基于树莓派的智能车运行控制系统

南京大学软件学院 冯国豪

关键词： 嵌入式；控制系统；自动寻迹；web

摘要：本文介绍了基于树莓派开发的智能车控制系统。该系统基于智能车硬件实现了其上的调用库，并基于调用库开发了上层应用，可以实现自动寻迹、简单避障、摄像头控制、小车遥控等等功能，系统还实现了 web 用户界面，可以使用网页对小车进行控制。

一、概述

树莓派是一种为计算机编程教育设计的卡片式计算机，具有体积小，非常容易集成在其他硬件设备之中；提供大量的硬件设施和接口；提供简单易用的 Linux 操作系统；能够使用高级编程语言在其上进行编程工作等优点，非常适合用来开发嵌入式系统，控制硬件设备行为，进而为用户提供高可用的服务。

本文便介绍了基于树莓派开发的智能车运行控制系统，该系统基于微雪电子公司生产的 AlphaBot2-Pi 智能车设备，使用 Python 语言作为开发语言，基于车上提供的红外传感器、马达、蜂鸣器、超声波距离传感器等硬件设备能力开发了对应的调用库，并基于硬件基础设施调用库开发了上层应用，可以实现自动寻迹、LED 灯光的智能控制、遥控器遥控控制车辆、车辆简单避障、摄像头的远程控制等功能，并进行了一定的封装集成，最后还基于 Vue.js 与 Python 的 web 开发框架 Flask 实现了简单易用的 web 用户界面，能够基于 web 服务使客户端同车辆进行通信，进行小车功能的控制。同时，系统还提供了车辆的编程控制，用户可以通过简单的选择与点击创建小车运行控制脚本，并控制小车按照此脚本设置的任务顺序进行运行。

本文将介绍此系统的设计思路与设计架构，逐一介绍每部分实验的实验原理、所需设备与实验步骤，并在最后分析系统设计中存在的问题与可行的改进方案。您可以访问 github 仓库（地址：https://github.com/fengguohao/intelligent_vehicle_system）下载该系统的完整源代码，并在相同的软硬件环境下复现所有步骤。您同样可以通过此仓库下载观看系统的视频介绍。

二、系统架构

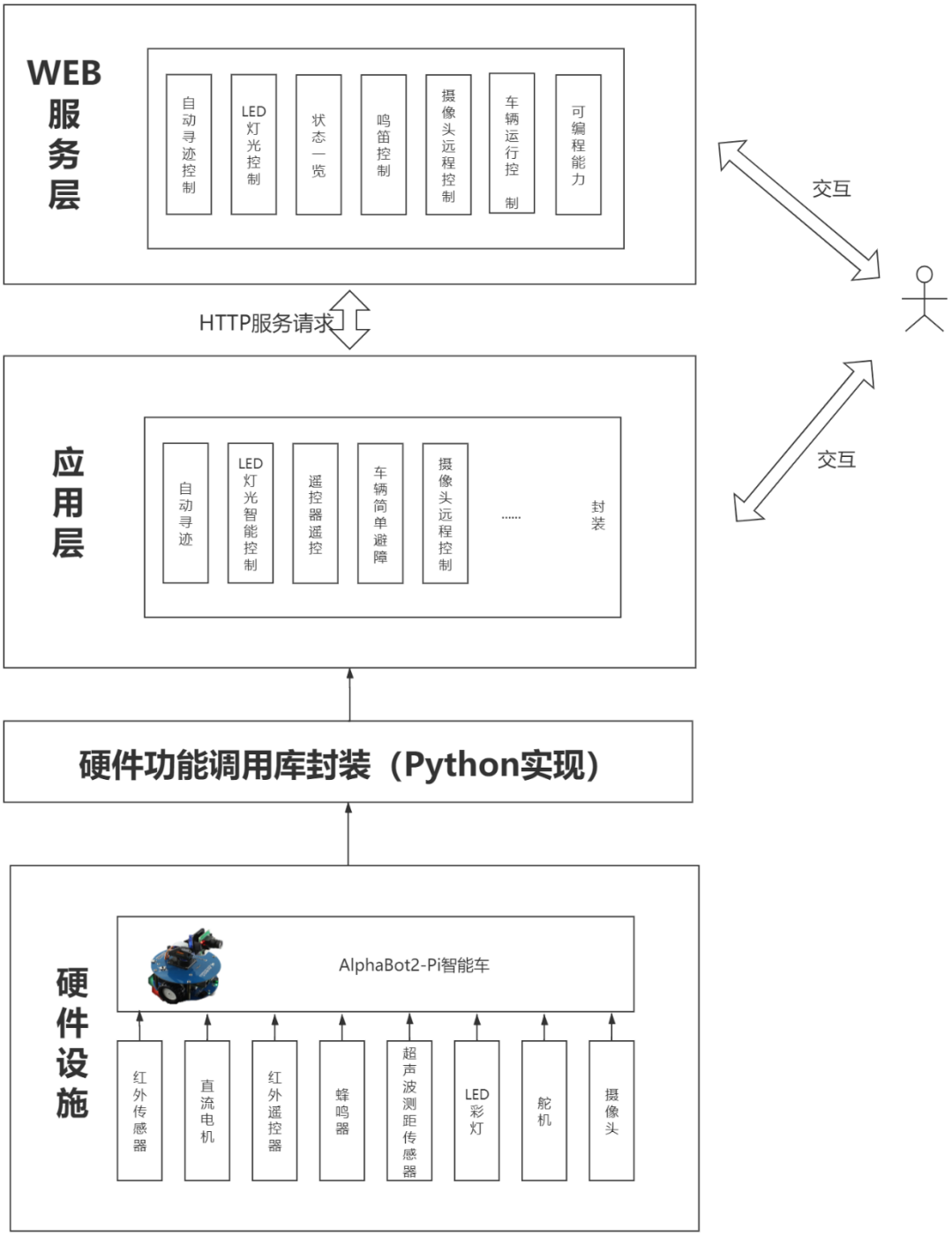


Fig .1 智能车控制系统架构图

如 Fig.1 智能车控制系统架构图所示，系统由硬件设施层、硬件功能调用库封装层（Python 实现）、应用层与 web 服务层实现四部分组成。

硬件层部分基于微雪电子 AlphaBot2-Pi 智能车设备，该设备搭载了红外传感器、直流电机、红外遥控器、蜂鸣器、超声波距离传感器、LED 彩灯、舵机、摄像头等硬件，这些硬件都能通过 GPIO 接口连接到树莓派开发板，进而在上层进行设备资源的调用。

为了便于硬件功能的调用，系统针对设备常用功能开发了相应的调用库，当需要使用设备的某个功能时直接引入所需的硬件功能调用库（carFacility），便可以使用对应的功能，大大简化了应用实现过程，提高了开发效率。该模块实现了与其他模块的解耦，具有较高的可用性。

基于硬件功能调用库封装，该系统又在上层开发了应用层，实现了自动寻迹、LED 灯光的智能控制、遥控器遥控控制车辆、车辆简单避障、摄像头的远程控制等一系列的功能，并做了简单的封装，以供用户使用遥控器的控制能力体验相关功能，您还可以通过简单地修改封装实现，快速将遥控器功能替换为您所需要的功能。

为了进一步方便用户使用，系统又对应用层进行了进一步的封装，将其实现为 WEB 用户界面，更加方便用户与系统进行交互。

系统在设计过程中，坚持高内聚、低耦合，将软件系统划分为不同的功能模块，提高各模块的独立性，从而力求最大程度上提升代码的可重用性与可修改性。在功能设计上，该控制系统一方面希望将小车的设备能力充分利用起来，便于用户使用；另一方面又希望尽可能挖掘功能背后的使用场景，提高功能的实用性。

三、硬件功能调用库封装

（一）相关原理介绍：

1. PWM：脉冲宽度调制

PWM 是 Pulse Width Modulation 的缩写，中文名称为脉冲宽度调制，是一种利用数字输出对模拟电路进行控制的有效技术。其关键技术是通过改变脉冲的宽度来实现不同的效果。其中有两个关键的参数为频率（单位：Hz）与占空比(单位：百分比)，其中占空比的计算公式如下所示

$$占空比 = \frac{每个周期内高电平的时间}{周期时间}$$

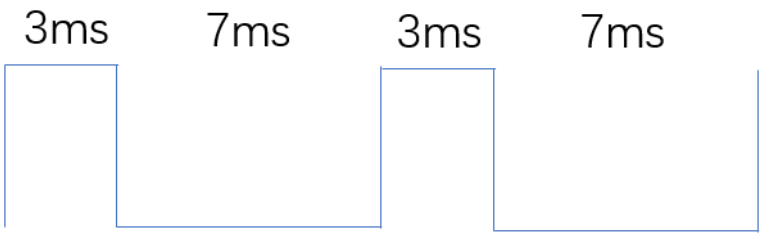


Fig.2 PWM 波形

例如对于 Fig.2 中所示的一组波形，我们可以观察到其高低电平的脉冲宽度是不同的，同时可以观察到，每个周期时间为 10ms，其中高电平时长为 3ms，进而我们可以求得其频率为 100Hz，其占空比为 30%。

利用这项技术，我们就可以通过数字信号控制蜂鸣器的频率和音量了。具体来说，PWM 波的频率与实际发出的声音的频率正相关，占空比和其音量正相关。同样的道理，我们也可以通过改变 PWM 波的占空比，控制直流电机的转动速度，控制舵机的转动速度，原理大致相仿，不再赘述。

2. 红外传感器

车辆上共搭载了三种红外传感器，分别是用于检测前方是否存在障碍物的红外避障传感器（只能返回 0/1 信号）、用于寻迹的红外传感器（能够量化红外强度值）和用于接收遥控器信号的红外线接收器。下面分别介绍这三种红外传感器的原理。

a. 红外传感器

Fig.3 红外传感器

如 Fig.3 所示，红外传感器由一个红外发射管与一个红外接收管组成，发射管与接收管之间光隔离，发射管发出的红外光被反射后由接收管接收，当发射管

离反射物体越近，反射物体反射能力越强时，接收管接收到的反射光也就越强，当达到光强度阈值时，便可以确定前方有障碍物。

而能够量化红外强度值的传感器相对上面一种区别在于加入了模数转换器，能够将红外接收管接收到的红外强度值模拟信号编码为数字信号。

具体到我们实验中所使用的传感器，其使用的数模转换器型号为 TLC1543，是一个 10 通道、10bits 的 SPI 接口模数转换器。可以在 IOCLK 产生时钟脉冲，ADDR 逐位输入通道编码，从 DOUT 中读出转换值。

b.红外接收传感器

红外接收传感器与红外遥控器配合使用。对于我们使用的硬件，红外遥控器发射的信号使用 38kHz 左右的载波对基带进行调制，接收端对信号监测、放大、滤波、解调，然后输出基带信号。收发方约定使用相同协议进行通信，这里我们使用的是 NEC 红外通信协议，具体来说，发送端会首先发送一个 9ms 低电平接 4.5ms 高电平的引导码，接收方检测到引导码后开始识别后面的数据，0.56ms 低+0.56ms 高表示“0”，0.56ms 低+1.69ms 高表示“1”，一组 01 序列便构成了一个按键特征字，接收方便可以根据收到的特征字产生一系列动作。

3.超声波传感器

超声波传感器通过发射超声波，检测发射时间与接收时间之间的时间差，再根据时间差进行一定的计算，便可以计算出障碍物与传感器之间的距离。计算公式为:

$$distance = \frac{V\mathrm{t}}{2}$$

其中 V 为声速，可以取 340m/s，t 即为发射时间与接收时间之间的时间差。

超声波传感器上有四个接口，分别是 VCC，Trig，Echo，GND，VCC 和 GND 之间形成电位差用于供电，在使用超声波传感器时要求在 Trig 端口产生不短于 10μs 的正脉冲,模块便会自动发出 8 个周期的超声波脉冲信号(40kHz),并在 Echo 端输出高电平。当检测到回波时将 Echo 置为低电平。

4.直流电机

直流电机控制电路由两个功率输出端和一个控制开关组成，当控制开关接通时，输出电流方向由两个输入端的电压差决定，这样便可以实现电机的双向转动调整。

（二）环境准备：

以下程序均在树莓派版 Linux 操作系统下进行，默认使用 Python3 编程，因此需要系统搭载 Python3 环境，该环境会随树莓派官方的 Linux 操作系统自带，如果环境正常则无需自行安装。若没有 Python3 环境，可尝试通过 sudo apt-get install python3 来进行安装。

为了减小开发难度，提高效率，我们还使用了树莓派 GPIO 模块进行开发，该模块集成了 GPIO 接口开发所需的各种函数库，在系统中安装此模块的方法如下。

1.进入终端（terminal）

2.安装 git

```
sudo apt-get update
sudo apt-get install git
```

3.下载 GPIO 模块源码

```
git clone https://github.com/yfang1644/RPi.GPIO
```

4.进入目录进行编译安装

```
cd RPi.gpio
python3 setup.py build
python3 setup.py install
```

附：GPIO 模块的使用方法

导入该模块

```
 import RPi.GPIO as GPIO
```

设置引脚编号系统

```
GPIO.setmode(GPIO.BOARD)
GPIO.setmode(GPIO.BCM)
```

BOARD 与 BCM 为两种不同的编号系统，前者按板上 2*20 针脚位置编号，后者按 BCM283X 芯片引出的 GPIO 下标编号。

本文统一采用 BCM 编号系统。

设置输入/输出功能

```
GPIO.setup(channel,direction)
```

Direction 可选参数为 GPIO.OUT/GPIO.IN
对于 GPIO.OUT，还可以通过指定 initial 参数确定初始点位水平
对于 GPIO.IN，可以指定 pull_up_down 参数确定使用上拉/下拉电阻
当 direction 为 GPIO.OUT 时，可以使用 GPIO.output(channel,GPIO.LOW)或 GPIO.output(channel,GPIO.HIGH)设置电位输出
当 direction 为 GPIO.IN 时,可以使用 GPIO.input(channel)来获取电位值
使用 pwm 功能

```
GPIO.setup(channel, GPIO.OUT)
p = GPIO.PWM(channel, frequency)
p.start(dc) # dc: 占空比0.0<=dc<=100.0
p.ChangeFrequency(freq) # 修改频率
p.ChangeDutyCycle(dc)  # 修改占空比
p.stop() # 停止pwm
```

（三）代码实现：

由于小车上所有的硬件都只有一组，所以在实现代码的时候，所有类都使用单例模式进行开发。

单例模式是一种常用的软件设计模式,通过单例模式可以保证系统中应用该模式的类一个类只有一个实例。

在使用相关类的时候，只需要 import 对应的单例对象即可使用。

1. 蜂鸣器（carFacility.Buzzer）

在实验设备上，蜂鸣器对应的 GPIO 端口为 4，在使用时，我们需要将其设置为输出模式，并使用 PWM,这里我们默认使用的频率为 300,占空比为 50。在使用过程中如需调整，还可使用 setDutyCycle 与 setFreq 两个成员方法修改。

在此类中，实现了一个常用的方法:doBim(timeDuty)，这个方法接受一个参数 timeDuty 表示蜂鸣持续时间，调用此方法将蜂鸣 timeDuty 秒。并且此方法采用多线程实现，在蜂鸣过程中其他任务还可继续执行。

如果需要使用阻塞方法，可以使用 bimb(timeDuty)。

```python
#!/usr/bin/python3
# 文件路径：./carFacility/Buzzer.py
import RPi.GPIO as GPIO
import _thread
import time
class Buzzer(object):
    BUZZER = 4
    BUZZER_PWM = None
    def __init__(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.BUZZER, GPIO.OUT)
        self.BUZZER_PWM = GPIO.PWM(self.BUZZER, 300)

    def startBuzzer(self):
        self.BUZZER_PWM.start(50)

    def setDutyCycle(self,cycle):
        self.BUZZER_PWM.ChangeDutyCycle(cycle)

    def setFreq(self,freq):
        self.BUZZER_PWM.ChangeFrequency(freq)

    def bimb(self,timeDuty):
        self.BUZZER_PWM.start(50)
        time.sleep(timeDuty)
        self.BUZZER_PWM.stop()

    """
    发出蜂鸣信号
    参数：timeDuty:蜂鸣时长，单位为秒（s）
    """
    def doBim(self,timeDuty):
        try:
            _thread.start_new_thread(self.bimb,(timeDuty,))
        except:
            print("fail")

buzzer = Buzzer()
```

需要使用此类时，只需要使用 from carFacility.Buzzer import buzzer 然后调用 buzzer 对象的方法即可，其他类的调用方法类似，后面不再重复。

2. 超声波传感器（carFacility. DistanceDetector）

在实验设备上，超声波传感器 Echo 对应的 GPIO 端口为 27，在使用时，我们需要将其设置为输入模式，Trig 对应的 GPIO 端口为 22，在使用时，我们需要

将其设置为输出模式。

按照前文所述原理，我们可以完成如下代码。

```
#!/usr/bin/python3
# 文件路径：./carFacility/DistanceDetector.py
import RPi.GPIO as GPIO
import time
class DistanceDetector(object):
    echo_port=27
    trig_port=22
    def __init__(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.trig_port,GPIO.OUT)
        GPIO.setup(self.echo_port,GPIO.IN)


    """
    返回测得前方障碍物的距离
    """
    def check_distance(self):
        sleepTime=0.000015
        GPIO.output(self.trig_port,GPIO.HIGH)
        time.sleep(sleepTime)
        GPIO.output(self.trig_port,GPIO.LOW)

        channel=GPIO.wait_for_edge(self.echo_port,
                                   GPIO.RISING,
                                   timeout=1000)

        if channel is None:
            print('Check fail at this time')
            return
        else:
            t1=time.time()
        channel=None
        channel=GPIO.wait_for_edge(self.echo_port,
                                   GPIO.FALLING,
                                   timeout=1000)
        if channel is None:
            print('Check fail at this time')
            return
        else:
            t2=time.time()
        return (t2-t1)*34000/2

distanceDetector = DistanceDetector()
```

这里使用了 GPIO 模块提供的 wait_for_edge 方法，此方法接受三个参数，

channel 代表要监听的端口，第二个参数代表要监听的事件，第三个参数代表等待时间，该方法是一个阻塞方法，可以在事件未发生时一直阻塞在该行。基于此，我们可以在执行完此行后立即获取时间，即可得到开始时间与结束时间，进而计算出时间差。

我们可以调用 checkDistance()获得与障碍物之间的距离，单位为厘米。

3. 遥控器（carFacility. IR_remote）

红外遥控器是常见的控制设备，Linux 系统为其提供了完善的驱动程序，我们可以按照此方法使用红外遥控接收驱动来简化编码。

①加载设备树文件，在系统引导区的 config.txt 文件下加入如下行。

```
dtoverlay=gpio-ir,gpio_pin=17,gpio_pull=up
```

②在 Linux 系统启动后加载 GPIO 红外遥控接收端驱动。

```
modprobe gpio-ir-recv
```

③设定 NEC 协议。

```
echo "nec" >/sys/class/rc/rc0/protocols
```

这样操作之后，我们便可以在/dev/input/event0 这个文件下获得红外信号的信息。每次读取此文件我们可以获得 48 个字节数据，我们读取键位对应部分即可得知按下的是那个按键。

```
import threading
import time
key_mapper={"45":"CH-","46":"CH","47":"CH+","44":"|<<","40":">>|","43":">||",
            "07":"-","15":"+","09":"EQ","16":"0","19":"100+","0d":"200+",

 "0c":"1","18":"2","5e":"3","08":"4","1c":"5","5a":"6","42":"7","52":"8","4a":"9"}



class IR_remote(object):
    def __init__(self):
        self.data=open('/dev/input/event0','rb')
        self.cache=None
        self.func_dirc={}
        self.cache_time=0

    def read(self):
        d=self.data.read(48)
        key=d.hex()[40:42]
        return key

    def readNoTwice(self):
        while True:
            temp=self.read()
            if temp!=None:
                temptime = time.time()
                if temptime - self.cache_time > 0.1:
                    print(temptime,self.cache_time)
                    self.cache=temp
                    print(key_mapper[temp])
                    self.cache_time=temptime
                    try:
                        if self.func_dirc[temp]!=None:
                            self.func_dirc[temp]()
                    except:
                        print("no function bind")
                    temp=None

    """
    使用此方法来启动按键读取，如果不使用，bind function将会没有作用
    """
    def readKey(self):
        try:
            threading.Thread(target=self.readNoTwice,args=()).start()
        except:
            print("fail")

    """
    为某一特定按键绑定事件
    key:按键键值，func:按键响应方法
    """
    def bind_function(self,key,func):
        self.func_dirc[key]=func

ir_remote=IR_remote()

```

在这份代码中，我们使用 Keymapper 表示按键之间的映射关系，便于查看及使用。

在这个类中，我们实现了按键功能的动态绑定与自动监听。

当我们使用 bind_function(key,func)时，我们可以为键值为 key 的按键绑定一个事件 func，当调用 readKey()之后系统会开始监听按键信息，一旦有按键按下，程序便会自动到 func_dirc，即已绑定的方法表中查看该按键是否已有绑定函数，如有，则会执行该函数。

当需要替换按键功能时，再次调用绑定函数即可。

实现了这个方法可以避免为每个按键写一个时间监听，能够增强灵活性。

4. 红外传感器（避障用）(carFacility. IR_SENSOR)

避障用红外传感器在车上有两个,左侧对应GPIO端口为16,右侧对应为19,在使用时要将这两个端口设置为输入模式。

当端口 input 对应为 0 时，说明检测到了反射信号。

在这个类中，实现了 safeBefore(),detect(side),getUnsafeSise(),三个方法，能够简化避障应用的编写。

```
import RPi.GPIO as GPIO
class IR_SENSOR(object):
    IR_SENSOR_R = 19
    IR_SENSOR_L = 16
    def __init__(self):
        GPIO.setup(self.IR_SENSOR_L, GPIO.IN)
        GPIO.setup(self.IR_SENSOR_R, GPIO.IN)

    """
    检测前方是否有障碍物
    side:"LEFT","RIGHT",表示选择读取的传感器侧
    """
    def detect(self,side):
        if side=="LEFT":
            return GPIO.input(self.IR_SENSOR_L)
        elif side=="RIGHT":
            return GPIO.input(self.IR_SENSOR_R)


    """
    检测前方是否安全
    """
    def safeBefore(self):
        if self.detect("LEFT")==0 or self.detect("LEFT")==0:
            return False
        else:
            return True

```

```
    def getUnsafeSide(self):
        if self.detect("LEFT") == 0 and self.detect("RIGHT") == 0:
            return "BOTH"
        elif self.detect("LEFT")==0:
            return "LEFT"
        elif self.detect("RIGHT")==0:
            return "RIGHT"

ir_sensor = IR_SENSOR()
```

5. 红外传感器（寻迹用）（carFacility. IR_SENSOR_ADC）

避障用红外传感器在车上有两个，左侧对应GPIO端口为16，右侧对应为19，在使用时要将这两个端口设置为输入模式。

当端口 input 对应为 0 时，说明检测到了反射信号。

此代码较长，仅展示关键部分，完整代码可以在 github 上下载。

```
def AnalogRead(self):
    val=[0]*6
    for j in range(6):
        GPIO.output(CS,GPIO.LOW)
        for i in range(10):
            if i < 4:
                bit = (((j)>>(3-i)) & 0x01)
                GPIO.output(ADDRESS,bit)
            val[j]<<=1
            val[j]|=GPIO.input(DATAOUT)
            GPIO.output(CLOCK, GPIO.HIGH)
            GPIO.output(CLOCK,GPIO.LOW)
        GPIO.output(CS,GPIO.HIGH)
        time.sleep(0.0001)
    average = (val[1]+val[2]+val[3]+val[4]+val[5])/5
    for i in range(1,6):
        val[i]=val[i]-average
    self.value=val.copy()
```

另实现了一个方法 check_side()，具体思路会在后面的自动寻迹中做进一步的解释。

6. LED（carFacility. LED）

```
from neopixel import *
import time
import random
class LED(object):
    LED_COUNT=4
    LED_PIN=18
    LED_FREQ_HZ=800000
    LED_DMA=10
    LED_BRIGHTNESS=255
    LED_INVERT=False
    LED_CHANNEL=0
    LED_STRIP=ws.WS2812_STRIP
    red=255
    green=0
    blue=0
```

```
    colorList={"0":{"red":0,"green":0,"blue":0},"1":{"red":0,"green":0,"blue":0},"2":{"red":0,"green":0,"blue":0},"3":{"red":0,"green":0,"blue":0},"4":{"red":0,"green":0,"blue":0}}
    strip=None

    def __init__(self):
        self.strip = Adafruit_NeoPixel(self.LED_COUNT,
                                       self.LED_PIN,
                                       self.LED_FREQ_HZ,
                                       self.LED_DMA,
                                       self.LED_INVERT,
                                       self.LED_BRIGHTNESS,
                                       self.LED_CHANNEL)
        self.strip.begin()
```

```
28
29      def lightOn(self,index,red=255,green=0,blue=0):
30          self.colorList[str(index)]["red"]=red
31          self.colorList[str(index)]["green"] = green
32          self.colorList[str(index)]["blue"] = blue
33          self.strip.setPixelColor(index,(red<<16|green<<8|blue))
34          self.strip.show()
35
36      """
37      车上LED光源随机颜色亮起
38      """
39      def randomColor(self):
40          for i in range(0, 4):
41              self.lightOn(i, int(random.random() * 255), int(random.random() * 255),
    int(random.random() * 255))
42
43      def lightOff(self):
44          for i in range(0,4):
45              self.lightOn(i, 0,0,0)
46
47  led = LED()
48
49  if __name__ == '__main__':
50      for i in range(4):
51          led.strip.setPixelColor(i, (0 << 16 | 0 << 8 | 0))
52          time.sleep(0.2)
53          led.strip.show()
54
```

代码实现了 lightOn(index,red,green,blue)函数，可以控制某一个位置上的 LED 灯颜色，randomColor()可以实现小车上的 LED 灯光随机颜色亮起。LightOff()可以实现关闭所有灯光。

7. 舵机（carFacility.servo）

设备上有两个舵机，可以控制摄像头在两个方向上移动，从而改变摄像头朝向。

要使用舵机，需要先加载两个驱动，具体做法是，在终端分别执行以下两条指令。

```
1  modprobe -v i2c-bcm2835
2  modprobe -v i2c-dev
```

也可以将这两条代码加入/etc/rc.d/rc.conf 的末尾，从而实现开机自动加载，避免每一次都需要手动挂载。

```python
import Adafruit_PCA9685
import time
class Servo(object):
    pwm=None
    degree=[60,60]
    def __init__(self):
        self.pwm=Adafruit_PCA9685.PCA9685()
        self.pwm.set_pwm_freq(50)
        self.set_servo_angle(0,self.degree[0])
        self.set_servo_angle(1, self.degree[1])

    """
    channel:0左右旋转，1上下旋转
    """
    def set_servo_angle(self,channel,angle):
        value=4095*(angle*11+500)/20000
        print(value)
        self.pwm.set_pwm(channel,0,int(value))

    def change(self,side,speed):
        channel=1
        sign=-1
        if side=="LEFT" or side=="RIGHT":
            channel=0
        if side=="RIGHT" or side=="DOWN":
            sign=1
        changed=self.degree[channel]+sign*speed
        if (changed<100 and changed>30 and channel==0) or (changed<150 and changed>10
and channel==1):
            self.degree[channel]=changed
        self.set_servo_angle(channel,self.degree[channel])

servo=Servo()
```

8. 直流电机（carFacility.wheel）

代码较长，故不在此处展示。

为了实现小车的转向，需要使两个轮形成差速或者反转，本代码基于反转实现，实现了 setDirection,resume_forward,steer 等函数用于调整方向，恢复直行，转向等等功能。

四、应用层

（一）自动寻迹：

1. 定义

这里定义的自动寻迹是指小车自动寻找轨道路径，并且沿着轨道路径行走。

轨道路径的要求是白色作为底色，黑色作为轨道路径。

2. 原理

小车上提供了 5 个可以实现反射信号量化的红外传感器，基于此硬件，可以有如下的避障方案。

总体思路：使用红外传感器判断小车处在轨道左侧/右侧/轨道上，并根据小车所在轨道的对应位置来调整小车的行为。

①如何判断小车处在轨道左侧/右侧/轨道上

对于红外传感器，黑色轨道的反射率小于白色地面，这样可以使轨道与地面之间形成反差，当传感器在黑色轨道上时，传感器返回值将会小于阈值，当传感器在白色地面上时，传感器会大于阈值，阈值可以通过实验获得，这样可以确定每一个传感器是否在轨道之上。

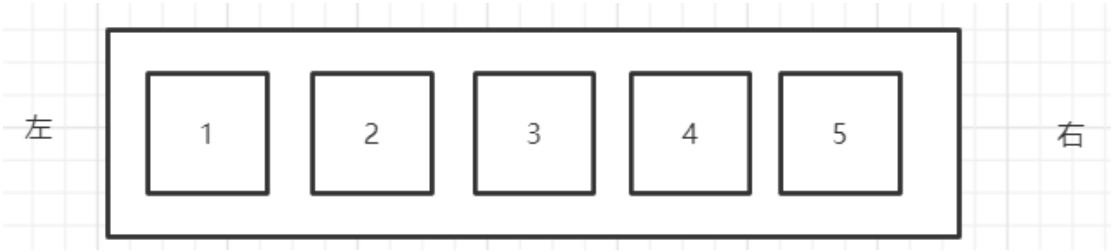


Fig.4 传感器示意图

如 Fig.4 所示，小车下方 5 个传感器的排列如图，如果 3 在轨道之上，我们可以断定小车在轨道上，当 3 不在轨道上时，我们可以推断存在三种情况 a.小车在轨道左侧 b.小车在轨道右侧 c.小车完全离开轨道

a 情况下,4 或 5 将会在轨道上，1 和 2 都不在轨道上。

b 情况下,1 或 2 将会在轨道上，4 和 5 都不在轨道上。

c 情况下,12345 都不在轨道上

②如何调整

3 在轨道上时，直行即可

在 a 情况下，小车要右转，直至小车重回轨道中央

在 b 情况下，小车要左转，直至小车重回轨道中央

c 情况比较麻烦,小车会采取一些随机尝试的方法来寻找轨道路径。

3. 代码实现

判断小车在轨迹左侧/右侧/轨道上

```
    """
    LEFT:小车向左偏离线路
    RIGHT:小车向右偏离线路
    NORMAL:正常运行
    """
    def check_side(self):
        if self.check_distance(3)>DIFF:
            if self.check_distance(1)>DIFF and self.check_distance(2)>DIFF:
                if self.check_distance(4)<DIFF or self.check_distance(5)<DIFF:
                    return "LEFT"
                elif self.check_distance(4)>DIFF or self.check_distance(5)>DIFF:
                    return "ABNORMAL"
            else:
                if self.check_distance(4)>DIFF and self.check_distance(5)>DIFF:
                    return "RIGHT"
                else:
                    return "ABNORMAL"
        else:
            return "NORMAL"

```

调整小车运行（autoDetectMode）

```
def auto_detect_progress():
    global isForward, isRight, isLeft,isWorking,countTime,lastTime
    print("working")
    while isWorking:
        a = ir_sensor_adc.check_side()
        if a == "NORMAL":
            countTime=0
            if not isForward:
                isForward = True
                wheels.start(speed, speed)
            if isRight:
                isRight = False
                wheels.resume_forward("LEFT")
                wheels.start(speed, speed)
            if isLeft:
                isLeft = False
                wheels.resume_forward("RIGHT")
                wheels.start(speed, speed)
        elif a == "RIGHT":
            wheels.steer_by_side("LEFT")
            isRight = True
        elif a == "LEFT":
            wheels.steer_by_side("RIGHT")
            isLeft = True
        else:
            # wheels.stop()
            # isForward = False
            countTime+=1
            print(countTime)
```

```
            if isRight:
                isRight = False
                wheels.resume_forward("LEFT")
                wheels.start(speed, speed)
            if isLeft:
                isLeft = False
                wheels.resume_forward("RIGHT")
                wheels.start(speed, speed)
            if countTime<=10:
                # 无法寻路后的调整策略
                if lastTime==1:
                    wheels.steer_by_side("RIGHT")
                    isLeft = True
                    lastTime=0
                    time.sleep(countTime * 0.003)
                else:
                    wheels.steer_by_side("LEFT")
                    isRight = True
                    lastTime=1
                    time.sleep(countTime * 0.003)
            elif countTime<=500:
                if random.random()>0.5:
                    wheels.steer_by_side("RIGHT")
                    isLeft = True
                    time.sleep(countTime*0.005)
                else:
                    wheels.steer_by_side("LEFT")
                    isRight = True
                    time.sleep(countTime * 0.005)
            else:
                wheels.stop()
        time.sleep(update_duty)
    print("thread was stopped")
```

其中 update_duty 为更新周期，表示小车多长时间再探测一次，此数值越小，准确度越高，但也不能将周期调整的过短，否则效果也会变差。

当小车离开轨道时，小车会尝试左右随机转动，并且转动角度大小会随着离开轨道时间变长而变大，小车还会尝试前后行走，这样会使得小车有一定可能重新找到轨道。

（二）简单避障：

1. 原理

小车左右两侧各有一个红外传感器，可以获得左右是否有障碍物（不安全）信息，当前方无障碍物时可以直行，前方左右均有障碍物时可以向某一个方向旋转 90 度，左侧有障碍物时可以向右侧旋转，右侧有障碍物时可以向左侧旋转，调整过方向之后再继续直行。这样便实现了最为简单的避障。

2. 代码实现

```
from carFacility import Wheels, IR_SENSOR
import time

ir_sensor=IR_SENSOR.ir_sensor
wheel=Wheels.wheels

isStop=False
isWorking=False
leftTurning=True
rightTurning=True

def startAvoid():
    global isStop,isWorking,leftTurning,rightTurning
    wheel.start(10,10)
    isWorking=True
    while isWorking:
        if not ir_sensor.safeBefore():
            safeState = ir_sensor.getUnsafeSide()
            print(safeState)
            if safeState == "BOTH" or safeState=="RIGHT":
                wheel.steer_by_side("LEFT")
                time.sleep(0.2)
                wheel.resume_forward("RIGHT")
            elif safeState=="LEFT":
                wheel.steer_by_side("RIGHT")
                time.sleep(0.2)
                wheel.resume_forward("LEFT")
        else:
            wheel.start(10, 10)
            wheel.resume_forward("LEFT")
            wheel.resume_forward("RIGHT")
    wheel.stop()
```

（三）功能组合：

在 Main.py 代码中实现了一个功能组合封装的样例，Main 运行之后可以实现遥控器控制，其中按键 1 可以实现随机切换 LED 颜色，按键 4 可以实现关闭所有灯光，按键 2 开始自动寻迹，按键 3 停止自动寻迹，按键 5 小车向前运行，按键 6 小车停止运行。

```
ir_remote.bind_function("0c",led.randomColor)
ir_remote.bind_function("18",start_auto_detect_call)
ir_remote.bind_function("5e",stop_auto_detect_call)
ir_remote.bind_function("08",led.lightOff)
ir_remote.bind_function("1c",runCar)
ir_remote.bind_function("5a",stopCar)
```

读者同样可以按照这样的思路实现其他的代码组合，从而充分利用相关的功能。

五、WEB 服务层

1. 原理

为了实现更加良好的用户界面，在应用层之上，系统又基于 Vue.js 与 Flask 两个框架开发了用户界面。

Flask 框架是 Python 常用的 web 开发框架，可以快速进行 web 服务开发。而 Vue.js 是常用的 javascript 渐进式开发框架，在使用组件库的情况下可以很快地实现 web 服务。

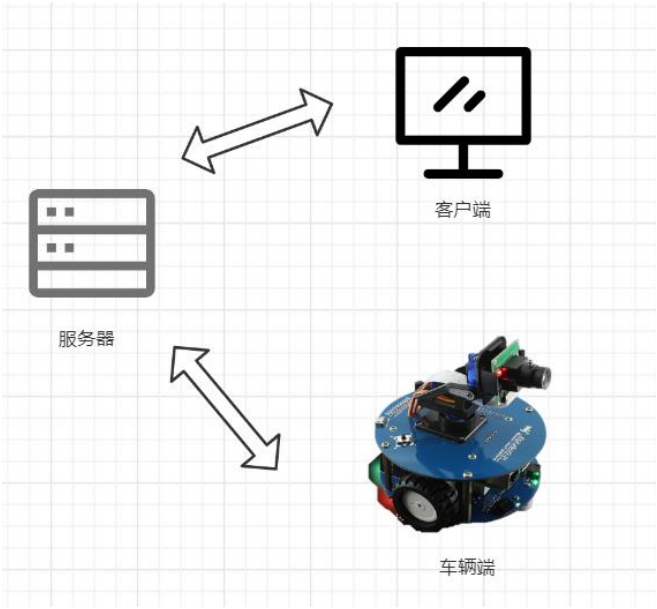


Fig.5 web 架构设想

嵌入式设备 web 实现比较常用的方法是 Fig.5 所示的结构，在原创搭建一个服务器，车辆端第一次连接时需要进行配网接入互联网，从而与服务器取得通信，再由服务器承担消息转发任务。客户端发送的指令都由服务器进行转发，这样可以使得应用具有比较强的拓展能力，还方便远程管理。

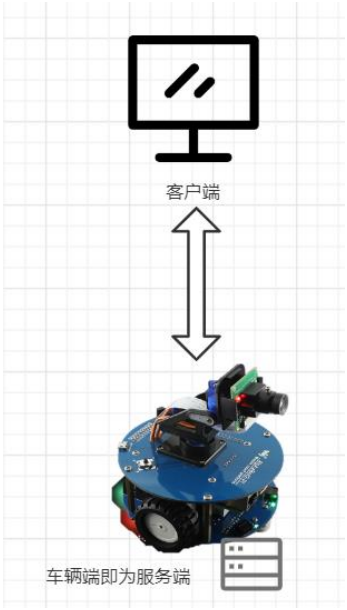


Fig.6 web 架构实现

为了简化实现，充分利用设备资源，我选择了使用 Fig.6 的架构，车辆上集成了服务器软件，可以直接和客户端进行通信，这样实现相对简单。

在 web 实现上我使用了前后端分离的技术架构，客户端向车辆端（服务端）发送 http 请求以获取数据，服务端以 json 或纯文本格式向前端返回数据，这样在界面实现与服务实现上也会更加模块化，更加便于实现。

2. 接口列表

接口名称	请求方法	功能
/	GET	获取一个字符串”Hello World!”
/status	GET	获取车辆当前状态
/keyBind	GET	获取遥控器绑定方法信息
/ledInfo	GET	获取 LED 灯光信息
/carSpeed	GET	获取车辆速度信息
/buzzer	GET	鸣笛
/setLed?index=_&red=_&green=_&blue=_	GET	设置 led 灯光
/setSevro?side=_&speed=_	GET	调整舵机控制摄像头角度
/runCar?dir=_&speed=_	GET	使车辆按设置的

		速度与方向前进
/startAutoDetect	GET	开始自动寻迹
/stopAutoDetect	GET	停止自动寻迹
/programRun	POST	可编程执行

Table.1 已经实现的接口

3. 实验步骤

在进行部署之前，请确定你的设备已经接入了智能车提供的热点，并确定智能车在热点环境中的 ip 地址，并修改前端项目的.env.development 下的 VUE_APP_BASE_API 参数改为 http://智能车 ip:5000

①前端部署

你可以选择将前端项目部署在你的个人电脑上或者部署在智能车上，他们的区别在于前者你可以直接访问 http://loocalhost:8000,而后者，你需要访问 http://智能车 ip:8000。

要部署前端，需要先配置 Node.js 环境，你可以访问 https://nodejs.org/zh-cn/ 下载对应版本的 node.js 环境并按照官网教程进行安装。

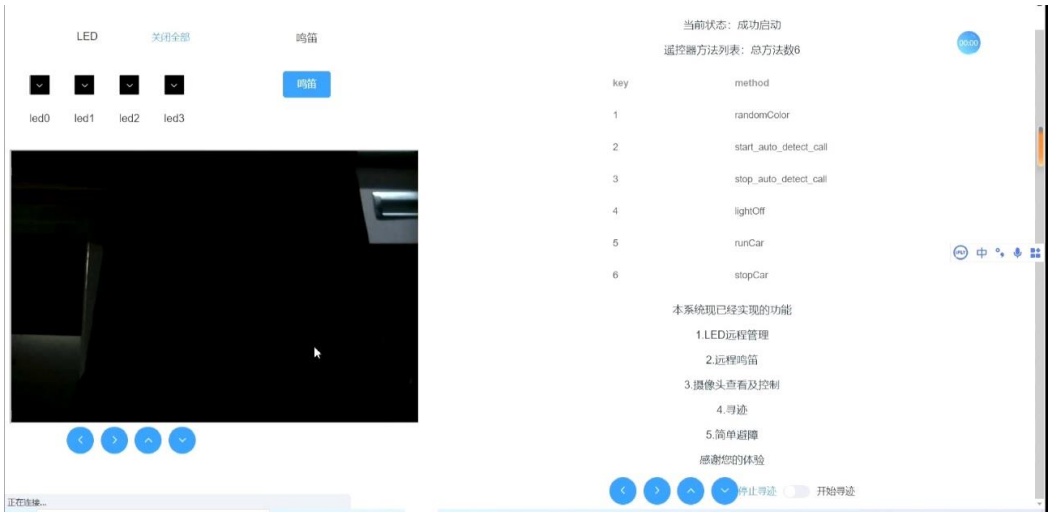
Node.js 环境安装完成后切换到前端目录下，执行 npm install 补全依赖，执行 npm serve 运行前端项目。

此时，你已经可以在浏览器中访问前端了。

②后端部署

直接运行程序中的 StatusController.py 即可使后端程序运行起来。

4. 运行效果



智能灯控：左上角的 LED 卡片可以查看 LED 灯光状态并控制 LED 灯光

鸣笛按钮：使车辆鸣笛

摄像头遥控：可以控制摄像头朝向角度

车辆遥控（右下角）：可以遥控车辆运动，可以控制车辆寻迹

查看遥控器按键表：右上角表格可以查看

状态监控:右上角

六、可编程功能

1. 原理

Web 前端实现了将控制流转换为标准 json 格式的功能，后端可以解析 json 格式获取需要执行的任务列表，然后按照顺序及对应要求执行即可。

这项功能的本质是实现了 RPC 远程调用与命令解析。

2. 代码实现

```
@app.route("/programRun",methods=['POST'])
def programRun():
    rawdata=request.data
    actions=json.loads(rawdata)
    print(actions)
    for action in actions["data"]:
        for i in range(int(action["repeat"])):
            if action["name"]=="LIGHTON":
                Main.led.randomColor()
            elif action["name"]=="LIGHTOFF":
                Main.led.lightOff()
            elif action["name"]=="BUZZER":
                Main.buzzer.doBim(1)
                time.sleep(1)
            elif action["name"]=="SLEEP":
                time.sleep(0.2)
            else:
                Main.changeDir(action["name"],20)

    return "success"
```

七、系统存在的问题与解决方案

（一）代码架构仍然不够完善

部分模块代码结构较为混乱，需要进一步重构代码，优化实现方案。

（二）小车行驶中完全离开路线无法解决

考虑采取与其他方案相结合，比如在出现转角时改变寻迹策略，尽可能避免完全驶离路线的状况。

（三）可编程结构不够通用

定义更加通用的结构，进一步优化执行模式。