

## 加餐 | 如何做HTTP服务的测试？

2019-10-04 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 12:29 大小 11.45M



你好，我是七牛云许式伟。

基于 HTTP 协议提供服务的好处是显然的。除了 HTTP 服务有很多现成的客户端、服务端框架可以直接使用外，在 HTTP 服务的调试、测试、监控、负载均衡等领域都有现成的相关工具支撑。

在七牛，我们绝大部分的服务，包括内部服务，都是基于 HTTP 协议来提供服务。所以我们需要思考如何更有效地进行 HTTP 服务的测试。

七牛早期 HTTP 服务的测试方法很朴素：第一步先写好服务端，然后写一个客户端 SDK，再基于这个客户端 SDK 写测试案例。

这种方法多多少少会遇到一些问题。首先，客户端 SDK 的修改可能会导致测试案例编不过。其次，客户端 SDK 通常是使用方友好，而不是测试方友好。服务端开发过程和客户端 SDK 的耦合容易过早地陷入“客户端 SDK 如何抽象更合理”的细节，而不能专注于测试服务逻辑本身。

我的核心诉求是对服务端开发过程和客户端开发过程进行解耦。在网络协议定好了以后，整个系统原则上就可以编写测试案例，而不用等客户端 SDK 的成熟。

不写客户端 SDK 而直接做 HTTP 测试，一个直观的思路是直接基于 `http.Client` 类来写测试案例。这种方式的问题是代码比较冗长，而且它的业务逻辑表达不直观，很难一眼就看出这句话想干什么。虽然可以写一些辅助函数来改观，但做多了就会逐渐有写测试专用 SDK 的倾向。这种写法看起来也不是很可取，毕竟为测试写一个专门的 SDK，看起来成本有些高了。

七牛当前的做法是引入一种 `httptest` DSL 文法。这是七牛为 HTTP 测试而写的领域专用语言。这个 `httptest` 工具当前已经开源，项目主页为：

<https://github.com/qiniu/httptest> (`httptest` 框架)

<https://github.com/qiniu/qiniutest> (支持七牛帐号与授权机制的 `qiniutest` 工具)

## httptest 基础文法

这个语言的文法大概在 2012 年就已经被加入到七牛的代码库，后来有个同事根据这个 DSL 文法写了第一版本 `qiniutest` 程序。在决定推广用这个 DSL 来进行测试的过程中，我们对 DSL 不断地进行了调整和加强。虽然总体思路没有变化，但最终定稿的 DSL 与最初版本有较大的差异。目前来说，我已经可以十分确定地说，这个 DSL 可以满足 90% 以上的测试需求。它被推荐做为七牛内部的首选测试方案。



```
hello.qtf  quick_start.qtf  x
1  #!/usr/bin/env qiniutest
2
3  #
4  # 这个例子算 qiniu httptest 工具的 Hello world 程序吧。
5  # 执行预期：下载 www.qiniu.com 首页，要求返回 200。
6  # 如果返回非 200，测试失败；否则测试通过，并打印返回的 response body（对于测试来说无价值，通常用于调试）。
7  #
8
9  get http://www.qiniu.com/
10 ret 200
11 echo ${resp.body}
12
```

上图是这套 DSL 的 “hello world” 程序。它的执行预期是：下载 [www.qiniu.com](http://www.qiniu.com) 首页，要求返回的 HTTP 状态码为 200。如果返回非 200，测试失败；否则测试通过，输出返回包的正文内容（resp.body 变量）。输出 resp.body 的内容通常是调试需要，而不是测试需要。自动化测试是不需要向屏幕去输出什么的。

```
hello.qtf quick_start.qtf
1  #!/usr/bin/env qiniutest
2
3  #
4  # qiniutest 整体基于命令行文法。其中 `...` 代表子命令，`...` 或 `""` 方便传递复杂参数。
5  # 以下是单HTTP请求的测试文法。如果你有 HTTP 协议的基础，理解这样一段测试代码所代表的含义并不困难：
6  #
7  # req <http-method> <url>
8  # header <key1> <val11> <val12> ...
9  # header <key2> <val21> <val22> ...
10 # auth <authorization>
11 # body <content-type> <body-data>
12 # ret <expected-status-code>
13 # header <resp-key1> <expected-val11> <expected-val12> ...
14 # header <resp-key2> <expected-val21> <expected-val12> ...
15 # body <expected-content-type> <expected-body-data>
16 #
17 # 上面的 req 和 body 指令，有诸多简写形式。比如：
18 #
19 # req GET http://www.qiniu.com/ 可以简写为：get http://www.qiniu.com/
20 # body 'application/json' '{"a": 1, "b": 2}' 可以简写为：json '{"a": 1, "b": 2}'
21 #
22
23 auth qiniutest `qiniu <AccessKey> <SecretKey>` #预先给auth取个别名，只是为了让下面写auth语句可以更简洁一些
24
25 post http://foo.com/objects
26 auth qiniutest #等价于：auth `qiniu <AccessKey> <SecretKey>`
27 json '{
28     "a": "value1", "b": 1
29 }'
30 ret 200
31 json '{
32     "id": ${id1} #重要！暂时先体会下，后面有详细的机制解析
33 }'
34
35 get http://foo.com/objects/${id1}
36 auth qiniutest
37 ret 200
38 json '{
39     "a": "value1", "b": 1
40 }'
41
```


我们再看该 DSL 的一个 “quick start（快速入门）” 样例。以 # 开始的内容是程序的注释部分。这里有一个很长很长的注释，描述了一个基本的 HTTP 请求测试的构成。后面我们会对这部分内容进行详细展开，这里暂时跳过。

这段代码的第一句话是定义了一个 auth 别名叫 qiniutest，这只是为了让后面具体的 HTTP 请求中授权语句更简短。紧接着是发起一个 POST 请求，创建一个内容为 { “a” : “value1” , “b” : 1} 的对象，并将返回的对象 id 赋值给一个名为 id1 的变量。后面我们会详细解释这个赋值过程是如何进行的。

接着我们发起一个获取对象内容的 GET 请求，需要注意的是 GET 的 URL 中引用了 id1 变量的值，这意味着我们不是要取别的对象的内容，而是取刚刚创建成功的对象的内容，并且

我们期望返回的对象内容和刚才 POST 上去的一样，也是 { "a" : "value1" , "b" : 1}。这就是一个最基础的 HTTP 测试，它创建了一个对象，确认创建成功，并且尝试去取回这个对象，确认内容与我们期望的一致。这里上下两个请求是通过 id1 这个变量来建立关联的。

对这套 DSL 语法有了一个大概的印象后，我们开始来解剖它。先来看看它的语法结构。首先这套 httptest DSL 基于命令行文法：

 复制代码

```
1 command switch1 switch2 ... arg1 arg2 ...
```

整个命令行先是一个命令，然后紧接着是一个个开关（可选），最后是一个个的命令参数。和大家熟悉的命令行比如 Linux Shell 一样，它也会有一些参数需要转义，如果参数包含空格或其他特殊字符，则可以用 \ 前缀来进行转义。比如 ‘\ ’ 表示 ‘ ’ （空格）， ‘\t’ 表示 TAB 等。另外，我们也支持用 ‘...’ 或者 “...” 去传递一个参数，比如 json 格式的多行文本。同 Linux Shell 类似， ‘...’ 里面的内容没有转义， ‘\ ’ 就是 ‘\ ’ ， ‘\t’ 就是 ‘\t’ ，而不是 TAB。而 “...” 则支持转义。

和 Linux Shell 不同的是，我们的 httptest DSL 虽然基于命令行文法，但是它的每一个参数都是有类型的，也就是说这个语言有类型系统，而不像 Linux Shell 命令行参数只有字符串。我们的 httptest DSL 支持且仅支持所有 json 支持的数据类型，包括：

string（如： "a" 、 application/json 等，在不引起歧义的情况下，可以省略双引号）

number（如： 3.14159）


boolean（如： true、 false）

array（如： [ "a" , 200, { "b" : 2} ]）

object/dictionary（如： { "a" : 1, "b" : 2}）

另外，我们的 httptest DSL 也有子命令的概念，它相当于一个函数，可以返回任意类型的数据。比如 qiniu f2weae23e6c9fjg35fae526kbce 返回一个 auth object，这是用常规字符串无法表达的。

理解了 httptest DSL 后，我们来看看如何表达一个 HTTP 请求。它的基本形式如下：

 复制代码

```
1 req <http-method> <url>
2 header <key1> <val11> <val12>
3 header <key2> <val21> <val22>
4 auth <authorization>
5 body <content-type> <body-data>
```

第一句是 req 指令，带两个参数：一个是 http method，即 HTTP 请求的方法，如 GET、POST 等。另一个是要请求的 URL。


接着是一个个自定义的 header（可选），每个 header 指令后面跟一个 key（键）和一个或多个 value（值）。

然后是一个可选的 auth 指令，用来指示这个请求的授权方式。如果没有 auth 语句，那么这个 HTTP 请求是匿名的，否则这就是一个带授权的请求。

最后一句是 body 指令，顾名思义它用来指定 HTTP 请求的正文。body 指令也有两个参数，一个是 content-type（内容格式），另一个是 body-data（请求正文）。

这样说比较抽象，我们看下实际的例子：

无授权的 GET 请求：

 复制代码

```
1 req GET http://www.qiniu.com/
```

带授权的 POST 请求：


 复制代码

```
1 req POST http://foo.com/objects
2 auth `qiniu f2weae23e6c9fjg35fae526kbce`
3 body application/json '{
4   "a": "hello1",
```

```
5   "b":2
6 }'
```


也可以简写成：

无授权的 GET 请求：

 复制代码

```
1 get http://www.qiniu.com/
```

带授权的 Post 请求：

 复制代码

```
1 post http://foo.com/objects
2 auth `qiniu f2weae23e6c9fjg35fae526kbce`
3 json '{
4   "a": "hello1",
5   "b":2
6 }'
```

发起了 HTTP 请求后，我们就可以收到 HTTP 返回包并对内容进行匹配。HTTP 返回包匹配的基本形式如下：


 复制代码

```
1 ret <expected-status-code>
2 header <key1> <expected-val11><expected-val12>
3 header <key2> <expected-val21><expected-val22>
4 body <expected-content-type><expected-body-data>
```

我们先看 ret 指令。实际上，请求发出去的时间是在 ret 指令执行的时候。前面 req、header、auth、body 指令仅仅表达了 HTTP 请求。如果没有调用 ret 指令，那么系统什么也不会发生。




ret 指令可以不带参数。不带参数的 ret 指令，其含义是发起 HTTP 请求，并将返回的 HTTP 返回包解析并存储到 resp 的变量中。而对于带参数的 ret 指令：

 复制代码

```
1 ret <expected-status-code>
```

它等价于：

 复制代码

```
1 ret
2 match <expected-status-code> $(resp.code)
```

## match 指令

这里我们引入了一个新的指令：match 指令。

- 这几乎是这套 DSL 中最核心的概念
  - match <expected> <source>
    - 要求 <expected> 必须和 <source> **匹配**
    - <source> 中不允许出现未绑定的变量
    - <expected> 中允许存在未绑定的变量
      - 如果 <expected> 中出现了已绑定的变量，则要求该变量必须匹配 <source> 中对应的值
      - 如果 <expected> 中出现了未绑定的变量，则该变量会被赋值为 <source> 中对应的值
  - **匹配**
    - 对于 number/string/boolean/array 类型
      - match A B 意味着要求 A == B
    - 对于 object(dictionary) 类型
      - match A B 意味着 A 中出现的 item，在 B 中必须出现并且匹配

七牛所有 HTTP 返回包匹配的匹配文法，都可以用这个 match 来表达：

- `ret <expected-status-code>`
  - 等价于

```
ret
match <expected-status-code> $(resp.code)
```
- `header <key> <expected-val1> <expected-val2> ...`
  - 等价于

```
match ' [<expected-val1>, <expected-val2>, ...]' $(resp.header.<key>)
```
- `body <expected-content-type> <expected-body-data>`
  - 等价于

```
match ' [<expected-content-type>]' $(resp.header.Content-Type)
match <expected-body-data> $(resp.body)
```

所以本质上来说，我们只需要一个不带参数的 `ret`，加上 `match` 指令，就可以搞定所有的返回包匹配过程。这也是我们为什么说 `match` 指令是这套 DSL 中最核心的概念的原因。


和其他自动化测试框架类似，这套 DSL 也提供了断言文法。它类似于 CppUnit 或 JUnit 之类的测试框架提供 `assertEqual`。具体如下：

 复制代码

```
1 equal <expected> <source>
```

与 `match` 不同，这里 `<expected>`、`<source>` 中都不允许出现未绑定的变量。

与 `match` 不同，`equal` 要求 `<expected>`、`<source>` 的值精确相等。

 复制代码

```
1 equalSet <expected> <source>
```

这里 SET 是指集合的意思。

与 `equal` 不同，`equalSet` 要求 `<expected>`、`<source>` 都是 array，并且对 array 的元素进行排序后判断两者是否精确相等。



equalSet 的典型使用场景是测试 list 类的 API，比如列出一个目录下的所有文件，你可能预期这个目录下有哪些文件，但是不能预期他们会以什么样的次序返回。


以上介绍基本上就是这套 DSL 最核心的内容了。内容非常精简，但满足了绝大部分测试场景的需求。

## 测试环境的参数化

下面我们谈谈最后一个话题：测试环境的参数化。

为了让测试案例更加通用，我们需要对测试依赖的环境进行参数化。比如，为了让测试脚本能够同时用于 stage 环境和 product 环境，我们需要把服务的 Host 信息参数化。另外，为了方便测试脚本入口，我们通常还需要把 用户名 / 密码、AK/SK 等敏感性信息参数化，避免直接硬编码到测试案例中。

为了把服务器的 Host 信息（也就是服务器的位置）参数化，我们引入了 host 指令。例如：

 复制代码

```
1 host foo.com 127.0.0.1:8888
2 get http://foo.com/objects/a325gea2kgfd
3 auth qiniutest
4 ret 200
5 json '{
6   "a": "hello1",
7   "b": 2
8 }'
```

这样，后文所有出现请求 foo.com 地方，都会把请求发送到 127.0.0.1:8888 这样一个服务器地址。要想让脚本测试另外的服务器实例，我们只需要调整 host 语句，将 127.0.0.1:8888 调整成其他即可。

除了服务器 Host 需要参数化外，其他常见的参数化需求是 用户名 / 密码、AK/SK 等。AK/SK 这样的信息非常敏感，如果在测试脚本里面硬编码这些信息，将不利于测试脚本代码的入库。一个典型的测试环境参数化后的测试脚本样例如下：

```
match $(testenv) `env QiniuTestEnv`  
match $(env) `envdecode QiniuTestEnv_$(testenv)`
```

```
host foo.com $(env.FooHost)  
auth qiniutest `qiniu $(env.AK) $(env.SK)`
```

```
post http://foo.com/objects  
auth qiniutest  
json '{"a": "hello1", "b": 2}'  
ret 200  
json '{"id": $(id1)}'
```

```
get http://foo.com/objects/\$\(id1\)  
auth qiniutest  
ret 200  
json '{"a": "hello1", "b": 2}'
```

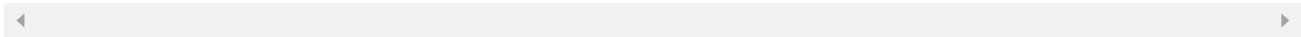
其中，`env` 指令用于取环境变量对应的值（返回值类型是 `string`），`envdecode` 指令则是先取得环境变量对应的值，然后对值进行 `json decode` 得到相应的 `object/dictionary`。有了

`(env)` 这个对象（*object*），就可以通过它获得各种测试环境参数，比如 `(env.FooHost)`、`(env.AK)`、`(env.SK)` 等。

写好了测试脚本后，在执行测试脚本之前，我们需要先配置测试环境：


 复制代码

```
1 export QiniuTestEnv_stage='{  
2   "FooHost": "192.168.1.10:8888",  
3   "AK": "...",  
4   "SK": "..."  
5 }'  
6  
7 export QiniuTestEnv_product='{  
8   "FooHost": "foo.com",  
9   "AK": "...",  
10  "SK": "..."  
11 }'
```

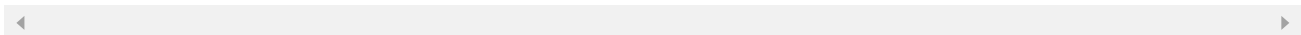


这样我们就可以执行测试脚本了：


测试 stage 环境：

 复制代码

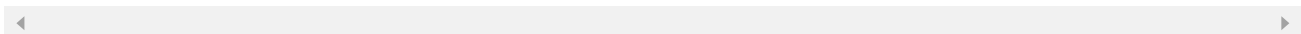
```
1 QiniuTestEnv=stage qiniutest ./testfoo.qtf
```



测试 product 环境：

 复制代码

```
1 QiniuTestEnv=product qiniutest ./testfoo.qtf
```



## 结语

测试是软件质量保障至关重要的一环。一个好的测试工具对提高开发效率的作用巨大。如果能够让开发人员的开发时间从一小时减少到半小时，那么日积月累就会得到惊人的效果。

去关注开发人员日常工作过程中的不爽和低效率是非常有必要的。任何开发效率提升相关的工作，其收益都是指数级的。这也是我们所推崇的做事风格。如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。


如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

# 许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟  
七牛云 CEO



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 46 | 服务端开发篇：回顾与总结

## 精选留言 (2)

写留言



debugtalk

2019-10-04

之前我做的一个开源项目 HttpRunner 和这个倒有些相似之处。

<https://github.com/httprunner/httprunner>

对于DSL，有个痛点就是没法像代码那样进行单步调试。所以我也在HttpRunner中探索了语法提示和自动补全功能，以及和 python代码的互转

展开 ▾

作者回复: 



2



mickey

2019-10-05

如果后端是连接数据库的，怎样做动态的自动化测试呢？

作者回复: 这个不影响，只需要在测试前数据库是在运行中的就行

