django Hello-Worlddd

django安装

> `pip install django==2.0.4`(版本号)
>
> `pip install django`默认安装最新版本

创建项目

> `django-admin startproject myproject`

开启开发服务器

> `cd myproject`：进入项目目录
>
> `python manage.py runserver`：开启服务
>
> `python manage.py runserver 7000`：改变服务监听端口
>
> `python manage.py runserver 0:8000`：改变服务监听IP:端口

项目文件夹

```
manage.py：用来管理当前项目的一个命令行工具
myproject/：  项目主文件夹
myproject/__init__.py：空文件，用来指明当前的myproject为一个可导入的模块包
myproject/settings.py：项目主要配置文件
myproject/urls.py：项目主要路由配置文件
myproject/wsgi.py：项目部署WSGI并发服务器时所需要的配置文件
```

Settings.py

> 该文件是整个项目的主控文件，其中相关配置选项如下
>
> https://docs.djangoproject.com/zh-hans/2.0/ref/settings/

```
- BASE_DIR：当前项目工作目录，用来在每一次开启项目时动态找到相关资源路径
- SECRET_KEY：加密的hash值以及保护某些签名数据的关键密钥
- DEBUG：调试模式
- ALLOWED_HOSTS：有哪些主机或域名可以访问当前django站点，如设置为*代表全部可访问。
- INSTALL_APPS：django项目中所有使用的应用名称，自创建子应用也要加到这里，不然ORM数据库无法被识别到！
- MIDDLEWARE：django中间件，用来在request或reponse过程中添加功能，比如确保安全性，传输保存Session等
    - SecurityMiddleware：xss脚本过滤，一些安全设置
    - SessionMiddleware：session支持中间件，在每次用户访问django项目时，添加session对每一个浏览器
    - CommonMiddleware：通用组件，比如为路由添加末尾斜杠
    - CsrfViewMiddleware：防跨站请求伪造令牌，为客户端添加csrf_token密钥，在表单提交时需提交该值
    - AuthenticationMiddleware：admin用户组件，每个request对象都会被添加admin下的user属性
    - MessageMiddleware：消息中间件 展示一些后台消息给前端
    - XFrameOptionsMiddleware：防止欺骗点击攻击出现；自身页面被嵌入到他人页面中，点击欺骗
```

- ROOT_URLCONF: 主路由配置文件，字符串填写url.py文件路径
- TEMPLATES: 模板文件配置项
- WSGI_APPLICATION: WSGI服务器配置项，找到当前django下的wsgi引入APP文件
- DATABASES: 数据库配置项，默认使用SQLite3，一个本地文件数据库
- AUTH_PASSWORD_VALIDATORS: 检查用户密码强度的验证程序列表，不过是针对admin界面下的用户，而非自定义
- LANGUAGE_CODE: django所使用语言文件
- TIME_ZONE: django所使用时区
- USE_I18N: 国际化支持 18表示Internationalization这个单词首字母I和结尾字母N之间的字母有18个
- USE_L10N: 是localization的缩写形式，意即在l和n之间有10个字母
- USE_TZ:开启了Time Zone功能，则所有的存储和内部处理，包括print显示的时间将是是UTC时间格式
- STATIC_URL: URL访问静态资源时的路径

来搞个Hello world

django创建子应用

> 项目和应用有啥区别?
>
> 应用是一个专门做某件事的网络应用程序：比如博客系统，或者公共记录的数据库，或者简单的投票程序
>
> 项目则是一个网站使用的配置和应用的集合。项目可以包含很多个`app`应用，应用可以被很多个项目使用

- `python manage.py startapp myapp`

  > 创建子应用

app目录

- admin.py: app在admin注册展示时需要的文件
- views.py: app的功能视图函数文件
- models.py: app需要使用数据库时的文件
- urls.py: 当使用include路由分发时，每个app应该有他自己的子路由文件，这个是默认没有创建好的

视图函数

> 打开`app`下的`views.py`文件
>
> `Web`访问起始就是通过一个`URL`连接地址访问到服务器上的一个函数
>
> 在`views.py`中我们通过编写函数的形式，接收用户请求的`request`并返回一个`response`

```
# 每一个视图函数都需要有一个必须参数 request,用来接收用户访问时的请求内容
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h1>Hello world</h1>")
```

- `HttpResponse`函数用来向用户返回一个字符串

路由配置

> 创建好了一个可以在请求时返回`H1`标签的视图函数，但是现在通过浏览器还是访问不到

> 需要我们为这个`app`下的函数进行路由配置

> 第一种简单的路由配置，直接在主控路由文件下，找到这个视图函数

```
#myproject/urls.py
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls), #admin控制界面路由
    path('',views.index)
    #path函数第一个参数为访问地址，空字符串代表：当用户直接访问首页时
    #第二个参数代表访问该地址时对应的视图函数，我们引入了app下的views中的index视图函数
]
```

- 接下来访问`127.0.0.1:8000`，那么你会看到一个非常大的`Hello world`

> 以上将视图函数的查找直接写到主控路由并不是最好的办法
>
> 我们的项目通常会有非常多的路由配置项，如果都堆到这个文件中肯定是非常乱的，难以维护

- 我们可以在对应`app`下创建一个子路由控制文件，并在其中设置视图的路由配置

```
#myapp/urls.py
from django.urls import path
from . import views
urlpatterns = [
    path("",views.index)
]
```

> 现在虽然配置了`app`下的路由文件，但是访问时，是看不到对应视图的结果
>
> 这是因为默认的`url`查找动作将会从主控路由文件开始，我们还需要在主控路由文件下进行路由分发设置
>
> 让主控路由可以找到子`app`下的路由映射文件

```
#myproject/urls.py
from django.contrib import admin
from django.urls import path,include
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    #path('',views.index)
    path('',include("myapp.urls")),
    # 函数 include() 允许引用其它 URLconfs
]
```

- 接下来再次尝试，在浏览器中访问主机域名；如果可以看到的话，恭喜你，效果已经很棒了！

路由查找流程

1. 查找主控路由文件下的`urlpatterns`全局变量，这是一个序列数据类型，其中每一个元素都是对应的一个路由匹配规则

2. 如果在规则中查找到符合匹配规则的，则执行其中的对应执行函数
3. 如果对应的不是一个执行函数，而是一个`include`路由包含，那么截断与此项匹配的`URL`的部分，并将剩余的路由字符串发送到`include`所包含的子路由文件中以供进一步处理
4. 如果没有匹配到的任何结果，`django`默认抛出`Page not found (404)`

> **注意**：`Django`的路由不考虑HTTP请求方式，仅根据`URL`进行路由，即，只要`URL`相同，无论`POST`、`GET`等哪种请求方式都指向同一个操作函数

path

> `path`函数用来处理一个路由对应的视图映射

- `path(route, view, name)`

  > `route`： 匹配规则，是一个字符串
  >
  > `view`：对应的视图函数
  >
  > `name`：未来我们会用到他，用来为匹配规则命名，这样方便日后修改路由而不影响全局下的路由使用

re_path

> `re_path`是`path`函数的加强版
>
> 可以在`re_path`函数的第一个位置的字符串参数，是一个标准`Python`正则表达式，其余参数与`path`相同

> **注意**：匹配模式的最开头不需要添加`/`，因为默认情况下，每个`url`都带一个最前面的`/`，既然大家都有的部分，就不用浪费时间特别写一个了，所以一定要注意在写路由映射关系时，记得加末尾的`/`

模板页面

> 返回一个字符串这肯定是不行的，太`low`了，也不好看，现在来返回一个正式的`HTML`页面
>
> 并在`HTML`页面中加入模板变量，由视图函数动态传递值；

- 配置`django`中模板页面的保存路径，在项目目录下的`settings.py`文件中

```
#myproject/settings.py
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'template')], # 就是这一行 设置静态模板路径
        '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',
            ],
        },
    },
]
```

- `创建template`目录并在其中创建`index.html`文件

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>hi</title>
</head>
<body>
    <h1>{{ message }}</h1>
</body>
</html>
```

> 在`HTML`页面中，我们并没有明确指出`H1`标签的内容；通过一个`{{ message }}`来等待接收视图函数传来的数据，在`HTML`页面中这样的变量也叫做**模板变量**，双大括号为使用语法

- 接下来修改之前的视图函数，由视图函数传递变量给到`HTML`页面

```
#myapp/views.py
from django.shortcuts import render
def index(request):
    #return HttpResponse("<h1>Hello world</h1>")
    content = {
        "message":"你好，世界" #此处的key值message对应页面中我们写的{{ message }}
        }
    return render(request,'index.html',content)
```

render

> render函数用来返回一个模板页面，并将一个字典组合成的模板变量传递到模板页面上，完成页面的渲染

- `render(request, template_name, context=None)`

  > 返回一个HTTP响应

  > `request`： 固定接收`request`请求
  >
  > `template_name`： 为一个可以找到的模板页面
  >
  > `context`： 模板页面所需模板变量

模板变量

> 在`django`中的`HTML`页面，不光可以编写原本的标签等内容，还可以像`Vue`一样在页面中使用双大括号，来提前定义一些模板变量，之后动态的渲染到`HTML`模板页面中

> 模板变量可以由后台视图函数构建一个**字典数据类型**传递，
>
> 字典的`key`是模板变量名，`value`值该模板变量对应的数据
>
> 当然，模板变量的内容远不止此，还会再后面继续为大家叙述

静态文件

> 虽然有了模板页面，可以来展示一些标签的效果，但是整个HTML还是感觉很丑陋
>
> 我们还要继续引入一些类似`css、img`这样的静态资源，来装饰我们的页面

> 在`django`中模板页面的静态资源使用，不能像之前写`HTML`代码直接引入
>
> 需要我们首先在项目中创建目录保存对应的静态资源，该目录名常为`static`

- 在settings中配置静态文件保存目录，**添加**如下内容

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static'),
)
# STATICFILES_DIRS 该配置项用来告诉django在查找静态资源时，应该访问哪个目录
```

- 在项目中创建`static`目录，`static`目录下创建专门保存图片的`img`目录，在里面存一张图片`1.jpg`

```
#此时的目录结构
myproject/
    myproject/
    myapp/
    template/
    static/
        img/
            1.jpg
```

- 有了图片，接下来在模板页面中去引入并使用它，打开`index.html`进行修

```
<!DOCTYPE html>
<html lang="en">
    {% load staticfiles %}
<head>
    <meta charset="UTF-8">
    <title>hi</title>
</head>
<body>
    <h1>{{ message }}</h1>
    <img src='{% static "img/1.jpg" %}' alt="图片">
</body>
</html>
```

> 这里用到了一个特殊语法：`{% tag %}`这个叫静态标签，静态标签不同于模板变量，静态标签经常用来加载数据，或创建逻辑，比如之后我们要学到的`{% if %}`，使用静态标签可以方便我们在模板页面上实现某些只有在后台代码中才可以实现的逻辑功能

> 在页面中要引入静态资源：图片，`CSS`，`JS`文件在引入时都需要通过`{% static "path" %}`来进行引入
>
> 最后，需要使用静态标签`static`前使用`{% load staticfiles %}`标签进行静态资源路径的加载

模型数据库

> 有了以上内容的修饰，现在感觉还是缺少一些什么，我们在视图函数中为前端页面返回的是一个提前定义好的变量，这显然在真正开发中是很少出现的，我们的数据大都来自于数据库中，那么现在需要我们在项目中加入数据库，并且在视图函数中通过对数据库的访问来拿到数据

- 创建数据库，这里使用项目自带的`SQLite3`数据库，默认已经是配置好的，接下来需要我们进入到`app`下的`models.py`文件中，编写一个类，这个类就对应数据库中的一张表

```
#myapp/models.py
from django.db import models

# Create your models here.
class Weather(models.Model):
    weather = models.CharField(max_length=100,verbose_name="天气")
    class Meta:
        verbose_name_plural = "天气"
        # 设置当前表名的一个可读的性更好的名字
    def __str__(self):
        return self.weather
```

> 在这里我们使用到了`django`的`orm`映射关系用来创建数据库表，继承自`django`的`models.Model`类，
>
> **一个类用来表示一张表，类中的一个属性代表一个字段，**

> 这里我们定义了一个类型为`CharField`，长度为`100`的字段，用来存储天气
>
> `models.CharField(max_length=100,verbose_name="天气")`

> 下面的`class Meta`是模型类的元类，用来设置当前表的一些属性；
>
> 这里我们使用`verbose_name_plural`属性设置当前表在`admin`后台查看时的名字

> 在这里我们还定义了一个属于实例的函数`__str__`，用来描述当前数据在返回时的默认展示结果，为`weather`字段的值

> `django`在创建模型类对应的数据表时，默认使用`应用名`加`下划线`加`模型类名`作为表的名字；比如当前`Weather`表名为：`myapp_Weather`

> `orm`映射关系，是`django`与数据库之间的一个桥梁，可以使开发者不再关注如何去编写`SQL`语句，直接通过一套`ORM`所提供的`API`接口即可方便对各种数据库进行交互

- 当某个子应用`APP`涉及到了数据库的使用时，要记得在`settings`文件中进行配置

```
#myproject/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp',
]
```

- 接下来通过`manage.py`命令行管理工具提供的两条，创建我们所需要的数据

> **注意**：默认`django`本身就已经需要一些数据的创建，所以我们在初次执行以下两条命令时可能会看到很多数据表和字段的创建，不要惊讶，这是正常的

> `python manage.py migrate`：根据数据库迁移文件生成对应`SQL`语句并执行
>
> 初次执行是为了先把默认django需要的数据库创建出来

> `python manage.py makemigrations`：创建数据库迁移文件
>
> 这次执行是为了创建APP中Weather模型类的迁移文件

> `python manage.py migrate`
>
> 将新添加的模型类迁移文件生成对应`SQL`，实际创建出对应的`weather`表

- 如果提示结果正常，那么代表相应的数据表已经创建好了，接下来就需要我们去到`django`为我们提供的`admin`（数据库管理界面）来进行相关表的操作了！

admin控制台

> `admin`控制台是`django`为我们提供的一个非常便捷的用来管理数据库的界面
>
> 在主控路由文件下，其实你已经看到了它对应的路由设置：`path('admin/', admin.site.urls),`

> 进入`admin`界面，初次访问连接：`127.0.0.1/admin`，会提示我们输入账号密码，这是因为`django`的`admin`界面是需要一个超级管理员来登陆访问的，所以还需要我们创建对应的`admin`界面下的超级用户

- 创建`admin`超级用户，使用`manage.py`命令行工具执行如下命令

```
python manage.py createsuperuser
```

```
Username (leave blank to use 'lienze'): root
Email address:
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Password:
Password (again):
This password is too common.
This password is entirely numeric.
Password:
Password (again):
Superuser created successfully.
```

> 以上是我们创建超级用户的过程，非常坎坷；
>
> 可以看到，在输入太短（不满足8位），或是只包含数字的简单密码，超级用户的创建都是被拒绝的
>
> 所以我们把用户账号创建为`root`，而密码创建为`a1234567`，

- 接下来开启测试服务器，并通过创建好的超级用户登陆访问，如果幸运的话，你已经可以看到后台的`admin`界面啦

> admin界面已经展示出了默认django所使用的两张表，用户表和组表，用来保存当前管理后台的用户以及对应权限分组，可以点入用户表查看其中我们刚创建的root。

admin注册表

> 问题还是有的，虽然admin界面已经可以登入，但是为什么看不到刚才创建的weather表呢
>
> 这是因为默认的表创建之后，还需要通过对应app下的admin.py文件进行admin后台注册，只有注册在这个文件中的模型类对应的表才可以在admin界面所看到

- 在app下的admin.py文件中进行模型类的注册

```
#myapp/admin.py
from django.contrib import admin
from myapp import models

admin.site.register(models.Weather)
#使用register函数接收模型类作为参数即可完成注册
```

> 注册成功之后，在服务器，通过浏览器访问admin界面，就可以看到创建好的weather表了

- 鼠标点击进去之后，就可以看到对应的表数据界面；右上角提供了可以添加功能的选项，试试给这个表来一些数据吧，这里我们添加了三条数据

```
阴天，晴天，打雷了
```

视图操作模型

> 最终我们希望可以在视图函数中通过orm接口来访问到表中的数据，那么来打开视图文件吧：views.py

```
#myapp/views.py
from django.shortcuts import render
from myapp import models
def index(request):

    weathers = models.Weather.objects.all()
    content = {
        "weathers":weathers,
    }
    return render(request, 'index.html', content)
```

- 光返回是不行的，虽然我们绑定到了模板版变量的字典中，但是还得修改一下对应的要渲染的HTML页面哦：

```
<!DOCTYPE html>
<html lang="en">
    {% load staticfiles %}
<head>
    <meta charset="UTF-8">
    <title>hi</title>
</head>
<body>
```

```
    {% for weather in weathers %}
        <p>{{ weather }}</p>
    {% empty %}
        <p>没有任何天气</p>
    {% endfor %}
</body>
</html>
```

> 模板标签`{% for xxx in xxxs %}`可以用来在模板页面出迭代访问取出每一个数据
>
> 具体对于不同序列数据的访问我们会在后面详细为大家介绍
>
> `{% empty %}`标签用来判断当循环访问数据为空时要做的事情，最后循环标签要有`{% endfor %}`标签进行结束；因为`HTML`中并没有像`Python`缩进这样的方式来控制代码块。

总结

> 至此，我们的`HELLO WORLD`项目已经涵盖了`django`框架中的大部分常用的组件；
>
> **路由、视图、模板、静态、模型，admin**
>
> 那么其中每一部分都还有很多内容等着我们去了解，在接下来的章节中我们会继续详细给大家介绍！