# Django 基础教程

## Django概述

### Django简介

Django是一个开放源代码的Web应用框架，由Python写成。采用了MVC的软件设计模式，即模型M，视图V和控制器C。它最初是被开发来用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的。

Django的主要目标是使得开发复杂的、数据库驱动的网站变得简单。Django注重组件的重用性和“可插拔性”，敏捷开发和DRY法则（Don't Repeat Yourself）。在Django中Python被普遍使用，甚至包括配置文件和数据模型。

在Python各种web框架中，Django的文档最完善、市场占有率最高、招聘职位最多！

* **优点：**
* 完美的文档，Django的成功，我觉得很大一部分原因要归功于Django近乎完美的官方文档（包括Djangobook）。
* 全套的解决方案，Django象Rails一样，提供全套的解决方案（full-stackframework + batteries included），基本要什么有什么（比如：cache、session、feed、orm、geo、auth），而且全部Django自己造，开发网站应手的工具Django基本都给你做好了，因此开发效率是不用说的，出了问题也算好找，不在你的代码里就在Django的源码里。
* 强大的URL路由配置，Django让你可以设计出非常优雅的URL，在Django里你基本可以跟丑陋的GET参数说拜拜。
* 自助管理后台，admin interface是Django里比较吸引眼球的一项contrib，让你几乎不用写一行代码就拥有一个完整的后台管理界面。
* **缺点：（**源自Django坚持自己造所有的轮子，整个系统相对封闭）
* 系统紧耦合，如果你觉得Django内置的某项功能不是很好，想用喜欢的第三方库来代替是很难的，比如下面将要说的ORM、Template。要在Django里用SQLAlchemy或Mako几乎是不可能，即使打了一些补丁用上了也会让你觉得非常非常别扭。
* Django自带的ORM远不如SQLAlchemy强大，除了在Django这一亩三分地，SQLAlchemy是Python世界里事实上的ORM标准，其它框架都支持SQLAlchemy了，唯独Django仍然坚持自己的那一套。Django的开发人员对SQLAlchemy的支持也是有过讨论和尝试的，不过最终还是放弃了，估计是代价太高且跟Django其它的模块很难合到一块。
* Template功能比较弱，不能插入Python代码，要写复杂一点的逻辑需要另外用Python实现Tag或Filter。
* URL配置虽然强大，但全部要手写，这一点跟Rails的Conventionover configuration的理念完全相左，高手和初识Django的人配出来的URL会有很大差异。
* 让人纠结的auth模块，Django的auth跟其它模块结合紧密，功能也挺强的，就是做的有点过了，用户的数据库schema都给你定好了，这样问题就来了，比如很多网站要求email地址唯一，可schema里这个字段的值不是唯一的，纠结是必须的了。
* Python文件做配置文件，而不是更常见的ini、xml或yaml等形式。这本身不是什么问题，可是因为理论上来说settings的值是能够动态的改变的（虽然大家不会这么干），但这不是最佳实践的体现。

Django大包大揽，用它来快速开发一些Web运用是很不错的。如果你顺着Django的设计哲学来，你会觉得Django很好用，越用越爽；相反，你如果不能融入或接受Django的设计哲学，你用Django一定会很痛苦，趁早放弃的好。

* **Django特点：**
* 自带强大的后台功能

用python的类继承，几行代码就可以拥有一个丰富，动态的数据库操作接口（API），如果需要你也能执行SQL语句

* 优雅的网址

用正则匹配网址，传递到对应函数，随意定义，如你所想！

* 模板功能

强大，易扩展的模板系统，设计简易，代码，样式分开设计，更容易管理。

* 缓存系统

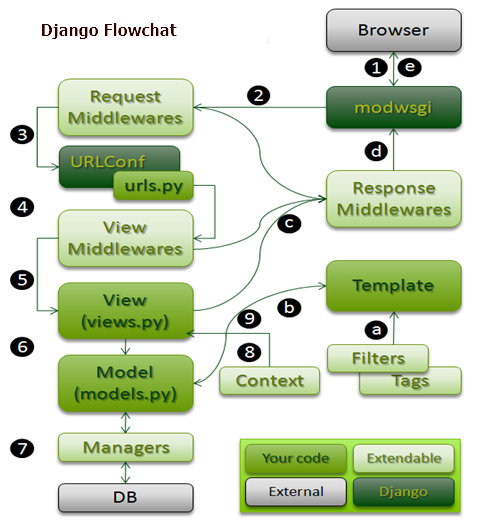
与memcached或其它的缓存系统联用，更出色的表现，更快的加载速度。

* 国际化

完全支持多语言应用，允许你定义翻译的字符，轻松翻译成不同国家的语言。

### Django运作流程

* **Django流程：**



1-1 Django流程图

* 1. 用户通过浏览器请求一个页面。
* 2. 请求到达Request Middlewares，中间件对request做一些预处理或者直接response请求
* 3. URLConf通过urls.py文件和请求的URL找到相应的View
* 4. View Middlewares被访问，它同样可以对request做一些处理或者直接返回response
* 5. 调用View中的函数
* 6.View中的方法可以选择性的通过Models访问底层的数据
* 7.所有的Model-to-DB的交互都是通过manager完成的

如果需要，Views可以使用一个特殊的Context

* 8.Context被传给Template用来生成页面
* **Django目录介绍**

这里介绍一个常见创建一个项目所包含目录的内容和作用：

Vdp(项目名)

├── db.sqlite3 （sqlite3数据库文件）

├── manage.py （项目管理命令文件）

├── static （静态文件存放目录）

├── media （媒体文件存放目录）

├── templates （模板html网页文件存放目录）

└── vdp （项目配置目录）

├── \_\_init\_\_.py

├── settings.py （项目配置文件）

├── urls.py （路由地址文件）

└── wsgi.py

firstapp(app名)

├── \_\_init\_\_.py

├── migrations （migration生成文件存放目录）

├── templatetags （自定义标签目录）

├── apps.py （app配置目录）

├── admin.py （管理后台文件）

├── models.py （模型数据文件）

├── tests.py （测试模块）

└── views.py （访问视图文件）

## 环境搭建

* **版本选择：**

Django 1.5.x 支持 Python 2.6.5 Python 2.7, Python 3.2 和 3.3.

Django 1.6.x 支持 Python 2.6.X, 2.7.X, 3.2.X 和 3.3.X

Django 1.7.x 支持 Python 2.7, 3.2, 3.3, 和 3.4 （注意：Python 2.6 不支持了）

**Django 1.8.x 支持 Python 2.7, 3.2, 3.3, 3.4 和 3.5.  （长期支持版本 LTS)**

Django 1.9.x 支持 Python 2.7, 3.4 和 3.5. 不支持 3.3 了

Django 1.10.x 支持 Python 2.7, 3.4 和 3.5.

**Django 1.11.x 下一个长期支持版本，将于2017年4月发布**

**更详细的可以**[**参考这里**](https://www.djangoproject.com/download/)**，**一般来说，选择长期支持版本比较好。

使用最新版本的问题就是，可能要用到的一些第三方插件没有及时更新，无法正常使用这些三方包。

如果是学习，可以选择目前的 Django 1.8.x 来进行，遇到问题也容易找到答案。

当然如果需要新版本的功能也可以使用新版本，毕竟 Django 1.9 以后admin界面还是更漂亮些。

* **安装Django**
* **Pip安装django**

（sudo) pip install Django  
或者 (sudo) pip install Django==1.8.16 或者 pip install Django==1.10.3

* **下载源码安装**

<https://www.djangoproject.com/download/>

如果是源码包, 比如 django-1.8.16.tar.gz

tar -xvf django-1.8.16.tar.gz  
cd django-1.8.16  
(sudo) python setup.py install

* **检查django版本**

终端上输入 python ,点击 Enter，进行 python 环境

>>> **import** django  
>>> django.VERSION  
(1, 8, 16, 'final', 0)  
>>>   
>>> django.get\_version()  
'1.8.16'

* **在虚拟环境下**

我们有的时候会发现，一个电脑上有多个项目，一个依赖 Django 1.8，另一个比较旧的项目又要用 Django 1.5，这时候怎么办呢？

我们需要一个依赖包管理的工具来处理不同的环境。

开发会用 virtualenv 来管理多个开发环境，virtualenvwrapper 使得virtualenv变得更好

用。

# 安装:  
(sudo) pip install virtualenv virtualenvwrapper

Linux/Mac OSX 下：

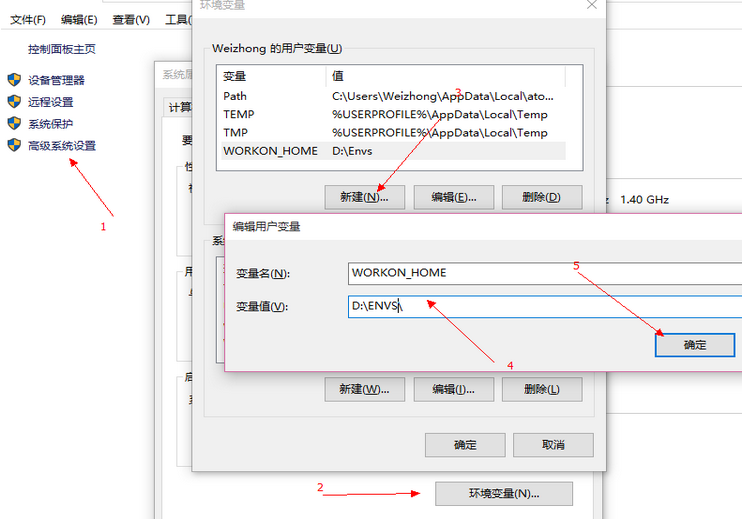
修改~/.bash\_profile或其它环境变量相关文件(如 .bashrc 或用 ZSH 之后的 .zshrc)，添加以下语句

export WORKON\_HOME=$HOME/.virtualenvs  
export PROJECT\_HOME=$HOME/workspace  
source /usr/local/bin/virtualenvwrapper.sh

source ~/.bash\_profile

windows下：

pip install virtualenvwrapper-win



使用方法：

**mkvirtualenv zqxt：创建运行环境zqxt**

**workon zqxt: 工作在 zqxt 环境 或 从其它环境切换到 zqxt 环境**

**deactivate**: 退出终端环境

**其它的：**

**rmvirtualenv** ENV：删除运行环境ENV

**mkproject** mic：创建mic项目和运行环境mic

**mktmpenv**：创建临时运行环境

**lsvirtualenv**: 列出可用的运行环境

**lssitepackages**: 列出当前环境安装了的包

创建的环境是独立的，互不干扰，无需sudo权限即可使用 pip 来进行包的管理。

## 基本命令

Django本身提供了许多基础命令，让我们快速的创建和操作项目。

* **新建项目**

django-admin.py startproject project-name  
特别是在 windows 上，如果报错，尝试用 django-admin 代替 django-admin.py 试试

一个 project 为一个项目，project-name 项目名称，改成你自己的，要符合Python 的变量命名规则（以下划线或字母开头）

* **新建app**

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。

* **同步数据库**

python manage.py syncdb  
  
注意：Django1.7.1及以上的版本需要用以下命令  
python manage.py makemigrations  
python manage.py migrate

这种方法可以创建表，当你在models.py中新增了类时，运行它就可以自动在数据库中创建表了，不用手动创建。

* **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，不要用在生产环境。

*python* manage.py runserver  
  
# 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）  
  
# 监听所有可用 ip （电脑可能有一个或多个内网ip，一个或多个外网ip，即有多个ip地址）  
*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

* **清空数据库**

*python* manage.py flush

* **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

* **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](http://www.ziqiangxuetang.com/django/django-import-data.html)[数据迁移](http://www.ziqiangxuetang.com/django/django-data-migration.html)，现在了解有这个用法就可以了。

* **django项目环境终端**

python manage.py shell

* **数据库命令行**

*python* manage.py dbshell

**更多命令查看**

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。

## 视图与网址

* **新建项目（名字为mysite）**

首先使用命令 django-admin startproject mysite 或 django-admin.py startproject mysite。新建出mysite的一个项目。

然后进入项目目录使用命令 python manager startapp learn 创建了一个learn的app。最终将得到一个项目目录，每个目录的含义可以参考上面的介绍。

在新创建的app，我们需要在项目的settings.py文件中加入到INSTALL\_APPS中，修改 mysite/mysite/settings.py

INSTALLED\_APPS = (  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles',  
  
 'learn',  
)

备注,这一步是干什么呢? 新建的 app 如果不加到 INSTALL\_APPS 中的话, django 就不能自动找到app中的**模板文件**(app-name/templates/下的文件)和**静态文件**(app-name/static/中的文件) , 后面你会学习到它们分别用来干什么.

* **定义视图函数（访问页面时的内容）**

我们在learn这个目录中,把views.py打开,修改其中的源代码,改成下面的

# coding:utf-8  
**from** django.http **import** HttpResponse  
  
  
**def index**(request):  
 **return** HttpResponse(u"第一个django web网站!")

第一行是声明编码为utf-8, 因为我们在代码中用到了中文,如果不声明就报错.

第二行引入HttpResponse，它是用来向网页返回内容的，就像Python中的 print 一样，只不过 HttpResponse 是把内容显示到网页上。

我们定义了一个**index()**函数，第一个参数必须是 **request**，与网页发来的请求有关，request 变量里面包含**get**或**post**的内容，用户浏览器，系统等信息在里面（后面会讲，先了解一下就可以）。

函数返回了一个 HttpResponse 对象，可以经过一些处理，最终显示几个字到网页上。

那问题来了，我们访问什么网址才能看到刚才写的这个函数呢？怎么让网址和函数关联起来呢？

* **定义视图函数相关的URL（网址）**

Django匹配路由地址是通过正则表达式去匹配的，我们打开 mysite/mysite/urls.py 这个文件, 修改其中的代码:

**from** django.conf.urls **import** url  
**from** django.contrib **import** admin  
**from** learn **import** views **as** learn\_views # new  
  
urlpatterns = [  
 url(r'^$', learn\_views.index), # new  
 url(r'^admin/', admin.site.urls),  
]

以上都修改并保存后,我们来看一下效果!

在终端上运行 **python manage.py runserver** 即可运行起开发测试环境，默认端口是8000，然后打开浏览器就可以访问到<http://127.0.0.1:8000/>

## 视图与网址进阶

为了了解django是怎么获取地址传参和获取，我们举个利用地址传参做加法运算的例子。

* **采用/add/?a=4&b=5这样GET方法进行**

在视图文件views.py中添加，

**from** django.shortcuts **import** render  
**from** django.http **import** HttpResponse  
  
  
**def add**(request):  
 a = request.GET['a']  
 b = request.GET['b']  
 c = int(a) + int(b)  
 **return** HttpResponse(str(c))

注：request.GET 类似于一个字典，更好的办法是用 request.GET.get('a', 0) 当没有传递 a 的时候默认 a 为 0，然后添加路由，

**from** django.conf.urls **import** url  
**from** django.contrib **import** admin  
**from** calc **import** views **as** calc\_views  
  
urlpatterns = [  
 url(r'^add/$', calc\_views.add, name='add'), # 注意修改了这一行  
 url(r'^admin/', admin.site.urls),  
]

* **采用/add/3/4这样的网址方式**

这种方式，我们可以增加一个视图函数，

**def add2**(request, a, b):  
 c = int(a) + int(b)  
 **return** HttpResponse(str(c))

然后在urls.py中添加新路由，

url(r'^add/(\d+)/(\d+)/$', calc\_views.add2, name='add2'),

然后我们再访问[http://127.0.0.1:8000/add/4/5/](http://127.0.0.1:8000/add2/4/5/)也可以看到同样的效果。

## Url name 详解

上面定义的路由中有个name参数，简单说，name 可以用于在 templates, models, views ……中得到对应的网址，相当于“给网址取了个名字”，只要这个名字不变，网址变了也能通过名字获取到。

* 在模板中使用url name

<a href="{% url 'add' %}”>相加</a>

* 在视图中使用url name

**from** django.http **import** HttpResponseRedirect  
**from** django.core.urlresolvers **import** reverse # django 1.4.x - django 1.10.x  
  
  
# from django.urls import reverse # new in django 1.10.x  
  
**def old\_add2\_redirect**(request, a, b):  
 **return** HttpResponseRedirect(  
 reverse('add2', args=(a, b))  
 )

urls.py中

url(r'^add/(\d+)/(\d+)/$', calc\_views.old\_add2\_redirect),  
url(r'^new\_add/(\d+)/(\d+)/$', calc\_views.add2, name='add2'),

当访问add/4/5时自动重定向到新的视图函数上了。

## 模板

视图函数可以返回一个html页面，通过render，如：

**from** django.shortcuts **import** render  
  
  
**def home**(request):  
 **return** render(request, 'home.html')

**Django 模板查找机制： Django 查找模板的过程是在每个 app 的 templates 文件夹中找（而不只是当前 app 中的代码只在当前的 app 的 templates 文件夹中找）。各个 app 的 templates 形成一个文件夹列表，Django 遍历这个列表，一个个文件夹进行查找，当在某一个文件夹找到的时候就停止，所有的都遍历完了还找不到指定的模板的时候就是 Template Not Found （过程类似于Python找包）。这样设计有利当然也有弊，有利是的地方是一个app可以用另一个app的模板文件，弊是有可能会找错了。所以我们使用的时候在 templates 中建立一个 app 同名的文件夹，这样就好了。**

一个网站一般存在一些通用的部分，如导航、底部、访问统计代码等，这样我们可以定义一个基础的base.html页面，让所有的页面都来继承它，子页面就都能获得父页面的东西了。例如：

<!DOCTYPE html>  
<html>  
<head>  
 <title>{% block title %}默认标题{% endblock %}</title>  
</head>  
<body>  
   
{% include 'nav.html' %}  
   
{% block content %}  
<div>这里是默认内容，所有继承自这个模板的，如果不覆盖就显示这里的默认内容。</div>  
{% endblock %}  
   
{% include 'bottom.html' %}  
   
{% include 'tongji.html' %}  
   
</body>  
</html>

然后我们的home.html就可以这样写：

{% extends 'base.html' %}  
   
{% block title %}欢迎光临首页{% endblock %}  
   
{% block content %}  
{% include 'ad.html' %}  
这里是首页，欢迎光临  
{% endblock %}

## 模板进阶

这里主要介绍Django模板中的循环，条件判断，常见的标签，过滤器的使用。

* **列表**，字典，类的实例的使用
* **循环**：迭代显示列表，字典等中的内容
* **条件判断**：判断是否显示该内容，比如判断是手机访问，还是电脑访问，给出不一样的代码。
* **标签**：for，if 这样的功能都是标签。
* **过滤器**：管道符号后面的功能，比如{{ var**|**length }}，求变量长度的 length 就是一个过滤器。

如果需要将一个或多个变量共享给多个网页或者所有网页使用，比如在网页上显示来访者的IP，这个可以使用 Django 上下文渲染器 来做。

### 循环与判断

* **显示一个基本的字符串在网页上**

Views.py中

# -\*- coding: utf-8 -\*-  
**from** django.shortcuts **import** render  
  
  
**def home**(request):  
 string = u"我在学习Django，用它来建网站"  
 **return** render(request, 'home.html', {'string': string})

home.html中写入{{ string }} 即可，便可看到。

* **基本的for循环和list内容的显示**

Views.py中

**def** home(request):  
 TutorialList = ["HTML", "CSS", "jQuery", "Python", "Django"]  
 **return** render(request, 'home.html', {'TutorialList': TutorialList})

home.html中

教程列表：  
{% for i **in** TutorialList %}  
{{ i }}  
{% endfor %}

**简单总结一下：**一般的变量之类的用 {{ }}（变量），功能类的，比如循环，条件判断是用 {%  %}（标签）

循环中的判断：

{% for item **in** List %}  
 {{ item }}{% if **not** forloop.last %},{% endif %}   
{% endfor %}

|  |  |
| --- | --- |
| forloop.counter | 索引从 1 开始算 |
| forloop.counter0 | 索引从 0 开始算 |
| forloop.revcounter | 索引从最大长度到 1 |
| forloop.revcounter0 | 索引从最大长度到 0 |
| forloop.first | 当遍历的元素为第一项时为真 |
| forloop.last | 当遍历的元素为最后一项时为真 |
| forloop.parentloop | 用在嵌套的 for 循环中，  获取上一层 for 循环的 forloop |

* **显示字典中的类容**

Views.py中

**def home**(request):  
 info\_dict = {'site': u'博客', 'content': u'欢迎参观'}  
 **return** render(request, 'home.html', {'info\_dict': info\_dict})

html.py中

站点：{{ info\_dict.site }} 内容：{{ info\_dict.content }}

模板取字典的值使用的是info\_dict.site而不能使用info\_dict[‘site’]

还可以这样来遍历：

{% for key, value **in** info\_dict.items %}  
 {{ key }}: {{ value }}  
{% endfor %}

* **模板中的逻辑操作**

  ==, !=, >=, <=, >, < 这些比较都可以在模板中使用，and, or, not, in, not in 也可以在模板中使用

* **在模板中获取request对象相关属性**

首先要确保设置文件中，模板中是否包含了django.template.context\_processors.request

TEMPLATES = [  
 {  
 'BACKEND': 'django.template.backends.django.DjangoTemplates',  
 'DIRS': [os.path.join(BASE\_DIR, 'templates')]  
 ,  
 'APP\_DIRS': True,  
 'OPTIONS': {  
 'context\_processors': [  
 'django.template.context\_processors.debug',  
 'django.template.context\_processors.request',  
 'django.contrib.auth.context\_processors.auth',  
 'django.contrib.messages.context\_processors.messages',  
 'blog.views.global\_setting',  
 ],  
 },  
 },  
]

* {{ request.user }} 当前的用户对象
* {{ request.user.username }} 当前用户的名字
* {{ request.path }} 当前网址
* {{ request.GET.urlencode }} 获取当前的GET参数

### 常用的内建模板标签

* {% autoescape %}{% endautoescape %} 自动转义所有变量内容会被转义HTML输出
* {%comment%}{%endcomment%} 之间的内容会被忽略
* {% csrf\_token %} 用户跨站请求伪造保护
* {%extends ‘base.html’ %} 继承一个父模板
* {%block%}{%endblock%} 一般在继承关系中使用，定义继承块
* {% include ‘other.html’ %} 引入其他模板
* {% load somelibrary/staticfiles %} 载入自定义的标签集/静态文件
* {% verbatim%} {%verbatim%} 停止渲染里面的{{}},防止与js冲突。

### 常用的内建模板过滤器

* {{ value | add:’2’}} 相加，如果value为4，输出为6
* {{value | addslashes }} 字符串中的引号加斜杠
* {{ value | capfirst }} 字符串的第一个字母大写
* {{ value | cut “ ” }} 切除字符串中的空格，（也可移除其他字符）
* {{value | date 'Y-m-d H:i:s' }} 格式化时间格式
* {{ value | default ‘nothing’ }} value为空或为假，输出默认值
* {{value | dictsort ‘key’}} 列表中的字典排序
* {{value | filesizeformat}} 将文件数值输出为可读格式，如4.2MB
* {{value | first }} 输出value的第一个元素
* {{value | join “//”}} value列表加入字符连接成字符串
* {{value | length}} 将输出value的长度
* {{value | linenumbers}} 将输出带行号的文本
* {{value | lower}} 将字符全部转化成小写

## 模型（数据库）

Django 模型是与数据库相关的，与数据库相关的代码一般写在 models.py 中，Django 支持 sqlite3, MySQL, PostgreSQL等数据库，只需要在settings.py中配置即可，不用更改models.py中的代码，丰富的API极大的方便了使用。

### 设置数据库

DATABASES\_OPTIONS = {  
 "MYSQL": {  
 'ENGINE': 'django.db.backends.mysql',  
 'HOST': '104.224.172.75',  
 'NAME': 'blog2',  
 'USER': 'blog2',  
 'PASSWORD': 'qwe123!@',  
 },  
 'SQLITE': {  
 'ENGINE': 'django.db.backends.sqlite3',  
 'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),  
 }  
}  
  
DATABASES = {  
 'default': DATABASES\_OPTIONS["SQLITE"]  
}

### 定义数据库

Django的数据库定义采用了ORM模型，所有的数据库定义都在models.py文件中。例如：

**from** django.db **import** models  
  
  
**class Person**(models.Model):  
 name = models.CharField(max\_length=30)  
 age = models.IntegerField()  
  
 **def** \_\_unicode\_\_(self):  
 # 在Python3中使用 def \_\_str\_\_(self)  
 **return** self.name

**class Meta**:  
 # abstract = True  
 # proxy = True   
 db\_table = 'Person'  
 ordering = ['name']

### 部分字段选项介绍

* null，是否存储空值，默认False
* blank，字段运行不填，默认False
* choices，值选择列表
* default，字段默认值
* help\_text，帮助内容
* primary\_key，字段为主键
* unique，字段的值唯一
* related\_name，外键反向名称
* related\_query\_name，外键关联过滤反向名称

### 部分字段类型

* AutoField，自增长字段，默认一个表会默认添加这个字段
* BigIntegerField，二进制字段
* BooleanField，布尔类型字段
* CharField，字符类型字段
* CommaSeparatedIntegerField，逗号分隔的整数字段
* DateField，日期字段，有个auto\_now选项，自动当前时间
* DateTimeField，时间日期字段
* DecimalField，十进制浮点数字段
* EmailFIeld，邮箱地址字段
* FileField，文件上传字段
* ImageField，图片字段
* IPAddressField，ip地址字段
* UUIDField，uuid字段
* ForeignField，外键（多对一）
* ManyToManyField，多对多关联
* OneToOneField，一对一关联

### 创建对象

同步数据库

python manage.py syncdb # 进入 manage.py 所在的那个文件夹下输入这个命令  
   
注意：Django 1.7 及以上的版本需要用以下命令  
python manage.py makemigrations  
python manage.py migrate

创建一个数据库对象数据，有四种方法：

* Person.objects.create(name=name,age=age)
* p = Person(name="WZ", age=23)

p.save()

* p = Person(name="TWZ")

p.age = 23

p.save()

* Person.objects.get\_or\_create(name="WZT", age=23)

这种方法是防止重复很好的方法，但是速度要相对慢些，返回一个元组，第一个为Person对象，第二个为True或False, 新建时返回的是True, 已经存在时返回False.

### 查询结果集

* Person.objects.all()
* Person.objects.row(“sql 语句”) # 执行自定义的sql语句
* Person.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
* Person.objects.get(name=name)

get是用来获取一个对象的，如果需要获取满足条件的一些人，就要用到filter

* Person.objects.filter(name="abc") # 等于Person.objects.filter(name\_\_exact="abc") 名称严格等于 "abc" 的人
* Person.objects.filter(name\_\_iexact="abc") # 名称为 abc 但是不区分大小写，可以找到 ABC, Abc, aBC，这些都符合条件
* Person.objects.filter(name\_\_contains="abc") # 名称中包含 "abc"的人
* Person.objects.filter(name\_\_icontains="abc") #名称中包含 "abc"，且abc不区分大小写
* Person.objects.filter(name\_\_regex="^abc") # 正则表达式查询
* Person.objects.filter(name\_\_iregex="^abc")# 正则表达式不区分大小写

filter是找出满足条件的，当然也有排除符合某条件的

* Person.objects.exclude(name\_\_contains="WZ") # 排除包含 WZ 的Person对象
* Person.objects.filter(name\_\_contains="abc").exclude(age=23) # 找出名称含有abc, 但是排除年龄是23岁的

更多参考：

### QuerySet Api

* 对查询结果排序

Author.objects.all().order\_by('name')  
Author.objects.all().order\_by('-name') # 在 column name 前加一个负号，可以实现倒序

* 链式查询

Author.objects.filter(name\_\_contains="Md").filter(email="mudong1991@163.com")  
Author.objects.filter(name\_\_contains="Mu").exclude(email="mudong1991@163.com")  
  
# 找出名称含有abc, 但是排除年龄是23岁的  
Person.objects.filter(name\_\_contains="abc").exclude(age=23)

* 不支持负索引

Person.objects.all()[:10]切片操作，前10条  
Person.objects.all()[-10:]会报错！！！  
  
# 1. 使用 reverse() 解决  
Person.objects.all().reverse()[:2] # 最后两条  
Person.objects.all().reverse()[0] # 最后一条  
  
# 2. 使用 order\_by，在栏目名（column name）前加一个负号  
Author.objects.order\_by('-id')[:20] # id最大的20条

* 多表查询

qs1 = Pathway.objects.filter(label\_\_name='x')  
qs2 = Pathway.objects.filter(reaction\_\_name='A + B >> C')  
qs3 = Pathway.objects.filter(inputer\_\_name='Md')  
  
# 合并到一起  
qs = qs1 | qs2 | qs3  
这个时候就有可能出现重复的  
  
# 去重方法  
qs = qs.distinct()

## QuerySet Api进阶

QuerySet 是惰性原则，也就是说，要用到或要去获取数据结果时时才会去真正查询数据库。

### 准备工作

在models.py中添加模型类

**from** \_\_future\_\_ **import** unicode\_literals  
  
**from** django.db **import** models  
**from** django.utils.encoding **import** python\_2\_unicode\_compatible

@python\_2\_unicode\_compatible  
**class Author**(models.Model):  
 name = models.CharField(max\_length=50)  
 qq = models.CharField(max\_length=20)  
 addr = models.TextField()  
 email = models.EmailField()  
  
 **def** \_\_str\_\_(self):  
 **return** self.name  
  
  
@python\_2\_unicode\_compatible  
**class Article**(models.Model):  
 title = models.CharField(max\_length=255)  
 author = models.ForeignKey(Author)  
 content = models.TextField()  
 score = models.IntegerField() # 文章的打分  
 tags = models.ManyToManyField('Tag')  
  
 **def** \_\_str\_\_(self):  
 **return** self.title  
  
  
@python\_2\_unicode\_compatible  
**class Tag**(models.Model):  
 name = models.CharField(max\_length=50)  
  
 **def** \_\_str\_\_(self):  
 **return** self.name

然后使用命令进行同步数据库：

python manage.py makemigrations  
python manage.py migrate

### 编写数据导入脚本

* **Django数据迁移自动导入数据**

如果希望数据在项目初始化数据库同步的时候，导入部分数据需要在，apps.py中添加：

**from** django.db.models.signals **import** post\_migrate  
**from** django.apps **import** AppConfig  
**from** firstapp.init\_permissions **import** thread\_init  
  
  
**class MyAppConfig**(AppConfig):  
 name = "firstapp" # app的名称  
  
 **def ready**(self):  
 # 通过django的signal来调用thread\_init方法  
 post\_migrate.connect(thread\_init, sender=self)

thread\_init就是数据库迁移的时候执行的函数。

* **手动脚本导入数据**

如果要手动导入数据，需要自己编写脚本：

**from** \_\_future\_\_ **import** unicode\_literals  
**import** os  
os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "django10.settings")  
**import** django  
django.setup()  
  
**import** random  
**from** firstapp.models **import** Author, Article, Tag  
  
  
author\_name\_list = ['Mudong', 'twz915', 'dachui', 'zhe', 'zhen']  
article\_title\_list = ['Django 基础', 'Python 基础', 'HTML 学习']  
  
  
**def create\_authors**():  
 **for** author\_name **in** author\_name\_list:  
 author, created = Author.objects.get\_or\_create(name=author\_name)  
 author.qq = ''.join([str(random.randint(1,9)) **for** \_ **in** range(9)])  
 author.addr = 'addr\_%s' % (random.randint(1, 3))  
 author.email = '%s@qq.com' % author.addr  
 author.save()  
  
  
**def create\_articles\_and\_tags**():  
 # 随机生成文章  
 **for** article\_title **in** article\_title\_list:  
 # 从文章标题中得到 tag  
 tag\_name = article\_title.split(' ', 1)[0]  
 tag, created = Tag.objects.get\_or\_create(name=tag\_name)  
  
 random\_author = random.choice(Author.objects.all())  
  
 **for** i **in** range(1, 11):  
 title = '%s\_%s' % (article\_title, i)  
 article, created = Article.objects.get\_or\_create(  
 title=title, defaults={  
 'author': random\_author, # 随机分配作者  
 'content': '%s 正文' % title,  
 'score': random.randrange(70, 101), # 随机给文章一个打分  
 }  
 )  
 article.tags.add(tag)  
  
  
**def main**():  
 create\_authors()  
 create\_articles\_and\_tags()  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()  
 **print**("Done!")

然后在shell面板中可以查看，数据是否导入成功：

*python manage.py shell*

*In* [1]: from blog.models import Article, Author, Tag  
*In* [2]: Article.objects.all()

### 查看Django queryset执行的SQL

*In* [1]: print str(Author.objects.all*()*.query)  
*In* [2]: print str(Author.objects.filter*(name="Mudong")*.query)

*'SELECT "firstapp\_author"."id", "firstapp\_author"."name", "firstapp\_author"."qq", "firstapp\_author"."addr", "firstapp\_author"."email" FROM "firstapp\_author"'*

### Values\_list获取元组形式结果

如果只需要获取某几个具体字段的结果，比如我们要获取作者的 name 和 qq

*In* [6]: authors = Author.objects.values\_list('name'*,* 'qq')  
*In* [7]: authors

也可以通过values属性，

*In* [8]:authors = Author.objects.values('name', 'qq')

区别就在于values是一个列表包含了字段的key-value形式字典元素，values\_list是一个列表只包含了value的元组。

如果只有一个字段可以指定flat=True

*In [9]: Author.objects.values\_list('name', flat=True)*

结果是一个列表，元素就是该查询该字段结果集。

### extra实现别名，条件，排序等

extra 中可实现别名，条件，排序等，后面两个用 filter, exclude 一般都能实现，排序用 order\_by 也能实现。我们主要看一下别名这个

*In* [44]: tags = Tag.objects.all()*.extra*(select=*{'tag\_name':* 'name'})

这个对应的sql就是: SELECT name AS tag\_name FROM blog\_tag;

这样的结果可以通过:tags[0].name和tags[0].tag\_name访问，想要去掉name属性，可以添加defer排除原来的name属性。

### annotate 聚合 计数，求和，平均数等

* **计数**

计算每个作者的文章数

In [66]: **from** django.db.models **import** Count  
In [66]: Article.objects.all().values('author\_id').annotate(count=Count('author\_id')).values('author\_\_name', 'count')

* **平均**

计算每个作者文章平均得分

In [6]: **from** django.db.models **import** Avg  
In [7]: Article.objects.values('author\_id').annotate(avg\_score=Avg('score')).values('author\_id', 'avg\_score')

* **求和**

计算作者所有文章总分

In [12]: **from** django.db.models **import** Sum  
In [13]: Article.objects.values('author\_\_name').annotate(sum\_score=Sum('score')).values('author\_\_name', 'sum\_score')

### ****select\_related 优化一对一，多对一查询****

在settings.py添加日志设置，来打印记录在数据库中执行的语句。

LOGGING = {  
 'version': 1,  
 'disable\_existing\_loggers': False,  
 'handlers': {  
 'console': {  
 'class': 'logging.StreamHandler',  
 },  
 },  
 'loggers': {  
 'django.db.backends': {  
 'handlers': ['console'],  
 'level': 'DEBUG' **if** DEBUG **else** 'INFO',  
 },  
 },  
}

当DEBUG为True时，就可以查看出django执行了什么sql语句了。

假如，我们取出10篇Django相关的文章，并需要用到作者的姓名

*In* [13]: articles = Article.objects.all()*[:10]  
In* [14]: a1 = articles[0] # 取第一篇  
*(0.000)* SELECT "blog\_article"*.*"id", "blog\_article"."title", "blog\_article"."author\_id", "blog\_article"."content", "blog\_article"."score" FROM "blog\_article" LIMIT 1; args=()  
*In* [15]: a1.title  
*Out[15]:* u'Django \u6559\u7a0b\_1'  
*In* [16]: a1.author\_id  
*Out[16]:* 5  
*In* [17]: a1.author.name # 再次查询了数据库，注意！！！  
*(0.000)* SELECT "blog\_author"*.*"id", "blog\_author"."name", "blog\_author"."qq", "blog\_author"."addr", "blog\_author"."email" FROM "blog\_author" WHERE "blog\_author"."id" = 5; args=(5,)  
*Out[17]:* u'zhen'

这样做的就查询了多次数据库，现在我们要一次查询出来，我可以通过select\_related来查询

*In* [18]: articles = Article.objects.all()*.select\_related*('author')*[:10]  
In* [19]: a1 = articles[0] # 取第一篇  
*(0.000)* SELECT "blog\_article"*.*"id", "blog\_article"."title", "blog\_article"."author\_id", "blog\_article"."content", "blog\_article"."score", "blog\_author"."id", "blog\_author"."name", "blog\_author"."qq", "blog\_author"."addr", "blog\_author"."email" FROM "blog\_article" INNER JOIN "blog\_author" ON ("blog\_article"*.*"author\_id" = "blog\_author"*.*"id") LIMIT *1*; args=()  
*In* [20]: a1.title  
*Out[20]:* u'Django \u6559\u7a0b\_1'  
*In* [21]: a1.author.name # 嘻嘻，没有再次查询数据库！！  
*Out[21]:* u'zhen'

### ****prefetch\_related 优化一对多，多对多查询****

和 select\_related 功能类似，但是实现不同。

select\_related 是使用 SQL JOIN 一次性取出相关的内容。

prefetch\_related 用于 一对多，多对多 的情况，这时 select\_related 用不了，因为当前一条有好几条与之相关的内容。

prefetch\_related是通过再**执行一条额外的SQL语句**，**然后用 Python 把两次SQL查询的内容关联（joining)到一起**

*In* [24]: articles = Article.objects.all()*.prefetch\_related*('tags')*[:10]  
In* [25]: articles  
*Out[25]:* (0.000) *SELECT* "blog\_article"."id", "blog\_article"."title", "blog\_article"."author\_id", "blog\_article"."content", "blog\_article"."score" FROM "blog\_article" LIMIT 10; args=()  
*(0.001)* SELECT ("blog\_article\_tags"."article\_id") AS "\_prefetch\_related\_val\_article\_id", "blog\_tag"."id", "blog\_tag"."name" FROM "blog\_tag" INNER JOIN "blog\_article\_tags" ON ("blog\_tag"*.*"id" = "blog\_article\_tags"*.*"tag\_id") WHERE "blog\_article\_tags"."article\_id" IN (1, *2,* 3, 4, 5, 6, 7, 8, 9, 10); args=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

## Django后台

Django自带了一个强大的后台管理系统，我们只需要添加少量的代码，就可以实现强大的功能，与后台相关的文件：admin.py

**from** django.contrib **import** admin  
**from** models **import** Article,Author,Tag  
  
# Register your models here.  
  
  
**class ArticleAdmin**(admin.ModelAdmin):  
 list\_display = ('title', 'author', 'content', 'score')  
  
  
admin.site.register(Article, ArticleAdmin)  
  
admin.site.register(Tag)  
  
admin.site.register(Author)

通过python manager.py createsuperuser 即可创建后台管理用户，然后登陆<http://127.0.0.1:8000/admin>即可进入后台。

Django自身的管理系统还不够美观，而Xadmin模块，就是针对admin进行了扩展，实现了更美观的后台管理系统，相关资料可以参考<http://sshwsfc.github.io/xadmin/>

## Django表单

### 表单的基本用法

* + 建立表单类

新建一个forms.py文件，添加：

**from** django **import** forms  
  
  
**class AddForm**(forms.Form):  
 a = forms.IntegerField(label='第一个数')  
 b = forms.IntegerField(label='第二个数')

* + 视图中使用

# coding:utf-8  
**from** django.shortcuts **import** render, render\_to\_response  
**from** django.http **import** HttpResponse  
**from** forms **import** AddForm  
# Create your views here.  
  
  
**def index**(request):  
 **if** request.method == 'POST':  
 form = AddForm(request.POST) # form 包含提交的数据  
 **if** form.is\_valid():# 如果提交的数据合法  
 a = form.cleaned\_data['a']  
 b = form.cleaned\_data['b']  
 **return** HttpResponse(str(int(a) + int(b)))  
 **else**:# 当正常访问时  
 form = AddForm()  
  
 **return** render(request, 'index.html', {'form': form})

* + 模板文件中使用

{% load staticfiles %}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>首页</title>  
</head>  
<body>  
 <form action="" method="POST">  
 {% csrf\_token %}  
 {{ form }}  
 <input type="submit" value="提交">  
 </form>  
</body>  
</html>

**Csrf\_token防止了夸张伪造**

**Django 的 forms 提供了：**

* 模板中表单的渲染
* 数据的验证工作，某一些输入不合法也不会丢失已经输入的数据。
* 还可以定制更复杂的验证工作，如果提供了10个输入框，必须必须要输入其中两个以上，在 forms.py 中都很容易实现

### 模板中渲染

对于<label>/<input> 对，还有几个输出选项：

{{ form.as\_table }} 以表格的形式将它们渲染在<tr> 标签中

{{ form.as\_p }}  将它们渲染在<p> 标签中

{{ form.as\_ul }} 将它们渲染在<li> 标签中

注意，你必须自己提供<table> 或<ul> 元素。

* **迭代表单中的字段**

如果你为你的表单使用相同的HTML，你可以使用{% for %} 循环迭代每个字段来减少重复的代码：

{% for field **in** form %}  
 <div class="fieldWrapper">  
 {{ field.errors }}  
 {{ field.label\_tag }} {{ field }}  
 </div>  
{% endfor %}

* + {{ field.label }} 字段的label
  + {{ field.value }} 字段的值
  + {{ field.html\_name }} 输入元素的name 属性中将使用的名称。
  + {{ field.help\_text }} 字段的帮助文档
  + {{ field.is\_hidden }} 字段隐藏

### 表单字段

* **字段核心参数**
  + rquired, 这个字段必须有值
  + lbel, 这个字段前显示的名称
  + wdget, 这个字段渲染表单的相关参数
  + help\_text, 这个字段的帮助信息
  + error\_message， 这个字段异常的异常信息
  + validators，这个的合法验证器

**class RegForm**(forms.Form):  
 *"""  
 注册表单  
 """* username = forms.CharField(widget=forms.TextInput(attrs={'id': 'user-username', 'class': 'form-control',  
 'placeholder': '用户名', 'required': 'required'}),  
 max\_length=50, error\_messages={'required': '用户名不能为空',})  
 password = forms.CharField(widget=forms.PasswordInput(attrs={'id': 'user-password', 'class': 'form-control',  
 'placeholder': '密码', 'required': 'required'}),  
 max\_length=30, error\_messages={'required': '密码不能为空',})  
 email = forms.EmailField(widget=forms.EmailInput(attrs={'id': 'user-email', 'class': 'form-control',  
 'placeholder': '邮箱', 'required': 'required'}),  
 max\_length=80, error\_messages={'required': '邮箱不能为空'})  
 avatar = forms.ImageField(widget=forms.FileInput(attrs={'id': 'user-avatar', 'accept':  
 'image/png, image/gif, image/jpg, image/bmp',  
 'placeholder': '头像'}),  
 max\_length=500, required=False)  
 url = forms.URLField(widget=forms.URLInput(attrs={'id': 'user-url', 'class': 'form-control', 'placeholder': '个人网址'}),  
 max\_length=100, required=False)

* **内建字段**

内建字段和models中的字段差不多。如CharField, IntegerField等

* **模型表单**

我们在创建一个表单类时，有时需要和数据中的一个表中的字段同步，这时我们可以方便的使用模型表单来创建表单类，如：

**from** django **import** forms  
**from** models **import** Author  
  
  
**class AuthorForm**(forms.ModelForm):  
 **class Meta**:  
 model = Author  
 fields = ['name', 'title', 'birth\_date']

**class ArticleForm**(forms.ModelForm):  
 authors =forms.ModelMultipleChoiceField(queryset=Author.objects.all()[:2])  
  
 **class Meta**:  
 model = Article  
 fields = '\_\_all\_\_'

## Django 配置

Django的配置文件在settings.py文件中，这里对比较常见和必须的配置项作一个说明。

* Debug

DEBUG = True  
TEMPLATE\_DEBUG = True

**DEBUG＝True**时，如果出现 bug 便于我们看见问题所在，但是部署时最好不要让用户看见bug的详情，可能一些不怀好心的人攻击网站，造成不必要的麻烦。

* ALLOWED\_HOSTS

ALLOWED\_HOSTS = ['\*']

ALLOWED\_HOSTS 允许你设置哪些域名可以访问，即使在 Apache 或 Nginx 等中绑定了，这里不允许的话，也是不能访问的。

当 **DEBUG=False**时，这个为必填项，如果不想输入，可以用 **ALLOW\_HOSTS = ['\*']** 来允许所有的。

* STATIC\_URL/STAIC\_ROOT

# Static files (CSS, JavaScript, Images)  
# https://docs.djangoproject.com/en/1.8/howto/static-files/  
  
STATIC\_URL = '/static/'  
  
# 当运行 python manage.py collectstatic 的时候  
# STATIC\_ROOT 文件夹 是用来将所有STATICFILES\_DIRS中所有文件夹中的文件，以及各app中static中的文件都复制过来  
# 把这些文件放到一起是为了用apache等部署的时候更方便  
STATIC\_ROOT = os.path.join(BASE\_DIR, 'collected\_static')  
  
# 其它 存放静态文件的文件夹，可以用来存放项目中公用的静态文件，里面不能包含 STATIC\_ROOT  
# 如果不想用 STATICFILES\_DIRS 可以不用，都放在 app 里的 static 中也可以  
STATICFILES\_DIRS = (  
 os.path.join(BASE\_DIR, "common\_static"),  
 '/path/to/others/static/', # 用不到的时候可以不写这一行  
)  
  
# 这个是默认设置，Django 默认会在 STATICFILES\_DIRS中的文件夹 和 各app下的static文件夹中找文件  
# 注意有先后顺序，找到了就不再继续找了  
STATICFILES\_FINDERS = (  
 "django.contrib.staticfiles.finders.FileSystemFinder",  
 "django.contrib.staticfiles.finders.AppDirectoriesFinder"  
)

static 是静态文件所有目录，比如 jquery.js, bootstrap.min.css 等文件。

一般来说我们只要把静态文件放在 APP 中的 static 目录下，部署时用 python manage.py collectstatic 就可以把静态文件收集到（复制到） STATIC\_ROOT 目录，但是有时我们有一些共用的静态文件，这时候可以设置 **STATICFILES\_DIRS** 另外弄一个文件夹。

* MEDIA\_URL/MEDIA\_ROOT

MEDIA\_URL = '/media/'  
MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

media文件夹用来存放用户上传的文件，与权限有关，详情见  [Django 静态文件](http://www.ziqiangxuetang.com/django/django-static-files.html) 和   [Django 部署](http://www.ziqiangxuetang.com/django/django-deploy.html)

* LOGGING日志配置
* DATABASE 数据库配置
* TEMPLATES 模板设置
* 其他

LANGUAGE\_CODE = 'zh-hans' # 设置语言  
  
TIME\_ZONE = 'Asia/Chongqing' # 设置时区

## Django 项目部署

部署参考：<http://blog.scmud.com/?p=103>

