

# Learning Uliweb

作者：Limodou

**Version 1.0**

**2009年11月**

*本书将带你领略 Uliweb 框架的风采。*

# 内容目录

前言.....	5
我的 Web 经历.....	5
我对 Web 开发的体会.....	7
预期读者.....	7
说明.....	8
第一章 Uliweb 介绍.....	9
目标.....	9
第三方组件.....	9
特点.....	10
MVT 模型.....	10
组织结构.....	10
资源共享的处理方式.....	11
URL 映射.....	12
View 和 Template.....	12
ORM.....	13
I18N.....	14
功能扩展.....	14
命令行工具.....	14
运行环境支持.....	15
其它特点.....	15
发展.....	15
第二章 Uliweb 的安装.....	16
小技巧：.....	17
第三章 Hello, Uliweb.....	18
准备工作.....	18
创建第一个项目.....	18
settings.ini.....	19
创建第一个 App.....	19
启动开发服务器.....	20
views.py 的秘密.....	21
总结.....	23
补充知识.....	23
project 和 App 的缺省文件是从哪里来的.....	23
开发服务器.....	23

第四章 图片显示.....	25
创建 ShowImage.....	25
准备图片.....	25
使用 staticfiles app.....	25
修改 settings.ini.....	26
理解 settings.ini.....	27
添加 show 函数.....	29
添加 show.html.....	29
url_for_static 函数.....	30
运行结果.....	30
staticfiles 的配置.....	31
总结.....	32
补充知识.....	32
更多关于 settings.ini.....	32
第五章 显示文本.....	33
创建 ch5.....	33
创建 ShowText.....	33
修改 views.py.....	33
创建 base.html.....	38
创建 show.html.....	39
运行.....	42
总结.....	43
补充知识.....	43
descriptor.....	43
第六章 调试与日志.....	46
调试工具.....	46
引发异常.....	47
修改 views.py.....	47
访问/debug.....	47
Print 调试.....	50
日志.....	51
404 和 500.....	52
使用 admin.....	56
总结.....	60
补充知识.....	61

App 间的重用和依赖.....	61
缺省模板.....	62
第七章 显示文本的改进.....	64
准备.....	64
yaml_shiny app.....	64
修改 ShowText.....	67
template app.....	69
Dispatch 机制.....	70
template 分析.....	71
formcss 分析.....	76
修改 show.html.....	77
创建 ShowText/config.ini.....	77
执行效果.....	78
总结.....	78

# 前言

Web 开发是一件既有趣，又困难的事情。随着我对 Web 开发的一点点深入，越来越觉得 Web 应用是一个趋势，特别是在日常工作中。我们经常需要工作上的协作，比如：统计信息，交流文档等，而这些工作可能在平时我们会通过各种各样的工具来辅助我们。那么如果你是一位开发爱好者，完全可以考虑将这些功能做成 Web 应用，方便大家的使用，并且可以起到一定的管理功能。本人最精通的语言就是 Python 了，因此这本书就是讲如何使用 Python 进行 Web 开发。在 Python 世界中，已经有许许多多的 Web 框架，如：Zope, Django, TurboGears, Pylons, Web2py, Karrigell 等。那么我用过其中不少，象：Zope, Django, Web2py, Karrigell，当然有些只是研究性质并没有太深入。在学习它们之后，我也有自己的一些想法。

## 我的 Web 经历

很早就开始学 Zope 了，那时还是为了做个人主页，在一个偶然的的机会接触到了它，给我带来了许多新奇的感觉。那时我用的是 Zope 2.0，国内没有提供 Zope 的网站，但是有一个 [freezope.org](http://freezope.org) 的外国站点可以免费提供，于是在上面申请了一个帐号，开始制做 [pyrecord.freezope.org](http://pyrecord.freezope.org) 站点。那时网站的功能很简单，就是写文章，发布，可以更新新闻，提供软件下载。而 Zope 提供在线开发，我使用 ZClass 编写，模板使用 DTML，CSS 之类的也是在上面写的。Zope 提供导入导出功能，可以在本地开发，然后上传。在 Zope 中，使用对象数据库，目录就是一个容器，文件、图片之类的就是容器中的元素。只不过对它们的操作，并不能象一般的 Python 语法一样来处理，要完全使用 Zope 的 API。所以使用 Zope，你需要了解它的 API，不能很简单的使用 Python 来处理。Zope 中可以把一些通用的功能做成一个个的 Product，那时有许多的 Product 可用，如论坛，问题跟踪等工具。于是乎我也开始想做 Product，但是在学习过程中我感觉非常困难，文档不多、步骤复杂，虽然我跟着做出了一个简单的例子，但是后来一直都没有真正做过一个可用的 product。慢慢地我放弃了，后来主要的重点转到的 GUI 上，再往后就是创建了 UliPad 项目，于是有好长时间不再搞 Web 了。

因为在当时，Web 环境基本上都是 PHP 的，Python 的没有几个，再加上没有好的框架，而且我的思路也没有转到 Web 上来，于是 Web 很长时间内不在我的视线中。

后来我又学习了 CherryPy 和 Karrigell。在研究它们的过程中我写了不少的源码的分析文章，那时它们都很简单。CherryPy 在当时有许多都不是自带的，如：模板、数据库。而 Karrigell 则内置了这些功能。于是在一段时间内 Karrigell 我比较喜欢。但是它仍然功能不够完整，比如：用户认证之类的。

直到后来看到 Django 要发布的消息，我开始跟踪 Django 的发展、学习每一个早期的版本、参加邮件列表、提问题、发表看法、提供补丁等。后来为了宣传 Django，还在《程序员》杂志发表了介绍 Django 框架的文章(见 2006 年第 11 期)。为了方便大家入门，还编写了《Django Step by Step》的教程。在那时，我认为 Django 是一个很好的选择。我也使用 Django 开发了不少的项目，并且都是开源的，如 Woodlog, OpenbookPlatform, Shareplate, Zipbook 等等，不过现在基本上不维护了。

随着对 Web 开发的实践越来越深入，我也向 Django 的开发小组提出一些我认为可以改进的地方，不过许多设计上的观点不被接受，这让我感觉很遗憾，于是我开始寻找其它的框架。渐渐地，我远离了 Django。

再后来，我开始学习 Web2Py，开始它的确很吸引我，许多地方让人上手非常快。我依旧向 Web2Py 贡献了一些代码。不过仍然是有一些设计上的观点无法同作者达成共识，我开始想我应该怎么办？我已经学习和研究了许多的框架，也有不少的实践，即然很难找到让我满意的框架，那么我又有能力，为什么不自己去开发一个框架，而是把时间花在同框架的作者去争论之上呢？别的框架好的地方，拿过来用就是了。终于 Uliweb 诞生了！

Uliweb 在我写这本书的时候发布时间还不长，还不是很完善，但是我希望通过写一些教程来让感兴趣的朋友对它有一个深入的了解。未来的走向我无法预测，因为开发开源软件只是我的一个爱好，并不是我的职业，因此投入的精力毕竟有限，而我个人能力也是有限的。因此 Uliweb 不可避免会有问题。再加上国内开源软件的环境也不是很让人满意。我只能尽最大的努力来开发 Uliweb。不过正是因为我是为了兴趣而做，我的压力也相对较小，可以让我充分享受编程的乐趣，可以让我在开发的过程中慢慢地总结与分享。如果你对 Uliweb 感兴趣，希望你也可以和我一起体会 Web 编程的滋味。

# 我对 Web 开发的体会

Web 开发不是一件容易的事，首先它是一种网络访问方式，因此就有浏览器和服务器的存在，也就是 B/S 结构。而 Web 开发就是考虑前端和后端两部分。从逻辑上还可以再分为：浏览器、Web server、应用层和数据库。有时应用层是直接部署到 Web server 中的，但是也有一些是独立于 Web server 的。所以从结构上远比 GUI 应用要复杂得多。

Web 开发涉及到多种技术，从前面的逻辑结构就可以看到，需要很全面的知识。光是前端展示就可能涉及：XML, HTML, CSS, Javascript, Flash 等内容。后端这块你可能要了解一些 CGI(Common Gateway Interface)、Web server 部署的知识、数据库的知识、HTTP 协议的知识等。随着 Ajax, RIA 的发展，你可能还要学习这些新的知识。在许多地方，不同的分层方式会导致不同的分工，比如专做 Ajax 的，CSS 的。但是对于我来说，只有一个人！所以什么东西都要搞。因此 Uliweb 从某种意义上来说，更适合程序员使用。

Web 框架的目的就是简化开发，把一些可积累的内容变成可复用的组件，减少重复开发。因此你会看到一般的框架都包括：URL 映射、模板系统、配置管理、Request 对象、Response 对象、数据库封装等。有些还包括象 Cache、Session 等一些可复用的组件等。因此不同的框架从功能上基本上都差不多，差别只是设计思想及具体的实现上，当然可能还包括具体使用的模块。

## 预期读者

由于本书是讲解 Uliweb 这个使用 Python 开发的 Web 框架，因此需要读者具备一定的 Python 基础，同时具备一定的 Web 开发的知识和经验。本书并不是从 Web 开发入门开始，因此涉及到的一些知识可能需要你自学。并且，由于 Uliweb 是我开发的新框架，许多功能没有经过大规模地验证，因此还是属于研究性质的，可以作为业余爱好或一个用来学习如何构建一个框架的对象。如果你想学习一个实用、适合商业应用的框架，可能我还是会建议你去学习 Django, Pylons 之类的更成熟的框架。不过我的确希望 Uliweb 可以发扬光大，可以用在一些正式场合。

## 说明

本书的源代码可以从 <http://code.google.com/p/uliweb-tests> 中的源码 learning\_uliweb 找到。

# 第一章 Uliweb 介绍

Uliweb 是一个使用 Python 开发的 Web 框架，它有许多的特点，而这些特点一方面来自于其它的框架，另一方面来自于我对 Web 开发的理解。

## 目标

Uliweb 的目标是提供一个易于开发、易于重用的框架。

易于开发就是让开发尽可能地简单，比如：

- 通过约定，减少配置
- 通过注入，减少导入
- 通过命令行工具，方便管理

易于重用是一个非常重要的目标。Web 的重用很难，因为一个功能涉及到的地方太多，因此复用可能要改许多地方，而 Uliweb 的目标就是让这些修改尽可能简单和自动。因此，在 Uliweb 中的许多设计都是针对重用的，这一点随着讲解你会慢慢地体会。

有得必有失，因为有许多自动化的特点，所以它的效率可能不是最高的，而且我的确也没有做过测试。

## 第三方组件

Werkzeug (<http://werkzeug.pocoo.org/>)它是一个 Web 工具的集合，它的功能很全，象 wsgi 的支持、Route、Request、Response、Template、Cache、Session 的支持，不过，它没有数据库的封装，因此你要自己去弄。在它的网站上有不少的例子教你如何搭建一个 Web 框架，因此它是一个很好的起点。它还带有一个 Debug 的 wsgi 中间件，很好用，可以显示出错时的堆栈信息，并且查看局部变量和执行一些代码。

SQLAlchemy (<http://www.sqlalchemy.org/>)提供了非常强大的数据库 ORM 的功能，支持多种数据库。Uliweb 在它的基础之上封装了简单的 ORM，但是这并不是必需的，不过有一些内置功能如果想用，则需要使用内置的 ORM。

pytz (<http://pytz.sourceforge.net/>)提供时区转换的功能。

simplejson (<http://pypi.python.org/pypi/simplejson/>)提供了 Python 数据类型与 Json 格式转换的功能。

上面的一些组件有些已经内置在 Uliweb 中了，减少了下载的要求，有些因为比较大，没有放进去，但是在安装过程中可以自动下载并安装。

## 特点

### *MVT 模型*

即 Model, View, Template，和 Django 的一样。

Model 可以使用 Uliweb 的 ORM，也可以使用其它的数据库。不过对于其它的数据库，初始化的工作要你自己来做。

View 是由函数组成的，没有采用类(Class)的方式组织。

Template 目前是集成在 Uliweb 中的，采用的是从 Web2Py 改造过来的 template 模块，支持象 Django 一样的 block 替换的功能。

### *组织结构*

Uliweb 是采用类似于 Django 的 App 的组织方式。App 是最小的开发单位，而不是运行单位。每个 App 会包含：

- `__init__.py` 表示它是一个包

- settings.ini(可选)配置文件，用于存放配置相关的信息
- config.ini (可选)配置文件，用于存放 app 间的依赖关系
- info.ini(可选)App 描述文件，描述本模块的基本信息，可以被 admin 功能用于识别当前目录是不是一个 Uliweb 的 App
- conf.py(可选)App 配置界面，当前 App 如果可以被 admin 管理，则通过这个文件可以提供简单的管理界面
- commands.py (可选)命令行文件，可以被 Uliweb 命令行调用
- views.py (可选)用来存放 View 相关的代码。
- templates/ (可选)用来存放模板
- static/ (可选)用来存放静态文件
- template\_plugins/ (可选)用于模板中的 use 指令扩展
- locale/ (可选)用于存放 i18n 的翻译文件

整个 Uliweb 项目的结构为：

project/	
apps/	用于存放项目的各个 app
locale/ (可选)	用于存放整个项目相关的翻译文件
app.yaml	GAE 的配置文件
gae_handler.py	GAE 的处理程序
runcgi.fcgi	FastCGI, SCGI, CGI 的执行程序
wsgi_handler.wsgi	wsgi 的处理程序

## 资源共享的处理方式

资源在这里主要是指：配置文件(settings.ini)、模板和静态文件。

从上面的 App 的结构我们可以看到，每个 App 下都可以有自己的 settings.ini、模板目录和静态目录。那么在启动 Uliweb 时，它会自动将所有有效的 App 的这些资源视为一个整体，对于象模板目录和静态目录，它会先查找当前 App 下的目录，如果找不到需要的资源则会去其它的 App 下相应的目录查找。因此一个 App 可以直接引用其它 App 中的资源。对于 settings.ini 这样的配置文件，Uliweb 在启动时会查找每一个 App 中的 settings.ini，如果存在，则进行合并。那么这里有一个顺序：

core/default\_settings.ini、每个 App 下的 settings.ini、apps/settings.ini

Uliweb 会按上面从左到右的顺序进行合并处理。core/default\_settings.ini 定义了一些缺省信息，如：TIME\_ZONE 等。

这种设计目的其实是为了重用及自动化。一般我们在开发一个 App 时，可能包括许多的资源，如：CSS、图片、模板、配置信息。许多框架是不按 App 进行区分的，它们的资源是混在一起的，比如放在一个公共的目录下，通过子目录来区分，造成重用非常困难。为什么？因为除非你有意识进行区分，否则无法从一个目录下分清楚资源与功能之间的关系。当然这种结构可能也不太考虑重用。而象 Django，它支持重用，但是象静态文件，配置信息是不提供重用机制的。而 Uliweb 的机制可以把可重用的部分划分成独立的 App，互相不干扰，但是在使用上又是一个整体，比较好的解决了功能划分与使用的问题。

## URL 映射

Uliweb 使用 werkzeug 自带的 Route 模块进行 URL 的处理，不过为了方便，Uliweb 提供了 **expose** 函数，可以方便地对 View 函数进行修饰，它可以把一个 URL 与 View 函数进行绑定。而且 expose 是可以被单独使用的。如同在 Django 中一样，expose 可以处理的 View 函数可以是函数对象或者是字符串表示方式。Uliweb 还提供了**命令行生成集中的 urls.py 的工具**。一旦有这个文件存在，则只会使用这个文件中定义的 URL，所有在 view 中的 expose 将失效。另外 Uliweb 还提供 **url\_for()** 函数，可以根据 views 函数反向生成 URL。

## View 和 Template

### View:

- 采用函数表达方式，不是类的方式。
- 当返回一个 dict 值时，会自动套用模板。这一点与 Web2Py 相同，但不同于 Django。Django 是需要你显示地指定一个模板，然后手动渲染。而 Uliweb 是自动的。
- 如果返回其它的值，则转为 Response 对象并返回。

- View 函数是在一个环境中运行的，因此在 View 函数中有一些可直接使用的对象，如：request, response, application, settings, env 等。
- Uliweb 提供方法可以注入新的 View 环境变量。

## Template:

- 由 web2py 的模板改进而来。
- Uliweb 的模板可以直接嵌入 Python 的代码，并且通过 `{{pass}}` 来自动处理缩进。支持简单的 Python 代码与 HTML 代码混用。
- 支持模板的包含(include)和扩展(extend)。
- 支持多个 Block 的替换，和 Django 很象。
- Uliweb 提供方法可以让你向模板注入新的环境变量用在模板的执行中。
- 可以将模板转为 Python 代码的功能。
- 可以自定义新的标签。
- 提供转义和不转义的标签。
- 允许设置缺省模板，在找不到对应的模板时自动使用。

## ORM

Django 的 ORM 用起来比较方便，API 比较简单，定义比较简单。但是 Django 的 ORM 并不方便移植，它与自身的其它模块关联太紧密了，所以我决定了在 SQLAlchemy 的基础上自己做。其实 SQLAlchemy 有自己的面向对象的封装，不过不太满足我的要求，比如 API 的调用方式，表的创建方式等。还有一个叫 Elixir 的项目也是基于 SQLAlchemy 的 ORM 的封装，也看过，但是最终还是决定自己做一个，你可以选择不用。

ORM 的封装是参考了 GAE 的 datastore 代码，感觉写得不错，又比较接近 Django 的风格。于是我拿过来做了改进。

Uliweb 的 ORM 有以下特点：

- 支持自动创建表结构。通过设置，可以在使用时自动创建相关的表。
- 配置简单。基本上通过配置数据库连接参数及是否自动建表就可以了。连接参数与 SQLAlchemy 一致。

- 支持多种关系的处理：OneToOne, ManyToOne, ManyToMany, 自引用等。
- 每个 Model 下有一个 table 属性，可以使用 SQLAlchemy 的方法来直接调用。

## I18N

Uliweb 采用标准的 gettext 方式的国际化处理。你可以在程序、模板甚至 settings.ini 中使用 \_() 来定义翻译字符串。Uliweb 提供工具可以从程序中提取翻译字符串。Uliweb 支持多种级别的提取方式，如：App 级，Project 级和核心级别。

## 功能扩展

Uliweb 支持多种方式的扩展：

- 组织级别，通过添加新的 App 进行扩展
- dispatch 扩展。在程序运行中，通过 dispatch 机制可以发出调用信息。而每个信息可能有 0 到若干个 receiver 在等待处理。而 dispatch 发出点事先并不需要提前知道。类似于发布/订阅机制。不过 Uliweb 中的 dispatch 是可以排序的。基本功能是从我开发的 UliPad 中搬过来的。若干个 receiver 将组成一个串，根据排序按顺序将被执行。Uliweb 提供几种调用方式，dispatch.call() 将全部执行，无返回值。dispatch.get() 将顺序执行，但是一旦某个 receiver 返回非 None 值时，执行结束，将返回这个非 None 值。
- Middleware 扩展。这个与 Django 中的一样。中间件主要是用来对 view 进行底层的加工，它可以在调用 View 函数之前，之后及抛出异常时被调用。
- wsgi middleware 扩展。前面的是 View 级别的中间件扩展。而这个是 wsgi application 级别的扩展。Uliweb 的核心就是一个 wsgi 的 application。

## 命令行工具

Uliweb 提供了不少命令行工具，可以方便开发和使用。比如通过命令行可以方便地创建 project，创建 App，提供静态文件，提供 I18N 的翻译字符串，启动开发服务器等等。

Uliweb 还支持每个 app 下定义自己的命令行工具的功能。

## 运行环境支持

Uliweb 可以运行在 GAE、wsgi、FastCGI、scgi、cgi 的环境下。对于 GAE，目前 Uliweb 的 ORM 是无法使用的。因为 SQLAlchemy 不支持 GAE。

## 其它特点

- 支持 TIME\_ZONE 的设置
- 支持对静态文件的 HTTP\_IF\_MODIFIED\_SINCE 和 ETAG 的处理功能
- 提供 Debug 调试功能
- 当修改程序后，开发服务器自动重启
- 自带一些可重用的 App，如：静态文件处理、用户认证、Cache、Session、Template、Uliweb ORM 等

## 发展

- 2008.5.15 项目创建，起名为 Kuaiboo
- 2008.5.26 项目改名为 Uliweb，迁移到 <http://code.google.com/p/uliweb>
- 2009.10.2 发布 Uliweb 0.0.1a1 版本，正式与大家见面。

可以看到 Uliweb 还很年轻，并且由于只有一个人，所以发展也相对慢一些，问题和不足也很多，许多功能还不完善，还需要不停地改进。但是我会象 Ulipad 一样尽我最大的努力发展下去。首先让自己满意！

## 第二章 Uliweb 的安装

最简单的方式是通过 easy\_install Uliweb 来进行安装。如果你想跟踪最新的特性，建议使用 svn 下载源码后进行安装。

在安装之前请先确认你的环境。建议的 Python 环境是 2.5 和 2.6。

为什么不是 3.0，因为大部分的模块都没有转过去呢，所以这个别想了。2.4 的问题不大，但是有些东西要自己装了。比如 setuptools 工具，sqlite3(如果你要使用 sqlite 数据库的话)。现在 2.5 是最常用的 Python 版本。Jython? IronPython? 对不起，都没试过，如果你试过了，请告诉我。

首先使用 svn 工具下载 Uliweb 源码(<http://code.google.com/p/uliweb>)。

```
svn checkout http://uliweb.googlecode.com/svn/trunk/ uliweb
```

然后请检查你的 setuptools 的版本是否在 0.6c9+，因为 0.6c8 版本有 bug，所以要升级。可以从 <http://pypi.python.org/pypi/setuptools> 找到。

切换到 Uliweb 的源码目录下。这里有两种做法：

- 通过: `python setup.py install`  
来安装。这样就安装到了 `Python/Lib/site-packages` 下了。
- 通过：`python setup.py develop`  
来安装。这时并不会将 Uliweb 装到 `Python/Lib/site-packages` 下，只是建了一个链接到源码目录。这个好处是，当 Uliweb 发生变化，可以方便地使用 svn 进行同步，然后不需要再执行 `setup.py install`。

在安装时，安装程序会安装几个第三方的组件：SQLAlchemy, docutils, Pygments, pytz。因此需要你的网络可以联通，当然如果你已经安装过了，它只会检查一下并不一定安装。

其实 docutils 和 Pygments 并不是必须的。只不过 Uliweb 的源码中的 apps 目录是 <http://uliwebproject.appspot.com> 的全部源码，而你要运行的话，它需要，所以就放在 setup.py 中了，以后可以考虑分离。

在“Uliweb 介绍”一章中你已经看到了许多模块，它们有一些已经被内置到了 Uliweb 中了，在 Uliweb/lib 目录下。因此这些是不需要安装的。特别是 werkzeug 还是一个我修改过的版本，所以不能被替换。

一旦安装成功，将会在 Python/Scripts 下创建一个名为 uliweb 的工具，可以用它来执行命令行。

在后面的开发过程中，我们会经常使用命令行，因此在进入开发之前请先检查你的环境：是否把 Python 和 Python/Script 目录加入到系统的 PATH 环境变量中了？如果没有请加入。

### 小技巧：

在 Windows 下我不知道你是怎么进入命令行的？其实有一个工具叫：Open Command Window Here，它可以在资源管理器中增加一个上下文的菜单。先选中一个目录，然后用它就可以直接打开一个命令行窗口，很方便。这是微软开发的一个小工具。不过其实内容就是向注册表添加一些信息，你自己加都是可以的。在网上搜一下吧，很容易找到

## 第三章 Hello, Uliweb

许多教程都是从 Hello, XXX 开始的。有这么一个网站(<http://www.roesler-ac.de/wolfram/hello.htm>)收集了各种语言写的 Hello, World 程序，我看了看，竟然已经包括了最新的 Python 3000 的程序，更新还真是及时啊。好，下面我带领大家去看一看如何使用 Uliweb 创建一个 Hello 程序。

### 准备工作

为了开始我们的工作，首先要保证在前面你已经正确安装了 Uliweb，并且在命令行下可使用 uliweb 工具。如何验证？让我们先进入命令行，然后在命令行下输入：

```
uliweb
```

如果可以执行成功，你会看到一个帮助信息，列出了当前可用的命令。如果没有，请检查你的安装和环境配置是否正确。

### 创建第一个项目

在 Uliweb 中，你的工作是按项目进行组织的，每个项目由若干个 App 构成。因此第一步就是创建项目。首先让我们找一个合适的根目录(这个随便你)，然后进入命令行，确保当前目录在这个根目录下。

然后执行：

```
uliweb makeproject ch3
```

makeproject 是要执行的命令，后面需要一个项目的名字。因为现在是第三章，所以我起为 ch3。注意，这个 ch3 将用来创建一个子目录，用于存放你的文件和代码。如果当前目录有重名的项目目录，它会提示你是否要覆盖。如果你忘记输入项目名称，则它会提示你输入一个。

当执行成功，会看到当前目录下有一个 ch3 的子目录。这样项目就创建好了。

下面进入 ch3 目录。可以看到有以下内容：

```
|-- app.yaml
|-- apps/
|   |-- settings.ini
|-- gae_handler.py
|-- runcgi.py
|-- wsgi_handler.wsgi
```

其中 apps 是用来存放 App 的，settings.ini 是全局性的配置文件，其它的是与部署相关的东西。

如果只是开发，我们只要关注 apps 和 settings.ini 就可以了。

## settings.ini

settings.ini 是一个很重要的东西，它是用来存放配置信息的。还记得第一章中关于组织结构的内容吗？每个 App 都可以有自己的 settings.ini。而在 apps 下的 settings.ini 是整个项目共用的，它将与每个 App 中的 settings.ini 合并成一个整体。这里我们先只是了解一下就好了，后面的章节会有更多关于 settings.ini 的说明。

## 创建第一个 App

前面应该已经进入了 ch3 目录，在命令行下输入：

```
uliweb makeapp Hello
```

它会象 makeproject 一样，检查当前目录是否有重名的 App，如果有会提示你。如果没有则创建对应的 App 目录。如果没有提供 App 名字，则会提示你输入一个。

让我们进入 Hello 子目录看一下有什么？

```
|-- __init__.py
|-- conf.py
```

```
|-- info.ini
|-- static/
|   |-- readme.txt
|-- templates/
|   |-- readme.txt
|-- views.py
```

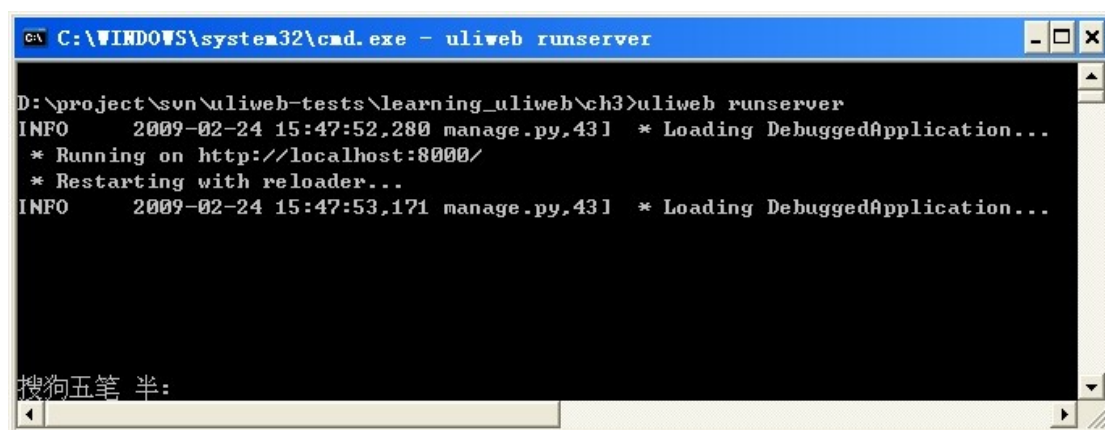
东西不多，基本上都是空的。static 将用来存放静态文件，templates 将用来存放模板，views.py 是用来放 view 函数的，而\_\_init\_\_.py 说明这是一个 Python 的包。现在可以告诉你，只有 views.py 中有一些东西。不过在了解它之前，我们已经可以运行了。而 info.ini 和 conf.py 是用来在 admin 中进行 App 配置的，目前我们不需要了解。

## 启动开发服务器

重新回到 ch3 目录，在命令行输入：

```
uliweb runserver
```

这个 runserver 和 Django 是一样的。一旦启动成功，你会看到一些提示信息，下面是一个图示：



```
C:\WINDOWS\system32\cmd.exe - uliweb runserver

D:\project\svn\uliweb-tests\learning_uliweb\ch3>uliweb runserver
INFO      2009-02-24 15:47:52,280 manage.py,431 * Loading DebuggedApplication...
* Running on http://localhost:8000/
* Restarting with reloader...
INFO      2009-02-24 15:47:53,171 manage.py,431 * Loading DebuggedApplication...

搜狗五笔 半:
```

打开浏览器，输入 <http://localhost:8000> 就可以看到结果了。你会看到在浏览器中显示出“Hello, Uliweb”的信息。

我们没有输入一行代码，但是通过几步简单的命令已经可以将你的 project 跑起来了。是不是很简单。不过，说句实话，入门可能总是简单的，随着我们越学越深，你要学的东西

也越来越多，你就会体会到 Web 开发真不是件容易的事。不过我希望通过我的努力可以把困难的事情简单化，因此我会更多的强调重用的重要性。

## views.py 的秘密

为什么不写代码就可以看到结果，答案就在 views.py 中，让我们打开看一下：

```
1 #coding=utf-8
2 from uliweb import expose
3
4 @expose('/')
5 def index():
6
7     return '<h1>Hello, Uliweb</h1>'
```

很简单，让我们一行行来说。

第 1 行声明 views.py 的编码是 utf-8，建议都使用这个编码，包括模板文件，数据库，这样会减少编码转换出错的几率。

第 2 行是从 uliweb.core.SimpleFrame 中导入 expose。SimpleFrame 是一个非常重要的模块，它定义了 Uliweb 的核心 wsgi 类 Dispatcher。同时它还定义了一些常用的方法。expose 就是一个用来定义 URL 的方法。

第 4 行声明了一个 URL，可以看到是 '/'，它就是与 http://localhost:8000/相匹配的 URL。@expose 是一种 decorator 的写法，它类似于：

```
index = expose(index)
```

它接受一个函数作为参数，然后返回一个新的函数，然后这个新函数仍然命名为 index。expose 其实还可以接受参数的，但是我们常用的就是后面跟一个 URL。在后面我们还会学到更多关于 expose 的用法。

第 5-6 行定义了 `index()` 函数。它就是用于与 `/` 对应的处理函数。一般来说一个 URL 请求可以对应一个函数，当然多个 URL 请求也可以对应一个函数。当存在多个 URL 对应一个处理函数时，可以写多个 `@expose`，比如：

```
@expose('/index')
@expose('/')
def index():
    return '<h1>Hello, Uliweb</h1>'
```

这个一个 `index()` 就对应两个 URL 了。

当然 `view` 函数的名字可以随便起，不过 Uliweb 有一些保留的名字，因此在定义时一旦与这些名字发生冲突，Uliweb 会抛出异常来。

如果你感兴趣可以看 `uliweb/core/SimpleFrame.py` 中的 `reversed_keys` 的定义，它是一个 `list`，目前是：

```
reversed_keys = ['settings', 'redirect', 'application',
                 'request', 'response', 'error']
```

这里的 `view` 函数与 Web2Py 的 `view` 很象，为什么呢？因为你并不需要定义象 Django 那样的 `request` 参数。而这个参数包括上面 `reversed_keys` 中定义的其实都是对象名，都是可以在 `view` 中直接使用的。后面我们就会看到。

一个 `view` 函数的返回值与后续的处理有很大的关系。你可以在 `view` 中返回多种类型的值：

- `dict` 将自动查找与 `view` 函数匹配的模板。上面的例子中，`view` 函数是 `index`，因此如果你返回一个 `dict`，则 Uliweb 会去所有有效的 App 中的 `templates` 中查找名为 `index.html` 的文件。找到后使用这个文件及返回的 `dict` 值进行模板的处理。
- 字符串将其封装为 `Response` 对象返回给前端。
- `Response` 对象 将不作处理直接返回给前端。
- 其它的对象则首先调用 `str()` 将对象转为字符串，然后封装为 `Response` 返回给前端。

因此，如果你有模板想要处理：

- 在 view 中返回一个 dict 对象
- 在 App 的 templates 中定义一个与 view 函数名相同的模板文件，只不过文件后缀应该是.html。
- 在 view 函数中，可以通过 response.template='templatefile'来指定其它的模板文件。这样就不会自动查找与 view 函数名相匹配的模板文件了。这一点是从 Web2Py 学来的。

## 总结

下面简单总结一下我们学到的东西：

- 如何创建一个 project
- 如何创建一个 App
- 如何启动开发服务器
- 如何进行 URL 的简单映射
- view 函数的返回与模板的映射

## 补充知识

*project 和 App 的缺省文件是从哪里来的*

在执行 makeproject 和 makeapp 时，会自动在当前目录下创建目录和文件。那么它们是从哪里来的？它们在 uliweb/template\_files 下，有 App 和 project 两个目录。一个是用来生成 App 的结构，一个是生成 project 的结构。

### *开发服务器*

在启动开发服务器之后，我们可以看到访问的日志输出、程序中的 print 输出(可以作为一种调试手段)、异常信息。如果你想终止服务器的运行，按 Ctrl+C 就可以了。

Uliweb 的开发服务器可以在你修改了文件之后自动重启，因此你只要刷新浏览器就可以看到新的结果了。

启动开发服务器还可以带有一些参数，这些信息可以参考 [uliweb 工具的帮助信息](#)。

## 第四章 图片显示

本章的目标是学习如何在页面上显示一张图片。我们将使用到模板和静态文件的处理，及内置 App 的使用。

### 创建 ShowImage

让我们继续在第三章的目录(ch3)下工作。在它的下面创建一个 ShowImage 的 App，如：

```
uliweb makeapp ShowImage
```

### 准备图片

我找到 Uliweb 的一张 logo.png，让我们把它放在 ShowImage/static 下面。

### 使用 staticfiles app

因为这是在开发环境下，因此你可以使用 Uliweb 自带的静态文件处理模块。在生产环境下，你可能会使用 web server 来处理静态文件，那么你需要通过 web server 的 rewrite 功能去处理某个 URL 开始的所有请求，比如/static/。在 Uliweb 中提供了一个命令，extractstatic，它可以将所有 App 的 static 目录下的子目录和文件导出到指定的目录下。这样就可以方便地使用 web server 的 rewrite 机制了。不过对于简单的项目，使用 Uliweb 的静态文件服务应该足够了，它有以下的特点：

- 它是一个 wsgi middleware，不是一个 view 级别的 middleware
- 它支持 HTTP\_IF\_MODIFIED\_SINCE 和 ETAG，因此当浏览器发出这样的请求时，如果文件没有发生改变，Uliweb 会返回 304，这样就不再重新下载文件，而使用本地的文件。这样可以提高访问的效率。

因此你可以看到，Uliweb 自带的静态文件服务功能还是可以的。

## 修改 settings.ini

使用它很简单。

打开 apps/settings.ini，我们可以看到有如下几行：

```
[GLOBAL]
DEBUG = True
#INSTALLED_APPS = [
#     'uliweb.contrib.staticfiles',
#     ]
```

将上面的注释删除即可。但同时要将修改为：

```
[GLOBAL]
DEBUG = True
INSTALLED_APPS = [
    'Hello',
    'ShowImage',
    'uliweb.contrib.staticfiles',
    ]
```

可以看到我们向 INSTALLED\_APPS(这里的选项名是从 Django 中学来的)中添加了'Hello'(这是我们创建的第一个 app)，'ShowImage'(这是我们创建的用来显示图片的 app)，'uliweb.contrib.staticfiles'(这是将用来支持静态文件的 App)。

INSTALLED\_APPS 是用来记录有效的 App 的。这里 Uliweb 有一个特性：当 INSTALLED\_APPS 为空时，Uliweb 会自动从 apps 目录下将所有子目录视为 App，在启动时自动添加到有效 App 的列表中去。但一旦用户设置了这个选项，则不再执行自动添加。因此，一旦你向其中添加了 App，则要注意添加完全。

那么这里添加的只是简单的字符串吗？'uliweb.contrib.staticfiles'又是何解呢？这里其实添加的是可以直接导入的模块名，但是以字符串来表示的。因此是有可能使用\_\_import\_\_来通过字符串来导入的。

那么它们是如何被导入的呢？对于'Hello'，'ShowImage'这样的 App，因为它们是在 apps 目录下。而 Uliweb 在启动时会自动将 apps 目录加到 sys.path 中，这样 Uliweb 就可

以直接从 apps 下导入模块了。所以你可以直接使用目录名作为模块名，不需要加什么前缀。因此为了防止与 Python 的标准库重名，建议以首字母大写的字符串来建立 App 目录。而对于'uliweb.contrib.staticfiles'，则是从 Uliweb 目录下导入的。因为我们在安装步骤中执行了安装操作，所以可以通过 uliweb 来导入它内部的包。在 Uliweb 下的 contrib 中，已经存放了写好的一些可以作为公共使用的 App，你是可以直接使用的。从这一点我们也可以想到，如果你有其它的 App 不在 apps 或不在 uliweb.contrib 下，只要可以正常导入，都可以加到 INSTALLED\_APPS 中去。

前面我们说了很多，其实操作很简单就是添加几个 App 配置项。在后面我还会向大家介绍不需要手工修改 settings.ini 的方法，利用 admin 这个 App 来实现，可以让我们的配置更简单一些。

## 理解 settings.ini

settings.ini 是一个非常重要的文件，在第一章的 Uliweb 介绍中我们已经介绍过了 settings.ini 的组织与处理(如果想不起来了，可以回去看一下)。简单说就是：每个 App 都可以有自己的 settings.ini，apps 下的 settings.ini 是全局共享的。这些 settings.ini 将被合并成一个统一的 settings 对象，可以在 view 函数和 template 中被直接使用。

settings.ini 的格式是 ini 的风格，即：由 section 构成，每个 section 有一个名称和若干 key=value 形式的选项。从上面的例子就可以看出：

```
[GLOBAL]
DEBUG = True
INSTALLED_APPS = [
    'Hello',
    'ShowImage',
    'uliweb.contrib.staticfiles',
]
```

section 的名称是 GLOBAL，DEBUG 和 INSTALLED\_APPS 都是 key，其后是它们的值。Uliweb 使用了 Python 的表达式语法，因此它的值就是标准的 Python 数据类型，可以是：int, float, bool, string, unicode, list, tuple, dict 等。甚至如果你第一行写上：

```
#coding=utf-8
```

就象一般的 Python 程序一样表明这个 ini 文件所使用的编码，Uliweb 会自动进行转换，特别是对 u'中文'这种方式定义的 unicode 字符串。

在 settings.ini 中，section 和 key 可以是一般的字符串，大小写不限，这一点与 Django 不同。在 Django 中只能使用大写的 key。

以下是正确的 ini 的写法：

```
#coding=utf-8
[a test]
key1 = '123'
a key = 3
```

在 view 函数中，正如第一章 View 和 Template 中所介绍的，可以直接使用一些对象，如：request, response, application, settings。那么这里的 settings 就是合并后的 settings 对象。使用 settings 的语法是：

```
settings.section.key
settings['section'].key
settings['section']['key']
```

这几种方式都是可以的，也就是说，你可以将 settings 看成二维结构的对象，通过 '.' 操作来获得属性，也可以将其看成是 dict 对象，通过 [] 来获得属性，也可以混用。但是如果 section 或 key 不是一个标识符，或者是 Python 的保留字，则一定要使用 [] 的方式来引用，否则会报错。

当合并多个 ini 到一个 settings 对象中时，如果存在同名的 section.key，则新的将替换旧的。由于 apps/settings.ini 是最后被处理的，因此定义在这个文件中的选项，一旦与某个 App 的选项发生冲突，则以 apps/settings.ini 为准。但 settings.ini 对于一些特殊的值有特殊的处理。**当值为 list 和 dict 这种可变数据结构时，不会进行简单的替换，而是合并。**对于 list 就是将新的 list 扩展到旧的 list 中，但是去掉了重复项，并且不保证顺序。对于 dict 就是将新的 dict 更新到旧的 dict 中。因此这种机制要特别注意，在后面我们会用到。

section 行为就象一个真正的 dict 对象，它有着 dict 常见的方法，如：keys(), items() 等。

可以在 section 和 key 前面添加注释，注释就是第一个字符为'#'的行。

Uliweb 在处理 settings.ini 时会保证 section、key、和注释的顺序。

## 添加 show 函数

下面让我们开始添加 view 函数。打开 ch3/apps/ShowImage/views.py，添加下面的代码：

```
@expose ('/show')
def show():
    return {}
```

这个 show()函数就很简单了。它映射到'/show'上，并且最后返回了一个空 dict 对象。那么说明，我们将要使用模板了。

## 添加 show.html

在 ch3/apps/ShowImage/templates 中创建 show.html，加入如下内容：

```
<head>
  <title>Hello World</title>
</head>

```

这是一个简单的模板，没有太多的内容。其中我定义了 HTML 页面的编码，就是那么 meta 标签所定义的。建议使用 utf-8 编码。

在 body 中是一个 img 标签，它的 src 属性就是我们要显示的图片的 URL。这里就是一行 Uliweb 模板特有的标签。在 Uliweb 中的模板标签都是通过{{}}来处理的。不象 django 有变量与代码的区分。在 Uliweb 中是不区分的。{{=expression}}标签用于输出一个表达式，它可以是一个变量，可以是一个函数的返回值，可以是一个表达式。什么？函数？不错，除了一些特殊的标签，在模板中使用的都是标准的 Python 代码，所以可以直接在模板中使用 Python 代码，象 import 之类的都没有问题。

`{{=expression}}`用于输出转义后的字符串，这里转义是表示将<>之类的字符转为&lt;和&gt;之类的 HTML 实体。而`{{<<expression}}`用于输出不转义的字符串，所以，如果你要输出原始的 HTML，你应该使用这个标签。

关于模板先了解这么多，其它的在后面的学习中再了解。

## url\_for\_static 函数

那么 `url_for_static('logo.jpg')` 从字面上理解，它是用来输出一个静态文件的。它接受一个文件名，它会从有效的 App 的 static 中查找这个 logo.jpg 文件，然后输出。因此，这个 logo.jpg 并不一定要在当前的 App 的 static 下面，可以在其它的 App 下面。只不过，Uliweb 在查找时会先在当前的 App 下查找，找不到再去其它的 App 下查找。

其实这个 `url_for_static()` 并不是模板模块本身提供的，它是由 `staticfiles` 这个 App 提供的。它是怎么做的？现在不需要了解，以后再介绍。你只要知道，在你使用了 `staticfiles` 这个 App 之后，在模板中你就可以直接使用 `url_for_static()` 这个函数了。

## 运行结果

下图是我的运行结果。



## staticfiles 的配置

那么 `url_for_static()` 生成的结果是什么呢？简单的方法是查看生成的页面的源码。可以看到是 `/static/logo.jpg`。在生的 URL 前自动添加了 `/static/`。那么你可能想知道，这个前缀是否可以修改呢？答案是：可以的。如果你打开 `uliweb/contrib/staticfiles` 目录，你可以看到它下面有一个 `settings.ini` 文件。打开看一下：

```
[GLOBAL]
WSGI_MIDDLEWARES = ['wsgi_middleware_staticfiles']
[wsgi_middleware_staticfiles]
CLASS =
'uliweb.contrib.staticfiles.wsgi_staticfiles.StaticFilesMiddlewa
re'
STATIC_URL = '/static/'
[STATICFILES]
STATIC_FOLDER = ''
```

其它的不用关心，我们只要看 `STATIC_URL` 就好了，它定义了静态 URL 的前缀。可以看到就是 `/static/`。那么只要修改它就可以了。下面让我们试一试。不过你并不需要在这个 `settings.ini` 中修改，因为它是 Uliweb 自带的，并不适合在这里修改。你应该在 `apps/settings.ini` 中进行修改。如下修改：

```
[GLOBAL]
DEBUG = True
INSTALLED_APPS = [
    'Hello',
    'ShowImage',
    'uliweb.contrib.staticfiles',
]
[wsgi_middleware_staticfiles]
STATIC_URL = '/media/'
```

好，让我们刷新一下浏览器看一下，结果看不出什么来。再让我们看一下源码，结果原来的 `/static/logo.jpg` 变成了 `/media/logo.jpg`。

从上面的修改我们可以看出，`staticfiles` 是有配置的，它有自己的 `settings.ini` 文件。以后我们介绍的其它的 App 也是类似的。我们可以将 App 下的 `settings.ini` 认为是缺省的配置，一旦我们想修改它，一种方式是修改这个 App 下的 `settings.ini`，但是这种方式会破坏原来的 App 的内容，不便于复用；因此我推荐使用第二种方式，在 `apps/settings.ini` 下修改。并且我们只需要修改必要的信息，不需要覆盖的可以不管它，Uliweb 会自动将 `settings.ini` 进行合并。

## 总结

其实这一章我们讲了许多东西，不过有些东西，如模板讲得并不深入，但是操作并不是很多，看一下我们做了些什么：

1. 创建了一个 ShowImage 的 App。
2. 准备了一个图片到 ShowImage/static 目录下。
3. 在 settings.ini 中的 INSTALLED\_APPS 加入了对 uliweb.contrib.staticfiles 的使用，同时将 'Hello', 'ShowImage' 也加进去了。
4. 在 views.py 中添加了 show 函数。
5. 在 templates 目录下添加了 show.html 模板。在模板中使用了 url\_for\_static 函数。
6. 测试

## 补充知识

### *更多关于 settings.ini*

在前面你看到 App 可以有自已的 settings.ini，并且是采用的文本 ini 的格式，可能会问为什么这样设计？答案是：为了重用和自动处理。在 Django 中你做不到，甚至 django 中不提供 static 目录的支持。那么这样的结果是：所有的 App 的配置信息都放在了 settings.py 中，所有 App 相关的静态文件被放在了一起。对于单个项目是没有关系，但是一旦你想把其中的某个 App 拿出来复用就会非常麻烦，你可能很难找到哪些是与这个 App 相关的资源。而且一个 App 的功能是多方面的，需要的资源也是多样的，配置信息也是其中一种，在 Django 中把配置信息从 settings.py 中找出来也是麻烦的。因此，这就是 Uliweb 的设计。使用 Python 文件作为配置文件是方便人们手工修改，而采用文本的 ini 格式，不仅方便人们手工修改，还方便程序自动修改。

所以在你设计一个 App 时，如果有一些配置信息，建议你把它们都放到 App 下的 settings.ini 中，作为缺省的配置信息。一方面让人方便了解有哪些配置信息，另一方面是当用户没有提供其它值时，可以有缺省值。

# 第五章 显示文本

我们已经知道了如何显示图片，下面我将向大家演示如何录入一段文本，并转为 HTML 代码。

## 创建 ch5

首先在工作根目录下创建一个新的项目：

```
uliweb makeproject ch5
```

## 创建 ShowText

然后在 ch5 下创建新的 ShowText app：

```
uliweb makeapp ShowText
```

## 修改 views.py

修改后的代码如下：

```
1     #coding=utf-8
2     from uliweb import GET, POST
3     from uliweb.form import *
4
5     class F(Form):
6         content = TextField(required=True)
7
8     @GET('/')
9     def show_get():
10        response.template = 'show.html'
11        f = F()
12        return {'form':f}
13
14    @POST('/')
15    def show_post():
16        response.template = 'show.html'
17        f = F()
```

```
18     if f.validate(request.POST):
19         return {'form':f, 'content':f.content.data}
20     else:
21         return {'form':f}
```

让我来一点点解释。

**第 2 行** 我们导入了 GET 和 POST，它们与 expose 的作用一样，是用来绑定 URL 与 view 函数的。不过一个对应于 GET 方法，另一个对应于 POST。如果只使用 expose 的话，你要自己来区分是使用 GET 还是 POST，比如：

```
if request.method == 'GET':
```

而通过 GET 或 POST 就限定了某种方法，因此不再需要上面的判断。因此使用哪种形式看你的选择了。

**第 3 行** 这里我们导入了 uliweb 的 form 库。我们将使用它创建 Form 对象，它可以生成 HTML 代码，对上传的数据进行校验，显示错误信息。它只是一般的库，所以在处理上没有什么特殊的与 Uliweb 绑定的地方。如果你愿意可以考虑使用其它的 Form 库。

**第 5-6 行** 这里定义了一个 Form 类，它只有一个字段 content，它将用来输入一段文本。我给它设定了 required=True 的参数，这个意思就是它不能为空。

这个 Form 类的定义和其它许多 Form 库是差不多的，如 Django。可以直接在 Form 类中定义字段属性，常用的属性有：

- StringField 用于接收字符串，显示为单行输入
- TextField 也用于接收字符串，显示为多行输入
- IntField 用于接收整数
- SelectField 用于定义单选控件，是传统的下拉控件
- RadioSelectField 用于定义 Radio 方式的单选控件
- FileField 定义文件上传字段
- ImageField 定义图片上传字段
- DateField 日期字段

- TimeField 时间字段

每个字段可以有不同的参数，但是它们有一些共同的参数，比如：

- label 每个字段的提示信息，可以生成为<label>标签。
- default 缺省值，当浏览器没有上送时，使用这个值。不同的字段缺省值不同。
- required 是否必输项，缺省为 False。如果为 True 时，前端没有上送则会生成出错信息。
- validators 是检查器列表。检查器是用来校验上传的数据是否满足要求。注意，每个字段有着对应的 Python 数据类型，而浏览器上送的都是字符串，因此在校验前会先进行类型的转换，转换成功后才会使用校验器来检查。所以校验器只需要处理转换后的数据类型就可以了。校验器可以是很简单的函数，在后面我们遇到后会进行详细说明。
- name 每个字段在类中定义时会对应一个属性名，缺省情况下这个属性名就是字段的 name 值。而这个 name 值在生成对应的 HTML 代码时，就是相应的 name 属性的值。但是你也可以传入一个有效的 name 值，这样在生成 HTML 代码时，name 属性的值就是你指定的值，而不是对应的类属性名称了。
- html\_attrs 用于传入这个字段想要输出的 HTML 相关的属性，可以在生成的 HTML 标签上显示这些属性。可以用来指定 class 属性。
- build 指定 HTML 代码的生成器。每个字段都有缺省的 HTML 代码的生成器，通过指定这个参数可以换成其它的。
- datatype 指定数据的转换类型，主要是将前端上送的数据类型转为字段要求的数据类型。每个字段都有缺省的数据类型，通过指定这个参数可以换成其它的。
- multiple 如果为 True，则表示一个字段对应多个值，这样将返回一个数组，而不是单个值。这个可以用在多个字段名一样，并且希望拼成一个 list 的情况下。
- help\_string 帮助信息。
- idtype HTML 代码中的 id 属性生成类型。如果为'name'，则生成的 id 为'field\_'+字段.name 的值。如果为其它非空的值，表示使用唯一的 id 号，这个号是由 Form 模块来提供的，生成的格式为'field\_'+数字。如果为 None，则不输出 id 属性。

Form 的内容很多，先介绍到这里。

**第 8 行** 用来绑定 URL 与 view 函数。它与前面介绍的 expose 功能是一样的，只不过，它会限定访问的 HTTP 协议中的方法，只有 GET 方法才可以生效。而 expose 是对所有方法

生效。前面已经作了简单的介绍。那么你不使用 GET 而使用 expose 也可以使用相同的效果，比如可以看一下 GET 的实现源码：

```
def GET(rule, **kw):
    kw['methods'] = ['GET']
    return expose(rule, **kw)
```

可以看出，expose 本身就支持一个 methods 的参数，它需要一个方法名的列表。GET 不过是 expose 的一个简化而已。后面的 POST 也是如此。

**第 10 行** 这里很有意思。它指定了 response.template 为 'show.html'。这里演示了如何指定要使用的模板。因此 show\_get() 将不会查找名为 'show\_get.html' 的模板文件，而是要查找 'show.html' 的模板文件。

**第 11 行** 创建 Form 对象。Form 类也支持许多的参数。不过这里先只介绍一个：title。指定它，可以显示出一个标题。在使用 Form 时，你可以使用它直接输出 HTML 代码，不用自己来一点点也，这对于简单情况是足够了。在自动生成的 HTML 中，我是使用了 fieldset 标签，它是可以有标题的。如果你不设置 title，则不显示标题。

**第 12 行** 将输出一个 dict 对象，用在模板中。

从前面的 Form 定义可以看出，我们没有给 content 设置初始值。那么我们经常会遇到的一个问题就是：在运行时，可能需要根据环境的变化，如：用户的切换，动成给 Form 设置不同的初始值。应该怎么做呢？在 Uliweb 中有两种做法：

在 form 的初始化函数中传入 data 参数，这个参数是一个 dict 对象，每个 key 对应 form 的字段属性，value 则为与对应的字段属性数据类型一致的 Python 值。如上面的例子，如果我们想要给 content 设置一个初始值，可以在创建 Form 实例时：

```
def show_get():
    response.template = 'show.html'
    data = {'content': '<empty>'}
    f = F(data=data)
    return {'form': f}
```

另一种方法，直接给 Form 的字段属性赋值，如：

```
def show_get():
    response.template = 'show.html'
    f = F()
    f.content.data = '<empty>'
    return {'form':f}
```

这两种做法都是可以的。要注意，使用字段属性时，`FormInst.Field.data` 才可以得到字段属性的值。直接 `FormInst.Field` 得到的是一个叫 `FieldProxy` 的对象。而这个 `FieldProxy` 对象有许多属性，如：

- `label` 用于输出与 `field` 对应的 `<label>` 标签
- `help_string` 用于输出与 `field` 对应的帮助信息
- `error` 用于输出与 `field` 对应的出错信息（前提是你前端提交的数据进行了校验并且有错误的时候）
- `html` 用于输出 `field` 的 HTML 代码
- `data` 它是一个可读/可写的字段，可以输出对应的字段值，也可以写入
- `__str__` 会调用 `html` 输出，因此可以在模板中使用 `{%=FormInst.field%}` 来输出一个字段的 HTML 代码。

请注意，我上面使用了 `FormInst`，这表明它是一个 `Form` 的实例，而不是 `Form` 类。`Form` 实例和 `Form` 类在使用 `field` 时是不同的，一个得到的是 `FieldProxy` 对象，而一个是真正的 `Field` 实例。为什么会有这种区别？因为一个 `Form` 类它只是用来定义的，不会与具体的数据相结合，而 `Form` 实例是与具体的实例相结合。所以不同的 `Form` 实例，它们的数据是可以不同的，但是 `Form` 类却可以是同一个。因此有这种区别。也许你对同样的属性但是对于类与实例的不同会是不同的结果感兴趣，我会在补充知识中告诉你是如何做到的。这个功能在 `Uliweb` 的 `ORM` 中也使用到了。

**第 18 行** 关于 `POST` 的处理没有太特殊的，因此我直接讲重点。通过前面的 `GET` 处理，会在第一次用户访问时去调用 `show.html` 模板，然后在这个模板中，我们会输出一个表格(`Form`)，用户可以在这个表格中输入文本，然后提交。注意提交是使用的 `POST` 方法。在 `Django` 中有一种约定，对于只读的使用 `GET`，对于修改的使用 `POST`。所以 `Form` 的方法建议都使用 `POST`，这样也方便我们在一个 `URL` 的处理中，根据不同的方法来执行不同的处理。也许大家听说过 `REST`，基本上就是这个样子：同样的一个 `URL` 可以有不同的功能，区别就在于使用的方法不同。

对于不同的方法提交的数据都放在 request 对象中，GET 方法提交的数据大 request.GET 中，因此 POST 方法提交的数据就自然在 request.POST 中。不过 request 还提供了一个属性：request.params，它综合了 GET 和 POST 的数据，所以如果你不想区分可以直接使用 request.params 属性。

Form 提供了 validate 方法用来对数据进行处理，它会首先从上传的数据中按字段属性的对应关系将数据取出来，然后调用数据类型的转换处理。在转换成功后，进行相应的数据校验，这里就有可能用到传入的 validator 了。有些特殊的字段已经内置了一些 validator。然后，如果你定义了对整个 form 进行校验的函数，还会调用它进行全局性的校验处理。上述工作全部成功的话，会返回一个 True。因此根据返回值可以知道校验是否成功。同时，当成功时，也意味着你可以得到转换后的 Python 类型，而不是原始的 HTML 字符串。一旦转换成功，通过 FormInst.Field.data 可以得到某个字段的值。整个转换后的结果是存放在 FormInst.data 属性中，它是一个 dict 对象，每个 key 就是字段属性的名字。你也可以使用它。如果转换失败，那么通过 FormInst.Field.error 可以得到出错的信息，也可以通过 FormInst.errors 得到一个 dict 对象，存放着所有字段属性的出错信息。同时对于整个 Form 的校验函数当出错时，在 FormInst.errors 中的 key 是 '\_'。

因此，第 18 行的意思就是对上传的 POST 数据进行校验和转换，并判断结果。

**第 19 行** 重新输出表格对象和 content 值。注意，这个表格对象是转换后的对象，因此它不再是初始状态了，而是有着上传后的数据。因此再输出时，表格中会带有上传后的数据。content 就不多说了，可以看到它使用 f.content.data 来得到对应的值。

**第 20 行** 这行是当校验失败时再重新输出表格。那么这里并没有特殊的处理，因为出错信息已经在表格对象之中了，在使用 Form 输出 HTML 代码时，会自动输出出错信息的。

## 创建 base.html

奇怪，为什么不是 show.html。别着急，base.html 将是一个通用的模板，它定义了基本的模板结构，可以作为一个基础模板，然后其它的模板可以从这个模板进行扩展，这样可以只定义相关内容就可以了，减少了重复开发，提高了复用性。许多 Web 框架的模板系统都有这样的功能。

在 ch5/apps/ShowText/templates 下创建 base.html，内容为：

```
1     <html>
2     <head>
3         <title>{{block title}}Untitled{{end}}</title>
4         <meta http-equiv="Content-type" content="text/html;
charset=utf-8" />
5     </head>
6     <body>
7         {{block main}}
8     </body>
9     </html>
```

**第3行** 定义了一个 title，注意这里是一个新的模板标签-block。它的语法为：

```
{{block blockname}}content{{end}}
```

一个 block 标签用于标识一个块，它有开始和结束标签，在开始标签中定义了这个块的名字。在一个模板中不能出现同名的块名。中间是它的内容。它是可以被子模板覆盖的。如果子模板中定义了同名的块，则子模板中的块将覆盖父模板。如果子模板中没有定义则仍然使用父模板中的块。在块的内容中，还可以定义块。因此，如果子模板没有定义 title 块的话，就会使用"Untitled"。

比较在 web2py 中是没有这个功能的，这是我扩展的。并且我将其发布到了 web2py 的邮件列表中。不过好象没有理会。也许这个功能不重要？

**第7行** 定义了一个新的 block：main。

## 创建 show.html

在 ch5/apps/ShowText/templates 下创建 show.html，内容为：

```
1     {{extend "base.html"}}
2     {{block main}}
3     <div style="width:600px">
4         {{<<form}}
5     </hr>
6     {{if defined('content')}}
7         {{from uliweb.utils.textconvert import text2html
8             out.write(text2html(content), escape=False)
9         }}
```

```
10     {{pass}}
11     </div>
12     {{end}}
```

**第 1 行** 这是一个新的标签。它的作用是从一个父模板进行扩展，就象面向对象中的继承一样。它的语法是`{{extend 父模板}}`，这里父模板可以是一个字符串，也可以是一个变量。这个例子中是一个字符串，正好是我们前面定义的 `base.html` 的名字。

**第 2 行** 表明这是对 `main` 块的覆盖。

**第 3 行** 定义了一个 `div` 标签，并设定宽度为 `500px`。

**第 4 行** 输出 `form` 对象。这里`<<`在第 4 章讲过，它不会对存在的 HTML 特殊符号，如：`<>`进行转义。如果一个对象不是字符串，在使用时，它会自动调用 `str()` 进行处理。而 `form` 对象正好有一个 `__str__` 的定义，可以直接输出为字符串。因为输出的是一个表格，所以带有 `<>` 符号，而正因为我们想保持这些符号，所以不希望转义，因此要使用这个标签。

**第 6 行** `defined()` 函数是模板本身提供的，它用来检查一个变量是否在模板中存在。那么它接受一个字符串形式的变量名。这一点非常重要。这与模板本身的实现相关。

Uliweb 的模板系统是从 `web2py` 来的，但是还是算比较简单的一种。它的主要思想就是把`{{}}`标签之外的内容转为 `out.write()` 这样的输出，然后把`{{}}`中的内容在处理与 `out.wirte()` 这样的代码混合在一起，最终形成一个 `python` 代码，然后通过执行输出模板处理后的结果。因此象`{{if}}`这样的标签，其实并不是一个有着特殊的标签，它只是被认为是 `Python` 的代码。所以你还可以看到这个 `if` 语句后面有一个 `':`，这就是 `Python` 的语法！只有很少数据的标签是有着特殊处理的，如：`extend`, `include`, `=`, `<<`, `use`, `end` 等。其它的均被视为嵌入的 `Python` 代码。

而判断一个变量是否存在并不是很容易，因为这个变量可能在任何地方出现，比如 `if` 语句，或表达式中。而这些并不是特殊的标签，只是普通的 `Python` 代码。有些模板系统，如：`mako` 和 `werkzeug` 自带的模板有判断变量是否存在的功能，它们是通过 `ast`(抽象语法树)来做的，但是比较难。并且如果是在 `GAE` 环境下，`ast` 模板的功能是被禁止的，因此根

本用不了。而 mako 为此有一个特殊的版本，不过说实在的，我的确没有看懂，感觉很复杂，所以在 Uliweb 中实现这一功能。但是提供了一个变通的 defined() 函数。

我们可以想象实际的场景。一般我们需要判断的并不是模板中出现的变量，而是外部传入的模板变量，因此只要判断外部的模板变量是否存在即可。而外部的模板变量是可以在模板模块中得到的，它是一个 dict 字典。因此这个 defined() 就是检查外部提供的模板变量中是否存在指定的变量名，同时它也检查了运行时的环境是否存在。因此这就是为什么你要提供的是变量名字符串，而不是变量名的字面形式的原因。比如：defined(test) 这样是不行的，需要为：defined('test')。

这句话是用来判断当 content 不存在时的处理。还记得上面我们的 view 函数吗？有些地方在返回时并没有定义 content 变量，所以有可能在调用 show.html 时，content 是不存在的。所以通过 defined('content') 来检查是否存在。

第 7,8,9 行 它们就是普通的 Python 代码。从 uliweb.utils.textconvert 模块中导入 text2html 函数。这个函数的功能就是将一个文本转换为 HTML，但是它会保留原有的空格，会将普通的回车转换为 <br> 标签，同时会将文本中存在的 http:// 形式的链接转为 <a> 标签。比较适合留言的处理。

比较 嵌入代码是与 Django 极大的区别。其实这块处理是可以放在 view 中完成，不过这里就是为了演示代码的使用。

从代码中可以看到，缩进是乱的。对，这是 Uliweb 或 web2py 模板的一个特点：不用考虑缩进。但是你要遵守规则，那就是：块语句结束时一定要加上 {{pass}}。而 pass 也正是 Python 的标准空语句。象 if, for, while, def, try 之类的结束时都要加上 {{pass}} 不然会报错。

7,8,9 行还可以改写成：

```

{{from uliweb.utils.textconvert import text2html}}
{{<<text2html(content)}}

```

这里{{<<}}与{{out.write(xxx, escape=False)}}是一样的，其实{{<<}}最终就是转为out.write 的代码。out 是 Uliweb 模板的一个内置的对象，专门用来输出的。

## 运行

在浏览器中输入 <http://localhost:8000/>可以看到：

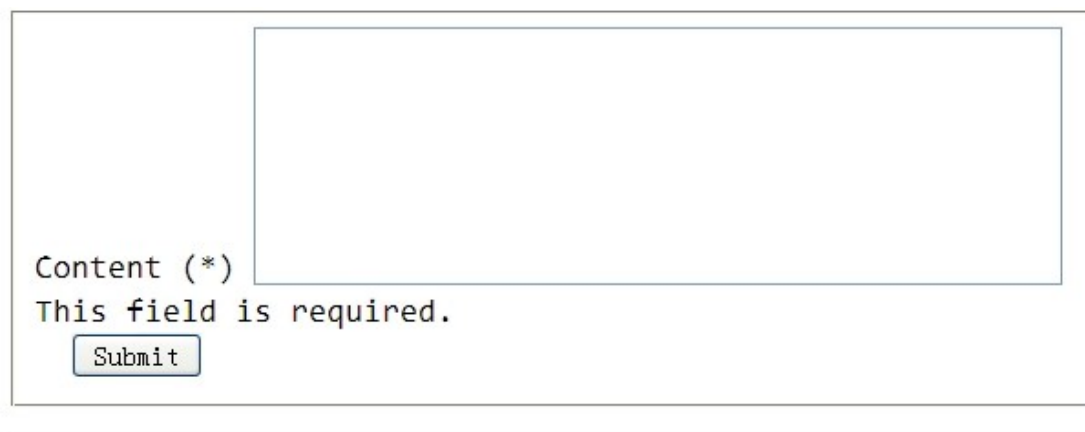
---



A screenshot of a web form. On the left, the text "Content (\*)" is displayed. To its right is a large, empty rectangular text input field. Below the label "Content (\*)" is a button labeled "Submit".

这是刚进入时的样子，有一个空白的输入区。下面让我们直接点击 Submit 看一下。

---



A screenshot of the same web form after the "Submit" button has been clicked. The text "Content (\*)" is still present. Below it, the error message "This field is required." is displayed. The "Submit" button remains visible below the error message.

报了一个错，说是这个字段是必输项。好，我们输入一些内容，再看：

```
    {{if defined('content')}}
      {{from uliweb.utils.textconvert
import text2html
      out.write('http://localhost:8000')
      }}
    {{pass}}
```

Content (\*)

---

```
    {{if defined('content')}}
      {{from uliweb.utils.textconvert import text2html
      out.write('http://localhost:8000')
      }}
    {{pass}}
```

内容没有什么变化，空白和回车都保留了，链接也转换了。

## 总结

看一看这章我们学了哪些东西：

- GET 和 POST 的使用
- Form 的简单使用：生成，数据校验，生成 HTML 代码
- response.template 的使用
- 模板中嵌入 Python 代码
- `{{<<}}` 和 `out.write()` 在模板中的使用
- `text2html` 的使用

## 补充知识

### *descriptor*

前面我说了，`Form.Field` 和 `FormInst.Field` 会得到不同的属性，这个功能的实现就是通过 Python 的 `descriptor` 机制。那么我在网上找了找，有一些相关说明的中文的文章有兴趣可以参考一下。这里不会讲得太多，因为有点复杂，我会尽量说清楚我的理解。

在设计框架的时候，你可能需要通过定义类属性的方式来绑定一些特殊的对象，就象 Form 中定义字段，在 ORM 中定义字段一样。这些字段看上去，是直接定义在类属性上，那么如果你了解 Python 的类的知识，你会知道它们就是类的属性，可以通过 `Class.attribute` 来访问。如果你创建了类的实例，并且没有类似 `Inst.attribute = xxx` 这样的代码，那么在读取时是可以使用 `self.attribute` 来引用类中定义的属性的。但是一旦你想修改它，简单的 `Inst.attribute = xxx`，只会在实例上定义一个新的属性，而不是修改类属性，要通过 `Class.attribute = xxx` 来做到。

从上面我们可以看出，一个属性，随着前面限定的对象不同可能是不一样的，可能是类属性，也可能是实例属性。很乱。让我们整理一下我们的需求：

- 在定义类时创建属性
- 使用 `Class.attribute` 时就是对 `Class` 的处理
- 使用 `Inst.attribute` 时就是针对 `Inst` 的处理

在 `uliewb` 中，Form 和 ORM 都在类和实例属性上是不同的。对于 Form 来说，类属性是公共的，不会因为实例的不同而不同，它就是对应的字段对象，相当于字段的定义。但当创建实例之后，通过实例来引用属性时，就变成对字段相对应的具体值的一个处理。这样的要求是很合理的。做法就是使用 `descriptor`。如果你想让一个实例成为 `descriptor`，那么这个类就至少需要有 `__get__` 或 `__set__` 方法。它们分别对应：使用和赋值的操作。例如，在 Form 中，每个字段类都是从 `BaseField` 类派生而来，在其中就定义了这两个方法，代码是：

```
def __get__(self, model_instance, model_class):
    if model_instance is None:
        return self
    else:
        return FieldProxy(model_instance, self)

def __set__(self, model_instance, value):
    raise Exception('Virtual property is read-only')
```

这样不管是通过 `Class` 还是 `Inst` 来访问一个属性，一旦这个属性是一个实例，并且它对应的类有相应的 `__get__` 或 `__set__` 方法时，这个属性相应的方法就会被调用。其实，这一点非常象 `property` 的处理。在 Python 2.4 版本之后，有一个内置的 `property` 方法，它可以定义一个属性，并且指定 `get`, `set` 时所使用的的方法。只不过那些方法是定义在这个类的内部，并且它无法区分引用者是类还是类的实例。而把属性定义为 `descriptor` 后，它们的

`__get__`和`__set__`有一个`model_instance`属性，如果它为`None`，表示是类引用，如果不是`None`，则表示是实例引用。因此上面的`__get__`就根据`model_instance`就可以区分是通过类还是实例来使用这个属性，这样就可以进行不同的处理了。

因此`descriptor`更多是在你想要对`Class`或区分`Class`和`Inst`时可能才需要使用，如果只是处理`Inst`，使用`property`就足够了。

# 第六章 调试与日志

越来越复杂了，这章让我们来点轻松的吧。在本章我们要学习一下如何使用调试和日志。

## 调试工具

当创建项目之后，在自动生成的 settings.ini 中就已经存在了一个选项 DEBUG 它缺省就是 True。当这个选项为 True 时会自动会进入调试状态，也就是说，一旦程序发生错误引发异常，Uliweb 会弹出一个 Debug 页面，在这个页面你可以看到出错的异常信息，traceback 栈的信息。甚至它还提供了 console 可以让你输入一些命令，提供了 inspect 可以让你查看内部变量的值。

这个功能并不是我自己写的，是底层的 werkzeug 提供的，不过我做了扩展。首先是针对 Uliweb 的模板。因为在执行模板时，Uliweb 会首先将模板转为 Python 代码，然后通过 exec 来执行。调试功能是可以显示源码的，但是对于模板却不适用，因为转换后的 Python 代码在内存中，无法直接找到。好在这个 Debug 工具提供了一种扩展机制，它可以查找定义在前面的调用者中的 `_loader_` 的信息，用它来显示源码。于是根据这一功能，作了一些扩展，可以在需要时，将渲染后的 Python 源码传给 `_loader_`，这样就可以显示出模板中的 Python 代码了。因此，现在不管是 Python 代码出错，还是模板出错都可以查看源代码。

另一个修改的地方就是将调试功能中的模板全部由 werkzeug 的模板处理改成了 Uliweb 的模板处理。因为 werkzeug 的模板用到了 ast 模块，而这在 GAE 是被禁止的，所以做了改造。

让我们先体验一下。

## 引发异常

可能在前面的学习中，你已经犯过错误看到了调试页面，也可能很顺利没有问题，那么我们可以人为制造问题。

在本章中我们不再创建新的项目了，就使用第五章的代码好了。

## 修改 views.py

在 ch5/apps/ShowText/views.py 中添加如下代码：

```
@GET('/debug')
def debug():
    a = 15
    raise Exception, 'This is a test'
    return {}
```

我们绑定/debug 到 debug()函数。很简单，定义了一个变量，将用来测试 inspect 功能，然后引发了一个异常，强行进入调试页面，然后返回一个空 dict。

其实强行插入异常代码可以用于调试，查看一些内部的数据。

## 访问/debug

在浏览器输入 <http://localhost:8000/debug> ，可以看到如下页面：



## Exception

This is a test

D:\project\svn\uliweb-tests\learning\_uliweb\ch5\apps\ShowText\views.py in debug, line 26

### Traceback

A problem occurred in your Python WSGI application. Here is the sequence of function calls leading up to the error, in the order they occurred. Activate a code line to toggle context lines.

```
__call__ in d:\project\svn\uliweb\uliweb\core\SimpleFrame.py [console] [inspect]
631 response = self.call_endpoint(mod, handler, req, res, **values)
call_endpoint in d:\project\svn\uliweb\uliweb\core\SimpleFrame.py [console] [inspect]
444 result = self.call_handler(handler, request, response, **values)
call_handler in d:\project\svn\uliweb\uliweb\core\SimpleFrame.py [console] [inspect]
493 result, env = self._call_function(handler, request, response, **values)
_call_function in d:\project\svn\uliweb\uliweb\core\SimpleFrame.py [console] [inspect]
489 result = handler(**values)
debug in D:\project\svn\uliweb-tests\learning_uliweb\ch5\apps\ShowText\views.py [console] [inspect]
26 raise Exception, 'This is a test'
```

### Request Data

The following list contains all important request variables. Select a header to expand the list.

### WSGI Environ

Brought to you by DON'T PANIC, your friendly Werkzeug powered traceback interpreter.

你可以看到引发的异常，可以看到执行栈的情况。当你某个栈上点击时，它会打开，显示出错行相关的源码，如：

```
_call_function in d:\project\svn\uliweb\uliweb\core\SimpleFrame.py [console] [inspect]
489 result = handler(**values)
debug in D:\project\svn\uliweb-tests\learning_uliweb\ch5\apps\ShowText\views.py [console] [inspect]
19     return {'form':f, 'content':f.content.data}
20     else:
21         return {'form':f}
22
23 @GET('/debug')
24 def debug():
25     a = 15
26     raise Exception, 'This is a test'
27     return {}
```

在每条执行栈的右边有 console 和 inspect 链接，可以分别用来测试代码和显示内部变量。让我们看一下：

```
26 raise Exception, 'This is a test'
27 return {}
```

Name	Value
'a'	15

```
[console ready]
>>> print a
15
>>> b = '123'
>>> print b
123
>>> application
<uliweb.core.SimpleFrame.Dispatcher object at 0x00E564F0>
```

上面是 inspect 的结果，可以看到 a 这个变量。下面的 console 窗口，可以执行代码，并且可以看到可以直接使用 application。

有了这个调试器，查看源码定义还是挺方便的。当然它也有不足，比如源码显示只有一部分，不是全部的。

另外这个调试器在 GAE 上只能看，不能使用 console 功能。为什么？因为这个调试器在处理时要通过线程的 local 对象来保存当时的执行栈，但是 GAE 是集群部署，无法保证这个栈还可以被下次访问到，因此无法使用。

## Print 调试

除了使用调试器，还可以简单地输出 print 语句来调试。比如在 view 函数中直接加上 print request，然后在执行时，你可以在启动 Uliweb 的命令行窗口看到输出结果。这是一种非常方便的做法。

另外在模板中也可以使用 print，如：

```
{{print var}}
```

## 日志

一旦部署到 web server，通过调试器或 Print 可以就不方便了。因为调试器不会保存，Print 也是，而且 Print 有可能根本看不到。这时我们只能通过日志输出了。其实不需要做特别的处理，只要使用 logging 模块就好了。不过 Uliweb 提供了一个简单的封装，它使用标准的 logging 模块，所以 api 没有变化，只是在输出的日志上有些处理，比如：

```
INFO      2009-02-27 16:25:40,842 manage.py,43] Message
```

先是日志的级别，然后是时间，后面是执行的程序名，然后是输出日志时的行数，再往后是消息了。

不过这个格式是可以改的，缺省是这个样子。

在 ch5/apps/ShowText/views.py 中添加如下代码：

```
@GET('/log')
def _log():
    from uliweb.utils.common import log
    log.info('This is a test message')
    return {'a':[1,2,3]}
```

通过 uliweb.utils.common 模块导入 log 函数。然后输出日志，使用 info 级别。可以在 console 中看到：

```
INFO      2009-02-27 17:00:44,092 views.py,32] This is a
test message
```

同时在浏览器我们可以看到：

# Default Template

This page is created from default template, because Uliweb can't find a matched template.

```
{'a': [1, 2, 3]}
```

有什么想法吗？因为我们返回的是一个 dict，但是我们并没有定义对应的模板文件啊。怎么还有东西输出。不过，从上面的页面描述可以看到，它是一个缺省模板，是当你没有定义时才会显示。并且它只在 Debug 状态下才会出现。如果你把 settings.ini 中的 DEBUG 改为 False，则再执行会显示出调试界面。

也许细心的朋友会问，为什么 DEBUG 已经是 False 了还会有调试界面？其实在启动开发服务器时，在命令行也可以设置调试状态。因此是否使用调试器在开发服务器下是由两个状态来决定的：一个就是 settings.ini 中的 DEBUG 信息，另一个就是命令行的 debug 状态。这两者只要有一个生效就会启动调试器。而且调试器本身是一个 wsgi 的 middleware，所以并不在 Uliweb 的核心 Dispatcher 中运行，是独立于 Dispatcher 的。而 Dispatcher 中只判断了 settings 中的 DEBUG 标志。

前面说到，日志输出格式是可以调整的，在 uliweb.utils.common 中定义了一个 get\_logger() 方法，可以向它传入一个 format 参数。缺省的格式为：

```
FORMAT = "%(levelname)-8s %(asctime)-15s %(filename)s,%  
(lineno)d] %(message)s"
```

如果不喜欢，可以改成你想要的格式。我写过一篇 Blog 《如何用 logging 输出一条有用的日志》关于如何设置的，有兴趣可以看一看。

## 404 和 500

404 表示页面没有找到，500 表示服务器内部错。在 Uliweb 中你可以自己定义输出的页面。很简单，只要在任何一个有效的 app 的 templates 下面定义 404.html 和 500.html

就可以了。Uliweb 会向它们传入一个 url 的变量，它就是 request.path 的值，即当前访问路径。

那么在调试状态下，Uliweb 提供了一个缺省的显示页面，如：





# Page Not Found

The requested URL `"/admin"` was not found on the server.

## Current URL Mapping is

URL	View Functions
<code>/ GET</code>	<code>ShowText.views.show_get</code>
<code>/ POST</code>	<code>ShowText.views.show_post</code>
<code>/debug GET</code>	<code>ShowText.views.debug</code>
<code>/log GET</code>	<code>ShowText.views._log</code>

它会显示出当前已经存在的 URL 及对应的 view 函数。如果定义了 GET 或 POST 方法，还会显示相应的方法。

## 使用 admin

admin 是什么？是一个管理工具。不过目前它可不象 Django 的 admin 那样可以录入数据，在 Uliweb 中的 admin 比较简单，目前只是显示一些 Uliweb 的内部信息。

admin 是 Uliweb 自带的一个 app，在 contrib 下面，有两种使用方式：

1. 加到 settings.ini 中的 INSTALLED\_APPS 中，作为一个标准的 app 来使用。
2. 使用 uliweb runadmin 来启动开发服务器。

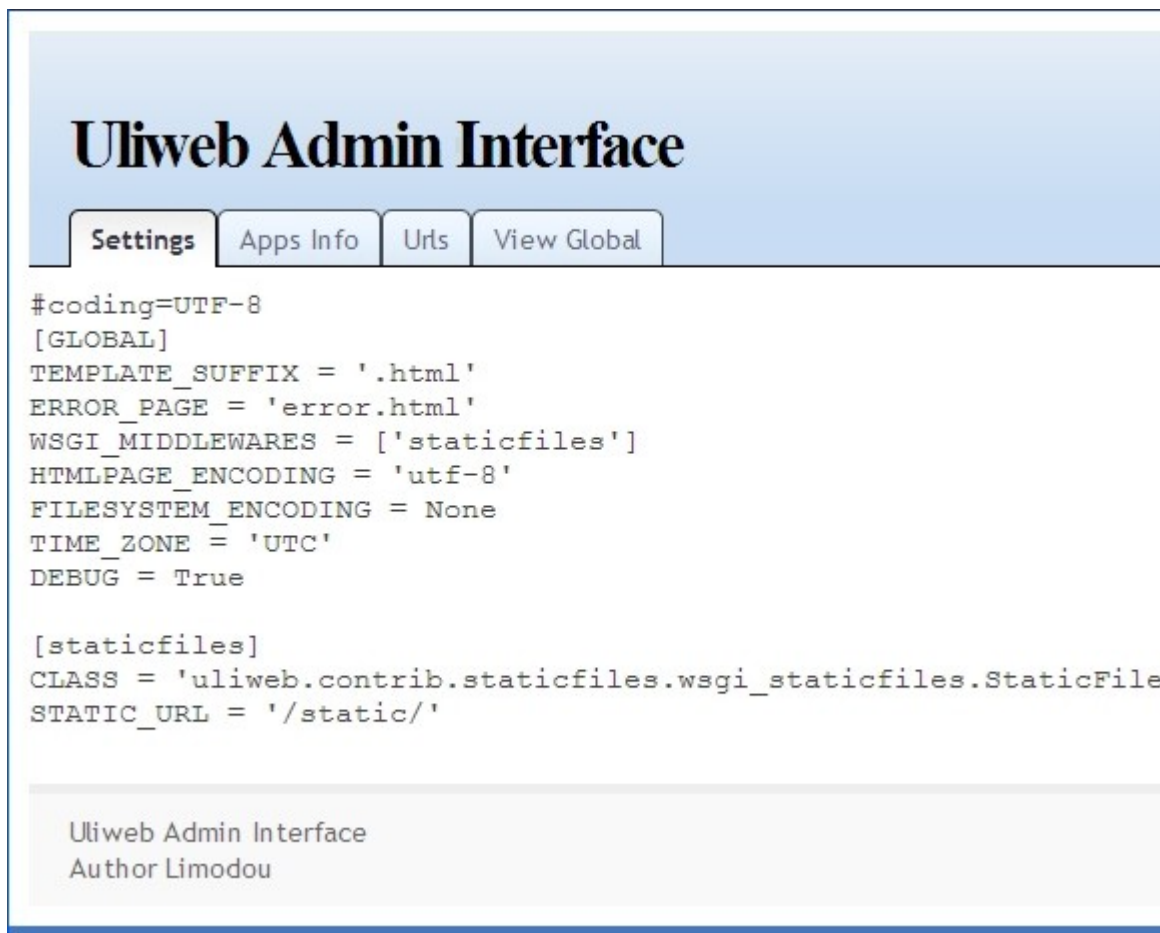
不错，Uliweb 为了方便使用 admin 功能，它提供了一个命令就是 runadmin，使用它尽管你没有在 INSTALLED\_APPS 中包含 admin，也一样会自动加入到有效 app 中。

目前提供的主要功能有：显示 settings 的完整信息、显示有效的 app、显示所有 URL 的定义等。让我们分别看一下。

首先使用:

```
uliweb runadmin
```

来启动开发服务器。然后在浏览器输入 <http://localhost:8000/admin> :



The screenshot displays the Uliweb Admin Interface. At the top, there is a navigation bar with four tabs: 'Settings' (which is selected and highlighted), 'Apps Info', 'Urts', and 'View Global'. Below the navigation bar, the main content area shows a configuration file snippet in a monospaced font. The configuration includes settings for template suffix, error page, WSGI middlewares, HTML page encoding, filesystem encoding, time zone, and debug mode. A section for 'staticfiles' is also visible, showing the class and static URL. At the bottom of the interface, there is a footer with the text 'Uliweb Admin Interface' and 'Author Limodou'.

```
#coding=UTF-8
[GLOBAL]
TEMPLATE_SUFFIX = '.html'
ERROR_PAGE = 'error.html'
WSGI_MIDDLEWARES = ['staticfiles']
HTMLPAGE_ENCODING = 'utf-8'
FILESYSTEM_ENCODING = None
TIME_ZONE = 'UTC'
DEBUG = True

[staticfiles]
CLASS = 'uliweb.contrib.staticfiles.wsgi_staticfiles.StaticFile:
STATIC_URL = '/static/'
```

Uliweb Admin Interface  
Author Limodou

不同的显示内容位于不同的 tab 中。首先看到的是 Settings，可以看到里面有不少的东西。而我们的 apps/settings.ini 下并没有太多东西啊。就是因为有些是所使用的 app 所带的。所以你看的是一个合并后的结果。比如上面的 DEBUG 之前的几项都是来自于 uliweb/core/default\_settings.ini，它是最初始的 settings 信息的配置，用来存放一些系统的默认值。在下面还有 staticfiles 相关的一些内容。而 WSGI\_MIDDLEWARES 的值是 ['staticfiles']，它的初始值是在 default\_settings.ini 中定义的，是空，但是这里有值是因为它是在 staticfiles 的 settings.ini 中定义的，然后因为是可变对象，所以合并了。

# Uliweb Admin Interface

Settings

**Apps Info**

Urts

View Global

App Name	Information
ShowText	
uliweb.contrib.admin	
uliweb.contrib.yaml	
uliweb.contrib.staticfiles	

Uliweb Admin Interface  
Author Limodou

然后是 Apps Info，可以看到有 4 个 app 了。ShowText 是我们创建的，其它的 3 个都是 admin 使用的。在 Uliweb 中已经内置了一些 app，都在 uliweb/contrib 目录下。其中 admin 是我们现在看到的这个 app，它会自动依赖 yaml 和 staticfiles。关于 app 间的依赖会在补充知识中进行介绍。

# Uliweb Admin Interface

[Settings](#)[Apps Info](#)[Urls](#)[View Global](#)

URL	View Functions
/ GET	ShowText.views.show_get
/ POST	ShowText.views.show_post
<a href="#">/admin</a>	uliweb.contrib.admin.views.index
<a href="#">/admin/appsinfo</a>	uliweb.contrib.admin.views.appsinfo
<a href="#">/admin/global</a>	uliweb.contrib.admin.views.globals_
<a href="#">/admin/urls</a>	uliweb.contrib.admin.views.urls
<a href="#">/debug</a> GET	ShowText.views.debug
<a href="#">/log</a> GET	ShowText.views._log
<a href="#">/static/&lt;path:filename&gt;</a>	uliweb.contrib.staticfiles.static

上面是在 project 中定义的 URL 的列表。前面是 URL，后面是对应的 view 函数，包括它所在的模板信息。同时如果你使用 GET 或 POST 处理的，它会显示对应的方法。

# Uliweb Admin Interface

[Settings](#)[Apps Info](#)[Urts](#)[View Global](#)

This page will display all available objects which can be used in a view function.

Global	Value
redirect	<function redirect at 0x00E44830>
url_for	<function url_for at 0x00E447B0>
settings	<pre>#coding=UTF-8 [GLOBAL] TEMPLATE_SUFFIX = '.html' ERROR_PAGE = 'error.html' WSGI_MIDDLEWARES = ['staticfiles'] HTMLPAGE_ENCODING = 'utf-8' FILESYSTEM_ENCODING = None TIME_ZONE = 'UTC' DEBUG = True  [staticfiles] CLASS = 'uliweb.contrib.staticfiles.wsgi_staticfiles.StaticFilesMiddleware' STATIC_URL = '/static/'</pre>
	<pre>GET /admin/global Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7 Accept-Encoding: gzip,deflate</pre>

这将显示在 view 中可用的对象和它们的值。这些对象可以直接在 view 函数中使用。

以上介绍了目前 admin 所具备的功能，以后可能还会有变化。通过 admin 可以了解你的项目的一些最终状态，特别是 settings，apps 和 urls 信息。目前是不具备 Django 那样的数据维护能力的。

## 总结

在本章我们主要学习了一些有关调试相关的一些内容：

- 调试器的使用
- 可以使用 print 进行简单地调试
- 使用日志记录信息，并且如何修改日志格式
- 404 和 500 错误页面的处理
- admin 功能的使用

## 补充知识

### *App 间的重用和依赖*

重用，再强调一下，我认为很重要。那么 app 就是在 Uliweb 中的最小重用单位。如果你有一些静态文件想重用，使用 app 进行封装，这样别人只要包括这个 app 就可以直接使用这些静态文件。你有一些全局性的模板，比如 404.html, 500.html, layout.html 之类的模板想全局使用，并且想以后复用，那么可以封装为 app。你有一些通用的 view 函数想使用，可以封装为 app，只要包括了 app，这些 view 自然生效。你有一些配置信息需要重用，可以只封装 settings.ini。甚至 uliweb 还支持在 app 中定义一个 commands.py 文件，可以用在命令行，增加新的命令。所以，通过 app 的形式可以包括实现一个功能所需要的几乎所有的要素，因此在 app 级别进行重用，在多数情况下只需要将这个 app 添加到 INSTALLED\_APPS 中去就可以了。有一点要强调的是，在 INSTALLED\_APPS 中定义的 app 是不关心顺序的。因此建议不要依赖 app 的顺序去做一些事情。

但是，当我们将功能分割得很细时，会发现一个功能需要依赖其它的 app 才可以工作，这就是 app 的依赖关系。在我们平时的开发中也存在这个问题：我们开发的类可能也需要其它的类，开发的库也需要其它的库，在 Uliweb 中也一样。那么 Uliweb 中提供了对这种依赖的支持。只在在需要依赖其它 app 的那个 app 目录下定义一个 config.ini 文件，如 admin 的 config.ini 的内容为：

```
[DEFAULT]
REQUIRED_APPS = ['uliweb.contrib.yaml']
```

在 REQUIRED\_APPS 中填入你要依赖的 app 的路径信息就可以了，可以有多个，它是一个 list。也许细心的你会想到，admin 不是只依赖了'uliweb.contrib.yaml'了吗？那么前

面的 admin 界面中的'uliweb.contrib.staticfiles'是从哪里来的。如果你再看一下 uliweb/contrib/yaml 目录下，它也有一个 config.ini，里面正好就是对'uliweb.contrib.staticfiles'的依赖。所以你可以看到，依赖是可以嵌套的。Uliweb 可以自动找到所有的依赖，并且把它们注入到有效的 app 列表中。

在找到所有的依赖之后，处理就简单了，Uliweb 只是简单地对全部有效的 app 进行导入处理，通过 `__import__` 导入 app 下的 start.py。这有什么用？这样会首先导入每个 app 下的 `__ini__.py`，然后再导入 start.py。可是你在前面会看到，从来没有定义过 start.py 呀。它可以不存在。其实这样的做法只是保证每个 app 会在启动时会处理到。因此，如果有一些要初始化的处理需要执行，你可以把它放在 `__init__.py` 中，也可以创建一个 start.py，将初始化处理放在 start.py 中。为什么要这样设计？原来是不存在 start.py 的处理的，我只是导入了模块，这样就自动导入了 `__init__.py`，也就是 `__init__.py` 无论如何，只要你导入了 app 模块，它就会被导入。那么 app 可能会有两种使用方式：一是作为真正的 app 被使用，这样 `__init__.py` 的初始化处理正好是你需要的。二是你并不真正想使用这个 app，它可能还提供了一些其它的模块可以被使用，因此你只想将其作为一个简单的模块来使用，因此并不想执行 `__init__.py` 中的初始化处理。因此将初始化处理放在 `__init__.py` 是解决不了这个问题的。因此，我增加了对 start.py 的导入，这样当一个 app 可以作为一个普通模块一样被使用时，并且有些初始化处理不想在普通情况下被使用，就可以放在 start.py 中，否则，如果无所谓，则可以还是放在 `__init__.py` 中，不用创建 start.py。因此 start.py 是用于只针对 Uliweb 启动时初始化的使用目的。当然，如果根本没有初始化的工作，自然不用关心这个问题了。

## 缺省模板

在前面也讲到了缺省模板是当找不到对应的模板时，如果正好处于调试状态，Uliweb 会使用缺省模板来显示当前的结果。Uliweb 提供的缺省模板就是 uliweb/core/default.html。同时用户也可以重新定义它，只要在任何一个有效的 app 下放一个 default.html 就可以了。

```
<h1>Default Template</h1>
<p>This page is created from default template, because Uliweb
can't
find a matched template.</p>
{{import pprint
from uliweb.utils.textconvert import text2html}}
{{<<text2html(pprint.pformat(_vars.copy()))}}
```

它不是太复杂。它会使用 `text2html` 来输出文本。同时它还使用了 `pprint` 模块来格式化变量。这里要注意的就是 `_vars`，它就是我们在调用模板时，传给模板的变量字典，在模板中可以使用 `_vars` 来使用它。

# 第七章 显示文本的改进

在第 5 章，我们学习了如何将用户录入的文本显示为 HTML 代码。在本章，我将对其进行改进，比如添加外部的 CSS，让它变得更好看一些。

## 准备

因为是在第 5 章的基础之上进行修改，因此让我们把 ch5 拷贝到 ch7。

## yaml\_shiny app

为了生成好看的 CSS 的效果，就象 admin 功能中看到的，我引入了 yaml 这套 CSS 框架。现在 CSS 在布局的处理上有 grid 和可变布局。Grid 是采用固定的表格，使用简单，但是一般不会随着窗口的大小变化而变化，比如 blueprint, yui 等。而 yaml 是一种可变的框架，当窗口大小变化时会比较好的适应，但是可能是就稍微复杂一些。在 yaml 的网站上还有一个 builder 工具，可以方便设计出你想要的效果。同时在下下载的包中有一些例子，可以将其中的 CSS 选出来直接使用。因此我使用了一套带有菜单条的叫 shiny 效果的 CSS 示例。为了方便重用，我将其封装成为一个 app。因此你可以直接从 <http://code.google.com/p/uliweb-tests> 代码中的 learning\_uliweb 目录下 ch7 找到这个 app，它的名字叫 yaml\_shiny。

在它的目录下有 config.ini。还记得我前面曾讲过 app 的依赖吗？在这里正好就用上了。打开它看一下：

```
[DEFAULT]
REQUIRED_APPS = [
    'uliweb.contrib.yaml',
    'uliweb.contrib.staticfiles',
]
```

它依赖两个其它的 app，一个是 yaml，这个是 yaml 框架所要使用的 css 文件，没有其它特别的。已经是放在 uliweb/contrib 下的，可以被其它的项目直接复用。因此，在 yaml\_shiny 这个 app 中，只是定义了基于 yaml 的一些扩展的 CSS。一般来说 yaml 不会直

接使用，而是需要用户把它作为一个基础，定制出符合自己要求的 CSS。另一个是 staticfiles，因为要处理静态文件，所以需要它。

在这个 app 的 templates 目录下，定义了两个模板，一个是 layout.html，它用来处理整个项目的基本布局。另一个是 menu.html，它将用来处理 CSS 菜单。

layout.html 因为内容挺多，就不在这里整个列出来了。我只讲一下主要的内容：

```
<title>{{block title}}Untitled{{end}}</title>
```

定义了一个 title 的块，可以被子模板替换。

```
<link href="{{=url_for_static('css/layout_sliding_door.css')}}"
rel="stylesheet" type="text/css"/>
<!--[if lte IE 7]>
<link
href="{{=url_for_static('css/patches/patch_sliding_door.css')}}"
rel="stylesheet" type="text/css" />
<![endif]-->
```

定义了要使用的 CSS 文件。这里使用 url\_for\_static 来包括静态文件。它是在 staticfiles 中定义的。

```
<div id="header">
  <div id="topnav">
    {{block topnav}}
    <span><a href="/login">Login</a> | <a
href="/register">Register</a></span>
    {{end}}
  </div>
  <h1><a href="/">{{block
project_title}}Project{{end}}</a></h1>
  <span>{{block project_description}}Description{{end}}</span>
</div>
```

最外层定义了一个 header 的 div 标签。然后先是一个 topnav 的 div，但是它的内容是通过 block 来定义的，因此是可以被替换的。同时在这个块中定义了几个链接。后面我们会看到，在子模板中，我们会把它定义为空，这样 topnav 就不会显示东西了。然后是项目的名称，也是通过块来定义的。再下面是项目的描述，也是一个块。

使用 block 的好处就是，它提供了可替换的内容块，并且当子模板不提供相应值的时候，可以使用原来定义的内容。

```
<div id="nav"> <a id="navigation" name="navigation"></a>
  <!-- skiplink anchor: navigation -->
  <div id="nav_main">
    <ul>
      {{block nav_menu}}
      <li id="current"><strong>Home</strong></li>
      {{end}}
    </ul>
  </div>
</div>
```

这里定义了导航菜单。它先引入了 menu.html，通过 include 标签。一会儿会讲到 menu.html。然后是菜单结构。因为是一个 CSS 菜单，因此是通过 <ul> 和 <li> 来定义的。而具体的菜单项 <li> 则使用 block 进行了处理，表明我们要在后面来覆盖。如果是当前菜单，需要在某个 <li> 添加 id="current" 属性。

```
<div id="main">
  <div>{{block main}}</div>
</div>
<!-- end: #main -->
<!-- begin: #footer -->
<div id="footer">{{block footer}}Footer<br/>
  Author Limodou</div>
```

这里分别定义了 main 和 footer 块。

再让我们看一下 menu.html。

```
{{
menus = [
  ('home', 'Home', '/'),
  ('page1', 'Page1', '/page1'),
]
def menu(current='settings'):
  for id, caption, link in menus:
    if id == current:
      out.write('<li id="current"><strong>
%s</strong></li>' % caption, escape=False)
    else:
      out.write('<li><a href="%s">%s</a></li>' % (link,
caption), escape=False)
  pass
pass
}}
```

首先是全部的菜单定义。现在为了示例只定义了两个菜单项：Home 和 Page1。每个菜单项是一个 3 元素的 tuple。每个 tuple 的第一个元素是菜单项的标识，第二个元素是显示的名字，第三个是对应的 URL。如果你有新的菜单项，可以修改这个文件进行添加。

然后是 menu 函数的定义。它是用来输出的，使用了模板中内置的 out 对象的 write 方法。同时要注意 escape=False，表示不进行 HTML 代码的转义。要注意，块语句结束后一定要添加 pass 语句。menu 函数接收一个参数，表示当前活动的菜单标识。因此，我定义 menu.html 的目的是为了通过定义函数来自动生成菜单，可以减化调用。注意，menu.html 这只是我为了方便才使用的方式，你不一定要这么做。甚至可以考虑放在 settings.ini 中，通过变量的自动合并来处理分布式的 app 的菜单项的定义。

## 修改 ShowText

不过，仅仅加入 yaml\_shiny 并不能让 ShowText 就自动用起来，因为它们之间没有任何关系。因此我们需要对 ShowText/templates/base.html 先做一点修改。原来的 base.html 定义了一个 HTML 页面的框架，现在我们将使用 layout.html 来替换。我们只要将内容替换为：

```
{{extend "layout.html"}}
```

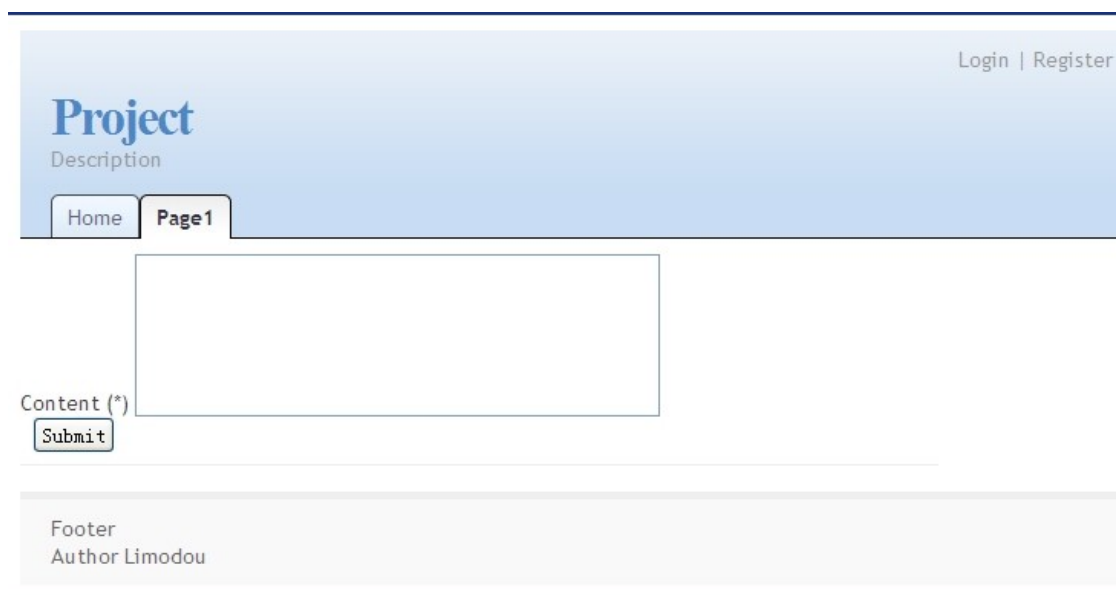
虽然我们主要显示用的是 show.html，但是因为 show.html 是从 base.html 派生来的。而现在 base.html 又是从 layout.html 派生来的，因此 show.html 也相当于从 layout.html 派生来。

下面打开 ShowText/templates/show.html，增加对 layout.html 的 block 的替换代码，其它的都不改，就在最后添加：

```
{{block nav_menu}}  
{{menu('page1')}}  
{{end}}
```

可以看到我们替换了 nav\_menu 这个块，然后调用了 menu 函数，而这个 menu 函数是在 yaml\_shiny 的 menu.html 中定义的。

改完之后，让我们启动开发服务器看一下效果。



可以看到，界面漂亮多了。当前的菜单中 page1，与我们调用 menu 函数想要的结果一样。其它的地方因为没有修改，所以都是缺省值。

那么为了生成这个菜单，我们修改了 yam1\_shiny 中的 menu.html，不过，这个方式并不是最完美的。因为 menu.html 是存在于 yam1\_shiny，因此它并不通用。解决办法就是把 menu.html 从 yam1\_shiny 转移到 ShowText 中。或者是使用其它的方法。为了简单，就让我们转移 menu.html。重新执行，程序应该不会有问题。不过这样，我们仍然需要知道在使用 yam1\_shiny 时，需要在你的 app 中定义一个 menu.html 文件。那么我们还可以将 layout.html 中的 include("menu.html") 去掉，将这句加到需要调用 menu 函数之间的地方，比如 show.html 中。好，让我们再改一下：

- 删除 yam1\_shiny 中 layout.html 中的 include("menu.html")
- 在 ShowText 中的 show.html 中，在 {{menu('page1')}} 之前加上 {{include("menu.html")}}

测试没有问题。经过上面的修改，我们让 menu.html 只与 ShowText 相关，并且如果你不使用 menu 函数，手工生成 <li> 的菜单项也不是很麻烦。

现在整体外观是好看多了，但是这个 Form 显示得还是很难看啊。让我们再调整一下。其实就是定义简单的 CSS 就可以了，不过这次我不想只是生成一个 css 文件，然后简单地向页面加入链接，我换一种方式。

## template app

不管是处理 CSS、静态文件还是 Javascript 都有一个麻烦事就是：要修改 HTML 文件的链接信息，而且有时可能要修改多处。因此当添加新的与展示相关的 app 时，手工修改 HTML 就使得重用变得很麻烦。因此我希望让修改可以做到最简单。那么我想到的一种方法就是在模板处理后，针对生成的 HTML 去动态添加所需要的链接信息。这样模板处理在前，链接修改在后，同时要保证不能有重复的链接。一种方法是把整个处理完全放在模板处理外面，比如通过 middleware，但是如何做到处理与页面的关联并不容易。要知道，并不是所有的页面想要的处理都是一样的。因此我认为比较好的方式是在模板中去定义，然后在模板处理之后再进行处理。这样我就对模板再次进行了扩展：首先是允许用户自定义标签，这样可以方便扩展。然后通过这种机制我开发了 `uliweb.contrib.template` 这个 app，它扩展了一个新的 `use` 标签。同时还修改了 `SimpleFrame.py` 中关于模板处理的代码，增加了一些 `dispatch` 处理。`dispatch` 你可能还不了解，我会在下面对它进行详细地介绍。通过 `dispatch` 机制，我定义了一些调用点，如：模板转换为 Python 代码前，模板处理之后。然后在 `template` 这个 app 中对这些点进行处理。在转换前的调用点，我生成了一个采集器对象。然后用户在模板中通过 `use` 来使用不同的模板插件，用来添加新的链接信息，但这些信息是被采集器收集。然后在模板处理之后的调用点，根据生成后的 HTML 代码，分析已经存在的链接，然后根据采集器中的链接进行过滤，去掉重复的，然后再添加到 HTML 代码中，就完成了链接信息的嵌入。这些链接信息本身也可以有模板代码，如使用 `{{=url_for_static}}` 来生成真正的 URL。同时为了处理更精确，还可以允许用户在要处理的 HTML 代码中加上一些指示标识，如：`toplinks`, `bottomlinks`, `codes`，表示相应的代码所要插入的位置。链接最终都将插入到 `<head>` 标签中。如果不定义指示标识，对于 `toplinks` 将插入到 `<head>` 内容最开始的地方。而 `bottomlinks` 将插入到 `</head>` 前也就是 `<head>` 内容的最后。`codes` 一般是跟在 `bottomlinks` 的后面。一个指示标识可以这样定义：`<!--toplinks -->`，它是使用标准的 HTML 注释，然后是关键字，关键字前后可以有 0 或多个空格。

对于用户来说，如果你不喜欢手工改 HTML 代码来插入新的链接信息，你可以通过定义模板插件来添加这些链接，然后使用 `use` 标签来引用这些链接。`template app` 会自动为

你处理好链接应该所在的位置。这种方式其实对于将 app 进行封装以实现更方便的重用更为有用。普通用户可能还是希望手工写链接，这样可以更简单。

在分析 template 之前，先介绍一下 Uliweb 中的 dispatch 机制。

## Dispatch 机制

Dispatch 机制是许多 web 框架都有的，如 Django。它其实就是类似于 Pub/Sub(发布/订阅)或信号，事件机制，无所谓叫什么。它其实是一种扩展机制。在我的 Uliweb 项目中应用非常多。在许多时候，程序在执行到一个特殊位置时，我们可能希望未来可以对其进行扩展，比如执行可扩展的处理，得到一个未知的信息。在这些地方，我们希望程序的处理并不是预先写死的，而是可以动态扩展的，因此我们可以只发出一个信号或事件，表明需要某些特殊的处理。而在程序的其它地方，有一些接收者，它们会等待着信号的出现。一旦发现有自己想关注的信号出现，它们就去做一些相关的工作。这样信号发出的地方，在开发时并不知道这些信号的接收者在哪里，它也不用去关心。而信号接收者也并不去关心何时、在哪会引发这些信号，它们只关心当信号来到后做哪些处理就行了。这样的处理使得我们的程序具有很强的扩展性。在 Uliweb 就提供了这种机制。它是通过 `uliweb.core.dispatch` 模块来实现的。

想要实现这种机制需要注意几个地方：

1. 分发点或调用点，它是发出信号的地方，可以有一些参数。在 Uliweb 中的信号是由两个东西构成的，一个是叫 `topic`，它是一个字符串，表示发出的信号是作什么用的。另一个叫 `signal`，表示一些特征。简单情况下，`topic` 是必须存在的，而 `signal` 则不必存在，缺省为 `None`。调用点就是一个函数调用，如：`dispatch.call(sender, topic, *args, **kwargs)`。它是依赖于 `dispatch` 模块来实现的，会直接对采集到的信息进行处理。
2. 接收者。它会绑定到某个 `topic` 和 `signal`。如果 `signal` 为 `None`，则只匹配 `topic`。否则两者全部匹配成功才会调用相应的接收者。接收者就是一个函数。
3. 采集系统。就是由 `dispatch` 模块提供。当接收者绑定到 `topic` 和 `signal` 上时，它们之间的对应关系就被保存到 `dispatch` 的内部数据中，这样当通过 `dispatch.call()` 调用时，就可以通过采集到的信息找到对应的接收者函数进行处理。

因此，使用它离不开 dispatch 模块。首先使用 dispatch.call()或 dispatch.get()来定义一个调用点。然后定义接收者函数，通过 dispatch.bind()将 topic 和 signal 与接收者函数绑定，完成接收者函数的定义和信息采集的工作。然后在执行时就可以查找了。

在 Uliweb 中，一个 topic 可以有一个以上的接收者，它们构成一个处理的链。因为处理是有先后的，它们在执行时也是有先后的。因此为了可以让用户定义执行的先后，在绑定 topic 与接收者函数时可以指定执行的顺序。有两种方式，一种是指定 kind，它是用级别表示的，有：HIGH, MIDDLE, LOW，分别表示：100，500，900 这几个数值。还可以使用 nice 来表示，它是精确的数值。在执行时，根据由小到大的顺序。因此 100 的优先级要高于 500 的优先级。

那么 call 和 get 的区别在于：call 只是单纯的执行，从匹配的接收者中的第一个执行到最后一个，不关心返回值。而 get 则关心返回值，一旦某个接收者返回一个非 None 的值，则执行结束，并返回这个结果。所以 call 和 get 的处理是不同的。

在 SimpleFrame.py 中已经预先定义了一些调用点，因此可以直接使用，比如：初始化，准备环境变量等。用户在自己的开发中也可以自由定义。

Uliweb 没有采用其它的 dispatch 库，而是将 Uliweb 项目中的 mixin 机制引入进来，并做了简化。主要是因为 Uliweb 中的 dispatch 可以进行排序，并且具备 call 和 get 两种执行方式。

## template 分析

先看一下它的 \_\_init\_\_.py。它里面的东西不少，我会挑一些主要的来讲。

```
1     @bind('startup_installed')
2     def startup(sender):
3         from uliweb.core import template
4         if sender.settings.TEMPLATE.USE_TEMPLATE_TEMP_DIR:
5             template.use_tmpdir(sender.settings.TEMPLATE.TEMPLATE_TEMP_DIR)
6         register_tag('use', use_tag_handler(sender))
```

**第 1 行** bind 是从 uliweb.core.dispatch 中来的。它的作用就是前面 dispatch 中所讲的绑定 topic 与接收者函数的。'startup\_installed'就是 topic，它是在 SimpleFrame.py 中定义的，用来表示初始化完成了。绑定这个 topic 可以当初始化完成之后再继续进行后续的一些处理。

**第 2 行** 定义接收者函数。函数名没有特殊要求。但是函数的第一个参数一定是发出 topic 的对象。因为我们要绑定到'startup\_installed'上去，它的发出者就是 SimpleFrame.Dispatche 的实例。这里你还可以定义 signal 参数，但是因为发出这个信号的调用点并没有设置，所以 signal 并不需要，除非调用点会设置这个参数。其它的参数则需要与调用点保持一致。

**第 3,4,5 行** Uliweb 中的模板中可以将中间生成的 Python 代码保存起来的，下次再使用时，如果模板没有发生变化，则直接使用缓冲的 Python 代码，可以提高执行的效率。这几行代码就是用来设置缓冲目录和是否启动缓冲的处理。你可以看到，它使用了 sender.settings。而 sender 正是当前的 Dispatcher 实例，它有许多的属性，其中 settings 就是读取所有 settings.ini 的对象。通过对 settings 对象的使用，可以猜到，这里使用了两个配置项：USE\_TEMPLATE\_TEMP\_DIR 和 TEMPLATE\_TEMP\_DIR。它们都是在 TEMPLATE 节中定义的。那么在 Uliweb 的 app 开发中我的建议是，为每个 app 中要使用到的与 app 相关的配置项，全部在 app 目录下的 settings.ini 中定义缺省值。因此，打开 uliweb/contrib/template 下的 settings.ini 可以看到：

```
[TEMPLATE]
USE_TEMPLATE_TEMP_DIR = False
TEMPLATE_TEMP_DIR = None
```

在省状态下是不启用缓冲机制的，因此缓冲路径为 None。如果你要启动，可以在 apps/settings.ini 下进行覆盖。

**第 6 行** 非常重要！它注册了一个新的标签：use 的处理函数。

因为我们一直都是通过 Uliweb 来自动处理模板，所以并没有手动来调用模板函数进行过处理。下面让我们简单看一看如何手工调用模板。在 uliweb.core.template 模块中，常用的有两个函数：template()和 template\_file()。前者是用来对字符串进行模板处理的，后一个是对模板文件进行处理的，功能差不多。而 template\_file()函数的原型是：

```
def template_file(filename, vars=None, env=None, dirs=None,
                  default_template=None, handlers=None)
```

- filename 是模板文件名。
- vars 是传入模板的变量，它是一个 dict 对象。
- env 是模板运行环境，在这个环境中的对象可以被模板直接使用。它也是一个 dict 对象。向它和向 vars 参数注入的对象都可以被模板直接使用。
- dirs 是模板文件搜索路径，它是一个 list 对象，可以有多个目录。
- default\_template 当模板没有找到时所使用的模板文件，可以进行缺省处理。
- handlers 用户定义标签的处理函数。它是一个 dict 对象。key 就是标签的名字，如'use'，value 就是对应的处理函数。在 Uliweb 的模板中，一个标签就是类似 {{tagname ...}} 的东西。

Uliweb 的模板在处理过程中，每解析到一个标签，都会去 handlers 中查找是否有匹配的标签和处理函数，如果有，则进行调用，没有则继续处理。一个处理函数的原型为：

```
def handler(value, container, stack, vars, env, dirs, writer):
```

具体的内容不会细说，如果以后介绍如何写 handler 再介绍。你只要知道，在 Uliweb 中是可以定义新的标签的。不过这里还没有考虑象 django 那样的块标签。

因此有了 handler 就可以在处理模板时对新的标签进行处理了。

第 6 行中的 use\_tag\_handler 就是生成了一个 handler。它接受一个 sender 参数，然后生成一个真正的函数。可以理解为它是一个 decorator。那么 use\_tag\_handler 也是在 \_\_init\_\_.py 中定义的。不过解释起来比较复杂，我只是说一下它的功能：

当解析出一个 use 标签后，将由模板系统自动调用这个 handler。在 handler 中根据标签的值，得到对就的字符串。其实它就是一个模块的名字。use 标签实现了一个微型框架，即它可以到每个有效的 app 的 template\_plugins 下去找指定的模块名，如果找到则导入。每个 plugin 模块需要定义一个 register 函数，它的原型是：

```
def register(app, vars, env, *args)
```

这个函数的返回值有特殊的要求，它需要一个 dict 对象，key 为：'toplinks', 'bottomlinks', 'codes'，分别对应不同的 HTML 代码，它们将用于与原 HTML 代码的合并。

这里有一个技巧，env 因为是一个字典，并且它会继续在模板中使用，因此可以在这里注入一些对象，这样当真正进行模板渲染时就可以直接使用了。注意：use 的处理是在由模板转为 Python 时进行的，因此这时并不会输出结果，所以在 use 中注入对象是可以被整个模板使用的。

use 的作用是调用相应的模板 plugin 模块，将相应的信息取出来，然后将这些信息合并，过滤过重复的，等着整个模板都渲染之后再使用它们。因此，象 toplinks 之类的信息将在模板处理之后使用，并不会在模板渲染的时候被使用，这一点与 env 不同。

那么这个 use\_tag\_handler 是何时被使用的呢？首先，在 SimpleFrame.py 中定义了一个调用点：

```
handlers = dispatch.get(self, 'get_template_tag_handlers')
```

可以看它，它使用 dispatch.get 来得到一个返回的 handler 集。那么你可能会想，这样不是只会返回一个吗？的确。但是我在 template 这个 app 中定义了一个方法，可以用来收集 handler，这样只要使用这个方法就可以将 handler 收集到一起，而不需要自己去绑定 'get\_template\_tag\_handlers' topic 了。这个函数就是 register\_tag，你可以从 template app 中导出在你的应用中使用。如：

```
from uliweb.contrib.template import register_tag
```

在 template app 中绑定了 'get\_template\_tag\_handlers'，这样你就不必再绑定这个 topic，直接使用 register\_tag 就好了。

```
@bind('get_template_tag_handlers')
def get_template_tag_handlers(sender):
    return get_handlers()
```

很简单。get\_handlers() 会返回通过 register\_tag 收集的 handler 集。

这样，所有注册的 handler 会传入模板处理函数。然后由模板系统来自动调用。

当模板处理完毕，SimpleFrame.py 会再发出一个 topic：

```
output = dispatch.get(self, 'after_render_template', text, vars,
e)
```

而 `_init_.py` 中则定义了对应的绑定：

```
@bind('after_render_template')
def after_render_template(sender, text, vars, env):
    from htmlmerger import merge
    collections = []
    for i in env.dicts:
        if 'collection' in i:
            collections.append(i['collection'])
    return merge(text, collections, vars, env.to_dict())
```

这个接收者函数将完成对 use 所收集的信息与 HTML 代码的合并。具体的内容就不再详述了。

它绑定了 `'prepare_template_env'` 这个 topic。这个 topic 是允许用户向缺省的模板环境中注入新的对象，这样可以直接在模板中使用。上面的代码则创建了一个 cycle 的 generator，可以直接用在模板中。这样的作法将使用加入的对象全局有效。因此如果你有一些对象或函数想直接使用在模板中，也可以这样做。

好，template 的 `_init_.py` 基本解释完了，小结一下：

- 处理模板缓冲的相关配置
- 注册 use 处理函数
- 提供注册标签的通用方法
- 绑定 `'get_template_tag_handlers'` topic 以返回所有注册的标签处理函数
- 绑定 `'after_render_template'` 以进行最后的 HTML 的加工
- 向模板环境注册新的 cycle 函数

在 template 中的其它文件就没有特别的了。其中 `config.ini` 中定义了 `staticfiles` 的依赖。

## formcss 分析

了解了 template 的功能，我们知道了可以在 app 下定义 template\_plugins 以添加新的 css 或 js 的链接，还可以向环境中注入一些新的对象。那么我想做的下一步就是再增加一个 formcss 的 app，它会定义一个好看一些的 css 用于 form 元素。在使用 Form 模块来生成 HTML 代码时，它会为一些特殊的字段设置相应的 css class，这样方便进行定制。

在 ch7 的源码中已经有一个写好的 formcss 的 app，它是放在 apps/helper/formcss 中的。下面让我们分析一下。

我们首先注意到 formcss 并不是直接放在 apps 下的，而是放在 helper。这样就说明，app 本身是可以分级管理的。当然，要注意 helper 下面会有一个 \_\_init\_\_.py 以确保它是一个 Python 的 package。

formcss 里面的东西不多：

- config.ini 它依赖于 staticfiles
- \_\_init\_\_.py 空文件
- static/haml-forms.css 我们定制的 form css
- template\_plugins/\_\_init\_\_.py 空文件
- template\_plugins/form.py 模板插件，用于 use。这里 form.py 与 use 将使用的"form"是匹配的

要说的就是 form.py。打开看一下：

```
def register(app, vars, env):  
    return {'bottomlinks': '{{url_for_static("haml-  
forms.css")}}'}
```

没什么特别的。首先是 register 函数，它是 use 标签在调用模板插件的要求。然后，它需要定义三个参数：

- app 就是当前 Dispatcher 实例

- vars 传入模板的对象
- env 模板运行的环境

它返回一个 dict。这里只定义了 bottomlinks，说明对应的链接将插入到 HTML 代码中的 head 的最后面。

## 修改 show.html

下面让我们修改 ShowText/templates/show.html，加上对 form 的调用。在原来的代码 `{{extend "base.html"}}` 中添加 `{{use "form"}}`。如：

```
    {{extend "base.html"}}
    {{use "form"}}
    {{block main}}
    <div style="width:600px">
```

## 创建 ShowText/config.ini

在修改完 show.html，你会发现，刷新页面没有什么变化。为什么？如果你使用 runadmin 看一看当前的 app，会发现 formcss 并不在有效的 app 之中，但是却有一个 helper。这是因为，我们创建这个项目中，并没有修改过 settings.ini，你可以在 admin 中查看 settings.ini 的信息，而 INSTALLED\_APPS 不存在。因此，这里会使用缺省的 app 查找机制，也就是会把 apps 下的视为 app 进行添加。但是 formcss 是在 helper 下的，因此只会把 helper 添加进来，它的子模块 formcss 不会添加。因此这里有两种做法：

1. 修改 apps/settings.ini 中的 INSTALLED\_APPS，把所有有效的 app 添加进去。但是这样添加的内容比较多。
2. 修改 ShowText/config.ini，添加依赖关系，这样我们仍然让 Uliweb 使用缺省的 app 查找策略，但是自动处理依赖。

我选择第二种方法，因此在 ShowText 下创建一个新的 config.ini 文件，然后加入下面的内容：

```
[DEFAULT]
REQUIRED_APPS = [
    'helper.formcss',
]
```

## 执行效果

再刷新页面，界面变了：



The screenshot shows a web application interface. At the top, there is a blue header with the word "Project" in a large, bold, blue font, and "Description" in a smaller, grey font below it. Below the header, there are two navigation buttons: "Home" and "Page1", with "Page1" being the active page. The main content area is a light grey box containing a form. The form has a label "Content (\*)" to the left of a large, empty text input field. Below the input field is a "Submit" button.

是不是这次 form 好看多了！

## 总结

本章我们学习的东西挺多的：

- app 的复用
- template app 的功能，使用及分析
- Dispatch 机制
- use 标签与 template\_plugins 机制
- app 依赖的实例