Request 对象和 Response 对象 JsonResponse 对象 和 Django shortcut functions 和 QueryDict 对象

阅读目录

request

request 属性

```
属性：
    django 将请求报文中的请求行、头部信息、内容主体封装成 HttpRequest 类中的属性。
   除了特殊说明的之外，其他均为只读的。
'''

0.HttpRequest.scheme
   表示请求方案的字符串（通常为 http 或 https）
1.HttpRequest.body
    返回一个字符串，代表请求报文的主体。在处理非 HTTP 形式的报文时非常有用，例如：二进制图片、XML,Json 等。
    例如:b'username=alex&password=123456'
    但是，如果要处理表单数据的时候，推荐还是使用 HttpRequest.POST 。
    另外，我们还可以用 python 的类文件方法去操作它，详情参考 HttpRequest.read() 。
2.HttpRequest.path
    一个字符串，表示请求的路径组件（不含域名）。
    例如："/music/bands/the_beatles/"
3.HttpRequest.method
    一个字符串，表示请求使用的 HTTP 方法。必须使用大写。
    例如："GET"、"POST"
4.HttpRequest.encoding
    一个字符串，表示提交的数据的编码方式（如果为 None 则表示使用 DEFAULT_CHARSET 的设置，默认为 'utf-8'）。
   这个属性是可写的，你可以修改它来修改访问表单数据使用的编码。
   接下来对属性的任何访问（例如从 GET 或 POST 中读取数据）将使用新的 encoding 值。
```

如果你知道表单数据的编码不是 DEFAULT_CHARSET ，则使用它。

5.HttpRequest.GET 得到一个类 QueryDict 的对象，包含 HTTP GET 的所有参数也就是?后边的。<QueryDict: {'a': ['1', '2'], 'b': ['1']

6.HttpRequest.POST 一个类似于字典的对象，如果请求中包含表单数据，则将这些数据封装成 QueryDict 对象。 POST 请求可以带有空的 POST 字典 —— 如果通过 HTTP POST 方法发送一个表单，

但是表单中没有任何的数据，QueryDict 对象依然会被创建。 因此，不应该使用 if request.POST 来检查使用的是否是 POST 方法；应该使用 if request.method == "POST"

另外：如果使用 POST 上传文件的话，文件信息将包含在 FILES 属性中。

7.HttpRequest.COOKIES

一个标准的 Python 字典，包含所有的 cookie。键和值都为字符串。

8.HttpRequest.FILES

一个类似于字典的对象，包含所有的上传文件信息。 FILES 中的每个键为<input type="file" name="" /> 中的 name，值则为对应的数据。

注意，FILES 只有在请求的方法为 POST 且提交的<form> 带有 enctype="multipart/form-data" 的情况下才会 包含数据。否则，FILES 将为一个空的类似于字典的对象。

9.HttpRequest.META

一个标准的 Python 字典，包含所有的 HTTP 首部。具体的头部信息取决于客户端和服务器，

下面是一些示例：

CONTENT_LENGTH —— 请求的正文的长度（是一个字符串）。

CONTENT_TYPE —— 请求的正文的 MIME 类型。

HTTP_ACCEPT —— 响应可接收的 Content-Type。

HTTP_ACCEPT_ENCODING —— 响应可接收的编码。

HTTP_ACCEPT_LANGUAGE —— 响应可接收的语言。

HTTP_HOST —— 客服端发送的 HTTP Host 头部。

HTTP_REFERER —— Referring 页面。

HTTP_USER_AGENT —— 客户端的 user-agent 字符串。

QUERY_STRING —— 单个字符串形式的查询字符（未解析的形式）

REMOTE_ADDR —— 客户端的 IP 地址。

REMOTE_HOST —— 客户端的主机名。

REMOTE_USER —— 服务器认证后的用户。

REQUEST_METHOD —— 一个字符串，例如"GET" 或"POST"。

SERVER_NAME —— 服务器的主机名。

SERVER_PORT —— 服务器的端口（是一个字符串）。

从上面可以看到，除 CONTENT_LENGTH 和 CONTENT_TYPE 之外，请求中的任何 HTTP 首部转换为 META 的键时，

都会将所有字母大写并将连接符替换为下划线最后加上 HTTP_ 前缀。 所以，一个叫做 X-Bender 的头部将转换成 META 中的 HTTP_X_BENDER 键。

10.HttpRequest.user

一个 AUTH_USER_MODEL 类型的对象，表示当前登录的用户。 如果用户当前没有登录，user 将设置为 django.contrib.auth.models.AnonymousUser 的一个实例。
你可以通过 is_authenticated() 区分它们。 例如： if request.user.is_authenticated(): # Do something for logged-in users. else: # Do something for anonymous users.
user 只有当 Django 启用 AuthenticationMiddleware 中间件时才可用。 ---------- 匿名用户
class models.AnonymousUser django.contrib.auth.models.AnonymousUser 类实现了 django.contrib.auth.models.User 接口，但具有下面几个不同点： id 永远为 None。
username 永远为空字符串。 get_username() 永远返回空字符串。 is_staff 和 is_superuser 永远为 False。 is_active 永远为 False。 groups 和 user_permissions 永远为空。
is_anonymous() 返回 True 而不是 False。 is_authenticated() 返回 False 而不是 True。 set_password()、check_password()、save() 和 delete() 引发 NotImplementedError。
New in Django 1.8: 新增 AnonymousUser.get_username() 以更好地模拟 django.contrib.auth.models.User。
11.HttpRequest.session 一个既可读又可写的类似于字典的对象，表示当前的会话。只有当 Django 启用会话的支持时才可用。 完整的细节参见会话的文档。 '''

上传文件实例

上传文件实例

```
def upload(request):
    """
    保存上传文件前，数据需要存放在某个位置。默认当上传文件小于 2.5M 时，django 会将上传文件的全部内容读进内存。从内存读取一次，写磁盘一次。
    但当上传文件很大时，django 会把上传文件写到临时文件中，然后存放到系统临时文件夹中。
    :param request:
    :return:
    """
    if request.method == "POST":
        # 从请求的 FILES 中获取上传文件的文件名，file 为页面上 type=files 类型 input 的 name 属性值
        filename = request.FILES["file"].name
```

```
            # 在项目目录下新建一个文件
            with open(filename, "wb") as f:
                # 从上传的文件对象中一点一点读
                for chunk in request.FILES["file"].chunks():
                    # 写入本地文件
                    f.write(chunk)
            return HttpResponse("上传 OK")
```

request 的方法()

'''

1.HttpRequest.get_host()

根据从 HTTP_X_FORWARDED_HOST（如果打开 USE_X_FORWARDED_HOST，默认为 False）和 HTTP_HOST 头部信息返回请求的原始主机。

如果这两个头部没有提供相应的值，则使用 SERVER_NAME 和 SERVER_PORT，在 PEP 3333 中有详细描述。

USE_X_FORWARDED_HOST：一个布尔值，用于指定是否优先使用 X-Forwarded-Host 首部，仅在代理设置了该首部的情况下，才可以被使用。

例如："127.0.0.1:8000"

注意：当主机位于多个代理后面时，get_host() 方法将会失败。除非使用中间件重写代理的首部。

注意和 request.path 的区别

2.HttpRequest.get_full_path()

返回 path，如果后边有参数的话将加上查询参数。

例如："/music/bands/the_beatles/?print=true"

3.HttpRequest.get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)

返回签名过的 Cookie 对应的值，如果签名不再合法则返回 django.core.signing.BadSignature。

如果提供 default 参数，将不会引发异常并返回 default 的值。

可选参数 salt 可以用来对安全密钥强力攻击提供额外的保护。max_age 参数用于检查 Cookie 对应的时间戳以确保 Cookie 的时间不会超过 max_age 秒。

```
复制代码
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
```

```
        'Tony' # 假设在设置 cookie 的时候使用的是相同的 salt
        >>> request.get_signed_cookie('non-existing-cookie')
        ...
        KeyError: 'non-existing-cookie'    # 没有相应的键时触发异常
        >>> request.get_signed_cookie('non-existing-cookie', False)
        False
        >>> request.get_signed_cookie('cookie-that-was-tampered-with')
        ...
        BadSignature: ...
        >>> request.get_signed_cookie('name', max_age=60)
        ...
        SignatureExpired: Signature age 1677.3839159 > 60 seconds
        >>> request.get_signed_cookie('name', False, max_age=60)
        False
        复制代码

4.HttpRequest.is_secure()
    如果请求时是安全的，则返回 True；即请求通是过 HTTPS 发起的。
5.HttpRequest.is_ajax() 判断一个请求是否为 ajax 请求
    如果请求是通过 XMLHttpRequest 发起的，则返回 True，方法是检查 HTTP_X_REQUESTED_WITH 相应的首部是否是字符串'XMLHttpRequest'。
    大部分现代的 JavaScript 库都会发送这个头部。如果你编写自己的 XMLHttpRequest 调用（在浏览器端），你必须手工设置这个值来让 is_ajax() 可以工作。
    如果一个响应需要根据请求是否是通过 AJAX 发起的，并且你正在使用某种形式的缓存例如 Django 的 cache middleware，
   你应该使用 vary_on_headers('HTTP_X_REQUESTED_WITH') 装饰你的视图以让响应能够正确地缓存。
'''
```

注意：前端 POST 提交的数据有多个键值的时候,比如 checkbox 类型的 input 标签，select 标签，需要用：

```
request.POST.getlist("hobby")
```

面试题:

后端的 request.POST.get 中取不到数据是哪里的问题?

检查前端发过来的请求 header 中的'Content-Type'：是不是'application/x-www-form-urlencoded?看源码从 class HttpRequest(object)中获取 POST QueryDict 的函数中可以看出

从 elif self.content_type == 'application/x-www-form-urlencoded':这个分支能看到只有请求 header 中的'Content-Type': 'application/x-www-form-urlencoded'才会填充 request.POST, 其它情况下只有一个空的<QueryDict: {}>。

源码: 从别的博客中找到的,但是没有验证,

Response 对象

与由 Django 自动创建的 HttpRequest 对象相比，HttpResponse 对象是我们的职责范围了。我们写的每个视图都需要实例化，填充和返回一个 HttpResponse。

HttpResponse 类位于 django.http 模块中。

使用

传递字符串

```
from django.http import HttpResponse
response = HttpResponse("Here's the text of the Web page.")
response = HttpResponse("Text only, please.", content_type="text/plain")
```

设置或删除响应头信息

```
response = HttpResponse()
response['Content-Type'] = 'text/html; charset=UTF-8'
del response['Content-Type']
```

属性

HttpResponse.content：响应内容

HttpResponse.charset：响应内容的编码

HttpResponse.status_code：响应的状态码

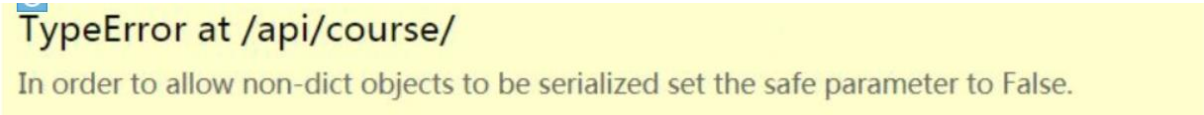
JsonResponse 对象

class JsonResponse(data, encoder=DjangoJSONEncoder, safe=True, json_dumps_params=None,**kwargs)

JsonResponse 是 HttpResponse 的子类，专门用来生成 JSON 编码的字符串。它和父类的区别主要有

1.它的默认 content_type 为: application/json

2.第一个参数 data 传入的应该是**一个字典类型**,如果想要传入其他的数据类型,应该设为 safe=False, 如果你不设置这个参数,传入非字典类型的话,会发生以下错误



3.序列化中文时候会出现编码错误解决方法.

```
return JsonResponse(course_list, safe=False, json_dumps_params={'ensure_ascii':False})
```

```
from django.http import JsonResponse
response = JsonResponse({'foo': 'bar'})
print(response.content)
b'{"foo": "bar"}'
```

默认只能传递字典类型，如果要传递非字典类型需要设置一下 safe 关键字参数。

```
response = JsonResponse([1, 2, 3], safe=False)
```

Django shortcut functions

官方文档

render()

render(request, template_name[, context])

结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的 **HttpResponse 对象**。

其默认的 Content-Type 标头设置为 **application/json**。

参数：

```
request：  用于生成响应的请求对象。
     template_name：要使用的模板的完整名称，可选的参数
     context：添加到模板上下文的一个字典。默认是一个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。
     content_type：生成的文档要使用的 MIME 类型。默认为 DEFAULT_CONTENT_TYPE 设置的值。
     status：响应的状态码。默认为 200。
```

一个简单的例子：

```
from django.shortcuts import render
def my_view(request):
    # 视图的代码写在这里
    return render(request, 'myapp/index.html', {'foo': 'bar'})
```

上面的代码等于：

```
from django.http import HttpResponse
from django.template import loader
```

```
def my_view(request):
    # 视图代码写在这里
    t = loader.get_template('myapp/index.html')
    c = {'foo': 'bar'}
    return HttpResponse(t.render(c, request))
```

redirect()

参数可以是:

- 一个模型: 将调用模型的 get_absolute_url() 函数
- 一个视图, 可以带有参数: 将使用 urlresolvers.reverse 来反向解析名称
- 一个绝对的或相对的 URL, 将原封不动的作为重定向的位置。

默认返回一个临时的重定向; 传递 permanent=True 可以返回一个永久的重定向。

示例:

你可以用多种方式使用 redirect() 函数。

传递一个对象（ORM 相关）

将调用 get_absolute_url() 方法来获取重定向的 URL:

```
from django.shortcuts import redirect
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)
```

传递一个视图的名称

```
def my_view(request):
    ...
    return redirect('some-view-name', foo='bar')
```

传递要重定向到的一个具体的网址

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

当然也可以是一个完整的网址

```
def my_view(request):
```

```
    ...
    return redirect('http://example.com/')
```

默认情况下，redirect() 返回一个临时重定向。以上所有的形式都接收一个 permanent 参数；如果设置为 True，将返回一个永久的重定向：

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

扩展阅读：

永久重定向（响应状态码：302）和临时重定向（响应状态码：301）对普通用户来说是没什么区别的，它主要面向的是搜索引擎的机器人。

A 页面临时重定向到 B 页面，那搜索引擎收录的就是 A 页面。

A 页面永久重定向到 B 页面，那搜索引擎收录的就是 B 页面。

QueryDict 对象

定义在 from django.http import QueryDict 中，

在 HttpRequest 对象中，GET 和 POST 是 django.http.QueryDict 的实例的对象，像类定制来处理同一个键的多个值。 这个类的需求来自某些 HTML 表单元素传递多个值给同一个键，<select multiple> 是一个显著的例子。

request.POST 和 request.GET 的 QueryDict 在一个正常的请求/响应过程中是不可变的。 若要获得可变的版本，需要使用 copy() 方法。

方法

QueryDict 是字典的子类,它拥有字典的所有的标准方法.

QueryDict.__init__(*query_string=None*, *mutable=False*, *encoding=None*)

```
from django.http import QueryDict
import os,django
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "text1.settings")# project_name 项目名称
django.setup()
#以上是 python 配置
ret=QueryDict("a=1&a=2&b=1")#实例化一个对象
print(type(ret))
print(ret)
```

结果:

```
<class 'django.http.request.QueryDict'>
<QueryDict: {'a': ['1', '2'], 'b': ['1']}>
```

如果没有传值(*query_string=None*)将会得到 QueryDict 类型的空字典

注意: request.POST 和 request.GET 得到的 QueryDict,是不能改变的,只能查询,不能修改和赋值,但是你自己构造的 QueryDict 对象需要修改时,可以通过把 mutable=True,

```
from django.http import QueryDict
ret=QueryDict("a=1&a=2&b=1",mutable=True)#把这里改为 True
ret["key"]="111"
print(ret)
```

结果:

```
<QueryDict: {'a': ['1', '2'], 'b': ['1'], 'key': ['111']}>
```

设置键和值的字符串都将从 encoding 转换为 unicode。 如果 encoding 未设置，则默认为 DEFAULT_CHARSET。

QueryDict.copy()

使用 copy.deepcopy()返回对象的副本。 此副本是**可变**的即使原始对象是不可变的。

```
import copy
from django.http import QueryDict
ret=QueryDict("a=1&a=2&b=1")#这里不用 mutable=True
new_ret=copy.deepcopy(ret)
new_ret["key"]="111"
print(new_ret)
```

结果:

```
<QueryDict: {'a': ['1', '2'], 'b': ['1'], 'key': ['111']}>
```

QueryDict.dict()

把 QueryDict 转化为标准的字典来表示。 对于 QueryDict 中的每个（键，列表）对，dict 将具有（key，item），其中 item 是列表的一个元素，使用与 QueryDict.__getitem__()

```
#别忘记导入
ret=QueryDict("a=1&a=2&b=1")
new_ret=ret.dict()
print(ret)
print(new_ret)
```

结果:

```
<QueryDict: {'a': ['1', '2'], 'b': ['1']}>
{'a': '2', 'b': '1'}
```

QueryDict.urlencode(*safe=None*)

以查询字符串格式返回数据的字符串。也就是把 QueryDict 字典格式转变为 url 地址中?后边参数的形式.和 QueryDict()方法相反

```
ret=QueryDict("a=1&a=2&b=1")
print(ret)

print(ret.urlencode())
```

结果:

```
<QueryDict: {'a': ['1', '2'], 'b': ['1']}>
a=1&a=2&b=1
```

使用 safe 参数传递不需要编码的字符。 像这样:

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```