DRF框架

CBV

源码涉及的py知识点
- 闭包
- 类方法、实例方法 - 自动传参
  思考: 何为方法?何为函数?关键在于是否自动传参
- super
- *args, **kwargs 作为形参和实参的区别

视图类.as_view() 执行这个类方法,得到一个闭包.
路由匹配成功后, 会自动加括号传递路由参数,执行该闭包! 大体干了两件事:
1. self = 视图类()
2. self.dispatch()

self.dispatch

源码涉及的py知识点
- try..except..
- 反射 - 反射的属性查找规则 self-父类-..-object
  注: 变量的变量查找规则 作用域LEGB依次找
  <不同的作用域相同的变量名不会引起冲突>

```
self.args = args
self.kwargs = kwargs
Django request的二次封装
self.request = request
try:  注:★它们向上raise的异常都能在这里捕获到!
    版本
    认证  源码切入点request.user
    权限
    限流
    反射执行视图函数
except:
    返回
```

看似没毛病,但有一点很容易忽略!
经过认证、权限、限流后,才会执行 当前请求相关的视图函数.
这意味着,drf框架的源码里,可没管当前请求是GET、POST哦.. 所有请求 通通都使用的是 局部-视图类/全局-配置文件 的配置..
(eg: 即配置的认证类作用于视图类下的所有 视图函数/接口)

那如果 我想 GET请求使用 AB两个认证类, POST请求使用 CD这两个认证类, 就需要重写下源码啦!!
嘻嘻 (*¯︶¯*)

二次封装request

源码涉及的py知识点
- 前面加单下划线,约定俗成 内部使用
  前面加双下划线 - 封装
- @property装饰器 - 将方法变成属性
  扩展思考:
  封装+暴露方法 -- 封装+@property -- 描述符
  注: 描述符 _set_、_get_
- 伪装术
  - 组合 - 在B实例所在类的_init_方法中实现, 将A实例作为B实例的一个属性 、
  - _getattr_、_getattribute_

⚠注意两个地方:
1. 视图函数中执行 request.xxx
   先去drf_request中查找,若没有,就去django_request中查找.
   若还是没有, 会报错AttributeError, 很神奇的是 报的是 在 drf_request没有找到该属性 的错误!!
2. query_params方法

此部分源码的_getattr_中使用了getattr(), 乍一看是错的.
但你得分析, 调用_getattr_的是 drf的request, 而里面的getattr反射的对象是django的request!

版本

⚠注意两个地方:
1. 三个渠道传递版本信息 ?params、路由、请求头
   设置版本类后,若版本合法性验证通过, request.version中可取到
2. 同一接口要同时支持网页端和小程序, 在视图函数中判断request.version . 然后写相应的代码即可!

认证

源码涉及的py知识点
- 延迟加载
  - 资源仅在第一次调用时就会载入内存, 往后再次调用加载内存里的即可
  - 利用if、@property和反射来实现
- for循环的可迭代对象bool值为假时, for循环直接结束!
- oop的继承,属性查找顺序 跟 反射时属性查找顺序一样
  实例自身-实例所在的类-父类--直到object
- 多态 - 抽象类可约束 多个类中名字相同的函数

⚠ 开发API过程中, 有些接口需要登录才能访问, 有些无需登录. drf中的认证组件主要就是用来实现此功能!
登陆成功后会有个凭证 token, 凭证可能出现的地方?params、请求头等

★ 一定一定要分析清楚哪些认证类作用于当前请求:
step1> 搞清有哪些认证类
   先 该请求所在视图类关于认证的局部配置 - 无局部看全局 项目关于认证的全局配置
step2> 这些认证类哪些是作用于当前请求呢?
   这关乎是否重写源码的get_authenticators方法, (※ 这是一个扩展点)
   不重写的话, 第一步搞清楚的那些认证类会无差别作用于 视图类下的 所有接口/所有请求

认证流程"多个认证类" — 理清楚作用于当前请求的那些认证类后, 开始进行认证流程
- 伪代码如下:

```
for 认证类 in [认证类1,认证类2,认证类3]:
    user_auth_tuple = 认证类.authenticate(request)
    if user_auth_tuple:
        user, auth = user_auth_tuple
```

- 不进行认证
  - 项目局部和全局都没进行认证类的配置, 会使用的是drf默认配置的认证类 - 它两返回的都是None
  - 局部配置 authentication_classes = []
    注: 源码中直接for循环结束
  - (匿名访问)
- 进行认证
  - 认证成功
    - (user用户信息, auth Token信息)
  - 认证失败
    - 抛出异常
    - 所有认证类都返回None
      - 匿名访问
  - 或的关系 某个认证类 - 认证成功 、抛出异常 后续认证类都不用管
- ok, 至此, 除了认证失败抛出异常会被dispatch中的try..except.. 捕获; 认证流程中的其它情况皆可进入权限!!

关于视图函数中打印 request.user 、request.auth 的值: (该处讨论匿名访问!!)
>> 这涉及 UNAUTHENTICATED_USER & UNAUTHENTICATED_TOKEN
   规则: 优先使用项目setting中配置 - 没有配则使用drf-setting中默认的配置

注: drf默认的 - 匿名访问时, request.user 其值是AnonymousUser ; request.auth 其值是 None

补充: 认证返回的http状态码 、认证失败,返回的结果 , 详见笔记. ^_^

权限

源码涉及的py知识点
- A类实例方法的形参是B类的实例
  eg: 权限类的has_permission的view参数是视图类的实例
- not and or
- 继承 (源码扩展点"或"关系用到了它)
- for-else else子句在for循环正常完成时执行

⚠ 在开发API过程中, 有些接口的访问必须同时满足条件A、B、C; -- "且"关系 源码中已经实现
在开发API过程中, 有些接口的访问只需满足其中任意一个条件. -- "或"关系 需要修改源码进行扩展!!
(或关系举个栗子: 比如角色,该接口只有员工或经理才能访问)

★ 一定一定要分析清楚哪些权限类作用于当前请求!! 过程同认证..

权限流程"多个权限类" — 理清楚作用于当前请求的那些权限类后, 开始进行权限流程
- 伪代码如下:

```
for 权限类 in [权限类1,权限类2,权限类3]:
    if not 权限类.has_permission(request,视图类实例):
        raise 抛出异常
```

- 不进行权限
  - 项目局部和全局都没进行权限类的配置, 会使用的是drf默认配置的权限类 - 它返回的是True
  - 局部配置 permission_classes = []
    注: 源码中直接for循环结束
- 进行权限
  - 权限成功
    - 所有的权限类的has_permission都返回True.
  - 权限失败
    - 任一的权限类的has_permission返回False
  - "且" 的关系!!
- ok, 至此, 除了权限失败 权限流程中的其它情况皆可进入下一步, 限流!

针对权限流程中 权限失败时 drf权限源码中会抛出的异常 的剖析! (该异常同样会被dispatch中的try..except..捕获)
>> 这要结合 认证的流程. 回顾下,认证如何才能走到权限?大体分为两种 -- 匿名访问 、认证成功.
   - 认证成功 drf权限源码抛出的异常 -- raise PermissionDenied 没有执行该操作的权限!
   - 认证匿名访问 drf权限源码抛出的异常 -- raise NotAuthenticated 没有提供身份验证凭据!

※ 扩展点!! 权限扩展 "或" 关系 -- 重写check_permissions方法,结合继承更好
怎么个继承法? eg: 视图类OrderView继承APIView, 权限的验证就是且的关系; 若继承BaseApiView, 权限的校验就是或的关系!!

补充: 权限返回的http状态码 、权限不通过,返回的结果 , 详见笔记. ^_^

限流

⚠ 限流/限制访问频率.
开发过程中,某个接口不想让用户访问过于频繁,就需要用到限流机制.

++限流的基本逻辑++
假设限制 3次/10min <"用户ID" 维护一个 该用户ID 访问该接口的时间点列表/访问记录 栗如- "9123563":[16:43, 16:42, 16:30, 15:20]>
我们设定新加入的时间点是插入到列表下标为0的位置.
-1-> 获取当前时间 16:45
-2-> 当前时间-10分钟=计数开始时间 16:35
-3-> 剔除列表中时间点小于 16:35 的数据, 列表中剩余的数据 [16:43, 16:42]
-4-> 计算列表的长度. 若超过,错误; 若未超过,可以访问.
len([16:43, 16:42]) < 3 所以可以访问, 并将16:45这个时间点放到列表中 [16:45, 16:43, 16:42]
若超过了,就需算最早的访问记录失效需要多久,公式: 时间间隔 - (当前时间戳 - 历史中最早访问的时间戳)

★关于限流,我们先达成点共识:
- 我们绝大多数场景下都选择进行局部配置! 几乎不会全局配置.
- 每个限流类的get_cache_key方法返回的唯一标识都应是不一样的!!
  该唯一标识对应redis中的一个key值. eg: '唯一标识':[时间戳1,时间戳2,时间戳3]

按照 笔记中 限流的实现步骤 + 参考小demo 实现限流基本的代码编写后.
★ 一定要明确两个东西:
1> 分析清楚哪些权限类作用于当前请求!! 过程同认证.
2> 作用于当前请求的这些权限类 还作用于 哪些 请求/接口! (明确这个超级重要!)
   因为限流类A作用于接口1和接口2, 那么无论是访问接口1还是访问接口2, 都会消耗限流A的速率的一次访问次数

限流流程"多个限流类" — 明确上方提到的两方面东西后, 开始进行限流流程
- 伪代码如下:

```
for 限流类 in [限流类1,限流类2,限流类]:
    等待时间列表 = []
    if not 限流类.allow_request(request,视图类实例):
        等待时间列表.append(限流类.wait())

if 等待时间列表:
    max(等待时间列表)
```

  - 每个限流类都会继承SimpleRateThrottle
  - allow_request的过程就是上面提到的限流的基本逻辑
    该方法返回True就是该限流类通过,返回False不通过!
- 不进行限流
  - 项目局部和全局都没进行限流类的配置, 会使用的是drf默认配置的限流类 - 默认为空列表,哈哈哈
  - 局部配置 throttle_classes = [] 注: 源码中直接for循环结束
  - 若限流类的频率设置为None 那么相当于该限流没设置;
  - 若get_cache_key方法的返回值为None,那么相当于该限流没设置;
- 进行限流
  - 限流通过
    - 所有限流类的allow_request都返回True
  - 限流不通过
    - 收集不通过的限流类的等待时间, raise 抛出异常, 异常信息中提示最大等待时间!
- ok, 至此, 除了限流不通过抛出异常会被dispatch中的try..except.. 捕获; 限流流程中的其它情况皆可往下走!!

++限流的应用场景 and 解决方案++:
Q1: [场景1] 视图类A中的接口1不需要限流,接口2需要限流.
A1: 视图类A中重写get_throttles方法!! 借助self.request.method来实现. (认证、权限同理)

Q2: [场景2] 视图类A的接口1限流频率 3/m、视图类A的接口2限流频率 5/m+200/h (即不仅要限制每分钟的请求次数,还需要限制每小时的)
A2: 视图类A中重写get_throttles方法!! 借助self.request.method来实现; 5/m+200/h 给接口2配置两个限流类就可以实现!

Q3: [场景3] 视图类A中的接口1在认证成功后限流频率是10/m, 匿名用户/未认证时限流频率是2/m
A3: 视图类A中重写get_throttles方法!! 借助self.request.method + self.request.user来实现.

※ 扩展点!! 自定义限流不通过时, 返回信息的格式. 解决关键在于, 自己写一个类来继承 APIException! (注: _init_里的detail)

※ 扩展点!! allow_request方法默认返回值只有True和Fasle, 我们新增一种 raise.
当前限流类不允许访问时,不返回False,而是直接抛出异常APIException类型的异常. 该异常的抛出会导致循环结束,程序终止.
也就是说, '[<后续的限流类都不会执行,不会管啦>]' . 异常会被APIView.dispatch里的try..except..捕获到!!

补充: 详见笔记. ^_^
-1- redis中某条数据的更新有两方面的操作
-2- 思考等待时间
-3- 思考下:若攻击者找了10万个手机号进行访问, 如何限制呢? 可以设置两个限流类 IP相关+ID相关.
-4- UserRateThrottle 、AnonRateThrottle

执行视图函数

- 前后端交互
- 解析器
- ORM原理
- 序列化器
  - 纯序列化
  - 验证、验证+存储、验证+存储+序列化、验证+序列化
- 分页器
- 视图组件
- 路由组件
- 过滤器

前后端交互

前端向后端发送请求, 可以有4种方式:
-1- 直接输入url -2- a标签 -3- form表单 -4- ajax

请求类型: GET、POST (注:POST比GET多一个请求体)

前端向后端传递数据,数据可在的位置:
-1- url ?params中 、url的路由参数中
-2- 请求头中
-3- 请求体中 - ★三种编码方式: json、urlencoded、formdata

ORM原理

源码涉及的py知识点
- import 包 我们可以在_init_.py文件中将本模块的所有文件都导入
- ★ 元类
- 继承dict+_setattr_+_getattr_ 使得既能[]取值,也能 . 取值
- 装饰器 用来实现连接数据库,执行构建好的sql

++简单阐述下原理 / 切入点++
用类创建表结构 <==> ORM表/类中 得有相应的表信息+每列字段的特征
用sql语句新增记录 <==> ORM表/类的实例 得有该记录的内容
用已有的信息(ORM表和ORM表的实例)去构建sql, 然后用pymysql连接数据库执行该sql语句.

解析器

其实可以啥也不用配置, 注意几点即可:
1> drf中可以使用 `request.data` 获取请求体中的数据(三种编码格式的数据都可以通过它取到).
2> 关于数据格式 可以看 前后端交互部分数据格式的截图, 一目了然!
3> 关于drf解析器的核心原理:
   -1- 通过DefaultContentNegotiation类的select_parser方法,将请求头匹配的 解析器 筛选出来.
   -2- 然后解析器使用parse方法,解析body里的数据.

序列化过程

源码涉及的py知识点
- 计数器 所有的对象都看得到的且值都是一样的,这得是个类属性.
- kwargs.pop
- MutableMapping + _setitem_ + _getitem

■ 序列化器类定义阶段:
-1- 创建字段对象, 每个字段对象都有实例属性_creation_counter 表明其创建的先后顺序
-2- 元类创建序列化器类, 剔除 序列化器类中"Field字段类型的类变量" 并加到 新增的类变量_declared_fields中,其余类变量Meta等不变

■ 序列化器类类实例化阶段:
单条 - ser1 = InfoSerializer(instance=instance, many=False) - InfoSerializer类的实例
多条 - ser2 = InfoSerializer(instance=queryset, many=True) - ListSerializer类的实例, ser.child = ser1

ser.data 开始序列化
■ 单条数据序列化
-1- 根据序列化器类中Meta的配置, 看fileds或exclude 获取到需要处理的字段名
-2- 根据已获得的字段名找到对应的字段对象. 分为两组, 一组优先级高于第二组
   - 第一组, 字段类型的类变量 保留该字段对象
   - 第二组, ORM表中的字段名 db匹配自动创建字段对象
-3- 对这些字段对象进行bind处理!! 属性加工 -- field_name、parent、label、source、source_attrs
-4- 取到可序列化的字段对象
-5- 到这一步才会用到序列化器类实例化时传进来的instance参数!!

```
ret = {}
for 字段对象 in 可序列化的字段对象:
    attribute = 字段对象.get_attribute(instance)
    ret[field.field_name] = 字段对象.to_representation(attribute)
简单来说, 就是 (复杂点来说,就跟source_attrs有关啦
    str(instance.name)
    str(instance.gender)
    str(instance.get_gender_display())
    str(instance.depart.title)
    get_xxx(instance)
```

■ 多条数据序列化
   简单来说就是for循环多条数据, 每条数据经历单条数据的序列化后的步骤后的结果都放到同一个列表中!!

视图

■ APIView
- path('api/demo/', demo.DemoView.as_view()), -- 一个视图两个方法 GET POST
- path('api/demo/<int:pk>/', demo.DemoDetailView.as_view()) -- 一个视图三个方法 GET PUT DELETE

■ ViewSetMixin + APIView == ViewSet
- ViewSetMixin重写了as_view方法! (原来APIView是要调用Django的as_view的,现在它调用ViewSetMixin的as_view
  它做了啥? eg: self.get = self.list 往下程序运行APIView里的dispatch方法时,反射执行的视图函数就变啦!!

■ ViewSetMixin + GenericAPIView == GenericViewSet
- GenericAPIView就是脱了裤子放屁,将我们在APIView里的代码规整了下.提供给五个视图扩展类用!

■ 五个视图扩展类 + GenericViewSet == ModelViewSet
- ListModelMixin -- list
- CreateModelMixin -- create
- RetrieveModelMixin -- retrieve
- UpdateModelMixin -- update
- DestroyModelMixin -- destory

```
urlpatterns = [
    path('api/user/', demo.UserView.as_view({"get":"list","post":"create"})),
    path('api/user/<int:pk>/', demo.UserView.as_view({"get":"retrieve","put":"update","delete":"destory"}))
]
```

一定要注意: 写了对应关系就得继承ViewSetMixin. 橙色的两个地方等同..

路由

■ 每次都写这些路由,太麻烦了.我们可以自动生成路由! [查多条/查一条/增/删/改]

```
from rest_framework.routers import SimpleRouter
router = SimpleRouter()
router.register('api/user', views.UserView, basename='user')
urlpatterns = [] + router.urls
```

■ 五个接口已经不能满足我了,我要在UserView类里再写几个接口!

```
from rest_framework.routers import SimpleRouter
router = SimpleRouter()
router.register('api/user', views.UserView, basename='user')
urlpatterns = [
    # ★ 这两个格外的路由这样写,跟SimpleRouter没有半毛钱关系!!就是普通的路由.
    # >>> 你可以实验,这样写+action,相当于有两个路由匹配会执行test1 一个是'test1/' 一个是'api/user/test1/'
    # >>> path('test1/', views.UserView.as_view({'get': 'test1'})),
    # - 路由匹配,且是GET请求方式,那么就执行TestView类中的test1方法!!
    path('api/user/test1/', views.UserView.as_view({'get': 'test1'})),
    # - 路由匹配,且是PUT请求方式,那么就执行TestView类中的test方法!!
    path('api/user/<int:pk>/test2/', views.UserView.as_view({'put': 'test2'})),
] + router.urls
```

★ 我们还可以不用自己写这个路由,在login方法上写action装饰器,让它帮我们生成!! >> action往往是与SimpleRouter搭配使用的!!

```
from rest_framework.routers import SimpleRouter
router = SimpleRouter()
router.register('api/user', views.UserView, basename='user')
urlpatterns = [] + router.urls

from rest_framework.decorators import action
class UserView(ModelViewSet):
    @action(methods=['GET'], detail=False, url_path='test1', url_name='test1')  # -- 'api/user/test1/'
    def test1(self,request):
        pass

    @action(methods=['PUT'], detail=True, url_path='test2', url_name='test2')  # -- 'api/user/<int:pk>/test2/'
    def test2(self,request,pk):
        pass
```

其余的查看相关对应博客吧, 有空再总结.