Bottle 中文文档

Release 0.11.dev

Marcel Hellkamp
译者: 小朋光年

October 10, 2012

Contents

Bottle 是一个快速，简单，轻量级的 Python WSGI Web 框架。单一文件，只依赖 Python 标准库 。

- **URL 映射 (Routing):** 将 URL 请求映射到 Python 函数，使 URL 更简洁。
- **模板 (Templates):** 快速且 pythonic 的内置模板引擎 ，同时支持 mako, jinja2 和 cheetah 等模板。
- **基础功能 (Utilities):** 方便地访问表单数据，上传文件，使用 cookie，查看 HTTP 元数据。
- **开发服务器 (Server):** 内置了开发服务器，且支持 paste, fapws3 , bjoern, Google App Engine, cherrypy 等符合 WSGI 标准的 HTTP 服务器。

示例: "Hello World"

```
from bottle import route, run

@route('/hello/:name')
def index(name='World'):
    return '<b>Hello %s!</b>' % name

run(host='localhost', port=8080)
```

将其保存为 py 文件并执行，用浏览器访问 http://localhost:8080/hello/world 即可看到效果。就这么简单！

下载和安装

通过 PyPi (`easy_install -U bottle`) 安装最新的稳定版，或下载 bottle.py (开发版) 到你的项目文件夹。Bottle 除了 Python 标准库无任何依赖 [1] ，同时支持 **Python 2.5+ and 3.x** 。

[1] 如果使用了第三方的模板或 HTTP 服务器，则需要安装相应的第三方模块。

CHAPTER 1

用户指南

如果你有将 Bottle 用于 Web 开发的打算，请继续看下去。如果这份文档没有解决你遇到的问题，尽管在 邮件列表 中吼出来吧（译者注：用英文哦）。

1.1 教程

这份教程将向你介绍 Bottle 的开发理念和功能特性。既介绍 Bottle 的基本用法，也包含了进阶用法。你可以从头到尾通读一遍，也可当做开发时的参考。你也许对自动生成的 *API Reference*(不译) 感兴趣。它包含了更多的细节，但解释没有这份教程详细。在秘诀 或常见问题 可找到常见问题的解决办法。如果需要任何帮助，可加入我们的 邮件列表 或在 IRC 频道 和我们交流。

1.1.1 安装

Bottle 不依赖其他库，你需要做的仅是下载 bottle.py (开发版) 到你的项目文件夹，然后开始写代码。

```
$ wget http://bottlepy.org/bottle.py
```

在终端运行以上命令，即可下载到 Bottle 的最新开发版，包含了所有新功能特性。如果更需要稳定性，你应该坚持使用 Bottle 的稳定版本。可在 PyPi 下载稳定版本，然后通过 **pip** (推荐), **easy_install** 或你的包管理软件安装。

```
$ sudo pip install bottle             # 推荐
$ sudo easy_install bottle            # 若无 pip，可尝试这个
$ sudo apt-get install python-bottle  # 适用于 debian, ubuntu, ...
```

你需要 Python 2.5 或以上版本（包括 3.x）来运行 bottle 应用。如果你没有管理员权限或你不想将 Bottle 安装为整个系统范围内可用，可先创建一个 virtualenv：

```
$ virtualenv develop              # 创建一个虚拟环境
$ source develop/bin/activate     # 使用虚拟环境里的 Python 解析器
(develop)$ pip install -U bottle  # 在虚拟环境中安装 Bottle
```

如果还未安装 virtualenv:

```
$ wget https://raw.github.com/pypa/virtualenv/master/virtualenv.py
$ python virtualenv.py develop    # 创建一个虚拟环境
$ source develop/bin/activate     # 使用虚拟环境里的 Python 解析器
(develop)$ pip install -U bottle  # 在虚拟环境中安装 Bottle
```

1.1.2 Quickstart: “Hello World”

到目前为止，我假设你已经安装 好了 bottle 或已将 bottle.py 拷贝到你的项目文件夹。接下来我们就可以写一个非常简单的”Hello World” 了:

```
from bottle import route, run

@route('/hello')
def hello():
    return "Hello World!"

run(host='localhost', port=8080, debug=True)
```

就这么简单！保存为 py 文件并执行，用浏览器访问 http://localhost:8080/hello 就可以看到”Hello World!”。它的执行流程大致如下:

`route()` 函数将一段代码绑定到一个 URL，在这个例子中, 我们将 `hello()` 函数绑定给了 `/hello` 。我们称之为 *route* (也是该修饰器的函数名) ，这是 Bottle 框架最重要的开发理念。你可以根据需要定义任意多的 *route* 。在浏览器请求一个 URL 的时候，框架自动调用与之相应的函数，接着将函数的返回值发送给浏览器。就这么简单！

最后一行调用的 `run()` 函数启动了内置的开发服务器。它监听 *localhost* 的 8080 端口并响应请求，`Control-c` 可将其关闭。到目前为止，这个内置的开发服务器已经足够用于日常的开发测试了。它根本不需要安装，就可以让你的应用跑起来。在教程的后面，你将学会如何让你的应用跑在其他服务器上面 (译者注：内置服务器不能满足生产环境的要求)

调试模式 在早期开发的时候非常有用，但请务必记得，在生产环境中将其关闭。

毫无疑问，这是一个十分简单例子，但它展示了用 Bottle 做应用开发的基本理念。接下来你将了解到其他开发方式。

默认应用

基于简单性考虑，这份教程中的大部分例子都使用一个模块层面的 `route()` 修饰器函数来定义 route。这样的话，所有 route 都添加到了一个全局的“默认应用”里面，即是在第一次调用 `route()` 函数时，创建的一个 `Bottle` 类的实例。其他几个模块层面的修饰器函数都与这个“默认应用”有关，如果你偏向于面向对象的做法且不介意多打点字，你可以创建一个独立的应用对象，这样就可避免使用全局范围的“默认应用”。

```
from bottle import Bottle, run

app = Bottle()

@app.route('/hello')
def hello():
    return "Hello World!"

run(app, host='localhost', port=8080)
```

接下来的默认应用 章节中将更详细地介绍这种做法。现在，你只需知道不止有一种选择就好了。

1.1.3 URL 映射

在上一章中，我们实现了一个十分简单的 web 应用，只有一个 URL 映射 (route)。让我们再来看一下“Hello World”中与 routing 有关的部分。

```
@route('/hello')
def hello():
    return "Hello World!"
```

route() 函数将一个 URL 路径与一个回调函数关联了起来，然后在 *default application* 中添加了一个 URL 映射 (route)。但只有一个 route 的应用未免太无聊了，让我们试着再添加几个 route 吧。:

```
@route('/')
@route('/hello/<name>')
def greet(name='Stranger'):
    return 'Hello %s, how are you?' % name
```

这个例子说明了两件事情，一个回调函数可绑定多个 route，你也可以在 URL 中添加通配符，然后在回调函数中使用它们。

动态 URL 映射

包含通配符的 route，我们称之为动态 route(与之对应的是静态 route)，它能匹配多个 URL 地址。一个通配符包含在一对尖括号里面 (像这样 <name>)，通配符之间用"/" 分隔开来。如果我们将 URL 定义为 /hello/<name> 这样，那么它就能匹配 /hello/alice 和 /hello/bob 这样的浏览器请求，但不能匹配 /hello , /hello/ 和 /hello/mr/smith 。

URL 中的通配符都会当作参数传给回调函数，直接在回调函数中使用。这样可以漂亮地实现 RESTful 形式的 URL。例子如下:

```
@route('/wiki/<pagename>')            # 匹配 /wiki/Learning_Python
def show_wiki_page(pagename):
    ...

@route('/<action>/<user>')            # 匹配 /follow/defnull
def user_api(action, user):
    ...
```

从 0.10 版本开始，过滤器 (Filter) 可被用来定义特殊类型的通配符，在传通配符给回调函数之前，先自动转换通配符类型。包含过滤器的通配符定义一般像 <name:filter> 或 <name:filter:config> 这样。config 部分是可选的，其语法由你使用的过滤器决定。

已实现下面几种形式的过滤器，后续可能会继续添加:

- **:int** 匹配一个数字，自动将其转换为 int 类型。
- **:float** 与:int 类似，用于浮点数。
- **:path** 匹配一个路径 (包含"/")
- **:re** 匹配 config 部分的一个正则表达式，不更改被匹配到的值

让我们来看看具体的使用例子

```
@route('/object/<id:int>')
def callback(id):
    assert isinstance(id, int)

@route('/show/<name:re:[a-z]+>')
def callback(name):
    assert name.isalpha()

@route('/static/<path:path>')
def callback(path):
    return static_file(path, ...)
```

你可以添加自己的过滤器。详见 *URL* 映射 。(译者注：参考 Bottle 的 Router 类，源码 300 行左右)

从 **Bottle 0.10** 版本开始，可以用新的语法来在 URL 中定义通配符，更简单了。新旧语法之间的对比如下:

旧	新
:name	<name>
:name#regexp#	<name:re:regexp>
:#regexp#	<:re:regexp>
:##	<:re>

请尽可能在新项目中使用新的语法。虽然现在依然兼容旧语法，但终究会将其废弃的。

HTTP 请求方法

HTTP 协议定义了多个 请求方法 来满足不同的需求。所有 route 默认使用 GET 方法，只响应 GET 请求。可给 `route()` 函数指定 `method` 参数。或用 `get()` ，`post()` ，`put()` 或 `delete()` 等函数来代替 `route()` 函数。

POST 方法一般用于 HTML 表单的提交。下面是一个使用 POST 来实现用户登录的例子:

```
from bottle import get, post, request

@get('/login') # 或 @route('/login')
def login_form():
    return '''<form method="POST" action="/login">
                <input name="name"     type="text" />
                <input name="password" type="password" />
                <input type="submit" />
              </form>'''

@post('/login') # 或 @route('/login', method='POST')
def login_submit():
    name     = request.forms.get('name')
    password = request.forms.get('password')
    if check_login(name, password):
        return "<p> 登录成功 </p>"
    else:
        return "<p> 登录失败 </p>"
```

在这个例子中，`/login` 绑定了两个回调函数，一个回调函数响应 GET 请求，一个回调函数响应 POST 请求。如果浏览器使用 GET 请求访问 `/login` ，则调用 login_form() 函数来返回登录页面，浏览器使用 POST 方法提交表单后，调用 login_submit() 函数来检查用户有效性，并返回登录结果。接下来的请求数据 *(Request Data)* 章节中，会详细介绍 `Request.forms` 的用法。

特殊请求方法: HEAD 和 ANY

HEAD 方法类似于 GET 方法，但服务器不会返回 HTTP 响应正文，一般用于获取 HTTP 原数据而不用下载整个页面。Bottle 像处理 GET 请求那样处理 HEAD 请求，但是会自动去掉 HTTP 响应正文。你无需亲自处理 HEAD 请求。

另外，非标准的 ANY 方法做为一个低优先级的 fallback：在没有其它 route 的时候，监听 ANY 方法的 route 会匹配所有请求，而不管请求的方法是什么。这对于用做代理的 route 很有用，可将所有请求都重定向给子应用。

总而言之：HEAD 请求被响应 GET 请求的 route 来处理，响应 ANY 请求的 route 处理所有请求，但仅限于没有其它 route 来匹配原先的请求的情况。就这么简单。

静态文件映射

Bottle 不会处理像图片或 CSS 文件的静态文件请求。你需要给静态文件提供一个 route，一个回调函数 (用于查找和控制静态文件的访问)。

```
from bottle import static_file
@route('/static/<filename>')
def server_static(filename):
    return static_file(filename, root='/path/to/your/static/files')
```

:func:`static_file` 函数用于响应静态文件的请求。（详见 :ref:`tutorial-static-files` ）这个例子只能响应在 ``/path/t

因为 <filename> 这样的通配符定义不能匹配一个路径 (路径中包含"/")。为了响应子目录下的文件请求，我们需要更改 *path* 过滤器的定义。

```
@route('/static/<filepath:path>')
def server_static(filepath):
    return static_file(filepath, root='/path/to/your/static/files')
```

使用 root='./static/files' 这样的相对路径的时候，请注意当前工作目录 (./) 不一定是项目文件夹。

错误页面

如果出错了，Bottle 会显示一个默认的错误页面，提供足够的 debug 信息。你也可以使用 error() 函数来自定义你的错误页面:

```
from bottle import error
@error(404)
def error404(error):
    return 'Nothing here, sorry'
```

从现在开始，在遇到 404 错误的时候，将会返回你在上面自定义的页面。传给 error404 函数的唯一参数，是一个 HTTPError 对象的实例。除此之外，这个回调函数与我们用来响应普通请求的回调函数没有任何不同。你可以从 request 中读取数据，往 response 中写入数据和返回所有支持的数据类型，除了 HTTPError 的实例。

只有在你的应用返回或 raise 一个 HTTPError 异常的时候 (就像 abort() 函数那样)，处理 Error 的函数才会被调用。更改 Request.status 或返回 HTTPResponse 不会触发错误处理函数。

1.1.4 生成内容

在纯 WSGI 环境里，你的应用能返回的内容类型相当有限。应用必须返回一个 iterable 的字节型字符串。你可以返回一个字符串 (因为字符串是 iterable 的)，但这会导致服务器按字符来传输你的内容。Unicode 字符串根本不允许。这不是很实用。

Bottle 支持返回更多的内容类型，更具弹性。它甚至能在合适的情况下，在 HTTP 头中添加 *Content-Length* 字段和自动转换 unicode 编码。下面列出了所有你能返回的内容类型，以及框架处理方式的一个简述。

字典类型: 上面已经提及，Python 中的字典类型 (或其子类) 会被自动转换为 JSON 字符串。返回给浏览器的时候，HTTP 头的 Content-Type 字段被自动设置为 " application/json" 。可十分简单地实现基于 JSON 的 API。Bottle 同时支持 json 之外的数据类型，详见 *tutorial-output-filter* 。

空字符串: False , None 或其它非真值: 输出为空，Content-Length 设为 0。

Unicode 字符串: Unicode 字符串 (or iterables yielding unicode strings) 被自动转码，Content-Type 被默认设置为 utf8，接着视之为普通字符串 (见下文)。

字节型字符串: Bottle 将字符串当作一个整体来返回 (而不是按字符来遍历)，并根据字符串长度添加 Content-Length 字段。包含字节型字符串的列表先被合并。其它 iterable 的字节型字符串不会被合并，因为它们也许太大来，耗内存。在这种情况下，Content-Length 字段不会被设置。

(译者注：这段翻译不顺畅，保留原文) Bottle returns strings as a whole (instead of iterating over each char) and adds a Content-Length header based on the string length. Lists of byte strings are joined first. Other iterables yielding byte strings are not joined because they may grow too big to fit into memory. The Content-Length header is not set in this case.

HTTPError 或 HTTPResponse 的实例: 返回它们和直接 raise 出来有一样的效果。对于 HTTPError 来说，会调用错误处理程序。详见错误页面 。

文件对象: 任何有 .read() 方法的对象都被当成一个 file-like 对象来对待，会被传给 WSGI Server 框架定义的 wsgi.file_wrapper callable 对象来处理。一些 WSGI Server 实现会利用优化过的系统调用 (sendfile) 来更有效地传输文件，另外就是分块遍历。可选的 HTTP 头，例如 Content-Length 和 Content-Type 不会被自动设置。尽可能使用 send_file() 。详见静态文件 。

Iterables and generators: 你可以在回调函数中使用 yield 语句，或返回一个 iterable 的对象，只要该对象返回的是字节型字符串，unicode 字符串，HTTPError 或 HTTPResponse 实例。不支持嵌套 iterable 对象，不好意思。注意，在 iterable 对象返回第一个非空值的时候，就会把 HTTP 状态码和 HTTP 头发送给浏览器。稍后再更改它们就起不到什么作用了。

以上列表的顺序非常重要。在你返回一个 str 类的子类的时候，即使它有 .read() 方法，它依然会被当成一个字符串对待，而不是文件，因为字符串先被处理。

改变默认编码

Bottle 使用 Content-Type 的 *charset* 参数来决定编码 unicode 字符串的方式。默认的 Content-Type 是 text/html;charset=UTF8 ，可在 Response.content_type 属性中修改，或直接设置 Response.charset 的值。关于 Response 对象的介绍，详见 *Response* 对象 。

```
from bottle import response
@route('/iso')
def get_iso():
    response.charset = 'ISO-8859-15'
    return u'This will be sent with ISO-8859-15 encoding.'

@route('/latin9')
def get_latin():
    response.content_type = 'text/html; charset=latin9'
    return u'ISO-8859-15 is also known as latin9.'
```

在极少情况下，Python 中定义的编码名字和 HTTP 标准中的定义不一样。这样，你就必须同时修改 Response.content_type` (发送给客户端的) 和设置 Response.charset 属性 (用于编码 unicode)。

静态文件

你可直接返回文件对象。但我们更建议你使用 static_file() 来提供静态文件服务。它会自动猜测文件的 mime-type，添加 Last-Modified 头，将文件路径限制在一个 root 文件夹下面来保证安全，且返回合适的 HTTP 状态码 (由于权限不足导致的 401 错误，由于文件不存在导致的 404 错误)。它甚至支持 If-Modified-Since 头，如果文件未被修改，则直接返回 304 Not Modified 。你可指定 MIME 类型来避免其自动猜测。

```
from bottle import static_file
@route('/images/<filename:re:.*\.png>#')
def send_image(filename):
    return static_file(filename, root='/path/to/image/files', mimetype='image/png')

@route('/static/<filename:path>')
def send_static(filename):
    return static_file(filename, root='/path/to/static/files')
```

如果确实需要，你可将 static_file() 的返回值当作异常 raise 出来。

强制下载

大多数浏览器在知道 MIME 类型的时候，会尝试直接调用相关程序来打开文件 (例如 PDF 文件)。如果你不想这样，你可强制浏览器只是下载该文件，甚至提供文件名。:

```
@route('/download/<filename:path>')
def download(filename):
    return static_file(filename, root='/path/to/static/files', download=filename)
```

如果 download 参数的值为 True ，会使用原始的文件名。

HTTP 错误和重定向

abort() 函数是生成 HTTP 错误页面的一个捷径。

```
from bottle import route, abort
@route('/restricted')
def restricted():
    abort(401, "Sorry, access denied.")
```

为了将用户访问重定向到其他 URL，你在 Location 中设置新的 URL，接着返回一个 303 See Other 。redirect() 函数可以帮你做这件事情。

```
from bottle import redirect
@route('/wrong/url')
def wrong():
    redirect("/right/url")
```

你可以在第二个参数中提供另外的 HTTP 状态码。

Note: 这两个函数都会抛出 HTTPError 异常，终止回调函数的执行。

其他异常

除了 HTTPResponse 或 HTTPError 以外的其他异常，都会导致 500 错误，所以不会造成 WSGI 服务器崩溃。你将 bottle.app().catchall 的值设为 False 来关闭这种行为，以便在你的中间件中处理异常。

Response 对象

诸如 HTTP 状态码，HTTP 响应头，用户 cookie 等元数据都保存在一个名字为 response 的对象里面，接着被传输给浏览器。你可直接操作这些元数据或使用一些更方便的函数。在 API 章节可查到所有相关 API(详见 Response)，这里主要介绍一些常用方法。

响应头

Cache-Control 和 Location 之类的响应头通过 Response.set_header() 来定义。这个方法接受两个参数，一个是响应头的名字，一个是它的值，名字是大小写敏感的。

```
@route('/wiki/<page>')
def wiki(page):
    response.set_header('Content-Language', 'en')
    ...
```

大多数的响应头是唯一的，meaning that only one header per name is send to the client。一些特殊的响应头在一次 response 中允许出现多次。使用 Response.add_header() 来添加一个额外的响应头，而不是 Response.set_header() 。

```
response.set_header('Set-Cookie', 'name=value')
response.add_header('Set-Cookie', 'name2=value2')
```

请注意，这只是一个例子。如果你想使用 cookie，详见 *ahead* 。

Cookies

Cookie 是储存在浏览器配置文件里面的一小段文本。你可通过 `Request.get_cookie()` 来访问已存在的 Cookie，或通过 `Response.set_cookie()` 来设置新的 Cookie。

```
@route('/hello')
def hello_again():
    if request.get_cookie("visited"):
        return "Welcome back! Nice to see you again"
    else:
        response.set_cookie("visited", "yes")
        return "Hello there! Nice to meet you"
```

`Response.set_cookie()` 方法接受一系列额外的参数，来控制 Cookie 的生命周期及行为。一些常用的设置如下:

- **max_age:** 最大有效时间，以秒为单位 (默认: `None`)
- **expires:** 一个 datetime 对象或一个 UNIX timestamp (默认: `None`)
- **domain:** 可访问该 Cookie 的域名 (默认: 当前域名)
- **path:** 限制 cookie 的访问路径 (默认: `/`)
- **secure:** 只允许在 HTTPS 链接中访问 cookie (默认: off)
- **httponly:** 防止客户端的 javascript 读取 cookie (默认: off, 要求 python 2.6 或以上版本)

如果 *expires* 和 *max_age* 两个值都没设置，cookie 会在当前的浏览器 session 失效或浏览器窗口关闭后失效。在使用 cookie 的时候，应该注意一下几个陷阱。

- 在大多数浏览器中，cookie 的最大容量为 4KB。
- 一些用户将浏览器设置为不接受任何 cookie。大多数搜索引擎也忽略 cookie。确保你的应用在无 cookie 的时候也能工作。
- cookie 被储存在客户端，也没被加密。你在 cookie 中储存的任何数据，用户都可以读取。更坏的情况下，cookie 会被攻击者通过 XSS 偷走，一些已知病毒也会读取浏览器的 cookie。既然如此，就不要在 cookie 中储存任何敏感信息。
- cookie 可以被伪造，不要信任 cookie！

Cookie 签名

上面提到，cookie 容易被客户端伪造。Bottle 可通过加密 cookie 来防止此类攻击。你只需在读取和设置 cookie 的时候，通过 *secret* 参数来提供一个密钥。如果 cookie 未签名或密钥不匹配，`Request.get_cookie()` 方法返回 `None`

```
@route('/login')
def login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_user_credentials(username, password):
        response.set_cookie("account", username, secret='some-secret-key')
        return "Welcome %s! You are now logged in." % username
    else:
        return "Login failed."

@route('/restricted')
def restricted_area():
    username = request.get_cookie("account", secret='some-secret-key')
    if username:
        return "Hello %s. Welcome back." % username
    else:
        return "You are not logged in. Access denied."
```

例外，Bottle 自动序列化储存在签名 cookie 里面的数据。你可在 cookie 中储存任何可序列化的对象 (不仅仅是字符串)，只要对象大小不超过 4KB。

> **Warning:** 签名 cookie 在客户端不加密 (译者注：即在客户端没有经过二次加密)，也没有写保护 (客户端可使用之前的 cookie)。给 cookie 签名的主要意义在于在 cookie 中存储序列化对象和防止伪造 cookie，依然不要在 cookie 中存储敏感信息。

1.1.5 请求数据 (Request Data)

Bottle 通过 `request` 对象来提供对 HTTP 相关元数据的访问，例如 cookie，HTTP 头，通过 POST 方法提交的表单。只要在回调函数中使用，该对象总是包含当前这次浏览器请求的信息。即使在多线程环境中，多个请求同时被响应，这依然有效。关于全局对象是如何做到线程安全的，详见 `contextlocal`.

Note: Bottle 在 `FormsDict` 类的一个实例里 (译者注：参见源码 1763 行左右)，存储大多数解析后的 HTTP 元数据。其行为和普通的字典类似，但有一些额外的特性: 所有在字典中的值都可像访问类的属性那样访问。这类虚拟的属性总是返回一个 unicode 字符串，如果缺失该属性，将返回一个空字符串。

`FormsDict` 类继承自 `MultiDict` 类，一个 key 可存储多个值。标准的字典访问方法只会返回一个值，但可通过 `MultiDict.getall()` 方法来获取一个包含所有值的 list (或许为空)。

在 API 的章节，有关于 API 和特性的完整描述，这里我们只谈常见的用例和特性。

Cookies

cookie 存储在 `BaseRequest.cookie` 中，是 `FormsDict` 类的一个实例。`BaseRequest.get_cookie()` 方法提供了对签名 *cookie* 的访问。下面是一个基于 cookie 的计数器例子。

```
from bottle import route, request, response
@route('/counter')
def counter():
    count = int( request.cookies.get('counter', '0') )
    count += 1
    response.set_cookie('counter', str(count))
    return 'You visited this page %d times' % count
```

HTTP 头

所有发送到客户端的 HTTP 头 (例如 `Referer`, `Agent` 和 `Accept-Language`) 存储在一个 `WSGIHeaderDict` 中，可通过 `BaseRequest.headers` 访问。`WSGIHeaderDict` 是一个字典，其 key 大小写敏感。

```
from bottle import route, request
@route('/is_ajax')
def is_ajax():
    if request.headers.get('X-Requested-With') == 'XMLHttpRequest':
        return 'This is an AJAX request'
    else:
        return 'This is a normal request'
```

查询变量

查询字符串 (例如 `/forum?id=1&page=5` 中的 id=5 和 page=5) 一般用于向服务器传输键值对。你可通过 `BaseRequest.query` (`FormsDict` 类的实例) 来访问，和通过 `BaseRequest.query_string` 来获取整个字符串。

```
from bottle import route, request, response
@route('/forum')
def display_forum():
    forum_id = request.query.id
    page = request.query.page or '1'
    return 'Forum ID: %s (page %s)' % (forum_id, page)
```

POST 表单数据和文件上传

通过 POST 和 PUT 方式提交的请求也许会包含经过编码的表单数据。BaseRequest.forms 字典包含了解析过后的文本字段，BaseRequest.files 字典存储上传的文件，BaseRequest.POST 同时包含两者。这三个都是 FormsDict 类的实例，在你使用它们的时候才会被创建。上传的文件保存为一个包含其他元数据的 cgi.FieldStorage 对象。最后，你可像访问文件一样，通过 BaseRequest.body 访问未经处理的请求内容。

下面是一个简单的文件上传例子。

```
<form action="/upload" method="post" enctype="multipart/form-data">
  <input type="text" name="name" />
  <input type="file" name="data" />
</form>
```

```
from bottle import route, request
@route('/upload', method='POST')
def do_upload():
    name = request.forms.name
    data = request.files.data
    if name and data and data.file:
        raw = data.file.read() # This is dangerous for big files
        filename = data.filename
        return "Hello %s! You uploaded %s (%d bytes)." % (name, filename, len(raw))
    return "You missed a field."
```

Unicode issues

在 **Python2** 中，所有的 key 和 value 都是 byte-string。如果你需要 unicode，可使用 FormsDict.getunicode() 方法或像访问属性那样访问。这两种方法都试着将字符串转码 (默认: utf8)，如果失败，将返回一个空字符串。无需捕获 UnicodeError 异常。

```
>>> request.query['city']
'G\xc3\xb6ttingen'  # A utf8 byte string
>>> request.query.city
u'Gttingen'         # The same string as unicode
```

在 **Python3** 中，所有的字符串都是 unicode。但 HTTP 是基于字节的协议，在 byte-string 被传给应用之前，服务器必须将其转码。安全起见，WSGI 协议建议使用 ISO-8859-1 (即是 latin1)，一个可反转的单字节编码，可被转换为其他编码。Bottle 通过 FormsDict.getunicode() 和属性访问实现了转码，但不支持字典形式的访问。通过字典形式的访问，将直接返回服务器返回的字符串，未经处理，这或许不是你想要的。

```
>>> request.query['city']
'Gttingen' # An utf8 string provisionally decoded as ISO-8859-1 by the server
>>> request.query.city
'Gttingen'  # The same string correctly re-encoded as utf8 by bottle
```

如果你整个字典包含正确编码后的值 (e.g. for WTForms)，可通过 FormsDict.decode() 方法来获取一个转码后的拷贝 (译者注：一个新的实例)。

WSGI 环境

每一个 BaseRequest 类的实例都包含一个 WSGI 环境的字典。最初存储在 BaseRequest.environ 中，但 request 对象也表现的像一个字典。大多数有用的数据都通过特定的方法或属性暴露了出来，但如果你想直接访问 WSGI 环境变量，可以这样做。

```
@route('/my_ip')
def show_ip():
    ip = request.environ.get('REMOTE_ADDR')
    # or ip = request.get('REMOTE_ADDR')
    # or ip = request['REMOTE_ADDR']
    return "Your IP is: %s" % ip
```

1.1.6 模板

Bottle 内置了一个快速的，强大的模板引擎，称为 *SimpleTemplate* 模板引擎 。可通过 template() 函数或 view() 修饰器来渲染一个模板。只需提供模板的名字和传递给模板的变量。下面是一个简单的例子。

```
@route('/hello')
@route('/hello/<name>')
def hello(name='World'):
    return template('hello_template', name=name)
```

这会加载 hello_template.tpl 模板文件，并提供 name 变量。默认情况，Bottle 会在 ./views/ 目录查找模板文件 (译者注：或当前目录)。可在 bottle.TEMPLATE_PATH 这个列表中添加更多的模板路径。

view() 修饰器允许你在回调函数中返回一个字典，并将其传递给模板，和 template() 函数做同样的事情。

```
@route('/hello')
@route('/hello/<name>')
@view('hello_template')
def hello(name='World'):
    return dict(name=name)
```

语法

模板语法类似于 Python 的语法。它要确保语句块的正确缩进，所以你在写模板的时候无需担心会出现缩进问题。详细的语法描述可看 *SimpleTemplate* 模板引擎 。

简单的模板例子

```
%if name == 'World':
    <h1>Hello {{name}}!</h1>
    <p>This is a test.</p>
%else:
    <h1>Hello {{name.title()}}!</h1>
    <p>How are you?</p>
%end
```

缓存

模板在经过编译后被缓存在内存里。你在修改模板文件后，要调用 bottle.TEMPLATES.clear() 函数清除缓存才能看到效果。在 debug 模式下，缓存被禁用了，无需手动清除缓存。

1.1.7 插件

New in version 0.9. Bottle 的核心功能覆盖了常见的使用情况，但是作为一个迷你框架，它有它的局限性。所以我们引入了插件机制，插件可以给框架添加其缺少的功能，集成第三方的库，或是自动化一些重复性的工作。

我们有一个不断增长的可用插件列表 插件列表，大多数插件都被设计为可插拔的。有很大可能，你的问题已经被解决，而且已经有现成的插件可以使用了。如果没有现成的插件，插件开发指南 有介绍如何开发一个插件。

插件扮演着各种各样的角色。例如，SQLitePlugin 插件给每个 route 的回调函数都添加了一个 db 参数，在回调函数被调用的时候，会新建一个数据库连接。这样，使用数据库就非常简单了。

```
from bottle import route, install, template
from bottle_sqlite import SQLitePlugin

install(SQLitePlugin(dbfile='/tmp/test.db'))

@route('/show/<post_id:int>')
def show(db, post_id):
    c = db.execute('SELECT title, content FROM posts WHERE id = ?', (post_id,))
    row = c.fetchone()
    return template('show_post', title=row['title'], text=row['content'])

@route('/contact')
def contact_page():
    ''' 这个回调函数不需要数据库连接。因为没有"db"参数，sqlite 插件会完全忽略这个回调函数。 '''
    return template('contact')
```

其它插件或许在线程安全的 local 对象里面发挥作用，改变 request 对象的细节，过滤回调函数返回的数据或完全绕开回调函数。举个例子，一个用于登录验证的插件会在调用原先的回调函数响应请求之前，验证用户的合法性，如果是非法访问，则返回登录页面而不是调用回调函数。具体的做法要看插件是如何实现的。

整个应用的范围内安装插件

可以在整个应用的范围内安装插件，也可以只是安装给某些 route。大多数插件都可安全地安装给所有 route，也足够智能，可忽略那些并不需要它们的 route。

让我们拿 SQLitePlugin 插件举例，它只会影响到那些需要数据库连接的 route，其它 route 都被忽略了。正因为如此，我们可以放心地在整个应用的范围内安装这个插件。

调用 install() 函数来安装一个插件

```
from bottle_sqlite import SQLitePlugin
install(SQLitePlugin(dbfile='/tmp/test.db'))
```

插件没有马上应用到所有 route 上面，它被延迟执行来确保没有遗漏任何 route。你可以先安装插件，再添加 route。有时，插件的安装顺序很重要，如果另外一个插件需要连接数据库，那么你就需要先安装操作数据库的插件。

卸载插件

调用 uninstall() 函数来卸载已经安装的插件

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')
install(sqlite_plugin)

uninstall(sqlite_plugin) # 卸载指定插件
uninstall(SQLitePlugin)  # 卸载这种类型的插件
uninstall('sqlite')      # 卸载名字是 sqlite 的插件
uninstall(True)          # 马上卸载所有插件
```

在任何时候，插件都可以被安装或卸载，即使是在服务器正在运行的时候。一些小技巧应用到了这个特征，例如在需要的时候安装一些供 debug 和性能测试的插件，但不可滥用这个特性。每一次安装或卸载插件的时候，route 缓存都会被刷新，所有插件被重新加载。

Note: 模块层面的 `install()` 和 `unistall()` 函数会影响默认应用 。针对应用来管理插件，可使用 Bottle 应用对象的相应方法。

安装给特定的 route

`route()` 修饰器的 `apply` 参数可以给指定的 route 安装插件

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')

@route('/create', apply=[sqlite_plugin])
def create(db):
    db.execute('INSERT INTO ...')
```

插件黑名单

如果你想显式地在一些 route 上面禁用某些插件，可使用 `route()` 修饰器的 `skip` 参数。

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')
install(sqlite_plugin)

@route('/open/<db>', skip=[sqlite_plugin])
def open_db(db):
    # 这次，插件不会修改'db' 参数了
    if db in ('test', 'test2'):
        # The plugin handle can be used for runtime configuration, too.
        sqlite_plugin.dbfile = '/tmp/%s.db' % db
        return "Database File switched to: /tmp/%s.db" % db
    abort(404, "No such database.")
```

`skip` 参数接受单一的值或是一个 list。你可使用插件的名字，类，实例来指定你想要禁用的插件。如果 `skip` 的值为 True，则禁用所有插件。

插件和子应用

大多数插件只会影响到安装了它们的应用。因此，它们不应该影响通过 `Bottle.mount()` 方法挂载上来的子应用。这里有一个例子。

```
root = Bottle()
root.mount('/blog', apps.blog)

@root.route('/contact', template='contact')
def contact():
    return {'email': 'contact@example.com'}

root.install(plugins.WTForms())
```

在你挂载一个应用的时候，Bottle 在主应用上面创建一个代理 route，将所有请求转接给子应用。在代理 route 上，默认禁用了插件。如上所示，我们的 `WTForms` 插件影响了 `/contact` route，但不会影响挂载在 root 上面的 `/blog` 。

这个是一个合理的行为，但可被改写。下面的例子，在指定的代理 route 上面应用了插件。

```
root.mount('/blog', apps.blog, skip=None)
```

这里存在一个小难题: 插件会整个子应用当作一个 route 看待，即是上面提及的代理 route。如果想在子应用的每个 route 上面应用插件，你必须显式地在子应用上面安装插件。

1.1.8 开发

到目前为止，你已经学到一些开发的基础，并想写你自己的应用了吧？这里有一些小技巧可提高你的生产力。

默认应用

Bottle 维护一个全局的 `Bottle` 实例的栈，模块层面的函数和修饰器使用栈顶实例作为默认应用。例如 `route()` 修饰器，相当于在默认应用上面调用了 `Bottle.route()` 方法。

```
@route('/')
def hello():
    return 'Hello World'
```

对于小应用来说，这样非常方便，可节约你的工作量。但这同时意味着，在你的模块导入的时候，你定义的 route 就被安装到全局的默认应用中了。为了避免这种模块导入的副作用，Bottle 提供了另外一种方法，显式地管理应用。

```
app = Bottle()

@app.route('/')
def hello():
    return 'Hello World'
```

分离应用对象，大大提高了可重用性。其他开发者可安全地从你的应用中导入 `app` 对象，然后通过 `Bottle.mount()` 方法来合并到其它应用中。

作为一种选择，你可通过应用栈来隔离你的 route，依然使用方便的 `route` 修饰器。

```
default_app.push()

@route('/')
def hello():
    return 'Hello World'

app = default_app.pop()
```

`app()` 和 `default_app()` 都是 `AppStack` 类的一个实例，实现了栈形式的 API 操作。你可 push 应用到栈里面，也可将栈里面的应用 pop 出来。在你需要导入一个第三方模块，但它不提供一个独立的应用对象的时候，尤其有用。

```
default_app.push()

import some.module

app = default_app.pop()
```

调试模式

在开发的早期阶段，调试模式非常有用。

```
bottle.debug(True)
```

在调试模式下，当错误发生的时候，Bottle 会提供更多的调试信息。同时禁用一些可能妨碍你的优化措施，检查你的错误设置。

下面是调试模式下会发生改变的东西，但这份列表不完整:

- 默认的错误页面会打印出运行栈。
- 模板不会被缓存。
- 插件被马上应用。

请确保不要在生产环境中使用调试模式。

自动加载

在开发的时候，你需要不断地重启服务器来验证你最新的改动。自动加载功能可以替你做这件事情。在你编辑完一个模块文件后，它会自动重启服务器进程，加载最新版本的代码。

```
from bottle import run
run(reloader=True)
```

它的工作原理，主进程不会启动服务器，它使用相同的命令行参数，创建一个子进程来启动服务器。请注意，所有模块级别的代码都被执行了至少两次。

子进程中 `os.environ['BOOTLE_CHILD']` 变量的值被设为 `True` ，它运行一个不会自动加载的服务器。在代码改变后，主进程会终止掉子进程，并创建一个新的子进程。更改模板文件不会触发自动重载，请使用 debug 模式来禁用模板缓存。

自动加载需要终止子进程。如果你运行在 Windows 等不支持 `signal.SIGINT` (会在 Python 中 raise `KeyboardInterrupt` 异常) 的系统上，会使用 `signal.SIGTERM` 来杀掉子进程。在子进程被 `SIGTERM` 杀掉的时候，exit handlers 和 finally 等语句不会被执行。

命令行接口

从 0.10 版本开始，你可像一个命令行工具那样使用 Bottle:

```
$ python -m bottle

Usage: bottle.py [options] package.module:app

选项:
  -h, --help            显示帮助信息并退出
  --version             显示版本号
  -b ADDRESS, --bind=ADDRESS
                        绑定 socket 到 ADDRESS.
  -s SERVER, --server=SERVER
                        使用 SERVER 对应的后端服务器
  -p PLUGIN, --plugin=PLUGIN
                        安装插件
  --debug               启动到调试模式
  --reload              自动重载
```

ADDRESS 参数接受一个 IP 地址或 IP: 端口，其默认为 `localhost:8080` 。其它参数都很好地自我解释了。

插件和应用都通过一个导入表达式来指定。包含了导入的路径 (例如: `package.module`) 和模块命名空间内的一个表达式，两者用":" 分开。下面是一个简单例子，详见 `load()` 。

```
# 从'myapp.controller'模块中获取'app'对象
# 在 80 端口启动一个 paste 服务器，监听所有来源的请求
python -m bottle -server paste -bind 0.0.0.0:80 myapp.controller:app

# 启动一个自动重载的开发服务器，使用全局的默认应用。route 定义在'test.py'文件中
python -m bottle --debug --reload test

# 安装一个自定义的调试插件，赋予一些参数
python -m bottle --debug --reload --plugin 'utils:DebugPlugin(exc=True)'' test
```

```
# 加载一个通过'myapp.controller.make_app()'按需创建的应用
python -m bottle 'myapp.controller:make_app()''
```

1.1.9 部署

Bottle 默认运行在内置的 wsgiref 服务器上面。这个单线程的 HTTP 服务器在开发的时候特别有用，但其性能低下，在服务器负载不断增加的时候也许会是性能瓶颈。

最早的解决办法是让 Bottle 使用 paste 或 cherrypy 等多线程的服务器。

```
bottle.run(server='paste')
```

在 deployment 章节中，会介绍更多部署的选择。

1.1.10 词汇表

回调函数
Programmer code that is to be called when some external action happens.
In the context of web frameworks, the mapping between URL paths and
application code is often achieved by specifying a callback function
for each URL.

修饰器
A function returning another function, usually applied
as a function transformation using the ``@decorator`` syntax.
See `python documentation for function definition <http://docs.python.org/reference/compound_stmts.html#functio

环境 (environ)
A structure where information about all documents under the root is
saved, and used for cross-referencing. The environment is pickled
after the parsing stage, so that successive runs only need to read
and parse new and changed documents.

handler function
A function to handle some specific event or situation. In a web
framework, the application is developed by attaching a handler function
as callback for each specific URL comprising the application.

source directory
The directory which, including its subdirectories, contains all
source files for one Sphinx project.

1.2 URL 映射

Bottle 内置一个强大的 route 引擎，可以给每个浏览器请求找到正确的回调函数。*tutorial* 中已经介绍了一些基础知识。接下来是一些进阶知识和 route 的规则。

1.2.1 route 的语法

Router 类中明确区分两种类型的 route：静态 **route** (例如 /contact) 和 动态 **route** (例如 /hello/<name>)。包含了 通配符 的 route 即是动态 route，除此之外的都是静态的。Changed in version 0.10. 包含通配符，最简单的形式就是将其放到一对 <> 里面 (例如 <name>)。在同一个 route 里面，这个变量名需要是唯一的。因为稍后会将其当作参数传给回调函数，所以这个变量名的第一个字符应该是字母。

每一个通配符匹配一个或多个字符，直到遇到 / 。类似于 [^/]+ 这样一个正则表达式，确保在 route 包含多个通配符的时候不出现歧义。

/<action>/<item> 这个规则匹配的情况如下

:

Path	Result
/save/123	{'action': 'save', 'item': '123'}
/save/123/	不匹配
/save/	不匹配
//123	不匹配

你可通过过滤器来改变这一行为，稍后会介绍。

1.2.2 通配符过滤器

New in version 0.10. 过滤器被用于定义更特殊的通配符，可在 URL 中" 被匹配到的部分" 被传递给回调函数之前，处理其内容。可通过 <name:filter> 或 <name:filer:config> 这样的语句来声明一个过滤器。"config" 部分的语法由被使用的过滤器决定。

Bottle 中已实现以下过滤器：

- **:int** 匹配一个整形数，并将其转换为 int
- **:float** 同上，匹配一个浮点数
- **:path** 匹配所有字符，包括'/'
- **:re[:config]** 允许在 config 中写一个正则表达式

你可在 route 中添加自己写的过滤器。过滤器是一个有三个返回值的函数：一个正则表达式，一个 callable 的对象 (转换 URL 片段为 Python 对象)，另一个 callable 对象 (转换 Python 对象为 URL 片段)。过滤器仅接受一个参数，就是设置字符串 (译者注：例如 re 过滤器的 config 部分)。以下是一个过滤器的例子，将逗号分开的字符串转换为数字列表。

```
app = Bottle()

def list_filter(config):
    ''' 匹配这样的字符串:'1,2,3,4,5,6' '''
    delimiter = config or ','
    regexp = r'\d+(%s\d)*' % re.escape(delimiter)

    def to_python(match):
        return map(int, match.split(delimiter))

    def to_url(numbers):
        return delimiter.join(map(str, numbers))

    return regexp, to_python, to_url

app.router.add_filter('list', list_filter)

@app.route('/follow/<ids:list>')
def follow_users(ids):
    for id in ids:
        ...
```

1.2.3 旧语法

Changed in version 0.10. 在 **Bottle 0.10** 版本中引入了新的语法，来简单化一些常见用例，但依然兼容旧的语法。新旧语法的区别如下。

旧语法	新语法
:name	<name>
:name#regexp#	<name:re:regexp>
:#regexp#	<:re:regexp>
:##	<:re>

请尽量在新项目中避免使用旧的语法，虽然它现在还没被废弃，但终究会的。

1.2.4 route 的顺序 (URL 映射的顺序)

在通配符和正则表达式的支持下，我们是可以定义重叠的 route 的。如果多个 route 匹配同一个 URL，它们的行为也许会变得难以琢磨。为了了解这里面究竟发生了什么事情，你需要了解 route 被执行的顺序。(译者注：router 会检查 route 的顺序)

首先，route 是按照它们的规则来分组的。两个规则相同但回调函数不同的 route 分到同一组，第一个 route 决定这两个 route 的位置。如果两个 route 完全相同 (同样的规则，同样的回调函数)，则先定义的 route 会被后来的 route 替换掉，但它的位置位置会被保留下来。

(译者注：我搞不明白这里发生了什么，保留原文) First you should know that routes are grouped by their path rule. Two routes with the same path rule but different methods are grouped together and the first route determines the position of both routes. Fully identical routes (same path rule and method) replace previously defined routes, but keep the position of their predecessor.

基于性能考虑，默认先查找静态 route，这个默认设置可以被改掉。如果没有静态 route 匹配浏览器请求，则会按 route 被定义的顺序，去查找动态 route，只要找到一个匹配的动态 route，便不会继续查找。如果也找不到匹配请求的动态 route，则返回一个 404 错误。

第二步，会检查浏览器请求的 HTTP 方法。如果不能精确匹配，且浏览器发的是 HEAD 请求，那个 router 会按照 GET 方法去查找相应的 route，否则便按照 ANY 方法去查找 route。如果都失败了，则返回一个 405 错误。

下面这个例子也许会令你迷惑

```
@route('/<action>/<name>', method='GET')
@route('/save/<name>', method='POST')
```

第二个 route 永远不会被命中，因为第一个 route 已经能匹配该 URL，所以 router 会在查找到第一个 route 后停止查找，接着按照浏览器请求的 HTTP 方法来查找，如果找不到则返回 405 错误。

看起来挺复杂的，确实很复杂。这是为了性能而付出的代价。最好是避免创建容易造成歧义的 route。将来或许会改变 route 实现的细节，我们在慢慢改善中。

1.2.5 显式的 route 配置

route 修饰器也可以直接当作函数来调用。在复杂的部署中，这种方法或许更灵活，直接由你来控制“何时”及“如何”配置 route。

下面是一个简单的例子

```
def setup_routing():
    bottle.route('/', method='GET', index)
    bottle.route('/edit', method=['GET', 'POST'], edit)
```

实际上，bottle 可以是任何 Bottle 类的实例

```
def setup_routing(app):
    app.route('/new', method=['GET', 'POST'], form_new)
    app.route('/edit', method=['GET', 'POST'], form_edit)

app = Bottle()
setup_routing(app)
```

1.3 SimpleTemplate 模板引擎

Bottle 自带了一个快速，强大，易用的模板引擎，名为 *SimpleTemplate* 或简称为 *stpl* 。它是 `view()` 和 `template()` 两个函数默认调用的模板引擎。接下来会介绍该引擎的模板语法和一些常见用例。

基础 API：

`SimpleTemplate` 类实现了 `BaseTemplate` 接口

```
>>> from bottle import SimpleTemplate
>>> tpl = SimpleTemplate('Hello {{name}}!')
>>> tpl.render(name='World')
u'Hello World!'
```

简单起见，我们在例子中使用 `template()` 函数

```
>>> from bottle import template
>>> template('Hello {{name}}!', name='World')
u'Hello World!'
```

注意，编译模板和渲染模板是两件事情，尽管 `template()` 函数隐藏了这一事实。通常，模板只会被编译一次，然后会被缓存起来，但是会根据不同的参数，被多次渲染。

1.3.1 SimpleTemplate 的语法

虽然 Python 是一门强大的语言，但它对空白敏感的语法令其很难作为一个模板语言。SimpleTemplate 移除了一些限制，允许你写出干净的，有可读性的，可维护的模板，且保留了 Python 的强大功能。

> **Warning:** SimpleTemplate 模板会被编译为 Python 字节码，且在每次通过 `SimpleTemplate.render()` 渲染的时候执行。请不要渲染不可靠的模板！它们也许包含恶意代码。

内嵌语句

你已经在上面的"Hello World!" 例子中学习到了 `{{...}}` 语句的用法。只要在 `{{...}}` 中的 Python 语句返回一个字符串或有一个字符串的表达形式，它就是一个有效的语句。

```
>>> template('Hello {{name}}!', name='World')
u'Hello World!'
>>> template('Hello {{name.title() if name else "stranger"}}!', name=None)
u'Hello stranger!'
>>> template('Hello {{name.title() if name else "stranger"}}!', name='mArC')
u'Hello Marc!'
```

{{}} 中的 Python 语句会在渲染的时候被执行，可访问传递给 `SimpleTemplate.render()` 方法的所有参数。默认情况下，自动转义 HTML 标签以防止 XSS 攻击。可在语句前加上"!" 来关闭自动转义。

```
>>> template('Hello {{name}}!', name='<b>World</b>')
u'Hello &lt;b&gt;World&lt;/b&gt;!'
>>> template('Hello {{!name}}!', name='<b>World</b>')
u'Hello <b>World</b>!'
```

嵌入 Pyhton 代码

一行以 % 开头，表明这一行是 Python 代码。它和真正的 Python 代码唯一的区别，在于你需要显式地在末尾添加 %end 语句，表明一个代码块结束。这样你就不必担心 Python 代码中的缩进问题，*SimpleTemplate* 模板引擎的 parser 帮你处理了。不以 % 开头的行，被当作普通文本来渲染。

```
%if name:
  Hi <b>{{name}}</b>
%else:
  <i>Hello stranger</i>
%end
```

只有在行首的 % 字符才有意义，可以使用 %% 来转义。

```
这一行包含 % 但不是 Python 代码。
%% 以'%' 开头的一行
%%% 以'%%' 开头的一行
```

防止换行

你可以在一行代码前面加上 \\ 来防止换行。

```
<span>\\
%if True:
nobreak\\
%end
</span>
```

该模板的输出:

```
<span>nobreak</span>
```

%include 语句

你可以使用 %include sub_template [kwargs] 语句来包含其他模板。sub_template 参数是模板的文件名或路径。[kwargs] 部分是以逗号分开的键值对，是传给其他模板的参数。**kwargs 这样的语法来传递一个字典也是允许的。

```
%include header_template title='Hello World'
<p>Hello World</p>
%include footer_template
```

%rebase 语句

%rebase base_template [kwargs] 语句会渲染 base_template 这个模板，而不是原先的模板。然后 base_template 中使用一个空 %include 语句来包含原先的模板，并可访问所有通过 kwargs 传过来的参数。这样就可以使用模板来封装另一个模板，或者是模拟某些模板引擎中的继承机制。

让我们假设，你现在有一个与内容有关的模板，想在它上面加上一层普通的 HTML 层。为了避免 include 一堆模板，你可以使用一个基础模板。

名为 layout.tpl 的基础模板

```
<html>
<head>
  <title>{{title or 'No title'}}</title>
</head>
<body>
  %include
</body>
</html>
```

名为 content.tpl 的主模板

```
This is the page content: {{content}}
%rebase layout title='Content Title'
```

渲染 content.tpl

```
>>> print template('content', content='Hello World!')

<html>
<head>
  <title>Content Title</title>
</head>
<body>
  This is the page content: Hello World!
</body>
</html>
```

一个更复杂的使用场景 involves chained rebases and multiple content blocks. block_content.tpl 模板定义了两个函数，然后将它们传给 columns.tpl 这个基础模板。

```
%def leftblock():
  Left block content.
%end
%def rightblock():
  Right block content.
%end
%rebase columns leftblock=leftblock, rightblock=rightblock, title=title
```

columns.tpl 这个基础模板使用两个 callable(译者注：Python 中除了函数是 callable 的，类也可以是 callable 的，在这个例子里，是函数) 来渲染分别位于左边和右边的两列。然后将其自身封装在之前定义的 layout.tpl 模板里面。

```
%rebase layout title=title
<div style="width: 50%; float:left">
  %leftblock()
</div>
<div style="width: 50%; float:right">
  %rightblock()
</div>
```

让我们看一看 block_content.tpl 模板的输出

```
>>> print template('block_content', title='Hello World!')

<html>
<head>
  <title>Hello World</title>
</head>
<body>
<div style="width: 50%; float:left">
  Left block content.
</div>
<div style="width: 50%; float:right">
  Right block content.
</div>
</body>
</html>
```

模板内置函数 (Namespace Functions)

在模板中访问一个未定义的变量会导致 NameError 异常，并立即终止模板的渲染。这是 Python 的正常行为，并不奇怪。在抛出异常之前，你无法检查变量是否被定义。这在你想让输入更灵活，或想在不同情况下使用同一个模板的时候，就很烦人了。SimpleTemplate 模板引擎内置了三个函数来帮你解决这个问题，可以在模板的任何地方使用它们。

defined(*name*)
: 如果变量已定义则返回 True，反之返回 False。

get(*name*, *default=None*)
返回该变量，或一个默认值

setdefault(*name*, *default*)
如果该变量未定义，则定义它，赋一个默认值，返回该变量

下面是使用了这三个函数的例子，实现了模板中的可选参数。

```
% setdefault('text', 'No Text')
<h1>{{get('title', 'No Title')}}</h1>
<p> {{ text }} </p>
% if defined('author'):
  <p>By {{ author }}</p>
% end
```

1.3.2 SimpleTemplate API

class SimpleTemplate(*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, ***settings*)

classmethod split_comment(*code*)
Removes comments (#...) from python code.

render(**args*, ***kwargs*)
Render the template using keyword arguments as local variables.

1.3.3 已知 Bug

不兼容某些 Python 语法, 例子如下:

- 多行语句语句必须以 \ 结束, 如果出现了注释, 则不能再包含其他 # 字符.
- 不支持多行字符串 (译者注:"“" “"“这样的字符串)

1.4 API Reference(不译)

This is a mostly auto-generated API. If you are new to bottle, you might find the narrative 教程 more helpful.

1.4.1 Module Contents

The module defines several functions, constants, and an exception.

debug(*mode=True*)
Change the debug level. There is only one debug level supported at the moment.

run(*app=None*, *server='wsgiref'*, *host='127.0.0.1'*, *port=8080*, *interval=1*, *reloader=False*, *quiet=False*, *plugins=None*, *debug=False*, ***kargs*)
Start a server instance. This method blocks until the server terminates.

Parameters

- **app** – WSGI application or target string supported by `load_app()`. (default: `default_app()`)
- **server** – Server adapter to use. See `server_names` keys for valid names or pass a `ServerAdapter` subclass. (default: *wsgiref*)
- **host** – Server address to bind to. Pass `0.0.0.0` to listens on all interfaces including the external one. (default: 127.0.0.1)

- **port** – Server port to bind to. Values below 1024 require root privileges. (default: 8080)
- **reloader** – Start auto-reloading server? (default: False)
- **interval** – Auto-reloader interval in seconds (default: 1)
- **quiet** – Suppress output to stdout and stderr? (default: False)
- **options** – Options passed to the server adapter.

`load`(*target*, ***namespace*)
Import a module or fetch an object from a module.

- `package.module` returns *module* as a module object.
- `pack.mod:name` returns the module variable *name* from *pack.mod*.
- `pack.mod:func()` calls *pack.mod.func()* and returns the result.

The last form accepts not only function calls, but any type of expression. Keyword arguments passed to this function are available as local variables. Example: `import_string('re:compile(x)', x='[a-z]')`

`load_app`(*target*)
Load a bottle application from a module and make sure that the import does not affect the current default application, but returns a separate application object. See `load()` for the target parameter.

`request` = <LocalRequest: GET http://127.0.0.1/>
A thread-safe instance of `LocalRequest`. If accessed from within a request callback, this instance always refers to the *current* request (even on a multithreaded server).

`response` = Content-Type: text/html; charset=UTF-8
A thread-safe instance of `LocalResponse`. It is used to change the HTTP response for the *current* request.

`HTTP_CODES` = {300: 'Multiple Choices', 301: 'Moved Permanently', 302: 'Found', 303: 'See Other', 304:
A dict to map HTTP status codes (e.g. 404) to phrases (e.g. 'Not Found')

`app`()
`default_app`()
Return the current 默认应用. Actually, these are callable instances of `AppStack` and implement a stack-like API.

Routing

Bottle maintains a stack of `Bottle` instances (see `app()` and `AppStack`) and uses the top of the stack as a *default application* for some of the module-level functions and decorators.

`route`(*path*, *method='GET'*, *callback=None*, ***options*)
`get`(...)
`post`(...)
`put`(...)
`delete`(...)
Decorator to install a route to the current default application. See `Bottle.route()` for details.

`error`(...)
Decorator to install an error handler to the current default application. See `Bottle.error()` for details.

WSGI and HTTP Utilities

`parse_date`(*ims*)
Parse rfc1123, rfc850 and asctime timestamps and return UTC epoch.

`parse_auth`(*header*)
Parse rfc2617 HTTP authentication header string (basic) and return (user,pass) tuple or None

`cookie_encode`(*data*, *key*)
Encode and sign a pickle-able object. Return a (byte) string

`cookie_decode`(*data*, *key*)
Verify and decode an encoded string. Return an object or None.

`cookie_is_encoded`(*data*)
Return True if the argument looks like a encoded cookie.

`yieldroutes`(*func*)
Return a generator for routes that match the signature (name, args) of the func parameter. This may yield more than one route if the function takes optional keyword arguments. The output is best described by example:

```
a()         -> '/a'
b(x, y)     -> '/b/:x/:y'
c(x, y=5)   -> '/c/:x' and '/c/:x/:y'
d(x=5, y=6) -> '/d' and '/d/:x' and '/d/:x/:y'
```

`path_shift`(*script_name*, *path_info*, *shift=1*)
Shift path fragments from PATH_INFO to SCRIPT_NAME and vice versa.

Returns The modified paths.

Parameters

- **script_name** – The SCRIPT_NAME path.
- **script_name** – The PATH_INFO path.
- **shift** – The number of path fragments to shift. May be negative to change the shift direction. (default: 1)

Data Structures

class `MultiDict`(**a*, ***k*)
This dict stores multiple values per key, but behaves exactly like a normal dict in that it returns only the newest value for any given key. There are special methods available to access the full list of values.

`get`(*key*, *default=None*, *index=-1*, *type=None*)
Return the most recent value for a key.

Parameters

- **default** – The default value to be returned if the key is not present or the type conversion fails.
- **index** – An index for the list of available values.
- **type** – If defined, this callable is used to cast the value into a specific type. Exception are suppressed and result in the default value to be returned.

`append`(*key*, *value*)
Add a new value to the list of values for this key.

`replace`(*key*, *value*)
Replace the list of values with a single value.

`getall`(*key*)
Return a (possibly empty) list of values for a key.

`getone`(*key*, *default=None*, *index=-1*, *type=None*)
Aliases for WTForms to mimic other multi-dict APIs (Django)

`getlist`(*key*)
Return a (possibly empty) list of values for a key.

class `HeaderDict`(**a*, ***ka*)
A case-insensitive version of `MultiDict` that defaults to replace the old value instead of appending it.

class `FormsDict`(**a*, ***k*)
This `MultiDict` subclass is used to store request form data. Additionally to the normal dict-like item access methods (which return unmodified data as native strings), this container also supports attribute-like access to its values. Attributes are automatically de- or recoded to match `input_encoding` (default: 'utf8'). Missing attributes default to an empty string.

`input_encoding` = **'utf8'**
Encoding used for attribute values.

`recode_unicode` = **True**
If true (default), unicode strings are first encoded with *latin1* and then decoded to match `input_encoding`.

`decode`(*encoding=None*)
Returns a copy with all keys and values de- or recoded to match `input_encoding`. Some libraries (e.g. WTForms) want a unicode dictionary.

class `WSGIHeaderDict`(*environ*)
This dict-like class wraps a WSGI environ dict and provides convenient access to HTTP_* fields. Keys and values are native strings (2.x bytes or 3.x unicode) and keys are case-insensitive. If the WSGI environment contains non-native string values, these are de- or encoded using a lossless 'latin1' character set.

The API will remain stable even on changes to the relevant PEPs. Currently PEP 333, 444 and 3333 are supported. (PEP 444 is the only one that uses non-native strings.)

`cgikeys` = **('CONTENT_TYPE', 'CONTENT_LENGTH')**
List of keys that do not have a '**HTTP_**' prefix.

`raw`(*key*, *default=None*)
Return the header value as is (may be bytes or unicode).

class `AppStack`
A stack-like list. Calling it returns the head of the stack.

`pop`()
Return the current default application and remove it from the stack.

`push`(*value=None*)
Add a new `Bottle` instance to the stack

class `ResourceManager`(*base='./'*, *opener=<built-in function open>*, *cachemode='all'*)
This class manages a list of search paths and helps to find and open aplication-bound resources (files).

Parameters

- **base** – default value for `add_path()` calls.
- **opener** – callable used to open resources.
- **cachemode** – controls which lookups are cached. One of 'all', 'found' or 'none'.

`path` = **None**
A list of search paths. See `add_path()` for details.

`cache` = **None**
A cache for resolved paths. `res.cache.clear()` clears the cache.

`add_path`(*path*, *base=None*, *index=None*, *create=False*)
Add a new path to the list of search paths. Return False if the path does not exist.

Parameters

- **path** – The new search path. Relative paths are turned into an absolute and normalized form. If the path looks like a file (not ending in /), the filename is stripped off.
- **base** – Path used to absolutize relative search paths. Defaults to `base` which defaults to `os.getcwd()`.
- **index** – Position within the list of search paths. Defaults to last index (appends to the list).
- **create** – Create non-existent search paths. Off by default.

The *base* parameter makes it easy to reference files installed along with a python module or package:

```
res.add_path('./resources/', __file__)
```

`lookup`(*name*)
Search for a resource and return an absolute file path, or *None*.

The `path` list is searched in order. The first match is returend. Symlinks are followed. The result is cached to speed up future lookups.

`open`(*name*, *mode='r'*, **args*, ***kwargs*)
Find a resource and return a file object, or raise IOError.

Exceptions

exception `BottleException`
A base class for exceptions used by bottle.

exception `HTTPResponse`(*output=''*, *status=200*, *header=None*)
Used to break execution and immediately finish the response

exception `HTTPError`(*code=500*, *output='Unknown Error'*, *exception=None*, *traceback=None*, *header=None*)
Used to generate an error page

exception `RouteReset`
If raised by a plugin or request handler, the route is reset and all plugins are re-applied.

1.4.2 The `Bottle` Class

class `Bottle`(*catchall=True*, *autojson=True*)
Each Bottle object represents a single, distinct web application and consists of routes, callbacks, plugins, resources and configuration. Instances are callable WSGI applications.

Parameters **catchall** – If true (default), handle all exceptions. Turn off to let debugging middleware handle exceptions.

`catchall` = **None**
If true, most exceptions are caught and returned as `HTTPError`

`resources` = **None**
A **:cls:‘ResourceManager‘** for application files

`config` = **None**
A **:cls:‘ConfigDict‘** for app specific configuration.

`mount`(*prefix*, *app*, ***options*)
Mount an application (`Bottle` or plain WSGI) to a specific URL prefix. Example:

```
root_app.mount('/admin/', admin_app)
```

Parameters

- **prefix** – path prefix or *mount-point*. If it ends in a slash, that slash is mandatory.
- **app** – an instance of `Bottle` or a WSGI application.

All other parameters are passed to the underlying `route()` call.

`merge`(*routes*)
Merge the routes of another `Bottle` application or a list of `Route` objects into this application. The routes keep their 'owner', meaning that the `Route.app` attribute is not changed.

`install`(*plugin*)
Add a plugin to the list of plugins and prepare it for being applied to all routes of this application. A plugin may be a simple decorator or an object that implements the `Plugin` API.

`uninstall`(*plugin*)
Uninstall plugins. Pass an instance to remove a specific plugin, a type object to remove all plugins that match that type, a string to remove all plugins with a matching `name` attribute or `True` to remove all plugins. Return the list of removed plugins.

`run`(***kwargs*)
Calls `run()` with the same parameters.

`reset`(*route=None*)
Reset all routes (force plugins to be re-applied) and clear all caches. If an ID or route object is given, only that specific route is affected.

`close`()
Close the application and all installed plugins.

`match`(*environ*)
Search for a matching route and return a (`Route` , urlargs) tuple. The second value is a dictionary with parameters extracted from the URL. Raise `HTTPError` (404/405) on a non-match.

`get_url`(*routename*, ***kargs*)
Return a string that matches a named route

`add_route`(*route*)
Add a route object, but do not change the `Route.app` attribute.

`route`(*path=None*, *method='GET'*, *callback=None*, *name=None*, *apply=None*, *skip=None*, ***config*)
A decorator to bind a function to a request URL. Example:

```
@app.route('/hello/:name')
def hello(name):
    return 'Hello %s' % name
```

The `:name` part is a wildcard. See `Router` for syntax details.

Parameters

- **path** – Request path or a list of paths to listen to. If no path is specified, it is automatically generated from the signature of the function.
- **method** – HTTP method (*GET*, *POST*, *PUT*, ...) or a list of methods to listen to. (default: *GET*)
- **callback** – An optional shortcut to avoid the decorator syntax. `route(..., callback=func)` equals `route(...)(func)`

- **name** – The name for this route. (default: None)
- **apply** – A decorator or plugin or a list of plugins. These are applied to the route callback in addition to installed plugins.
- **skip** – A list of plugins, plugin classes or names. Matching plugins are not installed to this route. `True` skips all.

Any additional keyword arguments are stored as route-specific configuration and passed to plugins (see `Plugin.apply()`).

`get`(*path=None*, *method='GET'*, ***options*)
Equals `route()`.

`post`(*path=None*, *method='POST'*, ***options*)
Equals `route()` with a `POST` method parameter.

`put`(*path=None*, *method='PUT'*, ***options*)
Equals `route()` with a `PUT` method parameter.

`delete`(*path=None*, *method='DELETE'*, ***options*)
Equals `route()` with a `DELETE` method parameter.

`error`(*code=500*)
Decorator: Register an output handler for a HTTP error code

`hook`(*name*)
Return a decorator that attaches a callback to a hook. Three hooks are currently implemented:

- before_request: Executed once before each request
- after_request: Executed once after each request
- app_reset: Called whenever `reset()` is called.

`handle`(*path*, *method='GET'*)
(deprecated) Execute the first matching route callback and return the result. `HTTPResponse` exceptions are caught and returned. If `Bottle.catchall` is true, other exceptions are caught as well and returned as `HTTPError` instances (500).

`wsgi`(*environ*, *start_response*)
The bottle WSGI-interface.

class `Route`(*app*, *rule*, *method*, *callback*, *name=None*, *plugins=None*, *skiplist=None*, ***config*)
This class wraps a route callback along with route specific metadata and configuration and applies Plugins on demand. It is also responsible for turing an URL path rule into a regular expression usable by the Router.

`app` = **None**
The application this route is installed to.

`rule` = **None**
The path-rule string (e.g. `/wiki/:page`).

`method` = **None**
The HTTP method as a string (e.g. `GET`).

`callback` = **None**
The original callback with no plugins applied. Useful for introspection.

`name` = **None**
The name of the route (if specified) or `None`.

`plugins` = **None**
A list of route-specific plugins (see `Bottle.route()`).

`skiplist` = **None**
A list of plugins to not apply to this route (see `Bottle.route()`).

`config` = **None**
Additional keyword arguments passed to the `Bottle.route()` decorator are stored in this dictionary. Used for route-specific plugin configuration and meta-data.

`reset()`
Forget any cached values. The next time `call` is accessed, all plugins are re-applied.

`prepare()`
Do all on-demand work immediately (useful for debugging).

`all_plugins()`
Yield all Plugins affecting this route.

1.4.3 The `Request` Object

The `Request` class wraps a WSGI environment and provides helpful methods to parse and access form data, cookies, file uploads and other metadata. Most of the attributes are read-only.

You usually don't instantiate `Request` yourself, but use the module-level `bottle.request` instance. This instance is thread-local and refers to the *current* request, or in other words, the request that is currently processed by the request handler in the current context. *Thread locality* means that you can safely use a global instance in a multithreaded environment.

`Request`
alias of `LocalRequest`

class `LocalRequest`(*environ=None*)
A thread-local subclass of `BaseRequest` with a different set of attribues for each thread. There is usually only one global instance of this class (`request`). If accessed during a request/response cycle, this instance always refers to the *current* request (even on a multithreaded server).

`bind`(*environ=None*)
Wrap a WSGI environ dictionary.

`environ`
Thread-local property stored in `_lctx.request_environ`

class `BaseRequest`(*environ=None*)
A wrapper for WSGI environment dictionaries that adds a lot of convenient access methods and properties. Most of them are read-only.

Adding new attributes to a request actually adds them to the environ dictionary (as 'bottle.request.ext.<name>'). This is the recommended way to store and access request-specific data.

`MEMFILE_MAX` = **102400**
Maximum size of memory buffer for `body` in bytes.

`MAX_PARAMS` = **100**
Maximum number pr GET or POST parameters per request

`environ`
The wrapped WSGI environ dictionary. This is the only real attribute. All other attributes actually are read-only properties.

`app`
Bottle application handling this request.

`path`
The value of `PATH_INFO` with exactly one prefixed slash (to fix broken clients and avoid the "empty path" edge case).

`method`
The `REQUEST_METHOD` value as an uppercase string.

`headers`
A `WSGIHeaderDict` that provides case-insensitive access to HTTP request headers.

`get_header`(*name*, *default=None*)
Return the value of a request header, or a given default value.

`cookies`
Cookies parsed into a `FormsDict`. Signed cookies are NOT decoded. Use `get_cookie()` if you expect signed cookies.

`get_cookie`(*key*, *default=None*, *secret=None*)
Return the content of a cookie. To read a *Signed Cookie*, the *secret* must match the one used to create the cookie (see `BaseResponse.set_cookie()`). If anything goes wrong (missing cookie or wrong signature), return a default value.

`query`
The `query_string` parsed into a `FormsDict`. These values are sometimes called "URL arguments" or "GET parameters", but not to be confused with "URL wildcards" as they are provided by the `Router`.

`forms`
Form values parsed from an *url-encoded* or *multipart/form-data* encoded POST or PUT request body. The result is retuned as a `FormsDict`. All keys and values are strings. File uploads are stored separately in `files`.

`params`
A `FormsDict` with the combined values of `query` and `forms`. File uploads are stored in `files`.

`files`
File uploads parsed from an *url-encoded* or *multipart/form-data* encoded POST or PUT request body. The values are instances of `cgi.FieldStorage`. The most important attributes are:

filename The filename, if specified; otherwise None; this is the client side filename, *not* the file name on which it is stored (that's a temporary file you don't deal with)

file The file(-like) object from which you can read the data.

value The value as a *string*; for file uploads, this transparently reads the file every time you request the value. Do not do this on big files.

`json`
If the `Content-Type` header is `application/json`, this property holds the parsed content of the request body. Only requests smaller than `MEMFILE_MAX` are processed to avoid memory exhaustion.

`body`
The HTTP request body as a seek-able file-like object. Depending on `MEMFILE_MAX`, this is either a temporary file or a `io.BytesIO` instance. Accessing this property for the first time reads and replaces the `wsgi.input` environ variable. Subsequent accesses just do a *seek(0)* on the file object.

`GET`
An alias for `query`.

`POST`
The values of `forms` and `files` combined into a single `FormsDict`. Values are either strings (form values) or instances of `cgi.FieldStorage` (file uploads).

`COOKIES`
Alias for `cookies` (deprecated).

`url`
The full request URI including hostname and scheme. If your app lives behind a reverse proxy or load balancer and you get confusing results, make sure that the `X-Forwarded-Host` header is set correctly.

urlparts
The `url` string as an `urlparse.SplitResult` tuple. The tuple contains (scheme, host, path, query_string and fragment), but the fragment is always empty because it is not visible to the server.

fullpath
Request path including `script_name` (if present).

query_string
The raw `query` part of the URL (everything in between `?` and `#`) as a string.

script_name
The initial portion of the URL's *path* that was removed by a higher level (server or routing middleware) before the application was called. This script path is returned with leading and tailing slashes.

path_shift(*shift=1*)
Shift path segments from `path` to `script_name` and vice versa.

Parameters **shift** – The number of path segments to shift. May be negative to change the shift direction. (default: 1)

content_length
The request body length as an integer. The client is responsible to set this header. Otherwise, the real length of the body is unknown and -1 is returned. In this case, `body` will be empty.

content_type
The Content-Type header as a lowercase-string (default: empty).

is_xhr
True if the request was triggered by a XMLHttpRequest. This only works with JavaScript libraries that support the *X-Requested-With* header (most of the popular libraries do).

is_ajax
Alias for `is_xhr`. "Ajax" is not the right term.

auth
HTTP authentication data as a (user, password) tuple. This implementation currently supports basic (not digest) authentication only. If the authentication happened at a higher level (e.g. in the front web-server or a middleware), the password field is None, but the user field is looked up from the `REMOTE_USER` environ variable. On any errors, None is returned.

remote_route
A list of all IPs that were involved in this request, starting with the client IP and followed by zero or more proxies. This does only work if all proxies support the `X-Forwarded-For` header. Note that this information can be forged by malicious clients.

remote_addr
The client IP as a string. Note that this information can be forged by malicious clients.

copy()
Return a new `Request` with a shallow `environ` copy.

1.4.4 The `Response` Object

The `Response` class stores the HTTP status code as well as headers and cookies that are to be sent to the client. Similar to `bottle.request` there is a thread-local `bottle.response` instance that can be used to adjust the *current* response. Moreover, you can instantiate `Response` and return it from your request handler. In this case, the custom instance overrules the headers and cookies defined in the global one.

Response
alias of `LocalResponse`

class `LocalResponse`(*body=''*, *status=None*, ***headers*)
A thread-local subclass of `BaseResponse` with a different set of attribues for each thread. There is usually only one global instance of this class (`response`). Its attributes are used to build the HTTP response at the end of the request/response cycle.

`body`
Thread-local property stored in `_lctx.response_body`

class `BaseResponse`(*body=''*, *status=None*, ***headers*)
Storage class for a response body as well as headers and cookies.

This class does support dict-like case-insensitive item-access to headers, but is NOT a dict. Most notably, iterating over a response yields parts of the body and not the headers.

`copy()`
Returns a copy of self.

`status_line`
The HTTP status line as a string (e.g. `404 Not Found`).

`status_code`
The HTTP status code as an integer (e.g. 404).

`status`
A writeable property to change the HTTP response status. It accepts either a numeric code (100-999) or a string with a custom reason phrase (e.g. "404 Brain not found"). Both `status_line` and `status_code` are updated accordingly. The return value is always a status string.

`headers`
An instance of `HeaderDict`, a case-insensitive dict-like view on the response headers.

`get_header`(*name*, *default=None*)
Return the value of a previously defined header. If there is no header with that name, return a default value.

`set_header`(*name*, *value*, *append=False*)
Create a new response header, replacing any previously defined headers with the same name.

`add_header`(*name*, *value*)
Add an additional response header, not removing duplicates.

`iter_headers()`
Yield (header, value) tuples, skipping headers that are not allowed with the current response status code.

`headerlist`
WSGI conform list of (header, value) tuples.

`content_type`
Current value of the 'Content-Type' header.

`content_length`
Current value of the 'Content-Length' header.

`charset`
Return the charset specified in the content-type header (default: utf8).

`COOKIES`
A dict-like SimpleCookie instance. This should not be used directly. See `set_cookie()`.

`set_cookie`(*name*, *value*, *secret=None*, ***options*)
Create a new cookie or replace an old one. If the *secret* parameter is set, create a *Signed Cookie* (described below).

Parameters

- **name** – the name of the cookie.

- **value** – the value of the cookie.
- **secret** – a signature key required for signed cookies.

Additionally, this method accepts all RFC 2109 attributes that are supported by `cookie.Morsel`, including:

Parameters

- **max_age** – maximum age in seconds. (default: None)
- **expires** – a datetime object or UNIX timestamp. (default: None)
- **domain** – the domain that is allowed to read the cookie. (default: current domain)
- **path** – limits the cookie to a given path (default: current path)
- **secure** – limit the cookie to HTTPS connections (default: off).
- **httponly** – prevents client-side javascript to read this cookie (default: off, requires Python 2.6 or newer).

If neither *expires* nor *max_age* is set (default), the cookie will expire at the end of the browser session (as soon as the browser window is closed).

Signed cookies may store any pickle-able object and are cryptographically signed to prevent manipulation. Keep in mind that cookies are limited to 4kb in most browsers.

Warning: Signed cookies are not encrypted (the client can still see the content) and not copy-protected (the client can restore an old cookie). The main intention is to make pickling and unpickling save, not to store secret information at client side.

`delete_cookie`(*key*, ***kwargs*)
Delete a cookie. Be sure to use the same *domain* and *path* settings as used to create the cookie.

1.4.5 Templates

All template engines supported by `bottle` implement the `BaseTemplate` API. This way it is possible to switch and mix template engines without changing the application code at all.

class `BaseTemplate`(*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, ***settings*)
Base class and minimal API for template adapters

`__init__`(*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, ***settings*)
Create a new template. If the source parameter (str or buffer) is missing, the name argument is used to guess a template filename. Subclasses can assume that self.source and/or self.filename are set. Both are strings. The lookup, encoding and settings parameters are stored as instance variables. The lookup parameter stores a list containing directory paths. The encoding parameter should be used to decode byte strings or files. The settings parameter contains a dict for engine-specific settings.

classmethod `search`(*name*, *lookup=[]*)
Search name in all directories specified in lookup. First without, then with common extensions. Return first hit.

classmethod `global_config`(*key*, **args*)
This reads or sets the global settings stored in class.settings.

`prepare`(***options*)
Run preparations (parsing, caching, ...). It should be possible to call this again to refresh a template or to update settings.

`render`(**args*, ***kwargs*)
Render the template with the specified local variables and return a single byte or unicode

string. If it is a byte string, the encoding must match self.encoding. This method must be thread-safe! Local variables may be provided in dictionaries (*args) or directly, as keywords (**kwargs).

`view`(*tpl_name*, ***defaults*)
Decorator: renders a template for a handler. The handler can control its behavior like that:

- return a dict of template vars to fill out the template
- return something other than a dict and the view decorator will not process the template, but return the handler result as is. This includes returning a HTTPResponse(dict) to get, for instance, JSON with autojson or other castfilters.

`template`(**args*, ***kwargs*)
Get a rendered template as a string iterator. You can use a name, a filename or a template string as first parameter. Template rendering arguments can be passed as dictionaries or directly (as keyword arguments).

You can write your own adapter for your favourite template engine or use one of the predefined adapters. Currently there are four fully supported template engines:

Class	URL	Decorator	Render function
`SimpleTemplate`	*SimpleTemplate* 模板引擎	`view()`	`template()`
`MakoTemplate`	http://www.makotemplates.org	`mako_view()`	`mako_template()`
`CheetahTemplate`	http://www.cheetahtemplate.org/	`cheetah_view()`	`cheetah_template()`
`Jinja2Template`	http://jinja.pocoo.org/	`jinja2_view()`	`jinja2_template()`

To use `MakoTemplate` as your default template engine, just import its specialised decorator and render function:

```
from bottle import mako_view as view, mako_template as template
```

1.5 可用插件列表

这是一份第三方插件的列表，扩展 Bottle 的核心功能，或集成其它类库。

在插件 查看常见的插件问题 (安装，使用)。如果你计划开发一个新的插件，插件开发指南 也许对你有帮助。

Bottle-Extras Meta package to install the bottle plugin collection.

Bottle-Flash flash plugin for bottle

Bottle-Hotqueue FIFO Queue for Bottle built upon redis

Macaron Macaron is an object-relational mapper (ORM) for SQLite.

Bottle-Memcache Memcache integration for Bottle.

Bottle-MongoDB MongoDB integration for Bottle

Bottle-Redis Redis integration for Bottle.

Bottle-Renderer Renderer plugin for bottle

Bottle-Servefiles A reusable app that serves static files for bottle apps

Bottle-Sqlalchemy SQLAlchemy integration for Bottle.

Bottle-Sqlite SQLite3 database integration for Bottle.

Bottle-Web2pydal Web2py Dal integration for Bottle.

Bottle-Werkzeug Integrates the *werkzeug* library (alternative request and response objects, advanced debugging middleware and more).

这里列出的插件不属于 Bottle 或 Bottle 项目，是第三方开发并维护的。

1.5.1 Bottle-SQLite

SQLite 是一个" 自包含" 的 SQL 数据库引擎，不需要任何其它服务器软件或安装过程。sqlite3 模块是 Python 标准库的一部分，在大部分系统上面已经安装了。在开发依赖数据库的应用原型的时候，它是非常有用的，可在部署的时候再使用 PostgreSQL 或 MySQL。

这个插件让在 Bottle 应用中使用 sqlite 数据库更简单了。一旦安装了，你只要在 route 的回调函数里添加一个"db" 参数 (可更改为其它字符)，就能使用数据库链接了。

安装

下面两个命令任选一个

```
$ pip install bottle-sqlite
$ easy_install bottle-sqlite
```

或从 github 下载最新版本

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/sqlite
$ python setup.py install
```

使用

一旦安装到应用里面，如果 route 的回调函数包含一个名为"db" 的参数，该插件会给该参数传一个 sqlite3.Connection 类的实例。

```
import bottle

app = bottle.Bottle()
plugin = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
app.install(plugin)

@app.route('/show/:item')
def show(item, **db** ):
    row = db.execute('SELECT * from items where name=?', item).fetchone()
    if row:
        return template('showitem', page=row)
    return HTTPError(404, "Page not found")
```

不包含"db" 参数的 route 不会受影响。

可通过下标 (像元组) 来访问 sqlite3.Row 对象，且对命名的大小写敏感。在请求结束后，自动提交事务和关闭连接。如果出现任何错误，自上次提交后的所有更改都会被回滚，以保证数据库的一致性。

配置

有下列可配置的选项:

- **dbfile**: 数据库文件 (默认：存在在内存中)。
- **keyword**: route 中使用数据库连接的参数 (默认：'db')。
- **autocommit**: 是否在请求结束后提交事务 (默认：True)。
- **dictrows**: 是否可像字典那样访问 row 对象 (默认：True)。

你可在 route 里面覆盖这些默认值。

```
@app.route('/cache/:item', sqlite={'dbfile': ':memory:'})
def cache(item, db):
    ...
```

或在同一个应用里面安装 keyword 参数不同的两个插件。

```
app = bottle.Bottle()
test_db = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
cache_db = bottle.ext.sqlite.Plugin(dbfile=':memory:', keyword='cache')
app.install(test_db)
app.install(cache_db)

@app.route('/show/:item')
def show(item, db):
    ...

@app.route('/cache/:item')
def cache(item, cache):
    ...
```

1.5.2 Bottle-Werkzeug

Werkzeug 是一个强大的 WSGI 类库。它包含一个交互式的调试器和包装好的 request 和 response 对象。

这个插件集成了 werkzeug.wrappers.Request 和 werkzeug.wrappers.Response 用于取代内置的相应实现，支持 werkzeug.exceptions , 用交互式的调试器替换了默认的错误页面。

安装

下面两个命令任选一个

```
$ pip install bottle-werkzeug
$ easy_install bottle-werkzeug
```

或从 github 上下载最新版本

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/werkzeug
$ python setup.py install
```

使用

一旦安装到应用中，该插件将支持 werkzeug.wrappers.Response ，所有类型的 werkzeug.exceptions 和 thread-local 的 werkzeug.wrappers.Request 的实例 (每次请求都更新)。插件它本身还扮演着 werkzeug 模块的角色，所以你无需在应用中导入 werkzeug。下面是一个例子。

```
import bottle

app = bottle.Bottle()
werkzeug = bottle.ext.werkzeug.Plugin()
app.install(werkzeug)

req = werkzueg.request # For the lazy.

@app.route('/hello/:name')
def say_hello(name):
    greet = {'en':'Hello', 'de':'Hallo', 'fr':'Bonjour'}
    language = req.accept_languages.best_match(greet.keys())
    if language:
        return werkzeug.Response('%s %s!' % (greet[language], name))
    else:
        raise werkzeug.exceptions.NotAcceptable()
```

使用调试器

该插件用一个更高级的调试器替换了默认的错误页面。如果你启用了 *evalex* 特性，你可在浏览器中使用一个交互式的终端来检查错误。在你启用该特性之前，请看通过 werkzeug 来调试。

配置

有下列可配置的选项:

- **evalex**: 启用交互式的调试器。要求一个单进程的服务器，且有安全隐患。请看通过 werkzeug 来调试 (默认：False)
- **request_class**: 默认是 `werkzeug.wrappers.Request` 的实例
- **debugger_class**: 默认是 `werkzeug.debug.DebuggedApplication` 的子类，遵守 `bottle.DEBUG` 中的设置

CHAPTER 2

知识库

收集文章，使用指南和 HOWTO

2.1 Tutorial: Todo-List 应用

Note: 这份教程是 noisefloor 编写的，并在不断完善中。

这份教程简单介绍了 Bottle 框架，目的是让你看完后能在项目中使用 Bottle。它没有涵盖所有东西，但介绍了 URL 映射，模板，处理 GET/POST 请求等基础知识。

读懂这份教程，你不需要事先了解 WSGI 标准，Bottle 也一直避免用户直接接触 WSGI 标准。但你需要了解 Python 这门语言。更进一步，这份教程中的例子需要从 SQL 数据库中读写数据，所以事先了解一点 SQL 知识是很有帮助的。例子中使用了 SQLite 来保存数据。因为是网页应用，所以事先了解一点 HTML 的知识也很有帮助。

作为一份教程，我们的代码尽可能做到了简明扼要。尽管教程中的代码能够工作，但是我们还是不建议你在公共服务器中使用教程中的代码。如果你想要这样做，你应该添加足够的错误处理，并且加密你的数据库，处理用户的输入。

目录

2.1.1 目标

在这份教程结束的时候，我们将完成一个简单的，基于 Web 的 ToDo list(待办事项列表)。列表中的每一个待办事项都包含一条文本 (最长 100 个字符) 和一个状态 (0 表示关闭，1 表示开启)。通过网页，已开启的待办事项可以被查看和编辑，可添加待办事项到列表中。

在开发过程中，所有的页面都只可以通过 `localhost` 来访问，完了会介绍如何将应用部署到” 真实” 服务器的服务器上面，包括使用 mod_wsgi 来部署到 Apache 服务器上面。

Bottle 会负责 URL 映射，通过模板来输出页面。待办事项列表被存储在一个 SQLite 数据库中，通过 Python 代码来读写数据库。

我们会完成以下页面和功能:

- 首页 `http://localhost:8080/todo`
- 添加待办事项: `http://localhost:8080/new`
- 编辑待办事项: `http://localhost:8080/edit/:no`
- 通过 @validate 修饰器来验证数据合法性
- 捕获错误

2.1.2 开始之前...

安装

假设你已经安装好了 Python (2.5 或更改版本)，接下来你只需要下载 Bottle 就行了。除了 Python 标准库，Bottle 没有其他依赖。

你可通过 Python 的 esay_install 命令来安装 Bottle: `easy_install bottle`

其它软件

因为我们使用 SQLite3 来做数据库，请确保它已安装。如果是 Linux 系统，大多数的发行版已经默认安装了 SQLite3。SQLite 同时可工作在 Windows 系统和 MacOS X 系统上面。Pyhton 标准库中，已经包含了 *sqlite3* 模块。

创建一个 SQL 数据库

首先，我们需要先创建一个数据库，稍后会用到。在你的项目文件夹执行以下脚本即可，你也可以在 Python 解释器逐条执行。

```
import sqlite3
con = sqlite3.connect('todo.db') # Warning: This file is created in the current directory
con.execute("CREATE TABLE todo (id INTEGER PRIMARY KEY, task char(100) NOT NULL, status bool NOT NULL)")
con.execute("INSERT INTO todo (task,status) VALUES ('Read A-byte-of-python',0)")
con.execute("INSERT INTO todo (task,status) VALUES ('Visit the Python website',1)")
con.execute("INSERT INTO todo (task,status) VALUES ('Test various editors for syntax highlighting',1)")
con.execute("INSERT INTO todo (task,status) VALUES ('Choose your favorite WSGI-Framework',0)")
con.commit()
```

现在，我们已经创建了一个名字为 *todo.db* 的数据库文件，数据库中有一张名为 `todo` 的表，表中有 `id` , `task` , 及 `status` 这三列。每一行的 `id` 都是唯一的，稍后会根据 id 来获取数据。`task` 用于保存待办事项的文本，最大长度为 100 个字符。最后 `status` 用于标明待办事项的状态，0 为开启，1 为关闭。

2.1.3 基于 Bottle 的待办事项列表

为了创建我们的 Web 应用，我们先来介绍一下 Bottle 框架。首先，我们需要了解 Bottle 中的 route，即 URL 映射。

route URL 映射

基本上，浏览器访问的每一页面都是动态生成的。Bottle 通过 route，将浏览器访问的 URL 映射到具体的 Python 函数。例如，在我们访问 `http://localhost:8080/todo` 的时候，Bottle 会查找 `todo` 这个 route 映射到了哪个函数上面，接着调用该函数来响应浏览器请求。

第一步 - 显示所有已开启的待办事项

在我们了解什么是 route 后，让我们来试着写一个。访问它即可查看所有已开启的待办事项。

```
import sqlite3
from bottle import route, run

@route('/todo')
def todo_list():
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT id, task FROM todo WHERE status LIKE '1'")
    result = c.fetchall()
    return str(result)

run()
```

将上面的代码保存为 `todo.py` ，放到 `todo.db` 文件所在的目录。如果你想将它们分开放，则需要在 `sqlite3.connect()` 函数中写上 `todo.db` 文件的路径。

来看看我们写的代码。1. 导入了必须的 `sqlite3` 模块，从 Bottle 中导入 `route` 和 `run` 。2. `run()` 函数启动了 Bottle 的内置开发服务器，默认情况下，开发服务器在监听本地的 8080 端口。3. `route` 是 Bottle 实现 URL 映射功能的修饰器。你可以看到，我们定义了一个 `todo_list()` 函数，读取了数据库中的数据。然后我们使用 `@route('/todo')` 来将 `todo_list()` 函数和 “todo“ 这个 route 绑定在一起。每一次浏览器访问 `http://localhost:8080/todo` 的时候，Bottle 都会调用 `todo_list()` 函数来响应请求，并返回页面，这就是 route 的工作方式了。

事实上，你可以给一个函数添加多个 route。

```
@route('/todo')
@route('/my_todo_list')
def todo_list():
    ...
```

这样是正确的。但是反过来，你不能将一个 route 和多个函数绑定在一起。

你在浏览器中看到的即是你在 `todo_list()` 函数中返回的页面。在这个例子中，我们通过 `str()` 函数将结果转换成字符串，因为 Bottle 期望函数的返回值是一个字符串或一个字符串的列表。但 Python DB API 中规定了，数据库查询的返回值是一个元组的列表。

现在，我们已经了解上面的代码是如何工作的，是时候运行它来看看效果了。记得在 Linux 或 Unix 系统中，`todo.py` 文件需要标记为可执行 (译者注：没有必要)。然后，通过 `python todo.py` 命令来执行该脚本，接着用浏览器访问 `http://localhost:8080/todo` 来看看效果。如果代码没有写错，你应该会在页面看到以下输出。

```
[(2, u'Visit the Python website'), (3, u'Test various editors for and check the syntax highlighting')]
```

如果是这样，那么恭喜你！如果出现错误，那么你需要检查代码时候写错，修改完后记得重启 HTTP 服务器，要不新的版本不会生效。

实际上，这个输出很难看，只是 SQL 查询的结果。所以，下一步我们会把它变得更好看。

调试和自动加载

或许你已经注意到了，如果代码出错的话，Bottle 会在页面上显示一个简短的错误信息。例如，连接数据库失败。为了方便调试，我们希望错误信息更加具体，可加上以下语句。

```
from bottle import run, route, debug
...
#add this at the very end:
debug(True)
run()
```

开启调试模式后，出错时页面会打印出完整的 Python 运行栈。另外，在调试模式下，模板也不会被缓存，任何对模板的修改会马上生效，而不用重启服务器。

> **Warning:** `debug(True)` 是为开发时的调试服务的，不应 在生产环境中开启调试模式。

另外一个十分有用的功能是自动加载，可修改 `run()` 语句来开启。

```
run(reloader=True)
```

这样会自动检测对脚本的修改，并自动重启服务器来使其生效。同上，这个功能并不建议在生产环境中使用。

使用模板来格式化输出

现在我们试着格式化脚本的输出，使其更适合查看。

实际上 Bottle 期望 route 的回调函数返回一个字符串或一个字符串列表，通过内置的 HTTP 服务器将其返回给浏览器。Bottle 不关心字符串的内容，所以我们可以将其格式化成 HTML 格式。

Bottle 内置了独创的模板引擎。模板是后缀名为 `.tpl` 的文本文件。模板的内容混合着 HTML 标签和 Python 语句，模板也可以接受参数。例如数据库的查询结果，我们可以在模板内将其漂亮地格式化。

接下来，我们要将数据库的查询结果格式化为一个两列的表格。表格的第一列为待办事项的 ID，第二列为待办事项的内容。查询结果是一个元组的列表，列表中的每个元组后包含一个结果。

在例子中使用模板，只需要添加以下代码。

```
from bottle import route, run, debug, template
...
result = c.fetchall()
c.close()
output = template('make_table', rows=result)
return output
...
```

我们添加了两样东西。首先我们从 Bottle 中导入了 `template` 函数以使用模板功能，接着，我们渲染 `make_table` 这个模板 (参数是 rows=result)，把模板函数的返回值赋予 `output` 变量，并返回 `output` 。如有必要，我们可添加更多的参数。

模板总是返回一个字符串的列表，所以我们无须转换任何东西。当然，我们可以将返回写为一行以减少代码量。

```
return template('make_table', rows=result)
```

对应的模板文件。

```
%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or ...)
<p> 所有已开启的待办事项:</p>
<table border="1">
%for row in rows:
  <tr>
  %for col in row:
    <td>{{col}}</td>
  %end
  </tr>
%end
</table>
```

将上面的代码保存为 `make_table.tpl` 文件，和 `todo.py` 放在同一个目录。

看看上面的代码，以% 开头的行被当作 Python 代码来执行。请注意，只有正确的 Python 语句才能通过编译，要不模板就会抛出一个异常。除了 Python 语句，其它都是普通的 HTML 标记。

如你所见，为了遍历 rows ，我们两次使用了 Python 的 for 语句。rows``是持有查询结果的变量，一个元组的列表。第一个 ``for 语句遍历了列表中所有的元组，第二个 for 语句遍历了元组中的元素，将其放进表格中。for , if , while 语句都需要通过 %end 来关闭，要不会得到不正确的结果。

如果想要在不以% 开头的行中访问变量，则需要把它放在两个大括号中间。这告诉模板，需要用变量的实际值将其替换掉。

再次运行这个脚本，页面输出依旧不是很好看，但是更具可读性了。当然，你可给模板中的 HTML 标签加上 CSS 样式，使其更好看。

使用 GET 和 POST

能够查看所有代码事项后，让我们进入到下一步，添加新的待办事项到列表中。新的待办事项应该在一个常规的 HTML 表单中，通过 GET 方式提交。

让我们先来添加一个接受 GET 请求的 route。

```
from bottle import route, run, debug, template, request
...
return template('make_table', rows=result)
...

@route('/new', method='GET')
def new_item():

    new = request.GET.get('task', '').strip()

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()

    c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
    new_id = c.lastrowid

    conn.commit()
    c.close()

    return '<p> 新的待办事项已添加到数据库中,ID 为 %s</p>' % new_id
```

为了访问 GET(或 POST) 中的数据，我们需要从 Bottle 中导入 request ，通过 request.GET.get('task', '').strip() 来获取表单中 task 字段的数据。可多次使用 request.GET.get() 来获取表单中所有字段的数据。

接下来是对数据的操作：写入数据库，获取返回的 ID，生成页面。

因为我们是从 HTML 表单中获取数据，所以现在让我们来创建这个表单吧。我们通过 /new 这个 URL 来添加待办事项。

```
...
@route('/new', method='GET')
def new_item():

if request.GET.get('save','').strip():

    new = request.GET.get('task', '').strip()
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()

    c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
    new_id = c.lastrowid

    conn.commit()
    c.close()

    return '<p> 新的待办事项已添加到数据库中,ID 为 %s</p>' % new_id
```

```
else:
    return template('new_task.tpl')
```

对应的 new_task.tpl 模板如下。

```
<p> 添加一个待办事项:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlength="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

如你所见，这个模板只是纯 HTML 的，不包含 Python 代码。这样，我们就完成了添加待办事项这个功能。如果你想通过 POST 来获取数据，那么用 request.POST.get() 来代替 request.GET.get() 就行了。

修改已有待办事项

最后，我们需要做的是修改已有待办事项。

仅使用我们当前了解到的 route 类型，是可以完成这个任务的，但太取巧了。Bottle 还提供了一种 动态 route ，可以更简单地实现。

基本的动态 route 声明如下:

```
@route('/myroute/:something')
```

关键的区别在于那个冒号。它告诉了 Bottle，在下一个 / 之前，:something 可以匹配任何字符串。:something 匹配到的字符串会传递给回调函数，进一步地处理。

在我们的待办事项应用里，我们创建一个 route(@route('edit/:no')), no 是待办事项在数据库里面的 ID。

对应的代码如下。

```
@route('/edit/:no', method='GET')
def edit_item(no):

    if request.GET.get('save','').strip():
        edit = request.GET.get('task','').strip()
        status = request.GET.get('status','').strip()

        if status == 'open':
            status = 1
        else:
            status = 0

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit, status, no))
        conn.commit()

        return '<p>ID 为%s的待办事项已更新 </p>' % no
    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no)))
        cur_data = c.fetchone()

        return template('edit_task', old=cur_data, no=no)
```

这和之前的添加待办事项类似，主要的不同点在于使用了动态的 route(:no)，它可将 ID 传给 route 对应的回调函数。如你所见，我们在 edit_item 函数中使用了 no ，从数据库中获取数据。

对应的 edit_task.tpl 模板如下。

```
%#template for editing a task
%#the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

再一次，模板中混合了 HTML 代码和 Python 代码，之前已解释过。

你也可在动态 route 中使用正则表达式，稍后会提及。

验证动态 route

在某些场景下，需要验证 route 中的可变部分。例如，在上面的例子中，我们的 no 需要是一个整形数，如果我们的输入是一个浮点数，或字符串，Python 解释器将会抛出一个异常，这并不是我们想要的结果。

对应上述情况，Bottle 提供了一个 @validate 修饰器，可在用户输入被传递给回调函数之前，检验用户数据的合法性。代码例子如下。

```
from bottle import route, run, debug, template, request, validate
...
@route('/edit/:no', method='GET')
@validate(no=int)
def edit_item(no):
...
```

首先，我们从 Bottle 中导入了 validate ，然后在 route 中使用了。在这里，我们验证 no 是否是一个整形数。基本上，validate 可用于其它类型，例如浮点数，列表等等。

保存更改，如果用户提供的 :no 不是一个整形数，而是一个浮点数或其他类型，将返回一个”403 forbidden” 页面，而不是抛出异常。

在动态 route 中使用正则表达式

Bottle 允许在动态 route 中使用正则表达式。

我们假设需要通过 item1 这样的形式来访问数据库中 id 为 1 的待办事项。显然，我们不想为每个待办事项都创建一个 route。鉴于 route 中的”item” 部分是固定的，简单的 route 就无法满足需求了，我们需要在 route 中使用正则表达式。

使用正则表达式的解决方法如下。

```
@route('/item:item#[1-9]+#')
def show_item(item):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
    result = c.fetchall()
    c.close()
    if not result:
        return '该待办事项不存在!'
    else:
        return '待办事项: %s' %result[0]
```

当然，这个例子是我们想象出来的，去掉”item1” 中的”item”，直接使用”1” 会更简单。虽然如此，我们还是想为你展示在 route 的正则表达式: `@route(/item:item_#[1-9]+#)` 和一个普通的 route 差不多，但是在两个”#” 中的字符就是一个正则表达式，匹配从 0 到 9 的数字。在处理”/item9” 这样的请求的时候，正则表达式会匹配到”9”，然后将”9” 做为 item 参数传递给 show_item 函数，而不是”item9”。

返回静态文件

有时候，我们只是想返回已有的静态文件。例如我们的应用中有个静态的帮助页面 help.html，我们不希望每次访问帮助页面的时候都动态生成。

```
from bottle import route, run, debug, template, request, validate, static_file

@route('/help')
def help():
    return static_file('help.html', root='/path/to/file')
```

首先，我们需要从 Bottle 中导入 `static_file` 函数。它接受至少两个参数，一个是需要返回的文件的文件名，一个是该文件的路径。即使该文件和你的应用在同一个目录下，还是要指定文件路径 (可以使用”.”)。Bottle 会猜测文件的 MIME 类型，并自动设置。如果你想显式指定 MIME 类型，可以在 static_file 函数里面加上例如 `mimetype='text/html'` 这样的参数。`static_file` 函数可和任何 route 配合使用，包括动态 route。

返回 JSON 数据

有时我们希望返回 JSON，以便在客户端使用 JavaScript 来生成页面，Bottle 直接支持返回 JSON 数据了。

我们假设现在需要返回 JSON 数据。

```
@route('/json:json#[1-9]+#')
def show_json(json):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (json))
    result = c.fetchall()
    c.close()

    if not result:
        return {'task':'对应的待办事项不存在'}
    else:
        return {'Task': result[0]}
```

很简单，只需要返回一个 Python 中的字典就可以了，Bottle 会自动将其转换为 JSON，再传输到客户端。如果你访问”http://localhost/json1“，你应能得到 `{"Task": ["Read A-byte-of-python to get a good introduction into Python"]}` 类型的 JSON 数据。

捕获错误

为了避免用户看到出错信息，我们需要捕获应用运行时出现的错误，以提供更友好的错误提示。Bottle 提供了专门用于捕获错误的 route。

例如，我们想捕获 403 错误。

```
from bottle import error

@error(403)
def mistake(code):
    return '参数格式错误！'
```

首先，我们需要从 Bottle 中导入 `error` ，然后通过 `error(403)` 来定义创建一个 route，用于捕获所有"403 forbidden" 错误。注意，该 route 总是会将 error-code 传给 `mistake()` 函数，即使你不需要它。所以回调函数至少要接受一个参数，否则会失效。

一样的，同一个回调函数可以捕获多种错误。

```
@error(404)
@error(403)
def mistake(code):
    return '出错啦！'
```

效果和下面一样。

```
@error(403)
def mistake403(code):
    return '参数格式错误！'

@error(404)
def mistake404(code):
    return '该页面不存在！'
```

总结

通过以上章节，你应该对 Bottle 框架有了一个大致的了解，可以使用 Bottle 进行开发了。

接下来的章节会简单介绍一下，如何在大型项目中使用 Bottle。此外，我们还会介绍如何将 Bottle 部署到更高性能的 Web 服务器上。

2.1.4 部署服务器

到目前为止，我们还是使用 Bottle 内置的，随 Python 一起发布的 WSGI reference Server 服务器。尽管该服务器十分适合用于开发环境，但是它确实不适用于大项目。在我们介绍其他服务器之前，我们先看看如何优化内置服务器的设置。

更改服务器的端口和 IP

默认的，Bottle 会监听 127.0.0.1(即 `localhost`) 的 8080 端口。如果要更改该设置，更改 `run` 函数的参数即可。

更改端口，监听 80 端口

```
run(port=80)
```

更改 IP 地址

```
run(host='123.45.67.89')
```

可同时使用

```
run(port=80, host='123.45.67.89')
```

当 Bottle 运行在其他服务器上面时，`port` 和 `host` 参数依然适用，稍后会介绍。

在其他服务器上运行

在大型项目上，Bottle 自带的服务器会成为一个性能瓶颈，因为它是单线程的，一次只能响应一个请求。Bottle 已经可以工作在很多多线程的服务器上面了，例如 Cherrypy, Fapws3, Flup 和 Paste ，所以我们建议在大型项目上使用高性能的服务器。

如果想运行在 Paste 服务器上面，代码如下 (译者注：需要先安装 Paste)。

```
from bottle import PasteServer
...
run(server=PasteServer)
```

其他服务器如 FlupServer, CherryPyServer 和 FapwsServer 也类似。

使用 mod_wsgi 运行在 Apache 上

或许你已经有了一个 Apache 服务器，那么可以考虑使用 mod_wsgi 。

我们假设你的 Apache 已经能跑起来，且 mod_wsgi 也能工作了。在很多 Linux 发行版上，都能通过包管理软件简单地安装 mod_wsgi。

Bottle 已经自带用于 mod_wsgi 的适配器，所以让 Bottle 跑在 mod_wsgi 上面是很简单的。

接下来的例子里，我们假设你希望通过 http://www.mypage.com/todo 来访问"ToDo list" 这个应用，且代码、模板、和 SQLite 数据库存放在 /var/www/todo 目录。

如果通过 mod_wsgi 来运行你应用，那么必须从代码中移除 run() 函数。

然后，创建一个 adapter.wsgi 文件，内容如下。

```
import sys, os, bottle

sys.path = ['/var/www/todo/'] + sys.path
os.chdir(os.path.dirname(__file__))

import todo # This loads your application

application = bottle.default_app()
```

将其保存到 /var/www/todo 目录下面。其实，可以给该文件起任何名字，只要后缀名为 .wsgi 即可。

最后，我们需要在 Apache 的配置中添加一个虚拟主机。

```
<VirtualHost *>
    ServerName mypage.com

    WSGIDaemonProcess todo user=www-data group=www-data processes=1 threads=5
    WSGIScriptAlias / /var/www/todo/adapter.wsgi

    <Directory /var/www/todo>
        WSGIProcessGroup todo
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

重启 Apache 服务器后，即可通过 http://www.mypage.com/todo 来访问你的应用。

2.1.5 结语

现在，我们这个教程已经结束了。我们学习了 Bottle 的基础知识，然后使用 Bottle 来写了第一个应用。另外，我们还介绍了如何在大型项目中使用 Bottle，以及使用 mod_wsgi 在 Apache 中运行 Bottle 应用。

我们并没有在这份教程里介绍 Bottle 的方方面面。我们没有介绍如何上传文件，验证数据的可靠性。还有，我们也没介绍如何在模板中调用另一个模板。以上，可以在 Bottle documentation 中找到答案。

2.1.6 完整代码

译者注：以下内容不翻译

todo.py 文件

```
import sqlite3
from bottle import route, run, debug, template, request, validate, static_file, error

# only needed when you run Bottle on mod_wsgi
from bottle import default_app

@route('/todo')
def todo_list():

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT id, task FROM todo WHERE status LIKE '1';")
    result = c.fetchall()
    c.close()

    output = template('make_table', rows=result)
    return output

@route('/new', method='GET')
def new_item():

    if request.GET.get('save','').strip():

        new = request.GET.get('task', '').strip()
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()

        c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
        new_id = c.lastrowid

        conn.commit()
        c.close()

        return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id

    else:
        return template('new_task.tpl')

@route('/edit/:no', method='GET')
@validate(no=int)
def edit_item(no):

    if request.GET.get('save','').strip():
        edit = request.GET.get('task','').strip()
        status = request.GET.get('status','').strip()

        if status == 'open':
            status = 1
        else:
            status = 0

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit,status,no))
        conn.commit()

        return '<p>The item number %s was successfully updated</p>' %no
```

```
    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no)))
        cur_data = c.fetchone()

        return template('edit_task', old = cur_data, no = no)

@route('/item:item#[1-9]+#')
def show_item(item):

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
        result = c.fetchall()
        c.close()

        if not result:
            return 'This item number does not exist!'
        else:
            return 'Task: %s' %result[0]

@route('/help')
def help():

    static_file('help.html', root='.')

@route('/json:json#[1-9]+#')
def show_json(json):

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (json))
    result = c.fetchall()
    c.close()

    if not result:
        return {'task':'This item number does not exist!'}
    else:
        return {'Task': result[0]}


@error(403)
def mistake403(code):
    return 'There is a mistake in your url!'

@error(404)
def mistake404(code):
    return 'Sorry, this page does not exist!'


debug(True)
run(reloader=True)
#remember to remove reloader=True and debug(True) when you move your application from development to a producti
```

Template `make_table.tpl`:

```
%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or ...)
<p>The open items are as follows:</p>
<table border="1">
%for row in rows:
  <tr>
```

```
  %for col in row:
    <td>{{col}}</td>
  %end
  </tr>
%end
</table>
```

Template `edit_task.tpl`:

```
%#template for editing a task
%#the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

Template `new_task.tpl`:

```
%#template for the form for a new task
<p>Add a new task to the ToDo list:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlength="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

2.2 异步应用入门

异步设计模式和 WSGI 的同步本质并不能很好地兼容。这就是为什么大部分的异步框架 (tornado, twisted, ...) 都实现了专有的 API 来暴露它们的异步特征。Bottle 是一个 WSGI 框架，也继承了 WSGI 的同步本质，但是谢谢优秀的 gevent 项目 ，我们可以使用 Bottle 来编写异步应用了。这份文档介绍了在 Bottle 如何使用异步 WSGI。

2.2.1 同步 WSGI 的限制

简单来说，WSGI 标准 (pep 3333) 定义了下面这一个 request/response 的循环：每次请求到达的时候，应用中的 callable 会被调用一次，返回一个主体 iterator。接着服务器会遍历该主体，分块写入 socket。遍历完整个主体，就关闭客户端的连接。

足够简单，但是存在一个小问题：所有这些都是同步的。如果你的应用需要等待数据 (IO, socket, 数据库, ...)，除了返回一个空字符串 (忙等)，就只能阻塞当前进程了。两种办法都会占用当前进程，这样它就不能处理新的请求了。这样就导致每个线程只能处理一个请求了。

大部分服务器都限制了线程的数量，避免伴随它们而来的资源消耗。常见的是一个线程池，内有 20 个或更少数量的线程。一旦所有的线程都被占用了，任何新的请求都会阻塞。事实上，对于其他人来说，服务器已经宕机了。如果你想实现一个聊天程序，使用 ajax 轮询来获取实时消息，很快你就会受到线程数量的限制。这个程序也未免太小众了。

2.2.2 救星，Greenlet

大多数服务器的线程池都限制了线程池中线程的数量，避免创建和切换线程的代价。尽管和进程 (fork) 比起来，线程还是挺便宜的。但是也没便宜到可以接受为每一个请求创建一个线程。

gevent 模块添加了 *greenlet* 的支持。greenlet 和传统的线程类似，但其创建只需消耗很少的资源。基于 gevent 的服务器可以生成成千上万的 greenlet，为每个连接分配一个 greenlet 也毫无压力。阻塞 greenlet，也不会影响到服务器接受新的请求。同时处理的连接数理论上是没有限制的。

这令创建异步应用难以置信的简单，因为它们看起来很想同步程序。基于 gevent 服务器实际上不是异步的，是大规模多线程。下面是一个例子。

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

from time import sleep
from bottle import route, run

@route('/stream')
def stream():
    yield 'START'
    sleep(3)
    yield 'MIDDLE'
    sleep(5)
    yield 'END'

run(host='0.0.0.0', port=8080, server='gevent')
```

第一行很重要。它让 gevent monkey-patch 了大部分 Python 的阻塞 API，让它们不阻塞当前经常，将 CPU 让给下一个 greenlet。它实际上用基于 gevent 的伪线程替换了 Python 的线程。这就是你依然可以使用 `time.sleep()` 这个照常来说会阻塞线程的函数。如果这种 monkey-patch 的方式感令你感到不舒服，你依然可以使用 gevent 中相应的函数 `gevent.sleep()` 。

如果你运行了上面的代码，接着访问 `http://localhost:8080/stream` ，你可看到 *START*, *MIDDLE*, 和 *END* 这几个字样依次出现 (用时大约 8 秒)。它像普通的线程一样工作，但是现在你的服务器能同时处理成千上万的连接了。

Note: 一些浏览器在开始渲染一个页面之前，会缓存确定容量的数据。在这些浏览器上，你需要返回更多的数据才能看到效果。另外，很多浏览器限制一个 URL 只使用一个连接。在这种情况下，你可以使用另外的浏览器，或性能测试工具 (例如: *ab* 或 *httperf*) 来测试性能。

2.2.3 事件回调函数

异步框架的常见设计模式 (包括 tornado, twisted, node.js 和 friends)，是使用非阻塞的 API，绑定回调函数到异步事件上面。在显式地关闭之前，socket 会保持打开，以便稍后回调函数往 socket 里面写东西。下面是一个基于 tornado 的例子。

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        worker = SomeAsyncWorker()
        worker.on_data(lambda chunk: self.write(chunk))
        worker.on_finish(lambda: self.finish())
```

主要的好处就是 MainHandler 能早早结束，在回调函数继续写 socket 来响应之前的请求的时候，当前线程能继续接受新的请求。这样就是为什么这类框架能同时处理很多请求，只使用很少的操作系统线程。

对于 Gevent 和 WSGI，情况就不一样了：首先，早早结束没有好处，因为我们的 (伪) 线程池已经没有限制了。第二，我们不能早早结束，因为这样会关闭 socket(WSGI 要求如此)。第三，我们必须返回一个 iterator，以遵守 WSGI 的约定。

为了遵循 WSGI 规范，我们只需返回一个 iterable 的实体，异步地将其写回客户端。在 gevent.queue 的帮助下，我们可以 模拟 一个脱管的 socket，上面的例子可写成这样。

```
@route('/fetch')
def fetch():
    body = gevent.queue.Queue()
```

```
    worker = SomeAsyncWorker()
    worker.on_data(lambda chunk: body.put(chunk))
    worker.on_finish(lambda: body.put(StopIteration))
    return body
```

从服务器的角度来看，queue 对象是 iterable 的。如果为空，则阻塞，一旦遇到 StopIteration 则停止。这符合 WSGI 规范。从应用的角度来看，queue 对象表现的像一个不会阻塞 socket。你可以在任何时刻写入数据，pass it around，甚至启动一个新的 (伪) 线程，异步写入。这是在大部分情况下，实现长轮询。

2.2.4 最后：WebSockets

让我们暂时忘记底层的细节，来谈谈 WebSocket。既然你正在阅读这篇文章，你有可能已经知道什么是 WebSocket 了，一个在浏览器 (客户端) 和 Web 应用 (服务端) 的双向的交流通道。

感谢 gevent-websocket 包帮我们做的工作。下面是一个 WebSocket 的简单例子，接受消息然后将其发回客户端。

```
from bottle import request, Bottle, abort
app = Bottle()

@app.route('/websocket')
def handle_websocket():
    wsock = request.environ.get('wsgi.websocket')
    if not wsock:
        abort(400, 'Expected WebSocket request.')

    while True:
        try:
            message = wsock.receive()
            wsock.send("Your message was: %r" % message)
        except WebSocketError:
            break

from gevent.pywsgi import WSGIServer
from geventwebsocket import WebSocketHandler, WebSocketError
server = WSGIServer(("0.0.0.0", 8080), app,
                    handler_class=WebSocketHandler)
server.serve_forever()
```

while 循环直到客户端关闭连接的时候才会终止。You get the idea :)

客户端的 JavaScript API 也十分简洁明了。

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript">
    var ws = new WebSocket("ws://example.com:8080/websocket");
    ws.onopen = function() {
        ws.send("Hello, world");
    };
    ws.onmessage = function (evt) {
        alert(evt.data);
    };
  </script>
</head>
</html>
```

2.3 秘诀

这里收集了一些常见用例的代码片段和例子.

2.3.1 使用 Session

Bottle 自身并没有提供 Session 的支持，因为在一个迷你框架里面，没有合适的方法来实现。根据需求和使用环境，你可以使用 beaker 中间件或自己来实现。下面是一个使用 beaker 的例子，Session 数据存放在"./data" 目录里面。

```
import bottle
from beaker.middleware import SessionMiddleware

session_opts = {
    'session.type': 'file',
    'session.cookie_expires': 300,
    'session.data_dir': './data',
    'session.auto': True
}
app = SessionMiddleware(bottle.app(), session_opts)

@bottle.route('/test')
def test():
  s = bottle.request.environ.get('beaker.session')
  s['test'] = s.get('test',0) + 1
  s.save()
  return 'Test counter: %d' % s['test']

bottle.run(app=app)
```

2.3.2 Debugging with Style: 调试中间件

Bottle 捕获所有应用抛出的异常，防止异常导致 WSGI 服务器崩溃。如果内置的 `debug()` 模式不能满足你的要求，你想在你自己写的中间件里面处理这些异常，那么你可以关闭这个功能。

```
import bottle
app = bottle.app()
app.catchall = False # 现在，Bottle 会重新抛出所有捕获到的异常
myapp = DebuggingMiddleware(app) # 这里替换成你的中间件
bottle.run(app=myapp)
```

现在，Bottle 仅会捕获并处理它自己抛出的异常（`HTTPError`，`HTTPResponse` 和 `BottleException`），你的中间件可以处理剩下的那些异常。

werkzeug 和 paste 这两个第三方库都提供了非常强大的调试中间件。如果是 werkzeug，可看看 `werkzeug.debug.DebuggedApplication`，如果是 paste，可看看 `paste.evalexception.middleware.EvalException`。它们都可让你检查运行栈，甚至在保持运行栈上下文的情况下，执行 Python 代码。所以 不要在生产环境中使用它们 。

2.3.3 Unit-Testing

Unit 测试一般用于测试应用中的函数，但不需要一个 WSGI 环境。

使用 Nose 的简单例子。

```
import bottle

@bottle.route('/')
```

```
def index():
    return 'Hi!'

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

测试代码:

```
import mywebapp

def test_webapp_index():
    assert mywebapp.index() == 'Hi!'
```

在这个例子中，Bottle 的 route() 函数没有被执行，仅测试了 index() 函数。

2.3.4 Functional Testing

任何基于 HTTP 的测试系统都可用于测试 WSGI 服务器，但是有些测试框架与 WSGI 服务器工作得更好。它们可以在一个可控环境里运行 WSGI 应用，充分利用 traceback 和调试工具。Testing tools for WSGI 是一个很好的上手工具。

使用 WebTest 和 Nose 的例子。

```
from webtest import TestApp
import mywebapp

def test_functional_login_logout():
    app = TestApp(mywebapp.app)

    app.post('/login', {'user': 'foo', 'pass': 'bar'}) # 登录， 获取一个 cookie

    assert app.get('/admin').status == '200 OK'        # 成功抓取一个页面

    app.get('/logout')                                 # 登出
    app.reset()                                        # 丢弃 cookie

    # 抓取同一个页面，失败！
    assert app.get('/admin').status == '401 Unauthorized'
```

2.3.5 嵌入其他 WSGI 应用

并不建议你使用这个方法，你应该在 Bottle 前面使用一个中间件来做这样的事情。但你确实可以在 Bottle 里面调用其他 WSGI 应用，让 Bottle 扮演一个中间件的角色。下面是一个例子。

```
from bottle import request, response, route
subproject = SomeWSGIApplication()

@route('/subproject/:subpath#.*#', method='ALL')
def call_wsgi(subpath):
    new_environ = request.environ.copy()
    new_environ['SCRIPT_NAME'] = new_environ.get('SCRIPT_NAME','') + '/subproject'
    new_environ['PATH_INFO'] = '/' + subpath
    def start_response(status, headerlist):
        response.status = int(status.split()[0])
        for key, value in headerlist:
            response.add_header(key, value)
    return app(new_environ, start_response)
```

再次强调，并不建议使用这种方法。之所以介绍这种方法，是因为很多人问起，如何在 Bottle 中调用 WSGI 应用。

2.3.6 忽略尾部的反斜杠

在 Bottle 看来，/example 和 /example/ 是两个不同的 route [1] 。为了一致对待这两个 URL，你应该添加两个 route。

```
@route('/test')
@route('/test/')
def test(): return '反斜杠? 不?'
```

或者添加一个 WSGI 中间件来处理这种情况。

```
class StripPathMiddleware(object):
  def __init__(self, app):
    self.app = app
  def __call__(self, e, h):
    e['PATH_INFO'] = e['PATH_INFO'].rstrip('/')
    return self.app(e,h)

app = bottle.app()
myapp = StripPathMiddleware(app)
bottle.run(app=myapp)
```

2.3.7 Keep-alive 请求

Note: 详见异步应用入门 。

像 XHR 这样的"push" 机制，需要在 HTTP 响应头中加入 "Connection: keep-alive" ，以便在不关闭连接的情况下，写入响应数据。WSGI 并不支持这种行为，但如果在 Bottle 中使用 gevent 这个异步框架，还是可以实现的。下面是一个例子，可配合 gevent HTTP 服务器或 paste HTTP 服务器使用 (也许支持其他服务器，但是我没试过)。在 run() 函数里面使用 server='gevent' 或 server='paste' 即可使用这两种服务器。

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

import time
from bottle import route, run

@route('/stream')
def stream():
    yield 'START'
    time.sleep(3)
    yield 'MIDDLE'
    time.sleep(5)
    yield 'END'

run(host='0.0.0.0', port=8080, server='gevent')
```

通过浏览器访问 http://localhost:8080/stream ，可看到'START'，'MIDDLE'，和'END' 这三个字眼依次出现，一共用了 8 秒。

2.3.8 Gzip 压缩

Note: 详见 compression

Gzip 压缩，可加速网站静态资源 (例如 CSS 和 JS 文件) 的访问。人们希望 Bottle 支持 Gzip 压缩，(但是不支持)……

[1] 因为确实如此，见 <http://www.ietf.org/rfc/rfc3986.txt>

支持 Gzip 压缩并不简单，一个合适的 Gzip 实现应该满足以下条件。

- 压缩速度要快
- 如果浏览器不支持，则不压缩
- 不压缩那些已经充分压缩的文件 (图像，视频)
- 不压缩动态文件
- 支持两种压缩算法 (gzip 和 deflate)
- 缓存那些不经常变化的压缩文件
- 不验证缓存中那些已经变化的文件 (De-validate the cache if one of the files changed anyway)
- 确保缓存不太大
- 不缓存小文件，因为寻道时间或许比压缩时间还长

因为有上述种种限制，建议由 WSGI 服务器来处理 Gzip 压缩而不是 Bottle。像 cherrypy 就提供了一个 GzipFilter 中间件来处理 Gzip 压缩。

2.3.9 使用钩子

例如，你想提供跨域资源共享，可参考下面的例子。

```
from bottle import hook, response, route

@hook('after_request')
def enable_cors():
    response.headers['Access-Control-Allow-Origin'] = '*'

@route('/foo')
def say_foo():
    return 'foo!'

@route('/bar')
def say_bar():
    return {'type': 'friendly', 'content': 'Hi!'}
```

你也可以使用 `before_callback` ，这样在 route 的回调函数被调用之前，都会调用你的钩子函数。

2.3.10 在 Heroku 中使用 Bottle

Heroku ，一个流行的云应用平台，提供 Python 支持。

这份教程基于 Heroku Quickstart, 用 Bottle 特有的代码替换了 Getting Started with Python on Heroku/Cedar 中的 Write Your App 这部分。

```
import os
from bottle import route, run

@route("/")
def hello_world():
        return "Hello World!"

run(host="0.0.0.0", port=int(os.environ.get("PORT", 5000)))
```

Heroku 使用 *os.environ* 字典来提供 Bottle 应用需要监听的端口。

2.4 常见问题

2.4.1 关于 Bottle

Bottle 适合用于复杂的应用吗？

Bottle 是一个 迷你 框架，被设计来提供原型开发和创建小的 Web 应用和服务。它能快速上手，并帮助你快速完成任务，但缺少一些高级功能和一些其他框架已提供的已知问题解决方法 (例如:MVC，ORM，表单验证，手脚架，XML-RPC)。尽管 可以 添加这些功能，然后通过 Bottle 来开发复杂的应用，但我们还是建议你使用一些功能完备的 Web 框架，例如 pylons 或 paste 。

2.4.2 常见的，意料之外的问题

“Template Not Found” in mod_wsgi/mod_python

Bottle 会在 `./` 和 `./views/` 目录中搜索模板。在一个 mod_python 或 mod_wsgi 环境，当前工作目录 (`./`) 是由 Apache 的设置决定的。你应该在模板的搜索路径中添加一个绝对路径。

```
bottle.TEMPLATE_PATH.insert(0,'/absolut/path/to/templates/')
```

这样，Bottle 就能在正确的目录下搜索模板了

动态 route 和反斜杠

在 *dynamic route syntax* 中，`:name` 匹配任何字符，直到出现一个反斜杠。工作方式相当与 `[^/]+` 这样一个正则表达式。为了将反斜杠包涵进来，你必须在 `:name` 中添加一个自定义的正则表达式。例如: `/images/:filepath#.*#` 会匹配 `/images/icons/error.png` ，但不匹配 `/images/:filename` 。

反向代理的问题

只用 Bottle 知道公共地址和应用位置的情况下，重定向和 url-building 才会起作用。(译者注：保留原文) Redirects and url-building only works if bottle knows the public address and location of your application. 如果 Bottle 应用运行在反向代理或负载均衡后面，一些信息也许会丢失。例如，`wsgi.url_scheme` 的值或 `Host` 头或许只能获取到代理的请求信息，而不是真实的用户请求。下面是一个简单的中间件代码片段，处理上面的问题，获取正确的值。

```
def fix_environ_middleware(app):
  def fixed_app(environ, start_response):
    environ['wsgi.url_scheme'] = 'https'
    environ['HTTP_X_FORWARDED_HOST'] = 'example.com'
    return app(environ, start_response)
  return https_app

app = bottle.default_app()
app.wsgi = fix_environ_middleware(app.wsgi)
```

CHAPTER 3

开发和贡献

这些章节是为那些对 Bottle 的开发和发布流程感兴趣的开发者准备的。

3.1 发布摘要和更改历史 (不译)

3.1.1 Release 0.11

- Native support for Python 2.x and 3.x syntax. No need to run 2to3 anymore.
- Support for partial downloads (`Range` header) in `static_file()`.
- The new `ResourceManager` interface helps locating files bundled with the application.
- Added a server adapter for waitress.
- New `Bottle.merge()` method to install all routes from one application into another.
- New `BaseRequest.app` property to get the application object that handles that request.
- Added `FormsDict.decode()` to get an all-unicode version (needed by WTForms).
- `MultiDict` and subclasses are now pickle-able.

API Changes

- `Response.status` is a read-write property that can be assigned either a numeric status code or a status string with a reason phrase (`200 OK`). The return value is now a string to better match existing APIs (WebOb, werkzeug). To be absolutely clear, you can use the read-only properties `BaseResponse.status_code` and `BaseResponse.status_line`.

API Deprecations

- `SimpleTALTemplate` is now deprecating. There seems to be no demand.

3.1.2 Release 0.10

- Plugin API v2
  - To use the new API, set `Plugin.api` to 2.

  - `Plugin.apply()` receives a `Route` object instead of a context dictionary as second parameter. The new object offers some additional information and may be extended in the future.
  - Plugin names are considered unique now. The topmost plugin with a given name on a given route is installed, all other plugins with the same name are silently ignored.
- The Request/Response Objects
  - Added `BaseRequest.json`, `BaseRequest.remote_route`, `BaseRequest.remote_addr`, `BaseRequest.query` and `BaseRequest.script_name`.
  - Added `BaseResponse.status_line` and `BaseResponse.status_code` attributes. In future releases, `BaseResponse.status` will return a string (e.g. `200 OK`) instead of an integer to match the API of other common frameworks. To make the transition as smooth as possible, you should use the verbose attributes from now on.
  - Replaced `MultiDict` with a specialized `FormsDict` in many places. The new dict implementation allows attribute access and handles unicode form values transparently.
- Templates
  - Added three new functions to the SimpleTemplate default namespace that handle undefined variables: `stpl.defined()`, `stpl.get()` and `stpl.setdefault()`.
  - The default escape function for SimpleTemplate now additionally escapes single and double quotes.
- Routing
  - A new route syntax (e.g. `/object/<id:int>`) and support for route wildcard filters.
  - Four new wildcard filters: *int*, *float*, *path* and *re*.
- Oher changes
  - Added command line interface to load applications and start servers.
  - Introduced a `ConfigDict` that makes accessing configuration a lot easier (attribute access and auto-expanding namespaces).
  - Added support for raw WSGI applications to `Bottle.mount()`.
  - `Bottle.mount()` parameter order changed.
  - `Bottle.route()` now accpets an import string for the `callback` parameter.
  - Dropped Gunicorn 0.8 support. Current supported version is 0.13.
  - Added custom options to Gunicorn server.
  - Finally dropped support for type filters. Replace with a custom plugin of needed.

3.1.3 Release 0.9

Whats new?

- A brand new plugin-API. See **插件** and **插件开发指南** for details.
- The `route()` decorator got a lot of new features. See `Bottle.route()` for details.
- New server adapters for gevent, meinheld and bjoern.
- Support for SimpleTAL templates.
- Better runtime exception handling for mako templates in debug mode.
- Lots of documentation, fixes and small improvements.
- A new `Request.urlparts` property.

Performance improvements

- The `Router` now special-cases `wsgi.run_once` environments to speed up CGI.
- Reduced module load time by ~30% and optimized template parser. See 8ccb2d, f72a7c and b14b9a for details.
- Support for "App Caching" on Google App Engine. See af93ec.
- Some of the rarely used or deprecated features are now plugins that avoid overhead if the feature is not used.

API changes

This release is mostly backward compatible, but some APIs are marked deprecated now and will be removed for the next release. Most noteworthy:

- The `static` route parameter is deprecated. You can escape wild-cards with a backslash.
- Type-based output filters are deprecated. They can easily be replaced with plugins.

3.1.4 Release 0.8

API changes

These changes may break compatibility with previous versions.

- The built-in Key/Value database is not available anymore. It is marked deprecated since 0.6.4
- The Route syntax and behaviour changed.
  - Regular expressions must be encapsulated with `#`. In 0.6 all non-alphanumeric characters not present in the regular expression were allowed.
  - Regular expressions not part of a route wildcard are escaped automatically. You don't have to escape dots or other regular control characters anymore. In 0.6 the whole URL was interpreted as a regular expression. You can use anonymous wildcards (`/index:#(\.html)?#`) to achieve a similar behaviour.
- The `BreakTheBottle` exception is gone. Use `HTTPResponse` instead.
- The `SimpleTemplate` engine escapes HTML special characters in `{{bad_html}}` expressions automatically. Use the new `{{!good_html}}` syntax to get old behaviour (no escaping).
- The `SimpleTemplate` engine returns unicode strings instead of lists of byte strings.
- `bottle.optimize()` and the automatic route optimization is obsolete.
- Some functions and attributes were renamed:
  - `Request._environ` is now `Request.environ`
  - `Response.header` is now `Response.headers`
  - `default_app()` is obsolete. Use `app()` instead.
- The default `redirect()` code changed from 307 to 303.
- Removed support for `@default`. Use `@error(404)` instead.

New features

This is an incomplete list of new features and improved functionality.

- The `Request` object got new properties: `Request.body`, `Request.auth`, `Request.url`, `Request.header`, `Request.forms`, `Request.files`.
- The `Response.set_cookie()` and `Request.get_cookie()` methods are now able to encode and decode python objects. This is called a *secure cookie* because the encoded values are signed and protected from changes on client side. All pickle-able data structures are allowed.
- The new `Router` class drastically improves performance for setups with lots of dynamic routes and supports named routes (named route + dict = URL string).
- It is now possible (and recommended) to return `HTTPError` and `HTTPResponse` instances or other exception objects instead of raising them.
- The new function `static_file()` equals `send_file()` but returns a `HTTPResponse` or `HTTPError` instead of raising it. `send_file()` is deprecated.
- New `get()`, `post()`, `put()` and `delete()` decorators.
- The `SimpleTemplate` engine got full unicode support.
- Lots of non-critical bugfixes.

3.2 Contributors

Bottle is written and maintained by Marcel Hellkamp <marc@bottlepy.org>.

Thanks to all the people who found bugs, sent patches, spread the word, helped each other on the mailing-list and made this project possible. I hope the following (alphabetically sorted) list is complete. If you miss your name on that list (or want your name removed) please `tell me` or add it yourself.

- acasajus
- Adam R. Smith
- Alexey Borzenkov
- Alexis Daboville
- Anton I. Sipos
- Anton Kolechkin
- apexi200sx
- apheage
- BillMa
- Brad Greenlee
- Brandon Gilmore
- Branko Vukelic
- Brian Sierakowski
- Brian Wickman
- Carl Scharenberg
- Damien Degois
- David Buxton
- Duane Johnson

- fcamel
- Frank Murphy
- Frederic Junod
- goldfaber3012
- Greg Milby
- gstein
- Ian Davis
- Itamar Nabriski
- Iuri de Silvio
- Jaimie Murdock
- Jeff Nichols
- Jeremy Kelley
- joegester
- Johannes Krampf
- Jonas Haag
- Joshua Roesslein
- Karl
- Kraken
- Kyle Fritz
- m35
- Marcos Neves
- masklinn
- Michael Labbe
- Michael Soulier
- reddit
- Robert Rollins
- rogererens
- rwxrwx
- Santiago Gala
- Sean M. Collins
- Sebastian Wollrath
- Seth
- Sigurd Hgsbro
- Stuart Rackham
- Sun Ning
- Tom. Schertel
- Tristan Zajonc
- voltron
- Wieland Hoffmann

- zombat

3.3 开发者笔记

这份文档是为那些对 Bottle 的开发和发布流程感兴趣的开发者和软件包维护者准备的。如果你想要做贡献，看它是对的！

3.3.1 参与进来

有多种加入社区的途径，保持消息的灵通。这里是一些方法:

- 邮件列表: 发送邮件到 bottlepy+subscribe@googlegroups.com 以加入我们的邮件列表 (无需 Google 账户)
- **Twitter**: 在 Twitter 上关注我们或搜索 bottlepy 标签
- **IRC**: 加入 #irc.freenode.net 上的 bottlepy 频道或使用 web 聊天界面
- **Google plus**: 有时我们会在 Google+ 页面上发表一些 与 Bottle 有关的博客

3.3.2 获取源代码

Bottle 的 开发仓库 和 问题追踪 都搭建在 github 上面。如果你打算做贡献，创建一个 github 帐号然后 fork 一下吧。你做出的更改或建议都可被其他开发者看到，可以一起讨论。即使没有 github 帐号，你也可以 clone 代码仓库，或下载最新开发版本的压缩包。

- **git:** `git clone git://github.com/defnull/bottle.git`
- **git/https:** `git clone https://github.com/defnull/bottle.git`
- **Download:** Development branch as tar archive or zip file.

3.3.3 发布和更新

Bottle 不定时发布正式版本，通过 PyPi 来分发。RC 版或 bugfix 版本只会在 github 上面发布。一些 Linux 发行版也许会提供一些过时的版本。

Bottle 的版本号分隔为三个部分 (**major.minor.revision**)。版本号 不会 用于引入新特征，只是为了指出重要的 bug-fix 或 API 的改动。关键的 bug 会在近两个新的版本中修复，然后通过所有渠道发布 (邮件列表, twitter, github)。不重要的 bug 修复或特征不保证添加到旧版本中。这个情况在未来也许会改变。

Major 发布版本 (x.0) 在重要的里程碑达成或新的更新和旧版本彻底不兼容时，会增加 major 版本号。为了使用新版本，你也许需要修改整个应用，但 major 版本号极少会改变。

Minor 发布版本 (x.y) 在更改了 API 之后，或增加 minor 版本号。你可能会收到一些 API 已过时的警告，调整设置以继续使用旧的特征，但在大多数情况下，这些更新都会对旧版本兼容。你应当保持更新，但这不是必须的。0.8 版本是一个特例，它 不兼容 旧版本 (所以我们跳过了 0.7 版本直接发布 0.8 版本)，不好意思。

Revision 发布版本 (x.y.z) 在修复了一些 bug，和改动不会修改 API 的时候，会增加 revision 版本号。你可以放心更新，而不用修改你应用的代码。事实上，你确实应该更新这类版本，因为它常常修复一些安全问题。

Pre-Release 发布版本 RC 版本会在版本号中添加 `rc` 字样。API 已基本稳定，已经开放测试，但还没正式发布。你不应该在生产环境中使用 rc 版本。

3.3.4 代码仓库结构

代码仓库的结构如下:

`master` 分支 该分支用于集成，测试和开发。所有计划添加到下一版本的改动，会在这里合并和测试。

`release-x.y` 分支 只要 master 分支已经可以用来发布一个新的版本，它会被安排到一个新的发行分支里面。在 RC 阶段，不再添加或删除特征，只接受 bug-fix 和小改动，直到认为它可以用于生产环境和正式发布。基于这点考虑，我们称之为“支持分支 (support branch)”，依然接受 bug-fix，但仅限关键的 bug-fix。每次更改后，版本号都会更新，这样你就能及时获取重要的改动了。

`bugfix_name-x.y` 分支 这些分支是临时性的，用于修复现有发布版本的 bug。在合并到其他分支后，它们就会被删除。

特征分支 所有这类分支都是用于新增特征的。基于 master 分支，在开发、合并完毕后，就会被删除。

对于开发者，这意味着什么？

如果你想添加一个特征，可以从 `master` 分支创建一个分支。如果你想修复一个 bug，可从 `release-x.y` 这类分支创建一个分支。无论是添加特征还是修复 bug，都建议在一个独立的分支上进行，这样合并工作就简单了。就这些了！在页面底部会有 git 工作流程的例子。

Oh，请不要修改版本号。我们会在集成的时候进行修改。因为你不知道我们什么时候会将你的 request pull 下来:)

对于软件包维护者，这意味着什么？

关注那些 bugfix 和新版本的 tag，还有邮件列表。如果你想从代码仓库中获取特定的版本，请使用 tag，而不是分支。分支中也许会包含一些未发布的改动，但 tag 会标记是那个 commit 更改了版本号。

3.3.5 提交补丁

让你的补丁被集成进来的最好方法，是在 github 上面 fork 整个项目，创建一个新的分支，修改代码，然后发送一个 pull-request。页面下方是 git 工作流程的例子，也许会有帮助。提交 git 兼容的补丁文件到邮件列表也是可以的。无论使用什么方法，请遵守以下的基本规则:

- 文档： 告诉我们你的补丁做了什么。注释你的代码。如果你添加了新的特征，请添加相应的使用方法。
- 测试： 编写测试以证明你的补丁如期工作，且没有破坏任何东西。如果你修复了一个 bug，至少写一个测试用例来触发这个 bug。在提交补丁之前，请确保所有测试已通过。
- 一次只提交一个补丁： 一次只修改一个 bug，一次只添加一个新特征。保持补丁的干净。
- 与上流同步： 如果在你编写补丁的时候，`upstream/master` 分支被修改了，那么先 rebase 或将新的修改 pull 下来，确保你的补丁在合并的时候不会造成冲突。

3.3.6 生成文档

你需要一个 Sphinx 的新版本来生产整份文档。建议将 Sphinx 安装到一个 **virtualenv** 环境。

```
# Install prerequisites
which virtualenv || sudo apt-get install python-virtualenv
virtualenv --no-site-dependencies venv
./venv/pip install -U sphinx

# Clone or download bottle from github
git clone https://github.com/defnull/bottle.git

# Activate build environment
```

```
source ./venv/bin/activate

# 生成 HTML 格式的文档
cd bottle/docs
make html

# 可选：安装生成 PDF 所需软件包
sudo apt-get install texlive-latex-extra \
                     texlive-latex-recommended \
                     texlive-fonts-recommended

# 可选：生成 PDF 文件
make latex
cd ../build/docs/latex
make pdf
```

3.3.7 GIT 工作流程

接下来的例子都假设你已经有一个 git 的免费帐号 。虽然不是必须的，但可简单化很多东西。

首先，你需要从官方代码仓库创建一个 fork。只需在 bottle 项目页面 点击一下"fork" 按钮就行了。创建玩 fork 之后，会得到一个关于这个新仓库的简介。

你刚刚创建的 fork 托管在 github 上面，对所有人都是可见的，但只有你有修改的权限。现在你需要将其从线上 clone 下面，做出实际的修改。确保你使用的是 (可写 -可读) 的私有 URL，而不是 (只读) 的公开 URL。

```
git clone git@github.com:your_github_account/bottle.git
```

在你将代码仓库 clone 下来后，就有了一个"origin" 分支，指向你在 github 上的 fork。不要让名字迷惑了你，它并不指向 bottle 的官方代码仓库，只是指向你自己的 fork。为了追踪官方的代码仓库，可添加一个新的"upstream" 远程分支。

```
cd bottle
git remote add upstream git://github.com/defnull/bottle.git
git fetch upstream
```

注意，"upstream" 分支使用的是公开的 URL，是只读的。你不能直接往该分支 push 东西，而是由我们来你的公开代码仓库 pull，后面会讲到。

提交一个特征

在独立的特征分支内开发新的特征，会令集成工作更简单。因为它们会被合并到 master 分支，所有它们必须是基于 upstream/master 的分支。下列命令创建一个特征分支。

```
git checkout -b cool_feature upstream/master
```

现在可开始写代码，写测试，更新文档。在提交更改之前，记得确保所有测试已经通过。

```
git commit -a -m "Cool Feature"
```

与此同时，如果 upstream/master 这个分支有改动，那么你的提交就有可能造成冲突，可通过 rebase 操作来解决。

```
git fetch upstream
git rebase upstream
```

这相当于先撤销你的所有改动，更新你的分支到最新版本，再重做你的所有改动。如果你已经发布了你的分支 (下一步会提及)，这就不是一个好主意了，因为会覆写你的提交历史。这种情况下，你应该先将最新版本 pull 下来，手动解决所有冲突，运行测试，再提交。

现在，你已经做好准备发一个 pull-request 了。但首先你应该公开你的特征分支，很简单，将其 push 到你 github 的 fork 上面就行了。

```
git push origin cool_feature
```

在你 push 完你所有的 commit 之后，你需要告知我们这个新特征。一种办法是通过 github 发一个 pull-request。另一种办法是把这个消息发到邮件列表，这也是我们推荐的方式，这样其他开发者就能看到和讨论你的补丁，你也能免费得到一些反馈:)

如果我们接受了你的补丁，我们会将其集成到官方的开发分支中，它将成为下个发布版本的一部分。

修复 Bug

修复 Bug 和添加一个特征差不多，下面是一些不同点:

1. 修复所有受影响的分支，而不仅仅是开发分支 (Branch off of the affected release branches instead of just the development branch)。
2. 至少编写一个触发该 Bug 的测试用例。
3. 修复所有受影响的分支，包括 `upstream/master` ，如果它也受影响。`git cherry-pick` 可帮你完成一些重复工作。
4. 名字后面要加上其修复的版本号，以防冲突。例子: `my_bugfix-x.y` 或 `my_bugfix-dev` 。

3.4 插件开发指南

这份指南介绍了插件的 API，以及如何编写自己的插件。我建议先阅读插件 这一部分，再看这份指南。可用插件列表 这里也有一些实际的例子。

Note: 这是一份初稿。如果你发现了任何错误，或某些部分解释的不够清楚，请通过 邮件列表 或 bug report 告知。

3.4.1 插件工作方式：基础知识

插件的 API 是通过 Python 的 修饰器 来实现的。简单来说，一个插件就是应用在 route 回调函数上的修饰器。

当然，例子被我们简化了，除了作为 route 的回调函数修饰器，插件还可以做更多的事情，先看看代码吧。

```
from bottle import response, install
import time

def stopwatch(callback):
    def wrapper(*args, **kwargs):
        start = time.time()
        body = callback(*args, **kwargs)
        end = time.time()
        response.headers['X-Exec-Time'] = str(end - start)
        return body
    return wrapper

bottle.install(stopwatch)
```

这个插件计算每次请求的响应时间，并在响应头中添加了 `X-Exec-Time` 字段。如你所见，插件返回了一个 wrapper 函数，由它来调用原先的回调函数。这就是修饰器的常见工作方式了。

最后一行，将该插件安装到 Bottle 的默认应用里面。这样，应用中的所有 route 都会应用这个插件了。就是说，每次请求都会调用 `stopwatch()` ，更改了 route 的默认行为。

插件是按需加载的，就是在 route 第一次被访问的时候加载。为了在多线程环境下工作，插件应该是线程安全的。在大多数情况下，这都不是一个问题，但务必提高警惕。

一旦 route 中使用了插件后，插件中的回调函数会被缓存起来，接下来都是直接使用缓存中的版本来响应请求。意味着每个 route 只会请求一次插件。在应用的插件列表变化的时候，这个缓存会被清空。你的插件应当可以多次修饰同一个 route。

这种修饰器般的 API 受到种种限制。你不知道 route 或相应的应用对象是如何被修饰的，也不知道如何有效地存储那些在 route 之间共享的数据。但别怕！插件不仅仅是修饰器函数。只要一个插件是 callable 的或实现了一个扩展的 API(后面会讲到)，Bottle 都可接受。扩展的 API 给你更多的控制权。

3.4.2 插件 API

`Plugin` 类不是一个真正的类 (你不能从 bottle 中导入它)，它只是一个插件需要实现的接口。只要一个对象实现了以下接口，Bottle 就认可它作为一个插件。

class `Plugin`(*object*)
插件应该是 callable 的，或实现了 `apply()` 方法。如果定义了 `apply()` 方法，那么会优先调用，而不是直接调用插件。其它的方法和属性都是可选的。

`name`
`Bottle.uninstall()` 方法和 `Bottle.route()` 中的 *skip* 参数都接受一个与名字有关的字符串，对应插件或其类型。只有插件中有一个 name 属性的时候，这才会起作用。

`api`
插件的 API 还在逐步改进。这个整形数告诉 Bottle 使用哪个版本的插件。如果没有这个属性，Bottle 默认使用第一个版本。当前版本是 2 。详见插件 *API* 的改动 。

`setup`(*self*, *app*)
插件被安装的时候调用 (见 `Bottle.install()`)。唯一的参数是相应的应用对象。

`__call__`(*self*, *callback*)
如果没有定义 `apply()` 方法，插件本身会被直接当成一个修饰器使用 (译者注：Python 的 Magic Method，调用一个类即是调用类的 ___call___ 函数)，应用到各个 route。唯一的参数就是其所修饰的函数。这个方法返回的东西会直接替换掉原先的回调函数。如果无需如此，则直接返回未修改过的回调函数即可。

`apply`(*self*, *callback*, *route*)
如果存在，会优先调用，而不调用 `__call__()` 。额外的 *route* 参数是 `Route` 类的一个实例，提供很多该 route 信息和上下文。详见 *Route* 上下文 。

`close`(*self*)
插件被卸载或应用关闭的时候被调用，详见 `Bottle.uninstall()` 或 `Bottle.close()` 。

`Plugin.setup()` 方法和 `Plugin.close()` 方法 不 会被调用，如果插件是通过 `Bottle.route()` 方法来应用到 route 上面的，但会在安装插件的时候被调用。

插件 API 的改动

插件的 API 还在不断改进中。在 Bottle 0.10 版本中的改动，定位了 route 上下文字典中已确定的问题。为了保持对 0.9 版本插件的兼容，我们添加了一个可选的 `Plugin.api` 属性，告诉 Bottle 使用哪个版本的 API。API 之间的不同点总结如下。

- **Bottle 0.9 API 1** (无 `Plugin.api` 属性)
  - Original Plugin API as described in the 0.9 docs.
- **Bottle 0.10 API 2** (`Plugin.api` 属性为 2)
  - `Plugin.apply()` 方法中的 *context* 参数，现在是 `Route` 类的一个实例，不再是一个上下文字典。

3.4.3 Route 上下文

Route 的实例被传递给 Plugin.apply() 函数，以提供更多该 route 的相关信息。最重要的属性总结如下。

属性	描述
app	安装该 route 的应用对象
rule	route 规则的字符串 (例如：/wiki/:page)
method	HTTP 方法的字符串 (例如：GET)
callback	未应用任何插件的原始回调函数，用于内省。
name	route 的名字，如未指定则为 None
plugins	route 安装的插件列表，除了整个应用范围内的插件，额外添加的 (见 Bottle.route())
skiplist	应用安装了，但该 route 没安装的插件列表 (见 meth:*Bottle.route*)
config	传递给 Bottle.route() 修饰器的额外参数，存在一个字典中，用于特定的设置和元数据

对你的应用而言，Route.config 也许是最重要的属性了。记住，这个字典会在所有插件中共享，建议添加一个独一无二的前缀。如果你的插件需要很多设置，将其保存在 *config* 字典的一个独立的命名空间吧。防止插件之间的命名冲突。

改变 Route 对象

Route 的一些属性是不可变的，改动也许会影响到其它插件。坏主意就是，monkey-patch 一个损坏的 route，而不是提供有效的帮助信息来让用户修复问题。

在极少情况下，破坏规则也许是恰当的。在你更改了 Route 实例后，抛一个 RouteReset 异常。这会从缓存中删除当前的 route，并重新应用所有插件。无论如何，router 没有被更新。改变 *rule* 或 *method* 的值并不会影响到 router，只会影响到插件。这个情况在将来也许会改变。

3.4.4 运行时优化

插件应用到 route 以后，被插件封装起来的回调函数会被缓存，以加速后续的访问。如果你的插件的行为依赖一些设置，你需要在运行时更改这些设置，你需要在每次请求的时候读取设置信息。够简单了吧。

然而，为了性能考虑，也许值得根据当前需求，选择一个不同的封装，通过闭包，或在运行时使用、禁用一个插件。让我们拿内置的 HooksPlugin 作为一个例子 (译者注：可在 bottle.py 搜索该实现)：如果没有安装任何钩子，这个插件会从所有受影响的 route 中删除自身，不做任何工作。一旦你安装了第一个钩子，这个插件就会激活自身，再次工作。

为了达到这个目的，你需要控制回调函数的缓存：Route.reset() 函数清空单一 route 的缓存，Bottle.reset() 函数清空所有 route 的缓存。在下一次请求的时候，所有插件被重新应用到 route 上面，就像第一次请求时那样。

如果在 route 的回调函数里面调用，两种方法都不会影响当前的请求。当然，可以抛出一个 RouteReset 异常，来改变当前的请求。

3.4.5 插件例子：SQLitePlugin

这个插件提供对 sqlite3 数据库的访问，如果 route 的回调函数提供了关键字参数 (默认是"db")，则"db"可做为数据库连接，如果 route 的回调函数没有提供该参数，则忽略该 route。wrapper 不会影响返回值，但是会处理插件相关的异常。Plugin.setup() 方法用于检查应用，查找冲突的插件。

```
import sqlite3
import inspect

class SQLitePlugin(object):
    ''' This plugin passes an sqlite3 database handle to route callbacks
    that accept a `db` keyword argument. If a callback does not expect
    such a parameter, no connection is made. You can override the database
    settings on a per-route basis. '''
```

```
    name = 'sqlite'
    api = 2

    def __init__(self, dbfile=':memory:', autocommit=True, dictrows=True,
                 keyword='db'):
         self.dbfile = dbfile
         self.autocommit = autocommit
         self.dictrows = dictrows
         self.keyword = keyword

    def setup(self, app):
        ''' Make sure that other installed plugins don't affect the same
            keyword argument.'''
        for other in app.plugins:
            if not isinstance(other, SQLitePlugin): continue
            if other.keyword == self.keyword:
                raise PluginError("Found another sqlite plugin with "\
                "conflicting settings (non-unique keyword).")

    def apply(self, callback, context):
        # Override global configuration with route-specific values.
        conf = context.config.get('sqlite') or {}
        dbfile = conf.get('dbfile', self.dbfile)
        autocommit = conf.get('autocommit', self.autocommit)
        dictrows = conf.get('dictrows', self.dictrows)
        keyword = conf.get('keyword', self.keyword)

        # Test if the original callback accepts a 'db' keyword.
        # Ignore it if it does not need a database handle.
        args = inspect.getargspec(context.callback)[0]
        if keyword not in args:
            return callback

        def wrapper(*args, **kwargs):
            # Connect to the database
            db = sqlite3.connect(dbfile)
            # This enables column access by name: row['column_name']
            if dictrows: db.row_factory = sqlite3.Row
            # Add the connection handle as a keyword argument.
            kwargs[keyword] = db

            try:
                rv = callback(*args, **kwargs)
                if autocommit: db.commit()
            except sqlite3.IntegrityError, e:
                db.rollback()
                raise HTTPError(500, "Database Error", e)
            finally:
                db.close()
            return rv

        # Replace the route callback with the wrapped one.
        return wrapper
```

这个插件十分有用，已经和 Bottle 提供的那个版本很类似了 (译者注：=。= 一模一样)。只要 60 行代码，还不赖嘛！下面是一个使用例子。

```
sqlite = SQLitePlugin(dbfile='/tmp/test.db')
bottle.install(sqlite)

@route('/show/:page')
def show(page, db):
```

```
    row = db.execute('SELECT * from pages where name=?', page).fetchone()
    if row:
        return template('showpage', page=row)
    return HTTPError(404, "Page not found")

@route('/static/:fname#.*#')
def static(fname):
    return static_file(fname, root='/some/path')

@route('/admin/set/:db#[a-zA-Z]+#', skip=[sqlite])
def change_dbfile(db):
    sqlite.dbfile = '/tmp/%s.db' % db
    return "Switched DB to %s.db" % db
```

第一个 route 提供了一个"db" 参数，告诉插件它需要一个数据库连接。第二个 route 不需要一个数据库连接，所以会被插件忽略。第三个 route 确实有一个"db" 参数，但显式的禁用了 sqlite 插件，这样，"db" 参数不会被插件修改，还是包含 URL 传过来的那个值。

3.5 可用插件列表

这是一份第三方插件的列表，扩展 Bottle 的核心功能，或集成其它类库。

在插件 查看常见的插件问题 (安装，使用)。如果你计划开发一个新的插件，插件开发指南 也许对你有帮助。

Bottle-Extras Meta package to install the bottle plugin collection.

Bottle-Flash flash plugin for bottle

Bottle-Hotqueue FIFO Queue for Bottle built upon redis

Macaron Macaron is an object-relational mapper (ORM) for SQLite.

Bottle-Memcache Memcache integration for Bottle.

Bottle-MongoDB MongoDB integration for Bottle

Bottle-Redis Redis integration for Bottle.

Bottle-Renderer Renderer plugin for bottle

Bottle-Servefiles A reusable app that serves static files for bottle apps

Bottle-Sqlalchemy SQLAlchemy integration for Bottle.

Bottle-Sqlite SQLite3 database integration for Bottle.

Bottle-Web2pydal Web2py Dal integration for Bottle.

Bottle-Werkzeug Integrates the *werkzeug* library (alternative request and response objects, advanced debugging middleware and more).

这里列出的插件不属于 Bottle 或 Bottle 项目，是第三方开发并维护的。

3.5.1 Bottle-SQLite

SQLite 是一个" 自包含" 的 SQL 数据库引擎，不需要任何其它服务器软件或安装过程。sqlite3 模块是 Python 标准库的一部分，在大部分系统上面已经安装了。在开发依赖数据库的应用原型的时候，它是非常有用的，可在部署的时候再使用 PostgreSQL 或 MySQL。

这个插件让在 Bottle 应用中使用 sqlite 数据库更简单了。一旦安装了，你只要在 route 的回调函数里添加一个"db" 参数 (可更改为其它字符)，就能使用数据库链接了。

安装

下面两个命令任选一个

```
$ pip install bottle-sqlite
$ easy_install bottle-sqlite
```

或从 github 下载最新版本

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/sqlite
$ python setup.py install
```

使用

一旦安装到应用里面，如果 route 的回调函数包含一个名为"db" 的参数，该插件会给该参数传一个 sqlite3.Connection 类的实例。

```
import bottle

app = bottle.Bottle()
plugin = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
app.install(plugin)

@app.route('/show/:item')
def show(item, **db** ):
    row = db.execute('SELECT * from items where name=?', item).fetchone()
    if row:
        return template('showitem', page=row)
    return HTTPError(404, "Page not found")
```

不包含"db" 参数的 route 不会受影响。

可通过下标 (像元组) 来访问 sqlite3.Row 对象，且对命名的大小写敏感。在请求结束后，自动提交事务和关闭连接。如果出现任何错误，自上次提交后的所有更改都会被回滚，以保证数据库的一致性。

配置

有下列可配置的选项:

- **dbfile**: 数据库文件 (默认：存在在内存中)。
- **keyword**: route 中使用数据库连接的参数 (默认：'db')。
- **autocommit**: 是否在请求结束后提交事务 (默认：True)。
- **dictrows**: 是否可像字典那样访问 row 对象 (默认：True)。

你可在 route 里面覆盖这些默认值。

```
@app.route('/cache/:item', sqlite={'dbfile': ':memory:'})
def cache(item, db):
    ...
```

或在同一个应用里面安装 keyword 参数不同的两个插件。

```
app = bottle.Bottle()
test_db = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
cache_db = bottle.ext.sqlite.Plugin(dbfile=':memory:', keyword='cache')
app.install(test_db)
app.install(cache_db)

@app.route('/show/:item')
```

```
def show(item, db):
    ...

@app.route('/cache/:item')
def cache(item, cache):
    ...
```

3.5.2 Bottle-Werkzeug

Werkzeug 是一个强大的 WSGI 类库。它包含一个交互式的调试器和包装好的 request 和 response 对象。

这个插件集成了 `werkzeug.wrappers.Request` 和 `werkzeug.wrappers.Response` 用于取代内置的相应实现，支持 `werkzeug.exceptions` , 用交互式的调试器替换了默认的错误页面。

安装

下面两个命令任选一个

```
$ pip install bottle-werkzeug
$ easy_install bottle-werkzeug
```

或从 github 上下载最新版本

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/werkzeug
$ python setup.py install
```

使用

一旦安装到应用中，该插件将支持 `werkzeug.wrappers.Response` ，所有类型的 `werkzeug.exceptions` 和 thread-local 的 `werkzeug.wrappers.Request` 的实例 (每次请求都更新)。插件它本身还扮演着 werkzeug 模块的角色，所以你无需在应用中导入 werkzeug。下面是一个例子。

```
import bottle

app = bottle.Bottle()
werkzeug = bottle.ext.werkzeug.Plugin()
app.install(werkzeug)

req = werkzueg.request # For the lazy.

@app.route('/hello/:name')
def say_hello(name):
    greet = {'en':'Hello', 'de':'Hallo', 'fr':'Bonjour'}
    language = req.accept_languages.best_match(greet.keys())
    if language:
        return werkzeug.Response('%s %s!' % (greet[language], name))
    else:
        raise werkzeug.exceptions.NotAcceptable()
```

使用调试器

该插件用一个更高级的调试器替换了默认的错误页面。如果你启用了 *evalex* 特性，你可在浏览器中使用一个交互式的终端来检查错误。在你启用该特性之前，请看通过 werkzeug 来调试。

配置

有下列可配置的选项:

- **evalex**: 启用交互式的调试器。要求一个单进程的服务器，且有安全隐患。请看通过 werkzeug 来调试 (默认：False)
- **request_class**: 默认是 `werkzeug.wrappers.Request` 的实例
- **debugger_class**: 默认是 `werkzeug.debug.DebuggedApplication` 的子类，遵守 `bottle.DEBUG` 中的设置

CHAPTER 4

许可证

代码和文件皆使用 MIT 许可证:

```
Copyright (c) 2011, Marcel Hellkamp.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

然而，许可证 不包含 Bottle 的 logo。logo 用作指向 Bottle 主页的连接，或未修改过的类库。如要用于其它用途，请先请求许可。

Python Module Index

b