第 5 章

智能 Web App

本章基于 Flask 框架搭建 Web 服务器，客户端通过浏览器向服务器提交待识别的图像，服务器基于第 4 章创建的 DenseNet121 模型做出预测，并将预测结果返回给客户端，从而实现智能化的 Web 服务能力。服务器采用 RESTful 风格的 API 搭建 Web 服务，客户端 App 无论采用何种语言编程，均可通过 HTTP 访问服务器。

5.1 环境准备

视频讲解

Flask 是一个基于 Python 的轻量级 WSGI Web 应用程序框架，可扩展性好，拥有丰富的第三方支持库。用 pip install flask 命令在项目虚拟环境中安装 Flask 框架。

Postman 是一款模拟 HTTP 请求的客户端调试工具，帮助程序员完成简单直观的 HTTP 请求测试。DB Browser for SQLite 是实现 SQLite 数据库可视化管理的工具软件。可到官方网站下载安装 Postman 和 DB Browser for SQLite。

本章开发的智能 Web App 基本架构如图 5.1 所示，服务器部署第 3 章创建的 apple.db 数据库和第 4 章完成的 DenseNet121 模型，客户机通过浏览器向服务器发送 HTTP 请求，服务器运行基于 Flask 开发的 RESTful API，向客户机返回响应消息。

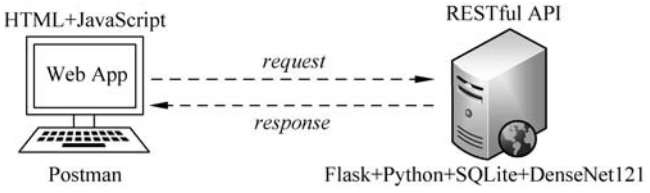

图 5.1 Web App 智能服务框架

为了便于观察客户机与服务器之间的 HTTP 请求/响应关系，客户机采用 Postman 发送请求并接收响应消息。

5.2 项目概要设计

视频讲解

服务器基于 Flask 框架实现 Web API 设计，包含的服务模块如图 5.2 所示，各模块的功能简述如下。

(1) HTTP 状态码模块。用于学习和理解如何捕获 Web 响应的状态。

(2) 获取 URL 参数模块。用于理解 Web API 获取客户机请求参数的方法。

(3) 用户注册模块。获取客户机提交的注册数据,将其写入 apple.db 数据库中。

(4) 用户登录模块。配合 JSON Web Token 机制,完成用户的身份验证。

(5) 发送邮件模块。用户登录成功时向用户发送邮件,或者需要找回密码时,向用户发送邮件。

(6) 查询记录模块。实现两种查询方法:一是查询 apple.db 所有记录;二是有条件查询。操作结果反馈给客户机。

(7) 添加记录模块。向 apple.db 插入一条来自客户机的新数据。操作结果反馈给客户机。

(8) 更新记录模块。根据客户机指定的 ID 修改 apple.db 中的指定记录。操作结果反馈给客户机。

(9) 删除记录模块。根据指定记录的 ID 从数据库 apple.db 中删除指定的记录。操作结果反馈给客户机。

(10) 分类预测模块。调用 DenseNet 预测模型,对客户机发送过来的图像做预测,并将预测结果返回给客户机。

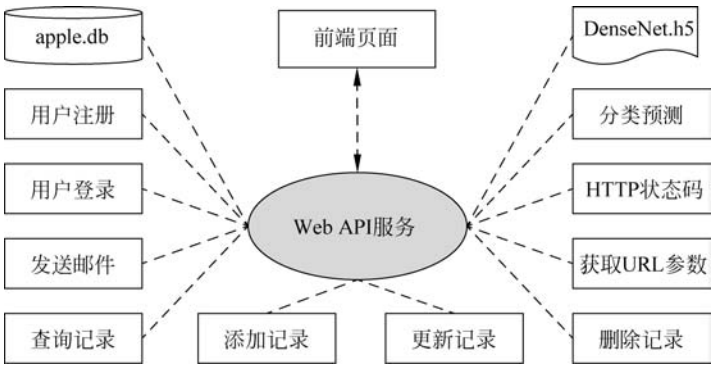

图 5.2 Web 服务模块概要设计


视频讲解

5.3 新建 Flask Web 项目

在 NetworkProgram 项目下新建文件夹 chapter5,在 chapter5 下新建子目录 static、templates 和 models。将第 4 章完成的 DenseNet121 模型复制到子目录 models。第 3 章爬虫收集的数据存储在 apple.db 中,将数据库 apple.db 复制到 chapter5 目录。

在 chapter5 目录新建主程序文件 app.py,当前项目结构如图 5.3 所示。

完成 app.py 的初始测试程序段 P5.1,运行 app.py,分别在浏览器和 Postman 观察客户端输出结果。

P5.1 # Web 服务测试

```
01  from flask import Flask, jsonify
```

```
02  import os
03  app = Flask('__name__')                                          # 初始化 App
04  db_path = os.path.join(os.getcwd(), 'apple.db')                  # 数据库路径
05  model_path = os.path.join(os.getcwd(), 'models/densenet121.h5')  # 模型路径
06  @app.route('/')                                                  # 根目录 Web 服务
07  def index():
08      message = {'db':'apple.db', 'model': 'densenet121.h5'}
09      return jsonify(message)
10  if __name__ == '__main__':
11      app.run(debug = True)
```

图 5.3　Web App 项目初始结构

程序段第 6 行定义路径为根目录的 Web 服务，第 7 行定义服务调用的函数，第 8 行用字典模式表达返回客户端的消息格式，第 9 行用 jsonify()函数将消息转换为 JSON 格式后返回。第 11 行设置 Flask 运行于 Debug 模式，便于调试程序时，客户端能够动态捕获服务器端的变化与调整。

5.4　HTTP 状态码


视频讲解

Web 基于 HTTP 实现数据交换过程，请求过程与响应过程均包含 HTTP 控制头，HTTP 控制头包含对终端用户不可见的元数据信息，分别用于描述请求过程和响应过程。

状态码是 HTTP 响应控制头中包含的数据，用于提示请求的结果类型。常见的 HTTP 状态码如表 5.1 所示。

表 5.1　常见的 HTTP 状态码

状态码	描　　述	举　　例
1××	服务器收到请求，需要请求者继续执行操作	100：客户端应继续提交请求
2××	请求被成功接收并处理	200：请求成功
3××	重定向，需要进一步的操作以完成请求	301：资源被转移到其他 URL
4××	客户端错误，请求包含语法错误或无法完成请求	404：请求的资源不存在
5××	服务器错误，服务器在处理请求的过程中发生了错误	500：服务器内部错误

HTTP状态码极其有用,客户机可以通过HTTP状态码判断请求的结果以及需要进一步采取的操作。

程序段P5.2演示了在Web服务中添加状态响应码的编程方法。

```
P5.2  # HTTP状态码测试
01  @app.route('/')      # 根目录Web服务
02  def index():
03      message = {'db':'apple.db', 'model': 'densenet121.h5'}
04      return jsonify(message), 200
05  @app.route('/not_found')
06  def not_found():
07      return jsonify(message = '请求的资源不存在!'), 404
```

第4行添加了成功响应请求的状态码200,第7行语句添加了无法访问资源的状态码404。在Postman中分别对两个服务测试,注意观察状态码的反馈值是否与对应的服务相匹配。


视频讲解

5.5 获取URL参数

客户端可以通过在URL末尾附加参数的方式向服务器提交数据,URL与参数之间以问号"?"间隔。为了测试URL参数获取方法,假定携带苹果树病虫害名称参数的URL如下所示:http://localhost?name=苹果黑星病。

程序段P5.3演示了服务器端获取URL参数值的方法。

```
P5.3  # 获取URL参数
01  @app.route('/parameters')
02  def parameters():
03      name = request.args.get('name')      # 获取URL参数
04      if name in ['健康叶片','苹果黑星病','苹果锈病','多种病症']:
05          return jsonify(msessage = '数据库存在请求的数据:' + name), 200
06      else:
07          return jsonify(message = '数据库不存在请求的数据:' + name), 401
```

第3行用Flask自带的request模块读取参数值。第5行、第7行在返回信息到客户端的同时,指定了HTTP状态码。

在Postman中测试客户端请求,观察反馈的信息与HTTP状态码。


视频讲解

5.6 定义用户数据表

用DB Browser for SQLite打开chpater5目录下的数据库apple.db,新增用户数据表users,表结构定义如图5.4所示。

字段id为自增长主键,name、email、password三个字段非空,其中email必须唯一。

图 5.4 用户数据表的结构定义

5.7 用户注册


视频讲解

定义用户注册 API 如程序段 P5.4 所示。用户注册数据一般通过表单提交,所以 register 服务被指定为用 post()方法获取数据。对于邮箱账号已存在的用户,取消注册逻辑,反馈状态码为 409 的提示信息,表示禁止注册。如果邮箱账号不存在,则允许注册,完成注册逻辑后,反馈状态码为 201 的提示信息,提示用户注册成功。

P5.4 # 用户注册

```
@app.route('/register', methods = ['post'])
def register():
    email = request.form['email']
    conn = lite.connect(db_path)          # 打开数据库
    with conn:
        cur = conn.cursor()
        sql = f"select count(email) from users where email = '{email}'"
        cur.execute(sql)
        count = cur.fetchone()[0]
        if count > 0:                     # 邮箱账号已存在
            return jsonify(message = '注册的邮箱已存在!'), 409
        else:
            name = request.form['name']
            password = request.form['password']
            sql = f"insert into users(name, email, password) values ('{name}',
                  '{email}', '{password}')"
```

```
17              cur.execute(sql)
18              return jsonify(message = '用户注册成功!'), 201
```

用 Postman 做表单数据提交测试,如图 5.5 所示,收到的状态码 201 表示注册成功。

图 5.5 Postman 提交用户注册数据并接收 Web 服务反馈信息

修改 Postman 表单数据,验证邮箱存在的情况,注意观察服务器响应的状态码以及提示信息。

视频讲解

5.8 JSON Web 令牌

如果需要限定只有 users 表中注册的用户才能对 apples 表中的数据做增、删、改、查操作,则需要在服务器端定义验证用户登录的服务模块。

HTTP 是一种无状态的协议,如果用户向服务器提供了用户名和密码做身份认证,那么在做下一次请求时,需要重新提交用户名和密码进行身份验证,常用的解决方案是在服务器端存储一份用户登录的信息,这就是传统的基于 Session 的认证机制。

随着认证用户的增多,基于 Session 的认证在服务器端的开销会明显增大,在面对多服务器集群模式时,用户的 Session 会被迫绑定到一台服务器上,限制了集群服务器的负载均衡能力。

基于 JSON Web 令牌的认证机制不需要在服务器端保留用户的认证信息或会话信息,这就意味着基于令牌的认证机制不需要考虑用户在哪一台服务器登录,可扩展性明显优于 Session 机制。

本章采用 JSON Web 令牌作为身份验证方法。在项目虚拟环境执行命令:

```
pip install flask - jwt - extended
```

完成 Flask-JWT-Extended 模块的安装。调用 create_access_token() 函数创建 JSON Web

令牌，调用 jwt_required()函数保护 Web 服务，get_jwt_identity()返回用户的身份标识。

JSON Web Token 的基本结构如图 5.6 所示，包括 Header、Payload、Signature 三部分。Header 用 Base64 编码封装加密算法等信息，Payload 用 Base64 编码封装传递的敏感数据信息，Signature 封装基于 Header、Payload 以及密钥 secret 生成的签名。

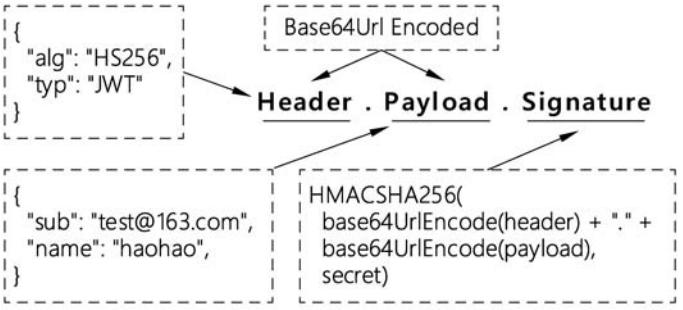

图 5.6　JSON Web Token 的基本结构

JSON Web 令牌的认证机制如图 5.7 所示，简述如下。

(1) 用户通过用户名和密码来请求服务器，服务器验证用户的信息。

(2) 验证通过后，服务器向用户发送一个令牌(Token)。

(3) 客户端存储令牌，每次新请求时附加自己的令牌，服务器端通过验证令牌，决定是否提供服务。

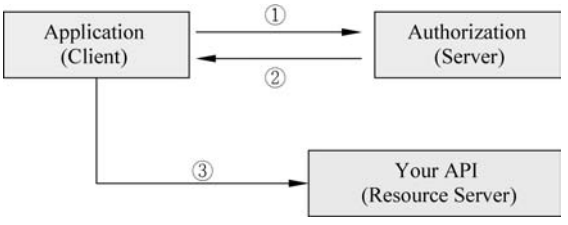

图 5.7　JSON Web Token 认证机制

5.9　用户登录


视频讲解

程序段 P5.5 实现了基于 JSON Web Token 的登录逻辑。

P5.5　# 用户登录

```
from flask_jwt_extended import JWTManager, jwt_required, create_access_token
app.config['SECRET_KEY'] = 'my-secret'
jwt = JWTManager(app)
@app.route('/login', methods=['post'])
def login():
    email = request.form['email']
    password = request.form['password']
    conn = lite.connect(db_path)                                    # 打开数据库
    with conn:
```

```
        cur = conn.cursor()
        sql = f"select count(email) from users where email = '{email}' and password = '{password}'"
        cur.execute(sql)
        count = cur.fetchone()[0]
        if count > 0:                                                    # 允许登录
            acess_token = create_access_token(identity = email)  # 创建 Token
            return jsonify(message = '登录成功!', acess_token = acess_token), 202
        else:                                                            # 拒绝登录
            return jsonify(message = '用户 Email 或者密码错误,登录失败!'), 401
```

复制图 5.8 所示的 access_token,粘贴到 https://jwt.io/网站的编码-解码框,可以观察解码后的 Token 结构信息。

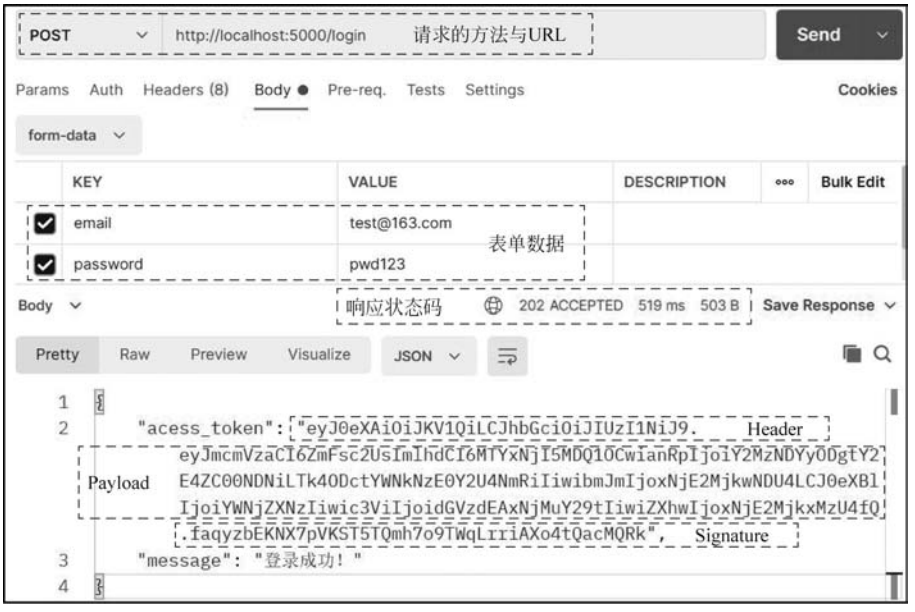

图 5.8　用 Postman 提交用户登录数据并接收 Token


视频讲解

5.10　发送邮件找回密码

应用系统经常需要向用户自动发送邮件,来实现找回密码、激活账户以及推送某些信息。

Python 标准库自带的 smtplib 包可用于发送电子邮件,基于 smtplib 实现的 Flask-Mail 邮件扩展框架提供了更为简单的 API 接口,与 Flask 无缝集成,只需在应用程序中设置好邮件服务器的 SMTP 参数,即可轻松编写发送邮件的脚本。

在项目虚拟环境执行命令 pip install flask-mail,安装 Flask-Mail 扩展库。

实践中密码是加密存储的,原则上不允许找回原密码,但可以根据收到的验证码去设置新密码,程序段 P5.6 为了演示发送邮件的逻辑,暂且允许用户找回原密码,根据用户提供的邮箱,向用户发送一封包含密码信息的邮件。

P5.6 # 发送邮件,找回密码

```
from flask_mail import Mail, Message
app.config['MAIL_SERVER'] = 'smtp.163.com'          # 发送邮件的服务器,需要启用 SMTP 服务
app.config['MAIL_PORT'] = 465                       # 采用 SSL 通信的端口,否则为 25
app.config['MAIL_USE_TLS'] = False
app.config['MAIL_USE_SSL'] = True                   # 采用 SSL 通信
app.config['MAIL_USERNAME'] = 'your_webmaster'  # 用户账号,此处为示例
app.config['MAIL_PASSWORD'] = 'ANURWYMPLMNDEFXLQK'   # SMTP 授权码,此处为示例
mail = Mail(app)                                    # 创建邮件服务对象
@app.route('/get_password', methods = ['post'])
def get_password():
    email = request.form['email']
    conn = lite.connect(db_path)                    # 打开数据库
    with conn:
        cur = conn.cursor()
        sql = f"select email, password from users where email = '{email}'"
        cur.execute(sql)
        records = cur.fetchall()
        if records:                                 # 允许找回
            password = records[0][1]
            msg = Message(f'找回的密码是:{password}',
                          sender = 'your_webmaster@163.com',
                          recipients = [email])# 定义邮件消息
            mail.send(msg)                          # 发送邮件
            return jsonify(message = f'密码已发送至邮箱:{email}'), 202
        else:                                       # 拒绝找回密码
            return jsonify(message = '用于找回密码的 Email 地址不正确!'), 402
```


视频讲解

5.11 查询记录

程序段 P5.7 根据指定的 ID 对 apples 数据表进行查询,返回记录详细信息。

P5.7 # 根据 ID 查询苹果数据集单条记录

```
@app.route('/get_details/<int:apple_id>', methods = ['get'])
def get_details(apple_id:int):
    conn = lite.connect(db_path) # 打开数据库
    with conn:
        cur = conn.cursor()
        sql = f"select id,name,feature,regular,cure,img_url from apples where id = '{apple_id}'"
        cur.execute(sql)
        records = cur.fetchall()
        if records:                 # 找到记录
            record = dict(zip(['id', 'name', 'feature', 'regular', 'cure', 'img_url'], records[0]))
            return jsonify(record), 200
        else:                       # 没找到
            return jsonify(message = f'ID 为{apple_id}的记录不存在!'), 403
```

程序段 P5.8 对 apples 数据表进行查询,返回所有记录详细信息。

```
P5.8  # 查询苹果数据集所有记录
@app.route('/get_all_details', methods = ['get'])
def get_all_details():
    all_records = []
    conn = lite.connect(db_path) # 打开数据库
    with conn:
        cur = conn.cursor()
        sql = f"select * from apples"
        cur.execute(sql)
        records = cur.fetchall()
        if records:                # 找到记录
            for record in records:
                item = dict(zip(['id', 'name', 'feature', 'regular', 'cure', 'img_url'], record))
                all_records.append(item)
            return jsonify(all_records), 200
        else:                      # 没找到
            return jsonify(message = f'数据集为空!'), 404
```


视频讲解

5.12 添加记录

程序段 P5.9 向 apples 数据表添加一条新记录。第 2 行语句要求用户提供身份验证，即提供其合法的 JSON Web Token 才能访问该模块，完成添加新数据的操作。

```
P5.9  # 添加记录
@app.route('/add_apple', methods = ['post'])
@jwt_required()                    # 需要身份验证
def add_apple():
    name = request.form['name']
    conn = lite.connect(db_path) # 打开数据库
    with conn:
        cur = conn.cursor()
        sql = f"select count(name) from apples where name = '{name}'"
        cur.execute(sql)
        count = cur.fetchone()[0]
        if count > 0:              # 记录已存在
            return jsonify(message = '添加的记录已存在!'), 409
        else:
            feature = request.form['feature']
            regular = request.form['regular']
            cure = request.form['cure']
            img_url = request.form['img_url']
            sql = f"insert into apples(name, feature, regular, cure, img_url) " \
                  f"values ('{name}', '{feature}', '{regular}', '{cure}', '{img_url}')"
            cur.execute(sql)
            return jsonify(message = '成功添加新记录!'), 201
```

在 Postman 中做应用场景测试，观察反馈的输出结果，验证模块逻辑的正确性。

5.13 更新记录


视频讲解

程序段 P5.10 更新 apples 数据表中的记录。第 2 行语句要求用户提供身份验证，即提供其合法的 JSON Web Token 才能访问该模块，完成数据修改操作。

P5.10 # 更新记录

```
01 @app.route('/update_apple', methods=['post'])
02 @jwt_required()                    # 需要身份验证
03 def update_apple():
04     id = request.form['id']
05     conn = lite.connect(db_path) # 打开数据库
06     with conn:
07         cur = conn.cursor()
08         sql = f"select count(id) from apples where id='{id}'"
09         cur.execute(sql)
10         count = cur.fetchone()[0]
11         if count <= 0:              # 记录不存在
12             return jsonify(message='修改的记录不存在!'), 409
13         else:
14             name = request.form['name']
15             feature = request.form['feature']
16             regular = request.form['regular']
17             cure = request.form['cure']
18             img_url = request.form['img_url']
19             sql = f"update apples set name = '{name}', feature = '{feature}', " \
20                   f"regular = '{regular}', cure = '{cure}', img_url = '{img_url}'" \
21                   f"where id = '{id}'"
22             cur.execute(sql)
23             return jsonify(message='成功修改记录!'), 201
```

在 Postman 中做应用场景测试，观察反馈的输出结果，验证模块逻辑的正确性。

5.14 删除记录


视频讲解

从数据库中删除记录，一般有两种策略：一种策略是不做数据的物理删除，而是添加一个新字段，用以标记记录已被删除，删除的数据可被恢复；另一种策略是直接物理删除数据。为简单起见，程序段 P5.11 直接物理删除 apples 数据表中的记录。第 2 行语句要求用户提供身份验证，即提供其合法的 JSON Web Token 才能访问该模块，完成记录删除操作。

P5.11 # 删除记录

```
01 @app.route('/delete_apple/<int:id>', methods=['delete'])
02 @jwt_required()                    # 需要身份验证
03 def delete_apple(id:int):
04     conn = lite.connect(db_path)    # 打开数据库
05     with conn:
06         cur = conn.cursor()
```

```
        sql = f"select count(id) from apples where id = '{id}'"
        cur.execute(sql)
        count = cur.fetchone()[0]
        if count <= 0:                    # 记录不存在
            return jsonify(message = '删除的记录不存在!'), 404
        else:
            sql = f"delete from apples where id = '{id}'"
            cur.execute(sql)
            return jsonify(message = '成功删除记录!'), 202
```

在 Postman 中做应用场景测试,观察反馈的输出结果,验证模块逻辑的正确性。

视频讲解

5.15 分类预测

程序段 P5.12 定义了一个 Web 预测服务模块。

P5.12 # 定义模型预测服务

```
import io
import base64
import numpy as np
from PIL import Image
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import img_to_array
model = load_model('./models/densenet121.h5')                    # 加载模型
# 图像预处理
def preprocess_image(image, target_size):
    if image.mode != 'RGB':
        image = image.convert('RGB')
    image = image.resize(target_size)
    image = img_to_array(image)
    image = image / 255.
    image = np.expand_dims(image, axis = 0)
    return image
# 模型预测
@app.route('/predict', methods = ['post'])
def predict():
    message = request.get_json(force = True)
    image = message['image']
    decode_image = base64.b64decode(image)
    image = Image.open(io.BytesIO(decode_image))
    processed_image = preprocess_image(image, target_size = (512, 512))
    prediction = model.predict(processed_image)[0].tolist()      # 预测
    response = {
        'prediction': {
            'healthy': prediction[0],
            'multiple_diseases': prediction[1],
            'rust': prediction[2],
            'scab': prediction[3]
        }
    }
    return jsonify(response), 200
```

5.16 前端页面


视频讲解

程序段 P5.13 实现 Web 前端页面的基本逻辑。

P5.13 # Web 前端页面设计

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>苹果树病虫害预测</title>
</head>
<body>
<input id='image-selector' type='file'>
<button id='predict-button' type='button'>预 测</button>
<p style="font-weight: bold">预测结果:</p>
<p>healthy:<span id="healthy"></span></p>
<p>multiple_diseases:<span id="multiple_diseases"></span></p>
<p>rust:<span id="rust"></span></p>
<p>scab:<span id="scab"></span></p>
<img id="selected-image" src=""/>
<script src="jquery-3.6.0.min.js"></script>
<script>
    let base64Image;
    <!-- 选择图片 -->
    $('#image-selector').change(function () {
        let reader = new FileReader();
        reader.onload = function (e) {
            let dataUrl = reader.result;
            $('#selected-image').attr('src', dataUrl);
            base64Image = dataUrl.replace('', '');
            console.log(base64Image)
        } <!-- end on load -->
        reader.readAsDataURL($('#image-selector')[0].files[0]);
        $('#healthy').text('');
        $('#multiple_diseases').text('');
        $('#rust').text('');
        $('#scab').text('');
    }); <!-- end on change -->
    <!-- 单击"预测"按钮 -->
    $('#predict-button').click(function (event) {
        let message = {
            image: base64Image
        }
        console.log(message)
        $.post('http://localhost:5000/predict', JSON.stringify(message), function (response) {
            $('#healthy').text(response.prediction.healthy.toFixed(6));
            $('#multiple_diseases').text(response.prediction.multiple_diseases.toFixed(6));
            $('#rust').text(response.prediction.rust.toFixed(6));
            $('#scab').text(response.prediction.scab.toFixed(6));
        });
    });
</script>
```

```
48  </body>
49  </html>
```

启动服务器，打开浏览器，在地址栏中输入预测页面地址 http://localhost:5000/static/predict.html，从测试集目录中任意选择一幅测试图片，单击“预测”按钮，观察服务器返回的预测结果，如图 5.9 所示。

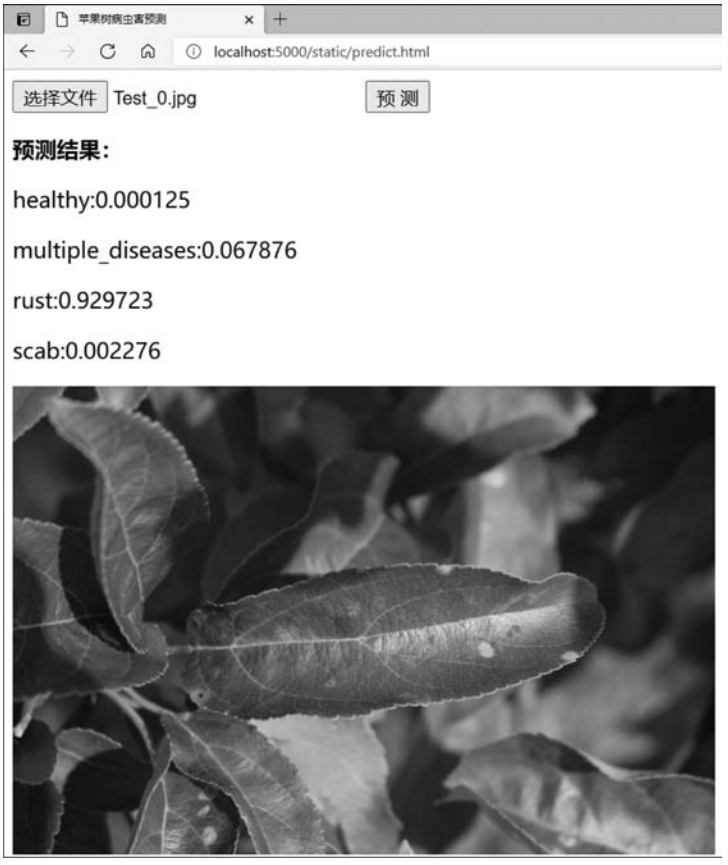

图 5.9　前端页面测试结果

本章设计的 Web 服务器已经部署于腾讯云服务器，读者可以访问页面 http://120.53.107.28/static/predict.html 做应用测试。


视频讲解

5.17　小结

本章以第 3 章网络爬虫生成的数据库 apple.db 为基础，运用 Flask 构建 Web 服务框架，实现了用户注册，登录，邮件找回密码，对数据库的增、删、改、查，基于 JSON Web Token 的认证，全面介绍了基于 RESTful 风格的 Web API 设计方法。

基于第 4 章完成的 DenseNet121 训练模型，定义了 Web API 预测服务，通过与前端页面的联合测试，展示了智能 Web App 的设计原理与方法，项目的可扩展性好，易于部署推广，应用价值高。

5.18 习题

一、简答题

1. 智能 Web App 将智能模型部署于服务器端，其优点是什么？缺点是什么？

2. Flask 作为基于 Python 的轻量级 Web 框架，有哪些特点？

3. Postman 作为模拟 HTTP 请求的客户端调试工具，有哪些优点？

4. DB Browser for SQLite 作为 SQLite 数据库可视化管理工具，有哪些优点？

5. 简述基于 Flask 框架定义的主程序和 Web API 服务的一般语法形式。

6. 如何实现 Web API 服务与客户机之间的 JSON 数据交换？

7. 简述 HTTP 状态码的分类及其含义。

8. 客户端可以通过在 URL 末尾附加参数的方式向服务器提交数据，简述 URL 参数的一般语法形式。

9. 描述用 DB Browser for SQLite 创建用户数据表的基本步骤。

10. 在 Web 服务器端，如何区分客户机提交的请求类型是 get 还是 post？

11. JSON Web Token 的基本结构是什么？

12. 发送邮件，既可以采用 Python 自带的库模块 smtplib，也可以采用第三方扩展库模块 Flask-Mail。描述用 Flask-Mail 发送邮件的基本步骤。

13. 前端页面将待发送的图片进行 Base64 编码后再发给服务器，简述 Base64 编码的特点。

14. 简述以 JSON 格式表达数据的优点。

二、编程题

修改客户机向服务器发送图片的逻辑，不采用 Base64 编码，而是直接将图片的字节流数据发送到服务器，服务器端的接收逻辑也需要做出同步修改。