑树性质5）；
 * 3、为维持红黑树的性质，调整红黑树的节点（着色并旋转），使其满足红黑树的性质；
 */
void
```

```
void
ngx_rbtree_insert(ngx_thread_volatile ngx_rbtree_t *tree,
    ngx_rbtree_node_t *node)
{
    ngx_rbtree_node_t  **root, *temp, *sentinel;

    /* a binary tree insert */

    root = (ngx_rbtree_node_t **) &tree->root;
    sentinel = tree->sentinel;

    /* 若红黑树为空，则比较简单，把新节点作为根节点，
     * 并初始化该节点使其满足红黑树性质
     */
    if (*root == sentinel) {
        node->parent = NULL;
        node->left = sentinel;
        node->right = sentinel;
        ngx_rbt_black(node);
        *root = node;

        return;
    }

    /* 若红黑树不为空，则按照二叉查找树的插入操作进行
     * 该操作由函数指针提供
     */
    tree->insert(*root, node, sentinel);

    /* re-balance tree */

    /* 调整红黑树，使其满足性质，
     * 其实这里只是破坏了性质4：若一个节点是红色，则孩子节点都为黑色；
     * 若破坏了性质4，则新节点 node 及其父亲节点 node->parent 都为红色；
     */
    while (node != *root && ngx_rbt_is_red(node->parent)) {

        /* 若node的父亲节点是其祖父节点的左孩子 */
        if (node->parent == node->parent->parent->left) {
            temp = node->parent->parent->right;/* temp节点为node的叔叔节点 */

            /* case1：node的叔叔节点是红色 */
            /* 此时，node的父亲及叔叔节点都为红色；
             * 解决办法：将node的父亲及叔叔节点着色为黑色，将node祖父节点着色为红色；
             * 然后沿着祖父节点向上判断是否会破会红黑树的性质；
             */
            if (ngx_rbt_is_red(temp)) {
                ngx_rbt_black(node->parent);
                ngx_rbt_black(temp);
                ngx_rbt_red(node->parent->parent);
                node = node->parent->parent;

            } else {
                /* case2：node的叔叔节点是黑色且node是父亲节点的右孩子 */
                /* 则此时，以node父亲节点进行左旋转，使case2转变为case3；
```

```
                 */
                if (node == node->parent->right) {
                    node = node->parent;
                    ngx_rbtree_left_rotate(root, sentinel, node);
                }

                /* case3：node的叔叔节点是黑色且node是父亲节点的左孩子 */
                /* 首先，将node的父亲节点着色为黑色，祖父节点着色为红色；
                 * 然后以祖父节点进行一次右旋转；
                 */
                ngx_rbt_black(node->parent);
                ngx_rbt_red(node->parent->parent);
                ngx_rbtree_right_rotate(root, sentinel, node->parent->parent);
            }

        } else {/* 若node的父亲节点是其祖父节点的右孩子 */
            /* 这里跟上面的情况是对称的，就不再进行讲解了
             */
            temp = node->parent->parent->left;

            if (ngx_rbt_is_red(temp)) {
                ngx_rbt_black(node->parent);
                ngx_rbt_black(temp);
                ngx_rbt_red(node->parent->parent);
                node = node->parent->parent;

            } else {
                if (node == node->parent->left) {
                    node = node->parent;
                    ngx_rbtree_right_rotate(root, sentinel, node);
                }

                ngx_rbt_black(node->parent);
                ngx_rbt_red(node->parent->parent);
                ngx_rbtree_left_rotate(root, sentinel, node->parent->parent);
            }
        }
    }

    /* 根节点必须为黑色 */
    ngx_rbt_black(*root);
}

/* 这里只是将节点插入到红黑树中，并没有判断是否满足红黑树的性质；
 * 类似于二叉查找树的插入操作，这个函数为红黑树插入操作的函数指针；
 */
void
ngx_rbtree_insert_value(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
    ngx_rbtree_node_t *sentinel)
{
    ngx_rbtree_node_t  **p;

    for ( ;; ) {

        /* 判断node节点键值与temp节点键值的大小，以决定node插入到temp节点的左子树还是右子树 */
```

```
        p = (node->key < temp->key) ? &temp->left : &temp->right;

        if (*p == sentinel) {
            break;
        }

        temp = *p;
    }

    /* 初始化node节点，并着色为红色 */
    *p = node;
    node->parent = temp;
    node->left = sentinel;
    node->right = sentinel;
    ngx_rbt_red(node);
}

void
ngx_rbtree_insert_timer_value(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
    ngx_rbtree_node_t *sentinel)
{
    ngx_rbtree_node_t  **p;

    for ( ;; ) {

        /*
         * Timer values
         * 1) are spread in small range, usually several minutes,
         * 2) and overflow each 49 days, if milliseconds are stored in 32 bits.
         * The comparison takes into account that overflow.
         */

        /*  node->key < temp->key */

        p = ((ngx_rbtree_key_int_t) (node->key - temp->key) < 0)
            ? &temp->left : &temp->right;

        if (*p == sentinel) {
            break;
        }

        temp = *p;
    }

    *p = node;
    node->parent = temp;
    node->left = sentinel;
    node->right = sentinel;
    ngx_rbt_red(node);
}
```

删除操作

```
/* 删除节点 */
void
ngx_rbtree_delete(ngx_thread_volatile ngx_rbtree_t *tree,
    ngx_rbtree_node_t *node)
{
    ngx_uint_t          red;
    ngx_rbtree_node_t  **root, *sentinel, *subst, *temp, *w;

    /* a binary tree delete */

    root = (ngx_rbtree_node_t **) &tree->root;
    sentinel = tree->sentinel;

    /* 下面是获取temp节点值，temp保存的节点是准备替换节点node ；
     * subst是保存要被替换的节点的后继节点；
     */

    /* case1：若node节点没有左孩子（这里包含了存在或不存在右孩子的情况）*/
    if (node->left == sentinel) {
        temp = node->right;
        subst = node;

    } else if (node->right == sentinel) {/* case2：node节点存在左孩子，但是不存在右孩子 */
        temp = node->left;
        subst = node;

    } else {/* case3：node节点既有左孩子，又有右孩子 */
        subst = ngx_rbtree_min(node->right, sentinel);/* 获取node节点的后续节点 */

        if (subst->left != sentinel) {
            temp = subst->left;
        } else {
            temp = subst->right;
        }
    }

    /* 若被替换的节点subst是根节点，则temp直接替换subst称为根节点 */
    if (subst == *root) {
        *root = temp;
        ngx_rbt_black(temp);

        /* DEBUG stuff */
        node->left = NULL;
        node->right = NULL;
        node->parent = NULL;
        node->key = 0;

        return;
    }

    /* red记录subst节点的颜色 */
    red = ngx_rbt_is_red(subst);

    /* temp节点替换subst 节点 */
    if (subst->parent == subst->parent->left) {
```

```
    if (subst == subst->parent->left) {
        subst->parent->left = temp;

    } else {
        subst->parent->right = temp;
    }

    /* 根据subst是否为node节点进行处理 */
    if (subst == node) {

        temp->parent = subst->parent;

    } else {

        if (subst->parent == node) {
            temp->parent = subst;

        } else {
            temp->parent = subst->parent;
        }

        /* 复制node节点属性 */
        subst->left = node->left;
        subst->right = node->right;
        subst->parent = node->parent;
        ngx_rbt_copy_color(subst, node);

        if (node == *root) {
            *root = subst;

        } else {
            if (node == node->parent->left) {
                node->parent->left = subst;
            } else {
                node->parent->right = subst;
            }
        }

        if (subst->left != sentinel) {
            subst->left->parent = subst;
        }

        if (subst->right != sentinel) {
            subst->right->parent = subst;
        }
    }

    /* DEBUG stuff */
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;
    node->key = 0;

    if (red) {
        return;
```

```
    }

    /* 下面开始是调整红黑树的性质 */
    /* a delete fixup */

    /* 根据temp节点进行处理 ，若temp不是根节点且为黑色 */
    while (temp != *root && ngx_rbt_is_black(temp)) {

        /* 若temp是其父亲节点的左孩子 */
        if (temp == temp->parent->left) {
            w = temp->parent->right;/* w为temp的兄弟节点 */

            /* case A：temp兄弟节点为红色 */
            /* 解决办法：
             * 1、改变w节点及temp父亲节点的颜色；
             * 2、对temp父亲节的做一次左旋转，此时，temp的兄弟节点是旋转之前w的某个子节点，该子节点颜色为黑色；
             * 3、此时，case A已经转换为case B、case C 或 case D；
             */
            if (ngx_rbt_is_red(w)) {
                ngx_rbt_black(w);
                ngx_rbt_red(temp->parent);
                ngx_rbtree_left_rotate(root, sentinel, temp->parent);
                w = temp->parent->right;
            }

            /* case B：temp的兄弟节点w是黑色，且w的两个子节点都是黑色 */
            /* 解决办法：
             * 1、改变w节点的颜色；
             * 2、把temp的父亲节点作为新的temp节点；
             */
            if (ngx_rbt_is_black(w->left) && ngx_rbt_is_black(w->right)) {
                ngx_rbt_red(w);
                temp = temp->parent;

            } else {/* case C：temp的兄弟节点是黑色，且w的左孩子是红色，右孩子是黑色 */
                /* 解决办法：
                 * 1、将改变w及其左孩子的颜色；
                 * 2、对w节点进行一次右旋转；
                 * 3、此时，temp新的兄弟节点w有着一个红色右孩子的黑色节点，转为case D；
                 */
                if (ngx_rbt_is_black(w->right)) {
                    ngx_rbt_black(w->left);
                    ngx_rbt_red(w);
                    ngx_rbtree_right_rotate(root, sentinel, w);
                    w = temp->parent->right;
                }

                /* case D：temp的兄弟节点w为黑色，且w的右孩子为红色 */
                /* 解决办法：
                 * 1、将w节点设置为temp父亲节点的颜色，temp父亲节点设置为黑色；
                 * 2、w的右孩子设置为黑色；
                 * 3、对temp的父亲节点做一次左旋转；
                 * 4、最后把根节点root设置为temp节点；*/
                ngx_rbt_copy_color(w, temp->parent);
```

```
                ngx_rbt_copy_color(w, temp->parent);
                ngx_rbt_black(temp->parent);
                ngx_rbt_black(w->right);
                ngx_rbtree_left_rotate(root, sentinel, temp->parent);
                temp = *root;
            }

        } else {/* 这里针对的是temp节点为其父亲节点的左孩子的情况 */
            w = temp->parent->left;

            if (ngx_rbt_is_red(w)) {
                ngx_rbt_black(w);
                ngx_rbt_red(temp->parent);
                ngx_rbtree_right_rotate(root, sentinel, temp->parent);
                w = temp->parent->left;
            }

            if (ngx_rbt_is_black(w->left) && ngx_rbt_is_black(w->right)) {
                ngx_rbt_red(w);
                temp = temp->parent;

            } else {
                if (ngx_rbt_is_black(w->left)) {
                    ngx_rbt_black(w->right);
                    ngx_rbt_red(w);
                    ngx_rbtree_left_rotate(root, sentinel, w);
                    w = temp->parent->left;
                }

                ngx_rbt_copy_color(w, temp->parent);
                ngx_rbt_black(temp->parent);
                ngx_rbt_black(w->left);
                ngx_rbtree_right_rotate(root, sentinel, temp->parent);
                temp = *root;
            }
        }
    }

    ngx_rbt_black(temp);
}
```

Nginx 模块开发

关注校招、实习信息

Nginx 模块概述

Nginx 模块有三种角色：

- 处理请求并产生输出的 Handler 模块；
- 处理由 Handler 产生的输出的 Filter（滤波器）模块；
- 当出现多个后台服务器时，Load-balancer （负载均衡器）模块负责选择其中一个后台服务器发送请求；

　　通常，服务器启动时，任何 Handler 模块都有可能去处理配置文件中的 location 定义。若出现多个Handler 模块被配置成需要处理某一特定的 location 时，最终只有其中一个Handler 模块是成功的。Handler 模块有三种返回方式：

1. 接收请求，并成功返回；
2. 接收请求，但是出错返回；
3. 拒绝请求，使默认的 Handler 模块处理该请求；

　　若 Handler 模块的作用是把一个请求反向代理到后台服务器，则会出现另一种类型的空间模块 —— Load-balancer。 Load-balancer 负责决定将请求发送给哪个后端服务器。Nginx 目前支持两种 Load-balancer 模块：round-robin （轮询，处理请求就像打扑克时发牌那样）和"IP hash" method（众多请求时，保证来自同一 IP 的请求被分发的同一个后端服务器）。

　　若 Handler 模块没有产生错误返回时，则会调用 Filter 模块。每个location 配置里都可以添加多个Filter 模块 ，因此响应可以被压缩和分块。Filter 模块之间的处理顺序是在编译时就已经确定的。

Filter 模块采用“CHAIN OF RESPONSIBILITY”链式的设计模式。当有请求到达时，请求依次经过这条链上的全部 Filter 模块，一个Filter 被调用并处理，接下来调用下一个Filter，直到最后一个Filter 被调用完成，Nginx 才真正完成响应流程。

总结如下，典型的处理形式如下：

```
Client sends HTTP request → Nginx chooses the appropriate handler based on the location config →
 (if applicable) load-balancer picks a backend server →
Handler does its thing and passes each output buffer to the first filter →
First filter passes the output to the second filter → second to third → third to fourth → etc.
→ Final response sent to client
```

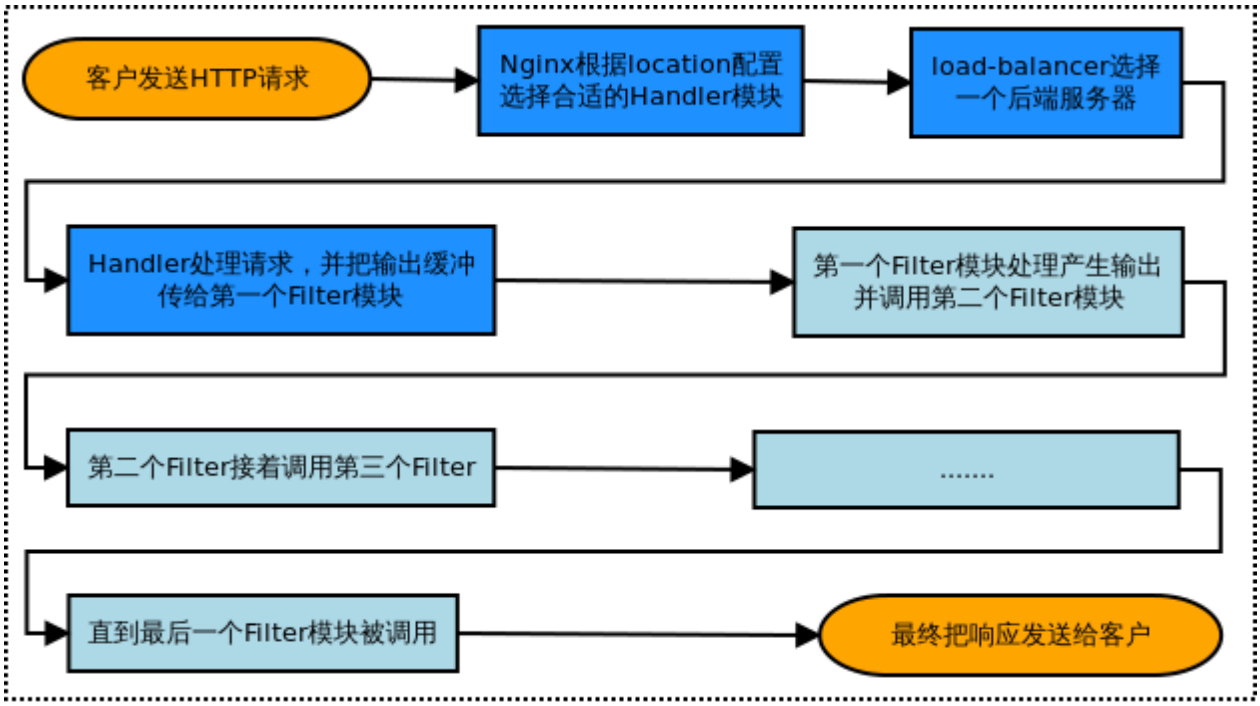


图 1 Nginx服务器的模块工作流程

Nginx 模块的结构

模块的配置结构

模块最多可以定义三个配置结构：main、server、location。绝大多数模块仅需要一个location 配置。名称约定如下以ngx_http__(main|srv|loc)_conf_t为例的dav module：

```
typedef struct {
    ngx_uint_t methods;
    ngx_flag_t create_full_put_path;
    ngx_uint_t access;
} ngx_http_dav_loc_conf_t;
```

Nginx 模块的数据结构如下定义：

```
/* Nginx 模块的数据结构 */
#define NGX_MODULE_V1          0, 0, 0, 0, 0, 0, 1
#define NGX_MODULE_V1_PADDING  0, 0, 0, 0, 0, 0, 0, 0

struct ngx_module_s {
    /* 模块类别由type成员决定，ctx_index表示当前模块在type类模块中的序号 */
    ngx_uint_t            ctx_index;
    /* index 区别与ctx_index，index表示当前模块在所有模块中的序号 */
    ngx_uint_t            index;

    /* spare 序列保留变量，暂时不被使用 */
    ngx_uint_t            spare0;
    ngx_uint_t            spare1;
    ngx_uint_t            spare2;
    ngx_uint_t            spare3;

    /* 当前模块的版本 */
    ngx_uint_t            version;

    /* ctx指向特定类型模块的公共接口，例如在HTTP模块中，ctx指向ngx_http_module_t结构体 */
    void                  *ctx;
    /* 处理nginx.conf中的配置项 */
    ngx_command_t         *commands;
    /* type表示当前模块的类型 */
    ngx_uint_t            type;

    /* 下面的7个函数指针是在Nginx启动或停止时，分别调用的7中方法 */
    /* 在master进程中回调init_master */
    ngx_int_t             (*init_master)(ngx_log_t *log);

    /* 初始化所有模块时回调init_module */
    ngx_int_t             (*init_module)(ngx_cycle_t *cycle);

    /* 在worker进程提供正常服务之前回调init_process初始化进程 */
    ngx_int_t             (*init_process)(ngx_cycle_t *cycle);
    /* 初始化多线程 */
    ngx_int_t             (*init_thread)(ngx_cycle_t *cycle);
    /* 退出多线程 */
    void                  (*exit_thread)(ngx_cycle_t *cycle);
    /* 在worker进程停止服务之前回调exit_process */
    void                  (*exit_process)(ngx_cycle_t *cycle);

    /* 在master进程退出之前回调exit_master */
    void                  (*exit_master)(ngx_cycle_t *cycle);

    /* 保留字段，未被使用 */
    uintptr_t             spare_hook0;
    uintptr_t             spare_hook1;
    uintptr_t             spare_hook2;
    uintptr_t             spare_hook3;
    uintptr_t             spare_hook4;
    uintptr_t             spare_hook5;
    uintptr_t             spare_hook6;
    uintptr_t             spare_hook7;
```

```
};
```

在该数据结构中，其中最重要的是两个成员 ctx和commands，这里两个成员会在分别在下面的模块配置指令和模块上下文中讲解；若是HTTP 模块时，type 字段必须定义为NGX_HTTP_MODULE；

模块配置指令

模块指令存储在一个 ngx_command_t 类型的静态数组结构中，例如：

```
static ngx_command_t  ngx_http_circle_gif_commands[] = {
    { ngx_string("circle_gif"),
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
      ngx_http_circle_gif,
      NGX_HTTP_LOC_CONF_OFFSET,
      0,
      NULL },

    { ngx_string("circle_gif_min_radius"),
      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_num_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_circle_gif_loc_conf_t, min_radius),
      NULL },
      ...
      ngx_null_command
};
```

ngx_command_t 类型定义在 core/ngx_conf_file.h：

```
struct ngx_command_s {
    /* 配置项名称 */
    ngx_str_t             name;
    /* 配置项类型，type将指定配置项可以出现的位置以及携带参数的个数 */
    ngx_uint_t            type;
    /* 处理配置项的参数 */
    char               *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    /* 在配置文件中的偏移量，conf与offset配合使用 */
    ngx_uint_t            conf;
    ngx_uint_t            offset;
    /* 配置项读取后的处理方法，必须指向ngx_conf_post_t 结构 */
    void                  *post;
};
```

name ：配置指令的名称；

type ：该配置的类型，指定配置项的出现位置以及可携带参数的个数，下面规定只是其中一部分，更多信息可查看文件core/ngx_conf_file.h：

```
NGX_HTTP_MAIN_CONF:  directive is valid in the main config
NGX_HTTP_SRV_CONF:   directive is valid in the server (host) config
NGX_HTTP_LOC_CONF:   directive is valid in a location config
NGX_HTTP_UPS_CONF:   directive is valid in an upstream config
NGX_CONF_NOARGS:     directive can take 0 arguments
NGX_CONF_TAKE1:      directive can take exactly 1 argument
NGX_CONF_TAKE2:      directive can take exactly 2 arguments
...
NGX_CONF_TAKE7:      directive can take exactly 7 arguments
NGX_CONF_FLAG:       directive takes a boolean ("on" or "off")
NGX_CONF_1MORE:      directive must be passed at least one argument
NGX_CONF_2MORE:      directive must be passed at least two arguments
```

set　：这是一个函数指针，当Nginx 在解析配置时，若遇到该配置指令，将会把读取到的值传递给这个函数进行分解处理。因为具体每个配置指令的值如何处理，只有定义这个配置指令的人是最清楚的。来看一下这个函数指针要求的函数原型。

```
char *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
```

该函数处理成功时，返回 NGX_OK，否则返回 NGX_CONF_ERROR 或者是一个自定义的错误信息的字符串。该函数传入三个类型的参数：

1. cf　：指向ngx_conf_t 结构的指针，该结构包括从配置指令传递的参数；
2. cmd：指向当前ngx_command_t 结构；
3. conf：指向模块配置结构；

为了方便实现对配置指令参数的读取，Nginx 已经默认提供了对一些标准类型的参数进行读取的函数，可以直接赋值给set 字段使用。下面是一部分已经实现的set 类型函数，更多可参考文件core/ngx_conf_file.h：

- ngx_conf_set_flag_slot : 把 "on" 或 "off" 解析为 1 或 0；
- ngx_conf_set_str_slot　: 解析字符串并保存 ngx_str_t类型；
- ngx_conf_set_num_slot: 解析一个数字并将其保存为int 类型；
- ngx_conf_set_size_slot: 解析数据大小 ("8k", "1m", etc.) 并将其保存为size_t；

conf　：用于指示配置项所处内存的相对偏移量，仅在type 中没有设置NGX_DIRECT_CONF 和 NGX_MAIN_CONF 时才生效。对于HTTP 模块，conf 必须设置，它的取值如下：

- NGX_HTTP_MAIN_CONF_OFFSET：使用create_main_conf 方法产生的结构体来存储解析出的配置项参数；
- NGX_HTTP_SRV_CONF_OFFSET：使用 create_srv_conf 方法产生的结构体来存储解析出的配置项参数；
- NGX_HTTP_LOC_CONF_OFFSET：使用 create_loc_conf 方法产生的结构体来存储解析出的配置项参

数；

offset ：表示当前配置项在整个存储配置项的结构体中的偏移位置。

模块上下文

这是一个静态的 ngx_http_module_t 结构，它的名称是ngx_http_ _module_ctx。以下是该结构的定义，具体可查阅文件 http/ngx_http_config.h：

- preconfiguration
- postconfiguration
- creating the main conf (i.e., do a malloc and set defaults)
- initializing the main conf (i.e., override the defaults with what's in nginx.conf)
- creating the server conf
- merging it with the main conf
- creating the location conf
- merging it with the server conf

```
typedef struct{/* 可以把不需要调用的函数指针设置为 NULL */
    /* 解析配置文件之前被调用 */
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    /* 完成配置文件的解析后被调用 */
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    /* 创建存储main级别的全局配置项的结构体（直属于http块） */
    void        *(*create_main_conf)(ngx_conf_t *cf);
    /* 初始化main级别的配置项 */
    char        *(*init_main_conf)(ngx_conf_t *cf);

    /* 创建存储srv级别的配置项的结构体（直属于server块） */
    void        *(*create_srv_conf)(ngx_conf_t *cf);
    /* 合并main级别与srv级别下的同名配置项 */
    char        *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    /* 创建存储loc级别的配置项的结构体（直属于location块） */
    void        *(*create_loc_conf)(ngx_conf_t *cf);
    /* 合并srv级别与loc级别下的同名配置项 */
    char        *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
}ngx_http_module_t;
```

在以上的结构内容中，大多数模块只使用最后两项：ngx_http_ *create_loc_conf和 ngx_http*_merge_loc_conf；例如：

```
static ngx_http_module_t  ngx_http_circle_gif_module_ctx = {
    NULL,                          /* preconfiguration */
    NULL,                          /* postconfiguration */

    NULL,                          /* create main configuration */
    NULL,                          /* init main configuration */

    NULL,                          /* create server configuration */
    NULL,                          /* merge server configuration */

    ngx_http_circle_gif_create_loc_conf,  /* create location configuration */
    ngx_http_circle_gif_merge_loc_conf /* merge location configuration */
};
```

下面针对最后两项进行说明，以下是以 circle_gif 模块为例子，该 模块源码；

create_loc_conf 函数

该函数是传入一个 ngx_conf_t 结构的参数，返回新创建模块的配置结构，在这里是返回：ngx_http_circle_gif_loc_conf_t

```
static void *
ngx_http_circle_gif_create_loc_conf(ngx_conf_t *cf)
{
    ngx_http_circle_gif_loc_conf_t  *conf;

    conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_circle_gif_loc_conf_t));
    if (conf == NULL) {
        return NGX_CONF_ERROR;
    }
    conf->min_radius = NGX_CONF_UNSET_UINT;
    conf->max_radius = NGX_CONF_UNSET_UINT;
    return conf;
}
```

merge_loc_conf 函数

Nginx 为不同的数据类型提供了merge 函数，可查阅 core/ngx_conf_file.h；merge_loc_conf 函数定义如下：

```
static char *
ngx_http_circle_gif_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
{
    ngx_http_circle_gif_loc_conf_t *prev = parent;
    ngx_http_circle_gif_loc_conf_t *conf = child;

    ngx_conf_merge_uint_value(conf->min_radius, prev->min_radius, 10);
    ngx_conf_merge_uint_value(conf->max_radius, prev->max_radius, 20);

    if (conf->min_radius < 1) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "min_radius must be equal or more than 1");
        return NGX_CONF_ERROR;
    }
    if (conf->max_radius < conf->min_radius) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "max_radius must be equal or more than min_radius");
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;
}
```

模块的定义

对任何开发模块，都需要定义一个 ngx_module_t 类型的变量来说明这个模块本身的信息，它告诉了 Nginx 这个模块的一些信息。这个变量是 ngx_http__module；例如：更多例子可查找文件 core/ngx_conf_file.h；

```
ngx_module_t  ngx_http_<module name>_module = {
    NGX_MODULE_V1,
    &ngx_http_<module name>_module_ctx, /* module context */
    ngx_http_<module name>_commands,   /* module directives */
    NGX_HTTP_MODULE,               /* module type */
    NULL,                          /* init master */
    NULL,                          /* init module */
    NULL,                          /* init process */
    NULL,                          /* init thread */
    NULL,                          /* exit thread */
    NULL,                          /* exit process */
    NULL,                          /* exit master */
    NGX_MODULE_V1_PADDING
};
```

Handler 模块

Handler 模块必须提供一个真正的处理函数，这个函数负责处理来自客户端的请求。该函数既可以选择自己直接生成内容，也可以选择拒绝处理，并由后续的 Handler 去进行处理，或者是选择丢给后续的 Filter 模块进行处理。以下是该函数的原型：

```
typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
```

其中r 是 request 结构http 请求，包含客户端请求所有的信息，例如：request method, URI, and headers。 该函数处理成功返回NGX_OK，处理发生错误返回NGX_ERROR，拒绝处理（留给后续的 Handler 进行处理）返回NGX_DECLINE。 返回NGX_OK 也就代表给客户端的响应已经生成，否则返回 NGX_ERROR 就发生错误了。

Handler 模块处理过程中做了四件事情： 获取 location 配置、生成合适的响应、发送响应的 header 头部、发送响应的 body 包体。

获取 location 配置

获取 location 配置 指向调用 ngx_http_get_module_loc_conf 函数即可，该函数传入的参数是 request 结构和 自定义的 module 模块。例如：circle gif模块；

```
static ngx_int_t
ngx_http_circle_gif_handler(ngx_http_request_t *r)
{
    ngx_http_circle_gif_loc_conf_t  *circle_gif_config;
    circle_gif_config = ngx_http_get_module_loc_conf(r, ngx_http_circle_gif_module);
    ...
}
```

生成合适的响应

这里主要是 request 结构，其定义如下：更多可参考文件 http/ngx_http_request.h；

```
typedef struct {
...
/* the memory pool, used in the ngx_palloc functions */
    ngx_pool_t                 *pool;
    ngx_str_t                  uri;
    ngx_str_t                  args;
    ngx_http_headers_in_t            headers_in;

...
} ngx_http_request_t;
```

其中参数的意义如下：

- uri 是 request 请求的路径,e.g. "/query.cgi".
- args 是请求串参数中问号后面的参数(e.g. "name=john").
- headers_in 包含有用的stuff，例如：cookies 和browser 信息。

发送响应的 header 头部

发送响应头部有函数ngx_http_send_header(r) 实现。响应的header 头部在 headers_out 结构中，定义如下：更多可参考文件 http/ngx_http_request.h ；

```
typedef stuct {
...
    ngx_uint_t                 status;
    size_t                     content_type_len;
    ngx_str_t                  content_type;
    ngx_table_elt_t            *content_encoding;
    off_t                      content_length_n;
    time_t                     date_time;
    time_t                     last_modified_time;
..
} ngx_http_headers_out_t;
```

例如，一个模块设置为 Content-Type to "image/gif", Content-Length to 100, and return a 200 OK response，则其实现为：

```
r->headers_out.status = NGX_HTTP_OK;
    r->headers_out.content_length_n = 100;
    r->headers_out.content_type.len = sizeof("image/gif") - 1;
    r->headers_out.content_type.data = (u_char *) "image/gif";
    ngx_http_send_header(r);
```

假如content_encoding 是 (ngx_table_elt_t*)类型时，则模块需要为这些类型分配内存，可以调用 ngx_list_push 函数，实现如下：

```
r->headers_out.content_encoding = ngx_list_push(&r->headers_out.headers);
if (r->headers_out.content_encoding == NULL) {
return NGX_ERROR;
}
r->headers_out.content_encoding->hash = 1;
r->headers_out.content_encoding->key.len = sizeof("Content-Encoding") - 1;
r->headers_out.content_encoding->key.data = (u_char *) "Content-Encoding";
r->headers_out.content_encoding->value.len = sizeof("deflate") - 1;
r->headers_out.content_encoding->value.data = (u_char *) "deflate";
ngx_http_send_header(r);
```

发送响应的 body 包体

到此，该模块已经产生响应，并把它存储在内存中。发送包体的步骤是：首先分配响应特殊的缓冲区，然后分配缓冲区链接到chain link，然后在 chain link 调用发送函数。

1、chain links 是 Nginx 使 Handler 模块在缓冲区中产生响应。在 chain 中每个 chain link 有一个指向下一个 link 的指针。首先，模块声明缓冲区 buffer 和 chain link：

```
ngx_buf_t    *b;
ngx_chain_t   out;
```

2、然后分配缓冲区 buffer，使响应数据指向它：

```
b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    if (b == NULL) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
            "Failed to allocate response buffer.");
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    b->pos = some_bytes; /* first position in memory of the data */
    b->last = some_bytes + some_bytes_length; /* last position */

    b->memory = 1; /* content is in read-only memory */
    /* (i.e., filters should copy it rather than rewrite in place) */

    b->last_buf = 1; /* there will be no more buffers in the request */
```

3、接着，把模块挂载到 chain link 上：

```
out.buf = b;
out.next = NULL;
```

4、最后，发送包体：

```
return ngx_http_output_filter(r, &out);
```

Handler 模块挂载

Handler 模块真正的处理函数通过两种方式挂载到处理过程中：按处理阶段挂载；按需挂载。

按处理阶段挂载

为了更精细地控制对于客户端请求的处理过程，Nginx 把这个处理过程划分成了11个阶段。依次列举如下：

```
NGX_HTTP_POST_READ_PHASE:
/* 读取请求内容阶段 */
NGX_HTTP_SERVER_REWRITE_PHASE:
/* Server请求地址重写阶段 */
NGX_HTTP_FIND_CONFIG_PHASE:
/* 配置查找阶段: */
NGX_HTTP_REWRITE_PHASE:
/* Location请求地址重写阶段 */
NGX_HTTP_POST_REWRITE_PHASE:
/* 请求地址重写提交阶段 */
NGX_HTTP_PREACCESS_PHASE:
/* 访问权限检查准备阶段 */
NGX_HTTP_ACCESS_PHASE:
/* 访问权限检查阶段 */
NGX_HTTP_POST_ACCESS_PHASE:
/* 访问权限检查提交阶段 */
NGX_HTTP_TRY_FILES_PHASE:
/* 配置项try_files处理阶段 */
NGX_HTTP_CONTENT_PHASE:
/* 内容产生阶段 */
NGX_HTTP_LOG_PHASE:
/* 日志模块处理阶段 */
```

一般情况下，我们自定义的模块，大多数是挂载在NGX_HTTP_CONTENT_PHASE阶段的。挂载的动作一般是在模块上下文调用的postconfiguration 函数中。注意：有几个阶段是特例，它不调用挂载任何的Handler，也就是你就不用挂载到这几个阶段了：

```
NGX_HTTP_FIND_CONFIG_PHASE
NGX_HTTP_POST_ACCESS_PHASE
NGX_HTTP_POST_REWRITE_PHASE
NGX_HTTP_TRY_FILES_PHASE
```

按需挂载

以这种方式挂载的Handler 也被称为content handler。当一个请求进来以后，Nginx 从NGX_HTTP_POST_READ_PHASE 阶段开始依次执行每个阶段中所有 Handler。执行到 NGX_HTTP_CONTENT_PHASE 阶段时，如果这个location 有一个对应的content handler 模块，那么就去执行这个content handler 模块真正的处理函数。否则继续依次执行NGX_HTTP_CONTENT_PHASE阶段中所有content phase handlers，直到某个函数处理返回NGX_OK 或者NGX_ERROR。但是使用这个方法挂载上去的handler 有一个特点是必须在NGX_HTTP_CONTENT_PHASE 阶段才能被执行。如果你想自己的handler 更早的阶段被执行，那就不要使用这种挂载方式。

以下是例子：

```
circle gif ngx_command_t looks like this:

    { ngx_string("circle_gif"),
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
      ngx_http_circle_gif,
      0,
      0,
      NULL }
```

挂载函数：

```
static char *
ngx_http_circle_gif(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_core_loc_conf_t  *clcf;

    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
    clcf->handler = ngx_http_circle_gif_handler;

    return NGX_CONF_OK;
}
```

Handler 模块编写

Handler 模块编写步骤如下：

1. 编写模块基本结构：包括模块的定义，模块上下文结构，模块的配置结构等；
2. 实现 handler 的挂载函数；根据模块的需求选择正确的挂载方式；
3. 编写 handler 处理函数；模块的功能主要通过这个函数来完成；

Filter 模块

Filter 处理由Handler 模块产生的响应，即仅处理由服务器发往客户端的HTTP 响应，并不处理由客户端发往服务器的 HTTP 请求。Filter 模块包括过滤头部（Header Filter）和过滤包体（Body Filter），Filter 模块过滤头部处理HTTP 的头部（HTTP headers），Filter 包体处理响应内容（response content）（即HTTP 包体），这两个阶段可以对HTTP 响应头部和内容进行修改。

Filter 模块 HTTP 响应的方法如下：定义在文件 src/http/ngx_http_core_module.h

```
typedef ngx_int_t (*ngx_http_output_header_filter_pt) (ngx_http_request_t *r);
typedef ngx_int_t (*ngx_http_output_body_filter_pt) (ngx_http_request_t *r, ngx_chain_t *chain);
```

其中，参数 r 是当前的请求，chain 是待发送的 HTTP 响应包体；

所有 HTTP 过滤模块都需要实现上面的两个方法，在 HTTP 过滤模块组成的链表中，链表元素就是处理方法。HTTP 框架定义了链表入口：

```
extern ngx_http_output_header_filter_pt  ngx_http_top_header_filter;
extern ngx_http_output_body_filter_pt  ngx_http_top_body_filter;
```

过滤模块链表中通过 next 遍历，其定义如下：

```
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
```

当执行发送 HTTP 头部或 HTTP 响应包体时，HTTP 框架是从 ngx_http_top_header_filter 和 ngx_http_top_body_filter 开始遍历 HTTP 头部过滤模块和 HTTP 包体过来模块。其源码实现在文件：src/http/ngx_http_core_module.c

```
/* 发送 HTTP 响应头部 */
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
    if (r->header_sent) {
        ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
                      "header already sent");
        return NGX_ERROR;
    }

    if (r->err_status) {
        r->headers_out.status = r->err_status;
        r->headers_out.status_line.len = 0;
    }

    return ngx_http_top_header_filter(r);
}

/* 发送HTTP 响应包体 */
ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t          rc;
    ngx_connection_t  *c;

    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http output filter \"%V?%V\"", &r->uri, &r->args);

    rc = ngx_http_top_body_filter(r, in);

    if (rc == NGX_ERROR) {
        /* NGX_ERROR may be returned by any filter */
        c->error = 1;
    }

    return rc;
}
```

Filter 模块相关结构

Filter 模块是采用链表形式的，其基本结构是ngx_chain_t 和 ngx_buf_t；这两种结构定义如下：

```
typedef struct ngx_chain_s ngx_chain_t;

        struct ngx_chain_s {
                ngx_buf_t    *buf;
                ngx_chain_t  *next;
        };

struct ngx_buf_s {
```

```
    u_char          *pos;       /* 当前buffer真实内容的起始位置 */
    u_char          *last;      /* 当前buffer真实内容的结束位置 */
    off_t           file_pos;  /* 在文件中真实内容的起始位置   */
    off_t           file_last; /* 在文件中真实内容的结束位置   */

    u_char          *start;    /* buffer内存的开始分配的位置 */
    u_char          *end;      /* buffer内存的结束分配的位置 */
    ngx_buf_tag_t    tag;      /* buffer属于哪个模块的标志 */
    ngx_file_t      *file;     /* buffer所引用的文件 */

    /* 用来引用替换过后的buffer，以便当所有buffer输出以后，
     * 这个影子buffer可以被释放。
     */
    ngx_buf_t       *shadow;

    /* the buf's content could be changed */
    unsigned         temporary:1;

    /*
     * the buf's content is in a memory cache or in a read only memory
     * and must not be changed
     */
    unsigned         memory:1;

    /* the buf's content is mmap()ed and must not be changed */
    unsigned         mmap:1;

    unsigned         recycled:1; /* 内存可以被输出并回收 */
    unsigned         in_file:1;  /* buffer的内容在文件中 */
    /* 马上全部输出buffer的内容, gzip模块里面用得比较多 */
    unsigned         flush:1;
    /* 基本上是一段输出链的最后一个buffer带的标志，标示可以输出，
     * 有些零长度的buffer也可以置该标志
     */
    unsigned         sync:1;
    /* 所有请求里面最后一块buffer，包含子请求 */
    unsigned         last_buf:1;
    /* 当前请求输出链的最后一块buffer         */
    unsigned         last_in_chain:1;
    /* shadow链里面的最后buffer，可以释放buffer了 */
    unsigned         last_shadow:1;
    /* 是否是暂存文件 */
    unsigned         temp_file:1;

    /* 统计用，表示使用次数 */
    /* STUB */ int    num;
};
```

Filter 过滤头部

header filter 包含三个基本步骤：

1. 决定是否处理响应；

2. 对响应进行处理；

3. 调用下一个 filter；

例如下面的"not modified" header filter：其中 headers_out 结构可参考文件 http/ngx_http_request.h；

```
static
ngx_int_t ngx_http_not_modified_header_filter(ngx_http_request_t *r)
{
time_t if_modified_since;

if_modified_since = ngx_http_parse_time(r->headers_in.if_modified_since->value.data,
r->headers_in.if_modified_since->value.len);

/* step 1: decide whether to operate */
if (if_modified_since != NGX_ERROR &&
if_modified_since == r->headers_out.last_modified_time) {

/* step 2: operate on the header */
r->headers_out.status = NGX_HTTP_NOT_MODIFIED;
r->headers_out.content_type.len = 0;
ngx_http_clear_content_length(r);
ngx_http_clear_accept_ranges(r);
}

/* step 3: call the next filter */
return ngx_http_next_header_filter(r);
}
```

Filter 过滤包体

Filter 包体只能在chain link缓冲区buffer 中操作。模块必须决定是否修改输入缓冲区，或分配新的缓冲区替换当前缓冲区，或是在当前缓冲区之后还是之前插入新的缓冲区。很多模块接收多个缓冲区，导致这些模块在不完整的chain 缓冲区中操作。Filter 包体操作如下：

```
static ngx_int_t ngx_http_chunked_body_filter(ngx_http_request_t *r, ngx_chain_t *in);
```

以下是一个例子：

```
/*
 * Let's take a simple example.
 * Suppose we want to insert the text "<!-- Served by Nginx -->" to the end of every request.
 * First, we need to figure out if the response's final buffer is included in the buffer chain we were give
n.
 * Like I said, there's not a fancy API, so we'll be rolling our own for loop:
 */

    ngx_chain_t *chain_link;
```

```
    int chain_contains_last_buffer = 0;

    chain_link = in;
    for ( ; ; ) {
        if (chain_link->buf->last_buf)
            chain_contains_last_buffer = 1;
        if (chain_link->next == NULL)
            break;
        chain_link = chain_link->next;
    }
/*
 * Now let's bail out if we don't have that last buffer:
 */

    if (!chain_contains_last_buffer)
        return ngx_http_next_body_filter(r, in);
/*
 * Super, now the last buffer is stored in chain_link.
 * Now we allocate a new buffer:
 */

    ngx_buf_t    *b;
    b = ngx_calloc_buf(r->pool);
    if (b == NULL) {
        return NGX_ERROR;
    }
/*
 * And put some data in it:
 */

    b->pos = (u_char *) "<!-- Served by Nginx -->";
    b->last = b->pos + sizeof("<!-- Served by Nginx -->") - 1;
/*
 * And hook the buffer into a new chain link:
 */

    ngx_chain_t   *added_link;

    added_link = ngx_alloc_chain_link(r->pool);
    if (added_link == NULL)
        return NGX_ERROR;

    added_link->buf = b;
    added_link->next = NULL;
/*
 * Finally, hook the new chain link to the final chain link we found before:
 */

    chain_link->next = added_link;
/*
 * And reset the "last_buf" variables to reflect reality:
 */

    chain_link->buf->last_buf = 0;
    added_link->buf->last_buf = 1;
```

```
/*
 * And pass along the modified chain to the next output filter:
 */

    return ngx_http_next_body_filter(r, in);

/*
 * The resulting function takes much more effort than what you'd do with, say, mod_perl ($response->body =~ s/$/<!-- Served by mod_perl -->/),
 * but the buffer chain is a very powerful construct, allowing programmers to process data incrementally so that the client gets something as soon as possible.
 * However, in my opinion, the buffer chain desperately needs a cleaner interface so that programmers can't leave the chain in an inconsistent state.
 * For now, manipulate it at your own risk.
 */
```

Filter 模块挂载

Filters 模块和Handler 模块一样，也是挂载到post-configuration ，如下面代码所示：

```
static ngx_http_module_t ngx_http_chunked_filter_module_ctx = {
NULL, /* preconfiguration */
ngx_http_chunked_filter_init, /* postconfiguration */
...
};
```

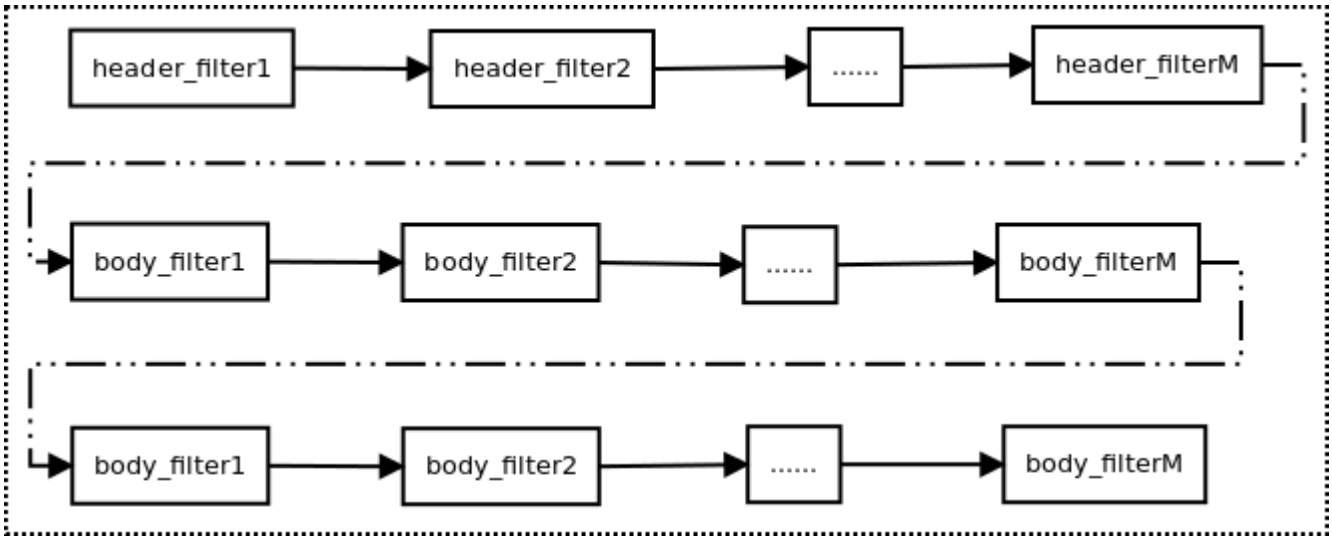
其中 ngx_http_chunked_filter_init 处理如下定义：

```
static ngx_int_t
ngx_http_chunked_filter_init(ngx_conf_t *cf)
{
ngx_http_next_header_filter = ngx_http_top_header_filter;
ngx_http_top_header_filter = ngx_http_chunked_header_filter;

ngx_http_next_body_filter = ngx_http_top_body_filter;
ngx_http_top_body_filter = ngx_http_chunked_body_filter;

return NGX_OK;
}
```

由于 Filter 模块是 “CHAIN OF RESPONSIBILITY” 链表模式的。Handler 模块生成响应后，Filter 模块调用两个函数：ngx_http_output_filter 和 ngx_http_send_header，其中ngx_http_output_filter 函数是调用全局函数 ngx_http_top_body_filter；ngx_http_send_header 函数是调用全局函数 ngx_http_top_header_filter。

```
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
        ...

        return ngx_http_top_header_filter(r);
}

ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t          rc;
    ngx_connection_t  *c;

    c = r->connection;

    rc = ngx_http_top_body_filter(r, in);

    if (rc == NGX_ERROR) {
        /* NGX_ERROR may be returned by any filter */
        c->error = 1;
    }

    return rc;
}
```

Filter 模块的执行方式如下图所示：



Filter 模块编写

Filter 模块编写步骤如下

- 编写基本结构：模块定义，上下文结构，基本结构；
- 初始化过滤模块：把本模块中处理的 HTTP 头部的 ngx_http_output_header_filter_pt 方法与处理 HTTP 包体的ngx_http_output_body_filter_pt 方法插入到过滤模块链表首部；
- 实现处理 HTTP 响应的方法：处理 HTTP 头部，即 ngx_http_output_header_filter_pt 方法的实现，

处理HTTP 包体的方法，即ngx_http_output_body_filter_pt 方法的实现；

- 编译安装；

开发 Nginx 新模块

把自己开发的模块编译到 Nginx 中需要编写两个文件：

1. "config"，该文件会被 ./configure 包含；
2. "ngx_http__module.c"，该文件是定义模块的功能；

config 文件的编写如下：

```
/*
 * "config" for filter modules:
 */

ngx_addon_name=ngx_http_<your module>_module    /* 模块的名称 */
HTTP_AUX_FILTER_MODULES="$HTTP_AUX_FILTER_MODULES ngx_http_<your module>_module"    /
* 保存所有 HTTP 模块*/
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_<your module>_module.c"
 /* 指定新模块的源码路径 */

/*
 * "config" for other modules:
 */

ngx_addon_name=ngx_http_<your module>_module
HTTP_MODULES="$HTTP_MODULES ngx_http_<your module>_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_<your module>_module.c"
```

关于 "ngx_http_ _module.c" 文件的编写，可参考上面的Handler 模块，同时可参考Nginx 现有的模块：src/http/modules/；例如下面的“Hello World ” 代码：

```
#include <ngx_config.h>
#include <ngx_core.h>
#include <ngx_http.h>

typedef struct
{
        ngx_str_t hello_string;
        ngx_int_t hello_counter;
}ngx_http_hello_loc_conf_t;

static ngx_int_t ngx_http_hello_init(ngx_conf_t *cf);

static void *ngx_http_hello_create_loc_conf(ngx_conf_t *cf);

static char *ngx_http_hello_string(ngx_conf_t *cf, ngx_command_t *cmd,
        void *conf);
static char *ngx_http_hello_counter(ngx_conf_t *cf, ngx_command_t *cmd,
```

```
        void *conf);

static ngx_command_t ngx_http_hello_commands[] = {
  {
        ngx_string("hello_string"),
        NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS|NGX_CONF_TAKE1,
        ngx_http_hello_string,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_string),
        NULL },

    {
        ngx_string("hello_counter"),
        NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
        ngx_http_hello_counter,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_counter),
        NULL },

    ngx_null_command
};

/*
static u_char ngx_hello_default_string[] = "Default String: Hello, world!";
*/
static int ngx_hello_visited_times = 0;

static ngx_http_module_t ngx_http_hello_module_ctx = {
    NULL,                       /* preconfiguration */
    ngx_http_hello_init,          /* postconfiguration */

    NULL,                       /* create main configuration */
    NULL,                       /* init main configuration */

    NULL,                       /* create server configuration */
    NULL,                       /* merge server configuration */

    ngx_http_hello_create_loc_conf, /* create location configuration */
    NULL                         /* merge location configuration */
};

ngx_module_t ngx_http_hello_module = {
    NGX_MODULE_V1,
    &ngx_http_hello_module_ctx,    /* module context */
    ngx_http_hello_commands,       /* module directives */
    NGX_HTTP_MODULE,               /* module type */
    NULL,                       /* init master */
    NULL,                       /* init module */
    NULL,                       /* init process */
    NULL,                       /* init thread */
    NULL,                       /* exit thread */
    NULL,                       /* exit process */
    NULL,                       /* exit master */
    NGX_MODULE_V1_PADDING
};
```

```
};

static ngx_int_t
ngx_http_hello_handler(ngx_http_request_t *r)
{
    ngx_int_t    rc;
    ngx_buf_t   *b;
    ngx_chain_t  out;
    ngx_http_hello_loc_conf_t* my_conf;
    u_char ngx_hello_string[1024] = {0};
    ngx_uint_t content_length = 0;

    ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0, "ngx_http_hello_handler is called!");

    my_conf = ngx_http_get_module_loc_conf(r, ngx_http_hello_module);
    if (my_conf->hello_string.len == 0 )
    {
        ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0, "hello_string is empty!");
        return NGX_DECLINED;
    }

    if (my_conf->hello_counter == NGX_CONF_UNSET
        || my_conf->hello_counter == 0)
    {
        ngx_sprintf(ngx_hello_string, "%s", my_conf->hello_string.data);
    }
    else
    {
        ngx_sprintf(ngx_hello_string, "%s Visited Times:%d", my_conf->hello_string.data,
                ++ngx_hello_visited_times);
    }
    ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0, "hello_string:%s", ngx_hello_string);
    content_length = ngx_strlen(ngx_hello_string);

    /* we response to 'GET' and 'HEAD' requests only */
    if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
        return NGX_HTTP_NOT_ALLOWED;
    }

    /* discard request body, since we don't need it here */
    rc = ngx_http_discard_request_body(r);

    if (rc != NGX_OK) {
        return rc;
    }

    /* set the 'Content-type' header */
    /*
     *r->headers_out.content_type.len = sizeof("text/html") - 1;
     *r->headers_out.content_type.data = (u_char *)"text/html";
     */
    ngx_str_set(&r->headers_out.content_type, "text/html");

    /* send the header only, if the request type is http 'HEAD' */
    if (r->method == NGX_HTTP_HEAD) {
```

```
        r->headers_out.status = NGX_HTTP_OK;
        r->headers_out.content_length_n = content_length;

        return ngx_http_send_header(r);
    }

    /* allocate a buffer for your response body */
    b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    if (b == NULL) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    /* attach this buffer to the buffer chain */
    out.buf = b;
    out.next = NULL;

    /* adjust the pointers of the buffer */
    b->pos = ngx_hello_string;
    b->last = ngx_hello_string + content_length;
    b->memory = 1;    /* this buffer is in memory */
    b->last_buf = 1;  /* this is the last buffer in the buffer chain */

    /* set the status line */
    r->headers_out.status = NGX_HTTP_OK;
    r->headers_out.content_length_n = content_length;

    /* send the headers of your response */
    rc = ngx_http_send_header(r);

    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
        return rc;
    }

    /* send the buffer chain of your response */
    return ngx_http_output_filter(r, &out);
}

static void *ngx_http_hello_create_loc_conf(ngx_conf_t *cf)
{
    ngx_http_hello_loc_conf_t* local_conf = NULL;
    local_conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_hello_loc_conf_t));
    if (local_conf == NULL)
    {
        return NULL;
    }

    ngx_str_null(&local_conf->hello_string);
    local_conf->hello_counter = NGX_CONF_UNSET;

    return local_conf;
}

/*
static char *ngx_http_hello_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
{
```

```
{
    ngx_http_hello_loc_conf_t* prev = parent;
    ngx_http_hello_loc_conf_t* conf = child;

    ngx_conf_merge_str_value(conf->hello_string, prev->hello_string, ngx_hello_default_string);
    ngx_conf_merge_value(conf->hello_counter, prev->hello_counter, 0);

    return NGX_CONF_OK;
}*/

static char *
ngx_http_hello_string(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{

    ngx_http_hello_loc_conf_t* local_conf;

    local_conf = conf;
    char* rv = ngx_conf_set_str_slot(cf, cmd, conf);

    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_string:%s", local_conf->hello_string.data);

    return rv;
}

static char *ngx_http_hello_counter(ngx_conf_t *cf, ngx_command_t *cmd,
    void *conf)
{
    ngx_http_hello_loc_conf_t* local_conf;

    local_conf = conf;

    char* rv = NULL;

    rv = ngx_conf_set_flag_slot(cf, cmd, conf);

    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_counter:%d", local_conf->hello_counter);
    return rv;
}

static ngx_int_t
ngx_http_hello_init(ngx_conf_t *cf)
{
    ngx_http_handler_pt        *h;
    ngx_http_core_main_conf_t  *cmcf;

    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

    h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
    if (h == NULL) {
        return NGX_ERROR;
    }

    *h = ngx_http_hello_handler;

    return NGX_OK;
```

```
}
```

写好上面的两个文件后，在编译 Nginx 时，步骤如下：

```
./configure --add-module=path/to/your/new/module/directory
make
make install
```

参考资料：

《Emiller's Guide To Nginx Module Development》

《nginx模块开发篇》

《https://github.com/simpl/ngx_devel_kit》

Nginx 启动初始化过程

Nginx 启动过程

Nginx 的启动初始化由 main 函数完成，该函数是整个 Nginx 的入口，该函数完成 Nginx 启动初始化任务，也是所有功能模块的入口。Nginx 的初始化工作主要是一个类型为 ngx_cycle_t 类型的全局变量。main 函数定义在文件：src/core/nginx.c

Nginx 启动过程如下。

- 调用 ngx_get_options() 解析命令参数；
- 显示版本号与帮助信息；
- 调用 ngx_time_init() 初始化并更新时间；
- 调用 ngx_log_init() 初始化日志；
- 创建全局变量 init_cycle 的内存池 pool；
- 调用 ngx_save_argv() 保存命令行参数至全局变量ngx_os_argv、ngx_argc、ngx_argv 中；
- 调用 ngx_process_options() 初始化 init_cycle 的 prefix, conf_prefix, conf_file, conf_param 等字段；
- 调用 ngx_os_init() 初始化系统相关变量；
- 调用 ngx_crc32_table_init() 初始化CRC表；
- 调用 ngx_add_inherited_sockets() 继承 sockets；
- 通过环境变量 NGINX 完成 socket 的继承，将其保存在全局变量 init_cycle 的 listening 数组中；
- 初始化每个模块 module 的index，并计算 ngx_max_module；
- 调用 ngx_init_cycle() 进行初始化全局变量 init_cycle，这个步骤非常重要；
- 调用 ngx_signal_process() 处理进程信号；
- 调用 ngx_init_signals() 注册相关信号；
- 若无继承 sockets，则调用 ngx_daemon() 创建守护进程，并设置其标志；
- 调用 ngx_create_pidfile() 创建进程 ID 记录文件；
- 进入进程处理：
- 单进程工作模式；

- 多进程工作模式，即 master-worker 多进程工作模式；

```
int ngx_cdecl
main(int argc, char *const *argv)
{
    ngx_int_t         i;
    ngx_log_t        *log;
    ngx_cycle_t      *cycle, init_cycle;
    ngx_core_conf_t  *ccf;

    ngx_debug_init();

    if (ngx_strerror_init() != NGX_OK) {
        return 1;
    }

    /* 解析命令行参数 */
    if (ngx_get_options(argc, argv) != NGX_OK) {
        return 1;
    }

    /* 显示版本号与帮助信息 */
    if (ngx_show_version) {
        ngx_write_stderr("nginx version: " NGINX_VER NGX_LINEFEED);

        if (ngx_show_help) {
            ngx_write_stderr(
                "Usage: nginx [-?hvVtq] [-s signal] [-c filename] "
                            "[-p prefix] [-g directives]" NGX_LINEFEED
                            NGX_LINEFEED
                "Options:" NGX_LINEFEED
                "  -?,-h         : this help" NGX_LINEFEED
                "  -v            : show version and exit" NGX_LINEFEED
                "  -V            : show version and configure options then exit"
                                   NGX_LINEFEED
                "  -t            : test configuration and exit" NGX_LINEFEED
                "  -q            : suppress non-error messages "
                                   "during configuration testing" NGX_LINEFEED
                "  -s signal     : send signal to a master process: "
                                   "stop, quit, reopen, reload" NGX_LINEFEED
#ifdef NGX_PREFIX
                "  -p prefix     : set prefix path (default: "
                                   NGX_PREFIX ")" NGX_LINEFEED
#else
                "  -p prefix     : set prefix path (default: NONE)" NGX_LINEFEED
#endif
                "  -c filename   : set configuration file (default: "
                                   NGX_CONF_PATH ")" NGX_LINEFEED
                "  -g directives : set global directives out of configuration "
                                   "file" NGX_LINEFEED NGX_LINEFEED
                );
        }

        if (ngx_show_configure) {
```

```
            if (ngx_show_configure) {
                ngx_write_stderr(
#ifdef NGX_COMPILER
                    "built by " NGX_COMPILER NGX_LINEFEED
#endif
#if (NGX_SSL)
#ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
                    "TLS SNI support enabled" NGX_LINEFEED
#else
                    "TLS SNI support disabled" NGX_LINEFEED
#endif
#endif
                    "configure arguments:" NGX_CONFIGURE NGX_LINEFEED);
            }

            if (!ngx_test_config) {
                return 0;
            }
        }

        /* TODO */ ngx_max_sockets = -1;

        /* 初始化并更新时间 */
        ngx_time_init();

#if (NGX_PCRE)
        ngx_regex_init();
#endif

        ngx_pid = ngx_getpid();

        /* 初始化日志信息 */
        log = ngx_log_init(ngx_prefix);
        if (log == NULL) {
            return 1;
        }

        /* STUB */
#if (NGX_OPENSSL)
        ngx_ssl_init(log);
#endif

        /*
         * init_cycle->log is required for signal handlers and
         * ngx_process_options()
         */

        /* 全局变量init_cycle清零，并创建改变量的内存池pool */
        ngx_memzero(&init_cycle, sizeof(ngx_cycle_t));
        init_cycle.log = log;
        ngx_cycle = &init_cycle;

        init_cycle.pool = ngx_create_pool(1024, log);
        if (init_cycle.pool == NULL) {
            return 1;
```

```
    }

    /* 保存命令行参数至全局变量ngx_os_argv、ngx_argc、ngx_argv */
    if (ngx_save_argv(&init_cycle, argc, argv) != NGX_OK) {
        return 1;
    }

    /* 初始化全局变量init_cycle中的成员：prefix、conf_prefix、conf_file、conf_param 等字段 */
    if (ngx_process_options(&init_cycle) != NGX_OK) {
        return 1;
    }

    /* 初始化系统相关变量，如：内存页面大小ngx_pagesize、最大连接数ngx_max_sockets等 */
    if (ngx_os_init(log) != NGX_OK) {
        return 1;
    }

    /*
     * ngx_crc32_table_init() requires ngx_cacheline_size set in ngx_os_init()
     */

    /* 初始化 CRC 表（循环冗余校验表） */
    if (ngx_crc32_table_init() != NGX_OK) {
        return 1;
    }

    /* 通过环境变量NGINX完成socket的继承，将其保存在全局变量init_cycle的listening数组中 */
    if (ngx_add_inherited_sockets(&init_cycle) != NGX_OK) {
        return 1;
    }

    /* 初始化每个模块module的index，并计算ngx_max_module */
    ngx_max_module = 0;
    for (i = 0; ngx_modules[i]; i++) {
        ngx_modules[i]->index = ngx_max_module++;
    }

    /* 初始化全局变量init_cycle ，这里很重要 */
    cycle = ngx_init_cycle(&init_cycle);
    if (cycle == NULL) {
        if (ngx_test_config) {
            ngx_log_stderr(0, "configuration file %s test failed",
                           init_cycle.conf_file.data);
        }

        return 1;
    }

    if (ngx_test_config) {
        if (!ngx_quiet_mode) {
            ngx_log_stderr(0, "configuration file %s test is successful",
                           cycle->conf_file.data);
        }

        return 0;
```

```
    }

    /* 信号处理 */
    if (ngx_signal) {
        return ngx_signal_process(cycle, ngx_signal);
    }

    ngx_os_status(cycle->log);

    ngx_cycle = cycle;

    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

    if (ccf->master && ngx_process == NGX_PROCESS_SINGLE) {
        ngx_process = NGX_PROCESS_MASTER;
    }

#if !(NGX_WIN32)

    /* 初始化信号，注册相关信号 */
    if (ngx_init_signals(cycle->log) != NGX_OK) {
        return 1;
    }

    /* 若无socket继承，则创建守护进程，并设置守护进程标志 */
    if (!ngx_inherited && ccf->daemon) {
        if (ngx_daemon(cycle->log) != NGX_OK) {
            return 1;
        }

        ngx_daemonized = 1;
    }

    if (ngx_inherited) {
        ngx_daemonized = 1;
    }

#endif

    /* 记录进程ID */
    if (ngx_create_pidfile(&ccf->pid, cycle->log) != NGX_OK) {
        return 1;
    }

    if (ngx_log_redirect_stderr(cycle) != NGX_OK) {
        return 1;
    }

    if (log->file->fd != ngx_stderr) {
        if (ngx_close_file(log->file->fd) == NGX_FILE_ERROR) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          ngx_close_file_n " built-in log failed");
        }
    }
```

```
    ngx_use_stderr = 0;

    /* 进入进程处理 */
    if (ngx_process == NGX_PROCESS_SINGLE) {
        /* 单进程工作模式 */
        ngx_single_process_cycle(cycle);

    } else {
        /* master-worker 多进程模式工作 */
        ngx_master_process_cycle(cycle);
    }

    return 0;
}
```

ngx_cycle_t 变量初始化

ngx_cycle_t 结构体初始化

在初始化过程中，ngx_cycle_t 类型的全局变量最为重要，该类型结构定义如下：src/core/ngx_cycle.h

```
/* ngx_cycle_t 全局变量数据结构 */
struct ngx_cycle_s {
    /*
     * 保存所有模块配置项的结构体指针，该数组每个成员又是一个指针，
     * 这个指针又指向存储指针的数组
     */
    void                  **conf_ctx; /* 所有模块配置上下文的数组 */
    ngx_pool_t              *pool;    /* 内存池 */

    ngx_log_t               *log;     /* 日志 */
    ngx_log_t               new_log;

    ngx_uint_t              log_use_stderr;  /* unsigned  log_use_stderr:1; */

    ngx_connection_t        **files;    /* 连接文件 */
    ngx_connection_t        *free_connections; /* 空闲连接 */
    ngx_uint_t              free_connection_n;/* 空闲连接的个数 */

    /* 可再利用的连接队列 */
    ngx_queue_t             reusable_connections_queue;

    ngx_array_t             listening;    /* 监听数组 */
    ngx_array_t             paths;        /* 路径数组 */
    ngx_list_t              open_files;   /* 已打开文件的链表 */
    ngx_list_t              shared_memory;/* 共享内存链表 */

    ngx_uint_t              connection_n; /* 已连接个数 */
    ngx_uint_t              files_n;      /* 已打开文件的个数 */

    ngx_connection_t        *connections;  /* 连接 */
    ngx_event_t             *read_events;  /* 读事件 */
    ngx_event_t             *write_events; /* 写事件 */

    /* old 的 ngx_cycle_t 对象，用于引用前一个 ngx_cycle_t 对象的成员 */
    ngx_cycle_t             *old_cycle;

    ngx_str_t               conf_file;    /* nginx 配置文件 */
    ngx_str_t               conf_param;   /* nginx 处理配置文件时需要特殊处理的，在命令行携带的参数 */
    ngx_str_t               conf_prefix;  /* nginx 配置文件的路径 */
    ngx_str_t               prefix;       /* nginx 安装路径 */
    ngx_str_t               lock_file;    /* 加锁文件 */
    ngx_str_t               hostname;     /* 主机名 */
};
```

ngx_init_cycle() 初始化函数

该结构的初始化是通过函数 ngx_init_cycle() 完成的，该函数定义如下：src/core/ngx_cycle.c

ngx_cycle_t 结构全局变量初始化过程如下：

- 更新时区与时间；

- 创建内存池；
- 分配 ngx_cycle_t 结构体内存，创建该结构的变量 cycle 并初始化；
- 遍历所有 core模块，并调用该模块的 create_conf() 函数；
- 配置文件解析；
- 遍历所有core模块，并调用core模块的init_conf()函数；
- 遍历 open_files 链表中的每一个文件并打开；
- 创建新的共享内存并初始化；
- 遍历 listening 数组并打开所有侦听；
- 提交新的 cycle 配置，并调用所有模块的init_module；
- 关闭或删除不被使用的在 old_cycle 中的资源：
- 释放多余的共享内存；
- 关闭多余的侦听 sockets；
- 关闭多余的打开文件；

```
ngx_cycle_t *
ngx_init_cycle(ngx_cycle_t *old_cycle)
{
    void               *rv;
    char              **senv, **env;
    ngx_uint_t          i, n;
    ngx_log_t          *log;
    ngx_time_t         *tp;
    ngx_conf_t          conf;
    ngx_pool_t         *pool;
    ngx_cycle_t        *cycle, **old;
    ngx_shm_zone_t      *shm_zone, *oshm_zone;
    ngx_list_part_t     *part, *opart;
    ngx_open_file_t     *file;
    ngx_listening_t     *ls, *nls;
    ngx_core_conf_t     *ccf, *old_ccf;
    ngx_core_module_t   *module;
    char                hostname[NGX_MAXHOSTNAMELEN];

    /* 更新时区 */
    ngx_timezone_update();

    /* force localtime update with a new timezone */

    tp = ngx_timeofday();
    tp->sec = 0;
```

```
    /* 更新时间 */
    ngx_time_update();

    /* 创建内存池pool，并初始化 */
    log = old_cycle->log;

    pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);
    if (pool == NULL) {
        return NULL;
    }
    pool->log = log;

    /* 创建ngx_cycle_t 结构体变量 */
    cycle = ngx_pcalloc(pool, sizeof(ngx_cycle_t));
    if (cycle == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    /* 初始化ngx_cycle_t 结构体变量 cycle */
    cycle->pool = pool;
    cycle->log = log;
    cycle->old_cycle = old_cycle;

    cycle->conf_prefix.len = old_cycle->conf_prefix.len;
    cycle->conf_prefix.data = ngx_pstrdup(pool, &old_cycle->conf_prefix);
    if (cycle->conf_prefix.data == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    cycle->prefix.len = old_cycle->prefix.len;
    cycle->prefix.data = ngx_pstrdup(pool, &old_cycle->prefix);
    if (cycle->prefix.data == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    cycle->conf_file.len = old_cycle->conf_file.len;
    cycle->conf_file.data = ngx_pnalloc(pool, old_cycle->conf_file.len + 1);
    if (cycle->conf_file.data == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }
    ngx_cpystrn(cycle->conf_file.data, old_cycle->conf_file.data,
                old_cycle->conf_file.len + 1);

    cycle->conf_param.len = old_cycle->conf_param.len;
    cycle->conf_param.data = ngx_pstrdup(pool, &old_cycle->conf_param);
    if (cycle->conf_param.data == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }
```

```
    n = old_cycle->paths.nelts ? old_cycle->paths.nelts : 10;

    cycle->paths.elts = ngx_pcalloc(pool, n * sizeof(ngx_path_t *));
    if (cycle->paths.elts == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    cycle->paths.nelts = 0;
    cycle->paths.size = sizeof(ngx_path_t *);
    cycle->paths.nalloc = n;
    cycle->paths.pool = pool;

    if (old_cycle->open_files.part.nelts) {
        n = old_cycle->open_files.part.nelts;
        for (part = old_cycle->open_files.part.next; part; part = part->next) {
            n += part->nelts;
        }

    } else {
        n = 20;
    }

    if (ngx_list_init(&cycle->open_files, pool, n, sizeof(ngx_open_file_t))
        != NGX_OK)
    {
        ngx_destroy_pool(pool);
        return NULL;
    }

    if (old_cycle->shared_memory.part.nelts) {
        n = old_cycle->shared_memory.part.nelts;
        for (part = old_cycle->shared_memory.part.next; part; part = part->next)
        {
            n += part->nelts;
        }

    } else {
        n = 1;
    }

    if (ngx_list_init(&cycle->shared_memory, pool, n, sizeof(ngx_shm_zone_t))
        != NGX_OK)
    {
        ngx_destroy_pool(pool);
        return NULL;
    }

    n = old_cycle->listening.nelts ? old_cycle->listening.nelts : 10;

    cycle->listening.elts = ngx_pcalloc(pool, n * sizeof(ngx_listening_t));
    if (cycle->listening.elts == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }
```

```
    cycle->listening.nelts = 0;
    cycle->listening.size = sizeof(ngx_listening_t);
    cycle->listening.nalloc = n;
    cycle->listening.pool = pool;

    ngx_queue_init(&cycle->reusable_connections_queue);

    cycle->conf_ctx = ngx_pcalloc(pool, ngx_max_module * sizeof(void *));
    if (cycle->conf_ctx == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    if (gethostname(hostname, NGX_MAXHOSTNAMELEN) == -1) {
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "gethostname() failed");
        ngx_destroy_pool(pool);
        return NULL;
    }

    /* on Linux gethostname() silently truncates name that does not fit */

    hostname[NGX_MAXHOSTNAMELEN - 1] = '\0';
    cycle->hostname.len = ngx_strlen(hostname);

    cycle->hostname.data = ngx_pnalloc(pool, cycle->hostname.len);
    if (cycle->hostname.data == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    ngx_strlow(cycle->hostname.data, (u_char *) hostname, cycle->hostname.len);

    /* 遍历所有core模块 */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_CORE_MODULE) {
            continue;
        }

        module = ngx_modules[i]->ctx;

        /* 若有core模块实现了create_conf()，则就调用它，并返回地址 */
        if (module->create_conf) {
            rv = module->create_conf(cycle);
            if (rv == NULL) {
                ngx_destroy_pool(pool);
                return NULL;
            }
            cycle->conf_ctx[ngx_modules[i]->index] = rv;
        }
    }

    senv = environ;

    ngx_memzero(&conf, sizeof(ngx_conf_t));
```

```
    ngx_memzero(&conf, sizeof(ngx_conf_t));
    /* STUB: init array ? */
    conf.args = ngx_array_create(pool, 10, sizeof(ngx_str_t));
    if (conf.args == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    conf.temp_pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);
    if (conf.temp_pool == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }

    conf.ctx = cycle->conf_ctx;
    conf.cycle = cycle;
    conf.pool = pool;
    conf.log = log;
    conf.module_type = NGX_CORE_MODULE;
    conf.cmd_type = NGX_MAIN_CONF;

#if 0
    log->log_level = NGX_LOG_DEBUG_ALL;
#endif

    /* 配置文件解析 */
    if (ngx_conf_param(&conf) != NGX_CONF_OK) {
        environ = senv;
        ngx_destroy_cycle_pools(&conf);
        return NULL;
    }

    if (ngx_conf_parse(&conf, &cycle->conf_file) != NGX_CONF_OK) {
        environ = senv;
        ngx_destroy_cycle_pools(&conf);
        return NULL;
    }

    if (ngx_test_config && !ngx_quiet_mode) {
        ngx_log_stderr(0, "the configuration file %s syntax is ok",
                       cycle->conf_file.data);
    }

    /* 遍历所有core模块，并调用core模块的init_conf()函数 */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_CORE_MODULE) {
            continue;
        }

        module = ngx_modules[i]->ctx;

        if (module->init_conf) {
            if (module->init_conf(cycle, cycle->conf_ctx[ngx_modules[i]->index])
                == NGX_CONF_ERROR)
            {
```

```
            environ = senv;
            ngx_destroy_cycle_pools(&conf);
            return NULL;
        }
    }
}

if (ngx_process == NGX_PROCESS_SIGNALLER) {
    return cycle;
}

ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

if (ngx_test_config) {

    if (ngx_create_pidfile(&ccf->pid, log) != NGX_OK) {
        goto failed;
    }

} else if (!ngx_is_init_cycle(old_cycle)) {

    /*
     * we do not create the pid file in the first ngx_init_cycle() call
     * because we need to write the demonized process pid
     */

    old_ccf = (ngx_core_conf_t *) ngx_get_conf(old_cycle->conf_ctx,
                                               ngx_core_module);
    if (ccf->pid.len != old_ccf->pid.len
        || ngx_strcmp(ccf->pid.data, old_ccf->pid.data) != 0)
    {
        /* new pid file name */

        if (ngx_create_pidfile(&ccf->pid, log) != NGX_OK) {
            goto failed;
        }

        ngx_delete_pidfile(old_cycle);
    }
}

if (ngx_test_lockfile(cycle->lock_file.data, log) != NGX_OK) {
    goto failed;
}

/* 初始化paths 数组 */
if (ngx_create_paths(cycle, ccf->user) != NGX_OK) {
    goto failed;
}

if (ngx_log_open_default(cycle) != NGX_OK) {
    goto failed;
}

/* open the new files */
```

```
    /* open the new files */

    part = &cycle->open_files.part;
    file = part->elts;

    /* 遍历open_file 链表中每一个文件，并打开该文件 */
    for (i = 0; /* void */ ; i++) {

        if (i >= part->nelts) {
            if (part->next == NULL) {
                break;
            }
            part = part->next;
            file = part->elts;
            i = 0;
        }

        if (file[i].name.len == 0) {
            continue;
        }

        file[i].fd = ngx_open_file(file[i].name.data,
                                   NGX_FILE_APPEND,
                                   NGX_FILE_CREATE_OR_OPEN,
                                   NGX_FILE_DEFAULT_ACCESS);

        ngx_log_debug3(NGX_LOG_DEBUG_CORE, log, 0,
                       "log: %p %d \"%s\"",
                       &file[i], file[i].fd, file[i].name.data);

        if (file[i].fd == NGX_INVALID_FILE) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                          ngx_open_file_n " \"%s\" failed",
                          file[i].name.data);
            goto failed;
        }

#if !(NGX_WIN32)
        if (fcntl(file[i].fd, F_SETFD, FD_CLOEXEC) == -1) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                          "fcntl(FD_CLOEXEC) \"%s\" failed",
                          file[i].name.data);
            goto failed;
        }
#endif
    }

    cycle->log = &cycle->new_log;
    pool->log = &cycle->new_log;

    /* 创建共享内存并初始化 */
    /* create shared memory */

    part = &cycle->shared_memory.part;
    shm_zone = part->elts;
```

```
    for (i = 0; /* void */ ; i++) {

        if (i >= part->nelts) {
            if (part->next == NULL) {
                break;
            }
            part = part->next;
            shm_zone = part->elts;
            i = 0;
        }

        if (shm_zone[i].shm.size == 0) {
            ngx_log_error(NGX_LOG_EMERG, log, 0,
                          "zero size shared memory zone \"%V\"",
                          &shm_zone[i].shm.name);
            goto failed;
        }

        shm_zone[i].shm.log = cycle->log;

        opart = &old_cycle->shared_memory.part;
        oshm_zone = opart->elts;

        /*
         * 新的共享内存与旧的共享内存链表进行比较，
         * 相同则保留，不同创建新的，旧的被释放
         */
        for (n = 0; /* void */ ; n++) {

            if (n >= opart->nelts) {
                if (opart->next == NULL) {
                    break;
                }
                opart = opart->next;
                oshm_zone = opart->elts;
                n = 0;
            }

            if (shm_zone[i].shm.name.len != oshm_zone[n].shm.name.len) {
                continue;
            }

            if (ngx_strncmp(shm_zone[i].shm.name.data,
                            oshm_zone[n].shm.name.data,
                            shm_zone[i].shm.name.len)
                != 0)
            {
                continue;
            }

            if (shm_zone[i].tag == oshm_zone[n].tag
                && shm_zone[i].shm.size == oshm_zone[n].shm.size)
            {
                shm_zone[i].shm.addr = oshm_zone[n].shm.addr;
```

```
            shm_zone[i].shm.addr = oshm_zone[n].shm.addr;

            if (shm_zone[i].init(&shm_zone[i], oshm_zone[n].data)
                != NGX_OK)
            {
                goto failed;
            }

            goto shm_zone_found;
        }

        ngx_shm_free(&oshm_zone[n].shm);

        break;
    }

    if (ngx_shm_alloc(&shm_zone[i].shm) != NGX_OK) {
        goto failed;
    }

    if (ngx_init_zone_pool(cycle, &shm_zone[i]) != NGX_OK) {
        goto failed;
    }

    if (shm_zone[i].init(&shm_zone[i], NULL) != NGX_OK) {
        goto failed;
    }

shm_zone_found:

    continue;
}

/* 遍历listening 数组，并打开所有监听 */
/* handle the listening sockets */

if (old_cycle->listening.nelts) {
    ls = old_cycle->listening.elts;
    for (i = 0; i < old_cycle->listening.nelts; i++) {
        ls[i].remain = 0;
    }

    nls = cycle->listening.elts;
    for (n = 0; n < cycle->listening.nelts; n++) {

        for (i = 0; i < old_cycle->listening.nelts; i++) {
            if (ls[i].ignore) {
                continue;
            }

            if (ngx_cmp_sockaddr(nls[n].sockaddr, nls[n].socklen,
                                 ls[i].sockaddr, ls[i].socklen, 1)
                == NGX_OK)
            {
                nls[n].fd = ls[i].fd;
```

```
                nls[n].previous = &ls[i];
                ls[i].remain = 1;

                if (ls[i].backlog != nls[n].backlog) {
                    nls[n].listen = 1;
                }

#if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)

                /*
                 * FreeBSD, except the most recent versions,
                 * could not remove accept filter
                 */
                nls[n].deferred_accept = ls[i].deferred_accept;

                if (ls[i].accept_filter && nls[n].accept_filter) {
                    if (ngx_strcmp(ls[i].accept_filter,
                                   nls[n].accept_filter)
                        != 0)
                    {
                        nls[n].delete_deferred = 1;
                        nls[n].add_deferred = 1;
                    }

                } else if (ls[i].accept_filter) {
                    nls[n].delete_deferred = 1;

                } else if (nls[n].accept_filter) {
                    nls[n].add_deferred = 1;
                }
#endif

#if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)

                if (ls[i].deferred_accept && !nls[n].deferred_accept) {
                    nls[n].delete_deferred = 1;

                } else if (ls[i].deferred_accept != nls[n].deferred_accept)
                {
                    nls[n].add_deferred = 1;
                }
#endif
                break;
            }
        }

        if (nls[n].fd == (ngx_socket_t) -1) {
            nls[n].open = 1;
#if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
            if (nls[n].accept_filter) {
                nls[n].add_deferred = 1;
            }
#endif
#if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
            if (nls[n].deferred_accept) {
```

```
                if (nls[n].deferred_accept) {
                    nls[n].add_deferred = 1;
                }
#endif
            }
        }

    } else {
        ls = cycle->listening.elts;
        for (i = 0; i < cycle->listening.nelts; i++) {
            ls[i].open = 1;
#if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
            if (ls[i].accept_filter) {
                ls[i].add_deferred = 1;
            }
#endif
#if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
            if (ls[i].deferred_accept) {
                ls[i].add_deferred = 1;
            }
#endif
        }
    }

    /* 打开监听 */
    if (ngx_open_listening_sockets(cycle) != NGX_OK) {
        goto failed;
    }

    if (!ngx_test_config) {
        ngx_configure_listening_sockets(cycle);
    }

    /* 提交新的cycle配置，并调用所有模块的 init_module */
    /* commit the new cycle configuration */

    if (!ngx_use_stderr) {
        (void) ngx_log_redirect_stderr(cycle);
    }

    pool->log = cycle->log;

    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->init_module) {
            if (ngx_modules[i]->init_module(cycle) != NGX_OK) {
                /* fatal */
                exit(1);
            }
        }
    }

    /* 关闭或删除不被使用的old_cycle 资源 */
    /* close and delete stuff that lefts from an old cycle */

    /* free the unnecessary shared memory */
```

```
    opart = &old_cycle->shared_memory.part;
    oshm_zone = opart->elts;

    for (i = 0; /* void */ ; i++) {

        if (i >= opart->nelts) {
            if (opart->next == NULL) {
                goto old_shm_zone_done;
            }
            opart = opart->next;
            oshm_zone = opart->elts;
            i = 0;
        }

        part = &cycle->shared_memory.part;
        shm_zone = part->elts;

        for (n = 0; /* void */ ; n++) {

            if (n >= part->nelts) {
                if (part->next == NULL) {
                    break;
                }
                part = part->next;
                shm_zone = part->elts;
                n = 0;
            }

            if (oshm_zone[i].shm.name.len == shm_zone[n].shm.name.len
                && ngx_strncmp(oshm_zone[i].shm.name.data,
                               shm_zone[n].shm.name.data,
                               oshm_zone[i].shm.name.len)
                == 0)
            {
                goto live_shm_zone;
            }
        }

        ngx_shm_free(&oshm_zone[i].shm);

    live_shm_zone:

        continue;
    }

old_shm_zone_done:

    /* close the unnecessary listening sockets */

    ls = old_cycle->listening.elts;
    for (i = 0; i < old_cycle->listening.nelts; i++) {

        if (ls[i].remain || ls[i].fd == (ngx_socket_t) -1) {
            continue;
```

```
        }

        if (ngx_close_socket(ls[i].fd) == -1) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
                          ngx_close_socket_n " listening socket on %V failed",
                          &ls[i].addr_text);
        }

#if (NGX_HAVE_UNIX_DOMAIN)

        if (ls[i].sockaddr->sa_family == AF_UNIX) {
            u_char  *name;

            name = ls[i].addr_text.data + sizeof("unix:") - 1;

            ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
                          "deleting socket %s", name);

            if (ngx_delete_file(name) == NGX_FILE_ERROR) {
                ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
                              ngx_delete_file_n " %s failed", name);
            }
        }

#endif
    }

    /* close the unnecessary open files */

    part = &old_cycle->open_files.part;
    file = part->elts;

    for (i = 0; /* void */ ; i++) {

        if (i >= part->nelts) {
            if (part->next == NULL) {
                break;
            }
            part = part->next;
            file = part->elts;
            i = 0;
        }

        if (file[i].fd == NGX_INVALID_FILE || file[i].fd == ngx_stderr) {
            continue;
        }

        if (ngx_close_file(file[i].fd) == NGX_FILE_ERROR) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                          ngx_close_file_n " \"%s\" failed",
                          file[i].name.data);
        }
    }
```

```
    ngx_destroy_pool(conf.temp_pool);

    if (ngx_process == NGX_PROCESS_MASTER || ngx_is_init_cycle(old_cycle)) {

        /*
         * perl_destruct() frees environ, if it is not the same as it was at
         * perl_construct() time, therefore we save the previous cycle
         * environment before ngx_conf_parse() where it will be changed.
         */

        env = environ;
        environ = senv;

        ngx_destroy_pool(old_cycle->pool);
        cycle->old_cycle = NULL;

        environ = env;

        return cycle;
    }

    if (ngx_temp_pool == NULL) {
        ngx_temp_pool = ngx_create_pool(128, cycle->log);
        if (ngx_temp_pool == NULL) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
                          "could not create ngx_temp_pool");
            exit(1);
        }

        n = 10;
        ngx_old_cycles.elts = ngx_pcalloc(ngx_temp_pool,
                                          n * sizeof(ngx_cycle_t *));
        if (ngx_old_cycles.elts == NULL) {
            exit(1);
        }
        ngx_old_cycles.nelts = 0;
        ngx_old_cycles.size = sizeof(ngx_cycle_t *);
        ngx_old_cycles.nalloc = n;
        ngx_old_cycles.pool = ngx_temp_pool;

        ngx_cleaner_event.handler = ngx_clean_old_cycles;
        ngx_cleaner_event.log = cycle->log;
        ngx_cleaner_event.data = &dumb;
        dumb.fd = (ngx_socket_t) -1;
    }

    ngx_temp_pool->log = cycle->log;

    old = ngx_array_push(&ngx_old_cycles);
    if (old == NULL) {
        exit(1);
    }
    *old = old_cycle;

    if (!ngx_cleaner_event.timer_set) {
```

```
        ngx_add_timer(&ngx_cleaner_event, 30000);
        ngx_cleaner_event.timer_set = 1;
    }

    return cycle;

failed:

    if (!ngx_is_init_cycle(old_cycle)) {
        old_ccf = (ngx_core_conf_t *) ngx_get_conf(old_cycle->conf_ctx,
                                                   ngx_core_module);
        if (old_ccf->environment) {
            environ = old_ccf->environment;
        }
    }

    /* rollback the new cycle configuration */

    part = &cycle->open_files.part;
    file = part->elts;

    for (i = 0; /* void */ ; i++) {

        if (i >= part->nelts) {
            if (part->next == NULL) {
                break;
            }
            part = part->next;
            file = part->elts;
            i = 0;
        }

        if (file[i].fd == NGX_INVALID_FILE || file[i].fd == ngx_stderr) {
            continue;
        }

        if (ngx_close_file(file[i].fd) == NGX_FILE_ERROR) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                          ngx_close_file_n " \"%s\" failed",
                          file[i].name.data);
        }
    }

    if (ngx_test_config) {
        ngx_destroy_cycle_pools(&conf);
        return NULL;
    }

    ls = cycle->listening.elts;
    for (i = 0; i < cycle->listening.nelts; i++) {
        if (ls[i].fd == (ngx_socket_t) -1 || !ls[i].open) {
            continue;
        }
```

```
        if (ngx_close_socket(ls[i].fd) == -1) {
            ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
                          ngx_close_socket_n " %V failed",
                          &ls[i].addr_text);
        }
    }

    ngx_destroy_cycle_pools(&conf);

    return NULL;
}
```

参考资料：

《 nginx源码分析—启动流程》

《Nginx启动初始化过程》

Nginx 配置解析

概述

在上一篇文章《 Nginx 启动初始化过程》简单介绍了 Nginx 启动的过程，并分析了其启动过程的源码。在启动过程中有一个步骤非常重要，就是调用函数 ngx_init_cycle()，该函数的调用为配置解析提供了接口。配置解析接口大概可分为两个阶段：准备数据阶段和配置解析阶段；

准备数据阶段包括：

- 准备内存；
- 准备错误日志；
- 准备所需数据结构；

配置解析阶段是调用函数：

```
/* 配置文件解析 */
if (ngx_conf_param(&conf) != NGX_CONF_OK) {/* 带有命令行参数'-g' 加入的配置 */
    environ = senv;
    ngx_destroy_cycle_pools(&conf);
    return NULL;
}

if (ngx_conf_parse(&conf, &cycle->conf_file) != NGX_CONF_OK) {/* 解析配置文件*/
    environ = senv;
    ngx_destroy_cycle_pools(&conf);
    return NULL;
}
```

配置解析

ngx_conf_t 结构体

该结构体用于 Nginx 在解析配置文件时描述每个指令的属性，也是Nginx 程序中非常重要的一个数据结构，其定义于文件：src/core/ngx_conf_file.h

```
/* 解析配置时所使用的结构体 */
struct ngx_conf_s {
    char               *name;     /* 当前解析到的指令 */
    ngx_array_t        *args;     /* 当前指令所包含的所有参数 */

    ngx_cycle_t        *cycle;    /* 待解析的全局变量ngx_cycle_t */
    ngx_pool_t         *pool;     /* 内存池 */
    ngx_pool_t         *temp_pool;/* 临时内存池，分配一些临时数组或变量 */
    ngx_conf_file_t     *conf_file;/* 待解析的配置文件 */
    ngx_log_t          *log;      /* 日志信息 */

    void               *ctx;      /* 描述指令的上下文 */
    ngx_uint_t          module_type;/* 当前解析的指令的模块类型 */
    ngx_uint_t          cmd_type; /* 当前解析的指令的指令类型 */

    ngx_conf_handler_pt   handler;  /* 模块自定义的handler，即指令自定义的处理函数 */
    char               *handler_conf;/* 自定义处理函数需要的相关配置 */
};
```

配置文件信息 conf_file

conf_file 是存放 Nginx 配置文件的相关信息。ngx_conf_file_t 结构体的定义如下：

```
typedef struct {
    ngx_file_t          file;     /* 文件的属性 */
    ngx_buf_t          *buffer;   /* 文件的内容 */
    ngx_uint_t          line;     /* 文件的行数 */
} ngx_conf_file_t;
```

配置上下文 ctx

Nginx 的配置文件是分块配置的，常见的有http 块、server 块、location 块以及upsteam 块和 mail 块等。每一个这样的配置块代表一个作用域。高一级配置块的作用域包含了多个低一级配置块的作用域，也就是有作用域嵌套的现象。这样，配置文件中的许多指令都会同时包含在多个作用域内。比如，http 块中的指令都可能同时处于http 块、server 块和location 块等三层作用域内。

在 Nginx 程序解析配置文件时，每一条指令都应该记录自己所属的作用域范围，而配置文件上下文 ctx 变量就是用来存放当前指令所属的作用域的。在Nginx 配置文件的各种配置块中，http 块可以包含子配置块，这在存储结构上比较复杂。

指令类型 type

Nginx 程序中的不同的指令类型以宏的形式定义在不同的源码头文件中，指令类型是core 模块类型的定义在文件：src/core/ngx_conf_file.h

```
#define NGX_DIRECT_CONF        0x00010000
#define NGX_MAIN_CONF          0x01000000
#define NGX_ANY_CONF           0x0F000000
```

这些是 core 类型模块支持的指令类型。其中的 NGX_DIRECT_CONF类指令在 Nginx 程序进入配置解析函数之前已经初始化完成，所以在进入配置解析函数之后可以将它们直接解析并存储到实际的数据结构中，从配置文件的结构上来看，它们一般指的就是那些游离于配置块之外、处于配置文件全局块部分的指令。NGX_MAIN_CONF 类指令包括event、http、mail、upstream 等可以形成配置块的指令，它们没有自己的初始化函数。Nginx 程序在解析配置文件时如果遇到 NGX_MAIN_CONF 类指令，将转入对下一级指令的解析。

以下是 event 类型模块支持的指令类型。

```
#define NGX_EVENT_CONF         0x02000000
```

以下是 http 类型模块支持的指令类型，其定义在文件： src/http/ngx_http_config.h

```
#define NGX_HTTP_MAIN_CONF      0x02000000
#define NGX_HTTP_SRV_CONF       0x04000000
#define NGX_HTTP_LOC_CONF       0x08000000
#define NGX_HTTP_UPS_CONF       0x10000000
#define NGX_HTTP_SIF_CONF       0x20000000
#define NGX_HTTP_LIF_CONF       0x40000000
#define NGX_HTTP_LMT_CONF       0x80000000
```

通用模块配置解析

配置解析模块在 src/core/ngx_conf_file.c 中实现。模块提供的接口函数主要是ngx_conf_parse。另外，模块提供另一个单独的接口ngx_conf_param，用来解析命令行传递的配置，这个接口也是对 ngx_conf_parse 的包装。首先看下配置解析函数 ngx_conf_parse，其定义如下：

```
/*
 * 函数功能：配置文件解析；
 * 支持三种不同的解析类型：
 * 1、解析配置文件；
 * 2、解析block块设置；
 * 3、解析命令行配置；
 */
char *
ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename)
{
    char            *rv;
    ngx_fd_t         fd;
    ngx_int_t        rc;
    ngx_buf_t        buf;
    ngx_conf_file_t  *prev, conf_file;
    enum {
```

```
        parse_file = 0,
        parse_block,
        parse_param
    } type;

#if (NGX_SUPPRESS_WARN)
    fd = NGX_INVALID_FILE;
    prev = NULL;
#endif

    if (filename) {/* 若解析的是配置文件 */

        /* open configuration file */

        /* 打开配置文件 */
        fd = ngx_open_file(filename->data, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);
        if (fd == NGX_INVALID_FILE) {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
                               ngx_open_file_n " \"%s\" failed",
                               filename->data);
            return NGX_CONF_ERROR;
        }

        prev = cf->conf_file;

        cf->conf_file = &conf_file;

        if (ngx_fd_info(fd, &cf->conf_file->file.info) == NGX_FILE_ERROR) {
            ngx_log_error(NGX_LOG_EMERG, cf->log, ngx_errno,
                          ngx_fd_info_n " \"%s\" failed", filename->data);
        }

        cf->conf_file->buffer = &buf;

        buf.start = ngx_alloc(NGX_CONF_BUFFER, cf->log);
        if (buf.start == NULL) {
            goto failed;
        }

        buf.pos = buf.start;
        buf.last = buf.start;
        buf.end = buf.last + NGX_CONF_BUFFER;
        buf.temporary = 1;

        /* 复制文件属性及文件内容 */
        cf->conf_file->file.fd = fd;
        cf->conf_file->file.name.len = filename->len;
        cf->conf_file->file.name.data = filename->data;
        cf->conf_file->file.offset = 0;
        cf->conf_file->file.log = cf->log;
        cf->conf_file->line = 1;

        type = parse_file;  /* 解析的类型是配置文件 */

    } else if (cf->conf_file->file.fd != NGX_INVALID_FILE) {
```

```
    } else if (cf->conf_file->file.fd != NGX_INVALID_FILE) {

        type = parse_block; /* 解析的类型是block块 */

    } else {
        type = parse_param; /* 解析的类型是命令行配置 */
    }

    for ( ;; ) {
        /* 语法分析函数 */
        rc = ngx_conf_read_token(cf);

        /*
         * ngx_conf_read_token() may return
         *
         *    NGX_ERROR             there is error
         *    NGX_OK                the token terminated by ";" was found
         *    NGX_CONF_BLOCK_START  the token terminated by "{" was found
         *    NGX_CONF_BLOCK_DONE   the "}" was found
         *    NGX_CONF_FILE_DONE    the configuration file is done
         */

        if (rc == NGX_ERROR) {
            goto done;
        }

        /* 解析block块设置 */
        if (rc == NGX_CONF_BLOCK_DONE) {

            if (type != parse_block) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "unexpected \"}\"");
                goto failed;
            }

            goto done;
        }

        /* 解析配置文件 */
        if (rc == NGX_CONF_FILE_DONE) {

            if (type == parse_block) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                                   "unexpected end of file, expecting \"}\"");
                goto failed;
            }

            goto done;
        }

        if (rc == NGX_CONF_BLOCK_START) {

            if (type == parse_param) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                                   "block directives are not supported "
                                   "in -g option");
```

```
                goto failed;
            }
        }

        /* rc == NGX_OK || rc == NGX_CONF_BLOCK_START */

        /* 自定义指令处理函数 */
        if (cf->handler) {

            /*
             * the custom handler, i.e., that is used in the http's
             * "types { ... }" directive
             */

            if (rc == NGX_CONF_BLOCK_START) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "unexpected \"{\"");
                goto failed;
            }

            /* 命令行配置处理函数 */
            rv = (*cf->handler)(cf, NULL, cf->handler_conf);
            if (rv == NGX_CONF_OK) {
                continue;
            }

            if (rv == NGX_CONF_ERROR) {
                goto failed;
            }

            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, rv);

            goto failed;
        }

        /* 若自定义指令处理函数handler为NULL，则调用Nginx内建的指令解析机制 */
        rc = ngx_conf_handler(cf, rc);

        if (rc == NGX_ERROR) {
            goto failed;
        }
    }

failed:

    rc = NGX_ERROR;

done:

    if (filename) {/* 若是配置文件 */
        if (cf->conf_file->buffer->start) {
            ngx_free(cf->conf_file->buffer->start);
        }

        if (ngx_close_file(fd) == NGX_FILE_ERROR) {
            ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno
```

```
                ngx_close_file_n " %s failed",
                filename->data);
            return NGX_CONF_ERROR;
        }

        cf->conf_file = prev;
    }

    if (rc == NGX_ERROR) {
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;
}
```

从配置解析函数的源码可以看出，该函数分为两个阶段： 语法分析和 指令解析。语法分析由ngx_conf_read_token()函数完成。指令解析有两种方式：一种是Nginx 内建的指令解析机制；另一种是自定义的指令解析机制。自定义指令解析源码如下所示：

```
        /* 自定义指令处理函数 */
        if (cf->handler) {

            /*
             * the custom handler, i.e., that is used in the http's
             * "types { ... }" directive
             */

            if (rc == NGX_CONF_BLOCK_START) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "unexpected \"{\"");
                goto failed;
            }

            /* 命令行配置处理函数 */
            rv = (*cf->handler)(cf, NULL, cf->handler_conf);
            if (rv == NGX_CONF_OK) {
                continue;
            }

            if (rv == NGX_CONF_ERROR) {
                goto failed;
            }

            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, rv);

            goto failed;
        }
```

而Nginx 内置解析机制有函数ngx_conf_handler() 实现。其定义如下：

```
/* Nginx内建的指令解析机制 */
static ngx_int_t
```

```
ngx_conf_handler(ngx_conf_t *cf, ngx_int_t last)
{
    char          *rv;
    void          *conf, **confp;
    ngx_uint_t     i, found;
    ngx_str_t     *name;
    ngx_command_t  *cmd;

    name = cf->args->elts;

    found = 0;

    for (i = 0; ngx_modules[i]; i++) {

        cmd = ngx_modules[i]->commands;
        if (cmd == NULL) {
            continue;
        }

        for ( /* void */ ; cmd->name.len; cmd++) {

            if (name->len != cmd->name.len) {
                continue;
            }

            if (ngx_strcmp(name->data, cmd->name.data) != 0) {
                continue;
            }

            found = 1;

            /*
             * 只处理模块类型为NGX_CONF_MODULE 或是当前正在处理的模块类型；
             */
            if (ngx_modules[i]->type != NGX_CONF_MODULE
                && ngx_modules[i]->type != cf->module_type)
            {
                continue;
            }

            /* is the directive's location right ? */

            if (!(cmd->type & cf->cmd_type)) {
                continue;
            }

            /* 非block块指令必须以";"分号结尾，否则出错返回 */
            if (!(cmd->type & NGX_CONF_BLOCK) && last != NGX_OK) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                                   "directive \"%s\" is not terminated by \";\"",
                                   name->data);
                return NGX_ERROR;
            }
```

```
        /* block块指令必须后接"{"大括号，否则出粗返回 */
        if ((cmd->type & NGX_CONF_BLOCK) && last != NGX_CONF_BLOCK_START) {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "directive \"%s\" has no opening \"{\"",
                               name->data);
            return NGX_ERROR;
        }

        /* is the directive's argument count right ? */

        /* 验证指令参数个数是否正确 */
        if (!(cmd->type & NGX_CONF_ANY)) {

            /* 指令携带的参数只能是 1 个，且其参数值只能是 on 或 off */
            if (cmd->type & NGX_CONF_FLAG) {

                if (cf->args->nelts != 2) {
                    goto invalid;
                }

            } else if (cmd->type & NGX_CONF_1MORE) {/* 指令携带的参数必须超过 1 个 */

                if (cf->args->nelts < 2) {
                    goto invalid;
                }

            } else if (cmd->type & NGX_CONF_2MORE) {/* 指令携带的参数必须超过 2 个 */

                if (cf->args->nelts < 3) {
                    goto invalid;
                }

            } else if (cf->args->nelts > NGX_CONF_MAX_ARGS) {

                goto invalid;

            } else if (!(cmd->type & argument_number[cf->args->nelts - 1]))
            {
                goto invalid;
            }
        }

        /* set up the directive's configuration context */

        conf = NULL;

        if (cmd->type & NGX_DIRECT_CONF) {/* 在core模块使用 */
            conf = ((void **) cf->ctx)[ngx_modules[i]->index];

        } else if (cmd->type & NGX_MAIN_CONF) {/* 指令配置项出现在全局配置中，不属于任何{}配置块 *
/
            conf = &(((void **) cf->ctx)[ngx_modules[i]->index]);

        } else if (cf->ctx) {/* 除了core模块，其他模块都是用该项 */
            confp = *(void **) ((char *) cf->ctx + cmd->conf);
```

```
            if (confp) {
                conf = confp[ngx_modules[i]->ctx_index];
            }
        }

        /* 执行指令解析回调函数 */
        rv = cmd->set(cf, cmd, conf);

        if (rv == NGX_CONF_OK) {
            return NGX_OK;
        }

        if (rv == NGX_CONF_ERROR) {
            return NGX_ERROR;
        }

        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                           "\"%s\" directive %s", name->data, rv);

        return NGX_ERROR;
      }
    }

    if (found) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                           "\"%s\" directive is not allowed here", name->data);

        return NGX_ERROR;
    }

    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                       "unknown directive \"%s\"", name->data);

    return NGX_ERROR;

invalid:

    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                       "invalid number of arguments in \"%s\" directive",
                       name->data);

    return NGX_ERROR;
}
```

HTTP 模块配置解析

这里主要是结构体 *ngx_command_t* ，我们在文章 《Nginx 模块开发》 对该结构体作了介绍，其定义如下：

```
struct ngx_command_s {
    /* 配置项名称 */
    ngx_str_t             name;
    /* 配置项类型，type将指定配置项可以出现的位置以及携带参数的个数 */
    ngx_uint_t            type;
    /* 处理配置项的参数 */
    char               *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    /* 在配置文件中的偏移量，conf与offset配合使用 */
    ngx_uint_t            conf;
    ngx_uint_t            offset;
    /* 配置项读取后的处理方法，必须指向ngx_conf_post_t 结构 */
    void                  *post;
};
```

若在上面的通用配置解析中，定义了如下的 http 配置项结构，则回调用http 配置项，并对该http 配置项进行解析。此时，解析的是http block 块设置。

```
static ngx_command_t  ngx_http_commands[] = {

    { ngx_string("http"),
      NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
      ngx_http_block,
      0,
      0,
      NULL },

      ngx_null_command
};
```

http 是作为一个 core 模块被 nginx 通用解析过程解析的，其核心就是http{} 块指令回调，它完成了http 解析的整个功能，从初始化到计算配置结果。http{} 块指令的流程是：

- 创建并初始化上下文结构；
- 调用通用模块配置解析流程解析；
- 根据解析结果进行配置项合并处理；

创建并初始化上下文结构

当 Nginx 检查到 http{...} 配置项时，HTTP 配置模型就会启动，则会建立一个 *ngx_http_conf_ctx_t* 结构，该结构定义在文件中：src/http/ngx_http_config.h

```
typedef struct{
      /*  指针数组，数组中的每个元素指向所有 HTTP 模块 create_main_conf 方法产生的结构体 */
   void **main_conf;
   /*  指针数组，数组中的每个元素指向所有 HTTP 模块 create_srv_conf 方法产生的结构体 */
   void **srv_conf;
   /*  指针数组，数组中的每个元素指向所有 HTTP 模块 create_loc_conf 方法产生的结构体 */
   void **loc_conf;
}ngx_http_conf_ctx_t;
```

此时，HTTP 框架为所有 HTTP 模块建立 3 个数组，分别存放所有 HTTP 模块的*create_main_conf*、*create_srv_conf* 、*create_loc_conf* 方法返回的地址指针。*ngx_http_conf_ctx_t* 结构的三个成员分别指向这 3 个数组。例如下面的例子是设置 *create_main_conf*、*create_srv_conf* 、*create_loc_conf* 返回的地址。

```
ngx_http_conf_ctx *ctx;
/* HTTP 框架生成 1 个 ngx_http_conf_ctx_t 结构变量 */
ctx = ngx_pcalloc(cf->pool, sizeof(ngx_http_conf_ctx_t));

*(ngx_http_conf_ctx_t **) conf = ctx;


...
/* 分别生成 3 个数组存储所有的 HTTP 模块的 create_main_conf、create_srv_conf、create_loc_conf 方法返回的地址 */
ctx->main_conf = ngx_pcalloc(cf->pool,
                             sizeof(void *) * ngx_http_max_module);

ctx->srv_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);

ctx->loc_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);

/* 遍历所有 HTTP 模块 */
for (m = 0; ngx_modules[m]; m++) {
    if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
        continue;
    }

    module = ngx_modules[m]->ctx;
    mi = ngx_modules[m]->ctx_index;

    /* 若实现了create_main_conf 方法，则调用该方法，并把返回的地址存储到 main_conf 中 */
    if (module->create_main_conf) {
        ctx->main_conf[mi] = module->create_main_conf(cf);
    }
    /* 若实现了create_srv_conf 方法，则调用该方法，并把返回的地址存储到 srv_conf 中 */
    if (module->create_srv_conf) {
        ctx->srv_conf[mi] = module->create_srv_conf(cf);
    }
    /* 若实现了create_loc_conf 方法，则调用该方法，并把返回的地址存储到 loc_conf 中 */
    if (module->create_loc_conf) {
        ctx->loc_conf[mi] = module->create_loc_conf(cf);
    }
```

```
    }

    pcf = *cf;
    cf->ctx = ctx;

    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->preconfiguration) {
            if (module->preconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

调用通用模块配置解析流程解析

从源码 src/http/ngx_http.c 中可以看到，http 块的配置解析是调用通用模块的配置解析函数，其实现如下：

```
    /* 调用通用模块配置解析 */
    /* parse inside the http{} block */

    cf->module_type = NGX_HTTP_MODULE;
    cf->cmd_type = NGX_HTTP_MAIN_CONF;
    rv = ngx_conf_parse(cf, NULL);

    if (rv != NGX_CONF_OK) {
        goto failed;
    }
```

根据解析结果进行配置项合并处理

```
    /* 根据解析结构进行合并处理 */
    /*
     * init http{} main_conf's, merge the server{}s' srv_conf's
     * and its location{}s' loc_conf's
     */

    cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];
    cscfp = cmcf->servers.elts;

    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }
```

```
        module = ngx_modules[m]->ctx;
        mi = ngx_modules[m]->ctx_index;

        /* init http{} main_conf's */

        if (module->init_main_conf) {
            rv = module->init_main_conf(cf, ctx->main_conf[mi]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }

        rv = ngx_http_merge_servers(cf, cmcf, module, mi);
        if (rv != NGX_CONF_OK) {
            goto failed;
        }
    }

    /* create location trees */

    for (s = 0; s < cmcf->servers.nelts; s++) {

        clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];

        if (ngx_http_init_locations(cf, cscfp[s], clcf) != NGX_OK) {
            return NGX_CONF_ERROR;
        }

        if (ngx_http_init_static_location_trees(cf, clcf) != NGX_OK) {
            return NGX_CONF_ERROR;
        }
    }

    if (ngx_http_init_phases(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    if (ngx_http_init_headers_in_hash(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->postconfiguration) {
            if (module->postconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

```
    if (ngx_http_variables_init_vars(cf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /*
     * http{}'s cf->ctx was needed while the configuration merging
     * and in postconfiguration process
     */

    *cf = pcf;

    if (ngx_http_init_phase_handlers(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /* optimize the lists of ports, addresses and server names */

    if (ngx_http_optimize_servers(cf, cmcf, cmcf->ports) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;

failed:

    *cf = pcf;

    return rv;
```

HTTP 配置解析流程

从上面的分析中可以总结出 HTTP 配置解析的流程如下：

- Nginx 进程进入主循环，在主循环中调用配置解析器解析配置文件*nginx.conf*;
- 在配置文件中遇到 http{} 块配置，则 HTTP 框架开始初始化并启动，其由函数 ngx_http_block() 实现；
- HTTP 框架初始化所有 HTTP 模块的序列号，并创建 3 个类型为 *ngx_http_conf_ctx_t *结构的数组用于存储所有HTTP 模块的*create_main_conf、create_srv_conf、create_loc_conf*方法返回的指针地址；
- 调用每个 HTTP 模块的 preconfiguration 方法；
- HTTP 框架调用函数 ngx_conf_parse() 开始循环解析配置文件 *nginx.conf *中的http{}块里面的所有配置项；
- HTTP 框架处理完毕 http{} 配置项，根据解析配置项的结果，必要时进行配置项合并处理；
- 继续处理其他 http{} 块之外的配置项，直到配置文件解析器处理完所有配置项后通知Nginx 主循环配置项解析完毕。此时，Nginx 才会启动Web 服务器；

合并配置项

HTTP 框架解析完毕 http{} 块配置项时，会根据解析的结果进行合并配置项操作，即合并 http{}、server{}、location{} 不同块下各HTTP 模块生成的存放配置项的结构体。其合并过程如下所示：

- 若 HTTP 模块实现了 *merge_srv_conf* 方法，则将 http{} 块下 *create_srv_conf* 生成的结构体与遍历每一个 server{}配置块下的结构体进行 *merge_srv_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，则将 http{} 块下 *create_loc_conf* 生成的结构体与嵌套每一个server{} 配置块下的结构体进行 *merge_loc_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，则将server{} 块下 *create_loc_conf* 生成的结构体与嵌套每一个location{}配置块下的结构体进行 *merge_loc_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，则将location{} 块下 *create_loc_conf* 生成的结构体与嵌套每一个location{}配置块下的结构体进行 *merge_loc_conf* 操作；

以下是合并配置项操作的源码实现：

```
/* 合并配置项操作 */
static char *
ngx_http_merge_servers(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf,
    ngx_http_module_t *module, ngx_uint_t ctx_index)
{
    char                        *rv;
    ngx_uint_t                   s;
    ngx_http_conf_ctx_t         *ctx, saved;
    ngx_http_core_loc_conf_t    *clcf;
    ngx_http_core_srv_conf_t   **cscfp;

    cscfp = cmcf->servers.elts;
    ctx = (ngx_http_conf_ctx_t *) cf->ctx;
    saved = *ctx;
    rv = NGX_CONF_OK;

    /* 遍历每一个server{}块 */
    for (s = 0; s < cmcf->servers.nelts; s++) {

        /* merge the server{}s' srv_conf's */

        ctx->srv_conf = cscfp[s]->ctx->srv_conf;

        /*
         * 若定义了merge_srv_conf 方法；
         * 则进行http{}块下create_srv_conf 生成的结构体与遍历server{}块配置项生成的结构体进行merge_srv_
conf操作；
         */
        if (module->merge_srv_conf) {
            rv = module->merge_srv_conf(cf, saved.srv_conf[ctx_index],
                                        cscfp[s]->ctx->srv_conf[ctx_index]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }
```

```
        }

        /*
         * 若定义了merge_loc_conf 方法；
         * 则进行http{}块下create_loc_conf 生成的结构体与嵌套server{}块配置项生成的结构体进行merge_loc_
conf操作；
         */
        if (module->merge_loc_conf) {

            /* merge the server{}'s loc_conf */

            ctx->loc_conf = cscfp[s]->ctx->loc_conf;

            rv = module->merge_loc_conf(cf, saved.loc_conf[ctx_index],
                                        cscfp[s]->ctx->loc_conf[ctx_index]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }

            /* merge the locations{}' loc_conf's */

            /*
             * 若定义了merge_loc_conf 方法；
             * 则进行server{}块下create_loc_conf 生成的结构体与嵌套location{}块配置项生成的结构体进行mer
ge_loc_conf操作；
             */
            clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];

            rv = ngx_http_merge_locations(cf, clcf->locations,
                                          cscfp[s]->ctx->loc_conf,
                                          module, ctx_index);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }
    }

failed:

    *ctx = saved;

    return rv;
}

static char *
ngx_http_merge_locations(ngx_conf_t *cf, ngx_queue_t *locations,
    void **loc_conf, ngx_http_module_t *module, ngx_uint_t ctx_index)
{
    char                      *rv;
    ngx_queue_t               *q;
    ngx_http_conf_ctx_t       *ctx, saved;
    ngx_http_core_loc_conf_t  *clcf;
    ngx_http_location_queue_t  *lq;

    if (locations == NULL) {
```

```
        return NGX_CONF_OK;
    }

    ctx = (ngx_http_conf_ctx_t *) cf->ctx;
    saved = *ctx;

    /*
     * 若定义了merge_loc_conf 方法；
     * 则进行location{}块下create_loc_conf 生成的结构体与嵌套location{}块配置项生成的结构体进行merge_l
oc_conf操作；
     */
    for (q = ngx_queue_head(locations);
         q != ngx_queue_sentinel(locations);
         q = ngx_queue_next(q))
    {
        lq = (ngx_http_location_queue_t *) q;

        clcf = lq->exact ? lq->exact : lq->inclusive;
        ctx->loc_conf = clcf->loc_conf;

        rv = module->merge_loc_conf(cf, loc_conf[ctx_index],
                                    clcf->loc_conf[ctx_index]);
        if (rv != NGX_CONF_OK) {
            return rv;
        }

        /*
         * 递归调用该函数；
         * 因为location{}继续内嵌location{}
         */
        rv = ngx_http_merge_locations(cf, clcf->locations, clcf->loc_conf,
                                      module, ctx_index);
        if (rv != NGX_CONF_OK) {
            return rv;
        }
    }

    *ctx = saved;

    return NGX_CONF_OK;
}
```

处理自定义的配置

在文章中 《 Nginx 模块开发》，我们给出了“Hello World” 的开发例子，在这个开发例子中，我们定义了自己的配置项，配置项名称的结构体定义如下：

```
typedef struct
{
        ngx_str_t hello_string;
        ngx_int_t hello_counter;
}ngx_http_hello_loc_conf_t;
```

为了处理我们定义的配置项结构，因此，我们把 *ngx_command_t* 结构体定义如下：

```
static ngx_command_t ngx_http_hello_commands[] = {
  {
                ngx_string("hello_string"),
                NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS|NGX_CONF_TAKE1,
                ngx_http_hello_string,
                NGX_HTTP_LOC_CONF_OFFSET,
                offsetof(ngx_http_hello_loc_conf_t, hello_string),
                NULL },

        {
                ngx_string("hello_counter"),
                NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
                ngx_http_hello_counter,
                NGX_HTTP_LOC_CONF_OFFSET,
                offsetof(ngx_http_hello_loc_conf_t, hello_counter),
                NULL },

        ngx_null_command
};
```

处理方法 *ngx_http_hello_string* 和*ngx_http_hello_counter* 定义如下：

```
static char *
ngx_http_hello_string(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{

        ngx_http_hello_loc_conf_t* local_conf;


        local_conf = conf;
        char* rv = ngx_conf_set_str_slot(cf, cmd, conf);

        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_string:%s", local_conf->hello_string.data);

        return rv;
}


static char *ngx_http_hello_counter(ngx_conf_t *cf, ngx_command_t *cmd,
        void *conf)
{
        ngx_http_hello_loc_conf_t* local_conf;

        local_conf = conf;

        char* rv = NULL;

        rv = ngx_conf_set_flag_slot(cf, cmd, conf);


        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_counter:%d", local_conf->hello_counter);
        return rv;
}
```

error 日志

Nginx 日志模块为其他模块提供了基本的日志记录功能，日志模块定义如下： src/core/ngx_log.c

```
static ngx_command_t  ngx_errlog_commands[] = {

    {ngx_string("error_log"),
     NGX_MAIN_CONF|NGX_CONF_1MORE,
     ngx_error_log,
     0,
     0,
     NULL},

    ngx_null_command
};

static ngx_core_module_t  ngx_errlog_module_ctx = {
    ngx_string("errlog"),
    NULL,
    NULL
};

ngx_module_t  ngx_errlog_module = {
    NGX_MODULE_V1,
    &ngx_errlog_module_ctx,              /* module context */
    ngx_errlog_commands,                 /* module directives */
    NGX_CORE_MODULE,                     /* module type */
    NULL,                                /* init master */
    NULL,                                /* init module */
    NULL,                                /* init process */
    NULL,                                /* init thread */
    NULL,                                /* exit thread */
    NULL,                                /* exit process */
    NULL,                                /* exit master */
    NGX_MODULE_V1_PADDING
};
```

Nginx 日志模块对于支持可变参数提供了三个接口，这三个接口定义在文件： src/core/ngx_log.h

```
#define ngx_log_error(level, log, ...)                                       \
    if ((log)->log_level >= level) ngx_log_error_core(level, log, __VA_ARGS__)

void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
    const char *fmt, ...);

#define ngx_log_debug(level, log, ...)                                       \
    if ((log)->log_level & level)                                            \
        ngx_log_error_core(NGX_LOG_DEBUG, log, __VA_ARGS__)
```

Nginx 日志模块记录日志的核心功能是由ngx_log_error_core 方法实现，ngx_log_error 和 ngx_log_debug 宏定义只是对其进行简单的封装，一般情况下日志调用只需要这两个宏定义。

ngx_log_error 和 ngx_log_debug 宏定义都包括参数 level、log、err、fmt，下面分别对这些参数进行简单的介绍：

level 参数：对于 ngx_log_error 宏来说，level 表示当前日志的级别，其取值如下所示：

```
/* ngx_log_error中level参数的取值；下面 9 个日志的级别依次从高到低 */
#define NGX_LOG_STDERR            0    /* 最高级别日志，将日志输出到标准错误设备 */
#define NGX_LOG_EMERG             1
#define NGX_LOG_ALERT             2
#define NGX_LOG_CRIT              3
#define NGX_LOG_ERR               4
#define NGX_LOG_WARN              5
#define NGX_LOG_NOTICE            6
#define NGX_LOG_INFO              7
#define NGX_LOG_DEBUG             8    /* 最低级别日志，属于调试级别 */
```

使用 ngx_log_error 宏记录日志时，若传入的level 级别小于或等于log 参数中的日志级别，就会输出日志内容，否则忽略该日志。

在使用 ngx_log_debug 宏时，参数level 不同于ngx_log_error 宏的level 参数，它表达的不是日志级别，而是日志类型。ngx_log_debug 宏记录日志时必须是NGX_LOG_DEBUG 调试级别，这里的level 取值如下：

```
/* ngx_log_debug中level参数的取值 */
#define NGX_LOG_DEBUG_CORE        0x010 /* nginx核心模块的调试日志 */
#define NGX_LOG_DEBUG_ALLOC       0x020 /* nginx在分配内存时使用的调试日志 */
#define NGX_LOG_DEBUG_MUTEX       0x040 /* nginx在使用进程锁时使用的调试日志 */
#define NGX_LOG_DEBUG_EVENT       0x080 /* nginx event模块的调试日志 */
#define NGX_LOG_DEBUG_HTTP        0x100 /* nginx http模块的调试日志 */
#define NGX_LOG_DEBUG_MAIL        0x200 /* nginx mail模块的调试日志 */
#define NGX_LOG_DEBUG_MYSQL       0x400 /* 与MySQL相关的nginx模块所使用的调试日志 */
```

当 HTTP 模块调用ngx_log_debug 宏记录日志时，传入的level 参数是NGX_LOG_DEBUG_HTTP，此时，若log 参数不属于HTTP 模块，若使用event 事件模块的log，则不会输出任何日志。

log 参数：log 参数的结构定义如下：src/core/ngx_log.h；从其结构中可以知道，若只想把相应的信息记录到日志文件中，则不需要关系参数 log 的构造。

```
/* ngx_log_t 结构的定义 */
struct ngx_log_s {
    /* 日志级别或日志类型 */
    ngx_uint_t           log_level;
    /* 日志文件 */
    ngx_open_file_t     *file;

    /* 连接数，不为0时会输出到日志文件中 */
    ngx_atomic_uint_t    connection;

    /* 记录日志时的回调方法，不是DEBUG调试级别才会被调用 */
    ngx_log_handler_pt   handler;
    /* 模块的data */
    void                *data;

    /*
     * we declare "action" as "char *" because the actions are usually
     * the static strings and in the "u_char *" case we have to override
     * their types all the time
     */

    char                *action;

    /* 指向日志链表的下一个日志 */
    ngx_log_t           *next;
};
```

err 参数：err 参数是错误编码，一般是执行系统调用失败后取得的errno 参数。当err 不为 0 时，Nginx 日志模块将会在正常日志输出这个错误编码以及其对应的字符串形成的错误信息。

fmt 参数：fmt 参数类似于C 语言中的printf 函数的输出格式。

参考资料：

《深入理解 Nginx 》

《nginx 启动阶段》

《Nginx高性能Web服务器详解》

Nginx 中的 upstream 与 subrequest 机制

概述

Nginx 提供了两种全异步方式与第三方服务进行通信： upstream和subrequest。upstream 在与第三方服务器交互时（包括建立TCP 连接、发送请求、接收响应、关闭TCP 连接），不会阻塞Nginx 进程处理其他请求。subrequest 只是分解复杂请求的一种设计模式，它可以把原始请求分解为多个子请求，使得诸多请求协同完成一个用户请求，并且每个请求只关注一个功能。subrequest 访问第三方服务最终也是基于 upstream 实现的。

upstream 被定义为访问上游服务器，它把Nginx 定义为反代理服务器，首要功能是透传，其次才是以TCP 获取第三方服务器的内容。Nginx 的HTTP 反向代理模块是基于 upstream 方式实现的。subrequest 是子请求，也就是说subrequest 将会为用户创建子请求，即将一个复杂的请求分解为多个子请求，每个子请求负责一种功能项，而最初的原始请求负责构成并发送响应给用户。当subrequest 访问第三服务时，首先派生出子请求访问上游服务器，父请求在完全取得上游服务器的响应后再决定如何处理来自客户端的请求。

因此，若希望把是第三方服务的内容原封不动地返回给用户时，则使用 upstream 方式。若访问第三方服务是为了获取某些信息，再根据这些信息来构造响应并发给用户，则应使用 subrequest 方式。

upstream 使用方式

upstream 模块不产生自己的内容，而是通过请求后端服务器得到内容。Nginx 内部封装了请求并取得响应内容的整个过程，所以upstream 模块只需要开发若干回调函数，完成构造请求和解析响应等具体的工作。

ngx_http_request_t 结构体

首先了解 upstream 是如何嵌入到一个请求中，这里必须从请求结构体 ngx_http_request_t 入手，在该结构体中具有一个ngx_http_upstream_t 结构体类型的成员upstream。请求结构体 ngx_http_request_t 定义在文件 src/http/ngx_http_request.h 中如下:

```
struct ngx_http_request_s {
    uint32_t                         signature;         /* "HTTP" */

    /* 当前请求所对应的客户端连接 */
    ngx_connection_t                 *connection;

    /*
     * 以下四个成员是保存模块对应的上下文结构指针；
     * ctx 对应的是自定义的上下文结构指针数组，若是HTTP框架，则存储所有HTTP模块上下文结构；
     * main_conf 对应的是main级别配置结构体的指针数组；
     * srv_conf 对应的是srv级别配置结构体的指针数组；
     * loc_conf 对应的是loc级别配置结构体的指针数组；
```

```
     */
    void                  **ctx;
    void                  **main_conf;
    void                  **srv_conf;
    void                  **loc_conf;

    /*
     * 以下两个是处理http请求；
     * 当http头部接收完毕，第一次在业务上处理http请求时，http框架提供的处理方法是ngx_http_process_r
equest;
     * 若该方法无法一次性处理完该请求的全部业务时，当控制权归还给epoll事件模块后，若该请求再次被回调
时，
     * 此时，将通过ngx_http_request_handler方法进行处理，而这个方法中对于可读或可写事件的处理就是由
函数
     * read_event_handler或write_event_handler 来处理请求；
     */
    ngx_http_event_handler_pt        read_event_handler;
    ngx_http_event_handler_pt        write_event_handler;

#if (NGX_HTTP_CACHE)
    ngx_http_cache_t               *cache;
#endif

    /* 若使用upstream机制，则需要以下的结构体 */
    ngx_http_upstream_t            *upstream;
    ngx_array_t                    *upstream_states;
                                   /* of ngx_http_upstream_state_t */

    /* 当前请求的内存池 */
    ngx_pool_t                     *pool;
    /* 主要用于接收http请求头部内容的缓冲区 */
    ngx_buf_t                      *header_in;

    /*
     * 调用函数ngx_http_request_headers 接收并解析http请求头部完毕后，
     * 则把解析完成的每一个http头部加入到结构体headers_in的成员headers链表中，
     * 同时初始化该结构体的其他成员；
     */
    ngx_http_headers_in_t            headers_in;
    /*
     * http模块将待发送的http响应的信息存放在headers_out中，
     * 并期望http框架将headers_out中的成员序列化为http响应包体发送个客户端；
     */
    ngx_http_headers_out_t           headers_out;

    /* 接收请求包体的数据结构 */
    ngx_http_request_body_t        *request_body;

    /* 延迟关闭连接的时间 */
    time_t                         lingering_time;
    /* 当前请求初始化的时间 */
    time_t                         start_sec;
    /* 相对于start_sec的毫秒偏移量 */
    ngx_msec_t                     start_msec;
```

```
    /*
     * 以下的 9 个成员是函数ngx_http_process_request_line在接收、解析http请求行时解析出的信息 */
    ngx_uint_t                 method;       /* 方法名称 */
    ngx_uint_t                 http_version; /* 协议版本 */

    ngx_str_t                  request_line; /* 请求行 */
    ngx_str_t                  uri;          /* 客户请求中的uri */
    ngx_str_t                  args;         /* uri 中的参数 */
    ngx_str_t                  exten;        /* 客户请求的文件扩展名 */
    ngx_str_t                  unparsed_uri; /* 没经过URI 解码的原始请求 */

    ngx_str_t                  method_name;  /* 方法名称字符串 */
    ngx_str_t                  http_protocol;/* 其data成员指向请求中http的起始地址 */

    /*
     * 存储待发送给客户的http响应；
     * out保存着由headers_out序列化后的表示http头部的TCP流；
     * 调用ngx_http_output_filter方法后，out还保存这待发送的http包体；
     */
    ngx_chain_t                *out;
    /*
     * 当前请求可能是用户请求，或是派生的子请求；
     * main标识一序列相关的派生子请求的原始请求；
     * 即通过main与当前请求的地址对比来判断是用户请求还是派生子请求；
     */
    ngx_http_request_t         *main;
    /*
     * 当前请求的父亲请求，但不一定是原始请求 */
    ngx_http_request_t         *parent;
    /* 以下两个是与subrequest子请求相关的功能 */
    ngx_http_postponed_request_t     *postponed;
    ngx_http_post_subrequest_t       *post_subrequest;
    /* 连接子请求的链表 */
    ngx_http_posted_request_t        *posted_requests;

    /*
     * 全局结构体ngx_http_phase_engine_t定义了一个ngx_http_phase_handler_t回调方法的数组；
     * 而这里的phase_handler作为该数组的序列号表示指定数组中的回调方法，相当于数组的下标；
     */
    ngx_int_t                  phase_handler;
    /*
     * 表示NGX_HTTP_CONTENT_PHASE阶段提供给http模块请求的一种方式，它指向http模块实现的请求处
理方法 */
    ngx_http_handler_pt        content_handler;
    /*
     * 在NGX——HTTP_CONTENT_PHASE阶段需要判断请求是否具有访问权限时，
     * 可通过access_code来传递http模块的handler回调方法的返回值来判断，
     * 若为0表示具备权限，否则不具备；
     */
    ngx_uint_t                 access_code;

    ngx_http_variable_value_t  *variables;

#if (NGX_PCRE)
```

```
    ngx_uint_t                  ncaptures;
    int                        *captures;
    u_char                     *captures_data;
#endif

    /* 限制当前请求的发送的速率 */
    size_t                      limit_rate;
    size_t                      limit_rate_after;

    /* http响应的长度，不包括http响应头部 */
    /* used to learn the Apache compatible response length without a header */
    size_t                      header_size;

    /* http请求的长度，包括http请求头部、http请求包体 */
    off_t                       request_length;

    /* 表示错误状态标志 */
    ngx_uint_t                  err_status;

    /* http 连接 */
    ngx_http_connection_t       *http_connection;
#if (NGX_HTTP_SPDY)
    ngx_http_spdy_stream_t      *spdy_stream;
#endif

    /* http日志处理函数 */
    ngx_http_log_handler_pt     log_handler;

    /* 释放资源 */
    ngx_http_cleanup_t          *cleanup;

    /* 以下都是一些标志位 */
    /* 派生子请求 */
    unsigned                    subrequests:8;
    /* 作为原始请求的引用计数，每派生一个子请求，原始请求的成员count会增加1 */
    unsigned                    count:8;
    /* 阻塞标志位，仅用于aio */
    unsigned                    blocked:8;

    /* 标志位：为1表示当前请求是异步IO方式 */
    unsigned                    aio:1;

    unsigned                    http_state:4;

    /* URI with "/." and on Win32 with "//" */
    unsigned                    complex_uri:1;

    /* URI with "%" */
    unsigned                    quoted_uri:1;

    /* URI with "+" */
    unsigned                    plus_in_uri:1;

    /* URI with " " */
    unsigned                    space_in_uri:1;
```

```
    unsigned                  space_in_uri:1;

    unsigned                  invalid_header:1;

    unsigned                  add_uri_to_alias:1;
    unsigned                  valid_location:1;
    unsigned                  valid_unparsed_uri:1;
    /* 标志位：为1表示URI已经被重写 */
    unsigned                  uri_changed:1;
    /* 表示URI被重写的次数 */
    unsigned                  uri_changes:4;

    unsigned                  request_body_in_single_buf:1;
    unsigned                  request_body_in_file_only:1;
    unsigned                  request_body_in_persistent_file:1;
    unsigned                  request_body_in_clean_file:1;
    unsigned                  request_body_file_group_access:1;
    unsigned                  request_body_file_log_level:3;

    /* 决定是否转发响应，若该标志位为1，表示不转发响应，否则转发响应 */
    unsigned                  subrequest_in_memory:1;
    unsigned                  waited:1;

#if (NGX_HTTP_CACHE)
    unsigned                  cached:1;
#endif

#if (NGX_HTTP_GZIP)
    unsigned                  gzip_tested:1;
    unsigned                  gzip_ok:1;
    unsigned                  gzip_vary:1;
#endif

    unsigned                  proxy:1;
    unsigned                  bypass_cache:1;
    unsigned                  no_cache:1;

    /*
     * instead of using the request context data in
     * ngx_http_limit_conn_module and ngx_http_limit_req_module
     * we use the single bits in the request structure
     */
    unsigned                  limit_conn_set:1;
    unsigned                  limit_req_set:1;

#if 0
    unsigned                  cacheable:1;
#endif

    unsigned                  pipeline:1;
    unsigned                  chunked:1;
    unsigned                  header_only:1;
    /* 标志位，为1表示当前请求是keepalive模式请求 */
    unsigned                  keepalive:1;
    /* 延迟关闭标志位，为1表示需要延迟关闭 */
```

```
    unsigned                  lingering_close:1;
    /* 标志位，为1表示正在丢弃HTTP请求中的包体 */
    unsigned                  discard_body:1;
    /* 标志位，为1表示请求的当前状态是在做内部跳转 */
    unsigned                  internal:1;
    unsigned                  error_page:1;
    unsigned                  ignore_content_encoding:1;
    unsigned                  filter_finalize:1;
    unsigned                  post_action:1;
    unsigned                  request_complete:1;
    unsigned                  request_output:1;
    /* 标志位，为1表示待发送的HTTP响应头部已发送给客户端 */
    unsigned                  header_sent:1;
    unsigned                  expect_tested:1;
    unsigned                  root_tested:1;
    unsigned                  done:1;
    unsigned                  logged:1;

    /* 标志位，表示缓冲区是否存在待发送内容 */
    unsigned                  buffered:4;

    unsigned                  main_filter_need_in_memory:1;
    unsigned                  filter_need_in_memory:1;
    unsigned                  filter_need_temporary:1;
    unsigned                  allow_ranges:1;
    unsigned                  single_range:1;

#if (NGX_STAT_STUB)
    unsigned                  stat_reading:1;
    unsigned                  stat_writing:1;
#endif

    /* used to parse HTTP headers */

    /* 当前的解析状态 */
    ngx_uint_t                state;

    ngx_uint_t                header_hash;
    ngx_uint_t                lowcase_index;
    u_char                    lowcase_header[NGX_HTTP_LC_HEADER_LEN];

    u_char                   *header_name_start;
    u_char                   *header_name_end;
    u_char                   *header_start;
    u_char                   *header_end;

    /*
     * a memory that can be reused after parsing a request line
     * via ngx_http_ephemeral_t
     */

    u_char                   *uri_start;
    u_char                   *uri_end;
    u_char                   *uri_ext;
    u_char                   *args_start;
```

```
    u_char                   *args_start;
    u_char                   *request_start;
    u_char                   *request_end;
    u_char                   *method_end;
    u_char                   *schema_start;
    u_char                   *schema_end;
    u_char                   *host_start;
    u_char                   *host_end;
    u_char                   *port_start;
    u_char                   *port_end;

    unsigned                  http_minor:16;
    unsigned                  http_major:16;
};
```

若没有实现 upstream 机制，则请求结构体 ngx_http_request_t 中的upstream成员设置为NULL，否则必须设置该成员。首先看下 HTTP 模块启动upstream 机制的过程：

1. 调用函数 ngx_http_upstream_create 为请求创建upstream；
2. 设置上游服务器的地址；可通过配置文件 nginx.conf 配置好上游服务器地址；也可以通过 ngx_http_request_t 中的成员resolved 设置上游服务器地址；
3. 设置 upstream 的回调方法；
4. 调用函数 ngx_http_upstream_init 启动upstream；

upstream 启动过程如下图所示：

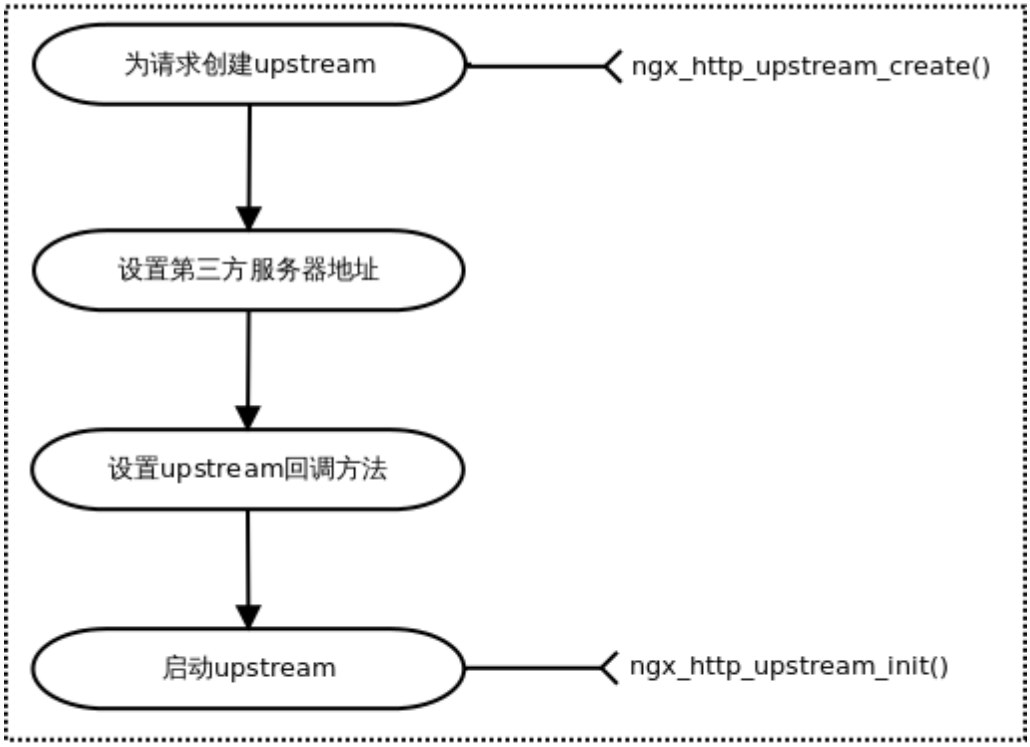


图 1 upstream启动过程

ngx_http_upstream_t 结构体

upstream 结构体是 ngx_http_upstream_t，该结构体只在 upstream 模块内部使用，其定义在文件：src/http/ngx_http_upstream.h

```
/* ngx_http_upstream_t 结构体 */
struct ngx_http_upstream_s {
    /* 处理读事件的回调方法，每一个阶段都有不同的 read_event_handler */
    ngx_http_upstream_handler_pt     read_event_handler;
    /* 处理写事件的回调方法，每一个阶段都有不同的 write_event_handler */
    ngx_http_upstream_handler_pt     write_event_handler;

    /* 表示主动向上游服务器发起的连接 */
    ngx_peer_connection_t            peer;

    /*
     * 当向 下游客户端转发响应时（此时，ngx_http_request_t 结构体中的subrequest_in_memory标志位为0
 ），
     * 若已打开缓存且认为上游网速更快，此时会使用pipe成员来转发响应；
     * 使用这种方式转发响应时，在HTTP模块使用upstream机制前必须构造pipe结构体；
     */
    ngx_event_pipe_t                 *pipe;

    /* 发送给上游服务器的请求，在实现create_request方法时需设置它 */
    ngx_chain_t                      *request_bufs;

    /* 定义了向下游发送响应的方式 */
    ngx_output_chain_ctx_t           output;
    ngx_chain_writer_ctx_t           writer;

    /* 指定upstream机制的运行方式 */
    ngx_http_upstream_conf_t         *conf;

    /*
     * HTTP模块实现process_header方法时，若希望upstream直接转发响应，
     * 则需把解析出来的响应头部适配为HTTP的响应头部，同时需要把包头中的
     * 信息设置到headers_in结构体中
     */
    ngx_http_upstream_headers_in_t   headers_in;

    /* 解析主机域名，用于直接指定的上游服务器地址 */
    ngx_http_upstream_resolved_t     *resolved;

    /* 接收客户信息的缓冲区 */
    ngx_buf_t                        from_client;

    /*
     * 接收上游服务器响应包头的缓冲区，当不直接把响应转发给客户端，
     * 或buffering标志位为0的情况转发包体时，接收包体的缓冲区仍然使用buffer
     */
    ngx_buf_t                        buffer;
    off_t                            length;

    /*
     * out_bufs有两种不同意义：
```

```
     * 1、当不需要转发包体，且默认使用input_filter方法处理包体时，
     *    out_bufs将会指向响应包体，out_bufs链表中产生多个ngx_but_t缓冲区，
     *    每个缓冲区都指向buffer缓存中的一部分，而这里只是调用recv方法接收到的一段TCP流；
     * 2、当需要向下游转发包体时，这个链表指向上一次向下游转发响应到现在这段时间内接收自上游的缓存
响应；
     */
    ngx_chain_t               *out_bufs;
    /*
     * 当需要向下游转发响应包体时，它表示上一次向下游转发响应时没有发送完的内容；
     */
    ngx_chain_t               *busy_bufs;
    /*
     * 这个链表用于回收out_bufs中已经发送给下游的ngx_buf_t结构体；
     */
    ngx_chain_t               *free_bufs;

    /*
     * 处理包体前的初始化方法；
     * 其中data参数用于传递用户数据结构，就是下面成员input_filter_ctx
     */
    ngx_int_t                 (*input_filter_init)(void *data);
    /*
     * 处理包体的方法；
     * 其中data参数用于传递用户数据结构，就是下面成员input_filter_ctx，
     * bytes表示本次接收到包体的长度；
     */
    ngx_int_t                 (*input_filter)(void *data, ssize_t bytes);
    /* 用于传递HTTP自定义的数据结构 */
    void                      *input_filter_ctx;

#if (NGX_HTTP_CACHE)
    ngx_int_t                 (*create_key)(ngx_http_request_t *r);
#endif
    /* HTTP模块实现的create_request方法用于构造发往上游服务器的请求 */
    ngx_int_t                 (*create_request)(ngx_http_request_t *r);
    /* 与上游服务器的通信失败后，若想再次向上游服务器发起连接，则调用该函数 */
    ngx_int_t                 (*reinit_request)(ngx_http_request_t *r);
    /*
     * 解析上游服务器返回的响应包头，该函数返回四个值中的一个：
     * NGX_AGAIN                        表示包头没有接收完整；
     * NGX_HTTP_UPSTREAM_INVALID_HEADER    表示包头不合法；
     * NGX_ERROR                        表示出现错误；
     * NGX_OK                           表示解析到完整的包头；
     */
    ngx_int_t                 (*process_header)(ngx_http_request_t *r);
    /* 当客户端放弃请求时被调用，由于系统会自动关闭连接，因此，该函数不会进行任何具体操作 */
    void                      (*abort_request)(ngx_http_request_t *r);
    /* 结束upstream请求时会调用该函数 */
    void                      (*finalize_request)(ngx_http_request_t *r,
                                  ngx_int_t rc);
    /*
     * 在上游返回的响应出现location或者refresh头部表示重定向时，
     * 会通过ngx_http_upstream_process_headers方法调用到可由HTTP模块
     * 实现的rewrite_redirect方法；
     */
```

```
    ngx_int_t                  (*rewrite_redirect)(ngx_http_request_t *r,
                                ngx_table_elt_t *h, size_t prefix);
    ngx_int_t                  (*rewrite_cookie)(ngx_http_request_t *r,
                                ngx_table_elt_t *h);

    ngx_msec_t                   timeout;

    /* 用于表示上游响应的状态：错误编码、包体长度等信息 */
    ngx_http_upstream_state_t       *state;

    ngx_str_t                  method;
    /* 用于记录日志 */
    ngx_str_t                  schema;
    ngx_str_t                  uri;

    /* 清理资源 */
    ngx_http_cleanup_pt          *cleanup;

    /* 以下是一些标志位 */

    /* 指定文件缓存路径 */
    unsigned                   store:1;
    /* 启用文件缓存 */
    unsigned                   cacheable:1;
    unsigned                   accel:1;
    /* 基于ssl协议访问上游服务器 */
    unsigned                   ssl:1;
#if (NGX_HTTP_CACHE)
    unsigned                   cache_status:3;
#endif

    /* 开启更大的内存及临时磁盘文件用于缓存来不及发送到下游的响应包体 */
    unsigned                   buffering:1;
    /* keepalive机制 */
    unsigned                   keepalive:1;
    unsigned                   upgrade:1;

    /* 表示是否已向上游服务器发送请求 */
    unsigned                   request_sent:1;
    /* 表示是否已经转发响应报头 */
    unsigned                   header_sent:1;
};
```

下面看下 upstream 处理上游响应包体的三种方式：

1. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 1 时，upstream 不转发响应包体到下游，并由HTTP 模块实现的 input_filter() 方法处理包体；
2. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 0 时，且 ngx_http_upstream_conf_t 配置结构体中的成员buffering 标志位为 1 时，upstream 将开启更多的内存和磁盘文件用于缓存上游的响应包体（此时，上游网速更快），并转发响应包体；
3. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 0 时，且

ngx_http_upstream_conf_t 配置结构体中的成员buffering 标志位为 0 时，upstream 将使用固定大小的缓冲区来转发响应包体；

ngx_http_upstream_conf_t 结构体

在结构体 ngx_http_upstream_t 的成员conf 中，conf 是一个结构体ngx_http_upstream_conf_t 变量，该变量设置了upstream 的限制性参数。ngx_http_upstream_conf_t 结构体定义如下：src/http/ngx_http_upstream.h

```
/* ngx_http_upstream_conf_t 结构体 */
typedef struct {
    /*
     * 若在ngx_http_upstream_t结构体中没有实现resolved成员时，
     * upstream这个结构体才会生效，定义上游服务器的配置；
     */
    ngx_http_upstream_srv_conf_t    *upstream;

    /* 建立TCP连接的超时时间 */
    ngx_msec_t                      connect_timeout;
    /* 发送请求的超时时间 */
    ngx_msec_t                      send_timeout;
    /* 接收响应的超时时间 */
    ngx_msec_t                      read_timeout;
    ngx_msec_t                      timeout;

    /* TCP的SO_SNOLOWAT选项，表示发送缓冲区的下限 */
    size_t                          send_lowat;
    /* ngx_http_upstream_t中的buffer大小 */
    size_t                          buffer_size;

    size_t                          busy_buffers_size;
    /* 临时文件的最大长度 */
    size_t                          max_temp_file_size;
    /* 表示缓冲区中的响应写入到临时文件时一次写入字符流的最大长度 */
    size_t                          temp_file_write_size;

    size_t                          busy_buffers_size_conf;
    size_t                          max_temp_file_size_conf;
    size_t                          temp_file_write_size_conf;

    /* 以缓存响应的方式转发上游服务器的包体时所使用的内存大小 */
    ngx_bufs_t                      bufs;

    /* ignore_headers使得upstream在转发包头时跳过对某些头部的处理 */
    ngx_uint_t                      ignore_headers;
    /*
     * 以二进制位来处理错误码，若处理上游响应时发现这些错误码，
     * 那么在没有将响应转发给下游客户端时，将会选择一个上游服务器来重发请求；
     */
    ngx_uint_t                      next_upstream;
    /* 表示所创建的目录与文件的权限 */
    ngx_uint_t                      store_access;
```

```
    /*
     * 转发响应方式的标志位，为1表示启用更多内存和磁盘文件缓存来自上游响应(即上游网速优先)；
     * 若为0，则启用固定内存大小缓存上游响应(即下游网速优先)；
     */
    ngx_flag_t                   buffering;
    ngx_flag_t                   pass_request_headers;
    ngx_flag_t                   pass_request_body;

    /* 不检查Nginx与下游之间的连接是否断开 */
    ngx_flag_t                   ignore_client_abort;
    ngx_flag_t                   intercept_errors;
    /* 复用临时文件中已使用过的空间 */
    ngx_flag_t                   cyclic_temp_file;

    /* 存放临时文件的目录 */
    ngx_path_t                   *temp_path;

    /* 不转发的头部 */
    ngx_hash_t                    hide_headers_hash;
    /*
     * 当转发上游响应头部到下游客户端时，
     * 若不希望将某些头部转发，则设置在这个数组中
     */
    ngx_array_t                  *hide_headers;
    /*
     * 当转发上游响应头部到下游客户端时，
     * 若希望将某些头部转发，则设置在这个数组中
     */
    ngx_array_t                  *pass_headers;

    /* 连接上游服务器的本机地址 */
    ngx_http_upstream_local_t       *local;

#if (NGX_HTTP_CACHE)
    ngx_shm_zone_t               *cache;

    ngx_uint_t                   cache_min_uses;
    ngx_uint_t                   cache_use_stale;
    ngx_uint_t                   cache_methods;

    ngx_flag_t                   cache_lock;
    ngx_msec_t                    cache_lock_timeout;

    ngx_flag_t                   cache_revalidate;

    ngx_array_t                  *cache_valid;
    ngx_array_t                  *cache_bypass;
    ngx_array_t                  *no_cache;
#endif

    /*
     * 当ngx_http_upstream_t 中的store标志位为1时，
     * 如果需要将上游的响应存放在文件中，
     * store_lengths表示存放路径的长度；
     * store_values表示存放路径：
```

```
     store_values表示存放路径，
     */
    ngx_array_t                  *store_lengths;
    ngx_array_t                  *store_values;

    /* 文件缓存的路径 */
    signed                        store:2;
    /* 直接将上游返回的404错误码转发给下游 */
    unsigned                      intercept_404:1;
    /* 根据返回的响应头部，动态决定是以上游网速还是下游网速优先 */
    unsigned                      change_buffering:1;

#if (NGX_HTTP_SSL)
    ngx_ssl_t                    *ssl;
    ngx_flag_t                    ssl_session_reuse;
#endif

    /* 使用upstream的模块名称，仅用于记录日志 */
    ngx_str_t                     module;
} ngx_http_upstream_conf_t;
```

在 HTTP 反向代理模块在配置文件 nginx.conf 提供的配置项大都是用来设置结构体 ngx_http_upstream_conf_t 的成员。3 个超时时间成员是必须要设置的，因为他们默认是 0，即若不设置这 3 个成员，则无法与上游服务器建立TCP 连接。每一个请求都有独立的ngx_http_upstream_conf_t 结构体，因此，每个请求都可以拥有不同的网络超时时间等配置。

例如，将 nginx.conf 文件中的 upstream_conn_timeout 配置项解析到 ngx_http_hello_conf_t 结构体中的成员upstream.conn_timeout 中。可定义如下的连接超时时间，并把ngx_http_hello_conf_t 配置项的 upstream 成员赋给 ngx_http_upstream_t 中的conf 即可；

```
typedef struct
{
        ...
        ngx_http_upstream_conf_t    upstream;
}ngx_http_hello_conf_t;


static ngx_command_t ngx_http_hello_commands[] = {
  {
                ngx_string("upstream_conn_timeout"),
                NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
                ngx_conf_set_msec_slot,
                NGX_HTTP_LOC_CONF_OFFSET,
                offsetof(ngx_http_hello_conf_t, upstream.conn_timeout),
                NULL },


        ngx_null_command
};
/* 在 ngx_http_hello_handler 方法中如下定义 */

static ngx_int_t
ngx_http_hello_handler(ngx_http_request_t *r)
{
    ...
    ngx_http_hello_conf_t *mycf = (ngx_http_hello_conf_t *)ngx_http_get_module_loc_conf(r,ngx_http_
hello_module);
    r->upstream->conf = &mycf->upstream;
    ...
}
```

设置第三方服务器地址

在 ngx_http_upstream_t 结构体中的resolved 成员可直接设置上游服务器的地址，也可以由 nginx.conf 文件中配置upstream 模块，并指定上游服务器的地址。resolved 类型定义如下：

```
typedef struct {
    /* 主机名称 */
    ngx_str_t                       host;
    /* 端口号 */
    in_port_t                       port;
    ngx_uint_t                       no_port; /* unsigned no_port:1 */

    /* 地址个数 */
    ngx_uint_t                       naddrs;
    /* 地址 */
    ngx_addr_t                      *addrs;

    /* 上游服务器地址 */
    struct sockaddr                 *sockaddr;
    /* 上游服务器地址长度 */
    socklen_t                       socklen;

    ngx_resolver_ctx_t              *ctx;
} ngx_http_upstream_resolved_t;
```

设置回调方法

在结构体 ngx_http_upstream_t 中定义了 8 个回调方法：

```
/*
 * 处理包体前的初始化方法；
 * 其中data参数用于传递用户数据结构，就是下面成员input_filter_ctx
 */
ngx_int_t                  (*input_filter_init)(void *data);
/*
 * 处理包体的方法；
 * 其中data参数用于传递用户数据结构，就是下面成员input_filter_ctx，
 * bytes表示本次接收到包体的长度；
 */
ngx_int_t                  (*input_filter)(void *data, ssize_t bytes);
/* 用于传递HTTP自定义的数据结构 */
void                       *input_filter_ctx;

/* HTTP模块实现的create_request方法用于构造发往上游服务器的请求 */
ngx_int_t                  (*create_request)(ngx_http_request_t *r);
/* 与上游服务器的通信失败后，若想再次向上游服务器发起连接，则调用该函数 */
ngx_int_t                  (*reinit_request)(ngx_http_request_t *r);
/*
 * 解析上游服务器返回的响应包头，该函数返回四个值中的一个：
 * NGX_AGAIN                        表示包头没有接收完整；
 * NGX_HTTP_UPSTREAM_INVALID_HEADER    表示包头不合法；
 * NGX_ERROR                        表示出现错误；
 * NGX_OK                           表示解析到完整的包头；
 */
ngx_int_t                  (*process_header)(ngx_http_request_t *r);
/* 当客户端放弃请求时被调用，由于系统会自动关闭连接，因此，该函数不会进行任何具体操作 */
void                       (*abort_request)(ngx_http_request_t *r);
/* 结束upstream请求时会调用该函数 */
void                       (*finalize_request)(ngx_http_request_t *r,
                             ngx_int_t rc);
/*
 * 在上游返回的响应出现location或者refresh头部表示重定向时，
 * 会通过ngx_http_upstream_process_headers方法调用到可由HTTP模块
 * 实现的rewrite_redirect方法；
 */
ngx_int_t                  (*rewrite_redirect)(ngx_http_request_t *r,
                             ngx_table_elt_t *h, size_t prefix);
```

在这些回调方法中，其中有 3 个非常重要，在模块中是必须要实现的，这 3 个回调函数为：

```
/* HTTP模块实现的create_request方法用于构造发往上游服务器的请求 */
    ngx_int_t                  (*create_request)(ngx_http_request_t *r);
/*
     * 解析上游服务器返回的响应包头，该函数返回四个值中的一个：
     * NGX_AGAIN                          表示包头没有接收完整；
     * NGX_HTTP_UPSTREAM_INVALID_HEADER     表示包头不合法；
     * NGX_ERROR                          表示出现错误；
     * NGX_OK                             表示解析到完整的包头；
     */
    ngx_int_t                  (*process_header)(ngx_http_request_t *r);
/* 结束upstream请求时会调用该函数 */
    void                       (*finalize_request)(ngx_http_request_t *r,
                                   ngx_int_t rc);
```

create_request 在初始化 upstream 时被调用，生成发送到后端服务器的请求缓冲（缓冲链）。reinit_request 在某台后端服务器出错的情况，Nginx 会尝试连接到另一台后端服务器。Nginx 选定新的服务器以后，会先调用此函数，以重新初始化upstream 模块的工作状态，然后再次进行 upstream 连接。process_header 是用于解析上游服务器返回的基于TCP 的响应头部。finalize_request 在正常完成与后端服务器的请求后 或 失败 导致销毁请求时，该方法被调用。input_filter_init 和input_filter 都用于处理上游的响应包体，因为在处理包体前HTTP 模块可能需要做一些初始化工作。初始化工作由 input_filter_init 完成，实际处理包体由 input_filter 方法完成。

启动 upstream 机制

调用 ngx_http_upstream_init 方法便可启动upstream 机制，此时，必须通过返回NGX_DONE 通知 HTTP 框架暂停执行请求的下一个阶段，并且需要执行r->main->count++ 告知HTTP 框架将当前请求的引用计数增加 1，即告知ngx_http_hello_handler 方法暂时不要销毁请求，因为HTTP 框架只有在引用计数为 0 时才真正销毁请求。例如：

```
static ngx_int_t ngx_http_hello_handler(ngx_http_request_t *r)
{
    ...
    r->main->count++;
    ngx_http_upstream_init(r);
    return NGX_DONE;
}
```

subrequest 使用方式

subrequest 只是分解复杂请求的一种设计模式，它可以把原始请求分解为多个子请求，使得诸多请求协同完成一个用户请求，并且每个请求只关注一个功能。首先，若不是完全将上游服务器的响应包体转发到下游客户端，基本都会使用subrequest 创建子请求，并由子请求使用upstream 机制访问上游服务器，然后由父请求根据上游响应重新构造返回给下游客户端的响应。

subrequest 的使用步骤如下：

1. 在 nginx.conf 配置文件中配置好子请求的处理方式；
2. 启动 subrequest 子请求；
3. 实现子请求执行结束时的回调函数；
4. 实现父请求被激活时的回调函数；

配置子请求的处理方式

子请求并不是由 HTTP 框架解析所接收到客户端网络包而得到的，而是由父请求派生的。它的配置和普通请求的配置相同，都是在nginx.conf 文件中配置相应的处理模块。例如：可以在配置文件nginx.conf 中配置以下的子请求访问 https://github.com

```
location /subrq {
        rewrite ^/subrq(.*)$ $1 break;
        proxy_pass https://github.com;
    }
```

启动 subrequest 子请求

subrequest 是在父请求的基础上派生的子请求，subrequest 返回的内容会被附加到父请求上面，他的实现方法是调用ngx_http_subrequest 函数，该函数定义在文件：src/http/ngx_http_core_module.h

```
ngx_int_t ngx_http_subrequest(ngx_http_request_t *r,
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
    ngx_http_post_subrequest_t *ps, ngx_uint_t flags);
```

该函数的参数如下：引用自文件《Emiller's Advanced Topics In Nginx Module Development》

- *r is the original request（当前的请求，即父请求）；
- *uri and*argsrefer to the sub-request（*uri 是子请求的URI*，args是子请求URI 的参数）；
- psr is a reference to a NULL pointer that will point to the new (sub-)request structure（ psr 是指向返回子请求，相当于值-结果传递，作为参数传递进去是指向 NULL 指针，输出结果是指向新创建的子请求）；
- *ps is a callback for when the subrequest is finished.（* ps 是指出子请求结束时必须回调的处理方法）；
- flags can be a bitwise-OR'ed combination of:
- NGX_HTTP_ZERO_IN_URI: the URI contains a character with ASCII code 0 (also known as '\0'), or contains "%00"
- NGX_HTTP_SUBREQUEST_IN_MEMORY: store the result of the subrequest in a contiguous chunk of memory (usually not necessary)（将子请求的subrequest_in_memory 标志位为 1，表

示发起的子请求，访问的网络资源返回的响应将全部在内存中处理）；

- NGX_HTTP_SUBREQUEST_WAITED: store the result of the subrequest in a contiguous chunk of memory (usually not necessary)（将子请求的waited 标志位为 1，表示子请求完成后会设置自身的 r->done 标志位，可以通过判断该标志位得知子请求是否完成）；

该函数 ngx_http_subrequest 的返回值如下：

- NGX_OK:the subrequest finished without touching the network（成功建立子请求）；
- NGX_DONE:the client reset the network connection（客户端重置网络连接）；
- NGX_ERROR:there was a server error of some sort（建立子请求失败）；
- NGX_AGAIN:the subrequest requires network activity（子请求需要激活网络）；

该子请求返回的结果附加在你期望的位置。若要修改子请求的结果，可以使用 another filter（或同一个）。并告知该 filter 对父请求或子请求进行操作：具体实例可参照模块"addition" module

```
if (r == r->main) {
    /* primary request */
} else {
    /* subrequest */
}
```

以下是子请求函数 ngx_http_subrequest 的源码剖析，其源码定义在文件：src/http/ngx_http_core_module.c

```
/* ngx_http_subrequest 函数 */
ngx_int_t
ngx_http_subrequest(ngx_http_request_t *r,
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
    ngx_http_post_subrequest_t *ps, ngx_uint_t flags)
{
    ngx_time_t                  *tp;
    ngx_connection_t            *c;
    ngx_http_request_t          *sr;
    ngx_http_core_srv_conf_t    *cscf;
    ngx_http_postponed_request_t  *pr, *p;

    /* 原始请求的子请求减少一个 */
    r->main->subrequests--;

    /* 若没有子请求则出错返回 */
    if (r->main->subrequests == 0) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                      "subrequests cycle while processing \"%V\"", uri);
        r->main->subrequests = 1;
        return NGX_ERROR;
    }
```

```
    /* 分配内存sr */
    sr = ngx_pcalloc(r->pool, sizeof(ngx_http_request_t));
    if (sr == NULL) {
        return NGX_ERROR;
    }

    /* 设置为 HTTP 模块 */
    sr->signature = NGX_HTTP_MODULE;

    /* 设置sr的客户端连接 */
    c = r->connection;
    sr->connection = c;

    /* 为自定义上下文结构分配内存 */
    sr->ctx = ngx_pcalloc(r->pool, sizeof(void *) * ngx_http_max_module);
    if (sr->ctx == NULL) {
        return NGX_ERROR;
    }

    /* 初始化headers链表，该链表存储待发送的http响应包体 */
    if (ngx_list_init(&sr->headers_out.headers, r->pool, 20,
                      sizeof(ngx_table_elt_t))
        != NGX_OK)
    {
        return NGX_ERROR;
    }

    /* 设置main、server、location级别的配置结构体指针 */
    cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
    sr->main_conf = cscf->ctx->main_conf;
    sr->srv_conf = cscf->ctx->srv_conf;
    sr->loc_conf = cscf->ctx->loc_conf;

    /* 设置内存池 */
    sr->pool = r->pool;

    /* 设置headers_in成员，该成员保存解析完成的http头部 */
    sr->headers_in = r->headers_in;

    ngx_http_clear_content_length(sr);
    ngx_http_clear_accept_ranges(sr);
    ngx_http_clear_last_modified(sr);

    /* 设置接收请求包体的数据结构 */
    sr->request_body = r->request_body;

#if (NGX_HTTP_SPDY)
    sr->spdy_stream = r->spdy_stream;
#endif

    /* 请求的方法名称 */
    sr->method = NGX_HTTP_GET;
    /* 请求协议的版本 */
    sr->http_version = r->http_version;
```

```
    /* 请求行 */
    sr->request_line = r->request_line;
    /* 请求中的uri */
    sr->uri = *uri;

    /* uri中的参数 */
    if (args) {
        sr->args = *args;
    }

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http subrequest \"%V?%V\"", uri, &sr->args);

    /* 标志位 */
    sr->subrequest_in_memory = (flags & NGX_HTTP_SUBREQUEST_IN_MEMORY) != 0;
    sr->waited = (flags & NGX_HTTP_SUBREQUEST_WAITED) != 0;

    sr->unparsed_uri = r->unparsed_uri;
    sr->method_name = ngx_http_core_get_method;
    sr->http_protocol = r->http_protocol;

    ngx_http_set_exten(sr);

    /* 原始请求 */
    sr->main = r->main;
    sr->parent = r;/* 当前请求，即新创建子请求的父请求 */
    sr->post_subrequest = ps;/* 子请求执行结束时，执行的回调方法 */
    /* http请求的可读或可写事件的处理方法 */
    sr->read_event_handler = ngx_http_request_empty_handler;
    sr->write_event_handler = ngx_http_handler;

    /* 保存当前可以向out chain输出数组的请求 */
    if (c->data == r && r->postponed == NULL) {
        c->data = sr;
    }

    /* 默认共享父请求的变量，也可以根据需求创建完子请求后，再创建子请求独立的变量集 */
    sr->variables = r->variables;

    /* 日志处理方法 */
    sr->log_handler = r->log_handler;

    pr = ngx_palloc(r->pool, sizeof(ngx_http_postponed_request_t));
    if (pr == NULL) {
        return NGX_ERROR;
    }

    pr->request = sr;
    pr->out = NULL;
    pr->next = NULL;

    /* 把该子请求挂载到其父请求的postponed链表队尾 */
    if (r->postponed) {
        for (p = r->postponed; p->next; p = p->next) { /* void */ }
```

```
        for (p = r->postponed; p->next; p = p->next) { /* void */ }
        p->next = pr;

    } else {
        r->postponed = pr;
    }

    /* 子请求为内部请求 */
    sr->internal = 1;

    /* 继承父请求的部分状态 */
    sr->discard_body = r->discard_body;
    sr->expect_tested = 1;
    sr->main_filter_need_in_memory = r->main_filter_need_in_memory;

    sr->uri_changes = NGX_HTTP_MAX_URI_CHANGES + 1;

    tp = ngx_timeofday();
    sr->start_sec = tp->sec;
    sr->start_msec = tp->msec;

    /* 增加原始请求的引用计数 */
    r->main->count++;

    *psr = sr;/* 指向新创建的子请求 */

    /* 将该子请求挂载到原始请求的posted_requests链表队尾 */
    return ngx_http_post_request(sr, NULL);
}
/* 其中 ngx_http_post_request 定义在文件 src/http/ngx_http_request.c */
ngx_int_t
ngx_http_post_request(ngx_http_request_t *r, ngx_http_posted_request_t *pr)
{
    ngx_http_posted_request_t  **p;

    if (pr == NULL) {
        pr = ngx_palloc(r->pool, sizeof(ngx_http_posted_request_t));
        if (pr == NULL) {
            return NGX_ERROR;
        }
    }

    pr->request = r;
    pr->next = NULL;

    for (p = &r->main->posted_requests; *p; p = &(*p)->next) { /* void */ }

    *p = pr;

    return NGX_OK;
}
```

子请求结束时的回调函数

在子请求结束时（正常或异常结束）Nginx 会调用ngx_http_post_subrequest_pt 回调处理方法。下面是回调方法的定义：

```
typedef struct {
    ngx_http_post_subrequest_pt       handler;
    void                              *data;
} ngx_http_post_subrequest_t;

typedef ngx_int_t (*ngx_http_post_subrequest_pt)(ngx_http_request_t *r,
    void *data, ngx_int_t rc);
```

在结构体 ngx_http_post_subrequest_t 中，生成该结构体的变量时，可把用户的任意数据赋给指针data ，ngx_http_post_subrequest_pt 回调方法的参数data 就是用户把数据赋给结构体ngx_http_post_subrequest_t 中的成员指针data 所指的数据。ngx_http_post_subrequest_pt 回调方法中的参数rc 是子请求结束时的状态，它的取值由函数ngx_http_finalize_request 销毁请求时传递给参数rc。 函数ngx_http_finalize_request 的部分源码，具体可查阅文件：src/http/ngx_http_request.c

```
void
ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
{
  ...
    /* 如果当前请求是某个原始请求的一个子请求，检查它是否有回调handler处理函数，若存在则执行 */
    if (r != r->main && r->post_subrequest) {
        rc = r->post_subrequest->handler(r, r->post_subrequest->data, rc);
    }

  ...

    /* 若 r 是子请求 */
    if (r != r->main) {
        /* 该子请求还有未处理完的数据或者子请求 */
        if (r->buffered || r->postponed) {
            /* 添加一个该子请求的写事件，并设置合适的write event hander，
               以便下次写事件来的时候继续处理，这里实际上下次执行时会调用ngx_http_output_filter函数，
               最终还是会进入ngx_http_postpone_filter进行处理 */
            if (ngx_http_set_write_handler(r) != NGX_OK) {
                ngx_http_terminate_request(r, 0);
            }

            return;
        }
        ...
        pr = r->parent;


        /* 该子请求已经处理完毕，如果它拥有发送数据的权利，则将权利移交给父请求， */
        if (r == c->data) {

            r->main->count--;
```

```
        if (!r->logged) {

            clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

            if (clcf->log_subrequest) {
                ngx_http_log_request(r);
            }

            r->logged = 1;

        } else {
            ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                          "subrequest: \"%V?%V\" logged again",
                          &r->uri, &r->args);
        }

        r->done = 1;
        /* 如果该子请求不是提前完成，则从父请求的postponed链表中删除 */
        if (pr->postponed && pr->postponed->request == r) {
            pr->postponed = pr->postponed->next;
        }
        /* 将发送权利移交给父请求，父请求下次执行的时候会发送它的postponed链表中可以
         * 发送的数据节点，或者将发送权利移交给它的下一个子请求 */
        c->data = pr;

    } else {
        /* 该子请求提前执行完成，而且它没有产生任何数据，则它下次再次获得
         * 执行机会时，将会执行ngx_http_request_finalzier函数，它实际上是执行
         * ngx_http_finalzie_request（r,0），不做具体操作，直到它发送数据时，
         * ngx_http_finalzie_request函数会将它从父请求的postponed链表中删除
         */
        r->write_event_handler = ngx_http_request_finalizer;

        if (r->waited) {
            r->done = 1;
        }
    }
    /* 将父请求加入posted_request队尾，获得一次运行机会 */
    if (ngx_http_post_request(pr, NULL) != NGX_OK) {
        r->main->count++;
        ngx_http_terminate_request(r, 0);
        return;
    }

    return;
}
/* 这里是处理主请求结束的逻辑，如果主请求有未发送的数据或者未处理的子请求，
 * 则给主请求添加写事件，并设置合适的write event hander，
 * 以便下次写事件来的时候继续处理 */
if (r->buffered || c->buffered || r->postponed || r->blocked) {

    if (ngx_http_set_write_handler(r) != NGX_OK) {
        ngx_http_terminate_request(r, 0);
    }
```

```
        return;
    }

  ...
}
```

父请求被激活后的回调方法

父请求被激活后的回调方法由指针 ngx_http_event_pt 实现。该方法负责把响应包发送给用户。如下所示：

```
typedef void(*ngx_http_event_handler_pt)(ngx_http_request_t *r);

struct ngx_http_request_s{
    ...
    ngx_http_event_handler_pt      write_event_handler;
    ...
};
```

一个请求中，只能调用一次 subrequest，即不能一次创建多个子请求，但是可以在新创建的子请求中再创建新的子请求。

参考资料：

《深入理解Nginx 》

《Emiller's Advanced Topics In Nginx Module Development》

《nginx subrequest的实现解析》

《ngx_http_request_t结构体》

Nginx 源码结构分析

Nginx 源码基本结构

学习 Nginx 的构架之前，对 Nginx 源码结构进行简单的分析，可以了解 Nginx 模块结构以及模块之间的关系。充分理解Nginx 的基本构架。解压源码到相应的文件后，我们可以看到有一个存放源码的目录文件src，该目录文件存储Nginx 所有的源代码。首先，我们通过命令查看源码的组织结构：

```
$ tree -L 1
.
├── core
├── event
├── http
├── mail
├── misc
└── os

6 directories, 0 files
```

输出结果显示有 6 个目录文件，以下是这些目录文件的功能：

- core ：Nginx的核心源代码，包括常用数据结构的以及Nginx 内核实现的核心代码；
- event：Nginx事件驱动模型，以及定时器的实现相关代码；
- http ：Nginx 实现http 服务器相关的代码；
- mail ：Nginx 实现邮件代理服务器相关的代码；
- misc ：辅助代码，测试C++头 的兼容性，以及对Google_PerfTools 的支持；
- os ：不同体系统结构所提供的系统函数的封装，提供对外统一的系统调用接口；

下面主要针对重要的三个目录进行简单的介绍：core 目录、http 目录、event 目录；

core 核心模块结构

core 目录中的源码定义了 Nginx 服务器最基本的数据结构以及最基本的核心模块（核心模块为其他模块提供了公共调用的基本功能）。首先看下该核心模块的源码结构：

```
/* 实现对各模块的整体控制，是 Nginx 程序 main 函数 */
├── nginx.c
├── nginx.h

/* 以下是基本数据结构及其操作 */
├── ngx_array.c
├── ngx_array.h
```

```
├── ngx_hash.c
├── ngx_hash.h
├── ngx_list.c
├── ngx_list.h
├── ngx_queue.c
├── ngx_queue.h
├── ngx_radix_tree.c
├── ngx_radix_tree.h
├── ngx_rbtree.c
├── ngx_rbtree.h
├── ngx_output_chain.c
├── ngx_buf.c
├── ngx_buf.h
/* 整个Nginx 模块构架基本配置管理  */
├── ngx_conf_file.c
├── ngx_conf_file.h
├── ngx_config.h
/* 网络连接管理 */
├── ngx_connection.c
├── ngx_connection.h
/* 定义一些头文件与结构别名 */
├── ngx_core.h
├── ngx_cpuinfo.c
/* CRC 校验表信息 */
├── ngx_crc32.c
├── ngx_crc32.h
├── ngx_crc.h
/* 实现对系统运行过程参数、资源的通用管理 */
├── ngx_cycle.c
├── ngx_cycle.h
/* 实现文件读写相关的功能 */
├── ngx_file.c
├── ngx_file.h
/* socket 网络套接字功能 */
├── ngx_inet.c
├── ngx_inet.h
/* 实现日志输出、管理的相关功能 */
├── ngx_log.c
├── ngx_log.h
├── ngx_syslog.c
├── ngx_syslog.h
/* hash字符串操作 */
├── ngx_md5.c
├── ngx_md5.h
├── ngx_murmurhash.c
├── ngx_murmurhash.h
/* 内存管理相关文件 */
├── ngx_open_file_cache.c
├── ngx_open_file_cache.h
├── ngx_palloc.c
├── ngx_palloc.h
├── ngx_shmtx.c
├── ngx_shmtx.h
├── ngx_slab.c
├── ngx_slab.h
```

```
├── ngx_slab.h
/* PCRE 上层封装 */
├── ngx_parse.c
├── ngx_parse.h
/* 反向代理的协议信息 */
├── ngx_proxy_protocol.c
├── ngx_proxy_protocol.h
/* 实现支持正则表达式 */
├── ngx_regex.c
├── ngx_regex.h
/* 字符串处理功能 */
├── ngx_string.c
├── ngx_string.h
/* 时间获取与管理功能 */
├── ngx_times.c
└── ngx_times.h
/* 其他文件 */
├── ngx_resolver.c
├── ngx_resolver.h
├── ngx_sha1.h
├── ngx_spinlock.c
├── ngx_crypt.c
├── ngx_crypt.h
```

event 事件驱动模型结构

event 目录里面包含一种子目录 module 以及一些文件，除了 module 子目录，其他文件提供了事件驱动模型相关数据结构的定义、初始化、事件接收、传递、管理功能以及事件驱动模型调用功能。module 子目录里面的源码实现了Nginx 支持的事件驱动模型：AIO、epoll、kqueue、select、/dev/poll、poll 等事件驱动模型；

```
.
├── modules
│   ├── ngx_aio_module.c          /* AIO 事件驱动模型 */
│   ├── ngx_devpoll_module.c      /* dev/poll 事件驱动模型 */
│   ├── ngx_epoll_module.c        /* epoll 事件驱动模型 */
│   ├── ngx_eventport_module.c    /* 事件驱动模型端口 */
│   ├── ngx_kqueue_module.c       /* kqueue 事件驱动模型 */
│   ├── ngx_poll_module.c         /* poll 事件驱动模型 */
│   ├── ngx_rtsig_module.c        /* rtsing 事件驱动模型 */
│   ├── ngx_select_module.c       /* Linux 平台下的 select 事件驱动模型 */
│   └── ngx_win32_select_module.c  /* Win32 平台下的 select 事件驱动模型 */
├── ngx_event_accept.c
├── ngx_event_busy_lock.c
├── ngx_event_busy_lock.h
├── ngx_event.c
├── ngx_event_connect.c
├── ngx_event_connect.h
├── ngx_event.h
├── ngx_event_mutex.c
├── ngx_event_openssl.c
├── ngx_event_openssl.h
├── ngx_event_openssl_stapling.c
├── ngx_event_pipe.c
├── ngx_event_pipe.h
├── ngx_event_posted.c
├── ngx_event_posted.h
├── ngx_event_timer.c
└── ngx_event_timer.h

1 directory, 26 files
```

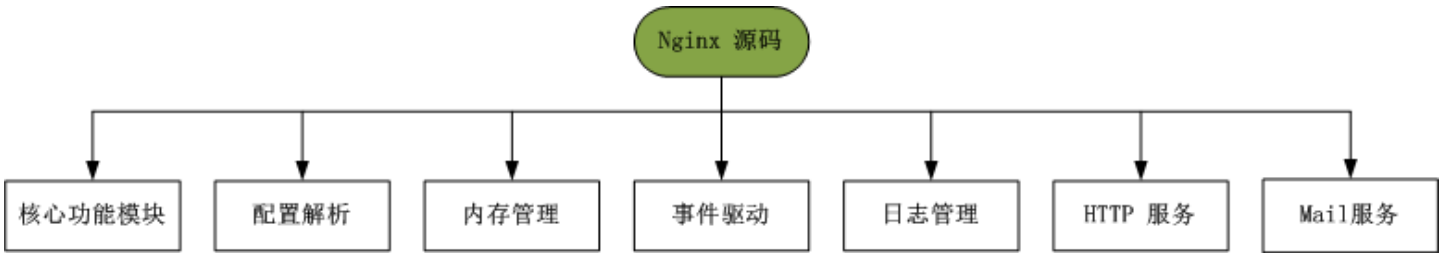
http 模块结构

http 目录和 event 目录一样，通用包含了模块实现源码的 module 目录文件以及一些结构定义、初始化、网络连接建立、管理、关闭，以及数据报解析、服务器组管理等功能的源码文件。module 目录文件实现了HTTP 模块的功能。

```
.
├── modules
├── ngx_http_busy_lock.c
├── ngx_http_busy_lock.h
├── ngx_http.c
├── ngx_http_cache.h
├── ngx_http_config.h
├── ngx_http_copy_filter_module.c
├── ngx_http_core_module.c
├── ngx_http_core_module.h
├── ngx_http_file_cache.c
├── ngx_http.h
├── ngx_http_header_filter_module.c
├── ngx_http_parse.c
├── ngx_http_parse_time.c
├── ngx_http_postpone_filter_module.c
├── ngx_http_request_body.c
├── ngx_http_request.c
├── ngx_http_request.h
├── ngx_http_script.c
├── ngx_http_script.h
├── ngx_http_spdy.c
├── ngx_http_spdy_filter_module.c
├── ngx_http_spdy.h
├── ngx_http_spdy_module.c
├── ngx_http_spdy_module.h
├── ngx_http_special_response.c
├── ngx_http_upstream.c
├── ngx_http_upstream.h
├── ngx_http_upstream_round_robin.c
├── ngx_http_upstream_round_robin.h
├── ngx_http_variables.c
├── ngx_http_variables.h
└── ngx_http_write_filter_module.c

1 directory, 32 files
```

Nginx 源码的模块化结构

根据各模块的功能，可把 Nginx 源码划分为以下几种功能，如下图所示：



- 核心模块功能：为其他模块提供一些基本功能：字符串处理、时间管理、文件读写等功能；
- 配置解析：主要包括文件语法检查、配置参数解析、参数初始化等功能；
- 内存管理：内存池管理、共享内存的分配、缓冲区管理等功能；

- 事件驱动：进程创建与管理、信号接收与处理、所有事件驱动模型的实现、高级 IO 等功能；
- 日志管理：错误日志的生成与管理、任务日志的生成与管理等功能；
- HTTP 服务：提供 Web 服务，包括客户度连接管理、客户端请求处理、虚拟主机管理、服务器组管理等功能；
- Mail 服务：与 HTTP 服务类似，但是增加了邮件协议的实现；

Nginx 事件模块

概述

Nginx 是以事件的触发来驱动的，事件驱动模型主要包括事件收集、事件发送、事件处理（即事件管理）三部分。在Nginx 的工作进程中主要关注的事件是IO 网络事件 和 定时器事件。在生成的 objs 目录文件中，其中ngx_modules.c 文件的内容是Nginx 各种模块的执行顺序，我们可以从该文件的内容中看到事件模块的执行顺序为以下所示：注意：由于是在Linux 系统下，所以支持具体的 epoll 事件模块，接下来的文章结构按照以下顺序来写。

```
extern ngx_module_t  ngx_events_module;
extern ngx_module_t  ngx_event_core_module;
extern ngx_module_t  ngx_epoll_module;
```

事件模块接口

ngx_event_module_t 结构体

在 Nginx 中，结构体 ngx_module_t 是 Nginx 模块最基本的接口。对于每一种不同类型的模块，都有一个具体的结构体来描述这一类模块的通用接口，该接口由ngx_module_t 中的成员ctx 管理。在 Nginx 中定义了事件模块的通用接口ngx_event_module_t 结构体，该结构体定义在文件src/event/ngx_event.h 中：

```
/* 事件驱动模型通用接口ngx_event_module_t结构体 */
typedef struct {
    /* 事件模块名称 */
    ngx_str_t             *name;

    /* 解析配置项前调用，创建存储配置项参数的结构体 */
    void                *(*create_conf)(ngx_cycle_t *cycle);
    /* 完成配置项解析后调用，处理当前事件模块感兴趣的全部配置 */
    char                *(*init_conf)(ngx_cycle_t *cycle, void *conf);

    /* 每个事件模块具体实现的方法，有10个方法，即IO多路复用模型的统一接口 */
    ngx_event_actions_t     actions;
} ngx_event_module_t;
```

在 ngx_event_module_t 结构体中actions 的类型是ngx_event_actions_t 结构体，该成员结构实现了事件驱动模块的具体方法。该结构体定义在文件src/event/ngx_event.h 中：

```
/* IO多路复用模型的统一接口 */
typedef struct {
    /* 添加事件，将某个描述符的某个事件添加到事件驱动机制监控描述符集中 */
    ngx_int_t  (*add)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    /* 删除事件，将某个描述符的某个事件从事件驱动机制监控描述符集中删除 */
    ngx_int_t  (*del)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    /* 启动对某个指定事件的监控 */
    ngx_int_t  (*enable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    /* 禁用对某个指定事件的监控 */
    ngx_int_t  (*disable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    /* 将指定连接所关联的描述符添加到事件驱动机制监控中 */
    ngx_int_t  (*add_conn)(ngx_connection_t *c);
    /* 将指定连接所关联的描述符从事件驱动机制监控中删除 */
    ngx_int_t  (*del_conn)(ngx_connection_t *c, ngx_uint_t flags);

    /* 监控事件是否发生变化，仅用在多线程环境中 */
    ngx_int_t  (*process_changes)(ngx_cycle_t *cycle, ngx_uint_t nowait);
    /* 等待事件的发生，并对事件进行处理 */
    ngx_int_t  (*process_events)(ngx_cycle_t *cycle, ngx_msec_t timer,
                ngx_uint_t flags);

    /* 初始化事件驱动模块 */
    ngx_int_t  (*init)(ngx_cycle_t *cycle, ngx_msec_t timer);
    /* 在退出事件驱动模块前调用该函数回收资源 */
    void       (*done)(ngx_cycle_t *cycle);
} ngx_event_actions_t;
```

ngx_event_t 结构体

在 Nginx 中，每一个具体事件的定义由结构体ngx_event_t 来表示，该结构体ngx_event_t 用来保存具体事件。该结构体定义在文件 src/event/ngx_event.h 中：

```
/* 描述每一个事件的ngx_event_t结构体 */
struct ngx_event_s {
    /* 事件相关对象的数据，通常指向ngx_connect_t连接对象 */
    void            *data;

    /* 标志位，为1表示事件可写，即当前对应的TCP连接状态可写 */
    unsigned         write:1;

    /* 标志位，为1表示事件可以建立新连接 */
    unsigned         accept:1;

    /* used to detect the stale events in kqueue, rtsig, and epoll */
    unsigned         instance:1;

    /*
     * the event was passed or would be passed to a kernel;
     * in aio mode - operation was posted.
     */
```

```
     */
    /* 标志位，为1表示事件处于活跃状态 */
    unsigned         active:1;

    /* 标志位，为1表示禁用事件 */
    unsigned         disabled:1;

    /* the ready event; in aio mode 0 means that no operation can be posted */
    /* 标志位，为1表示当前事件已经准备就绪 */
    unsigned         ready:1;

    /* 该标志位只用于kqueue,eventport模块，对Linux上的驱动模块没有任何意义 */
    unsigned         oneshot:1;

    /* aio operation is complete */
    /* 该标志位用于异步AIO事件处理 */
    unsigned         complete:1;

    /* 标志位，为1表示当前处理的字符流已经结束 */
    unsigned         eof:1;
    /* 标志位，为1表示当前事件处理过程中出错 */
    unsigned         error:1;

    /* 标志位，为1表示当前事件已超时 */
    unsigned         timedout:1;
    /* 标志位，为1表示当前事件存在于定时器中 */
    unsigned         timer_set:1;

    /* 标志位，为1表示当前事件需要延迟处理 */
    unsigned         delayed:1;

    /*
     * 标志位，为1表示TCP建立需要延迟，即完成建立TCP连接的三次握手后，
     * 不会立即建立TCP连接，直到接收到数据包才建立TCP连接；
     */
    unsigned         deferred_accept:1;

    /* the pending eof reported by kqueue, epoll or in aio chain operation */
    /* 标志位，为1表示等待字符流结束 */
    unsigned         pending_eof:1;

    /* 标志位，为1表示处理post事件 */
    unsigned         posted:1;

#if (NGX_WIN32)
    /* setsockopt(SO_UPDATE_ACCEPT_CONTEXT) was successful */
    unsigned         accept_context_updated:1;
#endif

#if (NGX_HAVE_KQUEUE)
    unsigned         kq_vnode:1;

    /* the pending errno reported by kqueue */
    int              kq_errno;
#endif
```

```
    /*
     * kqueue only:
     *   accept:     number of sockets that wait to be accepted
     *   read:       bytes to read when event is ready
     *               or lowat when event is set with NGX_LOWAT_EVENT flag
     *   write:      available space in buffer when event is ready
     *               or lowat when event is set with NGX_LOWAT_EVENT flag
     *
     * iocp: TODO
     *
     * otherwise:
     *   accept:     1 if accept many, 0 otherwise
     */

#if (NGX_HAVE_KQUEUE) || (NGX_HAVE_IOCP)
    int              available;
#else
    /* 标志位，在epoll事件机制中表示一次尽可能多地建立TCP连接 */
    unsigned         available:1;
#endif

    /* 当前事件发生时的处理方法 */
    ngx_event_handler_pt  handler;

#if (NGX_HAVE_AIO)

#if (NGX_HAVE_IOCP)
    ngx_event_ovlp_t ovlp;
#else
    /* Linux系统aio机制中定义的结构体 */
    struct aiocb     aiocb;
#endif

#endif

    /* epoll机制不使用该变量 */
    ngx_uint_t       index;

    /* 日志记录 */
    ngx_log_t       *log;

    /* 定时器 */
    ngx_rbtree_node_t   timer;

    /* the posted queue */
    ngx_queue_t      queue;

    /* 标志位，为1表示当前事件已经关闭 */
    unsigned         closed:1;

    /* to test on worker exit */
    unsigned         channel:1;
    unsigned         resolver:1;
```

```
    unsigned         cancelable:1;

#if 0

    /* the threads support */

    /*
     * the event thread context, we store it here
     * if $(CC) does not understand __thread declaration
     * and pthread_getspecific() is too costly
     */

    void             *thr_ctx;

#if (NGX_EVENT_T_PADDING)

    /* event should not cross cache line in SMP */

    uint32_t         padding[NGX_EVENT_T_PADDING];
#endif
#endif
};
```

在每个事件结构体 ngx_event_t 最重要的成员是handler 回调函数，该回调函数定义了当事件发生时的处理方法。该回调方法原型在文件src/core/ngx_core.h 中：

```
typedef void (*ngx_event_handler_pt)(ngx_event_t *ev);
```

ngx_connection_t 结构体

当客户端向 Nginx 服务器发起连接请求时，此时若Nginx 服务器被动接收该连接，则相对Nginx 服务器来说称为被动连接，被动连接的表示由基本数据结构体ngx_connection_t 完成。该结构体定义在文件src/core/ngx_connection.h 中：

```
/* TCP连接结构体 */
struct ngx_connection_s {
    /*
     * 当Nginx服务器产生新的socket时，
     * 都会创建一个ngx_connection_s 结构体，
     * 该结构体用于保存socket的属性和数据；
     */

    /*
     * 当连接未被使用时，data充当连接池中空闲连接表中的next指针；
     * 当连接被使用时，data的意义由具体Nginx模块决定；
     */
    void               *data;
    /* 设置该链接的读事件 */
    ngx_event_t        *read;
    /* 设置该连接的写事件 */
```

```
    ngx_event_t       *write;

    /* 用于设置socket的套接字描述符 */
    ngx_socket_t       fd;

    /* 接收网络字符流的方法，是一个函数指针，指向接收函数 */
    ngx_recv_pt        recv;
    /* 发送网络字符流的方法，是一个函数指针，指向发送函数 */
    ngx_send_pt        send;
    /* 以ngx_chain_t链表方式接收网络字符流的方法 */
    ngx_recv_chain_pt  recv_chain;
    /* 以ngx_chain_t链表方式发送网络字符流的方法 */
    ngx_send_chain_pt  send_chain;

    /*
     * 当前连接对应的ngx_listening_t监听对象，
     * 当前连接由ngx_listening_t成员的listening监听端口的事件建立；
     * 成员connection指向当前连接；
     */
    ngx_listening_t    *listening;

    /* 当前连接已发生的字节数 */
    off_t              sent;

    /* 记录日志 */
    ngx_log_t          *log;

    /* 内存池 */
    ngx_pool_t         *pool;

    /* 对端的socket地址sockaddr属性*/
    struct sockaddr    *sockaddr;
    socklen_t          socklen;
    /* 字符串形式的IP地址 */
    ngx_str_t          addr_text;

    ngx_str_t          proxy_protocol_addr;

#if (NGX_SSL)
    ngx_ssl_connection_t  *ssl;
#endif

    /* 本端的监听端口对应的socket的地址sockaddr属性 */
    struct sockaddr    *local_sockaddr;
    socklen_t          local_socklen;

    /* 用于接收、缓存对端发来的字符流 */
    ngx_buf_t          *buffer;

    /*
     * 表示将当前连接作为双向连接中节点元素，
     * 添加到ngx_cycle_t结构体的成员
     * reuseable_connections_queue的双向链表中；
     */
```

```
    ngx_queue_t        queue;

    /* 连接使用次数 */
    ngx_atomic_uint_t   number;

    /* 处理请求的次数 */
    ngx_uint_t          requests;

    unsigned            buffered:8;

    unsigned            log_error:3;     /* ngx_connection_log_error_e */

    /* 标志位，为1表示不期待字符流结束 */
    unsigned            unexpected_eof:1;
    /* 标志位，为1表示当前连接已经超时 */
    unsigned            timedout:1;
    /* 标志位，为1表示处理连接过程出错 */
    unsigned            error:1;
    /* 标志位，为1表示当前TCP连接已经销毁 */
    unsigned            destroyed:1;

    /* 标志位，为1表示当前连接处于空闲状态 */
    unsigned            idle:1;
    /* 标志位，为1表示当前连接可重用 */
    unsigned            reusable:1;
    /* 标志为，为1表示当前连接已经关闭 */
    unsigned            close:1;

    /* 标志位，为1表示正在将文件的数据发往对端 */
    unsigned            sendfile:1;
    /*
     * 标志位，若为1，则表示只有连接对应的发送缓冲区满足最低设置的阈值时，
     * 事件驱动模块才会分发事件；
     */
    unsigned            sndlowat:1;
    unsigned            tcp_nodelay:2;   /* ngx_connection_tcp_nodelay_e */
    unsigned            tcp_nopush:2;    /* ngx_connection_tcp_nopush_e */

    unsigned            need_last_buf:1;

#if (NGX_HAVE_IOCP)
    unsigned            accept_context_updated:1;
#endif

#if (NGX_HAVE_AIO_SENDFILE)
    /* 标志位，为1表示使用异步IO方式将磁盘文件发送给网络连接的对端 */
    unsigned            aio_sendfile:1;
    unsigned            busy_count:2;
    /* 使用异步IO发送文件时，用于待发送的文件信息 */
    ngx_buf_t          *busy_sendfile;
#endif

#if (NGX_THREADS)
    ngx_atomic_t        lock;
#endif
```

```
};
```

在处理请求的过程中，若 Nginx 服务器主动向上游服务器建立连接，完成连接建立并与之进行通信，这种相对Nginx 服务器来说是一种主动连接，主动连接由结构体ngx_peer_connection_t 表示，但是该结构体 ngx_peer_connection_t 也是 ngx_connection_t 结构体的封装。该结构体定义在文件src/event/ngx_event_connect.h 中：

```
/* 主动连接的结构体 */
struct ngx_peer_connection_s {
    /* 这里是对ngx_connection_t连接结构体的引用 */
    ngx_connection_t                *connection;

    /* 远端服务器的socket的地址sockaddr信息 */
    struct sockaddr                 *sockaddr;
    socklen_t                       socklen;
    /* 远端服务器的名称 */
    ngx_str_t                       *name;

    /* 连接重试的次数 */
    ngx_uint_t                      tries;

    /* 获取连接的方法 */
    ngx_event_get_peer_pt           get;
    /* 释放连接的方法 */
    ngx_event_free_peer_pt          free;
    /* 配合get、free使用 */
    void                            *data;

#if (NGX_SSL)
    ngx_event_set_peer_session_pt    set_session;
    ngx_event_save_peer_session_pt   save_session;
#endif

#if (NGX_THREADS)
    ngx_atomic_t                    *lock;
#endif

    /* 本地地址信息 */
    ngx_addr_t                      *local;

    /* 接收缓冲区 */
    int                             rcvbuf;

    /* 记录日志 */
    ngx_log_t                       *log;

    /* 标志位，为1表示connection连接已经缓存 */
    unsigned                        cached:1;

                                    /* ngx_connection_log_error_e */
    unsigned                        log_error:2;
};
```

ngx_events_module 核心模块

ngx_events_module 核心模块的定义

ngx_events_module 模块是事件的核心模块，该模块的功能是：定义新的事件类型，并为每个事件模

块定义通用接口ngx_event_module_t 结构体，管理事件模块生成的配置项结构体，并解析事件类配置项。首先，看下该模块在文件src/event/ngx_event.c 中的定义：

```
/* 定义事件核心模块 */
ngx_module_t  ngx_events_module = {
    NGX_MODULE_V1,
    &ngx_events_module_ctx,              /* module context */
    ngx_events_commands,                 /* module directives */
    NGX_CORE_MODULE,                     /* module type */
    NULL,                                /* init master */
    NULL,                                /* init module */
    NULL,                                /* init process */
    NULL,                                /* init thread */
    NULL,                                /* exit thread */
    NULL,                                /* exit process */
    NULL,                                /* exit master */
    NGX_MODULE_V1_PADDING
};
```

其中，模块的配置项指令结构 ngx_events_commands 决定了该模块的功能。配置项指令结构 ngx_events_commands 在文件src/event/ngx_event.c 中定义如下：

```
/* 配置项结构体数组 */
static ngx_command_t  ngx_events_commands[] = {

    { ngx_string("events"),
      NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
      ngx_events_block,
      0,
      0,
      NULL },

      ngx_null_command
};
```

从配置项结构体中可以知道，该模块只对 events{...} 配置块感兴趣，并定义了管理事件模块的方法 ngx_events_block；ngx_events_block 方法在文件src/event/ngx_event.c 中定义：

```
/* 管理事件模块 */
static char *
ngx_events_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    char                *rv;
    void               ***ctx;
    ngx_uint_t           i;
    ngx_conf_t           pcf;
    ngx_event_module_t   *m;

    if (*(void **) conf) {
        return "is duplicate";
```

```
    }

    /* count the number of the event modules and set up their indices */

    /* 计算模块类中模块的总数，并初始化模块在模块类中的序号 */
    ngx_event_max_module = 0;
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
            continue;
        }

        ngx_modules[i]->ctx_index = ngx_event_max_module++;
    }

    ctx = ngx_pcalloc(cf->pool, sizeof(void *));
    if (ctx == NULL) {
        return NGX_CONF_ERROR;
    }

    /* 分配指针数组，用于存储所有事件模块生成的配置项结构体指针 */
    *ctx = ngx_pcalloc(cf->pool, ngx_event_max_module * sizeof(void *));
    if (*ctx == NULL) {
        return NGX_CONF_ERROR;
    }

    *(void **) conf = ctx;

    /* 若是事件模块，并且定义了create_conf方法，则调用该方法创建存储配置项参数的结构体 */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
            continue;
        }

        m = ngx_modules[i]->ctx;

        if (m->create_conf) {
            (*ctx)[ngx_modules[i]->ctx_index] = m->create_conf(cf->cycle);
            if ((*ctx)[ngx_modules[i]->ctx_index] == NULL) {
                return NGX_CONF_ERROR;
            }
        }
    }

    /* 初始化配置项结构体cf */
    pcf = *cf;
    cf->ctx = ctx;/* 描述事件模块的配置项结构 */
    cf->module_type = NGX_EVENT_MODULE;/* 当前解析指令的模块类型 */
    cf->cmd_type = NGX_EVENT_CONF;/* 当前解析指令的指令类型 */

    /* 为所有事件模块解析配置文件nginx.conf中的event{}块中的指令 */
    rv = ngx_conf_parse(cf, NULL);

    *cf = pcf;

    if (rv != NGX_CONF_OK)
```

```
    if (rv != NGX_CONF_OK)
        return rv;

    /* 遍历所有事件模块，若定义了init_conf方法，则调用该方法用于处理事件模块感兴趣的配置项 */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
            continue;
        }

        m = ngx_modules[i]->ctx;

        if (m->init_conf) {
            rv = m->init_conf(cf->cycle, (*ctx)[ngx_modules[i]->ctx_index]);
            if (rv != NGX_CONF_OK) {
                return rv;
            }
        }
    }

    return NGX_CONF_OK;
}
```

另外，在 ngx_events_module 模块的定义中有一个成员ctx 指向了核心模块的通用接口结构。核心模块的通用接口结构体定义在文件src/core/ngx_conf_file.h 中：

```
/* 核心模块的通用接口结构体 */
typedef struct {
    /* 模块名称 */
    ngx_str_t             name;
    /* 解析配置项前，调用该方法 */
    void               *(*create_conf)(ngx_cycle_t *cycle);
    /* 完成配置项解析后，调用该函数 */
    char               *(*init_conf)(ngx_cycle_t *cycle, void *conf);
} ngx_core_module_t;
```

因此，ngx_events_module 作为核心模块，必须定义核心模块的通用接口结构。ngx_events_module 模块的核心模块通用接口在文件src/event/ngx_event.c 中定义：

```
/* 实现核心模块通用接口 */
static ngx_core_module_t  ngx_events_module_ctx = {
    ngx_string("events"),
    NULL,
    /*
     * 以前的版本这里是NULL，现在实现了一个获取events配置项的函数，*
     * 但是没有什么作用，因为每个事件模块都会去获取events配置项，
     * 并进行解析与处理；
     */
    ngx_event_init_conf
};
```

所有事件模块的配置项管理

Nginx 服务器在结构体 ngx_cycle_t 中定义了一个四级指针成员 conf_ctx，整个Nginx 模块都是使用该四级指针成员管理模块的配置项结构，以下events 模块为例对该四级指针成员进行简单的分析，如下图所示：

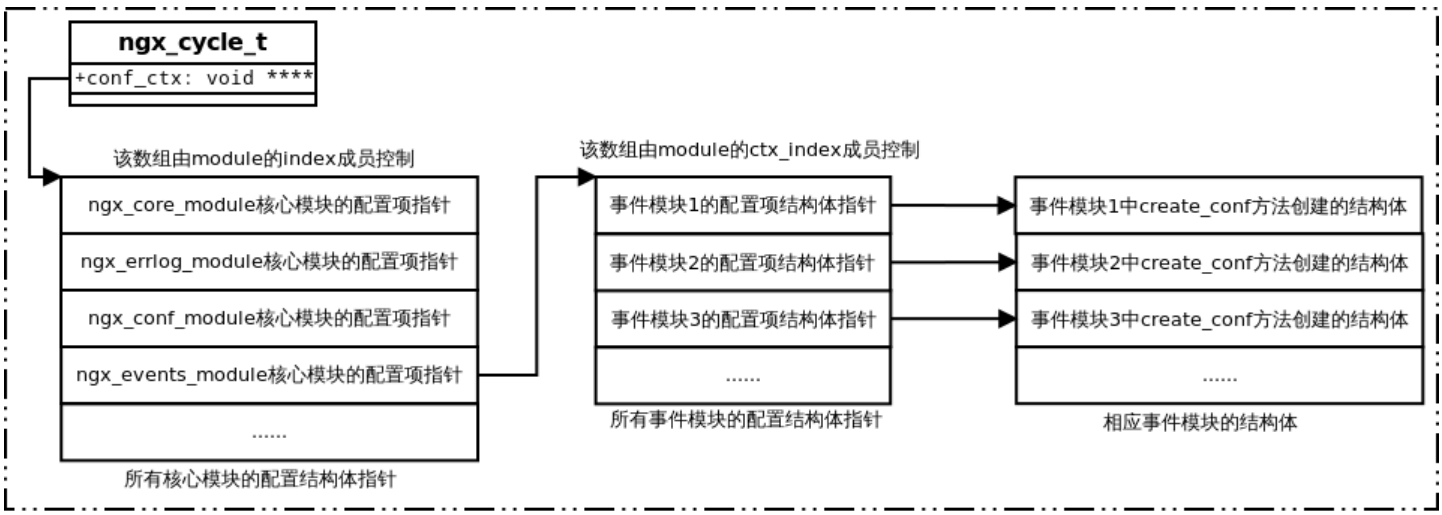


图 1 所有事件模块的配置结构体管理

每个事件模块可以通过宏定义 ngx_event_get_conf 获取它在create_conf 中分配的结构体的指针；该宏中定义如下：

```
#define ngx_event_get_conf(conf_ctx, module) \
(*(ngx_get_conf(conf_ctx, ngx_events_module))) [module.ctx_index];

/* 其中 ngx_get_conf 定义如下 */
#define ngx_get_conf(conf_ctx, module) conf_ctx[module.index]
```

从上面的宏定义可以知道，每个事件模块获取自己在 create_conf 中分配的结构体的指针，只需在 ngx_event_get_conf 传入参数ngx_cycle_t 中的 conf_ctx 成员，并且传入自己模块的名称即可获取自己分配的结构体指针。

ngx_event_core_module 事件模块

ngx_event_core_module 模块是一个事件类型的模块，它在所有事件模块中的顺序是第一，是其它事件类模块的基础。它主要完成以下任务：

1. 创建连接池；
2. 决定使用哪些事件驱动机制；
3. 初始化将要使用的事件模块；

ngx_event_conf_t 结构体

ngx_event_conf_t 结构体是用来保存ngx_event_core_module 事件模块配置项参数的。该结构体在文件src/event/ngx_event.h 中定义：

```
/* 存储ngx_event_core_module事件模块配置项参数的结构体 ngx_event_conf_t */
typedef struct {
    /* 连接池中最大连接数 */
    ngx_uint_t    connections;
    /* 被选用模块在所有事件模块中的序号 */
    ngx_uint_t    use;

    /* 标志位，为1表示可批量建立连接 */
    ngx_flag_t    multi_accept;
    /* 标志位，为1表示打开负载均衡锁 */
    ngx_flag_t    accept_mutex;

    /* 延迟建立连接 */
    ngx_msec_t    accept_mutex_delay;

    /* 被使用事件模块的名称 */
    u_char       *name;

#if (NGX_DEBUG)
    /* 用于保存与输出调试级别日志连接对应客户端的地址信息 */
    ngx_array_t   debug_connection;
#endif
} ngx_event_conf_t;
```

ngx_event_core_module 事件模块的定义

该模块在文件 src/event/ngx_event.c 中定义：

```
/* 事件模块的定义 */
ngx_module_t  ngx_event_core_module = {
    NGX_MODULE_V1,
    &ngx_event_core_module_ctx,            /* module context */
    ngx_event_core_commands,               /* module directives */
    NGX_EVENT_MODULE,                      /* module type */
    NULL,                                  /* init master */
    ngx_event_module_init,                 /* init module */
    ngx_event_process_init,                /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

其中，模块的配置项指令结构 ngx_event_core_commands 决定了该模块的功能。配置项指令结构 ngx_event_core_commands 在文件 src/event/ngx_event.c 中定义如下：

```
static ngx_str_t  event_core_name = ngx_string("event_core");

/* 定义ngx_event_core_module 模块感兴趣的配置项 */
```

```
static ngx_command_t  ngx_event_core_commands[] = {

    /* 每个worker进程中TCP最大连接数 */
    { ngx_string("worker_connections"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_event_connections,
      0,
      0,
      NULL },

    /* 与上面的worker_connections配置项相同 */
    { ngx_string("connections"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_event_connections,
      0,
      0,
      NULL },

    /* 选择事件模块作为事件驱动机制 */
    { ngx_string("use"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_event_use,
      0,
      0,
      NULL },

    /* 批量接收连接 */
    { ngx_string("multi_accept"),
      NGX_EVENT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      0,
      offsetof(ngx_event_conf_t, multi_accept),
      NULL },

    /* 是否打开accept_mutex负载均衡锁 */
    { ngx_string("accept_mutex"),
      NGX_EVENT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      0,
      offsetof(ngx_event_conf_t, accept_mutex),
      NULL },

    /* 打开accept_mutex负载均衡锁后，延迟处理新连接事件 */
    { ngx_string("accept_mutex_delay"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_msec_slot,
      0,
      offsetof(ngx_event_conf_t, accept_mutex_delay),
      NULL },

    /* 对指定IP的TCP连接打印debug级别的调试日志 */
    { ngx_string("debug_connection"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_event_debug_connection,
      0
```

```
    0,
    NULL },

    ngx_null_command
};
```

其中，每个事件模块都需要实现事件模块的通用接口结构 ngx_event_module_t，ngx_event_core_module 模块的上下文结构 ngx_event_core_module_ctx 并不真正的负责网络事件的驱动，所有不会实现ngx_event_module_t 结构体中的成员 actions 中的方法。上下文结构 ngx_event_core_module_ctx 在文件 src/event/ngx_event.c 中定义如下：

```
/* 根据事件模块通用接口，实现ngx_event_core_module事件模块的上下文结构 */
ngx_event_module_t  ngx_event_core_module_ctx = {
    &event_core_name,
    ngx_event_core_create_conf,            /* create configuration */
    ngx_event_core_init_conf,              /* init configuration */
    { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};
```

在模块定义中，实现了两种方法分别为 ngx_event_module_init 和ngx_event_process_init 方法。在 Nginx 启动过程中没有使用 fork 出 worker 子进程之前，先调用 ngx_event_core_module 模块中的 ngx_event_module_init 方法，当fork 出 worker 子进程后，每一个 worker 子进程则会调用 ngx_event_process_init 方法。

ngx_event_module_init 方法在文件 src/event/ngx_event.c 中定义：

```
/* 初始化事件模块 */
static ngx_int_t
ngx_event_module_init(ngx_cycle_t *cycle)
{
    void              ***cf;
    u_char              *shared;
    size_t              size, cl;
    ngx_shm_t           shm;
    ngx_time_t          *tp;
    ngx_core_conf_t     *ccf;
    ngx_event_conf_t    *ecf;

    /* 获取存储所有事件模块配置结构的指针数据的首地址 */
    cf = ngx_get_conf(cycle->conf_ctx, ngx_events_module);
    /* 获取事件模块ngx_event_core_module的配置结构 */
    ecf = (*cf)[ngx_event_core_module.ctx_index];

    /* 在错误日志中输出被使用的事件模块名称 */
    if (!ngx_test_config && ngx_process <= NGX_PROCESS_MASTER) {
        ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
                      "using the \"%s\" event method", ecf->name);
    }
```

```
    /* 获取模块ngx_core_module的配置结构 */
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

    ngx_timer_resolution = ccf->timer_resolution;

#if !(NGX_WIN32)
    {
    ngx_int_t      limit;
    struct rlimit  rlmt;

    /* 获取当前进程所打开的最大文件描述符个数 */
    if (getrlimit(RLIMIT_NOFILE, &rlmt) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "getrlimit(RLIMIT_NOFILE) failed, ignored");

    } else {
        /*
         * 当前事件模块的连接数大于最大文件描述符个数，
         * 或者大于由配置文件nginx.conf指定的worker_rlinit_nofile设置的最大文件描述符个数时，
         * 出错返回；
         */
        if (ecf->connections > (ngx_uint_t) rlmt.rlim_cur
            && (ccf->rlimit_nofile == NGX_CONF_UNSET
                || ecf->connections > (ngx_uint_t) ccf->rlimit_nofile))
        {
            limit = (ccf->rlimit_nofile == NGX_CONF_UNSET) ?
                        (ngx_int_t) rlmt.rlim_cur : ccf->rlimit_nofile;

            ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
                          "%ui worker_connections exceed "
                          "open file resource limit: %i",
                          ecf->connections, limit);
        }
    }
    }
#endif /* !(NGX_WIN32) */

    /*
     * 模块ngx_core_module的master进程为0，表示不创建worker进程，
     * 则初始化到此结束，并成功返回；
     */
    if (ccf->master == 0) {
        return NGX_OK;
    }

    /*
     * 若master不为0，且存在负载均衡锁，则表示初始化完毕，并成功返回；
     */
    if (ngx_accept_mutex_ptr) {
        return NGX_OK;
    }

    /* 不满足以上两个条件，则初始化下列变量 */
    /* cl should be equal to or greater than cache line size */
```

```
    /* 缓存行的大小 */
    cl = 128;

    /*
     * 统计需要创建的共享内存大小；
     * ngx_accept_mutex用于多个worker进程之间的负载均衡锁；
     * ngx_connection_counter表示nginx处理的连接总数；
     * ngx_temp_number表示在连接中创建的临时文件个数；
     */
    size = cl            /* ngx_accept_mutex */
           + cl          /* ngx_connection_counter */
           + cl;         /* ngx_temp_number */

#if (NGX_STAT_STUB)

    /*
     * 下面表示某种情况的连接数；
     * ngx_stat_accepted    表示已成功建立的连接数；
     * ngx_stat_handled     表示已获取ngx_connection_t结构并已初始化读写事件的连接数；
     * ngx_stat_requests    表示已被http模块处理过的连接数；
     * ngx_stat_active      表示已获取ngx_connection_t结构体的连接数；
     * ngx_stat_reading     表示正在接收TCP字符流的连接数；
     * ngx_stat_writing     表示正在发送TCP字符流的连接数；
     * ngx_stat_waiting     表示正在等待事件发生的连接数；
     */
    size += cl           /* ngx_stat_accepted */
           + cl          /* ngx_stat_handled */
           + cl          /* ngx_stat_requests */
           + cl          /* ngx_stat_active */
           + cl          /* ngx_stat_reading */
           + cl          /* ngx_stat_writing */
           + cl;         /* ngx_stat_waiting */

#endif

    /* 初始化共享内存信息 */
    shm.size = size;
    shm.name.len = sizeof("nginx_shared_zone");
    shm.name.data = (u_char *) "nginx_shared_zone";
    shm.log = cycle->log;

    /* 创建共享内存 */
    if (ngx_shm_alloc(&shm) != NGX_OK) {
        return NGX_ERROR;
    }

    /* 获取共享内存的首地址 */
    shared = shm.addr;

    ngx_accept_mutex_ptr = (ngx_atomic_t *) shared;
    /* -1表示以非阻塞模式获取共享内存锁 */
    ngx_accept_mutex.spin = (ngx_uint_t) -1;

    if (ngx_shmtx_create(&ngx_accept_mutex, (ngx_shmtx_sh_t *) shared,
```

```
                        cycle->lock_file.data)
        != NGX_OK)
    {
        return NGX_ERROR;
    }

    /* 初始化变量 */
    ngx_connection_counter = (ngx_atomic_t *) (shared + 1 * cl);

    (void) ngx_atomic_cmp_set(ngx_connection_counter, 0, 1);

    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                   "counter: %p, %d",
                   ngx_connection_counter, *ngx_connection_counter);

    ngx_temp_number = (ngx_atomic_t *) (shared + 2 * cl);

    tp = ngx_timeofday();

    ngx_random_number = (tp->msec << 16) + ngx_pid;

#if (NGX_STAT_STUB)

    ngx_stat_accepted = (ngx_atomic_t *) (shared + 3 * cl);
    ngx_stat_handled = (ngx_atomic_t *) (shared + 4 * cl);
    ngx_stat_requests = (ngx_atomic_t *) (shared + 5 * cl);
    ngx_stat_active = (ngx_atomic_t *) (shared + 6 * cl);
    ngx_stat_reading = (ngx_atomic_t *) (shared + 7 * cl);
    ngx_stat_writing = (ngx_atomic_t *) (shared + 8 * cl);
    ngx_stat_waiting = (ngx_atomic_t *) (shared + 9 * cl);

#endif

    return NGX_OK;
}
```

ngx_event_process_init 方法在文件 src/event/ngx_event.c 中定义：

```
static ngx_int_t
ngx_event_process_init(ngx_cycle_t *cycle)
{
    ngx_uint_t           m, i;
    ngx_event_t         *rev, *wev;
    ngx_listening_t     *ls;
    ngx_connection_t    *c, *next, *old;
    ngx_core_conf_t     *ccf;
    ngx_event_conf_t    *ecf;
    ngx_event_module_t  *module;

    /* 获取ngx_core_module核心模块的配置结构 */
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
    /* 获取ngx_event_core_module事件核心模块的配置结构 */
    ecf = ngx_event_get_conf(cycle->conf_ctx, ngx_event_core_module);
```

```
    /*
     * 在事件核心模块启用accept_mutex锁的情况下，
     * 只有在master-worker工作模式并且worker进程数量大于1，
     * 此时，才确定进程启用负载均衡锁；
     */
    if (ccf->master && ccf->worker_processes > 1 && ecf->accept_mutex) {
        ngx_use_accept_mutex = 1;
        ngx_accept_mutex_held = 0;
        ngx_accept_mutex_delay = ecf->accept_mutex_delay;

    } else {/* 否则关闭负载均衡锁 */
        ngx_use_accept_mutex = 0;
    }

#if (NGX_WIN32)

    /*
     * disable accept mutex on win32 as it may cause deadlock if
     * grabbed by a process which can't accept connections
     */

    ngx_use_accept_mutex = 0;

#endif

    ngx_queue_init(&ngx_posted_accept_events);
    ngx_queue_init(&ngx_posted_events);

    /* 初始化由红黑树实现的定时器 */
    if (ngx_event_timer_init(cycle->log) == NGX_ERROR) {
        return NGX_ERROR;
    }

    /* 根据use配置项所指定的事件模块，调用ngx_actions_t中的init方法初始化事件模块 */
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
            continue;
        }

        if (ngx_modules[m]->ctx_index != ecf->use) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->actions.init(cycle, ngx_timer_resolution) != NGX_OK) {
            /* fatal */
            exit(2);
        }

        break;
    }

#if !(NGX_WIN32)
```

```
    /*
     * NGX_USE_TIMER_EVENT只有在eventport和kqueue事件模型中使用，
     * 若配置文件nginx.conf设置了timer_resolution配置项，
     * 并且事件模型不为eventport和kqueue时，调用settimer方法，
     */
    if (ngx_timer_resolution && !(ngx_event_flags & NGX_USE_TIMER_EVENT)) {
        struct sigaction  sa;
        struct itimerval  itv;

        ngx_memzero(&sa, sizeof(struct sigaction));
        /*
         * ngx_timer_signal_handler的实现如下：
         * void ngx_timer_signal_handler(int signo)
         * {
         *      ngx_event_timer_alarm = 1;
         * }
         * ngx_event_timer_alarm 为1时表示需要更新系统时间，即调用ngx_time_update方法；
         * 更新完系统时间之后，该变量设为0；
         */
        /* 指定信号处理函数 */
        sa.sa_handler = ngx_timer_signal_handler;
        /* 初始化信号集 */
        sigemptyset(&sa.sa_mask);

        /* 捕获信号SIGALRM */
        if (sigaction(SIGALRM, &sa, NULL) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "sigaction(SIGALRM) failed");
            return NGX_ERROR;
        }

        /* 设置时间精度 */
        itv.it_interval.tv_sec = ngx_timer_resolution / 1000;
        itv.it_interval.tv_usec = (ngx_timer_resolution % 1000) * 1000;
        itv.it_value.tv_sec = ngx_timer_resolution / 1000;
        itv.it_value.tv_usec = (ngx_timer_resolution % 1000 ) * 1000;

        /* 使用settimer函数发送信号 SIGALRM */
        if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "setitimer() failed");
        }
    }

    /* 对poll、/dev/poll、rtsig事件模块的特殊处理 */
    if (ngx_event_flags & NGX_USE_FD_EVENT) {
        struct rlimit  rlmt;

        if (getrlimit(RLIMIT_NOFILE, &rlmt) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "getrlimit(RLIMIT_NOFILE) failed");
            return NGX_ERROR;
        }
```

```
        cycle->files_n = (ngx_uint_t) rlmt.rlim_cur;

        cycle->files = ngx_calloc(sizeof(ngx_connection_t *) * cycle->files_n,
                                  cycle->log);
        if (cycle->files == NULL) {
            return NGX_ERROR;
        }
    }

#endif

    /* 预分配连接池 */
    cycle->connections =
        ngx_alloc(sizeof(ngx_connection_t) * cycle->connection_n, cycle->log);
    if (cycle->connections == NULL) {
        return NGX_ERROR;
    }

    c = cycle->connections;

    /* 预分配读事件结构，读事件个数与连接数相同 */
    cycle->read_events = ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n,
                                   cycle->log);
    if (cycle->read_events == NULL) {
        return NGX_ERROR;
    }

    rev = cycle->read_events;
    for (i = 0; i < cycle->connection_n; i++) {
        rev[i].closed = 1;
        rev[i].instance = 1;
    }

    /* 预分配写事件结构，写事件个数与连接数相同 */
    cycle->write_events = ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n,
                                    cycle->log);
    if (cycle->write_events == NULL) {
        return NGX_ERROR;
    }

    wev = cycle->write_events;
    for (i = 0; i < cycle->connection_n; i++) {
        wev[i].closed = 1;
    }

    i = cycle->connection_n;
    next = NULL;

    /* 按照序号，将读、写事件与连接对象对应，即设置到每个ngx_connection_t 对象中 */
    do {
        i--;

        c[i].data = next;
        c[i].read = &cycle->read_events[i];
```

```
        c[i].write = &cycle->write_events[i];
        c[i].fd = (ngx_socket_t) -1;

        next = &c[i];

#if (NGX_THREADS)
        c[i].lock = 0;
#endif
    } while (i);

    /* 设置空闲连接链表 */
    cycle->free_connections = next;
    cycle->free_connection_n = cycle->connection_n;

    /* for each listening socket */

    /* 为所有ngx_listening_t监听对象中的connections成员分配连接，并设置读事件的处理方法 */
    ls = cycle->listening.elts;
    for (i = 0; i < cycle->listening.nelts; i++) {

        /* 为监听套接字分配连接，并设置读事件 */
        c = ngx_get_connection(ls[i].fd, cycle->log);

        if (c == NULL) {
            return NGX_ERROR;
        }

        c->log = &ls[i].log;

        c->listening = &ls[i];
        ls[i].connection = c;

        rev = c->read;

        rev->log = c->log;
        rev->accept = 1;

#if (NGX_HAVE_DEFERRED_ACCEPT)
        rev->deferred_accept = ls[i].deferred_accept;
#endif

        if (!(ngx_event_flags & NGX_USE_IOCP_EVENT)) {
            if (ls[i].previous) {

                /*
                 * delete the old accept events that were bound to
                 * the old cycle read events array
                 */

                old = ls[i].previous->connection;

                if (ngx_del_event(old->read, NGX_READ_EVENT, NGX_CLOSE_EVENT)
                    == NGX_ERROR)
                {
                    return NGX_ERROR;
```

```
                return NGX_ERROR;
            }

            old->fd = (ngx_socket_t) -1;
        }
    }

#if (NGX_WIN32)

    if (ngx_event_flags & NGX_USE_IOCP_EVENT) {
        ngx_iocp_conf_t  *iocpcf;

        rev->handler = ngx_event_acceptex;

        if (ngx_use_accept_mutex) {
            continue;
        }

        if (ngx_add_event(rev, 0, NGX_IOCP_ACCEPT) == NGX_ERROR) {
            return NGX_ERROR;
        }

        ls[i].log.handler = ngx_acceptex_log_error;

        iocpcf = ngx_event_get_conf(cycle->conf_ctx, ngx_iocp_module);
        if (ngx_event_post_acceptex(&ls[i], iocpcf->post_acceptex)
            == NGX_ERROR)
        {
            return NGX_ERROR;
        }

    } else {
        rev->handler = ngx_event_accept;

        if (ngx_use_accept_mutex) {
            continue;
        }

        if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
            return NGX_ERROR;
        }
    }

#else

    /* 为监听端口的读事件设置处理方法ngx_event_accept */
    rev->handler = ngx_event_accept;

    if (ngx_use_accept_mutex) {
        continue;
    }

    if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
        if (ngx_add_conn(c) == NGX_ERROR) {
            return NGX_ERROR;
```

```
            }

        } else {
        /* 将监听对象连接的读事件添加到事件驱动模块中 */
            if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
                return NGX_ERROR;
            }
        }

 #endif

    }

    return NGX_OK;
}
```

参考资料：

《深入理解 Nginx 》

《nginx事件模块分析(二)》

Nginx 的 epoll 事件驱动模块

概述

在前面的文章中《 Nginx 事件模块》介绍了Nginx 的事件驱动框架以及不同类型事件驱动模块的管理。本节基于前面的知识，简单介绍下在Linux 系统下的 epoll 事件驱动模块。关于 epoll 的使用与原理可以参照文章 《epoll 解析》。在这里直接介绍Nginx 服务器基于事件驱动框架实现的epoll 事件驱动模块。

ngx_epoll_module 事件驱动模块

ngx_epoll_conf_t 结构体

ngx_epoll_conf_t 结构体是保存ngx_epoll_module 事件驱动模块的配置项结构。该结构体在文件src/event/modules/ngx_epoll_module.c 中定义：

```
/* 存储epoll模块配置项结构体 */
typedef struct {
    ngx_uint_t  events;         /* 表示epoll_wait函数返回的最大事件数 */
    ngx_uint_t  aio_requests;   /* 并发处理异步IO事件个数 */
} ngx_epoll_conf_t;
```

ngx_epoll_module 事件驱动模块的定义

所有模块的定义都是基于模块通用接口 ngx_module_t 结构，ngx_epoll_module 模块在文件src/event/modules/ngx_epoll_module.c 中定义如下：

```
/* epoll模块定义 */
ngx_module_t  ngx_epoll_module = {
    NGX_MODULE_V1,
    &ngx_epoll_module_ctx,              /* module context */
    ngx_epoll_commands,                 /* module directives */
    NGX_EVENT_MODULE,                   /* module type */
    NULL,                               /* init master */
    NULL,                               /* init module */
    NULL,                               /* init process */
    NULL,                               /* init thread */
    NULL,                               /* exit thread */
    NULL,                               /* exit process */
    NULL,                               /* exit master */
    NGX_MODULE_V1_PADDING
};
```

在 ngx_epoll_module 模块的定义中，其中定义了该模块感兴趣的配置项ngx_epoll_commands 数

组，该配置项数组在文件src/event/modules/ngx_epoll_module.c 中定义：

```
/* 定义epoll模块感兴趣的配置项结构数组 */
static ngx_command_t  ngx_epoll_commands[] = {

    /*
     * epoll_events配置项表示epoll_wait函数每次返回的最多事件数(即第3个参数)，
     * 在ngx_epoll_init函数中会预分配epoll_events配置项指定的epoll_event结构体；
     */
    { ngx_string("epoll_events"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_num_slot,
      0,
      offsetof(ngx_epoll_conf_t, events),
      NULL },

    /*
     * 该配置项表示创建的异步IO上下文能并发处理异步IO事件的个数，
     * 即io_setup函数的第一个参数；
     */
    { ngx_string("worker_aio_requests"),
      NGX_EVENT_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_num_slot,
      0,
      offsetof(ngx_epoll_conf_t, aio_requests),
      NULL },

      ngx_null_command
};
```

在 ngx_epoll_module 模块的定义中，定义了该模块的上下文结构ngx_epoll_module_ctx，该上下文结构是基于事件模块的通用接口ngx_event_module_t 结构来定义的。该上下文结构在文件src/event/modules/ngx_epoll_module.c 中定义：

```
/* 由事件模块通用接口ngx_event_module_t定义的epoll模块上下文结构 */
ngx_event_module_t  ngx_epoll_module_ctx = {
    &epoll_name,
    ngx_epoll_create_conf,               /* create configuration */
    ngx_epoll_init_conf,                 /* init configuration */

    {
        ngx_epoll_add_event,             /* add an event */
        ngx_epoll_del_event,             /* delete an event */
        ngx_epoll_add_event,             /* enable an event */
        ngx_epoll_del_event,             /* disable an event */
        ngx_epoll_add_connection,        /* add an connection */
        ngx_epoll_del_connection,        /* delete an connection */
        NULL,                            /* process the changes */
        ngx_epoll_process_events,        /* process the events */
        ngx_epoll_init,                  /* init the events */
        ngx_epoll_done,                  /* done the events */
    }
};
```

在 ngx_epoll_module 模块的上下文事件接口结构中，重点定义了ngx_event_actions_t 结构中的接口回调方法。

ngx_epoll_module 事件驱动模块的操作

ngx_epoll_module 模块的操作由ngx_epoll_module 模块的上下文事件接口结构中成员actions 实现。该成员实现的方法如下所示：

```
        ngx_epoll_add_event,             /* add an event */
        ngx_epoll_del_event,             /* delete an event */
        ngx_epoll_add_event,             /* enable an event */
        ngx_epoll_del_event,             /* disable an event */
        ngx_epoll_add_connection,        /* add an connection */
        ngx_epoll_del_connection,        /* delete an connection */
        NULL,                            /* process the changes */
        ngx_epoll_process_events,        /* process the events */
        ngx_epoll_init,                  /* init the events */
        ngx_epoll_done,                  /* done the events */
```

ngx_epoll_module 模块的初始化

ngx_epoll_module 模块的初始化由函数ngx_epoll_init 实现。该函数主要做了两件事：创建epoll 对象 和 创建 event_list 数组（调用epoll_wait 函数时用于存储从内核复制的已就绪的事件）；该函数在文件src/event/modules/ngx_epoll_module.c 中定义：

```
static int                ep = -1;    /* epoll对象描述符 */
static struct epoll_event  *event_list; /* 作为epoll_wait函数的第二个参数，保存从内存复制的事件 */
static ngx_uint_t          nevents;    /* epoll_wait函数返回的最多事件数 */
```

```
/* epoll模块初始化函数 */
static ngx_int_t
ngx_epoll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
{
    ngx_epoll_conf_t  *epcf;

    /* 获取ngx_epoll_module模块的配置项结构 */
    epcf = ngx_event_get_conf(cycle->conf_ctx, ngx_epoll_module);

    if (ep == -1) {
        /* 调用epoll_create函数创建epoll对象描述符 */
        ep = epoll_create(cycle->connection_n / 2);

        /* 若创建失败，则出错返回 */
        if (ep == -1) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "epoll_create() failed");
            return NGX_ERROR;
        }

#if (NGX_HAVE_FILE_AIO)

        /* 若系统支持异步IO，则初始化异步IO */
        ngx_epoll_aio_init(cycle, epcf);

#endif
    }

    /*
     * 预分配events个epoll_event结构event_list，event_list是存储产生事件的数组；
     * events由epoll_events配置项指定；
     */
    if (nevents < epcf->events) {
        /*
         * 若现有event_list个数小于配置项所指定的值epcf->events，
         * 则先释放，再从新分配；
         */
        if (event_list) {
            ngx_free(event_list);
        }

        /* 预分配epcf->events个epoll_event结构，并使event_list指向该地址 */
        event_list = ngx_alloc(sizeof(struct epoll_event) * epcf->events,
                               cycle->log);
        if (event_list == NULL) {
            return NGX_ERROR;
        }
    }

    /* 设置正确的epoll_event结构个数 */
    nevents = epcf->events;

    /* 指定IO的读写方法 */
    /*
```

```
    * 初始化全局变量ngx_io, ngx_os_io定义为:
      ngx_os_io_t ngx_os_io = {
          ngx_unix_recv,
          ngx_readv_chain,
          ngx_udp_unix_recv,
          ngx_unix_send,
          ngx_writev_chain,
          0
      }; ( 位于src/os/unix/ngx_posix_init.c )
    */
    ngx_io = ngx_os_io;

    /* 设置ngx_event_actions 接口 */
    ngx_event_actions = ngx_epoll_module_ctx.actions;

#if (NGX_HAVE_CLEAR_EVENT)
    /* ET模式 */
    ngx_event_flags = NGX_USE_CLEAR_EVENT
#else
    /* LT模式 */
    ngx_event_flags = NGX_USE_LEVEL_EVENT
#endif
                      |NGX_USE_GREEDY_EVENT
                      |NGX_USE_EPOLL_EVENT;

    return NGX_OK;
}
```

ngx_epoll_module 模块的事件处理

ngx_epoll_module 模块的事件处理由函数ngx_epoll_process_events 实现。ngx_epoll_process_events 函数是实现事件收集、事件发送的接口。该函数在文件src/event/modules/ngx_epoll_module.c 中定义：

```
/* 处理已准备就绪的事件 */
static ngx_int_t
ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
{
    int               events;
    uint32_t          revents;
    ngx_int_t         instance, i;
    ngx_uint_t        level;
    ngx_err_t         err;
    ngx_event_t       *rev, *wev, **queue;
    ngx_connection_t  *c;

    /* NGX_TIMER_INFINITE == INFTIM */

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                   "epoll timer: %M", timer);

    /* 调用epoll_wait在规定的timer时间内等待监控的事件准备就绪 */
    events = epoll_wait(ep, event_list, (int) nevents, timer);
```

```
    events = epoll_wait(ep, event_list, (int) nevents, timer);

    /* 若出错，设置错误编码 */
    err = (events == -1) ? ngx_errno : 0;

    /*
     * 若没有设置timer_resolution配置项时，
     * NGX_UPDATE_TIME 标志表示每次调用epoll_wait函数返回后需要更新时间；
     * 若设置timer_resolution配置项，
     * 则每隔timer_resolution配置项参数会设置ngx_event_timer_alarm为1，表示需要更新时间；
     */
    if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
        /* 更新时间，将时间缓存到一组全局变量中，方便程序高效获取事件 */
        ngx_time_update();
    }

    /* 处理epoll_wait的错误 */
    if (err) {
        if (err == NGX_EINTR) {

            if (ngx_event_timer_alarm) {
                ngx_event_timer_alarm = 0;
                return NGX_OK;
            }

            level = NGX_LOG_INFO;

        } else {
            level = NGX_LOG_ALERT;
        }

        ngx_log_error(level, cycle->log, err, "epoll_wait() failed");
        return NGX_ERROR;
    }

    /*
     * 若epoll_wait返回的事件数events为0，则有两种可能：
     * 1、超时返回，即时间超过timer；
     * 2、在限定的timer时间内返回，此时表示出错error返回；
     */
    if (events == 0) {
        if (timer != NGX_TIMER_INFINITE) {
            return NGX_OK;
        }

        ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
                      "epoll_wait() returned no events without timeout");
        return NGX_ERROR;
    }

    /* 仅在多线程环境下有效 */
    ngx_mutex_lock(ngx_posted_events_mutex);

    /* 遍历由epoll_wait返回的所有已准备就绪的事件，并处理这些事件 */
    for (i = 0; i < events; i++) {
```

```
        /*
         * 获取与事件关联的连接对象；
         * 连接对象地址的最低位保存的是添加事件时设置的事件过期标志位；
         */
        c = event_list[i].data.ptr;

        /* 获取事件过期标志位，即连接对象地址的最低位 */
        instance = (uintptr_t) c & 1;
        /* 屏蔽连接对象的最低位，即获取连接对象的真正地址 */
        c = (ngx_connection_t *) ((uintptr_t) c & (uintptr_t) ~1);

        /* 获取读事件 */
        rev = c->read;

        /*
         * 同一连接的读写事件的instance标志位是相同的；
         * 若fd描述符为-1，或连接对象读事件的instance标志位不相同，则判为过期事件；
         */
        if (c->fd == -1 || rev->instance != instance) {

            /*
             * the stale event from a file descriptor
             * that was just closed in this iteration
             */

            ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                           "epoll: stale event %p", c);
            continue;
        }

        /* 获取连接对象中已准备就绪的事件类型 */
        revents = event_list[i].events;

        ngx_log_debug3(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                       "epoll: fd:%d ev:%04XD d:%p",
                       c->fd, revents, event_list[i].data.ptr);
        /* 记录epoll_wait的错误返回状态 */
        /*
         * EPOLLERR表示连接出错；EPOLLHUP表示收到RST报文；
         * 检测到上面这两种错误时，TCP连接中可能存在未读取的数据；
         */
        if (revents & (EPOLLERR|EPOLLHUP)) {
            ngx_log_debug2(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                           "epoll_wait() error on fd:%d ev:%04XD",
                           c->fd, revents);
        }

    #if 0
        if (revents & ~(EPOLLIN|EPOLLOUT|EPOLLERR|EPOLLHUP)) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
                          "strange epoll_wait() events fd:%d ev:%04XD",
                          c->fd, revents);
        }
    #endif
```

```
        /*
         * 若连接发生错误且未设置EPOLLIN、EPOLLOUT，
         * 则将EPOLLIN、EPOLLOUT添加到revents中；
         * 即在调用读写事件时能够处理连接的错误；
         */
        if ((revents & (EPOLLERR|EPOLLHUP))
             && (revents & (EPOLLIN|EPOLLOUT)) == 0)
        {
            /*
             * if the error events were returned without EPOLLIN or EPOLLOUT,
             * then add these flags to handle the events at least in one
             * active handler
             */

            revents |= EPOLLIN|EPOLLOUT;
        }

        /* 连接有可读事件，且该读事件是active活跃的 */
        if ((revents & EPOLLIN) && rev->active) {

#if (NGX_HAVE_EPOLLRDHUP)
            /* EPOLLRDHUP表示连接对端关闭了读取端 */
            if (revents & EPOLLRDHUP) {
                rev->pending_eof = 1;
            }
#endif

            if ((flags & NGX_POST_THREAD_EVENTS) && !rev->accept) {
                rev->posted_ready = 1;

            } else {
                /* 读事件已准备就绪 */
                /*
                 * 这里要区分active与ready：
                 * active是指事件被添加到epoll对象的监控中，
                 * 而ready表示被监控的事件已经准备就绪，即可以对其进程IO处理；
                 */
                rev->ready = 1;
            }

            /*
             * NGX_POST_EVENTS表示已准备就绪的事件需要延迟处理，
             * 根据accept标志位将事件加入到相应的队列中；
             */
            if (flags & NGX_POST_EVENTS) {
                queue = (ngx_event_t **) (rev->accept ?
                            &ngx_posted_accept_events : &ngx_posted_events);

                ngx_locked_post_event(rev, queue);

            } else {
                /* 若不延迟处理，则直接调用事件的处理函数 */
                rev->handler(rev);
            }
        }
```

```
        }

        /* 获取连接的写事件，写事件的处理逻辑过程与读事件类似 */
        wev = c->write;

        /* 连接有可写事件，且该写事件是active活跃的 */
        if ((revents & EPOLLOUT) && wev->active) {

            /* 检查写事件是否过期 */
            if (c->fd == -1 || wev->instance != instance) {

                /*
                 * the stale event from a file descriptor
                 * that was just closed in this iteration
                 */

                ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                               "epoll: stale event %p", c);
                continue;
            }

            if (flags & NGX_POST_THREAD_EVENTS) {
                wev->posted_ready = 1;

            } else {

                /* 写事件已准备就绪 */
                wev->ready = 1;
            }

            /*
             * NGX_POST_EVENTS表示已准备就绪的事件需要延迟处理，
             * 根据accept标志位将事件加入到相应的队列中；
             */
            if (flags & NGX_POST_EVENTS) {
                ngx_locked_post_event(wev, &ngx_posted_events);

            } else {
                /* 若不延迟处理，则直接调用事件的处理函数 */
                wev->handler(wev);
            }
        }
    }

    ngx_mutex_unlock(ngx_posted_events_mutex);

    return NGX_OK;
}
```

ngx_epoll_module 模块的事件添加与删除

ngx_epoll_module 模块的事件添加与删除分别由函数ngx_epoll_add_event 与ngx_epoll_del_event 实现。这两个函数的实现都是通过调用epoll_ctl 函数。具体实现在文件

src/event/modules/ngx_epoll_module.c 中定义：

```
/* 将某个描述符的某个事件添加到epoll对象的监控机制中 */
static ngx_int_t
ngx_epoll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
{
    int                op;
    uint32_t           events, prev;
    ngx_event_t        *e;
    ngx_connection_t   *c;
    struct epoll_event  ee;

    /* 每个事件的data成员都存放着其对应的ngx_connection_t连接 */
    /* 获取事件关联的连接 */
    c = ev->data;

    /* events参数是方便下面确定当前事件是可读还是可写 */
    events = (uint32_t) event;

    /*
     * 这里在判断事件类型是可读还是可写，必须根据事件的active标志位来判断事件是否活跃；
     * 因为epoll_ctl函数有添加add和修改mod模式，
     * 若一个事件所关联的连接已经在epoll对象的监控中，则只需修改事件的类型即可；
     * 若一个事件所关联的连接没有在epoll对象的监控中，则需要将其相应的事件类型注册到epoll对象中；
     * 这样做的情况是避免与事件相关联的连接两次注册到epoll对象中；
     */

    if (event == NGX_READ_EVENT) {
        /*
         * 若待添加的事件类型event是可读；
         * 则首先判断该事件所关联的连接是否将写事件添加到epoll对象中，
         * 即先判断关联的连接的写事件是否为活跃事件；
         */
        e = c->write;
        prev = EPOLLOUT;
#if (NGX_READ_EVENT != EPOLLIN|EPOLLRDHUP)
        events = EPOLLIN|EPOLLRDHUP;
#endif

    } else {
        e = c->read;
        prev = EPOLLIN|EPOLLRDHUP;
#if (NGX_WRITE_EVENT != EPOLLOUT)
        events = EPOLLOUT;
#endif
    }

    /* 根据active标志位确定事件是否为活跃事件，以决定到达是修改还是添加事件 */
    if (e->active) {
        /* 若当前事件是活跃事件，则只需修改其事件类型即可 */
        op = EPOLL_CTL_MOD;
        events |= prev;

    } else {
```

```
        /* 若当前事件不是活跃事件，则将该事件添加到epoll对象中 */
        op = EPOLL_CTL_ADD;
    }

    /* 将flags参数加入到events标志位中 */
    ee.events = events | (uint32_t) flags;
    /* prt存储事件关联的连接对象ngx_connection_t以及过期事件instance标志位 */
    ee.data.ptr = (void *) ((uintptr_t) c | ev->instance);

    ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                   "epoll add event: fd:%d op:%d ev:%08XD",
                   c->fd, op, ee.events);

    /* 调用epoll_ctl方法向epoll对象添加事件或在epoll对象中修改事件 */
    if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
        ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                      "epoll_ctl(%d, %d) failed", op, c->fd);
        return NGX_ERROR;
    }

    /* 将该事件的active标志位设置为1，表示当前事件是活跃事件 */
    ev->active = 1;
#if 0
    ev->oneshot = (flags & NGX_ONESHOT_EVENT) ? 1 : 0;
#endif

    return NGX_OK;
}

/* 将某个连接的某个事件从epoll对象监控中删除 */
static ngx_int_t
ngx_epoll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
{
    int                  op;
    uint32_t             prev;
    ngx_event_t         *e;
    ngx_connection_t    *c;
    struct epoll_event   ee;

    /*
     * when the file descriptor is closed, the epoll automatically deletes
     * it from its queue, so we do not need to delete explicitly the event
     * before the closing the file descriptor
     */

    /* 当事件关联的文件描述符关闭后，epoll对象自动将其事件删除 */
    if (flags & NGX_CLOSE_EVENT) {
        ev->active = 0;
        return NGX_OK;
    }

    /* 获取事件关联的连接对象 */
    c = ev->data;
```

```
    /* 根据event参数判断当前删除的是读事件还是写事件 */
    if (event == NGX_READ_EVENT) {
        /* 若要删除读事件，则首先判断写事件的active标志位 */
        e = c->write;
        prev = EPOLLOUT;

    } else {
        /* 若要删除写事件，则判断读事件的active标志位 */
        e = c->read;
        prev = EPOLLIN|EPOLLRDHUP;
    }

    /*
     * 若要删除读事件，且写事件是活跃事件，则修改事件类型即可；
     * 若要删除写事件，且读事件是活跃事件，则修改事件类型即可；
     */
    if (e->active) {
        op = EPOLL_CTL_MOD;
        ee.events = prev | (uint32_t) flags;
        ee.data.ptr = (void *) ((uintptr_t) c | ev->instance);

    } else {
        /* 若读写事件都不是活跃事件，此时表示事件未准备就绪，则将其删除 */
        op = EPOLL_CTL_DEL;
        ee.events = 0;
        ee.data.ptr = NULL;
    }

    ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                   "epoll del event: fd:%d op:%d ev:%08XD",
                   c->fd, op, ee.events);

    /* 删除或修改事件 */
    if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
        ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                      "epoll_ctl(%d, %d) failed", op, c->fd);
        return NGX_ERROR;
    }

    /* 设置当前事件的active标志位 */
    ev->active = 0;

    return NGX_OK;
}
```

ngx_epoll_module 模块的连接添加与删除

ngx_epoll_module 模块的连接添加与删除分别由函数ngx_epoll_add_connection 与 ngx_epoll_del_connection 实现。这两个函数的实现都是通过调用epoll_ctl 函数。具体实现在文件 src/event/modules/ngx_epoll_module.c 中定义：

```
/* 将指定连接所关联的描述符添加到epoll对象中 */
static ngx_int_t
```

```
static ngx_int_t
ngx_epoll_add_connection(ngx_connection_t *c)
{
    struct epoll_event  ee;

    /* 设置事件的类型：可读、可写、ET模式 */
    ee.events = EPOLLIN|EPOLLOUT|EPOLLET|EPOLLRDHUP;
    ee.data.ptr = (void *) ((uintptr_t) c | c->read->instance);

    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
                   "epoll add connection: fd:%d ev:%08XD", c->fd, ee.events);

    /* 调用epoll_ctl方法将连接所关联的描述符添加到epoll对象中 */
    if (epoll_ctl(ep, EPOLL_CTL_ADD, c->fd, &ee) == -1) {
        ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
                      "epoll_ctl(EPOLL_CTL_ADD, %d) failed", c->fd);
        return NGX_ERROR;
    }

    /* 设置读写事件的active标志位 */
    c->read->active = 1;
    c->write->active = 1;

    return NGX_OK;
}
Nginx

/* 将连接所关联的描述符从epoll对象中删除 */
static ngx_int_t
ngx_epoll_del_connection(ngx_connection_t *c, ngx_uint_t flags)
{
    int                 op;
    struct epoll_event  ee;

    /*
     * when the file descriptor is closed the epoll automatically deletes
     * it from its queue so we do not need to delete explicitly the event
     * before the closing the file descriptor
     */

    if (flags & NGX_CLOSE_EVENT) {
        c->read->active = 0;
        c->write->active = 0;
        return NGX_OK;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_EVENTNginx, c->log, 0,
                   "epoll del connection: fd:%d", c->fd);

    op = EPOLL_CTL_DEL;
    ee.events = 0;
    ee.data.ptr = NULL;

    /* 调用epoll_ctl方法将描述符从epoll对象中删除 */
    if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
```

```
        ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
                      "epoll_ctl(%d, %d) failed", op, c->fd);
        return NGX_ERROR;
    }

    /* 设置描述符读写事件的active标志位 */
    c->read->active = 0;
    c->write->active = 0;

    return NGX_OK;
}
```

ngx_epoll_module 模块的异步 I/O

在 Nginx 中，文件异步 I/O 事件完成后的通知是集成到 epoll 对象中。该模块的文件异步I/O 实现如下：

```
#if (NGX_HAVE_FILE_AIO)

int                     ngx_eventfd = -1;   /* 用于通知异步IO的事件描述符 */
aio_context_t           ngx_aio_ctx = 0;    /* 异步IO的上下文结构，由io_setup 函数初始化 */

static ngx_event_t      ngx_eventfd_event;  /* 异步IO事件 */
static ngx_connection_t ngx_eventfd_conn;   /* 异步IO事件所对应的连接ngx_connection_t */

#endif

#if (NGX_HAVE_FILE_AIO)

/*
 * We call io_setup(), io_destroy() io_submit(), and io_getevents() directly
 * as syscalls instead of libaio usage, because the library header file
 * supports eventfd() since 0.3.107 version only.
 *
 * Also we do not use eventfd() in glibc, because glibc supports it
 * since 2.8 version and glibc maps two syscalls eventfd() and eventfd2()
 * into single eventfd() function with different number of parameters.
 */

/* 初始化文件异步IO的上下文结构 */
static int
io_setup(u_int nr_reqs, aio_context_t *ctx)
{
    return syscall(SYS_io_setup, nr_reqs, ctx);
}

/* 销毁文件异步IO的上下文结构 */
static int
io_destroy(aio_context_t ctx)
{
    return syscall(SYS_io_destroy, ctx);
}
```

```
/* 从文件异步IO操作队列中读取操作 */
static int
io_getevents(aio_context_t ctx, long min_nr, long nr, struct io_event *events,
    struct timespec *tmo)
{
    return syscall(SYS_io_getevents, ctx, min_nr, nr, events, tmo);
}

/* 异步IO的初始化 */
static void
ngx_epoll_aio_init(ngx_cycle_t *cycle, ngx_epoll_conf_t *epcf)
{
    int                 n;
    struct epoll_event  ee;

    /* 使用Linux系统调用获取一个描述符句柄 */
    ngx_eventfd = syscall(SYS_eventfd, 0);

    if (ngx_eventfd == -1) {
        ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                      "eventfd() failed");
        ngx_file_aio = 0;
        return;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                   "eventfd: %d", ngx_eventfd);

    n = 1;

    /* 设置ngx_eventfd描述符句柄为非阻塞IO模式 */
    if (ioctl(ngx_eventfd, FIONBIO, &n) == -1) {
        ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                      "ioctl(eventfd, FIONBIO) failed");
        goto failed;
    }

    /* 初始化文件异步IO的上下文结构 */
    if (io_setup(epcf->aio_requests, &ngx_aio_ctx) == -1) {
        ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                      "io_setup() failed");
        goto failed;
    }

    /* 设置异步IO事件ngx_eventfd_event，该事件是ngx_eventfd对应的ngx_event事件 */

    /* 用于异步IO完成通知的ngx_eventfd_event事件，它与ngx_eventfd_conn连接对应 */
    ngx_eventfd_event.data = &ngx_eventfd_conn;
    /* 在异步IO事件完成后，调用ngx_epoll_eventfd_handler处理方法 */
    ngx_eventfd_event.handler = ngx_epoll_eventfd_handler;
    /* 设置事件相应的日志 */
    ngx_eventfd_event.log = cycle->log;
    /* 设置active标志位 */
    ngx_eventfd_event.active = 1;
```

```
    /* 初始化ngx_eventfd_conn 连接 */
    ngx_eventfd_conn.fd = ngx_eventfd;
    /* ngx_eventfd_conn连接的读事件就是ngx_eventfd_event事件 */
    ngx_eventfd_conn.read = &ngx_eventfd_event;
    /* 设置连接的相应日志 */
    ngx_eventfd_conn.log = cycle->log;

    ee.events = EPOLLIN|EPOLLET;
    ee.data.ptr = &ngx_eventfd_conn;

    /* 向epoll对象添加异步IO通知描述符ngx_eventfd */
    if (epoll_ctl(ep, EPOLL_CTL_ADD, ngx_eventfd, &ee) != -1) {
        return;
    }

    /* 若添加出错，则销毁文件异步IO上下文结构，并返回 */
    ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                  "epoll_ctl(EPOLL_CTL_ADD, eventfd) failed");

    if (io_destroy(ngx_aio_ctx) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "io_destroy() failed");
    }

failed:

    if (close(ngx_eventfd) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "eventfd close() failed");
    }

    ngx_eventfd = -1;
    ngx_aio_ctx = 0;
    ngx_file_aio = 0;
}

#endif

#if (NGX_HAVE_FILE_AIO)

/* 处理已完成的异步IO事件 */
static void
ngx_epoll_eventfd_handler(ngx_event_t *ev)
{
    int               n, events;
    long              i;
    uint64_t          ready;
    ngx_err_t         err;
    ngx_event_t      *e;
    ngx_event_aio_t  *aio;
    struct io_event   event[64];    /* 一次最多处理64个事件 */
    struct timespec   ts;

    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, 0, "eventfd handler");
```

```
    /* 获取已完成的事件数，并将其设置到ready */
    n = read(ngx_eventfd, &ready, 8);

    err = ngx_errno;

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0, "eventfd: %d", n);

    if (n != 8) {
        if (n == -1) {
            if (err == NGX_EAGAIN) {
                return;
            }

            ngx_log_error(NGX_LOG_ALERT, ev->log, err, "read(eventfd) failed");
            return;
        }

        ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
                      "read(eventfd) returned only %d bytes", n);
        return;
    }

    ts.tv_sec = 0;
    ts.tv_nsec = 0;

    /* 遍历ready，处理异步IO事件 */
    while (ready) {

        /* 获取已完成的异步IO事件 */
        events = io_getevents(ngx_aio_ctx, 1, 64, event, &ts);

        ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                       "io_getevents: %l", events);

        if (events > 0) {
            ready -= events;/* ready减去已经取出的事件数 */

            /* 处理已被取出的事件 */
            for (i = 0; i < events; i++) {

                ngx_log_debug4(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                               "io_event: %uXL %uXL %L %L",
                               event[i].data, event[i].obj,
                               event[i].res, event[i].res2);

                /* 获取异步IO事件对应的实际事件 */
                e = (ngx_event_t *) (uintptr_t) event[i].data;

                e->complete = 1;
                e->active = 0;
                e->ready = 1;

                aio = e->data;
                aio->res = event[i].res;
```

```
                /* 将实际事件加入到ngx_posted_event队列中等待处理 */
                ngx_post_event(e, &ngx_posted_events);
            }

            continue;
        }

        if (events == 0) {
            return;
        }

        /* events == -1 */
        ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                      "io_getevents() failed");
        return;
    }
}

#endif
```

参考资料：

《深入理解Nginx》

《模块ngx_epoll_module详解》

《 nginx epoll详解》

《Nginx源码分析-Epoll模块》

《关于ngx_epoll_add_event的一些解释》

Nginx 定时器事件

概述

在 Nginx 中定时器事件的实现与内核无关。在事件模块中，当等待的事件不能在指定的时间内到达，则会触发Nginx 的超时机制，超时机制会对发生超时的事件进行管理，并对这些超时事件作出处理。对于定时事件的管理包括两方面：定时事件对象的组织形式 和 定时事件对象的超时检测。

定时事件的组织

Nginx 的定时器由红黑树实现的。在保存事件的结构体ngx_event_t 中有三个关于时间管理的成员，如下所示：

```
struct ngx_event_s{
    ...
    /* 标志位，为1表示当前事件已超时 */
    unsigned         timedout:1;
    /* 标志位，为1表示当前事件存在于由红黑树维护的定时器中 */
    unsigned         timer_set:1;
    /* 由红黑树维护的定时器 */
    ngx_rbtree_node_t    timer;
    ...
};
```

Nginx 设置两个关于定时器的全局变量。在文件 src/event/ngx_event_timer.c中定义：

```
/* 所有定时器事件组成的红黑树 */
ngx_thread_volatile ngx_rbtree_t  ngx_event_timer_rbtree;
/* 红黑树的哨兵节点 */
static ngx_rbtree_node_t          ngx_event_timer_sentinel;
```

这棵红黑树的每一个节点代表一个事件 ngx_event_t 结构体中的成员timer，ngx_rbtree_node_t 节点代表事件的超时时间，以这个超时时间的大小组成的红黑树ngx_event_timer_rbtree，则该红黑树中最左边的节点代表最可能超时的事件。

定时器事件初始化实际上调用红黑树的初始化，其在文件 src/event/ngx_event_timer.c中定义：

```
/* 定时器事件初始化 */
ngx_int_t
ngx_event_timer_init(ngx_log_t *log)
{
    /* 初始化红黑树 */
    ngx_rbtree_init(&ngx_event_timer_rbtree, &ngx_event_timer_sentinel,
                    ngx_rbtree_insert_timer_value);

    /* 下面是针对多线程环境 */
#if (NGX_THREADS)

    if (ngx_event_timer_mutex) {
        ngx_event_timer_mutex->log = log;
        return NGX_OK;
    }

    ngx_event_timer_mutex = ngx_mutex_init(log, 0);
    if (ngx_event_timer_mutex == NULL) {
        return NGX_ERROR;
    }

#endif

    return NGX_OK;
}
```

定时事件的超时检测

当需要对某个事件进行超时检测时，只需要将该事件添加到定时器红黑树中即可，由函数 ngx_event_add_timer，将一个事件从定时器红黑树中删除由函数 ngx_event_del_timer 实现。以下的函数都在文件 src/event/ngx_event_timer.h中定义：

```
/* 从定时器中移除事件 */
static ngx_inline void
ngx_event_del_timer(ngx_event_t *ev)
{
    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                   "event timer del: %d: %M",
                    ngx_event_ident(ev->data), ev->timer.key);

    ngx_mutex_lock(ngx_event_timer_mutex);

    /* 从红黑树中移除指定事件的节点对象 */
    ngx_rbtree_delete(&ngx_event_timer_rbtree, &ev->timer);

    ngx_mutex_unlock(ngx_event_timer_mutex);

#if (NGX_DEBUG)
    ev->timer.left = NULL;
    ev->timer.right = NULL;
    ev->timer.parent = NULL;
```

```
#endif

    /* 设置相应的标志位 */
    ev->timer_set = 0;
}

/* 将事件添加到定时器中 */
static ngx_inline void
ngx_event_add_timer(ngx_event_t *ev, ngx_msec_t timer)
{
    ngx_msec_t      key;
    ngx_msec_int_t  diff;

    /* 设置事件对象节点的键值 */
    key = ngx_current_msec + timer;

    /* 判断事件的相应标志位 */
    if (ev->timer_set) {

        /*
         * Use a previous timer value if difference between it and a new
         * value is less than NGX_TIMER_LAZY_DELAY milliseconds: this allows
         * to minimize the rbtree operations for fast connections.
         */

        diff = (ngx_msec_int_t) (key - ev->timer.key);

        if (ngx_abs(diff) < NGX_TIMER_LAZY_DELAY) {
            ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                           "event timer: %d, old: %M, new: %M",
                           ngx_event_ident(ev->data), ev->timer.key, key);
            return;
        }

        ngx_del_timer(ev);
    }

    ev->timer.key = key;

    ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                   "event timer add: %d: %M:%M",
                   ngx_event_ident(ev->data), timer, ev->timer.key);

    ngx_mutex_lock(ngx_event_timer_mutex);

    /* 将事件对象节点插入到红黑树中 */
    ngx_rbtree_insert(&ngx_event_timer_rbtree, &ev->timer);

    ngx_mutex_unlock(ngx_event_timer_mutex);

    /* 设置标志位 */
    ev->timer_set = 1;
}
```

判断一个函数是否超时由函数 ngx_event_find_timer 实现，检查定时器所有事件由函数 ngx_event_expire_timer 实现。以下的函数都在文件src/event/ngx_event_timer.c中定义：

```
/* 找出定时器红黑树最左边的节点 */
ngx_msec_t
ngx_event_find_timer(void)
{
    ngx_msec_int_t      timer;
    ngx_rbtree_node_t  *node, *root, *sentinel;

    /* 若红黑树为空 */
    if (ngx_event_timer_rbtree.root == &ngx_event_timer_sentinel) {
        return NGX_TIMER_INFINITE;
    }

    ngx_mutex_lock(ngx_event_timer_mutex);

    root = ngx_event_timer_rbtree.root;
    sentinel = ngx_event_timer_rbtree.sentinel;

    /* 找出红黑树最小的节点，即最左边的节点 */
    node = ngx_rbtree_min(root, sentinel);

    ngx_mutex_unlock(ngx_event_timer_mutex);

    /* 计算最左节点键值与当前时间的差值timer，当timer大于0表示不超时，不大于0表示超时 */
    timer = (ngx_msec_int_t) (node->key - ngx_current_msec);

    /*
     * 若timer大于0，则事件不超时，返回该值；
     * 若timer不大于0，则事件超时，返回0，标志触发超时事件；
     */
    return (ngx_msec_t) (timer > 0 ? timer : 0);
}

/* 检查定时器中所有事件 */
void
ngx_event_expire_timers(void)
{
    ngx_event_t        *ev;
    ngx_rbtree_node_t  *node, *root, *sentinel;

    sentinel = ngx_event_timer_rbtree.sentinel;

    /* 循环检查 */
    for ( ;; ) {

        ngx_mutex_lock(ngx_event_timer_mutex);

        root = ngx_event_timer_rbtree.root;

        /* 若定时器红黑树为空，则直接返回，不做任何处理 */
        if (root == sentinel) {
```

```
            return;
        }

        /* 找出定时器红黑树最左边的节点，即最小的节点，同时也是最有可能超时的事件对象 */
        node = ngx_rbtree_min(root, sentinel);

        /* node->key <= ngx_current_time */

        /* 若检查到的当前事件已超时 */
        if ((ngx_msec_int_t) (node->key - ngx_current_msec) <= 0) {
            /* 获取超时的具体事件 */
            ev = (ngx_event_t *) ((char *) node - offsetof(ngx_event_t, timer));

            /* 下面是针对多线程 */
#if (NGX_THREADS)

            if (ngx_threaded && ngx_trylock(ev->lock) == 0) {

                /*
                 * We cannot change the timer of the event that is being
                 * handled by another thread.  And we cannot easy walk
                 * the rbtree to find next expired timer so we exit the loop.
                 * However, it should be a rare case when the event that is
                 * being handled has an expired timer.
                 */

                ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                               "event %p is busy in expire timers", ev);
                break;
            }
#endif

            ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                           "event timer del: %d: %M",
                           ngx_event_ident(ev->data), ev->timer.key);

            /* 将已超时事件对象从现有定时器红黑树中移除 */
            ngx_rbtree_delete(&ngx_event_timer_rbtree, &ev->timer);

            ngx_mutex_unlock(ngx_event_timer_mutex);

#if (NGX_DEBUG)
            ev->timer.left = NULL;
            ev->timer.right = NULL;
            ev->timer.parent = NULL;
#endif

            /* 设置事件的在定时器红黑树中的监控标志位 */
            ev->timer_set = 0;/* 0表示不受监控 */

            /* 多线程环境 */
#if (NGX_THREADS)
            if (ngx_threaded) {
                ev->posted_timedout = 1;
```

```
                ngx_post_event(ev, &ngx_posted_events);

                ngx_unlock(ev->lock);

                continue;
            }
#endif

            /* 设置事件的超时标志位 */
            ev->timedout = 1;/* 1表示已经超时 */

            /* 调用已超时事件的处理函数对该事件进行处理 */
            ev->handler(ev);

            continue;
        }

        break;
    }

    ngx_mutex_unlock(ngx_event_timer_mutex);
}
```

参考资料

《深入剖析Nginx》

《深入理解Nginx》

Nginx 事件驱动模块连接处理

概述

由于 Nginx 工作在 master-worker 多进程模式，若所有 worker 进程在同一时间监听同一个端口，当该端口有新的连接事件出现时，每个worker 进程都会调用函数ngx_event_accept 试图与新的连接建立通信，即所有worker 进程都会被唤醒，这就是所谓的“惊群”问题，这样会导致系统性能下降。幸好在Nginx 采用了ngx_accept_mutex 同步锁机制，即只有获得该锁的worker 进程才能去处理新的连接事件，也就在同一时间只能有一个worker 进程监听某个端口。虽然这样做解决了“惊群”问题，但是随之会出现另一个问题，若每次出现的新连接事件都被同一个worker 进程获得锁的权利并处理该连接事件，这样会导致进程之间不均衡的状态，即在所有worker 进程中，某些进程处理的连接事件数量很庞大，而某些进程基本上不用处理连接事件，一直处于空闲状态。因此，这样会导致worker 进程之间的负载不均衡，会影响Nginx 的整体性能。为了解决负载失衡的问题，Nginx 在已经实现同步锁的基础上定义了负载阈值ngx_accept_disabled，当某个worker 进程的负载阈值大于 0 时，表示该进程处于负载超重的状态，则Nginx 会控制该进程，使其没机会试图与新的连接事件进行通信，这样就会为其他没有负载超重的进程创造了处理新连接事件的机会，以此达到进程间的负载均衡。

连接事件处理

新连接事件由函数 ngx_event_accept 处理。

```
/* 处理新连接事件 */
void
ngx_event_accept(ngx_event_t *ev)
{
    socklen_t          socklen;
    ngx_err_t          err;
    ngx_log_t         *log;
    ngx_uint_t         level;
    ngx_socket_t       s;
    ngx_event_t       *rev, *wev;
    ngx_listening_t   *ls;
    ngx_connection_t  *c, *lc;
    ngx_event_conf_t  *ecf;
    u_char             sa[NGX_SOCKADDRLEN];
#if (NGX_HAVE_ACCEPT4)
    static ngx_uint_t  use_accept4 = 1;
#endif

    if (ev->timedout) {
        if (ngx_enable_accept_events((ngx_cycle_t *) ngx_cycle) != NGX_OK) {
            return;
        }

        ev->timedout = 0;
    }
```

```
    /* 获取ngx_event_core_module模块的配置项参数结构 */
    ecf = ngx_event_get_conf(ngx_cycle->conf_ctx, ngx_event_core_module);

    if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
        ev->available = 1;

    } else if (!(ngx_event_flags & NGX_USE_KQUEUE_EVENT)) {
        ev->available = ecf->multi_accept;
    }

    lc = ev->data;/* 获取事件所对应的连接对象 */
    ls = lc->listening;/* 获取连接对象的监听端口数组 */
    ev->ready = 0;/* 设置事件的状态为未准备就绪 */

    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
                   "accept on %V, ready: %d", &ls->addr_text, ev->available);

    do {
        socklen = NGX_SOCKADDRLEN;

        /* accept 建立一个新的连接 */
#if (NGX_HAVE_ACCEPT4)
        if (use_accept4) {
            s = accept4(lc->fd, (struct sockaddr *) sa, &socklen,
                        SOCK_NONBLOCK);
        } else {
            s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
        }
#else
        s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
#endif

        /* 连接建立错误时的相应处理 */
        if (s == (ngx_socket_t) -1) {
            err = ngx_socket_errno;

            if (err == NGX_EAGAIN) {
                ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, err,
                               "accept() not ready");
                return;
            }

            level = NGX_LOG_ALERT;

            if (err == NGX_ECONNABORTED) {
                level = NGX_LOG_ERR;

            } else if (err == NGX_EMFILE || err == NGX_ENFILE) {
                level = NGX_LOG_CRIT;
            }

#if (NGX_HAVE_ACCEPT4)
            ngx_log_error(level, ev->log, err,
```

```
                      use_accept4 ? "accept4() failed" : "accept() failed");

            if (use_accept4 && err == NGX_ENOSYS) {
                use_accept4 = 0;
                ngx_inherited_nonblocking = 0;
                continue;
            }
#else
            ngx_log_error(level, ev->log, err, "accept() failed");
#endif

            if (err == NGX_ECONNABORTED) {
                if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
                    ev->available--;
                }

                if (ev->available) {
                    continue;
                }
            }

            if (err == NGX_EMFILE || err == NGX_ENFILE) {
                if (ngx_disable_accept_events((ngx_cycle_t *) ngx_cycle)
                    != NGX_OK)
                {
                    return;
                }

                if (ngx_use_accept_mutex) {
                    if (ngx_accept_mutex_held) {
                        ngx_shmtx_unlock(&ngx_accept_mutex);
                        ngx_accept_mutex_held = 0;
                    }

                    ngx_accept_disabled = 1;

                } else {
                    ngx_add_timer(ev, ecf->accept_mutex_delay);
                }
            }

            return;
        }

#if (NGX_STAT_STUB)
        (void) ngx_atomic_fetch_add(ngx_stat_accepted, 1);
#endif

        /*
         * ngx_accept_disabled 变量是负载均衡阈值，表示进程是否超载；
         * 设置负载均衡阈值为每个进程最大连接数的八分之一减去空闲连接数；
         * 即当每个进程accept到的活动连接数超过最大连接数的7/8时，
         * ngx_accept_disabled 大于0，表示该进程处于负载过重；
         */
        ngx_accept_disabled = ngx_cycle->connection_n / 8
```

```
                    - ngx_cycle->free_connection_n;

        /* 从connections数组中获取一个connection连接来维护新的连接 */
        c = ngx_get_connection(s, ev->log);

        if (c == NULL) {
            if (ngx_close_socket(s) == -1) {
                ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_socket_errno,
                              ngx_close_socket_n " failed");
            }

            return;
        }

#if (NGX_STAT_STUB)
        (void) ngx_atomic_fetch_add(ngx_stat_active, 1);
#endif

        /* 为新的连接创建一个连接池pool，直到关闭该连接时才释放该连接池pool */
        c->pool = ngx_create_pool(ls->pool_size, ev->log);
        if (c->pool == NULL) {
            ngx_close_accepted_connection(c);
            return;
        }

        c->sockaddr = ngx_palloc(c->pool, socklen);
        if (c->sockaddr == NULL) {
            ngx_close_accepted_connection(c);
            return;
        }

        ngx_memcpy(c->sockaddr, sa, socklen);

        log = ngx_palloc(c->pool, sizeof(ngx_log_t));
        if (log == NULL) {
            ngx_close_accepted_connection(c);
            return;
        }

        /* set a blocking mode for aio and non-blocking mode for others */

        /* 设置套接字的属性 */
        if (ngx_inherited_nonblocking) {
            if (ngx_event_flags & NGX_USE_AIO_EVENT) {
                if (ngx_blocking(s) == -1) {
                    ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_socket_errno,
                                  ngx_blocking_n " failed");
                    ngx_close_accepted_connection(c);
                    return;
                }
            }

        } else {
            /* 使用epoll模型时，套接字的属性为非阻塞模式 */
```

```
            if (!(ngx_event_flags & (NGX_USE_AIO_EVENT|NGX_USE_RTSIG_EVENT))) {
                if (ngx_nonblocking(s) == -1) {
                    ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_socket_errno,
                                  ngx_nonblocking_n " failed");
                    ngx_close_accepted_connection(c);
                    return;
                }
            }
        }

        *log = ls->log;

        /* 初始化新连接 */
        c->recv = ngx_recv;
        c->send = ngx_send;
        c->recv_chain = ngx_recv_chain;
        c->send_chain = ngx_send_chain;

        c->log = log;
        c->pool->log = log;

        c->socklen = socklen;
        c->listening = ls;
        c->local_sockaddr = ls->sockaddr;
        c->local_socklen = ls->socklen;

        c->unexpected_eof = 1;

#if (NGX_HAVE_UNIX_DOMAIN)
        if (c->sockaddr->sa_family == AF_UNIX) {
            c->tcp_nopush = NGX_TCP_NOPUSH_DISABLED;
            c->tcp_nodelay = NGX_TCP_NODELAY_DISABLED;
#if (NGX_SOLARIS)
            /* Solaris's sendfilev() supports AF_NCA, AF_INET, and AF_INET6 */
            c->sendfile = 0;
#endif
        }
#endif

        /* 获取新连接的读事件、写事件 */
        rev = c->read;
        wev = c->write;

        /* 写事件准备就绪 */
        wev->ready = 1;

        if (ngx_event_flags & (NGX_USE_AIO_EVENT|NGX_USE_RTSIG_EVENT)) {
            /* rtsig, aio, iocp */
            rev->ready = 1;
        }

        if (ev->deferred_accept) {
            rev->ready = 1;
#if (NGX_HAVE_KQUEUE)
            rev->available = 1;
```

```
#endif
    }

    rev->log = log;
    wev->log = log;

    /*
     * TODO: MT: - ngx_atomic_fetch_add()
     *             or protection by critical section or light mutex
     *
     * TODO: MP: - allocated in a shared memory
     *           - ngx_atomic_fetch_add()
     *             or protection by critical section or light mutex
     */

    c->number = ngx_atomic_fetch_add(ngx_connection_counter, 1);

#if (NGX_STAT_STUB)
    (void) ngx_atomic_fetch_add(ngx_stat_handled, 1);
#endif

#if (NGX_THREADS)
    rev->lock = &c->lock;
    wev->lock = &c->lock;
    rev->own_lock = &c->lock;
    wev->own_lock = &c->lock;
#endif

    if (ls->addr_ntop) {
        c->addr_text.data = ngx_pnalloc(c->pool, ls->addr_text_max_len);
        if (c->addr_text.data == NULL) {
            ngx_close_accepted_connection(c);
            return;
        }

        c->addr_text.len = ngx_sock_ntop(c->sockaddr, c->socklen,
                                         c->addr_text.data,
                                         ls->addr_text_max_len, 0);
        if (c->addr_text.len == 0) {
            ngx_close_accepted_connection(c);
            return;
        }
    }

#if (NGX_DEBUG)
    {

    struct sockaddr_in   *sin;
    ngx_cidr_t           *cidr;
    ngx_uint_t           i;
#if (NGX_HAVE_INET6)
    struct sockaddr_in6  *sin6;
    ngx_uint_t           n;
#endif
```

```
        cidr = ecf->debug_connection.elts;
        for (i = 0; i < ecf->debug_connection.nelts; i++) {
            if (cidr[i].family != (ngx_uint_t) c->sockaddr->sa_family) {
                goto next;
            }

            switch (cidr[i].family) {

#if (NGX_HAVE_INET6)
            case AF_INET6:
                sin6 = (struct sockaddr_in6 *) c->sockaddr;
                for (n = 0; n < 16; n++) {
                    if ((sin6->sin6_addr.s6_addr[n]
                        & cidr[i].u.in6.mask.s6_addr[n])
                        != cidr[i].u.in6.addr.s6_addr[n])
                    {
                        goto next;
                    }
                }
                break;
#endif

#if (NGX_HAVE_UNIX_DOMAIN)
            case AF_UNIX:
                break;
#endif

            default: /* AF_INET */
                sin = (struct sockaddr_in *) c->sockaddr;
                if ((sin->sin_addr.s_addr & cidr[i].u.in.mask)
                    != cidr[i].u.in.addr)
                {
                    goto next;
                }
                break;
            }

            log->log_level = NGX_LOG_DEBUG_CONNECTION|NGX_LOG_DEBUG_ALL;
            break;

        next:
            continue;
        }

        }
#endif

        ngx_log_debug3(NGX_LOG_DEBUG_EVENT, log, 0,
                       "*%uA accept: %V fd:%d", c->number, &c->addr_text, s);

        /* 将新连接对应的读事件注册到事件监控机制中；
         * 注意：若是epoll事件机制，这里是不会执行，
         * 因为epoll事件机制会在调用新连接处理函数ls->handler(c)
         *（实际调用ngx_http_init_connection）时，才会把新连接对应的读事件注册到epoll事件机制中；
```

```
         */
        if (ngx_add_conn && (ngx_event_flags & NGX_USE_EPOLL_EVENT) == 0) {
            if (ngx_add_conn(c) == NGX_ERROR) {
                ngx_close_accepted_connection(c);
                return;
            }
        }

        log->data = NULL;
        log->handler = NULL;

        /*
         * 设置回调函数，完成新连接的最后初始化工作，
         * 由函数ngx_http_init_connection完成
         */
        ls->handler(c);

        /* 调整事件available标志位，该标志位为1表示Nginx一次尽可能多建立新连接 */
        if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
            ev->available--;
        }

    } while (ev->available);
}

/* 将监听socket连接的读事件加入到监听事件中 */
static ngx_int_t
ngx_enable_accept_events(ngx_cycle_t *cycle)
{
    ngx_uint_t         i;
    ngx_listening_t   *ls;
    ngx_connection_t  *c;

    /* 获取监听数组的首地址 */
    ls = cycle->listening.elts;
    /* 遍历整个监听数组 */
    for (i = 0; i < cycle->listening.nelts; i++) {

        /* 获取当前监听socket所对应的连接 */
        c = ls[i].connection;

        /* 当前连接的读事件是否处于active活跃状态 */
        if (c->read->active) {
            /* 若是处于active状态，表示该连接的读事件已经在事件监控对象中 */
            continue;
        }

        /* 若当前连接没有加入到事件监控对象中，则将该链接注册到事件监控中 */
        if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {

            if (ngx_add_conn(c) == NGX_ERROR) {
                return NGX_ERROR;
            }

        } else {
```

```
        } else {
            /* 若当前连接的读事件不在事件监控对象中，则将其加入 */
            if (ngx_add_event(c->read, NGX_READ_EVENT, 0) == NGX_ERROR) {
                return NGX_ERROR;
            }
        }
    }

    return NGX_OK;
}

/* 将监听连接的读事件从事件驱动模块中删除 */
static ngx_int_t
ngx_disable_accept_events(ngx_cycle_t *cycle)
{
    ngx_uint_t         i;
    ngx_listening_t   *ls;
    ngx_connection_t  *c;

    /* 获取监听接口 */
    ls = cycle->listening.elts;
    for (i = 0; i < cycle->listening.nelts; i++) {

        /* 获取监听接口对应的连接 */
        c = ls[i].connection;

        if (!c->read->active) {
            continue;
        }

        /* 从事件驱动模块中移除连接 */
        if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
            if (ngx_del_conn(c, NGX_DISABLE_EVENT) == NGX_ERROR) {
                return NGX_ERROR;
            }

        } else {
            /* 从事件驱动模块移除连接的读事件 */
            if (ngx_del_event(c->read, NGX_READ_EVENT, NGX_DISABLE_EVENT)
                == NGX_ERROR)
            {
                return NGX_ERROR;
            }
        }
    }

    return NGX_OK;
}
```

当出现新连接事件时，只有获得同步锁的进程才可以处理该连接事件，避免了“惊群”问题，进程试图处理新连接事件由函数 ngx_trylock_accept_mutex 实现。

```
/* 试图处理监听端口的新连接事件 */
```

```
ngx_int_t
ngx_trylock_accept_mutex(ngx_cycle_t *cycle)
{
    /* 获取ngx_accept_mutex锁，成功返回1，失败返回0 */
    if (ngx_shmtx_trylock(&ngx_accept_mutex)) {

        ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                       "accept mutex locked");

        /*
         * 标志位ngx_accept_mutex_held为1表示当前进程已经获取了ngx_accept_mutex锁；
         * 满足下面条件时，表示当前进程在之前已经获得ngx_accept_mutex锁；
         * 则直接返回；
         */
        if (ngx_accept_mutex_held
            && ngx_accept_events == 0
            && !(ngx_event_flags & NGX_USE_RTSIG_EVENT))
        {
            return NGX_OK;
        }

        /* 将所有监听连接的读事件添加到当前的epoll事件驱动模块中 */
        if (ngx_enable_accept_events(cycle) == NGX_ERROR) {
            /* 若添加失败，则释放该锁 */
            ngx_shmtx_unlock(&ngx_accept_mutex);
            return NGX_ERROR;
        }

        /* 设置当前进程获取锁的情况 */
        ngx_accept_events = 0;
        ngx_accept_mutex_held = 1;/* 表示当前进程已经得到ngx_accept_mutex锁 */

        return NGX_OK;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
                   "accept mutex lock failed: %ui", ngx_accept_mutex_held);

    /*
     * 若当前进程获取ngx_accept_mutex锁失败，并且ngx_accept_mutex_held为1，
     * 此时是错误情况
     */
    if (ngx_accept_mutex_held) {
        /* 将所有监听连接的读事件从事件驱动模块中移除 */
        if (ngx_disable_accept_events(cycle) == NGX_ERROR) {
            return NGX_ERROR;
        }

        ngx_accept_mutex_held = 0;
    }

    return NGX_OK;
}
```

Nginx 通过负载阈值 ngx_accept_disabled 控制进程是否处理新连接事件，避免进程间负载均衡问题。

```
if(ngx_accept_disabled > 0){
   ngx_accept_disabled --;
}else{
  if(ngx_trylock_accept_mutex(cycle) == NGX_ERROR){
        return;
   }
...
}
```

参考资料：

《深入理解Nginx》

《深入剖析Nginx》

《Nginx源码分析-事件循环》

《关于ngx_trylock_accept_mutex的一些解释》

Nginx 中 HTTP 模块初始化

概述

在前面的文章《 Nginx 配置解析》简单讲解了通用模块的配置项解析，并且大概讲解了HTTP 模块的配置项解析过程，本文更具体的分析HTTP 模块的初始化过程。HTTP 模块初始化过程主要有：上下文结构初始化、配置项解析、配置项合并、server 相关端口设置。

HTTP 模块接口

ngx_http_module_t 结构体

在 Nginx 中，结构体 ngx_module_t 是 Nginx 模块最基本的接口。对于每一种不同类型的模块，都有一个具体的结构体来描述这一类模块的通用接口。在Nginx 中定义了HTTP 模块的通用接口 ngx_http_module_t 结构体，该结构体定义在文件src/http/ngx_http_config.h：我们把直属于http{}、server{}、location{} 块的配置项分别称为main、srv、loc 级别配置项。

```
/* 所有HTTP模块的通用接口结构ngx_http_module_t */
typedef struct {
    /* 在解析http{}块内的配置项前回调 */
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    /* 在解析http{}块内的配置项后回调 */
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    /*
     * 创建用于存储HTTP全局配置项的结构体；
     * 该结构体中的成员将保存直属于http{}块的配置项参数；
     * 该方法在解析main配置项前调用；
     */
    void       *(*create_main_conf)(ngx_conf_t *cf);
    /* 解析完main配置项后回调 */
    char       *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    /*
     * 创建用于存储可同时出现在main、srv级别配置项的结构体；
     * 该结构体中的成员与server配置是相关联的；
     */
    void       *(*create_srv_conf)(ngx_conf_t *cf);
    /*
     * 由create_srv_conf产生的结构体所要解析的配置项，
     * 可能同时出现在main、srv级别中，
     * merge_srv_conf 方法可以将出现在main级别中的配置项值合并到srv级别的配置项中；
     */
    char       *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    /*
     * 创建用于存储同时出现在main、srv、loc级别配置项的结构体，
     * 该结构体成员与location配置相关联；
     */
    void       *(*create_loc_conf)(ngx_conf_t *cf);
    /*
     * 由create_loc_conf 产生的结构体所要解析的配置项，
     * 可能同时出现在main、srv、loc级别的配置项中，
     * merge_loc_conf 方法将出现在main、srv级别的配置项值合并到loc级别的配置项中；
     */
    char       *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

ngx_http_conf_ctx_t 结构体

在 HTTP 模块中，管理 HTTP 模块配置项的结构由 ngx_http_conf_ctx_t 实现，该结构有三个成员，分别指向三个指针数组，指针数组是由相应地 HTTP 模块create_main_conf、create_srv_conf、create_loc_conf 方法创建的结构体指针组成的数组。ngx_http_conf_ctx_t 结构体定义在文件src/http/ngx_http_config.h 中：

```
/* HTTP框架的上下文结构体ngx_http_conf_ctx_t */
typedef struct {
    /*
     * 指向一个指针数组；
     * 数组中的每个成员都是由所有HTTP模块create_main_conf方法创建的存放全局配置项的结构体，
     * 它们存放着解析直属于http{}块内main级别的配置项参数；
     */
    void        **main_conf;
    /*
     * 指向一个指针数组；
     * 数组中的每个成员都是由所有HTTP模块create_srv_conf方法创建的与server相关的配置项结构体，
     * 它们存放着main级别，或srv级别的配置项参数；
     * 这与当前的ngx_http_conf_ctx_t是在解析http{}或server{}块时创建有关；
     */
    void        **srv_conf;
    /*
     * 指向一个指针数组；
     * 数组中的每个成员都是由所有HTTP模块create_loc_conf方法创建的与location有关的配置项结构体，
     * 它们存放着main级别、srv级别、loc级别的配置项参数；
     * 这样当前ngx_http_conf_ctx_t是在解析http{}、server{}或location{}块时创建有关；
     */
    void        **loc_conf;
} ngx_http_conf_ctx_t;
```

ngx_http_module 核心模块

ngx_http_module 核心模块定义

ngx_http_module 是 HTTP 模块的核心模块，该模块的功能是：定义新的 HTTP 模块类型，并为每个HTTP 模块定义通用接口ngx_http_module_t 结构体，管理HTTP 模块生成的配置项结构体，并解析 HTTP 类配置项。该模块定义在文件src/http/ngx_http.c 中：

```
/* 定义http核心模块 */
ngx_module_t  ngx_http_module = {
    NGX_MODULE_V1,
    &ngx_http_module_ctx,                  /* module context */
    ngx_http_commands,                     /* module directives */
    NGX_CORE_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

ngx_http_module 作为核心模块，必须定义核心模块的通用接口上下文结构，其通用接口上下文结构

定义在文件src/http/ngx_http.c 中：在 ngx_http_module 核心模块中只定义了 http 模块的名称。

```
/* 定义核心模块的通用接口上下文结构 */
static ngx_core_module_t  ngx_http_module_ctx = {
    ngx_string("http"),
    NULL,
    NULL
};
```

在 ngx_http_module 模块中定义了http{} 块感兴趣的配置项数组，配置项数组定义在文件src/http/ngx_http.c 中：

```
/* 定义http模块感兴趣的配置项，即配置项指令数组 */
static ngx_command_t  ngx_http_commands[] = {

    { ngx_string("http"),
      NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
      ngx_http_block,
      0,
      0,
      NULL },

      ngx_null_command
};
```

从 ngx_http_module 模块的定义中可以知道，该模块只有一个初始化处理方法ngx_http_block，该处理方法是HTTP 模块的初始化作用。

ngx_http_module 核心模块初始化

在上面 ngx_http_module 模块的定义中已经知道，HTTP 模块的初始化过程由函数ngx_http_block 实现，首先先给出该函数的整体分析，接着对该函数进行具体的分析。该函数定义在文件src/http/ngx_http.c 中：

```
/* HTTP框架初始化 */
static char *
ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    char                        *rv;
    ngx_uint_t                  mi, m, s;
    ngx_conf_t                  pcf;
    ngx_http_module_t           *module;
    ngx_http_conf_ctx_t         *ctx;
    ngx_http_core_loc_conf_t    *clcf;
    ngx_http_core_srv_conf_t    **cscfp;
    ngx_http_core_main_conf_t   *cmcf;

    /* the main http context */
```

```
    /* 分配HTTP框架的上下文结构ngx_http_conf_ctx_t 空间 */
    ctx = ngx_pcalloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
    if (ctx == NULL) {
        return NGX_CONF_ERROR;
    }

    /*
     * conf 是结构体ngx_cycle_t 成员conf_ctx数组中的元素，
     * 该元素conf指向ngx_http_module模块所对应的配置项结构信息；
     */
    *(ngx_http_conf_ctx_t **) conf = ctx;

    /* count the number of the http modules and set up their indices */

    /* 初始化所有HTTP模块的ctx_index序号 */
    ngx_http_max_module = 0;
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        ngx_modules[m]->ctx_index = ngx_http_max_module++;
    }

    /*
     * 分配存储HTTP模块main级别下的main_conf配置项的空间；
     */
    /* the http main_conf context, it is the same in the all http contexts */

    ctx->main_conf = ngx_pcalloc(cf->pool,
                                 sizeof(void *) * ngx_http_max_module);
    if (ctx->main_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /*
     * 分配存储HTTP模块main级别下的srv_conf配置项的空间；
     */
    /*
     * the http null srv_conf context, it is used to merge
     * the server{}s' srv_conf's
     */

    ctx->srv_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);
    if (ctx->srv_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /*
     * 分配存储HTTP模块main级别下的loc_conf配置项的空间；
     */
    /*
     * the http null loc_conf context, it is used to merge
     * the server{}s' loc_conf's
     */
```

```
     */

    ctx->loc_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);
    if (ctx->loc_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /*
     * create the main_conf's, the null srv_conf's, and the null loc_conf's
     * of the all http modules
     */

    /*
     * 遍历所有HTTP模块，为每个HTTP模块创建main级别的配置项结构main_conf、srv_conf、loc_conf；
     */
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;
        mi = ngx_modules[m]->ctx_index;

        /*
         * 调用create_main_conf创建main级别的配置项结构main_conf；
         */
        if (module->create_main_conf) {
            ctx->main_conf[mi] = module->create_main_conf(cf);
            if (ctx->main_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }

        /*
         * 调用create_srv_conf创建main级别的配置项结构srv_conf；
         */
        if (module->create_srv_conf) {
            ctx->srv_conf[mi] = module->create_srv_conf(cf);
            if (ctx->srv_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }

        /*
         * 调用create_loc_conf创建main级别的配置项结构loc_conf；
         */
        if (module->create_loc_conf) {
            ctx->loc_conf[mi] = module->create_loc_conf(cf);
            if (ctx->loc_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }
    }

    /*
```

```
     * 保存待解析配置项结构cf的副本为pcf，待解析完毕后恢复cf；
     * 这里备份是由于配置指令解析函数ngx_conf_parse递归调用，因此为了不影响外层的调用环境；
     */
    pcf = *cf;
    /*
     * 把HTTP模块解析指令的上下文参数保存到配置项结构ngx_http_conf_ctx_t ctx中；
     */
    cf->ctx = ctx;/* 值-结果 模式 */

    /* 遍历所有HTTP模块，并调用每个模块的preconfiguration回调函数 */
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->preconfiguration) {
            if (module->preconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }

    /*
     * 调用模块通用配置项解析函数ngx_conf_parse解析http{}块内的指令；
     */
    /* parse inside the http{} block */

    cf->module_type = NGX_HTTP_MODULE;/* 模块类型为HTTP模块 */
    cf->cmd_type = NGX_HTTP_MAIN_CONF;/* 指令类型为HTTP模块的main级别指令 */
    /*
     * 开始解析http{}块内的指令；
     *     这里必须注意的是：http{}块内可能会包含server{}块，
     * 而server{}可能会包含location{}块，location{}块会嵌套location{}块；
     *     还需注意的是http{}块内可能有多个server{}块，
     * location{}块也可能有多个location{}块；
     * 因此，配置项解析函数ngx_conf_parse是被递归调用的；*/
    rv = ngx_conf_parse(cf, NULL);

    if (rv != NGX_CONF_OK) {
        goto failed;
    }

    /*
     * 解析完成http{}块内的所有指令后（包括server{}、location{}块的解析），
     * 进行下面的程序
     */
    /*
     * init http{} main_conf's, merge the server{}s' srv_conf's
     * and its location{}s' loc_conf's
     */

    /* 获取ngx_http_core_module模块的main_conf配置项结构 */
    cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];
```

```
    cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];
    /* 获取所有srv_conf配置项结构 */
    cscfp = cmcf->servers.elts;

    /*
     * 遍历所有HTTP模块，并初始化每个HTTP模块的main_conf结构，
     * 同时合并srv_conf 结构（当然srv_conf结构里面包含loc_conf结构，所有也合并loc_conf结构）；
     */
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        /*
        * ngx_modules[m]是一个ngx_module_t模块结构体，
        * 它的ctx成员相对于HTTP模块来说是ngx_http_module_t接口；
        */
       module = ngx_modules[m]->ctx;
       /* 获取当前HTTP模块在HTTP模块类的序号 */
       mi = ngx_modules[m]->ctx_index;

        /* 初始化HTTP模块的main_conf结构 */
        /* init http{} main_conf's */

        if (module->init_main_conf) {
            rv = module->init_main_conf(cf, ctx->main_conf[mi]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }

        /* 合并当前HTTP模块不同级别的配置项结构 */
        rv = ngx_http_merge_servers(cf, cmcf, module, mi);
        if (rv != NGX_CONF_OK) {
            goto failed;
        }
    }

    /* 以下是监听端口管理的内容 */

    /* 静态二叉查找树来保存location配置 */
    /* create location trees */

    /* 遍历http{}块下的所有server{}块 */
    for (s = 0; s < cmcf->servers.nelts; s++) {

        /* 获取server{}块下location{}块所对应的ngx_http_core_loc_conf_t loc_conf结构体 */
        clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];

        /*
         * 将ngx_http_core_loc_conf_t 组成的双向链表按照location匹配字符串进行排序；
         * 注意：location{}块可能嵌套location{}块，所以该函数是递归调用；
         */
        if (ngx_http_init_locations(cf, cscfp[s], clcf) != NGX_OK) {
            return NGX_CONF_ERROR;
```

```
        }

        /*
         * 按照已排序的location{}的双向链表构建静态的二叉查找树，
         * 该方法也是递归调用；
         */
        if (ngx_http_init_static_location_trees(cf, clcf) != NGX_OK) {
            return NGX_CONF_ERROR;
        }
    }

    /* 初始化可添加自定义处理方法的7个HTTP阶段的动态数组 */
    if (ngx_http_init_phases(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /* 将HTTP请求的头部header初始化成hash结构 */
    if (ngx_http_init_headers_in_hash(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /* 调用所有HTTP模块的postconfiguration方法 */
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->postconfiguration) {
            if (module->postconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }

    if (ngx_http_variables_init_vars(cf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /*
     * http{}'s cf->ctx was needed while the configuration merging
     * and in postconfiguration process
     */

    *cf = pcf;

    /* 初始化phase_engine_handlers数组 */
    if (ngx_http_init_phase_handlers(cf, cmcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    /* optimize the lists of ports, addresses and server names */

    /* 设置server与监听端口的关系，并设置新连接事件的处理方法 */
```

```
    /* 设置server监听端口的大小，并设置新连接事件的处理方法 */
    if (ngx_http_optimize_servers(cf, cmcf, cmcf->ports) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;

failed:

    *cf = pcf;

    return rv;
}
```

从上面的分析中可以总结出 HTTP 模块初始化的流程如下：

- Nginx 进程进入主循环，在主循环中调用配置解析器解析配置文件*nginx.conf*;
- 在配置文件中遇到 http{} 块配置，则 HTTP 框架开始初始化并启动，其由函数 ngx_http_block() 实现；
- HTTP 框架初始化所有 HTTP 模块的序列号，并创建 3 个类型为 *ngx_http_conf_ctx_t *结构的数组用于存储所有HTTP 模块的*create_main_conf、create_srv_conf、create_loc_conf*方法返回的指针地址；
- 调用每个 HTTP 模块的 preconfiguration 方法；
- HTTP 框架调用函数 ngx_conf_parse() 开始循环解析配置文件 *nginx.conf *中的http{}块里面的所有配置项，http{} 块内可嵌套多个server{} 块，而 server{} 块可嵌套多个 location{}，location{} 依旧可以嵌套location{}，因此配置项解析函数是递归调用；
- HTTP 框架处理完毕 http{} 配置项，根据解析配置项的结果，必要时调用ngx_http_merge_servers 方法进行配置项合并处理，即合并main、srv、loc 级别下server、location 相关的配置项；
- 初始化可添加处理方法的 HTTP 阶段的动态数组；
- 调用所有 HTTP 模块的 postconfiguration 方法使 HTTP 模块可以处理HTTP 阶段，即将HTTP 模块的ngx_http_handler_pt 处理方法添加到HTTP 阶段中；
- 根据 HTTP 模块处理 HTTP 阶段的方法构造 phase_engine_handlers 数组；
- 构造 server 相关的监听端口，并设置新连接事件的回调方法为ngx_http_init_connection ；
- 继续处理其他 http{} 块之外的配置项，直到配置文件解析器处理完所有配置项后通知Nginx 主循环配置项解析完毕。此时，Nginx 才会启动Web 服务器；

ngx_http_core_module 模块

ngx_http_core_main_conf_t 结构体

在初始化 HTTP 模块的过程中，会调用配置项解析函数ngx_conf_parse 解析http{} 块内的配置项，当遇到server{} 块、location{} 块配置项时，此时会调用配置项解析函数解析server{} 和location{} 块，在解析这两个配置块的过程中依旧会创建属于该块的配置项结构srv_conf、loc_conf，因此就会导致与http{} 块所创建的配置项结构体重复，这时候就需要对这些配置项进行管理与合并。首先先看下结构体 ngx_http_core_main_conf_t，ngx_http_core_main_conf_t 是ngx_http_core_module 的 main_conf，存储了 http{} 层的配置参数。该结构体定义在文件src/http/ngx_http_core_module.h中：

```
/* ngx_http_core_main_conf_t 结构体 */
typedef struct {
    /* 指针数组，每个指针指向表示server{}块的ngx_http_core_srv_conf_t结构体地址 */
    ngx_array_t                servers;         /* ngx_http_core_srv_conf_t */

    /* 各HTTP阶段处理方法构成的phases数组构建的阶段索引 */
    ngx_http_phase_engine_t    phase_engine;

    ngx_hash_t                 headers_in_hash;

    ngx_hash_t                 variables_hash;

    ngx_array_t                variables;       /* ngx_http_variable_t */
    ngx_uint_t                 ncaptures;

    /* 配置项中散列桶bucket最大数量 */
    ngx_uint_t                 server_names_hash_max_size;
    /* 配置项中每个散列桶bucket占用内存的最大值 */
    ngx_uint_t                 server_names_hash_bucket_size;

    ngx_uint_t                 variables_hash_max_size;
    ngx_uint_t                 variables_hash_bucket_size;

    ngx_hash_keys_arrays_t    *variables_keys;

    /* 存放http{}配置块下监听的所有ngx_http_conf_port_t 端口*/
    ngx_array_t                *ports;

    ngx_uint_t                 try_files;       /* unsigned  try_files:1 */

    /*
     * 在HTTP模块初始化时，使各HTTP模块在HTTP阶段添加处理方法，
     * 数组中每一个成员ngx_http_phase_t结构对应一个HTTP阶段；
     */
    ngx_http_phase_t           phases[NGX_HTTP_LOG_PHASE + 1];
} ngx_http_core_main_conf_t;
```

ngx_http_core_srv_conf_t 结构体

结构体 ngx_http_core_srv_conf_t，ngx_http_core_srv_conf_t 是ngx_http_core_module 的 srv_conf，存储了server{} 层的配置参数。该结构体定义在文件src/http/ngx_http_core_module.h中：

```
/* ngx_http_core_srv_conf_t结构体 */
typedef struct {
    /* array of the ngx_http_server_name_t, "server_name" directive */
    ngx_array_t                server_names;

    /* server ctx */
    /* 指向当前server{}块所属的ngx_http_conf_ctx_t结构体 */
    ngx_http_conf_ctx_t        *ctx;

    /* 当前server{}块的虚拟主机名 */
    ngx_str_t                  server_name;

    size_t                     connection_pool_size;
    size_t                     request_pool_size;
    size_t                     client_header_buffer_size;

    ngx_bufs_t                 large_client_header_buffers;

    ngx_msec_t                 client_header_timeout;

    ngx_flag_t                 ignore_invalid_headers;
    ngx_flag_t                 merge_slashes;
    ngx_flag_t                 underscores_in_headers;

    unsigned                   listen:1;
#if (NGX_PCRE)
    unsigned                   captures:1;
#endif

    ngx_http_core_loc_conf_t  **named_locations;
} ngx_http_core_srv_conf_t;
```

ngx_http_core_loc_conf_t 结构体

结构体 ngx_http_core_loc_conf_t，ngx_http_core_loc_conf_t 是ngx_http_core_module 的loc_conf，存储了location{} 层的配置参数。该结构体定义在文件src/http/ngx_http_core_module.h中：

```
struct ngx_http_core_loc_conf_s {
    /* location名称，即nginx.conf配置文件中location后面的表达式 */
    ngx_str_t     name;          /* location name */

#if (NGX_PCRE)
    ngx_http_regex_t  *regex;
#endif

    unsigned      noname:1;   /* "if () {}" block or limit_except */
    unsigned      lmt_excpt:1;
    unsigned      named:1;

    unsigned      exact_match:1;
    unsigned      noregex:1;
```

```
    unsigned      auto_redirect:1;
#if (NGX_HTTP_GZIP)
    unsigned      gzip_disable_msie6:2;
#if (NGX_HTTP_DEGRADATION)
    unsigned      gzip_disable_degradation:2;
#endif
#endif

    ngx_http_location_tree_node_t   *static_locations;
#if (NGX_PCRE)
    ngx_http_core_loc_conf_t       **regex_locations;
#endif

    /* 指向所属location{}块内ngx_http_conf_ctx_t 结构体中的loc_conf指针数组 */
    /* pointer to the modules' loc_conf */
    void        **loc_conf;

    uint32_t      limit_except;
    void        **limit_except_loc_conf;

    ngx_http_handler_pt  handler;

    /* location name length for inclusive location with inherited alias */
    size_t        alias;
    ngx_str_t     root;                   /* root, alias */
    ngx_str_t     post_action;

    ngx_array_t  *root_lengths;
    ngx_array_t  *root_values;

    ngx_array_t  *types;
    ngx_hash_t    types_hash;
    ngx_str_t     default_type;

    off_t         client_max_body_size;    /* client_max_body_size */
    off_t         directio;                /* directio */
    off_t         directio_alignment;      /* directio_alignment */

    size_t        client_body_buffer_size; /* client_body_buffer_size */
    size_t        send_lowat;              /* send_lowat */
    size_t        postpone_output;         /* postpone_output */
    size_t        limit_rate;              /* limit_rate */
    size_t        limit_rate_after;        /* limit_rate_after */
    size_t        sendfile_max_chunk;      /* sendfile_max_chunk */
    size_t        read_ahead;              /* read_ahead */

    ngx_msec_t    client_body_timeout;     /* client_body_timeout */
    ngx_msec_t    send_timeout;            /* send_timeout */
    ngx_msec_t    keepalive_timeout;       /* keepalive_timeout */
    ngx_msec_t    lingering_time;          /* lingering_time */
    ngx_msec_t    lingering_timeout;       /* lingering_timeout */
    ngx_msec_t    resolver_timeout;        /* resolver_timeout */

    ngx_resolver_t  *resolver;             /* resolver */
```

```
    time_t       keepalive_header;   /* keepalive_timeout */

    ngx_uint_t   keepalive_requests; /* keepalive_requests */
    ngx_uint_t   keepalive_disable;  /* keepalive_disable */
    ngx_uint_t   satisfy;            /* satisfy */
    ngx_uint_t   lingering_close;    /* lingering_close */
    ngx_uint_t   if_modified_since;  /* if_modified_since */
    ngx_uint_t   max_ranges;         /* max_ranges */
    ngx_uint_t   client_body_in_file_only; /* client_body_in_file_only */

    ngx_flag_t   client_body_in_single_buffer;
                                     /* client_body_in_singe_buffer */
    ngx_flag_t   internal;           /* internal */
    ngx_flag_t   sendfile;           /* sendfile */
#if (NGX_HAVE_FILE_AIO)
    ngx_flag_t   aio;                /* aio */
#endif
    ngx_flag_t   tcp_nopush;         /* tcp_nopush */
    ngx_flag_t   tcp_nodelay;        /* tcp_nodelay */
    ngx_flag_t   reset_timedout_connection; /* reset_timedout_connection */
    ngx_flag_t   server_name_in_redirect; /* server_name_in_redirect */
    ngx_flag_t   port_in_redirect;   /* port_in_redirect */
    ngx_flag_t   msie_padding;       /* msie_padding */
    ngx_flag_t   msie_refresh;       /* msie_refresh */
    ngx_flag_t   log_not_found;      /* log_not_found */
    ngx_flag_t   log_subrequest;     /* log_subrequest */
    ngx_flag_t   recursive_error_pages;   /* recursive_error_pages */
    ngx_flag_t   server_tokens;      /* server_tokens */
    ngx_flag_t   chunked_transfer_encoding; /* chunked_transfer_encoding */
    ngx_flag_t   etag;               /* etag */

#if (NGX_HTTP_GZIP)
    ngx_flag_t   gzip_vary;          /* gzip_vary */

    ngx_uint_t   gzip_http_version;  /* gzip_http_version */
    ngx_uint_t   gzip_proxied;       /* gzip_proxied */

#if (NGX_PCRE)
    ngx_array_t  *gzip_disable;      /* gzip_disable */
#endif
#endif

#if (NGX_HAVE_OPENAT)
    ngx_uint_t   disable_symlinks;   /* disable_symlinks */
    ngx_http_complex_value_t  *disable_symlinks_from;
#endif

    ngx_array_t  *error_pages;       /* error_page */
    ngx_http_try_file_t    *try_files;     /* try_files */

    ngx_path_t   *client_body_temp_path;   /* client_body_temp_path */

    ngx_open_file_cache_t  *open_file_cache;
    time_t       open_file_cache_valid;
```

```
    time_t       open_file_cache_valid;
    ngx_uint_t    open_file_cache_min_uses;
    ngx_flag_t    open_file_cache_errors;
    ngx_flag_t    open_file_cache_events;

    ngx_log_t    *error_log;

    ngx_uint_t    types_hash_max_size;
    ngx_uint_t    types_hash_bucket_size;

    /* 将同一个server{}块内多个表达location{}块的ngx_http_core_loc_conf_t 结构体以双向链表方式组织，
     * 该指针指向ngx_http_location_queue_t结构体
     */
    ngx_queue_t  *locations;

#if 0
    ngx_http_core_loc_conf_t  *prev_location;
#endif
};

typedef struct{

    /* 作为ngx_queue_t 双向链表容器，将ngx_http_location_queue_t结构体连接起来 */
    ngx_queue_t queue;

    /* 若location中字符串可以精确匹配（包括正则表达式），
     * exact将指向对应的ngx_http_core_loc_conf_t结构体，否则为NULL
     */
     ngx_http_core_loc_conf_t *exact;

     /* 若location中字符串无精确匹配（包括自定义通配符），
     * inclusive将指向对应的ngx_http_core_loc_conf_t结构体，否则为NULL
     */
     ngx_http_core_loc_conf_t *inclusive;

     /* 指向location的名称 */
     ngx_str_t *name;
     ...

}ngx_http_location_queue_t;
```

ngx_http_core_module 模块定义

ngx_http_core_module 模块是HTTP 模块中的第一个模块，该模块管理srv、loc 级别的配置项结构。该模块在文件src/http/ngx_http_core_module.c中定义：

```
ngx_module_t  ngx_http_core_module = {
    NGX_MODULE_V1,
    &ngx_http_core_module_ctx,             /* module context */
    ngx_http_core_commands,                /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

在模块的定义中，其中定义了 HTTP 模块的上下文结构ngx_http_core_module_ctx，该上下文结构体定义如下：

```
static ngx_http_module_t  ngx_http_core_module_ctx = {
    ngx_http_core_preconfiguration,        /* preconfiguration */
    NULL,                                  /* postconfiguration */

    ngx_http_core_create_main_conf,        /* create main configuration */
    ngx_http_core_init_main_conf,          /* init main configuration */

    ngx_http_core_create_srv_conf,         /* create server configuration */
    ngx_http_core_merge_srv_conf,          /* merge server configuration */

    ngx_http_core_create_loc_conf,         /* create location configuration */
    ngx_http_core_merge_loc_conf           /* merge location configuration */
};
```

由于该模块中感兴趣的配置项太多，这里只列出 server、location 配置项。定义如下：

```
    ...

    { ngx_string("server"),
      NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
      ngx_http_core_server,
      0,
      0,
      NULL },

    ...

    { ngx_string("location"),
      NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE12,
      ngx_http_core_location,
      NGX_HTTP_SRV_CONF_OFFSET,
      0,
      NULL },

      ...
      ngx_null_command
};
```

管理 HTTP 模块不同级别的配置项结构体

在 HTTP 模块的 http{} 配置项解析过程中，可能遇到多个嵌套 server{} 块以及location{}，不同块之间的解析都会创建相应的结构体保存配置项参数，但是由于属于嵌套关系，所有必须管理好不同块之间的配置项结构体，方便解析完毕后合并相应的配置项，以下针对不同级别的配置项结构体进行分析。

获取不同级别配置项结构

根据不同结构体变量的参数获取不同级别的配置项结构体由宏定义实现，在文件src/http/ngx_http_config.h中定义如下：

```
/* 利用结构体变量ngx_http_request_t r获取HTTP模块main、srv、loc级别的配置项结构体 */
#define ngx_http_get_module_main_conf(r, module)                             \
    (r)->main_conf[module.ctx_index]
#define ngx_http_get_module_srv_conf(r, module)  (r)->srv_conf[module.ctx_index]
#define ngx_http_get_module_loc_conf(r, module)  (r)->loc_conf[module.ctx_index]

/* 利用结构体变量ngx_conf_t cf获取HTTP模块的main、srv、loc级别的配置项结构体 */
#define ngx_http_conf_get_module_main_conf(cf, module)                       \
    ((ngx_http_conf_ctx_t *) cf->ctx)->main_conf[module.ctx_index]
#define ngx_http_conf_get_module_srv_conf(cf, module)                        \
    ((ngx_http_conf_ctx_t *) cf->ctx)->srv_conf[module.ctx_index]
#define ngx_http_conf_get_module_loc_conf(cf, module)                        \
    ((ngx_http_conf_ctx_t *) cf->ctx)->loc_conf[module.ctx_index]

/* 利用全局变量ngx_cycle_t cycle获取HTTP模块的main级别配置项结构体 */
#define ngx_http_cycle_get_module_main_conf(cycle, module)                   \
    (cycle->conf_ctx[ngx_http_module.index] ?                                \
        ((ngx_http_conf_ctx_t *) cycle->conf_ctx[ngx_http_module.index])     \
            ->main_conf[module.ctx_index]:                                   \
        NULL)
```

main 级别的配置项结构体

在 ngx_http_module 模块 http{} 块配置项解析的初始化过程中由函数 ngx_http_block 实现，在实现过程中创建并初始化了HTTP 模块main 级别的配置项main_conf、srv_conf、loc_conf 结构体。main 级别的配置项结构体之间的关系如下图所示：

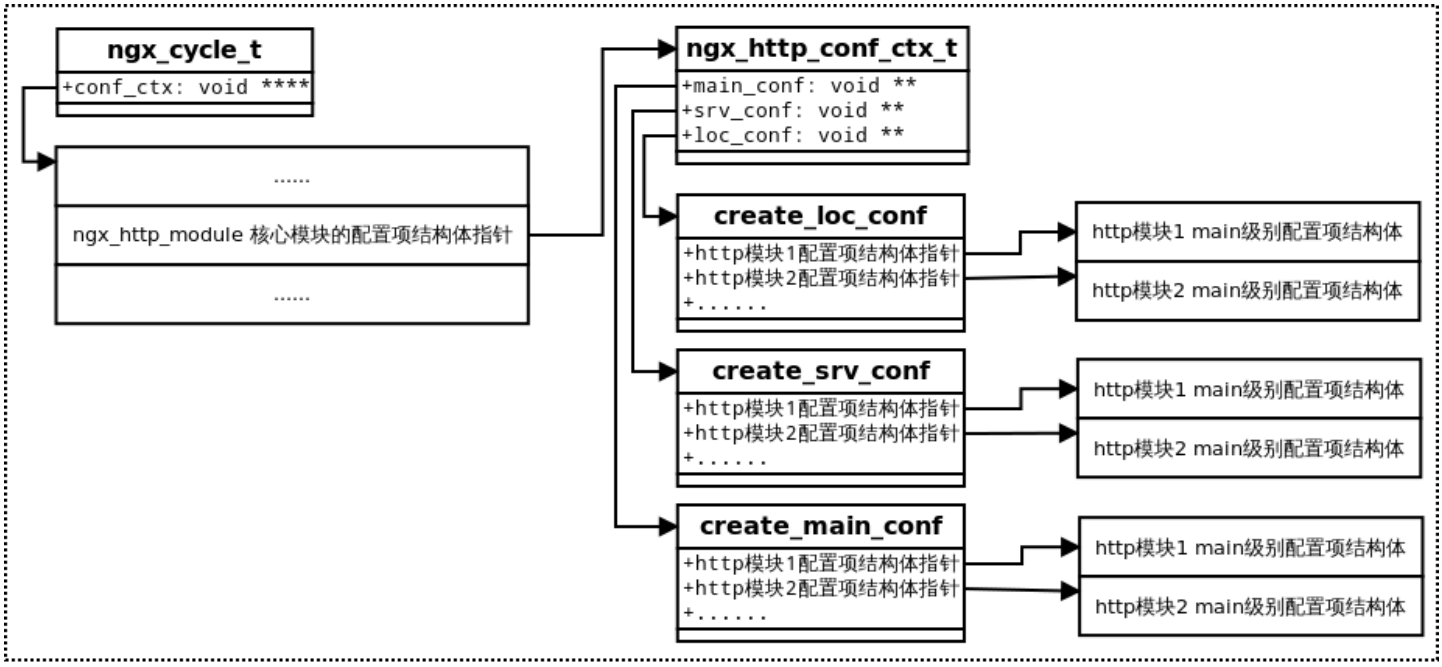


图 1 HTTP模块main级别配置项结构体

server 级别的配置项结构体

在 ngx_http_module 模块在调用函数ngx_conf_parse 解析 http{} 块main 级别配置项时，若遇到 server{} 块配置项，则会递归调用函数ngx_conf_parse 解析ngx_http_core_module 模块中 server{} 块

配置项，并调用方法ngx_http_core_server 初始化server{} 块 ，该方法创建并初始化了HTTP 模块srv 级别的配置项srv_conf、loc_conf 结构体。server{} 块配置项的初始化函数创建配置项结构体的源码如下所示：

```
static char *
ngx_http_core_server(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
{
    char                        *rv;
    void                        *mconf;
    ngx_uint_t                   i;
    ngx_conf_t                   pcf;
    ngx_http_module_t           *module;
    struct sockaddr_in          *sin;
    ngx_http_conf_ctx_t         *ctx, *http_ctx;
    ngx_http_listen_opt_t        lsopt;
    ngx_http_core_srv_conf_t    *cscf, **cscfp;
    ngx_http_core_main_conf_t   *cmcf;

    /* 分配HTTP框架的上下文结构ngx_http_conf_ctx_t */
    ctx = ngx_pcalloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
    if (ctx == NULL) {
        return NGX_CONF_ERROR;
    }

    /* 其中main_conf将指向所属于http{}块下ngx_http_conf_ctx_t 结构体的main_conf指针数组 */
    http_ctx = cf->ctx;
    ctx->main_conf = http_ctx->main_conf;

    /* the server{}'s srv_conf */

    /* 分配存储HTTP模块srv级别下的srv_conf配置项空间 */
    ctx->srv_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);
    if (ctx->srv_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /* the server{}'s loc_conf */

    /* 分配存储HTTP模块srv级别下的loc_conf配置项空间 */
    ctx->loc_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);
    if (ctx->loc_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /* 遍历所有HTTP模块，为每个模块创建srv级别的配置项结构srv_conf、loc_conf */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[i]->ctx;

        /* 调用create_srv_conf创建srv级别的配置项结构srv_conf */
```

```
        if (module->create_srv_conf) {
            mconf = module->create_srv_conf(cf);
            if (mconf == NULL) {
                return NGX_CONF_ERROR;
            }

            ctx->srv_conf[ngx_modules[i]->ctx_index] = mconf;
        }

        /* 调用create_loc_conf创建srv级别的配置项结构loc_conf */
        if (module->create_loc_conf) {
            mconf = module->create_loc_conf(cf);
            if (mconf == NULL) {
                return NGX_CONF_ERROR;
            }

            ctx->loc_conf[ngx_modules[i]->ctx_index] = mconf;
        }
    }

    /*
     * 将属于当前server{}块的ngx_http_core_srv_conf_t 添加到
     * 结构体ngx_http_core_main_conf_t成员servers的动态数组中；
     */
    /* the server configuration context */

    cscf = ctx->srv_conf[ngx_http_core_module.ctx_index];
    cscf->ctx = ctx;

    cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];

    cscfp = ngx_array_push(&cmcf->servers);
    if (cscfp == NULL) {
        return NGX_CONF_ERROR;
    }

    *cscfp = cscf;

    /* 解析当前server{}块下的全部srv级别的配置项 */
    /* parse inside server{} */

    pcf = *cf;
    cf->ctx = ctx;
    cf->cmd_type = NGX_HTTP_SRV_CONF;

    rv = ngx_conf_parse(cf, NULL);

    /* 设置listen监听端口 */
    *cf = pcf;

    if (rv == NGX_CONF_OK && !cscf->listen) {
        ngx_memzero(&lsopt, sizeof(ngx_http_listen_opt_t));

        sin = &lsopt.u.sockaddr_in;
```

```
        sin->sin_family = AF_INET;
#if (NGX_WIN32)
        sin->sin_port = htons(80);
#else
        sin->sin_port = htons((getuid() == 0) ? 80 : 8000);
#endif
        sin->sin_addr.s_addr = INADDR_ANY;

        lsopt.socklen = sizeof(struct sockaddr_in);

        lsopt.backlog = NGX_LISTEN_BACKLOG;
        lsopt.rcvbuf = -1;
        lsopt.sndbuf = -1;
#if (NGX_HAVE_SETFIB)
        lsopt.setfib = -1;
#endif
#if (NGX_HAVE_TCP_FASTOPEN)
        lsopt.fastopen = -1;
#endif
        lsopt.wildcard = 1;

        (void) ngx_sock_ntop(&lsopt.u.sockaddr, lsopt.socklen, lsopt.addr,
                             NGX_SOCKADDR_STRLEN, 1);

        if (ngx_http_add_listen(cf, cscf, &lsopt) != NGX_OK) {
            return NGX_CONF_ERROR;
        }
    }

    return rv;
}
```

srv 级别的配置项结构体之间的关系如下图所示：

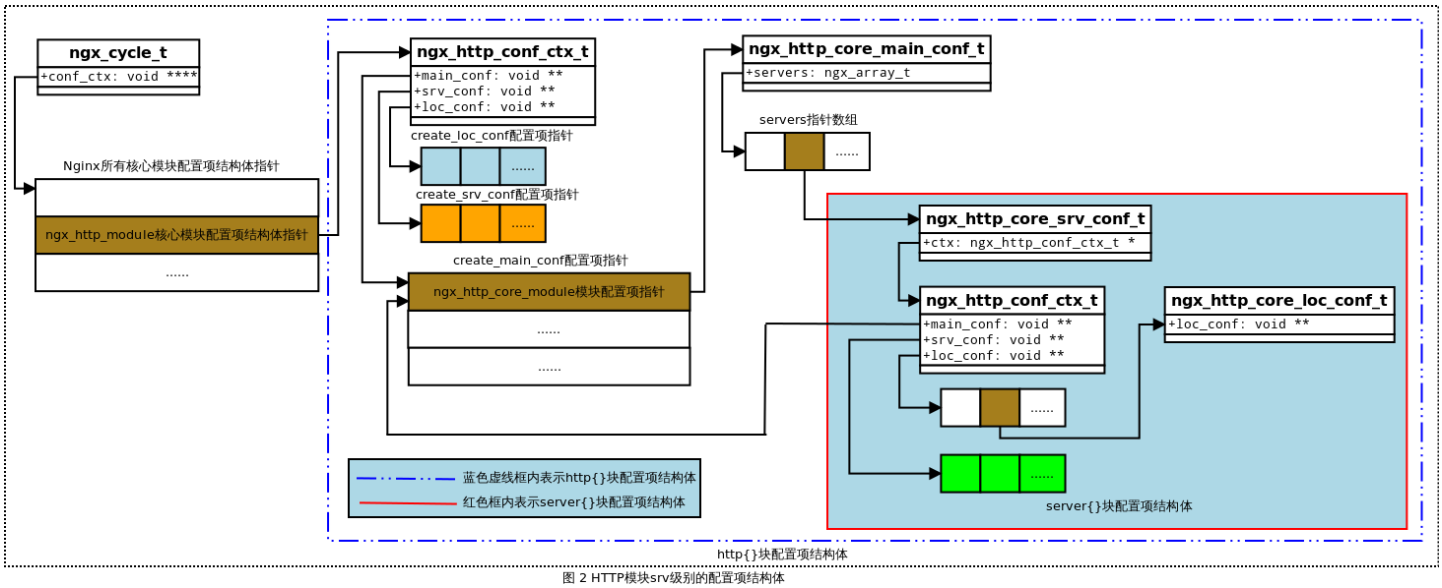


图 2 HTTP模块srv级别的配置项结构体

location 级别的配置项结构体

在 ngx_http_core_module 模块在调用函数ngx_conf_parse 解析 server{} 块srv 级别配置项时，若

遇到 location{} 块配置项，则会递归调用函数ngx_conf_parse 解析ngx_http_core_module 模块中 location{} 块配置项，并调用方法ngx_http_core_location 初始化location{} 块 ，该方法创建并初始化了HTTP 模块loc 级别的配置项loc_conf 结构体。location{} 块配置项的初始化函数创建配置项结构体的源码如下所示：

```
static char *
ngx_http_core_location(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
{
    char                       *rv;
    u_char                     *mod;
    size_t                     len;
    ngx_str_t                  *value, *name;
    ngx_uint_t                 i;
    ngx_conf_t                 save;
    ngx_http_module_t          *module;
    ngx_http_conf_ctx_t        *ctx, *pctx;
    ngx_http_core_loc_conf_t  *clcf, *pclcf;

    /* 分配HTTP框架的上下文结构ngx_http_conf_ctx_t */
    ctx = ngx_pcalloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
    if (ctx == NULL) {
        return NGX_CONF_ERROR;
    }

    /*
     * 其中main_conf、srv_conf将指向所属于server{}块下ngx_http_conf_ctx_t 结构体
     * 的main_conf、srv_conf指针数组；
     */
    pctx = cf->ctx;
    ctx->main_conf = pctx->main_conf;
    ctx->srv_conf = pctx->srv_conf;

    /* 分配存储HTTP模块loc级别下的loc_conf配置项空间 */
    ctx->loc_conf = ngx_pcalloc(cf->pool, sizeof(void *) * ngx_http_max_module);
    if (ctx->loc_conf == NULL) {
        return NGX_CONF_ERROR;
    }

    /* 遍历所有HTTP模块，为每个模块创建loc级别的配置项结构体loc_conf */
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[i]->ctx;

        /* 调用模块的create_loc_conf创建loc级别的配置项结构体loc_conf */
        if (module->create_loc_conf) {
            ctx->loc_conf[ngx_modules[i]->ctx_index] =
                                       module->create_loc_conf(cf);
            /* 将loc_conf配置项结构体按照ctx_index顺序保存到loc_conf指针数组中 */
            if (ctx->loc_conf[ngx_modules[i]->ctx_index] == NULL) {
                return NGX_CONF_ERROR
```

```
                return NGX_CONF_ERROR;
            }
        }
    }

    clcf = ctx->loc_conf[ngx_http_core_module.ctx_index];
    clcf->loc_conf = ctx->loc_conf;

    value = cf->args->elts;

    /* 以下是对正则表达式的处理 */
    if (cf->args->nelts == 3) {

        len = value[1].len;
        mod = value[1].data;
        name = &value[2];

        if (len == 1 && mod[0] == '=') {

            clcf->name = *name;
            clcf->exact_match = 1;

        } else if (len == 2 && mod[0] == '^' && mod[1] == '~') {

            clcf->name = *name;
            clcf->noregex = 1;

        } else if (len == 1 && mod[0] == '~') {

            if (ngx_http_core_regex_location(cf, clcf, name, 0) != NGX_OK) {
                return NGX_CONF_ERROR;
            }

        } else if (len == 2 && mod[0] == '~' && mod[1] == '*') {

            if (ngx_http_core_regex_location(cf, clcf, name, 1) != NGX_OK) {
                return NGX_CONF_ERROR;
            }

        } else {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "invalid location modifier \"%V\"", &value[1]);
            return NGX_CONF_ERROR;
        }

    } else {

        name = &value[1];

        if (name->data[0] == '=') {

            clcf->name.len = name->len - 1;
            clcf->name.data = name->data + 1;
            clcf->exact_match = 1;
```

```
        } else if (name->data[0] == '^' && name->data[1] == '~') {

            clcf->name.len = name->len - 2;
            clcf->name.data = name->data + 2;
            clcf->noregex = 1;

        } else if (name->data[0] == '~') {

            name->len--;
            name->data++;

            if (name->data[0] == '*') {

                name->len--;
                name->data++;

                if (ngx_http_core_regex_location(cf, clcf, name, 1) != NGX_OK) {
                    return NGX_CONF_ERROR;
                }

            } else {
                if (ngx_http_core_regex_location(cf, clcf, name, 0) != NGX_OK) {
                    return NGX_CONF_ERROR;
                }
            }

        } else {

            clcf->name = *name;

            if (name->data[0] == '@') {
                clcf->named = 1;
            }
        }
    }

    pclcf = pctx->loc_conf[ngx_http_core_module.ctx_index];

    if (pclcf->name.len) {

        /* nested location */

#if 0
        clcf->prev_location = pclcf;
#endif

        if (pclcf->exact_match) {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "location \"%V\" cannot be inside "
                               "the exact location \"%V\"",
                               &clcf->name, &pclcf->name);
            return NGX_CONF_ERROR;
        }

        if (pclcf->named) {
```

```
        if (pclcf->named) {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "location \"%V\" cannot be inside "
                               "the named location \"%V\"",
                               &clcf->name, &pclcf->name);
            return NGX_CONF_ERROR;
        }

        if (clcf->named) {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "named location \"%V\" can be "
                               "on the server level only",
                               &clcf->name);
            return NGX_CONF_ERROR;
        }

        len = pclcf->name.len;

#if (NGX_PCRE)
        if (clcf->regex == NULL
            && ngx_filename_cmp(clcf->name.data, pclcf->name.data, len) != 0)
#else
        if (ngx_filename_cmp(clcf->name.data, pclcf->name.data, len) != 0)
#endif
        {
            ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "location \"%V\" is outside location \"%V\"",
                               &clcf->name, &pclcf->name);
            return NGX_CONF_ERROR;
        }
    }

    /* 将ngx_http_location_queue_t添加到双向链表中 */
    if (ngx_http_add_location(cf, &pclcf->locations, clcf) != NGX_OK) {
        return NGX_CONF_ERROR;
    }

    save = *cf;
    cf->ctx = ctx;
    cf->cmd_type = NGX_HTTP_LOC_CONF;

    /* 解析当前location{}块下的所有loc级别配置项 */
    rv = ngx_conf_parse(cf, NULL);

    *cf = save;

    return rv;
}
```

loc 级别的配置项结构体之间的关系如下图所示：若 location 是精确匹配、正则表达式、@命名则 exact 字段有效，否则就是 inclusive 字段有效，画图过程中只画出exact 字段有效。

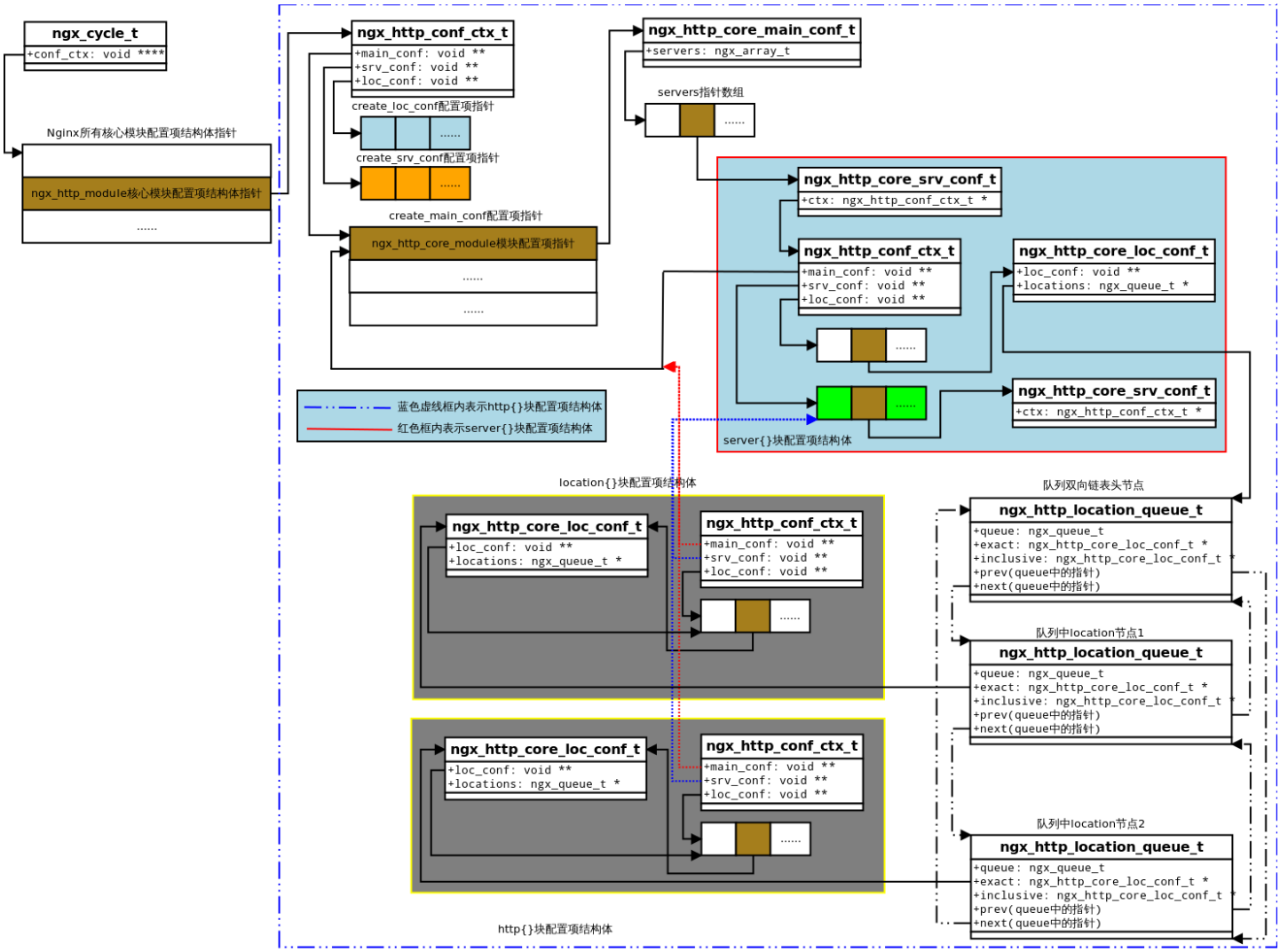


图 3 HTTP模块loc级别的配置项结构体

合并配置项

HTTP 框架解析完毕 http{} 块配置项时，会根据解析的结果进行合并配置项操作，即合并 http{}、server{}、location{} 不同级别下各 HTTP 模块生成的存放配置项的结构体。其合并过程在文件src/http/ngx_http.c中定义，如下所示：

- 若 HTTP 模块实现了 *merge_srv_conf* 方法，则将 http{} 块下由 *create_srv_conf* 生成的 *main* 级别结构体与遍历每一个 server{}块下由 *create_srv_conf*生成的*srv* 级别的配置项结构体进行 *merge_srv_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，则将 http{} 块下由 *create_loc_conf* 生成的 *main* 级别的配置项结构体与嵌套在每一个server{} 块下由 *create_loc_conf *生成的*srv级别的配置项结构体进行*merge_loc_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，由于在上一步骤已经将main、srv级别由*create_loc_conf* 生成的结构体进行合并，只要把上一步骤合并的结果在 server{} 块下与嵌套每一个location{}块下由*create_loc_conf* 生成的配置项结构体再次进行*merge_loc_conf* 操作；
- 若 HTTP 模块实现了 *merge_loc_conf* 方法，则将上一步骤的合并结果与与嵌套每一个location{}块下由 *create_loc_conf* 生成的的配置项结构体再次进行*merge_loc_conf* 操作；

具体合并过程如下图所示：

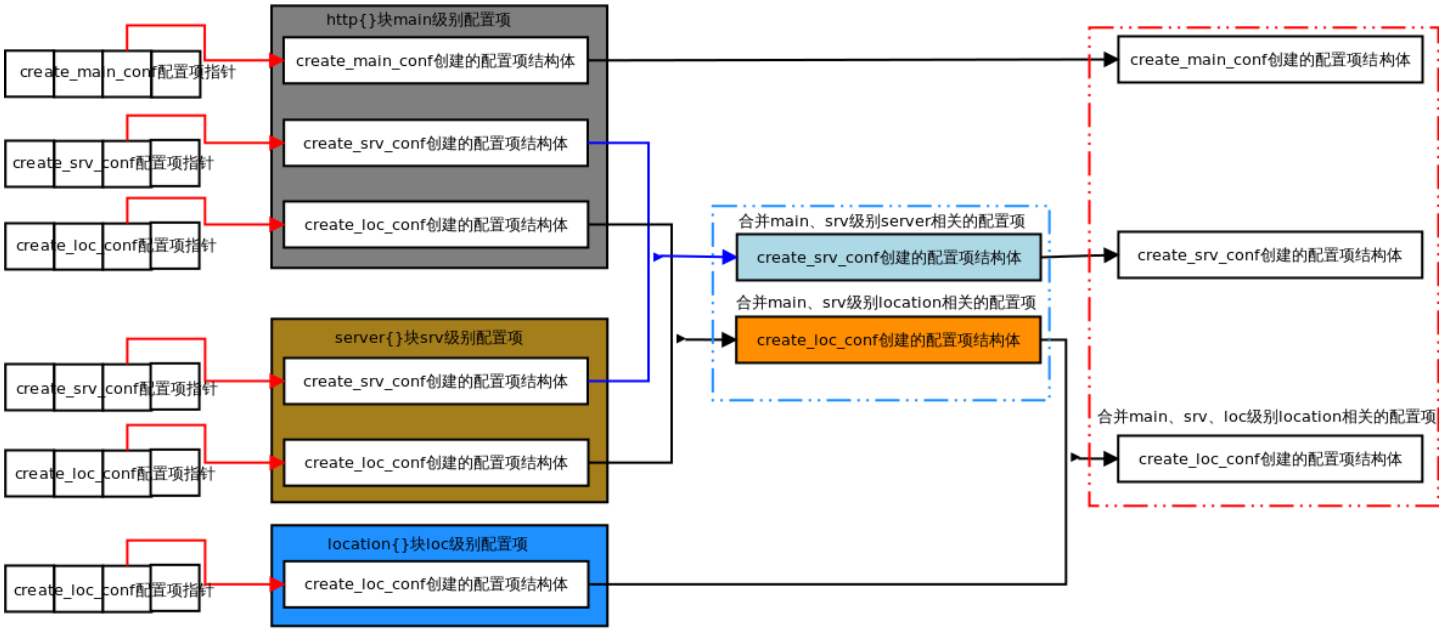


图 4 合并配置项

```
/* 合并配置项操作 */
static char *
ngx_http_merge_servers(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf,
    ngx_http_module_t *module, ngx_uint_t ctx_index)
{
    char                        *rv;
    ngx_uint_t                   s;
    ngx_http_conf_ctx_t         *ctx, saved;
    ngx_http_core_loc_conf_t    *clcf;
    ngx_http_core_srv_conf_t   **cscfp;

    /*
     * ngx_http_core_main_conf_t 中的成员servers是指针数组，
     * servers数组中的指针指向ngx_http_core_srv_conf_t结构体；
     */
    cscfp = cmcf->servers.elts;
    ctx = (ngx_http_conf_ctx_t *) cf->ctx;
    saved = *ctx;
    rv = NGX_CONF_OK;

    /* 遍历每一个server{}块内对应的ngx_http_core_srv_conf_t结构体 */
    for (s = 0; s < cmcf->servers.nelts; s++) {

        /* merge the server{}s' srv_conf's */

        /* srv_conf指向所有HTTP模块产生的server相关的srv级别配置项结构体 */
        ctx->srv_conf = cscfp[s]->ctx->srv_conf;

        /*
         * 这里合并http{}块下main、server{}块下srv级别与server相关的配置项结构；
         *
         * 若定义了merge_srv_conf 方法；
         * 则将当前HTTP模块在http{}块下由create_srv_conf 生成的结构体
         * 与遍历每个server{}块由create_srv_conf生成的配置项结构体进行merge_srv_conf合并操作；
         * saved.srv_conf[ctx_index]表示当前HTTP模块在http{}块下由create_srv_conf方法创建的结构体；
```

```
        * saved.srv_conf[ctx_index]表示当前HTTP模块在http{}块下由create_srv_conf方法创建的结构体；
        * cscfp[s]->ctx->srv_conf[ctx_index]表示当前HTTP模块在server{}块下由create_srv_conf方法创建的
结构体；
        */
        if (module->merge_srv_conf) {
            rv = module->merge_srv_conf(cf, saved.srv_conf[ctx_index],
                                        cscfp[s]->ctx->srv_conf[ctx_index]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }

        /*
         * 这里合并http{}块下main、server{}块下srv级别与location相关的配置项结构；
         *
         * 若定义了merge_loc_conf 方法；
         * 则将当前HTTP模块在http{}块下由create_loc_conf 生成的结构体
         * 与嵌套在server{}块内由create_loc_conf生成的配置项结构体进行merge_loc_conf合并操作；
         *
         * 其中saved.loc_conf[ctx_index]表示当前HTTP模块在http{}块下由create_loc_conf方法生成的配置项
结构体；
         * cscfp[s]->ctx->loc_conf[ctx_index]表示当前HTTP模块在server{}块下由create_loc_conf方法创建的
配置项结构体；
         */
        if (module->merge_loc_conf) {

            /* merge the server{}'s loc_conf */

            ctx->loc_conf = cscfp[s]->ctx->loc_conf;

            rv = module->merge_loc_conf(cf, saved.loc_conf[ctx_index],
                                        cscfp[s]->ctx->loc_conf[ctx_index]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }

            /* merge the locations{}' loc_conf's */

            /*
             * 若定义了merge_loc_conf 方法；
             * 则进行server{}块下create_loc_conf 生成的结构体与嵌套location{}块配置项生成的结构体进行mer
ge_loc_conf操作；
             */

            /* clcf表示ngx_http_core_module模块在server{}块下由create_loc_conf方法创建的ngx_http_core_l
oc_conf_t 结构体 */
            clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];

            rv = ngx_http_merge_locations(cf, clcf->locations,
                                          cscfp[s]->ctx->loc_conf,
                                          module, ctx_index);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }
```

```
    }

failed:

    *ctx = saved;

    return rv;
}

static char *
ngx_http_merge_locations(ngx_conf_t *cf, ngx_queue_t *locations,
    void **loc_conf, ngx_http_module_t *module, ngx_uint_t ctx_index)
{
    char                      *rv;
    ngx_queue_t               *q;
    ngx_http_conf_ctx_t       *ctx, saved;
    ngx_http_core_loc_conf_t  *clcf;
    ngx_http_location_queue_t  *lq;

    /* 若locations链表为空，即server{}块下没有嵌套location{}块，则立即返回 */
    if (locations == NULL) {
        return NGX_CONF_OK;
    }

    ctx = (ngx_http_conf_ctx_t *) cf->ctx;
    saved = *ctx;

    /*
     * 若定义了merge_loc_conf 方法；
     * 则进行location{}块下create_loc_conf 生成的结构体与嵌套location{}块配置项生成的结构体进行merge_loc_conf操作；
     */

    /* 遍历locations双向链表 */
    for (q = ngx_queue_head(locations);
         q != ngx_queue_sentinel(locations);
         q = ngx_queue_next(q))
    {
        lq = (ngx_http_location_queue_t *) q;

        /* exact 与 inclusive 的区别在文章中已经说过 */
        clcf = lq->exact ? lq->exact : lq->inclusive;
        /* 获取由create_loc_conf方法创建的结构体指针 */
        ctx->loc_conf = clcf->loc_conf;

        /* 合并srv、loc级别的location相关的配置项结构 */
        rv = module->merge_loc_conf(cf, loc_conf[ctx_index],
                                    clcf->loc_conf[ctx_index]);
        if (rv != NGX_CONF_OK) {
            return rv;
        }

        /*
         * 递归调用该函数；
         * 因为location{}继续内嵌location{}
```

```
            因为location{}还续内嵌location{}
             */
            rv = ngx_http_merge_locations(cf, clcf->locations, clcf->loc_conf,
                                          module, ctx_index);
            if (rv != NGX_CONF_OK) {
                return rv;
            }
        }

        *ctx = saved;

        return NGX_CONF_OK;
    }
```

HTTP 请求处理阶段

按照下列顺序将各个模块设置的phase handler依次加入cmcf->phase_engine.handlers列表，各个phase的phase handler的checker不同。checker主要用于限定某个phase的框架逻辑，包括处理返回值。 在Nginx 定义了 11 个处理阶段，有一部分是不能添加 phase handler 方法的。在文件src/http/ngx_http_core_module.h中定义，如下所示：

```
/* HTTP请求的11个处理阶段 */
typedef enum {
    /* 接收到完整的HTTP头部后处理的HTTP阶段，可自定义handler处理方法 */
    NGX_HTTP_POST_READ_PHASE = 0,

    /* 将请求的URI与location表达式匹配前，修改请求的URI的HTTP阶段，可自定义handler处理方法 */
    NGX_HTTP_SERVER_REWRITE_PHASE,

    /* 根据请求的URI寻找匹配的location表达式，只能由ngx_http_core_module模块实现，
     * 且不可自定义handler处理方法 */
    NGX_HTTP_FIND_CONFIG_PHASE,
    /* 在NGX_HTTP_FIND_CONFIG_PHASE阶段寻找到匹配的location之后再修改请求的URI，
     * 可自定义handler处理方法 */
    NGX_HTTP_REWRITE_PHASE,
    /* 在rewrite重写URI后，防止错误的nginx.conf配置导致死循环，
     * 只能用ngx_http_core_module模块处理，不可自定义handler处理方法 */
    NGX_HTTP_POST_REWRITE_PHASE,

    /* 在处理NGX_HTTP_ACCESS_PHASE阶段决定请求的访问权限前，处理该阶段，可自定义handler处理方
法 */
    NGX_HTTP_PREACCESS_PHASE,

    /* 由HTTP模块判断是否允许请求访问Nginx服务器，可自定义handler处理方法 */
    NGX_HTTP_ACCESS_PHASE,
    /* 向用户发送拒绝服务的错误响应，不可自定义handler处理方法 */
    NGX_HTTP_POST_ACCESS_PHASE,

    /* 使请求顺序的访问多个静态文件资源，不可自定义handler处理方法 */
    NGX_HTTP_TRY_FILES_PHASE,
    /* 处理HTTP请求内容，可自定义handler处理方法 */
    NGX_HTTP_CONTENT_PHASE,

    /* 处理完请求后记录日志阶段，可自定义handler处理方法 */
    NGX_HTTP_LOG_PHASE
} ngx_http_phases;
```

每个HTTP的checker方法与handler处理如下所示：

```
typedef struct ngx_http_phase_handler_s  ngx_http_phase_handler_t;

typedef ngx_int_t (*ngx_http_phase_handler_pt)(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph);

struct ngx_http_phase_handler_s {
    /*
     * 在HTTP框架中所有的checker方法都有ngx_http_core_module模块实现，其他普通模块不能对其重定义
;
     * 在处理某一个HTTP阶段时，HTTP框架会首先调用checker方法，然后在checker方法里面再调用handler
方法；
     */
    ngx_http_phase_handler_pt  checker;
    /* 由HTTP模块实现的handler方法处理HTTP阶段，一般用于普通HTTP模块 */
    ngx_http_handler_pt        handler;
    /* 下一个将要执行的HTTP阶段 */
    ngx_uint_t                 next;
};
```

完成 http{} 块的解析后，根据 *nginx.conf *文件中配置产生由 ngx_http_phase_handler_t 组成的数组，在处理 HTTP 请求时，一般情况下按照阶段的方向顺序 phase handler 加入到回调表中。ngx_http_phase_engine_t 结构体由所有ngx_http_phase_handler_t 组成的数组，如下所示：

```
typedef struct {
    /* 由ngx_http_phase_handler_t 构成的数组首地址，
     * 表示一个请求可能经历的所有ngx_http_handler_pt处理方法 */
    ngx_http_phase_handler_t  *handlers;
    /* 表示NGX_HTTP_SERVER_REWRITE_PHASE阶段第一个ngx_http_phase_handler_pt处理方法在handler
s数组中的序号；*/
    ngx_uint_t                server_rewrite_index;
    /* 表示NGX_HTTP_REWRITE_PHASE阶段第一个ngx_http_phase_handler_pt处理方法在handlers数组中
的序号；*/
    ngx_uint_t                location_rewrite_index;
} ngx_http_phase_engine_t;
```

ngx_http_phase_engine_t 中保存在当前 *nginx.conf* 配置下，一个用户请求可能经历的所有 ngx_http_handler_pt 处理方法。

```
typedef struct {
    /* 保存在每一个HTTP模块初始化时添加到当前阶段的处理方法 */
    ngx_array_t               handlers;
} ngx_http_phase_t;
```

在 HTTP模块初始化过程中，HTTP模块通过postconfiguration方法将自定义的方法添加到handler数组中，即该方法会被添加到phase_engine数组中。下面以NGX_HTTP_POST_READ_PHASE阶段为例，讲解了该阶段的checker方法的实现：

```
ngx_int_t
ngx_http_core_generic_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
{
    ngx_int_t  rc;

    /*
     * generic phase checker,
     * used by the post read and pre-access phases
     */

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "generic phase: %ui", r->phase_handler);

    /* 调用当前阶段各HTTP模块中的handler处理方法 */
    rc = ph->handler(r);

    /* 进入下一阶段处理，忽略当前阶段其他的处理方法 */
    if (rc == NGX_OK) {
        r->phase_handler = ph->next;
        return NGX_AGAIN;
    }

    /* 进入下一个处理方法，该处理方法可能属于当前阶段，也可能属于下一个阶段 */
    if (rc == NGX_DECLINED) {
        r->phase_handler++;
        return NGX_AGAIN;
    }

    /* 当前请求依旧处于当前处理阶段 */
    if (rc == NGX_AGAIN || rc == NGX_DONE) {
        return NGX_OK;
    }

    /* rc == NGX_ERROR || rc == NGX_HTTP_...  */

    /* 若出错，结束请求 */
    ngx_http_finalize_request(r, rc);

    return NGX_OK;
}
```

Nginx 中处理 HTTP 请求

概述

在 Nginx 的初始化启动过程中，worker 工作进程会调用事件模块的ngx_event_process_init 方法为每个监听套接字ngx_listening_t 分配一个 ngx_connection_t 连接，并设置该连接上读事件的回调方法 handler 为ngx_event_accept，同时将读事件挂载到epoll 事件机制中等待监听套接字连接上的可读事件发生，到此，Nginx 就可以接收并处理来自客户端的请求。当监听套接字连接上的可读事件发生时，即该连接上有来自客户端发出的连接请求，则会启动读事件的handler 回调方法ngx_event_accept，在 ngx_event_accept 方法中调用accept() 函数接收来自客户端的连接请求，成功建立连接之后，ngx_event_accept 函数调用监听套接字上的handler 回调方法ls->handler(c)（该回调方法就是 ngx_http_init_connection）。因此，成功建立连接之后由ngx_http_init_connection 方法开始处理该连接上的请求数据。

接收 HTTP 请求报文

在接收 HTTP 请求之前，首先会初始化已成功建立的连接；ngx_http_init_connection 函数的功能是设置读、写事件的回调方法，而实际上写事件的回调方法并不进行任何操作，读事件的回调方法是对 HTTP 请求进程初始化工作。

ngx_http_init_connection 函数的执行流程：

- 设置当前连接上写事件的回调方法 handler 为 ngx_http_empty_handler（实际上该方法不进行任何操作）；
- 设置当前连接上读事件的回调方法 handler 为 ngx_http_wait_request_handler；
- 检查当前连接上读事件是否准备就绪（即 ready 标志位为1）：
- 若读事件 ready 标志位为1，表示当前连接上有可读的TCP 流，则执行读事件的回调方法 ngx_http_wait_request_handler；
- 若读事件 ready 标志位为0，表示当前连接上没有可读的TCP 流，则将读事件添加到定时器事件机制中（监控可读事件是否超时），同时将读事件注册到epoll 事件机制中，等待可读事件的发生；

函数 ngx_http_init_connection 在文件src/http/ngx_http_request.c 中定义如下：

```
void
ngx_http_init_connection(ngx_connection_t *c)
{
    ngx_uint_t              i;
    ngx_event_t            *rev;
    struct sockaddr_in     *sin;
    ngx_http_port_t        *port;
```

```
    ngx_http_in_addr_t     *addr;
    ngx_http_log_ctx_t     *ctx;
    ngx_http_connection_t  *hc;
#if (NGX_HAVE_INET6)
    struct sockaddr_in6    *sin6;
    ngx_http_in6_addr_t    *addr6;
#endif

    /* 分配http连接ngx_http_connection_t结构体空间 */
    hc = ngx_pcalloc(c->pool, sizeof(ngx_http_connection_t));
    if (hc == NULL) {
        ngx_http_close_connection(c);
        return;
    }

    c->data = hc;

    /* find the server configuration for the address:port */

    port = c->listening->servers;

    if (port->naddrs > 1) {

        /*
         * there are several addresses on this port and one of them
         * is an "*:port" wildcard so getsockname() in ngx_http_server_addr()
         * is required to determine a server address
         */

        if (ngx_connection_local_sockaddr(c, NULL, 0) != NGX_OK) {
            ngx_http_close_connection(c);
            return;
        }

        switch (c->local_sockaddr->sa_family) {

#if (NGX_HAVE_INET6)
        ...
#endif

        default: /* AF_INET */
            sin = (struct sockaddr_in *) c->local_sockaddr;

            addr = port->addrs;

            /* the last address is "*" */

            for (i = 0; i < port->naddrs - 1; i++) {
                if (addr[i].addr == sin->sin_addr.s_addr) {
                    break;
                }
            }

            hc->addr_conf = &addr[i].conf;
```

```
            break;
        }

    } else {

        switch (c->local_sockaddr->sa_family) {

#if (NGX_HAVE_INET6)
        ...
#endif

        default: /* AF_INET */
            addr = port->addrs;
            hc->addr_conf = &addr[0].conf;
            break;
        }
    }

    /* the default server configuration for the address:port */
    hc->conf_ctx = hc->addr_conf->default_server->ctx;

    ctx = ngx_palloc(c->pool, sizeof(ngx_http_log_ctx_t));
    if (ctx == NULL) {
        ngx_http_close_connection(c);
        return;
    }

    ctx->connection = c;
    ctx->request = NULL;
    ctx->current_request = NULL;

    /* 设置当前连接的日志属性 */
    c->log->connection = c->number;
    c->log->handler = ngx_http_log_error;
    c->log->data = ctx;
    c->log->action = "waiting for request";

    c->log_error = NGX_ERROR_INFO;

    /* 设置当前连接读、写事件的handler处理方法 */
    rev = c->read;
    /* 设置当前连接读事件的处理方法handler为ngx_http_wait_request_handler */
    rev->handler = ngx_http_wait_request_handler;
    /*
     * 设置当前连接写事件的处理方法handler为ngx_http_empty_handler，
     * 该方法不执行任何实际操作，只记录日志；
     * 因为处理请求的过程不需要write方法；
     */
    c->write->handler = ngx_http_empty_handler;

#if (NGX_HTTP_SPDY)
    ...
#endif

#if (NGX_HTTP_SSL)
```

```
#if (NGX_HTTP_SSL)
    ...
#endif

    if (hc->addr_conf->proxy_protocol) {
        hc->proxy_protocol = 1;
        c->log->action = "reading PROXY protocol";
    }

    /* 若读事件准备就绪，则判断是否使用同步锁，
     * 根据同步锁情况判断决定是否立即处理该事件；
     */
    if (rev->ready) {
        /* the deferred accept(), rtsig, aio, iocp */

        /*
         * 若使用了同步锁ngx_use_accept_mutex，
         * 则将该读事件添加到待处理事件队列ngx_post_event中，
         * 直到退出锁时，才处理该读事件；
         */
        if (ngx_use_accept_mutex) {
            ngx_post_event(rev, &ngx_posted_events);
            return;
        }

        /* 若没有使用同步锁，则直接处理该读事件；
         * 读事件的处理函数handler为ngx_http_wait_request_handler；
         */
        rev->handler(rev);
        return;
    }

    /*
     * 若当前连接的读事件未准备就绪，
     * 则将其添加到定时器事件机制，并注册到epoll事件机制中；
     */

    /* 将当前连接的读事件添加到定时器机制中 */
    ngx_add_timer(rev, c->listening->post_accept_timeout);
    ngx_reusable_connection(c, 1);

    /* 将当前连接的读事件注册到epoll事件机制中 */
    if (ngx_handle_read_event(rev, 0) != NGX_OK) {
        ngx_http_close_connection(c);
        return;
    }
}
```

当连接上第一次出现可读事件时，会调用 ngx_http_wait_request_handler 函数，该函数的功能是初始化HTTP 请求，但是它并不会在成功建立连接之后就立刻初始化请求，而是在当前连接所对应的套接字缓冲区上确定接收到来自客户端的实际请求数据时才真正进行初始化工作，这样做可以减少不必要的内存消耗（若当成功建立连接之后，客户端并不进行实际数据通信，而此时Nginx 却因为初始化工作分配内

存）。

ngx_http_wait_request_handler 函数的执行流程：

- 首先判断当前读事件是否超时（即读事件的 timedout 标志位是否为1）：
- 若 timedout 标志位为1，表示当前读事件已经超时，则调用ngx_http_close_connection 方法关闭当前连接，return 从当前函数返回；
- 若 timedout 标志位为0，表示当前读事件还未超时，则继续检查当前连接的close标志位；
- 若当前连接的 close 标志位为1，表示当前连接要关闭，则调用ngx_http_close_connection 方法关闭当前连接，return 从当前函数返回；
- 若当前连接的 close 标志位为0，表示不需要关闭当前连接，进而调用recv() 函数尝试从当前连接所对应的套接字缓冲区中接收数据，这个步骤是为了确定客户端是否真正的发送请求数据，以免因为客户端不发送实际请求数据，出现初始化请求而导致内存被消耗。根据所读取的数据情况n 来判断是否要真正进行初始化请求工作：
- 若 n = NGX_AGAIN，表示客户端发起连接请求，但是暂时还没发送实际的数据，则将当前连接上的读事件添加到定时器机制中，同时将读事件注册到epoll 事件机制中，return 从当前函数返回；
- 若 n = NGX_ERROR，表示当前连接出错，则直接调用ngx_http_close_connection 关闭当前连接，return 从当前函数返回；
- 若 n = 0，表示客户端已经主动关闭当前连接，所有服务器端调用ngx_http_close_connection 关闭当前连接，return 从当前函数返回；
- 若 n 大于 0，表示读取到实际的请求数据，因此决定开始初始化当前请求，继续往下执行；
- 调用 ngx_http_create_request 方法构造ngx_http_request_t 请求结构体，并设置到当前连接的data 成员；
- 设置当前读事件的回调方法为 ngx_http_process_request_line，并执行该回调方法开始接收并解析请求行；

函数 ngx_http_wait_request_handler 在文件src/http/ngx_http_request.c 中定义如下：

```
/* 处理连接的可读事件 */
static void
ngx_http_wait_request_handler(ngx_event_t *rev)
{
    u_char                    *p;
    size_t                    size;
    ssize_t                   n;
    ngx_buf_t                 *b;
    ngx_connection_t          *c;
    ngx_http_connection_t     *hc;
    ngx_http_core_srv_conf_t  *cscf;
```

```
    /* 获取读事件所对应的连接ngx_connection_t 对象 */
    c = rev->data;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "http wait request handler");

    /* 若当前读事件超时，则记录错误日志，关闭所对应的连接并退出 */
    if (rev->timedout) {
        ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
        ngx_http_close_connection(c);
        return;
    }

    /* 若当前读事件所对应的连接设置关闭连接标志位，则关闭该链接 */
    if (c->close) {
        ngx_http_close_connection(c);
        return;
    }

    /* 若当前读事件不超时，且其所对应的连接不设置close标志位，则继续指向以下语句 */

    hc = c->data;
    /* 获取当前读事件请求的相关配置项结构 */
    cscf = ngx_http_get_module_srv_conf(hc->conf_ctx, ngx_http_core_module);

    size = cscf->client_header_buffer_size;

    /* 以下内容是接收缓冲区的操作 */
    b = c->buffer;

    /* 若当前连接的接收缓冲区不存在，则创建该接收缓冲区 */
    if (b == NULL) {
        b = ngx_create_temp_buf(c->pool, size);
        if (b == NULL) {
            ngx_http_close_connection(c);
            return;
        }

        c->buffer = b;

    } else if (b->start == NULL) {
        /* 若当前接收缓冲区存在，但是为空，则为其分配内存 */

        b->start = ngx_palloc(c->pool, size);
        if (b->start == NULL) {
            ngx_http_close_connection(c);
            return;
        }

        /* 初始化接收缓冲区各成员指针 */
        b->pos = b->start;
        b->last = b->start;
        b->end = b->last + size;
    }
```

```
    /* 在当前连接上开始接收HTTP请求数据 */
    n = c->recv(c, b->last, size);

    if (n == NGX_AGAIN) {

        if (!rev->timer_set) {
            ngx_add_timer(rev, c->listening->post_accept_timeout);
            ngx_reusable_connection(c, 1);
        }

        if (ngx_handle_read_event(rev, 0) != NGX_OK) {
            ngx_http_close_connection(c);
            return;
        }

        /*
         * We are trying to not hold c->buffer's memory for an idle connection.
         */

        if (ngx_pfree(c->pool, b->start) == NGX_OK) {
            b->start = NULL;
        }

        return;
    }

    if (n == NGX_ERROR) {
        ngx_http_close_connection(c);
        return;
    }

    if (n == 0) {
        ngx_log_error(NGX_LOG_INFO, c->log, 0,
                      "client closed connection");
        ngx_http_close_connection(c);
        return;
    }

    /* 若接收HTTP请求数据成功，则调整接收缓冲区成员指针 */
    b->last += n;

    if (hc->proxy_protocol) {
        hc->proxy_protocol = 0;

        p = ngx_proxy_protocol_parse(c, b->pos, b->last);

        if (p == NULL) {
            ngx_http_close_connection(c);
            return;
        }

        b->pos = p;

        if (b->pos == b->last) {
```

```
            c->log->action = "waiting for request";
            b->pos = b->start;
            b->last = b->start;
            ngx_post_event(rev, &ngx_posted_events);
            return;
        }
    }

    c->log->action = "reading client request line";

    ngx_reusable_connection(c, 0);

    /* 为当前连接创建一个请求结构体ngx_http_request_t */
    c->data = ngx_http_create_request(c);
    if (c->data == NULL) {
        ngx_http_close_connection(c);
        return;
    }

    /* 设置当前读事件的处理方法为ngx_http_process_request_line */
    rev->handler = ngx_http_process_request_line;
    /* 执行该读事件的处理方法ngx_http_process_request_line，接收HTTP请求行 */
    ngx_http_process_request_line(rev);
}
```

接收 HTTP 请求行

HTTP 请求的初始化完成之后会调用 ngx_http_process_request_line 方法开始接收并解析 HTTP 请求行。在 HTTP 协议中我们可以知道，请求行的长度并不是固定的，它与URI 长度相关，若当内核套接字缓冲区不能一次性完整的接收HTTP 请求行时，会多次调用ngx_http_process_request_line 方法继续接收，即ngx_http_process_request_line 方法重新作为当前连接上读事件的回调方法，必要时将读事件添加到定时器机制，注册到epoll 事件机制，直到接收并解析出完整的HTTP 请求行。

ngx_http_process_request_line 处理HTTP 请求行函数执行流程：

- 首先，判断当前请求是否超时，若超时（即读事件的 timedout 标志位为1），则设置当前连接的超时标志位为1（c->timedout = 1），调用ngx_http_close_request 方法关闭该请求，并return 从当前函数返回；
- 若当前请求未超时（读事件的 timedout 标志位为 0），调用 ngx_http_read_request_header 方法开始读取当前请求行，根据该函数的返回值n 进行以下判断：
- 若返回值 n = NGX_AGAIN，表示当前连接上套接字缓冲区不存在可读TCP 流，则需将当前读事件添加到定时器机制，注册到epoll 事件机制中，等待可读事件发生。return 从当前函数返回；
- 若返回值 n = NGX_ERROR，表示当前连接出错，则调用ngx_http_finalize_request 方法结束请求，return 从当前函数返回；
- 若返回值 n 大于 0，表示读取请求行成功，调用函数 ngx_http_parse_request_line 开始解析由函数

ngx_http_read_request_header 读取所返回的请求行，根据函数ngx_http_parse_request_line 函数返回值rc 不同进行判断；

- 若返回值 rc = NGX_ERROR，表示解析请求行时出错，此时，调用ngx_http_finalize_request 方法终止该请求，并return 从当前函数返回；
- 若返回值 rc = NGX_AGAIN，表示没有解析到完整的请求行，即仍需接收请求行，首先根据要求调整接收缓冲区header_in 的内存空间，则继续调用函数ngx_http_read_request_header 读取请求数据进入请求行自动处理机制，直到请求行解析完毕；
- 若返回值 rc = NGX_OK，表示解析到完整的 HTTP 请求行，则设置请求行的成员信息（例如：方法名称、URI 参数、HTTP 版本等信息）；
- 若 HTTP 协议版本小于 1.0 版本，表示不需要处理 HTTP 请求头部，则直接调用函数 ngx_http_process_request 处理该请求，return 从当前函数返回；
- 若HTTP协议版本不小于 1.0 版本，表示需要处理HTTP请求头部：
- 调用函数 ngx_list_init 初始化保存 HTTP 请求头部的结构体 ngx_http_request_t 中成员headers_in 链表容器（该链表缓冲区是保存所接收到的HTTP 请求数据）；
- 设置当前读事件的回调方法为 ngx_http_process_request_headers 方法，并调用该方法 ngx_http_process_request_headers 开始处理HTTP 请求头部。return 从当前函数返回；

函数 ngx_http_process_request_line 在文件src/http/ngx_http_request.c 中定义如下：

```
/* 处理HTTP请求行 */
static void
ngx_http_process_request_line(ngx_event_t *rev)
{
    ssize_t              n;
    ngx_int_t            rc, rv;
    ngx_str_t            host;
    ngx_connection_t    *c;
    ngx_http_request_t  *r;

    /* 获取当前读事件所对应的连接 */
    c = rev->data;
    /* 获取连接中所对应的请求结构 */
    r = c->data;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0,
                   "http process request line");

    /* 若当前读事件超时，则进行相应地处理，并关闭当前请求 */
    if (rev->timedout) {
        ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
        c->timedout = 1;
        ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);
        return;
```

```
    }

    /* 设置NGX_AGAIN标志，表示请求行还没解析完毕 */
    rc = NGX_AGAIN;

    for ( ;; ) {

        /* 若请求行还没解析完毕，则继续解析 */
        if (rc == NGX_AGAIN) {
            /* 读取当前请求未解析的数据 */
            n = ngx_http_read_request_header(r);

            /* 若没有数据，或读取失败，则直接退出 */
            if (n == NGX_AGAIN || n == NGX_ERROR) {
                return;
            }
        }

        /* 解析接收缓冲区header_in中的请求行 */
        rc = ngx_http_parse_request_line(r, r->header_in);

        /* 若请求行解析完毕 */
        if (rc == NGX_OK) {

            /* the request line has been parsed successfully */

            /* 设置请求行的成员，请求行是ngx_str_t类型 */
            r->request_line.len = r->request_end - r->request_start;
            r->request_line.data = r->request_start;
            /* 设置请求长度，包括请求头部、请求包体 */
            r->request_length = r->header_in->pos - r->request_start;

            ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                           "http request line: \"%V\"", &r->request_line);

            /* 设置请求方法名称字符串 */
            r->method_name.len = r->method_end - r->request_start + 1;
            r->method_name.data = r->request_line.data;

            /* 设置HTTP请求协议 */
            if (r->http_protocol.data) {
                r->http_protocol.len = r->request_end - r->http_protocol.data;
            }

            /* 处理请求中的URI */
            if (ngx_http_process_request_uri(r) != NGX_OK) {
                return;
            }

            if (r->host_start && r->host_end) {

                host.len = r->host_end - r->host_start;
                host.data = r->host_start;

                rc = ngx_http_validate_host(&host, r->pool, 0);
```

```
        rc = ngx_http_validate_host(&host, r->pool, 0);

        if (rc == NGX_DECLINED) {
            ngx_log_error(NGX_LOG_INFO, c->log, 0,
                          "client sent invalid host in request line");
            ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
            return;
        }

        if (rc == NGX_ERROR) {
            ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        if (ngx_http_set_virtual_server(r, &host) == NGX_ERROR) {
            return;
        }

        r->headers_in.server = host;
    }

    /* 设置请求协议版本 */
    if (r->http_version < NGX_HTTP_VERSION_10) {

        if (r->headers_in.server.len == 0
            && ngx_http_set_virtual_server(r, &r->headers_in.server)
               == NGX_ERROR)
        {
            return;
        }

        /* 若HTTP版本小于1.0版本，则表示不需要接收HTTP请求头部，则直接处理请求 */
        ngx_http_process_request(r);
        return;
    }

    /* 初始化链表容器，为接收HTTP请求头部做准备 */
    if (ngx_list_init(&r->headers_in.headers, r->pool, 20,
                      sizeof(ngx_table_elt_t))
        != NGX_OK)
    {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    c->log->action = "reading client request headers";

    /* 若请求行解析完毕，则接下来处理请求头部 */

    /* 设置连接读事件的回调方法 */
    rev->handler = ngx_http_process_request_headers;
    /* 开始处理HTTP请求头部 */
    ngx_http_process_request_headers(rev);

    return;
```

```
        }

        /* 解析请求行出错 */
        if (rc != NGX_AGAIN) {

            /* there was error while a request line parsing */

            ngx_log_error(NGX_LOG_INFO, c->log, 0,
                          ngx_http_client_errors[rc - NGX_HTTP_CLIENT_ERROR]);
            ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
            return;
        }

        /* NGX_AGAIN: a request line parsing is still incomplete */

        /* 请求行仍然未解析完毕，则继续读取请求数据 */

        /* 若当前接收缓冲区内存不够，则分配更大的内存空间 */
        if (r->header_in->pos == r->header_in->end) {

            rv = ngx_http_alloc_large_header_buffer(r, 1);

            if (rv == NGX_ERROR) {
                ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }

            if (rv == NGX_DECLINED) {
                r->request_line.len = r->header_in->end - r->request_start;
                r->request_line.data = r->request_start;

                ngx_log_error(NGX_LOG_INFO, c->log, 0,
                              "client sent too long URI");
                ngx_http_finalize_request(r, NGX_HTTP_REQUEST_URI_TOO_LARGE);
                return;
            }
        }
    }
}
```

在接收并解析请求行的过程中会调用 ngx_http_read_request_header 读取请求数据，我们看下该函数是如何读取到请求数据的。

ngx_http_read_request_header 读取请求数据函数执行流程：

- 检测当前请求的接收缓冲区 header_in 是否有数据，若有直接返回该数据n；
- 若接收缓冲区 header_in 没有数据，检查当前读事件是否准备就绪（即判断ready 标志位是否为0）：
- 若当前读事件未准备就绪（即当前读事件 ready 标志位为0），则设置返回值n= NGX_AGAIN；

- 若当前读事件已经准备就绪（即 ready 标志位为 1），则调用 recv() 方法从当前连接套接字中读取数据并保存到接收缓冲区header_in 中，并设置n 为recv()方法所读取的数据的返回值；
- 下面根据 n 的取值执行不同的操作：
- 若 n = NGX_AGAIN（此时，n 的值可能当前事件未准备就绪而设置的NGX_AGAIN，也可能是recv() 方法返回的NGX_AGAIN 值，但是只能是其中一种情况），将当前读事件添加到定时器事件机制中，将当前读事件注册到epoll 事件机制中，等待事件可读，n 从当前函数返回；
- 若 n = 0 或 n = ERROR，则调用 ngx_http_finalize_request 结束请求，并返回NGX_ERROR 退出当前函数；

函数 ngx_http_read_request_header 在文件src/http/ngx_http_request.c 中定义如下：

```
static ssize_t
ngx_http_read_request_header(ngx_http_request_t *r)
{
    ssize_t                   n;
    ngx_event_t               *rev;
    ngx_connection_t          *c;
    ngx_http_core_srv_conf_t  *cscf;

    /* 获取当前请求所对应的连接 */
    c = r->connection;
    /* 获取当前连接的读事件 */
    rev = c->read;

    /* 获取当前请求接收缓冲区的数据，header_in 是ngx_buf_t类型 */
    n = r->header_in->last - r->header_in->pos;

    /* 若接收缓冲区有数据，则直接返回该数据 */
    if (n > 0) {
        return n;
    }

    /* 若当前接收缓冲区没有数据，首先判断当前读事件是否准备就绪 */
    if (rev->ready) {
        /* 若当前读事件已准备就绪，则从其所对应的连接套接字读取数据，并保存到接收缓冲区中 */
        n = c->recv(c, r->header_in->last,
                    r->header_in->end - r->header_in->last);
    } else {
        /* 若接收缓冲区没有数据，且读事件未准备就绪，则设置为NGX_AGAIN */
        n = NGX_AGAIN;
    }

    /* 若接收缓冲区没有数据，且读事件未准备就绪，则设置为NGX_AGAIN */
    /* 将当前读事件添加到定时器机制；
     * 将当前读事件注册到epoll事件机制；
     */
    if (n == NGX_AGAIN) {
        if (!rev->timer_set) {
            cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
```

```
            /* 将当前读事件添加到定时器机制中 */
            ngx_add_timer(rev, cscf->client_header_timeout);
        }

        /* 将当前读事件注册到epoll事件机制中 */
        if (ngx_handle_read_event(rev, 0) != NGX_OK) {
            ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
            return NGX_ERROR;
        }

        return NGX_AGAIN;
    }

    if (n == 0) {
        ngx_log_error(NGX_LOG_INFO, c->log, 0,
                      "client prematurely closed connection");
    }

    if (n == 0 || n == NGX_ERROR) {
        c->error = 1;
        c->log->action = "reading client request headers";

        ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
        return NGX_ERROR;
    }

    r->header_in->last += n;

    return n;
}
```

接收 HTTP 请求头部

前面已经成功接收并解析了 HTTP 请求行，这里根据读事件的回调方法 ngx_http_process_request_headers 开始接收并解析HTTP 请求头部，但是并不一定能够一次性接收到完整的HTTP 请求头部，因此，可以多次调用该函数，直到接收到完整的HTTP 请求头部。

ngx_http_process_request_headers 处理HTTP 请求头部函数执行流程：

- 首先，判断当前请求读事件是否超时，若超时（即读事件的 timedout 标志位为1），则设置当前连接超时标志位为1（c->timedout = 1），并调用ngx_http_close_request 方法关闭该请求，并return 从当前函数返回；
- 若当前请求读事件未超时（即读事件的 timedout 标志位为0），检查接收HTTP 请求头部的 header_in 缓冲区是否有剩余内存空间，若没有剩余的内存空间，则调用 ngx_http_alloc_large_header_buffer 方法分配更大的缓冲区。若有剩余的内存，则无需再分配内存空间。
- 调用 ngx_http_read_request_header 方法开始读取当前请求头部保存到header_in 缓冲区中，根据

该函数的返回值 n 进行以下判断：

- 若返回值 n = NGX_AGAIN，表示当前连接上套接字缓冲区不存在可读TCP 流，则需将当前读事件添加到定时器机制，注册到epoll 事件机制中，等待可读事件发生。return 从当前函数返回；
- 若返回值 n = NGX_ERROR，表示当前连接出错，则调用ngx_http_finalize_request 方法结束请求，return 从当前函数返回；
- 若返回值 n 大于 0，表示读取请求头部成功，调用函数 ngx_http_parse_request_line 开始解析由函数ngx_http_read_request_header 读取所返回的请求头部，根据函数 ngx_http_parse_request_line 函数返回值rc不同进行判断；
- 若返回值 rc = NGX_ERROR，表示解析请求行时出错，此时，调用ngx_http_finalize_request 方法终止该请求，并return 从当前函数返回；
- 若返回值 rc = NGX_AGAIN，表示没有解析到完整一行的请求头部，仍需继续接收TCP 字符流才能够是完整一行的请求头部，则continue 继续调用函数ngx_http_read_request_header 和 ngx_http_parse_request_line 方法读取并解析下一行请求头部，直到全部请求头部解析完毕；
- 若返回值 rc = NGX_OK，表示解析出一行 HTTP 请求头部（*注意：一行请求头部只是整个请求头部的一部分*），判断当前解析出来的一行请求头部是否合法，若非法，则忽略当前一行请求头部，继续读取并解析下一行请求头部。若合法，则调用ngx_list_push 方法将该行请求头部设置到当前请求 ngx_http_request_t 结构体 header_in 缓冲区成员的headers 链表中，设置请求头部名称的hash 值，并continue 继续调用函数ngx_http_read_request_header 和ngx_http_parse_request_line 方法读取并解析下一行请求头部，直到全部请求头部解析完毕；
- 若返回值 rc = NGX_HTTP_PARSE_HEADER_DONE，则表示已经读取并解析出全部请求头部，此时，调用ngx_http_process_request 方法开始处理请求，return 从当前函数返回；

函数 ngx_http_process_request_headers 在文件src/http/ngx_http_request.c 中定义如下：

```
/* 处理HTTP请求头部 */
static void
ngx_http_process_request_headers(ngx_event_t *rev)
{
    u_char                  *p;
    size_t                  len;
    ssize_t                 n;
    ngx_int_t               rc, rv;
    ngx_table_elt_t         *h;
    ngx_connection_t        *c;
    ngx_http_header_t       *hh;
    ngx_http_request_t      *r;
    ngx_http_core_srv_conf_t  *cscf;
    ngx_http_core_main_conf_t  *cmcf;

    /* 获取当前读事件所对应的连接 */
    c = rev->data;
```

```
    /* 获取当前连接的HTTP请求 */
    r = c->data;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0,
                   "http process request header line");

    /* 若当前读事件超时，则关闭该请求，并退出 */
    if (rev->timedout) {
        ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
        c->timedout = 1;
        ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);
        return;
    }

    /* 获取ngx_http_core_module模块的main级别配置项结构 */
    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    /* 表示当前请求头部未解析完毕 */
    rc = NGX_AGAIN;

    for ( ;; ) {

        if (rc == NGX_AGAIN) {
            /* 若当前请求头部未解析完毕，则首先判断接收缓冲区是否有内存空间再次接收请求数据 */

            if (r->header_in->pos == r->header_in->end) {

                /* 若接收缓冲区没有足够内存空间，则分配更大的内存空间 */
                rv = ngx_http_alloc_large_header_buffer(r, 0);

                if (rv == NGX_ERROR) {
                    ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                    return;
                }

                if (rv == NGX_DECLINED) {
                    p = r->header_name_start;

                    r->lingering_close = 1;

                    if (p == NULL) {
                        ngx_log_error(NGX_LOG_INFO, c->log, 0,
                                      "client sent too large request");
                        ngx_http_finalize_request(r,
                                          NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
                        return;
                    }

                    len = r->header_in->end - p;

                    if (len > NGX_MAX_ERROR_STR - 300) {
                        len = NGX_MAX_ERROR_STR - 300;
                        p[len++] = '.'; p[len++] = '.'; p[len++] = '.';
                    }
```

```
                ngx_log_error(NGX_LOG_INFO, c->log, 0,
                              "client sent too long header line: \"%*s\"",
                              len, r->header_name_start);

                ngx_http_finalize_request(r,
                                  NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
                return;
            }
        }

        /* 读取未解析请求数据 */
        n = ngx_http_read_request_header(r);

        /* 若没有可读的数据，或读取失败，则直接退出 */
        if (n == NGX_AGAIN || n == NGX_ERROR) {
            return;
        }
    }

    /* the host header could change the server configuration context */

    /* 获取ngx_http_core_module模块的srv级别配置项结构 */
    cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);

    /* 开始解析HTTP请求头部 */
    rc = ngx_http_parse_header_line(r, r->header_in,
                                    cscf->underscores_in_headers);

    /* 解析出一行请求头部（注意：一行请求头部只是HTTP请求头部的一部分） */
    if (rc == NGX_OK) {

        /* 设置当前请求的长度 */
        r->request_length += r->header_in->pos - r->header_name_start;

        /*
         * 若当前解析出来的一行请求头部是非法的，或Nginx当前版本不支持，
         * 则记录错误日志，并继续解析下一行请求头部；
         */
        if (r->invalid_header && cscf->ignore_invalid_headers) {

            /* there was error while a header line parsing */

            ngx_log_error(NGX_LOG_INFO, c->log, 0,
                          "client sent invalid header line: \"%*s\"",
                          r->header_end - r->header_name_start,
                          r->header_name_start);
            continue;
        }

        /* a header line has been parsed successfully */

        /*
         * 若当前解析出来的一行请求头部是合法的，表示成功解析出该行请求头部，
         * 将该行请求头部保存在当前请求的headers_in的headers链表中；
```

```
             * 接着继续解析下一行请求头部；
             */
            h = ngx_list_push(&r->headers_in.headers);
            if (h == NULL) {
                ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }

            /* 设置请求头部名称的hash值 */
            h->hash = r->header_hash;

            h->key.len = r->header_name_end - r->header_name_start;
            h->key.data = r->header_name_start;
            h->key.data[h->key.len] = '\0';

            h->value.len = r->header_end - r->header_start;
            h->value.data = r->header_start;
            h->value.data[h->value.len] = '\0';

            h->lowcase_key = ngx_pnalloc(r->pool, h->key.len);
            if (h->lowcase_key == NULL) {
                ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }

            if (h->key.len == r->lowcase_index) {
                ngx_memcpy(h->lowcase_key, r->lowcase_header, h->key.len);

            } else {
                ngx_strlow(h->lowcase_key, h->key.data, h->key.len);
            }

            hh = ngx_hash_find(&cmcf->headers_in_hash, h->hash,
                               h->lowcase_key, h->key.len);

            if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
                return;
            }

            ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                           "http header: \"%V: %V\"",
                           &h->key, &h->value);

            continue;
        }

        /* 若成功解析所有请求头部，则接下来就开始处理该请求 */
        if (rc == NGX_HTTP_PARSE_HEADER_DONE) {

            /* a whole header has been parsed successfully */

            ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                           "http header done");

            r->request_length += r->header_in->pos - r->header_name_start;
```

```
            r->request_length += r->header_in->pos - r->header_name_start;

            /* 设置当前请求的解析状态 */
            r->http_state = NGX_HTTP_PROCESS_REQUEST_STATE;

            /*
             * 调用该函数主要目的有两个：
             * 1、根据HTTP头部的host字段，调用ngx_http_find_virtual_server查找虚拟主机的配置块；
             * 2、对HTTP请求头部协议版本进行检查，例如http1.1版本，host头部不能为空，否则会返回400 Ba
d Request错误；
             */
            rc = ngx_http_process_request_header(r);

            if (rc != NGX_OK) {
                return;
            }

            /* 开始处理当前请求 */
            ngx_http_process_request(r);

            return;
        }

        /* 表示当前行的请求头部未解析完毕，则继续读取请求数据进行解析 */
        if (rc == NGX_AGAIN) {

            /* a header line parsing is still not complete */

            continue;
        }

        /* rc == NGX_HTTP_PARSE_INVALID_HEADER: "\r" is not followed by "\n" */

        /* 解析请求头部出错，则关闭该请求，并退出 */
        ngx_log_error(NGX_LOG_INFO, c->log, 0,
                      "client sent invalid header line: \"%*s\\r...\"",
                      r->header_end - r->header_name_start,
                      r->header_name_start);
        ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
        return;
    }
}
```

处理 HTTP 请求

前面的步骤已经接收到完整的 HTTP 请求头部，此时，已经有足够的信息开始处理HTTP 请求。处理 HTTP 请求的过程有11 个 HTTP 阶段，在不同的阶段由各个 HTTP 模块进行处理。有关多阶段处理请求的描述可参考《HTTP request processing phases in Nginx》；

ngx_http_process_request 处理HTTP 请求函数执行流程：

- 若当前读事件在定时器机制中，则调用 ngx_del_timer 函数将其从定时器机制中移除，因为在处理

HTTP 请求时不存在接收HTTP 请求头部超时的问题；

- 由于处理 HTTP 请求不需要再接收 HTTP 请求行或头部，则需重新设置当前连接读、写事件的回调方法，读、写事件的回调方法都设置为 ngx_http_request_handler，即后续处理 HTTP 请求的过程都是通过该方法进行；
- 设置当前请求 ngx_http_request_t 结构体中的成员read_event_handler 的回调方法为 ngx_http_block_reading，该回调方法实际不做任何操作，即在处理请求时不会对请求的读事件进行任何处理，除非某个HTTP模块重新设置该回调方法；
- 接下来调用函数 ngx_http_handler 开始处理HTTP 请求；
- 调用函数 ngx_http_run_posted_requests 处理post 子请求；

函数 ngx_http_process_request 在文件src/http/ngx_http_request.c 中定义如下：

```
/* 处理HTTP请求 */
void
ngx_http_process_request(ngx_http_request_t *r)
{
    ngx_connection_t  *c;

    /* 获取当前请求所对应的连接 */
    c = r->connection;

#if (NGX_HTTP_SSL)
    ...

#endif

    /*
     * 由于现在不需要再接收HTTP请求头部超时问题，
     * 则需要把当前连接的读事件从定时器机制中删除；
     * timer_set为1表示读事件已添加到定时器机制中，
     * 则将其从定时器机制中删除，0表示不在定时器机制中；
     */
    if (c->read->timer_set) {
        ngx_del_timer(c->read);
    }

#if (NGX_STAT_STUB)
    ...
#endif

    /* 重新设置当前连接的读、写事件的回调方法 */
    c->read->handler = ngx_http_request_handler;
    c->write->handler = ngx_http_request_handler;

    /*
     * 设置请求读事件的回调方法，
     * 其实ngx_http_block_reading函数实际对读事件不做任何处理；
     * 即在处理请求时，不会对读事件任何操作，除非有HTTP模块重新设置处理方法；
     */
    r->read_event_handler = ngx_http_block_reading;

    /* 开始处理各个HTTP模块的handler方法，该函数定义于ngx_http_core_module.c中*/
    ngx_http_handler(r);

    /* 处理post请求 */
    ngx_http_run_posted_requests(c);
}
```

ngx_http_handler 函数的执行流程：

- 检查当前请求 ngx_http_request_t 的 internal 标志位：
- 若 internal 标志位为 0，表示当前请求不需要重定向，判断是否使用 keepalive 机制，并设置 phase_handler 序号为0，表示执行ngx_http_phase_engine_t 结构成员ngx_http_phase_handler_t

*handlers数组中的第一个回调方法；

- 若 internal 标志位为 1，表示需要将当前请求做内部跳转，并将 phase_handler 设置为 server_rewriter_index，表示执行ngx_http_phase_engine_t 结构成员ngx_http_phase_handler_t *handlers 数组在NGX_HTTP_SERVER_REWRITE_PHASE 处理阶段的第一个回调方法；
- 设置当前请求 ngx_http_request_t 的成员写事件write_event_handler 为 ngx_http_core_run_phases；
- 执行n gx_http_core_run_phases 方法；

函数 ngx_http_handler 在文件 src/http/ngx_http_core_module.c 中定义如下：

```
void
ngx_http_handler(ngx_http_request_t *r)
{
    ngx_http_core_main_conf_t  *cmcf;

    r->connection->log->action = NULL;

    r->connection->unexpected_eof = 0;

    /* 若当前请求的internal标志位为0，表示不需要重定向 */
    if (!r->internal) {
        /* 下面语句是决定是否使用keepalive机制 */
        switch (r->headers_in.connection_type) {
        case 0:
            r->keepalive = (r->http_version > NGX_HTTP_VERSION_10);
            break;

        case NGX_HTTP_CONNECTION_CLOSE:
            r->keepalive = 0;
            break;

        case NGX_HTTP_CONNECTION_KEEP_ALIVE:
            r->keepalive = 1;
            break;
        }

        /* 设置延迟关闭标志位 */
        r->lingering_close = (r->headers_in.content_length_n > 0
                              || r->headers_in.chunked);
        /*
         * phase_handler序号设置为0，表示执行ngx_http_phase_engine_t结构体成员
         * ngx_http_phase_handler_t *handlers数组中的第一个回调方法；
         */
        r->phase_handler = 0;

    } else {
    /* 若当前请求的internal标志位为1，表示需要做内部跳转 */
        /* 获取ngx_http_core_module模块的main级别的配置项结构 */
        cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
```

```
        /*
         * 将phase_handler序号设为server_rewriter_index，
         * 该phase_handler序号是作为ngx_http_phase_engine_t结构中成员
         * ngx_http_phase_handler_t *handlers回调方法数组的序号，
         * 即表示回调方法在该数组中所处的位置；
         *
         * server_rewrite_index则是handlers数组中NGX_HTTP_SERVER_REWRITE_PHASE阶段的
         * 第一个ngx_http_phase_handler_t回调的方法；
         */
        r->phase_handler = cmcf->phase_engine.server_rewrite_index;
    }

    r->valid_location = 1;
#if (NGX_HTTP_GZIP)
    r->gzip_tested = 0;
    r->gzip_ok = 0;
    r->gzip_vary = 0;
#endif

    /* 设置当前请求写事件的回调方法 */
    r->write_event_handler = ngx_http_core_run_phases;
    /*
     * 执行该回调方法，将调用各个HTTP模块共同处理当前请求，
     * 各个HTTP模块按照11个HTTP阶段进行处理；
     */
    ngx_http_core_run_phases(r);
}
```

ngx_http_core_run_phases 函数的执行流程：

- 判断每个 ngx_http_phase_handler_t 处理阶段是否实现checker 方法：
- 若实现 checker 方法，则执行 phase_handler 序号在 ngx_http_phase_handler_t *handlers数组中指定的checker 方法；执行完checker 方法，若返回NGX_OK 则退出；若返回非NGX_OK，则继续执行下一个HTTP 模块在该阶段的checker 方法；
- 若没有实现 checker 方法，则直接退出；

函数 ngx_http_core_run_phases 在文件src/http/ngx_http_core_module.c 中定义如下：

```
void
ngx_http_core_run_phases(ngx_http_request_t *r)
{
    ngx_int_t                   rc;
    ngx_http_phase_handler_t   *ph;
    ngx_http_core_main_conf_t  *cmcf;

    /* 获取ngx_http_core_module模块的main级别的配置项结构体 */
    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    /* 获取各个HTTP模块处理请求的回调方法数组 */
    ph = cmcf->phase_engine.handlers;

    /* 若实现了checker方法 */
    while (ph[r->phase_handler].checker) {

        /* 执行phase_handler序号在数组中指定的checker方法 */
        rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);

        /* 成功执行checker方法，则退出，否则继续执行下一个HTTP模块的checker方法 */
        if (rc == NGX_OK) {
            return;
        }
    }
}
```

处理子请求

post 子请求是基于 subrequest 机制的，首先看下 post 子请求结构体类型：

```
/* 子请求的单链表结构 */
typedef struct ngx_http_posted_request_s  ngx_http_posted_request_t;

struct ngx_http_posted_request_s {
    /* 指向当前待处理子请求的ngx_http_request_t结构体 */
    ngx_http_request_t              *request;
    /* 指向下一个子请求 */
    ngx_http_posted_request_t       *next;
};
```

在请求结构体 ngx_http_request_t 中有一个与post 子请求相关的成员posted_requests，该成员把各个post 子请求按照子请求结构体ngx_http_posted_request_t 的结构连接成单链表的形式，请求结构体 ngx_http_request_t 中main 成员是子请求的原始请求，parent 成员是子请求的父请求。下面是子请求的处理过程。

ngx_http_run_posted_requests 函数执行流程：

- 判断当前连接是否已被销毁（即标志位 destroyed 是否为0），若被销毁则直接return 退出，否则继续执行；

- 获取原始请求的子请求链表，若子请求链表为空（表示没有 post 请求）则直接return 退出，否则继续执行；
- 遍历子请求链表，执行每个 post 请求的写事件回调方法write_event_handler；

函数 ngx_http_run_posted_requests 在文件src/http/ngx_http_request.c 中定义如下：

```
void
ngx_http_run_posted_requests(ngx_connection_t *c)
{
    ngx_http_request_t         *r;
    ngx_http_log_ctx_t         *ctx;
    ngx_http_posted_request_t  *pr;

    for ( ;; ) {

        /* 若当前连接已被销毁，则直接退出 */
        if (c->destroyed) {
            return;
        }

        /* 获取当前连接所对应的请求 */
        r = c->data;
        /* 获取原始请求的子请求单链表 */
        pr = r->main->posted_requests;

        /* 若子请求单链表为空，则直接退出 */
        if (pr == NULL) {
            return;
        }

        /* 将原始请求的posted_requests指向单链表的下一个post请求 */
        r->main->posted_requests = pr->next;

        /* 获取子请求链表中的第一个post请求 */
        r = pr->request;

        ctx = c->log->data;
        ctx->current_request = r;

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http posted request: \"%V?%V\"", &r->uri, &r->args);

        /*
         * 调用当前post请求写事件的回调方法write_event_handler；
         * 子请求不被网络事件驱动，因此不需要调用read_event_handler；
         */
        r->write_event_handler(r);
    }
}
```

处理 HTTP 请求包体

下面开始要分析 HTTP 框架是如何处理HTTP 请求包体，HTTP 框架有两种处理请求包体的方法：接收请求包体、丢弃请求包体；但是必须要注意的是丢弃请求包体并不意味着就不接受请求包体，只是把接收到的请求包体进行丢弃，不进一步对其进行处理。

其中有一个很重要的成员就是请求结构体 ngx_http_request_t 中的引用计数count，引用计数是用来决定是否真正结束当前请求，若引用计数为0 时，表示没有其他动作在处理该请求，则可以终止该请求；若引用计数不为0 时，表示当前请求还有其他动作在操作，因此不能结束当前请求，以免发生错误；那怎么样控制这个引用计数呢？例如，当一个请求添加新事件，或是把一些原本从定时器、epoll 事件机制中移除的事件从新加入到其中等等，出现这些情况都是要对引用计数增加1；当要结束请求时，首先会把引用计数减1，并判断该引用计数是否为0，再进一步判断是否决定真的结束当前请求。

接收 HTTP 请求包体

HTTP 请求包体保存在结构体 ngx_http_request_body_t 中，该结构体是存放在保存着请求结构体 ngx_http_request_t 的成员 request_body 中，该结构体定义如下：

```
/* 存储HTTP请求包体的结构体ngx_http_request_body_t */
typedef struct {
    /* 存放HTTP请求包体的临时文件 */
    ngx_temp_file_t                 *temp_file;
    /*
     * 指向接收HTTP请求包体的缓冲区链表表头，
     * 因为当一个缓冲区ngx_buf_t无法容纳所有包体时，就需要多个缓冲区形成链表；
     */
    ngx_chain_t                     *bufs;
    /* 指向当前保存HTTP请求包体的缓冲区 */
    ngx_buf_t                       *buf;
    /*
     * 根据content-length头部和已接收包体长度，计算还需接收的包体长度；
     * 即当前剩余的请求包体大小；
     */
    off_t                           rest;
    /* 接收HTTP请求包体缓冲区链表空闲缓冲区 */
    ngx_chain_t                     *free;
    /* 接收HTTP请求包体缓冲区链表已使用的缓冲区 */
    ngx_chain_t                     *busy;
    /* 保存chunked的解码状态，供ngx_http_parse_chunked方法使用 */
    ngx_http_chunked_t              *chunked;
    /*
     * HTTP请求包体接收完毕后执行的回调方法；
     * 即ngx_http_read_client_request_body方法传递的第 2 个参数；
     */
    ngx_http_client_body_handler_pt    post_handler;
} ngx_http_request_body_t;
```

接收 HTTP 请求包体 ngx_http_read_client_request_body 函数执行流程：

- 原始请求引用计算 r->main->count 增加 1；引用计数 count 的管理是：当逻辑开启流程时，引用计

数就增加1，结束此流程时，引用计数就减1。在ngx_http_read_client_request_body 函数中，首先将原始请求的引用计数增加1，当遇到异常终止时，引用计数会在该函数返回之前减1；若正常结束时，引用计数由post_handler回调方法继续维护；

- 判断当前请求包体是否已被完整接收（r->request_body 为1）或被丢弃（r->discard_body为1），若满足其中一个则不需要再次接收请求包体，直接执行post_handler 回调方法，并NGX_OK 从当前函数返回；
- 若需要接收 HTTP 请求包体，则首先调用 ngx_http_test_expect 方法，检查客户端是否发送 Expect:100-continue 头部期望发送请求包体，服务器会回复 HTTP/1.1 100 Continue 表示允许客户端发送请求包体；
- 分配当前请求 ngx_http_request_t 结构体request_body 成员，准备接收请求包体；
- 检查请求的 content-length 头部，若请求头部的 content-length 字段小于0，则表示不需要继续接收请求包体（即已经接收到完整的请求包体），直接执行post_handler 回调方法，并 NGX_OK 从当前函数返回；
- 若请求头部的 content-length 字段大于 0，则表示需要继续接收请求包体。首先判断当前请求 ngx_http_request_t 的header_in 成员是否存在未处理数据，若存在未被处理的数据，表示该缓冲区 header_in 在接收请求头部期间已经预接收了请求包体，因为在接收HTTP 请求头部期间有可能预接收请求包体，由于在接收请求包体之前，请求头部已经被接收完毕，所以若该缓冲区存在未被处理的数据，那就是请求包体。
- 若 header_in 缓冲区存在未被处理的数据，即是预接收的请求包体，首先检查缓冲区请求包体长度 preread 是否大于请求包体长度的content-length 字段，若大于则表示已经接收到完整的HTTP 请求包体，不需要继续接收，则执行post_handler 回调方法；
- 若 header_in 缓冲区存在未被处理的数据，即是预接收的请求包体，但是缓冲区请求包体长度 preread 小于请求包体长度的content-length 字段，表示已接收的请求包体不完整，则需要继续接收请求包体。调用函数ngx_http_request_body_filte 解析并把已接收的请求包体挂载到请求 ngx_http_request_t r 的 request_body->bufs，header_in 缓冲区剩余的空间足够接收剩余的请求包体大小rest，则不需要分配新的缓冲区，进而设置当前请求ngx_http_request_t 的 read_event_handler 读事件回调方法为ngx_http_read_client_request_body_handler，写事件 write_event_handler 回调方法为ngx_http_request_empty_handler (即不执行任何操作)，然后调用方法ngx_http_do_read_client_request_body 真正接收HTTP 请求包体，该方法将TCP 连接上的套接字缓冲区中的字符流全部读取出来，并判断是否需要写入到临时文件，以及是否接收全部的请求包体，同时在接收到完整包体后执行回调方法post_handler；
- 若 header_in 缓冲区存在未被处理的数据，即是预接收的请求包体，但是缓冲区请求包体长度 preread 小于请求包体长度的content-length 字段，或者header_in 缓冲区不存在未被处理的数据，且header_in 剩余的空间不足够接收HTTP 请求包体，则会重新分配接收请求包体的缓冲区，再进而设置当前请求ngx_http_request_t 的read_event_handler 读事件回调方法为

ngx_http_read_client_request_body_handler，写事件write_event_handler 回调方法为
ngx_http_request_empty_handler (即不执行任何操作)，然后调用方法
ngx_http_do_read_client_request_body 真正接收HTTP 请求包体；

函数 ngx_http_read_client_request_body 在文件src/http/ngx_http_request_body.c 中定义如下：

```
/* 接收HTTP请求包体 */
ngx_int_t
ngx_http_read_client_request_body(ngx_http_request_t *r,
    ngx_http_client_body_handler_pt post_handler)
{
    size_t                     preread;
    ssize_t                    size;
    ngx_int_t                  rc;
    ngx_buf_t                  *b;
    ngx_chain_t                out, *cl;
    ngx_http_request_body_t    *rb;
    ngx_http_core_loc_conf_t  *clcf;

    /*
     * 当有逻辑开启流程时，引用计数会增加1，此流程结束时，引用计数将减1；
     * 在ngx_http_read_client_request_body方法中，首先将原始请求引用计数增加1，
     * 当遇到异常终止时，则在该函数返回前会将引用计数减1,；
     * 若正常结束时，引用计数由post_handler方法继续维护；
     */
    /* 原始请求的引用计数count加1 */
    r->main->count++;

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream && r == r->main) {
        rc = ngx_http_spdy_read_request_body(r, post_handler);
        goto done;
    }
#endif

    /* HTTP请求包体未被处理时，request_body结构是不被分配的，只有处理时才会分配 */
    /*
     * 若当前HTTP请求不是原始请求，或HTTP请求包体已被读取或被丢弃；
     * 则直接执行HTTP模块的回调方法post_handler，并返回NGX_OK；
     */
    if (r != r->main || r->request_body || r->discard_body) {
        post_handler(r);
        return NGX_OK;
    }

    /*
     * ngx_http_test_expect 用于检查客户端是否发送Expect:100-continue头部，
     * 若客户端已发送该头部表示期望发送请求包体数据，则服务器回复HTTP/1.1 100 Continue；
     * 具体意义是：客户端期望发送请求包体，服务器允许客户端发送，
     * 该函数返回NGX_OK；
     */
    if (ngx_http_test_expect(r) != NGX_OK) {
        rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
```

```
    goto done;
}

/* 只有在确定要接收请求包体时才分配存储HTTP请求包体的结构体 ngx_http_request_body_t 空间 */
rb = ngx_pcalloc(r->pool, sizeof(ngx_http_request_body_t));
if (rb == NULL) {
    rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
    goto done;
}

/*
 * set by ngx_pcalloc():
 *
 *     rb->bufs = NULL;
 *     rb->buf = NULL;
 *     rb->free = NULL;
 *     rb->busy = NULL;
 *     rb->chunked = NULL;
 */

/* 初始化存储请求包体结构成员 */
rb->rest = -1;/* 待接收HTTP请求包体的大小 */
rb->post_handler = post_handler;/* 接收完包体后的回调方法 */

/* 令当前请求的post_body成员指向存储请求包体结构 */
r->request_body = rb;

/*
 * 若指定HTTP请求包体的content_length字段小于0，则表示不需要接收包体；
 * 执行post_handler方法，并返回；
 */
if (r->headers_in.content_length_n < 0 && !r->headers_in.chunked) {
    post_handler(r);
    return NGX_OK;
}

/* 若指定HTTP请求包体的content_length字段大于0，则表示需要接收包体；*/

/*
 * 在请求结构ngx_http_request_t 成员中header_in缓冲区保存的是HTTP请求头部，
 * 由于在处理HTTP请求之前，HTTP头部已被完整接收，所以若header_in缓冲区里面
 * 还存在未处理的数据，则证明在接收HTTP请求头部期间，已经预接收了HTTP请求包体；
 */
preread = r->header_in->last - r->header_in->pos;

/*
 * 若header_in缓冲区存在预接收的HTTP请求包体，
 * 则计算还需接收HTTP请求包体的大小rest；
 */
if (preread) {

    /* there is the pre-read part of the request body */

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http client request body preread %uz", preread);
```

```
                http client request body preread %uz", preread);

    /* 将out的缓冲区指向header_in缓冲区中的请求包体数据 */
    out.buf = r->header_in;
    out.next = NULL;

    /*
     * 将预接收的HTTP请求包体数据添加到r->request_body->bufs中，
     * 即将请求包体存储在新分配的ngx_http_request_body_t rb 结构体的bufs中；
     */
    rc = ngx_http_request_body_filter(r, &out);

    if (rc != NGX_OK) {
        goto done;
    }

    /* 若ngx_http_request_body_filter返回NGX_OK，则继续执行以下程序 */

    /* 更新当前HTTP请求长度：包括请求头部与请求包体 */
    r->request_length += preread - (r->header_in->last - r->header_in->pos);

    /*
     * 若已接收的请求包体不完整，即rest大于0，表示需要继续接收请求包体；
     * 若此时header_in缓冲区仍然有足够的剩余空间接收剩余的请求包体长度，
     * 则不再分配缓冲区内存；
     */
    if (!r->headers_in.chunked
        && rb->rest > 0
        && rb->rest <= (off_t) (r->header_in->end - r->header_in->last))
    {
        /* the whole request body may be placed in r->header_in */

        b = ngx_calloc_buf(r->pool);
        if (b == NULL) {
            rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
            goto done;
        }

        b->temporary = 1;
        b->start = r->header_in->pos;
        b->pos = r->header_in->pos;
        b->last = r->header_in->last;
        b->end = r->header_in->end;

        rb->buf = b;

        /* 设置当前请求读事件的回调方法 */
        r->read_event_handler = ngx_http_read_client_request_body_handler;
        r->write_event_handler = ngx_http_request_empty_handler;

        /*
         * 真正开始接收请求包体数据；
         * 将TCP套接字连接缓冲区中当前的字符流全部读取出来，
         * 并判断是否需要写入临时文件，以及是否接收全部的请求包体，
         * 同时在接收到完整包体后执行回调方法post_handler；
```

```
            */
            rc = ngx_http_do_read_client_request_body(r);
            goto done;
        }

    } else {
        /*
         * 若在接收HTTP请求头部过程没有预接收HTTP请求包体数据，
         * 或者预接收了不完整的HTTP请求包体，但是header_in缓冲区不够继续存储剩余的包体；
         * 进一步计算待需接收HTTP请求的大小rest；
         */
        /* set rb->rest */

        if (ngx_http_request_body_filter(r, NULL) != NGX_OK) {
            rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
            goto done;
        }
    }

    /* 若rest为0，表示无需继续接收HTTP请求包体，即已接收到完整的HTTP请求包体 */
    if (rb->rest == 0) {/* 若已接收完整的HTTP请求包体 */
        /* the whole request body was pre-read */

        /*
         * 检查client_body_in_file_only配置项是否打开，若打开，
         * 则将r->request_body->bufs中的包体数据写入到临时文件；
         */
        if (r->request_body_in_file_only) {
            if (ngx_http_write_request_body(r) != NGX_OK) {
                rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
                goto done;
            }

            if (rb->temp_file->file.offset != 0) {

                cl = ngx_chain_get_free_buf(r->pool, &rb->free);
                if (cl == NULL) {
                    rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
                    goto done;
                }

                b = cl->buf;

                ngx_memzero(b, sizeof(ngx_buf_t));

                b->in_file = 1;
                b->file_last = rb->temp_file->file.offset;
                b->file = &rb->temp_file->file;

                rb->bufs = cl;

            } else {
                rb->bufs = NULL;
            }
        }
```

```
        }

        /* 执行回调方法 */
        post_handler(r);

        return NGX_OK;
    }

    /* rest小于0表示出错 */
    if (rb->rest < 0) {
        ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
                      "negative request body rest");
        rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
        goto done;
    }

    /* 若rest大于0，则表示需要继续接收HTTP请求包体数据，执行以下程序 */

    /* 获取ngx_http_core_module模块的loc级别配置项结构 */
    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    /* 获取缓存请求包体的buffer缓冲区大小 */
    size = clcf->client_body_buffer_size;
    size += size >> 2;

    /* TODO: honor r->request_body_in_single_buf */

    if (!r->headers_in.chunked && rb->rest < size) {
        size = (ssize_t) rb->rest;

        if (r->request_body_in_single_buf) {
            size += preread;
        }

    } else {
        size = clcf->client_body_buffer_size;
    }

    rb->buf = ngx_create_temp_buf(r->pool, size);
    if (rb->buf == NULL) {
        rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
        goto done;
    }

    /* 设置当前请求读事件的回调方法 */
    r->read_event_handler = ngx_http_read_client_request_body_handler;
    r->write_event_handler = ngx_http_request_empty_handler;

    /* 接收请求包体 */
    rc = ngx_http_do_read_client_request_body(r);

done:

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        r->main->count--;
```

```
    }

    return rc;
}
```

读取 HTTP 请求包体 ngx_http_do_read_client_request_body 函数执行流程：

- 若 request_body->buf 缓冲区没有剩余的空间，则先调用函数ngx_http_write_request_body 将该缓冲区的数据写入到文件中；此时，该缓冲区就有空间；或者 request_body->buf 缓冲区有剩余的空间；接着分别计算request_body->buf 缓冲区所剩余的可用空间大小 size、待接收 HTTP 请求包体的长度 rest；若当前缓冲区剩余大小足够接收HTTP 请求包体，即size > rest，则调用recv 方法从 TCP 连接套接字缓冲区中读取请求包体数据到当前缓冲区request_body->buf 中，下面根据recv 方法的返回值n 做不同的判断：
- 返回值 n 为 NGX_AGAIN，表示 TCP 连接套接字缓冲区上的字符流未读取完毕，则需继续读取；
- 返回值 n 为 0 或 NGX_ERROR，表示读取失败，设置当前请求的errno 标志位错误编码，并退出；
- 返回值 n 不是以上的值，则表示读取成功，此时，更新当缓冲区request_body->buf的使用情况，更新当前请求的长度。判断已成功读取的长度n 是否等于待接收HTTP 请求包体的长度rest，若n = rest，则将已读取的请求包体挂载到当前请求的request body->buf链表中；并重新更新待接收的剩余请求包体长度rest 值；
- 根据 rest 值判断是否已经接收到完整的 HTTP 请求包体：
- rest 值大于 0，表示未接收到完整的 HTTP 请求包体，且当前套接字缓冲区已经没有可读数据，则需要调用函数ngx_add_timer 将当前连接的读事件添加到定时器机制，调用函数 ngx_handler_read_event 将当前连接读事件注册到epoll 事件机制中，等待可读事件的发生；此时，ngx_http_do_read_client_reuqest_body 返回NGX_AGAIN；
- rest 等于 0，表示已经接收到完整的 HTTP 请求包体，则把读事件从定时器机制移除，把缓冲区数据写入到文件中，设置读事件的回调方法为ngx_http_block_reading（不进行任何操作），最后执行 post_handler 回调方法；

函数 ngx_http_do_read_client_request_body 在文件src/http/ngx_http_request_body.c 中定义如下：

```
/* 读取HTTP请求包体 */
static ngx_int_t
ngx_http_do_read_client_request_body(ngx_http_request_t *r)
{
    off_t                     rest;
    size_t                    size;
    ssize_t                   n;
    ngx_int_t                 rc;
    ngx_buf_t                 *b;
    ngx_chain_t               *cl, out;
    ngx_connection_t          *c;
    ngx_http_request_body_t   *rb;
```

```
    ngx_http_request_body_t    *rb;
    ngx_http_core_loc_conf_t  *clcf;

    /* 获取当前请求所对应的连接 */
    c = r->connection;
    /* 获取当前请求的请求包体结构体 */
    rb = r->request_body;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http read client request body");

    for ( ;; ) {
        for ( ;; ) {
            /* 若当前缓冲区buf已满 */
            if (rb->buf->last == rb->buf->end) {

                /* pass buffer to request body filter chain */

                out.buf = rb->buf;
                out.next = NULL;

                rc = ngx_http_request_body_filter(r, &out);

                if (rc != NGX_OK) {
                    return rc;
                }

                /* write to file */

                /* 将缓冲区的字符流写入文件 */
                if (ngx_http_write_request_body(r) != NGX_OK) {
                    return NGX_HTTP_INTERNAL_SERVER_ERROR;
                }

                /* update chains */

                rc = ngx_http_request_body_filter(r, NULL);

                if (rc != NGX_OK) {
                    return rc;
                }

                if (rb->busy != NULL) {
                    return NGX_HTTP_INTERNAL_SERVER_ERROR;
                }

                /* 由于已经将当前缓冲区的字符流写入到文件，则该缓冲区有空间继续使用 */
                rb->buf->pos = rb->buf->start;
                rb->buf->last = rb->buf->start;
            }

            /* 计算当前缓冲区剩余的可用空间size */
            size = rb->buf->end - rb->buf->last;
            /* 计算需要继续接收请求包体的大小rest */
            rest = rb->rest - (rb->buf->last - rb->buf->pos);
```

```
        /* 若当前缓冲区有足够的空间接收剩余的请求包体，则不需要再分配缓冲区 */
        if ((off_t) size > rest) {
            size = (size_t) rest;
        }

        /* 从TCP连接套接字读取请求包体，并保存到当前缓冲区 */
        n = c->recv(c, rb->buf->last, size);

        ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http client request body recv %z", n);

        /* 若连接上套接字字符流还未读取完整，则继续读取 */
        if (n == NGX_AGAIN) {
            break;
        }

        if (n == 0) {
            ngx_log_error(NGX_LOG_INFO, c->log, 0,
                          "client prematurely closed connection");
        }

        /* 读取错误，设置错误编码 */
        if (n == 0 || n == NGX_ERROR) {
            c->error = 1;
            return NGX_HTTP_BAD_REQUEST;
        }

        /* 调整当前缓冲区的使用情况 */
        rb->buf->last += n;
        /* 设置已接收HTTP请求长度 */
        r->request_length += n;

        /* 若已完整接收HTTP请求包体，则将该包体数据存储到r->request_body->bufs中 */
        if (n == rest) {
            /* pass buffer to request body filter chain */

            out.buf = rb->buf;
            out.next = NULL;

            /* 将已读取的请求包体数据挂载到r->request_body->bufs中，并重新计算rest值 */
            rc = ngx_http_request_body_filter(r, &out);

            if (rc != NGX_OK) {
                return rc;
            }
        }

        if (rb->rest == 0) {
            break;
        }

        if (rb->buf->last < rb->buf->end) {
            break;
        }
```

```
        }

        ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http client request body rest %O", rb->rest);

        if (rb->rest == 0) {
            break;
        }

        /*
         * 若未接接收到完整的HTTP请求包体，且当前连接读事件未准备就绪，
         * 则需将读事件添加到定时器机制，注册到epoll事件机制中，等待可读事件发生；
         */
        if (!c->read->ready) {
            clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
            ngx_add_timer(c->read, clcf->client_body_timeout);

            if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
                return NGX_HTTP_INTERNAL_SERVER_ERROR;
            }

            return NGX_AGAIN;
        }
    }

    /* 到此，已经完整接收到HTTP请求，则需要将读事件从定时器机制中移除 */
    if (c->read->timer_set) {
        ngx_del_timer(c->read);
    }

    /* 若设置将请求包体保存到临时文件，则必须将缓冲区的请求包体数据写入到文件中 */
    if (rb->temp_file || r->request_body_in_file_only) {

        /* save the last part */

        /* 将缓冲区的请求包体数据写入到文件中 */
        if (ngx_http_write_request_body(r) != NGX_OK) {
            return NGX_HTTP_INTERNAL_SERVER_ERROR;
        }

        if (rb->temp_file->file.offset != 0) {

            cl = ngx_chain_get_free_buf(r->pool, &rb->free);
            if (cl == NULL) {
                return NGX_HTTP_INTERNAL_SERVER_ERROR;
            }

            b = cl->buf;

            ngx_memzero(b, sizeof(ngx_buf_t));

            b->in_file = 1;
            b->file_last = rb->temp_file->file.offset;
            b->file = &rb->temp_file->file;
```

```
            rb->bufs = cl;

        } else {
            rb->bufs = NULL;
        }
    }

    /*
     * 由于已经完成请求包体的接收，则需重新设置读事件的回调方法；
     * read_event_handler 设置为 ngx_http_block_reading 表示阻塞读事件
     * 即再有读事件发生将不会做任何处理；
     */
    r->read_event_handler = ngx_http_block_reading;

    /* 接收HTTP请求包体完毕后，调用回调方法post_handler */
    rb->post_handler(r);

    return NGX_OK;
}
```

ngx_http_read_client_request_body_handler 方法执行流程：

- 检查连接上读事件 timeout 标志位是否超时，若超时则调用函数ngx_http_finalize_request 终止当前请求；
- 若不超时，调用函数 ngx_http_do_read_client_request_body 开始读取HTTP 请求包体数据；

函数 ngx_http_read_client_request_body_handler 在文件src/http/ngx_http_request_body.c 中定义如下：

```
static void
ngx_http_read_client_request_body_handler(ngx_http_request_t *r)
{
    ngx_int_t  rc;

    /* 检查连接上读事件timeout标志位是否超时，若超时则终止该请求 */
    if (r->connection->read->timedout) {
        r->connection->timedout = 1;
        ngx_http_finalize_request(r, NGX_HTTP_REQUEST_TIME_OUT);
        return;
    }

    /* 开始接收HTTP请求包体数据 */
    rc = ngx_http_do_read_client_request_body(r);

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        ngx_http_finalize_request(r, rc);
    }
}
```

丢弃 HTTP 请求包体

当 HTTP 框架不需要请求包体时，会在接收完该请求包体之后将其丢弃，并不会进行下一步处理，以下是相关函数。

丢弃 HTTP 请求包体 ngx_http_discard_request_body 函数执行流程：

- 若当前是子请求，或请求包体已经被完整接收，或请求包体已被丢弃，则不需要继续，直接返回 NGX_OK 结束该函数；
- 调用函数 ngx_http_test_expect 检查客户端是否要发送请求包体，若服务器允许发送，则继续执行；
- 若当前请求连接上的读事件在定时器机制中（即 timer_set 标志位为1），则将该读事件从定时器机制中移除（丢弃请求包体不需要考虑超时问题，除非设置linger_timer）；
- 由于此时，待丢弃包体长度 content_length_n 为请求content-length 头部字段大小，所有判断content-length头部字段是否小于0，若小于0，表示已经成功丢弃完整的请求包体，直接返回 NGX_OK；若大于0，表示需要继续丢弃请求包体，则继续执行；
- 检查当前请求的 header_in 缓冲区是否预接收了 HTTP 请求，设此时 header_in 缓冲区里面未处理的数据大小为size，若size 不为0，表示已经预接收了HTTP 请求包体数据，则调用函数 ngx_http_discard_request_body_filter 将该请求包体丢弃，并根据已经预接收请求包体长度和请求 content-length 头部字段长度，重新计算需要待丢弃请求包体的长度content_length_n 的值；根据 ngx_http_discard_request_body_filter 函数的返回值rc 进行不同的判断：
- 若 rc = NGX_OK，且 content_length_n 的值为 0，则表示已经接收到完整请求包体，并将其丢弃；
- 若 rc ！= NGX_OK，则表示需要继续接收请求包体，根据content_length_n 的值来表示待丢弃请求包体的长度；
- 若还需继续丢弃请求包体，则调用函数 ngx_http_read_discard_request_body 读取剩余的请求包体数据，并将其丢弃；并根据该函数返回值rc 不同进行判断：
- 若 rc = NGX_OK，表示已成功丢弃完整的请求包体；
- 若 rc ！= NGX_OK，则表示接收到请求包体依然不完整，且此时连接套接字上已经没有剩余数据可读，则设置当前请求读事件的回调方法read_event_handler 为 ngx_http_discarded_request_body_handler，并调用函数ngx_handle_read_event 将该请求连接上的读事件注册到epoll 事件机制中，等待可读事件发生以便继续读取请求包体；同时将引用计数增加1（防止继续丢弃包体），当前请求的discard_body 标志位设置为1，表示正在丢弃，并返回 NGX_OK（这里并不表示已经成功丢弃完整的请求包体，只是表示ngx_http_discard_request_body 执行完毕，接下来的是等待读事件发生并继续丢弃包体）；

函数 ngx_http_discard_request_body 在文件src/http/ngx_http_request_body.c 中定义如下：

```
/* 丢弃HTTP请求包体 */
```

```
/* 丢弃HTTP请求包体 */
ngx_int_t
ngx_http_discard_request_body(ngx_http_request_t *r)
{
    ssize_t       size;
    ngx_int_t     rc;
    ngx_event_t  *rev;

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream && r == r->main) {
        r->spdy_stream->skip_data = NGX_SPDY_DATA_DISCARD;
        return NGX_OK;
    }
#endif

    /*
     * 若当前HTTP请求不是原始请求，或HTTP请求包体已被读取或被丢弃；
     * 则直接返回NGX_OK；
     */
    if (r != r->main || r->request_body || r->discard_body) {
        return NGX_OK;
    }

    /*
     * ngx_http_test_expect 用于检查客户端是否发送Expect:100-continue头部，
     * 若客户端已发送该头部表示期望发送请求包体数据，则服务器回复HTTP/1.1 100 Continue；
     * 具体意义是：客户端期望发送请求包体，服务器允许客户端发送，
     * 该函数返回NGX_OK；
     */
    if (ngx_http_test_expect(r) != NGX_OK) {
        rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    /* 获取当前连接的读事件 */
    rev = r->connection->read;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0, "http set discard body");

    /* 若读事件在定时器机制中，则将其移除 */
    if (rev->timer_set) {
        ngx_del_timer(rev);
    }

    /* 若请求content-length头部字段小于0，直接返回NGX_OK */
    if (r->headers_in.content_length_n <= 0 && !r->headers_in.chunked) {
        return NGX_OK;
    }

    /* 获取当前请求header_in缓冲区中预接收请求包体数据 */
    size = r->header_in->last - r->header_in->pos;

    /* 若已经预接收了HTTP请求包体数据 */
    if (size || r->headers_in.chunked) {
        /*
         * 丢弃预接收请求包体数据，并根据预接收请求包体大小与请求content-length头部大小，
```

```
         * 重新计算content_length_n的值；
         */
        rc = ngx_http_discard_request_body_filter(r, r->header_in);

        /* 若rc不为NGX_OK表示预接收的请求包体数据不完整，需继续接收 */
        if (rc != NGX_OK) {
            return rc;
        }

        /* 若返回rc=NGX_OK，且待丢弃请求包体大小content-length_n为0，表示已丢弃完整的请求包体 */
        if (r->headers_in.content_length_n == 0) {
            return NGX_OK;
        }
    }

    /* 读取剩余的HTTP请求包体数据，并将其丢弃 */
    rc = ngx_http_read_discarded_request_body(r);

    /* 若已经读取到完整请求包体，则返回NGX_OK */
    if (rc == NGX_OK) {
        r->lingering_close = 0;/* 不需要延迟关闭请求 */
        return NGX_OK;
    }

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        return rc;
    }

    /* rc == NGX_AGAIN */

    /*
     * 若读取到的请求包体依然不完整，但此时已经没有剩余数据可读，
     * 则将当前请求读事件回调方法设置为ngx_http_discard_request_body_handler，
     * 并将读事件注册到epoll事件机制中，等待可读事件发生以便继续读取请求包体；
     */
    r->read_event_handler = ngx_http_discarded_request_body_handler;

    if (ngx_handle_read_event(rev, 0) != NGX_OK) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    /* 由于已将读事件注册到epoll事件机制中，则引用计数增加1，discard_body标志为1 */
    r->count++;
    r->discard_body = 1;

    return NGX_OK;
}
```

ngx_http_discarded_request_body_handler 函数执行流程如下：

- 判断当前请求连接上的读事件是否超时，若超时（即标志位 timeout 为1），则调用函数 ngx_http_finalize_request 将引用计数减1，若此时引用计数为0，则终止当前请求；

- 调用函数 ngx_http_read_discarded_request_body 开始读取请求包体，并将所读取的请求包体丢弃；同时根据该函数的返回值rc 不同进行判断：
- 若返回值 rc = NGX_OK，表示已经接收到完整请求包体，并成功将其丢弃，则此时设置discard_body 标志位为0，设置lingering_close 标志位为0，并调用函数ngx_http_finalize_request 结束当前请求；
- 若返回值 rc ！= NGX_OK，则表示读取的请求包体依旧不完整，调用函数ngx_handle_read_event 将读事件注册到epoll 事件机制中，等待可读事件发生；

函数 ngx_http_discarded_request_body_handler 在文件src/http/ngx_http_request_body.c 中定义如下：

```
void
ngx_http_discarded_request_body_handler(ngx_http_request_t *r)
{
    ngx_int_t               rc;
    ngx_msec_t              timer;
    ngx_event_t            *rev;
    ngx_connection_t       *c;
    ngx_http_core_loc_conf_t  *clcf;

    c = r->connection;
    rev = c->read;

    /*
     * 判断读事件是否超时，若超时则调用ngx_http_finalize_request方法将引用计数减1，
     * 若此时引用计数是0，则直接终止该请求；
     */
    if (rev->timedout) {
        c->timedout = 1;
        c->error = 1;
        ngx_http_finalize_request(r, NGX_ERROR);
        return;
    }

    /* 若需要延迟关闭，则设置延迟关闭连接的时间 */
    if (r->lingering_time) {
        timer = (ngx_msec_t) r->lingering_time - (ngx_msec_t) ngx_time();

        if ((ngx_msec_int_t) timer <= 0) {
            r->discard_body = 0;
            r->lingering_close = 0;
            ngx_http_finalize_request(r, NGX_ERROR);
            return;
        }

    } else {
        timer = 0;
    }

    /* 读取剩余请求包体，并将其丢弃 */
```

```
    /* 读取剩余请求包体，并将其丢弃 */
    rc = ngx_http_read_discarded_request_body(r);

    /* 若返回rc=NGX_OK，则表示已接收到完整请求包体，并成功将其丢弃 */
    if (rc == NGX_OK) {
        r->discard_body = 0;
        r->lingering_close = 0;
        ngx_http_finalize_request(r, NGX_DONE);
        return;
    }

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        c->error = 1;
        ngx_http_finalize_request(r, NGX_ERROR);
        return;
    }

    /* rc == NGX_AGAIN */

    /* 若读取的请求包体依旧不完整，则再次将读事件注册到epoll事件机制中 */
    if (ngx_handle_read_event(rev, 0) != NGX_OK) {
        c->error = 1;
        ngx_http_finalize_request(r, NGX_ERROR);
        return;
    }

    /* 若设置了延迟，则将读事件添加到定时器事件机制中 */
    if (timer) {

        clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

        timer *= 1000;

        if (timer > clcf->lingering_timeout) {
            timer = clcf->lingering_timeout;
        }

        ngx_add_timer(rev, timer);
    }
}
```

ngx_http_read_discarded_request_body 函数执行流程：

- 若待丢弃请求包体长度 content_length_n 为0，表示已经接收到完整请求包体，并成功将其丢弃，则此时，设置读事件的回调方法为ngx_http_block_reading（不进行任何操作），同时返回NGX_OK，表示已成功丢弃完整请求包体；
- 若需要继续丢弃请求包体数据，且此时，连接上套接字缓冲区没有可读数据，即读事件未准备就绪，则返回 NGX_AGAIN，表示需要等待读事件再次被触发时继续读取请求包体并丢弃；
- 调用函数 recv 读取请求包体数据，根据不同返回值 n，进行不同的判断：

- 若返回值 n = NGX_AGAIN，表示读取的请求包体依旧不完整，需要等待下次读事件被触发，继续读取请求包体数据；
- 若 n = NGX_ERROR 或 n = 0，表示客户端主动关闭当前连接，则不需要读取请求包体，即直接返回NGX_OK，表示结束丢弃包体动作；
- 若返回值 n = NGX_OK，则表示读取请求包体成功，此时调用函数ngx_http_discard_request_body_filter 将已经读取的请求包体丢弃，并更新content_length_n 的值；根据content_length_n 的值进行判断是否继续读取请求包体数据（此时又回到步骤1，因此是一个for 循环），直到读取到完整的请求包体，并将其丢弃，才结束for 循环，并从该函数返回；

函数 ngx_http_read_discarded_request_body 在文件src/http/ngx_http_request_body.c 中定义如下：

```
/* 读取请求包体，并将其丢弃 */
static ngx_int_t
ngx_http_read_discarded_request_body(ngx_http_request_t *r)
{
    size_t     size;
    ssize_t    n;
    ngx_int_t  rc;
    ngx_buf_t  b;
    u_char     buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http read discarded body");

    ngx_memzero(&b, sizeof(ngx_buf_t));

    b.temporary = 1;

    for ( ;; ) {
        /* 若待丢弃的请求包体大小content_length_n 为0，表示不需要接收请求包体 */
        if (r->headers_in.content_length_n == 0) {
            /* 重新设置读事件的回调方法，其实该回调方法不进行任何操作 */
            r->read_event_handler = ngx_http_block_reading;
            return NGX_OK;
        }

        /* 若当前连接的读事件未准备就绪，则不能读取数据，即返回NGX_AGAIN */
        if (!r->connection->read->ready) {
            return NGX_AGAIN;
        }

        /* 若需要读取请求包体数据，计算需要读取请求包体的大小size */
        size = (size_t) ngx_min(r->headers_in.content_length_n,
                                NGX_HTTP_DISCARD_BUFFER_SIZE);

        /* 从连接套接字缓冲区读取请求包体数据 */
        n = r->connection->recv(r->connection, buffer, size);
```

```
        if (n == NGX_ERROR) {
            r->connection->error = 1;
            return NGX_OK;
        }

        /* 若读取的请求包体数据不完整，则继续读取 */
        if (n == NGX_AGAIN) {
            return NGX_AGAIN;
        }

        /* 若n=0或n=NGX_ERROR表示读取失败，即该连接已关闭，则不需要丢弃包体 */
        if (n == 0) {
            return NGX_OK;
        }

        /* 若返回n=NGX_OK ，表示读取到完整的请求包体，则将其丢弃 */
        b.pos = buffer;
        b.last = buffer + n;

        /* 将读取的完整请求包体丢弃 */
        rc = ngx_http_discard_request_body_filter(r, &b);

        if (rc != NGX_OK) {
            return rc;
        }
    }
}
```

发送 HTTP 响应报文

HTTP 的响应报文由 Filter 模块处理并发送，Filter 模块包括过滤头部（Header Filter）和过滤包体（Body Filter），Filter 模块过滤头部处理 HTTP 响应头部（HTTP headers），Filter 包体处理HTTP 响应包体（response content）。HTTP 响应报文发送的过程需要经过Nginx 的过滤链，所谓的过滤链就是由多个过滤模块组成有序的过滤链表，每个链表元素就是对应过滤模块的处理方法。在HTTP 框架中，定义了过滤链表表头（即链表的第一个元素，也是处理方法）如下：

```
extern ngx_http_output_header_filter_pt  ngx_http_top_header_filter;/* 发送响应头部 */
extern ngx_http_output_body_filter_pt  ngx_http_top_body_filter;/* 发送响应包体 */
```

其中，ngx_http_output_header_filter_pt 和 ngx_http_output_body_filter_pt 是函数指针，定义如下：

```
typedef ngx_int_t (*ngx_http_output_header_filter_pt) (ngx_http_request_t *r);/* 发送响应头部 */
typedef ngx_int_t (*ngx_http_output_body_filter_pt) (ngx_http_request_t *r, ngx_chain_t *chain);/* 发送响应包体 */
```

参数 r 是当前的请求，chain 是待发送的HTTP 响应包体；上面提到的只有过滤链表的表头，那么使用什么把所有过滤模块连接起来呢？该工作由下面定义完成，即：

```
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
```

这样就可以遍历整个过滤链表，把 HTTP 响应报文发送出去，按照过滤链表的顺序，调用链表元素的回调方法可能会对响应报文数据进行检测、截取、新增、修改 或 删除等操作，即FIlter 模块可以对响应报文进行修改。但是必须注意的是只有最后一个链表元素才会真正的发送响应报文。

发送 HTTP 响应头部

HTTP 响应状态行、响应头部由函数 ngx_http_send_header 发送，该发送函数的执行过程中会遍历过滤链表，该过滤链表的过滤模块是那些对 HTTP 响应头部感兴趣的过滤模块组成，ngx_http_send_header 函数按照过滤链表的顺序依次处理响应头部，直到最后一个链表元素处理响应头部并把该响应头部发送给客户端。ngx_http_send_header 函数定义在文件src/http/ngx_http_core_module.c 中如下：

```
/* 发送 HTTP 响应头部 */
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
    if (r->header_sent) {
        ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
                      "header already sent");
        return NGX_ERROR;
    }

    if (r->err_status) {
        r->headers_out.status = r->err_status;
        r->headers_out.status_line.len = 0;
    }

    return ngx_http_top_header_filter(r);
}
```

从函数的实现过程中我们可以知道，该函数调用 ngx_http_top_header_filter 方法开始顺序遍历过滤链表的每一个元素处理方法，直到最后一个把响应头部发送出去为止。在Nginx中，过滤链表的顺序如下：

根据发送响应头部的过滤链表顺序可以知道，除了最后一个模块是真正发送响应头部给客户端之外，其他模块都只是对响应头部进行修改，最后一个过来模块是ngx_http_header_filter_module，该模块提供的处理方法是ngx_http_header_filter 根据请求结构体ngx_http_request_t 中的 header_out 成员序列化字符流，并发送序列化之后的响应头部；

ngx_http_header_filter_module 模块的定义如下：

```
ngx_module_t  ngx_http_header_filter_module = {
    NGX_MODULE_V1,
    &ngx_http_header_filter_module_ctx,    /* module context */
    NULL,                                  /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

从该模块的定义中可以知道，该模块只调用上下文结构 ngx_http_header_filter_module_ctx，该上下文结构定义如下：

```
static ngx_http_module_t  ngx_http_header_filter_module_ctx = {
    NULL,                                  /* preconfiguration */
    ngx_http_header_filter_init,           /* postconfiguration */

    NULL,                                  /* create main configuration */
    NULL,                                  /* init main configuration */

    NULL,                                  /* create server configuration */
    NULL,                                  /* merge server configuration */

    NULL,                                  /* create location configuration */
    NULL,                                  /* merge location configuration */
};
```

上下文结构指定了 postconfiguration 的方法为 ngx_http_header_filter_init，该方法定义如下：

```
/* 初始化ngx_http_header_filter_module模块 */
static ngx_int_t
ngx_http_header_filter_init(ngx_conf_t *cf)
{
    /* 调用ngx_http_header_filter方法发送响应头部 */
    ngx_http_top_header_filter = ngx_http_header_filter;

    return NGX_OK;
}
```

最终该模块由方法 ngx_http_header_filter 执行即发送HTTP 响应头部，下面对该方法进行分析；

ngx_http_header_filter 函数执行流程：

- 首先检查当前请求 ngx_http_request_t 结构的header_sent 标志位，若该标志位为1，则表示已经发

送过响应头部，因此，无需重复发送，直接返回NGX_OK 结束该函数；

- 若之前未发送过响应头部（即 headr_sent 标志位为0），此时，准备发送响应头部，并设置 header_sent 标志位为1（防止重复发送），表示正要发送响应头部，同时检查当前请求是否为原始请求，若不是原始请求（即为子请求），则不需要发送响应头部返回 NGX_OK，因为子请求不存在响应头部概念。继而检查HTTP 协议版本，若HTTP 协议版本小于 1.0（即不支持请求头部，也就没有所谓的响应头部）直接返回NGX_OK，若是原始请求且HTTP 协议版本不小于1.0版本，则准备发送响应头部；
- 根据 HTTP 响应报文的状态行、响应头部将字符串序列化为发送响应头部所需的字节数len，方便下面分配缓冲区空间存在待发送的响应头部；
- 根据前一步骤计算的 len 值在当前请求内存池中分配用于存储响应头部的字符流缓冲区b，并将响应报文的状态行、响应头部按照HTTP 规范序列化地复制到刚分配的缓冲区b 中；
- 将待发送响应头部的缓冲区 b 挂载到链表缓冲区 out.buf 中；挂载的目的是：当响应头部不能一次性发送完毕时，ngx_http_header_filter 方法会返回NGX_AGAIN，表示发送的响应头部不完整，则把剩余的响应头部数据保存在out 链表缓冲区中，以便调用ngx_http_filter_request 时，再次调用 HTTP 框架将 out 链表缓冲区的剩余响应头部字符流发送出去；
- 调用 ngx_http_writer_filter 方法将out 链表缓冲区的响应头部发送出去，但是不能保证一次性发送完毕；

函数 ngx_http_header_filter 在文件src/http/ngx_http_header_filter_module.c 中定义如下：

```
/* 发送HTTP响应头部 */
static ngx_int_t
ngx_http_header_filter(ngx_http_request_t *r)
{
    u_char                    *p;
    size_t                    len;
    ngx_str_t                 host, *status_line;
    ngx_buf_t                 *b;
    ngx_uint_t                status, i, port;
    ngx_chain_t               out;
    ngx_list_part_t           *part;
    ngx_table_elt_t           *header;
    ngx_connection_t          *c;
    ngx_http_core_loc_conf_t  *clcf;
    ngx_http_core_srv_conf_t  *cscf;
    struct sockaddr_in        *sin;
#if (NGX_HAVE_INET6)
    struct sockaddr_in6       *sin6;
#endif
    u_char                    addr[NGX_SOCKADDR_STRLEN];

    /*
     * 检查当前请求结构的header_sent标志位，若该标志位为1，
     * 表示已经发送HTTP请求响应，则无需再发送，此时返回NGX_OK；
     */
    if (r->header_sent) {
        return NGX_OK;
    }
```

```
    }

    /* 若之前未发送HTTP请求响应，则现在准备发送，并设置header_sent标志位 */
    r->header_sent = 1;

    /* 当前请求不是原始请求，则返回NGX_OK */
    if (r != r->main) {
        return NGX_OK;
    }

    /*
     * 若HTTP版本为小于1.0 则直接返回NGX_OK；
     * 因为这些版本不支持请求头部，所有就没有响应头部；
     */
    if (r->http_version < NGX_HTTP_VERSION_10) {
        return NGX_OK;
    }

    if (r->method == NGX_HTTP_HEAD) {
        r->header_only = 1;
    }

    if (r->headers_out.last_modified_time != -1) {
        if (r->headers_out.status != NGX_HTTP_OK
            && r->headers_out.status != NGX_HTTP_PARTIAL_CONTENT
            && r->headers_out.status != NGX_HTTP_NOT_MODIFIED)
        {
            r->headers_out.last_modified_time = -1;
            r->headers_out.last_modified = NULL;
        }
    }

    /* 以下是根据HTTP响应报文的状态行、响应头部字符串序列化为所需的字节数len */
    len = sizeof("HTTP/1.x ") - 1 + sizeof(CRLF) - 1
          /* the end of the header */
          + sizeof(CRLF) - 1;

    /* status line */

    if (r->headers_out.status_line.len) {
        len += r->headers_out.status_line.len;
        status_line = &r->headers_out.status_line;

     ...
     ...

    /* 分配用于存储响应头部字符流缓冲区 */
    b = ngx_create_temp_buf(r->pool, len);
    if (b == NULL) {
        return NGX_ERROR;
    }

    /* 将响应报文的状态行、响应头部按照HTTP规范序列化地复制到刚分配的缓冲区b中 */
    /* "HTTP/1.x " */
    b->last = ngx_cpymem(b->last, "HTTP/1.1 ", sizeof("HTTP/1.x ") - 1);
```

```
    /* status line */
    if (status_line) {
        b->last = ngx_copy(b->last, status_line->data, status_line->len);

    } else {
        b->last = ngx_sprintf(b->last, "%03ui ", status);
    }

    ...
    ...

    /*
     * 将待发送的响应头部挂载到out链表缓冲区中，
     * 挂载的目的是：当响应头部不能一次性发送完成时，
     * ngx_http_header_filter方法返回NGX_AGAIN，表示发送的响应头部不完整，
     * 则把剩余的响应头部保存在out链表中，以便调用ngx_http_finalize_request时，
     * 再次调用HTTP框架将out链表中剩余的响应头部字符流继续发送；
     */
    out.buf = b;
    out.next = NULL;

    /*
     * 调用方法ngx_http_write_filter将响应头部字符流发送出去；
     * 所有实际发送响应头部数据的由ngx_http_write_filter方法实现；
     */
    return ngx_http_write_filter(r, &out);
}
```

发送 HTTP 响应包体

HTTP 响应包体由函数 ngx_http_output_filter 发送，该发送函数的执行过程中会遍历过滤链表，该过滤链表的过滤模块是那些对 HTTP 响应包体感兴趣的过滤模块组成，ngx_http_output_filter 函数按照过滤链表的顺序依次处理响应包体，直到最后一个链表元素处理响应包体并把该响应包体发送给客户端。ngx_http_output_filter 函数定义在文件 src/http/ngx_http_core_module.c 中如下：

```
/* 发送HTTP 响应包体 */
ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t          rc;
    ngx_connection_t  *c;

    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http output filter \"%V?%V\"", &r->uri, &r->args);

    rc = ngx_http_top_body_filter(r, in);

    if (rc == NGX_ERROR) {
        /* NGX_ERROR may be returned by any filter */
        c->error = 1;
    }

    return rc;
}
```

从函数的实现过程中我们可以知道，该函数调用 ngx_http_top_body_filter 方法开始顺序遍历过滤链表的每一个元素处理方法，直到最后一个把响应包体发送出去为止。在Nginx 中，过滤链表的顺序如下：

```
+-------------------------+
|ngx_http_range_body_filter|
+----------+--------------+
           |
           v
+----------+---------+
|ngx_http_copy_filter|
+----------+---------+
           |
           v
+----------+-----------------+
|ngx_http_charset_body_filter|
+----------+-----------------+
           |
           v
+----------+-------------+
|ngx_http_ssi_body_filter|
+----------+-------------+
           |
           v
+----------+-------------+
|ngx_http_postpone_filter|
+----------+-------------+
           |
           v
+----------+--------------+
|ngx_http_gzip_body_filter|
+----------+--------------+
           |
           v
+----------+-----------------+
|ngx_http_chunked_body_filter|
+----------+-----------------+
           |
           v
+--------------------+
|ngx_http_write_filter|
+--------------------+
```

根据发送响应包体的过滤链表顺序可以知道，除了最后一个模块是真正发送响应包体给客户端之外，其他模块都只是对响应包体进行修改，最后一个过来模块是ngx_http_write_filter_module，该模块提供的处理方法是ngx_http_write_filter；

ngx_http_write_filter_module 模块的定义如下：

```
ngx_module_t  ngx_http_write_filter_module = {
    NGX_MODULE_V1,
    &ngx_http_write_filter_module_ctx,     /* module context */
    NULL,                                  /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

该模块的上下文结构 ngx_http_write_filter_module_ctx 定义如下：

```
static ngx_http_module_t  ngx_http_write_filter_module_ctx = {
    NULL,                                  /* preconfiguration */
    ngx_http_write_filter_init,            /* postconfiguration */

    NULL,                                  /* create main configuration */
    NULL,                                  /* init main configuration */

    NULL,                                  /* create server configuration */
    NULL,                                  /* merge server configuration */

    NULL,                                  /* create location configuration */
    NULL,                                  /* merge location configuration */
};
```

上下文结构中只调用 ngx_http_write_filter_init 方法，该方法定义如下：

```
/* 初始化模块 */
static ngx_int_t
ngx_http_write_filter_init(ngx_conf_t *cf)
{
    /* 调用模块的回调方法 */
    ngx_http_top_body_filter = ngx_http_write_filter;

    return NGX_OK;
}
```

该模块最终调用 ngx_http_write_filter 方法发送HTTP 响应包体，该方法的实现如下分析；

ngx_http_writer_filter 函数执行流程：

- 检查当前连接的错误标志位 error，若该标志位为 1，表示当前请求出粗，则返回 NGX_ERROR 结束该函数，否则继续；

- 遍历当前请求 ngx_http_request_t 结构体中的链表缓冲区成员out，计算剩余响应报文的长度size。因为当响应报文一次性不能发送完毕时，会把剩余的响应报文保存在out 中，相对于本次待发送的响应报文 in (即是该函数所传入的参数in)来说，out 链表缓冲区保存的是前一次剩余的响应报文；
- 将本次待发送的响应报文的缓冲区 in 添加到 out 链表缓冲区的尾部，并计算待发送响应报文的总长度 size；
- 若缓冲区 ngx_buf_t 块的 last_buf (即 last)、flush 标志位为0，则表示待发送的out 链表缓冲区没有一个是需要立刻发送响应报文，并且本次待发送的in 不为空，且待发送的响应报文数据总长度 size 小于postpone_output 参数（该参数由nginx.conf配置文件中设置），则不需要发送响应报文，即返回NGX_OK 结束该函数；
- 若需要发送响应报文，则检查当前连接上写事件的 delayed 标志位，若为1，表示发送响应超速，则需要在epoll 事件机制中减速，所有相当于延迟发送响应报文，则返回NGX_AGIAN；
- 若不需要延迟发送响应报文，检查当前请求的限速标志位 limit_rate，若该标志位设置为大于0，表示当前发送响应报文的速度不能超过limit_rate 值；
- 根据限速值 r->limit_rate、当前客户开始接收响应的时间r->start_sec、在当前连接上已发送响应的长度c->sent、和limit_after 值计算本次可以发送的字节数limit，若limit 值不大于0，表示当前连接上发送响应的速度超过limit_rate 限速值，即本次不可以发送响应，因此将写事件的delayed 标志位设置为1，把写事件添加到定时器机制，并设置当前连接ngx_connection_t 结构体中的成员buffered 为NGX_HTTP_WRITE_BUFFERED（即可写状态），同时返回NGX_AGAIN，表示链表缓冲区out 还保存着剩余待发送的响应报文；
- 若 limit 值大于 0，则根据 limit 值、配置项参数 sendfile_max_chunk 和待发送字节数 size 来计算本次发送响应的长度(即三者中的最小值)；
- 根据前一步骤计算的可发送响应的长度，再次检查 limit_rate 标志位，若limit_rate 还是为1，表示继续需要限速检查。再按照前面的计算方法判断是否超过限速值limit_rate，若超过该限速值，则需再次把写事件添加到定时器机制中，标志位delayed 设置为1；
- 若不会超过限速值，则发送响应，并重新调整链表缓冲区 out 的情况，把已发送响应数据的缓冲区进行回收内存；
- 继续检查链表缓冲区 out 是否还存在数据，若存在数据，则表示未发送完毕，返回NGX_AGAIN，表示等待下次HTTP 框架被调用发送out 缓冲区剩余的响应数据；若不存在数据，则表示成功发送完整的响应数据，并返回NGX_OK；

函数 ngx_http_write_filter 在文件 src/http/ngx_http_write_filter_module.c 中定义如下：

```
/* 发送响应报文数据 */

/* 参数r是对应的请求，in是保存本次待发送数据的链表缓冲区 */
ngx_int_t
ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    off_t                      size, sent, nsent, limit;
    ngx_uint_t                 last, flush;
    ngx_msec_t                 delay;
    ngx_chain_t               *cl, *ln, **ll, *chain;
```

```
    ngx_connection_t          *c;
    ngx_http_core_loc_conf_t  *clcf;

    /* 获取当前请求所对应的连接 */
    c = r->connection;

    /*
     * 检查当前连接的错误标志位error，若该标志位为1，
     * 表示当前请求出错，返回NGX_ERROR；
     */
    if (c->error) {
        return NGX_ERROR;
    }

    size = 0;
    flush = 0;
    last = 0;
    ll = &r->out;

    /* find the size, the flush point and the last link of the saved chain */

    /*
     * 遍历当前请求out链表缓冲区，计算剩余响应报文的长度；
     * 因为当响应报文一次性不能发送完成时，会把剩余的响应报文保存在out中，
     * 相对于本次发送的响应报文数据in来说（即该方法所传入的参数in），
     * out链表缓冲区保存的是前一次剩余的响应报文；
     */
    for (cl = r->out; cl; cl = cl->next) {
        ll = &cl->next;

        ngx_log_debug7(NGX_LOG_DEBUG_EVENT, c->log, 0,
                       "write old buf t:%d f:%d %p, pos %p, size: %z "
                       "file: %O, size: %z",
                       cl->buf->temporary, cl->buf->in_file,
                       cl->buf->start, cl->buf->pos,
                       cl->buf->last - cl->buf->pos,
                       cl->buf->file_pos,
                       cl->buf->file_last - cl->buf->file_pos);

#if 1
        ...
#endif

        size += ngx_buf_size(cl->buf);

        if (cl->buf->flush || cl->buf->recycled) {
            flush = 1;
        }

        if (cl->buf->last_buf) {
            last = 1;
        }
    }

    /* add the new chain to the existent one */
```

```
    /*
     * 将本次待发送的响应报文的缓冲区in添加到out链表缓冲区的尾部，
     * 并计算待发送响应报文总的长度size；
     */
    for (ln = in; ln; ln = ln->next) {
        cl = ngx_alloc_chain_link(r->pool);
        if (cl == NULL) {
            return NGX_ERROR;
        }

        cl->buf = ln->buf;
        *ll = cl;/* 由上面可知 ll=&r->out */
        ll = &cl->next;

        ngx_log_debug7(NGX_LOG_DEBUG_EVENT, c->log, 0,
                       "write new buf t:%d f:%d %p, pos %p, size: %z "
                       "file: %O, size: %z",
                       cl->buf->temporary, cl->buf->in_file,
                       cl->buf->start, cl->buf->pos,
                       cl->buf->last - cl->buf->pos,
                       cl->buf->file_pos,
                       cl->buf->file_last - cl->buf->file_pos);

#if 1
        ...
#endif

        size += ngx_buf_size(cl->buf);

        if (cl->buf->flush || cl->buf->recycled) {
            flush = 1;
        }

        if (cl->buf->last_buf) {
            last = 1;
        }
    }

    *ll = NULL;

    ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http write filter: l:%d f:%d s:%O", last, flush, size);

    /* 获取ngx_http_core_module模块的loc级别配置项结构体 */
    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    /*
     * avoid the output if there are no last buf, no flush point,
     * there are the incoming bufs and the size of all bufs
     * is smaller than "postpone_output" directive
     */

    /*
     * 若out链表最后一块缓冲区last为空，且没有强制性刷新flush链表缓冲区out
```

```
    * 若out链表最后一块缓冲区last为空，且没有强制性刷新flush链表缓冲区out，
    * 且当前有待发响应报文in，但是待发送响应报文总的长度size小于预设可发送条件值postpone_output,
    * 则本次不能发送响应报文，继续保存在out链表缓冲区中，以待下次才发送；
    * 其中postpone_output预设值我们可以在配置文件nginx.conf中设置；
    */
    if (!last && !flush && in && size < (off_t) clcf->postpone_output) {
        return NGX_OK;
    }

    /*
    * 检查当前连接上写事件的delayed标志位，
    * 若该标志位为1，表示需要延迟发送响应报文，
    * 因此，返回NGX_AGAIN，表示延迟发送；
    */
    if (c->write->delayed) {
        c->buffered |= NGX_HTTP_WRITE_BUFFERED;
        return NGX_AGAIN;
    }

    if (size == 0
        && !(c->buffered & NGX_LOWLEVEL_BUFFERED)
        && !(last && c->need_last_buf))
    {
        if (last || flush) {
            for (cl = r->out; cl; /* void */) {
                ln = cl;
                cl = cl->next;
                ngx_free_chain(r->pool, ln);
            }

            r->out = NULL;
            c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

            return NGX_OK;
        }

        ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                      "the http output chain is empty");

        ngx_debug_point();

        return NGX_ERROR;
    }

    /*
    * 检查当前请求的限速标志位limit_rate，
    * 若该标志位为大于0，表示发送响应报文的速度不能超过limit_rate指定的速度；
    */
    if (r->limit_rate) {
        if (r->limit_rate_after == 0) {
            r->limit_rate_after = clcf->limit_rate_after;
        }

        /* 计算发送速度是否超过限速值 */
        limit = (off_t) r->limit_rate * (ngx_time() - r->start_sec + 1)
```

```
                - (c->sent - r->limit_rate_after);

        /*
         * 若当前发送响应报文的速度超过限速值，则写事件标志位delayed设为1，
         * 并把该写事件添加到定时器机制中，并且将buffered设置为可写状态，
         * 返回NGX_AGAIN，表示链表缓冲区out还保存剩余待发送的响应报文；
         */
        if (limit <= 0) {
            c->write->delayed = 1;
            ngx_add_timer(c->write,
                          (ngx_msec_t) (- limit * 1000 / r->limit_rate + 1));

            c->buffered |= NGX_HTTP_WRITE_BUFFERED;

            return NGX_AGAIN;
        }

        if (clcf->sendfile_max_chunk
            && (off_t) clcf->sendfile_max_chunk < limit)
        {
            limit = clcf->sendfile_max_chunk;
        }

    } else {
        limit = clcf->sendfile_max_chunk;
    }

    /* 若不需要减速，或没有设置速度限制，则向客户端发送响应字符流 */
    sent = c->sent;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http write filter limit %O", limit);

    chain = c->send_chain(c, r->out, limit);

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http write filter %p", chain);

    if (chain == NGX_CHAIN_ERROR) {
        c->error = 1;
        return NGX_ERROR;
    }

    /* 再次检查limit_rate标志位 */
    if (r->limit_rate) {

        nsent = c->sent;

        if (r->limit_rate_after) {

            sent -= r->limit_rate_after;
            if (sent < 0) {
                sent = 0;
            }
```

```
            nsent -= r->limit_rate_after;
            if (nsent < 0) {
                nsent = 0;
            }
        }

        /* 再次计算当前发送响应报文速度是否超过限制值 */
        delay = (ngx_msec_t) ((nsent - sent) * 1000 / r->limit_rate);

        /* 若超过，需要限速，并把写事件添加到定时器机制中 */
        if (delay > 0) {
            limit = 0;
            c->write->delayed = 1;
            ngx_add_timer(c->write, delay);
        }
    }

    if (limit
        && c->write->ready
        && c->sent - sent >= limit - (off_t) (2 * ngx_pagesize))
    {
        c->write->delayed = 1;
        ngx_add_timer(c->write, 1);
    }

    /* 重新调整链表缓冲区out的情况，把已发送数据的缓冲区内存回收 */
    for (cl = r->out; cl && cl != chain; /* void */) {
        ln = cl;
        cl = cl->next;
        ngx_free_chain(r->pool, ln);
    }

    /* 检查out链表缓冲区是否还有数据 */
    r->out = chain;

    /* 若还有数据，返回NGX_AGAIN，表示还存在待发送的响应报文数据 */
    if (chain) {
        c->buffered |= NGX_HTTP_WRITE_BUFFERED;
        return NGX_AGAIN;
    }

    c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

    if ((c->buffered & NGX_LOWLEVEL_BUFFERED) && r->postponed == NULL) {
        return NGX_AGAIN;
    }

    /* 若已发送全部数据则返回NGX_OK */
    return NGX_OK;
}
```

ngx_http_write 函数执行流程如下：

- 检查写事件的 timedout 标志位，若该标志位为 1（表示超时），进而判断属于哪种情况引起的超时（第一种：网络异常或客户端长时间不接收响应；第二种：由于响应发送速度超速，导致写事件被添加到定时器机制（注意一点：delayed 标志位此时是为1），有超速引起的超时，不算真正的响应发送超时）；
- 检查 delayed 标志位，若 delayed 为 0，表示由第一种情况引起的超时，即是真正的响应超时，此时设置timedout 标志位为1，并调用函数ngx_http_finalize_request 结束请求；
- 若 delayed 为 1，表示由第二种情况引起的超时，不算真正的响应超时，此时，把标志位 timedout、delayed 都设置为 0，继续检查写事件的 ready 标志位，若 ready 为 0，表示当前写事件未准备就绪（即不可写），因此，将写事件添加到定时器机制，注册到epoll 事件机制中，等待可写事件发送，返回return 结束该方法；
- 若写事件 timedout 为 0，且 delayed 为 0，且 ready 为 1，则调用函数 ngx_http_output_filter 发送响应；该函数的第二个参数为NULL，表示需要调用各个包体过滤模块处理链表缓冲区out 中剩余的响应，最后由ngx_http_write_filter 方法把响应发送出去；

函数 ngx_http_writer 在文件 src/http/ngx_http_request.c 中定义如下：

```
static void
ngx_http_writer(ngx_http_request_t *r)
{
    int                       rc;
    ngx_event_t              *wev;
    ngx_connection_t         *c;
    ngx_http_core_loc_conf_t  *clcf;

    /* 获取当前请求的连接 */
    c = r->connection;
    /* 获取连接上的写事件 */
    wev = c->write;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, wev->log, 0,
                   "http writer handler: \"%V?%V\"", &r->uri, &r->args);

    /* 获取ngx_http_core_module模块的loc级别配置项结构 */
    clcf = ngx_http_get_module_loc_conf(r->main, ngx_http_core_module);

    /*
     * 写事件超时有两种可能：
     * 1、由于网络异常或客户端长时间不接收响应，导致真实的发送响应超时；
     * 2、由于响应发送速度超过了请求的限速值limit_rate，导致写事件被添加到定时器机制中，
     *    这是由超速引起的，并不是真正的响应发送超时；注意：写事件被添加到定时器机制时，delayed标志位设置为1；
     */

    /* 检查写事件是否超时，若超时(即timedout为1)，进而判断属于哪种情况引起的超时 */
    if (wev->timedout) {
        /*
         * 若是响应真的超时，即网络异常或客户端长时间未接收响应引起的超时；
         * 则将timedout标志位设置为1，并调用ngx_http_finalize_request结束请求；
         * 并return返回结束当前方法；
```

```
         */
        if (!wev->delayed) {
            ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
                          "client timed out");
            c->timedout = 1;

            ngx_http_finalize_request(r, NGX_HTTP_REQUEST_TIME_OUT);
            return;
        }

        /*
         * 若是由超速发送响应引起的超时，则将timedout、delayed标志位都设为0；
         * 再继续检查写事件的ready标志位；
         */
        wev->timedout = 0;
        wev->delayed = 0;

        /*
         * 检查写事件的ready标志位，若写事件未准备就绪(ready=0)，即表示当前写事件不可写，
         * 则将写事件添加到定时器机制中，同时将写事件注册到epoll事件机制中，等待可写事件发生；
         * 并return结束当前方法；
         */
        if (!wev->ready) {
            ngx_add_timer(wev, clcf->send_timeout);

            if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
                ngx_http_close_request(r, 0);
            }

            return;
        }

    }

    /* 当timedout为0，但是delayed为1或是aio，则将写事件注册到epoll事件机制中，并return返回 */
    if (wev->delayed || r->aio) {
        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, wev->log, 0,
                       "http writer delayed");

        if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
            ngx_http_close_request(r, 0);
        }

        return;
    }

    /* 若写事件timedout为0，且delayed为0，且ready为1，则调用ngx_http_output_filter 发送响应报文 */
    rc = ngx_http_output_filter(r, NULL);

    ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http writer output filter: %d, \"%V?%V\"",
                   rc, &r->uri, &r->args);

    /* 若发送响应错误，则调用ngx_http_finalize_request结束请求，并return返回 */
```

```
    if (rc == NGX_ERROR) {
        ngx_http_finalize_request(r, rc);
        return;
    }

    /*
     * 若成功发送响应，则检查当前请求的out链表缓冲区是否存在剩余待发送的响应报文，
     * 若存在剩余待发送响应，又因为此时写事件不可写，则将其添加到定时器机制，注册到epoll事件机制中，
     * 等待可写事件的发生生；*/
    if (r->buffered || r->postponed || (r == r->main && c->buffered)) {

        if (!wev->delayed) {
            ngx_add_timer(wev, clcf->send_timeout);
        }

        if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
            ngx_http_close_request(r, 0);
        }

        return;
    }

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, wev->log, 0,
                   "http writer done: \"%V?%V\"", &r->uri, &r->args);

    /*
     * 若当前out链表缓冲区不存在未发送的响应数据，则表示已成功发送完整的响应数据，
     * 此时，重新设置写事件的回调方法为ngx_http_request_empty_handler即不进行任何操作；
     */
    r->write_event_handler = ngx_http_request_empty_handler;

    /* 最终调用ngx_http_finalize_request结束请求 */
    ngx_http_finalize_request(r, rc);
}
```

总结：真正发送响应的是 ngx_http_write_filter 函数，但是该函数不能保证一次性把响应发送完毕，若发送不完毕，把剩余的响应保存在out 链表缓冲区中，继而调用ngx_http_writer 把剩余的响应发送出去，函数ngx_http_writer 最终调用的是ngx_http_output_filter 函数发送响应，但是要知道的是 ngx_http_output_filter 函数是需要调用个包体过滤模块来处理剩余响应的out 链表缓冲区，并由最后一个过滤模块 ngx_http_write_filter_module 调用ngx_http_write_filter 方法将响应发送出去；因此，我们可知道，真正发送响应的函数是ngx_http_write_filter；

关闭连接请求

当一个动作结束时，会根据引用计数判断是否结束其处理的请求，以下是有关关闭请求的函数；以下函数均在在文件 src/http/ngx_http_request.c 中定义如下；

ngx_http_finalize_request 函数执行流程：

- 若所传入的参数 rc = NGX_DONE，则直接调用ngx_http_finalize_connection 函数结束连接，并

return 退出当前函数；

- 若参数 rc = NGX_DECLINED，表示需要按照11 个HTTP 阶段继续处理，此时，设置r->content_handler = NULL（为了让ngx_http_core_content_phase 方法可以继续调用NGX_HTTP_CONTENT_PHASE 阶段的其他处理方法），并设置写事件的回调方法为ngx_http_core_run_phases，最后调用ngx_http_core_run_phases 方法处理请求，return 从当前函数返回；
- 若 rc != NGX_DONE 且 rc != NGX_DECLINED，检查当前请求是否为子请求：
- 若当前请求是子请求，则调用 post_subrequest 的回调方法handler；
- 若不是子请求则继续执行以下程序；
- 若 rc 为 NGX_ERROR、NGX_HTTP_REQUEST_TIME_OUT、NGX_HTTP_CLIENT_CLOSED_REQUEST，或当前连接的错误码标志位c->error为1，则调用ngx_http_terminate_request 强制关闭请求，return 从当前函数返回；
- 若 rc 为 NGX_HTTP_CREATED、NGX_HTTP_NO_CONTENT，或rc 不小于NGX_HTTP_SPECIAL_RESPONSE，接着检查当前请求是否为原始请求，若是原始请求，则检查读、写事件的timer_set 标志位，若 timer_set 为1，将读、写事件从定时器机制中移除，重新设置当前连接的读、写事件的回调方法都为ngx_http_request_handler，并调用ngx_http_finalize_request，此时应该注意的是ngx_http_finalize_request 函数的第二个参数是ngx_http_special_response_handler(r, rc)函数的返回值，ngx_http_special_response_handler(r, rc) 函数根据参数rc 构造完整的HTTP 响应，根据ngx_http_special_response_handler 函数的返回值调用ngx_http_finalize_request 方法结束请求。return 从当前函数返回；
- 若参数 rc 不是以上步骤所描述的值，检查当前请求是否为原始请求：
- 若当前请求不是原始请求，
- 若当前请求是原始请求，检查当前请求的 buffered、postponed、blocked 标志位 或当前连接的 buffered 标志位：
- 若这些标志位有一个是 1，则调用 ngx_http_set_write_handler 函数（该函数的功能就是设置当前请求写事件的回调方法为 ngx_http_writer 发送 out 链表缓冲区的剩余响应，若写事件未准备就绪，则将写事件添加到定时器机制，注册到epoll 事件机制中，最终返回NGX_OK），并return 返回当前函数；
- 若这些标志位都不为 1，则检查读、写事件的 timer_set 标志位，若 timer_set 标志位为1，则将读、写事件从定时器机制中移除，最后调用ngx_http_finalize_connection 释放请求，并关闭连接；

```
/* 结束请求 */
void
ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
{
```

```
    ngx_connection_t          *c;
    ngx_http_request_t        *pr;
    ngx_http_core_loc_conf_t  *clcf;

    c = r->connection;

    ngx_log_debug5(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http finalize request: %d, \"%V?%V\" a:%d, c:%d",
                   rc, &r->uri, &r->args, r == c->data, r->main->count);

    /* 若传入的参数rc=NGX_DONE，则直接调用ngx_http_finalize_connection方法 */
    if (rc == NGX_DONE) {
        ngx_http_finalize_connection(r);
        return;
    }

    if (rc == NGX_OK && r->filter_finalize) {
        c->error = 1;
    }

    /*
     * 若传入的参数rc=NGX_DECLINED，则表示需按照11个HTTP阶段继续处理；
     * 此时，写事件调用ngx_http_core_run_phases；
     */
    if (rc == NGX_DECLINED) {
        r->content_handler = NULL;
        r->write_event_handler = ngx_http_core_run_phases;
        ngx_http_core_run_phases(r);
        return;
    }

    /* 若传入的参数rc != NGX_DONE 且 rc != NGX_DECLINED，则执行以下程序 */

    /*
     * 若当前处理的请求是子请求，且post_subrequest标志位为1，
     * 则调用post_subrequest的handler回调方法；
     */
    if (r != r->main && r->post_subrequest) {
        rc = r->post_subrequest->handler(r, r->post_subrequest->data, rc);
    }

    /* 若处理的当前请求不是子请求，则执行以下程序 */

    /* 若rc是以下这些值，或error标志位为1，则调用ngx_http_terminate_request方法强制关闭请求 */
    if (rc == NGX_ERROR
        || rc == NGX_HTTP_REQUEST_TIME_OUT
        || rc == NGX_HTTP_CLIENT_CLOSED_REQUEST
        || c->error)
    {
        if (ngx_http_post_action(r) == NGX_OK) {
            return;
        }

        if (r->main->blocked) {
```

```
            r->write_event_handler = ngx_http_request_finalizer;
        }

        ngx_http_terminate_request(r, rc);
        return;
    }

    /*
     * 若rc为以下值，表示请求的动作是上传文件，
     * 或HTTP模块需要HTTP框架构造并发送响应码不小于300的特殊响应；
     * 则首先检查当前请求是否为原始请求，若不是则调用ngx_http_terminate_request强制关闭请求，
     * 若是原始请求，则将读、写事件从定时器机制中移除；
     * 并重新设置读、写事件的回调方法为ngx_http_request_handler,
     * 最后调用ngx_http_finalize_request关闭请求（指定特定的rc参数）；
     */
    if (rc >= NGX_HTTP_SPECIAL_RESPONSE
        || rc == NGX_HTTP_CREATED
        || rc == NGX_HTTP_NO_CONTENT)
    {
        if (rc == NGX_HTTP_CLOSE) {
            ngx_http_terminate_request(r, rc);
            return;
        }

        if (r == r->main) {
            if (c->read->timer_set) {
                ngx_del_timer(c->read);
            }

            if (c->write->timer_set) {
                ngx_del_timer(c->write);
            }
        }

        c->read->handler = ngx_http_request_handler;
        c->write->handler = ngx_http_request_handler;

        ngx_http_finalize_request(r, ngx_http_special_response_handler(r, rc));
        return;
    }

    /* 若rc不是以上的值，则执行以下程序 */

    /* 再次检查当前请求是否为原始请求 */
    if (r != r->main) {

        /*
         * 若当前请求不是原始请求，即当前请求是子请求；
         * 若子请求的buffered 或 postponed 标志位为1，
         * 则调用 ngx_http_set_write_handler;
         */
        if (r->buffered || r->postponed) {

            if (ngx_http_set_write_handler(r) != NGX_OK) {
                ngx_http_terminate_request(r, 0);
```

```
        }

        return;
    }

    /*
     * 若子请求的buffered且postponed标志位都为0，则找到当前子请求的父亲请求；
     */
    pr = r->parent;

    /*
     * 将父亲请求放置在ngx_http_posted_request_t结构体中，
     * 并将该结构体添加到原始请求的posted_requests链表中；
     */
    if (r->buffered || r->postponed) {

        if (ngx_http_set_write_handler(r) != NGX_OK) {
            ngx_http_terminate_request(r, 0);
        }
    if (r == c->data) {

        r->main->count--;
        r->main->subrequests++;

        if (!r->logged) {

            clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

            if (clcf->log_subrequest) {
                ngx_http_log_request(r);
            }

            r->logged = 1;

        } else {
            ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                          "subrequest: \"%V?%V\" logged again",
                          &r->uri, &r->args);
        }

        r->done = 1;

        if (pr->postponed && pr->postponed->request == r) {
            pr->postponed = pr->postponed->next;
        }

        c->data = pr;

    } else {

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http finalize non-active request: \"%V?%V\"",
                       &r->uri, &r->args);

        r->write_event_handler = ngx_http_request_finalizer;
```

```
            r->write_event_handler = ngx_http_request_finalizer;

            if (r->waited) {
                r->done = 1;
            }
        }

        if (ngx_http_post_request(pr, NULL) != NGX_OK) {
            r->main->count++;
            ngx_http_terminate_request(r, 0);
            return;
        }

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http wake parent request: \"%V?%V\"",
                       &pr->uri, &pr->args);

        return;
    }

    /* 若当前请求是原始请求 */

    /*
     * 若r->buffered或c->buffered 或 r->postponed 或 r->blocked 标志位为1；
     * 则调用ngx_http_set_write_handler方法；
     */
    if (r->buffered || c->buffered || r->postponed || r->blocked) {

        if (ngx_http_set_write_handler(r) != NGX_OK) {
            ngx_http_terminate_request(r, 0);
        }

        return;
    }

    if (r != c->data) {
        ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                      "http finalize non-active request: \"%V?%V\"",
                      &r->uri, &r->args);
        return;
    }

    r->done = 1;
    r->write_event_handler = ngx_http_request_empty_handler;

    if (!r->post_action) {
        r->request_complete = 1;
    }

    if (ngx_http_post_action(r) == NGX_OK) {
        return;
    }

    /*
     * 将读、写事件从定时器机制中移除；
```

```
     */
    if (c->read->timer_set) {
        ngx_del_timer(c->read);
    }

    if (c->write->timer_set) {
        c->write->delayed = 0;
        ngx_del_timer(c->write);
    }

    if (c->read->eof) {
        ngx_http_close_request(r, 0);
        return;
    }

    /* 关闭连接，并结束请求 */
    ngx_http_finalize_connection(r);
}
```

ngx_http_set_write_handler 函数执行流程：

- 设置当前请求的读事件回调方法为：ngx_httpp_discarded_request_body_handler（丢弃包体）或 ngx_http_test_reading；
- 设置当前请求的写事件回调方法为 ngx_http_writer（发送out 链表缓冲区剩余的响应）；
- 若当前写事件准备就绪（即 ready 和 delayed 标志位为 1）开始限速的发送 out 链表缓冲区中的剩余响应；
- 若当前写事件未准备就绪，则将写事件添加到定时器机制，注册到 epoll 事件机制中；

```
static ngx_int_t
ngx_http_set_write_handler(ngx_http_request_t *r)
{
    ngx_event_t               *wev;
    ngx_http_core_loc_conf_t  *clcf;

    r->http_state = NGX_HTTP_WRITING_REQUEST_STATE;

    /* 设置当前请求读事件的回调方法：丢弃包体或不进行任何操作 */
    r->read_event_handler = r->discard_body ?
                            ngx_http_discarded_request_body_handler:
                            ngx_http_test_reading;
    /* 设置写事件的回调方法为ngx_http_writer，即发送out链表缓冲区剩余的响应 */
    r->write_event_handler = ngx_http_writer;

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream) {
        return NGX_OK;
    }
#endif

    wev = r->connection->write;

    /* 若写事件的ready标志位和delayed标志为都为1，则返回NGX_OK */
    if (wev->ready && wev->delayed) {
        return NGX_OK;
    }

    /*
     * 若写事件的ready标志位为0，或delayed标志位为0，则将写事件添加到定时器机制中；
     * 同时将写事件注册到epoll事件机制中；
     * 最后返回NGX_OK；
     */
    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
    if (!wev->delayed) {
        ngx_add_timer(wev, clcf->send_timeout);
    }

    if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
        ngx_http_close_request(r, 0);
        return NGX_ERROR;
    }

    return NGX_OK;
}
```

ngx_http_finalize_connection 函数的执行流程：

- 检查原始请求的引用计数，若原始请求的引用计数不为 1，则表示其他动作在操作该请求，检查当前请求的discard_body 标志位：
- 若 discard_body 标志位为 1，表示当前请求正在丢弃包体，把读事件的回调方法设为

ngx_http_discarded_request_body_handler 方法，并将读事件添加到定时器机制中（超时时间为 lingering_timeout），最后调用ngx_http_close_request 关闭请求，并 return 从当前函数返回；

- 若 discard_body 标志位为 0，直接调用 ngx_http_close_request 关闭请求，并return 从当前函数返回；
- 若原始请求的引用计数为 1，检查当前请求的 keepalive 标志位：
- 若 keepalive 标志位为 1，则调用 ngx_http_set_keepalive方 法将当前连接设置为keepalive 状态，并return 从当前函数返回；
- 若 keepalive 标志位为 0，检查 lingering_close 标志位：
- 若 lingering_close 标志位为 1，则调用 ngx_http_set_lingering_close 延迟关闭请求，return 从当前函数返回；
- 若 lingering_close 标志位为 0，则调用 ngx_http_close_request 方法关闭请求；

```
/* 结束当前连接 */
static void
ngx_http_finalize_connection(ngx_http_request_t *r)
{
    ngx_http_core_loc_conf_t  *clcf;

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream) {
        ngx_http_close_request(r, 0);
        return;
    }
#endif

    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    /*
     * 检查原始请求的引用计数，若原始请求的引用计数不为1，表示有其他动作在操作该请求；
     */
    if (r->main->count != 1) {

        /*
         * 检查当前请求的discard_body标志位，若该标志位为1，表示当前请求正在丢弃包体；
         */
        if (r->discard_body) {
            /* 设置当前请求读事件的回调方法，并将读事件添加到定时器机制中 */
            r->read_event_handler = ngx_http_discarded_request_body_handler;
            ngx_add_timer(r->connection->read, clcf->lingering_timeout);

            if (r->lingering_time == 0) {
                r->lingering_time = ngx_time()
                                  + (time_t) (clcf->lingering_time / 1000);
            }
        }
```

```
        /* 关闭当前请求 */
        ngx_http_close_request(r, 0);
        return;
    }

    /* 若原始请求的引用计数为1，则执行以下程序 */

    /*
     * 若keepalive标志为1，表示只需要释放请求，但是当前连接需要复用；
     * 则调用ngx_http_set_keepalive 设置当前连接为keepalive状态；
     */
    if (!ngx_terminate
        && !ngx_exiting
        && r->keepalive
        && clcf->keepalive_timeout > 0)
    {
        ngx_http_set_keepalive(r);
        return;
    }

    /*
     * 若keepalive标志为0，但是lingering_close标志为1，表示需要延迟关闭连接；
     * 则调用ngx_http_set_lingering_close方法延迟关闭请求；
     */
    if (clcf->lingering_close == NGX_HTTP_LINGERING_ALWAYS
        || (clcf->lingering_close == NGX_HTTP_LINGERING_ON
            && (r->lingering_close
                || r->header_in->pos < r->header_in->last
                || r->connection->read->ready())))
    {
        ngx_http_set_lingering_close(r);
        return;
    }

    /* 若keepalive标志为0，且lingering_close标志也为0，则立刻关闭请求 */
    ngx_http_close_request(r, 0);
}
```

ngx_http_close_request 函数的执行流程：

- 将原始请求的引用计数 count 减 1，若此时引用计数 count 不为 0，或当前请求的 blocked 标志位为 1，即不需要正在关闭该请求，因为该请求有其他动作在操作，return 从当前函数返回；
- 若引用计数 count 为 0（表示没有其他动作操作当前请求），且 blocked 标志位为0（表示没有HTTP模块会处理当前请求），因此，调用ngx_http_free_request 释放当前请求的结构体 ngx_http_request_t，并调用函数ngx_http_close_connection 关闭当前连接；

```
/* 关闭当前请求 */
static void
ngx_http_close_request(ngx_http_request_t *r, ngx_int_t rc)
{
    ngx_connection_t  *c;

    r = r->main;
    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http request count:%d blk:%d", r->count, r->blocked);

    if (r->count == 0) {
        ngx_log_error(NGX_LOG_ALERT, c->log, 0, "http request count is zero");
    }

    /* 将原始请求的引用计数减1 */
    r->count--;

    /*
     * 若此时引用计数不为0，或blocked标志位不为0，则该函数到此结束；
     * 到此，即ngx_http_close_request方法的功能只是将原始请求引用计数减1；
     */
    if (r->count || r->blocked) {
        return;
    }

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream) {
        ngx_http_spdy_close_stream(r->spdy_stream, rc);
        return;
    }
#endif

    /*
     * 若引用计数此时为0（表示请求没有其他动作要使用），
     * 且blocked也为0（表示没有HTTP模块还需要处理请求），
     * 则调用ngx_http_free_request释放请求所对应的结构体ngx_http_request_t，
     * 调用ngx_http_close_connection关闭当前连接；
     */
    ngx_http_free_request(r, rc);
    ngx_http_close_connection(c);
}
```

ngx_http_free_request 函数的执行流程：

- 调用当前请求的 cleanup 链表的回调方法 handler 开始清理工作释放资源；
- 在 HTTP 的 NGX_HTTP_LOG_PHASE 阶段调用所有回调方法记录日志；
- 调用 ngx_destroy_pool 方法销毁保存请求结构ngx_http_request_t 的内存池pool；

```
/* 释放当前请求 ngx_http_request_t 的数据结构 */
void
```

```
void
ngx_http_free_request(ngx_http_request_t *r, ngx_int_t rc)
{
    ngx_log_t                *log;
    ngx_pool_t               *pool;
    struct linger             linger;
    ngx_http_cleanup_t       *cln;
    ngx_http_log_ctx_t       *ctx;
    ngx_http_core_loc_conf_t  *clcf;

    log = r->connection->log;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, log, 0, "http close request");

    if (r->pool == NULL) {
        ngx_log_error(NGX_LOG_ALERT, log, 0, "http request already closed");
        return;
    }

    /* 获取当前请求的清理cleanup方法 */
    cln = r->cleanup;
    r->cleanup = NULL;

    /* 调用清理方法cleanup的回调方法handler开始清理工作 */
    while (cln) {
        if (cln->handler) {
            cln->handler(cln->data);
        }

        cln = cln->next;
    }

#if (NGX_STAT_STUB)

    if (r->stat_reading) {
        (void) ngx_atomic_fetch_add(ngx_stat_reading, -1);
    }

    if (r->stat_writing) {
        (void) ngx_atomic_fetch_add(ngx_stat_writing, -1);
    }

#endif

    /* 记录日志 */
    if (rc > 0 && (r->headers_out.status == 0 || r->connection->sent == 0)) {
        r->headers_out.status = rc;
    }

    log->action = "logging request";

    ngx_http_log_request(r);

    log->action = "closing request";
```

```
    if (r->connection->timedout) {
        clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

        if (clcf->reset_timedout_connection) {
            linger.l_onoff = 1;
            linger.l_linger = 0;

            if (setsockopt(r->connection->fd, SOL_SOCKET, SO_LINGER,
                           (const void *) &linger, sizeof(struct linger)) == -1)
            {
                ngx_log_error(NGX_LOG_ALERT, log, ngx_socket_errno,
                              "setsockopt(SO_LINGER) failed");
            }
        }
    }

    /* the various request strings were allocated from r->pool */
    ctx = log->data;
    ctx->request = NULL;

    r->request_line.len = 0;

    r->connection->destroyed = 1;

    /*
     * Setting r->pool to NULL will increase probability to catch double close
     * of request since the request object is allocated from its own pool.
     */

    pool = r->pool;
    r->pool = NULL;

    /* 销毁请求ngx_http_request_t 所对应的内存池 */
    ngx_destroy_pool(pool);
}
```

ngx_http_close_connection 函数的执行流程：

- 将当前连接的 destroyed 标志位设置为 1，表示即将销毁该连接；
- 调用 ngx_close_connection 方法开始销毁连接；
- 最后销毁该链接所使用的内存池 pool；

```
/* 关闭TCP连接 */
void
ngx_http_close_connection(ngx_connection_t *c)
{
    ngx_pool_t  *pool;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "close http connection: %d", c->fd);

#if (NGX_HTTP_SSL)

    if (c->ssl) {
        if (ngx_ssl_shutdown(c) == NGX_AGAIN) {
            c->ssl->handler = ngx_http_close_connection;
            return;
        }
    }

#endif

#if (NGX_STAT_STUB)
    (void) ngx_atomic_fetch_add(ngx_stat_active, -1);
#endif

    /* 设置当前连接的destroyed标志位为1，表示即将销毁该连接 */
    c->destroyed = 1;

    pool = c->pool;

    /* 关闭套接字连接 */
    ngx_close_connection(c);

    /* 销毁连接所使用的内存池 */
    ngx_destroy_pool(pool);
}
```

ngx_close_connection 函数的执行流程：

- 检查读、写事件的 timer_set 标志位，若该标志位都为1，则调用ngx_del_timer 方法将读、写事件从定时器机制中移除；
- 若定义了 ngx_del_conn 宏调用 ngx_del_conn 方法将当前连接上的读、写事件从 epoll 事件机制中移除，若没定义ngx_del_conn 宏，则调用ngx_del_event 方法将读、写事件从epoll 事件机制中移除；
- 调用 ngx_free_connection 方法释放当前连接结构；
- 调用 ngx_close_socket 方法关闭套接字连接；

```
/* 关闭套接字连接 */
void
ngx_close_connection(ngx_connection_t *c)
{
```

```
    ngx_err_t     err;
    ngx_uint_t    log_error, level;
    ngx_socket_t  fd;

    if (c->fd == (ngx_socket_t) -1) {
        ngx_log_error(NGX_LOG_ALERT, c->log, 0, "connection already closed");
        return;
    }

    /* 将当前连接的读、写事件从定时器机制中移除 */
    if (c->read->timer_set) {
        ngx_del_timer(c->read);
    }

    if (c->write->timer_set) {
        ngx_del_timer(c->write);
    }

    /* 将当前连接的读、写事件从epoll事件机制中移除 */
    if (ngx_del_conn) {
        ngx_del_conn(c, NGX_CLOSE_EVENT);

    } else {
        if (c->read->active || c->read->disabled) {
            ngx_del_event(c->read, NGX_READ_EVENT, NGX_CLOSE_EVENT);
        }

        if (c->write->active || c->write->disabled) {
            ngx_del_event(c->write, NGX_WRITE_EVENT, NGX_CLOSE_EVENT);
        }
    }

#if (NGX_THREADS)

    /*
     * we have to clean the connection information before the closing
     * because another thread may reopen the same file descriptor
     * before we clean the connection
     */

    ngx_mutex_lock(ngx_posted_events_mutex);

    if (c->read->prev) {
        ngx_delete_posted_event(c->read);
    }

    if (c->write->prev) {
        ngx_delete_posted_event(c->write);
    }

    c->read->closed = 1;
    c->write->closed = 1;

    ngx_unlock(&c->lock);
    c->read->locked = 0;
```

```
    c->read->locked = 0;
    c->write->locked = 0;

    ngx_mutex_unlock(ngx_posted_events_mutex);

#else

    if (c->read->prev) {
        ngx_delete_posted_event(c->read);
    }

    if (c->write->prev) {
        ngx_delete_posted_event(c->write);
    }

    c->read->closed = 1;
    c->write->closed = 1;

#endif

    ngx_reusable_connection(c, 0);

    log_error = c->log_error;

    /* 释放当前连接结构体 */
    ngx_free_connection(c);

    fd = c->fd;
    c->fd = (ngx_socket_t) -1;

    /* 关闭套接字连接 */
    if (ngx_close_socket(fd) == -1) {

        err = ngx_socket_errno;

        if (err == NGX_ECONNRESET || err == NGX_ENOTCONN) {

            switch (log_error) {

            case NGX_ERROR_INFO:
                level = NGX_LOG_INFO;
                break;

            case NGX_ERROR_ERR:
                level = NGX_LOG_ERR;
                break;

            default:
                level = NGX_LOG_CRIT;
            }

        } else {
            level = NGX_LOG_CRIT;
        }
```

```
        /* we use ngx_cycle->log because c->log was in c->pool */

        ngx_log_error(level, ngx_cycle->log, err,
                      ngx_close_socket_n " %d failed", fd);
    }
}
```

ngx_http_terminate_request 函数的执行流程：

- 调用原始请求的 cleanup 链表的回调方法 handler 开始清理工作；
- 调用 ngx_http_close_request 方法关闭请求；

```
/* 强制关闭连接 */
static void
ngx_http_terminate_request(ngx_http_request_t *r, ngx_int_t rc)
{
    ngx_http_cleanup_t    *cln;
    ngx_http_request_t    *mr;
    ngx_http_ephemeral_t  *e;

    mr = r->main;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http terminate request count:%d", mr->count);

    if (rc > 0 && (mr->headers_out.status == 0 || mr->connection->sent == 0)) {
        mr->headers_out.status = rc;
    }

    /* 调用原始请求的cleanup的回调方法，开始清理工作 */
    cln = mr->cleanup;
    mr->cleanup = NULL;

    while (cln) {
        if (cln->handler) {
            cln->handler(cln->data);
        }

        cln = cln->next;
    }

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http terminate cleanup count:%d blk:%d",
                   mr->count, mr->blocked);

    if (mr->write_event_handler) {

        if (mr->blocked) {
            return;
        }

        e = ngx_http_ephemeral(mr);
        mr->posted_requests = NULL;
        mr->write_event_handler = ngx_http_terminate_handler;
        (void) ngx_http_post_request(mr, &e->terminal_posted_request);
        return;
    }

    /* 释放请求，并关闭连接 */
    ngx_http_close_request(mr, rc);
}
```

Nginx 中 upstream 机制的实现

概述

upstream 机制使得 Nginx 成为一个反向代理服务器，Nginx 接收来自下游客户端的 http 请求，并处理该请求，同时根据该请求向上游服务器发送 tcp 请求报文，上游服务器会根据该请求返回相应地响应报文，Nginx 根据上游服务器的响应报文，决定是否向下游客户端转发响应报文。另外 upstream 机制提供了负载均衡的功能，可以将请求负载均衡到集群服务器的某个服务器上面。

启动 upstream

在 Nginx 中调用 ngx_http_upstream_init 方法启动 upstream 机制，但是在使用 upstream 机制之前必须调用 ngx_http_upstream_create 方法创建 ngx_http_upstream_t 结构体，因为默认情况下 ngx_http_request_t 结构体中的 upstream 成员是指向 NULL，该结构体的具体初始化工作还需由 HTTP 模块完成。有关 ngx_http_upstream_t 结构体 和ngx_http_upstream_conf_t 结构体的相关说明可参考文章《Nginx 中 upstream 机制》。

下面是函数 ngx_http_upstream_create 的实现：

```
/* 创建 ngx_http_upstream_t 结构体 */
ngx_int_t
ngx_http_upstream_create(ngx_http_request_t *r)
{
    ngx_http_upstream_t  *u;

    u = r->upstream;

    /*
     * 若已经创建过ngx_http_upstream_t 且定义了cleanup成员，
     * 则调用cleanup清理方法将原始结构体清除；
     */
    if (u && u->cleanup) {
        r->main->count++;
        ngx_http_upstream_cleanup(r);
    }

    /* 从内存池分配ngx_http_upstream_t 结构体空间 */
    u = ngx_pcalloc(r->pool, sizeof(ngx_http_upstream_t));
    if (u == NULL) {
        return NGX_ERROR;
    }

    /* 给ngx_http_request_t 结构体成员upstream赋值 */
    r->upstream = u;

    u->peer.log = r->connection->log;
    u->peer.log_error = NGX_ERROR_ERR;
#if (NGX_THREADS)
    u->peer.lock = &r->connection->lock;
#endif

#if (NGX_HTTP_CACHE)
    r->cache = NULL;
#endif

    u->headers_in.content_length_n = -1;

    return NGX_OK;
}
```

关于 upstream 机制的启动方法 ngx_http_upstream_init 的执行流程如下：

- 检查 Nginx 与下游服务器之间连接上的读事件是否在定时器中，即检查 timer_set 标志位是否为 1，若该标志位为 1，则把读事件从定时器中移除；
- 调用 ngx_http_upstream_init_request 方法启动 upstream 机制；

ngx_http_upstream_init_request 方法执行流程如下所示：

- 检查 ngx_http_upstream_t 结构体中的 store 标志位是否为 0；检查 ngx_http_request_t 结构体中的 post_action 标志位是否为0；检查 ngx_http_upstream_conf_t 结构体中的ignore_client_abort

是否为 0；若上面的标志位都为 0，则设置ngx_http_request_t 请求的读事件的回调方法为 ngx_http_upstream_rd_check_broken_connection；设置写事件的回调方法为 ngx_http_upstream_wr_check_broken_connection；这两个方法都会调用 ngx_http_upstream_check_broken_connection方法检查 Nginx 与下游之间的连接是否正常，若出现错误，则终止连接；

- 若不满足上面的标志位，即至少有一个不为 0 ，调用请求中ngx_http_upstream_t 结构体中某个 HTTP 模块实现的create_request 方法，构造发往上游服务器的请求；
- 调用 ngx_http_cleanup_add 方法向原始请求的 cleanup 链表尾端添加一个回调 handler 方法，该回调方法设置为ngx_http_upstream_cleanup，若当前请求结束时会调用该方法做一些清理工作；
- 调用 ngx_http_upstream_connect 方法向上游服务器发起连接请求；

```
/* 初始化启动upstream机制 */
void
ngx_http_upstream_init(ngx_http_request_t *r)
{
    ngx_connection_t     *c;

    /* 获取当前请求所对应的连接 */
    c = r->connection;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http init upstream, client timer: %d", c->read->timer_set);

#if (NGX_HTTP_SPDY)
    if (r->spdy_stream) {
        ngx_http_upstream_init_request(r);
        return;
    }
#endif

    /*
     * 检查当前连接上读事件的timer_set标志位是否为1，若该标志位为1，
     * 表示读事件在定时器机制中，则需要把它从定时器机制中移除；
     * 因为在启动upstream机制后，就不需要对客户端的读操作进行超时管理；
     */
    if (c->read->timer_set) {
        ngx_del_timer(c->read);
    }

    if (ngx_event_flags & NGX_USE_CLEAR_EVENT) {

        if (!c->write->active) {
            if (ngx_add_event(c->write, NGX_WRITE_EVENT, NGX_CLEAR_EVENT)
                == NGX_ERROR)
            {
                ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }
        }
    }

    ngx_http_upstream_init_request(r);
}
```

```
static void
ngx_http_upstream_init_request(ngx_http_request_t *r)
{
    ngx_str_t                  *host;
    ngx_uint_t                  i;
    ngx_resolver_ctx_t         *ctx, temp;
    ngx_http_cleanup_t         *cln;
    ngx_http_upstream_t        *u;
    ngx_http_core_loc_conf_t   *clcf;
    ngx_http_upstream_srv_conf_t  *uscf, **uscfp;
```

```
    ngx_http_upstream_srv_conf_t  **uscf, **uscfp;
    ngx_http_upstream_main_conf_t  *umcf;

    if (r->aio) {
        return;
    }

    u = r->upstream;

#if (NGX_HTTP_CACHE)
    ...
    ...
#endif

    /* 文件缓存标志位 */
    u->store = (u->conf->store || u->conf->store_lengths);

    /*
     * 检查ngx_http_upstream_t 结构中标志位 store；
     * 检查ngx_http_request_t 结构中标志位 post_action；
     * 检查ngx_http_upstream_conf_t 结构中标志位 ignore_client_abort；
     * 若上面这些标志位为1，则表示需要检查Nginx与下游(即客户端)之间的TCP连接是否断开；
     */
    if (!u->store && !r->post_action && !u->conf->ignore_client_abort) {
        r->read_event_handler = ngx_http_upstream_rd_check_broken_connection;
        r->write_event_handler = ngx_http_upstream_wr_check_broken_connection;
    }

    /* 把当前请求包体结构保存在ngx_http_upstream_t 结构的request_bufs链表缓冲区中 */
    if (r->request_body) {
        u->request_bufs = r->request_body->bufs;
    }

    /* 调用create_request方法构造发往上游服务器的请求 */
    if (u->create_request(r) != NGX_OK) {
        ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    /* 获取ngx_http_upstream_t结构中主动连接结构peer的local本地地址信息 */
    u->peer.local = ngx_http_upstream_get_local(r, u->conf->local);

    /* 获取ngx_http_core_module模块的loc级别的配置项结构 */
    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    /* 初始化ngx_http_upstream_t结构中成员output向下游发送响应的方式 */
    u->output.alignment = clcf->directio_alignment;
    u->output.pool = r->pool;
    u->output.bufs.num = 1;
    u->output.bufs.size = clcf->client_body_buffer_size;
    u->output.output_filter = ngx_chain_writer;
    u->output.filter_ctx = &u->writer;

    u->writer.pool = r->pool;
```

```
    /* 添加用于表示上游响应的状态，例如：错误编码、包体长度等 */
    if (r->upstream_states == NULL) {

        r->upstream_states = ngx_array_create(r->pool, 1,
                                              sizeof(ngx_http_upstream_state_t));
        if (r->upstream_states == NULL) {
            ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

    } else {

        u->state = ngx_array_push(r->upstream_states);
        if (u->state == NULL) {
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        ngx_memzero(u->state, sizeof(ngx_http_upstream_state_t));
    }

    /*
     * 调用ngx_http_cleanup_add方法原始请求的cleanup链表尾端添加一个回调handler方法，
     * 该handler回调方法设置为ngx_http_upstream_cleanup，若当前请求结束时会调用该方法做一些清理工
作；
     */
    cln = ngx_http_cleanup_add(r, 0);
    if (cln == NULL) {
        ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    cln->handler = ngx_http_upstream_cleanup;
    cln->data = r;
    u->cleanup = &cln->handler;

    if (u->resolved == NULL) {

        /* 若没有实现u->resolved标志位，则定义上游服务器的配置 */
        uscf = u->conf->upstream;

    } else {

        /*
         * 若实现了u->resolved标志位，则解析主机域名，指定上游服务器的地址；
         */

        /*
         * 若已经指定了上游服务器地址，则不需要解析，
         * 直接调用ngx_http_upstream_connection方法向上游服务器发起连接；
         * 并return从当前函数返回；
         */
        if (u->resolved->sockaddr) {
```

```
    if (ngx_http_upstream_create_round_robin_peer(r, u->resolved)
        != NGX_OK)
    {
        ngx_http_upstream_finalize_request(r, u,
                                NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    ngx_http_upstream_connect(r, u);

    return;
}

/*
 * 若还没指定上游服务器的地址，则需解析主机域名；
 * 若成功解析出上游服务器的地址和端口号，
 * 则调用ngx_http_upstream_connection方法向上游服务器发起连接；
 */
host = &u->resolved->host;

umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);

uscfp = umcf->upstreams.elts;

for (i = 0; i < umcf->upstreams.nelts; i++) {

    uscf = uscfp[i];

    if (uscf->host.len == host->len
        && ((uscf->port == 0 && u->resolved->no_port)
            || uscf->port == u->resolved->port)
        && ngx_strncasecmp(uscf->host.data, host->data, host->len) == 0)
    {
        goto found;
    }
}

if (u->resolved->port == 0) {
    ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                  "no port in upstream \"%V\"", host);
    ngx_http_upstream_finalize_request(r, u,
                                NGX_HTTP_INTERNAL_SERVER_ERROR);
    return;
}

temp.name = *host;

ctx = ngx_resolve_start(clcf->resolver, &temp);
if (ctx == NULL) {
    ngx_http_upstream_finalize_request(r, u,
                                NGX_HTTP_INTERNAL_SERVER_ERROR);
    return;
}
```

```
        if (ctx == NGX_NO_RESOLVER) {
            ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                          "no resolver defined to resolve %V", host);

            ngx_http_upstream_finalize_request(r, u, NGX_HTTP_BAD_GATEWAY);
            return;
        }

        ctx->name = *host;
        ctx->handler = ngx_http_upstream_resolve_handler;
        ctx->data = r;
        ctx->timeout = clcf->resolver_timeout;

        u->resolved->ctx = ctx;

        if (ngx_resolve_name(ctx) != NGX_OK) {
            u->resolved->ctx = NULL;
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        return;
    }

found:

    if (uscf == NULL) {
        ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
                      "no upstream configuration");
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    if (uscf->peer.init(r, uscf) != NGX_OK) {
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    ngx_http_upstream_connect(r, u);
}

static void
ngx_http_upstream_rd_check_broken_connection(ngx_http_request_t *r)
{
    ngx_http_upstream_check_broken_connection(r, r->connection->read);
}

static void
ngx_http_upstream_wr_check_broken_connection(ngx_http_request_t *r)
{
    ngx_http_upstream_check_broken_connection(r, r->connection->write);
}
```

```
}
```

建立连接

upstream 机制与上游服务器建立 TCP 连接时，采用的是非阻塞模式的套接字，即发起连接请求之后立即返回，不管连接是否建立成功，若没有立即建立成功，则需在 epoll 事件机制中监听该套接字。向上游服务器发起连接请求由函数ngx_http_upstream_connect 实现。在分析 ngx_http_upstream_connect 方法之前，首先分析下 ngx_event_connect_peer 方法，因为该方法会被ngx_http_upstream_connect 方法调用。

ngx_event_connect_peer 方法的执行流程如下所示：

- 调用 ngx_socket 方法创建一个 TCP 套接字；
- 调用 ngx_nonblocking 方法设置该 TCP 套接字为非阻塞模式；
- 设置套接字连接接收和发送网络字符流的方法；
- 设置套接字连接上读、写事件方法；
- 将 TCP 套接字以期待 EPOLLIN | EPOLLOUT 事件的方式添加到epoll 事件机制中；
- 调用 connect 方法向服务器发起 TCP 连接请求；

ngx_http_upstream_connect 方法表示向上游服务器发起连接请求，其执行流程如下所示：

- 调用 ngx_event_connect_peer 方法主动向上游服务器发起连接请求，需要注意的是该方法已经将相应的套接字注册到epoll事件机制来监听读、写事件，该方法返回值为 rc；
- 若 rc = NGX_ERROR，表示发起连接失败，则调用ngx_http_upstream_finalize_request 方法关闭连接请求，并 return 从当前函数返回；
- 若 rc = NGX_BUSY，表示当前上游服务器处于不活跃状态，则调用 ngx_http_upstream_next 方法根据传入的参数尝试重新发起连接请求，并 return 从当前函数返回；
- 若 rc = NGX_DECLINED，表示当前上游服务器负载过重，则调用 ngx_http_upstream_next 方法尝试与其他上游服务器建立连接，并 return 从当前函数返回；
- 设置上游连接 ngx_connection_t 结构体的读事件、写事件的回调方法 handler 都为 ngx_http_upstream_handler，设置 ngx_http_upstream_t 结构体的写事件 write_event_handler 的回调为 ngx_http_upstream_send_request_handler，读事件 read_event_handler 的回调方法为 ngx_http_upstream_process_header；
- 若 rc = NGX_AGAIN，表示当前已经发起连接，但是没有收到上游服务器的确认应答报文，即上游连接的写事件不可写，则需调用 ngx_add_timer 方法将上游连接的写事件添加到定时器中，管理超时确认应答；
- 若 rc = NGX_OK，表示成功建立连接，则调用 ngx_http_upsream_send_request 方法向上游服务器发送请求；

```
/* 向上游服务器建立连接 */
static void
ngx_http_upstream_connect(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
    ngx_int_t          rc;
    ngx_time_t        *tp;
    ngx_connection_t  *c;

    r->connection->log->action = "connecting to upstream";

    if (u->state && u->state->response_sec) {
        tp = ngx_timeofday();
        u->state->response_sec = tp->sec - u->state->response_sec;
        u->state->response_msec = tp->msec - u->state->response_msec;
    }

    u->state = ngx_array_push(r->upstream_states);
    if (u->state == NULL) {
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    ngx_memzero(u->state, sizeof(ngx_http_upstream_state_t));

    tp = ngx_timeofday();
    u->state->response_sec = tp->sec;
    u->state->response_msec = tp->msec;

    /* 向上游服务器发起连接 */
    rc = ngx_event_connect_peer(&u->peer);

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http upstream connect: %i", rc);

    /* 下面根据rc不同返回值进行分析 */

    /* 若建立连接失败，则关闭当前请求，并return从当前函数返回 */
    if (rc == NGX_ERROR) {
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    u->state->peer = u->peer.name;

    /*
     * 若返回rc = NGX_BUSY，表示当前上游服务器不活跃，
     * 则调用ngx_http_upstream_next向上游服务器重新发起连接，
     * 实际上，该方法最终还是调用ngx_http_upstream_connect方法；
     * 并return从当前函数返回；
     */
    if (rc == NGX_BUSY) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no live upstreams");
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_NOLIVE);
```

```
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_NOLIVE);
        return;
    }

    /*
     * 若返回rc = NGX_DECLINED，表示当前上游服务器负载过重，
     * 则调用ngx_http_upstream_next向上游服务器重新发起连接，
     * 实际上，该方法最终还是调用ngx_http_upstream_connect方法；
     * 并return从当前函数返回；
     */
    if (rc == NGX_DECLINED) {
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
        return;
    }

    /* rc == NGX_OK || rc == NGX_AGAIN || rc == NGX_DONE */

    c = u->peer.connection;

    c->data = r;

    /* 设置当前连接ngx_connection_t 上读、写事件的回调方法 */
    c->write->handler = ngx_http_upstream_handler;
    c->read->handler = ngx_http_upstream_handler;

    /* 设置upstream机制的读、写事件的回调方法 */
    u->write_event_handler = ngx_http_upstream_send_request_handler;
    u->read_event_handler = ngx_http_upstream_process_header;

    c->sendfile &= r->connection->sendfile;
    u->output.sendfile = c->sendfile;

    if (c->pool == NULL) {

        /* we need separate pool here to be able to cache SSL connections */

        c->pool = ngx_create_pool(128, r->connection->log);
        if (c->pool == NULL) {
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }
    }

    c->log = r->connection->log;
    c->pool->log = c->log;
    c->read->log = c->log;
    c->write->log = c->log;

    /* init or reinit the ngx_output_chain() and ngx_chain_writer() contexts */

    u->writer.out = NULL;
    u->writer.last = &u->writer.out;
    u->writer.connection = c;
    u->writer.limit = 0;
```

```
    /*
     * 检查当前ngx_http_upstream_t 结构的request_sent标志位，
     * 若该标志位为1，则表示已经向上游服务器发送请求，即本次发起连接失败；
     * 则调用ngx_http_upstream_reinit方法重新向上游服务器发起连接；
     */
    if (u->request_sent) {
        if (ngx_http_upstream_reinit(r, u) != NGX_OK) {
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }
    }

    if (r->request_body
        && r->request_body->buf
        && r->request_body->temp_file
        && r == r->main)
    {
        /*
         * the r->request_body->buf can be reused for one request only,
         * the subrequests should allocate their own temporary bufs
         */

        u->output.free = ngx_alloc_chain_link(r->pool);
        if (u->output.free == NULL) {
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        u->output.free->buf = r->request_body->buf;
        u->output.free->next = NULL;
        u->output.allocated = 1;

        r->request_body->buf->pos = r->request_body->buf->start;
        r->request_body->buf->last = r->request_body->buf->start;
        r->request_body->buf->tag = u->output.tag;
    }

    u->request_sent = 0;

    /*
     * 若返回rc = NGX_AGAIN，表示没有收到上游服务器允许建立连接的应答；
     * 由于写事件已经添加到epoll事件机制中等待可写事件发生，
     * 所有在这里只需将当前连接的写事件添加到定时器机制中进行超时管理；
     * 并return从当前函数返回；
     */
    if (rc == NGX_AGAIN) {
        ngx_add_timer(c->write, u->conf->connect_timeout);
        return;
    }

#if (NGX_HTTP_SSL)
```

```
    if (u->ssl && c->ssl == NULL) {
        ngx_http_upstream_ssl_init_connection(r, u, c);
        return;
    }

#endif

    /*
     * 若返回值rc = NGX_OK，表示连接成功建立，
     * 调用此方法向上游服务器发送请求 */
    ngx_http_upstream_send_request(r, u);
}
```

发送请求

当 Nginx 与上游服务器成功建立连接之后，会调用 ngx_http_upstream_send_request 方法发送请求，若是该方法不能一次性把请求内容发送完成时，则需等待 epoll 事件机制的写事件发生，若写事件发生，则会调用写事件 write_event_handler 的回调方法 ngx_http_upstream_send_request_handler 继续发送请求，并且有可能会多次调用该写事件的回调方法， 直到把请求发送完成。

下面是 ngx_http_upstream_send_request 方法的执行流程：

- 检查 ngx_http_upstream_t 结构体中的标志位 request_sent 是否为 0，若为 0 表示未向上游发送请求。 且此时调用 ngx_http_upstream_test_connect 方法测试是否与上游建立连接，若返回非 NGX_OK， 则需调用 ngx_http_upstream_next 方法试图与上游建立连接，并return 从当前函数返回；
- 调用 ngx_output_chain 方法向上游发送保存在 request_bufs 链表中的请求数据，该方法返回值为 rc，并设置 request_sent 标志位为 1，检查连接上写事件 timer_set 标志位是否为1，若为 1 调用 ngx_del_timer 方法将写事件从定时器中移除；
- 若 rc = NGX_ERROR，表示当前连接上出错，则调用 ngx_http_upstream_next 方法尝试再次与上游建立连接，并 return 从当前函数返回；
- 若 rc = NGX_AGAIN，并是当前请求数据未完全发送，则需将剩余的请求数据保存在 ngx_http_upstream_t 结构体的 output 成员中，并且调用 ngx_add_timer 方法将当前连接上的写事件添加到定时器中，调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中，等待可写事件发生，并return 从当前函数返回；
- 若 rc = NGX_OK，表示已经发送全部请求数据，则准备接收来自上游服务器的响应报文；
- 先调用 ngx_add_timer 方法将当前连接的读事件添加到定时器机制中，检测接收响应是否超时，检查当前连接上的读事件是否准备就绪，即标志位 ready 是否为1，若该标志位为 1，则调用 ngx_http_upstream_process_header 方法开始处理响应头部，并 return 从当前函数返回；
- 若当前连接上读事件的标志位 ready 为0，表示暂时无可读数据，则需等待读事件再次被触发，由于原始读事件的回调方法为 ngx_http_upstream_process_header，所有无需重新设置。由于请求已经全部发送，防止写事件的回调方法 ngx_http_upstream_send_request_handler 再次被触发，因此需要重新设置写事件的回调方法为 ngx_http_upstream_dummy_handler，该方法实际上不执行任何操

作，同时调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中；

```
/* 向上游服务器发送请求 */
static void
ngx_http_upstream_send_request(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
    ngx_int_t         rc;
    ngx_connection_t  *c;

    /* 获取当前连接 */
    c = u->peer.connection;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream send request");

    /*
     * 若标志位request_sent为0，表示还未发送请求；
     * 且ngx_http_upstream_test_connect方法返回非NGX_OK，标志当前还未与上游服务器成功建立连接；
     * 则需要调用ngx_http_upstream_next方法尝试与下一个上游服务器建立连接；
     * 并return从当前函数返回；
     */
    if (!u->request_sent && ngx_http_upstream_test_connect(c) != NGX_OK) {
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
        return;
    }

    c->log->action = "sending request to upstream";

    /*
     * 调用ngx_output_chain方法向上游发送保存在request_bufs链表中的请求数据；
     * 值得注意的是该方法的第二个参数可以是NULL也可以是request_bufs，那怎么来区分呢？
     * 若是第一次调用该方法发送request_bufs链表中的请求数据时，request_sent标志位为0，
     * 此时，第二个参数自然就是request_bufs了，那么为什么会有NULL作为参数的情况呢？
     * 当在第一次调用该方法时，并不能一次性把所有request_bufs中的数据发送完毕时，
     * 此时，会把剩余的数据保存在output结构里面，并把标志位request_sent设置为1，
     * 因此，再次发送请求数据时，不用指定request_bufs参数，因为此时剩余数据已经保存在output中；
     */
    rc = ngx_output_chain(&u->output, u->request_sent ? NULL : u->request_bufs);

    /* 向上游服务器发送请求之后，把request_sent标志位设置为1 */
    u->request_sent = 1;

    /* 下面根据不同rc的返回值进行判断 */

    /*
     * 若返回值rc=NGX_ERROR，表示当前连接上出错，
     * 将错误信息传递给ngx_http_upstream_next方法，
     * 该方法根据错误信息决定是否重新向上游服务器发起连接；
     * 并return从当前函数返回；
     */
    if (rc == NGX_ERROR) {
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
        return;
    }
```

```
    /*
     * 检查当前连接上写事件的标志位timer_set是否为1，
     * 若该标志位为1，则需把写事件从定时器机制中移除；
     */
    if (c->write->timer_set) {
        ngx_del_timer(c->write);
    }

    /*
     * 若返回值rc = NGX_AGAIN，表示请求数据并未完全发送，
     * 即有剩余的请求数据保存在output中，但此时，写事件已经不可写，
     * 则调用ngx_add_timer方法把当前连接上的写事件添加到定时器机制，
     * 并调用ngx_handle_write_event方法将写事件注册到epoll事件机制中；
     * 并return从当前函数返回；
     */
    if (rc == NGX_AGAIN) {
        ngx_add_timer(c->write, u->conf->send_timeout);

        if (ngx_handle_write_event(c->write, u->conf->send_lowat) != NGX_OK) {
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        return;
    }

    /* rc == NGX_OK */

    /*
     * 若返回值 rc = NGX_OK，表示已经发送完全部请求数据，
     * 准备接收来自上游服务器的响应报文，则执行以下程序；
     */
    if (c->tcp_nopush == NGX_TCP_NOPUSH_SET) {
        if (ngx_tcp_push(c->fd) == NGX_ERROR) {
            ngx_log_error(NGX_LOG_CRIT, c->log, ngx_socket_errno,
                          ngx_tcp_push_n " failed");
            ngx_http_upstream_finalize_request(r, u,
                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        c->tcp_nopush = NGX_TCP_NOPUSH_UNSET;
    }

    /* 将当前连接上读事件添加到定时器机制中 */
    ngx_add_timer(c->read, u->conf->read_timeout);

    /*
     * 若此时，读事件已经准备就绪，
     * 则调用ngx_http_upstream_process_header方法开始接收并处理响应头部；
     * 并return从当前函数返回；
     */
    if (
```

```
    if (c->read->ready) {
        ngx_http_upstream_process_header(r, u);
        return;
    }

    /*
     * 若当前读事件未准备就绪；
     * 则把写事件的回调方法设置为ngx_http_upstream_dumy_handler方法(不进行任何实际操作)；
     * 并把写事件注册到epoll事件机制中；
     */
    u->write_event_handler = ngx_http_upstream_dummy_handler;

    if (ngx_handle_write_event(c->write, 0) != NGX_OK) {
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }
}
```

当无法一次性将请求内容全部发送完毕，则需等待 epoll 事件机制的写事件发生，一旦发生就会调用回调方法 ngx_http_upstream_send_request_handler。

ngx_http_upstream_send_request_handler 方法的执行流程如下所示：

- 检查连接上写事件是否超时，即timedout 标志位是否为 1，若为 1 表示已经超时，则调用 ngx_http_upstream_next 方法重新向上游发起连接请求，并 return 从当前函数返回；
- 若标志位 timedout 为0，即不超时，检查 header_sent 标志位是否为 1，表示已经接收到来自上游服务器的响应头部，则不需要再向上游发送请求，将写事件的回调方法设置为 ngx_http_upstream_dummy_handler，同时将写事件注册到 epoll 事件机制中，并return 从当前函数返回；
- 若标志位 header_sent 为 0，则调用 ngx_http_upstream_send_request 方法向上游发送请求数据；

```
static void
ngx_http_upstream_send_request_handler(ngx_http_request_t *r,
    ngx_http_upstream_t *u)
{
    ngx_connection_t  *c;

    c = u->peer.connection;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "http upstream send request handler");

    /* 检查当前连接上写事件的超时标志位 */
    if (c->write->timedout) {
        /* 执行超时重连机制 */
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_TIMEOUT);
        return;
    }

#if (NGX_HTTP_SSL)

    if (u->ssl && c->ssl == NULL) {
        ngx_http_upstream_ssl_init_connection(r, u, c);
        return;
    }

#endif

    /* 已经接收到上游服务器的响应头部，则不需要再向上游服务器发送请求数据 */
    if (u->header_sent) {
        /* 将写事件的回调方法设置为不进行任何实际操作的方法ngx_http_upstream_dumy_handler */
        u->write_event_handler = ngx_http_upstream_dummy_handler;

        /* 将写事件注册到epoll事件机制中，并return从当前函数返回 */
        (void) ngx_handle_write_event(c->write, 0);

        return;
    }

    /* 若没有接收来自上游服务器的响应头部，则需向上游服务器发送请求数据 */
    ngx_http_upstream_send_request(r, u);
}
```

接收响应

接收响应头部

当 Nginx 已经向上游发送请求，准备开始接收来自上游的响应头部，由方法 ngx_http_upstream_process_header 实现，该方法接收并解析响应头部。

ngx_http_upstream_process_header 方法的执行流程如下：

- 检查上游连接上的读事件是否超时，若标志位 timedout 为 1，则表示超时，此时调用 ngx_http_upstream_next 方法重新与上游建立连接，并 return 从当前函数返回；
- 若标志位 timedout 为 0，接着检查 ngx_http_upstream_t 结构体中的标志位 request_sent，若该标志位为 0，表示未向上游发送请求，同时调用 ngx_http_upstream_test_connect 方法测试连接状态，若该方法返回值为非 NGX_OK，表示与上游已经断开连接，则调用 ngx_http_upstream_next 方法重新与上游建立连接，并 return 从当前函数返回；
- 检查 ngx_http_upstream_t 结构体中接收响应头部的 buffer 缓冲区是否有内存空间以便接收响应头部，若 buffer.start 为 NULL，表示该缓冲区为空，则需调用 ngx_palloc 方法分配内存，该内存大小 buffer_size 由 ngx_http_upstream_conf_t 配置结构体的 buffer_size 成员指定；
- 调用 recv 方法开始接收来自上游服务器的响应头部，并根据该方法的返回值 n 进行判断：
- 若 n = NGX_AGAIN，表示读事件未准备就绪，需要等待下次读事件被触发时继续接收响应头部，此时，调用 ngx_add_timer 方法将读事件添加到定时器中，同时调用 ngx_handle_read_event 方法将读事件注册到epoll 事件机制中，并 return 从当前函数返回；
- 若 n = NGX_ERROR 或 n = 0，表示上游连接发生错误 或 上游服务器主动关闭连接，则调用 ngx_http_upstream_next 方法重新发起连接请求，并 return 从当前函数返回；
- 若 n 大于 0，表示已经接收到响应头部，此时，调用 ngx_http_upstream_t 结构体中由 HTTP 模块实现的 process_header 方法解析响应头部，且返回 rc 值；
- 若 rc = NGX_AGAIN，表示接收到的响应头部不完整，检查接收缓冲区 buffer 是否还有剩余的内存空间，若缓冲区没有剩余的内存空间，表示接收到的响应头部过大，此时调用 ngx_http_upstream_next 方法重新建立连接，并 return 从当前函数返回；若缓冲区还有剩余的内存空间，则continue 继续接收响应头部；
- 若 rc = NGX_HTTP_UPSTREAM_INVALID_HEADER，表示接收到的响应头部是非法的，则调用 ngx_http_upstream_next 方法重新建立连接，并 return 从当前函数返回；
- 若 rc = NGX_ERROR，表示连接出错，此时调用 ngx_http_upstream_finalize_request 方法结束请求，并 return 从当前函数返回；
- 若 rc = NGX_OK，表示已接收到完整的响应头部，则调用 ngx_http_upstream_process_headers 方法处理已解析的响应头部，该方法会将已解析出来的响应头部保存在 ngx_http_request_t 结构体中的 headers_out 成员；
- 检查 ngx_http_request_t 结构体的 subrequest_in_memory 成员决定是否需要转发响应给下游服务器；
- 若 subrequest_in_memory 为 0，表示需要转发响应给下游服务器，则调用 ngx_http_upstream_send_response 方法开始转发响应给下游服务器，并 return 从当前函数返回；

- 若 subrequest_in_memory 为 1，表示不需要将响应转发给下游，此时检查 HTTP 模块是否定义了 ngx_http_upstream_t 结构体中的 input_filter 方法处理响应包体；
- 若没有定义 input_filter 方法，则使用 upstream 机制默认方法 ngx_http_upstream_non_buffered_filter 代替 input_filter 方法；
- 若定义了自己的 input_filter 方法，则首先调用 input_filter_init 方法为处理响应包体做初始化工作；
- 检查接收缓冲区 buffer 在解析完响应头部之后剩余的字符流，若有剩余的字符流，则表示已经预接收了响应包体，此时调用 input_filter 方法处理响应包体；
- 设置 upstream 机制读事件 read_event_handler 的回调方法为 ngx_http_upstream_process_body_in_memory，并调用该方法开始接收并解析响应包体；

```
/* 接收并解析响应头部 */
static void
ngx_http_upstream_process_header(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
    ssize_t            n;
    ngx_int_t          rc;
    ngx_connection_t  *c;

    c = u->peer.connection;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process header");

    c->log->action = "reading response header from upstream";

    /* 检查当前连接上的读事件是否超时 */
    if (c->read->timedout) {
        /*
         * 若标志位timedout为1，表示读事件超时；
         * 则把超时错误传递给ngx_http_upstream_next方法，
         * 该方法根据允许的错误进行重连接策略；
         * 并return从当前函数返回；
         */
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_TIMEOUT);
        return;
    }

    /*
     * 若标志位request_sent为0，表示还未发送请求；
     * 且ngx_http_upstream_test_connect方法返回非NGX_OK，标志当前还未与上游服务器成功建立连接；
     * 则需要调用ngx_http_upstream_next方法尝试与下一个上游服务器建立连接；
     * 并return从当前函数返回；
     */
    if (!u->request_sent && ngx_http_upstream_test_connect(c) != NGX_OK) {
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
        return;
    }
```

```
    /*
     * 检查ngx_http_upstream_t结构体中接收响应头部的buffer缓冲区；
     * 若接收缓冲区buffer未分配内存，则调用ngx_palloce方法分配内存，
     * 该内存的大小buffer_size由ngx_http_upstream_conf_t配置结构的buffer_size指定；
     */
    if (u->buffer.start == NULL) {
        u->buffer.start = ngx_palloc(r->pool, u->conf->buffer_size);
        if (u->buffer.start == NULL) {
            ngx_http_upstream_finalize_request(r, u,
                                    NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

        /* 调整接收缓冲区buffer，准备接收响应头部 */
        u->buffer.pos = u->buffer.start;
        u->buffer.last = u->buffer.start;
        u->buffer.end = u->buffer.start + u->conf->buffer_size;
        /* 表示该缓冲区内存可被复用、数据可被改变 */
        u->buffer.temporary = 1;

        u->buffer.tag = u->output.tag;

        /* 初始化headers_in的成员headers链表 */
        if (ngx_list_init(&u->headers_in.headers, r->pool, 8,
                          sizeof(ngx_table_elt_t))
            != NGX_OK)
        {
            ngx_http_upstream_finalize_request(r, u,
                                    NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }

#if (NGX_HTTP_CACHE)

        if (r->cache) {
            u->buffer.pos += r->cache->header_start;
            u->buffer.last = u->buffer.pos;
        }
#endif
    }

    for ( ;; ) {

        /* 调用recv方法从当前连接上读取响应头部数据 */
        n = c->recv(c, u->buffer.last, u->buffer.end - u->buffer.last);

        /* 下面根据 recv 方法不同返回值 n 进行判断 */

        /*
         * 若返回值 n = NGX_AGAIN，表示读事件未准备就绪，
         * 需等待下次读事件被触发时继续接收响应头部，
         * 即将读事件注册到epoll事件机制中，等待可读事件发生；
         * 并return从当前函数返回；
         */
        if (n == NGX_AGAIN) {
```

```
        if (n == NGX_AGAIN) {
#if 0
            ngx_add_timer(rev, u->read_timeout);
#endif

            if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
                ngx_http_upstream_finalize_request(r, u,
                                                   NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }

            return;
        }

        if (n == 0) {
            ngx_log_error(NGX_LOG_ERR, c->log, 0,
                          "upstream prematurely closed connection");
        }

        /*
         * 若返回值 n = NGX_ERROR 或 n = 0，则表示上游服务器已经主动关闭连接；
         * 此时，调用ngx_http_upstream_next方法决定是否重新发起连接；
         * 并return从当前函数返回；
         */
        if (n == NGX_ERROR || n == 0) {
            ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
            return;
        }

        /* 若返回值 n 大于 0，表示已经接收到响应头部 */
        u->buffer.last += n;

#if 0
        u->valid_header_in = 0;

        u->peer.cached = 0;
#endif

        /*
         * 调用ngx_http_upstream_t结构体中process_header方法开始解析响应头部；
         * 并根据该方法返回值进行不同的判断；
         */
        rc = u->process_header(r);

        /*
         * 若返回值 rc = NGX_AGAIN，表示接收到的响应头部不完整，
         * 需等待下次读事件被触发时继续接收响应头部；
         * continue继续接收响应；
         */
        if (rc == NGX_AGAIN) {

            if (u->buffer.last == u->buffer.end) {
                ngx_log_error(NGX_LOG_ERR, c->log, 0,
                              "upstream sent too big header");
```

```
            ngx_http_upstream_next(r, u,
                                   NGX_HTTP_UPSTREAM_FT_INVALID_HEADER);
            return;
        }

        continue;
    }

    break;
}

/*
 * 若返回值 rc = NGX_HTTP_UPSTREAM_INVALID_HEADER，
 * 则表示接收到的响应头部是非法的，
 * 调用ngx_http_upstream_next方法决定是否重新发起连接；
 * 并return从当前函数返回；
 */
if (rc == NGX_HTTP_UPSTREAM_INVALID_HEADER) {
    ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_INVALID_HEADER);
    return;
}

/*
 * 若返回值 rc = NGX_ERROR，表示出错，
 * 则调用ngx_http_upstream_finalize_request方法结束该请求；
 * 并return从当前函数返回；
 */
if (rc == NGX_ERROR) {
    ngx_http_upstream_finalize_request(r, u,
                                       NGX_HTTP_INTERNAL_SERVER_ERROR);
    return;
}

/* rc == NGX_OK */

/*
 * 若返回值 rc = NGX_OK，表示成功解析到完整的响应头部；*/
if (u->headers_in.status_n >= NGX_HTTP_SPECIAL_RESPONSE) {

    if (ngx_http_upstream_test_next(r, u) == NGX_OK) {
        return;
    }

    if (ngx_http_upstream_intercept_errors(r, u) == NGX_OK) {
        return;
    }
}

/* 调用ngx_http_upstream_process_headers方法处理已解析处理的响应头部 */
if (ngx_http_upstream_process_headers(r, u) != NGX_OK) {
    return;
}

/*
 * 检查ngx_http_request_t结构体的成员变量keepalive是否为1，keepalive标志位表示是否转发响应给下游服务器
```

```
    * 检查ngx_http_request_t 结构体的subrequest_in_memory成员决定是否转发响应给下游服务器；
    * 若该标志位为0，则需调用ngx_http_upstream_send_response方法转发响应给下游服务器；
    * 并return从当前函数返回；
    */
    if (!r->subrequest_in_memory) {
        ngx_http_upstream_send_response(r, u);
        return;
    }

    /* 若不需要转发响应，则调用ngx_http_upstream_t中的input_filter方法处理响应包体 */
    /* subrequest content in memory */

    /*
    * 若HTTP模块没有定义ngx_http_upstream_t中的input_filter处理方法；
    * 则使用upstream机制默认方法ngx_http_upstream_non_buffered_filter；
    *
    * 若HTTP模块实现了input_filter方法，则不使用upstream默认的方法；
    */
    if (u->input_filter == NULL) {
        u->input_filter_init = ngx_http_upstream_non_buffered_filter_init;
        u->input_filter = ngx_http_upstream_non_buffered_filter;
        u->input_filter_ctx = r;
    }

    /*
    * 调用input_filter_init方法为处理包体做初始化工作；
    */
    if (u->input_filter_init(u->input_filter_ctx) == NGX_ERROR) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    /*
    * 检查接收缓冲区是否有剩余的响应数据；
    * 因为响应头部已经解析完毕，若接收缓冲区还有未被解析的剩余数据，
    * 则该数据就是响应包体；
    */
    n = u->buffer.last - u->buffer.pos;

    /*
    * 若接收缓冲区有剩余的响应包体，调用input_filter方法开始处理已接收到响应包体；
    */
    if (n) {
        u->buffer.last = u->buffer.pos;

        u->state->response_length += n;

        /* 调用input_filter方法处理响应包体 */
        if (u->input_filter(u->input_filter_ctx, n) == NGX_ERROR) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
    }

    if (u->length == 0) {
```

```
        ngx_http_upstream_finalize_request(r, u, 0);
        return;
    }

    /* 设置upstream机制的读事件回调方法read_event_handler为ngx_http_upstream_process_body_in_me
mory */
    u->read_event_handler = ngx_http_upstream_process_body_in_memory;

    /* 调用ngx_http_upstream_process_body_in_memory方法开始处理响应包体 */
    ngx_http_upstream_process_body_in_memory(r, u);
}
```

接收响应包体

接收并解析响应包体由 ngx_http_upstream_process_body_in_memory 方法实现；

ngx_http_upstream_process_body_in_memory 方法的执行流程如下所示：

- 检查上游连接上读事件是否超时，若标志位 timedout 为 1，则表示已经超时，此时调用 ngx_http_upstream_finalize_request 方法结束请求，并 return 从当前函数返回；
- 检查接收缓冲区 buffer 是否还有剩余的内存空间，若没有剩余的内存空间，则调用 ngx_http_upstream_finalize_request 方法结束请求，并 return 从当前函数返回；若有剩余的内存空间则调用 recv 方法开始接收响应包体；
- 若返回值 n = NGX_AGAIN，表示等待下一次触发读事件再接收响应包体，调用 ngx_handle_read_event 方法将读事件注册到 epoll 事件机制中，同时将读事件添加到定时器机制中；
- 若返回值 n = 0 或 n = NGX_ERROR，则调用 ngx_http_upstream_finalize_request 方法结束请求，并 return 从当前函数返回；
- 若返回值 n 大于 0，则表示成功接收到响应包体，调用 input_filter 方法开始处理响应包体，检查读事件的 ready 标志位；
- 若标志位 ready 为 1，表示仍有可读的响应包体数据，因此回到步骤 2 继续调用 recv 方法读取响应包体，直到读取完毕；
- 若标志位 ready 为 0，则调用 ngx_handle_read_event 方法将读事件注册到epoll事件机制中，同时调用 ngx_add_timer 方法将读事件添加到定时器机制中；

```
/* 接收并解析响应包体 */
static void
ngx_http_upstream_process_body_in_memory(ngx_http_request_t *r,
    ngx_http_upstream_t *u)
{
    size_t            size;
    ssize_t           n;
    ngx_buf_t        *b;
```

```
    ngx_buf_t         b;
    ngx_event_t       *rev;
    ngx_connection_t  *c;

    c = u->peer.connection;
    rev = c->read;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process body on memory");

    /*
     * 检查读事件标志位timedout是否超时，若该标志位为1，表示响应已经超时；
     * 则调用ngx_http_upstream_finalize_request方法结束请求；
     * 并return从当前函数返回；
     */
    if (rev->timedout) {
        ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");
        ngx_http_upstream_finalize_request(r, u, NGX_HTTP_GATEWAY_TIME_OUT);
        return;
    }

    b = &u->buffer;

    for ( ;; ) {

        /* 检查当前接收缓冲区是否剩余的内存空间 */
        size = b->end - b->last;

        /*
         * 若接收缓冲区不存在空闲的内存空间，
         * 则调用ngx_http_upstream_finalize_request方法结束请求；
         * 并return从当前函数返回；
         */
        if (size == 0) {
            ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                          "upstream buffer is too small to read response");
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }

        /*
         * 若接收缓冲区有可用的内存空间，
         * 则调用recv方法开始接收响应包体；
         */
        n = c->recv(c, b->last, size);

        /*
         * 若返回值 n = NGX_AGAIN，表示等待下一次触发读事件再接收响应包体；
         */
        if (n == NGX_AGAIN) {
            break;
        }

        /*
         * 若返回值n = 0(表示上游服务器主动关闭连接)，或n = NGX_ERROR(表示出错)；
```

```
            * 则调用ngx_http_upstream_finalize_request方法结束请求；
            * 并return从当前函数返回；
            */
            if (n == 0 || n == NGX_ERROR) {
                ngx_http_upstream_finalize_request(r, u, n);
                return;
            }

            /* 若返回值 n 大于0，表示成功读取到响应包体 */
            u->state->response_length += n;

            /* 调用input_filter方法处理本次接收到的响应包体 */
            if (u->input_filter(u->input_filter_ctx, n) == NGX_ERROR) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                return;
            }

            /* 检查读事件的ready标志位，若为1，继续读取响应包体 */
            if (!rev->ready) {
                break;
            }
        }

        if (u->length == 0) {
            ngx_http_upstream_finalize_request(r, u, 0);
            return;
        }

        /*
         * 若读事件的ready标志位为0，表示读事件未准备就绪，
         * 则将读事件注册到epoll事件机制中，添加到定时器机制中；
         * 读事件的回调方法不改变，即依旧为ngx_http_upstream_process_body_in_memory；
         */
        if (ngx_handle_read_event(rev, 0) != NGX_OK) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }

        if (rev->active) {
            ngx_add_timer(rev, u->conf->read_timeout);

        } else if (rev->timer_set) {
            ngx_del_timer(rev);
        }
    }
```

转发响应

下面看下 upstream 处理上游响应包体的三种方式：

1. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 1 时，upstream 不转发响应包体到下游，并由HTTP 模块实现的input_filter() 方法处理包体；

2. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 0 时，且 ngx_http_upstream_conf_t 配置结构体中的成员buffering 标志位为 1 时，upstream 将开启更多的内存和磁盘文件用于缓存上游的响应包体（此时，上游网速更快），并转发响应包体；
3. 当请求结构体 ngx_http_request_t 中的成员subrequest_in_memory 标志位为 0 时，且 ngx_http_upstream_conf_t 配置结构体中的成员buffering 标志位为 0 时，upstream 将使用固定大小的缓冲区来转发响应包体；

转发响应由函数 ngx_http_upstream_send_response 实现，该函数的执行流程如下：

- 调用 ngx_http_send_header 方法转发响应头部，并将 ngx_http_upstream_t 结构体中的 header_sent 标志位设置为 1，表示已经转发响应头部；
- 若临时文件还保存着请求包体，则需调用 ngx_pool_run_cleanup_filter 方法清理临时文件；
- 检查标志位 buffering，若该标志位为 1，表示需要开启文件缓存，若该标志位为 0，则不需要开启文件缓存，只需要以固定的内存块大小转发响应包体即可；
- 若标志位 buffering 为0；
- 则检查 HTTP 模块是否实现了自己的 input_filter 方法，若没有则使用 upstream 机制默认的方法 ngx_http_upstream_non_buffered_filter；
- 设置 ngx_http_upstream_t 结构体中读事件 read_event_handler 的回调方法为 ngx_http_upstream_process_non_buffered_upstream，当接收上游响应时，会通过 ngx_http_upstream_handler 方法最终调用 ngx_http_upstream_process_non_buffered_uptream 来接收响应；
- 设置 ngx_http_upstream_t 结构体中写事件 write_event_handler 的回调方法为 ngx_http_upstream_process_non_buffered_downstream，当向下游发送数据时，会通过 ngx_http_handler 方法最终调用 ngx_http_upstream_process_non_buffered_downstream 方法来发送响应包体；
- 调用 input_filter_init 方法为 input_filter 方法处理响应包体做初始化工作；
- 检查接收缓冲区 buffer 在解析完响应头部之后，是否还有剩余的响应数据，若有表示预接收了响应包体：
- 若在解析响应头部区间，预接收了响应包体，则调用 input_filter 方法处理该部分预接收的响应包体，并调用 ngx_http_upstream_process_non_buffered_downstream 方法转发本次接收到的响应包体给下游服务器；
- 若在解析响应头部区间，没有接收响应包体，则首先清空接收缓冲区 buffer 以便复用来接收响应包体，检查上游连接上读事件是否准备就绪，若标志位 ready 为1，表示准备就绪，则调用 ngx_http_upstream_process_non_buffered_upstream 方法接收上游响应包体；若标志位 ready 为 0，则 return 从当前函数返回；

- 若标志位 buffering 为1；
- 初始化 ngx_http_upstream_t 结构体中的 ngx_event_pipe_t pipe 成员；
- 调用 input_filter_init 方法为 input_filter 方法处理响应包体做初始化工作；
- 设置上游连接上的读事件 read_event_handler 的回调方法为 ngx_http_upstream_process_upstream；
- 设置上游连接上的写事件 write_event_handler 的回调方法为 ngx_http_upstream_process_downstream；
- 调用 ngx_http_upstream_proess_upstream 方法处理由上游服务器发来的响应包体；

```
/* 转发响应包体 */
static void
ngx_http_upstream_send_response(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
    int                       tcp_nodelay;
    ssize_t                   n;
    ngx_int_t                 rc;
    ngx_event_pipe_t          *p;
    ngx_connection_t          *c;
    ngx_http_core_loc_conf_t  *clcf;

    /* 调用ngx_http_send_hander方法向下游发送响应头部 */
    rc = ngx_http_send_header(r);

    if (rc == NGX_ERROR || rc > NGX_OK || r->post_action) {
        ngx_http_upstream_finalize_request(r, u, rc);
        return;
    }

    /* 将标志位header_sent设置为1 */
    u->header_sent = 1;

    if (u->upgrade) {
        ngx_http_upstream_upgrade(r, u);
        return;
    }

    /* 获取Nginx与下游之间的TCP连接 */
    c = r->connection;

    if (r->header_only) {

        if (u->cacheable || u->store) {

            if (ngx_shutdown_socket(c->fd, NGX_WRITE_SHUTDOWN) == -1) {
                ngx_connection_error(c, ngx_socket_errno,
                                     ngx_shutdown_socket_n " failed");
            }
```

```
            r->read_event_handler = ngx_http_request_empty_handler;
            r->write_event_handler = ngx_http_request_empty_handler;
            c->error = 1;

        } else {
            ngx_http_upstream_finalize_request(r, u, rc);
            return;
        }
    }

    /* 若临时文件保存着请求包体，则调用ngx_pool_run_cleanup_file方法清理临时文件的请求包体 */
    if (r->request_body && r->request_body->temp_file) {
        ngx_pool_run_cleanup_file(r->pool, r->request_body->temp_file->file.fd);
        r->request_body->temp_file->file.fd = NGX_INVALID_FILE;
    }

    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    /*
     * 若标志位buffering为0，转发响应时以下游服务器网速优先；
     * 即只需分配固定的内存块大小来接收来自上游服务器的响应并转发，
     * 当该内存块已满，则暂停接收来自上游服务器的响应数据，
     * 等待把内存块的响应数据转发给下游服务器后有剩余内存空间再继续接收响应；
     */
    if (!u->buffering) {

        /*
         * 若HTTP模块没有实现input_filter方法，
         * 则采用upstream机制默认的方法ngx_http_upstream_non_buffered_filter；
         */
        if (u->input_filter == NULL) {
            u->input_filter_init = ngx_http_upstream_non_buffered_filter_init;
            u->input_filter = ngx_http_upstream_non_buffered_filter;
            u->input_filter_ctx = r;
        }

        /*
         * 设置ngx_http_upstream_t结构体中读事件的回调方法为ngx_http_upstream_non_buffered_upstrea
m，(即读取上游响应的方法)；
         * 设置当前请求ngx_http_request_t结构体中写事件的回调方法为ngx_http_upstream_process_non_bu
ffered_downstream，(即转发响应到下游的方法)；
         */
        u->read_event_handler = ngx_http_upstream_process_non_buffered_upstream;
        r->write_event_handler =
                            ngx_http_upstream_process_non_buffered_downstream;

        r->limit_rate = 0;

        /* 调用input_filter_init为input_filter方法处理响应包体做初始化工作 */
        if (u->input_filter_init(u->input_filter_ctx) == NGX_ERROR) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
```

```
        if (clcf->tcp_nodelay && c->tcp_nodelay == NGX_TCP_NODELAY_UNSET) {
            ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "tcp_nodelay");

            tcp_nodelay = 1;

            if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
                           (const void *) &tcp_nodelay, sizeof(int)) == -1)
            {
                ngx_connection_error(c, ngx_socket_errno,
                                     "setsockopt(TCP_NODELAY) failed");
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                return;
            }

            c->tcp_nodelay = NGX_TCP_NODELAY_SET;
        }

        /* 检查解析完响应头部后接收缓冲区buffer是否已接收了响应包体 */
        n = u->buffer.last - u->buffer.pos;

        /* 若接收缓冲区已经接收了响应包体 */
        if (n) {
            u->buffer.last = u->buffer.pos;

            u->state->response_length += n;

            /* 调用input_filter方法开始处理响应包体 */
            if (u->input_filter(u->input_filter_ctx, n) == NGX_ERROR) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                return;
            }

            /* 调用该方法把本次接收到的响应包体转发给下游服务器 */
            ngx_http_upstream_process_non_buffered_downstream(r);

        } else {
            /* 若接收缓冲区中没有响应包体，则将其清空，即复用这个缓冲区 */
            u->buffer.pos = u->buffer.start;
            u->buffer.last = u->buffer.start;

            if (ngx_http_send_special(r, NGX_HTTP_FLUSH) == NGX_ERROR) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                return;
            }

            /*
             * 若当前连接上读事件已准备就绪，
             * 则调用ngx_http_upstream_process_non_buffered_upstream方法接收响应包体并处理；
             */
            if (u->peer.connection->read->ready || u->length == 0) {
                ngx_http_upstream_process_non_buffered_upstream(r, u);
            }
        }

        return;
```

```
        return;
    }

    /*
     * 若ngx_http_upstream_t结构体的buffering标志位为1，则转发响应包体时以上游网速优先；
     * 即分配更多的内存和缓存，即一直接收来自上游服务器的响应，把来自上游服务器的响应保存的内存或缓
存中；
     */
    /* TODO: preallocate event_pipe bufs, look "Content-Length" */

#if (NGX_HTTP_CACHE)
    ...
    ...
#endif

    /* 初始化ngx_event_pipe_t结构体 p */
    p = u->pipe;

    p->output_filter = (ngx_event_pipe_output_filter_pt) ngx_http_output_filter;
    p->output_ctx = r;
    p->tag = u->output.tag;
    p->bufs = u->conf->bufs;
    p->busy_size = u->conf->busy_buffers_size;
    p->upstream = u->peer.connection;
    p->downstream = c;
    p->pool = r->pool;
    p->log = c->log;

    p->cacheable = u->cacheable || u->store;

    p->temp_file = ngx_pcalloc(r->pool, sizeof(ngx_temp_file_t));
    if (p->temp_file == NULL) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    p->temp_file->file.fd = NGX_INVALID_FILE;
    p->temp_file->file.log = c->log;
    p->temp_file->path = u->conf->temp_path;
    p->temp_file->pool = r->pool;

    if (p->cacheable) {
        p->temp_file->persistent = 1;

    } else {
        p->temp_file->log_level = NGX_LOG_WARN;
        p->temp_file->warn = "an upstream response is buffered "
                             "to a temporary file";
    }

    p->max_temp_file_size = u->conf->max_temp_file_size;
    p->temp_file_write_size = u->conf->temp_file_write_size;

    /* 初始化预读链表缓冲区preread_bufs */
    p->preread_bufs = ngx_alloc_chain_link(r->pool);
```

```
    if (p->preread_bufs == NULL) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    p->preread_bufs->buf = &u->buffer;
    p->preread_bufs->next = NULL;
    u->buffer.recycled = 1;

    p->preread_size = u->buffer.last - u->buffer.pos;

    if (u->cacheable) {

        p->buf_to_file = ngx_calloc_buf(r->pool);
        if (p->buf_to_file == NULL) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }

        p->buf_to_file->start = u->buffer.start;
        p->buf_to_file->pos = u->buffer.start;
        p->buf_to_file->last = u->buffer.pos;
        p->buf_to_file->temporary = 1;
    }

    if (ngx_event_flags & NGX_USE_AIO_EVENT) {
        /* the posted aio operation may corrupt a shadow buffer */
        p->single_buf = 1;
    }

    /* TODO: p->free_bufs = 0 if use ngx_create_chain_of_bufs() */
    p->free_bufs = 1;

    /*
     * event_pipe would do u->buffer.last += p->preread_size
     * as though these bytes were read
     */
    u->buffer.last = u->buffer.pos;

    if (u->conf->cyclic_temp_file) {

        /*
         * we need to disable the use of sendfile() if we use cyclic temp file
         * because the writing a new data may interfere with sendfile()
         * that uses the same kernel file pages (at least on FreeBSD)
         */

        p->cyclic_temp_file = 1;
        c->sendfile = 0;

    } else {
        p->cyclic_temp_file = 0;
    }

    p->read_timeout = u->conf->read_timeout;
```

```
    p->send_timeout = clcf->send_timeout;
    p->send_lowat = clcf->send_lowat;

    p->length = -1;

    /* 调用input_filter_init方法进行初始化工作 */
    if (u->input_filter_init
        && u->input_filter_init(p->input_ctx) != NGX_OK)
    {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    /* 设置上游读事件的方法 */
    u->read_event_handler = ngx_http_upstream_process_upstream;
    /* 设置下游写事件的方法 */
    r->write_event_handler = ngx_http_upstream_process_downstream;

    /* 处理上游响应包体 */
    ngx_http_upstream_process_upstream(r, u);
}
```

当以下游网速优先转发响应包体给下游时，由函数 ngx_http_upstream_process_non_buffered_downstrean 实现，该函数的执行流程如下所示：

- 检查下游连接上写事件是否超时，若标志位 timedout 为1，则表示超时，此时调用 ngx_http_upstream_finalize_request 方法接收请求，并 return 从当前函数返回；
- 调用 ngx_http_upstream_process_non_bufferd_request 方法向下游服务器发送响应包体，此时第二个参数为 1；

```
/* buffering 标志位为0时，转发响应包体给下游服务器 */
static void
ngx_http_upstream_process_non_buffered_downstream(ngx_http_request_t *r)
{
    ngx_event_t          *wev;
    ngx_connection_t     *c;
    ngx_http_upstream_t  *u;

    /* 获取Nginx与下游服务器之间的TCP连接 */
    c = r->connection;
    /* 获取ngx_http_upstream_t结构体 */
    u = r->upstream;
    /* 获取当前连接的写事件 */
    wev = c->write;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process non buffered downstream");

    c->log->action = "sending to client";

    /* 检查写事件是否超时，若超时则结束请求 */
    if (wev->timedout) {
        c->timedout = 1;
        ngx_connection_error(c, NGX_ETIMEDOUT, "client timed out");
        ngx_http_upstream_finalize_request(r, u, NGX_HTTP_REQUEST_TIME_OUT);
        return;
    }

    /* 若不超时，以固定内存块方式转发响应包体给下游服务器 */
    ngx_http_upstream_process_non_buffered_request(r, 1);
}
```

由于 buffering 标志位为0时，没有开启文件缓存，只有固定大小的内存块作为接收响应缓冲区，当上游的响应包体比较大时，此时，接收缓冲区内存并不能够满足一次性接收完所有响应包体， 因此，在接收缓冲区已满时，会阻塞接收响应包体，并先把已经收到的响应包体转发给下游服务器。所有在转发响应包体时，有可能会接收上游响应包体。此过程由 ngx_http_upstream_process_non_buffered_upstream 方法实现；

ngx_http_upstream_process_non_buffered_upstream 方法执行流程如下：

- 检查上游连接上的读事件是否超时，若标志位 timedout 为 1，表示已经超时，此时调用 ngx_http_upstream_finalize_request 方法结束请求，并 return 从当前函数返回；
- 调用 ngx_http_upstream_process_non_buffered_request 方法接收上游响应包体，此时第二个参数为 0；

```
/* 接收上游响应包体(buffering为0的情况) */
static void
ngx_http_upstream_process_non_buffered_upstream(ngx_http_request_t *r,
    ngx_http_upstream_t *u)
{
    ngx_connection_t  *c;

    /* 获取Nginx与上游服务器之间的TCP连接 */
    c = u->peer.connection;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process non buffered upstream");

    c->log->action = "reading upstream";

    /* 判断读事件是否超时，若超时则结束当前请求 */
    if (c->read->timedout) {
        ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");
        ngx_http_upstream_finalize_request(r, u, NGX_HTTP_GATEWAY_TIME_OUT);
        return;
    }

    /*
     * 若不超时，则以固定内存块方式转发响应包体给下游服务器，
     * 注意：转发的过程中，会接收来自上游服务器的响应包体；
     */
    ngx_http_upstream_process_non_buffered_request(r, 0);
}
```

在上面函数中向下游服务器转发响应包体过程中，最终会调用 ngx_http_upstream_process_non_buffered_request 方法来实现，而且转发响应包体给下游服务器时，同时会接收来自上游的响应包体，接收上游响应包体最终也会调用该函数，只是调用的时候第二个参数指定不同的值；

ngx_http_upstream_process_non_buffered_request 方法执行流程如下所示：

- *步骤1*：若 do_write 参数的值为 0，表示需要接收来自上游服务器的响应包体，则直接跳到*步骤3*开始执行；
- *步骤2*：若 do_write 参数的值为 1，则开始向下游转发响应包体；
- 检查 ngx_http_upstream_t 结构体中的 out_bufs 链表 或 busy_bufs 链表是否有数据：
- 若 out_bufs 或 busy_bufs 链表缓冲区中有响应包体，则调用 ngx_http_output_filter 方法向下游发送响应包体，并调用 ngx_chain_update_chains 方法更新 ngx_http_upstream_t 结构体中的 free_bufs、busy_bufs、out_bufs 链表缓冲区；
- 若 out_bufs 和 busy_bufs 链表缓冲区中都没有数据，则清空接收缓冲区 buffer 以便再次接收来自上游服务器的响应包体；

- *步骤3*：计算接收缓冲区 buffer 剩余的内存空间 size，若有剩余的内存空间，且此时上游连接上有可读的响应包体(即读事件的 ready 标志位为 1)，则调用 recv 方法读取上游响应包体，并返回 n；
- 若返回值 n 大于 0，表示已经接收到上游响应包体，则调用 input_filter 方法处理响应包体，并设置 do_write 标志位为 1，表示已经接收到响应包体，此时可转发给下游服务器，又回到*步骤2*继续执行；
- 若返回值 n = NGX_AGAIN，表示需要等待下一次读事件的发生以便继续接收上游响应包体，则直接跳至*步骤5*开始执行；
- *步骤4*：若接收缓冲区 buffer 没有剩余内存空间 或 上游连接上读事件未准备就绪，则从 *步骤5*开始执行；
- *步骤5*：调用 ngx_add_timer 方法将下游连接上写事件添加到定时器机制中，调用 ngx_handle_write_event 方法将下游连接上写事件注册到 epoll 事件机制中；
- *步骤6*：调用 ngx_handle_read_event 方法将上游连接上读事件注册到 epoll 事件机制中，调用 ngx_add_timer 方法将上游连接上读事件添加到定时器机制中；

```
/* 以固定内存块方式转发响应包体给下游服务器 */

/*
 * 第二个参数表示本次是否需要向下游发送响应；若为0时，需要接收来自上游服务器的响应，也需要转发响应给下游；
 * 若为1，只负责转发响应给下游服务器；
 */
static void
ngx_http_upstream_process_non_buffered_request(ngx_http_request_t *r,
    ngx_uint_t do_write)
{
    size_t                     size;
    ssize_t                    n;
    ngx_buf_t                  *b;
    ngx_int_t                  rc;
    ngx_connection_t           *downstream, *upstream;
    ngx_http_upstream_t        *u;
    ngx_http_core_loc_conf_t  *clcf;

    /* 获取ngx_http_upstream_t结构体 */
    u = r->upstream;
    /* 获取Nginx与下游服务器之间的TCP连接 */
    downstream = r->connection;
    /* 获取Nginx与上游服务器之间的TCP连接 */
    upstream = u->peer.connection;

    /* 获取ngx_hhtp_upstream_t结构体的接收缓冲区buffer */
    b = &u->buffer;

    /*
     * 获取do_write的值，该值决定是否还要接收来自上游服务器的响应；
     * 其中length表示还需要接收的上游响应包体长度；
     */
```

```
    do_write = do_write || u->length == 0;

    for ( ;; ) {

        if (do_write) {/* 若do_write为1，则开始向下游服务器转发响应包体 */

            /*
             * 检查是否有响应包体需要转发给下游服务器；
             * 其中out_bufs表示本次需要转发给下游服务器的响应包体；
             * busy_bufs表示上一次向下游服务器转发响应包体时没有转发完的响应包体内存；
             * 即若一次性转发不完所有的响应包体，则会保存在busy_bufs链表缓冲区中，
             * 这里的保存只是将busy_bufs指向未发送完毕的响应数据；
             */
            if (u->out_bufs || u->busy_bufs) {
                /* 调用ngx_http_output_filter方法将响应包体发送给下游服务器 */
                rc = ngx_http_output_filter(r, u->out_bufs);

                /* 若返回值 rc = NGX_ERROR，则结束请求 */
                if (rc == NGX_ERROR) {
                    ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                    return;
                }

                /*
                 * 调用ngx_chain_update_chains方法更新free_bufs、busy_bufs、out_bufs链表；
                 * 即清空out_bufs链表，把out_bufs链表中已发送完的ngx_buf_t缓冲区清空，并将其添加到free_b
ufs链表中；
                 * 把out_bufs链表中未发送完的ngx_buf_t缓冲区添加到busy_bufs链表中；
                 */
                ngx_chain_update_chains(r->pool, &u->free_bufs, &u->busy_bufs,
                                        &u->out_bufs, u->output.tag);
            }

            /*
             * busy_bufs为空，表示所有响应包体已经转发到下游服务器，
             * 此时清空接收缓冲区buffer以便再次接收来自上游服务器的响应包体；
             */
            if (u->busy_bufs == NULL) {

                if (u->length == 0
                    || (upstream->read->eof && u->length == -1))
                {
                    ngx_http_upstream_finalize_request(r, u, 0);
                    return;
                }

                if (upstream->read->eof) {
                    ngx_log_error(NGX_LOG_ERR, upstream->log, 0,
                                  "upstream prematurely closed connection");

                    ngx_http_upstream_finalize_request(r, u,
                                                       NGX_HTTP_BAD_GATEWAY);
                    return;
                }
```

```
            if (upstream->read->error) {
                ngx_http_upstream_finalize_request(r, u,
                                                   NGX_HTTP_BAD_GATEWAY);
                return;
            }

            b->pos = b->start;
            b->last = b->start;
        }
    }

    /* 计算接收缓冲区buffer剩余可用的内存空间 */
    size = b->end - b->last;

    /*
     * 若接收缓冲区buffer有剩余的可用空间，
     * 且此时读事件可读，即可读取来自上游服务器的响应包体；
     * 则调用recv方法开始接收来自上游服务器的响应包体，并保存在接收缓冲区buffer中；
     */
    if (size && upstream->read->ready) {

        n = upstream->recv(upstream, b->last, size);

        /* 若返回值 n = NGX_AGAIN，则等待下一次可读事件发生继续接收响应 */
        if (n == NGX_AGAIN) {
            break;
        }

        /*
         * 若返回值 n 大于0，表示接收到响应包体，
         * 则调用input_filter方法处理响应包体；
         * 并把do_write设置为1；
         */
        if (n > 0) {
            u->state->response_length += n;

            if (u->input_filter(u->input_filter_ctx, n) == NGX_ERROR) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
                return;
            }
        }

        do_write = 1;

        continue;
    }

    break;
}

clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

/* 调用ngx_handle_write_event方法将Nginx与下游之间的连接上的写事件注册的epoll事件机制中 */
if (downstream->data == r) {
```

```
        if (ngx_handle_write_event(downstream->write, clcf->send_lowat)
            != NGX_OK)
        {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
    }

    /* 调用ngx_add_timer方法将Nginx与下游之间的连接上的写事件添加到定时器事件机制中 */
    if (downstream->write->active && !downstream->write->ready) {
        ngx_add_timer(downstream->write, clcf->send_timeout);

    } else if (downstream->write->timer_set) {
        ngx_del_timer(downstream->write);
    }

    /* 调用ngx_handle_read_event方法将Nginx与上游之间的连接上的读事件注册的epoll事件机制中 */
    if (ngx_handle_read_event(upstream->read, 0) != NGX_OK) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    /* 调用ngx_add_timer方法将Nginx与上游之间的连接上的读事件添加到定时器事件机制中 */
    if (upstream->read->active && !upstream->read->ready) {
        ngx_add_timer(upstream->read, u->conf->read_timeout);

    } else if (upstream->read->timer_set) {
        ngx_del_timer(upstream->read);
    }
}
```

```
/* upstream机制默认的input_filter方法 */
static ngx_int_t
ngx_http_upstream_non_buffered_filter(void *data, ssize_t bytes)
{
    /*
     * data参数是ngx_http_upstream_t结构体中的input_filter_ctx，
     * 当HTTP模块没有实现input_filter方法时，
     * input_filter_ctx指向ngx_http_request_t结构体；
     */
    ngx_http_request_t  *r = data;

    ngx_buf_t            *b;
    ngx_chain_t          *cl, **ll;
    ngx_http_upstream_t  *u;

    u = r->upstream;

    /* 找到out_bufs链表的最后一个缓冲区，并由ll指向该缓冲区 */
    for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
        ll = &cl->next;
    }

    /* 从free_bufs空闲链表缓冲区中获取一个ngx_buf_t结构体给cl */
    cl = ngx_chain_get_free_buf(r->pool, &u->free_bufs);
    if (cl == NULL) {
        return NGX_ERROR;
    }

    /* 将新分配的ngx_buf_t缓冲区添加到out_bufs链表的尾端 */
    *ll = cl;

    cl->buf->flush = 1;
    cl->buf->memory = 1;

    /* buffer是保存来自上游服务器的响应包体 */
    b = &u->buffer;

    /* 将响应包体数据保存在cl缓冲区中 */
    cl->buf->pos = b->last;
    b->last += bytes;
    cl->buf->last = b->last;
    cl->buf->tag = u->output.tag;

    if (u->length == -1) {
        return NGX_OK;
    }

    /* 更新length长度，表示需要接收的包体长度减少bytes字节 */
    u->length -= bytes;

    return NGX_OK;
}
```

当 buffering 标志位为 1 转发响应包体给下游时，由函数 ngx_http_upstream_process_downstream 实现。

ngx_http_upstream_process_downstream 方法的执行流程如下所示：

- 若下游连接上写事件的 timedout 标志位为 1，表示写事件已经超时；
- 若下游连接上写事件的 delayed 标志位为 1；
- 若下游连接上写事件 ready 标志位为 1，表示写事件已经准备就绪，则调用 ngx_event_pipe 方法转发响应包体；
- 若下游连接上写事件 ready 标志位为 0，表示写事件未准备就绪，则调用 ngx_add_timer 方法将写事件添加到定时器机制中，调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中，并 return 从当前函数返回；
- 若下游连接上写事件的 delayed 标志位为 0，则该连接已经出错，设置 ngx_event_pipe_t 结构体中 downstream_error标志位为 1，设置 ngx_connection_t 结构体中 timedout 标志位为 1，并调用 ngx_connection_error 方法；
- 若下游连接上写事件的 timedout 标志位为 0，表示写事件不超时；
- 若下游连接上写事件的 delayed 标志位为 1，则调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中，并 return 从当前函数返回；
- 若下游连接上写事件的 delayed 标志位为 0，则调用 ngx_event_pipe 方法转发响应包体；
- 最终调用 ngx_http_upstream_process_request 方法；

```
static void
ngx_http_upstream_process_downstream(ngx_http_request_t *r)
{
    ngx_event_t          *wev;
    ngx_connection_t     *c;
    ngx_event_pipe_t     *p;
    ngx_http_upstream_t  *u;

    /* 获取 Nginx 与下游服务器之间的连接 */
    c = r->connection;
    /* 获取 Nginx 与上游服务器之间的连接 */
    u = r->upstream;
    /* 获取 ngx_event_pipe_t 结构体 */
    p = u->pipe;
    /* 获取下游连接上的写事件 */
    wev = c->write;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process downstream");

    c->log->action = "sending to client";
```

```
/* 检查下游连接上写事件是否超时，若标志位 timedout 为 1，表示超时 */
if (wev->timedout) {

    /* 若下游连接上写事件的delayed 标志位为 1 */
    if (wev->delayed) {

        wev->timedout = 0;
        wev->delayed = 0;

        /*
         * 检查写事件是否准备就绪，若 ready 标志位为 0，
         * 表示未准备就绪，则调用 ngx_add_timer 方法将写事件添加到定时器机制中；
         * 调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中；
         * 并 return 从当前函数返回；
         */
        if (!wev->ready) {
            ngx_add_timer(wev, p->send_timeout);

            if (ngx_handle_write_event(wev, p->send_lowat) != NGX_OK) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            }

            return;
        }

        /*
         * 若写事件已经准备就绪，即ready 标志位为 1；
         * 则调用 ngx_event_pipe 方法将响应包体转发给下游服务器；
         * 并 return 从当前函数返回；
         */
        if (ngx_event_pipe(p, wev->write) == NGX_ABORT) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }

    } else {
        /* 若写事件的delayed标志位为 0，则设置downstream_error标志位为 1,，表示连接出错 */
        p->downstream_error = 1;
        c->timedout = 1;
        ngx_connection_error(c, NGX_ETIMEDOUT, "client timed out");
    }

} else {/* 若下游连接上写事件不超时，即timedout 标志位为 0 */

    /*
     * 检查写事件 delayed 标志位，若该标志位为 1；
     * 则调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中；
     * 并 return 从当前函数返回；
     */
    if (wev->delayed) {

        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                       "http downstream delayed");
```

```
            if (ngx_handle_write_event(wev, p->send_lowat) != NGX_OK) {
                ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            }

            return;
        }

        /* 若写事件的delayed 标志位为 0，则调用 ngx_event_pipe 方法转发响应 */
        if (ngx_event_pipe(p, 1) == NGX_ABORT) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
    }
    /* 最终调用该函数 */
    ngx_http_upstream_process_request(r);
}
```

ngx_http_upstream_process_upstream 方法执行流程如下所示：

- 若上游连接上读事件的 timedout 标志位为 1，表示读事件已经超时，则设置 upstream_error 为 1，调用 ngx_connection_error 方法接收当前函数；
- 若上游连接上读事件的 timedout 标志位为 0，则调用 ngx_event_pipe 方法接收上游响应包体；
- 最终调用 ngx_http_upstream_process_request 方法；

```
static void
ngx_http_upstream_process_upstream(ngx_http_request_t *r,
    ngx_http_upstream_t *u)
{
    ngx_connection_t  *c;

    c = u->peer.connection;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http upstream process upstream");

    c->log->action = "reading upstream";

    if (c->read->timedout) {
        u->pipe->upstream_error = 1;
        ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");

    } else {
        if (ngx_event_pipe(u->pipe, 0) == NGX_ABORT) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
    }

    ngx_http_upstream_process_request(r);
}
```

ngx_event_pipe 方法的执行流程如下所示：

- *步骤1*：若参数 do_write 为 1，表示向下游转发响应；
- 调用 ngx_event_pipe_write_to_downstream 方法向下游转发响应，并返回值为 rc；
- 若返回值 rc = NGX_ABORT，则 return NGX_ABORT 从当前函数返回；
- 若返回值 rc = NGX_BUSY，表示不需要往下执行，则 return NGX_OK 从当前函数返回；
- 若返回值 rc = NGX_OK，则直接跳至*步骤3*执行；
- *步骤2*：若参数 do_write 为 0，表示需要接收上游响应，直接跳至*步骤3*执行；
- *步骤3*：设置 ngx_event_pipe_t 结构体中的 read 标志位为 0，upstream_blocked 标志位为 0；
- *步骤4*：调用 ngx_event_pipe_read_upstream 方法读取上游响应；
- *步骤5*：检查 read 和 upstream_blocked 标志位，若 read 和 upstream_blocked 标志位都为 0，则跳至*步骤7*执行；
- *步骤6*：若 read 或 upstream_blocked 标志位为 1，表示需要向下游发送刚刚读取到的响应，则设置 do_write 标志为 1，跳至*步骤1*继续执行；
- *步骤7*：调用 ngx_add_timer 方法将上游连接上的读事件添加到定时器机制中，调用 ngx_handle_read_event 方法将读事件注册到 epoll 事件机制中；
- *步骤8*：调用 ngx_add_timer 方法将下游连接上的写事件添加到定时器机制中，调用 ngx_handle_write_event 方法将写事件注册到 epoll 事件机制中；
- *步骤9*：return NGX_OK 从当前函数返回；

```
/* 转发响应的ngx_event_pipe_t结构体 */
ngx_int_t
ngx_event_pipe(ngx_event_pipe_t *p, ngx_int_t do_write)
{
    u_int         flags;
    ngx_int_t     rc;
    ngx_event_t  *rev, *wev;

    for ( ;; ) {
        if (do_write) {/* 若 do_write标志位为1，表示向下游转发响应 */
            p->log->action = "sending to client";

            /* 调用ngx_event_pipe_write_to_downstream方法向下游转发响应 */
            rc = ngx_event_pipe_write_to_downstream(p);

            if (rc == NGX_ABORT) {
                return NGX_ABORT;
            }
```

```
            if (rc == NGX_BUSY) {
                return NGX_OK;
            }
        }

        /* 若do_write标志位为0，则接收上游响应 */
        p->read = 0;
        p->upstream_blocked = 0;

        p->log->action = "reading upstream";

        /* 调用ngx_event_pipe_read_upstream方法读取上游响应 */
        if (ngx_event_pipe_read_upstream(p) == NGX_ABORT) {
            return NGX_ABORT;
        }

        /*
         * 若标志位read和upstream_blocked为0，
         * 则没有可读的响应数据，break退出for循环；
         */
        if (!p->read && !p->upstream_blocked) {
            break;
        }

        /* 否则，设置do_write标志位为1继续进行for循环操作 */
        do_write = 1;
    }

    /*
     * 将上游读事件添加到定时器机制中，注册到epoll事件机制中；
     */
    if (p->upstream->fd != (ngx_socket_t) -1) {
        rev = p->upstream->read;

        flags = (rev->eof || rev->error) ? NGX_CLOSE_EVENT : 0;

        if (ngx_handle_read_event(rev, flags) != NGX_OK) {
            return NGX_ABORT;
        }

        if (rev->active && !rev->ready) {
            ngx_add_timer(rev, p->read_timeout);

        } else if (rev->timer_set) {
            ngx_del_timer(rev);
        }
    }

    /*
     * 将下游写事件添加到定时器机制中，注册到epoll事件机制中；
     */
    if (p->downstream->fd != (ngx_socket_t) -1
        && p->downstream->data == p->output_ctx)
    {
        wev = p->downstream->write;
```

```
        if (ngx_handle_write_event(wev, p->send_lowat) != NGX_OK) {
            return NGX_ABORT;
        }

        if (!wev->delayed) {
            if (wev->active && !wev->ready) {
                ngx_add_timer(wev, p->send_timeout);

            } else if (wev->timer_set) {
                ngx_del_timer(wev);
            }
        }
    }

    return NGX_OK;
}
```

ngx_event_pipe_read_upstream 方法的执行流程如下所示：

- *步骤1*：检查 ngx_event_pipe_t 结构体中的 upstream_eof(若为 1 表示上游连接通信已经结束)、upstream_error(若为 1，表示上游连接出错)、upstream_done(若为 1，表示上游连接已经关闭) 标志位，若其中一个标志位为 1，表示上游连接关闭，则 return NGX_OK 从当前函数返回；
- *步骤2*：进入 for 循环，再次检查以上三个标志位，若其中有一个为 1，则 break 退出for 循环，跳至*步骤8*执行；
- *步骤3*：若 preread_bufs 链表缓冲区为空，表示接收响应头部区间，没有预接收响应包体，且此时上游连接上读事件未准备就绪，即ready标志位为0，则 break 退出for 循环，跳至*步骤8*执行；
- *步骤4*：若 preread_bufs 链表缓冲区不为空，表示预接收了响应包体，则将 preread_bufs 链表缓冲区挂载到 chain 链表中，并计算预接收到响应包体的长度 n，若 n 大于 0，则设置 read 标志位为 1；
- *步骤5*：若 preread_bufs 链表缓冲区为空：
- 若 free_raw_bufs 不为空，则将该链表缓冲区挂载到chain链表中；
- 若 free_raw_bufs 为空：
- 若 allocated 小于 bufs.num，则调用 ngx_create_temp_buf 方法分配一个新的缓冲区 b，并将新分配的缓冲区挂载到 chain 链表中；
- 若 allocated 大于 bufs.num：
- 若 cacheable 标志位为 0，且下游连接上写事件已准备就绪，即写事件的 ready 标志位为 1，表示可以向下游发送响应包体，此时，设置upstream_blocked 标志位为 1，表示阻塞读取上游响应包体，因为没有缓冲区来接收上游响应包体，并break 退出for循环，跳至*步骤8*执行；
- 若 cacheable 标志位为 1，即开启了文件缓存，且此时临时文件长度未达到最大长度，则调用 ngx_event_pipe_write_chain_to_temp_file 方法将上游响应写入到临时文件中，以便使

free_raw_bufs 有空余缓冲区来继续接收上游响应，并将此 free_raw_bufs 链表缓冲区挂载到 chain 链表中；

- 若以上条件都不满足，则break 退出 for 循环，跳至 *步骤8*执行；
- *步骤6*：调用 recv_chain 方法接收上游响应包体，返回值为 n，并把接收到的上游响应包体缓冲区添加到 free_raw_bufs 链表的尾端；
- 若返回值 n = NGX_ERROR，表示上游连接出错，return NGX_ERROR 从当前函数返回；
- 若返回值 n = NGX_AGAIN，表示没有读取到上游响应包体，则 break 退出 for 循环，跳至 *步骤8*执行；
- 若返回值 n = 0，表示上游服务器主动关闭连接，则 break 退出 for 循环，跳至 *步骤8*执行；
- 若返回值 n 大于 0，则设置 read 标志位为 1，表示已经接收到上游响应包体；
- *步骤7*：开始处理已接收到的上游响应包体，遍历待处理缓冲区链表 chain 中的每一个 ngx_buf_t 缓冲区：
- 调用 ngx_event_pipe_remove_shadow_links 方法释放当前缓冲区 ngx_buf_t 中的 shadow 域；
- 计算当前缓冲区剩余的内存空间大小为 size：
- 若本次接收到的上游响应包体 n 不小于 size，表示当前缓冲区已满，调用 input_filter 方法处理当前缓冲区的响应包体，把其挂载到 in 链表中；
- 若本次接收到的上游响应包体 n 小于 size 值，则表示当前缓冲区还有剩余空间继续接收上游响应包体，先把本次接收到的响应包体缓冲区添加到 free_raw_bufs 链表尾端；
- *步骤8*：由于上面步骤接收到的上游响应包体最终会方法 free_raw_bufs 链表缓冲区中，再次检查 free_raw_bufs 链表缓冲区，若该缓冲区不为空，则调用 input_filter 方法处理该缓冲区的响应包体，同时调用 ngx_free_chain 方法是否 chain 缓冲区数据；
- *步骤9*：若 upstream_eof 或 upstream_error 标志位为 1，且 free_raw_bufs 不为空，再次调用 input_filter 方法处理 free_raw_bufs 缓冲区数据，若 free_bufs 标志位为 1，则调用 ngx_free 释放 shadow 域为空的缓冲区；

```
/* 读取上游响应 */
static ngx_int_t
ngx_event_pipe_read_upstream(ngx_event_pipe_t *p)
{
    ssize_t       n, size;
    ngx_int_t     rc;
    ngx_buf_t    *b;
    ngx_chain_t  *chain, *cl, *ln;

    /*
     * 若Nginx与上游之间的通信已经结束、
```

```
     * 或Nginx与上游之间的连接出错、
     * 或Nginx与上游之间的连接已经关闭；
     * 则直接return NGX_OK 从当前函数返回；
     */
    if (p->upstream_eof || p->upstream_error || p->upstream_done) {
        return NGX_OK;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
                   "pipe read upstream: %d", p->upstream->read->ready);

    /* 开始接收上游响应包体，并调用input_filter方法进行处理 */
    for ( ;; ) {

        /*
         * 若Nginx与上游之间的通信已经结束、
         * 或Nginx与上游之间的连接出错、
         * 或Nginx与上游之间的连接已经关闭；
         * 则直接break 从当前for循环退出；
         */
        if (p->upstream_eof || p->upstream_error || p->upstream_done) {
            break;
        }

        /*
         * 若preread_bufs链表缓冲区为空(表示在接收响应头部区间，未预接收响应包体)，
         * 且上游读事件未准备就绪，即没有可读的响应包体；
         * break从for循环退出；
         */
        if (p->preread_bufs == NULL && !p->upstream->read->ready) {
            break;
        }

        /*
         * 若preread_bufs链表缓冲区有未处理的数据，需要把它挂载到chain链表中
         * 即该数据是接收响应头部区间，预接收的响应包体；
         */
        if (p->preread_bufs) {/* the preread_bufs is not empty */

            /* use the pre-read bufs if they exist */

            chain = p->preread_bufs;/* 将预接收响应包体缓冲区添加到chain链表尾端 */
            p->preread_bufs = NULL;/* 使该缓冲区指向NULL，表示没有响应包体 */
            n = p->preread_size;/* 计算预接收响应包体的长度 n */

            ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
                           "pipe preread: %z", n);

            /*
             * 若preread_bufs链表缓冲区不为空，
             * 则设置read标志位为1，表示当前已经接收了响应包体；
             */
            if (n) {
                p->read = 1;
            }
```

```
            }

        } else {/* the preread_bufs is NULL */
        /*
         * 若preread_bufs链表缓冲区没有未处理的响应包体，
         * 则需要有缓冲区来接收上游响应包体；
         */

#if (NGX_HAVE_KQUEUE)
    ...
    ...
#endif

            /*
             * 若 free_raw_bufs不为空，则使用该链表缓冲区接收上游响应包体；
             * free_raw_bufs链表缓冲区用来保存调用一次ngx_event_pipe_read_upstream方法所接收到的上游响应包体；
             */
            if (p->free_raw_bufs) {

                /* use the free bufs if they exist */

                chain = p->free_raw_bufs;/* 将接收响应包体缓冲区添加到chain链表中 */
                if (p->single_buf) {/* 表示每一次只能接收一个ngx_buf_t缓冲区的响应包体 */
                    p->free_raw_bufs = p->free_raw_bufs->next;
                    chain->next = NULL;
                } else {
                    p->free_raw_bufs = NULL;
                }

            } else if (p->allocated < p->bufs.num) {
                /*
                 * 若 free_raw_bufs为空，且已分配的缓冲区数目allocated小于缓冲区数目bufs.num；
                 * 则需要分配一个新的缓冲区来接收上游响应包体；
                 */

                /* allocate a new buf if it's still allowed */

                /* 分配新的缓冲区来接收上游响应 */
                b = ngx_create_temp_buf(p->pool, p->bufs.size);
                if (b == NULL) {
                    return NGX_ABORT;
                }

                p->allocated++;

                /* 分配一个ngx_chain_t 链表缓冲区 */
                chain = ngx_alloc_chain_link(p->pool);
                if (chain == NULL) {
                    return NGX_ABORT;
                }

                /* 把新分配接收响应包体的缓冲区添加到chain链表中 */
                chain->buf = b;
                chain->next = NULL;
```

```
        } else if (!p->cacheable
                   && p->downstream->data == p->output_ctx
                   && p->downstream->write->ready
                   && !p->downstream->write->delayed)
        {
            /* 若free_raw_bufs为空，且allocated大于bufs.num，若cacheable标志位为0，即不启用文件缓存
,
             * 检查Nginx与下游之间连接，并检查该连接上写事件是否准备就绪，
             * 若已准备就绪，即表示可以向下游发送响应包体；
             */
            /*
             * if the bufs are not needed to be saved in a cache and
             * a downstream is ready then write the bufs to a downstream
             */

            /*
             * 设置upstream_blocked标志位为1，表示阻塞读取上游响应，
             * 因为没有缓冲区或文件缓存来接收响应包体，则应该阻塞读取上游响应包体；
             * 并break退出for循环，此时会向下游转发响应，释放缓冲区，以便再次接收上游响应包体；
             */
            p->upstream_blocked = 1;

            ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                           "pipe downstream ready");

            break;/* 退出for循环 */

        } else if (p->cacheable
                   || p->temp_file->offset < p->max_temp_file_size)
        {/* 若cacheable标志位为1，即开启了文件缓存，则检查临时文件是否达到最大长度，若未达到最大长
度 */

            /*
             * if it is allowed, then save some bufs from p->in
             * to a temporary file, and add them to a p->out chain
             */

            /* 将上游响应写入到临时文件中，此时free_raw_bufs有缓冲区空间来接收上游 响应包体 */
            rc = ngx_event_pipe_write_chain_to_temp_file(p);

            ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
                           "pipe temp offset: %O", p->temp_file->offset);

            if (rc == NGX_BUSY) {
                break;
            }

            if (rc == NGX_AGAIN) {
                if (ngx_event_flags & NGX_USE_LEVEL_EVENT
                    && p->upstream->read->active
                    && p->upstream->read->ready)
                {
                    if (ngx_del_event(p->upstream->read, NGX_READ_EVENT, 0)
                        == NGX_ERROR)
```

```
                == NGX_ERROR)
            {
                return NGX_ABORT;
            }
        }
    }

    if (rc != NGX_OK) {
        return rc;
    }

    chain = p->free_raw_bufs;
    if (p->single_buf) {
        p->free_raw_bufs = p->free_raw_bufs->next;
        chain->next = NULL;
    } else {
        p->free_raw_bufs = NULL;
    }

} else {
    /* 若没有缓冲区或文件缓存接收上游响应包体，则暂时不收受上游响应包体，break退出循环 */

    /* there are no bufs to read in */

    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                   "no pipe bufs to read in");

    break;
}
/* end of check the free_raw_bufs */

/*
 * 若有缓冲区接收上游响应包体，则调用recv_chain方法接收上游响应包体；
 * 把接收到的上游响应包体缓冲区添加到free_raw_bufs链表的尾端；
 */
n = p->upstream->recv_chain(p->upstream, chain);

ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
               "pipe recv chain: %z", n);

/* 将保存接收到上游响应包体的缓冲区添加到free_raw_bufs链表尾端 */
if (p->free_raw_bufs) {
    chain->next = p->free_raw_bufs;
}
p->free_raw_bufs = chain;

/* 下面根据所接收上游响应包体的返回值n来进行判断 */

/*
 * n = NGX_ERROR，表示发生错误，
 * 则设置upstream_error标志位为1，
 * 并return NGX_ERROR从当前函数返回；
 */
if (n == NGX_ERROR) {
    p->upstream_error = 1;
```

```
            return NGX_ERROR;
        }

        /*
         * n = NGX_AGAIN，表示没有读取到上游响应包体，
         * 则break跳出for循环；
         */
        if (n == NGX_AGAIN) {
            if (p->single_buf) {
                ngx_event_pipe_remove_shadow_links(chain->buf);
            }

            break;
        }

        /*
         * n 大于0，表示已经接收到上游响应包体，
         * 则设置read标志位为1；
         */
        p->read = 1;

        /*
         * n = 0，表示上游服务器主动关闭连接，
         * 则设置upstream_eof标志位为1，表示已关闭连接；
         * 并break退出for循环；
         */
        if (n == 0) {
            p->upstream_eof = 1;
            break;
        }
    }

    /* checking the preread_bufs is end */

    /* 下面开始处理已接收到的上游响应包体数据 */
    p->read_length += n;
    cl = chain;
    p->free_raw_bufs = NULL;

    /* 遍历待处理缓冲区链表chain中的ngx_buf_t缓冲区 */
    while (cl && n > 0) {

        /* 调用该方法将当前ngx_buf_t缓冲区中的shadow域释放 */
        ngx_event_pipe_remove_shadow_links(cl->buf);

        /* 计算当前缓冲区剩余的空间大小 */
        size = cl->buf->end - cl->buf->last;

        /* 若本次接收到上游响应包体的长度大于缓冲区剩余的空间，表示当前缓冲区已满 */
        if (n >= size) {
            cl->buf->last = cl->buf->end;

            /* STUB */ cl->buf->num = p->num++;

            /* 调用input_filter方法处理当前缓冲区响应包体，把其挂载到in链表中 */
```

```
                /* 调用input_filter方法处理当前缓冲区响应包体，把其挂载到in链表中 */
                if (p->input_filter(p, cl->buf) == NGX_ERROR) {
                    return NGX_ABORT;
                }

                n -= size;
                ln = cl;
                cl = cl->next;
                ngx_free_chain(p->pool, ln);

            } else {
            /*
             * 若本次接收到上游响应包体的长度小于缓冲区剩余的空间，
             * 表示当前缓冲区还有剩余空间接收上游响应包体；
             * 则先把本次接收到的响应包体缓冲区添加到free_raw_bufs链表尾端；
             */
                cl->buf->last += n;
                n = 0;
            }
        }

        if (cl) {
            for (ln = cl; ln->next; ln = ln->next) { /* void */ }

            ln->next = p->free_raw_bufs;
            p->free_raw_bufs = cl;
        }
    }

    /* end of the For cycle */

#if (NGX_DEBUG)
    ...
    ...
#endif

    /* 若free_raw_bufs不为空 */
    if (p->free_raw_bufs && p->length != -1) {
        cl = p->free_raw_bufs;

        if (cl->buf->last - cl->buf->pos >= p->length) {

            p->free_raw_bufs = cl->next;

            /* STUB */ cl->buf->num = p->num++;

            /* 调用input_filter方法处理free_raw_bufs缓冲区 */
            if (p->input_filter(p, cl->buf) == NGX_ERROR) {
                return NGX_ABORT;
            }

            /* 释放已被处理的chain缓冲区数据 */
            ngx_free_chain(p->pool, cl);
        }
    }
```

```
    if (p->length == 0) {
        p->upstream_done = 1;
        p->read = 1;
    }

    /*
     * 检查upstream_eof或upstream_error标志位是否为1，若其中一个为1，表示连接已经关闭，
     * 若连接已经关闭，且free_raw_bufs缓冲区不为空；
     */
    if ((p->upstream_eof || p->upstream_error) && p->free_raw_bufs) {

        /* STUB */ p->free_raw_bufs->buf->num = p->num++;

        /* 再次调用input_filter方法处理free_raw_bufs缓冲区的响应包体数据 */
        if (p->input_filter(p, p->free_raw_bufs->buf) == NGX_ERROR) {
            return NGX_ABORT;
        }

        p->free_raw_bufs = p->free_raw_bufs->next;

        /* 检查free_bufs标志位，若为1，则释放shadow域为空的缓冲区 */
        if (p->free_bufs && p->buf_to_file == NULL) {
            for (cl = p->free_raw_bufs; cl; cl = cl->next) {
                if (cl->buf->shadow == NULL) {
                    ngx_pfree(p->pool, cl->buf->start);
                }
            }
        }
    }

    if (p->cacheable && p->in) {
        if (ngx_event_pipe_write_chain_to_temp_file(p) == NGX_ABORT) {
            return NGX_ABORT;
        }
    }

    /* 返回NGX_OK，结束当前函数 */
    return NGX_OK;
}
```

ngx_event_pipe_write_to_downstream 方法将 in 链表和 out 链表中管理的缓冲区发送到下游服务器，由于 out 链表中缓冲区的内容在响应中的位置比 in 链表靠前，因此优先发送 out 链表内容给下游服务器。

ngx_event_pipe_write_to_downstream 方法的执行流程如下：

- *步骤1*：检查上游连接是否结束，即标志位 upstream_eof、upstream_error、upstream_done 有一个为 1，则表示不需要再接收上游响应包体，跳至*步骤2*执行，否则跳至*步骤5*执行；
- *步骤2*：调用 output_filter 方法将 out 链表缓冲区中的响应包体发送给下游服务器；

- *步骤3*：调用 output_filter 方法将 in 链表缓冲区中的响应包体发送给下游服务器；
- *步骤4*：设置 downstream_done 标志位为 1，结束当前函数；
- *步骤5*：计算 busy 链表缓冲区中待发送的响应包体长度 bsize，若 bsize 大于配置项规定值 busy_size，则跳至*步骤7*执行，否则继续向下游准备发送 out 或 in 链表缓冲区中的响应包体；
- *步骤6*：检查 out 链表是否为空：
- 若 out 链表不为空，取出 out 链表首个缓冲区 ngx_buf_t 作为发送响应包体，跳至 *步骤7*执行；
- 若 out 链表为空，检查 in 链表是否为空：
- 若 in 链表为空，则说明本次没有需要发送的响应包体，则返回 NGX_OK，结束当前函数；
- 若 in 链表不为空，取出 in 链表首部的第一个缓冲区作为待发送响应包体缓冲区，跳至 *步骤7*执行；
- *步骤7*：检查以前待发送响应包体长度加上本次本次需要发送的响应包体长度是否大于 busy_size，若大于 busy_size，跳至*步骤8*执行；否则跳至*步骤5*执行；
- *步骤8*：调用 output_filter 方法向下游服务器发送存储响应包体的 out 缓冲区链表；
- *步骤9*：调用 ngx_chain_update_chain 方法更新 free、busy、out 缓冲区；
- *步骤10*：遍历 free 链表，释放缓冲区中的 shadow 域；

```
/* 向下游服务器转发响应包体 */
static ngx_int_t
ngx_event_pipe_write_to_downstream(ngx_event_pipe_t *p)
{
    u_char           *prev;
    size_t            bsize;
    ngx_int_t         rc;
    ngx_uint_t        flush, flushed, prev_last_shadow;
    ngx_chain_t       *out, **ll, *cl;
    ngx_connection_t  *downstream;

    /* 获取Nginx与下游服务器之间的连接 */
    downstream = p->downstream;

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
                   "pipe write downstream: %d", downstream->write->ready);

    flushed = 0;

    for ( ;; ) {
        /* downstream_error标志位为1，表示与下游之间的连接出现错误 */
        if (p->downstream_error) {
            return ngx_event_pipe_drain_chains(p);
        }

        /* 检查与上游之间的连接是否关闭，若已关闭 */
```

```
        if (p->upstream_eof || p->upstream_error || p->upstream_done) {

            /* pass the p->out and p->in chains to the output filter */

            for (cl = p->busy; cl; cl = cl->next) {
                cl->buf->recycled = 0;
            }

            /* 调用output_filter方法将out链表中的缓冲区响应包体转发给下游 */
            if (p->out) {
                ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                               "pipe write downstream flush out");

                for (cl = p->out; cl; cl = cl->next) {
                    cl->buf->recycled = 0;
                }

                rc = p->output_filter(p->output_ctx, p->out);

                if (rc == NGX_ERROR) {
                    p->downstream_error = 1;
                    return ngx_event_pipe_drain_chains(p);
                }

                p->out = NULL;
            }

            /* 调用output_filter方法将in链表中的缓冲区响应包体转发给下游 */
            if (p->in) {
                ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                               "pipe write downstream flush in");

                for (cl = p->in; cl; cl = cl->next) {
                    cl->buf->recycled = 0;
                }

                rc = p->output_filter(p->output_ctx, p->in);

                if (rc == NGX_ERROR) {
                    p->downstream_error = 1;
                    return ngx_event_pipe_drain_chains(p);
                }

                p->in = NULL;
            }

            if (p->cacheable && p->buf_to_file) {
                ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                               "pipe write chain");

                if (ngx_event_pipe_write_chain_to_temp_file(p) == NGX_ABORT) {
                    return NGX_ABORT;
                }
            }
```

```
            ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
                           "pipe write downstream done");

            /* TODO: free unused bufs */

            p->downstream_done = 1;
            break;
        }

        /*
         * 若上游连接没有关闭，则检查下游连接上的写事件是否准备就绪；
         * 若准备就绪，则表示可以向下游转发响应包体；
         * 若未准备就绪，则break退出for循环，return NGX_OK从当前函数返回；
         */
        if (downstream->data != p->output_ctx
            || !downstream->write->ready
            || downstream->write->delayed)
        {
            break;
        }

        /* bsize is the size of the busy recycled bufs */

        prev = NULL;
        bsize = 0;

        /* 计算busy链表缓冲区中待发送响应包体的长度bsize */
        for (cl = p->busy; cl; cl = cl->next) {

            if (cl->buf->recycled) {
                if (prev == cl->buf->start) {
                    continue;
                }

                bsize += cl->buf->end - cl->buf->start;
                prev = cl->buf->start;
            }
        }

        ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
                       "pipe write busy: %uz", bsize);

        out = NULL;

        /* 检查bsize是否大于busy_size配置项 */
        if (bsize >= (size_t) p->busy_size) {
            flush = 1;
            goto flush;
        }

        /* 若bsize小于busy_size配置项 */
        flush = 0;
        ll = NULL;
        prev_last_shadow = 1;
```

```
        /*
         * 检查in、out链表缓冲区是否为空，若不为空；
         * 将out、in链表首个缓冲区作为发送内容；
         */
        for ( ;; ) {
            if (p->out) {/* out链表不为空，则取出首个缓冲区作为发送响应内容 */
                cl = p->out;

                if (cl->buf->recycled) {
                    ngx_log_error(NGX_LOG_ALERT, p->log, 0,
                                  "recycled buffer in pipe out chain");
                }

                p->out = p->out->next;

            } else if (!p->cacheable && p->in) {
                /* 若out为空，检查in是否为空，若in不为空，则首个缓冲区作为发送内容 */
                cl = p->in;

                ngx_log_debug3(NGX_LOG_DEBUG_EVENT, p->log, 0,
                               "pipe write buf ls:%d %p %z",
                               cl->buf->last_shadow,
                               cl->buf->pos,
                               cl->buf->last - cl->buf->pos);

                if (cl->buf->recycled && prev_last_shadow) {
                    /* 判断待发送响应包体长度加上本次缓冲区的长度是否大于busy_size */
                    if (bsize + cl->buf->end - cl->buf->start > p->busy_size) {
                        flush = 1;
                        break;
                    }

                    bsize += cl->buf->end - cl->buf->start;
                }

                prev_last_shadow = cl->buf->last_shadow;

                p->in = p->in->next;

            } else {
                break;
            }

            cl->next = NULL;

            if (out) {
                *ll = cl;
            } else {
                out = cl;
            }
            ll = &cl->next;
        }

    flush:
```

```
        ngx_log_debug2(NGX_LOG_DEBUG_EVENT, p->log, 0,
                       "pipe write: out:%p, f:%d", out, flush);

        if (out == NULL) {

            if (!flush) {
                break;
            }

            /* a workaround for AIO */
            if (flushed++ > 10) {
                return NGX_BUSY;
            }
        }

        /* 调用output_filter方法发送out链表缓冲区 */
        rc = p->output_filter(p->output_ctx, out);

        /* 更新free、busy、out链表缓冲区 */
        ngx_chain_update_chains(p->pool, &p->free, &p->busy, &out, p->tag);

        if (rc == NGX_ERROR) {
            p->downstream_error = 1;
            return ngx_event_pipe_drain_chains(p);
        }

        /* 遍历free链表，释放缓冲区中的shadow域 */
        for (cl = p->free; cl; cl = cl->next) {

            if (cl->buf->temp_file) {
                if (p->cacheable || !p->cyclic_temp_file) {
                    continue;
                }

                /* reset p->temp_offset if all bufs had been sent */

                if (cl->buf->file_last == p->temp_file->offset) {
                    p->temp_file->offset = 0;
                }
            }

            /* TODO: free buf if p->free_bufs && upstream done */

            /* add the free shadow raw buf to p->free_raw_bufs */

            if (cl->buf->last_shadow) {
                if (ngx_event_pipe_add_free_buf(p, cl->buf->shadow) != NGX_OK) {
                    return NGX_ABORT;
                }

                cl->buf->last_shadow = 0;
            }
```

```
            cl->buf->shadow = NULL;
        }
    }

    return NGX_OK;
}
```

```
static void
ngx_http_upstream_process_request(ngx_http_request_t *r)
{
    ngx_temp_file_t      *tf;
    ngx_event_pipe_t     *p;
    ngx_http_upstream_t  *u;

    u = r->upstream;
    p = u->pipe;

    if (u->peer.connection) {

        if (u->store) {

            if (p->upstream_eof || p->upstream_done) {

                tf = p->temp_file;

                if (u->headers_in.status_n == NGX_HTTP_OK
                    && (p->upstream_done || p->length == -1)
                    && (u->headers_in.content_length_n == -1
                        || u->headers_in.content_length_n == tf->offset))
                {
                    ngx_http_upstream_store(r, u);
                    u->store = 0;
                }
            }
        }

#if (NGX_HTTP_CACHE)
    ...
    ...
#endif

        if (p->upstream_done || p->upstream_eof || p->upstream_error) {
            ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                           "http upstream exit: %p", p->out);

            if (p->upstream_done
                || (p->upstream_eof && p->length == -1))
            {
                ngx_http_upstream_finalize_request(r, u, 0);
                return;
            }

            if (p->upstream_eof) {
                ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
```

```
                ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                              "upstream prematurely closed connection");
            }

            ngx_http_upstream_finalize_request(r, u, NGX_HTTP_BAD_GATEWAY);
            return;
        }
    }

    if (p->downstream_error) {
        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                       "http upstream downstream error");

        if (!u->cacheable && !u->store && u->peer.connection) {
            ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        }
    }
}
```

结束 upstream 请求

结束 upstream 请求由函数 ngx_http_upstream_finalize_request 实现，该函数最终会调用 HTTP 框架的 ngx_http_finalize_request 方法来结束请求。

```
static void
ngx_http_upstream_finalize_request(ngx_http_request_t *r,
    ngx_http_upstream_t *u, ngx_int_t rc)
{
    ngx_uint_t   flush;
    ngx_time_t  *tp;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "finalize http upstream request: %i", rc);

    /* 将 cleanup 指向的清理资源回调方法设置为 NULL */
    if (u->cleanup) {
        *u->cleanup = NULL;
        u->cleanup = NULL;
    }
    /* 释放解析主机域名时分配的资源 */
    if (u->resolved && u->resolved->ctx) {
        ngx_resolve_name_done(u->resolved->ctx);
        u->resolved->ctx = NULL;
    }

    /* 设置当前时间为 HTTP 响应结束的时间 */
    if (u->state && u->state->response_sec) {
        tp = ngx_timeofday();
        u->state->response_sec = tp->sec - u->state->response_sec;
        u->state->response_msec = tp->msec - u->state->response_msec;

        if (u->pipe && u->pipe->read_length) {
            u->state->response_length = u->pipe->read_length;
```

```
        }
    }

    /* 调用该方法执行一些操作 */
    u->finalize_request(r, rc);

    /* 调用 free 方法释放连接资源 */
    if (u->peer.free && u->peer.sockaddr) {
        u->peer.free(&u->peer, u->peer.data, 0);
        u->peer.sockaddr = NULL;
    }

    /* 若上游连接还未关闭，则调用 ngx_close_connection 方法关闭该连接 */
    if (u->peer.connection) {

#if (NGX_HTTP_SSL)
    ...
    ...
#endif

        ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                       "close http upstream connection: %d",
                       u->peer.connection->fd);

        if (u->peer.connection->pool) {
            ngx_destroy_pool(u->peer.connection->pool);
        }

        ngx_close_connection(u->peer.connection);
    }

    u->peer.connection = NULL;

    if (u->pipe && u->pipe->temp_file) {
        ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                       "http upstream temp fd: %d",
                       u->pipe->temp_file->file.fd);
    }

    /* 若使用了文件缓存，则调用 ngx_delete_file 方法删除用于缓存响应的临时文件 */
    if (u->store && u->pipe && u->pipe->temp_file
        && u->pipe->temp_file->file.fd != NGX_INVALID_FILE)
    {
        if (ngx_delete_file(u->pipe->temp_file->file.name.data)
            == NGX_FILE_ERROR)
        {
            ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
                          ngx_delete_file_n " \"%s\" failed",
                          u->pipe->temp_file->file.name.data);
        }
    }

#if (NGX_HTTP_CACHE)
    ...
```

```
    ...
#endif

    if (r->subrequest_in_memory
        && u->headers_in.status_n >= NGX_HTTP_SPECIAL_RESPONSE)
    {
        u->buffer.last = u->buffer.pos;
    }

    if (rc == NGX_DECLINED) {
        return;
    }

    r->connection->log->action = "sending to client";

    if (!u->header_sent
        || rc == NGX_HTTP_REQUEST_TIME_OUT
        || rc == NGX_HTTP_CLIENT_CLOSED_REQUEST)
    {
        ngx_http_finalize_request(r, rc);
        return;
    }

    flush = 0;

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        rc = NGX_ERROR;
        flush = 1;
    }

    if (r->header_only) {
        ngx_http_finalize_request(r, rc);
        return;
    }

    if (rc == 0) {
        rc = ngx_http_send_special(r, NGX_HTTP_LAST);

    } else if (flush) {
        r->keepalive = 0;
        rc = ngx_http_send_special(r, NGX_HTTP_FLUSH);
    }

    /* 调用 HTTP 框架实现的 ngx_http_finalize_request 方法关闭请求 */
    ngx_http_finalize_request(r, rc);
}
```

Nginx 中 upstream 机制的负载均衡

目录

负载均衡

upstream 机制使得 Nginx 以反向代理的形式运行，因此 Nginx 接收客户端的请求，并根据客户端的请求，Nginx 选择合适后端服务器来处理该请求。但是若存在多台后端服务器时，Nginx 是根据怎样的策略来决定哪个后端服务器负责处理请求？这就涉及到后端服务器的负载均衡问题。
Nginx 的负载均衡策略可以划分为两大类：内置策略 和 扩展策略。内置策略包含 加权轮询 和 IP hash，在默认情况下这两种策略会编译进 Nginx 内核，只需在 Nginx 配置中指明参数即可。扩展策略有第三方模块策略：fair、URL hash、consistent hash等，默认不编译进 Nginx 内核。本文只讲解 加权轮询 和 IP_hash 策略。

加权轮询

加权轮询策略是先计算每个后端服务器的权重，然后选择权重最高的后端服务器来处理请求。

相关结构体

ngx_http_upstream_peer_t 结构体

```
typedef struct {
    /* 负载均衡的类型 */
    ngx_http_upstream_init_pt        init_upstream;
    /* 负载均衡类型的初始化函数 */
    ngx_http_upstream_init_peer_pt   init;
    /* 指向 ngx_http_upstream_rr_peers_t 结构体 */
    void                             *data;
} ngx_http_upstream_peer_t;
```

ngx_http_upstream_server_t 结构体

```
/* 服务器结构体 */
typedef struct {
    /* 指向存储 IP 地址的数组，因为同一个域名可能会有多个 IP 地址 */
    ngx_addr_t                       *addrs;
    /* IP 地址数组中元素个数 */
    ngx_uint_t                       naddrs;
    /* 权重 */
    ngx_uint_t                       weight;
    /* 最大失败次数 */
    ngx_uint_t                       max_fails;
    /* 失败时间阈值 */
    time_t                           fail_timeout;

    /* 标志位，若为 1，表示不参与策略选择 */
    unsigned                         down:1;
    /* 标志位，若为 1，表示为备用服务器 */
    unsigned                         backup:1;
} ngx_http_upstream_server_t;
```

ngx_http_upstream_rr_peer_t 结构体

```
typedef struct {
    /* 后端服务器 IP 地址 */
    struct sockaddr                *sockaddr;
    /* 后端服务器 IP 地址的长度 */
    socklen_t                       socklen;
    /* 后端服务器的名称 */
    ngx_str_t                       name;

    /* 后端服务器当前的权重 */
    ngx_int_t                       current_weight;
    /* 后端服务器有效权重 */
    ngx_int_t                       effective_weight;
    /* 配置项所指定的权重 */
    ngx_int_t                       weight;

    /* 已经失败的次数 */
    ngx_uint_t                      fails;
    /* 访问时间 */
    time_t                         accessed;
    time_t                         checked;

    /* 最大失败次数 */
    ngx_uint_t                      max_fails;
    /* 失败时间阈值 */
    time_t                         fail_timeout;

    /* 后端服务器是否参与策略，若为1，表示不参与 */
    ngx_uint_t                      down;          /* unsigned  down:1; */

#if (NGX_HTTP_SSL)
    ngx_ssl_session_t              *ssl_session;   /* local to a process */
#endif
} ngx_http_upstream_rr_peer_t;
```

ngx_http_upstream_rr_peers_t 结构体

```
typedef struct ngx_http_upstream_rr_peers_s  ngx_http_upstream_rr_peers_t;

struct ngx_http_upstream_rr_peers_s {
    /* 竞选队列中后端服务器的数量 */
    ngx_uint_t                      number;

 /* ngx_mutex_t                     *mutex; */

    /* 所有后端服务器总的权重 */
    ngx_uint_t                      total_weight;

    /* 标志位，若为 1，表示后端服务器仅有一台，此时不需要选择策略 */
    unsigned                        single:1;
    /* 标志位，若为 1，表示所有后端服务器总的权重等于服务器的数量 */
    unsigned                        weighted:1;

    ngx_str_t                      *name;

    /* 后端服务器的链表 */
    ngx_http_upstream_rr_peers_t   *next;

    /* 特定的后端服务器 */
    ngx_http_upstream_rr_peer_t     peer[1];
};
```

ngx_http_upstream_rr_peer_data_t 结构体

```
typedef struct {
    ngx_http_upstream_rr_peers_t   *peers;
    ngx_uint_t                      current;
    uintptr_t                      *tried;
    uintptr_t                       data;
} ngx_http_upstream_rr_peer_data_t;
```

加权轮询策略的启动

在 Nginx 启动过程中，在解析完 http 配置块之后，会调用各个 http 模块对应的初始函数。对于 upstream 机制的 `ngx_http_upstream_module` 模块来说，对应的 main 配置初始函数是 `ngx_http_upstream_init_main_conf()` 如下所示：

```
for (i = 0; i < umcf->upstreams.nelts; i++) {

    init = uscfp[i]->peer.init_upstream ?
        uscfp[i]->peer.init_upstream:                    ngx_http_upstream_init_round_robin;

    if (init(cf, uscfp[i]) != NGX_OK) {
            return NGX_CONF_ERROR;
    }
}
```

在 ngx_http_upstream_module 模块中，如果用户没有做任何策略选择，那么执行默认采用加权轮询策略初始函数为 ngx_http_upstream_init_round_robin 。否则的话执行的是 uscfp[i]->peer.init_upstream 指针函数。

当接收到来自客户端的请求时，Nginx 会调用 ngx_http_upstream_init_request 初始化请求的过程中，调用 uscf->peer.init(r, uscf) ，对于 upstream 机制的加权轮询策略来说该方法就是 ngx_http_upstream_init_round_robin_peer ，该方法完成请求初始化工作。

```
static void
ngx_http_upstream_init_request(ngx_http_request_t *r)
{
...
if (uscf->peer.init(r, uscf) != NGX_OK) {
        ngx_http_upstream_finalize_request(r, u,
                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    ngx_http_upstream_connect(r, u);
}
```

完成客户端请求的初始化工作之后，会选择一个后端服务器来处理该请求，选择后端服务器由函数 ngx_http_upstream_get_round_robin_peer 实现。该函数在 ngx_event_connect_peer 中被调用。

```
ngx_int_t
ngx_event_connect_peer(ngx_peer_connection_t *pc)
{
...
/* 调用 ngx_http_upstream_get_round_robin_peer  */
   rc = pc->get(pc, pc->data);
   if (rc != NGX_OK) {
       return rc;
   }

  s = ngx_socket(pc->sockaddr->sa_family, SOCK_STREAM, 0);
...
}
```

当已经选择一台后端服务器来处理请求时，接下来就会测试该后端服务器的连接情况，测试连接由函数 ngx_http_upstream_test_connect 实现，在函数 ngx_http_upstream_send_request 中被调用。

```
static void
ngx_http_upstream_send_request(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
...
    if (!u->request_sent && ngx_http_upstream_test_connect(c) != NGX_OK) {
        /* 测试连接失败 */
        ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
        return;
    }
...
}
```

若连接测试失败，会由函数 `ngx_http_upstream_next` 发起再次测试，若测试成功，则处理完请求之后，会调用 `ngx_http_upstream_free_round_robin_peer` 释放后端服务器。

加权轮询工作流程

加权轮询策略的基本工作过程是：初始化负载均衡服务器列表，初始化后端服务器，选择合适后端服务器处理请求，释放后端服务器。

初始化服务器列表

初始化服务器列表由函数 `ngx_http_upstream_init_round_robin` 实现，该函数的执行流程如下所示：

- 第一种情况：若 upstream 机制配置项中配置了服务器：
- 初始化非备用服务器列表，并将其挂载到 `us->peer.data` 中；
- 初始化备用服务器列表，并将其挂载到 `peers->next` 中；
- 第二种情况：采用默认的方式 proxy_pass 配置后端服务器地址；
- 初始化非备用服务器列表，并将其挂载到 `us->peer.data` 中；

该方法执行完成之后得到的结构如下图所示：

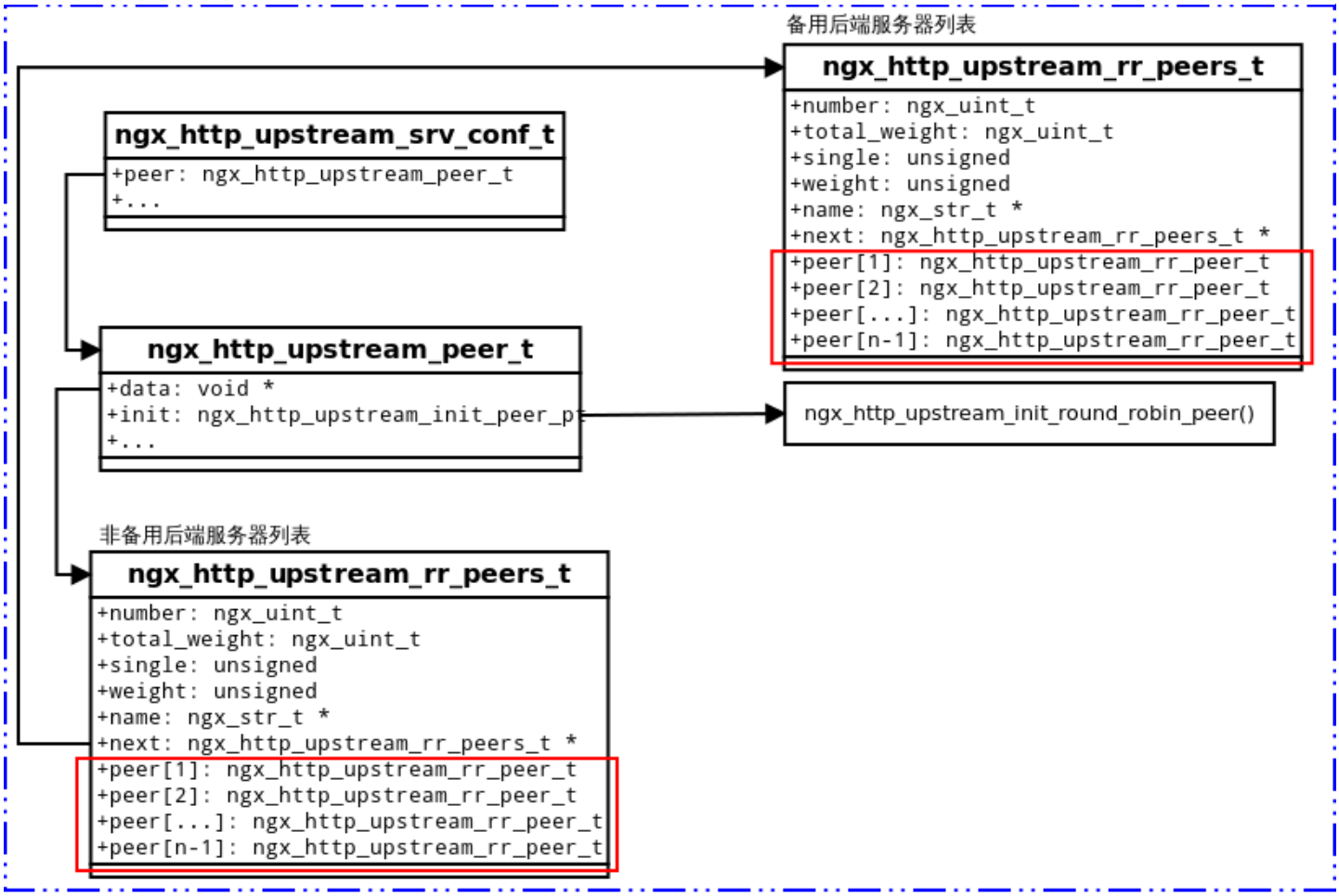


图 1 初始化后端服务器列表

```
/* 初始化服务器负载均衡列表 */
ngx_int_t
ngx_http_upstream_init_round_robin(ngx_conf_t *cf,
    ngx_http_upstream_srv_conf_t *us)
{
    ngx_url_t                  u;
    ngx_uint_t                 i, j, n, w;
    ngx_http_upstream_server_t    *server;
    ngx_http_upstream_rr_peers_t  *peers, *backup;

    /* 设置 ngx_http_upstream_peer_t 结构体中 init 的回调方法 */
    us->peer.init = ngx_http_upstream_init_round_robin_peer;

    /* 第一种情况：若 upstream 机制中有配置后端服务器 */
    if (us->servers) {
        /* ngx_http_upstream_srv_conf_t us 结构体成员 servers 是一个指向服务器数组 ngx_array_t 的指针，
*/
        server = us->servers->elts;

        n = 0;
        w = 0;
        /* 在这里说明下：一个域名可能会对应多个 IP 地址，upstream 机制中把一个 IP 地址看作一个后端服务
器 */

        /* 遍历服务器数组中所有后端服务器，统计非备用后端服务器的 IP 地址总个数(即非备用后端服务器总的
个数) 和 总权重 */
        for (i = 0; i < us->servers->nelts; i++) {
```

```
            /* 若当前服务器是备用服务器，则 continue 跳过以下检查，继续检查下一个服务器 */
            if (server[i].backup) {ngx_http_upstream_peer_t
                continue;
            }

            /* 统计所有非备用后端服务器 IP 地址总的个数(即非备用后端服务器总的个数) */
            n += server[i].naddrs;
            /* 统计所有非备用后端服务器总的权重 */
            w += server[i].naddrs * server[i].weight;
        }

        /* 若 upstream 机制中配置项指令没有设置后端服务器，则出错返回 */
        if (n == 0) {
            ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
                          "no servers in upstream \"%V\" in %s:%ui",
                          &us->host, us->file_name, us->line);
            return NGX_ERROR;
        }

        /* 值得注意的是：备用后端服务器列表 和 非备用后端服务器列表 是分开挂载的，因此需要分开设置 */
        /* 为非备用后端服务器分配内存空间 */
        peers = ngx_pcalloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
                            + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));
        if (peers == NULL) {
            return NGX_ERROR;
        }

        /* 初始化非备用后端服务器列表 ngx_http_upstream_rr_peers_t 结构体 */
        peers->single = (n == 1);/* 表示只有一个非备用后端服务器 */
        peers->number = n;/* 非备用后端服务器总的个数 */
        peers->weighted = (w != n);/* 设置默认权重为 1 或 0 */
        peers->total_weight = w;/* 设置非备用后端服务器总的权重 */
        peers->name = &us->host;/* 非备用后端服务器名称 */

        n = 0;

        /* 遍历服务器数组中所有后端服务器，初始化非备用后端服务器 */
        for (i = 0; i < us->servers->nelts; i++) {
            if (server[i].backup) {/* 若为备用服务器则 continue 跳过 */
                continue;
            }
            /* 以下关于 ngx_http_upstream_rr_peer_t 结构体中三个权重值的说明 */
            /*
             * effective_weight 相当于质量(来源于配置文件配置项的 weight)，current_weight 相当于重量。
             * 前者反应本质，一般是不变的。current_weight 是运行时的动态权值，它的变化基于 effective_wei
ght。
             * 但是 effective_weight 在其对应的 peer 服务异常时，会被调低，
             * 当服务恢复正常时，effective_weight 会逐渐恢复到实际值（配置项的weight）；
             */

            /* 遍历非备用后端服务器所对应 IP 地址数组中的所有 IP 地址(即一个后端服务器域名可能会对应多个 I
P 地址) */
            for (j = 0; j < server[i].naddrs; j++) {
                /* 为每个非备用后端服务器初始化 */
                peers->peer[n].sockaddr = server[i].addrs[j].sockaddr;/* 设置非备用后端服务器 IP 地址 */
```

```
            peers->peer[n].sockaddr = server[i].addrs[j].sockaddr;/* 设置非备用后端服务器 IP 地址 */
            peers->peer[n].socklen = server[i].addrs[j].socklen;/* 设置非备用后端服务器 IP 地址长度 */
            peers->peer[n].name = server[i].addrs[j].name;/* 设置非备用后端服务器域名 */
            peers->peer[n].weight = server[i].weight;/* 设置非备用后端服务器配置项权重 */
            peers->peer[n].effective_weight = server[i].weight;/* 设置非备用后端服务器有效权重 */
            peers->peer[n].current_weight = 0;/* 设置非备用后端服务器当前权重 */
            peers->peer[n].max_fails = server[i].max_fails;/* 设置非备用后端服务器最大失败次数 */
            peers->peer[n].fail_timeout = server[i].fail_timeout;/* 设置非备用后端服务器失败时间阈值 */
            peers->peer[n].down = server[i].down;/* 设置非备用后端服务器 down 标志位，若该标志位为 1
，则不参与策略 */
            n++;
        }
    }

    /*
     * 将非备用服务器列表挂载到 ngx_http_upstream_srv_conf_t 结构体成员结构体
     * ngx_http_upstream_peer_t peer 的成员 data 中；
     */
    us->peer.data = peers;

    /* backup servers */

    n = 0;
    w = 0;

     /* 遍历服务器数组中所有后端服务器，统计备用后端服务器的 IP 地址总个数(即备用后端服务器总的个数
) 和 总权重 */
    for (i = 0; i < us->servers->nelts; i++) {
        if (!server[i].backup) {
            continue;
        }

        n += server[i].naddrs;/* 统计所有备用后端服务器的 IP 地址总的个数 */
        w += server[i].naddrs * server[i].weight;/* 统计所有备用后端服务器总的权重 */
    }

    if (n == 0) {/* 若没有备用后端服务器，则直接返回 */
        return NGX_OK;
    }

    /* 分配备用服务器列表的内存空间 */
    backup = ngx_pcalloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
                         + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));
    if (backup == NULL) {
        return NGX_ERROR;
    }

    peers->single = 0;
    /* 初始化备用后端服务器列表 ngx_http_upstream_rr_peers_t 结构体 */
    backup->single = 0;
    backup->number = n;
    backup->weighted = (w != n);
    backup->total_weight = w;
    backup->name = &us->host;
```

```
        n = 0;

        /* 遍历服务器数组中所有后端服务器，初始化备用后端服务器 */
        for (i = 0; i < us->servers->nelts; i++) {
            if (!server[i].backup) {/* 若是非备用后端服务器，则 continue 跳过当前后端服务器，检查下一个后端服务器 */
                continue;
            }

            /* 遍历备用后端服务器所对应 IP 地址数组中的所有 IP 地址(即一个后端服务器域名可能会对应多个 IP 地址) */
            for (j = 0; j < server[i].naddrs; j++) {
                backup->peer[n].sockaddr = server[i].addrs[j].sockaddr;/* 设置备用后端服务器 IP 地址 */
                backup->peer[n].socklen = server[i].addrs[j].socklen;/* 设置备用后端服务器 IP 地址长度 */
                backup->peer[n].name = server[i].addrs[j].name;/* 设置备用后端服务器域名 */
                backup->peer[n].weight = server[i].weight;/* 设置备用后端服务器配置项权重 */
                backup->peer[n].effective_weight = server[i].weight;/* 设置备用后端服务器有效权重 */
                backup->peer[n].current_weight = 0;/* 设置备用后端服务器当前权重 */
                backup->peer[n].max_fails = server[i].max_fails;/* 设置备用后端服务器最大失败次数 */
                backup->peer[n].fail_timeout = server[i].fail_timeout;/* 设置备用后端服务器失败时间阈值 */
                backup->peer[n].down = server[i].down;/* 设置备用后端服务器 down 标志位，若该标志位为 1 ，则不参与策略 */
                n++;
            }
        }

        /*
         * 将备用服务器列表挂载到 ngx_http_upstream_rr_peers_t 结构体中
         * 的成员 next 中；
         */
        peers->next = backup;

        /* 第一种情况到此返回 */
        return NGX_OK;
    }

     /* 第二种情况：若 upstream 机制中没有直接配置后端服务器，则采用默认的方式 proxy_pass 配置后端服务器地址 */
    /* an upstream implicitly defined by proxy_pass, etc. */

    /* 若端口号为 0，则出错返回 */
    if (us->port == 0) {
        ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
                      "no port in upstream \"%V\" in %s:%ui",
                      &us->host, us->file_name, us->line);
        return NGX_ERROR;
    }

    /* 初始化 ngx_url_t 结构体所有成员为 0 */
    ngx_memzero(&u, sizeof(ngx_url_t));

    u.host = us->host;
    u.port = us->port;

    /* 解析 IP 地址 */
```

```
    /* 解析 IP 地址 */
    if (ngx_inet_resolve_host(cf->pool, &u) != NGX_OK) {
        if (u.err) {
            ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
                          "%s in upstream \"%V\" in %s:%ui",
                          u.err, &us->host, us->file_name, us->line);
        }

        return NGX_ERROR;
    }

    n = u.naddrs;

    /* 分配非备用后端服务器列表的内存空间 */
    peers = ngx_pcalloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
                        + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));
    if (peers == NULL) {
        return NGX_ERROR;
    }

    /* 初始化非备用后端服务器列表 */
    peers->single = (n == 1);
    peers->number = n;
    peers->weighted = 0;
    peers->total_weight = n;
    peers->name = &us->host;

    for (i = 0; i < u.naddrs; i++) {
        peers->peer[i].sockaddr = u.addrs[i].sockaddr;
        peers->peer[i].socklen = u.addrs[i].socklen;
        peers->peer[i].name = u.addrs[i].name;
        peers->peer[i].weight = 1;
        peers->peer[i].effective_weight = 1;
        peers->peer[i].current_weight = 0;
        peers->peer[i].max_fails = 1;
        peers->peer[i].fail_timeout = 10;
    }

    /* 挂载非备用后端服务器列表 */
    us->peer.data = peers;

    /* implicitly defined upstream has no backup servers */

    return NGX_OK;
}
```

选择合适的后端服务器

在选择合适的后端服务器处理客户请求时，首先需要初始化后端服务器，然后根据后端服务器的权重，选择权重最高的后端服务器来处理请求。

初始化后端服务器

上面的初始化负载服务器列表的全局初始化工作完成之后，当客户端发起请求时，Nginx 会选择一个合适的后端服务器来处理该请求。在本轮选择后端服务器之前，Nginx 会对后端服务器进行初始化工作，该工作由函数 `ngx_http_upstream_init_round_robin_peer` 实现。

`ngx_http_upstream_init_round_robin_peer` 函数的执行流程如下所示：

- 计算服务器列表中的数量 n，n 的取值为 非备用后端服务器数量 与 备用后端服务器数量 较大者；
- 根据 n 的取值，创建一个位图 `tried`，该位图是记录后端服务器是否被选择过：
- 若 n 不大于 32， 只需要在一个 `int` 中记录所有后端服务器的状态；
- 若 n 大于 32，则需要从内存池申请内存来存储所有后端服务器的状态；
- 设置 `ngx_peer_connection_t` 结构体中 `get` 的回调方法为 `ngx_http_upstream_get_round_robin_peer`；`free` 的回调方法为 `ngx_http_upstream_free_round_robin_peer`，设置 `tries` 重试连接的次数为非备用后端服务器的个数；

```
/* 当客户端发起请求时，upstream 机制为本轮选择一个后端服务器做初始化工作 */
ngx_int_t
ngx_http_upstream_init_round_robin_peer(ngx_http_request_t *r,
    ngx_http_upstream_srv_conf_t *us)
{
    ngx_uint_t                      n;
    ngx_http_upstream_rr_peer_data_t  *rrp;

    /* 注意：r->upstream->peer 是 ngx_peer_connection_t 结构体类型 */

    /* 获取当前客户端请求中的 ngx_http_upstream_rr_peer_data_t 结构体 */
    rrp = r->upstream->peer.data;

    if (rrp == NULL) {
        rrp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_rr_peer_data_t));
        if (rrp == NULL) {
            return NGX_ERROR;
        }

        r->upstream->peer.data = rrp;
    }

    /* 获取非备用后端服务器列表 */
    rrp->peers = us->peer.data;
    rrp->current = 0;/* 若采用遍历方式选择后端服务器时，作为起始节点编号 */

    /* 下面是取值 n，若存在备用后端服务器列表，则 n 的值为非备用后端服务器个数 与 备用后端服务器个数之间的较大者 */

    n = rrp->peers->number;

    if (rrp->peers->next && rrp->peers->next->number > n) {
```

```
        n = rrp->peers->next->number;
    }

    /* rrp->tried 是一个位图，在本轮选择中，该位图记录各个后端服务器是否被选择过 */

    /*
     * 如果后端服务器数量 n 不大于 32，则只需在一个 int 中即可记录下所有后端服务器状态；
     * 如果后端服务器数量 n 大于 32，则需在内存池中申请内存来存储所有后端服务器的状态；
     */
    if (n <= 8 * sizeof(uintptr_t)) {
        rrp->tried = &rrp->data;
        rrp->data = 0;

    } else {
        n = (n + (8 * sizeof(uintptr_t) - 1)) / (8 * sizeof(uintptr_t));

        rrp->tried = ngx_pcalloc(r->pool, n * sizeof(uintptr_t));
        if (rrp->tried == NULL) {
            return NGX_ERROR;
        }
    }

    /*
     * 设置 ngx_peer_connection_t 结构体中 get 、free 的回调方法；
     * 设置 ngx_peer_connection_t 结构体中 tries 重试连接的次数为非备用后端服务器的个数；
     */
    r->upstream->peer.get = ngx_http_upstream_get_round_robin_peer;
    r->upstream->peer.free = ngx_http_upstream_free_round_robin_peer;
    r->upstream->peer.tries = rrp->peers->number;
#if (NGX_HTTP_SSL)
    r->upstream->peer.set_session =
                        ngx_http_upstream_set_round_robin_peer_session;
    r->upstream->peer.save_session =
                        ngx_http_upstream_save_round_robin_peer_session;
#endif

    return NGX_OK;
}
```

根据权重选择后端服务器

完成后端服务器的初始化工作之后，根据各个后端服务器的权重来选择权重最高的后端服务器处理客户端请求，由函数 `ngx_http_upstream_get_round_robin_peer` 实现。

`ngx_http_upstream_get_round_robin_peer` 函数的执行流程如下所示：

- *步骤1*：检查 `ngx_http_upstream_rr_peers_t` 结构体中的 `single` 标志位：
- 若 `single` 标志位为 1，表示只有一台非备用后端服务器：
- 接着检查该非备用后端服务器的 `down` 标志位：
- 若 `down` 标志位为 0，则选择该非备用后端服务器来处理请求；

- 若 `down` 标志位为 1, 该非备用后端服务器表示不参与策略选择，则跳至 `goto failed` 步骤从备用后端服务器列表中选择后端服务器来处理请求；
- 若 `single` 标志位为 0，则表示不止一台非备用后端服务器，则调用 `ngx_http_upstream_get_peer` 方法根据非备用后端服务器的权重来选择一台后端服务器处理请求，根据该方法的返回值 `peer` 进行判断：
- 若该方法返回值 `peer = NULL`， 表示在非备用后端服务器列表中没有选中到合适的后端服务器来处理请求，则跳至 `goto failed` 从备用后端服务器列表中选择一台后端服务器来处理请求；
- 若该方法返回值 peer 不为 NULL，表示已经选中了合适的后端服务器来处理请求，设置该服务器重试连接次数 tries，并 `return NGX_OK` 从当前函数返回；
- *goto failed 步骤*：计算备用后端服务器在位图 tried 中的位置 n，并把他们在位图的记录都设置为 0，此时，把备用后端服务器列表作为参数调用 `ngx_http_upstream_get_round_robin_peer` 选择一台后端服务器来处理请求；

```
/* 选择一个后端服务器来处理请求 */
ngx_int_t
ngx_http_upstream_get_round_robin_peer(ngx_peer_connection_t *pc, void *data)
{
    ngx_http_upstream_rr_peer_data_t  *rrp = data;

    ngx_int_t                      rc;
    ngx_uint_t                     i, n;
    ngx_http_upstream_rr_peer_t   *peer;
    ngx_http_upstream_rr_peers_t  *peers;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                   "get rr peer, try: %ui", pc->tries);

    /* ngx_lock_mutex(rrp->peers->mutex); */

    pc->cached = 0;
    pc->connection = NULL;

    /*
     * 检查 ngx_http_upstream_rr_peers_t 结构体中的 single 标志位;
     * 若 single 标志位为 1，表示只有一台非备用后端服务器，
     * 接着检查该非备用后端服务器的 down 标志位，若 down 标志位为 0，则选择该非备用后端服务器来处理请求；
     * 若 down 标志位为 1, 该非备用后端服务器表示不参与策略选择，
     * 则跳至 goto failed 步骤从备用后端服务器列表中选择后端服务器来处理请求；
     */
    if (rrp->peers->single) {
        peer = &rrp->peers->peer[0];

        if (peer->down) {
            goto failed;
        }
```

```
    } else {/* 若 single 标志位为 0，表示不止一台非备用后端服务器 */

        /* there are several peers */

        /* 根据非备用后端服务器的权重来选择一台后端服务器处理请求 */
        peer = ngx_http_upstream_get_peer(rrp);

        if (peer == NULL) {
            /*
             * 若从非备用后端服务器列表中没有选择一台合适的后端服务器处理请求，
             * 则 goto failed 从备用后端服务器列表中选择一台后端服务器来处理请求；
             */
            goto failed;
        }

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                       "get rr peer, current: %ui %i",
                       rrp->current, peer->current_weight);
    }

    /*
     * 若从非备用后端服务器列表中已经选到了一台合适的后端服务器处理请求;
     * 则获取该后端服务器的地址信息；
     */
    pc->sockaddr = peer->sockaddr;/* 获取被选中的非备用后端服务器的地址 */
    pc->socklen = peer->socklen;/* 获取被选中的非备用后端服务器的地址长度 */
    pc->name = &peer->name;/* 获取被选中的非备用后端服务器的域名 */

    /* ngx_unlock_mutex(rrp->peers->mutex); */

    /*
     * 检查被选中的非备用后端服务器重试连接的次数为 1，且存在备用后端服务器列表，
     * 则将该非备用后端服务器重试连接的次数设置为 备用后端服务器个数加 1；
     * 否则不用重新设置；
     */
    if (pc->tries == 1 && rrp->peers->next) {
        pc->tries += rrp->peers->next->number;
    }

    /* 到此，表示已经选择到了一台合适的非备用后端服务器来处理请求，则成功返回 */
    return NGX_OK;

failed:
      /*
       * 若从非备用后端服务器列表中没有选择到后端服务器处理请求，
       * 若存在备用后端服务器，则从备用后端服务器列表中选择一台后端服务器来处理请求；
       */

    peers = rrp->peers;

    /* 若存在备用后端服务器，则从备用后端服务器列表中选择一台后端服务器来处理请求；*/
    if (peers->next) {

        /* ngx_unlock_mutex(peers->mutex); */
```

```
        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0, "backup servers");

        /* 获取备用后端服务器列表 */
        rrp->peers = peers->next;
        /* 把后端服务器重试连接的次数 tries 设置为备用后端服务器个数 number */
        pc->tries = rrp->peers->number;

        /* 计算备用后端服务器在位图中的位置 n */
        n = (rrp->peers->number + (8 * sizeof(uintptr_t) - 1))
                / (8 * sizeof(uintptr_t));

        /* 初始化备用后端服务器在位图 rrp->tried[i] 中的值为 0 */
        for (i = 0; i < n; i++) {
            rrp->tried[i] = 0;
        }

        /* 把备用后端服务器列表当前非备用后端服务器列表递归调用 ngx_http_upstream_get_round_robin_p
    eer 选择一台后端服务器 */
        rc = ngx_http_upstream_get_round_robin_peer(pc, rrp);

        /* 若选择成功则返回 */
        if (rc != NGX_BUSY) {
            return rc;
        }

        /* ngx_lock_mutex(peers->mutex); */
    }

    /*
     * 若从备用后端服务器列表中也没有选择到一台后端服务器处理请求，
     * 则重新设置非备用后端服务器连接失败的次数 fails 为 0 ,以便重新被选择；
     */
    /* all peers failed, mark them as live for quick recovery */

    for (i = 0; i < peers->number; i++) {
        peers->peer[i].fails = 0;
    }

    /* ngx_unlock_mutex(peers->mutex); */

    pc->name = peers->name;

    /* 选择失败，则返回 */
    return NGX_BUSY;
}
```

`ngx_http_upstream_get_peer` 函数是计算每一个后端服务器的权重值，并选择一个权重最高的后端服务器。

`ngx_http_upstream_get_peer` 函数的执行流程如下所示：

- for 循环遍历后端服务器列表，计算当前后端服务器在位图 tried 中的位置 n，判断当前服务器是否在

位图中记录过，若已经记录过，则 continue 继续检查下一个后端服务器；若没有记录过则继续当前后端服务器检查；

- 检查当前后端服务器的标志位 down，若该标志位为 1，表示该后端服务器不参与选择策略，则 continue 继续检查下一个后端服务器；若该标志位为 0，继续当前后端服务器的检查；
- 若当前后端服务器的连接失败次数已到达 max_failes，且睡眠时间还没到 fail_timedout ，则 continue 继续检查下一个后端服务器；否则继续当前后端服务器的检查；
- 计算当前后端服务器的权重，设置当前后端服务器的权重 current_weight 的值为原始值加上 effective_weight；设置总的权重 total 为原始值加上 effective_weight；
- 判断当前后端服务器是否异常，若 effective_weight 小于 weight，表示正常，则调整 effective_weight 的值 effective_weight++；
- 根据权重在后端服务器列表中选择权重最高的后端服务器 best；
- 计算被选中后端服务器咋服务器列表中的为 i，记录被选中后端服务器在 `ngx_http_upstream_rr_peer_data_t` 结构体 current 成员的值，在释放后端服务器时会用到该值；
- 计算被选中后端服务器在位图中的位置 n，并在该位置记录 best 后端服务器已经被选中过；
- 更新被选中后端服务器的权重，并返回被选中的后端服务器 best；

```
 /* 根据后端服务器的权重来选择一台后端服务器处理请求 */
static ngx_http_upstream_rr_peer_t *
ngx_http_upstream_get_peer(ngx_http_upstream_rr_peer_data_t *rrp)
{
    time_t                        now;
    uintptr_t                     m;
    ngx_int_t                     total;
    ngx_uint_t                    i, n;
    ngx_http_upstream_rr_peer_t  *peer, *best;

    now = ngx_time();

    best = NULL;
    total = 0;

    /* 遍历后端服务器列表 */
    for (i = 0; i < rrp->peers->number; i++) {

        /* 计算当前后端服务器在位图中的位置 n */
        n = i / (8 * sizeof(uintptr_t));
        m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));

        /* 当前后端服务器在位图中已经有记录，则不再次被选择，即 continue 检查下一个后端服务器 */
        if (rrp->tried[n] & m) {
            continue;
        }

        /* 若当前后端服务器在位图中没有记录，则可能被选中，接着计算其权重 */
        peer = &rrp->peers->peer[i];

        /* 检查当前后端服务器的 down 标志位，若为 1 表示不参与策略选择，则 continue 检查下一个后端服务器 */
```

```
        if (peer->down) {
            continue;
        }

        /*
         * 当前后端服务器的 down 标志位为 0,接着检查当前后端服务器连接失败的次数是否已经达到 max_fails
 ;
         * 且睡眠的时间还没到 fail_timeout，则当前后端服务器不被选择，continue 检查下一个后端服务器；
         */
        if (peer->max_fails
            && peer->fails >= peer->max_fails
            && now - peer->checked <= peer->fail_timeout)
        {
            continue;
        }

        /* 若当前后端服务器可能被选中，则计算其权重 */

        /*
         * 在上面初始化过程中 current_weight = 0，effective_weight = weight；
         * 此时，设置当前后端服务器的权重 current_weight 的值为原始值加上 effective_weight；
         * 设置总的权重为原始值加上 effective_weight；
         */
        peer->current_weight += peer->effective_weight;
        total += peer->effective_weight;

        /* 服务器正常，调整 effective_weight 的值 */
        if (peer->effective_weight < peer->weight) {
            peer->effective_weight++;
        }

        /* 若当前后端服务器的权重 current_weight 大于目前 best 服务器的权重，则当前后端服务器被选中 */
        if (best == NULL || peer->current_weight > best->current_weight) {
            best = peer;
        }
    }

    if (best == NULL) {
        return NULL;
    }

    /* 计算被选中后端服务器在服务器列表中的位置 i */
    i = best - &rrp->peers->peer[0];

    /* 记录被选中后端服务器在 ngx_http_upstream_rr_peer_data_t 结构体 current 成员的值，在释放后端服
务器时会用到该值 */
    rrp->current = i;

    /* 计算被选中后端服务器在位图中的位置 */
    n = i / (8 * sizeof(uintptr_t));
    m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));

    /* 在位图相应的位置记录被选中后端服务器 */
    rrp->tried[n] |= m;
```

```
    /* 更新被选中后端服务器的权重 */
    best->current_weight -= total;

    if (now - best->checked > best->fail_timeout) {
        best->checked = now;
    }

    /* 返回被选中的后端服务器 */
    return best;
}
```

释放后端服务器

成功连接后端服务器并且正常处理完成客户端请求后需释放后端服务器，由函数 `ngx_http_upstream_free_round_robin_peer` 实现。

```
/* 释放后端服务器 */
void
ngx_http_upstream_free_round_robin_peer(ngx_peer_connection_t *pc, void *data,
    ngx_uint_t state)
{
    ngx_http_upstream_rr_peer_data_t  *rrp = data;

    time_t                       now;
    ngx_http_upstream_rr_peer_t  *peer;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                   "free rr peer %ui %ui", pc->tries, state);

    /* TODO: NGX_PEER_KEEPALIVE */

    /* 若只有一个后端服务器，则设置 ngx_peer_connection_t 结构体成员 tries 为 0，并 return 返回 */
    if (rrp->peers->single) {
        pc->tries = 0;
        return;
    }

    /* 若不止一个后端服务器，则执行以下程序 */

    /* 获取已经被选中的后端服务器 */
    peer = &rrp->peers->peer[rrp->current];

    /*
     * 若在本轮被选中的后端服务器在进行连接测试时失败，或者在处理请求过程中失败，
     * 则需要进行重新选择后端服务器；
     */
    if (state & NGX_PEER_FAILED) {
        now = ngx_time();

        /* ngx_lock_mutex(rrp->peers->mutex); */

        peer->fails++;/* 增加当前后端服务器失败的次数 */
        /* 设置当前后端服务器访问的时间 */
```

```
        /* 设置当前后端服务器访问的时间 */
        peer->accessed = now;
        peer->checked = now;

        if (peer->max_fails) {
            /* 由于当前后端服务器失败，表示发生异常，此时降低 effective_weight 的值 */
            peer->effective_weight -= peer->weight / peer->max_fails;
        }

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                       "free rr peer failed: %ui %i",
                       rrp->current, peer->effective_weight);

        /* 保证 effective_weight 的值不能小于 0 */
        if (peer->effective_weight < 0) {
            peer->effective_weight = 0;
        }

        /* ngx_unlock_mutex(rrp->peers->mutex); */

    } else {/* 若被选中的后端服务器成功处理请求，并返回，则将其 fails 设置为 0 */

        /* mark peer live if check passed */

        /* 若 fail_timeout 时间已过，则将其 fails 设置为 0 */
        if (peer->accessed < peer->checked) {
            peer->fails = 0;
        }
    }

    /* 减少 tries 的值 */
    if (pc->tries) {
        pc->tries--;
    }

    /* ngx_unlock_mutex(rrp->peers->mutex); */
}
```

IP 哈希

IP 哈希策略选择后端服务器时，将来自同一个 IP 地址的客户端请求分发到同一台后端服务器处理。在 Nginx 中，IP 哈希策略的一些初始化工作是基于加权轮询策略的，这样减少了一些工作。

Nginx 使用 IP 哈希负载均衡策略时，在进行策略选择之前由 `ngx_http_upstream_init_ip_hash` 函数进行全局初始化工作，其实该函数也是调用加权轮询策略的全局初始化函数。当一个客户端请求过来时，Nginx 将调用 `ngx_http_upstream_init_ip_hash_peer()` 为选择后端服务器处理该请求做初始化工作。在多次哈希选择失败后，Nginx 会将选择策略退化到加权轮询。

`ngx_http_upstream_get_ip_hash_peer` 函数会在选择后端服务器时计算客户端请求 IP 地址的哈希值，并根据哈希值得到被选中的后端服务器，判断其是否可用，如果可用则保存服务器地址，若不可用则在上次哈希选择结果基础上再次进行哈希选择。如果哈希选择失败次数达到 20 次以上，此时回退到采用轮询

策略进行选择。

初始化后端服务器列表

初始化服务器列表工作是调用加权轮询策略的初始化函数，只是最后设置 IP 哈希的回调方法为 `ngx_http_upstream_init_ip_hash_peer`。

```
static ngx_int_t
ngx_http_upstream_init_ip_hash(ngx_conf_t *cf, ngx_http_upstream_srv_conf_t *us)
{
    /* 调用加权轮询策略的初始化函数 */
    if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
        return NGX_ERROR;
    }

    /* 由于 ngx_http_upstream_init_round_robin 方法的选择后端服务器处理客户请求的初始化函数
     * 为 us->peer.init = ngx_http_upstream_init_round_robin_peer;
     */
    /* 重新设置 ngx_http_upstream_peer_t 结构体中 init 的回调方法为  ngx_http_upstream_init_ip_hash_p
eer */

    us->peer.init = ngx_http_upstream_init_ip_hash_peer;

    return NGX_OK;
}
```

选择后端服务器

选择后端服务器之前会调用函数 `ngx_http_upstream_init_ip_hash_peer` 进行一些服务器初始化工作。最终由函数 `ngx_http_upstream_get_ip_hash_peer` 进行 IP 哈希选择。

`ngx_http_upstream_init_ip_hash_peer` 函数执行流程：

- 调用加权轮询策略的初始化函数 `ngx_http_upstream_init_round_robin_peer`；
- 设置 IP hash 的决策函数为 `ngx_http_upstream_get_ip_hash_peer`；
- 保存客户端 IP 地址；
- 初始化 `ngx_http_upstream_ip_hash_peer_data_t` 结构体成员 hash 值为 89；tries 重试连接次数为 0；`get_rr_peer` 为加权轮询的决策函数 `ngx_http_upstream_get_round_robin_peer`；

`ngx_http_upstream_get_ip_hash_peer` 函数执行流程：

- 若重试连接的次数 tries 大于 20，或 只有一台后端服务器，则直接调用加权轮询策略 get_rr_peer 选择当前后端服务器处理请求；
- 计算 IP 地址的 hash 值，下面根据哈希值进行选择后端服务器；
- 若 `ngx_http_upstream_rr_peers_t` 结构体中 weighted 标志位为 1，则被选中的后端服务器在后端服务器列表中的位置为 hash 值与后端服务器数量的余数 p；

- 若 ngx_http_upstream_rr_peers_t 结构体中 weighted 标志位为 0，首先计算 hash 值与后端服务器总权重的余数 w； 将 w 值减去后端服务器的权重，直到有一个后端服务器使 w 值小于 0，则选中该后端服务器来处理请求，并记录在后端服务器列表中的位置 p；
- 计算被选中后端服务器在位图中的位置 n；
- 若当前被选中的后端服务器已经在位图记录过，则跳至 goto next 执行；
- 检查当前被选中后端服务器的 down 标志位：
- 若该标志位为1，则跳至 goto next_try 执行；
- 若 down 标志位为 0，接着检查当前被选中后端服务器失败连接次数是否到达 max_fails，若已经达到 max_fails 次，并且睡眠时间还没到 fail_timeout，则跳至 goto next_try 执行；
- 若不满足以上条件，表示选择成功，记录当前后端服务器的地址信息，把当前后端服务器记录在位图相应的位置，更新哈希值，最后返回该后端服务器；
- *goto next*：tries 重试连接的次数加 1，并判断 tries 是否大于阈值 20，若大于，则采用加权轮询策略；
- *goto next_try *：把当前后端服务器记录在位图中，减少当前后端服务器重试连接的次数 tries；

```
static ngx_int_t
ngx_http_upstream_init_ip_hash_peer(ngx_http_request_t *r,
    ngx_http_upstream_srv_conf_t *us)
{
    struct sockaddr_in                  *sin;
#if (NGX_HAVE_INET6)
    struct sockaddr_in6                 *sin6;
#endif
    ngx_http_upstream_ip_hash_peer_data_t  *iphp;

    /* 分配 ngx_http_upstream_ip_hash_peer_data_t 结构体内存空间 */
    iphp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_ip_hash_peer_data_t));
    if (iphp == NULL) {
        return NGX_ERROR;
    }

    r->upstream->peer.data = &iphp->rrp;

    /* 调用加权轮询策略的初始化函数 ngx_http_upstream_init_round_robin_peer */
    if (ngx_http_upstream_init_round_robin_peer(r, us) != NGX_OK) {
        return NGX_ERROR;
    }

    /* 设置 IP hash 的决策函数  */
    r->upstream->peer.get = ngx_http_upstream_get_ip_hash_peer;

    switch (r->connection->sockaddr->sa_family) {
```

```
    /* 保存客户端 IP 地址 */

    /* IPv4 地址 */
    case AF_INET:
        sin = (struct sockaddr_in *) r->connection->sockaddr;
        iphp->addr = (u_char *) &sin->sin_addr.s_addr;
        iphp->addrlen = 3;
        break;

    /* IPv6 地址 */
#if (NGX_HAVE_INET6)
    case AF_INET6:
        sin6 = (struct sockaddr_in6 *) r->connection->sockaddr;
        iphp->addr = (u_char *) &sin6->sin6_addr.s6_addr;
        iphp->addrlen = 16;
        break;
#endif

    /* 非法地址 */
    default:
        iphp->addr = ngx_http_upstream_ip_hash_pseudo_addr;
        iphp->addrlen = 3;
    }

    /* 初始化 ngx_http_upstream_ip_hash_peer_data_t结构体成员 */
    iphp->hash = 89;
    iphp->tries = 0;
    /* 这个是设置为加权轮询策略的决策函数 */
    iphp->get_rr_peer = ngx_http_upstream_get_round_robin_peer;

    return NGX_OK;
}
```

```
/* 选择后端服务器处理请求 */
static ngx_int_t
ngx_http_upstream_get_ip_hash_peer(ngx_peer_connection_t *pc, void *data)
{
    ngx_http_upstream_ip_hash_peer_data_t  *iphp = data;

    time_t                        now;
    ngx_int_t                     w;
    uintptr_t                     m;
    ngx_uint_t                    i, n, p, hash;
    ngx_http_upstream_rr_peer_t  *peer;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                   "get ip hash peer, try: %ui", pc->tries);

    /* TODO: cached */

    /* 若重试连接的次数 tries 大于 20，或 只有一台后端服务器，则直接调用加权轮询策略选择当前后端服务器
处理请求 */
    if (iphp->tries > 20 || iphp->rrp.peers->single) {
```

```
        return iphp->get_rr_peer(pc, &iphp->rrp);
    }

    now = ngx_time();

    pc->cached = 0;
    pc->connection = NULL;

    hash = iphp->hash;

    for ( ;; ) {

        /* 计算 IP 地址的 hash 值 */
        for (i = 0; i < (ngx_uint_t) iphp->addrlen; i++) {
            hash = (hash * 113 + iphp->addr[i]) % 6271;/* hash 函数 */
        }

        /* 以下是根据 hash 值选择合适的后端服务器来处理请求 */

        /* 若 ngx_http_upstream_rr_peers_t 结构体中 weighted 标志位为 1，
         * 表示所有后端服务器的总权重 与 后端服务器的数量 相等，
         * 则被选中的后端服务器在后端服务器列表中的位置为 hash 值与后端服务器数量的余数 p；
         */
        if (!iphp->rrp.peers->weighted) {
            p = hash % iphp->rrp.peers->number;

        } else {
            /* 若 ngx_http_upstream_rr_peers_t 结构体中 weighted 标志位为 0，
             * 首先计算 hash 值与后端服务器总权重的余数 w；
             * 将 w 值减去后端服务器的权重，直到有一个后端服务器使 w 值小于 0，
             * 则选中该后端服务器来处理请求，并记录在后端服务器列表中的位置 p；
             */
            w = hash % iphp->rrp.peers->total_weight;

            for (i = 0; i < iphp->rrp.peers->number; i++) {
                w -= iphp->rrp.peers->peer[i].weight;
                if (w < 0) {
                    break;
                }
            }

            p = i;
        }

        /* 计算被选中后端服务器在位图中的位置 n */
        n = p / (8 * sizeof(uintptr_t));
        m = (uintptr_t) 1 << p % (8 * sizeof(uintptr_t));

        /* 若当前被选中的后端服务器已经在位图记录过，则跳至 goto next 执行 */
        if (iphp->rrp.tried[n] & m) {
            goto next;
        }

        ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                       "get ip hash peer, hash: %ui %04XA", p, m);
```

```
        /* 获取当前被选中的后端服务器 */
        peer = &iphp->rrp.peers->peer[p];

        /* ngx_lock_mutex(iphp->rrp.peers->mutex); */

        /* 检查当前被选中后端服务器的 down 标志位，若该标志位为1，则跳至 goto next_try 执行 */
        if (peer->down) {
            goto next_try;
        }

        /* 若 down 标志位为 0，接着检查当前被选中后端服务器失败连接次数是否到达 max_fails，
         * 若已经达到 max_fails 次，并且睡眠时间还没到 fail_timeout，则跳至 goto next_try 执行；
         */
        if (peer->max_fails
            && peer->fails >= peer->max_fails
            && now - peer->checked <= peer->fail_timeout)
        {
            goto next_try;
        }

        /* 若不满足以上条件，则表示选择后方服务器成功 */
        break;

    next_try:

        /* 把当前后端服务器记录在位图中 */
        iphp->rrp.tried[n] |= m;

        /* ngx_unlock_mutex(iphp->rrp.peers->mutex); */

        /* 减少当前后端服务器重试连接的次数 */
        pc->tries--;

    next:

        /* tries 重试连接的次数加 1，并判断 tries 是否大于阈值 20，若大于，则采用加权轮询策略 */
        if (++iphp->tries >= 20) {
            return iphp->get_rr_peer(pc, &iphp->rrp);
        }
    }

    /* 到此已经成功选择了后端服务器来处理请求 */

    /* 记录当前后端服务器在后端服务器列表中的位置，该位置方便释放后端服务器调用 */
    iphp->rrp.current = p;

    /* 记录当前后端服务器的地址信息 */
    pc->sockaddr = peer->sockaddr;
    pc->socklen = peer->socklen;
    pc->name = &peer->name;

    if (now - peer->checked > peer->fail_timeout) {
        peer->checked = now;
    }
```

```
    }

    /* ngx_unlock_mutex(iphp->rrp.peers->mutex); */

    /* 把当前后端服务器记录在位图相应的位置 */
    iphp->rrp.tried[n] |= m;
    /* 记录 hash 值 */
    iphp->hash = hash;

    return NGX_OK;
}
```

总结

加权轮询策略：不依赖于客户端的任何信息，完全依靠后端服务器的情况来进行选择。但是同一个客户端的多次请求可能会被分配到不同的后端服务器进行处理，无法满足做会话保持的应用的需求。

IP哈希策略：把同一个 IP 地址的客户端请求分配到同一台服务器处理，避免了加权轮询无法适用会话保持的需求。但是来自同一的 IP 地址的请求比较多时，会导致某台后端服务器的压力可能非常大，而其他后端服务器却空闲的不均衡情况。