

## 16-基于类的通用视图：ListView 和 DetailView

在开发网站的过程中，有一些视图函数虽然处理的对象不同，但是其大致的代码逻辑是一样的。比如一个博客和一个论坛，通常其首页都是展示一系列的文章列表或者帖子列表。对处理首页的视图函数来说，虽然其处理的对象一个是文章，另一个是帖子，但是其处理的过程是非常类似的。首先是从数据库取出文章或者帖子列表，然后将这些数据传递给模板并渲染模板。于是，Django 把这些相同的逻辑代码抽取了出来，写成了一系列的通用视图函数，即基于类的通用视图（Class Based View）。

使用类视图是 Django 推荐的做法，而且熟悉了类视图的使用方法后，能够减少视图函数的重复代码，节省开发时间。接下来就让我们把博客应用中的视图函数改成基于类的通用视图。

### 1、ListView

在我们的博客应用中，有几个视图函数是从数据库中获取文章（Post）列表数据的：

```
【blog/views.py】  
  
def index(request):  
    # ...  
  
def archives(request, year, month):  
    # ...  
  
def category(request, pk):  
    # ...
```

这些视图函数都是从数据库中获取文章（Post）列表，唯一的区别就是获取的文章列表可能不同。比如 index 获取全部文章列表，category 获取某个分类下的文章列表。

#### （1）将 index 视图函数改写为类视图

针对这种从数据库中获取**某个模型列表数据**（比如这里的 Post 列表）的视图，Django 专门提供了一个 **ListView** 类视图。下面我们通过一个例子来看 ListView 的使用方法。我们首先把 index 视图函数改造成类视图函数。

【blog/views.py】

通用视图

```
from django.views.generic import ListView
```

```
class IndexView(ListView):
    model = Post
    template_name = 'blog/index.html'
    context_object_name = 'post_list'
```

要写一个类视图，首先需要继承 Django 提供的某个类视图。至于继承哪个类视图，需要根据你的视图功能而定。比如这里 IndexView 的功能是从数据库中获取文章（Post）列表，ListView 就是从数据库中获取某个模型列表数据的，所以 IndexView 继承 ListView。

然后就是通过一些属性来指定这个视图函数需要做的事情。这里我们指定了三个属性。

- **model**：将 model 指定为 Post，告诉 Django 我要获取的模型是 Post。
- **template\_name**：指定这个视图渲染的模板。
- **context\_object\_name**：指定获取的模型列表数据保存的变量名。这个变量会被传递给模板。

如果还是有点难以理解，不妨将类视图的代码和 index 视图函数的代码对比一下：

【blog/views.py】

```
def index(request):
    post_list = Post.objects.all()
    return render(request, 'blog/index.html', context={'post_list': post_list})
```

index 视图函数首先通过 Post.objects.all() 从数据库中获取文章（Post）列表数据，并将其保存到 post\_list 变量中。而在类视图中这个过程 ListView 已经帮我们做了。我们只需告诉 ListView 去数据库获取的模型是 Post，而不是 Comment 或者其它什么模型，即指定 model = Post。将获得的模型数据列表保存到 post\_list 里，即指定 context\_object\_name = 'post\_list'。然后渲染 blog/index.html 模板文件，index 视图函数中使用 render 函数。但这个过程 ListView 已经帮我们做了，我们只需指定渲染哪个模板即可。

接下来就是要将类视图转换成函数视图。为什么需要将类视图转换成函数视图呢？

我们来看一看 blog 的 URL 配置：

【blog/urls.py】

```
app_name = 'blog'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    ...
]
```

前面已经说过每一个 URL 对应着一个视图函数，这样当用户访问这个 URL 时，Django 就知道调用哪个视图函数去处理这个请求了。在 Django 中 URL 模式的配置方式就是通过 url 函数将 URL 和视图函数绑定。比如 url(r'^\$', views.index, name='index')，它的第一个参数是 URL 模式，第二个参数是视图函数 index。对 url 函数来说，第二个参数传入的值必须是一个函数。而 IndexView 是一个类，不能直接替代 index 函数。好在将类视图转换成函数视图非常简单，只需调用类视图的 as\_view() 方法即可（至于 as\_view 方法究竟是如何将一个类转换成一个函数的目前不必关心，只需要在配置 URL 模式是调用 as\_view 方法就可以了。具体的实现我们以后会专门开辟一个专栏分析类视图的源代码，到时候就能看出 Django 使用的魔法了）。

现在在 URL 配置中把 index 视图替换成类视图 IndexView：

【blog/urls.py】

```
app_name = 'blog'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    ...
]
```

访问一下首页，可以看到首页依然显示全部文章列表，和使用视图函数 index 时效果一模一样。

## （2）将 category 视图函数改写为类视图

category 视图函数的功能也是从数据库中获取文章列表数据，不过其和 index 视图函数不同的是，它获取的是某个分类下的全部文章。因此 category 视图函数中多了一步，即首先需要根据从 URL 中捕获的分类 id 并从数据库获取分类，然后使用 filter 函数过滤出该分类下的全部文章。来看看这种情况下类视图该怎么写：

【blog/views.py】

```
class CategoryView(ListView):
    model = Post
    template_name = 'blog/index.html'
    context_object_name = 'post_list'
```

实际上是从模型当中获取数据

python2的super写法，python3中直接super()

复写基类的方法，要注意，一定会使用super来调用基类的方法

```
def get_queryset(self):
    cate = get_object_or_404(Category, pk=self.kwargs.get('pk'))
    return super(CategoryView, self).get_queryset().filter(category=cate)
```

和 `IndexView` 不同的地方是，我们覆写了父类的 `get_queryset` 方法。该方法默认获取指定模型的全部列表数据。为了获取指定分类下的文章列表数据，我们覆写该方法，改变它的默认行为。

首先是需要根据从 URL 中捕获的分类 id（也就是 `pk`）获取分类，这和 `category` 视图函数中的过程是一样的。不过注意一点的是，在类视图中，从 URL 捕获的命名组参数值保存在实例的 `kwargs` 属性（是一个字典）里，非命名组参数值保存在实例的 `args` 属性（是一个列表）里。所以我们使用了 `self.kwargs.get('pk')` 来获取从 URL 捕获的分类 id 值。然后我们调用父类的 `get_queryset` 方法获得全部文章列表，紧接着就对返回的结果调用了 `filter` 方法来筛选该分类下的全部文章并返回。

此外我们可以看到 `CategoryView` 类中指定的属性值和 `IndexView` 中是一模一样的，所以如果为了进一步节省代码，甚至可以直接继承 `IndexView`：

【blog/views.py】

```
class CategoryView(IndexView):
    def get_queryset(self):
        cate = get_object_or_404(Category, pk=self.kwargs.get('pk'))
        return super(CategoryView, self).get_queryset().filter(category=cate)
```

然后就在 URL 配置中把 `category` 视图替换成类视图 `CategoryView`：

```
blog/urls.py
```

```
app_name = 'blog'
urlpatterns = [
    ...
    url(r'^category/(?P<pk>[0-9]+)/$', views.CategoryView.as_view(), name='category'),
]
```

访问以下某个分类页面，可以看到依然显示的是该分类下的全部文章列表，和使用视图函数 `category` 时效果一模一样。

### （3）将 archives 视图函数改写成类视图

方法一样：

【blog/views.py】

```
class ArchivesView(IndexView):

    def get_queryset(self):
```

```

        return super(ArchivesView, self).get_queryset().filter(

            created_time__year=self.kwargs.get('year'),

            created_time__month=self.kwargs.get('month')

        )

```

## 2、DetailView

除了从数据库中获取模型列表的数据外，从数据库获取模型的一条记录数据也是常见的需求。比如查看某篇文章的详情，就是从数据库中获取这篇文章的记录然后渲染模板。对于这种类型的需求，Django 提供了一个 `DetailView` 类视图。下面我们就来将 `detail` 视图函数转换为等价的类视图 `PostDetailView`，代码如下：

【blog/views.py】

```
from django.views.generic import ListView, DetailView
```

# 记得在顶部导入 `DetailView`

```
class PostDetailView(DetailView):
```

# 这些属性的含义和 `ListView` 是一样的

```
model = Post
```

```
template_name = 'blog/detail.html'
```

```
context_object_name = 'post'
```

```
def get(self, request, *args, **kwargs):
```

# 覆写 `get` 方法的目的是因为每当文章被访问一次，就得将文章阅读量+1

# `get` 方法返回的是一个 `HttpResponse` 实例

# 之所以需要先调用父类的 `get` 方法，是因为只有当 `get` 方法被调用后，

# 才有 `self.object` 属性，其值为 `Post` 模型实例，即被访问的文章 `post`

```
response = super(PostDetailView, self).get(request, *args, **kwargs)
```

# 将文章阅读量+1

# 注意 `self.object` 的值就是被访问的文章 `post`

```
self.object.increase_views()
```

# 视图必须返回一个 `HttpResponse` 对象

```
return response
```

```
def get_object(self, queryset=None):
```

# 覆写 `get_object` 方法的目的是因为需要对 `post` 的 `body` 值进行渲染

```
post = super(PostDetailView, self).get_object(queryset=None)
```

```
post.body = markdown.markdown(post.body,
```

```
    extensions=[
```

```
        'markdown.extensions.extra',
```

要注意的是，复写基类的方法，该返回的必须得返回，不能少

每次请求文章详情页，都会调用 `get` 方法，因此 `increase_views` 写在 `get()` 当中比较合适

```

        'markdown.extensions.codehilite',
        'markdown.extensions.toc',
    ])
    return post

def get_context_data(self, **kwargs):
    # 覆写 get_context_data 的目的是因为除了将 post 传递给模板外 (DetailView 已经帮
    # 我们还要把评论表单、post 下的评论列表传递给模板。也就是往 context 里添加内容
    context = super(PostDetailView, self).get_context_data(**kwargs) 自动往context里加上self.object
    form = CommentForm()
    comment_list = self.object.comment_set.all()
    context.update({
        'form': form,
        'comment_list': comment_list
    })
    return context

```

PostDetailView 稍微复杂一点，主要是等价的 detail 视图函数本来就比较复杂，下面来一步步对照 detail 视图函数中的代码讲解。

首先我们为 PostDetailView 类指定了一些属性的值，这些属性的含义和 ListView 中是一样的，这里不再重复讲解。

紧接着我们覆写了 get 方法。这对应着 detail 视图函数中将 post 的阅读量 +1 的那部分代码。事实上，你可以简单地把 get 方法的调用看成是 detail 视图函数的调用。

接着我们又复写了 get\_object 方法。这对应着 detail 视图函数中根据文章的 id (也就是 pk) 获取文章，然后对文章的 post.body 进行 Markdown 渲染的代码部分。

最后我们复写了 get\_context\_data 方法。这部分对应着 detail 视图函数中生成评论表单、获取 post 下的评论列表的代码部分。这个方法返回的值是一个字典，这个字典就是模板变量字典，最终会被传递给模板。

你也许会被这么多方法搞乱，为了便于理解，你可以简单地把 get 方法看成是 detail 视图函数，至于其它的像 get\_object、get\_context\_data 都是辅助方法，这些方法最终在 get 方法中被调用，这里你没有看到被调用的原因是它们隐含在了 super(PostDetailView, self).get(request, \*args, \*\*kwargs) 即父类 get 方法的调用中。最终传递给浏览器的 HTTP 响应就是 get 方法返回的 HttpResponse 对象。

**这些方法的相同点：都执行了父类方法，然后对父类方法的返回值进行一些操作，最后返回这个修改后的返回值。**

参考官方文档：[基于类的视图概述](#)。