基于类的视图（CBV）

视图是可调用的，它接收请求并返回响应。这可能不仅仅是一个函数，Django提供了一些可用作视图的类的示例。这些允许您通过利用继承和mixin来构建视图并重用代码。

基于类的视图（Class-based views）提供了另一种将视图实现为Python对象而不是函数的方法。它们不替换基于函数的视图，但与基于函数的视图相比具有一定的差异和优势：

- 提高了代码的复用性，可以使用面向对象的技术，比如Mixin（多继承）
- 可以用不同的函数针对不同的HTTP方法处理，而不是通过很多if判断，提高代码可读性

FBV 实现

```python
# 用户注册
def register(request):
    if request.method == 'POST':
        # 注册用户逻辑
        ...
        # 转到登录界面
        return redirect(reverse("app:login"))
    else:
        # 显示注册页面
        return render(request,'app/register.html')
```

内建的基于类的视图的层次结构：

- 基本视图：view 、TemplateView、RedirectView
- 通用显示视图：DetailView、ListView
- 通用编辑视图：FormView、CreateView、 UpdateView、DeleteView
- 通用日期视图： ArchiveIndexView、YearArchiveView、MonthArchiveView、 WeekArchiveView

  DayArchiveView、TodayArchiveView、DateDetailView
- 基于类的视图mixins
  - 简单的mixins：ContextMixin、TemplateResponseMixin
  - 单个对象mixins：SingleObjectMixin、SingleObjectTemplateResponseMixin
  - 多个对象混合：MultipleObjectMixin、MultipleObjectTemplateResponseMixin

一、类视图的基本使用

所有类视图都继承自Django提供的父类View，可以使用from django.views import View或from django.views.generic import View来导入父类View。

```
#views.py
```

```
from django.urls import reverse
from django.views import View
from django.http import HttpResponse
from django.shortcuts import render, redirect

class Register(View):
    # 处理GET请求
    def get(self,request, *args, **kwargs):
        return render(request,"App/register.html")

    # 处理POST请求
    def post(self, request, *args, **kwargs):
        # 注册业务处理
        ...
        return redirect(reverse("App:login"))
```

路由注册：

```
urlpatterns = [
    # 函数注册
    path("register/",views.register,name='register')
    # 类视图注册
    path(r'register/',views.RegisterView.as_view(),name='register')
]
```

二、基本视图

1.根视图View类

提供适合各种应用程序的基本视图类。所有视图都继承自`View`该类，该类处理将视图链接到URL，HTTP方法调度和其他简单功能。

View类核心代码在as_view和dispatch方法中，其中as_view是类方法（@classonlymethod），只能通过类调用，不能通过对象调用，它是类视图的入口点。注意这里调用的时候是通过类名.as_view()调用的。

其中，as_view方法主要执行逻辑：

```
    @classonlymethod
    def as_view(cls, **initkwargs):
        """Main entry point for a request-response process."""
        # 参数检查
        for key in initkwargs:
            if key in cls.http_method_names: # 参数名不能是指定http的方
法名
                raise TypeError("You tried to pass in the %s method
name as a "
                                "keyword argument to %s(). Don't do
that."
                                % (key, cls.__name__))
            if not hasattr(cls, key): # 参数名必须是类已有属性
                raise TypeError("%s() received an invalid keyword %r.
as_view "
                                "only accepts arguments that are
already "
                                "attributes of the class." %
(cls.__name__, key))
        # 视图处理函数
        def view(request, *args, **kwargs):
            self = cls(**initkwargs) # 实例化当前类的对象
            if hasattr(self, 'get') and not hasattr(self, 'head'):
```

```
                self.head = self.get
            self.setup(request, *args, **kwargs)
            if not hasattr(self, 'request'):
                raise AttributeError(
                    "%s instance has no 'request' attribute. Did you
override "
                    "setup() and forget to call super()?" %
cls.__name__
                )
            # 方法派发
            return self.dispatch(request, *args, **kwargs)
        view.view_class = cls
        view.view_initkwargs = initkwargs

        # take name and docstring from class
        update_wrapper(view, cls, updated=())

        # and possible attributes set by decorators
        # like csrf_exempt from dispatch
        update_wrapper(view, cls.dispatch, assigned=())
        return view  # 返回视图函数
```

整个as_view方法是一个装饰器方法，它返回内部函数view，所以as_view()执行其实就是内部函数view执行。内部函数view主要逻辑就是：
as_view()=>view()=>dispatch()=>相应的http方法

- 实例化本类对象
- 接收请求对象和参数（setup）
- 调用dispatch方法进行派发

调用as_view方法可以传递参数，但要注意：

- 不能使用请求方法的名字作为参数的名字
- 只能接受视图类已经存在的属性对应的参数

dispatch方法是实例方法，它的主要代码：

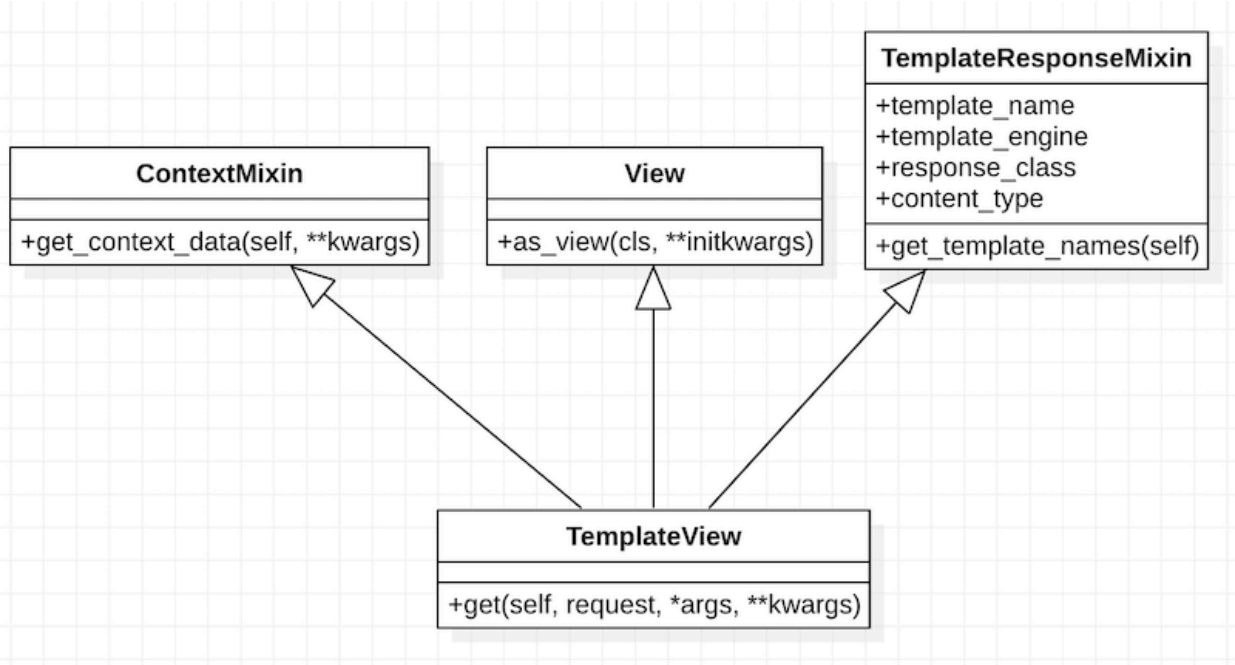
```
def dispatch(self, request, *args, **kwargs):
        #检查请求方法是不是在http_method_names中包含
        #http_method_names包括八种方法：['get', 'post', 'put',
'patch', 'delete', 'head',
        #'options', 'trace']
        if request.method.lower() in self.http_method_names:
            handler = getattr(self, request.method.lower(),
self.http_method_not_allowed)
        else: #不在调用http_method_not_allowed报错
            handler = self.http_method_not_allowed
        #调用和请求方法同名的实例方法处理用户请求，实例方法需要用户自己定义
        return handler(request, *args, **kwargs)
```

dispatch主要完成http请求方法的派发，调用视图类对应实例方法处理用户请求，所有用户需要定义和http请求方法同名的实例方法完成功能，所以一般CBV的模块写法是：

```
from django.views import View

class IndexView(View):
    def get(self,request):
        return HttpResponse("get")
    def post(self,request):
        return HttpResponse("post")
    def put(self,request):
        return HttpResponse("put")
    def delete(self,request):
        return HttpResponse("delete")
```

2.TemplateView

TemplateView可以根据上下文渲染指定模板，返回响应对象。它继承了ContentMixin、View、TemplateResponseMixin。



- ContentMixin用于获取渲染模板的变量。你可以重写get_context_data方法返回模板渲染的参数
- TemplateResponseMixin 用于渲染模板
  - template_name模板文名(必须设置)
  - template_engine模板引擎（有默认值）
  - response_class模板渲染类，默认是TemplateResponse
  - content_type内容类型，默认是text/html
  - get_template_names你可以重写这个方法返回模板名称

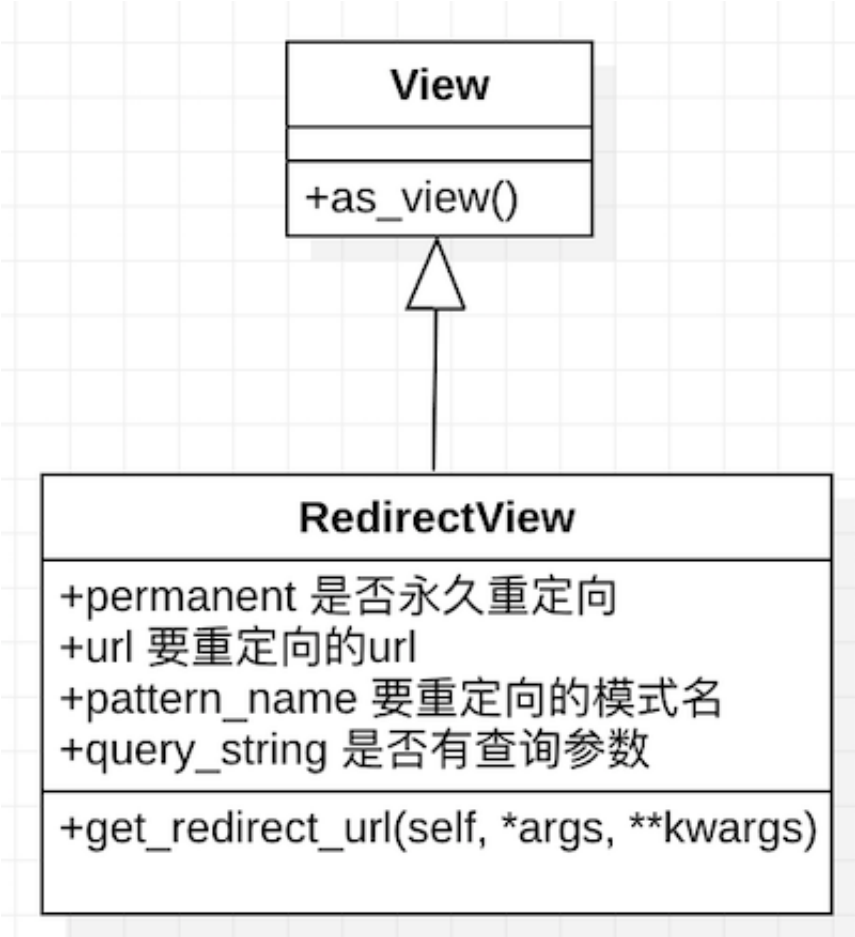
```
#路由
urlpatterns = [

url(r'^hello/(\w+)/$',views.HelloView.as_view(template_name='hello.ht
ml'),name='hello'),
]

#views.py
class HelloView(TemplateView):

    def get(self, request, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        context['name'] = args[0]
        return self.render_to_response(context)
```

- 注意as_view方法参数只能是template_name、template_engine、

response_class、content_type

3.RedirectView

重定向的指定url



get_redirect_url用于构造重定向的目标URL，可以重写。

默认实现url用作起始字符串，并%使用URL中捕获的命名组在该字符串中执行命名参数的扩展。

如果url未设置，则get_redirect_url()尝试反转 pattern_name使用URL中捕获的内容（使用已命名和未命名的组）。

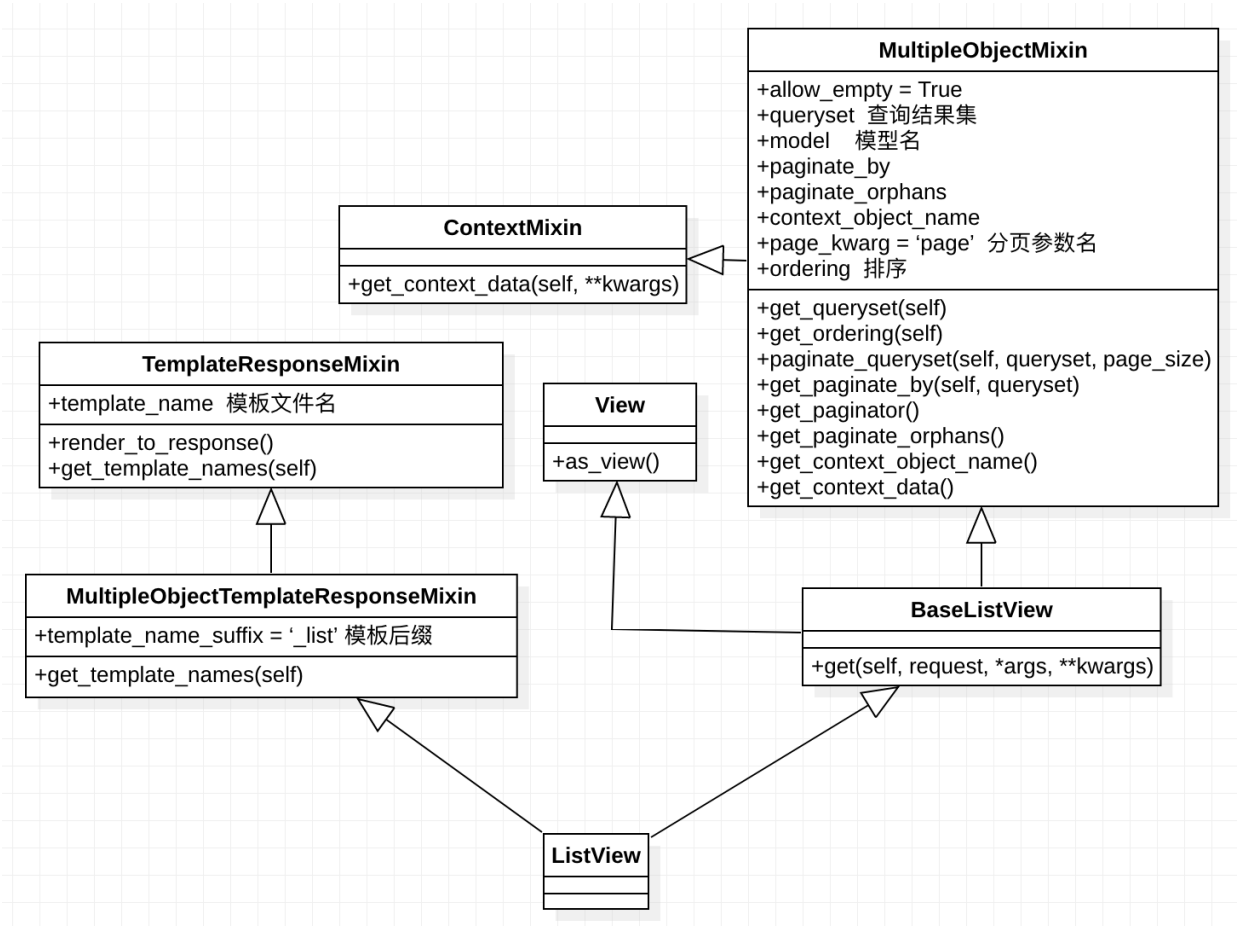
如果请求query_string，它还会将查询字符串附加到生成的URL。子类可以实现他们希望的任何行为，只要该方法返回可重定向的URL字符串即可。

三、通用显示视图

本类视图主要用户数据展示，包括ListView显示对象列表信息和DetailView显示对象详细信息

3.1 ListView

显示对象列表页面



- MultipleObjectTemplateResponseMixin
  - 提供了模板文件名
  - 如果没有指定模板文件名，则默认模板文件名规则是：应用名/模型名_list.html
- MultipleObjectMixin

  核心类，提供了渲染模板所有需要的模型或查询结果集（不一定是QuerySet，可以是对象列表），分页。
  - queryset属性用于渲染模板所需对象列表，也可以重写get_queryset方法获取
  - model，如果没指定queryset，则根据指定model获取对象列表
  - context_object_name模板中对象列表的名称，如果不指定，则根据model获取对象列表名称：model_list
  - paginate_by指定分页每页的记录个数，默认是None，不分页

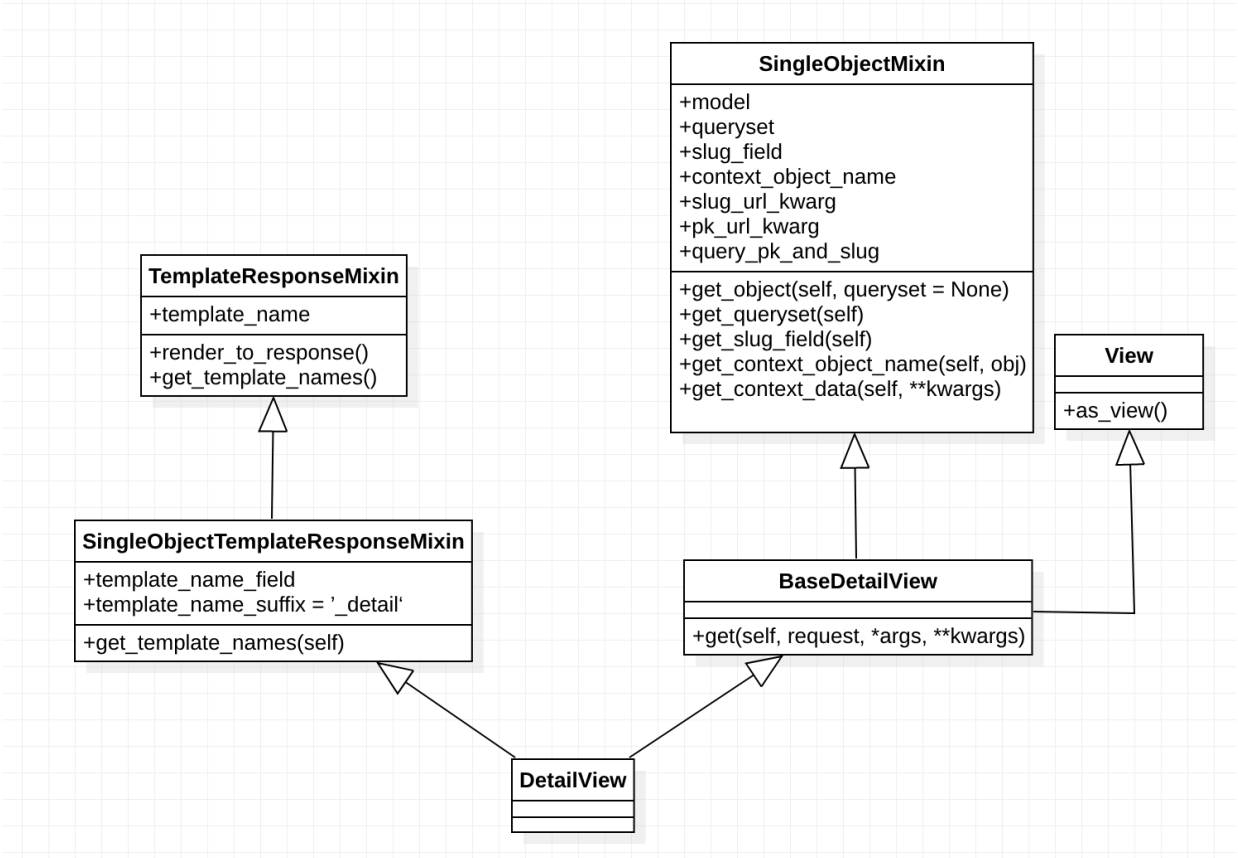
- page_kwarg指定分页请求路径中命名分组名或get传参中键的名称，默认是page
- paginate_orphans是指分页最后一页如果记录不满一页的处理方式，默认是0，和前一页合并显示，如果不为0，则单独显示一页
- 分页具体实现：

```
 def get_context_data(self, **kwargs):
        """
        Get the context for this view.
        """
        queryset = kwargs.pop('object_list',
self.object_list)
        page_size = self.get_paginate_by(queryset)
        context_object_name =
self.get_context_object_name(queryset)
        if page_size:
            paginator, page, queryset, is_paginated =
self.paginate_queryset(queryset, page_size)
            context = {
                'paginator': paginator,  #在模板中可以使用分页
器
                'page_obj': page,        #分页对象
                'is_paginated': is_paginated,
                'object_list': queryset   #当前页数据
            }
        else:
            context = {
                'paginator': None,
                'page_obj': None,
                'is_paginated': False,
                'object_list': queryset
            }
        if context_object_name is not None: #如果
context_object_name不为空
            context[context_object_name] = queryset
        context.update(kwargs)
        return super(MultipleObjectMixin,
self).get_context_data(**context)
```

- BaseListView默认实现get请求

3.2 DetailView

显示对象的详细信息



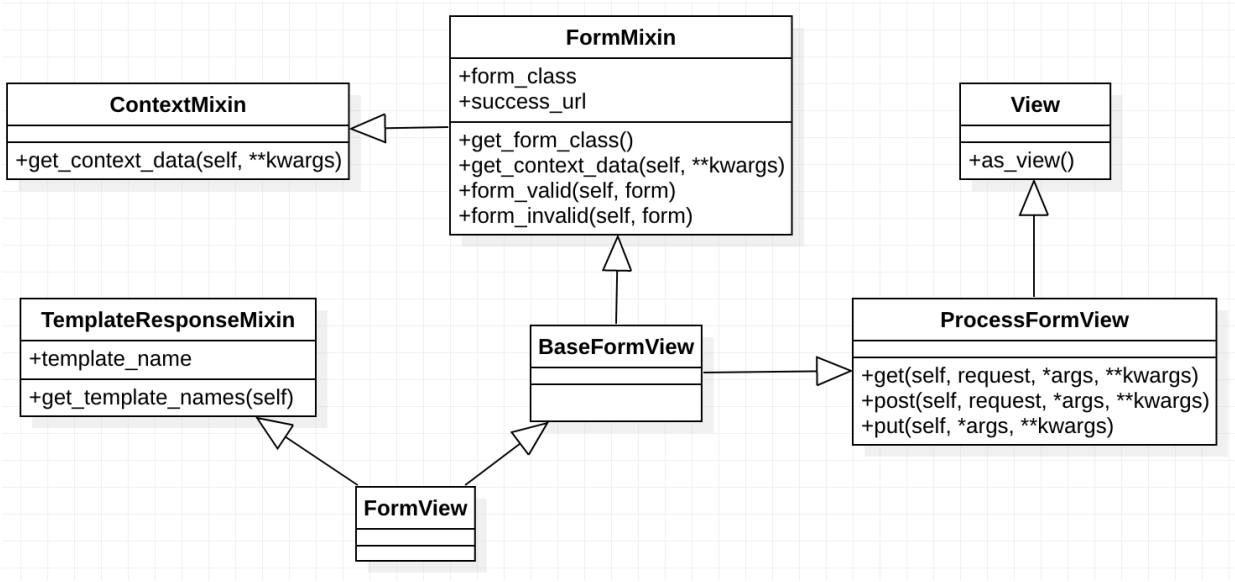
- SingleObjectMixin
  - pk_url_kwarg 默认值pk，从请求路径中获取主键的值，请求路径中参数必须是命名组，组名必须和pk_url_kwarg的值一样
  - slug_url_kwarg默认值是slug，从请求路径中获取查询参数sug的值，请求路径中参数必须是命名组，组名必须和slug_url_kwarg的值一样，如果参数中有pk_url_kwarg的值，则slug_url_kwarg不起作用
  - slug_field查询字段的名称
  - context_object_name模板中引用对象的名称，默认模板中对象名称是object

四、通用编辑视图

本类视图主要完成对象的增删改。包括FormView、CreateView、 UpdateView、DeleteView

4.1 FormView

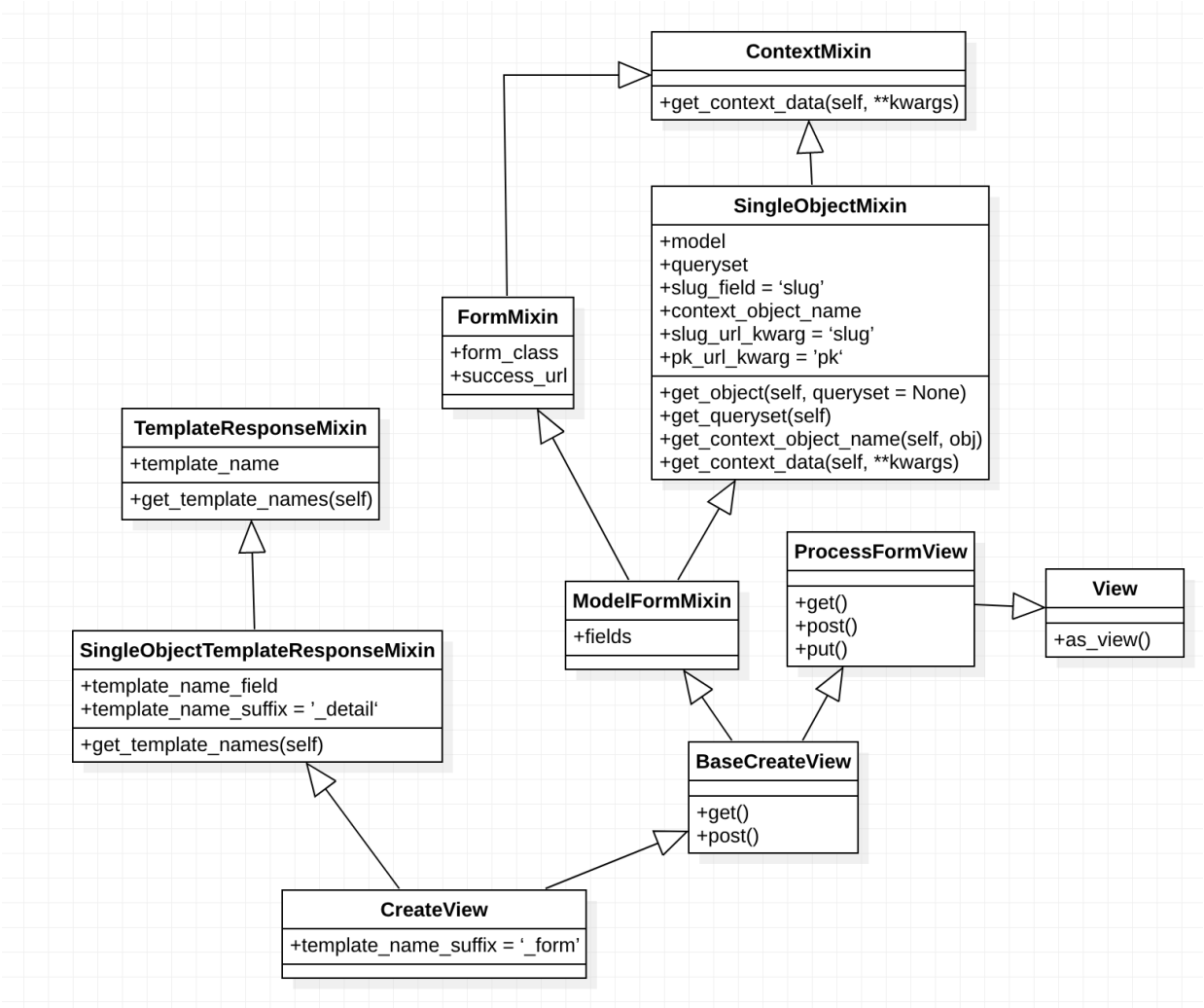
显示表单的视图。出错时，重新显示带有验证错误的表单; 成功时，重定向到新的URL。



- FormMixin
  - form_class 表单类名
  - success_url表单验证成功后调整的url
  - form_valid() 验证数据成功后的处理
  - form_invalid() 验证不成功的处理
- ProcessFormView
  - get渲染表单
  - post表单提交
  - put创建或修改对象

4.2 CreateView

显示用于创建对象的表单的视图，使用验证错误（如果有）重新显示表单并保存对象。



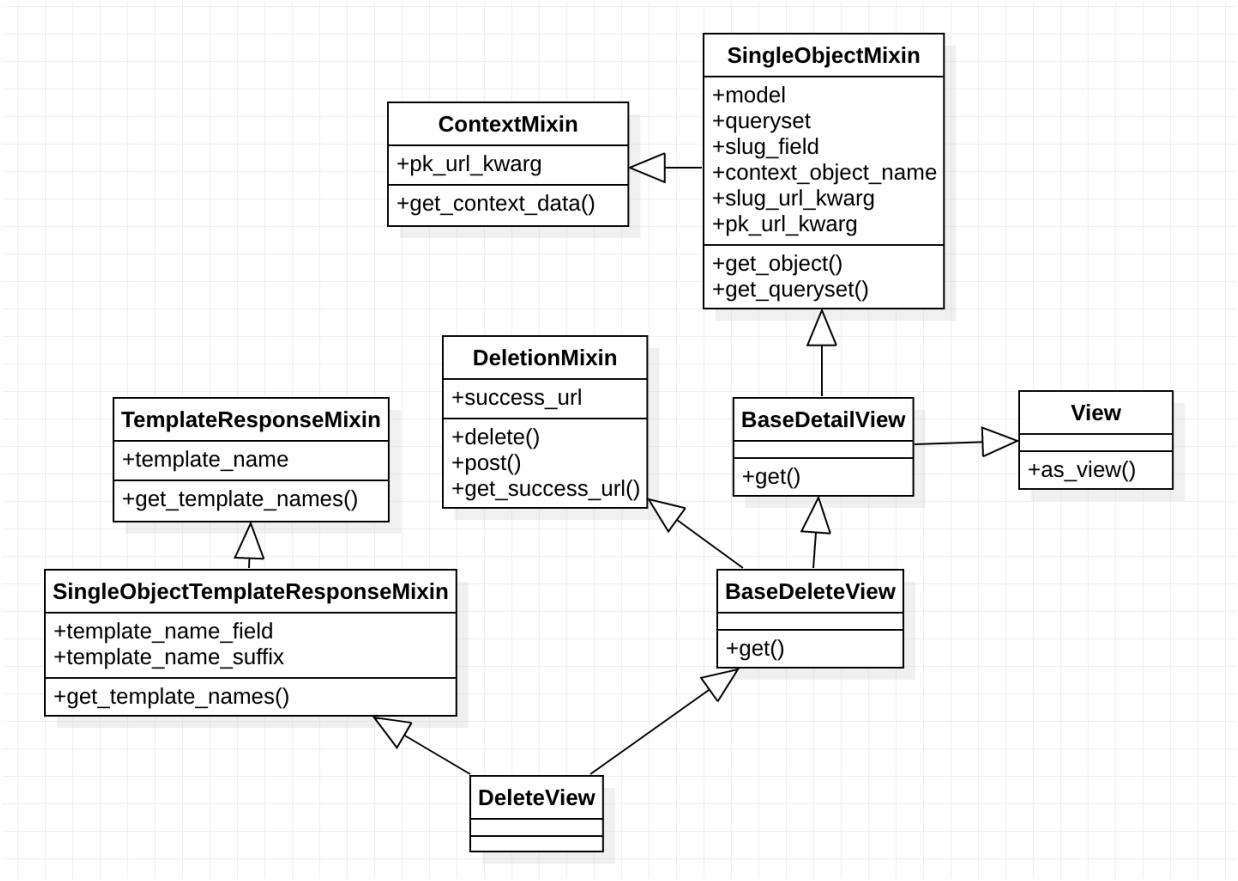
- 重要属性：
  - template_name 模板文件名
  - fields指定的字段列表
  - model关联模型名
  - form_class表单类，如果没有设置会默认是模型名

4.3 UpdateView

UpdateView的用法和CreateView基本一样

4.4 DeleteView

删除指定对象的视图



五、类视图使用装饰器

为类视图添加装饰器，可以使用两种方法。

为了理解方便，我们先来定义一个**为函数视图准备的装饰器**（在设计装饰器时基本都以函数视图作为考虑的被装饰对象），及一个要被装饰的类视图。

```
def my_decorator(func):
    def wrapper(request, *args, **kwargs):
        print('自定义装饰器被调用了')
        print('请求路径%s' % request.path)
        return func(request, *args, **kwargs)
    return wrapper

class DemoView(View):
    def get(self, request):
        print('get方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post方法')
        return HttpResponse('ok')
```

在类视图中使用为函数视图准备的装饰器时，不能直接添加装饰器，需要使用**method_decorator**将其转换为适用于类视图方法的装饰器。

method_decorator装饰器使用name参数指明被装饰的方法

```
# 为全部请求方法添加装饰器
@method_decorator(my_decorator, name='dispatch')
class DemoView(View):
    def get(self, request):
        print('get方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post方法')
        return HttpResponse('ok')


# 为特定请求方法添加装饰器
@method_decorator(my_decorator, name='get')
class DemoView(View):
    def get(self, request):
        print('get方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post方法')
        return HttpResponse('ok')
```

如果需要为类视图的多个方法添加装饰器，但又不是所有的方法（为所有方法添加装饰器参考上面例子），可以直接在需要添加装饰器的方法上使用method_decorator，如下所示

```
from django.utils.decorators import method_decorator

# 为特定请求方法添加装饰器
class DemoView(View):

    @method_decorator(my_decorator)  # 为get方法添加了装饰器
    def get(self, request):
        print('get方法')
        return HttpResponse('ok')
```

```
    @method_decorator(my_decorator)  # 为post方法添加了装饰器
    def post(self, request):
        print('post方法')
        return HttpResponse('ok')

    def put(self, request):  # 没有为put方法添加装饰器
        print('put方法')
        return HttpResponse('ok')
```