8 加速度

當你轉動你的智慧型手機時，該螢幕可能會從直式轉變成橫式。手機是如何進行辨識的呢？智慧型手機有一個內建的加速器，而且重力只能向下方加速，因此就是這個往下拉的重力讓手機知道它本身的方位。

有一些遊戲是使用加速器，藉由在空中傾斜手機來操控的。時下流行的 Wii 遊戲機是一種使用加速器的控制器。在本章節中，你會破解一台 Wii 控制器並且利用它的感測器。我們在 *Make：Arduino 機器人及小裝置專題製作* 使用手機的加速器來控制來足球機器人。

當前的筆記型電腦和桌上型電腦中的硬碟都可以承受很大的衝擊。許多可以接受 150g（關機時）的加速度，這是一個可以讓人致命的加速度。為了讓該裝置可以承受該撞擊所受的影響，有些硬碟會先自行關機：如果該硬碟加速器偵測出它是在自由落體的狀態時，它會自動將驅動臂移離敏感的硬碟區。

你有沒有試著騎乘過能自動平衡的裝置，例如 Segway 或 Solowheel ？也許開始時會有些搖晃，但該裝置卻能夠保持直立不倒。

當一個自動平衡的裝置偵測到它即將向前傾倒時，它會很快地將車輪向前移動，再將該裝置本身直立起來。自動平衡裝置會用陀螺儀來測量角速度。（加速器會蒐集過多的累積誤差在自我平衡的裝置上作業。）

加速度與角速度

加速度是物體速度的變化率（當它減速或加速時）。角速度是用來測量物體的旋轉速度，以及它所賴以旋轉的軸。依據你的實驗項目，你可能需要用到加速度、角速度、或甚至於兩者都需要。

加速度的單位是 g，它是由地球重力所形成加速度的倍數。另一種常用加速度的單位是每秒平方米（米 / 秒 ^{2}）。自由落體的加速度（1g）是 9.81 米 / 秒 ^{2}。

為什麼加速度的單位是每秒平方呢？加速度是一種速度的變化。如果你使用每秒米（米 / 秒）為速度的單位，那麼加速度的單位（速度的變化）是每秒每秒米，或米 / 秒 ^{2}。

陀螺儀用來測量角速度，或是感測器環繞著軸心旋轉的速度。例如，陀螺儀可能會回報它的轉速為每秒 10 度。它們是用在自我的平衡儀以及飛機的陀螺儀。

表 8-1　加速器與陀螺儀

感測器	測量	說明	單位	重力
加速器	加速度	速度變化，加速或減速	m/s / s = m/s^2	是的，1g 向下
陀螺儀	角速度	角度變化，旋轉	rad/s（SI），一般為 deg/s 或 RPM	不考慮重力

實驗：用 MX2125 來加速

MX2125 是一個簡單的雙軸加速度感測器（如圖 8-1 所示）。它把加速度當作一個脈衝長度來回報，藉以簡化它的使用介面與程式碼。

真實的物質世界是一個三度的空間。物體可以向上及向下移動（y），向左及向右移動（x），和往復的移動（z）。雙軸感測器只用來測量其中的兩個軸。

MX2125 只測量到每個軸 3g 的範圍內。但有些感測器可以測量到極限的加速度。例如，ADXL377 的最大測量加速度是（200g），遠超過會致人於死的程度。因此，它比在經驗豐富的太空梭發射或高速 g 情境下的戰鬥機操控的要求還嚴格。它可以測量物體的加速度比從手槍射出的子彈還要快。當我們提出 Aalto-1 衛星太陽感測器的最初原型時，即使衛星所要求加速度的規格也沒有這個來得嚴格。

需要測量這種極端加速度的情況是很少發生的，而且也不太可能用麵包板來達成（因為光是需要測試的加速度就會讓你的實驗項目解體！）。它的成本雖然很低。但是，測試加速度的範圍越寬廣時（從 -250 g 至 +250 g）該裝置的精確度就會越差。

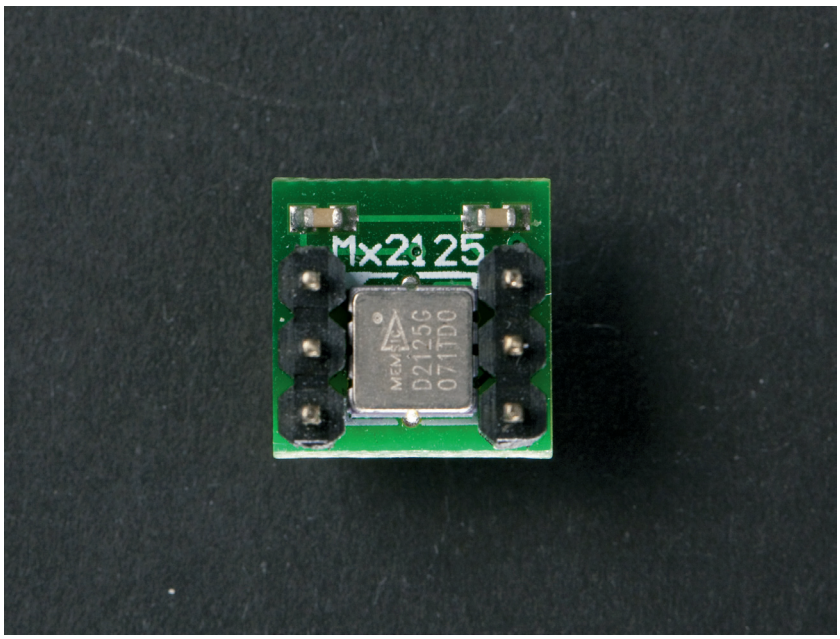

圖 8-1　MX2125 感測器

解碼 MX2125 的脈衝長度

一般情況下，加速器的換算公式可以在數據表裡找到。在這種情況下，需要動用更多的搜尋作業。

搜尋數據表

找尋數據表。最明顯的方式是搜尋零組件的名稱及「數據表（datasheet）」這個詞，但你也可能會在你購買零組件的網站找到（它通常是在產品明細的網頁裡）。

例如，當我們搜尋「MX2125 數據表（MX2125 datasheet）」時，我們會找到 Parallax 接口板的數據表（*http://bit.ly/Prbb3c*），而不是 Memsic 半導體有關 MX2125 晶片的數據表。Parallax 的數據表並沒有包含你所需要的訊息。

然而，脈衝長度對 g 作用力的換算公式可以在 Parallax Memsic 2125 的加速器示範組合文件裡找到（*http://bit.ly/PrblHL*）。很幸運的是，當我們後來檢查 Parallax 網頁時，我們看到了實際晶片的數據表。我們還能用稍微不同的搜尋詞「MXD2125」找到該數據資料，這是該電路板上晶片的零組件編號。該實際的數據表包含了其他文件所看不到的大量訊息。

MX2125 的作業原理是將裝置內的氣體加熱產生氣泡，然後測量該氣泡是如何移動的。

當電源開啟時，MX2125 會回報每個軸的加速度，也就是每秒 100 個脈衝。連續的高電平與低電平的訊號構成 100 Hz 的方波。加速度越快，該方波停留在高電平的時間越長，而相對停留在低電平的時間則越少。你可以讀取這些脈衝來決定加速度的值。

一個完整的波長包含一個高電平和一個低電平。一個波長（高電平 + 低電平）所耗費的時間被稱為週期（T）。讓我們把高電平這部份的時間稱為 *tHIGH*（高電平的時間）。

該**工作週期**告訴你該方波有多少高電平。工作週期是一個百分比，例如 50%（0.50）或 80%（0.80）。

```
dutyCycle = tHIGH / T
```

根據數據表和其他文件，週期 T 的標準設定是 10 毫秒：

```
dutyCycle = tHIGH / 10 ms
```

下面是數據表中加速度的公式：

```
A = (tHIGH/T-0.50)/20%
```

或者，用 `dutyCycle` 替換 `tHIGH/T` 以及用 `.2` 替換 `20%`：

```
A = (dutyCycle-0.50)/.2
```

現在它可以書寫如下（因為 `x/.2` 等於 `5*x`）：

```
A = 5*(dutyCycle-0.50)
```

或是：

```
A = 5*(tHIGH/T-0.50)
```

當沒有加速度（0 g）時，工作週期為 50%：

```
0   = 5*(dutyCycle-0.50)
0/5 = dutyCycle-0.50
0   = dutyCycle-0.50
.50 = dutyCycle
```

在我們最初檢查的時候，Parallax 以及 Memsic 半導體文件的乘數不相吻合：在 Memsic 半導體的文件使用 1/20%（5），而 Parallax 使用 1/12.5%（8）。在實驗中，我們發現 1/12.5%（8）可以透過 Parallax 接口板提供合適的讀取值。而事實上，當我們稍後從 Memsic 半導體檢驗了儲存於 Parallax 網站的 MXD2125 數據表，兩者都符合 1/12.5% 的作法。這就是為什麼你需要小心使用你在網路上找到的數據表：一定要透過實驗驗證該參數值。因此，在此我們將使用 8 當作乘數使用：

```
A = 8*(tHIGH/T-0.50)
```

因為 Arduino 的 `pulseIn()` 回傳以微秒為單位的脈衝長度（1 微秒 = 0.001 毫秒 = 1e-6 秒），這個公式可以被修改以微秒為單位：

```
A = 8 * ( tHIGH/(10*1000) - 0.5)
```

A 的單位是 g，等於 9.81 m/s^2。

例如，如果 tHIGH 是 5000 微秒（5 毫秒），該工作週期為

```
dutyCycle = 5 ms / 10 ms = 0.50 = 50%
```

結果等於 0 g：

```
A = 8*(0.50-0.5) = 8*0 = 0          // g
```

在此提供另一個範例，將 tHIGH 視為 6250 微秒（6.25 毫秒）

```
A = 8*(6250/10000-0.5) = 8*(0.625-0.5) = 8*0.125 = 1          // g
```

因此，一個 6.25 毫秒的脈衝代表 1 g 的加速度。

Arduino 的加速器程式碼與連接

如圖 8-2 所示為 Arduino 的電路簡圖。如圖所示將它連接，透過範例 8-1 載入程式碼。

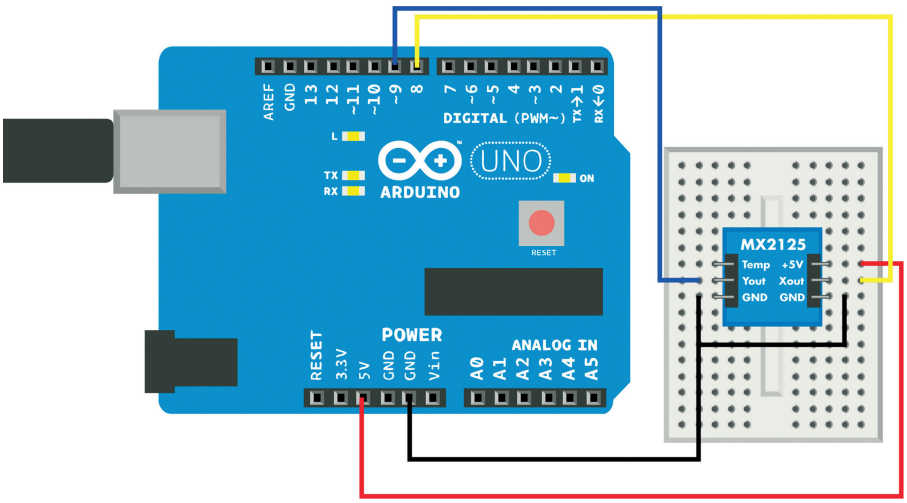

圖 8-2　Arduino 的 MX2125 雙軸加速器電路

範例 8-1　mx2125.ino

```
// mx2125.ino - 用 MX2125 在兩個軸上偵測它的加速度，並將結果列印到串列埠
// 版權所有 (c) BotBook.com - Karvinen, Karvinen, Valtokari

const int xPin = 8;
const int yPin = 9;


void setup() {
  Serial.begin(115200);
  pinMode(xPin, INPUT);
  pinMode(yPin, INPUT);
}

void loop() {
  int x = pulseIn(xPin, HIGH);  // ❶
  int y = pulseIn(yPin, HIGH);
  int x_mg = ((x / 10) - 500) * 8;      // ❷
  int y_mg = ((y / 10) - 500) * 8;
  Serial.print("Axels x: ");
  Serial.print(x_mg);
  Serial.print(" y: ");
  Serial.println(y_mg);
  delay(10);
}
```

❶ 脈衝的長度告訴你它的加速度。`pulseIn()` 回傳的脈衝長度以微秒（µs）為單位。1 微秒 = 0.001 毫秒 = 0.000001 秒 = 1e-6 秒。

❷ 將輸出值換算為 *millig*，相等於重力常數 *g* 的千分之一。

Raspberry Pi 的加速器程式碼與連接

圖 8-3 所示為 Raspberry Pi 的接線簡圖。將所有零組件都連接起來，如圖所示，然後執行範例 8-2 所示的程式碼。

範例 8-2　mx2125.py

```
# mx2125.py - 列印加速度的軸參數值
# 版權所有 (c) BotBook.com - Karvinen, Karvinen, Valtokari
import time
import botbook_gpio as gpio

xPin = 24
yPin = 23

def readAxel(pin):
        gpio.mode(pin, "in")
        gpio.interruptMode(pin, "both")
        return gpio.pulseInHigh(pin)    # ❶

def main():
        x_g = 0
        y_g = 0
        while True:
                x = readAxel(xPin) * 1000
                y = readAxel(yPin) * 1000
                if(x < 10):     # ❷
                        x_g = ((x / 10) - 0.5) * 8      # ❸
                if(y < 10):
                        y_g = ((y / 10) - 0.5) * 8
                print ("Axels x: %fg, y: %fg" % (x_g, y_g))     #
                time.sleep(0.5)

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

❶ 當引腳為高電平時，測量該脈衝的長度。

❷ 捨棄超出範圍甚多的讀取值。

❸ 計算出沿著 x 軸的加速度（以 g 為單位）。一 g 是地球的重力所引發的加速度，每秒平方 9.81 米。

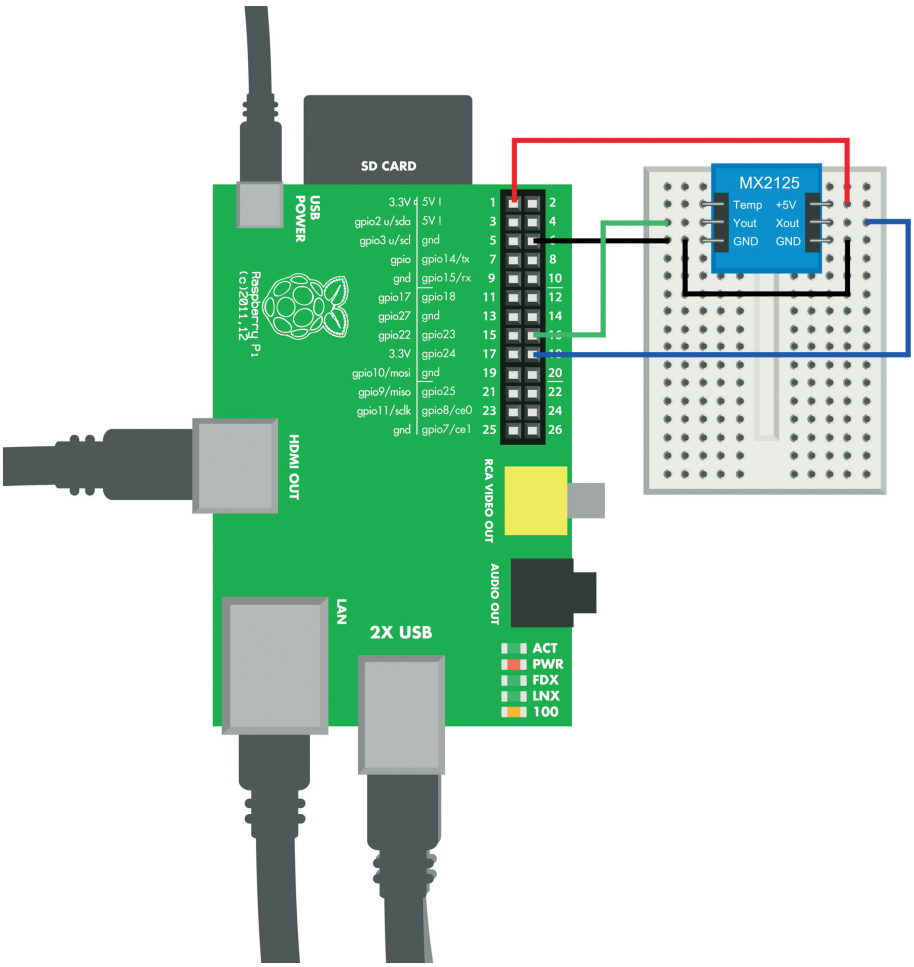

圖 8-3　Raspberry Pi 的 MX2125 雙軸加速器電路

實驗：加速器和陀螺儀一起使用

當加速器是靜止時，它會偵測到重力，而且可以分辨該物體上下的方位。陀螺儀可以準確地辨識方位，即使當它處於不停旋轉的狀態下。陀螺儀也不會受到重力的影響。

我們能否結合加速器和陀螺儀，而同時獲得兩者的優點呢？答案毫無疑問是肯定的。

一個 IMU（慣性測量器），將多個感測器和（可選）結合一些邏輯運算藉以獲得更精準且可靠的動作訊息。在這個實驗中，你可以測試 MPU 6050 微處理器的基本功能。

在一般情況下，慣性測量器比普通的加速器和陀螺儀更精確、價格更昂貴。它們還使用更進階的通訊協定來通訊，譬如，用 I2C 取代脈衝寬度訊號的通訊協定。

MPU 6050（如圖 8-4 所示）具備有加速器、陀螺儀、和微控制器在同一個晶片上。雖然當你在麵包板原型開發階段時，該空間不是那麼侷限，萬一你曾經在製作過程中經歷過開發期間，遭遇到空間不夠使用的窘境，至少你可以明白，這裡所有的功能都可以安置在一個小小的表面黏著元件裡，例如，早期的太陽光源感測器的原型設計，勉強放入一個大小約 10×10×10 公分的盒子裡。最後的成品必須符合衛星上的表面一個非常平坦的 5 毫米 ×5 毫米的範圍內。

MPU 6050 採用 I2C 通訊協定。幸好有 *python-smbus* 函式庫，Raspberry Pi 的程式碼比同等的 Arduino 程式碼來得更簡單、更容易。在一般情況下，Raspberry Pi 用較少的程式碼處理比 Arduino 更複雜的通訊協定。

業界標準的通訊協定

大多數裝置使用業界標準的通訊協定，而不是自行研發。

I2C 是最簡單的業界標準通訊協定之一。因為它的嚴格定義以及包括如何把數據進行編碼和解碼的通訊協定，通常是最方便使用的。你可以在本章節中找到 I2C 多個應用的實例。

SPI 也是一種常見的業界標準通訊協定。由於 SPI 留下了很多實際應用時的可選擇範例，編寫一個連接新 SPI 組件的介面可能會是一項艱鉅的任務。另外一方面，如果有程式碼範例或參考說明資訊，這只是一個複製貼上的事件而已，因為別人已經替你完成了艱難的作業部分。你可以在 *http://botbook.com/satellite* 找到 SPI 組件的使用範例，而不需要參考說明資訊。

經常發現串列方式有不同變化的作法：透過 USB 的串列、透過藍牙的串列方式、透過一些跳線的串列方式。你一直都在使用的舊款串列埠與 Arduino 的串列監控器異常地類似。它只解決一部分的問題：串列是用來定義如何透過導線傳送字元，但作業者仍然必須決定，如何用數字進行編碼和解碼的作業。

你可以在我們 *Make a Mind-Controlled Arduino Robot* 這本書找到一個透過 USB 破解串列埠的範例。

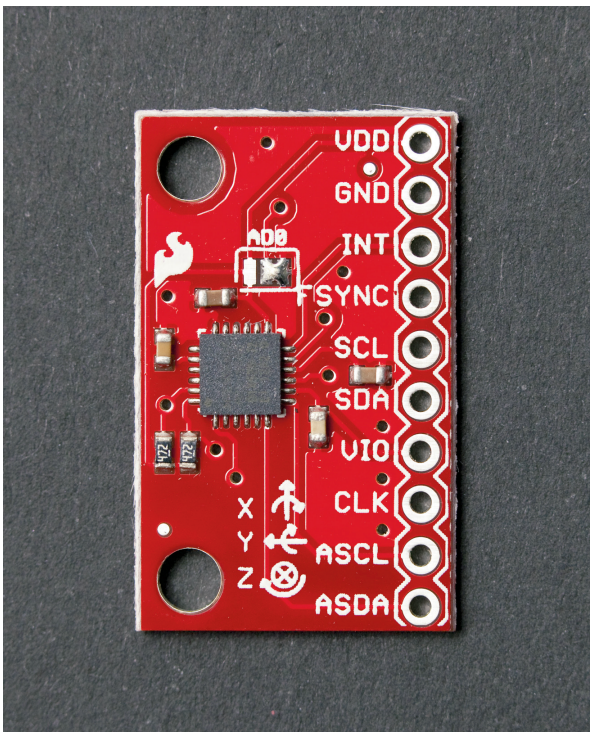

圖 8-4　MPU 6050

Arduino 的 MPU 6050 程式碼與連接

圖 8-5 所示為 Arduino 的接線簡圖。將所有元件連接起來，然後執行範例 8-3 的程式碼。

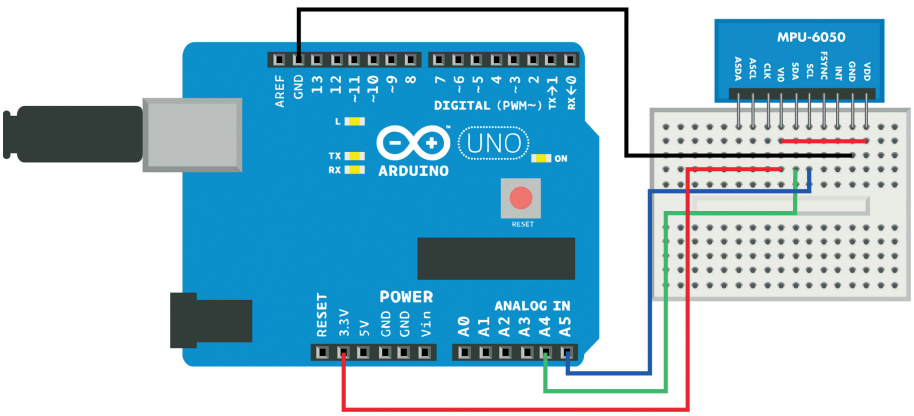

圖 8-5　Arduino 的 MPU 6050（加速器 + 陀螺儀）電路

> 難度高的程式碼！ *MPU 6050* 的程式碼裡含有艱難編程的概念比本書大部分其他的程式碼範例還要多。如果你覺得大小端序（*endianness*）、位元移位（*bit shifting*）、和結構很複雜，你不需要深入了解就可以直接使用該程式碼及參數值。
>
> 如果你想了解該程式碼，閱讀程式碼之後的說明，如第 *232* 頁「十六進制、二進制和其他編碼系統」及第 *236* 頁的「位元運算」。

範例 8-3　mpu_6050.ino

```
// mpu_6050.ino - 列印加速度 (m/s**2) 及角速度 (gyro, deg/s)
// 版權所有 (c) BotBook.com - Karvinen, Karvinen, Valtokari

#包含 <Wire.h>       // ❶

const char i2c_address = 0x68;          // ❷

const unsigned char sleep_mgmt = 0x6B;  // ❸
const unsigned char accel_x_out = 0x3B;

struct data_pdu // ❹
{
  int16_t x_accel;      // ❺
  int16_t y_accel;
  int16_t z_accel;
  int16_t temperature;  // ❻
  int16_t x_gyro;       // ❼
  int16_t y_gyro;
  int16_t z_gyro;
};

void setup() {
  Serial.begin(115200);
  Wire.begin(); // ❽
  write_i2c(sleep_mgmt,0x00);   // ❾
}

int16_t swap_int16_t(int16_t value)    // ❿
{
  int16_t left = value << 8;    // ⓫
  int16_t right = value >> 8;   // ⓬
  right = right & 0xFF; // ⓭
  return left | right;  // ⓮
}
```

```
void loop() {
  data_pdu pdu; // ⓯
  read_i2c(accel_x_out, (uint8_t *)&pdu, sizeof(data_pdu));    // ⓰

  pdu.x_accel = swap_int16_t(pdu.x_accel);     // ⓱
  pdu.y_accel = swap_int16_t(pdu.y_accel);
  pdu.z_accel = swap_int16_t(pdu.z_accel);
  pdu.temperature = swap_int16_t(pdu.temperature);     // ⓲
  pdu.x_gyro = swap_int16_t(pdu.x_gyro);
  pdu.y_gyro = swap_int16_t(pdu.y_gyro);
  pdu.z_gyro = swap_int16_t(pdu.z_gyro);

  float acc_x = pdu.x_accel / 16384.0f; // ⓳
  float acc_y = pdu.y_accel / 16384.0f;
  float acc_z = pdu.z_accel / 16384.0f;
  Serial.print("Accelerometer: x,y,z (");
  Serial.print(acc_x,3); Serial.print("g, ");   // ⓴
  Serial.print(acc_y,3); Serial.print("g, ");
  Serial.print(acc_z,3); Serial.println("g)");

  int zero_point = -512 - (340 * 35);   // ㉑
  double temperature = (pdu.temperature - zero_point) / 340.0;  // ㉒
  Serial.print("Temperature (C): ");
  Serial.println(temperature,2);

  Serial.print("Gyro: x,y,z (");
  Serial.print(pdu.x_gyro / 131.0f); Serial.print(" deg/s, ");  // ㉓
  Serial.print(pdu.y_gyro / 131.0f); Serial.print(" deg/s, ");
  Serial.print(pdu.z_gyro / 131.0f); Serial.println(" deg/s)");
  delay(1000);
}

void read_i2c(unsigned char reg, uint8_t *buffer, int size)     // ㉔
{
  Wire.beginTransmission(i2c_address);  // ㉕
  Wire.write(reg);      // ㉖
  Wire.endTransmission(false);  // ㉗
  Wire.requestFrom(i2c_address,size,true);      // ㉘

  int i = 0;    // ㉙
  while(Wire.available() && i < size) { // ㉚
    buffer[i] = Wire.read();    // ㉛
    i++;
  }
```

```
  if(i != size) {       // ㉜
    Serial.println("Error reading from i2c");
  }

}

void write_i2c(unsigned char reg, const uint8_t data)   // ㉝
{
  Wire.beginTransmission(i2c_address);  // ㉞
  Wire.write(reg);
  Wire.write(data);
  Wire.endTransmission(true);
}
```

❶ *Wire.h* 是 Arduino 用於 I2C 通訊協定的函式庫。*Wire.h* 是 Arduino IDE 整合環境的一部分，所以你可以直接在程式碼裡包含該函式庫。就不需要單獨安裝函式庫或手動複製任何函式庫檔案。

❷ MPU 6050 感測器的 I2C 位址。一個三條導線匯流排可以有很多的從屬裝置。每一個從屬裝置透過它的位址來辨識。通常情況下，一個 I2C 匯流排可以有 128（2^7）個從屬裝置。I2C 傳輸線只有大約 2 公尺長，這樣也實際上限制了導線的長度。該數字以十六進制表示，參考第 232 頁「十六進制、二進制和其他編碼系統」了解這個符號的意義。

❸ 該命令的寄存器是來自 MPU 6050 文件。如果你需要一個完整的命令列表，在網路上搜尋「MPU 6050 數據表（MPU 6050 data sheet）」和「MPU 6050 寄存器映射（MPU 6050 register map）」。該數字是十六進制，但你也可以用十進制來表示。

❹ *struct* 用於透過感測器對該回應進行解碼。一個結構（struct）將多個參數值結合在一起。C 結構僅用於數據，而且一個結構中不能包含任何函數。這使得該結構與你可能經由其他程式語言所認識的物件和類別是不一樣的。`struct data_pdu` 宣告了一個新的數據類型，你會在稍後用來宣告 `data_pdu` 類型的變數。這個結構擁有與感測器在通訊協定裡的數據欄位大小相同的變數。稍後，你就會從感測器讀取位元組直接進入結構體。然後你會去透過嵌入該結構的變數來獲得該參數值。的確，這是一個很巧妙的方法！

❺ 一種可用於儲存沿著水平 x 軸的加速度的變數。`int16_t` 類別是一個透過 avr-libc 函式庫定義特殊大小的整數（使用 Arduino 編譯器的 C 函式庫）。它是一個附帶符號（正數或負數）的 16 位元（2 個位元組）的整數。由於

該結構體被用於透過感測器的原始數據進行解碼，該數據類型必須有確實的大小。

❻ 這個感測器也會回報溫度值。即使你不需要該訊息，你還是必須設定一個變數，以便在 `data_pdu` 結構最終呈現的是一個正確的大小。

❼ x 軸（roll）周圍的角速度，透過 MPU 6050 陀螺儀的部分來讀取。

❽ 使用 *Wire.h* 函式庫初始化 I2C 通訊。

❾ 透過編寫命令 `0` 到睡眠管理寄存器 `0x6B` 來喚醒感測器。MPU 6050 一開始是在睡眠模式，所以這是一個必要的步驟。

❿ 交換參數值中的 2 個位元組。MPU 6050 是大端序（*big endian*）。而 Arduino 是小端序（*little endian*）就像一般的處理器一樣。請參考第 239 頁「位元端序，通常在小端序」。

⓫ 這種新的 2 個位元組（16 位元）的 `left` 變數最終成為該位元組最右邊的參數值。在 1 個位元組（8 位元）左移（<<）之後，最左邊的位元組被捨棄不用，因為它無法配合 `left` 變數的 2 個位元組。`left` 變數的右側位元組在位元寄存器裡被填入許多零。

⓬ 這個新的 2 個位元組（16 位元）`right` 變數現在是 `value` 最左邊的位元組。該 `right` 變數最左邊的位元組是零。

⓭ 將 `right` 變數最左邊的位元組歸零，只是為了確保它是空的。另外請參考第 238 頁「用位元運算 AND & 來進行位元遮罩」。

⓮ 結合左邊和右邊的位元組。`left` 變數實際上是 2 個位元組（16 位元），它的最右邊的位元組（8 位元）全部都是零。另請參考第 238 頁「位元運算 | OR」。

⓯ 製作一個 `data_pdu` 類型的新變數 `pdu`。這是你稍早製作的結構類型。

⓰ 用感測器的數據填滿 `pdu` 結構。第一個參數是一個讀取的寄存器 `(accel_x_out, 0x3B)`。第二個參數是關於 `pdu` 結構。它被視為參考資料使用，這樣該函數可以自己修改結構體，而不是回傳參數值。最後一個參數是讀取的位元組數量。為了方便起見，可以使用結構的大小，以指定讀取位元組的數量。

⓱ 將感測器大端序所讀取的數字，換算成在 Arduino 使用的小端序格式。

⓲ 可以用 `structname.var` 參考結構中的變數，如 `pdu.temperature`。

⓳ 原始加速度值被換算成實際生活中的單位 g。標準重力加速度 g 等於 9.81 m/s²。該換算係數是從數據表取得。為了得到一個浮點數（小數）的結果，該分頻器必須是一個浮點數。

⓴ 將加速度列印到串列監視器。單位是 g，它是重力的加速度。1 g = 9.81m/s²。

㉑ 使用數據表的資料計算出零點，用來將原始測量值換算成攝氏溫度。該溫度在 -40° C 至 +85° C 之間。原始值 -512 代表 35° C。從這一點開始，每 1° C 的變化代表一個 340 原始值的變化量，所以要找出 0° C 該點的原始值，你可以用 -512 減去 340 * 35 的乘積（扣除 35° C 原始的測量值，每 ° C 代表 340）。該算式是 `-512 - (340 * 35)` 等於 -12412。但是，與其寫下計算出的值 -12412，你應該寫下數據表中的計算式讓該程式碼更清晰易懂。

㉒ 依據數據表的公式，將原始測量值換算成攝氏溫度。

㉓ 列印陀螺儀所測得的角速度。將原始值換算成度 / 秒，它的換算係數根據數據表是 1/131.0。為了得到一個浮點數（小數）的結果，分頻器同時也必須是浮點數。

㉔ 讀取寄存器 `point` 位元組 `size` 的函數。該結果被覆寫入該結構，它是用 `*buffer` 指標來表示。因為有了指標，該結構的實際參數值會被修改，而不只是回傳一個參數值而已。

㉕ 發送 I2C 命令到裝置（在 MPU 6050 感測器 0x69 的位址）。

㉖ 指定要讀取的寄存器位址。在這個程式中，`read_i2c()` 只是用於透過 `accel_x_out`（0x3B）讀取。

㉗ 保持接通的狀態，這樣就可以在下一行讀取數據。

㉘ 請求 `size` 位元組感測器的數據。稍早的時候，你說過你要從寄存器 `accel_x_out`（0x3B）開始作業。該 `true` 參數（第三個參數，在 Arduino 的文件中被命名為 `stop`），它代表的意思是讀取作業結束後，該連接會被關閉，進而釋放該 I2C 匯流排供稍後使用。

㉙ 為了即將到來的 `while` 迴圈宣告一個新變數。該迴圈變數 `i` 持有多少位元組被讀取的計量。該計量 `i` 等於該迴圈已執行迭代的數量。

㉚ 只有在有可讀取的位元組才能進入該迴圈，而你還沒有完成讀取所有位元組的請求。在這個程式中 `read_i2c()` 被讀取的方式，該變數的大小一定是 `data_pdu` 結構體的長度。

㉛ 讀取 1 個位元組（8 位元），並將它儲存到緩衝區裡。想想 `read_i2c()` 是如何在這個程式中被呼叫的，我們可以執行第一次迭代。該指標 `*buffer` 指向 `pdu` 第 1 個位元組，它是一個 `data_pdu` 結構類型。在第一次迭代中，`i` 是 0，因此 `buffer[i]` 指向 `pdu` 第 1 個位元組。因為 `pdu` 與一個指標被傳送到該函數，實際 `pdu` 的內容（主程式裡的變數）將被覆寫。不需要回傳，因此 `read_i2c()` 的類型是無效的。在第二次迭代中，`buffer[1]` 指向 `pdu` 第 2 個位元組。這種情況持續到整個緩衝區（`pdu`）的作業。當 `i == size`，該 `while` 迴圈不會再次進入，而繼續執行該 `while` 迴圈之後的程式碼。

㉜ 如果沒有足夠的位元組可供使用，該 loop 迴圈變數 `i` 小於 `size`。因為 `i` 是在迴圈外宣告的，在該迴圈之後你就可以使用該項功能。

㉝ 寫入 1 個位元組 `data` 到感測器的寄存器 `reg`。

㉞ 感測器的位址來自全域變數 `i2c_address`。

Raspberry Pi 的 MPU 6050 程式碼與連接

圖 8-6 為 Raspberry Pi 接線簡圖。將它連接如圖所示，然後執行範例 8-4 的程式碼。

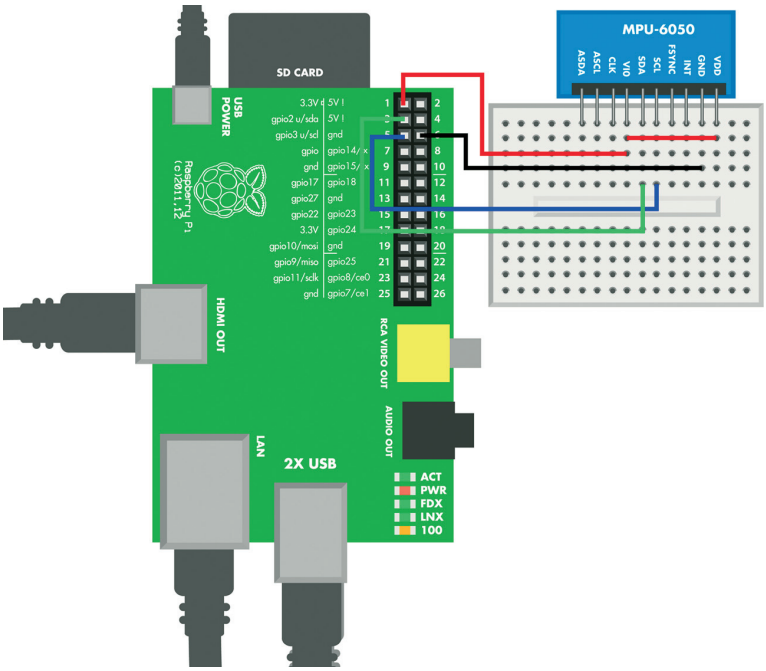

圖 8-6 Raspberry Pi 的 MPU 6050 六軸加速器電路

範例 8-4　mpu_6050.py

```
# mpu_6050.py - 列印加速度 (m/s**2) 及角速度 (gyro, deg/s)
# 版權所有 (c) BotBook.com - Karvinen, Karvinen, Valtokari
import time
import smbus # sudo apt-get -y 安裝 python-smbus  # ❶
import struct

i2c_address =  0x68    # ❷
sleep_mgmt = 0x6B      # ❸
accel_x_out = 0x3B     # ❹

bus = None      # ❺
acc_x = 0
acc_y = 0
acc_z = 0
temp = 0
gyro_x = 0
gyro_y = 0
gyro_z = 0

def initmpu():
  global bus    # ❻
  bus = smbus.SMBus(1)  # ❼
  bus.write_byte_data(i2c_address, sleep_mgmt, 0x00)    # ❽

def get_data():
  global acc_x,acc_y,acc_z,temp,gyro_x,gyro_y,gyro_z
  bus.write_byte(i2c_address, accel_x_out)      # ❾
  rawData = ""
  for i in range(14):   # ❿
    rawData += chr(bus.read_byte_data(i2c_address,accel_x_out+i))      # ⓫
  data = struct.unpack('>hhhhhhh', rawData)     # ⓬

  acc_x = data[0] / 16384.0     # ⓭
  acc_y = data[1] / 16384.0
  acc_z = data[2] / 16384.0
  zero_point = -512 - (340 * 35)        # ⓮
  temp = (data[3] - zero_point) / 340.0 # ⓯

  gyro_x = data[4] / 131.0       # ⓰
  gyro_y = data[5] / 131.0
  gyro_z = data[6] / 131.0
```

```
def main():
  initmpu()
  while True:   # ⓱
    get_data()  # ⓲
    print("DATA:")
    print("Acc (%.3f,%.3f,%.3f) g, " % (acc_x, acc_y, acc_z))   # ⓳
    print("temp %.1f C, " % temp)
    print("gyro (%.3f,%.3f,%.3f) deg/s" % (gyro_x, gyro_y, gyro_z))
    time.sleep(0.5) # 秒 # ⓴

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

❶ SMBus 建立了 I2C 業界通訊協定標準的一個子集。與 Arduino 相對應功能的程式比較起來，SMBus 的函式庫使得 Raspberry Pi 程式碼精簡許多。python-smbus 套件必須安裝在 Raspberry Pi 上（參考第 231 頁「SMBus 和不需要 Root 權限的 I2C」的說明）。

❷ MPU 6050 感測器的 I2C 位址，在數據表中找到的數字以十六進制顯示（參考第 232 頁「十六進制，二進制和其他編碼系統」）。

❸ 命令的寄存器位址。你可以透過網頁搜尋「MPU 6050 寄存器的映射（MPU 6500 register map）」來找尋寄存器的映射。

❹ 以 X 軸加速度的寄存器位址為起始位址，當作你感興趣的參數值：加速度、溫度、和角速度。

❺ 讓 `bus` 變數對所有函數而言是一個全域變數。

❻ 要修改函數中全域變數的參數值時，就必須在函數一開始的地方先指出該變數為全域變數。

❼ 初始化 SMBus（I2C）。將 SMBus 類別的新物件儲存到全域 `bus` 變數裡。

❽ MPU 6050 一開始會先進入睡眠模式。要進行任何作業之前需要先將它喚醒。發送到感測器的命令是經由 I2C（SMBus）裝置的位址、寄存器、及寫入寄存器的參數值來進行的。

❾ 請求數據，從 X 軸加速度的位址開始。

❿ 以 `i` 的參數值，從 0 到 13 重複執行 14 次。

⓫ 讀取當前的位元組，將它轉換為 ASCII 碼，並把它添加到 `rawData` 字串裡。

⓬ 將 `rawData` 字串轉換為 Python 元組（tuple）。該字串格式中的字元以小端序 `<`，含正負號的 2 個位元組（16 位元）短整數 `h` 顯示。

⓭ 將原始加速度換算成實際生活中的單位 g。標準重力 g 等於 9. 81m/s^2。

⓮ 計算溫度的零點。Raspberry Pi 處理的非常快速，並且將整個公式寫入，使得程式碼更容易閱讀並減少拼寫錯誤的機率 。

⓯ 將原始溫度轉換為攝氏溫度。為了得到一個浮點數的結果，分頻器也必須是浮點數。換算公式是根據數據表（Google「MPU 6050 的數據表（MPU 6050 data sheet）」）。

⓰ 將原始角速度換算為實際生活中的單位（每秒的度數）。

⓱ 該程式持續執行，直到你按下 Control-C 為止。

⓲ `get_data()` 會更新全域變數，所以它不需要從函數回報參數值。

⓳ 使用字串格式列印加速度。該 `%.3f` 替換值代表小數點後三位的浮點值。

⓴ 為了讓用戶讀取列印值，及避免占用 100% CPU 的時間，我們在此添加了一個小小的延遲。

找出走動、跑步、跳躍、或滑步對讀取值的影響。以及抽動、或蠕動的影響又是怎麼一回事呢？

SMBus 和不需要 Root 權限的 I2C

Raspberry Pi 程式碼使用了 I2C 的 Python smbus 的函式庫。幸運的是，在 Linux 安裝軟體是一件輕而易舉的事。你可以在 Raspbian 安裝任何軟體就像你在 Debian、Ubuntu、或 Mint 安裝任何軟體的道理是一樣的。雙擊 Raspbian 桌面左側的 LXTerminal 圖標。然後：

```
$ sudo apt-get update
$ sudo apt-get install python-smbus
```

要啟用 I2C 支援，你需要開啟 I2C 模組。首先，確保它們沒被關閉。用 `sudoedit /etc/modprobe.d/raspi-blacklist.conf` 命令來編輯 */etc/modprobe.d/raspi-blacklist.conf* 並且刪除這一行程式：

```
blacklist i2c-bcm2708
```

儲存該檔案：按下 Ctrl-X，輸入 y，然後按 Enter 鍵或 Return 鍵。

接下來，使用命令 `sudoedit /etc/modules` 來編輯 */etc/modules* 並添加這兩行程式：

```
i2c-bcm2708
i2c-dev
```

儲存檔案：按下 Ctrl-X，輸入 y，然後按 Enter 鍵或 Return 鍵。

要使用 I2C 不需要 root 權限，製作 udev 規則的檔案 *99 i2c.rules*（如範例 8-5 所示），並將它放在定位。（為了避免打字和不可抗力的錯別字產生，你可以從 *http://botbook.com* 下載 *99-i2c.rules* 檔案。）

```
$ sudo cp 99-i2c.rules /etc/udev/rules.d/99-i2c.rules
```

範例 8-5　99-i2c.rules

```
# /etc/udev/rules.d/99-i2c.rules - 在 Raspberry Pi 不需要 root 授權的 I2C
# 版權所有 2013 http://BotBook.com

SUBSYSTEM=="i2c-dev", MODE="0666"
```

重新啟動 Raspberry Pi，開啟 LXTerminal，並確認你可以看到 I2C 裝置和以及使用的權限正確無誤：

```
$ ls -l /dev/i2c*
```

該列表應顯示兩個檔案，它們應該列出 `crw-rw-rwT` 的權限。如果沒有，重覆執行上述的步驟。

十六進制、二進制和其他編碼系統

同樣的數字可以用多種方式來表示。例如，十進制的數字 65 等於十六進制的 0x41 及二進制的 0b1000001。也許你最熟悉的是一般十進制，其中 5+5 等於 10。

不同的表達方式都會在數字前添加一個前綴。一般的十進制數字沒有使用前綴。十六進制的數字是以 `0x` 為開頭、二進制數字以 `0b` 為開頭、及八進制數字以 `0` 為開頭。

不同的編碼系統在表 8-2 進行比較。

表 8-2　數字的表達方式

前綴	表示方式	底數	使用	範例	算式
	十進制	10	一般的系統	10	$0*10^0 + 1*10^1$
0x	十六進制	16	C 和 C++ 程式碼，數據表	0xA	$10*16^0$
0b	二進制	2	低階通訊協定，位元脈衝	0b1010	$0*2^0 + 1*2^1 + 0*2^2 + 1*2^3$
0	八進制	8	Linux 系統的 Chmod 授權	012	$2*8^0 + 1*8^1$

以數字 42 為例。在任何編碼系統裡它是一個完全一樣的號碼，所以

```
42 = 0x2a = 0b101010 = 052
```

只有底數發生變化。使用熟悉、一般的十進位制，它是以 10 為底，從右側開始先數個位數，然後再數十位數。

```
2*1 + 4*10 = 42
```

如果有一個很大的數字，例如 1917，十位數顯而易見的。個位數之後，就是十位數（10）、百位數（10*10）和千位數（10*10*10）。你可以輕鬆地用指數來表達這些數字：

$10*10 = 10^2$
$10*10*10 = 10^3$

那麼數字 1 如何表示呢？任何數字的零次方都是 1，所以它表達方式是：

$10^0 = 1$

因此，數字 42 變為：

$42 = 4*10^1 + 2*10^0$

42 的十六進制的表達方式是 0x2A。在十六進制式裡，大於 9 的數字用英文字母來表示：A=10，B=11 ... F=15。從右側開始，請注意，A 代表 10 以此類推：

```
0x2A = 10*1 + 2*16 = 10 + 32 = 42
```

使用指數來表達，可以寫成

$10*16^0 + 2*16^1 = 0x2A$

嘗試一些其他的數字。要檢查你的計算式，使用 Python 控制台（請參考第 235 頁「Python 的控制台」或表 8-3）。你也可以運用你的技巧將該數字換算成二進制嗎？

你可以在 Python 控制台練習數字的換算。數字（1 == 0x1 == 0b1）的表達方式在 Python、C、和 C++ 不同語言裡都是一樣的。你可以在控制台裡執行任何 Python 的命令：

```
>>> print("Botbook.com")
Botbook.com
>>> 2+2
4
```

你輸入的任何數字將換算為十進制系統來顯示：

```
>>> 0x42
66
>>> 66
66
>>> 0b1000010
66
```

有函數可以將任何數字換算成二進制、十六進制、八進制、和 ASCII 字元：

```
>>> bin(3)
'0b11'
>>> hex(10)
'0xa'
>>> oct(8)
'010'
>>> chr(0x42)
'B'
```

表 8-3 不同數字表達方式的範例

十進制	十六進制	二進制	八進制	ASCII 碼
0	0x0	0b0	0	*\0* NUL，終止字串
1	0x1	0b1	01	
2	0x2	0b10	02	
3	0x3	0b11	03	
4	0x4	0b100	04	EOT，文字的結尾，在 Linux 用 CTRL-D
5	0x5	0b101	05	
6	0x6	0b110	06	

十進制	十六進制	二進制	八進制	ASCII 碼
7	0x7	0b111	07	
8	0x8	0b1000	010	
9	0x9	0b1001	011	
10	0xA	0b1010	012	*\n* 換行
11	0xB	0b1011	013	
16	0x10	0b10000	020	
17	0x11	0b10001	021	
32	0x20	0b100000	040	' ' 空格，第一個可列印的字元
48	0X30	0b110000	060	*0* ASCII 碼的零不等於數字的零
65	0×41	0b1000001	0101	*A*
97	0x61	0b1100001	0141	*a*
126	0x7e	0b1111110	0176	~ 波浪符號是最後一個可列印的 ASCII 字元

在終端機上輸入 `man ascii` 命令，該操作手冊頁面在 Linux 或 OS X 中顯示出一個 ASCII 圖表，例如，你可以看到 ASCII 的 A 字元是十進制 65、而十六進制則是 0x41、及八進制是 0101 的數字。手冊頁面會顯示在 less 功能的選項裡，當你按空格鍵時向前移動一個頁面，按 b 鍵時向後移動一個頁面，而按 q 鍵時表示退出。

Python 的控制台

在終端機裡輸入 *python* 命令會啟動 Python 的控制台。在 Linux 平台，終端機（bash, shell）在主選單的底部可以找到：「Accessories → Terminal」，或在終端機裡搜尋 Dash 關鍵字。在 Macintosh 平台，終端機是 /Applications/Utilities/Terminal。在 Windows 平台，你就可以開啟終端機「Applications → Accessories → Command Prompt」或從開始選單中打開 IDLE（如果你已經安裝了 Python）。

Python 的提示符號 >>> 讓你知道，你的命令將被解讀為 Python 的程式碼。要結束 Python 的事件，請輸入 Python 的命令 `exit()`。

如果你想要有設計精美、能提供互動式推測輔助選項的 Python 控制台，試試 *ipython*。

位元運算

有些感測器以組成一個位元組的一大群位元為單位來發送數據，如 *01010000*。要使用二進制方式來表達數據，你必須執行位元運算，在此你可以同時操控一群位元。

位元可以是 1 或是 0。一個位元可以代表一個真（true）值，0-false 或 1-true。

1 個位元組等於 8 位元，例如 0b1010100。1 個位元組可以代表一個 ASCII 字元，例如 T、3、或 X。它也可以代表一個數字，例如 256、或 -127。

位元運算是非常低階的函數（你幾乎與電腦記憶體最基礎的格式一同作業），想要閱讀使用該功能的程式碼，基本上是有難度的，所以你應該僅在需要時再求助於它。而另一方面，有時候只是想讓該零組件能正常作業，我們有時候會遇到要求用到二進制運算的感測器。MPU 6050 整合動作裝置就是這一類型的感測器。

最常見的二進制數學運算是位元位移和二進制布林數運算。

為了取得使用位元運算的信心，在 Python 的控制台上練習（第 235 頁「Python 的控制台」）。位元運算的作業在 C 和 C++（Arduino）的方式是一樣的，但這些語言沒有互動控制台，因此在 Python 的作業更方便。

你可能已經學過一般的布林數代數。「AND」表示兩個條件必須為真，才能讓整個表達式被認為是真（true）。「OR」是指任一條件可能為真，整個表達式則為真（true）。

```
>>> if (True): print "Yes"
...
Yes
>>> if (False): print "Yes"
...     # nothing printed
>>> if (True and True): print "Yes"
...
Yes
```

在 Python 中，真（True）與假（False）都必須將第一個英文字母大寫。

即使沒有「if」的語句，邏輯真（True）還是存在：

```
>>> True
True
>>> False
False
>>> True and True
True
```

在大多數的語言中，1 代表真（True）而 0 代表假（False）：

```
>>> 1 and 1
1
>>> 1 and 0
0
```

使用位元運算，你可以在每個位元組的 1 與 0，用同樣的作法。

> *在 Python 裡的位元運算時，你不能使用英文單字「and」或「or」。而是使用字元&（位元運算 AND）和字元|（位元運算 OR）來代替。*

```
>>> 0b0101 & 0b0110
4
```

或許該解答用二進制更容易閱讀：

```
>>> bin(0b101 & 0b110)
'0b100'
```

位元運算 AND（&），只是採用每一個位元，及每一對使用布林數 AND 的作法：

```
    0b0101
    0b0110
==========
&   0b0100
```

從左邊開始，依序是 0-false 和 0-false 代表 0-false。1-truth 和 1-truth 代表 1-truth。0-false 和 1-truth 代表 0-false。1-truth 和 0-false 代表 0-false。

用位元運算 AND & 來進行位元遮罩

位元遮罩讓你可以得到你想要的位元。例如，你可以用下列的作業得到四個最左邊的數字 0b 1010 1010：

```
>>> bin(0b10101010 & 0b11110000)
'0b10100000'
```

以下是它的作業方式：

```
0b 1010 1010    #  一個數字
0b 1111 0000    #  位元遮罩
==========================
0b 1010 0000    # & (位元運算 AND)
```

只有當兩個輸入都為真（true）的時候 AND 代表真（表 8-4）。

表 8-4　AND 真值表

a	b	a AND b
0	0	0
1	0	0
0	1	0
1	1	1

位元運算 OR |

如果分別製作左半側和右半側的位元組，你如何把它們放回一塊兒呢？在 Python 裡，你可以使用位元運算 OR（|）：

```
>>> left=0b10100000
>>> right=0b1111
>>> bin(left|right)
'0b10101111'
```

讓我們詳細觀察位元的組成：

```
0b 1010 0000    # 左側，右側必須有零
0b      1111    # 右側，左側有沒有零都無礙
========================
0b 1010 1111    # 位元運算 OR |
```

位元運算 OR「|」與加號「+」是不一樣的。例如，考慮使用 0b1 + 0b1。

位元移位 <<

位元移位是將位元移動到右側或左側。你可以在 Python 嘗試：

```
>>> bin(0b0011<<1)
'0b110'
>>> bin(0b0011<<2)
'0b1100'
```

將位元移動到右側時，會將多出的位元刪除：

```
>>> bin(0b0011>>2)
'0b0'
```

位元端序，通常在小端序

大多數硬體都是小端序（*little endian*），就像你已經習慣每天都要使用的數字一樣。一般的數字都是小端序：在 1991 這個數字裡，千位數排在前面，接下來是百位數、十位數、最後是個位數。最重要的數字優先表示。

大多數的電腦也都是小端序，因此最重要的位元組優先表示。你的電腦工作站和筆記型電腦有可能在 x86、amd64、或 x86_64 的電路板架構下執行，因此它們都是小端序。Arduino 是以 Atmel AVR 和它的小端序為基本架構。Raspberry Pi 和 Android Linux（智慧型手機平台的領頭羊）兩個作業系統，都是以小端序模式在 ARM 運行。

135 這個數字可以儲存為小端序（標準值）或大端序：

```
0b 1000 0111    # 小端序，標準值，Arduino，Raspberry Pi，電腦工作站
0b 0111 1000    # 大端序，MPU 6050
```

有些裝置有非典型的位元組序。MPU 6050 感測器是大端序，這代表最重要的位元組放在最後。當它與 Arduino 和 MPU 6050 通訊時，該位元組序必須在小端序與大端序之間互換。

只有當執行類似位元和位元組的低階作業時位元組序才會有影響。在高階編程作業裡，你可以在浮點變數設定一個參數值，並且取回一個浮點數。在更高階的編程作業裡，該環境會替你處理位元組序及其他的細項。

實驗：破解 Wii Nunchuk（用 I2C）

你想要在一個價廉的組合裡擁有一個加速器、一支搖桿、和一個按鈕？不必再到其他地方找尋了，因為 Wii 遊戲機的 Nunchuk 的控制器具備了上述所有的配備。如果你還想要購買更便宜的裝置，也有一些便宜的共容裝置可供使用。

而 Wii Nunchuk 也給我們上了一堂重要的破解課題：工程師和大家一樣都是凡人。就像我們所有的人一樣，工程師都希望用測試好而且是貨真價實的通訊協定，而且還提供了函式庫和工具可供使用。Wii 有它自己專用的連接器，而且是原創、提供數量極少的文件資料。但實際上，它只是一個標準的 I2C 通訊協定。正如先前所看到的（第 221 頁「業界標準的通訊協定」，I2C 是我們最喜歡的業界標準的短程通訊協定。

任天堂所生產大批量的 Wii Nunchuk，維持了高品質和低價格。Nunchuk 的普及性也代表著有大量的範例程式碼和文件資料可供使用。你甚至都不需要裁剪導線，因為它還提供了 WiiChuck 轉接器讓你可以直接插接 Nunchuk 的連接器，將它連接到 Arduino 或麵包板（如圖 8-7 所示）。

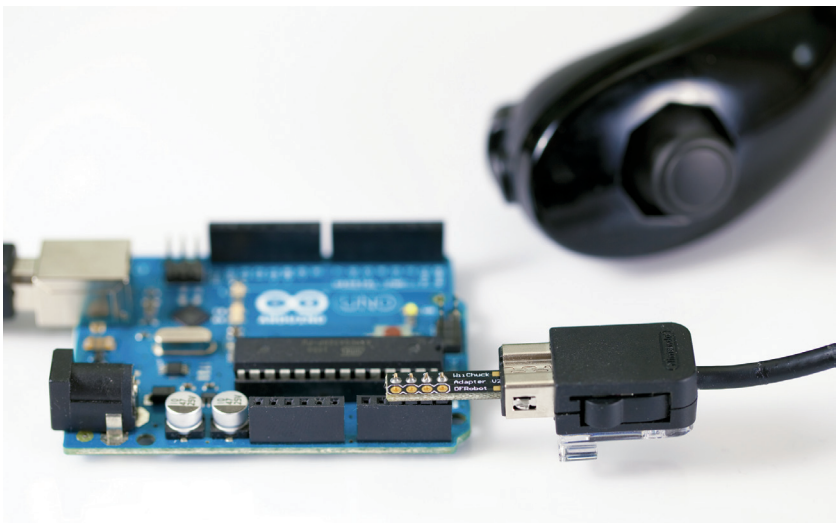

圖 8-7　用 WiiChuck 轉接器將 Nunchuk 與 Arduino 連接

Wii Nunchuk 可以使用 SMBus 的通訊協定來進行通訊。這是一個簡化的通訊協定，因為它是業界標準 I2C 通訊協定的一個子集，這使得它應有的溝通作業方式更加明確。

Arduino 的 Nunchuk 程式碼與連接

圖 8-8 為 Arduino 的接線簡圖。如圖所示將它連接，並執行範例 8-6 所示的 sketch 程式。

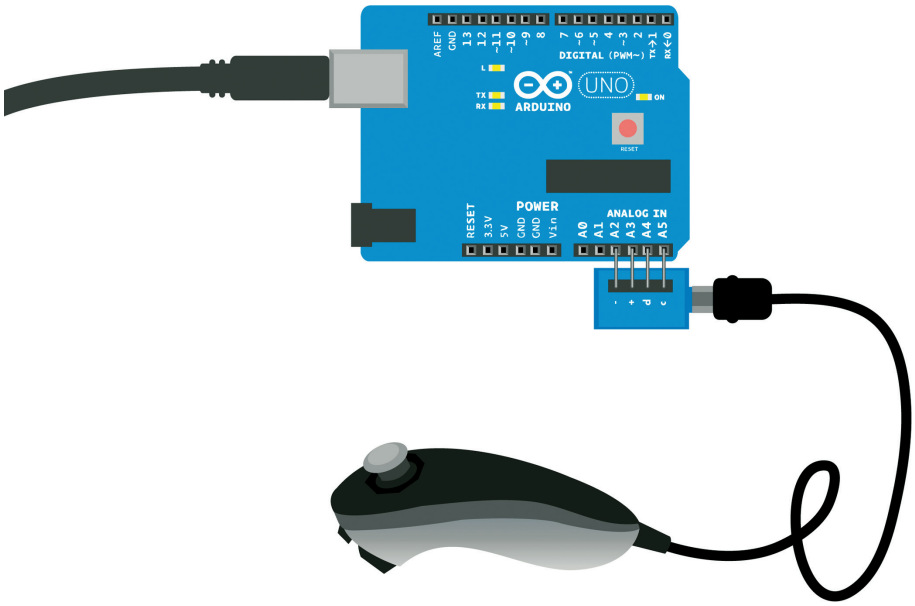

圖 8-8　Arduino 的 WiiChuck 電路

範例 8-6　wiichuck_adapter.ino

```
// wiichuck_adapter.ino - 列印搖桿、加速器和按鈕的數據到串列埠
// (c) 版權所有  BotBook.com - Karvinen, Karvinen, Valtokari


#包含 <Wire.h>

const char i2c_address = 0x52;

unsigned long lastGet=0; // 毫秒
int jx = 0, jy = 0, accX = 0, accY = 0, accZ = 0, buttonZ = 0, buttonC = 0;  // ❶

void setup() {
  Serial.begin(115200);
  Wire.begin();
  pinMode(A2, OUTPUT);
  pinMode(A3, OUTPUT);
```

```
  digitalWrite(A2, LOW);       // ❷
  digitalWrite(A3, HIGH);      // ❸
  delay(100);
  initNunchuck();      // ❹

}

void loop() {
  if(millis() - lastGet > 100) {      // ❺
    get_data();  // ❻
    lastGet = millis(); // ❼
  }
  Serial.print("Button Z: ");
  Serial.print(buttonZ);        // ❽
  Serial.print(" Button C: ");
  Serial.print(buttonC);
  Serial.print(" Joystick: (x,y) (");
  Serial.print(jx);     // ❾
  Serial.print(",");
  Serial.print(jy);
  Serial.print(") Acceleration (x,y,z) (");
  Serial.print(accX);   // ❿
  Serial.print(",");
  Serial.print(accY);
  Serial.print(",");
  Serial.print(accZ);
  Serial.println(")");

  delay(10); // 毫秒
}

void get_data() {
  int buffer[6]; // ⓫
  Wire.requestFrom(i2c_address, 6);     // ⓬
  int i = 0;    // ⓭
  while(Wire.available()) {     // ⓮
    buffer[i] = Wire.read();    // ⓯
    buffer[i] ^= 0x17;  // ⓰
    buffer[i] += 0x17;  // ⓱
    i++;
  }
  if(i != 6) {  // ⓲
    Serial.println("Error reading from i2c");
  }
  write_i2c_zero();     // ⓳
```

```
  buttonZ = buffer[5] & 0x01;   // ⓴
  buttonC = (buffer[5] >> 1) & 0x01;    // ㉑
  jx = buffer[0];       // ㉒
  jy = buffer[1];
  accX = buffer[2];
  accY = buffer[3];
  accZ = buffer[4];
}

void write_i2c_zero() {
  Wire.beginTransmission(i2c_address);
  Wire.write(0x00);
  Wire.endTransmission();
}

void initNunchuck()
{
  Wire.beginTransmission(i2c_address);
  Wire.write(0x40);
  Wire.write(0x00);
  Wire.endTransmission();
}
```

❶ 宣告全域變數。因為 Arduino 的函數無法輕易回傳多個參數值，數據是透過全域變數的使用，從一個函數傳送到另一個函數。

❷ 將類比引腳 A2 當作接地（GND，0 V，低電平）。透過這種方式，WiiChuck 可以無需任何麵包板或跳線連接 Arduino 引腳。

❸ 將 A3 視為 +5 V 電源（高電平，正極，VCC）。

❹ 透過呼叫 `initNunchuck()` 來初始化 Wii Nunchuk，它會透過 I2C 發送 0x40 和 0x00。

❺ 每隔 100 毫秒讀取數據。這是一種常見的每隔 *x* 毫秒執行某個事件的程式模式。該 `millis()` 函數會以毫秒為單位回傳 Arduino 的正常工作時間（它被開啟的時間有多長）。該 `lastGet` 變數，從最後一次 `get_data()` 被呼叫所持有的時間，此正常工作時間的單位為毫秒。（在第一次迭代中，`lastGet` 是 0）。`millis()` 和 `lastGet` 之間的時間差異是從 `get_data()` 最後一次被呼叫所經歷的時間。如果所經歷的時間大於 100 毫秒，Arduino 會執行以下的程式區塊。

❻ 由於 `get_data()` 會更新全域變數，所以它並不需要回傳一個參數值。

❼ 更新最後一次呼叫的時間到 `get_data()`。

❽ 按鈕的狀態是 0 - 向下，1 - 向上。

❾ 這兩個搖桿的軸（`jx`、`jy`）是一個介於 30 到 220 的原始值。

❿ 所有加速器的軸（`accX`、`accY`、`accZ`）都是介於 80 到 190 的原始值。

⓫ 宣告一個六個整數的新矩陣。

⓬ 從 Nunchuk 要求六個位元組的數據。

⓭ 該迴圈變數 `i` 將包含被處理過的位元組數字。

⓮ 只有當位元組需要被讀取時，才會進入該迴圈。

⓯ 讀取 1 個位元組並將它儲存到當前 `buffer[]` 裡的儲存格。在第一迭代時是 `buffer[0]`。在最後一次迭代時是 `buffer[5]`。

⓰ 用 0x17 以當前的參數值執行 XOR，並且用該結果取代當前的參數值（這就是所謂的*原地*（*in place*）運算）。XOR 是一個類似 OR 的布林數作業，但它只有在 a 或 b 任何一個是真時它才是真（true），而不是在兩者都是真（true）的時候。

⓱ 添加 0x17 到當前的參數值。

⓲ 如果讀取的位元組數字不是 6 時，列印一個警告訊號。

⓳ 透過 I2C 發送 0x00 來要求另一個讀取作業。

⓴ 取得最後一個位元組的最後一個位元。最後 1 個位元組是 `buffer[5]`。最後一個被位元遮罩提取的位元；該程式碼對 ob 0000 0000 0000 0001 在位元組執行位元運算 AND 的作業。參考第 238 頁「用位元運算 AND & 來進行位元遮罩」及表 8-5。按鈕狀態是 0 - 向下，1 - 向上。

㉑ 取得最後 1 個位元組的倒數第二個位元。倒數第二個位元用位元移位 `>> 1` 被移到最後一個位元，接著最後一個位元如同在前一行命令一樣被提取。 參考第 239 頁「位元移位 <<」 。

㉒ `jx` 搖桿的 x 軸就是一個完整的位元組。搖桿和加速器剩餘部分的軸以類似的方式被讀取。

表 8-5　Nunchuk 6 位元組的數據區塊

位元組	使用
0	搖桿 X
1	搖桿 Y
2	加速器 X
3	加速器 Y
4	加速器 Z
5	ZCxxyyzz：按鈕「Z」和「C」，精密的加速器

Raspberry Pi 的 Nunchuk 程式碼與連接

圖 8-9 為 Raspberry Pi 接線簡圖。替它接線，並執行範例 8-7 所示的程式碼。

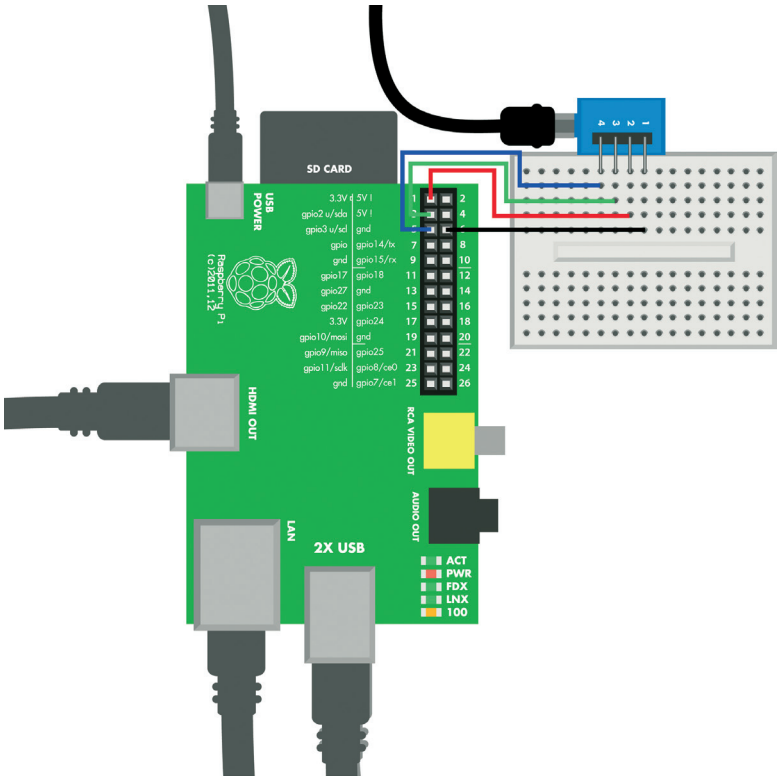

圖 8-9　Raspberry Pi 的 WiiChuck 電路簡圖

範例 8-7 wiichuck_adapter.py

```
# wiichuck_adapter.py - 列印 Wii Nunchuck 的加速器及搖桿
# 版權所有 BotBook.com - Karvinen, Karvinen, Valtokari

import time
import smbus # sudo apt-get -y 安裝 python-smbus    # ❶

bus = None
address = 0x52    # ❷

z = 0    # ❸
c = 0
joystick_x = 0
joystick_y = 0
ax_x = 0
ax_y = 0
ax_z = 0

def initNunchuck():
    global bus
    bus = smbus.SMBus(1) # ❹
    bus.write_byte_data(address, 0x40, 0x00)    # ❺

def send_request():
    bus.write_byte(address, 0x00)    # ❻

def get_data():
    global bus, z, c, joystick_x, joystick_y, ax_x, ax_y, ax_z
    data = [0]*6
    for i in range(len(data)): # ❼
        data[i] = bus.read_byte(address)
        data[i] ^= 0x17
        data[i] += 0x17


    z = data[5] & 0x01    # ❽
    c = (data[5] >> 1) & 0x01    # ❾

    joystick_x = data[0]
    joystick_y = data[1]
    ax_x = data[2]
    ax_y = data[3]
    ax_z = data[4]
    send_request()
```

```
def main():
    initNunchuck()
    while True:
        get_data()
        print("Button Z: %d Button C: %d joy (x,y) (%d,%d) \
               acceleration (x,y,z) (%d,%d,%d)" \
               % (z,c,joystick_x,joystick_y,ax_x, ax_y, ax_z))
        time.sleep(0.1)


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

❶ python-smbus 函式庫必須安裝在 Raspberry Pi（參考第 231 頁「SMBus 和不需要 Root 權限的 I2C」）。

❷ Wii nunchuk 的位址。該參數值以十六進制顯示。請參考第 232 頁「十六進制、二進制、和其他編碼系統」有關十六進制參數值更詳細的說明。

❸ 其中一個按鈕的全域變數。

❹ 製作一個 `SMBus` 類別的新物件，並將它儲存到新的變數 `bus`。`SMBus()` 建構子有一個參數，該裝置的編號。數字 1 代表檔案 `/dev/i2c-1`。這個裝置編號在新的 Raspberry Pi 電路板而言是很常見的。如果你有一個舊款的 Raspberry Pi 第一版的電路板，你可能需要在 `/dev/i2c-0` 使用數字 0 來代替。

❺ Nunchuk 必須用 I2C 命令來初始化。`bus.write_byte_data(addr=0x52, cmd=0x40, val=0x00)` 用參數值 0x00 發送 0x40 命令給 Nunchuk。

❻ 你可以用標準字元 0x00 詢問 Nunchuk 的下一套參數值，這相當於是 *\0* 或單純只是 0 而已。

❼ 讀取 6 個位元組的數據。這包含在表 8-5 中所描述的數據區塊。剩餘部分的函數將對該數據進行解碼。

❽ 取得「Z」鍵的狀態：1 代表向下，0 代表向上。該位元遮罩只是一個位元，`0b1`。當使用位元運算 AND（`&`）時，它是最右側的位元，該位元左側的任何位元都被視為零。因此，使用 `0b1` 與位元運算 AND，只會回傳最右側的位元。另請參考第 238 頁「用位元運算 AND & 來進行位元遮罩」。

❾ 取得「C」按鈕的狀態，1 - 向下或 0 - 向上。要取得倒數第二個位元，將倒數第二個位元移到最後的位置（`x >> 1`），並且用一個位元遮罩只取得最後一個位元。

測試項目：用 Wii Nunchuk 控制機械手臂

用 Nunchuk 來控制機械手臂。因為你已經可以讀取加速度和搖桿位置，你可以根據這些數字輕鬆地轉動伺服機。添加機械裝置，你就可以擁有一個透過 Nunchuk 控制的機械手臂。

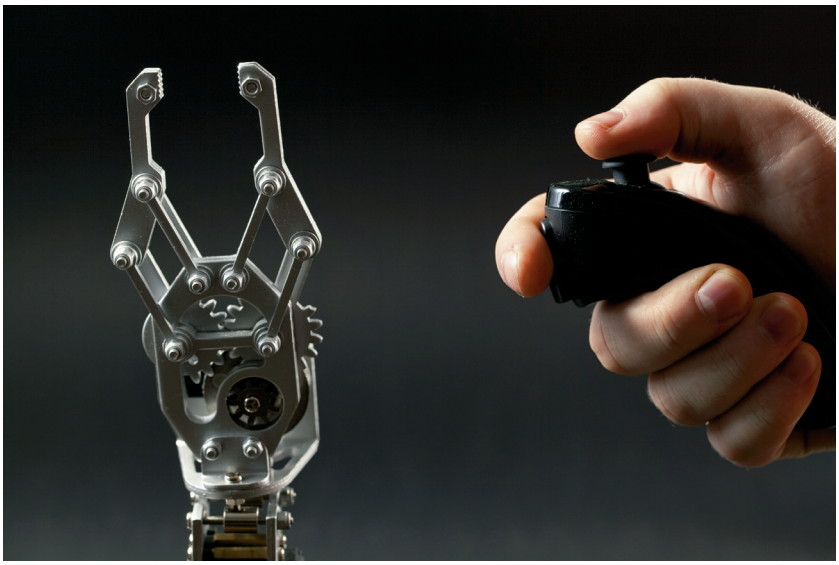

圖 8-10　Nunchuk 控制的機器手臂

你將學習到的內容

在機械手臂的實驗項目裡，你將學習如何：

- 使用加速器和附帶輸出功能的機械搖桿。
- 結合伺服機來完成複雜的動作。

你還可以更新你對伺服機控制的技術及用移動平均值來過濾雜訊（參考第 124 頁「伺服馬達」和第 199 頁「移動平均值」）。

一開始只用到伺服機（如圖 8-11 所示）。一旦你取得一些動作之後，你可以繼續使用手部機械式的動作。

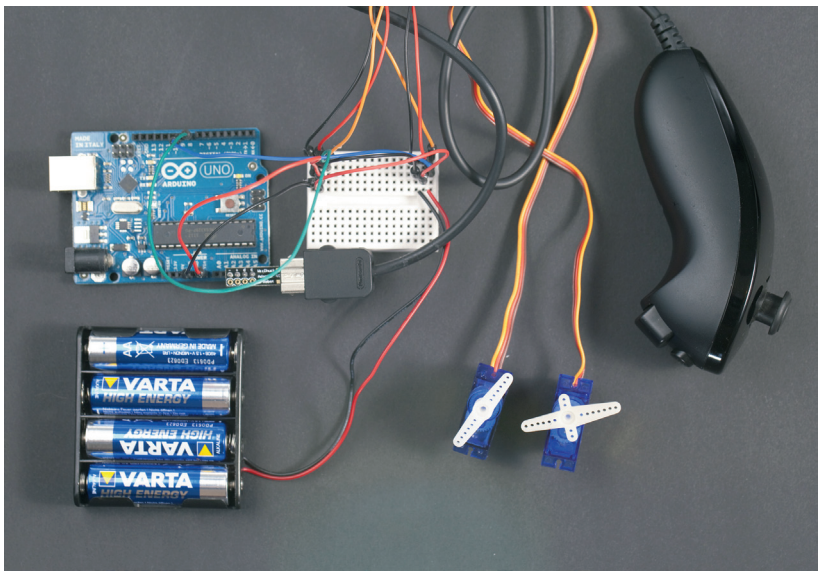

圖 8-11　Nunchuk 使用 Arduino 來控制兩個伺服機

如圖 8-12 所示為接線簡圖。按照圖示將它連接，然後執行範例 8-8 所示的程式碼。

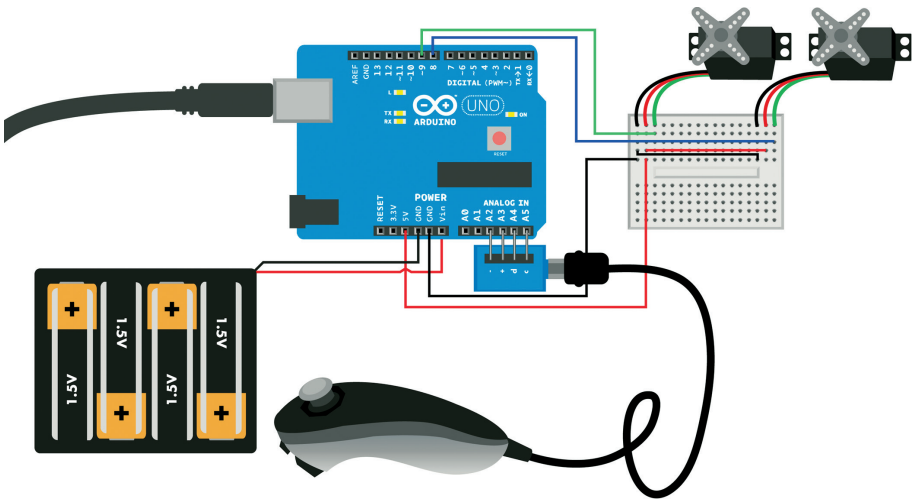

圖 8-12　Arduino 的夾具電路

有關 Wii Nunchuk 與 Arduino 的更多應用，請參考第 241 頁「Arduino 的 Nunchuk 程式碼與連接」。

範例 8-8 wiichuck_adapter_claw.ino

```
// wiichuck_adapter_claw.ino - 用 Nunchuck 控制機器手臂
// 版權所有 (c) BotBook.com - Karvinen, Karvinen, Valtokari

#包含 <Wire.h>

const int clawPin = 8;
const int armPin = 9;
int armPos=0, clawPos=0;
float wiiP = 0.0;         // ❶
float wiiPAvg = 0.0;      // ❷
int lastarmPos = 350;

const char i2c_address = 0x52;
int jx = 0, jy = 0, accX = 0, accY = 0, accZ = 0, buttonZ = 0, buttonC = 0;

void setup() {
  Serial.begin(115200);

  // Nunchuck
  Wire.begin();
  pinMode(A2, OUTPUT);
  pinMode(A3, OUTPUT);
  digitalWrite(A2, LOW);
  digitalWrite(A3, HIGH);
  delay(100);
  initNunchuck();

  // 伺服機
  pinMode(clawPin, OUTPUT);
  pinMode(armPin, OUTPUT);
}

void loop() {
  get_data();   // ❸

  wiiP = (accZ-70.0)/(178.0-70.0);     // ❹
  if (accY>120 && accZ>100) wiiP=1;
  if (accY>120 && accZ<100) wiiP=0;
  if (wiiP>1) wiiP=1;
  if (wiiP<0) wiiP=0;
  wiiPAvg = runningAvg(wiiP, wiiPAvg);  // ❺
  armPos = map(wiiPAvg*10*1000, 0, 10*1000, 2200, 350);
```

```
  clawPos = map(jy, 30, 220, 1600, 2250);       // ❻

  pulseServo(armPin, armPos);   // ❼
  pulseServo(clawPin, clawPos);

  printDebug();
}

float runningAvg(float current, float old) {
  float newWeight=0.3;
  return newWeight*current + (1-newWeight)*old; // ❽
}

// 伺服機

void pulseServo(int servoPin, int pulseLenUs)   // ❾
{
  digitalWrite(servoPin, HIGH);
  delayMicroseconds(pulseLenUs);
  digitalWrite(servoPin, LOW);
  delay(15);
}

// i2c

void get_data() {
  int buffer[6];
  Wire.requestFrom(i2c_address,6);
  int i = 0;
  while(Wire.available()) {
    buffer[i] = Wire.read();
    buffer[i] ^= 0x17;
    buffer[i] += 0x17;
    i++;
  }
  if(i != 6) {
    Serial.println("Error reading from i2c");
  }
  write_i2c_zero();

  buttonZ = buffer[5] & 0x01;
  buttonC = (buffer[5] >> 1) & 0x01;
  jx = buffer[0];
  jy = buffer[1];
  accX = buffer[2];
```

```
  accY = buffer[3];
  accZ = buffer[4];

}

void write_i2c_zero() {
  Wire.beginTransmission(i2c_address);
  Wire.write((byte)0x00);
  Wire.endTransmission();
}

void initNunchuck()
{
  Wire.beginTransmission(i2c_address);
  Wire.write((byte)0x40);
  Wire.write((byte)0x00);
  Wire.endTransmission();
}

// 除錯

void printDebug()
{
  Serial.print("accZ:");
  Serial.print(accZ);
  Serial.print("        wiiP:");
  Serial.print(wiiP);
  Serial.print("        wiiPAvg:");
  Serial.print(wiiPAvg);
  Serial.print("        jy:");
  Serial.print(jy);
  Serial.print("        clawPos:");
  Serial.println(clawPos);
}
```

❶ `WiiP` 持有 Wii 的傾斜度是一個百分比的值。向後是 0.0 及向前是 1.0。

❷ `WiiP` 的移動平均值。

❸ 透過 i2c 讀取 Wii Nunchuk 時兩個讀取作業之間需要 20 毫秒的延遲。稍後在 `loop()` 迴圈裡兩次對 `pulseServo()` 的呼叫提供了這個延遲。

❹ 計算 Wii 的原始讀取值的百分比。這個公式與內建的 `map()` 函數類似。

❺ 過濾隨機尖峰值，對 z- 軸的加速度使用最後兩個樣本的平均值。

❻ 取得原始搖桿的參數值（30 至 220），並將它映射到伺服機脈衝（1500 至 2400）。例如，搖桿的參數值 30 映射到的伺服脈衝長度為 1500 微秒。

❼ 發送一個脈衝到使用伺服控制的機器手臂。為了讓伺服機持續旋轉，你必須持續發送這些脈衝串流到該伺服機。

❽ 為了在多個參數值裡取得一個移動平均值，你將使用加權平均值。這樣一來，只需要儲存一個先前的數據點，但是舊的參數值仍然可以影響該平均值。

❾ 發送一個脈衝給伺服機。為了有效地控制伺服機，該函數應該每秒被呼叫 50 次。請參考第 124 頁「伺服馬達」。

添加機械手的動作

你已經知道如何用 Wii Nunchuk 來控制伺服馬達，而且要將這些機器人手臂和手掌商業化並不困難。有許多可用的選擇，如果你有很紮實的機械技術，你甚至可以製作自己的裝置。我們的機器人手臂是從 *http://dx.com* 購買的。

無論你選擇的是手臂或是手掌，將它安裝在一個穩固的基座上不失為是一個好主意，它可以讓該裝置保持直立，防止其傾倒的現象。我們的機械手臂選擇用厚木料當作基座並且用螺絲固定（如圖 8-13 所示）。

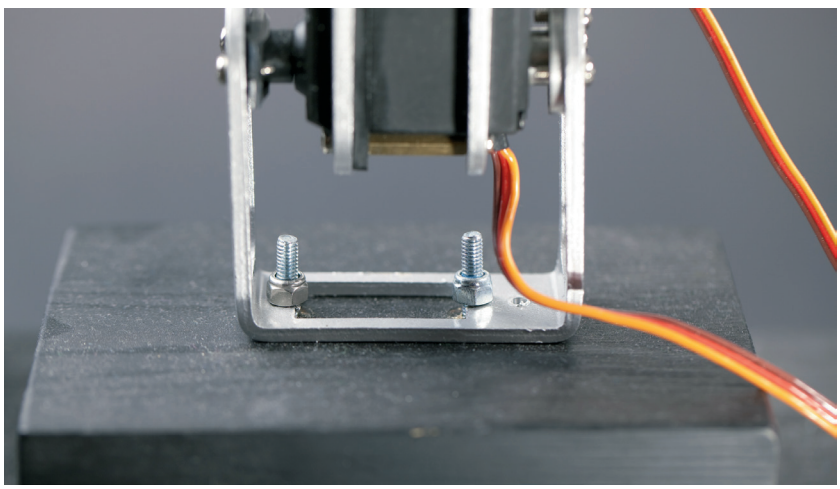

圖 8-13　機器人手臂固定在基座

所有的程式碼已經完成了（如範例 8-8 所示）。將伺服機連接，讓你可以用 Nunchuk 的搖桿和手臂運動的加速器來控制夾具。如圖 8-14 所示為最終的結果。

現在，你可以測量加速度和角速度。你知道該裝置的確實方位及它的旋轉方向。如果你的實驗項目有此需求，你可以只測量其中一項，或者用整合移動裝置來測量兩個訊號。

使用任何一種感測器時，你可能已經注意到，可以將程式碼範例以選單的方式選用。但是，如果你選擇較難的方式，並學會位元運算來了解該程式碼是如何運作的，你可以給自己一個獎勵。如果你的實驗項目需要用到一個新的、高難度的感測器而沒有範例程式碼可供使用時，這些技術就可以適時派上用場。

圖 8-14　用 Wii 控制的機器人手臂

或許你可以停止把玩你的機器手臂實驗項目一段時間，因為現在有些事情需要立即確認。在下一章節中，你將學習如何讀取指紋和 RFID 標籤。