

[笔记][跟老齐学 Python Django 实战 第2版][3]

[笔记][跟老齐学 Python Django 实战 第2版][3]

3. 文章管理和展示

3.1 管理文章栏目

3.2 发布和显示文章

3.3 删除和修改文章

3.4 文章展示

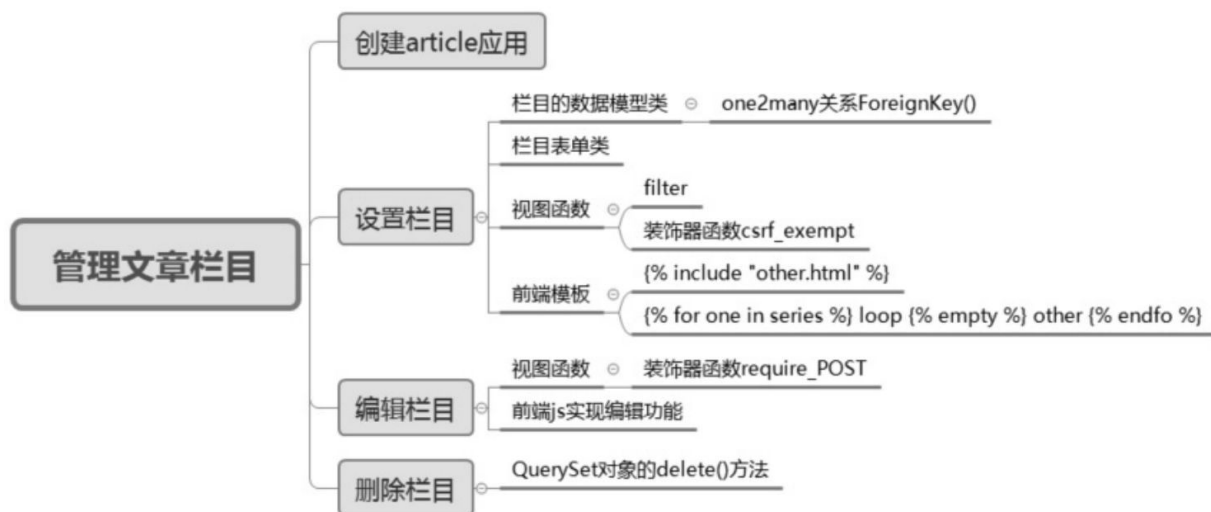
3. 文章管理和展示

网站中必须有内容，如果按照内容产生的方式来分类，就目前来讲可以分为两类，一类是“用户生成内容（UGC）”，比如“YouTube”“Twitter”；另一类是“专业生产内容（PGC/PPC）”，比如“跟老齐学itdiffer.com”。

本章要学习的项目是做“用户生成内容”的网站。这种网站有前后两部分，“前面”是让访客浏览网站上的文档，“后面”是用户管理自己的文章。

3.1 管理文章栏目

每个发布文章的用户都希望能用“**栏目**”来对自己的文章进行归类，不至于让自己的页面显得凌乱



设置栏目

在实现对文章的管理功能之前，要创建“文章管理”的应用，这是必需的，以区别前面的各种应用。

```
python manage.py startapp article
```

首先创建一个名为 `article` 的应用，并且要在 `./mysite/settings.py` 中进行配置。

```
INSTALLED_APPS = [  
    ...,  
    'article',  
]
```

然后就是我们已经熟悉的步骤了，创建数据模型、表单、视图函数、前端模板和配置URL。

团子注：这个顺序很好，记住它！
模型 – 表单 – 视图 – 路由 – 模板

1. 栏目的数据模型

对于文章栏目，这里不做多级栏目设置，只设置一级栏目（**笔者不建议让用户一层一层地单击才能找到想要的东西**），所以相对来讲，栏目的数据模型稍微简单一些。编辑 `./article/models.py` 文件，输入如下代码。

```

from django.contrib.auth.models import User
from django.db import models

class ArticleColumn(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='article_column')
    column = models.CharField(max_length=200)
    created = models.DateField(auto_now_add=True)

    def __str__(self):
        return self.column

```

在 Django 中，模型对象之间的关系可以概括为“一对一”、“一对多”和“多对多”三种关系，分别对应 `OneToOneField`、`ForeignKey`、`ManyToManyField`。

数据模型建好后就要迁移数据了，生成数据库表。

```

python manage.py makemigrations article
Migrations for 'article':
  article/migrations/0001_initial.py
    - Create model ArticleColumn

python manage.py migrate article
Operations to perform:
  Apply all migrations: article
Running migrations:
  Applying article.0001_initial... OK

```

有了数据模型，自然少不了表单类，因为要通过表单填写栏目名称，即为 `column` 字段赋值。所以，要创建 `./article/forms.py` 文件，并编写如下代码。

```

from django import forms

from .models import ArticleColumn

class ArticleColumnForm(forms.ModelForm):
    class Meta:
        model = ArticleColumn
        fields = ('column',)

```

2.简易视图函数

为了能够对即将做的东西有一个直观的感受，我们暂时不写表单类，而是写一个简单的视图函数，先看看想要达到的效果。

编辑 `./article/views.py` 文件，输入如下代码。

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render

from .models import ArticleColumn

login_required(login_url='/account/login/')
def article_column(request):
    columns = ArticleColumn.objects.filter(user=request.user)
    return render(request, 'article/column/article_column.html',
        {'columns': columns})
```

`ArticleColumn.objects.filter(user=request.user)` 等价于：

- `ArticleColumn.objects.all()`
- `filter(user=request.user)`

接下来配置 `./mysite/urls.py` 中的 URL。

```
path('article/', include(('article.urls', 'article'),
    namespace='article')),
```

再创建 `./article/urls.py` 文件，设置本应用的 URL。

```
from django.urls import path

from . import views

# app_name = 'article'

urlpatterns = [
```

```
path('article-column/', views.article_column, name='article_
column'),
]
```

下面就要编写前端模板了。

3.前端模板

现在要实现用户管理自己的文章栏目，对于这种管理功能，笔者把它视为所谓的“后台”，即不是显示给所有用户看的，只有发布内容的用户才能使用。所以，显示样式上也有所变化。

在 `./templates` 目录中建立 `article` 子目录，然后创建 `./templates/article/header.html` 文件，其代码如下。

```
{% load static %}

<div class="container">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <a href="http://www.itdiffer.com" class="navbar-brand">
        
      </a>
    </div>
    <div>
      <ul class="nav navbar-nav" role="tablist">
        <li><a href="{% url 'article:article_column' %}">文章管
理</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right" style="margin-right: 10px;">
        <li><a href="{% url 'blog:blog_title' %}">网站首页</a>
</li>
        <li><span>{{ user.username }}</span></li>
        <li><a href="{% url 'account:user_logout' %}">Logout</a>
</li>
      </ul>
    </div>
  </nav>
</div>
```

读者仔细观察上述代码，能够发现与 `./templates/header.html` 源码相差不大。

在用户登录后，要管理自己的文章栏目，就要进入所谓的“后台”，可是入口在哪里？所以，也要为登录用户设置入口，入口位置与前面设置的“修改密码”和“个人信息”的位置一样，所以顺便修改 `./templates/header.html` 文件设置入口。注意，这里通过入口进入后台的页面，笔者选择了现在正准备做的这个页面，当然也可以修改为其他页面。

```
<li><a href="{% url 'article:article_column' %}">后台管理</a></li>
```

后台部分的 `footer.html` 文件，可以继续使用 `./templates/footer.html`。

一般的管理后台，左侧都有一个功能栏目，我们也来做一个。创建 `./templates/article/leftslider.html` 文件，代码如下。

```
<div class="bg-info">
  <div class="text-center" style="margin-top: 5px;">
    <p><a href="{% url 'article:article_column' %}">栏目管理</a>
  </p>
</div>
</div>
```

下面就组装 `./templates/article/base.html` 文件，代码如下。

```
{% load static %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>{% block title %}{% endblock title %}</title>
  <link rel="stylesheet" href="{% static 'css/bootstrap.css' %}">
</head>
<body>
  <div class="container">
    {% include "article/header.html" %}
    <div class="col-md-2">
      {% include "article/leftslider.html" %}
    </div>
    <div class="col-md-10">
      {% block content %}
      {% endblock content %}
    </div>
  </div>
</body>
</html>
```

```

    </div>
    {% include "footer.html" %}
</div>
</body>
</html>

```

编写视图函数所要求的模板，建立

`./templates/article/column/article_column.html` 文件。

```

{% extends "article/base.html" %}
{% load static %}
{% block title %}
article column
{% endblock title %}

{% block content %}
<div>
  <p class="text-right">
    <button class="btn btn-primary">add column</button>
  </p>
  <tabel class="table table-hover">
    <tr>
      <td>序号</td>
      <td>栏目名称</td>
      <td>操作</td>
    </tr>
    {% for column in columns %}
    <tr>
      <td>{{ forloop.counter }}</td>
      <td>{{ column.column }}</td>
      <td>--</td>
    </tr>
    {% empty %}
    <p>还没有设置栏目，太懒了。</p>
    {% endfor %}
  </tabel>
</div>
{% endblock content %}

```

团子注：注意第一句，`{% extends "article/base.html" %}`，所以 `extends` 的开始目录就是 `templates`。

`{{ forloop.counter }}` 中的 `forloop` 只在循环内部起作用，它是一个模板变量，具有提示循环进度的属性，比如这里使用 `forloop.counter` 的效果是得到每个循环的序号，其本质是显示当前循环次数的计数器（所以从1开始了）。如果变量 `columns` 引用的对象为空，则通过 `{% empty %}` 执行后面的内容。循环语句中的 `{% empty %}` 省略了通过 `if` 来判断。

在浏览器的地址栏中输入<http://localhost:8000/article/article-column/>

团子注：调试了老半天，又是 404，这个错误出现好多次了。注意箭头，一个有 / 一个没 / 。

Page not found (404)

Request Method: GET

Request URL: http://localhost:8000/article/article-column/

Using the URLconf defined in `mysite.urls`, Django tried these URL patterns, in this order:

1. admin/
2. blog/
3. account/
4. article/ article-column [name='article_column']

The current path, `article/article-column/`, didn't match any of these.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

错误原因在二级路由，少了一个 / !!!

ItDiffer.com 管理后台 文章管理 jpch89 网站首页 Logout

栏目管理 add column

还没有设置栏目，太懒了。

序号	栏目名称	操作
----	------	----

copy right www.itdiffer.com

团子注：疑问，“还没有设置栏目，太懒了”这一句怎么跑到表格上面了??

4.增加新栏目

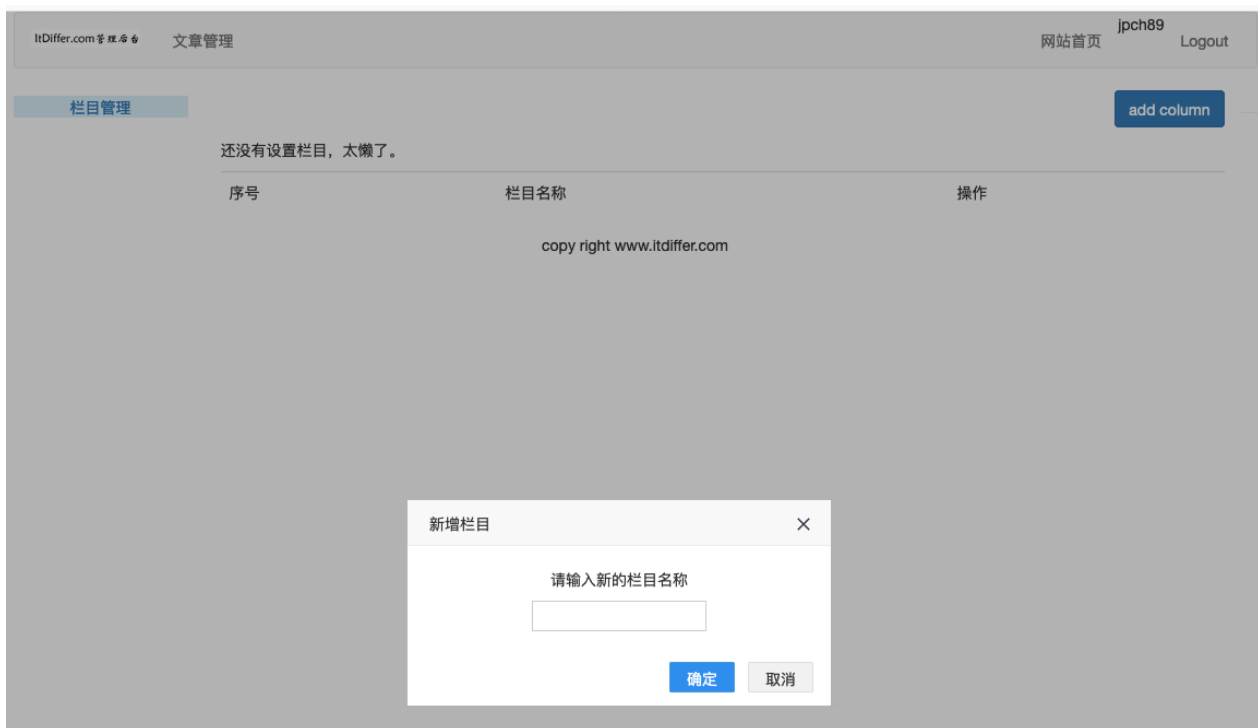
增加新栏目的操作流程是单击 `add column` 按钮，弹出一个对话框，在这个对话框中输入新的栏目名称。

为了实现弹出对话框，还是用第 2 章中使用的 `layer.js` 插件。编辑 `./templates/article/column/article_column.html` 文件，将原来的 `<button>` 进行适当修改，代码如下。


```
<button class="btn btn-primary" onclick="add_column()" id="add c
olumn">
    add column
</button>
```

当用户单击这个按钮时，触发 `add_column()` 函数（JavaScript函数），这个函数代码如下（下面的代码放在文件尾部 `{% endblock %}` 之内）。提醒读者，这种将 `JavaScript` 函数和 `HTML` 绑定到一起写的方法，在真实的项目中不值得提倡，本书中的项目因为主要是学习 `Django`，所以为了阅读和理解方便，就没有将代码分开，请读者注意。

```
<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'js/layer.js' %}"></script>
<script>
    function add_column() {
        var index = layer.open({
            type: 1,
            skin: 'layui-layer-rim',
            area: ['400px', '200px'],
            title: '新增栏目',
            content: '\
                <div class="text-center" style="margin-top: 20px;">\
                    <p>请输入新的栏目名称</p>\
                    <p><input type="text"></p>\
                </div>',
            btn: ['确定', '取消'],
            yes: function(index, layero) {
                column_name = $('#id_column').val()
                alert(column_name)
            },
            btn2: function(index, layero) {
                layer.close(index)
            }
        })
    }
</script>
```



这仅仅是效果，要实现真正的业务，还要继续完善后端的代码。

编辑 `./article/views.py` 文件，重写 `article_column()` 视图函数，下面是所有代码。

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt

from .forms import ArticleColumnForm
from .models import ArticleColumn

login_required
csrf_exempt
def article_column(request):
    if request.method == 'GET':
        columns = ArticleColumn.objects.filter(user=request.user)
        column_form = ArticleColumnForm()
        return render(request, 'article/column/article_column.html', {
            'columns': columns,
            'column_form': column_form
        })

    if request.method == 'POST':
        column_name = request.POST['column']
```

```

        columns = ArticleColumn.objects.filter(
            user_id=request.user.id,
            column=column_name
        )
        if columns:
            return HttpResponse('2')
        else:
            ArticleColumn.objects.create(user=request.user, column=column_name)
            return HttpResponse('1')

```

团子注：目前对 `csrf_exempt` 的理解是，取消了 `csrf` 验证，从而可以让 `post` 请求直接提交到被装饰的视图函数。下面讲了两种取消的方法。

未读参考文档：<https://www.jianshu.com/p/a178f08d9389>

另外，`filter` 带两个条件好像有什么陷阱？我忘记了。

根据要展示的效果，对 `./templates/article/column/article_column.html` 中的 `JavaScript` 部分代码进行适当修改，修改之后的代码如下。

```

<script>
function add_column() {
    var index = layer.open({
        type: 1,
        skin: 'layui-layer-rim',
        area: ['400px', '200px'],
        title: '新增栏目',
        content: '<div class="text-center" style="margin-top:20px">\
                <p>请输入新的栏目名称</p>\
                <p>{{ column_form.column }}</p>\
            </div>',
        btn: ['确定', '取消'],
        yes: function(index, layero) {
            column_name = $('#id_column').val()
            $.ajax({
                url: '{% url "article:article_column" %}',
                type: 'POST',
                data: {column: column_name},
                // data: {"column": column_name},
                success: function(e) {
                    if (e === '1') {
                        parent.location.reload()
                    }
                }
            })
        }
    })
}

```

```
        layer.msg('good')
    } else {
        layer.msg('此栏目已有，请更换名称')
    }
}
}),
btn2: function(index, layero) {
    layer.close(index)
}
})
}
</script>
```

团子注：原书的 `data` 被我注释掉了。

特别提醒读者，在实际的项目中，上述代码还缺少一些东西，那就是对用户输入的内容进行判断。一般来讲，我们要限制用户输入的字符内容，比如栏目名称只允许是字母和汉字，为此需要在得到栏目名称 `column_name` 之后，用正则表达式来判断该名称是否合法。如果合法，就使用 `Ajax` 传送数据，否则提示用户重新为栏目命名。读者可以自行修改上述 `JavaScript` 代码，实现上述判断。

团子注：意思是在前台校验？

完成上述代码后，通过测试可以检查是否实现预期功能。

ItDiffer.com管理后台

文章管理

网站首页

jpch89

Logout

栏目管理

add column

序号	栏目名称	操作
1	Python	--
2	Django	--
3	Rust	--

copy right www.itdiffer.com

添加栏目的功能已经实现，但这仅仅是栏目管理的部分功能，还有栏目的删除、编辑等功能需要添加。

编辑栏目

在栏目列表中，有一项“操作”列，其下面应该列出能够对本行栏目名称实施的操作。在

本项目中，计划设置“删除”和“编辑”两个操作。

首先，通过修改前端 `./templates/article/column/article_column.html` 文件，显示两种操作图标。将原来“操作”列下面对应的代码进行修改（原来是 `<td>--</td>`），其中“`--`”用如下代码替换）。

```
<a href="javascript:" name="edit" onclick="edit_column(this, {{
    column.id }})">
    <span class="glyphicon glyphicon-pencil"></span>
</a>
<a href="javascript:" name="delete" onclick="del_column(this, {{
    column.id }})">
    <span class="glyphicon glyphicon-trash" style="margin-left: 20
px;"></span>
</a>
```

刷新<http://localhost:8000/article/article-column/>页面，可以看到出现了两个图标，通常“铅笔”图标对应“编辑”功能，“垃圾桶”图标对应“删除”功能。

为了能够实现对栏目名称的修改，要再写一个视图函数（位于 `./article/views.py`），代码如下。

```
from django.views.decorators.http import require_POST

login_required(login_url='/account/login/')
require_POST
csrf_exempt
def rename_article_column(request):
    column_name = request.POST['column_name']
    column_id = request.POST['column_id']
    try:
        line = ArticleColumn.objects.get(id=column_id)
        line.column = column_name
        line.save()
        return HttpResponse('1')
    except:
        return HttpResponse('0')
```

在上述代码中，多了一个装饰器 `@require_POST`，所以一定要在文件顶部声明 `from django.views.decorators.http import require_POST`，使用这个装饰器的目的就是保证此视图函数只接收通过 `POST` 方式提交的数据。

在 `./article/urls.py` 中配置 `URL`，代码如下。

```
path('rename-column/', views.rename_article_column, name='rename_article_column'),
```

后端工作完成，继续修改前端模板文件。在

`./templates/article/column/article_column.html` 文件中编写 `JavaScript` 代码，如下所示，实现“操作”列下的“编辑”按钮中 `onclick` 请求的函数功能。

```
function edit_column(the, column_id) {
    var name = $(the).parents('tr').children('td').eq(1).text()
    var index = layer.open({
        type: 1,
        skin: 'layui-layer-rim',
        area: ['400px', '200px'],
        title: '编辑栏目',
        content: '<div class="text-center" style="margin-top: 20px;">\
                <p>请编辑的栏目名称</p>\
                <p><input type="text" id="new_name" value="' + name + '"></p>\
                </div>',
        btn: ['确定', '取消'],
        yes: function(index, layero) {
            new_name = $('#new_name').val()
            $.ajax({
                url: '{% url "article:rename_article_column" %}',
                type: 'POST',
                data: {
                    column_id,
                    column_name: new_name,
                },
            },
            success: function(e) {
                if (e === '1') {
                    parent.location.reload()
                    layer.msg('good')
                } else {
                    layer.msg('新的名称没有保存，修改失败。')
                }
            }
        })
    })
}
```

```
}
```

团子注： `data` 对象中我用到了 `Javascript` 对象的简写方法。

在<http://localhost:8000/article/article-column/>页面中，单击代表“编辑”的铅笔图标，实现修改栏目名称的功能。

删除栏目

先编写视图函数，编辑 `./article/views.py` 文件。

```
login_required
@require_POST
@csrf_exempt
def del_article_column(request):
    column_id = request.POST['column_id']
    try:
        line = ArticleColumn.objects.get(id=column_id)
        line.delete()
        return HttpResponse('1')
    except:
        return HttpResponse('2')
```

然后设置 `URL`，在 `./article/urls.py` 文件中增加如下代码。

```
path('del-column/', views.del_article_column, name='del_article_
column'),
```

接下来在模板文件 `./templates/article/column/article_column.html` 中编写一个名为 `del_column` 的 `JavaScript` 函数。

```
function del_column(the, column_id) {
    const name = $(the).parents('tr').children('td').eq(1).text()
    layer.open({
        type: 1,
        skin: 'layui-layer-rim',
        area: ['400px', '200px'],
        title: '删除栏目',
```

```


content: '<div class="text-center" style="margin-top: 20px;">\
    <p>是否确定删除{' + name + '}栏目</p>\
    </div>',
btn: ['确定', '取消'],
yes: function() {
    $.ajax({
        url: '{% url "article:del_article_column" %}',
        type: 'POST',
        data: {column_id},
        success: function(e) {
            if (e==='1') {
                parent.location.reload()
                layer.msg('has been deleted.')
            } else {
                layer.msg('删除失败')
            }
        }
    })
}
})
}
}
}

```

团子注：这里记一下调试 Bug 记录。

点击控制台报错信息，找了半天原来把 `function` 写错了。。。

❗ SyntaxError: missing } after property list [\[Learn More\]](#) article-column:181:20
 note: { opened at line 172, column 13 article-column:172:13




```

function del_column(the, column_id) {
    var name = $(the).parents('tr').children('td').eq(1).text()
    layer.open({
        type: 1,
        skin: 'layui-layer-rim',
        area: ['400px', '200px'],
        title: '删除栏目',
        content: '<div class="text-center" style="margin-top: 20px;">\
            <p>是否确定删除{ ' + name + '}栏目</p>\
            </div>'
        btn: ['确定', '取消'],
        yes: function() {
            $.ajax({
                url: '/article/del-column/',
                type: 'POST',
                data: {column_id},
                success: function(e) {
                    if (e==='1') {
                        parent.location.reload()
                        layer.msg('has been deleted.')
                    } else {
                        layer.msg('删除失败')
                    }
                }
            })
        }
    })
}

```



另外以后可以根据颜色来查错，VS Code 有颜色的，function 写对的时候是紫色。

最后，这里的 layer.open 没有写 btn2，默认就是关闭当前窗口，值得注意！

如果要将“文章栏目”的管理权限赋给超级管理员，就编辑 ./article/admin.py 文件，输入如下代码。

团子注：什么叫赋给超级管理员，其实就是让 Django 管理后台来管理文章栏目这一模型，所以需要在 admin 站点上面注册。

```

from django.contrib import admin

from .models import ArticleColumn

class ArticleColumnAdmin(admin.ModelAdmin):
    class Meta:
        list_display = ('column', 'created', 'user')
        list_filter = ('column', )
    admin.site.register(ArticleColumn, ArticleColumnAdmin)

```

站点管理

ACCOUNT		
User infos	+ 增加	✎ 修改
User profiles	+ 增加	✎ 修改
ARTICLE		
Article columns	+ 增加	✎ 修改
BLOG		
Blog articles	+ 增加	✎ 修改
认证和授权		
用户	+ 增加	✎ 修改
组	+ 增加	✎ 修改

最近动作

我的动作

无可用的

不为所动，掩卷细思本节的所有代码，这是一种重要的学习方法——**反省**（程序员的修养）

知识点

1.模板语法：继承和包含

在编写前端模板时，通常会使用模板的继承和包含，这都是为了尽可能避免重复前端代码。

首先要定义一个基础模板。在基础模板中，通常会使用很多的 `{% block %}`，其基本格式是：

```
{% block name %}
    <!--html-->
{% endblock %}
```

每个块中可以写 `HTML` 代码，也可以为空。一般来说，在基础模板中定义 `{% block %}` 数量多一些，是绝对有好处的。如果将来在子模板中有相同名称（name）的块，就会将 `base.html` 中定义的块覆盖。

在子模板中继承的方式是使用 `{% extends "base.html" %}`，并且放在该页面中的第一个模板标签位置，一般放在第一行最保险。在子模板中如果重写了父模板中的某个块，则按照子模板中的方式显示。

此外，在模板中还可以使用 `{% include "templatename.html" %}` 包含其他模板。

为了更灵活地使用 `JavaScript` 和 `CSS`，通常不在 `base.html` 里面引入相关文件。在真实的项目中，建议在 `base.html` 中设定 `{% block js %}` `{% endblock %}` 和 `{% block css %}` `{% endblock %}`，然后在子模板中重写这两个块，最后调入该模板页面所需要的 `JavaScript` 和 `CSS`。

团子注：其实就是按需加载。

2.模型：查询

前面已经提到，`Django` 封装了对数据的操作，可以使用更 `Python` 的方式实现数据查询，而不是使用 `SQL` 语句。

```
>>> from blog.models import BlogArticles
>>> BlogArticles.objects.all()
<QuerySet [ <BlogArticles: Canglaoshi is a good teacher>, <BlogArticles: Physics create>,
<BlogArticles: Python is for you>, <BlogArticles: php is the best computer language>]>
```

`BlogArticles` 是一个数据模型类，可以用 `Python` 中常用的 `dir()` 和 `help()` 两个方法来研究这个类（也是一个对象）所具有的属性和方法。

```
>>> dir(BlogArticles)
['DoesNotExist', 'MultipleObjectsReturned', '__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', '_check_column_name_clashes',
 '_check_field_name_clashes', '_check_fields', '_check_id_field', '_check_index_together',
 '_check_local_fields', '_check_long_column_names', '_check_m2m_through_same_relationship',
 '_check_managers', '_check_model', '_check_ordering', '_check_swappable',
 '_check_unique_together', '_do_insert', '_do_update', '_get_FIELD_display',
 '_get_next_or_previous_by_FIELD', '_get_next_or_previous_in_order', '_get_pk_val',
 '_get_unique_checks', '_meta', '_perform_date_checks', '_perform_unique_checks',
 '_save_parents', '_save_table', '_set_pk_val', 'author', 'author_id', 'body', 'check',
 'clean', 'clean_fields', 'date_error_message', 'delete', 'from_db', 'full_clean',
 'get_deferred_fields', 'get_next_by_publish', 'get_previous_by_publish', 'id',
 'objects', 'pk', 'prepare_database_save', 'publish', 'refresh_from_db', 'save',
 'save_base', 'serializable_value', 'title', 'unique_error_message', 'validate_unique']
```

仔细观察不难发现，上面所列出来的属性和方法包括我们在定义 `BlogArticles` 类时自己定义的，也包括它从父类那里继承过来的。

其中有一个 `objects` 属性是经常被使用的，读者可以继续用

`dir(BlogArticles.objects)` 和 `help(BlogArticles.objects)` 查看其详细的属性和方法。

Help on Manager in module django.db.models.manager object:

```
class Manager(BaseManagerFromQuerySet)
|   Method resolution order:
|       Manager
|       BaseManagerFromQuerySet
|       BaseManager
|       builtins.object
#省略下面的各种方法和属性
```

这里展示的结果是一个 `Manager` 类（对象），每个数据模型类都有这个对象，或者说除非特别操作，每个数据模型类都有一个 `objects` 属性，并借着 `Manager` 对象中的各种属性和方法实现对数据库的基本查询操作。例如通过 `BlogArticles.objects.all()` 查询得到的结果是一个被称为 `QuerySet` 对象组成的列表，列表中的每个元素就是数据库表中的一条记录，也可以理解是 `BlogArticles` 类的一个实例。

```
>>> BlogArticles.objects.get(id=1)
<BlogArticles: php is the best computer language>
>>> BlogArticles.objects.get(title="Canglaoshi is a good teacher")
<BlogArticles: Canglaoshi is a good teacher>
>>> BlogArticles.objects.get(title="teacher")
Traceback (most recent call last):
  File "/usr/lib/python3.5/code.py", line 91, in runcode
    exec(code, self.locals)
  File "<console>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/manager.py", line 85,
in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/query.py", line 385, in
get
    self.model._meta.object_name
blog.models.DoesNotExist: BlogArticles matching query does not exist.
```

`get()` 是必须要精准匹配的查询方法，如果查询内容不存在，则会报错。因为是精准匹配，所以查询到的第一个总是符合条件的记录。下面分别列出几种常用的查询方式，供读者学习参考。

- 查询所有对象

```
>>> from django.contrib.auth.models import User
>>> users = User.objects.all()
>>> users
<QuerySet [<User: admin>, <User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>,
<User: zhoulaooshi>, <User: wulaoshi>, <User: danglaoshi>, <User: canglaoshi>, <User:
hulaoshi>, <User: yinlaoshi>, <User: lulaoshi>, <User: chenlaoshi>, <User: kuanglaoshi>,
<User: xiaodou>, <User: xiaojin>]>
```

查询结果 `users` 所引用的是一个序列（类列表的 `QuerySet` 对象，不可变），可以使用 `Python` 中某些对序列的操作方法来操作。

```
>>> lu = users[10]
>>> lu
<User: lulaoshi>
>>> lu.email
'lu@laoshi.com'
```

- 根据条件查询对象

在 SQL 中可以通过 `where` 设置查询条件，在 Django 的 `QuerySet` 中也有一个类似的方法 `filter()`。

```
>>> users_laoshi = User.objects.filter(username__endswith="laoshi")
>>> users_laoshi
```

```
<QuerySet [<User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>, <User:
zhoulaoshi>, <User: wulaoshi>, <User: danglaoshi>, <User: canglaoshi>, <User: hulaoshi>,
<User: yinlaoshi>, <User: lulaoshi>, <User: chenlaoshi>, <User: kuanglaoshi>]>
```

这里查询出 `username` 中以“laoshi”这个字符串结尾的对象。`filter()` 还有很多参数可以使用，读者可以参考官方文档。

除 `filter()` 能够实现根据条件查询外，还有 `exclude()` 和 `get()`，如下所示。

团子注：查询三剑客 `get()`、`filter()`、`exclude()`

`get()` 和 `filter()` 的区别在于，如果查询对象不存在，`get()` 会报错，而 `filter()` 返回空。

此外，查询操作还支持链接过滤。

```
>>> users = User.objects.filter(username__endswith="laoshi").exclude(id__gt=7)
>>> users
<QuerySet [<User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>, <User:
zhoulaoshi>, <User: wulaoshi>, <User: danglaoshi>]>
```

- 查询结果排序

对于查询返回的 `QuerySet` 结果，默认是按照数据模型类中定义的字段的排序的，如果要重新排序，可以使用 `order_by()` 方法。

```
>>> u = User.objects.filter(username__startswith="z").order_by("username")
>>> u
<QuerySet [ <User: zhanglaoshi>, <User: zhaolaoshi>, <User: zhoulaooshi> ]>
>>> u = User.objects.filter(username__startswith="z").order_by("-username")
>>> u
<QuerySet [ <User: zhoulaooshi>, <User: zhaolaoshi>, <User: zhanglaoshi> ]>
```

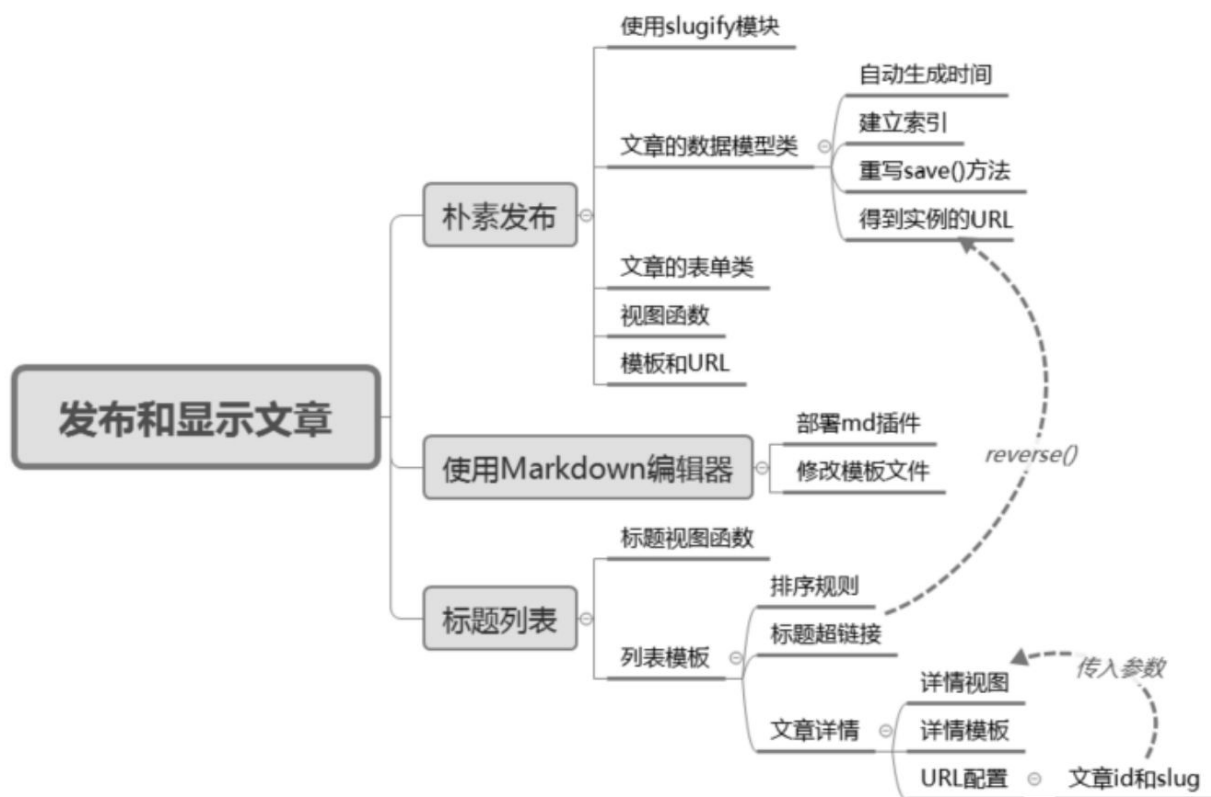
当然，`order_by()` 的参数中可以写多个字段名称，即按照字段前后顺序分别作为排序的“主关键词”和“次关键词”。

3.文档导读

- (1) Templates, <https://docs.djangoproject.com/en/2.1/topics/templates/>。
- (2) QuerySet API reference, <https://docs.djangoproject.com/en/2.1/ref/models/queriesets/>。
- (3) django-queryset-csv, <https://pypi.python.org/pypi/django-queryset-csv/>。

3.2 发布和显示文章

对文章的管理主要是发布、删除和编辑。对于每个部分，都有一些详细的要求，本节就围绕这三个功能，开发管理文章的程序。



简单的文章发布

首先介绍一个预备知识 `slug`，举例说明。

假如在数据库中有一篇标题为 `Learn Python in itdiffer.com` 的文章，要访问这篇文章，一种方法是使用这篇文章的 `id`，比如 `http://www.itdiffer.com/article/231`，`231` 就是这篇文章在数据库表中的 `id`；还有另外一种方法，是直接在URL中显示文章的标题（`http://www.itdiffer.com/article/Learn Python in itdiffer.com`），但实际上不能这样，因为在URL里面，空格都是用“%20”来表示的，所以上面那个地址的最终表现形式应该是 `http://www.itdiffer.com/article/Learn%20Python%20in%20itdiffer.com`。这种显示不是一个很友好的方案，这时候 `slug` 就可以发挥作用了，它实现的结果是 `http://www.itdiffer.com/article/Learn-Python-in-itdiffer.com`，看起来清清爽爽了，并且这种方式对搜索引擎也是友好的。

`Django` 中就提供了实现上述 `slug` 的方法。在本项目的根目录中执行 `python3 manage.py shell`，进入交互模式。

```
>>> from django.utils.text import slugify
>>> slugify('Learn python in itdiffer.com')
'learn-python-in-itdiffercom'
```

表现完美。不过我们有时也会用中文写标题，不总是用英文的。那么中文怎样转换呢？

```
>>> slugify('跟老齐学')
''
```

这就不完美了，`Django` 内置的这个方法不能处理中文，它返回的是空。安装一个能够解决中文的第三方库。

```
pip install awesome-slugify
```

安装之后，还是在交互模式中，代码如下。

```
>>> from slugify import slugify
>>> slugify('跟老齐学Django')
'Gen-Lao-Qi-Xue-Django'
>>> slugify('Learn python in itdiffer.com')
'Learn-python-in-itdiffer-com'
```

自动将汉字转换为拼音，这样就解决了前面遇到的汉字问题，并且英语依然适用。仔细观察，与 `Django` 自带的还稍微有一些区别呢！

下面开始按套路出牌。

1.创建数据模型类

在 `./article/models.py` 文件中建立一个数据模型，命名为 `ArticlePost`，作为文章的数据模型对象。

```
class ArticlePost(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='article')
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=500)
    column = models.ForeignKey(ArticleColumn, on_delete=models.CASCADE, related_name='article_column')
    body = models.TextField()
    created = models.DateTimeField(default=timezone.now)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('title', )
        index_together = (('id', 'slug'), )

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super(ArticlePost, self).save(*args, **kwargs)

    def get_absolute_url(self):
        return reverse('article:article_detail', args=[self.id, self.slug])
```

在 `./mysite/settings.py` 文件中，有一个 `TIME_ZONE` 项，读者要确认一下是否使用 `TIME_ZONE= 'Asia/Shanghai'` 时区，`default=timezone.now` 即得到文章发布时的日期和时间。

`index_together` 作用是对数据库中这两个字段建立索引，在后面，会通过每篇文章的 `id` 和 `slug` 获取该文章对象，这样建立了索引之后，能提高读取文章对象的速度。

每个数据模型类都有一个 `save` 方法，对此方法进行重写，其目的就是要实现 `self.slug = slugify(self.title)`。

`get_absolute_url` 是要获取某篇文章对象的 `URL`，这里暂时用不到，后面会用到。前面引入模块时看到的 `reverse` 就在这个方法中使用了。

新的数据模型建立后，就要**迁移数据，从而建立数据库表**。

```
python manage.py makemigrations
Migrations for 'article':
  article/migrations/0002_articlepost.py
    - Create model ArticlePost

python manage.py migrate
Operations to perform:
  Apply all migrations: account, admin, article, auth, blog, contenttypes, sessions
Running migrations:
  Applying article.0002_articlepost... OK
```

2.创建表单类

编辑 `./article/forms.py` 文件，增加如下代码。

```
from .models import ArticleColumn, ArticlePost

class ArticlePostForm(forms.ModelForm):
    class Meta:
        model = ArticlePost
        fields = ('title', 'body')
```

3.创建视图函数

编辑 `./article/views.py` 文件。

```
from .forms import ArticlePostForm

@login_required(login_url='/account/login/')
@csrf_exempt
def article_post(request):
    if request.method == 'POST':
        # article_post_form = ArticlePostForm(data=request.POST)
        # 团子注：应该可以不用关键字参数
        article_post_form = ArticlePostForm(request.POST)
        if article_post_form.is_valid():
            cd = article_post_form.cleaned_data
```

```

        try:
            new_article =
            article_post_form.save(commit=False)
            new_article.author = request.user
            new_article.column = request.user.article_column.
            get(
                id=request.POST['column_id']
            )
            new_article.save()
            return HttpResponse('1')
        except:
            return HttpResponse('2')
    else:
        return HttpResponse('3')
else:
    article_post_form = ArticlePostForm()
    article_columns = request.user.article_column.all()
    return render(request,
        'article/column/article_post.html', {
            'article_post_form': article_post_form,
            'article_columns': article_columns
        })

```

团子注：见到了好多返回 `HttpResponse('数字')` 的形式，这种应该都是通过弹窗提交，或者 `AJAX` 提交数据，然后根据 `数字` 响应，来判断应该进行的操作。

为什么没有使用比较熟悉的

`article_columns=ArticleColumn.objects.filter(user=request.user)` 呢？这么写当然可以，在此使用 `request.user.article_column.all()` 的目的在于引起读者关注 `ArticleColumn` 数据模型类中的 `user=models.ForeignKey(User,related_name='article_column')`，这里有一个参数 `related_name`，为了理解它的作用，还是启用交互模式。

团子注： `related_name` 就是反向关联名。

```

>>> from article.models import ArticleColumn
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(username='jpch89')
>>> user
<User: jpch89>

```

```
>>> user.article_column.all()
<QuerySet [<ArticleColumn: Python>, <ArticleColumn: Rust>]>
```

`user` 是一个用户对象实例，在 `ArticleColumn` 中通过 `user=models.ForeignKey(User,related_name='article_column')` 将 `ArticleColumn` 类与 `User` 类建立外键关系。那么，如何通过 `user` 实例查找到其名下 `ArticleColumn` 的所有实例？因为在 `ForeignKey()` 中使用了 `related_name` 参数，于是就可以通过 `user.article_column.all()` 的方式得到。

初步理解了 `related_name` 之后，再来看上述代码中的 `request.user`，在前面也用到过，这里稍微讲解一下。

当一个页面被请求时，`Django` 创建一个包含请求元数据的 `HttpRequest` 对象，然后把 `HttpRequest` 作为视图函数的第一个参数传入，每个视图要负责返回一个 `HttpResponse` 对象。这个 `HttpResponse` 对象有很多属性，其中包括读者熟知的 `method`、`GET` 和 `POST`。`user` 也是一个属性，在用户登录之后，`request.user` 即为 `User` 类的一个实例。

4.设置URL

编辑 `./article/urls.py` 文件，添加如下 `URL` 配置语句。

```
path('article-post/', views.article_post, name='article_post'),
```

5.编写模板

在 `./templates/article/column` 目录中创建一个名为 `article_post.html` 的文件，其代码如下。

```
{% extends "article/base.html" %}

{% load static %}

{% block title %}
article column
{% endblock title %}

{% block content %}
<div class="margin-left:10px">
  <form action="." class="form-horizontal" method="post">
    {% csrf_token %}
    <div class="row" style="margin-top: 10px;">
      <div class="col-md-2 text-right"><span>标题：</span></div>
```

```

        <div class="col-md-10 text-left">{{ article_post_form.title }}</div>
    </div>
    <div class="row" style="margin-top: 10px;">
        <div class="col-md-2 text-right"><span>栏目: </span></div>
        <div class="col-md-10 text-left">
            <select id="which_column">
                {% for column in article_columns %}
                <option value="{{ column.id }}">{{ column.column }}</o
ption>
                {% endfor %}
            </select>
        </div>
    </div>
    <div class="row" style="margin-top: 10px;">
        <div class="col-md-2 text-right"><span>内容: </span></div>
        <div class="col-md-10 text-left">{{ article_post_form.body
}}</div>
    </div>
    <div class="row">
        <input type="button" value="发布" class="btn btn-primary bt
n-lg" onclick="publish_article()">
    </div>
</form>
</div>
<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'js/layer.js' %}"></script>
<script>
function publish_article() {
    var title = $('#id_title').val()
    var column_id = $('#which_column').val()
    var body = $('#id_body').val()
    $.ajax({
        url: '{% url "article:article_post" %}',
        type: 'POST',
        data: {title, body, column_id},
        success: function(e) {
            if (e === '1') {
                layer.msg('successful')
            } else if (e === '2') {
                layer.msg('sorry.')
            } else {
                layer.msg('项目名称必须写，不能空。')
            }
        }
    })
}

```

```
    })
  }
</script>
{% endblock content %}
```

前端模板还没有结束，还要修改 `./templates/article/leftslider.html` 文件，目的是设置一个发布文章的入口。

```
<p><a href="{% url 'article:article_post' %}">发布文章</a></p>
```

将上面这行代码放在 `leftslider.html` 文件的合适位置。

6.测试本功能

在浏览器的地址栏中输入<http://localhost:8000/article/article-post/>，然后在打开的页面中输入要发布的内容。

ItDiffer.com管理后台

文章管理

网站首页

jpch89

Logout

栏目管理

发布文章

标题:

Python Django 学习笔记

栏目:

Python

内容:

一起学 Django 啦~~~

发布

copy right www.itdiffer.com

successful

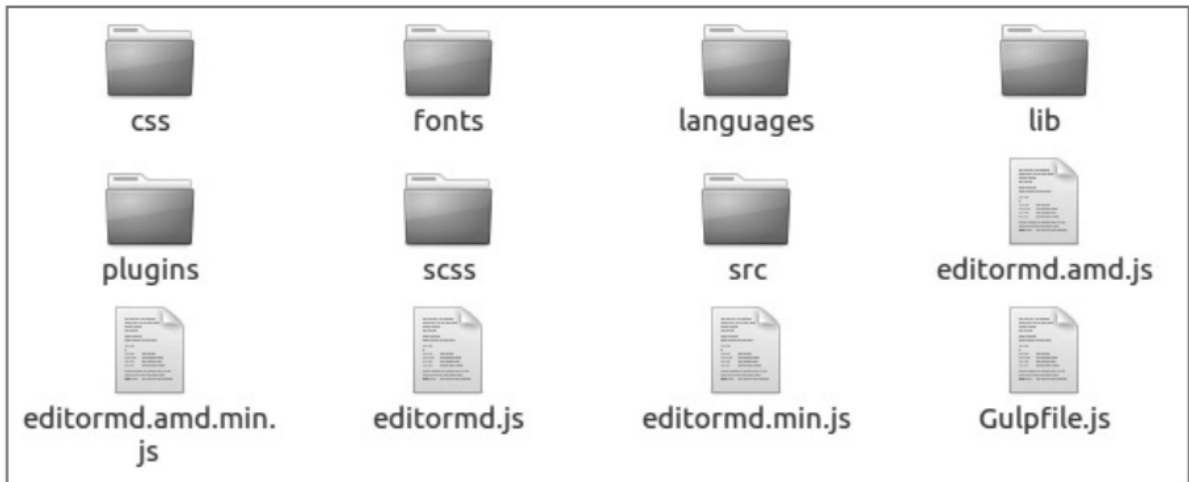
使用Markdown

Markdown 是一种轻量级的标记语言，由 **John Gruber** 于 **2004** 年创立。虽然是一种“标记语言”（HTML也是标记语言），但是用起来并不难，也不需要用户痛苦地记忆，因为它语法简单、标记符数量少，更重要的是可以完全可视化操作，就像使用某种可视化的文档编辑工具一样。

在本项目中，我们就使用这样一个 **Markdown** 编辑器，其来源是<https://github.com/qiwsir/editor.md>。在此对这个 **Markdown** 插件的开发者 **pandao** 表示感谢。

1.部署相关文件

从 [GitHub](#) 上把刚才推荐的插件下载并解压缩，然后在本项目的 `static` 目录中建立一个子目录 `editor`，从已经解压缩的 `editor.md-master` 目录中，复制一些目录到刚刚建立的 `./static/editor` 目录中，所复制的项目如图所示。



2.修改模板

这个插件主要是基于前端的，所以只需要修改

`./templates/article/column/article_post.html` 文件。首先，在文件中引入两个 `CSS` 文件。

```
<link rel="stylesheet" href="{% static 'editor/css/style.css'
%}">
<link rel="stylesheet" href="{% static 'editor/css/editormd.css'
%}">
```

其中，`editormd.css` 文件是所下载的插件自带的，另外一个 `style.css` 文件是笔者根据所下载的插件中的例子改写的，代码如下。

```
#layout {
    text-align: left;
}

#layout > header, .btns {
    padding: 15px 0;
    width: 90%;
    margin: 0 auto;
}

#layout > header > h1 {
```

```
font-size: 20px;
margin-bottom: 10px;
}
```

在 `./templates/article/column/article_post.html` 文件中找到如下代码。

```
<div class="col-md-10 text-left">{{ article_post_form.body }}</div>
```

用下面的代码替换上面找到的代码。

```
<div id="editormd" class="col-md-10 text-left">
  <textarea id="id_body" style="display:none;"></textarea>
</div>
```

最后在模板文件尾部引入 `JavaScript` 文件和相应的脚本代码。

```
<script src="{% static 'editor/editormd.min.js' %}"></script>
<script>
  $(function() {
    var editor = editormd('editormd', {
      width: '100%',
      height: 640,
      syncScrolling: 'single',
      path: '{% static "editor/lib/" %}',
    })
  })
</script>
```

在浏览器的地址栏中输入<http://localhost:8000/article/article-post/>（如果用户没有登录，系统会提示登录），就可以看到发布文章的界面。

通过单击“发布”按钮，就能够把文章发布，但是还不能浏览，接下来就要显示文章及其标题列表。

文章标题列表

1.简单的标题列表

编辑 `./article/views.py` 文件，编写一个简单的视图函数。

```
@login_required(login_url='/account/login/')
def article_list(request):
    articles = ArticlePost.objects.filter(author=request.user)
    return render(request, 'article/column/article_list.html',
                  {'articles': articles})
```

创建模板文件 `./templates/article/column/article_list.html`，输入如下代码。

```
{% extends "article/base.html" %}

{% load static %}
```



```

{% block title %}
articles list
{% endblock title %}

{% block content %}
<div>
  <table class="table table-hover">
    <tr>
      <td>序号</td>
      <td>标题</td>
      <td>栏目</td>
      <td>操作</td>
    </tr>
    {% for article in articles %}
      <tr>
        <td>{{ forloop.counter }}</td>
        <td>{{ article.title }}</td>
        <td>{{ article.column }}</td>
        <td>--</td>
      </tr>
    {% endfor %}
  </table>
</div>
{% endblock content %}

```

上述模板代码与文章栏目名称列表代码类似，实现了简单的标题列表。最后一步就是设置 URL，编辑 `./article/urls.py` 文件，增加下面的代码。

```
path('article-list/', views.article_list, name='article_list'),
```

<http://localhost:8000/article/article-list/>

ItDiffer.com 管理后台 文章管理		网站首页 jpch89 Logout	
栏目管理	序号	标题	栏目
发布文章	1	Python Django 学习笔记	Python
	2	你好Python	Python

copy right www.itdiffer.com

多发布几篇文章，观察一下这些文章是按照什么顺序排列的。请读者翻阅前面写的 `ArticlePost` 类，其中有下面的代码。

```
class Meta:
    ordering = ("title",)
    index_together = (('id', 'slug'),)
```

如果以前没有体会到 `ordering=("title",)` 的作用，现在应该体会到了，仔细观察文章标题列表的结果，是不是按照标题名称的顺序排列的？的确是。

如果要修改为按照文章编辑/发布的时间倒序排列，就应该修改 `ordering` 的值，代码如下。

```
ordering = ('-updated', )
```

保存之后，再次刷新页面，就看到所期望的结果了。

为了访问方便，左侧的功能栏当然要增加**显示文章标题**的入口，编辑 `./templates/article/leftslider.html` 文件，在读者已经熟悉的位置增加下面的代码。

```
<p><a href="{% url 'article:article_list' %}">文章列表</a></p>
```

ItDiffer.com 管理后台	文章管理	网站首页	jpch89	Logout
栏目管理	序号	标题	栏目	操作
发布文章	1	你好Python	Python	--
文章列表	2	Python Django 学习笔记	Python	--

copy right www.itdiffer.com

2. 查看文章

文章标题列出来了，接下来就是为其设置超链接，单击标题后就能查看文章内容。在第1章中我们通过文章的 `id` 查看详细内容，这里要换一种方式查看。

请读者再次阅读 `ArticlePost` 类的代码，里面有如下内容。

```
def get_absolute_url(self):
    return reverse('article:article_detail', args=[self.id, self.slug])
```

`reverse()` 函数的调用方式是：

```
reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)
```

参数 `viewname` 就是在每个应用的 `urls.py` 中设置 URL 时 `name` 的值。比如，前面刚刚在 `./article/urls.py` 中增加的 URL (`path('article-list/', views.article_list, name="article_list")`) 配置中的 `name="article_list"`。

还是以上述 URL 为例，如果要向该视图函数发出请求，可以使用如 `localhost:8000/article/article-list/` 的形式，其中 `localhost:8000` 是域名和端口部分，而 `/article/article-list/` 就是路径部分，严格说是请求资源的 URI (Uniform Resource Identifier，统一资源定位符)。

团子注：URL 是一大坨东西，URI 是其中的路径部分。

在模板文件中，遇到了需要使用链接地址时，可以使用 `{% url viewname %}` 的形式，比如：

```
<a href='{% url "article:article_list" %}'>文章列表</a>
```

其实，上述代码也可以写成如下形式。

```
<a href='/article/article-list/'>文章列表</a>
```

为了避免硬编码问题，所以采用前述方式，只要规定的名称不变化，路径设置的修改不会影响这里的超链接对象的访问。

这是在模板中，如果在视图函数中呢？比如：

```
HttpResponse('/article/article-list/')
```

如果这样写，也是“硬编码”的风格，要避免。代替它的就是使用 `reverse()` 函数。

```
HttpResponse(reverse('article:article_list'))
```

通过 `reverse('article:article_list')` 实现了 `'/article/article-list/'`，即从 `name` 到 `path`，而在 URL 配置中是从 `path` 到 `name`，因此 `reverse()` 起到

了“逆向、颠倒”的作用。

`reverse()` 参数中除 `viewsname` 外，还可以传入其他数值，比如前面的 `reverse("article:article_detail",args=[self.id,self.slug])`。

如果读者打算深入了解这个函数的执行过程，还可以看一下该函数的源码，网址是<https://docs.djangoproject.com/en/2.1/ref/urlresolvers/>。

在 `ArticlePost` 类中的 `get_absolute_url()` 方法，就是要得到相应文章的路径，因此可以将其用于模板文件 `./templates/article/column/article_list.html` 的 `<a>` 标签中。

```
<td>
    <a href="{ article.get_absolute_url }">{{ article.title }}
</a>
</td>
```

团子注：也就是说，使用模型类的 `get_absolute_url` 方法在模板中获取链接。而在第 1 章里面，是这样获得文章超链接的： `{{ blog.title }}`。

我还注意到一点，以 `/` 开头的 `URI` 就是绝对路径，而不以 `/` 开头的 `URI` 就是相对路径。

这样修改模板文件，实现的效果是让每个标题都实现超链接，链接对象是该文章的详细内容。

刷新页面，会看到报错信息，这没有什么意外。

`NoReverseMatch at /article/article-list/`

“`NoReverseMatch at /article/article-list/`”，这意味着 `URL` 配置存在问题。另外，从图中信息可知，`self.id` 和 `self.slug` 已经被读取到，但是没有对应的 `URL` 来接收这两个值。所以，要增加 `article_detail` 的 `URL` 配置，用来显示文章的详情。在 `./article/urls.py` 中增加如下 `URL` 配置代码。

团子注：

原书说的不对，报错信息明明是找不到 `article_detail` 的反向解析，并没有说 `self.id` 和 `self.slug` 的事情。

```
from django.urls import re_path

urlpatterns = [
    ...,
    re_path('article-detail/(?P<id>\d+)/(?P<slug>[-\w]+)/$', view
ws.article_detail, name='article_detail'),
]
```

这种写法以前没有见到过，其中包含了两个参数 `<id>` 和 `<slug>`，并且用正则表达式约定了相应参数的形式，与之对应的视图函数 `article_detail()` 就应该包含这两个参数。

关于 `path()` 和 `re_path()` 的比较，请阅读

<https://docs.djangoproject.com/en/2.0/ref/urls/>中的相关内容。通过比较不难发现，`re_path()` 的用法相当于 Django 1.x中的 `url()`。

编辑 `./article/views.py` 文件，增加如下函数。

```
from django.shortcuts import get_object_or_404

@login_required(login_url='/account/login/')
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    return render(request, 'article/column/article_detail.html',
        {'article': article})
```

同时还要再次编写 `./templates/article/column/article_detail.html` 模板文件，为了快速地查看效果，先写一个简单的——这里暂时不要纠结内容是否按照输入格式显示问题。

```
{% extends "article/base.html" %}

{% block title %}
articles list
{% endblock title %}
```

```
{% block content %}
<div>
  <h1>{{ article.title }}</h1>
  <p>{{ user.username }}</p>
  <div>{{ article.body }}</div>
</div>
{% endblock content %}
```

刷新文章列表页面，可以看到每个文章标题都已经做好了超链接

单击某个标题，可以查看该标题所对应的文章

虽然文章内容的显示方式没有进行美化，但不要在乎其外表，还是要看本质，已经实现了基本功能。此外，还要特别观察一下 **URL** 的特点，与我们所设计的模式正好符合。

3.按排版格式显示

继续编辑显示文章内容的模板文件

`./templates/article/column/article_detail.html`，主要是将 **Markdown** 标记符转换为 **HTML** 标记符并显示在网页上，重新编辑之后的代码如下。

```
{% extends "article/base.html" %}

{% load static %}

{% block title %}
articles list
{% endblock title %}

{% block content %}
<div>
  <header>
    <h1>{{ article.title }}</h1>
    <p>{{ user.username }}</p>
  </header>

  <link rel="stylesheet" href="{% static 'editor/css/editormd.pr
view.css' %}">
  <div id="editormd-view">
    <textarea id="append-test" style="display:none;">
      {{ article.body }}
```

```

        </textarea>
    </div>
</div>

<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'editor/lib/marked.min.js' %}"></script>
<script src="{% static 'editor/lib/prettify.min.js' %}">
</script>
<script src="{% static 'editor/lib/raphael.min.js' %}"></script>
<script src="{% static 'editor/lib/underscore.min.js' %}"></scri
pt>
<script src="{% static 'editor/lib/sequence-diagram.min.js' %}">
</script>
<script src="{% static 'editor/lib/flowchart.min.js' %}"></scrip
t>
<script src="{% static 'editor/lib/jquery.flowchart.min.js' %}">
</script>
<script src="{% static 'editor/editormd.js' %}"></script>

<script>
$(function(){
    editormd.markdownToHTML('editormd-view', {
        htmlDecode: 'style,script,iframe',
        emoji: true,
        tasklist: true,
        tex: true, // 默认不解析
        flowChart: true, // 默认不解析
        sequenceDiagram: true, // 默认不解析
    })
})
</script>
{% endblock content %}

```

发现标题有问题！此时需要删除 `{{ article.body }}` 前面的空格。

因为默认把 `textarea` 里面的东西当做是 `Markdown` 文本来解析，前面如果有空格，就相当于产生了缩进块，缩进块会被当做 `<pre>` 即 `Preformatted Text` 预格式化文本来解析。

删除前面的空格后，刷新页面：

上述代码中引入了大量的 `JavaScript` 文件，都是我们所使用的 `Markdown` 插件所提供的。

然而，“发布文章”的功能还要优化，目前成功发布一篇文章之后，仅仅是提示成功，页面并没有跳转。

4.发布文章后页面的跳转

文章发布之后，页面应该跳转到“文章列表”，这样就可以通过列表中的文章标题查看该文章的内容，当然也可以跳转到文章内容页，下面演示的是跳转到文章标题列表页面，读者可以在学习了下面的内容之后，自己实现跳转到该文章内容页的功能。

需要修改的仅仅是 `./templates/article/column/article_post.html` 中的 `JavaScript` 部分代码。

```
function publish_article() {
    var title = $('#id_title').val()
    var column_id = $('#which_column').val()
    var body = $('#id_body').val()
    $.ajax({
        url: '{% url "article:article_post" %}',
        type: 'POST',
        data: {
            title,
            body,
            column_id
        },
        success: function (e) {
            if (e === '1') {
                layer.msg('successful')
                location.href = '{% url "article:article_list" %}'
            } else if (e === '2') {
                layer.msg('sorry.')
            } else {
                layer.msg('项目名称必须写，不能空。')
            }
        }
    })
}
```

知识点

1.关于slug

`slug` 的目的在于为每条记录生成一个 `URL`，并且让这个 `URL` 更易读。比如一篇文章

的 `URL`，有的是<http://www.itdiffer.com/course/38>，这里使用文章在数据库中的 `id` (`id=38`)，这种方式虽然能够显示对应内容，但是不容易阅读；如果用类似<http://www.itdiffer.com/course/learn-djangowith-laoqi>的样式，那么这个 `URL` 的可读性就很好了。当然，不能仅仅如此，**因为文章标题有可能重复，最好是**<http://www.itdiffer.com/course/38/learn-django-with-laoqi>。这里的“`learn-djangowith-laoqi`”就是要保存在数据库中的 `slug`。为此，在数据模型的字段属性中有专门的一个属性 `models.SlugField()`，这个属性的源码读者可以阅读https://docs.djangoproject.com/en/1.10/_modules/django/db/models/fields/#SlugField，它其实也是 `CharField` 的子类。`Django` 这么安排，对发布文章的永久链接是非常友好的（不要忘记 `Django` 的起源）。

但是，上述的所有阐述是针对英文标题的，如果在中文环境中，则可以使用本节中的方法转换为汉语拼音。

2.模型：“一对多”

“一对多”这种说法翻译自“one-to-many”，不管是翻译还是原文，都容易造成一种误解，误以为有方向性，即误以为“一对多”和“多对一”是有区别的，**而真实的“一对多”没有方向性。**

在数据库中，为了建立两个表之间的“一对多”的关联，要使用外键。所谓外键（Foreign Key），是用于建立两个数据库表之间的链接的一个或多个字段。比如有一个数据库表 `A` 和用户数据库表 `User` 建立了外键关系，在表 `A` 中有一个名为 `user_id` 的字段，它就是表 `A` 的外键。当表 `A` 的数据被保存时，会自动在 `user_id` 记录中保存 `User` 表中用户的 `id`，用这种方式创建了数据库表 `A` 和用户表 `User` 之间的链接，也就明确了表 `A` 中的每条记录是属于哪一个用户的。

`Django` 的数据模型类中的某个字段的属性被设置为某个表的外键后，在数据库表中，以“**字段名**”+“**_id**”的方式命名一个记录，并保存关联数据库表中记录的 `id` 值（或者理解为关联数据模型类实例的 `id`）。下面是本节所创建的数据模型类 `ArticlePost` 对应的数据库表结构：

外键有几个常用的参数，能够为数据访问带来很多便利，下面仅列出两个。

- `limit_choices_to`，专门用于为字段设置选项，例如 `staff_member=models.ForeignKey(User,limit_choices_to={'is_staff':True})`（此示例引自官方文档）。

团子注：这个需要在想想，不是很明白。

- `related_name`，用于关联对象的反向查询，比如在 `ArticlePost` 类中的 `author=models.ForeignKey(User,related_name="article")`，可以通过 `User` 实例的 `article` 属性得到该用户的所有 `ArticlePost` 实例（某个用户的所有文章）。这种反向查询在本书后续项目中会经常用到，读者在实践中可以逐渐理解。如果在这里定义了 `related_name`，那么使用“`_set`”进行的查询就不能使用了，这一点请读者注意。

团子注：其实通过反向关联名获取到的是**模型管理器**，在此基础上，继续调用 `all()` 才能得到所有实例。

3.文档导读

(1) Many-to-one relationships,

https://docs.djangoproject.com/en/2.1/topics/db/examples/many_to_one/。

(2)

ForeignKey, <https://docs.djangoproject.com/en/2.1/ref/models/fields/#foreignkey>。

(3) Making queries, <https://docs.djangoproject.com/en/2.1/topics/db/queries/>。

3.3 删除和修改文章

网站上对文章的操作通常简称为“增删改查”：“增”即发布，“删”即删除，“改”即修改，“查”即查看。继上节内容，本节学习“删改”之法

删除

如同“栏目管理”中的功能，对文章也有删除和修改的操作。还是采用 `bootstrap` 默认的两个图标，先在 `./templates/article/column/article_list.html` 文件中的相应 `<td>` 内将该图标显示出来，代码如下。

```
<span class="glyphicon glyphicon-pencil"></span>
<span class="glyphicon glyphicon-trash" style="margin-left: 20px;"></span>
```

实现“删除”操作相对简单一些，这样容易使读者建立信心。

这次换一个开发顺序，不再先写视图函数了（当然数据模型和表单类都不用写了），而是“从前向后”先写“删除”的前端功能。因此，还要继续编辑上述模板文件，在“删除”功能的代码外围加上 `<a>` 标签。

```
<a href="javascript:;" name="delete" onclick="del_article(this,
    {{ article.id }})">
    <span class="glyphicon glyphicon-trash" style="margin-left: 20
px;"></span>
</a>
```

显然需要写一个提交要删除的文章 `id` 的 `JavaScript` 脚本。

```
<script>
    function del_article(the, article_id) {
        var article_name =
$(the).parents('tr').children('td').eq(1).text()
        layer.open({
            type: 1,
            skin: 'layui-layer-rim',
            area: ['400px', '200px'],
            title: '删除文章',
            content: '<div class="text-center" style="margin-top: 20p
x;">\
                <p>是否确认删除《' + article_name + '》</p>\
                </div>',
            btn: ['确定', '取消'],
            yes: function () {
                $.ajax({
                    url: '{% url "article:del_article" %}',
                    type: 'POST',
                    data: { article_id },
                    success: function(e) {
                        if (e === '1') {
                            parent.location.reload()
                            layer.msg('has been deleted')
                        } else {
                            layer.msg('删除失败')
                        }
                    }
                })
            }
        })
    }
</script>
```

```
    })  
    }  
</script>
```

前端设置完成后，就要编写删除文章的视图函数了，编辑 `./article/views.py` 文件，增加 `del_article()` 函数。

```
@login_required(login_url='/account/login/')  
@require_POST  
@csrf_exempt  
def del_article(request):  
    article_id = request.POST['article_id']  
    try:  
        article = ArticlePost.objects.get(id=article_id)  
        article.delete()  
        return HttpResponse('1')  
    except:  
        return HttpResponse('2')
```

别忘了，还要在 `./article/urls.py` 文件中对删除操作进行 URL 设置。

```
path('del-article/', views.del_article, name='del_article'),
```

修改

在编写修改文章的功能之前，先梳理清楚实现这个功能的流程。

- (1) 单击修改文章的图标，将该文章的 `id` 传给相应的视图函数。
- (2) 视图函数从数据库中读取该文章 `id` 的相关内容。
- (3) 将文章相关内容呈现在页面上，并且处于编辑状态，要求依然使用 `Markdown` 编辑器。
- (4) 用户编辑之后，单击“提交”按钮，再次将页面中编辑的内容保存在上述 `id` 所对应的数据库记录中。

从上述流程中可以看出，我们必须编写一个视图函数（依然在 `./article/views.py` 文件中），这个视图函数能够接收文章的 `id`，而且支持 `GET` 和 `POST` 两种请求方式。先编写满足 `GET` 请求的部分，代码如下。

```

@login_required(login_url='/account/login/')
@csrf_exempt
def redit_article(request, article_id):
    if request.method == 'GET':
        article = ArticlePost.objects.get(id=article_id)
        article_columns = request.user.article_column.all()
        this_article_form = ArticlePostForm(initial={'title': article.title})
        # 我试试我的方法，分两步走，能不能行。结果：不行!!!
        # this_article_form = ArticlePostForm()
        # this_article_form.title = article.title
        this_article_column = article.column
        return render(request, 'article/column/redit_article.html', {
            'article': article,
            'article_columns': article_columns,
            'this_article_column': this_article_column,
            'this_article_form': this_article_form,
        })

```

团子注：为啥把 `this_article_form` 单独给了一个 `title`，那 `body` 呢？

另外为啥不先 `this_article_form = ArticlePostForm()` 然后

`this_article_form.title = article.title`，是因为更简洁吗？

官方文档是这么说的：**Form.initial**, Use initial to declare the initial value of form fields at runtime. For example, you might want to fill in a username field with the username of the current session.

我来试试我想的办法能不能行。结果证明是不行的。。。

以后需要仔细研究一下 `dir(this_article_form)` 里面的东西，也就是要研究下

`Form` 实例里面的属性和方法。目前知道的是，直接

`print(this_article_form)` 得到的是一串 `HTML` 文本。

接着在 `./templates/article/column/` 目录中创建名为 `redit_article.html` 的文件，这就是上述视图函数所指定的前端模板文件，其代码如下。

```

{% extends "article/base.html" %}

{% load static %}

{% block title %}
article column
{% endblock title %}

```

```

{% block content %}
    <link rel="stylesheet" href="{% static 'editor/css/style.css'
    %}">
    <link rel="stylesheet" href="{% static 'editor/css/editormd.cs
s' %}">
    <div class="container">
        <div class="col-md-10">
            <div style="margin-left: 10px;">
                <form action="." class="form-horizontal" method="post">
                    {% csrf_token %}
                    <div class="row" style="margin-top: 10px;">
                        <div class="col-md-2 text-right"><span>标题: </span>
</div>
                        <div class="col-md-10 text-left">{{ this_article_for
m.title }}</div>
                    </div>
                    <div class="row" style="margin-top: 10px;">
                        <div class="col-md-2 text-right"><span>栏目: </span>
</div>
                        <div class="col-md-10 text-left">
                            <select id="which_column">
                                {% for column in article_columns %}
                                    {% if column == this_article_column.column %}
                                        <option value="{{ column.id }}" selected="se
lected">
                                            {{ column.column }}
                                        </option>
                                    {% else %}
                                        <option value="{{ column.id }}">
                                            {{ column.column }}
                                        </option>
                                    {% endif %}
                                {% endfor %}
                            </select>
                        </div>
                    </div>
                    <div class="row" style="margin-top: 10px;">
                        <div class="col-md-2 text-right"><span>内容: </span>
</div>
                        <div id="editormd" class="col-md-10 text-left">
                            <!-- {{ article_post_form.body }} -->
                            <textarea style="display:none;" id="id_body">
                                {{ article.body }}
                            </textarea>

```

```

        </div>
    </div>
    <div class="row">
        <input type="button" value="发布" class="btn btn-primary btn-lg" onclick="redit_article()">
    </div>
</form>
</div>
</div>
</div>

<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'editor/editormd.min.js' %}"></script>
<script src="{% static 'js/layer.js' %}"></script>
<script>
    $(function() {
        var editor = editormd('editormd', {
            width: '100%',
            height: 640,
            // syncScrolling: 'single',
            path: '{% static "editor/lib/" %}',
        })
    })
</script>
{% endblock content %}

```

在循环体内部要通过断当前文章原有的栏目名称，并将下拉菜单中的该名称确定为选定状态 `selected="selected"`。

下面配置 `URL`，在 `./article/urls.py` 文件中增加如下内容。

```

path('redit-article/<int:article_id>/', views.redit_article, name='redit_article'),

```

检查 `Django` 服务是否已经启动，在浏览器的地址栏中输入

<http://localhost:8000/article/redit-article/5/>，最后的数字就是某篇文章的 `id`，读者可以根据自已的项目输入其他数值，但不能省略，也不能输入不存在的 `id` 值

现在就可以在本页面中编辑文章了。

至此，完成了编辑文章的一部分功能。但是，总不能让用户记住自己每篇文章的id，所以要再做一个单击“铅笔”图标就能够跳转到上述界面的功能，需要再次对

`./templates/article/column/article_list.html` 文件进行编辑。

前面已经给“删除”图标加上了 `<a>` 标签，实现了删除功能。现在再给“编辑”图标加上 `<a>` 标签，实现编辑功能，代码如下。

```
<a href="{% url 'article:redit_article' article.id %}" name="edit">
    <span class="glyphicon glyphicon-pencil"></span>
</a>
```

`{% url 'article:redit_article' article.id %}` 这个写法较以往的 `URL` 写法稍有不同，后面多了一个参数 `article.id`，它对应 `URL` 中的 `article_id`。

完成之后，刷新<http://localhost:8000/article/article-list/>页面，单击“编辑”图标，检查是否跳转到上述编辑文章的界面。

就差最后一步了。在文章编辑界面单击“发布”按钮时，将数据提交给视图函数，还是用熟悉的 `Ajax` 方法。其实，在模板文件中已经有了伏笔

`onclick="redit_article()"`，在 `./templates/article/column/redit_article.html` 的尾部编写 `redit_article()` 函数，代码如下。

```
function redit_article() {
    var title = $('#id_title').val()
    var column_id = $('#which_column').val()
    var body = $('#id_body').val()
    $.ajax({
        url: '{% url "article:redit_article" article.id %}',
        type: 'POST',
        data: {
            title, body, column_id,
        },
        success: function(e) {
            if (e === '1') {
                layer.msg('successful')
                location.href = '{% url "article:article_list" %}'
            } else {
                layer.msg('sorry.')
            }
        }
    })
}
```


在 `POST` 对象地址中，使用了 `{% url 'article:redit_article' article.id %}`，这种形式的 `URL` 不仅可以用在 `HTML` 代码中，还可以用在 `JavaScript` 代码中。

团子注：从网友处得知，首先进行 `Django` 模板语言的解析，然后再按照 `HTML`、`JS`、`CSS` 解析。

前面 `./article/views.py` 中的视图函数 `redit_article()` 只编写了响应 `GET` 请求的部分，现在编写响应 `POST` 请求的部分，对此视图函数增加如下代码。

```
@login_required(login_url='/account/login/')
@csrf_exempt
def redit_article(request, article_id):
    if request.method == 'GET':
        article = ArticlePost.objects.get(id=article_id)
        article_columns = request.user.article_column.all()
        this_article_form = ArticlePostForm(initial={'title': article.title})
        # 我试试我的方法，分两步走，能不能行。结果：不行!!!
        # this_article_form = ArticlePostForm()
        # this_article_form.title = article.title
        print('=' * 30)
        print(this_article_form.fields)
        print(dir(this_article_form))
        print('=' * 30)
        this_article_column = article.column
        return render(request, 'article/column/redit_article.html', {
            'article': article,
            'article_columns': article_columns,
            'this_article_column': this_article_column,
            'this_article_form': this_article_form,
        })
    else:
        redit_article = ArticlePost.objects.get(id=article_id)
        try:
            redit_article.column = request.user.article_column.get(
                id=request.POST['column_id']
            )
            redit_article.title = request.POST['title']
            redit_article.body = request.POST['body']
            redit_article.save()
            return HttpResponse('1')
```

```
except:
    return HttpResponse('2')
```

试一下吧！从文章列表界面单击“编辑”图标，进入文章编辑界面，对文章进行修改，再单击“发布”按钮，看看是否实现了修改的目的。

设置分页功能

在文章标题列表中，应该再增加一个分页功能，因为在发布的文章多了之后，在同一个页面中显示出所有的标题是要耗费较长时间的，所以分页是必需的。

`Django` 也认为分页是常见的操作，所以提供了内置的分页方法，可以在视图函数 `./article/views.py` 的顶部引入分页功能用到的三个类。

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
```

下面重写 `article_list()` 视图函数，代码如下。

```
@login_required(login_url='/account/login/')
def article_list(request):
    articles_list = ArticlePost.objects.filter(author=request.user)
    paginator = Paginator(articles_list, 2)
    page = request.GET.get('page')
    try:
        current_page = paginator.page(page)
        articles = current_page.object_list
    except PageNotAnInteger:
        current_page = paginator.page(1)
        articles = current_page.object_list
    except EmptyPage:
        current_page = paginator.page(paginator.num_pages)
        articles = current_page.object_list
    return render(request, 'article/column/article_list.html', {
        'articles': articles, 'page': current_page
    })
```

`Paginator(articles_list, 2)` 根据所查询到的文章对象 `articles_list` 创建分页器实例对象，并且规定每页最多 `2` 个。`Paginator` 类的初始化的完整参数列表是

`Paginator(object_list, per_page, orphans=0, allow_empty_first_page=True)` ,
详细代码可以参考

<https://docs.djangoproject.com/en/2.1/topics/pagination/#django.core.paginator.Paginator>。

`page = request.GET.get('page')` 获得当前浏览器 `GET` 请求的参数 `page` 的值,
也就是当前浏览器所请求的页码数值。在后面的前端模板中, 我们会看到, 通过浏览器
发出的 `URL` 请求中都包含 `page` 参数, 即 `URL` 的样式诸如

`http://localhost:8000/article/article-list/?page=1`, 通过本句获得 `URL` 中的参数 `page`
的值 `1`, 即当前所请求的页码是 `1` (第 `1` 页), 并赋值给 `page` 变量。

`current_page = paginator.page(page)` 中的 `page()` 是 `Paginator` 对象的一个
方法, 其作用在于**得到指定页面内容**, 其参数必须是大于或等于 `1` 的整数。

团子注: `page` 从 `1` 开始。分页器对象的 `page(页码)` 方法返回一个页面 `Page`
对象。

`object_list` 是 `Page` 对象的属性, 能够得到该页所有的对象列表。类似的属性还有
`Page.number` (返回页码) 等。

团子注: 对象列表是 `object_list`, 所以按理讲, 老齐这里的文章列表应改为
`article_list` 而不是 `articles_list`。

另外注意, 页面对象的一些属性, 一个是获取对象列表 `p.object_list`, 还有一个
是获取页码 `p.number`。

`except PageNotAnInteger` 和 `except EmptyPage` 捕获两个异常, 一个是请求的页
码数值不是整数 (`PageNotAnInteger`), 另一个是请求的页码数值为空或者在 `URL` 参
数中没有 `page` (`EmptyPage`)。

`paginator.num_pages` 返回的是页数, `num_pages` 是 `Paginator` 对象的一个属性。

重写了视图函数之后, 还要再写一个专门显示分页的模板文件, 创建
`./templates/paginator.html` 文件, 并输入如下代码。

```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{ page.previous_page_number
    }}">Previous</a>
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}
```

```
</span>
{% if page.has_next %}
    <a href="?page={{ page.next_page_number }}">Next</a>
{% endif %}
</span>
</div>
```

在上述代码中，使用了 `Page` 对象的几个属性。

- `has_previous`：判断是否有上一页。
- `previous_page_number`：返回上一页的页码。
- `number`：返回当前页的页码。
- `paginator.num_pages`：关联 `Paginator` 对象（`page.paginator`），并得到其总页码数（`paginator.num_pages`）。
- `next_page_number`：返回下一页的页码。

团子注：这里 `href="?page={{ page.previous_page_number }}"` 直接拼接有点不明白，它是**替换了**之前的查询字符串吗？

另外，要做分页，直接传一个当前的页面对象即可，不用传递分页器对象。

完成分页模板的代码编写之后，将它引入到 `article_list.html` 文件中，编辑 `./templates/article/column/article_list.html` 文件，并在表格代码 `<table>` `</table>` 的后面加入如下内容。

```
{% include "paginator.html" %}
```

团子注：相当于单独编写一个**分页组件**，然后拼接，这个方法好！！

访问<http://localhost:8000/article/article-list/>，就会看到一个简明且全面的分页功能。

知识点

1.Markdown

在本项目中，编辑文章所用的编辑器是Markdown，没有用别的编辑器。Markdown编辑器是目前最好的编辑器之一。

感谢这个编辑器的创始人John Gruber。

关于Markdown的语法，可以查看<http://wowubuntu.com/markdown/>，或者访问官方网站<http://daringfireball.net/projects/markdown/>。

2.模型：插入和更新

SQL 语句中的 `INSERT` 和 `UPDATE` 可以实现数据库表中记录的插入和更新。在 ORM 中，要实现这种操作，可以使用完全类似 Python 中创建实例或者属性赋值的方式进行。

(1) 插入数据

上述代码等效于 SQL 语句中的 `INSERT`，最后要使用 `new_user` 实例的 `save()` 方法才能完成**数据的入库操作**。除这种方法外，还可以使用如下方式。

团子注：`User.objects.create` 是有返回值的！

`create()` 函数直接让数据入库。

(2) 修改数据

注意，`save()` 函数是必需的。

从上述操作可以看出，插入和更新操作不返回 `QuerySet` 结果，这与查询操作是有显著差别的。

团子注：查询操作也只有 `filter` 和 `exclude` 返回 `QuerySet`，而 `get` 是返回单个模型实例的。

3.模型：删除

ORM 中删除记录的方法非常简单。

团子注： `get` 必须要求返回实例，有且仅有一个，查找不到或者多于一个都会报错！而 `filter` 和 `exclude` 则没有此要求。

另外， `delete()` 的返回值是个啥？？？ 解答在下面。

`delete()` 删除后返回该记录（实例）的值。

4.文档导读

(1) jQuery Pagination plugin, <https://github.com/esimakin/twbs-pagination>。

(2) django-markdown, <https://pypi.python.org/pypi/django-markdown>。

(3) QuerySet API

reference, <https://docs.djangoproject.com/en/2.1/ref/models/queriesets/>。

3.4 文章展示

前文已经实现了用户注册、登录、发布文章，以及超级管理员的后台管理。下面要做的就是向访问网站的用户展示作者及其文章。

新写文章标题列表

前面已经编写过文章标题列表，这里要“新”写，肯定与前面的不同。不过，基本的逻辑还是一样的，不同之处在于展示效果。

首先要编写的是视图函数，这次要**新建一个编写视图函数的文件**

`./article/list_views.py`，这样做的目的主要是**让读者理解视图函数的文件不一定是 `views.py`**，然后创建一个展示文章标题的视图函数 `article_titles()`，其代码如下。

```
from django.core.paginator import EmptyPage, PageNotAnInteger, Paginator
from django.shortcuts import render

from .models import ArticleColumn, ArticlePost

def article_titles(request):
    articles_title = ArticlePost.objects.all()
```

```

paginator = Paginator(articles_title, 2)
page = request.GET.get('page')
try:
    current_page = paginator.page(page)
    articles = current_page.object_list
except PageNotAnInteger:
    current_page = paginator.page(1)
    articles = current_page.object_list
except EmptyPage:
    current_page = paginator.page(paginator.num_pages)
    articles = current_page.object_list

return render(request, 'article/list/article_titles.html', {
    'articles': articles,
    'page': current_page,
})

```

下面配置本应用的 `URL`，就是 `./article/urls.py` 文件，并输入如下代码。

```

path('list-article-titles/', list_views.article_titles, name='article_titles'),

```

因为视图函数 `article_titles()` 在 `list_views.py` 文件中，所以要导入 `from . import views, list_views`

接下来是编写前端模板文件。在 `./templates/article` 目录中创建 `list` 目录，再创建 `article_titles.html` 模板文件，即 `./templates/article/list/article_titles.html`。

```

{% extends "base.html" %}

{% block title %}
articles
{% endblock title %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>阅读，丰富头脑，善化行为</h1>
</div>
<div class="container">
    {% for article in articles %}
        <div class="list-group">

```

```

        <a href="#" class="list-group-item active">
            <h4 class="list-group-item-heading">{{ article.title }}
        </h4>
        <p class="list-group-item-text">作者: {{ article.author.u
        sername }}</p>
        <p class="list-group-item-text">概要: {{ article.body | s
        lice:'70' | linebreaks }}</p>
        </a>
    </div>
{% endfor %}
{% include "paginator.html" %}
</div>
{% endblock content %}

```

团子注：这个 `list-group-item` 类是什么样式的？是 `Bootstrap` 的类吗？

- `{{ article.author.username }}`：用于得到该文章作者的用户名。在 `./article/models.py` 中我们定义了 `ArticlePost` 数据模型类，其中属性 `author=models.ForeignKey(User,related_name="article")`，所以就能够以 `{{ article.author.username }}` 方式得到用户名。
- `{{ article.body | slice:'70' | linebreaks }}` 中的 `{{ article.body }}` 用于显示该文章对象的所有文章内容，但是后面用管道符“`|`”对显示的内容做了限制（有一种常见的说法叫作“过滤器”），“`slice:'70'`”的含义是将前面的变量所导入的内容“切下”前 70 个字符。“slice”是根据字符数量来截取部分内容的，与之类似的还有一个 `truncatewords`，读者不妨将变量更换为 `{{ article.body | truncatewords:'70' | linebreaks }}`，看看有什么效果。`silce` 是“切下”一定数量的字符，从内容的开头计算，只要是一个字符就算一个，空格也算一个字符；而 `truncatewords` 则是截取一定数量的 `words`（单词），在英语中 `words` 之间用空格分隔，所以**它会根据空格进行截取**，但是遇到中文就麻烦了，中文不用空格分开各个词语，除非分段的地方。所以，**如果读者按照笔者所说的更换了，就会看到对中文基本没有截取（读者的文章估计没有70个段落）**。为了测试出上面说的效果，读者不仅要发布几篇英文文章，也要发布几篇中文文章。
- 过滤器 `linebreaks`，作用是允许原文中的换行 `HTML` 标记符继续产生效用。

在浏览器的地址栏中输入<http://localhost:8000/article/list-article-titles/>地址，不管用户是否登录，都能看到下图：

列表完成后，下面要查看文章内容。

其实前面编写过查看文章内容功能，为了叙述完整，同时也是跟读者一起复习，下面快速地重复一遍。

在 `./article/list_views.py` 文件中增加如下视图函数（不要忘记在文件顶部引入 `from django.shortcuts import get_object_or_404`）。

```
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    return render(request, 'article/list/article_content.html',
                  {'article': article})
```

上述代码中的 `article_detail()` 函数与以往不同，它不检查用户是否处于登录状态。

还是熟悉的步骤，在 `./article/urls.py` 文件中增加如下 `URL` 配置代码。

```
path('article-content/<int:id>/<slug:slug>/', list_views.article_
    detail, name='article_content'),
```

在 `urls.py` 文件中，请读者将上面的 `URL` 和前面已经有的一个 `URL` 配置（`re_path(r'^articledetail/(?P<id>\d+)/(?P<slug>[-\w]+)/$', views.article_detail, name="article_detail")`，）进行对比。

两个 `URL` 配置因为使用了不同的函数（`path()` 和 `re_path()`），所以在写法上就有了很大的区别。另外，虽然都有 `article_detail()` 函数，但因为在不同模块（`views / list_views`）中，所以两个函数互不影响。注意，`name` 的值要区分开，在同一个 `URL` 配置文件中不要出现相同的 `name`。

在 `./templates/article/list` 中创建模板文件 `article_content.html`，代码如下。

```
{% extends "base.html" %}

{% load static %}

{% block title %}
articles detail
{% endblock title %}

{% block content %}
<div class="container">
    <header>
        <h1>{{ article.title }}</h1>
```

```

    <p>{{ article.author.username }}</p>
</header>

<link rel="stylesheet" href="{% static 'editor/css/editormd.pr
view.css' %}">
<div id="editormd-view">
    <textarea id="append-test" style="display: none;">
{{ article.body }}
    </textarea>
</div>
</div>
<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'editor/lib/marked.min.js' %}"></script>
<script src="{% static 'editor/lib/prettify.min.js' %}">
</script>
<script src="{% static 'editor/lib/raphael.min.js' %}"></script>
<script src="{% static 'editor/lib/underscore.min.js' %}"></scrip
t>
<script src="{% static 'editor/lib/sequence-diagram.min.js' %}">
</script>
<script src="{% static 'editor/lib/flowchart.min.js' %}"></scrip
t>
<script src="{% static 'editor/lib/jquery.flowchart.min.js' %}">
</script>
<script src="{% static 'editor/editormd.js' %}"></script>

<script>
$(function() {
    editormd.markdownToHTML('editormd-view', {
        htmlDecode: 'style,script,iframe', // you can filter tags d
ecode
        emoji: true,
        taskList: true,
        tex: true, // 默认不解析
        flowChart: true, // 默认不解析
        sequenceDiagram: true, // 默认不解析
    })
})
</script>
{% endblock content %}

```

再编辑 `./templates/article/list/article_titles.html` 模板文件，在文章标题外面的 `<a>` 标签中增加超链接对象，代码如下。

```
<a href="{{ article.get_absolute_url }}" class="list-group-item
active">
...
</a>
```

一切都与以往一样，没有什么特别的。如果现在处于登录状态，请退出，否则测试不到下述现象。对于浏览文章，我们约定任何人都可以。

访问<http://localhost:8000/article/list-article-titles/>页面，把鼠标移到文章标题上，先不要单击，看浏览器底端，一般会有对象地址，显示的超链接地址是不是预想的那个呢？应该不是。根据前面的 `URL` 配置，文章详细内容的地址应该是符合类似“<http://localhost:8000/article/article-content/2/learn-python>”的样式，以符合 `URL` 配置中“`article-content/<int:id>/<slug:slug>/`”的要求，而不是<http://localhost:8000/article/article-detail/4/how-to-learn-program/>的样式，这种样式是用户发布文章后查看详情的链接地址。并且，如果此时点击页面中的文章标题，会跳转到登录窗口。

其实一开始的 <http://localhost:8000/article/article-content/2/learn-python> 是后台管理页面，而不是前台展示页面。我们现在需要拿到的是前台展示页面的 `URL`。

导致上述现象的原因是在 `./templates/article/list/article_titles.html` 文件中使用了 `{{ article.get_absolute_url }}`，而深藏在 `./article/models.py` 中的 `ArticlePost` 类的 `get_absolute_url()` 方法中有 `reverse("article:article_detail",args=[self.id,self.slug])`，“`article:article_detail`”导致了上述跳转对象。

重新编写“查看文章”功能

直接套用时遇到了问题，不能再使用 `{{ article.get_absolute_url }}`。

如何修改呢？

既然问题的根源是 `ArticlePost` 类中的 `get_absolute_url()` 方法，那么就可以弃之不用，再写一个方法。将如下方法追加到 `./article/models.py` 的 `ArticlePost` 类中。

```
def get_url_path(self):
    return reverse('article:article_content', args=[self.id, self.slug])
```

将 `./article/list/article_titles.html` 中的超链接修改为：

```
<a href="{{ article.get_url_path }}" class="list-group-item active">
</a>
```

再次访问文章列表页<http://localhost:8000/article/list-article-titles/>，单击文章标题，这时候应该能够看到文章详细内容了。

貌似到这里就已经能够实现所有功能了。但是，这种方法其实存在一个隐患。在文章的 URL 中，我们使用的是数据库自增的 `id`，这就是隐患，如果因为某种原因，自增 `id` 发生了变化，比如合并数据库表（可能是低概率，就怕万一），这时候再按照原来的 URL 访问，就不能找到那篇文章了。就一般情况而言，网上发布的文章，应该具有一个永久的 URL，所以，应该给每篇文章增加一个单独的固定 `id`（或者别的标识），通过这个固定的 `id`（标识）访问该文章。

团子注：怎么加永久 URL ？？？

知识点

1.模板：过滤器

Django 模板中的过滤器，常常根据需要在模板中显示变量形式时使用。其基本样式是 `{{ value | filter }}`，其中 `filter` 是过滤器。Django 模板提供了多种针对变量数值的过滤器。

- `capfirst`，将变量的第一个字符转换为大写，如果第一个字符不是字母则过滤器失效。例如 `{{ value | capfirst }}`，如果 `value` 是“django”，则输出“Django”。
- `cut`，移除变量中的字符。例如 `{{ value | cut:" " }}`，如果 `value` 是“learn django”，则输出“learndjango”。
- `date`，根据指定的格式输出时间。例如 `{{ value | date:"D d M Y" }}`，如果 `value` 是 `datetime` 对象的实例（比如常用的 `datetime.datetime.now()`），则输出类似于“Web 09 Jan 2008”样式的字符串。官方文档中列出了各种格式字符的说明，读者可参阅。
- `join`，使用字符连接列表元素，其效果类似 Python 中的 `str.join(list)`。例如 `{{ value | join:"/" }}`，如果 `value` 是 `['a', 'b', 'c']`，则会输出“a//b//c”。

以上内容是根据官方文档的说明整理出来的，在官方文档中还有很多，请读者在用到的时候去查阅。

这些都是内置的过滤器，在某些时候还需要自定义。本书后面的项目中也会演示如何自定义一个过滤器。

2. 表单：表单类

在模板中，由 `<form></form>` 组成表单，里面是一个一个的 `<input>`，这个表单通过 `<form>` 规定的方式提交到 Django 的视图部分，通常使用 `POST` 或者 `GET` 方式。本书中的项目都使用了 `POST`。视图得到提交来的表单数据之后，在处理这些表单数据时，可以使用一个与数据模型（Model）类类似的类，即Form（表单）类。当然，不用也可以，但用了更简洁。

表单类也是一个普通的 Python 类，这个类里面定义了一些变量（类的属性），这些变量分别对应着 HTML 中的 `<input>`。我们知道，在 `<input>` 中有不同的类型，从而规定是文本输入框还是按钮等。在表单类中，也是通过类似在数据模型类中声明字段类型的方式，规定了变量的类型。

通过前面的编程实践，我们已经认识到，数据模型类中的字段及其类型、表单类中的变量及其类型、模板中的 `<input>` 及其类型，都是一一对应的，否则数据无法正确保存。因此，如果没有特别需要，在表单类中可以直接使用内部类 `class Meta` 声明与数据模型类中对应的变量。例如：

在内部类 `Meta` 中，以 `model=UserInfo` 说明表单中各个字段的来源，然后用 `fields` 变量说明在本表单类中用到的字段，如果选择所有字段，则可以使用 `fields="__all__"`，还可以使用 `exclude=["school"]` 的方式排除某些字段。

3. 文档导读

- (1) Working with forms, <https://docs.djangoproject.com/en/2.1/topics/forms/>。
- (2) Built-in template tags and filters, <https://docs.djangoproject.com/en/2.1/ref/templates/builtins/>。
- (3) django.urls utility functions, <https://docs.djangoproject.com/en/2.1/ref/urlresolvers/>。