

## 6. 直流电机和PWM

### 实验目的

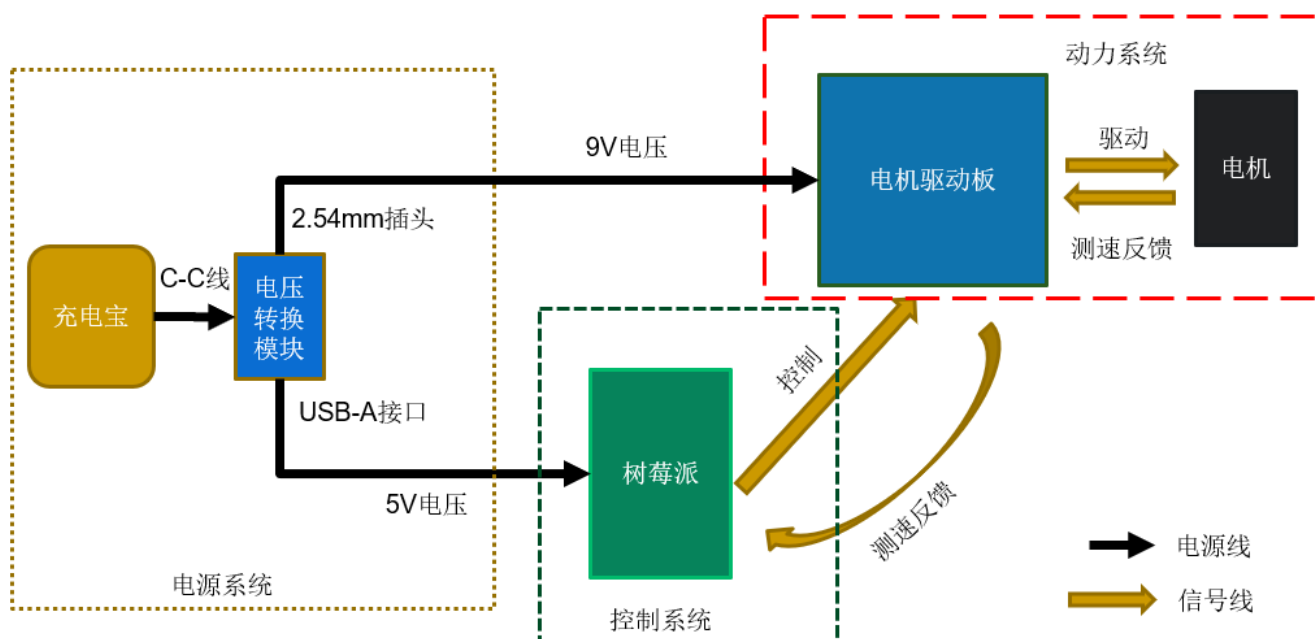
- 了解直流电机的控制方法
- 了解PWM的基本概念
- 掌握树莓派PWM的编程方法
- 掌握通过PWM来控制直流电机

### 准备工作——接线步骤

#### 操作

1. 用一根C-C充电线将电压转换模块和充电宝连接，提供9V电压给电机驱动。
2. 将电压转换模块的EH2.54插座和电机驱动模块的EH2.54插头相连，注意正负极。
3. 将ENA, IN2, IN1, ENB, IN4, IN3和树莓派的GPIO相连，具体取决于需求。
4. 电压转换模块的USB-A口连接A-C线给树莓派提供5V供电。
5. 如果线太长，小车要在地面上跑，可以想办法收一下线或固定一下。

#### 系统结构



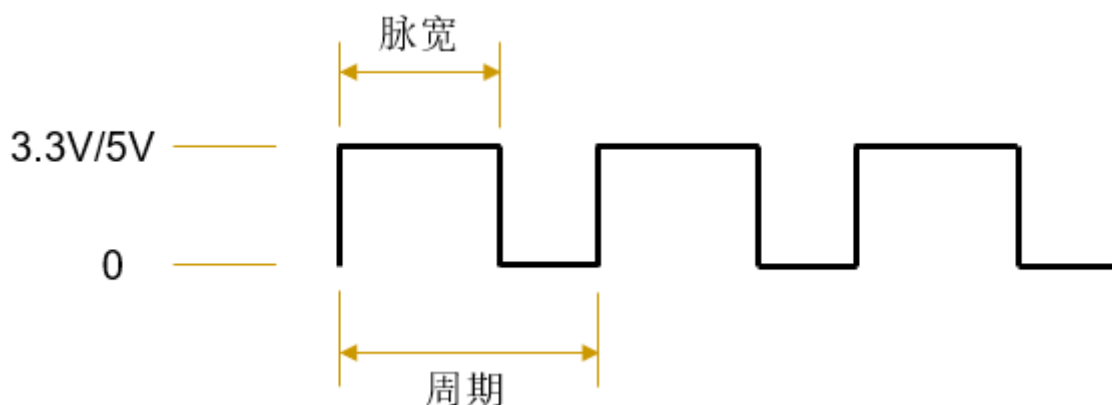
#### reference

# 直流电机

- 能将直流电能转换成机械能的旋转电机。
- 通常搭配减速齿轮（齿轮箱）使用
- 组成
  - 定子：产生磁场，永磁体或电磁铁
  - 转子：线圈，通电产生电磁扭矩
- 简单地说
  - 有两个输入端，不分正负，有电压差就会转
  - 把输入电压的正负对换，就会反着转
  - 电压差恒定，转速恒定，电压增大，转速增大
  - 如果给它加上间歇性通断的电压，电机就会“加速-减速-加速-减速-加速……”
  - 如果这个电压“通断”足够快，就会使电机转速较稳定地维持在某一数值。这个“数值”取决于输入电压的平均值。

## PWM

PWM(Pulse Width Modulation)，中文译为脉冲宽度调制，是利用数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用于从测量、通信到功率控制与变换的许多领域中。



上图是PWM波的波形示意图。在实际应用中，PWM波的占空比是PWM的主要特性。

- 占空比 = 脉宽 / 周期
- 调节占空比：可以固定脉宽，改变周期；也可以固定周期，改变脉宽。我们通常采用后者。
- 占空比越大，从整个周期来看，平均电压越高；占空比越小则平均电压越低。

## 用PWM给直流电机调速

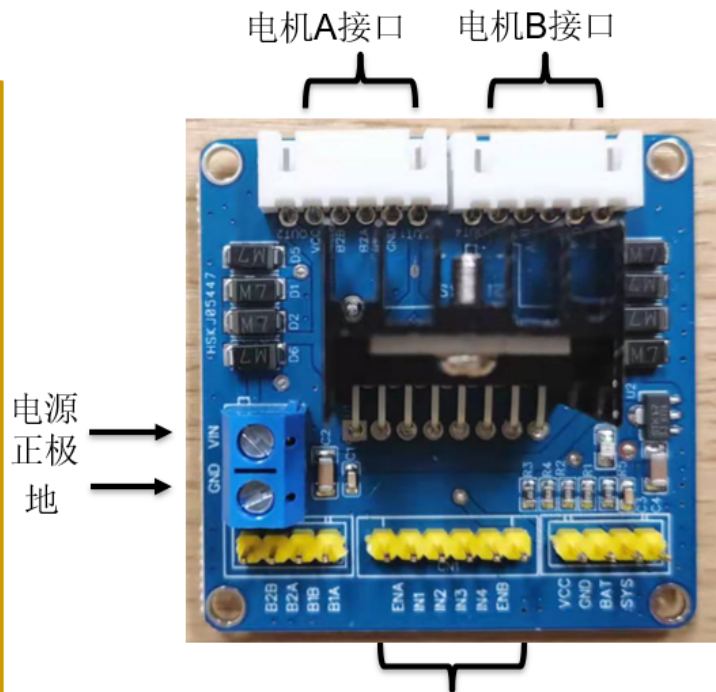
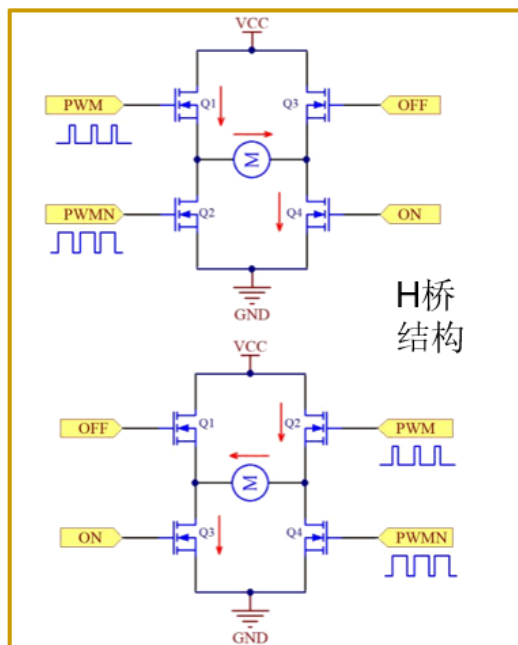
### 理论

- 常用PWM波作为直流电机的输入，可以通过改变PWM波的占空比方便地给直流电机调速。

- 但是，一般小型处理器的引脚不能直接驱动直流电机。因为直流电机将电能转换为机械能，需要比较大的电流提供电能，我们的小车上的直流电机所需的电流可能是安培级的，而树莓派的GPIO只能提供毫安级的电流。
- 所以需要有个驱动电路，它有输出大电流的能力，能接收树莓派的控制信号，并按照树莓派的意思让电机转或不转。

## 硬件（电机驱动板）

- 这块驱动板的核心是L298N芯片，该芯片包含了两个H桥结构
- 下方中间的6个引脚，分别控制两边的电机： ENA, IN1, IN2, IN3, IN4, ENB



从左至右为：  
ENA, IN1, IN2, IN3, IN4, ENB

注意事项：

- 树莓派和驱动板的逻辑部分需要共地即GND连在一起**，否则树莓派的逻辑就无法被驱动板识别。
- 如果树莓派未使用锂电池供电：树莓派的GND需要和电池负极一同接到GND端以共地
- 如果树莓派也使用锂电池供电，则本身就共地，GND端接入电池负极即可（**建议使用电池供电**）

## 控制方案

下面是一组电机接口控制信号的真值表。

ENA	IN1	IN2	直流电机状态
0	X	X	停止
1	0	0	制动
1	0	1	正转
1	1	0	反转
1	1	1	制动

树莓派控制驱动板通常有两种方法。

- 方法一：IN1和IN2端输入固定电平，ENA端输入PWM波。当PWM在高电平时，电机加速；PWM在低电平时，电机停止。
- 方法二：ENA端和IN1（或IN2）端输入固定电平，IN2（或IN1）端输入PWM波。  
**严防电机堵转！！会烧坏电机！！**

**那么，用多少频率的PWM比较合适？**

- 如果频率太低，直流电机将会产生明显的抖动。
- 如果频率太高，直流电机很可能会转不起来，因为电机转子的角速度的建立需要一定时间。
- 一般人的耳朵能听到的频率范围在几十Hz到近20kHz之间，如果频率在这个范围内，会听到比较明显的啸叫。
- 所以选择一个合适的频率是一个权衡的过程。

经过实验，对于我们小车上的电机，用50Hz到一两百Hz的频率已经不会产生明显的抖动，而且频率不算高，即便是软件PWM也能胜任。

## 如何用树莓派产生PWM波

- 对于微处理器，一般可以分为软件PWM和硬件PWM。
  - 软件PWM：先在目标GPIO上输出一个电平（高电平或低电平），持续一段时间，然后把电平取反，再持续一段时间，再取反，循环往复。这种方式的精度一般较低，受到定时器精度、操作系统调度等影响，并且一般依赖于CPU中断，会占用少量CPU资源。
  - 硬件PWM：有些CPU自带PWM硬件，只要给出期望的频率和占空比，这些硬件就会独立产生PWM波，依赖于内部的硬件定时电路，精度通常较高，而且不需要占用额外的CPU资源。
- 在接下来的例子里，我们将会使用RPi.GPIO模块的PWM类产生PWM波，这个模块产生的PWM全部是软件PWM。

# 直接控制GPIO

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# 实现功能：让GPIO21输出频率为40Hz，占空比为40%的PWM波
# 采用定时-翻转的方法实现。
import RPi.GPIO as GPIO
import time

# 参数设置
GPIO.setmode(GPIO.BCM)
PWM = 21
FREQUENCY = 40
DUTY = 0.4
GPIO.setup(PWM, GPIO.OUT) #GPIO21输出PWM波

# don't use too large freq
# 计算高电平和低电平的持续时间，单位是毫秒
def calc_delay_period(freq, duty):
    t = 1.0/freq
    ph = t*duty
    pl = t - ph
    return ph, pl

period_h, period_l = calc_delay_period(FREQUENCY, DUTY)

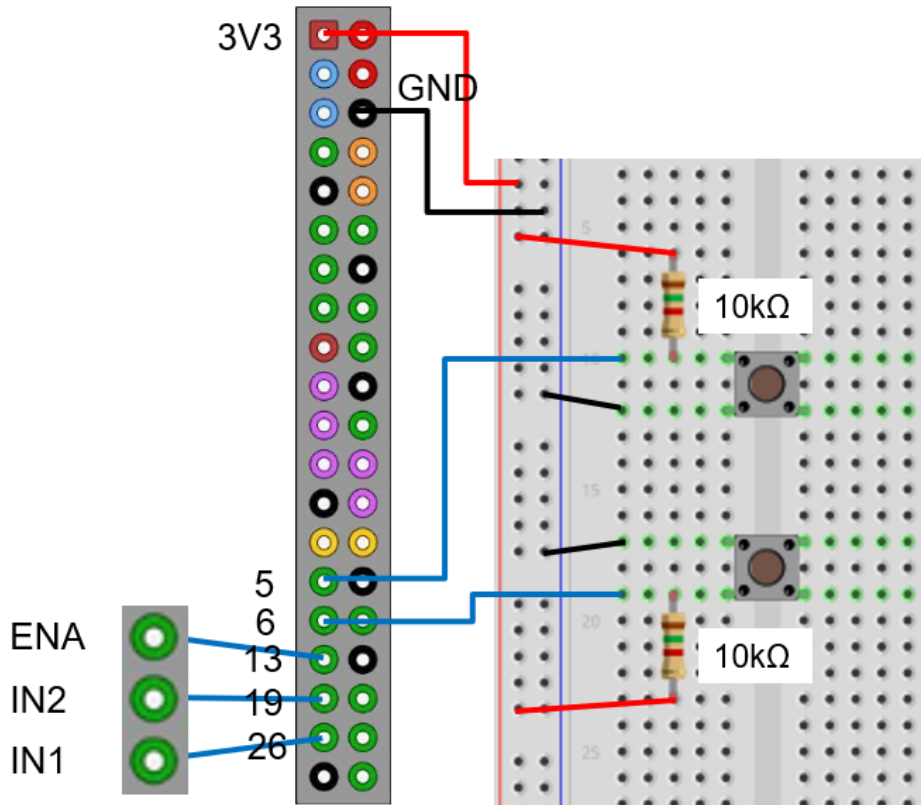
try:
    while True:
        GPIO.output(PWM, GPIO.HIGH)
        time.sleep(period_h)
        GPIO.output(PWM, GPIO.LOW)
        time.sleep(period_l)
except KeyboardInterrupt:
    pass
GPIO.cleanup()
```

这个例子有助于理解PWM波的产生原理，但是实际应用中肯定不能这样。因为这样需要持续占用CPU资源，而且time.sleep()的精度很低，所以频率和占空比的取值都受到很大的限制。

## 引入按键控制

利用RPi.GPIO库可以很方便地产生PWM波。现在，我们尝试用PWM波控制小车的电机，并使用两个按键改变占空比，以达到调速的目的，按上面的按键可以使转速档位增大，按下面的按键则使转速档位减小。

## 添加连线



## 使用代码

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# 按键控制电机转速
import RPi.GPIO as GPIO
import time
# 设置按键以及输出引脚，BTN1和BTN2分别为减速和加速的按键
EA, I2, I1 = (13, 19, 26)
BTN1, BTN2 = (6, 5)
FREQUENCY = 50

# DUTYS是查找表，表示不同档位时占空比的值（0~100）；
DUTYS = (0, 20, 40, 60, 80, 100)
# duty_level是一个变量，表示当前占空比的档位，此处初始化为最大档位，即占空比100%
duty_level = len(DUTYS) - 1

# 设置各GPIO的输入输出模式和初始值。
# 其中I1和I2控制了电机的转向，如果发现实际情况与预期不符，只要在代码中或硬件连线上把I1和I2
# 的值互换即可。
GPIO.setmode(GPIO.BCM)
GPIO.setup([EA, I2, I1], GPIO.OUT)
```

```

GPIO.output([EA, I2], GPIO.LOW)
GPIO.output(I1, GPIO.HIGH)
GPIO.setup([BTN1, BTN2], GPIO.IN,
           pull_up_down = GPIO.PUD_UP)

# 使用GPIO库的pwm功能，创建一个PWM类的实例pwm
# 创建时需要指定两个参数：第一个参数指定输出引脚，第二个参数指定PWM波的频率。
pwm = GPIO.PWM(EA, FREQUENCY)

# 执行下面的代码之后，相应的引脚开始持续产生PWM输出。需要指定一个参数：占空比的值。范围是
0~100。
pwm.start(DUTYS[duty_level])
print("duty = %d" % DUTYS[duty_level])

# 检查按键是否被按下
def btn_pressed(btn):
    return GPIO.input(btn) == GPIO.LOW

# update_duty_level(delta)的delta为+1或者-1，表示占空比的档位增加或减少一档
# 据此算出新的占空比档位，然后调用ChangeDutyCycle(duty)方法更新占空比。参数duty同样是
0~100之间的数。
def update_duty_level(delta):
    global duty_level
    old = duty_level
    duty_level = (duty_level + delta) % len(DUTYS)
    pwm.ChangeDutyCycle(DUTYS[duty_level])
    print("duty: %d → %d" % (DUTYS[old], DUTYS[duty_level]))

btn1_released = True
btn2_released = True
try: # 保证程序在按下Ctrl+C之后能够正常退出
    # 无限的循环，用以检测按键是否按下，并进行相应的操作
    while True:
        if btn1_released:
            if btn_pressed(BTN1):
                time.sleep(0.01) # 按键消抖
                if btn_pressed(BTN1):
                    update_duty_level(-1)
                    btn1_released = False
            else:
                if not btn_pressed(BTN1):
                    btn1_released = True
        if btn2_released:
            if btn_pressed(BTN2):

```

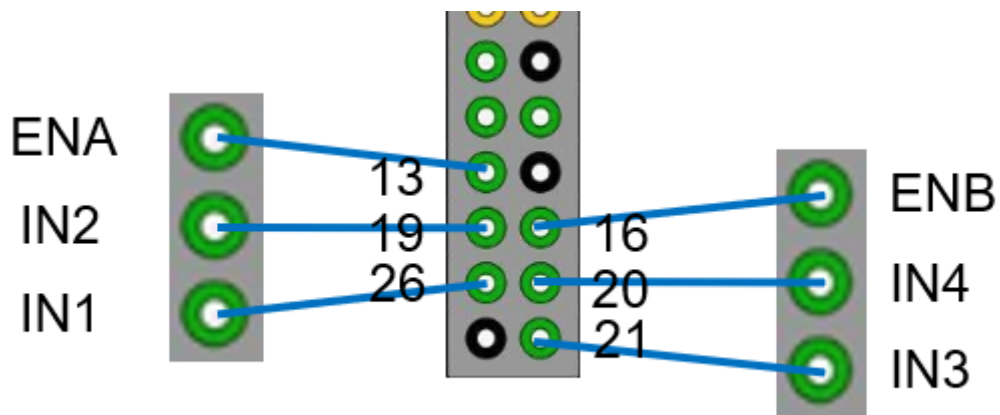
```

        time.sleep(0.01)
        if btn_pressed(BTN2):
            update_duty_level(1)
            btn2_released = False
        else:
            if not btn_pressed(BTN2):
                btn2_released = True
except KeyboardInterrupt:
    pass
# 在程序结束前，不仅要调用GPIO.cleanup()，还要调用pwm.stop()来停止PWM输出
# 否则程序结束后，这个PWM会持续输出，造成不必要的能耗以及以外损坏的风险。
pwm.stop()
GPIO.cleanup()

```

## 用键盘控制

### 修改连线



用到的GPIO口为：13, 19, 26; 16, 20, 21

### 使用代码

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import RPi.GPIO as GPIO
# GPIO引脚设置，可以在这里反转引脚设置来让电机反转
EA, I2, I1, EB, I4, I3 = (13, 19, 26, 16, 20, 21)
# 常量设置
FREQUENCY = 50
DUTYS_A = {'w':20, 'a':0, 's':0, 'd':20}
DUTYS_B = {'w':20, 'a':20, 's':0, 'd':0 }
EXPRESSIONS = {'w':'move forward!',
               'a':'turn left!',

```



```

        's':'stop!',
        'd':'turn right!'}

# setup
GPIO.setmode(GPIO.BCM)
GPIO.setup([EA, I2, I1, EB, I4, I3], GPIO.OUT)
GPIO.output([EA, I2, EB, I3], GPIO.LOW)
GPIO.output([I1, I4], GPIO.HIGH)
# 左右轮PWM设置
pwma = GPIO.PWM(EA, FREQUENCY)
pwmb = GPIO.PWM(EB, FREQUENCY)
# 初始化为不动
pwma.start(DUTYS_A['s'])
pwmb.start(DUTYS_B['s'])
print("ready!")

while True:
    cmd = input("command >> ")
    if cmd == 'q':
        pwma.stop()
        pwmb.stop()
        GPIO.cleanup()
        break
    elif (cmd=='w') or (cmd=='a') or (cmd=='s') or (cmd=='d'):
        pwma.ChangeDutyCycle(DUTYS_A[cmd])
        pwmb.ChangeDutyCycle(DUTYS_B[cmd])
        print(EXPRESSIONS[cmd])
    else:
        pass


```

## 更精确的PWM波——硬件PWM

下图是使用RPi.GPIO.PWM产生PWM波的期望值和实际值（来自Raspberry Pi Cookbook）。可以看到这个模块产生的PWM的稳定性其实很差。实际测量结果和这个表格基本相符。

Requested frequency	Measured frequency
50 Hz	50 Hz
100 Hz	98.7 Hz
200 Hz	195 Hz
500 Hz	470 Hz
1 kHz	890 Hz
10 kHz	4.4 kHz

树莓派上还有其他的库可以产生性能更好的PWM波，比如下面要介绍的pigpio库。

- pigpio是一个用C语言编写的高效的树莓派GPIO库，适用于所有版本的树莓派，有python2和python3的接口，并且已经预装在Raspbian上了
- pigpio基于底层硬件，因此控制精度高，速度快。
- 可以在GPIO0~31上独立产生基于硬件定时器的软件PWM
- BCM2711上有两个channel的硬件PWM，用pigpio可以调用，不过树莓派4 model B只支持调用channel 0
- pigpio文档: <http://abyz.me.uk/rpi/pigpio> 

## pigpio使用

### 确认服务开启

pigpio库包含一个名为pigpiod的daemon（常驻内存提供服务的进程），在使用pigpio的任何功能以前，必须先确保daemon处于运行状态，然后创建一个实例连接该daemon。连接成功后，这个实例就能提供pigpio的所有功能。具体地说：

- 首先启动pigpiod. 在终端输入 `sudo pigpiod`
- 如果要确保pigpiod是否已成功开启，可以用这个命令查看 `ps -A | grep pigpiod`
- 在python代码中，引入pigpio模块 `import pigpio`
- 然后，调用以下方法创建一个连接daemon的实例 pi `pi = pigpio.pi()`
- 可以检查是否连接成功：

```
if not pi.connected:
    exit()
```

### 使用代码

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
# start daemon first: sudo pigpiod
import pigpio
PWM = 21
pi = pigpio.pi()
# 在GPIO21上产生频率为8000Hz，占空比为30%的PWM波
pi.set_PWM_frequency(PWM, 8000)
pi.set_PWM_range(PWM, 100)
pi.set_PWM_dutycycle(PWM, 30)
```

然后就可以任意使用pigpio了。

- 21表示GPIO21，pigpio库强制使用Broadcom number表示GPIO，相当于RPi.GPIO.BCM。0~31号GPIO都可以用这些方法独立产生PWM波（不过我们的板子上只有2~27号GPIO）
- pi.set\_PWM\_frequency(channel, freq)设置PWM波的频率。注意它不能设置任意频率，只能在18个频率中选一个，具体由当前底层硬件的采样率决定（见后一页），底层的采样率则是由daemon启动时决定的。在启动daemon的命令后面加上-s value参数就可以把采样率设定为value(us)，采样率可选的值为 1, 2, 4, 5, 8, 10，默认值为5us。
- pi.set\_PWM\_range(channel, range)方法用来更改占空比的范围，默认范围是0~255，经过更改后变为100，即100表示满占空比。
- pi.set\_PWM\_dutycycle(channel, dutycycle)方法用来调整占空比，调用该方法后，占空比将立即被更新。同时它也是PWM波的开关，设置占空比为0就表示关闭PWM输出。

对于每种采样率，有18种可选的频率。

		Hertz								
sample rate (us)	1:	40000	20000	10000	8000	5000	4000	2500	2000	1600
		1250	1000	800	500	400	250	200	100	50
	2:	20000	10000	5000	4000	2500	2000	1250	1000	800
		625	500	400	250	200	125	100	50	25
	4:	10000	5000	2500	2000	1250	1000	625	500	400
		313	250	200	125	100	63	50	25	13
5:	8000	4000	2000	1600	1000	800	500	400	320	
	250	200	160	100	80	50	40	20	10	
	8:	5000	2500	1250	1000	625	500	313	250	200
	156	125	100	63	50	31	25	13	6	
10:	4000	2000	1000	800	500	400	250	200	160	
	125	100	80	50	40	25	20	10	5	