用例执行系统整体思路

首先目前hr3执行用例采取的方法是将json文件转化成py文件再用pytest去运行，我们可以做的就是将数据库中用例数据取出，再转化成对应格式的json文件即可。

用例的增删改查问题

目前用例以及其他模型字段都发生了改变，和现有业务不符，需要进行修改。

增加

首先序列化器部分的展示数据需要进行变动

```
class CaseSerializer(serializers.ModelSerializer):
    config = ConfigSerializer()  # 只需要序列化输出，则定义read_only = True
    teststeps = StepSerializer(many=True, )  # 以列表形式展示 many=True

    class Meta:
        model = Case
        fields = ['config', 'teststeps']
```

前端人员在架构师的指示下适配好了页面，根据API规则发送对应的数据：

```
{desc: "123", name: "case001", project_id: "1"}
```

收到服务器异常：

```
rest_framework.exceptions.ValidationError: {'teststeps':
[ErrorDetail(string='This field is required.', code='required')]}
```

默认的情况下REST框架会调用自带的验证机制，当我们传递的数据和序列化器存在出入时，框架会抛出异常

teststeps字段不作为接口的入参，只做为出参，可以设置read_only = True，这样在校验入参时，不会因为没有传递teststeps而报错了

```
# 用例
class CaseSerializer(serializers.ModelSerializer):
    config = ConfigSerializer()  # 只需要序列化输出，则定义read_only = True
    teststeps = StepSerializer(many=True, read_only=True)  # 以列表形式展示
many=True
    ...
```

新增数据调用的是序列化器的create方法，如果默认的不满足要求就要重新父类方法:

```
# 用例
class CaseSerializer(serializers.ModelSerializer):
    config = ConfigSerializer()  # 只需要序列化输出，则定义read_only = True
    teststeps = StepSerializer(many=True, read_only=True)  # 以列表形式展示
many=True
```

```
    project_id = serializers.CharField(write_only=True)  # 只需要反序列化输入

    def create(self, validated_data):
        config_kws = validated_data.pop('config')  # 从请求参数弹出config参数用于创建
config
        project = Project.objects.get(pk=validated_data['project_id'])
        config = Config.objects.create(project=project,**config_kws) #注意关联
project
        file_path = f'{project.name}_{config.name}.json'  # 项目名+用例名.json
        instance = Case.objects.create(file_path=file_path, config=config)
        return instance

    class Meta:
        model = Case
        fields = ['config', 'teststeps','project_id','file_path']
```

测试新增，查看数据是否正确创建。

修改

接下来，定制化修改数据的方法，修改采用的是序列化器的update方法

```
    def update(self, instance, validated_data):
        config_kws = validated_data.pop('config')  # 从请求参数弹出config参数用于创建
config
        project = Project.objects.get(pk=validated_data['project_id'])
        config_kws['project'] = project
        # 此时instance为Case数据对象

conf_serializer=ConfigSerializer(instance=instance.config,data=config_kws)  # 调
用序列化器更新config数据
        if conf_serializer.is_valid():
            conf_serializer.save()
        else:
            raise ValidationError(conf_serializer.errors)

        # 更新case数据
        instance.file_path = validated_data['file_path']
        return instance
```

如果要连同步骤一期更新则需要更新与之相关的序列化器

```
class StepSerializer(serializers.ModelSerializer):
    request = RequestSerializer()
    belong_case_id = serializers.IntegerField(write_only=True,required=False)
    def create(self, validated_data):
        request_kws= validated_data.pop('request')
        serializer = RequestSerializer(data=request_kws)
        if serializer.is_valid():
            req_obj=serializer.save()
        else:
            raise ValidationError(serializer.errors)

        step_obj = Step.objects.create(testrequest=req_obj,**validated_data)
        return step_obj
```

```
    class Meta:
        model = Step
        fields = ['name', 'variables', 'request', 'extract',
'validate','belong_case_id']
```

更新用例的update方法

```
class CaseSerializer(serializers.ModelSerializer):
    ...
    def update(self, instance, validated_data):
        config_kws = validated_data.pop('config')  # 从请求参数弹出config参数用于创建
config
        project = Project.objects.get(pk=validated_data['project_id'])
        config_kws['project'] = project
        # 此时instance为Case数据对象

conf_serializer=ConfigSerializer(instance=instance.config,data=config_kws)  # 调
用序列化器更新config数据
        if conf_serializer.is_valid():
            conf_serializer.save()
        else:
            raise ValidationError(conf_serializer.errors)
        # 更新step数据
        steps_kw = validated_data.pop('teststeps')
        for kw in steps_kw:
            kw['belong_case_id']=self.data['id']
            ss=StepSerializer(data=kw)
            if ss.is_valid():
                ss.save()
            else:
                raise ValidationError(ss.errors)
        # 更新case数据
        instance.file_path = validated_data['file_path']
        return instance

   ...
```

删除

暂时使用框架自带的功能

查询

前端展示需要project的数据，需要从config中获取，因此config的project数据需要嵌套展示

```
class ConfigSerializer(serializers.ModelSerializer):
    project = ProjectSerializer(required=False)

    class Meta:
        model = Config
        fields = ['name', 'base_url', 'variables', 'parameters', 'verify',
'project']
```

知识点

一、序列化器中的类属性字段

序列化中所定义的类属性字段，一般情况下与模型类字段相对应
默认情况下，这些类属性字段既可以进行序列化输出，也可以进行反序列化输入
不需要输入（反序列化）、输出（序列化）的字段，则不需要定义，**定义的字段则必须出现在fields列表中**
只需要反序列化输入，则定义write_only = True
只需要序列化输出，则定义read_only = True
响应的参数如果是多个查询集，需要在JsonResponse()中传参many=True
label：当前字段在前端的api页面中所显示的字段名称
allow_null = False：当前字段是否允许传None，默认是False（必填字段False，反之则True）
allow_blank = False：当前字段是否运行为空，默认是False（必填字段False，反之则True）

required=False：当前字段允许不传，默认是True（必填字段True，反之则False）

二、反序列化_校验机制

调用序列化器对象的is_valid()方法，校验前端参数的正确性，不调用则不校验
校验成功返回True、校验失败返回False
is_valid(raise_exception = True)：校验失败后，则抛出异常
当调用is_valid()之后，才能调用序列化器对象的errors属性，内容为校验的错误提示（dict）
在views.py中，如果传参进行了输入反序列化的话，那么需要调用的是经过校验后的数据，比如说新增数据，应该是：xxx类.objects.create(**serializer.validated_data)

在视图集（ViewSet）中，REST都默认调用了is_valid()方法来校验入参。