1. render\_template()函数调用Flask框架原生依赖的[Jinja2](http://jinja.pocoo.org/)模板引擎。 Jinja2用render\_template()函数传入的参数中的相应值替换{{...}}块。
2. Jinja2有一个模板继承特性，专门解决编写重复的代码。从本质上来讲，就是将所有模板中相同的部分转移到一个基础模板中，然后再从它继承过来。

*app/templates/base.html*中编写代码如下：

<html>

<head>

{% if title %}

<title>{{ title }} - Microblog</title>

{% else %}

<title>Welcome to Microblog</title>

{% endif %}

</head>

<body>

<div>Microblog: <a href="/index">Home</a></div>

<hr>

{% block content %}{% endblock %}

</body>

</html>

通过从基础模板base.html继承HTML元素，我现在可以简化模板index.html了：

{% extends "base.html" %}

{% block content %}

<h1>Hi, {{ user.username }}!</h1>

{% for post in posts %}

<div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>

{% endfor %}

{% endblock %}

有几种途径来为应用指定配置选项。最基本的解决方案是使用app.config对象，它是一个类似字典的对象，可以将配置以键值的方式存储其中。例如，你可以这样做：

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'you-will-never-guess'

# ... add more variables here as needed

上面的代码虽然可以为应用创建配置，但是我有松耦合的癖好。因此，我不会让配置和应用代码处于同一个部分，而是使用稍微复杂点的结构，将配置保存到一个单独的文件中。

使用类来存储配置变量，才是我真正的风格。我会将这个配置类存储到单独的Python模块，以保持良好的组织结构。下面就让你见识一下这个存储在顶级目录下，名为config.py的模块的配置类吧：

import os

class Config(object):

SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'you-will-never-guess'

拥有了这样一份配置文件，我还需要通知Flask读取并使用它。可以在生成Flask应用之后，利用app.config.from\_object()方法来完成这个操作：

from flask import Flask

from config import Config

app = Flask(\_\_name\_\_)

app.config.from\_object(Config)

from app import routes

1. with结构在当前模板的上下文中来将get\_flashed\_messages()的结果赋值给变量messages。get\_flashed\_messages()是Flask中的一个函数，它返回用flash()注册过的消息列表。
2. 为了更好地管理这些链接，Flask提供了一个名为url\_for()的函数，它使用URL到视图函数的内部映射关系来生成URL。 例如，url\_for('login')返回/login，url\_for('index')返回/index。 url\_for()的参数是endpoint名称，也就是视图函数的名字。
3. 通常，在服务应用中使用UTC日期和时间是推荐做法。 这可以确保你使用统一的时间戳，无论用户位于何处，这些时间戳会在显示时转换为用户的当地时间。
4. shell命令是Flask在继run之后的实现第二个“核心”命令。 这个命令的目的是在应用的上下文中启动一个Python解释器。

使用常规的解释器会话时，除非明确地被导入，否则app对象是未知的，但是当使用flask shell时，该命令预先导入应用实例。 flask shell的绝妙之处不在于它预先导入了app，而是你可以配置一个“shell上下文”，也就是可以预先导入一份对象列表。