# Nginx源代码分析

## Nginx代码的目录和结构

nginx的源码目录结构层次明确，从自动编译脚本到各级的源码，层次都很清晰，是一个大型服务端软件构建的一个范例。以下是源码目录结构说明：

├─auto 自动编译安装相关目录

│ ├─cc 针对各种编译器进行相应的编译配置目录，包括Gcc、Ccc等

│ ├─lib 程序依赖的各种库，包括md5,openssl,pcre等

│ ├─os 针对不同操作系统所做的编译配置目录

│ └─types

├─conf 相关配置文件等目录，包括nginx的配置文件、fcgi相关的配置等

├─contrib

├─html index.html

└─src 源码目录

├─core 核心源码目录，包括定义常用数据结构、体系结构实现等

├─event 封装的事件系统源码目录

├─http http服务器实现目录

├─mail 邮件代码服务器实现目录

├─misc 该目录当前版本只包含google perftools包

└─os nginx对各操作系统下的函数进行封装以及实现核心调用的目录。

## 基本数据结构

### 简单的数据类型

在 core/ngx\_config.h 目录里面定义了基本的数据类型的映射，大部分都映射到c语言自身的数据类型。

typedef intptr\_t        ngx\_int\_t;

typedef uintptr\_t       ngx\_uint\_t;

typedef intptr\_t        ngx\_flag\_t;

其中ngx\_int\_t，nginx\_flag\_t，都映射为intptr\_t； ngx\_uint\_t映射为uintptr\_t。

这两个类型在/usr/include/stdint.h的定义为：

/\* 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;

#endif

所以基本的操作和整形/指针类型的操作类似。

### 字符串的数据类型

nginx对c语言的字符串类型进行了简单的封装， **core/ngx\_string.h/c** 里面包含这些封装的内容。

其中定义了ngx\_str\_t，ngx\_keyval\_t， ngx\_variable\_value\_t这几个基础类型的定义如下：

typedef struct {

    size\_t      len;

    u\_char     \*data;

} 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;

可以看出 ngx\_str\_t 在原有的uchar\* 的基础上加入的字符串长度的附加信息， 初始化使用ngx\_string宏进行，他的定义为：

#define ngx\_string(str)     { sizeof(str) - 1, (u\_char \*) str }

### 内存分配相关

1. 系统功能封装

内存相关的操作主要在 os/unix/ngx\_alloc.{h,c} 和 core/ngx\_palloc.{h,c} 下。

其中 os/unix/ngx\_alloc.{h,c} 封装了最基本的内存分配函数，是对c原有的malloc/free/memalign 等原有的函数的封装,对应的函数为：

* ngx\_alloc 使用malloc分配内存空间
* ngx\_calloc 使用malloc分配内存空间，并且将空间内容初始化为0
* ngx\_memalign 返回基于一个指定的alignment大小的数值为对齐基数的空间
* ngx\_free 对内存的释放操作

1. Nginx的内存池

为了方便系统模块对内存的使用，方便内存的管理，nginx自己实现了进程池的机制来进行内存的分配和释放， 首先nginx会在特定的生命周期帮你统一建立内存池，当需要进行内存分配的时候统一通过内存池中的内存进行分配，最后nginx会在适当的时候释放内存池的资源，开发者只要在需要的时候对内存进行申请即可，不用过多考虑内存的释放等问题，大大提高了开发的效率。

内存池的主要结构为：

//ngx\_palloc.h

struct ngx\_pool\_s {

    ngx\_pool\_data\_t       d;

    size\_t                max;

    ngx\_pool\_t           \*current;

    ngx\_chain\_t          \*chain;

    ngx\_pool\_large\_t     \*large;

    ngx\_pool\_cleanup\_t   \*cleanup;

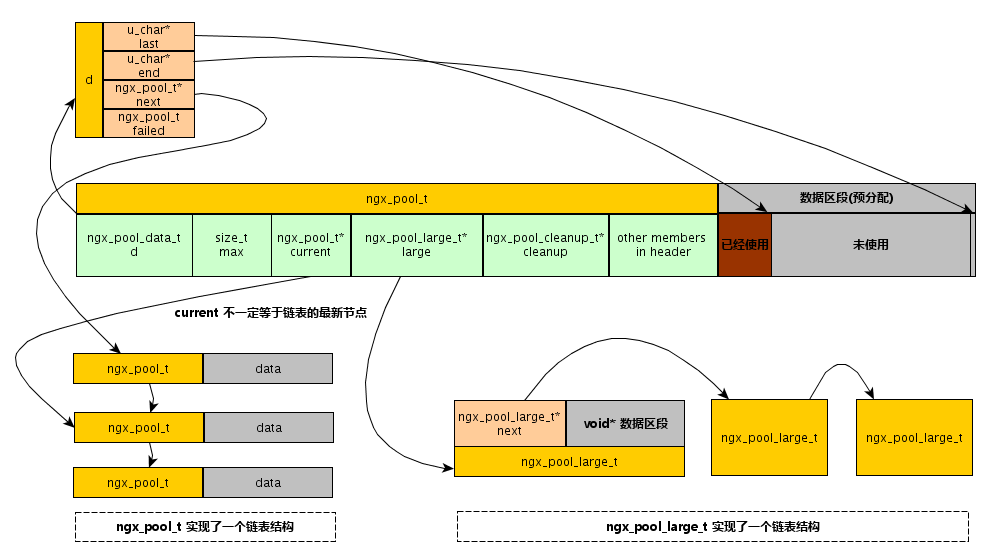
    ngx\_log\_t            \*log;

};

//ngx\_core.h

typedef struct ngx\_pool\_s        ngx\_pool\_t;

typedef struct ngx\_chain\_s       ngx\_chain\_t;



下面解释一下主要的几个操作：

// 创建内存池

ngx\_pool\_t \*ngx\_create\_pool(size\_t size, ngx\_log\_t \*log);

大致的过程是创建使用 ngx\_alloc 分配一个size大小的空间, 然后将 ngx\_pool\_t\* 指向这个空间， 并且初始化里面的成员, 其中

p->d.last = (u\_char \*) p + sizeof(ngx\_pool\_t); // 初始指向 ngx\_pool\_t 结构体后面

p->d.end = (u\_char \*) p + size; // 整个结构的结尾后面

p->max = (size < NGX\_MAX\_ALLOC\_FROM\_POOL) ? size : NGX\_MAX\_ALLOC\_FROM\_POOL; // 最大不超过 NGX\_MAX\_ALLOC\_FROM\_POOL,也就是getpagesize()-1 大小

其他大都设置为null或者0

// 销毁内存池

void ngx\_destroy\_pool(ngx\_pool\_t \*pool);

遍历链表，所有释放内存，其中如果注册了clenup(也是一个链表结构）， 会一次调用clenup 的 handler 进行清理。

// 重置内存池

void ngx\_reset\_pool(ngx\_pool\_t \*pool);

释放所有large段内存， 并且将d->last指针重新指向 ngx\_pool\_t 结构之后（和创建时一样）

// 从内存池里分配内存

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的过程一般为，首先判断待分配的内存是否大于 pool->max的大小，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） ，其他几个函数也基本上为 ngx\_palloc 的变种，实现方式大同小异

// 释放指定的内存

ngx\_int\_t ngx\_pfree(ngx\_pool\_t \*pool, void \*p);

这个操作只有在内存在large链表里注册的内存在会被真正释放，如果分配的是普通的内存，则会在destory\_pool的时候统一释放.

// 注册cleanup回叫函数（结构体）

ngx\_pool\_cleanup\_t \*ngx\_pool\_cleanup\_add(ngx\_pool\_t \*p, size\_t size);

这个过程和我们之前经常使用的有些区别， 他首先在传入的内存池中分配这个结构的空间（包括data段）， 然后将为结构体分配的空间返回， 通过操作返回的ngx\_pool\_cleanup\_t结构来添加回叫的实现。 (这个过程在nginx里面出现的比较多，也就是 xxxx\_add 操作通常不是实际的添加操作，而是分配空间并返回一个指针，后续我们还要通过操作指针指向的空间来实现所谓的add)

### Nginx的基本容器

1. ngx\_array

对应的文件为 core/ngx\_array.{c|h}

ngx\_array是nginx内部封装的使用 ngx\_pool\_t对内存池进行分配的数组容器，其中的数据是在一整片内存区中连续存放的。更新数组时只能在尾部压入1个或多个元素。

数组的实现结构为：

struct ngx\_array\_s {

    void        \*elts;

    ngx\_uint\_t   nelts;

    size\_t       size;

    ngx\_uint\_t   nalloc;

    ngx\_pool\_t  \*pool;

};

其中 elts 为具体的数据区域的指针， nelts 为数组实际包含的元素数量， size为数组单个元素的大小， nalloc为数组容器预先（或者重新）分配的内存大小， pool 为分配基于的内存池

常用的操作有：

// 创建一个新的数组容器

ngx\_array\_t \*ngx\_array\_create(ngx\_pool\_t \*p, ngx\_uint\_t n, size\_t size);

// 销毁数组容器

void ngx\_array\_destroy(ngx\_array\_t \*a);

// 将新的元素加入数组容器

void \*ngx\_array\_push(ngx\_array\_t \*a);

void \*ngx\_array\_push\_n(ngx\_array\_t \*a, ngx\_uint\_t n);　//返回ｎ个元素的指针

这里需要注意的是，和之前的ngx\_pool\_cleanup\_add一样， ngx\_array\_push只是进行内存分配的操作，我们需要对返回的指针指向的地址进行赋值等操作来实现实际数组值的添加。

具体一点的push操作的实现为：

1. 首先判断　nalloc是否和nelts相等，即数组预先分配的空间已经满了，如果没满则计算地址直接返回指针
2. 如果已经满了则先判断是否我们的pool中的当前链表节点还有剩余的空间，如果有则直接在当前的pool链表节点中分配内存，并返回
3. 如果当前链表节点没有足够的空间则使用ngx\_palloc重新分配一个2倍于之前数组空间大小的数组，然后将数据转移过来，并返回新地址的指针
4. ngx\_queue

ngx\_queue.{c,h} 实现了一个队列的操作逻辑，队列的基本结构为一个双向队列

基础的数据结构为：

typedef struct ngx\_queue\_s  ngx\_queue\_t;

struct ngx\_queue\_s {

    ngx\_queue\_t  \*prev;

    ngx\_queue\_t  \*next;

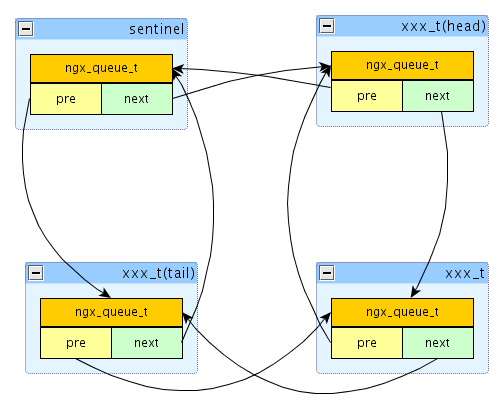
};

注意nginx的队列操作和结构只进行指针的操作，不负责节点内容空间的分配和保存，所以在定义自己的队列节点的时候，需要自己定义数据结构以及分配空间， 并包含一个ngx\_queue\_t类型的成员, 需要获得原始的数据节点的时候需要使用ngx\_queue\_data宏：

#define ngx\_queue\_data(q, type, link)                                         \

    (type \*) ((u\_char \*) q - offsetof(type, link))

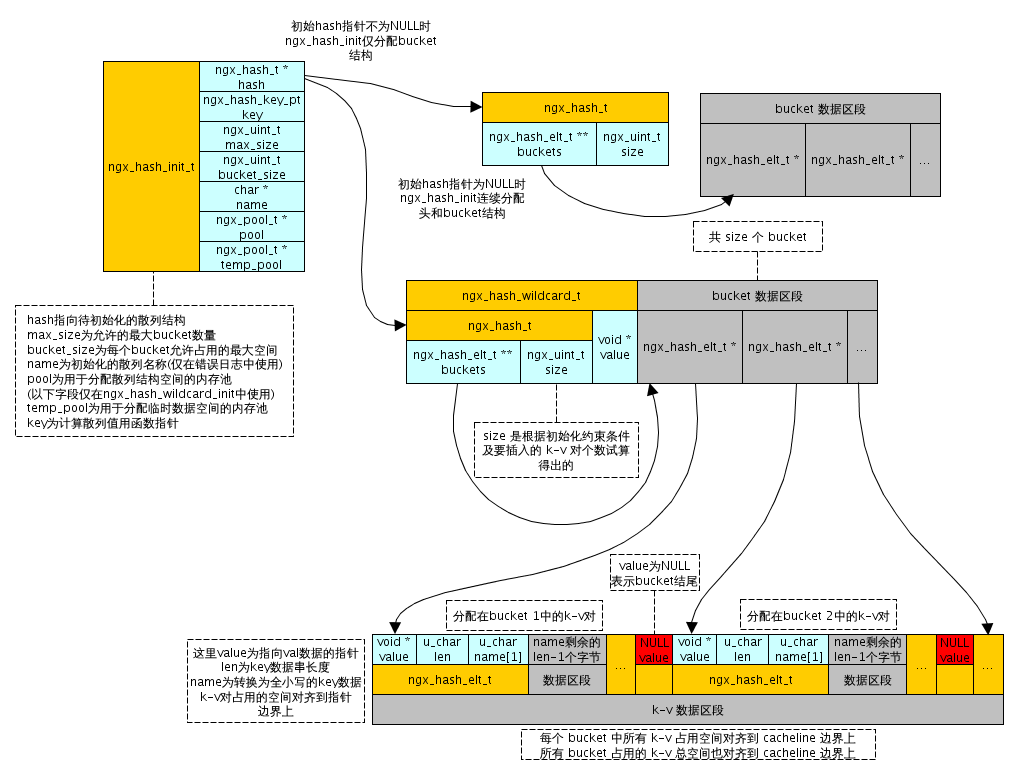
另外，整个queue结构中包含一个 sentinel（哨兵） 节点， 他指向队列的头和尾。



1. ngx\_hash

ngx\_hash.{c|h} 实现了nginx里面比较重要的一个hash结构， 这个在模块配置解析里经常被用到。该 hash 结构是只读的，即仅在初始创建时可以给出保存在其中的 key-val 对，其后就只能查询而不能进行增删改操作了。

下面是简单 hash 结构的内存布局：



虽然代码理解起来比较混乱，但是使用还是比较简单的，常用的有创建 hash 和在 hash 中进行查找两个操作，对于创建hash的操作,过程一般为：

1. 构造一个 ngx\_hash\_key\_t 为成员的数组， 包含 key, value 和 使用key计算出的一个hash值
2. 构建一个 ngx\_hash\_init\_t结构体的变量， 其中包含了ngx\_hash\_t 的成员， 为hash的结构体， 还包括一些其他初始设置，如bucket的大小，内存池等
3. 调用 ngx\_hash\_init 传入 ngx\_hash\_init\_t 结构， ngx\_hash\_key\_t 的数组，和数组的长度， 进行初始化，这样 ngx\_hash\_init\_t的hash成员就是我们要的hash结构

查找的过程很简单

1. 计算 key 的hash值
2. 使用 ngx\_hash\_find 进行查找，需要同时传入 hash值和key ,返回的就是value的指针

需要注意的是，nginx 的 hash 在查找时使用的是分桶后线性查找法，因此当分桶数确定时查找效率同其中的总 key-val 对数量成反比。

1. ngx\_list

ngx\_list 的结构并不复杂，ngx为我们封装了ngx\_list\_create, ngx\_list\_init, 和 ngx\_list\_push等(建立，初始化，添加）操作， 但是对于我们来说最常用的是遍历操作， 下面是nginx的注释里面提到的遍历的例子

   part = &list.part;

   data = part->elts;

   for (i = 0 ;; i++) {

       if (i >= part->nelts) {

           if (part->next == NULL) {

               break;

           }

           part = part->next;

           data = part->elts;

           i = 0;

       }

       ...  data[i] ...

   }

1. ngx\_buf

对应的文件为 core/ngx\_buf.{c|h}

buf分为两种类型,一种是file,一种是memory.因此这里会有文件的一些操作域。

可以看到buf相对于pool多了一个pos域(file\_pos).这里我们要知道我们发送往套接字异或者其他的设备，我们这里会现将数据放到buf中，然后当设备或者套接字准备好了，我们就会从buf中读取，因此这里pos指针就是放到buf中的已经被执行的数据(也就是已经送往套接字)的位置。

struct ngx\_buf\_s {

///pos表示已经执行的数据的位置。

u\_char \*pos;

///last和上面内存池中last一样，也就是使用的内存的最后一个字节的指针

u\_char \*last;

///文件指针

off\_t file\_pos;

off\_t file\_last;

///buf的开始指针

u\_char \*start; /\* start of buffer \*/

u\_char \*end; /\* end of buffer \*/

///这里表示这个buf从属于那个模块。

ngx\_buf\_tag\_t tag;

ngx\_file\_t \*file;

ngx\_buf\_t \*shadow;

///一些标记

/\* the buf's content could be changed \*/

unsigned temporary:1;

///在内存中是不能改变的。

unsigned memory:1;

///是否是mmap的内存

unsigned mmap:1;

unsigned recycled:1;

///是否文件。

unsigned in\_file:1;

unsigned flush:1;

unsigned sync:1;

unsigned last\_buf:1;

unsigned last\_in\_chain:1;

unsigned last\_shadow:1;

unsigned temp\_file:1;

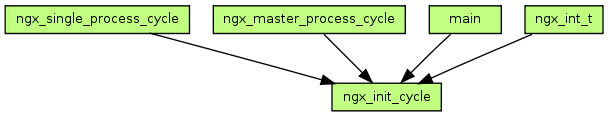
/\* STUB \*/ int num;

};

## nginx的core module 的结构和运行机制

### ngx\_init\_cycle

其中一个比较重要的函数调用是， ngx\_init\_cycle, 这个是使用kscope输出的他的调用关系，他被main, ngx\_master\_process\_cycle,ngx\_single\_process\_cycle 调用， 其中后两者是在reconfigure的时候被调用的



他主要做了如下几件事情:

初始化cycle是基于旧有的cycle进行的，比如这里的 init\_cycle，会继承old cycle的很多属性， 比如log等， 但是同时会对很多资源重新分配,比如pool, shared mem, file handler, listening socket 等，同时清除旧有的cycle的资源

另外，ngx\_master/single\_process\_cycle 里面会对init\_process进行调用， 并且循环调用 ngx\_process\_events\_and\_timers ， 其中里面会调用ngx\_process\_events(cycle, timer, flags); 对事件循环进行polliing 时间一般默认为 500 ms。

Nginx的OS module的结构和运行机制

## nginx的http module 的结构和运行机制

HTTP相关的Module都在 src/http 目录和其子目录下， 其中 src/http 下的文件为http模块的核心文件， src/http/modules 下的文件为http模块的扩展模块。

### ngx\_http.[c|h]

ngx\_http.c 中，注册了 http 这个指令的处理模块，对应ngx\_http\_block函数

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

};

这个函数里面会进行一些conf资源分配/Merge，配置文件解析等工作。 这里面有个一比较重要的工作是注册了nginx http 的 phase handler

    if (ngx\_http\_init\_phase\_handlers(cf, cmcf) != NGX\_OK) {

        return NGX\_CONF\_ERROR;

    }

phase handler的类型在 ngx\_http\_core\_module 这里定义:

typedef enum {

    NGX\_HTTP\_POST\_READ\_PHASE = 0,

    NGX\_HTTP\_SERVER\_REWRITE\_PHASE,

    NGX\_HTTP\_FIND\_CONFIG\_PHASE,

    NGX\_HTTP\_REWRITE\_PHASE,

    NGX\_HTTP\_POST\_REWRITE\_PHASE,

    NGX\_HTTP\_PREACCESS\_PHASE,

    NGX\_HTTP\_ACCESS\_PHASE,

    NGX\_HTTP\_POST\_ACCESS\_PHASE,

    NGX\_HTTP\_TRY\_FILES\_PHASE,

    NGX\_HTTP\_CONTENT\_PHASE,

    NGX\_HTTP\_LOG\_PHASE

} ngx\_http\_phases;

每一个phase的handlers 都是一个数组，里面可以包含多个元素，通过 ngx\_array\_push 添加新的handler。

其中每个phase的处理大都包含了对ngx\_request\_t 的 write 或者 read event的改写，其中在 ngx\_http\_core\_content\_phase 里面， 有对location handler的调用， 其中的 r->content\_handler 就是运行时刻从location handler中注册的，

    if (r->content\_handler) {

        r->write\_event\_handler = ngx\_http\_request\_empty\_handler;

        ngx\_http\_finalize\_request(r, r->content\_handler(r)); /\*实际的请求发送处理\*/

        return NGX\_OK;

    }

其中， 在各个phase的结束阶段，一般都是调用

    r->phase\_handler++;

    return NGX\_AGAIN;

移动request 中 phase\_handler的指针，并且示意主程序继续进行。

这里，无论是phase handler,还是 location handler，我们都是可以在程序里进行注册的。

另外， ngx\_http\_block 里面调用了 ngx\_http\_optimize\_servers ，这个函数对listening和connection相关的变量进行了初始化和调优，并最终在 ngx\_http\_add\_listening （被ngx\_http\_add\_listening调用） 中注册了listening 的 handler 为 ngx\_http\_init\_connection

    ls->handler = ngx\_http\_init\_connection;

ngx\_http\_init\_connection 在 ngx\_http\_request.c中定义，后续会进行详细的介绍。

### ngx\_http\_request.[c|h]

这里面，ngx\_http\_init\_connection 注册了connection事件的读操作的回叫函数， 并将写操作设置为空函数

    rev = c->read;

    rev->handler = ngx\_http\_init\_request;

    c->write->handler = ngx\_http\_empty\_handler;

当新的连接进入的时候，就执行到 ngx\_http\_init\_request, 开始对后面的流程进行处理，主要是将rev的handler 设置为ngx\_http\_process\_request\_line ， 然后ngx\_http\_process\_request\_line 会先后有调度到 ngx\_http\_process\_request\_headers 和 ngx\_http\_process\_request 函数对读取过来的event进行处理，其中， ngx\_http\_process\_request\_headers 里面会对http的请求头进行解析，ngx\_http\_process\_request 设置event handler 到ngx\_http\_request\_handler ，ngx\_http\_request\_handler 中会根据事件可能是读取还是写入的操作分别调用 request 的 read\_event\_handler 和 write\_event\_handler ,所以后续程序对 request 的 read/writeevent\_handler 调整 本质上类似对 rev 和 wev的handler的调整，只是回叫函数的参数变更为了 ngx\_request\_t 而不是之前的ngx\_event\_t

    c->read->handler = ngx\_http\_request\_handler;

    c->write->handler = ngx\_http\_request\_handler;

    r->read\_event\_handler = ngx\_http\_block\_reading;

根据上面代码可以看出， 模块开始使用 ngx\_http\_block\_reading 这个handler对后续的读请求进行处理

在注册完事件后， ngx\_http\_process\_request 会分别调用下面的两个函数

    ngx\_http\_handler(r);

    ngx\_http\_run\_posted\_requests(c);

其中， ngx\_http\_handler 在ngx\_http\_core\_module中定义，处理程序的主请求， ngx\_http\_run\_posted\_requests 在ngx\_http\_request.c 里定义，处理所有提交的子请求数据的输出。

### ngx\_http\_core\_module.[c|h]

对于 ngx\_http\_core\_module 是http 模块中比较重要的模块， 他本身是一个 NGX\_HTTP\_MODULE （不同于ngx\_http\_module, ngx\_http\_module本质上是一个 NGX\_CORE\_MODULE。

这里面对http block下面的一些指令进行了处理， 比如 server, location 等， 同时， 上面提到的 ngx\_http\_handler 也在这里面。

ngx\_http\_handler 所作的最核心的工作就是在最后调用 并将 write event 设置为 ngx\_http\_core\_run\_phases， 开始依次处理各个阶段的 handler。

当handler处理完成后，http的处理流程也就基本上完成了：

    while (ph[r->phase\_handler].checker) {

        rc = ph[r->phase\_handler].checker(r, &ph[r->phase\_handler]);

        if (rc == NGX\_OK) {

            return;

        }

    }

run\_phases 的过程实际上非常简单， 一次的运行每一个handler, 当任意一个handler返回ok或者所有handler执行完成后，整个流程结束。

这里需要注意的是， ph的下标变化是根据 r->phase\_handler 变量决定的， 所以在每个handler内部，如果想要让主程序继续处理下一个 handler,需要手动的 r->phase\_handler++ ,将phase handler数组的下标转移到下一个成员。

### subrequest

在 ngx\_http\_core\_module 里面，我们可以看到一些subrequest的函数，根据evan miller 的文章，我们知道 subrequest是在主请求的基础上发起的子请求，subrequest返回的内容会被附加到自请求上面，他的实现方法就是调用 ngx\_http\_subrequest 函数，subrequest函数的圆形是

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)

在里面，在这个函数里，他会

1)创建新的request变量，根据 主request的值去填充这个变量

2）注册新的变量的write event handler

    sr->read\_event\_handler = ngx\_http\_request\_empty\_handler;

    sr->write\_event\_handler = ngx\_http\_handler;

3) 并且把subrequet 的request 注册到主request的 posted\_requests 变量里

    for (p = &r->main->posted\_requests; \*p; p = &(\*p)->next) { /\* void \*/ }

也就是说，一但调用了 ngx\_http\_subrequest 只后，subrequest已经已经注册到了nginx的事件循环中，和主循环并行进行处理，所以根据evan miller的文章里我们也可以看到，subrequest的串行处理是比较困难的。

### internal redirect

nginx的 internal redirect 有两种方式， 一个是调用 ngx\_http\_internal\_redirect 命令， 一个是使用 X-Accel-Redirect 头，其中X-Accel-Redirect 头是交由upstream模块进行的， 在ngx\_http\_upstream\_process\_headers函数中，我们可以看到

   if (u->headers\_in.x\_accel\_redirect

        && !(u->conf->ignore\_headers & NGX\_HTTP\_UPSTREAM\_IGN\_XA\_REDIRECT))

   {

        .....

        .....

        ngx\_http\_internal\_redirect(r, uri, &args);

        return NGX\_DONE;

    }

也就是说，本质上，这两种方法都是通过 ngx\_http\_internal\_redirect 的函数实现的，而ngx\_http\_internal\_redirect 函数又做了些什么呢？

    r->internal = 1;

    ngx\_http\_handler(r);

    return NGX\_DONE;

可以看到，其实它只是简单的修改request结构体，并通过 ngx\_http\_handler 重新进行http处理流程而已。

关于internal redirect 和 subrequest,他们有类似的地方，都是在最后调用 ngx\_http\_handler 重新开始对request的处理，但是不同的地方我想主要是 subrequest会新建立一个request结构，原来的request处理并不会结束，而 internal redirect会结束当前的请求处理，使用嗯但前的request结构体来发起请求。

### upstream

upstream是nginx里面非常重要的一种处理模式，nginx的很多模块都是使用他来完成的， upstream和普通的handler有很大的不同，比如nginx的location handler 和phase handler, 他们的处理模式非常类似apache的handler,就是获取后同步的进行处理，但是对于有些场合，比如proxy或者fastcgi, nginx进程需要和上游的服务(http,或者php-cgi）进行交互，并且将结果返回给客户端，在交互的过程中，由于后端服务的响应时间各异，如果后端服务的响应时间过长，nginx等待后端服务响应的过程一之同步阻塞在这里是不现实的，所以nginx采用了异步的方案，每当上游服务器有响应的时候才进行处理，并将处理结果返回给前端，这样nginx在处理上游请求的时候，可以同时注册多个io时间，使用epoll\_wait(linux下默认) 也就是nginx的主事件循环进行处理，大大提高了nginx的并发能里，每个进程可以处理多个连接 ，不会因为一个上游连接阻塞整个nginx进程， 当然这也有依赖于你程序的实现方式。

关于upstream的使用方式， evan miller的文章里已经说的很清楚了，就是可以在location handler里（或者其它插入点中） 注册upstream的一些callback

    u->conf = &plcf->upstream;

/\* attach the callback functions \*/

    u->create\_request = ngx\_http\_proxy\_create\_request;

    u->reinit\_request = ngx\_http\_proxy\_reinit\_request;

    u->process\_header = ngx\_http\_proxy\_process\_status\_line;

    u->abort\_request = ngx\_http\_proxy\_abort\_request;

    u->finalize\_request = ngx\_http\_proxy\_finalize\_request;

    r->upstream = u;

并且最后在调用 ngx\_http\_read\_client\_request\_body 中传入 ngx\_http\_upstream\_init 作为callback function 参数初始化 upstream流程, 在 ngx\_http\_upsteam\_init 里面，会调用 ngx\_http\_upstream\_connect建立连接，并注册读写的event handler

        /\* event handler 最终会根据读／或者写的请求调用 write\_event\_handler和 read\_event\_handler ，也就是  ngx\_http\_upstream\_send\_request\_handler和

        ngx\_http\_upstream\_process\_header \*/

    c->write->handler = ngx\_http\_upstream\_handler;

    c->read->handler = ngx\_http\_upstream\_handler;

    u->write\_event\_handler = ngx\_http\_upstream\_send\_request\_handler;

    u->read\_event\_handler = ngx\_http\_upstream\_process\_header;

这里需要注意的是，在处理上游请求的时候，由于是异步的事件，所以每次上游服务器在调用callback的时候并不一定返回的是全部数据，process\_header 函数必须根据判断每次返回的数据包是否完成，如果没有处理完成则返回 NGX\_AGAIN,来指示nginx需要重新进入process\_header函数进行处理，并知道处理完成的时候返回NGX\_OK 来指示nginx完成处理。 具体的例子可以参考 fastcgi 模块的ngx\_http\_fastcgi\_process\_header函数。

另外， nginx可以针对上游服务的响应body进行处理， 通过注册 u->pipe->input\_filter 这个回叫来实现，同样可以参考fastcgi模块，下面是例子

    /\* 这里注册对于响应体的处理 \*/

    u->pipe->input\_filter = ngx\_http\_fastcgi\_input\_filter;

    u->pipe->input\_ctx = r;

通常在处理的过程中需要自己创建状态机来处理请求的不同状态，对于一些请求相关的信息，可以自己创建结构体，通过

#define ngx\_http\_get\_module\_ctx(r, module)  (r)->ctx[module.ctx\_index]

#define ngx\_http\_set\_ctx(r, c, module)      r->ctx[module.ctx\_index] = c;

这两个宏进行读取和存储。