Redpill Linpro

Extending functionality in nginx, with modules!

2012-01-31

Redpill Linpro

What is nginx?

- The 2nd most used HTTP-server (Apache is 1st)
- Asynchronous event-driven approach to request-handling.
- Written in C
- Readable source code!

Redpill Linpro

What kind of modules do we have?

- handler modules - they process a request and produce output
- filter modules - they manipulate the output produced by a handler
- load-balancers - they choose a backend server to send a request to, when more than one backend server is eligible

Redpill Linpro

What will you learn through the course of this presentation?

- Step 0: Building Nginx (with a module)
- Step 1: Create a stub http content module, aka "Hello, World"
- Step 2: Update the module to use an external library for content creation
- Step 3: Give your module its own configuration directives
- Step 4: Fetching arguments from the URL

Redpill Linpro

Let's get started

Step 0: Building Nginx (with a module)

Redpill Linpro

Download & build

```
# wget http://nginx.org/download/nginx-1.0.11.tar.gz
# tar zxvf nginx-1.0.11.tar.gz
# cd nginx-1.0.11.tar.gz
# ./configure
# make
```

Straight forward. No surprises there.

Redpill Linpro

Building a third party module

Modules are statically linked, so adding a new module requires a new nginx-build.

```
# ./configure --add-module=$HOME/git/some_cool_module_I_found_somewhere_on_github
# make
```

Easy. (Want more modules? --add-module can be added several times)

Redpill Linpro

This is boring, show me the code!

Step 1: Create a stub http content module, aka "Hello, World"

Creating your own module

A minimal HTTP module - let's call it "fun" - consists of a directory with two files:

```
config
```

and a source file

```
ngx_http_fun_module.c
```

A significant amount of the source-file will be stub-code.

The config file

Is sourced as a shell-script, and should set different environment variables to help the build-system understand what to do.

A minimal config-file for a module without any external dependencies could look like this:

```
ngx_addon_name=ngx_http_fun_module
HTTP_MODULES="$HTTP_MODULES ngx_http_fun_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_fun_module.c"
```

Redpill Linpro

The source file (1)

Include your headers.

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

Declare a handler.

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

Create our HTTP body.

```
static u_char ngx_fun_string[] = "This is fun!";
```

How does nginx know about my configuration directives?

Quoting the nginx source-code:

```
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;
};

#define ngx_null_command  { ngx_null_string, 0, NULL, 0, 0, NULL }

typedef struct ngx_command_s     ngx_command_t;
```

An array of ngx_command_t's, terminated by a ngx_null_command should be defined in your module.

Redpill Linpro

Declaring the configuration parameters available by your module

Create a static array of the ngx_command_t-type, terminate it with ngx_null_command. Populate it with the commands available for your module.

```
static ngx_command_t ngx_http_fun_commands[] = {
    { // Our command is named "fun":
      ngx_string("fun"),

      // The directive may be specified in the location-level of your nginx-config.
      // The directive does not take any arguments (NGX_CONF_NOARGS)
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,

      // A pointer to our handler-function.
      ngx_http_fun,

      // We're not using these two, they're related to the configuration structures.
      0, 0,

      // A pointer to a post-processing handler.  We're not using any here.
      NULL },
    ngx_null_command
};
```

Declaring the module context

Create a static ngx_http_module_t, populate all 8 elements with NULL.

```
static ngx_http_module_t ngx_http_fun_module_ctx = {
    NULL,                          // preconfiguration
    NULL,                          // 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
};
```

This is used to determine multiple levels of configuration, and gives you control over which directives beat which in a race. Our module does not use any configuration parameters other than "fun;", so we don't need any handlers.

Redpill Linpro

Declaring the module description structure

Create a (not static) structure of type ngx_module_t, reference the structures we've made like this:

```
ngx_module_t ngx_http_fun_module = {
    NGX_MODULE_V1,
    &ngx_http_fun_module_ctx,       // module context
    ngx_http_fun_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
};
```

Redpill Linpro

Creating the actual handler (1)

A static function that returns an ngx_int_t, and receives a pointer to a ngx_http_request_t.

```
static ngx_int_t
ngx_http_fun_handler(ngx_http_request_t * r)
{
    ngx_int_t    rc;
    ngx_buf_t    * b;
    ngx_chain_t  out;
```

We use rc to store the return value of certain function calls.
We use * b to store our buffer pointer.
out is the buffer chain.

Redpill Linpro

Creating the actual handler (2)

A simple test of the kind of request we're receiving.

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

Redpill Linpro

Creating the actual handler (3)

The request-body is useful if we're dealing with POST. We're not.

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

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

This attempts to detach the request body from the request. It's an optimization.

Redpill Linpro

Creating the actual handler (4)

Setting the content-type is important --

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

Redpill Linpro

Creating the actual handler (5)

```
// 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 = sizeof(ngx_fun_string) - 1;

    return ngx_http_send_header(r);
}
```

Redpill Linpro

Creating the actual handler (6)

```
// 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;
}
```

Redpill Linpro

Creating the actual handler (7)

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

Redpill Linpro

Creating the actual handler (8)

```
// adjust the pointers of the buffer
b->pos = ngx_fun_string;
b->last = ngx_fun_string + sizeof(ngx_fun_string) - 1;
b->memory = 1;    // This buffer is in read-only memory
                  // This means that filters should copy it, and not try to rewrite in place.
b->last_buf = 1;  // this is the last buffer in the buffer chain
```

Redpill Linpro

Creating the actual handler (9)

```
// set the status line
r->headers_out.status = NGX_HTTP_OK;
r->headers_out.content_length_n = sizeof(ngx_fun_string) - 1;
```

Creating the actual handler (10)

```
    // 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);
}
```

... almost done!

Redpill Linpro

Attaching the handler

```
static char *
ngx_http_fun(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_fun_handler; // handler to process the 'fun' directive

    return NGX_CONF_OK;
}
```

In this function, we can validate on the incoming ngx_command_t, and attach different handlers.

Redpill Linpro

Does this really work?

Trygve will attempt to demonstrate.

```
# make
# sudo make install
```

Update the nginx-configuration with a location handler, something like this:

```
Location: /fun {
    fun;
}
```

Start nginx, and access http://localhost/fun/

Of course it worked! But static text isn't any fun

Step 2: Update the module to use an external library for content creation

Redpill Linpro

What are we going to do? Tell me now!

- We're going to replace the static text with an image, dynamically created with libcairo.
- We're going to learn about buffers and chains

Redpill Linpro

The config file - dependency handling

Testing for dependencies is done with the feature tests provided by nginx

```
ngx_feature="cairo"
ngx_feature_name=
ngx_feature_run=no
ngx_feature_incs="#include <cairo.h>"
ngx_feature_path="/usr/include/cairo"
ngx_feature_libs=-lcairo
ngx_feature_test="cairo_version()"
. auto/feature

if [ $ngx_found = yes ]; then
    ngx_addon_name=ngx_http_fun_module
    HTTP_MODULES="$HTTP_MODULES ngx_http_fun_module"
    NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_fun_module.c"
    CORE_LIBS="$CORE_LIBS `pkg-config cairo cairo-png --libs`"
    CFLAGS="$CFLAGS `pkg-config cairo cairo-png --cflags`"
else
    cat << END
$0: error: the fun module requires the cairo library.
END
    exit 1
fi
```

The Cairo part (1)

Include the header, and define M_PI

```
#include <cairo.h>

#define M_PI 3.14159265

struct closure {
    ngx_http_request_t * r;
    ngx_chain_t * chain;
    uint32_t length;
};
```

The struct will be used by a callback function we're going to create later.

Throw away ngx_fun_string while you're at it, we're not going to be using that.

Redpill Linpro

The Cairo part (2)

In ngx_http_fun_handler(), remove the ngx_buf_t (we'll do these things in a callback function later), and add our struct.

```
static ngx_int_t
ngx_http_fun_handler(ngx_http_request_t * r)
{
    ngx_int_t   rc;
    ngx_chain_t out;
    struct closure c = { r, &out, 0 };
```

Redpill Linpro

The Cairo part (3)

Remove our header-handling from before - we won't be able to calculate content-length before we've created the png.

```
// set the 'Content-type' header
r->headers_out.content_type_len = sizeof("text/html") - 1;
r->headers_out.content_type.len = sizeof("text/html") - 1;
r->headers_out.content_type.data = (u_char * ) "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 = sizeof(ngx_fun_string) - 1;
    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;
}
```

Redpill Linpro

The Cairo part (4)

This is basically ripped from one of their many examples on their web page.

```
cairo_surface_t * surface;
cairo_t * cr;

surface = cairo_image_surface_create (CAIRO_FORMAT_ARGB32, 256, 256);
cr = cairo_create (surface);

double xc = 128.0;
double yc = 128.0;
double radius = 100.0;
double angle1 = 270.0 * (M_PI/180.0);  // angles are specified
double angle2 = 180.0 * (M_PI/180.0);  // in radians
```

Redpill Linpro

The Cairo part (5)

```
cairo_set_line_width (cr, 10.0);
cairo_arc (cr, xc, yc, radius, angle1, angle2);
cairo_stroke (cr);

// draw helping lines
cairo_set_source_rgba (cr, 1, 0.2, 0.2, 0.6);
cairo_set_line_width (cr, 6.0);

cairo_arc (cr, xc, yc, 10.0, 0, 2*M_PI);
cairo_fill (cr);

cairo_arc (cr, xc, yc, radius, angle1, angle1);
cairo_line_to (cr, xc, yc);
cairo_arc (cr, xc, yc, radius, angle2, angle2);
cairo_line_to (cr, xc, yc);
cairo_stroke (cr);
```

Redpill Linpro

The Cairo part (6)

Make sure that our buffer chain is NULL'ed. Remove any references to the ngx_buf_t from earlier.

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

// Copy the png image to our buffer chain (we provide our own callback-function)
rc = cairo_surface_write_to_png_stream(surface, copy_png_to_chain, &c);

// Free cairo stuff.
cairo_destroy(cr);
cairo_surface_destroy(surface);

// If we for some reason didn't manage to copy the png to our buffer, throw 503.
if ( rc != CAIRO_STATUS_SUCCESS )
{
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}
```

The cairo_surface_write_to_png_stream uses a callback-function to copy the png-data to a buffer, in the way we want. Remember to free resources, and check that everything went well.

Redpill Linpro

The Cairo part (7)

We're going to set our headers now. New content-type - the length has been calculated by our callback function and is stored in our struct.

```
// set the 'Content-type' header
r->headers_out.content_type_len = sizeof("image/png") - 1;
r->headers_out.content_type.data = (u_char * ) "image/png";
// set the status line
r->headers_out.status = NGX_HTTP_OK;
r->headers_out.content_length_n = c.length;

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

// We've added the NGX_HTTP_HEAD check here, because we're unable to set content length before
// we've actually calculated it (which is done by generating the image).
// This is a waste of resources, and is why caching solutions exist.
if (rc == NGX_ERROR || rc > NGX_OK || r->header_only || r->method == NGX_HTTP_HEAD) {
    return rc;
}
```

Redpill Linpro

Allocating memory in an nginx-module

- Nginx has its own memory management system.
- ngx_pcalloc(pool, amount) will give you a pointer to some memory you can do what you want with.
- 'pool' attaches the memory to an owner. Some memory will be allocated for the request (ngx_http_request_t->pool), and some will be allocated for the module configuration (ngx_conf_t->pool).
- Nginx deals with freeing/reusing this memory when it's owner doesn't need it anymore.

Redpill Linpro

How does buffers and chains work?

- ngx_chain_t is a linked list of ngx_buf_t's.
- You provide a pointer to the first ngx_chain_t in the chain to ngx_http_output_filter()
- It will walk through the chain, and pass all the buffers to the output filter
- You need to get comfortable with this, or attaching bits and pieces together will be tricky.

Redpill Linpro

Forgot what a linked list is?

```
struct element {
    int value;
    struct element * next;
};
```

A linked list is typically a struct, referencing an element of its own type. The last element is often set to NULL, but they can also be circular, or double (having an additional reference for the previous item in the list), etc.

Creating our callback function (1)

Cairo provides you with an interface for accessing png-images however you'd like. What we want to do, is to attach them to our chain. We do that by implementing our own callback function. This function will be called an undefined number of times.

```
static cairo_status_t
copy_png_to_chain(void * closure, const unsigned char * data, unsigned int length)
{
    // closure is a 'struct closure'
    struct closure * c = closure;

    // Just a helper pointer, to help us traverse the linked list.
    ngx_chain_t * ch = c->chain;

    // We track the size of the png-file in our closure struct.
    c->length += length;
```

Redpill Linpro

Creating our callback function (2)

```
// The allocated memory belongs to the request-pool.
ngx_buf_t * b = ngx_pcalloc(c->r->pool, sizeof(ngx_buf_t));
unsigned char * d = ngx_pcalloc(c->r->pool, length);

// We make sure to fail if we're unable to allocate memory.
if (b == NULL || d == NULL) {
    return CAIRO_STATUS_NO_MEMORY;
}

// Copy data to our new buffer, and set the pointers.
ngx_memcpy(d, data, length);

b->pos = d;
b->last = d + length;
b->memory = 1;
b->last_buf = 1;
```

Redpill Linpro

Creating our callback function (3)

If the first element isn't put into place, we can quit early.

```
// Handle the first element in our linked list.
if ( c->chain->buf == NULL )
{
    c->chain->buf = b;
    return CAIRO_STATUS_SUCCESS;
}
```

Redpill Linpro

Creating our callback function (4)

```
// Skip to the end of the linked list.
while ( ch->next )
{
    ch = ch->next;
}
```

Redpill Linpro

Creating our callback function (5)

```
    // Allocate a new link in our chain.
    ch->next = ngx_pcalloc(c->r->pool, sizeof(ngx_chain_t));
    if ( ch->next == NULL )
    {
        return CAIRO_STATUS_NO_MEMORY;
    }

    // Attach our buffer at the end.
    ch->next->buf = b;
    ch->next->next = NULL;

    return CAIRO_STATUS_SUCCESS;
}
```

Presto! That was easy :)

Redpill Linpro

We're ready to test this out!

Fingers crossed.

Redpill Linpro

Yeah, but what if I want to be able to configure my module?

- Step 3: Give your module its own configuration directives

Redpill Linpro

Create a datatype for config-storage

```
typedef struct {
    ngx_uint_t   radius;
} ngx_http_fun_loc_conf_t;
```

Redpill Linpro

Extend the ngx_http_fun_commands[]-array

```
{   // New parameter: "fun_radius":
    ngx_string("fun_radius"),
    // Can be specified on the main level of the config,
    // can be specified in the server level of the config,
    // can be specified in the location level of the config,
    // the directive takes 1 argument (NGX_CONF_TAKE1)
    NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
    // A builtin function for setting numeric variables
    ngx_conf_set_num_slot,
    // We tell nginx how we're storing the config.
    NGX_HTTP_LOC_CONF_OFFSET,
    offsetof(ngx_http_fun_loc_conf_t, radius),
    NULL
},
```

Redpill Linpro

Create a function for dealing with the config creation

```
static void *
ngx_http_fun_create_loc_conf(ngx_conf_t * cf)
{
    ngx_http_fun_loc_conf_t * conf;

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

Redpill Linpro

Create a function for merging config

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

    ngx_conf_merge_uint_value(conf->radius, prev->radius, 100);

    if (conf->radius < 1) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "radius must be equal or more than 1");
        return NGX_CONF_ERROR;
    }
    if (conf->radius > 1000 ) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "radius must be equal or less than 1000");
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;
}
```

Redpill Linpro

Add pointers to our new function in the ngx_htt_fun_module_ctx

```
ngx_http_fun_create_loc_conf,   // create location configuration
ngx_http_fun_merge_loc_conf     // merge location configuration
```

Redpill Linpro

Give our handler access to the configuration data

```
ngx_http_fun_loc_conf_t * cglcf;
cglcf = ngx_http_get_module_loc_conf(r, ngx_http_fun_module);
```

Redpill Linpro

Using the config-data in our module

We override the dimensions of our image, and we override the center-position.

```
surface = cairo_image_surface_create (CAIRO_FORMAT_ARGB32, cglcf->radius*2 + 64, cglcf->radius*2 + 64);
cr = cairo_create (surface);

double xc = cglcf->radius + 32;
double radius = cglcf->radius;
```

It can't possibly be that easy! Show me!

Update the configuration to something like this:

```
Location: /funbig {
    fun;
    fun_radius 500;
}

Location: /funsmall {
    fun;
    fun_radius 50;
}
```

The different URLs result in different sized images.

Redpill Linpro

I want to deal with user-input

- Step 4: Fetching arguments from the URL

Redpill Linpro

This is as easy as parsing strings in C. *cough*

The request has an uri-element, with members .data and .len.

Adding something like this to your handle will give you the 3 last characters of the URI as an integer;

```
char * uri;
int angle = 0;

if ( r->uri.len > 3 )
{
    uri = (char * )r->uri.data + r->uri.len - 3;
    angle = strtol(uri, NULL, 10);
}
```

Redpill Linpro

Which you then can use ...

```
double angle1 = 0.0;                    // angles are specified
double angle2 = angle * (M_PI/180.0);   // in radians
```

Redpill Linpro

Let's have a look at this too!

Last demo.

Redpill Linpro

References

- http://www.evanmiller.org/nginx-modules-guide.html
- The nginx source code.
- http://blog.zhuzhaoyuan.com/
- http://www.nginxguts.com/

Redpill Linpro

That's it!