第3章 CHAPTER 3 机器视觉应用

本章主要介绍在树莓派5上使用OpenCV实现计算机视觉领域中的典型应用，例如，目标跟踪、人脸检测、手势识别、文字识别以及通过DNN模块加载深度学习模型完成目标检测等。

3.1 OpenCV简介

OpenCV是一个开源、功能强大且易于上手的计算机视觉库，它集成了大量针对图像和视频处理、特征提取、目标检测等多种任务的算法和工具，在机器人视觉、视频监控、人机交互等众多领域均得到广泛应用。此外，OpenCV的新版本中还加入了深度神经网络相关的功能模块，进一步拓展了其应用场景。

3.1.1 安装OpenCV-Python库

在Python编程环境下借助OpenCV可以高效便捷地实现各种基于树莓派的机器视觉任务。OpenCV的Python库包括opencv-python和opencv-contrib-python两个版本，两者都为OpenCV提供了Python接口，但区别在于，前者只包含OpenCV的主要模块和功能，而后者是前者的扩展，还包括一些实验性的算法和功能。为适应大多数情况，建议安装opencv-contrib-python。在树莓派虚拟环境中使用pip来安装OpenCV的命令是**pip install opencv-contrib-python**。安装完成后，在终端输入如图3-1所示的命令来验证OpenCV是否安装成功。如果能够顺利导入并显示OpenCV版本信息，则表示安装成功；如果执行import cv2语句报错，一般是因为缺少某些必要的依赖项，可以根据错误提示补充安装对应的缺失库。

```
(env1) pi5@raspberrypi:~/MyProject $ python
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> cv2.__version__
'4.9.0'
>>>
```

图3-1 成功安装OpenCV

3.1.2 OpenCV库基本操作

下面从图像读取、颜色空间转换、图像显示和保存等基本操作入手来介绍OpenCV的

使用方法。新建脚本 opencv_img.py，输入以下代码。

```
import cv2

img = cv2.imread('car_plate.jpg')
height, width = img.shape[:2]                          # 获取图像的高度与宽度
resized = cv2.resize(img, (320,240))                   # 调整图像尺寸
gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)       # 转换为灰度图像
cv2.imshow('Image', gray)
cv2.waitKey(0)                                         # 等待按键，参数 0 表示无限期等待
cv2.imwrite('gray.jpg', gray)                          # 保存图像
cv2.destroyAllWindows()                                # 销毁 OpenCV 创建的所有窗口
```

运行脚本，读取的图像进行尺度变换后再转换为灰度图像并保存在当前目录下。类似地，新建脚本 opencv_vid.py，输入以下内容可以实现视频帧的读取、显示，通过按键交互可以将当前帧保存为图像文件。

```
import cv2

cap = cv2.VideoCapture(0)                              # 创建视频捕获对象
#cap = cv2.VideoCapture("test.flv")                    # 打开本地视频
cap.set(3, 640)                                        #设置摄像头参数，3 为宽 4 为高
cap.set(4, 480)
while cap.isOpened():
    ret, frame = cap.read()                            # 逐帧读取视频
    if not ret:
        break
    resized = cv2.resize(frame, None, fx=0.5, fy=0.5) # 宽度和高度缩小为原来的一半
    cv2.imshow('Image', resized)
    k = cv2.waitKey(1)                                 # 等待按键，返回按键的 ASCII 码
    if k == 27:                                        # 按 Esc 键退出
        break
    elif k == ord('s'):                                # 按 s 键保存图像
        cv2.imwrite('image.jpg', resized)
cap.release()                                          # 释放摄像头资源
cv2.destroyAllWindows()
```

需要指出的是，树莓派 5 无法直接通过 OpenCV 调用 CSI 摄像头，上例中视频读取采用的是 USB 摄像头。此外，OpenCV 默认采用 BGR 颜色空间，这与常规的 RGB 颜色通道存储顺序有所不同。当 OpenCV 与其他图像处理库（如 Pillow）结合使用时，需要进行颜色空间的转换。Pillow 也是 Python 中非常常用且功能强大的图像处理库，可以通过 **pip install Pillow** 命令进行安装。Pillow 与 OpenCV 库的区别在于，前者专注于图像处理，提供了丰富的图像文件操作功能；而后者更侧重于计算机视觉，不仅包含基本的图像处理功能，还提供了大量的图像处理和计算机视觉算法。

3.2 颜色检测

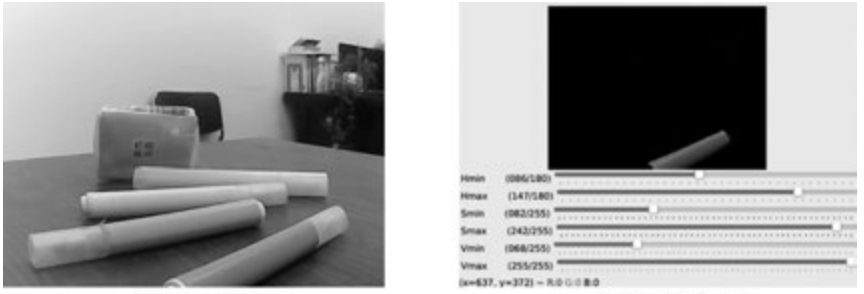
颜色检测是对图像或视频中的颜色进行识别和区分的过程，在印刷、涂料、纺织等行业发挥着关键作用。通过颜色检测，也可以自动识别和分类具有特定颜色的物体，以便进行后

续的处理、分类、跟踪等任务。然而，颜色检测的准确性易受光照条件变化、物体表面反射、阴影以及颜色相似性干扰的影响。在计算机视觉系统中，颜色通常以不同的颜色空间来表示。RGB 颜色空间虽然直观且应用广泛，但它对光照条件的变化尤为敏感。相比之下，HSV 对光照变化的鲁棒性更强，成为了颜色检测任务中的更优选择。

本节将介绍树莓派对特定颜色目标或图像区域的检测与提取过程。实现步骤是，使用 OpenCV 库来捕获摄像头的视频流，并通过滑动条来动态调整 HSV 阈值，以便在图像中检测特定颜色的对象。新建脚本 hsv_threshold.py，输入以下代码。

```
import cv2
import numpy as np

def nothing(x):                                    # 定义回调函数
    pass

# 创建窗口和滑动条
cv2.namedWindow('Trackbars',cv2.WINDOW_NORMAL)
cv2.createTrackbar('Hmin', 'Trackbars', 50, 180, nothing)
cv2.createTrackbar('Hmax', 'Trackbars', 100, 180, nothing)
cv2.createTrackbar('Smin', 'Trackbars', 100, 255, nothing)
cv2.createTrackbar('Smax', 'Trackbars', 255, 255, nothing)
cv2.createTrackbar('Vmin', 'Trackbars', 100, 255, nothing)
cv2.createTrackbar('Vmax', 'Trackbars', 255, 255, nothing)

# 初始化摄像头
cap = cv2.VideoCapture(0)              #摄像头索引可以通过 v4l2-ctl --list-devices 查看

def color_threshold():
    # 获取滑动条的当前值
    h_low = cv2.getTrackbarPos('Hmin', 'Trackbars')
    s_low = cv2.getTrackbarPos('Smin', 'Trackbars')
    v_low = cv2.getTrackbarPos('Vmin', 'Trackbars')
    h_up = cv2.getTrackbarPos('Hmax', 'Trackbars')
    s_up = cv2.getTrackbarPos('Smax', 'Trackbars')
    v_up = cv2.getTrackbarPos('Vmax', 'Trackbars')
    _, frame = cap.read()                          # 读取当前帧并显示
    #cv2.imshow('image', frame)
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV) # 转换到 HSV 空间
    # 通过 HSV 阈值来创建掩膜
    lower = np.array([h_low, s_low, v_low])
    upper = np.array([h_up, s_up, v_up])
    mask = cv2.inRange(hsv, lower, upper)
    res = cv2.bitwise_and(frame, frame, mask=mask) # 对原图像和掩码进行位运算
    cv2.imshow('Trackbars', res)                   # 显示结果

try:
    while True:
        color_threshold()                          # 调用回调函数来更新阈值并显示检测结果
        if cv2.waitKey(1) & 0xFF == ord('q'):      # 按 q 键退出
            break
except KeyboardInterrupt:                          # 释放摄像头并关闭窗口
    cap.release()
    cv2.destroyAllWindows()
```

上述代码创建一个窗口以及 6 个滑动条来分别控制 HSV 的最小和最大阈值，并为滑动条定义回调函数 nothing，该函数不做任何操作。color_threshold()函数获取滑动条的 HSV 阈值，并以此创建一个掩码，对摄像头读取的当前帧进行位运算，得到只包含特定颜色目标的图像。运行脚本，通过操作滑动条来调整 HSV 阈值，可以看到如图 3-2 所示的检测结果。当按下 q 键时，程序将退出循环，释放摄像头资源并关闭所有窗口。



(a) 原始图像　　(b) 特定颜色目标

图 3-2　颜色检测结果

3.3　目标跟踪

目标跟踪是计算机视觉领域中的一个重要分支，专注于在连续的视频序列中准确识别和定位目标对象，并对其进行追踪，已在视频监控、自动驾驶、运动分析、增强现实等领域得到广泛应用。目标跟踪的一般步骤是：首先，在视频序列的第一帧中，借助目标检测算法或人工选定方式来确定感兴趣的目标区域，并对此目标进行特征提取和建模；随后，在紧接着的视频帧中，启用跟踪算法以搜索并锁定目标位置，根据新的观测数据更新目标模型；最终，输出目标的连续跟踪结果。当前，目标跟踪技术仍面临一些困难和挑战，如目标形态变化、目标遮挡与快速移动、光照变化以及背景干扰等。

在树莓派上，使用 OpenCV 库能够高效且便捷地完成目标跟踪任务。OpenCV 提供了多样化的目标跟踪器，如 BOOSTING、MIL、KCF、TLD 和 MedianFlow 等。通过选择合适的跟踪算法，可以轻松构建出适用于各种应用场景的目标跟踪系统。下面介绍使用指定算法来实现运动目标跟踪的具体过程。新建脚本 object_tracking.py，输入以下代码。

```
import cv2
import numpy as np
import argparse

# 鼠标选择跟踪目标
def object_rectangle(event, x, y, flag, param):
    global bbox
    if event == cv2.EVENT_LBUTTONDOWN:
        bbox[0], bbox[1] = x, y
    elif event == cv2.EVENT_MOUSEMOVE and flag == cv2.EVENT_FLAG_LBUTTON:
        bbox[2], bbox[3] = x - bbox[0], y - bbox[1]
    elif event == cv2.EVENT_LBUTTONUP:
        bbox[2], bbox[3] = x - bbox[0], y - bbox[1] # bbox 格式(x, y, width, height)
```

```
# 跟踪器
Object_Trackers = {
    "csrt": cv2.TrackerCSRT_create,
    "kcf": cv2.TrackerKCF_create,
    "boosting": cv2.legacy.TrackerBoosting_create,
    "mil": cv2.TrackerMIL_create,
    "tld": cv2.legacy.TrackerTLD_create,
    "medianflow": cv2.legacy.TrackerMedianFlow_create,
    "mosse": cv2.legacy.TrackerMOSSE_create}

if __name__ == "__main__":
    # 通过参数选择跟踪器,默认为 KCF 跟踪器
    parser = argparse.ArgumentParser()
    parser.add_argument('-t', "--tracker", type=str, default='kcf')
    args = parser.parse_args()
    cap = cv2.VideoCapture(0)                                    # 打开摄像头
    #cap = cv2.VideoCapture('test.flv')                          #加载本地视频
    cap.set(3, 352)                                              # 设定摄像头分辨率为 352×288 像素
    cap.set(4, 288)
    cap.set(5, 10)                                               # 设定摄像头帧率
    framecount = 0
    bbox = np.zeros((4,),dtype=np.uint16)
    tracker = Object_Trackers[args.tracker]()                    # 通过 -t 参数指定跟踪器
    while cap.isOpened():
        ret, frame = cap.read()                                  # 读取新的帧
        if not ret:
            break
        if framecount == 0:                                      # 第一帧需要选择跟踪目标
            cv2.namedWindow("Object Tracking",cv2.WINDOW_NORMAL)
            cv2.imshow('Object Tracking', frame)
            # 在第一帧中手动选择目标区域
            cv2.setMouseCallback('Object Tracking',object_rectangle)
            cv2.waitKey(0)
            tracker.init(frame, bbox)                            # 初始化跟踪器
        else:
            timer = cv2.getTickCount()                           # 开始计时
            ok, bbox = tracker.update(frame)                     # 更新跟踪器
            print("目标框位置:",bbox)
            fps = cv2.getTickFrequency()/(cv2.getTickCount()-timer)    # 计算 FPS
            if ok:                                               # 跟踪成功
                p1 = (int(bbox[0]), int(bbox[1]))
                p2 = (int(bbox[0]+bbox[2]), int(bbox[1]+bbox[3]))
                cv2.rectangle(frame, p1, p2, (255,0,0), 2)
            else :                                               # 跟踪失败
                cv2.putText(frame, "Tracking failure", (50,100), cv2.FONT_HERSHEY_SIMPLEX,
1,(0,0,255),3)
            cv2.putText(frame, "FPS:" + str(int(fps)), (50,50), cv2.FONT_HERSHEY_SIMPLEX,
1, (0,0,255), 3)                                                 # 显示 FPS
            cv2.imshow('Object Tracking', frame)                 # 显示结果图像
            if cv2.waitKey(1) & 0xFF == ord('q'):                # 退出条件
                break
        framecount = framecount+1
    # 释放资源并关闭窗口
    cap.release()
    cv2.destroyAllWindows()
```

上例中，首先初始化一个视频捕获对象，可以是摄像头实时捕获的视频流，也可以是本地的视频文件。接着，选定并初始化跟踪器，在视频序列的第一帧中手动框选跟踪目标。随后，在一个循环中不断更新跟踪器并显示帧率、目标框等信息。在终端输入 **python object_tracking. py-t kcf**（不带参数时默认为 KCF 跟踪器）运行脚本，图 3-3(a)展示了通过摄像头实时捕获视频并对移动目标进行跟踪的结果，而图 3-3(b)则是读取本地视频并对选定目标进行跟踪的结果。当被跟踪目标移出视野范围时，由于跟踪器无法再获取目标的有效信息，因此会显示跟踪失败的信息。

(a) 摄像头目标跟踪　　(b) 本地视频目标跟踪

图 3-3　目标跟踪结果

3.4　人脸检测与追踪

人脸检测作为计算机视觉领域的一项核心任务，专注于在图像或视频中快速准确地定位人脸的位置，进而将其与背景区分开来。该技术的核心在于判断图像中是否存在人脸以及确定人脸的位置和尺寸大小。基于 OpenCV 进行人脸检测的常用方法是使用预训练的 Haar 级联分类器识别图像中的人脸区域。在树莓派上安装 OpenCV 后，其 cv2/data/目录下会包含一系列用于描述人脸特征的 XML 文件，如 haarcascade_frontalface_default. xml、haarcascade_frontalface_alt. xml 等，它们保存的是具有普适性的 Haar 人脸检测分类器。此外，OpenCV 还提供了其他类型的预训练分类器，以满足更精细化的面部特征检测需求。这些分类器涵盖了眼睛、鼻子等关键面部特征的检测器，用户可以根据需要进行选择和使用。

OpenCV 检测人脸的过程比较简单，首先加载 XML 文件生成级联分类器，然后通过该分类器对灰度图像进行多尺度检测，得到人脸矩形框的左上角坐标和宽高，最后标记出人脸区域。对于更高级的应用，如使用树莓派控制二自由度云台来追踪人脸，可以先按照上述步骤获取视频帧中的人脸位置，通过动态调整云台来控制摄像头跟踪人脸，并确保人脸始终处于画面的正中央。二自由度云台由两个舵机构成，并将 USB 摄像头固定在云台上，如图 3-4(a)所示。为了平稳流畅地实现人脸追踪，这里采用 PCA9685 驱动板来控制舵机的转动。

PCA9685 是一款高性能的 16 通道 PWM 控制器，具备 12 位高分辨率输出能力，使得它能够实现极为精细且准确的输出控制，该控制器的实物如图 3-4(b)所示。PCA9685 使用标准的 I^2C 串行总线接口与主控设备通信，4 个引脚 GND、SCL、SDA 和 VCC 分别与树莓派 GPIO 端口的 GND(6 脚)、SCL(3 脚)、SDA(2 脚)、3. 3V(1 脚)相连。需要注意的是，PCA9685 的 VCC 引脚仅为芯片供电，而 V+引脚则用于为所连接的舵机供电，它可以与树

莓派的5V引脚相连，也支持通过电源接线柱外接电源。连接好引脚后，在终端输入 **sudo i2cdetect -y 1**，可以查看到PCA9685的设备地址为0x40。为了实现对PCA9685的有效控制，树莓派需要使能I^2C接口，并通过执行 **pip install adafruit-pca9685** 命令安装相应的Python库。将舵机的旋转角度转换为PWM脉冲宽度，调用库函数就可以控制舵机旋转到相应的位置。

(a) 云台摄像头　　(b) PCA9685驱动板

图 3-4　二自由度云台跟踪系统

下面介绍树莓派控制云台摄像头来追踪人脸的实现过程。新建脚本face_tracking.py，输入以下代码。

```
import cv2
import time
import board
import Adafruit_PCA9685

Th = 30                                  # 人脸框中心与图像中心可接受的偏移像素
h_angle = 90                             # 舵机初始角度
v_angle = 90
# 初始化PCA9685，设置I²C地址(默认为0x40)和I²C总线号
pca = Adafruit_PCA9685.PCA9685(address = 0x40, busnum = 1)
pca.set_pwm_freq(50)                     # 设置PWM频率为50Hz

# 舵机连接在PCA9685的通道0和1
h_servo_channel = 0                      # 水平舵机
v_servo_channel = 1                      # 垂直舵机

def set_servo_angle(channel, angle):     # 按12位精度将角度转换成脉冲长度
    pulse = int(4096 * ((angle * 11) + 500) / 20000 + 0.5)   # 脉冲长度四舍五入
    pca.set_pwm(channel, 0, pulse)       # 控制特定通道的PWM输出

'''XML文件的完整路径为'/home/pi5/MyProject/env1/lib/python3.11/site-packages/cv2/data',
根据情况自行修改，也可以将其复制到当前目录下使用'''
face_cascade = cv2.CascadeClassifier('./haarcascade_frontalface_alt.xml')

cap = cv2.VideoCapture(0)                # 开启摄像头
cap.set(3,352)                           # 设置宽高和帧率
cap.set(4,288)
cap.set(5, 15)
# 舵机调整到初始位置
set_servo_angle(h_servo_channel, h_angle)
```

```
set_servo_angle(v_servo_channel, v_angle)

while cap.isOpened():
    ret, frame = cap.read()                          # 获取一帧图像
    if not ret:
        continue
    # 将图像转换为灰度图像
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # 多尺度人脸检测
    faces = face_cascade.detectMultiScale(gray, 1.3, minNeighbors=5)
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (255,0,0),2) # 蓝色框标记人脸
        center_x, center_y = x + w/2, y + h/2                # 人脸中心坐标
        # 计算人脸中心与图像中心的偏移量
        offset_x = center_x - frame.shape[1]/2
        offset_y = center_y - frame.shape[0]/2
        if abs(offset_x) > Th:                               # 小于30像素时舵机不动
            h_angle = h_angle - int(offset_x/50)        # 根据偏移量,舵机调整适当的角度
            if h_angle > 180:
               h_angle = 180
            if h_angle < 0:
               h_angle = 0
        if abs(offset_y) > Th:
            v_angle = v_angle + int(offset_y/50)          # 参数50可以根据实际情况修改
            if v_angle > 180:
               v_angle = 180
            if v_angle < 0:
               v_angle = 0
        # 发送指令给舵机控制器,调整舵机角度
        set_servo_angle(h_servo_channel, h_angle)
        set_servo_angle(v_servo_channel, v_angle)
        break
    cv2.imshow('Face Tracking', frame)                       # 显示图像
    if cv2.waitKey(1) & 0xff == ord("q"):
        break
# 释放摄像头并关闭所有窗口
cap.release()
cv2.destroyAllWindows()
```

上例中，摄像头循环读取视频帧，通过多尺度人脸检测得到人脸框的坐标信息，调整detectMultiScale()函数的参数（如缩放比例和最小邻域大小）可以优化人脸检测的效果。根据人脸中心位置与图像中心的偏移量，设定舵机需要调整的角度。为避免舵机频繁抖动，当偏移量小于某个阈值时，舵机角度不做调整。根据计算出的调整角度，set_servo_angle()函数先将角度转换为PWM脉冲宽度，随后分别控制负责水平和垂直方向舵机的通道输出相应的PWM信号，使舵机旋转指定的角度。运行该脚本，人脸追踪结果如图3-5所示，当移动手机时，摄像头能够自动追踪人脸。用户按q键即可退出循环并释放摄像头资源。

图3-5 人脸追踪结果

3.5 人脸识别

人脸检测通常被视为一个二分类任务，即判断给定的图像区域是否包含人脸。相对而言，人脸识别则是一个多分类问题，需要将图像或视频中检测到的人脸与已知人脸进行匹配，从而确定人脸的身份。两者的主要区别在于，人脸检测侧重人脸区域的定位和提取，而人脸识别则着重于提取人脸特征，并与已有的人脸特征进行精确比对。

局部二值模式直方图人脸识别器(LBPHFaceRecognizer)是 OpenCV 库中一种常用的人脸识别方法，它通过计算人脸区域每个像素点的局部二值模式直方图来提取特征，并利用这些特征进行人脸识别。该方法的步骤相对烦琐，首先，使用一组包含人脸的图像数据集和对应的标签来训练 LBPHFaceRecognizer 识别器；其次，需要使用 OpenCV 提供的 Haar 级联分类器或深度神经网络模型对待识别图像中的人脸进行检测与提取；最后，对于检测到的每个人脸区域，利用已训练好的 LBPHFaceRecognizer 模型进行人脸识别，并输出标签(人脸身份)和置信度。

采用 Python 第三方库 face_recognition 可以非常方便地实现人脸查找与面部识别，准确率也非常高。face_recognition 库基于机器学习开源库 dlib，提供了先进的人脸检测与人脸编码功能以及预训练的面部识别模型。在虚拟环境终端输入命令 **pip install face_recognition** 完成 face_recognition 库的安装，安装过程中会自动安装 dlib、face-recognition-models 等依赖库。

注意：如果安装过程出现关于 dlib 的报错，可以先安装指定版本的 dlib 库，如 pip install dlib==19.24.2，然后再通过 pip install face_recognition 安装 face_recognition 库。

下面举例说明基于 face_recognition 库实现人脸识别的具体过程。创建文件夹 known_faces，存放所有已知人脸的一张图像，图像命名为对应的人名，如图 3-6(a)所示。通过对已知人脸图像进行面部编码，与待检测的人脸进行比对，完成人脸识别。新建脚本 face_recognizer.py，输入以下内容。

```
import os
import cv2
import face_recognition

# 加载已知人脸的图像和标签
known_face_encodings = []
known_face_names = []
path = "./known_faces"                                    # 已知人脸所在的文件夹

# 从路径下加载每个人的人脸图像，进行面部编码
imagePaths = [os.path.join(path,f) for f in os.listdir(path)]
for imagePath in imagePaths:
    name = os.path.split(imagePath)[-1].split(".")[0]  # 从绝对目录中提取出图像名
    image = face_recognition.load_image_file(imagePath)  # 加载图像
    # 获取面部编码
    face_encodings = face_recognition.face_encodings(image)
```

```
        if len(face_encodings) > 0:
            # 取第一个面部编码(假设每个图像只有一个面部)
            known_face_encodings.append(face_encodings[0])
            known_face_names.append(name)              # 图名为人脸标签

unknown_image = cv2.imread("dhxy.jpg")                 # 加载要检测的图像
# 获取未知图片中所有面部的编码
unknown_face_encodings = face_recognition.face_encodings(unknown_image)
# 对未知面部的编码进行遍历
for unknown_face_encoding in unknown_face_encodings:
    # 尝试匹配已知面部的编码
    matches = face_recognition.compare_faces(known_face_encodings, unknown_face_encoding,
tolerance=0.5)                                         # tolerance参数值越小,人脸对比过程越严格
    name = "Unknown"
    # 如果匹配成功
    if True in matches:
        # 找到第一个匹配成功的索引
        first_match_index = matches.index(True)
        name = known_face_names[first_match_index]
    # 在未知图片上画出检测到的面部并显示名称
    (top, right, bottom, left) = face_recognition.face_locations(unknown_image)[0]
    cv2.rectangle(unknown_image, (left, top), (right, bottom), (0, 255, 0), 2)
    cv2.putText(unknown_image, name, (left + 6, bottom - 6), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(255, 255, 255), 1)
# 显示图片
cv2.imshow('Unknown Face', unknown_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

上述代码中，首先从已知人脸所在的文件夹加载所有人脸图片和标签，调用 face_recognition.face_encodings()函数为已知人脸生成面部编码。然后，加载读取包含未知人脸的图片，并提取其中的面部编码。通过 face_recognition.compare_faces()函数比较未知人脸编码和已知人脸编码，找到匹配的人脸并显示其名称，如果未找到匹配的人脸，则标注为“Unknown”。实际应用中，可以通过减少 tolerance 参数的值(默认为 0.6)使得人脸对比过程更加严格，从而提高人脸识别的准确度。图 3-6(b)为待检测图片及其人脸识别的结果，与图 3-6(a)中已知的人脸标签相符。

(a) known_faces文件夹内容　　(b) 待识别图片及结果

图 3-6　人脸识别

3.6　手势识别

手势识别旨在精准捕捉人类的手势动作，并将这些动作转换为具有明确语义的命令。这一技术赋予了用户通过简单手势操控设备或与设备进行互动的能力，成为当前一种非常

直观且高效的人机交互方式。

3.6.1　手部关键点检测

MediaPipe 是由 Google 开发并开源的一款跨平台多媒体处理框架，适合在移动设备、边缘终端和云端上快速构建和部署机器学习解决方案。它包含了人脸检测、人脸关键点、手势识别、头像分割和姿态识别等多种模型和工具库，其中 MediaPipe 的手部关键点检测模块能够实时识别和追踪双手的 21 个关键点，包括各手指的关节位置，如图 3-7 所示。

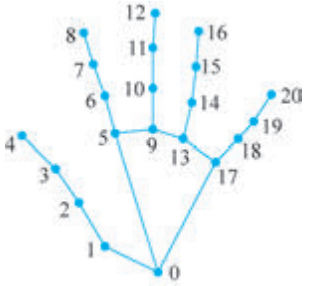


图 3-7　手部关键点

下面介绍使用 MediaPipe 的 Hands 模块来实现视频流中手部关键点的检测。首先在虚拟环境中通过 **pip install mediapipe** 命令安装 MediaPipe 库，接下来只需要实例化一个手部关键点检测对象，传入视频流中的每一帧图片，即可获取手部关键点的检测结果。新建脚本 hand_gesture.py 并输入如下代码。

```
import cv2
import mediapipe as mp

# 加载手部关键点模型，实例化对象
mp_hands = mp.solutions.hands
hands = mp_hands.Hands()

def find_hands(img, draw=True):
    # 将 OpenCV 获取的 BGR 转换为 MediaPipe 使用的 RGB 图像
    image_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # 进行关键点检测
    results = hands.process(image_rgb)
    mp_drawing = mp.solutions.drawing_utils
    if results.multi_hand_landmarks:
        for handlms in results.multi_hand_landmarks:
            if draw:                                    # 绘制关键点
                mp_drawing.draw_landmarks(img, handlms, mp_hands.HAND_CONNECTIONS)
    return img

# 打开摄像头或读取视频
cap = cv2.VideoCapture(0)                               # 使用摄像头
# cap = cv2.VideoCapture('test.mp4')                    # 使用当前目录下的视频文件 test.mp4
while cap.isOpened():
ret, image = cap.read()
    if not ret:
        continue                                        # 读取不成功则退出
    img = find_hands(image, draw=True)
    cv2.imshow('hand_keypoints', img)                   # 显示图像
    if cv2.waitKey(1) & 0xFF == ord('q'):               # 等待按键，按 q 键退出
        break
cap.release()                                           # 释放资源
cv2.destroyAllWindows()
```

上述代码首先初始化 MediaPipe 的 Hands 模块，打开摄像头循环读取视频帧，将其转

换为 MediaPipe 需要的 RGB 格式，再使用 hands.process()方法处理图像帧并返回手部关键点位置信息，最后通过 mp_drawing.draw_landmarks()函数绘制出手部关键点。用户按 q 键将退出循环并释放资源。如图 3-8 所示，MediaPipe 能够实现实时、精准的手部关键点检测。

图 3-8　手部关键点结果

3.6.2　手势人机交互

根据手部关键点位置信息可以进一步实现特定手势的识别。指尖判断是一种常用的手势识别方法，具体实现过程为：画一个包含关节点[0、1、2、3、5、6、9、10、13、14、19、18、17]的凸多边形，当指尖[4、8、12、16、20]在多边形内部时，代表所有手指完全弯曲，对应“拳头”手势；当只有指尖[4]位于多边形区域之外时，代表大拇指直立，可能对应“点赞”或“Diss”手势。以此类推，该方法可用于识别阿拉伯数字等其他手势。为精确区分是“点赞”还是“Diss”手势，可以结合坐标判断来比较大拇指指尖位置和手腕关键点位置的相对关系，从而确定大拇指的具体指向。此外，通过距离判断可以检测手指的弯曲情况，如果检测到大拇指与食指同时弯曲，可以将其判定为“Ok”手势。同样地，通过角度判断可以识别大拇指与食指是否存在交叉动作，从而检测“比心”手势。

基于以上分析，下面给出基于 MediaPipe 库实现“拳头”“比心”“点赞”“Diss”“Ok”五种常用手势的识别方法。新建脚本 gesture_recognizer.py，输入以下代码。

```
import mediapipe as mp
import cv2
import numpy as np
import math

def distance(x1, y1, x2, y2):                          # 计算两点之间的直线距离
    dist = math.sqrt(math.pow((x2 - x1), 2) + math.pow((y2 - y1), 2))
    return dist

def fingers_crossed(x1, x2, x3, x4):                   # 判断两个手指有无交叉动作
    distx1 = list_lms[x1][0] - list_lms[x2][0]         # 计算两个关键点横坐标之差
    distx2 = list_lms[x3][0] - list_lms[x4][0]
    disty1 = list_lms[x1][1] - list_lms[x2][1]         # 计算两个关键点纵坐标之差
    disty2 = list_lms[x3][1] - list_lms[x4][1]
    flag1 = distx1 * distx2                            # 小于 0 代表两指交叉(手势朝向上、下)
    flag2 = disty1 * disty2                            # 小于 0 代表两指交叉(手势朝向左、右)
    if flag1 < 0 or flag2 < 0:
```

```
        return True

def get_gesture(img, list_lms):                          # 识别手势
    global gesture
    hull_index = [0,1,2,3,6,10,14,19,18,17]              # 用于绘制凸多边形的手部关键点
    hull = cv2.convexHull(list_lms[hull_index,:])        # 构建凸多边形
    cv2.polylines(img,[hull], True, (0, 255, 0), 2)      # 绘制多边形

    list1 = [4,8,12,16,20]           # 检测手指伸直的关键点,即每个手指的指尖
    list2 = [3,6,10,14,18]           # 检测手指弯曲的关键点,即每个手指的第二个关节处
    up_fingers = []                  # 存储伸直手指的指尖关键点
    crooked_fingers = []             # 存储弯曲手指的关键点

    for i in list1:
        # 判断指尖关键点是否在凸多边形外部
        pt = (int(list_lms[i][0]),int(list_lms[i][1]))
        dist = cv2.pointPolygonTest(hull, pt, True)
        if dist < 0:                             # 指尖关键点位于凸多边形外部,代表手指伸直
            up_fingers.append(i)

    for j in list2:                              # 检测手指是否弯曲
        if j == 3:                               # 大拇指
            # 关键点 3 到 17 的距离
            dist1 = distance(list_lms[j][0],list_lms[j][1],list_lms[17][0],list_lms[17][1])
            # 关键点 4 到 17 的距离
            dist2 = distance(list_lms[j+1][0],list_lms[j+1][1],list_lms[17][0],list_
lms[17][1])
            if dist1 > dist2:
                crooked_fingers.append(j)
        else: # 其他手指
            # 第二个指关节处到手腕关键点 0 的距离
            dist1 = distance(list_lms[j][0],list_lms[j][1],list_lms[0][0],list_lms[0][1])
            # 对应手指指尖到手腕关键点 0 的距离
            dist2 = distance(list_lms[j+2][0],list_lms[j+2][1],list_lms[0][0],list_lms
[0][1])
            if dist1 > dist2:
                crooked_fingers.append(j)
    # 拳头,无手指伸出
    if len(up_fingers) == 0:
        gesture = "Fist"                         # 对应“拳头”手势
    # OK,大拇指和食指弯曲
    elif len(crooked_fingers) == 2 and crooked_fingers[0] == 3 and crooked_fingers[1] == 6 :
        gesture = "Ok"
    # 大拇指向上,大拇指伸直且指尖关键点 4 的纵坐标小于手腕关键点 0 的纵坐标
    elif len(up_fingers) == 1 and up_fingers[0] == 4 and list_lms[4][1]< list_lms[0][1]:
        gesture = "Thumb_up"                     # 对应“点赞”手势
    # 比心,根据 2,5,4,8 四个关键点的坐标判断大拇指和食指是否相交
    elif len(up_fingers) == 2 and up_fingers[0] == 4 and up_fingers[1] == 8 and fingers_
crossed(2,5,4,8):
        gesture = "Heart"                        # 对应“比心”手势
    # 大拇指向下,大拇指伸直且指尖关键点 4 的纵坐标大于手腕关键点 0 的纵坐标
```

```
        elif len(up_fingers) == 1 and up_fingers[0] == 4 and list_lms[4][1]> list_lms[0][1]:
            gesture = "Thumb_down"                                    # 对应“Diss”手势
        return gesture

    if __name__ == "__main__":
        cap = cv2.VideoCapture(0)
        mpHands = mp.solutions.hands
        hands = mpHands.Hands()
        mpDraw = mp.solutions.drawing_utils
        gesture = None                                                # 手势信息
        while True:
            ret, img = cap.read()                                     # 读取一帧图像
            image_height, image_width, _ = img.shape                  # 获取图像宽高
            imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)             # 转换为 RGB
            results = hands.process(imgRGB)                           # 得到检测结果
            if results.multi_hand_landmarks:
                for hand in results.multi_hand_landmarks:
                # 采集所有关键点坐标并绘制关键点
                    list_lms = []
                    for i in range(21):
                        # 归一化坐标转换为实际位置
                        pos_x = hand.landmark[i].x * image_width
                        pos_y = hand.landmark[i].y * image_height
                        cv2.circle(img, (int(pos_x), int(pos_y)), 3, (0,255,0), -1)
                        list_lms.append([int(pos_x), int(pos_y)])
                # 获取关键点列表
                list_lms = np.array(list_lms, dtype=np.int32)
                gesture = get_gesture(img, list_lms)
            else:
                gesture = None                                        # 未检测到
            cv2.putText(img, f'Gesture: {gesture}', (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 0.75,(0,
0,255),2)
            cv2.imshow("Gesture Recognition",img)
            key = cv2.waitKey(1) & 0xFF
            if key == ord('q') or gesture == 'Fist':                  # 按键或对应手势时退出程序
                break
        cap.release()                                                 # 释放摄像头
```

运行脚本，测试结果如图 3-9 所示，一旦识别到“拳头”手势，程序将立即退出循环并释放摄像头资源。尽管 MediaPipe 手势识别功能具备实时分析视频中手部动作并准确识别特定手势的能力，然而，由于采用的手势判断规则相对简单，这导致在实际应用中偶尔会出现误检现象，可以通过优化判断规则进一步提高手势识别的准确性和可靠性。

图 3-9　特定手势识别结果

本节最后，给出一个通过手势来控制灯明暗程度的应用示例。将 LED 正极与树莓派 GPIO 端口 40 引脚相连，负极则通过一个限流电阻连接至 GND 引脚。系统工作时，树莓派通过摄像头捕获手部图像，计算大拇指和食指两个指尖之间的距离，并将其转换成 PWM 信号的占空比，以此来调节 LED 的亮度。为了提供更直观的反馈，在图像上叠加显示当前 LED 的亮度条和对应的 PWM 占空比值。新建脚本 hand_control.py，输入以下内容。

```
import math
import cv2
import mediapipe as mp
import numpy as np
from gpiozero import PWMLED
from time import sleep

mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils

def find_position(img, hand_number = 0, draw = False):
    lm_list = []
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    results = hands.process(imgRGB)
    if results.multi_hand_landmarks:
        hand_landmarks = results.multi_hand_landmarks[hand_number]
        for id, lm in enumerate(hand_landmarks.landmark):
            h, w, c = img.shape
            cx, cy = int(lm.x * w), int(lm.y * h)
            lm_list.append([id, cx, cy])
            if draw:
                cv2.circle(img, (cx, cy), 7, (255, 0, 255), -1)
    return lm_list

def main():
    led = PWMLED(21)
    led.value = 0                                          # 初始状态为灭
    cap = cv2.VideoCapture(0)
    cap.set(3, 640)
    cap.set(4, 480)
    while True:
        ret, img = cap.read()
        lm_list = find_position(img)
        if len(lm_list) != 0:
            print(lm_list[4], lm_list[8])                  # 大拇指指尖和食指指尖
            x1, y1 = lm_list[4][1], lm_list[4][2]
            x2, y2 = lm_list[8][1], lm_list[8][2]
            cx, cy = (x1 + x2) // 2, (y1 + y2) // 2        # 两指尖的中点
            cv2.circle(img, (x1, y1), 10, (0, 0, 255), -1)
            cv2.circle(img, (x2, y2), 10, (0, 0, 255), -1)
            cv2.circle(img, (cx, cy), 10, (0, 0, 255), -1)
            cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 3)
            distance = math.hypot(x2 - x1, y2 - y1)        # 计算距离
            print("指尖距离：", distance)
            # 距离[30, 300]转换成占空比[10, 100]
```

```
            pwm = np.interp(distance, [30, 300], [10, 100])
            # 将占空比通过插值转换成与亮度条对应的范围
            pwm_bar = np.interp(pwm, [10, 100], [400, 150])
            led.value = pwm / 100.0
            sleep(0.01)
            cv2.rectangle(img, (50, 150), (85, 400), (255, 255, 0), 3)      # 画亮度条外框
            cv2.rectangle(img, (50, int(pwm_bar)), (85, 400), (255, 255, 0), -1) # 显示亮度条
            cv2.putText(img, f"{int(pwm)} %", (40, 450), cv2.FONT_HERSHEY_PLAIN, 3, (255,
255, 0), 2)                                                                  # 显示占空比
        else:
            led.value = 0
        cv2.imshow("Led Control", img)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

if __name__ == '__main__':
    main()
```

运行脚本，测试结果如图 3-10 所示。用户能够通过手势实时控制 LED 的亮度，还能直观地看到当前设置的亮度级别和占空比信息，提升了用户体验和互动性。

图 3-10　手势控制 LED 亮度

3.7　网络视频监控

Flask 是一个轻量级的 Web 应用框架。在最新的树莓派操作系统中，Flask 库已被预先安装。一个最基础的 Flask 项目结构通常包含一个 app.py 文件以及两个目录 templates 和 static(初始状态下，两个文件夹为空)，其中 app.py 是项目的入口文件，用于创建 Flask 应用实例，定义路由规则以及与之对应的视图函数；templates 目录存放 HTML 模板文件，而 static 目录则用于存放静态文件，如 CSS 样式表、JavaScript 脚本以及图片等。在 HTML 模板中，可以通过 Flask 提供的 URL 生成函数来引用这些静态文件，从而增强 Web 页面的交互性和视觉效果。

下面详细介绍基于树莓派搭建一个网络视频监控系统。该系统利用 OpenCV 库来捕获和处理摄像头的视频流，同时借助 Flask 框架创建一个 Web 服务器，以实现通过 HTTP 协议在浏览器页面查看监控画面的功能。首先，确保虚拟环境中安装了 OpenCV 和 Flask。如果尚未安装，可以通过 **pip install flask** 命令来安装 Flask 库。随后，新建一个名为 flask_app.py 的 Python 脚本，用于捕获视频并启动 Web 服务器，具体代码如下。

```
from flask import Flask, render_template, Response
import cv2
import threading, time

app = Flask(__name__)                                    # 使用 Flask 框架创建 Web 应用程序

# 摄像头对象和视频流线程
video_capture = None
video_stream_thread = None
camera_on = False                                        # 摄像头状态

# 摄像头处理函数
def video_stream():
    global video_capture
    while True:
        if video_capture is None:
        continue
        ret, frame = video_capture.read()                # 捕获帧
        # 将 OpenCV 的 BGR 格式转换为 Flask 需要的 RGB 格式
        _, jpeg = cv2.imencode('.jpg', frame)
        frame = jpeg.tobytes()
        # 定义生成器函数,对图像进行编码输出
        yield(b'--frame\r\n' + b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')
        time.sleep(0.1)                                  # 添加小的延迟,以减少发送的帧率

# 定义路由
@app.route('/')                                          #当根目录被访问时,执行 index()函数
def index():
    return render_template('index.html')
# 网页端实时显示视频流
@app.route('/video_feed')
def video_feed():
    return Response(video_stream(), mimetype='multipart/x-mixed-replace; boundary=frame')
# 切换摄像头的状态,同时开启或关闭视频流
@app.route('/toggle_camera', methods=['POST'])
def toggle_camera():
    global camera_on, video_capture,video_stream_thread
    if camera_on:
        camera_on = False
        if video_capture is not None:
            video_capture.release()
            video_capture = None
        if video_stream_thread is not None:
            video_stream_thread.join()                   # 等待摄像头控制线程结束
            video_stream_thread = None
        return {'status': 'off'}, 200                    # 返回字符串和一个状态码
    else:
        camera_on = True
        if video_capture is None:
            video_capture = cv2.VideoCapture(0)
        if video_stream_thread is None or not video_stream_thread.is_alive():
            # 启动摄像头控制线程
```

```
            video_stream_thread = threading.Thread(target=video_stream, daemon=True)
            video_stream_thread.start()
        return {'status': 'on'}, 200

if __name__ == '__main__':
    app.run(host='192.168.2.111', port=5000)
```

上述代码中，@app.route 装饰器用于将 URL 路径映射到视图函数上，接收到与这些路径相匹配的 HTTP 请求时，Flask 会调用相应的视图函数来处理请求，执行完毕后函数返回值会自动封装成 HTTP 响应发送给浏览器。使用 Flask 库的 render_template()函数来渲染模板文件 index.html，并将其作为 HTTP 响应发送给客户端；通过浏览器发起 POST 请求到/toggle_camera 路由时能够控制摄像头的开启与关闭；通过 video_stream()函数捕获实时视频流并进行编码输出，利用 Flask 的 Response 对象将编码后的视频流封装为 JPEG 格式的字节流返回给客户端。

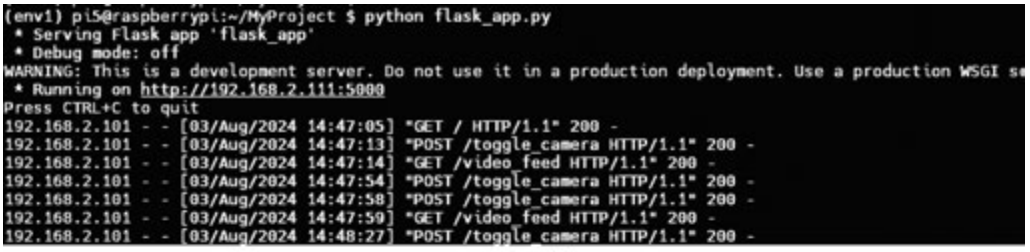
接下来，创建一个简单的 HTML 模板，通过一个按钮来控制摄像头的开启和关闭，并显示摄像头的视频流。在 flask_app.py 同一目录下，创建名为 templates 的文件夹，并在其中新建一个名为 index.html 的文件，用 VS Code 打开文件并输入如下内容。

```
<!DOCTYPE html>
  <html>
  <head>
      <title>Video Streaming with Flask</title>
  </head>
  <body>
  <div>  <img id="video-stream" src="about:blank" alt="Video Stream" style="width:
640px; height: 480px;">  </div>
  <div>   <button onclick="toggleCamera()">摄像头开/关</button>
              <p id="camera-status">摄像头：关闭</p>     </div>
  <script>
    function toggleCamera() {
    fetch('/toggle_camera', {method: 'POST'})
        .then(response => response.json())
        .then(data => {
           if (data.status === 'on') {
              document.getElementById('video-stream').src = '{{ url_for("video_feed") }}';
              document.getElementById('camera-status').textContent = '摄像头：开启';
              } else {
              document.getElementById('video-stream').src = "about:blank";
              document.getElementById('camera-status').textContent = '摄像头：关闭';
              }
          });
        }
  </script>
  </body>
  </html>
```

在这个例子中，单击“摄像头开/关”按钮时，JavaScript 会发送一个 POST 请求到/toggle_camera 路由。Flask 应用接收到请求后，会立即处理并改变摄像头的开关状态，并返回一个新的状态给前端。前端 JavaScript 会根据返回的状态信息执行相应的操作。如果

摄像头被打开，JavaScript 会更新视频源的 URL，以确保视频流能够正常显示在网页上。相反，如果摄像头被关闭，JavaScript 则会将视频源的 URL 设置为 about:blank，从而在网页上停止显示视频内容。

运行该脚本，在计算机的浏览器中输入“http://树莓派 IP 地址:5000”即可访问并查看树莓派摄像头捕获的实时视频流，结果如图 3-11 所示。其中，图 3-11(a)是在浏览器界面单击按钮时所触发的 GET 和 POST 请求消息，图 3-11(b)是实时监控画面。



(a) HTTP请求

(b) 浏览器监控画面

图 3-11　Flask 网络视频监控

3.8　图像拼接

图像拼接技术是一种将多张具有重叠区域的图像融合成一张宽广视角图像的方法，它在全景摄影、医学影像分析、虚拟现实以及视频监控等领域具有广阔的应用前景。使用 OpenCV 库可以实现多幅图像的无缝拼接，这一过程通常涉及关键点提取、特征匹配与图像

对齐、重叠区域融合处理等步骤，其关键在于通过特征匹配和单应性矩阵来确定图像之间的几何变换关系。

拼接多张图像的传统方法往往遵循从左至右或从右至左的顺序，逐步将图像两两配对并融合。具体来说，首先合并一对相邻图像以形成初步的全景图；随后，初步全景图再与下一张图像进行拼接，不断重复该过程直至得到最终的全景图像。然而，当拼接图像较多时，单纯地向一侧（左或右）连续投影往往会导致边缘区域的图像产生显著的扭曲和失真。

为克服上述问题，这里采用一种新的策略：以中间图像为分界把所有待拼接图像均匀分割为两个子集，并分别向中间方向进行投影拼接。具体而言，以中间图像作为参照基准，对于右侧图像集合，采用向左投影映射的方式进行拼接；而对于左侧图像集合，则采用向右投影映射。完成左右两侧的独立拼接后，再将左右两个全景图进行最终的合并，从而得到完整且连贯的全景图像。此外，每次图像拼接过程中，将源图像投影映射到目标图像平面后，需要去除图像背景（消除潜在的缝隙），实现无缝连接的全景图像效果。

下面给出多张图像拼接的具体实现代码，新建脚本 image_stitching.py，输入以下内容。

```
import os, glob, imageio
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage

# 采用指定方法计算关键点和特征描述符
def extract_feature(image, extractor):
    if extractor == 'sift':
        descriptor = cv2.xfeatures2d.SIFT_create()
    elif extractor == 'surf':  # 因版权问题，新版 OpenCV 可能不支持 SURF 算法
        descriptor = cv2.xfeatures2d.SURF_create()
    elif extractor == 'brisk':
        descriptor = cv2.BRISK_create()
    elif extractor == 'orb':
        descriptor = cv2.ORB_create()
    keypoints, features = descriptor.detectAndCompute(image, None)
    return keypoints, features

# 创建并返回 BFMatcher 对象
def create_matcher(method, mode):
    if method == 'sift' or method == 'surf':
        matcher = cv2.BFMatcher(cv2.NORM_L2, crossCheck = mode)
    elif method == 'orb' or method == 'brisk':
        matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = mode)
    return matcher

# 返回最佳匹配关键点
def match_BF(ft1, ft2, method):
    matcher = create_matcher(method, mode = True) # 交叉过滤
    best_matches = matcher.match(ft1,ft2)
    # 按距离从小到大排序
    rawMatches = sorted(best_matches, key = lambda x:x.distance)
    print("BF 匹配特征点:", len(rawMatches))
    return rawMatches
```

```
# 返回 k 个最佳匹配关键点(降序排列取前 k 个)
def match_KNN(ft1, ft2, ratio, method):
    matcher = create_matcher(method, mode = False)
    rawMatches = matcher.knnMatch(ft1, ft2, 2) # k = 2
    print("KNN 匹配特征点:", len(rawMatches))
    matches = []
    for m,n in rawMatches:                    # 当两个匹配之间的距离较小,才确认该匹配
        if m.distance < n.distance * ratio:   # 比例测试
            matches.append(m)
    return matches

# 找到单应性矩阵描述两幅图像之间的相对位置关系
def get_homography(kps1, kps2, matches, ReprojTh):
    kps1 = np.float32([kp.pt for kp in kps1]) # 将匹配点坐标转换为 numpy 数组
    kps2 = np.float32([kp.pt for kp in kps2])
    if len(matches) > 4:                      # 计算单应性矩阵至少需要 4 对匹配点
        pts1 = np.float32([kps1[m.queryIdx] for m in matches])    # 构造两组点集
        pts2 = np.float32([kps2[m.trainIdx] for m in matches])
        # 基于 RANSAC 算法(容许投影错误阈值为 ReprojTh)计算 3×3 的单应性矩阵 H
        H, _ = cv2.findHomography(pts1, pts2, cv2.RANSAC, ReprojTh)
        return H
    else:
        return None

def distance_transform(img_rgb):              # 计算图像中非零点到最近背景点(0)的距离
    _,thresh = cv2.threshold(img_rgb, 0, 255,cv2.THRESH_BINARY)
    thresh = thresh.any(axis = 2)
    thresh = np.pad(thresh,1,mode = 'maximum')
    dist = ndimage.distance_transform_edt(thresh)[1:-1,1:-1]
    dist = dist[:,:,None]
    return dist /dist.max() * 255.0

def mask_result(result):                      # 去除拼接图像周边的黑色背景区域
    mask = distance_transform(result)
    locs = np.where(mask > 0)
    xmin = np.min(locs[1])
    xmax = np.max(locs[1])
    ymin = np.min(locs[0])
    ymax = np.max(locs[0])
    result_mask = result[ymin:ymax,xmin:xmax]
    return result_mask

def img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir, show):
    global directory,iter
    kps1, ft1 = extract_feature(gray1, extractor = feature_extractor)
    kps2, ft2 = extract_feature(gray2, extractor = feature_extractor)
    if feature_matching == 'bf':              # 创建并返回 BFMatcher 对象
        matches = match_BF(ft1, ft2, method = feature_extractor)
        # 仅显示 50 个匹配点
        Img3 = cv2.drawMatches(Img1,kps1,Img2,kps2,matches[:50],
               None,flags = cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    elif feature_matching == 'knn':
        matches = match_KNN(ft1, ft2, ratio = 0.75, method = feature_extractor)
```

```
            Img3 = cv2.drawMatches(Img1,kps1,Img2,kps2, np.random.choice(matches,50),
                  None,flags = cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        if show:
            fig = plt.figure(figsize = (10,8))
            plt.imshow(Img3)
            plt.axis('off')
            plt.savefig(f'{directory}{feature_matching}_img.png', bbox_inches = 'tight', dpi =
300)
            plt.show()
        H = get_homography(kps1, kps2, matches, ReprojTh = 4)
        if H is not None:
            print("H = ",H)
        width = Img2.shape[1] + Img1.shape[1] # 设置水平全景图宽高
        height = max(Img2.shape[0], Img1.shape[0])
        if (dir == "right"):
            # 进行透视变换,目标图像 = H * 原图像
            result = cv2.warpPerspective(Img1, H,(width, height))
            result[0:Img2.shape[0], 0:Img2.shape[1]] = Img2
        elif(dir == "left"):
            dw = Img1.shape[1]//3 # 左边填充的宽度,可根据实际情况调整
            result = np.zeros((height, Img1.shape[1] + dw, 3), dtype = np.uint8)
            result[:Img1.shape[0], dw:] = Img1 # 先左边填充,以免透视变换后图像不完整
            result = cv2.warpPerspective(result, H,(width, height))
            result[:Img2.shape[0], dw:Img2.shape[1] + dw] = Img2
        result = mask_result(result)
        if iter == 1:
            if (dir == "right"):
                imageio.imwrite(f'{directory}right_panorama_img.png', result)
            elif(dir == "left"):
                imageio.imwrite(f'{directory}left_panorama_img.png', result)
        elif iter == 0:
            print("图像拼接完成")
            plt.imshow(result)
            plt.show()
            imageio.imwrite(f'{directory}panorama_img.png', result)
        else:
            imageio.imwrite(f'{directory}{iter}_img.png', result)

    def img_convert(Img):                                          # 图像格式转换
        # 将 OpenCV 读取的 BGR 转换为 Matplotlib 兼容的 RGB
        Img = cv2.cvtColor(Img,cv2.COLOR_BGR2RGB)
        gray = cv2.cvtColor(Img, cv2.COLOR_RGB2GRAY)               # 转换为灰度图像
        return Img,gray

    if __name__ == "__main__":
        feature_extractor = 'sift'
        feature_matching = 'knn'
        directory = './images/' # 设置图像目录路径
        # 按顺序读取目录下所有指定类型的图像文件,其中的图像拍摄顺序从左到右
        image_files = sorted(glob.glob(os.path.join(directory, 'img?.png'))) # 图像名为“img 数字”
        inter = int(len(image_files)/2)                            # 中间帧序号
        right_image_files = image_files[inter:len(image_files)]# 右边一半数量的图像
        iter = len(right_image_files) - 1                          # 右边全景图像需要拼接的次数
```

```
    # 采用最前面的两幅图像进行第一次拼接
    Img1 = cv2.imread(right_image_files[1])              # Img1 对应拼接在右边的图
    Img2 = cv2.imread(right_image_files[0])              # Img2 对应拼接在左边的图
    Img1, gray1 = img_convert(Img1)
    Img2, gray2 = img_convert(Img2)
    img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir = "right",show = False)
    iter = iter - 1
    for Id in range(2, len(right_image_files)):          # 循环读取下一幅相邻的图像进行拼接
        Img1 = cv2.imread(right_image_files[Id])         # 拼接在右边的图
        Img2 = cv2.imread(os.path.join(directory, f'{iter + 1}_img.png')) # 拼接在左边的图
        Img1, gray1 = img_convert(Img1)
        Img2, gray2 = img_convert(Img2)
        img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir = "right", show =
False)
        iter = iter - 1
    left_image_files = image_files[0:inter + 1]          # 左边的一半图像
    iter = len(left_image_files) - 1                     # 左边图像拼接次数
    # 采用最前面的两幅图像进行第一次拼接
    Img1 = cv2.imread(left_image_files[inter - 1])       # 拼接在左边的图
    Img2 = cv2.imread(left_image_files[inter])           # 拼接在右边的图
    Img1, gray1 = img_convert(Img1)
    Img2, gray2 = img_convert(Img2)
    img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir = "left", show = False)
    iter = iter - 1
    for Id in range(inter - 2, - 1, - 1):                # 循环读取下一幅相邻的图像进行拼接
        Img1 = cv2.imread(left_image_files[Id])          # 拼接在左边的图
        Img2 = cv2.imread(os.path.join(directory, f'{iter + 1}_img.png')) # 拼接在右边的图
        Img1, gray1 = img_convert(Img1)
        Img2, gray2 = img_convert(Img2)
        img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir = "left", show =
False)
        iter = iter - 1
    # 最后将得到的左右拼接图再次拼接为全景图
    Img1 = cv2.imread(os.path.join(directory, 'right_panorama_img.png'))
    Img2 = cv2.imread(os.path.join(directory, 'left_panorama_img.png'))
    Img1, gray1 = img_convert(Img1)
    Img2, gray2 = img_convert(Img2)
    img_stitching(Img1, gray1, Img2, gray2, feature_matching, dir = "right", show = False)
```

上例中，首先加载两幅图像并进行格式转换，选择 SIFT 特征提取器，使用 detectAndCompute()函数计算两者的关键点和特征描述符。采用暴力匹配或 KNN 匹配法挑选两幅图像中最相似的关键点，以线段相连的形式显示其中的 50 个匹配特征点对。基于匹配特征点集计算用于将两幅图像拼接在一起的单应性矩阵。为了有效减少错误匹配点的影响，采用对异常值具有鲁棒性的 RANSAC 算法来估计单应性矩阵。随后，利用 cv2.warpPerspective()函数对一幅图像进行透视变换，将其映射到另一幅图像的平面上，从而实现两幅图像的拼接。此外，自定义 mask_result()函数的作用是通过背景剔除去除图像投影变换中出现的缝隙，进一步优化拼接效果。

图 3-12 展示了在摄像头位置固定的情况下，通过旋转拍摄获取的 5 张分辨率为 352×288 像素且具有重叠区域的图像拼接过程及其结果。具体来说，图 3-12(a)是原始图像；

图 3-12(b)和图 3-12(c)对应以中间图像为基准，左右两边图像各自拼接的结果。由于摄像头拍摄角度的变化，边缘图像在拼接时未能实现完全无缝的对接，出现了少量背景区域；图 3-12(d)是左右两个拼接图像经过最后拼接得到的全景图。尽管在拼接过程中边缘部分存在些许瑕疵，但整体上，这幅全景图仍然较为准确地反映了摄像头旋转拍摄所捕获的场景全貌。

(a) 原始图像

(b) 左边拼接图

(c) 右边拼接图

(d) 全景图

图 3-12　图像拼接结果

3.9　文字识别

Tesseract 是目前最常用的用于机器打印字符识别的开源 OCR(光学字符识别)工具。它能从图像中精准地识别出文本内容，并将其转换为可编辑的文本格式，这一功能使其在纸质文档数字化、自动化办公、安全监控等领域得到广泛应用。在 Python 环境下使用 Tesseract 实现文字识别，首先要安装 Tesseract 的 Python 支持包 Pytesseract，命令为 **pip install pytesseract**。此外，还需通过 **sudo apt-get install tesseract-ocr** 安装 Tesseract OCR 引擎。值得注意的是，如果要识别图片中的中文，还需额外下载中文包(https://github.com/tesseract-ocr/tessdata_best)，并将下载好的文件 chi_sim.traineddata 保存到/usr/share/tesseract-ocr/5/tessdata 目录下。为确保 Tesseract 能正确访问该语言包，可能还需调整 tessdata 目录的权限，使用命令 **sudo chmod 777 -R /usr/share/tesseract-ocr/5/tessdata** 改变文件夹的可读、可写、可执行属性。安装及配置完成后，运行 tesseract --list-langs 命令可以查看 Tesseract 已支持的语言列表，其中应包含中文简体(chi_sim)。

下面首先给出一个使用 pytesseract 库来识别图像中文字的简单示例。新建 Python 脚

本 OCR.py,输入以下代码。

```
import cv2
import pytesseract
from PIL import Image

def ocr_with_opencv(image_path):
    # 使用 OpenCV 读取图像
    image = cv2.imread(image_path)
    # 转换为灰度图像
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # 阈值化操作,将图像转换为二值图像
    _, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    # 使用 Pillow 打开二值化后的图像
    pil_image = Image.fromarray(thresh)
    # 使用 Tesseract 引擎进行 OCR
    text = pytesseract.image_to_string(pil_image, lang='eng')          # 识别英文
    #text = pytesseract.image_to_string(pil_image, lang='chi_sim')    # 识别中文
    return text
#调用函数并打印识别结果
print(ocr_with_opencv('en.png'))
#print(ocr_with_opencv('ch.png'))
```

执行该脚本后,将自动识别给定图片中的文字内容,测试结果如图 3-13 所示。上例中,ocr_with_opencv()函数首先使用 OpenCV 读取图像文件,并将其转换为灰度图像;然后通过阈值化处理将灰度图像进一步转换为二值图像,以简化后续的文字识别过程;随后,将二值图像转换为 Pillow 库支持的图像对象格式;最后,调用 pytesseract.image_to_string()函数来识别图像中的文字,其中,lang='eng'参数指定了识别语言为英文。若需识别简体中文文本,只需将语言代码改为 lang='chi_sim'即可。

图 3-13　中英文图片与识别结果

接下来,介绍基于 OpenCV 和 Tesseract 实现树莓派文件扫描的功能,通过鼠标选定图像中需要识别的文字区域,然后通过 pytesseract 库实现文字识别。值得注意的是,由于拍摄角度可能导致图像出现倾斜,在进行文字识别前,需要先对图像进行方向校正,步骤为:首先,对图像执行 Canny 边缘检测算法,识别出图像中的边缘信息;然后,使用 OpenCV 提供的 HoughLinesP()函数在边缘图像上检测直线;最后,计算所有检测到的直线与水平方向的夹角,将这些角度的中值作为图像的倾斜角度,并以此为基准来旋转图像。新建脚本 roi_ocr.py,输入以下代码。

```
import cv2
import numpy as np
import math
from scipy import ndimage
import pytesseract

coordinates = []                                          # 用于存储 ROI 区域的坐标

def preProcessing(img, edge_min, edge_max):
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Canny 边缘检测,两个阈值通过滑块获取
    img_Canny = cv2.Canny(img_gray, edge_min, edge_max, apertureSize = 3)
    kernel = np.ones((5,5))   # 形态学预处理(非必须),可以根据需要去除或调整
    imgDial = cv2.dilate(img_Canny,kernel,iterations = 2)
    img_edges = cv2.erode(imgDial,kernel,iterations = 1)
    # 霍夫变换检测直线
    lines = cv2.HoughLinesP(img_edges, 1, math.pi / 180.0, 100, minLineLength = 100,
maxLineGap = 5)
    cv2.imshow("image_lines", img_edges)
    return lines

def orientation_correction(img, lines, save_image = False):# 图片方向校正
    angles = []
    for line in lines:                                    # 求直线的角度
        x1, y1, x2, y2 = line[0]
        angle = math.degrees(math.atan2(y2 - y1, x2 - x1))
        angles.append(angle)
    median_angle = np.median(angles)                      # 得到角度的中位值
    img_rotated = ndimage.rotate(img, median_angle)       # 按中值角度旋转图像
    if save_image:
        cv2.imwrite('image_corrected.jpg', img_rotated)
    return img_rotated

def roi_selection(event, x, y, flags, param):             # ROI 区域选择
    global coordinates
    if event == cv2.EVENT_LBUTTONDOWN:                    # 单击鼠标左键选择左上角
        coordinates = [(x, y)]
    elif event == cv2.EVENT_LBUTTONUP:                    # 拖动并释放鼠标左键确定右下角
        coordinates.append((x, y))

def nothing(x):
    pass

if __name__ == "__main__":
    # 创建参数调整滑块
    cv2.namedWindow("TrackBars")
    cv2.resizeWindow("TrackBars",640,60)
    cv2.createTrackbar("Edge Min","TrackBars",50,255,nothing)
    cv2.createTrackbar("Edge Max","TrackBars",150,255,nothing)
    input_img = cv2.imread("test_ocr.jpg")                # 读取待识别的图片
    while True:
        # 获取边缘检测参数
        edge_min = cv2.getTrackbarPos("Edge Min","TrackBars")
```

```
        edge_max = cv2.getTrackbarPos("Edge Max", "TrackBars")
        lines = preProcessing(input_img, edge_min, edge_max)
        key = cv2.waitKey(1) & 0xFF
        if key == ord('q'):
            cv2.destroyAllWindows()
            break
        elif key == 13:                                    # 按 Enter 键,往下执行
            img_rotated = orientation_correction(input_img, lines)
            cv2.imshow("image", img_rotated)
            cv2.namedWindow("image")
            cv2.setMouseCallback("image", roi_selection)  # 设置鼠标回调函数
            image_copy = img_rotated.copy()
            while True:
                if cv2.waitKey(1) & 0xFF == ord('c'):      # 等待选择 ROI,随后按 C 键继续
                    break
        if len(coordinates) == 2:
            cv2.rectangle(image_copy, coordinates[0], coordinates[1], (0,0,255), 2)
            cv2.imshow("image", image_copy)
            image_roi = image_copy[coordinates[0][1]:coordinates[1][1], coordinates[0]
[0]:coordinates[1][0]]
            cv2.imshow("ROI", image_roi)
            cv2.waitKey(0)                                 # 按任意键执行 OCR
            text = pytesseract.image_to_string(image_roi, lang = 'chi_sim')
            print(text)
            coordinates = []                    # 重新初始化,以便选择新的 ROI 再次执行 OCR
            image_copy = img_rotated.copy()
```

运行脚本后,可以通过滑动条调整边缘检测阈值,以获得最佳的直线检测结果。当阈值调整满意后,按 Enter 键,从显示的已校正图像中选择待识别的文字区域。随后,按 C 键,将截取并显示选择的区域。最后,再按任意键,便会启动文字识别流程。测试结果如图 3-14 所示,图中依次为原始图像、直线检测的结果、方向校正后图像中选择的识别区域以及对应的识别结果。需要说明的是,为了展示中间过程,上述代码中加入了多次按键操作,如果对同一张图片进行多次识别,除第一次需要按上述流程操作外,后续识别只需在图中选定新的区域后按任意键就可以得到识别结果。经过对多张图像的测试发现,成像条件的不同会对文字识别的准确性有一定的影响,可能会出现少量文字识别错误的情况,通过调整边缘检测和形态学处理的参数可以提高文字识别的准确度。

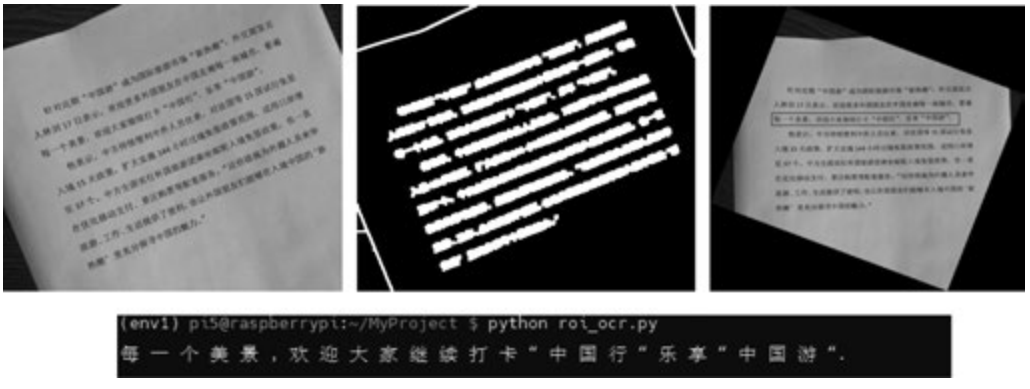


图 3-14　图片文字识别

3.10 DNN 模块目标检测

当前，深度神经网络(DNN)在计算机视觉领域的发展可谓空前绝后，在实际应用中展现出了巨大的潜力和价值。利用 OpenCV DNN 模块可以对图像和视频完成基于深度学习的计算机视觉推理。OpenCV DNN 支持很多常见的深度学习框架，如 Caffe、TensorFlow、Darknet、PyTorch，能够实现包括图像分类、目标检测、图像分割、文字检测和识别以及姿态估计等在内的大多数计算机视觉任务。OpenCV 支持的深度学习框架和相关模型可参见 https://github.com/opencv/opencv/ wiki/Deep-Learning-in-OpenCV。

本节将介绍采用 OpenCV DNN 模块加载 YOLOv4-Tiny 预训练模型，以实现图像中的目标检测任务。YOLOv4-Tiny 是 YOLOv4 的轻量化版本，它基于 Darknet 框架设计，旨在减少模型的计算复杂度和内存消耗，同时保持较高的检测精度。首先，从官方 GitHub 仓库(https://github.com/AlexeyAB/darknet)下载模型权重文件 yolov4-tiny.weights，网络配置文件 yolov4-tiny.cfg 和目标类别文件 coco.names。随后，使用 OpenCV DNN 模块加载模型的权重文件(.weights)与配置文件(.cfg)就可以实现目标检测功能。这意味着使用者无须安装或配置深度学习框架，也不需要深入了解底层框架的细节。

使用 DNN 模块进行目标检测的步骤包括加载网络模型、构建输入、执行推理、解析输出。新建脚本 opencv_dnn.py，输入以下代码。

```
import time
import numpy as np
import argparse
import cv2

with open('coco.names', 'r') as f:                                    # 加载类别名称
   CLASSES = [line.strip() for line in f.readlines()]
#每类目标随机选择一种颜色标记
COLORS = [np.random.randint(0, 255, size=3).tolist() for _ in range(len(CLASSES))]

if __name__ == "__main__":
    # 创建解析对象，添加模型、配置文件、测试图片与阈值参数
    ap = argparse.ArgumentParser()
    ap.add_argument("--model", default='yolov4-tiny.weights', help="pre-trained
model")
    ap.add_argument("--cfg", default='yolov4-tiny.cfg', help="config file")
    ap.add_argument("--img", default='street.jpeg', help="path to image")
    ap.add_argument("--conf", type=float, default=0.25,help="filter weak outs")
    ap.add_argument("--thre", type=float, default=0.4,help="nms threshold")
    args = ap.parse_args()

    # 加载网络模型
    print("[INFO] loading model...")
    net = cv2.dnn.readNet(args.model, args.cfg)
    # 获取输出层名称
    layer_names = net.getLayerNames()
    output_layers = [layer_names[i-1] for i in net.getUnconnectedOutLayers()]
```

```
    # 加载图像
    image = cv2.imread(args.img)
    H, W = image.shape[:2]
    start_time = time.time()                                    # 记录开始时间
    # 设置 blob 并传递给网络,执行前向传播获取预测结果
    blob = cv2.dnn.blobFromImage(image, 1.0/255, (416, 416), (0, 0, 0), swapRB = True)
    net.setInput(blob)
    outs = net.forward(output_layers)

    # 初始化检测到的对象列表
    class_ids = []
    confidences = []
    boxes = []
    for out in outs:                                            # 解析网络输出
        for detection in out:
            # 将目标指定为置信度最高的类别
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > args.conf:                          # 保留高于置信度阈值的检测结果
                # 目标检测框的坐标,左上角坐标和宽高
                center_x,center_y,w,h = detection[0:4] * np.array([W, H, W, H])
                x = center_x - w/2
                y = center_y - h/2
                box = np.array([x, y, w, h]).astype(np.int32)
                # 更新列表
                boxes.append(box)
                confidences.append(confidence)
                class_ids.append(class_id)

    # 非极大值抑制消除重叠框,参数: 边界框列表、最大置信度列表、置信度和 NMS 阈值
    indexes = cv2.dnn.NMSBoxes(boxes, confidences, args.conf, args.thre)
    for i in indexes:
        x, y, w, h = boxes[i]
        # 绘制检测到的目标
        label = "{}: {:.2f} % ".format(CLASSES[class_ids[i]], confidences[i] * 100)
        cv2.rectangle(image, (x, y), (x + w, y + h), COLORS[class_ids[i]], 2)
        cv2.putText(image, label, (x, y - 6), cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[class_
ids[i]], 2)
    end_time = time.time()                                      # 记录结束时间
    inference_time = "inference_time: {:.2f}s".format(end_time - start_time)
    cv2.putText(image, inference_time, (15, 15), cv2.FONT_HERSHEY_SIMPLEX, 0.75,(0,0,255), 2)
    cv2.imshow("Result", image)                                 # 显示结果图像

    k = cv2.waitKey(0)
    if k == 27:                                                 # 按 Esc 键 退出
        cv2.destroyAllWindows()
    elif k == ord('s'):                                         # 按 s 键保存图像并退出
        cv2.imwrite('result.jpg',image)
        cv2.destroyAllWindows()
```

上例中,首先使用 cv2.dnn.readNet()函数加载 DNN 模型,通过 cv2.dnn.blobFromImage()函数将图片转换成可以输入 DNN 的格式,使用 setInput()函数将数据送入网络模型中,调

用 forward()方法执行前向传播并返回预测结果，其中包括图片中所有目标的位置信息（目标框中心坐标和宽高的归一化值）、置信度以及目标对应每个类别的置信度分数。随后，排除低于置信度阈值的检测结果，对剩余可能的结果再执行非极大值抑制消除重叠，仅保留置信度最高的目标框。最后，绘制检测到的目标信息并显示推理时间。

运行脚本，将得到默认输入图像 street.jpeg 的目标检测结果，如图 3-15(a)所示。如果要对其他图像进行目标检测，只需通过--img 参数指定输入图像即可，命令为 **python opencv_dnn.py --img bus.jpg**，检测结果如图 3-15(b)所示。测试结果表明，输入图像大小为 416×416 像素时，YOLOv4-Tiny 的单帧推理时间约为 0.4s。需要说明的是，OpenCV DNN 模块并不能训练 DNN 模型，只支持加载训练好的模型对图片和视频进行推理。尽管如此，OpenCV DNN 模块仍可以作为初学者涉足机器视觉与深度学习领域的起始点。

(a) street.jpeg　　(b) bus.jpg

图 3-15　DNN 模块目标检测结果