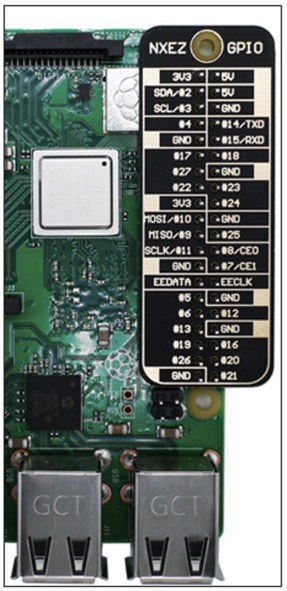
# 第 3 章 树莓派的 GPIO

通俗地说，GPIO(General Purpose I/O Ports，通用输入输出端口)就是一些引脚，可以通过它们输出高低电平或者读入引脚的状态(高电平或低电平)。树莓派的 GPIO 如图 3.1 所示。




(a) 实物参考卡片

| BCM 编码 | 功能名 | 物理引脚 BOARD编码 | | 功能名 | BCM 编码 |
|---|---|---|---|---|---|
| | 3.3V | 1 | 2 | 5V | |
| 2 | SDA.1 | 3 | 4 | 5V | |
| 3 | SCL.1 | 5 | 6 | GND | |
| 4 | GPIO.7 | 7 | 8 | TXD | 14 |
| | GND | 9 | 10 | RXD | 15 |
| 17 | GPIO.0 | 11 | 12 | GPIO.1 | 18 |
| 27 | GPIO.2 | 13 | 14 | GND | |
| 22 | GPIO.3 | 15 | 16 | GPIO.4 | 23 |
| | 3.3V | 17 | 18 | GPIO.5 | 24 |
| 10 | MOSI | 19 | 20 | GND | |
| 9 | MISO | 21 | 22 | GPIO.6 | 25 |
| 11 | SCLK | 23 | 24 | CE0 | 8 |
| | GND | 25 | 26 | CE1 | 7 |
| 0 | SDA.0 | 27 | 28 | SCL.0 | 1 |
| 5 | GPIO.21 | 29 | 30 | GND | |
| 6 | GPIO.22 | 31 | 32 | GPIO.26 | 12 |
| 13 | GPIO.23 | 33 | 34 | GND | |
| 19 | GPIO.24 | 35 | 36 | GPIO.27 | 16 |
| 26 | GPIO.25 | 37 | 38 | GPIO.28 | 20 |
| | GND | 39 | 40 | GPIO.29 | 21 |

(b) 引脚对照关系

图 3.1　GPIO 的引脚

GPIO 是一个比较重要的概念。用户可以通过 GPIO 与硬件进行数据交互(如 UART)、控制硬件工作(如 LED、蜂鸣器等)、读取硬件的工作状态信号(如中断信号)等。利用它们可以与外界交互，对树莓派进行各种各样的扩展，作为可编程开关控制其他事务，以及接收外界的信息。

GPIO 的使用非常广泛。掌握了 GPIO，就具备了操作硬件的能力。数字艺术家可以使用它们创建交互式显示，机器人建造者可使用它们提升自己的作品。

在开始构建电路之前，首先需要知道如何连接树莓派的 GPIO。

### 1. 内转外接头

使用内接转外接头是最简单的选择。作为转接头，内接头可以连接 GPIO 引脚，外接头可以插入面包板，这是访问 GPIO 的最简便方法。

### 2. GPIO 扩展板（T 型扩展板）

可以使用 40P 排线将树莓派的 GPIO 引脚通过 GPIO 扩展板连接起来，如图 3.2 所示。注意，连接的时候，必须将 40P 排线上有“小三角”符号的一端（1 号脚）对准树莓派的 1 号脚（图 3.2 中被排线遮住的下方引脚）。如果插反，则会导致树莓派被烧坏。

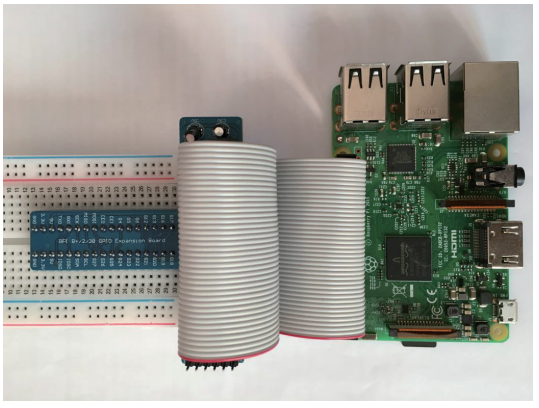


图 3.2　GPIO 扩展板与树莓派通过 40P 排线连接

一旦在 GPIO 上连接了其他硬件，就有可能直接给 CPU 供电，也就有可能将其损坏。绝对不能将 3.3V 电压连接到 GPIO 上。这一点非常重要，是需要特别注意。

与使用内转外接头相比，GPIO 虽然没有提供任何新特性，但是看起来更整洁，也不容易混淆引脚。

### 3. 无焊面包板

无论选择上述哪种方式，都需要使用面包板。通过它可以快速地将各个组件的电路连接在一起，完成之后可以轻松拆除。

面包板有不同的尺寸，但基本排列十分类似。在典型面包板的长边上，有两条平行接口，用来连接电源的正负极。它们中间是两排插孔，插孔中间留有间隙。与长边垂直的每 5 个孔板常用一条金属条连接。面包板中央的一条凹槽，是为需要集成电路、芯片的试验而设计的。

可以将元器件直接插入孔中，使用公对公连接线或者小段单芯线缆将各个元件连接起来。

通过树莓派的 I/O 口可以外接很多外设，如伺服电动机、红外发送接收模块、继电器、步进电动机、各类兼容传感器、屏幕等，通过这些外设可以进行很多有趣的设计。

### 4. RPi.GPIO

对于 Python 用户，可以使用 RPi.GPIO 提供的 API 对 GPIO 进行编程，RPi.GPIO 是一个控制树莓派 GPIO 通道的模块，它提供了一个类来控制树莓派上的 GPIO。通常在文件开头使用 import RPi.GPIO as GPIO 导入。

大多数树莓派的镜像默认安装了 RPi.GPIO，可以直接使用它。如果没有，则可以通过

执行命令 sudo apt-get install python-dev 进行安装。

## 3.1 LED

树莓派 GPIO 控制输出的入门案例都是从控制 LED(Light Emitting Diode,发光二极管)开始。

### 3.1.1 七彩 LED

图 3.3 所示的七彩 LED 上电后,会自动闪烁内置的颜色,常用于制作迷人的灯光效果,原理如图 3.4 所示。

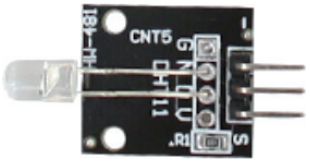



图 3.3 七彩 LED

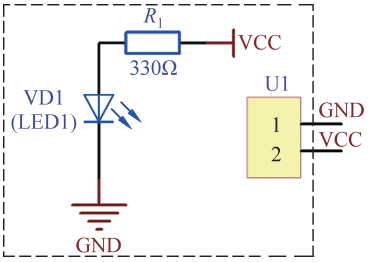



图 3.4 七彩 LED 的原理

树莓派的功能名、BCM 编码、物理引脚与七彩 LED 引脚之间的关系如表 3.1 所示,T 型转接板与 RGB LED 之间的连线如图 3.5 所示。实物如图 3.6 所示,七彩 LED 的 S 引脚连接 VCC,中间引脚连接 GND。

表 3.1 树莓派与七彩 LED 引脚之间的关系

| 功 能 名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | 七彩 LED 模块的引脚 |
|---|---|---|---|
| 5V | 5V | 2、4 | VCC |
| GND | GND | GND | GND |

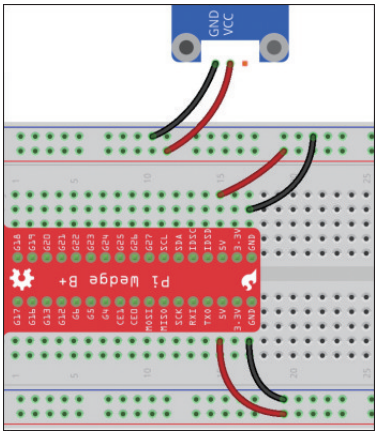



图 3.5 七彩 LED 的连线

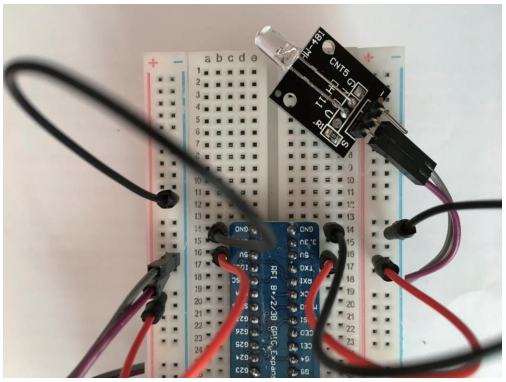



图 3.6 七彩 LED 的实物

### 3.1.2 双色 LED

双色 LED 能够发出两种不同的颜色，经常用作电视机、数字照相机和遥控器的指示灯，实验用的双色 LED 如图 3.7 所示（“—”引脚对应 GND，中间引脚对应 R，S 引脚对应 G）。

本实验将引脚 R 和 G 连接到树莓派的 GPIO，对树莓派进行编程，将 LED 的颜色从红色变为绿色，然后使用 PWM（Pulse Width Modulation，脉宽调制）混合成其他颜色。原理如图 3.8 所示。


图 3.7 双色 LED

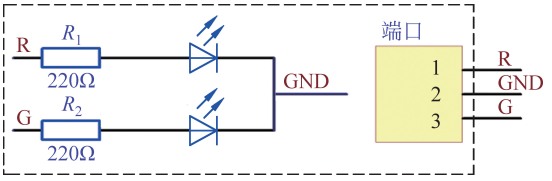



图 3.8 双色 LED 的原理

#### 1. 电路连接

树莓派的功能名、BCM 编码、物理引脚与双色 LED 引脚之间的关系如表 3.2 所示，T 型转接板与双色 LED 之间的连线如图 3.9 所示，实物如图 3.10 所示。注意，不同的实物引脚顺序有所不同，图 3.7 所示的实物，其“—”引脚对应 GND，中间引脚对应 R，S 引脚对应 G。

表 3.2 树莓派与双色 LED 引脚之间的关系

| 功 能 名 | BCM 编码（T 型转接板） | 物理引脚（BOARD 编码） | 双色 LED 模块的引脚 |
|---|---|---|---|
| GPIO.0 | 17 | 11 | R |
| GPIO.1 | 18 | 12 | G |
| GND | GND | GND | GND |

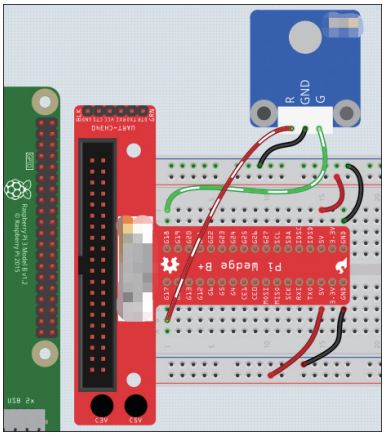



图 3.9 双色 LED 的连线

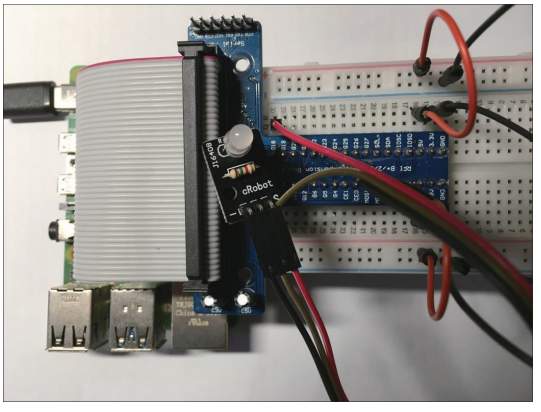



图 3.10 双色 LED 的实物

### 2. 软件编写

在 pi 目录下新建文件夹 book，在 book 目录下新建文件 ch3.1.2.py，代码如下：

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time

colors = [0xFF0000, 0x00FF00, 0x0FF000, 0xF00F00]
pins = {'pin_R':11, 'pin_G':12}

GPIO.setmode(GPIO.BOARD)
for i in pins:
    GPIO.setup(pins[i], GPIO.OUT)
    GPIO.output(pins[i], GPIO.HIGH)

p_R = GPIO.PWM(pins['pin_R'], 2000)
p_G = GPIO.PWM(pins['pin_G'], 2000)

p_R.start(0)
p_G.start(0)

def map(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def setColor(col):
    R_val = (col & 0xFF0000) >> 16
    G_val = (col & 0x00FF00) >> 8

    R_val = map(R_val, 0, 255, 0, 100)
    G_val = map(G_val, 0, 255, 0, 100)

    p_R.ChangeDutyCycle(R_val)
    p_G.ChangeDutyCycle(G_val)

def loop():
    while True:
        for col in colors:
            setColor(col)
            time.sleep(0.5)

def destroy():
    p_R.stop()
    p_G.stop()
```

```
        for i in pins:
            GPIO.output(pins[i], GPIO.HIGH)
        GPIO.cleanup()

    if __name__ == "__main__":
        try:
            loop()
        except KeyboardInterrupt:
            destroy()
```

(1) 语句 if __name__ == "__main__": 的含义。对于很多编程语言来说，程序都必须要有一个入口，比如 C、C++，以及完全面向对象的编程语言 Java、C# 等。

C 和 C++ 都需要有一个 main()函数来作为程序的入口，也就是程序的运行会从 main()函数开始。同样，Java 和 C# 必须要有一个包含 main()方法的主类作为程序入口。而 Python 则不同，它属于脚本语言，不像编译型语言那样，先将程序编译成二进制再运行，而是动态的逐行解释运行。也就是从脚本第一行开始运行，没有统一的入口。

一个 Python 源码文件除了可以被直接运行外，还可以作为模块(也就是库)被导入。不管是导入还是直接运行，最顶层的代码都会被运行(Python 用缩进来区分代码层次)。而实际上在导入的时候，有一部分代码我们是不希望被运行的。

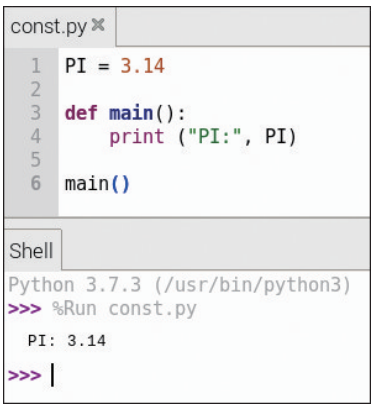



图 3.11 const.py 程序及运行结果

假设有一个 const.py 文件，代码及运行结果如图 3.11 所示。

在这个文件里定义了 PI 为 3.14，然后又写了一个 main()函数来输出定义的 PI，最后运行 main()函数就相当于对定义做一遍人工检查，看看值设置得对不对。

假设又有一个 area.py 文件，用于计算圆的面积，该文件里边需要用到 const.py 文件中的 PI，那么从 const.py 中把 PI 导入到 area.py 中，代码及运行结果如图 3.12 所示。

可以看到，const 中的 main()函数也被运行了，实际上是不希望它被运行，提供 main()也只是为了对常量定义进行测试。这时，if __name__ == '__main__' 就派上了用场。修改 const.py 代码，再运行 area.py，修改的代码以及输出结果如图 3.13 所示，这才是想要的效果。

if __name__ == '__main__'就相当于是 Python 模拟的程序入口。Python 本身并没有规定这么写，这只是一种编码习惯。由于模块之间相互引用，不同模块可能都有这样的定义，而入口程序只能有一个。到底哪个入口程序被选中，这取决于 __name__ 的值。

结论：if __name__ == '__main__' 的意思就是，当模块被直接运行时，以下代码块将被运行，当模块是被导入时，代码块不被运行。

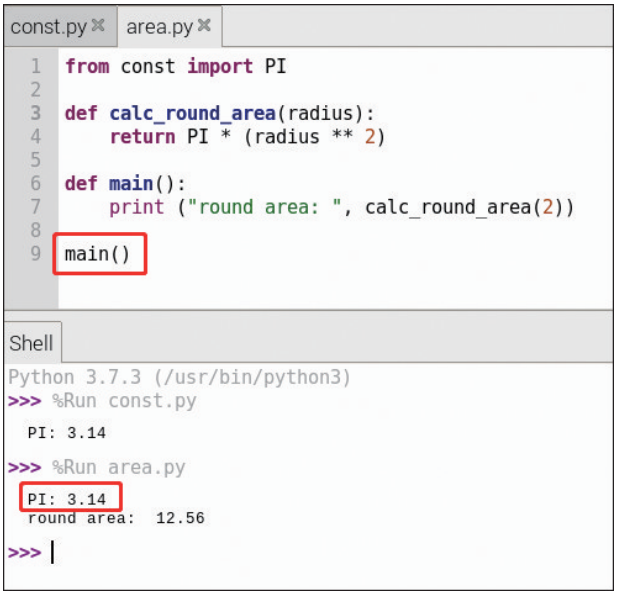



图 3.12 area.py 程序及运行结果

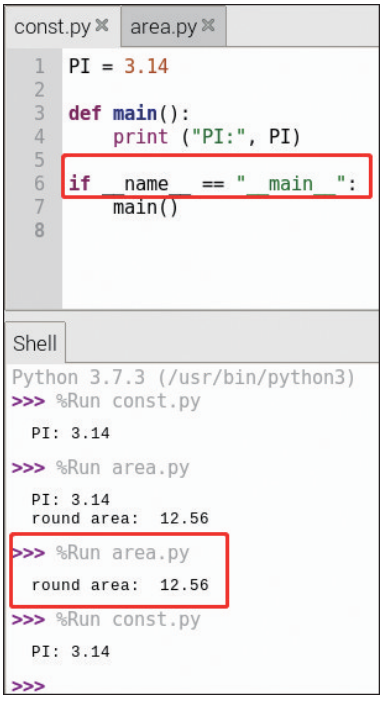



图 3.13 if __name__ == '__main__'的作用

(2) 异常处理。异常即是一个事件,该事件会在程序执行过程中发生,影响程序的正常执行。一般情况下,在 Python 无法正常处理程序时就会发生一个异常。当 Python 发生

异常时需要捕获处理它，否则程序将会终止执行。

捕捉异常可以使用 try…except 语句，try…except 语句用来检测 try 语句块中的错误，从而让 except 语句捕获异常信息并处理。

以下为简单的 try…except…else 的语法。

```
try:
<语句>                    #运行别的代码
except <名字>:
<语句>                    #如果在 try 部分引发了'name'异常
except <名字>,<数据>:
<语句>                    #如果引发了'name'异常，获得附加的数据
else:
<语句>                    #如果没有异常发生
```

在如图 3.14 所示的程序中，发生异常时执行函数 destroy()，关闭所有的 LED。其中，KeyboardInterrupt 表示用户中断执行(通常是按 Ctrl+C 键)。

const.py area.py ch3.1.1.py

```
32  def loop():
33      while True:
34          for col in colors:
35              setColor(col)
36              time.sleep(0.5)
37
38  def destroy():
39      p_R.stop()
40      p_G.stop()
41      for i in pins:
42          GPIO.output(pins[i], GPIO.HIGH)
43      GPIO.cleanup()
44
45  if __name__ == "__main__":
46      try:
47          loop()
48      except KeyboardInterrupt:
49          destroy()
```

图 3.14　异常处理

GPIO.cleanup()的作用是释放脚本中使用的 GPIO 引脚。一般来说，程序到达最后都需要释放资源，这个好习惯可以避免损坏树莓派。

正常情况下执行 loop()函数，循环执行 setColor(col) 函数和 time.sleep(0.5)方法(休息 0.5s)。

(3) #！/usr/bin/env python 的作用。这条语句的作用是为了防止操作系统用户没有将 Python 安装在默认的/usr/bin 路径里。当系统看到这一行的时候，首先会到 env 设置里查找 Python 的安装路径，再调用对应路径下的解释器程序完成操作。

(4) 设置模式 GPIO.setmode。GPIO.setmode(mode) 的 mode 参数有两个值，GPIO.BOARD 和 GPIO.BCM(注意，全是大写)。前者告诉程序按物理位置找 GPIO(或者称

channel)，后者按 GPIO 端口编号。两种模式各有各的好处，前者方便查找，后者方便程序在不同的树莓派版本上运行。

对照图 3.1 和表 3.2，树莓派 GPIO 的 0 端口(GPIO.0)，对应 BCM 编码为 17，对应物理引脚(BOARD 编码)为 11；树莓派 GPIO 的 1 端口(GPIO.1)，对应 BCM 编码为 18，对应物理引脚(BOARD 编码)为 12。

由于程序中使用了 GPIO.BOARD，所以 GPIO.setmode(GPIO.BOARD) 上面一行的代码，pins = {'pin_R': 11, 'pin_G': 12}，代表的是物理引脚 11 和 12，即 GPIO.0 和 GPIO.1。

(5) 设置 GPIO 的输入和输出 GPIO.setup。GPIO.setup(channel, mode) 的参数 channel 就是要用的 GPIO，参数 mode 分为输入 GPIO.IN 和输出 GPIO.OUT。

GPIO.output(channel, GPIO.HIGH)表示输出高电平，就是输出信号 1；GPIO.output(channel, GPIO.LOW) 表示输出低电平，就是输出信号 0。

(6) 设置调制脉宽，输出模拟信号 GPIO.PWM。树莓派本身既不能接收模拟信号，也不能输出模拟信号，要么输出 1，要么输出 0。不过可以通过改变数字信号的输出占空比(Duty Cycle，就是一个周期内 GPIO 的打开时间占总时间的比例)，使输出效果近似模拟信号。

占空比是指在一个周期内，信号处于高电平的时间占据整个信号周期的百分比，如图 3.15 所示。

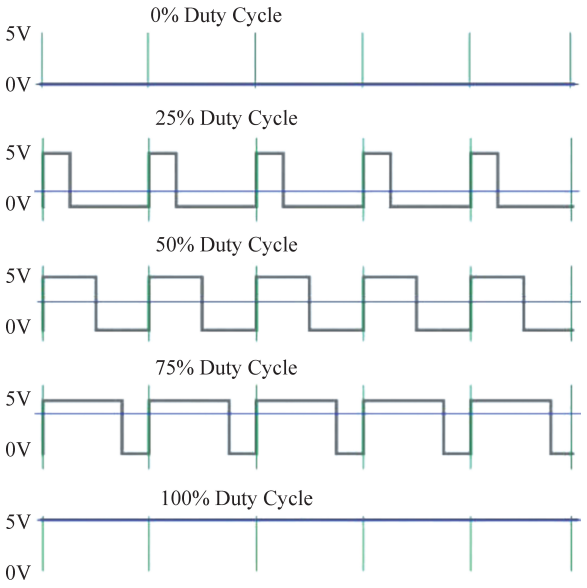



图 3.15 占空比示意图

对于 LED 的光度调节，有一个传统办法，就是串联一个可调电阻，改变电阻，灯的亮度就会改变。

还有一个办法，就是脉宽调制。该方法不用串联电阻，而是串联一个开关。假设在 1s 内，有 0.5s 的时间开关是打开的，0.5s 关闭，那么灯就亮 0.5s，灭 0.5s。这样持续下去，灯就会闪烁。如果把频率调高一点，例如 1ms(即 0.5ms 开，0.5ms 灭)，那么灯的闪烁频率就很高。当闪烁频率超过一定值时，人眼就会感觉不到，所以这时看不到灯的闪烁，只看到灯的亮度只有原来的一半。同理，如果 1ms 内(即 0.1ms 开，0.9ms 灭)，那么灯的亮度就只有原来的1/10。这就是 PWM 的基本原理。

在 GPIO.PWM(channel，frequency) 中，参数 channel 是 GPIO，参数 frequency 是频率。代码 GPIO.PWM(pins['pin_R']，2000)和 GPIO.PWM(pins['pin_G']，2000)，就是给 GPIO.0 和 GPIO.1(物理引脚 11 和 12)设置 2kHz 的频率。

在函数 setColor(col)中，ChangeDutyCycle(dc) 的作用就是改变占空比(0.0<=dc<=100.0)，确定“开启”时间与常规时间的比例。

(7) setColor(col)函数。在 loop() 中，执行循环 for col in colors。对于程序开始时的代码 colors = [0xFF0000，0x00FF00，0x0FF000，0xF00F00] 中的每个颜色，在 setColor(col) 函数中执行(col & 0xFF0000)>>16 和(col & 0x00FF00)>>8。

RGB 颜色是由红(Red)、绿(Green)、蓝(Blue)三原色组成的，所以可以使用这 3 个颜色的组合来代表一种具体的颜色，其中 R、G、B 的每个数值都为 0～255 的整数。

在表达颜色的时候，既可以使用 3 个十进制数字来表达，也可以使用 0X00RRGGBB 格式的十六进制来表达。下面是常见颜色的表达形式：红色(255，0，0)或 0x00FF0000、绿色(0，255，0)或 0x0000FF00、蓝色(255，255，255)或 0x00FFFFFF。

语句 (col & 0x00ff0000)>>16 的含义是，首先将颜色值与十六进制表示的 00ff0000 进行“与”运算，运算结果除了表示红色的数字值之外，GGBB 部分颜色都为 0。再将结果向右移位 16 位，得到的就是红色的值。所以这句代码主要用来从一个颜色中抽取其组成色(红色)的值。

同样也可以通过代码(color & 0x0000ff00)>>8 得到绿色的值，代码(col & 0x000000ff)>>0 得到蓝色的值。

由于双色 LED 只有两种颜色(红和绿)，所以程序在得到 R_val 和 G_val 的值后，通过 map() 函数得到 0～100 的占空比。最后执行 ChangeDutyCycle，实现 LED 从红色到绿色，再到混合色的效果。

### 3.1.3 RGB LED

RGB LED 可以发出各种颜色的光，红色、绿色和蓝色的 3 个 LED 被封装在透明或半透明的塑料外壳中，并带有 4 个引脚，如图 3.16 所示。

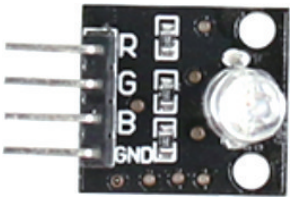


图 3.16 RGB LED

红色、绿色和蓝色三原色可以按照亮度混合并组合各种颜色，可以通过控制电路使 RGB LED 发出彩色光。原理如图 3.17 所示。

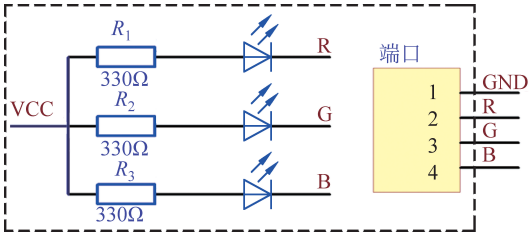



图 3.17 RGB LED 的原理

### 1. 电路连接

树莓派的功能名、BCM 编码、物理引脚与 RGB LED 引脚之间的关系如表 3.3 所示，T 型转接板与 RGB LED 模块之间的连线如图 3.18 所示，实物如图 3.19 所示。

表 3.3 树莓派与 RGB LED 引脚之间的关系

| 功 能 名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | RGB LED 模块的引脚 |
|---|---|---|---|
| GPIO.0 | 17 | 11 | R |
| GPIO.1 | 18 | 12 | G |
| GPIO.2 | 27 | 13 | B |
| GND | GND | GND | GND |

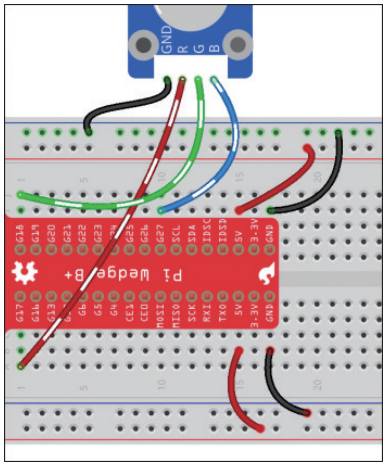



图 3.18 RGB LED 的连线

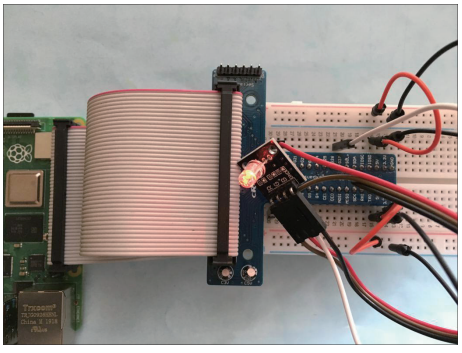


图 3.19 RGB LED 的实物

### 2. 软件编写

在 book 目录下新建文件 ch3.1.3.py，代码如下：

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time
```

```
colors = [0xFF0000, 0x00FF00, 0x0000FF, 0xFFFF00, 0xFF00FF, 0x00FFFF]
R = 11
G = 12
B = 13

def setup(Rpin, Gpin, Bpin):
    global pins
    global p_R, p_G, p_B
    pins = {'pin_R': Rpin, 'pin_G': Gpin, 'pin_B': Bpin}
    GPIO.setmode(GPIO.BOARD)
    for i in pins:
        GPIO.setup(pins[i], GPIO.OUT)
        GPIO.output(pins[i], GPIO.HIGH)

    p_R = GPIO.PWM(pins['pin_R'], 2000)
    p_G = GPIO.PWM(pins['pin_G'], 1999)
    p_B = GPIO.PWM(pins['pin_B'], 5000)

    p_R.start(100)
    p_G.start(100)
    p_B.start(100)

def map(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def off():
    for i in pins:
        GPIO.output(pins[i], GPIO.HIGH)

def setColor(col):
    R_val = (col & 0xff0000) >> 16
    G_val = (col & 0x00ff00) >> 8
    B_val = (col & 0x0000ff) >> 0

    R_val = map(R_val, 0, 255, 0, 100)
    G_val = map(G_val, 0, 255, 0, 100)
    B_val = map(B_val, 0, 255, 0, 100)

    p_R.ChangeDutyCycle(100-R_val)
    p_G.ChangeDutyCycle(100-G_val)
    p_B.ChangeDutyCycle(100-B_val)

def loop():
    while True:
```

```
            for col in colors:
                setColor(col)
                time.sleep(1)

    def destroy():
        p_R.stop()
        p_G.stop()
        p_B.stop()
        off()
        GPIO.cleanup()

    if __name__ == "__main__":
        try:
            setup(R, G, B)
            loop()
        except KeyboardInterrupt:
            destroy()
```

代码与上一节大同小异。运行程序，可以看到 RGB LED 灯亮起，依次显示不同的颜色。

## 3.2 继 电 器

如图 3.20 所示，继电器是一种电控制器件，用于在输入量的变化达到规定要求时，使电气输出电路的被控量发生预定的阶跃变化。它通常应用于自动化控制电路中控制器与受控设备之间的隔离。


图 3.20 继电器

继电器的基本原理是利用了电磁效应控制机械触点达到通或断的目的。当继电器通电时，电流开始流经控制线圈，电磁铁开始通电，衔铁被吸到线圈上，将触动点拉动，从而与常开触点连接，使带负载的电路通电；当继电器断电时，在弹簧的作用下，动触头被拉到常闭触点，使带负载的电路断电。这样，继电器的接通和断开就可以控制负载电路的状态。当需要用小信号控制大电流或电压时，继电器非常有用，在电路中起着自动调节、安全保护、转换电路的作用。

继电器的原理如图 3.21 所示。

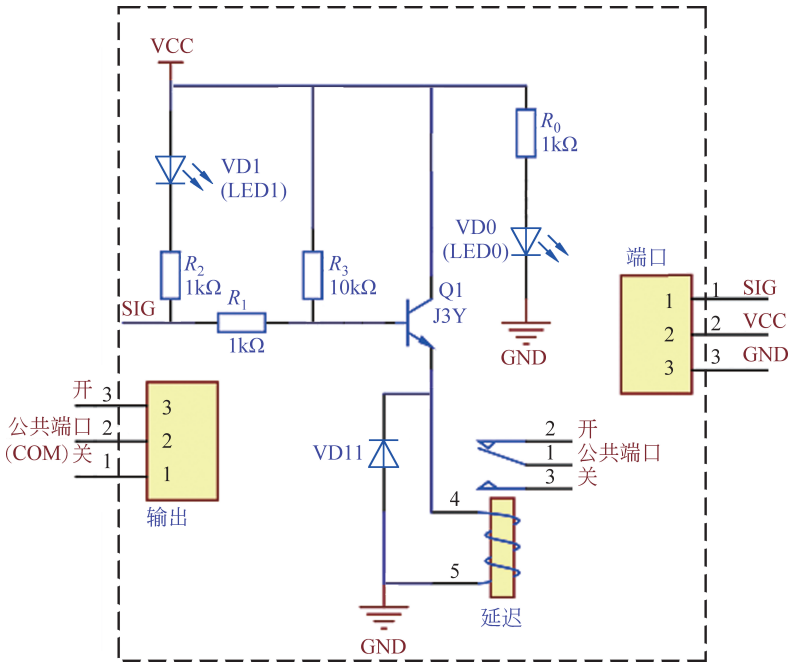



图 3.21　继电器的原理

### 1. 电路连接

树莓派的功能名、BCM 编码、物理引脚与继电器引脚之间的关系如表 3.4 所示，继电器模块与双色 LED 引脚之间的关系如表 3.5 所示。

表 3.4　树莓派与继电器引脚之间的关系

| 功　能　名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | 继电器模块的引脚 |
|---|---|---|---|
| GPIO.0 | 17 | 11 | SIG |
| 5V | 5V | 2 | VCC |
| 5V | 5V | 4 | COM |
| GND | GND | GND | GND |

表 3.5　继电器与双色 LED 引脚之间的关系

| 继电器模块的触点 | BCM 编码(T 型转接板) | 双色 LED 模块的引脚 |
|---|---|---|
| 常开 | | R |
| | GND | GND |
| 常闭 | | G |

T 型板、继电器、双色 LED 的连线如图 3.22 所示，实物如图 3.23 所示。

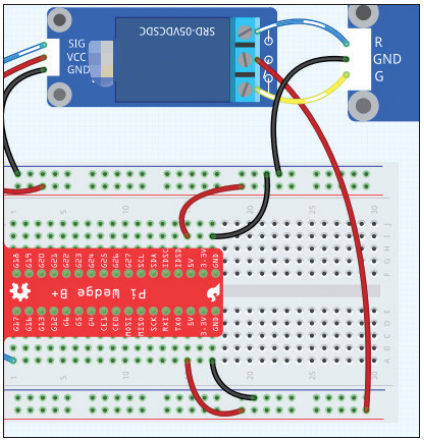



图 3.22 T 型板、继电器、双色 LED 的连线

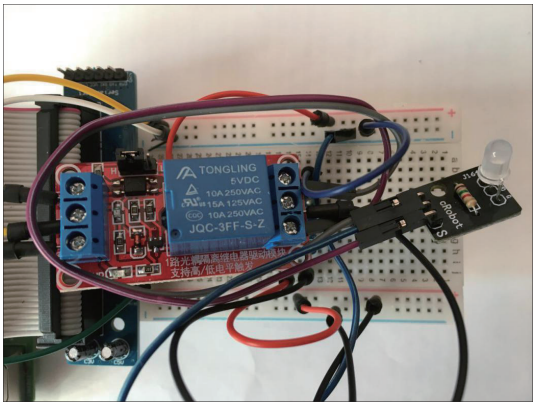



图 3.23 T 型板、继电器、双色 LED 的实物

## 2. 软件编写

在 book 目录下新建文件 ch3.2.py，代码如下：

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time

RelayPin = 11

def setup():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(RelayPin, GPIO.OUT)
    GPIO.output(RelayPin, GPIO.HIGH)

def loop():
    while True:
        print ('...relayd on')
        GPIO.output(RelayPin, GPIO.LOW)
        time.sleep(0.5)
        print ('relay off...')
        GPIO.output(RelayPin, GPIO.HIGH)
        time.sleep(0.5)

def destroy():
    GPIO.output(RelayPin, GPIO.HIGH)
    GPIO.cleanup()
```

```
if __name__ == '__main__':
    setup()
    try:
        loop()
    except KeyboardInterrupt:
        destroy()
```

运行程序，可以听到常闭触点打开、常开触点闭合时发出的嘀嗒声，同时看到 LED 双色灯的变化，以及如图 3.24 所示的 Shell 输出内容（直到按 Ctrl+C 键结束运行）。


图 3.24　Shell 显示效果

## 3.3　激光发射模块

激光是 20 世纪 60 年代的新光源，具有方向性好、亮度高、单色性好和高能量密度等特点。以激光器为基础的激光工业在全球发展迅猛，现在已广泛应用于工业生产、通信、信息处理、医疗卫生、军事、文化教育以及科研等方面。

激光发射模块如图 3.25 所示。它是一种可以发射激光的模块，原理如图 3.26 所示。

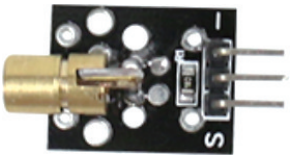


图 3.25　激光发射模块

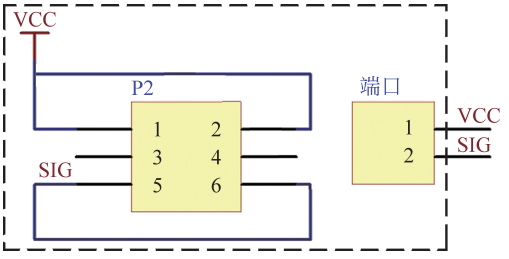



图 3.26　激光发射模块的原理

### 1. 电路连接

树莓派的功能名、BCM 编码、物理引脚与激光发射引脚之间的关系如表 3.6 所示。

表 3.6 树莓派与激光发射模块引脚之间的关系

| 功 能 名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | 激光发射模块的引脚 |
|---|---|---|---|
| GPIO.0 | 17 | 11 | SIG |
| 5V | 5V | | VCC |

激光发射模块的连线如图 3.27 所示,实物如图 3.28 所示,中间引脚不用连接。

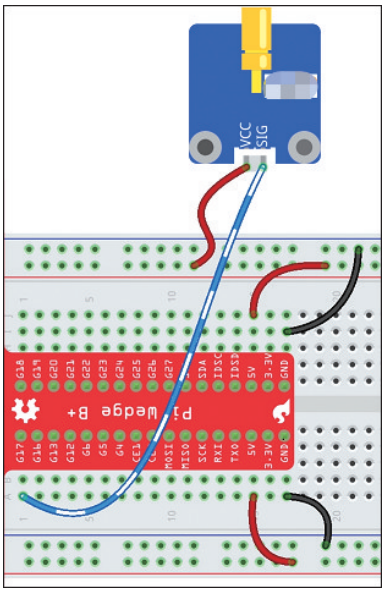



图 3.27 激光发射模块的连线

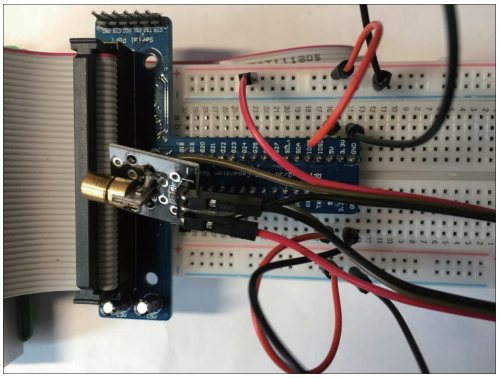


图 3.28 激光发射模块的实物

## 2. 软件编写

在 book 目录下新建文件 ch3.3.py,代码如下:

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time

LedPin = 11

def setup():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(LedPin, GPIO.OUT)
    GPIO.output(LedPin, GPIO.HIGH)

def loop():
    while True:
        print ('...Laser on')
```

```
            GPIO.output(LedPin, GPIO.LOW)
            time.sleep(0.5)
            print ('Laser off...')
            GPIO.output(LedPin, GPIO.HIGH)
            time.sleep(0.5)

    def destroy():
        GPIO.output(LedPin, GPIO.HIGH)
        GPIO.cleanup()

    if __name__ == '__main__':
        setup()
        try:
            loop()
        except KeyboardInterrupt:
            destroy()
```

运行程序，可以看到激光的发射，直到按 Ctrl+C 键结束运行。

注意，千万不要将激光对着眼睛照射！

## 3.4 开　　关

本节介绍轻触开关、倾斜开关、震动开关、干簧管和触摸开关的使用。

### 3.4.1 轻触开关

按钮模块是使用最为频繁的电子部件，内部由一对轻触拨盘组成，按下时闭合导通，松开时自动弹开断开，如图 3.29 所示。

使用常开按钮作为树莓派的输入设备，按下按钮时，GPIO 将变为低电平（0V）。通过编程检测 GPIO 的状态，如果 GPIO 变为低电平，则表示按按钮。原理如图 3.30 所示。

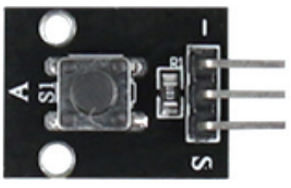



图 3.29　轻触开关

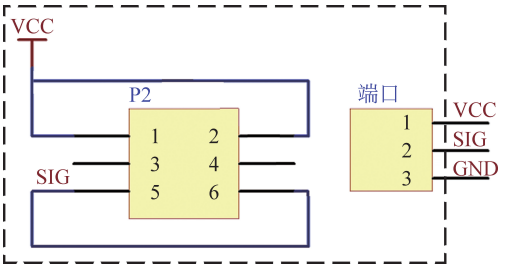



图 3.30　轻触开关的原理

#### 1. 电路连接

树莓派的功能名、BCM 编码、物理引脚与轻触开关引脚之间的关系如表 3.7 所示，与双色 LED 引脚之间的关系如表 3.8 所示。

表 3.7 树莓派与轻触开关引脚之间的关系

| 功 能 名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | 轻触开关模块的引脚 |
|---|---|---|---|
| GPIO.0 | 17 | 11 | SIG |
| 5V | 5V | 5V | VCC |
| GND | GND | GND | GND |

表 3.8 树莓派与双色 LED 引脚之间的关系

| 功 能 名 | BCM 编码(T 型转接板) | 物理引脚(BOARD 编码) | 双色 LED 模块的引脚 |
|---|---|---|---|
| GPIO.1 | 18 | 12 | R |
| GPIO.2 | 27 | 13 | G |
| GND | GND | GND | GND |

轻触开关的连线如图 3.31 所示,实物如图 3.32 所示。

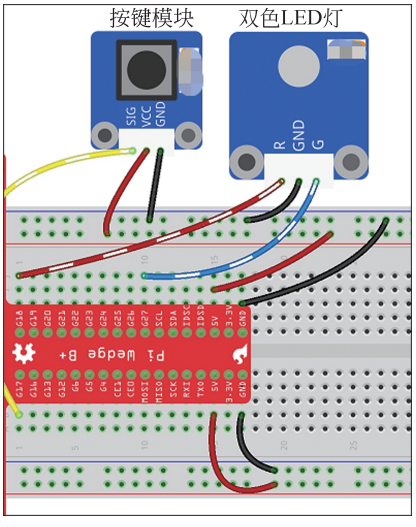



图 3.31 轻触开关的连线

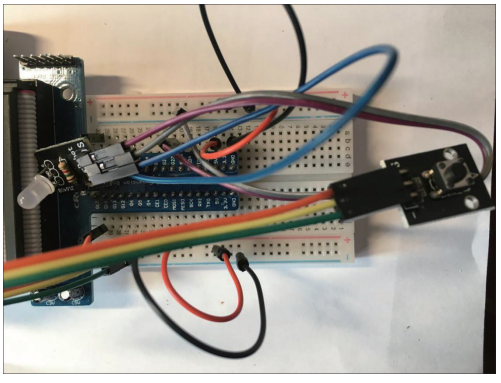


图 3.32 轻触开关的实物

## 2. 软件编写

在 book 目录下新建文件 ch3.4.1.py,代码如下:

```
#!/usr/bin/env python
import RPi.GPIO as GPIO

BtnPin = 11
Gpin = 12
Rpin = 13
```

```
def setup():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(Gpin, GPIO.OUT)
    GPIO.setup(Rpin, GPIO.OUT)
    GPIO.setup(BtnPin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
    GPIO.add_event_detect(BtnPin, GPIO.BOTH, callback=detect, bouncetime=200)

def Led(x):
    if x == 0:
        GPIO.output(Rpin, 1)
        GPIO.output(Gpin, 0)
    if x == 1:
        GPIO.output(Rpin, 0)
        GPIO.output(Gpin, 1)

def Print(x):
    if x == 0:
        print ('    **********************')
        print ('    *    Button Pressed!    * ')
        print ('    **********************')

def detect(chn):
    Led(GPIO.input(BtnPin))
    Print(GPIO.input(BtnPin))

def loop():
    while True:
        pass

def destroy():
    GPIO.output(Gpin, GPIO.HIGH)
    GPIO.output(Rpin, GPIO.HIGH)
    GPIO.cleanup()

if __name__ == '__main__':
    setup()
    try:
        loop()
    except KeyboardInterrupt:
        destroy()
```