Django-CBV

> **CBV（class base views）** 就是在视图里使用类处理请求
>
> 之前的代码中，我们的视图函数都是通过函数来进行`request`的响应以及`response`的返回，并且通常我们需要判断的请求方式`get`或是`post`都需要我们在代码中通过`if`进行条件判断，这样的视图功能编写就叫做`FBV`
>
> 但现在在`django`中还提供了一种方式叫做`CBV`，在类中编写视图功能， 并且将传统的`get`、`post`判断设置为了类中函数，这样当用户发起不同的请求，会自动进入到对应的类中函数上，像是下面这样

```
from django.views import View
class ArticleView(View):
    def get(self,request):
        raise Http404
    def post(self,request):
        if request.is_ajax():
            id_ = request.POST.get('id_')
            result = models.Article.objects.get(id=id_).content
            data = result.replace('\r\n','<br>')
            return HttpResponse(json.dumps(data,ensure_ascii=False) )
        raise Http404
```

> 通过将请求类型定义为函数，可以更加方便进行请求方式判断
>
> 用户访问时，会经由`View`基类中的`as_view -> dispatch`进行判断，通过请求类型分发到不同对应请求的函数名下；也就是通过`get`方式访问，那么对应会调用到名为`get`的函数
>
> 此外，类中函数必须为小写，

- 对应路由此时设置为，需要使用试图类的`as_view`函数进行实例化

```
#url.py
path('article/',ajaxviews.ArticleView.as_view())
```

> 通过类视图可以方便我们进行请求条件的判断
>
> 并且可以在进行接口开发时，实现同一资源路由在使用不同请求访问时的功能解耦和
>
> 意思就是不用再把所有的功能都堆到一个视图函数里啦。多方便!
>
> 并且，在`Django-Restframework`框架中，也将频繁使用`CBV`形式进行视图编写

类视图装饰器

> 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
>
> `from django.utils.decorators import method_decorator`

- 全部装饰

```
from django.views import View
from django.utils.decorators import method_decorator
```

```
def my_decorator(func):
    def nei(request): # dispatch函数有参数request
        print('这是装饰器在调用')
        return func(request)
    return nei
@method_decorator(my_decorator, name='dispatch')
# 为全部请求方法添加装饰器
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')
```

- 为部分装饰，只需要通过`method_decorator`方法的`name`参数选择装饰的函数名即可

```
@method_decorator(my_decorator, name='post')
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')
```

- 为特定的多个类视图函数进行装饰，只需要在每个函数上使用`method_decorator`装饰器即可

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

类视图`csrf_token`装饰

> 当类视图需要允许跨站提交数据时，使用`csrf_exempt`装饰器装饰函数可以被跨域访问
>
> 但是使用上面的方法进行`csrf_exempt`是不行的，需要在类视图基类的`dispatch`函数上进行装饰

```
from django.views.decorators.csrf import csrf_exempt
```

```python
#@method_decorator(csrf_exempt,name='dispatch') # 直接加载类视图上也是可以修饰的
class DemoView(View):
    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(DemoView,self).dispatch(request, *args, **kwargs)
    def get(self, request):
        print('get方法')
        return HttpResponse('ok')
    def post(self, request):
        print('post方法')
        return HttpResponse('ok')
```

> `csrf`装饰只能在类视图的`dispatch`函数上才能被生效

> 除了在类视图的`dispatch`函数上进行装饰，在路由映射处使用`csrf_exempt`函数修饰路由规则也是可以的

```python
#urls.py
from django.views.decorators.csrf import csrf_exempt
urlpatterns = [
    ...
    path('',csrf_exempt(ajaxviews.DemoView.as_view()))
]
```