



廖雪峰 2018年官方最新Python3教程(三)

共三册



扫一扫获取第一册和第二册

- Python教程
 - 一、Web开发
 - 1、HTTP协议简介
 - 2、HTML简介
 - 3、WSGI接口
 - 4、使用Web框架
 - 5、使用模板
 - 二、异步IO
 - 1、协程
 - 2、asyncio
 - 3、async/await
 - 4、aiohttp
 - 三、实战
 - 1、Day 1 - 搭建开发环境
 - 2、Day 2 - 编写Web App骨架
 - 3、Day 3 - 编写ORM
 - 4、Day 4 - 编写Model
 - 5、Day 5 - 编写Web框架
 - 6、Day 6 - 编写配置文件
 - 7、Day 7 - 编写MVC
 - 8、Day 8 - 构建前端
 - 9、Day 9 - 编写API
 - 10、Day 10 - 用户注册和登录
 - 11、Day 11 - 编写日志创建页
 - 12、Day 12 - 编写日志列表页
 - 13、Day 13 - 提升开发效率
 - 14、Day 14 - 完成Web App
 - 15、Day 15 - 部署Web App
 - 16、Day 16 - 编写移动App
 - 四、FAQ
 - 五、期末总结

Python教程

这是小白的Python新手教程，具有如下特点：

中文，免费，零起点，完整示例，基于最新的Python 3版本。

Python是一种计算机程序设计语言。你可能已经听说过很多种流行的编程语言，比如非常难学的C语言，非常流行的Java语言，适合初学者的Basic语言，适合网页编程的JavaScript语言等等。

那Python是一种什么语言？

首先，我们普及一下编程语言的基础知识。用任何编程语言来开发程序，都是为了让计算机干活，比如下载一个MP3，编写一个文档等等，而计算机干活的CPU只认识机器指令，所以，尽管不同的编程语言差异极大，最后都得“翻译”成CPU可以执行的机器指令。而不同的编程语言，干同一个活，编写的代码量，差距也很大。

比如，完成同一个任务，C语言要写1000行代码，Java只需要写100行，而Python可能只要20行。

所以Python是一种相当高级的语言。

你也许会问，代码少还不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的Python程序设计也是非常难学的，所以，高级程序语言不等于简单。

但是，对于初学者和完成普通任务，Python语言是非常简单易用的。连Google都在大规模使用Python，你就不用担心学了会没用。

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，很多著名的网站包括YouTube就是Python写的；可以做网络游戏的后台，很多在线游戏的后台都是Python开发的。总之就是能干很多很多事啦。

Python当然也有不能干的事情，比如写操作系统，这个只能用C语言写；写手机应用，只能用Swift/Objective-C（针对iPhone）和Java（针对Android）；写3D游戏，最好用C或C++。

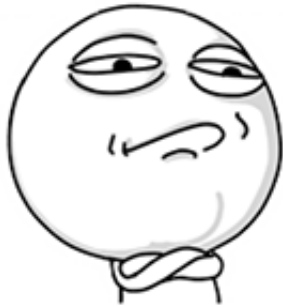
如果你是小白用户，满足以下条件：

- 会使用电脑，但从来没写过程序；
- 还记得初中数学学的方程式和一点点代数知识；
- 想从编程小白变成专业的软件架构师；
- 每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

准备好了吗？

CHALLENGE ACCEPTED !



一、Web开发

最早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。

1、HTTP协议简介

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：

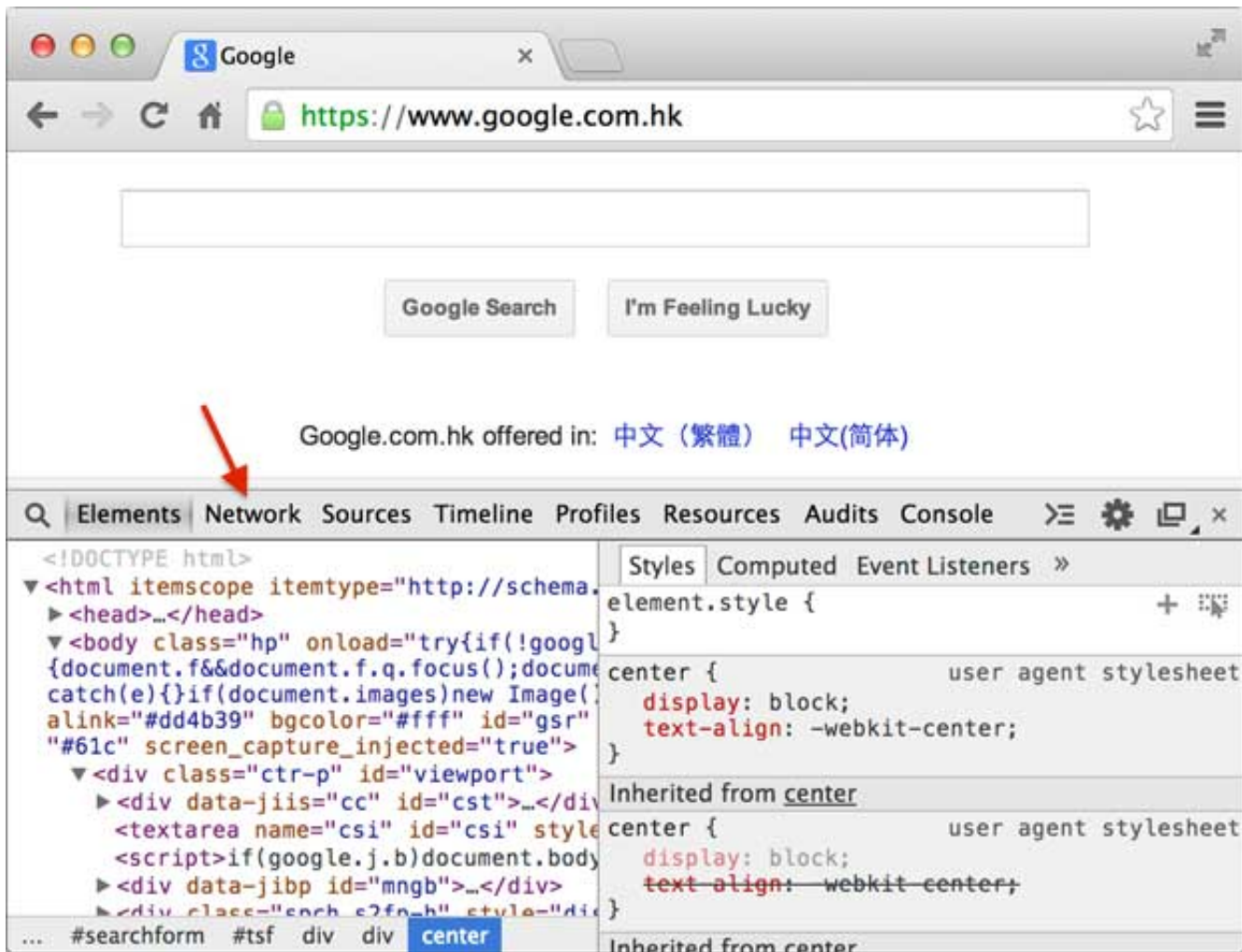
- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

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。

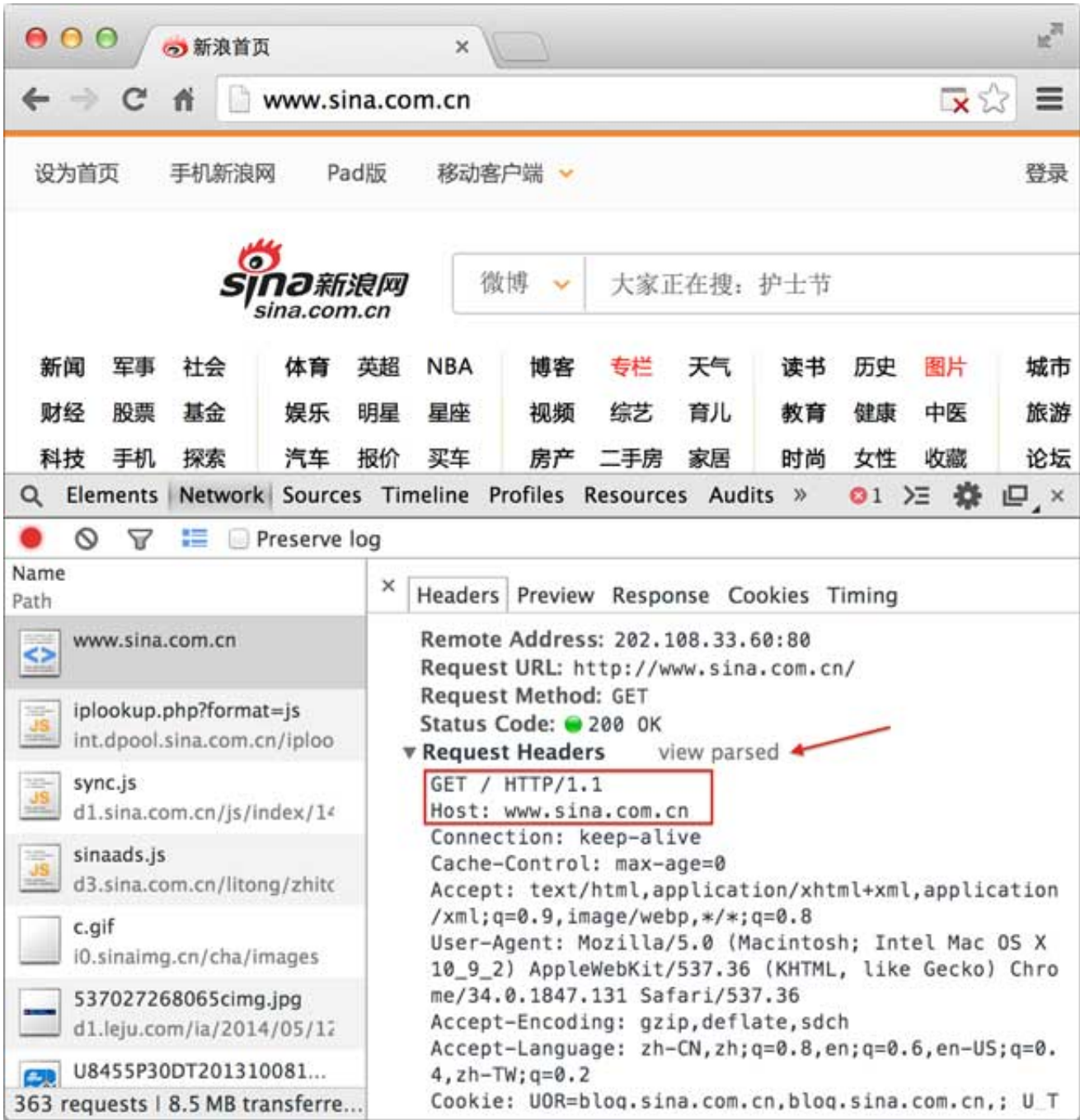
安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



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

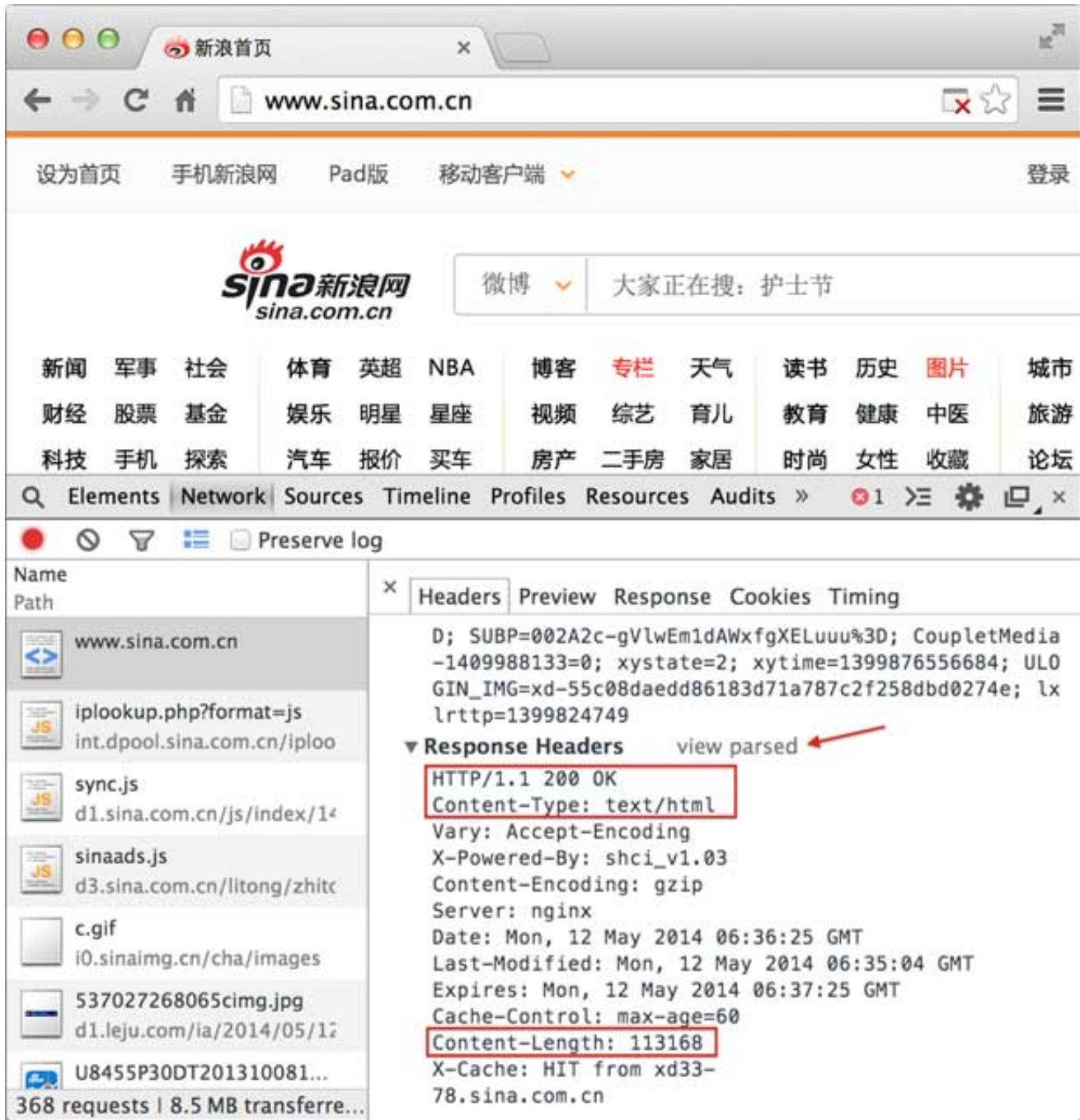
GET 表示一个读取请求，将从服务器获得网页数据， / 表示URL的路径，URL总是以 / 开头， / 就表示首页，最后的 HTTP/1.1 指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用同一个TCP连接，以加快传输速度。

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="6HtwmyppggdgP1NLw7NOuQBI2TW8+CfkYCoyeB8IDbn8=" />
18  <meta name="360-site-verification"
19  content="63349a2167ca11f4b9bd9a8d48354541" />
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" />
  <meta name="msapplication-TileColor" content="#ffbf27" />
  <meta name="sogou_site_verification" content="BVIdHxKGrl" />
  <link rel="apple-touch-icon"
  href="http://i3.sinaimg.cn/home/2013/0331/U586P30DT20130331093840.png"
  />
  <script type="text/javascript">
    //js异步加载管理
    (function(){var w=this,d=document,version='1.0.7',data=
    {},length=0,cbkLen=0;if(w.jsLoader){if(w.jsLoader.version>=version)
    {return};data=w.jsLoader.getData();length=data.length;var
    addEvent=function(obj,eventType,func){if(obj.attachEvent)
    {obj.attachEvent("on"+eventType,func)}else{obj.addEventListener(eventTy
    pe,func,false)}};var domReady=false,ondomReady=function()
    (function(){var w=this,d=document,version='1.0.7',data=
    {},length=0,cbkLen=0;if(w.jsLoader){if(w.jsLoader.version>=version)
    {return};data=w.jsLoader.getData();length=data.length;var
    addEvent=function(obj,eventType,func){if(obj.attachEvent)
    {obj.attachEvent("on"+eventType,func)}else{obj.addEventListener(eventTy
    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 `<img`

`src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">`，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权威指南](#)

2、HTML简介

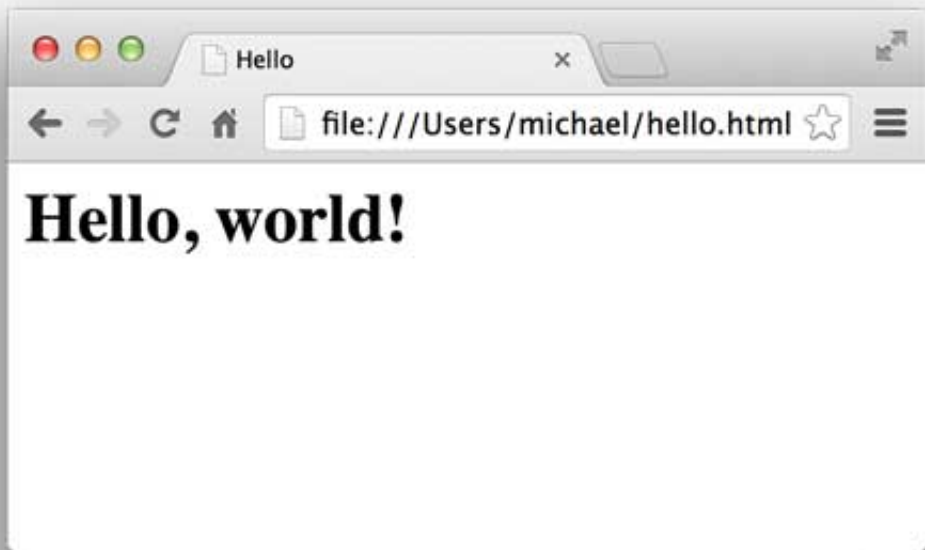
网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

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

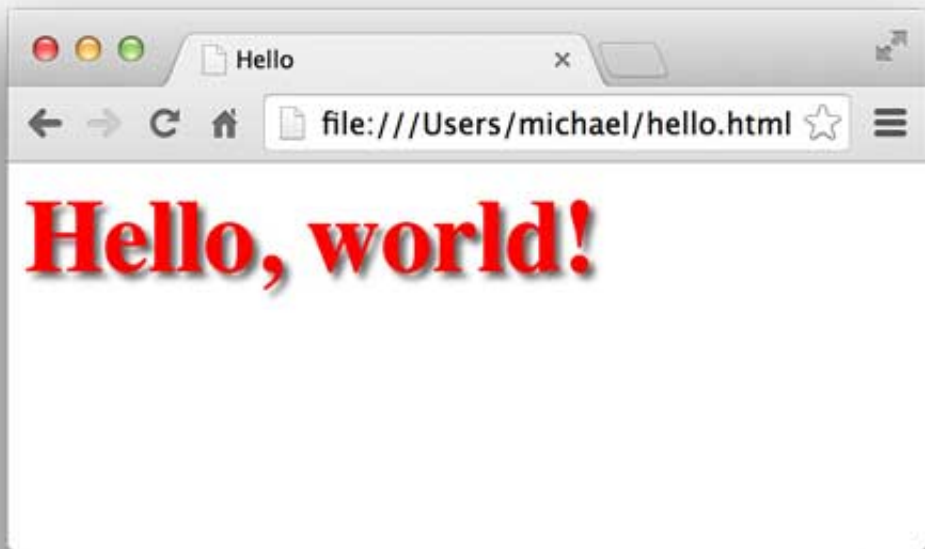
JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就必须通过JavaScript来实现：

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

点击标题后效果如下：



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。

3、WSGI接口

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，就得花个把月去读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 [b'<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 `b'<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 `bytes` 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

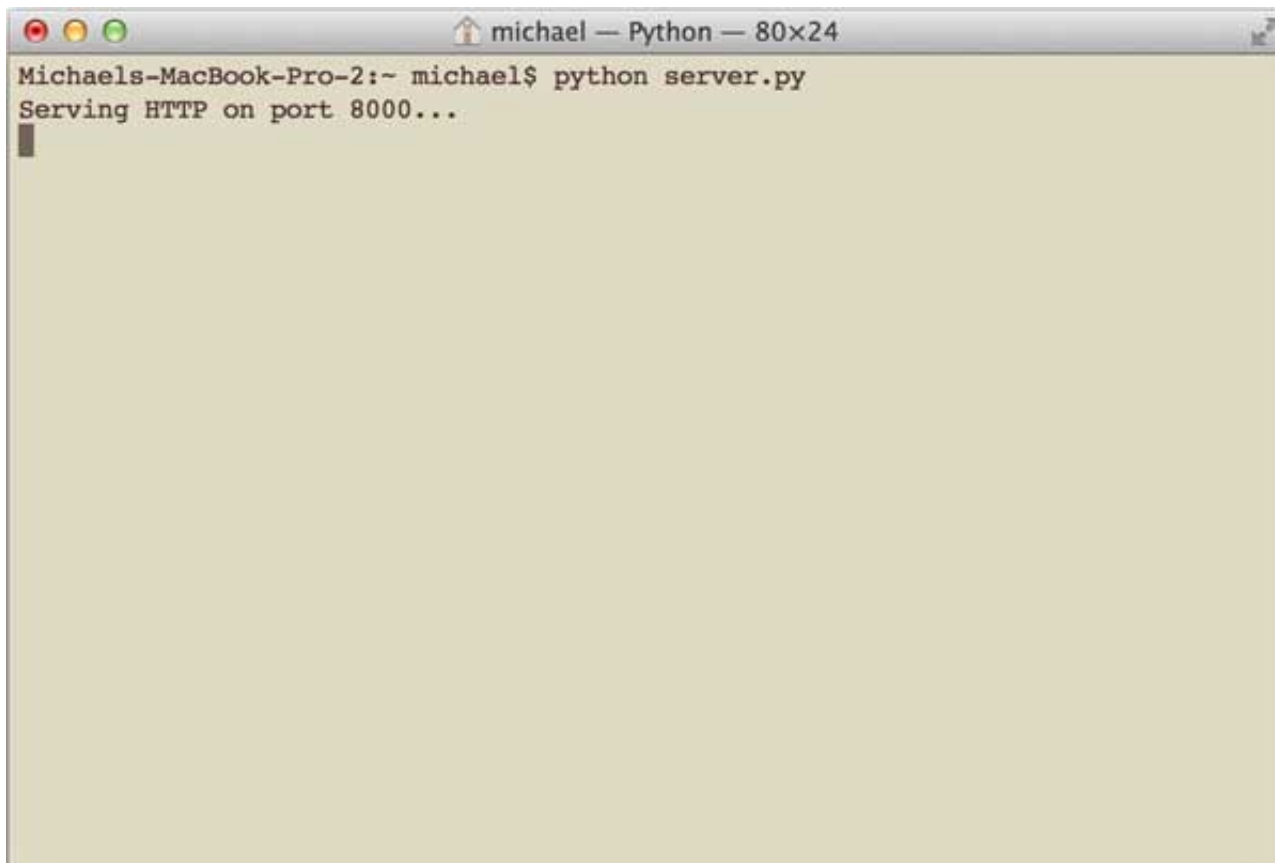
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']
```

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：

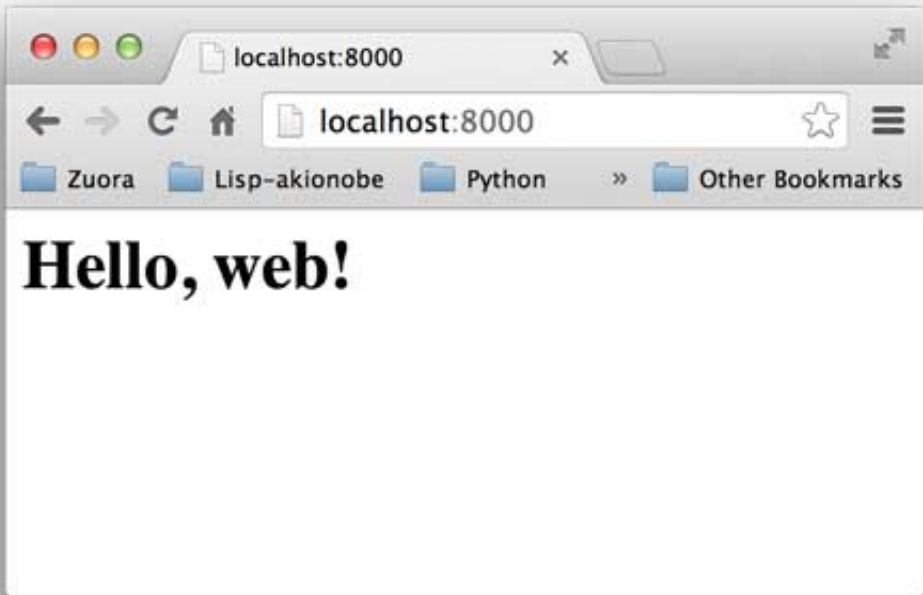


A screenshot of a terminal window titled "michael — Python — 80x24". The window shows the command "python server.py" being executed, followed by the output "Serving HTTP on port 8000...". The terminal background is a light beige color, and the text is in a monospaced font.

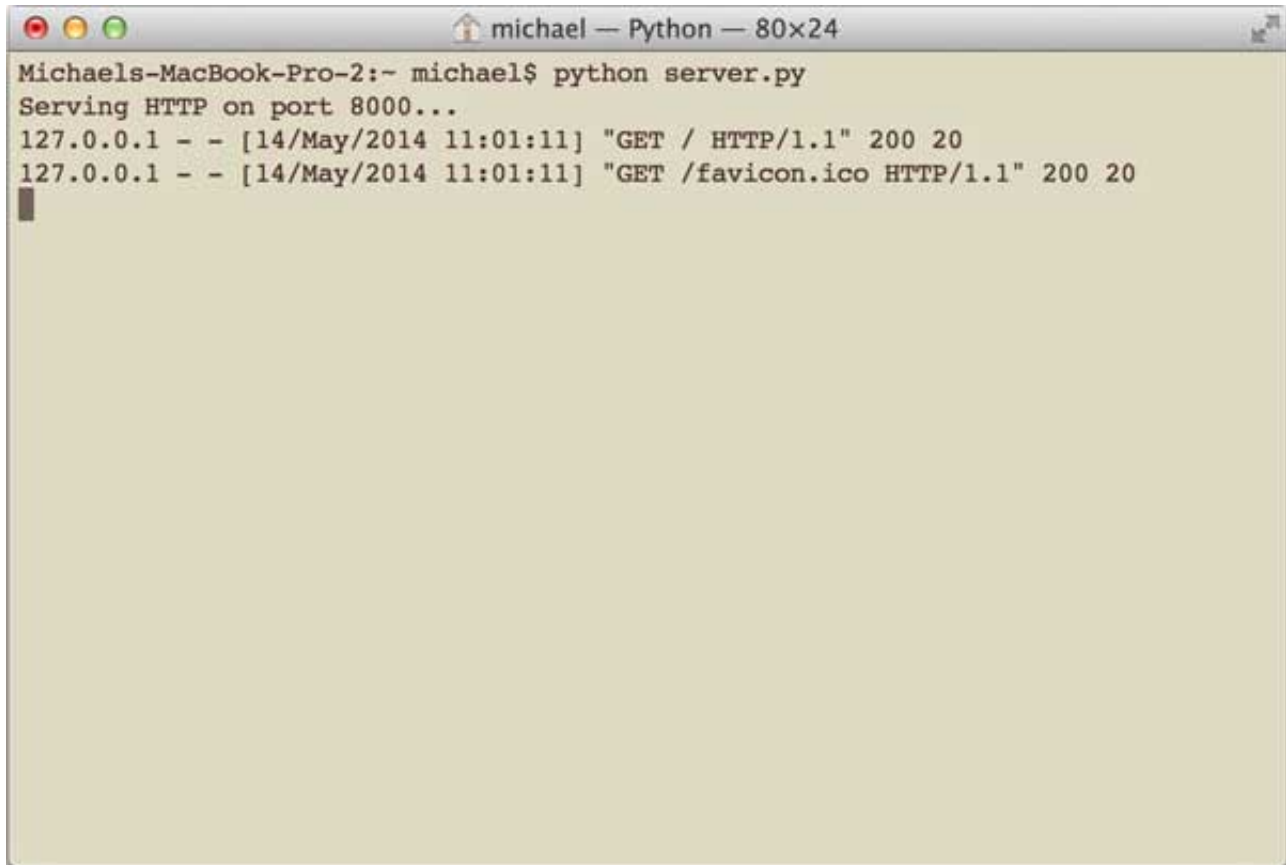
```
Michael's-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
```

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：



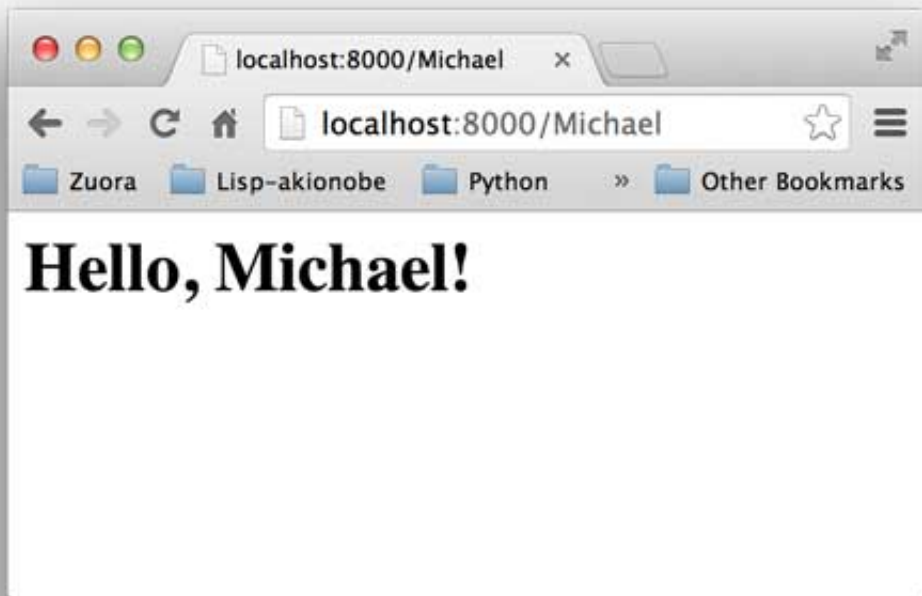
按 `Ctrl+C` 终止服务器。

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')])
    body = '<h1>Hello, %s!</h1>' % (environ['PATH_INFO'][1:] or 'web')
    return [body.encode('utf-8')]
```

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。

参考源码

[hello.py](#)

[do_wsgi.py](#)

4、使用Web框架

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 `pip` 安装Flask：

```
$ pip 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>'''

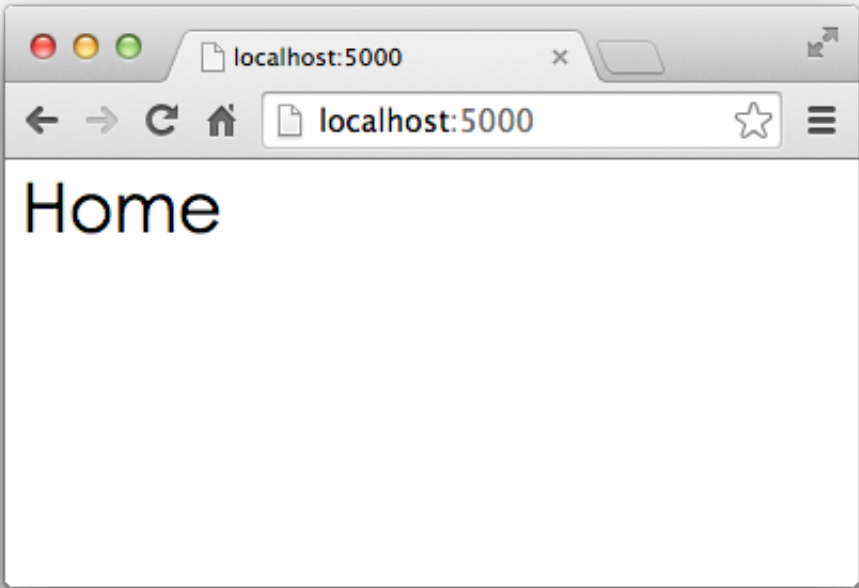
@app.route('/signin', methods=['POST'])
def signin():
    # 需要从request对象读取表单内容：
    if request.form['username']=='admin' and request.form['password']=='password':
        return '<h3>Hello, admin!</h3>'
    return '<h3>Bad username or password.</h3>'

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

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

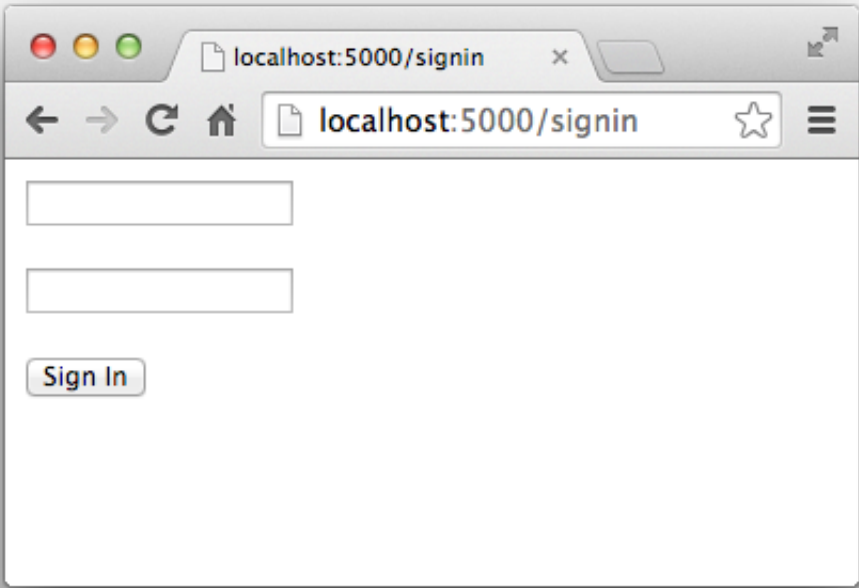
```
$ python app.py
* Running on http://127.0.0.1:5000/
```

打开浏览器，输入首页地址 `http://localhost:5000/`：

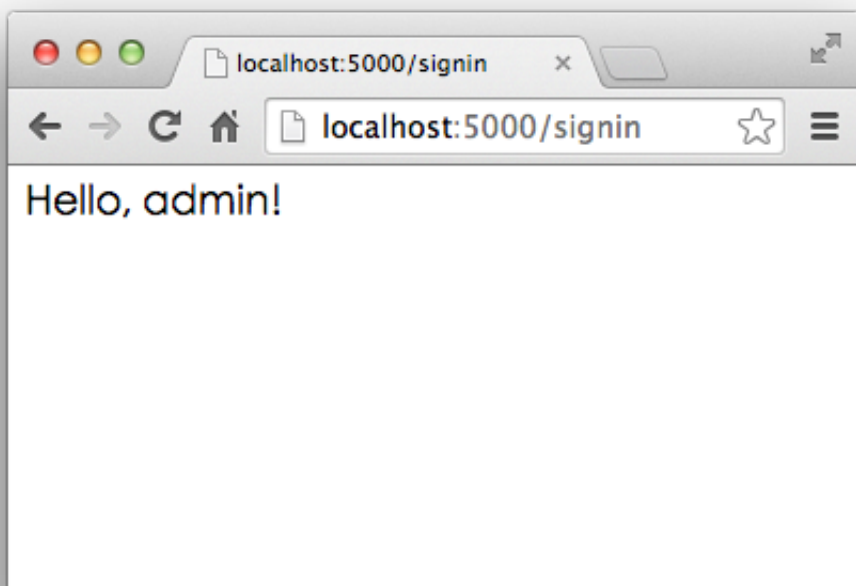


首页显示正确！

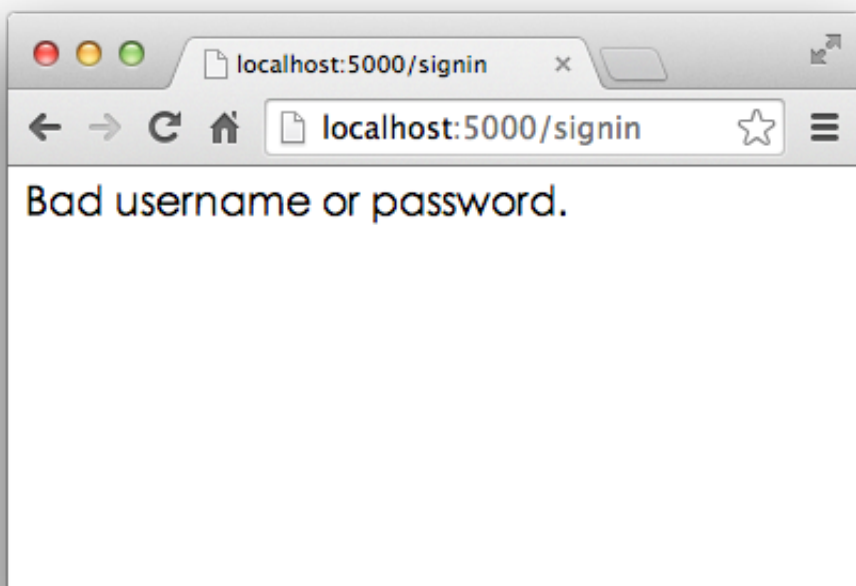
再在浏览器地址栏输入 `http://localhost:5000/signin`，会显示登录表单：



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，我们后面也会讲到开发Web框架的内容。

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。

参考源码

[do_flask.py](#)

5、使用模板

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：

```
$ pip 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：

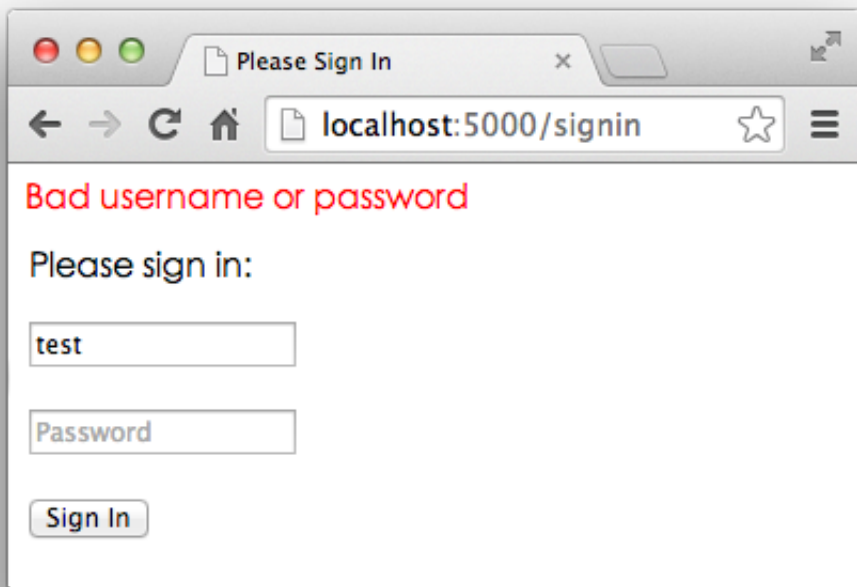
```
<html>
<head>
  <title>Welcome, {{ username }}</title>
</head>
<body>
  <p>Welcome, {{ username }}!</p>
</body>
</html>
```

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。

源码参考

[app.py](#)

二、异步IO

在IO编程一节中，我们已经知道，CPU的速度远远快于磁盘、网络等IO。在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。

在IO操作的过程中，当前线程被挂起，而其他需要CPU执行的代码就无法被当前线程执行了。

因为一个IO操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到IO导致线程被挂起，其他用户的线程不受影响。

多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。

由于我们要解决的问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。

另一种解决IO问题的方法是异步IO。当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

可以想象如果按普通顺序写出的代码实际上是没法完成异步IO的：

```
do_some_code()
f = open('/path/to/file', 'r')
r = f.read() # <== 线程停在此处等待IO操作结果
# IO操作完成后线程才能继续执行：
do_some_code(r)
```

所以，同步IO模型的代码是无法实现异步IO模型的。

异步IO模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

消息模型其实早在应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

由于GUI线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个GUI程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。

消息模型是如何解决同步IO必须等待IO操作这一问题的呢？当遇到IO操作时，代码只负责发出IO请求，不等待IO结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当IO操作完成后，将收到一条“IO完成”的消息，处理该消息时就可以直接获取IO操作结果。

在“发出IO请求”到收到“IO完成”的这段时间里，同步IO模型下，主线程只能挂起，但异步IO模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步IO模型下，一个线程就可以同时处理多个IO请求，并且没有切换线程的操作。对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力。

1、协程

在学习异步IO模型前，我们先来了解协程。

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对协程的支持是通过generator实现的。

在generator中，我们不但可以通过 `for` 循环来迭代，还可以不断调用 `next()` 函数获取由 `yield` 语句返回的下一个值。

但是Python的 `yield` 不但可以返回一个值，它还可以接收调用者发出的参数。

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：

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

def produce(c):
    c.send(None)
    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()

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`，把一个 `consumer` 传入 `produce` 后：

1. 首先调用 `c.send(None)` 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。”

参考源码

[coroutine.py](#)

2、asyncio

`asyncio` 是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

`asyncio` 的编程模型就是一个消息循环。我们从 `asyncio` 模块中直接获取一个 `EventLoop` 的引用，然后把需要执行的协程扔到 `EventLoop` 中执行，就实现了异步IO。

用 `asyncio` 实现 `Hello world` 代码如下：

```
import asyncio

@asyncio.coroutine
```

```
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

`@asyncio.coroutine` 把一个generator标记为coroutine类型，然后，我们就把这个 `coroutine` 扔到 `EventLoop` 中执行。

`hello()` 会首先打印出 `Hello world!`，然后，`yield from` 语法可以让我们方便地调用另一个 `generator`。由于 `asyncio.sleep()` 也是一个 `coroutine`，所以线程不会等待 `asyncio.sleep()`，而是直接中断并执行下一个消息循环。当 `asyncio.sleep()` 返回时，线程就可以从 `yield from` 拿到返回值（此处是 `None`），然后接着执行下一行语句。

把 `asyncio.sleep(1)` 看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行 `EventLoop` 中其他可以执行的 `coroutine` 了，因此可以实现并发执行。

我们用Task封装两个 `coroutine` 试试：

```
import threading
import asyncio

@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

观察执行过程：

```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
```



```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)\n(暂停约1秒)\nHello again! (<_MainThread(MainThread, started 140735195337472)>)\nHello again! (<_MainThread(MainThread, started 140735195337472)>)
```

由打印的当前线程名称可以看出，两个 `coroutine` 是由同一个线程并发执行的。

如果把 `asyncio.sleep()` 换成真正的IO操作，则多个 `coroutine` 就可以由一个线程并发执行。

我们用 `asyncio` 的异步网络连接来获取sina、sohu和163的网站首页：

```
import asyncio\n\n@asyncio.coroutine\ndef wget(host):\n    print('wget %s...' % host)\n    connect = asyncio.open_connection(host, 80)\n    reader, writer = yield from connect\n    header = 'GET / HTTP/1.0\\r\\nHost: %s\\r\\n\\r\\n' % host\n    writer.write(header.encode('utf-8'))\n    yield from writer.drain()\n    while True:\n        line = yield from reader.readline()\n        if line == b'\\r\\n':\n            break\n        print('%s header > %s' % (host, line.decode('utf-8').rstrip()))\n    # Ignore the body, close the socket\n    writer.close()\n\nloop = asyncio.get_event_loop()\ntasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com', 'www.163.com']]\nloop.run_until_complete(asyncio.wait(tasks))\nloop.close()
```

执行结果如下：

```
wget www.sohu.com...\nwget www.sina.com.cn...\nwget www.163.com...\n(等待一段时间)\n(打印出sohu的header)\nwww.sohu.com header > HTTP/1.1 200 OK\nwww.sohu.com header > Content-Type: text/html
```

```
...
(打印出sina的header)
www.sina.com.cn header > HTTP/1.1 200 OK
www.sina.com.cn header > Date: Wed, 20 May 2015 04:56:33 GMT
...
(打印出163的header)
www.163.com header > HTTP/1.0 302 Moved Temporarily
www.163.com header > Server: Cdn Cache Server V2.0
...
```

可见3个连接由一个线程通过 `coroutine` 并发完成。

小结

`asyncio` 提供了完善的异步IO支持；

异步操作需要在 `coroutine` 中通过 `yield from` 完成；

多个 `coroutine` 可以封装成一组Task然后并发执行。

参考源码

[async_hello.py](#)

[async_wget.py](#)

3、async/await

用 `asyncio` 提供的 `@asyncio.coroutine` 可以把一个generator标记为coroutine类型，然后在coroutine内部用 `yield from` 调用另一个coroutine实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法 `async` 和 `await`，可以让coroutine的代码更简洁易读。

请注意，`async` 和 `await` 是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把 `@asyncio.coroutine` 替换为 `async`；
2. 把 `yield from` 替换为 `await`。

让我们对比一下上一节的代码：

```
@asyncio.coroutine
def hello():
```

```
print("Hello world!")
r = yield from asyncio.sleep(1)
print("Hello again!")
```

用新语法重新编写如下：

```
async def hello():
    print("Hello world!")
    r = await asyncio.sleep(1)
    print("Hello again!")
```

剩下的代码保持不变。

小结

Python从3.5版本开始为 `asyncio` 提供了 `async` 和 `await` 的新语法；

注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

练习

将上一节的异步获取sina、sohu和163的网站首页源码用新语法重写并运行。

参考源码

[async_hello2.py](#)

[async_wget2.py](#)

4、aiohttp

`asyncio` 可以实现单线程并发IO操作。如果仅用在客户端，发挥的威力不大。如果把 `asyncio` 用在服务器端，例如Web服务器，由于HTTP连接就是IO操作，因此可以用单线程+ `coroutine` 实现多用户的高并发支持。

`asyncio` 实现了TCP、UDP、SSL等协议， `aiohttp` 则是基于 `asyncio` 实现的HTTP框架。

我们先安装 `aiohttp`：

```
pip install aiohttp
```

然后编写一个HTTP服务器，分别处理以下URL：

- `/` - 首页返回 `b'<h1>Index</h1>'` ；
- `/hello/{name}` - 根据URL参数返回文本 `hello, %s!` 。

代码如下：

```
import asyncio

from aiohttp import web

async def index(request):
    await asyncio.sleep(0.5)
    return web.Response(body=b'<h1>Index</h1>')

async def hello(request):
    await asyncio.sleep(0.5)
    text = '<h1>hello, %s!</h1>' % request.match_info['name']
    return web.Response(body=text.encode('utf-8'))

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    app.router.add_route('GET', '/hello/{name}', hello)
    srv = await loop.create_server(app.make_handler(), '127.0.0.1', 8000)
    print('Server started at http://127.0.0.1:8000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

注意 `aiohttp` 的初始化函数 `init()` 也是一个 `coroutine`，`loop.create_server()` 则利用 `asyncio` 创建TCP服务。

参考源码

[aio_web.py](#)

三、实战

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-python3-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大，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-python3-webapp/tree/day-N` 上。比如第1天就是：

<https://github.com/michaelliao/awesome-python3-webapp/tree/day-01>

以此类推。

要预览 `awesome-python3-webapp` 的最终页面效果，请猛击：

awesome.liaoxuefeng.com

1、Day 1 - 搭建开发环境

搭建开发环境

首先，确认系统安装的Python版本是3.6.x：

```
$ python3 --version
Python 3.6.1
```

然后，用 `pip` 安装开发Web App需要的第三方库：

异步框架：

```
$pip3 install aiohttp
```

前端模板引擎jinja2：

```
$ pip3 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异步驱动程序aiomysql：

```
$ pip3 install aiomysql
```

项目结构

选择一個工作目录，然后，我们建立如下的目录结构：

```
awesome-python3-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
|
+- ios/                   <-- 存放iOS App工程
|
+- 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，请参考[使用文本编辑器](#)。

参考源码

[day-01](#)

2、Day 2 - 编写Web App骨架

由于我们的Web App建立在asyncio的基础上，因此用aiohttp写一个基本的 `app.py`：

```
import logging; logging.basicConfig(level=logging.INFO)

import asyncio, os, json, time
from datetime import datetime

from aiohttp import web

def index(request):
    return web.Response(body=b'<h1>Awesome</h1>')

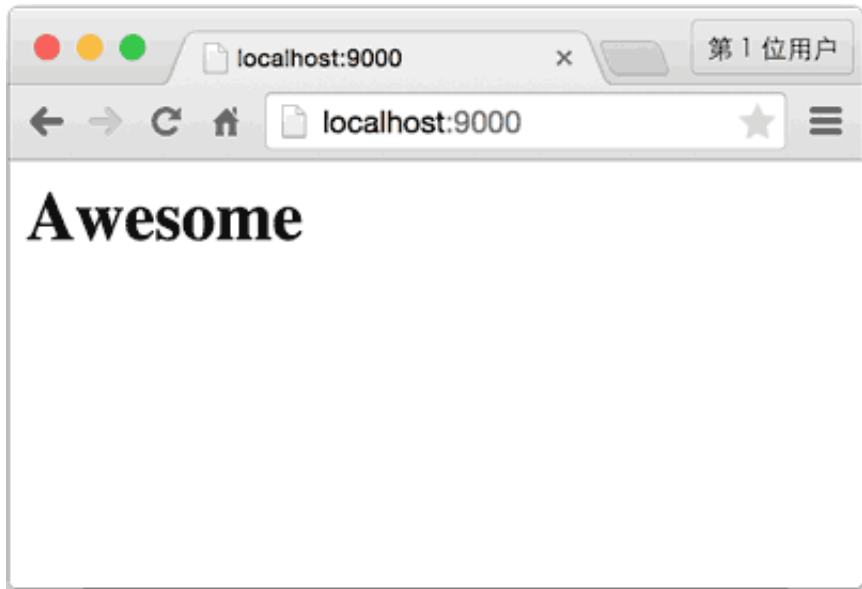
@asyncio.coroutine
def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    srv = yield from loop.create_server(app.make_handler(), '127.0.0.1', 9000)
    logging.info('server started at http://127.0.0.1:9000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

运行 `python app.py`，Web App将在 9000 端口监听HTTP请求，并且对首页 `/` 进行响应：

```
$ python3 app.py
INFO:root:server started at http://127.0.0.1:9000...
```

这里我们简单地返回一个 `Awesome` 字符串，在浏览器中可以看到效果：



这说明我们的Web App骨架已经搭好了，可以进一步往里面添加更多的东西。

参考源码

[day-02](#)

3、Day 3 - 编写ORM

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python3-webapp中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

所以，我们要首先把常用的SELECT、INSERT、UPDATE和DELETE操作用函数封装起来。

由于Web框架使用了基于asyncio的aiohttp，这是基于协程的异步模型。在协程中，不能调用普通的同步IO操作，因为所有用户都是由一个线程服务的，协程的执行速度必须非常快，才能处理大量用户的请求。而耗时的IO操作不能在协程中以同步的方式调用，否则，等待一个IO操作时，系统无法响应任何其他用户。

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步，“开弓没有回头箭”。

幸运的是 `aiomysql` 为MySQL数据库提供了异步IO的驱动。

创建连接池

我们需要创建一个全局的连接池，每个HTTP请求都可以从连接池中直接获取数据库连接。使用

连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。

连接池由全局变量 `__pool` 存储，缺省情况下将编码设置为 `utf8`，自动提交事务：

```
@asyncio.coroutine
def create_pool(loop, **kw):
    logging.info('create database connection pool...')
    global __pool
    __pool = yield from aiomysql.create_pool(
        host=kw.get('host', 'localhost'),
        port=kw.get('port', 3306),
        user=kw['user'],
        password=kw['password'],
        db=kw['db'],
        charset=kw.get('charset', 'utf8'),
        autocommit=kw.get('autocommit', True),
        maxsize=kw.get('maxsize', 10),
        minsize=kw.get('minsize', 1),
        loop=loop
    )
```

Select

要执行SELECT语句，我们用 `select` 函数执行，需要传入SQL语句和SQL参数：

```
@asyncio.coroutine
def select(sql, args, size=None):
    log(sql, args)
    global __pool
    with (yield from __pool) as conn:
        cur = yield from conn.cursor(aiomysql.DictCursor)
        yield from cur.execute(sql.replace('?', '%s'), args or ())
        if size:
            rs = yield from cur.fetchmany(size)
        else:
            rs = yield from cur.fetchall()
        yield from cur.close()
        logging.info('rows returned: %s' % len(rs))
    return rs
```

SQL语句的占位符是 `?`，而MySQL的占位符是 `%s`，`select()` 函数在内部自动替换。注意要始终坚持使用带参数的SQL，而不是自己拼接SQL字符串，这样可以防止SQL注入攻击。

注意到 `yield from` 将调用一个子协程（也就是在一个协程中调用另一个协程）并直接获得子协程的返回结果。

如果传入 `size` 参数，就通过 `fetchmany()` 获取最多指定数量的记录，否则，通过 `fetchall()` 获取所有记录。

Insert, Update, Delete

要执行INSERT、UPDATE、DELETE语句，可以定义一个通用的 `execute()` 函数，因为这3种SQL的执行都需要相同的参数，以及返回一个整数表示影响的行数：

```
@asyncio.coroutine
def execute(sql, args):
    log(sql)
    with (yield from __pool) as conn:
        try:
            cur = yield from conn.cursor()
            yield from cur.execute(sql.replace('?', '%s'), args)
            affected = cur.rowcount
            yield from cur.close()
        except BaseException as e:
            raise
    return affected
```

`execute()` 函数和 `select()` 函数所不同的是，`cursor`对象不返回结果集，而是通过 `rowcount` 返回结果数。

ORM

有了基本的 `select()` 和 `execute()` 函数，我们就可以开始编写一个简单的ORM了。

设计ORM需要从上层调用者角度来设计。

我们先考虑如何定义一个 `User` 对象，然后把数据库表 `users` 和它关联起来。

```
from orm import Model, StringField, IntegerField

class User(Model):
    __table__ = 'users'

    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在 `User` 类中的 `__table__`、`id` 和 `name` 是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述 `User` 对象和表的映射关系，而实例属性必须通过 `__init__()` 方法去初始化，所以两者互不干扰：

```
# 创建实例:
user = User(id=123, name='Michael')
# 存入数据库:
user.insert()
# 查询所有User对象:
users = User.findAll()
```

定义Model

首先要定义的是所有ORM映射的基类 `Model`：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def getValue(self, key):
        return getattr(self, key, None)

    def getValueOrDefault(self, key):
        value = getattr(self, key, None)
        if value is None:
            field = self.__mappings__[key]
            if field.default is not None:
                value = field.default() if callable(field.default) else field.default
                logging.debug('using default value for %s: %s' % (key, str(value)))
                setattr(self, key, value)
        return value
```


Model 从 dict 继承，所以具备所有 dict 的功能，同时又实现了特殊方法 `__getattr__()` 和 `__setattr__()`，因此又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

以及 Field 和各种 Field 子类：

```
class Field(object):

    def __init__(self, name, column_type, primary_key, default):
        self.name = name
        self.column_type = column_type
        self.primary_key = primary_key
        self.default = default

    def __str__(self):
        return '<%s, %s:%s>' % (self.__class__.__name__, self.column_type, self.name)
```

映射 varchar 的 StringField：

```
class StringField(Field):

    def __init__(self, name=None, primary_key=False, default=None, ddl='varchar(100)'):
        super().__init__(name, ddl, primary_key, default)
```

注意到 Model 只是一个基类，如何将具体的子类如 User 的映射信息读取出来呢？答案就是通过 metaclass: ModelMetaclass：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        # 排除Model类本身:
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        # 获取table名称:
```

```

        tableName = attrs.get('__table__', None) or name
        logging.info('found model: %s (table: %s)' % (name, tableName))
        # 获取所有的Field和主键名:
        mappings = dict()
        fields = []
        primaryKey = None
        for k, v in attrs.items():
            if isinstance(v, Field):
                logging.info('found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
                if v.primary_key:
                    # 找到主键:
                    if primaryKey:
                        raise RuntimeError('Duplicate primary key for field: %s' %
k)
                            primaryKey = k
                    else:
                        fields.append(k)
        if not primaryKey:
            raise RuntimeError('Primary key not found.')
        for k in mappings.keys():
            attrs.pop(k)
        escaped_fields = list(map(lambda f: '`%s`' % f, fields))
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = tableName
        attrs['__primary_key__'] = primaryKey # 主键属性名
        attrs['__fields__'] = fields # 除主键外的属性名
        # 构造默认的SELECT, INSERT, UPDATE和DELETE语句:
        attrs['__select__'] = 'select `%s`, %s from `%s`' % (primaryKey, ', '.join(
escaped_fields), tableName)
        attrs['__insert__'] = 'insert into `%s` (%s, `%s`) values (%s)' % (tableNam
e, ', '.join(escaped_fields), primaryKey, create_args_string(len(escaped_fields) +
1))
        attrs['__update__'] = 'update `%s` set %s where `%s`=?' % (tableName, ', '.
join(map(lambda f: '`%s`=?' % (mappings.get(f).name or f), fields)), primaryKey)
        attrs['__delete__'] = 'delete from `%s` where `%s`=?' % (tableName, primary
Key)

        return type.__new__(cls, name, bases, attrs)

```

这样，任何继承自Model的类（比如User），会自动通过ModelMetaclass扫描映射关系，并存储到自身的类属性如 `__table__`、`__mappings__` 中。

然后，我们往Model类添加class方法，就可以让所有子类调用class方法：

```
class Model(dict):
```

```
...

@classmethod
@asyncio.coroutine
def find(cls, pk):
    ' find object by primary key. '
    rs = yield from select('%s where `%(pk)s`=?' % (cls.__select__, cls.__primary_key__), [pk], 1)
    if len(rs) == 0:
        return None
    return cls(**rs[0])
```

User类现在就可以通过类方法实现主键查找：

```
user = yield from User.find('123')
```

往Model类添加实例方法，就可以让所有子类调用实例方法：

```
class Model(dict):

    ...

    @asyncio.coroutine
    def save(self):
        args = list(map(self.getValueOrDefault, self.__fields__))
        args.append(self.getValueOrDefault(self.__primary_key__))
        rows = yield from execute(self.__insert__, args)
        if rows != 1:
            logging.warn('failed to insert record: affected rows: %s' % rows)
```

这样，就可以把一个User实例存入数据库：

```
user = User(id=123, name='Michael')
yield from user.save()
```

最后一步是完善ORM，对于查找，我们可以实现以下方法：

- findAll() - 根据WHERE条件查找；

- findNumber() - 根据WHERE条件查找，但返回的是整数，适用于 `select count(*)` 类型的SQL。

以及 `update()` 和 `remove()` 方法。

所有这些方法都必须用 `@asyncio.coroutine` 装饰，变成一个协程。

调用时需要特别注意：

```
user.save()
```

没有任何效果，因为调用 `save()` 仅仅是创建了一个协程，并没有执行它。一定要用：

```
yield from user.save()
```

才真正执行了INSERT操作。

最后看看我们实现的ORM模块一共多少行代码？累计不到300多行。用Python写一个ORM是不是很容易呢？

参考源码

day-03

4、Day 4 - 编写Model

有了ORM，我们就可以把Web App需要的3个表用 `Model` 表示出来：

```
import time, uuid

from orm import Model, StringField, BooleanField, FloatField, TextField

def next_id():
    return '%015d%s000' % (int(time.time()) * 1000, uuid.uuid4().hex)

class User(Model):
    __table__ = 'users'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    email = StringField(ddl='varchar(50)')
    passwd = StringField(ddl='varchar(50)')
```

```
admin = BooleanField()
name = StringField(ddl='varchar(50)')
image = StringField(ddl='varchar(500)')
created_at = FloatField(default=time.time)

class Blog(Model):
    __table__ = 'blogs'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(default=time.time)

class Comment(Model):
    __table__ = 'comments'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    blog_id = StringField(ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(default=time.time)
```

在编写ORM时，给一个Field增加一个 `default` 参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用 `save()` 时自动计算。

例如，主键 `id` 的缺省值是函数 `next_id`，创建时间 `created_at` 的缺省值是函数 `time.time`，可以自动设置当前日期和时间。

日期和时间用 `float` 类型存储在数据库中，而不是 `datetime` 类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个 `float` 到 `str` 的转换，也非常容易。

初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

```
-- schema.sql

drop database if exists awesome;
```

```
create database awesome;

use awesome;

grant select, insert, update, delete on awesome.* to 'www-data'@'localhost' identified by 'www-data';

create table users (
    `id` varchar(50) not null,
    `email` varchar(50) not null,
    `passwd` varchar(50) not null,
    `admin` bool not null,
    `name` varchar(50) not null,
    `image` varchar(500) not null,
    `created_at` real not null,
    unique key `idx_email` (`email`),
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table blogs (
    `id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `name` varchar(50) not null,
    `summary` varchar(200) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table comments (
    `id` varchar(50) not null,
    `blog_id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;
```

如果表的数量很多，可以从 Model 对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

```
$ mysql -u root -p < schema.sql
```

我们就完成了数据库表的初始化。

编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于 `User` 对象，我们就可以做如下操作：

```
import orm
from models import User, Blog, Comment

def test():
    yield from orm.create_pool(user='www-data', password='www-data', database='awesome')

    u = User(name='Test', email='test@example.com', passwd='1234567890', image='about:blank')

    yield from u.save()

for x in test():
    pass
```

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

参考源码

[day-04](#)

5、Day 5 - 编写Web框架

在正式开始Web开发前，我们需要编写一个Web框架。

`aiohttp` 已经是一个Web框架了，为什么我们还需要自己封装一个？

原因是从使用者的角度来说，`aiohttp` 相对比较底层，编写一个URL的处理函数需要这么几步：

第一步，编写一个用 `@asyncio.coroutine` 装饰的函数：


```
@asyncio.coroutine
def handle_url_xxx(request):
    pass
```

第二步，传入的参数需要自己从 `request` 中获取：

```
url_param = request.match_info['key']
query_params = parse_qs(request.query_string)
```

最后，需要自己构造 `Response` 对象：

```
text = render('template', data)
return web.Response(text.encode('utf-8'))
```

这些重复的工作可以由框架完成。例如，处理带参数的URL `/blog/{id}` 可以这么写：

```
@get('/blog/{id}')
def get_blog(id):
    pass
```

处理 `query_string` 参数可以通过关键字参数 `**kw` 或者命名关键字参数接收：

```
@get('/api/comments')
def api_comments(*, page='1'):
    pass
```

对于函数的返回值，不一定是 `web.Response` 对象，可以是 `str`、`bytes` 或 `dict`。

如果希望渲染模板，我们可以这么返回一个 `dict`：

```
return {
    '__template__': 'index.html',
    'data': '...'
}
```

因此，Web框架的设计是完全从使用者出发，目的是让使用者编写尽可能少的代码。

编写简单的函数而非引入 `request` 和 `web.Response` 还有一个额外的好处，就是可以单独测试，否则，需要模拟一个 `request` 才能测试。

@get和@post

要把一个函数映射为一个URL处理函数，我们先定义 `@get()`：

```
def get(path):
    '''
    Define decorator @get('/path')
    '''
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            return func(*args, **kw)
        wrapper.__method__ = 'GET'
        wrapper.__route__ = path
        return wrapper
    return decorator
```

这样，一个函数通过 `@get()` 的装饰就附带了URL信息。

`@post` 与 `@get` 定义类似。

定义RequestHandler

URL处理函数不一定是一个 `coroutine`，因此我们用 `RequestHandler()` 来封装一个URL处理函数。

`RequestHandler` 是一个类，由于定义了 `__call__()` 方法，因此可以将其实例视为函数。

`RequestHandler` 目的就是从URL函数中分析其需要接收的参数，从 `request` 中获取必要的参数，调用URL函数，然后把结果转换为 `web.Response` 对象，这样，就完全符合 `aiohttp` 框架的要求：

```
class RequestHandler(object):

    def __init__(self, app, fn):
        self._app = app
        self._func = fn
        ...
```

```
@asyncio.coroutine
def __call__(self, request):
    kw = ... 获取参数
    r = yield from self._func(**kw)
    return r
```

再编写一个 `add_route` 函数，用来注册一个URL处理函数：

```
def add_route(app, fn):
    method = getattr(fn, '__method__', None)
    path = getattr(fn, '__route__', None)
    if path is None or method is None:
        raise ValueError('@get or @post not defined in %s.' % str(fn))
    if not asyncio.iscoroutinefunction(fn) and not inspect.isgeneratorfunction(fn):
        fn = asyncio.coroutine(fn)
    logging.info('add route %s %s => %s(%s)' % (method, path, fn.__name__, ', '.join(
        inspect.signature(fn).parameters.keys()))
    app.router.add_route(method, path, RequestHandler(app, fn))
```

最后一步，把很多次 `add_route()` 注册的调用：

```
add_route(app, handles.index)
add_route(app, handles.blog)
add_route(app, handles.create_comment)
...
```

变成自动扫描：

```
# 自动把handler模块的所有符合条件的函数注册了：
add_routes(app, 'handlers')
```

`add_routes()` 定义如下：

```
def add_routes(app, module_name):
    n = module_name.rfind('.')
    if n == (-1):
        mod = __import__(module_name, globals(), locals())
    else:
        name = module_name[n+1:]
```

```

        mod = getattr(__import__(module_name[:n], globals(), locals(), [name]), nam
e)
    for attr in dir(mod):
        if attr.startswith('_'):
            continue
        fn = getattr(mod, attr)
        if callable(fn):
            method = getattr(fn, '__method__', None)
            path = getattr(fn, '__route__', None)
            if method and path:
                add_route(app, fn)

```

最后，在 `app.py` 中加入 `middleware`、`jinja2` 模板和自注册的支持：

```

app = web.Application(loop=loop, middlewares=[
    logger_factory, response_factory
])
init_jinja2(app, filters=dict(datetime=datetime_filter))
add_routes(app, 'handlers')
add_static(app)

```

middleware

`middleware` 是一种拦截器，一个URL在被某个函数处理前，可以经过一系列的 `middleware` 的处理。

一个 `middleware` 可以改变URL的输入、输出，甚至可以决定不继续处理而直接返回。`middleware`的用处就在于把通用的功能从每个URL处理函数中拿出来，集中放到一个地方。例如，一个记录URL日志的 `logger` 可以简单定义如下：

```

@asyncio.coroutine
def logger_factory(app, handler):
    @asyncio.coroutine
    def logger(request):
        # 记录日志:
        logging.info('Request: %s %s' % (request.method, request.path))
        # 继续处理请求:
        return (yield from handler(request))
    return logger

```

而 `response` 这个 `middleware` 把返回值转换为 `web.Response` 对象再返回，以保证满足 `aiohttp` 的

要求：

```
@asyncio.coroutine
def response_factory(app, handler):
    @asyncio.coroutine
    def response(request):
        # 结果:
        r = yield from handler(request)
        if isinstance(r, web.StreamResponse):
            return r
        if isinstance(r, bytes):
            resp = web.Response(body=r)
            resp.content_type = 'application/octet-stream'
            return resp
        if isinstance(r, str):
            resp = web.Response(body=r.encode('utf-8'))
            resp.content_type = 'text/html; charset=utf-8'
            return resp
        if isinstance(r, dict):
            ...
```

有了这些基础设施，我们就可以专注地往 `handlers` 模块不断添加URL处理函数了，可以极大地提高开发效率。

参考源码

day-05

6、Day 6 - 编写配置文件

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的 `.properties` 或者 `.yaml` 等配置文件。

默认的配置应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置命名为 `config_default.py`：

```
# config_default.py
```

```
configs = {
    'db': {
        'host': '127.0.0.1',
        'port': 3306,
        'user': 'www-data',
        'password': 'www-data',
        'database': 'awesome'
    },
    'session': {
        'secret': 'AwEsOmE'
    }
}
```

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的host等信息，直接修改 `config_default.py` 不是一个好办法，更好的方法是编写一个 `config_override.py`，用来覆盖某些默认设置：

```
# config_override.py

configs = {
    'db': {
        'host': '192.168.0.100'
    }
}
```

把 `config_default.py` 作为开发环境的标准配置，把 `config_override.py` 作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从 `config_override.py` 读取。为了简化读取配置文件，可以把所有配置读取到统一的 `config.py` 中：

```
# config.py
configs = config_default.configs

try:
    import config_override
    configs = merge(configs, config_override.configs)
except ImportError:
    pass
```

这样，我们就完成了App的配置。

参考源码

day-06

7、Day 7 - 编写MVC

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的 `@get` 和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

```
@get('/')
def index(request):
    users = yield from User.findAll()
    return {
        '__template__': 'test.html',
        'users': users
    }
```

'__template__' 指定的模板文件是 `test.html`，其他参数是传递给模板的数据，所以我们在模板的根目录 `templates` 下创建 `test.html`：

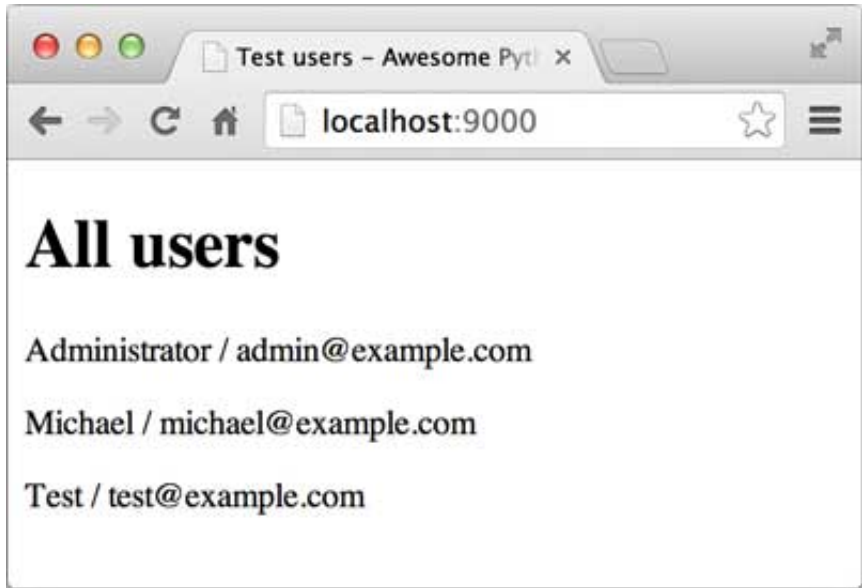
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test users - Awesome Python Webapp</title>
</head>
<body>
    <h1>All users</h1>
    {% for u in users %}
    <p>{{ u.name }} / {{ u.email }}</p>
    {% endfor %}
</body>
</html>
```

接下来，如果一切顺利，可以用命令行启动Web服务器：

```
$ python3 app.py
```


然后，在浏览器中访问 `http://localhost:9000/`。

如果数据库的 `users` 表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给 `users` 表添加几条记录，然后再访问：



参考源码

[day-07](#)

8、Day 8 - 构建前端

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择uikit这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从uikit首页下载打包的资源文件。

所有的静态资源文件我们统一放到 `www/static` 目录下，并按照类别归类：

```
static/
+- css/
| +- addons/
| | +- uikit.addons.min.css
| | +- uikit.almost-flat.addons.min.css
| | +- uikit.gradient.addons.min.css
| +- awesome.css
```

```
| +- uikit.almost-flat.addons.min.css
| +- uikit.gradient.addons.min.css
| +- uikit.min.css
+- fonts/
| +- fontawesome-webfont.eot
| +- fontawesome-webfont.ttf
| +- fontawesome-webfont.woff
| +- FontAwesome.otf
+- js/
  +- awesome.js
  +- html5.js
  +- jquery.min.js
  +- uikit.min.js
```

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

```
<html>
  <% include file="inc_header.html" %>
  <% include file="index_body.html" %>
  <% include file="inc_footer.html" %>
</html>
```

这样，相同的部分 inc_header.html 和 inc_footer.html 就可以共享。

但是include方法不利于页面整体结构的维护。jinja2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

```
<!-- base.html -->
<html>
  <head>
    <title>{% block title%} 这里定义了一个名为title的block {% endblock %}</title>
  </head>
  <body>
```

```

        {% block content %} 这里定义了一个名为content的block {% endblock %}
    </body>
</html>

```

对于子模板 `a.html`，只需要把父模板的 `title` 和 `content` 替换掉：

```

{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}
    <h1>Chapter A</h1>
    <p>blablabla...</p>
{% endblock %}

```

对于子模板 `b.html`，如法炮制：

```

{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}
    <h1>Chapter B</h1>
    <ul>
        <li>list 1</li>
        <li>list 2</li>
    </ul>
{% endblock %}

```

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板 `__base__.html` 的编写：

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    {% block meta %}<!-- block meta -->{% endblock %}
    <title>{% block title %} ? {% endblock %} - Awesome Python Webapp</title>
    <link rel="stylesheet" href="/static/css/uikit.min.css">
    <link rel="stylesheet" href="/static/css/uikit.gradient.min.css">

```

```

<link rel="stylesheet" href="/static/css/awesome.css" />
<script src="/static/js/jquery.min.js"></script>
<script src="/static/js/md5.js"></script>
<script src="/static/js/uikit.min.js"></script>
<script src="/static/js/awesome.js"></script>
{% block beforehead %}<!-- before head -->{% endblock %}
</head>
<body>
  <nav class="uk-navbar uk-navbar-attached uk-margin-bottom">
    <div class="uk-container uk-container-center">
      <a href="/" class="uk-navbar-brand">Awesome</a>
      <ul class="uk-navbar-nav">
        <li data-url="blogs"><a href="/"><i class="uk-icon-home"></i> 日志</a></li>
        <li><a target="_blank" href="#"><i class="uk-icon-book"></i> 教程</a></li>
        <li><a target="_blank" href="#"><i class="uk-icon-code"></i> 源码</a></li>
      </ul>
      <div class="uk-navbar-flip">
        <ul class="uk-navbar-nav">
          {% if user %}
            <li class="uk-parent" data-uk-dropdown>
              <a href="#0"><i class="uk-icon-user"></i> {{ user.name }}</a>
              <div class="uk-dropdown uk-dropdown-navbar">
                <ul class="uk-nav uk-nav-navbar">
                  <li><a href="/signout"><i class="uk-icon-sign-out">
</i> 登出</a></li>
                </ul>
              </div>
            </li>
          {% else %}
            <li><a href="/signin"><i class="uk-icon-sign-in"></i> 登陆</a></li>
            <li><a href="/register"><i class="uk-icon-edit"></i> 注册</a></li>
          {% endif %}
        </ul>
      </div>
    </div>
  </nav>

  <div class="uk-container uk-container-center">
    <div class="uk-grid">
      <!-- content -->
      {% block content %}
      {% endblock %}
    </div>
  </div>

```

```

        <!-- // content -->
    </div>
</div>

<div class="uk-margin-large-top" style="background-color:#eee; border-top:1px solid #ccc;">
    <div class="uk-container uk-container-center uk-text-center">
        <div class="uk-panel uk-margin-top uk-margin-bottom">
            <p>
                <a target="_blank" href="#" class="uk-icon-button uk-icon-weibo"></a>
                <a target="_blank" href="#" class="uk-icon-button uk-icon-github"></a>
                <a target="_blank" href="#" class="uk-icon-button uk-icon-linkedin-square"></a>
                <a target="_blank" href="#" class="uk-icon-button uk-icon-twitter"></a>
            </p>
            <p>Powered by <a href="#">Awesome Python Webapp</a>. Copyright &copy; 2014\.. [<a href="/manage/" target="_blank">Manage</a>]</p>
            <p><a href="http://www.liaoxuefeng.com/" target="_blank">www.liaoxuefeng.com</a>. All rights reserved.</p>
            <a target="_blank" href="#"><i class="uk-icon-html5" style="font-size:64px; color: #444;"></i></a>
        </div>
    </div>
</div>
</body>
</html>

```

`__base__.html` 定义的几个block作用如下:

用于子页面定义一些meta, 例如rss feed:

```
{% block meta %} ... {% endblock %}
```

覆盖页面的标题:

```
{% block title %} ... {% endblock %}
```

子页面可以在 `<head>` 标签关闭前插入JavaScript代码:

```
{% block beforehead %} ... {% endblock %}
```

子页面的content布局和内容：

```
{% block content %}
...
{% endblock %}
```

我们把首页改造一下，从 `__base__.html` 继承一个 `blogs.html`：

```
{% extends '__base__.html' %}

{% block title %}日志{% endblock %}

{% block content %}

    <div class="uk-width-medium-3-4">
        {% for blog in blogs %}
            <article class="uk-article">
                <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></h2>
                <p class="uk-article-meta">发表于{{ blog.created_at }}</p>
                <p>{{ blog.summary }}</p>
                <p><a href="/blog/{{ blog.id }}">继续阅读 <i class="uk-icon-angle-double-right"></i></a></p>
            </article>
            <hr class="uk-article-divider">
        {% endfor %}
    </div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">编程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">读书</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Python教程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Git教程</a></li>
            </ul>
        </div>
```

```
</div>

{% endblock %}
```

相应地，首页URL的处理函数更新如下：

```
@get('/')
def index(request):
    summary = 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eu
smod tempor incididunt ut labore et dolore magna aliqua.'
    blogs = [
        Blog(id='1', name='Test Blog', summary=summary, created_at=time.time()-120)
    ,
        Blog(id='2', name='Something New', summary=summary, created_at=time.time()-
3600),
        Blog(id='3', name='Learn Swift', summary=summary, created_at=time.time()-72
00)
    ]
    return {
        '__template__': 'blogs.html',
        'blogs': blogs
    }
```

Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

```
<p class="uk-article-meta">发表于{{ blog.created_at }}</p>
```

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个 `datetime` 的filter，在模板里用法如下：

```
<p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
```

filter需要在初始化jinja2时设置。相关代码如下：

```
def datetime_filter(t):
    delta = int(time.time() - t)
    if delta < 60:
        return '1分钟前'
```

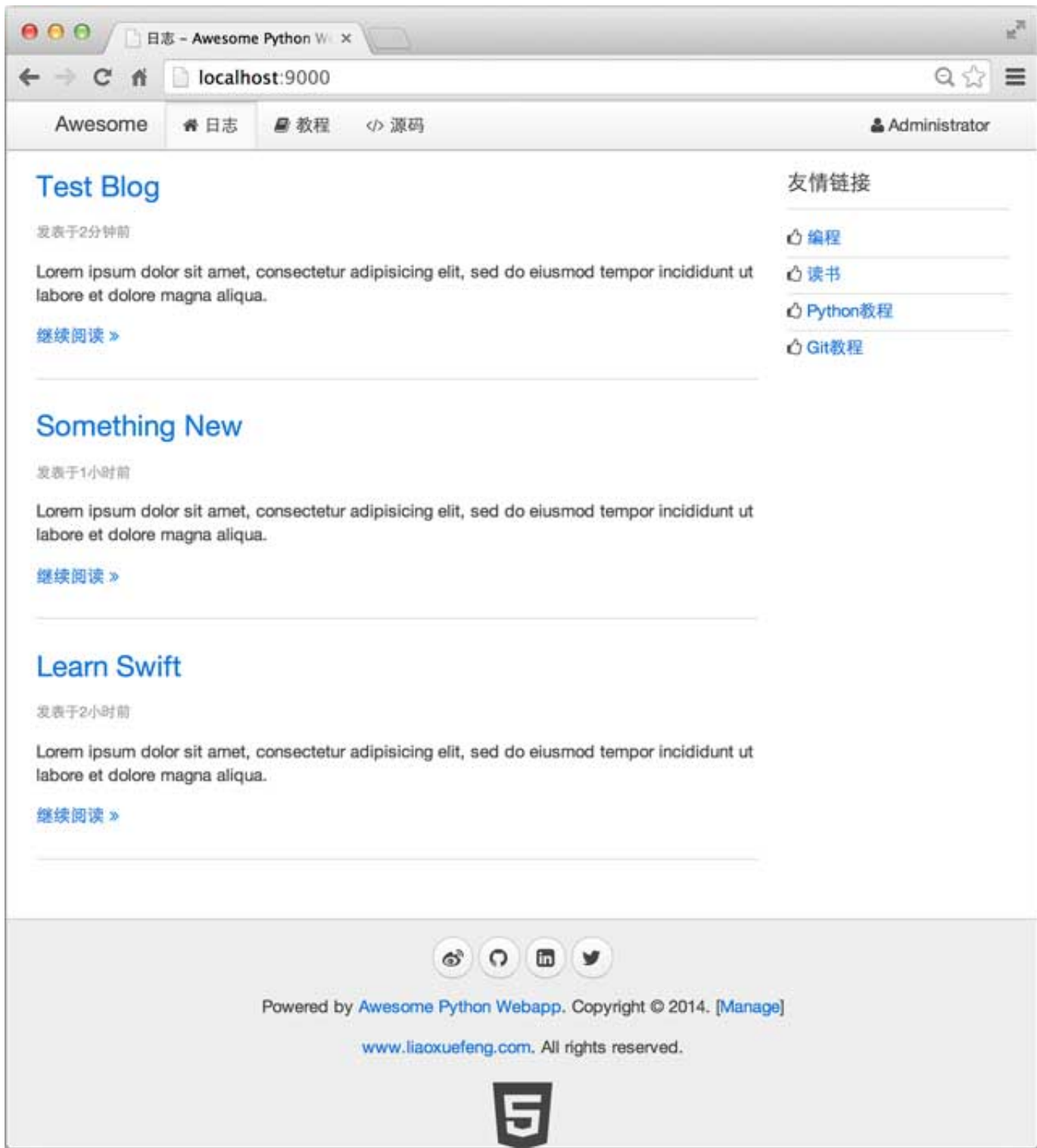
```

if delta < 3600:
    return '%s分钟前' % (delta // 60)
if delta < 86400:
    return '%s小时前' % (delta // 3600)
if delta < 604800:
    return '%s天前' % (delta // 86400)
dt = datetime.fromtimestamp(t)
return '%s年%s月%s日' % (dt.year, dt.month, dt.day)

...
init_jinja2(app, filters=dict(datetime=datetime_filter))
...

```

现在，完善的首页显示如下：



参考源码

day-08

9、Day 9 - 编写API

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入 `http://localhost:9000/blog/123`，就可以看到id为 123 的 Blog 页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取 `http://localhost:9000/api/blogs/123`，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个 `@api` 来把函数变成JSON格式的 REST API，这样，获取注册用户可以用一个API实现如下：

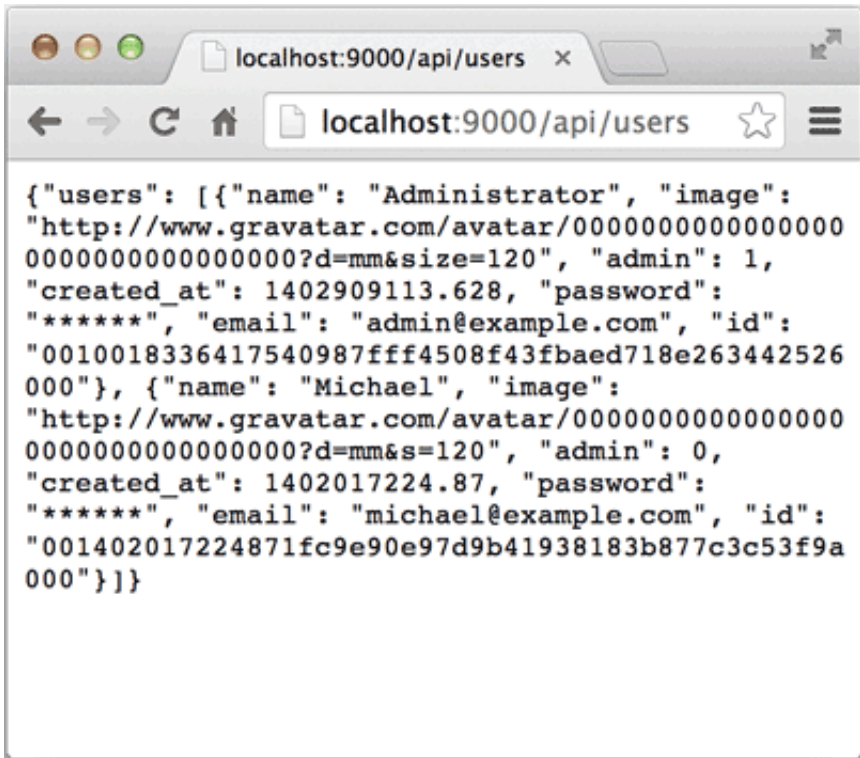
```
@get('/api/users')
def api_get_users(*, page='1'):
    page_index = get_page_index(page)
    num = yield from User.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, users=())
    users = yield from User.findAll(orderBy='created_at desc', limit=(p.offset, p.limit))
    for u in users:
        u.passwd = '*****'
    return dict(page=p, users=users)
```

只要返回一个 `dict`，后续的 `response` 这个 `middleware` 就可以把结果序列化为JSON并返回。

我们需要对Error进行处理，因此定义一个 `APIError`，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为 `internalerror`。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入 `http://localhost:9000/api/users`，就可以看到返回的JSON：



参考源码

day-09

10、Day 10 - 用户注册和登录

用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

```
_RE_EMAIL = re.compile(r'^[a-z0-9\.\-\_]+\@[a-z0-9\-\_]+(\.[a-z0-9\-\_]+){1,4}/div>
)
_RE_SHA1 = re.compile(r'^[0-9a-f]{40}/div>)

@post('/api/users')
def api_register_user(*, email, name, passwd):
    if not name or not name.strip():
        raise APIValueError('name')
    if not email or not _RE_EMAIL.match(email):
        raise APIValueError('email')
    if not passwd or not _RE_SHA1.match(passwd):
        raise APIValueError('passwd')
    users = yield from User.findAll('email=?', [email])
    if len(users) > 0:
        raise APIError('register:failed', 'email', 'Email is already in use.')
    uid = next_id()
    sha1_passwd = '%s:%s' % (uid, passwd)
    user = User(id=uid, name=name.strip(), email=email, passwd=hashlib.sha1(sha1_pa
```

```
sswd.encode('utf-8')).hexdigest(), image='http://www.gravatar.com/avatar/%s?d=mm&s=
120' % hashlib.md5(email.encode('utf-8')).hexdigest())
yield from user.save()
# make session cookie:
r = web.Response()
r.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=86400, httponly=True)

user.passwd = '*****'
r.content_type = 'application/json'
r.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
return r
```

注意用户口令是客户端传递的经过SHA1计算后的40位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

```
{% extends '__base__.html' %}

{% block title %}注册{% endblock %}

{% block beforehead %}

<script>
function validateEmail(email) {
    var re = /^[a-z0-9\.\-\_\_]+\@[a-z0-9\-\_\_]+\.(.[a-z0-9\-\_\_]+){1,4}$/;
    return re.test(email.toLowerCase());
}
$(function () {
    var vm = new Vue({
        el: '#vm',
        data: {
            name: '',
            email: '',
            password1: '',
            password2: ''
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm');
                if (! this.name.trim()) {
                    return $form.showFormError('请输入名字');
                }
                if (! validateEmail(this.email.trim().toLowerCase())) {
                    return $form.showFormError('请输入正确的Email地址');
                }
            }
        }
    });
});
```

```

    }
    if (this.password1.length < 6) {
        return $form.showFormError('口令长度至少为6个字符');
    }
    if (this.password1 !== this.password2) {
        return $form.showFormError('两次输入的口令不一致');
    }
    var email = this.email.trim().toLowerCase();
    $form.postJSON('/api/users', {
        name: this.name.trim(),
        email: email,
        passwd: CryptoJS.SHA1(email + ':' + this.password1).toString()
    }, function (err, r) {
        if (err) {
            return $form.showFormError(err);
        }
        return location.assign('/');
    });
    });
    $('#vm').show();
});
</script>

{% endblock %}

{% block content %}

<div class="uk-width-2-3">
    <h1>欢迎注册! </h1>
    <form id="vm" v-on="submit: submit" class="uk-form uk-form-stacked">
        <div class="uk-alert uk-alert-danger uk-hidden"></div>
        <div class="uk-form-row">
            <label class="uk-form-label">名字:</label>
            <div class="uk-form-controls">
                <input v-model="name" type="text" maxlength="50" placeholder="名字" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">电子邮件:</label>
            <div class="uk-form-controls">
                <input v-model="email" type="text" maxlength="50" placeholder="your-name@example.com" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">

```

```

        <label class="uk-form-label">输入口令:</label>
        <div class="uk-form-controls">
            <input v-model="password1" type="password" maxlength="50" place
holder="输入口令" class="uk-width-1-1">
        </div>
    </div>
    <div class="uk-form-row">
        <label class="uk-form-label">重复口令:</label>
        <div class="uk-form-controls">
            <input v-model="password2" type="password" maxlength="50" place
holder="重复口令" class="uk-width-1-1">
        </div>
    </div>
    <div class="uk-form-row">
        <button type="submit" class="uk-button uk-button-primary"><i class=
"uk-icon-user"></i> 注册</button>
    </div>
</form>
</div>

{% endblock %}

```

这样我们就把用户注册的功能完成了：



The screenshot shows a web browser window with the address bar displaying 'localhost:9000/register'. The page title is '注册 - Awesome Python W'. The navigation bar includes links for 'Awesome', '日志', '教程', '源码', '登陆', and '注册'. The main content area features a large heading '欢迎注册!' followed by registration fields: '名字:' (Name), '电子邮件:' (Email), '输入口令:' (Password), and '重复口令:' (Repeat Password). Each field has a placeholder text. Below the fields is a blue '注册' (Register) button. The footer contains social media icons, the text 'Powered by Awesome Python Webapp. Copyright © 2014. [Manage]', the URL 'www.liaoxuefeng.com. All rights reserved.', and a GitHub logo.

用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验

证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如SHA1），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

```
"用户id" + "过期时间" + SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

- 用户id
- 过期时间
- SHA1值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

```
SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

并与浏览器cookie中的哈希进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于SHA1是一种单向算法，即可以通过原始字符串计算出SHA1结果，但无法通过SHA1结果反推出原始字符串。

所以登录API可以实现如下：

```
@post('/api/authenticate')
def authenticate(*, email, passwd):
    if not email:
        raise APIValueError('email', 'Invalid email.')
    if not passwd:
        raise APIValueError('passwd', 'Invalid password.')
    users = yield from User.findAll('email=?', [email])
    if len(users) == 0:
        raise APIValueError('email', 'Email not exist.')
```



```

user = users[0]
# check passwd:
sha1 = hashlib.sha1()
sha1.update(user.id.encode('utf-8'))
sha1.update(b':')
sha1.update(passwd.encode('utf-8'))
if user.passwd != sha1.hexdigest():
    raise APIValueError('passwd', 'Invalid password.')
# authenticate ok, set cookie:
r = web.Response()
r.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=86400, httponly=True)

user.passwd = '*****'
r.content_type = 'application/json'
r.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
return r

# 计算加密cookie:
def user2cookie(user, max_age):
    # build cookie string by: id-expires-sha1
    expires = str(int(time.time() + max_age))
    s = '%s-%s-%s-%s' % (user.id, user.passwd, expires, _COOKIE_KEY)
    L = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdigest()]
    return '-'.join(L)

```

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用middle在处理URL之前，把cookie解析出来，并将登录用户绑定到 `request` 对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

```

@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' % (request.method, request.path))
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = yield from cookie2user(cookie_str)
            if user:
                logging.info('set current user: %s' % user.email)
                request.__user__ = user
        return (yield from handler(request))
    return auth

# 解密cookie:

```

```
@asyncio.coroutine
def cookie2user(cookie_str):
    '''
    Parse cookie and load user if cookie is valid.
    '''
    if not cookie_str:
        return None
    try:
        L = cookie_str.split('-')
        if len(L) != 3:
            return None
        uid, expires, sha1 = L
        if int(expires) < time.time():
            return None
        user = yield from User.find(uid)
        if user is None:
            return None
        s = '%s-%s-%s-%s' % (uid, user.passwd, expires, _COOKIE_KEY)
        if sha1 != hashlib.sha1(s.encode('utf-8')).hexdigest():
            logging.info('invalid sha1')
            return None
        user.passwd = '*****'
        return user
    except Exception as e:
        logging.exception(e)
        return None
```

这样，我们就完成了用户注册和登录的功能。

参考源码

day-10

11、Day 11 - 编写日志创建页

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog：

```
@post('/api/blogs')
def api_create_blog(request, *, name, summary, content):
    check_admin(request)
    if not name or not name.strip():
        raise APIValueError('name', 'name cannot be empty.')
    if not summary or not summary.strip():
```

```

        raise APIValueError('summary', 'summary cannot be empty.')
    if not content or not content.strip():
        raise APIValueError('content', 'content cannot be empty.')
    blog = Blog(user_id=request.__user__.id, user_name=request.__user__.name, user_
image=request.__user__.image, name=name.strip(), summary=summary.strip(), content=c
ontent.strip())
    yield from blog.save()
    return blog

```

编写后端Python代码不但很简单，而且非常容易测试，上面的API： `api_create_blog()` 本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对此三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

```

s = '<html><head><title>'
    + title
    + '</title></head><body>'
    + body
    + '</body></html>'

```

显然这种方式完全不具备可维护性。所以有第二种模板方式：

```

<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    {{ body }}
</body>
</html>

```

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与

负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的MVVM：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

```
<script>
  var blog = {
    name: 'hello',
    summary: 'this is summary',
    content: 'this is content...'
  };
</script>
```

View是纯HTML：

```
<form action="/api/blogs" method="post">
  <input name="name">
  <input name="summary">
  <textarea name="content"></textarea>
  <button type="submit">OK</button>
</form>
```

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择Vue这个简单易用的MVVM框架来实现创建Blog的页面 `templates/manage_blog_edit.html`：

```
{% extends '__base__.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}
```

```
<script>
var
    ID = '{{ id }}',
    action = '{{ action }}';
function initVM(blog) {
    var vm = new Vue({
        el: '#vm',
        data: blog,
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm').find('form');
                $form.postJSON(action, this.$data, function (err, r) {
                    if (err) {
                        $form.showFormError(err);
                    }
                    else {
                        return location.assign('/api/blogs/' + r.id);
                    }
                });
            }
        }
    });
    $('#vm').show();
}
$(function () {
    if (ID) {
        getJSON('/api/blogs/' + ID, function (err, blog) {
            if (err) {
                return fatal(err);
            }
            $('#loading').hide();
            initVM(blog);
        });
    }
    else {
        $('#loading').hide();
        initVM({
            name: '',
            summary: '',
            content: ''
        });
    }
});
</script>

{% endblock %}
```

```
{% block content %}
```

```

    <div class="uk-width-1-1 uk-margin-bottom">
        <div class="uk-panel uk-panel-box">
            <ul class="uk-breadcrumb">
                <li><a href="/manage/comments">评论</a></li>
                <li><a href="/manage/blogs">日志</a></li>
                <li><a href="/manage/users">用户</a></li>
            </ul>
        </div>
    </div>

    <div id="error" class="uk-width-1-1">
    </div>

    <div id="loading" class="uk-width-1-1 uk-text-center">
        <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加载..
    </span>
    </div>

    <div id="vm" class="uk-width-2-3">
        <form v-on="submit: submit" class="uk-form uk-form-stacked">
            <div class="uk-alert uk-alert-danger uk-hidden"></div>
            <div class="uk-form-row">
                <label class="uk-form-label">标题:</label>
                <div class="uk-form-controls">
                    <input v-model="name" name="name" type="text" placeholder="标题"
class="uk-width-1-1">
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">摘要:</label>
                <div class="uk-form-controls">
                    <textarea v-model="summary" rows="4" name="summary" placeholder
="摘要" class="uk-width-1-1" style="resize:none;"></textarea>
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">内容:</label>
                <div class="uk-form-controls">
                    <textarea v-model="content" rows="16" name="content" placeholde
r="内容" class="uk-width-1-1" style="resize:none;"></textarea>
                </div>
            </div>
            <div class="uk-form-row">
                <button type="submit" class="uk-button uk-button-primary"><i class=
"uk-icon-save"></i> 保存</button>

```

```

        <a href="/manage/blogs" class="uk-button"><i class="uk-icon-times">
</i> 取消</a>
    </div>
</form>
</div>

{% endblock %}

```

初始化Vue时，我们指定3个参数：

el：根据选择器查找绑定的View，这里是 `#vm`，就是id为 `vm` 的DOM，对应的是一个 `<div>` 标签；

data：JavaScript对象表示的Model，我们初始化为 `{ name: '', summary: '', content: '' }`；

methods：View可以触发的JavaScript函数，`submit` 就是提交表单时触发的函数。

接下来，我们在 `<form>` 标签中，用几个简单的 `v-model`，就可以让Vue把Model和View关联起来：

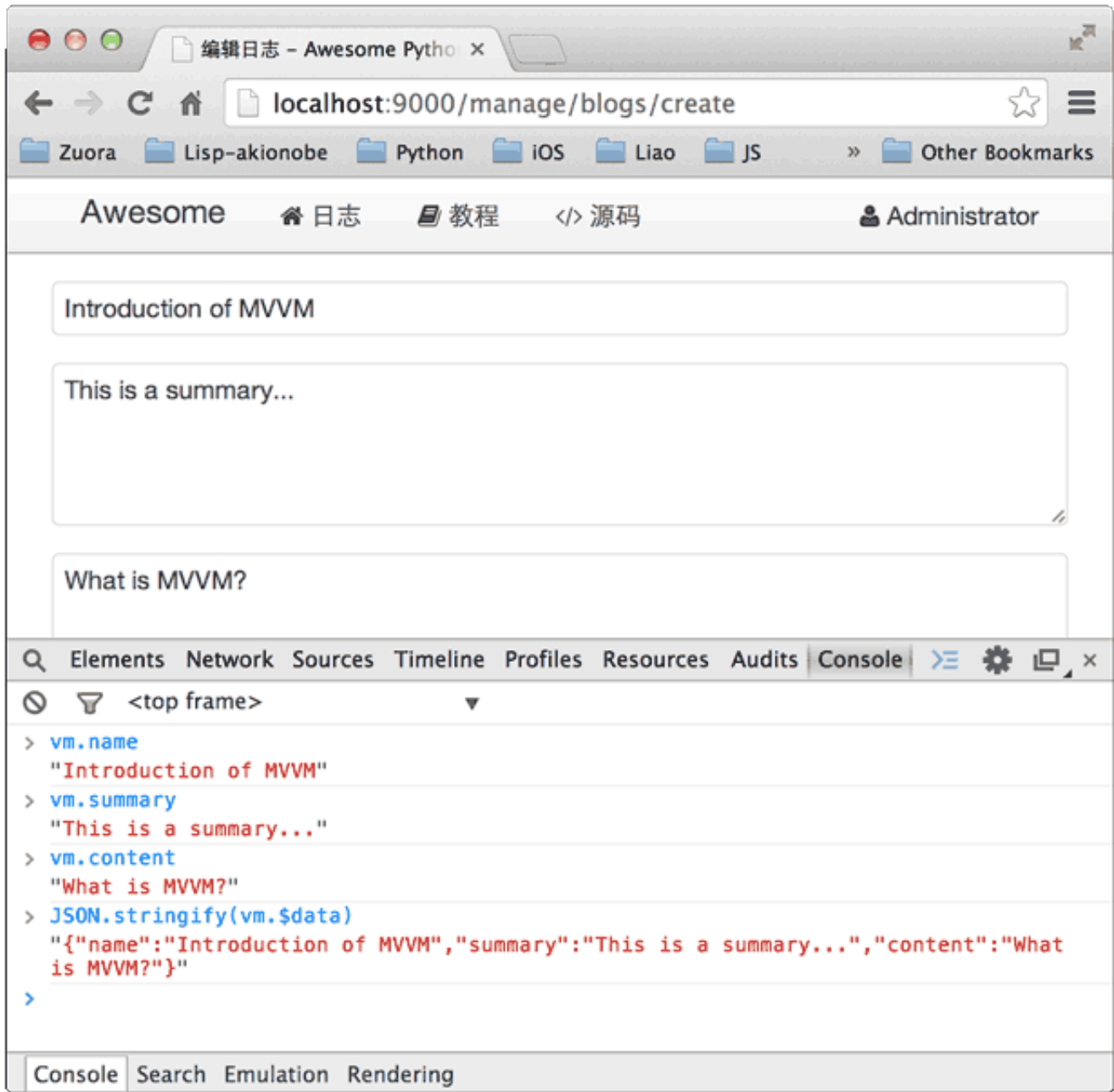
```

<!-- input的value和Model的name关联起来了 -->
<input v-model="name" class="uk-width-1-1">

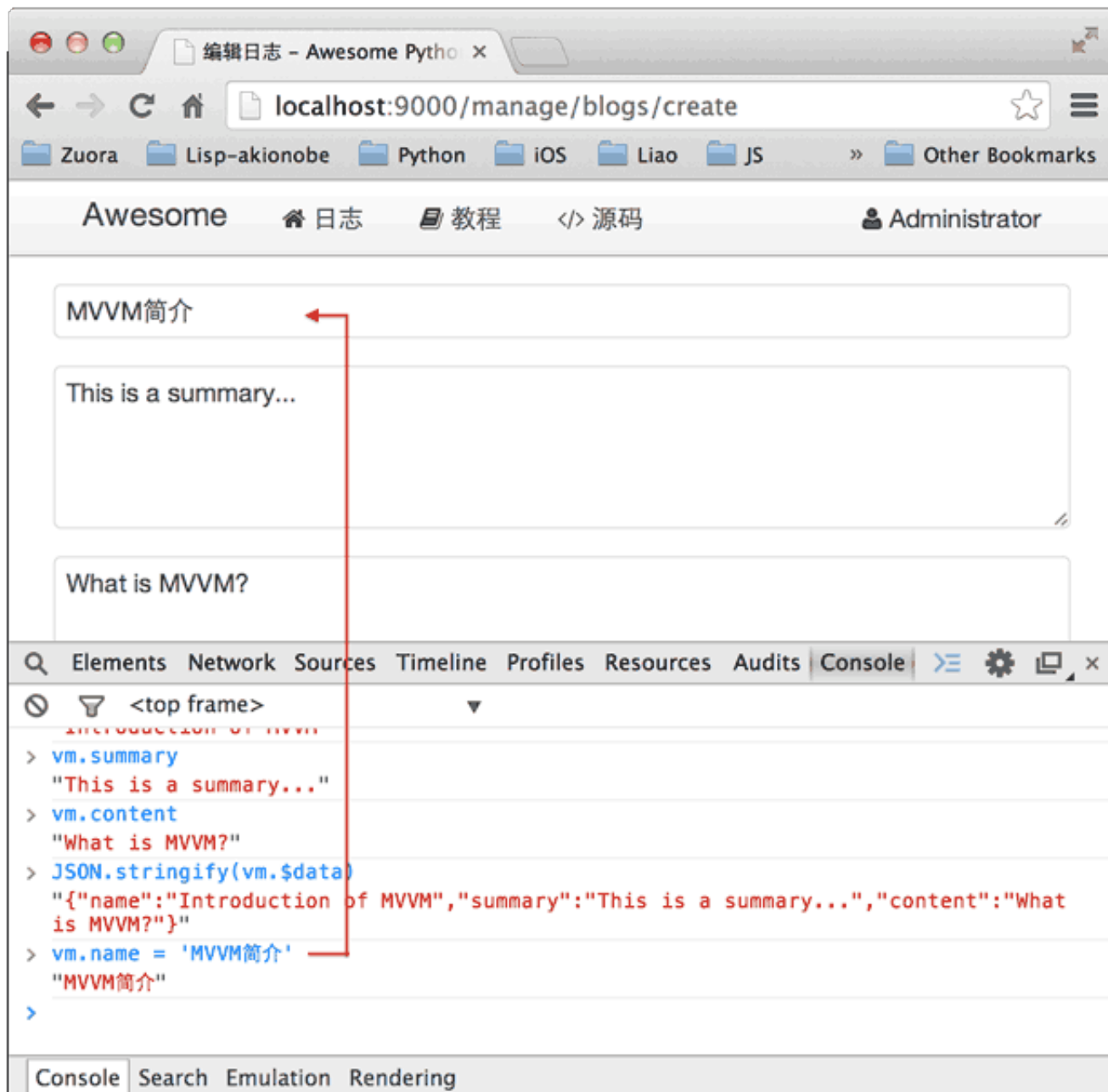
```

Form表单通过 `<form v-on="submit: submit">` 把提交表单的事件关联到 `submit` 方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过 `vm.name` 访问单个属性，或者通过 `vm.$data` 访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入 `vm.name = 'MVVM简介'`，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

参考源码

day-11

12、Day 12 - 编写日志列表页

MVVM模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来：

在 `apis.py` 中定义一个 `Page` 类用于存储分页信息：

```
class Page(object):
```

```
def __init__(self, item_count, page_index=1, page_size=10):
    self.item_count = item_count
    self.page_size = page_size
    self.page_count = item_count // page_size + (1 if item_count % page_size >
0 else 0)
    if (item_count == 0) or (page_index > self.page_count):
        self.offset = 0
        self.limit = 0
        self.page_index = 1
    else:
        self.page_index = page_index
        self.offset = self.page_size * (page_index - 1)
        self.limit = self.page_size
    self.has_next = self.page_index < self.page_count
    self.has_previous = self.page_index > 1

def __str__(self):
    return 'item_count: %s, page_count: %s, page_index: %s, page_size: %s, offs
et: %s, limit: %s' % (self.item_count, self.page_count, self.page_index, self.page_
size, self.offset, self.limit)

__repr__ = __str__
```

在 `handlers.py` 中实现API:

```
@get('/api/blogs')
def api_blogs(*, page='1'):
    page_index = get_page_index(page)
    num = yield from Blog.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, blogs=())
    blogs = yield from Blog.findAll(orderBy='created_at desc', limit=(p.offset, p.l
imit))
    return dict(page=p, blogs=blogs)
```

管理页面:

```
@get('/manage/blogs')
def manage_blogs(*, page='1'):
    return {
        '__template__': 'manage_blogs.html',
```

```
'page_index': get_page_index(page)
}
```

模板页面首先通过API: `GET /api/blogs?page=?` 拿到Model:

```
{
  "page": {
    "has_next": true,
    "page_index": 1,
    "page_count": 2,
    "has_previous": false,
    "item_count": 12
  },
  "blogs": [...]
}
```

然后, 通过Vue初始化MVVM:

```
<script>
function initVM(data) {
  var vm = new Vue({
    el: '#vm',
    data: {
      blogs: data.blogs,
      page: data.page
    },
    methods: {
      edit_blog: function (blog) {
        location.assign('/manage/blogs/edit?id=' + blog.id);
      },
      delete_blog: function (blog) {
        if (confirm('确认要删除"' + blog.name + '"? 删除后不可恢复! ')) {
          postJSON('/api/blogs/' + blog.id + '/delete', function (err, r)
{
            if (err) {
              return alert(err.message || err.error || err);
            }
            refresh();
          });
        }
      }
    }
  });
};
```

```

    $('#vm').show();
}
$(function() {
    getJSON('/api/blogs', {
        page: {{ page_index }}
    }, function (err, results) {
        if (err) {
            return fatal(err);
        }
        $('#loading').hide();
        initVM(results);
    });
});
</script>

```

View的容器是 `#vm`，包含一个table，我们用 `v-repeat` 可以把Model的数组 `blogs` 直接变成多行的 `<tr>`：

```

<div id="vm" class="uk-width-1-1">
    <a href="/manage/blogs/create" class="uk-button uk-button-primary"><i class="uk
-icon-plus"></i> 新日志</a>

    <table class="uk-table uk-table-hover">
        <thead>
            <tr>
                <th class="uk-width-5-10">标题 / 摘要</th>
                <th class="uk-width-2-10">作者</th>
                <th class="uk-width-2-10">创建时间</th>
                <th class="uk-width-1-10">操作</th>
            </tr>
        </thead>
        <tbody>
            <tr v-repeat="blog: blogs" >
                <td>
                    <a target="_blank" v-attr="href: '/blog/'+blog.id" v-text="blog
.name"></a>
                </td>
                <td>
                    <a target="_blank" v-attr="href: '/user/'+blog.user_id" v-text=
"blog.user_name"></a>
                </td>
                <td>
                    <span v-text="blog.created_at.toDateTime()"></span>
                </td>
                <td>
                    <a href="#0" v-on="click: edit_blog(blog)"><i class="uk-icon-ed

```

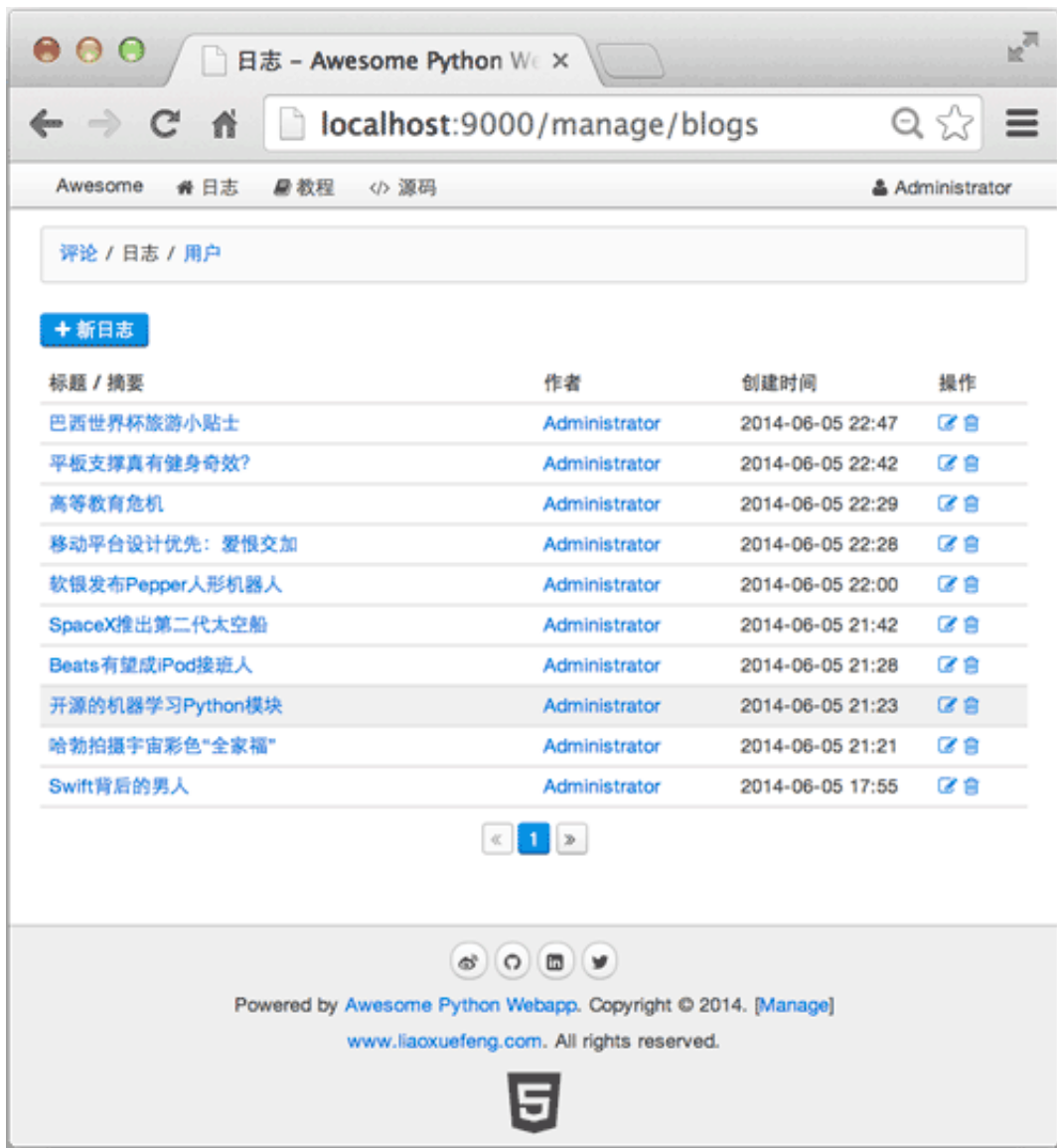
```
it"></i>
                <a href="#0" v-on="click: delete_blog(blog)"><i class="uk-icon-
trash-o"></i>
            </td>
        </tr>
    </tbody>
</table>

<div v-component="pagination" v-with="page"></div>
</div>
```

往Model的 `blogs` 数组中增加一个Blog元素，table就神奇地增加了一行；把 `blogs` 数组的某个元素删除，table就神奇地减少了一行。所有复杂的Model-View的映射逻辑全部由MVVM框架完成，我们只需要在HTML中写上 `v-repeat` 指令，就什么都不用管了。

可以把 `v-repeat="blog: blogs"` 看成循环代码，所以，可以在一个 `<tr>` 内部引用循环变量 `blog` 。 `v-text` 和 `v-attr` 指令分别用于生成文本和DOM节点属性。

完整的Blog列表页如下：



参考源码

day-12

13、Day 13 - 提升开发效率

现在，我们已经把一个Web App的框架完全搭建好了，从后端的API到前端的MVVM，流程已经跑通了。

在继续工作前，注意到每次修改Python代码，都必须在命令行先Ctrl-C停止服务器，再重启，改动才能生效。

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测 `www` 目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序 `pymonitor.py`，让它启动 `wsgiapp.py`，并时刻监控 `www` 目录下的代码改动，有改动时，先把当前 `wsgiapp.py` 进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库 `watchdog` 可以利用操作系统的API来监控目录文件的变化，并发送通知。我们先用 `pip` 安装：

```
$ pip3 install watchdog
```

利用 `watchdog` 接收文件变化的通知，如果是 `.py` 文件，就自动重启 `wsgiapp.py` 进程。

利用Python自带的 `subprocess` 实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

__author__ = 'Michael Liao'

import os, sys, time, subprocess

from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

def log(s):
    print('[Monitor] %s' % s)

class MyFileSystemEventHandler(FileSystemEventHandler):

    def __init__(self, fn):
        super(MyFileSystemEventHandler, self).__init__()
        self.restart = fn

    def on_any_event(self, event):
        if event.src_path.endswith('.py'):
            log('Python source file changed: %s' % event.src_path)
            self.restart()

command = ['echo', 'ok']
```

```

process = None

def kill_process():
    global process
    if process:
        log('Kill process [%s]...' % process.pid)
        process.kill()
        process.wait()
        log('Process ended with code %s.' % process.returncode)
        process = None

def start_process():
    global process, command
    log('Start process %s...' % ' '.join(command))
    process = subprocess.Popen(command, stdin=sys.stdin, stdout=sys.stdout, stderr=
sys.stderr)

def restart_process():
    kill_process()
    start_process()

def start_watch(path, callback):
    observer = Observer()
    observer.schedule(MyFileSystemEventHandler(restart_process), path, recursive=True)
    observer.start()
    log('Watching directory %s...' % path)
    start_process()
    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()

if __name__ == '__main__':
    argv = sys.argv[1:]
    if not argv:
        print('Usage: ./pymonitor your-script.py')
        exit(0)
    if argv[0] != 'python3':
        argv.insert(0, 'python3')
    command = argv
    path = os.path.abspath('.')
    start_watch(path, None)

```

一共70行左右的代码，就实现了Debug模式的自动重新加载。用下面的命令启动服务器：


```
$ python3 pymonitor.py wsgiapp.py
```

或者给 `pymonitor.py` 加上可执行权限，启动服务器：

```
$ ./pymonitor.py app.py
```

在编辑器中打开一个 `.py` 文件，修改后保存，看看命令行输出，是不是自动重启了服务器：

```
$ ./pymonitor.py app.py
[Monitor] Watching directory /Users/michael/Github/awesome-python3-webapp/www...
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-python3-webapp/www) will start
at 0.0.0.0:9000...
[Monitor] Python source file changed: /Users/michael/Github/awesome-python-webapp/w
ww/handlers.py
[Monitor] Kill process [2747]...
[Monitor] Process ended with code -9.
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-python3-webapp/www) will start
at 0.0.0.0:9000...
```

现在，只要一保存代码，就可以刷新浏览器看到效果，大大提升了开发效率。

14、Day 14 - 完成Web App

在Web App框架和基本流程跑通后，剩下的工作全部是体力活了：在Debug开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：

把当前用户绑定到 `request` 上，并对URL `/manage/` 进行拦截，检查当前用户是否是管理员身份：

```
@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' % (request.method, request.path))
```

```
request.__user__ = None
cookie_str = request.cookies.get(COOKIE_NAME)
if cookie_str:
    user = yield from cookie2user(cookie_str)
    if user:
        logging.info('set current user: %s' % user.email)
        request.__user__ = user
    if request.path.startswith('/manage/') and (request.__user__ is None or not
request.__user__.admin):
        return web.HTTPFound('/signin')
    return (yield from handler(request))
return auth
```

后端API包括：

- 获取日志：GET /api/blogs
- 创建日志：POST /api/blogs
- 修改日志：POST /api/blogs/:blog_id
- 删除日志：POST /api/blogs/:blog_id/delete
- 获取评论：GET /api/comments
- 创建评论：POST /api/blogs/:blog_id/comments
- 删除评论：POST /api/comments/:comment_id/delete
- 创建新用户：POST /api/users
- 获取用户：GET /api/users

管理页面包括：

- 评论列表页：GET /manage/comments
- 日志列表页：GET /manage/blogs
- 创建日志页：GET /manage/blogs/create
- 修改日志页：GET /manage/blogs/
- 用户列表页：GET /manage/users

用户浏览页面包括：

- 注册页：GET /register
- 登录页：GET /signin
- 注销页：GET /signout
- 首页：GET /
- 日志详情页：GET /blog/:blog_id

把所有的功能实现，我们第一个Web App就宣告完成！

参考源码

day-14

15、Day 15 - 部署Web App

作为一个合格的开发者，在本地环境下完成开发还远远不够，我们需要把Web App部署到远程服务器上，这样，广大用户才能访问到网站。

很多做开发的同学把部署这件事情看成是运维同学的工作，这种看法是完全错误的。首先，最近流行DevOps理念，就是说，开发和运维要变成一个整体。其次，运维的难度，其实跟开发质量有很大的关系。代码写得垃圾，运维再好也架不住天天挂掉。最后，DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务？那就得在开发时考虑到这一点。

下面，我们就来把awesome-python3-webapp部署到Linux服务器。

搭建Linux服务器

要部署到Linux，首先得有一台Linux服务器。要在公网上体验的同学，可以在Amazon的AWS申请一台EC2虚拟机（免费使用1年），或者使用国内的一些云服务器，一般都提供Ubuntu Server的镜像。想在本地部署的同学，请安装虚拟机，推荐使用VirtualBox。

我们选择的Linux服务器版本是Ubuntu Server 14.04 LTS，原因是apt太简单了。如果你准备使用其他Linux版本，也没有问题。

Linux安装完成后，请确保ssh服务正在运行，否则，需要通过apt安装：

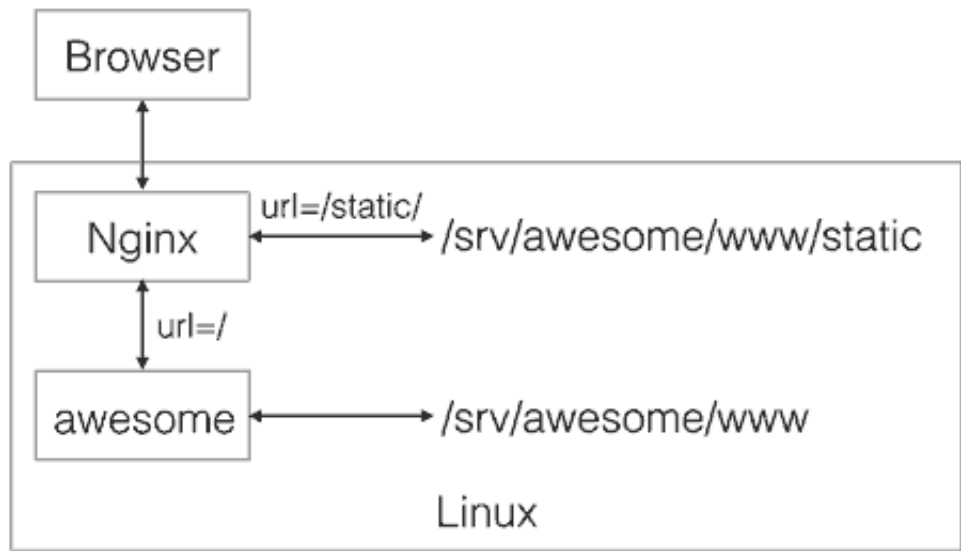
```
$ sudo apt-get install openssh-server
```

有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户

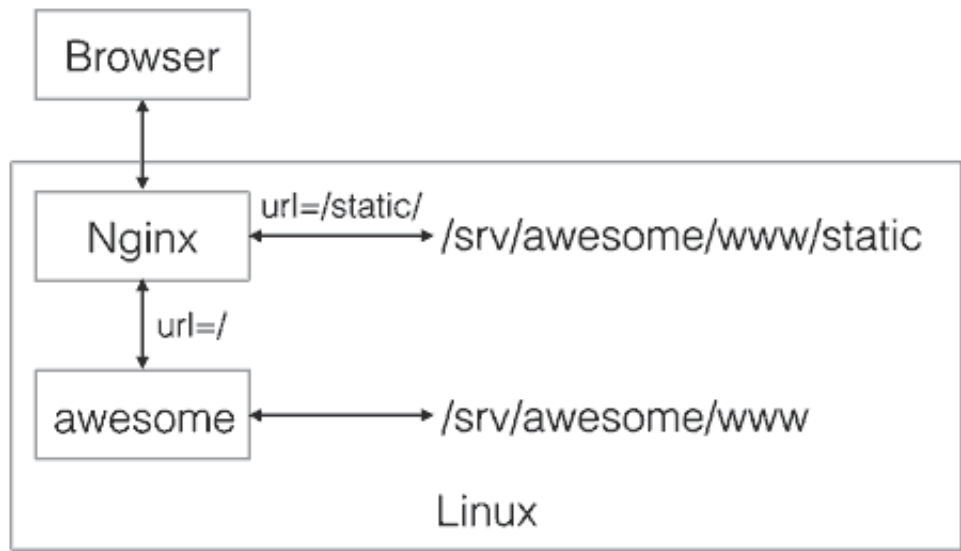
的 `.ssh/authorized_keys` 中，这样，就可以通过证书实现无密码连接。

部署方式

利用Python自带的asyncio，我们已经编写了一个异步高性能服务器。但是，我们还需要一个高性能的Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。这个模型如下：



Nginx负责分发请求：



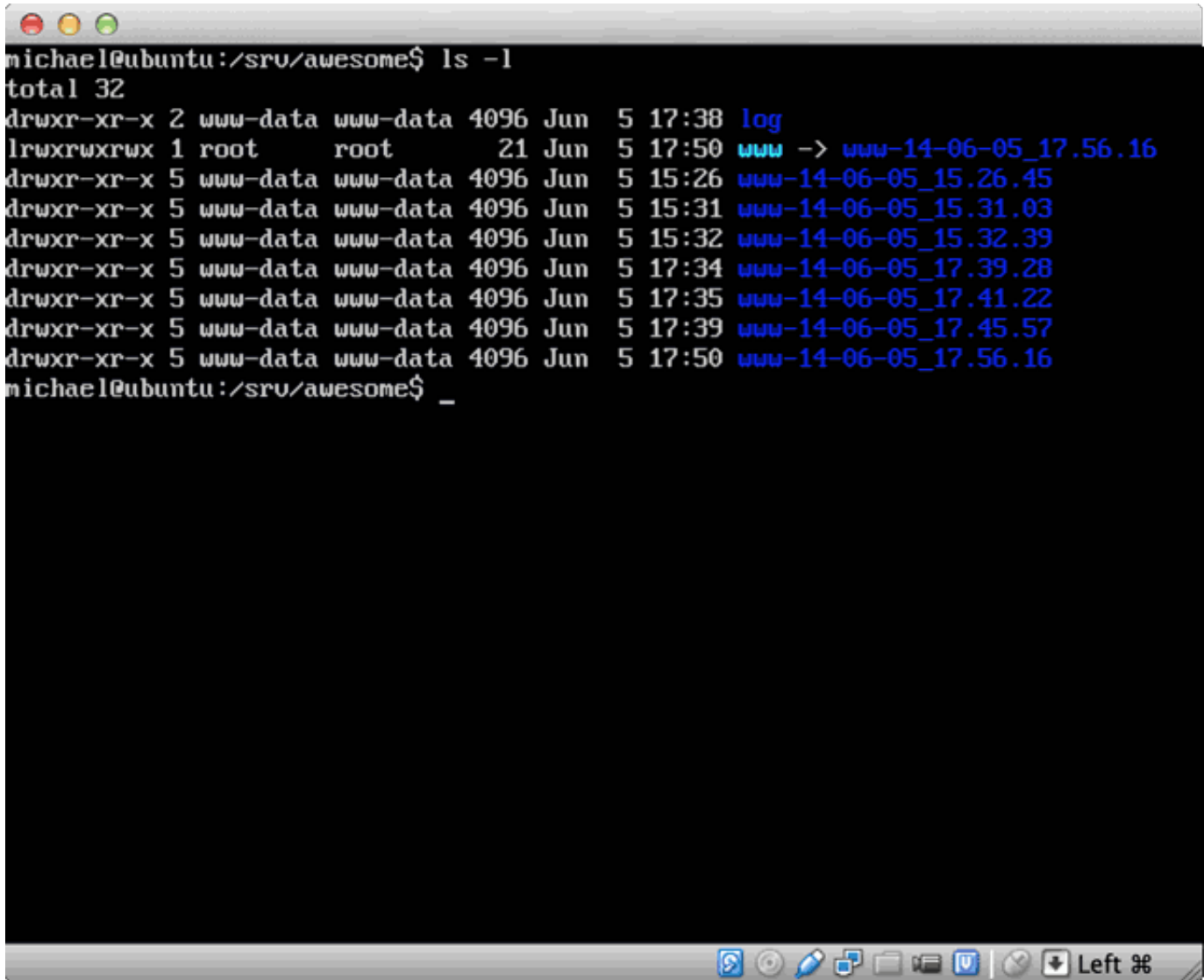
在服务器端，我们需要定义好部署的目录结构：

```

/
+- srv/
  +- awesome/      <-- Web App根目录
    +- www/        <-- 存放Python源码
      | +- static/  <-- 存放静态资源文件
      +- log/       <-- 存放Log

```

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把 `www` 作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本：



```
michael@ubuntu:/srv/awesome$ ls -l
total 32
drwxr-xr-x 2 www-data www-data 4096 Jun  5 17:38 log
lrwxrwxrwx 1 root      root      21 Jun  5 17:50 www -> www-14-06-05_17.56.16
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:26 www-14-06-05_15.26.45
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:31 www-14-06-05_15.31.03
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:32 www-14-06-05_15.32.39
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:34 www-14-06-05_17.39.28
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:35 www-14-06-05_17.41.22
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:39 www-14-06-05_17.45.57
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:50 www-14-06-05_17.56.16
michael@ubuntu:/srv/awesome$ _
```

而Nginx和python代码的配置文件只需要指向 `www` 目录即可。

Nginx可以作为服务进程直接启动，但 `app.py` 还不行，所以，[Supervisor](#)登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

- Nginx：高性能Web服务器+负责反向代理；
- Supervisor：监控服务进程的工具；
- MySQL：数据库服务。

在Linux服务器上[用apt](#)可以直接安装上述服务：

```
$ sudo apt-get install nginx supervisor python3 mysql-server
```

然后，再把我们自己的Web App用到的Python库安装了：

```
$ sudo pip3 install jinja2 aiomysql aiohttp
```

在服务器上创建目录 `/srv/awesome/` 以及相应的子目录。

在服务器上初始化MySQL数据库，把数据库初始化脚本 `schema.sql` 复制到服务器上执行：

```
$ mysql -u root -p < schema.sql
```

服务器端准备就绪。

部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。[Fabric](#)就是一个自动化部署工具。由于Fabric是用Python 2.x开发的，所以，部署脚本要用Python 2.7来编写，本机还必须安装Python 2.7版本。

要用Fabric部署，需要在本机（是开发机器，不是Linux服务器）安装Fabric：

```
$ easy_install fabric
```

Linux服务器上不需要安装Fabric，Fabric使用SSH直接登录服务器并执行部署命令。

下一步是编写部署脚本。Fabric的部署脚本叫 `fabfile.py`，我们把它放到 `awesome-python-webapp` 的目录下，与 `www` 目录平级：

```
awesome-python-webapp/  
+- fabfile.py
```

```
+ - www/
+ - ...
```

Fabric的脚本编写很简单，首先导入Fabric的API，设置部署时的变量：

```
# fabfile.py
import os, re
from datetime import datetime

# 导入Fabric API:
from fabric.api import *

# 服务器登录用户名:
env.user = 'michael'
# sudo用户为root:
env.sudo_user = 'root'
# 服务器地址，可以有多个，依次部署:
env.hosts = ['192.168.0.3']

# 服务器MySQL用户名和口令:
db_user = 'www-data'
db_password = 'www-data'
```

然后，每个Python函数都是一个任务。我们先编写一个打包的任务：

```
_TAR_FILE = 'dist-awesome.tar.gz'

def build():
    includes = ['static', 'templates', 'transwarp', 'favicon.ico', '*.py']
    excludes = ['test', '.*', '*.pyc', '*.pyo']
    local('rm -f dist/%s' % _TAR_FILE)
    with lcd(os.path.join(os.path.abspath('.'), 'www')):
        cmd = ['tar', '--dereference', '-czvf', '../dist/%s' % _TAR_FILE]
        cmd.extend(['--exclude=%s' % ex for ex in excludes])
        cmd.extend(includes)
        local(' '.join(cmd))
```

Fabric提供 `local('...')` 来运行本地命令，`with lcd(path)` 可以把当前命令的目录设定为 `lcd()` 指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要Cygwin环境。

在 `awesome-python-webapp` 目录下运行：

```
$ fab build
```

看看是否在 `dist` 目录下创建了 `dist-awesome.tar.gz` 的文件。

打包后，我们就可以继续编写 `deploy` 任务，把打包文件上传至服务器，解压，重置 `www` 软链接，重启相关服务：

```
_REMOTE_TMP_TAR = '/tmp/%s' % _TAR_FILE
_REMOTE_BASE_DIR = '/srv/awesome'

def deploy():
    newdir = 'www-%s' % datetime.now().strftime('%y-%m-%d_%H.%M.%S')
    # 删除已有的tar文件:
    run('rm -f %s' % _REMOTE_TMP_TAR)
    # 上传新的tar文件:
    put('dist/%s' % _TAR_FILE, _REMOTE_TMP_TAR)
    # 创建新目录:
    with cd(_REMOTE_BASE_DIR):
        sudo('mkdir %s' % newdir)
    # 解压到新目录:
    with cd('%s/%s' % (_REMOTE_BASE_DIR, newdir)):
        sudo('tar -xzf %s' % _REMOTE_TMP_TAR)
    # 重置软链接:
    with cd(_REMOTE_BASE_DIR):
        sudo('rm -f www')
        sudo('ln -s %s www' % newdir)
        sudo('chown www-data:www-data www')
        sudo('chown -R www-data:www-data %s' % newdir)
    # 重启Python服务和nginx服务器:
    with settings(warn_only=True):
        sudo('supervisorctl stop awesome')
        sudo('supervisorctl start awesome')
        sudo('/etc/init.d/nginx reload')
```

注意 `run()` 函数执行的命令是在服务器上运行，`with cd(path)` 和 `with lcd(path)` 类似，把当前目录在服务器端设置为 `cd()` 指定的目录。如果一个命令需要 `sudo` 权限，就不能用 `run()`，而是用 `sudo()` 来执行。

配置Supervisor

上面让Supervisor重启awesome的命令会失败，因为我们还没有配置Supervisor呢。

编写一个Supervisor的配置文件 `awesome.conf`，存放到 `/etc/supervisor/conf.d/` 目录下：

`[program:awesome]`

```
command      = /srv/awesome/www/app.py
directory    = /srv/awesome/www
user         = www-data
startsecs    = 3

redirect_stderr      = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups  = 10
stdout_logfile        = /srv/awesome/log/app.log
```

配置文件通过 `[program:awesome]` 指定服务名为 `awesome`，`command` 指定启动 `app.py`。

然后重启Supervisor后，就可以随时启动和停止Supervisor管理的 services 了：

```
$ sudo supervisorctl reload
$ sudo supervisorctl start awesome
$ sudo supervisorctl status
awesome                                RUNNING    pid 1401, uptime 5:01:34
```

配置Nginx

Supervisor只负责运行 `app.py`，我们还需要配置Nginx。把配置文件 `awesome` 放到 `/etc/nginx/sites-available/` 目录下：

```
server {
    listen      80; # 监听80端口

    root        /srv/awesome/www;
    access_log  /srv/awesome/log/access_log;
    error_log   /srv/awesome/log/error_log;

    # server_name awesome.liaoxuefeng.com; # 配置域名

    # 处理静态文件/favicon.ico:
    location /favicon.ico {
        root /srv/awesome/www;
    }
}
```

```
# 处理静态资源:
location ~ ^/static/.*$ {
    root /srv/awesome/www;
}

# 动态请求转发到9000端口:
location / {
    proxy_pass          http://127.0.0.1:9000;
    proxy_set_header    X-Real-IP $remote_addr;
    proxy_set_header    Host $host;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
}
}
```

然后在 `/etc/nginx/sites-enabled/` 目录下创建软链接:

```
$ pwd
/etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/awesome .
```

让Nginx重新加载配置文件, 不出意外, 我们的 `awesome-python3-webapp` 应该正常运行:

```
$ sudo /etc/init.d/nginx reload
```

如果有任何错误, 都可以在 `/srv/awesome/log` 下查找Nginx和App本身的log。如果Supervisor启动时报错, 可以在 `/var/log/supervisor` 下查看Supervisor的log。

如果一切顺利, 你可以在浏览器中访问Linux服务器上的 `awesome-python3-webapp` 了:



如果在开发环境更新了代码，只需要在命令行执行：

```
$ fab build
$ fab deploy
```

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

友情链接

嫌国外网速慢的童鞋请移步网易和搜狐的镜像站点：

<http://mirrors.163.com/>

<http://mirrors.sohu.com/>

参考源码

[day-15](#)

16、Day 16 - 编写移动App

网站部署上线后，还缺点啥呢？

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动App，出门根本不好意思跟人打招呼。

所以，`awesome-python3-webapp` 必须得有一个移动App版本！

开发iPhone版本

我们首先来看看如何开发iPhone App。前置条件：一台Mac电脑，安装XCode和最新的iOS SDK。

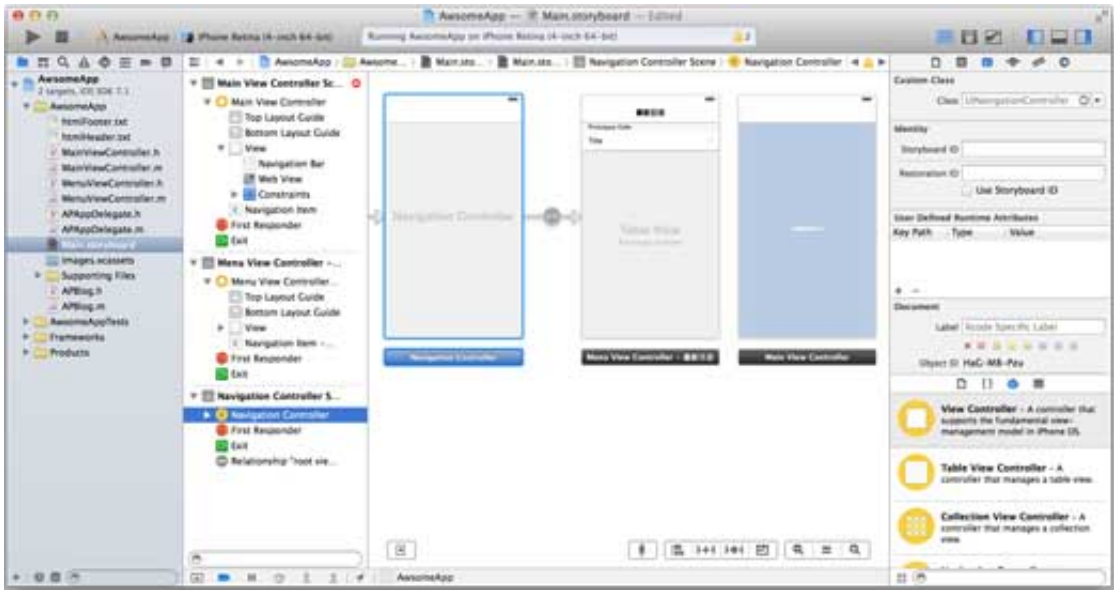
在使用MVVM编写前端页面时，我们就能感受到，用REST API封装网站后台的功能，不但能清晰地分离前端页面和后台逻辑，现在这个好处更加明显，移动App也可以通过REST API从后端拿到数据。

我们来设计一个简化版的iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容：



只需要调用API：`/api/blogs`。

在XCode中完成App编写：



由于我们的教程是Python，关于如何开发iOS，请移步[Develop Apps for iOS](#)。

[点击下载iOS App源码](#)。

如何编写Android App? 这个当成作业了。

参考源码

[day-16](#)

四、FAQ

常见问题

本节列出常见的一些问题。

如何获取当前路径

当前路径可以用 `'.'` 表示，再用 `os.path.abspath()` 将其转换为绝对路径：

```
# -*- coding:utf-8 -*-
# test.py

import os

print(os.path.abspath('.'))
```

运行结果：

```
$ python3 test.py
/Users/michael/workspace/testing
```

如何获取当前模块的文件名

可以通过特殊变量 `__file__` 获取：

```
# -*- coding:utf-8 -*-
# test.py

print(__file__)
```

输出：

```
$ python3 test.py
test.py
```

如何获取命令行参数

可以通过 `sys` 模块的 `argv` 获取：

```
# -*- coding:utf-8 -*-
# test.py

import sys

print(sys.argv)
```

输出：

```
$ python3 test.py -a -s "Hello world"
['test.py', '-a', '-s', 'Hello world']
```

`argv` 的第一个元素永远是命令行执行的 `.py` 文件名。

如何获取当前Python命令的可执行文件路径

sys 模块的 `executable` 变量就是Python命令可执行文件的路径：

```
# -*- coding:utf-8 -*-
# test.py

import sys

print(sys.executable)
```

在Mac下的结果：

```
$ python3 test.py
/usr/local/opt/python3/bin/python3.4
```

五、期末总结

终于到了期末总结的时刻了！

经过一段时间的学习，相信你对Python已经初步掌握。一开始，可能觉得Python上手很容易，可是越往后学，会越困难，有的时候，发现理解不了代码，这时，不妨停下来思考一下，先把概念搞清楚，代码自然就明白了。

Python非常适合初学者用来进入计算机编程领域。Python属于非常高级的语言，掌握了这门高级语言，就对计算机编程的核心思想——抽象有了初步理解。如果希望继续深入学习计算机编程，可以学习Java、C、JavaScript、Lisp等不同类型的语言，只有多掌握不同领域的语言，有比较才更有收获。

