数据库查询强化训练

字段条件查询

字段查询是指如何指定SQL WHERE子句的内容。它们用作QuerySet的filter(), exclude()和get()方法的关键字参数。

其基本格式是：field__lookuptype=value，注意其中是双下划线。

默认查找类型为exact(精确匹配)。

lookuptype的类型有：

Django的数据库API支持20多种查询类型，下表列出了所有的字段查询参数：

字段名	说明
exact	精确匹配
iexact	不区分大小写的精确匹配
contains	包含匹配
icontains	不区分大小写的包含匹配
in	在..之内的匹配
gt	大于
gte	大于等于
lt	小于
lte	小于等于
startswith	从开头匹配
istartswith	不区分大小写从开头匹配
endswith	从结尾处匹配
iendswith	不区分大小写从结尾处匹配
range	范围匹配
date	日期匹配
year	年份
iso_year	以ISO 8601标准确定的年份
month	月份
day	日期
week	第几周
week_day	周几
iso_week_day	以ISO 8601标准确定的星期几
quarter	季度
time	时间
hour	小时
minute	分钟
second	秒
regex	区分大小写的正则匹配
iregex	不区分大小写的正则匹配

案例：

```
# 字段查询
class TestFieldQuery(TestCase):
    def setUp(self) -> None:
        Request.objects.create(url='/api/request/demo1',data={'name':'小
明','age':18,'addr':'nanjing','father':{'name':'xiaogang','age':40}})

    def test_contains_query(self):
        req = Request.objects.filter(url__contains='demo')
        print(req)

    def test_in_query(self):
        req = Request.objects.filter(url__in=
['/api/request/demo1','/api/request/demo2'])
        print(req)
```

执行测试

```
python manage.py test sqtp.tests.TestFieldQuery
```

跨关系查询练习

Django提供了强大并且直观的方式解决跨越关联的查询，它在后台自动执行包含JOIN的SQL语句。要跨越某个关联，只需使用关联的模型字段名称，并使用双下划线分隔，直至你想要的字段（可以链式跨越，无限跨度）.

例如

#查找标签是xxx的用例

res1=Case.objects.filter(tags__name='smoketest')

案例:

```
#跨关系查询
class TestOverRelations(TestCase):
    def setUp(self) -> None:
        # 创建套件和用例

        self.suite_conf = Config.objects.create(name='套件1')
        self.suite = Suite.objects.create(config=self.suite_conf)
        self.config = Config.objects.create(name='用例1')
        self.case = Case.objects.create(config=self.config,suite=self.suite)

    # 查找某套件下的用例步骤
    def test_case_step(self):
        Step.objects.create(belong_case=self.case,name='步骤1')
        Step.objects.create(belong_case=self.case,name='步骤2')
        Step.objects.create(belong_case=self.case,name='步骤3')
        steps=Step.objects.filter(belong_case__suite__config__name='套件1')
        print(steps)
```

接口开发-DRF入门

Django REST framework（以下简称 DRF或REST框架）是一个开源的 Django 扩展，提供了便捷的 REST API 开发框架，拥有以下特性：

- 直观的 API web 界面。
- 多种身份认证和权限认证方式的支持。
- 内置了 OAuth1 和 OAuth2 的支持。
- 内置了限流系统。
- 根据 Django ORM 或者其它库自动序列化。
- 丰富的定制层级：函数视图、类视图、视图集合到自动生成 API，满足各种需要。
- 可扩展性，插件丰富。
- 广泛使用，文档丰富。

安装djangorestframework

和django-restfamework（已安装则忽略）

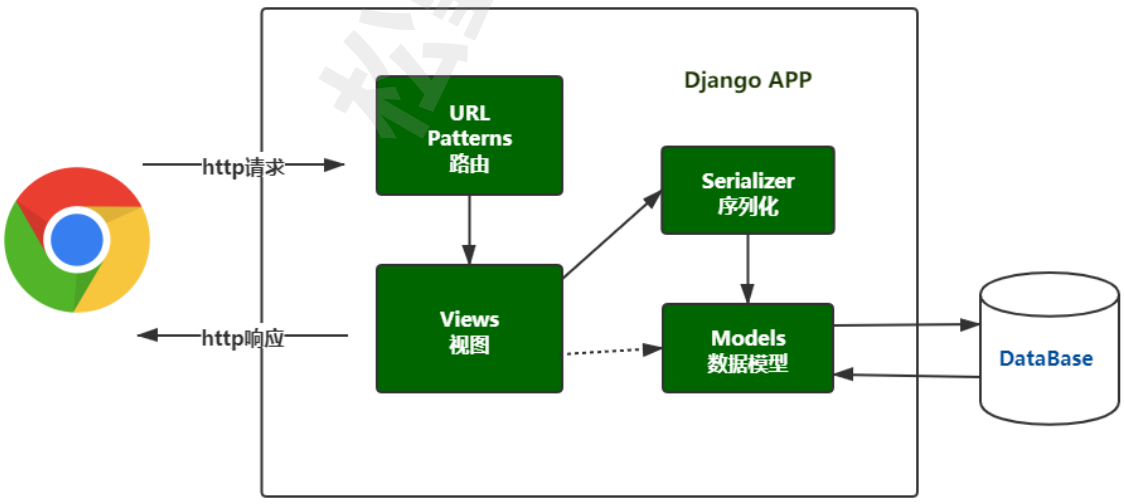
```
pip install djangorestframework
```

配置djangorestframework

autotpsite/settings.py

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'sqtp',
]
```

djangorestframework运行原理



相比于原生django开发的web应用，多了一层序列化器(Serializer)，如果用过Django表单(Form)，应该会对其原理有所了解，序列化器和表单都是基于Field进行字段验证，而Field都来自于rest_framework.fields模块，相当于把django的封装了一层。

DRF基本组件-Serializer

序列化（Serializer）是 DRF 的核心概念，提供了数据的验证和渲染功能，其工作方式类似于 Django Form

Serializer的作用是实现序列化和反序列化

所谓序列化就是将python数据对象转化成方便传输的文本格式，如：json/xml等

反序列化就是将这个过程反过来。

和models类似，序列化通常定义在应用程序下单独文件中serializer.py

如果采用了DRF模式开发django，那么model将直接和serializer交互。

感受序列化与反序列化（理解）

1.模型改造，添加元类 Meta

知识点：

模型的元数据，指的是“除了字段外的所有内容”，例如排序方式、数据库表名、人类可读的单数或者复数名等等。所有的这些都是非必须的，甚至元数据本身对模型也是非必须的。但是，我要说但是，有些元数据选项能给予你极大的帮助，在实际使用中具有重要的作用，是实际应用的‘必须’。

想在模型中增加元数据，方法很简单，在模型类中添加一个子类，名字是固定的`Meta`，然后在这个Meta类下面增加各种元数据选项或者说设置项。参考下面的例子：

```
from django.db import models

class Request(models.Model):
    ...

    class Meta:  # 模型元类的作用：为模型增加额外的信息，如模型对应表名，
        ordering = ['id']  # 数据根据ID排序
        db_table = ['reqquest']  # 模型对应的数据库表名，不设置则默认为app名_模型名
```

其他可以设置的Meta选项参考附录

2.创建1个序列化类

开发我们的Web API的第一件事是为我们的Web API提供一种将代码片段实例序列化和反序列化为诸如`json`之类的表示形式的方式。 在`sqtp的目录下创建一个名为`serializers.py`文件，并添加以下内容。

```
from rest_framework import serializers

from sqtp.models import Step, Request


class RequestSerializer(serializers.Serializer):
    method_choices = (  # method可选字段，二维元组
        (0, 'GET'),  # 参数1：保存在数据库中的值，参数2：对外显示的值
        (1, 'POST'),
        (2, 'PUT'),
        (3, 'DELETE'),
    )
```

```
    step = serializers.RelatedField(queryset=Step.objects.all(),allow_null=True)
    method = serializers.ChoiceField(choices=method_choices, default=0)
    url = serializers.CharField()
    params = serializers.JSONField(allow_null=True)
    headers = serializers.JSONField(allow_null=True)
    cookies = serializers.JSONField(allow_null=True)
    data = serializers.JSONField(allow_null=True)
    json = serializers.JSONField(allow_null=True)

    def create(self, validated_data):
        """
        根据提供的验证过的数据创建并返回一个新的`Snippet`实例。
        """
        return Request.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        根据提供的验证过的数据创建并返回一个新的`Snippet`实例。
        """
        instance.step =validated_data.get('step',instance.step)
        instance.method =validated_data.get('method',instance.method)
        instance.url =validated_data.get('url',instance.url)
        instance.params =validated_data.get('params',instance.params)
        instance.headers =validated_data.get('headers',instance.headers)
        instance.cookies =validated_data.get('cookies',instance.cookies)
        instance.data =validated_data.get('data',instance.data)
        instance.json =validated_data.get('json',instance.json)
```

序列化器类的第一部分定义了序列化/反序列化的字段。`create()`和`update()`方法定义了在调用`serializer.save()`时如何创建和修改完整的实例。

3.使用序列化类

在我们进一步了解之前，我们先来熟悉使用我们新的Serializer类。新建测试文件sqtp/test_serializers.py,

好的，像下面一样导入几个模块，然后开始创建一些代码片段来处理。

```
"""
@author: haiwen
@date: 2021/6/7
@file: test_serializers.py.py
"""
from django.test import TestCase
from sqtp.models import Step, Request
from sqtp.serializers import RequestSerializer
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

class TestRequestSerializer(TestCase):

    req1=Request.objects.create(url='/demo1',data={'name':'小
明','age':18,'addr':'nanjing'})
    req2=Request.objects.create(url='/demo1',params={'name':'小
明','age':18,'addr':'nanjing'})

    # 序列化:数据对象-->>python原生数据类型
```

```
    req1_serializer = RequestSerializer(req1)
    print(req1_serializer.data)  # 序列化数据存储在序列化对象的data属性中

    # 序列化:python原生数据类型-->>json
    content = JSONRenderer().render(req1_serializer.data)
    print(content)
```

执行测试

```
python manage.py test sqtp.test_serializers
```

返回

```
{'step': None, 'method': 0, 'url': '/demo1', 'params': None, 'headers': None,
'cookies': None, 'data': {'name': '
小明', 'age': 18, 'addr': 'nanjing'}, 'json': None}
b'{"step":null,"method":0,"url":"/demo1","params":null,"headers":null,"cookies":
null,"data":{"name":"\xe5\xb0\x8f
\xe6\x98\x8e","age":18,"addr":"nanjing"},"json":null}'
```

反序列化是类似的。测试类再新增以下代码，实现反序列化

```
    # 反序列化1: 将流（stream）解析为Python原生数据类型.
    import io

    stream = io.BytesIO(content)    # 构建1个steam流
    data = JSONParser().parse(stream)  # 转成python原生数据类型

    # 反序列化2: Python原生数据类型恢复成正常的对象实例。
    serializer = RequestSerializer(data=data)  # 序列化器
    print(serializer.is_valid())               # 校验数据是否合法
    print(serializer.validated_data)           # 查看数据对象
    serializer.save()                          # 保存数据
```

执行测试

```
python manage.py test sqtp.test_serializers
```

返回

```
...
True
OrderedDict([('step', None), ('method', 0), ('url', '/demo1'), ('params', None),
('headers', None), ('cookies', N
one), ('data', {'name': '小明', 'age': 18, 'addr': 'nanjing'}), ('json', None)])
```

除了序列化模型实例，序列化器还可以序列化查询结果集（querysets），只需要为serializer添加一个`many=True`标志。

```
    # 序列化查询结果集（querysets）
    serializers= RequestSerializer(Request.objects.all(),many=True)
    print(serializers.data)
```

改进序列化器（掌握）

以上案例是为了大家可以了解序列化器的工作过程，在实际的项目中不会写这么繁复的代码，我们可以用`ModelSerializer`代替`Serializer`类作为自定义序列化器的父类，减少重复代码的编写。

打开sqtp/serializers.py,使用`ModelSerializer`重构序列化类。

```
class RequestSerializer(serializers.ModelSerializer):
    class Meta:
        model = Request
        fields =
['step','method','url','params','headers','params','data','json']
```

其实，相应的字段已经隐含在类中了，继续在测试类中添加代码，查看序列化器的内部结构

```
print(repr(serializer))  # 打印序列化器类实例的结构(representation)查看它的所有字段
```

输出

```
RequestSerializer(data={'id': 13, 'step': None, 'method': 0, 'url': '/demo1',
'params': None, 'headers': None,
'data': {'name': '小明', 'age': 18, 'addr': 'nanjing'}, 'json': None}):
    id = IntegerField(label='ID', read_only=True)
    step = PrimaryKeyRelatedField(allow_null=True, queryset=Step.objects.all(),
required=False, validators=[<Un
iqueValidator(queryset=Request.objects.all())>])
    method = ChoiceField(choices=((0, 'GET'), (1, 'POST'), (2, 'PUT'), (3,
'DELETE')), label='请求方法', requir
ed=False, validators=[<django.core.validators.MinValueValidator object>,
<django.core.validators.MaxValueValida
tor object>])
    url = CharField(label='请求路径', max_length=1000, required=False)
    params = JSONField(allow_null=True, decoder=None, encoder=None, label='Url参
数', required=False, style={'ba
se_template': 'textarea.html'})
    headers = JSONField(allow_null=True, decoder=None, encoder=None, label='请求
头', required=False, style={'ba
se_template': 'textarea.html'})
    data = JSONField(allow_null=True, decoder=None, encoder=None, label='Data参
数', required=False, style={'bas
e_template': 'textarea.html'})
    json = JSONField(allow_null=True, decoder=None, encoder=None, label='Json参
数', required=False, style={'bas
e_template': 'textarea.html'})
```

`ModelSerializer`类并不会做任何特别神奇的事情，它们只是创建序列化器类的快捷方式：

- 一组自动确定的字段。
- 默认简单实现的`create()`和`update()`方法

DRF实践--视图编写（理解）

序列化器最终的作用是为视图提供转化后的数据，让我们看看如何使用我们新的Serializer类编写一些API视图。目前我们不会使用任何REST框架的其他功能，我们只需将视图作为常规Django视图编写。

编辑 sqtp/views.py，导入以下库

```
from django.http import HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.parsers import JSONParser
from sqtp.models import Request
from sqtp.serializers import RequestSerializer
```

编写一个视图可以返回所有的请求数据

```
def request_list(request):
    if request.method == 'GET':
        serializer= RequestSerializer(Request.objects.all(),many=True)
        return JsonResponse(serializer.data, safe=False)  # safe=False是为了支持{}以外的python对象转json
```

创建sqtp/urls.py, 写入路由

```
from django.urls import path
from sqtp import views

urlpatterns = [
    path('requests/',views.request_list)
]
```

总路由引入子路由 autotpsite/urls.py

```
from django.urls import path,include

urlpatterns = [
    path('',include('sqtp.urls'))
]
```

启动server服务并测试

```
python manage.py runserver
```

浏览器访问：http://127.0.0.1:8000/requests/

可以看到返回json格式的数据。

接口开发-DRF-进阶（掌握）

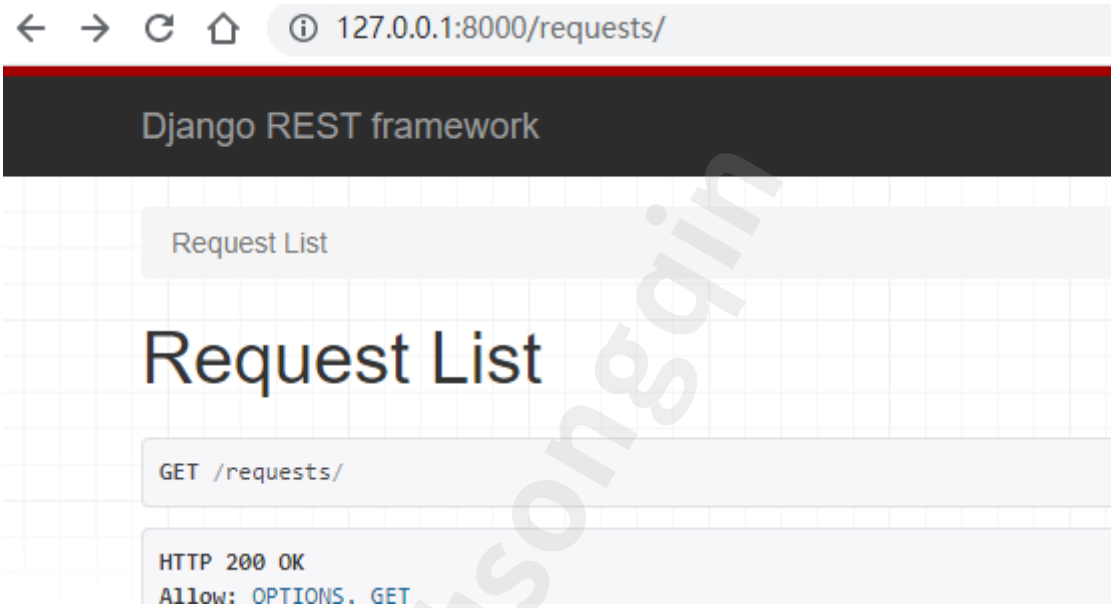
接口开发本质上是处理请求和响应，包括了处理请求参数，判断请求方法，处理响应字段，响应码等，本身是个枯燥的活，DRF框架为你提供自动处理这些枯燥工具的方法。先来看第一个工具，函数视图装饰器@api_view

```
from django.http import HttpResponse, JsonResponse
from rest_framework.decorators import api_view

from sqtp.models import Request
```

```python
from rest_framework.response import Response
from sqtp.serializers import RequestSerializer


@api_view(['GET'])
def request_list(request):
    if request.method == 'GET':
        serializer = RequestSerializer(Request.objects.all(), many=True)
        return Response(serializer.data)
        # safe=False是为了支持{}以外的python对象转json
```

浏览器访问：http://127.0.0.1:8000/requests/

哇塞，和单纯的显示数据相比，有网页内容了。



该装饰器的作用是 确保你在视图中接收到 `Request` 实例，并将上下文添加到 `Response`，以便可以执行内容协商。

包装器还提供了诸如在适当时候返回 `405 Method Not Allowed` 响应，并处理在使用格式错误的输入来访问 `request.data` 时发生的任何 `ParseError` 异常。

知识点：请求与响应

在纯django中，request和response只能实现有限的功能。REST框架对其作了扩展，其中：

请求对象（Request objects）

REST框架引入了一个扩展了常规 `HttpRequest` 的 `Request` 对象，并提供了更灵活的请求解析。`Request` 对象的核心功能是 `request.data` 属性，它与 `request.POST` 类似，但对于使用Web API更为有用。

```python
request.POST  # 只处理表单数据  只适用于'POST'方法
request.data  # 处理任意数据  适用于'POST'，'PUT'和'PATCH'方法
```

响应对象（Response objects）

REST框架还引入了一个 `Response` 对象，这是一种获取未渲染（unrendered）内容的 `TemplateResponse` 类型，并使用内容协商来确定返回给客户端的正确内容类型。

```
return Response(data)  # 渲染成客户端请求的内容类型。
```

默认情况下，Response把响应内容嵌套在默认的模板中返回出去，所以我们看到的内容是网页形式，而不是单纯的数据。

扩展-响应数据格式

如果我们要开发前后端分离的系统，又不能返回这种html格式的，需要json格式的该怎么处理呢?

REST框架充分考虑道了这一点，为了充分利用我们的响应不再与单一内容类型连接，我们可以为API路径添加对格式后缀的支持。使用格式后缀给我们明确指定了给定格式的URL，这意味着我们的API将能够处理诸如http://example.com/api/items/4.json之类的URL。

视图中添加一个`format`关键字参数。

```
@api_view(['GET'])
def request_list(request,format=None):
    if request.method == 'GET':
        serializer = RequestSerializer(Request.objects.all(), many=True)
        return Response(serializer.data)
        # safe=False是为了支持{}以外的python对象转json
```

修改路由文件 sqtp/urls.py

```
from django.urls import path
from sqtp import views
from rest_framework.urlpatterns import format_suffix_patterns
urlpatterns = [
    path('requests/',views.request_list)
]
urlpatterns = format_suffix_patterns(urlpatterns)
```

重启服务器，访问http://127.0.0.1:8000/requests/ ，默认情况下还是网页

访问http://127.0.0.1:8000/requests.json ，返回了json格式的数据，完美!

扩展-查询单个数据

前面我们开发的接口是批量列出，如果要开发列出单个数据，需要开发另外接口

```
@api_view(['GET'])
def request_detail(request,_id,format=None):
    """
    获取，更新或删除一个req实例。
    """
    try:
        req_obj = Request.objects.get(pk=_id)
    except Request.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)  # status提供常用http状态
码

    if request.method == 'GET':
        serializer = RequestSerializer(req_obj)
        return Response(serializer.data)
```

更新路由：sqtp/urls.py

```
urlpatterns = [
    path('requests/', views.request_list),
    path('requests/<int:_id>', views.request_detail)
]
urlpatterns = format_suffix_patterns(urlpatterns)
```

访问http://127.0.0.1:8000/requests/2

或 http://127.0.0.1:8000/requests/2.json

查看对应数据

其中数字代表对应数据的ID

附录

可用的Meta选项：模型 Meta 选项 | Django 文档 | Django (djangoproject.com)