

## 请求传参

利用HTTP请求报文传递参数有3种方式

1. 查询字符串，GET方法，参数在URL中， `http://127.0.0.1:8000/emp/?x=123&x=abc&y=789`
2. 表单，前端网页中填写表单，一般使用POST方法，参数在body中

```
1 POST /xxx/yyy?id=5&name=magedu HTTP/1.1
2 HOST: 127.0.0.1:9999
3 content-length: 26
4 content-type: application/x-www-form-urlencoded
5
6 age=5&weight=80&height=170
```

也可以POST、PUT提交Json格式数据

3. URL本身就是数据的表达， `http://127.0.0.1:8080/python/2010/u101`

## APIView

在Django中，View是视图类基类，路由配置中需要把类通过`as_view()`伪装成视图函数，请求通过路由进入到这个视图函数中，内部为每一个请求实例化一个视图类实例，并根据`request.method`找到对应的handler。而这个基本流程已经被View类固定在其内部，我们只需要定义`get`、`post`等方法即可，简化了编程。

使用DRF，基于Django，也要使用View类。请求参数是Json格式，先要序列化它，然后验证，验证合格可以入库。响应的数据应该序列化成Json格式。你会发现这也是固定的套路，是否也能够简化呢？

参考 <https://www.django-rest-framework.org/api-guide/views/>

APIView

- `as_view()`调用基类View的，但是使用了`csrf_exempt(view)`来排除CSRF保护
- 重新定义了Request类来替代Django的，虽是重写，但是依然有联系
- 重新定义了Response类来增强替代Django的
- 异常类都是基于APIException类的
- 对请求进行认证和授权

APIView没有提供增删改查的handler方法，也就是说和View一样，需要自己定义`get`、`post`、`put`、`delete`方法。

## GET请求测试

GET请求: <http://127.0.0.1:8000/emp/?x=123&x=abc&y=789>

```

1 from rest_framework.views import APIView, Request, Response
2
3 class TestIndex(APIView):
4     def get(self, request:Request):
5         print('~' * 30)
6         print(request.method)          # HttpRequest的属性
7         print(request.GET)              # HttpRequest的属性
8         print(request.query_params)     # Request的属性，使用小写的
9         print(request.content_type)
10        print('~' * 30)
11        return Response({})

```

```

1 GET
2 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
3 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
4 text/plain

```

QueryDict本质就是字典，对于同一个参数有多值情况使用了列表。

## POST请求测试

POST请求: <http://127.0.0.1:8000/emp/?x=123&x=abc&y=789>

### 一、采用表单提交方式

```

1 from rest_framework.views import APIView, Request, Response
2
3 class TestIndex(APIView):
4     def post(self, request:Request):
5         print('~' * 30)
6         print(request.method)          # HttpRequest的属性
7         print(request.GET)              # HttpRequest的属性
8         print(request.query_params)     # Request的属性，使用小写的
9         print(request.content_type)
10        print(request.POST)              # HttpRequest的属性
11        print(request.data)              # Request的属性，使用小写的
12        print('~' * 30)
13        return Response({})

```

POST http://127.0.0.1:8000/emp/?x=123&x=abc&y=789 Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	m	111			
<input checked="" type="checkbox"/>	n	222			
<input checked="" type="checkbox"/>	t	333			
	Key	Value	Description		

Body Cookies Headers (9) Test Results Status: 200 OK Time: 20 ms Size: 288 B Save Response

```
1 b'm=111&n=222&t=333' 这是request报文的body的原始数据
2
3 POST
4 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
5 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
6 application/x-www-form-urlencoded 表单提交的方法
7 <QueryDict: {'m': ['111'], 'n': ['222'], 't': ['333']}>
8 <QueryDict: {'m': ['111'], 'n': ['222'], 't': ['333']}>
```

## 二、Json数据

The screenshot shows a REST client interface with a POST request to `http://127.0.0.1:8000/emp/?x=123&x=abc&y=789`. The body is set to 'raw' and the media type is 'Text'. The response status is 415 Unsupported Media Type. The error message is: `"detail": "Unsupported media type 'text/plain' in request."`

```
1 {
2   ... "a":123,
3   ... "b":"abc",
4   ... "c":true,
5   ... "d":null
6 }
```

Body Cookies Headers (9) Test Results Status: 415 Unsupported Media Type Time: 35 ms Size: 369 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "detail": "Unsupported media type 'text/plain' in request."
3 }
```

```
1 b'{\r\n  "a":123,\r\n  "b":"abc",\r\n  "c":true,\r\n  "d":null\r\n}'
2 text/plain
3
4 Unsupported Media Type: /emp/
```

说明上例解析失败，原因就是类型选择了Text，改为Json

The screenshot shows the same REST client interface, but the media type is now 'JSON'. The response status is 200 OK. The body is still the same JSON object.

```
1 {
2   ... "a":123,
3   ... "b":"abc",
4   ... "c":true,
5   ... "d":null
6 }
```

Body Cookies Headers (9) Test Results Status: 200 OK Time: 9 ms Size: 288 B Save Response

```

1 b'{\r\n    "a": 123,\r\n    "b": "abc",\r\n    "c": true,\r\n    "d":
  null\r\n}'
2
3 POST
4 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
5 <QueryDict: {'x': ['123', 'abc'], 'y': ['789']}>
6 application/json
7 <QueryDict: {}>
8 {'a': 123, 'b': 'abc', 'c': True, 'd': None}

```

## 请求总结

- GET请求，查询字符串使用`query_params`提取
- POST请求，访问`data`属性
  - 支持了POST、PUT、PATCH方法（增、改）
  - 统一将请求处理放到`data`属性中
  - 如果是Json数据，帮我们序列化

## 响应

DRF对Django的响应类也做了增强，使用更加简单方便。

```
Response(data=None, status=None, template_name=None, headers=None,
content_type=None)
```

- 兼具了模板渲染的能力
- `data`将要序列化的数据，例如字典
- `status`状态码，默认200
- `headers`响应报文头，字典
- `content_type`响应内容类型，有时候需要手动设置

```

1 from rest_framework.views import APIView, Request, Response
2
3 class TestIndex(APIView):
4     def get(self, request:Request):
5         print(request.query_params) # Request的属性，使用小写的
6         return Response({})
7
8     def post(self, request:Request):
9         print(request.query_params) # Request的属性，使用小写的
10        print(request.data)         # Request的属性，使用小写的
11        return Response({
12            'host':'python', 'domain':'magedu.com'
13        }, status=201, headers={'X-Server':'Magedu'})

```

## 应用

需要构建2个类

- 列表页，返回列表和新增功能
- 详情页，基于**主键**的查看详情、修改、删除

## 路由

```
1 from django.urls import path
2 from .views import EmpsView, EmpView
3
4 urlpatterns = [
5     path('', EmpsView.as_view()), # /emp/
6     path('<int:pk>/', EmpView.as_view()), # /emp/10021/
7 ]
```

## 列表页

```
1 from rest_framework.views import APIView, Request, Response
2
3 class EmpsView(APIView):
4     """
5     实现列表页get、新增post
6     http://127.0.0.1:8000/emp/
7     """
8     def get(self, request):
9         pass
10
11     def post(self, request):
12         pass
```

## 详情页

```
1 class EmpView(APIView):
2     """
3     实现详情页get、修改put、删除delete
4     http://127.0.0.1:8000/emp/10021
5     """
6     def get(self, request, pk:int):
7         pass
8
9     def put(self, request, pk:int):
10         pass
11
12     def delete(self, request, pk:int):
13         pass
14
```

## 异常处理

参考 <https://www.django-rest-framework.org/api-guide/exceptions/>

测试<http://127.0.0.1:8000/emp/100>，返回500服务器内部错误。

按平常做法，就应该出什么问题，返回什么出错信息。但这样做会有安全风险，且在浏览器端的普通用户根本不管什么错误，就认为网站出问题了。所以，返回更加友好的出错页面是有必要的，例如更换404页面。

我们的项目采用前后端分离，返回出错页面不合适，需要返回json格式的出错信息，有必要拦截所有的异常做处理。

## 异常全局配置

自定义全局异常处理器，参考 `rest_framework.views.exception_handler` 实现

```
1 REST_FRAMEWORK = {
2     'EXCEPTION_HANDLER': 'utils.exceptions.global_exception_handler'
3 }
```

## 全局异常处理

`exception_handler`会转化Django的`Http404`、`PermissionDenied`为DRF的基于`APIException`的类。

项目根目录下新建包`utils`，其中新建模块`exceptions.py`，内容如下

```
1 from django.http import Http404
2 from django.core.exceptions import PermissionDenied
3 from rest_framework.views import set_rollback
4 from rest_framework import exceptions
5 from rest_framework.views import Response, exception_handler
6
7 class MagBaseException(exceptions.APIException):
8     """基类定义基本的异常"""
9     code = 10000 # code为0表示正常，非0都是错误
10    message = '非法请求' # 错误描述
11
12    @classmethod
13    def get_message(cls):
14        return {'code': cls.code, 'message': cls.message}
15
16    # 内部异常暴露细节
17    exc_map = {
18
19    }
20
21
22 def global_exception_handler(exc, context):
23     """
24     全局异常处理
25
26     照抄rest_framework.views.exception_handler，略作修改
27     不管什么异常这里统一处理。根据不同类型显示不同的
28     为了前端解析方便，这里响应的状态码采用默认的200
29     异常对应处理后返回对应的错误码和错误描述
30     异常找不到对应就返回缺省
31     """
32     if isinstance(exc, Http404):
33         exc = exceptions.NotFound()
34     elif isinstance(exc, PermissionDenied):
35         exc = exceptions.PermissionDenied()
36
37     print('异常', '=' * 30)
38     print(type(exc), exc.__dict__)
39     print('=' * 30)
40
41     if isinstance(exc, exceptions.APIException):
42         headers = {}
```

```

43         if getattr(exc, 'auth_header', None):
44             headers['WWW-Authenticate'] = exc.auth_header
45         if getattr(exc, 'wait', None):
46             headers['Retry-After'] = '%d' % exc.wait
47
48         if isinstance(exc.detail, (list, dict)):
49             data = exc.detail
50         else:
51             data = {'detail': exc.detail}
52
53         set_rollback()
54
55         errmsg = exc_map.get(exc.__class__.__name__,
MagBaseException).get_message()
56         return Response(errmsg, status=200) # 状态恒为200
57         #return Response(data, status=exc.status_code, headers=headers)
58
59     return None

```

抛出异常后，拦截它们，这里做统一的处理

## 列表页和新增实现

```

1  from rest_framework.views import APIView, Request, Response
2  from .models import Employee
3  from .serializers import EmpSerializer
4
5  class EmpsView(APIView):
6      """
7      实现列表页get、新增post
8      http://127.0.0.1:8000/emp/
9      """
10     def get(self, request):
11         emps = Employee.objects.all()
12         return Response(EmpSerializer(emps, many=True).data)
13
14     def post(self, request):
15         serializer = EmpSerializer(data=request.data)
16         serializer.is_valid(True)
17         serializer.save()
18         return Response(serializer.data)

```

post测试数据

```

1  {
2      "emp_no": 10023,
3      "birth_date": "2000-06-01",
4      "first_name": "sam",
5      "last_name": "lee",
6      "gender": 1,
7      "hire_date": "2020-08-24"
8  }

```

## 详情页、修改和删除实现

```
1 from rest_framework.views import APIView, Request, Response
2 from .models import Employee
3 from .serializers import EmpSerializer
4
5 class Empview(APIView):
6     """
7     实现详情页get、修改put、删除delete
8     http://127.0.0.1:8000/emp/10021
9     """
10    def get(self, request, pk:int):
11        obj = Employee.objects.get(pk=pk)
12        return Response(EmpSerializer(obj).data)
13
14
15    def put(self, request, pk:int):
16        # 必须先查后改
17        obj = Employee.objects.get(pk=pk)
18        serializer = EmpSerializer(obj, data=request.data)
19        serializer.is_valid(True)
20        serializer.save()
21        return Response(serializer.data, 201)
22
23
24    def delete(self, request, pk:int):
25        Employee.objects.get(pk=pk).delete()
26        return Response(status=204)
27
```

PUT测试用json

```
1 {
2     "emp_no": 10023,
3     "birth_date": "2000-06-01",
4     "first_name": "sam",
5     "last_name": "lee",
6     "gender": 2,
7     "hire_date": "2019-08-24"
8 }
```

完整参考代码

employee/views.py

```
1 from rest_framework.views import APIView, Request, Response
2 from .models import Employee
3 from .serializers import EmpSerializer
4
5
6 class Empsview(APIView):
7     """
8     实现列表页get、新增post
9     http://127.0.0.1:8000/emp/
```



```

10 """
11 def get(self, request):
12     emps = Employee.objects.all()
13     return Response(EmpSerializer(emps, many=True).data)
14
15 def post(self, request):
16     serializer = EmpSerializer(data=request.data)
17     serializer.is_valid(True)
18     serializer.save()
19     return Response(serializer.data)
20
21 class EmpView(APIView):
22     """
23     实现详情页get、修改put、删除delete
24     http://127.0.0.1:8000/emp/10021
25     """
26     def get(self, request, pk:int):
27         obj = Employee.objects.get(pk=pk)
28         return Response(EmpSerializer(obj).data)
29
30     def put(self, request, pk:int):
31         # 必须先查后改
32         obj = Employee.objects.get(pk=pk)
33         serializer = EmpSerializer(obj, data=request.data)
34         serializer.is_valid(True)
35         serializer.save()
36         return Response(serializer.data, 201)
37
38     def delete(self, request, pk:int):
39         Employee.objects.get(pk=pk).delete()
40         return Response(status=204)

```

employee/models.py

```

1 from django.db import models
2
3 class Employee(models.Model):
4     class Gender(models.IntegerChoices): # 枚举类型, 限定取值范围
5         MAN = 1, '男'
6         FEMALE = 2, '女'
7     class Meta:
8         db_table = 'employees'
9         verbose_name = '员工'
10     emp_no = models.IntegerField(primary_key=True, verbose_name='工号')
11     birth_date = models.DateField(verbose_name='生日')
12     first_name = models.CharField(max_length=14, verbose_name='名')
13     last_name = models.CharField(max_length=16, verbose_name='姓')
14     gender = models.SmallIntegerField(verbose_name='性别',
15     choices=Gender.choices)
16     hire_date = models.DateField()
17
18 class Salary(models.Model):
19     class Meta:
20         db_table = "salaries"
21     #id = models.AutoField(primary_key=True) # 额外增加的主键, Django不支持联合主
    键
22     emp_no = models.ForeignKey(Employee, on_delete=models.CASCADE,

```

```
22         db_column='emp_no', related_name='salaries')
23     from_date = models.DateField()
24     salary = models.IntegerField(verbose_name='工资')
25     to_date = models.DateField()
```

employee/serializers.py

```
1  from rest_framework import serializers
2  from .models import Employee, Salary
3
4  class SalarySerializer(serializers.ModelSerializer):
5      class Meta:
6          model = Salary
7          fields = '__all__'
8
9  class EmpSerializer(serializers.ModelSerializer):
10     class Meta:
11         model = Employee
12         fields = '__all__'
```

