Django课件

本课程学习目标：搭建完整的Django开发环境，了解项目目录下各文件目录的含义和作用。掌握开发Templates、Models、Admin的基本配置方法和路由URL的设置等。能够开发一个简易网站。

# Django入门

## １．１主要知识点

1、http开发流程和django设计思想

2、环境搭建，运行第一个程序

3、了解django部分原理以及各组件含义

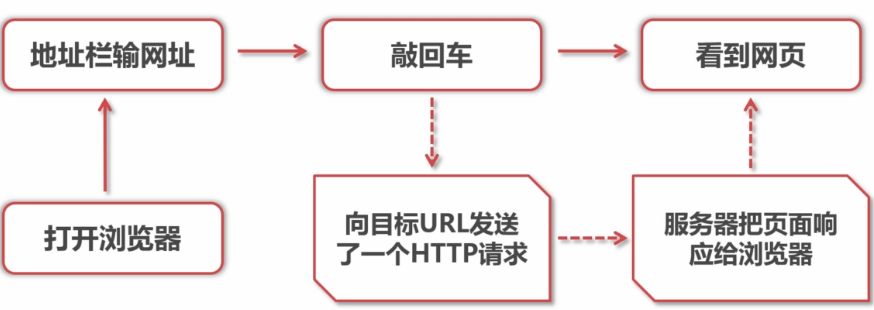
## １.２django的介绍：

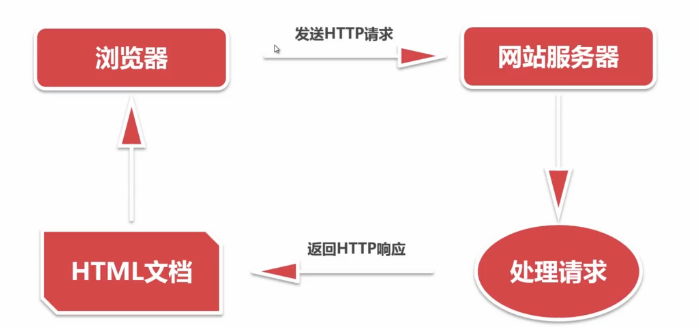
简介：Django作为一个Python的高级Web框架，功能非常强大，其自身封装了大量底层逻辑，并实现了ORM，同时，MVT结构保证了代码结构的完整性，这一切给了开发人员巨大的便利，让Web开发变得高效且简洁。

Django是一个基于Python的高级web开发框架，他能够让开发人员进行高效且快速的开发，高度集成（不用自己造轮子），免费并且开源。

Django官网：<https://www.djangoproject.com/>

## １.３上网流程：





## １.４什么是域名？什么网站名？什么是URL

http://mail.163.com/index.html

1)http://:这个是协议，也就是HTTP超文本传输协议，也就是网页在网上传输的协议。

2）mail：这个是服务器名，代表着是一个邮箱服务器，所以是mail.

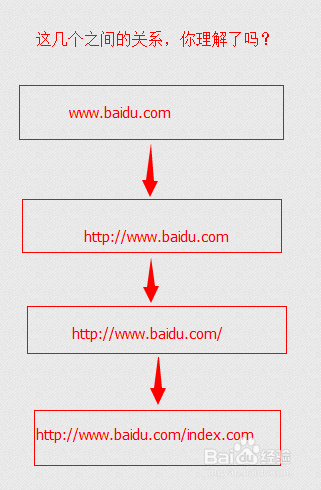
3)163.com:这个是域名，是用来定位网站的独一无二的名字。

4)mail.163.com：这个是网站名，由服务器名+域名组成。

5）/：这个是根目录，也就是说，通过网站名找到服务器，然后在服务器存放网页的根目录

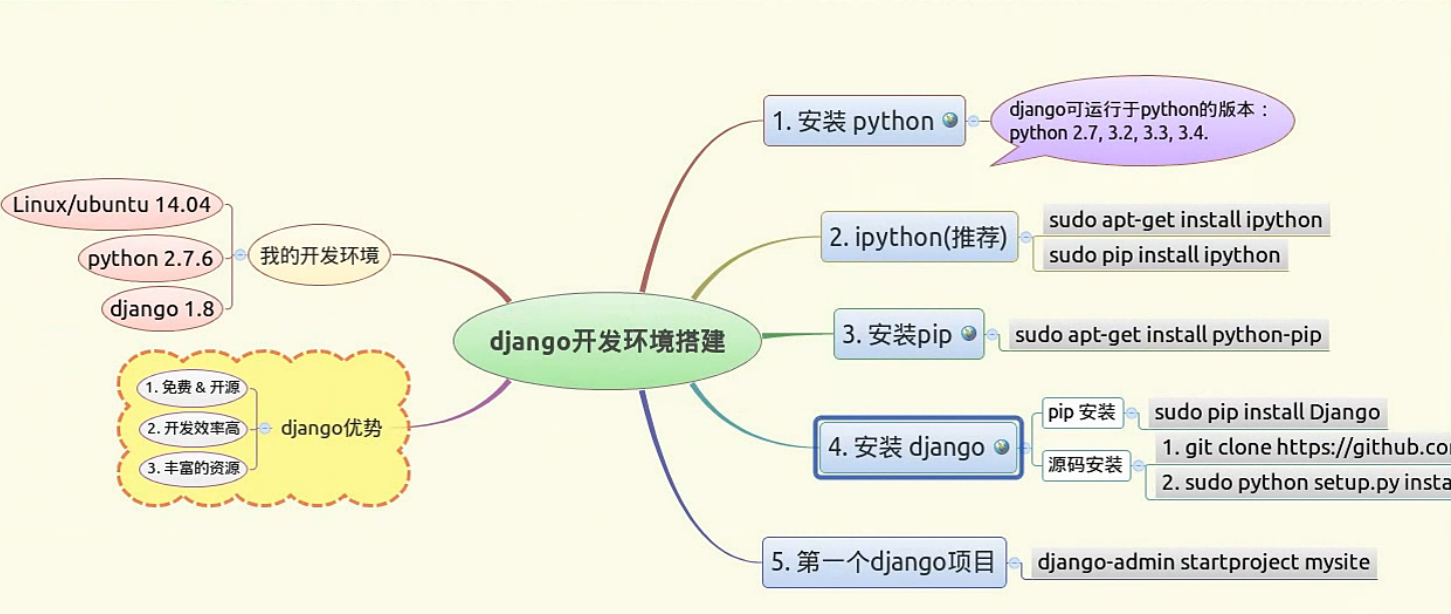
6:）index.html：这个是根目录下的默认网页（当然，163的默认网页是不是这个我不知道，只是大部分的默认网页，都是index.html）

7）http://mail.163.com/index.html:这个叫做URL，统一资源定位符，全球性地址，用于定位网上的资源。

[](http://jingyan.baidu.com/album/2c8c281df0afd00008252aa7.html?picindex=5)

# 第二章：安装调试第一个hellodjango程序

## ２.１搭建开发环境



### **２．１．１安装python**

（django支持版本最低2.7，上课用的2.7，但是本阶段课程内容的django部分可以兼容2.7以上的python，不涉及到python2和python3不同的不部分，所以大家可以自由选择）<https://www.python.org/>

### **２．１．2、安装django**

建议安装1.8.2版本，这是一个稳定性高、使用广、文档多的版本。安装方法如下：用PIP工具，执行pip install Django==1.8.2

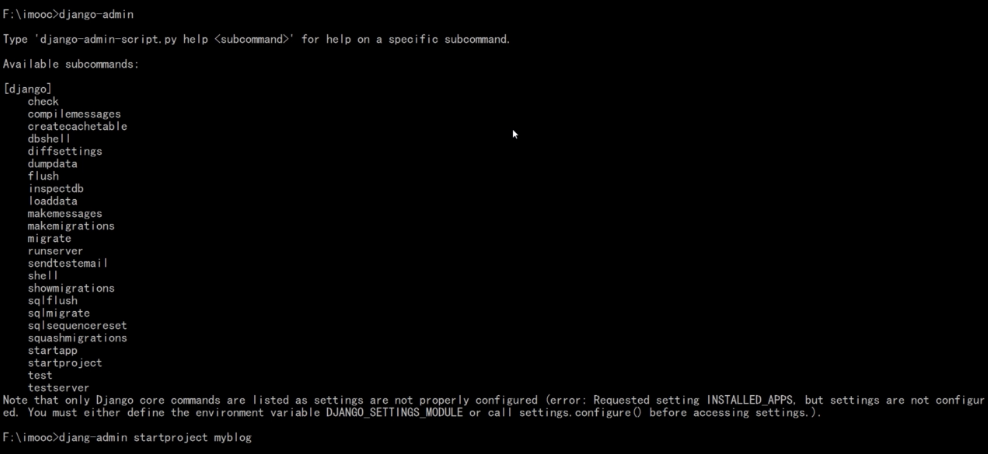
或者用离线安装，去github下载，执行python setup.py install安装。

## ２.２创建项目并调试

### **２.２.１、安装完成后，开始创建项目**

方法有多种，下面介绍2种

方法一（用控制台创建）

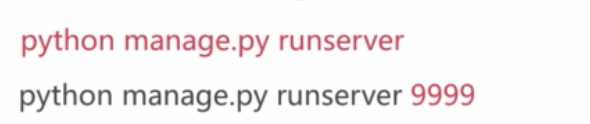


方法二用pycharm创建，具体的请参考操作教程

### ２.２.2、创建完毕后，可以启动项目

同理可以再控制台和pycharm里面启动。

启动项目 （端口）



### ２．２．３可以看到正常运行。

### ２．２．4、实现ｈｅｌｌｏｄｊａｎｇｏ

然后在view视图里面添加HttpRepons响应函数，实现第一个helldjango。

### ２.２.５保存即可重新启动，然后连接浏览器

### 2.2.6 django基本命令

1. 新建一个 django project

django-admin.py startproject project\_name

2. 新建 app

要先进入项目目录下，cd project\_name 然后执行下面的命令（下同，已经在项目目录下则不需要 cd project\_name）

python manage.py startapp app\_name

或 django-admin.py startapp app\_name

一般一个项目有多个app, 当然通用的app也可以在多个项目中使用。

与项目名类似 app name 也需要为合法的 Python 包名，如 blog，news, aboutus 等都是合法的 app 名称。

3. 创建数据库表 或 更改数据库表或字段

Django 1.7.1及以上 用以下命令

# 1. 创建更改的文件

python manage.py makemigrations

# 2. 将生成的py文件应用到数据库

python manage.py migrate

4. 使用开发服务器

开发服务器，即开发时使用，一般修改代码后会自动重启，方便调试和开发，但是由于性能问题，建议只用来**[测试](http://lib.csdn.net/base/softwaretest" \t "_blank" \o "软件测试知识库)**，不要用在生产环境。

python manage.py runserver 0.0.0.0:8000

# 当提示端口被占用的时候，可以用其它端口：

python manage.py runserver 8001

python manage.py runserver 9999

（当然也可以kill掉占用端口的进程，具体后面有讲，此处想知道的同学可查下 lsof 命令用法）

# 监听机器所有可用 ip （电脑可能有多个内网ip或多个外网ip）

python manage.py runserver 0.0.0.0:8000

# 如果是外网或者局域网电脑上可以用其它电脑查看开发服务器

# 访问对应的 ip加端口，比如 http://172.16.20.2:8000

5. 清空数据库

python manage.py flush

此命令会询问是 yes 还是 no, 选择 yes 会把数据全部清空掉，只留下空表。

6. 创建超级管理员

python manage.py createsuperuser

# 按照提示输入用户名和对应的密码就好了邮箱可以留空，用户名和密码必填

# 修改 用户密码可以用：

python manage.py changepassword username

1

7. 导出数据 导入数据

python manage.py dumpdata appname > appname.json

python manage.py loaddata appname.json

关于数据操作 详见：数据导入数据迁移，现在了解有这个用法就可以了。

8. Django 项目环境终端

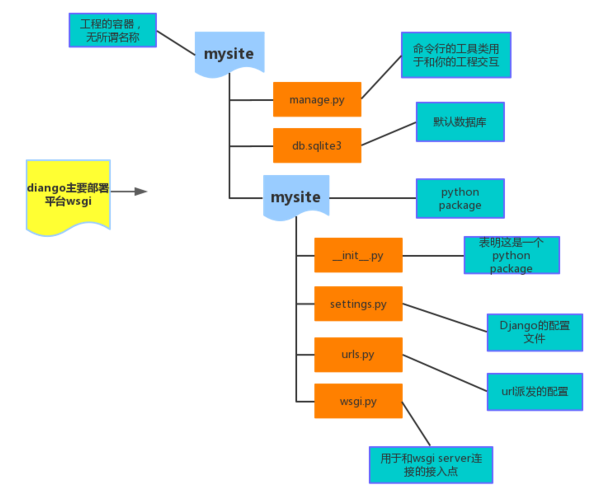
python manage.py shell

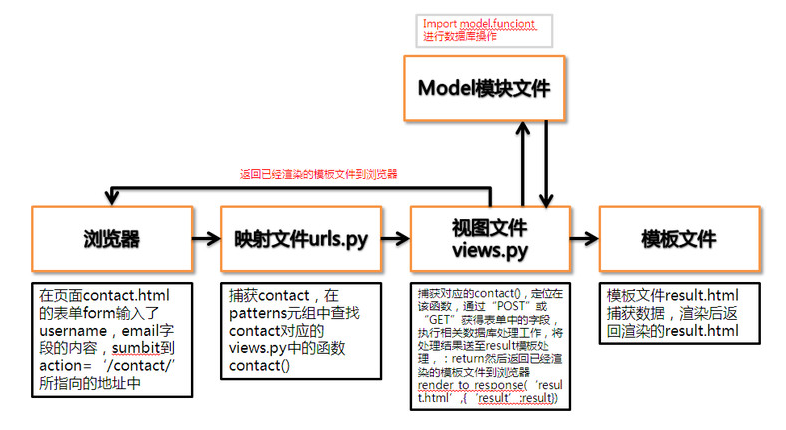
这个命令和 直接运行 python 或 bpython 进入 shell 的区别是：你可以在这个 shell 里面调用当前项目的 models.py 中的 API，对于操作数据，还有一些小测试非常方便。

9. 更多命令

终端上输入 python manage.py 可以看到详细的列表，在忘记子名称的时候特别有用。

## ２.３分析Django的目录结构





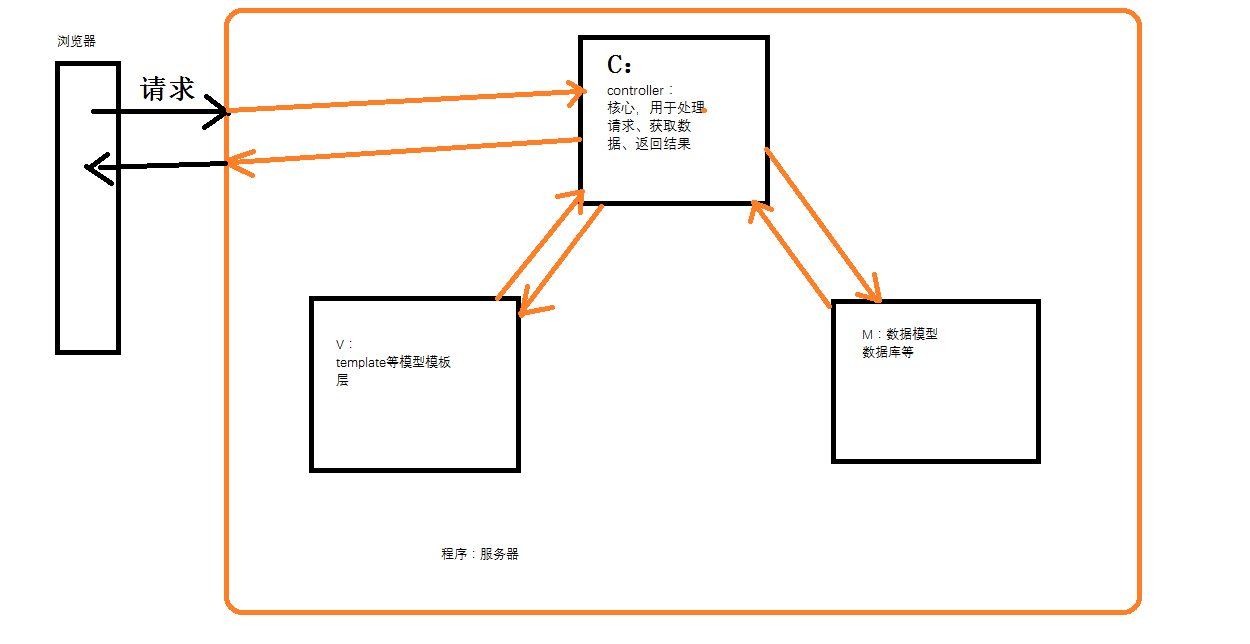


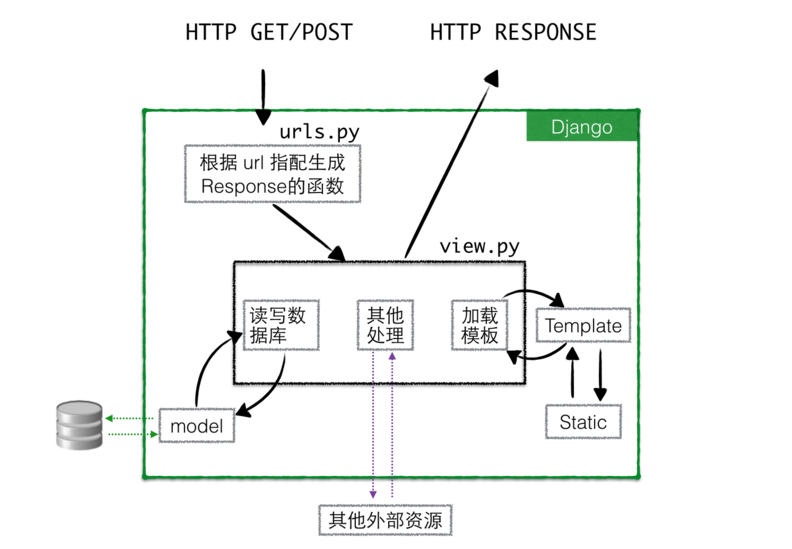


## ２.４深入分析MVT结构

MVC：当前各种开发语言中都拥有MVC框架，这种框架的核心思想便是：解耦。降低各功能模块之间的耦合性，方便将来变化时，更容易重构代码，最大程度上实现代码的重用。m表示model，主要用于对数据库层的封装；v表示view，用于向用户展示结果；c表示controller，是核心，用于处理请求、获取数据、返回结果。思想：低耦合高内聚（可以举例子 计算机结构），方便代码的更改、高可扩展性，向后兼容， 实现代码的复用等

MVT：Django是一款分层框架，用于使用python进行web开发，与MVC有所不同，属于MVT框架。m表示model，负责与数据库交互，v表示view，是核心，负责接收请求、获取数据、返回结果,t表示template，负责呈现内容到流程器。





# 第三章：虚拟环境搭建

之所以需要python的虚拟环境，是因为python安装的插件版本问题。分为Windows和LINUX两个内容。

## ３.１Windows

### ３.１.１安装virtualenv

如果你的python环境有pip，那么直接使用命令：pip install virtualenv 安装即可；否则需要下载源码，然后使用命令：python install setup.py 来安装。

### ３.１.２创建虚拟环境

安装之后需要新建virtualenv的独立环境，使用命令：

1. virtualenv envname  ##创建一个新的隔离环境
2. cd envname
3. Scripts\activate   ##激活并切换到virtualenv环境

具体可以查看其帮助命令：virtualenv -h，常用的参数有：

-p：指定一个版本python环境；通常当你的系统中安装了多个python版本时会用到；默认情况下virtualenv会优先选取它的宿主python环境，即它安装在那个python版本下就会默认选择哪个版本作为默认python隔离环境。

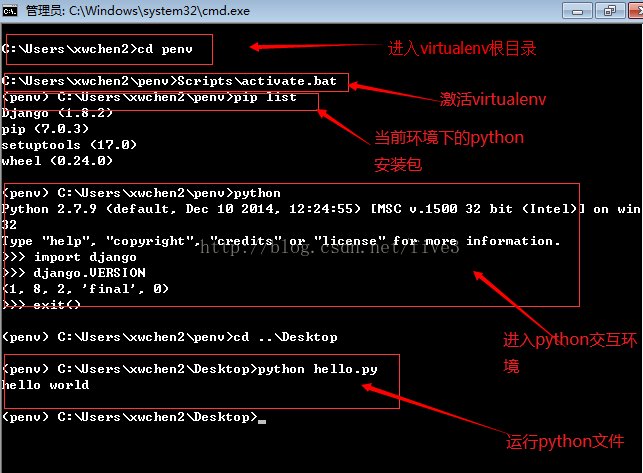
--no-site-packages：不使用系统环境的python安装包，即隔离包中不能使用真实python环境的安装包；当前版本这个选项是默认的。

--system-site-packages：与上面相反，使隔离环境能访问系统环境的python安装包

--distribute：copy一个python环境的分支，默认会安装setup、pip、wheel等基础模块。

### ３.１.３进入虚拟环境

我们在需要使用virtualenv环境之前，只要先激活然后使用以往同样的方式来执行python操作、运行python文件即可



创建并激活virtualenv环境后我们就可以正常的方式来使用和安装python安装包及模块了。也可以直接切换到需要执行的python文件目录，使用python xxx.py来保证在隔离环境下执行python文件。

### ３.１.４集成到开发环境

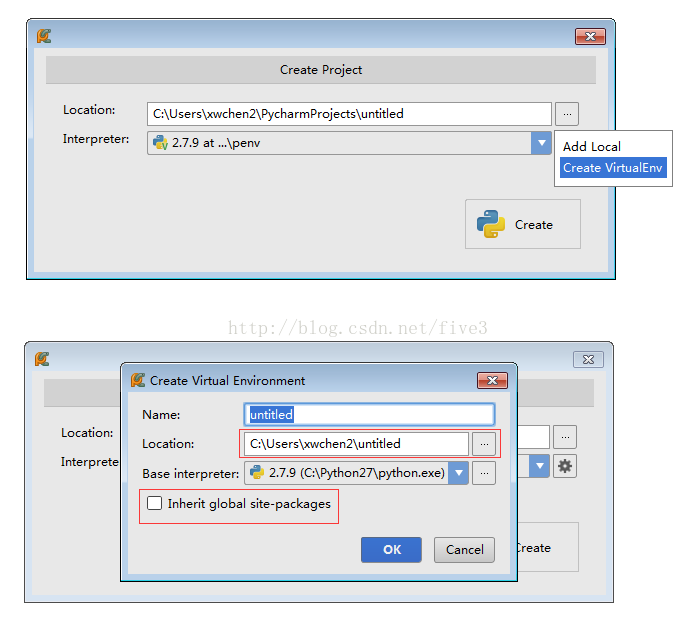
当前在PyCharm中是支持virtualenv环境集成的，其它的IDE暂时未研究，集成也是很方便的：

1、新建一个项目

2、点击解释器栏的设置icon

3、选择virtualenv

4、可以新建也可以选择一个已有的virtualenv环境



## ３.２ＬＩＮＵＸ

### ３.２．１安装virtualenv

**virtualenv实际上是一个python包，通过命令：sudo apt-get install python-virtualenv或者 sudo pip install virtualenv**

### ３.２.２创建python虚拟环境：

Mac:~ michael$ mkdir myproject

Mac:~ michael$ cd myproject/

创建一个独立的Python运行环境，命名为venv：

Mac:myproject michael$ virtualenv --no-site-packages venv

命令virtualenv就可以创建一个独立的Python运行环境，我们还加上了参数--no-site-packages，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

执行后，在本地会生成一个与虚拟环境同名的文件夹。新建的Python环境被放到当前目录下的venv目录。如果你的系统里安装有不同版本的python，可以使用--python参数指定虚拟环境的python版本

[root@localhost ~]# virtualenv --python=/usr/local/python-2.7.8/bin/python2.7 env1

实测默认情况下虚拟环境不会依赖系统环境的global site-packages。比如系统环境里安装了MySQLdb模块，在虚拟环境里import MySQLdb会提示ImportError。如果想依赖系统环境的第三方软件包，可以使用参数--system-site-packages。此外，也可使用virtualenvwrapper的toggleglobalsitepackages命令控制当前环境是否使用global site-packages，比如：virtualenv --python=F:\Python3K\python.exe --system-site-packages p3

[root@localhost ~]# virtualenv --system-site-packages env1

### ３.２.３启动虚拟环境

有了venv这个Python环境，可以用source进入该环境：



此时命令行前面会多出一个括号，括号里为虚拟环境的名称。以后easy\_install或者pip安装的所有模块都会安装到该虚拟环境目录里。

下面正常安装各种第三方包，并运行python命令：

(venv)Mac:myproject michael$ pip install jinja2

...

Successfully installed jinja2-2.7.3 markupsafe-0.23

(venv)Mac:myproject michael$ python myapp.py

...

在venv环境下，用pip安装的包都被安装到venv这个环境下，系统Python环境不受任何影响。也就是说，venv环境是专门针对myproject这个应用创建的。

### ３.２.４退出虚拟环境： deactivate

(env1)[root@localhost env1]# deactivate

[root@localhost env1]#

此时就回到了正常的环境，现在pip或python均是在系统Python环境下执行。完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令source venv/bin/activate进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令python和pip均指向当前的virtualenv环境。

## ****３.３扩展知识：使用virtualenvwrapper****

virtualenvwrapper是virtualenv的扩展工具，可以方便的创建、删除、复制、切换不同的虚拟环境。

### ３.３.１、安装virtualenvwrapper：

pip install virtualenvwrapper

创建一个文件夹，用于存放所有的虚拟环境：mkdir ~/workspaces

设置环境变量，把下面两行添加到~/.bashrc里。

[root[@localhost](http://my.oschina.net/u/570656" \t "_blank) ~]# export WORKON\_HOME=~/workspaces  
[root[@localhost](http://my.oschina.net/u/570656" \t "_blank) ~]# source /usr/bin/virtualenvwrapper.sh

然后就可以使用virtualenvwrapper了。

### ３.３.２、创建虚拟环境：mkvirtualenv [虚拟环境名称]

[root@localhost ~]# mkvirtualenv env1

New python executable in env1/bin/python

Installing setuptools, pip...done.

(env1)[root@localhost ~]# mkvirtualenv env2

New python executable in env2/bin/python

Installing setuptools, pip...done.

(env2)[root@localhost ~]#

注意：mkvirtualenv可以使用virtualenv的参数，比如--python来指定python版本。创建虚拟环境后，会自动切换到此虚拟环境里。虚拟环境目录都在WORKON\_HOME里。

### ３.３.３.列出虚拟环境：lsvirtualenv -b

(env2)[root@localhost ~]# lsvirtualenv -b

env1

env2

### ３.３.４.切换虚拟环境：workon [虚拟环境名称]

(env2)[root@localhost ~]# workon env1

(env1)[root@localhost ~]# echo $VIRTUAL\_ENV

/root/workspaces/env1

### ３.３.５.查看环境里安装了哪些包：lssitepackages

### ３.３.６.进入当前环境的目录：cdvirtualenv [子目录名]

(env1)[root@localhost ~]# cdvirtualenv

(env1)[root@localhost env1]# pwd

/root/workspaces/env1

(env1)[root@localhost env1]# cdvirtualenv bin

(env1)[root@localhost bin]# pwd

/root/workspaces/env1/bin

进入当前环境的site-packages目录：cdsitepackages [子目录名]

(env1)[root@localhost env1]# cdsitepackages

(env1)[root@localhost site-packages]# pwd

/root/workspaces/env1/lib/python2.6/site-packages

(env1)[root@localhost site-packages]# cdsitepackages pip

(env1)[root@localhost pip]# pwd

/root/workspaces/env1/lib/python2.6/site-packages/pip

### ３.３.７控制环境是否使用global site-packages：toggleglobalsitepackages

### ３.３.８.复制虚拟环境：cpvirtualenv [source] [dest]

[root@localhost ~]# cpvirtualenv env1 env3

ing env1 as env3...

(env3)[root@localhost ~]#

### ３.３.９退出虚拟环境：deactivate

### ３.３.１０.删除虚拟环境：rmvirtualenv [虚拟环境名称]

[root@localhost ~]# rmvirtualenv env2

Removing env2...

### ３.３.１１ 生成可打包环境

某些特殊需求下,可能没有网络, 我们期望直接打包一个ENV, 可以解压后直接使用, 这时候可以使用virtualenv -relocatable指令将ENV修改为可更改位置的ENV

#对当前已经创建的虚拟环境更改为可迁移

[root@localhost ~]# virtualenv --relocatable ./

Making script ./bin/easy\_install relative

Making script ./bin/easy\_install-3.4 relative

Making script ./bin/pip relative

Making script ./bin/pip3 relative

Making script ./bin/pip3.4 relative

当前的ENV都被修改为相对路径, 可以打包当前目录, 上传到其他位置使用，但这并不能使虚拟环境跨平台使用

### ３.３.１２ 获得帮助

 virtualenv -h

**一些资料查询：**

[1] 使用virtualenv搭建独立的Python环境

[http://qicheng0211.blog.51cto.com/3958621/1561685](http://qicheng0211.blog.51cto.com/3958621/1561685" \t "_blank)

[2] python virtualenv使用 解决依赖、版本等问题

[http://www.ttlsa.com/python/python-virtualenv/](http://www.ttlsa.com/python/python-virtualenv/" \t "_blank)

[3] Virtual Environments

[http://docs.python-guide.org/en/latest/dev/virtualenvs/](http://docs.python-guide.org/en/latest/dev/virtualenvs/" \t "_blank)

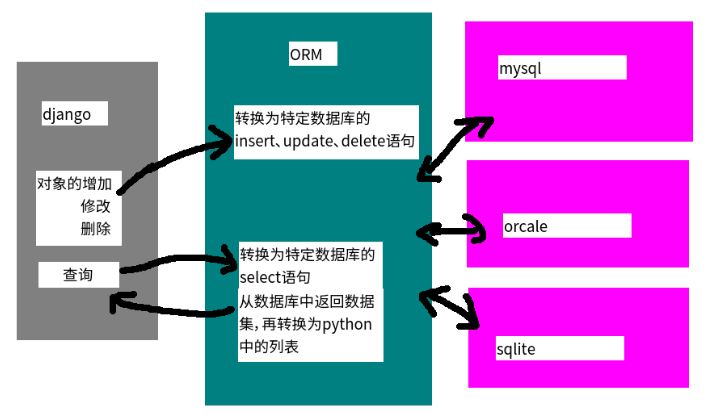
[4] Virtualenv + SublimeText 的Python开发环境

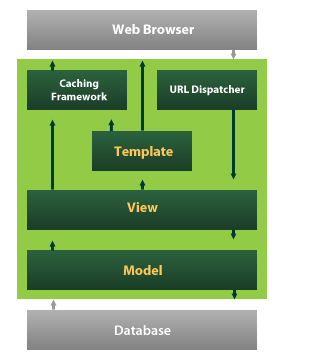
[http://python.jobbole.com/85307/](http://python.jobbole.com/85307/" \t "_blank)

# 第四章 模型

## ４.１ORM简介

1. MVC框架中包括一个重要的部分，就是ORM，它实现了数据模型与数据库的解耦，即数据模型的设计不需要依赖于特定的数据库，通过简单的配置就可以轻松更换数据库
2. ORM是“对象-关系-模型”的简称，主要任务是：根据对象的类型生成表结构，将对象、列表的操作，转换为sql语句，将从sql语句中查询到的结果转换为对象、列表。
3. 这极大的减轻了开发人员的工作量，不需要面对因数据库变更而导致的无效劳动。
4. Django中的模型包含存储数据的字段和约束，对应着数据库中唯一的表





## ４.２使用MySQL3实现一个数据的操作。

例子：博客信息中文章标题和内容(参考投票或者mysql项目)

## ４.３使用MySql数据库

### ４.３.１在虚拟环境中安装mysql包

pip install mysql-python。有时有错误，需要运行一些插件，可以在下面网站中下载：<http://www.lfd.uci.edu/~gohlke/pythonlibs/#mysql-python>

### ４.３.２数据库配置

django的数据库相关表配置在models.py文件中，数据库的连接相关信息配置在settings.py中

#### 4.3.2.1models.py相关相关参数配置

定义属性时，需要字段类型。字段类型被定义在django.db.models.fields目录下，为了方便使用，被导入到django.db.models中。

使用方式：导入from django.db import models，通过models.Field创建字段类型的对象，赋值给属性。

from django.db import models

class userinfo(models.Model):

name = models.CharField(max\_length=30)

email = models.EmailField()

memo = models.TextField()

可用的字段：

1、models.AutoField　　自增列 = int(11)

　　如果没有的话，默认会生成一个名称为 id 的列，如果要显示的自定义一个自增列，必须将给列设置为主键 primary\_key=True。

2、models.CharField　　字符串字段

　　必须 max\_length 参数

3、models.BooleanField　　布尔类型=tinyint(1)

　　不能为空，Blank=True

4、models.ComaSeparatedIntegerField　　用逗号分割的数字=varchar

　　继承CharField，所以必须 max\_lenght 参数

5、models.DateField　　日期类型 date

　　对于参数，auto\_now = True 则每次更新都会更新这个时间；auto\_now\_add 则只是第一次创建添加，之后的更新不再改变。

6、models.DateTimeField　　日期类型 datetime

　　同DateField的参数

7、models.Decimal　　十进制小数类型 = decimal

　　必须指定整数位max\_digits和小数位decimal\_places

8、models.EmailField　　字符串类型（正则表达式邮箱） =varchar

　　对字符串进行正则表达式

9、models.FloatField　　浮点类型 = double

10、models.IntegerField　　整形

11、models.BigIntegerField　　长整形

　　integer\_field\_ranges = {

　　　　'SmallIntegerField': (-32768, 32767),

　　　　'IntegerField': (-2147483648, 2147483647),

　　　　'BigIntegerField': (-9223372036854775808, 9223372036854775807),

　　　　'PositiveSmallIntegerField': (0, 32767),

　　　　'PositiveIntegerField': (0, 2147483647),

　　}

12、models.IPAddressField　　字符串类型（ip4正则表达式）

13、models.GenericIPAddressField　　字符串类型（ip4和ip6是可选的）

　　参数protocol可以是：both、ipv4、ipv6

　　验证时，会根据设置报错

14、models.NullBooleanField　　允许为空的布尔类型

15、models.PositiveIntegerFiel　　正Integer

16、models.PositiveSmallIntegerField　　正smallInteger

17、models.SlugField　　减号、下划线、字母、数字

18、models.SmallIntegerField　　数字

　　数据库中的字段有：tinyint、smallint、int、bigint

19、models.TextField　　字符串=longtext

20、models.TimeField　　时间 HH:MM[:ss[.uuuuuu]]

21、models.URLField　　字符串，地址正则表达式

22、models.BinaryField　　二进制

23、models.ImageField 图片

24、models.FilePathField 文件

字段相关参数：

1、null=True

　　数据库中字段是否可以为空

2、blank=True

　　django的 Admin 中添加数据时是否可允许空值

3、primary\_key = False

　　主键，对AutoField设置主键后，就会代替原来的自增 id 列

4、auto\_now 和 auto\_now\_add

　　auto\_now 自动创建---无论添加或修改，都是当前操作的时间

　　auto\_now\_add 自动创建---永远是创建时的时间

5、choices

GENDER\_CHOICE = (

(u'M', u'Male'),

(u'F', u'Female'),

)

gender = models.CharField(max\_length=2,choices = GENDER\_CHOICE)

6、max\_length

7、default　　默认值

8、verbose\_name　　Admin中字段的显示名称

9、name|db\_column　　数据库中的字段名称

10、unique=True　　不允许重复

11、db\_index = True　　数据库索引

12、editable=True　　在Admin里是否可编辑

13、error\_messages=None　　错误提示

14、auto\_created=False　　自动创建

15、help\_text　　在Admin中提示帮助信息

16、validators=[]

1. upload-to

连表关系：

* 一对多，models.ForeignKey(ColorDic)
* 一对一，models.OneToOneField(OneModel)
* 多对多，authors = models.ManyToManyField(Author)

应用场景：

* 一对一：在某表中创建一行数据时，有一个单选的下拉框（下拉框中的内容被用过一次就消失了）。 例如：原有含10列数据的一张表保存相关信息，经过一段时间之后，10列无法满足需求，需要为原来的表再添加5列数据。
* 一对多：当一张表中创建一行数据时，有一个单选的下拉框（可以被重复选择）。 例如：创建用户信息时候，需要选择一个用户类型【普通用户】【金牌用户】【铂金用户】等。
* 多对多：在某表中创建一行数据是，有一个可以多选的下拉框。 例如：创建用户信息，需要为用户指定多个爱好。

元选项

* 在模型类中定义类Meta，用于设置元信息
* ordering：对象的默认排序字段，获取对象的列表时使用，接收属性构成的列表

class BookInfo(models.Model):

...

class Meta():

ordering = ['id']

* 字符串前加-表示倒序，不加-表示正序

class BookInfo(models.Model):

...

class Meta():

ordering = ['-id']

* 排序会增加数据库的开销

#### 4.3.2.2settings.py相关配置

数据库连接配置：

[复制代码](javascript:void(0);)

DATABASES = {

'default': {

#'ENGINE': 'django.db.backends.', # Add 'postgresql\_psycopg2', 'mysql', 'sqlite3' or 'oracle'.

'ENGINE': 'django.db.backends.sqlite3', #添加数据库引擎；选项['postgresql\_psycopg2', 'mysql', 'sqlite3' or 'oracle'].

'NAME': 'F:/TestPython/blog/blog/db/data.db', # 数据库文件的路径.

# The following settings are not used with sqlite3:

# 下面的配置不适用于sqlite3:

'USER': '', # 数据库登陆用户名

'PASSWORD': '', # 数据库登陆密码

'HOST': '', # Empty for localhost through domain sockets or '127.0.0.1' for localhost through TCP. # 主机名

'PORT': '', # Set to empty string for default. # 端口号

}

}

举例：mysql

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'mysql\_user',

'USER': '用户名',

'PASSWORD': '密码',

'HOST': '数据库服务器ip，本地可以使用localhost',

'PORT': '端口，默认为3306',

}

}

4.3注册app：  
在settings里的INSTALLED\_APPS里面加上你的app名字

[复制代码](javascript:void(0);)

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'app01',

]

[复制代码](javascript:void(0);)

## 4.4生成数据库表

python manage.py makemigrations

# 生成一个数据库结构migrations里面一个表

python manage.py migrate

# 根据migrations里面的表来创建数据库

## 4.5将app的数据库注册到admin

在app01下面的admin.py写入以下内容：

[复制代码](javascript:void(0);)

from django.contrib import admin

# Register your models here.

# 导入app01的数据库模块

from app01 import models

# 注册咱们创建的类,通过他来访问

admin.site.register(models.UserInfo)

[复制代码](javascript:void(0);)

## 4.6数据库相关操作

增删改查

[复制代码](javascript:void(0);)

# 插入数据：

UserInfo.objects.create(name='rain',password='123',email='rain@163.com')

# 删除数据：

UserInfo.objects.filter(name='rain').delete()

# 修改数据：

UserInfo.objects.filter(name='rain').update(name='Rain1')

# 查找数据：

UserInfo.objects.filter(name='rain')

UserInfo.objects.all()

UserInfo.objects.all()[:10]

# 切片操作，获取10个人，不支持负索引，切片可以节约内存

UserInfo.objects.get(name='rain')

#查找所有

models.UserInfo.objects.all()

#查找指定条件

models.UserInfo.objects.filter(age=18)

#查找第一个

models.UserInfo.objects.filter(age=18).first()

#查找所有并且显示出来

user\_list\_obj = models.UserInfo.objects.all()

for line in user\_list\_obj:

print(line.username,line.age)

[复制代码](javascript:void(0);)

### 4.6.1Django默认数据库管理器

* objects：是Manager类型的对象，用于与数据库进行交互
* 当定义模型类时没有指定管理器，则Django会为模型类提供一个名为objects的管理器
* 支持明确指定模型类的管理器

class BookInfo(models.Model):

...

books = models.Manager()

* 当为模型类指定管理器后，django不再为模型类生成名为objects的默认管理器

### ４.6.２管理器Manager

* 管理器是Django的模型进行数据库的查询操作的接口，Django应用的每个模型都拥有至少一个管理器
* 自定义管理器类主要用于两种情况
* 情况一：向管理器类中添加额外的方法：见下面“创建对象”中的方式二
* 情况二：修改管理器返回的原始查询集：重写get\_queryset()方法

class BookInfoManager(models.Manager):

def get\_queryset(self):

return super(BookInfoManager, self).get\_queryset().filter(isDelete=False)

class BookInfo(models.Model):

...

books = BookInfoManager()

### ４.6.３创建对象

* 当创建对象时，django不会对数据库进行读写操作
* 调用save()方法才与数据库交互，将对象保存到数据库中
* 使用关键字参数构造模型对象很麻烦，但是不要重写init 方法，可以使用下面的两种之式
* 方式一：在模型类中增加一个类方法

class BookInfo(models.Model):

...

@classmethod

def create(cls, title, pub\_date):

book = cls(btitle=title, bpub\_date=pub\_date)

book.bread=0

book.bcommet=0

book.isDelete = False

return book

引入时间包：from datetime import \*

调用：book=BookInfo.create("hello",datetime(1980,10,11));

保存：book.save()

* 方式二：在自定义管理器中添加一个方法
* 在管理器的方法中，可以通过self.model来得到它所属的模型类

class BookInfoManager(models.Manager):

def create\_book(self, title, pub\_date):

book = self.model()

book.btitle = title

book.bpub\_date = pub\_date

book.bread=0

book.bcommet=0

book.isDelete = False

return book

class BookInfo(models.Model):

...

books = BookInfoManager()

调用：book=Book.books.create\_book("abc",datetime(1980,1,1))

保存：book.save()

* 在方式二中，可以调用self.create()创建并保存对象，不需要再手动save()

class BookInfoManager(models.Manager):

def create\_book(self, title, pub\_date):

book = self.create(btitle = title,bpub\_date = pub\_date,bread=0,bcommet=0,isDelete = False)

return book

class BookInfo(models.Model):

...

books = BookInfoManager()

调用：book=Book.books.create\_book("abc",datetime(1980,1,1))

查看：book.pk

### ４.6.４实例的属性

* DoesNotExist：在进行单个查询时，模型的对象不存在时会引发此异常，结合try/except使用

### ４.6.５实例的方法

* str (self)：重写object方法，此方法在将对象转换成字符串时会被调用
* save()：将模型对象保存到数据表中
* delete()：将模型对象从数据表中删除

### 4.6.7扩展

默认user表有时不满足要求，需要进行扩展user表，比如添加生日等，可以再官方文档进行查看扩展方法

方法：继承**from** django.contrib.auth.models **import** AbstractUser(就是数据库中user基本的本来的字段内容)

表的显示信息，表名和复数时的表名

**class** Meta:  
 verbose\_name = **'用户'** verbose\_name\_plural = verbose\_name  
 ordering = [**'id'**]

UserAdmin相对于基本的ModelAdmin

增加了fieldsets

add\_fieldsets

等字段

**from** django.contrib.auth.models **import** AbstractUser的使用：

步骤一：导入，之后在model中继承实现自己的用户类

步骤二：在setting中设置AUTH\_USER\_MODEL = 'myapp.MyUser'

上面的'myapp值表示Django应用的名称（必须位于INSTALLLED\_APPS中）和你MyUser'想使用的User模型的名称。不需要再setting中导入AbstractUser，否则容易报错

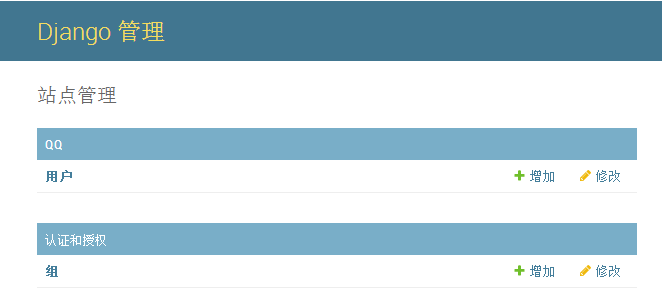
步骤三：在admin中注册，admin.site.register(models.Article,ArticleAdin)

否则不会在其中显示，



效果：按照上述操作完成后，效果如下：

新的用户类



自带的用户类：



# 第五章 Django模板系统

HTML是硬编码在Python代码中的，这会导致几个问题：  
1，显然，任何页面的改动会牵扯到Python代码的改动，网站的设计改动会比Python代码改动更频繁，所以如果我们将两者分离开会更方便  
2，其次，写后台Python代码与设计HTML是不同的工作，更专业的Web开发应该将两者分开，页面设计者和HTML/CSS程序员不应该编辑Python代码，他们应该与HTML打交道。  
3，程序员写Python代码同时页面设计者写HTML模板会更高效，而不是一个人等待另一个人编辑同样的文件  
因此，使用Django的模板系统分离设计和Python代码会更干净更易维护

## 5．1****模板系统基础****

Django模板是一个string文本，它用来分离一个文档的展现和数据，模板定义了placeholder和表示多种逻辑的tags来规定文档如何展现。通常模板用来输出HTML，但是Django模板也能生成其它基于文本的形式。  
让我们来看看一个简单的模板例子：

1. <html>
2. <head><title>Ordering notice</title></head>
3. <body>
4. <p>Dear {{ person\_name }},</p>
5. <p>Thanks **for** placing an order from {{ company }}. It's scheduled to
6. ship on {{ ship\_date|date:"F j, Y" }}.</p>
7. <p>Here are the items you've ordered:</p>
8. <ul>
9. {% **for** item in item\_list %}
10. <li>{{ item }}</li>
11. {% endfor %}
12. </ul>
13. {% **if** ordered\_warranty %}
14. <p>Your warranty information will be included in the packaging.</p>
15. {% endif %}
16. <p>Sincerely,<br />{{ company }}</p>
17. </body>
18. </html>

这个模板本质上是HTML，但是夹杂了一些变量和模板标签：  
1，用{{}}包围的是变量，如{{person\_name}}，这表示把给定变量的值插入，如何指定这些变量的值我们即将说明  
2，用{%%}包围的是块标签，如{%if ordered\_warranty%}  
块标签的含义很丰富，它告诉模板系统做一些事情  
在这个例子模板中包含两个块标签:for标签表现为一个简单的循环结构，让你按顺序遍历每条数据  
if标签则表现为逻辑的if语句  
在这里，上面的标签检查ordered\_warranty变量的值是否为True  
如果是True，模板系统会显示{%if ordered\_warranty%}和{%endif%}之间的内容  
否则，模板系统不会显示这些内容  
模板系统也支持{%else%}等其它逻辑语句  
3，上面还有一个过滤器的例子，过滤器是改变变量显示的方式  
上面的例子中{{ship\_date|date:"F j, Y"}}把ship\_date变量传递给过滤器  
并给date过滤器传递了一个参数“F j, Y”，date过滤器以给定参数的形式格式化日期  
类似于Unix，过滤器使用管道字符“|”  
4.Django模板支持多种内建的块标签，并且你可以写你自己的标签

## 5.2 ****使用模板系统****

在Python代码中使用模板系统，请按照下面的步骤：  
1，用模板代码创建一个Template对象  
Django也提供指定模板文件路径的方式创建Template对象  
2，使用一些给定变量context调用Template对象的render()方法  
这将返回一个完全渲染的模板，它是一个string，其中所有的变量和块标签都会根据context得到值

### **5.2.1创建模板对象**

最简单的方式是直接初始化它，Template类在django.template模块中，初始化方法需要一个参数  
下面进入Python交互环境看看：

1. >>> from django.template **import** Template
2. >>> t = Template("My name is {{my\_name}}.")
3. >>> print t

你将看到如下信息

1. <django.template.Template object at 0xb7d5f24c>

当你创建Template对象，模板系统会编译模板代码，并准备渲染  
如果你的模板代码有语法错误，调用Template()方法会触发TemplateSyntaxError异常

1. >>> from django.template **import** Template
2. >>> t = Template('{%notatag%}')
3. Traceback (most recent call last):
4. File "<stdin>", line 1, in ?
5. ...
6. django.template.TemplateSyntaxError: Invalid block tag: 'notatag'

系统触发TemplateSyntaxError异常可能出于以下情况：  
1，不合法的块标签  
2，合法块标签接受不合法的参数  
3，不合法的过滤器  
4，合法过滤器接受不合法的参数  
5，不合法的模板语法  
6，块标签没关

### 5.2.2****渲染模板****

一旦你拥有一个Template对象，你可以通过给一个context来给它传递数据  
context是一个变量及赋予的值的集合，模板使用它来得到变量的值，或者对于块标签求值  
这个context由django.template模块的Context类表示  
它的初始化函数有一个可选的参数：一个映射变量名和变量值的字典  
通过context调用Template对象的render()方法来填充模板，例如：

1. >>> from django.template **import** Context, Template
2. >>> t = Template("My name is {{name}}.")
3. >>> c = Context({"name": "Stephane"})
4. >>> t.render(c)
5. 'My name is Stephane.'

变量名必须以字母(A-Z或a-z)开始，可以包含数字，下划线和小数点，变量名大小写敏感  
下面是一个模板编译和渲染的例子，使用这章开始时的模板例子：

1. >>> from django.template **import** Template, Context
2. >>> raw\_template = """<p>Dear {{ person\_name }},</p>
3. ...
4. ... <p>Thanks **for** ordering {{ product }} from {{ company }}. It's scheduled to
5. ... ship on {{ ship\_date|date:"F j, Y" }}.</p>
6. ...
7. ... {% **if** ordered\_warranty %}
8. ... <p>Your warranty information will be included in the packaging.</p>
9. ... {% endif %}
10. ...
11. ... <p>Sincerely,<br />{{ company }}</p>"""
12. >>> t = Template(raw\_template)
13. >>> **import** datetime
14. >>> c = Context({'person\_name': 'John Smith',
15. ...     'product': 'Super Lawn Mower',
16. ...     'company': 'Outdoor Equipment',
17. ...     'ship\_date': datetime.date(2009, 4, 2),
18. ...     'ordered\_warranty': True})
19. >>> t.render(c)
20. "<p>Dear John Smith,</p>\n\n<p>Thanks **for** ordering Super Lawn Mower from Outdoor Equipment.
21. It's scheduled to ship on April 2, 2009.</p>\n\n<p>Your warranty information will be included
22. in the packaging.</p>\n\n\n<p>Sincerely,<br />Outdoor Equipment</p>"

让我们来看看都做了些什么：  
1，我们import Template和Context类，它们都在django.template模块里面  
2，我们把模板文本存储在raw\_template变量里，我们使用"""来构建string，它可以跨越多行  
3，我们创建模板对象t，并给Template类的初始化函数传递raw\_template变量  
4，我们从Python的标准库import datetime模块，下面会用到它  
5，我们创建一个context对象c，它的初始化函数使用一个映射变量名和变量值的字典  
例如我们指定person\_name的值为'John Smith'，product的值为'Super Lawn Mower'等等  
6，最后，我们调用模板对象的render()方法，参数为context对象c  
这将返回渲染后的模板，将模板中的变量值替换并计算块标签的结果  
如果你刚接触Python，你可能会问为什么输出中包含了新行字符'\n'而不是换行  
这是因为Python交互环境中调用t.render(c)会显示string的representation而不是string的值  
如果你想看到换行而不是'\n'，使用print t.render(c)即可  
上面是使用Django模板系统的基础，只是创建一个模板对象和context对象然后调用render()方法  
同一个模板，多个context的情况：  
一旦你创建了一个模板对象，你可以渲染多个context，例如：

1. >>> from django.template **import** Template, Context
2. >>> t = Template('Hello, {{ name }}')
3. >>> print t.render(Context({'name': 'John'}))
4. Hello, John
5. >>> print t.render(Context({'name': 'Julie'}))
6. Hello, Julie
7. >>> print t.render(Context({'name': 'Pat'}))
8. Hello, Pat

无论何时，你使用同一个模板来渲染多个context的话，创建一次Template对象然后调用render()多次会更高效

1. # Bad
2. **for** name in ('John', 'Julie', 'Pat'):
3. t = Template('Hello, {{ name }}')
4. print t.render(Context({'name'： name}))
5. # Good
6. t = Template('Hello, {{ name }}')
7. **for** name in ('John', 'Julie', 'Pat'):
8. print t.render(Context({'name': name}))

Django的模板解析非常快，在后台，大部分的解析通过一个单独的对正则表达式的调用来做＝

## 5.3****Context变量查找****

上面的例子中，我们给模板context传递了简单的值，大部分是string，以及一个datetime.date  
尽管如此，模板系统优雅的处理更复杂的数据结构，如列表，字典和自定义对象  
在Django模板系统中处理复杂数据结构的关键是使用(.)字符  
使用小数点来得到字典的key，属性，对象的索引和方法  
下面通过例子来解释，通过(.)访问字典的key：

1. >>> from django.template **import** Template, Context
2. >>> person = {'name': 'Sally', 'age': '43'}
3. >>> t = Template('{{ person.name }} is {{ person.age }} years old.')
4. >>> c= Context({'person': person})
5. >>> t.render(c)
6. 'Sally is 43 years old.'

通过(.)来访问对象的属性：

1. >>> from django.template **import** Template, Context
2. >>> **import** datetime
3. >>> d = datetime.date(1993, 5, 2)
4. >>> d.year
5. 1993
6. >>> d.month
7. 5
8. >>> d.day
9. 2
10. >>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
11. >>> c = Context({'date': d})
12. >>> t.render(c)
13. 'The month is 5 and the year is 1993.'

下面的例子使用一个自定义类：

1. >>> from django.template **import** Template, Context
2. >>> **class** Person(object):
3. ...    def \_\_init\_\_(self, first\_name, last\_name):
4. ...        self.first\_name, self.last\_name = first\_name, last\_name
5. >>> t = Template('Hello, {{ person.first\_name }} {{ person.last\_name }}.')
6. >>> c = Context({'person': Person('John', 'Smith')})
7. >>> t.render(c)
8. 'Hello, John Smith.'

小数点也可以用来调用列表的索引：

1. >>> from django.template **import** Template, Context
2. >>> t = Template('Item 2 is {{ items.2 }}.')
3. >>> c = Contexst({'items': ['apples', 'bananas', 'carrots']})
4. >>> t.render(c)
5. 'Item 2 is carrots.'

负数的列表索引是不允许的，例如模板变量{{ items.-1 }}将触发TemplateSyntaxError  
最后小数点还可以用来访问对象的方法，例如Python的string有upper()和isdigit()方法：

1. >>> from django.template **import** Template, Context
2. >>> t = Template('{{ var }} -- {{var.upper }} -- {{ var.isdigit }}')
3. >>> t.render(Context({'var': 'hello'}))
4. 'hello -- HELLO -- False'
5. >>> t.render(Context({'var': '123'}))
6. '123 - 123 - True'

注意，调用方法时你不能加括号，你也不能给方法传递参数  
你只能调用没有参数的方法，后面我们会解释这些  
总结一下，当模板系统遇到变量名里有小数点时会按以下顺序查找：  
1，字典查找，如foo["bar"]  
2，属性查找，如foo.bar  
3，方法调用，如foo.bar()  
3，列表的索引查找，如foo[bar]  
小数点可以多级纵深查询，例如{{ person.name.upper }}表示字典查询person['name']然后调用upper()方法

1. >>> from django.template **import** Template, Context
2. >>> person = {'name': 'Sally', 'age': '43'}
3. >>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
4. >>> c = Context({'person': person})
5. >>> t.render(c)
6. 'SALLY is 43 years old.'

## 5.4****不合法的变量怎样处理****

默认情况下如果变量不存在，模板系统会把它渲染成空string，例如：

1. >>> from django.template **import** Template, Context
2. >>> t = Template('Your name is {{ name }}.')
3. >>> t.render(Context())
4. 'Your name is .'
5. >>> t.render(Context({'var': 'hello'}))
6. 'Your name is .'
7. >>> t.render(Context({'NAME': 'hello'}))
8. 'Your name is .'
9. >>> t.render(Context({'Name': 'hello'}))
10. 'Your name is .'

**模板标签和过滤器基础**  
我们已经提到模板系统使用内建的标签和过滤器  
这里我们看看常见的，附录6包含了完整的内建标签和过滤器，你自己熟悉那个列表来了解可以做什么是个好主意  
  
**if/else**  
{% if %}标签计算一个变量值，如果是“true”，即它存在、不为空并且不是false的boolean值  
系统则会显示{% if %}和{% endif %}间的所有内容：

1. {% **if** today\_is\_weekend %}
2. <p>Welcome to the weekend!</p>
3. {% **else** %}
4. <p>Get back to work.</p>
5. {% endif %}

{% if %}标签接受and，or或者not来测试多个变量值或者否定一个给定的变量，例如：

1. {% **if** athlete\_list and coach\_list %}
2. Both athletes and coaches are available.
3. {% endif %}
4. {% **if** not athlete\_list %}
5. There are no athletes.
6. {% endif %}
7. {% **if** athlete\_list or coach\_list %}
8. There are some athletes or some coaches.
9. {% endif %}
10. {% **if** not athlete\_list or coach\_list %}
11. There are no athletes or there are some coaches.
12. {% endif %}
13. {% **if** athlete\_list and not coach\_list %}
14. There are some athletes and absolutely no coaches.
15. {% endif %}

{% if %}标签不允许同一标签里同时出现and和or，否则逻辑容易产生歧义，例如下面的标签是不合法的：

1. {% **if** athlete\_list and coach\_list or cheerleader\_list %}

如果你想结合and和or来做高级逻辑，只需使用嵌套的{% if %}标签即可：

1. {% **if** athlete\_list %}
2. {% **if** coach\_list or cheerleader\_list %}
3. We have athletes, and either coaches or cheerleaders!
4. {% endif %}
5. {% endif %}

多次使用同一个逻辑符号是合法的：

1. {% **if** athlete\_list or coach\_list or parent\_list or teacher\_list %}

没有{% elif %}标签，使用嵌套的{% if %}标签可以做到同样的事情：

1. {% **if** athlete\_list %}
2. <p>Here are the athletes: {{ athlete\_list }}.</p>
3. {% **else** %}
4. <p>No athletes are available.</p>
5. {% **if** coach\_list %}
6. <p>Here are the coaches: {{ coach\_list }}.</p>
7. {% endif %}
8. {% endif %}

确认使用{% endif %}来关闭{% if %}标签，否则Django触发TemplateSyntaxError  
  
**for**  
{% for %}标签允许你按顺序遍历一个序列中的各个元素  
Python的for语句语法为for X in Y，X是用来遍历Y的变量  
每次循环模板系统都会渲染{% for %}和{% endfor %}之间的所有内容  
例如，显示给定athlete\_list变量来显示athlete列表：

1. <ul>
2. {% **for** athlete in athlete\_list %}
3. <li>{{ athlete.name }}</li>
4. {% endfor %}
5. </ul>

在标签里添加reversed来反序循环列表：

1. {% **for** athlete in athlete\_list reversed %}
2. ...
3. {% endfor %}
4. {% **for** %}标签可以嵌套：
5. {% **for** country in countries %}
6. <h1>{{ country.name }}</h1>
7. <ul>
8. {% **for** city in country.city\_list %}
9. <li>{{ city }}</li>
10. {% endfor %}
11. </ul>
12. {% endfor %}

系统不支持中断循环，如果你想这样，你可以改变你想遍历的变量来使得变量只包含你想遍历的值  
类似的，系统也不支持continue语句，本章后面的“哲学和限制”会解释设计的原则  
{% for %}标签内置了一个forloop模板变量，这个变量含有一些属性可以提供给你一些关于循环的信息  
1，forloop.counter表示循环的次数，它从1开始计数，第一次循环设为1，例如：

1. {% **for** item in todo\_list %}
2. <p>{{ forloop.counter }}: {{ item }}</p>
3. {% endfor %}

2，forloop.counter0类似于forloop.counter，但它是从0开始计数，第一次循环设为0  
3，forloop.revcounter表示循环中剩下的items数量，第一次循环时设为items总数，最后一次设为1  
4，forloop.revcounter0类似于forloop.revcounter，但它是表示的数量少一个，即最后一次循环时设为0  
5，forloop.first当第一次循环时值为True，在特别情况下很有用：

1. {% **for** object in objects %}
2. {% **if** forloop.first %}<li **class**="first">{% **else** %}<li>{% endif %}
3. {{ object }}
4. </li>
5. {% endfor %}

6，forloop.last当最后一次循环时值为True

1. {% **for** link in links %}{{ link }}{% **if** not forloop.last %} | {% endif %}{% endfor %}

7，forloop.parentloop在嵌套循环中表示父循环的forloop：

1. {% **for** country in countries %}
2. <table>
3. {% **for** city in country.city\_list %}
4. <tr>
5. <td>Country #{{ forloop.parentloop.counter }} </td>
6. <td>City #{{ forloop.counter }}</td>
7. <td>{{ city }}</td>
8. </tr>
9. {% endfor %}
10. </table>
11. {% endfor %}

富有魔力的forloop变量只能在循环中得到，当模板解析器到达{% endfor %}时forloop就消失了  
如果你的模板context已经包含一个叫forloop的变量，Django会用{% for %}标签替代它  
Django会在for标签的块中覆盖你定义的forloop变量的值  
在其他非循环的地方，你的forloop变量仍然可用  
我们建议模板变量不要使用forloop，如果你需要这样做来访问你自定义的forloop，你可以使用forloop.parentloop  
  
**ifequal/ifnotequal**  
Django模板系统并不是一个严格意义上的编程语言，所以它并不允许我们执行Python语句  
（我们会在‘哲学和限制‘一节详细讨论）。  
然而在模板语言里比较两个值并且在他们一致的时候显示一些内容，确实是一个在常见不过的需求了——所以Django提供了ifequal标签。  
{% ifequal %}比较两个值，如果相等，则显示{% ifequal %}和{% endifequal %}之间的所有内容：

1. {% ifequal user currentuser %}
2. <h1>Welcome!</h1>
3. {% endifequal %}

参数可以是硬编码的string，单引号和双引号均可，下面的代码是合法的：

1. {% ifequal section 'sitenews' %}
2. <h1>Site News</h1>
3. {% endifequal %}
4. {% ifequal section "community" %}
5. <h1>Community</h1>
6. {% endifequal %}

和{% if %}一样，{% ifequal %}标签支持{% else %}

1. {% ifequal section 'sitenews' %}
2. <h1>Site News</h1>
3. {% **else** %}
4. <h1>No News Here</h1>
5. {% endifequal %}

其它的模板变量，strings，integers和小数都可以作为{% ifequal %}的参数：

1. {% ifequal variable 1 %}
2. {% ifequal variable 1.23 %}
3. {% ifequal variable 'foo' %}
4. {% ifequal variable "foo" %}

其它的Python类型，如字典、列表或booleans不能硬编码在{% ifequal %}里面，下面是不合法的：

1. {% ifequal variable True %}
2. {% ifequal variable [1, 2, 3,]%}
3. {% ifequal variable {'key': 'value'} %

如果你需要测试某个变量是true或false，用{% if %}即可  
  
**注释**  
和HTML或编程语言如Python一样，Django模板语言允许注释{# #}，如：

1. {# This is a comment #}

模板渲染时注释不会输出，一个注释不能分成多行  
下面的模板渲染时会和模板中的内容一样，注释标签不会解析成注释  
This is a {# comment goes here  
and spans another line #}  
test.  
  
**过滤器**  
本章前面提到，模板过滤器是变量显示前转换它们的值的方式，看起来像下面这样：

1. {{ name|lower }}

这将显示通过lower过滤器过滤后{{ name }}变量的值，它将文本转换成小写  
使用(|)管道来申请一个过滤器  
过滤器可以串成链，即一个过滤器的结果可以传向下一个  
下面是escape文本内容然后把换行转换成p标签的习惯用法：

1. {{ my\_text|escape|linebreaks }}

有些过滤器需要参数，需要参数的过滤器的样子：

1. {{ bio|truncatewords:"30" }}

这将显示bio标量的前30个字，过滤器参数一直使用双引号  
下面是一些最重要的过滤器：  
1，addslashed，在任何后斜线，单引号，双引号前添加一个后斜线  
当你把一些文本输出到一个JavaScript字符串时这会十分有用  
2，date，根据一个格式化string参数来格式化date或datetime对象，例如：

1. {{ pub\_date|date:"F j, Y" }}

格式化string会在附录6定义  
3，escape，避免给定的string里出现and符，引号，尖括号  
当你处理用户提交的数据和确认合法的XML和XHTML数据时这将很有用  
escape将作如下的一些转换：

1. Converts & to &amp;
2. Converts < to &lt;
3. Converts > to &gt;
4. Converts "(双引号) to &quot;
5. Converts '(单引号) to &#39;

4，length，返回值的长度，你可以在一个list或string上做此操作  
或者在任何知道怎样决定自己的长度的Python对象上做此操作(即有一个\_\_len\_\_()方法的对象)  
  
**哲学和限制**  
现在我们对于Django地模板系统有了一个感性的认识，下面我们将指出一些有意为之的限制和它工作的哲学  
不像其他Web程序组件，程序员对模板系统的意见非常不一致  
一个很有意思的事实：Python至少拥有数十个——如果没有上百个——的开源模板语言实现，而且看来每一个都是因为其创造者认为现有的模板不能满足他们的要求。  
（事实上，据说写一个自己的模板系统是已经成了Python开发者必经的仪式了。如果你还没有写过自己的模板系统，试试看吧，真是很有意思。）  
所以，Django的第一个哲学就是Django不强求你使用它的模板语言  
Django的目标是提供一个full-stack框架，提供所有必须的web开发模块进行高效开发  
很多时候，使用Django的模板系统很方便，但不强求你使用它  
下面的“在视图中使用模板”一节我们会看到在Django中使用另一套模板语言，它同样简单易用  
但我们仍强烈需要Django的模板语言的工作方式，模板系统深植于World Online和Django发明者的  
Web开发方式中，下面是其中一些哲学：  
1，业务逻辑应该和呈现逻辑分离  
模板系统应该只负责控制显示和显示相关的逻辑我们视模板为一种控制显示和显示相关逻辑的工具，仅此而已。模板系统的功能就止于此。  
基于这个原因，Django模板无法直接调用Python代码。在Django模板里，所有的程序设计活动都止于对标签的使用。  
虽然你可以自定义模板标签来做任意的事情，但Django自己的模板标签不允许执行Python代码。  
2，语法应该和HTML/XML解耦  
Django的模板系统采用非HTML格式，如普通的文本，有些其它的模板语言是基于XML的  
XML的格式容易输错，并且XML的模板解析速度也容易变得很慢而难以接受  
3，页面设计者被假定为熟悉HTML代码  
Django模板系统没有设计成可以在Dreamweaver等WYSISYG编辑器中显示良好  
这类编辑器有很多限制，Django希望模板作者直接编辑HTML时感到舒适  
4，页面设计者被假定为不是Python程序员  
模板系统的作者意识到大部分Web页面的模板是页面设计者写的而不是Python程序员写的  
他们不具备Python知识，但Django也允许模板用Python来写，它提供了一种直接编写Python代码  
来扩展模板系统的方法(第10章会介绍更多)  
5，目标不是发明一种编程语言  
目标只是提供足够的编程功能，如分支和循环等决定呈现相关的逻辑用  
由于上述的设计哲学，Django模板系统产生如下限制：  
1，模板不能设置和改变变量的值  
可以通过自定义模板标签来达到这个目标(I参看第10章)，但是内置Django模板标签不允许这样做  
2，模板不能调用原生Python代码  
但是也可以通过自定义标签来做这件事情  
  
**在视图里使用模板**  
我们已经学习了使用模板系统的基础，现在我们来在前一章中的current\_datetime视图中使用它：

1. from django.http **import** HttpResponse
2. **import** datetime
4. def current\_datetime(request):
5. now = datetime.datetime.now()
6. html = "<html><body>It is now %s.</body></html>" % now
7. **return** HttpResponse(html)

让我们把这个试图改成Django模板系统的做法，首先你可能想这样做：

1. from django.template **import** Template, Context
2. from django.http **import** HttpResponse
3. **import** datetime
5. def current\_datetime(request):
6. now = datetime.datetime.now()
7. t = Template("<html><body>It is now {{ current\_date }}.</body></html>")
8. html = t.render(Context({'current\_date': now}))
9. **return** HttpResponse(html)

这当然用到了模板系统，但它并没有解决我们本章开始介绍的问题，模板仍然嵌在Python代码里面  
让我们通过把模板放在一个单独的文件里来解决它，一个简陋的方式就是把模板保存在文件系统中然后使用Python内建的文件读取功能得到模板的内容，下面来看看这样做的例子：

1. from django.template **import** Template, Context
2. from django.http **import** HttpResponse
3. **import** datetime
5. def current\_datetime(request):
6. now = datetime.datetime.now()
7. # Simple, "dumb" way of saving templates on the filesystem.
8. # This doesn't account **for** missing files!
9. fp = open('/home/djangouser/templates/mytemplate.html')
10. t = Template(fp.read())
11. fp.close()
12. html = t.render(Context({'current\_date': now}))
13. **return** HttpResponse(html)

这种方式非常不优雅“  
1，它不会处理丢失的文件，如果mytemplate.html不存在或者不可读，调用open()将触发IOError异常  
2，它硬编码了你的模板位置，如果你使用这个技术来处理每个视图方法，你就会重复复制模板的位置  
3，它引入了许多无聊代码，调用open(),fp.reand()和fp.close()需要很多输入而且毫无创造性  
为了解决这个问题，我们将使用模板载入和模板目录  
  
**模板载入**  
Django提供了方便和强大的API来从硬盘载入模板，从而减少调用模板和模板本身的冗余  
为了使用Django的模板载入API，首先你需要在settings文件里告诉Django你把模板放在哪  
Django的settings文件时存放你的Django实例的配置的地方，它是一个简单的具有  
模块级变量的Python模块，其中每个设置都是一个变量  
当你运行django-admin.py startproject mysite时脚本会为你创建一个默认的settings文件settings.py  
看看这个文件的内容，它包含了像下面这样的变量：

1. DEBUG = True
2. TIME\_ZONE = 'America/Chicago'
3. USE\_I18N = True
4. ROOT\_URLCONF = 'mysite.urls'

它把自己解释的很清楚，这些设置和对应的值是简单的Python变量  
由于settings文件仅仅是一个普通的Python模块，你可以在设置新变量前做类似于检查某个变量的值等动态的事情，这将避免你的settings文件出现Python语法错误  
这也意味着你应该避免在settings文件里面出现Python的语法错误  
后面我们会深入讲解settings文件，现在先来看看TEMPLATE\_DIRS设置，它告诉Django的模板载入机制在哪里寻找模板  
默认情况下它是一个空的元组，选择一个你喜欢的存放模板的地方并添加到TEMPLATE\_DIRS中去：

1. TEMPLATE\_DIRS = (
2. '/home/django/mysite/templates',
3. )

需要注意的一些事情：  
1，你可以指定任何目录，只要那个目录下的目录和模板对于你的Web服务器运行时的用户是可读的  
如果你找不到一个放置模板的位置，我们推荐你在Django工程目录下创建一个templates目录  
2，不要忘了模板目录最后的逗号，Python需要逗号来区分单元素元组和括号括起来的语句  
这是新手经常犯的错误，如果你想避免这个错误，可以用列表来替代元组，单元素列表不需要结尾的逗号

1. TEMPLATE\_DIRS = [
2. '/home/django/mysite/templates'
3. ]

元组比列表略微高效，所以我们推荐使用元组  
3，使用绝对路径很简单，如果你想更灵活和松耦合，你可利用Django的settings文件是简单的Python代码  
这点来动态构建TEMPLATE\_DIRS内容，例如：

1. **import** os.path
3. TEMPLATE\_DIRS = (
4. os.path.join(os.path.dirname(\_\_file\_\_), 'templates'),
5. )

这个例子使用了富有魔力的Python变量\_\_file\_\_，它会被自动设成当前代码所在的Python模块的文件名  
4，如果你使用Windows，加上硬盘号并使用Unix风格的前斜线而不是后斜线，例如：

1. TEMPLATE\_DIRS = (
2. 'C:/www/django/templates',
3. )

设置好TEMPLATE\_DIRS，下一步就是使用Django的模板载入功能而不是硬编码模板路径来测试代码  
让我们回到current\_datetime视图看看：

1. from django.template.loader **import** get\_template
2. from django.template **import** Context
3. from django.http **import** HttpResponse
4. **import** datetime
6. def current\_datetime(request):
7. now = datetime.datetime.now()
8. t = get\_template('current\_datetime.html')
9. html = t.render(Context({'current\_date': now}))
10. **return** HttpResponse(html)

这个例子中我们使用了django.template.loarder.get\_template()方法而不是从文件系统手动载入模板  
get\_template()方法使用模板名作为参数，算出模板在文件系统的什么地方，打开它并返回编译好的Template对象  
如果get\_template()方法不能找到给定名字的模板，它将触发TemplateDoesNotExist异常  
为了看看到底是什么样子，启动Djang server，打开浏览器访问[http://127.0.0.1:8000/now/](http://127.0.0.1:8000/now/" \t "_blank)  
假设你的DEBUG设为True并且你没有创建current\_datetime.html模板，你将看到一个高亮显示  
TemplateDoesNotExist异常的出错页面  
出错页面和第3章那个很类似，但它还有一个“Template-loader postmortem”部分  
这个部分告诉你Django尝试载入哪个模板以及每个尝试失败了的原因(如“File does not exist”)  
当你尝试debug模板载入错误时这些信息是非常有价值的  
如同你能在出错信息中看到的一样，Django试图把TEMPLATE\_DIRS中设置的值和传入get\_template()方法的模板名字组合起来查找模板文件。  
如果你的TEMPLATE\_DIRS中包含'/home/django/templates'，最后查找到的文件可能像这样：'/home/django/templates/current\_datetime.html.'  
接下来，在你的模板目录下创建current\_datetime.html文件并使用如下的模板代码：

1. <html><body>It is now {{ current\_date }}.</body></html>

刷新浏览器页面你将看到完整渲染的页面  
  
**render\_to\_response()**  
Django提供了一个捷径来使用一行代码完成载入模板，填充Context，渲染模板，返回HttpResponse对象的工作  
这就是render\_to\_response()，它在django.shortcuts模块下  
大部分情况下，你都会使用render\_to\_response()而不是手动完成上述的事情  
下面是利用render\_to\_response()把current\_datetime重写后的例子：

1. from django.shortcuts **import** render\_to\_response
2. **import** datetime
4. def current\_datetime(request):
5. now = datetime.datetime.now()
6. **return** render\_to\_response('current\_datetime.html', {'current\_date': now})

多么不同啊!我们来看看这些代码:  
1，我们不在import get\_template,Template,Context或者HttpResponse  
相反，我们import django.shortcuts.render\_to\_response，import datetime仍然存在  
2，使用current\_datetime方法，我们仍然计算now，但载入模板，创建context，渲染模板和  
创建HttpResponse全部被render\_to\_response()替换，render\_to\_response返回HttpResponse对象  
render\_to\_response()的第一个参数应该是使用的模板名，对应到模板目录的相对路径  
第二个参数如果有的话应该是一个用来创建Context的字典  
如果你不提供第二个参数，render\_to\_response()将使用一个空的字典  
  
**locals()小技巧**  
看看最近的current\_datetime：

1. def current\_datetime(request):
2. now = datetime.datetime.now()
3. **return** render\_to\_response('current\_datetime.html', {'current\_date': now})

这个例子中你会发现你自己计算一些值后存储在变量中(例如now)并传递给模板  
懒程序员可能会觉得有点繁琐，既要给临时变量取名又要给模板变量取名  
这不仅仅是冗余，这是过度输入  
如果你很懒或者你想保持代码整洁，使用Python内建的locals()方法  
locals()返回一个包含当前作用域里面的所有变量和它们的值的字典，上面的代码可以重写：

1. def current\_datetime(request):
2. current\_date = datetime.datetime.now()
3. **return** render\_to\_response('current\_datetime.html', locals())

这里我们传递locals()的值而不是手动指定context字典，locals()包含了所有定义在当前方法的变量  
而且，我们把now变量重命名为current\_date，因为模板需要的是这个变量名  
这个例子中locals()不会给你太大改善，但这个技术可以帮你少敲键盘  
使用locals()需要注意的是它包含了所有当前变量，可能包括比你的模板想访问的更多的变量  
上面的例子中，locals()也包括request变量，这依赖于你的程序  
最后要注意的是locals()导致了一点点开销，因为Python不得不动态创建字典  
如果你手动指定context字典则可以避免这项开销

**include模板标签**  
我们已经学习了模板载入机制，我们要介绍一个利用这个机制的内建标签:{% include %}  
这个标签允许你引入另一个模板的内容，标签的参数是你想引入的模板的名字，名字可以是变量，  
也可以是单引号或双引号表示的string  
下面两个例子引入了模板nav.html的内容，这表示单引号和双引号都是允许的：

1. {% include 'nav.html' %}
2. {% include "nav.html" %}

下面的例子引入了includes/nav.html模板：

1. {% include 'includes/nav.html' %}

下面的例子引入了一个名字存在于template\_name变量中的模板：

1. {% include template\_name %}

和get\_template()一样，请求的模板名前面会加上TEMPLATE\_DIRS  
如果被引入的模板中包含任何的模板代码，如标签和变量等，它将用父模板的context计算它们  
如果给定的模板名不存在，Django将做下面两件事情中的一件：  
1，如果DEBUG设置为True，你将看到一个TemplateDoesNotExist异常的错误页面  
2，如果DEBUG设置为False，标签将什么也不显示  
  
**模板继承**  
我们的模板例子现在还是HTML片断，但是真实世界你将使用Django模板系统输出完整的HTML页面  
这将导致常见的Web开发问题：怎样减少一个常见页面区域的重复和冗余(如全站导航)？  
解决这个问题的经典方式是使用服务器端引入和导向，你可以在你的HTML里面嵌套另一个页面  
Django确实也支持这种方式，上面介绍的{% include %}模板标签就是这种方案  
但是解决这个问题的更好的方式是Django的更优雅的方式模板继承  
本质上来说，模板继承使你能够构建一个“骨架”模板，里面包含你的网站的通用部分，并且在里面  
定义子模板可以覆盖的“块”，让我们看看前面的例子，编辑current\_datetime.html文件：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>The current time</title>
5. </head>
6. <body>
7. <h1>My helpful timestamp site</h1>
8. <p>It is now {{ current\_date }}.</p>
10. <hr>
11. <p>Thanks **for** visiting my site.</p>
12. </body>
13. </html>

看起来不错，但是当我们为另一个视图创建另一个模板时(如hours\_ahead视图)，如果我们想再创建  
一个完整的合法的HTML模板，我们将创建下面的内容：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>Future time</title>
5. </head>
6. <body>
7. <h1>My helpful timestamp site</h1>
8. <p>In {{ hour\_offset }} hour(s), it will be {{ next\_time }}.</p>
10. <hr>
11. <p>Thanks **for** visiting my site.</p>
12. </body>
13. </html>

显然我们重复了很多HTML内容，想象一下，如果我们在每个页面都有一些样式表，导航条，JavaScript...  
我们将会在每个模板加入重复的HTML内容  
这个问题的服务器端解决方案是取出模板中通用的部分然后存放在一个单独的模板中，然后被每个模板引入  
可能你会把它们存放在一个叫header.html中：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>

可能还需把底下的东西存在一个叫footer.html的文件中：  
<hr>  
    <p>Thanks for visiting my site.</p>  
</body>  
</html>  
使用基于引入的策略，头和尾很容易，但是中间的东西就很混乱  
例子中，每个页面有一个title

1. <h1>My helpful timestamp site</h1>

但是title不能放到hear.html中，因为每个页面中的title是不同的  
Django的模板继承系统解决了这种问题，你可以认为它是服务器引入的“相反”版本  
我们定义不同的部分而不是定义相同的部分  
第一步是建立基本模板，即你的子模板的框架，下面是一个基本模板的例子：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>{% block title %}{% endblock %}</title>
5. </head>
6. <body>
7. <h1>My helpful timestamp site</h1>
8. {% block content %}{% endblock %}
9. {% block footer %}
10. <hr>
11. <p>Thanks **for** visiting my site.</p>
12. {% endblock %}
13. </body>
14. </html>

我们把这个模板叫做base.html，它定义了我们用来在其它页面使用的基本HTML框架  
现在就是子模板覆盖的工作了，要么添加内容，要么不改变块的内容  
(如果你在照着例子做，把base.html保存到模板目录下)  
这里我们使用了{% block %}模板标签，它告诉模板引擎一个子模板可能覆盖模板的这部分内容  
既然我们有了基本模板，下面我们来编辑current\_datetme.html来使用它：

1. {% **extends** "base.html" %}
3. {% block title %}The current time{% endblock %}
5. {% block content %}
6. <p>It is now {{ current\_date }}.</p>
7. {% endblock %}

同时我们也创建一个hours\_ahead模板来使用基本模板：

1. {% **extends** "base.html" %}
3. {% block title %}Future time{% endblock %}
5. {% block content %}
6. <p>In {{ hour\_offset }} hour(s), it will be {{ next\_time }}.</p>
7. {% endblock %}

这样是不是更美观一些？每个模板只包含属于自己的代码，根本没有冗余  
如果你想做整个网站的改动，只需要更改base.html即可，其它的模板也会立即响应改动  
它是这样工作的：  
1，当你载入模板current\_datetime.html，模板引擎发现{% extends %}标签，注意到这是一个子模板  
模板引擎马上载入父模板base.html  
2，模板引擎在base.html里发现了3个{% block %}标签，就用子模板的内容替换了这些块  
于是我们定义的{% block title %}和{% block content %}被使用  
注意，既然子模板没有定义footer块，那么模板系统直接使用父模板的值  
父模板里{% block %}标签的内容一直可以作为后援方案  
你可以使用任意等级的继承，使用继承的常用方式是按以下三个步骤：  
1，创建base.html模板来掌控你的网站的整体外观，它的内容很少改变  
2，为你的网站创建base\_SECTION.html模板，例如，base\_photos.html，base\_forum.html  
这些模板继承base.html并且包括部分专有的风格和设计  
3，为每个类别的页面创建单独的模板，例如论坛页面护着照片图库页面  
这些模板继承相应的部分模板  
这个方法最大化了代码重用并且很容易向公用区域添加东西，例如部分专有的导航  
下面是一些关于模板继承的小提示：  
1，如果在模板里使用{% extends %}的话，这个标签必须在所有模板标签的最前面，否则模板继承不工作  
2，通常基本模板里的{% block %}越多越好，子模板不必定义所有的父block，钩子越多越好  
3，如果你在很多模板里复制代码，很可能你应该把这些代码移动到父模板里  
4，如果你需要得到父模板的块内容，{{ block.super }}变量可以帮你完成工作  
当你需要给父块添加内容而不是取代它的时候这就很有用  
5，不能在同一模板里定义多个同名的{% block %}，因为块标签同时在两个地方工作，不仅仅  
在子模板中，而且在父模板中也填充内容，如果子模板有两个同名的标签，父模板将不能决定  
使用哪个块内容来使用  
6，你给{% extends %}传递的模板名同样会被get\_template()使用，所以会加上TEMPLATE\_DIRS设置  
7，大部分情况下，{% extends %}的参数是string，但是也可以是变量，如果知道运行时才知道  
父模板的名字，这可以帮助你做一些很cool的动态内容  
  
**练习**  
下面是一些巩固你所学本章知识的练习，这里我们介绍了一些新的技巧  
1，你有一个音乐家和他们的音乐的列表，它们存储在一个字典的列表里，并且硬编码在你的视图模块  
(通常我们使用数据库来存放这些数据，但是目前我们还没讲到Django的数据库层)，列表如下：

1. MUSICIANS = [
2. {'name': 'Django Reinhardt', 'genre': 'jazz'},
3. {'name': 'Jimi Hendrix',     'genre': 'rock'},
4. {'name': 'Louis Armstrong',  'genre': 'jazz'},
5. {'name': 'Pete Townsend',    'genre': 'rock'},
6. {'name': 'Yanni',            'genre': 'new age'},
7. {'name': 'Ella Fitzgerald',  'genre': 'jazz'},
8. {'name': 'Wesley Willis',    'genre': 'casio'},
9. {'name': 'John Lennon',      'genre': 'rock'},
10. {'name': 'Bono',             'genre': 'rock'},
11. {'name': 'Garth Brooks',     'genre': 'country'},
12. {'name': 'Duke Ellington',   'genre': 'jazz'},
13. {'name': 'William Shatner',  'genre': 'spoken word'},
14. {'name': 'Madonna',          'genre': 'pop'},
15. ]

写一个Django视图来显示HTML的table，列表中的每个音乐家按顺序显示为一行  
每行有两列，分别显示音乐家名字和他的音乐  
2，一旦完成上述任务，把table中音乐是jazz或者rock的音乐家的名字样式设为粗体  
使用style="font-weight: bold;"来修饰td格  
3，一旦完成上述任务：给名字为一个字的音乐家的名字后加上星号  
并且在页面上添加脚注“\* Pretentious”前面的粗体字不变  
4，下面有3个模板，请你设计继承关系并且尽可能多的去除冗余  
模板1：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <link rel="stylesheet" href="default.css" type="text/css">
5. <title>My to-**do** list</title>
6. </head>
7. <body>
8. <h1 id="top">Latest tasks</h1>
9. {% **if** task\_list %}
10. <ul>
11. {% **for** task in task\_list %}<li>{{ task }}</li>{% endfor %}
12. </ul>
13. {% **else** %}
14. <p>You have no tasks.</p>
15. {% endif %}
16. <hr>
17. <p><a href="#top">Back to top</a>.</p>
18. </body>
19. </html>

模板2：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>Task: {{ task.title }} | To-**do** list</title>
5. <link rel="stylesheet" href="default.css" type="text/css">
6. </head>
7. <body>
8. <h1 id="top">{{ task.title }}</h1>
9. <p>{{ task.description }}</p>
10. <hr>
11. <p><a href="#top">Back to top</a>.</p>
12. </body>
13. </html>

模板3：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>Completed tasks | To-**do** list</title>
5. <link rel="stylesheet" href="default.css" type="text/css">
6. <script type="text/javascript" src="completed.js">
7. </head>
8. <body>
9. <h1 id="top">{{ task.title }}</h1>
10. <p>{{ task.description }}</p>
11. <hr>
12. <p><a href="#top">Back to top</a>.</p>
13. </body>
14. </html>

**练习答案**  
1，下面是一个可能的视图实现：

1. from django.shortcuts **import** render\_to\_response
3. MUSICIANS = [
4. # ...
5. ]
7. def musician\_list(request):
8. **return** render\_to\_response('musician\_list.html', {'musicians': MUSICIANS})

以及模板：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <title>Musician list</title>
5. </head>
6. <body>
7. <table>
8. <tr><th>Musician</th><th>Genre</th></tr>
9. {% **for** musician in musicians %}
10. <tr>
11. <td>{{ musician.name }}</td>
12. <td>{{ musician.genre }}</td>
13. </tr>
14. {% endfor %}
15. </table>
16. </body>
17. </html>

2，笨拙的方式是使用在模板中使用{% ifequal %}，视图和上面的保持不变，模板如下：

1. {% **for** musician in musicians %}
2. <tr>
3. <td {% ifequal musician.genre 'jazz' %}style="font-weight: bold;"{% endifequal %}
4. {% ifequal musician.genre 'rock' %}style="font-weight: bold;"{% endifequal %}>
5. {{ musician.name }}
6. </td>
7. <td>{{ musician.genre }}</td>
8. </tr>
9. {% endfor %}

这显得很罗嗦而且容易出错，Django模板系统的关键是知道显示什么  
因为模板没有完备的编程语言环境的能力，在视图里做尽可能多的业务逻辑更重要  
这样一来，更清晰的解决问题的方式就是预处理音乐家的名字是否粗体  
毕竟这是业务逻辑而不是呈现逻辑，呈现逻辑指出怎样显示特殊的类别而不是决定哪些类别是特殊的  
这是很重要的区别，下面是视图的代码：

1. def musician\_list(request):
2. musicians = []
3. **for** m in MUSICIANS:
4. musicians.append({
5. 'name': m['name'],
6. 'genre': m['genre'],
7. 'is\_important': m['genre'] in ('rock', 'jazz'),
8. })
9. **return** render\_to\_response('musician\_list.html', {'musicians': musicians})

然后这样使用模板代码：

1. {% **for** musician in musicians %}
2. <tr>
3. <td{% **if** musician.is\_important %} style="font-weight: bold;"{% endif %}>
4. {{ musician.name }}
5. </td>
6. <td>{{ musician.genre }}</td>
7. </tr>
8. {% endfor %}

看看这个模板是不是更清晰？这比通常情况更复杂，通常你会和数据库对象打交道，而数据库对象  
会有自定义方法(如is\_important())，下一章我们会讲到数据库对象  
3，同上一题很类似，解决方法也很类似，关键是预处理音乐家是否需要在名字后面加星号  
这属于业务逻辑，它属于视图，下面是视图的一种实现：

1. def musician\_list(request):
2. musicians = []
3. **for** m in MUSICIANS:
4. musicians.append({
5. 'name': m['name'],
6. 'genre': m['genre'],
7. 'is\_important': m['genre'] in ('rock', 'jazz'),
8. 'is\_pretentious': ' ' not in m['name'],
9. })
10. **return** render\_to\_response('musician\_list.html', {'musicians': musicians})

我们使用' ' not in m['name']表达式，如果m['name']不包含空格就返回True，你也可以使用.find()方法：

1. 'is\_pretentious': m['name'].find(' ') == -1

注意我们调用的是is\_pretentious而不是has\_asterisk，因为使用星号是由呈现层来决定的  
我们使用下面的模板代码：

1. {% **for** musician in musicians %}
2. <tr>
3. <td{% **if** musician.is\_important %} style="font-weight: bold;"{% endif %}>
4. {{ musician.name }}{% **if** musician.is\_pretentious %}\*{% endif %}
5. </td>
6. <td>{{ musician.genre }}</td>
7. </tr>
8. {% endfor %}

别忘了模板底部加上“\* Pretentious.”  
为了加分，你应该成为专家而仅当至少有一个被修饰的音乐家时显示“\* Pretentious”脚注  
想下面这样决定视图里是否有被修饰的音乐家

1. def musician\_list(request):
2. musicians = []
3. has\_pretentious = False
4. **for** m in MUSICIANS:
5. **if** ' ' not in m['name']:
6. has\_pretentious = True
7. musicians.append({
8. 'name': m['name'],
9. 'genre': m['genre'],
10. 'is\_important': m['genre'] in ('rock', 'jazz'),
11. 'is\_pretentious': ' ' not in m['name'],
12. })
13. **return** render\_to\_response('musician\_list.html', {
14. 'musicians': musicians,
15. 'has\_pretentious': has\_pretentious,
16. })

我们多传递一个模板变量has\_pretentious，这样在模板中使用它：

1. {% **if** has\_pretentious %}\* Pretentious{% endif %}

4，这里是基本模板的一种实现：

1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2. <html lang="en">
3. <head>
4. <link rel="stylesheet" href="default.css" type="text/css">
5. <title>{% block title %}{% endblock %}</title>
6. {% block extrahead %}{% endblock %}
7. </head>
8. <body>
9. <h1 id="top">{% block headline %}{% endblock %}</h1>
10. {% block content %}{% endblock %}
11. <hr>
12. <p><a href="#top">Back to top</a>.</p>
13. </body>
14. </html>

模板1：

1. {% **extends** "base.html" %}
3. {% block title %}My to-**do** list{% endblock %}
5. {% block headline %}Latest tasks{% endblock %}
7. {% block content %}
8. {% **if** task\_list %}
9. <ul>
10. {% **for** task in task\_list %}<li>{{ task }}</li>{% endfor %}
11. </ul>
12. {% **else** %}
13. <p>You have no tasks.</p>
14. {% endif %}
15. {% endblock %}

模板2：

1. {% **extends** "base.html" %}
3. {% block title %}Task: {{ task.title }} | To-**do** list{% endblock %}
5. {% block headline %}{{ task.title }}{% endblock %}
7. {% block content %}<p>{{ task.description }}</p>{% endblock %}

模板3：

1. {% **extends** "base.html" %}
3. {% block title %}Completed tasks | To-**do** list{% endblock %}
5. {% block extrahead %}<script type="text/javascript" src="completed.js">{% endblock %}
7. {% block headline %}{{ task.title }}{% endblock %}
9. {% block content %}<p>{{ task.description }}</p>{% endblock %}

注意我们喜欢在几个{% block %}部分之间放置一个空行，但这只是个人风格  
子模板中{% block %}标签以外的任何内容都不会被渲染

## 5.5[静态文件配置 static](http://blog.csdn.net/sinat_21302587/article/details/74059078)

我们可以使用Template 设置我们的网页，同时，一个完美的网页需要css，**[js](http://lib.csdn.net/base/javascript" \t "_blank" \o "JavaScript知识库)**，image 等静态文件的支持。django中配置方式貌似有不少总，因为很多相关的博客写的方式并不一致，当然这可能是django 的版本不同导致的。当我们在一个项目下创建一个app后，我们就需要为该app下创建一个static 文件夹来存放相关静态资源。但创建了多个app后，就需要在多个app下创建static。 这样引入了一个问题，因为，我们的可能用了同一个js文件。分别存放显然是浪费加载时间。于是在实际部署的时候，我们会将不同app下的static 文件放到一个“合适”的地方。提高资源加载速度，同时也方便管理。好了，大概就是这个思路，我们下面介绍如何配置

**配置**

### 5.5.1step one

--app

--migrations

--static

--css

--js

--image

--templates

--\_\_init\_\_.py

...

文件结构如上所示，注意，我这里只展示了某app下的结构，而不是整个项目结构，可能你的没有templates这个文件夹，这不要紧，这个是我创建的。

我们最好在static下分类好不同文件夹 css , js，image.(这是推荐的做法)

多个app时，我们就在不同的app 下创建static。（后面我们在讲更合理的方法，这里需要这样做为了你理清步骤）

### 5.5.2step two

在settings.py中的STATIC\_URL = '/static/'后面添加

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

这个STATIC\_ROOT 就是我们最终部署时候需要使用的，解决资源分散的问题。   
BASE\_DIR 这个变量在settings.py的开始部分被定义，就是项目根目录的目录名。   
os.path.join 方法，在项目根目录下新建一个static文件夹。

我看到很多博客这里配置的很多，很乱，让人摸不着头脑。   
这个地方完全可以这样配置，加载我们的静态资源。

### 5.5.3step three

创建文件夹完，配置文件这两步都完成后，我们需要的就是把他放到网页中显示。   
在app下新建一个templates,如果有就不用创建

--app

--migrations

--static

--templates

--appname

--index.html

--\_\_init\_\_.py

...

大家注意，我并没有直接在templates下直接创建index.html ,而是创建了一个”appname”(就根据你的app来命名这样不会冲突)。并在appname 下创建的index.html。

因为django去找template的时候是吧app下的templates ，存为一个list。如果我们多个app，就可能导致，想访问app2的index,结果却返回了app1的index.html。

index.html 内容

{% load static %} #这个地方引入static这个文件

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<img src="{% static "image/logo.png" %}" alt="My image"/> #{% static "image/logo.png" %}表示路径

</body>

</html>

这里的#号注释是方便大家理解，并不是html支持的格式，粘贴的时候需要去掉

并在image文件夹下加入一张名为logo.png的图片，因为图片可以直观的判读我们的设置是否生效。

**[Python](http://lib.csdn.net/base/python" \t "_blank" \o "Python知识库)** manage.py runserver 0.0.0.0:9000   
访问你的9000端口，查看时候生效。注意我们到这里还未配置nginx.

如果正确，说明路径没问题。

### 5.5.4step four

**[python](http://lib.csdn.net/base/python" \t "_blank" \o "Python知识库)** manage.py collectstatic   
该命令收集项目下的静态文件，统一保存到 STATIC\_ROOT 就是我们第二步，刚刚配置的。

--project

--project

--static

--app1

--app2

--manage.py

...

### 5.5.5step five

nginx 中配置

location ^~ /static/ {

root /home/project/;

}

这个地方 注意配置到static的上级就可以了。   
我之前配置成了root /home/project/static 就会一直提示404

在部署的时候，django也建议将settings.py中的debug=True 改为debug=False。以保证安全性。

重新启动nginx ,看看是否成功了呢？

如果成功了，可以将app下的static删掉再试试，理论上也是可以成功的，因为此时我们已经将项目所有的静态文件全都集中到了根目录下的static中。

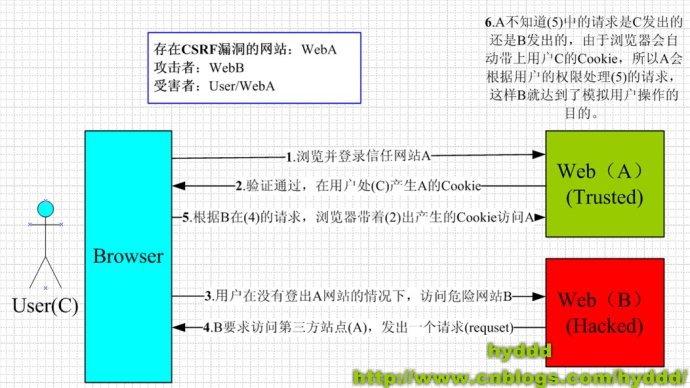
**总结**

配置静态文件还是要先了解他是如何生效的，one by one，搞清逻辑。   
那样在配置才不会混乱，否则不同的教程只会让人抓狂。

如果找不到相关博客的话，静下心来阅读官方文档。你可以做到的。

## 5.6 SRF

下面这张图片说明了CSRF的攻击原理：

[](http://photo.blog.sina.com.cn/showpic.html%23blogid=3fe961ae010164i5&url=http://s13.sinaimg.cn/orignal/3fe961aeg79f797297fdc)

Django中如何防范CSRF

Django使用专门的中间件（CsrfMiddleware）来进行CSRF防护。具体的原理如下：

1.它修改当前处理的请求，向所有的 POST 表单增添一个隐藏的表单字段，使用名称是 csrfmiddlewaretoken ，值为当前会话 ID 加上一个密钥的散列值。 如果未设置会话 ID ，该中间件将不会修改响应结果，因此对于未使用会话的请求来说性能损失是可以忽略的。

2.对于所有含会话 cookie 集合的传入 POST 请求，它将检查是否存在 csrfmiddlewaretoken 及其是否正确。 如果不是的话，用户将会收到一个 403 HTTP 错误。 403 错误页面的内容是检测到了跨域请求伪装。 终止请求。

该步骤确保只有源自你的站点的表单才能将数据 POST 回来。

另外要说明的是，未使用会话 cookie 的 POST 请求无法受到保护，但它们也不 需要 受到保护，因为恶意网站可用任意方法来制造这种请求。为了避免转换非 HTML 请求，中间件在编辑响应结果之前对它的 Content-Type 头标进行检查。 只有标记为 text/html 或 application/xml+xhtml 的页面才会被修改。

Django防范CSRF的具体操作

1.将'django.middleware.csrf.CsrfViewMiddleware'添加到Django的settings.py文件中的MIDDLEWARE\_CLASSES列表中（默认已经添加）。 该中间件必须在 SessionMiddleware 之后执行，因此在列表中 CsrfMiddleware 必须出现在SessionMiddleware 之前 （因为响应中间件是自后向前执行的）。 同时，它也必须在响应被压缩或解压之前对响应结果进行处理，因此CsrfMiddleware必须在GZipMiddleware之后执行。

MIDDLEWARE\_CLASSES = (

*'django.middleware.common.CommonMiddleware'*,

*'django.contrib.sessions.middleware.SessionMiddleware'*,

*'django.middleware.csrf.CsrfViewMiddleware'*,

*'django.contrib.auth.middleware.AuthenticationMiddleware'*,

*'django.contrib.messages.middleware.MessageMiddleware'*,

    # Uncomment the next line for simple clickjacking protection:

    # 'django.middleware.clickjacking.XFrameOptionsMiddleware',

)

2.在使用到POST方法提交FORM的页面中，添加csrf\_token标签，例如：

<form action=*"."* method=*"post"*>{% csrf\_token %}

3.在相应的view中，确保“django.core.context\_processors.csrf” 上下文处理器被正确使用，有两种方法实现这一点，一是使用[RequestContext](http://blog.sina.com.cn/s/blog_3fe961ae010164i5.html#subclassing-context-requestcontext)，它内部会自动使用到“django.core.context\_processors.csrf”。另一种方法是手动使用这个处理器，示例代码如下：

from django.core.context\_processors import csrf

from django.shortcuts import render\_to\_response

def **my\_view**(request):

    c = {}

    c.update(csrf(request))

    # ... view code here

return render\_to\_response(*"a\_template.html"*, c)

 (http://blog.sina.com.cn/s/blog\_3fe961ae010164i5.html)

## 5.7 介绍了Django中使用locals()函数的技巧,:

对 current\_datetime 的一次赋值操作:

def current\_datetime(request):

  now = datetime.datetime.now()

  return render\_to\_response('current\_datetime.html', {'current\_date': now})

很多时候，就像在这个范例中那样，你发现自己一直在计算某个变量，保存结果到变量中（比如前面代码中的 now ），然后将这些变量发送给模板。 尤其喜欢偷懒的程序员应该注意到了，不断地为临时变量和临时模板命名有那么一点点多余。 不仅多余，而且需要额外的输入。

如果你是个喜欢偷懒的程序员并想让代码看起来更加简明，可以利用 Python 的内建函数 locals() 。它返回的字典对所有局部变量的名称与值进行映射。 因此，前面的视图可以重写成下面这个样子：

def current\_datetime(request):

  current\_date = datetime.datetime.now()

  return render\_to\_response('current\_datetime.html', locals())

在此，我们没有像之前那样手工指定 context 字典，而是传入了 locals() 的值，它囊括了函数执行到该时间点时所定义的一切变量。 因此，我们将 now 变量重命名为 current\_date ，因为那才是模板所预期的变量名称。 在本例中， locals() 并没有带来多 大 的改进，但是如果有多个模板变量要界定而你又想偷懒，这种技术可以减少一些键盘输入。

使用 locals() 时要注意是它将包括 所有 的局部变量，它们可能比你想让模板访问的要多。 在前例中， locals() 还包含了 request 。对此如何取舍取决你的应用程序。

## 5.8 filter（django自定义simple\_tag）

a、在app中创建templatetags模块

b、创建任意.py文件，如：xxoo.py

[复制代码](javascript:void(0);)

from django import template

from django.utils.safestring import mark\_safe

register = template.Library()

@register.filter

def ya(value,v):

temp = '<a href="http://www.baidu.com?t=%s">%s</a>'%(value,v)

return mark\_safe(temp)

c、在使用自定义filter的html文件中导入之前创建的xxoo.py文件名

{% load xxoo %}

d、使用filter

{{ 8|ya:'跳转' }}

注：8是第一个参数，‘跳转’是第二个参数，filter只能传两个参数。

e.在settings中配置当前app，不然django无法找到自定义的simple\_tag

[复制代码](javascript:void(0);)

INSTALLED\_APPS = (

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'app01',

)

[复制代码](javascript:void(0);)

simple\_tag

a、在app中创建templatetags模块

b、创建任意 .py 文件，如：xx.py

from django import template

register = template.Library()

@register.simple\_tag

def my\_simple\_time(v1,v2,v3):

return v1 + v2 + v3

@register.simple\_tag

def my\_input(id,arg):

result = "<input type='text' id='%s' class='%s' />" %(id,arg,)

return result

c、在使用自定义simple\_tag的html文件中导入之前创建的 xx.py 文件名

{% load xx %}

d、使用simple\_tag

{% my\_simple\_time 1 2 3%}

{% my\_input 'id\_username' 'hide'%}

e、在settings中配置当前app，不然django无法找到自定义的simple\_tag

INSTALLED\_APPS = (

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'app01',

)

[复制代码](javascript:void(0);)

simple\_tag缺点：不能在前端用作if else判断条件,filter可以。

filter缺点：不能传任意参数，simple\_tag可以。

# 第六章 视图和路由

## 6.1URLconf

* 在Django中，定义URLconf包括正则表达式、视图两部分,Django使用正则表达式匹配请求的URL，一旦匹配成功，则调用应用的视图
* 在settings.py文件中通过ROOT\_URLCONF指定根级url的配置
* urlpatterns是一个url()实例的列表
* 一个url()对象包括：
  + - 正则表达式
    - 视图函数
    - 名称name
* 编写URLconf的注意：
  + - 若要从url中捕获一个值，需要在它周围设置一对圆括号
    - 不需要添加一个前导的反斜杠，如应该写作'test/'，而不应该写作'/test/'
    - 每个正则表达式前面的r表示字符串不转义
* 请求的url被看做是一个普通的python字符串，进行匹配时不包括get或post请求的参数及域名

http://www.baidu.cn/python/1/?i=1&p=new，只匹配“/python/1/”部分

* 正则表达式非命名组，通过位置参数传递给视图

url(r'^([0-9]+)/$', views.detail, name='detail'),

* 正则表达式命名组，通过关键字参数传递给视图，本例中关键字参数为id

url(r'^(?P<id>[0-9]+)/$', views.detail, name='detail'),

* 每个捕获的参数都作为一个普通的python字符串传递给视图

## **6.2包含其它的URLconfs**

* 在应用中创建urls.py文件，定义本应用中的urlconf，再在项目的settings中使用include()

from django.conf.urls import include, url

urlpatterns = [

url(r'^', include('booktest.urls', namespace='booktest')),

]

* 匹配过程：先与主URLconf匹配，成功后再用剩余的部分与应用中的URLconf匹配

请求http://www.baidu.cn/booktest/1/

在sesstings.py中的配置：

url(r'^booktest/', include('booktest.urls', namespace='booktest')),

在booktest应用urls.py中的配置

url(r'^([0-9]+)/$', views.detail, name='detail'),

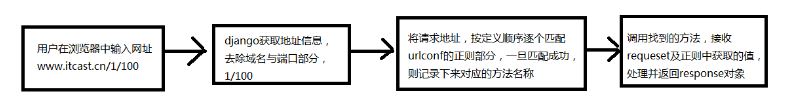
匹配部分是：/booktest/1/

匹配过程：在settings.py中与“booktest/”成功，再用“1/”与booktest应用的urls匹配

* 使用include可以去除urlconf的冗余
* 在include中通过namespace定义命名空间

## 6.3视图基础部分

响应处理过程如下图



前面写了**[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)**，或者说模型和模板，MTV里面就差V了，也就是视图。

所谓的视图，只不过是一个接受Web请求并返回Web响应的**[Python](http://lib.csdn.net/base/python" \t "_blank" \o "Python知识库)**函数。而这个响应不仅仅是html内容，还可以是一次重定向、一条404错误、一张图片。

打开views.py文件，添加代码：

**[python]** [view plain](http://blog.csdn.net/whoami021/article/details/19133149" \t "_blank" \o "view plain)

1. **from** django.http **import** HttpResponse
2. **import** datetime
4. **def** current\_datetime(request):
5. now = datetime.datetime.now()
6. html = "<html><body>Itis now %s.</body></html>" % now
7. **return** HttpResponse(html)

挑重点分析一下，视图就是current\_datetime()函数。这个视图返回一个HttpResponse对象。

这个视图怎么被找到呢？也就是说，怎么将一个URL[http://mysite.com/time/](http://mysite.com/current_datetime/%E6%98%A0%E5%B0%84%E5%88%B0%E8%AF%95%E5%9B%BE%E5%91%A2" \t "_blank)映射到试图呢？django使用URLconf。

打开urls.py文件，

1.引入current\_datetime视图，假如这个视图写在mysite.views模块中，就添加以下代码：

**[python]** [view plain](http://blog.csdn.net/whoami021/article/details/19133149" \t "_blank" \o "view plain)

1. **from** mysite.views importcurrent\_datetime

2.找到urlpatterns= patterns('',

)

在里面添加一项：(r'^time/$',current\_datetime),

这样遇到URL/time/的请求，都会到mysite.views模块中找到current\_datetime试图来处理。

## 6.5错误视图

* Django原生自带几个默认视图用于处理HTTP错误

### 6.5.1 404 (page not found) 视图

* defaults.page\_not\_found(request, template\_name='404.html')
* 默认的404视图将传递一个变量给模板：request\_path，它是导致错误的URL
* 如果Django在检测URLconf中的每个正则表达式后没有找到匹配的内容也将调用404视图
* 如果在settings中DEBUG设置为True，那么将永远不会调用404视图，而是显示URLconf 并带有一些调试信息
* 在templates中创建404.html

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

找不到了

<hr/>

{{request\_path}}

</body>

</html>

* 在settings.py中修改调试

DEBUG = False

ALLOWED\_HOSTS = ['\*', ]

* 请求一个不存在的地址

http://127.0.0.1:8000/test/

### 6.5.2 500 (server error) 视图

* defaults.server\_error(request, template\_name='500.html')
* 在视图代码中出现运行时错误
* 如果在settings中DEBUG设置为True，那么将永远不会调用500视图，而是显示URLconf 并带有一些调试信息

## 6.6 HttpReqeust对象

* 服务器接收到http协议的请求后，会根据报文创建HttpRequest对象
* 视图函数的第一个参数是HttpRequest对象
* 在django.http模块中定义了HttpRequest对象的API

属性

* 下面除非特别说明，属性都是只读的
* path：一个字符串，表示请求的页面的完整路径，不包含域名
* method：一个字符串，表示请求使用的HTTP方法，常用值包括：'GET'、'POST'
* GET：一个类似于字典的对象，包含get请求方式的所有参数
* POST：一个类似于字典的对象，包含post请求方式的所有参数
* FILES：一个类似于字典的对象，包含所有的上传文件
* COOKIES：一个标准的Python字典，包含所有的cookie，键和值都为字符串
* session：一个既可读又可写的类似于字典的对象，表示当前的会话，只有当Django 启用会话的支持时才可用，详细内容见“状态保持”

方法

* is\_ajax()：如果请求是通过XMLHttpRequest发起的，则返回True

### 6.6.1 QueryDict对象

* request对象的属性GET、POST都是QueryDict类型的对象
* 与python字典不同，QueryDict类型的对象用来处理同一个键带有多个值的情况
* 方法get()：根据键获取值
  + 只能获取键的一个值
  + 如果一个键同时拥有多个值，获取最后一个值

dict.get('键',default)

或简写为

dict['键']

* 方法getlist()：根据键获取值
  + 将键的值以列表返回，可以获取一个键的多个值

dict.getlist('键',default)

### 6.6.2 GET属性

* QueryDict类型的对象
* 包含get请求方式的所有参数
* 与url请求地址中的参数对应，位于?后面
* 参数的格式是键值对，如key1=value1
* 多个参数之间，使用&连接，如key1=value1&key2=value2
* 键是开发人员定下来的，值是可变的
* 示例如下
* 创建视图getTest1用于定义链接，getTest2用于接收一键一值，getTest3用于接收一键多值

def getTest1(request):

return render(request,'booktest/getTest1.html')

def getTest2(request):

return render(request,'booktest/getTest2.html')

def getTest3(request):

return render(request,'booktest/getTest3.html')

* 配置url

url(r'^getTest1/$', views.getTest1),

url(r'^getTest2/$', views.getTest2),

url(r'^getTest3/$', views.getTest3),

* 创建getTest1.html，定义链接

<html>

<head>

<title>Title</title>

</head>

<body>

链接1：一个键传递一个值

<a href="/getTest2/?a=1&b=2">gettest2</a><br>

链接2：一个键传递多个值

<a href="/getTest3/?a=1&a=2&b=3">gettest3</a>

</body>

</html>

* 完善视图getTest2的代码

def getTest2(request):

a=request.GET['a']

b=request.GET['b']

context={'a':a,'b':b}

return render(request,'booktest/getTest2.html',context)

* 创建getTest2.html，显示接收结果

<html>

<head>

<title>Title</title>

</head>

<body>

a:{{ a }}<br>

b:{{ b }}

</body>

</html>

* 完善视图getTest3的代码

def getTest3(request):

a=request.GET.getlist('a')

b=request.GET['b']

context={'a':a,'b':b}

return render(request,'booktest/getTest3.html',context)

* 创建getTest3.html，显示接收结果

<html>

<head>

<title>Title</title>

</head>

<body>

a:{% for item in a %}

{{ item }}

{% endfor %}

<br>

b:{{ b }}

</body>

</html>

### 6.6.3 POST属性

* QueryDict类型的对象
* 包含post请求方式的所有参数
* 与form表单中的控件对应
* 问：表单中哪些控件会被提交？
* 答：控件要有name属性，则name属性的值为键，value属性的值为键，构成键值对提交
  + 对于checkbox控件，name属性一样为一组，当控件被选中后会被提交，存在一键多值的情况
* 键是开发人员定下来的，值是可变的
* 示例如下
* 定义视图postTest1

def postTest1(request):

return render(request,'booktest/postTest1.html')

* 配置url

url(r'^postTest1$',views.postTest1)

* 创建模板postTest1.html

<html>

<head>

<title>Title</title>

</head>

<body>

<form method="post" action="/postTest2/">

姓名：<input type="text" name="uname"/><br>

密码：<input type="password" name="upwd"/><br>

性别：<input type="radio" name="ugender" value="1"/>男

<input type="radio" name="ugender" value="0"/>女<br>

爱好：<input type="checkbox" name="uhobby" value="胸口碎大石"/>胸口碎大石

<input type="checkbox" name="uhobby" value="跳楼"/>跳楼

<input type="checkbox" name="uhobby" value="喝酒"/>喝酒

<input type="checkbox" name="uhobby" value="爬山"/>爬山<br>

<input type="submit" value="提交"/>

</form>

</body>

</html>

* 创建视图postTest2接收请求的数据

def postTest2(request):

uname=request.POST['uname']

upwd=request.POST['upwd']

ugender=request.POST['ugender']

uhobby=request.POST.getlist('uhobby')

context={'uname':uname,'upwd':upwd,'ugender':ugender,'uhobby':uhobby}

return render(request,'booktest/postTest2.html',context)

* 配置url

url(r'^postTest2$',views.postTest2)

* 创建模板postTest2.html

<html>

<head>

<title>Title</title>

</head>

<body>

{{ uname }}<br>

{{ upwd }}<br>

{{ ugender }}<br>

{{ uhobby }}

</body>

</html>

* 注意：使用表单提交，注释掉settings.py中的中间件crsf

## 6.7HttpResponse对象

* 在django.http模块中定义了HttpResponse对象的API
* HttpRequest对象由Django自动创建，HttpResponse对象由程序员创建
* 不调用模板，直接返回数据

#coding=utf-8

from django.http import HttpResponse

def index(request):

return HttpResponse('你好')

* 调用模板

from django.http import HttpResponse

from django.template import RequestContext, loader

def index(request):

t1 = loader.get\_template('polls/index.html')

context = RequestContext(request, {'h1': 'hello'})

return HttpResponse(t1.render(context))

属性

content：表示返回的内容，字符串类型;

charset：表示response采用的编码字符集，字符串类型;

status\_code：响应的HTTP响应状态码;

方法

* 子类HttpResponseRedirect
* 请求地址栏如图：

请求地址栏

* 请求结果的地址栏如图：

请求地址栏

* 推荐使用反向解析

from django.core.urlresolvers import reverse

def index(request):

return HttpResponseRedirect(reverse('booktest:index2', args=(1,)))

子类JsonResponse

* 返回json数据，一般用于异步请求
* \_init \_(data)
* 帮助用户创建JSON编码的响应
* 参数data是字典对象
* JsonResponse的默认Content-Type为application/json

from django.http import JsonResponse

def index2(requeset):

return JsonResponse({'list': 'abc'})

**简写函数**

render

* render(request, template\_name[, context])
* 结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的HttpResponse对象
* request：该request用于生成response
* template\_name：要使用的模板的完整名称
* context：添加到模板上下文的一个字典，视图将在渲染模板之前调用它

from django.shortcuts import render

def index(request):

return render(request, 'booktest/index.html', {'h1': 'hello'})