agentzh's Nginx Tutorials (version 2016.07.21)

Table of Contents

Foreword

I've been doing a lot of work in the Nginx world over the last few years and I've also been thinking about writing a series of tutorial-like articles to explain to more people what I've done and what I've learned in this area. Now I have finally decided to post serial articles to the Sina Blog http://blog.sina.com.cn/openresty in Chinese. Every article will roughly cover a single topic and will be in a rather casual style. But at some point in the future I may restructure the articles and their style in order to turn them into a "real" book.

The articles are divided into series. For example, the first series is "Nginx Variables". Each series can be thought of as mapping to a chapter in the Nginx book that I may publish in the future.

The articles are intended for Nginx users of all experience levels, including users with extensive Apache and Lighttpd experience who may have never used Nginx before.

The examples in the articles are at least compatible with Nginx `0.8.54`. Do not try the examples with older versions of Nginx. The latest stable version of Nginx as of this writing is `1.7.9`.

All of the Nginx modules referenced in the articles are production-ready. I will not be covering any Nginx core modules that are either experimental or buggy. Additionally, I will be making extensive use of 3rd-party Nginx modules in the examples. If it's inconvenient for you to download and install the individual modules one at a time then I highly recommend that you download and install the `ngx_openresty` software bundle that I maintain.

http://openresty.org/

All of the modules referenced in the articles, including the core Nginx modules that are new (but stable), are included in the OpenResty bundle.

A principle that I will be trying to adhere to is to use small concise examples to explain and validate the concepts and behaviors being described. My hope is that it will help the reader to develop the good habit of not accepting others' viewpoints or statements at face value without testing them first. This approach may have something to do with my QA background. In fact, I keep tweaking and correcting the articles based on the results of running the examples while writing.

The examples in the articles fall into one of two categories, good and problematic. The purpose of the problematic examples is to highlight potential pitfalls and other areas where Nginx or its modules behave in ways that readers may not expect. Problematic examples are easy to identify because each line of text in the example will be prefixed with a question mark, i.e., "`?`". Here is an example:

```
? server {
?     listen 8080;
?
?     location /bad {
?         echo $foo;
?     }
? }
```

Do not reproduce these articles without explicit permissions from us. Copyright reserved.

I encourage readers to send feedback (`agentzh@gmail.com`), especially constructive criticism.

The source for all the articles is on GitHub:

http://github.com/agentzh/nginx-tutorials/

The source files are under the `en/` directory. I am using a little markup language that is a mixture of `Wiki` and `POD` to write these articles. They are the `.tut` files. You are welcome to create forks and/or provide patches.

The e-books files that are suitable for cellphones, Kindle, iPad/iPhone, Sony Readers, and other devices can be downloaded from here:

http://openresty.org/#eBooks

Special thanks go to Kai Wu (kai10k) who kindly translates these articles to English.

agentzh at home in the Fuzhou city

October 30, 2011

Writing Plan for the Tutorials

Here lists the tutorial series that have already been published or to be published.

- Getting Started with Nginx
- How Nginx Matches URIs
- [Nginx Variables](#)
- [Nginx Directive Execution Order](#)
- Nginx's if is Evil
- Nginx Subrequests
- Nginx Static File Services
- Nginx Log Services
- Application Gateways based on Nginx
- Reverse-Proxies based on Nginx
- Nginx and Memcached
- Nginx and Redis
- Nginx and MySQL
- Nginx and PostgreSQL
- Application caching Based on Nginx
- Security and Access Control in Nginx
- Web Services Based on Nginx
- AJAX Applications Driven by Nginx
- Performance Testing for Nginx and its Applications
- Strength of the Nginx Community

The series names can roughly correspond to the chapter names in my final Nginx book, but they are unlikely to stay exactly the same. The actual series names may change and the relative order of the series may change as well.

The list above will be constantly updated to always reflect the latest plan.

Nginx Variables (01)

Variables as Value Containers

Nginx's configuration files use a micro programming language. Many real-world Nginx configuration files are essentially small programs. This language's design is heavily influenced by Perl and Bourne Shell as far as I can see, despite the fact that it might not be Turing-Complete and it is declarative in many places. This is a distinguishing feature of Nginx, as compared to other web servers like Apache or Lighttpd. Being a programming language, "variables" are thus a natural part of it (exceptions do exist, of course, as in pure functional languages like Haskell).

Variables are just containers holding various values in imperative languages like Perl, Bourne Shell, and C/C++. And "values" can be numbers like `3.14`, strings like `hello world`, or even complicated things like references to arrays or hash tables in those languages. For the Nginx configuration language, however, variables can hold only one type of values, that is, strings (there is an interesting exception: the 3rd-party module [ngx_array_var](#) extends Nginx variables to hold arrays, but it is implemented by encoding a C pointer as a binary string value behind the scene).

Variables are value containers
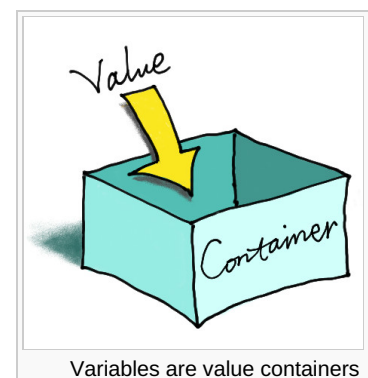
Variable Syntax and Interpolation

Let's say our `nginx.conf` configuration file has the following line:

```
set $a "hello world";
```

We assign a value to the variable `$a` via the [set](#) configuration directive coming from the standard [ngx_rewrite](#) module. In particular, we assign the string value `hello world` to `$a`.

We can see that the Nginx variable name takes a dollar sign `$`) in front of it. This is required by the language syntax: whenever we want to reference an Nginx variable in the configuration file, we must add a `$` prefix. This looks very familiar to those Perl and PHP programmers.

Such variable prefix modifiers may discomfort some Java and C# programmers, this notation does have an obvious advantage though, that is, variables can be embedded directly into a string literal:

```
set $a hello;
set $b "$a, $a";
```

Here we use the value of the existing Nginx variable `$a` to construct the value for the variable `$b`. So after these two directives complete execution, the value of `$a` is `hello`, and `$b` is `hello, hello`. This technique is called "variable interpolation" in the Perl world, which makes ad-hoc string concatenation operators no longer that necessary. Let's use the same term for the Nginx world from now on.

Let's see another complete example:

```
server {
    listen 8080;

    location /test {
        set $foo hello;
        echo "foo: $foo";
    }
}
```

This example omits the `http` directive and `events` configuration blocks in the outer-most scope for brevity. To request this `/test` interface via `curl`, an HTTP client utility, on the command line, we get

```
$ curl 'http://localhost:8080/test'
foo: hello
```

Here we use the [echo](#) directive of the 3rd party module [ngx_echo](#) to print out the value of the `$foo` variable as the HTTP response.

Apparently the arguments of the [echo](#) directive does support "variable interpolation", but we can not take it for granted for other directives. Because not all the configuration directives support "variable interpolation" and it is in fact up to the implementation of the directive in that module. Always look up the documentation to be sure.

Escaping "$"

We've already learned that the `$` character is special and it serves as the variable name prefix, but now consider that we want to output a literal `$` character via the [echo](#) directive. The following naive example does not work at all:

```
? :nginx
? location /t {
?     echo "$";
? }
```

We will get the following error message while loading this configuration:

```
[emerg] invalid variable name in ...
```

Obviously Nginx tries to parse `$"` as a variable name. Is there a way to escape `$` in the string literal? The answer is "no" (it is still the case in the latest Nginx stable release `1.2.7`) and I

have been hoping that we could write something like `$$` to obtain a literal `$`.

Luckily, workarounds do exist and here is one proposed by Maxim Dounin: first we assign to a variable a literal string containing a dollar sign character via a configuration directive that does *not* support "variable interpolation" (remember that not all the directives support "variable interpolation"?), and then reference this variable later whenever we need a dollar sign. Here is such an example to demonstrate the idea:

```
geo $dollar {
    default "$";
}

server {
    listen 8080;

    location /test {
        echo "This is a dollar sign: $dollar";
    }
}
```

Let's test it out:

```
$ curl 'http://localhost:8080/test'
This is a dollar sign: $
```

Here we make use of the [geo](#) directive of the standard module [ngx_geo](#) to initialize the `$dollar` variable with the string `"$"`, thereafter variable `$dollar` can be used in places that require a dollar sign. This works because the [geo](#) directive does not support "variable interpolation" at all. However, the [ngx_geo](#) module is originally designed to set a Nginx variable to different values according to the remote client address, and in this example, we just abuse it to initialize the `$dollar` variable with the string `"$"` unconditionally.

Disambiguating Variable Names

There is a special case for "variable interpolation", that is, when the variable name is followed directly by characters allowed in variable names (like letters, digits, and underscores). In such cases, we can use a special notation to disambiguate the variable name from the subsequent literal characters, for instance,

```
server {
    listen 8080;

    location /test {
        set $first "hello ";
        echo "${first}world";
    }
}
```

Here the variable `$first` is concatenated with the literal string `world`. If it were written directly as `"$firstworld"`, Nginx's "variable interpolation" engine (also known as the "script engine") would try to access the variable `$firstworld` instead of `$first`. To resolve the ambiguity here, curly braces must be used around the variable name (excluding the `$` prefix), as in `${first}`. Let's test this sample:

```
$ curl 'http://localhost:8080/test
hello world
```

Variable Declaration and Creation

In languages like C/C++, variables must be declared (or created) before they can be used so that the compiler can allocate storage and perform type checking at compile-time. Similarly, Nginx creates all the Nginx variables while loading the configuration file (or in other words, at "configuration time"), therefore Nginx variables are also required to be declared somehow.

Fortunately the [set](#) directive and the [geo](#) directive mentioned above do have the side effect of declaring or creating Nginx variables that they will assign values to later at "request time". If we do not declare a variable this way and use it directly in, say, the [echo](#) directive, we will get an error. For example,

```
? server {
?    listen 8080;
?
?    location /bad {
?        echo $foo;
?    }
? }
```

Here we do not declare the `$foo` variable and access its value directly in <u>echo</u>. Nginx will just refuse loading this configuration:

```
[emerg] unknown "foo" variable
```

Yes, we cannot even start the server!

Nginx variable creation and assignment happen at completely different phases along the time-line. Variable creation only occurs when Nginx loads its configuration. On the other hand, variable assignment occurs when requests are actually being served. This also means that we can never create new Nginx variables at "request time".

Variable Scope

Once an Nginx variable is created, it is visible to the entire configuration, even across different virtual server configuration blocks, regardless of the places it is declared at. Here is an example:

```
server {
    listen 8080;

    location /foo {
        echo "foo = [$foo]";
    }

    location /bar {
        set $foo 32;
        echo "foo = [$foo]";
    }
}
```

Here the variable `$foo` is created by the <u>set</u> directive within `location /bar`, and this variable is visible to the entire configuration, therefore we can reference it in `location /foo` without worries. Below is the result of testing these two interfaces via the `curl` tool.

```
$ curl 'http://localhost:8080/foo'
foo = []

$ curl 'http://localhost:8080/bar'
foo = [32]

$ curl 'http://localhost:8080/foo'
foo = []
```

We can see that the assignment operation is only performed in requests that access `location /bar`, since the corresponding <u>set</u> directive is only used in that location. When requesting the `/foo` interface, we always get an empty value for the `$foo` variable because that is what we get when accessing an uninitialized variable.

Another important characteristic that we can observe from this example is that even though the scope of Nginx variables is the entire configuration, each request does have its own version of all those variables' containers. Requests do not interfere with each other even if they are referencing a variable with the same name. This is very much like local variables in C/C++ function bodies. Each invocation of the C/C++ function does use its own version of those local variables (on the stack).

For instance, in this sample, we request `/bar` and the variable `$foo` gets the value `32`, which does not affect the value of `$foo` in subsequent requests to `/foo` (it is still uninitialized!), because they correspond to different value containers.

One common mistake for Nginx newcomers is to regard Nginx variables as something

shared among all the requests. Even though the scope of Nginx variable *names* go across configuration blocks at "configuration time", its *value container*'s scope never goes beyond request boundaries at "request time". Essentially here we do have two different kinds of scope here.

Nginx Variables (02)

Variable Lifetime & Internal Redirection

We already know that Nginx variables are bound to each request handled by Nginx, for this reason they have exactly the same lifetime as the corresponding request.

There is another common misunderstanding here though: some newcomers tend to assume that the lifetime of Nginx variables is bound to the `location` configuration block. Let's consider the following counterexample:

```
server {
    listen 8080;

    location /foo {
        set $a hello;
        echo_exec /bar;
    }

    location /bar {
        echo "a = [$a]";
    }
}
```

Here in `location /foo` we use the [echo_exec](#) directive (provided by the 3rd-party module [ngx_echo](#)) to initiate an "internal redirection" to `location /bar`. The "internal redirection" is an operation that makes Nginx jump from one `location` to another while processing a request. This "jumping" happens completely within the server itself. This is different from those "external redirections" based on the HTTP `301` and `302` responses because the latter is collaborated externally, by the HTTP clients. Also, in case of "external redirections", the end user could usually observe the change of the URL in her web browser's address bar while this is not the case for internal ones. "Internal redirections" are very similar to the `exec` command in Bourne Shell; it is a "one way trip" and never returns. Another similar example is the `goto` statement in the C language.

Being an "internal redirection", the request after the redirection remains the original one. It is just the current `location` that is changed, so we are still using the original copy of the Nginx variable containers. Back to our example, the whole process looks like this: Nginx first assigns to the `$a` variable the string value `hello` via the [set](#) directive in `location /foo`, and then it issues an internal redirection via the [echo_exec](#) directive, thus leaving `location /foo` and entering `location /bar`, and finally it outputs the value of `$a`. Because the value container of `$a` remains untouched, we can expect the response output to be `hello`. The test result confirms this:

```
$ curl localhost:8080/foo
a = [hello]
```

But when accessing `/bar` directly from the client side, we will get an empty value for the `$a` variable, since this variable relies on `location /foo` to get initialized.

It can be observed that during a request's lifetime, the copy of Nginx variable containers does not change at all even when Nginx goes across different `location` configuration blocks. Here we also encounter the concept of "internal redirections" for the first time and it's worth mentioning that the [rewrite](#) directive of the [ngx_rewrite](#) module can also be used to initiate "internal redirections". For instance, we can rewrite the example above with the [rewrite](#) directive as follows:

```
server {
    listen 8080;

    location /foo {
        set $a hello;
        rewrite ^ /bar;
    }
```

```
    location /bar {
        echo "a = [$a]";
    }
}
```

It's functionally equivalent to echo_exec. We will discuss the rewrite directive in more depth in later chapters, like initiating "external redirections" like `301` and `302`.

To conclude, the lifetime of Nginx variable containers is indeed bound to the request being processed, and is irrelevant to `location`.

Nginx Built-in Variables

The Nginx variables we have seen so far are all (implicitly) created by directives like set. We usually call such variables "user-defined varaibles", or simply "user variables". There is also another kind of Nginx variables that are *pre-defined* by either the Nginx core or Nginx modules. Let's call this kind of variables "built-in variables".

$uri & $request_uri

One common use of Nginx built-in variables is to retrieve various types of information about the current request or response. For instance, the built-in variable $uri provided by ngx_http_core is used to fetch the (decoded) URI of the current request, excluding any query string arguments. Another example is the $request_uri variable provided by the same module, which is used to fetch the raw, non-decoded form of the URI, including any query string. Let's look at the following example.

```
location /test {
    echo "uri = $uri";
    echo "request_uri = $request_uri";
}
```

We omit the `server` configuration block here for brevity. Just as all those samples above, we still listen to the `8080` local port. In this example, we output both the $uri and $request_uri into the response body. Below is the result of testing this `/test` interface with different requests:

```
$ curl 'http://localhost:8080/test'
uri = /test
request_uri = /test

$ curl 'http://localhost:8080/test?a=3&b=4'
uri = /test
request_uri = /test?a=3&b=4

$ curl 'http://localhost:8080/test/hello%20world?a=3&b=4'
uri = /test/hello world
request_uri = /test/hello%20world?a=3&b=4
```

Variables with Infinite Names

There is another very common built-in variable that does not have a fixed variable name. Instead, It has *infinite* variations. That is, all those variables whose names have the prefix `arg_`, like `$arg_foo` and `$arg_bar`. Let's just call it the $arg_XXX "variable group". For example, the `$arg_name` variable is evaluated to the value of the `name` URI argument for the current request. Also, the URI argument's value obtained here is not decoded yet, potentially containing the `%XX` sequences. Let's check out a complete example:

```
location /test {
    echo "name: $arg_name";
    echo "class: $arg_class";
}
```

Then we test this interface with various different URI argument combinations:

```
$ curl 'http://localhost:8080/test'
```

```
name:
class:

$ curl 'http://localhost:8080/test?name=Tom&class=3'
name: Tom
class: 3

$ curl 'http://localhost:8080/test?name=hello%20world&class=9'
name: hello%20world
class: 9
```

In fact, `$arg_name` does not only match the `name` argument name, but also `NAME` or even `Name`. That is, the letter case does not matter here:

```
$ curl 'http://localhost:8080/test?NAME=Marry'
name: Marry
class:

$ curl 'http://localhost:8080/test?Name=Jimmy'
name: Jimmy
class:
```

Behind the scene, Nginx just converts the URI argument names into the pure lower-case form before matching against the name specified by $arg_XXX.

If you want to decode the special sequences like `%20` in the URI argument values, then you could use the set_unescape_uri directive provided by the 3rd-party module ngx_set_misc.

```
location /test {
    set_unescape_uri $name $arg_name;
    set_unescape_uri $class $arg_class;

    echo "name: $name";
    echo "class: $class";
}
```

Let's check out the actual effect:

```
$ curl 'http://localhost:8080/test?name=hello%20world&class=9'
name: hello world
class: 9
```

The space has indeed been decoded!

Another thing that we can observe from this example is that the set_unescape_uri directive can also implicitly create Nginx user-defined variables, just like the set directive. We will discuss the ngx_set_misc module in more detail in future chapters.

This type of variables like $arg_XXX possesses infinite number of possible names, so they do not correspond to any value containers. Furthermore, such variables are handled in a very specific way within the Nginx core. It is thus not possible for 3rd-party modules to introduce such magical built-in variables of their own.

The Nginx core offers a lot of such built-in variables in addition to $arg_XXX, like the $cookie_XXX variable group for fetching HTTP cookie values, the $http_XXX variable group for fetching request headers, as well as the $sent_http_XXX variable group for retrieving response headers. We will not go into the details for each of them here. Interested readers can refer to the official documentation for the ngx_http_core module.

Read-only Built-in Variables

All the user-defined variables are writable. Actually the way that we declare or create such variables so far is to use a configure directive, like set, that performs value assignment at request time. But it is *not* necessarily the case for built-in variables.

Most of the built-in variables are effectively *read-only*, like the $uri and $request_uri variables that we just introduced earlier. Assignments to such read-only variables must always be avoided. Otherwise it will lead to unexpected consequences, for example,

```
? location /bad {
?     set $uri /blah;
?     echo $uri;
? }
```

This problematic configuration just triggers a confusing error message when Nginx is started:

```
[emerg] the duplicate "uri" variable in ...
```

Attempts of writing to some other read-only built-in variables like $arg_XXX will just lead to server crashes in some particular Nginx versions.

Nginx Variables (03)

Writable Built-in Variable $args

Some built-in variables are writable as well. For instance, when reading the built-in variable $args, we get the URL query string of the current request, but when writing to it, we are effectively modifying the query string. Here is such an example:

```
location /test {
    set $orig_args $args;
    set $args "a=3&b=4";

    echo "original args: $orig_args";
    echo "args: $args";
}
```

Here we first save the original URL query string into our own variable `$orig_args`, then modify the current query string by overriding the $args variable, and finally output the variables `$orig_args` and $args, respectively, with the echo directive. Let's test it like this:

```
$ curl 'http://localhost:8080/test'
original args:
args: a=3&b=4

$ curl 'http://localhost:8080/test?a=0&b=1&c=2'
original args: a=0&b=1&c=2
args: a=3&b=4
```

In the first test, we did not provide any URL query string, hence the empty output for the `$orig_args` variable. And in both tests, the current query string was forcibly overridden to the new value `a=3&b=4`, regardless of the presence of a query string in the original request.

It should be noted that the $args variable here no longer owns a value container as user variables, just like $arg_XXX. When reading $args, Nginx will execute a special piece of code, fetching data from a particular place where the Nginx core stores the URL query string for the current request. On the other hand, when we overwrite $args, Nginx will execute another special piece of code, storing new value into the same place in the core. Other parts of Nginx also read the same place whenever the query string is needed, so our modification to $args will immediately affect all the other parts' functionality later on. Let's see an example for this:

```
location /test {
    set $orig_a $arg_a;
    set $args "a=5";
    echo "original a: $orig_a";
    echo "a: $arg_a";
}
```

Here we first save the value of the built-in varaible `$arg_a`, the value of the original request's URL argument `a`, into our user variable `$orig_a`, then change the URL query string to `a=5` by assigning the new value to the built-in variable $args, and finally output the variables `$orig_a` and `$arg_a`, respectively. Because modifications to $args effectively change the URL query string of the current request for the whole server, the value of the built-in variable $arg_XXX should also change accordingly. The test result verifies this:

```
$ curl 'http://localhost:8080/test?a=3'
original a: 3
a: 5
```

We can see that the initial value of `$arg_a` is `3` since the URL query string of the original request is `a=3`. But the final value of `$arg_a` automatically becomes `5` after we modify `$args` with the value `a=5`.

Below is another example to demonstrate that assignments to `$args` also affect the HTTP proxy module `ngx_proxy`.

```
server {
    listen 8080;

    location /test {
        set $args "foo=1&bar=2";
        proxy_pass http://127.0.0.1:8081/args;
    }
}

server {
    listen 8081;

    location /args {
        echo "args: $args";
    }
}
```

Two virtual servers are defined here in the `http` configuration block (omitted for brevity).

The first virtual server is listening at the local port 8080. Its `/test` location first updates the current URL query string to the value `foo=1&bar=2` by writing to `$args`, then sets up an HTTP reverse proxy via the `proxy_pass` directive of the `ngx_proxy` module, targeting the HTTP service `/args` on the local port 8081. By default the `ngx_proxy` module automatically forwards the current URL query string to the remote HTTP service.

The "remote HTTP service" on the local port 8081 is provided by the second virtual server defined by ourselves, where we output the current URL query string via the `echo` directive in `location /args`. By doing this, we can investigate the actual URL query string forwarded by the `ngx_proxy` module from the first virtual server.

Let's access the `/test` interface exposed by the first virtual server.

```
$ curl 'http://localhost:8080/test?blah=7'
args: foo=1&bar=2
```

We can see that the URL query string is first rewritten to `foo=1&bar=2` even though the original request takes the value `blah=7`, then it is forwarded to the `/args` interface of the second virtual server via the `proxy_pass` directive, and finally its value is output to the client.

To summarize, the assignment to `$args` also successfully influences the behavior of the `ngx_proxy` module.

Variable "Get Handlers" and "Set Handlers"

We have already learned in previous sections that when reading the built-in variable `$args`, Nginx executes a special piece of code to obtain a value on-the-fly and when writing to this variable, Nginx executes another special piece of code to propagate the change. In Nginx's terminology, the special code executed for reading the variable is called "get handler" and the code for writing to the variable is called "set handler". Different Nginx modules usually prepare different "get handlers" and "set handlers" for their own variables, which effectively put magic into these variables' behavior.

Such techniques are not uncommon in the computing world. For example, in object-oriented programming (OOP), the class designer usually does not expose the member variable of the class directly to the user programmer, but instead provides two methods for reading from and writing to the member variable, respectively. Such class methods are often called "accessors". Below is an example in the C++ programming language:

```
#include <string>
using namespace std;

class Person {
public:
    const string get_name() {
        return m_name;
    }

    void set_name(const string name) {
        m_name = name;
    }

private:
    string m_name;
};
```

In this C++ class `Person`, we provide two public methods, `get_name` and `set_name`, to serve as the "accessors" for the private member variable `m_name`.

The benefits of such design are obvious. The class designer can execute arbitrary code in the "accessors", to implement any extra business logic or useful side effects, like automatically updating other member variables depending on the current member, or updating the corresponding field in a database associated with the current object. For the latter case, it is possible that the member variable does not exist at all, or that the member variable just serves as a data cache to mitigate the pressure on the back-end database.

Corresponding to the concept of "accessors" in OOP, Nginx variables also support binding custom "get handlers" and "set handlers". Additionally, not all Nginx variables own a container to hold values. Some variables without a container just behave like a magical container by means of its fancy "get handler" and "set handler". In fact, when a variable is being created at "configure time", the creating Nginx module must make a decision on whether to allocate a value container for it and whether to attach a custom "get handler" and/or a "set handler" to it.

Those variables owning a value container are called "indexed variables" in Nginx's terminology. Otherwise, they are said to be not indexed.

We already know that the "variable groups" like [$arg_XXX](#) discussed in earlier sections do not have a value container and thus are not indexed. When reading [$arg_XXX](#), it is its "get handler" at work, that is, its "get handler" scans the current URL query string on-the-fly, extracting the value of the specified URL argument. Many beginners misunderstand the way [$arg_XXX](#) is implemented; they assume that Nginx will parse all the URL arguments in advance and prepare the values for all those non-empty [$arg_XXX](#) variables before they are actually read. This is not true, however. Nginx never tries to parse all the URL arguments beforehand, but rather scans the whole URL query string for a particular argument in a "get handler" every time that argument is requested by reading the corresponding [$arg_XXX](#) variable. Similarly, when reading the built-in variable [$cookie_XXX](#), its "get handler" just scans the `Cookie` request headers for the cookie name specified.

Nginx Variables (04)

Value Containers for Caching & ngx_map

Some Nginx variables choose to use their value containers as a data cache when the "get handler" is configured. In this setting, the "get handler" is run only once, i.e., at the first time the variable is read, which reduces overhead when the variable is read multiple times during its lifetime. Let's see an example for this.

```
map $args $foo {
    default    0;
    debug      1;
}
server {
    listen 8080;

    location /test {
        set $orig_foo $foo;
        set $args debug;
```

```
        echo "original foo: $orig_foo";
        echo "foo: $foo";
    }
}
```

Here we use the [map](#) directive from the standard module [ngx_map](#) for the first time, which deserves some introduction. The word `map` here means mapping or correspondence. For example, functions in Maths are a kind of "mapping". And Nginx's [map](#) directive is used to define a "mapping" relationship between two Nginx variables, or in other words, "function relationship". Back to this example, we use the [map](#) directive to define the "mapping" relationship between user variable `$foo` and built-in variable `$args`. When using the Math function notation, $y = f(x)$, our `$args` variable is effectively the "independent variable", x, while `$foo` is the "dependent variable", y. That is, the value of `$foo` depends on the value of [$args](#), or rather, we *map* the value of [$args](#) onto the `$foo` variable (in some way).

Now let's look at the exact mapping rule defined by the [map](#) directive in this example.

```
map $args $foo {
    default    0;
    debug      1;
}
```

The first line within the curly braces is a special rule condition, that is, this condition holds if and only if other conditions all fail. When this "default" condition holds, the "dependent variable" `$foo` is assigned by the value `0`. The second line within the curly braces means that the "dependent variable" `$foo` is assigned by the value `1` if the "independent variable" `$args` matches the string value `debug`. Combining these two lines, we obtain the following complete mapping rule: if the value of [$args](#) is `debug`, variable `$foo` gets the value `1`; otherwise `$foo` gets the value `0`. So essentially, this is a conditional assignment to the variable `$foo`.

Now that we understand what the [map](#) directive does, let's look at the definition of `location /test`. We first save the value of `$foo` into another user variable `$orig_foo`, then overwrite the value of [$args](#) to `debug`, and finally output the values of `$orig_foo` and `$foo`, respectively.

Intuitively, after we overwrite the value of [$args](#) to `debug`, the value of `$foo` should automatically get adjusted to `1` according to the mapping rule defined earlier, regardless of the original value of `$foo`. But the test result suggests the other way around.

```
$ curl 'http://localhost:8080/test'
original foo: 0
foo: 0
```

The first output line indicates that the value of `$orig_foo` is `0`, which is exactly what we expected: the original request does not take a URL query string, so the initial value of [$args](#) is empty, leading to the `0` initial value of `$foo`, according to the "default" condition in our mapping rule.

But surprisingly, the second output line indicates that the final value of `$foo` is still `0`, even after we overwrite [$args](#) to the value `debug`. This apparently violates our mapping rule because when [$args](#) takes the value `debug`, the value of `$foo` should really be `1`. So what is happening here?

Actually the reason is pretty simple: when the first time variable `$foo` is read, its value computed by [ngx_map](#)'s "get handler" is cached in its value container. We already learned earlier that Nginx modules may choose to use the value container of the variable created by themselves as a data cache for its "get handler". Obviously, the [ngx_map](#) module considers the mapping computation between variables expensive enough and caches the result automatically, so that the next time the same variable is read within the lifetime of the current request, Nginx can just return the cached result without invoking the "get handler" again.

To verify this further, we can try specifying the URL query string as `debug` in the original request.

```
$ curl 'http://localhost:8080/test?debug'
original foo: 1
foo: 1
```

It can be seen that the value of `$orig_foo` becomes `1`, complying with our mapping rule. And subsequent readings of `$foo` always yield the same cached result, `1`, regardless of the new value of [$args](#) later on.

The map directive is actually a unique example, because it not only registers a "get handler" for the user variable, but also allows the user to define the computing rule in the "get handler" directly in the Nginx configuration file. Of course, the rule that can be defined here is limited to simple mapping relations with another variable. Meanwhile, it must be made clear that not all the variables using a "get handler" will cache the result. For instance, we have already seen earlier that the $arg_XXX variable does not use its value container at all.

Similar to the ngx_map module, the standard module ngx_geo that we encountered earlier also enables value caching for the variables created by its geo directive.

A Side Note for Use Contexts of Directives

In the previous example, we should also note that the map directive is put outside the `server` configuration block, that is, it is defined directly within the outermost `http` configuration block. Some readers may be curious about this setting, since we only use it in `location /test` after all. If we try putting the map statement within the `location` block, however, we will get the following error while starting Nginx:

```
[emerg] "map" directive is not allowed here in ...
```

So it is explicitly prohibited. In fact, it is only allowed to use the map directive in the `http` block. Every configure directive does have a pre-defined set of use contexts in the configuration file. When in doubt, always refer to the corresponding documentation for the exact use contexts of a particular directive.

Lazy Evaluation of Variable Values

Many Nginx freshmen would worry that the use of the map directive within the global scope (i.e., the `http` block) will lead to unnecessary variable value computation and assignment for all the `location`s in all the virtual servers even if only one `location` block actually uses it. Fortunately, this is *not* what is happening here. We have already learned how the map directive works. It is the "get handler" (registered by the ngx_map module) that performs the value computation and related assignment. And the "get handler" will not run at all unless the corresponding user variable is actually being read. Therefore, for those requests that never access that variable, there cannot be any (useless) computation involved.

The technique that postpones the value computation off to the point where the value is actually needed is called "lazy evaluation" in the computing world. Programming languages natively offering "lazy evaluation" is not very common though. The most famous example is the Haskell programming language, where lazy evaluation is the default semantics. In contrast with "lazy evaluation", it is much more common to see "eager evaluation". We are lucky to see examples of lazy evaluation here in the ngx_map module, but the "eager evaluation" semantics is also much more common in the Nginx world. Consider the following set statement that cannot be simpler:

```
set $b "$a,$a";
```

When running the set directive, Nginx eagerly computes and assigns the new value for the variable `$b` without postponing to the point when `$b` is actually read later on. Similarly, the set_unescape_uri directive also evaluates eagerly.

Nginx Variables (05)

Variables in Subrequests

A Detour to Subrequests

We have seen earlier that the lifetime of variable containers is bound to the request, but I own you a formal definition of "requests" there. You might have assumed that the "requests" in that context are just those HTTP requests initiated from the client side. In fact, there are two kinds of "requests" in the Nginx world. One is called "main requests", and the other is called "subrequests".

Main requests are those initiated externally by HTTP clients. All the examples that we have seen so far involve main requests only, including those doing "internal redirections" via the echo_exec or rewrite directive.

Whereas subrequests are a special kind of requests initiated from within the Nginx core. But

please do not confuse subrequests with those HTTP requests created by the [ngx_proxy](#) modules! Subrequests may look very much like an HTTP request in appearance, their implementation, however, has nothing to do with neither the HTTP protocol nor any kind of socket communication. A subrequest is an abstract invocation for decomposing the task of the main request into smaller "internal requests" that can be served independently by multiple different `location` blocks, either in series or in parallel. "Subrequests" can also be recursive: any subrequest can initiate more sub-subrequests, targeting other `location` blocks or even the current `location` itself. According to Nginx's terminology, if request A initiates a subrequest B, then A is called the "parent request" of B. It is worth mentioning that the Apache web server also has the concept of subrequests for long, so readers coming from that world should be no stranger to this.

Let's check out an example using subrequests:

```
location /main {
    echo_location /foo;
    echo_location /bar;
}

location /foo {
    echo foo;
}

location /bar {
    echo bar;
}
```

Here in `location /main`, we use the [echo_location](#) directive from the [ngx_echo](#) module to initiate two `GET`-typed subrequests targeting `/foo` and `/bar`, respectively. The subrequests initiated by [echo_location](#) are always running sequentially according to their literal order in the configuration file. Therefore, the second `/bar` request will not be fired until the first `/foo` request completes processing. The response body of these two subrequests get concatenated together according to their running order, to form the final response body of their parent request (for `/main`):

```
$ curl 'http://localhost:8080/main'
foo
bar
```

It should be noted that the communication of `location` blocks via subrequests is limited within the same `server` block (i.e., the same virtual server configuration), so when the Nginx core processes a subrequest, it just calls a few C functions behind the scene, without doing any kind of network or UNIX domain socket communication. For this reason, subrequests are extremely efficient.

Independent Variable Containers in Subrequests

Back to our earlier discussion for the lifetime of Nginx variable containers, now we can still state that the lifetime is bound to the current request, and every request does have its own copy of all the variable containers. It is just that the "request" here can be either a main request, or a subrequest. Variables with the same name between a parent request and a subrequest will generally not interfere with each other. Let's do a small experiment to confirm this:

```
location /main {
    set $var main;

    echo_location /foo;
    echo_location /bar;

    echo "main: $var";
}

location /foo {
    set $var foo;
    echo "foo: $var";
}

location /bar {
    set $var bar;
    echo "bar: $var";
```

```
    }
```

In this sample, we assign different values to the variable `$var` in three `location` blocks, `/main`, `/foo`, and `/bar`, and output the value of `$var` in all these locations. In particular, we intentionally output the value of `$var` in `location /main` *after* calling the two subrequests, so if value changes of `$var` in the subrequests can affect their parent request, we should see a new value output in location `/main`. The result of requesting `/main` is as follows:

```
$ curl 'http://localhost:8080/main'
foo: foo
bar: bar
main: main
```

Apparently, the assignments to variable `$var` in those two subrequests do not affect the main request `/main` at all. This successfully verifies that both the main request and its subrequests do own different copies of variable containers.

Shared Variable Containers among Requests

Unfortunately, subrequests initiated by certain Nginx modules do share variable containers with their parent requests, like those initiated by the 3rd-party module [ngx_auth_request](). Below is such an example:

```
location /main {
    set $var main;
    auth_request /sub;
    echo "main: $var";
}

location /sub {
    set $var sub;
    echo "sub: $var";
}
```

Here in `location /main`, we first assign the initial value `main` to variable `$var`, then fire a subrequest to `/sub` via the `auth_request` directive from the [ngx_auth_request]() module, and finally output the value of `$var`. Note that in `location /sub` we intentionally overwrite the value of `$var` to `sub`. When accessing `/main`, we get

```
$ curl 'http://localhost:8080/main'
main: sub
```

Obviously, the value change of `$var` in the subrequest to `/sub` does affect the main request to `/main`. Thus the variable container of `$var` is indeed shared between the main request and the subrequest created by the [ngx_auth_request]() module.

For the previous example, some readers might ask: "why doesn't the response body of the subrequest appear in the final output?" The answer is simple: it is just because the `auth_request` directive discards the response body of the subrequest it manages, and only checks the response status code of the subrequest. When the status code looks good, like `200`, `auth_request` will just allow Nginx continue processing the main request; otherwise it will immediately abort the main request by returning a `403` error page, for example. In our example, the subrequest to `/sub` just return a `200` response implicitly created by the [echo]() directive in `location /sub`.

Even though sharing variable containers among the main request and all its subrequests could make bidirectional data exchange easier, it could also lead to unexpected subtle issues that are hard to debug in real-world configurations. Because users often forget that a variable with the same name is actually used in some deeply embedded subrequest and just use it for something else in the main request, this variable could get unexpectedly modified during processing. Such bad side effects make many 3rd-party modules like [ngx_echo](), [ngx_lua]() and [ngx_srcache]() choose to disable the variable sharing behavior for subrequests by default.

Nginx Variables (06)

Built-in Variables in Subrequests

There are some subtleties involved in using Nginx built-in variables in the context of a subrequest. We will discuss the details in this section.

Built-in Variables Sensitive to the Subrequest Context

We already know that most built-in variables are not simple value containers. They behave differently than user variables by registering "get handlers" and/or "set handlers". Even when they do own a value container, they usually just use the container as a result cache for their "get handlers". The $args variable we discussed earlier, for example, just uses its "get handler" to return the URL query string for the current request. The current request here can also be a subrequest, so when reading $args in a subrequest, its "get handler" should naturally return the query string for the subrequest. Let's see such an example:

```
location /main {
    echo "main args: $args";
    echo_location /sub "a=1&b=2";
}

location /sub {
    echo "sub args: $args";
}
```

Here in the /main interface, we first echo out the value of $args for the current request, and then use echo_location to initiate a subrequest to /sub. It should be noted that here we give a second argument to the echo_location directive, to specify the URL query string for the subrequest being fired (the first argument is the URI for the subrequest, as we already know). Finally, we define the /sub interface and print out the value of $args in there. Querying the /main interface gives

```
$ curl 'http://localhost:8080/main?c=3'
main args: c=3
sub args: a=1&b=2
```

It is clear that when $args is read in the main request (to /main), its value is the URL query string of the main request; whereas when in the subrequest (to /foo), it is the query string of the subrequest, a=1&b=2. This behavior indeed matches our intuition.

Just like $args, when the built-in variable $uri is used in a subrequest, its "get handler" also returns the (decoded) URI of the current subrequest:

```
location /main {
    echo "main uri: $uri";
    echo_location /sub;
}

location /sub {
    echo "sub uri: $uri";
}
```

Below is the result of querying /main:

```
$ curl 'http://localhost:8080/main'
main uri: /main
sub uri: /sub
```

The output is what we would expect.

Built-in Variables for Main Requests Only

Unfortunately, not all built-in variables are sensitive to the context of subrequests. Several built-in variables always act on the main request even when they are used in a subrequest. The built-in variable $request_method is such an exception.

Whenever $request_method is read, we always get the request method name (such as GET and POST) for the main request, no matter whether the current request is a subrequest or not. Let's test it out:

```
location /main {
```

```
        echo "main method: $request_method";
        echo_location /sub;
    }

    location /sub {
        echo "sub method: $request_method";
    }
```

In this example, the `/main` and `/sub` interfaces both output the value of $request_method. Meanwhile, we initiate a `GET` subrequest to `/sub` via the echo_location directive in `/main`. Now let's do a `POST` request to `/main`:

```
    $ curl --data hello 'http://localhost:8080/main'
    main method: POST
    sub method: POST
```

Here we use the `--data` option of the `curl` utility to specify our POST request body, also this option makes `curl` use the `POST` method for the request. The test result turns out as we predicted: the variable $request_method is evaluated to the main request's method name, `POST`, despite its use in a `GET` subrequest.

Some readers might challenge our conclusion here by pointing out that we did not rule out the possibility that the value of $request_method got cached at its first reading in the main request and what we were seeing in the subrequest was actually the cached value that was evaluated earlier in the main request. This concern is unnecessary, however, because we have also learned that the variable container required by data caching (if any) is always bound to the current request, also the subrequests initiated by the ngx_echo module always disable variable container sharing with their parent requests. Back to the previous example, even if the built-in variable $request_method in the main request used the value container as the data cache (actually it does not), it cannot affect the subrequest by any means.

To further address the concern of these readers, let's slightly modify the previous example by putting the echo statement for $request_method in `/main` *after* the echo_location directive that runs the subrequest:

```
    location /main {
        echo_location /sub;
        echo "main method: $request_method";
    }

    location /sub {
        echo "sub method: $request_method";
    }
```

Let's test it again:

```
    $ curl --data hello 'http://localhost:8080/main'
    sub method: POST
    main method: POST
```

No change in the output can be observed, except that the two output lines reversed the order (since we exchange the order of those two ngx_echo module's directives).

Consequently, we cannot obtain the method name of a subrequest by reading the $request_method variable. This is a common pitfall for freshmen when dealing with method names of subrequests. To overcome this limitation, we need to turn to the built-in variable $echo_request_method provided by the ngx_echo module:

```
    location /main {
        echo "main method: $echo_request_method";
        echo_location /sub;
    }

    location /sub {
        echo "sub method: $echo_request_method";
    }
```

We are finally getting what we want:
```

```
$ curl --data hello 'http://localhost:8080/main'
main method: POST
sub method: GET
```

Now within the subrequest, we get its own method name, `GET` , as expected, and the main request method remains `POST` .

Similar to $request_method, the built-in variable $request_uri also always returns the (non-decoded) URL for the main request. This is more understandable, however, because subrequests are essentially faked requests inside Nginx, which do not really take a non-decoded raw URL.

Variable Container Sharing and Value Caching Together

In the previous section, some of the readers were worried about the case that variable container sharing in subrequests and value caching for variable's "get handlers" were working together. If it were indeed the case, then it would be a nightmare because it would be really really hard to predict what is going on by just looking at the configuration file. In previous sections, we already learned that the subrequests initiated by the ngx_auth_request module are sharing the same variable containers with their parents, so we can maliciously construct such a horrible example:

```
map $uri $tag {
    default    0;
    /main      1;
    /sub       2;
}

server {
    listen 8080;

    location /main {
        auth_request /sub;
        echo "main tag: $tag";
    }

    location /sub {
        echo "sub tag: $tag";
    }
}
```

Here we use our old friend, the map directive, to map the value of the built-in variable $uri to our user variable `$tag` . When $uri takes the value `/main` , the value `1` is assigned to `$tag` ; when $uri takes the value `/sub` , the value `2` is assigned instead to `$tag` ; under all the other conditions, `0` is assigned. Next, in `/main` , we first initiate a subrequest to `/sub` by using the `auth_request` directive, and then output the value of `$tag` . And within `/sub` , we directly output the value of `$tag` . Guess what we will get when we access `/main` ?

```
$ curl 'http://localhost:8080/main'
main tag: 2
```

Ouch! Didn't we map the value `/main` to `1` ? Why the actual output for `/main` is the value, `2` , for `/sub` ? What is going on here?

Actually it worked like this: our `$tag` variable was first read in the subrequest to `/sub` , and the "get handler" registered by map computed the value `2` for `$tag` in that context (because $uri was `/sub` in the subrequest) and the value `2` got cached in the value container of `$tag` from then on. Because the parent request shared the same container as the subrequest created by `auth_request` , when the parent request read `$tag` later (after the subrequest was finished), the cached value `2` was directly returned! Such results can indeed be very surprising at first glance.

From this example, we can conclude again that it can hardly be a good idea to enable variable container sharing in subrequests.

Nginx Variables (07)

Special Value "Invalid" and "Not Found"

We have mentioned that the values of Nginx variables can only be of one single type, that is, the string type, but variables could also have no meaningful values at all. Variables without any meaningful values still take a special value though. There are two possible special values: "invalid" and "not found".

For example, when a user variable $foo is created but not assigned yet, $foo takes the special value of "invalid". And when the current URL query string does not have the XXX argument at all, the built-in variable $arg_XXX takes the special value of "not found".

Both "invalid" and "not found" are special values, completely different from an empty string value (""). This is very similar to those distinct special values in some dynamic programing languages, like undef in Perl, nil in Lua, and null in JavaScript.

We have seen earlier that an uninitialized variable is evaluated to an empty string when used in an interpolated string, its real value, however, is not an empty string at all. It is the "get handler" registered by the set directive that automatically converts the "invalid" special value into an empty string. To verify this, let's return to the example we have discussed before:

```
location /foo {
    echo "foo = [$foo]";
}

location /bar {
    set $foo 32;
    echo "foo = [$foo]";
}
```

When accessing /foo, the user variable $foo is uninitialized when used in the interpolated string for the echo directive. The output shows that the variable is evaluated to an empty string:

```
$ curl 'http://localhost:8080/foo'
foo = []
```

From the output, the uninitialized $foo variable behaves just like taking an empty string value. But careful readers should have already noticed that, for the request above, there is a warning in the Nginx error log file (which is logs/error.log by default):

```
[warn] 5765#0: *1 using uninitialized "foo" variable, ...
```

Who on earth generates this warning? The answer is the "get handler" of $foo, registered by the set directive. When $foo is read, Nginx first checks the value in its container but sees the "invalid" special value, then Nginx decides to continue running $foo's "get handler", which first prints the warning (as shown above) and then returns an empty string value, which thereafter gets cached in $foo's value container.

Careful readers should have identified that this process for user variables is exactly the same as the mechanism we discussed earlier for built-in variables involving "get handlers" and result caching in value containers. Yes, it is the same mechanism in action. It is also worth noting that only the "invalid" special value will trigger the "get handler" invocation in the Nginx core while "not found" will not.

The warning message above usually indicates a typo in the variable name or misuse of uninitialized variables, not necessarily in the context of an interpolated string. Because of the existence of value caching in the variable container, this warning will not get printed multiple times in the lifetime of the current request. Also, the ngx_rewrite module provides the uninitialized_variable_warn directive for disabling this warning altogether.

Testing Special Values of Nginx Variables in Lua

As we have just mentioned, the built-in variable $arg_XXX takes the special value "not found" when the URL argument XXX does not exist, but unfortunately, it is not easy to distinguish it from the empty string value directly in the Nginx configuration file, for example:

```
location /test {
    echo "name: [$arg_name]";
}
```

Here we intentionally omit the URL argument name in our request:

```
$ curl 'http://localhost:8080/test'
name: []
```

We can see that we are still getting an empty string value, because this time it is the Nginx "script engine" that automatically converts the "not found" special value to an empty string when performing variable interpolation.

Then how can we test the special value "not found"? Or in other words, how can we distinguish it from normal empty string values? Obviously, in the following example, the URL argument `name` does take an ordinary value, which is a true empty string:

```
$ curl 'http://localhost:8080/test?name='
name: []
```

But we cannot really differentiate this from the earlier case that does not mention the `name` argument at all.

Luckily, we can easily achieve this in Lua by means of the 3rd-party module [ngx_lua](). Please look at the following example:

```
location /test {
    content_by_lua '
      if ngx.var.arg_name == nil then
        ngx.say("name: missing")
      else
        ngx.say("name: [", ngx.var.arg_name, "]")
      end
    ';
}
```

This example is very close to the previous one in terms of functionality. We use the [content_by_lua]() directive from the [ngx_lua]() module to embed a small piece of our own Lua code to test against the special value of the Nginx variable `$arg_name`. When `$arg_name` takes a special value (either "not found" or "invalid"), we will get the following output when requesting `/foo`:

```
$ curl 'http://localhost:8080/test'
name: missing
```

This is our first time meeting the [ngx_lua]() module, which deserves a brief introduction. This module embeds the Lua language interpreter (or LuaJIT's Just-in-Time compiler) into the Nginx core, to allow Nginx users directly run their own Lua programs inside the server. The user can choose to insert her Lua code into different running phases of the server, to fulfill different requirements. Such Lua code are either specified directly as literal strings in the Nginx configuration file, or reside in external `.lua` source files (or Lua binary bytecode files) whose paths are specified in the Nginx configuration.

Back to our example, we cannot directly write something like `$arg_name` in our Lua code. Instead, we reference Nginx variables in Lua by means of the `ngx.var` API provided by the [ngx_lua]() module. For example, to reference the Nginx variable `$VARIABLE` in Lua, we just write [ngx.var.VARIABLE](). When the Nginx variable `$arg_name` takes the special value "not found" (or "invalid"), `ngx.var.arg_name` is evaluated to the `nil` value in the Lua world. It should also be noting that we use the Lua function [ngx.say]() to print out the response body contents, which is functionally equivalent to the [echo]() directive we are already very familiar with.

If we provide a `name` URI argument that takes an empty value in the request, the output is now very different:

```
$ curl 'http://localhost:8080/test?name='
name: []
```

In this test, the value of the Nginx variable `$arg_name` is a true empty string, neither "not found" nor "invalid". So in Lua, the expression `ngx.var.arg_name` evaluates to the Lua empty string (`""`), clearly distinguished from the Lua `nil` value in the previous test.

This differentiation is important in certain application scenarios. For instance, some web services have to decide whether to use a column value to filter the data set by checking the *existence* of the corresponding URI argument. For these serives, when the `name` URI

argument is absent, the whole data set are just returned; when the `name` argument takes an empty value, however, only those records that take an empty value are returned.

It is worth mentioning a few limitations in the standard $arg_XXX variable. Consider using the following request to test /test in our previous example using Lua:

```
$ curl 'http://localhost:8080/test?name'
name: missing
```

Now the `$arg_name` variable still reads the "not found" special value, which is apparently counter-intuitive. Additionally, when multiple URI arguments with the same name are specified in the request, $arg_XXX just returns the first value of the argument, discarding other values silently:

```
$ curl 'http://localhost:8080/test?name=Tom&name=Jim&name=Bob'
name: [Tom]
```

To solve these problems, we can use the Lua function ngx.req.get_uri_args provided by the ngx_lua module instead.

Nginx Variables (08)

In (02) we mentioned that another category of builtin variables $cookie_XXX are like $arg_XXX. Similarly when there exist no cookie named `XXX`, its corresponding Nginx variable $cookie_XXX has non-value "not found".

```
location /test {
    content_by_lua '
      if ngx.var.cookie_user == nil then
        ngx.say("cookie user: missing")
      else
        ngx.say("cookie user: [", ngx.var.cookie_user, "]")
      end
    ';
}
```

The `curl` utility offers the `--cookie name=value` option, which designates `name=value` as a cookie of its request (by adding the `Cookie` header). Let's test a few cases containing cookies.

```
$ curl --cookie user=agentzh 'http://localhost:8080/test'
cookie user: [agentzh]

$ curl --cookie user= 'http://localhost:8080/test'
cookie user: []

$ curl 'http://localhost:8080/test'
cookie user: missing
```

As expected, when cookie `user` does not exist, Lua variable `ngx.var. cookie_user` is `nil`. So we have successfully distinguished the case with empty string and the case with non-value.

A nice add-on with module ngx_lua is when lua references an undeclared variable of Nginx, the variable is `nil` and Nginx will not aborts it loading as before.

```
location /test {
    content_by_lua '
      ngx.say("$blah = ", ngx.var.blah)
    ';
}
```

User variable `$blah` is never declared in the Nginx configuration `nginx. conf`, but it is referenced as `ngx.var.blah` in Lua code. Nginx can be started still, because when Nginx loads its configuration, Lua code is only compiled but not executed, So Nginx has no idea a variable `$blah` is referenced. When lua command is executed in run time by command content_by_lua, the lua variable is evaluated as `nil`. Module ngx_lua and its command ngx.say will convert Lua `nil` into string `"nil"` before it is printed, so the output will be:

```
curl 'http://localhost:8080/test'
$blah = nil
```

This is indeed what we want.

We should have noticed also, when command content_by_lua includes $blah in its parameter, it is never evaluated as "variable interpolation" does (otherwise Nginx will be complaining variable $blah is not declared). This is because command content_by_lua does not really support "variable interpolation" . As we have said earlier in (01), Nginx command does not necessarily support "variable interpolation" and it is entirely up to the module implementation.

It's actually difficult to return an "invalid" non-value. As we learnt in(07), variables which are declared but not initialized by set has non-value "invalid". However, as soon as the variable is devalued, the "get handler" is executed and an empty string is computed and cached, so eventually empty string is returned, not the "invalid" non-value. Following lua code can prove this:

```
location /foo {
    content_by_lua '
      if ngx.var.foo == nil then
          ngx.say("$foo is nil")
      else
          ngx.say("$foo = [", ngx.var.foo, "]")
      end
    ';
}

location /bar {
    set $foo 32;
    echo "foo = [$foo]";
}
```

By requesting to location /foo we have:

```
$ curl 'http://localhost:8080/foo'
$foo = []
```

As we can tell, when Lua references uninitialized Nginx variable $foo , it obtains empty string.

Last not the least, we should have pointed out, although Nginx variable can have only strings as valid value. The 3rd party module ngx_array_var can support array like operations for Nginx variable.Here is an example:

```
location /test {
    array_split "," $arg_names to=$array;
    array_map "[$array_it]" $array;
    array_join " " $array to=$res;

    echo $res;
}
```

Module ngx_array_var provides commands array_split , array_map and array_join . The semantics is pretty close to the builtin functions split , map and join in Perl (other languages support similar functionalities too). Now let's check what happens when location /test is requested:

```
$ curl 'http://localhost:8080/test?names=Tom,Jim,Bob'
[Tom] [Jim] [Bob]
```

Clearly module ngx_array_var make it easier to handle inputs with variable length, such as the URL parameter name , which composes of multiple comma delimited names. Still we must emphasize, module ngx_lua is a much better choice to execute this kind of complicated tasks, usually it is more flexible and maintainable.

Till now the tutorial covers the Nginx variable. In the process we have been discussing many builtin and 3rd party Nginx modules, these modules help us better understand features and internals of Nginx variable by composing various mini constructs. Later on the tutorial will be covering more details of those modules.

With these examples, we should understand that Nginx variable plays a key role in the Nginx mini language: variables are the ways and means Nginx communicate internally, they contain all the needed information (including the request information) and they are the cornerstone elements which bridge every other Nginx modules. Nginx variables are everywhere in the coming tutorials, understand them is absolutely necessary.

In the coming tutorial "Nginx Directive Execution Order", we will be discussing in detail the Nginx execution ordering and the phases every request traverses. It' s indispensable to understand them since for the Nginx mini language, the ordering of writing can be dramatically different from the ordering of executing in the timeline. It usually confuses many Nginx users.

Nginx directive execution order (01)

When there are multiple Nginx module commands in a `location` directive, the execution order can be different from what you expect. Busy Nginx users who attempt to configure Nginx by "trial and error" may be very confused by this behavior. This series is to uncover the mysteries and help you better understand the execution ordering behind the scenes.

We start with a confused example:

```
? location /test {
?     set $a 32;
?     echo $a;
?
?     set $a 56;
?     echo $a;
? }
```

Clearly, we'd expect to output `32`, followed by `56`. Because variable `$a` has been reset after command echo "is executed". Really? the reality is:

```
$ curl 'http://localhost:8080/test
56
56
```

Wow, statement `set $a 56` must have had been executed before the first `echo $a` command, but why? Is it a Nginx bug?

No, this is not an Nginx bug. When Nginx handles every request, the execution follows a few predefined phases.

There can be altogether 11 phases when Nginx handles a request, let's start with three most common ones: `rewrite`, `access` and `content` (The other phases will be addressed later.)

Usually an Nginx module and its commands register their execution in only one of those phases. For example command set runs in phase `rewrite`, and command echo runs in phase `content`. Since phase `rewrite` occurs before phase `content` for every request processing, its commands are executed earlier as well. Therefore, command set always gets executed before command echo within one `location` directive, regardless of their statement ordering in the configuration.

Back to our example:

```
set $a 32;
echo $a;

set $a 56;
echo $a;
```

The actual execution ordering is:

```
set $a 32;
set $a 56;
echo $a;
echo $a;
```

It's clear now, two commands set are executed in phase `rewrite`, two commands echo are executed afterwards in phase `content`. Commands in different phases cannot be executed

back and forth.

To prove this, we can enable Nginx's "debug log".

If you have not worked with Nginx "debug log" before, here is a brief introduction. The "debug log" is disabled by default because performance is degraded when it is enabled. To enable "debug log" you must reconfigure and recompile Nginx, and set the `--with-debug` option for the package's `./configure` script. When building under Linux or Mac OS X from source:

```
tar xvf nginx-1.0.10.tar.gz
cd nginx-1.0.10/
./configure --with-debug
make
sudo make install
```

In case the package `ngx_openresty` is used. The option `--with-debug` can be used with its `./configure` script as well.

After we rebuild the Nginx debug binary with `--with-debug` option, we still need to explicitly use the `debug` log level (it's the lowest level) for command `error_log`, in Nginx configuration:

```
error_log logs/error.log debug;
```

`debug`, the second parameter of command `error_log` is crucial. Its first parameter is error log's file path, `logs/error.log`. Certainly we can use another file path but do remember the location because we need to check its content right away.

Now let's restart Nginx (Attention, it's not enough to reload Nginx. It needs to be killed and restarted because we've updated the Nginx binary). Then we can send the request again:

```
$ curl 'http://localhost:8080/test'
56
56
```

It's time to check Nginx's error log, which is becoming a lot more verbose (more than 700 lines for the request in my setup). So let's apply the `grep` command to filter what we would be interested:

```
grep -E 'http (output filter|script (set|value))' logs/error.log
```

It's approximately like below (for clearness, I've edited the `grep` output and remove its timestamp etc) :

```
[debug] 5363#0: *1 http script value: "32"
[debug] 5363#0: *1 http script set $a
[debug] 5363#0: *1 http script value: "56"
[debug] 5363#0: *1 http script set $a
[debug] 5363#0: *1 http output filter "/test?"
[debug] 5363#0: *1 http output filter "/test?"
[debug] 5363#0: *1 http output filter "/test?"
```

It barely makes any senses, does it? So let me interpret. Command `set` dumps two lines of debug info which start with `http script`, the first line tells the value which command `set` has possessed, and the second line being the variable name it will be given to, so for the leading filtered log:

```
[debug] 5363#0: *1 http script value: "32"
[debug] 5363#0: *1 http script set $a
```

These two lines are generated by this statement:

```
set $a 32;
```

And for the following filtered log:

```
[debug] 5363#0: *1 http script value: "56"
[debug] 5363#0: *1 http script set $a
```

They are generated by this statement:

```
set $a 56;
```

Besides, whenever Nginx outputs its response, its "output filter" will be executed, our favorite command echo is no exception. As soon as Nginx's "output filter" is executed, it generates debug log like below:

```
[debug] 5363#0: *1 http output filter "/test?"
```

Of course the debug log might not have `/test?`, since this part corresponds to the actual request URI. By putting everything together, we can finally conclude those two commands set are indeed executed before the other two commands echo.

Considerate readers must have noticed that there are three lines of `http output filter` debug log but we were having only two output commands echo. In fact, only the first two debug logs are generated by the two echo statements. The last debug log is added by module ngx_echo because it needs to flag the end of output. The flag operation itself causes Nginx's "output filter" to be executed again. Many modules including ngx_proxy has similar behavior, when they need to give output data.

All right, there are no surprises with those duplicated `56` outputs. We are not given a chance to execute echo in front of the second set command. Luckily, we can still achieve this with a few techniques:

```
location /test {
    set $a 32;
    set $saved_a $a;
    set $a 56;

    echo $saved_a;
    echo $a;
}
```

Now we have what we have wanted:

```
$ curl 'http://localhost:8080/test'
32
56
```

With the help of another user variable `$saved_a`, the value of `$a` is saved before it is overwritten. Be careful, the execution order of multiple set commands are ensured to be like their order of writing by module . Similarly, module ngx_echo ensures multiple echo commands get executed in the same order of their writing.

If we recall examples in Nginx Variables, this technique has been applied extensively. It bypasses the execution ordering difficulties introduced by Nginx phased processing.

You might need to ask : "how would I know the phase a Nginx command belongs to ?" Indeed, the answer is RTFD. (Surely advanced developers can examine the C source code directly). Many module marks explicitly its applicable phase in the module's documentation, such as command echo writes below in its documentation:

```
phase: content
```

It says the command is executed in phase `content`. If you encounters a module which misses the applicable phase in the document, you can write to its authors right away and ask for it. However, we shall be reminded, not every command has an applicable phase. Examples are command geo introduced in Nginx Variables (01) and command map introduced in Nginx Variables (04). These commands, who have no explicit applicable phase, are declarative and unrelated to the conception of execution ordering. Igor Sysoev, the author of Nginx, has made the statements a few times publicly, that Nginx mini language in its configuration is "declarative" not "procedural".

Nginx directive execution order (02)

We've just learnt, all set commands within location are executed in rewrite phase. In fact, almost all commands implemented by module rewrite are executed in rewrite phase under the specific context. Commad rewrite introduced in Nginx Variables (02) is one of them. However, we shall point out that when these commands are found in server directive, they will be executed in an earlier phase we've not addressed: the server rewrite phase.

Command set_unescape_uri, introduced in Nginx Variables (02) is also executed in rewrite phase. Actually, commands implemented by module ngx_set_misc can mix with commands implemented by module ngx_rewrite and the execution ordering is ensured. Let's check an example:

```
location /test {
    set $a "hello%20world";
    set_unescape_uri $b $a;
    set $c "$b!";

    echo $c;
}
```

By sending a request accordingly we have:

```
$ curl 'http://localhost:8080/test'
hello world!
```

Apparently, the set_unescape_uri command and its neighboring set commands are all executed in the order of their writing.

To further demonstrate our assertion, we check again Nginx "debug log" (in case it's unclear for you how to check "debug log", please reference steps found in (01)).

```
grep -E 'http script (value|copy|set)' logs/error.log
```

The debug logs are filtered as:

```
[debug] 11167#0: *1 http script value: "hello%20world"
[debug] 11167#0: *1 http script set $a
[debug] 11167#0: *1 http script value (post filter): "hello world"
[debug] 11167#0: *1 http script set $b
[debug] 11167#0: *1 http script copy: "!"
[debug] 11167#0: *1 http script set $c
```

The leading two lines:

```
[debug] 11167#0: *1 http script value: "hello%20world"
[debug] 11167#0: *1 http script set $a
```

They correspond to the command

```
set $a "hello%20world";
```

The following two lines:

```
[debug] 11167#0: *1 http script value (post filter): "hello world"
[debug] 11167#0: *1 http script set $b
```

They are generated by command

```
set_unescape_uri $b $a;
```

There are minor differences in the first line, if we compare to the logs generated by command set: the "(post filter)" addition. In the end of the line, URL decoding has successfully

executed as we wish. `"hello%20world"` is decoded as `"hello world"`.

The last two lines of debug log:

```
[debug] 11167#0: *1 http script copy: "!"
[debug] 11167#0: *1 http script set $c
```

They are generated by the last [set](#) command

```
set $c "$b!";
```

As you might have noticed, since "variable interpolation" is evaluated when variable `$c` is declared and initialized, the debug log starts with `http script copy`. In the end of the log it is the string constant `"!"` to be concatenated.

With the log information, it's fairly easy to tell the command execution ordering:

```
set $a "hello%20world";
set_unescape_uri $b $a;
set $c "$b!";
```

It is a perfect match to the statements ordering.

Just like the commands implemented in module [ngx_set_misc](#), command [set_by_lua](#) implemented in 3rd party module [ngx_lua](#), can mix with commands of module [ngx_rewrite](#) as well. As introduced in [Nginx Variables (07)](#), command [set_by_lua](#) supports computation with given Lua code, and assigns the computed result to a Nginx variable. As command [set](#) does, command [set_by_lua](#) declares Nginx variable before initialization if the variable does not exist.

Let's check a mixed example which comprises command [set_by_lua](#) and [set](#):

```
location /test {
    set $a 32;
    set $b 56;
    set_by_lua $c "return ngx.var.a + ngx.var.b";
    set $equation "$a + $b = $c";

    echo $equation;
}
```

Variable `$a` and `$b` are initialized with numerical value `32` and `56` respectively, then command [set_by_lua](#) is used together with given Lua code to compute the sum of `$a` and `$b`. Variable `$c` is initialized with the computed value. Finally, variables `$a`, `$b` and `$c` are concatenated by "variable interpolation" and assigns the result to variable `$equation`, which is printed by command [echo](#).

We shall pay attention to a few points in the example: Firstly Nginx variable `$VARIABLE` is referenced as [ngx.var.VARIABLE](#) in Lua code. Secondly, since Nginx variables are strings, the value of variable `ngx.var.a` and `ngx.var.b` are actually strings `"32"` and `"56"`, however they are automatically converted to numerical values by Lua in the addition operation. Thirdly Lua code returns to Nginx variable `$c` the computed sum value by statement `return`. Finally when Lua code returns, it actually converts the numerical value back to string. (because string is the only valid value for Nginx variable)

The actual output meets our expectation:

```
$ curl 'http://localhost:8080/test'
32 + 56 = 88
```

This in fact asserts that command [set_by_lua](#) can mix with commands implemented by module [ngx_rewrite](#), such as [set](#).

Many other 3rd party modules support the mix with module [ngx_rewrite](#) as well. The examples include module [ngx_array_var](#), discussed in [Nginx Variables (08)](#) and module [ngx_encrypted_session](#), which encrypts sessions. The latter will be studied in detail shortly.

Since builtin module [ngx_rewrite](#) is virtually indispensable, it's a great advantage for the 3rd party module has the caliber of being mixed with. Truth is, all of those 3rd party modules

have adopted a special technique, which allows the "injection" of their execution into commands of module `rewrite` (with the help of a 3rd party module `ngx_devel_kit` developed by Marcus Clyne). For the rest regular 3rd party modules, which also register their execution in phase `rewrite`, their commands are executed separately from module `ngx_rewrite` in runtime. In fact, it's hardly accurate to tell the commands execution ordering in between different modules (strictly speaking they are usually executed in the order of loading, but exception does exist). For example both modules, `A` and `B` register their commands to be executed in phase `rewrite`, then it is either the case in which commands of `A` are executed followed by `B` or the other complete way around. Unless it is explicitly documented, we cannot rely on the uncertain ordering in our configurations.

Nginx directive execution order (03)

As discussed earlier, unless special techniques are utilized as module `ngx_set_misc` does, a module can not mix its commands with `ngx_rewrite`, and expects the correct execution order. Even if the commands are registered in the `rewrite` phase as well. We can demonstrate with some examples.

3rd party module `ngx_headers_more` provides a few commands, which deal with the current request header and response header. One of them is `more_set_input_header`. The command can modify a given request header in `rewrite` phase (or add the specific header if it's not available in current request). As described in its documentation, the command always executes in the end of `rewrite` phase:

```
phase: rewrite tail
```

Being terse though, `rewrite tail` means the end of phase `rewrite`.

Since it executes in the end of phase `rewrite`, the implication is its execution is always after the commands implemented in module `ngx_rewrite`. Even if it is written at the very beginning:

```
? location /test {
?     set $value dog;
?     more_set_input_headers "X-Species: $value";
?     set $value cat;
?
?     echo "X-Species: $http_x_species";
? }
```

As briefly introduced in Nginx Variables (02), Builtin variable `$http_XXX` has the header `XXX` for the current request. We must be careful though, variable <$http_XXX> matches to the normalized request header, i.e. it lower cases capital letters and turns minus `-` into underscore `_` for the request header names. Therefore variable `$http_x_species` can successfully catches the request header `X-Species`, which is declared by command more_set_input_header.

Because of the statement ordering, we might have mistakenly concluded header `X-Species` has the value `dog` when `/test` is requested. But the actual result is different:

```
$ curl 'http://localhost:8080/test'
X-Species: cat
```

Clearly, statement `set $value cat` is executed earlier than more_set_input_headers, although it is written afterwards.

This example tells us that commands of different modules are executed independently from each other, even if they are all registered in the same processing phase. (unless it is implemented as module ngx_set_misc, whose commands are specifically tuned with module ngx_rewrite). In other words, every processing phase is further divided into sub-phases by Nginx modules.

Similar to more_set_input_headers, command rewrite_by_lua provided by 3rd party module ngx_lua execute in the end of `rewrite` phase as well. We can verify this:

```
? location /test {
?     set $a 1;
?     rewrite_by_lua "ngx.var.a = ngx.var.a + 1";
?     set $a 56;
?
```

```
?    echo $a;
? }
```

By using Lua code specified by command rewrite_by_lua Nginx variable `$a` is incremented by 1.We might have expected the result be `56` if we are looking at the writing sequence.The actual result is `57` because command is always executed after all the set statements.

```
$ curl 'http://localhost:8080/test'
57
```

Admittedly command rewrite_by_lua has different behavior than command set_by_lua, which is discussed in (02).

Out of sheer curiosity, we shall ask immediately that what would be execution ordering in between more_set_input_headers and rewrite_by_lua, since they both ride on `rewrite` tail? The answer is : undefined. We must avoid a configuration which relies on their execution orders.

Nginx phase `rewrite` is a rather early processing phase. Usually commands registered in this phase execute various rewrite tasks on the request (for example rewrite the URL or the URL parameters), the commands might also declare and initialize Nginx variables which are needed in the subsequent handling. Certainly, one cannot forbid others to complicate themselves by checking the request body, or visit a database etc. After all, command like rewrite_by_lua offers the caliber to stuff in any potentially mind twisted Lua code.

After phase `rewrite`, Nginx has another phase called `access`. The commands provided by 3rd party module ngx_auth_request, which is discussed in Nginx Variables (05), execute in phase `access`. Commands registered in `access` phase mostly carry out ACL functionalities, such as guarding user clearance, checking user origins, examining source IP validity etc.

For example command allow and deny provided by builtin module ngx_access can control which IP addresses have the privileges to visit, or which IP addresses are rejected:

```
location /hello {
    allow 127.0.0.1;
    deny all;

    echo "hello world";
}
```

Location `/hello` allows visit from localhost (IP address `127.0.0.1`) and reject requests from all other IP addresses (returns http error `403`) The rules defined by ngx_access commands are asserted in the writing sequence. Once one rule is matched, the assertion stops and all the rest allow or deny commands are ignored. If no rule is matched, handling continues in the following statements. If the matched rule is deny, handing is aborted and error `403` is returned immediately. In our example, request issued from localhost matches to the rule `allow 127.0.0.1` and handing continues to the other statements, however request issued from every other IP addresses will match rule `deny all` handling is therefore aborted and error `403` is returned.

We can give it a test, by sending request from localhost:

```
$ curl 'http://localhost:8080/hello'
hello world
```

If request is sent from another machine (suppose Nginx runs on IP `192.168.1.101`) we have:

```
$ curl 'http://192.168.1.101:8080/hello'
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

By the way, module ngx_access supports the "CIDR notation" to designate a sub-network. For example `169.200.179.4/24` represents the sub-network which has the routing prefix `169.200.179.0` (or subnet mask `255.255. 255.0`)

Because commands of module ngx_access execute in `access` phase, and phase `access` is behind `rewrite` phase. So for those commands we have been discussing, regardless of the writing order they always execute in `rewrite` phase, which is earlier than allow or deny. Keep this in mind, we shall try our best to keep the writing and execution order consistent.

Nginx directive execution order (04)

Module ngx_lua implements another command access_by_lua. The command allows lua code to be executed in the end of `access` phase, which means it always executes after allow and deny even they belong to the same phase. In many cases, we examine the request' s source IP address with ngx_access, and use command access_by_lua to execute more complicated verifications with Lua. For example by querying a database or other backend services, the current user's identity and privileges are examined.

We can check a simple example, which uses command access_by_lua to implement the IP filtering functionality of module ngx_access

```
location /hello {
    access_by_lua '
        if ngx.var.remote_addr == "127.0.0.1" then
            return
        end

        ngx.exit(403)
    ';

    echo "hello world";
}
```

Nginx's builtin variable $remote_addr is referenced in Lua to get the client's IP address. Then Lua statement `if` is used to determine if the address equals `127.0.0.1`. Lua returns if it equals, Nginx thus continues the subsequent handling (including the `content` phase where command echo applies to). If it is not the localhost address, current handling is aborted by using ngx_lua module's Lua function ngx.exit Client gets a http error `403`.

The example is equivalent to the other example using ngx_access module in terms of functionality, which was discussed in (03):

```
location /hello {
    allow 127.0.0.1;
    deny all;

    echo "hello world";
}
```

However we shall point out, performance wise the two still have differences. Module ngx_access performs better because it is specifically implemented as a Nginx module in C.

We can measure the performance differences of the two. After all, performance is what we are after by using Nginx. On the other hand, it's absolutely necessary to be equipped with measuring techniques, because only actual data distinguishes amateurs and professionals. In fact, both ngx_lua and ngx_access perform pretty good for IP filtering. To minimize measuring errors we could measure directly the elapsed time of `access` phase. Traditionally, this means hacking Nginx source code with timing code and statistical code, or recompile Nginx binary so that it can be monitored by specific profiling tools like `GNU gprof`.

We are lucky, because current releases of Solaris, Mac OSX or FreeBSD offer a system utility `dtrace`, which allows micro monitoring of user process in terms of performance (and functionality as well). The tool spares us from hacking source code or recompilation with profiling. Let's demonstrate the measuring scenario on the MacBook Air because `dtrace` is available since Mac OS X 10.5

First, open the Terminal application of Mac OSX, change to your preferable path and create a file named as `nginx-access-time.d`, edit the file with following content:

```
#!/usr/bin/env dtrace -s

pid$1::ngx_http_handler:entry
{
    elapsed = 0;
}
```

```
pid$1::ngx_http_core_access_phase:entry
{
    begin = timestamp;
}

pid$1::ngx_http_core_access_phase:return
/begin > 0/
{
    elapsed += timestamp - begin;
    begin = 0;
}

pid$1::ngx_http_finalize_request:return
/elapsed > 0/
{
    @elapsed = avg(elapsed);
    elapsed = 0;
}
```

Save the file and make it executable.

```
$ chmod +x ./nginx-access-time.d
```

The `.d` file actually contains code written in `D` language offered by utility `dtrace` (attention, the `D` language is not the other `D` language, which is advocated by Walter Bright for a better C++). So far we cannot really explain in detail the code because it requires a thorough understanding of Nginx internals. Anyway we shall be clear of the code's purpose: measure requests being handled by specific Nginx worker process and calculate the average time elapsed in `access` phase.

Now we can get the `D` script running. The script takes a command line parameter, which is the process id (pid) of Nginx worker. Since Nginx supports multiple worker processes and the requests can be randomly handled by anyone of them, we'd like to configure Nginx in its configuration `nginx.conf` so that only one worker is requested.

```
worker_processes 1;
```

After Nginx binary is restarted, the worker process id can be obtained by command `ps`.

```
$ ps ax|grep nginx|grep worker|grep -v grep
```

Typically we have:

```
10975   ??  S     0:34.28 nginx: worker process
```

`10975` is my Nginx worker pid. In case you have multiple lines, you must have started multiple Nginx server instances or the current Nginx server has started multiple worker processes.

Then as root, script `nginx-access-time.d` is executed with the worker pid

```
$ sudo ./nginx-access-time.d 10975
```

We shall have one output message if everything goes OK.

```
dtrace: script './nginx-access-time.d' matched 4 probes
```

The message says our `D` script has successfully deployed 4 probes on the target process. Then the script is ready to trace process `10975` constantly.

Let's open another Terminal, and send multiple requests with `curl` to our monitored process

```
$ curl 'http://localhost:8080/hello'
hello world
```

```
$ curl 'http://localhost:8080/hello'
hello world
```

Back to our Terminal where `D` script is running, press keys `Ctrl-C` to interrupt it. When the script bails out it prints on console the statistical result. For example my console has following result:

```
$ sudo ./nginx-access-time.d 10975
dtrace: script './nginx-access-time.d' matched 4 probes
^C
    19219
```

The final `19219` is the average time elapsed in `access` phase in nano seconds (1 second = 1000x1000x1000 nano seconds)

Done with the steps. We can run the `nginx-access-time.d` script to calculate average elapsed time in phase `access` for three different Nginx setups respectively. They are IP filtering with module [ngx_access](), IP filtering with command [access_by_lua](), and finally no filtering for `access` phase. The last result helps eliminate the side effect caused by probes or other "systematic errors". Besides, we can use traffic loader tools such as `ab` to sends half a million requests to minimize "random errors", as below:

```
$ ab -k -c1 -n100000 'http://127.0.0.1:8080/hello'
```

Therefore the statistical result of `D` script is as close as possible to the "actual" time.

In the Mac OSX, a typical run has following results:

```
ngx_access              18146
access_by_lua               35011
no filtering           15887
```

We minus the last value from the former two:

```
ngx_access           2259
access_by_lua          19124
```

Well, module [ngx_access]() out performs command [access_by_lua]() by a magnitude, as we might have expected. Still the absolute difference is tiny. For the `Intel Core2Due 1.86 GHz` CPU of mine, there is only a few micro seconds.

In fact the [access_by_lua]() example can be further optimized using builtin variable [$binary_remote_addr](). This variable has the IP address in binary form whereas variable [$remote_addr]() has the address in a longer string format. Shorter address can be compared quicker when Lua executes its string operations.

Be careful, if "debug log" is enabled as introduced in[(01)]() the computed elapsed time will increase dramatically, because "debug log" has a huge overhead.

Nginx directive execution order (05)

`content` is by all means the most significant phase in Nginx's request handling, because commands running in the phase have the responsibility to generate "content" and output HTTP response. Because of its importance, Nginx has a rich set of commands running in it. The commands include [echo](), [echo_exec](), [proxy_pass](), [echo_location](), [content_by_lua](), which were discussed in [Nginx Variables (02)](), [Nginx Variables (03)](), [Nginx Variables (05)]() and [Nginx Variables (07)]() respectively.

`content` is a phase which runs later than `rewrite` and `access`. Therefore its commands always execute in the end when they are used together with commands of `rewrite` and `access`.

```
location /test {
    # rewrite phase
    set $age 1;
    rewrite_by_lua "ngx.var.age = ngx.var.age + 1";

    # access phase
```

```
    deny 10.32.168.49;
    access_by_lua "ngx.var.age = ngx.var.age * 3";

    # content phase
    echo "age = $age";
}
```

This is a perfect example, in which commands are executed in an exact sequence as they are written. The testing result matches to our expectations too.

```
$ curl 'http://localhost:8080/test'
age = 6
```

In fact, the commands' writing order can be completely shuffled and it won't have any impact to their execution sequence. Command set, which is implemented by module ngx_rewrite, executes in rewrite phase. Command rewrite_by_lua from module ngx_lua executes in the end of rewrite phase. Command deny from module ngx_access executes in access phase. Command access_by_lua from module ngx_lua executes in the end of access phase. Finally, our favorite command echo, implemented by module ngx_echo, executes in content phase.

The example also demonstrates the collaborating in between commands running on each different Nginx phase. In the process, Nginx variable is the data carrier interconnecting commands and modules. The execution order of these commands is largely decided by the phase each applies to.

As matter of fact, multiple commands from different modules could coexist in phase rewrite and access. As the example shows, command set and command rewrite_by_lua both belong to phase rewrite. Command deny and command access_by_lua both belong to phase access. However it is not the same story for phase content.

Most modules, when they implement commands for phase content, they are actually inserting "content handler" for the current location directive, however there can be one and only one "content handler" for a location. So only one module could beat the rest when multiple modules are contending the role. Consider following problematic example:

```
? location /test {
?     echo hello;
?     content_by_lua 'ngx.say("world")';
? }
```

Command echo from module ngx_echo and command content_by_lua from module ngx_lua both execute in phase content. But only one of them could successfully become "content handler":

```
$ curl 'http://localhost:8080/test'
world
```

Our test indicates, that the winner is content_by_lua although it is written afterwards, and command echo never really has a chance to run. We cannot be assured which module wins in the circumstance. For example, module ngx_echo wins and the output becomes hello if we swap the content_by_lua and echo statements. So we shall avoid to use multiple commands for phase content, if the commands are implemented by different modules.

The example can be modified by replacing command content_by_lua with command echo and we will get what we need:

```
location /test {
    echo hello;
    echo world;
}
```

Again test proves:

```
$ curl 'http://localhost:8080/test'
hello
world
```

We can use multiple echo commands, there is no problem with this because they all belong

to module ngx_echo. Module ngx_echo regulates the execution ordering of them. Be careful though, not every module supports the commands being executed multiple times within one `location`. Command content_by_lua for an instance, can be used only once, so following example is incorrect:

```
? location /test {
?     content_by_lua 'ngx.say("hello")';
?     content_by_lua 'ngx.say("world")';
? }
```

Nginx dumps error for the configuration:

```
[emerg] "content_by_lua" directive is duplicate ...
```

The correct way of doing it is:

```
location /test {
    content_by_lua 'ngx.say("hello") ngx.say("world")';
}
```

Instead of using twice the content_by_lua command in `location`, the approach is to call function ngx.say twice in the Lua code, which is executed by command content_by_lua

Similarly, command proxy_pass from module ngx_proxy cannot coexist with command echo within one `location` because they both execute in `content` phase. Many Nginx newbies make following mistake:

```
? location /test {
?     echo "before...";
?     proxy_pass http://127.0.0.1:8080/foo;
?     echo "after...";
? }
?
? location /foo {
?     echo "contents to be proxied";
? }
```

The example tries to output strings `before...` and `after...` with command echo before and after module ngx_proxy returns its content. However only one module could execute in `content`. The test indicates module ngx_proxy wins and command echo from module ngx_echo never runs

```
$ curl 'http://localhost:8080/test'
contents to be proxied
```

To implement what the example had wanted to, we shall use two other commands provided by module ngx_echo, echo_before_body and echo_after_body:

```
location /test {
    echo_before_body "before...";
    proxy_pass http://127.0.0.1:8080/foo;
    echo_after_body "after...";
}

location /foo {
    echo "contents to be proxied";
}
```

Test tells we make it:

```
$ curl 'http://localhost:8080/test'
before...
contents to be proxied
after...
```

The reason commands echo_before_body and echo_after_body could coexist with other modules in `content` phase, is they are not "content handler" but "output filter" of Nginx. Back in (01) when we examine the "debug log" generated by command echo , we've learnt Nginx calls its "output filter" whenever Nginx outputs data. So that module ngx_echo takes the advantage of it to modify content generated by module ngx_proxy (by adding surrounding content). We shall point out though, "output filter" is not one of those 11 phases mentioned in (01) (many phases could trigger "output filter" when they output data). Still it's perfectly all right to document commands echo_before_body and echo_after_body as following:

```
phase: output filter
```

It means the command executes in "output filter".

Nginx directive execution order (06)

We've learnt in (05) that when a command executes in `content` phase for a specific `location`, it usually means its Nginx module registers a "content handler" for the `location`. However, what happens if no module registers its command as "content handler" for phase `content` ? Who will be taking the glory of generate content and output responses ? The answer is the static resource module, which maps the request URI to the file system. Static resource module only comes into play when there is none "content handler", otherwise it hands off the duty to "content handler".

Typically Nginx has three static resource modules for the `content` phase (unless one or more of those modules are disabled explicitly, or some other conflicting modules are enabled when Nginx is built) The three modules, in the order of their execution order, are ngx_index module, ngx_autoindex module and `ngx_static` module. Let's discuss them one by one.

Module ngx_index and ngx_autoindex only apply to those request URI, which ends with `/` . For the other request URI which does not end with `/`, both modules ignore them and let the following `content` phase module handle. Module `ngx_static` however, has an exact opposite strategy. It ignores the request URI which ends with `/` and handles the rest.

Module ngx_index mainly looks for a specific home page file, such as `index.html` or `index.htm` in the file system. For example:

```
location / {
    root /var/www/;
    index index.htm index.html;
}
```

When address `/` is requested, Nginx looks for file `index.htm` and `index.html` (in this order) in a path in the file system. The path is specified by command root. If file `index.htm` exists, Nginx jumps internally to location `index.htm` ; if it does not exist and file `index.html` exists, Nginx jumps internally to location `index.html` . If file `index.html` does not exist either, and handling is transferred to the other module which executes it commands in phase `content` .

We have learnt in Nginx Variables (02), commands echo_exec and rewrite can trigger "internal redirects" as well. The jump modifies the request URI, and looks for the corresponding `location` directive for subsequent handling. In the process, phases `rewrite` , `access` and `content` are reiterated for the `location` . The "internal redirect" is different from the "external redirect" defined by HTTP response code 302 and 301, client browser won't update its URI addresses. Therefore as soon as internal jump occurs when module ngx_index finds the files specified by command index, the net effect is like client would have been requesting the file's URI at the very beginning.

We can check following example to witness the "internal redirect" triggered by module ngx_index, when it finds the needed file.

```
location / {
    root /var/www/;
    index index.html;
}

location /index.html {
    set $a 32;
    echo "a = $a";
}
```

We need to create an empty file `index.html` under the path `/var/www/` , and make sure the file is

readable for the Nginx worker process. Then we could send request to `/`:

```
$ curl 'http://localhost:8080/'
a = 32
```

What happened ? Why the output is not the content of file `index.html` (which shall be empty) ? Firstly Nginx uses directive `location /` to handle original `GET /` request, then module `ngx_index` executes in `content` phase, and it finds file `index.html` under path `/var/www/`. At this moment, it triggers an "internal redirect" to location `/index.html`.

So far so good. But here comes the surprises ! When Nginx looks for `location` directive which matches to `/index.html`, `location /index.html` has a higher priority than `location /`. This is because Nginx uses "longest matched substring" semantics to match `location` directives to request URI's prefix. When directive is chosen, phases `rewrite`, `access` and `content` are reiterated, and eventually it outputs `a = 32`.

What if we remove file `/var/www/index.html` in the example, and request to `/` again ? The answer is error `403 Forbidden`. Why? When module `ngx_index` cannot find the file specified by command `index` (`index.html` in here), it transfers the handling to the following module which executes in `content`. But none of those following modules can fulfill the request, Nginx bails out and dumps us error. Meanwhile it logs the error in Nginx error log:

```
[error] 28789#0: *1 directory index of "/var/www/" is forbidden
```

The meaning of `directory index` is to generate "indexes". Usually this implies to generate a web page, which lists every file and sub directories under path `/var/www/`. If we use module `ngx_autoindex` right after `ngx_index`, it can generate such a page just like what we need. Now let's modify the example a little bit:

```
location / {
    root /var/www/;
    index index.html;
    autoindex on;
}
```

When `/` is requested again meanwhile file `/var/www/index.html` is kept missing. A nice html page is generated:

```
$ curl 'http://localhost:8080/'
<html>
<head><title>Index of /</title></head>
<body bgcolor="white">
<h1>Index of /</h1><hr><pre><a href="../">../</a>
<a href="cgi-bin/">cgi-bin/</a>  08-Mar-2010 19:36   -
<a href="error/">error/</a>     08-Mar-2010 19:36   -
<a href="htdocs/">htdocs/</a>   05-Apr-2010 03:55   -
<a href="icons/">icons/</a>     08-Mar-2010 19:36   -
</pre><hr></body>
</html>
```

The page shows there are a few subdirectories under my `/var/www/`. They are `cgi-bin/`, `error/`, `htdocs/` and `icons/`. The output might be different if you have tried by yourself.

Again, if file `/var/www/index.hmtl` does exist, module `ngx_index` will trigger "internal redirect", and module `ngx_autoindex` will not have a chance to execute, you may test it yourself too.

The "goal keeper" module executed in phase `content` is `ngx_static`. which is also used intensively. The module serves the static files, including the static resources of a web site, such as static `.html` files, static `.css` files, static `.js` files and static image files etc. Although `ngx_index` could trigger an "internal redirect" to the specified home page, but the actual output task (takes the file content as response, and marks the corresponding response headers) is carried out by module `ngx_static`.

Nginx directive execution order (07)

Let's check an example in which module `ngx_static` serves disk files, with following configuration snippet:

```
location / {
    root /var/www/;
}
```

Meanwhile two files are created under `/var/www/`. One file is named `index.html` and its content contains one line of text `this is my home`. Another file is named `hello.html` and its content contains one line of text `hello world`. Again be aware of the files' privileges and make sure they are readable by Nginx worker process.

Now we send requests to the files' corresponding URI:

```
$ curl 'http://localhost:8080/index.html'
this is my home

$ curl 'http://localhost:8080/hello.html'
hello world
```

As we can see, the created file contents are sent as outputs.

We can examine what is happening here: `location /` does not have any command to execute in phase `content`, therefore no module has registered a "content handler" in the `location`. The handling thus falls to the three static resource modules which are the last resorts of phase `content`. The former two modules [ngx_index](#) and [ngx_autoindex](#) notices that the request URI does not end with `/` so they hand off immediately to module `ngx_static`, which runs in the end. According to the "document root" specified by command [root](#), module `ngx_static` maps the request URIs `/index.html` and `/hello.html` to disk files `/var/www/index.html` and `/var/www/hello.html` respectively. As both files can be found, their content are outputted as response, meanwhile response header `Content-Type`, `Content-Length` and `Last-Modified` are accordingly indicated.

To verify module `ngx_static` has executed, we could enable the "debug log" introduced in [(01)](#). Again we send request to `/index.html` and Nginx error log will contain following debug information:

```
[debug] 3033#0: *1 http static fd: 8
```

This line is generated by module `ngx_static`. Its meaning is " outputting static resource whose file handle is `8` ". Of course the numerical file handle changes every time, and the line is only a typical output in my setup. To be reminded, builtin module [ngx_gzip_static](#) could generate the same debug info as well, by default it is not enabled though, which will be discussed later.

Command [root](#) only declares a "document root", it does not enables the `ngx_static` module. The module is as matter of fact, always enabled already, but it might not have the chance to execute. This is entirely up to the other modules, which execute earlier in `content` phase. Module `ngx_static` execute only when all of them have "gave up". To prove this, check following blank `location` definition:

```
location / {
}
```

Because there is no [root](#) command, Nginx computes a default "document root" when the location is requested. The default shall be the `html/` subdirectory under "configure prefix". For example suppose our "configure prefix" is `/foo/bar/`, the default "document root" is `/foo/bar/html/`.

So who decides "configure prefix" ? Actually it the Nginx root directory when it is installed (or the value of `--prefix` option of script `./configure` when Nginx is built). If Nginx is installed into `/usr/local/nginx/`, "configure prefix" is `/usr/local/nginx/` and default "document root" is therefore `/usr/local/nginx/html/`. Certainly a command line option `--prefix` can be given when Nginx is started, to change the "configure prefix" (so that we can easily test multiple setups). Suppose Nginx is started as following:

```
nginx -p /home/agentzh/test/
```

For this server, its "configure prefix" becomes `/home/agentzh/test/` and its "document root" becomes `/home/agentzh/test/html/`. The "configure prefix" not only determines "document root", it actually determines the way many relational path resolutes to absolute path in Nginx configuration. We will encounter many examples which reference "configure prefix".

In fact there is a simple way of telling current "document root", which is to request a non-

existed file, Such as:

```
$ curl 'http://localhost:8080/blah-blah.txt'
<html>
<head><title>404 Not Found</title></head>
<body bgcolor="white">
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Naturally, the `404` error page is returned. Again when we check Nginx error log, we shall have following error message:

```
[error] 9364#0: *1 open() "/home/agentzh/test/html/blah-blah.txt" failed (2: No such file or directory)
```

The error message is printed by module `ngx_static`, since it cannot find a file `blah-blah.txt` in its corresponding path. And because the error message contains the absolute path, which `ngx_static` attempts to open with, it's quite obvious that current "document root" is `/home/agentzh/test/html/`.

Many newbies might take it for granted that error `404` is caused when the needed `location` does not exist. The former example tells us, `404` error could be returned even if the needed `location` is configured and matched. This is because error `404` means the non-existence of an abstract "resource", not the specific `location`.

Another frequent mistake is missing the command for phase `content`, when they actually don't expect the default static modules to come into play,for example:

```
location /auth {
    access_by_lua '
        -- a lot of Lua code omitted here...
    ';
}
```

Apparently, only commands for phase `access` are given for `/auth`, which is access_by_lua. And it has no commands for phase `content`. So when `/auth` is requested, the Lua code specified in `access` phase will execute, then the static resource will be served in phase `content` by module `ngx_static`. Since it actually looks for the file `/auth` on the disk normally it dumps a `404` error unless we are luckily and file `/auth` is created on the corresponding path. So the thumb of rule, when error `404` is encountered under no static resource circumstances, we shall first check if the `location` has properly configured its commands for phase `content`, the commands can be content_by_lua, echo and proxy_pass etc. In fact, Nginx error log `error.log` could only give very confusing message for the case. As the ones below, which is found for the above example:

```
[error] 9364#0: *1 open() "/home/agentzh/test/html/auth" failed (2: No such file or directory)
```

Nginx directive execution order (08)

So far we have addressed in detail `rewrite`, `access` and `content`, which are also the most frequently encountered phases in Nginx request processing. We have learnt many Nginx modules and their commands that execute in those phases, and it's clear to us that the commands' execution order is directly decided by the phase they are running in. Understanding the phase is our keynote for correct configuration which orchestrates various Nginx modules. Therefore let's cover the rest phases we've not met.

As mentioned in [(01)](01), altogether there can be 11 phases when Nginx handles a request. In their execution order the phases are `post-read`, `server-rewrite`, `find-config`, `rewrite`, `post-rewrite`, `preaccess`, `access`, `post-access`, `try-files`, `content`, and finally `log`.

Phase `post-read` is the very first, commands registered in this phase execute right after Nginx has processed the request headers. Similar to phase `rewrite` we've learnt earlier, `post-read` supports hooks by Nginx modules. Built-in module ngx_realip is an example, it hooks its handler in `post-read` phase, and forcefully rewrite the request's original address as the value of a specific request header. The following case illustrates ngx_realip module and its commands set_real_ip_from, real_ip_header.

```
server {
    listen 8080;

    set_real_ip_from 127.0.0.1;
    real_ip_header   X-My-IP;

    location /test {
        set $addr $remote_addr;
        echo "from: $addr";
    }
}
```

The configuration tells Nginx to forcefully rewrite the original address of every request coming from `127.0.0.1` to be the value of the request header `X-My-IP`. Meanwhile it uses the built-in variable [$remote_addr](#) to output the request's original address, so that we know if the rewrite is successful.

First we send a request to `/test` from localhost:

```
$ curl -H 'X-My-IP: 1.2.3.4' localhost:8080/test
from: 1.2.3.4
```

The test utilizes `-H` option provided by curl, the option incorporates an extra HTTP header `X-My-IP: 1.2.3.4` in the request. As we can tell, variable [$remote_addr](#) has become `1.2.3.4` in `rewrite` phase, the value comes from the request header `X-My-IP`. So when does Nginx rewrite the request's original address ? yes it's in the `post-read` phase. Since phase `rewrite` is far behind phase `post-read`, when command [set](#) reads variable [$remote_addr](#), its value has already been rewritten in `post-read` phase.

If however, the request sent from localhost to `/test` does not have a `X-My-IP` header or the header value is an invalid IP address, Nginx will not modify the original address. For example:

```
$ curl localhost:8080/test
from: 127.0.0.1

$ curl -H 'X-My-IP: abc' localhost:8080/test
from: 127.0.0.1
```

If a request is sent from another machine to `/test`, it original address won't be overwritten by Nginx either, even if it has a perfect `X-My-IP` header. It is because our previous case marks explicitly with command [set_real_ip_from](#), that the rewriting only occurs for the requests coming from `127.0.0.1`. This filtering mechanism protect Nginx from malicious requests sent by untrusted sources. As you might have expected, command [set_real_ip_from](#) can designate a IP subnet (by using CIDR notation introduced earlier in [(03)](#)). Besides, command [set_real_ip_from](#) can be used multiple times so that we can setup multiple trusted sources, below is an example:

```
set_real_ip_from 10.32.10.5;
set_real_ip_from 127.0.0.0/24;
```

You might be asking, what's the benefit module [ngx_realip](#) brings to us? Why would we rewrite a request's original address ? The answer is: when the request has come through one or more HTTP proxies, the module becomes very handy. When a request is forwarded by a proxy, its original address will become the proxy server's IP address, consequently Nginx and the services running on it will no longer have the actual source. However, we could let proxy server record the original address in a specific header (such as `X-My-IP`) and recover it in Nginx, so that its subsequent processing (and the services running on Nginx) will take the request as if it comes right from its original address and the proxies in between are transparent. For this exact purpose, module [ngx_realip](#) needs hook handlers in the first phase, the `post-read` phase, so the rewriting occurs as early as possible.

Behind `post-read` is the `server-rewrite` phase. We briefly mentioned in [(02)](#), when module [ngx_rewrite](#) and its commands are configured in `server` directive, they basically execute in `server-rewrite` phase. We have an example below:

```
server {
    listen 8080;
```

```
    location /test {
        set $b "$a, world";
        echo $b;
    }

    set $a hello;
}
```

Attention the `set $a hello` statement is put in `server` directive, so it runs in `server-rewrite` phase, which runs earlier than `rewrite` phase. Therefore statement `set $b "$a, world"` in `location` directive is executed afterwards and it obtains the correct $a value:

```
$ curl localhost:8080/test
hello, world
```

Since phase `server-rewrite` executes later than `post-read` phase, command [set](#) in `server` directive always runs later than module [ngx_realip](#), which rewrites the request's original address, example:

```
server {
    listen 8080;

    set $addr $remote_addr;

    set_real_ip_from 127.0.0.1;
    real_ip_header   X-Real-IP;

    location /test {
        echo "from: $addr";
    }
}
```

Send request to `/test` we have:

```
$ curl -H 'X-Real-IP: 1.2.3.4' localhost:8080/test
from: 1.2.3.4
```

Again, command [set](#) is written in front of commands of [ngx_realip](#), its actual execution is only afterwards. So when command [set](#) assigns variable `$addr` in `server-rewrite` phase, the variable [$remote_addr](#) has been overwritten.

Nginx directive execution order (09)

Right after `server-rewrite` is the phase `find-config`. This phase does not allow Nginx modules to register their handlers, instead it is a phase when Nginx core matches the current request to the `location` directives. It means a request is not catered by any `location` directive until it reaches `find-config`. Apparently, for phases like `post-read` and `server-rewrite`, the effective commands are those which get specified only in `server` directives and their outer directives, because the two phases are executed earlier than `find-config`. This explains that commands of module [ngx_rewrite](#) are executed in phase `server-rewrite` only if they are written within `sever` directive. Similarly, the former examples configure the commands of module [ngx_realip](#) in `server` directive to make sure the handlers registered in `post-read` phase could function correctly.

As soon as Nginx matches a `location` directive in the `find-config` phase, it prints a debug log in the error log file. Let's check following example:

```
location /hello {
    echo "hello world";
}
```

If Nginx enables the "debug log", a debug log can be captured in file `error.log` whenever interface `/hello` is requested.

```
$ grep 'using config' logs/error.log
[debug] 84579#0: *1 using configuration "/hello"
```

For the purpose of convenience, the log's time stamp has been omitted.

After phase `find-config`, it is our old buddy `rewrite`. Since Nginx already matches the request to a specific `location` directive, starting from this phase, commands written within `location` directives are becoming effective. As illustrated earlier, commands of module ngx_rewrite are executed in `rewrite` phase when they are written in `location` directives. Likewise, commands of module ngx_set_misc and module ngx_lua (set_by_lua and rewrite_by_lua) are also executed in phase `rewrite`.

After `rewrite`, it is the `post-rewrite` phase. Just like `find-config`, this phase does not allow Nginx modules to register their handlers either, instead it carries out the needed "internal redirects" by Nginx core (if this has been requested in `rewrite` phase). We have addressed the "internal jump" concept in (02), and demonstrated how to issue the "internal redirect" with command echo_exec or command rewrite. However, let's focus on command rewrite for the moment since command echo_exec is executed in `content` phase and becomes irrelevant to `post-rewrite`, the former draws greater interest because it executes in `rewrite` phase. Back to our example in (02):

```
server {
    listen 8080;

    location /foo {
        set $a hello;
        rewrite ^ /bar;
    }

    location /bar {
        echo "a = [$a]";
    }
}
```

The command rewrite found in directive `location /foo`, rewrites the URI of current request as `/bar` unconditionally, meanwhile, it issues an "internal redirect" and execution continues from `location /bar`. What ultimately intrigues us, is the magical bits and pieces of "internal redirect" mechanism, "internal redirect" effectively rewinds our processing of current request back to the `find-config` phase, so that the `location` directives can be matched again to the request URI, which usually has been rewritten. Just like our example, whose URI is rewritten as `/bar` by command rewrite, the `location /bar` directive is matched and execution repeats the `rewrite` phase thereafter.

It might not be obvious, that the actual act of rewinding to `find-config` does not occur in `rewrite` phase, instead it occurs in the following `post-rewrite` phase. Command rewrite in the former example, simply requests Nginx to issue an "internal redirect" in its `post-rewrite` phase. This design is usually questioned by Nginx beginners and they tend to come up with an idea to execute the "internal jump" directly by command rewrite. The answer however, is fairly simple. The design allows URI be rewritten multiple times in the `location` directive, which is matched at the very beginning. Such as:

```
location /foo {
    rewrite ^ /bar;
    rewrite ^ /baz;

    echo foo;
}

location /bar {
    echo bar;
}

location /baz {
    echo baz;
}
```

The request URI has been rewritten twice in `location /foo` directive: firstly it becomes `/bar`, secondly it becomes `/baz`. As the net effect of both rewrite statements, "internal redirect" occurs only once in `post-rewrite` phase. If it would have executed the "internal redirect" at the first URI rewrite, the second would have no chance to be executed since processing would have left current `location` directive. To prove this we send a request to `/foo`:

```
$ curl localhost:8080/foo
baz
```

It can be asserted from the output, the actual jump is from `/foo` to `/baz`. We could further prove this by enabling Nginx "debug log" and interrogate the debug log generated in `find-config` phase for the matched:

```
$ grep 'using config' logs/error.log
[debug] 89449#0: *1 using configuration "/foo"
[debug] 89449#0: *1 using configuration "/baz"
```

Clearly, for the specific request, Nginx only matches two `location` directives: `/foo` and `/baz`, and "internal jump" occurs only once.

Quite obviously, if command `ngx_rewrite/rewrite` is used to rewrite the request URI in `server` directive, there won't be any "internal redirects", this is because the URI rewrite is happening in `server-rewrite` phase, which gets executed earlier than `find-config` phase that matches in between the `location` directives. We can check the example below:

```
server {
    listen 8080;

    rewrite ^/foo /bar;

    location /foo {
        echo foo;
    }

    location /bar {
        echo bar;
    }
}
```

In the example, every request whose URI starts with `/foo` gets its URI rewritten as `/bar`. The rewriting occurs in `server-rewrite` phase, and the request has never been matched to any `location` directive. Only afterwards Nginx executes the matches in `find-config` phase. So if we send a request to `/foo`, `location /foo` never gets matched because when the match occurs in `find-config` phase, the request URI has been rewritten as `/bar`. So `location /bar` is the one and the only one matched directive. Actual output illustrates this:

```
$ curl localhost:8080/foo
bar
```

Again let's check Nginx "debug log":

```
$ grep 'using config' logs/error.log
[debug] 92693#0: *1 using configuration "/bar"
```

As we can tell, Nginx altogether finishes once the `location` match, and there is no "internal redirect".

Nginx directive execution order (10)

After `post-rewrite`, it is the `preaccess` phase. Just as its name implies, the phase is called `preaccess` simply because it is executed right before `access` phase.

Built-in module [ngx_limit_req](#) and [ngx_limit_zone](#) are executed in this phase. The former limits the number of requests per hour/minute, and the latter limits the number of simultaneous requests. We will be discussing them more thoroughly afterwards.

Actually, built-in module [ngx_realip](#) registers its handler in `preaccess` as well. You might need to ask then: "why do it again? Did it register its handlers in `post-read` phase already". Before the answer is uncovered let's study following example:

```
server {
    listen 8080;

    location /test {
        set_real_ip_from 127.0.0.1;
        real_ip_header X-Real-IP;
```

```
        echo "from: $remote_addr";
    }
}
```

Comparing to the earlier example, the major difference is that commands of module ngx_realip are written in a specific `location` directive. As we have learnt before, Nginx matches its `location` directives in `find-config` phase, which is far behind `post-read`, hence the request has nothing to do with commands written in any `location` directive in `post-read` phase. Back to our example, it is exactly the case where commands are written in a `location` directive and module ngx_realip won't carry out any rewrite of the remote address, because it is not instructed as such in `post-read` phase.

What if we do need the rewrite? To help resolve the issue, module ngx_realip registers its handlers in `preaccess` again, so that it is given the chance to execute in a `location` directive. Now the example runs as we would've expected:

```
$ curl -H 'X-Real-IP: 1.2.3.4' localhost:8080/test
from: 1.2.3.4
```

Be really careful though, module ngx_realip could easily be misused, as our following example illustrates:

```
server {
    listen 8080;

    location /test {
        set_real_ip_from 127.0.0.1;
        real_ip_header X-Real-IP;

        set $addr $remote_addr;
        echo "from: $addr";
    }
}
```

In the example, we introduces a variable `$addr`, to which the value of $remote_addr is saved in `rewrite` phase. The variable is then used in the output. Slow down right here and you might have noticed the issue, phase `rewrite` occurs earlier than `preaccess`, so variable assignment actually happens before module ngx_realip has the chance to rewrite the remote address in `preaccess` phase. The output proves our observation:

```
$ curl -H 'X-Real-IP: 1.2.3.4' localhost:8080/test
from: 127.0.0.1
```

The output gives the actual remote address (not the rewritten one) Again Nginx "debug log" helps assert it too:

```
$ grep -E 'http script (var|set)|realip' logs/error.log
[debug] 32488#0: *1 http script var: "127.0.0.1"
[debug] 32488#0: *1 http script set $addr
[debug] 32488#0: *1 realip: "1.2.3.4"
[debug] 32488#0: *1 realip: 0100007F FFFFFFFF 0100007F
[debug] 32488#0: *1 http script var: "127.0.0.1"
```

Among the logs, the first line writes:

```
[debug] 32488#0: *1 http script var: "127.0.0.1"
```

The log is generated when variable $remote_addr is fetched by command set, string `"127.0.0.1"` is the fetched value.

The second line writes:

```
[debug] 32488#0: *1 http script set $addr
```

It indicates Nginx assigns value to variable `$addr`.

For the following two lines:

```
[debug] 32488#0: *1 realip: "1.2.3.4"
[debug] 32488#0: *1 realip: 0100007F FFFFFFFF 0100007F
```

They are generated when module ngx_realip rewrites the remote address in `preaccess` phase. As we can tell, the new address becomes `1.2.3.4` as expected but it happens only after the variable assignment and that's already too late.

Now the last line:

```
[debug] 32488#0: *1 http script var: "127.0.0.1"
```

It is generated when command echo outputs variable `$addr`, clearly the value is the original remote address, not the rewritten one.

Some people might come up with a solution immediately:" what if module ngx_realip registers its handlers in `rewrite` phase instead, not in `preacccess` phase ?" The solution however is, not necessarily correct. This is because module ngx_rewrite registers its handlers in `rewrite` phase too, and we have learnt in (02) that the execution order, under the circumstances, can not be guaranteed, so there is a good chance that module ngx_realip still executes its commands after command set.

Always we have the backup option: instead of `preaccess`, try use ngx_realip module in `server` directive, it bypasses the bothersome situations encountered above.

After phase `preaccess`, it is another old friend, the `access` phase. As we've learnt, built-in module ngx_access, 3rd party module ngx_auth_request and 3rd party module ngx_lua (access_by_lua) have their commands executed in this phase.

After phase `access`, it is the `post-access` phase. Again as the name implies, we can easily spot that the phase is executed right after `access` phase. Similar to `post-rewrite`, the phase does not allow Nginx module to register their handlers, instead it runs a few tasks by Nginx core, among them, primarily is the satisfy functionality, provided by module ngx_http_core.

When multiple Nginx module execute their commands in `access` phase, command satisfy controls their relationships in between. For example, both module A and module B register their access control handlers in `access` phase, we may have two working modes, one is to let access when both A and B pass their control, the other is to let access when either A or B pass their control. The first one is called `all` mode ("AND" relation), the second one is called `any` mode ("OR" relation) By default, Nginx uses `all` mode, below is an example:

```
location /test {
    satisfy all;

    deny all;
    access_by_lua 'ngx.exit(ngx.OK)';

    echo something important;
}
```

Under `/test` directive, both ngx_access and ngx_lua are used, so we have two modules monitoring access in `access` phase. Specifically, statement `deny all` tells module ngx_access to rejects all access, whereas statement `access_by_lua 'ngx.exit(ngx.OK)'` allows all access. When `all` mode is used with command satisfy, it means to let access only if every module allows access. Since module ngx_access always rejects in our case, the request is rejected:

```
$ curl localhost:8080/test
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Careful readers might find following error log in the Nginx error log file:

```
[error] 6549#0: *1 access forbidden by rule
```

If however, we change the `satisfy all` statement to `satisfy any` .

```
location /test {
    satisfy any;

    deny all;
    access_by_lua 'ngx.exit(ngx.OK)';

    echo something important;
}
```

The outcome is completely different:

```
$ curl localhost:8080/test
something important
```

The request is allowed to access. Because overall access is allowed whenever one module passes the control in `any` mode. In our example, module [ngx_lua](#) and its command [access_by_lua](#) always allow the access.

Certainly, if every module rejects the access in the `satisfy any` circumstances, the request will be rejected:

```
location /test {
    satisfy any;

    deny all;
    access_by_lua 'ngx.exit(ngx.HTTP_FORBIDDEN)';

    echo something important;
}
```

Now request to `/test` will encounter `403 Forbidden` error page. In the process, the "OR" relation of access control of each `access` module, is implemented in `post-access` .

Please note that this example requires at least [ngx_lua](#) 0.5.0rc19 or later; earlier versions cannot work with the `satisfy any` statement.