**Django框架学习**

# Django介绍

## 简介

Python的WEB框架有Django、Tornado、Flask 等多种，Django相较与其他WEB框架其优势为：大而全，框架本身集成了ORM、模型绑定、模板引擎、缓存、Session等功能，是一个全能型框架用于开发快速，拥有自己的Admin数据管理后台，第三方工具齐全，性能折中。缺点：功能太多，数据分表复杂，高性能扩展复杂。

Django 是免费开源的！

## 安装

* 本课程基于 2.0.5
* 通过pip： pip install Django
* 在 pycharm 中，通过搜索 Django 进行安装
* 在官网：<https://www.djangoproject.com/download/> ， 下载 django-master.tar，解压后，在解压目录中 运行 cmd，然后通过命令： python setup.py install 安装
* 安装完成后，在 python 命令窗口中：输入

import django

django.get\_version()

可以查看版本号

## 文档

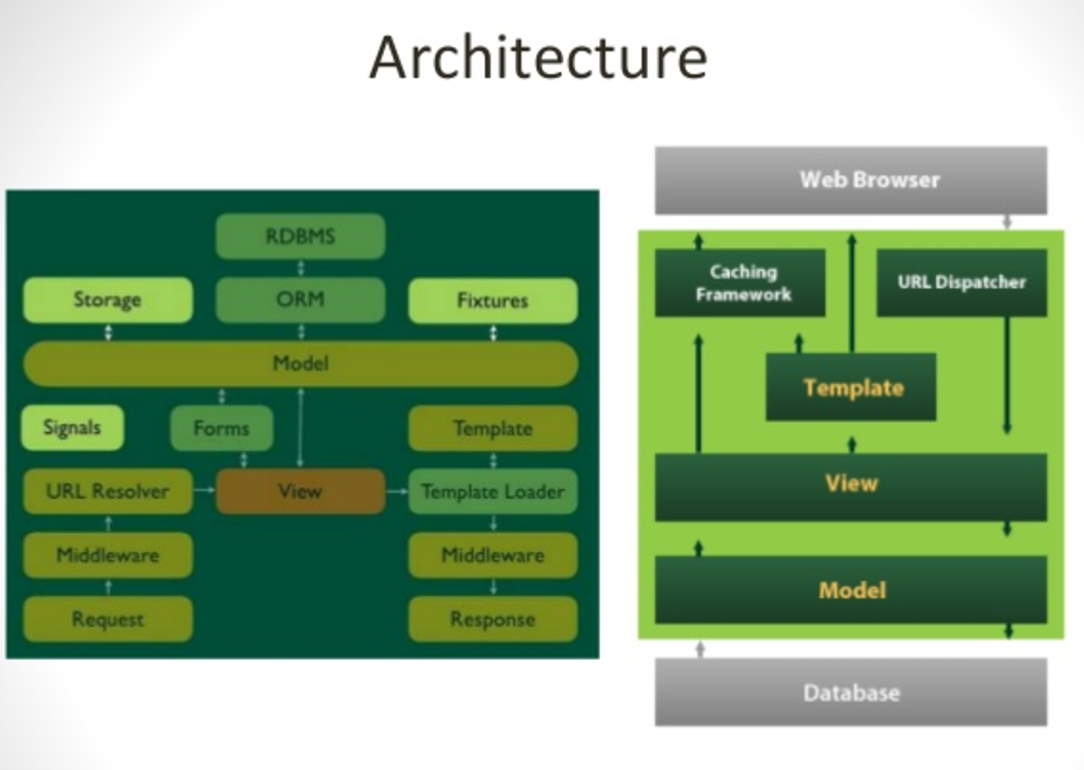
* 官方文档： <https://docs.djangoproject.com/en/2.0/> ，点击页面右下角的 language 按钮，可以选择 zh-hans ，显示中文

PS：只有部分页面有中文

* 1.8.2 中文文档： <http://python.usyiyi.cn/documents/django_182/index.html>

# Django相关知识学习

## Django架构



## 相关术语和规范

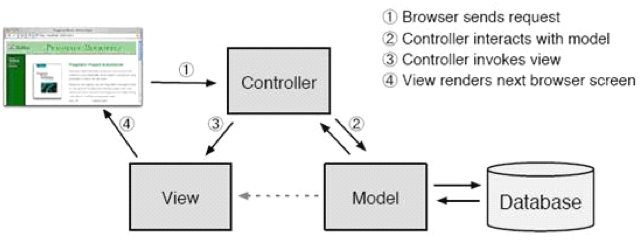
### MVC

MVC全名是Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

其中：

* M： 管理应用程序的状态（通常存储到数据库中），并约束改变状态的行为（或者叫做“业务规则”）。
* V： 负责把数据格式化后呈现给用户。
* C： 接受外部用户的操作，根据操作访问模型获取数据，并调用“视图”显示这些数据。控制器是将“模型”和“视图”隔离，并成为二者之间的联系纽带。

在《Agile Web Development with Rails》中有这样一张图对MVC模式进行了很好的解释：



### MTV

Django也是一个MVC框架。但是在Django中，控制器接受用户输入的部分由框架自行处理，所以 Django 里更关注的是模型（Model）、模板(Template)和视图（Views），称为 MTV模式。

其中：

* M： 代表模型（Model），即数据存取层。 该层处理与数据相关的所有事务： 如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。
* T： 代表模板(Template)，即表现层。 该层处理与表现相关的决定： 如何在页面或其他类型文档中进行显示。
* V： 代表视图（View），即业务逻辑层。 该层包含存取模型及调取恰当模板的相关逻辑。 你可以把它看作模型与模板之间的桥梁。

需要注意的是，不能简单的把 Django 视图认为是MVC控制器，把 Django 模板认为MVC视图。

区别在于：

Django 视图 不处理用户输入，而仅仅决定要展现哪些数据给用户；

Django 模板 仅仅决定如何展现Django视图指定的数据。

或者说, Django将MVC中的视图进一步分解为 Django视图 和 Django模板两个部分，分别决定 “展现哪些数据” 和 “如何展现”，使得Django的模板可以根据需要随时替换，而不仅仅限制于内置的模板。

### ORM

对象关系映射（英语：(Object Relational Mapping，简称ORM，或O/RM，或O/R mapping），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的--“虚拟对象数据库”。

ORM方法论基于三个核心原则：

* 简单：以最基本的形式建模数据。
* 传达性：数据库结构被任何人都能理解的语言文档化。
* 精确性：基于数据模型创建正确标准化的结构。

ORM是随着面向对象的软件开发方法发展而产生的。用来把对象模型表示的对象映射到基于S Q L 的关系模型数据库结构中去。这样，我们在具体的操作实体对象的时候，就不需要再去和复杂的 SQ L 语句打交道，只需简单的操作实体对象的属性和方法。O R M 技术是在对象和关系之间提供了一条桥梁，前台的对象型数据和数据库中的关系型的数据通过这个桥梁来相互转化。

# Django快速构建应用

## 命令行构建

打开命令行，cd 到一个你想放置你代码的目录，然后运行以下命令：

django-admin startproject mysite

这行代码将会在当前目录下创建一个 mysite 目录.

让我们看看 startproject 创建了些什么:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

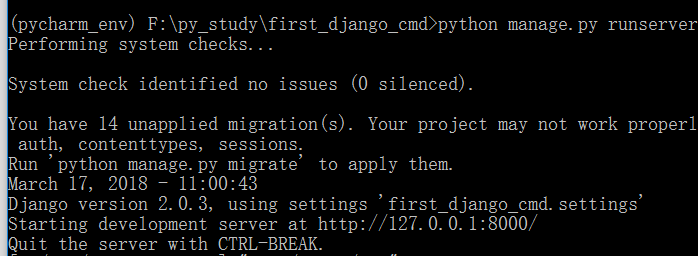
这些目录和文件的用处是：

* 最外层的:file: mysite/ 根目录只是你项目的容器， Django 不关心它的名字，你可以将它重命名为任何你喜欢的名字。
* manage.py: 一个让你用各种方式管理 Django 项目的命令行工具。. 你可以阅读 https://docs.djangoproject.com/zh-hans/2.0/ref/django-admin/ 获取更多细节。
* 里面一层的 mysite/ 目录包含你的项目 它是一个纯 Python 包。它的名字就是当你引用它内部任何东西时需要用到的 Python 包名。
* mysite/\_\_init\_\_.py：一个空文件，告诉 Python 这个目录应该被认为是一个 Python 包。
* mysite/settings.py : Django 项目的配置文件。请查看 https://docs.djangoproject.com/zh-hans/2.0/topics/settings/ 了解更多细节。
* mysite/urls.py: Django 项目的 URL 声明，就像你网站的 "目录" 阅读 <https://docs.djangoproject.com/zh-hans/2.0/topics/http/urls/> 获取更多关于 URL 的内容。.
* mysite/wsgi.py: 用于你的项目的与WSGI兼容的Web服务器入口. 阅读 https://docs.djangoproject.com/zh-hans/2.0/howto/deployment/wsgi/ 了解更多细节。

在命令行执行 cd mysite ，进入到 项目根目录

接着执行 python manage.py runserver

会查看到：

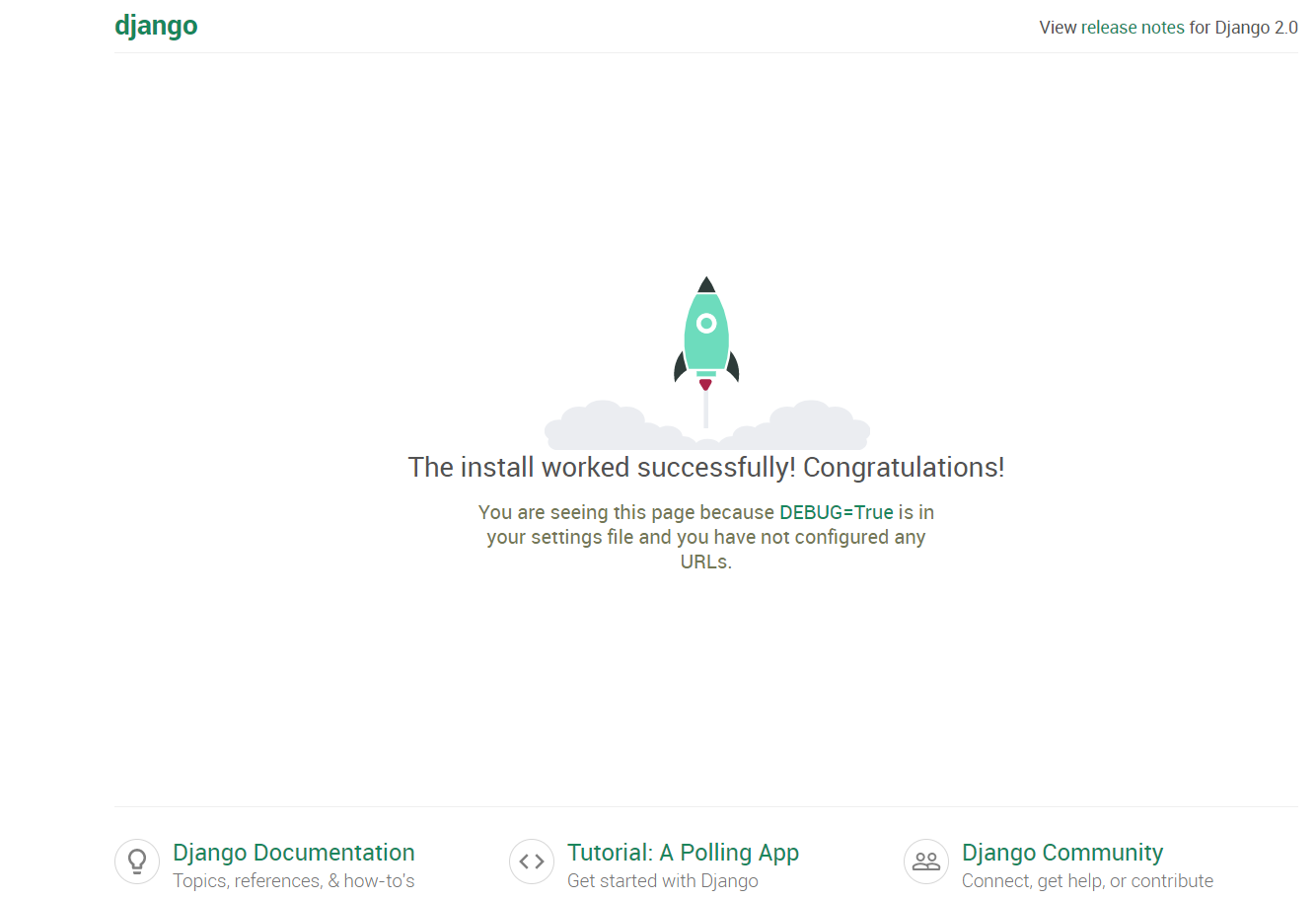


可以在 runserver 后加参数：

python manage.py runserver 8081

python manage.py runserver 127.0.0.1:8082

在浏览器中 访问 ip:port ，譬如：localhost:8080 ，会显示如下页面：



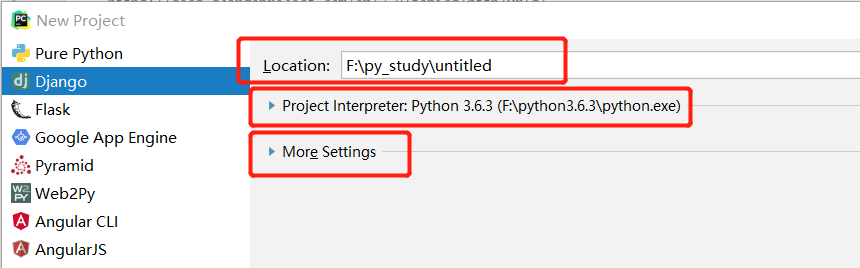
刚刚启动的是 Django 自带的用于开发的简易服务器，它是一个用纯 Python 写的轻量级的 Web 服务器。仅仅用于开发环境，不要在生产环境使用。

在这个服务器中，在需要的情况下会对每一次的访问请求重新载入一遍 Python 代码。所以你不需要为了让修改的代码生效而频繁的重新启动服务器。然而，一些动作，比如添加新文件，将不会触发自动重新加载，这时你得自己手动重启服务器。

## pycharm构建

### 新建项目

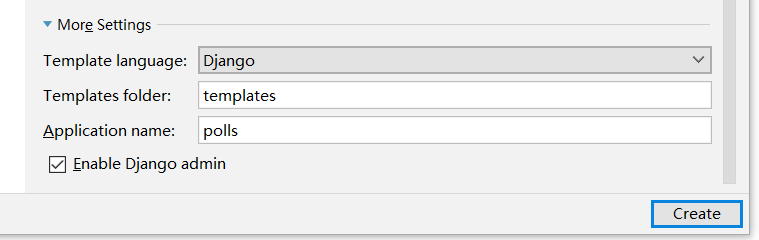
1. pycharm 中 file 》》 new project 》》 django



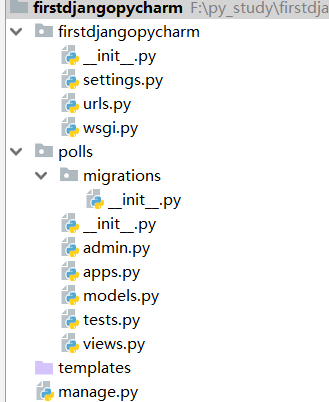
1. location 中，前面路径，选择你自己的项目路径， untitled 项目名改为 firstdjangopycharm 变成如下：

F:\py\_study\firstdjangopycharm

1. project interpreter 点击 左边的 三角形 ，选择一个合适的 解释器
2. more settings， 点击 左边的 三角形，设置 applications name，点击 create



1. 查看目录结构如下：



1. 可以查看 settings,py 文件：

|  |
| --- |
| INSTALLED\_APPS = [  **'django.contrib.admin'**,  **'django.contrib.auth'**,  **'django.contrib.contenttypes'**,  **'django.contrib.sessions'**,  **'django.contrib.messages'**,  **'django.contrib.staticfiles'**,  **'polls.apps.PollsConfig'**, ] |

模块的作用是：

* + - django.contrib.admin - 管理网站
    - django.contrib.auth - 一个认证系统
    - django.contrib.contenttypes - 内容类型的框架
    - django.contrib.sessions - 会话框架
    - django.contrib.messages - 消息传递框架
    - django.contrib.staticfiles - 一个管理静态文件的框架

有这样一条记录：

**'polls.apps.PollsConfig'**,

这是把polls 应用添加到项目中，如果不加到 INSTALLED\_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) , 后面会学习到它们分别用来干什么.

1. 查看 polls 目录：

* migrations ： 执行数据库的迁移操作，就是把 models 中的 model 映射到数据库
* admin ：管理界面文件
* apps： 本应用的相关配置文件
* models：数据库的model文件
* tests： 测试函数文件
* views： 视图文件

### 修改views

|  |
| --- |
| **from** django.http **import** HttpResponse  **def** index(request):  **return** HttpResponse(**"欢迎在光环学习Django!"**) |

这是 Django 中最简单的视图。

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）。

### polls目录下新建 urls.py

|  |
| --- |
| **from** django.urls **import** path **from** . **import** views  urlpatterns = [  path(**''**, views.index, name=**'index'**), ] |

from . imort views 是导入当前目录下的 views

path(route, view, kwargs=**None**, name=**None**, Pattern=**None**)

path() 参数： route

``route``是一个匹配 URL 的准则（类似正则表达式）。当Django响应一个请求时，它会从 ``urlpatterns``的第一项开始，按顺序依次匹配列表中的项，知道找到匹配的项。

这些准则不会匹配 GET 和 POST 参数或域名。例如，URLconf在处理请求'https://www.example.com/myapp/'时，它会尝试匹配``myapp/''。处理请求``https://www.example.com/myapp/?page=3``时，也只会尝试匹配``myapp/``。

path() 参数： view

当 Django 找到了一个匹配的准则，就会调用这个特定的视图函数，并传入一个:class:`~django.http.HttpRequest`对象作为第一个参数，被“捕获”的参数以关键字参数的形式传入。稍后，我们会给出一个例子。

path() 参数： kwargs

任意个关键字参数可以作为一个字典传递给目标视图函数。一般不会使用这一特性。

path() 参数： name

为你的 URL 取名能使你在 Django 的任意地方唯一地引用它，尤其是在模板中。这个有用的特性允许你只改一个文件就能全局地修改某个 URL 模式。

### firstdjangopycharm/urls 中修改

|  |
| --- |
| **from** django.contrib **import** admin **from** django.urls **import** path, include  urlpatterns = [  path('polls/', include('polls.urls')),  path('admin/', admin.site.urls),  ] |

django.urls.include 的理念是使其可以即插即用。因为 polls 应用有他自己的 URLconf(`polls/urls.py` )，他们能够被放在 "/polls/" 下、 "/fun\_polls/" 下、 "/content/polls/" 下，或者其他任何路径下，这个应用都能够正常工作。

当包括其它URL模式时你应该总是使用 include() ， admin.site.urls 是唯一例外。

### 运行 manage.py

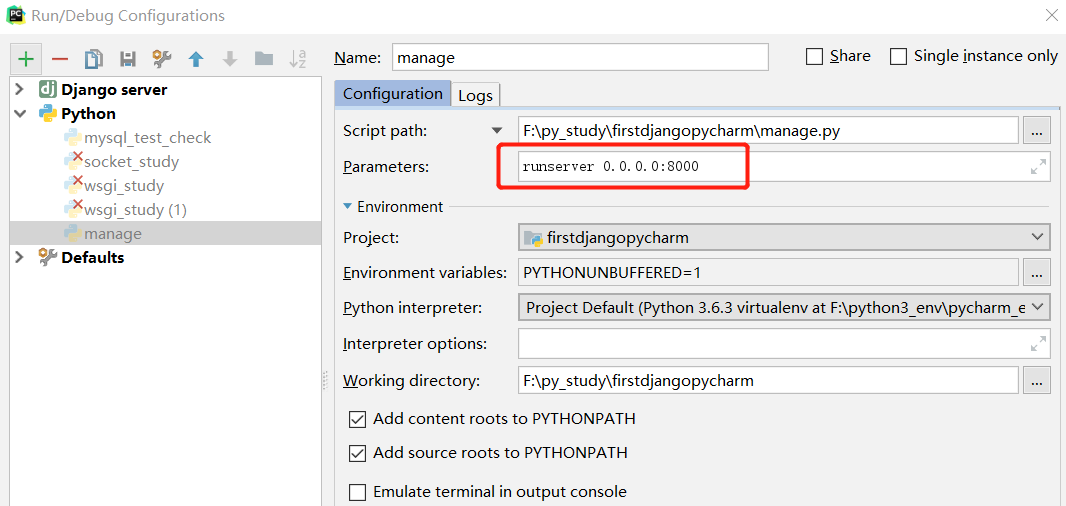
会失败

1. 在 pycharm 的右上角

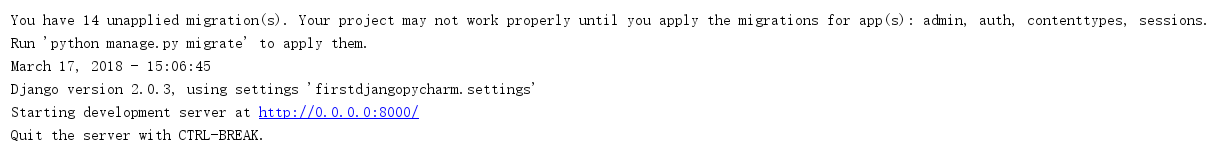


选择 edit configurations

1. 在 parameters 中写入 runserver 0.0.0.0:8000



1. 再次运行 manage.py



### 在浏览器中访问

输入 <http://localhost:8000/polls/>

### 数据库配置

在 mysite/settings.py 中配置数据库

|  |
| --- |
| DATABASES = {  **'default'**: {  **'ENGINE'**: **'django.db.backends.sqlite3'**,  **'NAME'**: os.path.join(BASE\_DIR, **'db.sqlite3'**),  } } |

**'ENGINE'**: 数据库的引擎，这里默认的是sqlite3，其他数据库可以修改为其它的配置，譬如mysql是：'django.db.backends.mysql'

**'NAME'**: 数据库的名称。如果您使用SQLite，数据库将成为您计算机上的文件; 在这种情况下，NAME 应该是该文件的完整绝对路径，包括文件名。

使用其他数据库，会额外多出一些配置，譬如 用户名、密码等，mysql示例如下：

|  |
| --- |
| DATABASES = {  **'default'**: {  *# mysql引擎固定写法* **'ENGINE'**: **'django.db.backends.mysql'**,  *# 数据库名！！！！* **'NAME'**: **'django1802'**,  *# 用户名* **'USER'**: **'root'**,  *# 密码* **'PASSWORD'**: **'123456'**,  *# 数据库IP* **'HOST'**: **'127.0.0.1'**,  *# 数据库端口* **'PORT'**: **'3306'**,  } } |

在python3中，使用mysql数据库，由于django默认的是使用MySQLdb，直接运行，会报如下错误：

|  |
| --- |
| django.core.exceptions.ImproperlyConfigured: Error loading MySQLdb module.  Did you install mysqlclient? |

需要在 \_\_init\_\_.py 文件中，进行修改：

|  |
| --- |
| **import** pymysql  pymysql.install\_as\_MySQLdb() |

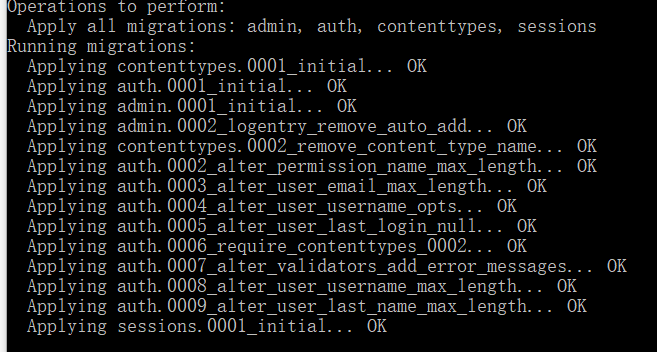
更多配置，请查询文档中的 数据库配置 部分

修改完成后，在项目根目录的cmd窗口中，可以使用如下命令，生成默认配置的模块中的数据，譬如管理系统，认证系统

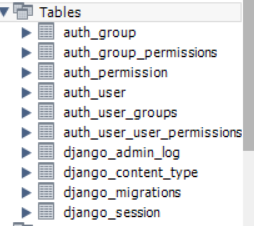
在执行命令之前，先在数据库中建立对应的database

|  |
| --- |
| python manage.py migrate |

命令执行如下：



接着查看数据库，可以看到数据库中已经建立了一些表格。



### 修改时区

在settings文件中：

|  |
| --- |
| TIME\_ZONE = **'Asia/Shanghai'**  USE\_TZ = **False** |

所有的时区在

<https://en.wikipedia.org/wiki/List_of_tz_database_time_zones>

可以查看

### 模型操作

修改 polls 目录下的 models.py 文件

|  |
| --- |
| **from** django.db **import** models   **class** Question(models.Model):  question\_text = models.CharField(max\_length=200)  pub\_date = models.DateTimeField(**'date published'**)   **class** Choice(models.Model):  question = models.ForeignKey(Question, on\_delete=models.CASCADE)  choice\_text = models.CharField(max\_length=200)  votes = models.IntegerField(default=0) |

确保在settings中，包含了 polls 应用

|  |
| --- |
| INSTALLED\_APPS  中设置了：  **'polls.apps.PollsConfig'** |

在根目录的cmd中运行：

|  |
| --- |
| python manage.py makemigrations polls |

运行结果如下：

|  |
| --- |
| Migrations for 'polls':  polls/migrations/0001\_initial.py:  - Create model Choice  - Create model Question  - Add field question to choice |

会在 polls 目录下的 migrations 目录下生成：

0001\_initial.py

文件中包含了建立数据库的类对象

运行如下命令：

|  |
| --- |
| python manage.py sqlmigrate polls 0001 |

能看的如下内容：

|  |
| --- |
| BEGIN;  --  -- Create model Choice  --  CREATE TABLE `polls\_choice` (`id` integer AUTO\_INCREMENT NOT NULL PRIMARY KEY, `choice\_text` varchar(200) NOT NULL, `votes` integer NOT NULL);  --  -- Create model Question  --  CREATE TABLE `polls\_question` (`id` integer AUTO\_INCREMENT NOT NULL PRIMARY KEY, `question\_text` varchar(200) NOT NULL, `pub\_date` datetime(6) NOT NULL);  --  -- Add field question to choice  --  ALTER TABLE `polls\_choice` ADD COLUMN `question\_id` integer NOT NULL;  ALTER TABLE `polls\_choice` ADD CONSTRAINT `polls\_choice\_question\_id\_c5b4b260\_fk\_polls\_question\_id` FOREIGN KEY (`question\_id`) REFERENCES `polls\_question` (`id`);  COMMIT; |

该命令不会真正执行，只是显示对应的SQL语句。

如果要真正的在数据库生成对应的表，那么需要执行如下命令：

|  |
| --- |
| python manage.py migrate |

显示结果如下：

|  |
| --- |
| Operations to perform:  Apply all migrations: admin, auth, contenttypes, polls, sessions  Running migrations:  Applying polls.0001\_initial... OK |

在数据库查看，会看到对应的表格已经生成

总结，进行模型的更改的三个步骤：

* 修改你的模型（in models.py）
* 为这些更改创建迁移：python manage.py makemigrations
* 更改应用于数据库：python manage.py migrate

可以在shell中，通过 django API 操纵模型：

cmd中：

|  |
| --- |
| python manage.py shell |

在shell中：

|  |
| --- |
| >>> from polls.models import Question, Choice # Import the model classes we just wrote.  # 显示question的所有记录  >>> Question.objects.all()  <QuerySet []>  # 实例一个新的ORM类  >>> from django.utils import timezone  >>> q = Question(question\_text="What's new?", pub\_date=timezone.now())  # 将记录保存到数据库中  >>> q.save()  # 获取到记录的ID  >>> q.id  1  # 访问实例的属性  >>> q.question\_text  "What's new?"  >>> q.pub\_date  datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)  # 修改实例的属性，并且通过save进行update  >>> q.question\_text = "What's up?"  >>> q.save()  # 显示所有记录  >>> Question.objects.all()  <QuerySet [<Question: Question object (1)>]> |

在models中进行修改：

|  |
| --- |
| **class** **Question**(models.Model):  *# ...*  **def** \_\_str\_\_(self):  **return** self.question\_text  **def** was\_published\_recently(self):  **return** self.pub\_date >= timezone.now() - datetime.timedelta(days=1)  **class** **Choice**(models.Model):  *# ...*  **def** \_\_str\_\_(self):  **return** self.choice\_text |

再次开启新的 shell： exit() 退出后，重新执行 python manage.py shell

|  |
| --- |
| **>>> from** **polls.models** **import** Question, Choice  # 如果添加了\_\_str\_\_方法，这里可以查看到  **>>>** Question.objects.all()  <QuerySet [<Question: What's up?>]>  # 通过关键字参数进行检索  **>>>** Question.objects.filter(id=1)  <QuerySet [<Question: What's up?>]>  **>>>** Question.objects.filter(question\_text\_\_startswith='What')  <QuerySet [<Question: What's up?>]>  # 根据日期的年份来匹配  **>>> from** **django.utils** **import** timezone  **>>>** current\_year = timezone.now().year  **>>>** Question.objects.get(pub\_date\_\_year=current\_year)  <Question: What's up?>  # 根据ID来获取对象，如果不存在会抛出异常  **>>>** Question.objects.get(id=2)  Traceback (most recent call last):  ...  DoesNotExist: Question matching query does not exist.  # 通过主键来获取对象，这里等同于 id=1  **>>>** Question.objects.get(pk=1)  <Question: What's up?>  # 调用对象的自定义方法  **>>>** q = Question.objects.get(pk=1)  **>>>** q.was\_published\_recently()  True  # 根据主键获取对象  **>>>** q = Question.objects.get(pk=1)  # 通过主外关联获取从表对应的对象的列表  **>>>** q.choice\_set.all()  <QuerySet []>  # 通过主外关联，生成外键对应，并且在数据库生成对应记录.  **>>>** q.choice\_set.create(choice\_text='Not much', votes=0)  <Choice: Not much>  **>>>** q.choice\_set.create(choice\_text='The sky', votes=0)  <Choice: The sky>  **>>>** c = q.choice\_set.create(choice\_text='Just hacking again', votes=0)  # 获取外键对象对应的主键对象  **>>>** c.question  <Question: What's up?>  # 显示所有外键类对象的集合  **>>>** q.choice\_set.all()  <QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>  **>>>** q.choice\_set.count()  3  # 通过主外关联，通过对应的主表的条件，查询子表的集合  **>>>** Choice.objects.filter(question\_\_pub\_date\_\_year=current\_year)  <QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>  # 获取对应的对象，并且删除.  **>>>** c = q.choice\_set.filter(choice\_text\_\_startswith='Just hacking')  **>>>** c.delete() |

### admin功能

创建管理员：

首先，我们需要创建一个可以登录管理网站的用户。运行以下命令：

python manage.py createsuperuser

输入你想要的用户名并按回车。

Username: admin

您将被提示输入您想要的电子邮件地址：

Email address: [admin@example.com](mailto:admin@example.com)

最后一步是输入您的密码。您将被要求输入两次密码，第二次作为第一次确认。

Password: \*\*\*\*\*\*\*\*\*\*

Password (again): \*\*\*\*\*\*\*\*\*

Superuser created successfully.

PS：如果settings中配置了 AUTH\_PASSWORD\_VALIDATORS，那么会进行对应的检测

确保在 settings 中配置：

|  |
| --- |
| INSTALLED\_APPS = [  **'django.contrib.admin'**,  ] |

在admin.py中添加

|  |
| --- |
| **from** django.contrib **import** admin  **from** .models **import** Question  admin.site.register(Question) |

上述配置，将Question纳入到了admin的管理中。

访问 <http://127.0.0.1:8000/admin> 并且登录，在polls中查看

### view功能

在当前系统中增加4个视图显示：

* 问题“index”页面 - 显示最新的几个问题。
* 问题“detail”页面 - 显示问题文本，没有结果，但有投票表格。
* 问题“results”页面 - 显示特定问题的结果。
* “vote”页面 - 在特定问题中处理针对特定选择的投票。

在views.py 中，增加：

|  |
| --- |
| **def** detail(request, question\_id):  **return** HttpResponse(**"You're looking at question %s."** % question\_id)  **def** results(request, question\_id):  response = **"You're looking at the results of question %s."  return** HttpResponse(response % question\_id)  **def** vote(request, question\_id):  **return** HttpResponse(**"You're voting on question %s."** % question\_id) |

在 polls/urls.py 中修改：

|  |
| --- |
| **from** django.urls **import** path **from** . **import** views  urlpatterns = [  *# ex: /polls/* path(**''**, views.index, name=**'index'**),  *# ex: /polls/5/* path(**'<int:question\_id>/'**, views.detail, name=**'detail'**),  *# ex: /polls/5/results/* path(**'<int:question\_id>/results/'**, views.results, name=**'results'**),  *# ex: /polls/5/vote/* path(**'<int:question\_id>/vote/'**, views.vote, name=**'vote'**), ] |

千万别这么写：

|  |
| --- |
| *# 千万别这么写，需要http://127.0.0.1:8000/polls/polls/latest.html才能访问到* path(**'polls/latest.html'**, views.index), |

接着，我们修改index 函数：

|  |
| --- |
| **def** index(request):  latest\_question\_list = Question.objects.order\_by(**'-pub\_date'**)[:5]  output = **', '**.join([q.question\_text **for** q **in** latest\_question\_list])  **return** HttpResponse(output) |

在页面查看修改后的效果

### templates功能

在polls目录下，新增 templates 文件夹（不需要建 源目录 ），新增文件：

index.html，内容如下：

|  |
| --- |
| {% **if latest\_question\_list** %}  <**ul**>  {% **for question in latest\_question\_list** %}  <**li**><**a href="/polls/**{{ **question**.**id** }}**/"**>问题{{ **question**.**id** }}:{{ **question**.**question\_text** }}</**a**></**li**>  {% **endfor** %}  </**ul**> {% **else** %}  <**p**>No polls are available.</**p**> {% **endif** %} |

修改 views.py 中的 index 函数：

|  |
| --- |
| **from** django.template **import** loader  **def** index(request):  latest\_question\_list = Question.objects.order\_by(**'-pub\_date'**)[:5]  template = loader.get\_template(**'index.html'**)  context = {  **'latest\_question\_list'**: latest\_question\_list,  }  **return** HttpResponse(template.render(context, request)) |

简便写法：

|  |
| --- |
| **from** django.shortcuts **import** render  **def** index(request):  latest\_question\_list = Question.objects.order\_by(**'-pub\_date'**)[:5]  context = {**'latest\_question\_list'**: latest\_question\_list}  **return** render(request, **'index.html'**, context) |

在templates中新增detail.html，内容如下：

|  |
| --- |
| {{ **question** }} |

修改views.py：

|  |
| --- |
| **from** django.http **import** HttpResponse, Http404  **def** detail(request, question\_id):  **try**:  question = Question.objects.get(pk=question\_id)  **except** Question.DoesNotExist:  **raise** Http404(**"Question does not exist"**)  **return** render(request, **'detail.html'**, {**'question'**: question}) |

404错误的简便写法：

|  |
| --- |
| **from** django.shortcuts **import** render, get\_object\_or\_404  **def** detail(request, question\_id):  question = get\_object\_or\_404(Question, pk=question\_id)  **return** render(request, **'detail.html'**, {**'question'**: question}) |

编写更丰富的template，修改 detail 函数：

|  |
| --- |
| <**h1**>{{ **question**.**question\_text** }}</**h1**> <**ul**> {% **for choice in question**.**choice\_set**.**all** %}  <**li**>{{ **choice**.**choice\_text** }}</**li**> {% **endfor** %} </**ul**> |

删除index.html的硬编码：

|  |
| --- |
| <**a href="/polls/**{{ **question**.**id** }}**/"**>  修改为  <**a href="**{% **url 'detail' question**.**id** %}**"**> |

应用命名空间，多个类似polls的应用之间，模板名可能冲突，那么就需要使用命名空间了，在urls.py 中增加：

|  |
| --- |
| *# 这个名字不能冲突，项目中值允许出现一次*  app\_name = **'polls'** |

在模板文件中修改：

|  |
| --- |
| <**li**><**a href="**{% **url 'detail' question**.**id** %}**"**>  改为：  <**li**><**a href="**{% **url 'polls:detail' question**.**id** %}**"**> |

template中编写form，修改detail.html

|  |
| --- |
| <**h1**>{{ **question**.**question\_text** }}</**h1**>  {% **if error\_message** %}<**p**><**strong**>{{ **error\_message** }}</**strong**></**p**>{% **endif** %}  <**form action="**{% **url 'polls:vote' question**.**id** %}**" method="post"**> {% **csrf\_token** %} {% **for choice in question**.**choice\_set**.**all** %}  <**input type="radio" name="choice" id="choice**{{ **forloop**.**counter** }}**" value="**{{ **choice**.**id** }}**"** />  <**label for="choice**{{ **forloop**.**counter** }}**"**>{{ **choice**.**choice\_text** }}</**label**><**br** /> {% **endfor** %} <**input type="submit" value="Vote"** /> </**form**> |

简要说明：

* 上面的模板通过for循环（endfor 结束）为每个问题选项显示一个单选按钮。value是相关联的问题的选择的ID。 name是"choice"。这意味着，当有人选择其中一个单选按钮并提交表单时，它将发送POST数据choice=#，其中＃是所选选项的value，这是HTML表单的基本概念。
* 设置form的action到 {% url 'polls:vote' question.id %} ，method使用post，而不是get，记住：所有修改服务器内容的提交都必须使用post方法！
* forloop.counter表示for标签经过其循环的次数，从1开始
* 防止跨站点请求伪造，所有post提交的form表单中都应该设置模板标记：{% csrf\_token %}

修改 views.py 的 vote 函数：

|  |
| --- |
| **def** vote(request, question\_id):  question = get\_object\_or\_404(Question, pk=question\_id)  **try**:  selected\_choice = question.choice\_set.get(pk=request.POST[**'choice'**])  **except** (KeyError, Choice.DoesNotExist):  *# 出错，重新定向到 detail.html* **return** render(request, **'detail.html'**, {  **'question'**: question,  **'error\_message'**: **"You didn't select a choice."**,  })  **else**:  selected\_choice.votes += 1  selected\_choice.save()  *# 跳转到 results 页面*  *# 这里使用 HttpResponseRedirect ，所有post请求这样使用，防止用户后退、刷新等操作，使用reverse避免硬编码* **return** HttpResponseRedirect(reverse(**'polls:results'**, args=(question.id,))) |

简要说明：

* request.POST是一个类似字典的对象，可让通过键名访问提交的数据， request.POST['choice']以字符串形式返回所选选项的ID，request.POST值总是字符串
* Django还提供request.GET了以相同的方式访问GET数据 - 但我们明确地request.POST在我们的代码中使用，以确保数据仅通过POST调用进行更改
* request.POST['choice']的异常：KeyError，如果 choice在POST数据中没有提供，将会引发
* 增加选择计数后，代码将返回一个 HttpResponseRedirect而不是一个普通 HttpResponse。HttpResponseRedirect只有一个参数：用户将被重定向到的URL（关于这种情况下我们如何构造URL，请参阅以下几点：
  + post成功后，返回HttpResponseRedirect
  + 使用reverse()避免使用url的硬编码

修改 views.py中的results 函数：

|  |
| --- |
| **def** results(request, question\_id):  question = get\_object\_or\_404(Question, pk=question\_id)  **return** render(request, **'results.html'**, {**'question'**: question}) |

在templates目录中增加 results.html 页面：

|  |
| --- |
| <**h1**>{{ **question**.**question\_text** }}</**h1**>  <**ul**> {% **for choice in question**.**choice\_set**.**all** %}  <**li**>{{ **choice**.**choice\_text** }} -- {{ **choice**.**votes** }} vote{{ **choice**.**votes**|**pluralize** }}</**li**> {% **endfor** %} </**ul**>  <**a href="**{% **url 'polls:detail' question**.**id** %}**"**>Vote again?</**a**> |

### 通用视图

我们观察views.py中的功能，是否发现detail和results函数几乎一样，这种情况，可以使用通用视图功能，节省代码。

修改views.py

|  |
| --- |
| **from** django.shortcuts **import** get\_object\_or\_404, render **from** django.http **import** HttpResponseRedirect **from** django.urls **import** reverse **from** django.views **import** generic  **from** .models **import** Choice, Question   **class** IndexView(generic.ListView):  *"""  使用通用的list视图  """  # 使用的模板名* template\_name = **'index.html'** *# 在 模板页面 中使用的变量名，需要和 index.html 中的变量名对应* context\_object\_name = **'latest\_question\_list'** *# 获取显示的list对象的方法* **def** get\_queryset(self):  *"""Return the last five published questions."""* **return** Question.objects.order\_by(**'-pub\_date'**)[:5]   **class** DetailView(generic.DetailView):  *"""  使用通用的详情视图  """  # 对应的模型对象* model = Question  *# 使用的模板名* template\_name = **'detail.html'**  **class** ResultsView(generic.DetailView):  model = Question  template\_name = **'results.html'** |

然后修改 urls.py：

|  |
| --- |
| urlpatterns = [  path(**''**, views.IndexView.as\_view(), name=**'index'**),  *# 使用通用视图，必须将questsion\_id 修改为 pk* path(**'<int:pk>/'**, views.DetailView.as\_view(), name=**'detail'**),  path(**'<int:pk>/results/'**, views.ResultsView.as\_view(), name=**'results'**),  path(**'<int:question\_id>/vote/'**, views.vote, name=**'vote'**), ] |

### 自动化测试

测试第一个错误：

Question.was\_published\_recently()

即使时间是在未来，也会返回True，但是显示应该返回False

在shell进行测试：

|  |
| --- |
| **>>> import** **datetime**  **>>> from** **django.utils** **import** timezone  **>>> from** **polls.models** **import** Question  **>>>** *# create a Question instance with pub\_date 30 days in the future*  **>>>** future\_question = Question(pub\_date=timezone.now() + datetime.timedelta(days=30))  **>>>** *# was it published recently?*  **>>>** future\_question.was\_published\_recently()  True |

编写自动化测试类，在tests.py 中增加：

|  |
| --- |
| **import** datetime  **from** django.utils **import** timezone **from** django.test **import** TestCase  **from** .models **import** Question   **class** QuestionModelTests(TestCase):   **def** test\_was\_published\_recently\_with\_future\_question(self):  *"""  was\_published\_recently() returns False for questions whose pub\_date  is in the future.  """* time = timezone.now() + datetime.timedelta(days=30)  future\_question = Question(pub\_date=time)  self.assertIs(future\_question.was\_published\_recently(), **False**) |

接着在shell中运行：

|  |
| --- |
| python manage.py test polls |

结果如下：

|  |
| --- |
| Creating test database for alias 'default'...  System check identified no issues (0 silenced).  F  ======================================================================  FAIL: test\_was\_published\_recently\_with\_future\_question (polls.tests.QuestionModelTests)  ----------------------------------------------------------------------  Traceback (most recent call last):  File "F:\py\_study\firstdjangopycharm\polls\tests.py", line 20, in test\_was\_published\_recently\_with\_future\_question  self.assertIs(future\_question.was\_published\_recently(), False)  AssertionError: True is not False  ----------------------------------------------------------------------  Ran 1 test in 0.003s  FAILED (failures=1)  Destroying test database for alias 'default'... |

上述结果的产生过程是这样的：

* python manage.py test polls在polls应用程序中寻找测试
* 它找到了这个类的一个子django.test.TestCase类
* 它创建了一个专门用于测试目的的数据库
* 它寻找测试方法 -以test开头的方法名
* 在test\_was\_published\_recently\_with\_future\_question它创建了一个Question 实例，其pub\_date属性设置为将来30天
* 使用assertIs() 方法，断言 was\_published\_recently() 返回 True，但是我们的希望值是： False

测试通知我们哪个测试失败，发生错误的位置

知道错误后，我们进行修复：

修改models.py：

|  |
| --- |
| **def** was\_published\_recently(self):  now = timezone.now()  **return** now - datetime.timedelta(days=1) <= self.pub\_date <= now |

再次运行，测试通过

继续编写views测试，

修改views.py ，只显示当前时间之前的时间最近的5条记录，不显示时间在未来的

|  |
| --- |
| **def** get\_queryset(self):  *"""Return the last five published questions."""* **return** Question.objects.filter(pub\_date\_\_lte=timezone.now()).order\_by(**'-pub\_date'**)[:5] |

在tests中增加：

|  |
| --- |
| **from** django.urls **import** reverse  **def** create\_question(question\_text, days):  *"""  Create a question with the given `question\_text` and published the  given number of `days` offset to now (negative for questions published  in the past, positive for questions that have yet to be published).  """* time = timezone.now() + datetime.timedelta(days=days)  **return** Question.objects.create(question\_text=question\_text, pub\_date=time)   **class** QuestionIndexViewTests(TestCase):  **def** test\_no\_questions(self):  *"""  If no questions exist, an appropriate message is displayed.  """* response = self.client.get(reverse(**'polls:index'**))  self.assertEqual(response.status\_code, 200)  self.assertContains(response, **"No polls are available."**)  self.assertQuerysetEqual(response.context[**'latest\_question\_list'**], [])   **def** test\_past\_question(self):  *"""  Questions with a pub\_date in the past are displayed on the  index page.  """* create\_question(question\_text=**"Past question."**, days=-30)  response = self.client.get(reverse(**'polls:index'**))  self.assertQuerysetEqual(  response.context[**'latest\_question\_list'**],  [**'<Question: Past question.>'**]  )   **def** test\_future\_question(self):  *"""  Questions with a pub\_date in the future aren't displayed on  the index page.  """* create\_question(question\_text=**"Future question."**, days=30)  response = self.client.get(reverse(**'polls:index'**))  self.assertContains(response, **"No polls are available."**)  self.assertQuerysetEqual(response.context[**'latest\_question\_list'**], [])   **def** test\_future\_question\_and\_past\_question(self):  *"""  Even if both past and future questions exist, only past questions  are displayed.  """* create\_question(question\_text=**"Past question."**, days=-30)  create\_question(question\_text=**"Future question."**, days=30)  response = self.client.get(reverse(**'polls:index'**))  self.assertQuerysetEqual(  response.context[**'latest\_question\_list'**],  [**'<Question: Past question.>'**]  )   **def** test\_two\_past\_questions(self):  *"""  The questions index page may display multiple questions.  """* create\_question(question\_text=**"Past question 1."**, days=-30)  create\_question(question\_text=**"Past question 2."**, days=-5)  response = self.client.get(reverse(**'polls:index'**))  self.assertQuerysetEqual(  response.context[**'latest\_question\_list'**],  [**'<Question: Past question 2.>'**, **'<Question: Past question 1.>'**]  ) |

### 静态文件

静态文件，包括普通的html、css、js、image、视频等，都放入到static文件夹中

在polls目录下，创建 polls/static/css目录，

在目录中新建 style.css 文件，内容如下：

|  |
| --- |
| **li a** {  **color**: **green**; } |

修改index.html，增加：

|  |
| --- |
| {% **load static** %} <**link rel="stylesheet" type="text/css" href="**{% **static 'css/style.css'** %}**"** /> |

{% **load static** %} 生成static目录的路径

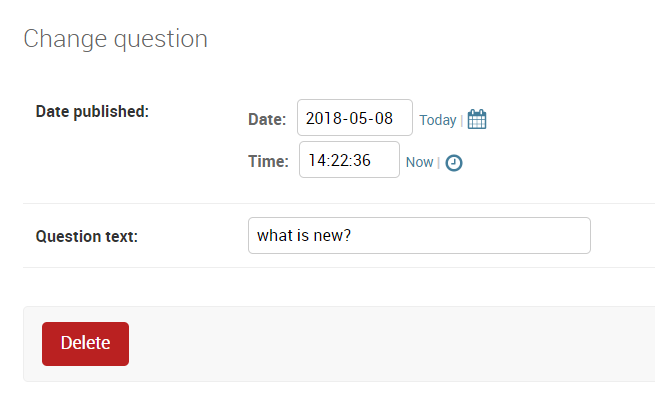
重启服务后，查看效果

### admin进一步完善

修改admin.py：

|  |
| --- |
| **from** django.contrib **import** admin **from** .models **import** Question   **class** QuestionAdmin(admin.ModelAdmin):  fields = [**'pub\_date'**, **'question\_text'**]   admin.site.register(Question, QuestionAdmin) |

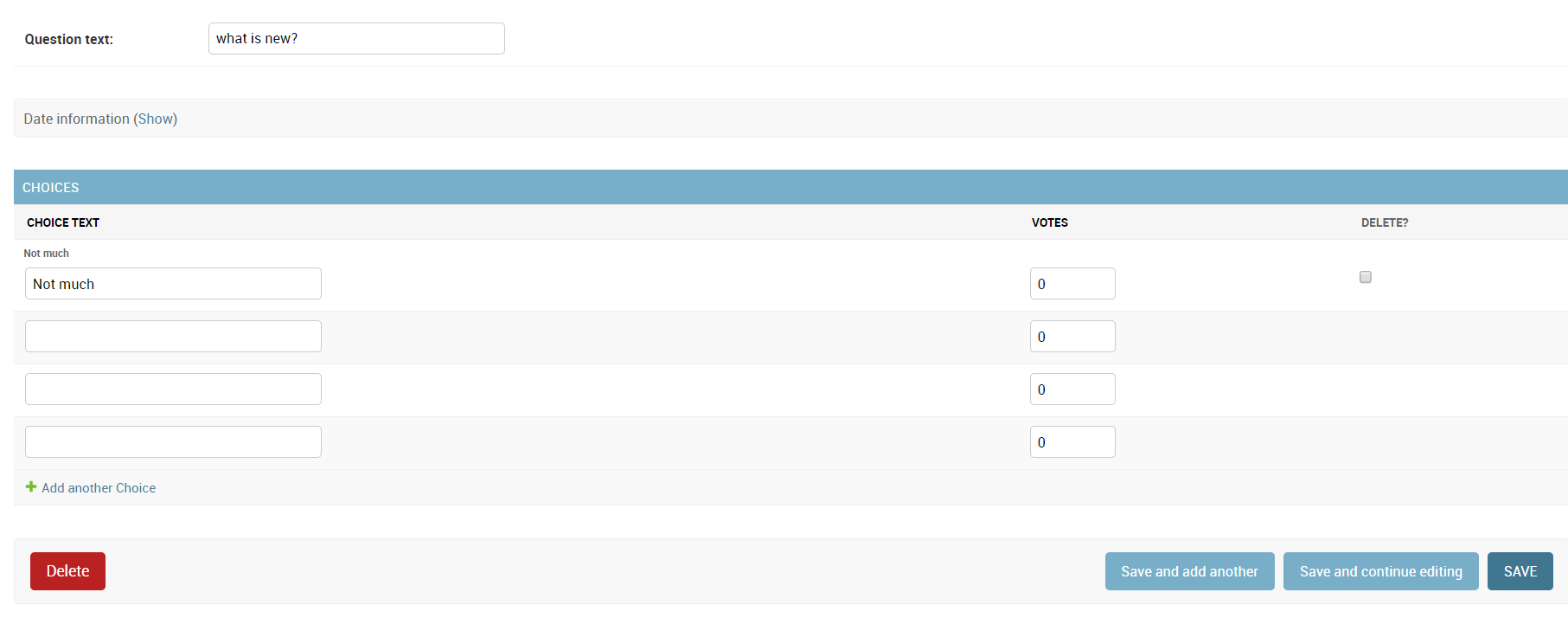
运行查看效果如下：



继续修改：

|  |
| --- |
| **from** django.contrib **import** admin  **from** .models **import** Choice, Question   *# TabularInline 节约空间 # StackedInline 更占页面空间* **class** ChoiceInline(admin.TabularInline):  model = Choice  extra = 3   **class** QuestionAdmin(admin.ModelAdmin):  fieldsets = [  (**None**, {**'fields'**: [**'question\_text'**]}),  (**'Date information'**, {**'fields'**: [**'pub\_date'**], **'classes'**: [**'collapse'**]}),  ]   inlines = [ChoiceInline]    admin.site.register(Question, QuestionAdmin) |

结果如下：



要删除 choices，勾选后面的 delete，然后点击 save

删除 question，点击 delete

# settings

settings文件包含Django安装的所有配置，是一个包含模块级变量的python模块，所以该模块本身必须符合python规则，并且可以使用python的语法。

settings中的所有配置项的key必须全部大写

settings中每一个配置项都有默认值，默认配置内容在django/conf/global\_settings.py中可以查看到，项目中不需要导入该模块，系统会自动获取

在项目中需要获取settings中的配置项，这样获取：

|  |
| --- |
| *# 切记不要导入真实的settings模块的路径，会形成高耦合 # 这样是不可取的：from mysite import settings* **from** django.conf **import** settings  d = settings.DEBUG |

切记不要在项目中修改配置项，只能读取！！！

## 应用settings

### manage.py配置

通过manage.py中配置：

|  |
| --- |
| **if** \_\_name\_\_ == **"\_\_main\_\_"**:  os.environ.setdefault(**"DJANGO\_SETTINGS\_MODULE"**, **"firstdjangopycharm.settings"**) |

### cmd命令启动

|  |
| --- |
| python manage.py runserver 0.0.0.0:8000 --settings=firstdjangopycharm.settings |

### 服务器部署启动

在wsgi.py中配置：

|  |
| --- |
| os.environ.setdefault(**"DJANGO\_SETTINGS\_MODULE"**, **"firstdjangopycharm.settings"**) |

## 常用配置项

django有一个全局的默认配置项：

django.confg.global\_settings.py

|  |
| --- |
| **import** os   **"""  当前文件所在文件夹的上一级目录的绝对路径  切记2个 os.path.dirname """** BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))  **"""  用于加密session，一个随机的字符串  这样生成：  from django.core.management import utils  utils.get\_random\_secret\_key() """** SECRET\_KEY = **'=\*f&bx760nyar7@8lb8!w$9h(3ea6p3apl$iua!td1q%-u5r4='** *# 调试模式，可以看到错误的所有相信信息，部署时一定要修改为False* DEBUG = **True  """  允许访问的域名设置  开发环境不用理会  运行环境，配置 DEBUG = False后，  如果允许所有网站访问，则设置 ALLOW\_HOSTS = ['\*']  如果指定某些网站可以访问，则设置 ALLOW\_HOSTS = ['\*.baidu.com'] """** ALLOWED\_HOSTS = []   **"""  应用的配置，  如：'polls.apps.PollsConfig'  如果没有 PollsConfig ，那么可以配置为 'polls' """** INSTALLED\_APPS = [  **'django.contrib.admin'**,  **'django.contrib.auth'**,  **'django.contrib.contenttypes'**,  **'django.contrib.sessions'**,  **'django.contrib.messages'**,  **'django.contrib.staticfiles'**,  **'polls'** ]  **"""  中间层配置  自己编写的 中间层 需要配置在最后  譬如：  mymidlle.md.TestMiddleware """** MIDDLEWARE = [  **'django.middleware.security.SecurityMiddleware'**,  **'django.contrib.sessions.middleware.SessionMiddleware'**,  **'django.middleware.common.CommonMiddleware'**,  **'django.middleware.csrf.CsrfViewMiddleware'**,  **'django.contrib.auth.middleware.AuthenticationMiddleware'**,  **'django.contrib.messages.middleware.MessageMiddleware'**,  **'django.middleware.clickjacking.XFrameOptionsMiddleware'**, ]  *# 配置基础的urls* ROOT\_URLCONF = **'firstdjangopycharm.urls'** *# 配置模板* 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'**,  ],  },  }, ]  *# 服务器部署的WSGI配置* WSGI\_APPLICATION = **'firstdjangopycharm.wsgi.application'   """  数据库配置  mysql在python3的使用，需要在 \_\_init\_\_.py 中加入以下代码：  import pymysql   pymysql.install\_as\_MySQLdb() """** *# DATABASES = { # 'default': { # 'ENGINE': 'django.db.backends.sqlite3', # 'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'), # } # }* DATABASES = {  **'default'**: {  **'ENGINE'**: **'django.db.backends.mysql'**,  **'NAME'**: **'django\_test1'**,  **'USER'**: **'root'**,  **'PASSWORD'**: **'123456'**,  **'HOST'**: **'127.0.0.1'**,  **'PORT'**: **'3306'**,  } }   **"""  用户密码验证 """** AUTH\_PASSWORD\_VALIDATORS = [  {  **'NAME'**: **'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator'**,  },  {  **'NAME'**: **'django.contrib.auth.password\_validation.MinimumLengthValidator'**,  },  {  **'NAME'**: **'django.contrib.auth.password\_validation.CommonPasswordValidator'**,  },  {  **'NAME'**: **'django.contrib.auth.password\_validation.NumericPasswordValidator'**,  }, ]   *# 国际化* LANGUAGE\_CODE = **'en-us'** *# 时区* TIME\_ZONE = **'Asia/Shanghai'** USE\_I18N = **True** USE\_L10N = **True** *# 使用时区，配套TIME\_ZONE使用，必须设置为 False* USE\_TZ = **False  """  静态文件的默认路径  如果在各自项目的static目录以外，还有目录存放静态文件，需要添加如下属性：  STATICFILES\_DIRS = (  os.path.join(BASE\_DIR, "common\_static1"),  '/var/www/static/',  ) """** STATIC\_URL = **'/static/'** |

# 模型

python中的类对应数据库中的表

譬如：

在polls项目中，模型：

|  |
| --- |
| **class** Question(models.Model):  question\_text = models.CharField(max\_length=200)  *# 'date published' 是属性的详细名称，*  *# 如果没有指定，那么默认是属性名，并且用 空格 替换 \_ ，这里如果没有指定，那么名称是 'pub date'*  pub\_date = models.DateTimeField(**'date published'**) |

对应mysql数据库中的表：

|  |
| --- |
| CREATE TABLE `polls\_question` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `question\_text` varchar(200) NOT NULL,  `pub\_date` datetime(6) NOT NULL,  PRIMARY KEY (`id`)  ) |

PS：模型必须继承models.Model

每个属性对应数据库中的一个列

表名自动使用 mysite\_类名的小写，可以覆盖

会自动添加id字段，可以覆盖

## 应用模型

当编写了模型之后，需要将模型应用到数据库中：

### 配置应用

将模型对应的应用程序添加到项目的settings中：

|  |
| --- |
| INSTALLED\_APPS = [  **'polls'** ] |

### 预备迁移

在根目录的cmd中运行：

|  |
| --- |
| python manage.py makemigrations polls |

### 正式迁移

|  |
| --- |
| python manage.py migrate |

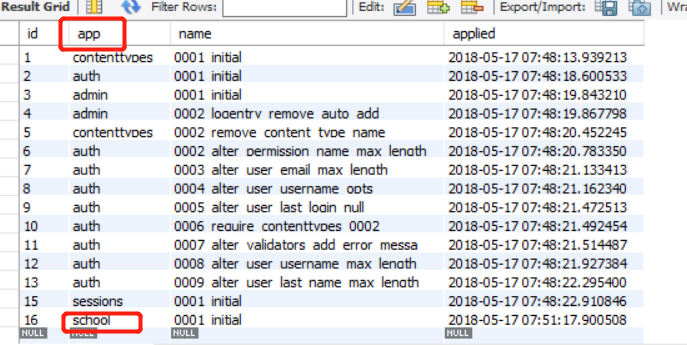
### 模型修改后应用

应用修改模型后，需要重新生成数据库中的表格

* 删除对应的应用中migrations目录下的0001\_initial.py 文件
* 在数据库中的django\_migrations表，删除关于应用的记录:

如下图中，删除 app字段为school的记录

delete from django\_migrations where app='school';



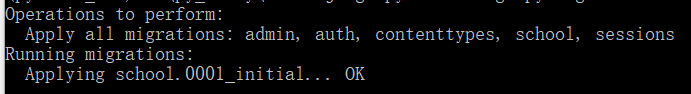
* 在数据库中，删除所有 school\_ 开头的表，不然重新建立时，会提示表已经存在
* 在cmd中切换到 manage.py 同级目录下，执行

|  |
| --- |
| # 其中 school 是应用名  python manage.py makemigrations school |

* 再次执行

|  |
| --- |
| python manage.py migrate |

即可看到：



重新生成数据库成功！

## 字段Field

模型类的属性对应数据库中表的列，都是对应的Field类的实例

### AutoField、ID、PRIMARY\_KEY

默认会自动创建一个自增，主键的id列

如果指定了 primary\_key 为其它列，那么不会自动创建id列

可以在模型中指定：

|  |
| --- |
| id = models.AutoField(primary\_key=**True**) |

### 常见field

所有的field类型，见

<https://docs.djangoproject.com/en/2.0/ref/models/fields/#model-field-types>

AutoField

BooleanField

CharField

DateField

DateTimeField

FloatField

SmallIntegerField

IntegerField

TextField

UUIDField：这样使用：

|  |
| --- |
| **import** uuid **from** django.db **import** models  **class** MyUUIDModel(models.Model):  id = models.UUIDField(primary\_key=**True**, default=uuid.uuid4, editable=**False**) |

### field常见参数

* max\_length：字段长度，用于字符串等
* null：如果True，Django将像NULL数据库中那样存储空值。默认是False
* [blank](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.blank)：如果True，该字段被允许为空白。默认是False。请注意，这不同于[null](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.null)。[null](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.null)纯粹是与数据库相关的，而[blank](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.blank)与验证相关。如果一个字段有[blank=True](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.blank)，表单验证将允许输入一个空值。如果一个字段有[blank=False](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.blank)，该字段将是必需的。
* [choices](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.choices)：类似YEAR\_IN\_SCHOOL\_CHOICES = (('FR', 'Freshman'),('SO', 'Sophomore'),('JR', 'Junior'),('SR', 'Senior'),('GR', 'Graduate')) 元组中的第一个元素是将存储在数据库中的值，第二个元素是将在页面中显示的值
* [default](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.default)：字段的默认值
* [help\_text](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.help_text)：用于显示额外的“帮助”文本
* [primary\_key](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.primary_key)：如果True，这个字段是模型的主键
* [unique](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.unique)：如果True，该字段在整个表格中必须是唯一的
* [verbose\_name](https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.Field.verbose_name)：详细字段名，不指定则是属性名的小写，并且用 空格 替换 '\_'

### 模型之间的关系

#### 一对多

使用django.db.models.ForeignKey

例如，如果一个Car模型有一个Manufacturer- 也就是说，一个 Manufacturer多个汽车但Car只有一个 汽车Manufacturer- 使用以下定义：

|  |
| --- |
| **from** django.db **import** models  **class** Manufacturer(models.Model):  *# ...* **pass  class** Car(models.Model):  manufacturer = models.ForeignKey(Manufacturer, on\_delete=models.CASCADE) |

#### 一对一

使用django.db.models.OneToOneField

|  |
| --- |
| **from** django.db **import** models  **class** Place(models.Model):  name = models.CharField(max\_length=50)  address = models.CharField(max\_length=80)   **def** \_\_str\_\_(self):  **return "%s the place"** % self.name  **class** Restaurant(models.Model):  place = models.OneToOneField(  Place,  on\_delete=models.CASCADE,  primary\_key=**True**,  )  serves\_hot\_dogs = models.BooleanField(default=**False**)  serves\_pizza = models.BooleanField(default=**False**)   **def** \_\_str\_\_(self):  **return "%s the restaurant"** % self.place.name  **class** Waiter(models.Model):  restaurant = models.ForeignKey(Restaurant, on\_delete=models.CASCADE)  name = models.CharField(max\_length=50)   **def** \_\_str\_\_(self):  **return "%s the waiter at %s"** % (self.name, self.restaurant) |

#### 多对多

使用django.db.models.ManyToManyField

### 字段命名限制

* 字母，数字，下划线，首字母不能是数字
* 字段名称不能是Python保留字
* 由于Django查询查找语法的工作方式，字段名称不能在一行中包含多个下划线，譬如“abc\_\_123”就是不允许的

## 模型方法

### 新增

save或create：

|  |
| --- |
| p = Person.objects.create(first\_name=**"Bruce"**, last\_name=**"Springsteen"**)  或 p = Person(first\_name=**"Bruce"**, last\_name=**"Springsteen"**) p.save() |

### 修改

|  |
| --- |
| p = Person(first\_name=**"Bruce"**, last\_name=**"Springsteen"**) p.first\_name = **'James'**  *# force\_insert=True或 force\_update=True，强制刷新或者新增*  *# 不能同时使用2个参数，几乎用不上这2个参数*p.save(force\_insert=**True**)  obj = MyModel.objects.create(val=1)  *# 需要使用F来保证不会出现并发冲突* MyModel.objects.filter(pk=obj.pk).update(age=F(**'age'**) + 1)  *# 更新多个值* Entry.objects.filter(pub\_date\_\_year=2007).update(headline=**'Everything is the same'**) |

### 查询

#### 查询方法

|  |
| --- |
| *# 查询所有，得到的QuerySets对象* all\_persons = Person.objects.all()  *# 查询 日期年份 是 2006 的* persons = Person.objects.filter(pub\_date\_\_year=2006)  *# 查询 日期年份 不是2006的* persons = Person.objects.exclude(pub\_date\_\_year=2006)  *# 查询主键等于 1 的* person = Person.objects.get(pk=1)  *# limit 5 ：前5个* persons = Person.objects.filter(pub\_date\_\_year=2006)[:5] *# limit 5,5 ： 第6个到10个* persons = Person.objects.filter(pub\_date\_\_year=2006)[5:10]  *# Entry.objects.all()[-1] 不支持*  *# 返回前10个记录的， 0 ，2 ， 4， 6， 8， 10 ，并且会真实执行，而不是懒惰执行* Entry.objects.all()[:10:2]  *# 可以在结果集上，继续检索*  *# 结果集是一个QuerySet* q1 = Entry.objects.filter(headline\_\_startswith=**"What"**) q2 = q1.exclude(pub\_date\_\_gte=datetime.date.today()) q3 = q1.filter(pub\_date\_\_gte=datetime.date.today())  # 等同于：  q3 = Entry.objects.filter(headline\_\_startswith=**"What"**).exclude(pub\_date\_\_gte=datetime.date.today()).filter(pub\_date\_\_gte=datetime.date.today()) |

检索是懒惰的，只在真实获取值的时候，才会真正去连接数据库获取数据：

|  |
| --- |
| q = Entry.objects.filter(headline\_\_startswith=**"What"**) q = q.filter(pub\_date\_\_lte=datetime.date.today()) q = q.exclude(body\_text\_\_icontains=**"food"**) print(q) *# 此处才会真的去连接数据库获取记录* |

#### 字段查找

字段检索，是在字段名后加 '\_\_' 双下划线，再加关键字，类似 SQL 语句中的 where 后面的部分

在查找中指定的字段必须是模型字段的名称，但有一个例外，如果ForeignKey您可以指定带后缀的字段名称\_id

|  |
| --- |
| Entry.objects.filter(blog\_id=4) |

完整的字段检索文档：

<https://docs.djangoproject.com/en/2.0/ref/models/querysets/#field-lookups>

常见的字段检索：

|  |
| --- |
| *# 确定的搜索 ，SQL： where id = 14* Blog.objects.get(id\_\_exact=14) *# 等同于* Blog.objects.get(id=14) *# 不区分大小写的确定搜索，匹配 beatles blog 、Beatles blog等* Blog.objects.get(name\_\_iexact=**"beatles blog"**)   *# 包含，contains ，SQL：WHERE headline LIKE '%Lennon%'* Entry.objects.get(headline\_\_contains=**'Lennon'**) *# 不区分大小写的包含* Entry.objects.get(headline\_\_icontains=**'Lennon'**)  *# 以什么开头， SQL： WHERE headline LIKE 'Lennon%' # 还有 不区分大小写的 istartwith* Entry.objects.get(headline\_\_startswith=**'Lennon'**) *# 同样有 endswith ，SQL ： WHERE headline LIKE '%Lennon' # 还有 不区分大小写的 iendswith* Entry.objects.get(headline\_\_endswith=**'Lennon'**) |

#### 多对象关联

|  |
| --- |
| *# 检索所有Blog具有至少一个对象Entry ，其headline包含'Lennon'* Blog.objects.filter(entry\_\_headline\_\_contains=**'Lennon'**)  *# Blog中 有一个对象 entry 的 headline 中包含“Lennon”并且是 2008年发布的* Blog.objects.filter(entry\_\_headline\_\_contains=**'Lennon'**).filter(entry\_\_pub\_date\_\_year=2008) Blog.objects.filter(entry\_\_headline\_\_contains=**'Lennon'**, entry\_\_pub\_date\_\_year=2008)  *# 取上面相反的值* Blog.objects.exclude(entry\_\_in=Entry.objects.filter(headline\_\_contains=**'Lennon'**, pub\_date\_\_year=2008, )) |

#### Q对象

|  |
| --- |
| **from** **django.db.models** **import** Q  Q(question\_\_startswith='What')  Poll.objects.get(  Q(question\_\_startswith='Who'),  Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6))  )  Q(question\_\_startswith='Who') | ~Q(pub\_date\_\_year=2005) |

### 删除

|  |
| --- |
| obj.delete() |

### refresh\_from\_db

|  |
| --- |
| *# 重新从数据库刷新读取数据* obj.refresh\_from\_db() |

### 执行SQL

#### 通过模型使用SQL

执行原始SQL语句进行查询，主键字段必须包含在查询的字段中，不然会引发错误

|  |
| --- |
| *# 定义个 person 模型* **class** Person(models.Model):  first\_name = models.CharField()  last\_name = models.CharField()  birth\_date = models.DateField()  *# 执行 原始 SQL* **for** p **in** Person.objects.raw(**'SELECT** *\** **FROM myapp\_person'**):  print(p)  *# 字段先后顺序没关系* Person.objects.raw(**'SELECT id, first\_name, last\_name, birth\_date FROM myapp\_person'**) *# 等同于* Person.objects.raw(**'SELECT last\_name, birth\_date, first\_name, id FROM myapp\_person'**)  *# 可以从其他表格中查询出匹配 person 模型的记录集* Person.objects.raw(**'SELECT first AS first\_name, last AS last\_name,bd AS birth\_date,pk AS id,FROM some\_other\_table'**)  *# 返回的结果集一样可以执行切片* first\_person = Person.objects.raw(**'SELECT** *\** **FROM myapp\_person'**)[0] *# 但是上述语句会返回所有结果，基于节省传输的需要，在数据库缩小结果集范围更正确* first\_person = Person.objects.raw(**'SELECT** *\** **FROM myapp\_person LIMIT 1'**)[0]  *# 传递参数* lname = **'Doe'** Person.objects.raw(**'SELECT** *\** **FROM myapp\_person WHERE last\_name = %s'**, [lname]) |

#### 避开模型使用SQL

不应用模型，直接使用SQL语句进行insert、update和delete

|  |
| --- |
| **from** django.db **import** connection  **def** my\_custom\_sql(obj):  **with** connection.cursor() **as** cursor:  cursor.execute(**"UPDATE bar SET foo = 1 WHERE baz = %s"**, [obj.baz])  cursor.execute(**"SELECT foo FROM bar WHERE baz = %s"**, [obj.baz])  row = cursor.fetchone()   **return** row |

# 视图

视图用于封装负责处理用户请求和返回响应的逻辑，通过urlconfig来进行跳转控制

## URL调度器

### 工作原理

django通过urlconf来映射视图函数，只区分路径，不区分http方法post和get

* Django确定要使用的根URLconf模块，一般是在settings中的ROOT\_URLCONF设置的值，但是如果传入 HttpRequest对象具有一个urlconf 属性（由中间件设置），则其值将用于代替 ROOT\_URLCONF设置。
* Django加载该URLconf模块并查找变量 urlpatterns，它是一个列表django.urls.path() 和 / 或django.urls.re\_path()实例。
* Django按顺序遍历每个URL模式，并停在与请求的URL匹配的第一个URL模式
* 一旦某个URL模式匹配，Django就会导入并调用给定的视图，该视图是一个简单的Python函数（或基于类的视图方法）。该视图通过以下参数传递：
  + 一个实例HttpRequest。
  + 如果匹配的URL模式没有返回任何命名组，则来自正则表达式的匹配作为位置参数提供。
  + 关键字参数由路径表达式匹配的任何命名部分组成，并由可选的kwargs参数传给 django.urls.path()或django.urls.re\_path()。
* 如果没有URL模式匹配，或者在此过程中的任何点发生异常，Django将调用适当的错误处理视图

### 简单示例

|  |
| --- |
| **from** django.urls **import** path  **from** . **import** views  urlpatterns = [  path(**'articles/2003/'**, views.special\_case\_2003),  path(**'articles/<int:year>/'**, views.year\_archive),  path(**'articles/<int:year>/<int:month>/'**, views.month\_archive),  path(**'articles/<int:year>/<int:month>/<slug:slug>/'**, views.article\_detail), ] |

笔记：

* 要从URL中捕获值，请使用尖括号。
* 捕获的值可以选择包含转换器类型。例如，用于 <int:name>捕获整数参数。
* 没有必要添加一个前导斜杠，因为每个URL都有。例如，使用articles而不是/articles。
* 示例请求：
  + 请求/articles/2005/03/匹配列表中的第三个条目。Django会调用这个函数 。views.month\_archive(request, year=2005, month=3)
  + /articles/2003/会匹配列表中的第一个模式，而不是第二个模式，因为模式是按顺序测试的，而第一个模式是第一个要传递的测试。看看利用匹配顺序插入像这样的特殊情况。在这里，Django会调用这个函数 views.special\_case\_2003(request)
  + /articles/2003/ 不匹配任何这些模式，因为每种模式都要求URL以斜线结尾，不过在浏览器访问时，会自动添加 / 。
  + /articles/2003/03/building-a-django-site/将匹配最终模式。Django会调用这个函数 。views.article\_detail(request, year=2003, month=3, slug="building-a-django-site")

### 路径转换器

以下路径转换器默认可用：

* str- 匹配任何非空字符串，不包括路径分隔符'/'。如果转换器不包含在表达式中，这是默认值。
* int - 匹配零或任何正整数。返回一个int。
* slug - 匹配由ASCII字母或数字组成的字符串，以及横线和下划线字符。例如， building-your-1st-django\_site。
* uuid - 匹配格式化的UUID。为防止多个URL映射到同一页面，必须包含破折号，并且字母必须是小写。例如，075194d3-6885-417e-a8a8-6c931e272f00。返回一个 UUID实例。
* path- 匹配任何非空字符串，包括路径分隔符 '/'。这使您可以匹配完整的URL路径，而不仅仅是URL路径的一部分str。

### 自定义路径转换器

转换器是一个包含以下内容的类：

* 一个regex类属性，作为一个re匹配字符串。
* to\_python(self, value)方法，它处理匹配的字符串转换成要传递到视图函数的类型。
* to\_url(self, value)方法，用于处理将Python类型转换为URL中使用的字符串。

使用register\_converter()以下命令在URLconf中注册自定义转换器类 ：

|  |
| --- |
| **from** django.urls **import** register\_converter, path  **from** . **import** converters, views  register\_converter(converters.FourDigitYearConverter, **'yyyy'**)  urlpatterns = [  path(**'articles/2003/'**, views.special\_case\_2003),  path(**'articles/<yyyy:year>/'**, views.year\_archive),  ... ] |

### 使用正则表达式

使用正则表达式匹配路径，请使用 re\_path()而不是path()

在Python正则表达式中，命名正则表达式组的语法是(?P<name>pattern)，其中name是组的名称，并且 pattern是一些要匹配的模式

示例：

|  |
| --- |
| **from** django.urls **import** path, re\_path  **from** . **import** views  *# url() 是 re\_path 的别名，不推荐使用* urlpatterns = [  path(**'articles/2003/'**, views.special\_case\_2003),  path(**'articles/<int:year>/'**, views.year\_archive),  re\_path(**r'^articles/(?P<year>[0-9]{4})/$'**, views.year\_archive),  re\_path(**r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$'**, views.month\_archive),  re\_path(**r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$'**, views.article\_detail), ] |

注意事项：

* 匹配的URL会受到一些限制。例如，年份10000将不再匹配，因为年份整数限制为四位数字。
* 无论正则表达式匹配什么类型，每个捕获的参数都以字符串的形式发送到视图。
* 除了命名的组语法，例如(?P<year>[0-9]{4})，也可以使用较短的未命名组，例如([0-9]{4})，但是不建议这样使用，会引起未知的匹配

嵌套参数：

|  |
| --- |
| **from** django.urls **import** re\_path  urlpatterns = [  *# 不推荐*  re\_path(**r'^blog/(page-(\d+)/)?$'**, blog\_articles), *# 推荐 ，匹配：comments/page-2/ 路径到 comments(request, page\_numer)* re\_path(**r'^comments/(?:page-(?P<page\_number>\d+)/)?$'**, comments), ] |

### 使用默认值

|  |
| --- |
| *# URLconf* **from** django.urls **import** path  **from** . **import** views  urlpatterns = [  *# http://127.0.0.1:8000/polls/blog/*  *# http://127.0.0.1:8000/polls/blog/page1/*  path(**'blog/'**, views.page),  *# http://127.0.0.1:8000/polls/blog/page10*  *# http://127.0.0.1:8000/polls/blog/page99*  path(**'blog/page<int:num>/'**, views.page), ]  *# View (in blog/views.py)* **def** page(request, num=1):  *# Output the appropriate page of blog entries, according to num.* |

### 错误处理

* handler400- 状态码400
* handler403- 状态码403
* handler404- 状态码404
* handler500- 状态码500

### 引用其他URL调度器

|  |
| --- |
| **from** django.urls **import** include, path  **from** apps.main **import** views **as** main\_views **from** credit **import** views **as** credit\_views  extra\_patterns = [  path(**'reports/'**, credit\_views.report),  path(**'reports/<int:id>/'**, credit\_views.report),  path(**'charge/'**, credit\_views.charge), ]  urlpatterns = [  path(**''**, main\_views.homepage),  path(**'help/'**, include(**'apps.help.urls'**)),  path(**'credit/'**, include(extra\_patterns)), ] |

使用include消除重复前缀：

|  |
| --- |
| **from** django.urls **import** path **from** . **import** views  urlpatterns = [  path(**'<page\_slug>-<page\_id>/history/'**, views.history),  path(**'<page\_slug>-<page\_id>/edit/'**, views.edit),  path(**'<page\_slug>-<page\_id>/discuss/'**, views.discuss),  path(**'<page\_slug>-<page\_id>/permissions/'**, views.permissions), ]  *# 修改为：*  **from** django.urls **import** include, path **from** . **import** views  urlpatterns = [  path(**'<page\_slug>-<page\_id>/'**, include([  path(**'history/'**, views.history),  path(**'edit/'**, views.edit),  path(**'discuss/'**, views.discuss),  path(**'permissions/'**, views.permissions),  ])), ] |

传递捕获的参数：

|  |
| --- |
| *# In settings/urls/main.py* **from** django.urls **import** include, path  urlpatterns = [  path(**'<username>/blog/'**, include(**'foo.urls.blog'**)), ]  *# In foo/urls/blog.py* **from** django.urls **import** path **from** . **import** views  urlpatterns = [  path(**''**, views.blog.index),  path(**'archive/'**, views.blog.archive), ] |

### 额外参数

|  |
| --- |
| **from** django.urls **import** path **from** . **import** views  urlpatterns = [  *# 会传递给 views.year\_archive(request, year=2005, foo='bar')* path(**'blog/<int:year>/'**, views.year\_archive, {**'foo'**: **'bar'**}), ] |

### URL反向解析

url调度器除了从用户发起请求，到匹配对应的view，还能在python程序中调用进行匹配

* 在模板中：使用url模板标签
* 在Python代码中：使用该reverse()函数
* 在与处理Django模型实例的URL相关的更高级别的代码中：get\_absolute\_url()方法

示例：

|  |
| --- |
| **from** django.urls **import** path  **from** . **import** views  urlpatterns = [  *#...* path(**'articles/<int:year>/'**, views.year\_archive, name=**'news-year-archive'**),  *#...* ]  *# 模板中：* <a href=**"{% url 'news-year-archive' 2012 %}"**>2012 Archive</a> {*# Or with the year in a template context variable: #}* <ul> {% **for** yearvar **in** year\_list %} <li><a href=**"{% url 'news-year-archive' yearvar %}"**>{{ yearvar }} Archive</a></li> {% endfor %} </ul>  *# python代码中* **from** django.urls **import** reverse **from** django.http **import** HttpResponseRedirect  **def** redirect\_to\_year(request):  *# ...* year = 2006  *# ...* **return** HttpResponseRedirect(reverse(**'news-year-archive'**, args=(year,)))  **"""  在模型中实现方法：  def get\_absolute\_url(self):  from django.urls import reverse  return reverse('people.views.details', args=[str(self.id)])   然后在 模板 中如下使用： """** <a href=**"{{ object.get\_absolute\_url }}"**>{{ object.name }}</a> |

### 命名空间

通过在 url调度器的模块中，定义 app\_name = 'polls' 来命名

|  |
| --- |
| **from** django.urls **import** path  **from** . **import** views  app\_name = **'polls'** urlpatterns = [  path(**''**, views.IndexView.as\_view(), name=**'index'**),  path(**'<int:pk>/'**, views.DetailView.as\_view(), name=**'detail'**),  ... ]  *# 调用*  reverse(**'polls:index'**, current\_app=self.request.resolver\_match.namespace) |

命名空间可以嵌套： name1:name2:index

也可以在单模块中使用嵌套：

|  |
| --- |
| **from** django.urls **import** include, path  **from** . **import** views  polls\_patterns = ([  path(**''**, views.IndexView.as\_view(), name=**'index'**),  path(**'<int:pk>/'**, views.DetailView.as\_view(), name=**'detail'**), ], **'polls'**)  urlpatterns = [  path(**'polls/'**, include(polls\_patterns)), ] |

## 视图功能

### 简单应用

|  |
| --- |
| **from** django.http **import** HttpResponse **import** datetime  **def** current\_datetime(request):  now = datetime.datetime.now()  html = **"<html><body>It is now %s.</body></html>"** % now  **return** HttpResponse(html) |

笔记：

* 首先，HttpResponse从 django.http模块中导入类，以及Python的datetime库。
* 接下来，定义一个叫做的函数current\_datetime。这是查看功能。每个视图函数都将一个HttpRequest 对象作为其第一个参数，该参数通常被命名request。
* 请注意，视图函数的名称无关紧要; 它不必以某种方式命名，以便Django能够识别它。我们在current\_datetime这里叫它，因为这个名字清楚地表明了它的功能。
* 该视图返回一个HttpResponse包含生成的响应的对象。每个视图函数负责返回一个HttpResponse对象。（有例外，但我们会在稍后讨论。）

### 快捷方式

render方法：

必需的参数：

request：用于生成此响应的请求对象。

template\_name：要使用的模板的全名或模板名称的序列。

可选参数：

context：要添加到模板上下文的值的字典。默认情况下，这是一个空字典。如果字典中的值是可调用的，视图将在渲染模板之前调用它。

content\_type：用于生成文档的MIME类型。默认为DEFAULT\_CONTENT\_TYPE设置的值。

status：响应的状态码。默认为200。

using：该NAME模板引擎的使用加载的模板。

|  |
| --- |
| **from** django.shortcuts **import** render  **def** my\_view(request):  *# View code here...* **return** render(request, **'myapp/index.html'**, {  **'foo'**: **'bar'**,  }, content\_type=**'application/xhtml+xml'**)  *# 等同于*  **from** django.http **import** HttpResponse **from** django.template **import** loader  **def** my\_view(request):  *# View code here...* t = loader.get\_template(**'myapp/index.html'**)  c = {**'foo'**: **'bar'**}  **return** HttpResponse(t.render(c, request), content\_type=**'application/xhtml+xml'**) |

redirect：

通过传递一些对象; 该对象的 get\_absolute\_url()方法将被调用以找出重定向URL：

|  |
| --- |
| **from** django.shortcuts **import** redirect  **def** my\_view(request):  ...  object = MyModel.objects.get(...)  **return** redirect(object) |

通过传递一个视图的名称和可选的一些位置或关键字参数; 该URL将使用以下reverse()方法反向解析 ：

|  |
| --- |
| **def** my\_view(request):  ...  **return** redirect(**'some-view-name'**, foo=**'bar'**) |

通过传递硬编码的URL重定向到：

|  |
| --- |
| **def** my\_view(request):  ...  **return** redirect(**'/some/url/'**) |

默认情况下，redirect()返回一个临时重定向（302）。所有上述形式都接受permanent参数; 如果设置为True永久重定向（301）将被返回：

|  |
| --- |
| **def** my\_view(request):  ...  object = MyModel.objects.get(...)  **return** redirect(object, permanent=**True**) |

get\_object\_or\_404，get\_list\_or\_404：

必需的参数：

klass：一Model类，一个Manager，或QuerySet从哪个实例来获取对象

\*\*kwargs：查找参数，应该采用get()和所 接受的格式filter()。

|  |
| --- |
| **from** django.shortcuts **import** get\_object\_or\_404  **def** my\_view(request):  poll = get\_object\_or\_404(MyModel, pk=1)  *#等同于* **def** my\_view(request):  **try**:  poll = Poll.objects.get(pk=1)  **except** Poll.DoesNotExist:  **raise** Http404() |

### 返回错误

|  |
| --- |
| **from** django.http **import** HttpResponse, HttpResponseNotFound  **def** my\_view(request):  *# ...* **if** foo:  **return** HttpResponseNotFound(**'<h1>Page not found</h1>'**)  **else**:  **return** HttpResponse(**'<h1>Page was found</h1>'**)  *# 还可以直接返回错误代码* **from** django.http **import** HttpResponse  **def** my\_view(request):  *# ...   # Return a "created" (201) response code.* **return** HttpResponse(status=201)  *# 特殊的 404 错误* **from** django.http **import** Http404 **from** django.shortcuts **import** render **from** polls.models **import** Poll  **def** detail(request, poll\_id):  **try**:  p = Poll.objects.get(pk=poll\_id)  **except** Poll.DoesNotExist:  **raise** Http404(**"Poll does not exist"**)  **return** render(request, **'polls/detail.html'**, {**'poll'**: p}) |

## 视图装饰器

|  |
| --- |
| **from** django.views.decorators.http **import** require\_http\_methods  *# 允许 GET和POST方法* @require\_http\_methods([**"GET"**, **"POST"**]) **def** my\_view(request):  *# I can assume now that only GET or POST requests make it this far  # ...* **pass** |

require\_GET()：仅仅允许GET方法

require\_POST()：仅仅允许POST方法.

require\_safe()：仅仅允许GET和HEAD方法

## 请求和响应

详细的文档见：

<https://docs.djangoproject.com/en/2.0/ref/request-response/>

### HttpRequest

属性：

除非另有说明，否则所有属性均应视为只读。

HttpRequest.scheme：

表示请求方案的字符串（http或https 通常）。

HttpRequest.body：

原始HTTP请求主体作为字节串。这对于以不同于传统HTML表单的方式处理数据很有用：二进制图像，XML有效载荷等。用于处理传统表单数据，请使用 HttpRequest.POST，取url中的参数，用HttpRequest.GET。

HttpRequest.path：

表示请求页面的完整路径的字符串，不包括scheme和域。

例： "/music/bands/the\_beatles/"

HttpRequest.path\_info：

在某些Web服务器配置下，主机名后的URL部分被分成脚本前缀部分和路径信息部分。path\_info无论使用什么Web服务器，该属性始终包含路径的路径信息部分。使用此代替path可以使代码更容易在测试和部署服务器之间移动。

例如，如果WSGIScriptAlias你的应用程序设置为 "/minfo"，则path可能是"/minfo/music/bands/the\_beatles/" 和path\_info会"/music/bands/the\_beatles/"。

HttpRequest.method：

表示请求中使用的HTTP方法的字符串。这保证是大写的。例如：

if request.method == 'GET':

do\_something()

elif request.method == 'POST':

do\_something\_else()

HttpRequest.encoding：

表示当前编码的字符串，用于解码表单提交数据（或者None，表示使用该DEFAULT\_CHARSET设置）。您可以写入此属性来更改访问表单数据时使用的编码。任何后续的属性访问（例如读取GET或POST）将使用新encoding值。

HttpRequest.content\_type：

表示请求的MIME类型的字符串，从CONTENT\_TYPE标头解析 。

HttpRequest.content\_params：

包含在CONTENT\_TYPE 标题中的键/值参数字典。

HttpRequest.GET：

包含所有给定的HTTP GET参数的类似字典的对象。请参阅QueryDict下面的 文档。

HttpRequest.POST：

包含所有给定HTTP POST参数的类似字典的对象，前提是请求包含表单数据。请参阅QueryDict下面的 文档。如果您需要访问请求中发布的原始数据或非表单数据，请HttpRequest.body改为通过属性访问此数据 。

有可能请求可以通过POST发送一个空POST 字典 - 例如，如果通过POST HTTP方法请求表单但不包含表单数据。因此，您不应该使用 检查POST方法的用法; 相反，使用（见）。if request.POST:

if request.method == "POST":

HttpRequest.method

POST它不包含文件上传信息。看FILES。

HttpRequest.COOKIES：

包含所有Cookie的字典。键和值是字符串。

HttpRequest.FILES：

包含所有上传文件的类似字典的对象。每个关键 FILES是name从。每个值都是一个。<input type="file" name="" />FILESUploadedFile

FILES将只包含数据，如果请求方法是POST和<form>发布到请求的方法 enctype="multipart/form-data"。否则，FILES将是一个空白字典式的对象。

HttpRequest.META：

包含所有可用HTTP标头的字典。可用的头文件取决于客户端和服务器

应用程序代码设置的属性：

Django本身并没有设置这些属性，但是如果你的应用程序设置了它们，就可以使用它们。

HttpRequest.current\_app：

该url模板标签将使用它的值作为current\_app 参数reverse()。

HttpRequest.urlconf：

这将用作当前请求的根URLconf，覆盖ROOT\_URLCONF设置。

中间件设置的属性：

Django的contrib应用程序中包含的一些中间件在请求中设置了属性。如果您在请求中看不到该属性，请确保列出了相应的中间件类MIDDLEWARE。

HttpRequest.session：

来自SessionMiddleware：代表当前会话的可读写字典对象。

HttpRequest.site：

来自CurrentSiteMiddleware： 代表当前网站的实例Site或 RequestSite返回 get\_current\_site()。

HttpRequest.user：

来自AuthenticationMiddleware：AUTH\_USER\_MODEL代表当前登录用户的实例

### QueryDict

在一个HttpRequest对象中，GET和 POST属性是实例django.http.QueryDict，定制为相同的键的多个值的字典状类

|  |
| --- |
| QueryDict(**'a=1&a=2&c=3'**) *#<QueryDict: {'a': ['1', '2'], 'c': ['3']}>* |

QueryDict.get（键，默认=无）：

使用相同的逻辑\_\_getitem\_\_()，如果该键不存在，则使用挂钩返回默认值。

QueryDict.setdefault（key，默认=无）[source] ：

喜欢dict.setdefault()，除了它在\_\_setitem\_\_()内部使用。

QueryDict.update（other\_dict）：

采取一个QueryDict或字典

|  |
| --- |
| **>>>** q = QueryDict('a=1', mutable=**True**)  **>>>** q.update({'a': '2'})  **>>>** q.getlist('a')  ['1', '2']  **>>>** q['a'] *# returns the last*  '2' |

QueryDict.items（）：

像dict.items()

|  |
| --- |
| **>>>** q = QueryDict('a=1&a=2&a=3')  **>>>** list(q.items())  [('a', '3')] |

### HttpResponse

传递字符串：

|  |
| --- |
| **>>> from** **django.http** **import** HttpResponse  **>>>** response = HttpResponse("Here's the text of the Web page.")  **>>>** response = HttpResponse("Text only, please.", content\_type="text/plain") |

属性:

HttpResponse.content:

表示内容的字符串，如果需要，从字符串编码。

HttpResponse.charset:

表示响应将被编码的字符集的字符串。如果在HttpResponse实例化时没有给出，则会从中提取 content\_type，如果不成功，DEFAULT\_CHARSET则会使用该设置。

HttpResponse.status\_code:

该 响应的 HTTP状态码。

除非reason\_phrase明确设置，否则修改 status\_code构造函数外部的值也会修改值 reason\_phrase。

HttpResponse.reason\_phrase:

响应的HTTP原因短语。它使用HTTP标准的默认原因短语。

除非明确设置，否则reason\_phrase由值决定 status\_code。

HttpResponse.streaming:

这总是False。

该属性存在，因此中间件可以将流式响应与常规响应区别开来。

HttpResponse.closed:

True 如果response已经结束。

### JsonResponse

|  |
| --- |
| **>>> from** **django.http** **import** JsonResponse  **>>>** response = JsonResponse({'foo': 'bar'})  **>>>** response.content  b'{"foo": "bar"}' |

### FileResponse

|  |
| --- |
| **>>> from** **django.http** **import** FileResponse  **>>>** response = FileResponse(open('myfile.png', 'rb')) |

## 内置视图

包含在 **from** django.views **import** generic 中

具体视图类型：

\_\_all\_\_ = [  
 **'View'**, **'TemplateView'**, **'RedirectView'**, **'ArchiveIndexView'**,  
 **'YearArchiveView'**, **'MonthArchiveView'**, **'WeekArchiveView'**, **'DayArchiveView'**,  
 **'TodayArchiveView'**, **'DateDetailView'**, **'DetailView'**, **'FormView'**,  
 **'CreateView'**, **'UpdateView'**, **'DeleteView'**, **'ListView'**, **'GenericViewError'**,  
]

用法：

|  |
| --- |
| *# views：*  **class** DetailView(generic.DetailView):  *"""  使用通用的详情视图  """  # 对应的模型对象* model = Question  *# 使用的模板名* template\_name = **'detail.html'**  *# template：*  <**h1**>{{ **question**.**question\_text** }}</**h1**>  {% **if error\_message** %}<**p**><**strong**>{{ **error\_message** }}</**strong**></**p**>{% **endif** %}  <**form action="**{% **url 'polls:vote' question**.**id** %}**" method="post"**> {% **csrf\_token** %} {% **for choice in question**.**choice\_set**.**all** %}  <**input type="radio" name="choice" id="choice**{{ **forloop**.**counter** }}**" value="**{{ **choice**.**id** }}**"** />  <**label for="choice**{{ **forloop**.**counter** }}**"**>{{ **choice**.**choice\_text** }}</**label**><**br** /> {% **endfor** %} <**input type="submit" value="Vote"** /> </**form**>  *# urls：*  path(**'<int:pk>/'**, views.DetailView.as\_view(), name=**'detail'**), |

# 模板

作为一个Web框架，Django需要一种方便的方式来动态生成HTML。最常用的方法依赖于模板。模板包含所需HTML输出的静态部分以及描述如何插入动态内容的特殊语法。

对模板引擎的一般支持和Django模板语言的实现都存在于django.template 命名空间中

## 配置

在settings中配置：

|  |
| --- |
| *# 配置模板* 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'**,  ],  },  }, ] |

* BACKEND：是实现Django模板后端API的模板引擎类的路径。内置是django.template.backends.django.DjangoTemplates和 django.template.backends.jinja2.Jinja2（使用这个需要额外安装jinja2库）
* DIRS ：按搜索顺序定义引擎应该查找模板源文件的目录列表。
* APP\_DIRS：告诉引擎是否应该在已安装的应用程序中查找模板。每个后端为其模板应存储在的应用程序内的子目录定义一个常规名称。
* OPTIONS 包含后端特定的设置

## 用法

假如settings中配置如下：

|  |
| --- |
| TEMPLATES = [  {  **'BACKEND'**: **'django.template.backends.django.DjangoTemplates'**,  **'DIRS'**: [  **'/home/html/example.com'**,  **'/home/html/default'**,  ],  },  {  **'BACKEND'**: **'django.template.backends.jinja2.Jinja2'**,  **'DIRS'**: [  **'/home/html/jinja2'**,  ],  }, ] |

### get\_template

该函数使用给定名称加载模板并返回一个 Template对象

|  |
| --- |
| **get\_template('story\_detail.html')**  Django将要查找的文件，依次为：   * **/home/html/example.com/story\_detail.html**（**'django'**发动机） * **/home/html/default/story\_detail.html**（**'django'**发动机） * **/home/html/jinja2/story\_detail.html**（**'jinja2'**发动机） |

|  |
| --- |
| **from** django.shortcuts **import** render  **def** my\_view(request):  *# View code here...* **return** render(request, **'myapp/index.html'**, {  **'foo'**: **'bar'**,  }, content\_type=**'application/xhtml+xml'**)  # 相当于：  **from** django.http **import** HttpResponse **from** django.template **import** loader  **def** my\_view(request):  *# View code here...* t = loader.get\_template(**'myapp/index.html'**)  c = {**'foo'**: **'bar'**}  **return** HttpResponse(t.render(c, request), content\_type=**'application/xhtml+xml'**) |

### select\_template

select\_template()就像get\_template()，除了它需要一个模板名称的列表。它按顺序尝试每个名称并返回存在的第一个模板

|  |
| --- |
| **select\_template(['story\_253\_detail.html','story\_detail.html'])**  Django将要查找的文件，依次为：   * **/home/html/example.com/story\_253\_detail.html**（**'django'**发动机） * **/home/html/default/story\_253\_detail.html**（**'django'**发动机） * **/home/html/jinja2/story\_253\_detail.html**（**'jinja2'**发动机） * **/home/html/example.com/story\_detail.html**（**'django'**发动机） * **/home/html/default/story\_detail.html**（**'django'**发动机） * **/home/html/jinja2/story\_detail.html**（**'jinja2'**发动机）   当Django发现存在的模板时，它停止查找 |

## 模板语言

Django模板只是一个文本文档或使用Django模板语言标记的Python字符串。一些结构被模板引擎识别和解释。主要的是变量和标签。

### 变量

一个变量从上下文中输出一个值，这是一个类似字典的对象将键映射到值。

变量被这样包围 {{变量}} 像这样：

|  |
| --- |
| My first name is {{ **first\_name** }}. My last name is {{ **last\_name** }}. |

显示：

{**'first\_name'**: **'John'**, **'last\_name'**: **'Doe'**}

|  |
| --- |
| My first name is John. My last name is Doe. |

字典查找，属性查找和列表索引查找用点符号实现：

|  |
| --- |
| {{ my\_dict.key }}  {{ my\_object.attribute }}  {{ my\_list.0 }} |

### 标签

标签在渲染过程中提供任意的逻辑。

例如，标签可以输出内容，用作控制结构，

例如“if”语句或“for”循环，从数据库中获取内容，甚至可以访问其他模板标签。

所有标签见：

<https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#ref-templates-builtins-tags>

#### 基本用法

标签被包围 {%并且%} 像这样：

|  |
| --- |
| {% **csrf\_token** %} |

接收参数：

|  |
| --- |
| {% **cycle** 'odd' 'even' %} |

#### if

判断标签 if endif ：

|  |
| --- |
| {% **if** athlete\_list %}  Number of athletes: {{ athlete\_list|length }}  {% **elif** athlete\_in\_locker\_room\_list %}  Athletes should be out of the locker room soon!  {% **else** %}  No athletes.  {% **endif** %}  上面{{ athlete\_list|length }} ，如果 athlete\_list 不为空，则显示athlete\_list的长度 |

#### firstof

输出不是False的第一个参数，所有参数都为False，则什么都不输出

|  |
| --- |
| {% **firstof** var1 var2 var3 %}  等同于  {% **if** var1 %}  {{ var1 }}  {% **elif** var2 %}  {{ var2 }}  {% **elif** var3 %}  {{ var3 }}  {% **endif** %}  使用默认值：  {% **firstof** var1 var2 var3 "默认值" %}  这个标签会自动转义，如果不想转义：  {% **autoescape** off %}  {% **firstof** var1 var2 var3 "<strong>fallback value</strong>" %}  {% **endautoescape** %}  部分参数不想被转义：  {% **firstof** var1 var2|safe var3 "<strong>fallback value</strong>"|safe %} |

#### for

循环， endfor结束 ，可以遍历列表，字典等

|  |
| --- |
| <**ul**>  {% **for** athlete **in** athlete\_list %}  <**li**>{{ athlete.name }}</**li**>  {% **endfor** %}  </**ul**>  反向循环：  {% **for** athlete **in** athlete\_list reversed%}  字典：  {% **for** key, value **in** data.items %}  {{ key }}: {{ value }}  {% **endfor** %}  设置默认值， for empty ，类似python的 for else：  <**ul**>  {% **for** athlete **in** athlete\_list %}  <**li**>{{ athlete.name }}</**li**>  {% **empty** %}  <**li**>Sorry, no athletes in this list.</**li**>  {% **endfor** %}  </**ul**>  等同于，但是比下面的写法更加简便：  <**ul**>  {% **if** athlete\_list %}  {% **for** athlete **in** athlete\_list %}  <**li**>{{ athlete.name }}</**li**>  {% **endfor** %}  {% **else** %}  <**li**>Sorry, no athletes in this list.</**li**>  {% **endif** %}  </**ul**> |

for循环在循环中设置了许多变量：

| **变量** | **描述** |
| --- | --- |
| **forloop.counter** | 循环的当前迭代（1索引） |
| **forloop.counter0** | 循环的当前迭代（0索引） |
| **forloop.revcounter** | 循环结束时的迭代次数（1索引） |
| **forloop.revcounter0** | 循环结束时的迭代次数（0索引） |
| **forloop.first** | 如果这是通过循环的第一次，则为真 |
| **forloop.last** | 如果这是通过循环的最后一次，则为真 |
| **forloop.parentloop** | 对于嵌套循环，这是围绕当前循环的循环 |

#### 布尔运算符

and、or和not

|  |
| --- |
| {% **if** athlete\_list **and** coach\_list %}  Both athletes and coaches are available.  {% **endif** %}  {% **if** **not** athlete\_list %}  There are no athletes.  {% **endif** %}  {% **if** athlete\_list **or** coach\_list %}  There are some athletes or some coaches.  {% **endif** %}  {% **if** **not** athlete\_list **or** coach\_list %}  There are no athletes or there are some coaches.  {% **endif** %}  {% **if** athlete\_list **and** **not** coach\_list %}  There are some athletes and absolutely no coaches.  {% **endif** %} |

允许在同一个标签中使用两个and和or子句， and优先级高于or例如：

|  |
| --- |
| {% **if** athlete\_list **and** coach\_list **or** cheerleader\_list %}  将被解释为：  **if** (athlete\_list **and** coach\_list) **or** cheerleader\_list |

但是在if标签中使用实际的括号是无效的语法。如果你需要它们来表示优先级，你应该使用嵌套if标签

#### 逻辑运算符

==, !=, <, >, <=, >=, in, not in, is, 和 is not

|  |
| --- |
| {% **if** somevar == "x" %}  This appears if variable somevar equals the string "x"  {% **endif** %}  {% **if** "bc" **in** "abcdef" %}  This appears since "bc" is a substring of "abcdef"  {% **endif** %}  {% **if** somevar **is** **not** True %}  This appears if somevar is not True, or if somevar is not found in the  context.  {% **endif** %} |

优先级，从低到高：

* or
* and
* not
* in
* ==，!=，<，>，<=，>=

#### 过滤器

|  |
| --- |
| {% **if** messages|length >= 100 %}  You have lots of messages today!  {% **endif** %} |

messages|length这里判断 messages 不为空，并且长度大于等于100

add：相加(减法：|add:-2 )

|  |
| --- |
| {{ value|add:"2" }}  如果value是4，那么输出6  {{ first|add:second }}  如果first=[1,2,3] ，second=[4,5,6]，那么输出：[1,2,3,4,5,6] |

divisibleby：能否整除，返回 True和False

|  |
| --- |
| {{ value|divisibleby:"2" }} |

widthratio：乘法和除法

|  |
| --- |
| {% **widthratio** this\_value max\_value max\_width %}  结果是： this\_value/max\_value\*max\_width  如果要计算 value \* 10 ： widthratio value 1 10  如果要计算 value / 8 ： widthratio 8 1  还可以这样使用：  {% **widthratio** this\_value max\_value max\_width **as** width %}  The width is: {{ width }} |

addslashes：单引号前加 \

|  |
| --- |
| {{ value|addslashes }}  value="I'm using Django"，输出："I\'m using Django". |

capfirst：首字母大写

|  |
| --- |
| {{ value|capfirst }}  如果**value**是**"django"**，输出将是**"Django"**。 |

center：将值置于给定宽度的字段中。

|  |
| --- |
| "{{ value|center:"15" }}"  如果**value**是**"Django"**，输出将是。**"** **Django** **"，前面5个，后面4个空格** |

cut：删除给定字符

|  |
| --- |
| {{ value|cut:" " }}  如果value是"String with spaces"，输出将是："Stringwithspaces" |

date：日期字符串

|  |
| --- |
| 格式字符 描述 示例 输出  b 月，文字，3个字母，小写。 'jan'  d 每月的一天，2位数字前导零。 '01' 至 '31'  D 星期几，文字，3个字母。 'Fri'  e 时区名称。可能是任何格式，或者可能会返回一个空字符串，具体取决于日期时间。 ''，'GMT'，'-500'，'US/Eastern'，等。  E 月，通常用于长日期表示的区域设置特定备选表示。 'listopada'（对于波兰语区而言'Listopad'）  f 时间在12小时和分钟之内，如果它们为零，则分钟时间不再。专有扩展。 '1'， '1:30'  F 月，文字，长。 'January'  g 小时，12小时制，无前导零。 '1' 至 '12'  G 小时，24小时制，无前导零。 '0' 至 '23'  h 小时，12小时制。 '01' 至 '12'  H 小时，24小时制。 '00' 至 '23'  i 分钟。 '00' 至 '59'  I 夏令时，无论是否有效。 '1' or '0'  J 没有前导零的月份的一天。 '1' 至 '31'  n 月没有前导零。 '1' 至 '12'  N 月份 'Jan.', 'Feb.', 'March', 'May'  s 秒 00-59  t 给定月份的天数。 28 至 31  T 这台机器的时区。 'EST'， 'MDT'  u 微秒。 000000 至 999999  U Unix时代以来的秒数（1970年1月1日00:00:00 UTC）。  w ^ 没有前导零的数字。 '0'（星期日）至'6'（星期六）  W ^ ISO-8601周数，周数从周一开始。 1， 53  y 年，2位数字。 '99'  Y 年，4位数字。 '1999'  z 一年中的一天。 0 至 365  Z 以秒为单位的时区偏移量。UTC以西时区的偏移总是负值，而UTC以东的偏移总是正值。 -43200 至 43200 |

|  |
| --- |
| {{ value|date:"D d M Y" }}  {{ value|date:"Y-m-d H:i:s" }}  value：**datetime.datetime.now()** |

#### include

包含另外一个模板

模板名称可以是变量，也可以是单引号或双引号的硬编码（带引号）的字符串

|  |
| --- |
| {% **include** "foo/bar.html" %}  {% **include** template\_name %}  该变量也可以是任何具有**render()**接受上下文的方法的对象 |

include传参数：

下面这个例子显示：

"Hello, John!"

* 上下文：变量person设置为"John"，变量greeting 设置为"Hello"
* name\_snippet.html模板

|  |
| --- |
| {{ greeting }}, {{ person|default:"friend" }}! |

* 原模板通过with传递参数

|  |
| --- |
| {% **include** "name\_snippet.html" **with** person="Jane" greeting="Hello" %} |

如果只想使用include传递的参数（甚至没有参数）渲染模板，使用only：

|  |
| --- |
| {% **include** "name\_snippet.html" **with** greeting="Hi" only %} |

#### url

给定视图和可选参数匹配的绝对路径引用（没有域名的URL）

|  |
| --- |
| {% **url** 'some-url-name' v1 v2 %}  第一个参数是url模式名称，后面跟着的是参数，以空格分隔  可以使用关键字：  {% **url** 'some-url-name' arg1=v1 arg2=v2 %}  如果您想检索命名空间的URL，请指定完全限定的名称：  {% **url** 'myapp:view-name' %} |

#### with

简单的名称缓存复杂变量，当访问多次耗时的方法（例如操作数据库的方法）

|  |
| --- |
| {% **with** total=business.employees.count%}  {{ total }} employee{{ total|pluralize }}  {% **endwith** %}  上述的：{% **with** total=business.employees.count%}  等同于：{% **with** business.employees.count as total %} |

# 中间件

中间件是Django请求/响应处理的钩子框架。这是一个轻量级的低级“插件”系统，用于在全局范围内改变Django的输入或输出。

每个中间件组件都负责完成一些特定的功能。例如，Django包含一个中间件组件 AuthenticationMiddleware，它使用会话将用户与请求相关联。

中间件的执行顺序，在settings中配置，request时，是从第一个到最后一个，response时，则反过来，从最后一个到第一个

## 中间件函数

|  |
| --- |
| **def** simple\_middleware(get\_response):  *"""  自定义中间件函数，配置信息在 里层的 middleware函数 外定义  只会调用一次* **:param** *get\_response: 如果本中间件是最后的中间件，那么这个参数是最后的视图函数，  如果不是最后一层中间件，那么这个参数是后一层的中间件的函数* **:return***: HttpResponse  """* **def** middleware(request):  *"""  每个reqeust都会执行* **:param** *request: HttpRequest* **:return***: HttpResponse  """   # 在这里实现 request 的过滤操作* response = get\_response(request)   *# 在这里实现 response 的过滤操作* **return** response   **return** middleware |

## 中间件类

|  |
| --- |
| **class** SimpleMiddleware:  **def** \_\_init\_\_(self, get\_response):  *"""  自定义中间件类，配置信息在 这里定义  只会调用一次  不允许添加任何的其他参数* **:param** *get\_response: 如果本中间件是最后的中间件，那么这个参数是最后的视图函数，  如果不是最后一层中间件，那么这个参数是后一层的中间件的函数  """* self.get\_response = get\_response    **def** \_\_call\_\_(self, request):  *# 在这里实现 request 的过滤操作* response = self.get\_response(request)   *# 在这里实现 response 的过滤操作* **return** response |

## 使用中间件

在settings中配置：

|  |
| --- |
| MIDDLEWARE = [  **'django.middleware.security.SecurityMiddleware'**,  **'django.contrib.sessions.middleware.SessionMiddleware'**,  **'django.middleware.common.CommonMiddleware'**,  **'django.middleware.csrf.CsrfViewMiddleware'**,  **'django.contrib.auth.middleware.AuthenticationMiddleware'**,  **'django.contrib.messages.middleware.MessageMiddleware'**,  **'django.middleware.clickjacking.XFrameOptionsMiddleware'**,   **'polls.middleware.polls\_middleware.SimpleMiddleware'**,  **'polls.middleware.polls\_middleware.simple\_middleware'**,  ] |

# 自定义错误页面

需要在settings中，配置：

|  |
| --- |
| DEBUG = **False** |

在urls.py 中添加：

|  |
| --- |
| *# 404 错误视图函数* handler404 = views.page\_not\_found *# 500 错误视图函数* handler500 = views.page\_error |

在views中添加：

|  |
| --- |
| **def** page\_not\_found(request):  **return** render\_to\_response(**'404.html'**)  **def** page\_error(request):  **return** render\_to\_response(**'500.html'**) |

在templates中增加 404.html 和 500.html 页面

# 登录模块

登录模块可以使用django的auth模块，也可以自己实现，下面演示使用django自带的

## 启用AuthenticationMiddleware

在settings中配置增加：

|  |
| --- |
| INSTALLED\_APPS = [  **'django.contrib.auth'**,  ]  MIDDLEWARE = [  **'django.contrib.sessions.middleware.SessionMiddleware'**,  **'django.contrib.auth.middleware.AuthenticationMiddleware'**, ] |

## 编写login页面

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head lang="en"**>  <**meta charset="UTF-8"**>  <**title**></**title**> </**head**> <**body**> <**div class="login"**>  <**div style="font-size**: 25**px**; **font-weight**: **bold**;**text-align**: **center**;**"**>  用户登陆  </**div**>  {% **if error\_message** %}<**p**><**strong**>{{ **error\_message** }}</**strong**></**p**>{% **endif** %}  <**form role="form" action="**{% **url 'polls:login\_submit'** %}**" method="POST"**>  {% **csrf\_token** %}  <**div class="form-group"**>  <**label for="username"**>用户名</**label**>  <**input type="text" name="username" placeholder="请输入用户名"**>  </**div**>  <**div class="form-group"**>  <**label for="password"**>密码</**label**>  <**input type="password" name="password" placeholder="请输入密码"**>  </**div**>  <**button type="submit"**>登 陆</**button**>  </**form**> </**div**> </**body**> </**html**> |

## 修改urls.py

|  |
| --- |
| path(**'login/'**, views. polls\_login, name=**'login'**), path(**'login\_submit/'**, views.login\_submit, name=**'login\_submit'**), |

## 修改 views.py

|  |
| --- |
| **def** polls\_login(request):  error\_message = request.session.get(**'error\_message'**)  request.session[**'error\_message'**] = **None  return** render(request, **'login.html'**, {**'error\_message'**: error\_message})  **def** login\_submit(request):  username = request.POST[**'username'**]  password = request.POST[**'password'**]  user = authenticate(username=username, password=password)  **if** user **is not None**:  **if** user.is\_active:  login(request, user)  **return** HttpResponseRedirect(reverse(**'polls:index'**))  **else**:  request.session[**'error\_message'**] = **"用户已经停用！"  else**:  request.session[**'error\_message'**] = **"用户名或密码错误！"   return** HttpResponseRedirect(reverse(**'polls:login'**)) |

登出使用：

|  |
| --- |
| **def** logout\_view(request):  logout(request)  **return** HttpResponseRedirect(reverse(**'polls:login'**)) |

## 限制必须登录才能访问

|  |
| --- |
| **def** my\_view(request):  **if** **not** request.user.is\_authenticated:  **return** render(request, 'myapp/login\_error.html')  *...* |

或者使用装饰器：

|  |
| --- |
| @login\_required  **def** my\_view(request):  ...  # 用户未登录，则重定向到[**settings.LOGIN\_URL**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-LOGIN_URL)  @login\_required(login\_url='/polls/login/')  **def** my\_view(request): |

PS：login\_url的值必须是绝对路径，以 '/' 开头，是 url 中的所有的path

通用视图类的处理：

|  |
| --- |
| **from** **django.contrib.auth.mixins** **import** LoginRequiredMixin  **class** **MyView**(LoginRequiredMixin, View):  login\_url = '/login/'  redirect\_field\_name = 'redirect\_to' |

PS：在settings中设置了 LOGIN\_URL 后， 通用视图类中的login\_url和redirect\_field\_name都可以省略不写

## 使用验证码

### 生成验证码

|  |
| --- |
| **import** os **from** PIL **import** Image, ImageDraw, ImageFont, ImageFilter **import** random   *# 随机码 默认长度=1* **def** random\_code(lenght=1):  code = **''  for** char **in** range(lenght):  code += chr(random.randint(65, 90))  **return** code   *# 随机颜色 默认颜色范围【1，255】* **def** random\_color(s=1, e=255):  **return** (random.randint(s, e), random.randint(s, e), random.randint(s, e))   *# 生成验证码图片 # length 验证码长度 # width 图片宽度 # height 图片高度 # 返回验证码和图片* **def** veri\_code(lenght=4, width=160, height=40):  *# 创建Image对象* image = Image.new(**'RGB'**, (width, height), (255, 255, 255))  *# 创建Font对象* file = os.path.dirname(os.path.abspath(\_\_file\_\_))  font = ImageFont.truetype(**f'{file}/ubuntu.ttf'**, 32)  *# 创建Draw对象* draw = ImageDraw.Draw(image)  *# 随机颜色填充每个像素* **for** x **in** range(width):  **for** y **in** range(height):  draw.point((x, y), fill=random\_color(64, 255))  *# 验证码* code = random\_code(lenght)  *# 随机颜色验证码写到图片上* **for** t **in** range(lenght):  draw.text((40 \* t + 5, 5), code[t], font=font, fill=random\_color(32, 127))  *# 模糊滤镜* image = image.filter(ImageFilter.BLUR)  **return** image, code   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  img, strs = veri\_code()  **with** open(**'test.png'**, **'wb'**) **as** f:  img.save(f) |

### login页面修改

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head lang="en"**>  <**meta charset="UTF-8"**>  <**title**></**title**> </**head**> <**body**> <**div class="login"**>  <**div style="font-size**: 25**px**; **font-weight**: **bold**;**text-align**: **center**;**"**>  用户登陆  </**div**>  {% **if error\_message** %}<**p**><**strong**>{{ **error\_message** }}</**strong**></**p**>{% **endif** %}  <**form role="form" action="**{% **url 'polls:login\_submit'** %}**" method="POST"**>  {% **csrf\_token** %}  <**div class="form-group"**>  <**label for="username"**>用户名</**label**>  <**input type="text" name="username" placeholder="请输入用户名"**>  </**div**>  <**div class="form-group"**>  <**label for="password"**>密码</**label**>  <**input type="password" name="password" placeholder="请输入密码"**>  </**div**>  <**div class="form-group"**>  <**label for="password"**>验证码</**label**>   <**div class="row"**>  <**div class="col-xs-7"**>  <**input type="text" class="form-control" placeholder="请输入验证码" name="check\_code"**>  </**div**>  <**div class="col-xs-5"**>  <**img src="**{% **url 'polls:check\_code'** %}**" onclick="***changeCheckCode*(**this**);**"**>  </**div**>  </**div**>  </**div**>  <**button type="submit"**>登 陆</**button**>  </**form**> </**div**> <**script**>  **function** *changeCheckCode*(ths){  ths.**src** = ths.**src** + **'?'**; *<!--改变URL，刷新图片。-->* } </**script**> </**body**> </**html**> |

### urls.py修改

|  |
| --- |
| path(**'check\_code/'**, views.check\_code, name=**'check\_code'**), |

### views.py修改

|  |
| --- |
| **def** check\_code(request):  stream = BytesIO()  img, code = veri\_code()  img.save(stream, **'PNG'**)  request.session[**'check\_code'**] = code  **return** HttpResponse(stream.getvalue())  **def** login\_submit(request):  check\_code = request.POST[**'check\_code'**]   **if** check\_code.upper() != request.session[**'check\_code'**].upper():  request.session[**'error\_message'**] = **"验证码错误！"  else**:  username = request.POST[**'username'**]  password = request.POST[**'password'**]  user = authenticate(username=username, password=password)  **if** user **is not None**:  **if** user.is\_active:  login(request, user)  **return** HttpResponseRedirect(reverse(**'polls:index'**))  **else**:  request.session[**'error\_message'**] = **"用户已经停用！"  else**:  request.session[**'error\_message'**] = **"用户名或密码错误！"   return** HttpResponseRedirect(reverse(**'polls:login'**)) |

## 退出登录

|  |
| --- |
| logout(request) |

## 检验密码是否正确

|  |
| --- |
| user.check\_password('raw\_password') |

## session不记录用户登录状态

不记录登录状态就是关闭浏览器后，session失效，在views中，设置request.session.set\_expiry(0) 单位秒，0是关闭浏览器就失效