# 一、使用虚拟环境

## 虚拟环境原理介绍：

虚拟环境相当于一个抽屉，在这个抽屉中安装的任何软件包都不会影响到其他抽屉。并且在项目中，我可以指定这个项目的虚拟环境来配合我的项目。比如我们现在有一个项目是基于Django 1.10.x版本，又有一个项目是基于Django 0.9.x的版本，那么这时候就可以创建两个虚拟环境，在这两个虚拟环境中分别安装Django 1.10.x和Django 0.9.x来适配我们的项目。

## 安装virtualenv：

virtualenv是用来创建虚拟环境的软件工具，我们可以通过pip或者pip3来安装：

pip install virtualenv

pip3 install virtualenv

## 创建虚拟环境：

创建虚拟环境非常简单，通过以下命令就可以创建了：

virtualenv 虚拟环境的名字

## 进入环境：

虚拟环境创建好了以后，那么可以进入到这个虚拟环境中，然后安装一些第三方包，进入虚拟环境在不同的操作系统中有不同的方式，一般分为两种，第一种是Windows，第二种是\*nix：

1. windows进入虚拟环境：进入到虚拟环境的Scripts文件夹中，然后执行activate。

2. \*nix进入虚拟环境：source /path/to/virtualenv/bin/activate 一旦你进入到了这个虚拟环境中，你安装包，卸载包都是在这个虚拟环境中，不会影响到外面的环境。

## 退出虚拟环境：

退出虚拟环境很简单，通过一个命令就可以完成：deactivate。

## 创建虚拟环境的时候指定Python解释器：

在电脑的环境变量中，一般是不会去更改一些环境变量的顺序的。也就是说比如你的Python2/Scripts在Python3/Scripts的前面，那么你不会经常去更改他们的位置。但是这时候我确实是想在创建虚拟环境的时候用Python3这个版本，这时候可以通过-p参数来指定具体的Python解释器：

virtualenv -p C:\Python36\python.exe virutalenv\_name

## virtualenvwrapper：

virtualenvwrapper这个软件包可以让我们管理虚拟环境变得更加简单。不用再跑到某个目录下通过virtualenv来创建虚拟环境，并且激活的时候也要跑到具体的目录下去激活。

### 安装virtualenvwrapper：

1. \*nix：pip install virtualenvwrapper。
2. windows：pip install virtualenvwrapper-win。

### virtualenvwrapper基本使用：

1. 创建虚拟环境：

mkvirtualenv my\_env

那么会在你当前用户下创建一个Env的文件夹，然后将这个虚拟环境my\_env安装到这个目录下。 以后创建的虚拟环境都安装在这个Env目录下。如果你电脑中安装了python2和python3，并且两个版本中都安装了virtualenvwrapper，那么将会使用环境变量中第一个出现的Python版本来作为这个虚拟环境的Python解释器。

1. 切换到某个虚拟环境：

workon my\_env

1. 退出当前虚拟环境：

deactivate

1. 删除某个虚拟环境：

rmvirtualenv my\_env

1. 列出所有虚拟环境：

lsvirtualenv

1. 进入到虚拟环境所在的目录：

cdvirtualenv

### 修改mkvirtualenv的默认路径：

在我的电脑->右键->属性->高级系统设置->环境变量->系统变量中添加一个参数WORKON\_HOME，将这个参数的值设置为你需要的路径。

### 创建虚拟环境的时候指定Python版本：

在使用mkvirtualenv的时候，可以指定--python的参数来指定具体的python路径：

mkvirtualenv --**python**==C:\Python36\**python**.**exe** hy\_env

# 二、URL组成部分详解

URL是Uniform Resource Locator的简写，统一资源定位符。

一个URL由以下几部分组成：

scheme://host:port/path/?query-string=xxx#anchor

scheme：代表的是访问的协议，一般为http或者https以及ftp等。

host：主机名，域名，比如www.baidu.com。

port：端口号。当你访问一个网站的时候，浏览器默认使用80端口。

path：查找路径。比如：www.jianshu.com/trending/now，后面的trending/now就是path。

query-string：查询字符串，比如：www.baidu.com/s?wd=python，后面的wd=python就是查询字符串。

anchor：锚点，后台一般不用管，前端用来做页面定位的。

注意：URL中的所有字符都是ASCII字符集，如果出现非ASCII字符，比如中文，浏览器会进行编码再进行传输。

# Django

## Django的下载和基本命令

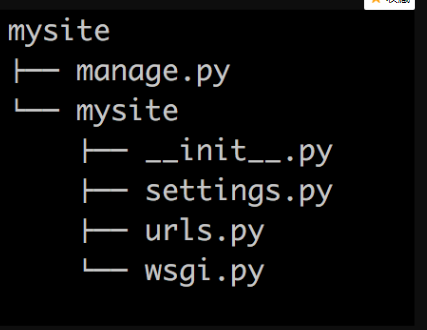
### 1、下载Django：

pip3 install django

### 2、创建一个django project

#### 1.通过命令行创建

django-admin.py startproject mysite



当前目录下会生成mysite的工程，目录结构如下：

manage.py :Django项目里面的工具，通过它可以调用django shell和数据库等，和

Django进行交互。以后和项目交互基本上都是基于这个文件。一般都是在终端输入python manage.py [子命令]。可以输入python manage.py help看下能做什么事情。除非你知道你自己在做什么，一般情况下不应该编辑这个文件。

settings.py:包含了项目的默认设置，包括数据库信息，调试标志以及其他一些工作变

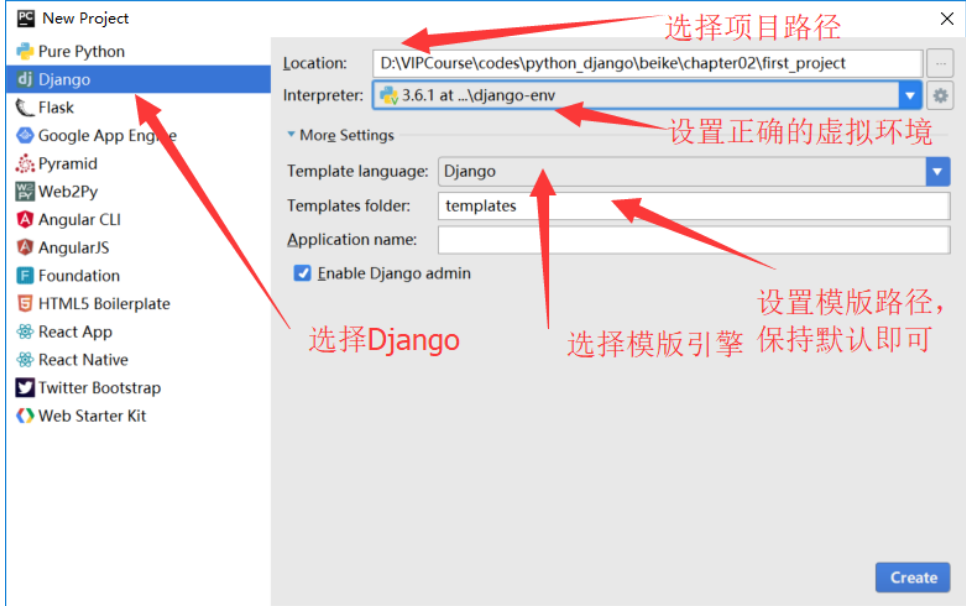
量。本项目的设置项，以后所有和项目相关的配置都是放在这个里面。

urls.py :负责把URL模式映射到应用程序。这个文件是用来配置URL路由的。比如访问http://127.0.0.1/news/是访问新闻列表页，这些东西就需要在这个文件中完成。

wsgi.py：项目与WSGI协议兼容的web服务器入口，部署的时候需要用到的，一般情况下也是不需要修改的。

#### 2.通过pycharm创建

用pycharm新建一个Django项目，新建项目的截图如下：



使用pycharm创建完项目后，还是需要重新进入到命令行单独创建app的。

### 3、在mysite目录下创建应用

创建项目中的app。所谓app就是一个大项目可以分为几个大功能，每个主要功能模块可以建立一个app，便于管理。

python manage.py startapp blog



### 4.启动服务器/设置访问ip

python manage.py runserver

如果想更改端口

python manage.py runserver 8010（端口名）

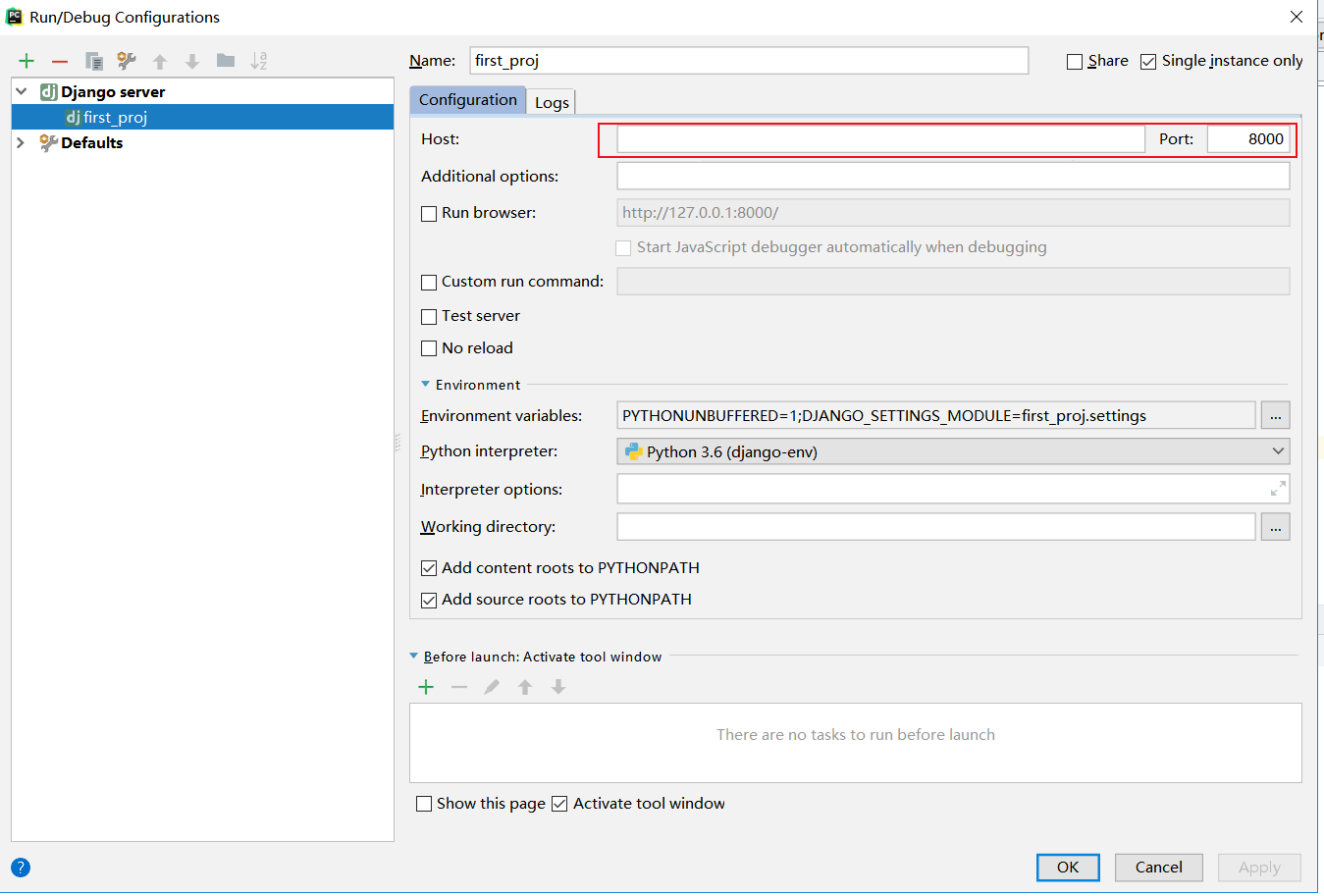
时服务器监听指定ip

python manage.py runserver ip:端口

如果ip为0.0.0.0 则为监听服务器所有开放ip

python manage.py runserver 0.0.0.0:端口，同时需要将服务器的的ip字符串保存在settings.py中的ALLOWED\_HOST中。

也可通过pycharm设置：



为避免一个实例运行多次，可以勾选上图的single instace only

## debug模式

setting.py中令DEBUG = True

优点：

1.修改完django代码后，django会自动重启服务器，不用手动重启

2.会在浏览器端和控制台打印错误信息。

需要关闭DEBUG模式的场景：

在部署上线网站时，需要关闭debug模式。

在关闭debug模式时，必须要设置ALLOW\_HOSTS。

## URL映射：

django之所以会通过urls.py文件来匹配url，是因为在settings中的设置:

ROOT\_URLCONF = **'first\_proj.urls' #first\_proj为该django项目的项目名**

会在该文件中找urlpatterns这个列表，进行遍历：

urlpatterns = [  
 path(**'admin/'**, admin.site.urls),  
]

所以要把url匹配写在这个列表中。通过匹配url来调用相应的视图函数。

视图写完后，要与URL进行映射，也即用户在浏览器中输入什么url的时候可以请求到这个视图函数。在用户输入了某个url，请求到我们的网站的时候，django会从项目的urls.py文件中寻找对应的视图。在urls.py文件中有一个urlpatterns变量，以后django就会从这个变量中读取所有的匹配规则。匹配规则需要使用django.urls.path函数进行包裹，这个函数会根据传入的参数返回URLPattern或者是URLResolver的对象。示例代码如下：

path函数参数包括一个回调函数，就是该url对应的视图函数。path将url中的字符串进行匹配，匹配成功后，调用回调函数。

from django.contrib import admin

from django.urls import path

from book import views #此处导入视图文件

urlpatterns = [

path('admin/', admin.site.urls),

path('book/',views.book\_list) #匹配url，调用相应视图函数

]

### url向视图传参数

有时候，url中包含了一些参数需要动态调整。比如简书某篇文章的详情页的url，是https://www.jianshu.com/p/a5aab9c4978e后面的a5aab9c4978e就是这篇文章的id，那么简书的文章详情页面的url就可以写成https://www.jianshu.com/p/<id>，其中id就是文章的id。那么如何在django中实现这种需求呢。

#### 方法一：使用<>括号

这时候我们可以在path函数中，使用尖括号的形式来定义一个参数。比如我现在想要获取一本书籍的详细信息，那么应该在url中指定这个参数。示例代码如下：

from django.contrib import admin

from django.urls import path

from book import views

urlpatterns = [

path('admin/', admin.site.urls),

path('book/',views.book\_list),

path('book/<book\_id>/',views.book\_detail)

]

而views.py中的代码如下：

def book\_detail(request,book\_id):

text = "您输入的书籍的id是：%s" % book\_id

return HttpResponse(text)

注意，url中变量名必须与view中变量名一致，不然会报错。

#### 方法二：通过查询字符串

当然，也可以通过查询字符串的方式传递一个参数过去。示例代码如下：

urlpatterns = [

path('admin/', admin.site.urls),

path('book/',views.book\_list),

path('book/detail/',views.book\_detail)

]

在views.py中的代码如下：

def book\_detail(request):

book\_id = request.GET.get("id")

text = "您输入的书籍id是：%s" % book\_id

return HttpResponse(text)

以后在访问的时候就是通过/book/detail/?id=1即可将参数传递过去。

### url映射设定默认参数

建立两个url映射，一个视图函数。

一个url是带参数的，一个是不带参数的，这两个url映射指向同一个视图函数。

视图函数带参数，但是参数指定默认值。

这样，如果进来的url不带参数，则视图函数则会调用默认参数。如果带参数，则使用传递的参数。

urls中：

path(‘content/’,view.show\_content)

path(‘content/<int:content\_page>/’,view.show\_content)

视图函数中：

def show\_content(request,content\_page=1):

…..

return …..

### path规定传递参数类型/path转换器

#### path规定匹配类型及转换

route参数：url的匹配规则。这个参数中可以指定url中需要传递的参数，比如在访问文章详情页的时候，可以传递一个id。传递参数是通过<>尖括号来进行指定的。并且在传递参数的时候，可以指定这个参数的数据类型，比如文章的id都是int类型，那么可以这样写<int:id>，以后匹配的时候，就只会匹配到id为字符型数字的url，并自动将其转换为整型int，而不会匹配其他的url，并且在视图函数中获取这个参数的时候，就已经被转换成一个int类型了。其中还有几种常用的类型：

* str：非空的字符串类型。默认的转换器。不包含斜杠。（包含纯数字字符）
* int：匹配任意的零或者正数的整形。到视图函数中就是一个int类型。（即整数字符），传递到视图函数时，会转换为int类型。
* slug：由英文中的横杠-，或者下划线\_连接英文字符或者数字而成的字符串。
* uuid：匹配uuid字符串。
* path：匹配非空的英文字符串，可以包含斜杠。

path函数如果没有使用转换器，就是使用的默认str转换器。

#### 自定义path转换器

如果想要自己写一个转换器，可以

from django.urls import converters

观察通用转换器如何写，进行模仿。

自定义url转换器按照以下五个步骤来走就可以了：

1. 定义一个类，直接继承自object就可以了。

2. 在类中定义一个属性regex，这个属性是用来限制url转换器规则的正则表达式。

3. 实现to\_python(self,value)方法，这个方法是将url中的值进行转换，然后传给视图函数。

4. 实现to\_url(self,value)方法，这个方法是在做url反转的时候，将传进来的参数进行转换并返回给reverse，进行url拼接。 一般将value进行拼接、类型转换，最终转换为字符串。

5. 将定义好的转换器，使用`django.urls.converters.register\_converter`方法注册到django中。

注册语句要写在可以自动运行的文件中。如urls中；或者直接在定义转换器的文件中注册，但是要导入到urls中；或者直接放在项目下的\_\_init\_\_文件中

from django.urls import register\_converter, path

from . import converters, views

#注册自定义转换器，并为其起名字。

register\_converter(converters.FourDigitYearConverter, 'yyyy') #‘yyyy’为转换器的名字 ，converters.FourDigitYearConverter为我们自定义的转化器类

示例代码如下：

from django.urls import register\_converter

#定义转换器

class CategoryConverter(object):

regex = r'\w+|(\w+\+\w+)+'

def to\_python(self,value):

# python+django+flask

# ['python','django','flask']

result = value.split("+")

return result

def to\_url(self,value):

# value：['python','django','flask']

# python+django+flask

if isinstance(value,list):

result = "+".join(value)

return result

else:

raise RuntimeError("转换url的时候，分类参数必须为列表！")

register\_converter(CategoryConverter,'cate') #注册转换器

### re\_path

1. re\_path和path的作用都是一样的。只不过`re\_path`是在写url的时候可以用正则表达式，功能更加强大。

2. 写正则表达式都推荐使用原生字符串。也就是以`r`开头的字符串。

3. 在正则表达式中定义变量，需要使用圆括号括起来。如果这个参数是有名字的，那么需要使用`?P<参数的名字>`。然后在后面添加正则表达式的规则。示例代码如下：

```python

from django.urls import re\_path

from . import views

urlpatterns = [

# r""：代表的是原生字符串（raw）

re\_path(r'^$',views.article),

# /article/list/<year>/

re\_path(r"^list/(?P<year>\d{4})/$",views.article\_list), #具有参数名的分组，参数名为year

re\_path(r"^list/(?P<month>\d{2})/$",views.article\_list\_month)

]

```

4. 如果不是特别要求。直接使用`path`就够了，省的把代码搞的很麻烦（因为正则表达式其实是非常晦涩的，特别是一些比较复杂的正则表达式，今天写的明天可能就不记得了）。除非是url中确实是需要使用正则表达式来解决才使用`re\_path`。

### 路由分发分层/include详解

#### 路由分发

当url过多，全都写在一个文件中就不方便管理。所以可以使用include进行url分级管理。

from django.urls import include



当第一级urls匹配到app01之后，会按照include指定的文件地址中，继续接下来的匹配。

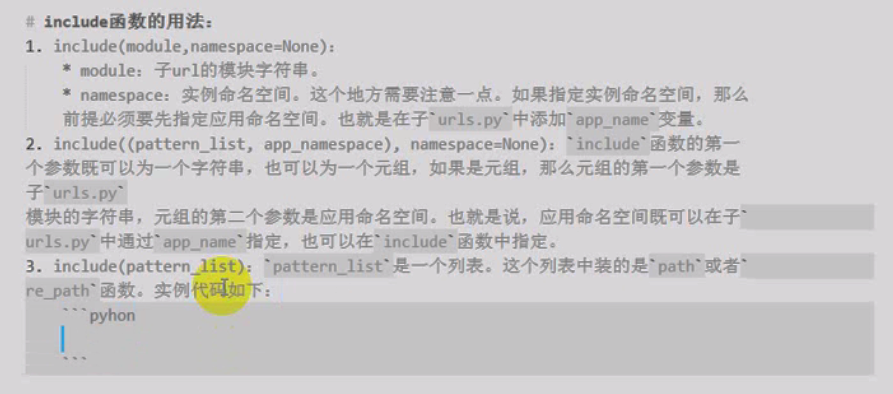
注意，include指定的文件也要符合urls的规范，即有urlpattern列表。

后续的urls：



#### include详解

几种用法：



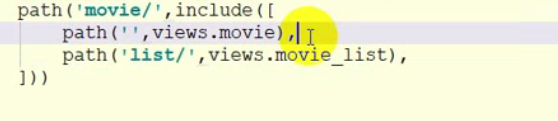
第一种：

path(‘movie’,include(‘app01.urls’,namespace=’csm’))，此种方法必须在应用命名空间存在情况下才可以

第二种：

path(‘movie’,include((‘app01.urls’,’app01’),’csm’)) ，此种方法可以直接使用，因为在include中就指明的应用命名空间。

第三种不常用，其实就是将url分发直接放在include中。



直接将movie/的下一个url拼接放在include中。

### url的命名与反转reverse

#### url命名

url的命名：类似于变量和变量名的关系，在视图函数和模板中使用url的别名，url具体内容可以改变，但是别名不会变，django会根据别名找到url的具体内容。批量修改该url时方便。

命名：

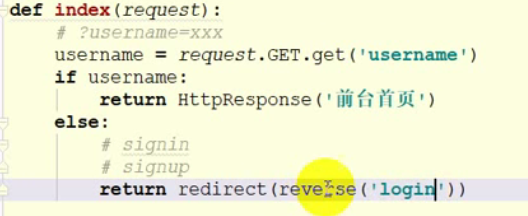
在path或re\_path中添加name参数。

path(‘singup/’,view.index,name=’login’) #此处login就是别名，相当于url的变量名，singup/ 这个url的具体内容可以改变，在视图函数中可以一直使用login别名来指代此处的url。

在视图函数中使用：

需要使用reverse反向解析：

form django.shorcuts import reverse



示例中login就是一个url的别名。

reverse主要做的就是找到login代表的url，并返回该url。

#### 应用命名空间

当项目的app较多时，不同app间的url命名就有可能会重复，此时可以指定url命名的名称空间，名称空间不同，即使url命名相同，相互之间也不影响。其实就是将url命名价格前缀。

方法：

在该app的urls.py中设置变量 app\_name = …..

一般情况下会将app的名称赋值给app\_name

这样该app的名称空间即为该名称。

view中使用

使用reverse反向解析时，需要加上该名称空间：



front即为app\_name的值

#### 实例命名空间include

使用实例命名空间的前提是必须使用应用命名空间。

使用include时，指定实例名称空间

re\_path（r'app01/’,include((“app01.urls”,”instance\_name”))）

可以在一个app下设置多个实例命名空间。

#### 正则表达式的反向解析/传递参数

1.需要向命名中传递参数：

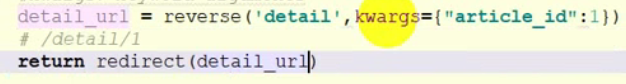


此url中含有一个正则表达式分组。即需要参数

则在解析时，要填入一个相应的参数，位数和分组要求的位数一致，大小随意。



如果参数有名称的话，也可以使用关键字参数kwargs = {‘article\_id’:所需具体内容}



如果在模板层设置别名时指定了名称空间，则在反向解析时也要指定名称空间。

2.如果想要添加查询字符串的参数，则必须手动的进行拼接。示例代码如下：

```python

login\_url = reverse('login') + "?next=/"

```

#### 获取当前名称空间

current\_namespace = request.resolver\_match.namespace

reverse(‘%s:url别名’%current\_namespace)

## 模板

### 渲染模板方式

渲染模板有多种方式。这里讲下两种常用的方式。

#### render\_to\_string：

找到模板，然后将模板编译后渲染成Python的字符串格式。最后再通过HttpResponse类包装成一个HttpResponse对象返回回去。示例代码如下：

from django.template.loader import render\_to\_string

from django.http import HttpResponse

def book\_detail(request,book\_id):

html = render\_to\_string("detail.html")

return HttpResponse(html)

以上方式虽然已经很方便了。但是django还提供了一个更加简便的方式，

#### 直接渲染

将模板渲染成字符串和包装成HttpResponse对象一步到位完成。示例代码如下：

from django.shortcuts import render

def book\_list(request):

return render(request,'list.html')

这些文件都是放在项目中templates根目录中的，可以直接写文件名。经过settings配置后，django会直接到项目templates文件夹中找模板文件。

### 模板查找路径配置：

在项目的settings.py文件中。有一个TEMPLATES配置，这个配置包含了模板引擎的配置，模板查找路径的配置，模板上下文的配置等。模板路径可以在两个地方配置。

1. `DIRS`：这是一个列表，在这个列表中可以存放所有的模板路径，以后在视图中使用render或者render\_to\_string渲染模板的时候，会在这个列表的路径中查找模板。

2. `APP\_DIRS`：默认为True，这个设置为True后，会在INSTALLED\_APPS的安装了的APP下的templates文件加中查找模板。需要在settings中INSTALLED\_APPS注册创建的APP。直接将app的名称写入该列表即可。

3. 查找顺序：比如代码render('list.html')。先会在DIRS这个列表中依次查找路径下有没有这个模板，如果有，就返回。如果DIRS列表中所有的路径都没有找到，那么会先检查当前这个视图所处的app是否已经安装，如果已经安装了，那么就先在当前这个app下的templates文件夹中查找模板，如果没有找到，那么会在其他已经安装了的app中查找。如果所有路径下都没有找到，那么会抛出一个TemplateDoesNotExist的异常。

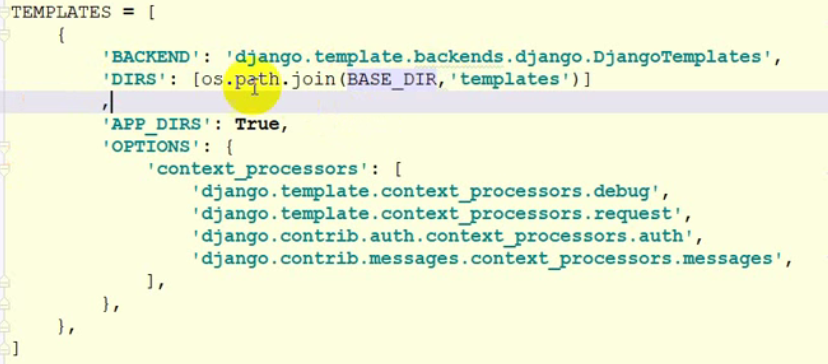


BASE\_DIR指的是项目默认根目录。

os.path.abspath(\_\_file\_\_)：获取当前此代码所在的文件的地址

os.path.dirname(filepath)：获取地址参数的上一层地址

os.getcwd()：获取主程序入口地址。



### 模板变量

#### 1.传递使用参数

在模版中使用变量，需要将变量放到 {{ 变量名 }} 中。变量名与视图函数render传递参数字典的键是一样的。

# profile.html模板代码

<p>{{ username }}</p>

# views.py代码

def profile(request):

return render(request,'profile.html',context={'username':'huangyong'})

render传递参数的形参名是context，是个字典形式。

#### 2.对字典、对象、列表、元组类型参数的使用

模板中的变量同样也支持点(.)的形式。在出现了点的情况，比如person.username，模板是按照以下方式进行解析的：

如果person是一个字典，那么就会查找这个字典的username这个key对应的值。

如果person是一个对象，那么就会查找这个对象的username属性，或者是username这个方法。

列表或元组如果出现的是person.0，会判断persons是否是一个列表或者元组或者任意的可以通过下标访问的对象，如果是的话就取这个列表的第1个值。如果不是就获取到的是一个空的字符串。

#### 3.注意：

1.不能通过中括号的形式访问字典和列表中的值，比如dict['key']和list[1]是不支持的！

2.传递参数时设置字典的key不能与字典固有方法重名。因为使用点（.）语法获取对象值的时候，可以获取这个对象的属性，如果这个对象是一个字典，也可以获取这个字典的值。所以在给这个字典添加key的时候，千万不能和字典中的一些属性重复。比如items/keys，items是字典的方法，那么如果给这个字典添加一个items作为key，那么以后就不能再通过item来访问这个字典的键值对了。

#### 4.keys/items/values

在模板中使用 .keys可以获取字典的所有key

.values 可以获取该字典的值

items是将键和值组合成为元组，并返回所有元组。

以上可以通过模板语法中的for循环进行遍历。

### 模板标签

使用{% %}进行标识

更多标签请参考官方文档：https://docs.djangoproject.com/en/2.0/ref/templates/builtins/

#### 1.if标签

if标签中可以使用==、!=、<、<=、>、>=、in、not in、is、is not等判断运算符。

{% if num > 100 or num < 0 %}

<p>无效</p>

{% elif num > 80 and num < 100 %}

<p>优秀</p>

{% else %}

<p>凑活吧</p>

{% endif %}

#### 2.for…in标签/遍历与反向遍历

for...in...标签：for...in...类似于Python中的for...in...。可以遍历列表、元组、字符串、字典等一切可以遍历的对象。

##### ①遍历字典/对象/列表/元组

{% for person in persons %} #persons为字典或对象

<p>{{ person.name }}</p>

{% endfor %}

{% for day in days %} #days为元组和列表

<p>{{ day }}</p>

{% endfor %}

##### ②反向遍历

如果想要反向遍历，那么在遍历的时候就加上一个reversed。示例代码如下：

{% for person in persons reversed %}

<p>{{ person.name }}</p>

{% endfor %}

##### ③使用items/keys/values

遍历字典的时候，需要使用items、keys和values等方法。在DTL中，执行一个方法不能使用圆括号的形式。遍历字典示例代码如下：

{% for key,value in persons.items %} #

<p>key：{{ key }}</p>

<p>value：{{ value }}</p>

{% endfor %}

{{ persons.items }} #将字典键值存放在一个元组中，并返回

{{ persons.values }} #返回存放字典内容的列表

{{ persons.keys }} #返回存放字典key的列表

##### ④for循环中自带的变量

在for循环中，DTL提供了一些变量可供使用。这些变量如下：

forloop.counter：当前循环的下标。以1作为起始值。

forloop.counter0：当前循环的下标。以0作为起始值。

forloop.revcounter：当前循环的反向下标值。比如列表有5个元素，那么第一次遍历这个属性是等于5，第二次是4，以此类推。并且是以1作为最后一个元素的下标。显示还可以循环几次+1

forloop.revcounter0：类似于forloop.revcounter。不同的是最后一个元素的下标是从0开始。显示还可以循环几次

forloop.first：是否是第一次遍历。

forloop.last：是否是最后一次遍历。

forloop.parentloop：如果有多个循环嵌套，那么这个属性代表的是上一级的for循环。

#### 3.for..in..empty

这个标签使用跟for...in...是一样的，只不过是在遍历的对象如果没有元素的情况下，会执行empty中的内容。示例代码如下：

{% for person in persons %}

<li>{{ person }}</li>

{% empty %}

暂时还没有任何人

{% endfor %}

#### 4.with标签/变量别名

在模版中定义变量。有时候一个变量访问的时候比较复杂，那么可以先把这个复杂的变量缓存到一个变量上，以后就可以直接使用这个变量就可以了。示例代码如下：

context = {

"persons": ["张三","李四"]

}

{% with lisi=persons.1 %} #在with范围内的代码都可以使用该缩写

<p>{{ lisi }}</p>

{% endwith %}

有几点需要强烈的注意：

在with语句中定义的变量，只能在{%with%}{%endwith%}中使用，不能在这个标签外面使用。

定义变量的时候，不能在等号左右两边留有空格。比如{% with lisi = persons.1%}是错误的。

还有另外一种写法同样也是支持的：as

{% with persons.1 as lisi %}

<p>{{ lisi }}</p>

{% endwith %}

#### 5.url标签/传递参数

在模板文件中使用url别名

在模版中，我们经常要写一些url，比如某个a标签中需要定义href属性。当然如果通过硬编码的方式直接将这个url写死在里面也是可以的。但是这样对于以后项目维护可能不是一件好事。因此建议使用这种反转的方式来实现，类似于django中的reverse一样。示例代码如下：

<a href="{% url 'book:list' %}">图书列表页面</a>

如果url反转的时候需要传递参数，那么可以在后面传递。但是参数分位置参数和关键字参数。位置参数和关键字参数不能同时使用。示例代码如下：

# path部分 指定url别名

path('detail/<book\_id>/',views.book\_detail,name='detail')

# url反转，使用位置参数 传递参数 1 book:为app名称空间

<a href="{% url 'book:detail' 1 %}">图书详情页面</a>

# url反转，使用关键字参数传递参数

<a href="{% url 'book:detail' book\_id=1 %}">图书详情页面</a>

如果想要在使用url标签反转的时候要传递查询字符串的参数，那么必须要手动在在后面添加。示例代码如下：

<a href="{% url 'book:detail' book\_id=1 %}?page=1">图书详情页面</a>

如果需要传递多个参数，那么通过空格的方式进行分隔。示例代码如下：

<a href="{% url 'book:detail' book\_id=1 page=2 %}">图书详情页面</a>

#### 6. spaceless标签/去除标签空白

移除html标签中的空白字符。包括空格、tab键、换行等。示例代码如下：

{% spaceless %}

<p>

<a href="foo/">Foo</a>

</p>

{% endspaceless %}

那么在渲染完成后，会变成以下的代码：

<p><a href="foo/">Foo</a></p>

spaceless只会移除html标签之间的空白字符。而不会移除标签与文本之间的空白字符。看以下代码：

{% spaceless %}

<strong>

Hello

</strong>

{% endspaceless %}

这个将不会移除strong中的空白字符。

#### 7. autoescape标签/转义开关

开启和关闭这个标签内元素的自动转义功能。自动转义是可以将一些特殊的字符。比如< 转义成html语法能识别的字符，比如<会被转义成&lt;，而>会被自动转义成&gt;。模板中默认是已经开启了自动转义的。autoescape的示例代码如下：

# 传递的上下文信息 views：

context = {

"info":"<a href='www.baidu.com'>百度</a>"

}

# 模板中关闭自动转义 不会自动转义，所见即所得

{% autoescape on %}

{{ info }}

{% endautoescape %}

那么就会显示百度的一个超链接。如果把on成off，那么就会显示成一个普通的字符串。示例代码如下：

{% autoescape off %}

{{ info }}

{% endautoescape %}

#### 8.verbatim标签/关闭解析引擎

默认在DTL模板中是会去解析那些特殊字符的。比如{%和%}以及{{等。如果你在某个代码片段中不想使用DTL的解析引擎。那么你可以把这个代码片段放在verbatim标签中。示例代码下：

{% verbatim %}

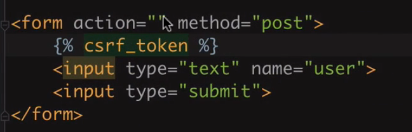
{{if dying}}Still alive.{{/if}}

{% endverbatim %}

#### 9. csrf\_token

这个标签用于跨站请求伪造保护

对于form表单等post请求，由于受到保护，所以不会显示。当使用csrf\_token标签时，在先发送get请求时，会在html加一个隐藏的input标签，获取一个码值，再发送post请求时，用于告诉服务端，这个post请求是安全的，使该页面显现。



post请求被隐藏的原因是因为防止浏览器一访问服务器就直接提交post的请求，如果第一次访问服务器上来就发送post请求，就不会获得该码值，该请求就会被拦截，防止页面攻击。

#### 10.load

用来加载模板文件或者过滤器

{% load ….. %}

### 过滤器

相当于模板文件中执行函数。

view文件可以通过传递参数的形式向模板传递函数，函数也可以在模板文件中执行（直接{{ 函数名 }}即可运行，不可以加括号），但是无法向该函数中传递参数，对象的方法可以使用本身自带的参数。其实这样失去了再模板中使用函数的意义，因为不可以向函数传递模板中参数，那直接在后台运行函数将结果传递给模板即可。

#### 1.过滤器使用方法

在模版中，有时候需要对一些数据进行处理以后才能使用。一般在Python中我们是通过函数的形式来完成的。而在模版中，则是通过过滤器来实现的。过滤器使用的是|来使用。比如使用add过滤器，那么示例代码如下：

{{ value | 过滤器名称:参数 }}

过滤器最多有两个参数（含自动添加的value）

{{ value|add:"2" }} #django直接将value传递给add，如果想传递其他参数，还可以在add后面添加，后面只能添加一个参数。过滤器最多有两个参数。 add(value,”2”)

那么以下就讲下在开发中常用的过滤器。

#### 2.常用过滤器

##### 2.add

将传进来的参数添加到原来的值上面。这个过滤器会尝试将值和参数转换成整形然后进行相加。如果转换成整形过程中失败了，那么会将值和参数进行拼接。如果是字符串，那么会拼接成字符串，如果是列表，那么会拼接成一个列表。示例代码如下：

{{ value|add:"2" }}

如果value是等于4，那么结果将是6。如果value是等于一个普通的字符串，比如abc，那么结果将是abc2。add过滤器的源代码如下：

def add(value, arg):

"""Add the arg to the value."""

try:

return int(value) + int(arg) #能转换为整型的情况下，转为整型

except (ValueError, TypeError):

try:

return value + arg #如果不能，则进行字符串拼接或者列表拼接

except Exception:

return ''

对于列表拼接：要求value 和 arg 都必须是列表。如果只有一个为列表，将会返回‘’

##### 3.cut

移除值中所有指定的字符串。类似于python中的replace(args,"")。示例代码如下：

{{ value|cut:" " }}

直接将制定的字符移除掉

以上示例将会移除value中所有的空格字符。cut过滤器的源代码如下：

def cut(value, arg):

"""Remove all values of arg from the given string."""

safe = isinstance(value, SafeData)

value = value.replace(arg, '')

if safe and arg != ';':

return mark\_safe(value)

return value

##### 4. default

如果值被评估为False。比如[]，""，None，{}等这些在if判断中为False的值，都会使用default过滤器提供的默认值。示例代码如下：

{{ value|default:"nothing" }}

如果value是等于一个空的字符串。比如""，那么以上代码将会输出nothing。

##### 5.default\_if\_none

如果值是None，那么将会使用default\_if\_none提供的默认值。这个和default有区别，default是所有被评估为False的都会使用默认值。而default\_if\_none则只有这个值是等于None的时候才会使用默认值。示例代码如下：

{{ value|default\_if\_none:"nothing" }}

如果value是等于""也即空字符串，那么以上会输出空字符串。如果value是一个None值，以上代码才会输出nothing。

##### 6. date

将一个日期按照指定的格式，格式化成字符串。示例代码如下：

# 数据

context = {

"birthday": datetime.now()

}

# 模版

{{ birthday|date:"Y/m/d" }}

那么将会输出`2018/02/01`。其中`Y`代表的是四位数字的年份，`m`代表的是两位数字的月份，`d`代表的是两位数字的日。

还有更多时间格式化的方式。见下表。



##### 7.first

返回列表/元组/字符串中的第一个元素。示例代码如下：

{{ value|first }}

如果value是等于['a','b','c']，那么输出将会是a。

##### 8.last

返回列表/元组/字符串中的最后一个元素。示例代码如下：

{{ value|last }}

如果value是等于['a','b','c']，那么输出将会是c。

##### 9.floatformat

使用四舍五入的方式格式化一个浮点类型。如果这个过滤器没有传递任何参数。那么只会在小数点后保留一个小数，如果小数后面全是0，那么只会保留整数。当然也可以传递一个参数，标识具体要保留几个小数。

###### 如果没有传递参数：

保留一位小数，如果小数全是0，则保留整数。

value 模版代码 输出

34.23234 {{ value|floatformat }} 34.2

34.000 {{ value|floatformat }} 34

34.260 {{ value|floatformat }} 34.3

###### 如果传递参数：

value 模版代码 保留三位小数 输出

34.23234 {{ value|floatformat:3 }} 34.232

34.000 {{ value|floatformat:3 }} 34.000

34.260 {{ value|floatformat:3 }} 34.260

##### 10.join

类似与Python中的join，将列表/元组/字符串用指定的字符进行拼接。示例代码如下：

{{ value|join:"/" }}

如果value是等于['a','b','c']，那么以上代码将输出a/b/c。

##### 11.length

获取一个列表/元组/字符串/字典的长度。示例代码如下：

{{ value|length }}

如果value是等于['a','b','c']，那么以上代码将输出3。如果value为None，那么以上将返回0。

##### 12.lower

将值中所有的字符全部转换成小写。示例代码如下：

{{ value|lower }}

如果value是等于Hello World。那么以上代码将输出hello world。

##### 13.upper

类似于lower，只不过是将指定的字符串全部转换成大写。

##### 14.random

在被给的列表/字符串/元组中随机的选择一个值。示例代码如下：

{{ value|random }}

如果value是等于['a','b','c']，那么以上代码会在列表中随机选择一个。

##### 15.safe

标记一个字符串是安全的。也即会关掉这个字符串的自动转义。示例代码如下：

{{value|safe}}

如果value是一个不包含任何特殊字符的字符串，比如<a>这种，那么以上代码就会把字符串正常的输入。如果value是一串html代码，那么以上代码将会把这个html代码渲染到浏览器中。

##### 16.slice

类似于Python中的切片操作。示例代码如下：

{{ some\_list|slice:"2:4" }}

以上代码将会给some\_list从2开始做切片操作。

##### 17.stringtags

删除字符串中所有的html标签。示例代码如下：

{{ value|striptags }}

如果value是<strong>hello world</strong>，那么以上代码将会输出hello world。

##### 18.truncatechars

如果给定的字符串长度超过了过滤器指定的长度。那么就会进行切割，并且会拼接三个点来作为省略号。示例代码如下：

{{ value|truncatechars:5 }}

如果value是等于北京欢迎您~，那么输出的结果是北京...。可能你会想，为什么不会北京欢迎您...呢。因为三个点也占了三个字符，所以北京+三个点的字符长度就是5。

##### 19.truncatechars\_html

类似于truncatechars，只不过是不会切割html标签。示例代码如下：

{{ value|truncatechars:5 }}

如果value是等于<p>北京欢迎您~</p>，那么输出将是<p>北京...</p>。

#### 3.自定义过滤器

1. 首先在某个app中，创建一个python包（含有\_\_init\_\_的文件夹），叫做`templatetags`，注意，这个包的名字必须为`templatetags`，不然就找不到。

2. 在这个`templatetags`包下面，创建一个python文件用来存储过滤器。

3. 在新建的python文件中，定义过滤器（也就是函数），这个函数的第一个参数永远是被过滤的那个值，并且如果在使用过滤器的时候传递参数，那么还可以定义另外一个参数。但是过滤器最多只能有2个参数。

4. 在写完过滤器（函数）后，要使用`django.template.Library.filter`进行注册。

5. 还要把这个过滤器所在的这个app添加到`settings.INSTALLED\_APS`中，不然Django也找不到这个过滤器。

6. 在模板中使用`load`标签加载过滤器所在的python包。

7. 可以使用过滤器了。

#此filter写在名为appname/templatetags/my\_filter.py的python文件中

from django import template

#register 注册器，名称固定  
register = template.Library()

#过滤器主体  
def my\_fun(value):  
 return type(value)

#对过滤器进行注册  
register.filter('back\_type',my\_fun) #此处back\_type 为定义的过滤器名称，可以与函数名相同，也可以不同，最后调用的都是这个myfun函数。

在模板中使用时，先使用load加载过滤器所在的文件，然后使用过滤器

{% load my\_filter %}  
 <p>  
 变量的类型是{{ a | back\_type }}  
 </p>

8. `django.template.Library.filter`还可以当作装饰器来使用。如果`filter`函数没有传递任何参数，那么将会使用这个函数的名字来作为过滤器的名字。当然如果你不想使用函数的名字来作为过滤器的名字，也可以传递一个`name`参数。示例代码如下：

```python

#此装饰器即可注册该过滤器，不用再单独注册，如果括号中参数省略，则过滤器的名称为函数名

@register.filter('my\_greet')

def greet(value,word):

return value + word

```

#### 自定义时间计算过滤器：

有时候经常会在朋友圈、微博中可以看到一条信息发表的时间，并不是具体的时间，而是距离现在多久。比如刚刚，1分钟前等。这个功能DTL是没有内置这样的过滤器的，因此我们可以自定义一个这样的过滤器。示例代码如下：

# time\_filter.py文件

from datetime import datetime

from django import template

register = template.Library()

def time\_since(value):

"""

time距离现在的时间间隔

1. 如果时间间隔小于1分钟以内，那么就显示“刚刚”

2. 如果是大于1分钟小于1小时，那么就显示“xx分钟前”

3. 如果是大于1小时小于24小时，那么就显示“xx小时前”

4. 如果是大于24小时小于30天以内，那么就显示“xx天前”

5. 否则就是显示具体的时间 2017/10/20 16:15

"""

if isinstance(value,datetime):

now = datetime.now()

timestamp = (now - value).total\_seconds()

if timestamp < 60:

return "刚刚"

elif timestamp >= 60 and timestamp < 60\*60:

minutes = int(timestamp / 60)

return "%s分钟前" % minutes

elif timestamp >= 60\*60 and timestamp < 60\*60\*24:

hours = int(timestamp / (60\*60))

return "%s小时前" % hours

elif timestamp >= 60\*60\*24 and timestamp < 60\*60\*24\*30:

days = int(timestamp / (60\*60\*24))

return "%s天前" % days

else:

return value.strftime("%Y/%m/%d %H:%M")

else:

return value

register.filter("time\_since",time\_since)

在模版中使用的示例代码如下：

{% load time\_filter %}

...

{% value|time\_since %}

...

### 模板结构优化include/extends

#### 1.引入模版include标签

有时候一些代码是在许多模版中都用到的。如果我们每次都重复的去拷贝代码那肯定不符合项目的规范。一般我们可以把这些重复性的代码抽取出来，就类似于Python中的函数一样，以后想要使用这些代码的时候，就通过include包含进来。这个标签就是include。示例代码如下：

# header.html

<p>我是header</p>

# footer.html

<p>我是footer</p>

# main.html

{% include 'header.html' %}

<p>我是main内容</p>

{% include 'footer.html' %}

include标签寻找路径的方式。也是跟render渲染模板的函数是一样的。

参数引用：

默认include标签包含模版，会自动的使用主模版中的上下文，也即可以自动的使用主模版中的变量。

也可以手动传递参数。如果想传入一些其他的参数，那么可以使用with语句。示例代码如下：

# header.html

<p>用户名：{{ username }}</p>

# main.html

{% include "header.html" with username='huangyong' %}

如果传递多个参数，用空格隔开即可：

{% include 'header.html' with username='user' pwd='111'%}

引入模板中需要变量，如果在include时给定，则优先使用给定的变量。如果没给定，则会寻找主模板是否有相同变量名的变量，如果有，则使用，如果没有，则空白。

#### 2.模板继承

在前端页面开发中。有些代码是需要重复使用的。这种情况可以使用include标签来实现。也可以使用另外一个比较强大的方式来实现，那就是模版继承。模版继承类似于Python中的类，在父类中可以先定义好一些变量和方法，然后在子类中实现。模版继承也可以在父模版中先定义好一些子模版需要用到的代码，然后子模版直接继承就可以了。并且因为子模版肯定有自己的不同代码，因此可以在父模版中定义block接口，然后子模版再去实现。建立一个模板文件，并预留扩展接口。

以下是父模版的代码：

{% load static %}

<!DOCTYPE html>

<html lang="en">

<head>

<link rel="stylesheet" href="{% static 'style.css' %}" />

<title>{% block title %}我的站点{% endblock %}</title>

</head>

<body>

<div id="sidebar">

{% block sidebar %}

<ul>

<li><a href="/">首页</a></li>

<li><a href="/blog/">博客</a></li>

</ul>

{% endblock %}

</div>

<div id="content">

{% block content %}{% endblock %}

</div>

</body>

</html>

这个模版，我们取名叫做base.html，定义好一个简单的html骨架，然后定义好两个block接口，让子模版来根据具体需求来实现。子模板然后通过extends标签来实现，示例代码如下：

{% extends "base.html" %}

{% block title %}博客列表{% endblock %}

{% block content %}

{% for entry in blog\_entries %}

<h2>{{ entry.title }}</h2>

<p>{{ entry.body }}</p>

{% endfor %}

{% endblock %}

**需要注意的是：extends标签必须放在模版的第一行。**  
**子模板中的代码必须放在block中，否则将不会被渲染。**  
如果在某个block中需要使用父模版的内容，那么可以使用{{block.super}}来继承。比如上例，{%block title%}，如果想要使用父模版的title，那么可以在子模版的title block中使用{{ block.super }}来实现。

如果不对原接口进行补充，则模板会自动使用该接口内的内容。

如果对接口进行补充。会把原接口内的内容覆盖掉。

{% block 接口名 %}

加入html标签内容

{% endblock %}

如果不想覆盖原接口内的内容，保留原内容，并进行添加：使用block.super

{% block 接口名 %}

{{ block.super }}

加入html标签内容

{% endblock %}

在定义block的时候，除了在block开始的地方定义这个block的名字，还可以在block结束的时候定义名字。

比如{% block title %}{% endblock title %}。

这在大型模版中显得尤其有用，能让你快速的看到block包含在哪里。

### 加载静态文件css/js/img等文件

第2点中有对静态文件地址本质进行解释。

在一个网页中，不仅仅只有一个html骨架，还需要css样式文件，js执行文件以及一些图片等。因此在DTL中加载静态文件是一个必须要解决的问题。在DTL中，使用static标签来加载静态文件。要使用static标签，首先需要{% load static %}。加载静态文件的步骤如下：

1.首先确保django.contrib.staticfiles已经添加到settings.INSTALLED\_APPS中。

2.确保在settings.py中设置了STATIC\_URL。STATIC\_URL是用来指定浏览器请求静态文件的url。如服务器是127.0.0.1，则浏览器请求静态文件的地址为：

127.0.0.1/STATIC\_URL/文件名 注意STATIC\_URL是个变量名，其值代表静态文件地址在前端的代号

STATIC\_URL = '/static/'#由于静态文件夹都是默认在app下的名叫static文件夹。django会自动存放静态文件夹列表。此处定义的 STATIC\_URL是静态文件夹在前端浏览器中叫什么名字。可以随意起名字。如STATIC\_URL=’/abc’,那么，只要是浏览器下载静态文件时，碰到/abc/这个目录，即默认为是static静态文件地址。会往存放静态文件夹列表地址的列表中依次寻找。

如果存放静态文件的文件夹不在app下，或者不叫static，则可以通过STATICFILES\_DIRS = [

os.path.join(BASE\_DIR,"静态文件相对地址")

]

来将该地址放入地址存放列表中。

其实STATIC\_URL的值就是静态文件地址在前端的一个代号。如果碰到该代号，django会就自动往存放静态文件地址的列表中迭代，选择经静态文件。

存放静态文件的列表中的地址默认为app/static。可以通过STATICFILES\_DIRS添加。

如果不同app的静态文件中有同名的文件，则django寻找的时候，会按照静态地址列表中地址的顺序依次进行寻找，只要找到一个，就不会再往下进行寻找。因此，相同名称的文件之后加载一个。所有有相同文件的话，尽量改下名字或者在静态文件目录下再建立一个不同名称的字文件夹（如app名称来命名）

3.在已经安装了的app下创建一个文件夹叫做static，然后再在这个static文件夹下创建一个当前app的名字的文件夹，再把静态文件放到这个文件夹下。例如你的app叫做book，有一个静态文件叫做zhiliao.jpg，那么路径为book/static/book/zhiliao.jpg。（为什么在app下创建一个static文件夹，还需要在这个static下创建一个同app名字的文件夹呢？原因是如果直接把静态文件放在static文件夹下，那么在模版加载静态文件的时候就是使用zhiliao.jpg，如果在多个app之间有同名的静态文件，这时候可能就会产生混淆。而在static文件夹下加了一个同名app文件夹，在模版中加载的时候就是使用app/zhiliao.jpg，这样就可以避免产生混淆。）

4.如果有一些静态文件是不和任何app挂钩的。那么可以在settings.py中添加STATICFILES\_DIRS，以后DTL就会在这个列表的路径中查找静态文件。比如可以设置为:

1. STATICFILES\_DIRS = [
2. os.path.join(BASE\_DIR,"static")
3. ]

5.使用

方法一：

直接使用STATICFILES\_DIRS的内容。

STATICFILES\_DIRS=’/static/’

如果需要在前端加载静态文件，直接用STATICFILES\_DIRS的内容代替静态文件地址。

src=’static/11.jpg’ 此处的’static/’就是STATICFILES\_DIRS的内容。

不推荐的原因是，如果STATICFILES\_DIRS的内容改变，则后续的修改需要一个src一个src的修改。

方法二：

在模版中使用load标签加载static标签。比如要加载在项目的static文件夹下的style.css的文件。那么示例代码如下：

此处的static其实又是STATICFILES\_DIRS的一个别名，代表的就是STATICFILES\_DIRS的内容，不管STATICFILES\_DIRS怎么变，都会自动加载正确的。

{% load static %}

<link rel="stylesheet" href="{% static 'style.css' %}">

6.如果不想每次在模版中加载静态文件都使用load加载static标签，那么可以在settings.py中的TEMPLATES/OPTIONS添加'builtins':['django.templatetags.static']，这样以后在模版中就可以直接使用static标签，而不用手动的load了。

7.如果没有在settings.INSTALLED\_APPS中添加django.contrib.staticfiles。那么我们就需要手动的将请求静态文件的url与静态文件的路径进行映射了。示例代码如下：

1. from django.conf import settings
2. from django.conf.urls.static import static
3. urlpatterns = [
4. # 其他的url映射

] + static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)

## 数据库/模型层

### Django配置连接数据库

在操作数据库之前，首先先要连接数据库。这里我们以配置MySQL为例来讲解。Django连接数据库，不需要单独的创建一个连接对象。只需要在settings.py文件中做好数据库相关的配置就可以了。示例代码如下：

DATABASES = {

'default': {

# 数据库引擎（是mysql还是oracle等）

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

# 数据库的名字

'NAME': 'dfz',

# 连接mysql数据库的用户名

'USER': 'root',

# 连接mysql数据库的密码

'PASSWORD': 'root',

# mysql数据库的主机地址

'HOST': '127.0.0.1',

# mysql数据库的端口号

'PORT': '3306',

}

}

查看django支持什么类型的数据库，可以通过Lib/sitepackage/django/db/backends 下的文件夹名来看。

### 使用原生sql语句操作数据库

#### 原生sql接口介绍

在Django中操作数据库有两种方式。第一种方式就是使用原生sql语句操作，第二种就是使用ORM模型来操作。这节课首先来讲下第一种。

在Django中使用原生sql语句操作其实就是使用python db api的接口来操作。如果你的mysql驱动使用的是pymysql，那么你就是使用pymysql来操作的，只不过Django将数据库连接的这一部分封装好了（django已经通过setting连接好了数据库，不用重新连接，直接获取cursor使用即可。其余的使用和pymysql一样。），我们只要在settings.py中配置好了数据库连接信息后直接使用Django封装好的接口就可以操作了。示例代码如下：

# 使用django封装好的connection对象，会自动读取settings.py中数据库的配置信息

from django.db import connection

# 获取游标对象

cursor = connection.cursor()

# 拿到游标对象后执行sql语句

cursor.execute("select \* from book")

# 获取所有的数据

rows = cursor.fetchall()

# 遍历查询到的数据

for row in rows:

print(row)

以上的execute以及fetchall方法都是Python DB API规范中定义好的。任何使用Python来操作MySQL的驱动程序都应该遵循这个规范。所以不管是使用pymysql或者是mysqlclient或者是mysqldb，他们的接口都是一样的。更多规范请参考：<https://www.python.org/dev/peps/pep-0249/。>

使用pymysql：

import pymysql  
  
user=input('user>>: ').strip()  
pwd=input('password>>: ').strip()  
  
# 建立链接  
conn=pymysql.connect(  
 host='192.168.10.15',  
 port=3306,  
 user='root',  
 password='123',  
 db='db9',#数据库名称  
 charset='utf8'  
)  
  
# 拿到游标  
cursor=conn.cursor()  
  
# 执行sql语句 两种方法，推荐下面一种  
# sql='select \* from userinfo where user = "%s" and pwd="%s"' %(user,pwd)  
# print(sql)

#查询  
sql='select \* from userinfo where user = %s and pwd=%s'  
rows=cursor.execute(sql,(user,pwd)) #返回的是获取数据的条数

#此时数据存储于cursor中，cursor游标指向第一条数据

#返回查询到的第一条数据，游标下移1.

print(cursor.fetchone())

#返回当前游标指向开始的两条数据，游标下移2

print(cursor.fetchmany(2))

#返回从游标指向开始的后面所有数据

print(cursor.fetchall())

#移动游标至第几条数据 第一个索引为0

cursor.scroll(0, mode='absolute')

print(cursor.fetchone()) #此时又重新显示第一条数据

#游标相对于当前位置 向前或者向后移动，可以为负数，但是如果结果超出索引，将会报错

cursor.scroll(-2,mode='relative')

print(cursor.fetchone())

#插入多行数据 executemany

sql='insert into userinfo(user,pwd) values(%s,%s)'  
rows=cursor.executemany(sql,[('egon3','123'),('egon4','111'),('egon5','2222')])

conn.commit()

cursor.close() #关闭当前游标  
conn.close() #关闭当前连接。

#### Python DB API下规范下cursor对象常用接口：

1.description：如果cursor执行了查询的sql代码。那么读取cursor.description属性的时候，将返回一个列表，这个列表中装的是元组，元组中装的分别是(name,type\_code,display\_size,internal\_size,precision,scale,null\_ok)，其中name代表的是查找出来的数据的字段名称，其他参数暂时用处不大。

2.rowcount：代表的是在执行了sql语句后受影响的行数。

3.close：关闭游标。关闭游标以后就再也不能使用了，否则会抛出异常。

4.execute(sql[,parameters])：执行某个sql语句。如果在执行sql语句的时候还需要传递参数，那么可以传给parameters参数。示例代码如下：

cursor.execute("select \* from article where id=%s",(1,))

5.fetchone：在执行了查询操作以后，获取第一条数据。

6.fetchmany(size)：在执行查询操作以后，获取多条数据。具体是多少条要看传的size参数。如果不传size参数，那么默认是获取第一条数据。

7.fetchall：获取所有满足sql语句的数据。

### ORM模型

#### 创建ORM模型

## 视图：

视图一般都写在app的views.py中。并且视图的第一个参数永远都是request（一个HttpRequest）对象。这个对象存储了请求过来的所有信息，包括携带的参数以及一些头部信息等。在视图中，一般是完成逻辑相关的操作。比如这个请求是添加一篇博客，那么可以通过request来接收到这些数据，然后存储到数据库中，最后再把执行的结果返回给浏览器。视图函数必须要返回结果，且返回结果必须是HttpResponseBase对象或者子类的对象。示例代码如下：

from django.http import HttpResponse

def book\_list(request):

return HttpResponse("书籍列表！")

如果较大型的网址，需要视图函数较多，可以分模块建立视图函数文件package（不是directory），相应文件导入到urls中即可。其实就是app的雏形。