[笔记][Python Django 开发实战][2]

Django

06. 视图

每一个请求对应一个 `URL`，`URL` 映射到具体的视图函数（类），传递 `HttpRequest`，视图处理之后返回 `HttpResponse`。

06.1 视图初探

每一个视图都会接收一个请求，对请求进行自定义处理，最后返回一个响应，这就包含了一个视图的完整定义。对于项目的规范性而言，通常会将视图放置于项目或应用的 `views.py` 文件中。

查看 `post` 应用，可以看到应用目录下面已经有 `views.py` 文件了，这是 `Django` 在创建应用的时候自动创建的。`post/views.py`

```
from django.http import HttpResponse


def hello_django_bbs(request):
    html = '<h1>Hello Django BBS</h1>'
    return HttpResponse(html)
```

这段代码非常简单，只有四句话，但是已经包含了一个视图的完整功能了。下面依次对这四句话进行解释。
（1）引入 `HttpResponse`，作为视图的返回类型。
（2）视图函数声明，当前的函数名是 `hello_django_bbs`，它仅仅描述自身的用途，`Django` 对此不做要求。视图函数的第一个参数是 `HttpRequest` 类型的对象，且通常定义为 `request`，同样，`Django` 对此也没有要求，是一种约定俗成的习惯。

（3）函数内部定义业务处理逻辑，这里简单定义了视图的响应内容。
（4）视图最后返回一个 `HttpResponse` 对象，标识一次 `Web` 请求的结束。

将当前的视图与一个 `URL` 进行映射：
`my_bbs/urls.py`

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

from post import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('post/hello/', views.hello_django_bbs),
]
```

`urls.py` 文件中有一大段注释内容

```
"""my_bbs URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
```

这段注释介绍了 `Django` 中三种定义URL配置的方法

- 针对基于函数的视图：首先将视图定义文件引入（`import`），然后利用 `path` 定义 `URL` 和视图的对应关系，这与当前 `hello_django_bbs` 视图配置的形式是一样的。需要注意，当前给 `path` 函数传递了两个参数，第一个参数定义了 `URL` 的匹配规则，第二个参数定义了映射的视图函数，这两个参数都是必填项，在将来会看到 `path` 函数还可以接受其他的参数。
- 第二种，针对基于类的视图：首先将视图类对象引入，之后用类似的方法配置 `URL` 和视图类的对应关系。

```
from other_app.views import Home
```

```
    path('', Home.as_view(), name='home')
```

第三种，针对项目中存在多 `App` 的场景：利用 `include` 实现 `App` 与项目的解耦。`include` 将 `App` 的 `URL` 配置文件导入，之后就可以实现根据不同的 `App` 来分发请求，实现每个 `App` 自己管理自己的视图与 `URL` 映射，配置管理 `App` 的模式。

```
from django.urls import include, path

path('blog/', include('blog.urls'))
```

根据 `Django` 的提示，重新实现 `hello_django_bbs` 的映射关系，应该在 `post` 应用中添加一个 `URL` 配置文件，然后在项目的 `urls.py` 文件中引入。

> 团子注：`URL` 配置文件，就是 `URLConf`，也就是 `urls.py`。

`post/urls.py`

```
from django.urls import path

from post import view

urlpatterns = [
    path('hello/', views.hello_django_bbs),
]
```

同时，修改项目的 `urls.py` 文件：
`my_bbs/urls.py`

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('post/', include('post.urls')),
]
```

经过这样的配置，以 `post/` 开头的请求都会交给 `post` 应用的 `URL` 配置去处理，即实现了根据 `App` 来分发不同的请求。同时，这样的配置也更加清晰，便于维护，特别是当项目越来越大，应用越来越多时，这种方式会显得更加优雅。

视图的请求与响应对象
目前已经定义了一个视图，且执行了调用，在浏览器中看到了它的返回。这个过程中会涉及两个对象：`HttpRequest` 和 `HttpResponse`，即请求与响应对象。

1.HttpRequest
`HttpRequest` 对象定义于 `django/http/request.py` 文件中，每当请求到来的时候，`Django` 就会创建一个携带

有请求元数据的 `HttpRequest` 对象，传递给视图函数的第一个参数。视图函数中的处理逻辑就是根据这些元数据做出相应的动作。`HttpRequest` 定义了很多属性和方法：

（1）method

`method` 是一个字符串类型的值，标识请求所使用的 `HTTP` 方法，例如 `GET`、`POST`、`PUT` 等。这是最常用到的一个属性，在视图函数中用这个属性判断请求的类型，再给出对应的处理逻辑。所以，常常可以看到如下的视图定义：

```
if request.method == 'GET':
    get_something()
elif request.method == 'POST':
    post_something()
elif request.method == 'PUT':
    put_something()
```

之所以可以这样，是因为 `Django` 框架在路由分发时，不会考虑请求所使用的 `HTTP` 方法。也就是说，对于同一个 `URL`，不论是使用 `GET` 还是 `POST` 都会路由到同一个视图函数去处理。

例如，对于 `hello_django_bbs` 而言，给它添加 `@csrf_exempt` 装饰器，用 `POST` 方法请求对应的 `URL`，可以获得与 `GET` 请求同样的响应：

```
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt


csrf_exempt
def hello_django_bbs(request):
    html = '<h1>Hello Django BBS</h1>'
    return HttpResponse(html)
```

但有时候，可能需要明确只能接受特定的请求方法，利用 `@require_http_methods` 装饰器指定视图可以接受的方法。例如，只允许 `hello_django_bbs` 接受 `GET` 和 `POST` 请求，那么，可以这样配置：

```
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from django.views.decorators.http import require_http_methods


csrf_exempt
require_http_methods(['GET', 'POST'])
def hello_django_bbs(request):
    html = '<h1>Hello Django BBS</h1>'
    return HttpResponse(html)
```

当向这个视图发送GET、POST之外的方法时（如PUT），Django将会抛出405错误，并显示如下错误信息：

```
Method Not Allowed (PUT): /post/hello/
```

另外，为了简化使用，`Django` 提供了几个简单的装饰器指定可以接受的请求方法，它们定义于 `django/views/decorators/http.py` 文件中，如 `require_GET` 只允许 `GET` 方法、`require_POST` 只允许 `POST` 方法等。

> 团子注：`@require_GET` 相当于 `@require_http_methods(['GET'])`

（2）scheme

这是一个被 `@property` 装饰的方法，返回字符串类型的值。它可以被当作属性直接调用（`request.scheme`）。它用来标识请求的协议类型（`http` 或 `https`）。

（3）path

其为字符串类型，返回当前请求页面的路径，但是不包括协议类型（`scheme`）和域名。例如，对于请求http://127.0.0.1:8000/post/hello/，`path`返回的是 `/post/hello/`。

（4）GET

这是一个类字典对象，包含 `GET` 请求中的所有参数。
大多数 `HTTP` 请求都会携带有参数，例如，将之前访问的 `URL` 修改成http://127.0.0.1:8000/post/hello/?a=1&b=2&c=3，就可以通过 `GET` 属性获取到 `a`、`b`、`c` 三个参数了。`GET` **属性中的键和值都是字符串类型**，所以，可能需要对获取到的参数值进行类型转换。通常，获取参数的方法有两种。

- `request.GET['a']`：这样可以获取到参数 `a`，但是使用这种方式需要确保参数 `a` 是存在的，否则会抛出 `MultiValueDictKeyError` 错误。
- `request.GET.get('d',0)`：这种方式尝试从 `GET` 属性中获取属性 `d`，获取不到，则返回数字 `0`，与第一种方式相比更加安全。对于 `GET` 属性需要注意，它并不是 `Python` 中的字典类型，实际上它是一个 `QueryDict`（`django.http.QueryDict`）类型的实例，且它是只读的。如果需要修改它，可以通过 `copy` 方法获取它的副本，并在副本上执行修改。

  > 团子注：类字典类型

（5）POST

与 `GET` 属性类似，`POST` 属性中保存的是 `POST` 请求中提交的表单数据，同样，它也是一个 `QueryDict` 类型的实例对象。获取 `POST` 属性中参数的方式与操作 `GET` 属性是类似的，可以通过 `request.POST['a']` 或 `request.POST.get('d',0)` 的方式得到。需要注意的是，在使用 `POST` 方法上传文件时，文件相关的信息不会保存在 `POST` 中，而是保存在 `FILES` 属性中。

（6）FILES

这个属性只有在上传文件的时候才会用到，它也是一个类字典对象，包含所有的上传文件数据。
`FILES` 属性中的每个键是 `<input type="file" name="" />` 中的 `name` 值，`FILES` 中的每个值是一个 `UploadedFile`。

（7）COOKIES

它是一个 `Python` 字典（`dict`）对象，键和值都是字符串，包含了一次请求的 `Cookie` 信息。

（8）META

它也是一个 `Python` 字典对象，包含了所有的 `HTTP` 头部信息（具体可用的头部信息还需要依赖客户端和服务器）。

这里简单介绍一些常用的请求头。

- `CONTENT_LENGTH`：标识请求消息正文的长度，对于 `POST` 请求来说，这个请求头是必需的。
- `CONTENT_TYPE`：请求正文的 `MIME` 类型，对应于 `/post/hello/` 请求，它的值可以是 `text/plain`。
- `HTTP_HOST`：客户端发送的 `HTTP` 主机头。
- `HTTP_USER_AGENT`：通常被称为 `UA`，用于标识浏览器的类型，如果视图返回的数据是需要区分浏览器的，那么这个字段会非常有用。
- `REMOTE_ADDR`：客户端的 `IP` 地址，这个字段通常用于记录日志或根据 `IP` 确定地域再做处理。
- `REMOTE_HOST`：客户端的主机名。
- `REQUEST_METHOD`：标识 `HTTP` 请求方法，例如 `GET`、`POST` 等。
- `SERVER_NAME`：服务器的主机名。

> 团子注：`HTTP_HOST` 和 `REMOTE_HOST` 有什么区别？

- `SERVER_PORT`：服务器的端口号，用一个字符串标识，例如“8000”。

（9）user

标识当前登录用户的 `AUTH_USER_MODEL` 实例，它其实是 `Django` 用户系统中的 `User`（`auth.User`）类型。这个属性由 `AuthenticationMiddleware` 中间件完成设置，在用户未登录的情况下，即匿名访问，`user` 会被设置为 `AnonymousUser` 类型的实例。因为 `Web` 站点通常都会针对特定的用户提供服务（账户系统），所以，这个属性几乎在每个视图处理逻辑中都会用到。

2.HttpResponse

`HttpResponse` 对象定义于 `django/http/response.py` 文件中，在视图中主动创建并返回。下面介绍 `HttpResponse` 的属性、方法和它常用的子类。

（1）`status_code`

状态码（`status_code`）是 `HttpResponse` 最重要的属性，用来标识一次请求的状态。常见的状态码有 `200` 标识请求成功、`404` 标识请求的资源不存在、`500` 标识服务器内部错等。

（2）content

存储响应内容的二进制字符串。

（3）write方法

这个方法将 `HttpResponse` 视为类文件对象，可以向其中添加响应数据。例如，可以将视图函数 `hello_django_bbs` 修改为：

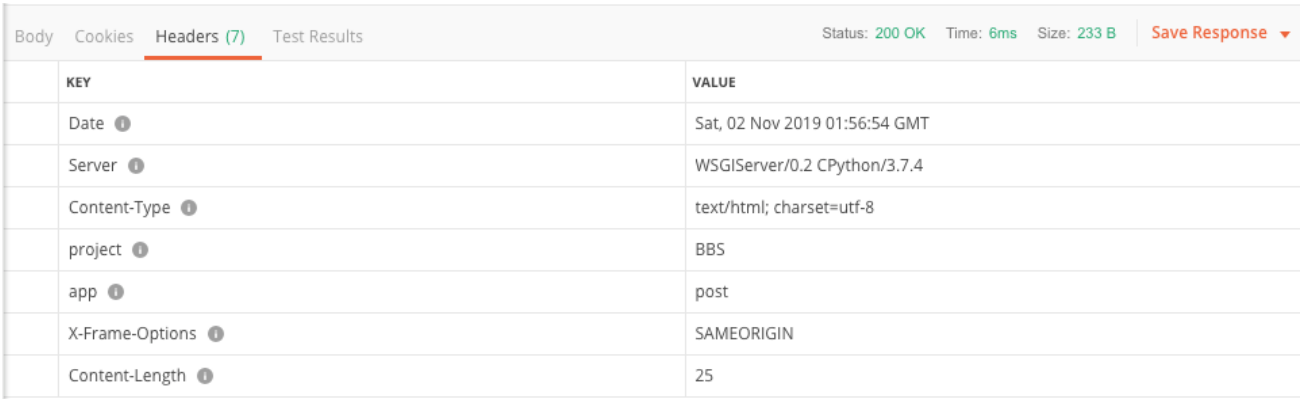
```
def hello_django_bbs(request):
    response = HttpResponse('<h1>Hello Django BBS</h1>')
    response.write('<h2>Hello Django BBS</h2>')
    response.write('<h3>Hello Django BBS</h3>')
    return response
```

（4）操作响应头

由于 `HttpResponse` 对象定义了 `__getitem__`、`__setitem__` 和 `__delitem__`，所以，可以**像操作字典一样**操作 `HttpResponse` 实例对象，且它控制的是响应头信息。

例如，可以在 `hello_django_bbs` 视图中给 `HttpResponse` 添加响应头：

```
csrf_exempt
require_http_methods(['GET', 'POST'])
def hello_django_bbs(request):
    html = '<h1>Hello Django BBS</h1>'
    response = HttpResponse(html)
    response['project'] = 'BBS'
    response['app'] = 'post'
    del response['Python']
    return response
```



重新访问视图，查看响应头已经有了 `project` 和 `app`，但同时注意到，视图代码中删除响应头的地方没有报错，这是 `__delitem__` 方法做的特殊处理，用 `try…catch` 将 `KeyError` 捕获了，所以，即使删除的头字段不存在，也不会抛出异常。

为简化开发过程，`Django` 提供了许多方便使用的 `HttpResponse` 子类，用来处理不同类型的 `HTTP` 响应。下面介绍常用的 `HttpResponse` 子类对象。

- `JsonResponse`。`JsonResponse` 是最常用的子类对象，用于创建 `JSON` 编码的响应值，定义于 `django/http/response.py` 文件中：

```
class JsonResponse(HttpResponse):
    def __init__(self, data, encoder=DjangoJSONEncoder, safe=True,
                 json_dumps_params=None, **kwargs):
        if safe and not isinstance(data, dict):
            raise TypeError(
                'In order to allow non-dict objects to be serialized set the '
                'safe parameter to False.'
            )
        if json_dumps_params is None:
            json_dumps_params = {}
        kwargs.setdefault('content_type', 'application/json')
        data = json.dumps(data, cls=encoder, **json_dumps_params)
        super().__init__(content=data, **kwargs)
```

> 团子注：默认只能序列化 `Python` 字典对象，如果要序列化其他对象，需要把 `safe=False`。

如源码中所示，`JsonResponse` 将响应头 `ContentType` 设置为 `application/json`，标识响应消息格式为 `JSON`。默认的 `JSON` 序列化器是 `DjangoJSONEncoder`，可以根据需要自行替换，但是，通常没有必要这样做。

`safe` 参数默认值为 `True`，指定 `data` 需要是字典类型，如果传递非字典类型的对象，则会抛出 `TypeError` 错误。

```
>>> from django.import JsonResponse
>>> response = JsonResponse({'project': 'BBS'})
>>> response.content
b'{"project": "BBS"}'
```

如果将safe设置为False，那么，data能接受任何可以JSON序列化的对象，例如

```
>>> response = JSONResponse([1, 2, 3], safe=False)
>>> response.content
b'[1, 2, 3]'
```

- `HttpResponseRedirect`。`HttpResponseRedirect` 用于实现响应重定向，返回的状态是 `302`。

> 团子注：`302` 是临时重定向。

它有一个必填的参数，用于指定重定向的 `URL` 地址。这个地址可以是一个特定的 `URL` 地址，例如 http://www.example.com；也可以是一个针对当前路径的相对路径地址，例如 `test/`，浏览器将会完成 `URL` 的拼接；还可以是一个绝对路径地址，例如 `/post/test`。

> 团子注：前面带 `/` 就是绝对路径地址，不带 `/` 的就是相对路径地址。

- `HttpResponseNotFound`。`HttpResponseNotFound` 继承自 `HttpResponse`，只是将状态码修改为 `404`。当请求的资源不存在时，可以使用这个响应对象。与之类似地，`Django` 还定义了 `HttpResponseBadRequest`（错误请求，状态码是400）、`HttpResponseForbidden`（禁止访问，状态码是403）、`HttpResponseServerError`（内部服务器错，状态码是500）等子类对象。

基于类的视图

类视图最大的特点是可以**利用不同的实例方法响应不同的HTTP请求方法**（GET、POST），且可以**利用面向对象**的技术将代码分解为可重用的组件。

一个简单的类视图定义

这里将 `hello_django_bbs` 利用类视图重新实现，同时，添加一个可以接受 `POST` 请求的方法，如下所示：

```
from django.http import HttpResponse
# 团子注：method_decorator 方法装饰器，可以让函数装饰器能够装饰方法
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt

from django.views import view


class FirstView(View):
    html = '(%s) Hello Django BBS'

    def get(self, request):
        return HttpResponse(self.html % 'GET')

    def post(self, request):
        return HttpResponse()

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(FirstView, self).dispatch(request, *args, **kwargs)
```

`FirstView` 继承自 `View`，它是所有基于类的视图的基类。其中定义了 `get` 和 `post` 方法，映射到 `GET` 和 `POST` 请求类型。另外，`FirstView` 重写了父类的 `dispatch` 方法，`dispatch` 根据 `HTTP` 类型实现请求分发。例如，如果是 `GET` 请求，则分发给 `get` 方法。**如果 `View` 中没有实现对应请求类型的方法，则会返回 `HttpResponseNotAllowed`**。类对象中定义的方法与普通的函数并不相同，所以，应用于函数的装饰器不能直接应用到类方法上，**需要将它转换为类方法的装饰器**。可以看到在 `dispatch` 方法上添加了 `@method_decorator` 装饰器，**这个装饰器可以将函数装饰器转换为类方法装饰器**，因此，`csrf_exempt` 装饰器可以被用在类方法上

了。Django 的 URL 解析器会将请求发送到一个函数而不是一个类，所以，需要用到 View 提供的 as_view 方法完成 URL 的定义。将 post 应用的 urlpatterns 修改为：

```
from django.urls import path

from post import views
from post.views import FirstView

urlpatterns = [
    path('hello/', views.hello_django_bbs),
    path('hello_class/', FirstView.as_view()),
]
```

对应当前类视图的请求 URL 为 http://127.0.0.1:8000/post/hello_class/。可以尝试发送 GET 或 POST 请求到当前的 URL，验证响应结果。但是如果发送 PUT 请求，Django 会提示不可访问的信息：MethodNotAllowed(PUT):/post/hello_class/。

设置类属性

在 FirstView 中定义了一个类属性 html，这是很常见的设计，将公用的部分放在类属性中，所有的方法都能看到。如果要设置类属性，那么有两种方法可以做到。第一种方法是 Python 的语言特性，实现子类并覆盖父类中的属性。例如，SecondView 继承自 FirstView，并重新定义了 html 属性：

```
class SecondView(FirstView):
    html = 'Second: (%s) Hello Django BBS'
```

第二种方法是直接在配置 URL 的时候在 as_view 方法中指定类属性，看起来更为简单直接：

```
path('second_hello_class/', FirstView.as_view(html='Second:(%s) Hello Django BBS'))
```

基于类的视图，每个请求到来的时候，Django 都会实例化一个类的实例，但是 as_view 中设置的类属性只在 URL 第一次导入时设置。

利用Mixin实现代码复用

一个 Mixin 就是一个 Python 对象，但是这个类不一定需要有明确的语义，其主要目的是实现代码的复用。Mixin 在表现形式上是多重继承，在运行期间，实现动态改变类的父类或类的方法。一个视图类可以继承多个 Mixin，但是只能继承一个 View，**写法上通常会把 Mixin 放在 View 的前面**。

例如，将 FirstView 中重写的 dispatch 方法放到 Mixin 里，同时，将第5章中定义的可以打印方法执行时间的装饰器也加入进去，可以这样实现：

```
import functools
import time

from django.http import HttpResponse
# 团子注：method_decorator 方法装饰器，可以让函数装饰器能够装饰方法
from django.utils.decorators import method_decorator
from django.views import View
from django.views.decorators.csrf import csrf_exempt


def hello_django_bbs(request):
    html = '<h1>Hello Django BBS</h1>'
```

```
    response = HttpResponse(html)
    response['project'] = 'BBS'
    response['app'] = 'post'


def exec_time(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        res = func(*args, **kwargs)
        print('%ss elapsed for %s' % (time.time() - start, func.__name__))
        return res
    return wrapper


class ExecTimeMixin(object):
    @method_decorator(csrf_exempt)
    @method_decorator(exec_time)
    def dispatch(self, request, *args, **kwargs):
        return super(ExecTimeMixin, self).dispatch(request, *args, **kwargs)


class FirstView(ExecTimeMixin, View):
    html = '(%s) Hello Django BBS'

    def get(self, request):
        return HttpResponse(self.html % 'GET')

    def post(self, request):
        return HttpResponse()
```

注意，不需要在 `FirstView` 中重写 `dispatch` 方法了。重新访问视图对应的 `URL`，可以看到会打印视图函数的执行时间。`Django` 自身也定义了一些 `Mixin` 简化开发，例如 `LoginRequiredMixin` 验证当前请求必须是登录用户，否则就会禁止访问或重定向到登录页。

基于函数的视图被称作 `FBV`（FunctionBasedViews），基于类的视图被称作 `CBV`（ClassBasedViews），`Django` 并没有对这两种实现视图的方式做出评价，所以，不存在哪一种方式更好的说法。

动态路由

因为视图函数也是普通的 `Python` 函数，所以，除了 `Django` 规定的第一个 `HttpRequest` 参数之外，还可以定义额外的参数。那么，对于视图函数中的其他参数，需要怎么给它们传值呢？这就引出了**动态路由**的概念，即 **`URL` 不是固定的，`URL` 中包含了传递给视图的参数变量**。相对于动态路由的概念，之前定义的视图映射的 `URL` 都可以被称作**静态路由**，即 **`URL` 是固定的**。同样，配置动态路由也需要用到 `path` 函数，只是 `URL` 配置的语法上有些不同。

1. 使用 `path` 配置动态路由

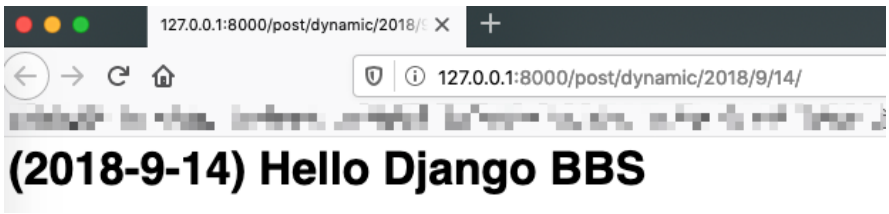
在分析它的用法之前，先简单地看一个接受其他参数的视图示例：

```
def dynamic_hello(request, year, month, day):
    html = '<h1>(%s) Hello Django BBS</h1>'
    return HttpResponse(html % ('%s-%s-%s' % (year, month, day)))
```

> 团子注：注意这里 `%` 的嵌套用法！第一次见。

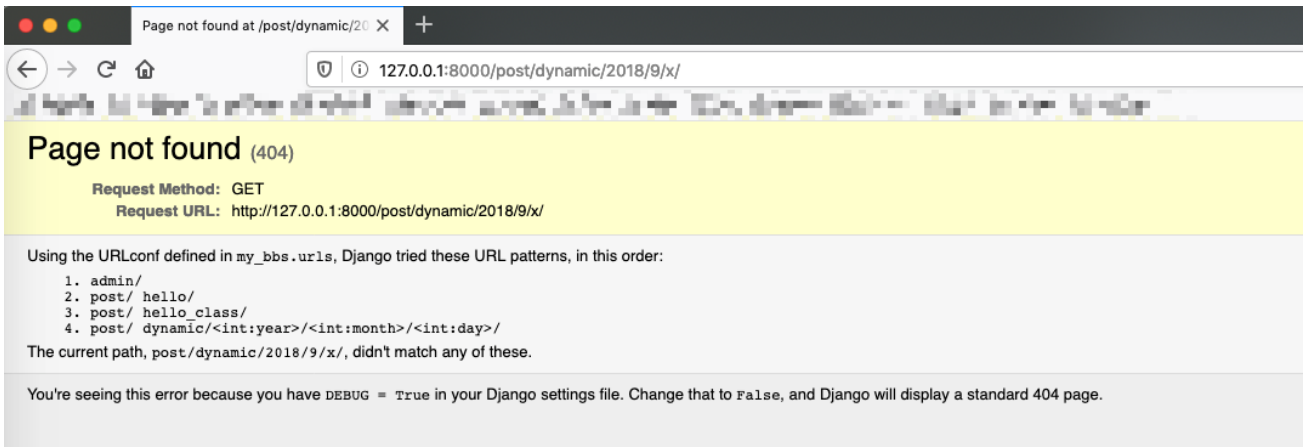
`dynamic_hello` 中除了第一个 `request` 参数之外，还定义了 `year`、`month` 和 `day` 三个参数标识年月日。因为这些参数有具体的含义，所以，它们也应该有具体的类型。
在 `post` 应用的 `urlpatterns` 加入如下路由定义：
`path('dynamic/<int:year>/<int:month>/<int:day>/', views.dynamic_hello)`

之后，在浏览器中输入 http://127.0.0.1:8000/post/dynamic/2018/9/14/，就可以看到带有打印日期的欢迎语了。



`path` 中定义的类似 `<int:year>` 规则会捕获到 `URL` 中的值，映射到视图中的同名参数 `year`，**并根据指定的转换器将参数值转换为对应的类型**，这里对应 `int`（**大于等于0的数**）。
之所以需要定义转换器，有两个原因：第一是可以将捕获到的字符值转换为对应的类型；第二是对 `URL` 中传值的一种限制，避免视图处理出错。

http://127.0.0.1:8000/post/dynamic/2018/9/x/



从图中可看到，错误页面的最下方已经给出了提示，当前处于 `DEBUG` 状态，所以，会在当前页面打印方便调试的信息。

除了 `int` 之外，`Django` 还提供了其他的转换器。
（1）`str`：匹配除了“/”（路径分隔符）之外的非空字符串，它是默认的转换器，即如果没有指定转换器，例如 `<day>`，那么，相当于 `<str:day>`。
（2）`slug`：匹配字母、数字、连字符和下画线组成的字符串。
（3）`uuid`：匹配格式化的 `UUID`（通用唯一识别码），并将捕获到的参数值转换为 `UUID` 实例对象。
（4）`path`：匹配任意的非空字符串，包含了路径分隔符。

2.自定义转换器
`Django` 内置的转换器都定义于 `django/urls/converters.py` 文件中，下面介绍 `int` 转换器的源码分析转换器的实现方法：

```
class IntConverter:
    regex = '[0-9]+'

    def to_python(self, value):
        return int(value)

    def to_url(self, value):
        return str(value)
```

它包含三个部分，也是每一个转换器都需要实现的三要素。
（1）`regex`：字符串类型的类属性，根据属性名可以猜测，这是一个正则表达式，用于匹配 `URL` 对应位置的参数值。
（2）`to_python`：参数 `value` 是从 `URL` 中匹配到的参数值，**通过强转成对应的类型传递给视图函数**。需要注意，这里可能因为强转抛出 `ValueError`。
（3）`to_url`：将一个 `Python` 类型的对象转换为字符串，`to_python` 的反向操作。

`dynamic_hello` 的 `URL` 配置虽然能实现将捕获到的参数值转换为 `int`，但是并没有做参数含义的校验。例如，访问http://127.0.0.1:8000/post/dynamic/2018/15/14/也没问题，但是，月份显然不能大于 `12`，所以，可以**自定义一个转换器捕获一个正确的月份**：

```
from django.urls.converters import IntConverter
class MonthConverter(IntConverter):
    regex = '0?[1-9]|1[0-2]'
```

由于月份也是数字，所以，只需要继承自 `IntConverter`，并重新定义 `regex` 规定匹配规则就可以了。

定义好了转换器，还需要完成注册才能使用，在 `post` 应用的 `urls.py` 文件中注册：

```
from django.urls import register_converter
from post.views import MonthConverter
register_converter(MonthConverter, 'mth')
```

可以看到，这里将转换器的类型名设定为 `mth`，接着，重新设定 `dynamic_hello` 视图的 `URL`：

```
path('dynamic/<int:year>/<mth:month>/<int:day>/', views.dynamic_hello)
```

此时，再去访问 `dynamic_hello`，`month` 字段就只能传递 `1~12` 的数字了。否则，会返回 `404` 响应。同样，对于 `year` 和 `day` 也可以自定义对应的转换器，限制数值范围。

3.带默认参数的视图

由于视图是普通的 `Python` 函数，所以，视图中的参数也可以带有默认值。例如，给 `dynamic_hello` 中的 `day` 添加默认值：

```
def dynamic_hello(request, year, month, day=15):
    html = '<h1>(%s) Hello Django BBS</h1>'
    return HttpResponse(html % ('%s-%s-%s' % (year, month, day)))
```

同时，给视图配置两个 `URL` 模式：

```
path('dynamic/<int:year>/<mth:month>/', views.dynamic_hello),
path('dynamic/<int:year>/<mth:month>/<int:day>/', views.dynamic_hello),
```

那么，此时，类似http://127.0.0.1:8000/post/dynamic/2018/12/的请求也可以访问到 `dynamic_hello`，对应到第一个 `URL` 模式（不含 `day`），且 `day` 直接使用默认值 `15`。

4.正则表达式
`Python` 的正则表达式中**命名分组**的语法为：`(?P<name>pattern)`，其中 `name` 是分组名，`pattern` 是匹配模式。引用分组可以使用分组名，也可以使用分组编号。

```
>>> import re
>>> r = re.compile('(?P<year>[0-9]{4})')
>>> s = r.search('2018')
>>> s.group('year')
'2018'
```

`path` 方法定义于 `django/urls/conf.py` 文件中，同时，这个文件中还定义了一个与它很像的方法：`re_path`，可以使用命名分组来定义 `URL`。

使用 `re_path` 的理由是 `path` 方法和转换器都不能满足需求。

```
re_path('re_dynamic/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/', views.dynamic_hello),
```

这样可以实现与之前 `path` 结合转换器类似的匹配功能，但是有两点需要注意。
（1）`year` 命名分组匹配四位整数，`int` 转换器可以匹配任意大于等于0的整数，所以，两者之间的匹配范围不同（`month` 和 `day` 也有同样的特性）。
（2）命名分组传递到视图中的参数是**字符串类型**，需要根据视图逻辑自行调整。

给 `post` 应用添加视图
视图中的代码应该尽量简洁，**复杂的业务逻辑不应该出现在视图中**，因此，通常会把逻辑或者 `service` 部分单独放到一个文件中，如 `post_service.py`。

1.实现Topic列表视图

由于还没有讲解到模板，所以，这里暂时不把视图的结果渲染出来，而是采用返回 `JSON` 格式数据的方式，即返回 `JsonResponse`。

首先，**创建存储业务处理逻辑的文件** `post/post_service.py`（位于 `post` 应用下），并实现构造 `Topic` 实例对象基本信息的业务逻辑：

```
def build_topic_base_info(topic):
    """
    构造 Topic 基本信息
    :param topic:
    :return:
    """
    return {
        'id': topic.id,
        'title': topic.title,
        'user': topic.user.username,
        'created_time': topic.created_time.strftime('%Y-%m-%d %H:%M:%S')
    }
```

`build_topic_base_info` 仅返回一个 `Topic` 实例的基本信息，这些信息用于展示 `Topic` 列表已经够用了。但是，需要注意，这样的实现其实并不安全，因为并没有对传递进来的参数进行校验，例如，它可能不是一个 `Topic` 类型的实例对象，可能是 `None`（空对象）等。

接下来，在 `post/views.py` 文件中定义 `Topic` 列表信息的视图函数：

```
from post.models import Topic
from post.post_service import build_topic_base_info

def topic_list_view(request):
    """
    话题列表
    :param request:
    :return:
    """
    topic_qs = Topic.objects.all()
    result = {
        'count': topic_qs.count(),
        'info': [build_topic_base_info(topic) for topic in topic_qs]
    }

    return JsonResponse(result)
```

这个视图实现的功能非常简单：先获取当前所有的 `Topic` 实例对象，之后返回数量和每一个 `Topic` 对象的基本信息。

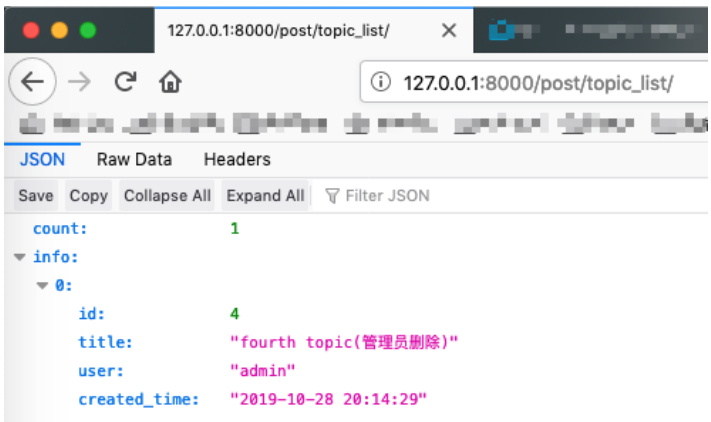
需要注意，当前的视图也仅实现了基本的功能，还有很多不完善的地方，例如没有过滤已经被删除的 `Topic`（`is_online` 为 `False`）、没有对 `Topic` 列表进行分页等。

做实际的业务处理逻辑可能需要思考更多的问题，更细致化地实现功能。本书中给出的示例程序更加专注于介绍实现方法和过程，因此不会考虑太多“可能出现的问题”。

最后，一个完整的视图还需要 `URL` 配置，在 `post/urls.py` 文件中添加：

```
path('topic_list/', views.topic_list_view),
```

到此，`Topic` 列表视图就已经完成了。打开http://127.0.0.1:8000/post/topic_list/可以看到 `JSON` 格式的 `Topic` 列表数据。



2.实现Topic实例对象信息视图

有了 `Topic` 列表信息，再根据列表信息中提供的 `id`（Topic主键字段）查看每一个 `Topic` 的详细信息就很简单了。
首先，每一个 `Topic` 都有可能会有多个 `Comment`，展示一个 `Topic` 的详细信息，应该把 `Comment` 也展示出来，所以，需要一个构造 `Comment` 信息的方法：

```
def build_comment_info(comment):
```

```
    """
    构造 Comment 信息
    :param comment:
    :return:
    """
    return {
        'id': comment.id,
        'content': comment.content,
        'up': comment.up,
        'down': comment.down,
        'created_time': comment.created_time.strftime('%Y-%m-%d %H:%M:%S'),
        'last_modified': comment.last_modified.strftime('%Y-%m-%d %H:%M:%S'),
    }
```

接下来，定义构造 `Topic` 详细信息的方法：

```
def build_topic_detail_info(topic):
    """
    构造 Topic 详细信息
    :param topic:
    :return:
    """
    comment_qs = Comment.objects.filter(topic=topic)
    return {
        'id': topic.id,
        'title': topic.title,
        'content': topic.content,
        'user': topic.user.username,
        'created_time': topic.created_time.strftime('%Y-%m-%d %H:%M:%S'),
        'last_modified': topic.last_modified.strftime('%Y-%m-%d %H:%M:%S'),
        'comments': [build_comment_info(comment) for comment in comment_qs],
    }
```

除了 `Topic` 自身的基本信息之外，这里还利用 `build_comment_info` 将它所对应的所有 `Comment` 的信息展示出来。

之后，在 `post/views.py` 文件中完成 `Topic` 详细信息的视图：

```
from post.models import Topic
from post.post_service import build_topic_detail_info

def topic_detail_view(request, topic_id):
    """
    话题详细信息
    :param request:
    :param topic_id:
    :return:
    """
    result = {}
    try:
        result = build_topic_detail_info(Topic.objects.get(pk=topic_id))
    except Topic.DoesNotExist:
        pass
    return JsonResponse(result)
```

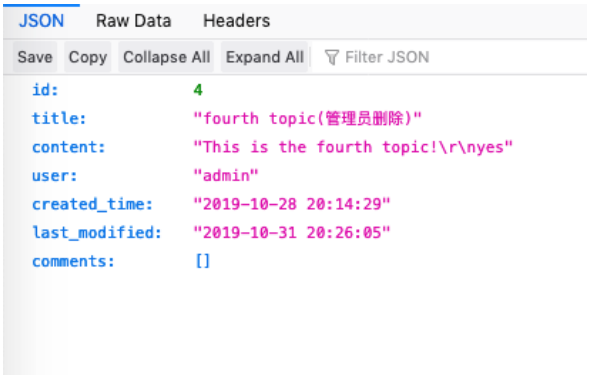
> 团子注：业务逻辑实际上应该是 `business logic`。

首先根据传递的 `topic_id` 通过 `get` 方法获取到 `Topic` 实例对象，注意，可能会抛出 `DoesNotExist` 异常，代表实例不存在；之后，传递给构造详细信息的方法，获取结果字典；最后，返回 `JsonResponse`。

在 `post/urls.py` 文件中添加如下 `URL` 模式：

```
path('topic/<int:topic_id>/', views.topic_detail_view),
```

`topic_id` 字段带有 `int` 转换器，所以，不能传递一个小于 `0` 或者非整数的值，这也符合id字段的基本要求。打开http://127.0.0.1:8000/post/topic/4/



3.给Topic实例对象添加评论的视图

用户在 `BBS` 站点上看到 `Topic` 的信息，给 `Topic` 添加评论也是一个很常见的需求。首先，还是先写出添加评论的业务逻辑。

```
def add_comment_to_topic(topic, comment):
    """
    给话题添加评论
    :param topic:
    :param content:
    :return:
    """
    return Comment.objects.create(topic=topic, content=content)
```

> 团子注：`create` 之后会返回创建出来的那个对象。

这里简单地根据传递的 `Topic` 实例对象和评论内容（`content`）创建了 `Comment` 实例对象。注意，`up` 和 `down` 可以使用默认值，不需要指定。
接下来，编写视图函数：

```
csrf_exempt
def add_comment_to_topic_view(request):
    """
    给话题添加评论
    :param request:
    :return:
    """
```

```
    topic_id = int(request.POST.get('id', 0))
    content = request.POST.get('content', '')
    topic = None
    try:
        topic = Topic.objects.get(pk=topic_id)
    except Topic.DoesNotExist:
        pass
    if topic and content:
        return JsonResponse({
            'id': add_comment_to_topic_view(topic, content).id
        })

    return JsonResponse({})
```

`GET` 方法通常用于从服务器中获取一些资源，而 `POST` 方法用来通过表单向服务器提交一些资源，所以，当前方法的请求类型是 `POST`。视图内部对传递的参数做了简单的校验，并调用 `add_comment_to_topic` 方法创建了一条评论。之后，返回 `Comment` 对象的主键字段。

最后，在 `post/urls.py` 文件中添加 `URL` 模式：

```
path('topic_comment/', views.add_comment_to_topic_view),
```

目前，已经给 `post` 应用添加了三个视图：`Topic` 列表、`Topic` 详细信息、给 `Topic` 添加评论，这也是最常用的功能。当然，可以根据自己的需要对应用功能进行扩展，例如，给评论添加赞同（`up`）和反对（`down`）的功能等，以进一步丰富应用。

06.2 视图的高级特性和快捷方法

URL的反向解析

在处理业务需求的过程中可能会需要视图的 `URL` 模式，如返回重定向或在模板中用于链接到其他的视图。但是，由于 `URL` 可能随着业务调整发生变化，因此**将 `URL` 硬编码到代码中并不友好**。`Django` 为了解决这个问题，提供了 `URL` 反向解析的方法，通过给 `URL` 模式命名即可反向解析得到完整的 `URL`。

1.reverse方法

之前已经看到在配置 `URL` 模式的时候使用了 `path` 方法，且传递了两个参数：`URL` 模式和视图。除了这两个最低配置之外，`path` 还可以接受一个 `name` 参数，用于指定当前 `URL` 模式的名字。`URL` 的反向解析就可以利用这个指定的 `name` 去完成。

给之前定义的 `dynamic_hello` 视图的 `URL` 模式添加 `name` 参数，如下所示：

```
path('dynamic/<int:year>/<mth:month>/<int:day>/', views.dynamic_hello, name='dynamic_hel
lo'),
```

能够实现反向解析的 `reverse` 方法的定义
`reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)`
除了第一个 `viewname` 是必填参数之外，其他参数都是可选的。

（1）`viewname`：它可以是 `URL` 模式的名字，即 `name` 所指定的名称；也可以是可调用的视图对象，例如 `dynamic_hello` 视图。

> 团子注：可以是 `视图类.as_view()` 吗？

（2） `urlconf`：这个属性用于决定当前的反向解析使用哪个 `URLconf` 模块，默认是根 `URLconf`。
（3） `args`：它用于传递参数，可以是**元组或者列表**，顺序填充 `URL` 中的位置参数。
（4） `kwargs`：它与 `args` 一样，也用于传递参数；但它是字典类型的，使用关键字指定参数和数值。需要注意，**`args` 与 `kwargs` 不可以同时出现。**

> 团子注：为什么后面讲 `RedirectView` 的时候，源代码中出现了同时指定 `args` 和 `kwargs` 的情况呢？官方文档中跟这里说的一样，的确是不能同时传递。

（5） `current_app`：**它指示当前执行的视图所属的应用程序。**

可以尝试在项目的 `Shell` 环境中使用 `reverse` 方法反向获取 `URL`，例如可以通过传递 `name` 指定的名称：

```
>>> from django.urls import reverse
>>> reverse('dynamic_hello', args=(2018, 9, 16))
'/post/dynamic/2018/9/16/'
```

可以看到，`args` 中的参数按照顺序依次安装到 `URL` 中的特定位置。同样，可以使用字典类型的 `kwargs` 指定参数：

```
>>> reverse('dynamic_hello', kwargs={'year': 2018, 'day': 16, 'month': 9})
'/post/dynamic/2018/9/16/'
```

除了可以传递 `name`，也可以向 `reverse` 中传递视图对象：

```
>>> from post.views import dynamic_hello
>>> reverse(dynamic_hello, kwargs={'year': 2018, 'day': 16, 'month': 9})
'/post/dynamic/2018/9/16/'
```

> 团子注：我来试试 `as_view`，失败！
> 问题：如何根据类名来反向解析 `url`？

```
>>> reverse(FirstView.as_view())
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/jpch89/code/bbs_python37/my_bbs/venv/lib/python3.7/site-packages/django/urls/base.py", line 90, in reverse
    return iri_to_uri(resolver._reverse_with_prefix(view, prefix, *args, **kwargs))
  File "/Users/jpch89/code/bbs_python37/my_bbs/venv/lib/python3.7/site-packages/django/urls/resolvers.py", line 636, in _reverse_with_prefix
    raise NoReverseMatch(msg)
django.urls.exceptions.NoReverseMatch: Reverse for 'post.views.FirstView' not found. 'post.views.FirstView' is not a valid view function or pattern name.
```

`reverse` 方法常常用于视图的重定向，例如，定义如下视图：

```
def dynamic_hello_reverse(request):
    return HttpResponseRedirect(
```

```
            reverse('dynamic_hello', args=(2018, 9, 16))
        )
```

可以给这个视图配置 `URL` 并做个试验，当请求到 `dynamic_hello_reverse` 视图之后浏览器就会重定向到http://127.0.0.1:8000/post/dynamic/2018/9/16/。

```
    path('dynamic_hello_reverse/', views.dynamic_hello_reverse),
```

`Django` 项目可以包含多个应用（App），每一个应用都可能会定义很多视图，从而会有很多 `URL` 模式，那么，`URL` 命名冲突就是很常见的事了。为了解决这个问题，需要学习命名空间的概念。

2.命名空间

`URL` 命名空间使得即使在不同的应用中定义了相同的 `URL` 名称，也能够反向解析到正确的 `URL`。

`URL` 命名空间分为两部分：**应用命名空间**（`app_name`）和**实例命名空间**（`namespace`）。
（1）`app_name`：`Django` 对它的解释是**正在部署的应用名称**，**应用的每一个实例都有相同的命名空间**。所以，可以通过组合 `app_name` 和 `name`（URL名称）区分不同应用的URL，它们之间使用“`:`”连接。

（2）`namespace`：它用来标识**一个应用的特定实例**，主要功能是**区分同一个应用部署的多个不同实例**。

为了更好地理解命名空间的概念，这里将之前的示例进行改造。首先，在项目的 `urls.py` 文件中给 `post` 应用添加 `namespace`：
`my_bbs/urls.py`

```
    path('post/', include('post.urls', namespace='bbs_post')),
```

`post/urls.py`

```
    app_name = 'post'
```

最后，需要修改 `dynamic_hello_reverse` 视图的定义：

```
def dynamic_hello_reverse(request):
    return HttpResponseRedirect(
        reverse('post:dynamic_hello', args=(2018, 9, 16), current_app=request.resolver_m
atch.namespace)
    )
```

> 团子注：`current_app=request.resolver_match.namespace`

这样设置了命名空间之后，即使是不同的应用存在同名的 `URL` 也不会出现问题了。

`URL` 反向解析不仅可以用在视图的重定向中，还可以用在模板中。`Django` 的模板语言中，使用 `URL` 模板标签同样可以通过 `URL` 的 `name` 获取到 `URL` 地址（可以结合命名空间），这将在模板一节中介绍。

视图重定向
之前已经介绍过使用 `HttpResponseRedirect` 完成视图的重定向，除此之外，`Django` 在 `django/shortcuts.py`

文件中提供了重定向的快捷方法 `redirect`。它比 `HttpResponseRedirect` 更加强大，除了传递 `URL`，还可以接受对象和视图完成重定向。接下来介绍这个方法的相关特性。

> 团子注：潜台词是，`HttpResponseRedirect` 只能接受 `URL`，也就是说只能写死 `URL` 或者配合 `reverse` 来反向解析。

1.永久重定向和临时重定向

`HttpResponseRedirect` 响应的状态码（`status_code`）是 `302`，它也被称为临时重定向。同时，还可以使用 `HttpResponsePermanentRedirect` 完成重定向，它与 `HttpResponseRedirect` 唯一的区别是状态码为 `301`，其被称为永久重定向。

`HTTP` 对 `301` 的解释是**被请求的资源已经永久移动到新的位置，将来任何对此资源的引用都应该使用本响应返回的若干个URI之一，它最常用到的场景是域名跳转。**

`302` 表示**当前请求的资源临时从不同的 `URI` 响应**，常用**在同一个站点内的跳转**，如未登录用户访问页面重定向到登录页。

2.redirect方法

在介绍redirect的用法之前，先来看它的定义：

`redirect(to, *args, permanent=False, **kwargs)`

`*args` 和 `**kwargs` 最终会传递到 `reverse` 方法中，所以，它们是用来标注 `URL` 的参数。`permanent` 默认为 `False`，实现 `302` 重定向（`HttpResponseRedirect`），如果设置为 `True`，则是 `301` 重定向（`HttpResponsePermanentRedirect`）。

`redirect` 方法只有一个必填参数 `to`，它可以接受三种类型的参数。

（1）带有 `get_absolute_url` 属性（方法）的对象，例如，可以定义如下视图：

```
def hello_redirect(request):
    class A:
        @classmethod
        def get_absolute_url(cls):
            return '/post/topic_list/'
    return redirect(A)
```

> 团子注：也就是 `redirect` 会自动调用 `A.get_absolute_url()`。

此时访问 `hello_redirect` 视图将跳转到话题列表页。当然，这只是一个示例，更常见的用法是在 `Model` 中定义 `get_absolute_url` 方法，传递 `Model` 实例对象到 `redirect` 方法中，跳转到 `Model` 实例的详细信息页。例如，可以给 `Topic` 对象添加 `get_absolute_url` 方法：

```
class Topic(BaseModel):
    """
    BBS 论坛发布的话题
    """
    title = models.CharField(max_length=255, unique=True, help_text='话题标题')
    content = models.TextField(help_text=u'话题内容')
    is_online = models.BooleanField(default=True, help_text=u'话题是否在线')
    user = models.ForeignKey(to=User, to_field='id', on_delete=models.CASCADE, help_text
=u'关联用户表')
```

```
    class Meta:
        verbose_name = u'话题'
        verbose_name_plural = u'话题'

    def __str__(self):
        return '%d: %s' % (self.id, self.title[:20])

    def get_absolute_url(self):
        return '/post/topic/%s/' % self.id
```

（2）传递 `URL` 模式的名称，即 `path` 中配置的 `name` 值（如果配置了命名空间，也需要指定）。内部实现是通过 `reverse` 方法反向解析得到 `URL`。例如：

```
def hello_redirect(request):
    return redirect('post:dynamic_hello', 2018, 9, 16)
```

（3）传递绝对或相对 `URL`，即直接指定需要跳转的位置。例如，可以指定重定向到话题列表页：

```
def hello_redirect(request):
    return redirect('/post/topic_list/')
```

相对 `URL` 传递可以使用 `./` 和 `../` 的形式，与文件系统路径类似。例如：

```
def hello_redirect(request):
    return redirect('./xxx/')
```

> 团子注：相对 `URL` 还是第一次见。那我之前的关于绝对和相对的理解错了吗？之前的理解是 `/` 开头的是绝对，不带 `/` 的是相对。

假设对于当前的视图 `URL` 是 `/post/hello_redirect/`，那么，当访问它时，会重定向到 `/post/hello_redirect/xxx/`。对于第二种形式：

```
def hello_redirect(request):
    return redirect('../topic_list/')
```

它会回到当前 `URL` 的“上一层”，再与传递的相对路径拼接，最终得到话题列表的 `URL`：`/post/topic_list/`。

常用的快捷方法

1.render方法

这个方法将给定的模板和上下文字典组合，渲染返回一个 `HttpResponse` 对象。首先，看这个方法的定义：

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

除了前两个是必需的之外，其他都是可选的。

（1） `request`： `HttpRequest` 对象，即视图函数的第一个参数。

（2） `template_name`：可以是字符串或字符串列表。**字符串代表模板的完整路径；如果是列表，则按顺序找到第一个存在的模板。**

> 团子注：居然还能是字符串列表。。。

（3）`context`：默认是一个空字典，可以通过传递它渲染模板。

（4）`content_type`：生成文档的 `MIME` 类型，默认为 `DEFAULT_CONTENT_TYPE` 设定的值。

（5）`status`：响应状态码，默认为 `200`。

（6）`using`：用于指定加载模板的模板引擎。

可以将话题列表页修改为：

```
def topic_list_view(request):
    """
    话题列表
    :params request:
    :return:
    """
    topic_qs = Topic.objects.all()
    result = {
        'count': topic_qs.count(),
        'info': [build_topic_base_info(topic) for topic in topic_qs]
    }
    return render(request, 'post/topic_list.html', result)
```

这样，话题列表页的展示就会使用 `result` 字典渲染 `post/topic_list.html` 模板（模板相关的内容将在以后介绍）

2.render_to_response方法

这个方法的功能与 `render` 是一样的，根据一个给定的上下文字典渲染模板并返回 `HttpResponse`。方法定义如下：

```
render_to_response(template_name, context=None, content_type=None, status=None, using=No
ne)
```

从定义中可以看出，它与 `render` 方法的**区别是不需要传递 `request` 参数**，其他都是相同的。由于没有传递 `request`，因此其在模板中的使用会受到一定的限制，如不能直接通过 `request` 对象获取它的相关属性。所以，如果需要在模板中使用 `request`，应该使用 `render`，而不是 `render_to_response`。另外，`Django` 在源码中给出了提示，**这个方法可能在将来被废弃**，所以，在应用开发中首先考虑使用 `render` 方法。

3.get_object_or_404方法

这个方法通过 `Model` 对象的 `get` 方法获取实例对象，但是当实例不存在的时候，它会捕获 `Does NotExist` 异常，并返回 `404` 响应。同样，看一看它的定义：

```
get_object_or_404(klass, *args, **kwargs)
```

其中 `*args` 和 `**kwargs` 是查询对象时用到的查询参数，且应该是 `get` 和 `filter` 可以接受的格式。`klass` 可以是 `Model` 对象、`Manager` 或 `QuerySet` 实例。

> 团子注：可以是模型管理器的原因是，模型管理器拷贝了大部分的查询集的方法。

需要注意，虽然 `get_object_or_404` 不会抛出 `DoesNotExist` 异常，但是如果传递的条件匹配了多个实例对象，则仍然会抛出 `MultipleObjectsReturned` 异常。

> 团子注：本质上 `get_object_or_404` 还是 `get` 方法，所以不能返回多个实例。

接下来，以 `Topic` 对象为例，看它的几种使用方法。首先，传递 `Topic` 对象：

```
>>> from post.models import Topic
>>> from django.shortcuts import get_object_or_404
>>> get_object_or_404(Topic, pk=4)
<Topic: 4: fourth topic(管理员删除)>
```

通过 `Topic` 对象并指定主键值获取了 `id` 为 `4` 的实例对象。这里需要理解，这个方法在正常执行时返回的是 `Model` 实例对象，并不是 `HttpResponse`。
还可以给它传递 `Manager` 实例：

```
>>> get_object_or_404(Topic.objects, pk=4)
<Topic: 4: fourth topic(管理员删除)>
```

由于可以传递 `Manager` 实例，所以，也可以这样应用：

```
>>> topic = Topic.objects.get(pk=1)
>>> get_object_or_404(topic.comment_set, pk=1)
<Comment: 1: very good!>
```

如果已经从其他地方获取了 `Topic` 的 `QuerySet` 实例，那么，直接把 `QuerySet` 传递到方法中是非常方便的：

```
>>> topic_qs = Topic.objects.filter(id__gte=1)
>>> get_object_or_404(topic_qs, pk=1)
<Topic: 1: first topic>
```

以上的这些都是“正常”情况，即可以根据传递的条件唯一地获取到 `Topic` 实例对象。但是，如果传递的参数匹配不了，例如：

```
>>> get_object_or_404(Topic, pk=10)
```

则会返回 `404`：

```
django.http.response.Http404: No Topic matches the given query.
```

4.get_list_or_404方法
根据这个方法的名字可以知道，它是用来获取 `Model` 实例对象的列表，当获取的结果为空时，返回 `404` 响应。方法的定义如下：
`get_list_or_404(klass, *args, **kwargs)`

可以看到，它接受的参数与 `get_object_or_404` 是一样的，**只是在做匹配时使用 `filter` 方法而不是 `get` 方法。**

使用 `get_list_or_404` 最简单的形式是只传递 `Model` 对象，例如：

```
>>> from post.models import Topic
>>> from django.shortcuts import get_list_or_404
>>> get_list_or_404(Topic)
[<Topic: 1: first topic>, <Topic: 2: second topic>, <Topic: 3: third topic!(管理员删除)>]
```

如果传递的参数不能匹配任何实例对象，将会返回 `404`，例如：

```
>>> get_list_or_404(Topic, id__gt=10)
django.http.response.Http404: No Topic matches the given query.
```

`get_object_or_404` 和 `get_list_or_404` 常常用于**不考虑“兼容”的场景中，即匹配不到实例对象就返回找不到资源（404）**。这两个方法会比自己去查询校验并返回 `404` 响应要简单很多，所以，如果需要这样的场景，就首先考虑使用它们。

06.3 基于类的通用视图

对于 `post` 应用，之前已经实现了*话题列表*和*话题详情*视图，这是非常常见的功能。如果再有一个 `book`（图书）应用，那么，可能需要实现*图书列表*和*图书详情*视图。但是，应用越来越多，这样的重复性工作也会越来越多，会使开发过程变得枯燥乏味。`Django` 意识到了这个问题，因此它将常用的功能抽象出来，给开发者提供了**基于类的通用视图**。

用于渲染模板的TemplateView

`Django` 中基于类的视图都应该继承自 `View`，`TemplateView` 也不例外，当视图中没有复杂的业务逻辑，如系统的**引导页面、欢迎页面**等，使用 `TemplateView` 是非常简单方便的。首先，看一看这个视图类的定义（位于 `django/views/generic/base.py` 文件中）：

> 团子注：也就是说，通用视图也是继承自 `View`。

`class TemplateView(TemplateResponseMixin, ContextMixin, View)`
除了基本的 `View` 之外，`TemplateView` 还继承了两个 `Mixin`。

（1）`ContextMixin`：这个类中定义了一个方法：`get_context_data`。它返回一个字典对象，用于渲染模板上下文。通常，在使用 `TemplateView` 时都会重写这个方法，给模板提供上下文数据。

（2）`TemplateResponseMixin`：这个类中定义了两个重要的属性：`template_name` 和 `render_to_response` 方法。其中，`template_name` 用于指定模板路径，**它是必须要提供的**；`render_to_response` 方法**根据模板路径和上下文数据（context）返回TemplateResponse。**

Django模板路径的查找策略

可以在项目的 `settings.py` 文件中找到 `TEMPLATES` 的定义，其中 `APP_DIRS` 默认为 `True`，它告诉模板引擎**搜索应用下的 `templates` 目录**。如果将它设置为 `False`，就需要设置 `DIRS` 来指定模板的位置，例如，将 `DIRS` 设置为：

```
'DIRS': [os.path.join(BASE_DIR, 'templates')]
```

那么，模板引擎会去 `BASE_DIR/templates` 查找模板文件。但当设置了 `DIRS` 且把 `APP_DIRS` 也设置为 `True` 时，还是会使用 `DIRS` 指定路径下的模板文件。

> 团子问：这里有点疑问，难道两个都设置的时候，就不会访问 `app` 下面的文件夹了，只会访问 `DIRS` 指定的路径吗？
> 参考了一下这个回答：https://stackoverflow.com/questions/18029547/django-templates-lookup-order
> 如果两者同时存在，首先找 `APP_DIRS`，然后找 `DIRS`。
> 但是这个还需要验证，因为涉及到了 `loader`，并且这个回答版本较老。

这里为了方便，不修改默认配置，即 `APP_DIRS` 设置为 `True`，`DIRS` 为空列表。为了让模板引擎能够找到 `post` 应用的模板，在 `post` 应用中创建 `templates/post` 目录，并在这个目录下面创建一个模板文件 `index.html`，内容如下：
`post/templates/post/index.html`

```
<h1>{{ hello }}</h1>
```

`{{}}` 是 `Django` 模板系统中引用变量的形式，在使用当前的模板时，`context` 需要包含 `hello`。另外，当前模板文件的完整路径是 `my_bbs/post/templates/post/index.html`。

接下来，利用 `TemplateView` 实现对 `index.html` 的渲染：

```
from django.views.generic import TemplateView

class IndexView(TemplateView):
    template_name = 'post/index.html'

    def get_context_data(self, **kwargs):
        context = super(IndexView, self).get_context_data(**kwargs)
        context['hello'] = 'Hello Django BBS'
        return context
```

`IndexView` 继承自 `TemplateView`，并提供了 `template_name` 指定模板的位置。注意，由于 `Django` 模板引擎会去 `post` 应用的 `templates` 目录下寻找模板，所以，这里只需要给出相对 `templates` 目录的路径。

重写了 `get_context_data` 方法，并在上下文字典中加入了 `hello`，所以，`index.html` 模板渲染的结果就是：`Hello Django BBS`。
最后，还需要给 `IndexView` 配置 `URL` 模式，在 `post/urls.py` 文件中添加：

```
path('index/', views.IndexView.as_view())
```

`View` 的 `as_view` 方法**给出视图类的可调用入口**。访问http://127.0.0.1:8000/post/index/可以看到模板渲染后的效果。

除了指定 `template_name` 参数之外，还可以提供 `get_template_names` 方法指定模板的路径，例如：

```
class IndexView(TemplateView):
    ...
    def get_template_names(self):
        return 'post/index.html'
```

如果不使用 `TemplateView`，而是使用视图函数的形式，则实现同样的功能可以这样：

```
from django.shortcuts import render
```

```
def index_view(request):
    return render(request, 'post/index.html', context={'hello': 'Hello Django BBS'})
```

可以看到，两种渲染模板的实现中，最大的差别就是视图函数的代码量少了很多。那么，既然视图函数实现同样的效果只需要很少的代码量，为什么 `Django` 还要提供 `TemplateView` 这个通用视图呢？

类的特点是抽象，将共性抽离出来，再利用继承去实现特定的逻辑，可以在很大程度上实现代码复用。而这一点对于函数来说，是很难做到的。虽然可以使用装饰器给函数添加额外的功能，但是这也增加了代码实现的难度和复杂性。

对于通用视图来说，使用它们的优势是可以**更加专注地实现业务逻辑**，避免了两类样板式的代码。
第一类，对应 `HTTP` 请求类型的同名（小写）请求方法。例如，`IndexView` 中并没有提供 `get` 方法，但是可以接受 `GET` 请求。
第二类，返回 `HttpResponse` 对象。同样，也没有在 `IndexView` 中返回任何响应，这也是在 `TemplateView` 中完成的。

> 团子注：也就是说，不用定义 `get` 方法，也不用返回对象，只需要重写方法、设置类属性即可。

`TemplateView` 是 `Django` 提供的一个最简单的通用视图，**用于展示给定的模板，但是，不应该在这里实现创建或更新对象的操作。**

用于重定向的 `RedirectView`
页面重定向在 `Web` 开发中也是很常见的行为，所以，`Django` 为重定向功能的实现提供了通用类视图 `RedirectView`。它定义于 `django/views/generic/base.py` 文件中，首先，看一看它的定义：

```
class RedirectView(View):
    permanent = False
    url = None
    pattern_name = None
    query_string = False

    def get_redirect_url(self, *args, **kwargs):
        ...
    def get(self, request, *args, **kwargs):
        ...
```

`RedirectView` 中定义了**四个类属性和两个方法**，下面依次介绍它们的作用和功能。

（1）`permanent`：标识是否使用永久重定向，默认是 `False`。所以，默认情况下其实现的是临时重定向，即 `302` 响应。

（2）`url`：重定向的地址。

（3）`pattern_name`：重定向目标 `URL` 模式的名称（即 `path` 中的 `name` 参数），同时在默认的实现中会将传递给视图的相同位置的参数和关键字参数一并传递给 `reverse` 方法获取反向解析得到的 `URL`。**`url` 和 `pattern_name` 至少需要提供一个**，否则由于获取不到需要重定向的地址将会返回 `410`（`HttpResponseGone`）。

> 团子问：两个都有用哪个？？？

（4）`query_string`：是否将查询字符串拼接到新地址中，默认为 `False`，将**丢弃原地址中的查询字符串。**

`get_redirect_url` 方法用于构造重定向的目标 `URL`，为了更好地理解它的构造过程，下面介绍它的实现：

```
def get_redirect_url(self, *args, **kwargs):
    # 如果指定了 url，则使用 kwargs 替换 url 中的命名参数
    if self.url:
        url = self.url % kwargs
    # 如果指定了 pattern_name，则使用 reverse 方法反向解析得到 url
    elif self.pattern_name:
        url = reverse(self.pattern_name, args=args, kwargs=kwargs)
    # url 和 pattern_name 都没有指定，返回 None
    else:
        return None
    args = self.request.META.get('QUERY_STRING', '')
    # 如果存在查询字符串且 query_string 为 True，则还要完成 url 的拼接
    if args and self.query_string:
        url = '%s?%s' % (url, args)
    return url
```

> 团子注：从这里看出，如果同时指定了 `url` 和 `pattern_name`，则优先使用 `url`，而不会采用 `pattern_name`。

通常，在使用 `RedirectView` 实现重定向时都会重写 `get_redirect_url` 方法，在其内部完成视图的业务逻辑，并返回重定向地址。

> 团子注：也就是说，如果简单的话，直接写到 `url` 或者 `pattern_name` 里就可以了。如果复杂的话，不如写成 `get_redirect_url`。这三个东西是三选一。

`GET` 请求类型会调用 `RedirectView` 中的 `get` 方法，下面来看其内部实现：

```
def get(self, request, *args, **kwargs):
    # 从 get_redirect_url 中获取重定向地址
    url = self.get_redirect_url(*args, **kwargs)
    if url:
        # permanent 为 True，永久重定向
        if self.permanent:
            return HttpResponsePermanentRedirect(url)
        else:
            return HttpResponseRedirect(url)
    else:
        logger.warning(
            'Gone: %s', request.path,
            extra={'status_code': 410, 'request': request}
        )
        # 不存在 url，返回 410
        return HttpResponseGone()
```

考虑这样一个场景：给 `Comment` 实例添加点赞（up）的功能，完成之后重定向到它所对应的 `Topic` 实例的详情页。下面使用 `RedirectView` 来完成这个功能。

首先，给话题详情 `URL` 模式命名：

```
path('topic/<int:topic_id>/', views.topic_detail_view, name='topic_detail'),
```

`topic_detail` 将会设置给参数 `pattern_name`（需要指定命名空间），`Comment` 点赞功能视图类定义如下：

```
class CommentUpRedirectView(RedirectView):
    pattern_name = 'post:topic_detail'
    query_string = False  # 这个默认就是 False，可以不用写

    def get_redirect_url(self, *args, **kwargs):
        comment = Comment.objects.get(pk=kwargs['comment_id'])
        comment.up = F('up') + 1
        comment.save()
        del kwargs['comment_id']
        kwargs['topic_id'] = comment.topic_id
        return super(CommentUpRedirectView, self).get_redirect_url(*args, **kwargs)
```

`CommentUpRedirectView` 在 `get_redirect_url` 方法中实现了业务逻辑：根据传递的 `comment_id` 获取 `Comment` 实例对象，并给它的 `up` 字段加 `1`。为了替换话题详情 `URL` 的位置参数（`topic_id`），先删除了 `kwargs` 中的 `comment_id`，再把 `topic_id` 添加进去。最终，按照默认构造规则返回重定向地址。
最后，还需要给重定向视图类定义URL模式，在 `post/urls.py` 文件中添加：

```
path('comment_up/<int:comment_id>/', views.CommentUpRedirectView.as_view()),
```

可以尝试访问http://127.0.0.1:8000/post/comment_up/6/，执行点赞功能之后，当前页面会重定向到http://127.0.0.1:8000/post/topic/4/。

TemplateView与RedirectView非常有用，它们将视图的功能表达得更为直接，**隐藏了样板式的重复代码**。不仅如此，`Django` 也为操作 `Model` 提供了通用视图类。

用于展示Model列表的ListView
展示 `Model` 列表的视图几乎在任何一个应用中都会出现，而且可能还会出现很多次。例如，在 `post` 应用中，展示 `Topic` 列表的 `topic_list_view`。`Django` 为此进行了抽象，提供了通用 `Model` 列表视图类：`ListView`。

`ListView` 定义于 `django/views/generic/list.py` 文件中，其内部并没有声明任何属性：

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView)
```

`MultipleObjectTemplateResponseMixin` 定义如下：

```
class MultipleObjectTemplateResponseMixin(TemplateResponseMixin):
    # 模板名称后缀
    template_name_suffix = '_list'

    def get_template_names(self):
        """
        返回模板名称的列表
        """
        try:
            # 获取父类中设置的 template_name
            names = super().get_template_names()
        except ImproperlyConfigured:
            names = []
```

```
        # 如果 object_list 包含 model 属性，则获取 model 的元信息构造模板名称
        if hasattr(self.object_list, 'model'):
            opts = self.object_list.model._meta
            names.append('%s/%s%s.html' % (opts.app_label, opts.model_name, self.templat
e_name_suffix))

        return names
```

`MultipleObjectTemplateResponseMixin` 继承自 `TemplateResponseMixin`。

`get_template_names` 返回模板名称（路径）列表，它除了会获取父类中的 `template_name` 之外，还会根据 `Model` 的元信息构造一个默认的模板名称（需要 `object_list` 是 `QuerySet`），并添加到列表的末尾（使用 `append` 方法）。所以，对于模板名称，`ListView` 有这样的两个特性。

（1）可以不需要提供模板名称，使用默认的规则。例如，对于 `Topic`，默认的模板名称即为 `post/topic_list.html`。

（2）由于默认构造的模板名称在列表的末尾，而 `Django` 模板引擎会使用列表中的第一个可用模板，所以，如果提供了 `template_name`，则会使用自定义的模板。

从 `MultipleObjectTemplateResponseMixin` 的实现中可以得出结论，它的主要功能是对基于 `Model` 列表操作的视图执行模板的渲染操作。同时，提供了默认模板名称的构造规则，提高了视图类的易用性。

接下来，介绍 `BaseListView` 的定义：

```
class BaseListView(MultipleObjectMixin, View):
    def get(self, request, *args, **kwargs):
        self.object_list = self.get_queryset()
        allow_empty = self.get_allow_empty()
        # 是否允许空 Model 集合
        if not allow_empty:
            ...
            if is_empty:
                raise Http404(...)
        context = self.get_context_data()
        return self.render_to_response(context)
```

`BaseListView` 只是为 `GET` 请求类型定义了 `get` 方法，其中 `get_queryset`、`get_allow_empty` 和 `get_context_data` 都来自它的父类 `MultipleObjectMixin`。接下来介绍这三个方法的定义。

`get_queryset` 方法用于获取视图展示的 `Model` 列表，需要注意，返回值**必须是可迭代对象**，且**可以是 `QuerySet` 实例**。它的定义如下：

```
def get_queryset(self):
    # queryset 可以是可迭代对象或 QuerySet 实例
    if self.queryset is not None:
        queryset = self.queryset
        if isinstance(queryset, QuerySet):
            queryset = queryset.all()
        # 使用 Model 默认模型管理器获取对象实例
    elif self.model is not None:
        queryset = self.model._default_manager.all()
    # queryset 或 model 至少要定义一个
    else:
        raise ImproperlyConfigured(...)
```

```
        ordering = self.get_ordering()
        # 如果定义了排序规则，则需要对 queryset 排序
        if ordering:
            if isinstance(ordering, str):
                ordering = (ordering, )
            queryset = queryset.order_by(*ordering)
        return queryset
```

这里需要注意一个问题，如果提供的 `queryset` 是一个 `Model` 列表，那么，它并不包含 `model` 属性，因此，需要自定义模板名称。

> 团子注：看不懂?

`get_allow_empty` 方法返回 `allow_empty` 属性，它是一个布尔值，默认是 `True`，代表允许展示空的 `Model` 列表。如果设置为 `False`，且 `Model` 列表为空，则会返回 `404`。

`get_context_data` 方法返回用于渲染模板的上下文数据，在分析它的实现之前，先看 `get_context_object_name` 方法的定义：

```
    def get_context_object_name(self, object_list):
        if self.context_object_name:
            return self.context_object_name
        elif hasattr(object_list, 'model'):
            return '%s_list' % object_list.model._meta.model_name
        else:
            return None
```

这个方法返回视图操作的数据列表的**上下文变量名称**。`object_list` 参数即 `get_queryset` 方法的返回值，如果它有 `model` 属性，那么这个方法会返回 `model` 名称与 `list` 拼接得到的字符串（在没有设置 `context_object_name` 的情况下）。例如，对于 `Topic` 来说，这里的返回值就是 `topic_list`。

> 团子注：`context_object_name` 翻译成上下文对象名称比较好吧。

最后，`get_context_data` 方法的定义如下：

```
    def get_context_data(self, *, object_list=None, **kwargs):
        queryset = object_list if object_list is not None else self.object_list
        # page_size 指定每一页显示多少个 Model 对象
        page_size = self.get_paginate_by(queryset)
        # 获取上下文变量名称
        context_object_name = self.get_context_object_name(queryset)
        if page_size:
            ...
        else:
            context = {
                ...
                # context 中设置 key 为 object_list 的数据列表
                'object_list': queryset
            }
        # 如果存在上下问变量名称，则将它作为 key 填充数据列表
        if context_object_name is not None:
```

```
            context[context_object_name] = queryset
        context.update(kwargs)
        return super().get_context_data(**context)
```

接下来，使用 `ListView` 实现 `Topic` 列表视图。首先，在 `post` 应用的 `templates/post` 目录下新建模板文件 `topic_list.html`：

```
{% block content %}
  <h2>Topic</h2>
  <ul>
    {% for topic in object_list %}
      <li><a href="{% url 'post:topic_detail' topic.id %}">{{ topic.title }}</a></li>
    {% endfor %}
  </ul>
{% endblock content %}
```

除了可以使用 `object_list` 之外，在特定情况下，还可以使用 `topic_list`。

> 团子注：什么情况？

这里之所以将模板命名为 `topic_list.html`，也是因为这样可以利用默认的模板名称构造规则（`MultipleObjectTemplateResponseMixin` 的 `get_template_names` 方法），不需要在 `ListView` 中显式地指定 `template_name` 属性。

使用 `ListView` 最简单的形式是只定义 `model` 属性：

```
class TopicList(ListView):
    model = Topic
```

根据之前对 `ListView` 的分析，可以知道，这样会展示所有的 `Topic` 实例，且会根据默认的模板名称构造规则查询模板。
最后，定义视图的 `URL` 模式，在 `post/urls.py` 文件中添加：

```
path('topics/', views.TopicList.as_view()),
```

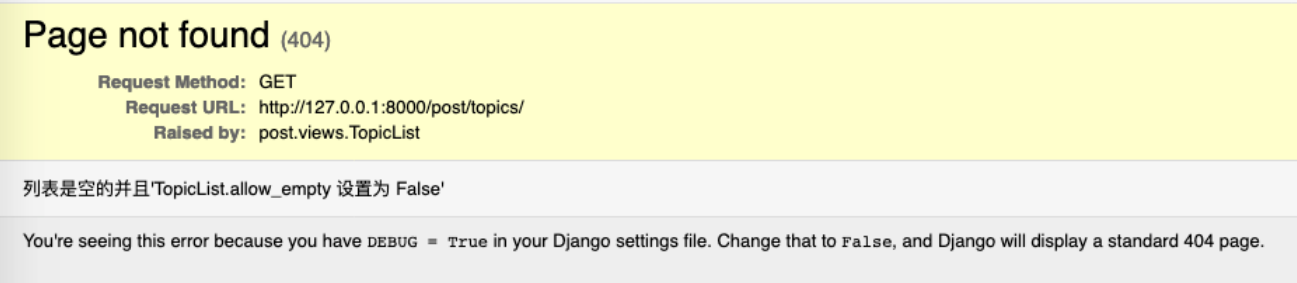
由此可以在浏览器中访问http://127.0.0.1:8000/post/topics/查看Topic实例列表，且可以单击每一个 `Topic` 实例跳转到详情页。

除了可以设置 `model` 属性之外，还可以设置 `queryset` 属性实现同样的效果：

```
class TopicList(ListView):
    queryset = Topic.objects.all()
```

如果设置了 `allow_empty` 为 `False`，则当 `queryset` 为空列表时，会返回 `404` 响应：

```
class TopicList(ListView):
    queryset = Topic.objects.filter(pk__gt=10)
    allow_empty = False
```



也可以尝试设置 `ListView` 的一些其他属性，验证效果是否符合预期，如可以通过定义 `template_name` 指定模板的名称、设置 `context_object_name` 指定模板中的迭代关键字、设置 `paginate_by` 实现分页效果等。

用于展示Model详情的DetailView

`DetailView` 的源码组织与 `ListView` 非常相似，它用于展示单个 `Model` 对象的详情。这也是非常常见的场景，如 `post` 应用中的 `Topic` 详情。

`DetailView` 定义于 `django/views/generic/detail.py 文件中，其内部同样没有声明任何属性和方法：

```
class DetailView(SingleObjectTemplateResponseMixin, BaseDetailView)
```

`SingleObjectTemplateResponseMixin` 对象中定义了 `get_template_names` 方法返回模板名称的列表，同时，它也定义了规则生成默认的模板名称：`app_label/model_name_detail.html`。由于这个过程比较简单，且与 `ListView` 类似，所以，这里重点分析 `BaseDetailView`。

`BaseDetailView` 定义如下：

```
class BaseDetailView(SingleObjectMixin, View):
    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        return self.render_to_response(context)
```

`BaseDetailView` 同样为 `GET` 请求类型定义了 `get` 方法，其中 `get_object` 和 `get_context_data` 方法都来自它的父类 `SingleObjectMixin`。接下来，就来看一看这两个方法的实现。

```
def get_object(self, queryset=None):
    if queryset is None:
        # 1. 如果设置了 queryset 属性，则直接使用
        # 2. 通过 model 属性默认模型管理器的 all 方法获取 QuerySet
        queryset = self.get_queryset()

    pk = self.kwargs.get(self.pk_url_kwarg)
    slug = self.kwargs.get(self.slug_url_kwarg)
    # 通过主键查询 Model 实例对象
    if pk is not None:
        queryset = queryset.filter(pk=pk)
    # 定义了 slug 且 pk 不存在则使用 slug 条件查询
    if slug is not None and (pk is None or self.query_pk_and_slug):
        slug_field = self.get_slug_field()
        queryset = queryset.filter(**{slug_field:slug})
    # pk 和 slug 至少提供一个
    if pk is None and slug is None:
        raise AttributeError(...)
```

```
    try:
        obj = queryset.get()
    except queryset.model.DoesNotExist:
        raise Http404(...)
    return obj
```

`get_object` 方法的实现比较简单，它会根据 `URL` 中给出的条件（`pk` 或 `slug`）过滤 `Model` 实例对象，如果不存在则会返回 `404` 响应。

`get_context_data` 方法返回用于渲染模板的字典上下文数据

```
def get_context_data(self, **kwargs):
    context = {}
    if self.object:
        context['object'] = self.object
        # get_context_object_name 方法可能会返回 Model 的名称
        context_object_name = self.get_context_object_name(self.object)
        if context_object_name:
            context[context_object_name] = self.object
    context.update(kwargs)
    return super().get_context_data(**context)
```

通常，这个方法会被重写，因为 `Model` 实例对象除了自身的基本信息之外，还可能需要展示关联对象的信息，如每一个 `Topic` 实例都可能会有多个 `Comment` 实例，这些同样需要展示。

理解了 `DetailView` 的实现原理，下面我们使用它重新实现 `Topic` 详情视图

```
class TopicDetailView(DetailView):
    model = Topic

    def get_context_data(self, **kwargs):
        context = super(TopicDetailView, self).get_context_data(**kwargs)
        pk = self.kwargs.get(self.pk_url_kwarg)
        context.update({
            'comment_list': Comment.objects.filter(topic=pk)
        })
        return context
```

> 团子注：这里的 `pk = self.kwargs.get(self.pk_url_kwarg)` 不是很明白！

在 `post/urls.py` 文件中添加 `TopicDetailView` 的 `URL` 模式：

```
path('topic_view/<int:pk>/', views.TopicDetailView.as_view()),
```

`URL` 中定义了 `pk` 命名参数，所以，`TopicDetailView` 会按照主键查询 `Topic` 实例对象，可以通过 `pk_url_kwarg` 属性获取 `URL` 中的主键值。

`TopicDetailView` 非常简单，只是设置了 `model` 和 `get_context_data` 属性，根据当前的设置可以知道：

（1） `model` 设置为 `Topic` 对象，所以可以根据 `get_template_names` 中的规则生成默认的模板名称 `post/topic_detail.html`。

（2） `get_context_data` 方法返回的 `context` 字典中除了包含 `object` 和 `topic`（父类中填充的 `Topic` 实例对象）之外，还主动添加了 `comment_list`，指定当前 `Topic` 对应的 `Comment` 实例。

> 团子注：`object` 应该就是根据条件过滤出来的那个实例吧？

由此可以简单地写一个模板用来展示 `Topic` 详情，创建 `topic_detail.html` 模板文件（注意路径要正确）：

```
{% block content %}
<h2>Topic</h2>
<ul>
  <li>(id): {{ topic.id }}</li>
  <li>(title): {{ topic.title }}</li>
  <li>(content): {{ topic.content }}</li>
  <li>(user): {{ topic.user.username }}</li>
  <li>(created_time): {{ topic.created_time }}</li>
  <li>(last_modified): {{ topic.last_modified }}</li>
  <li>(评论):
    <table border="1">
      <tr>
        <th>评论id</th>
        <th>内容</th>
        <th>赞同</th>
        <th>反对</th>
      </tr>
      {% for comment in comment_list %}
      <tr>
        <td>{{ comment.id }}</td>
        <td>{{ comment.content }}</td>
        <td>{{ comment.up }}</td>
        <td>{{ comment.down }}</td>
      </tr>
      {% endfor %}
    </table>
  </li>
</ul>
{% endblock content %}
```

尝试访问http://127.0.0.1:8000/post/topic_view/5/，可以看到id为5的Topic实例对象的信息，同时包含了它所对应的所有Comment的信息。

Topic

- (id): 5
- (title): 我的英雄学院
- (content): 我喜欢看我的英雄学院，你们呢?
- (user): admin
- (created_time): 2019年11月5日 08:21
- (last_modified): 2019年11月5日 08:21
- (评论):

评论id	内容	赞同	反对
7	我也喜欢看，我喜欢绿谷。	10	5
8	我也是，我喜欢御茶子。	100	10

06.4 视图工作原理分析

- `HttpRequest` 在请求到达视图的时候创建，并作为第一个参数传入视图，那么，`Django` 是怎么完成创建的呢？
- 在 `HttpRequest` 对象中，`GET` 和 `POST` 都是 `QueryDict` 类型的实例，为什么 `Django` 不直接使用 `dict` 类型呢？
- 基于类的视图都需要继承自 `View`，且配置 `URL` 模式需要使用 `View` 的 `as_view` 方法，这里面都做了些什么呢？又是怎样实现请求分发的呢？

解决一键多值问题的QueryDict

对于 `Python` 的字典（dict）类型，如果一个键对应多个值，那么，对应键只会保留最后一个值。但是在 `HTML` 表单中，一个键对应多个值是很正常的，例如复选框就是一种很常见的情况。`QueryDict` 就是用来解决这个问题的，它允许一个键对应多个值，且它在一次请求到响应的过程中是不可变的。

> 团子注：复选框这个例子不懂。

`QueryDict` 继承自 `MultiValueDict`，`MultiValueDict` 又继承自 `dict`，所以，它们都是字典的子类。接下来，先来看一看 `MultiValueDict` 中定义的重要方法，再去看 `QueryDict` 对查询字符串的转换过程。

1.MultiValueDict

`MultiValueDict` 定义于 `django/utils/datastructures.py` 文件中，它是 `dict` 的子类，用来处理多个值对应相同键的场景。同时，`Django` 在这个文件中还定义了一些其他的数据结构以适用于特定的场景。

在 `MultiValueDict` 的注释中已经给出了常用的使用过程，如下所示：

```
>>> from django.utils.datastructures import MultiValueDict
>>> d = MultiValueDict({'name': ['Adrian', 'Simon'], 'position': ['Developer']})
>>> d['name']
'Simon'
>>> d.getlist('name')
['Adrian', 'Simon']
>>> d.getlist('doesnotexist')
[]
>>> d.getlist('doesnotexist', ['Adrian', 'Simon'])
['Adrian', 'Simon']
>>> d.get('lastname', 'nonexistent')
'nonexistent'
```

对于一个字典来说，最重要的当然是**根据键获取值**，所以，这里考虑 `MultiValueDict` 的三个重要方法：`__getitem__`、`get`、`getlist`。

`__getitem__` 是 `Python` 中的魔术方法，当类对象中定义了这个方法时，类实例就可以通过 `[]` 运算符取值。这里 `MultiValueDict` 重写了其父类 `dict` 的实现，源码如下：

```
def __getitem__(self, key):
    try:
        list_ = super().__getitem__(key)
    except KeyError:
        raise MultiValueDictKeyError(key)
    try:
```

```
            return list_[-1]
        except IndexError:
            return []
```

对于 `__getitem__` 的实现可以得出以下两个结论。

（1）如果 `key` 不存在，会抛出 `django.utils.datastructures.MultiValueDictKeyError` 异常。

（2）当一个 `key` 有多个值时，获取最后一个值。

在视图中使用 `request.GET['a']` 获取 `a` 的值，即使用 `MultiValueDict` 的 `__getitem__` 方法。当然，也可以通过 `request.GET.get('d',0)` 获取 `d` 的值，这里使用的就是 `get` 方法：

```
    def get(self, key, default=None):
        try:
            val = self[key]
        except KeyError:
            return default
        if val == []:
            return default
        return val
```

从 `get` 的实现中可以看出，它尝试使用 `__getitem__` 方法获取 `key` 的值，如果获取不到则返回给定的默认值。需要注意，在通过 `self[key]` 获取 `key` 的值时，捕获的是 `KeyError` 异常，这是因为 `MultiValueDictKeyError` 是 `Python` 标准 `KeyError` 的一个子类。

`get` 和 `__getitem__` 都只能获取到 `key` 的最后一个值，如果需要获取 `key` 的所有值，则应使用 `getlist` 方法：

```
    def getlist(self, key, default=None):
        return self._getlist(key, default, force_list=True)
```

`getlist` 与 `get` 方法类似，可以接受一个默认值，其内部实现调用了 `_getlist` 方法，需要注意，最后传递了一个 `force_list=True`。下面介绍 `_getlist` 的实现：

```
    def _getlist(self, key, default=None, force_list=False):
        try:
            values = super().__getitem__(key)
        except KeyError:
            if default is None:
                return []
            return default
        else:
            if force_list:
                values = list(values) if values is not None else None
            return values
```

`_getlist` 调用父类的 `__getitem__` 方法获取 `key` 对应的 `values`，如果不存在 `key`，则考虑使用 `default`。最后，`force_list` 的作用是获取 `values` 的副本。

2.QueryDict

`QueryDict` 中常用的方法都继承自 `MultiValueDict`，下面主要介绍 `QueryDict` 的构造函数。

`URL` 中的查询字符串可以直接传递给 `QueryDict` 构造实例对象，例如：

```
>>> from django.http.request import QueryDict
>>> QueryDict('a=1&a=2&a=3')
<QueryDict: {'a': ['1', '2', '3']}>
```

> 团子注：有点理解多值字典是个啥意思了。就是可以给同一个键多次赋值，这些赋值不会覆盖，而是会依次保存起来。返回的时候返回最后一次保存的值。所以复选框相当于爱好为唱跳rap篮球。

接下来介绍 `QueryDict` 的构造函数是怎样处理查询字符串的：

```
def __init__(self, query_string=None, mutable=False, encoding=None):
    super().__init__()
    ...
    # 团子注：注意这样的用法，如果没有传递 query_string 就给它一个 空字符串
    query_string = query_string or ''
    parse_qsl_kwargs = {
        'keep_blank_values': True,
        'fields_limit': settings.DATA_UPLOAD_MAX_NUMBER_FIELDS,
        'encoding': encoding,
    }
    ...
    for key, value in limited_parse_qsl(query_string, **parse_qsl_kwargs):
        self.appendlist(key, value)
    self._mutable = mutable
```

其中 `_mutable` 属性用来标记当前的 `QueryDict` 实例是否是可变的。除此之外，`limited_parse_qsl` 方法将 `query_string` 分解为 `key` 和 `value`，并最终通过父类的 `appendlist` 方法将 `key` 映射到 `list`。这里重点掌握 `limited_parse_qsl` 方法（定义于 `django/utils/http.py` 文件中）的实现：

> 团子注：130页错误，默认为 `False`，不是 `True`

```
def limited_parse_qsl(qs, keep_blank_values=False, encoding='utf-8', errors='replace', f
ields_limit=None):
    if fields_limit:
        # fields_limit 限制字段个数的最大值，默认是 1000
        pairs = FIELDS_MATCH.split(qs, fields_limit)
        if len(pairs) > fields_limit:
            raise TooManyFieldsSent(
                'The number of GET/POST parameters exceeded '
                'settings.DATA_UPLOAD_MAX_NUMBER_FIELDS.'
            )
    else:
        pairs = FIELDS_MATCH.split(qs)
    r = []
    for name_value in pairs:
        if not name_value:
            continue
        nv = name_value.split('=', 1)
        ...
        # keep_blank_values 标识是否保留空字符，默认为 False
        if len(nv[1]) or keep_blank_values:
            name = nv[0].replace('+', ' ')
            value = nv[1].replace('+', ' ')
            r.append((name, value))
```

```
        return r
```

这个方法返回一个 `list`，其中每一个元素都是 `key` 和 `value` 的二元组。`FIELDS_MATCH` 通过正则匹配将 `qs`（查询字符串）分割并返回一个 `list`（pairs），例如：

```
>>> from django.utils.http import FIELDS_MATCH
>>> FIELDS_MATCH.split('a=1&a=2&c=3')
['a=1', 'a=2', 'c=3']
```

之后，再对 `pairs` 中的每一个元素按照等号“`=`”分割：第一个元素作为 `name`，第二个元素作为 `value`。所以，最终的返回值是：

```
[('a', '1'), ('a', '2'), ('c', '3')]
```

类视图基类View源码分析

`View` 定义于 `django/views/generic/base.py` 文件中，其功能实现主要依赖于三个重要的方法：`http_method_not_allowed`、`dispatch` 和 `as_view`。

1.http_method_not_allowed

这个方法返回 `HttpResponseNotAllowed`（405）响应，标识当前的请求类型不被支持。例如，`GetView` 只定义了 `get` 方法，当它收到 `POST` 请求时，由于找不到 `post` 方法的定义，则会被分发（dispatch）到 `http_method_not_allowed`。

2.dispatch

`dispatch` 方法根据 `HTTP` 请求类型调用 `View` 中的同名函数，实现了请求的分发。其源码如下：

```
def dispatch(self, request, *args, **kwargs):
    if request.method.lower() in self.http_method_names:
        handler = getattr(self, request.method.lower(), self.http_method_not_allowed)
    else:
        handler = self.http_method_not_allowed
    return handler(request, *args, **kwargs)
```

`http_method_names` 定义当前 `View` 可以接受的请求类型：

```
http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options',
'trace']
```

首先，判断当前的请求类型是否可以被接受（是否定义在 `http_method_names` 中）。

（1）可以接受：尝试获取 `View` 中的同名方法，如果不存在，则会将 `handler` 指定为 `http_method_not_allowed`。
（2）不被接受：将 `handler` 指定为 `http_method_not_allowed`。

`handler` 即为视图处理函数，所以，向其传递了 `request` 等参数。

3.as_view

`Django` 给 `as_view` 方法加了 `@classonlymethod` 装饰器，作用是**只允许类对象调用这个方法**，实例调用将抛出 `AttributeError` 异常。这个装饰器非常有用，即使不是 `Django` 项目，也可以参照 `Django` 的实现给类定义增加

限制。

`Django` 将一个 `HTTP` 请求映射到一个可调用的函数，而不是一个类对象。所以，在定义 `URL` 模式的时候总是需要调用 `View` 的 `as_view` 方法。它的实现如下：

```
classmethod
def as_view(cls, **initkwargs):
    ...
    def view(request, *args, **kwargs):
        # 创建 View 类实例
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        # 调用 view 实例的 dispatch 方法
        return self.dispatch(request, *args, **kwargs)
    view.view_class = cls
    view.view_intikwargs = initkwargs
    ...
    return view
```

可以看到，它的实现非常简单：创建了 `View` 类的实例，然后调用 `dispatch` 方法根据请求类型分发处理函数。

> 团子注：`as_view` 返回一个叫做 `view` 的函数，`view` 内部使用了 `View` 实例的 `dispatch` 方法进行了不同类型请求的分发。

至此，我们讲解了 `View` 的实现原理，也介绍了为什么需要在类视图中定义与 `HTTP` 请求类型同名的函数。接下来，介绍 `Django` 创建 `HttpRequest` 的过程。

HttpRequest的创建过程

在分析 `HttpRequest` 的创建过程之前，需要先理解 `WSGI` 的含义。`WSGI` 是 `Web Server Gateway Interface` 的缩写，即 **`Web` 服务器网关接口**。

`WSGI` 是一个规范，它定义了 `Web` 服务器与 `Python` 应用程序交互的协议。定义一个 `WSGI` 可接受的应用程序非常简单：

```
def python_app(environ, start_response):
    pass
```

`python_app` 就是一个符合 `WSGI` 标准的 `HTTP` 处理函数，同时，它接收服务器传递的两个参数。

（1）`environ`：包含 `HTTP` 请求信息的 `dict` 对象，存放了所有与客户端相关的信息，如 `REQUEST_METHOD`（请求类型）、`QUERY_STRING`（查询字符串）等。

（2）`start_response`：发送 `HTTP` 响应的可调用对象，返回响应码和响应头信息。

> 团子注：`start_response` 这个是给 `Web` 服务器用来回调的函数吗？
> 不是，下面有解释。`start_response` 是 `WSGI` 传递给 `Web` 应用程序的回调函数，由 `Web` 应用程序调用，目的是给 `WSGI` 返回响应码和响应头。

`Django` 作为一个 `Web` 框架，当然也需要实现 `WSGI` 协议，它就是 `WSGIHandler`。当 `Django` 应用启动时，会初始化一个 `WSGIHandler` 实例，在其中完成请求与响应的过程。所以，创建 `HttpRequest` 的工作就由 `WSGIHandler` 完成。

`WSGIHandler` 定义于 `django/core/handlers/wsgi.py` 文件中，源码实现如下所示：

```
class WSGIHandler(base.BaseHandler):
    request_class = WSGIRequest
    ...
    def __call__(self, environ, start_response):
        ...
        # 根据 environ 创建 HttpRequest 对象
        request = self.request_class(environ)
        # 中间件和视图函数对 request 进行处理，得到响应
        response = self.get_response(request)
        response._handler_class = self.__class__
        status = '%d %s' % (response.status_code, response.reason_phrase)
        response_headers = list(response.items())
        for c in response.cookies.values():
            response_headers.append(('Set-Cookie', c.output(header='')))
        # 调用 WSGI 服务器传入的 start_response 发送响应码和响应头
        start_response(status, response_headers)
        ...
        return response
```

`__call__` 是 `Python` 中的一个魔术方法，它可以**让类实例的行为表现得像函数一样**，即**允许一个类的实例像函数一样被调用**。

注意到 `WSGIHandler` 中的 `request_class` 属性定义为 `WSGIRequest`，所以，视图函数中的第一个参数实际是 `WSGIRequest` 类型的实例。

> 团子注：不是 `HttpRequest` 的实例，而是 `WSGIRequest` 的实例！`WSGIRequest` 是 `HttpRequest` 的子类。

首先，看一看 `WSGIRequest` 对象的定义：

```
class WSGIRequest(HttpRequest):
    def __init__(self, environ):
        script_name = get_script_name(environ)
        path_info = get_path_info(environ)
        ...
        self.META = environ
        self.META['PATH_INFO'] = path_info
        self.META['SCRIPT_NAME'] = script_name
        self.method = environ['REQUEST_METHOD'].upper()
```

> 团子注：`request.method` 获取到的方法是**大写的**。

可以看到，它继承自 `HttpRequest`，这也解释了之前多次强调的 `request` 参数的类型问题。在初始化函数中，给实例对象添加了 `path`、`method`、`META` 等属性，属性值来自 **`WSGI` 服务器解包 `HTTP` 请求生成的 `environ`**。之前介绍的 `HttpRequest` 属性都来自这里。

最后，分析 `WSGIRequest` 中的两个重要属性：`GET` 和 `POST`。下面是 `GET` 的定义：

```
    cached_property
    def GET(self):
        raw_query_string = get_bytes_from_wsgi(self.environ, 'QUERY_STRING', '')
        return QueryDict(raw_query_string, encoding=self._encoding)
```

`@cached_property` 装饰器相当于给 `Python` 中的 `@property` 加上了缓存功能，所以，可以直接通过 `request.GET` 获取返回值。可以看到，在 `GET` 中，获取到 `environ` 中的查询字符串，传递给 `QueryDict` 的初始化方法创建了 `QueryDict` 实例对象。

接下来介绍 `POST` 的定义：

```
    POST = property(_get_post, _set_post)
```

`property` 是 `Python` 内置的描述符，`request.POST` 实际会调用 `_get_post` 方法，下面介绍它的实现：

```
    def _get_post(self):
        if not hasattr(self, '_post'):
            self._load_post_and_files()
        return self._post
```

由于 `WSGIRequest` 初始化时并没有定义 `_post` 属性，所以，需要 `_load_post_and_files`：

```
    def _load_post_and_files(self):
        ...
        # 文件上传
        if self.content_type = 'multipart/form-data':
            ...
            self._post, self._files = self.parse_file_upload(self.META, data)
            ...
        # 表单提交，并将提交的数据进行 urlencode
        elif self.content_type == 'application/x-www-form-urlencoded':
            self._post, self._files = QueryDict(self.body, encoding=self._encoding), MultiVa
    lueDict()
        else:
            self._post, self._files = QueryDict(encoding=self._encoding), MultiValueDict()
```

`_load_post_and_files` 方法根据不同的 `content_type` 解析表单数据，得到 `_post` 和 `_files` 两个属性。不论采用哪一种方式，这两个属性的类型都是固定的：`_post` 是 `QueryDict` 类型的实例，`_files` 是 `MultiValueDict` 类型的实例。最终，`request.POST` 获取到的是 `_post`。

需要注意的是，`request.POST` 中不包含上传文件的数据，`Django` 将上传文件数据放在 `_files` 属性中，且只有当 `content_type` 是 `multipart/form-data` 时才可能会有数据（存在解析失败的情况）。

`HttpRequest` 在 `HTTP` 请求到来的时候，由 `Django` 解析服务器传递的数据（environ）填充到 `WSGIRequest` 的属性得到。关于 `WSGI` 服务器以及 `Django` 项目的启动过程将在后面详细介绍。

> 团子注：这样说太误导人了，应该说 `request` 对象就好了。

HttpResponse的返回过程

`WSGIRequest` 在 `WSGIHandler` 中创建之后，就直接传递到了 `get_response` 方法（来自 `WSGIHandler` 的父类

`BaseHandler`）中，也就是这个方法返回了一次请求的响应。

大多数场景下，`HTTP` 请求的响应是在视图函数中手动创建的 `HttpResponse` 对象。除了要执行视图函数之外，还需要经过许多中间件的处理，由于本书还没有介绍到中间件，这里简单介绍 `WSGIHandler` 是怎样执行到视图函数并返回 `HttpResponse` 的。

首先，看 `get_response` 方法（定义于 `django/core/handlers/base.py` 文件中）的实现：

```
def get_response(self, request):
    # 设置根 URL 的路径
    set_urlconf(settings.ROOT_URLCONF)
    # 处理请求返回响应
    reponse = self._middleware_chain(request)
    ...
    # 404 响应的日志
    if response.status_code = 404:
        logger.warning(
            'Not Found: %s', request.path,
            extra={'status_code': 404, 'request': request},
        )
    return response
```

可以看到，`response` 返回的地方调用了 `_middleware_chain`，而它是在 `load_middleware` 方法中完成赋值的：

```
def load_middleware(self):
    ...
    handler = convert_exception_to_response(self._get_response)
    ...
    self._middleware_chain = handler
```

`convert_exception_to_response` 是个装饰器，用于捕获传递进来的函数执行中的异常。`_get_response` 方法的实现如下：

```
def _get_response(self, request):
    response = None
    # 获取 URL 解析器
    resolver = get_resolver()
    # 通过 path_info 解析 URL 映射到的对象
    resolver_match = resolver.resolve(request.path_info)
    # callback 是定义的视图函数
    # callback_args 是执行视图函数提供的列表参数
    # callback_kwargs 是执行视图函数提供的字典参数
    callback, callback_args, callback_kwargs = resolver_match
    ...
    if response is None:
        # 为支持数据库事务将函数封装起来
        wrapped_callback = self.make_view_atomic(callback)
        try:
            # 执行视图函数，request 作为第一个参数传递
            response = wrapped_callback(request, *callback_args, **callback_kwargs)
        except Exception as e:
            pass
    ...
    return response
```

可以看到，在 `_get_response` 方法中执行了视图函数，且给视图函数传递了相应的参数，这也使我们清楚了为什么视图函数的第一个参数是 `HttpRequest` 类型的实例。

`HttpRequest` 创建与 `HttpResponse` 返回是一次 `HTTP` 请求的标准行为，本章首先通过定义函数视图和基于类的视图解释了 `Django` 中视图的使用方法；之后介绍了 `Django` 内置的基于类的通用视图，这大幅降低了特定场景下的重复性代码；最后，通过分析源码介绍了 `Django` 实现这些功能的原理。

通常，视图都会和模板一起使用，接下来介绍 `Django` 的模板系统。

---

07. Django 模板系统

Django的模板系统将Python代码与HTML代码（模板用于生成HTML）解耦，动态地生成HTML页面。Django项目可以配置一个或多个模板引擎，但是通常使用Django的模板系统时，应该首先考虑其内置的后端DTL（Django Template Language，Django模板语言）。

07.1 模板系统基础

初次使用模板系统

直接将HTML写在Python代码中（例如hello_django_bbs视图中的html变量）是非常不友好的，因为在实际的项目开发中，修改Web页面展示是很频繁的，而且两种不同的语言写在一起不利于项目的维护。Django提供了模板系统将模板（HTML）与服务（Python）分开，降低了项目开发和维护的成本。

通常，模板用于动态地生成 `HTML`，不过 `Django` 的模板可以生成任何文本格式。使用 `Django` 的模板系统通常需要三个步骤的操作。

（1）在项目的 `settings.py` 文件中配置 `TEMPLATES`，指定可用的模板后端。在创建 `Django` 项目的时候，默认会配置内置的**后端DTL**，这也是 `Django` 推荐使用的。因此，通常只需要修改它所对应的默认选项。

（2）创建 `Template` 对象，并提供字符串形式的模板代码。

（3）调用 `Template` 对象的 `render` 方法，并传入字典上下文（Context）。**`render` 方法的返回值是模板代码渲染后的字符串，且变量和标签（模板语法）会由传入的 `Context` 解释替换。**

根据上述步骤的描述可以知道，模板系统主要依赖三个角色：模板后端、`Template` 和 `Context`。下面先介绍 `Template` 和 `Context`，之后，再介绍它们的简单用法。

1.Template

`Template` 对象定义在 `django/template/base.py` 文件中，它的构造函数如下：

```
def __init__(self, template_string, origin=None, name=None, engine=None)
```

它只有一个必填的参数：**字符串表示的模板代码**。可以在 `Shell` 中初始化一个实例：

```
>>> from django.template import Template
>>> Template("This is {{ project }}")
<django.template.base.Template object at 0x10e071d50>
```

在使用构造函数创建了 `Template` 实例时，传递的模板代码就已经被解析了，它被存储为一个树形结构来提高性能。

如果在解析的过程中遇到错误，则会抛出 `TemplateSyntaxError` 异常，例如：

```
>>> Template("This is {{ project %}}")
Traceback (most recent call last):
...
django.template.exceptions.TemplateSyntaxError: ...
```

有了 `Template` 实例之后，就可以使用 `Context` 渲染出来了。对于上面的例子，渲染的过程就是完成对 `project` 变量的替换。

2.Context

`Context` 对象定义于 `django/template/context.py` 文件中，它的构造函数如下：

```
def __init__(self, dict_=None, autoescape=True, use_l10n=None, use_tz=None)
```

通常使用 `Context` 对象会给它传递一个字典对象，即填充构造函数的第一个参数。将 `Context` 对象实例化之后，就可以使用 `Python` 的字典语法来操作“上下文”了：

```
>>> from django.template import Context
>>> c = Context({'project': 'Django BBS'})
>>> c['project']
'Django BBS'
>>> c['test'] = 'test'
>>> del c['test']
>>> c['test']
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/jpch89/code/bbs_python37/my_bbs/venv/lib/python3.7/site-packages/django/template/context.py", line 83, in __getitem__
    raise KeyError(key)
KeyError: 'test'
```

在 `Template` 实例上调用它的 `render` 方法，并传入 `Context`，即可实现模板的渲染：

```
>>> Template("This is {{ project }}").render(Context({'project': 'Django BBS'}))
'This is Django BBS'
```

可以看到，模板中的 `project` 变量被替换了，这几乎就是最简单的使用模板系统的方式了。关于 `Context` 对象，下面再看一个例子：

```
>>> Template("This is {{ project }}").render(Context({'project': 'Django BBS'}))
'This is Django BBS'
>>> t = Template("This is {{ project }}, {{ True.real }}")
>>> c = Context({'project': 'Django BBS'})
>>> t.render(c)
'This is Django BBS, 1'
>>> c
[{'True': True, 'False': False, 'None': None}, {'project': 'Django BBS'}]
>>>
```

定义的 `Context` 实例并没有变量 `True`，但是渲染过程没有出错，且最后打印的实例是一个 `list`，包含了两个字典对象。

第一个字典对象是 `Context` 内置的，包含了 `True`、`False` 和 `None` 三个 `Python` 对象。

第二个字典对象是传递给 `Context` 的参数，如果没有传递（`None`），则 `list` 中不会包含第二个字典对象。

为了搞清楚三个内置变量是怎么加入上下文中的，下面介绍 `Context` 构造函数的实现：

```
class Context(BaseContext):
    def __init__(self, dict_=None, autoescape=True, use_l10n=None, use_tz=None):
    ...
    super().__init__(dict_)
```

这里调用了父类 `BaseContext` 的构造函数，并传递了字典参数 `dict_`：

```
class BaseContext:
    def __init__(self, dict_=None):
        self._reset_dicts(dict_)

    def _reset_dicts(self, value=None):
        # 内置变量
        builtins = {'True': True, 'False': False, 'None': None}
        self.dicts = [builtins]
        # 传递了字典对象
        if value is not None:
            self.dict.append(value)
```

可以看到，构造函数中调用了 `_reset_dicts` 方法，其中定义了三个内置的变量，并作为 `list` 中的第一个元素（`self.dicts`）。如果传递了字典对象，则一并加入 `list` 中。

模板后端的默认配置

`Django` 模板系统默认支持 `DTL` 和 `Jinja2` 模板后端，当然也可以配置其他第三方的模板引擎。`DTL` 是内置于 `Django` 框架中的，也是官方极力推荐使用的模板后端，之前介绍的 `Django` 管理系统就使用 `DTL`。如果没有特别的需要，应该遵循 `Django` 的默认行为，不要更换模板后端。

在 `Django` 项目中，直接能接触到模板后端的就是在 `settings.py` 文件中配置的 `TEMPLATES` 列表。列表中的每一个元素都是一个字典对象，每个字典对象代表了配置的模板后端。下面介绍这些配置项。

`Django` 在创建项目的时候，默认定义的 `TEMPLATES` 如下：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

- `BACKEND`：指定了要使用的模板引擎类的 `Python` 路径，`Django` 默认使用的是 `django.template.backends.django.DjangoTemplates`。
- `DIRS`：一个目录列表，指定模板文件的存放路径。模板引擎将按照列表中定义的顺序查找模板文件。
- `APP_DIRS`：一个布尔值，为 `True` 时，模板引擎会在已安装应用的 `templates` 子目录中查找模板。由于大多数引擎都会从文件目录查找并加载模板，所以，模板引擎都会有 `APP_DIRS` 和 `DIRS` 这两个通用配置。
- `OPTIONS`：指定额外的选项，不同的模板引擎有着不同的可选额外参数。例如这里的 `context_processors` 用于配置模板上下文处理器，在使用 `RequestContext` 时将看到它们的作用。
- `DIRS` 和 `APP_DIRS` 这两个选项决定了模板文件的存放路径，且匹配规则在第6章中已经说明，这里不再赘述。接下来看Django默认配置的上下文处理器。

上下文处理器其实就是一个**返回字典对象的函数**，它们都只有一个 `HttpRequest` 类型的参数。**处理器返回的字典将被加入模板字典上下文**（`Context`）中，所以，**上下文处理器其实是一种通用的模板赋值函数。**

1.django.template.context_processors.debug

`debug` 处理器用于辅助调试，它定义于 `django/template/context_processors.py` 文件中，其源码如下：

```
def debug(request):
    context_extras = {}
    # 当前的项目处于 DEBUG 模式且请求的 IP 地址位于 INTERNAL_IPS 中
    if settings.DEBUG and request.META.get('REMOTE_ADDR') in settings.INTERNAL_IPS:
    # 用于标记当前处于 DEBUG 模式
    context_extras['debug'] = True
    from django.db import connections
    # 数据库的查询记录
    context_extras['sql_queries'] = lazy(
        lambda: list(itertools.chain.from_iterable(
            connections[x].queires for x in connections)), list)
```

`context_extras` 字典中包含两个 `key`：

- `debug` 标记当前处于调试环境；
- `sql_queries` 是一个列表，其中的每一个元素都是字典对象，结构为 `{'sql':...,'time':...}`，记录一次请求发生的 `SQL` 查询和每次查询耗时。

2.django.template.context_processors.request

`request` 处理器可以将当前HTTP请求对象（`HttpRequest`）传递到模板中，它定义于 `django/template/context_processors.py` 文件中，源码实现非常简单：

```
def request(request):
    return {'request': request}
```

其只是将传递进来的 `HttpRequest` 对象原样返回，并设定 `key` 为 `request`。

3.django.contrib.auth.context_processors.auth

`auth` 处理器定义于 `django/contrib/auth/context_processors.py` 文件中，它的源码如下：

```
def auth(request):
    if hasattr(request, 'user'):
        user = request.user
    else:
        from django.contrib.auth.models import AnonymousUser
```

```
            user = AnonymousUser()
        return {
            'user': user,
            'perms': PermWrapper(user),
        }
```

`auth` 处理器会返回两个变量。

（1）`user`：从 `request` 中获取当前登录的用户，如果处于未登录状态，则返回匿名用户（`AnonymousUser`）。

（2）`perms`：`PermWrapper` 实例，标识当前用户所拥有的权限。

4.django.contrib.messages.context_processors.messages

`messages` 处理器定义于 `django/contrib/messages/context_processors.py` 文件中，源码实现如下：

```
def messages(request):
    return {
        'messages': get_message(request),
        'DEFAULT_MESSAGE_LEVELS': DEFAULT_LEVELS,
    }
```

`messages` 是通过 `Django` 消息框架设置的**消息列表**；`DEFAULT_MESSAGE_LEVELS` 是**消息级别名称到对应数值的映射**。

`Django` 默认对模板系统的配置对于一个简单的项目开发已经足够用了，特别是模板的匹配规则定义（`APP_DIRS` 设置为 `True`），将模板文件的存储位置设计得非常清晰。内置的上下文处理器包含了很多有用的信息，如可以直接从 `auth` 处理器中拿到登录用户，简化了视图函数的上下文传递。因此，在决定修改这些配置之前，一定要清楚当前的操作是否会比 `Django` 推荐的更好。

将模板应用到视图中

模板生成的 `HTML` 文档需要通过视图的返回才能在浏览器中显示出来，因此，模板系统不能离开视图。下面就来看一看怎样将模板应用到视图中。

第6章中定义的第一个视图函数（`hello_django_bbs`）**直接将 `HTML` 代码写在了函数定义中，使得 `HTML` 与 `Python` 之间形成了耦合。**下面，使用 `Django` 的模板系统改写这一行为，先来看第一个改写方式：

```
from django.http import HttpResponse
from django.template import Template, Context

def hello_django_bbs(request):
    t = Template('<h1>Hello {{ project }}</h1>')
    c = Context({'project': 'Django BBS'})
    html = t.render(c)
    return HttpResponse(html)
```

里虽然使用了模板系统，**但是 `HTML` 依然存在于视图函数的定义中**，仍然没有解决刚刚提到的问题。所以，更好的办法是分开定义。

在 `post` 应用的 `templates/post` 目录下创建 `hello_django_bbs.html` 文件，内容就是传递到 `Template` 对象中的字符串：

> `template_string`，模板字符串

```
<h1>Hello {{ project }}</h1>
```

定义了模板文件之后，再次改写 `hello_django_bbs` 视图函数：

```
from django.template.loader import get_template

def hello_django_bbs(request):
    t = get_template('post/hello_django_bbs.html')
    html = t.render({'project': 'Django BBS'})
    return HttpResponse(html)
```

`get_template` 方法用于**从文件系统中加载模板，只需要传递模板路径信息，并返回 `Template` 对象实例。**需要注意的是，`render` 方法中传递的是字典，而不是 `Context` 对象。因为这里的 `Template` 位于 `django/template/backends/django.py` 文件中，需要与之前见到的 `Template` 区别对待。

> 团子注：两个 `Template` 不一样。前面的是 `from django.template import Template` 得来的。

下面简单地介绍 `get_template` 方法的实现过程：

```
def get_template(template_name, using=None):
    chain = []
    # 获取当前项目中配置的模板引擎
    engines = _engine_list(using)
    # 依次遍历各个引擎，尝试加载模板文件
    for engine in engines:
        try:
            return engine.get_template(template_name)
        except TemplateDoesNotExist as e:
            chain.append(e)
    # 最终没能找到模板文件，抛出 TemplateDoesNotExist 异常
    raise TemplateDoesNotExist(template_name, chain=chain)
```

`engines` 是一个列表，代表当前项目中配置的模板引擎。如果使用的是 `Django` 的默认配置，那么列表中只有一个元素，即 `DjangoTemplates` 实例。

接下来，调用 `DjangoTemplates` 的 `get_template` 方法，源码如下所示：

```
def get_template(self, template_name):
    try:
        return Template(self.engine.get_template(template_name), self)
    except TemplateDoesNotExist as exc:
        reraise(exc, self)
```

在这里返回了 `Template` 对象，可注意到，`Template` 接受两个参数：第一个参数是通过引擎匹配并加载得到的模板对象实例，第二个参数是 `DjangoTemplates` 实例自身。引擎的匹配过程暂时不去分析，先来看一看 `Template` 对象的初始化方法和 `render` 方法的实现过程：

```
class Template:
    def __init__(self, template, backend):
        self.template = template
        self.backend = backend
```

```
    def render(self, context=None, request=None):
        context = make_context(context, request, autoescape=self.backend.engine.autoesca
pe)
        try:
            return self.template.render(context)
        except TemplateDoesNotExist as exc:
            reraise(exc, self.backend)
```

> 团子注：这个 `Template` 不是 `from django.template import Template` 的这个，两者不一样。

初始化方法中简单地将传递进来的参数赋值给了 `template` 和 `backend` 变量。所以，其核心实现就在 `render` 方法中。

`render` 方法首先通过 `make_context` 获取到 `Context` 对象实例，其实现如下：

```
def make_context(context, request=None, **kwargs):
    # context 不为 None 则必须是字典类型
    if context is not None and not isinstance(context, dict):
        raise TypeError('context must be a dict rather than %s.' % context.__class__.__n
ame__)
    if request is None:
        context = Context(context, **kwargs)
    else:
        ...
    return context
```

这里也就解释了为什么要给 `hello_django_bbs` 视图中的 `render` 方法传递字典。最后，可以看出，`render` 方法的返回过程与第一次改写 `hello_django_bbs` 视图的实现是一样的。

> 团子注：都是拿到 `context`，然后传递给一个 `template` 实例。

第6章中介绍过 `Django` 提供的快捷方法 `render`，它可以将视图的实现过程变得更为简单。例如，下面使用 `render` 方法再次改写 `hello_django_bbs`：

```
def hello_django_bbs(request):
    return render(request, 'post/hello_django_bbs.html', {'project': 'Django BBS'})
```

可以看到，这次的实现比之前的方式要简单许多。那么，`render` 快捷方法是怎么做到的呢？`render` 方法的实现过程如下：

```
def render(request, template_name, context=None, content_type=None, status=None, using=N
one):
    content = loader.render_to_string(template_name, context, request, using=using)
    return HttpResponse(content, content_type, status)
```

`render` 方法最终会返回 `HttpResponse` 对象，**这是 `Django` 对视图函数最基本的要求**。所以，其中的 `content` 应该就是模板内容了。

`content` 是由 `render_to_string` 方法返回的，它定义于 `django/template/loader.py` 文件中，其实现源码如下：

```
def render_to_string(template_name, context=None, request=None, using=None):
    # 如果 template_name 传递的是列表或元组（即多个模板的路径声明）
    if isinstance(template_name, (list, tuple)):
        template = select_template(template_name, using=using)
    else:
        # 调用 get_template 方法获得 Template 对象
        template = get_template(template_name, using=using)
    return template.render(context, request)
```

> 团子注：`isinstance` 居然还可以这样用！第二个参数传元组。难道传递列表也可以吗？
> 查了一下，第二个参数可以传一个类，或者传第一个元组。当传递一个元组的时候，等价于多个 `isinstance` 用 `or` 连接。

最终，`render_to_string` 还是调用了 `get_template` 方法，获取了 `Template` 对象，并调用其 `render` 方法返回模板内容。即使是传递了列表或元组对象，`select_template` 获取 `Template` 的过程也是使用了 `get_template` 方法。读者有兴趣的话，可以查看 `select_template` 的源码实现，这里就不做具体的分析了。

> 团子注：使用 `render` 快捷方法，内部实现还是调用了 `get_template` 方法，然后给它传递 `context`，调用它的 `render`。

最后，需要注意，`render_to_string` 方法的 `request` 参数并不为空，所以，`template.render` 中的 `request` 参数也不为空。这个条件会影响 `make_context` 方法（`render` 中会调用）的返回值类型：对于 `request` 不为空的情况，实际返回的是 `Context` 的子类 `RequestContext`。

> 团子注：这个在上面的源码当中并没有写出来。

RequestContext和上下文处理器

`RequestContext` 是 `Context` 对象的子类，定义于 `django/template/context.py` 文件中。它与普通的 `Context` 相比，主要有以下两个区别。

（1）实例化需要一个 `HttpRequest` 对象，并作为第一个参数。

（2）根据模板引擎配置的 `context_processors` 自动填充上下文变量（合并到上下文字典中），且内置的 `django.template.context_processors.csrf` 处理器总是会被加入进去。

每个处理器按照顺序依次填充上下文字典，如果存在两个处理器返回了同样的 `key`，那么第二个处理器会覆盖第一个处理器的变量。这同时也说明了**在视图中传递的 `dict` 中的 `key` 有可能会被处理器覆盖，因此，应该尽量避免使用同名的 `key`**。

还可以给 `RequestContext` 提供额外的处理器，即传递 `processors` 参数。为了说明这个参数的使用方法，接下来，自定义一个处理器函数（为了简化演示，将其直接定义在 `post` 应用的 `views.py` 文件中）：

> 团子注：前面说过，**上下文处理器就是一个接收 `request`，返回字典的函数。**

```
def project_signature(request):
    return {'project': 'Django BBS'}
```

`project_signature` 处理器的实现非常简单：按照 `Django` 的规定，接受一个 `HttpRequest` 类型的参数，并返回一个字典对象。那么，当 `RequestContext` 启用了这个处理器后，就不需要主动地在上下文中填充 `project` 了。

例如，可以将 `hello_django_bbs` 视图修改为：

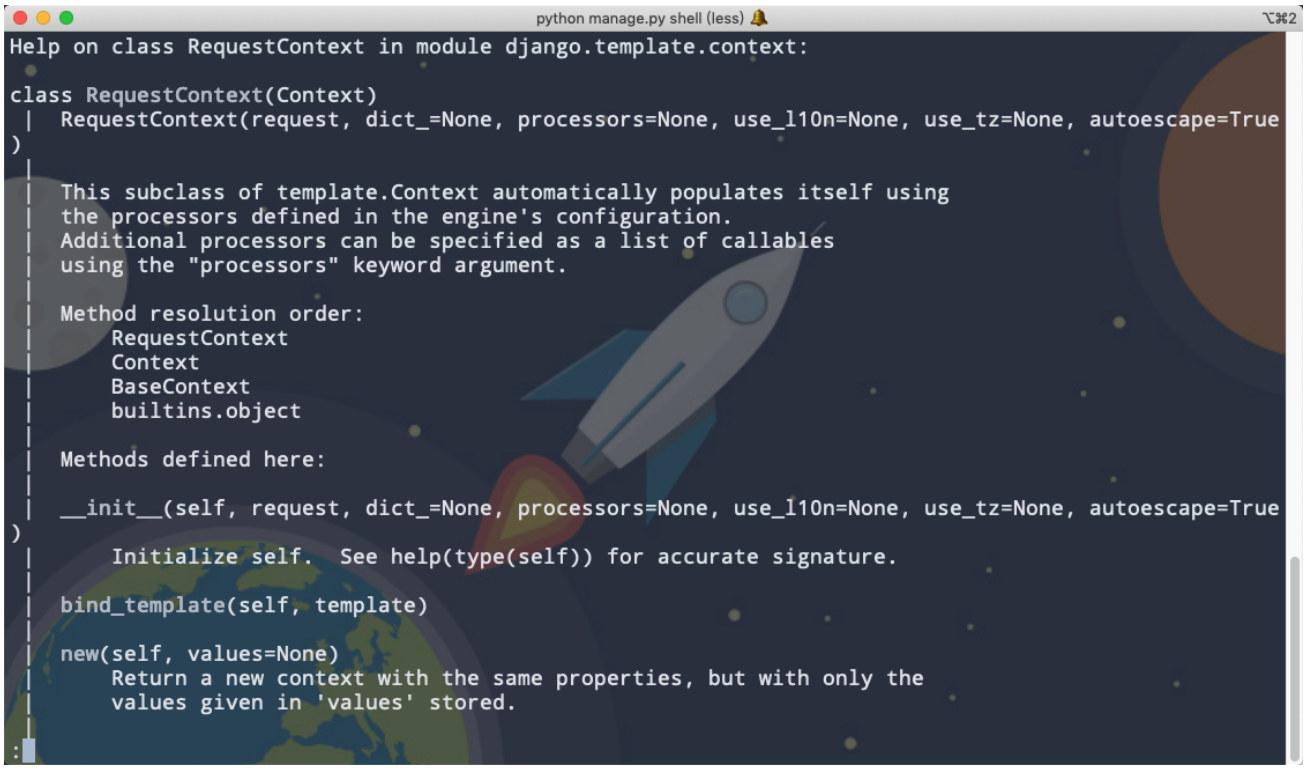
```
from django.template import RequestContext
def hello_django_bbs(request):
    t = Template('<h1>Hello {{ project }}, {{ user.username }}</h1>')
    c = RequestContext(request, processors=[project_signature])
    html = t.render(c)
    return HttpResponse(html)
```

模板中不仅声明了 `project` 变量，还多了一个 `user.username` 变量，传递给 `render` 方法的不再是 `Context` 对象，而是 `RequestContext` 实例。`RequestContext` 中并没有显式地指定上下文字典，只是声明了启用 `project_signature` 处理器，所以，`project` 变量对应的值即为 `Django BBS`。

根据之前所说，**模板引擎中配置的处理器都会用于填充上下文字典**，所以，`user` 对象来自 `django.contrib.auth.context_processors.auth` 处理器。

如果当前的登录用户是 `admin`，那么，访问 `hello_django_bbs` 视图，返回的内容即为：

```
Hello Django BBS, admin
```



> 团子注：我查看了一下 `RequestContext` 的帮助文档。

```
RequestContext(request, dict_=None, processors=None, use_l10n=None, use_tz=None, autoesc
ape=True)
```

> 首先它会使用引擎中配置的上下文处理器生成一波上下文，此外还可以用列表的形式指定 `processors` 关键字参数，列表中放可调用对象。

自定义上下文处理器是很常见的需求，而且**通常许多模板都会使用到同一个处理器**。如果按照之前的实现方式，则需要在每一个视图中指定 `processors` 参数，这会非常麻烦。考虑到 `context_processors` 中配置的处理器在所有的模板中都可用，那么，是否可以将自定义的处理器也配置到 `context_processors` 中实现全局可用呢？答案是肯定的，可以将 `TEMPLATES` 变量中的 `context_processors` 修改如下：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        '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',
                'post.views.project_signature',
            ],
        },
    },
]
```

由此，`hello_django_bbs` 视图中不需要再去指定 `project_signature` 处理器了，修改实现如下所示：

```
def hello_django_bbs(request):
    return render(request, 'post/hello_django_bbs.html')
```

介绍了上下文处理器的定义规则和使用方法之后，最后再介绍自定义上下文处理器时需要注意的地方。
第一，上下文处理器中不应该有复杂的业务处理逻辑，应该有易用和通用的性质。
第二，可以在项目的任何文件、任何位置定义上下文处理器，就像之前在views.py文件中定义 `project_signature` 一样。但是，**`Django` 的建议是在项目或者应用下创建一个 `context_processors.py` 文件（参考内置处理器的定义规则），用于保存上下文处理器。**
第三，由于同名的 `key` 可能会被其他的处理器覆盖，所以，应该保证处理器返回的字典中的 `key` 不存在重复。

将模板应用到视图中是非常简单的，视图函数处理自身的业务逻辑，并通过上下文字典传递属性值用于模板渲染。

07.2 模板系统语法

模板是文本文件，即它可以是**任何基于文本的文件格式**，如HTML、CSV、TXT等。`Django` 模板语言的语法包括了四个部分：变量、标签、过滤器和注释。

模板变量与替换规则

`Django` 对变量的命名限制较少，它可以是**任何字母、数字和下画线的组合**，但是变量名称中不可以有空格或标点符号。

> 团子注：应该开头也不能是数字吧。

最简单的变量替换规则即直接获取上下文字典中的同名 `key`，例如：

```
>>> from django.template import Context, Template
>>> t = Template('Hello: {{ project }}')
>>> c = Context({'project': 'Django BBS'})
>>> t.render(c)
'Hello: Django BBS'
```

变量名中的点是比较特殊的，如 `{{user.username}}`，这会让 `Django` 按照如下的顺序在上下文中查找变量。

（1）**字典查询**：如{{a.b}}查询a['b']。

（2）**属性查询**：如{{a.b}}查询a.b。

（3）**方法调用**：如{{a.b}}调用a.b()。

（4）**数字索引查询**：如{{a.1}}查询a[1]。

模板系统按照顺序依次查找，直到找到第一个可用的值完成变量替换。

1.字典查询

如果给上下文字典中传递了一个字典对象，那么，想要引用这个字典对象中的 `value`，就需要用点号去指向字典对应的 `key`。例如：

```
>>> t = Template('Hello: {{ user.username }}')
>>> user = {'username': 'admin'}
>>> c = Context({'user': user})
>>> t.render(c)
'Hello: admin'
```

2.属性查询

像 `Python` 中的对象实例一样，在模板中也可以通过点号去访问对象的属性。下面看一个例子：

```
>>> import datetime
>>> d = datetime.date(2018, 10, 2)
>>> t = Template('Hello: {{ date.year }}')
>>> c = Context({'date': d})
>>> t.render(c)
'Hello: 2018'
```

自定义的类对象，在模板中同样可以使用点去引用属性。例如：

```
>>> class A:
...     def __init__(self, x):
...         self.x = x
...
>>> t = Template('Hello: {{ a.x }}')
>>> c = Context({'a': A('Django BBS')})
>>> t.render(c)
'Hello: Django BBS'
```

3.方法调用

在模板中访问对象的方法也是可以的，与 `Python` 中不同的是，模板中的方法调用不需要使用括号，且**只可以调用不带参数（不可以有必需的参数）的方法。**

首先，定义对象 `A`，如下所示：

```
class A:
    def x(self):
        return 'Django'
    def y(self, v='BBS'):
        return v
```

对象A中定义了两个方法： `x` 不带参数，所以可以在模板中被调用； `y` 虽然定义了一个参数，但是提供了默认值，所以，仍然可以在模板中被调用。

下面给出一个在模板中调用方法的简单例子：

```
>>> t = Template('Hello: {{ a.x }} {{ a.y }}')
>>> c = Context({'a': A()})
>>> t.render(c)
'Hello: Django BBS'
```

如果模板调用的方法中抛出异常会怎么样呢？

```
>>> class A:
...     def z(self):
...         raise Exception('z error')
>>> t = Template('Hello: {{ a.z }}')
>>> c = Context({'a': A()})
>>> t.render(c)
...
Exception: z error
```

异常会向上传递，并对外抛出。如果想隐藏异常，让模板正常返回，那么需要异常类中存在 `silent_variable_failure` 属性，且值为 `True`：

```
class CustomException(Exception):
    silent_variable_failure = True
```

尝试在对象 `A` 的 `z` 方法中抛出 `CustomException` 异常，并在模板中调用：

```
>>> class A:
...     def z(self):
...         raise CustomException('z error')
...
>>> t = Template('Hello: {{ a.z }}')
>>> c = Context({'a': A()})
>>> t.render(c)
'Hello: '
```

可以看到，这样不会向外抛出异常，取而代之的是用空字符串（实际是模板引擎中的 `string_if_invalid` 属性，默认是空字符串）替换变量值。

由于在模板中可以调用对象的方法，所以，对于 `Model` 实例对象的 `save`、`delete` 等方法也可以在模板中执行。但是，这样的行为是非常危险的， `Django` 为了避免这些方法应用到模板中，提供了 `alters_data` 属性。如果方法

中存在 `alters_data` 属性，且值为 `True`，那么，模板引擎不会执行这个方法，同样会使用 `string_if_invalid` 替换。所以，对于`save`、`delete` 这样的方法，`Django` 都给它们指定了 `alters_data=True`。

> 团子注：`alters_data` 这个名字，翻译过来就是改变数据，也就是说这个变量表示某个方法是否会改变数据。

如果不希望自定义对象中的方法在模板中被调用，可以设置 `alters_data` 属性，例如：

```
>>> class A:
...     def z(self):
...         return 'Django BBS'
...     z.alters_data = True
...
>>> t = Template('Hello: {{ a.z }}')
>>> c = Context({'a': A()})
>>> t.render(c)
'Hello: '
```

4.数字索引查询

可以给上下文中传递列表或元组，在模板中使用索引的方式获取到对应的值。`Python` 列表的索引从0开始，所以，第一个元素的索引值就是 `0`。下面看一个简单的例子：

```
>>> t = Template('Hello: {{ item.0 }} {{ item.1 }}')
>>> c = Context({'item': ['Django', 'BBS']})
>>> t.render(c)
'Hello: Django BBS'
```

需要注意的是，模板中不可以使用负数索引，如 `{{item.-1}}` 传递给 `Template` 会抛出 `TemplateSyntaxError` 异常。

如果模板中定义的变量在上下文中没有提供，那么模板系统会怎样处理这个问题呢？

```
>>> t = Template('Hello: {{ Project }}')
>>> c = Context({'project': 'Django BBS'})
>>> t.render(c)
'Hello: '
```

> 团子注：模板语言区分大小写！

模板中定义了变量 `{{Project}}`，注意，首字母是大写的。但是上下文字典中只有小写的 `project`，最终，引擎渲染模板时使用了 `string_if_invalid` 替换模板变量。

从这个例子中可以看出，如果模板中的变量没有在上下文中提供，则模板系统并不会抛出异常，而是会使用空字符串替换。同时，也可以从中得出结论，模板变量是大小写敏感的。

模板标签

`Django` 模板系统对标签的解释是**在渲染的过程中提供任意的逻辑**，它看起来像 `{%tag%}` 这样。标签常常用于在输出时创建文本、控制循环和判断逻辑以及装载外部信息。

1.判断执行逻辑的if标签

`if` 标签判断条件是否成立，如果成立则显示块中的内容。`if` 需要与 `endif` 成对出现，且它与 `Python` 中的 `if`、`elif` 和 `else` 用法是类似的。使用形式如下：

```
{% if variable %}
    ...
{% elif variable %}
    ...
{% else %}
    ...
{% endif %}
```

`variable` 即需要比对的条件，可以使用 `and` 和 `or` 连接多个条件，也可以使用 `not` 对当前的条件取反。同时，也可以在条件中使用 `>`、`==`、`<=` 等算术操作符。

`elif` 和 `else` 这两个标签是可选的，可以不提供。同时，`elif` 可以不止一个，用来对多种条件进行判断。

下面，看一个简单的例子，介绍 `if` 标签的使用方法：

```
>>> t = Template("""
...     {% if user.username == 'admin' %}
...         <p>Hello {{ user.username }}</p>
...     {% elif user.username == 'Admin' %}
...         <p>Welcome {{ user.username }}</p>
...     {% else %}
...         <p>AnonymousUser</p>
...     {% endif %}
...     """)
>>> c = Context({'user': {'username': 'admin'}})
>>> t.render(c)
' <p>Hello admin</p> '
```

当需要判断的条件太多时，为了避免过多连接操作符的出现，影响阅读，可以考虑使用嵌套的 `if` 标签。例如：

```
>>> t = Template("""
...     {% if user.username == 'admin' %}
...         {% if print %}
...             <p>Hello {{ user.username }}</p>
...         {% else %}
...             <p>Nothing</p>
...         {% endif %}
...     {% endif %}
...     """)
>>> c = Context({'user': {'username': 'admin'}, 'print': True})
>>> t.render(c)
' <p>Hello admin</p> '
```

2.迭代序列元素的for标签

`for` 标签用于对列表或元组中的元素进行迭代，它与 `Python` 中的 `for` 语法是类似的。同时，它需要与结束标签 `endfor` 配合使用。`for` 标签支持一个可选的 `empty` 子句，用于定义当列表不存在或没有元素时显示的内容。使用形式如下：

```
{% for varibale in list %}
    ...
{% empty %}
    ...
{% endfor %}
```

```
>>> t = Template("""
...     {% for item in signature %}
...         <li>{{ item }}</li>
...     {% empty %}
...         <p>Nothing</p>
...     {% endfor %}
...     """)
>>> c = Context({'signature': ['Django', 'BBS']})
>>> t.render(c)
'<li>Django</li><li>BBS</li>'
```

如果没有传递 `signature` 或者它是一个空列表，则会显示 `Nothing`。
可以在 `for` 标签中添加 `reversed` 关键字，实现对列表元素的逆序迭代。

```
>>> t = Template("""
...     {% for item in signature reversed %}
...         <li>{{ item }}</li>
...     {% empty %}
...         <p>Nothing</p>
...     {% endfor %}
...     """)
>>> c = Context({'signature': ['Django', 'BBS']})
```

```
>>> t.render(c)
'<li>BBS</li><li>Django</li> '
```

与 `Python` 中的 `for` 循环不同的是，`for` 标签只能一次性地遍历完列表中的元素，不能中断（break），也不能跳过（continue）。这就要求传递到上下文中的变量包含确定需要的值，即**用业务逻辑控制列表中的元素**。

与 `if` 类似，`for` 标签同样可以嵌套使用。

在 `for` 标签的内部，可以通过访问 `forloop` 变量的属性获取迭代过程中的一些信息。

`forloop.counter` 从 `1` 开始计数。除此之外，`forloop` 还包含如下一些属性。

（1）`counter0`：与 `counter` 一样用来计数，但是它从 `0` 开始。

（2）`revcounter`：用来表示当前循环中剩余元素的数量。第一次迭代时，返回的是列表中元素的总数，最后一次的返回值是 `1`。

（3）`revcounter0`：与 `revcounter` 的含义相同，但是由于其索引是基于 `0` 的，因此它的值等于 `revcounter` 减去 `1`。

（4）`first`：返回一个布尔值，`True` 标识为当前迭代的是第一个元素，其他位置的元素返回 `False`。

（5）`last`：也是一个布尔值，迭代最后一个元素时返回 `True`，其他情况为 `False`。

（6）`parentloop`：对于嵌套迭代的场景，用来引用父级循环的 `forloop` 变量。

需要注意，`forloop` 只可以在 `for` 与 `endfor` 之间使用。对于 `first` 和 `last` 属性，可以通过 `if` 标签对特定的元素做特殊处理，而像 `counter` 这类属性，常常用于调试程序。

3.获取视图访问地址的url标签
与 `reverse` 函数功能类似，使用 `url` 标签可以**避免在模板中对访问地址进行硬编码**，即使是将来修改了视图的访问地址，也可以不用修改模板定义。使用形式如下：

```
{% url ns:name arg1,arg2... %}
```

> 团子注：`arg1, arg2` 这里是有逗号吗？

其中，`ns` 是视图的命名空间，`name` 是视图的名称。如果需要，还可以给定参数构造动态的 `url`。

之前并没有给 `hello_django_bbs` 视图指定 `name`，现在修改它的 `URL` 模式定义：
`path('hello/', views.hello_django_bbs, name='hello'),`

此时，可以利用 `url` 标签获取到 `hello_django_bbs` 的访问地址，例如：

```
>>> t = Template("{% url 'post:hello' %}")
>>> t.render(Context())
'/post/hello/1/'
```

对于需要给定参数构造动态 `url` 的视图，例如：

```
path('topic/<int:topic_id>/', views.topic_detail_view, name='topic_detail')
```

可以在 `url` 标签中指定参数：

```
>>> t = Template("{% url 'post:topic_detail' 1 %}")
>>> t.render(Context())
'/post/topic/1/'
```

或者也可以指定参数的名称：

```
>>> t = Template("{% url 'post:topic_detail' topic_id=1 %}")
```

> 团子注：这里的 `topic_id=1` 中间可以有空格吗？

4.用于多行注释的comment标签

在模板中注释内容需要使用 `{##}`，如 `{#Django BBS#}`，引擎不会对注释的内容进行解释。但很多情况下，由于要解释当前的代码执行流程或逻辑，故可能导致注释的内容比较多，注释内容会分散为多行。

为了解决这个问题，模板系统提供了 `comment` 标签。使用形式如下：

```
{% comment %}
...
{% endcomment %}
```

模板系统会忽略 `comment` 与 `endcomment` 之间的内容。但是，需要注意，`comment` 标签不可以嵌套使用，因为这本身就没有意义。

使用 `comment` 标签将多行内容注释的例子如下：

```
>>> t = Template("""
...     <p>Hello</p>
...     {% comment %}
...         Hello
...         Django
...         BBS
...     {% endcomment %}
...     <p>Django BBS</p>
...     """)
>>> t.render(Context())
'<p>Hello</p><p>Django BBS</p>'
```

5.判断变量相等或不相等的标签

判断两个变量的值是否相等或不相等是很常见的需求，其可以通过 `if` 标签配合等于或不等于运算符来完成。`Django` 模板系统为了简化操作过程，提供了 `ifequal` 和 `ifnotequal` 标签，用于判断变量是否相等。

这两个标签的用法是相同的，这里以 `ifequal` 为例介绍它的使用形式：

```
{% ifequal v1 v2 %}
    ...
{% else %}
    ...
{% endifequal %}
```

如果 `v1` 和 `v2` 相等，则执行 `ifequal` 与 `else` 之间的逻辑，否则执行 `else` 与 `endifequal` 之间的逻辑。

```
>>> t = Template("""
...     {% ifequal v1 v2 %}
...         <p>{{ v1 }} equal {{ v2 }}</p>
...     {% else %}
...         <p>{{ v1 }} not equal {{ v2 }}</p>
...     {% endifequal %}
...     """)
>>> c = Context({'v1': 'admin', 'v2': 'admin'})
>>> t.render(c)
'<p>admin equal admin</p>'
>>> c = Context({'v1': 'admin', 'v2': 'Admin'})
>>> t.render(c)
'<p>admin not equal Admin</p>'
```

`v1`、`v2` 除了可以是模板变量，也可以是硬编码的字符串、整数或小数，**但不可以是字典、列表等类型**。

标签可以做任何事情，所以，它实现起来要相对复杂。自定义一个标签可以分为三种类型：**简单标签**（simple tag）、**引入标签**（inclusion tag）和**赋值标签**（assignment tag）。接下来，依次实现自定义这三类标签。

在自定义标签之前，需要做一些准备工作。

（1）在应用（如 `post` 应用）下面创建一个名称为 `templatetags` 的 `Python` 包（包含 `__init__.py` 文件），并在其中新建 `custom_tags.py` 文件。

（2）在 `settings.py` 文件中将应用加入 `INSTALLED_APPS` 列表中，让Django能够扫描这个应用。

一个有效的标签库必须有一个模块层变量 `register`，且它的值是 `template.Library` 的实例。打开 `custom_tags.py` 文件，在其中加入语句：

```
from django import template
```

```
register = template.Library()
```

6.简单标签

这类标签通过接收参数，对输入的参数做一些处理并返回结果。如需要给字符串添加一个前缀，可以在 `custom_tags.py` 文件中定义 `prefix_tag` 标签：

`prefix_tag` 使用 `register.simple_tag` 装饰器修饰，**目的是能够将 `prefix_tag` 注册到模板系统中**。如果想要使用自定义的标签，则需要在模板中使用 `load` 标签声明（装载）：

```
{% load custom_tags %}
```

`custom_tags` 是自定义标签所在的文件名称，之后，`prefix_tag` 就可以像内置标签一样使用了。

```
>>> t = Template("""
...     {% load custom_tags %}
...     {% prefix_tag 'Django BBS' %}
...     """)
>>> t.render(Context())
'Hello Django BBS'
```

如果想要在标签中访问字典上下文，则在注册标签时需要指定 `takes_context=True`，且标签的第一个参数必须是 `context`。例如

```
@register.simple_tag(takes_context=True)
def prefix_tag(contex, cur_str):
    return '%s %s' % (context['prefix'], cur_str)
```

在使用 `prefix_tag` 时需要在 `Context` 中指定 `prefix`：

```
>>> t = Template("""
...     {% load custom_tags %}
...     {% prefix_tag 'Django BBS' %}
...     """)
>>> t.render(Context({'prefix': 'Hello'}))
'Hello Django BBS'
```

此外，如果不想直接使用函数的名称作为标签名，那么还可以使用 `name` 参数给标签起一个别名，例如：

```
@register.simpletag(takes_context=True, name='prefix')
def prefix_tag(contex, cur_str):
    return '%s %s' % (context['prefix'], cur_str)
```

此时，这个标签的名字就变成了 `prefix`。

7.引入标签

这类标签可以被其他模板进行渲染，然后将渲染结果输出。这个概念比较抽象，不易理解，下面举例说明这类标签的用法。

首先，在 `post` 应用中定义模板文件 `inclusion.html`（templates/post/inclusion.html）：

```
<p>{{ hello }}</p>
```

在 `custom_tags.py` 文件中定义 `hello_inclusion_tag` 标签：

```
@register.inclusion_tag('post/inclusion.html', takes_context=True)
def hello_inclusion_tag(context, cur_str):
    return {'hello': '%s %s' % (context['prefix'], cur_str)}
```

可以看到，引入标签使用 `register.inclusion_tag` 注册。第一个参数指定模板文件，即刚刚定义的 `inclusion.html`；第二个参数是为了使用上下文字典。

```
>>> t = Template("""
...     {% load custom_tags %}
...     {% hello_inclusion_tag 'Django BBS' %}
...     """)
>>> t.render(Context({'prefix': 'Hello'}))
'<p>Hello Django BBS</p>'
```

可以看到，`inclusion.html` 在模板中被渲染了。**对于具有通用的展现样式但需要不同数据去渲染的页面，可以考虑使用引入标签。**

> 团子注：**引入标签**就是把别的模板引入到当前模板当中的标签。它可以接收当前模板的上下文，也可以接收当前模板传递的参数。它的返回值是要用在被引入模板当中的 `context` 对象。

8.赋值标签

它与简单标签非常相似，但是结果不被直接输出，而是存储在指定的上下文变量中，目的是**降低传递上下文的成本**。

赋值标签同样使用 `register.simple_tag` 注册，例如，在 `custom_tags.py` 文件中定义：

```
@register.simple_tag
def hello_assignment_tag(cur_str):
    return 'Hello: %s' % cur_str
```

```
>>> t = Template("""
...     {% load custom_tags %}
...     {% hello_assignment_tag 'Django BBS' as hello %}
...     <p>{{ hello }}</p>
...     """)
>>> t.render(Context())
'<p>Hello: Django BBS</p>'
```

可以看到，模板中使用 `hello_assignment_tag` 标签的地方用 `as` 参数将标签的返回结果保存在 `hello` 中，所以，在模板渲染的时候不需要传递 `hello` 到上下文中。

> 团子注：赋值标签其实等**简单标签加上 `as`**，此时简单标签的结果不直接显示，它的结果被赋值给了 `as` 后面的名字，它可以作为标签使用。另外，使用 `as` 其实跟在注册简单标签的时候传递 `name` 参数指定名字是差不多的。

过滤器

过滤器用于在显示变量之前对变量的值进行调整，使用管道符号（`|`）指定。有些过滤器可以接受参数，如果参数中带有空格，则需要用引号括起来。过滤器的特色是可以通过组合多个过滤器实现**链式调用**。

1.获取变量长度的length过滤器

过滤器相比标签要简单许多，**可以认为过滤器就是一个 `Python` 函数，传递参数给它，处理完之后返回到模板中。**

```
>>> t = Template("""
...     <p>hello : {{ hello | length }}</p>
...     """)
>>> t.render(Context({'hello': 'hello'}))
'<p>hello : 5</p>'
```

模板变量 `hello` 使用管道符号连接 `length` 过滤器，最终得到了字符串的长度。如果传递的不是字符串，而是列表，则会返回列表长度；如果是字典则会返回 `key` 的个数；变量未定义的情况下，返回 `0`。

```
>>> t = Template("""
...     <p>hello : {{ hello | length }}</p>
...     """)
>>> t.render(Context({'hello': ['Django', 'BBS']}))
'<p>hello : 2</p>'
>>> t.render(Context({'hello': {'v1': 1, 'v2': 2}}))
'<p>hello : 2</p>'
>>> t.render(Context())
'<p>hello : 0</p>'
```

2.转换字符大小写的过滤器

`lower` 和 `upper` 用于将字符串转换为小写和大写的形式。

```
>>> t = Template("""
...     <p>hello : {{ hello | lower }}</p>
...     """)
>>> t.render(Context({'hello': 'Django BBS'}))
'<p>hello : django bbs</p>'
```

`lower` 将 `Django BBS` 的所有字符变成了小写。同样，如果要把所有的字符变成大写，只需替换成 `upper`：

```
>>> t = Template("""
...     <p>hello : {{ hello | upper }}</p>
...     """)
>>> t.render(Context({'hello': 'Django BBS'}))
'<p>hello : DJANGO BBS</p>'
```

3.获取首个或末尾元素的过滤器

`first` 和 `last` 过滤器用于获取变量的首个或末尾元素。对于不同的变量类型：字符串会返回第一个或最后一个字符、列表或元组会返回第一个或最后一个元素。例如：

```
>>> t = Template("""
...     <p>hello : {{ hello | first }}</p>
...     """)
>>> t.render(Context({'hello': 'Django BBS'}))
'<p>hello : D</p>'
>>> t.render(Context({'hello': ['Django', 'BBS']}))
'<p>hello : Django</p>'
```

可以通过配合使用 `lower` 或 `upper` 将字符串转换为小写或大写样式，即链式调用：

```
>>> t = Template("""
...     <p>hello : {{ hello | last | lower }}</p>
...     """)
>>> t.render(Context({'hello': ['Django', 'BBS']}))
'<p>hello : bbs</p>'
```

模板变量 `hello` 首先通过 `last` 过滤器得到列表的最后一个元素 `BBS`，再通过 `lower` 过滤器将字符串变成小写样式，最终得到 `bbs`。

4.truncatewords过滤器截取指定个数的词

`truncatewords` 过滤器接受一个参数，指定需要保留的单词个数，多出来的词用省略号替换。参数需要与过滤器用冒号分开，例如：

```
>>> t = Template("""
...     <p>hello : {{ hello | truncatewords:1 }}</p>
...     """)
>>> t.render(Context({'hello': 'Django BBS'}))
'<p>hello : Django ...</p>'
```

自定义过滤器与自定义标签需要做同样的准备工作，即模块层变量 `register` 和应用装载到项目环境中，这在自定义标签的时候已经完成了。接下来需要做的是实现过滤器函数与过滤器注册。

> 团子注：**自定义标签**和**自定义过滤器**差不多。那么两者的区别是什么？
>
> - 标签使用 `{% %}`，过滤器使用 `|`
> - 自定义标签使用 `@register.xxx_tag` 来注册，而自定义过滤器使用 `@register.filter` 来注册

在 `custom_tags.py` 文件（在自定义标签时创建）中创建 `replace_django` 过滤器：

```
@register.filter
def replace_django(value):
    return value.replace('django', 'Django')
```

`replace_django` 函数**接受一个参数，这个参数即为模板变量**，在函数内部实现对 `django` 字符串到 `Django` 的替换。最后，还需要使用 `register.filter` 实现对过滤器的注册。

在模板中使用自定义的过滤器同样需要 `{%load custom_tags%}`，例如：

```
>>> t = Template("""
...     {% load custom_tags %}
...     <p>hello : {{ hello | replace_django }}</p>
...     """)
>>> t.render(Context({'hello': 'django BBS'}))
'<p>hello : Django BBS</p>'
```

如果不想使用函数名作为过滤器的名称，可以在 `register.filter` 中使用 `name` 为过滤器指定名称。实现带参数的过滤器也非常简单，即过滤器函数需要两个参数：

```
@register.filter(name='r_django')
def replace_django(value, base):
    return value.replace('django', base)
```

由于使用了 `name` 指定了过滤器的名称，所以，模板中也需要使用 `r_django`，并传递所需参数：

```
>>> t = Template("""
...     {% load custom_tags %}
...     <p>hello : {{ hello | r_django:"Django" }}</p>
...     """)
>>> t.render(Context({'hello': 'django BBS'}))
'<p>hello : Django BBS</p>'
```

过滤器的思想与使用方法非常简单，但用处非常大。特别是**多个过滤器可以通过管道实现连接**，很大程度上方便了对变量的处理过程。

模板继承
一些高级语言有继承的功能，将通用的功能或属性写在父类（或基类）里面，子类继承自父类，即自动拥有父类的所有属性和方法。同时，还可以通过重写父类中的属性和方法实现定制。这样的继承特性，通过抽象共性，减少了大量的重复代码。

模板继承使用起来非常简单，**只需要定义好被继承的父模板，其中包含通用元素和可以被子模板覆盖的 `block` 部分即可。**

在 `post` 目录（`post/templates/post/`）下创建父模板文件 `base.html`：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Welcome{% endblock %}</title>
  </head>
  <body>
    <h1>Hello Django BBS</h1>
    {% block content %} Nothing {% endblock %}
    <hr>
    {% block footer %}
      <p>Thanks for visiting my site.</p>
    {% endblock footer %}
  </body>
</html>
```

可以看到，这个父模板文件中使用了 `{%block%}` 标签，这就是刚刚提到过的**可以被子模板覆盖的 `block`**。另外，需要说明以下几点。

（1）`block` 标签成对出现，需要 `{%endblock%}` 标记结束。

（2）需要给 `block` 标签起个名字，子模板中具有同样名称的 `block` 块完成对父模板的替换。

（3）子模板不需要定义父模板中的所有 `block`，**未定义时，子模板将原样使用父模板中的内容。**

（4）子模板需要使用 `{%extends%}` 标签继承父模板，**且其必须是模板中的第一个标签，通常继承声明会放在文件的第一行。**

接下来，实现一个子模板，展示 `Topic` 或 `Comment` 的 `id` 和 `content`。继承 `base.html`，并通过 `Template` 对象实例化：

```
>>> t = Template("""
...     {% extends "post/base.html" %}
...     {# 页面 Title #}
...     {% block title %} Model Content {% endblock %}
...     {# 页面内容 #}
...     {% block content %}
...     {% for item in model %}
...         <p>{{ item.id }}: {{ item.content }}</p>
...     {% endfor %}
...     {% endblock %}
...     {# 页面结尾 #}
...     {% block footer %}
...        <p>Thanks for visiting {{ name }} Content.</p>
...     {% endblock %}
...     """)
```

可注意到，子模板重写了父类中的三个 `block`，且需要两个模板变量：`model` 和 `name`。下面传递上下文渲染模板对象：

```
>>> from post.models import Topic
>>> topic_qs = Topic.objects.all()
>>> t.render(Context({'model': topic_qs, 'name': 'Topic'}))
```

`render` 方法渲染返回的结果如下所示

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title> Model Content </title>
    </head>
    <body>
        <h1>Hell Django BBS</h1>
        <p>1: This is the first topic!</p>
        <p>2: This is the second topic!</p>
        <p>3: This is the third topic!</p>
        <hr>
        <p>Thanks for visiting Topic Content.</p>
    </body>
</html>
```

可以看到，子模板继承了父模板的结构，并对其中的 `block` 部分完成了替换。

但是，这里需要考虑一个问题：假如不是替换，而是对父模板中的 `block` 添加一些内容的话，需要怎么做呢？这种场景也是很常见的，即**对父模板内容的扩展**。

`Django` 为此提供了 `{{block.super}}` 变量，可以**获取到父模板中渲染后的结果**。举个例子，将 `base.html` 的内容修改为：

```
<html>
     <body>
         {% block hello %} hello {% endblock %}
     </body>
</html>
```

创建子模板，继承自 `base.html`，并通过 `Template` 对象实例化：

```
>>> t = Template("""
...     {% extends "post/base.html" %}
...     {% block hello %}
...     {{ block.super }}Django BBS
...     {% endblock %}
...     """)
>>> t.render(Context())
```

`render` 方法渲染后的结果为：

```
<html>
    <body>hello Django BBS</body>
</html>
```

模板继承是模板系统中最强大也是最复杂的功能，它的用途非常广泛。通常，**如果是在多个模板中出现了大量的重复代码，那么，就应该考虑使用模板继承来减少重复性代码。**另外，`Django` 建议，父模板中的 `{%block%}` 标签应该越多越好，毕竟，子模板不需要完全重写所有的标签，所以，**可以多定义一些通用或者默认的内容。**

07.3 模板系统工作原理分析

模板引擎根据配置加载模板文件（`HTML` 文件），是怎么去查找加载的呢？模板引擎完成对模板的渲染，包括变量替换、执行过滤器等，这又是怎么实现的呢？

模板文件实现加载的过程

在介绍模板系统基础内容的时候，已经看到过在视图中使用 `get_template` 方法（定义于 `django/template/loader.py` 文件中）加载模板。方法内部实现首先会获取到当前系统中可用的模板后端，再去调用后端引擎的 `get_template` 方法。所以，这里的第一个问题是怎样获取到当前系统中的模板后端？

1._engine_list方法获取模板后端

`get_template` 方法中获取模板后端使用了 `_engine_list` 方法，这个方法接受一个 `using` 参数，默认为 `None`。方法实现如下：

```
def _engine_list(using=None):
    return engines.all() if using is None else [engines[using]]
```

该方法返回后端列表，主要实现依赖于 `engines`，它是一个 `EngineHandler` 类型的实例，下面介绍 `EngineHandler` 的定义：

```
class EngineHandler:
    def __init__(self, templates=None):
        self._templates = templates
        self._engines = {}
```

初始化函数可以传递一个 `templates` 参数，它是一个列表，结构与 `settings.TEMPLATES` 类似，通常不需要显式地指定。再看 `EngineHandler` 的 `all` 方法：

```
def all(self):
    return [self[alias] for alias in self]
```

对自身实例进行迭代，并使用 `[]` 获取结果列表。这里涉及对象的 `__iter__` 和 `__getitem__` 两个方法。接下来，首先看 `__iter__` 的实现源码：

```
def __iter__(self):
    return iter(self.templates)
```

这里调用了 `templates` 属性，它是一个用 `@cached_property` 修饰的方法，可以像属性一样使用。实现如下：

```
@cached_property
def tempaltes(self):
    if self._templates is None:
        # settings 配置赋值给 self._templates
        self._templates = settings.TEMPLATES
    # templates 是有序字典类型
    templates = OrderedDict()
    backend_names = []
    # 遍历模板后配置
    for tpl in self._templates:
        tpl = tpl.copy()
        try:
            ...
            default_name = tpl['BACKEND'].rsplit('.', 2)[-2]
        except Exception:
            ...
        tpl.setdefault('NAME', default_name)
        tpl.setdefault('DIRS', [])
        tpl.setdefault('APP_DIRS', False)
        tpl.setdefault('OPTIONS', {})
        # key 为 default_name, value 为模板后端配置
        templates[tpl['NAME']] = tpl
        ...
    return templates
```

所以，对于默认配置的情况，`templates` 的结果是一个字典对象（`OrderedDict`），且只有一个键值对。其中 `key` 为 `django`（default_name），`value` 也是一个字典，与 `settings` 配置中的内容相同，只是多了一个 `NAME` 属性。

下面再来看 `__getitem__` 方法的实现：

```
def __getitem__(self, alias):
    ...
    params = self.templates[alias]
    params = params.copy()
    backend = params.pop('BACKEND')
    engine_cls = import_string(backend)
    # 实例化模板后端
    engine = engine_cls(params)
    ...
    return engine
```

可以看到，这个方法根据传递的 `key` 返回了模板后端实例。

所以，最终 `_engine_list` 方法得到了模板后端列表，且对于默认配置，列表中只有一个元素，即 `DjangoTemplates` 实例。

获取到了后端实例之后，再来考虑第二个问题：模板后端是怎样实现加载模板的?

2.DjangoTemplates加载模板文件

加载模板文件使用 `DjangoTemplates` 的 `get_template` 方法，并传递了 `template_name`（模板名称），源码实现如下：

```
def get_template(self, template_name):
    try:
        return Template(self.engine.get_template(template_name), self)
    except TemplateDoesNotExist as exc:
        reraise(exc, self)
```

`self.engine` 定义在初始化函数中，是 `Engine`（定义于 `django/template/engine.py` 文件中）类型的实例，它的 `get_template` 方法定义如下：

```
def get_template(self, template_name):
    template, origin = self.find_template(template_name)
    ...
    return template
```

`find_template` 方法定义如下：

```
def find_template(self, name, dirs=None, skip=None):
    tried = []
    for loader in self.template_loaders:
        try:
            template = loader.get_template(name, skip=skip)
            return template, template.origin
        except TemplateDoesNotExist as e:
            tried.extend(e.tried)
    raise TemplateDoesNotExist(name, tried=tried)
```

通过对 `self.template_loaders` 进行迭代，并调用它的 `get_template` 方法获取到了 `template` 对象。`template_loaders` 是 `Engine` 中定义的一个方法，利用 `@cached_property` 修饰，其内部调用了 `self.get_template_loaders`，并传递了 `self.loaders`。

> **property**让方法可以像属性那样被访问。

`self.loaders` 在 `Engine` 的初始化函数中完成赋值，过程如下所示：

```
def __init__(self, dirs=None, app_dirs=False, loaders=None ...):
    if loaders is None:
        loaders = ['django.template.loaders.filesystem.Loader']
        if app_dirs:
            loaders += ['django.template.loaders.app_directories.Loader']
        if not debug:
            loaders += [('django.template.loaders.cached.Loader', loaders)]
    ...
    self.loaders = loaders
    ...
```

> 团子注：这就是文档中说 `app_dirs` 默认为 `False` 的原因，但是使用 `django-admin startproject` 生成的项目配置是把 `APP_DIRS` 设置为 `True` 的。
> 可以参看 *Django 模板查找顺序* https://stackoverflow.com/questions/18029547/django-templates-lookup-order
> 上文讨论了当有 `loader` 和不指定 `loader` 的时候，模板的加载顺序有什么不同。

从定义中可以看出，对于默认的配置，`self.loaders` 是一个包含两个元素的列表：`['django.template.loaders.filesystem.Loader','django.template.loaders.app_directories.Loader']`。

将列表传递到 `get_template_loaders` 方法中完成对两个 `Loader` 对象的实例化，所以，最终 `self.template_loaders` 就是两个 `Loader` 实例。

再回到 `find_template` 方法中，依次对 `Loader` 实例进行迭代，**调用它们的 `get_template` 方法**，直到某一个 `Loader` 返回，或最终抛出 `TemplateDoesNotExist` 异常。

> 团子注：可见模板加载器的顺序是 `filesystem.Loader` 然后是 `app_directories.Loader`。最后是 `DIRS`，如果定义了的话。

于默认的配置会使用 `django.template.loaders.app_directories.Loader` 完成模板的加载，所以，下面分析它的实现过程。另外一个 `Loader` 实现原理类似，不做具体介绍。`app_directories.Loader` 定义如下：

```
class Loader(FilesystemLoader):
    # 重写父类的 get_dirs
    def get_dirs(self):
        return get_app_template_dirs('templates')
```

它只定义了一个方法，所以 `get_template` 来自父类（实际是 `FilesystemLoader` 的父类），查看父类中的实现：

```
def get_template(self, template_name, skip=None):
    ...
    for origin in self.get_template_sources(template_name):
        ...
        try:
           contents = self.get_contents(origin)
        except TemplateDoesNotExist:
            continue
        else:
            return Template(
                contents, origin, origin.template_name, self.engine,
            )
    raise TemplateDoesNotExist(template_name, tried=tried)
```

对 `get_template_sources` 的返回进行迭代，直到找到第一个有效的 `contents` 构造 `Template` 对象返回。所以，将模板文件从文件系统中读取，构造 `Template` 实例就是在这里实现的。在看 `get_template_sources` 方法的实现之前，先来看 `get_dirs` 方法返回了什么。`get_dirs` 中调用了 `get_app_template_dirs` 方法。

```
def get_app_template_dirs(dirname):
    template_dirs = []
    # 对当前系统中安装的应用进行迭代
    for app_config in apps.get_app_configs():
        # app_config.path 是应用在文件系统下的路径
        if not app_config.path:
            continue
```

```
            # dirname 传递的是 templates
            template_dir = os.path.join(app_config.path, dirname)
            # 所以，需要在应用下创建 templates 目录存储模板文件
            if os.path.isdir(template_dir):
                template_dirs.append(template_dir)
        return tuple(template_dirs)
```

该方法返回了一个元组，如果应用中包含 `templates` 目录，那么，`templates` 的文件路径将会被加入这个结果元组中。所以，对于 `post` 应用来说，这个方法返回的结果会包含路径 `my_bbs/post/templates`。

> 团子注：这里是按照 `app` 的注册顺序来遍历的。
> `get_dirs` 的返回值就是 `get_app_template_dirs('template')` 的返回值，也就是注册过的 `APP` 下面的 `templates` 文件路径组成的元组。

`get_template_sources` 方法位于 `app_directories.Loader` 的父类中，源码如下：

```
    def get_template_sources(self, template_name):
        # template_dir 遍历各个 `APP/templates` 路径
        for template_dir in self.get_dirs():
            try:
                # 组合 template_dir 与传递进来的 template_name
                name = safe_join(template_dir, template_name)
            except SuspiciousFileOperation:
                continue
            yield Origin(
                name=name,
                template_name=template_name,
                loader=self,
            )
```

从方法实现中可以看出，`get_template_sources` 实际是一个**生成器函数**，`get_dirs` 方法返回的应用下 `templates` 目录再与传递的 `template_name` 拼接得到 `Origin` 的 `name` 属性。如对于传递的 `post/base.html` 来说，迭代生成器时就会包含 `my_bbs/post/templates/post/base.html` 文件路径（也同时会包含其他应用与 `post/base.html` 拼接得到的路径）。

文件路径传递到 `get_contents` 方法中，如果没有抛出异常，则利用返回的结果实例化 `Template` 对象。`get_contents` 方法（同样位于 `app_directories.Loader` 父类中）定义如下：

```
    def get_contents(self, origin):
        try:
            # origin.name 即为模板文件在文件系统中的路径
            with open(origin.name, encoding=self.engine.file_charset) as fp:
                # 这里是一次性读取
                return fp.read()
        except FileNotFoundError:
            raise TemplateDoesNotExist(origin)
```

`get_contents` 方法尝试读取当前传递进来的模板文件路径，如果文件存在，返回文件内容，之后用来构造 `django.template.base.Template` 对象。

最终 `DjangoTemplates` 对象的 `get_template` 中利用这里的模板对象（Template）和自身（Self）构造了 `django.template.backends.django.Template` 对象并返回。

至此，视图中调用的 `get_template` 方法的工作原理介绍完成，这个方法是模板系统的核心方法，在许多地方都会被使用。最后，总结它的实现流程。

（1）查询当前系统中配置的可用模板后端。
（2）对可用模板后端依次进行迭代，尝试加载模板，具体实现过程取决于特定的后端实现。
（3）返回第一个成功加载的模板对象或最终抛出 `TemplateDoesNotExist` 异常。

模板渲染机制实现分析

通过 `get_template` 加载了模板之后，就可以通过它的 `render` 方法对模板进行渲染，这其中包括了变量替换、过滤器执行等。最终，`render` 方法的返回就可以传递给 `HttpResponse` 作为视图的响应了。

`render` 方法（`django.template.backends.django.Template.render`）的实现如下所示：

```
def render(self, context=None, request=None):
    context = make_context(context, request, autoescape=self.backend.engine.autoescape)
    try:
        # 这里调用了 django.template.base.Template 的 render 方法
        return self.template.render(context)
    except TemplateDoesNotExist as exc:
        reraise(exc, self.backend)
```

这个方法的核心是调用了 `django.template.base.Template` 的 `render` 方法。在分析 `render` 方法的实现之前，先来看 `django.template.base.Template` 的定义：

```
class Template:
    def __init__(self, template_string, origin=None, name=None, engine=None):
    if engine is None:
        from .engines import Engine
        engine = Engine.get_default()
    if origin is None:
        origin = Origin(UNKNOWN_SOURCE)
    self.name = name
    self.origin = origin
    self.engine = engine
    # source 中存储的就是模板文件中的内容
    self.source = template_string
    # 将 source 的内容解析为 nodelist
    self.nodelist = self.compile_nodelist()
```

`Template` 初始化的核心是调用其 `compile_nodelist` 方法将模板内容解析为 `nodelist`。什么是 `nodelist` 呢？下面介绍 `compile_nodelist` 方法的实现：

```
def compile_nodelist(self):
    if self.engine.debug:
        lexer = DebugLexer(self.source)
    else:
        lexer = Lexer(self.source)
    # Lexer 将模板内容分割成 tokens 列表
    tokens = lexer.tokenize()
    parser = Parser(
        tokens, self.engine.template_libraries, self.engine.template_builtins, self.orig
in,
    )
```

```
    try:
        # 将 token 解析为 django.template.base.NodeList
        return parser.parse()
    except Exception as e:
        if self.engine.debug:
            e.template_debug = self.get_exception_info(e, e.token)
        raise
```

> 团子注：`lexer` 是*词法分析程序*的意思。我联想到了 `JavaScript` 的编译运行过程，词法分析、语法分析（生成抽象语法树）、代码生成。

`Lexer` 的 `tokenize` 方法将模板文件内容**按照预定义的正则表达式分割为一个列表**，其中的每一个元素都是 `django.template.base.Token` 类型的实例。

`Parser` 的 `parse` 方法将每一个 `Token` 实例解析为 `django.template.base.Node` 实例对象，最终 `compile_nodelist` 的返回就是 `NodeList`。

下面，首先来看 `Lexer` 的 `tokenize` 方法是怎样得到 `Token` 列表的。`tokenize` 方法的实现如下

```
def tokenize(self):
    in_tag = False
    lineno = 1
    result = []
    # tag_re 是正则表达式模式对象，split 方法匹配子串并分割返回列表
    for bit in tag_re.split(self.template_string):
        if bit:
            # create_token 实现词法分析，可以将子串转换为四种 Token
            result.append(self.create_token(bit, None, lineno, in_tag))
        in_tag = not in_tag
        lineno += bit.count('\n')
    return result
```

`tag_re` 将模板中可能出现的变量、标签和注释符号编译成正则表达式，之后利用 `split` 方法将模板字符串切分成多个部分。

通过正则表达式的切分，`create_token` 判断这些词的开头部分（startswith）以确定 `Token` 的类型，其核心实现如下：

```
def create_token(self, token_string, position, lineno, in_tag):
    ...
    if token_string.startwith(VARIABLE_TAG_START):
        token = Token(TOKEN_VAR, token_string[2:-2].strip(), position, lineno)
    elif token_string.startswith(BLOCK_TAG_START):
        ...
        token = Token(TOKEN_BLOCK, block_content, position, lineno)
    elif token_string.startswith(TOKEN_COMMENT, content, position, lineno)
    else:
        token = Token(TOKEN_TEXT, token_string, position, lineno)
    return token
```

可以看到，`create_token` 将 `token` 字符串解析为以下 `4` 种 `Token` 类型。

`TOKEN_VAR`：值为1，变量类型，即开头为{{的字符串。

`TOKEN_BLOCK`：值为2，块类型，即开头为{%的字符串。

`TOKEN_COMMENT`：值为3，注释类型，即开头为{#的字符串。

`TOKEN_TEXT`：值为0，文本类型，即字符串字面值。

解析模板字符串得到了 `Token` 列表之后，就将它传递给了 `Parser` 的构造函数。同时，在构造函数中将 `Django` 模板系统内置的标签和过滤器也加载进来了。`parse` 方法将给定的 `tokens` 解析为对应的 `Node`，方法实现如下：

```
def parse(self, parse_until=None):
    ...
    # NodeList 继承自 Python 列表类型
    nodelist = NodeList()
    # 迭代 Token 列表
    while self.tokens:
        token = self.next_token()
        # 文本类型简单地存储其字面值，表示为 TextNode 类型实例
        if token.token_type == 0:  # TOKEN_TEXT
            self.extend_nodelist(nodelist, TextNode(token.contents), token)
        # 变量类型
        elif token.token_type == 1:  # TOKEN_VAR
            if not token.contents:
                raise self.error(token, ...)
            try:
                # 获取 FilterExpression 类型的实例
                filter_expression = self.compile_filter(token.contents)
            except TemplateSyntaxError as e:
                raise self.error(error, e)
            # 变量类型的 Token 表示为 VariableNode 类型实例
            var_node = VariableNode(filter_expression)
            self.extend_nodelist(nodelist, var_node, token)
        # 块类型
        elif token.token_type == 2:  # TOKEN_BLOCK
            ...
    ...

    return nodelist
```

`parse` 方法中根据每一个 `Token` 的类型（由 `token_type` 属性标识）执行对应的逻辑。其中，文本类型的处理最为简单，直接将字符串字面值存储到 `TextNode` 中；变量类型比较复杂，它将 `token` 的内容存储于 `FilterExpression` 实例中，所以，`VariableNode` 中存储的实际是 `FilterExpression` 实例；块类型解析方式类似，但是它可以得到的 `Node` 类型丰富，这里不做具体分析了。同时，可以注意到，**`parse` 方法会忽略注释类型的文本。**

最终，`compile_nodelist` 方法返回了 `Node` 列表，而**实际上模板的渲染过程正是通过这些 `Node` 完成的。**

接下来，介绍模板渲染调用的方法 `render` 的实现：

```
def render(self, context):
    with context.render_context.push_state(self):
        if context.template is None:
            with context.bind_template(self):
                context.template_name = self.name
                return self._render(context)
        else:
            return self._render(context)
```

可见，`render` 方法中并没有做什么工作，只是调用了 `_render` 方法并传递了 `Context`。下面继续介绍 `_render` 的实现：

```
def _render(self, context):
    # self.nodelist 是 compile_nodelist 方法的返回值
    return self.nodelist.render(context)
```

`_render` 方法中调用了 `NodeList` 的 `render` 方法，下面是它的实现：

```
def render(self, context):
    bits = []
    for node in self:
        if isinstance(node, Node):
            bit = node.render_annotated(context)
        else:
            bit = node
        bits.append(str(bit))
    # 将 node 的返回结果拼接在一起返回，即最终渲染后的模板
    return mark_safe(''.join(bits))
```

依次对 `NodeList` 中的各个元素进行迭代，如果是 `Node` 类型（子类），则会调用 `render_annotated` 方法获取渲染结果，否则直接将元素本身作为结果。

`Node` 对象的 `render_annotated` 方法实现如下：

```
def render_annotated(self, context):
    try:
        return self.render(context)
    except Exception as e:
        ...
        raise
```

`Node` 子类各自去实现自己的 `render` 方法。下面，就来分析 `Node` 的两个子类 `TextNode` 和 `VariableNode` 中 `render` 的实现。

`TextNode` 不会用到上下文字典，只是将传递进来的字符串原样返回，其实现如下：

```
class TextNode(Node):
    def __init__(self, s):
        self.s = s

    def render(self, context):
        # 直接将字符串字面值返回
        return self.s
```

`VariableNode` 的实现如下：

```
class VariableNode(Node):
    def __init__(self, filter_expression):
        self.filter_expression = filter_expression

    def render(self, context):
        try:
```

```
            # 调用 FilterExpression 的 resolve 方法
            output = self.filter_expression.resolve(context)
        except UnicodeDecodeError:
            return ''
        return render_value_in_context(output, context)
```

`VariableNode` 在 `render` 方法中调用了 `FilterExpression` 的 `resolve` 方法，`resolve` 方法的实现依赖于两个变量：`var` 和 `filters`。这两个变量在构造函数中通过对模板内容进行解析实现初始化。

`var`：可能是 `django.template.base.Variable` 实例，如模板内容为 `{{variable}}`；也有可能是字符串，如模板内容为 `{{"Django BBS"|length}}`，此时 `var` 就是字符串 `Django BBS`。

`filters`：过滤器列表，其中的每一个元素都是一个二元组。二元组的第一个元素是过滤器函数对象，第二个元素是需要传递给过滤器的参数。列表内容可能为空，即不使用过滤器的情况。同时，**由于它是列表类型，所以，可以依次执行多个过滤器，即实现链式调用**。

最后，再看 `FilterExpression` 的 `resolve` 方法实现：

```
def resolve(self, context, ignore_failures=False):
    if isinstance(self.var, Variable):
        try:
            # 如果 var 是 Variable 实例，从上下文查找并替换变量
            obj = self.var.resolve(context)
        except VariableDoesNotExist:
            ...
            # 上下文不存在则设置为 string_if_invalid
            obj = string_if_invalid
    else:
        # var 可以是字符串字面值
        obj = self.var
    # 依次执行各个过滤器，对变量进行处理
    for func, args in self.filters:
        ...
        obj = new_obj
    return obj
```

`resolve` 方法通过两个步骤实现模板的渲染：第一，替换模板变量；第二，执行各个过滤器。这就是 `VariableNode` 渲染模板的过程。对于其他的 `Node` 类型，同样可以按照上述方法去分析它们渲染模板的过程。

`Django` 模板系统涉及的概念比较多，但是总体来看有两个核心的部分：模板语言和模板渲染。模板语言的语法非常简单，`Django` 不仅内置了许多功能强大的过滤器，而且支持自定义满足特定的场景。模板渲染即对模板语言进行解释，实现变量替换、逻辑判断和过滤器执行等功能。至此，这两个核心的部分就已经介绍完了。

---

08. Django 表单系统

在Web站点中与后端服务进行交互，通常使用表单提交的方式。**表单提交数据到达后端，首先要对数据做校验，对于不合法的数据需要拒绝并提示给前端，通过校验之后才能执行服务返回响应**。由于所有的表单创建与处理流程都是相似的，所以，`Django` 将这一过程抽象出来，形成**表单系统**。从在浏览器中显示表单到数据验证，再到对错误的处理，都可以由表单系统来完成。不仅如此，基**于数据表（Model）创建表单**也是很常见的情况，`Django` 同样考虑到了这一点，并提供了 `ModelForm` 来简化功能实现。本章将介绍 `Django` 的表单系统是怎样简化表单开发的。

08.1 认识表单

在页面中提交表单可以使用 `GET` 请求也可以使用 `POST` 请求，相应地，就可以通过 `request.GET` 或 `request.POST` 在视图中获取表单数据。`GET` 和 `POST` 这两种 `HTTP` 请求类型用于不同的目的，**对于改变系统状态的请求**，如给数据表中添加一条记录，**应该使用 `POST`**；**而不改变系统状态的请求**，如查询数据表的数据，**应该使用 `GET`**。

通过 `title` 去筛选 `Topic` 是很常见的需求，这需要用户在页面中输入想要查询的 `title` 内容，单击“搜索”按钮，返回符合要求的 `Topic` 信息。这其实就是一个简单的表单功能。下面通过视图和模板实现这个功能。

首先，在 `post` 目录下创建一个模板 `search_topic.html` 用于提交表单：
`post/templates/post/search_topic.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Topics</title>
</head>
<body>
  <form action="/post/search_topic/" method="GET">
    <input type="text" name="title">
    <input type="submit" value="search_topic">
  </form>
</body>
</html>
```

这个模板很简单，渲染到 `Web` 页面中只有一个输入框和名称为“`search_topic`”的按钮，单击此按钮会将输入框中输入的“`title`”作为参数传递到 `post/search_topic` 对应的视图中。**由于是 `GET` 请求（method指定），所以，`title` 会拼接在 `URL` 中。**

接下来，**还需要实现两个视图**：`search_topic_form` 用于显示表单模板，`search_topic` 用于接收表单查询参数，显示查询结果。首先，实现 `search_topic_form` 视图：

> 团子注：自己手写表单的话，这里已经有点麻烦了，需要两个视图

```
def search_topic_form(request):
    return render(request, 'post/search_topic.html')
```

之后，还需要在 `post/urls.py` 文件中定义 `URL` 模式：

> 团子注：我觉得，`URL` 模式还是不如**路由**这个词准确。**路由即映射，从 `URL` 模式到视图的映射。**

```
path('search_topic_form/', views.search_topic_form),
```

`search_topic` 视图的结果是 `Topic` 列表（QuerySet），这里直接使用在视图一章中实现的模板 `topic_list.html`，其中需要在上下文字典中指定 `object_list`。实现如下：

```
def search_topic(request):
    topic_qs = Topic.objects.filter(title__contains=request.GET['title'])
    return render(request, 'post/topic_list.html', context={'object_list': topic_qs})
```

> 团子注：这里应该也可以使用类视图，定义一个类，继承自 `ListView` 吧。

`search_topic` 视图的 `URL` 模式需要与 `search_topic.html` 中的 `action` 对应：

```
path('search_topic/', views.search_topic),
```

此时，可以访问http://127.0.0.1:8000/post/search_topic_form/，在文本框中输入想要查询的title，单击“search_topic”按钮就可以看到查询结果了。

使用POST方法的表单与之类似，最大的不同是数据不再附加在URL的后面了。这里不再举例说明。

完善表单处理存在的问题

对于刚刚实现的表单查询，需要假设用户熟悉这个功能，不会输入错误。但是，实际情况是**用户可能没输入查询词就单击“搜索”按钮，导致搜索结果出错**。同时，**也没有告知用户问题出在了哪里**。所以，这就暴露出当前对表单的处理存在以下一些问题。

（1）表单页面没有错误提示，如当前没有输入 `title`。

（2）`search_topic` 视图中缺少校验逻辑，即对用户的输入没有做校验，如是否为空、数据格式是否正确、类型是否满足条件等。

这些问题并不是很难解决，只需要修改模板的定义（给模板添加错误提示信息）和视图的处理逻辑（校验表单数据是否符合要求）即可。

1.修改视图处理逻辑

在 `search_topic` 中，直接使用 `request.GET['title']` 获取表单传递的 `title`，这是很危险的。因为在没有传递 `title` 的情况下，**会抛出 `MultiValueDictKeyError` 异常**。需要判断表单数据是否有效：

> 团子注：经过试验，发现不会报错，而是返回了所有的文章。

```
def search_topic(request):
    if not request.GET.get('title', ''):
        return HttpResponse('title is invalid!')
    topic_qs = Topic.objects.filter(title__contains=request.GET['title'])
    return render(request, 'post/topic_list.html', context={'object_list': topic_qs})
```

首先判断 `title` 参数是否传递且是否不为空，如果不满足条件，返回错误提示信息。否则，执行原逻辑。

这样的处理比原来的处理逻辑要好很多，兼容了可能出现的错误。但是，需要注意，当 `title` 不符合条件时，对于当前的处理方案，会在浏览器中显示“title is invalid!”如果需要重新回到表单页面，只能单击“后退”按钮或者重新输入 `url`。

> 团子注：见上面的注解，实际上这样还不如不判断。。。

这样的设计并不友好，更好的实现方式是将错误提示放在表单中，例如：

```
def search_topic(request):
    if not request.GET.get('title', ''):
        errors = ['title is invalid']
        return render(request, 'post/search_topic.html', context={'errors': errors})
    topic_qs = Topic.objects.filter(title__contains=request.GET['title'])
    return render(request, 'post/topic_list.html', context={'object_list': topic_qs})
```

对于这个修改的版本，当 `title` 不符合条件时，会停留在当前的表单模板页面。但是，页面中除了显示文本框和搜索按钮之外，还会显示出错信息。

出错信息 `errors` 之所以使用列表类型，是因为随着表单功能的修改，可能需要传递多个字段，如根据 `title` 和 `user` 去检索 `Topic` 数据，这时，可能会有多个不同的出错信息需要展示。

由于给 `search_topic.html` 模板传递了字典上下文，所以，接下来需要对表单模板进行修改。

> 团子注：`return render(...)` 和 `return redirect(...)` 有何区别？

2.修改表单模板

需要在 `search_topic.html` 模板中判断当前的上下文中是否有 `errors`，如果存在，则直接显示。`form` 标签内容并不需要修改，如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Topics</title>
</head>
<body>
  {% if errors %}
    <ul>
      {% for error in errors %}
        <li>{{ error }}</li>
      {% endfor %}
    </ul>
  {% endif %}
  <form action="/post/search_topic/" method="GET">
    <input type="text" name="title">
    <input type="submit" value="search_topic">
  </form>
</body>
</html>
```

修改完成之后，再次回到查询页面，在不输入任何内容的情况下，单击“搜索”按钮，可以看到在当前页面的上方显示了错误提示信息。

至此，才算完成了一个表单的处理过程。简单对实现表单的过程做个总结，大致包含以下几点内容。

（1）创建表单模板，模板中包含需要提交给后端处理的数据以及对错误提示信息的显示。

（2）处理提交表单的视图，视图中包含对表单数据的校验和业务处理逻辑，当表单数据不合法时，还需要给前端提示。

对于简单的表单处理，可以按照这样的思路去实现。但是，通常的业务处理中，表单会比较复杂，而且像这种只有一个字段的提交也是不太现实的。**如果仍然按照之前的方式为每一个字段编写模板代码，在业务逻辑中完成校验并给出错误提示，会非常麻烦，而且大多是样板式的重复代码。** `Django` 为此提供了**表单系统，将重复的过程抽象出来，使开发工作的重心移动到业务逻辑上去。**下面介绍Django的表单系统。

08.2 使用表单系统实现表单

表单系统的核心是 **`Form` 对象**，它将表单中的字段封装成一系列 `Field` 和验证规则，以此来自动地生成 `HTML` 表单标签。本节首先使用 `Form` 对象重新实现上一节中定义的表单，再详细讲解 `Form` 对象的构成，最后介绍 `Django` 为 `Model` 驱动的表单而设计的 `ModelForm` 对象。

使用Form对象定义表单
与其他对象的定义规则类似，可以将 `Form` 对象定义在任何位置。但是， `Django` 的建议是将它们定义在应用下的 `forms.py` 文件中。**这样做的原因是归类存储**，如视图函数需要放置在 `views.py` 文件中，当看到文件名就能够知道文件中存储的是什么。

在 `post` 应用下新建 `forms.py` 文件，并定义如下内容：

```
from django import forms

class TopicSearchForm(forms.Form):
    title = forms.CharField(label='Topic title')
```

以发现，Form的定义与Model的定义非常相似，比较容易理解。

（1） `Django` 规定，所有的 `Form` 对象都必须继承自 `django.forms.Form`，所以这里的 `TopicSearchForm` 符合条件。

（2）定义了一个 `title` 属性，它是 `forms.CharField` 类型的 `Field`，根据名称可以猜测此处将 `title` 指定为字符类型。`label` 标签显式地指定了这个字段的名称，且 `Field` 有一个默认属性 `required` 为 `True`，代表这个字段是必填的。

> 团子注：
>
> - 问题是 `label` 是干啥的？
> `label` 就是显示在页面中的文字，如果不写的话，会自动使用首字母大写的字段名。可以这样记忆， `HTML` 语言中的 `label` 标签的作用和表单字段的 `label` 属性其实效果是一样的。

接下来，先来看 `Form` 对象都有哪些特性。
1.实现对所有字段的验证

每一个 `Form` 对象实例都会有一个 `is_valid` 方法，这个方法**根据字段的定义验证实例的各个字段是否合法。**如果所有的字段都是合法的，它会返回 `True`，并且将数据存储到字典类型的 `cleaned_data` 属性中。

可以在 `Shell` 中定义并实例化 `Form` 对象，看它验证数据的过程。为了更好地说明 `Form` 的验证规则，定义一个具有3个 `Field` 的 `ExampleForm`：

```
>>> from django import forms
>>> class ExampleForm(forms.Form):
...     a = forms.CharField(max_length=10)
...     b = forms.CharField(required=False)
```

```
...     c = forms.IntegerField(min_value=0, max_value=10)
```

其中 `a` 和 `b` 是字符类型的 `Field`，且 `a` 规定了最大长度为 `10`，`b` 设置了可以不提供。`c` 是整数类型的 `Field`，规定了取值范围。

创建 `Form` 实例可以传递字典对象到构造函数中，例如：

```
>>> ef = ExampleForm({'a': 'Django', 'b': 'BBS', 'c': 6})
```

如之前所说，`ef` 有一个 `is_valid` 方法，可以验证 `Form` 实例是否有效。如下所示：

```
>>> ef.is_valid()
True
```

由于**这里传递的字典可以完成对 `ExampleForm` 中定义的 `Field` 实现填充，所以，数据验证结果为真，即这个表单可以使用。**

验证有效的表单会有一个字典类型的 `cleaned_data` 属性，它之所以叫作“cleaned”，是因为**表单对象会对传递进来的数据做清理，把值转换成合适的 `Python` 类型。**这将在视图中应用 `Form` 时看到它的用途。

如果传递的字典不能与 `Field` 对应，那么表单对象就是无效的。例如：

由于字典中没有 `key` 为 `a` 的字段，且 `forms.CharField` 默认的 `required` 属性为 `True`，所以，`is_valid` 方法返回了 `False`。此时，可以通过表单实例的 `errors` 属性查看错误信息：

`errors` 是 `django.forms.utils.ErrorDict` 类型的实例，它是 `Python` 字典类型的子类，可以将错误信息包装成 `HTML` 模板。所以，使用表单对象只需要对字段的取值做限定，**可以不需要考虑错误信息的展示。**

表单对象也提供了查看各个字段错误信息的方法：

> 访问错误信息字典的某个键，得到的是一个列表。

以上这些就是表单对象实现的对字段验证的功能。通常，在使用表单对象时，会传递参数初始化表单实例，调用其 `is_valid` 方法，如果为 `True`，则从 `cleaned_data` 属性中获取清理之后的字段值执行业务逻辑。否则，返回错误提示信息。另外，`cleaned_data` 属性在 `is_valid` 执行之前并不存在，所以，在实际使用的时候需要注意调用它们的顺序。

2.根据字段定义生成HTML

> 团子注：`Django` 表单系统的两个功能，验证表单，渲染表单。

表单对象另一个强大的功能是可以根据定义的字段自动生成 `HTML`。例如，可以打印之前看到的 `ExampleForm` 实例 `ef`：

```
>>> print(ef)
```

```
    <tr>
        <th><label for="id_a">A:</label></th>
        <td><input type="text" name="a" value="Django" maxlength="10" required
        id="id_ a"/></td>
    </tr>
    <tr>
        <th><label for="id_b">B:</label></th>
        <td><input type="text" name="b" value="BBS" id="id_b" /></td>
    </tr>
    <tr>
        <th><label for="id_c">C:</label></th>
        <td><input type="number" name="c" value="6" min="0" max="10" required
        id="id_c" /></td>
    </tr>
```

可以看到，表单实例可以自动生成 `HTML` 表单元素，且默认输出使用 `HTML` 表格，**但是并不提供 `<table>` 起始和结束标签**。同时，也可以使用实例的 `as_ul` 方法**获取列表形式的表单**或使用 `as_p` 方法**获取段落形式的表单**。

除了可以显示整个表单元素之外，也可以指定显示某个字段的 `HTML` 元素：

```
>>> print(ef['a'])
<input type="text" name="a" value="Django" maxlength="10" required id="id_a" />
>>> print(ef['b'])
<input type="text" name="b" value="BBS" id="id_b" />
>>> print(ef['c'])
<input type="number" name="c" value="6" min="0" max="10" required id="id_c" />
```

由于表单实例可以直接返回 `HTML` 表单元素，所以，可以**用它来替换模板文件中的字段定义**。更方便的是，在没有正确填充表单时，它还可以返回错误信息的提示。

接下来，使用表单系统（`TopicSearchForm`）重新实现筛选 `Topic` 的模板。如下所示，将 `search_topic.html` 的 `body` 标签部分修改为：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Topics</title>
</head>
<body>
  <form action="/post/search_topic/" method="GET">
    {{ form }}
    <input type="submit" value="search_topic">
  </form>
</body>
</html>
```

模板中使用 `form` 变量替换了 `title` 输入框，根据之前的说明可以知道，`form` 变量需要是 `Form` 对象实例。同时，模板中的 `errors` 也被删除了，由 `form` 来给出提示。

由于这里只有一个 `title` 输入框，所以，`form` 变量带来的便捷性体现得不是很明显。但是可以想象，当页面中的表单字段非常多的时候，也只需要使用一个 `form` 变量来替换，这种简单易用的特性就很容易表现出来了。

将 `TopicSearchForm` 对象应用到视图中去，修改 `search_topic`：

```
def search_topic(request):
    """接收 form 的 action 的视图函数"""
    # 团子注：注意这里，request.GET 是一个 QueryDict 对象，之前在 Shell 中使用的是字典对象
    form = TopicSearchForm(request.GET)
    if form.is_valid():
        topic_qs = Topic.objects.filter(title__contains=form.cleaned_data['title'])
        return render(request, 'post/topic_list.html', context={'object_list': topic_q
s})
    else:
        return render(request, 'post/search_topic.html', context={'form': form})
```

在 `search_topic` 中，首先将 `request.GET` 传递给 `TopicSearchForm` 对象的构造函数用来初始化表单实例，之后调用实例的 `is_valid` 方法判断当前的实例是否有效。

（1）返回 `True`，则从表单实例的 `cleaned_data` 属性中获取清理之后的表单字段，完成业务逻辑，返回响应。

（2）返回 `False`，**将带有错误信息的表单实例**作为上下文传递到需要渲染的模板中。

同时，修改 `search_topic_form`，只需要创建一个 `TopicSearchForm` 实例传递到上下文中就可以了：

```
def search_topic_form(request):
    """渲染表单"""
    return render(request, 'post/search_topic.html', context={'form': TopicSearchForm
()})
```

> 团子注：注意这里是新建的一个表单实例，是没有 `errors` 的。

重新操作按照 `title` 去检索 `Topic` 的流程，可以发现，当 `title` 输入框中没有输入任何内容时，单击“搜索”按钮，表单对象会给出默认的错误提示。

常用的表单字段类型

1.表单字段的基类Field

基类 `Field` 定义于 `django/forms/fields.py` 文件中，这里重点关注它的构造函数中定义的属性以及校验给定字段值是否有效的 `clean` 方法。

`Field` 的构造函数中定义了很多属性，子类中也可以根据需要设定这些属性值。下面介绍一些最常用的属性。

（1）required

设定当前的 `Field` 是否是必须提供的，默认值是 `True`，即必须提供。`Field` 中定义了一个类属性 `empty_values`，`clean` 方法会判断当前 `Field` 的值（value）是否是空值：

```
if value in self.empty_values and self.required:
    raise ValidationError(self.error_messages['required'], code='required')
```

`empty_values` 定义为 `list(validators.EMPTY_VALUES)`，`EMPTY_VALUES` 定义为：

```
EMPTY_VALUES = (None, '', [], (), {})
```

当 `value` 是 `None`、空字符串等空值时，会抛出 `ValidationError` 异常。如果将 `required` 指定为 `False`，那么 `Field` 将会变成可选的，即使不提供也不会抛出异常。

（2）widget

指定在页面中显示字段的控件，可以是 `Widget` 类或者 `Widget` 类实例。对于大多数情况，默认的控件使用 `TextInput`。

例如，可以将 `title` 字段的 `widget` 属性设定为 `forms.Textarea`：

```
title = forms.CharField(label='Topic title', widget=forms.Textarea())
```

（3）label

指定在页面中显示的字段的名称（标签）提示。这在定义 `TopicSearchForm` 时已经见到过了，将 `title` 的 `label` 属性设定为 `Topic title`。如果不显式地指定，页面中将**直接显示字段定义的变量名（首字母大写）**。

（4）initial

指定字段的初始值，默认为 `None`。当给字段的 `initial` 属性设定一个非空值时，**页面中的对应表单将使用这个值填充**。

例如，可以给 `title` 字段设置初始值：

```
title = forms.CharField(label='Topic title', initial='Django BBS')
```

此时，再次打开表单页面，可以看到 `title` 的输入框中有了默认值 `Django BBS`。

话题标题: Django BBS search_topic

（5）help_text

用于给当前的字段添加描述性信息，提示当前字段的作用或需要输入的内容解释。如果设定了该属性，则其在页面中将会显示在 `Field` 的旁边。

（6）error_messages

这个属性用于覆盖 `Field` 默认的错误消息。为了更好地说明它的作用，下面用 `title` 字段举例说明。

首先，在不设置这个属性的情况下：

```
>>> from django import forms
>>> title = forms.CharField(label='Topic title')
>>> title.clean('')
...
django.core.exceptions.ValidationError: ['This field is required.']
```

默认的错误信息显示：“This field is required.”。下面，指定 `error_messages` 属性：

```
    >>> title = forms.CharField(label='Topic title', error_messages={'required': '请填写话题标题!'})
    >>> title.clean('')
    ...
    django.core.exceptions.ValidationError: ['请填写话题标题!']
```

可以看到，此时抛出的异常中携带的错误信息就是设定的 `error_messages` 属性。

> 团子注：给表单字段指定 `error_messages`，这里是一个字典。

（7）disabled

其默认值是 `False`，如果修改为 `True`，则当前的表单字段将不可编辑。**当设置字段为不可编辑时，需要提供初始值（initial），否则，这个字段也就没有意义了。**

每一个 `Field` 实例都有 `clean` 方法，它接受一个参数，即字段值。 `Field` 实例调用 `clean` 方法用来对传递的数据做“清理”和校验：对数据做清理可以将数据转换成对应的 `Python` 对象（如字段定义为 `forms.IntegerField`，`clean` 方法可以接受能够强转为整数的字符串，并返回整数数值）；校验检验当前给定的数据是否满足 `Field` 属性的约束，如果不满足，则会抛出 `ValidationError` 异常。

2.常用的表单字段类型

目前，已经看到过 `CharField` 和 `IntegerField` 的基本使用方法，它们都是 `Field` 的子类。除了可以设定 `Field` 中定义的属性之外，它们还定义了一些额外的属性用来限制对字段的赋值。下面介绍表单系统中常用的内置字段类型。

（1）CharField

其为字符串类型的表单字段，是最常见的表单字段类型，`widget` 默认使用 `TextInput`。它的构造函数如下：

```
def __init__(self, *, max_length=None, min_length=None, strip=True, empty_value='', **kw
args)
```

`max_length` 和 `min_length` 限定了字段值的最大、最小长度。对于不满足限定条件的字段值，将会引起 `ValidationError` 异常。例如：

```
    >>> x = forms.CharField(min_length=5, max_length=10)
    >>> x.clean('Django')
    'Django'
    >>> x.clean('Hello Django BBS')
    ...
```

```
    django.core.exceptions.ValidationError: ['Ensure this value has at most 10 characters
(it has 16).']
```

`strip` 属性默认会执行 `Python` 字符串的 `strip` 方法，用于去除字符串开头和尾部的空格。如果不需要这样做，可以将 `strip` 属性设置为 `False`：

```
>>> x = forms.CharField()
>>> x.clean('  Django  ')
'Django'
>>> x = forms.CharField(strip=False)
>>> x.clean('  Django  ')
'  Django  '
```

当传递的字段值是空值（`empty_values` 属性）时，将会使用 `empty_value` 属性设定的值。例如：

> `empty_values` 是一个类属性。

```
>>> x = forms.CharField(empty_value='Django')
>>> x.clean('')
'Django'
```

可以看到，`x` 调用 `clean` 方法传递了空字符串并没有抛出异常，这是因为设定了非空值的 `empty_value`。

> 团子注：但是默认情况下，`empty_value` 为空字符串 `''`，这样的话，如果传递了空字符串，得到的值还是一个空字符串，这样就会报错了。

（2）IntegerField

其为整数类型的表单字段，`widget` 默认使用 `NumberInput`。它的构造函数如下：

```
def __init__(self, *, max_value=None, min_value=None, **kwargs)
```

可选的 `max_value` 与 `min_value` 参数用于限定字段值的取值范围，且它们都是闭区间。如果提供了这两个参数（或其中之一），且给定的字段值不在取值范围内，将会抛出异常，并带有错误提示信息。例如：

```
    >>> x = forms.IntegerField(max_value=10, min_value=5)
    >>> x.clean('10')
    10
    >>> x.clean(10)
    10
    >>> x.clean(12)
    ...
    django.core.exceptions.ValidationError: ['Ensure this value is less than or equal to
10.']
```

从示例中可以看到，**字段值可以是通过强转得到整数的字符串**。

> 团子注：就是因为字段实例的 `clean` 方法，完成了清洗（类型转换，确切的说应该是到 `Python` 对象的转换）和校验的工作。

（3）BooleanField

其为布尔类型的表单字段，`widget` 默认使用 `CheckboxInput`。由于基类 `Field` 的 `required` 属性默认是 `True`，所以，在不做设置的情况下，`BooleanField` 实例的 `required` 属性也是 `True`。

由于 `required` 为 `True` 要求这个字段值必须提供，所以，这种情况下，`BooleanField` 类型的实例必须是 `True`，否则将抛出异常，并提示：“This field is required.”。

如果要在表单中使用 `BooleanField` 字段，则需要指定 `required` 为 `False`：

```
>>> x = forms.BooleanField(required=False)
>>> x.clean(False)
False
```

> 团子注：亲测是正确的。。。

（4）ChoiceField

其为选择类型的表单字段，`widget` 默认使用 `Select`。它的构造函数如下：

```
def __init__(self, *, choices=(), **kwargs)
```

`choices` 参数需要一个可迭代的二元组或能够返回可迭代二元组的函数对象。使用方法如下：

```
>>> Gender = (
...         ('M', 'male'),
...         ('F', 'female')
...    )
>>> x = forms.ChoiceField(choices=Gender)
```

此时，页面中 `x` 字段对应的选择框中有两个可选值：`male` 和 `female`。但是，表单提交的字段值是二元组中的第一个元素，即 `M` 或 `F`。

（5）EmailField

它继承自 `CharField`，但是提供了 `Email` 验证器，用于校验传递的字段值是否是合法的电子邮件地址。`widget` 默认使用 `EmailInput`。由于需要输入电子邮件的表单比较常见，如用户注册表单、身份验证表单等，所以，这个字段类型用在这些场景中会非常方便。

它的使用方法与 `CharField` 是类似的，例如：

```
>>> x = forms.EmailField()
>>> x.clean('admin@email.com')
'admin@email.com'
>>> x.clean('admin')
...
django.core.exceptions.ValidationError: ['Enter a valid email address.']
```

（6）DateTimeField

它是用于表示时间的表单字段，`widget` 默认使用 `DateTimeInput`。它接受一个可选的参数 `input_formats`，这个参数是一个列表，列表元素规定了可以转换为 `datetime.datetime` 的时间格式。默认接受的时间格式列表如下所示：

```
['%Y-%m-%d %H:%M:%S',
 '%Y-%m-%d %H:%M:%S.%f',
 '%Y-%m-%d %H:%M',
 '%Y-%m-%d',
 '%m/%d/%Y %H:%M:%S',
 '%m/%d/%Y %H:%M:%S.%f',
 '%m/%d/%Y %H:%M',
 '%m/%d/%Y',
 '%m/%d/%y %H:%M:%S',
 '%m/%d/%y %H:%M:%S.%f',
 '%m/%d/%y %H:%M',
 '%m/%d/%y']
```

`DateTimeField` 的 `clean` 方法接受的值类型可以是 `datetime.datetime`、`datetime.date` 或符合特定格式的字符串，最终会返回 `datetime.datetime` 对象或抛出异常。

自定义表单字段类型

这里由于要自定义 `Field` 类，因此需要理解 `Field` 的工作原理。`Field` 通过 `clean` 方法校验并获取字段值，所以，自定义 `Field` 通常就是自己去实现 `clean` 方法。

首先，看一看 `Field` 类的 `clean` 方法都做了些什么：

```
def clean(self, value):
    value = self.to_python(value)
    self.validate(value)
    self.run_validators(value)
    return value
```

`clean` 接受传递的 `value`，经过 `3` 个方法的处理，返回或抛出异常，这 `3` 个方法的作用分别如下。

（1）`to_python`：实现数据转换，将传递进来的 `value` 转换成需要的 `Python` 对象。例如，对于 `DateTimeField`，它的 `to_python` 方法会将 `value` 转换成 `datetime.datetime` 对象。

（2）`validate`：验证经过转换的 `value` 是否合法，如果不合法，需要抛出 `ValidationError` 异常。`Field` 中实现的 `validate` 方法只是**简单地对 `required` 属性限制的条件进行校验**，即如果 `required` 为 `True`，且 `value` 为空值，则会抛出异常。

（3）`run_validators`：这个方法会执行当前实例中包含的验证器（由 `default_validators` 属性和 `validators` 属性指定），如果出现错误，则会抛出 `ValidationError` 异常。

最后，经过转换和校验的 `value` 返回，代表的是已经经过清理的有效数据。因此，对于自定义的 `Field` 类，可以只去实现 `clean` 中调用的方法，就能够实现自定义数据的转换和校验规则。

下面，实现一个 `TopicField`，它继承自 `Field`，可以实现输入 `Topic` 的 `id` 获取 `Topic` 实例对象的功能：

```
from django import forms
from post.models import Topic
from django.core.exceptions import ValidationError


class TopicField(forms.Field):
    default_error_messages = {
        'invalid': '输入一个整数',
        'not_exist': '模型不存在',
    }

    def to_python(self, value):
        try:
            value = int(str(value).strip())
            return Topic.objects.get(pk=value)
        except (ValueError, TypeError):
            # 团子注：ValidationError 的第一个参数是错误信息，第二个参数是通过关键字指定的错误代码
            raise ValidationError(self.error_messages['invalid'], code='invalid')
        except Topic.DoesNotExist:
            raise ValidationError(self.error_messages['not_exist'], code='not_exist')
```

`TopicField` 只是重新实现了 `to_python` 方法，把 `value` 当作主键去查询 `Topic` 实例，返回实例对象或抛出 `ValidationError` 异常。

使用 `TopicField` 的方法与内置的 `Field` 相同，例如：

```
>>> x = TopicField()
>>> x.clean(1)
<Topic: 1: first topic>
>>> x.clean(6)
...
django.core.exceptions.ValidationError: ['Model Not Exist']
>>> x.clean('x')
...
django.core.exceptions.ValidationError: ['Enter a whole number.']
```

继承基类 `Field` 去自定义表单字段可能需要考虑比较多的问题，所以，通常自定义的 `Field` 都会继承自 `CharField`、`IntegerField` 等内置的 `Field` 子类。

例如，实现一个简单的 `SignField`，对输入的字符串添加 `django` 前缀：

```python
class SignField(forms.CharField):
    def clean(self, value):
        return 'django %s' % super().clean(value)
```

`SignField` 在 `clean` 中调用了父类的 `clean` 方法，也就使用了 `CharField` 的数据校验规则。使用方法如下所示：

```
>>> x = SignField()
>>> x.clean('bbs')
'django bbs'
```

到目前为止，还没有使用过验证器去校验数据的合法性。下面，实现一个 `EvenField` 用于表示偶数，**利用验证器去校验数据**。

实现偶数验证器：

```python
def even_validator(value):
    if value % 2 != 0:
        raise ValidationError('%d is not a even number' % value)
```

> 团子注：这里为什么不给 `ValidationError` 指定 `code` 参数了呢？

验证器就是一个可调用对象，接受一个参数，并验证参数是否符合预期，如果不符合预期则抛出 `ValidationError` 异常。

`EvenField` 继承自 `IntegerField`，并在初始化函数中设置验证器：

```python
class EvenField(forms.IntegerField):
    def __init__(self, **kwargs):
        super().__init__(validators=[even_validator], **kwargs)
```

`EvenField` 只可以接受偶数，否则，将会抛出异常：

```
>>> x = EvenField()
>>> x.clean(1)
...
django.core.exceptions.ValidationError: ['1 is not a even number']
>>> x.clean(2)
2
```

自定义表单的验证规则

大多数场景下使用自定义的表单字段类型是为了添加额外的数据校验逻辑，但是，这种为满足特定的业务场景去实现的 `Field` 类使用频率往往很低。所以，**如果只需要对一些表单字段做额外的校验，可以将校验逻辑写在 `Form` 中。**

表单系统会自动查找以 `clean_` 开头，以字段名结尾的方法，它会在验证字段合法性的过程中被调用。因此，如果想要自定义 `TopicSearchForm` 中 `title` 的验证逻辑，可以在表单对象中实现 `clean_title` 方法。例如：

```
class TopicSearchForm(forms.Form):
    title = forms.CharField(label='Topic title')

    def clean_title(self):
        title = self.cleaned_data['title']
        if len(title) < 5:
            raise forms.ValidationError('字符串长度太短')
        return title
```

`clean_title` 方法会在 `title` 字段的默认验证逻辑执行完成之后执行，所以，可以直接通过 `cleaned_data` 属性获取到符合 `CharField` 约束的数据值。

在 `clean_title` 中只是简单校验当前的数据值长度不能小于 `5`，若不符合要求则会抛出异常，并给出错误提示信息。

需要注意，**在自定义验证方法结束时，需要将字段值返回**，否则，这个字段的值就会变成 `None`，这也是常见的错误。

最后，验证当前自定义的验证规则是否生效：

```
>>> form = TopicSearchForm({'title': 'BBS'})
>>> form.is_valid()
False
>>> form['title'].errors
['字符串长度太短!']
>>> form = TopicSearchForm({'title': 'Django BBS'})
>>> form.is_valid()
True
```

基于Model定制的表单

将 `Model` 翻译成表单是很常见的业务场景，如需要提供一个表单页面给用户创建 `Topic`，这个页面中的表单字段就需要与 `Topic` 的 `Model` 定义相对应。利用 `Form` 对象并不难实现，只需要将 `Model` 中定义的字段翻译成 `Form` 对象中的表单字段即可。但是，**如果这种需求很多，且 `Model` 中定义的字段也较多，那么重复实现这种表单的过程会很烦琐的。**

`Django` 表单系统考虑到了这个问题，提供了 `ModelForm`，可以基于 `Model` 的定义自动生成表单，很大程度上简化了 `Model` 翻译成表单的过程。

`ModelForm` 虽然可以自动地把 `Model` 中的字段翻译成表单字段类型，但是，**并不会翻译所有的字段**，`editable=False` 的模型字段都不会出现在 `ModelForm` 中，如自增主键、自动添加的时间字段等。

1.一个简单的ModelForm

首先，实现一个简单的 `ModelForm`，主要看它的使用方法与需要注意的地方：

```
class TopicModelForm(forms.ModelForm):
    class Meta:
        model = Topic
        exclude = ('is_online', 'user')
```

`ModelForm` 需要使用 `Meta` 来设置必要的元信息，这与 `Model` 的定义非常相似。在 `TopicModelForm` 中 `Meta` 设置了两个选项：`model` 指定需要生成表单的模型对象，`exclude` 标识不需要在表单中展示的字段。

这几乎也是定义一个 `ModelForm` 的最小化配置了。

对于当前定义的 `TopicModelForm`，由于在 `exclude` 中指定了不需要的 `Model` 字段，所以，它只有 `title` 和 `content` 两个表单字段。

`ModelForm` 的使用方法与 `Form` 类似，同样可以使用 `is_valid` 方法校验字段值的合法性和通过 `cleaned_data` 属性获取清理后的字段值。例如：

```
>>> form = TopicModelForm({'title': 'bbs', 'content': 'django'})
>>> form.is_valid()
True
>>> form.cleaned_data
```

```
{'title': 'bbs', 'content': 'django'}
```

`ModelForm` 也会**校验模型字段中设置的限制条件**。如 `Topic` 中的 `title` 字段存在唯一性限制，那么，当表单对象执行 `is_valid` 方法时，不仅会校验 `title` 的字面值，同时还会查询数据库确认不存在重复的记录。

由于当前 `Topic` 表中存在 `title` 为 `first topic` 的记录，所以，传递同样的 `title` 后，`is_valid` 方法会返回 `False`，并携带错误信息：

```
>>> form = TopicModelForm({'title': 'first topic', 'content': 'django'})
>>> form.is_valid()
False
>>> form.errors
{'title': ['话题 with this Title already exists.']}
```

将表单应用到视图中才是有意义的，下面，利用 `TopicModelForm` 去实现创建 `Topic` 的表单。首先，在 `post` 目录下创建模板文件 `topic_model_form.html`：
`post/templates/post/topic_model_form.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Topic</title>
</head>
<body>
  <form action="" method="POST">
    <table>
```

```
        {{ form }}
      </table>
      {% csrf_token %}
      <input type="submit" value="提交">
    </form>
  </body>
  </html>
```

模板的内容与 `search_topic.html` 几乎相同，但需要注意两个地方：由于指定了 `POST` 请求类型，需要 `CSRF` 保护机制，所以添加了 `{%csrf_token%}` 模板标签； `action` 没有设定 `URL` 的意思是将表单提交到与当前页面相同的 `URL`。
接下来，在视图中应用 `TopicModelForm`：

```
def topic_model_form(request):
    if request.method == 'POST':
        topic = TopicModelForm(request.POST)
        if topic.is_valid():
            topic = Topic.objects.create(title=topic.cleaned_data['title'], content=topi
c.cleaned_data['content'], user=request.user)
            # 团子注：第一次看到这种用法，还可以不经过路由直接访问视图函数的吗?
            return topic_detail_view(request, topic.id)
        else:
            return render(request, 'post/topic_model_form.html', context={'form': topi
c})
    else:
        return render(request, 'post/topic_model_form.html', context={'form': TopicModel
Form()})
```

视图 `topic_model_form` 的实现也非常简单，包含了3个条件判断。

（1）如果是 `POST` 请求，且表单字段值合法，则创建 `Topic` 对象实例，并**返回当前创建的实例详细信息页面**。

（2）如果是 `POST` 请求，但表单字段值不合法，则返回表单页面同时显示错误提示信息。

（3）不是 `POST` 请求，显示表单页面。

最后，给视图定义 `URL` 模式，在 `post/urls.py` 文件中添加：

```
path('topic_model_form/', views.topic_model_form),
```

在浏览器中访问http://127.0.0.1:8000/post/topic_model_form/

填充表单字段值，单击“提交”按钮即可保存新的 `Topic` 对象或给出错误提示信息（如输入的 `title` 与当前数据记录重复等）。

> 团子注：“话题标题”和“话题内容”都是在 `Model` 字段中的 `help_text` 中指定的。

2.常用的Meta选项

目前已经见到 `ModelForm` 中的 `Meta` 设定了 `model` 和 `exclude` 两个选项，下面介绍 `Meta` 的其他常用选项。

（1）fields

其为列表或元组类型，与 `exclude` 相反，它指定当前的表单应该包含哪些字段，如果要所有的 `Model` 字段都包含在表单中，可以设定 `fields='__all__'`。

`ModelForm` 的定义中**必须要包含 `fields` 或 `exclude` 选项**，否则将会抛出异常，同时给出错误提示：Creating a ModelForm without either the'fields'attribute or the'exclude'attribute is prohibited。

（2）labels

其为字典类型，用于定义表单字段的名称（输入框左边显示的名称）。**表单字段的名称首先会使用 `Model` 字段定义设置的 `verbose_name`，如果没有设置，则直接使用字段名。**因此当没有定义 `verbose_name` 时，就可以使用 `labels` 选项来指定字段名。例如：

```
labels = {
    'title': 标题,
    'content': 内容,
}
```

> 团子注：`ModelForm` 的 `Meta` 里面设置的 `labels` 字典**会覆盖掉**模型字段的 `verbose_name`。

（3）help_texts

其为字典类型，用于给表单字段添加帮助信息。目前页面中表单字段的帮助信息（输入框下方显示的内容）来自 `Model` 字段的 `help_text` 定义，如果没有定义则什么都不显示。`help_texts` 的定义方式与 `labels` 选项类似，例如：

```
help_texts = {
    'title': '简短的话题标题',
    'content': '话题的详细内容',
}
```

（4）widgets

其为字典类型，用于定义表单字段选用的控件。默认情况下，**`ModelForm` 会根据 `Model` 字段的类型映射表单 `Field` 类**，因此会应用 `Field` 类中默认定义的 `widget`。这个选项用于自定义控件类型，例如：

```
from django.forms import widgets
widgets = {
    'content': widgets.Textarea(attrs={'cols': '60', 'rows': '5'})
}
```

> 团子注：这样子难道不会把 `widgets` 覆盖掉吗？等到你定义第二个 `ModelForm` 的时候，就会出错了！

此时，就指定了 `content` 表单字段使用长 `60` 列、宽 `5` 行的 `Textarea`。

（5）field_classes

其为字典类型，**用于指定表单字段使用的 `Field` 类型。**默认情况下，对于 `title` 字段，`ModelForm` 会将它映射为 `fields.CharField` 类型。可以根据需要改变这种默认行为，例如，将 `title` 设置为 `forms.EmailField` 类型：

> 团子注：默认情况，`Django` 把模型字段映射为表单字段，每个表单字段对应了控件类型。如果要修改控件类型，需要指定 `widgets` 字典，如果要修改表单字段类型，需要指定 `field_classes` 字典。

```
field_classes = {
    'title': forms.EmailField
}
```

3.ModelForm的save方法

在之前的示例 `topic_model_form` 视图中，对于表单提交的数据，最后使用了 `Model` 查询管理器的 `save` 方法保存 `Topic` 对象。这是因为除了表单中定义的 `title` 和 `content` 字段之外，还需要给定 `user` 这个必填字段。但是，对于有些 `Model` 定义，需要将所有的 `Model` 字段都定义在 `ModelForm` 中，此时，字段值通过校验（is_valid）之后，可以使用 `ModelForm` 提供的 `save` 方法实现 `Model` 对象的保存。

`ModelForm` 的 `save` 方法定义于它的基类 `BaseModelForm` 中，它的实现如下：

```
def save(self, commit=True):
    ...
    if commit:
        # 除了保存当前的 Model 实例，还会保存多对多关系
        self.instance.save()
        self._save_m2m()
    else:
        # 将保存多对多数据的方法赋值给 save_m2m.save 返回后可以手动提交
        self.save_m2m = self._save_m2m
    # 返回 Model 实例
    return self.instance
```

`save` 方法接受一个 `commit` 参数，默认为 `True`，可以实现 `Model` 实例的保存以及多对多关系数据的保存。如果在使用 `save` 方法时设置了 `commit` 为 `False`，则不会执行保存动作。此时，可以对返回的实例对象做一些加工，再执行保存操作。

例如，对于 `topic_model_form` 视图中保存 `Topic` 的实现可以修改为：

```
topic = TopicModelForm(request.POST)
if topic.is_valid():
    topic = topic.save(commit=False)
    topic.user = request.user
    topic.save()
    return topic_detial_view(request, topic.id)
```

> 团子注：当表单的定义就是为了修改模型的时候，可以使用 `ModelForm`。

08.3 表单系统的工作原理

通过继承 `Form` 对象，定义所需要的表单字段，基本上完成了表单的定义。它可以自动生成 `HTML`，完成字段值的校验，并给出相应错误的提示信息。

> 团子注：自动生成 `HTML` 其实就是所谓的“渲染表单”。

表单对象的创建过程

所有的表单对象都继承自 `Form`，首先来看 `Form` 的定义：

```
class Form(BaseForm, metaclass=DeclarativeFieldsMetaclass)
```

`Form` 中指定了基类 `BaseForm` 和元类 `DeclarativeFieldsMetaclass`。`BaseForm` 中定义了生成 `HTML` 与字段值校验的方法，而 `DeclarativeFieldsMetaclass` 则定义了创建 `Form` 对象的过程。

`DeclarativeFieldsMetaclass` 的实现如下：

```
class DelarativeFieldsMetaclass(MediaDefiningClass):
    def __new__(mcs, name, bases, attrs):
        current_fields = []
        # 遍历当前类中定义的属性
        for key, value in list(attrs.items()):
            # 只添加 Field 类型的实例
            if isinstance(value, Field):
                current_fields.append((key, value))
        attrs['declared_fields'] = OrderedDict(current_fields)
        # 调用父类的 new 方法创建类对象
        new_class = super(DeclarativeFieldsMetaclass, mcs).__new__(mcs, name, bases, att
rs)

        declared_fields = OrderedDict()
        for base in reversed(new_class.__mro__):
            # 继承父类的字段定义
            if hasattr(base, 'declared_fields'):
                declared_fields.update(base.declared_fields)
            ...
        # 这里给创建的类对象添加了 base_fields 和 declared_fields 两个属性
        new_class.base_fields = declared_fields
        new_class.declared_fields = declared_fields
        return new_class
```

`DeclarativeFieldsMetaclass` 继承自 `MediaDefiningClass`，并调用了它的 `new` 方法创建了类对象。`MediaDefiningClass` 的实现如下：

```
class MediaDefiningClass(type):
    def __new__(mcs, name, base, attrs):
        new_class = super(MediaDefiningClass, mcs).__new__(mcs, name, bases, attrs)
        # 如果属性中没有 media，则通过 media_property 添加
        if 'media' not in attrs:
            new_class.media = media_property(new_class)
        return new_class
```

`MediaDefiningClass` 中并没有做太多工作，只是给类对象添加了 `media` 属性，用于实现对 `JavaScript` 和 `CSS` 的引用。

最后，在类对象返回之前，给它附加了两个属性，且都指向了 `declared_fields`。这是一个 `OrderedDict` 类型的实例，存储了表单中定义的 `Field`。例如，可以查看 `TopicSearchForm` 的 `base_fields` 属性：

```
>>> TopicSearchForm.base_fields
OrderedDict([('title', <django.forms.fields.CharField object at 0x10ae0be80>)])
```

表单对象校验的实现过程

表单对象创建之后，就可以传递字段值实现实例化，再之后就可以调用 `is_valid` 方法校验字段值是否合法。所以，**校验过程的实现也就是 `is_valid` 方法的实现。**

`is_valid` 方法定义于 `BaseForm` 中，实现如下：

```
def is_valid(self):
    return self.is_bound and not self.errors
```

`is_bound` 是一个布尔值，在 `BaseForm` 的构造函数中定义：

```
def __init__(self, data=None, files=None, ...):
    self.is_bound = data is not None or files is not None
```

因此，在传递了 `data` 或 `files` 属性且不为 `None` 的情况下，`is_bound` 就是 `True`。`errors` 是 `@property` 修饰的方法，返回 `ErrorDict` 实例。实现如下：

```
def errors(self):
    if self._errors is None:
        self.full_clean()
    return self._errors
```

`_errors` 属性在构造函数中被初始化为 `None`，所以，当第一次调用 `errors` 时，一定会执行 `full_clean` 方法：

```
def full_clean(self):
    self._errors = ErrorDict()
    if not self.is_bound:
        return
    # 定义 cleaned_data 属性
    self.cleaned_data = {}
    # 如果表单允许为空且初始化之后的字段值没有过更改，直接返回
    if self.empty_permitted and not self.has_changed():
        return

    self._clean_fields()
    self._clean_form()
    self._post_clean()
```

`full_clean` 方法首先对 `_errors` 重新赋值，之后定义了 `cleaned_data` 属性。故在表单对象执行 `full_clean` 方法之前，`cleaned_data` 是不存在的。最后，执行了三个名称中带有 `clean` 的方法。下面看 `_clean_fields`：

```
def _clean_fields(self):
    # 依次迭代各个表单字段
    for name, field in self.fields.items():
        # 当前表单字段不可编辑
        if field.disabled:
            # 获取字段初始值
            value = self.get_initial_for_field(field, name)
        else:
            # 使用 Field 的 widget 属性获取字段值
            value = field.widget.value_from_datadict(self.data, self.files, self.add_pre
fix(name))
        try:
            ...
            # 调用 Field 的 clean 方法完成字段值到 Python 对象的转换和校验
            value = field.clean(value)
            # 给 cleaned_data 赋值
            self.cleaned_data[name] = value
            # 表单自定义的验证规则，如果存在，最后执行
            if hasattr(self, 'clean_%s' % name):
                value = getattr(self, 'clean_%s' %name)()
                self.cleaned_data[name] = value
        except ValidationError as e:
            # 字段校验抛出异常，将异常加入 _errors 属性中
            self.add_error(name, e)
```

`_clean_fields` 方法主要完成了 `cleaned_data` 和 `_errors` 这两个属性的填充。同时，我们也详细介绍了为什么在表单对象中定义以 `clean_` 开头、字段名结尾的方法会被用来校验字段的合法性。

填充 `cleaned_data` 属性的过程比较简单，利用 `Field` 类的 `clean` 方法和自定义规则完成两次校验。下面介绍填充 `_errors` 属性的过程，`add_error` 方法实现如下：

```
def add_error(self, field, error):
    # 如果 error 不是 ValidationError 的实例，构造 ValidationError 实例
    if not isinstance(error, ValidationError):
        error = ValidationError(error)

    # 解析 ValidationError 获取 field 与错误提示的字典
    if hasattr(error, 'error_dict'):
        if field is not None:
            raise TypeError(...)
        else:
            error = error.error_dict
    else:
        error = {field or NONE_FIELD_ERRORS: error.error_list}

    for field, error_list in error.items():
        # 如果当前 field 没有出现在 errors 中
        if field not in self.errors:
            ...
            # 实例化一个 ErrorList 赋值给 _errors[field]
            self._errors[field] = self.error_class()
        # 向 _errors[field] 中添加错误提示信息
        self._errors[field].extend(error_list)
        # 从 cleaned_data 属性中删除 field
        if field in self.cleaned_data:
            del self.cleaned_data[field]
```