agentzh 的 Nginx 教程（版本 2015.03.19）

目录

缘起

其实这两年为 Nginx 世界做了这么多的事情，一直想通过一系列教程性的文章把我的那些工作成果和所学所知都介绍给更多的朋友。现在终于下决心在新浪博客 http://blog.sina.com.cn/openresty 上面用中文写点东西，每一篇东西都会有一个小主题，但次序和组织上就不那么讲究了，毕竟并不是一本完整的图书，或许未来我会将之整理出书也不一定。

我现在编写的教程是按所谓的“系列”来划分的，比如首先连载的“Nginx 变量漫谈”系列。每一个系列基本上都可以粗略对应到未来出的 Nginx 书中的一“章”（当然内部还会重新组织内容并划分出“节”来）。我面向的读者是各个水平层次的 Nginx 用户，同时也包括未使用过 Nginx 的 Apache、Lighttpd 等服务器的老用户。

我只保证这些教程中的例子至少兼容到 Nginx `0.8.54`，别用更老的版本来找我的错处，我一概不管，毕竟眼下最新的稳定版已经是 `1.0.10` 了。

凡在教程里面提到的模块，都是经过生产环境检验过的。即便是标准模块，如果没有达到生产标准，或者有重要的 bug，我也不会提及。

我在教程中会大量使用非标准的第三方模块，如果你怕麻烦，不愿自己一个一个从网上下载和安装那些个模块，我推荐你下载和安装我维护的 `ngx_openresty` 这个软件包：

http://openresty.org/

教程里提及的模块，包括足够新的 Nginx 稳定版核心，都包含在了这个软件包中。

我在这些教程中遵循的一个原则是，尽量通过短小精悍的实例来佐证我陈述的原理和观点。我希望帮助读者养成不随便听信别人现成的观点和陈述，而通过自己运行实例来验证的好习惯。这种风格或许也和我在 QA 方面的背景有关。事实上我在写作过程中也经常按我设计的小例子的实际运行结果，不断地对我的理解以及教程的内容进行修正。

对于有问题的代码示例，我们会有意在排版上让它们和其他合法示例所有区别，即在问题示例的每一行代码前添加问号字符，即（`?`），一个例子是：

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

未经我的同意，请不要随便转载或者以其他方式使用这些教程。因为其中的每一句话，除了特别引用的“名句”，都是我自己的，我保留所有的权利。我不希望读者转载的另一大原因在于：转载后的拷贝版本是死的，我就不能再同步更新了。而我经常会按照读者的反馈，对已发表的老文章进行了大面积的修订。

我欢迎读者多提宝贵意见，特别是建设性的批评意见。类似“太烂了！”这样无聊中伤的意见我看还是算了。

所有这些文章的源都已经放在 GitHub 网站上进行版本控制了：

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

源文件都在此项目的 `zh-cn/` 目录下。我使用了一种自己设计的 `Wiki` 和 `POD` 标记语言的混合物来撰写这些文章，就是那些 `.tut` 文件。欢迎建立分支和提供补丁。

本教程适用于普通手机、Kindle、iPad/iPhone、Sony 等电子阅读器的 `.html`、`.mobi`、`.epub` 以及 `.pdf` 等格式的电子书文件可以从下面这个位置下载：

http://openresty.org/#eBooks

章亦春 (agentzh) 于福州家中

2011 年 11 月 30 日

Nginx 教程的连载计划

下面以教程系列为单位，列举出了已经发表和计划发表的连载教程：

- Nginx 新手起步
- Nginx 是如何匹配 URI 的
- Nginx 变量漫谈
- Nginx 配置指令的执行顺序
- Nginx 的 if 是邪恶的
- Nginx 子请求
- Nginx 静态文件服务
- Nginx 的日志服务
- 基于 Nginx 的应用网关
- 基于 Nginx 的反向代理
- Nginx 与 Memcached
- Nginx 与 Redis
- Nginx 与 MySQL
- Nginx 与 PostgreSQL
- 基于 Nginx 的应用缓存
- Nginx 中的安全与访问控制
- 基于 Nginx 的 Web Service
- Nginx 驱动的 Web 2.0 应用
- 测试 Nginx 及其应用的性能
- 借助 Nginx 社区的力量

这些系列的名字和最终我的 Nginx 书中的“章”名可以粗略地对应上，但不会等同。同时未发表的系列的名字也可能发生变化，同时实际发表的顺序也会和这里列出的顺序不太一样。

上面的列表会随时更新，以保证总是反映了最新的计划和发表情况。

Nginx 变量漫谈（一）

Nginx 的配置文件使用的就是一门微型的编程语言，许多真实世界里的 Nginx 配置文件其实就是一个一个的小程序。当然，是不是“图灵完全的”暂且不论，至少据我观察，它在设计上受 Perl 和 Bourne Shell 这两种语言的影响很大。在这一点上，相比 Apache 和 Lighttpd 等其他 Web 服务器的配置记法，不能不说算是 Nginx 的一大特色了。既然是编程语言，一般也就少不了“变量”这种东西（当然，Haskell 这样奇怪的函数式语言除外了）。

熟悉 Perl、Bourne Shell、C/C++ 等命令式编程语言的朋友肯定知道，变量说白了就是存放“值”的容器。而所谓“值”，在许多编程语言里，既可以是 `3.14` 这样的数值，也可以是 `hello world` 这样的字符串，甚至可以是像数组、哈希表这样的复杂数据结构。然而，在 Nginx 配置中，变量只能存放一种类型的值，因为也只存在一种类型的值，那就是字符串。

比如我们的 `nginx.conf` 文件中有下面这一行配置：

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

我们使用了标准 ngx_rewrite 模块的 set 配置指令对变量 `$a` 进行了赋值操作。特别地，我们把字符串 `hello world` 赋给了它。

我们看到，Nginx 变量名前面有一个 `$` 符号，这是记法上的要求。所有的 Nginx 变量在 Nginx 配置文件中引用时都须带上 `$` 前缀。这种表示方法和 Perl、PHP 这些语言是相似的。

虽然 `$` 这样的变量前缀修饰会让正统的 `Java` 和 `C#` 程序员不舒服，但这种表示方法的好处也是显而易见的，那就是可以直接把变量嵌入到字符串常量中以构造出新的字符串：

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

这里我们通过已有的 Nginx 变量 `$a` 的值，来构造变量 `$b` 的值，于是这两条指令顺序执行完之后，`$a` 的值是 `hello`，而 `$b` 的值则是 `hello, hello`. 这种技术在 Perl 世界里被称为“变量插值”（variable interpolation），它让专门的字符串拼接运算符变得不再那么必要。我们在这里也不妨采用此术语。

我们来看一个比较完整的配置示例：

```
server {
    listen 8080;

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

这个例子省略了 `nginx.conf` 配置文件中最外围的 `http` 配置块以及 `events` 配置块。使用 `curl` 这个 HTTP 客户端在命令行上请求这个 `/test` 接口，我们可以得到

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

这里我们使用第三方 ngx_echo 模块的 echo 配置指令将 `$foo` 变量的值作为当前请求的响应体输出。

我们看到， echo 配置指令的参数也支持“变量插值”。不过，需要说明的是，并非所有的配置指令都支持“变量插值”。事实上，指令参数是否允许“变量插值”，取决于该指令的实现模块。

如果我们想通过 echo 指令直接输出含有“美元符”（$）的字符串，那么有没有办法把特殊的 $ 字符给转义掉呢？答案是否定的（至少到目前最新的 Nginx 稳定版 `1.0.10`）。不过幸运的是，我们可以绕过这个限制，比如通过不支持“变量插值”的模块配置指令专门构造出取值为 $ 的 Nginx 变量，然后再在 echo 中使用这个变量。看下面这个例子：

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

server {
    listen 8080;

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

测试结果如下：

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

这里用到了标准模块 ngx_geo 提供的配置指令 geo 来为变量 $dollar 赋予字符串 "$"，这样我们在下面需要使用美元符的地方，就直接引用我们的 $dollar 变量就可以了。其实 ngx_geo 模块最常规的用法是根据客户端的 IP 地址对指定的 Nginx 变量进行赋值，这里只是借用它以便“无条件地”对我们的 $dollar 变量赋予“美元符”这个值。

在“变量插值”的上下文中，还有一种特殊情况，即当引用的变量名之后紧跟着变量名的构成字符时（比如后跟字母、数字以及下划线），我们就需要使用特别的记法来消除歧义，例如：

```
server {
    listen 8080;

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

这里，我们在 echo 配置指令的参数值中引用变量 $first 的时候，后面紧跟着 world 这个单词，所以如果直接写作 "$firstworld" 则 Nginx “变量插值”计算引擎会将之识别为引用了变量 $firstworld. 为了解决这个难题，Nginx 的字符串记法支持使用花括号在 $ 之后把变量名围起来，比如这里的 ${first}. 上面这个例子的输出是：

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

set 指令（以及前面提到的 geo 指令）不仅有赋值的功能，它还有创建 Nginx 变量的副作用，即当作为赋值对象的变量尚不存在时，它会自动创建该变量。比如在上面这个例子中，如果 $a 这个变量尚未创建，则 set 指令会自动创建 $a 这个用户变量。如果我们不创建就直接使用它的值，则会报错。例如

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

此时 Nginx 服务器会拒绝加载配置:

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

是的，我们甚至都无法启动服务！

有趣的是，Nginx 变量的创建和赋值操作发生在全然不同的时间阶段。Nginx 变量的创建只能发生在 Nginx 配置加载的时候，或者说 Nginx 启动的时候；而赋值操作则只会发生在请求实际处理的时候。这意味着不创建而直接使用变量会导致启动失败，同时也意味着我们无法在请求处理时动态地创建新的 Nginx 变量。

Nginx 变量一旦创建，其变量名的可见范围就是整个 Nginx 配置，甚至可以跨越不同虚拟主机的 server 配置块。我们来看一个例子：

```
server {
    listen 8080;

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

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

这里我们在 location /bar 中用 set 指令创建了变量 $foo，于是在整个配置文件中这个变量都是可见的，因此我们可以在 location /foo 中直接引用这个变量而不用担心 Nginx 会报错。

下面是在命令行上用 curl 工具访问这两个接口的结果：

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

```
foo = []

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

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

从这个例子我们可以看到，set 指令因为是在 location /bar 中使用的，所以赋值操作只会在访问 /bar 的请求中执行。而请求 /foo 接口时，我们总是得到空的 $foo 值，因为用户变量未赋值就输出的话，得到的便是空字符串。

从这个例子我们可以窥见的另一个重要特性是，Nginx 变量名的可见范围虽然是整个配置，但每个请求都有所有变量的独立副本，或者说都有各变量用来存放值的容器的独立副本，彼此互不干扰。比如前面我们请求了 /bar 接口后，$foo 变量被赋予了值 32，但它丝毫不会影响后续对 /foo 接口的请求所对应的 $foo 值（它仍然是空的！），因为各个请求都有自己独立的 $foo 变量的副本。

对于 Nginx 新手来说，最常见的错误之一，就是将 Nginx 变量理解成某种在请求之间全局共享的东西，或者说“全局变量”。而事实上，Nginx 变量的生命期是不可能跨越请求边界的。

Nginx 变量漫谈（二）

关于 Nginx 变量的另一个常见误区是认为变量容器的生命期，是与 location 配置块绑定的。其实不然。我们来看一个涉及“内部跳转”的例子：

```
server {
    listen 8080;

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

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

这里我们在 location /foo 中，使用第三方模块 ngx_echo 提供的 echo_exec 配置指令，发起到 location /bar 的“内部跳转”。所谓“内部跳转”，就是在处理请求的过程中，于服务器内部，从一个 location 跳转到另一个 location 的过程。这不同于利用 HTTP 状态码 301 和 302 所进行的“外部跳转”，因为后者是由 HTTP 客户端配合进行跳转的，而且在客户端，用户可以通过浏览器地址栏这样的界面，看到请求的 URL 地址发生了变化。内部跳转和 Bourne Shell（或 Bash）中的 exec 命令很像，都是“有去无回”。另一个相近的例子是 C 语言中的 goto 语句。

既然是内部跳转，当前正在处理的请求就还是原来那个，只是当前的 location 发生了变化，所以还是原来的那一套 Nginx 变量的容器副本。对应到上例，如果我们请求的是 /foo 这个接口，那么整个工作流程是这样的：先在 location /foo 中通过 set 指令将 $a 变量的值赋为字符串 hello，然后通过 echo_exec 指令发起内部跳转，又进入到 location /bar 中，再输出 $a 变量的值。因为 $a 还是原来的 $a，所以我们可以期望得到 hello 这行输出。测试证实了这一点：

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

但如果我们从客户端直接访问 /bar 接口，就会得到空的 $a 变量的值，因为它依赖于 location /foo 来对 $a 进行初始化。

从上面这个例子我们看到，一个请求在其处理过程中，即使经历多个不同的 location 配置块，它使用的还是同一套 Nginx 变量的副本。这里，我们也首次涉及到了“内部跳转”这个概念。值得一提的是，标准 ngx_rewrite 模块的 rewrite 配置指令其实也可以发起“内部跳转”，例如上面那个例子用 rewrite 配置指令可以改写成下面这样的形式：

```
server {
    listen 8080;

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

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

其效果和使用 echo_exec 是完全相同的。后面我们还会专门介绍这个 rewrite 指令的更多用法，比如发起 301 和 302 这样的“外部跳转”。

从上面这个例子我们看到，Nginx 变量值容器的生命期是与当前正在处理的请求绑定的，而与 location 无关。

前面我们接触到的都是通过 set 指令隐式创建的 Nginx 变量。这些变量我们一般称为“用户自定义变量”，或者更简单一些，“用户变量”。既然有“用户自定义变量”，自然也就有由 Nginx 核心和各个 Nginx 模块提供的“预定义变量”，或者说“内建变量”（builtin variables）。

Nginx 内建变量最常见的用途就是获取关于请求或响应的各种信息。例如由 ngx_http_core 模块提供的内建变量 $uri，可以用来获取当前请求的 URI（经过解码，并且不含请求参数），而 $request_uri 则用来获取请求最原始的 URI（未经解码，并且包含请求参数）。请看下面这个例子：

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

这里为了简单起见，连 server 配置块也省略了，和前面所有示例一样，我们监听的依然是 8080 端口。在这个例子里，我们把 $uri 和 $request_uri 的值输出到响应体中去。下面我们用不同的请求来测试一下这个 /test 接口：

```
$ 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
```

另一个特别常用的内建变量其实并不是单独一个变量，而是有无限多变种的一群变量，即名字以 arg_ 开头的所有变量，我们估且称之为 $arg_XXX 变量群。一个例子是 $arg_name，这个变量的值是当前请求名为 name 的 URI 参数的值，而且还是未解码的原始形式的值。我们来看一个比较完整的示例：

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

然后在命令行上使用各种参数组合去请求这个 /test 接口：

```
$ 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
```

其实 $arg_name 不仅可以匹配 name 参数，也可以匹配 NAME 参数，抑或是 Name，等等：

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

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

Nginx 会在匹配参数名之前，自动把原始请求中的参数名调整为全部小写的形式。

如果你想对 URI 参数值中的 %XX 这样的编码序列进行解码，可以使用第三方 ngx_set_misc 模块提供的 set_unescape_uri 配置指令：

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

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

现在我们再看一下效果：

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

空格果然被解码出来了！

从这个例子我们同时可以看到，这个 set_unescape_uri 指令也像 set 指令那样，拥有自动创建 Nginx 变量的功能。后面我们还会专门介绍到 ngx_set_misc 模块。

像 $arg_XXX 这种类型的变量拥有无穷无尽种可能的名字，所以它们并不对应任何存放值的容器。而且这种变量在 Nginx 核心中是经过特别处理的，第三方 Nginx 模块是不能提供这样充满魔法的内建变量的。

类似 $arg_XXX 的内建变量还有不少，比如用来取 cookie 值的 $cookie_XXX 变量群，用来取请求头的 $http_XXX 变量群，以及用来取响应头的 $sent_http_XXX 变量群。这里就不一一介绍了，感兴趣的读者可以参考 ngx_http_core 模块的官方文档。

需要指出的是，许多内建变量都是只读的，比如我们刚才介绍的 $uri 和 $request_uri. 对只读变量进行赋值是应当绝对避免的，因为会有意想不到的后果，比如：

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

```
    ? }
```

这个有问题的配置会让 Nginx 在启动的时候报出一条令人匪夷所思的错误：

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

如果你尝试改写另外一些只读的内建变量，比如 $arg_XXX 变量，在某些 Nginx 的版本中甚至可能导致进程崩溃。

Nginx 变量漫谈（三）

也有一些内建变量是支持改写的，其中一个例子是 $args. 这个变量在读取时返回当前请求的 URL 参数串（即请求 URL 中问号后面的部分，如果有的话），而在赋值时可以直接修改参数串。我们来看一个例子：

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

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

这里我们把原始的 URL 参数串先保存在 $orig_args 变量中，然后通过改写 $args 变量来修改当前的 URL 参数串，最后我们用 echo 指令分别输出 $orig_args 和 $args 变量的值。接下来我们这样来测试这个 /test 接口：

```
$ 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
```

在第一次测试中，我们没有设置任何 URL 参数串，所以输出 $orig_args 变量的值时便得到空。而在第一次和第二次测试中，无论我们是否提供 URL 参数串，参数串都会在 location /test 中被强行改写成 a=3&b=4.

需要特别指出的是，这里的 $args 变量和 $arg_XXX 一样，也不再使用属于自己的存放值的容器。当我们读取 $args 时，Nginx 会执行一小段代码，从 Nginx 核心中专门存放当前 URL 参数串的位置去读取数据；而当我们改写 $args 时，Nginx 会执行另一小段代码，对相同位置进行改写。Nginx 的其他部分在需要当前 URL 参数串的时候，都会从那个位置去读数据，所以我们对 $args 的修改会影响到所有部分的功能。我们来看一个例子：

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

这里我们先把内建变量 $arg_a 的值，即原始请求的 URL 参数 a 的值，保存在用户变量 $orig_a 中，然后通过对内建变量 $args 进行赋值，把当前请求的参数串改写为 a=5 ，最后再用 echo 指令分别输出 $orig_a 和 $arg_a 变量的值。因为对内建变量 $args 的修改会直接导致当前请求的 URL 参数串发生变化，因此内建变量 $arg_XXX 自然也会随之变化。测试的结果证实了这一点：

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

我们看到，因为原始请求的 URL 参数串是 a=3, 所以 $arg_a 最初的值为 3, 但随后通过改写 $args 变量，将 URL 参数串又强行修改为 a=5, 所以最终 $arg_a 的值又自动变为了 5.

我们再来看一个通过修改 $args 变量影响标准的 HTTP 代理模块 ngx_proxy 的例子：

```
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";
    }
}
```

这里我们在 http 配置块中定义了两个虚拟主机。第一个虚拟主机监听 8080 端口，其 /test 接口自己通过改写 $args 变量，将当前请求的 URL 参数串无条件地修改为 foo=1&bar=2. 然后 /test 接口再通过 ngx_proxy 模块的 proxy_pass 指令配置了一个反向代理，指向本机的 8081 端口上的 HTTP 服务 /args. 默认情况下， ngx_proxy 模块在转发 HTTP 请求到远方 HTTP 服务的时候，会自动把当前请求的 URL 参数串也转发到远方。

而本机的 8081 端口上的 HTTP 服务正是由我们定义的第二个虚拟主机来提供的。我们在第二个虚拟主机的 location /args 中利用 echo 指令输出当前请求的 URL 参数串，以检查 /test 接口通过 ngx_proxy 模块实际转发过来的 URL 请求参数串。

我们来实际访问一下第一个虚拟主机的 /test 接口：

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

我们看到，虽然请求自己提供了 URL 参数串 blah=7，但在 location /test 中，参数串被强行改写成了 foo=1&bar=2. 接着经由 proxy_pass 指令将我们被改写掉的参数串转发给了第二个虚拟主机上配置的 /args 接口，然后再把 /args 接口的 URL 参数串输出。事实证明，我们对 $args 变量的赋值操作，也成功影响到了 ngx_proxy 模块的行为。

在读取变量时执行的这段特殊代码，在 Nginx 中被称为“取处理程序”（get handler）；而改写变量时执行的这段特殊代码，则被称为“存处理程序”（set handler）。不同的 Nginx 模块一般会为它们的变量准备不同的“存取处理程序”，从而让这些变量的行为充满魔法。

其实这种技巧在计算世界并不鲜见。比如在面向对象编程中，类的设计者一般不会把类的成员变量直接暴露给类的用户，而是另行提供两个方法（method），分别用于该成员变量的读操作和写操作，这两个方法常常被称为“存取器”（accessor）。下面是 C++ 语言中的一个例子：

```
#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;
};
```

在这个名叫 Person 的 C++ 类中，我们提供了 get_name 和 set_name 这两个公共方法，以作为私有成员变量 m_name 的“存取器”。

这样设计的好处是显而易见的。类的设计者可以在“存取器”中执行任意代码，以实现所需的业务逻辑以及“副作用”，比如自动更新与当前成员变量存在依赖关系的其他成员变量，抑或是直接修改某个与当前对象相关联的数据库表中的对应字段。而对于后一种情况，也许“存取器”所对应的成员变量压根就不存在，或者即使存在，也顶多扮演着数据缓存的角色，以缓解被代理数据库的访问压力。

与面向对象编程中的“存取器”概念相对应，Nginx 变量也是支持绑定“存取处理程序”的。Nginx 模块在创建变量时，可以选择是否为变量分配存放值的容器，以及是否自己提供与读写操作相对应的“存取处理程序”。

不是所有的 Nginx 变量都拥有存放值的容器。拥有值容器的变量在 Nginx 核心中被称为“被索引的”（indexed）；反之，则被称为“未索引的”（non-indexed）。

我们前面在 （二） 中已经知道，像 $arg_XXX 这样具有无数变种的变量群，是“未索引的”。当读取这样的变量时，其实是它的“取处理程序”在起作用，即实时扫描当前请求的 URL 参数串，提取出变量名所指定的 URL 参数的值。很多新手都会对 $arg_XXX 的实现方式产生误解，以为 Nginx 会事先解析好当前请求的所有 URL 参数，并且把相关的 $arg_XXX 变量的值都事先设置好。然而事实并非如此，Nginx 根本不会事先就解析好 URL 参数串，而是在用户读取某个 $arg_XXX 变量时，调用其“取处理程序”，即时去扫描 URL 参数串。类似地，内建变量 $cookie_XXX 也是通过它的“取处理程序”，即时去扫描 Cookie 请求头中的相关定义的。

Nginx 变量漫谈（四）

在设置了“取处理程序”的情况下，Nginx 变量也可以选择将其值容器用作缓存，这样在多次读取变量的时候，就只需要调用“取处理程序”计算一次。我们下面就来看一个这样的例子：

```
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";
    }
}
```

这里首次用到了标准 ngx_map 模块的 map 配置指令，我们有必要在此介绍一下。map 在英文中除了“地图”之外，也有“映射”的意思。比方说，中学数学里讲的“函数”就是一种“映射”。而 Nginx 的这个 map 指令就可以用于定义两个 Nginx 变量之间的映射关系，或者说是函数关系。回到上面这个例子，我们用 map 指令定义了用户变量 $foo 与 $args 内建变量之间的映射关系。特别地，用数学上的函数记法 y = f(x) 来说，我们的 $args 就是“自变量” x，而 $foo 则是“因变量” y，即 $foo 的值是由 $args 的值来决定的，或者按照书写顺序可以说，我们将 $args 变量的值映射到了 $foo 变量上。

现在我们再来看 map 指令定义的映射规则：

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

花括号中第一行的 default 是一个特殊的匹配条件，即当其他条件都不匹配的时候，这个条件才匹配。当这个默认条件匹配时，就把“因变量” $foo 映射到值 0. 而花括号中第二行的意思是说，如果“自变量” $args 精确匹配了 debug 这个字符串，则把“因变量” $foo 映射到值 1. 将这两行合起来，我们就得到如下完整的映射规则：当 $args 的值等于 debug 的时候，$foo 变量的值就是 1，否则 $foo 的值就为 0.

明白了 map 指令的含义，再来看 location /test. 在那里，我们先把当前 $foo 变量的值保存在另一个用户变量 $orig_foo 中，然后再强行把 $args 的值改写为 debug，最后我们再用 echo 指令分别输出 $orig_foo 和 $foo 的值。

从逻辑上看，似乎当我们强行改写 $args 的值为 debug 之后，根据先前的 map 映射规则，$foo 变量此时的值应当自动调整为字符串 1, 而不论 $foo 原先的值是怎样的。然而测试结果并非如此：

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

第一行输出指示 $orig_foo 的值为 0，这正是我们期望的：上面这个请求并没有提供 URL 参数串，于是 $args 最初的取值就是空，再根据我们先前定义的映射规则，$foo 变量在第一次被读取时的值就应当是 0（即匹配默认的那个 default 条件）。

而第二行输出显示，在强行改写 $args 变量的值为字符串 debug 之后，$foo 的条件仍然是 0 ，这显然不符合映射规则，因为当 $args 为 debug 时，$foo 的值应当是 1. 这究竟是为什么呢？

其实原因很简单，那就是 $foo 变量在第一次读取时，根据映射规则计算出的值被缓存住了。刚才我们说过，Nginx 模块可以为其创建的变量选择使用值容器，作为其“取处理程序”计算结果的缓存。显然， ngx_map 模块认为变量间的映射计算足够昂贵，需要自动将因变量的计算结果缓存下来，这样在当前请求的处理过程中如果再次读取这个因变量，Nginx 就可以直接返回缓存住的结果，而不再调用该变量的“取处理程序”再行计算了。

为了进一步验证这一点，我们不妨在请求中直接指定 URL 参数串为 debug:

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

我们看到，现在 $orig_foo 的值就成了 1，因为变量 $foo 在第一次被读取时，自变量 $args 的值就是 debug，于是按照映射规则，“取处理程序”计算返回的值便是 1. 而后续再读取 $foo 的值时，就总是得到被缓存住的 1 这个结果，而不论 $args 后来变成什么样了。

map 指令其实是一个比较特殊的例子，因为它可以为用户变量注册“取处理程序”，而且用户可以自己定义这个“取处理程序”的计算规则。当然，此规则在这里被限定为与另一个变量的映射关系。同时，也并非所有使用了“取处理程序”的变量都会缓存结果，例如我们前面在 （三） 中已经看到 $arg_XXX 并不会使用值容器进行缓存。

类似 ngx_map 模块，标准的 ngx_geo 等模块也一样使用了变量值的缓存机制。

在上面的例子中，我们还应当注意到 map 指令是在 server 配置块之外，也就是在最外围的 http 配置块中定义的。很多读者可能会对此感到奇怪，毕竟我们只是在 location /test 中用到了它。这倒不是因为我们不想把 map 语句直接挪到 location 配置块中，而是因为 map 指令只能在 http 块中使用！

很多 Nginx 新手都会担心如此“全局”范围的 map 设置会让访问所有虚拟主机的所有 location 接口的请求都执行一遍变量值的映射计算，然而事实并非如此。前面我们已经了解到 map 配置指令的工作原理是为用户变量注册 “取处理程序”，并且实际的映射计算是在“取处理程序”中完成的，而“取处理程序”只有在该用户变量被实际读取时才会执行（当然，因为缓存的存在，只在请求生命期中的第一次读取中才被执行），所以对于那些根本没有用到相关变量的请求来说，就根本不会执行任何的无用计算。

这种只在实际使用对象时才计算对象值的技术，在计算领域被称为“惰性求值”（lazy evaluation）。提供“惰性求值” 语义的编程语言并不多见，最经典的例子便是 Haskell. 与之相对的便是“主动求值” （eager evaluation）。我们有幸在 Nginx 中也看到了“惰性求值”的例子，但“主动求值”语义其实在 Nginx 里面更为常见，例如下面这行再普通不过的 set 语句：

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

这里会在执行 set 规定的赋值操作时，“主动”地计算出变量 $b 的值，而不会将该求值计算延缓到变量 $b 实际被读取的时候。

Nginx 变量漫谈（五）

前面在 （二） 中我们已经了解到变量值容器的生命期是与请求绑定的，但是我当时有意避开了“请求”的正式定义。大家应当一直默认这里的“请求”都是指客户端发起的 HTTP 请求。其实在 Nginx 世界里有两种类型的“请求”，一种叫做“主请求”（main request），而另一种则叫做“子请求”（subrequest）。我们先来介绍一下它们。

所谓“主请求”，就是由 HTTP 客户端从 Nginx 外部发起的请求。我们前面见到的所有例子都只涉及到“主请求”，包括 （二） 中那两个使用 echo_exec 和 rewrite 指令发起“内部跳转”的例子。

而“子请求”则是由 Nginx 正在处理的请求在 Nginx 内部发起的一种级联请求。“子请求”在外观上很像 HTTP 请求，但实现上却和 HTTP 协议乃至网络通信一点儿关系都没有。它是 Nginx 内部的一种抽象调用，目的是为了方便用户把“主请求”的任务分解为多个较小粒度的“内部请求”，并发或串行地访问多个 location 接口，然后由这些 location 接口通力协作，共同完成整个“主请求”。当然，“子请求”的概念是相对的，任何一个“子请求”也可以再发起更多的“子子请求”，甚至可以玩递归调用（即自己调用自己）。当一个请求发起一个“子请求”的时候，按照 Nginx 的术语，习惯把前者称为后者的“父请求”（parent request）。值得一提的是，Apache 服务器中其实也有“子请求”的概念，所以来自 Apache 世界的读者对此应当不会感到陌生。

下面就来看一个使用了“子请求”的例子：

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

location /foo {
    echo foo;
}
```

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

这里在 location /main 中，通过第三方 ngx_echo 模块的 echo_location 指令分别发起到 /foo 和 /bar 这两个接口的 GET 类型的“子请求”。由 echo_location 发起的“子请求”，其执行是按照配置书写的顺序串行处理的，即只有当 /foo 请求处理完毕之后，才会接着处理 /bar 请求。这两个“子请求”的输出会按执行顺序拼接起来，作为 /main 接口的最终输出：

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

我们看到，“子请求”方式的通信是在同一个虚拟主机内部进行的，所以 Nginx 核心在实现“子请求”的时候，就只调用了若干个 C 函数，完全不涉及任何网络或者 UNIX 套接字（socket）通信。我们由此可以看出“子请求”的执行效率是极高的。

回到先前对 Nginx 变量值容器的生命期的讨论，我们现在依旧可以说，它们的生命期是与当前请求相关联的。每个请求都有所有变量值容器的独立副本，只不过当前请求既可以是“主请求”，也可以是“子请求”。即便是父子请求之间，同名变量一般也不会相互干扰。让我们来通过一个小实验证明一下这个说法：

```
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";
}
```

在这个例子中，我们分别在 /main，/foo 和 /bar 这三个 location 配置块中为同一名字的变量，$var，分别设置了不同的值并予以输出。特别地，我们在 /main 接口中，故意在调用过 /foo 和 /bar 这两个“子请求”之后，再输出它自己的 $var 变量的值。请求 /main 接口的结果是这样的：

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

显然，/foo 和 /bar 这两个“子请求”在处理过程中对变量 $var 各自所做的修改都丝毫没有影响到“主请求” /main. 于是这成功印证了“主请求”以及各个“子请求”都拥有不同的变量 $var 的值容器副本。

不幸的是，一些 Nginx 模块发起的“子请求”却会自动共享其“父请求”的变量值容器，比如第三方模块 ngx_auth_request. 下面是一个例子：

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

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

这里我们在 /main 接口中先为 $var 变量赋初值 main，然后使用 ngx_auth_request 模块提供的配置指令 auth_request，发起一个到 /sub 接口的“子请求”，最后利用 echo 指令输出变量 $var 的值。而我们在 /sub 接口中则故意把 $var 变量的值改写成 sub. 访问 /main 接口的结果如下：

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

我们看到，/sub 接口对 $var 变量值的修改影响到了主请求 /main. 所以 ngx_auth_request 模块发起的“子请求”确实是与其“父请求”共享一套 Nginx 变量的值容器。

对于上面这个例子，相信有读者会问：“为什么‘子请求’ /sub 的输出没有出现在最终的输出里呢？”答案很简单，那就是因为 auth_request 指令会自动忽略“子请求”的响应体，而只检查“子请求”的响应状态码。当状态码是 2XX 的时候，auth_request 指令会忽略“子请求”而让 Nginx 继续处理当前的请求，否则它就会立即中断当前（主）请求的执行，返回相应的出错页。在我们的例子中，/sub “子请求”只是使用 echo 指令作了一些输出，所以隐式地返回了指示正常的 200 状态码。

如 ngx_auth_request 模块这样父子请求共享一套 Nginx 变量的行为，虽然可以让父子请求之间的数据双向传递变得极为容易，但是对于足够复杂的配置，却也经常导致不少难于调试的诡异 bug. 因为用户时常不知道“父请求”的某个 Nginx 变量的值，其实已经在它的某个“子请求”中被意外修改了。诸如此类的因共享而导致的不好的“副作用”，让包括 ngx_echo， ngx_lua，以及 ngx_srcache 在内的许多第三方模块都选择了禁用父子请求间的变量共享。

Nginx 变量漫谈（六）

Nginx 内建变量用在“子请求”的上下文中时，其行为也会变得有些微妙。

前面在 （三） 中我们已经知道，许多内建变量都不是简单的“存放值的容器”，它们一般会通过注册“存取处理程序”来表现得与众不同，而它们即使有存放值的容器，也只是用于缓存“存取处理程序”的计算结果。我们之前讨论过的 $args 变量正是通过它的“取处理程序”来返回当前请求的 URL 参数串。因为当前请求也可以是“子请求”，所以在“子请求”中读取 $args，其“取处理程序”会很自然地返回当前“子请求”的参数串。我们来看这样的一个例子：

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

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

这里在 /main 接口中，先用 echo 指令输出当前请求的 $args 变量的值，接着再用 echo_location 指令发起子请求 /sub. 这里值得注意的是，我们在 echo_location 语句中除了通过第一个参数指定“子请求”的 URI 之外，还提供了第二个参数，用以指定该“子请求”的 URL 参数串（即 a=1&b=2）。最后我们定义了 /sub 接口，在里面输出了一下 $args 的值。请求 /main 接口的结果如下：

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

显然，当 $args 用在“主请求” /main 中时，输出的就是“主请求”的 URL 参数串，c=3；而当用在“子请求” /sub 中时，输出的则是“子请求”的参数串，a=1&b=2。这种行为正符合我们的直觉。

与 $args 类似，内建变量 $uri 用在“子请求”中时，其“取处理程序”也会正确返回当前“子请求”解析过的 URI:

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

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

请求 /main 的结果是

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

这依然是我们所期望的。

但不幸的是，并非所有的内建变量都作用于当前请求。少数内建变量只作用于“主请求”，比如由标准模块 ngx_http_core 提供的内建变量 $request_method.

变量 $request_method 在读取时，总是会得到“主请求”的请求方法，比如 GET、POST 之类。我们来测试一下：

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

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

在这个例子里，/main 和 /sub 接口都会分别输出 $request_method 的值。同时，我们在 /main 接口里利用 echo_location 指令发起一个到 /sub 接口的 GET “子请求”。我们现在利用 curl 命令行工具来发起一个到 /main 接口的 POST 请求：

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

这里我们利用 curl 程序的 --data 选项，指定 hello 作为我们的请求体数据，同时 --data 选项会自动让发送的请求使用 POST 请求方法。测试结果证明了我们先前的预言， $request_method 变量即使在 GET “子请求” /sub 中使用，得到的值依然是“主请求” /main 的请求方法，POST.

有的读者可能觉得我们在这里下的结论有些草率，因为上例是先在“主请求”里读取（并输出） $request_method 变量，然后才发“子请求”的，所以这些读者可能认为这并不能排除 $request_method 在进入子请求之前就已经把第一次读到的值给缓存住，从而影响到后续子请求中的输出结果。不过，这样的顾虑是多余的，因为我们前面在 （五） 中也特别提到过，缓存所依赖的变量的值容器，是与当前请求绑定的，而由 ngx_echo 模块发起的“子请求”都禁用了父子请求之间的变量共享，所以在上例中， $request_method 内建变量即使真的使用了值容器作为缓存（事实上它也没有），它也不可能影响到 /sub 子请求。

为了进一步消除这部分读者的疑虑，我们不妨稍微修改一下刚才那个例子，将 /main 接口输出 $request_method 变量的时间推迟到“子请求”执行完毕之后：

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

location /sub {
```

```
        echo "sub method: $request_method";
    }
```

让我们重新测试一下：

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

可以看到，再次以 POST 方法请求 /main 接口的结果与原先那个例子完全一致，除了父子请求的输出顺序颠倒了过来（因为我们在本例中交换了 /main 接口中那两条输出配置指令的先后次序）。

由此可见，我们并不能通过标准的 $request_method 变量取得“子请求”的请求方法。为了达到我们最初的目的，我们需要求助于第三方模块 ngx_echo 提供的内建变量 $echo_request_method：

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

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

此时的输出终于是我们想要的了：

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

我们看到，父子请求分别输出了它们各自不同的请求方法，POST 和 GET.

类似 $request_method，内建变量 $request_uri 一般也返回的是“主请求”未经解析过的 URL，毕竟“子请求”都是在 Nginx 内部发起的，并不存在所谓的“未解析的”原始形式。

如果真如前面那部分读者所担心的，内建变量的值缓存在共享变量的父子请求之间起了作用，这无疑是灾难性的。我们前面在 （五） 中已经看到 ngx_auth_request 模块发起的“子请求”是与其“父请求”共享一套变量的。下面是一个这样的可怕例子：

```
    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";
        }
    }
```

这里我们使用久违了的 map 指令来把内建变量 $uri 的值映射到用户变量 $tag 上。当 $uri 的值为 /main 时，则赋予 $tag 值 1，当 $uri 取值 /sub 时，则赋予 $tag 值 2，其他情况都赋 0. 接着，我们在 /main 接口中先用 ngx_auth_request 模块的 auth_request 指令发起到 /sub 接口的子请求，然后再输出变量 $tag 的值。而在 /sub 接口中，我们直接输出变量 $tag. 猜猜看，如果我们访问接口 /main，将会得到什么样的输出呢？

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

咦？我们不是分明把 /main 这个值映射到 1 上的么？为什么实际输出的是 /sub 映射的结果 2 呢？

其实道理很简单，因为我们的 $tag 变量在“子请求” /sub 中首先被读取，于是在那里计算出了值 2（因为 $uri 在那里取值 /sub，而根据 map 映射规则，$tag 应当取值 2），从此就被 $tag 的值容器给缓存住了。而 auth_request 发起的“子请求”又是与“父请求”共享一套变量的，于是当 Nginx 的执行流回到“父请求”输出 $tag 变量的值时，Nginx 就直接返回缓存住的结果 2 了。这样的结果确实太意外了。

从这个例子我们再次看到，父子请求间的变量共享，实在不是一个好主意。

Nginx 变量漫谈（七）

在 （一） 中我们提到过，Nginx 变量的值只有一种类型，那就是字符串，但是变量也有可能压根就不存在有意义的值。没有值的变量也有两种特殊的值：一种是“不合法”（invalid），另一种是“没找到”（not found）。

举例说来，当 Nginx 用户变量 $foo 创建了却未被赋值时，$foo 的值便是“不合法”；而如果当前请求的 URL 参数串中并没有提及 XXX 这个参数，则 $arg_XXX 内建变量的值便是“没找到”。

无论是“不合法”也好，还是“没找到”也罢，这两种 Nginx 变量所拥有的特殊值，和空字符串（""）这种取值是完全不同的，比如 JavaScript 语言中也有专门的 undefined 和 null 这两种特殊值，而 Lua 语言中也有专门的 nil 值: 它们既不等同于空字符串，也不等同于数字 0，更不是布尔值 false. 其实 SQL 语言中的 NULL 也是类似的一种东西。

虽然前面在 （一） 中我们看到，由 set 指令创建的变量未初始化就用在“变量插值”中时，效果等同于空字符串，但那是因为 set 指令为它创建的变量自动注册了一个“取处理程序”，将“不合法”的变量值转换为空字符串。为了验证这一点，我们再重新看一下 （一） 中讨论过的那个例子：

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

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

这里为了简单起见，省略了原先写出的外围 server 配置块。在这个例子里，我们在 /bar 接口中用 set 指令隐式地创建了 $foo 变量这个名字，然后我们在 /foo 接口中不对 $foo 进行初始化就直接使用 echo 指令输出。我们当时测试 /foo 接口的结果是

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

从输出上看，未初始化的 $foo 变量确实和空字符串的效果等同。但细心的读者当时应该就已经注意到，对于上面这个请求，Nginx 的错误日志文件（一般文件名叫做 error.log）中多出一行类似下面这样的警告：

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

这一行警告是谁输出的呢？答案是 set 指令为 $foo 注册的“取处理程序”。当 /foo 接口中的 echo 指令实际执行的时候，它会对它的参数 "foo = $foo]" 进行“变量插值”计算。于是，参数串中的 $foo 变量会被读取，而 Nginx 会首先检查其值容器里的取值，结果它看到了“不合法”这个特殊值，于是它这才决定继续调用 $foo 变量的“取处理程序”。于是 $foo 变量的“取处理程序”开始运行，它向 Nginx 的错误日志打印出上面那条警告消息，然后返回一个空字符串作为 $foo 的值，并从此缓存在 $foo 的值容器中。

细心的读者会注意到刚刚描述的这个过程其实就是那些支持值缓存的内建变量的工作原理，只不过 set 指令在这里借用了这套机制来处理未正确初始化的 Nginx 变量。值得一提的是，只有“不合法”这个特殊值才会触发 Nginx 调用变量的“取处理程序”，而特殊值“没找到”却不会。

上面这样的警告一般会指示出我们的 Nginx 配置中存在变量名拼写错误，抑或是在错误的场合使用了尚未初始化的变量。因为值缓存的存在，这条警告在一个请求的生命期中也不会打印多次。当然， ngx_rewrite 模块专门提供了一条 uninitialized_variable_warn 配置指令可用于禁止这条警告日志。

刚才提到，内建变量 $arg_XXX 在请求 URL 参数 XXX 并不存在时会返回特殊值“找不到”，但遗憾的是在 Nginx 原生配置语言（我们估且这么称呼它）中是不能很方便地把它和空字符串区分开来的，比如：

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

这里我们输出 $arg_name 变量的值同时故意在请求中不提供 URL 参数 name:

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

我们看到，输出特殊值“找不到”的效果和空字符串是相同的。因为这一回是 Nginx 的“变量插值”引擎自动把“找不到”给忽略了。

那么我们究竟应当如何捕捉到“找不到”这种特殊值的踪影呢？换句话说，我们应当如何把它和空字符串给区分开来呢？显然，下面这个请求中，URL 参数 name 是有值的，而且其值应当是空字符串：

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

但我们却无法将之和前面完全不提供 name 参数的情况给区分开。

幸运的是，通过第三方模块 ngx_lua，我们可以轻松地在 Lua 代码中做到这一点。请看下面这个例子：

```
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
    ';
}
```

这个例子和前一个例子功能上非常接近，除了我们在 /test 接口中使用了 ngx_lua 模块的 content_by_lua 配置指令，嵌入了一小段我们自己的 Lua 代码来对 Nginx 变量 $arg_name 的特殊值进行判断。在这个例子中，当 $arg_name 的值为“没找到”（或者“不合法”）时，/foo 接口会输出 name: missing 这一行结果:

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

因为这是我们第一次接触到 ngx_lua 模块，所以需要先简单介绍一下。 ngx_lua 模块将 Lua 语言解释器（或者 LuaJIT 即时编译器）嵌入到了 Nginx 核心中，从而可以让用户在 Nginx 核心中直接运行 Lua 语言编写的程序。我们可以选择在 Nginx 不同的请求处理阶段插入我们的 Lua 代码。这些 Lua 代码既可以直接内联在 Nginx 配置文件中，也可以单独放置在外部 .lua 源码文件（或者 Lua 字节码文件）里，然后在 Nginx 配置文件中引用这些文件的路径。

回到上面这个例子，我们在 Lua 代码里引用 Nginx 变量都是通过 ngx.var 这个由 ngx_lua 模块提供的 Lua 接口。比如引用 Nginx 变量 $VARIABLE 时，就在 Lua 代码里写作 ngx.var.VARIABLE 就可以了。当 Nginx 变量 $arg_name 为特殊值“没找到”（或者“不合法”）时， ngx.var.arg_name 在 Lua 世界中的值就是 nil，即 Lua 语言里的“空”（不同于 Lua 空字符串）。我们在 Lua 里输出响应体内容的时候，则使用了 ngx.say 这个 Lua 函数，也是 ngx_lua 模块提供的，功能上等价于 ngx_echo 模块的 echo 配置指令。

现在，如果我们提供空字符串取值的 name 参数，则输出就和刚才不相同了：

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

在这种情况下，Nginx 变量 $arg_name 的取值便是空字符串，这既不是“没找到”，也不是“不合法”，因此在 Lua 里，ngx.var.arg_name 就返回 Lua 空字符串（""），和刚才的 Lua nil 值就完全区分开了。

这种区分在有些应用场景下非常重要，比如有的 web service 接口会根据 name 这个 URL 参数是否存在来决定是否按 name 属性对数据集合进行过滤，而显然提供空字符串作为 name 参数的值，也会导致对数据集中取值为空串的记录进行筛选操作。

不过，标准的 $arg_XXX 变量还是有一些局限，比如我们用下面这个请求来测试刚才那个 /test 接口：

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

此时，$arg_name 变量仍然读出“找不到”这个特殊值，这就明显有些违反常识。此外， $arg_XXX 变量在请求 URL 中有多个同名 XXX 参数时，就只会返回最先出现的那个 XXX 参数的值，而默默忽略掉其他实例：

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

要解决这些局限，可以直接在 Lua 代码中使用 ngx_lua 模块提供的 ngx.req.get_uri_args 函数。

Nginx 变量漫谈（八）

与 $arg_XXX 类似，我们在 （二） 中提到过的内建变量 $cookie_XXX 变量也会在名为 XXX 的 cookie 不存在时返回特殊值“没找到”：

```
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
    ';
}
```

利用 curl 命令行工具的 --cookie name=value 选项可以指定 name=value 为当前请求携带的 cookie（通过添加相应的 Cookie 请求头）。下面是若干次测试结果：

```
$ 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
```

我们看到，cookie user 不存在以及取值为空字符串这两种情况被很好地区分开了：当 cookie user 不存在时，Lua 代码中的 ngx.var.cookie_user 返回了期望的 Lua nil 值。

在 Lua 里访问未创建的 Nginx 用户变量时，在 Lua 里也会得到 nil 值，而不会像先前的例子那样直接让 Nginx 拒绝加载配置：

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

这里假设我们并没有在当前的 nginx.conf 配置文件中创建过用户变量 $blah，然后我们在 Lua 代码中通过 ngx.var.blah 直接引用它。上面这个配置可以顺利启动，因为 Nginx 在加载配置时只会编译 content_by_lua 配置指令指定的 Lua 代码而不会实际执行它，所以 Nginx 并不知道 Lua 代码里面引用了 $blah 这个变量。于是我们在运行时也会得到 nil 值。而 ngx_lua 提供的 ngx.say 函数会自动把 Lua 的 nil 值格式化为字符串 "nil" 输出，于是访问 /test 接口的结果是：

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

这正是我们所期望的。

上面这个例子中另一个值得注意的地方是，我们在 content_by_lua 配置指令的参数中提及了 $blah 符号，但却并没有触发“变量插值”（否则 Nginx 会在启动时抱怨 $blah 未创建）。这是因为 content_by_lua 配置指令并不支持参数的“变量插值”功能。我们前面在 （一） 中提到过，配置指令的参数是否允许“变量插值”，其实取决于该指令的实现模块。

设计返回“不合法”这一特殊值的例子是困难的，因为我们前面在 （七） 中已经看到，由 set 指令创建的变量在未初始化时确实是“不合法”，但一旦尝试读取它们时，Nginx 就会自动调用其“取处理程序”，而它们的“取处理程序”会自动返回空字符串并将之缓存住。于是我们最终得到的是完全合法的空字符串。下面这个使用了 Lua 代码的例子证明了这一点：

```
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]";
    }
```

请求 `/foo` 接口的结果是：

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

我们看到在 Lua 里面读取未初始化的 Nginx 变量 `$foo` 时得到的是空字符串。

最后值得一提的是，虽然前面反复指出 Nginx 变量只有字符串这一种数据类型，但这并不能阻止像 ngx_array_var 这样的第三方模块让 Nginx 变量也能存放数组类型的值。下面就是这样的一个例子：

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

        echo $res;
    }
```

这个例子中使用了 ngx_array_var 模块的 `array_split`、 `array_map` 和 `array_join` 这三条配置指令，其含义很接近 Perl 语言中的内建函数 `split`、`map` 和 `join`（当然，其他脚本语言也有类似的等价物）。我们来看看访问 `/test` 接口的结果：

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

我们看到，使用 ngx_array_var 模块可以很方便地处理这样具有不定个数的组成元素的输入数据，例如此例中的 `names` URL 参数值就是由不定个数的逗号分隔的名字所组成。不过，这种类型的复杂任务通过 ngx_lua 来做通常会更灵活而且更容易维护。

至此，本系列教程对 Nginx 变量的介绍终于可以告一段落了。我们在这个过程中接触到了许多标准的和第三方的 Nginx 模块，这些模块让我们得以很轻松地构造出许多有趣的小例子，从而可以深入探究 Nginx 变量的各种行为和特性。在后续的教程中，我们还会有很多机会与这些模块打交道。

通过前面讨论过的众多例子，我们应当已经感受到 Nginx 变量在 Nginx 配置语言中所扮演的重要角色：它是获取 Nginx 中各种信息（包括当前请求的信息）的主要途径和载体，同时也是各个模块之间传递数据的主要媒介之一。在后续的教程中，我们会经常看到 Nginx 变量的身影，所以现在很好地理解它们是非常重要的。

在下一个系列的教程，即 Nginx 配置指令的执行顺序系列 中，我们将深入探讨 Nginx 配置指令的执行顺序以及请求的各个处理阶段，因为很多 Nginx 用户都搞不清楚他们书写的众多配置指令之间究竟是按照何种时间顺序执行的，也搞不懂为什么这些指令实际执行的顺序经常和配置文件里的书写顺序大相径庭。

Nginx 配置指令的执行顺序（一）

大多数 Nginx 新手都会频繁遇到这样一个困惑，那就是当同一个 `location` 配置块使用了多个 Nginx 模块的配置指令时，这些指令的执行顺序很可能会跟它们的书写顺序大相径庭。于是许多人选择了“试错法”，然后他们的配置文件就时常被改得一片狼藉。这个系列的教程就旨在帮助读者逐步地理解这些配置指令背后的执行时间和先后顺序的奥秘。

现在就来看这样一个令人困惑的例子：

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

从这个例子的本意来看，我们期望的输出是一行 `32` 和一行 `56`，因为我们第一次用 echo 配置指令输出了 `$a` 变量的值以后，又紧接着使用 set 配置指令修改了 `$a`. 然而不幸的是，事实并非如此：

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

我们看到，语句 `set $a 56` 似乎在第一条 `echo $a` 语句之前就执行过了。这究竟是为什么呢？难道我们遇到了 Nginx 中的一个 bug？

显然，这里并没有 Nginx 的 bug；要理解这里发生的事情，就首先需要知道 Nginx 处理每一个用户请求时，都是按照若干个不同阶段（phase）依次处理的。

Nginx 的请求处理阶段共有 11 个之多，我们先介绍其中 3 个比较常见的。按照它们执行时的先后顺序，依次是 `rewrite` 阶段、`access` 阶段以及 `content` 阶段（后面我们还有机会见到其他更多的处理阶段）。

所有 Nginx 模块提供的配置指令一般只会注册并运行在其中的某一个处理阶段。比如上例中的 set 指令就是在 `rewrite` 阶段运行的，而 echo 指令就只会在 `content` 阶段运行。前面我们已经知道，在单个请求的处理过程中，`rewrite` 阶段总是在 `content` 阶段之前执行，因此属于 `rewrite` 阶段的配置指令也总是会无条件地在 `content` 阶段的配置指令之前执行。于是在同一个 `location` 配置块中， set 指令总是会在 echo 指令之前执行，即使我们在

配置文件中有意把 set 语句写在 echo 语句的后面。

回到刚才那个例子，

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

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

实际的执行顺序应当是

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

即先在 rewrite 阶段执行完这里的两条 set 赋值语句，然后再在后面的 content 阶段依次执行那两条 echo 语句。分属两个不同处理阶段的配置指令之间是不能穿插着运行的。

为了进一步验证这一点，我们不妨借助 Nginx 的“调试日志”（debug log）来一窥 Nginx 的实际执行过程。

因为这是我们第一次提及 Nginx 的“调试日志”，所以有必要先简单介绍一下它的启用方法。“调试日志”默认是禁用的，因为它会引入比较大的运行时开销，让 Nginx 服务器显著变慢。一般我们需要重新编译和构造 Nginx 可执行文件，并且在调用 Nginx 源码包提供的 ./configure 脚本时传入 --with-debug 命令行选项。例如我们下载完 Nginx 源码包后在 Linux 或者 Mac OS X 等系统上构建时，典型的步骤是这样的：

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

如果你使用的是我维护的 ngx_openresty 软件包，则同样可以向它的 ./configure 脚本传递 --with-debug 命令行选项。

当我们启用 --with-debug 选项重新构建好调试版的 Nginx 之后，还需要同时在配置文件中通过标准的 error_log 配置指令为错误日志使用 debug 日志级别（这同时也是最低的日志级别）：

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

这里重要的是 error_log 指令的第二个参数，debug，而前面第一个参数是错误日志文件的路径，logs/error.log. 当然，你也可以指定其他路径，但后面我们会检查这个文件的内容，所以请特别留意一下这里实际配置的文件路径。

现在我们重新启动 Nginx（注意，如果 Nginx 可执行文件也被更新过，仅仅让 Nginx 重新加载配置是不够的，需要关闭再启动 Nginx 主服务进程），然后再请求一下我们刚才那个示例接口：

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

现在可以检查一下前面配置的 Nginx 错误日志文件中的输出。因为文件中的输出比较多（在我的机器上有 700 多行），所以不妨用 grep 命令在终端上过滤出我们感兴趣的部分：

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

在我机器上的输出是这个样子的（为了方便呈现，这里对 grep 命令的实际输出作了一些简单的编辑，略去了每一行的行首时间戳）：

```
[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?"
```

这里需要稍微解释一下这些调试信息的具体含义。 set 配置指令在实际运行时会打印出两行以 http script 起始的调试信息，其中第一行信息是 set 语句中被赋予的值，而第二行则是 set 语句中被赋值的 Nginx 变量名。于是上面首先过滤出来的

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

这两行就对应我们例子中的配置语句

```
set $a 32;
```

而接下来这两行调试信息

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

则对应配置语句

```
set $a 56;
```

此外，凡在 Nginx 中输出响应体数据时，都会调用 Nginx 的所谓“输出过滤器”（output filter），我们一直在使用的 echo 指令自然也不例外。而一旦调用

Nginx 的“输出过滤器”，便会产生类似下面这样的调试信息：

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

当然，这里的 "/test?" 部分对于其他接口可能会发生变化，因为它显示的是当前请求的 URI. 这样联系起来看，就不难发现，上例中的那两条 set 语句确实都是在那两条 echo 语句之前执行的。

细心的读者可能会问，为什么这个例子明明只使用了两条 echo 语句进行输出，但却有三行 http output filter 调试信息呢？其实，前两行 http output filter 信息确实分别对应那两条 echo 语句，而最后那一行信息则是对应 ngx_echo 模块输出指示响应体末尾的结束标记。正是为了输出这个特殊的结束标记，才会多出一次对 Nginx “输出过滤器”的调用。包括 ngx_proxy 在内的许多模块在输出响应体数据流时都具有此种行为。

现在我们就不会再为前面那个例子输出两行一模一样的 56 而感到惊讶了。我们根本没有机会在第二条 set 语句之前用 echo 输出。幸运的是，仍然可以借助一些小技巧来达到最初的目的：

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

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

此时的输出便符合那个问题示例的初衷了：

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

这里通过引入新的用户变量 $saved_a，在改写 $a 之前及时保存了 $a 的初始值。而对于多条 set 指令而言，它们之间的执行顺序是由 ngx_rewrite 模块来保证与书写顺序相一致的。同理， ngx_echo 模块自身也会保证它的多条 echo 指令之间的执行顺序。

细心的读者应当发现，我们在 Nginx 变量漫谈系列 的示例中已经广泛使用了这种技巧，来绕过因处理阶段而引起的指令执行顺序上的限制。

看到这里，有的读者可能会问：“那么我在使用一条陌生的配置指令之前，如何知道它究竟运行在哪一个处理阶段呢？”答案是：查看该指令的文档（当然，高级开发人员也可以直接查看模块的 C 源码）。在许多模块的文档中，都会专门标记其配置指令所运行的具体阶段。例如 echo 指令的文档中有这么一行：

```
phase: content
```

这一行便是说，当前配置指令运行在 content 阶段。如果你使用的 Nginx 模块碰巧没有指示运行阶段的文档，可以直接联系该模块的作者请求补充。不过，值得一提的是，并非所有的配置指令都与某个处理阶段相关联，例如我们先前在 Nginx 变量漫谈（一） 中提到过的 geo 指令以及在 Nginx 变量漫谈（四） 中介绍过的 map 指令。这些不与处理阶段相关联的配置指令基本上都是“声明性的”（declarative），即不直接产生某种动作或者过程。Nginx 的作者 Igor Sysoev 在公开场合曾不止一次地强调，Nginx 配置文件所使用的语言本质上是“声明性的”，而非“过程性的”（procedural）。

Nginx 配置指令的执行顺序（二）

我们前面已经知道，当 set 指令用在 location 配置块中时，都是在当前请求的 rewrite 阶段运行的。事实上，在此上下文中， ngx_rewrite 模块中的几乎全部指令，都运行在 rewrite 阶段，包括 Nginx 变量漫谈（二） 中介绍过的 rewrite 指令。不过，值得一提的是，当这些指令使用在 server 配置块中时，则会运行在一个我们尚未提及的更早的处理阶段，server-rewrite 阶段。

Nginx 变量漫谈（二） 中介绍过的 ngx_set_misc 模块的 set_unescape_uri 指令同样也运行在 rewrite 阶段。特别地， ngx_set_misc 模块的指令还可以和 ngx_rewrite 的指令混合在一起依次执行。我们来看这样的一个例子：

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

    echo $c;
}
```

访问这个接口可以得到：

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

我们看到， set_unescape_uri 语句前后的 set 语句都按书写时的顺序一前一后地执行了。

为了进一步确认这一点，我们不妨再检查一下 Nginx 的“调试日志”（如果你还不清楚如何开启“调试日志”的话，可以参考 （一） 中的步骤）：

```
grep -E 'http script (value|copy|set)' t/servroot/logs/error.log
```

过滤出来的调试日志信息如下所示：

```
[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
```

开头的两行信息

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

就对应我们的配置语句

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

而接下来的两行

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

则对应配置语句

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

我们看到第一行信息与 set 指令略有区别，多了 "(post filter)" 这个标记，而且最后显示出 URI 解码操作确实如我们期望的那样工作了，即 "hello%20world" 在这里被成功解码为 "hello world".

而最后两行调试信息

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

则对应最后一条 set 语句：

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

注意，因为这条指令在为 $c 变量赋值时使用了“变量插值”功能，所以第一行调试信息是以 http script copy 起始的，后面则是拼接到最终取值的字符串常量 "!".

把这些调试信息联系起来看，我们不难发现，这些配置指令的实际执行顺序是：

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

这与它们在配置文件中的书写顺序完全一致。

我们在 Nginx 变量漫谈（七） 中初识了第三方模块 ngx_lua，它提供的 set_by_lua 配置指令也和 ngx_set_misc 模块的指令一样，可以和 ngx_rewrite 模块的指令混合使用。 set_by_lua 指令支持通过一小段用户 Lua 代码来计算出一个结果，然后赋给指定的 Nginx 变量。和 set 指令相似， set_by_lua 指令也有自动创建不存在的 Nginx 变量的功能。

下面我们就来看一个 set_by_lua 指令与 set 指令混合使用的例子：

```
    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;
    }
```

这里我们先将 $a 和 $b 变量分别初始化为 32 和 56，然后利用 set_by_lua 指令内联一行我们自己指定的 Lua 代码，计算出 Nginx 变量 $a 和 $b 的“代数和”（sum），并赋给变量 $c，接着利用“变量插值”功能，把变量 $a、 $b 和 $c 的值拼接成一个字符串形式的等式，赋予变量 $equation，最后再用 echo 指令输出 $equation 的值。

这个例子值得注意的地方是：首先，我们在 Lua 代码中是通过 ngx.var.VARIABLE 接口来读取 Nginx 变量 $VARIABLE 的；其次，因为 Nginx 变量的值只有字符串这一种类型，所以在 Lua 代码里读取 ngx.var.a 和 ngx.var.b 时得到的其实都是 Lua 字符串类型的值 "32" 和 "56"；接着，我们对两个字符串作加法运算会触发 Lua 对加数进行自动类型转换（Lua 会把两个加数先转换为数值类型再求和）；然后，我们在 Lua 代码中把最终结果通过 return 语句返回给外面的 Nginx 变量 $c；最后， ngx_lua 模块在给 $c 实际赋值之前，也会把 return 语句返回的数值类型的结果，也就是 Lua 加法计算得出的“和”，自动转换为字符串（这同样是因为 Nginx 变量的值只能是字符串）。

这个例子的实际运行结果符合我们的期望：

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

于是这验证了 set_by_lua 指令确实也可以和 set 这样的 ngx_rewrite 模块提供的指令混合在一起工作。

还有不少第三方模块，例如 Nginx 变量漫谈（八） 中介绍过的 ngx_array_var 以及后面即将接触到的用于加解密用户会话（session）的 ngx_encrypted_session，也都可以和 ngx_rewrite 模块的指令无缝混合工作。

标准 ngx_rewrite 模块的应用是如此广泛，所以能够和它的配置指令混合使用的第三方模块是幸运的。事实上，上面提到的这些第三方模块都采用了特殊的技术，将它们自己的配置指令“注入”到了 ngx_rewrite 模块的指令序列中（它们都借助了 Marcus Clyne 编写的第三方模块 ngx_devel_kit）。换句话说，更多常规的在 Nginx 的 rewrite 阶段注册和运行指令的第三方模块就没那么幸运了。这些“常规模块”的指令虽然也运行在 rewrite 阶段，但其配置指令和 ngx_rewrite 模块（以及同一阶段内的其他模块）都是分开独立执行的。在运行时，不同模块的配置指令集之间的先后顺序一般是不确定的（严格来说，一般是由模块的加载顺序决定的，但也有例外的情况）。比如 A 和 B 两个模块都在 rewrite 阶段运行指令，于是要么是 A 模块的所有指令全部执行完再执行 B 模块的那些指令，要么就是反过来，把 B 的指令全部执行完，再去运行 A 的指令。除非模块的文档中有明确的交待，否则用户一般不应编写依赖于此种不确定顺序的配置。

Nginx 配置指令的执行顺序（三）

如前文所述，除非像 ngx_set_misc 模块那样使用特殊技术，其他模块的配置指令即使是在 rewrite 阶段运行，也不能和 ngx_rewrite 模块的指令混合使用。不妨来看几个这样的例子。

第三方模块 ngx_headers_more 提供了一系列配置指令，用于操纵当前请求的请求头和响应头。其中有一条名叫 more_set_input_headers 的指令可以在 rewrite 阶段改写指定的请求头（或者在请求头不存在时自动创建）。这条指令总是运行在 rewrite 阶段的末尾，该指令的文档中有这么一行标记：

```
phase: rewrite tail
```

其中的 rewrite tail 的意思就是 rewrite 阶段的末尾。

既然运行在 rewrite 阶段的末尾，那么也就总是会运行在 ngx_rewrite 模块的指令之后，即使我们在配置文件中把它写在前面，例如：

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

这个例子用到的 $http_XXX 内建变量在读取时会返回当前请求中名为 XXX 的请求头，我们在 Nginx 变量漫谈（二） 中曾经简单提过它。需要注意的是， $http_XXX 变量在匹配请求头时会自动对请求头的名字进行归一化，即将名字的大写字母转换为小写字母，同时把间隔符（-）替换为下划线（_），所以变量名 $http_x_species 才得以成功匹配 more_set_input_headers 语句中设置的请求头 X-Species.

此例书写的指令顺序会误导我们认为 /test 接口输出的 X-Species 头的值是 dog，然而实际的结果却并非如此：

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

显然，写在 more_set_input_headers 指令之后的 set $value cat 语句却先执行了。

上面这个例子证明了即使运行在同一个请求处理阶段，分属不同模块的配置指令也可能会分开独立运行（除非像 ngx_set_misc 等模块那样针对 ngx_rewrite 模块提供特殊支持）。换句话说，在单个请求处理阶段内部，一般也会以 Nginx 模块为单位进一步地划分出内部子阶段。

第三方模块 ngx_lua 提供的 rewrite_by_lua 配置指令也和 more_set_input_headers 一样运行在 rewrite 阶段的末尾。我们来验证一下：

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

这里我们在 rewrite_by_lua 语句内联的 Lua 代码中对 Nginx 变量 $a 进行了自增计算。从该例的指令书写顺序上看，我们或许会期望输出是 56，可是因为 rewrite_by_lua 会在所有的 set 语句之后执行，所以结果是 57：

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

显然， rewrite_by_lua 指令的行为不同于我们前面在 （二） 中介绍过的 set_by_lua 指令。

有的读者可能要问，既然 more_set_input_headers 和 rewrite_by_lua 指令都运行在 rewrite 阶段的末尾，那么它们之间的先后顺序又是怎样的呢？答案是：不一定。我们应当避免写出依赖它们二者间顺序的配置。

Nginx 的 rewrite 阶段是一个比较早的请求处理阶段，这个阶段的配置指令一般用来对当前请求进行各种修改（比如对 URI 和 URL 参数进行改写），或者创建并初始化一系列后续处理阶段可能需要的 Nginx 变量。当然，也不能阻止一些用户在 rewrite 阶段做一系列更复杂的事情，比如读取请求体，或者访问数据库等远方服务，毕竟有 rewrite_by_lua 这样的指令可以嵌入任意复杂的 Lua 代码。

在 rewrite 阶段之后，有一个名叫 access 的请求处理阶段。Nginx 变量漫谈（五） 中介绍过的第三方模块 ngx_auth_request 的指令就运行在 access 阶段。在 access 阶段运行的配置指令多是执行访问控制性质的任务，比如检查用户的访问权限，检查用户的来源 IP 地址是否合法，诸如此类。

例如，标准模块 ngx_access 提供的 allow 和 deny 配置指令可用于控制哪些 IP 地址可以访问，哪些不可以：

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

    echo "hello world";
}
```

这个 /hello 接口被配置为只允许从本机（IP 地址为保留的 127.0.0.1）访问，而从其他 IP 地址访问都会被拒（返回 403 错误页）。 ngx_access 模块自己的多条配置指令之间是按顺序执行的，直到遇到第一条满足条件的指令就不再执行后续的 allow 和 deny 指令。如果首先匹配的指令是 allow，则会继续执行后续其他模块的指令或者跳到后续的处理阶段；而如果首先满足的是 deny 则会立即中止当前整个请求的处理，并立即返回 403 错误页。所以看上面这个例子，如果是从本地访问的，则首先匹配 allow 127.0.0.1 这一条语句，于是 Nginx 就继续往下执行其他模块的指令以及后续的处理阶段；而如果是从其他机器访问，则首先匹配的则是 deny all 这一条语句，即拒绝所有地址，它会导致 403 错误页立即返回给客户端。

我们来实测一下。从本机访问这个接口可以得到

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

而从另一台机器访问这台机器（假设运行 Nginx 的机器地址是 192.168.1.101）提供的接口时则得到

```
$ 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>
```

值得一提的是， ngx_access 模块还支持所谓的“CIDR 记法”来表示一个网段，例如 `169.200.179.4/24` 则表示路由前缀是 `169.200.179.0`（或者说子网掩码是 `255.255.255.0`）的网段。

因为 ngx_access 模块的指令运行在 `access` 阶段，而 `access` 阶段又处于 `rewrite` 阶段之后，所以前面我们见到的所有那些在 `rewrite` 阶段运行的配置指令，都总是在 allow 和 deny 之前执行，而无论它们在配置文件中的书写顺序是怎样的。所以，为了避免阅读配置时的混乱，我们应该总是让指令的书写顺序和它们的实际执行顺序保持一致。

Nginx 配置指令的执行顺序（四）

ngx_lua 模块提供了配置指令 access_by_lua，用于在 `access` 请求处理阶段插入用户 Lua 代码。这条指令运行于 `access` 阶段的末尾，因此总是在 allow 和 deny 这样的指令之后运行，虽然它们同属 `access` 阶段。一般我们通过 access_by_lua 在 ngx_access 这样的模块检查过客户端 IP 地址之后，再通过 Lua 代码执行一系列更为复杂的请求验证操作，比如实时查询数据库或者其他后端服务，以验证当前用户的身份或权限。

我们来看一个简单的例子，利用 access_by_lua 来实现 ngx_access 模块的 IP 地址过滤功能：

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

        ngx.exit(403)
    ';

    echo "hello world";
}
```

这里在 Lua 代码中通过引用 Nginx 标准的内建变量 $remote_addr 来获取字符串形式的客户端 IP 地址，然后用 Lua 的 `if` 语句判断是否为本机地址，即是否等于 `127.0.0.1`. 如果是本机地址，则直接利用 Lua 的 `return` 语句返回，让 Nginx 继续执行后续的请求处理阶段（包括 echo 指令所处的 `content` 阶段）；而如果不是本机地址，则通过 ngx_lua 模块提供的 Lua 函数 ngx.exit 中断当前的整个请求处理流程，直接返回 `403` 错误页给客户端。

这个例子在功能上完全等价于先前在 （三） 中介绍过的那个使用 ngx_access 模块的例子：

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

    echo "hello world";
}
```

虽然这两个例子在功能上完全相同，但在性能上还是有区别的，毕竟 ngx_access 是用纯 C 实现的专门化的 Nginx 模块。

下面我们不妨来实际测量一下这两个例子的性能差别。因为我们使用 Nginx 就是为了追求性能，而量化的性能比较，在工程上具有很大的现实意义，所以我们顺便介绍一下重要的测量技术。由于无论是 ngx_access 还是 ngx_lua 在进行 IP 地址验证方面的性能都非常之高，所以为了减少测量误差，我们希望能对 `access` 阶段的用时进行直接测量。为了做到这一点，传统的做法一般会涉及到修改 Nginx 源码，自己插入专门的计时代码和统计输出代码，抑或是重新编译 Nginx 以启用像 `GNU gprof` 这样专门的性能监测工具。

幸运的是，在新一点的 Solaris, Mac OS X, 以及 FreeBSD 等系统上存在一个叫做 `dtrace` 的工具，可以对任意的用户程序进行微观性能分析（以及行为分析），而无须对用户程序的源码进行修改或者对用户程序进行重新编译。因为 Mac OS X 10.5 以后就自带了 `dtrace`，所以为方便起见，下面在我的 MacBook Air 笔记本上演示一下这里的测量过程。

首先，在 Mac OS X 系统中打开一个命令行终端，在某一个文件目录下面创建一个名为 `nginx-access-time.d` 的文件，并编辑内容如下：

```
#!/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;
}
```

保存好此文件后，再赋予它可执行权限：

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

这个 .d 文件中的代码是用 dtrace 工具自己提供的 D 语言来编写的（注意，这里的 D 语言并不同于 Walter Bright 作为另一种“更好的 C++”而设计的 D 语言）。由于本系列教程并不打算介绍如何编写 dtrace 的 D 脚本，同时理解这个脚本需要不少有关 Nginx 内部源码实现的细节，所以这里我们不展开介绍。大家只需要知道这个脚本的功能是：统计指定的 Nginx worker 进程在处理每个请求时，平均花费在 access 阶段上的时间。

现在来演示一下这个 D 脚本的运行方法。这个脚本接受一个命令行参数用于指定监视的 Nginx worker 进程的进程号（pid）。由于 Nginx 支持多 worker 进程，所以我们测试时发起的 HTTP 请求可能由其中任意一个 worker 进程服务。为了确保所有测试请求都为固定的 worker 进程处理，不妨在 nginx.conf 配置文件中指定只启用一个 worker 进程：

```
worker_processes 1;
```

重启 Nginx 服务器之后，可以利用 ps 命令得到当前 worker 进程的进程号：

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

在我机器上的一次典型输出是

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

其中第一列的数值便是我的 nginx worker 进程的进程号，10975。如果你得到的输出不止一行，则通常意味着你的系统中同时运行着多个 Nginx 服务器实例，或者当前 Nginx 实例启用了多个 worker 进程。

接下来使用刚刚得到的 worker 进程号以及 root 身份来运行 nginx-access-time.d 脚本：

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

如果一切正常，则会看到这样一行输出：

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

这行输出是说，我们的 D 脚本已成功向目标进程动态植入了 4 个 dtrace “探针”（probe）。紧接着这个脚本就挂起了，表明 dtrace 工具正在对进程 10975 进行持续监视。

然后我们再打开一个新终端，在那里使用 curl 这样的工具多次请求我们正在监视的接口

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

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

最后我们回到原先那个一直在运行 D 脚本的终端，按下 Ctrl-C 组合键中止 dtrace 的运行。而该脚本在退出时会向终端打印出最终统计结果。例如我的终端此时是这个样子的：

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

最后一行输出 19219 便是那几次 curl 请求在 access 阶段的平均用时（以纳秒，即 10 的负 9 次方秒为单位）。

通过上面介绍的步骤，可以通过 nginx-access-time.d 脚本分别统计出各种不同的 Nginx 配置下 access 阶段的平均用时。针对我们感兴趣的三种情况可以进行三组平行试验，即使用 ngx_access 过滤 IP 地址的情况，使用 access_by_lua 过滤 IP 地址的情况，以及不在 access 阶段使用任何配置指令的情况。最后一种情况属于“空白对照组”，用于校正测试过程中因 dtrace 探针等其他因素而引入的“系统误差”。另外，为了最小化各种不可控的“随机误差”，可以用 ab 这样的批量测试工具来取代 curl 发起连续十万次以上的请求，例如

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

这样我们的 D 脚本统计出来的平均值将更加接近“真实值”。

在我的苹果系统上，一次典型的测试结果如下：

```
ngx_access 组               18146
access_by_lua 组            35011
空白对照组                  15887
```

把前两组的结果分别减去“空白对照组”的结果可以得到

```
ngx_access 组                2259
access_by_lua 组            19124
```

可以看到， ngx_access 组比 access_by_lua 组快了大约一个数量级，这正是我们所预期的。不过其绝对时间差是极小的，对于我的 Intel Core2Duo 1.86 GHz 的 CPU 而言，也只有区区十几微秒，或者说是在十万分之一秒的量级。

当然，上面使用 access_by_lua 的例子还可以通过换用 $binary_remote_addr 内建变量进行优化，因为 $binary_remote_addr 读出的是二进制形式的 IP 地

址，而 $remote_addr 则返回更长一些的字符串形式的地址。更短的地址意味着用 Lua 进行字符串比较时通常可以更快。

值得注意的是，如果按 （一） 中介绍的方法为 Nginx 开启了“调试日志”的话，上面统计出来的时间会显著增加，因为“调试日志”自身的开销是很大的。

Nginx 配置指令的执行顺序（五）

Nginx 的 content 阶段是所有请求处理阶段中最为重要的一个，因为运行在这个阶段的配置指令一般都肩负着生成“内容”（content）并输出 HTTP 响应的使命。正因为其重要性，这个阶段的配置指令也异常丰富，例如前面我们一直在示例中广泛使用的 echo 指令，在 Nginx 变量漫谈（二） 中接触到的 echo_exec 指令， Nginx 变量漫谈（三） 中接触到的 proxy_pass 指令，Nginx 变量漫谈（五） 中介绍过的 echo_location 指令，以及 Nginx 变量漫谈（七） 中介绍过的 content_by_lua 指令，都运行在这个阶段。

content 阶段属于一个比较靠后的处理阶段，运行在先前介绍过的 rewrite 和 access 这两个阶段之后。当和 rewrite、access 阶段的指令一起使用时，这个阶段的指令总是最后运行，例如：

```
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";
}
```

这个例子中各个配置指令的执行顺序便是它们的书写顺序。测试结果完全符合预期：

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

即使改变它们的书写顺序，也不会影响到执行顺序。其中， set 指令来自 ngx_rewrite 模块，运行于 rewrite 阶段；而 rewrite_by_lua 指令来自 ngx_lua 模块，运行于 rewrite 阶段的末尾；接下来， deny 指令来自 ngx_access 模块，运行于 access 阶段；再下来， access_by_lua 指令同样来自 ngx_lua 模块，运行于 access 阶段的末尾；最后，我们的老朋友 echo 指令则来自 ngx_echo 模块，运行在 content 阶段。

这个例子展示了通过同时使用多个处理阶段的配置指令来实现多个模块协同工作的效果。在这个过程中，Nginx 变量则经常扮演着在指令间乃至模块间传递（小份）数据的角色。这些配置指令的执行顺序，也强烈地受到请求处理阶段的影响。

进一步地，在 rewrite 和 access 这两个阶段，多个模块的配置指令可以同时使用，譬如上例中的 set 指令和 rewrite_by_lua 指令同处 rewrite 阶段，而 deny 指令和 access_by_lua 指令则同处 access 阶段。但不幸的是，这通常不适用于 content 阶段。

绝大多数 Nginx 模块在向 content 阶段注册配置指令时，本质上是在当前的 location 配置块中注册所谓的“内容处理程序”（content handler）。每一个 location 只能有一个“内容处理程序”，因此，当在 location 中同时使用多个模块的 content 阶段指令时，只有其中一个模块能成功注册“内容处理程序”。考虑下面这个有问题的例子：

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

这里， ngx_echo 模块的 echo 指令和 ngx_lua 模块的 content_by_lua 指令同处 content 阶段，于是只有其中一个模块能注册和运行这个 location 的“内容处理程序”：

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

实际运行结果表明，写在后面的 content_by_lua 指令反而胜出了，而 echo 指令则完全没有运行。具体哪一个模块的指令会胜出是不确定的，例如把上例中的 echo 语句和 content_by_lua 语句交换顺序，则输出就会变成 hello，即 ngx_echo 模块胜出。所以我们应当避免在同一个 location 中使用多个模块的 content 阶段指令。

将上例中的 content_by_lua 指令替换为 echo 指令就可以如愿了：

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

测试结果证明了这一点：

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

这里使用多条 echo 指令是没问题的，因为它们同属 ngx_echo 模块，而且 ngx_echo 模块规定和实现了它们之间的执行顺序。值得一提的是，并非所有模块的指令都支持在同一个 location 中被使用多次，例如 content_by_lua 就只能使用一次，所以下面这个例子是错误的：

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

这个配置在 Nginx 启动时就会报错：

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

正确的写法应当是：

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

即在 content_by_lua 内联的 Lua 代码中调用两次 ngx.say 函数，而不是在当前 location 中使用两次 content_by_lua 指令。

类似地， ngx_proxy 模块的 proxy_pass 指令和 echo 指令也不能同时用在一个 location 中，因为它们也同属 content 阶段。不少 Nginx 新手都会犯类似下面这样的错误：

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

这个例子表面上是想在 ngx_proxy 模块返回的内容前后，通过 ngx_echo 模块的 echo 指令分别输出字符串 "before..." 和 "after..."，但其实只有其中一个模块能在 content 阶段运行。测试结果表明，在这个例子中是 ngx_proxy 模块胜出，而 ngx_echo 模块的 echo 指令根本没有运行：

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

要实现这个例子希望达到的效果，需要改用 ngx_echo 模块提供的 echo_before_body 和 echo_after_body 这两条配置指令：

```
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";
}
```

测试结果表明这一次我们成功了：

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

配置指令 echo_before_body 和 echo_after_body 之所以可以和其他模块运行在 content 阶段的指令一起工作，是因为它们运行在 Nginx 的“输出过滤器”中。前面我们在 （一） 中分析 echo 指令产生的“调试日志”时已经知道，Nginx 在输出响应体数据时都会调用“输出过滤器”，所以 ngx_echo 模块才有机会在“输出过滤器”中对 ngx_proxy 模块产生的响应体输出进行修改（即在首尾添加新的内容）。值得一提的是，“输出过滤器”并不属于 （一） 中提到的那 11 个请求处理阶段（毕竟许多阶段都可以通过输出响应体数据来调用“输出过滤器”），但这并不妨碍 echo_before_body 和 echo_after_body 指令在文档中标记下面这一行：

```
phase: output filter
```

这一行的意思是，当前配置指令运行在“输出过滤器”这个特殊的阶段。

Nginx 配置指令的执行顺序（六）

前面我们在 （五） 中提到，在一个 location 中使用 content 阶段指令时，通常情况下就是对应的 Nginx 模块注册该 location 中的“内容处理程序”。那么当一个 location 中未使用任何 content 阶段的指令，即没有模块注册“内容处理程序”时，content 阶段会发生什么事情呢？谁又来担负起生成内容和输出响应的重担呢？答案就是那些把当前请求的 URI 映射到文件系统的静态资源服务模块。当存在“内容处理程序”时，这些静态资源服务模块并不会起作用；反之，请求的处理权就会自动落到这些模块上。

Nginx 一般会在 content 阶段安排三个这样的静态资源服务模块（除非你的 Nginx 在构造时显式禁用了这三个模块中的一个或者多个，又或者启用了这种类型的其他模块）。按照它们在 content 阶段的运行顺序，依次是 ngx_index 模块， ngx_autoindex 模块，以及 ngx_static 模块。下面就来逐一介绍一下这三个模块。

ngx_index 和 ngx_autoindex 模块都只会作用于那些 URI 以 / 结尾的请求，例如请求 GET /cats/，而对于不以 / 结尾的请求则会直接忽略，同时把处理权移交给 content 阶段的下一个模块。而 ngx_static 模块则刚好相反，直接忽略那些 URI 以 / 结尾的请求。

ngx_index 模块主要用于在文件系统目录中自动查找指定的首页文件，类似 index.html 和 index.htm 这样的，例如：

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

这样，当用户请求 / 地址时，Nginx 就会自动在 root 配置指令指定的文件系统目录下依次寻找 index.htm 和 index.html 这两个文件。如果 index.htm 文件存在，则直接发起“内部跳转”到 /index.htm 这个新的地址；而如果 index.htm 文件不存在，则继续检查 index.html 是否存在。如果存在，同样发起“内部跳转”到 /index.html；如果 index.html 文件仍然不存在，则放弃处理权给 content 阶段的下一个模块。

我们前面已经在 Nginx 变量漫谈（二） 中提到， echo_exec 指令和 rewrite 指令可以发起“内部跳转”。这种跳转会自动修改当前请求的 URI，并且重新匹配与之对应的 location 配置块，再重新执行 rewrite、access、content 等处理阶段。因为是“内部跳转”，所以有别于 HTTP 协议中定义的基于 302 和 301 响应的“外部跳转”，最终用户的浏览器的地址栏也不会发生变化，依然是原来的 URI 位置。而 ngx_index 模块一旦找到了 index 指令中列举的文件之后，就会发起这样的“内部跳转”，仿佛用户是直接请求的这个文件所对应的 URI 一样。

为了进一步确认 ngx_index 模块在找到文件时的“内部跳转”行为，我们不妨设计下面这个小例子：

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

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

此时我们在本机的 /var/www/ 目录下创建一个空白的 index.html 文件，并确保该文件的权限设置对于运行 Nginx worker 进程的帐户可读。然后我们来请求一下根位置（/）：

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

这里发生了什么？为什么输出不是 index.html 文件的内容（即空白）？首先对于用户的原始请求 GET /，Nginx 匹配出 location / 来处理它，然后 content 阶段的 ngx_index 模块在 /var/www/ 下找到了 index.html，于是立即发起一个到 /index.html 位置的“内部跳转”。

到这里，相信大家都不会有问题。接下来有趣的事情发生了！在重新为 /index.html 这个新位置匹配 location 配置块时，location /index.html 的优先级要高于 location /，因为 location 块按照 URI 前缀来匹配时遵循所谓的“最长子串匹配语义”。这样，在进入 location /index.html 配置块之后，又重新开始执行 rewrite 、access、以及 content 等阶段。最终输出 a = 32 自然也就在情理之中了。

我们接着研究上面这个例子。如果此时把 /var/www/index.html 文件删除，再访问 / 又会发生什么事情呢？答案是返回 403 Forbidden 出错页。为什么呢？因为 ngx_index 模块找不到 index 指令指定的文件（在这里就是 index.html），接着把处理权转给 content 阶段的后续模块，而后续的模块也都无法处理这个请求，于是 Nginx 只好放弃，输出了错误页，并且在 Nginx 错误日志中留下了类似这一行信息：

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

所谓 directory index 便是生成“目录索引”的意思，典型的方式就是生成一个网页，上面列举出 /var/www/ 目录下的所有文件和子目录。而运行在 ngx_index 模块之后的 ngx_autoindex 模块就可以用于自动生成这样的“目录索引”网页。我们来把上例修改一下：

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

此时仍然保持文件系统中的 /var/www/index.html 文件不存在。我们再访问 / 位置时，就会得到一张漂亮的网页：

```
$ 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>
```

生成的 HTML 源码显示，我本机的 /var/www/ 目录下还有 cgi-bin/, error/, htdocs/, 以及 icons/ 这几个子目录。在你的系统中尝试上面的例子，输出很可能会不太一样。

值得一提的是，当你的文件系统中存在 /var/www/index.html 时，优先运行的 ngx_index 模块就会发起“内部跳转”，根本轮不到 ngx_autoindex 执行。感兴趣的读者可以自己测试一下。

在 content 阶段默认“垫底”的最后一个模块便是极为常用的 ngx_static 模块。这个模块主要实现服务静态文件的功能。比方说，一个网站的静态资源，包括静态 .html 文件、静态 .css 文件、静态 .js 文件、以及静态图片文件等等，全部可以通过这个模块对外服务。前面介绍的 ngx_index 模块虽然可以在指定的首页文件存在时发起“内部跳转”，但真正把相应的首页文件服务出去（即把该文件的内容作为响应体数据输出，并设置相应的响应头），还是得靠这个 ngx_static 模块来完成。

Nginx 配置指令的执行顺序（七）

来看一个 ngx_static 模块服务磁盘文件的例子。我们使用下面这个配置片段：

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

同时在本机的 /var/www/ 目录下创建两个文件，一个文件叫做 index.html，内容是一行文本 this is my home；另一个文件叫做 hello.html，内容是一行文本 hello world. 同时注意这两个文件的权限设置，确保它们都对运行 Nginx worker 进程的系统帐户可读。

现在来通过 HTTP 协议请求一下这两个文件所对应的 URI：

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

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

我们看到，先前创建的那两个磁盘文件的内容被分别输出了。

不妨来分析一下这里发生的事情：`location /` 中没有使用运行在 `content` 阶段的模块指令，于是也就没有模块注册这个 `location` 的“内容处理程序”，处理权便自动落到了在 `content` 阶段“垫底”的那 3 个静态资源服务模块。首先运行的 ngx_index 和 ngx_autoindex 模块先后看到当前请求的 URI，`/index.html` 和 `/hello.html`，并不以 `/` 结尾，于是直接弃权，将处理权转给了最后运行的 `ngx_static` 模块。`ngx_static` 模块根据 root 指令指定的“文档根目录”（document root），分别将请求 URI `/index.html` 和 `/hello.html` 映射为文件系统路径 `/var/www/index.html` 和 `/var/www/hello.html`，在确认这两个文件存在后，将它们的内容分别作为响应体输出，并自动设置 `Content-Type`、`Content-Length` 以及 `Last-Modified` 等响应头。

为了确认 `ngx_static` 模块确实运行了，可以启用 （一） 中介绍过的 Nginx “调试日志”，然后再次请求 `/index.html` 这个接口。此时，在 Nginx 错误日志文件中可以看到类似下面这一行的调试信息：

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

这一行信息便是 `ngx_static` 模块生成的，其含义是“正在输出的静态文件的描述符是数字 8”。当然，具体的文件描述符编号会经常发生变化，这里只是我机器的一次典型输出。值得一提的是，能生成这一行调试信息的还有标准模块 ngx_gzip_static ，但它默认是不启用的，后面会专门介绍到这个模块。

注意上面这个例子中使用的 root 配置指令只起到了声明“文档根目录”的作用，并不是它开启了 `ngx_static` 模块。`ngx_static` 模块总是处于开启状态，但是否轮得到它运行就要看 `content` 阶段先于它运行的那些模块是否“弃权”了。为了进一步确认这一点，来看下面这个空白 `location` 的定义：

```
location / {
}
```

因为没有配置 root 指令，所以在访问这个接口时，Nginx 会自动计算出一个缺省的“文档根目录”。该缺省值是取所谓的“配置前缀”（configure prefix）路径下的 `html/` 子目录。举一个例子，假设“配置前缀”是 `/foo/bar/`，则缺省的“文档根目录”便是 `/foo/bar/html/`.

那么“配置前缀”是由什么来决定的呢？默认情况下，就是 Nginx 安装时的根目录（或者说 Nginx 构造时传递给 `./configure` 脚本的 `--prefix` 选项的路径值）。如果 Nginx 安装到了 `/usr/local/nginx/` 下，则“配置前缀”便是 `/usr/local/nginx/`，同时默认的“文档根目录”便是 `/usr/local/nginx/html/`. 不过，我们也可以在启动 Nginx 的时候，通过 `--prefix` 命令行选项临时指定自己的“配置前缀”路径。假设我们启动 Nginx 时使用的命令是

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

则对于该服务器实例，其“配置前缀”便是 `/home/agentzh/test/`，而默认的“文档根目录”便是 `/home/agentzh/test/html/`. “配置前缀”不仅会决定默认的“文档根目录”，还决定着 Nginx 配置文件中许多相对路径值如何解释为绝对路径，后面我们还会看到许多需要引用到“配置前缀”的例子。

获取当前“文档根目录”的路径有一个非常简便的方法，那就是请求一个肯定不存在的文件所对应的资源名，例如：

```
$ 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>
```

我们会很自然地得到 `404` 错误页。此时再看 Nginx 错误日志文件，应该会看到类似下面这一行错误消息：

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

这条错误消息是 `ngx_static` 模块打印出来的，因为它并不能在文件系统的对应路径上找到名为 `blah-blah.txt` 的文件。因为这条错误信息中包含有 `ngx_static` 试图打开的文件的绝对路径，所以从这个路径不难看出，当前的“文档根目录”是 `/home/agentzh/test/html/`.

很多初学者会想当然地把 `404` 错误理解为某个 `location` 不存在，其实上面这个例子表明，即使 `location` 存在并成功匹配，也是可能返回 `404` 错误页的。因为决定着 `404` 错误页的是抽象的“资源”是否存在，而非某个具体的 `location` 是否存在。

初学者常犯的一个错误是忘记配置 `content` 阶段的模块指令，而他们自己其实并不期望使用 `content` 阶段缺省运行的静态资源服务，例如：

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

显然，这个 `/auth` 接口只定义了 `access` 阶段的配置指令，即 access_by_lua，并未定义任何 `content` 阶段的配置指令。于是当我们请求 `/auth` 接口时，在 `access` 阶段的 Lua 代码会如期执行，然后 `content` 阶段的那些静态文件服务会紧接着自动发生作用，直至 `ngx_static` 模块去文件系统上找名为 `auth` 的文件。而经常地，`404` 错误页会抛出，除非运气太好，在对应路径上确实存在一个叫做 `auth` 的文件。所以，一条经验是，当遇到意外的 `404` 错误并且又不涉及静态文件服务时，应当首先检查是否在对应的 `location` 配置块中恰当地配置了 `content` 阶段的模块指令，例如 content_by_lua、 echo 以及 proxy_pass 之类。当然，Nginx 的 `error.log` 文件一般总是会提供各种意外问题的答案，例如对于上面这个例子，我的 `error.log` 中有下面这条错误信息：

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

Nginx 配置指令的执行顺序（八）

前面我们详细讨论了 rewrite、access 和 content 这三个最为常见的 Nginx 请求处理阶段，在此过程中，也顺便介绍了运行在这三个阶段的众多 Nginx 模块及其配置指令。同时可以看到，请求处理阶段的划分直接影响到了配置指令的执行顺序，熟悉这些阶段对于正确配置不同的 Nginx 模块并实现它们彼此之间的协同工作是非常必要的。所以接下来我们接着讨论余下的那些阶段。

前面在 （一） 中提到，Nginx 处理请求的过程一共划分为 11 个阶段，按照执行顺序依次是 post-read、server-rewrite、find-config、rewrite、post-rewrite、preaccess、access、post-access、try-files、content 以及 log.

最先执行的 post-read 阶段在 Nginx 读取并解析完请求头（request headers）之后就立即开始运行。这个阶段像前面介绍过的 rewrite 阶段那样支持 Nginx 模块注册处理程序。比如标准模块 ngx_realip 就在 post-read 阶段注册了处理程序，它的功能是迫使 Nginx 认为当前请求的来源地址是指定的某一个请求头的值。下面这个例子就使用了 ngx_realip 模块提供的 set_real_ip_from 和 real_ip_header 这两条配置指令：

```
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";
    }
}
```

这里的配置是让 Nginx 把那些来自 127.0.0.1 的所有请求的来源地址，都改写为请求头 X-My-IP 所指定的值。同时该例使用了标准内建变量 $remote_addr 来输出当前请求的来源地址，以确认是否被成功改写。

首先在本地请求一下这个 /test 接口：

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

这里使用了 curl 工具的 -H 选项指定了额外的 HTTP 请求头 X-My-IP: 1.2.3.4. 从输出可以看到， $remote_addr 变量的值确实在 rewrite 阶段就已经成为了 X-My-IP 请求头中指定的值，即 1.2.3.4. 那么 Nginx 究竟是在什么时候改写了当前请求的来源地址呢？答案是：在 post-read 阶段。由于 rewrite 阶段的运行远在 post-read 阶段之后，所以当在 location 配置块中通过 set 配置指令读取 $remote_addr 内建变量时，读出的来源地址已经是经过 post-read 阶段篡改过的。

如果在请求上例中的 /test 接口时没有指定 X-My-IP 请求头，或者提供的 X-My-IP 请求头的值不是合法的 IP 地址，那么 Nginx 就不会对来源地址进行改写，例如：

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

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

如果从另一台机器访问这个 /test 接口，那么即使指定了合法的 X-My-IP 请求头，也不会触发 Nginx 对来源地址进行改写。这是因为上例已经使用 set_real_ip_from 指令规定了来源地址的改写操作只对那些来自 127.0.0.1 的请求生效。这种过滤机制可以避免来自其他不受信任的地址的恶意欺骗。当然，也可以通过 set_real_ip_from 指令指定一个 IP 网段（利用 （三） 中介绍过的“CIDR 记法”）。此外，同时配置多个 set_real_ip_from 语句也是允许的，这样可以指定多个受信任的来源地址或地址段。下面是一个例子：

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

有的读者可能会问， ngx_realip 模块究竟有什么实际用途呢？为什么我们需要去改写请求的来源地址呢？答案是：当 Nginx 处理的请求经过了某个 HTTP 代理服务器的转发时，这个模块就变得特别有用。当原始的用户请求经过转发之后，Nginx 接收到的请求的来源地址无一例外地变成了该代理服务器的 IP 地址，于是 Nginx 以及 Nginx 背后的应用就无法知道原始请求的真实来源。所以，一般我们会在 Nginx 之前的代理服务器中把请求的原始来源地址编码进某个特殊的 HTTP 请求头中（例如上例中的 X-My-IP 请求头），然后再在 Nginx 一侧把这个请求头中编码的地址恢复出来。这样 Nginx 中的后续处理阶段（包括 Nginx 背后的各种后端应用）就会认为这些请求直接来自那些原始的地址，代理服务器就仿佛不存在一样。正是因为这个需求，所以 ngx_realip 模块才需要在第一个处理阶段，即 post-read 阶段，注册处理程序，以便尽可能早地改写请求的来源。

post-read 阶段之后便是 server-rewrite 阶段。我们曾在 （二） 中简单提到，当 ngx_rewrite 模块的配置指令直接书写在 server 配置块中时，基本上都是运行在 server-rewrite 阶段。下面就来看这样的一个例子：

```
server {
    listen 8080;

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

    set $a hello;
}
```

这里，配置语句 set $a hello 直接写在了 server 配置块中，因此它就运行在 server-rewrite 阶段。而 server-rewrite 阶段要早于 rewrite 阶段运行，因此写在 location 配置块中的语句 set $b "$a, world" 便晚于外面的 set $a hello 语句运行。该例的测试结果证明了这一点：

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

由于 server-rewrite 阶段位于 post-read 阶段之后，所以 server 配置块中的 set 指令也就总是运行在 ngx_realip 模块改写请求的来源地址之后。来看下面这个例子：

```
    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";
        }
    }
```

请求 /test 接口的结果如下：

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

在这个例子中，虽然 set 指令写在了 ngx_realip 的配置指令之前，但仍然晚于 ngx_realip 模块执行。所以 $addr 变量在 server-rewrite 阶段被 set 指令赋值时，从 $remote_addr 变量读出的来源地址已经是经过改写过的了。

Nginx 配置指令的执行顺序（九）

紧接在 server-rewrite 阶段后边的是 find-config 阶段。这个阶段并不支持 Nginx 模块注册处理程序，而是由 Nginx 核心来完成当前请求与 location 配置块之间的配对工作。换句话说，在此阶段之前，请求并没有与任何 location 配置块相关联。因此，对于运行在 find-config 阶段之前的 post-read 和 server-rewrite 阶段来说，只有 server 配置块以及更外层作用域中的配置指令才会起作用。这就是为什么只有写在 server 配置块中的 ngx_rewrite 模块的指令才会运行在 server-rewrite 阶段，这也是为什么前面所有例子中的 ngx_realip 模块的指令也都特意写在了 server 配置块中，以确保其注册在 post-read 阶段的处理程序能够生效。

当 Nginx 在 find-config 阶段成功匹配了一个 location 配置块后，会立即打印一条调试信息到错误日志文件中。我们来看这样的一个例子：

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

如果启用了 Nginx 的“调试日志”，那么当请求 /hello 接口时，便可以在 error.log 文件中过滤出下面这一行信息：

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

我们有意省略了信息行首的时间戳，以便放在这里。

运行在 find-config 阶段之后的便是我们的老朋友 rewrite 阶段。由于 Nginx 已经在 find-config 阶段完成了当前请求与 location 的配对，所以从 rewrite 阶段开始，location 配置块中的指令便可以产生作用。前面已经介绍过，当 ngx_rewrite 模块的指令用于 location 块中时，便是运行在这个 rewrite 阶段。另外， ngx_set_misc 模块的指令也是如此，还有 ngx_lua 模块的 set_by_lua 指令和 rewrite_by_lua 指令也不例外。

rewrite 阶段再往后便是所谓的 post-rewrite 阶段。这个阶段也像 find-config 阶段那样不接受 Nginx 模块注册处理程序，而是由 Nginx 核心完成 rewrite 阶段所要求的“内部跳转”操作（如果 rewrite 阶段有此要求的话）。先前在 （二） 中已经介绍过了“内部跳转”的概念，同时演示了如何通过 echo_exec 指令或者 rewrite 指令来发起“内部跳转”。由于 echo_exec 指令运行在 content 阶段，与这里讨论的 post-rewrite 阶段无关，于是我们感兴趣的便只剩下运行在 rewrite 阶段的 rewrite 指令。回顾一下 （二） 中演示过的这个例子：

```
    server {
        listen 8080;

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

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

这里在 location /foo 中通过 rewrite 指令把当前请求的 URI 无条件地改写为 /bar，同时发起一个“内部跳转”，最终跳进了 location /bar 中。这里比较有趣的地方是“内部跳转”的工作原理。“内部跳转”本质上其实就是把当前的请求处理阶段强行倒退到 find-config 阶段，以便重新进行请求 URI 与 location 配置块的配对。比如上例中，运行在 rewrite 阶段的 rewrite 指令就让当前请求的处理阶段倒退回了 find-config 阶段。由于此时当前请求的 URI 已经被 rewrite 指令修改为了 /bar，所以这一次换成了 location /bar 与当前请求相关联，然后再接着从 rewrite 阶段往下执行。

不过这里更有趣的地方是，倒退回 find-config 阶段的动作并不是发生在 rewrite 阶段，而是发生在后面的 post-rewrite 阶段。上例中的 rewrite 指令只是简单地指示 Nginx 有必要在 post-rewrite 阶段发起“内部跳转”。这个设计对于 Nginx 初学者来说，或许显得有些古怪：“为什么不直接在 rewrite 指令执行时立即进行跳转呢？”答案其实很简单，那就是为了在最初匹配的 location 块中支持多次反复地改写 URI，例如：

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

        echo foo;
    }

    location /bar {
```

```
    echo bar;
}

location /baz {
    echo baz;
}
```

这里在 location /foo 中连续把当前请求的 URI 改写了两遍：第一遍先无条件地改写为 /bar，第二遍再无条件地改写为 /baz. 而这两条 rewrite 语句只会最终导致 post-rewrite 阶段发生一次“内部跳转”操作，从而不至于在第一次改写 URI 时就直接跳离了当前的 location 而导致后面的 rewrite 语句没有机会执行。请求 /foo 接口的结果证实了这一点：

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

从输出结果可以看到，上例确实成功地从 /foo 一步跳到了 /baz 中。如果启用 Nginx “调试日志”的话，还可以从 find-config 阶段生成的 location 块的匹配信息中进一步证实这一点：

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

我们看到，对于该次请求，Nginx 一共只匹配过 /foo 和 /baz 这两个 location，从而只发生过一次“内部跳转”。

当然，如果在 server 配置块中直接使用 rewrite 配置指令对请求 URI 进行改写，则不会涉及“内部跳转”，因为此时 URI 改写发生在 server-rewrite 阶段，早于执行 location 配对的 find-config 阶段。比如下面这个例子：

```
server {
    listen 8080;

    rewrite ^/foo /bar;

    location /foo {
        echo foo;
    }

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

这里，我们在 server-rewrite 阶段就把那些以 /foo 起始的 URI 改写为 /bar，而此时请求并没有和任何 location 相关联，所以 Nginx 正常往下运行 find-config 阶段，完成最终的 location 匹配。如果我们请求上例中的 /foo 接口，那么 location /foo 根本就没有机会匹配，因为在第一次（也是唯一的一次）运行 find-config 阶段时，当前请求的 URI 已经被改写为 /bar，从而只会匹配 location /bar. 实际请求的输出正是如此：

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

Nginx “调试日志”可以再一次佐证我们的结论：

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

可以看到，Nginx 总共只进行过一次 location 匹配，并无“内部跳转”发生。

Nginx 配置指令的执行顺序（十）

运行在 post-rewrite 阶段之后的是所谓的 preaccess 阶段。该阶段在 access 阶段之前执行，故名 preaccess.

标准模块 ngx_limit_req 和 ngx_limit_zone 就运行在此阶段，前者可以控制请求的访问频度，而后者可以限制访问的并发度。这里我们仅仅和它们打个照面，后面还会有机会专门接触到这两个模块。

前面反复提到的标准模块 ngx_realip 其实也在这个阶段注册了处理程序。有些读者可能会问：“这是为什么呢？它不是已经在 post-read 阶段注册处理程序了吗？”我们不妨通过下面这个例子来揭晓答案：

```
server {
    listen 8080;

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

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

与先看前到的例子相比，此例最重要的区别在于把 ngx_realip 的配置指令放在了 location 配置块中。前面我们介绍过，Nginx 匹配 location 的动作发生在 find-config 阶段，而 find-config 阶段远远晚于 post-read 阶段执行，所以在 post-read 阶段，当前请求还没有和任何 location 相关联。在这个例子中，因为 ngx_realip 的配置指令都写在了 location 配置块中，所以在 post-read 阶段， ngx_realip 模块的处理程序没有看到任何可用的配置信息，便不会执行来源地址的改写工作了。

为了解决这个难题， ngx_realip 模块便又特意在 preaccess 阶段注册了处理程序，这样它才有机会运行 location 块中的配置指令。正是因为这个缘故，上面这个例子的运行结果才符合直觉预期：

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

不幸的是， ngx_realip 模块的这个解决方案还是存在漏洞的，比如下面这个例子：

```
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";
    }
}
```

这里，我们在 rewrite 阶段将 $remote_addr 的值保存到了用户变量 $addr 中，然后再输出。因为 rewrite 阶段先于 preaccess 阶段执行，所以当 ngx_realip 模块尚未在 preaccess 阶段改写来源地址时，最初的来源地址就已经在 rewrite 阶段被读取了。上例的实际请求结果证明了我们的结论：

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

输出的地址确实是未经改写过的。Nginx 的“调试日志”可以进一步确认这一点：

```
$ 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"
```

其中第一行调试信息

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

是 set 语句读取 $remote_addr 变量时产生的。信息中的字符串 "127.0.0.1" 便是 $remote_addr 当时读出来的值。

而第二行调试信息

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

则显示我们对变量 $addr 进行了赋值操作。

后面两行信息

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

是 ngx_realip 模块在 preaccess 阶段改写当前请求的来源地址。我们看到，改写后的新地址确实是期望的 1.2.3.4. 但很明显这个操作发生在 $addr 变量赋值之后，所以已经太迟了。

而最后一行信息

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

则是 echo 配置指令在输出时读取变量 $addr 时产生的，我们看到它的值是改写前的来源地址。

看到这里，有的读者可能会问：“如果 ngx_realip 模块不在 preaccess 阶段注册处理程序，而在 rewrite 阶段注册，那么上例不就可以工作了？”答案是：不一定。因为 ngx_rewrite 模块的处理程序也同样注册在 rewrite 阶段，而前面我们在 （二） 中特别提到，在这种情况下，不同模块之间的执行顺序一般是不确定的，所以 ngx_realip 的处理程序可能仍然在 set 语句之后执行。

一个建议是：尽量在 server 配置块中配置 ngx_realip 这样的模块，以避免上面介绍的这种棘手的例外情况。

运行在 preaccess 阶段之后的则是我们的另一个老朋友，access 阶段。前面我们已经知道了，标准模块 ngx_access、第三方模块 ngx_auth_request 以及第三方模块 ngx_lua 的 access_by_lua 指令就运行在这个阶段。

access 阶段之后便是 post-access 阶段。从这个阶段的名字，我们也能一眼看出它是紧跟在 access 阶段后面执行的。这个阶段也和 post-rewrite 阶段类似，并不支持 Nginx 模块注册处理程序，而是由 Nginx 核心自己完成一些处理工作。post-access 阶段主要用于配合 access 阶段实现标准 ngx_http_core 模块提供的配置指令 satisfy 的功能。

对于多个 Nginx 模块注册在 access 阶段的处理程序， satisfy 配置指令可以用于控制它们彼此之间的协作方式。比如模块 A 和 B 都在 access 阶段注册了与访问控制相关的处理程序，那就有两种协作方式，一是模块 A 和模块 B 都得通过验证才算通过，二是模块 A 和模块 B 只要其中任一个通过验证就算通过。第一种协作方式称为 all 方式（或者说“与关系”），第二种方式则被称为 any 方式（或者说“或关系”）。默认情况下，Nginx 使用的是 all 方式。下面是一个例子：

```
location /test {
    satisfy all;

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

    echo something important;
}
```

这里，我们在 /test 接口中同时配置了 ngx_access 模块和 ngx_lua 模块，这样 access 阶段就由这两个模块一起来做检验工作。其中，语句 deny all 会让 ngx_access 模块的处理程序总是拒绝当前请求，而语句 access_by_lua 'ngx.exit(ngx.OK)' 则总是允许访问。当我们通过 satisfy 指令配置了 all 方式时，就需要 access 阶段的所有模块都通过验证，但不幸的是，这里 ngx_access 模块总是会拒绝访问，所以整个请求就会被拒：

```
$ 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>
```

细心的读者会在 Nginx 错误日志文件中看到类似下面这一行的出错信息：

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

然而，如果我们把上例中的 satisfy all 语句更改为 satisfy any，

```
location /test {
    satisfy any;

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

    echo something important;
}
```

结果则会完全不同：

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

即请求反而最终通过了验证。这是因为在 any 方式下，access 阶段只要有一个模块通过了验证，就会认为请求整体通过了验证，而在上例中， ngx_lua 模块的 access_by_lua 语句总是会通过验证的。

在配置了 satisfy any 的情况下，只有当 access 阶段的所有模块的处理程序都拒绝访问时，整个请求才会被拒，例如：

```
location /test {
    satisfy any;

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

    echo something important;
}
```

此时访问 /test 接口才会得到 403 Forbidden 错误页。这里，post-access 阶段参与了 access 阶段各模块处理程序的“或关系”的实现。

值得一提的是，上面这几个的例子需要 ngx_lua 0.5.0rc19 或以上版本；之前的版本是不能和 satisfy any 配置语句一起工作的。

Nginx 配置指令的执行顺序（十一）

紧跟在 post-access 阶段之后的是 try-files 阶段。这个阶段专门用于实现标准配置指令 try_files 的功能，并不支持 Nginx 模块注册处理程序。由于 try_files 指令在许多 FastCGI 应用的配置中都有用到，所以我们不妨在这里简单介绍一下。

try_files 指令接受两个以上任意数量的参数，每个参数都指定了一个 URI. 这里假设配置了 N 个参数，则 Nginx 会在 try-files 阶段，依次把前 N-1 个参数映射为文件系统上的对象（文件或者目录），然后检查这些对象是否存在。一旦 Nginx 发现某个文件系统对象存在，就会在 try-files 阶段把当前请求的 URI 改写为该对象所对应的参数 URI（但不会包含末尾的斜杠字符，也不会发生“内部跳转”）。如果前 N-1 个参数所对应的文件系统对象都不存在，try-files 阶段就会立即发起“内部跳转”到最后一个参数（即第 N 个参数）所指定的 URI.

前面在 （六） 和 （七） 中已经看到静态资源服务模块会把当前请求的 URI 映射到文件系统，通过 root 配置指令所指定的“文档根目录”进行映射。例如，当“文档根目录”是 /var/www/ 的时候，请求 URI /foo/bar 会被映射为文件 /var/www/foo/bar，而请求 URI /foo/baz/ 则会被映射为目录 /var/www/foo/baz/. 注意这里是如何通过 URI 末尾的斜杠字符是否存在来区分“目录”和“文件”的。我们正在讨论的 try_files 配置指令使用同样的规则来完成其各个参数 URI 到文件系统对象的映射。

不妨来看下面这个例子：

```
root /var/www/;

location /test {
    try_files /foo /bar/ /baz;
    echo "uri: $uri";
}

location /foo {
    echo foo;
}

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

```
location /baz {
    echo baz;
}
```

这里通过 root 指令把“文档根目录”配置为 /var/www/，如果你系统中的 /var/www/ 路径下存放有重要数据，则可以把它替换为其他任意路径，但此路径对运行 Nginx worker 进程的系统帐号至少有可读权限。我们在 location /test 中使用了 try_files 配置指令，并提供了三个参数，/foo、/bar/ 和 /baz. 根据前面对 try_files 指令的介绍，我们可以知道，它会在 try-files 阶段依次检查前两个参数 /foo 和 /bar/ 所对应的文件系统对象是否存在。

不妨先来做一组实验。假设现在 /var/www/ 路径下是空的，则第一个参数 /foo 映射成的文件 /var/www/foo 是不存在的；同样，对于第二个参数 /bar/ 所映射成的目录 /var/www/bar/ 也是不存在的。于是此时 Nginx 会在 try-files 阶段发起到最后一个参数所指定的 URI（即 /baz）的“内部跳转”。实际的请求结果证实了这一点：

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

显然，该请求最终和 location /baz 绑定在一起，执行了输出 baz 字符串的工作。上例中定义的 location /foo 和 location /bar/ 完全不会参与这里的运行过程，因为对于 try_files 的前 N-1 个参数，Nginx 只会检查文件系统，而不会去执行 URI 与 location 之间的匹配。

对于上面这个请求，Nginx 会产生类似下面这样的“调试日志”：

```
$ grep trying logs/error.log
[debug] 3869#0: *1 trying to use file: "/foo" "/var/www/foo"
[debug] 3869#0: *1 trying to use dir: "/bar" "/var/www/bar"
[debug] 3869#0: *1 trying to use file: "/baz" "/var/www/baz"
```

通过这些信息可以清楚地看到 try-files 阶段发生的事情：Nginx 依次检查了文件 /var/www/foo 和目录 /var/www/bar，末了又处理了最后一个参数 /baz. 这里最后一条“调试信息”容易产生误解，会让人误以为 Nginx 也把最后一个参数 /baz 给映射成了文件系统对象进行检查，事实并非如此。当 try_files 指令处理到它的最后一个参数时，总是直接执行“内部跳转”，而不论其对应的文件系统对象是否存在。

接下来再做一组实验：在 /var/www/ 下创建一个名为 foo 的文件，其内容为 hello world（注意你需要有 /var/www/ 目录下的写权限）：

```
$ echo 'hello world' > /var/www/foo
```

然后再请求 /test 接口：

```
$ curl localhost:8080/test
uri: /foo
```

这里发生了什么？我们来看， try_files 指令的第一个参数 /foo 可以映射为文件 /var/www/foo，而 Nginx 在 try-files 阶段发现此文件确实存在，于是立即把当前请求的 URI 改写为这个参数的值，即 /foo，并且不再继续检查后面的参数，而直接运行后面的请求处理阶段。

上面这个请求在 try-files 阶段所产生的“调试日志”如下：

```
$ grep trying logs/error.log
[debug] 4132#0: *1 trying to use file: "/foo" "/var/www/foo"
```

显然，在 try-files 阶段，Nginx 确实只检查和处理了 /foo 这一个参数，而后面的参数都被“短路”掉了。

类似地，假设我们删除刚才创建的 /var/www/foo 文件，而在 /var/www/ 下创建一个名为 bar 的子目录：

```
$ mkdir /var/www/bar
```

则请求 /test 的结果也是类似的：

```
$ curl localhost:8080/test
uri: /bar
```

在这种情况下，Nginx 在 try-files 阶段发现第一个参数 /foo 对应的文件不存在，就会转向检查第二个参数对应的文件系统对象（在这里便是目录 /var/www/bar/）。由于此目录存在，Nginx 就会把当前请求的 URI 改写为第二个参数的值，即 /bar（注意，原始参数值是 /bar/，但 try_files 会自动去除末尾的斜杠字符）。

这一组实验所产生的“调试日志”如下：

```
$ grep trying logs/error.log
[debug] 4223#0: *1 trying to use file: "/foo" "/var/www/foo"
[debug] 4223#0: *1 trying to use dir: "/bar" "/var/www/bar"
```

我们看到， try_files 指令在这里只检查和处理了它的前两个参数。

通过前面这几组实验不难看到， try_files 指令本质上只是有条件地改写当前请求的 URI，而这里说的“条件”其实就是文件系统上的对象是否存在。当“条件”都不满足时，它就会无条件地发起一个指定的“内部跳转”。当然，除了无条件地发起“内部跳转”之外， try_files 指令还支持直接返回指定状态码的 HTTP 错误页，例如：

```
try_files /foo /bar/ =404;
```

这行配置是说，当 /foo 和 /bar/ 参数所对应的文件系统对象都不存在时，就直接返回 404 Not Found 错误页。注意这里它是如何使用等号字符前缀来标识 HTTP 状态码的。