# **一、**Django简介

## **1.Django的优点**

功能完善，容易上手，开发速度快，安全性高；

有完善的在线文档；

其模型自带数据库ORM(Object-Relational-Mapping，对象关系映射)组件，使得开发者无须学习其他数据库访问技术(如SQLAlchemy等)；

可以使用正则表达式管理路由映射，方便灵活。

## **MVC和MTV模式**

**MVC模式：**

目前的主流web框架，基本上都使用mvc模式开发web应用，其最大的优点是可以降低系统各模块间的耦合度。

Modle：模型，负责处理各个功能的实现，如增加，修改和删除，其中包含模型实体类和业务处理类；

View：视图，负责页面的显示和用户的交互，包含由HTML、CSS、JavaScript组成的各种页面；

Controller：控制器，用于将用户请求转发给相应的模型进行处理，并根据模型的处理结果向用户提供相应的相应。

**MTV模式：**

对传统的MVC模式进行了修改，修改后的模式称为MTV模式。

Model：模型，负责业务对象和数据库的关系映射；

Template：模版，负责页面的显示和用户的交互；

View：视图，负责业务逻辑，并在适当的时候调用Model和Template。

# **二、Django安装**

## 1.windows系统：

安装命令：

**pip install django==版本 -i <https://mirrors.aliyun.com/pypi/simple/>**

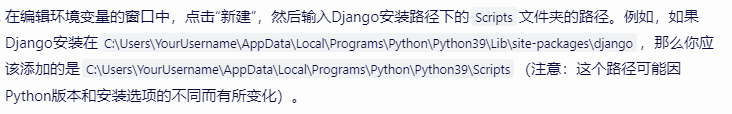
例子：

**pip install django==4.1 -i <https://mirrors.aliyun.com/pypi/simple/>**

查找Django的安装位置：

**pip show django**

配置环境变量：



查看Django版本：

**python -m django --version**

## **2.Linux系统：**

略

# **三、Django知识点**

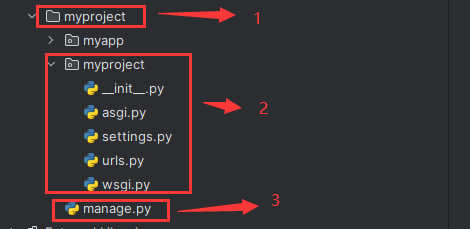
## 1.新建项目

命令：

django-admin startproject 项目名称

例子：

django-admin startproject myproject



**注意：**

manage.py：一个命令行实用程序，可以通过命令行方式与Django项目进行交互。

myproject：外面的为根目录，与Django无关，可以任意命名；里面的为项目目录，这个目录请勿修改。

asgi.py：与ASGI兼容的web服务器，为项目提供服务的入口。

settings：该项目的全局配置文件。

urls：该项目的路由配置文件。

wsgi.py：与WSGI兼容的web服务器，为项目提供服务的入口。

db.sqlit3：sqlit格式的数据库。新版本未显示。

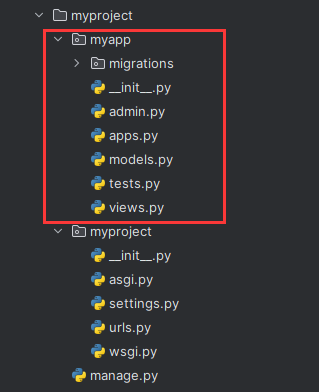
## **2.创建应用**

命令：

python manage.py startapp 应用名

例子：

python manage.py startapp myapp



**注意：**

myapp：应用目录，请勿修改这个目录名称。

admin.py：后台管理使用。

apps.py：应用管理文件。

models.py：模型文件。

tests.py：测试用例。

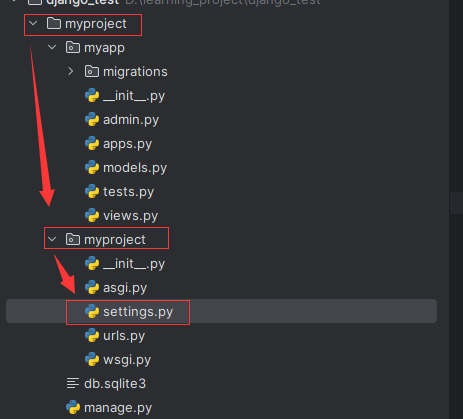
views.py：视图文件。

migrations：该目录包含数据的迁移文件。

创建应用时，需要切换至manage.py同级目录，再输入需要创建应用的命令。

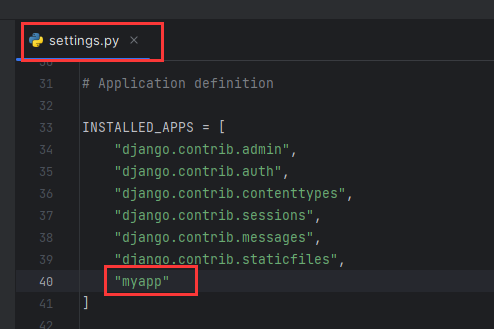
## **3.配置文件**

所有的配置信息都在setting.py文件中完成，该文件的位置如下：



### 3.1配置应用

在settings.py文件中找到INSTALLED\_APPS，将应用名称添加即可。



### **3.2配置数据库**

### **3.3配置静态文件**

### **3.4配置**

### **3.5配置**

### **3.6配置**

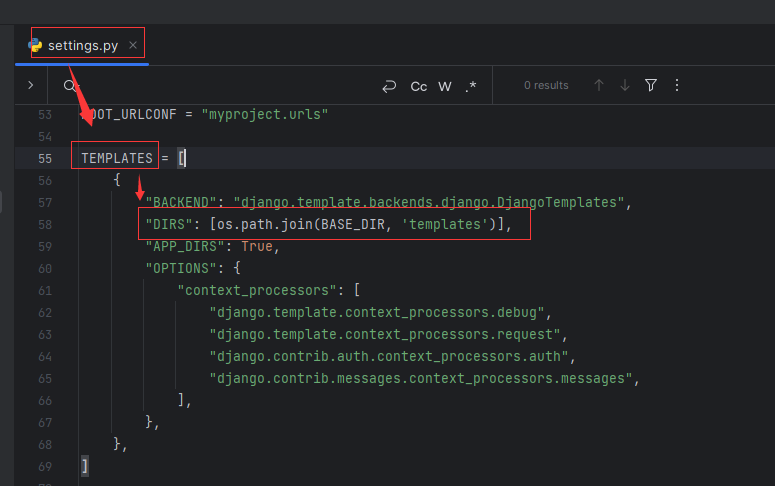
### **3.7配置**

## **4.创建模板**

使用命令行创建Django应用时，默认是没有模板目录，且目录名称为templates，可以更改，但建议保持不变。



模版创建完成后，还不能直接使用，需要在settings配置文件中进行注册。



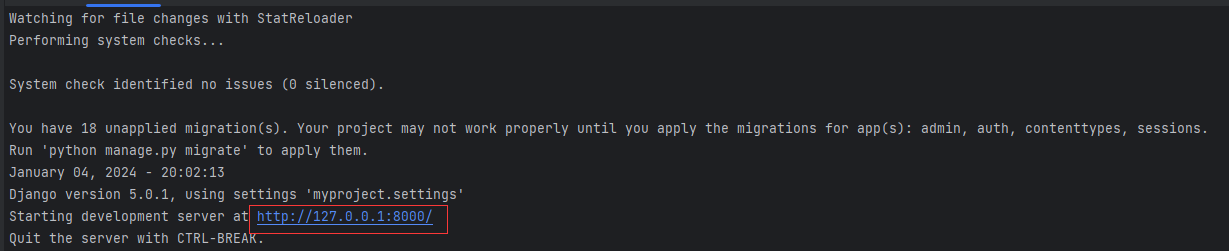
## **5.运行项目**

命令：

python manage.py runserver [port]

例子：

python manage.py runserver

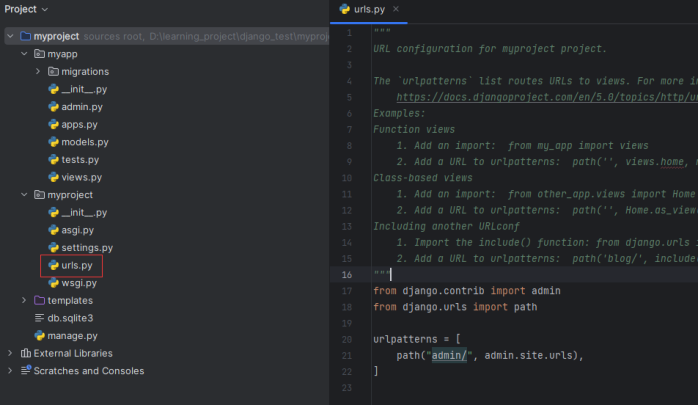


显示上述信息，并在浏览器中输入“http://localhost:8000/”，如果页面有详细信息，表示Django应用运行成功。

## **Django的路由**

### 6.1 简介

当我们创建一个Django项目时，在项目目录下会生成一个urls.py文件，在该文件下定义了Django项目的主要路由信息，该文件是整个项目路由解析的入口。



在新建的应用中也可以使用独立的urls.py，但是需要手动创建；随着业务模块越来越多，路由规则也越来越复杂。可以用“路由包含”来简化项目的复杂度，为每一个应用创建一个urls.py文件，把相关的路由配置都放在每个应用的urls.py文件中；当用户发送请求时，会从根路由开始寻找每个应用的路由信息，生成一个完整的路由列表。当Django从当前请求中获取路由地址后，会先在这个路由列表中进行匹配，然后执行路由相关的视图函数，从而完成整个请求过程。

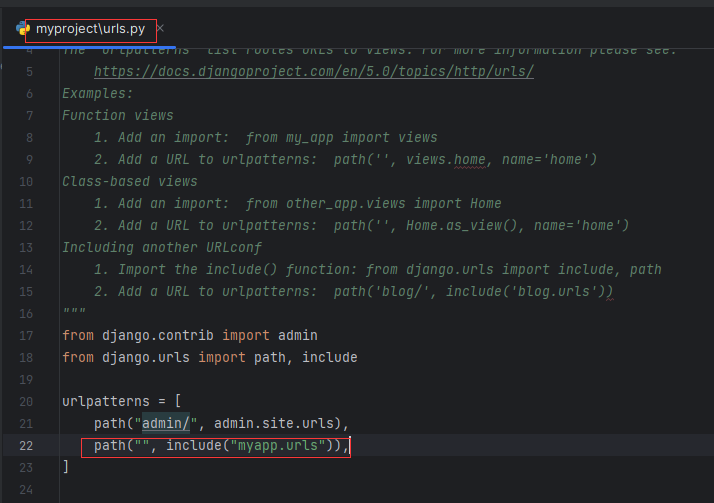
### 6.2 url配置

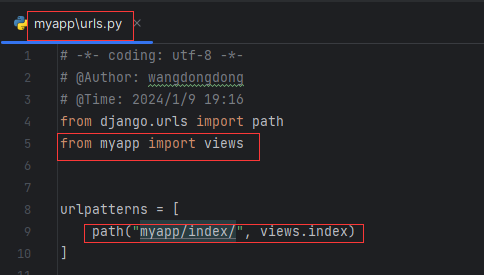
**路由匹配规则：**

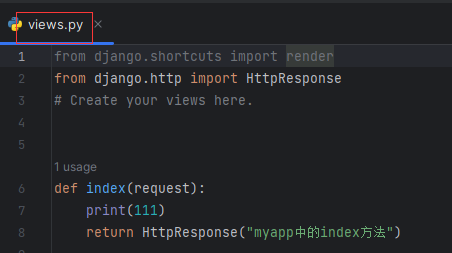
在项目的urls.py文件中，urlpatterns列表会从上往下进行匹配，如果匹配成功，则调用path()函数中第二个参数所指定的视图函数，且不再继续匹配，否则返回404错误。

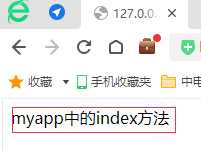
**注意：**

如果在应用中定义了子路由，则在根路由中使用include(“应用名.urls”)来加载子路由，路由信息一般以“/”结尾。









### 6.3 路由参数解析

路由规则：

<参数数据类型：参数名称>

例子：

urlpatterns=[

path(“show/<int:id>/”,views.show)

]

参数类型：

str：字符串，不包含“/”，默认类型；

int：匹配0和正整数；

slug：匹配任何ASCII字符，连接符和下划线；

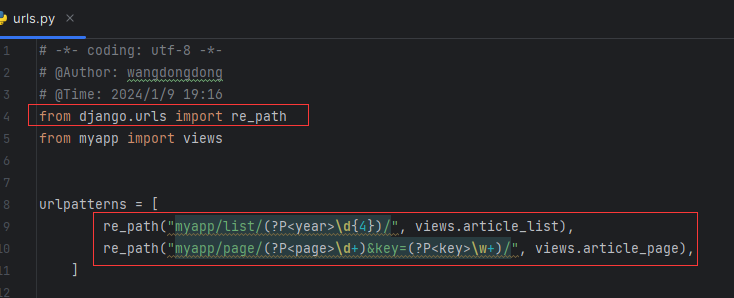
uuid：匹配一个UUID格式的字符串，该对象包含“-”，所有字母必须小写。

### 6.4 re\_path()方法正则匹配复杂路由

相对于path()，功能更强大。格式如下：

(?P<name>pattern)

例子：



“myapp/list/(?P<year>\d{4})/”：

接收以“myapp/list/”开头，后面跟4位整数的路由；

“myapp/page/(?P<page>\d+)&key=(?P<key>\w+)/”：

接收以“myapp/page/”开头，后面跟任意位数整数，且第2个参数可以是字母，数字和下划线的路由；

### 6.5 反向解析路由

在Django的路由配置中，可以给一个路由配置项命名，然后通过别名反向解析得到路由。

例子：

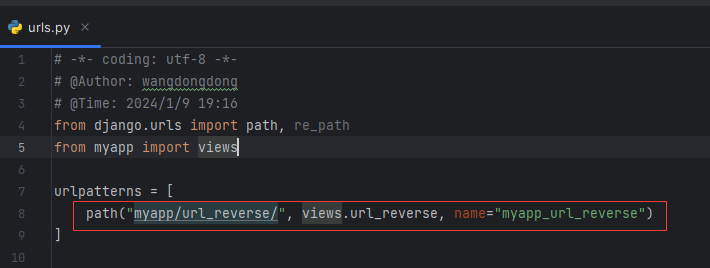
urlpatterns=[

path(“myapp/url\_rever/”,views.url\_rever,name=”myapp\_url\_rever)

]

注意：

name：在name后可以跟任意字符串，为了避免冲突，建议以“应用名\_配置项名”,根据name得到路由配置项中的URL地址，称作“反向解析路由”，这样的好处是，只要name不变，URL地址可以任意改变。







## **7.Django的视图**

### 7.1 简介

Django中的视图是MTV模式中的View层，用于处理客户端的请求并生成响应数据。在视图中使用函数处理请求的方式，被称为视图函数-FBV（Function Base Views），视图函数的代码一般放在应用目录下的views.py文件中。

### 7.2 视图函数的底层原理

视图函数主要使用HttpRequest请求对象和HttpResponse响应对象，当浏览器向服务端请求一个页面时，Django先创建一个HttpRequest请求对象（该对象包含请求元数据），然后加载相应的视图，将这个HttpRequest请求对象作为第一个参数传递给视图，视图函数会返回一个HttpResponse响应对象。

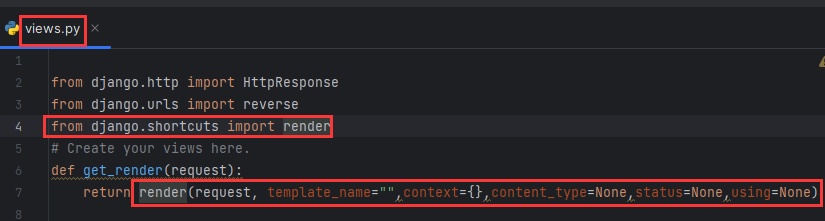
### 7.3 视图处理函数的使用

通过HttpRequest对象和HttpResponse对象，可以处理基本的数据请求并返回相应数据；但是这种方式繁琐，以下是几种简单方法：

1.用render()函数实现页面渲染

该函数根据模板文件和传递给模板文件的字典类型的变量，生成一个HttpResponse对象并返回。

格式：



参数含义：

request：传递给视图函数的所有请求；

template\_name：渲染的模板文件，一般放在templates目录下；

context：数据格式为字典类型，传递到HTML文件；

content\_type：用于生成文档的MIME，默认为text/html；

status：表示响应的状态码，默认为200；

using：设置模板引擎，用于解析模板文件。

2.用redirect()函数实现页面重定向

在项目开发中，经常遇到页面重定向的情况，如网站目录结构调整，网页被移到一个新地址这类情况，如果不做重定向，则通过用户收藏夹中的链接或搜索引擎数据库中的旧地址，会报404页面错误信息。

在Django中，使用重定向函数redirect()实现页面重定向，该函数的参数有3种情况：

(1)通过调用模型的get\_absolute\_url()函数进行重定向；

(2)通过路由反向解析进行重定向；

(3)通过一个绝对的或相对的URL，让浏览器跳转到指定的URL进行重定向。

### 7.4 视图类

在视图里使用类处理方式，被称为视图类-CBV(class base views)，视图类可以更好地处理不同的HTTP请求，在处理视图逻辑时，不用通过“if...else...”来区分请求是GET，还是POST，而是通过在视图类中定义的get()方法和post()方法进行区别。

视图类在调用时，只能是函数的方式，而不能是类的方式，因此，需要将视图类用as\_view()转化为视图函数。

## **8.Django的模版**

### 8.1 简介

Django模板，主要用于页面的展现，实现了业务逻辑和内容显示的分离。Django模板技术可分为两部分：

静态部分：如HMTL，CSS和JavaScript；

动态部分：如Django的模板语音DTL(Django Template Language)。

Django模板是一种带有DTL语言的HTML文件，这个HTML文件可以被Django编译，其中可以传递参数，以实现数据动态化，最终发送给客户端浏览器。Django模板语言包括模板变量，模板标签和模板过滤器。

### 8.2 模板变量

模板变量可以是字符串，列表，字典和类对象。

格式：

{{ 变量名 }}

例子：

我的姓名{{ name }}，我的年龄{{ age }}

注意：

在模板变量的名称前后都有空格，且模板变量的名称可以由字母，数字和下划线组成，但不能有空格等其他字符。

### 8.3 模板标签

模板标签需要用标签限定符{% %}进行包裹，常见的模板标签有{% if %}和{% endif %}，{% load %}，{% block %}和{% endblock %}等。有些标签属于闭合标签，例如，{% if %}的闭合标签是{% endif %}，模板标签的作用：载入代码渲染模板，或对传递过来的参数进行逻辑判断和计算。

举例：

{% if %}

{{ 内容1 }}

{% elif %}

{{ 内容2 }}

{% else %}

{{ 内容3 }}

{% endif %}

### 8.4 模板过滤器

模板过滤器用于对模板变量进行操作，常见的过滤器有字符串大小写转换过滤器，日期过滤器等。

格式：

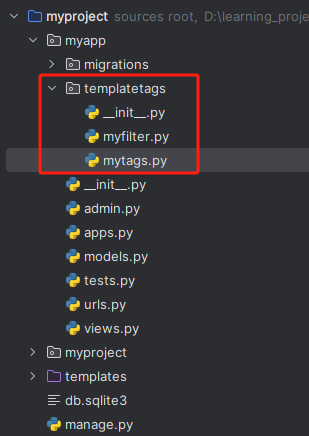
{{ 变量名 | 过滤器：参数 }}

例子：

{{ value | date：Y年m月j日H时i分s秒 }}

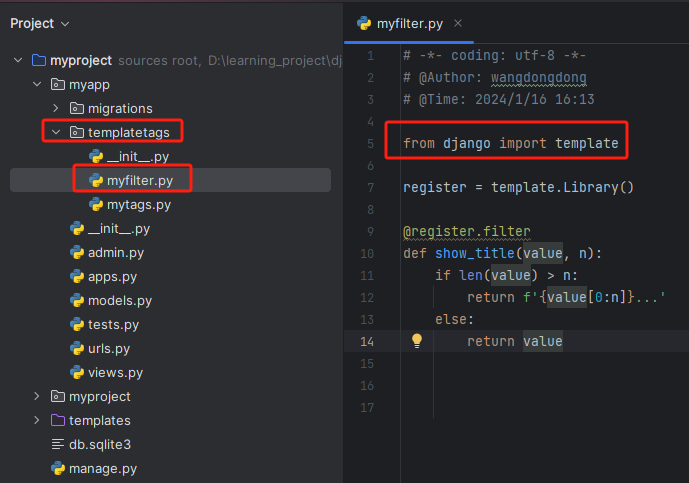
### 8.5 自定义过滤器

如果Django内置的过滤器无法满足需求，可以自定义过滤器。首先，需要在应用下创建一个名为“templatetags”的包，这个包名不可以更改，在该包下创建myfilter.py和\_\_init\_\_.py的文件。



编写自定义过滤器并注册：

自定义的过滤器本质上是一个python函数，要成为一个可用的过滤器，则在过滤器文件中必须包含一个名为“template.Library”的实例变量register。

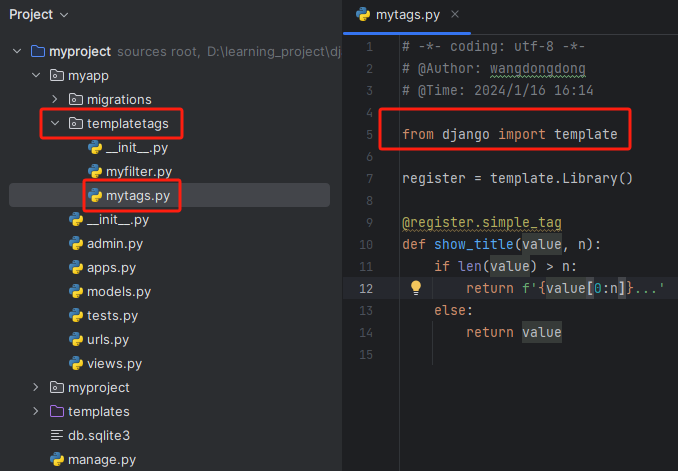


代码注释：@register.filter是一个装饰器，指明show\_title()是一个过滤器，函数的功能：当标题的长度大于要显示的长度n时，自动把多余的标题以省略号进行显示。

### 8.6 自定义标签

自定义标签比过滤功能强大，但是实现起来相对复杂，如果Django内置的标签无法满足需求，可以自定义标签。自定义标签分为：简单标签(Simple Tags)和包含标签(Inclusion Tags)。

简单标签的实现与自定义过滤器类似，也要创建一个名为“template.Library”的实例变量register，然后调用装饰器申明当前函数是一个自定义的简单标签。



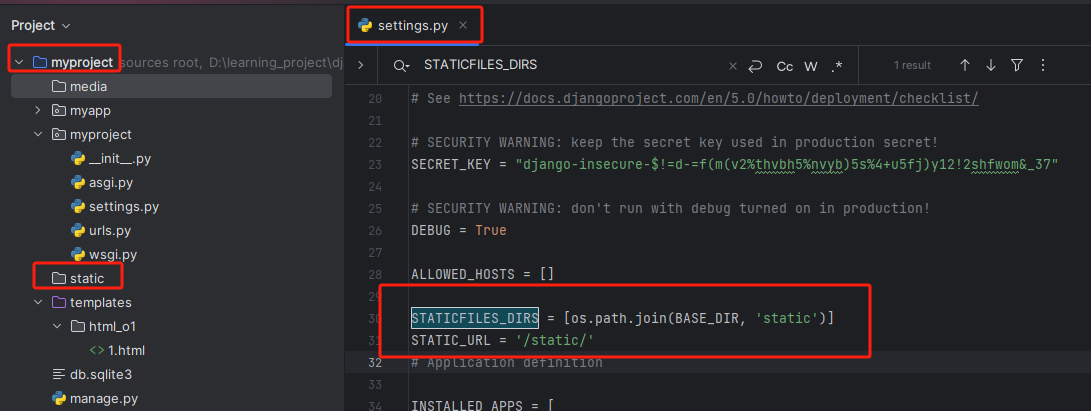
要在模版文件中使用自定义标签，则必须先使用“{{ load 自定义标签mytags }}”进行加载，然后在模板中使用自定义标签函数show\_title()。

包含标签，是指通过渲染一些模板来展示数据的标签。

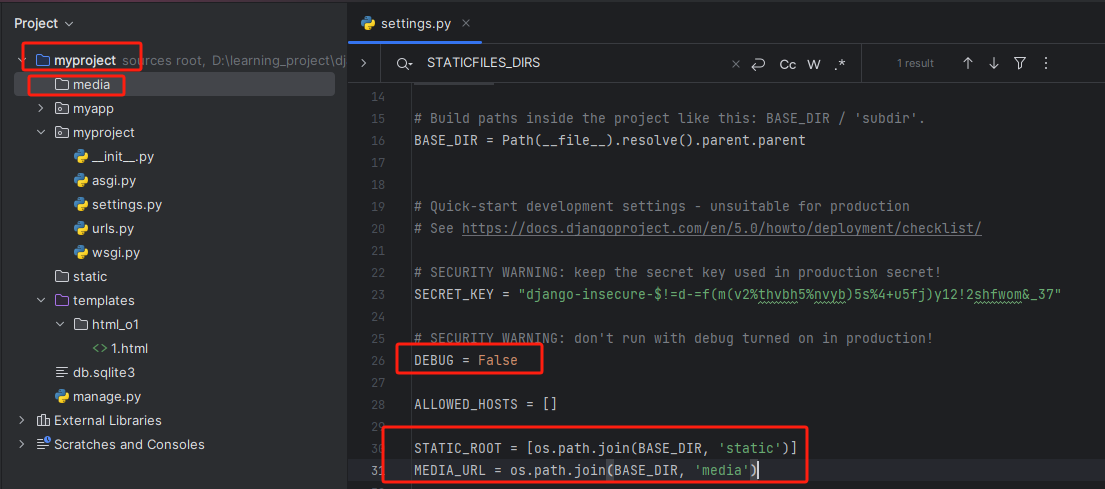
### 8.7 配置模板文件

静态文件包括static和media这两类，static指CSS，JavaScript，Images这类文件；media指媒体文件，比如用户上传的文件等。在Django中，需要在项目的根目录下，手动创建static和media文件夹来存放对应的静态文件。

Debug=True：指调试模式，用于测试环境，需要把“myproject/static”静态目录添加到settings.py的STATICFILES\_DIRS中，这样才能正常访问静态文件。



Debug=False：指生成模式，用于生产环境，同时也要在settings.py修改某些参数。



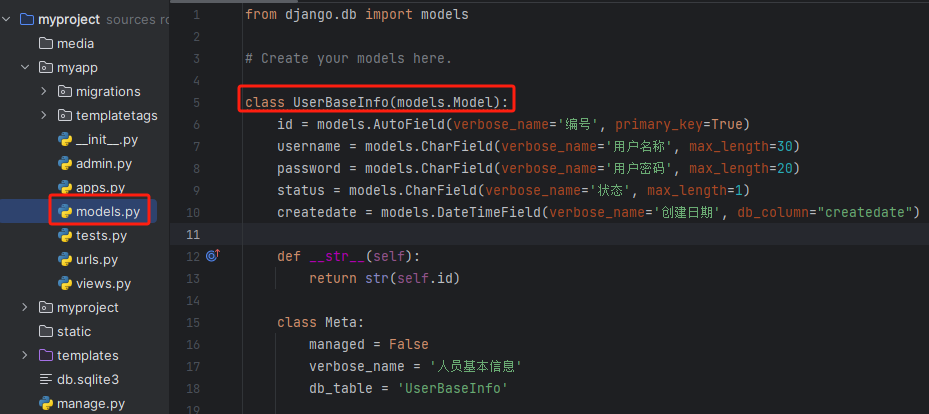
## **9.Django的模型**

在Django框架中，模型用于描述数据库表结构，一个模型对应一个数据库表，模型中的字段对应数据库表中的一个字段。在定义好一个模型后，Django提供访问数据库的一整套API，从而自动在数据库中生成相应的数据表，这样就不需要使用SQL语句创建表，大大提高了开发效率。

### 9.1定义模型

所有Django模型都必须继承Model类，而Model类位于django.db.models中。常用模型字段如下：





常用字段参数：

verbose\_name：设置字段的显示名称；

primary\_key：设置主键；

editable：是否可以编辑，一般用于Admin；

max\_length：设置字段的最大长度；

blank：若为Ture，则该字段允许为空值，在数据库中为空字符串，默认False；

null：若为Ture，则该字段允许为空值，在数据库中为null，默认False；

default：设置字段的默认值；

choices：设置字段的可选值；

db\_column：设置表中的列名称，如果不设置，则将字段名作为列名；

db\_index：数据库中的字段是否可以建立索引；

unique：数据库中字段是否可以建立唯一索引；

error\_messages：自定义错误信息(字典类型)；

validators：自定义错误验证(列表类型)。

**\_\_str\_\_()方法：**

用来设置模型的返回值，其默认返回值为“模型对象”，该方法只允许返回字符类型的字段，如果返回的是其他类型字段，需使用str()进行转换。

**Meta类：**

它是Django模型的一个内部类，用于定义一些Django模型的行为特性。其常用参数如下：

abstract：若为True，则该模型为抽象类，无法在数据库中建表；

db\_table：设置模型对象的数据表名称；

managed：默认True，管理数据表的生命周期，包括迁移等；

ordering：设置模型中需要排序的字段；

verbose\_name：模型类在后台管理中显示的名称，一般为中文。

index\_together：多个字段的联合索引；

unique\_together：多个字段的联合约束。

### 9.2 模型关系

**一对一关系：**

在模型中使用OneToOneField()方法来构建模型的一对一关系。

其参数如下：

to：要进行关联的模型名称；

to\_field：要进行关联的表中的字段名称；

on\_delete：在删除表中的字段时对关联表进行怎样的操作。

其中on\_delete参数选择如下：

CASCADE：在删除基本表信息时一并删除扩展表的信息，即级联删除；

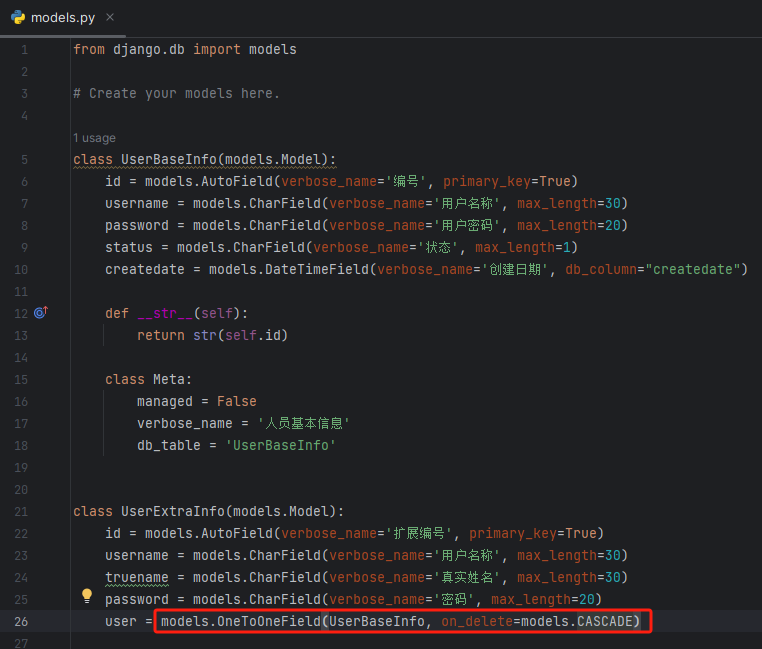
PROTECT：在删除基本表信息时不删除扩展表信息；

SETNULL：只有当字段属性null=True时才将关联的内容置空；

SET\_DEFAULT：设置为默认值；

SET：设置为指定的值；

DO\_NOTHING：删除基本表信息，对扩展表不做任何操作。



**注意：**

哪个数据表需要外键，就把OneToOneField()方法放到该数据表对应的模型中。

**一对多关系：**

在模型中，使用ForeignKey()方法来构建模型的“一对多”关系。

其参数如下：

to：要进行关联的模型名称；

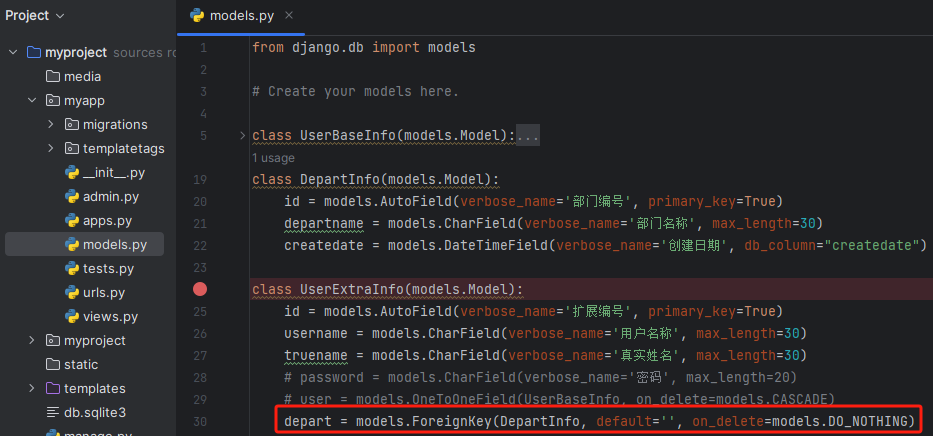
to\_field：要进行关联的表中的字段名称；

on\_delete：在删除表中的字段时对关联表进行怎样的操作；

related\_name：在反向操作时使用的字段名，用于代替“表名\_set”；

related\_query\_name：在反向操作时使用的链接前缀，用于替换“表名”；

db\_constraint：是否在数据库中创建外键约束。



**多对多关系：**

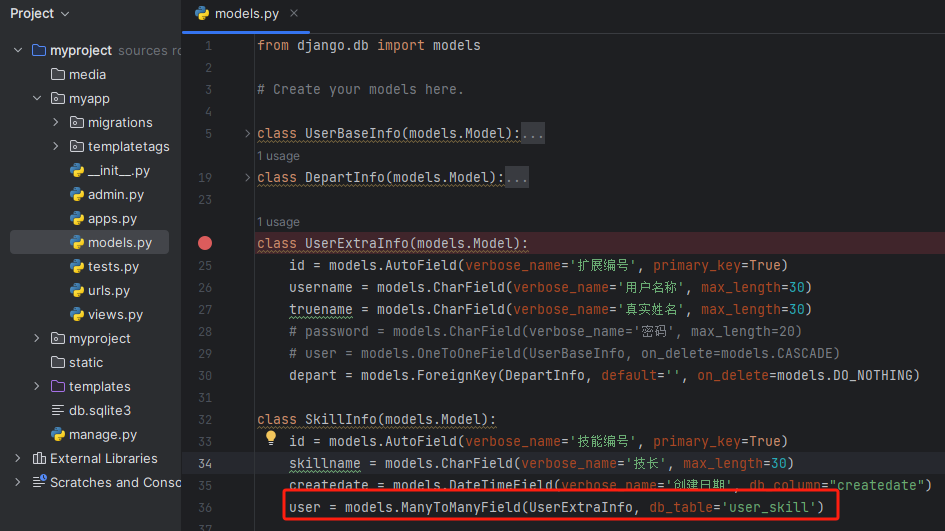
在模型中，使用ManyToManyField()方法来构建模型的“多对多”关系。

其参数如下：

to：要进行关联的模型名称；

db\_constraint：是否在数据库中创建外键约束；

db\_table：默认创建的多对多关系表的表名。

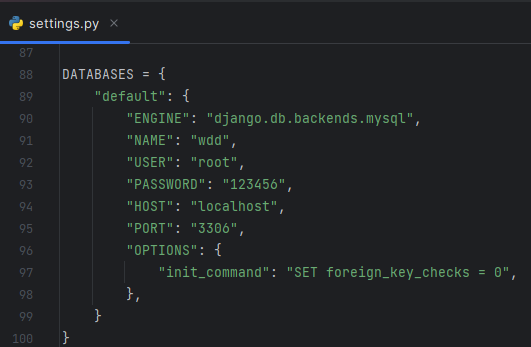


注意：

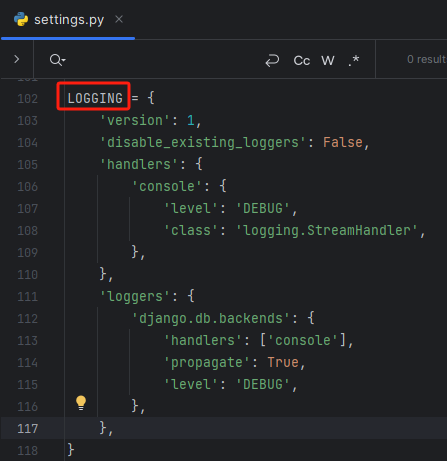
ManyToManyField()方法中没有on\_delete参数，db\_table参数表示模型迁移后生成的多对多关系表的表名，如果不设定，默认表名为“两个模型名称的相加”。

### 9.3 配置数据库信息

在settings.py文件中，找到“DATABASES”节点，然后进行配置。



如果想查看在ORM转换过程中产生SQL语句，则需要再settings.py文件中配置日志信息。



### 9.4 迁移数据

传统的编程方式是先建立数据库，然后在代码中创建对应的实体类，而CodeFirst是在代码中直接创建模型类，框架会根据我们创建的模型类调用数据迁移命令生成数据库和表。

在数据迁移前，首先要生成迁移文件，命令：

python manage.py makemigrations

执行迁移，命令：

python manage.py migrate

以上两个命令执行结束后，在数据库可以查看相关的数据表，且执行迁移命令时，一定要切换到“manage.py”所在目录。

### 9.5 Django模型的查询方法

ORM的作用是在关系型数据库与业务实体对象之间进行映射。

类：数据库中的数据表；

属性：数据库中的字段；

实例：数据库表中的数据行。

QuerySet对象也成查询集，表示从数据库获取的数据对象集合，查询集有一个最重要的特性，惰性执行，它是指在创建查询集后不会访问数据库，只有在调用相关方法时才会访问数据库，这些方法包括遍历迭代，序列化等。

**all()**方法：

该方法用于获取模型的QuerySet对象，即获取所有数据。

格式：

模型类.objects.all()

例子：

from myapp.models import \*

users=UserExtraInfo.objects.all()

print(users[0].truename)

**filter()**方法：

该方法用于实现数据过滤功能，相当于SQL语句中的where子句。该方法获取满足条件的数据，并返回一个QuerySet对象，如果没有获取数据，则返回空的QuerySet对象。

格式：

模型类.objects.filter(字段=value)

例子：

from myapp.models import \*

users=UserExtraInfo.objects.filter(sex=值)

print(users[0].truename)

**get()**方法：

该方法用于查询数据表记录，以模型对象的形式返回符合要求的一条数据。当查询没有记录或者记录超过一条时，会出现错误提示，因此需要做异常捕获处理。

格式：

模型类.objects.get(字段=value)

例子：

from myapp.models import \*

users=UserExtraInfo.objects.get(id=1)

print(users[0].truename)

**exclude()**方法：

该方法用于排除符合条件的数据，返回QuerySet对象。

格式：

模型类.objects.exclude(字段=value)

例子：

from myapp.models import \*

users=UserExtraInfo.objects.exclude(age\_\_lt=32)

print(users[0].truename)

**values()**方法：

该方法用于提取需要的字段，返回一个QuerySet对象，该对象包含的数据类型是由指定的字段和值形成的字典。

格式：

模型类.objects.values()

例子：

from myapp.models import \*

users=UserExtraInfo.objects.values(‘id’,’username’)

print(users[0].truename)

**distinct()**方法：

该方法用于去除重复数据，返回一个QuerySet对象。

格式：

模型类.objects.distinct()

例子：

from myapp.models import \*

users=UserExtraInfo.objects.distinct().values(‘name’)

print(users[0].truename)

**注意：**

filter()方法和get()方法都可以完成数据查询的操作，get()方法在找不到数据时会触发异常，filter()方法则不会有任何提示。

### 9.6 数据新增

**save()**方法：新增或更新

from myapp.models import \*

depart=DepartInfo()

depart.departname=’技术部’

depart.departcode=’部门编号’

depart.save()

**create()**方法：新增

from myapp.models import \*

d=dict(username=’张三’,password=’12345’,status=1,)

user=UserExtraInfo.objects.create(\*\*d)

**update()**方法：更新

from myapp.models import \*

user=UserExtraInfo.objects.update(status=1)

**delete()**方法：删除

from myapp.models import \*

user=UserExtraInfo.objects.all().delete()

### 9.7 操作关联表

“**一对一**”关联表的操作：

用户基本表和用户扩展表是“一对一”的关系，通过关联属性user来实现两者之间的关系。

例子1：

添加用户基本表：

d=dict(username=’李四’,password=’12345’,status=1)

one\_user=UserBaseInfo.objects.create(d)

添加用户扩展表：

d=dict(username=’李四’,truename=’李小四’**user=one\_user**)

extrainfo=UserExtraInfo.objects.create(\*\*d)

例子2：

从用户基本表访问用户扩展表：

user=UserBaseInfo.objects.get(id=1)

user.**userextrainfo**.username #扩展表名称需要小写

从用户扩展表访问用户基本表：

result=UserExtraInfo.objects.get(id=1)

result.**user**.username #user是外键

“**一对多**”关联表的操作：

用户基本表和用户卡信息表是“一对多”的关系，通过关联属性user来实现两者之间的关系。

例子1：

获取用户基本表：

user=UserBaseInfo.objects.get(id=1)

添加用户卡信息表：

card=CardInfo(cardno=’111111’,bank=”工商银行”,**user=user**)

card.save()

例子2：

从用户基本表访问用户卡信息表：

user=UserBaseInfo.objects.get(id=1)

user.**cardinfo\_set.**all() #通过“模型名小写\_set.all()”查询

或者

userinfo=UserBaseInfo.objects.get(id=1)

userinfo.**usercard.**all() #通过配置related\_name查询

从用户卡信息表访问用户基本表：

card=cardinfo.objects.get(id=1)

card.**user**.username #通过外键user

“**多对多**”关联表的操作：

一个用户可以拥有多项技能，一个技能被多个用户拥有。

例子1：

获取用户基本表：

user=UserBaseInfo.objects.all()

获取id=1的技能：

skill=SkillInfo.objects.get(id=1)

给所有的用户增加id=1的技能：

result=skill.user.add(\*user)

例子2：

从用户基本表访问技能表：

user=UserBaseInfo.objects.get(id=1)

user.**skillinfo**.username #扩展表名称需要小写

从技能表访问用户基本表：

result=SkillInfo.objects.get(id=1)

result.**user**.username #user是外键

例子3：

获取用户基本表：

user=UserBaseInfo.objects.all()

获取id=1的技能：

result=SkillInfo.objects.get(id=1)

修改全部用户中技能id为1的数据：

result=skill.user.set(user)

修改指定用户中技能为1的数据：

user=[1,2]

result=skill.user.set(user)

例子4：

获取用户基本表：

user=UserBaseInfo.objects.get(id=1)

获取id=1的技能：

Skill=SkillInfo.objects.get(id=1)

删除指定的用户：

result=skill.user.remove(user)

result=skill.user.remove(2)

result=skill.user.remove(user)

**关键点：**

**1.select\_related()方法：**

在访问某个模型数据时，可以将关联的模型数据提取出来，这样可以减少数据库查询的次数。比如，访问用户卡信息数据，可以使用select\_related()方法将用户信息提取出来，再次使用cardinfo.user时就不需要访问数据库了。

**使用该方法前：**

cards=CardInfo.objects.all()

for card in cards:

print(card.user)

**使用该方法后：**

cards=CardInfo.objects.select\_related(“user”)

for card in cards:

print(card.user)

**注意：**

该方法使用了SQL语句的多表关联，只执行了一次数据库查询，大大提升了效率，说明该方法只能用在“一对多”关系且设置了外键的模型中。

**2.prefetch\_related()方法：**

该方法与select\_related()方法类似，用于解决“多对一”和“多对多”关系的查询问题，在访问多个表中的数据时，使用它可以减少查询的次数。

**例子：**

skills=SkillInfo.objects.prefetch\_related(“user”)

for skill in skills:

print(skill.username)

users=skill.user.all()

for user in users:

print(user.username)

### 9.8 F()函数和Q()函数

1.F()函数

在Django的ORM语句中，F()函数用于实现数据表中字段的各种运算操作。

使用前：

users=UserExtraInfo.objects.all()

for user in users:

user.salary+=10

user.save()

使用后：

from django.db.models import F

users=UserExtraInfo.objects.all()

for user in users:

user.salary=F(“salary”) + 10

user.save()

注意：

使用F()函数可以直接在数据库中进行操作，可以减少某些操作所需的数据库查询次数。在使用F()函数对某个字段进行更新后，需要使用refresh\_from\_db()方法才能获取最新字段的值信息。

例子：

from django.db.models import Q

users=UserExtraInfo.objects.all()

for user in users:

User.refresh\_from\_db()

Print(user.salary)

2.Q()函数

在Django的ORM语句中，Q()函数用于对象进行多条件查询，支持&(and)，|(or)，~(not)操作符。

例子：

users=UserExtraInfo.objects.filter(Q(age\_\_gt=3)&Q(salary\_\_gt=50))

3.Raw()方法

在实际项目中，可能会遇到一些复杂的SQL语句，此时难以使用ORM中提供的API来完成，Django提供了直接执行SQL语句的方法。

例子：

users=UserExtraInfo.objects.raw(“select \* from userbaseinfo”)

for user in users:

print(user)

## **10.基于Django表单**

基于传统的表单开发项目，会显著增加HTML页面的代码量，从而增加开发难度和调试时间。Django的Form表单可以自动生成HTML组件标签，增加体验功能，极大地提高了编写代码的效率。而Django的模型表单更是在Form表单的基础上，进一步精简了代码，提高了效率。

HTML表单是一个包含表单元素的区域，表单的主要功能是收集用户信息，以完成人机交互，表单由一个个HTML标签组成，常见的HTML标签有文本框，单选框，复选框，下拉列表框等等。



### 10.1 用令牌CSRF保证表单的安全

HTML表单想要在Django中正常显示，还需要添加CSRF令牌。CSRF令牌是Django为了防止网站跨站请求伪造而默认开启的一项保护机制，在Django中，通过“{% csrf\_token %}”标签来为表单添加令牌，而“{% csrf\_token %}”标签放在网页<form></form>标签之间。

例子：

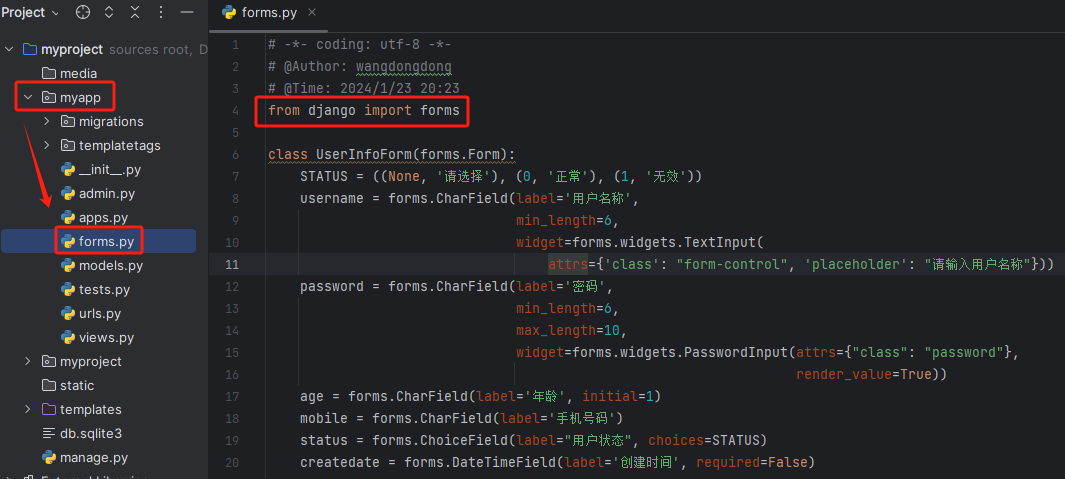
略。

### 10.2 Django的Form表单

Django框架提供了Form表单，用于生成页面可用的HTML标签。用户在表单中输入数据提交表单时，Django框架会自动进行表单数据验证，并将数据绑定到表单对象。

#### 10.2.1 认识表单

表单类被定义在每个应用目录下的forms.py文件中，该文件默认不存在，需要手动添加，且表单类必须继承自Form类，该Form类位于包django.forms中。



常见的表单字段：



常见字段参数：



表单元素的风格显示：

参数widget的作用是按照widget指定的类型在网页中生成对应的标签样式。



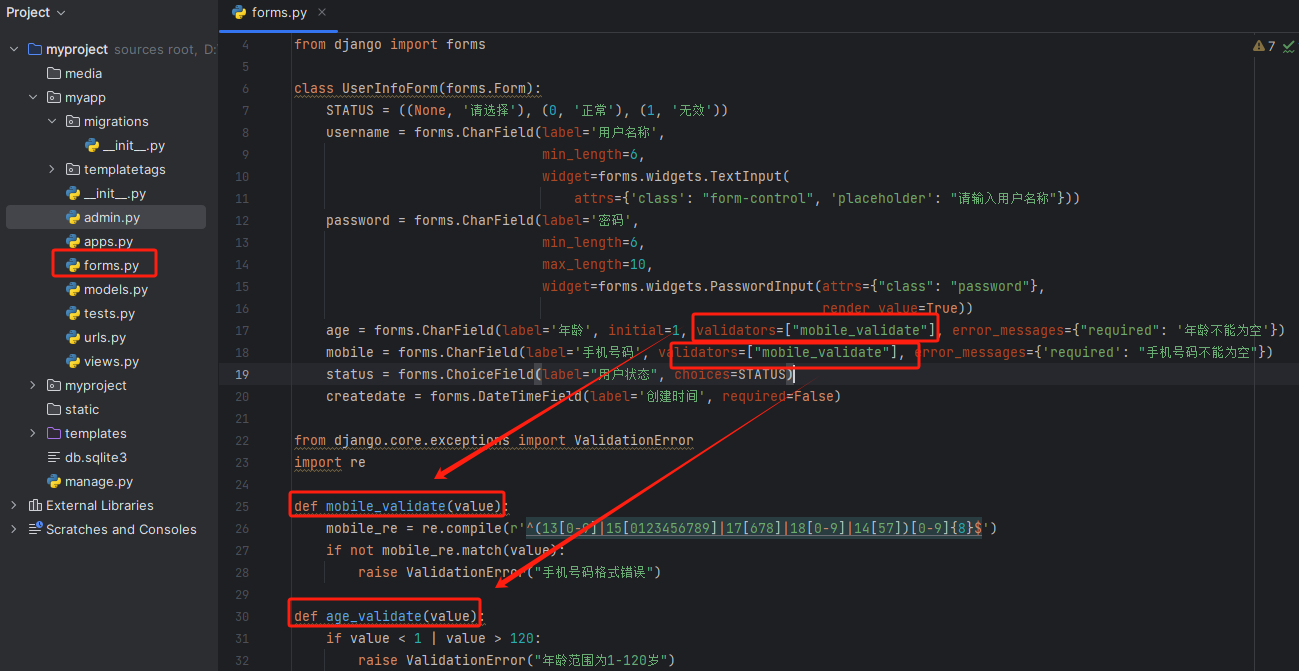
**注意**：同一种表单字段可以根据不同的widget被渲染为不同的展示效果。

#### **10.2.2 表单数据的校验**

在提交表单数据时，会涉及表单的数据校验。

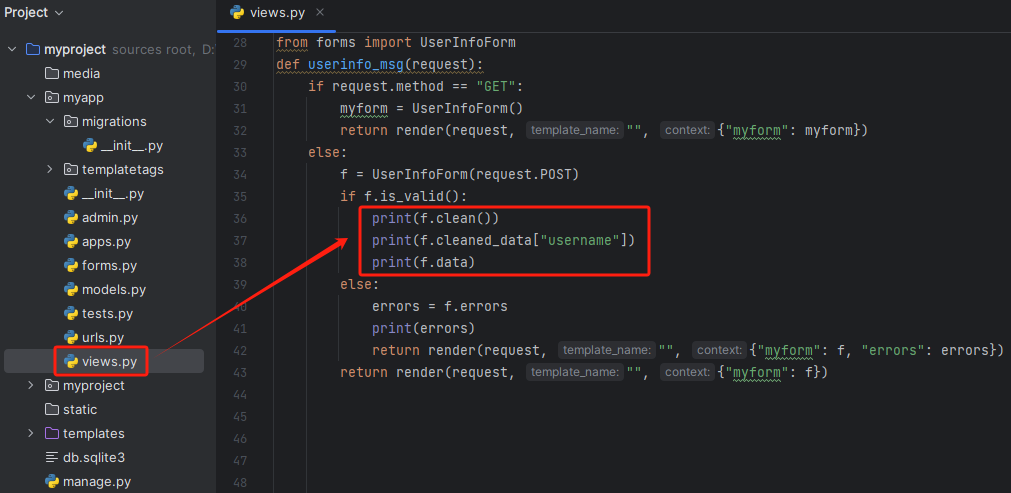


自定义验证规则：



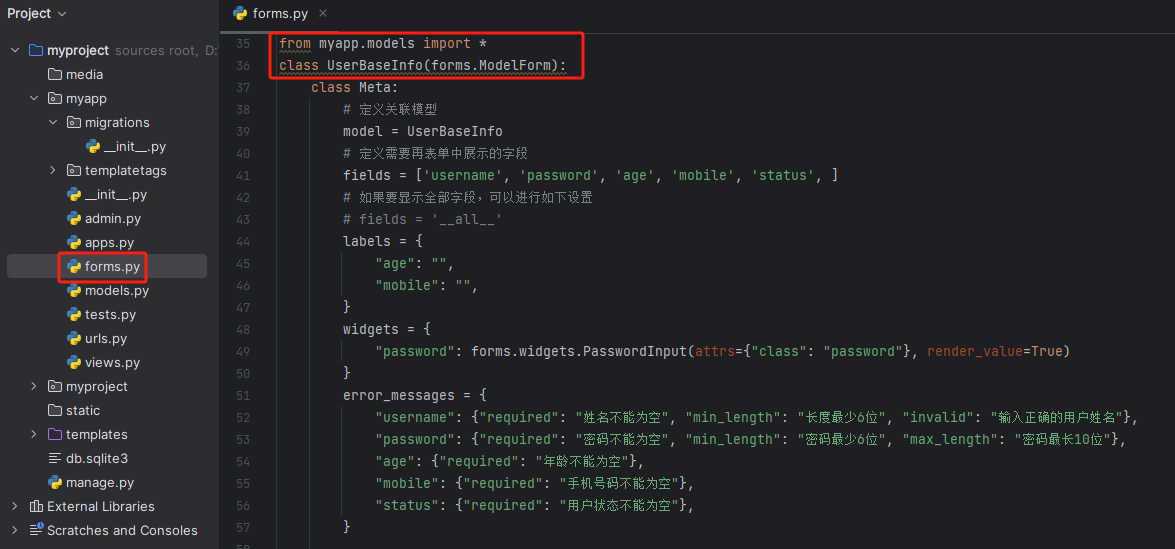
#### **10.2.3 表单数据的获取**

在表单验证成功后，可以通过forms.clean()方法或者forms.cleaned\_data属性获取表单提交的数据；此外还可以通过forms.data属性获取表单原始的数据。

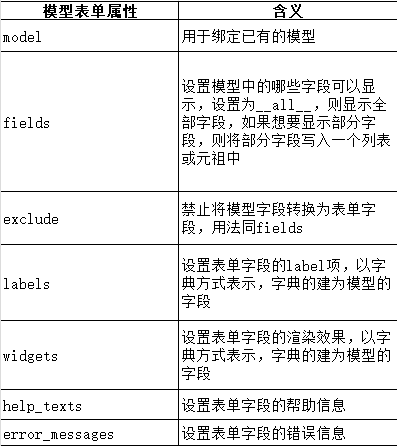


### 10.3 Django的模型表单

Django提供了模型表单ModelForm，它可以和模型直接关联，省略了在Form表单中定义表单字段的过程；简单来说，Django中的ModelForm就是利用数据模型Model来简化表单开发的一种高级封装技术。



常见的模型表单属性：



自定义校验函数：

在Django的ModelForm类中提供了自定义校验函数，以进行字段的校验，以下是2种校验方式：

对某个字段进行精确校验，校验函数的语法格式：clean\_字段名()，该字段中输入的数据必须经过验证函数的校验。

重写clean()函数来实现校验，使用该函数相当于对每个字段都进行处理。

**注意**：

Django的ModelForm类提供了save()方法，用于将表单绑定的数据直接保存到数据库中。

例子：

f = UserBaseInfoModelForm(request.POST) #模型类

New\_userinfo = f.save()

### 10.4 使用AJAX提交表单

AJAX即Asynchronous Javascript And XML(异步Javascript 和 XML)，是一个使用现有技术集合的新技术，包括HTML或XHTML,CSS,JavaScript,DOM,XML,XSLT,以及最重要的XMLHttpRequest。简单来说，AJAX就是在不重载整个网页的情况下，通过后台加载数据，并在网页上进行局部刷新和显示。

**11.用户认证**

## **12.Django的后台管理**

## **13.接口的设计与实现**

## **14.分层的自动化测试**

# **四、Django的开发流程**

# **五、Django的注意事项**

# **六、Django的常见问题**

# **七、学习网址**