视图层

> 在这一小节，将带大家深入了解**django**中关于视图函数的应用。
>
> 视图函数一般用来接收一个`Web`请求`HttpRequest`，之后返回一个Web响应`HttpResponse`

HttpRequest

> 一个视图函数用来响应用户的`Request`请求，每个视图函数默认的第一个位置参数`request`用来接收用户发起请求的`HttpRequest`信息。
>
> 视图函数的返回值，为一个`HttpResponse`值，包括我们要返回给用户的`HTML`页面或者字符串等等，以及对应的头部字段信息

```
from django.http import HttpResponse
def index(request):
    return HttpResponse('Hello world')
```

常见请求方式

> `POST`和`GET`是`HTTP`协议定义的与服务器交互的方法。
>
> `GET`一般用于获取/查询资源信息，而`POST`一般用于更新资源信息。另外，还有`PUT`和`DELETE`方法

get

> 常用来从指定地址请求数据；
>
> 如果需要在请求时提交某些数据，则以路由形式传递参数，查询`Query`字符串如下格式所示：

```
https://www.baidu.com/?key=abc&pos=shanxi
```

- `get`请求可被浏览器缓存，保存在历史记录中
- `get`不应在使用敏感数据时使用，明文包路在请求地址中
- `get`有长度限制

post

> 向指定的资源提交要被处理的数据
>
> 使用`POST`，提交的数据保存在`HTTP`协议中的消息主体部分

- `post`请求不会被浏览器缓存
- `post`提交数据长度无限制
- `post`比`get`更加安全

request

> 如果说`urls.py`是`Django`中前端页面和后台程序桥梁，那么`request`就是桥上负责运输的小汽车，可以说后端接收到的来至前端的信息几乎全部来自于`requests`中

request.method

> 获取当前用户请求方式，
>
> 请求方式字符串为纯大写：`'GET'`、`'POST'`
>
> 如用户以`get`方式发起请求，对应代码中获取到的结果以及在判断时像是这样

```
def index(request):
    if request.method == 'GET':
        …
```

request.GET

> 当用户通过`get`方式请求站点，并在路由中提供了查询参数，可以通过该属性获取到对应提交的值

```
def index(request):
    print(request.GET)
    # <QueryDict: {'name': ['jack'], 'id': ['1']}>
    print(type(request.GET))
    # <class 'django.http.request.QueryDict'>
    name_ = request.GET.get('name')
    id_ = request.GET.get('id')
    content = '%s:%s' % (name_,id_)
    return HttpResponse(content)
```

> `request.GET`是一个类似字典的数据类型：`QueryDict`
>
> 其中也支持类似对字典的`get`或直接`dict.[key]`键值访问方式，当然使用`get`方式进行对应`key`获取会更好，因为`get`在访问不到时不会报错

- 如果定义了如上所示的视图函数，那么在访问连接时，我们可以通过路由传参：

```
http://127.0.0.1:8000/?name=jack&id=1
```

- 这里对应页面会显示的结果：

```
jack:1
```

- > **注意**：使用`GET`方法在连接中进行参数提交，后台接收到的数据类型均是字符串

---

request.POST

> 获取用户以`post`形式提交的数据并保存在后台，为类字典数据，这里和`request.GET`是一个东西；
>
> 在网页中，一般我们通过`html`的表单进行数据的提交，`POST`方式可以提交空数据

- 因为涉及到了表单页面，所以我们先来弄一个`HTML`页面

```
<body>
    <div>这是一个关于POST的测试</div>
    <form action="/" method="POST">
        {% csrf_token %}
        账号:<input type="text" name="account">
        <br>
        密码:<input type="password" name="passwd">
        <input type="submit" value="提交">
    </form>
</body>
```

> 在模板页面中，一旦涉及到了表单提交，那么一定要注意在表单区域添加`{% csrf_token %}`标签进行防跨站伪造令牌的加载，否则表单数据的将被认为是无效的。

> 在接下来的视图函数中会使用到`input`标签中的`name`属性；
>
> `name`值属性维护了`post`的数据传入到后台时的标示，会与表单的数据组合成类字典格式
>
> 如`name`属性为`account`的输入框中输入了`test`，那么后台数据接收到的值类似：`{'account':'test'}`

- 写一个视图函数用来捕获当前表单使用POST形式提交的数据：

```
def index(request):
    if request.method="POST":
        print(request.POST)
        print(type(request.POST))
        account = request.POST.get("account")
        passwd = request.POST.get("passwd")
        content = "%s:%s" % (account,passwd)
        return HttpResponse(content)
    return render(request,"index.html") #在使用get形式请求时，返回表单页面
```

- 如果在表单页面中账号填写为test，密码为123456；在视图函数中捕捉到的结果为：

```
<QueryDict: {'csrfmiddlewaretoken':
['EmyGwsVcrXI2LDkYLS9qflkUH4N7bM1nfTQxr3fsOsZlI4vJFwci7TargtYRAGl2'], 'account': ['test'],
'passwd': ['123456']}>
```

表单多值提交

> 在`request.POST`中需要注意，某些情况下，使用POST提交数据的表单数据可能是多个值，类似复选框`CheckBox`，直接使用`request.POST.get()`进行获取是有一些问题的，比如修改模板页面如下所示

```
<form action="/" method="POST">
    {% csrf_token %}
    <input type="checkbox" name="taste" value="eat">吃
    <input type="checkbox" name="taste" value="sleep">睡
    <input type="checkbox" name="taste" value="play">耍
    <input type="submit" value="提交">
</form>
```

> 这是一个`name`值为`taste`的兴趣爱好采集的多选框，`value`值将会作为选中时，提交到后台的值，比如现在我们全选这些表单数据，那么后台接收到的值是这样的

```
<QueryDict: {'csrfmiddlewaretoken':
['nuaLzxc2EOartYKUZiefMPv5iHTX5gLFY1sCu8wi1vrKqpVFTWh7EnlCR64Hua5k'], 'taste': ['eat',
'sleep', 'play']}>
```

> 但是问题接踵而至，我们发现使用`get`函数获取不到对应全选的整个结果，而是只拿到了选中的最后一项

- `request.POST.get(key, default=None)`

  > 返回对应`key`值的数据中的**最后一个**数据单独返回；`key`值不存在，取`default`

> 要想真正拿出所有的结果，应该使用`getlist`函数

- `request.POST.getlist(key, default=None)`

  > 将对应`key`值的所有数据以**一个列表**形式返回；`key`值不存在，取`default`

request.META

> `request.MAT`E获取的是一个标准的`python`字典。它包含了所有的`HTTP`请求信息
>
> 比如用户IP地址和用户`Agent`（通常是浏览器的名称和版本号）。
>
> 注意，`Header`信息的完整列表取决于用户所发送的`Header`信息和服务器端设置的`Header`信息

- `CONTENT_LENGTH`：请求的正文的长度，字符串类型
- `CONTENT_TYPE`：请求的正文的`MIME`类型
- `HTTP_ACCEPT`：响应可接收的`Content-Type`
- `HTTP_ACCEPT_ENCODING`：响应可接收的编码
- `HTTP_ACCEPT_LANGUAGE`：响应可接收的语言
- `HTTP_HOST`：客服端发送的`HTTP Host`头部
- `HTTP_REFERER`：请求前的连接地址
- `HTTP_USER_AGENT`：客户端的`user-agent`字符串
- `QUERY_STRING`：单个字符串形式的查询字符串（未解析过的形式）
- `REMOTE_ADDR`：客户端的IP 地址
- `REMOTE_HOST`：客户端的主机名
- `REMOTE_USER`：服务器认证后的用户
- `REQUEST_METHOD`：一个字符串，例如`GET`或`POST`
- `SERVER_NAME`：服务器的主机名
- `SE0RVER_PORT`：服务器的端口，字符串类型

request.FILES

> 接收用户上传文件及相关信息。同样类似于`request.POST`，提取到的数据为一个类字典的数据类型，包含所有文件上传的信息

- `f = request.FILES.get('upload_file')`

  > `file_data = f.read()`：读取整个上传文件的内容，适合小文件上传

> `yiled = f.chunks()`：返回一个类似生成器`(<class 'generator'>)`的数据，每一次读取按块返回文件，可以通过`for`迭代访问其中数据；适合上传大文件到服务器。
>
> `f.multiple_chunks()`：返回文件大小，当文件大小大于`2.5M`时，返回`True`，反之返回`False`，可以通过该函数来选择是否使用`chunks`方法或`read`直接存储。
>
> 如果想要修改这个文件判定的默认值，可以通过：`FILE_UPLOAD_MAX_MEMORY_SIZE`在`settings`文件下进行设置
>
> `f.content_type`：上传文件时头部中的`Content-Type`字段值，参考MIME类型
>
> `f.name`：上传文件名字
>
> `f.charset`：上传文件编码
>
> `f.size`： 上传文件大小，字节为单位：`byte`

> 创建好静态资源目录，并在下面创建一个`img`文件夹，保存我们即将上传的图片；
>
> 完成上传文件的`HTML`表单页面

```
<form action="/" method="POST" enctype="multipart/form-data">
    {% csrf_token %}
    <input type="file" name="upload_file" />
    <input type="submit" value="提交">
</form>
<img src="{% static 'img/1.jpg' %}" alt="这是一张图片">
<!-- 这里使用的是即将要上传的文件名字，只做文件是否上传成功的简单测试 -->
```

> **注意**：上传文件的页面表单，一定要记得设置属性`enctype="multipart/form-data"`

- 视图函数如下编写，接收上传图片，并保存在静态目录下刚才创建好的img目录中

```
def index(request):
    if request.method == "POST":
        f = request.FILES.get("upload_files")
        path = os.path.join(settings.STATICFILES_DIRS[0],'img/'+f.name)
        # 上传文件本地保存路径
        with open(path,'wb') as fp:
            if f.multiple_chunks: #判断到上传文件为大于2.5MB的大文件
                for buf in f.chunks(): #迭代写入文件
                    fp.write(buf)
            else:
                fp.write(f.read())
            return HttpResponse("Success!")
    return render(request, 'index.html')
```

> 测试上传一个名为`1.jpg`的图片，如果成功上传，那么后台`static`目录下会出现该图片，并且模板页面也可以展示对应图片效果

HTTPResponse

> 一个视图的返回值经常是为了向用户返回一个`HttpResponse`响应，
>
> 有如下常用的可以返回`HttpResponse`的函数

response

- `HttpResponse(content=b'')`

  > 返回一个字符串内容
  >
  > *from* django.http *import* HttpResponse

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

  > 返回一个可渲染HTML页面，状态码为`200`
  >
  > *from* django.shortcuts *import* render

  > `request`：固定参数，响应的`request`请求，来自于参数部分接收的`HttpRequest`
  >
  > `template_name`：返回的模板页面路径
  >
  > `context`：模板页面渲染所需的数据，默认为字典格式
  >
  > `content_type`：生成之后的结果使用的`MIME`类型
  >
  > `status`：响应的状态码，默认为`200`

- `redirect(to, permanent=False)`

  > 一个重定向，浏览器通过该状态码自动跳转到一个新的路由地址，默认返回响应状态码`302`
  >
  > *from* django.shortcuts *import* redirect

  > `to`：可以是一个`django`项目中视图函数的路由映射，也可以是一个`reverse`的反向路由解析
  >
  > `permanent`：如果设置为`True`，将返回`301`状态码，代表永久重定向

  ```
  302：临时重定向，旧地址资源临时不能用了，搜索引擎只会暂时抓取新地址的内容而保存旧的地址。
  301：永久重定向，旧地址资源已经不复存在，搜索引擎不光会抓取新地址的内容，还会替换旧地址为新地址
  ```

视图错误处理

> 为了方便我们开发，`django`提供了一个异常叫做`Http404`异常，我们可以在视图函数的代码中按照需求进行抛出，抛出之后`django`项目会自动捕获该异常，并会展示默认的`404`页面

```
from django.http import Http404
def index(request):
    if request.GET.get("id") == "1":
        raise Http404
```

> 在`settings`中的`debug`配置项为`false`时，访问`http://127.0.0.1:8000/?id=1`，可以看到`django`为我们提供的错误页面；
>
> 除了`django`默认提供的，我们还可以可以在模板目录下定义全局`404.html`进行错误页面的定制

```
<h1>
    抱歉，找不到你要的东西
</h1>
```

自定义错误处理视图

> 除去`404`错误的自定义，`django`还提供了覆盖默认错误行为处理的办法；
>
> 有些时候，`django`自动的错误处理可能不能满足我们的需求，那么我们可以重新定义一些新的视图函数，
>
> 来覆盖掉`django`所提供的错误处理视图函数，最后在`urls.py`路由配置文件下通过定义全局变量来重新设置默认的错误处理视图函数

```
handler404: 覆盖page_not_found()视图。
handler500: 覆盖server_error()视图。
handler403: 覆盖permission_denied()视图。
handler400: 覆盖bad_request()视图
```

```
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include("viewapp.urls")),
]
handler404 = "viewapp.views.error_404"
# APP.模块.视图函数
handler500 = "viewapp.views.error_500"
```

> 相关定义好的错误处理视图函数

```
def error_404(request):
    return HttpResponse("这是404错误")

def error_403(request):
    return HttpResponse("这是403错误")

def error_500(request):
    return HttpResponse("这是500错误")
```