# handler模块(100%)

## handler模块简介

相信大家在看了前一章的模块概述以后，都对nginx的模块有了一个基本的认识。基本上作为第三方开发者最可能开发的就是三种类型的模块，即handler，filter和load-balancer。Handler模块就是接受来自客户端的请求并产生输出的模块。有些地方说upstream模块实际上也是一种handler模块，只不过它产生的内容来自于从后端服务器获取的，而非在本机产生的。

在上一章提到，配置文件中使用location指令可以配置content handler模块，当Nginx系统启动的时候，每个handler模块都有一次机会把自己关联到对应的location上。如果有多个handler模块都关联了同一个location，那么实际上只有一个handler模块真正会起作用。当然大多数情况下，模块开发人员都会避免出现这种情况。

handler模块处理的结果通常有三种情况: 处理成功，处理失败（处理的时候发生了错误）或者是拒绝去处理。在拒绝处理的情况下，这个location的处理就会由默认的handler模块来进行处理。例如，当请求一个静态文件的时候，如果关联到这个location上的一个handler模块拒绝处理，就会由默认的ngx\_http\_static\_module模块进行处理，该模块是一个典型的handler模块。

本章主要讲述的是如何编写handler模块，在研究handler模块编写之前先来了解一下模块的一些基本数据结构。

## 模块的基本结构

在这一节我们将会对通常的模块开发过程中，每个模块所包含的一些常用的部分进行说明。这些部分有些是必须的，有些不是必须的。同时这里所列出的这些东西对于其他类型的模块，例如filter模块等也都是相同的。

### 模块配置结构

基本上每个模块都会提供一些配置指令，以便于用户可以通过配置来控制该模块的行为。那么这些配置信息怎么存储呢？那就需要定义该模块的配置结构来进行存储。

大家都知道Nginx的配置信息分成了几个作用域(scope,有时也称作上下文)，这就是main, server, 以及location。同样的每个模块提供的配置指令也可以出现在这几个作用域里。那对于这三个作用域的配置信息，每个模块就需要定义三个不同的数据结构去进行存储。当然，不是每个模块都会在这三个作用域都提供配置指令的。那么也就不一定每个模块都需要定义三个数据结构去存储这些配置信息了。视模块的实现而言，需要几个就定义几个。

有一点需要特别注意的就是，在模块的开发过程中，我们最好使用nginx原有的命名习惯。这样跟原代码的契合度更高，看起来也更舒服。

对于模块配置信息的定义，命名习惯是ngx\_http<module name>\_(mainloc)\_conf\_t。这里有个例子，就是从我们后面将要展示给大家的hello module中截取的。

typedef struct  
{  
 ngx\_str\_t hello\_string;  
 ngx\_int\_t hello\_counter;  
}ngx\_http\_hello\_loc\_conf\_t;

### 模块配置指令

一个模块的配置指令是定义在一个静态数组中的。同样地，我们来看一下从hello module中截取的模块配置指令的定义。

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  
};

其实看这个定义，就基本能看出来一些信息。例如，我们是定义了两个配置指令，一个是叫hello\_string，可以接受一个参数，或者是没有参数。另外一个命令是hello\_counter，接受一个NGX\_CONF\_FLAG类型的参数。除此之外，似乎看起来有点迷惑。没有关系，我们来详细看一下ngx\_command\_t，一旦我们了解这个结构的详细信息，那么我相信上述这个定义所表达的所有信息就不言自明了。

ngx\_command\_t的定义，位于src/core/ngx\_conf\_file.h中。

struct ngx\_command\_s {  
 ngx\_str\_t name;  
 ngx\_uint\_t type;  
 char \*(\*set)(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf);  
 ngx\_uint\_t conf;  
 ngx\_uint\_t offset;  
 void \*post;  
};

name

: 配置指令的名称。

type

: 该配置的类型，其实更准确一点说，是该配置指令属性的集合。nginx提供了很多预定义的属性值（一些宏定义），通过逻辑或运算符可组合在一起，形成对这个配置指令的详细的说明。下面列出可在这里使用的预定义属性值及说明。

* NGX\_CONF\_NOARGS：配置指令不接受任何参数。
* NGX\_CONF\_TAKE1：配置指令接受1个参数。
* NGX\_CONF\_TAKE2：配置指令接受2个参数。
* NGX\_CONF\_TAKE3：配置指令接受3个参数。
* NGX\_CONF\_TAKE4：配置指令接受4个参数。
* NGX\_CONF\_TAKE5：配置指令接受5个参数。
* NGX\_CONF\_TAKE6：配置指令接受6个参数。
* NGX\_CONF\_TAKE7：配置指令接受7个参数。
* 可以组合多个属性，比如一个指令即可以不填参数，也可以接受1个或者2个参数。那么就是NGX\_CONF\_NOARGSNGX\_CONF\_TAKE2。如果写上面三个属性在一起，你觉得麻烦，那么没有关系，nginx提供了一些定义，使用起来更简洁。
* NGX\_CONF\_TAKE12：配置指令接受1个或者2个参数。
* NGX\_CONF\_TAKE13：配置指令接受1个或者3个参数。
* NGX\_CONF\_TAKE23：配置指令接受2个或者3个参数。
* NGX\_CONF\_TAKE123：配置指令接受1个或者2个或者3参数。
* NGX\_CONF\_TAKE1234：配置指令接受1个或者2个或者3个或者4个参数。
* NGX\_CONF\_1MORE：配置指令接受至少一个参数。
* NGX\_CONF\_2MORE：配置指令接受至少两个参数。
* NGX\_CONF\_MULTI: 配置指令可以接受多个参数，即个数不定。
* NGX\_CONF\_BLOCK：配置指令可以接受的值是一个配置信息块。也就是一对大括号括起来的内容。里面可以再包括很多的配置指令。比如常见的server指令就是这个属性的。
* NGX\_CONF\_FLAG：配置指令可以接受的值是\"on\"或者\"off\"，最终会被转成bool值。
* NGX\_CONF\_ANY：配置指令可以接受的任意的参数值。一个或者多个，或者\"on\"或者\"off\"，或者是配置块。
* 最后要说明的是，无论如何，nginx的配置指令的参数个数不可以超过NGX\_CONF\_MAX\_ARGS个。目前这个值被定义为8，也就是不能超过8个参数值。
* 下面介绍一组说明配置指令可以出现的位置的属性。
* NGX\_DIRECT\_CONF：可以出现在配置文件中最外层。例如已经提供的配置指令daemon，master\_process等。
* NGX\_MAIN\_CONF: http、mail、events、error\_log等。
* NGX\_ANY\_CONF: 该配置指令可以出现在任意配置级别上。
* 对于我们编写的大多数模块而言，都是在处理http相关的事情，也就是所谓的都是NGX\_HTTP\_MODULE，对于这样类型的模块，其配置可能出现的位置也是分为直接出现在http里面，以及其他位置。
* NGX\_HTTP\_MAIN\_CONF: 可以直接出现在http配置指令里。
* NGX\_HTTP\_SRV\_CONF: 可以出现在http里面的server配置指令里。
* NGX\_HTTP\_LOC\_CONF: 可以出现在http server块里面的location配置指令里。
* NGX\_HTTP\_UPS\_CONF: 可以出现在http里面的upstream配置指令里。
* NGX\_HTTP\_SIF\_CONF: 可以出现在http里面的server配置指令里的if语句所在的block中。
* NGX\_HTTP\_LMT\_CONF: 可以出现在http里面的limit\_except指令的block中。
* NGX\_HTTP\_LIF\_CONF: 可以出现在http server块里面的location配置指令里的if语句所在的block中。

set

: 这是一个函数指针，当nginx在解析配置的时候，如果遇到这个配置指令，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);

先看该函数的返回值，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。

再看一下这个函数被调用的时候，传入的三个参数。

* cf: 该参数里面保存从配置文件读取到的原始字符串以及相关的一些信息。特别注意的是这个参数的args字段是一个ngx\_str\_t类型的数组，该数组的首个元素是这个配置指令本身，第二个元素是指令的第一个参数，第三个元素是第二个参数，依次类推。
* cmd: 这个配置指令对应的ngx\_command\_t结构。
* conf: 就是定义的存储这个配置值的结构体，比如在上面展示的那个ngx\_http\_hello\_loc\_conf\_t。当解析这个hello\_string变量的时候，传入的conf就指向一个ngx\_http\_hello\_loc\_conf\_t类型的变量。用户在处理的时候可以使用类型转换，转换成自己知道的类型，再进行字段的赋值。

为了更加方便的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。下面来看一下这些已经实现的set类型函数。

* ngx\_conf\_set\_flag\_slot： 读取NGX\_CONF\_FLAG类型的参数。
* ngx\_conf\_set\_str\_slot:读取字符串类型的参数。
* ngx\_conf\_set\_str\_array\_slot: 读取字符串数组类型的参数。
* ngx\_conf\_set\_keyval\_slot： 读取键值对类型的参数。
* ngx\_conf\_set\_num\_slot: 读取整数类型(有符号整数ngx\_int\_t)的参数。
* ngx\_conf\_set\_size\_slot:读取size\_t类型的参数，也就是无符号数。
* ngx\_conf\_set\_off\_slot: 读取off\_t类型的参数。
* ngx\_conf\_set\_msec\_slot: 读取毫秒值类型的参数。
* ngx\_conf\_set\_sec\_slot: 读取秒值类型的参数。
* ngx\_conf\_set\_bufs\_slot： 读取的参数值是2个，一个是buf的个数，一个是buf的大小。例如： output\_buffers 1 128k;
* ngx\_conf\_set\_enum\_slot: 读取枚举类型的参数，将其转换成整数ngx\_uint\_t类型。
* ngx\_conf\_set\_bitmask\_slot: 读取参数的值，并将这些参数的值以bit位的形式存储。例如：HttpDavModule模块的dav\_methods指令。

conf

: 该字段被NGX\_HTTP\_MODULE类型模块所用 (我们编写的基本上都是NGX\_HTTP\_MOUDLE，只有一些nginx核心模块是非NGX\_HTTP\_MODULE)，该字段指定当前配置项存储的内存位置。实际上是使用哪个内存池的问题。因为http模块对所有http模块所要保存的配置信息，划分了main, server和location三个地方进行存储，每个地方都有一个内存池用来分配存储这些信息的内存。这里可能的值为 NGX\_HTTP\_MAIN\_CONF\_OFFSET、NGX\_HTTP\_SRV\_CONF\_OFFSET或NGX\_HTTP\_LOC\_CONF\_OFFSET。当然也可以直接置为0，就是NGX\_HTTP\_MAIN\_CONF\_OFFSET。

offset

: 指定该配置项值的精确存放位置，一般指定为某一个结构体变量的字段偏移。因为对于配置信息的存储，一般我们都是定义个结构体来存储的。那么比如我们定义了一个结构体A，该项配置的值需要存储到该结构体的b字段。那么在这里就可以填写为offsetof(A, b)。对于有些配置项，它的值不需要保存或者是需要保存到更为复杂的结构中时，这里可以设置为0。

post

: 该字段存储一个指针。可以指向任何一个在读取配置过程中需要的数据，以便于进行配置读取的处理。大多数时候，都不需要，所以简单地设为0即可。

看到这里，应该就比较清楚了。ngx\_http\_hello\_commands这个数组每5个元素为一组，用来描述一个配置项的所有情况。那么如果有多个配置项，只要按照需要再增加5个对应的元素对新的配置项进行说明。

**需要注意的是，就是在ngx\_http\_hello\_commands这个数组定义的最后，都要加一个ngx\_null\_command作为结尾。**

### 模块上下文结构

这是一个ngx\_http\_module\_t类型的静态变量。这个变量实际上是提供一组回调函数指针，这些函数有在创建存储配置信息的对象的函数，也有在创建前和创建后会调用的函数。这些函数都将被nginx在合适的时间进行调用。

typedef struct {  
 ngx\_int\_t (\*preconfiguration)(ngx\_conf\_t \*cf);  
 ngx\_int\_t (\*postconfiguration)(ngx\_conf\_t \*cf);  
  
 void \*(\*create\_main\_conf)(ngx\_conf\_t \*cf);  
 char \*(\*init\_main\_conf)(ngx\_conf\_t \*cf, void \*conf);  
  
 void \*(\*create\_srv\_conf)(ngx\_conf\_t \*cf);  
 char \*(\*merge\_srv\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);  
  
 void \*(\*create\_loc\_conf)(ngx\_conf\_t \*cf);  
 char \*(\*merge\_loc\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);  
} ngx\_http\_module\_t;

preconfiguration

: 在创建和读取该模块的配置信息之前被调用。

postconfiguration

: 在创建和读取该模块的配置信息之后被调用。

create\_main\_conf

: 调用该函数创建本模块位于http block的配置信息存储结构。该函数成功的时候，返回创建的配置对象。失败的话，返回NULL。

init\_main\_conf

: 调用该函数初始化本模块位于http block的配置信息存储结构。该函数成功的时候，返回NGX\_CONF\_OK。失败的话，返回NGX\_CONF\_ERROR或错误字符串。

create\_srv\_conf

: 调用该函数创建本模块位于http server block的配置信息存储结构，每个server block会创建一个。该函数成功的时候，返回创建的配置对象。失败的话，返回NULL。

merge\_srv\_conf

: 因为有些配置指令既可以出现在http block，也可以出现在http server block中。那么遇到这种情况，每个server都会有自己存储结构来存储该server的配置，但是在这种情况下http block中的配置与server block中的配置信息发生冲突的时候，就需要调用此函数进行合并，该函数并非必须提供，当预计到绝对不会发生需要合并的情况的时候，就无需提供。当然为了安全起见还是建议提供。该函数执行成功的时候，返回NGX\_CONF\_OK。失败的话，返回NGX\_CONF\_ERROR或错误字符串。

create\_loc\_conf

: 调用该函数创建本模块位于location block的配置信息存储结构。每个在配置中指明的location创建一个。该函数执行成功，返回创建的配置对象。失败的话，返回NULL。

merge\_loc\_conf

: 与merge\_srv\_conf类似，这个也是进行配置值合并的地方。该函数成功的时候，返回NGX\_CONF\_OK。失败的话，返回NGX\_CONF\_ERROR或错误字符串。

Nginx里面的配置信息都是上下一层层的嵌套的，对于具体某个location的话，对于同一个配置，如果当前层次没有定义，那么就使用上层的配置，否则使用当前层次的配置。

这些配置信息一般默认都应该设为一个未初始化的值，针对这个需求，Nginx定义了一系列的宏定义来代表各种配置所对应数据类型的未初始化值，如下：

#define NGX\_CONF\_UNSET -1  
#define NGX\_CONF\_UNSET\_UINT (ngx\_uint\_t) -1  
#define NGX\_CONF\_UNSET\_PTR (void \*) -1  
#define NGX\_CONF\_UNSET\_SIZE (size\_t) -1  
#define NGX\_CONF\_UNSET\_MSEC (ngx\_msec\_t) -1

又因为对于配置项的合并，逻辑都类似，也就是前面已经说过的，如果在本层次已经配置了，也就是配置项的值已经被读取进来了（那么这些配置项的值就不会等于上面已经定义的那些UNSET的值），就使用本层次的值作为定义合并的结果，否则，使用上层的值，如果上层的值也是这些UNSET类的值，那就赋值为默认值，否则就使用上层的值作为合并的结果。对于这样类似的操作，Nginx定义了一些宏操作来做这些事情，我们来看其中一个的定义。

#define ngx\_conf\_merge\_uint\_value(conf, prev, default) \  
 if (conf == NGX\_CONF\_UNSET\_UINT) { \  
 conf = (prev == NGX\_CONF\_UNSET\_UINT) ? default : prev; \  
 }

显而易见，这个逻辑确实比较简单，所以其它的宏定义也类似，我们就列具其中的一部分吧。

ngx\_conf\_merge\_value  
ngx\_conf\_merge\_ptr\_value  
ngx\_conf\_merge\_uint\_value  
ngx\_conf\_merge\_msec\_value  
ngx\_conf\_merge\_sec\_value

等等。

下面来看一下hello模块的模块上下文的定义，加深一下印象。

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 \*/  
};

**注意：这里并没有提供merge\_loc\_conf函数，因为我们这个模块的配置指令已经确定只出现在NGX\_HTTP\_LOC\_CONF中这一个层次上，不会发生需要合并的情况。**

### 模块的定义

对于开发一个模块来说，我们都需要定义一个ngx\_module\_t类型的变量来说明这个模块本身的信息，从某种意义上来说，这是这个模块最重要的一个信息，它告诉了nginx这个模块的一些信息，上面定义的配置信息，还有模块上下文信息，都是通过这个结构来告诉nginx系统的，也就是加载模块的上层代码，都需要通过定义的这个结构，来获取这些信息。

我们先来看下ngx\_module\_t的定义

typedef struct ngx\_module\_s ngx\_module\_t;  
struct ngx\_module\_s {  
 ngx\_uint\_t ctx\_index;  
 ngx\_uint\_t index;  
 ngx\_uint\_t spare0;  
 ngx\_uint\_t spare1;  
 ngx\_uint\_t abi\_compatibility;  
 ngx\_uint\_t major\_version;  
 ngx\_uint\_t minor\_version;  
 void \*ctx;  
 ngx\_command\_t \*commands;  
 ngx\_uint\_t type;  
 ngx\_int\_t (\*init\_master)(ngx\_log\_t \*log);  
 ngx\_int\_t (\*init\_module)(ngx\_cycle\_t \*cycle);  
 ngx\_int\_t (\*init\_process)(ngx\_cycle\_t \*cycle);  
 ngx\_int\_t (\*init\_thread)(ngx\_cycle\_t \*cycle);  
 void (\*exit\_thread)(ngx\_cycle\_t \*cycle);  
 void (\*exit\_process)(ngx\_cycle\_t \*cycle);  
 void (\*exit\_master)(ngx\_cycle\_t \*cycle);  
 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;  
};  
  
#define NGX\_NUMBER\_MAJOR 3  
#define NGX\_NUMBER\_MINOR 1  
#define NGX\_MODULE\_V1 0, 0, 0, 0, \  
 NGX\_DSO\_ABI\_COMPATIBILITY, NGX\_NUMBER\_MAJOR, NGX\_NUMBER\_MINOR  
#define NGX\_MODULE\_V1\_PADDING 0, 0, 0, 0, 0, 0, 0, 0

再看一下hello模块的模块定义。

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  
};

模块可以提供一些回调函数给nginx，当nginx在创建进程线程或者结束进程线程时进行调用。但大多数模块在这些时刻并不需要做什么，所以都简单赋值为NULL。

## handler模块的基本结构

除了上一节介绍的模块的基本结构以外，handler模块必须提供一个真正的处理函数，这个函数负责对来自客户端请求的真正处理。这个函数的处理，既可以选择自己直接生成内容，也可以选择拒绝处理，由后续的handler去进行处理，或者是选择丢给后续的filter进行处理。来看一下这个函数的原型申明。

typedef ngx\_int\_t (\*ngx\_http\_handler\_pt)(ngx\_http\_request\_t \*r);

r是http请求。里面包含请求所有的信息，这里不详细说明了，可以参考别的章节的介绍。 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好了，否则返回NGX\_ERROR就发生错误了。

## handler模块的挂载

handler模块真正的处理函数通过两种方式挂载到处理过程中，一种方式就是按处理阶段挂载；另外一种挂载方式就是按需挂载。

### 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。他们从前到后，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，它不调用挂载地任何的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

所以其实真正是有7个phase你可以去挂载handler。

挂载的代码如下（摘自hello module）:

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;  
}

使用这种方式挂载的handler也被称为 **content phase handlers**。

### 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阶段的时候，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。

换句话说，当某个location处理到NGX\_HTTP\_CONTENT\_PHASE阶段时，如果有content handler模块，那么NGX\_HTTP\_CONTENT\_PHASE挂载的所有content phase handlers都不会被执行了。

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阶段才能执行到。如果你想自己的handler在更早的阶段执行，那就不要使用这种挂载方式。

那么在什么情况会使用这种方式来挂载呢？一般情况下，某个模块对某个location进行了处理以后，发现符合自己处理的逻辑，而且也没有必要再调用NGX\_HTTP\_CONTENT\_PHASE阶段的其它handler进行处理的时候，就动态挂载上这个handler。

下面来看一下使用这种挂载方式的具体例子（摘自Emiller\'s Guide To Nginx Module Development）。

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的编写步骤

好，到了这里，让我们稍微整理一下思路，回顾一下实现一个handler的步骤:

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。

看起来不是那么难，对吧？还是那句老话，世上无难事，只怕有心人! 现在我们来完整的分析前面提到的hello handler module示例的功能和代码。

## 示例: hello handler 模块

在前面已经看到了这个hello handler module的部分重要的结构。该模块提供了2个配置指令，仅可以出现在location指令的作用域中。这两个指令是hello\_string, 该指令接受一个参数来设置显示的字符串。如果没有跟参数，那么就使用默认的字符串作为响应字符串。

另一个指令是hello\_counter，如果设置为on，则会在响应的字符串后面追加Visited Times:的字样，以统计请求的次数。

这里有两点注意一下：

1. 对于flag类型的配置指令，当值为off的时候，使用ngx\_conf\_set\_flag\_slot函数，会转化为0，为on，则转化为非0。
2. 另外一个是，我提供了merge\_loc\_conf函数，但是却没有设置到模块的上下文定义中。这样有一个缺点，就是如果一个指令没有出现在配置文件中的时候，配置信息中的值，将永远会保持在create\_loc\_conf中的初始化的值。那如果，在类似create\_loc\_conf这样的函数中，对创建出来的配置信息的值，没有设置为合理的值的话，后面用户又没有配置，就会出现问题。

下面来完整的给出ngx\_http\_hello\_module模块的完整代码。

#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;  
}

通过上面一些介绍，我相信大家都能对整个示例模块有一个比较好的理解。唯一可能感觉有些理解困难的地方在于ngx\_http\_hello\_handler函数里面产生和设置输出。但其实大家在本书的前面的相关章节都可以看到对ngx\_buf\_t和request等相关数据结构的说明。如果仔细看了这些地方的说明的话，应该对这里代码的实现就比较容易理解了。因此，这里不再赘述解释。

## handler模块的编译和使用

模块的功能开发完了之后，模块的使用还需要编译才能够执行，下面我们来看下模块的编译和使用。

### config文件的编写

对于开发一个模块，我们是需要把这个模块的C代码组织到一个目录里，同时需要编写一个config文件。这个config文件的内容就是告诉nginx的编译脚本，该如何进行编译。我们来看一下hello handler module的config文件的内容，然后再做解释。

ngx\_addon\_name=ngx\_http\_hello\_module  
HTTP\_MODULES="$HTTP\_MODULES ngx\_http\_hello\_module"  
NGX\_ADDON\_SRCS="$NGX\_ADDON\_SRCS $ngx\_addon\_dir/ngx\_http\_hello\_module.c"

其实文件很简单，几乎不需要做什么解释。大家一看都懂了。唯一需要说明的是，如果这个模块的实现有多个源文件，那么都在NGX\_ADDON\_SRCS这个变量里，依次写进去就可以。

### 编译

对于模块的编译，nginx并不像apache一样，提供了单独的编译工具，可以在没有apache源代码的情况下来单独编译一个模块的代码。nginx必须去到nginx的源代码目录里，通过configure指令的参数，来进行编译。下面看一下hello module的configure指令：

./configure --prefix=/usr/local/nginx-1.3.1 --add-module=/home/jizhao/open\_source/book\_module

我写的这个示例模块的代码和config文件都放在/home/jizhao/open\_source/book\_module这个目录下。所以一切都很明了，也没什么好说的了。

### 使用

使用一个模块需要根据这个模块定义的配置指令来做。比如我们这个简单的hello handler module的使用就很简单。在我的测试服务器的配置文件里，就是在http里面的默认的server里面加入如下的配置：

location /test {  
 hello\_string jizhao;  
 hello\_counter on;  
}

当我们访问这个地址的时候, lynx <http://127.0.0.1/test的时候，就可以看到返回的结果>。

jizhao Visited Times:1

当然你访问多次，这个次数是会增加的。

## 更多handler模块示例分析

### http access module

该模块的代码位于src/http/modules/ngx\_http\_access\_module.c中。该模块的作用是提供对于特定host的客户端的访问控制。可以限定特定host的客户端对于服务端全部，或者某个server，或者是某个location的访问。 该模块的实现非常简单，总共也就只有几个函数。

static ngx\_int\_t ngx\_http\_access\_handler(ngx\_http\_request\_t \*r);  
static ngx\_int\_t ngx\_http\_access\_inet(ngx\_http\_request\_t \*r,  
 ngx\_http\_access\_loc\_conf\_t \*alcf, in\_addr\_t addr);  
#if (NGX\_HAVE\_INET6)  
static ngx\_int\_t ngx\_http\_access\_inet6(ngx\_http\_request\_t \*r,  
 ngx\_http\_access\_loc\_conf\_t \*alcf, u\_char \*p);  
#endif  
static ngx\_int\_t ngx\_http\_access\_found(ngx\_http\_request\_t \*r, ngx\_uint\_t deny);  
static char \*ngx\_http\_access\_rule(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd,  
 void \*conf);  
static void \*ngx\_http\_access\_create\_loc\_conf(ngx\_conf\_t \*cf);  
static char \*ngx\_http\_access\_merge\_loc\_conf(ngx\_conf\_t \*cf,  
 void \*parent, void \*child);  
static ngx\_int\_t ngx\_http\_access\_init(ngx\_conf\_t \*cf);

对于与配置相关的几个函数都不需要做解释了，需要提一下的是函数ngx\_http\_access\_init，该函数在实现上把本模块挂载到了NGX\_HTTP\_ACCESS\_PHASE阶段的handler上，从而使自己的被调用时机发生在了NGX\_HTTP\_CONTENT\_PHASE等阶段前。因为进行客户端地址的限制检查，根本不需要等到这么后面。

另外看一下这个模块的主处理函数ngx\_http\_access\_handler。这个函数的逻辑也非常简单，主要是根据客户端地址的类型，来分别选择ipv4类型的处理函数ngx\_http\_access\_inet还是ipv6类型的处理函数ngx\_http\_access\_inet6。

而这个两个处理函数内部也非常简单，就是循环检查每个规则，检查是否有匹配的规则，如果有就返回匹配的结果，如果都没有匹配，就默认拒绝。

### http static module

从某种程度上来说，此模块可以算的上是"最正宗的"，"最古老"的content handler。因为本模块的作用就是读取磁盘上的静态文件，并把文件内容作为产生的输出。在Web技术发展的早期，只有静态页面，没有服务端脚本来动态生成HTML的时候。恐怕开发个Web服务器的时候，第一个要开发就是这样一个content handler。

http static module的代码位于src/http/modules/ngx\_http\_static\_module.c中，总共只有两百多行近三百行。可以说是非常短小。

我们首先来看一下该模块的模块上下文的定义。

ngx\_http\_module\_t ngx\_http\_static\_module\_ctx = {  
 NULL, /\* preconfiguration \*/  
 ngx\_http\_static\_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 \*/  
};

是非常的简洁吧，连任何与配置相关的函数都没有。对了，因为该模块没有提供任何配置指令。大家想想也就知道了，这个模块做的事情实在是太简单了，也确实没什么好配置的。唯一需要调用的函数是一个ngx\_http\_static\_init函数。好了，来看一下这个函数都干了些什么。

static ngx\_int\_t  
ngx\_http\_static\_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\_static\_handler;  
  
 return NGX\_OK;  
}

仅仅是挂载这个handler到NGX\_HTTP\_CONTENT\_PHASE处理阶段。简单吧？

下面我们就看一下这个模块最核心的处理逻辑所在的ngx\_http\_static\_handler函数。该函数大概占了这个模块代码量的百分之八九十。

static ngx\_int\_t  
ngx\_http\_static\_handler(ngx\_http\_request\_t \*r)  
{  
 u\_char \*last, \*location;  
 size\_t root, len;  
 ngx\_str\_t path;  
 ngx\_int\_t rc;  
 ngx\_uint\_t level;  
 ngx\_log\_t \*log;  
 ngx\_buf\_t \*b;  
 ngx\_chain\_t out;  
 ngx\_open\_file\_info\_t of;  
 ngx\_http\_core\_loc\_conf\_t \*clcf;  
  
 if (!(r->method & (NGX\_HTTP\_GET|NGX\_HTTP\_HEAD|NGX\_HTTP\_POST))) {  
 return NGX\_HTTP\_NOT\_ALLOWED;  
 }  
  
 if (r->uri.data[r->uri.len - 1] == '/') {  
 return NGX\_DECLINED;  
 }  
  
 log = r->connection->log;  
  
 /\*  
 \* ngx\_http\_map\_uri\_to\_path() allocates memory for terminating '\0'  
 \* so we do not need to reserve memory for '/' for possible redirect  
 \*/  
  
 last = ngx\_http\_map\_uri\_to\_path(r, &path, &root, 0);  
 if (last == NULL) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 path.len = last - path.data;  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, log, 0,  
 "http filename: \"%s\"", path.data);  
  
 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);  
  
 ngx\_memzero(&of, sizeof(ngx\_open\_file\_info\_t));  
  
 of.read\_ahead = clcf->read\_ahead;  
 of.directio = clcf->directio;  
 of.valid = clcf->open\_file\_cache\_valid;  
 of.min\_uses = clcf->open\_file\_cache\_min\_uses;  
 of.errors = clcf->open\_file\_cache\_errors;  
 of.events = clcf->open\_file\_cache\_events;  
  
 if (ngx\_http\_set\_disable\_symlinks(r, clcf, &path, &of) != NGX\_OK) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 if (ngx\_open\_cached\_file(clcf->open\_file\_cache, &path, &of, r->pool)  
 != NGX\_OK)  
 {  
 switch (of.err) {  
  
 case 0:  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
  
 case NGX\_ENOENT:  
 case NGX\_ENOTDIR:  
 case NGX\_ENAMETOOLONG:  
  
 level = NGX\_LOG\_ERR;  
 rc = NGX\_HTTP\_NOT\_FOUND;  
 break;  
  
 case NGX\_EACCES:  
#if (NGX\_HAVE\_OPENAT)  
 case NGX\_EMLINK:  
 case NGX\_ELOOP:  
#endif  
  
 level = NGX\_LOG\_ERR;  
 rc = NGX\_HTTP\_FORBIDDEN;  
 break;  
  
 default:  
  
 level = NGX\_LOG\_CRIT;  
 rc = NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 break;  
 }  
  
 if (rc != NGX\_HTTP\_NOT\_FOUND || clcf->log\_not\_found) {  
 ngx\_log\_error(level, log, of.err,  
 "%s \"%s\" failed", of.failed, path.data);  
 }  
  
 return rc;  
 }  
  
 r->root\_tested = !r->error\_page;  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, log, 0, "http static fd: %d", of.fd);  
  
 if (of.is\_dir) {  
  
 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, log, 0, "http dir");  
  
 ngx\_http\_clear\_location(r);  
  
 r->headers\_out.location = ngx\_palloc(r->pool, sizeof(ngx\_table\_elt\_t));  
 if (r->headers\_out.location == NULL) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 len = r->uri.len + 1;  
  
 if (!clcf->alias && clcf->root\_lengths == NULL && r->args.len == 0) {  
 location = path.data + clcf->root.len;  
  
 \*last = '/';  
  
 } else {  
 if (r->args.len) {  
 len += r->args.len + 1;  
 }  
  
 location = ngx\_pnalloc(r->pool, len);  
 if (location == NULL) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 last = ngx\_copy(location, r->uri.data, r->uri.len);  
  
 \*last = '/';  
  
 if (r->args.len) {  
 \*++last = '?';  
 ngx\_memcpy(++last, r->args.data, r->args.len);  
 }  
 }  
  
 /\*  
 \* we do not need to set the r->headers\_out.location->hash and  
 \* r->headers\_out.location->key fields  
 \*/  
  
 r->headers\_out.location->value.len = len;  
 r->headers\_out.location->value.data = location;  
  
 return NGX\_HTTP\_MOVED\_PERMANENTLY;  
 }  
  
#if !(NGX\_WIN32) /\* the not regular files are probably Unix specific \*/  
  
 if (!of.is\_file) {  
 ngx\_log\_error(NGX\_LOG\_CRIT, log, 0,  
 "\"%s\" is not a regular file", path.data);  
  
 return NGX\_HTTP\_NOT\_FOUND;  
 }  
  
#endif  
  
 if (r->method & NGX\_HTTP\_POST) {  
 return NGX\_HTTP\_NOT\_ALLOWED;  
 }  
  
 rc = ngx\_http\_discard\_request\_body(r);  
  
 if (rc != NGX\_OK) {  
 return rc;  
 }  
  
 log->action = "sending response to client";  
  
 r->headers\_out.status = NGX\_HTTP\_OK;  
 r->headers\_out.content\_length\_n = of.size;  
 r->headers\_out.last\_modified\_time = of.mtime;  
  
 if (ngx\_http\_set\_content\_type(r) != NGX\_OK) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 if (r != r->main && of.size == 0) {  
 return ngx\_http\_send\_header(r);  
 }  
  
 r->allow\_ranges = 1;  
  
 /\* we need to allocate all before the header would be sent \*/  
  
 b = ngx\_pcalloc(r->pool, sizeof(ngx\_buf\_t));  
 if (b == NULL) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 b->file = ngx\_pcalloc(r->pool, sizeof(ngx\_file\_t));  
 if (b->file == NULL) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 rc = ngx\_http\_send\_header(r);  
  
 if (rc == NGX\_ERROR || rc > NGX\_OK || r->header\_only) {  
 return rc;  
 }  
  
 b->file\_pos = 0;  
 b->file\_last = of.size;  
  
 b->in\_file = b->file\_last ? 1: 0;  
 b->last\_buf = (r == r->main) ? 1: 0;  
 b->last\_in\_chain = 1;  
  
 b->file->fd = of.fd;  
 b->file->name = path;  
 b->file->log = log;  
 b->file->directio = of.is\_directio;  
  
 out.buf = b;  
 out.next = NULL;  
  
 return ngx\_http\_output\_filter(r, &out);  
}

首先是检查客户端的http请求类型（r->method），如果请求类型为NGX\_HTTP\_GETNGX\_HTTP\_POST，则继续进行处理，否则一律返回NGX\_HTTP\_NOT\_ALLOWED从而拒绝客户端的发起的请求。

其次是检查请求的url的结尾字符是不是斜杠'/'，如果是说明请求的不是一个文件，给后续的handler去处理，比如后续的ngx\_http\_autoindex\_handler（如果是请求的是一个目录下面，可以列出这个目录的文件），或者是ngx\_http\_index\_handler（如果请求的路径下面有个默认的index文件，直接返回index文件的内容）。

然后接下来调用了一个ngx\_http\_map\_uri\_to\_path函数，该函数的作用是把请求的http协议的路径转化成一个文件系统的路径。

然后根据转化出来的具体路径，去打开文件，打开文件的时候做了2种检查，一种是，如果请求的文件是个symbol link，根据配置，是否允许符号链接，不允许返回错误。还有一个检查是，如果请求的是一个名称，是一个目录的名字，也返回错误。如果都没有错误，就读取文件，返回内容。其实说返回内容可能不是特别准确，比较准确的说法是，把产生的内容传递给后续的filter去处理。

### http log module

该模块提供了对于每一个http请求进行记录的功能，也就是我们见到的access.log。当然这个模块对于log提供了一些配置指令，使得可以比较方便的定制access.log。

这个模块的代码位于src/http/modules/ngx\_http\_log\_module.c，虽然这个模块的代码有接近1400行，但是主要的逻辑在于对日志本身格式等细节的处理。我们在这里进行分析主要是关注，如何编写一个log handler的问题。

由于log handler的时候，拿到的参数也是request这个东西，那么也就意味着我们如果需要，可以好好研究下这个结构，把我们需要的所有信息都记录下来。

对于log handler，有一点特别需要注意的就是，log handler是无论如何都会被调用的，就是只要服务端接受到了一个客户端的请求，也就是产生了一个request对象，那么这些个log handler的处理函数都会被调用的，就是在释放request的时候被调用的（ngx\_http\_free\_request函数）。

那么当然绝对不能忘记的就是log handler最好，也是建议被挂载在NGX\_HTTP\_LOG\_PHASE阶段。因为挂载在其他阶段，有可能在某些情况下被跳过，而没有执行到，导致你的log模块记录的信息不全。

还有一点要说明的是，由于nginx是允许在某个阶段有多个handler模块存在的，根据其处理结果，确定是否要调用下一个handler。但是对于挂载在NGX\_HTTP\_LOG\_PHASE阶段的handler，则根本不关注这里handler的具体处理函数的返回值，所有的都被调用。如下，位于src/http/ngx\_http\_request.c中的ngx\_http\_log\_request函数。

static void  
ngx\_http\_log\_request(ngx\_http\_request\_t \*r)  
{  
 ngx\_uint\_t i, n;  
 ngx\_http\_handler\_pt \*log\_handler;  
 ngx\_http\_core\_main\_conf\_t \*cmcf;  
  
 cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);  
  
 log\_handler = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.elts;  
 n = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.nelts;  
  
 for (i = 0; i < n; i++) {  
 log\_handler[i](r);  
 }  
}