

Web开发

747次阅读

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

1. 静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；
2. CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。
3. ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。
4. MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

Python的诞生历史比Web还要早，由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。

Python有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

本章我们会详细讨论Python Web开发技术。

HTTP协议简介

1068次阅读

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器的传输协议是HTTP，所以：

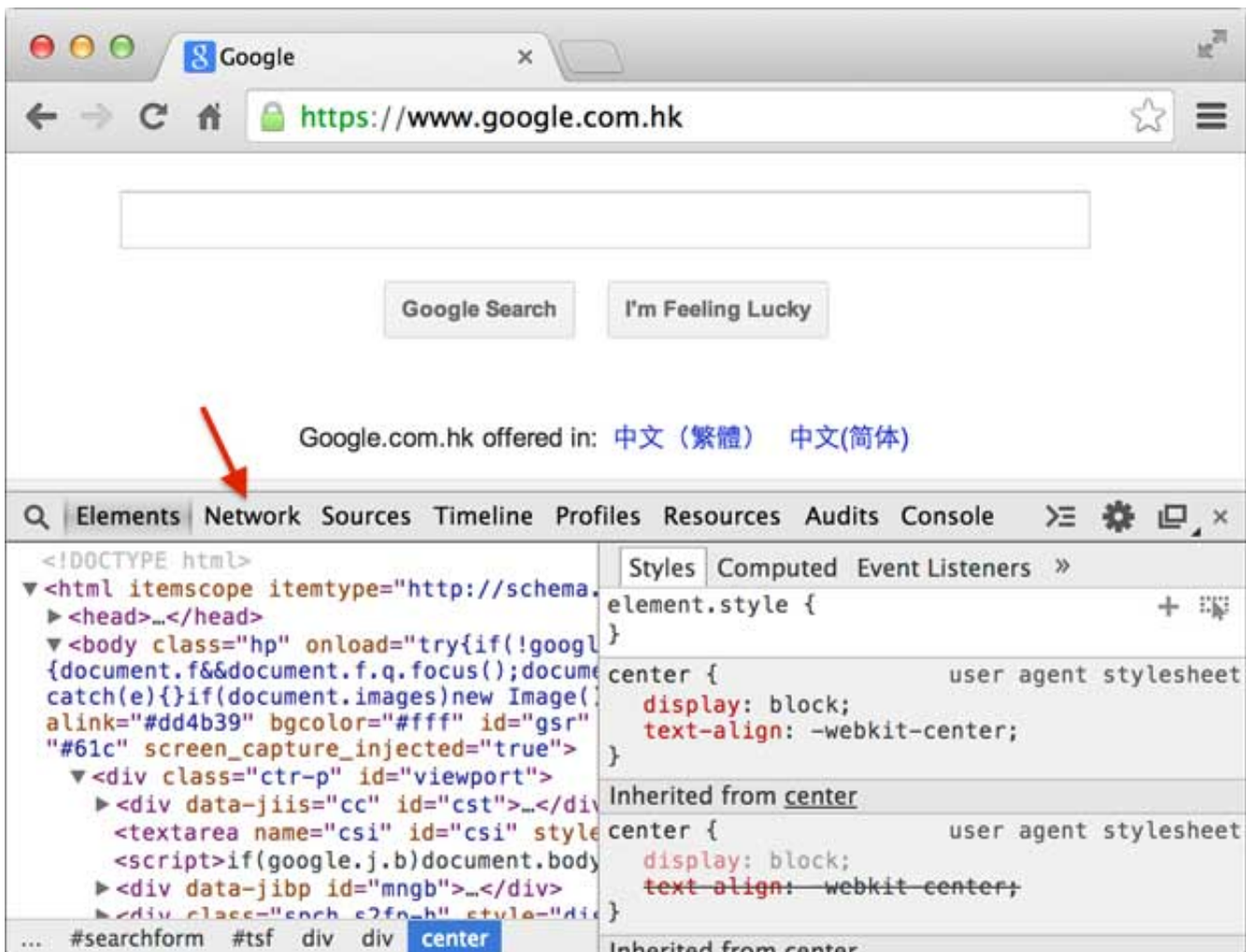
- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

在举例子之前，我们需要安装Google的[Chrome浏览器](#)。

为什么要使用Chrome浏览器而不是IE呢？因为IE实在是太慢了，并且，IE对于开发和调试Web应用程序完全是一点用也没有。

我们需要在浏览器很方便地调试我们的Web应用，而Chrome提供了一套完整地调试工具，非常适合Web开发。

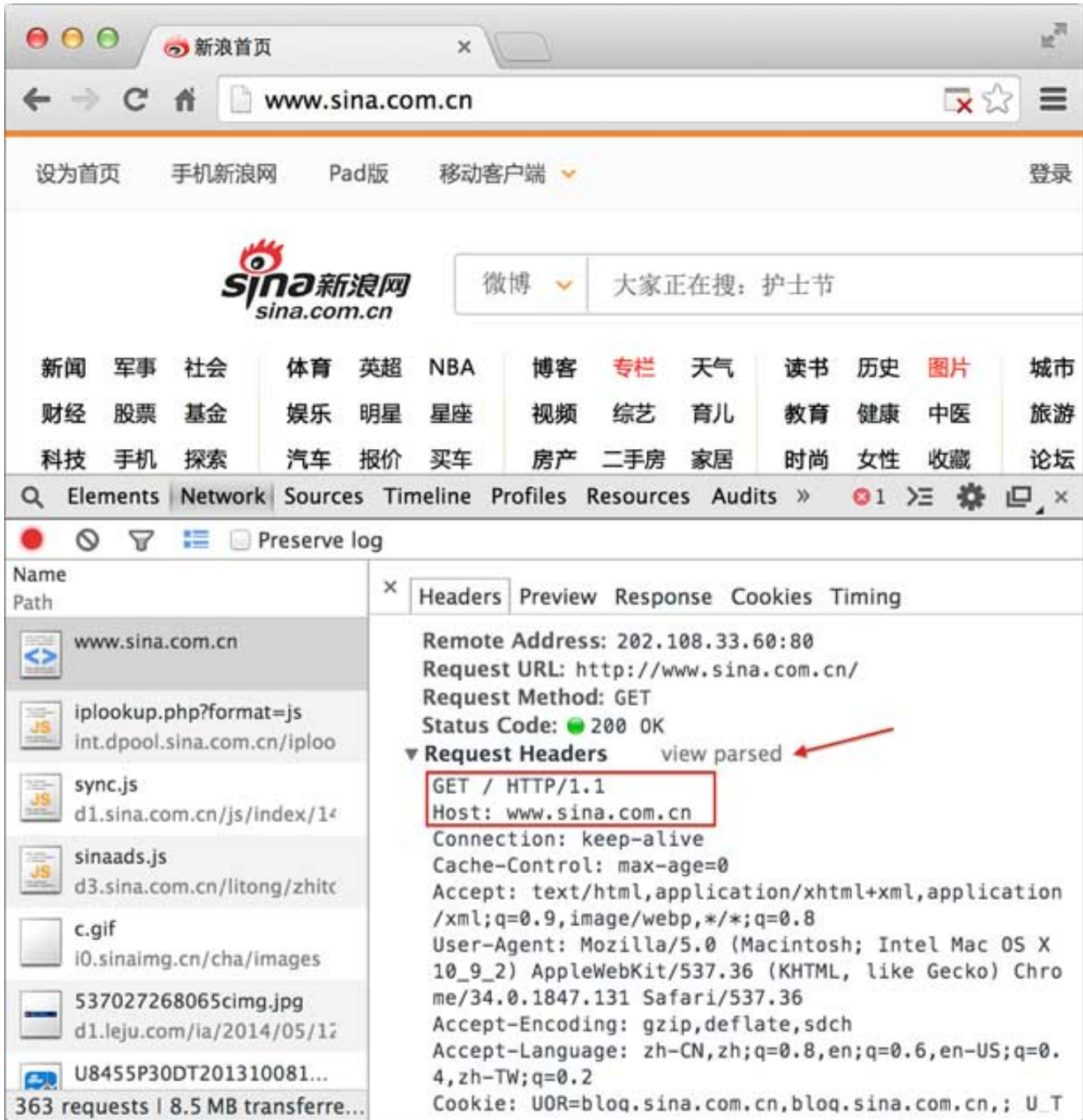
安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



Elements显示网页的结构，Network显示浏览器和服务器的通信。我们点Network，确保第一个小红灯亮着，Chrome就会记录所有浏览器和服务器的通信：



当我们在地址栏输入`www.sina.com.cn`时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过Network的记录，我们就可以知道。在Network中，定位到第一条记录，点击，右侧将显示Request Headers，点击右侧的view source，我们就可以看到浏览器发给新浪服务器的请求：



最主要的头两行分析如下，第一行：

GET / HTTP/1.1

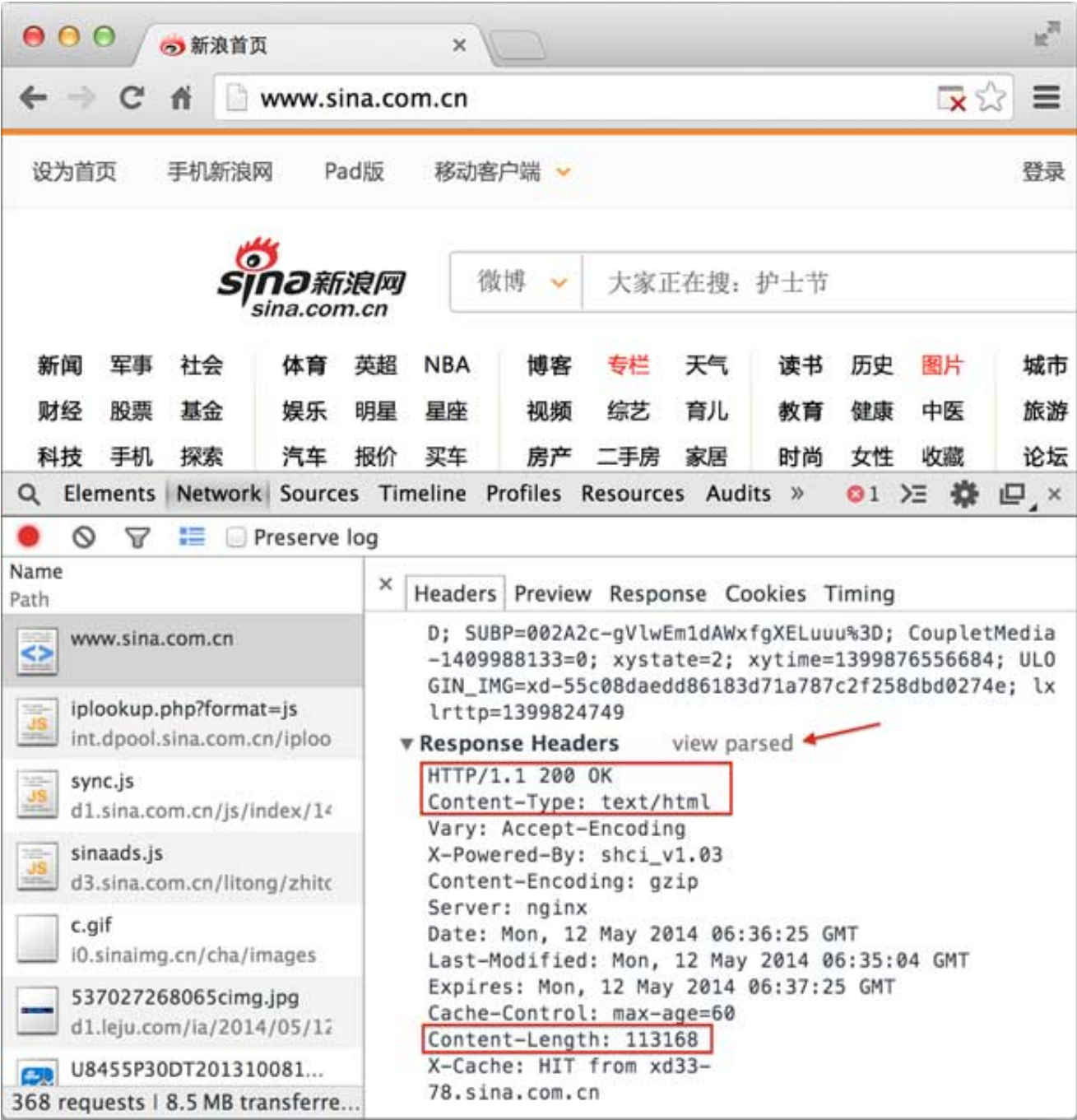
GET表示一个获取请求，将从浏览器获得数据，/表示URL的路径，URL总是以/开头，/就表示首页，最后的HTTP/1.1指示采用的HTTP协议版本是1.1。

从第二行开始，每一行都类似于Xxx: abcdefg:

Host: www.sina.com.cn

表示请求的域名是www.sina.com.cn。如果一台服务器有多个网站，服务器就需要通过Host来区分浏览器请求的是哪个网站。

继续往下找到Response Headers，点击view source，显示服务器返回的原始响应数据：



HTTP响应分为Header和Body两部分（Body是可选项），我们在Network中看到的Header最重要的几行如下：

200 OK

200表示一个成功的响应，后面的OK是说明。失败的响应有404 Not Found：网页不存在，500 Internal Server Error：服务器内部出错，等等。

Content-Type: text/html

Content-Type指示响应的内容，这里是text/html表示HTML网页。请注意，浏览器就是依靠Content-Type来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠URL来判断响应的内容，所以，即使URL是http://example.com/abc.jpg，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：



```
1 <!DOCTYPE html>
2 <!--[30,131,1] published at 2014-05-12 14:35:02 from #153 by 9018-->
3 <html>
4 <head>
5   <meta http-equiv="Content-type" content="text/html; charset=gb2312"
6   />
7   <title>新浪首页</title>
8   <meta name="keywords" content="新浪,新浪网,SINA,sina,sina.com.cn,
9   新浪首页,门户,资讯" />
10  <meta name="description" content="新浪网为全球用户24小时提供全面及时
11  的中文资讯,内容覆盖国内外突发新闻事件、体坛赛事、娱乐时尚、产业资讯、实用信息等,设有
12  新闻、体育、娱乐、财经、科技、房产、汽车等30多个内容频道,同时开设博客、视频、论坛等自
13  由互动交流空间。" />
14  <meta name="stencil" content="PGLS000022" />
15  <meta name="publishid" content="30,131,1" />
16  <meta name="verify-v1"
17  content="6HtwmypyggdgP1NLw7NOuQBI2TW8+CfkYCoyeB8IDbn8=" />
18  <meta name="360-site-verification"
19  content="63349a2167callf4b9bd9a8d48354541" />
20  <meta name="application-name" content="新浪首页" />
21  <meta name="msapplication-TileImage"
22  content="http://il.sinaimg.cn/dy/deco/2013/0312/logo.png" />
23  <meta name="msapplication-TileColor" content="#ffbf27" />
24  <meta name="sogou_site_verification" content="BVIdHxKGrl" />
25  <link rel="apple-touch-icon"
26  href="http://i3.sinaimg.cn/home/2013/0331/U586P30DT20130331093840.png"
27  />
28  <script type="text/javascript">
29    //js异步加载管理
30    (function(){var w=this,d=document,version='1.0.7',data=
31    {},length=0,cbkLen=0;if(w.jsLoader){if(w.jsLoader.version>=version)
32    {return};data=w.jsLoader.getData();length=data.length;var
33    addEvent=function(obj,eventType,func){if(obj.attachEvent)
34    {obj.attachEvent("on"+eventType,func)}else{obj.addEventListener(eventTy
35    pe,func,false)}};var domReady=false,ondomReady=function()
```

当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在Network下面能看到很多额外的HTTP请求。

HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: www.sina.com.cn

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据。

步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是http://www.sina.com.cn/的首页，但是新浪在HTML中可以链入其他服务器的资源，比如，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。HTTP GET请求的格式：

```
GET /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

每个Header一行一个，换行符是\r\n。

HTTP POST请求的格式：

```
POST /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

body data goes here...

当遇到连续两个\r\n时，Header部分结束，后面的数据全部是Body。

HTTP响应的格式：

```
200 OK
Header1: Value1
Header2: Value2
Header3: Value3
```

body data goes here...

HTTP响应如果包含body，也是通过\r\n\r\n来分隔的。请再次注意，Body的数据类型由Content-Type头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在Content-Encoding时，Body数据是被压缩的，最常见的压缩方式是gzip，所以，看到Content-Encoding: gzip时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

要详细了解HTTP协议，推荐“HTTP: The Definitive Guide”一书，非常不错，有中文译本：

<http://www.http-guide.com/>

HTML简介

694次阅读

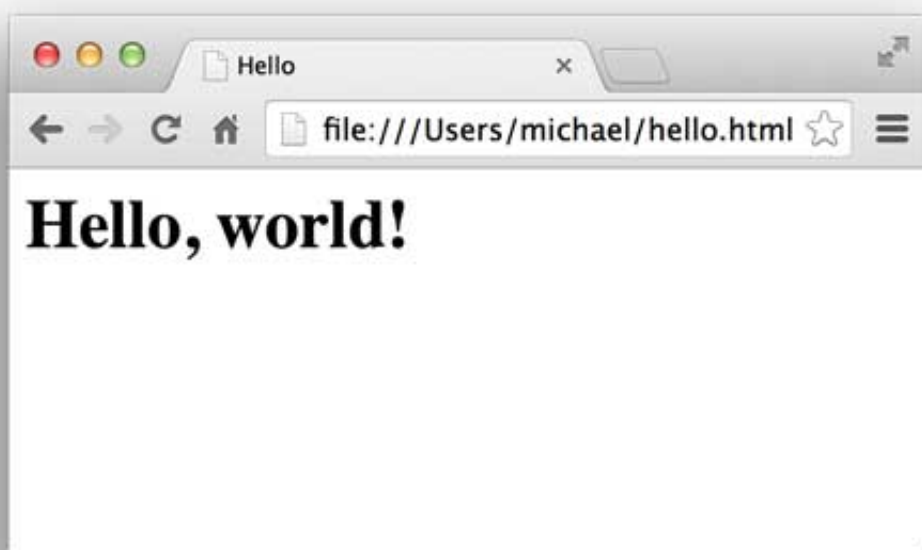
网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

HTML长什么样？上次我们看了新浪首页的HTML源码，如果仔细数数，竟然有6000多行！

所以，学HTML，就不要指望从新浪入手了。我们来看看最简单的HTML长什么样：

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

可以用文本编辑器编写HTML，然后保存为hello.html，双击或者把文件拖到浏览器中，就可以看到效果：



HTML文档就是一系列的Tag组成，最外层的Tag是<html>。规范的HTML也包含<head>...</head>和<body>...</body>（注意不要和HTTP的Header、Body搞混了），由于HTML是富文档模型，所以，还有一系列的Tag用来表示链接、图片、表格、表单等等。

CSS简介

CSS是Cascading Style Sheets（层叠样式表）的简称，CSS用来控制HTML里的所有元素如何展现，比如，给标题元素<h1>加一个样式，变成48号字体，灰色，带阴影：


```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

效果如下:



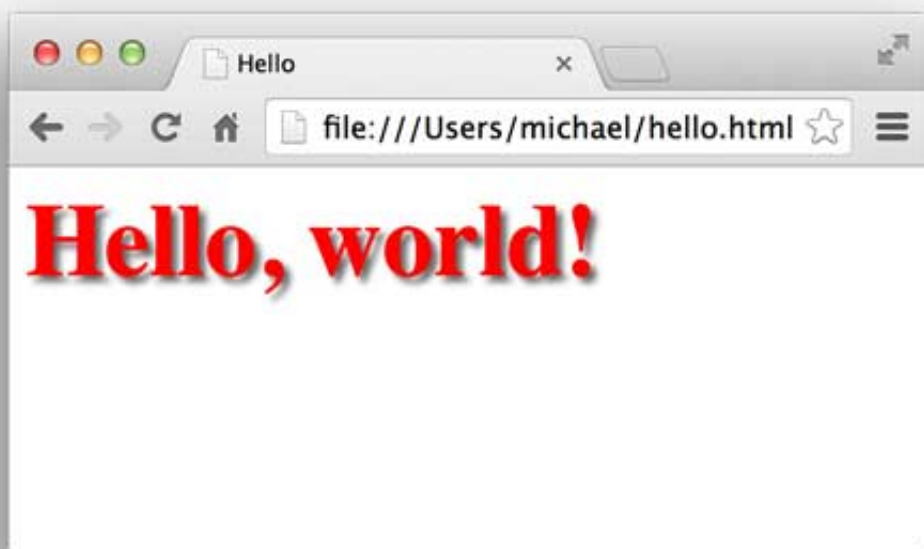
JavaScript简介

JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就必须通过JavaScript来实现：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
  <script>
    function change() {
      document.getElementsByTagName('h1')[0].style.color = '#ff0000';
    }
  </script>
```

```
    }  
    </script>  
</head>  
<body>  
    <h1 onclick="change()">Hello, world!</h1>  
</body>  
</html>
```

效果如下：



小结

如果要学习Web开发，首先要对HTML、CSS和JavaScript作一定的了解。HTML定义了页面的内容，CSS来控制页面元素的样式，而JavaScript负责页面的交互逻辑。

讲解HTML、CSS和JavaScript就可以写3本书，对于优秀的Web开发人员来说，精通HTML、CSS和JavaScript是必须的，这里推荐一个在线学习网站w3schools：

<http://www.w3schools.com/>

以及一个对应的中文版本：

<http://www.w3school.com.cn/>

当我们用Python或者其他语言开发Web应用时，我们就是要在服务器端动态创建出HTML，这样，浏览器就会向不同的用户显示出不同的Web页面。

WSGI接口

951次阅读

了解了HTTP协议和HTML文档，我们其实就明白了一个Web应用的本质就是：

1. 浏览器发送一个HTTP请求；
2. 服务器收到请求，生成一个HTML文档；
3. 服务器把HTML文档作为HTTP响应的Body发送给浏览器；
4. 浏览器收到HTTP响应，从HTTP Body取出HTML文档并显示。

所以，最简单的Web应用就是先把HTML用文件保存好，用一个现成的HTTP服务器软件，接收用户请求，从文件中读取HTML，返回。Apache、Nginx、Lighttpd等这些常见的静态服务器就是干这件事情的。

如果要动态生成HTML，就需要把上述步骤自己来实现。不过，接受HTTP请求、解析HTTP请求、发送HTTP响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态HTML呢，就得花个月去读HTTP规范。

正确的做法是底层代码由专门的服务器软件实现，我们用Python专注于生成HTML文档。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口，让我们专心用Python编写Web业务。

这个接口就是WSGI：Web Server Gateway Interface。

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello, web!”：

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, web!</h1>'
```

上面的application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

- environ：一个包含所有HTTP请求信息的dict对象；
- start_response：一个发送HTTP响应的函数。

在application()函数中，调用：

```
start_response('200 OK', [('Content-Type', 'text/html')])
```

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次start_response()函数。start_response()函数接收两个参数，一个是HTTP响应码，一个是一组list表示的HTTP Header，每个Header用一个包含两个str的tuple表示。

通常情况下，都应该把Content-Type头发送给浏览器。其他很多常用的HTTP Header也应该发送。

然后，函数的返回值'<h1>Hello, web!</h1>'将作为HTTP响应的Body发送给浏览器。

有了WSGI，我们关心的就是如何从environ这个dict对象拿到HTTP请求信息，然后构造HTML，通过start_response()发送Header，最后返回Body。

整个`application()`函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。

不过，等等，这个`application()`函数怎么调用？如果我们自己调用，两个参数`environ`和`start_response`我们没法提供，返回的`str`也没法发给浏览器。

所以`application()`函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器，我们可以挑选一个来用。但是现在，我们只想尽快测试一下我们编写的`application()`函数真的可以把HTML输出到浏览器，所以，要赶紧找一个最简单的WSGI服务器，把我们的Web应用程序跑起来。

好消息是Python内置了一个WSGI服务器，这个模块叫`wsgiref`，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

运行WSGI服务

我们先编写`hello.py`，实现Web应用程序的WSGI处理函数：

```
# hello.py

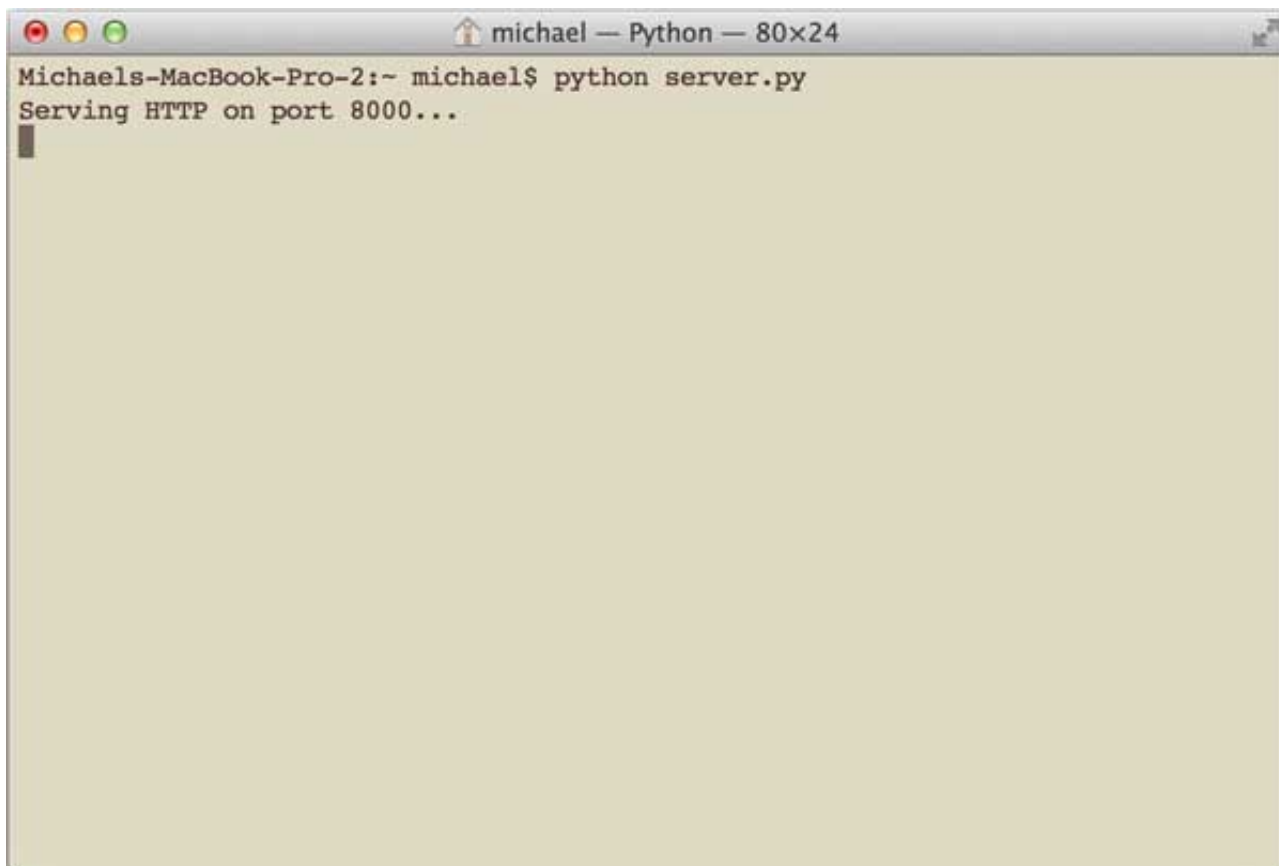
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, web!</h1>'
```

然后，再编写一个`server.py`，负责启动WSGI服务器，加载`application()`函数：

```
# server.py
# 从wsgiref模块导入：
from wsgiref.simple_server import make_server
# 导入我们自己编写的application函数：
from hello import application

# 创建一个服务器，IP地址为空，端口是8000，处理函数是application：
httpd = make_server('', 8000, application)
print "Serving HTTP on port 8000..."
# 开始监听HTTP请求：
httpd.serve_forever()
```

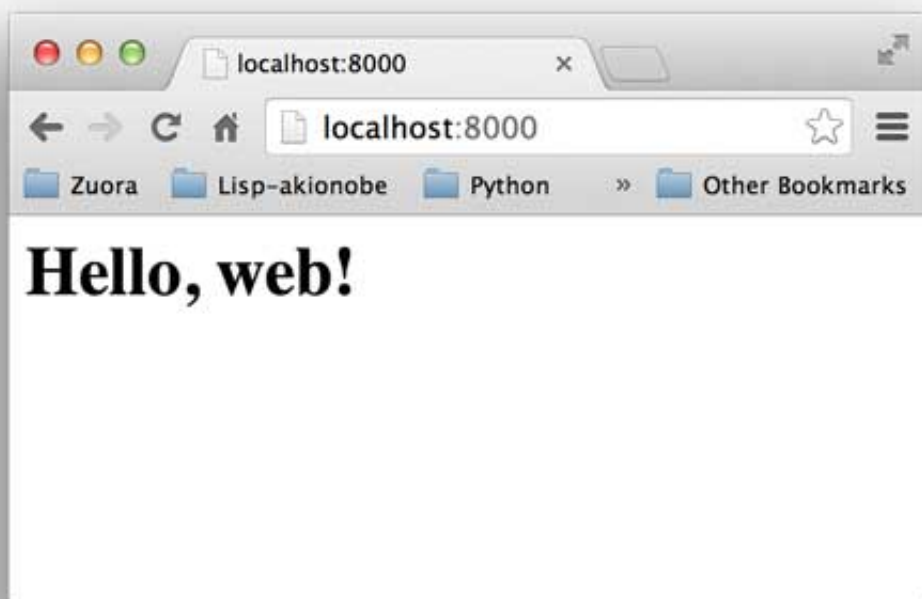
确保以上两个文件在同一个目录下，然后在命令行输入`python server.py`来启动WSGI服务器：



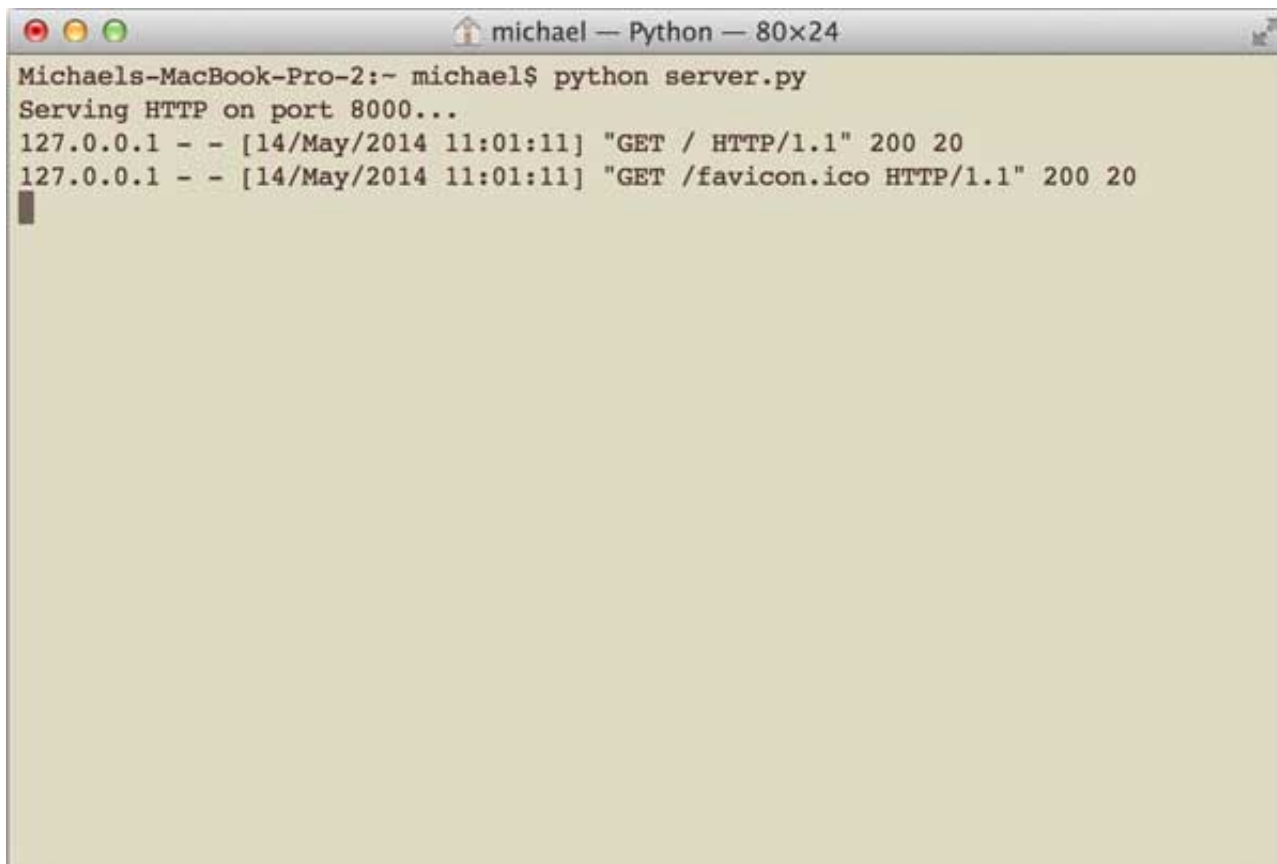
```
michael — Python — 80x24
Michaels-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
█
```

注意：如果8000端口已被其他程序占用，启动将失败，请修改成其他端口。

启动成功后，打开浏览器，输入<http://localhost:8000/>，就可以看到结果了：



在命令行可以看到wsgiref打印的log信息：



```
Michael's-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
127.0.0.1 - - [14/May/2014 11:01:11] "GET / HTTP/1.1" 200 20
127.0.0.1 - - [14/May/2014 11:01:11] "GET /favicon.ico HTTP/1.1" 200 20
```

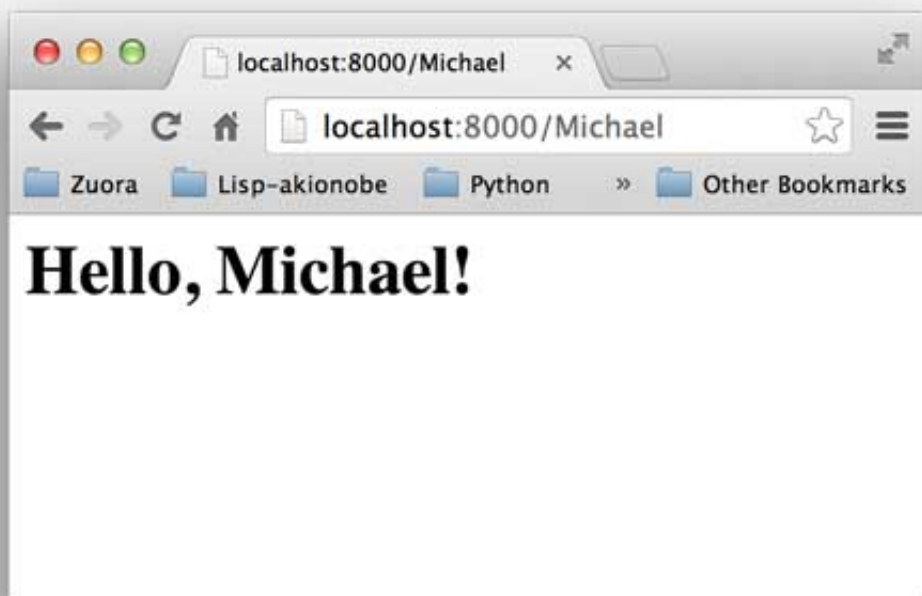
按Ctrl+C终止服务器。

如果你觉得这个Web应用太简单了，可以稍微改造一下，从`environ`里读取`PATH_INFO`，这样可以显示更加动态的内容：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, %s!</h1>' % (environ['PATH_INFO'][1:] or 'web')
```

你可以在地址栏输入用户名作为URL的一部分，将返回Hello, xxx!：



是不是有点Web App的感觉了？

小结

无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过`environ`获得，HTTP响应的输出都可以通过`start_response()`加上函数返回值作为Body。

复杂的Web应用程序，光靠一个WSGI函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

使用Web框架

1015次阅读

了解了WSGI框架，我们发现：其实一个Web App，就是写一个WSGI的处理函数，针对每个HTTP请求进行响应。

但是如何处理HTTP请求不是问题，问题是如何处理100个不同的URL。

每一个URL可以对应GET和POST请求，当然还有PUT、DELETE等请求，但是我们通常只考虑最常见的GET和POST请求。

一个最简单的想法是从environ变量里取出HTTP请求的信息，然后逐个判断：

```
def application(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    if method=='GET' and path=='/':
        return handle_home(environ, start_response)
    if method=='POST' and path='/signin':
        return handle_signin(environ, start_response)
    ...
```

只是这么写下去代码是肯定没法维护了。

代码这么写没法维护的原因是因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级，我们需要在WSGI接口之上能进一步抽象，让我们专注于用一个函数处理一个URL，至于URL到函数的映射，就交给Web框架来做。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。这里我们先不讨论各种Web框架的优缺点，直接选择一个比较流行的Web框架——[Flask](#)来使用。

用Flask编写Web App比WSGI接口简单（这不是废话么，要是比WSGI还复杂，用框架干嘛？），我们先用easy_install或者pip安装Flask：

```
$ easy_install flask
```

然后写一个app.py，处理3个URL，分别是：

- GET /: 首页，返回Home；
- GET /signin: 登录页，显示登录表单；
- POST /signin: 处理登录表单，显示登录结果。

注意噢，同一个URL/signin分别有GET和POST两种请求，映射到两个处理函数中。

Flask通过Python的[装饰器](#)在内部自动地把URL和函数给关联起来，所以，我们写出来的代码就像这样：

```
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
```



```
    return '<h1>Home</h1>'

@app.route('/signin', methods=['GET'])
def signin_form():
    return '''<form action="/signin" method="post">
        <p><input name="username"></p>
        <p><input name="password" type="password"></p>
        <p><button type="submit">Sign In</button></p>
    </form>'''

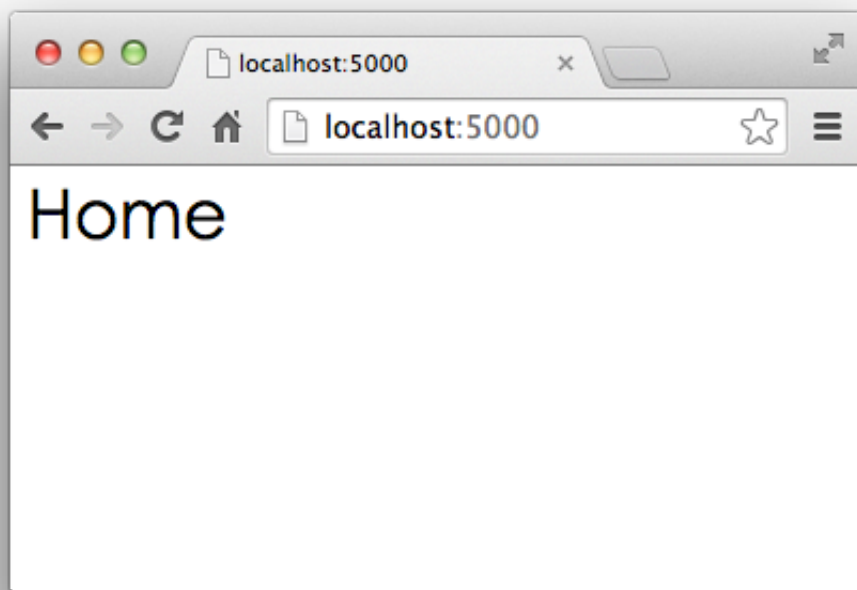
@app.route('/signin', methods=['POST'])
def signin():
    # 需要从request对象读取表单内容：
    if request.form['username'] != 'admin' and request.form['password'] == 'password':
        return '<h3>Hello, admin!</h3>'
    return '<h3>Bad username or password.</h3>'

if __name__ == '__main__':
    app.run()
```

运行python app.py, Flask自带的Server在端口5000上监听:

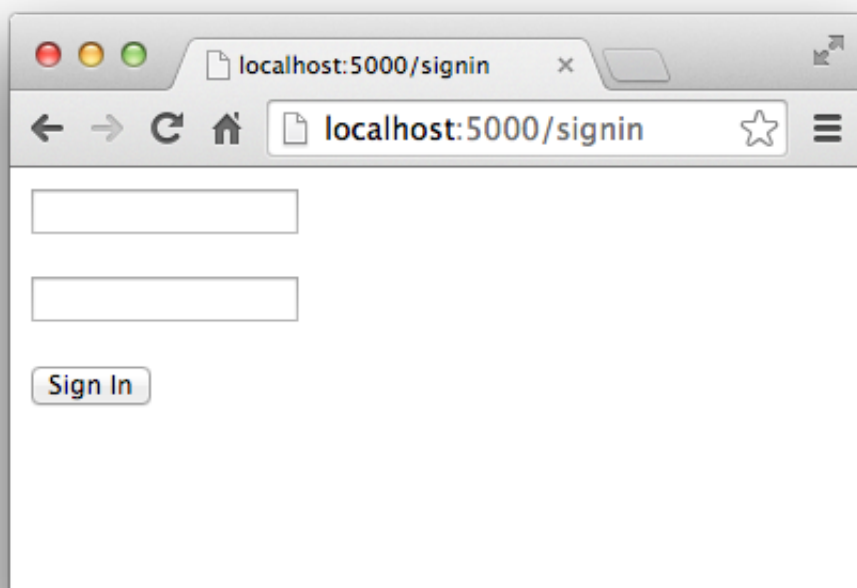
```
$ python app.py
* Running on http://127.0.0.1:5000/
```

打开浏览器, 输入首页地址http://localhost:5000/:

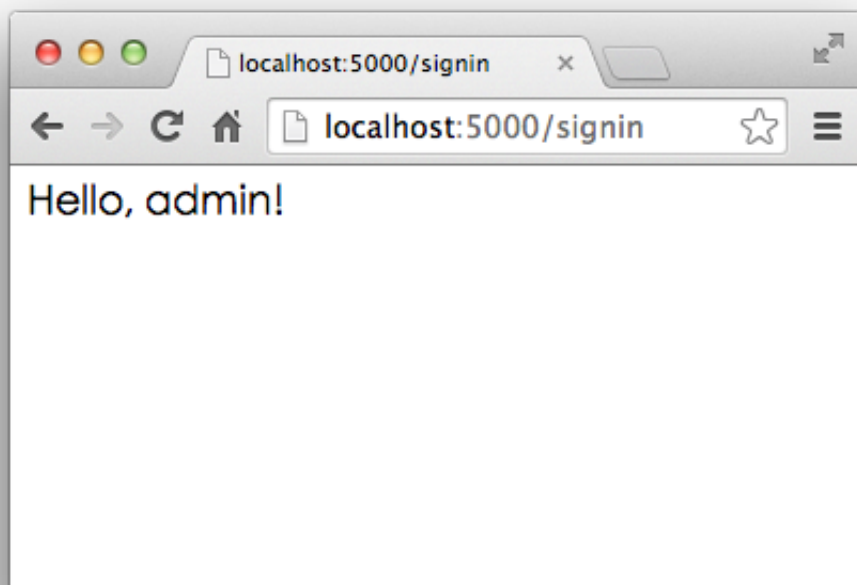


首页显示正确!

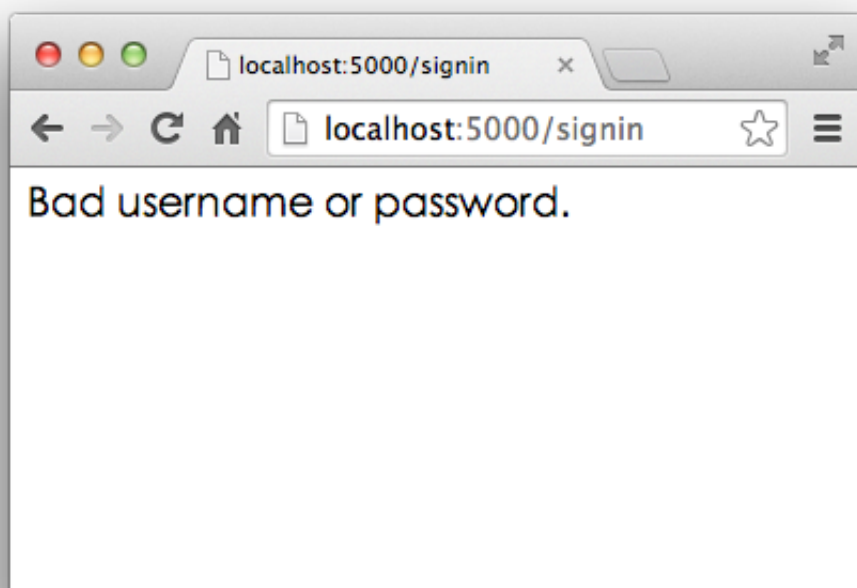
再在浏览器地址栏输入http://localhost:5000/signin, 会显示登录表单:



输入预设的用户名admin和口令password，登录成功：



输入其他错误的用户名和口令，登录失败：



实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

除了Flask，常见的Python Web框架还有：

- [Django](#)：全能型Web框架；
- [web.py](#)：一个小巧的Web框架；
- [Bottle](#)：和Flask类似的Web框架；
- [Tornado](#)：Facebook的开源异步Web框架。

当然了，因为开发Python的Web框架也不是什么难事，我们后面也会自己开发一个Web框架。

小结

有了Web框架，我们在编写Web应用时，注意力就从WSGI处理函数转移到URL+对应的处理函数，这样，编写Web App就更加简单了。

在编写URL处理函数时，除了配置URL外，从HTTP请求拿到用户数据也是非常重要的。Web框架都提供了自己的API来实现这些功能。Flask通过`request.form['name']`来获取表单的内容。

使用模板

676次阅读

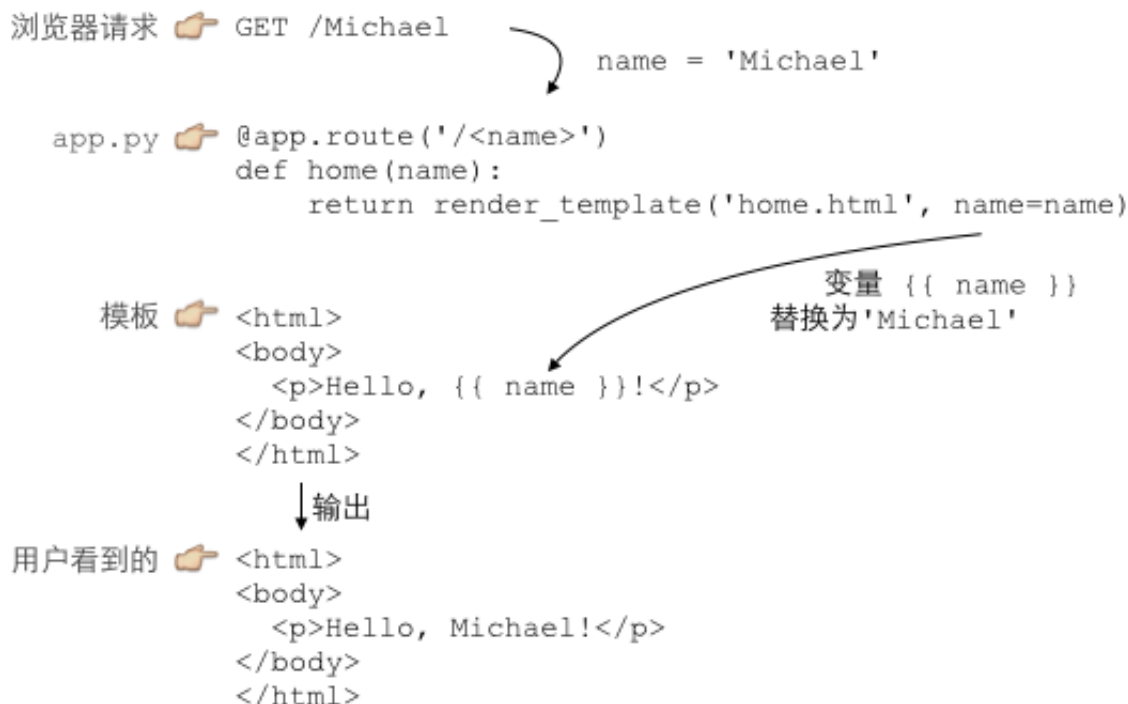
Web框架把我们从WSGI中拯救出来了。现在，我们只需要不断地编写函数，带上URL，就可以继续Web App的开发了。

但是，Web App不仅仅是处理逻辑，展示给用户的页面也非常重要。在函数中返回一个包含HTML的字符串，简单的页面还可以，但是，想想新浪首页的6000多行的HTML，你确信能在Python的字符串中正确地写出来么？反正我是做不到。

俗话说得好，不懂前端的Python工程师不是好的产品经理。有Web开发经验的同学都明白，Web App最复杂的部分就在HTML页面。HTML不仅要正确，还要通过CSS美化，再加上复杂的JavaScript脚本来实现各种交互和动画效果。总之，生成HTML页面的难度很大。

由于在Python代码里拼字符串是不现实的，所以，模板技术出现了。

使用模板，我们需要预先准备一个HTML文档，这个HTML文档不是普通的HTML，而是嵌入了一些变量和指令，然后，根据我们传入的数据，替换后，得到最终的HTML，发送给用户：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

Python处理URL的函数就是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量{{ name }}的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个dict：

```
{ 'name': 'Michael' }
```


只是因为Python支持关键字参数，很多Web框架允许传入关键字参数，然后，在框架内部组装出一个dict作为Model。

现在，我们把上次直接输出字符串作为HTML的例子用高端大气上档次的MVC模式改写一下：

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return render_template('home.html')

@app.route('/signin', methods=['GET'])
def signin_form():
    return render_template('form.html')

@app.route('/signin', methods=['POST'])
def signin():
    username = request.form['username']
    password = request.form['password']
    if username=='admin' and password=='password':
        return render_template('signin-ok.html', username=username)
    return render_template('form.html', message='Bad username or password', username=username)

if __name__ == '__main__':
    app.run()
```

Flask通过render_template()函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是[jinja2](#)，所以我们先直接安装jinja2：

```
$ easy_install jinja2
```

然后，开始编写jinja2模板：

home.html

用来显示首页的模板：

```
<html>
<head>
  <title>Home</title>
</head>
<body>
  <h1 style="font-style:italic">Home</h1>
</body>
</html>
```

form.html

用来显示登录表单的模板：

```
<html>
<head>
  <title>Please Sign In</title>
</head>
<body>
  {% if message %}
  <p style="color:red">{{ message }}</p>
  {% endif %}
  <form action="/signin" method="post">
```

```
<legend>Please sign in:</legend>
<p><input name="username" placeholder="Username" value="{{ username }}"></p>
<p><input name="password" placeholder="Password" type="password"></p>
<p><button type="submit">Sign In</button></p>
</form>
</body>
</html>
```

signin-ok.html

登录成功的模板:

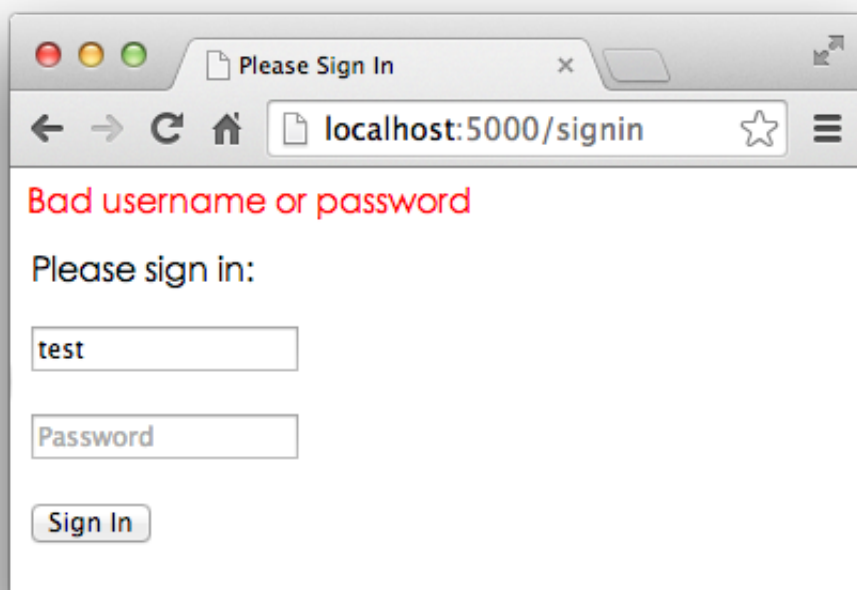
```
<html>
<head>
  <title>Welcome, {{ username }}</title>
</head>
<body>
  <p>Welcome, {{ username }}!</p>
</body>
</html>
```

登录失败的模板呢？我们在form.html中加了一点条件判断，把form.html重用为登录失败的模板。

最后，一定要把模板放到正确的templates目录下，templates和app.py在同级目录下：



启动python app.py，看看使用模板的页面效果：



通过MVC，我们在Python代码中处理M：Model和C：Controller，而V：View是通过模板处理的，这样，我们就成功地把Python代码和HTML代码最大限度地分离了。

使用模板的另一大好处是，模板改起来很方便，而且，改完保存后，刷新浏览器就能看到最新的效果，这对于调试HTML、CSS和JavaScript的前端工程师来说实在是太重要了。

在Jinja2模板中，我们用`{{ name }}`表示一个需要替换的变量。很多时候，还需要循环、条件判断等指令语句，在Jinja2中，用`{% ... %}`表示指令。

比如循环输出页码：

```
{% for i in page_list %}
    <a href="/page/{{ i }}">{{ i }}</a>
{% endfor %}
```

如果`page_list`是一个list: [1, 2, 3, 4, 5]，上面的模板将输出5个超链接。

除了Jinja2，常见的模板还有：

- [Mako](#)：用`<% ... %>`和`${xxx}`的一个模板；
- [Cheetah](#)：也是用`<% ... %>`和`${xxx}`的一个模板；
- [Django](#)：Django是一站式框架，内置一个用`{% ... %}`和`{{ xxx }}`的模板。

小结

有了MVC，我们就分离了Python代码和HTML代码。HTML代码全部放到模板里，写起来更有效率。

协程

577次阅读

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

```
def A():  
    print '1'  
    print '2'  
    print '3'  
  
def B():  
    print 'x'  
    print 'y'  
    print 'z'
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

```
1  
2  
x  
y  
3  
z
```

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
import time

def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        time.sleep(1)
        r = '200 OK'

def produce(c):
    c.next()
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

if __name__ == '__main__':
    c = consumer()
    produce(c)
```

执行结果：

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

注意到consumer函数是一个generator（生成器），把一个consumer传入produce后：

1. 首先调用c.next()启动生成器；
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；

5. produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

gevent

376次阅读

Python通过yield提供了对协程的基本支持，但是不完全。而第三方的gevent为Python提供了比较完善的协程支持。

gevent是第三方库，通过greenlet实现协程，其基本思想是：

当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO。

由于切换是在IO操作时自动完成，所以gevent需要修改Python自带的一些标准库，这一过程在启动时通过monkey patch完成：

```
from gevent import monkey; monkey.patch_socket()
import gevent
```

```
def f(n):
    for i in range(n):
        print gevent.getcurrent(), i
```

```
g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
g1.join()
g2.join()
g3.join()
```

运行结果：

```
<Greenlet at 0x10e49f550: f(5)> 0
<Greenlet at 0x10e49f550: f(5)> 1
<Greenlet at 0x10e49f550: f(5)> 2
<Greenlet at 0x10e49f550: f(5)> 3
<Greenlet at 0x10e49f550: f(5)> 4
<Greenlet at 0x10e49f910: f(5)> 0
<Greenlet at 0x10e49f910: f(5)> 1
<Greenlet at 0x10e49f910: f(5)> 2
<Greenlet at 0x10e49f910: f(5)> 3
<Greenlet at 0x10e49f910: f(5)> 4
<Greenlet at 0x10e49f4b0: f(5)> 0
<Greenlet at 0x10e49f4b0: f(5)> 1
<Greenlet at 0x10e49f4b0: f(5)> 2
<Greenlet at 0x10e49f4b0: f(5)> 3
<Greenlet at 0x10e49f4b0: f(5)> 4
```

可以看到，3个greenlet是依次运行而不是交替运行。

要让greenlet交替运行，可以通过gevent.sleep()交出控制权：

```
def f(n):
    for i in range(n):
        print gevent.getcurrent(), i
        gevent.sleep(0)
```

执行结果：

```
<Greenlet at 0x10cd58550: f(5)> 0
```

```

<Greenlet at 0x10cd58910: f(5)> 0
<Greenlet at 0x10cd584b0: f(5)> 0
<Greenlet at 0x10cd58550: f(5)> 1
<Greenlet at 0x10cd584b0: f(5)> 1
<Greenlet at 0x10cd58910: f(5)> 1
<Greenlet at 0x10cd58550: f(5)> 2
<Greenlet at 0x10cd58910: f(5)> 2
<Greenlet at 0x10cd584b0: f(5)> 2
<Greenlet at 0x10cd58550: f(5)> 3
<Greenlet at 0x10cd584b0: f(5)> 3
<Greenlet at 0x10cd58910: f(5)> 3
<Greenlet at 0x10cd58550: f(5)> 4
<Greenlet at 0x10cd58910: f(5)> 4
<Greenlet at 0x10cd584b0: f(5)> 4

```

3个greenlet交替运行，

把循环次数改为500000，让它们的运行时间长一点，然后在操作系统的进程管理器中看，线程数只有1个。

当然，实际代码里，我们不会用`gevent.sleep()`去切换协程，而是在执行到IO操作时，`gevent`自动切换，代码如下：

```

from gevent import monkey; monkey.patch_all()
import gevent
import urllib2

def f(url):
    print('GET: %s' % url)
    resp = urllib2.urlopen(url)
    data = resp.read()
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
    gevent.spawn(f, 'https://www.python.org/'),
    gevent.spawn(f, 'https://www.yahoo.com/'),
    gevent.spawn(f, 'https://github.com/'),
])

```

运行结果：

```

GET: https://www.python.org/
GET: https://www.yahoo.com/
GET: https://github.com/
45661 bytes received from https://www.python.org/.
14823 bytes received from https://github.com/.
304034 bytes received from https://www.yahoo.com/.

```

从结果看，3个网络操作是并发执行的，而且结束顺序不同，但只有一个线程。

小结

使用`gevent`，可以获得极高的并发性能，但`gevent`只能在Unix/Linux下运行，在Windows下不保证正常安装和运行。

由于`gevent`是基于IO切换的协程，所以最神奇的是，我们编写的Web App代码，不需要引入`gevent`的包，也不需要改任何代码，仅仅在部署的时候，用一个支持`gevent`的WSGI服务器，立刻就获得了数倍的性能提升。具体部署方式可以参考后续“实战” - “部署Web App”一节。

实战

1044次阅读

看完了教程，是不是有这么一种感觉：看的时候觉得很简单，照着教程敲代码也没啥大问题。于是准备开始独立写代码，就发现不知道从哪开始下手了。

这种情况是完全正常的。好比学写作文，学的时候觉得简单，写的时候就无从下笔了。

虽然这个教程是面向小白的零基础Python教程，但是我们的目标不是学到60分，而是学到90分。

所以，用Python写一个真正的Web App吧！

目标

我们设定的实战目标是一个Blog网站，包含日志、用户和评论3大部分。

很多童鞋会想，这是不是太简单了？

比如webpy.org上就提供了一个Blog的例子，目测也就100行代码。

但是，这样的页面：

Hello, world

My first web app...

你拿得出手么？

我们要写出用户真正看得上眼的页面，首页长得像这样：



评论区：



还有极其强大的后台管理页面:



是不是一下子变得高端大气上档次了?

项目名称

必须是高端大气上档次的名称，命名为awesome-python-webapp。

项目计划

项目计划开发周期为16天。每天，你需要完成教程中的内容。如果你觉得编写代码难度实在太太，可以参考一下当天在GitHub上的代码。

第N天的代码在<https://github.com/michaelliao/awesome-python-webapp/tree/day-N>上。比如第1天就是：

<https://github.com/michaelliao/awesome-python-webapp/tree/day-01>

以此类推。

要预览awesome-python-webapp的最终页面效果，请猛击：

awesome.liaoxuefeng.com

Day 1 - 搭建开发环境

3068次阅读

搭建开发环境

首先，确认系统安装的Python版本是2.7.x：

```
$ python --version
Python 2.7.5
```

然后，安装开发Web App需要的第三方库：

前端模板引擎jinja2：

```
$ easy_install jinja2
```

MySQL 5.x数据库，从[官方网站](#)下载并安装，安装完毕后，请务必牢记root口令。为避免遗忘口令，建议直接把root口令设置为password；

MySQL的Python驱动程序mysql-connector-python：

```
$ easy_install mysql-connector-python
```

项目结构

选择一个工作目录，然后，我们建立如下的目录结构：

```
awesome-python-webapp/  <-- 根目录
|
+- backup/              <-- 备份目录
|
+- conf/                <-- 配置文件
|
+- dist/                <-- 打包目录
|
+- www/                 <-- Web目录，存放.py文件
|   |
|   +- static/          <-- 存放静态文件
|   |
|   +- templates/       <-- 存放模板文件
|
+- LICENSE              <-- 代码LICENSE
```

创建好项目的目录结构后，建议同时建立Git仓库并同步至GitHub，保证代码修改的安全。

要了解Git和GitHub的用法，请移步[Git教程](#)。

开发工具

自备，推荐用Sublime Text。
