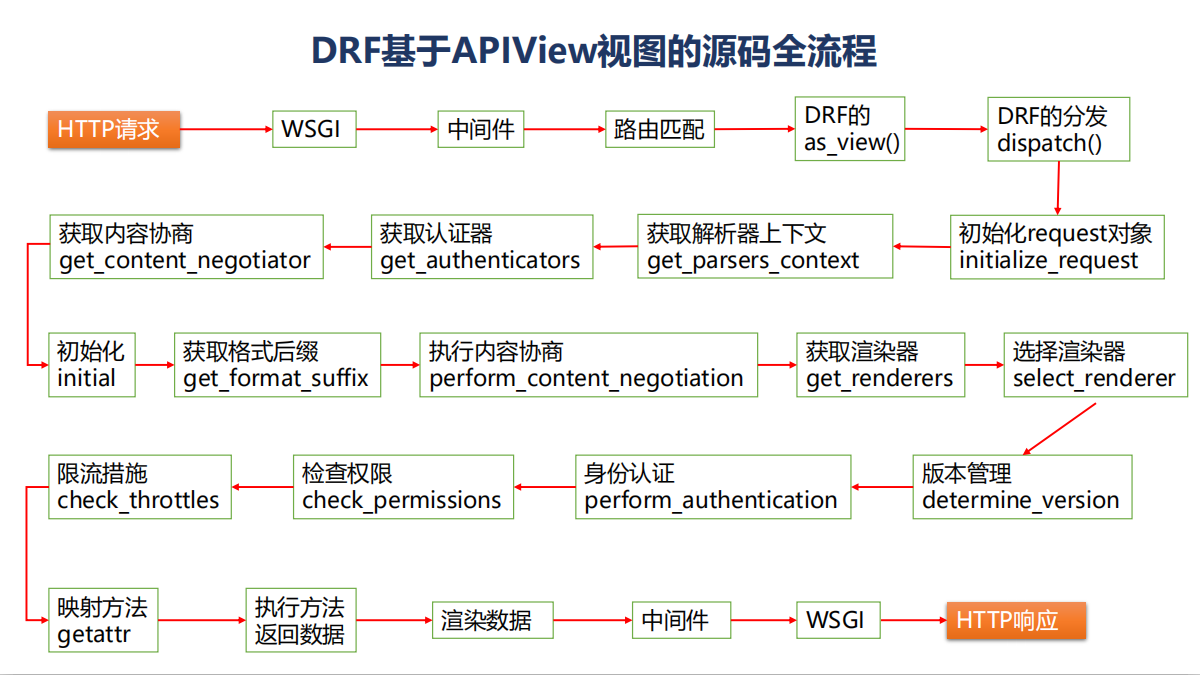


APIView



GenericAPIView

**get\_queryset(self)**

返回list视图中使用的查询集，该查询集还用作detail视图中的查找基础。

如：queryset = Post.objects.all()

**get\_object(self)**

返回用于detail视图的对象实例。默认使用 lookup\_field 参数过滤基本的查询集。 该方法可以被重写以提供更复杂的行为，例如基于多个 URL 参数的对象查找。

如：

filter\_kwargs = {self.lookup\_field: self.kwargs[lookup\_url\_kwarg]}

# 默认是{“pk”:”1”}

# 然后在根据过滤的来找，找不到就报错404

obj = get\_object\_or\_404(queryset, \*\*filter\_kwargs)

**filter\_queryset(self, queryset)**

给定一个queryset，使用任何过滤器后端进行过滤，返回一个新的queryset。

在过滤

**get\_serializer\_class(self)**

选择你想要使用的序列化类。默认返回 serializer\_class 属性的值。

可以被重写以提供动态的行为，例如对于读取和写入操作使用不同的序列化器或者为不同类 型的用户提供不同的序列化器。

get\_serializer\_class(self)本身没做啥，只是返回serializer\_class

这是一个钩子，通过这个可以指定返回自己需要的序列化类，如：

def get\_serializer\_class(self):

if self.request.user.is\_staff:

return FullAccountSerializer

return BasicAccountSerializer

**保存与删除操作中提供的钩子**:

以下方法由mixin类提供，并提供对象保存或删除行为的简单重写。

perform\_create(self, serializer) - 在保存新对象实例时 由 CreateModelMixin 调用。

perform\_update(self, serializer) - 在保存现有对象实例时 由 UpdateModelMixin 调用。

perform\_destroy(self, instance) - 在删除对象实例时由 DestroyModelMixin 调

用。

**三、Mixins**

Mixin 类用于提供视图的基本操作行为。注意mixin类提供动作方法，而不是直接定义处理程序

方法，例如 .get() 和 .post() ， 这允许更灵活的自定义。

Mixin 类可以从 rest\_framework.mixins 导入。

**ListModelMixin**

提供一个 .list(request, \*args, \*\*kwargs) 方法，返回查询结果的列表。

如果查询集被填充了数据，则返回 200 OK 响应，将查询集的序列化表示作为响应的主体。

相应数据可以任意分页。

**CreateModelMixin**

提供 .create(request, \*args, \*\*kwargs) 方法，实现创建和保存一个新model实例的功能。如果创建了一个对象，这将返回一个 201 Created 响应，将该对象的序列化表示作为响应的主体。如果序列化的表示中包含名为 url 的键，则响应的 Location 头将填充该值。

如果为创建对象提供的请求数据无效，将返回 400 Bad Request ，其中错误详细信息作为响 应的正文。

**RetrieveModelMixin**

提供一个 .retrieve(request, \*args, \*\*kwargs) 方法，返回响应中现有模型的实例。

如果可以检索对象，则返回 200 OK 响应，将该对象的序列化表示作为响应的主体。否则将

返回 404 Not Found 。

**UpdateModelMixin**

提供 .update(request, \*args, \*\*kwargs) 方法，实现更新和保存现有模型实例的功

能。 同时还提供了一个 .partial\_update(request, \*args, \*\*kwargs) 方法，这个方法

和 update 方法类似，但更新的所有字段都是可选的。这允许支持 HTTP PATCH 请求。

如果一个对象被更新，这将返回一个 200 OK 响应，并将对象的序列化表示作为响应的主 体。如果为更新对象提供的请求数据无效，将返回一个 400 Bad Request 响应，错误详细信息作为响应的正文。

**DestroyModelMixin**

提供一个 .destroy(request, \*args, \*\*kwargs) 方法，实现删除现有模型实例的功能。

如果成功删除对象，则返回 204 No Content 响应，否则返回 404 Not Found 。

DRF对mixin类的设计是让它们可以尽量的组合使用，不是一次只能继承一个mixin，可以同时继承多个mixin。

**注意：**

**1.**对于**CreateModelMixin，DestroyModelMixin，UpdateModelMixin**这三个可以对数据造成影响的，在他们对应的类都留下了一个钩子，用于对数据进行额外的操作

**2.**Create，Destroy，Update，List, Retrieve的叫法，分别对应Django的post,

delete,put,get,get方法，主要是为了区分Django原有的

class CreateAPIView(mixins.CreateModelMixin,

GenericAPIView):

def post()

# 就是调用mixins.CreateModelMixin里面的create()方法

return self.create()

------------------------------------------------------------------------------

class ListAPIView(mixins.ListModelMixin,

GenericAPIView):

def get()

# 就是调用mixins.ListModelMixin里面的list()方法

return self.list()

class RetrieveAPIView(mixins.RetrieveModelMixin,

GenericAPIView):

def get()

# 就是调用mixins.RetrieveModelMixin里面的retrieve()方法

return self.retrieve()

------------------------------------------------------------------------------ class DestroyAPIView(mixins.DestroyModelMixin,

GenericAPIView):

def delete()

# 就是调用mixins.DestroyModelMixin里面的destory()方法

return self.destory()

------------------------------------------------------------------------------ class UpdateAPIView(mixins.UpdateModelMixin,

GenericAPIView):

# 就是调用mixins.UpdateModelMixin里面的update()方法

def put(self, request, \*args, \*\*kwargs):

return self.update(request, \*args, \*\*kwargs)

# 就是调用mixins.UpdateModelMixin里面的partial\_update()方法

def patch(self, request, \*args, \*\*kwargs):

return self.partial\_update(request, \*args, \*\*kwargs)

其他组合类型同理也是一样的

ListCreateAPIView，RetrieveUpdateAPIView，RetrieveDestroyAPIView，RetrieveUpdateDestroyAPIView

**就是基于这上面的组合**

只继承GenericAPIView这个比较基础的

比如要通过get访问某条记录时就要写get()方法如：

ClassDetailPost(generics.GenericAPIView):

queryset = Post.objects.all()  
serializer\_class = PostSerializer

def get(self, request, \*args, \*\*kwargs):  
 instance = self.get\_object()  
 erializer = self.get\_serializer(instance)  
 return Response(serializer.data)

实际上就是重写RetrieveMixin里面的内容。

往上一步如果继承了RetrieveAPIView,只需要返回retrieve()

ClassDetailPost(mixins.RetrieveAPIView,generics.GenericAPIView):

queryset = Post.objects.all()  
serializer\_class = PostSerializer

def get(self, request, \*args, \*\*kwargs):  
 return self.retrieve(request, \*args, \*\*kwargs)

往上一步如果继承了RetrieveUpdateDestroyAPIView,只需写查询集和序列化类。

ClassDetailPost(mixins.RetrieveUpdateDestroyAPIView,generics.GenericAPIView):

queryset = Post.objects.all()  
serializer\_class = PostSerializer

提供读写删除单个模型实例。提供 get , put , patch 和 delete 方法

**自定义mixin对于多字段查询：**

# 自定义多字段查询(views里面)  
class MultipleFieldLookupMixin(object):  
 # 重写部分GenericAPIView里面的get\_object方法  
 def get\_object(self):  
 queryset = self.get\_queryset()  
 queryset = self.filter\_queryset(queryset)  
 filter = {}  
 for field in self.**lookup\_fields**:  
 if self.kwargs[field]:  
 filter[field] = self.kwargs[field]  
 obj = get\_object\_or\_404(queryset, \*\*filter)  
 self.check\_object\_permissions(self.request, obj)  
  
 return obj

# 针对改Mixin写一个视图

class RetrievePostView(MultipleFieldLookupMixin, generics.RetrieveAPIView):  
 queryset = Post.objects.all()  
 serializer\_class = PostSerializer  
  **lookup\_fields = ('pk', 'title'）**

# lookup\_fields为get\_object()提供重要参数

# url

path('posts/**<int:pk>/<str:title>/**', views.RetrievePostView().as\_view())

**ViewSet**

ViewSet 只是**一种基于类的视图，它不提供任何方法处理程序**（如 .get() 或 .post() ）而是提供诸如 .list() 和 .create() 之类的操作。

ViewSet 继承自 views.APIView 。你可以使用任何父类属性， 如 permission\_classes , authentication\_classes 以便控制视图集上的 API 策略。

ViewSet 类不提供任何操作的实现。为了使用 ViewSet 类，你需要重写该类并显式地定义动作实现。

ViewSet 是比前面的通用类视图更深入的封装，简化了更多的代码。它并不高大上，也没有提高性能，用于不用，取决于你的需求，既不重点推荐也不强制使用。

**ModelViewSet**

ModelViewSet 又继承了 GenericAPIView ，但实现了基本的HTTP请求方法。它提 供 .list() , .retrieve() , .create() , .update() , .partial\_update() 和 .destroy() 操作。这是我们真正使用的类。

使用 ViewSet 类有两个主要优点。 ModelViewSet 又继承了 GenericAPIView ，但实现了基本的HTTP请求方法。它提

供 .list() , .retrieve() , .create() , .update() , .partial\_update()

和 .destroy() 操作。这是我们真正使用的类

1.重复的逻辑可以组合成一个类。在上面的例子中，我们只需要指定一次 queryset ，它将 在多个视图中使用。

2.通过使用 routers, 不再需要自己处理URLconf。

from myapp.views import UserViewSet

from rest\_framework.routers import DefaultRouter

router = DefaultRouter()

router.register(r'users', UserViewSet, basename='user')

urlpatterns = router.urls最后一句或者改为

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

继承viewsets.ViewSet我们要写

class UserViewSet(viewsets.ViewSet):

def list(self, request):

pass

def create(self, request):

pass

def retrieve(self, request, pk=None):

pass

def update(self, request, pk=None):

pass

def partial\_update(self, request, pk=None):

pass

def destroy(self, request, pk=None):

pass

**为路由增加额外的方法**

**继承了viewsets.ModelViewSet类通常可以直接用@action来装饰表示添加额外的视图**

class UserViewSet(viewsets.ModelViewSet):

queryset = User.objects.all()

serializer\_class = UserSerializer

@action(detail=True, methods=['post'])

def set\_password(self, request, pk=None):  **# POST /users/<int:pk>/set\_password/**

user = self.get\_object()

serializer = PasswordSerializer(data=request.data)

if serializer.is\_valid():

user.set\_password(serializer.data['password'])

user.save()

return Response({'status': 'password set'})

else:

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**Detail 为True,即访问一个单独的detail视图（），Detail 为False,即访问一系列detail视图。**

**# POST 需要在methods里面注册**

**/users：表示访问**UserViewSet的url, **/<int:pk>**即访问一个单独的detail视图**/set\_password** 即为函数名 因此访问的url为**# POST /users/<int:pk>/set\_password/**

装饰器可以另外获取为路由视图设置的额外参数。例如...

# 认证、权限、限流

@action(detail=True, methods=['post'], **permission\_classes**= [IsAdminOrIsSelf])

def set\_password(self, request, pk=None):

...

urls users/<int:pk>/set\_password/

action装饰器将默认路由 GET 请求，但也可以通过使用 methods 参数接受其他 HTTP 方法。

@action(detail=True, **methods**=['post', 'delete'])

def unset\_password(self, request, pk=None):

...

urls users/<int:pk>/unset\_password/

**解析器：**