浅析Nginx配置文件中的变量的编写使用

更新时间：2016年01月06日 15:15:53 作者：agentzh

这篇文章主要介绍了Nginx配置文件中的变量的编写使用,包括从常用的rewrite等方面来深入变量的相关定义,需要的朋友可以参考下

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。

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。
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）。

Nginx 变量