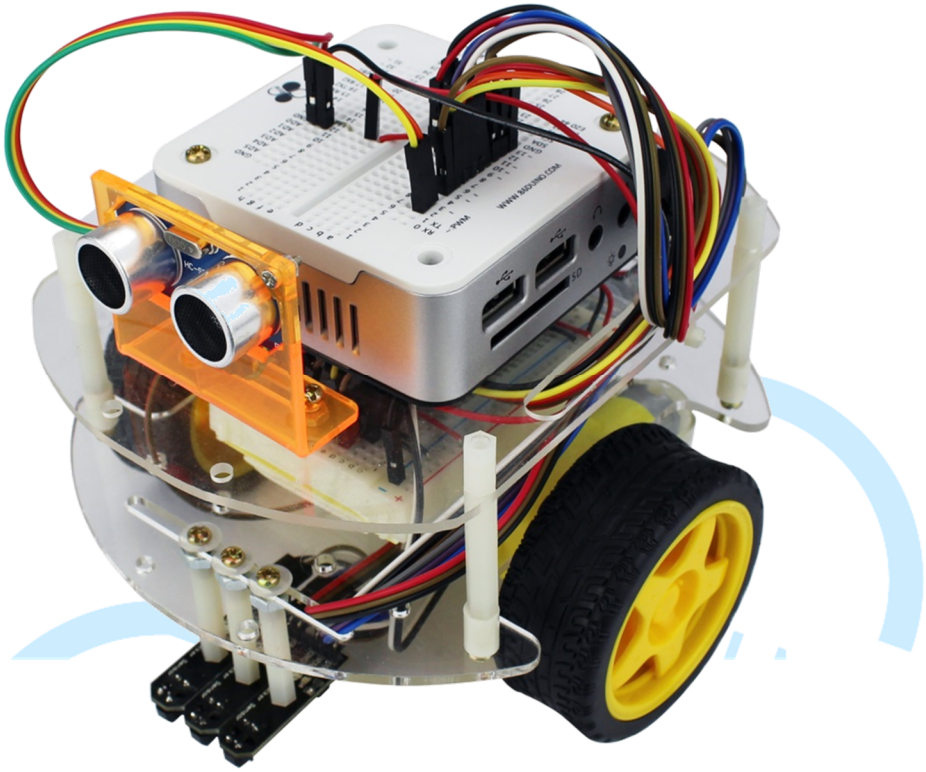
EduCake　实作自走车



一、　自走车该有哪些功能

大致介绍过 86Duino EduCake 各种讯号控制、互动的方法之后，这个章节我们就来实作一个实际的应用范例-自走车。自走车在很多地方都派得上用场，最大名鼎鼎的就是登陆火星，像普通的小轿车一样大的好奇号，至今还在火星地表为人类探险和侦查火星的各种大地和地表信息，不断的传回高解析火星地表照片，有兴趣的读者可以参阅维基网站 http://zh.wikipedia.org/wiki/%E5%A5%BD%E5%A5%87%E8%99%9F。登陆火星对你我来说是根本不太可能的事情，但眼光放到地球上，还是有不计其数的各种自走车应用在生活中，像是 GOOGLE 已经快接近实用阶段的自动驾驶车，都已经开在路上超过百万公里，也已经开始市区道路的测试动作 (http://www.ithome.com.tw/news/87223)，预计未来的五六年内就可以真正

实用到普通车上；像是公司的门面用来接待客人、送茶水的机器人目前也很多单位在实作和使用；像是各种工厂内，需要搬运大型零组件的时候，都会使用自动搬运车来帮忙精准快速的把货物搬到指定位置，提高生产效率；像是 Amazon 这种大型量贩中心，为了快速组合出客人的订单，都会有很多的自动运送自走车再帮忙在仓库内四处奔走收集货品；而生活中，最常见的自走车就是扫地机了，一般人若是要求不高，一千多新台币就可以买一台来家里帮忙打扫，而扫地机也有很高级一台三五万的版本，拥有扫地干净、路径规划精准、自动回归充电、定时清扫、不碰撞障碍回避等等的功能。

这些不同种类的车都由各种不同动力、传感器组成主要功能，底下就依照功能分类来谈谈这些功能如何使用 86Duino EduCake 来实作一台简单的自动搬运自走车。

二、 动力和运动控制

车子当然是要轮胎能动，才能带动车子前进，动力部分，这只是一台示范性的车，不需要载重和特殊的功能，使用简单的减速齿轮箱和普通的 DC 马达构和 65mm 塑料轮台构成的普通组合就行了，如附图 1。这种马达使用电压限制为 0~6V，0V 时车子完全不会动，慢慢的增加电压，约是到 1V 时马达就会开始慢慢的旋转，在继续增加电压，马达就会慢慢的越转越快，且因为需要的电流约是 100~500mA，两颗最大约 1A，这使用普通的 L293D 双马达驱动 IC 就可以轻易控制了。

图 1. 小型车用的减速马达和轮胎

控制马达转动的电路如图 2。因为需要有转速的控制，所以每颗马达共是两条方向讯号线和一条 PWM 速度控制讯号线，两颗马达共需要六条线才能正确使用。图中的 1.2.3 号线控制第一颗马达，4.5.6 号线控制第二颗马达，这六条线就是找 EduCake 上面尚未被其他装置使用到的 digital 脚位来安装。控制每颗马达的转速可使用 **analogWrite(EN,速度);** 这个指令，这部分因为前面在谈 PWM 的章节已经有说明过了，这里就跳过不再赘述。注意马达的电压是给最高 6V，所以电路图画 4 颗 1.5V 的电池串连当示意，实际使用，是用变压器和 DC converter 来降压到 6V 供给他使用；或是使用行动电源的话，那就是 5V 要升压到 6V 给他。另外这个极限电压没有一定要 6V，可以稍微高一点，笔者测试到 7V 都还可以使用，但在更高就会严重影响马达寿命了，尽量是不要。

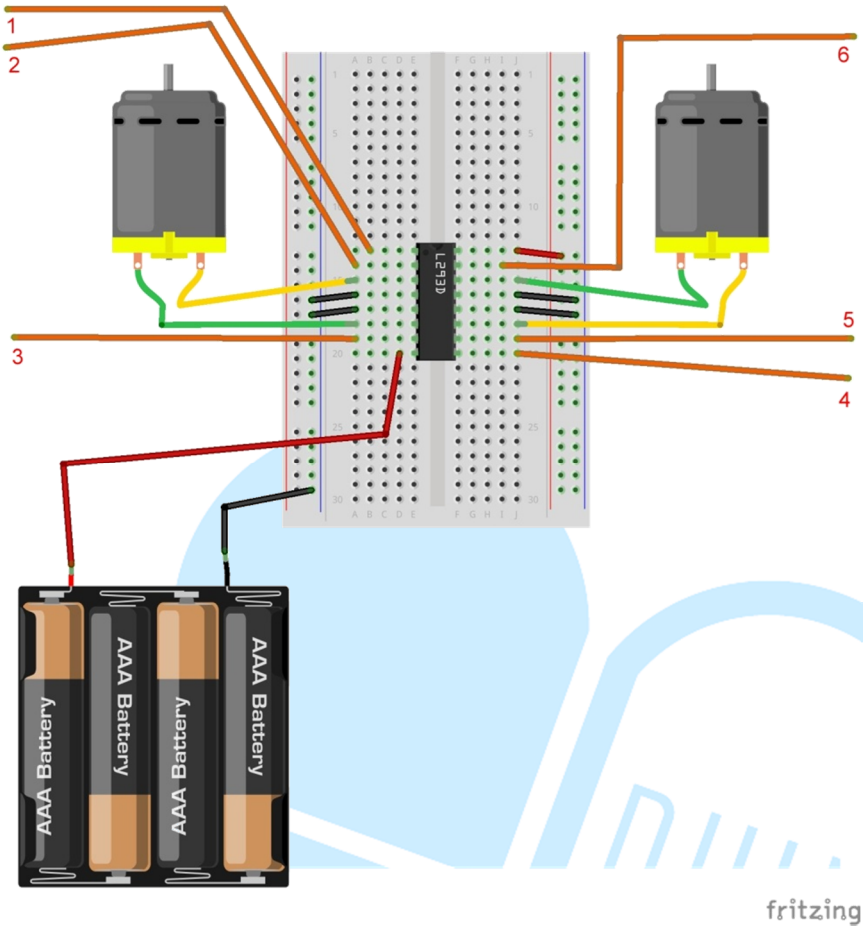


图 2. 马达控制电路

控制程序会长的像这样:

```
int M1a=6,M1b=7; //    這組用來控制第一顆馬達正反轉
int ENa=5; //第一顆馬達的速度
int M2a=9,M2b=8; //    這組用來控制第 2 顆馬達
int ENb=10; //第二顆馬達的速度
void setup()
{
  pinMode(M1a,OUTPUT);
  pinMode(M1b,OUTPUT);

  pinMode(ENa,OUTPUT);
```

```
    pinMode(M2a,OUTPUT);
    pinMode(M2b,OUTPUT);
    pinMode(ENb,OUTPUT);
}
void loop()
{
  int a;
  for(a = 0; a <= 255; a+=5)
  {
    digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, a); //  使第一顆馬達由慢到快慢慢加速
    delay(20);
  }
  digitalWrite(ENa, LOW); //  停止
}
```

再来就是想要知道车子的动作状况，需要使用编码器/编码盘(Encoder)这样的装备，编码器如图 3

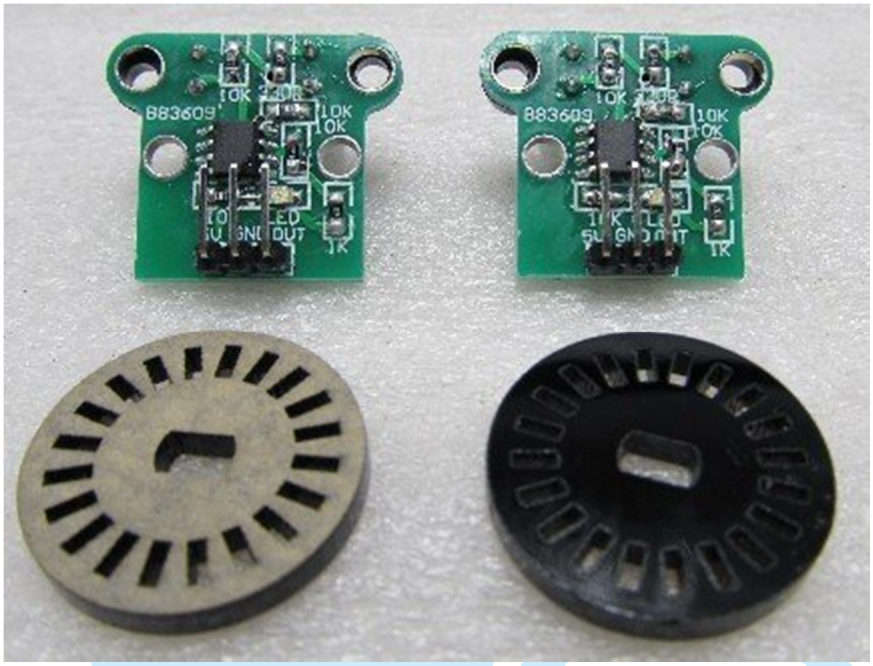


图 3. 编码器

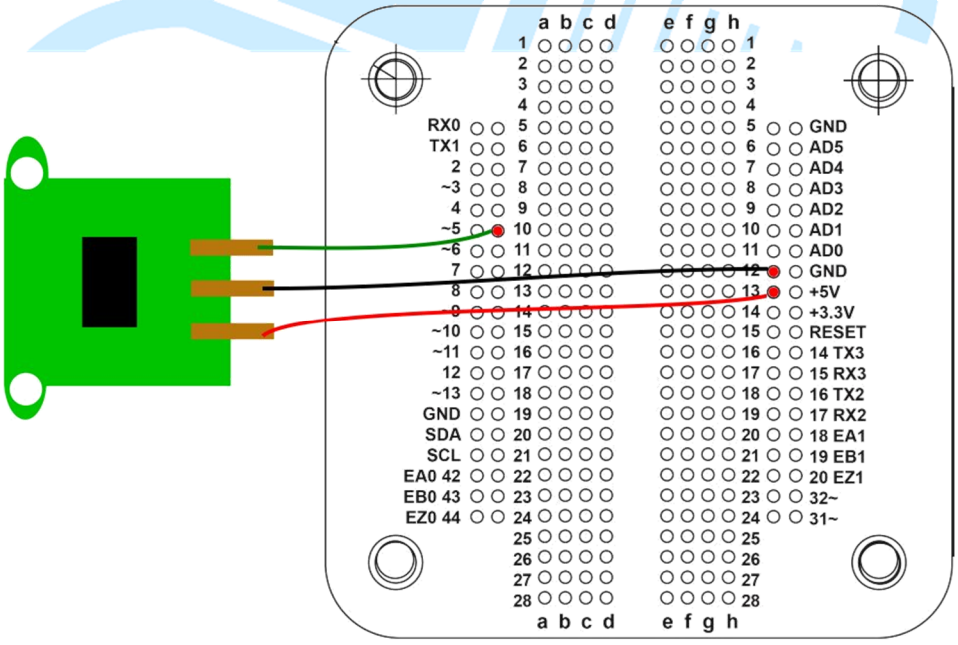


图 4. 编码器接线图

编码盘通常装在轮子轴心上面，轮子转动时，带动编码盘(画面中一格一格那个就是)一起转动，编码盘会遮住光线，但上面有洞的地方可以让光通过，利用光电传感器就可以直接计算取得到底转了几格，换算成轮子的转动距离。

实际使用的时候，因为轮子转动很快，这不能利用循环一直去作 digitalRead 读取传感器动作，除了效率差而且很不准，一直 Read 也不一定能读取到变化的位置，造成很大误差，还会害程序卡在那个位置。简单的作法应是使用 pulseIn(); 函数，因为编码盘有洞的地方会造成光电传感器传回 1(光有通过)，没有洞的地方会传回 0，这时候使用 pulseIn(pin, HIGH);可以量取到转过一格有洞的位置需要多少时间，由编码盘上面算得孔洞共有 20 格，没有孔洞的位置也是 20 格，把量到的时间乘上 40 就可以大概知道转一圈需要的时间，这就是转速了；而因为轮子的周长是已知，所以转动一格前进的距离也可以轻易的计算出来。举例来说，轮子直径是 6.5cm，则这颗轮子旋转一圈能前进的距离就是：6.5*3.14=20.41cm(当然，这里的圆周率使用越精确的数值，算出来的结果就越准，但得考虑选用的控制板的运算能力和能容纳的精确度)。编码盘被分割成有 20 孔洞，前面的数字为了计算方便概算 20cm 的话，则每一格就代表前进了 20cm/20 格=1cm。这样的结果其实是有很大的误差在的，除了圆周率的因素以外，车子可能在某些地方造成打滑、轮胎因为载重造成的变形等等都会使得量测结果和预料不一样，一般都会再搭配额外的方式来协助修正误差，像是检查点、影像、或是测距修正等等。

前面这个方法有个大问题，车子万一不动时，使用 pulseIn 函数有个问题，就是一样会让程序卡在那边直到函数的 timeout 时间，timeout 的长短也不容易设定，因为我们不晓得车子会跑快还是跑慢(除非，我们一开始就指定车子只能用某些速度跑)，这样很不理想，更好的方式是使用中断+millis()函数，可以很精确的量取到每格经过的时间，这也是前面章节谈过的，就请自行回去翻阅；另外把编码盘的挖空格数增加也是一个好方法，等于缩短每个空格的距离，像是由 20 格改成 100 格就能明显变精确；还有一个方式是改变安装位置，例如改装

到 DC 马达的后方出轴，而不是现有装到减速齿轮箱的轮轴位置，若以减速齿轮箱的减速比是 150:1 来说，轮轴位置转动一圈等于后方 DC 马达出轴已经转动 150 圈，这可以直接提高 150 倍精密度，但相对这样需要的处理速度也要更快，若使用中断的解法会造成中断太频繁。中断的触发也是要一点 CPU 时间的，很频繁的中断触发等于会消耗大量的 CPU 时间，万一 EduCake 还需要有别的运算要作，这样会影响效能，如何取舍还得实验看看。中断的程序代码如下:

```
volatile int counter ;
unsigned long oldtime;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(2, RPM, RISING);//使用訊號由 LOW 變 HIGH 的時候引發中斷
  Serial.begin(9600);
  oldtime= millis(); //先把目前時間記錄下來
  counter =0;
}

void loop()
{
   long rpm;
   if (counter   >= 20) //  前面講過一圈 20 格，count=20 代表轉一圈了
{
```

```
        rpm = millis() - oldtime; // 先取得共經過多少時間了，單位千分之一秒
        rpm = 60 * 1000 / rpm; // 計算轉速，一分鐘 60 秒，一秒 1000ms
        oldtime = millis(); //重新開始計時
        counter   = 0; // 計數器順便歸零
        Serial.println(rpm, DEC);
    }
}

void RPM ()
{
 // 每次引發中斷都會來這裡，只作一件事情，把變數加一
 // 中斷裡面的事情盡量簡單，這樣才可以減輕中斷的處理負擔
 // 也不會造成外面部分的程式受到太大影響
 // 編碼器的訊號變化如圖 5
 // 引發中斷的位置在最右邊那個紅框的地方，訊號由低變高的時候觸發
 // 每一個這種位置都會進來這個函數讓 counter 加 1.
 counter ++;
}
```

一分钟是 60 秒，一秒是 1000ms ，除以转一圈所花的时间，就能计算出每分钟的转速，取得转速以后就知道前进的距离，很多事情就可以用公式来计算取得了。不过这里要注意是，里面使用了函数 **millis()**，这个函数只能计算到千分之一秒，万一轮子的转动速度过快，这函数会很不精确，也可以考虑改用 micros() 函数，可以计算到百万分之一秒的精确，只是我们这台示范车采用的是每分钟转速才 70 转的减速马达，也就是一秒一圈左右，以中断来看，一秒触发约 20 次，每次约 50ms 的时间，millis()已经很足够了。

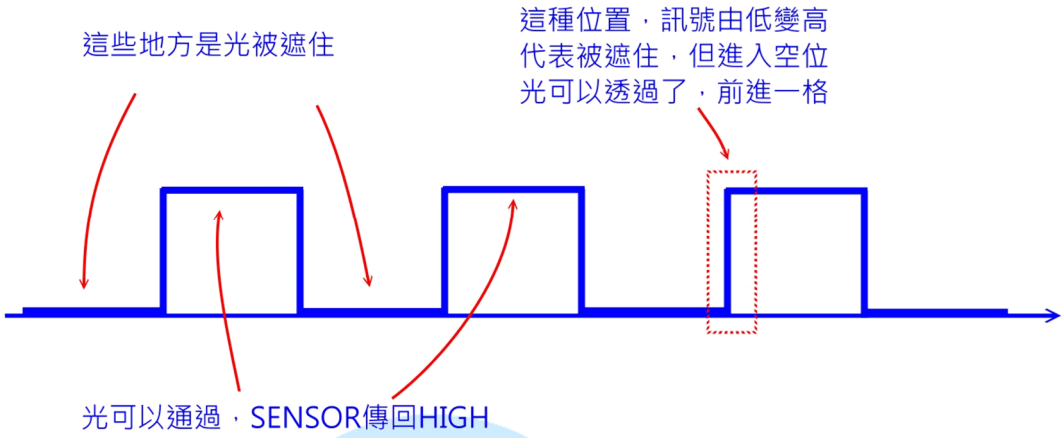


图 5 编码器的讯号变化

有了转速，又知道车轮胎周长是 65mm，那么要计算车子一分钟前进多远、或是想要知道前进一公尺需要多少时间就都很好计算了，就可以用来作未来的车子行进策略。只是这种作法的准确度相对比较低，真的要高一点的准确度就得使用像是旋转编码器，分辨率高一点的还得搭配指令周期比较快的 CPU 才有办法作的比较好，成本也会明显变高，这就后面再来谈。

三、 置物箱开合

既然是自动搬运自走车，那就会需要有置物的货架，这必须是一个密闭的箱子，想想看，若是想要把某个物品送到某人手上，那就不能让自走车在途中被不相干的人拿走物品；制作便当配送车也是这样，必须想办法确保每个人都只能拿到一个便当，不然第一个人就把所有便当拿走，那后面的人都不用吃了。

要实践这样的概念，就要有个可以开/关的密闭箱子，这有很多种方式可以作到，电磁铁、马达等等，这里选择简单的利用 SERVO 和简单的机构来办到，如图 6，这是一个有盖子可以打开的密闭盒子，和 SERVO 摆臂连接，就可以利用 SERVO 摆臂的上下来作盒子开盒的控制

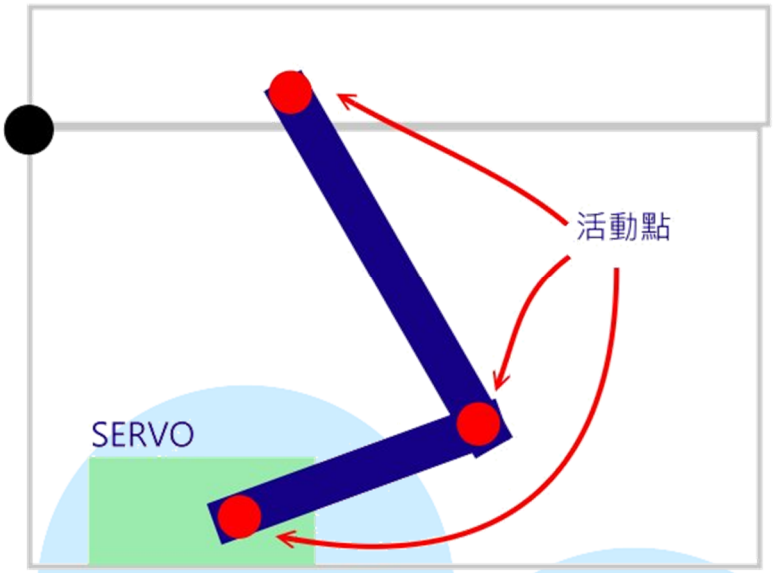


图 6. SERVO 控制盒子

盒子原本是关闭状态，这时只要 SERVO 的摆臂往上移动，就可以使盒子打开，如图 7 这样，SERVO 越往上推，盖子就开的越高，这样只要预先把盒子盖上时的 SERVO 位置(假设是 PWM 1000us 的位置)和盒子打开到想要的地方的 SERVO 位置(假设是 PWM 1750us 的位置)都先测试好，那程序就可以简单的控制盒子开关了。而至于什么时后才让 SERVO 作这个动作，则可以利用 RFID、遥控器、红外线、蓝芽...等等，有很多种解法，就看使用者如何选择，笔者之前作办公室便当自动配送车，是在每个员工的桌边有一个 RFID，车子走经过这种 RFID 时就会停下来等五秒，每个员工手上的手表背面也贴一个专属该员工的 RFID，员工看到车子停下，可以用手表去感应，车子就会打开，并推出一个便当，让他拿走，这样也可以顺便控制一个人只能领一个不会多拿。当然该员工万一当时没有在车子旁那就不用领便当了，稍后再到车子停车区补领就好。

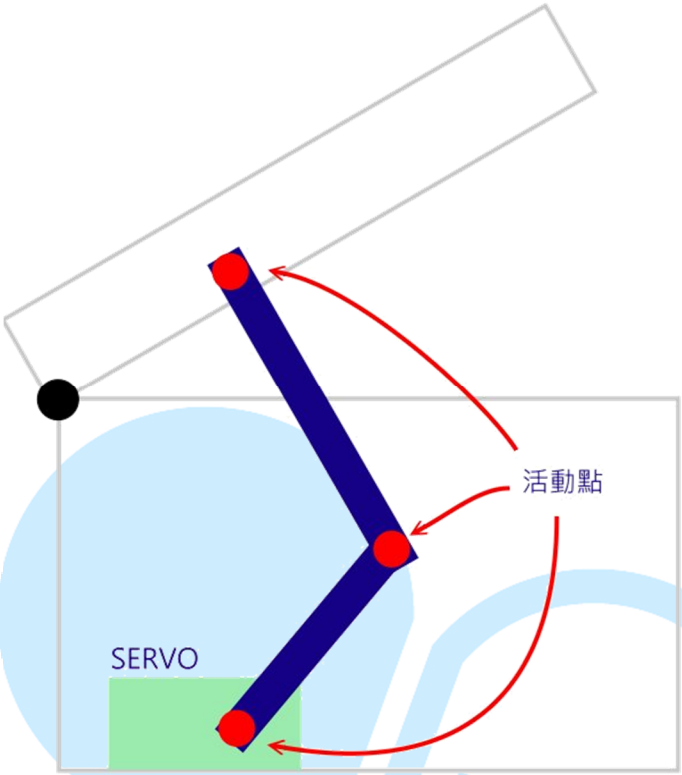


图 7. SERVO 控制盒子的打开

写好的范例程序代码会像是这样:

```
#include <Servo.h>
Servo myservo; // 建立 Servo 物件，控制伺服馬達
void setup()
{
  pinMode(2,INPUT); // 設定由 digital 2 接一個按鈕
  myservo.attach(3);   // 連接 digital 腳位 3
}
void loop()
{
  int b=digitalRead(2);
  if (b==HIGH) // 代表按鈕被按下
    myservo. writeMicroseconds(1750); // 控制打開盒子
 else
    myservo. writeMicroseconds(1000); // 關閉盒子
}
```

这个程序就可以简单的利用一个按钮来测试，按下按钮盒子会打开，放开按钮盒子就会关闭，后面我们就可以利用这个功能来控制。这里也阐述一个观念，以前我们写一个范例的时候都是以直接写出来为原则，但后面因为渐渐会偏向比较大的程序项目，直接一口气写出所有东西，万一最后项目不会动，会很难侦测出问题到底在哪里，所以笔者通常喜欢这样一个功能单独一段写出来，等最后功能都写完了，再一段一段的拼起来，作出最后的成品，通常根据经验这样比较容易，遇到错误也好解决。这个功能也可以使用滑轨+步进马达来作，效果会更好，但这只是台小车子，就不用作的那么复杂了。

四、 红外循线的应用

要让自走车能够精密的沿着既定的胶带轨道行走，那就要让自走车有感知的能力，使用影像辨识可以有最佳效果，但缺点是需要很强的 CPU 运算，摄影机取得影像也要够快，这里采用另外一种常见的简单解法，利用红外线寻线传感器，如图 8

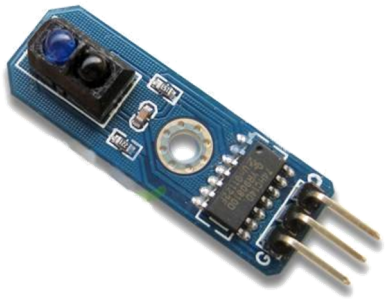


图 8 红外线寻线传感器

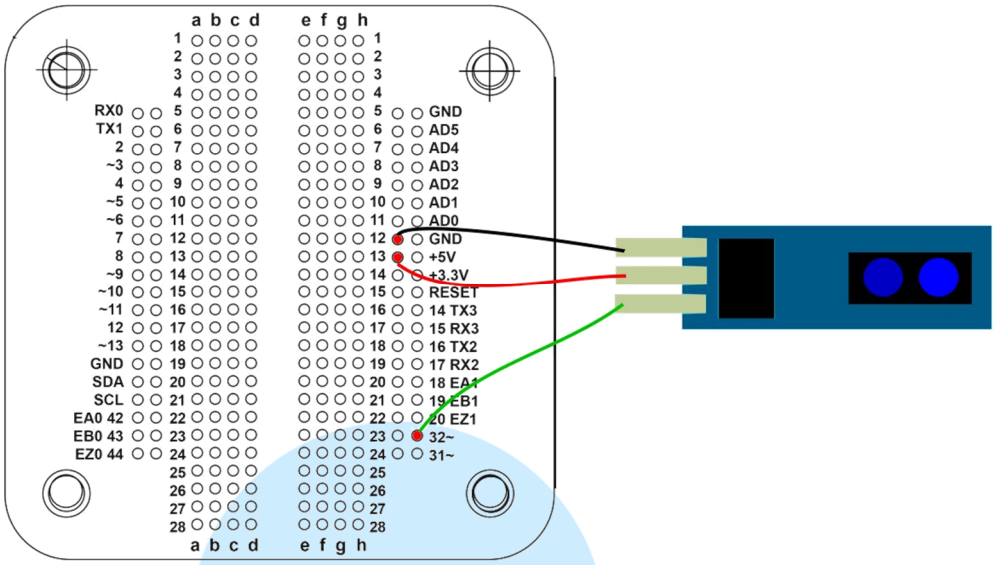


图 9 红外线寻线传感器接线图

使用方式一样是 GND、VCC 和一条接到 digital 脚位的 Sng。

这种传感器的原理一样是利用红外线，发射微弱的红外线出去，由另外一个接收头来感知反弹回来的光线，通常，多数颜色都能反射光回来，只有黑色会吸收大部分，就可以来判断是否在黑色的线上面，目前多数的自走车比赛，使用的循线行走方式，都是透过这种传感器来实作的。使用时通常为了尽量的避免干扰，会把这种传感器尽量的安装接近地面，约是 1~2mm 的位置。使用也很简单，它的三条线分别是:VCC、GND、SNG。SNG 讯号线会在遇到黑线的时候输出 0，其他多数颜色的时候输出 1，就可以利用 digitalRead 指令来读取传感器的状态以便确认目前是不是在黑色胶带上方。但这里若是需要贴不同颜色胶带来作额外的处理，那就需要使用颜色传感器了，作法会相对复杂很多，这里先跳过不讨论。

红外线循线通常需要三个一组或是甚至五~七个，并排在一起放置，为什么要这么多呢？主要是因为看”想要克服的线”到底有多复杂而定。先来看图 10

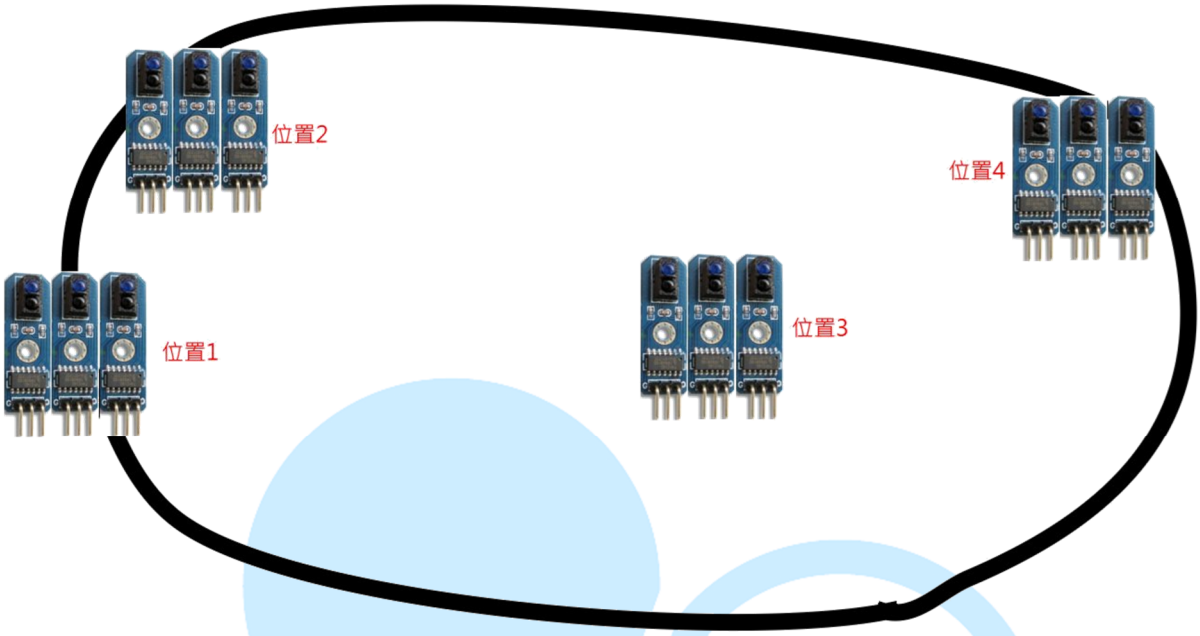


图 10 红外线寻线传感器遇到的各种状态

这是一个简单的地形，图中使用了三个红外线传感器并排在一起，用来侦测并沿着黑色的胶带前进，只使用三个是因为场景只是个单纯的一圈封闭圆，若是遇到复杂的形状或是很多叉路的，那就要多装几个才能处理的了。笔者制作这个轨道的黑色胶带(一定要黑色，这样和地板的白色会有高度反差，辨识结果会很正确)是 1.5cm 的宽度，三个红外线传感器彼此安排的距离是 1.2cm，所以若是把正中央的红外线传感器对准胶带的正中央时，中间的传感器会感应到黑色的胶带，并传回 digital　LOW 的讯号，左右两边则因为感应到地板白色的反光，会传回 digital　HIGH 的讯号，程序就可以据此判断，这时候车子正对胶带，可以令左右的轮子同样的速度向前转动，直线前进，就像图中的位置 1 那样的情况。

再来看位置 2 的情况，车子的左边和中间的红外线传感器因为都压在黑胶带上面，都会传回 LOW 的讯号，但右边的红外线传感器在地板上传回 HIGH 的讯号，程序可以判断车子偏右了，导致同时两个传感器都侦测到线，这时可令车子的右边轮子转快一点，左边轮子转慢点，让车子偏左，回复到线中央。关于如

何回到线中央，除了前面的讲的两个轮子转不同速度(使用 analogWrite 就可以控制速度)来达成外，还可以直接让两颗轮子转不同方向，以这个例子来说，可以使右边轮子转快，左边轮子”反转”，这样的方向修正效果会很明显比前一个快，主要是用在不同曲度的弯道，像是 90 度甚至更大角度的大转弯，就要用这种方式才有办法及时的过弯，甚至过弯前还得搭配两个轮子同时减速，避免过弯太急而翻车。这在比较高速需求的计算机鼠领域里面就会用上，得搭配更多个红外线传感器莱侦测更大的范围，取得较多的信息作判断。不过我们的车子行进速度并没有那么快，路况也只是大圆型没那么复杂，采用一种方式就好了。

位置 3，车子位于某处，红外线传感器没有侦测到任何东西，这时候有个问题，那就是车子到底在哪里?有可能在整个黑色胶带圈的范围内，那就很容易，随便往哪个方向开，都有机会直接遇到黑色胶带，这时只要使用前面讲过的方法就可以继续使车子沿着线前进；但万一车子这时是在线的外围呢?! 要导引车子回到在线的难度就会比较高，不能使用随便选个方向的方式，不然万一选到背离线的方向，那车子就永远回不来了。可使用在胶带范围中央放一个蓝芽发射器，车子上也装，这样就可以利用蓝芽的讯号强弱直接判断出车子是远离还是接近线圈，慢慢的修正位置回到在线，但这实作方式比较复杂容后再叙。

位置 4，这个状况和位置 2 颠倒，正好是右边和中间的传感器感应到黑线，前进时遇到这个情况代表车子偏左了，需要左边的轮子转快一点把车子带回右边；或是利用左边轮子正转，右边轮子逆转的方式，快速的修正方向回右边。

总观来说，以安装三个传感器的情况下，共会有 2 的三次方等于 8 种状况，

程序代码约是如下:

```
int Left_IR=5,Middle_IR=7,Right_IR=6;
int spd_a=255, spd_b=255; //  控制左右的速度
void setup()
{
  pinMode(Left_IR,INPUT);
  pinMode(Middle_IR,INPUT);
  pinMode(Right_IR,INPUT);
}
void loop()
{
  int L,M,R;
  int pos;
  L=digitalRead(Left_IR);
  M=digitalRead(Middle_IR);
  R=digitalRead(Right_IR);
  pos=L*4+M*2+R;
  switch (pos)
  {
  case 0://000   整個壓在線上，左轉(當然也可以使用右轉)
    turn_left();
    break;
  case 1://001，偏右一些，要左轉
    turn_left();
    break;
  case 2:// 010
  //左右有感應到線，中間沒有，這種情況在這個場景不會發生，先跳過不處理
  //但在有叉路的場景這可能代表正在叉路上，就不能跳過了
    break;
  case 3: //011，相對於 case1  ，這個情況偏右比較多，要轉多一點
```

```
      turn_left();
        turn_left(); //  連續執行兩次左轉
        // 或是另外用函數，或是用參數來處理旋轉時間，或是原地往左自轉
        break;
    case 4://100，  偏左一些，要右轉
        turn_right();
        break;
    case 5://101，在線正中央，直走
        move_forward()
        break;
    case 6://110，偏左很多，右轉多一點修正回來
        turn_right();
        turn_right();
        break;
    case 7://   111  沒感應到，直走，看能不能有機會遇到線
        move_forward();
        break;
    }
    delay(100);
}

void move_forward()
{
      digitalWrite(M1a,HIGH);
      digitalWrite(M1b, LOW);
      analogWrite(ENa, spd_a);   //左邊馬達速度
      digitalWrite(M2a,HIGH);
      digitalWrite(M2b, LOW);
      analogWrite(ENb, spd_b);   //右邊馬達速度
}

void turn_right()
{
```

```
  digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, spd_a);   //左邊馬達速度
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b-100);   //右邊馬達速度減慢，就有右轉效果
}

void turn_left()
{
    digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, spd_a-100);   //左邊馬達速度變慢就有左轉效果
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b);   //右邊馬達速度
}

void turn_left_rotate() //  向左邊原地旋轉
{
    digitalWrite(M1a, LOW); //  這兩行的狀態和 turn_left()顛倒，馬達反轉
    digitalWrite(M1b, HIGH);
    analogWrite(ENa, spd_a);   //左邊馬達速度
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b);   //右邊馬達速度
}

void move_back() //  後退，實際上就是前進的反方向旋轉
 {
```

```
        digitalWrite(M1a,LOW);
          digitalWrite(M1b, HIGH);
          analogWrite(ENa, spd_a);   //左邊馬達速度
          digitalWrite(M2a, LOW);
          digitalWrite(M2b, HIGH);
          analogWrite(ENb, spd_b);   //右邊馬達速度
    }
```

红外线感测的另外一个用途是防摔落，有些时候可能自走车刚好走到楼梯边，若是没有围住，车子可能就滚下去造成损坏了。一般这个问题有两种解法，一个是在车头使用红外线对地面作测距，假设是一般的地面，测距传回 10cm 就是正常，但走到楼梯边，因为楼梯往下是忽然高度向下，测距会忽然传回远大于 10cm 的数字，那就要令车子立刻停下或是后退，如图 11 这种 sharp 出品的红外线测距传感器，可使用距离为 10 - 80 cm，用来量取是否到楼梯边缘正适合。

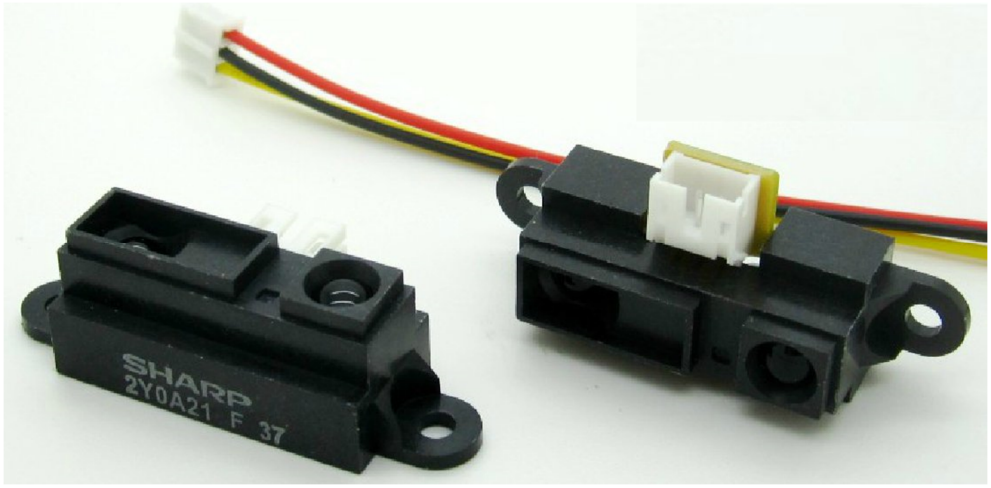


图 11 红外线测距传感器

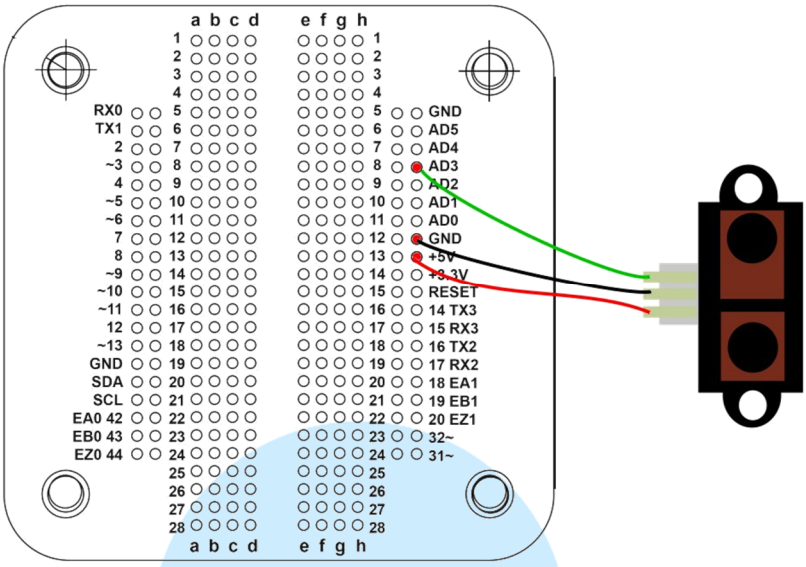


图 12 红外线测距传感器接线图

控制红外线测距读取距离信息的代码段如下:

```
// GP2D12 紅外線距離感測
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  int IR;
  IR = read_IR (3); //  接 analog P3  腳
  Serial.print("IR distance = ");
  Serial.print(IR);
  Serial.println(" cm");
  delay(500); //每隔 0.5 秒顯示一次距離
}

float   read_IR(byte pin)
{
  int tmp;
```

```
    tmp = analogRead(pin);
    if (tmp < 3) return -1; //  小於 3 不是正確數值
      return (6762.0 /((float)tmp - 9.0)) - 4.0;
    }
```

程序代码中的那段奇怪的公式(6762.0 /((float)tmp - 9.0)) - 4.0; 是因为传回的电压和距离成曲线比例关系，并不是直线的关系，所以，先取得约是 10~80cm 中每距离一公分的数值，填入 EXCEL 里面，作线性化公式转换以后得到的公式，这样才可以比较直觉的取得红外线测得的距离。

五、 超音波的应用

上面提到红外线测距，对于距离的测量还算准确，但有个问题是测量的不够远，这时候就需要超音波的帮忙了。红外线测距使用的原理是发出红外线，照射到物体以后，利用传感器去检测红外线的回波，主要会遇到的干扰就是会发出热源的物体在附近，会使得接到的回波受到干扰导致测得的距离不正确，但一般来说问题不大，最主要还是距离到一个程度，超音波过度的发散，导致接不到回波。超音波则是发出人耳听不到的 40KHz 的高频音波，撞到物体后会反弹回来，接收器检测到这个音波以后，就可以利用声音在空气中的传播速度约是每秒 331 公尺(摄氏 25 度 C 时)，乘上传感器发出音波到收到音波的这段时间的一半，来计算距离，算是很直觉的一种方式

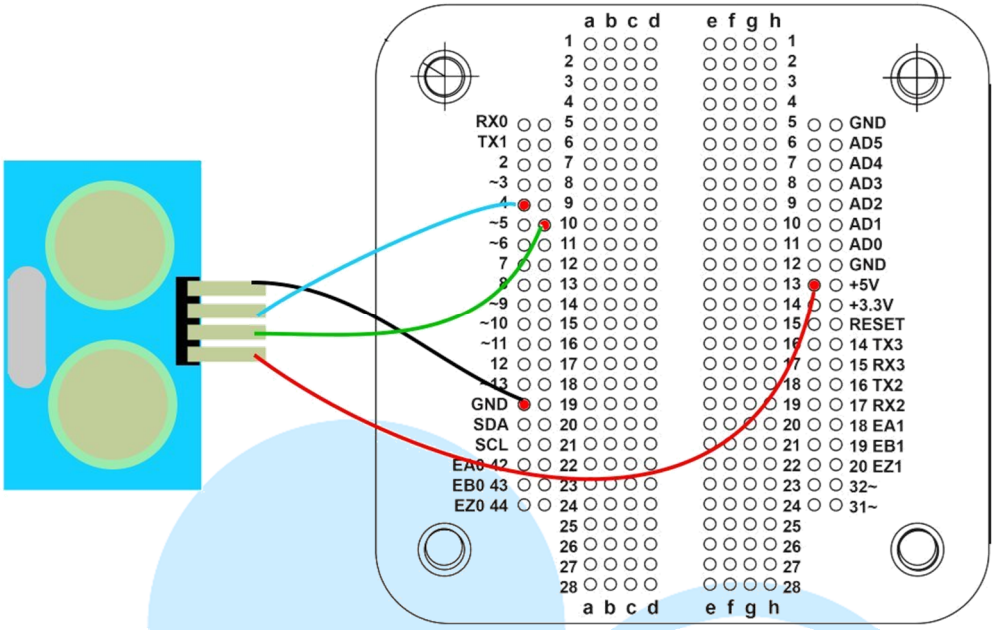


图 13 超音波测距传感器接线图

控制用的程序代码如下

```
int sonic_echo =4; //  接到 digital 4
int sonic_trig=5; //  接到 digital 5
void setup()
{
  Serial.begin(9600);
  pinMode(sonic_echo, INPUT);
  pinMode(sonic_trig, OUTPUT);
}
void loop()
{
    digitalWrite(sonic_trig, LOW);
    delayMicroseconds(2);
    digitalWrite(sonic_trig, HIGH); // 送出高電位 10μs
    delayMicroseconds(10); //  等待 10us  的時間，讓模組動作
準備好發出超音波
```

```
    digitalWrite(sonic_trig, LOW); //  時間到立刻設定為 LOW
      // 等待超音波回波讀取，並量取所經過的時間長短
      int distance = pulseIn(sonic_echo, HIGH);
      distance= distance/58;     // 轉換為 CM
      Serial.println(distance);
      delay(300);
   }
```

这个程序就可以简单的读取到超音波的距离，但其中的 int distance = pulseIn(sonic_echo, HIGH);取得的这个距离，约是号称介于 1~500cm，为何说号称呢？因为超音波相比于红外线的抗干扰能力更差，音波出去以后，有很多状况导致答案其实不正确，像是遇到柔软的海绵，可能完全被吸收没有回波、距离过远、没有和被测物 90 度垂直，导致回波反弹到别处或是散射的过于微弱等等很多因素，都会害 pulseIn 函数发生 timeout，计算出来的数值当然就不正确了。笔者实际使用，约是 1~220cm 左右的范围是可信的，超过范围就开始很不稳定甚至完全量不到，测试原因主要是选用的超音波的散射角度太大，导致距离超过两公尺以后就散的太大，根本抓不到回波，或说必须某个垂直正对障碍物的角度才能接到，但通常不可能刚好垂直，不过就算这样也比红外线好很多了，毕竟上面使用的红外线也只有 80cm 的距离而已。

六、 扫地机路径规划简介

接下来我们来谈谈扫地机路径规划的问题，以笔者在工厂碰过很多种扫地机的情况来说，一般扫地机的行进路线会想要尽量的涵盖整个房间的所有地面，然后扩充到整个房子所有的房间。这样的功能需求从简单到复杂有很多种作法，最常见就是沿着边线走，然后撞到物体随机改方向，这部分需要利用上极限开关来作碰撞感测(这样才能做到轻轻碰撞的效果，总不能用力撞障碍吧)，把所有可以碰撞到物体的方向都安装，一般来说是前方和侧面，碰到物体时，极限开关被

按下，程序就会知道这个方向有障碍，立刻改变马达转动的方向，然后继续走，这种作法通常用在一千多台币一台的入门扫地机，通常扫不干净，也扫不完全，这种作法我们只要用前面谈过的功能来组合就可以轻易作出来

一般想要完全涵盖整个室内面积，之字形、回字型两种基本路径规划是常见的，底下就来看看这些作法。首先，要先了解一件事情，以扫地机来说，通长的处理方式是以"房间" 为单位，扫完一间在扫另外一间，或是说另外一区，而怎么知道这是另外一区?通常得要车子有配备编码器，并且走过整个房子内部，已经建立好房子的整个室内地图，这时候就可以运算出每一个区域。

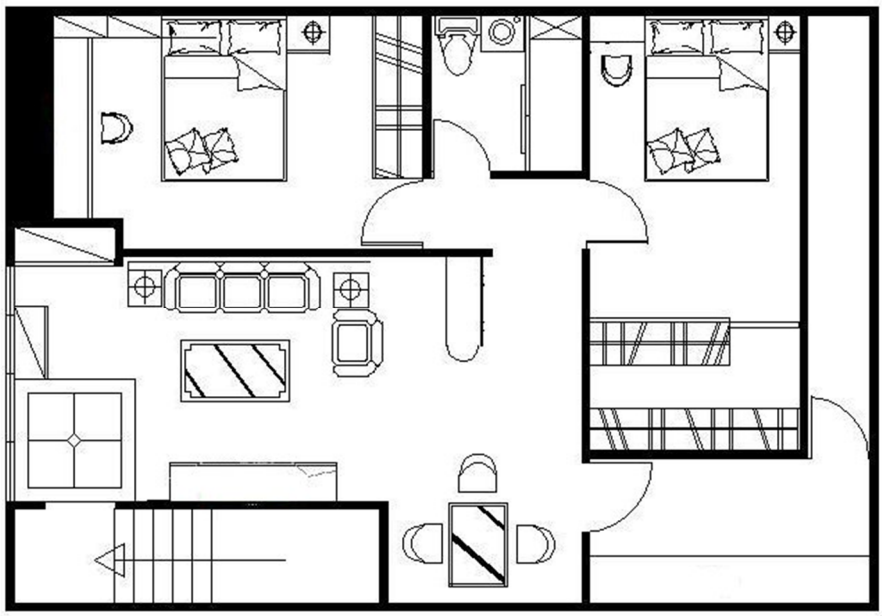


图 14 一个正常的室内的地图

图 14 是一个房子内部的平面图，车子先走过整个室内，因为车上的编码器，车子走的时候可以对每个经过的位置作记录，每个位置在车子的内存理仿真出来

的大地图上都是一个坐标，每个坐标可以简单的分成可通过和不可通过两种情况，等车子走过整个室内以后，一个虚拟的地图就能在车子的内存里面建构完成，这个动作叫做 SLAM(Simultaneous localization and mapping) 同步建图与定位算法，这实作的难度超过这个章节过多，容后再叙，这里先简述大致的理论就好。

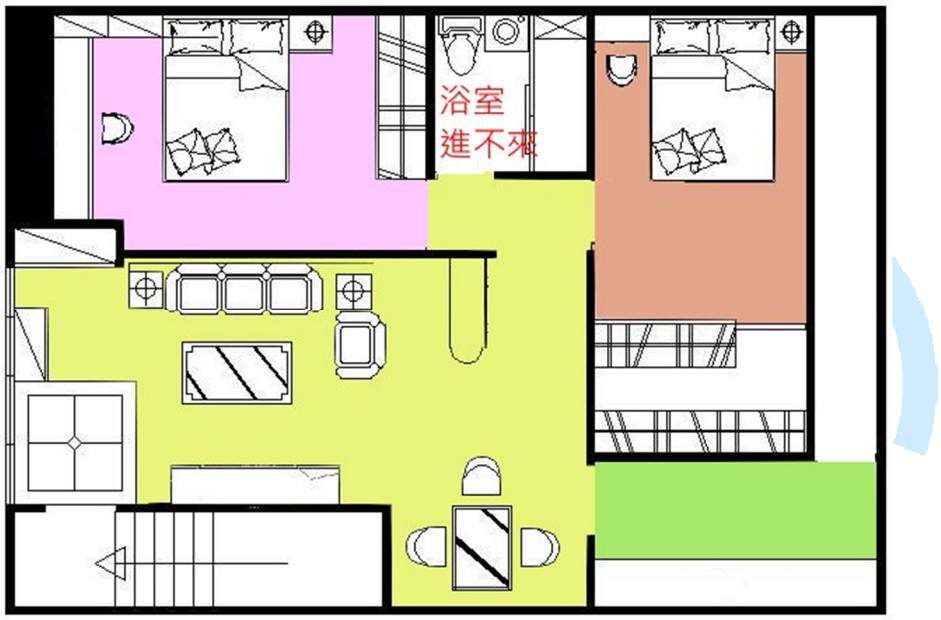


图 15 处理分割后的室内的地图

图 15 是经过 SLAM 处理，并分割计算后得出的区块图，用颜色来看，可以看到分成左右两间卧室，和左下的客厅和右下的厨房。主要的分割依据是门框。有了这样的虚拟地图和边界后，就可以开始来作清扫的动作，先用简单的图标来看之字形和回字型行走法是怎么回事，参考图 16

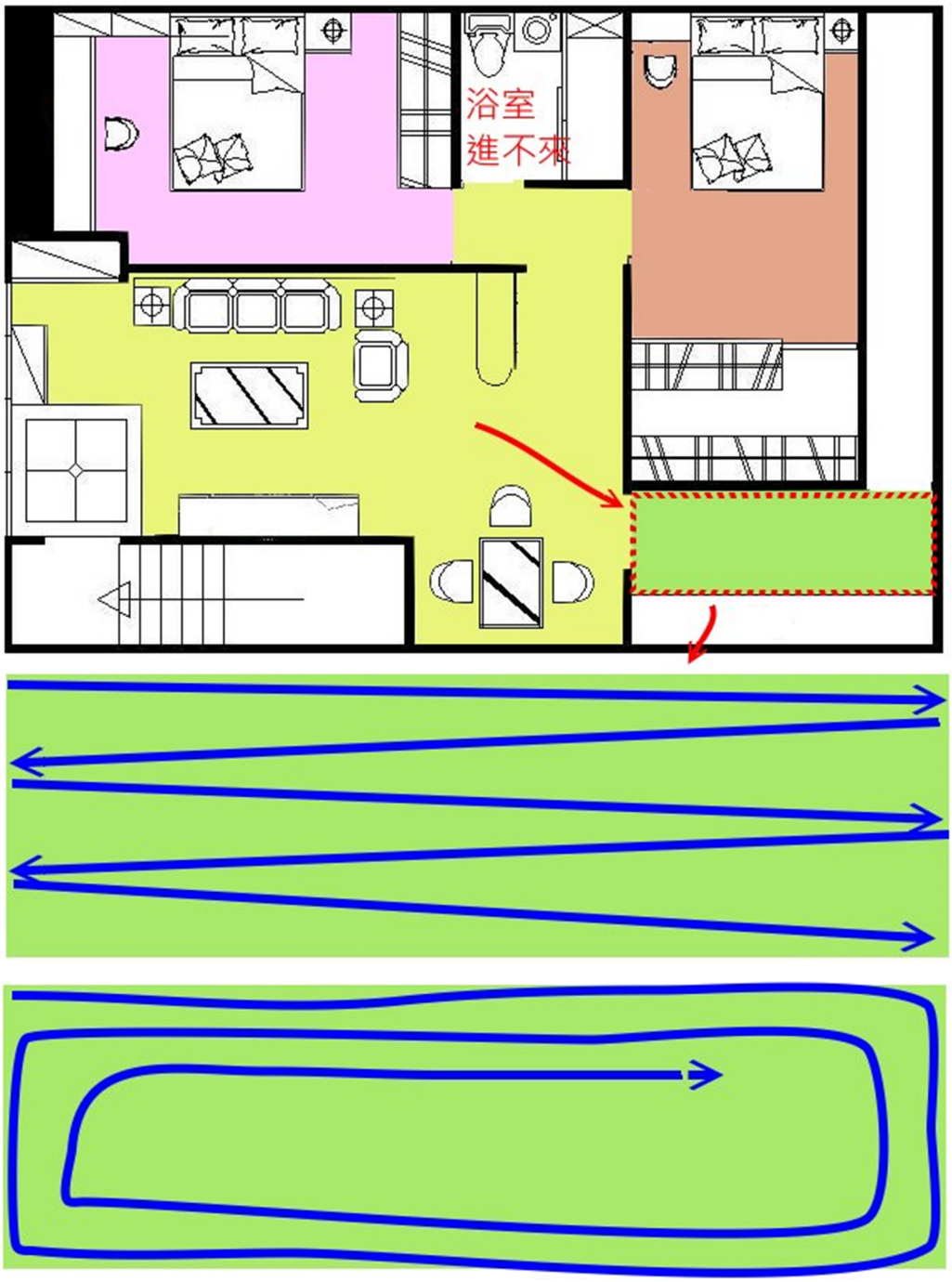


图 16 针对厨房的两种不同走法

可以看到图中的之字形就是左右来回走，把整个区域覆盖；回字型走法就是类似一直绕着区域边缘，一直往中心方向走，要作到这样的方式，就得善用传感器取得的信息，像之字形，就是一直往前走，直到撞到墙壁(撞墙侦测可以利

用红外线，或是扫地机常用的极限开关)或是遇到虚拟地图中的边界，原地旋转 175 度(180 度就往回走了)继续往前走到撞墙或虚拟边界，这种原地旋转 175 度的作法比起旋转 90 度走一点点在旋转 90 度的方式，容易造成有些地方会扫不到，但无所谓，通常扫地机为了扫干净会跑两次，已基本能覆盖到所有范围，而且这里只是为了简单的讲解原理，实作上还会因为硬件的区别或是环境的因素，有很多不同的细部调整，并不是很容易能办到的。

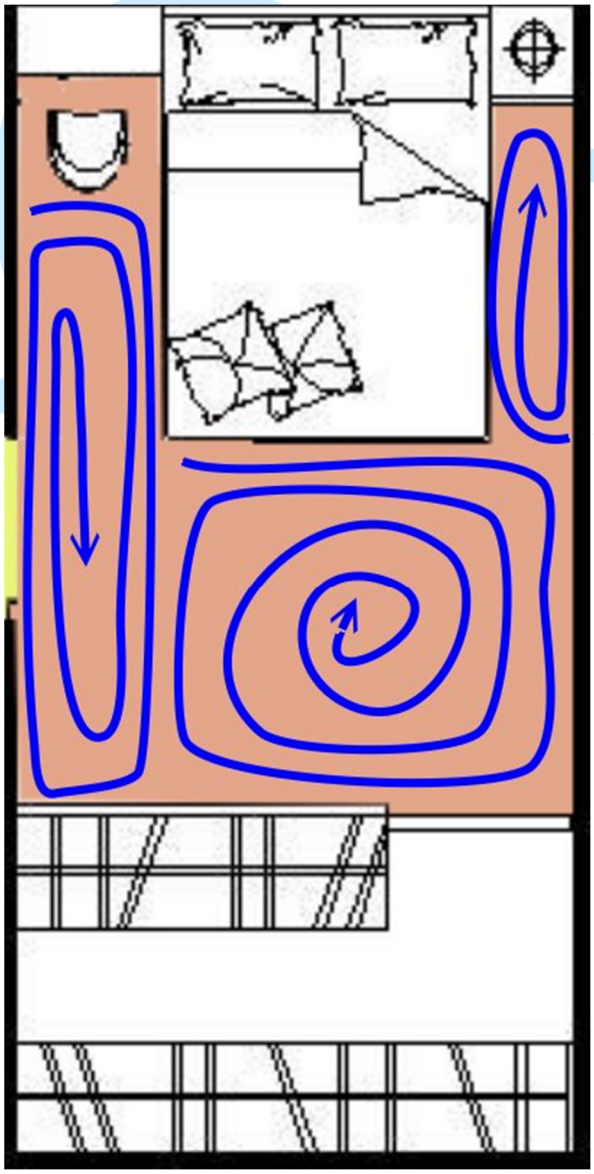


图 17 更复杂的环境的区域覆盖

七、 结论

自走车领域有很多的应用，而要达成这些应用需要很多各式各样的功能结合在一起才能办到。生活中的自走车也会因为各种需求的产生而越来越多，如何作出一台更实用的自走车还要靠读者的想象力去发挥和实作，我们后面的章节还会谈到更多的实作技巧来达成这些功能。