目　　录

项目 - 音乐蜂鸣器

在之前的课程中，我们学会了如何控制蜂鸣器发出不同的声音。那么，我们是否可以通过调整声音的频率和节拍，来组合出一段完整的旋律呢？在这个项目中，我们将使用蜂鸣器演奏音乐，并学习如何创建头文件（.h 文件），以导入别人已经编写好的“曲谱”。

在使用 Arduino IDE 编程时，每创建一个项目文件被称为一个“Sketch”。一个 Sketch 中可以包含多个代码文件。为了避免程序代码过长且不易阅读，可以在主文件中编写程序的主要逻辑部分，而将程序的独立功能模块存放在其他文件中，以便于管理和维护。这种做法不仅提高了代码的可读性，还便于功能的复用和扩展。

元件清单

Buzzer
蜂鸣器 x1

Jumper Cables M/M
跳线（公公头） x2

硬件连接

按如下连线图连接，注意蜂鸣器引脚正（+），负（-）。标有+的引脚接到数字口 8，另一个接 GND。蜂鸣器的正负极通过盖子上的+符号查看。

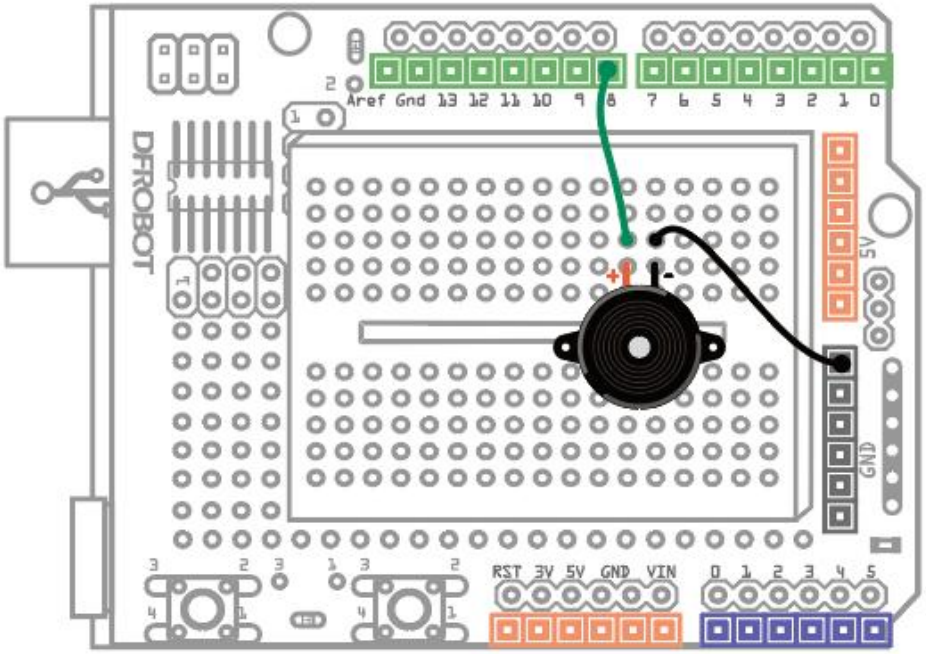


图 1 音乐蜂鸣器连线图

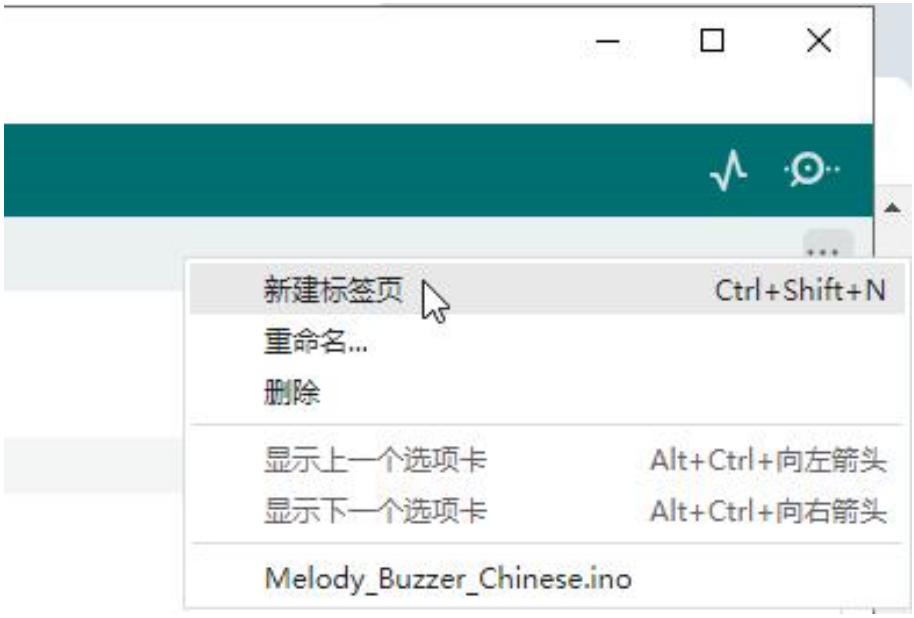
示例代码

样例代码 - Main Sketch:

```
//项目 - 音乐蜂鸣器
#include "pitches.h"  // 包含一个头文件

int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};  // 旋律中的音符序列

int noteTypes[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};  // 每个音符的类型: 4 = 四分音符, 8 = 八分音符
```

```
void setup() {

  for (int thisNote = 0; thisNote < 8; thisNote++) {
    int noteDuration = 1000 / noteTypes[thisNote];
    // 计算音符的持续时间：1 秒 / 音符类型
    tone(8, melody[thisNote], noteDuration);
    // 在引脚 8 上播放音符
    int pauseBetweenNotes = noteDuration * 1.30;
    // 根据音符持续时间计算暂停时间，使旋律听起来更自然
    delay(pauseBetweenNotes);  // 音符之间的暂停
    noTone(8);  // 停止在引脚 8 上播放的音调
  }
}

void loop() {
  // 不需要重复播放旋律
}
```

要创建一个新的头文件，点击串口监视器下方的按钮，选择新建标签页，输入文件名 pitches.h。



Pitches.h:

```
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
```

```
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
```

```
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

代码回顾

在这个项目中，我们新建了 pitches.h 文件，并复制了一段代码。从项目【温度报警器】中我们了解到，不同的频率会产生不同的音调。而我们复制的这段代码是最初由 Brett Hagman 编写的音符表，包含了所有典型音符所对应的频率。比如，NOTE_C4 是中央 C 音，对应的频率是 262。有了这张音符表，我们就可以通过输入音符的名称来编写旋律了。

在 Main Sketch 中使用了#include 语法来导入了新建的文件或外部库，将不同的文件关联起来。

在程序中，我们首先创建了两个数组来按顺序储存音调与节奏。

```
int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};
int noteTypes[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};
```

上面这个数组 noteTypes[] 定义了每个音符的持续时间。这里使用的是一种简化的表示方法，其中数字代表音符的“类型”，而不是确切的毫秒数。例如，4 代表四分音符，8 代表八分音符。实际的持续时间将在 setup()函数中计算得出。

数组是一个变量的集合，可以通过索引号来找到数组中的元素。可以看出我们创建的数组中分别有 8 个元素。数组是从 0 开始索引的，因此数组中的 8 个元素的索引号是 0~7。

在本程序中旋律只会播放一次，因此我们在 setup 里编写播放音符的程序。

通过一个 for 循环，来遍历数组中的每一个元素，通过变量 thisNote 表示目前的元素索引号。

```
for (int thisNote = 0; thisNote < 8; thisNote++) {
  int noteDuration = 1000 / noteTypes[thisNote];
  // 计算音符的持续时间：1 秒 / 音符类型
  tone(8, melody[thisNote], noteDuration);
  // 在引脚 8 上播放音符
  int pauseBetweenNotes = noteDuration * 1.30;
  // 根据音符持续时间计算暂停时间，使旋律听起来更自然

  delay(pauseBetweenNotes);  // 音符之间的暂停
  noTone(8);  // 停止在引脚 8 上播放的音调
}
```

在循环中，首先计算了音符的持续时间。在音乐的定义中，四分音符占一个小节的 1/4，因此可以通过小节时间（这里设置的是 1 秒）除以 4 来计算每个音符的持续时间。

然后使用 tone()函数设置引脚，frequency 和 duration，来播放音符。

每个音符中间需要有一定的间隔才能够让我们方便区分，因此需要在音符之间设置延迟。间隔时间根据音符持续时间调整会让旋律更加流畅，用

noteDuration * 1.30 会得到不错的效果，你也可以试试设置 delay(200) 的固定时间，来看看效果会有什么不一样。

在音符播放完，使用 noTone 让声音停止。

硬件回顾

Arduino 开发板通常都配备了一个复位键（Reset Button），这个按键在多个方面都非常有用，特别是在编程、调试和重新启动 Arduino 时。复位键的主要功能是使 Arduino 板上的微控制器（如 ATmega 系列）重置其状态，类似于计算机上的重启操作，但针对的是微控制器本身。

Arduino UNO 的复位键通常位于开发板的左上角（如图 2），靠近 USB 接口或其他 I/O 接口。它是一个小按钮，旁边标有“RESET”字样。同样，你也可以使用扩展板上的复位按键来重启 Arduino 里面的程序。

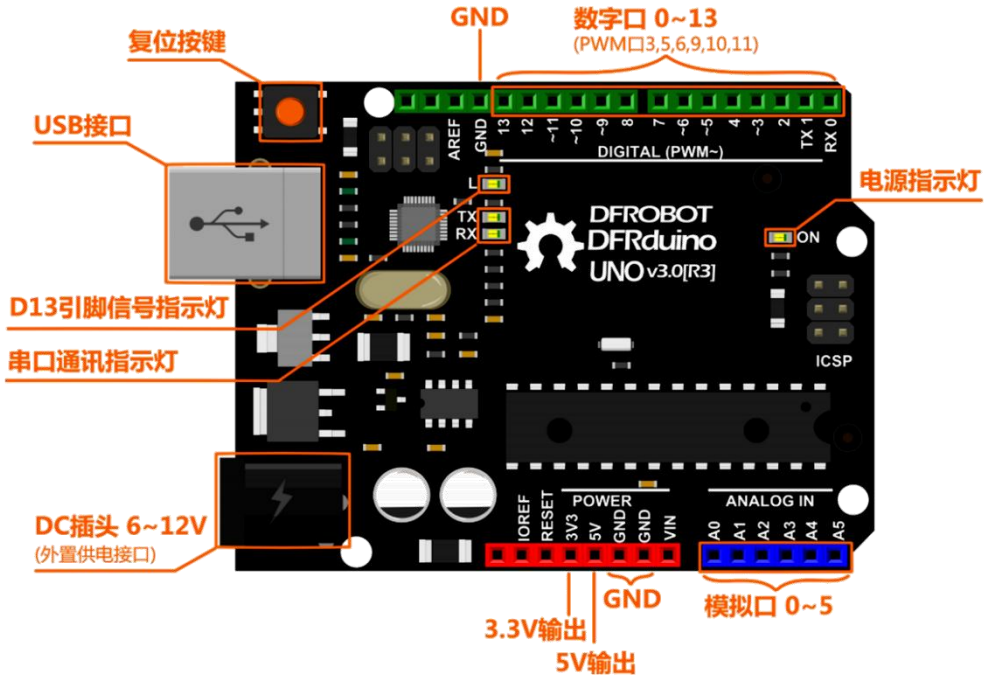


图 2 Arduino UNO 复位键位置

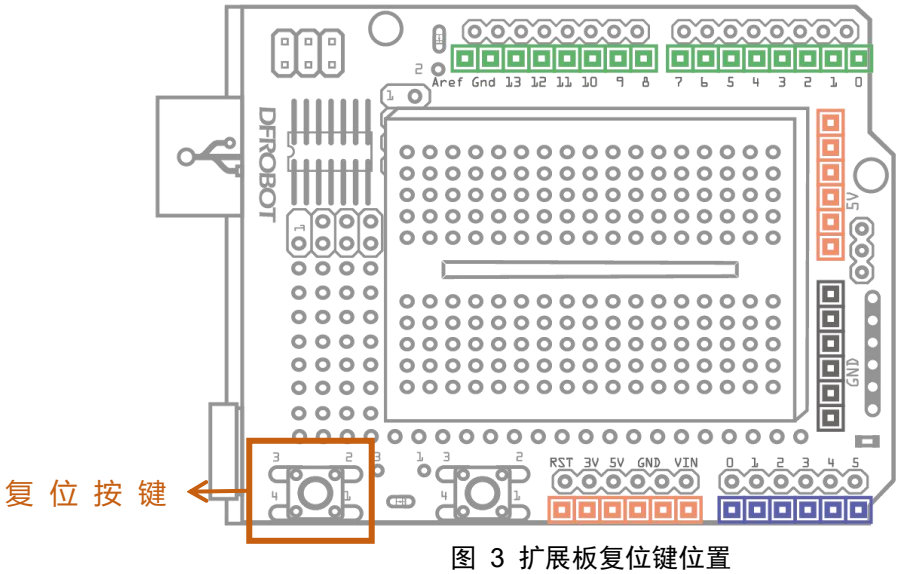


图 3 扩展板复位键位置

课后练习

你有没有熟悉的旋律想要通过蜂鸣器来播放呢？比如小星星或者玛丽有只小羊羔？来试试改变数组中的音符和节奏吧。别忘了改变 for 循环中的数量哦。

项目 - DJ 调音台

通过控制蜂鸣器的频率和节拍可以产生不同的声音。那么，如果可以自由调整这两个变量，会产生什么样的效果呢？在本节课中，我们将制作一个简易的 DJ 调音台。通过使用两个电位器，你将能够分别控制声音的频率和节拍，从而体验像 DJ 一样实时调音的乐趣！

元件清单

10K Potentimeter
10K 电位器 **x2**

Buzzer
蜂鸣器 **x1**

Jumper Cables M/M
跳线（公公头） **x8**

硬件连接

按如下连线图连接。

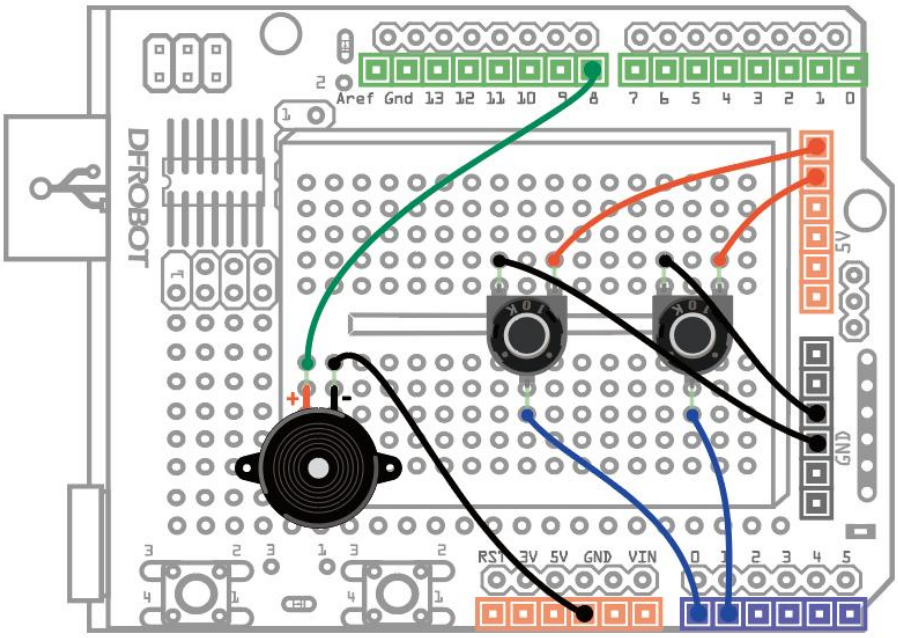


图 1 DJ 调音台连线图

示例代码

样例代码:

```
//项目 - DJ 调音台

int pitchPin = A0; // 电位器 1 - 模拟引脚 A0，用于控制音调

int beatPin = A1;  // 电位器 2 - 模拟引脚 A1，用于控制节奏

int piezo = 8;     // 压电蜂鸣器 - 数字引脚 8


void setup() {

  Serial.begin(9600); // 初始化串行端口，波特率为 9600
```

```
}

void loop() {
  int pitchVal = analogRead(pitchPin);
// 从音调电位器读取值
  int beatVal = analogRead(beatPin);
// 从节奏电位器读取值

  int frequency = map(pitchVal, 0, 1023, 0, 5000);
   // 将音调值映射到频率范围 0-5000 Hz
  int duration = map(beatVal, 0, 1023, 100, 5000);
   // 将节奏值映射到持续时间范围 100-5000 ms

  // 将频率和持续时间打印到串行端口
  Serial.print("频率: ");
  Serial.print(frequency);
  Serial.print(" Hz, 持续时间: ");
  Serial.print(duration);
  Serial.println(" ms");

  tone(piezo, frequency, duration);
   // 在压电蜂鸣器上播放指定频率和持续时间的音调
  int pauseBetweenNotes = duration * 1.30;
```

```
  // 计算音符之间的暂停时间
  delay(pauseBetweenNotes); // 等待暂停时间
  noTone(piezo); // 停止播放音调
}
```

成功下载完程序后，打开 Arduino IDE 的串口监视器。

设置串口监视器的波特率为 9600。

可以直接从串口读取当前的频率和节拍值，旋转旋钮即可看到读数变化。

代码回顾

这个程序读取两个电位器的值，用来控制压电蜂鸣器的音调频率和持续时间。这些值还会输出到串口监视器上，方便观察。

程序开始设置了三个变量，来定义两个电位器与蜂鸣器的引脚。

```
int pitchPin = A0;
int beatPin = A1;
int piezo = 8;
```

在 setup 中，我们使用了一个新的函数：

```
Serial.begin(9600);
```

这个函数用于初始化串口波特率，也就是数据传输的速率，是使用串口必不可少的函数。直接输入相应设定的数值就可以了，如果不是一些特定的无线模块对波特率有特殊要求的话，波特率设置只需和串口监视器保持一致即可。我们这里就只是用于串口监视器。

在 loop 中设置了两个新的变量，用于记录两个电位器的读数。

```
int pitchVal = analogRead(pitchPin);
int beatVal = analogRead(beatPin);
```

通常，程序开头定义的变量称为全局变量，可在程序的任何部分访问和使用。相比之下，像 pitchVal 和 beatVal 这样的变量可在循环中定义为局部变量，因为它们只在循环内部使用。

这里用到了一个新函数——analogRead(pin)。
这个函数用于从模拟引脚读值，pin 是指连接的模拟引脚。Arduino 的模拟引脚连接到一个 10 位 A/D 转换，输入 0~5V 的电压对应读到 0~1023 的数值，每个读到的数值对应的都是一个电压值。

我们读取的电位器输出的电压范围是从 0 到 1023。在之前的多节关于蜂鸣器的

课程中，我们已经了解到人类可以听到的声音频率大约在 0 到 5000 赫兹之间（请参考第项目【音乐蜂鸣器】的 pitch 文件）。因此，我们需要将 0 到 1023 的电压范围转换到 0 到 5000 赫兹的频率范围，这个功能称为映射。

```
int frequency = map(pitchVal, 0, 1023, 0, 5000);
```

Map 函数格式如下：

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

map 函数会将 fromLow 到 fromHigh 之间的值映射到 toLow 在 toHigh 之间的值。还可以用来翻转数值的范围，例如：y = map(x, 1, 50, 50, 1)。

同理，我们想要用另一个电位器控制声音的持续时间，范围是 100 毫秒到 5 秒之间。

```
int duration = map(beatVal, 0, 1023, 100, 5000);
```

那串口收到数据后，如何在串口监视器上显示呢?

```
Serial.print(val);
Serial.println(val);
```

println()与 print()区别就是，println()比 print()多了回车换行，其他完全相同。

我们可以使用下面这两个函数输出不同格式的内容。

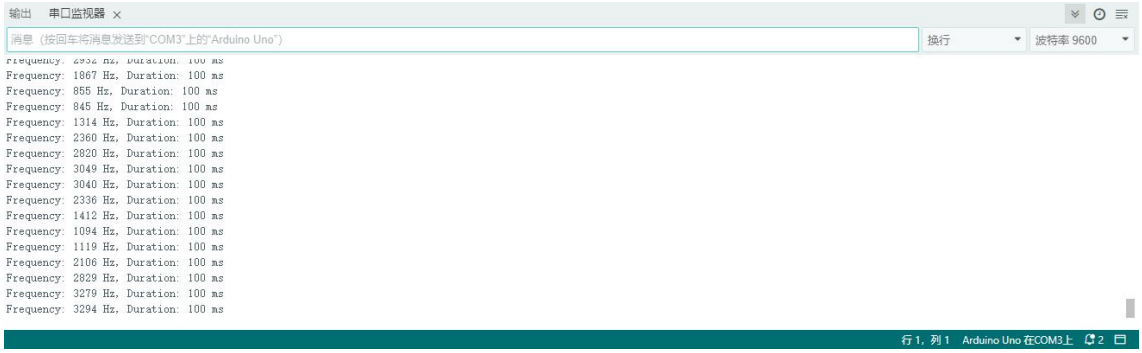
```
Serial.print(“Frequency: ”);
Serial.print(frequency);
```

这两条语句虽然看起来括号中的内容都是 frequency，但第一条语句中加了双引号，将引号中的内容作为字符串原样输出；而第二句中的 frequency 代表的是 frequency 变量，将输出该变量当前的数值。

在输出字符串时，可以在文字内容前后增加冒号和空格，让输出内容更容易阅读。

```
Serial.print("频率: ");
Serial.print(frequency);
Serial.print(" Hz, 持续时间: ");
Serial.print(duration);
Serial.println(" ms");
```

这段程序的输出的效果如下：



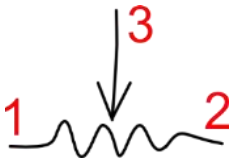
使用与项目【音乐蜂鸣器】相同的程序来播放声音并设置间隔，如有疑问，请参考之前的内容。

硬件回顾

电位器

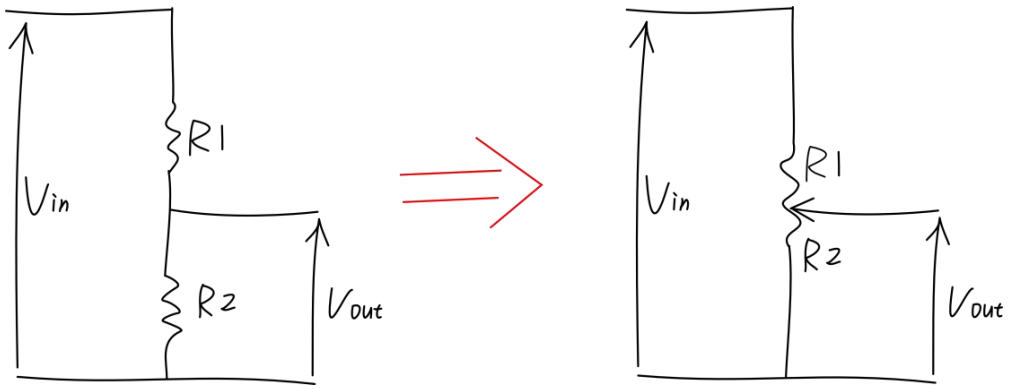
电位器可以理解为是一个电阻，只是这个电阻阻值可变。我们这里可调节的范围是 0~10KΩ。电阻两端接电源，通过中间引脚调节阻值，随着电阻值的改变而带动电压变化。我们用模拟口 0 读取到这个变化中的电压值，并转换为对应的音调和节奏。这就是整个的控制过程。

电位器在电路上的表示的图标为下图，分别对应器件上的 3 个引脚。



简单的看下原理，不知道还记不记得在第项目【可控舵机】中讲到的分压原理。电位器用的同样是分压原理。我们可以理解为，电位器被拆分为上下两个电阻

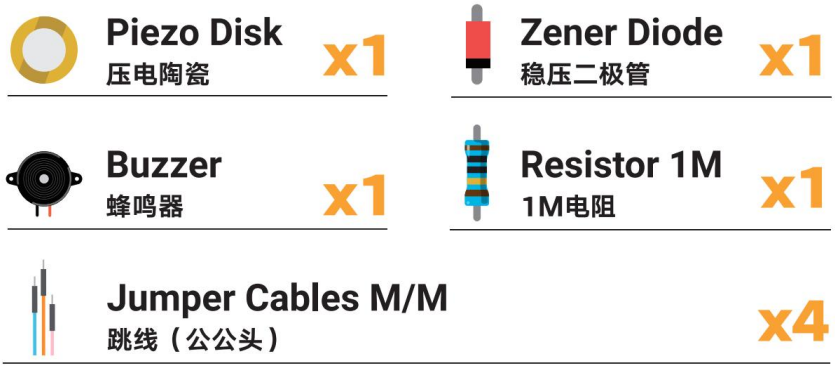
R1 和 R2，随着转动电位器，上下阻值发生变化，从而对应的输出电压就不同。我们可以想象成切蛋糕，分到的蛋糕越多（电阻），吃下去的能量（电压 Vout）也就越大。电压值大小的变化可以直接通过模拟口读到的值（0~1023）反应出来。



项目 - 敲击电子鼓

在这个项目中，我们将制作一个敲击电子鼓，探索 Arduino 如何捕捉敲击产生的信号，并控制蜂鸣器发出不同频率的声音。通过改变按压力度大小，你将能亲身体验到声音频率的实时变化，感受电子世界的奇妙与乐趣。

元件清单



硬件连接

蜂鸣器我们已经连接过很多次了，只要在此基础上加入压电陶瓷相关电路即可，压电陶瓷部分的电路还包含一个 1M 的电阻和一个 5.1V 的稳压二极管，请注意稳压二极管的连接方向（如图 1）。

因为当压电陶瓷受到敲击时，会产生非常高的电压，足以对 Arduino 造成损坏。稳压二极管可以保护 Arduino 免受高电压的损坏，而电阻的作用是泄放来自压电陶瓷的高电压。

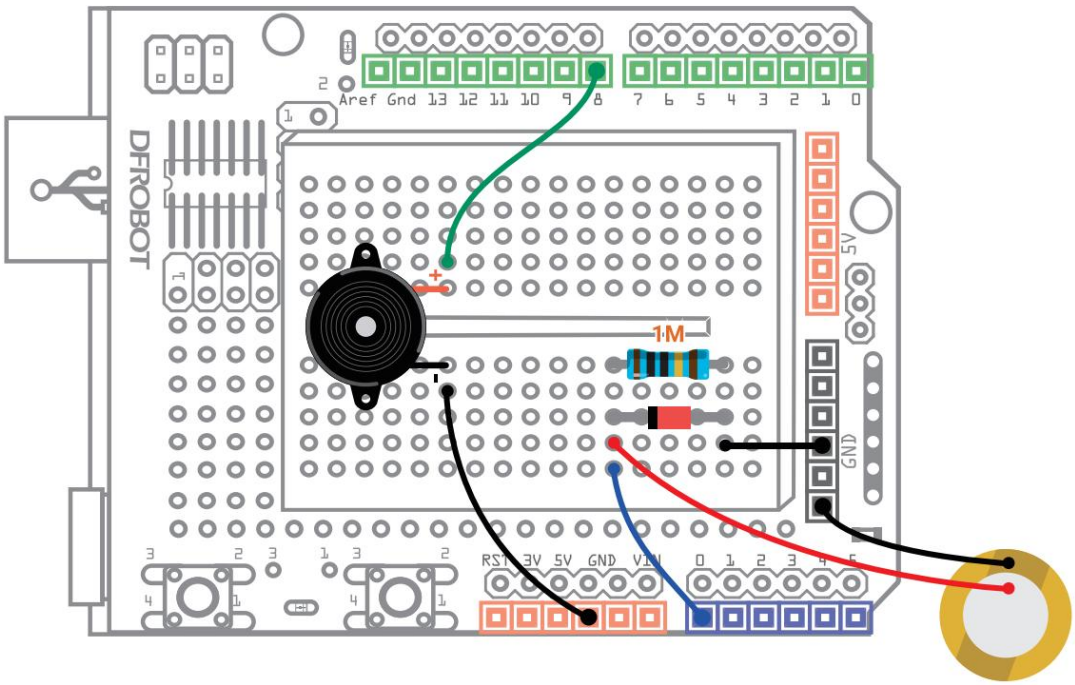


图 2 敲击电子鼓连线图

示例代码

样例代码:

```
//项目 - 敲击电子鼓
```

```
// 这些常量不会改变。它们用于为所使用的引脚命名：
const int analogInPin = A0;  // 模拟输入引脚，压电陶瓷连接到此引脚
const int analogOutPin = 8;  // 模拟输出引脚，蜂鸣器连接到此引脚

int sensorValue = 0;  // 从压电陶瓷读取的值
int outputValue = 0;  // 输出到 PWM（模拟输出）的值

void setup() {
  // 初始化串行通信，波特率为 9600：
  Serial.begin(9600);
}

void loop() {
  // 读取模拟输入值：
  sensorValue = analogRead(analogInPin);
  // 将其映射到模拟输出的范围：
  outputValue = map(sensorValue, 0, 1023, 0, 5000);
  /* 改变模拟输出值（但注意，Arduino 8 号引脚为数字输出，这里使用 tone 函数模拟发声）*/

  tone(analogOutPin, outputValue, 100);
  /* 使用 tone 函数在 8 号引脚（数字输出）上发声，频率为 outputValue，持续时间为 100 毫秒  */
```

```
  // 将结果打印到串行监视器:
  Serial.print("sensor = ");
  Serial.println(sensorValue);

}
```

当压电陶瓷受到击打时，蜂鸣器会发出带有音调变化声音。同时，观察你的串口监视器，sensor 这个数值会快速攀升到最大值，然后跌落回 0。不同的数值反映了压电陶瓷受到挤压或击打的强烈程度，值越大说明其受到的挤压越猛烈。如果什么都没观测到，请检查你的连接，尤其注意检查压电陶瓷和稳压二极管连接的方向。

代码回顾

在代码的开始阶段就出现了一种我们没有使用过的数据类型，在之前的项目中，我们接触了几种常见的数据类型（比如 int，long 等）。在这个项目中，将引入一种新的数据类型：const，也就是常量。

```
  const int analogInPin = A0;
  const int analogOutPin = 8;

  int sensorValue = 0;
  int outputValue = 0;
```

在上段代码中，analogInPin 和 analogOutPin 被定义为常量，分别表示压电陶瓷的连接的模拟输入引脚和蜂鸣器连接的模拟输出引脚。

接着声名了两个整数类型的变量 sensorValue 和 outputValue。前者用于存储从压电陶瓷读取到的原始模拟数值，后者用于存储经过映射处理后的值（转化为蜂鸣器可以发出的声音范围内的值）。

接下来初始化 setup()函数就非常简单了，只要设置一个串口波特率就可以了，这使得我们可以监控和调试压电陶瓷的输入信号。

在 loop()函数中，首先从 analogInPin 读取压电陶瓷产生的模拟信号值，并将其存储在 sensorValue 变量中。

接着，使用下方的语句，将 sensorValue 的值从 0 到 1023 的范围映射到 0 到 5000 的范围内。

```
outputValue = map(sensorValue, 0, 1023, 0, 5000);
```

这是因为压电陶瓷的输出范围通常是 0 到 1023，而 tone 函数的频率参数需要在 Arduino 可以发出的频率范围内（通常是 31Hz 到 65535Hz，但实际效果受硬件限制）。

```
tone(analogOutPin, outputValue, 100);
```

在 analogOutPin 引脚上生成一个频率为 outputValue（由压电陶瓷控制）的

声音，持续时间为 100 毫秒。tone()函数来驱动蜂鸣器发出不同频率的声音，是基于 PWM（脉冲宽度调制）技术实现的。PWM 技术允许数字引脚输出模拟信号的效果，通过快速切换数字信号的高低电平（即开和关），并以特定的速率重复这一过程，可以模拟出不同频率的信号。更多 PWM 技术相关知识详见项目【呼吸灯】。

```
  Serial.print("sensor = ");
  Serial.println(sensorValue);
```

最后，为了方便程序的调试和观察压电陶瓷的实时值使用，将 sensorValue 的值打印出来。

硬件回顾

压电陶瓷

压电陶瓷，也叫压电换能器，术语压电的意思就是“由压力产生电流”。当一个压电设备受到挤压时，它就会产生一个电荷，如图 2 所示。换能器在与 Arduino 一起使用时的一个典型应用就是将其作为敲击传感器。

当换能器受到击打或碰撞时，Arduino 就可以检测到，并触发相应的动作，在这个项目中就是发出与敲击力度相关的声音。

相反地，当一个电荷施加到压电换能器上时，它就会发生形变，如图 3 所示。如果你对其施加一个以一定频率变化的电压，换能器的振动就会使其发出声音

或者旋律。

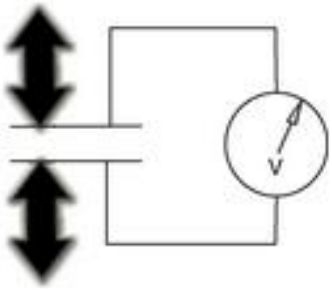


图 2 当压电换能器发生形变时就会产生一个电荷，挤压和松开换能器，就会产生一个变换的电压

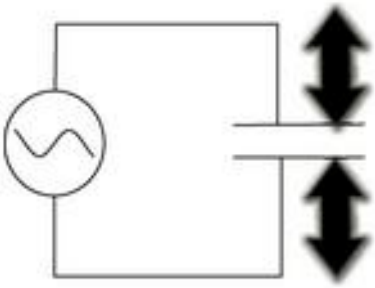


图 3 当对压电换能器施加一个变化的电压时，换能器就会发生形变

正如上面图 2 和图 3 展示的那样，一个压电换能器既可以作为输入设备，也可以作为输出设备。就是说它既能将敲击信号转变为电信号，也能将电信号转变为震动频率。

在使用压电陶瓷的时候，要特别注意它的极性，极性是通过由低熔点焊料焊接在上面的一红一黑两条线来区分。黑线连接到电路中的 GND 部分，红色连接到 Arduino 的模拟输入引脚。

稳压二极管

在项目【闪烁第一个 LED】中，我们介绍了发光二极管-LED。

这个项目中的稳压二极管是一种特殊的二极管，它被设计成在超过击穿电压 (膝

点电压）时允许电流反向通过。

稳压二极管需要以正确的方式连接才能保护 Arduino 的模拟输入端电压不超过 5 V。按照惯例，在二极管的阴极或者负极，一般会设计有一条黑色的条纹，这一极会被连接到 GND，但在这个项目的电路中，你要将二极管反向偏置，也就是将二极管的阴极连接到电路的正极。稳压二极管的工作原理是，只在超过反向击穿电压的时候导通，在本例中击穿电压是 5.1 V。任何超过 5.1 V 的电压都会导致二极管反向导通并与 GND 短路，从而保护 Arduino 的输入端。

课后练习

在硬件回顾中，我们了解到压电陶瓷既可以作为输入设备使用，也可以作为输出设备使用。请尝试使用 tone()函数向电压陶瓷写入信号，使其发出指定的频率（赫兹）。并通过组合发出一段旋律吧！（注意：压电陶瓷发出的声音要比蜂鸣器小很多，请注意聆听！）

项目 - 手势控制灯

想象一下，仅需在空中挥挥手，就可以轻松控制灯的开关。这不仅为日常生活增添了便利，还为智能家居技术带来了更高的互动性和趣味性。手势识别的技术有很多种，在这个项目中，我们将利用两个光敏电阻，通过检测手在不同位置的光遮挡情况，实现手势控制。

元件清单

硬件连接

按下图进行硬件连接。

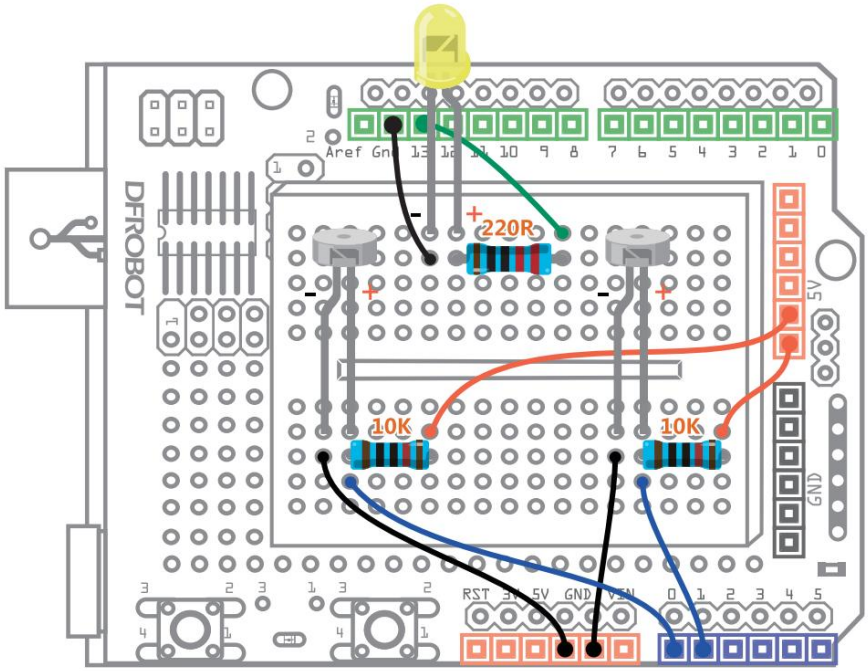


图 1 手势控制灯连线图

代码示例

样例代码:

```
int ledPin = 13;          // 定义 LED 引脚为 13
int lightSensor1 = 0; // 定义左侧光传感器引脚为 A0
int lightSensor2 = 1; // 定义右侧光传感器引脚为 A1


int cali1 = 0;            // 左侧光传感器的校准变量
int cali2 = 0;            // 右侧光传感器的校准变量
```

```
bool isOn = false;      // LED 状态控制变量

void setup() {
  Serial.begin(9600);       // 初始化串行通信

  pinMode(ledPin, OUTPUT);  // 将 LED 引脚设置为输出

  // 校准光传感器：读取 10 次数据并平均作为环境光基线
  for (int i = 0; i < 10; i++) {
    cali1 += analogRead(lightSensor1);
    cali2 += analogRead(lightSensor2);
    delay(500);
  }
  cali1 = (cali1 / 10);
  cali2 = (cali2 / 10);

  Serial.print("环境光: ");
  Serial.print(cali1);
  Serial.print(" , ");
  Serial.println(cali2);

  // 通过 LED 闪烁两次来指示程序的开始
```

```
  for (int i = 0; i < 2; i++) {
    digitalWrite(ledPin, HIGH);
    delay(1000);
    digitalWrite(ledPin, LOW);
    delay(1000);
  }

  Serial.println("开始");
}
void loop() {
  int sensorVal1 = analogRead(lightSensor1);
// 读取左侧光传感器的值
  int sensorVal2 = analogRead(lightSensor2);
// 读取右侧光传感器的值
  int threshold = 30;  // 设置手势检测阈值
  Serial.print("L: ");
  Serial.print(sensorVal1);
  Serial.print("   R: ");
  Serial.println(sensorVal2);
  // 根据传感器值确定手势方向并控制 LED
  if ((sensorVal1 > (cali1 + threshold)) && !(sensorVal2 > (cali2 +
threshold))) {
    isOn = true;
```

```
  }
  if (!(sensorVal1 > (cali1 + threshold)) && (sensorVal2 > (cali2 +
threshold))) {
    isOn = false;
  }
  if (isOn) {
    digitalWrite(ledPin, HIGH);  // 打开 LED
    Serial.println("打开");
  } else {
    digitalWrite(ledPin, LOW);   // 关闭 LED
    Serial.println("关闭");
  }
  delay(100);  // 延迟 100 毫秒以避免频繁读取
}
```

请将装置放置在光线稳定且无阴影遮挡的环境中。上传程序后，将自动进行 5 秒钟的校准，检测当前光线值。校准完成后，LED 会闪烁两次，指示程序启动。向左挥手可打开灯光，向右挥手可关闭灯光。如果效果不理想，可以参考串口打印的读数，调整阈值设置。

代码回顾

在程序开始时，定义 LED 和两个光敏电阻的引脚。然后，创建两个名为 cali 的变量来记录光敏电阻的初始读数。

```
int cali1 = 0;
int cali2 = 0;
```

接下来，我们介绍一种新的变量类型，bool，它代表布尔变量。它只有两个可能的值：true（真）和 false（假），通常用于表示条件的真实性或开关的状态。在这里，我们定义一个名为 isOn 的布尔变量来记录 LED 当前的开/关状态。

```
bool isOn = false;
```

光敏电阻的读数会根据其位置和一天中的时间而变化。因此，在程序的设置阶段，我们需要执行自动校准以获得传感器的基线读数。

```
for (int i = 0; i < 10; i++) {
    cali1 += analogRead(lightSensor1);
    cali2 += analogRead(lightSensor2);
    delay(500);
  }
```

在 For 循环中，光敏电阻的模拟值被读取了 10 次，每次读取之间间隔 500 毫秒，并将结果累加到变量 cali1 和 cali2 中。

具体来说，这个循环的目的是为了对光敏电阻进行校准，以获取在特定环境下（即程序开始时）它们的基础读数。通过多次读取并取平均值，可以减少因单次读数误差或环境光线的短暂变化而导致的误差。

之后，将累加得到的总读数除以 10，即可得到两个光敏电阻的校准值（即基线读数），这些值将用于后续的比较和判断，以确定 LED 的开/关状态。

```
  cali1 = (cali1 / 10);
  cali2 = (cali2 / 10);
```

循环结束后，计算左侧和右侧光敏电阻的平均值。这些平均值将作为后续光线比较和决策的校准基线值。

一旦完成校准，将平均值打印到串行监视器上，并闪烁 LED 两次以指示程序的开始。

在 loop 函数中，我们将比较光线的值并控制 LED 的开关。

首先，设置两个变量来记录光敏电阻的实时读数。

```
  int sensorVal1 = analogRead(lightSensor1);
  int sensorVal2 = analogRead(lightSensor2);
```

从之前关于光敏电阻的课程中，我们了解到较高的读数对应于较暗的光线条件。当手势发生时，投射到光敏电阻不同部分的阴影会导致其读数上升。因此，我们可以通过监控两侧光敏电阻读数的变化来区分左右挥手的手势。调整阈值变量可以让我们改进这个检测过程的灵敏度。

```
int threshold = 30;
```

当左侧光敏电阻的读数超过其基线值加上阈值时，表示左侧有遮挡物。

```
if (sensorVal1 > (cali1 + threshold))
```

然而，仅依赖左侧读数是不够的；它需要与右侧光敏电阻的读数相结合来进行准确评估。

当左侧被遮挡而右侧没有时，它表示一个从右向左挥动的手势。

```
if ((sensorVal1 > (cali1 + threshold)) && !(sensorVal2 > (cali2 +
threshold)))
```

在这行代码中，！作为逻辑非运算符，用于反转内部表达式的值。&& 作为逻辑与运算符，检查两个表达式是否同时为真。如果条件满足，isOn 被设置为 true，从而打开 LED。否则，isOn 被设置为 false。

```
  isOn = true;
```

最后，我们使用 isOn 的当前值来决定是否打开或关闭 LED。在这里，if (isOn) 是 if (isOn == true) 的简写形式。

```
  if (isOn) {
      digitalWrite(ledPin, HIGH);
      Serial.println("Turn On");
    } else {
      digitalWrite(ledPin, LOW);
      Serial.println("Turn Off");
    }
```

项目 - 反应速度测试

在这个项目中我们将用 Arduino 制作一个反应速度测试游戏。通过点亮 LED 灯并快速按下对应按钮来得分，游戏持续一分钟。准备挑战你的反应极限了吗？让我们开始吧！

元件清单

硬件连接

在这个项目中，我们使用到的 LED 灯和按键模块，都是在之前的项目中出现过的。不同的一点是之前项目【交通信号灯】中给按键外加了一个下拉电阻，在这个项目中，我们使用 Arduino 芯片内部的上拉电阻，详见本项目的硬件回顾。请按照下图将硬件连接正确。

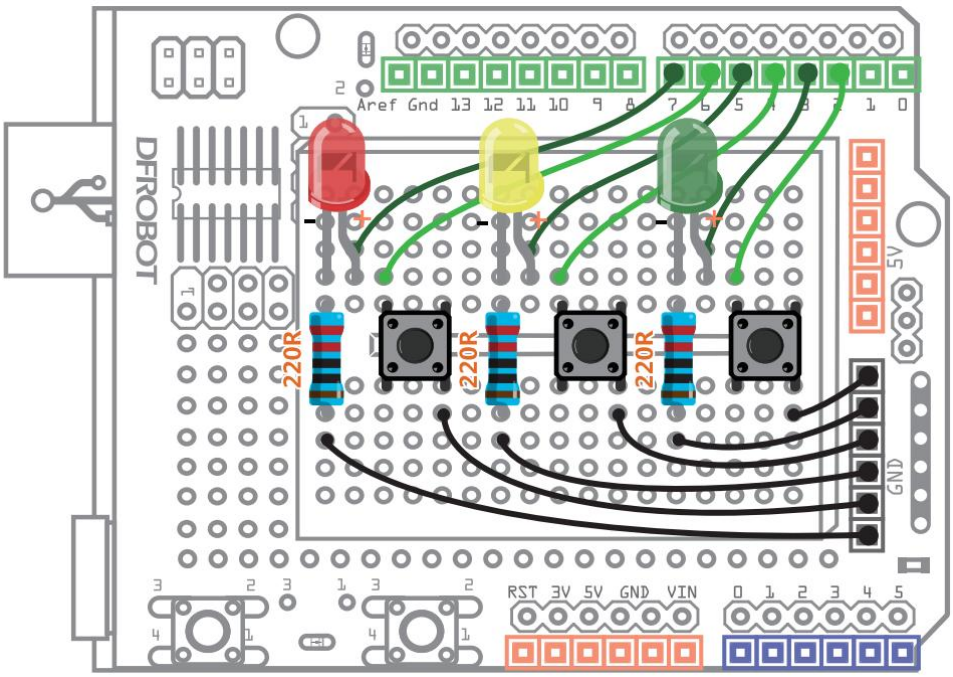


图 1 反应速度测试连线图

示例代码

样例代码:

```
//项目 - 反应速度测试

// 定义 LED 灯和按钮的引脚

const int LEDS[] = {3, 5, 7};

const int BUTTONS[] = {2, 4, 6};


int score = 0; // 玩家的得分

bool gameStarted = false; // 游戏是否开始的标志

unsigned long gameStartTime; // 记录游戏开始的时间戳

const unsigned long gameDuration = 60000;
```

```
// 游戏持续时间，单位为毫秒（1 分钟）


void setup() {
  Serial.begin(9600); // 初始串口通信
  // 设置 LED 灯和按钮的引脚模式
  for (int i = 0; i < 3; i++) {
    pinMode(LEDS[i], OUTPUT);
    pinMode(BUTTONS[i], INPUT_PULLUP);
  }
}


void loop() {
  // 检查是否同时按下了所有三个按钮来开始游戏
  if (!gameStarted && digitalRead(BUTTONS[0]) == LOW &&
digitalRead(BUTTONS[1]) == LOW && digitalRead(BUTTONS[2]) == LOW)
{
    Serial.println("游戏开始");
    gameStarted = true;
    score = 0; // 重置得分
    gameStartTime = millis(); // 记录游戏开始的时间
    delay(1000); // 等待一秒后开始游戏
  }
```

```
  // 如果游戏已经开始，检查游戏是否超时
  if (gameStarted && millis() - gameStartTime < gameDuration) {
    int led = random(0, 3); // 随机选择一个 LED
    digitalWrite(LEDS[led], HIGH); // 点亮这个 LED
    long startTime = millis(); // 记录点亮 LED 的时间
    bool buttonPressed = false; // 初始化按钮状态（未被按下）

    // 给玩家一定时间来按按钮
    while (millis() - startTime < 1000) {
      if (digitalRead(BUTTONS[led]) == LOW) {
   // 检查是否按下了对应的按钮
        buttonPressed = true;
        break; // 如果按下了按钮，跳出循环
      }
    }

    digitalWrite(LEDS[led], LOW); // 熄灭 LED

    if (buttonPressed) {
      score++; // 如果按下了按钮，增加得分
      Serial.println("得分 +1");
    }
    delay(200); // 等待一段时间再继续下一轮
```

```
  } else if (gameStarted) {
    // 如果游戏时间超过一分钟或玩家没有在时间内按下按钮，游戏结束
    gameStarted = false;
    Serial.print("游戏结束! 你的得分: ");
    Serial.println(score); // 通过串口输出得分
    delay(3000); // 等待 3 秒后可以重新开始游戏
  }
}
```

上传代码后，打开串口监视器，同时按下三个按键开始游戏，串口监视器上出现游戏开始字样。一旦游戏开始，系统将随机点亮一个 LED 灯，并给玩家一秒钟的时间来按下对应的按钮，便可获得 1 分。当游戏时间到达 1 分钟时，串口监视器出现游戏结束并告知你的游戏得分。

代码回顾

变量声明部分不做过多解释了，这个部分用于定义 LED 和按钮的引脚数组，以及用于跟踪得分、游戏状态和游戏时间的变量。

```
const int LEDS[] = {3, 5, 7};
const int BUTTONS[] = {2, 4, 6};

int score = 0;
bool gameStarted = false;
```

```
  unsigned long gameStartTime;
  const unsigned long gameDuration = 60000;
```

gameDuration 是一个非负长整型的常数，用于定义游戏的持续时间。在这里，它被设置为 60000 毫秒，即 1 分钟，以确保游戏持续一分钟。

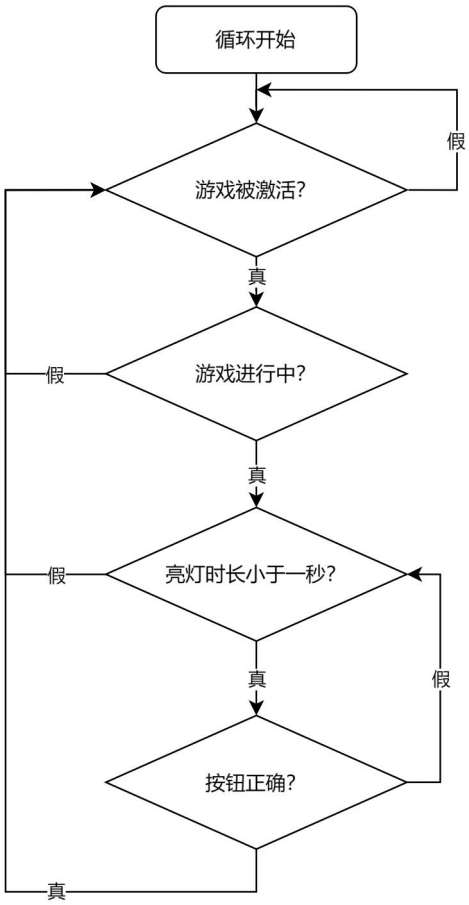
在 setup()函数中，初始化串口通信，并使用一个 for 循环设置 LED 和按钮的引脚模式。

在 loop()函数中，流程如下：

```
// 游戏开始前的检查
  if (!gameStarted && digitalRead(BUTTONS[0]) == LOW &&
digitalRead(BUTTONS[1]) == LOW && digitalRead(BUTTONS[2]) == LOW
  ) {
      // 游戏初始化
      gameStarted = true;
    }

// 游戏被激活后
  if (gameStarted && millis() - gameStartTime < gameDuration) {
      // 游戏进行中
    } else if (gameStarted) {
```

```
        // 游戏结束
    }
  // 返回游戏开始前的检查
```

如果用流程图，表示如下：



第一个部分是游戏开始前的检查。

用来判断游戏是否被激活，激活指的是从游戏的非激活状态（即未开始也未在进行中）转换到准备开始的状态。

```
if (!gameStarted && digitalRead(BUTTONS[0]) == LOW &&
digitalRead(BUTTONS[1]) == LOW && digitalRead(BUTTONS[2]) == LOW)
```

所以在第一个判断中，既要判断三个按键是否被同时按下，又要判断当时游戏是否处于未激活状态。

游戏被激活后进行游戏的初始化:

```
    Serial.println("游戏开始");
    gameStarted = true;
    score = 0; // 重置得分
    gameStartTime = millis(); // 记录游戏开始的时间
    delay(1000); // 等待一秒后开始游戏
```

通过串口输出“游戏开始”，将 gameStarted 设置为 true 以标记游戏已经开始，重置得分 score 为 0，记录游戏开始的时间 gameStartTime 为当前毫秒数（使用 millis()函数获取），然后暂停 1 秒以给用户一个视觉或心理上的准备时间。

第二部分是游戏激活后的逻辑结构。

```
if(gameStarted && millis() - gameStartTime < gameDuration) {
```

```
    // 游戏进行中
  } else if (gameStarted) {
    // 游戏结束
  }
```

当游戏已经开始且游戏时间未超过设定的持续时间 gameDuration 时，执行游戏进行中的逻辑。如果游戏已经开始但游戏时间超过了设定的持续时间 gameDuration，执行游戏结束的逻辑。接下来是对于游戏进行中的逻辑和游戏结束的逻辑的详解。

游戏进行中：

一旦游戏开始，系统将随机点亮一个 LED 灯，并给玩家一秒钟的时间来按下对应的按钮。

```
int led = random(0, 3); // 随机选择一个 LED
digitalWrite(LEDS[led], HIGH); // 点亮这个 LED
long startTime = millis();
bool buttonPressed = false;  // 初始化按钮状态（未被按下）
```

同时记录下点亮 LED 的时间 startTime，并且把 buttonpressed 的状态设为 fasle（初始化），之后进入一个 while 循环，等待玩家在 1 秒内按下对应的按钮。

```
while (millis() - startTime < 1000) {
      if (digitalRead(BUTTONS[led]) == LOW) {
   // 检查是否按下了对应的按钮
        buttonPressed = true;
        break; // 如果按下了按钮，跳出循环
      }
```

如果玩家在 1 秒内按下了对应的按钮（通过检查 BUTTONS[led]是否被按下），则标记 buttonPressed 为 true 并跳出循环。

while 循环结束后，无论是否按下了按钮，都会熄灭 LED。

```
if (buttonPressed) {
      score++; // 如果按下了按钮，增加得分
      Serial.println("得分 +1");
    }
    delay(200);
```

如果 buttonPressed 为 true，则增加得分并输出得分增加的消息。

最后，使用 delay(200)等待 200 毫秒后继续下一轮游戏。

游戏结束：

```
  else if (gameStarted) {
      // 如果游戏时间超过一分钟，游戏结束
      gameStarted = false;
      Serial.print("游戏结束! 你的得分: ");
      Serial.println(score); // 通过串口输出得分
      delay(3000); // 等待 3 秒后可以重新开始游戏
    }
```

否则如果游戏已经开始但游戏时间超过了设定的持续时间 gameDuration，则执行游戏结束的逻辑。

将 gameStarted 设置为 false 以标记游戏结束，通过串口输出游戏结束的消息和玩家的最终得分，然后暂停 3 秒，给用户一个重新开始游戏前的缓冲时间。

硬件回顾

上拉电阻

通常，如果没有输入，将输入引脚引导至已知状态非常有用。这可以通过在输入端添加一个上拉电阻（至+5V）或一个下拉电阻（接地电阻）来实现。这个项目中使用的按键，由于没有内置上拉电阻，则需要在代码中设置引脚为 INPUT_PULLUP 或使用外部电阻。

Arduino 的 ATmega 芯片内置了 20K 上拉电阻器，可通过软件访问。这些内

置上拉电阻可通过将 pinMode() 设置为 INPUT_PULLUP 来访问。这有效地反转了 INPUT 模式的行为，其中 HIGH 表示传感器关闭，LOW 表示传感器打开。

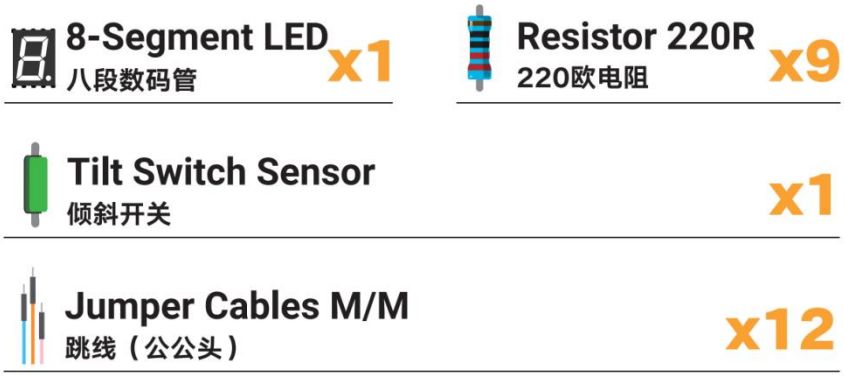
课后练习

尝试用现有的硬件，做一个颜色记忆游戏，实现一个简化版的记忆翻牌游戏，使用两个 LED（代表两张牌）和一个按钮。游戏开始时，两个 LED 随机亮起（模拟翻牌），然后熄灭。玩家需要记住这两个 LED 的亮起状态，并通过按钮输入来“翻牌”（实际上是重新点亮 LED），尝试匹配之前的亮起状态。根据结果配合串口输出“恭喜，你赢了！”；否则，输出“很遗憾，你输了！”。还有更多的玩法等待着你发现。

项目 - 数码管摇骰子

本项目旨在通过 Arduino 和数码管实现一个摇骰子的功能。当倾斜开关被触发时，数码管将随机显示 0 到 9 之间的一个数字，模拟摇骰子的效果。

元件清单



硬件连接

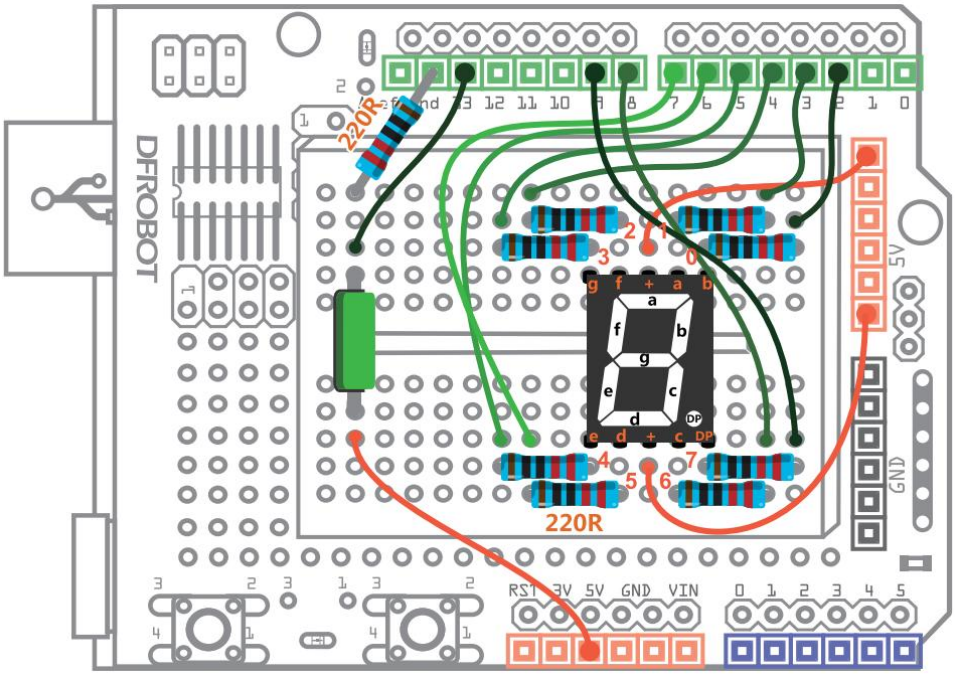


图 1 数码管摇骰子连线图

代码示例

样例代码:

```
// 项目 - 数码管摇骰子

// 定义数码管的引脚

const int pinA = 3;

const int pinB = 2;

const int pinC = 8;

const int pinDP = 9;

const int pinF = 4;

const int pinG = 5;
```

```
const int pinE = 6;

const int pinD = 7;


// 定义倾斜开关的引脚
const int tiltSwitchPin = 13;
int sensorValue;
int lastTiltState = HIGH;   // 上一次从倾斜传感器读取的状态
unsigned long lastDebounceTime = 0; // 上次检查时间
unsigned long debounceDelay = 500;   // 消抖延迟时间（毫秒）


// 二维数组定义数码管的数字 0-9 的显示模式
int numbers[10][8] = {
  {0, 0, 0, 0, 0, 0, 1, 1}, // 数字 0
  {1, 0, 0, 1, 1, 1, 1, 1}, // 数字 1
  {0, 0, 1, 0, 0, 1, 0, 1}, // 数字 2
  {0, 0, 0, 0, 1, 1, 0, 1}, // 数字 3
  {1, 0, 0, 1, 1, 0, 0, 1}, // 数字 4
  {0, 1, 0, 0, 1, 0, 0, 1}, // 数字 5
  {0, 1, 0, 0, 0, 0, 0, 1}, // 数字 6
  {0, 0, 0, 1, 1, 1, 1, 1}, // 数字 7
  {0, 0, 0, 0, 0, 0, 0, 1}, // 数字 8
  {0, 0, 0, 0, 1, 0, 0, 1}  // 数字 9
};
```

```
void setup() {
  Serial.begin(9600);
  // 设置数码管的引脚为输出
  pinMode(pinA, OUTPUT);
  pinMode(pinB, OUTPUT);
  pinMode(pinC, OUTPUT);
  pinMode(pinDP, OUTPUT);
  pinMode(pinF, OUTPUT);
  pinMode(pinG, OUTPUT);
  pinMode(pinE, OUTPUT);
  pinMode(pinD, OUTPUT);

  // 设置倾斜开关的引脚为输入并启用内部上拉电阻
  pinMode(tiltSwitchPin, INPUT_PULLUP);
}

void loop() {
  unsigned long currentTime = millis();
  int sensorValue = digitalRead(tiltSwitchPin);
  // 如果倾斜开关状态改变并且过来消抖时间：
  if (sensorValue != lastTiltState && (currentTime -
lastDebounceTime) > debounceDelay) {
```

```
    int randomNumber = random(0, 10); // 生成一个 0 到 9 的随机数
    displayNumber(randomNumber); // 显示随机数
    // 更新状态变量和消抖时间:
    lastTiltState = sensorValue;
    lastDebounceTime = currentTime;
  }

}

// 显示数码管的数字
void displayNumber(int number) {
  digitalWrite(pinA, numbers[number][0]);
  digitalWrite(pinB, numbers[number][1]);
  digitalWrite(pinC, numbers[number][2]);
  digitalWrite(pinD, numbers[number][3]);
  digitalWrite(pinE, numbers[number][4]);
  digitalWrite(pinF, numbers[number][5]);
  digitalWrite(pinG, numbers[number][6]);
  digitalWrite(pinDP, numbers[number][7]);
   // 如果需要显示小数点，将这里的状态改为 0
}
```

代码回顾

在 setup()之外的变量定义中，这些语句都是熟面孔了，请大家结合代码注释自行理解，定义数码管引脚请参考项目【红外遥控数码管】。

在 setup()函数中，设置数码管的引脚为输出，并设置倾斜开关引脚为输入模式并启用上拉电阻，如下：

```
pinMode(tiltSwitchPin, INPUT_PULLUP);
```

当没有触发时，引脚读取为高电平（HIGH）。

接着是 loop()函数，在 loop()中很关键的一点是采用软件编程来实现倾斜开关的消抖。
消抖的逻辑是通过比较当前状态和上一次状态，以及检查时间间隔是否超过设定的消抖延迟时间，来避免由于物理开关的接触抖动而导致的多次触发。

首先定义变量 lastDebounceTime，用于存储上一次检测到倾斜开关状态变化的时间点。debounceDelay 在这个项目中被设置为 500 毫秒。这意味着，如果倾斜开关的状态在 500 毫秒内频繁变化，那么这些变化将被视为抖动，并且不会被视为有效的状态变化。

```
unsigned long lastDebounceTime = 0;
```

```
unsigned long debounceDelay = 500;
```

在 loop() 函数内部，先获取当前的时间点（currentTime）和倾斜开关当前的状态值（sensorValue ）。

```
unsigned long currentTime = millis();
int sensorValue = digitalRead(tiltSwitchPin);
```

之后程序会检查倾斜开关的状态 (sensorValue) 。如果当前状态 (sensorValue) 与上一次状态（lastTiltState）不同，并且时间间隔超过设定的消抖延迟时间，那么程序将认为这是一个有效的状态变化。

```
if (sensorValue!=lastTiltState &&
(currentTime-lastDebounceTime) > debounceDelay){
    ......
  }
```

如果检测到是一个有效的变化，则执行 if 中的操作：

```
int randomNumber = random(0, 10);
displayNumber(randomNumber);
lastTiltState = sensorValue;
lastDebounceTime = currentTime;
```

生成一个随机数，并调用 displayNumber()函数将随机数显示在数码管上，接着更新状态变量和重置消抖时间。

项目 - 指尖开关

在这个项目中，我们利用 LED 灯、三极管和电阻，打造了一个指尖控制的开关装置。通过指尖轻触，LED 灯随即亮起，展现了电子元件的神奇互动。这个项目既简单又有趣，让我们在动手中探索电子世界的奥秘。

元件清单

硬件连接

注意这个项目中使用了两种阻值不同的电阻，在连接过程中注意区分。确保正确连接各元件，避免极性错误或引脚接反，请参考图 1 进行连接。

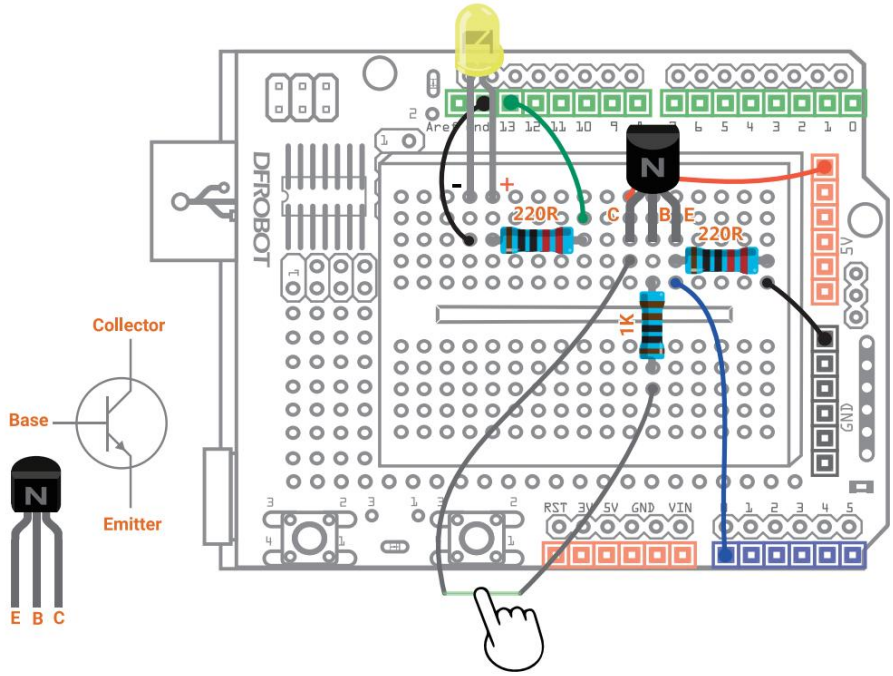


图 1 指尖开关连线图

示例代码

样例代码:

```
int ledPin = 13;

int NPNPin = 0;

void setup()

{

 pinMode(ledPin,OUTPUT);

 Serial.begin(9600);

}
```

```
void loop()
{
  int n=analogRead(NPNPin);    //读取 NPN 三极管模拟口数据
  Serial.println(n);
  if(n>0)                      //有电压反应就运行以下程序
  {
    digitalWrite(ledPin,HIGH); //点亮 led
    delay(100);                //延时，让 led 比较亮
    digitalWrite(ledPin,LOW);          //熄灭 led
  }

}
```

将代码上传完成后，用手指将两个跳线头相连，你会发现 LED 亮起，分开两个跳线头，LED 熄灭。

代码回顾

这个项目的代码比较容易理解，没有涉及到新的语句和概念。setup()函数做了两个常规操作，设置引脚的模式和初始化串口通信。

在 loop 函数中，int n = analogRead(NPNPin) 语句从模拟引脚 0 读取模拟值，并将其存储在变量 n 中。模拟信号范围通常是从 0 到 1023，将读取到的模拟信号通过串口发送到计算机，并在串口监视器中显示。

```
if(n>0)                    //有电压反应就运行以下程序
  {
    digitalWrite(ledPin,HIGH); //点亮 led
    delay(100);                //延时，让 led 比较亮
    digitalWrite(ledPin,LOW);          //熄灭 led
  }
```

if(n > 0) 语句检查变量 n（NPN 输出的模拟信号）的值是否大于 0。如果是，说明 NPN 三极管输出了大于零的模拟信号。

在 if 语句内部，digitalWrite(ledPin, HIGH) 语句向 LED 接入的数字引脚输出高电平，点亮 LED 灯。然后，delay(100) 语句使程序暂停执行 100 毫秒，这样 LED 灯就能保持点亮状态一段时间。

之后，digitalWrite(ledPin, LOW) 语句将 LED 接入的数字引脚设置为低电平，熄灭 LED 灯。

利用人体作为外部电阻接入电路中，通过判断 NPN 三极管的输出模拟信号，打开和关闭 LED 小灯，“真正”实现人与电路的交互。

硬件回顾

NPN 三极管

NPN 型三极管，它包含三个极：基极（B）、集电极（C）和发射极（E）（如图 2）。我们可以把三极管简单看作一个开关当 B 端没有电流或者电流极小时，CE 看作是未导通状态；当 B 端有电流进入时，CE 看作导通状态，此时 I_E 的大小与 B 端流入的电流大小有一定的比例关系（由放大倍数决定）。

发射极电流的变化可以间接反映输出模拟信号的强度变化，而三极管的放大作用则使得这种变化得以放大并输出为可用的模拟信号电压。

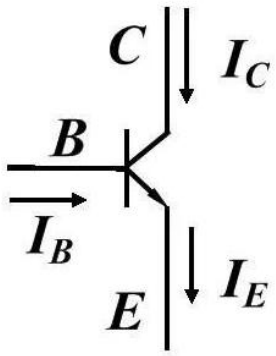


图 2 NPN 型三极管

由于在这个项目中，我们通过在基极（B）中接入电阻并连通，相当于给基极一个很小的电流，就能让三极管接通，电流从发射极（E）流出三极管。根据电压计算公式 U=IR，电压恒定，与基极串联的电阻越大时，电流就越小。所以在读取串口数值时会发现，与基极串联的电阻较大时（比如人体，电阻在 1MΩ~6MΩ），模拟读取出的信号就较小。如果将两端导线直接接触（电阻约等于 0Ω），读取的模拟信号就较大。

当基极（B）没有连通时，基极电流（I_B）为 0，则集电极电流（I_C）和发射极

电流（I_E）都为 0，那么串口读取到的模拟信号也为 0。

在本项目中，NPN 三极管参与的工作流程如下：

1. 输入：读取三极管基极（B）的模拟信号。
2. 处理：比较操作。
3. 输出：打开或关闭小灯。

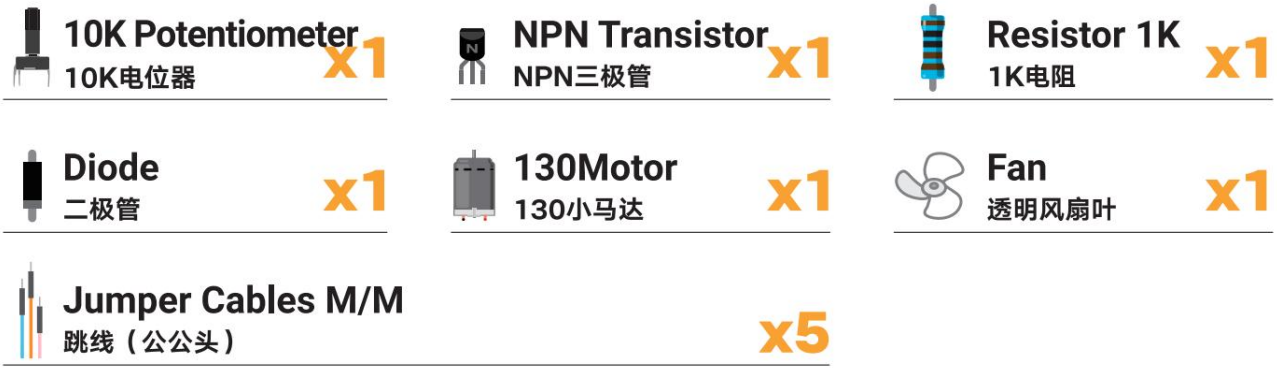
课后练习

本项目的电路可以通过读取 NPN 三极管的模拟信号来反映接入电路中的电阻的阻值。基于这个规律，我们想要制作一个阻值检测装置，它利用一个 LED 小灯来直观地展示电阻的大小。具体地，我们希望当接入的电阻阻值增大时，LED 小灯能够变得更亮；相反，当电阻阻值减小时，LED 小灯则变得较暗。

项目 - 换挡风扇

在这个教程中，我们将学习如何使用 Arduino 和电位器来控制一个风扇的转速。电位器是一种可变电阻器，它可以输出一个模拟信号，这个信号的电压随着电位器旋钮的旋转而改变。

元件清单



硬件连接

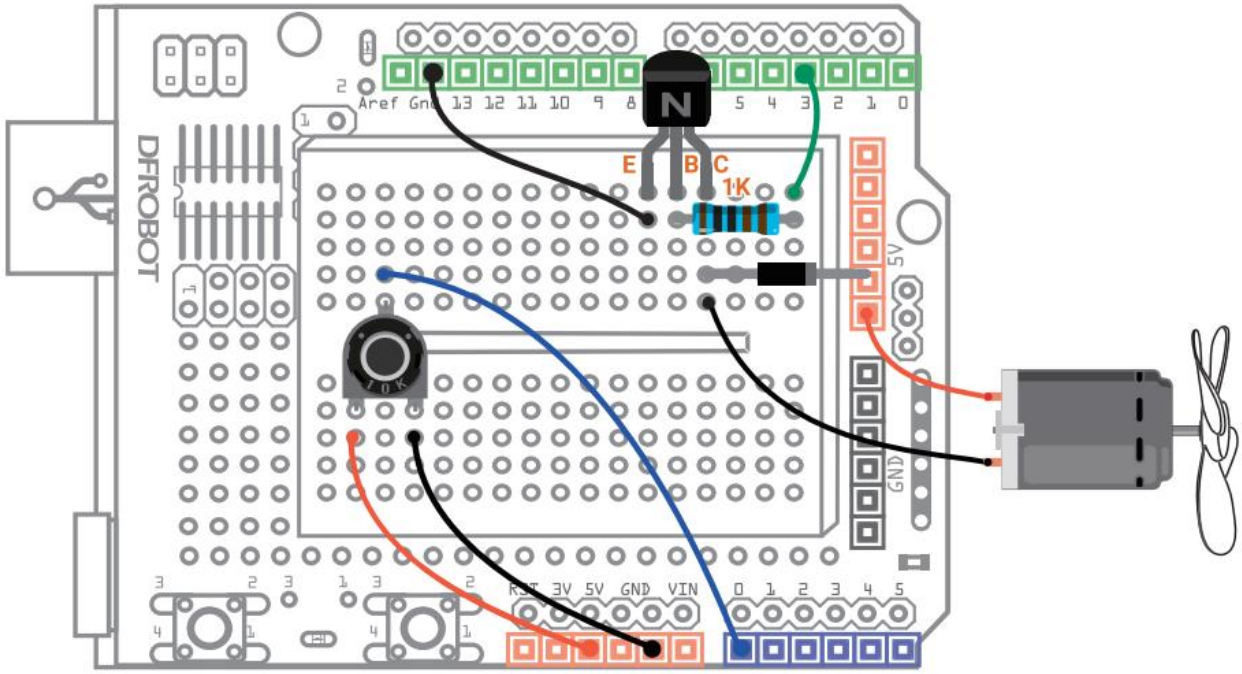


图 1 换挡风扇连线图

示例代码

样例代码:

```
//项目 - 换挡风扇

// 定义引脚编号

const int switchin = 0;

const int fanout = 3;

int switchval = 0;          // 存储从电位器读取的模拟值

int outputValue = 0;        // 存储映射后的值，用于控制风扇转速


void setup() {
```

```
  Serial.begin(9600);   // 初始化串行通信
}

void loop() {
  switchval = analogRead(switchin);    // 读取电位器的模拟值
  Serial.println(switchval);    // 通过串行监视器输出读取的值

  outputValue = map(switchval, 0, 1023, 0, 255);
// 将模拟值映射到 0-255 的范围内

  analogWrite(fanout, outputValue);     // 使用 PWM 控制风扇转速
}
```

代码上传成功后，转动电位器，感受小马达的转速的变化。

代码回顾

使用电位器控制风扇转速是一个典型的输入-计算-输出的过程控制流程。

输入：先从电位器连接的模拟引脚读取模拟值：

```
  analogRead(switchin)
```

计算：将读取的模拟值（0-1023）映射到一个新的范围（0-255），这个范围

适合 analogWrite()函数使用。

```
outputValue = map(switchval, 0, 1023, 0, 255)
```

输出：通过 PWM 信号控制连接到 fanout 引脚的风扇的转速。outputValue 决定了 PWM 信号的占空比，从而控制风扇的转速。

```
analogWrite(fanout, outputValue)
```

示例代码简明易懂，其他请结合注释自行理解。

硬件回顾

在之前的项目中，我们学习过使用电位器控制舵机的转动以及蜂鸣器的音调，在项目【可控舵机】和【DJ 调音台】中的硬件连接中，舵机和蜂鸣器的电路中并没有连入多余的电子元件。但在本项目中，小马达电路中还连入了 NPN 三极管和二极管（如下图）。为什么不能像蜂鸣器一样直接连入电路呢？接着会详细介绍 NPN 三极管和二极管在小马达电路中的作用。

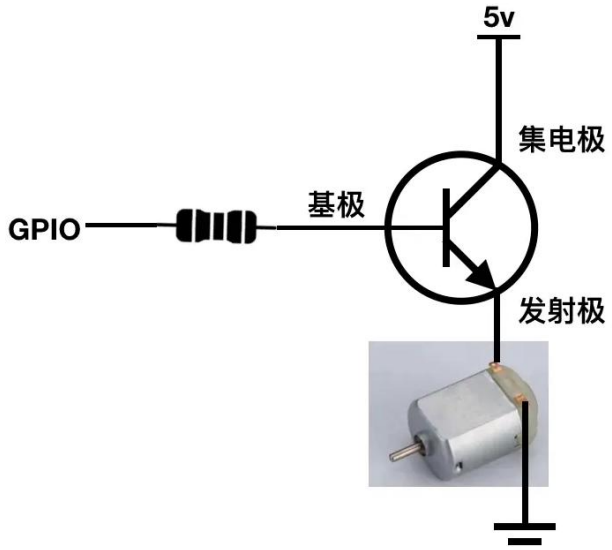


图 2 130 小马达驱动方式

NPN 三极管的作用

Arduino UNO R3 的主控板中，其每路输入/输出引脚的直流电流最大约为 20mA，而 130 小马达的空载电流最少需要 100mA。因此 Arduino 若想控制直流电机，只能采取放大电路驱动的方式。于是我们采用 NPN 三极管来驱动它（NPN 三极管相关知识见项目【指尖开关】）。

续流二极管

续流二极管，保护电路免受由电机等感性负载引起的电压尖峰和反向电压的影响。当电流突然被中断时，感性负载会产生感应电动势，这会导致反向电压的产生，这可能会损坏其他电路元件（比如三极管）。续流二极管并联连接在线圈（小马达）两端，当流过线圈中的电流突然中断时，线圈中产生的感应电动势会驱动电流通过二极管回流，从而防止电流冲击其他电路元件并消耗掉感应能量，以此防止潜在的损坏。起这种作用的二极管叫续流二极管。它本质上还

是二极管，但在这里主要起续流作用。

在本项目的电路中，当 130 小马达电机断电时，线圈会产生感应电动势并产生反向电压，该电压可能造成电路中元件（如三极管）的损坏。为防止这一风险，需在电机两端并联二极管以吸收此反向电压，保护电路。

如何区分电子元件的正负极呢？在本项目中，使用的二极管示意图如下，其中灰色环标注的一端为负极，另一端则为正极。

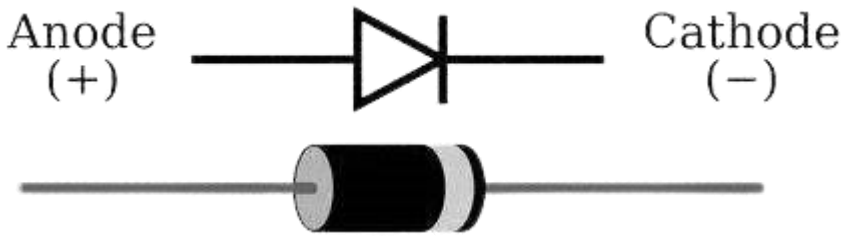
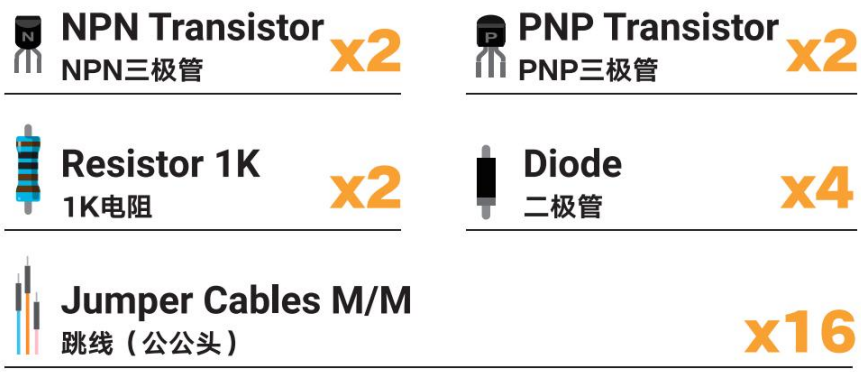


图 3 二极管正负极

项目 - H 桥电路驱动

该项目的学习重心在于理解并运用简单电路-H 桥电路来控制风扇的转动与停止，虽然程序中的语句多为基础且常见的 Arduino 语句，但理解这些语句如何与电路原理相结合是关键。

元件清单



硬件连接

请按照图 1 连接好硬件，本项目中使用到了单独的扩展面包板。在连接二极管时，注意灰色圆环的方向，在连接三极管时，看清方向（连接后对引脚进行核对）。

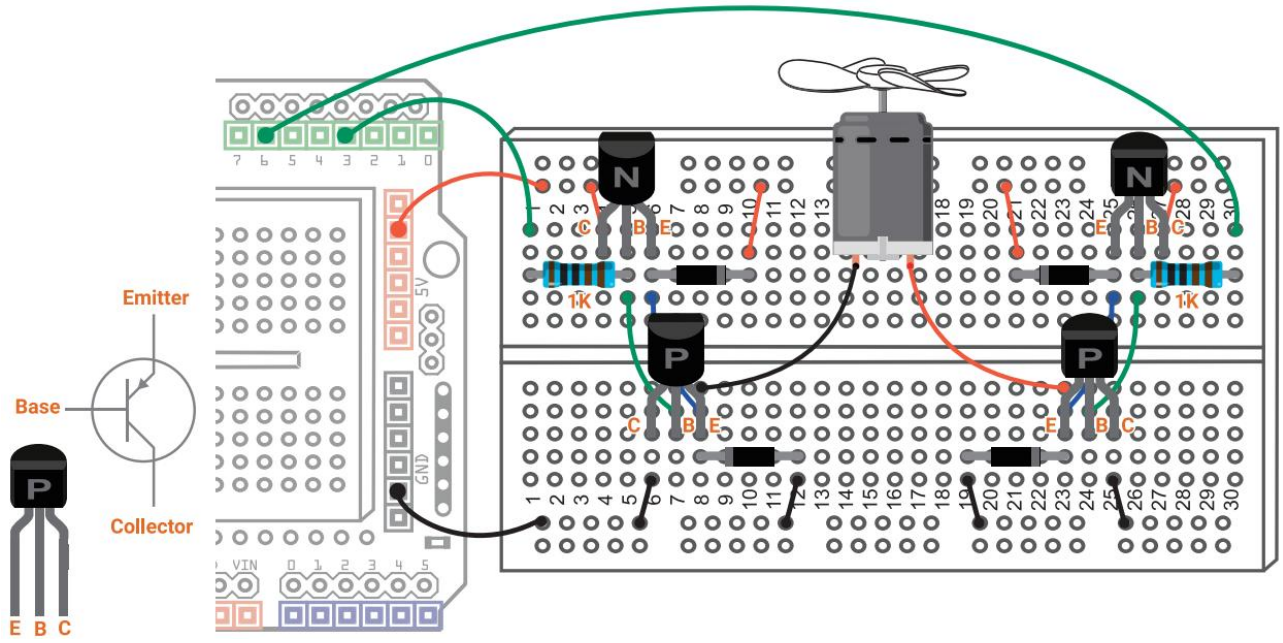


图 1 H 桥电路驱动

示例代码

```
// 项目 - H 桥电路驱动

const int fanPin1 = 3;

const int fanPin2 = 5;

void setup() {


pinMode(fanPin1, OUTPUT);

pinMode(fanPin2, OUTPUT);

}


void loop() {
```

```
digitalWrite(fanPin1, HIGH);
digitalWrite(fanPin2, LOW);
delay(1000);
digitalWrite(fanPin1, LOW);
digitalWrite(fanPin2, HIGH);
delay(1000);
}
```

上传程序后，风扇将会沿着一个方向转一秒钟，接着转向另一个方向再转一秒钟，这个过程会不断重复，在两个方向之间交替旋转。

在本文档中，我们先回顾相关的硬件组件及其作用，随后，结合电路原理详细解释程序部分。

硬件回顾

H 桥电路驱动

H 桥是一个比较简单的电路，通常它会包含四个独立控制的开关元器件（三极管），它们通常用于驱动电流较大的负载，比如电机，至于为什么要叫 H 桥，是因为长得比较像字母 H，如图 2 所示。

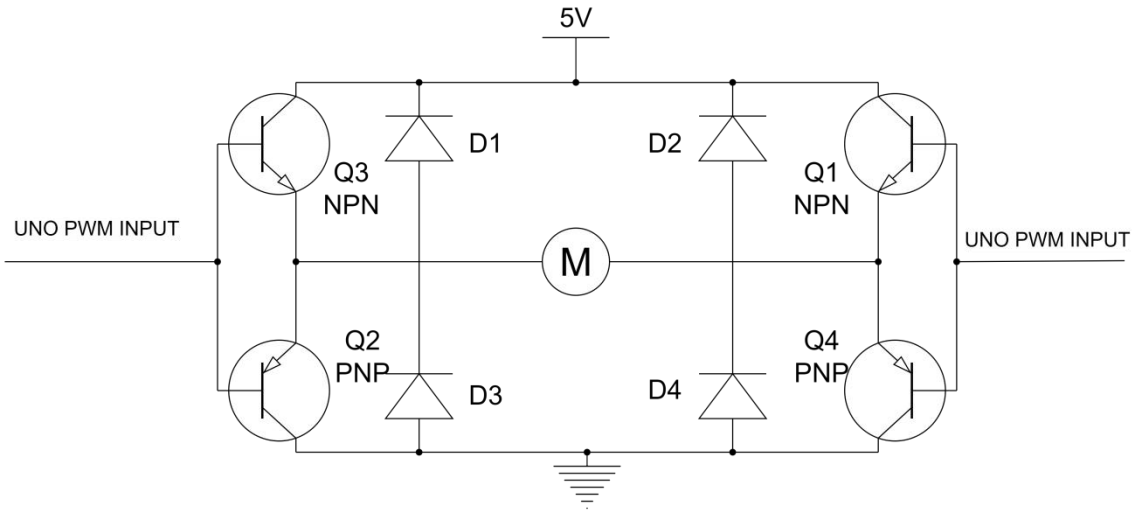
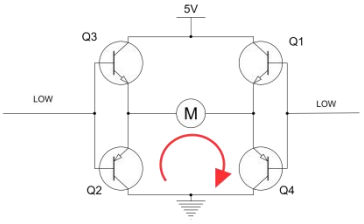


图 2 H 桥电路

这里有四个三极管 Q1，Q2，Q3，Q4，以及四个续流二极管 D1，D2，D3，D4，另外还有一个直流电机 M。这里的并联的 4 个续流二极管是为了防止小马达的尖峰电压对三极管元件造成的损伤（详见项目【指尖开关】）。

本项目的电路中使用到了 NPN 和 PNP 型两种三极管，两者在使用过程中可做开关使用，然而两者导通的基极点位却不相同，NPN 型可看作高电平导通，而 PNP 型可看作低电平导通。该电路中 Q1 和 Q3 为 NPN 型-高电平导通，而 Q2 和 Q4 为 PNP 型-低电平导通。

Q2Q3 输入状态	Q1Q4 输入状态	简易图示	马达状态
低电平	低电平		停止

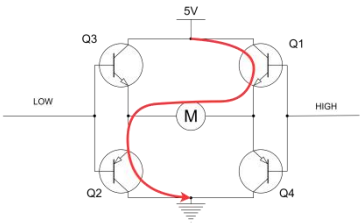
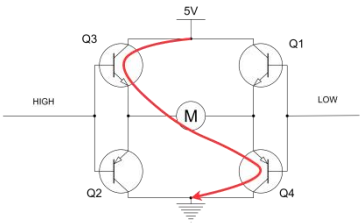
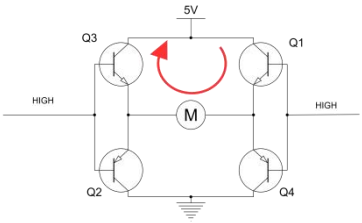
低电平	高电平		正转
高电平	低电平		反转
高电平	高电平		停止

表 1 H 桥电机控制逻辑

（上表格中的正转和反转取决于马达的正负极连接方向，仅针对本项目的硬件连接图）

下面的代码回顾将结合上面的电路知识对程序做一个梳理。

代码回顾

在 loop()函数中：

```
digitalWrite(fanPin1, HIGH);
digitalWrite(fanPin2, LOW);
delay(1000);
digitalWrite(fanPin1, LOW);
digitalWrite(fanPin2, HIGH);
delay(1000);
```

通过向 fanPin1 和 fanPin2 写入不同电平信号来控制电机的旋转。

当 fanPin1 设置为高电平（HIGH）且 fanPin2 设置为低电平（LOW）时，根据 H 桥电路的工作原理，电机将向一个方向旋转。delay(1000)使电机保持这个方向旋转 1 秒钟。

随后，fanPin1 被设置为低电平（LOW）且 fanPin2 被设置为高电平（HIGH），这将导致电机将往相反的方向旋转。同样，电机保持这个方向旋转 1 秒钟。

这个过程会不断重复，使电机在两个方向之间交替旋转。

课后练习

本项目的实现效果是电机在两个方向交替旋转，但实际上电机还有第三种状态：停止状态。根据硬件回顾中的电路驱动原理，尝试让电机实现转动一秒停止一秒，并不断重复这个过程。

项目 - 声控灯

声控灯是一种非常常见的家居设备，它能够通过检测周围环境的声音来自动开关灯光。本项目通过编写 Arduino 代码，我们可以实现当声音达到一定级别时，LED 灯自动亮起；声音减弱后，LED 灯自动熄灭的功能。

元件清单

硬件连接

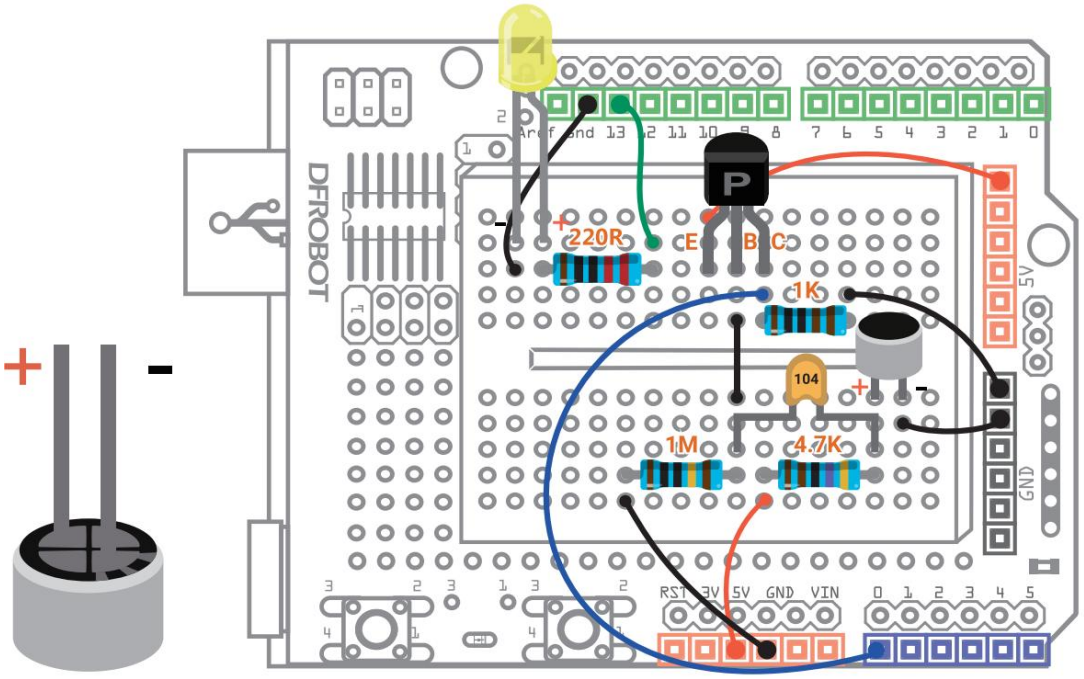


图 1 声控灯连线图

示例代码

样例代码:

```
//项目 - 声控灯

// 定义声音传感器和 LED 的引脚

const int soundSensorPin = A0; // 声音传感器连接到 A0

const int ledPin = 13;          // LED 连接到数字 13 号引脚


const int soundThreshold = 500; // 声音阈值，可以根据实际情况调整


void setup() {
```

```
  pinMode(soundSensorPin, INPUT);  // 设置声音传感器引脚为输入模式
  pinMode(ledPin, OUTPUT);         // 设置 LED 引脚为输出模式
  digitalWrite(ledPin, LOW);       // 确保启动时 LED 是关闭的

  Serial.begin(9600); // 初始化串口通信，并设置波特率为 9600
}

void loop() {
  // 读取声音传感器的值
  int sensorValue = analogRead(soundSensorPin);

  // 通过串口监视器打印声音传感器的值
  Serial.print("声音传感器的值: ");
  Serial.println(sensorValue);

  // 检查声音是否超过设定的阈值
  if (sensorValue > soundThreshold) {
    digitalWrite(ledPin, HIGH); // 如果超过阈值，点亮 LED
    delay(1000);
  } else {
    digitalWrite(ledPin, LOW);  // 如果没超过阈值，熄灭 LED
  }
```

```
  // 为了避免连续检测，可以在此添加短暂的延时
  delay(50); // 延时 50 毫秒
}
```

代码上传成功后，当你发出较大声音时，LED 会亮起并维持一秒钟。如果发现比较难以激活 LED（或过于敏感），可以适当调整声音阈值以达到最佳效果。

代码回顾

在 loop()函数开始之前，主要执行的是一些基础的准备工作，包括初始化用于控制 LED 的引脚，以及设置串口通信参数以便进行数据传输或调试。

在该项目的 loop()函数中，展现了典型的输入、处理、输出的控制流程。

输入，读取声音传感器的值并存放在 sensorValue 中：

```
  int sensorValue = analogRead(soundSensorPin);
```

计算处理，我们使用一个 if 语句来检查声音值是否超过了预设的阈值 soundThreshold：

```
  if (sensorValue > soundThreshold) {
      digitalWrite(ledPin, HIGH);
      delay(1000);
```

```
    } else {
      digitalWrite(ledPin, LOW);
    }
```

输出，如果超过了，就使用 digitalWrite(ledPin, HIGH) 点亮 LED，并通过 delay(1000) 保持 LED 点亮 1 秒钟。如果没有超过阈值，就使用 digitalWrite(ledPin, LOW) 熄灭 LED。

delay(50)作用是在每次检测声音之间加一个短暂的延时，这有助于减少 Arduino 的负担并提高程序的稳定性。

硬件回顾

驻极体麦克风

驻极体麦克风的基本原理就是一个可变电容，它的电容值随着声音震动而变化。这样就将声音信号转变为了电信号。在电路中，把它看作是一个随声音变化的电阻即可。

驻极体麦克风是极化的，因此它有一个正极引脚和一个接地引脚：

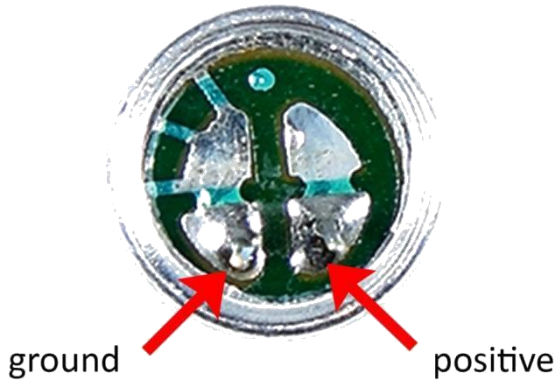


图 2 驻极体麦克风引脚

连接到金属外壳的引脚是接地（ground）引脚，另一个引脚是正极（positive）引脚。

麦克风产生的音频信号是一种交流电，类似于家中电源插座中的电流。但是，交流电流是具有静态频率和固定波长的正弦波，但音频波是频率和波长是可变的。

较高频率的音频信号会产生更高音调的声音。较低频率的音频信号会产生较低音调的声音：

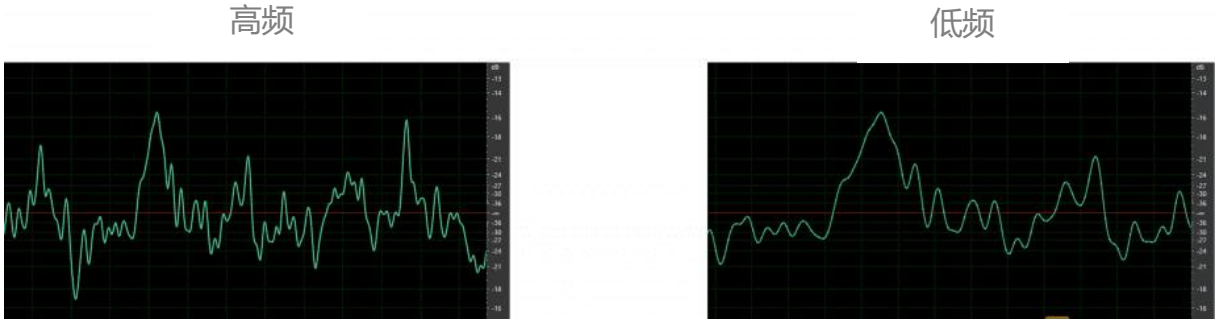


图 3 频率

声音的“音量”或“响度”与峰值的幅度直接相关。振幅较大的音频信号会产生更大音量的声音。振荡幅度较小的音频信号会产生音量较小的声音。

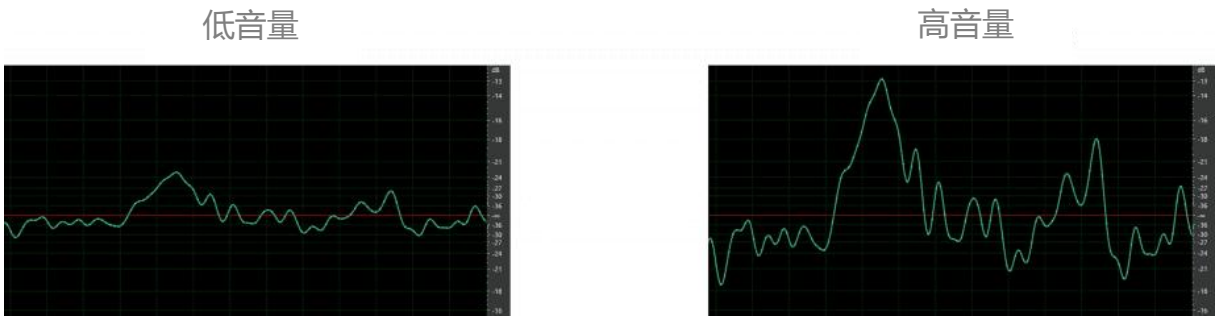


图 4 振幅

偏置电阻

为了保证麦克风的输出信号处于所要求的工作区间，因此我们需要给其添加一个直流偏置，让其输出的电压信号处于所需工作区间。

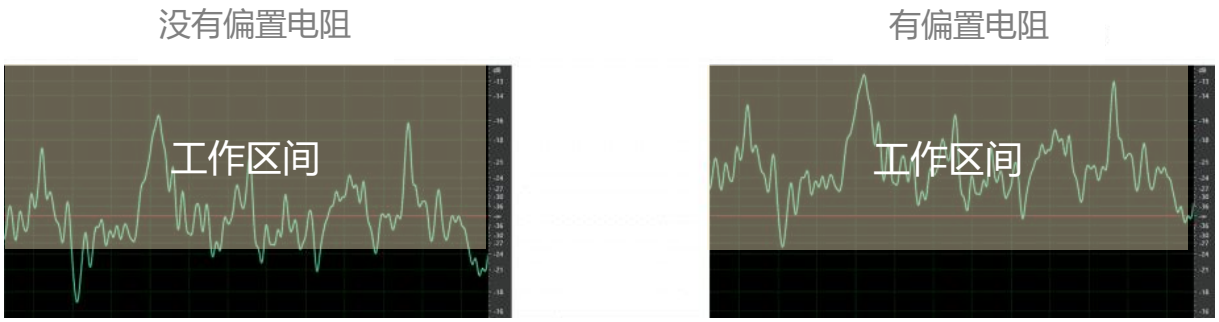


图 5 工作区间

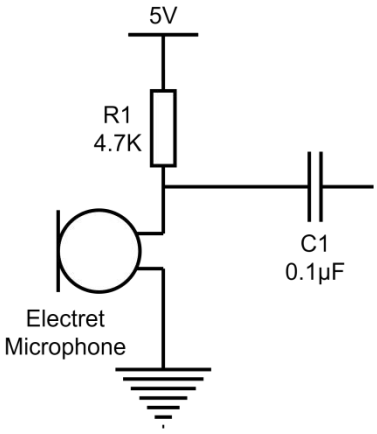


图 6 麦克风部分电路

就像图 6 这样，直流偏置的本质是通过加入电阻 R1 与麦克风分压，然后用一个

电容 C1 隔离偏置电阻产生的直流电压部分。

PNP 放大电路

接下来，我们来看一下放大电路部分，如图 7 所示：

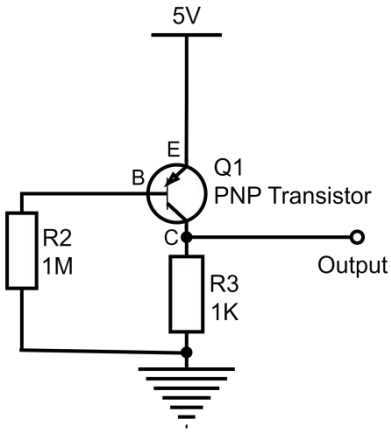


图 7 放大电路部分

从项目【指尖开关】中对 NPN 三极管的介绍来看，NPN 和 PNP 型三极管是可以当做开关来使用的，也可以当做放大器来使用，当它的基极，产生一个小变化，输出信号就会有很大的变化。这就是所谓的放大功能。由于麦克风通过使用 R1 分压形成的电压信号较微弱，因此使用三极管放大电路来获得比较好的效果。

基极电阻 R2 在 PNP 型三极管放大电路中起到了偏置的作用，而集电极电阻 R3 则作为负载元件，实现了电压的放大和电流到电压的转换。这两个电阻共同协作，确保了放大电路的正常工作和信号的有效放大。

那么 PNP 型三极管和 NPN 型三极管在作为放大电路时有什么不同吗？

由于这两种三极管的材料层序和电流流动的方向不同，导致它们在放大电路中的工作原理和特性有所不同。

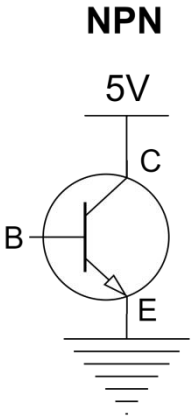


图 8 NPN 型三极管

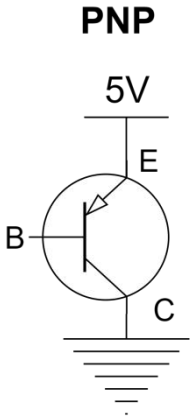


图 9 PNP 型三极管

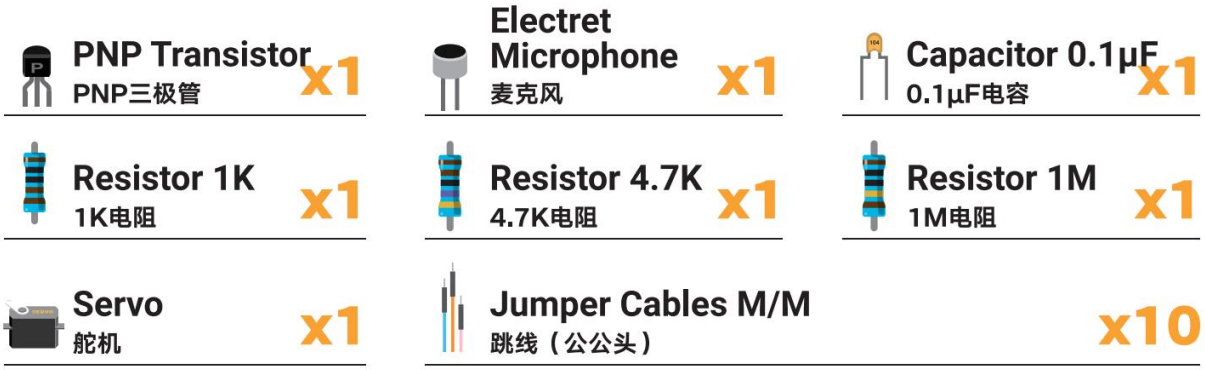
(C: 集电极; B: 基极; E: 发射极)

如图 8 和图 9 所示：在 NPN 电路中，C 接正极，E 接负极；PNP 电路则相反，E 接正极，C 接负极（这里说的正极和负极都是直接或间接连接）。一般的电路中，用了 NPN 的，你就可以按“上下对称交换”的方法得到 PNP 的版本。当然，要保证正常工作，还必须保证这些电压、电流满足一些定量条件，需要调整电阻的接入位置来保证电路的正常工作和信号的有效放大。

项目 - 指针式噪音计

在这个项目中，我们将学习如何使用 Arduino、麦克风和舵机来创建一个有趣的互动装置。将声音强度信息通过 Arduino 转换为舵机的旋转角度，使得舵机的叶片能够随着声音的大小而转动，从而直观地展示声音的变化。

元件清单



硬件连接

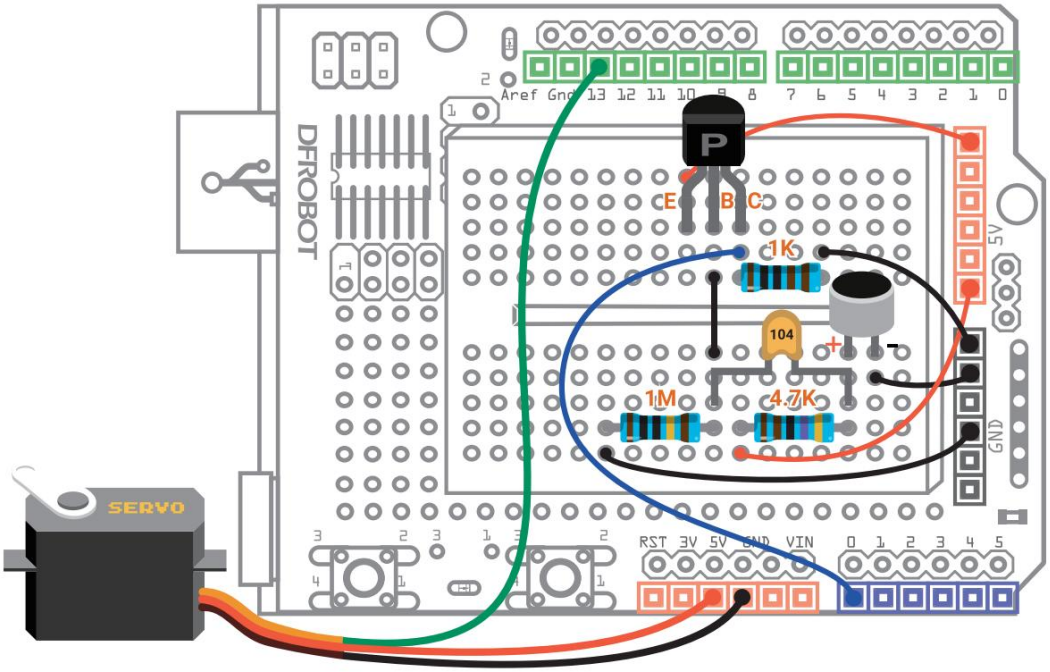


图 1 指针式噪音计

示例代码

样例代码:

```
//项目 - 指针式噪音计

#include <Servo.h>

// 定义声音传感器

const int soundSensorPin = A0; // 声音传感器连接到 A0

Servo myservo;  // 创建舵机对象


const float alpha = 0.1;

// 滤波因子，介于 0 和 1 之间，数值越小滤波效果越强

float filteredValue = 0;  // 经过滤波后的声音传感器值
```

```
void setup() {
  pinMode(soundSensorPin, INPUT);
  Serial.begin(9600);
  myservo.attach(13);
}

void loop() {
  // 读取声音传感器的值
  int sensorValue = analogRead(soundSensorPin);
  Serial.println(sensorValue);
  // 应用指数移动平均滤波器
  filteredValue = alpha * sensorValue + (1 - alpha) * filteredValue;

  // 通过串口监视器打印滤波后的声音传感器值
  Serial.print("滤波后的声音值: ");
  Serial.println(filteredValue);

  // 将滤波后的声音值映射到舵机的位置范围（通常是 0 到 180 度）
  int servoPos = map(filteredValue, 0, 1023, 0, 180);

  // 控制舵机转动到指定位置
  myservo.write(servoPos);
```

```
  // 延时一小段时间，以便于舵机稳定
  delay(100);
}
```

上传代码后，对着麦克风发出声音，舵机的叶片会发生转动。根据声音的音量大小，转动的角度会有所不同。

代码回顾

由于麦克风传输出的信号不是稳定平滑的，当我们把信号直接使用在舵机上时，会发现即使环境声音没有变化，舵机也处于抖动状态，实现效果不够稳定。

所以，在使用驻极体麦克风模块和舵机进行音量显示时，引入滤波机制是非常重要的，它可以提高系统的准确性和稳定性，提升用户体验。

首先，我们需要定义滤波器的滤波因子（Alpha）值。

```
  const float alpha = 0.1;
```

Alpha 值介于 0 和 1 之间，决定了滤波器对输入数据的敏感程度以及滤波效果的平滑程度。

```
  filteredValue = alpha * sensorValue + (1 - alpha) * filteredValue;
```

上面的计算实现了指数移动平均滤波算法，以平滑声音数据。

其中，filterValues 是滤波后的值，也就是当前时间点的输出值。它随时间更新，以反映最新的输入数据（即 sensorValue）与过去的滤波值之间的加权平均值。这里的权重就是滤波因子 Alpha。Alpha 越大，滤波后的结果对最新的输入数据（sensorValue）的响应越快，滤波结果越能反映近期的数据变化；反之，Alpha 越小，滤波后的结果越平滑，但对输入数据的响应也越慢。

```
int servoPos = map(filteredValue, 0, 1023, 0, 180);
myservo.write(servoPos);
```

最后，将处理好的数据映射成舵机工作区间的角度 servoPos，并使用 myservo.write()函数将声音响度以舵机角度的方式呈现出来。

硬件回顾

本项目使用的硬件和电路都与项目【声控灯】相似，如有不清楚的地方，请参考项目【声控灯】的教程。

课后练习

如在实际使用过程中，舵机转动效果不理想，请尝试修改滤波因子的数值，实现舵机的平滑转动吧！

项目 - 点亮点阵

在生活中，点阵屏是一种常用的显示设备，它能够以矩阵形式展示信息，如数字、字母或简单的图形。通过控制点阵屏上的每一个点（LED）的亮灭，我们可以创造出各种视觉效果。今天，我们将通过一个 Arduino 项目来学习如何控制一个 8x8 的点阵屏，包括如何初始化硬件、编写代码来点亮所有 LED、以及实现扫描显示技术。

元件清单

8x8 LED Matrix
8x8 LED点阵
x1

Resistor 4.7K
4.7K电阻
x8

Jumper Cables M/M
跳线（公公头）
x16

硬件连接

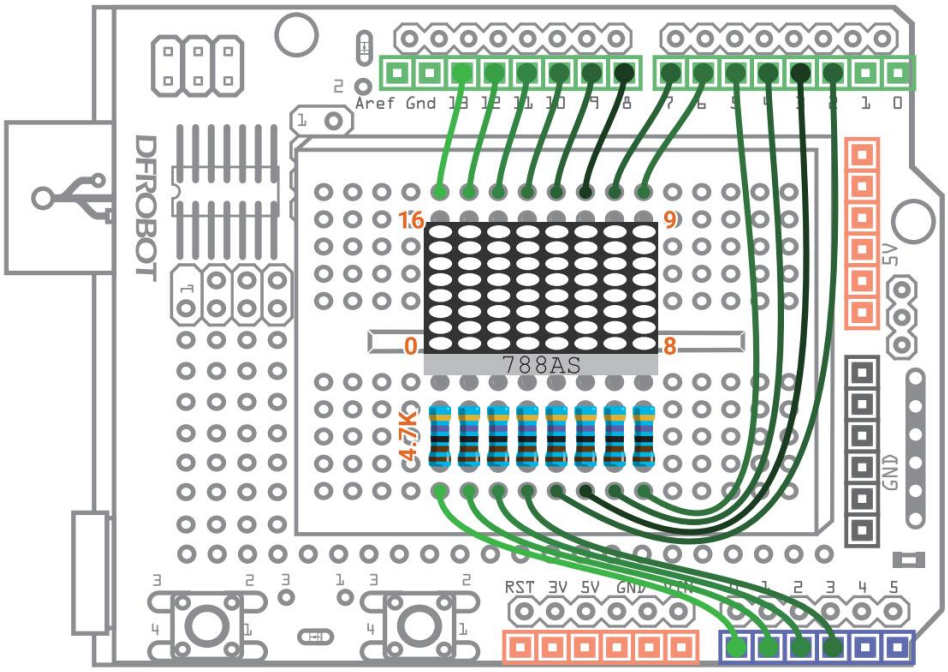


图 1 点亮点阵连线图

示例代码

样例代码:

```
//项目 - 点亮点阵
const int rowPins[8] = {6, 11, 5, 9, A0, 4, A1, 2}; // 点阵屏行引脚（负极）
const int colPins[8] = {10, A2, A3, 7, 3, 8, 12, 13}; // 点阵屏列引脚（正极）


void setup() {
  for (int i = 0; i < 8; i++) {
```

```
    pinMode(colPins[i], OUTPUT);
    pinMode(rowPins[i], OUTPUT);
    digitalWrite(rowPins[i], HIGH); // 初始化行引脚为高电平（关闭）
  }
}

void turnOnAllLeds() {
  for (int i = 0; i < 8; i++) {
    digitalWrite(colPins[i], HIGH);
    digitalWrite(rowPins[i], LOW);
  }
}

void turnOffAllLeds() {
  for (int i = 0; i < 8; i++) {
    digitalWrite(colPins[i], LOW);
    digitalWrite(rowPins[i], HIGH);
  }
}

void scanRows() {
  digitalWrite(colPins[0], HIGH);
  for (int col = 0; col < 8; col++) {
```

```
    digitalWrite(rowPins[col], LOW);
    delay(500);
  }
  turnOffAllLeds();
}

void scanColumns() {

  for(int col = 0; col < 8; col++){
    digitalWrite(colPins[col], HIGH);
    for(int row = 0; row < 8; row++){
      digitalWrite(rowPins[row], LOW);
    }
    delay(500);
    digitalWrite(colPins[col], LOW);
  }

}

void loop() {
  turnOnAllLeds();
  delay(1500);
  turnOffAllLeds();
```

```
  delay(1500);
  scanRows();
  scanColumns();
}
```

将程序上传成功后，可以看到点阵屏闪烁一次，接着第一列的 LED 从上到下依次亮起，之后点阵屏按照列为单位从左至右依次亮起，并不断重复所有上述效果。

代码回顾

在开始学习如何使用代码来控制 8*8 LED 点阵屏之前，建议您看一下硬件回顾部分，了解 8*8 LED 点阵屏的基本原理。这样，您就能更容易地理解和上手本项目中代码所演示的点阵屏点亮效果。

```
  const int rowPins[8] = {6, 11, 5, 9, A0, 4, A1, 2};
  const int colPins[8] = {10, A2, A3, 7, 3, 8, 12, 13};
```

开头这部分代码定义了两组引脚数组，rowPins 和 colPins，分别用于连接点阵屏的行和列。

```
  void setup() {
    for (int i = 0; i < 8; i++) {
```

```
    pinMode(colPins[i], OUTPUT);
    pinMode(rowPins[i], OUTPUT);
    digitalWrite(rowPins[i], HIGH);
  }
}
```

setup() 函数在 Arduino 板上电后只运行一次。

使用 for 循环将 colPins 和 rowPins 数组中的引脚设置为输出模式。

将所有行引脚设置为高电平（HIGH）确保点阵屏在初始化时处于关闭状态。

```
void turnOnAllLeds() {
  for (int i = 0; i < 8; i++) {
    digitalWrite(colPins[i], HIGH);
    digitalWrite(rowPins[i], LOW);
  }
}
```

turnOnAllLeds() 函数用 for 循环将所有列引脚设置为高电平（HIGH），并将所有行引脚设置为低电平（LOW），所有的 LED 灯将被点亮。

```
void turnOffAllLeds() {
  for (int i = 0; i < 8; i++) {
    digitalWrite(colPins[i], LOW);
```

```
    digitalWrite(rowPins[i], HIGH);
  }
}
```

turnOffAllLeds() 函数通过将所有列引脚设置为低电平（LOW）并将所有行引脚设置为高电平（HIGH），关闭点阵屏上的所有 LED。

```
void scanRows() {
  digitalWrite(colPins[0], HIGH);
  for (int row = 0; row < 8; row++) {
    digitalWrite(rowPins[row], LOW);
    delay(500);
  }
  turnOffAllLeds();
}
```

scanRows() 函数是扫描行，将第一列的 LED 灯从第一行到第八行依次亮起。

```
void scanColumns() {
  for(int col = 0; col < 8; col++){
    digitalWrite(colPins[col], HIGH);
    for(int row = 0; row < 8; row++){
      digitalWrite(rowPins[row], LOW);
```

```
    }
    delay(500);
    digitalWrite(colPins[col], LOW);
  }
}
```

scanColumns() 函数是扫描列，使用两个 for 循环，外层 for 循环用于遍历所有的列，内层嵌套 for 循环用于点亮该列所有 LED。

```
void loop() {
  turnOnAllLeds();
  delay(1500);
  turnOffAllLeds();
  delay(1500);
  scanRows();
  scanColumns();
}
```

loop() 函数是 Arduino 程序的主体循环部分。

尝试打开所有 LED，等待 1.5 秒，然后关闭所有 LED，再等待 1.5 秒。

接着调用 scanRows()和 scanColumns()函数，实现依次扫描行和扫描列的效果。

硬件回顾

8*8 LED 点阵屏

本项目中使用的点阵屏规格型号为 788AS，内部电路图如图 2 所示。从图 2 可以看出 LED 显示屏为 8*8 的点阵屏，由 64 个发光二极管组成，且每个发光二极管是放置在行线和列线的交叉点上，当对应的某一列置高电平，某一行置低电平，则相应的二极管就亮。

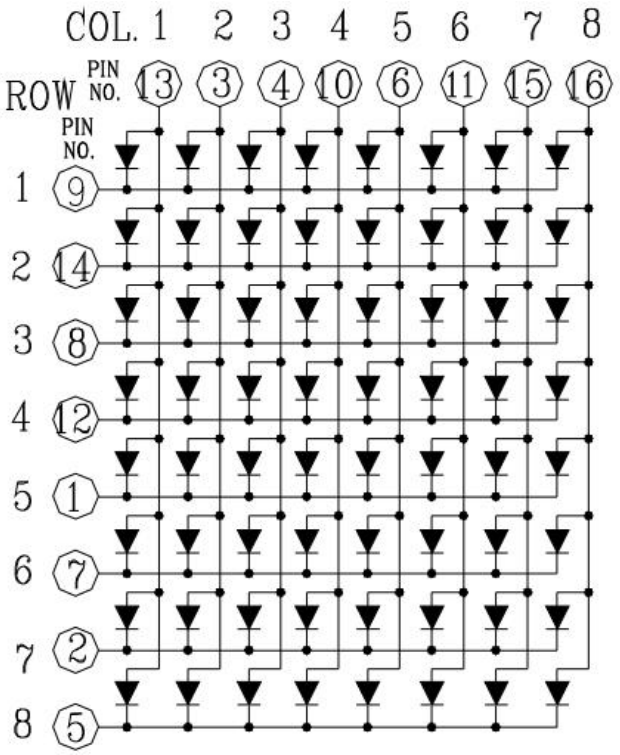


图 2 内部电路图

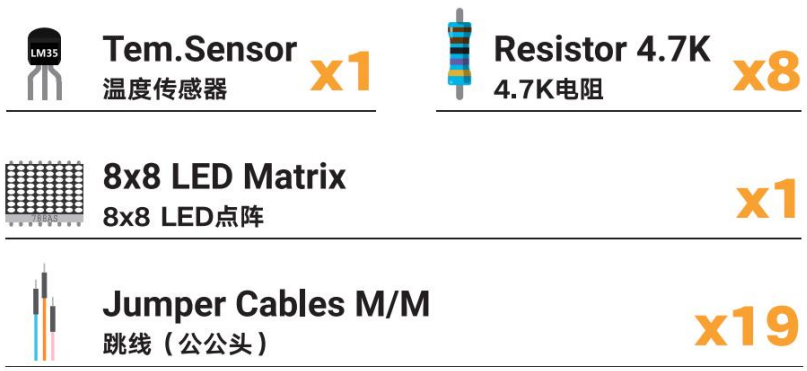
课后练习

本项目中是先扫描行再扫描列，请尝试对程序进行更改，实现先扫描列再扫描行。

项目 - 温度计

在这个项目中，我们将利用 Arduino 开发板和 8x8 点阵屏，将温度传感器 LM35 采集的温度数据以生动、直观的方式呈现出来。想象一下，随着环境温度的变化，点阵屏上滚动显示的温度值也随之更新，仿佛一个实时报告环境状况的微型显示屏。这不仅仅是一个技术实践的过程，更是一次将科技融入生活，用代码赋予物品新生命的有趣尝试。现在，就让我们共同开启这场数字与现实的交互之旅吧！

元件清单



硬件连接

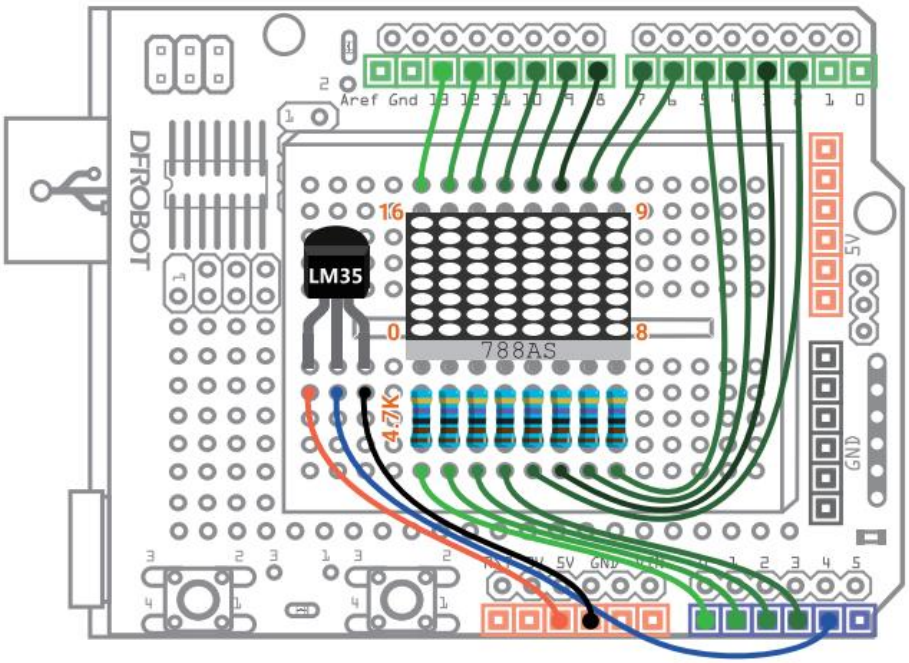


图 1 温度计连线图

按照上述连线图进行硬件连接，注意温度传感器和点阵屏的连接方向。

示例代码

样例代码:

```
//项目-温度计

#define TEMP_SENSOR_PIN A4 // LM35 温度传感器连接到 A4 引脚


const int rowPins[8] = {6, 11, 5, 9, A0, 4, A1, 2};

// 点阵屏行引脚（负极）

const int colPins[8] = {10, A2, A3, 7, 3, 8, 12, 13};

// 点阵屏列引脚（正极）


// 定义数字 0-9 的 8x8 点阵图案

byte numbers[10][8] = {
```

```
// 0
{0b00111100,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b00111100},
// 1
{0b00011000,
0b00111000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b11111110},

// 2
{0b00111100,
0b01100110,
0b00000110,
```

```
0b00001100,
0b00110000,
0b01100000,
0b01100110,
0b01111110},

// 3
{0b00111100,
0b01100110,
0b00000110,
0b00011100,
0b00000110,
0b00000110,
0b01100110,
0b00111100},

// 4
{0b00001100,
0b00011100,
0b00101100,
0b01001100,
0b01111110,
0b00001100,
```

```
0b00001100,
0b00001100},

// 5
{0b01111110,
0b01100000,
0b01100000,
0b01111100,
0b00000110,
0b00000110,
0b01100110,
0b00111100},

// 6
{0b00111100,
0b01100110,
0b01100000,
0b01111100,
0b01100110,
0b01100110,
0b01100110,
0b00111100},
```

```
// 7
{0b01111110,
0b01100110,
0b00001100,
0b00011000,
0b00110000,
0b00110000,
0b00110000,
0b00110000},

// 8
{0b00111100,
0b01100110,
0b01100110,
0b00111100,
0b01100110,
0b01100110,
0b01100110,
0b00111100},

// 9
{0b00111100,
0b01100110,
```

```
0b01100110,
0b00111110,
0b00000110,
0b00000110,
0b01100110,
0b00111100}
};
// 读取温度传感器的值并转换为摄氏度
float readTemperature() {
  int reading = analogRead(TEMP_SENSOR_PIN);
  float voltage = reading * 5.0 / 1023.0;
  float temperature = voltage * 100.0; // 将电压转换为温度
  return temperature;
}

void combineNumbers(byte first[], byte second[], byte combined[],
int width) {
  for (int i = 0; i < width; i++) {
    combined[i] = first[i];
  }
  for (int i = 0; i < width; i++) {
    combined[i + width] = second[i];
  }
```

```
}

void displayScrollingNumber(byte number[], int combinedWidth) {
  int width = 8; // 单个数字的宽度

  /* 滚动的总步数是组合后数字宽度加上点阵屏宽度，这样两个数字可以从右边完全不可见到左边完全不可见*/
  for (int offset = width; offset >= -combinedWidth; offset--) {
    for (int row = 0; row < width; row++) {
      digitalWrite(rowPins[row], LOW); // 激活当前行
      for (int col = 0; col < width; col++) {
        int colIndex = col - offset; // 注意这里是减去 offset
        if (colIndex >= 0 && colIndex < combinedWidth) {
          // 设置数字的列值，注意这里改为从左侧开始读取位
          digitalWrite(colPins[col], bitRead(number[row + ((colIndex / width) * width)], width - 1 - (colIndex % width)) ? HIGH : LOW);
        } else {
          // 如果列索引超出范围，则关闭对应的列
          digitalWrite(colPins[col], LOW);
        }
      }
      // 控制滚动速度
```

```
      delay(8);
      digitalWrite(rowPins[row], HIGH);
      // 关闭当前行，准备激活下一行
    }
  }
}

void setup() {
  Serial.begin(9600);
  for (int i = 0; i < 8; i++) {
    pinMode(colPins[i], OUTPUT);
    pinMode(rowPins[i], OUTPUT);
    digitalWrite(rowPins[i], HIGH); // 初始化行引脚为高电平（关闭）
  }

}

void loop() {
  byte combinedNumber[16]; // 假设我们只组合两个数字，所以大小是 16
  int temp = readTemperature();
  int tempDigits[2]; // 假设温度不会超过 99 度
  Serial.println(temp);
```

```
  tempDigits[0] = (temp / 10) % 10;
  tempDigits[1] = temp % 10;
  combineNumbers(numbers[tempDigits[0]], numbers[tempDigits[1]],
combinedNumber, 8);
  displayScrollingNumber(combinedNumber, 16);
// 传入组合后的数组和宽度
}
```

代码回顾

开始的三行代码对温度传感器和点阵屏的行和列所连接的引脚进行了定义。

```
  #define TEMP_SENSOR_PIN A4
  const int rowPins[8] = {6, 11, 5, 9, A0, 4, A1, 2};
  const int colPins[8] = {10,A3, A2, 7, 3, 8, 12, 13};
```

在二维数组 numbers[10][8] 中，numbers 是一个包含 10 个元素的数组，其中每个元素又是一个包含 8 个元素的一维数组（也叫这个数字的字模）。字模定义了每个数字在点阵屏上的显示方式，包含 8 个元素，对应 8x8 点阵屏的每一行。每个元素是一个 8 位的二进制数组（“0b”是一个前缀，用于明确指示随后的数字是二进制形式，不计入位长度），对应该行中的每一列。

```
byte numbers[10][8] = {
// 0
{0b00111100,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b00111100},
.....
// 9
{0b00111100,
0b01100110,
0b01100110,
0b00111110,
0b00000110,
0b00000110,
0b01100110,
0b00111100}
};
```

如果你想让点阵屏显示数字 0，你可以访问 numbers[0]来获取这个数字的字模（如下）。

```
{0b00111100,
 0b01100110,
 0b01100110,
 0b01100110,
 0b01100110,
 0b01100110,
 0b01100110,
 0b00111100}
```

接着通过以下步骤在点阵屏上显示出来：

1. 遍历点阵屏的每一行。
2. 对于每一行，激活该行的所有列。
3. 遍历每一列，根据数字 0 的一维数组中每个元素的位索引上的值，设置该列的 LED 是点亮还是熄灭。(比如在数组 0 号元素中，第 2-5 位索引的值为 1，其他为 0，效果为该行的第 3-6 列 LED 灯点亮)

   ```
   0b00111100,
   ```

4. 关闭当前行，准备显示下一行。
5. 重复以上步骤，直到所有行都被显示过一遍。

不断重复遍历操作，利用计算机的运行速度和人类的视觉暂留，就能在点阵屏上显示出数字 0 了。

现在大家对点阵屏如何显示数字有了一定的了解，但本项目中要进行实时的温度显示，通常我们生活的环境温度（摄氏度）不止为一位数字，更多的是两位数字组成的温度值，那么如何在这么小的点阵屏上表示两位数字呢?

通常会使用滚动的方式来进行多位数字的显示。接下来我们就一起来看看怎么让多位数字组合并滚动起来吧!

函数 readTemperature()用来读取当前的环境温度，相关计算过程在项目【温度报警器】中已经做了说明，这里就不赘述了。

在 setup()函数中，使用一个 for 循环将行和列引脚设为输出模式，并将所有行引脚设为高电平，即所有 LED 灯关闭状态。（点阵屏的相关知识请参考项目【点亮点阵】）

```
void setup() {
  Serial.begin(9600);
  for (int i = 0; i < 8; i++) {
    pinMode(colPins[i], OUTPUT);
    pinMode(rowPins[i], OUTPUT);
    digitalWrite(rowPins[i], HIGH); // 初始化行引脚为高电平（关闭）
  }

}
```

在 loop 函数中，在串口监视器中显示温度值：

```
int temp = readTemperature();
Serial.println(temp);
```

由于字模数组中只存储了 0-9 的字模，所以在显示温度时需要将其先拆分成为十位和个位，便于使用字模在点阵屏上显示。所以定义一个数组来存储个位和十位的数字：

```
int tempDigits[2]; // 假设温度不会超过 99 度，用来存储个位和十位
tempDigits[0] = (temp / 10) % 10;
tempDigits[1] = temp % 10;
```

接着，我们使用 combineNumbers()函数将两个数字的字模合并成一个长字模，然后在点阵屏上滚动显示这个长字模。

```
byte combinedNumber[16]; // 假设我们只组合两个数字，所以大小是 16
combineNumbers(numbers[tempDigits[0]], numbers[tempDigits[1]],
combinedNumber, 8);
displayScrollingNumber(combinedNumber, 16);
```

以上就是 loop 函数的全部内容了，下面将重点对 combineNumbers()和 displayScrollingNumber()两个函数进行详细的解释：

```
void combineNumbers(byte first[], byte second[], byte combined[],
int width) {
  for (int i = 0; i < width; i++) {
    combined[i] = first[i];
  }
  for (int i = 0; i < width; i++) {
    combined[i + width] = second[i];
  }
}
```

这段代码定义了一个名为 combineNumbers()的函数，其目的是将两个数字的字模（以字节数组形式表示）水平合并成一个新的字节数组。这个函数对于在点阵显示屏上并排显示两个数字非常有用。具体实现方式也非常简单，使用两个并列的 for 循环，将两个数字数组中的元素依次复制到 combined[]数组中，比如将数字 0 和 1 合并后的 combined[]数组如下：

```
{0b00111100,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
0b01100110,
```

```
  0b01100110,
  0b00111100,
  0b00011000,
  0b00111000,
  0b00011000,
  0b00011000,
  0b00011000,
  0b00011000,
  0b00011000,
  0b01111110}
```

最后，将合并后的数字滚动显示出来，滚动显示的逻辑如下：

外层循环（滚动步数）：

```
  for (int offset = width; offset >= -combinedWidth; offset--)
```

这个循环控制数字的滚动。offset 变量从数字的宽度开始（即数字完全在屏幕右侧外），递减到-combinedWidth（即数字完全在屏幕左侧外）。这样，数字从右向左滚动。

内层循环（行扫描）：

```
  for (int row = 0; row < width; row++)
```

这个循环遍历点阵屏的每一行。对于每一行，代码首先激活该行（将其设为低电平），然后设置该行的每一列。

内层循环（列设置）：

```
int colIndex = col - offset;
```

计算当前列相对于数字起始位置的索引。由于数字在滚动，所以使用 offset 来调整列索引。

```
if (colIndex >= 0 && colIndex < combinedWidth)
```

检查列索引是否在数字的有效范围内。

如果在有效范围内，使用：

```
bitRead(number[row + ((colIndex / width) * width)], width - 1 -
(colIndex % width))
```

来读取该位置的位值（0 或 1），并据此设置列的电平（高或低），从而点亮或熄灭 led。

否则，超出索引范围，使用 digitalWrite(colPins[col], LOW)关闭对应的列。

最后，使用 delay(8)和 digitalWrite(rowPins[row], HIGH)用来控制滚动速度和关闭当前航，为激活下一行做准备。

以上就是这个项目的代码回顾了，如果想使用点阵屏来显示其他的字符，可以尝试完成课后练习的部分哦。

课后练习

除了显示滚动的数字，点阵屏还可以显示其他种类的字符，比如汉字，但由于点阵屏 led 数量有限，不能合理的显示较复杂的汉字。英文似乎是个不错的选择，你可以尝试让点阵屏滚动显示单词吗？快来尝试一下吧！

项目 - 调光面板

在这个项目中，我们将探索 Processing 与 Arduino 的联合编程魅力，通过简单的界面交互实现创意的 RGB 灯光控制。想象一下，只需轻点 Processing 界面上的按钮，就能远程操控 Arduino 板上连接的 RGB LED 灯打开和关闭。这不仅是一次编程技能的实践，更是将数字创意融入日常生活的有趣尝试。让我们一起动手，用代码点亮生活的色彩吧！

元件清单

5MM RGB LED
5MM RGB LED灯 x1

Resistor 220R
220欧电阻 x3

Jumper Cables M/M
跳线（公公头） x4

硬件连接

按照下图进行硬件连接，RGB LED 的引脚图请参考项目【炫彩 LED】或【彩灯调光台】。

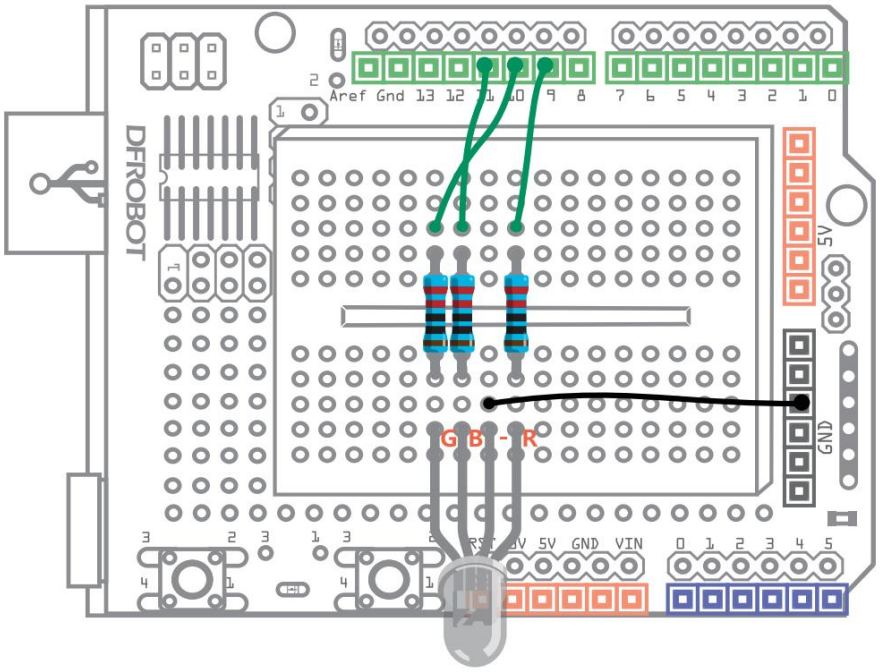


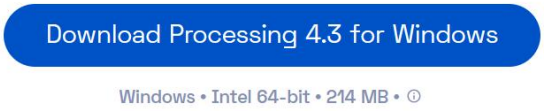
图 1 调光面板连线图

为了实现电脑上的 Processing 面板对 Arduino 控制的 RGB 炫彩灯进行颜色控制，首先需要确保安装了必要的软件。下面是 Processing 的详细安装步骤（由于版本更新等原因，以下步骤仅供参考）。

访问 Processing 的官方网站 https://processing.org/download

根据您的操作系统（Windows, macOS, Linux）选择合适的安装包进行下载。

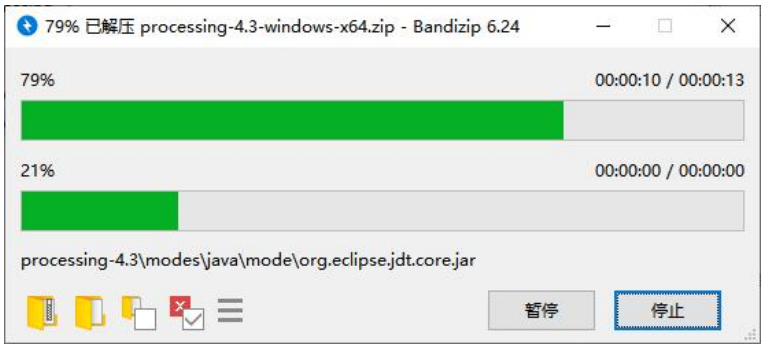
(以下安装步骤以 windows 为例)



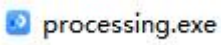
（如果你是用的其他系统的计算机，请参考 https://processing.org/tutorials/gettingstarted）

下载完成后，按照安装向导完成安装过程

将下载的文件解压缩



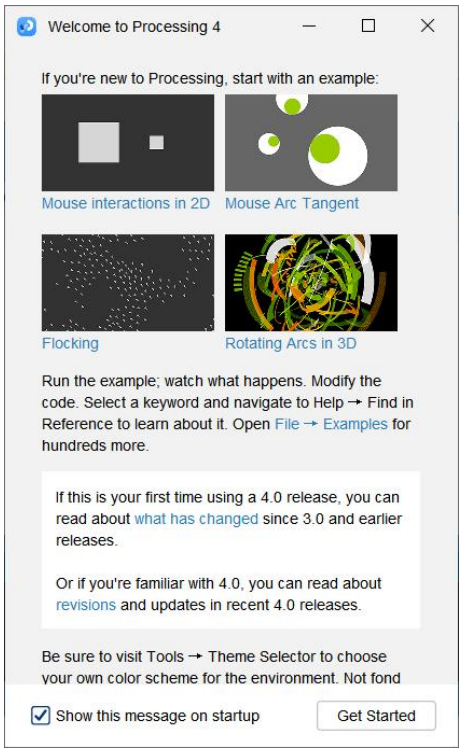
双击解压缩文件中的 processing.exe 文件以运行 processing。



如果一切顺利，现在将可以看到主要的 Processing 窗口。每个人的设置都不同，因此，如果程序没有启动，或者你遇到了其他问题，请访问 https://github.com/benfry/processing4/wiki/Troubleshooting 页面查找可

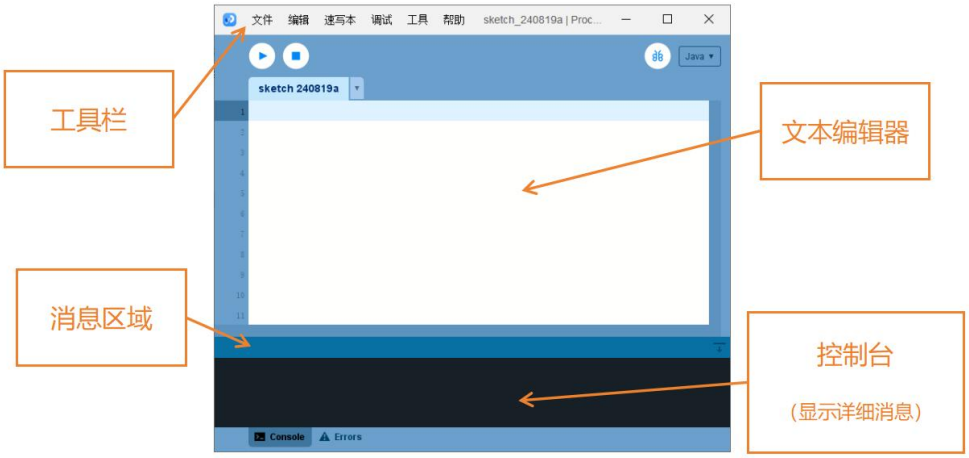
能的解决方案。

只需点击“Get Started”即可进入 processing 编程界面。



编写你的第一个 Processing 程序

你现在正在运行 Processing 开发环境（PDE）。界面功能如下所示：

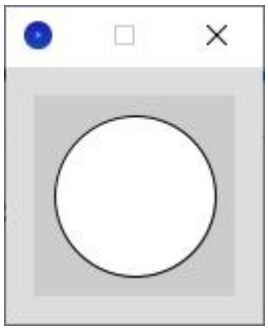


在文本编辑器中，输入以下内容：

ellipse(50, 50, 80, 80);

这行代码的意思是“绘制一个椭圆，其中心距离左边 50 像素，距离顶部 50 像素，宽度和高度均为 80 像素。” 点击工具栏上的运行按钮（三角形按钮）。

如果你输入的所有内容都是正确的，那么你会在屏幕上看到一个圆。



如果你没有正确输入，消息区域会变成红色，并显示错误信息。

如果发生这种情况，请确保你已完全复制了示例代码：数字应包含在括号内，并且每个数字之间用逗号隔开，行尾应有一个分号。

简单了解 Processing 软件的使用规则后，你就可以参考下方的程序代码实现其与 Arduino 的互动效果啦！

示例代码

Processing 样例代码：

```
// 项目 - 调光面板
import processing.serial.*; // 导入串口通信库

Serial myPort; // 创建串口对象

int circleX, circleY; // 按钮的位置
int circleSize = 100; // 按钮的直径

color currentColor, baseOffColor, baseOnColor; // 颜色变量
color circleColor; // 颜色变量
color circleHighlight; // 颜色变量

boolean circleOver = false; // 判断鼠标是否在按钮上
```

```
PFont f; // 字体类型
void setup() {
  size(600, 400); // 设置屏幕大小
  background(200); // 背景颜色

  circleColor = color(255, 30, 30); // 设置按钮原本颜色变量
  circleHighlight = color(255, 100, 100); // 设置按钮高亮颜色变量

  circleX = width / 4 * 2; // 设置按钮的 X 坐标
  circleY = height / 2; // 设置按钮的 Y 坐标

  printArray(PFont.list()); // 打印计算机上的字体列表
  f = createFont("Ebrima Bold", 20); // 设置字体
  textFont(f); // 应用字体

  String portName = "COM3"; // 输入你的 COM 端口
  myPort = new Serial(this, portName, 9600); // 选择第一个端口
  print(portName); // 打印端口名
}

void draw() {
  update(mouseX, mouseY); // 调用更新方法
```

```
  if (circleOver) {
    fill(circleHighlight); //  鼠标在按钮上时填充高亮颜色
  } else {
    fill(circleColor); //  鼠标不在按钮上时填充正常颜色
  }

  circle(circleX, circleY, circleSize); //  绘制圆形按钮
  stroke(255); //  边框颜色
  fill(140, 25, 25); //  文本颜色
  textAlign(CENTER); //  文本对齐方式
  text("RED", width / 4 * 2, height / 2 + 10); //  显示文本

}

void update(int x, int y) {
  if (circleOver(circleX, circleY, circleSize)) {
    circleOver = true; //  鼠标在按钮上
  } else {
    circleOver = false; //  鼠标不在按钮上
  }
}
```

```
boolean circleOver(int x, int y, int diameter) {
  float disX = x - mouseX;
  float disY = y - mouseY;
  if (sqrt(sq(disX) + sq(disY)) < diameter / 2) {
    return true; //  鼠标在按钮上
  } else {
    return false; //  鼠标不在按钮上
  }
}

int clicknumber; //  点击次数
void mousePressed() {
  if (circleOver) {

    clicknumber = clicknumber + 1;
    if (clicknumber % 2 == 1) {
      print("RedON"); //  打印信息"RedOn"
      myPort.write('1'); //  发送值 1
      background(200); //  背景颜色
    } else {
      myPort.write('2'); //  发送值 2
      print("RedOff"); //  打印信息"RedOff"
      background(50); //  背景颜色
```

```
    }
  }
}
```

Arduino IDE 样例代码:

```
// 项目 - 调光面板
int redPin;
void setup() {
  Serial.begin(9600); // 开启串口通信
  redPin = 9;
  pinMode(redPin, OUTPUT); // 设置 Arduino 引脚为输出
}
void loop() {
  if (Serial.available() > 0) { // 检查串口是否有数据
    char led = Serial.read(); // 读取串口数据
    if (led == '1') { // 检查串口读取的数据
        digitalWrite(redPin, HIGH); // 红灯亮
    } else if (led == '2') { // 检查串口读取的数据
        digitalWrite(redPin, LOW); // 红灯灭
    }
  }
}
```

代码回顾

Arduino IDE 与 Processing 的交互主要通过串口通信(Serial Communication)实现，使得 Arduino 可以控制硬件设备（如 RGB LED）并将数据发送到计算机，而 Processing 则能接收这些数据并进行可视化处理或发送控制指令给 Arduino。Arduino 通过接收到的指令不同，执行对应的操作。

虽然在代码示例部分 Processing 代码和 Arduino 代码是两组独立的代码程序。但两组代码不是完全割裂的，而是相互配合来运行和使用的。所以在下方的代码讲解过程中，虽然为了代码讲解的连续和完整性，不会把两个程序穿插进行讲解。但请将两组代码对照着理解，这有助于你更好的理解代码的逻辑原理。

Processing 程序代码回顾

在 setup()函数前，导入库和变量定义

```
import processing.serial.*;
Serial myPort;
int circleX, circleY;
int circleSize = 100;
color currentColor, baseColor;
color circlecolor;
color circleHighligh;
boolean circleOver = false;
```

```
PFont f;
```

这一部分导入了必要的库，定义了按钮的位置、大小、颜色，以及用于串行通信的 Serial 对象。同时，定义了字体和颜色变量。

在 setup()函数中：

```
size(600, 400);
circleColor = color(255, 100, 100);
circleHighlight = color(200, 50, 50);
baseColor = color(200);
currentColor = baseColor;
circleX = width / 4 * 2;
circleY = height / 2;
printArray(PFont.list());
f = createFont("Ebrima Bold", 20);
textFont(f);
String portName = "COM3";
myPort = new Serial(this, portName, 9600);
print(portName);
```

在 setup()函数中，设置了屏幕大小、按钮位置、颜色和字体。

很重要的是建立了与 Arduino 的串口通信连接，使用 String portName =

"COM3";输入 Arduino 连接的设备端口号，并初始化了串行通信。

draw()函数用于在屏幕上绘制所有元素：

```
update(mouseX, mouseY);
```

用于更新鼠标位置，update()函数用于判断鼠标是否停留在按钮上。

```
if (circleOver) {
  fill(circleHighlight);
} else {
  fill(circleColor);
}
```

用 if-else 进行判断来更新按钮颜色，当鼠标停留在按钮区域显示高亮颜色；否则显示按钮原本颜色。

```
circle(circleX, circleY, circleSize);
stroke(255);
fill(140, 25, 25);
textAlign(CENTER);
text("RED", width / 4 * 2, height / 2 + 10);
```

使用 circle(x, y, d)函数绘制圆形（配合对应函数做按钮使用），其中 x 和 y 是圆形中心的位置，d 对应的是圆形的直径。并绘制该圆形的边框，文本内容颜色和位置。

update()函数根据鼠标的当前位置(x, y)更新每个按钮的悬停状态。

```
void update(int x, int y) {
  if (circleOver(circleX, circleY, circleSize)) {
    circleOver = true; // 鼠标在按钮上
  } else {
    circleOver = false; // 鼠标不在按钮上
  }
}
```

这里虽然名为 update，但它实际上是 draw()函数中调用的一个函数，用于更新悬停状态，并不直接绘制任何内容。该函数中 circleOver(circleX, circleY, circleSize)为 update()函数的一个辅助方法，用于判断鼠标是否停在按钮圆形区域内。

```
boolean circleOver(int x, int y, int diameter) {
float disX = x - mouseX;
float disY = y - mouseY;
if (sqrt(sq(disX) + sq(disY)) < diameter / 2) {
```

```
        return true; //  鼠标在按钮上
    } else {
        return false; //  鼠标不在按钮上
    }
}
```

这些辅助方法用于判断鼠标是否悬停在特定的圆形按钮上。

disX 和 disY 是鼠标现在位置与按钮圆心横坐标和纵坐标的差值 (这个差值有可能为正有可能为负，后续只使用该差值的平方值，所以正负无影响）。利用勾股定理 $a^2+b^2=c^2$，使用 sqrt(sq(disX) + sq(disY))计算得出鼠标位置与按钮中心的距离，如果距离小于按钮半径，则返回 true，表示鼠标悬停在该按钮上。否则返回 false，表示鼠标悬停在按钮之外的位置。

最后，mousePressed()函数属于 processing 内置的事件处理函数，在用户点击鼠标时调用。

```
    void mousePressed() {
        if (circleOver) {
            clicknumber = clicknumber + 1;
            if (clicknumber % 2 == 1) {
                print("RedON");
                myPort.write('1');
                background(200);
```

```
        } else {
            myPort.write('2');
            print("RedOff");
            background(50);
        }
    }
}
```

当用户点击鼠标时鼠标

在 mousePressed()函数内进行判断，当点击发生在圆形按钮内，则按下按钮次数 clicknumber 加 1。接着对按下总次数进行判断，如果按下次数为奇数，向 Arduino 发送数字“1”，否则按下次数为偶数，向 Arduino 发送数字“2”。并在控制台输出对应的开关信息。

Processing 程序不直接控制 RGB LED 的高低电平，而是通过发送数字信号给 Arduino。Arduino 根据接收到的数字信号，向 RGB LED 输出相应的高低电平，以控制其颜色和亮度。这样，Processing 与 Arduino 协同工作，实现了对 RGB LED 的间接控制。

Arduino 程序代码回顾

在 loop()函数中，if (Serial.available() > 0)检查串口是否有新的数据可读。如果有，就执行大括号内的代码。

```
if (Serial.available() > 0) {
  }
```

在 if 执行块内，先将 Processing 发送的数字赋值给字符串型变量 led，接下来的两个 if 语句检查读取的数据 led。如果为'1'，则通过 digitalWrite(redPin, HIGH);打开 LED；如果为 '2'，则通过 digitalWrite(redPin, LOW);关闭 LED。

```
char led = Serial.read();
if (led == '1') { // 检查串口读取的数据
        digitalWrite(redPin, HIGH); // 红灯亮
    } else if (led == '2') { // 检查串口读取的数据
        digitalWrite(redPin, LOW); // 红灯灭
    }
```

课后练习

使用多个按钮实现 RGB LED 的光线明暗度的调节，或者颜色的变化。

发挥你的想象力，制作一个功能强大的调光面板，实现更绚丽多彩的灯光效果吧！（本项目附件程序中有多个按钮版本的程序供您参考）

项目 - 绘制光线

欢迎来到绘制光线这个项目！在这个项目中，我们将 Arduino 上读取的光线传感器数据实时传输到 Processing 中并实现其可视化。你将学习如何计算光线的平均值以减少噪声干扰。同时，你将在 Processing 中创建一个简单的界面，用于显示随着光线变化而起伏的光谱。现在，让我们开始这个有趣的旅程吧！

元件清单

	**Ambient Light Sensor** 光敏电阻	x1
	**Resistor 10K** 10K电阻	x1
	**Jumper Cables M/M** 跳线（公公头）	x3

硬件连接

请按照下图进行硬件连接。

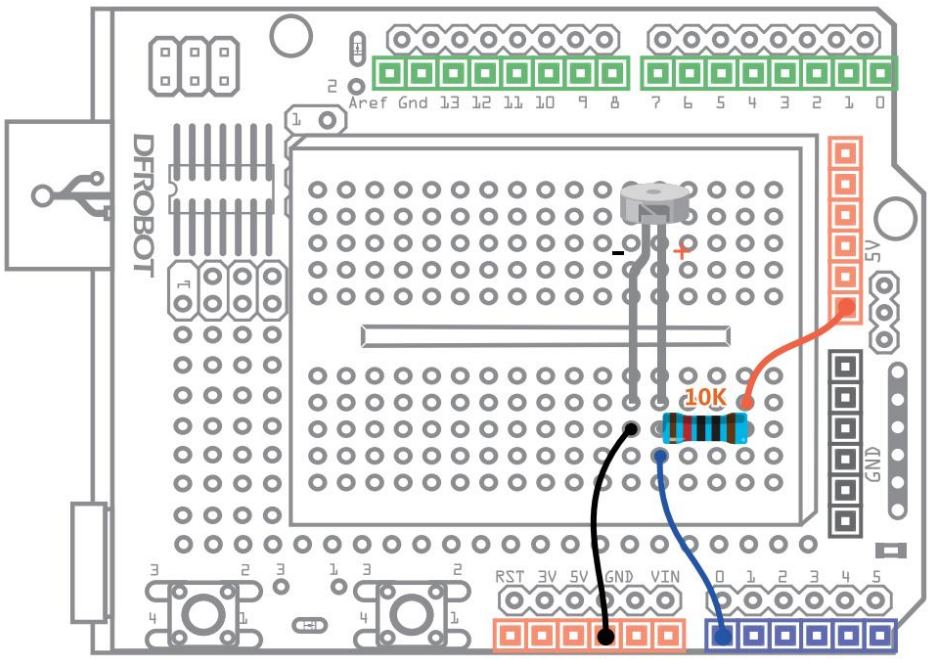


图 1 绘制光线连线图

示例代码

Processing 样例代码：

```
//项目 - 绘制光线
import processing.serial.*;
int[] Ypoint; // 垂直方向上的坐标
//int[] x = new int[0];

Serial myPort;  // 创建 Serial 类的对象
int val;        // 从串口接收到的数据

void setup()
```

```
{
  size(640, 360);
  colorMode(HSB,1000);
  Ypoint = new int[width];
  String portName = "COM6"; // 打开你正在使用的端口。
  myPort = new Serial(this, portName, 9600);
  print(portName);
}

void draw()
{
  background(900);                    // 设置背景为灰色

  for (int i = 1; i < width; i++) { // 记录 Ypoint
    Ypoint[i-1] = Ypoint[i];
  }
    if ( myPort.available() > 0) {  // 如果有数据可用,
    val = myPort.read(); // 读取数据并存储在 val 中
  }

  Ypoint[width-1] = val;
  int x=1;
  for(int i = 1; i < width; i++) {
```

```
    // 绘制曲线
    x++;
    line(i, height, i, map(Ypoint[i], 0, 255, 0, height)); // 将光强转换为
Ypoint
    stroke(x,800,800);

  }

    println(val);

}
```

Arduino IDE 样例代码:

```
//项目 - 绘制光线
#include <Arduino.h>
int lightSensorPin;
const int numReadings = 10;
int readings[numReadings];
int readIndex = 0;
long total = 0;
int average = 0;

void setup() {
```

```
  Serial.begin(9600);

  for (int thisReading = 0; thisReading < numReadings; thisReading++)
{//build the list
    readings[thisReading] = 0;
  }

  lightSensorPin = A0;
  pinMode(lightSensorPin, INPUT);

}

void loop() {
  int newReading = analogRead(lightSensorPin); // 读取新的光线传感
器值
  total = total - readings[readIndex] + newReading;
  readings[readIndex] = newReading;
  readIndex = (readIndex + 1) % numReadings;
  average = total / numReadings;
  delay(50);
  int mappedValue = map(average, 0, 1023, 0, 255); // 将平均值映射
到 0-255 范围内
  Serial.write(mappedValue);
```

```
}
```

上传 Arduino 的代码后需要关闭 Arduino IDE，并打开 processing 软件点击运行。等待数据传输稳定（大约 5 秒钟），使用手电筒照射光敏传感器或移开，可以观察到电脑上的光谱曲线会根据光线强度进行调节。

代码回顾

本项目主要利用 Arduino 进行光线强度的接收与数据处理，并通过串口通信将处理后的亮度值发送给 Processing 进行可视化。具体来说，Arduino 通过光线传感器实时捕捉环境中的光线亮度，并将其转换为数字值。随后，对这些亮度值进行必要的处理，以确保数据的准确性和可靠性。处理后的亮度值通过串口通信实时发送给 Processing。在 Processing 中，根据接收到的亮度值，绘制一个动态的光谱区域，该区域的高低起伏直观地表示了光线亮度的实时变化情况，从而为用户提供了一个直观、实时的光线亮度变化监测工具。

Processing 程序代码回顾

在 setup()中：

对窗口大小和颜色进行设置。并初始化 Ypoint 数组，长度为窗口的宽度（即 640），用于绘制竖线。

```
  size(640, 360);
```

```
  colorMode(HSB,1000);
  Ypoint = new int[width];
```

设置串口通信，并打印串口名称。

```
  String portName = "COM6";
  myPort = new Serial(this, portName, 9600);
  print(portName);
}
```

在 draw()中：

将 Ypoint 数组中的元素向前移动一位，为新的数据腾出空间。

```
  for (int i = 1; i < width; i++) {
    Ypoint[i-1] = Ypoint[i];
  }
```

如果串口有数据可读，从串口读取一个字节的数据，并存储在 val 变量中。

```
    if ( myPort.available() > 0) {
    val = myPort.read();
  }
```

将最新读取的数值赋值给 Ypoint 数组的最后一个元素

```
Ypoint[width-1] = val;
```

这段代码的目的是根据数组 Ypoint[]中的每个元素值，绘制一系列竖直线条。这些线条的 x 坐标与它们在数组中的索引相对应，而 y 坐标则通过映射函数 map()将 Ypoint[i]的值从 0 到 255 的范围转换到窗口的垂直范围（从 0 到 height）。并利用 x 值的自增来实现每条竖线颜色的变化。

```
int x=1;
for(int i = 0; i < width-1; i++) {
  x++;
  line(i, height, i, map(Ypoint[i], 0, 255, 0, height));
  stroke(x,800,800);
}
```

Arduino 程序代码回顾

Arduino IDE 代码

在 setup()函数中，循环初始化 readings 数组，将所有元素设置为 0。

```
for (int thisReading = 0; thisReading < numReadings; thisReading++)
{
    readings[thisReading] = 0;
  }
```

在 loop()中，

```
int newReading = analogRead(lightSensorPin);
```

读取当前光线的模拟值，将其存在 readings 数组的当前索引位置。

```
total = total - readings[readIndex] + newReading;
```

更新 total 的值，先减去要被替换的旧读数，再加上新的读数。

```
readings[readIndex] = newReading;
```

将新的读数存储在 readings 数组的当前索引位置。

```
readIndex = (readIndex + 1) % numReadings;
```

更新 readIndex 的值，使其指向下一个数组位置，如果到达数组末尾则循环回到开头。

```
average = total / numReadings;
```

计算所有读数的平均值，并赋值给 average 变量

```
int mappedValue = map(average, 0, 1023, 0, 255);
Serial.write(mappedValue);
```

由于 Serial.write 只能写入 8 位的二进制，也就是 0-255，所以使用 map 函数将平均值从 0-1023 范围映射到 0-255 范围，再通过串口发送 mappedValue 的整数部分（配合 processing 中的相关接收数据的语句来使用）。