Django学习笔记（一）

序章

Django框架的模型结构：MVC结构，分别是模型（Models），视图（Views），控制器（Controller）

MVC 以一种插件式的、松耦合的方式连接在一起。

- 模型（M）- 编写程序应有的功能，负责业务对象与数据库的映射(ORM)。
- 视图（V）- 图形界面，负责与用户的交互(页面)。
- 控制器（C）- 负责转发请求，对请求进行处理。

在Django中，采用MVC的一种变体MTV，即模型（Models），模板（Templates），视图（Views）

- M 表示模型（Model）：编写程序应有的功能，负责业务对象与数据库的映射(ORM)。
- T 表示模板 (Template)：负责如何把页面(html)展示给用户。
- V 表示视图（View）：负责业务逻辑，并在适当时候调用 Model和 Template。

除了以上三层之外，还需要一个 URL 分发器，它的作用是将一个个 URL 的页面请求分发给不同的 View 处理，View 再调用相应的 Model 和 Template

（以上部分参考自菜鸟教程： Django 简介 | 菜鸟教程 (runoob.com) ）

一、创建一个项目

1. 在python中安装django库： `pip install django`

默认安装的是最新的发行版，现已更新至4.0+，和网络中的2.0+相比，3.0时增添了asgi作为wsgi的异步处理版本，支持asyncio等异步的库

2. 创建项目文件：在控制台中输入 `django-admin startproject <项目名>`
则创建对应项目，并且项目结构如下（假定项目名称为 `mysite`）：

```
mysite: # 项目文件夹
  - mysite: # 配置文件夹
      - __init__.py # 初始化文件，告诉 Python 该目录是一个 Python 包
      - asgi.py # 异步服务器网关接口
      - sttings.py # 配置文件
      - urls.py # 路由文件，一份由 Django 驱动的网站"目录"
      - wsgi.py # 同步服务器网关接口
  - manage.py # 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互
```

3. 运行项目：切换到 `manage.py` 文件对应的文件夹目录后，运行 `python manage.py runserver`，此时服务被默认部署在 `127.0.0.1:8000`，如需更改，则在前面的命令末尾加上运行的地址（ip和端口）
此时会发现，运行报错：`DJANGO.CORE.EXCEPTIONS.IMPROPERLYCONFIGURED: SQLITE 3.9.0 OR LATER IS REQUIRED (FOUND 3.x.x).`，这说明默认的sqlite数据库的版本不支持当前的django框架版本，此时要想成功运行有以下三个选择：
    - 降低django版本，即在安装django的时候指定版本号
    - 升级sqlite（win系统中较为复杂）
    - 更换默认数据库

这里采用是第三种方法，更换默认数据库为mysql，需要安装python控制mysql的库并在配置文件中进行更改：

- python 3.x中控制mysql的库为 `pymysql`，安装这个库
- 更改如下配置文件：
    - 在 `__init__.py中` 加入：
    ```
    import pymysql
    pymysql.install_as_MySQLdb()
    ```
    因为在python2.x中，控制mysql的库是mysqldb，但是3.x中不再使用这个库了，需要适配
    - 在 `settings.py` 中，更改DATABASES项为：

```
'default': {
    'ENGINE': 'django.db.backends.mysql',  # 固定配置
    'HOST': '127.0.0.1',  # mysql地址
    'PORT': '3306',  # 端口号
    'NAME': 'django',  # 库名（组名）
    'USER': 'root',  # 你的用户名，一般默认是root
    'PASSWORD': '123456',  # 你root账户的密码
    'OPTIONS': {'init_command': "SET
sql_mode='STRICT_TRANS_TABLES'", },
    }
```

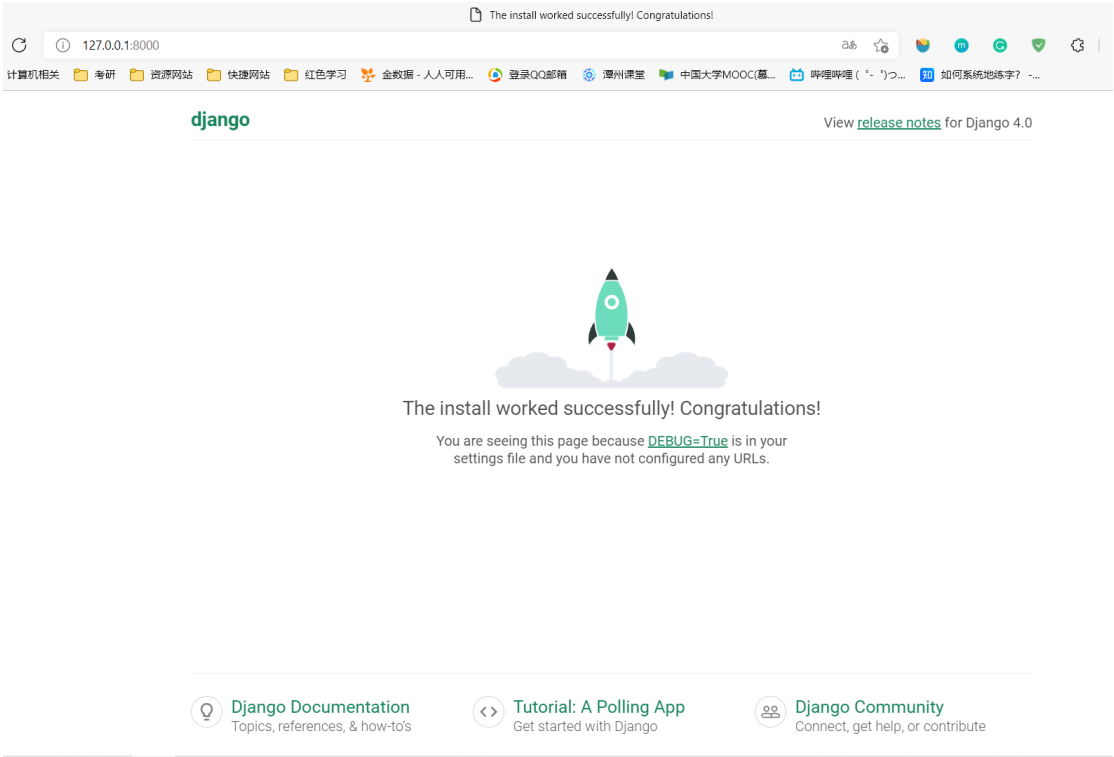
原来的默认配置是：

```
'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': BASE_DIR / 'db.sqlite3',
     }
```

可以注释掉

- 执行以下命令进行数据迁移：
  - `python manage.py makemigrations`
  - `python manage.py migrate`

4. 重新运行项目，在浏览器输入对应部署网址后看到初始界面表示项目创建成功：



二、路由管理初步&请求响应——HelloWorld

在项目中输出一个“HelloWorld”，需要通过配置相应的路由和对应的方法来完成

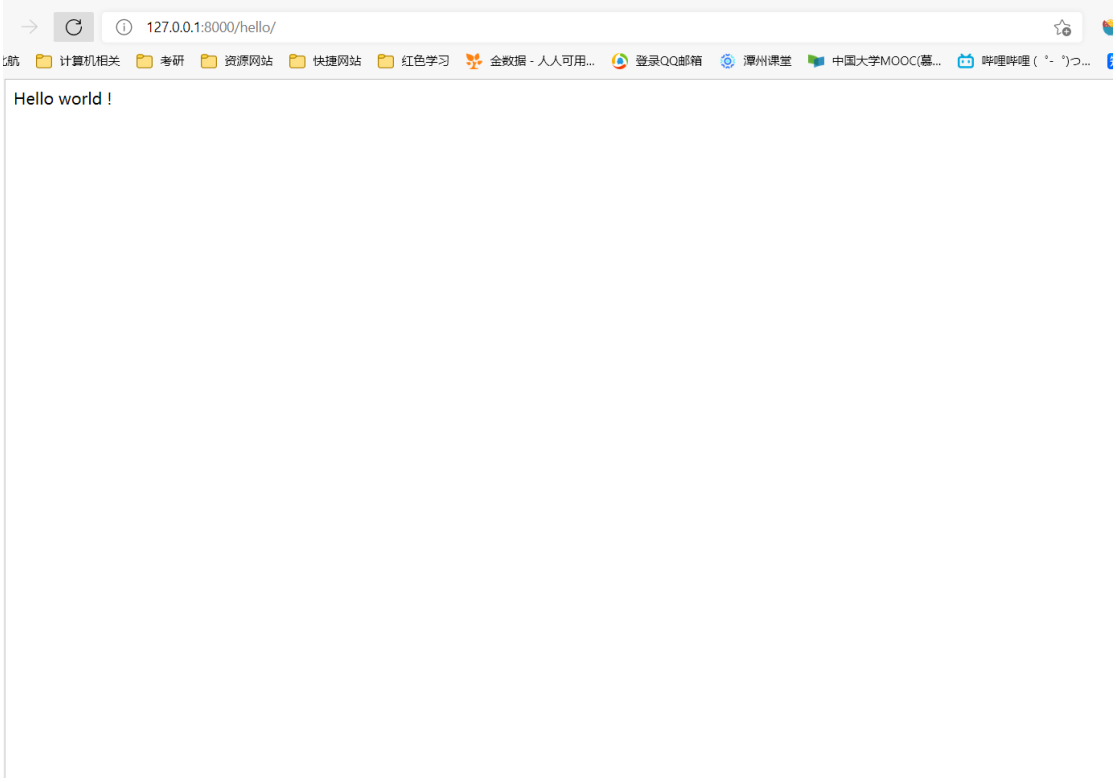
1. 在项目配置文件的目录中创建一个新文件 views.py （固定名称），并进行如下配置：

```python
from django.http import HttpResponse

def hello(request):  # 接受一个请求
    return HttpResponse("Hello world ! ")  # 返回一个响应
```

2. 在 urls.py 文件中进行配置，导入刚才创建的views文件并对urlpatterns列表进行添加：

```python
from . import views  # 添加这条导入信息

'''
这是原来的urlpatterns
urlpatterns = [
    path('admin/', admin.site.urls),
]
'''
# 新的urlpatterns
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', views.hello),
]
```

此时开启服务器，并在浏览器网址栏输入 127.0.0.1:8000/hello/ ，可以看到如下结果：



3. 关于路由配置：
   - `path(route, view, kwargs=None, name=None)` 方法：四个参数
     - route: 字符串，表示 URL 规则，与之匹配的 URL 会执行对应的第二个参数 view。
     - view: 用于执行与正则表达式匹配的 URL 请求。
     - kwargs: 视图使用的字典类型的参数。
     - name: 用来反向获取 URL。
   - `re_path(route, view, kwargs=None, name=None)` 方法，四个参数同上，用来匹配使用正则表达式的URL

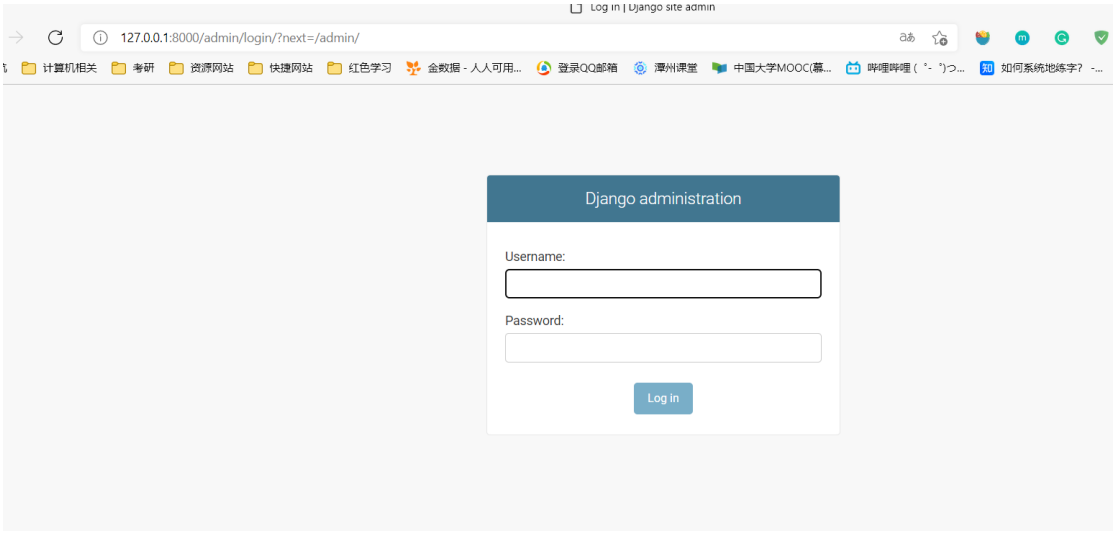
三、后台管理初步

1. 通过 `manage.py` 创建超级管理员用户： `python manage.py createsuperuser` 然后依次输入用户名和密码即可，中间的邮箱可以略过

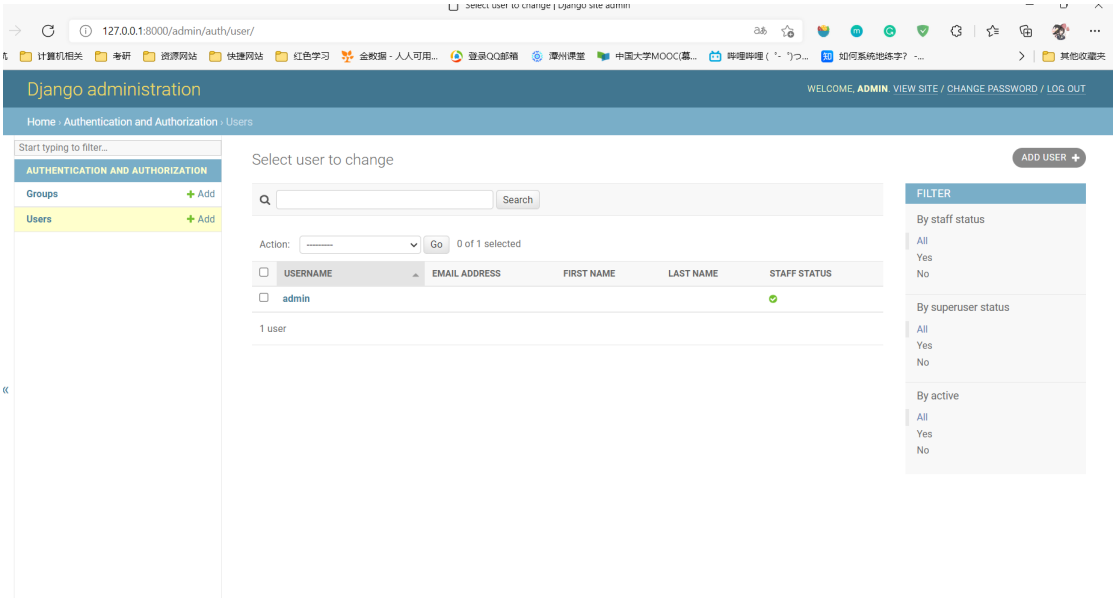
```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19043.1415]
(c) Microsoft Corporation。保留所有权利。

C:\Users\30266\Desktop\DjangoTest\mysite>python manage.py createsuperuser
Username (leave blank to use '30266'): admin
Email address:
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

   这里选择用户名为 `admin`，同时设定了密码也是admin，系统提示密码和用户名过于相似而且太短（少于8位），而且密码太简单，但是下面选择 `y` 表示仍然这样设置，则超级用户设置成功。

2. 运行服务，在浏览器输入 `127.0.0.1:8000/admin/` 即可打开后台管理的登录页面：



输入用户名和密码，进入后台：



发现这里有一个用户，就是刚才创建的 admin

四、创建应用（APP）

APP是Django框架中功能模块实现的部分，我们根据需要的功能来创建和设置不同的APP来进行使用。APP中有模型（Models）和视图（Views），是需要我们重点关注的部分。

1. 通过 manage.py 创建 APP： python manage.py startapp <名称>

这里创建了一个名为 article 的APP用来发布文章

此时，在我们的项目文件夹中出现了新的文件夹，就是刚创建的app的文件夹，因此整个项目的文件树变成了以下情况：

```
mysite: # 项目文件夹
  - mysite: # 配置文件夹
      - __init__.py # 初始化文件，告诉 Python 该目录是一个 Python 包
      - asgi.py # 异步服务器网关接口
      - sttings.py # 配置文件
      - urls.py # 路由文件，一份由 Django 驱动的网站"目录"
      - views.py # 在步骤二中创建的文件，表示视图，即用户看到的界面，也是URL对应的方法
      - wsgi.py # 同步服务器网关接口
  - article: # 刚刚创建的APP的文件夹
      - __init__.py
      - admin.py # APP的后台管理文件，如需通过后台向数据库中添加数据，可以进行操作
      - apps.py # APP的启动类
      - migrations # 数据库表、字段等变更的记录
      - models.py # 持久层，对数据库进行操作
      - tests.py # 单元测试
      - views.py # 视图，与配置文件夹中的作用一样
  - manage.py # 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互
```

2. 在全局设置中注册APP：

找到配置文件夹，修改其中的`settings.py`中`INSTALLED_APPS`字段，并在列表中添加上我们创建的APP。这里可以打开`article`文件夹下的`apps.py`文件，看到里面有一个类叫做`ArticleConfig`，里面保存了我们这个APP的配置，因此，我们在`INSTALLED_APPS`列表中添加`'article.apps.ArticleConfig',`就可以了（逗号也要添加进去）。

3. 创建一个简单的文章模型，并通过后台添加数据：

1. 首先在article的`models.py`中写入以下内容并保存：

```python
class Article(models.Model):
    title = models.CharField(max_length=30) # 设置一个属性为标题，对应数据库中的字符型，最大长度为30
    content = models.TextField() # 设置内容属性，对应数据库中的文本型
```

2. 进行数据库更新：

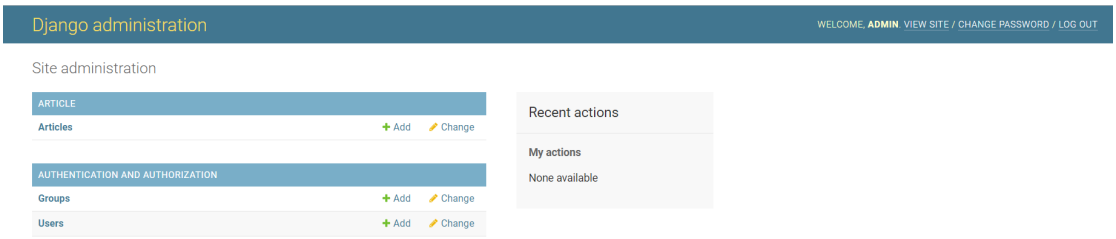
- python manage.py makemigrations
- python manage.py migrate

3. 进行后台注册：

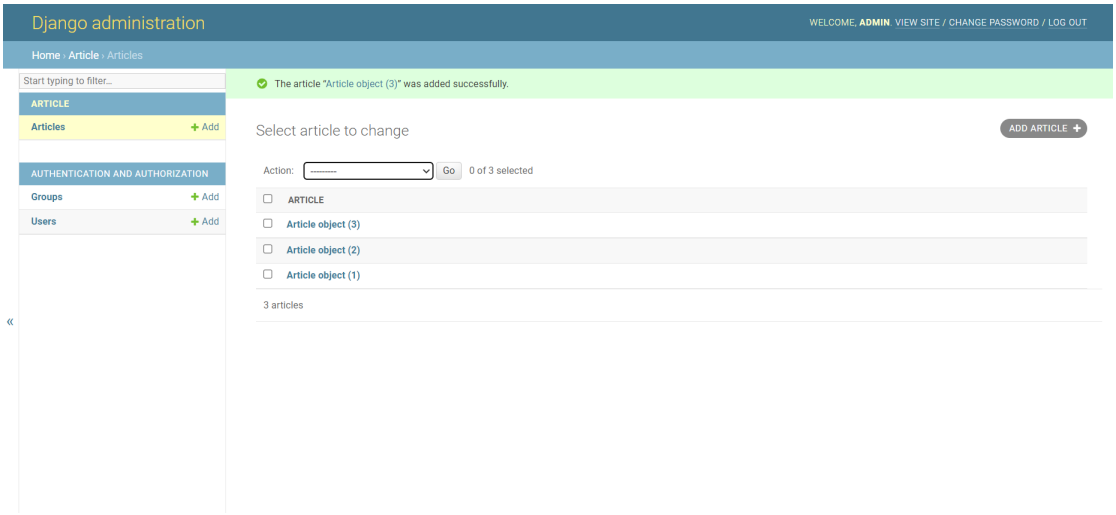
在article的 admin.py 中添加以下内容：

```
# 前面有文件自带的一些import信息，不用修改，直接添加以下内容即可
from .models import Article

admin.site.register(Article)
```

4. 打开后台，发现已经有了Article：



添加3个数据：



4. （可选）将后台改为中文界面：

将配置文件夹的 settings.py 中的 LANGUAGE_CODE 改为 zh-hans

五、模板的创建和使用

模板（Template）是为了方便同质化网页的部署而使用的，比如一般的个人博客，其每篇文章的页面都可能用的是一个html模板，只是标题和显示的内容不同，这里的html文件就是模板。我们在Django中可以自己制作或放置模板，来进行网页的批量处理。

1. 模板文件夹的位置和创建模板：

我们首先可以查看配置文件夹中的 `setings.py` 文件，找到 `TEMPLATES`，默认情况下，其中 `DIRS` 为空列表，而 `APP_DIRS` 为 `True` 表示在寻找模板文件时，会去APP文件夹内的模板文件夹找。

当 `DIRS` 为空时，项目文件夹中即使有 `templates` 文件夹，也不会优先去这里找模板文件，会按照 `settings.py` 中的注册顺序，从每个APP的 `templates` 文件夹找，只要找到对应名称的文件就会返回。

此时，我们可以先在article文件夹下创建 `templates`，并创建一个模板 `article_detail.html`

内容如下：

```
<html>
  <head>
  </head>
  <body>
    <!-- 这里用双大括号标出变量，因为传来的article_obj是一个对象，
    有title和content两个属性，可以直接调用 -->
    <h2>{{article_obj.title}}</h2>
    <p>{{article_obj.content}}</p>
  </body>
</html>
```

2. 视图配置和路由管理

1. 视图配置：

   在article文件夹的 `views.py` 中添加一个方法和对应的依赖：

```python
from django.shortcuts import get_object_or_404  # 这个函数如果能查到就获取，差不到就抛出404异常
from .models import Article

def article_detail(request, article_id):
    article = get_object_or_404(Article, id=article_id)  # 传入模型，数据库按传入的id进行查找
    content = {}  # 一个内容字典，字典的键对应的就是html模板中的变量名
    content["article_obj"] = article
    return render(request, 'article_detail.html', content)  # 传入request，模板，内容字典
```

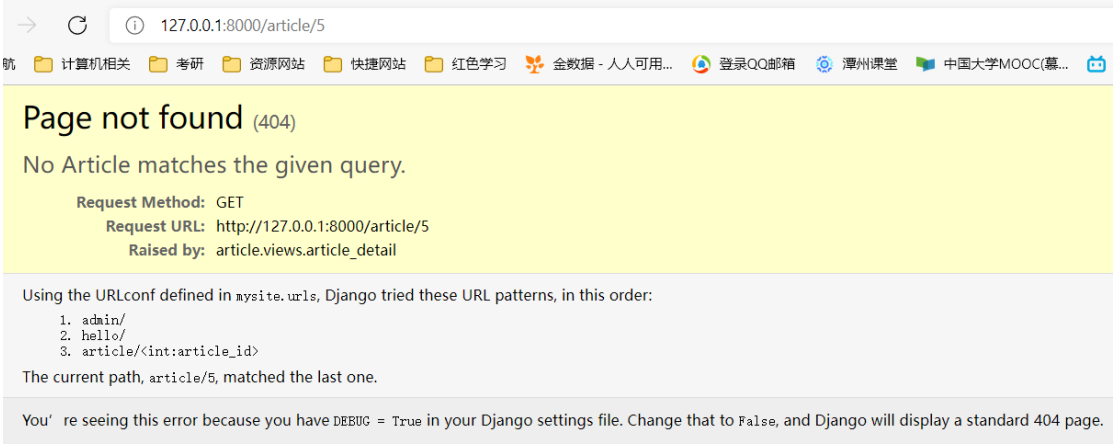
2. 路由管理：

在配置文件夹的 urls.py 中添加一条路由解析记录：

```python
from article.views import article_detail

path('article/<int:article_id>', article_detail, name='article_detail'),
# 路由是article/int的部分全都用一个方法处理
# article_id就是url中出现的数字
# name用来标记这个路由，是它的别名
```

3. 运行服务：

如果没有这个数据：



3. 通过模板获取文章列表

1. 在APP文件夹的 `templates` 目录下创建一个新的模板文件 `article_list.html`：

```
<html>
    <head>
    </head>
    <body>
        {% for article in articles %}
        <li><a href="{% url 'article_detail' article.id %}">{{ article.title }}</a></li>
        {% endfor %}
    </body>
</html>
```

   - 其中，`{% for %}{% endfor %}` 是Django的模板标签，可以将一个QuerySet遍历出来，如果这里不用for标签，直接{{ articles }}，就会在页面上显示一个QuerySet的对象，而不是需要的列表。
   - 而 `{% url %}` 也是Django的模板标签，负责查找路由配置
     `{% url <name> [<para1> , <para2>, ...]}`
     这表明了 `{% url %}` 标签的第一个参数就是路由配置中path（或re_path）函数中的name参数对应的值，从而将链接指定为这个路由；剩下的参数则是可选的，根据路由中的参数，按顺序填写。上文代码块中的 `article.id` 对应的就是其路由配置中 `article/<int:article_id>` 的 `article_id`
   - `<li>` 标签和 `<a>` 标签都是html的基本标签

2. 在APP文件夹下的 `views.py` 中添加一个新的方法如下：

```
def article_list(request):
    articles = Article.objects.all()  # 获取所有Article对象
    content = {}
    content["articles"] = articles
    return render(request, 'article_list.html', content)
```

   由此，html模板文件中的变量就是 `articles`，这是查到的Article对象的查询集合（QuerySet）

3. 在配置文件夹的 `urls.py` 中配置路由如下：

```
from article.views import article_list

path('article/', article_list, name='article_list'),
```

4. 运行服务，打开 `127.0.0.1:8000/article/`：

证明配置成功了

5. 再看 {% url %} 标签：
   由此我们就可以发现，url标签的使用正是通过如上3步进行配置的，每一步都互相影响。
   此外，这里选择使用看起来很复杂的 {% url %} 标签，而不是直接配置成 ，是因为如果不用url标签，一旦路由配置中的对应项路由进行了修改，所有使用这个路由的地方都需要重新改变；而使用url标签就可以大大简化这个过程，进一步解耦合。

六、路由管理进阶——通过总路由管理每个APP的路由

创建APP的路由文件：

在article文件夹下创建 urls.py ，并将总路由配置文件（配置文件夹下的 urls.py ）中与article有关的部分剪切到APP的路由配置文件中，得到**APP**的路由配置如下：

```
from django.urls import path
from . import views  # 这里是article的views

urlpatterns = [
    # 此时这里的路由地址已经是"127.0.0.1:8000/article"了
    path('', views.article_list, name='article_list'),
    path('<int:article_id>', views.article_detail, name='article_detail'),
]
```

同时，将总路由配置文件改为：

```python
from django.contrib import admin
from django.urls import path, include  # include是为了管理APP中的路由，引入的函数
from . import views  # 配置文件夹下的views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', views.hello),
    path('article/', include('article.urls')),  # 表示只要路由是article/xxx的部分全都由article的urls进行处理
]
```

七、静态资源

在开发过程中，一般将：

- 图片
- CSS
- js
- 插件

当做静态文件处理。

静态文件的文件夹默认名称是`static`，`static`的标准目录格式是：

```
- static
    - img
    - CSS
    - js
    - plugins
```

一般会将`static`放在对应的APP文件夹下，在html模板中引用对应位置的资源是，需要使用`{% static %}`标签引用：

```
{% load static %}  <!-- 如果下面用{% static %}标签，则在模板开头需要使用这个标签加载资源 -->

<img src="{% static 'img/1.png' %}" alt="">
```

静态文件的设置在配置文件夹下的`settings.py`文件中有对应的配置。

八、请求和响应

这一部分主要是处理用户发来的请求并向用户返回响应的过程。

请求

- 请求的方式：GET/POST
  在视图中获取请求：每个视图的函数的参数都是request，其中就包含了用户提交的所有请求信息。通过`reuqest.method`我们可以获取到用户请求的方式。
- GET请求：直接在URL上传递参数
  在URL的末尾增加"?n1=xxx&n2=xx"，通过`request.GET`就可以获取到传递参数的字典
- POST请求会发送一个请求体，通过`request.POST`获得请求体的数据

响应

- 简单的响应——直接返回字符串`HTttpResponse(str)`，如果这里的字符串符合html格式，也会被自动变成对应的样式
- 结合htnml模板的响应——`render()`
- 重定向响应——`redirect(new_url)`，收到这个响应的用户会对新的url发起请求

表单机制

在html中设置表单时，注意django的csrf_token机制，需要加上`{% csrf_token %}`标签

```
<form method="post" action="/login/">
    {% csrf_token %}
    <input type="text" name="username" placeholder="用户名">
    <input type="password" name="pwd" placeholder="密码">
    <input type="submit" value="提交">
</form>
```

同时在对应的路由的视图函数中，可以根据请求方法的不同，做出不同的响应（如果只是GET，就返回登录页面；如果是POST，就解析提交的用户名和密码，进行验证，返回结果）

九、数据库操作

django自带了ORM，可以帮助我们对数据库操作，因此我们只要学习django的ORM就可以完成对数据库的操作了。

django可以对数据库的操作有：

- 创建、删除、修改表（table）
- 对表中的数据进行增删改查

创建和修改表、字段

在APP文件夹下的`models.py`中进行修改，并进行数据库迁移的两条指令：

只要在这个文件中，写了继承了`models.Model`的类，django就会帮我们在数据库中创建`<app名_小写类名>`的表。如上文中写过的：

```
1  class Article(models.Model):
2      title = models.CharField(max_length=30) # 设置一个属性为标题，对应数据库中的字符型，最大长度为30
3      content = models.TextField() # 设置内容属性，对应数据库中的文本型
```

此时，因为我们的APP叫做`article`，所以MySQL数据库中，就创建了`article_article`的表，并创建了`title`和`content`字段。

如果此时需要删除旧字段，则直接在类中删去对应属性，并进行数据库迁移即可；如果需要添加新的字段，考虑到数据库可能原有数据，此时需要为新字段设置默认值或允许为空。

- 设置默认值：`data = models.TextField(default="hello")`
- 允许为空：`data = models.TextField(null=True, blank=True)`

注意：每次调整了**`models.py`**后都需要执行两条数据库迁移的命令

表中数据的增删改查

先从`models.py`中导入模型类，使用类方法进行增删改查，这里以上面的`Article`为例

- 查询
  - 获取所有数据

```
datas = Article.objects.all() # 得到的是一个QuerySet类型的对象
# 遍历这个对象，会得到<Article: Article object (1)> ...，即为实际查到的一行数据
# 每行数据显示的名称可以通过改变模型类中的__str__方法实现
# 获取数据的分量（不同字段的对应值），可以通过访问属性的方法实现
# 如： data.first().title  first()方法是取出QuerySet的第一个数据
```

  - 获取筛选后的数据

```
data = Article.objects.filter(id=1).first()
# 因为即使只查到了返回的也是QuerySet还需要用first()
# 这里限制了返回的对象的id为1
```

- 添加数据

```
Article.objects.create(title="four", content = "article four")
# 直接添加一条新数据，没写到的字段必须是有默认值或者允许为空的
```

- 删除数据
  - 删除指定数据

```
Article.objects.filter(id=1).delete()  # 删除所有id为1的数据
```

  - 删除所有数据

```
 Article.objects.all().delete()
```

- 更新数据
  - 更新所有数据的分量

```
Article.objects.all().update(title="None")
```

  - 更新特定数据的分量

```
Article.objects.filter(id=1).update(title="None")
```

Django学习笔记（三）

以个人博客网站为例

（笔记内容根据bilibili up主 @再敲一行代码 的Django 2.0视频教程和个人理解进行编辑）

一、虚拟环境配置和博客模型构建

1. 安装虚拟环境：pip install virtualenv

（如果使用**Anaconda**管理环境也可以不用安装）

在合适的位置打开控制台，输入`virtualenv <环境名>`在对应的目录下创建一个虚拟环境；进入虚拟环境所在文件夹，`Scripts\activate`激活环境，通过`pip`工具下载`Django和pymysql`

2. 创建一个项目和APP

- 可以直接在虚拟环境的目录下创建项目，命名为`mysite_test`，并按上期笔记完成初始化进行对mysql数据库的适配。
- 切换至项目根目录，创建一个APP命名为`blog`
  （上期笔记：Django学习笔记（一）| Desline - Keep moving ）

3. 创建与博客相关的表结构

首先在`models.py`中设置`Blog`和`BlogType`两个类，一个负责管理文章，另一个管理文章类型，相当于`categories`的功能，进行简单分类; 认为一篇博客对应一种分类。

```
from django.db import models
from django.contrib.auth.models import User

# Create your models here.

# 认为一篇博客对应一种分类

class BlogType(models.Model):
    type_name = models.CharField(max_length=15)

    def __str__(self):
        """
```

```
        在后台显示出每个数据时所用的方式，即以type_name作为BlogType对象的标题
        """
        return self.type_name

class Blog(models.Model):
    title = models.CharField(max_length=50)
    blog_type = models.ForeignKey(BlogType, null=True, blank=True, on_delete=models.SET_NULL)
    content = models.TextField()
    author = models.ForeignKey(User,null=True, blank=True, on_delete=models.SET_NULL) # 作者字典是用户表的外键，当删除User时，对用户的作者栏置空
    created_time = models.DateTimeField(auto_now_add=True) # 自动添加当前时间
    last_updated_time = models.DateTimeField(auto_now=True) # 每次修改后以当前时间记录

    def __str__(self):
        return "<Blog: {}>".format(self.title)
```

注意：为了方便，我们修改 **settings.py** 中的 **TIME_ZONE = 'Asia/Shanghai'**

其次，在 admin.py 中稍微修改一下后台配置，方便人工添加数据：

```
from django.contrib import admin
from .models import Blog, BlogType

# Register your models here.
@admin.register(BlogType) # 通过装饰器来进行注册
class BlogTypeAdmin(admin.ModelAdmin):
    list_display = ('id', 'type_name')

@admin.register(Blog)
class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'blog_type', 'author', 'created_time', 'last_updated_time') # 在后台以表格的形式显示数据，元组里的是列名，也是对应类的属性名
```

4. 进行数据库迁移并创建超级用户

- 使用 python manage.py makemigrations 和 python manage.py migrate 进行数据库迁移
- 创建超级用户 admin 并制造一些数据如下：

二、模板标签进阶

- 过滤器
  - length：获取当前对象长度
  - truncatechars:<limit>：限制对象的显示长度
  - date:"format str"：按照个是字符串显示日期

字符	含义	输出示例
Y	4位，且前面含0的年份	'0001', '2022'
y	2位，且前面含0的年份	'00' ~ '99'
m	月份，2 位数，前面加 0。	'01' 到 '12'
n	没有前导零的月份。	'1' 到 '12'
M	月，3 个字母的文本。	'Jan'
b	月，小写的 3 个字母的文本。	'jan'
h	小时，12 小时格式。	'01' 到 '12'
H	小时，24 小时格式。	'00' 到 '23'
i	分钟。	'00' 到 '59'
s	秒，2 位带前导零的数字。	'00' 到 '59'

其余部分可参考 内置模板标签和过滤器 | Django 文档

- 标签
  - {% empty %} : 属于之前提到过的 {% for %}{% endfor %} 的从句，用来标识如果整个循环为空的情况
  - {# #} : 用于注释的标签，和html的注释 <!-- --> 不同，标签内的注释不会出现在网页的源码中
  - {% if %}{% elif %}{% else %}{% endif %} : 除了 {% else %}和{% endif %} ，前两个标签后面要接一个变量，标签会对这个变量进行判断，如果变量为 True 时（存在、非空、不是 False ），就执行下面的语句。并且标签支持python中 if-elif-else 语句中的运算符，如 >= <= == != 和 and or not in is 等

同时，可以利用以上标签继续完善我们博客的功能：

- 主页显示博文列表，有超链接
- 博文详情页，显示作者、发布时间、分类等，可以回到主页，也可以按分类找博文
- 博文分类页，显示同一类别下的所有博文，与博文列表页相似

由此，我们需要：

- 配置路由
  - 总路由——home

    总配置文件的 urls.py 文件变成以下的样子：

```
from django.contrib import admin
from django.urls import path, include
from blog.views import blog_list

urlpatterns = [
    path('', blog_list, name='home'),
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),
]
```

  - 分路由——blog

    在 blog 目录下创建 urls.py 作为分路由的管理文件：

```
from django.urls import path
from . import views

urlpatterns = [
    # root: 127.0.0.1:8000/blog
    path("<int:blog_id>", views.blog_detail, name="blog_detail"),
    path("type/<int:blog_type_pk>", views.blogs_with_type,
name="blogs_with_type")
]
```

- 为每个路由写视图函数

```
from django.shortcuts import render, get_object_or_404
from .models import Blog, BlogType

# Create your views here.
def blog_list(request):
    context = {}
    context['blogs'] = Blog.objects.all()
    return render(request, 'blog_list.html', context)

def blog_detail(request, blog_id):
    context = {}
    context['blog'] = get_object_or_404(Blog, id=blog_id)
    return render(request, 'blog_detail.html', context)

def blogs_with_type(request, blog_type_pk):
    context = {}
    blog_type = get_object_or_404(BlogType, pk=blog_type_pk)
    context['blogs'] = Blog.objects.filter(blog_type=blog_type)
```

```
    context['blog_type'] = blog_type
    return render(request, 'blogs_with_type.html', context)
```

- 为每个视图函数写需要的html模板（源码过长，此处仅以blog_list.html为例）

```
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>我的博客</title>
</head>
<body>
    <div>
        <h3>个人博客网站</h3>
    </div>
    <hr>
    {% for blog in blogs %}
        <a href="{% url 'blog_detail' blog.id %}">
            <h3>{{ blog.title }}</h3>
        </a>
        <!-- 用truncatechars过滤器限制显示长度为30 -->
        <p>{{ blog.content|truncatechars:30 }}</p>
    {% empty %}
        <!-- 当for循环为空时的提示 -->
        <p>-- 暂无博客，敬请期待！ --</p>
    {% endfor %}
    <!-- 这是一个过滤器，用来获取blogs的数量 -->
    <p>一共有{{ blogs|length }}篇博客</p>
</body>
</html>
```

三、模板嵌套和全局模板文件夹

1. 模板嵌套

- 利用了 `{% block <blocktitle> %}{% endblock %}` 和 `{% extends '<basefilename>' %}` 进行模板的嵌套使用
- 可以将不同模板的重复部分提取出来，针对想要变化的部分设置 `{% block %}`，再在单独的文件中实现里面的内容

- 以上一部分的三个html模板为例，可以提取的公共部分作为 base.html ，写在APP的 templates 目录下：

```
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <a href="{% url 'home' %}">
        <h3>个人博客网站</h3>
    </a>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>
```

这里将网页的页面标题和页面内容作为自定义部分，其余部分是所有以这个模板为基准的模板的共有部分

由此可以将上一步的 blog_list.html 改进为：

```
{% extends 'base.html' %}

{# 页面标题 #}
{% block title %}
    <title>我的博客</title>
{% endblock %}

{# 页面内容 #}
{% block content %}
    {% for blog in blogs %}
    <a href="{% url 'blog_detail' blog.id %}">
        <h3>{{ blog.title }}</h3>
    </a>
    <!-- 用truncatechars过滤器限制显示长度为30 -->
    <p>{{ blog.content|truncatechars:30 }}</p>
    {% empty %}
    <!-- 当for循环为空时的提示 -->
    <p>-- 暂无博客，敬请期待！ --</p>
    {% endfor %}
    <!-- 这是一个过滤器，用来获取blogs的数量 -->
    <p>一共有{{ blogs|length }}篇博客</p>
{% endblock %}

```

- 可以将其余的两个模板文件也改造为嵌套的形式

2. 全局模板文件夹

从上面我们可知，`base.html`很可能是作为整个项目的基础模板，那么将它放在APP目录下的模板文件夹就不应该了；从逻辑上讲，应该放到项目根目录中，所以就需要在配置文件中设置全局模板文件夹的位置。

- 在配置文件夹中的`settings.py`中，找到`TEMPLATES`项下的`DIRS`，修改为`os.path.join(BASE_DIR, 'templates'),`
- 在项目文件夹下创建`templates`文件夹，将`base.html`放入总的`templates`文件夹中

Django学习笔记（二）

以项目：部门管理工具为例

一、创建项目和一个APP

通过pycharm创建一个项目为`Django_practice`，并创建APP`app01`（可以通过pycharm的工具中的`运行manage.py任务`，打开一个常驻的manage.py的命令进行创建），修改对应的`settings.py`文件，注册APP，将数据库改为支持mysql的形式，并将时区设置修改至当地。

二、创建表结构——部门表

修改app01中的models.py如下：

```python
from django.db import models

# Create your models here.
class Department(models.Model):
    """
    部门表
    """
    # verbose_name是别名，起到注释的作用
    title = models.CharField(verbose_name='标题', max_length=15)

class UserInfo(models.Model):
    """
    员工表
    """
    name = models.CharField(verbose_name='姓名', max_length=30)
    password = models.CharField(verbose_name='密码', max_length=20)
    age = models.IntegerField(verbose_name='年龄')
    # DecimalField是十进制小数（不是浮点数），max_digits表示最大长度，decimal_places表示小数位数
    account = models.DecimalField(verbose_name='账户余额', max_digits=10, decimal_places=2, default=0)
    create_time = models.DateTimeField(verbose_name='入职时间')
    # 利用外键进行约束，下面的命名在数据库中的会自动加上"_id"，即在数据库中，这个字段是depart_id
    # to后接关联的表（在哪个表做主键），to_field后接关联的字段
    depart = models.ForeignKey(to='Department', to_field='id', blank=True, null=True, on_delete=models.SET_NULL)
    # 这个是在django中做的约束，与数据库无关；数据库里存的是1和2，最后得到的是男和女
    gender_choice = (
        (1, '男'),
        (2, '女'),
    )
    gender = models.SmallIntegerField(verbose_name='性别', choices=gender_choice)

```

- 关于外键的on_delete：表示删除关联的表中的内容时，这个表的外键应该做的事情

  以上面代码为例，则表示当Department表中有删除时，UserInfo的depart字段的值应该如何变化

  - CASCADE级联删除，在UserInfo中删除对应的整行数据

- SET_NULL 置空，将对应位置设置为NULL ，需要允许为空
- SET_DEFAULT 设为默认值，需要设置默认值
- DO_NOTHING 什么也不做

三、部门表的展示

这里使用最原始的方式进行展示，后面将会使用Django的Form和ModelForm组件

1. 部门列表页面设计以及路由配置

- 路由配置：

```
path('depart/list', views.depart_list, name='depart_list'),
```

- 视图函数：

```
def depart_list(request):
    """
    部门列表
    """
    context = {}
    return render(request, 'depart_list.html', context)
```

- 页面设计：
  - 使用CSS进行美化
  - 这里选择bootstrap样式进行套用和修改
    - 选择导航栏，去掉边框的圆角；删掉搜索框和左侧的下拉选单
    - 选择按钮
    - 选择表格，添加编辑和删除按钮

depart_list.html 展示如下：

```
{%  load static %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    {#  引入压缩版的bootstrap  #}
```

```
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-3.4.1-dist/css/bootstrap.min.css' %}">
    <style>
        {#  设置边缘不是圆角  #}
        .navbar {
            border-radius: 0;
        }
    </style>
</head>
<body>
<nav class="navbar navbar-default">
    <div class="container">

        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">用户管理系统</a>
        </div>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li><a href="{% url 'depart_list' %}">部门管理</a></li>
                <li><a href="#">Link</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">登录</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">Desline <span class="caret"></span></a>
                    <ul class="dropdown-menu">
                        <li><a href="#">个人资料</a></li>
                        <li><a href="#">我的信息</a></li>
                        <li role="separator" class="divider"></li>
                        <li><a href="#">注销</a></li>
                    </ul>
                </li>
            </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
```

```
</nav>
<div class="container">
    <div style="margin-bottom: 10px">
        <a class="btn btn-success" href="#">
            <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
            新建部门
        </a>
    </div>

    <div>
        <div class="panel panel-default">
          <!-- Default panel contents -->
          <div class="panel-heading">
              <span class="glyphicon glyphicon-th-list" aria-hidden="true"></span>
              部门列表
          </div>

          <!-- Table -->
          <table class="table table-bordered">
            <thead>
              <tr>
                <th>ID</th>
                <th>名称</th>
                <th>操作</th>
              </tr>
            </thead>
            <tbody>
              <tr>
                <th scope="row">1</th>
                <td>销售部</td>
                <td>
                    <a class="btn btn-primary btn-xs" href="#">编辑</a>
                    <a class="btn btn-danger btn-xs" href="#">删除</a>
                </td>
              </tr>
            </tbody>
          </table>
        </div>
    </div>
</div>
{# 引入jquery和bootstrap压缩版的js  #}
<script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
<script src="{% static 'plugins/bootstrap-3.4.1-dist/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

2. 使用数据库的数据并在页面上展示

- 先在数据库中的 app01_department 中，为 title 列添加两个数据为 IT部门 和 销售部
  - 在mysql的控制台中，使用 insert into app01_department(title) values ('IT部门'), ('销售部') 插入数据
- 然后修改视图函数：

```python
def depart_list(request):
    """
    部门列表
    """
    context = {}
    depart_queryset = Department.objects.all()
    context['departs'] = depart_queryset
    return render(request, 'depart_list.html', context)
```

- 最后修改页面模板中展示的表格，添加循环：

```
<tbody>
        {% for depart in departs %}
          <tr>
            <th scope="row">{{ depart.id }}</th>
            <td>{{ depart.title }}</td>
            <td>
               <a class="btn btn-primary btn-xs" href="#">编辑</a>
               <a class="btn btn-danger btn-xs" href="#">删除</a>
            </td>
          </tr>
        {% endfor %}
</tbody>
```

四、利用部门表进行增删和编辑

1. 增添新的数据

- 首先增加新的路由为 depart/add ，作为增加数据的页面路径
- 然后编写对应的视图函数 depart_add ：

```python
ef depart_add(request):
    """
    添加新部门
    """
    context = {}
    if request.method == 'GET':
        return render(request, 'depart_add.html', context)
    # 获取填入的标题，这里默认都是合法的；后面可以使用组件进行判断
    title = request.POST.get('title')
    Department.objects.create(title=title)  # 存入数据库
    # 重定向至depart_list页面
    return redirect("/depart/list")
```

- 编写对应的页面模板 `depart_add.html`，在其中仍利用列表展示页面的导航栏，但是将下面的容器改为表单

  注意：要使用 **`{% csrf_token %}`**

  `depart_add.html`：

```
{%  load static %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    {#  引入压缩版的bootstrap  #}
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-3.4.1-dist/css/bootstrap.min.css' %}">
    <style>
        {#  设置边缘不是圆角  #}
        .navbar {
            border-radius: 0;
        }
    </style>
</head>
<body>
<nav class="navbar navbar-default">
    <div class="container">

        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
```

```
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">用户管理系统</a>
        </div>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li><a href="{% url 'depart_list' %}">部门管理</a></li>
                <li><a href="#">Link</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">登录</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">Desline <span class="caret"></span></a>
                    <ul class="dropdown-menu">
                        <li><a href="#">个人资料</a></li>
                        <li><a href="#">我的信息</a></li>
                        <li role="separator" class="divider"></li>
                        <li><a href="#">注销</a></li>
                    </ul>
                </li>
            </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
</nav>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">新建部门</h3>
        </div>
        <div class="panel-body">
            {# action不写默认提交到本地址#}
            <form method="post">
                {% csrf_token %}
                <div class="form-group">
                    <label>标题</label>
                    <input type="text" class="form-control" placeholder="标题" name="title">
                </div>

                <button type="submit" class="btn btn-primary">提 交</button>
            </form>
        </div>
    </div>
```

```
</div>

{# 引入jquery和bootstrap压缩版的js #}
<script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
<script src="{% static 'plugins/bootstrap-3.4.1-dist/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

2. 删除现有数据

- 增加删除所用的路由 `depart/delete/`

  注意，最后的地方一定要有斜杠，因为后面还要接**GET**的参数，总路由中没有最后的斜杠不能匹配

- 为展示页面的每个条目的删除按钮设置链接为 `/depart/delete/?nid={{ depart.id }}`，因为条目在循环中，所以可以为不同数据匹配不同的url

- 编写删除路由的视图函数 `depart_delete`：

```
def depart_delete(request):
    """
    删除部门
    """
    nid = request.GET.get('nid')
    print(nid)
    Department.objects.filter(id=nid).delete()
    # 重定向至depart_list页面
    return redirect("/depart/list")
```

3. 编辑现有数据

- 增加编辑所用的路由 `depart/<int:depart_id>/edit`
- 为展示页面的条目设置编辑页面的地址： `depart/{{ depart.id }}/edit`
- 编写编辑数据的路由对应的视图函数：

```python
def depart_edit(request, depart_id):
    """
    修改部门
    """
    context = {}
    if request.method == 'GET':
        depart = Department.objects.filter(id=depart_id).first()
        context['depart'] = depart
        return render(request, 'depart_edit.html', context)
    title = request.POST.get('title')
    Department.objects.filter(id=depart_id).update(title=title)  # 这里一定使用参数名参数
    return redirect('/depart/list')
```

- 创建编辑路由页面的模板 `depart_edit.html`：

```
{%  load static %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    {# 引入压缩版的bootstrap  #}
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-3.4.1-dist/css/bootstrap.min.css' %}">
    <style>
        {# 设置边缘不是圆角  #}
        .navbar {
            border-radius: 0;
        }
    </style>
</head>
<body>
<nav class="navbar navbar-default">
    <div class="container">

        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">用户管理系统</a>
```

```
        </div>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
          <ul class="nav navbar-nav">
            <li><a href="{% url 'depart_list' %}">部门管理</a></li>
            <li><a href="#">Link</a></li>
          </ul>
          <ul class="nav navbar-nav navbar-right">
            <li><a href="#">登录</a></li>
            <li class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">Desline <span class="caret"></span></a>
              <ul class="dropdown-menu">
                <li><a href="#">个人资料</a></li>
                <li><a href="#">我的信息</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="#">注销</a></li>
              </ul>
            </li>
          </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
</nav>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">编辑部门</h3>
        </div>
        <div class="panel-body">
            {# action不写默认提交到本地址#}
            <form method="post">
                {% csrf_token %}
              <div class="form-group">
                <label>标题</label>
                <input type="text" class="form-control" placeholder="{{ depart.title }}" name="title">
              </div>

              <button type="submit" class="btn btn-primary">保 存</button>
            </form>
        </div>
    </div>
</div>

```

```
{# 引入jquery和bootstrap压缩版的js #}
<script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
<script src="{% static 'plugins/bootstrap-3.4.1-dist/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

五、模板嵌套

- 利用了 `{% block <blocktitle> %}{% endblock %}` 和 `{% extends '<basefilename>' %}` 进行模板的嵌套使用
- 可以将不同模板的重复部分提取出来，针对想要变化的部分设置 `{% block %}`，再在单独的文件中实现里面的内容
- 提取基本模板 `base.html`：

```
{%  load static %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    {# 引入压缩版的bootstrap #}
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-3.4.1-dist/css/bootstrap.min.css' %}">
    <style>
        {# 设置边缘不是圆角 #}
        .navbar {
            border-radius: 0;
        }
    </style>
</head>
<body>
<nav class="navbar navbar-default">
    <div class="container">

        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
```

```
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">用户管理系统</a>
        </div>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li><a href="{% url 'depart_list' %}">部门管理</a></li>
                <li><a href="#">Link</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">登录</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">Desline <span class="caret"></span></a>
                    <ul class="dropdown-menu">
                        <li><a href="#">个人资料</a></li>
                        <li><a href="#">我的信息</a></li>
                        <li role="separator" class="divider"></li>
                        <li><a href="#">注销</a></li>
                    </ul>
                </li>
            </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
</nav>
<div>
    {% block content %}{% endblock %}
</div>
{# 引入jquery和bootstrap压缩版的js #}
<script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
<script src="{% static 'plugins/bootstrap-3.4.1-dist/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

- 其余模板则可以继承base模板，以depart_list为例：

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
    <div style="margin-bottom: 10px">
        <a class="btn btn-success" href="{% url 'depart_add' %}">
            <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
            新建部门
        </a>
```

```
    </div>

    <div>
        <div class="panel panel-default">
          <!-- Default panel contents -->
          <div class="panel-heading">
              <span class="glyphicon glyphicon-th-list" aria-hidden="true"></span>
              部门列表
          </div>

          <!-- Table -->
          <table class="table table-bordered">
            <thead>
              <tr>
                <th>ID</th>
                <th>名称</th>
                <th>操作</th>
              </tr>
            </thead>
            <tbody>
                {% for depart in departs %}
                  <tr>
                    <th scope="row">{{ depart.id }}</th>
                    <td>{{ depart.title }}</td>
                    <td>
                        <a class="btn btn-primary btn-xs" href="/depart/{{ depart.id }}/edit">编辑</a>
                        <a class="btn btn-danger btn-xs" href="/depart/delete/?nid={{ depart.id }}">删除</a>
                    </td>
                  </tr>
                {% endfor %}
            </tbody>
          </table>
        </div>
    </div>
</div>
{% endblock %}
```

2. 全局模板文件夹

从上面我们可知，如果 base.html 是作为整个项目的基础模板，那么将它放在APP目录下的模板文件夹就不应该了；从逻辑上讲，应该放到项目根目录中，所以就需要在配置文件中设置全局模板文件夹的位置。

- 在配置文件夹中的 settings.py 中，找到 TEMPLATES 项下的 DIRS ，修改为 os.path.join(BASE_DIR, 'templates'),
- 在项目文件夹下创建 templates 文件夹，将 base.html 放入总的 templates 文件夹中

注：在本项目中， **base.html** 暂且放在**APP**的模板文件夹下即可；此外，使用**pycharm**构建项目会自动再配置文件中设置好全局模板文件夹的位置，并创建全局模板文件夹

六、用户的展示

1. 如第四步，创建对应页面和路由以及视图函数

- 路由： path('user/list', views.user_list, name='user_list'),
- 视图函数：

```
def user_list(request):
    """
    用户列表
    """
    context = {}
    users = UserInfo.objects.all()
    context['users'] = users
    return render(request, 'user_list.html', context)

```

```
使用python语句对获取的数据进行调整的办法：
for user in users:
    user.create_time.strftime('%Y-%m-%d')  # 将日期类型调整为字符串
    user.get_gender_display()  # 利用了之前模型中设定的choices直接进行查找，将数据库对应的数据按规则映射为结果
    # 这个方法是user.get_字段名称_display()
    user.depart  # 相当于直接从被关联的表中取出对应id的行，所以这里是个Department表的查询对象
    user.depart.title  # 取到部门名称
注意：这些在Django的html模板中，仍需要调整
```

- 页面模板 user_list.html：

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
        <div style="margin-bottom: 10px">
            <a class="btn btn-success" href="#">
                <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
                新建用户
            </a>
        </div>

        <div>
            <div class="panel panel-default">
              <!-- Default panel contents -->
              <div class="panel-heading">
                  <span class="glyphicon glyphicon-th-list" aria-hidden="true"></span>
                  用户列表
              </div>

              <!-- Table -->
              <table class="table table-bordered">
                <thead>
                  <tr>
                      <th>ID</th>
                      <th>姓名</th>
                      <th>密码</th>
                      <th>年龄</th>
                      <th>账户余额</th>
                      <th>入职时间</th>
                      <th>部门ID</th>
                      <th>性别</th>
                      <th>操作</th>
                  </tr>
                </thead>
                <tbody>
                    {% for user in users %}
                      <tr>
                          <th scope="row">{{ user.id }}</th>
                          <td>{{ user.name }}</td>
                          <td>{{ user.password }}</td>
                          <td>{{ user.age }}</td>
                          <td>{{ user.account}}</td>
                          {# 过滤器，注意冒号和引号之间不能有空格 #}
                          <td>{{ user.create_time|date:"Y-m-d"}}</td>
```

```
                    <td>{{ user.depart.title }}</td>
                    {# 这里直接使用方法，不能加括号 #}
                    <td>{{ user.get_gender_display }}</td>
                    <td>
                      <a class="btn btn-primary btn-xs" href="#">编辑</a>
                      <a class="btn btn-danger btn-xs" href="#">删除</a>
                    </td>
                  </tr>
                {% endfor %}
              </tbody>
            </table>
          </div>
        </div>
      </div>
{% endblock %}
```

2. Form与ModelForm基础

通过部门表的展示和增删改的过程我们可以知道，如果使用最原始的逻辑进行页面布局设计，需要进行大量的工作，页面需要展示数据库中的每个字段，这些全需要我们手动完成，一旦数据表庞大时，工作量将非常大，不适合应用到工程当中。

此外，原始的方法还有以下几个缺点：

1. 用户提交没有数据检验
2. 每个字段都需要我们在页面上编写
3. 输入非法时，没有错误提示
4. 关联的数据，仍需要我们手动进行连表查询

为解决这些问题，Django提供了 `Form` 和 `ModelForm` 的组件，其中 `Form` 组件只能解决前三个的问题

- `Form` 组件示例:
  在 `views.py` 中，添加继承了 `Form` 组件的类，下面仅为demo示例：

```python
from xxx import Form

class MyForm(Form):
    name = forms.CharField(widget=forms.input)  # 在html中显示为输入框
    xxx...

def depart_add(request):
    form = MyForm
    context = {}
    context['form'] = form
    return render(request, 'depart_add.html', context)
```

在html模板中：

```
<!-- 可以通过这种方式使用 -->
{{ form.name }}
<!-- 也可以通过循环使用 -->
{% for field in form%}
    {{ field }}
{% endfor %}
```

但是通过使用我们可以发现，实际上 MyForm 中的属性和 models.py 中设置的属性是几乎一样的，因为为了更加简便的使用，我们引入了 ModelForm 组件（这个组件适合进行数据库的增删改）

- ModelForm 组件示例：

  在 views.py 中的demo:

```python
from xxx import ModelForm

class MyForm(ModelForm):
        xx = models.CharField()  # 自定义的字段
    class Meta:
            model = UserInfo  # 这是models.py中写好的类
                fields = ["name", "password", "xx"]  # 允许出现的字段

def depart_add(request):
    form = MyForm
    context = {}
    context['form'] = form
    return render(request, 'depart_add.html', context)
```

如果想使用 models.py 类中所有字段，可以在Meta中设置为 fields = '__all__'；

如果想排除某个字段，可以使用 exclude = ['xxx']

3. 基于ModelForm完成用户添加

- 配置路由 `user/add`
- 完成对应的视图函数的编写 `user_add`：

```python
def user_add(request):
    context = {}
    if request.method == 'GET':
        form = UserModelForm()
        context['form'] = form
        return render(request, 'user_add.html', context)

    form = UserModelForm(data=request.POST)
    context['form'] = form
    # 进行数据校验
    if form.is_valid():
        # 数据合法
        form.save()  # 保存数据到数据库
        return redirect('/user/list')
    else:
        # 返回提示了错误信息的页面
        return render(request, 'user_add.html', context)
```

- 完成页面模板设计 `user_add.html`：

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">新建员工</h3>
        </div>
        <div class="panel-body">
            {# action不写默认提交到本地址#}
            <form method="post" novalidate>  <!-- 让浏览器不进行检查 -->
                {% csrf_token %}
                {% for field in form %}
                    <div class="form-group">
                        {# 使用label标签引用之前在verbose_name写的别名 #}
                        <label>{{ field.label }}</label>
                        {{ field }}
                        {# field.errors是一个列表，这里取第一项作为错误提示信息 #}
                        <span style="color:red">{{ field.errors.0 }}</span>
                    </div>
                {% endfor %}
              <button type="submit" class="btn btn-primary">提 交</button>
            </form>
```

```
        </div>
    </div>
</div>
{% endblock %}
```

4. 用户删除和编辑

- 删除的部分和第四步相同
- 编辑的部分可以套用ModelForm进行完成，需要注意的是，要想确定显示个和更新数据库上的某一行，需要提前取出，并写入 `UserModelForm` 中的 `instance` 参数
- 视图函数 `user_edit` :

```python
def user_edit(request, user_id):
    """
    编辑用户信息
    """
    context = {}
    row_data = UserInfo.objects.filter(id=user_id).first()  # 从数据库取数据
    if request.method == 'GET':
        form = UserModelForm(instance=row_data)  # 使得生成的输入框中有默认数据
        context['form'] = form
        return render(request, 'user_edit.html', context)

    form = UserModelForm(instance=row_data, data=request.POST)
    context['form'] = form
    if form.is_valid():
        form.save()  # 在备注了instance后，save方法可以进行更新操作
        return redirect('/user/list')
    else:
        return render(request, 'user_add.html', context)
```

- 模板和添加所所用模板十分相似，稍作修改即可，这里不展示

❄ 七、ModelForm进阶

1. 使用正则表达式进行输入格式校验

示例demo：

```
from django.core.validators import RegexValidator

class MyForm(ModelForm):
      xx = models.CharField(validators=[
         RegexValidator(r'<正则表达式>', '不符合这个表达式报错信息'),
      ]) # 自定义的字段
   class Meta:
         model = UserInfo  # 这是models.py中写好的类
            fields = ["name", "password", "xx"]  # 允许出现的字段
```

validators是一个列表，可以放多个校验要求

2. 自定义校验——重写校验方法

- 首先明确的是，用于校验的方法的命名规则是：`<clean_字段名>`，要给哪个字段加校验，就重写对应的校验方法
- 其次，获取用户输入的方法是：`self.cleaned_data[<字段名>]`
- 然后可以对用户的输入进行校验，不通过可以raise错误；通过则直接返回用户输入
- 仅做demo:

```
from django.core.validators import ValidationError

class MyForm(ModelForm):
      xx = models.CharField(validators=[
         RegexValidator(r'<正则表达式>', '不符合这个表达式报错信息'),
      ]) # 自定义的字段
   class Meta:
         model = UserInfo  # 这是models.py中写好的类
            fields = ["name", "password", "xx"]  # 允许出现的字段

      def clean_name(self):
            input_name = self.cleaned_data['name']
            if len(input_name) < 2:
               raise ValidationError('长度太短！')

            return input_name
```

3. 建立Bootstrap父类

由于在视图函数中可能使用多个ModelForm类，并且需要生成的是Bootstrap的html样式，就可以建立一个集成了ModelForm类的Bootstrap类，方便其余类进行继承。

下面仅展示demo:

```
class BootstrapModelForm(ModelForm):
    def __init__(self, *args, **kwargs):
        """
        通过控制每个field的widget和widget的属性来使用CSS进行页面优化
        """
        super().__init__(*args, **kwargs)
        for name, field in self.fields.items():
            if name == "password":
                field.widget = forms.PasswordInput()
            field.widget.attrs['class'] = 'form-control'  # 为输入框组件添加Bootstrap规定的属性
            field.widget.attrs['placeholder'] = name
```

```
class MyForm(BootstrapModelForm):
    class Meta:
        ......
```

此时，子类一般情况下无需再写 `__init__` 方法了。

八、增加搜索功能——以用户管理为例

1. 关于搜索的ORM查询语句

```
# 以UserInfo为例
UserInfo.objects.filter(id=1)  # id = 1
UserInfo.objects.filter(id__gt=1)  # id > 1
UserInfo.objects.filter(id__gte=1)  # id >= 1
UserInfo.objects.filter(id__lt=1)  # id < 1
UserInfo.objects.filter(id__lte=1)  # id <= 1

UserInfo.objects.filter(name='小明')  # name是'小明'
UserInfo.objects.filter(name__startswith='小')  # 名字以'小'开头
UserInfo.objects.filter(name__endswith='明')  # 名字以'名'结尾
UserInfo.objects.filter(name__contains='明')  # 名字中含有'明'

UserInfo.objects.filter(name='小明', id=1)
data_dict = {'name': '小明', 'id': 1}
UserInfo.objects.filter(**data_dict)  # 与上面的等价
```

2. 添加按姓名搜索的功能

- 仍在 `user/list` 路由的基础上进行，使用 `GET` 方法传参
- 修改后的 `user_list`：

```python
def user_list(request):
    """
    用户列表
    """
    context = {}
    data_dict = {}
    search = request.GET.get('search')
    if search:
        data_dict['name__contains'] = search
        print(search)
    users = UserInfo.objects.filter(**data_dict)
    context['users'] = users
    return render(request, 'user_list.html', context)
```

- 修改后的页面模板 `user_list.html`：

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
        <div style="margin-bottom: 10px">
            <a class="btn btn-success" href="{% url 'user_add' %}">
                <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
                新建用户
            </a>
            <div style="float:right; width: 300px">
                <form method="get">
                    <div class="input-group">
                        <!-- 注意给input框加name，和视图函数中get接收的名字一样 -->
                        <input type="text" name="search" class="form-control"
placeholder="Search for...">
                        <span class="input-group-btn">
                            <button class="btn btn-default" type="submit">  <!-- 这里注意要改
为submit提交表单 -->
                                <span class="glyphicon glyphicon-search" aria-hidden="true">
</span>
                            </button>
                        </span>
                    </div><!-- /input-group -->
                </form>
            </div>
```

```
        </div>

        <div>
            <div class="panel panel-default">
              <!-- Default panel contents -->
              <div class="panel-heading">
                  <span class="glyphicon glyphicon-th-list" aria-hidden="true"></span>
                  用户列表
              </div>

              <!-- Table -->
              <table class="table table-bordered">
                <thead>
                  <tr>
                      <th>ID</th>
                      <th>姓名</th>
                      <th>密码</th>
                      <th>年龄</th>
                      <th>账户余额</th>
                      <th>入职时间</th>
                      <th>性别</th>
                      <th>所属部门</th>
                      <th>操作</th>
                  </tr>
                </thead>
                <tbody>
                    {% for user in users %}
                      <tr>
                          <th scope="row">{{ user.id }}</th>
                          <td>{{ user.name }}</td>
                          <td>{{ user.password }}</td>
                          <td>{{ user.age }}</td>
                          <td>{{ user.account}}</td>
                          {# 过滤器，注意冒号和引号之间不能有空格 #}
                          <td>{{ user.create_time|date:"Y-m-d"}}</td>
                          {# 这里直接使用方法，不能加括号 #}
                          <td>{{ user.get_gender_display }}</td>
                          <td>{{ user.depart.title }}</td>
                          <td>
                            <a class="btn btn-primary btn-xs" href="/user/{{ user.id }}/edit">编辑</a>
                            <a class="btn btn-danger btn-xs" href="/user/delete/?nid={{ user.id }}">删除</a>
                          </td>
                      </tr>
```

```
                {% endfor %}
            </tbody>
          </table>
        </div>
      </div>
    </div>
  {% endblock %}
```

九、增加分页功能

首先我们要知道，django是提供一个的分页器的类的，叫做`Paginator`，在`django.core.paginator`下，不过这里暂时不使用这个类，而是选择在视图函数中自行实现这个功能。

这里我们仍然使用`user_list`这个视图函数作为范例。

1. 制造数据

由于我们之前只是手动输入了部分数据，数据量太小，不方便进行分页的展示，因此我们制造一些数据并添加进数据库中。

这里需要用到一个**python**的库是**`faker`**，可以用来生成数据。

制造数据的代码如下：

```
1
2  # 制造数据
3
4  import faker
5  import random
6
7  fake = faker.Faker("zh-cn") # 生成中文数据
8  for i in range(100):
9      name = fake.name() # 制造姓名
10     password = fake.password(length=8, special_chars=False, digits=True,
   upper_case=False, lower_case=True) # 制造一个8位密码，不允许有大写字母和特殊字
   符，可以有数字和小写字母
11     age = random.randint(18, 55) # 随机生成年龄
12     account = round(random.uniform(0, 10000), 2) # 随机生成一个账户余额，约到两位小
   数
13     create_time = fake.date() # 生成一个随机日期
```

```
    depart_id = random.choice([i for i in Department.objects.all()]) # 从现有的部门实例中随机选择一个
    gender = random.choice([1, 2]) # 随机性别
    # print(f"name:{name}\npassword:{password}\nage:{age}\naccount:{account}\ncreate_time:{create_time}\ndepart_id:{depart_id}\ngender:{gender}")
    UserInfo.objects.create(name=name, password=password,
                            age=age, account=account, create_time=create_time, depart=depart_id, gender=gender)
    # 将上述随机数据存到数据库
    # print("OK")

```

注意：制造数据的代码要插在视图函数中间，不需要使用后注释掉即可。

2. 修改视图函数

```
def user_list(request):
    """
    用户列表
    """
   """这里可以插入制造数据的代码~"""

    context = {}
    data_dict = {}
    PAGE_SIZE = 10  # 一页里面展示的数据（一页中有几行数据）
    PAGE_BAR_SIZE = 10  # 一共展示出几个分页（底部分页栏中展示出几个页面的标记）
    search = request.GET.get('search')
    page = int(request.GET.get('page', 1))  # 获取当前页码

    # 这个部分是用来修复搜索之后分页的bug的，有了这个部分，可以用GET方法传多个参数，其分页会随之变化
    param_dict = request.GET.copy()  # request.GET得到一个QueryDict，这里对它进行拷贝
    if 'page' in param_dict:
        param_dict.pop('page')  # 去掉page参数
    other_param_url = param_dict.urlencode()  # 得到其他所有参数的url
    context['other_param_url'] = other_param_url

    if search:
        data_dict['name__contains'] = search

    users = UserInfo.objects.filter(**data_dict)
    start = (page - 1) * PAGE_SIZE
```

```
    end = page * PAGE_SIZE
    context['users'] = users[start:end]

    if len(users) % PAGE_SIZE != 0:
        page_num = len(users) // PAGE_SIZE + 1
    else:
        page_num = len(users) // PAGE_SIZE

    if page_num <= PAGE_BAR_SIZE:
        page_list = [i+1 for i in range(page_num)]
    else:
        if page < (PAGE_BAR_SIZE // 2):
            page_list = [i+1 for i in range(PAGE_BAR_SIZE)]
        elif page > (page_num - (PAGE_BAR_SIZE // 2)):
            temp = page_num - PAGE_BAR_SIZE + 1
            page_list = [i for i in range(temp, page_num+1)]
        else:
            end_page = page + (PAGE_BAR_SIZE // 2)
            start_page = page - (PAGE_BAR_SIZE // 2) + 1
            page_list = [i for i in range(start_page, end_page+1)]

    context['pages'] = page_list
    context['current_page'] = page
    context['total_page_num'] = page_num
    return render(request, 'user_list.html', context)
```

3. 修改页面模板

修改后的 `user_list.html` :

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
        <div style="margin-bottom: 10px">
            <a class="btn btn-success" href="{% url 'user_add' %}">
                <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
                新建用户
            </a>
            <div style="float:right; width: 300px">
                <form method="get">
                    <div class="input-group">
                        <!-- 注意给input框加name，和视图函数中get接收的名字一样 -->
```

```
                    <input type="text" name="search" class="form-control" placeholder="Search for...">
                    <span class="input-group-btn">
                      <button class="btn btn-default" type="submit">  <!-- 这里注意要改为submit提交表单 -->
                        <span class="glyphicon glyphicon-search" aria-hidden="true"></span>
                      </button>
                    </span>
                  </div><!-- /input-group -->
                </form>
              </div>
            </div>

            <div>
              <div class="panel panel-default">
                <!-- Default panel contents -->
                <div class="panel-heading">
                  <span class="glyphicon glyphicon-th-list" aria-hidden="true"></span>
                  用户列表
                </div>

                <!-- Table -->
                <table class="table table-bordered">
                  <thead>
                    <tr>
                      <th>ID</th>
                      <th>姓名</th>
                      <th>密码</th>
                      <th>年龄</th>
                      <th>账户余额</th>
                      <th>入职时间</th>
                      <th>性别</th>
                      <th>所属部门</th>
                      <th>操作</th>
                    </tr>
                  </thead>
                  <tbody>
                    {% for user in users %}
                      <tr>
                        <th scope="row">{{ user.id }}</th>
                        <td>{{ user.name }}</td>
                        <td>{{ user.password }}</td>
                        <td>{{ user.age }}</td>
                        <td>{{ user.account}}</td>
```

```
                    {# 过滤器，注意冒号和引号之间不能有空格 #}
                    <td>{{ user.create_time|date:"Y-m-d"}}</td>
                    {# 这里直接使用方法，不能加括号 #}
                    <td>{{ user.get_gender_display }}</td>
                    <td>{{ user.depart.title }}</td>
                    <td>
                      <a class="btn btn-primary btn-xs" href="/user/{{ user.id }}/edit">编辑</a>
                      <a class="btn btn-danger btn-xs" href="/user/delete/?nid={{ user.id }}">删除</a>
                    </td>
                  </tr>
                {% endfor %}
              </tbody>
            </table>
          </div>
        </div>

        <nav aria-label="Page navigation">
          <ul class="pagination">
            <li>
              <a href="?page=1&{{ other_param_url }}" aria-label="Begin">首页</a>
            </li>
            {# 第一页没有向前翻页的标记 #}
            {% if current_page != 1 %}
              <li>
                <a href="?page={{ current_page|add:-1 }}&{{ other_param_url }}" aria-label="Previous">
                  <span aria-hidden="true">&laquo;</span>
                </a>
              </li>
            {% endif %}
            {% for page in pages %}
              {# 在页面中标记当前的页 #}
              {% if page == current_page%}
                <li class="active"><a href="?page={{ page }}&{{ other_param_url }}">{{ page }}</a></li>
              {% else %}
                <li><a href="?page={{ page }}&{{ other_param_url }}">{{ page }}</a></li>
              {% endif %}
            {% endfor %}
            {# 最后一页没有向后翻页的标记 #}
            {% if current_page != total_page_num %}
              <li>
```

```
                    <a href="?page={{ current_page|add:1 }}&{{ other_param_url }}" aria-label="Next">
                      <span aria-hidden="true">&raquo;</span>
                    </a>
                  </li>
                {% endif %}
                <li>
                  <a href="?page={{ total_page_num }}&{{ other_param_url }}" aria-label="Begin">尾页</a>
                </li>
              </ul>
            </nav>
          </div>
{% endblock %}
```

十、使用bootstrap的时间选择组件

在本例中，我们选择用户入职时间时，如果手动输入可能不太方便，因此可以引入时间选择组件。需要注意的是时间选择组件和bootstrap一样需要提前下载。

1. 修改base.html

为了增加自定义的css和js文件，我们需要修改基础模板中的部分，增加可以放入自定义css和js文件的两个block。即在 `head` 部分增加 `{% block css %}{% endblock %}`，在末尾添加js文件的后面再加上 `{% block js %}{% endblock %}`

2. 对模板页面user_add.html进行修改

在 `user_add.html` 中添加如下内容：

```
{% load static %}

{% block css %}
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-datetimepicker/css/bootstrap-datetimepicker.min.css' %}">
{% endblock %}

{% block js %}
```

```
    <script src="{% static 'plugins/bootstrap-datetimepicker/js/bootstrap-datetimepicker.min.js' %}"></script>
    <script src="{% static 'plugins/bootstrap-datetimepicker/js/locales/bootstrap-datetimepicker.zh-CN.js' %}"></script>
    {# 其中id_create_time是django的ModelForm给create_time字段输入框的默认id，要和下面的js代码相对应 #}
    {# 这个也可以通过后端，将create_time输入框的id修改为dt，下面的也要改成#dt，方便一个页面上有多个需要使用组件的输入框 #}
    <script>
        $(function(){
            $('#id_create_time').datetimepicker({
                format: 'yyyy-mm-dd',
                startDate: '0',
                language: 'zh-CN',
                autoclose: true
            });
        })
    </script>
{% endblock %}
```

十一、增加管理员

1. 增加管理员表

ID	用户名	密码

在 models.py 中进行添加，并执行2条迁移命令：

```
class Admin(models.Model):
    """
    管理员表
    """
    username = models.CharField(verbose_name='用户名', max_length=32)
    password = models.CharField(verbose_name='密码', max_length=64)
```

2. 根据管理员表添加简单的登录功能

- 添加登录界面路由 login/
- 编写视图函数 login：

```python
class LoginForm(forms.Form):
    username = forms.CharField(label='用户名', max_length=32,
                               widget=forms.TextInput(attrs={'class': 'form-control',
                                                             'placeholder': 'username'}))
    password = forms.CharField(label='密码', max_length=64,
                               widget=forms.PasswordInput(attrs={'class': 'form-control',
                                                                 'placeholder': 'password'}))


def login(request):
    """
    用户登录
    """
    context = {}
    if request.method == 'GET':
        form = LoginForm()
        context['form'] = form
        return render(request, 'login.html', context)

    form = LoginForm(data=request.POST)
    context['form'] = form
    if form.is_valid():
        # 和数据库数据做比较
        row_data = Admin.objects.filter(**form.cleaned_data).first()  # 得到一个数据对象
        if row_data:
            # 如果能查到数据，通过验证
            # 自动生成cookies并在session中存数据到数据库，数据库中的session_id就是cookies
            request.session['info'] = {'id': row_data.id, 'username': row_data.username}
            return redirect('/admin/list')
        else:
            form.add_error('password', '用户名或密码错误')
            return render(request, 'login.html', context)
```

- 编写网页模板 login.html :

```
{% load static %}

<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <link rel="stylesheet" href="{% static 'plugins/bootstrap-3.4.1-dist/css/bootstrap.min.css' %}">
    <meta charset="UTF-8">
    <title>Title</title>
```

```
</head>
<body>
    <div class="container">

      <form class="form-signin" method="post">
          {% csrf_token %}
        <h2 class="form-signin-heading">用户登录</h2>
          <label >用户名</label>
          {{ form.username }}
          <span style="color:red">{{ form.username.errors.0 }}</span>
          <label >密码</label>
          {{ form.password }}
          <span style="color:red">{{ form.password.errors.0 }}</span>
        <button class="btn btn-lg btn-primary btn-block" type="submit">登 录</button>
      </form>

    </div>
<script src="{% static 'plugins/bootstrap-3.4.1-dist/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

此时，第一次进行登录时，就会自动生成cookies，并存入数据库中；我们可以对需要登录才能查看的网页进行简单的限制，在网页的视图函数最前面加上：

```
info = request.session.get('info')
    if not info:
        # info不存在说明没有登录，返回登录页面
        return redirect('/login')
```

但是，当我们的视图很多时，再这样进行操作就太过麻烦，因此我们引入django的中间件功能。

3. 中间件初步

- 编写中间件：

```python
from django.utils.deprecation import MiddlewareMixin

class M1(MiddlewareMixin):
    def process_request(self, request):
        xxxx

    def process_response(self, request, response):
        xxxx
        return response
```

注意，process_request一般是没有返回值的（是None），此时请求会继续往下走，经过后面的中间件，再向视图函数去；若有返回值（应是和视图函数一样的返回值），则直接返回，将不能到达视图函数。

- 在 settings.py 的 MIDDLEWARE 中注册中间件

4. 利用中间件完成登录控制

编写中间件并注册

```python
from django.shortcuts import redirect
from django.utils.deprecation import MiddlewareMixin


class LoginMiddleware(MiddlewareMixin):

    def process_request(self, request):
        # 1. 排除不需要认证就可以查看的页面
        if request.path_info == '/login/':
            return

        # 2. 进行认证
        info = request.session.get('info')
        if not info:
            # info不存在说明没有登录，返回登录页面
            return redirect('/login/')
```

5. 增加注销功能

- 增加url logout/
- 编写视图函数：

```python
def logout(request):
    request.session.clear()
    return redirect('/login/')
```

直接利用提供的方法删除数据库中的这条cookies记录

- 在页面中增加按钮，并绑定路由

6. 增加用户登录的页面显示

在html模板中，利用 `render` 传的request中的参数。因为我们之前在后端已经设定了当生成cookies的时候记录的数据叫'info'，因此我们可以直接在后端通过 `request.session['info']` 获取到我们当时存入的信息（在本例中是一个字典{'id':xxx, 'username':xxx}）。所以，在html模板中，我们可以通过 `{{ request.session.info.username }}` 获取当前登录的用户名。