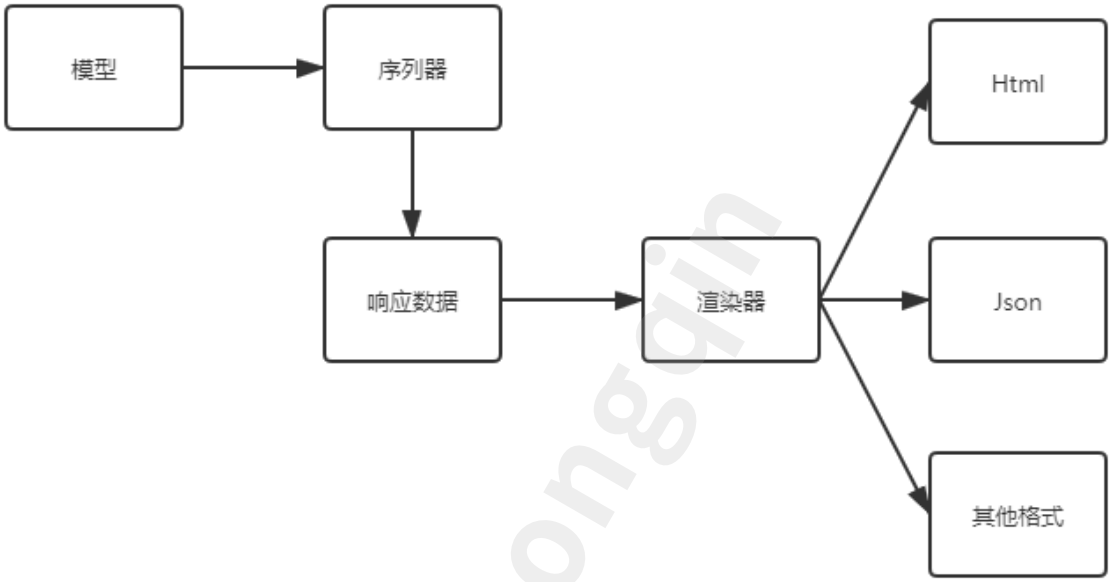
渲染器进阶

上节课我们了解了渲染器(renderer)的简单使用方法，这节课开始，我们深入了解下渲染器的原理和高级使用方法。

渲染器基本原理

回顾下REST渲染器基本原理



序列化在返回数据后并不是直接做为响应数据，而是经过渲染器的渲染，生成不同格式的响应内容，如html或json格式。REST本身支持HTML和json格式，有内置的渲染器。那么如果我们想要返回定制化的内容就要重写渲染器，按照我们指定的格式进行响应。

重构渲染器就是重写父类渲染器的render方法

```
render(self, data, accepted_media_type=None, renderer_context=None)
```

传递给 `.render()` 方法的参数是：

`data`

响应数据（序列化器的.data属性），等同于renderer_context[" `response` "].data的值

`media_type=None`

可选的。如果提供，这是由内容协商阶段确定的所接受的媒体类型。

根据客户端的 `Accept:` 头，这可能比渲染器的 `media_type` 属性更具体，可能包括媒体类型参数。例如 `"application/json; nested=true"` 。

`renderer_context=None`

可选的。如果提供，这是一个由view提供的上下文信息的字典。

默认情况下这个字典会包括以下键： `view` , `request` , `response` , `args` , `kwargs` 。

renderer_context["`view`"] 对应调用的视图函数

renderer_context["`request`"] 对应本次请求对象,包含所有请求数据，如请求头，请求参数等等

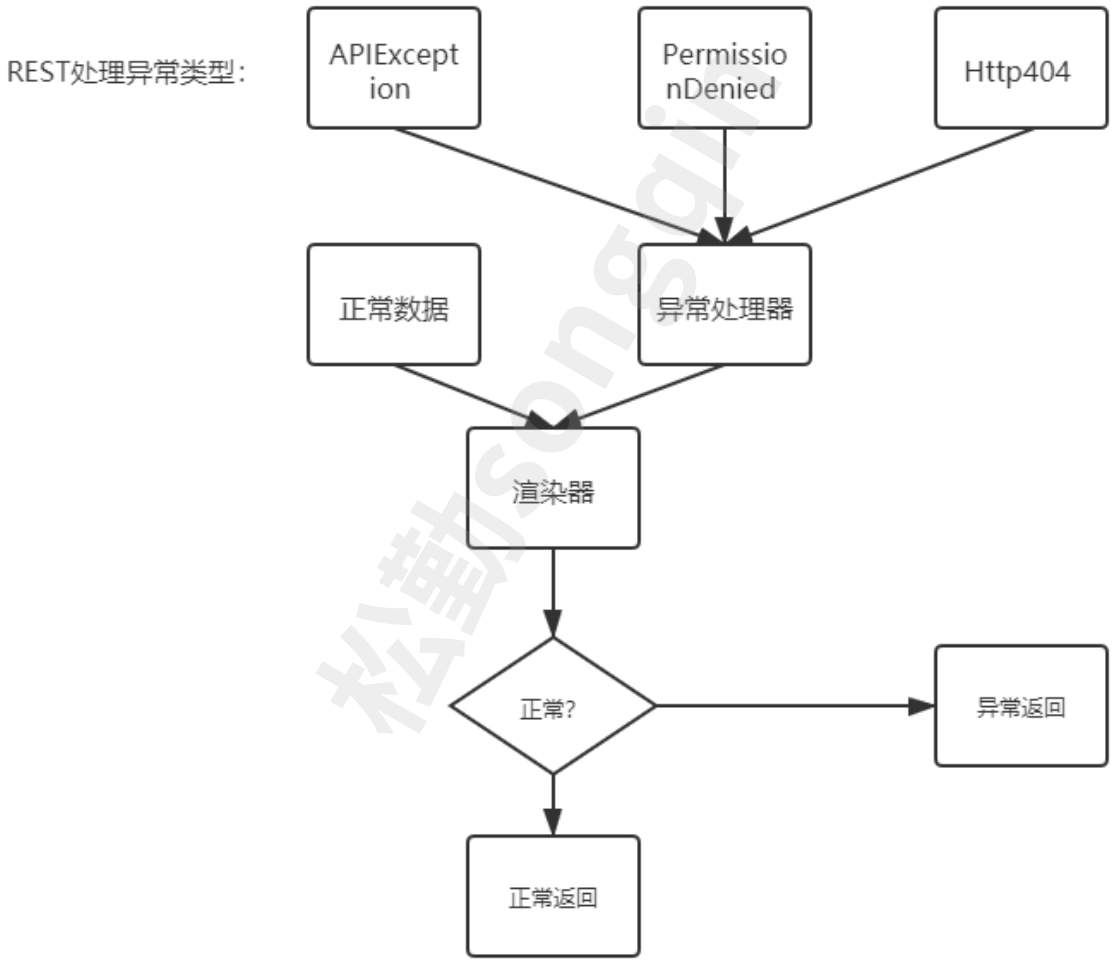
renderer_context["`response`"] 对应本次响应对象，包含所有响应数据，如响应头，状态码，响应数据等等

异常信息获取

当前情况下我们智能获取到正常情况下返回的数据，如果想返回异常信息，需要了解下REST异常处理机制

REST默认情况可以处理的异常有

- 在REST framework内部产生的`APIException`的子类异常。
- 原生Django的`Http404`异常.
- 原生Django的`PermissionDenied`异常.



如果我们想要获取异常数据，需要从异常处理器中提取，REST默认的是exception_handler，当触发异常时，可以获取到异常的响应信息，我们可以自由定义其格式。

使用方法是

1.配置异常处理模块

```
# rest框架配置
REST_FRAMEWORK = {
    # 全局配置异常模块
    'EXCEPTION_HANDLER': 'utils.exception.my_exception_handler',
    # 默认的渲染器
    'DEFAULT_RENDERER_CLASSES': (
        'utils.renderers.MyRenderer',
    ),
    'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema'
}
```

2.自定义异常处理

增加异常处理

```
# utils/exceptions.py
from rest_framework.views import exception_handler, Response

def my_exception_handler(exc, context):
    # 首先调用REST framework默认的异常处理，
    # 以获得标准的错误响应。
    response = exception_handler(exc, context)
    if response:
        # 成功捕获到异常的情况
        response.data['msg'] = 'error'  # 标记
        response.data['retcode'] = response.status_code  # 状态码
        response.data['error'] = str(exc)  # 详细错误原因
        response.data.pop('detail') # detail的异常信息比较简单
    return response
```

更新渲染器逻辑

```
# 通用返回过滤器
from rest_framework.renderers import JSONRenderer
# 继承空返回JSON的渲染器
class MyRenderer(JSONRenderer):
    # 重构 render方法
    def render(self, data, accepted_media_type=None, renderer_context=None):
        # 默认把data作为响应数据
        resp_content = data
        if renderer_context:
            status_code = renderer_context['response'].status_code  #响应状态码
            if str(status_code).startswith('2'): # 以2开头表示响应正常
                # 判断响应内容是否是列表形式，如果不是列表形式，则变成列表形式以对应接口要求
                if not isinstance(resp_content,list):
                    resp_content = [resp_content]
                res =
{'msg':'success','retcode':status_code,'retlist':resp_content}
                # 返回父类方法
                return super().render(res,accepted_media_type,renderer_context)

        return super().render(resp_content,accepted_media_type,renderer_context)
```

swagger在线接口文档

目前为止，我们的接口开发到了一定的阶段，已经初具规模，在和前端对接之前，需要规范化我们的接口文档。如果纯手写的话工作量大且重复枯燥，因此，我们可以用工具帮助我们实现接口文档的自动生成。

Django REST Swagger 项目已经不维护了，并且不支持最新的Django，所以我们选择 **drf-yasg**项目作为接口文档生成器。yasg的功能非常强大，可以同时支持多种文档格式。

快速开始

1.安装

```
pip install -U drf-yasg  # 安装最新版 drf-yasg
```

2.注册

老样子，这也属于django的插件，因此需要注册到配置文件中 settings.py

```
INSTALLED_APPS = [
   ...
   'django.contrib.staticfiles',  # required for serving swagger ui's css/js
files
   'drf_yasg',
   ...
]
```

3.配置路由

```
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="SQTP API",
        default_version='v1',
        description="SQTP接口文档",
        terms_of_service="https://www.songqin.net",
        contact=openapi.Contact(email="haiwen@sqtest.org"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    ...


 path('swagger/',schema_view.with_ui('swagger',cache_timeout=0,),name='schema-
swagger-ui'),  # 互动模式
    path('redoc/',schema_view.with_ui('redoc',cache_timeout=0),name='schema-
redoc'), # 文档模式
    ...
]
```

4.访问互动

访问http://127.0.0.1:8000/swagger/ 进入互动模式，该模式下可以对接口发起请求

访问http://127.0.0.1:8000/redoc/ 进入文档模式，该模式为静态模式，展示详细的接口内容

定制化用法（viewset模式）

我们当前的接口文档是没有定制化注释的，比如某个接口的功能是什么，虽然现在都是增删改查，从名称根据Rest风格就能猜测出来，但如果是些定制化的接口就需要加些注释了。

比如函数视图，采用swagger_auto_schema装饰器修饰视图函数

```
from drf_yasg.utils import swagger_auto_schema

@swagger_auto_schema(method='GET',operation_summary='定制化
API',operation_description='接口描述。。')
@api_view(['GET'])
def customer_api(request):
    return Response(data={"retcode":status.HTTP_200_OK,'msg':'building...'})
```

如果我们的视图采用类或者视图集，视图函数本身是继承父类，没有出现在我们的代码中该如何自定义接口描述呢?

这时，我们可以采用django装饰器配合swagger的装饰器来实现，直接装饰类视图。

```
from django.utils.decorators import method_decorator
from drf_yasg.utils import swagger_auto_schema

@method_decorator(name='list', decorator=swagger_auto_schema(
    operation_description="列出所有步骤数据"
))
@method_decorator(name='create', decorator=swagger_auto_schema(
    operation_description="创建步骤"
))
@method_decorator(name='update', decorator=swagger_auto_schema(
    operation_description="更新步骤"
))
@method_decorator(name='destroy', decorator=swagger_auto_schema(
    operation_description="删除步骤"
))
@method_decorator(name='retrieve', decorator=swagger_auto_schema(
    operation_description="提取单个步骤数据"
class StepViewSet(viewsets.ModelViewSet):
    queryset = Step.objects.all()
    serializer_class = StepSerializer
```

就是这么丧心病狂的叠加装饰器，一个装饰器修饰一个方法，如果多个就多层叠加...

前端对接

现在接口文档整活了，对应的核心接口也开发的差不多了，接下来我们希望看到自己的实际项目前端页面。

因为我们的项目是前后端分离的，我们后端开发不涉及前端页面开发，前端开发人员按照API文档去访问后端数据，然后渲染页面。

前端开发好对应的功能后，和后端进行配合就可以看到效果了。

前端环境其实就是 一些前端的代码和资源文件，包括 js文件、html文件、css文件 还有 图片视频文件等。

我们模拟前端团队开发的 前端 系统可以在百度云盘当天的课程里找到。

下载好dist压缩包后，可以解压到项目根目录下面，这个目录下面就是前端的代码资源文件。

Django的开发环境也可以从浏览器访问这些前端的资源文件。

但是前端文件都是静态文件，需要我们配置一下Django的配置文件， 指定http请求如果访问静态文件，Django在哪个目录下查找。

注意，接下来我们配置 Django 静态文件服务， 是 `开发时` 使用的 一种 `临时方案` ，性能很低，这是方便我们调试程序用的。

前面讲过，正式部署web服务的时候，不应该这样干，应该采用其它方法，比如Nginx等。后面的教程会有详细的讲解如何使用Nginx 和 Django 组合使用。

现在，请打开 `autotpsite/urls.py` 文件，在末尾 添加一个

```
+ static("/", document_root="dist")
```

并添加如下声明

```
# 静态文件服务
from django.conf.urls.static import static
```

最终，内容如下

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',include('sqtp.urls'))
] + static("/", document_root="dist")
```

最后的

```
+ static("/", document_root="dist")
```

就是在url 路由中加入 前端静态文件的查找路径。

这样如果 http请求的url 不是以 api/project 开头， Django 就会认为是要访问 dist目录下面的静态文件。

好了，现在我们 运行如下命令，启动Django 开发服务器

```
python manage.py runserver 8081
```

然后我们打开浏览器，输入如下网址：

```
http://127.0.0.1:8000/index.html
```

就会出现 管理员操作界面，如下

当我们访问html页面时，js代码会自动请求后台数据，渲染到当前页面上，为了适配前端，减少改动工作，先将后台URL进行改动

```
urlpatterns = [
    path('api/',include(router.urls)),  # 以api开头访问
    ...
]
```

此时访问：http://127.0.0.1:8000/testcase.html，可以看到后台返回的数据

目前前端这里只适配了一个页面，并且信息量比较少，其他页面需要等到后端功能完善再进行适配工作。

接口开发完善

嵌套字段

为了提供更多的信息给前端，需要对目前的接口进行开发完善。以用例为例，当前的查询接口返回的消息是这种格式：

```
{"msg":"success","retcode":200,"retlist":
[{"id":1,"file_path":"haiwen_Test.yml","config":1,"suite":null}]}
```

其中config只有id表示，没有具体的信息，如果前端想要获取config信息将会比较麻烦，所以直接通过后端提供相关数据效果比较好点。

由于我们使用了REST框架，这个问题交给序列化器就可以处理了。目前我们默认的序列化器很简单，只简单的定义了要展示的字段和对应模型。那么，默认情况下序列化器只会提取外键字段的id作为默认值，所以我们需要额外定义需要展示嵌套字段的关联数据。

这个问题处理起来也非常简单，我们只需指定config为对应的序列化器对象即可

```
# 配置
class ConfigSerializer(serializers.ModelSerializer):
    class Meta:
        model = Config
        fields = '__all__'

# 用例
class CaseSerializer(serializers.ModelSerializer):
    config = ConfigSerializer()  # config字段为Config序列化器，REST会自动提取其值
    class Meta:
        model = Case
        fields = '__all__'
```

注意两个类的顺序，由于python引用前需要先定义，所以ConfigSerializer要在上面。

再来看下最新的返回，已经展示了config嵌套字段。


```
{
  "msg": "success",
  "retcode": 200,
  "retlist": [
    {
      "id": 1,
      "config": {
        "id": 1,
        "name": "用例1",
        "base_url": "http://localhost",
        "variables": null,
        "parameters": null,
        "verify": false,
        "export": null
      },
      "file_path": "haiwen_Test.yml",
      "suite": null
    }
  ]
}
```


displayname

第二个案例我们来看下，请求数据的展示效果


```
{
  "msg": "success",
  "retcode": 200,
  "retlist": [
    {
      "id": 1,
      "method": 0,
      "url": "/demo1",
      "params": null,
      "headers": null,
      "cookies": null,
      "data": {
```

```
        "age": 18,
        "addr": "nanjing",
        "name": "小明"
      },
      "json": null,
      "step": 1
    }
  ]
}
```

不知道大家有没有发现问题？仔细看看

...

对了，method这里显示的不是GET/POST/PUT/DELETE,而是0，1，2，3 这种实际存储在数据中的值，因为我们定义了choice选择字段，但是并没有显示我们要的注释。所以这里需要修改成显示成对应的可读内容。

解决这个问题之前，我们可以看下django原生ORM是如何显示choice字段的可读内容的

python manage.py shell 先进入django shell

```
>>> from sqtp.models import Request
>>> req1 = Request.objects.all().first()
>>> req1.method   # 默认情况下还是实际值
1
>>> req1.get_method_display()   # 采用get_field_display 方法，field对应choice字段的名称
'POST'
```

这个时候我们再进入序列化器，修改字段的获取方式

```
# 请求模型的序列化器
class RequestSerializer(serializers.ModelSerializer):
    method = serializers.SerializerMethodField() 自定义字段序列化返回方法

    def get_method(self, obj):    # rest框架获取method时，自动调用该方法
        return obj.get_method_display()  # 返回choice的 displayname而不是实际值

    class Meta:
        model = Request  # 指定对应的模型
        fields = '__all__'  # 显示对应模型的所有字段
        # 定义可以显示和操作的字段
```

此时重新访问接口，method字段返回了可读内容

```
{
  "msg": "success",
  "retcode": 200,
  "retlist": [
    {
      "id": 1,
      "method": "GET",
      "url": "/demo1",
      "params": null,
      "headers": null,
      "cookies": null,
```

```
      "data": {
        "age": 18,
        "addr": "nanjing",
        "name": "小明"
      },
      "json": null,
      "step": 1
    }
  ]
}
```