

首先，我们需要将配置变量 WTF_I18N_ENABLED 设为 False，这会让 Flask-WTF 使用 WTForms 内置的错误消息翻译。然后我们需要在自定义基类中定义 Meta 类，并在 locales 列表中加入简体中文的地区字符串。在创建表单时，继承这个 MyBaseForm 即可将错误消息语言设为中文，比如上面定义的 HelloForm。另外，你也可以在实例化表单类时通过 meta 关键字传入 locales 值，比如：

```
form = MyForm(meta={'locales': ['en_US', 'en']})
```

 **附注** locales 属性是一个根据优先级排列的地区字符串列表。在 WTForms 中，简体中文和繁体中文的地区字符串分别为 zh 和 zh_TW。

在本书的第二部分，我们将学习为 Flask 程序添加国际化和本地化支持，这样程序会根据用户的语言偏好来自动显示正确的语言，而不是固定使用某一种语言。

4.4.2 使用宏渲染表单

在模板中渲染表单时，我们有大量的工作要做：

- 调用字段属性，获取 <input> 定义。
- 调用对应的 label 属性，获取 <label> 定义。
- 渲染错误消息。

为了避免为每一个字段重复这些代码，我们可以创建一个宏来渲染表单字段，如代码清单 4-9 所示。

代码清单4-9 macros.html：表单渲染宏

```
{% macro form_field(field) %}
    {{ field.label }}<br>
    {{ field(**kwargs) }}<br>
    {% if field.errors %}
        {% for error in field.errors %}
            <small class="error">{{ error }}</small><br>
        {% endfor %}
    {% endif %}
{% endmacro %}
```

这个 form_field() 宏接收表单类实例的字段属性和附加的关键字参数作为输入，返回包含 <label> 标签、表单字段、错误消息列表的 HTML 表单字段代码。使用这个宏渲染表单的示例如下所示：

```
{% from 'macros.html' import form_field %}
...
<form method="post">
    {{ form.csrf_token }}
    {{ form_field(form.username) }}<br>
    {{ form_field(form.password) }}<br>
    ...
</form>
```



在上面的代码中，我们调用 `form_field()` 宏逐个渲染表单中的字段，只要把每一个类属性传入 `form_field()` 宏，即可完成渲染。

 **提示** 同样的，我们可以编写一个宏渲染 Bootstrap 风格的表单。不过，这类复杂的工作可以交给扩展来完成，后面我们会介绍使用扩展简化在模板中渲染 Bootstrap 风格表单的工作。

4.4.3 自定义验证器

在 WTForms 中，验证器是指在定义字段时传入 `validators` 参数列表的可调用对象，这一节我们会介绍如何编写自定义验证器。

1. 行内验证器

除了使用 WTForms 提供的验证器来验证表单字段，我们还可以在表单类中定义方法来验证特定字段，如代码清单 4-10 所示。

代码清单4-10 form/forms.py：针对特定字段的验证器

```
from wtforms import IntegerField, SubmitField
from wtforms.validators import ValidationError

class FortyTwoForm(FlaskForm):
    answer = IntegerField('The Number')
    submit = SubmitField()

    def validate_answer(form, field):
        if field.data != 42:
            raise ValidationError('Must be 42.')
```

当表单类中包含以“`validate_` 字段属性名”形式命名的方法时，在验证字段数据时会同时调用这个方法来验证对应的字段，这也是为什么表单类的字段属性名不能以 `validate` 开头。验证方法接收两个位置参数，依次为 `form` 和 `field`，前者为表单类实例，后者是字段对象，我们可以通过 `field.data` 获取字段数据，这两个参数将在验证表单时被调用传入。验证出错时抛出从 `wtforms.validators` 模块导入的 `ValidationError` 异常，传入错误消息作为参数。因为这种方法仅用来验证特定的表单类字段，所以又称为行内验证器（in-line validator）。

2. 全局验证器

如果你想要创建一个可重用的通用验证器，可以通过定义一个函数实现。如果不需要传入参数定义验证器，那么一个和表单类中定义的验证方法完全相同的函数就足够了，如代码清单 4-11 所示。

代码清单4-11 全局验证器示例

```
from wtforms.validators import ValidationError

def is_42(form, field):
```



```

if field.data != 42:
    raise ValidationError('Must be 42')

class FortyTwoForm(FlaskForm):
    answer = IntegerField('The Number', validators=[is_42])
    submit = SubmitField()

```

当使用函数定义全局的验证器时，我们需要在定义字段时在 validators 列表里传入这个验证器。因为在 validators 列表中传入的验证器必须是可调用对象，所以这里传入了函数对象，而不是函数调用。

这仅仅是一个简单的示例，在现实中，我们通常需要让验证器支持传入参数来对验证过程进行设置。至少，我们应该支持 message 参数来设置自定义错误消息。这时验证函数应该实现成工厂函数，即返回一个可调用对象的函数，如代码清单 4-12 所示。

代码清单 4-12 工厂函数形式的全局验证器示例

```

from wtforms.validators import ValidationError

def is_42(message=None):
    if message is None:
        message = 'Must be 42.'

    def _is_42(form, field):
        if field.data != 42:
            raise ValidationError(message)

    return _is_42

class FortyTwoForm(FlaskForm):
    answer = IntegerField('The Number', validators=[is_42()])
    submit = SubmitField()

```

在现在的 is_42() 函数中，我们创建了另一个 _is_42() 函数，这个函数会被作为可调用对象返回。is_42() 函数接收的 message 参数用来传入自定义错误消息，默认为 None，如果没有设置就使用内置消息。在 validators 列表中，这时需要传入的是对工厂函数 is_42() 的调用。



附注 在更复杂的验证场景下，你可以使用实现了 __call__() 方法的类（可调用类）来编写验证器，具体请参考 WTForms 文档相关章节 (<http://wtforms.readthedocs.io/en/latest/validators.html#custom-validators>)。

4.4.4 文件上传

在 HTML 中，渲染一个文件上传字段只需要将 <input> 标签的 type 属性设为 file，即 <input type="file">。这会在浏览器中渲染成一个文件上传字段，单击文件选择按钮会打开文件选择窗口，选择对应的文件后，被选择的文件名会显示在文件选择按钮旁边。

在服务器端，可以和普通数据一样获取上传文件数据并保存。不过我们需要考虑安全问题，文件上传漏洞也是比较流行的攻击方式。除了常规的 CSRF 防范，我们还需要重点注意下面的问题：



- 验证文件类型。
- 验证文件大小。
- 过滤文件名。

1. 定义上传表单

在 Python 表单类中创建文件上传字段时，我们使用扩展 Flask-WTF 提供的 FileField 类，它继承 WTForms 提供的上传字段 FileField，添加了对 Flask 的集成。代码清单 4-13 创建了一个包含文件上传字段的表单。

代码清单4-13 form/forms.py：创建上传表单

```
from flask_wtf import FileField, FileRequired, FileAllowed
class UploadForm(FlaskForm):
    photo = FileField('Upload Image', validators=[FileRequired(),
        FileAllowed(['jpg', 'jpeg', 'png', 'gif'])])
    submit = SubmitField()
```

为了便于测试，我们创建一个用来上传图片的 photo 字段。和其他字段类似，我们也需要对文件上传字段进行验证。Flask-WTF 在 flask_wtf.file 模块下提供了两个文件相关的验证器，用法说明如表 4-5 所示。

表 4-5 Flask-WTF 提供的上传文件验证器

验证器	说 明
FileRequired(message=None)	验证是否包含文件对象
FileAllowed(upload_set, message=None)	用来验证文件类型，upload_set 参数用来传入包含允许的文件后缀名列表

我们使用 FileRequired 确保提交的表单字段中包含文件数据。出于安全考虑，我们必须对上传的文件类型进行限制。如果用户可以上传 HTML 文件，而且我们同时提供了视图函数获取上传后的文件，那么很容易导致 XSS 攻击。我们使用 FileAllowed 设置允许的文件类型，传入一个包含允许文件类型的后缀名列表。

顺便说一下，Flask-WTF 提供的 FileAllowed 是在服务器端验证上传文件，使用 HTML5 中的 accept 属性也可以在客户端实现简单的类型过滤。这个属性接收 MIME 类型字符串或文件格式后缀，多个值之间使用逗号分隔，比如：

```
<input type="file" id="profile_pic" name="profile_pic"
       accept=".jpg, .jpeg, .png, .gif">
```

当用户单击文件选择按钮后，打开的文件选择窗口会默认将 accept 属性值之外的文件过滤掉。尽管如此，用户还是可以选择设定之外的文件，所以我们仍然需要进行服务器端验证。



附注 扩展 Flask-Upserts (<https://github.com/maxcountryman/flask-uploads>) 内置了在 Flask 中实现文件上传的便利功能。Flask-WTF 提供的 FileAllowed() 也支持传入 Flask-Upserts 中的上传集对象（Upload Set）作为 upload_set 参数的值。另外，同类的扩展还有 Flask-Transfer (<https://github.com/justanr/Flask-Transfer>)。



除了验证文件的类型，我们通常还需要对文件大小进行验证，你肯定不想让用户上传超大的文件来拖垮你的服务器。通过设置 Flask 内置的配置变量 MAX_CONTENT_LENGTH，我们可以限制请求报文的最大长度，单位为字节（byte）。比如，下面将最大长度限制为 3M：

```
app.config['MAX_CONTENT_LENGTH'] = 3 * 1024 * 1024
```

当请求数据（上传文件大小）超过这个限制后，会返回 413 错误响应（Request Entity Too Large），如图 4-6 所示。

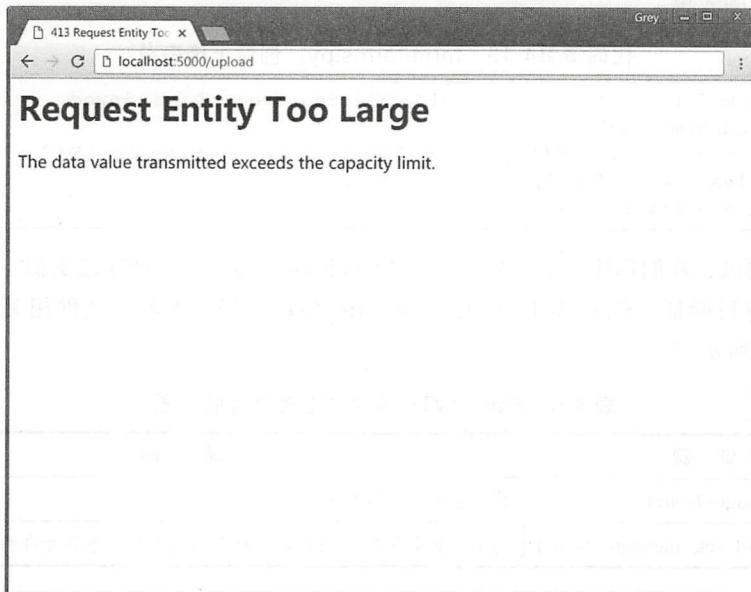


图 4-6 413 错误响应



我们可以创建对应的错误处理函数来返回自定义的 413 错误响应。需要注意，Flask 内置的开发服务器在抛出对应的异常时不会返回 413 响应，而是中断连接。不过我们不用担心这个问题，当使用生产环境下的服务器时，会正确返回 413 错误响应。

2. 渲染上传表单

在新创建的 upload 视图里，我们实例化表单类 UploadForm，然后传入模板：

```
@app.route('/upload', methods=['GET', 'POST'])
def upload():
    form = UploadForm()
    ...
    return render_template('upload.html', form=form)
```

代码清单 4-14 在模板中渲染了这个表单，渲染方式和其他字段相同。

代码清单4-14 form/templates/upload.html：在模板中渲染上传表单

```
<form method="post" enctype="multipart/form-data">
```



```

{{ form.csrf_token }}
{{ form_field(form.photo) }}
{{ form.submit }}
</form>

```

唯一需要注意的是，当表单中包含文件上传字段时（即 type 属性为 file 的 input 标签），需要将表单的 enctype 属性设为 "multipart/form-data"，这会告诉浏览器将上传数据发送到服务器，否则仅会把文件名作为表单数据提交。

3. 处理上传文件

和普通的表单数据不同，当包含上传文件字段的表单提交后，上传的文件需要在请求对象的 files 属性（request.files）中获取。我们在第 2 章介绍过，这个属性是 Werkzeug 提供的 ImmutableMultiDict 字典对象，存储字段的 name 键值和文件对象的映射，比如：

```
ImmutableMultiDict([('photo', <FileStorage: u'0f913b0ff95.JPG' ('image/jpeg')>)])
```

上传的文件会被 Flask 解析为 Werkzeug 中的 FileStorage 对象（werkzeug.datastructures.FileStorage）。当手动处理时，我们需要使用文件上传字段的 name 属性值作为键获取对应的文件对象。比如：

```
request.files.get('photo')
```

当使用 Flask-WTF 时，它会自动帮我们获取对应的文件对象，这里我们仍然使用表单类属性的 data 属性获取上传文件。处理上传表单提交请求的 upload 视图如代码清单 4-15 所示。

代码清单4-15 form/app.py：处理上传文件

```

import os

app.config['UPLOAD_PATH'] = os.path.join(app.root_path, 'uploads')

@app.route('/upload', methods=['GET', 'POST'])
def upload():
    form = UploadForm()
    if form.validate_on_submit():
        f = form.photo.data
        filename = random_filename(f.filename)
        f.save(os.path.join(app.config['UPLOAD_PATH'], filename))
        flash('Upload success.')
        session['filenames'] = [filename]
        return redirect(url_for('show_images'))
    return render_template('upload.html', form=form)

```

当表单通过验证后，我们通过 form.photo.data 获取存储上传文件的 FileStorage 对象。接下来，我们需要处理文件名，通常有三种处理方式：

(1) 使用原文件名

如果能够确定文件的来源安全，可以直接使用原文件名，通过 FileStorage 对象的 filename 属性获取：

```
filename = f.filename
```



(2) 使用过滤后的文件名

如果要支持用户上传文件，我们必须对文件名进行处理，因为攻击者可能会在文件名中加入恶意路径。比如，如果恶意用户在文件名中加入表示上级目录的 .. (比如 ../../../.home/username/.bashrc 或 ../../../.etc/passwd)，那么当我们保存文件时，如果这里表示上级目录的 .. 数量正确，就会导致服务器上的系统文件被覆盖或篡改，还有可能执行恶意脚本。我们可以使用 Werkzeug 提供的 `secure_filename()` 函数对文件名进行过滤，传递文件名作为参数，它会过滤掉所有危险字符，返回“安全的文件名”，如下所示：

```
>>> from werkzeug import secure_filename
>>> secure_filename('avatar!@#//#\%\^&.jpg')
'avatar.jpg'
>>> secure_filename('avatar头像.jpg')
'avatar.jpg'
```

(3) 统一重命名

`secure_filename()` 函数非常方便，它会过滤掉文件名中的非 ASCII 字符。但如果文件名完全由非 ASCII 字符组成，那么会得到一个空文件名：

```
>>> secure_filename('头像.jpg')
'jpg'
```

为了避免出现这种情况，更好的做法是使用统一的处理方式对所有上传的文件重新命名。随机文件名有很多种方式可以生成，下面是一个使用 Python 内置的 `uuid` 模块生成随机文件名的 `random_filename()` 函数：

```
def random_filename(filename):
    ext = os.path.splitext(filename)[1]
    new_filename = uuid.uuid4().hex + ext
    return new_filename
```

这个函数接收原文件名作为参数，使用内置的 `uuid` 模块中的 `uuid4()` 方法生成新的文件名，并使用 `hex` 属性获取十六进制字符串，最后返回包含后缀的新文件名。



附注 UUID (Universally Unique Identifier，通用唯一识别码) 是用来标识信息的 128 位数字，比如用作数据库表的主键。使用标准方法生成的 UUID 出现重复的可能性接近 0。在 UUID 的标准中，UUID 分为 5 个版本，每个版本使用不同的生成方法并且适用于不同的场景。我们使用的 `uuid4()` 方法对应的是第 4 个版本：不接收参数而生成随机 UUID。

在 `upload` 视图中，我们调用这个函数来获取随机文件名，传入原文件名作为参数：

```
filename = random_filename(f.filename)
```

处理完文件名后，是时候将文件保存到文件系统中了。我们在 `form` 目录下创建了一个 `uploads` 文件夹，用于保存上传后的文件。指向这个文件夹的绝对路径存储在自定义配置变量 `UPLOAD_PATH` 中：

```
app.config['UPLOAD_PATH'] = os.path.join(app.root_path, 'uploads')
```



这里的路径通过 `app.root_path` 属性构造，它存储了程序实例所在脚本的绝对路径，相当于 `os.path.abspath(os.path.dirname(__file__))`。为了保存文件，你需要提前手动创建这个文件夹。

对 `FileStorage` 对象调用 `save()` 方法即可保存，传入包含目标文件夹绝对路径和文件名在内的完整保存路径：

```
f.save(os.path.join(app.config['UPLOAD_PATH'], filename))
```

文件保存后，我们希望能够显示上传后的图片。为了让上传后的文件能够通过 URL 获取，我们还需要创建一个视图函数来返回上传后的文件，如下所示：

```
@app.route('/uploads/<path:filename>')
def get_file(filename):
    return send_from_directory(app.config['UPLOAD_PATH'], filename)
```

这个视图的作用与 Flask 内置的 `static` 视图类似，通过传入的文件路径返回对应的静态文件。在这个 `uploads` 视图中，我们使用 Flask 提供的 `send_from_directory()` 函数来获取文件，传入文件的路径和文件名作为参数。



提示 在 `get_file` 视图的 URL 规则中，`filename` 变量使用了 `path` 转换器以支持传入包含斜线的路径字符串。

在 `upload` 视图里保存文件后，我们使用 `flash()` 发送一个提示，将文件名保存到 `session` 中，最后重定向到 `show_images` 视图。`show_images` 视图返回的 `uploaded.html` 模板中将从 `session` 获得文件名，渲染出上传后的图片。

```
flash('Upload success.')
session['filenames'] = [filename]
return redirect(url_for('show_images'))
```



提示 这里将 `filename` 作为列表传入 `session` 只是为了兼容下面的多文件上传示例，这两个视图使用同一个模板，使用 `session` 可以在模板中统一从 `session` 获得文件名列表。

在 `uploaded.html` 模板里，我们将传入的文件名作为 URL 变量，通过上面的 `get_file` 视图获取文件 URL，作为 `` 标签的 `src` 属性值，如下所示：

```

```



创建表单类时，可以直接使用 WTForms 提供的 MultipleFileField 字段实现，添加一个 DataRequired 验证器来确保包含文件：

```
from wtforms import MultipleFileField
class MultiUploadForm(FlaskForm):
    photo = MultipleFileField('Upload Image', validators=[DataRequired()])
    submit = SubmitField()
```

表单提交时，在服务器端的程序中，对 request.files 属性调用 getlist() 方法并传入字段的 name 属性值会返回包含所有上传文件对象的列表。在 multi_upload 视图中，我们迭代这个列表，然后逐一对文件进行处理，如代码清单 4-16 所示。

代码清单4-16 form/app.py：处理多文件上传

```
from flask import request, session, flash, redirect, url_for
from flask_wtf.csrf import validate_csrf
from wtforms import ValidationError

@app.route('/multi-upload', methods=['GET', 'POST'])
def multi_upload():
    form = MultiUploadForm()
    if request.method == 'POST':
        filenames = []
        # 验证CSRF令牌
        try:
            validate_csrf(form.csrf_token.data)
        except ValidationError:
            flash('CSRF token error.')
            return redirect(url_for('multi_upload'))
        # 检查文件是否存在
        if 'photo' not in request.files:
            flash('This field is required.')
            return redirect(url_for('multi_upload'))

        for f in request.files.getlist('photo'):
            # 检查文件类型
            if f and allowed_file(f.filename):
                filename = random_filename(f.filename)
                f.save(os.path.join(
                    app.config['UPLOAD_PATH'], filename
                ))
                filenames.append(filename)
            else:
                flash('Invalid file type.')
                return redirect(url_for('multi_upload'))
        flash('Upload success.')
        session['filenames'] = filenames
        return redirect(url_for('show_images'))
    return render_template('upload.html', form=form)
```

在请求方法为 POST 时，我们对上传数据进行手动验证，主要包含下面几步：

- 1) 手动调用 flask_wtf.csrf.validate_csrf 验证 CSRF 令牌，传入表单中 csrf_token 隐藏字段的值。如果抛出 wtforms.ValidationError 异常则表明验证未通过。

2) 其中 if 'photo' not in request.files 用来确保字段中包含文件数据（相当于 FileRequired 验证器），如果用户没有选择文件就提交表单则 request.files 将为空。

3) if f 用来确保文件对象存在，这里也可以检查 f 是否是 FileStorage 实例。

4) allowed_file(f.filename) 调用了 allowed_file() 函数，传入文件名。这个函数相当于 FileAllowed 验证器，用来验证文件类型，返回布尔值，如代码清单 4-17 所示。

代码清单4-17 form/app.py：验证文件类型

```
app.config['ALLOWED_EXTENSIONS'] = ['png', 'jpg', 'jpeg', 'gif']
...
def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in app.config['ALLOWED_EXTENSIONS']
```

在上面的几个验证语句里，如果没有通过验证，我们使用 flash() 函数显示错误消息，然后重定向到 multi_upload 视图。

为了方便测试，我们还创建了一个临时的 filenames 列表，保存上传后的文件名到 session 中。访问 <http://localhost:5000/multi-upload> 打开多文件上传示例，单击按钮后可以选择多个文件，当上传的文件通过验证时，程序会重定向到 show_images 视图，这个视图返回的 uploaded.html 模板中将从 session 获取所有文件名，渲染出所有上传后的图片。

 提示 顺便说一句，在新版本的 Flask-WTF 发布后，你就可以使用和单文件上传相同的方式处理表单。比如，我们可以使用 Flask-WTF 提供的 MultipleFileField 来创建提供 Flask 支持的多文件上传字段，使用相应的验证器对文件进行验证。在视图函数中，我们则可以继续使用 form.validate_on_submit() 来验证表单，并通过 form.photo.data 来获取字段的数据——包含所有上传文件对象 (werkzeug.datastructures.FileStorage) 的列表。

 附注 多文件上传处理通常会使用 JavaScript 库在客户端进行预验证，并添加进度条来优化用户体验，具体我们会在本书的第二部分学习。

4.4.5 使用 Flask-CKEditor 集成富文本编辑器

富文本编辑器即 WYSIWYG (What You See Is What You Get，所见即所得) 编辑器，类似于我们经常使用的文本编辑软件。它提供一系列按钮和下拉列表来为文本设置格式，编辑状态的文本样式即最终呈现出来的样式。在 Web 程序中，这种编辑器也称为 HTML 富文本编辑器，因为它使用 HTML 标签来为文本定义样式。

CKEditor (<http://ckeditor.com/>) 是一个开源的富文本编辑器，它包含丰富的配置选项，而且有大量第三方插件支持。扩展 Flask-CKEditor 简化了在 Flask 程序中使用 CKEditor 的过程，我们将使用它来集成 CKEditor。首先使用 Pipenv 安装：

```
$ pipenv install flask-ckeditor
```

然后实例化 Flask-CKEditor 提供的 CKEditor 类，传入程序实例：

```
from flask_ckeditor import CKEditor
ckeditor = CKEditor(app)
```

1. 配置富文本编辑器

Flask-CKEditor 提供了许多配置变量来对编辑器进行设置，常用的配置及其说明如表 4-6 所示。

表 4-6 Flask-CKEditor 常用配置

配 置 键	默 认 值	说 明
CKEDITOR_SERVE_LOCAL	False	设为 True 会使用内置的本地资源
CKEDITOR_PKG_TYPE	'standard'	CKEditor 包类型，可选值为 basic、standard 和 full
CKEDITOR_LANGUAGE	" "	界面语言，传入 ISO 639 格式的语言码
CKEDITOR_HEIGHT	" "	编辑器高度
CKEDITOR_WIDTH	" "	编辑器宽度

 **附注** 完整的可用配置列表见 Flask-CKEditor 文档的配置部分：<https://flask-ckeditor.readthedocs.io/en/latest/configuration.html>

在示例程序中，为了方便开发，使用了内置的本地资源：

```
app.config['CKEDITOR_SERVE_LOCAL'] = True
```

 **提示** CKEDITOR_SERVE_LOCAL 和 CKEDITOR_PKG_TYPE 配置变量仅限于使用 Flask-CKEditor 提供的方法加载资源时有效，手动引入资源时可以忽略。

配置变量 CKEDITOR_LANGUAGE 用来固定界面的显示语言（简体中文和繁体中文对应的配置分别为 zh-cn 和 zh），如果不设置，默认 CKEditor 会自动探测用户浏览器的语言偏好，然后匹配对应的语言，匹配失败则默认使用英文。

Flask-CKEditor 内置了对常用第三方 CKEditor 插件的支持，你可以轻松地为编辑器添加图片上传与插入、插入语法高亮代码片段、Markdown 编辑模式等功能，具体可以访问 Flask-CKEditor 文档的插件集成部分 (<https://flask-ckeditor.readthedocs.io/en/latest/plugins.html>)。要使用这些功能，需要在 CKEditor 包中安装对应的插件，Flask-CKEditor 内置的资源已经包含了这些插件，你可以通过 Flask-CKEditor 提供的示例程序 (<https://github.com/greyli/flask-ckeditor/tree/master/examples>) 来了解这些功能的具体实现。

2. 渲染富文本编辑器

富文本编辑器在 HTML 中通过文本区域字段表示，即 <textarea></textarea>。Flask-CKEditor 通过包装 WTForms 提供的 TextAreaField 字段类型实现了一个 CKEditorField 字段类，我们使用它来构建富文本编辑框字段。代码清单 4-18 中的 RichTextForm 表单包含了一个标题字

段和一个正文字段。

代码清单4-18 form/forms.py：文章表单

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired, Length
from flask_ckeditor import CKEditorField # 从flask_ckeditor包导入

class RichTextForm(FlaskForm):
    title = StringField('Title', validators=[DataRequired(), Length(1, 50)])
    body = CKEditorField('Body', validators=[DataRequired()])
    submit = SubmitField('Publish')
```

文章正文字段（body）使用的 CKEditorField 字段类型从 Flask-CKEditor 导入。我们可以像其他字段一样定义标签、验证器和默认值。在使用上，这个字段和 WTForms 内置的其他字段完全相同。比如，在提交表单时，同样使用 data 属性获取数据。

在模板中，渲染这个 body 字段的方式和其他字段也完全相同，在示例程序中，我们在模板 ckeditor.html 渲染了这个表单，如代码清单 4-19 所示。

代码清单4-19 form/templates/ckeditor.html：渲染包含CKEditor编辑器的表单

```
{% extends 'base.html' %}
{% from 'macros.html' import form_field %}

{% block content %}
<h1>Integrate CKEditor with Flask-CKEditor</h1>
<form method="post">
    {{ form.csrf_token }}
    {{ form_field(form.title) }}
    {{ form_field(form.body) }}
    {{ form.submit }}
</form>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ ckeditor.load() }}
{% endblock %}
```

渲染 CKEditor 编辑器需要加载相应的 JavaScript 脚本。在开发时，为了方便开发，可以使用 Flask-CKEditor 在模板中提供的 ckeditor.load() 方法加载资源，它默认从 CDN 加载资源，将 CKEDITOR_SERVE_LOCAL 设为 True 会使用扩展内置的本地资源，内置的本地资源包含了几个常用的插件和语言包。ckeditor.load() 方法支持通过 pkg_type 参数传入包类型，这会覆盖配置 CKEDITOR_PKG_TYPE 的值，额外的 version 参数可以设置从 CDN 加载的 CKEditor 版本。

作为替代，你可以访问 CKEditor 官网提供的构建工具 (<https://ckeditor.com/cke4/builder>) 构建自己的 CKEditor 包，下载后放到 static 目录下，然后在需要显示文本编辑器的模板中加载包目录下的 ckeditor.js 文件，替换掉 ckeditor.load() 调用。

如果你使用配置变量设置了编辑器的高度、宽度和语言或是其他插件配置，需要使用

`ckeditor.config()` 方法加载配置，传入对应表单字段的 `name` 属性值，即对应表单类属性名。这个方法需要在加载 CKEditor 资源后调用：

```
{{ ckeditor.config(name='body') }}
```



为了支持为不同页面上的编辑器字段或单个页面上的多个编辑器字段使用不同的配置，大多数配置键都可以通过相应的关键字在 `ckeditor.config()` 方法中传入，比如 `language`、`height`、`width` 等，这些参数会覆盖对应的全局配置。另外，Flask-CKEditor 也允许你传入自定义配置字符串，更多详情可访问 Flask-CKEditor 文档的配置部分 (<https://flask-ckeditor.readthedocs.io/configuration.html>)。

访问 `http://localhost:5000/ckeditor` 可以看到渲染后的富文本编辑器，如图 4-7 所示。

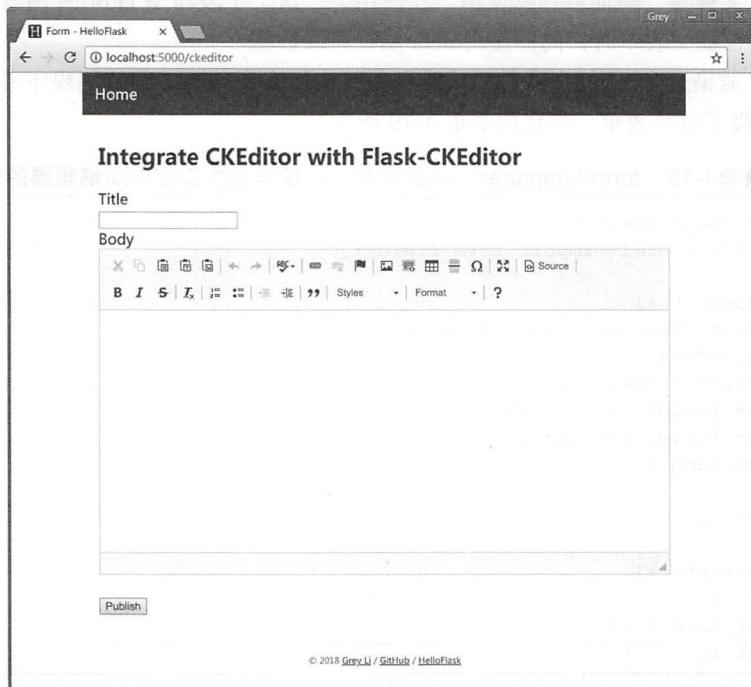


图 4-7 渲染后的编辑器页面



如果你不使用 Flask-WTF/WTForms，Flask-CKEditor 还提供了一个在模板中直接创建文本编辑器字段的 `ckeditor.create()` 方法，具体用法参考相关文档。

4.4.6 单个表单多个提交按钮

在某些情况下，我们可能需要为一个表单添加多个提交按钮。比如在创建文章的表单中添加发布新文章和保存草稿的按钮。当用户提交表单时，我们需要在视图函数中根据按下的按

钮来做出不同的处理。代码清单 4-20 创建了一个这样的表单，其中 save 表示保存草稿按钮，publish 表示发布按钮，正文字段使用 TextAreaField 字段。

代码清单4-20 form/forms.py：包含两个提交按钮的表单

```
class NewPostForm(FlaskForm):
    title = StringField('Title', validators=[DataRequired(), Length(1, 50)])
    body = TextAreaField('Body', validators=[DataRequired()])
    save = SubmitField('Save') # 保存按钮
    publish = SubmitField('Publish') # 发布按钮
```

当表单数据通过 POST 请求提交时，Flask 会把表单数据解析到 request.form 字典。如果表单中有两个提交字段，那么只有被单击的提交字段才会出现在这个字典中。当我们对表单类实例或特定的字段属性调用 data 属性时，WTForms 会对数据做进一步处理。对于提交字段的值，它会将其转换为布尔值：被单击的提交字段的值将是 True，未被单击的值则是 False。

基于这个机制，我们可以通过提交按钮字段的值来判断当前被单击的按钮，如代码清单 4-21 所示。

代码清单4-21 form/app.py：判断被单击的提交按钮

```
@app.route('/two-submits', methods=['GET', 'POST'])
def two_submits():
    form = NewPostForm()
    if form.validate_on_submit():
        if form.save.data: # 保存按钮被单击
            # save it...
            flash('You click the "Save" button.')
        elif form.publish.data: # 发布按钮被单击
            # publish it...
            flash('You click the "Publish" button.')
        return redirect(url_for('index'))
    return render_template('2submit.html', form=form)
```

访问 <http://localhost:5000/two-submits>，当你单击某个按钮时，重定向后的页面的提示信息中会包含你单击的按钮名称。

 提示 有些时候，你还想在表单添加非提交按钮。比如，添加一个返回主页的取消按钮。因为这类按钮和表单处理过程无关，最简单的方式是直接在 HTML 模板中手动添加。

4.4.7 单个页面多个表单

除了在单个表单上实现多个提交按钮，有时我们还需要在单个页面上创建多个表单。比如，在程序的主页上同时添加登录和注册表单。当在同一个页面上添加多个表单时，我们要解决的一个问题就是在视图函数中判断当前被提交的是哪个表单。

1. 单视图处理

创建两个表单，并在模板中分别渲染并不是难事，但是当提交某个表单时，我们就会遇到问题。Flask-WTF 根据请求方法判断表单是否提交，但并不判断是哪个表单被提交，所以我们需

要手动判断。基于上一节介绍的内容，我们知道被单击的提交字段最终的 data 属性值是布尔值，即 True 或 False。而解析后的表单数据使用 input 字段的 name 属性值作为键匹配字段数据，也就是说，如果两个表单的提交字段名称都是 submit，那么我们也无法判断是哪个表单的提交字段被单击。

解决问题的第一步就是为两个表单的提交字段设置不同的名称，示例程序中的这两个表单如代码清单 4-22 所示。

代码清单 4-22 form/forms.py：为两个表单设置不同的提交字段名称

```
class SigninForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(1, 20)])
    password = PasswordField('Password', validators=[DataRequired(), Length(8, 128)])
    submit1 = SubmitField('Sign in')

class RegisterForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(1, 20)])
    email = StringField('Email', validators=[DataRequired(), Email(), Length(1, 254)])
    password = PasswordField('Password', validators=[DataRequired(), Length(8, 128)])
    submit2 = SubmitField('Register')
```

在视图函数中，我们分别实例化这两个表单，根据提交字段的值来区分被提交的表单，如代码清单 4-23 所示。

代码清单 4-23 form/app.py：在视图函数中处理多个表单

```
@app.route('/multi-form', methods=['GET', 'POST'])
def multi_form():
    signin_form = SigninForm()
    register_form = RegisterForm()

    if signin_form.submit1.data and signin_form.validate():
        username = signin_form.username.data
        flash('%s, you just submit the Signin Form.' % username)
        return redirect(url_for('index'))

    if register_form.submit2.data and register_form.validate():
        username = register_form.username.data
        flash('%s, you just submit the Register Form.' % username)
        return redirect(url_for('index'))

    return render_template('2form.html', signin_form=signin_form, register_form=register_form)
```

在视图函数中，我们为两个表单添加了各自的 if 判断，在这两个 if 语句的内部，我们分别执行各自的代码逻辑。以登录表单（SigninForm）的 if 判断为例，如果 signin_form.submit1.data 的值为 True，那就说明用户提交了登录表单，这时我们手动调用 signin_form.validate() 对这个表单进行验证。

这两个表单类实例通过不同的变量名称传入模板，以便在模板中相应渲染对应的表单字段，如下所示：

```

...
<form method="post">
    {{ signin_form.csrf_token }}
    {{ form_field(signin_form.username) }}
    {{ form_field(signin_form.password) }}
    {{ signin_form.submit1 }}
</form>
<h2>Register Form</h2>
<form method="post">
    {{ register_form.csrf_token }}
    {{ form_field(register_form.username) }}
    {{ form_field(register_form.email) }}
    {{ form_field(register_form.password) }}
    {{ register_form.submit2 }}
</form>
...

```

访问 <http://localhost:5000/multi-form> 打开示例页面，当提交某个表单后，你会在重定向后的页面的提示消息里看到提交表单的名称。

2. 多视图处理

除了通过提交按钮判断，更简洁的方法是通过分离表单的渲染和验证实现。这时表单的提交字段可以使用同一个名称，在视图函数中处理表单时也只需使用我们熟悉的 `form.validate_on_submit()` 方法。

在介绍表单处理时，我们在同一个视图函数内处理两类工作：渲染包含表单的模板（GET 请求）、处理表单请求（POST 请求）。如果你想解耦这部分功能，那么也可以分离成两个视图函数处理。当处理多个表单时，我们可以把表单的渲染在单独的视图函数中处理，如下所示：

```

@app.route('/multi-form-multi-view')
def multi_form_multi_view():
    signin_form = SigninForm2()
    register_form = RegisterForm2()
    return render_template('2form2view.html', signin_form=signin_form, register_form=register_form)

```

这个视图只负责处理 GET 请求，实例化两个表单类并渲染模板。另外我们再为每一个表单单独创建一个视图函数来处理验证工作。处理表单提交请求的视图仅监听 POST 请求，如代码清单 4-24 所示。

代码清单 4-24 form/app.py：使用单独的视图函数处理表单提交的 POST 请求

```

@app.route('/handle-signin', methods=['POST']) # 仅传入POST到methods中
def handle_signin():
    signin_form = SigninForm2()
    register_form = RegisterForm2()

    if signin_form.validate_on_submit():
        username = signin_form.username.data
        flash('%s, you just submit the Signin Form.' % username)
        return redirect(url_for('index'))

```

```

        return render_template('2form2view.html', signin_form=signin_form, register_
form=register_form)

@app.route('/handle-register', methods=['POST'])
def handle_register():
    signin_form = SigninForm2()
    register_form = RegisterForm2()

    if register_form.validate_on_submit():
        username = register_form.username.data
        flash('%s, you just submit the Register Form.' % username)
        return redirect(url_for('index'))
    return render_template('2form2view.html', signin_form=signin_form, register_
form=register_form)

```

在 HTML 中，表单提交请求的目标 URL 通过 action 属性设置。为了让表单提交时将请求发送到对应的 URL，我们需要设置 action 属性，如下所示：

```

...
<h2>Login Form</h2>
<form method="post" action="{{ url_for('handle_signin') }}">
    ...
</form>
<h2>Register Form</h2>
<form method="post" action="{{ url_for('handle_register') }}">
    ...
</form>
...

```

虽然现在可以正常工作，但是这种方法有一个显著的缺点。如果验证未通过，你需要将错误消息的 form.errors 字典传入模板中。在处理表单的视图中传入表单错误信息，就意味着需要再次渲染模板，但是如果视图函数中还涉及大量要传入模板的变量操作，那么这种方式会带来大量的重复。

对于这个问题，一般的解决方式是通过其他方式传递错误消息，然后统一重定向到渲染表单页面的视图。比如，使用 flash() 函数迭代 form.errors 字典发送错误消息（这个字典包含字段名称与错误消息列表的映射），然后重定向到用来渲染表单的 multi_form_multi_view 视图。下面是一个使用 flash() 函数来发送表单错误消息的便利函数：

```

def flash_errors(form):
    for field, errors in form.errors.items():
        for error in errors:
            flash(u"Error in the %s field - %s" % (
                getattr(form, field).label.text,
                error
            ))

```

如果你希望像往常一样在表单字段下渲染错误消息，可以直接将错误消息字典 form.errors 存储到 session 中，然后重定向到用来渲染表单的 multi_form_multi_view 视图。在模板中渲染表单字段错误时添加一个额外的判断，从 session 中获取并迭代错误消息。

4.5 本章小结

除了普通的表单定义，WTForms 还提供了很多高级功能，比如自定义表单字段、动态表单等，你可以访问 WTForms 的官方文档学习更多内容。

下一章，我们会学习数据库知识，为 Flask 程序添加数据库支持，那时我们就可以把表单数据存储到数据库里了。

本章主要介绍了如何使用 WTForms 定义表单。首先介绍了表单的基本概念，包括表单的类和表单的字段。接着介绍了如何使用表单类来处理表单数据，包括表单验证、表单提交和表单重置。然后介绍了如何使用 WTForms 的表单类来处理表单数据，包括表单验证、表单提交和表单重置。最后介绍了如何使用 WTForms 的表单类来处理表单数据，包括表单验证、表单提交和表单重置。

单体应用的缺点是当系统规模变大时，单个应用的性能和可维护性都会受到严重影响。为了解决这个问题，我们可以将应用拆分成多个微服务。本章将介绍如何使用 Flask 和 SQLAlchemy 来构建一个微服务。

Chapter 3

第5章

数 据 库

数据库是大多数动态 Web 程序的基础设施，只要你想把数据存储下来，就离不开数据库。我们这里提及的数据库（Database）指的是由存储数据的单个或多个文件组成的集合，它是一种容器，可以类比为文件柜。而人们通常使用数据库来表示操作数据库的软件，这类管理数据库的软件被称为数据库管理系统（DBMS，Database Management System），常见的 DBMS 有 MySQL、PostgreSQL、SQLite、MongoDB 等。为了便于理解，我们可以把数据库看作一个大仓库，仓库里有一些负责搬运货物（数据）的机器人，而 DBMS 就是操控机器人搬运货物的程序。

这一章我们来学习如何给 Flask 程序添加数据库支持。具体来说，是学习如何在 Python 中使用这些 DBMS 来对数据库进行管理和操作。

本章新涉及的 Python 库如下所示：

- SQLAlchemy (1.2.7)
 - 主页: <http://www.sqlalchemy.org/>
 - 源码: <https://github.com/zzzeek/sqlalchemy>
 - 文档: <http://docs.sqlalchemy.org/en/latest/>
- Flask-SQLAlchemy (2.3.2)
 - 主页: <https://github.com/mitsuhiko/flask-sqlalchemy>
 - 文档: <http://flask-sqlalchemy.pocoo.org/2.3/>
- Alembic (0.9.9)
 - 主页: <https://bitbucket.org/zzzeek/alembic>
 - 文档: <http://alembic.zzzcomputing.com/en/latest/>
- Flask-Migrate (2.1.1)
 - 主页: <https://github.com/miguelgrinberg/Flask-Migrate>
 - 文档: <https://flask-migrate.readthedocs.io/en/latest/>

本章的示例程序在 helloflask/demos/database 目录下，确保当前目录在 helloflask/demos/database 下并激活了虚拟环境，然后执行 flask run 命令运行程序：

```
$ cd demos/database
$ flask run
```

 **注意** 因为所有示例程序的 CSS 文件名称、JavaScript 文件名称以及 Favicon 文件名称均相同，为了避免浏览器对不同示例程序中同名的文件进行缓存，请在第一次运行新的示例程序后按下 Ctrl+F5 或 Shift+F5 清除缓存。

5.1 数据库的分类

数据库一般分为两种，SQL (Structured Query Language, 结构化查询语言) 数据库和 NoSQL (Not Only SQL, 泛指非关系型) 数据库。

5.1.1 SQL

SQL 数据库指关系型数据库，常用的 SQL DBMS 主要包括 SQL Server、Oracle、MySQL、PostgreSQL、SQLite 等。关系型数据库使用表来定义数据对象，不同的表之间使用关系连接。表 5-1 是一个身份信息表的示例。

表 5-1 关系型数据库示例

id	name	sex	occupation
1	Nick	Male	Journalist
2	Amy	Female	Writer

在 SQL 数据库中，每一行代表一条记录 (record)，每条记录又由不同的列 (column) 组成。在存储数据前，需要预先定义表模式 (schema)，以定义表的结构并限定列的输入数据类型。

为了避免在措辞上引起误解，我们先了解几个基本概念：

- 1) 表 (table): 存储数据的特定结构。
- 2) 模式 (schema): 定义表的结构信息。
- 3) 列 / 字段 (column/field): 表中的列，存储一系列特定的数据，列组成表。
- 4) 行 / 记录 (row/record): 表中的行，代表一条记录。
- 5) 标量 (scalar): 指的是单一数据，与之相对的是集合 (collection)。

5.1.2 NoSQL

NoSQL 最初指 No SQL 或 No Relational，现在 NoSQL 社区一般会解释为 Not Only SQL。NoSQL 数据库泛指不使用传统关系型数据库中的表格形式的数据库。近年来，NoSQL 数据库越来越流行，被大量应用在实时 (real-time) Web 程序和大型程序中。与传统的 SQL 数据库相比，

它在速度和可扩展性方面有很大的优势，除此之外还拥有无模式（schema-free）、分布式、水平伸缩（horizontally scalable）等特点。

最常用的两种 NoSQL 数据库如下所示：

1. 文档存储 (document store)

文档存储是 NoSQL 数据库中最流行的种类，它可以作为主数据库使用。文档存储使用的文档类似 SQL 数据库中的记录，文档使用类 JSON 格式来表示数据。常见的文档存储 DBMS 有 MongoDB、CouchDB 等。表 5-1 的身份信息表中的第一条记录使用文档可以表示为：

```
{
    id: 1,
    name: "Nick",
    sex: "Male"
    occupation: "Journalist"
}
```

2. 键值对存储 (key-value store)

键值对存储在形态上类似 Python 中的字典，通过键来存取数据，在读取上非常快，通常用来存储临时内容，作为缓存使用。常见的键值对 DBMS 有 Redis、Riak 等，其中 Redis 不仅可以管理键值对数据库，还可以作为缓存后端（cache backend）和消息代理（message broker）。

另外，还有列存储（column store，又被称为宽列式存储）、图存储（graph store）等类型的 NoSQL 数据库，这里不再展开介绍。

5.1.3 如何选择？

NoSQL 数据库不需要定义表和列等结构，也不限定存储的数据格式，在存储方式上比较灵活，在特定的场景下效率更高。SQL 数据库稍显复杂，但不容易出错，能够适应大部分的应用场景。这两种数据库都各有优势，也各有擅长的领域。两者并不是对立的，我们需要根据使用场景选择适合的数据库类型。大型项目通常会同时需要多种数据库，比如使用 MySQL 作为主数据库存储用户资料和文章，使用 Redis（键值对型数据库）缓存数据，使用 MongoDB（文档型数据库）存储实时消息。

大多数情况下，SQL 数据库都能满足你的需求。为了便于开发和测试，本书中的示例程序都使用 SQLite 作为 DBMS。对于大型程序，在部署程序前，你需要根据程序的特点来改用更健壮的 DBMS。

5.2 ORM 魔法

在 Web 应用里使用原生 SQL 语句操作数据库主要存在下面两类问题：

- 手动编写 SQL 语句比较乏味，而且视图函数中加入太多 SQL 语句会降低代码的易读性。另外还会容易出现安全问题，比如 SQL 注入。
- 常见的开发模式是在开发时使用简单的 SQLite，而在部署时切换到 MySQL 等更健壮的

DBMS。但是对于不同的 DBMS，我们需要使用不同的 Python 接口库，这让 DBMS 的切换变得不太容易。

 **注意** 尽管使用 ORM 可以避免 SQL 注入问题，但你仍然需要对传入的查询参数进行验证。另外，在执行原生 SQL 语句时也要注意避免使用字符串拼接或字符串格式化的方式传入参数。

使用 ORM 可以很大程度上解决这些问题。它会自动帮你处理查询参数的转义，尽可能地避免 SQL 注入的发生。另外，它为不同的 DBMS 提供统一的接口，让切换工作变得非常简单。ORM 扮演翻译的角色，能够将我们的 Python 语言转换为 DBMS 能够读懂的 SQL 指令，让我们能够使用 Python 来操控数据库。

 **附注** 尽管 ORM 非常方便，但如果你对 SQL 相当熟悉，那么自己编写 SQL 代码可以获得更大的灵活性和性能优势。就像是使用 IDE 一样，ORM 对初学者来说非常方便，但进阶以后你也许会想要自己掌控一切。

ORM 把底层的 SQL 数据实体转化成高层的 Python 对象，这样一来，你甚至不需要了解 SQL，只需要通过 Python 代码即可完成数据库操作，ORM 主要实现了三层映射关系：

- 表 → Python 类。
- 字段（列）→类属性。
- 记录（行）→类实例。

比如，我们要创建一个 contacts 表来存储留言，其中包含用户名称和电话号码两个字段。在 SQL 中，下面的代码用来创建这个表：

```
CREATE TABLE contacts(
    name varchar(100) NOT NULL,
    phone_number varchar(32),
);
```

如果使用 ORM，我们可以使用类似下面的 Python 类来定义这个表：

```
from foo_orm import Model, Column, String

class Contact(Model):
    __tablename__ = 'contacts'
    name = Column(String(100), nullable=False)
    phone_number = Column(String(32))
```

要向表中插入一条记录，需要使用下面的 SQL 语句：

```
INSERT INTO contacts(name, phone_number)
VALUES('Grey Li', '12345678');
```

使用 ORM 则只需要创建一个 Contact 类的实例，传入对应的参数表示各个列的数据即可。下面的代码和使用上面的 SQL 语句效果相同：

```
contact = Contact(name='Grey Li', phone_number='12345678')
```

除了便于使用，ORM 还有下面这些优点：

- 灵活性好。你既能使用高层对象来操作数据库，又支持执行原生 SQL 语句。
- 提升效率。从高层对象转换成原生 SQL 会牺牲一些性能，但这微不足道的性能牺牲换取的是巨大的效率提升。
- 可移植性好。ORM 通常支持多种 DBMS，包括 MySQL、PostgreSQL、Oracle、SQLite 等。你可以随意更换 DBMS，只需要稍微改动少量配置。

使用 Python 实现的 ORM 有 SQLAlchemy、Peewee、PonyORM 等。其中 SQLAlchemy 是 Python 社区使用最广泛的 ORM 之一，我们将介绍如何在 Flask 程序中使用它。SQL-Alchemy，直译过来就是 SQL 炼金术，下一节我们见识到 SQLAlchemy 的神奇力量。

5.3 使用 Flask-SQLAlchemy 管理数据库

扩展 Flask-SQLAlchemy 集成了 SQLAlchemy，它简化了连接数据库服务器、管理数据库操作会话等各类工作，让 Flask 中的数据处理体验变得更加轻松。首先使用 Pipenv 安装 Flask-SQLAlchemy 及其依赖（主要是 SQLAlchemy）：

```
$ pipenv install flask-sqlalchemy
```

下面在示例程序中实例化 Flask-SQLAlchemy 提供的 SQLAlchemy 类，传入程序实例 app，以完成扩展的初始化：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

db = SQLAlchemy(app)
```

为了便于使用，我们把实例化扩展类的对象命名为 db。这个 db 对象代表我们的数据库，它可以使用 Flask-SQLAlchemy 提供的所有功能。



提示 虽然我们要使用的大部分类和函数都由 SQLAlchemy 提供，但在 Flask-SQLAlchemy 中，大多数情况下，我们不需要手动从 SQLAlchemy 导入类或函数。在 sqlalchemy 和 sqlalchemy.orm 模块中实现的类和函数，以及其他几个常用的模块和对象都可以作为 db 对象的属性调用。当我们创建这样的调用时，Flask-SQLAlchemy 会自动把这些调用转发到对应的类、函数或模块。

5.3.1 连接数据库服务器

DBMS 通常会提供数据库服务器运行在操作系统中。要连接数据库服务器，首先要为我们的程序指定数据库 URI (Uniform Resource Identifier，统一资源标识符)。数据库 URI 是一串包含各种属性的字符串，其中包含了各种用于连接数据库的信息。

 **附注** URI 代表统一资源标识符，是用来标示资源的一组字符串。URL 是它的子集。在大多数情况下，这两者可以交替使用。

表 5-2 是一些常用的 DBMS 及其数据库 URI 格式示例。

表 5-2 常用的数据库 URI 格式

DBMS	URI
PostgreSQL	postgresql://username:password@host/databasename
MySQL	mysql://username:password@host/databasename
Oracle	oracle://username:password@host:port/sidname
SQLite (UNIX)	sqlite:///absolute/path/to/foo.db
SQLite (Windows)	sqlite:///absolute\path\to\foo.db 或 r'sqlite:///absolute\path\to\foo.db'
SQLite (内存型)	sqlite:/// 或 sqlite://:memory:

在 Flask-SQLAlchemy 中，数据库的 URI 通过配置变量 `SQLALCHEMY_DATABASE_URI` 设置，默认为 SQLite 内存型数据库（`sqlite:///:memory:`）。SQLite 是基于文件的 DBMS，不需要设置数据库服务器，只需要指定数据库文件的绝对路径。我们使用 `app.root_path` 来定位数据库文件的路径，并将数据库文件命名为 `data.db`，如代码清单 5-1 所示。

代码清单 5-1 app.py：配置数据库URI

```
import os
...
app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL', 'sqlite:///+' +
os.path.join(app.root_path, 'data.db'))
```

在生产环境下更换到其他类型的 DBMS 时，数据库 URL 会包含敏感信息，所以这里优先从环境变量 `DATABASE_URL` 获取（注意这里为了便于理解使用了 URL，而不是 URI）。

 **注意** SQLite 的数据库 URI 在 Linux 或 macOS 系统下的斜线数量是 4 个；在 Windows 系统下的 URI 中的斜线数量为 3 个。内存型数据库的斜线固定为 3 个。

 **提示** SQLite 数据库文件名不限定后缀，常用的命名方式有 `foo.sqlite`, `foo.db`, 或是注明 SQLite 版本的 `foo.sqlite3`。

设置好数据库 URI 后，在 Python Shell 中导入并查看 `db` 对象会获得下面的输出：

```
>>> from app import db
>>> db
<SQLAlchemy engine=sqlite:///Path/to/your/data.db>
```

安装并初始化 Flask-SQLAlchemy 后，启动程序时会看到命令行下有一行警告信息。这是因为 Flask-SQLAlchemy 建议你设置 `SQLALCHEMY_TRACK_MODIFICATIONS` 配置变量，这个配置变量决定是否追踪对象的修改，这用于 Flask-SQLAlchemy 的事件通知系统。这个配置键的

默认值为 None，如果没有特殊需要，我们可以把它设为 False 来关闭警告信息：

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```



Flask-SQLAlchemy 计划在 3.0 版本默认将这个配置键设为 False，目前最新版本为 2.3.2。

5.3.2 定义数据库模型

用来映射到数据库表的 Python 类通常被称为数据库模型（model），一个数据库模型类对应数据库中的一个表。定义模型即使用 Python 类定义表模式，并声明映射关系。所有的模型类都需要继承 Flask-SQLAlchemy 提供的 db.Model 基类。本章的示例程序是一个笔记程序，笔记保存到数据库中，你可以通过程序查询、添加、更新和删除笔记。在代码清单 5-2 中，我们定义了一个 Note 模型类，用来存储笔记。

代码清单 5-2 app.py：定义 Note 模型

```
class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
```

在上面的模型类中，表的字段（列）由 db.Column 类的实例表示，字段的类型通过 Column 类构造方法的第一个参数传入。在这个模型中，我们创建了一个类型为 db.Integer 的 id 字段和类型为 db.Text 的 body 列，分别存储整型和文本。常用的 SQLAlchemy 字段类型如表 5-3 所示。

表 5-3 SQLAlchemy 常用的字段类型

字 段	说 明
Integer	整数
String	字符串，可选参数 length 可以用来设置最大长度
Text	较长的 Unicode 文本
Date	日期，存储 Python 的 datetime.date 对象
Time	时间，存储 Python 的 datetime.time 对象
DateTime	时间和日期，存储 Python 的 datetime 对象
Interval	时间间隔，存储 Python 的 datetime.timedelta 对象
Float	浮点数
Boolean	布尔值
PickleType	存储 Pickle 序列化的 Python 对象
LargeBinary	存储任意二进制数据

字段类型一般直接声明即可，如果需要传入参数，你也可以添加括号。对于类似 String 的字符串列，有些数据库会要求限定长度，因此最好为其指定长度。虽然使用 Text 类型可以存储

相对灵活的变长文本，但从性能上考虑，我们仅在必须的情况下使用 Text 类型，比如用户发表的文章和评论等不限长度的内容。

一般情况下，字段的长度是由程序设计者自定的。尽管如此，也有一些既定的约束标准，比如姓名（英语）的长度一般不超过 70 个字符，中文名一般不超过 20 个字符，电子邮件地址的长度不超过 254 个字符，虽然各主流浏览器支持长达 2048 个字符的 URL，但在网站中用户资料设置的限度一般为 255。尽管如此，对于超过一定长度的 Email 和 URL，比如 20 个字符，会在显示时添加省略号的形式。显示的用户名（username）允许重复，通常要短一些，以不超过 36 个字符为佳。当然，在程序中，你可以根据需要来自由设定这些限制值。

 **注** 当你在数据库模型类中限制了字段的长度后，在接收对应数据的表单类字段里，也需要使用 Length 验证器来验证用户的输入数据。

默认情况下，Flask-SQLAlchemy 会根据模型类的名称生成一个表名称，生成规则如下：

```
Message --> message # 单个单词转换为小写
FooBar --> foo_bar # 多个单词转换为小写并使用下划线分隔
```

Note 类对应的表名称即 note。如果你想自己指定表名称，可以通过定义 `_tablename_` 属性来实现。字段名默认为类属性名，你也可以通过字段类构造方法的第一个参数指定，或使用关键字 name。根据我们定义的 Note 模型类，最终将生成一个 note 表，表中包含 id 和 body 字段。

除了 name 参数，实例化字段类时常用的字段参数如表 5-4 所示。

表 5-4 常用的 SQLAlchemy 字段参数

参数名	说明
<code>primary_key</code>	如果设为 True，该字段为主键
<code>unique</code>	如果设为 True，该字段不允许出现重复值
<code>index</code>	如果设为 True，为该字段创建索引，以提高查询效率
<code>nullable</code>	确定字段值可否为空，值为 True 或 False，默认值为 True
<code>default</code>	为字段设置默认值

 **提** 不需要在所有列都建立索引。一般来说，取值可能性多（比如姓名）的列，以及经常被用来作为排序参照的列（比如时间戳）更适合建立索引。

在实例化字段类时，通过把参数 `primary_key` 设为 True 可以将其定义为主键。在我们定义的 Note 类中，id 字段即表的主键（primary key）。主键是每一条记录（行）独一无二的标识，也是模型类中必须定义的字段，一般命名为 id 或 pk。

5.3.3 创建数据库和表

如果把数据库（文件）看作一个仓库，为了方便取用，我们需要把货物按照类型分别放置在不同货架上，这些货架就是数据库中的表。创建模型类后，我们需要手动创建数据库和对应的表，也就是我们常说的建库和建表。这通过对我们的 db 对象调用 `create_all()` 方法实现：

```
$ flask shell
>>> from app import db
>>> db.create_all()
```



注意 如果你将模型类定义在单独的模块中，那么必须在调用 `db.create_all()` 方法前导入相应模块，以便让 SQLAlchemy 获得模型类被创建时生成的表信息，进而正确生成数据表。

通过下面的方式可以查看模型对应的 SQL 模式（建表语句）：

```
>>> from sqlalchemy.schema import CreateTable
>>> print(CreateTable(Note.__table__))
```

```
CREATE TABLE note (
    id INTEGER NOT NULL,
    body TEXT,
    PRIMARY KEY (id)
)
```



注意 数据库和表一旦创建后，之后对模型的改动不会自动作用到实际的表中。比如，在模型类中添加或删除字段，修改字段的名称和类型，这时再次调用 `create_all()` 也不会更新表结构。如果要使改动生效，最简单的方式是调用 `db.drop_all()` 方法删除数据库和表，然后再调用 `db.create_all()` 方法创建，后面会具体介绍。

我们也可以自己实现一个自定义 flask 命令完成这个工作，如代码清单 5-3 所示。

代码清单 5-3 `demos/database/app.py`: 用于创建数据库和表的 flask 命令

```
import click
...
@app.cli.command()
def initdb():
    db.create_all()
    click.echo('Initialized database.')
```

在命令行下输入 `flask initdb` 即可创建数据库和表：

```
$ flask initdb
Initialized database.
```

对于示例程序来说，这会在 `database` 目录下创建一个 `data.db` 文件。



提示 在开发程序或是部署后，我们经常需要在 Python Shell 中手动操作数据库（生产环境需注意备份），对于一次性操作，直接处理即可。对于需要重用的操作，我们可以编写成 Flask 命令、函数或是模型类的类方法。

5.4 数据库操作

现在我们创建了模型，也生成了数据库和表，是时候来学习常用的数据库操作了。数据库操作主要是 CRUD，即 Create（创建）、Read（读取 / 查询）、Update（更新）和 Delete（删除）。

SQLAlchemy 使用数据库会话来管理数据库操作，这里的数据库会话也称为事务（transaction）。Flask-SQLAlchemy 自动帮我们创建会话，可以通过 db.session 属性获取。



注意 SQLAlchemy 中的数据库会话对象和我们在前面介绍的 Flask 中的 session 无关。

数据库中的会话代表一个临时存储区，你对数据库做出的改动都会存放在那里。你可以调用 add() 方法将新创建的对象添加到数据库会话中，或是对会话中的对象进行更新。只有当你对数据库会话对象调用 commit() 方法时，改动才被提交到数据库，这确保了数据提交的一致性。另外，数据库会话也支持回滚操作。当你对会话调用 rollback() 方法时，添加到会话中且未提交的改动都将被撤销。

5.4.1 CRUD

这一节我们会在 Python Shell 中演示 CRUD 操作。默认情况下，Flask-SQLAlchemy(>=2.3.0 版本) 会自动为模型类生成一个 __repr__() 方法。当在 Python Shell 中调用模型的对象时，__repr__() 方法会返回一条类似“<模型类名 主键值>”的字符串，比如 <Note 2>。为了便于实际操作测试，示例程序中，所有的模型类都重新定义了 __repr__() 方法，返回一些更有用的信息，比如：

```
class Note(db.Model):
    ...
    def __repr__(self):
        return '<Note %r>' % self.body
```

在实际开发中，这并不是必须的。另外，为了节省篇幅，后面的模型类定义不会给出这部分代码，具体可到源码仓库中查看。

1. Create

添加一条新记录到数据库主要分为三步：

- 1) 创建 Python 对象（实例化模型类）作为一条记录。
- 2) 添加新创建的记录到数据库会话。
- 3) 提交数据库会话。

下面的示例向数据库中添加了三条留言：

```
>>> from app import db, Note
>>> note1 = Note(body='remember Sammy Jankis')
>>> note2 = Note(body='SHAVE')
>>> note3 = Note(body='DON'T BELIEVE HIS LIES, HE IS THE ONE, KILL HIM')
>>> db.session.add(note1)
>>> db.session.add(note2)
>>> db.session.add(note3)
>>> db.session.commit()
```

在这个示例中，我们首先从 app 模块导入 db 对象和 Note 类，然后分别创建三个 Note 实例表示三条记录，使用关键字参数传入字段数据。我们的 Note 类继承自 db.Model 基类，db.Model 基类会为 Note 类提供一个构造函数，接收匹配类属性名称的参数值，并赋值给对应的类属性，所以我们不需要自己在 Note 类中定义构造方法。接着我们调用 add() 方法把这三个 Note 对象添

加到会话对象 db.session 中，最后调用 commit() 方法提交会话。

 提示 除了依次调用 add() 方法添加多个记录，也可以使用 add_all() 一次添加包含所有记录对象的列表。

你可能注意到了，我们在创建模型类实例的时候并没有定义 id 字段的数据，这是因为主键由 SQLAlchemy 管理。模型类对象创建后作为临时对象 (transient)，当你提交数据库会话后，模型类对象才会转换为数据库记录写入数据库中，这时模型类对象会自动获得 id 值：

```
>>> note1.id
1
```

 注意 Flask-SQLAlchemy 提供了一个 SQLALCHEMY_COMMIT_ON_TEARDOWN 配置变量，将其设为 True 可以设置自动调用 commit() 方法提交数据库会话。因为存在潜在的 Bug，目前已不建议使用，而且未来版本中将移除该配置变量。请避免使用该配置变量，可使用手动调用 db.session.commit() 方法的方式提交数据库会话。

2. Read

我们已经知道了如何向数据库里添加记录，那么如何从数据库里取回数据呢？使用模型类提供的 query 属性附加调用各种过滤方法及查询方法可以完成这个任务。

一般来说，一个完整的查询遵循下面的模式：

<模型类>.query.<过滤方法>.<查询方法>

从某个模型类出发，通过在 query 属性对应的 Query 对象上附加的过滤方法和查询函数对模型类对应的表中的记录进行各种筛选和调整，最终返回包含对应数据库记录数据的模型类实例，对返回的实例调用属性即可获取对应的字段数据。

如果你执行了上面小节里的操作，我们的数据库现在一共会有三条记录，如表 5-5 所示。

表 5-5 note 表示意

id	body
1	remember Sammy Jankis.
2	SHAVE
3	DON'T BELIEVE HIS LIES, HE IS THE ONE, KILL HIM

SQLAlchemy 提供了许多查询方法用来获取记录，表 5-6 列出了常用的查询方法。

表 5-6 常用的 SQLAlchemy 查询方法

查询方法	说 明
all()	返回包含所有查询记录的列表
first()	返回查询的第一条记录，如果未找到，则返回 None
one()	返回第一条记录，且仅允许有一条记录。如果记录数量大于 1 或小于 1，则抛出错误

(续)

查询方法	说 明
get(ident)	传入主键值作为参数，返回指定主键值的记录，如果未找到，则返回 None
count()	返回查询结果的数量
one_or_none()	类似 one()，如果结果数量不为 1，返回 None
first_or_404()	返回查询的第一条记录，如果未找到，则返回 404 错误响应
get_or_404(ident)	传入主键值作为参数，返回指定主键值的记录，如果未找到，则返回 404 错误响应
paginate()	返回一个 Pagination 对象，可以对记录进行分页处理
with_parent(instance)	传入模型类实例作为参数，返回和这个实例相关联的对象，后面会详细介绍

 **附注** 表 5-6 中的 first_or_404()、get_or_404() 以及 paginate() 方法是 Flask-SQLAlchemy 附加的查询方法。

下面是对 Note 类进行查询的几个示例。all() 返回所有记录：

```
>>> Note.query.all()
[<Note u'remember Sammy Jankis'>, <Note u'SHAVE'>, <Note u'DON'T BELIEVE HIS
LIES, HE IS THE ONE, KILL HIM'>]
```

first() 返回第一条记录：

```
>>> note1 = Note.query.first()
>>> note1
<Note u'remember Sammy Jankis'>
>>> note1.body
u'remember Sammy Jankis'
```

get() 返回指定主键值 (id 字段) 的记录：

```
>>> note2 = Note.query.get(2)
>>> note2
<Note u'SHAVE'>
```

count() 返回记录的数量：

```
>>> Note.query.count()
3
```

SQLAlchemy 还提供了许多过滤方法，使用这些过滤方法可以获取更精确的查询，比如获取指定字段值的记录。对模型类的 query 属性存储的 Query 对象调用过滤方法将返回一个更精确的 Query 对象（后面我们简称为查询对象）。因为每个过滤方法都会返回新的查询对象，所以过滤器可以叠加使用。在查询对象上调用前面介绍的查询方法，即可获得一个包含过滤后的记录的列表。常用的查询过滤方法如表 5-7 所示。

 提示 完整的查询方法和过滤方法列表在 <http://docs.sqlalchemy.org/en/latest/orm/query.html> 可以看到。

表 5-7 常用的 SQLAlchemy 过滤方法

查询过滤器名称	说 明
filter()	使用指定的规则过滤记录，返回新产生的查询对象
filter_by()	使用指定规则过滤记录（以关键字表达式的形式），返回新产生的查询对象
order_by()	根据指定条件对记录进行排序，返回新产生的查询对象
limit(limit)	使用指定的值限制原查询返回的记录数量，返回新产生的查询对象
group_by()	根据指定条件对记录进行分组，返回新产生的查询对象
offset(offset)	使用指定的值偏移原查询的结果，返回新产生的查询对象

filter() 方法是最基础的查询方法。它使用指定的规则来过滤记录，下面的示例在数据库里找出了 body 字段值为“SHAVE”的记录：

```
>>> Note.query.filter(Note.body='SHAVE').first()
<Note u'SHAVE'>
```

直接打印查询对象或将其转换为字符串可以查看对应的 SQL 语句：

```
>>> print(Note.query.filter_by(body='SHAVE'))
SELECT note.id AS note_id, note.body AS note_body
FROM note
WHERE note.body = ?
```

在 filter() 方法中传入表达式时，除了“==”以及表示不等于的“!=”，其他常用的查询操作符以及使用示例如下所示：

LIKE:

```
filter(Note.body.like('%foo%'))
```

IN:

```
filter(Note.body.in_(['foo', 'bar', 'baz']))
```

NOT IN:

```
filter(~Note.body.in_(['foo', 'bar', 'baz']))
```

AND:

```
# 使用and_()
from sqlalchemy import and_
filter(and_(Note.body == 'foo', Note.title == 'FooBar'))
```

```
# 或在filter()中加入多个表达式，使用逗号分隔
filter(Note.body == 'foo', Note.title == 'FooBar')
```

```
# 或叠加调用多个filter()/filter_by()方法
filter(Note.body == 'foo').filter(Note.title == 'FooBar')
```

OR:

```
from sqlalchemy import or_
filter(or_(Note.body == 'foo', Note.body == 'bar'))
```

 **附注** 完整的可用操作符列表可以访问 <http://docs.sqlalchemy.org/en/latest/core/sqlelement.html#sqlalchemy.sql.operators.ColumnOperators> 查看。

和 filter() 方法相比，filter_by() 方法更易于使用。在 filter_by() 方法中，你可以使用关键字表达式来指定过滤规则。更方便的是，你可以在过滤器中直接使用字段名称。下面的示例使用 filter_by() 过滤器完成了同样的任务：

```
>>> Note.query.filter_by(body='SHAVE').first()
<Note u'SHAVE'>
```

其他的查询方法我们会在后面具体应用时详细介绍。

3. Update

更新一条记录非常简单，直接赋值给模型类的字段属性就可以改变字段值，然后调用 commit() 方法提交会话即可。下面的示例改变了一条记录的 body 字段的值：

```
>>> note = Note.query.get(2)
>>> note.body
u'SHAVE'
>>> note.body = 'SHAVE LEFT THIGH'
>>> db.session.commit()
```

 **提示** 只有要插入新的记录或要将现有的记录添加到会话中时才需要使用 add() 方法，单纯要更新现有的记录时只需要直接为属性赋新值，然后提交会话。

4. Delete

删除记录和添加记录很相似，不过要把 add() 方法换成 delete() 方法，最后都需要调用 commit() 方法提交修改。下面的示例删除了 id (主键) 为 2 的记录：

```
>>> note = Note.query.get(2)
>>> db.session.delete(note)
>>> db.session.commit()
```

5.4.2 在视图函数里操作数据库

在视图函数里操作数据库的方式和我们在 Python Shell 中的练习大致相同，只不过需要一些额外的工作。比如把查询结果作为参数传入模板渲染出来，或是获取表单的字段值作为提交到数据库的数据。在这一节，我们将把上一节学习的所有数据库操作知识运用到一个简单的笔记程序中。这个程序可以让你创建、编辑和删除笔记，并在主页列出所有保存后的笔记。

1. Create

为了支持输入笔记内容，我们先创建一个用于填写新笔记的表单，如下所示：

```
from flask_wtf import FlaskForm
from wtforms import TextAreaField, SubmitField
from wtforms.validators import DataRequired
```

```
class NewNoteForm(FlaskForm):
    body = TextAreaField('Body', validators=[DataRequired()])
    submit = SubmitField('Save')
```

我们创建一个 new_note 视图，这个视图负责渲染创建笔记的模板，并处理表单的提交，如代码清单 5-4 所示。

代码清单 5-4 demos/database/app.py：创建新笔记

```
@app.route('/new', methods=['GET', 'POST'])
def new_note():
    form = NewNoteForm()
    if form.validate_on_submit():
        body = form.body.data
        note = Note(body=body)
        db.session.add(note)
        db.session.commit()
        flash('Your note is saved.')
        return redirect(url_for('index'))
    return render_template('new_note.html', form=form)
```

我们先来看看 form.validate_on_submit() 返回 True 时的处理代码。当表单被提交且通过验证时，我们获取表单 body 字段的数据，然后创建新的 Note 实例，将表单中 body 字段的值作为 body 参数传入，最后添加到数据库会话中并提交会话。这个过程接收用户通过表单提交的数据并保存到数据库中，最后我们使用 flash() 函数发送提示消息并重定向到 index 视图。

表单在 new_note.html 模板中渲染，这里使用我们在第 4 章介绍的 form_field 宏渲染表单字段，传入 rows 和 cols 参数来定制 <textarea> 输入框的大小：

```
{% block content %}
<h2>New Note</h2>
<form method="post">
    {{ form.csrf_token }}
    {{ form_field(form.body, rows=5, cols=50) }}
    {{ form.submit }}
</form>
{% endblock %}
```

index 视图用来显示主页，目前它的所有作用就是渲染主页对应的模板：

```
@app.route('/')
def index():
    return render_template('index.html')
```

在对应的 index.html 模板中，我们添加一个指向创建新笔记页面的链接：

```
<h1>Notebook</h1>
<a href="{{ url_for('new_note') }}>New Note</a>
```

2. Read

在上一节我们为程序实现了添加新笔记的功能，当你在创建笔记的页面单击保存后，程序会重定向到主页，提示的消息告诉你刚刚提交的笔记已经成功保存了，可是你却无法看到创建后的笔记。为了在主页列出所有保存的笔记，我们需要修改 index 视图，修改后的 index 视图如

代码清单 5-5 所示。

代码清单5-5 demos/database/app.py：在视图函数中查询数据库记录并传入模板

```
@app.route('/')
def index():
    form = DeleteForm()
    notes = Note.query.all()
    return render_template('index.html', notes=notes, form=form)
```

在新的 index 视图里，我们像在 Python Shell 中一样使用 Note.query.all() 查询所有 note 记录，然后把这个包含所有记录的列表作为 notes 变量传入模板。你已经猜到下一步了，没错，我们将在模板中将笔记们显示出来，如代码清单 5-6 所示。

代码清单5-6 demos/database/templates/index.html：在模板中渲染数据库记录

```
<h1>Notebook</h1>
<a href="{{ url_for('new_note') }}>New Note</a>
<h4>{{ notes|length }} notes:</h4>
{% for note in notes %}
    <div class="note">
        <p>{{ note.body }}</p>
    </div>
{% endfor %}
```

在模板中，我们迭代这个 notes 列表，调用 Note 对象的 body 属性（note.body）获取 body 字段的值。另外，我们还通过 length 过滤器获取笔记的数量。渲染后的示例如图 5-1 所示。

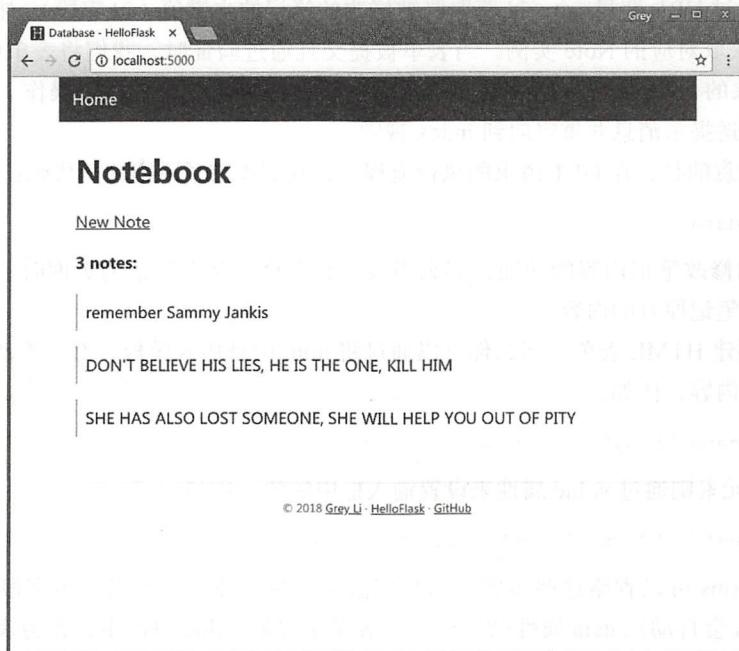


图 5-1 显示笔记列表

3. Update

更新一条笔记和创建一条新笔记的实现代码几乎完全相同，首先是编辑笔记的表单：

```
class EditNoteForm(FlaskForm):
    body = TextAreaField('Body', validators=[DataRequired()])
    submit = SubmitField('Update')
```

你会发现这和创建新笔记 NewNoteForm 唯一的不同就是提交字段的标签参数（作为 <input> 的 value 属性），因此这个表单的定义也可以通过继承来简化：

```
class EditNoteForm(NewNoteForm):
    submit = SubmitField('Update')
```

用来渲染更新笔记页面和处理更新表单提交的 edit_note 视图如代码清单 5-7 所示。

代码清单 5-7 database/app.py：更新笔记内容

```
@app.route('/edit/<int:note_id>', methods=['GET', 'POST'])
def edit_note(note_id):
    form = EditNoteForm()
    note = Note.query.get(note_id)
    if form.validate_on_submit():
        note.body = form.body.data
        db.session.commit()
        flash('Your note is updated.')
        return redirect(url_for('index'))
    form.body.data = note.body
    return render_template('edit_note.html', form=form)
```

这个视图通过 URL 变量 note_id 获取要被修改的笔记的主键值 (id 字段)，然后我们就可以使用 get() 方法获取对应的 Note 实例。当表单被提交且通过验证时，我们将表单中 body 字段的值赋给 note 对象的 body 属性，然后提交数据库会话，这样就完成了更新操作。和创建笔记相同，我们接着发送提示消息并重定向到 index 视图。

唯一需要注意的是，在 GET 请求的执行流程中，我们添加了下面这行代码：

```
form.body.data = note.body
```

因为要添加修改笔记内容的功能，那么当我们打开修改某个笔记的页面时，这个页面的表单中必然要包含笔记原有的内容。

如果手动创建 HTML 表单，那么你可以通过将 note 记录传入模板，然后手动为对应字段中填入笔记的原有内容，比如：

```
<textarea name="body">{{ note.body }}</textarea>
```

其他 input 元素则通过 value 属性来设置输入框中的值，比如：

```
<input name="foo" type="text" value="{{ note.title }}>
```

使用 WTForms 可以省略这些步骤，当我们渲染表单字段时，如果表单字段的 data 属性不为空，WTForms 会自动把 data 属性的值添加到表单字段的 value 属性中，作为表单的值填充进去，我们不用手动为 value 属性赋值。因此，将存储笔记原有内容的 note.body 属性赋值给表单