

Arduino 入门实验课程

从第一盏灯开始，真正学懂 Arduino

30 个循序渐进项目，从图形化编程走向 Arduino 代码，系统学习电子电路、传感器与硬件控制。

30 个项目式课程

35 类常用电子元件

图形化入门

Arduino IDE 代码进阶

适合 Arduino 零基础学习者、青少年、教师与创客 / STEM 课程。

DFROBOT

Beginner Kit
for Arduino

目 录

DFRobot 初识 Arduino 实验课程 · 30 个项目

项目一	初识 Arduino	3
项目二	SOS 求救信号器	16
项目三	通信信号灯	28
项目四	呼吸灯	45
项目五	炫彩 LED	53
项目六	报警器	63
项目七	温度报警器	70
项目八	震动探测	83
项目九	感光灯	90
项目十	舵机初动	96
项目十一	可控舵机	102
项目十二	彩灯调光台	106
项目十三	自制风扇	110
项目十四	红外遥控灯	117
项目十五	红外遥控数码管	126
项目十六	音乐蜂鸣器	150
项目十七	DJ 调音台	160
项目十八	敲击电子鼓	169
项目十九	手势控制灯	177
项目二十	反应速度测试	186
项目二十一	数码管摇骰子	198
项目二十二	指尖开关	206
项目二十三	换挡风扇	212
项目二十四	H 桥电路驱动	218
项目二十五	声控灯	224
项目二十六	指针式噪音计	233
项目二十七	点亮点阵	238
项目二十八	温度计	247
项目二十九	调光面板	265
项目三十	绘制光线	282

初识 Arduino

Arduino 是什么？

Arduino 是一个开放源码电子原型平台，拥有灵活、易用的硬件和软件。Arduino 专为设计师，工艺美术人员，业余爱好者，以及对开发互动装置或互动式开发环境感兴趣的人而设的。

Arduino 可以接收来自各种传感器的输入信号从而检测出运行环境，并通过控制光源，电机以及其他驱动器来影响其周围环境。板上的微控制器编程使用 Arduino 编程语言（基于 Wiring）和 Arduino 开发环境（以 Processing 为基础）。Arduino 可以独立运行，同时也支持通过串行通信与计算机上运行的软件（如 Processing, MaxMSP）进行交互，进一步明确这些软件与 Arduino 通信的具体方式或用途，以实现更复杂的功能。Arduino 开发 IDE 接口基于开放源代码，可以让您免费下载使用开发出更多令人惊艳的互动作品。

Arduino 是人们连接各种任务的粘合剂。要给 Arduino 下一个最准确的定义，最好用一些实例来描述。

- 您想当咖啡煮好时，咖啡壶就发出“吱吱”声提醒您吗？
- 您想当邮箱有新邮件时，电话会发出警报通知您吗？
- 想要一件闪闪发光的绒毛玩具吗？
- 想要一款具备语音和酒水配送功能的 X 教授蒸汽朋克风格轮椅吗？
- 想要一套按下快捷键就可以进行实验测试蜂音器吗？

- 想为您的儿子自制一个《银河战士》手臂炮吗？
- 想自制一个心率监测器，将每次骑脚踏车的记录存进存储卡吗？
- 想过自制一个能在地面上绘图，能在雪中驰骋的机器人吗

Arduino 都可以为您实现。

Arduino 诞生啦

Arduino 这个最经典的开源硬件项目，诞生于意大利的一间设计学校。Arduino 的核心开发团队成员包括：Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, David Mellis 和 Nicholas Zambetti。据说 Massimo Banzi 的学生们经常抱怨找不到便宜好用的微控制器，2005 年冬天，Massimo Banzi 跟朋友 David Cuartielles 讨论了这个问题，David Cuartielles 是一个西班牙籍晶片工程师，当时在这所学校做访问学者。两人决定设计自己的电路板，并找到 Banzi 的学生 David Mellis 为电路板设计编程语言。两天以后，David Mellis 就写出了程式码。又过了三天，电路板就完工了。这块电路板被命名为 Arduino。即使不懂电脑编程，你可以也能用 Arduino 做出酷炫的东西，比如闪烁灯光，也能对马达进行控制，还能通过传感器感受外界环境，并用执行器做出相应的反应。

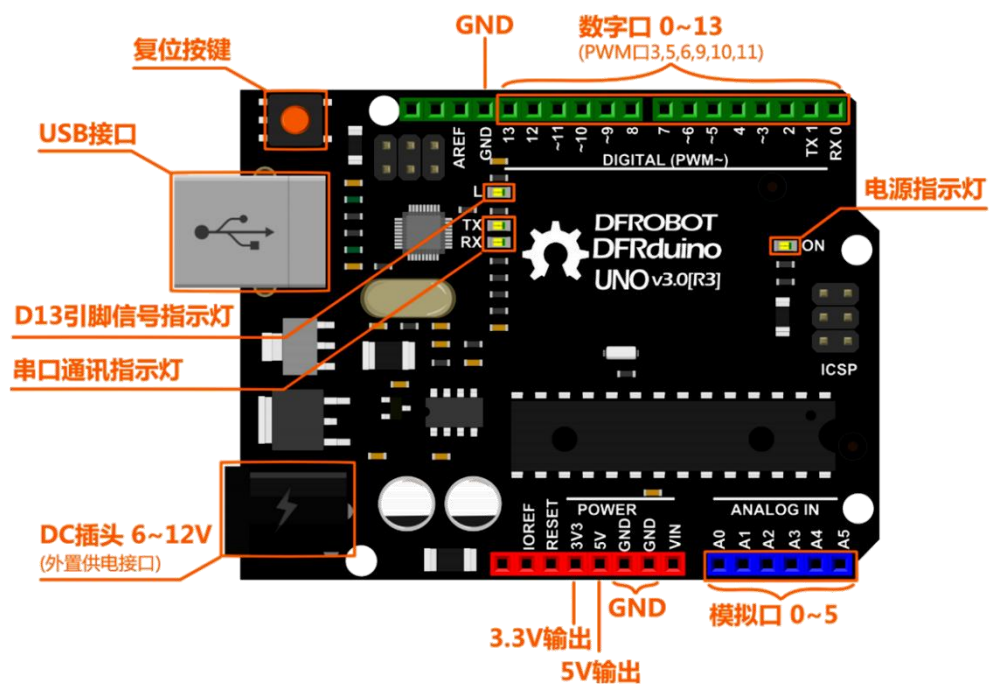
Arduino 名称由来

意大利北部一个如诗如画的小镇「Ivrea」，横跨过蓝绿色 Dora Baltea 河，它最著名的事迹是关于一位受压迫的国王。公元 1002 年，国王 Arduin 成为国家的统治者，不幸的是两年后即被德国亨利二世国王给废掉了。

今日，在这位无法成为新国王的出生地，cobblestone 街上有家叫「di Re Arduino」的酒吧纪念了这位国王。Massimo Banzi 经常光临这家酒吧，而他将这个电子产品计划命名为 Arduino 以纪念这个地方。

认识 Arduino UNO

先来简单的看下 Arduino UNO。下图中有标识的部分为常用部分。图中标出的数字口和模拟口，即为常说的 I/O。数字口有 0~13，模拟口有 0~5。除了最重要的 I/O 口外，还有电源部分。UNO 可以通过两种方式供电方式，一种通过 USB 供电，另一种是通过外接 6~12V 的 DC 电源。除此之外，还有 4 个 LED 灯和复位按键，稍微说下 4 个 LED。ON 是电源指示灯，通电就会亮了。L 是接在数字口 13 上的一个 LED，在下面一节会有个样例来说明的。TX、RX 是串口通讯指示灯，比如我们在下载程序的过程中，这两个灯就会不停闪烁。



初次使用

1. 下载 Arduino IDE

打开网页输入网址

<https://www.arduino.cc/en/software>

进入到页面后，找到下图显示的部分。



Arduino IDE 2.3.2

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the [Arduino IDE 2.0 documentation](#).

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

The Arduino IDE 2.0 is open source and its source code is hosted on [GitHub](#).

DOWNLOAD OPTIONS

Windows Win 10 and newer, 64 bits
Windows MSI installer
Windows ZIP file

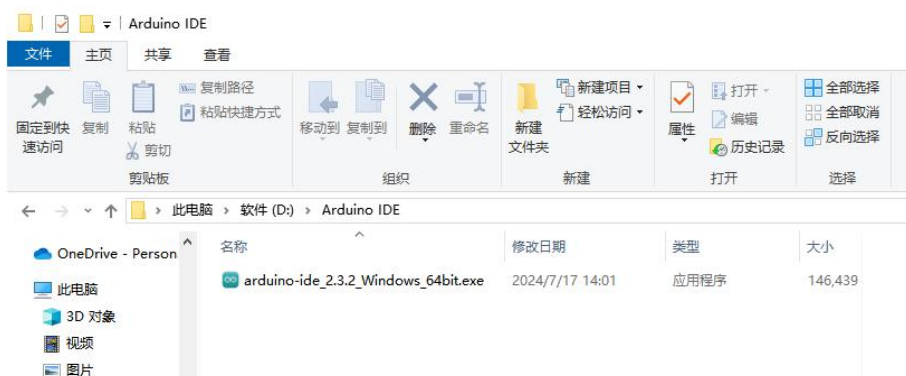
Linux AppImage 64 bits (X86-64)
Linux ZIP file 64 bits (X86-64)

macOS Intel, 10.15: "Catalina" or newer, 64 bits
macOS Apple Silicon, 11: "Big Sur" or newer, 64 bits

[Release Notes](#)

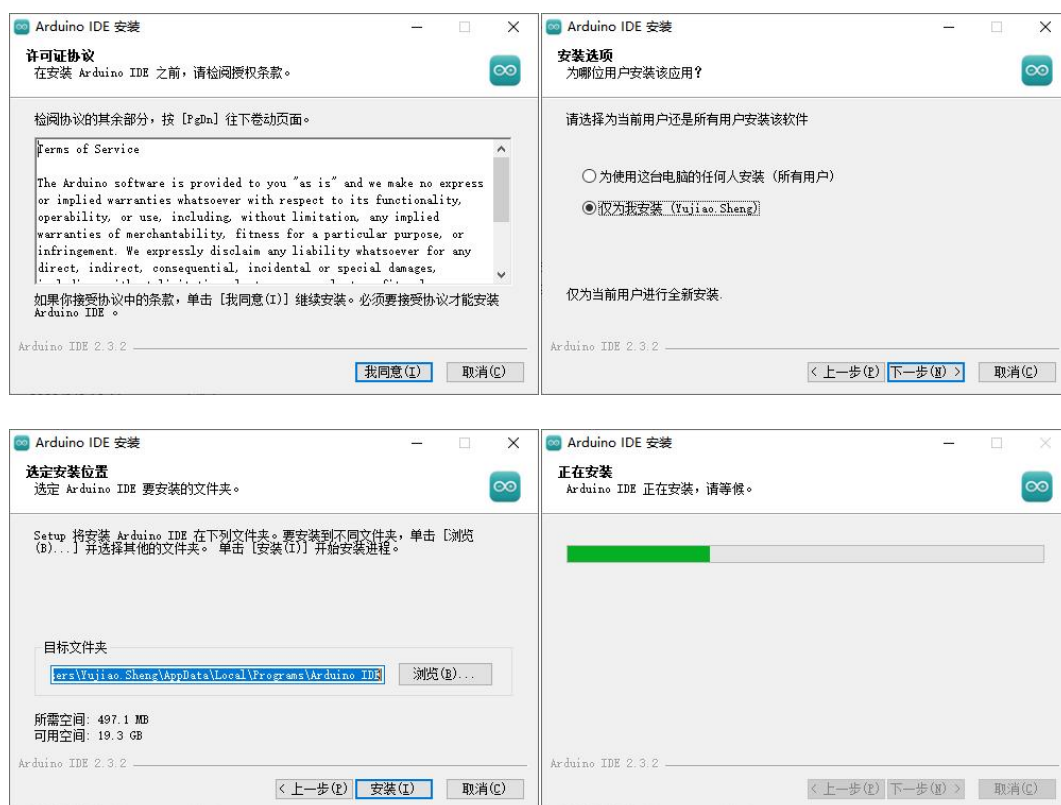
Windows10 以上用户，点击下载“**Windows**”的对应版本。如果 macOS，Linux 用户则选择相应的系统。

要在 Windows 计算机上安装 Arduino IDE 2，只需运行从官网软件页面下载的文件。



2. 安装 Arduino IDE

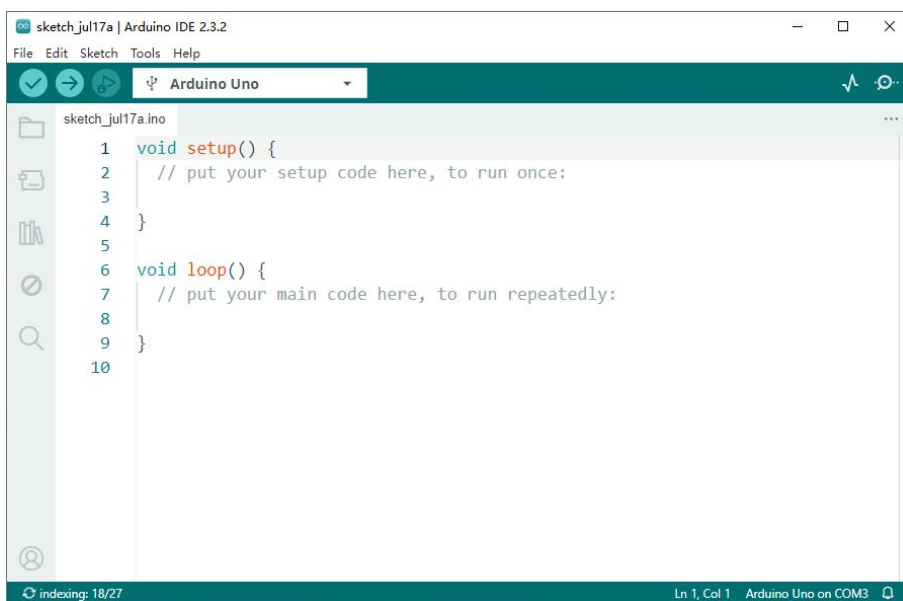
按照安装指南中的说明进行操作。安装可能需要几分钟。



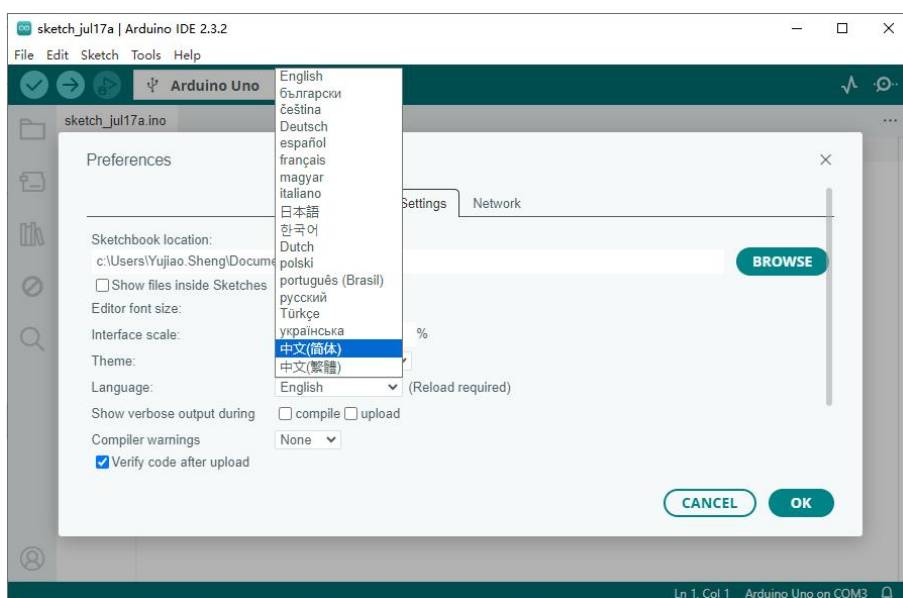
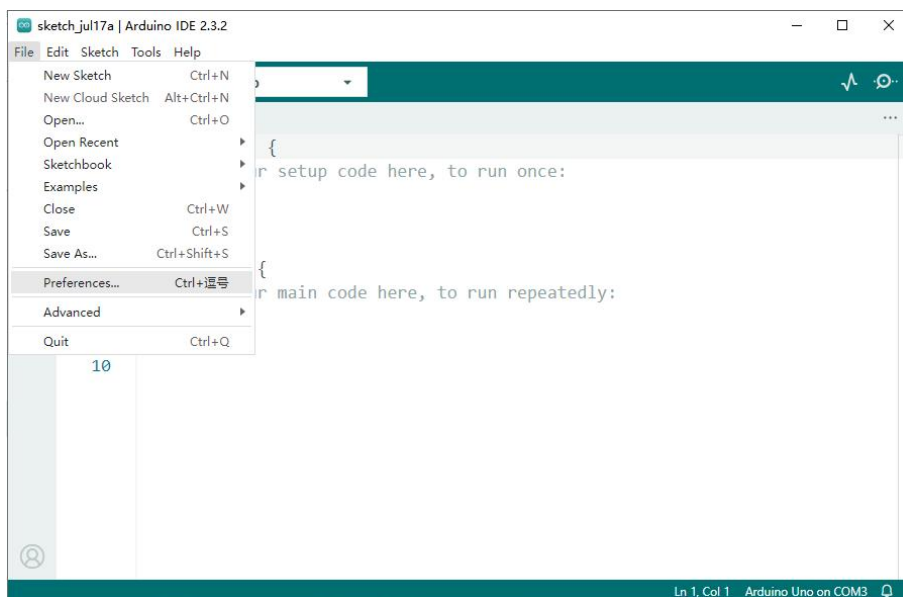


3. 认识 Arduino IDE

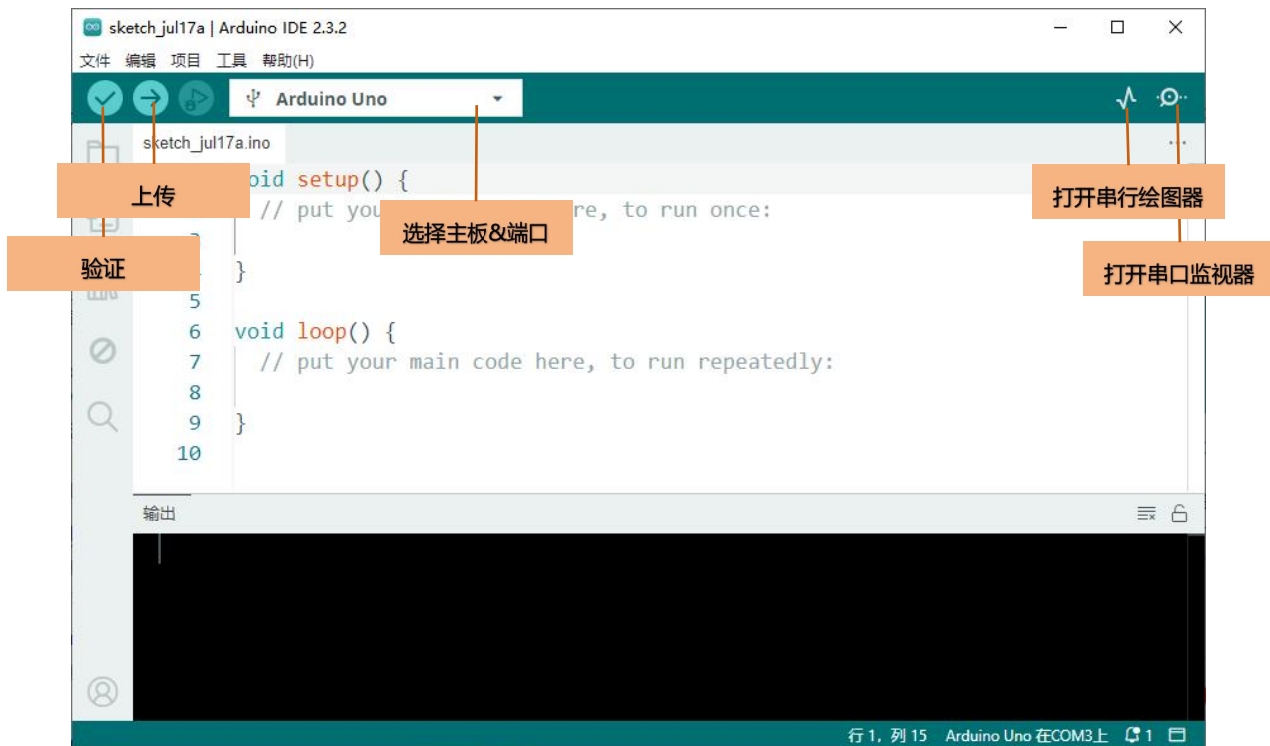
打开 Arduino IDE，就会出现 Arduino 的编辑界面。



如果英文界面，你不太习惯的话，可以先更改为中文界面。选择菜单栏 File -> Preferences。



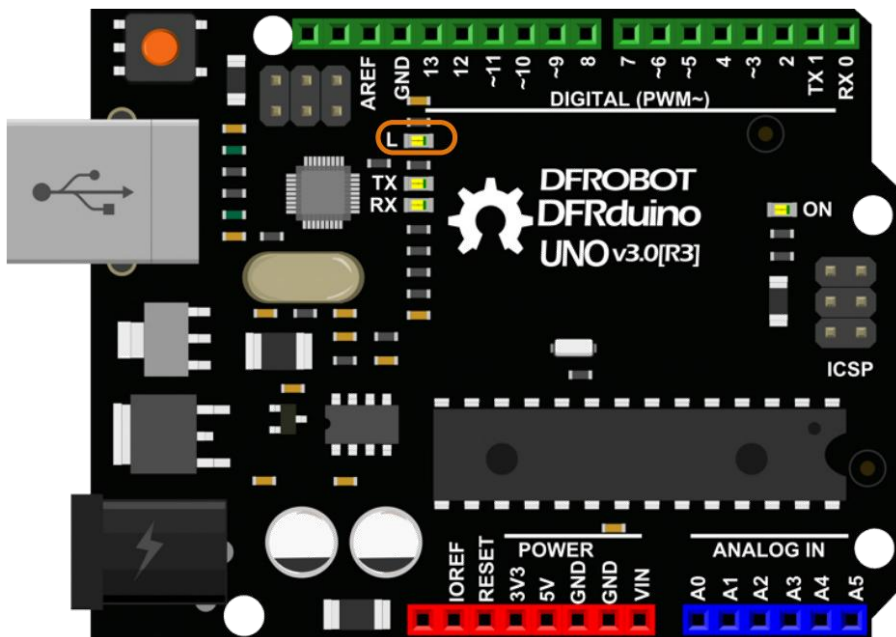
Arduino IDE 是 Arduino 产品的软件编辑环境。简单的说就是用来写代码，上传代码的地方。任何的 Arduino 产品都需要上传代码后才能运作。我们所搭建的硬件电路是辅助代码来完成的，两者是缺一不可的。如同人通过大脑来控制肢体活动是一个道理。如果代码就是大脑的话，外围硬件就是肢体，肢体的活动取决于大脑，所以硬件实现取决于代码。



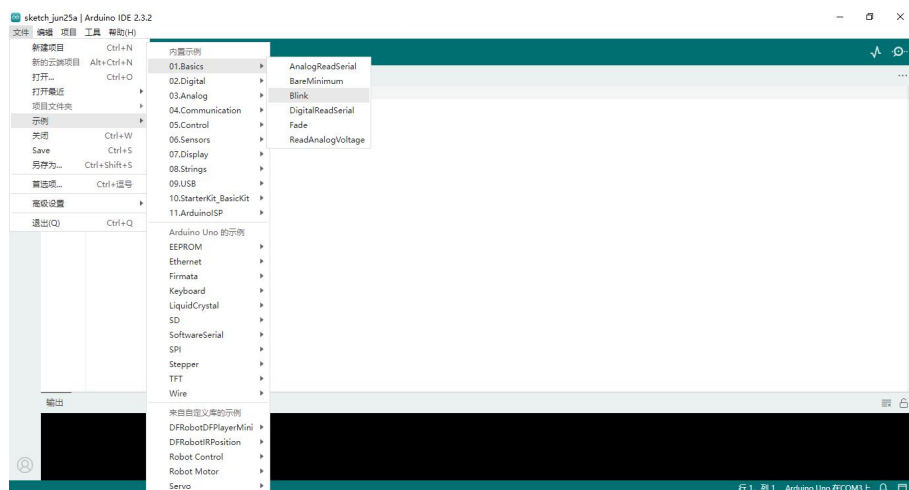
Arduino IDE 基本也只需要用到上面标示出来的部分就可以了，上图大部分的白色区域就是代码的编辑区，用来输入代码的。注意，输入代码时，要切换到英文输入法的模式。下面黑色的区域是消息提示区，会显示验证或者上传是否通过。

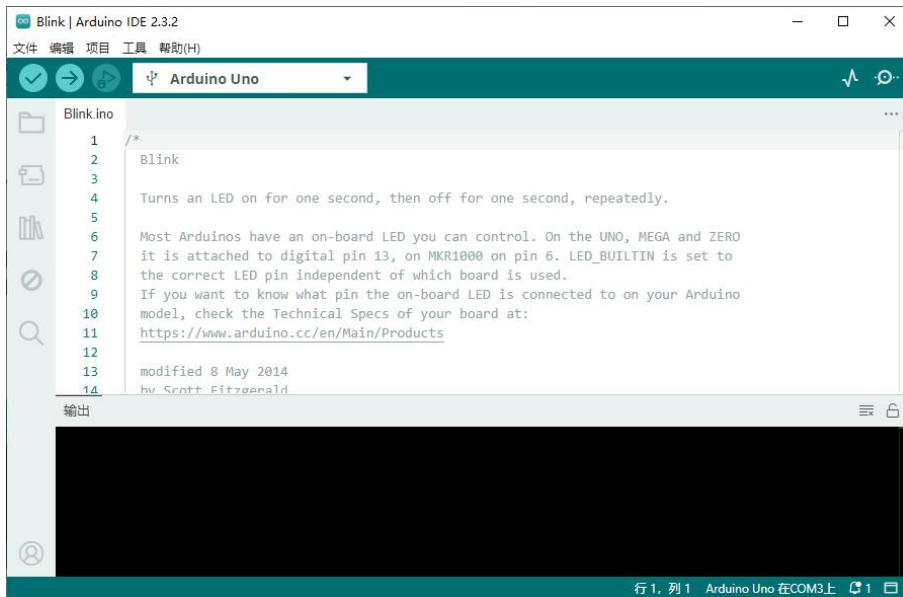
4. 运行一个 Blink 程序

上传一个最简单的代码，既可以帮你熟悉如何上传程序，同时也测试下板子好坏。UNO 板上标有 L 的 LED。这段测试代码就是让这个 LED 灯闪烁。

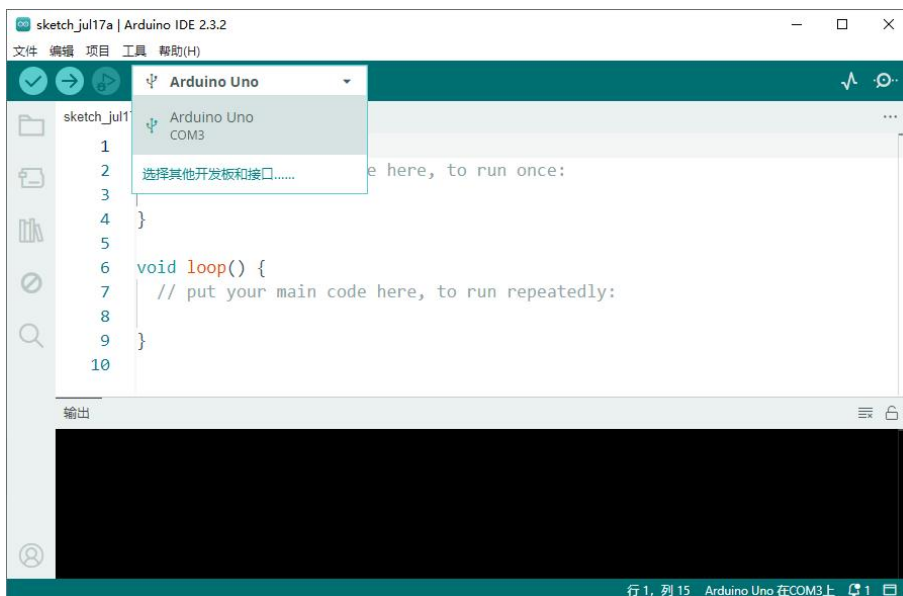


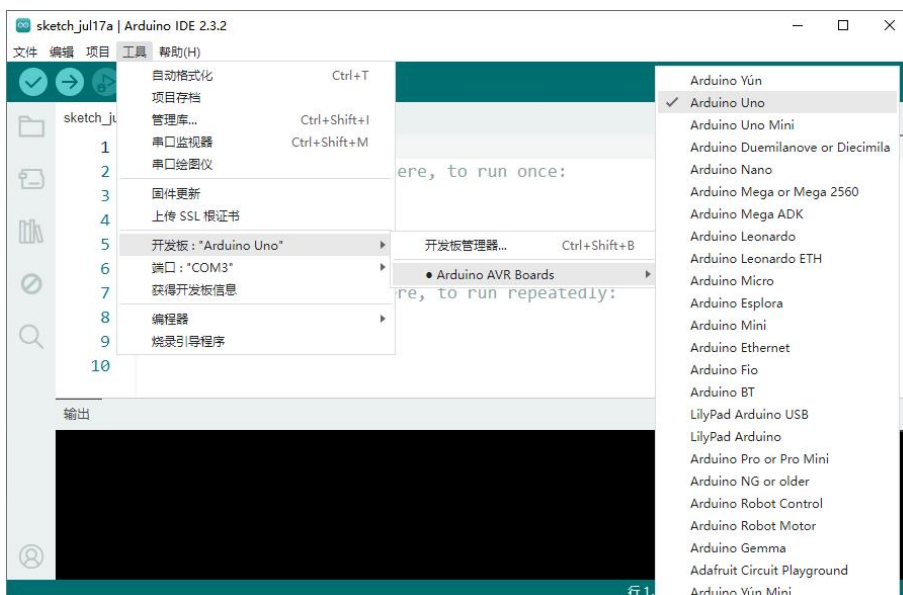
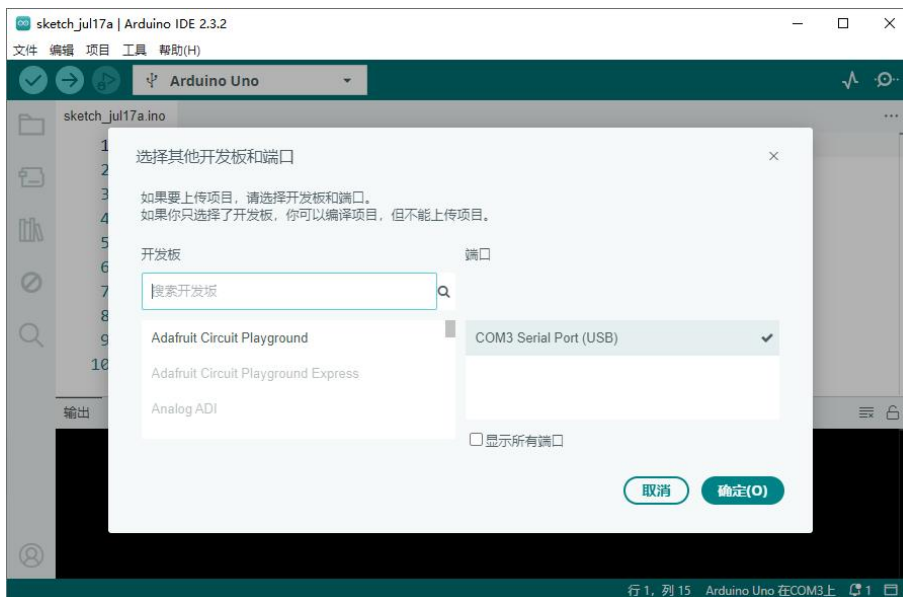
插上 USB 线，打开 Arduino IDE 后，按照下图找到 “Blink” 代码。





在执行其他操作之前，我们应该选择要上传到哪个主板。在验证和上传按钮旁边，您应该看到一个下拉菜单，在大多数情况下，它会显示连接到您计算机的 Arduino 主板。如果没有自动检测到您的主板，您可以按下拉菜单中的“选择其他开发板和接口.....”并按照说明进行操作，或者转到工具栏菜单中的工具 -> 开发板或端口以手动选择开发板和端口。





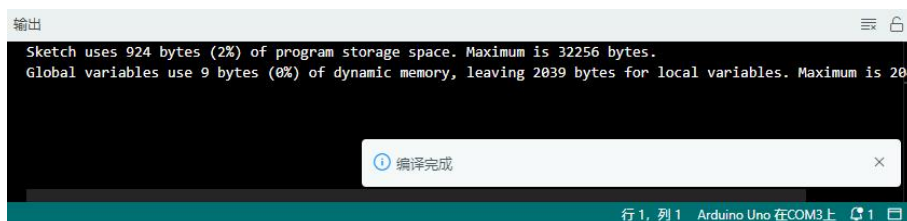
写完一段代码并选择好开发板和端口后，我们都需要编译一下，看看代码有没有错误。点击“验证”。



编译中.....



编译完成!

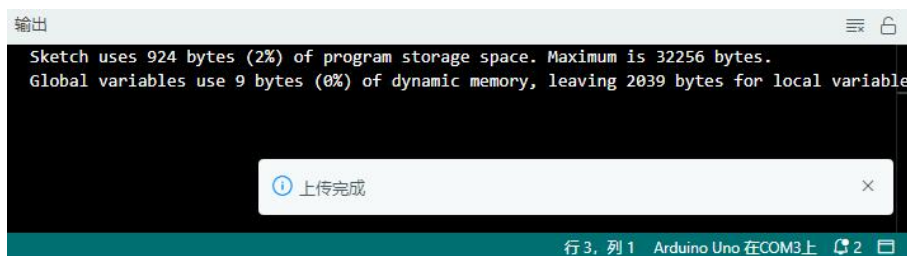


由于是样例代码，所以校验不会有错误，不过在以后写代码的过程中，输入完代码，都需要校验一下，然后再上传到 Arduino 中。

最后，点击“上传”。



上传成功!



你就可以观察到板载 L 灯会像程序编写的一样一直闪烁。

回顾

以后程序下载就照着这个步骤做就可以了，再理一下思路，分为三步走：

选择开发板和端口 -> 验证 -> 上传

项目 - SOS 求救信号器

本项目将继续使用【闪烁第一个 LED】所搭建的电路（注意引脚有所更换），只需要对代码进行一定的改动，就能让我们的 LED 发出 SOS 求救信号了。

国际摩斯码是一种独特的字符编码系统，其精妙之处在于它利用简单的两种信号：横杠（长音）和点（短音），通过不同的组合方式来代表英文中的每一个字母以及数字。这种设计极大地简化了信息传递的复杂性，因为仅需两种基本信号形态，就能构建出完整的通信语言。这种创造性的编码方式展现了前人非凡的智慧与创新能力，确实令人钦佩不已。

我们正好可以通过 LED 发光的长短来代表不同的字母。通过长闪烁和短闪烁来分别表示横杠和点。在这个项目中，我们就拼写 SOS 这三个字母。

通过查阅摩斯码表，我们可以知道，字母“S”的摩斯码是三个点，我们这里用三次短闪烁表示。字母“O”则是三个横杠，用三次长闪烁表示。

硬件连接

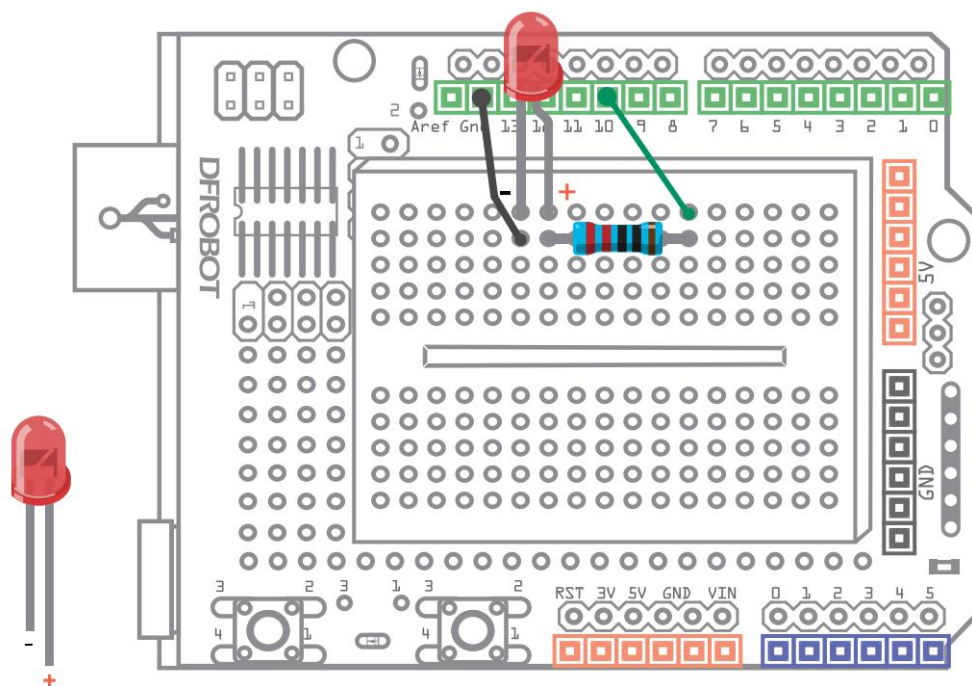


图 1 SOS 求救信号器

示例代码

有了前一个项目的基础, 不难理解下面样例代码 1。但先不要急着输入这段代码, 只是看一下。

样例代码 1:

```
int ledPin = 1010;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // 三个短闪烁来表示字母“S”
```

```
digitalWrite(ledPin,HIGH);  
  
delay(150);  
  
digitalWrite(ledPin,LOW);  
  
delay(100);  
  
  
  
digitalWrite(ledPin,HIGH);  
  
delay(150);  
  
digitalWrite(ledPin,LOW);  
  
delay(100);  
  
digitalWrite(ledPin,HIGH);  
  
delay(150);  
  
digitalWrite(ledPin,LOW);  
  
delay(100);  
  
  
  
delay(101000); //1000 毫秒延时产生字母之间的间隙  
  
  
  
//三个长闪烁来表示字母“0”  
  
digitalWrite(ledPin,HIGH);  
  
delay(400);  
  
digitalWrite(ledPin,LOW);  
  
delay(100);  
  
digitalWrite(ledPin,HIGH);  
  
delay(400);
```

```
digitalWrite(ledPin,LOW);

delay(100);

digitalWrite(ledPin,HIGH);

delay(400);

digitalWrite(ledPin,LOW);

delay(100);


delay(1000); //1000 毫秒延时产生字母之间的间隙


//再用三个短闪烁来表示字母“S”

digitalWrite(ledPin,HIGH);

delay(150);

digitalWrite(ledPin,LOW);

delay(100);


digitalWrite(ledPin,HIGH);

delay(150);

digitalWrite(ledPin,LOW);

delay(100);

digitalWrite(ledPin,HIGH);

delay(150);

digitalWrite(ledPin,LOW);

delay(100);
```

```
    delay(5000); // 在重复 SOS 信号前等待 5 秒
}
```

上面的写法固然正确，是不是觉得有点繁琐呢？如果有 100 个字母，难不成还重复 100 遍吗？有没有更好的编写程序的方法呢？想必发明编程的人也考虑到这个问题了，所以我们有了更好的一种写法。

我们先来看一下样例代码 2。

样例代码 2:

```
//项目 -- SOS 求救信号器

int ledPin = 10;

void setup() {
    pinMode(ledPin, OUTPUT);
}

void loop() {
    // 三个短闪烁来表示字母“S”
    for(int x=0;x<3;x++){
        digitalWrite(ledPin,HIGH); //设置 LED 为开
        delay(150);                //延时 150 毫秒
        digitalWrite(ledPin,LOW);  //设置 LED 为关
        delay(100);                //延时 100 毫秒
    }
}
```



```
//1000 毫秒延时产生字母之间的间隙

delay(1000);

//三个长闪烁来表示字母“O”

for(int x=0;x<3;x++){

digitalWrite(ledPin,HIGH); //设置 LED 为开

delay(400);                //延时 400 毫秒

digitalWrite(ledPin,LOW);  //设置 LED 为关

delay(100);                //延时 100 毫秒

}

//1000 毫秒延时产生字母之间的间隙

delay(1000);

// 再用三个短闪烁来表示字母“S”

for(int x=0;x<3;x++){

digitalWrite(ledPin,HIGH); //设置 LED 为开

delay(150);                //延时 150 毫秒

digitalWrite(ledPin,LOW);  //设置 LED 为关

delay(100);                //延时 100 毫秒

}

//在重复 SOS 信号前等待 5 秒

delay(5000);

}
```

在输入代码的时候，注意保持代码层次的清晰，除了美观外，也便于你日后检查代码。编译通过后，上传代码到 Arduino 中，如果一切顺利的话，我们将看

到 LED 闪烁出摩斯码 SOS 信号，等待 5 秒，重复闪烁。

通过使用 Arduino 板外接电池，并将整个装置封装在一个防水盒子里，我们就能创建出一个便携式设备，用于在紧急情况下发送国际通用的求救信号 SOS。

这一信号在航海和登山等户外活动中尤为重要，因为它能够快速有效地传达出求助信息。

我们接着来分析下代码。

代码回顾

代码 loop()前的部分与项目【闪烁第一个 LED】是完全一样的。也是初始化一个变量，设置数字引脚 10 的模式为输出模式。在主函数 loop()中，你可以看到与项目【闪烁第一个 LED】中类似的语句用来控制 LED 的打开和关闭，并通过延时 delay()实现长短闪烁功能。然而，有所不同的是这次主函数包含了三个独立的代码段。

第一段代码段是输出代表字母 “S” 的三个点：

```
for(int x=0;x<3;x++){  
  
    digitalWrite(ledPin,HIGH); //设置 LED 为开  
  
    delay(150);                //延时 150 毫秒  
  
    digitalWrite(ledPin,LOW);  //设置 LED 为关  
  
    delay(100);                //延时 100 毫秒  
  
}
```

LED 开关的控制代码位于一对花括号（也称为大括号）内部，这个区域被定义为一个代码段。花括号都必须成对出现，即开启一个花括号必须有一个相应的闭合花括号，否则编译器在处理代码时会因为语法错误而无法通过编译。有个小技巧大家可以学一下，在开始写花括号的时候，就先把 “{” “}” 都写上，之后再在两个花括号之间输入代码，这样就不会出现写到最后括号对应不上的情况。

当程序运行后我们可以看到，灯闪了 3 次而不是只闪了 1 次。产生这样的效果是因为使用了 for 循环（用来重复执行循环体中的代码块）。我们来看一下：for 语句格式如下：

```
for (1 循环初始化; 2 循环条件; 4 循环调整语句){  
  3 循环体语句;  
}
```

The diagram shows the syntax of a for loop with four numbered annotations: 1 points to the initialization part, 2 points to the condition part, 4 points to the adjustment part, and 3 points to the body part. A box labeled '条件为真' (Condition is true) has an arrow pointing to the condition part, indicating the loop continues when the condition is true.

for 循环顺序如下：

第一轮：1 → 2 → 3 →

第二轮：2 → 3 → 4

.....

直到 ② 不成立(为假)，则结束执行 for 循环，并继续向下执行程序。

（注意：for 循环首先执行的是初始化部分，然后进行循环条件判断，如果条件为真，则执行循环体，之后才执行循环调整语句的部分。）

来看下我们程序中的 for 循环：

```
for(int x=0;x<3;x++){  
    ...  
}
```

第一次循环：

- 声明并初始化变量 $x=0$ 。
- 判断 $x<3$ ，为真，则执行循环体中的程序。
- x 自加，更新 x 为 1。

第二次循环：

- 此时 $x=1$ 。
- 判断 $x<3$ ，为真，则执行循环体中的程序。
- x 自加，更新 x 为 2。

.....

循环结束：

- 在下一次循环开始前， x 的值被自加到 3。
- 判断 $3<3$ ，为假，因此循环结束，不再执行循环体。并继续向下执行。

($x++$ 是 $x=x+1$ 的简写，实现 x 的迭代（自加）功能。)

我们这里需要它循环 3 次，所以设置为 $x<3$ 。从 0 开始计算，0，1，2，循环

了 3 次。那如果要循环 100 次的话呢？

答案：

```
for(int x=0;x<100;x++){  
    ...  
}
```

我们在写一些判断语句的时候会经常用到一些比较运算符，比如大于，小于等等。下面就介绍一些常用的比较运算符。

比较运算符

for 循环中的 “<” 称之为比较运算符。比较运算符在代码中是用作判断的，用于比较两个值。

我们常用的比较运算符有：

== (等于)	> (大于)
!= (不等于)	<= (小于等于)
< (小于)	>= (大于等于)

特别要说明一下，比较运算中等于必须是两个等号（单等号的作用是赋值）。还有像小于等于和大于等于，<和=之间不能留有空格，否则编译不通过。

当然，除了比较运算符外，程序也可以用+、-、*、/（加、减、乘、除）这些常用的算术运算符。

现在知道 for 循环是如何运作的了吧！我们的主函数 loop()中有三个 for 循环，按照下面表格运行，最终完成 SOS 的输出。

第一个 for 循环	短闪烁 3 次，代表输出 3 个点，也就是字母 “S”
第二个 for 循环	长闪烁 3 次，代码输出 3 个横杠，也就是字母 “O”
第三个 for 循环	短闪烁 3 次，代表输出 3 个点，也就是字母 “S”

局部变量和全局变量

必须要注意的，我们这里要引用一个新的概念：局部变量和全局变量。局部变量，这类变量只在自己的代码块内起作用。像是该程序的 for 循环中定义的变量 x 就属于局部变量，这意味着每个 for 循环都拥有自己独立的 x 变量。这些 x 变量之间互不影响，各自在其所属的循环中控制迭代过程，确保了即使在同一程序中存在多个 for 循环，它们之间的变量也不会产生冲突。还有一种变量叫全局变量，不同之处是，它能在整个程序中起作用，但条件是，必须在 `setup()`、`loop()` 函数外声明。像我们这里的 `ledPin`，就能在整个程序中起作用。

在构建 SOS 求救信号器项目时，为了确保 SOS 信号中的每个字母（S、O、S）能够被清晰区分，我们在每个 for 循环迭代完成（表示一个字母）后，特意插入了 1000 毫秒的延时。这个延时使得 SOS 字母之间的停顿变得明显，有助于接收方准确识别每个字母。而在整个 SOS 信号序列完成后，即三个字母（S、O、S）均通过 LED 灯的闪烁表示完毕后，我们进一步设置了一个 5 秒的延时。这个较长的延时是在程序准备重新进入主循环 `loop()` 并执行下一轮 SOS 信号之前的一个等待时间，确保了 SOS 信号之间有足够的间隔，避免信号间的混淆。

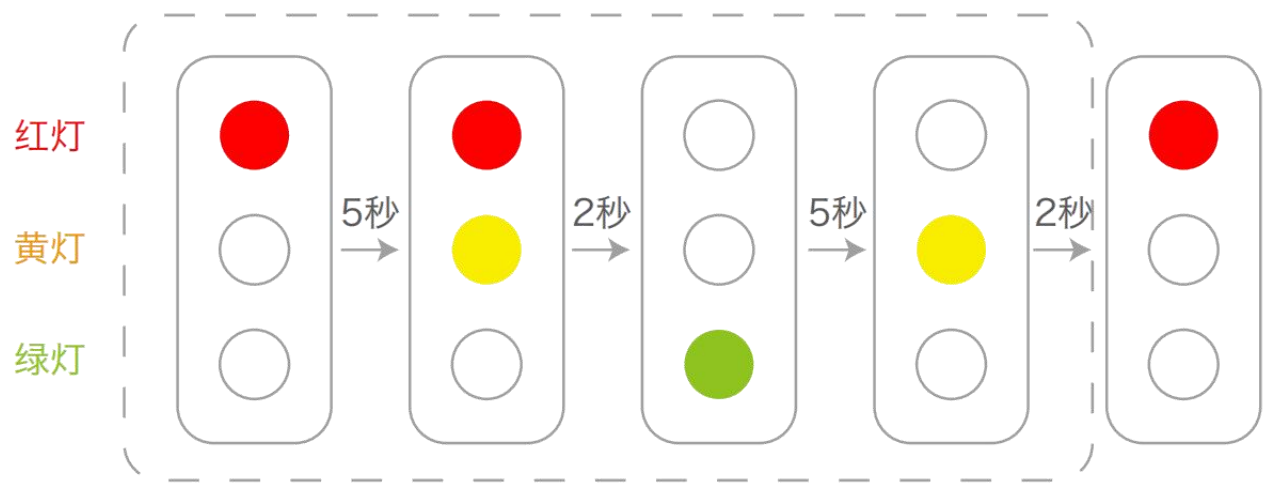
好了，我们 SOS 求救信号器项目就算告一个段落了。有所收获吗？

课后练习

我们以【闪烁第一个 LED】和【SOS 求救信号器】两个项目作为基础，现在尝试自己做个交通信号灯吧，下图是交通信号灯的运行过程，虚线框的是程序的循环部分。自己动手试一下吧！

提示：原先我们只控制了一个 LED 灯，现在要扩展到同时控制三个不同颜色的 LED 灯。电路连接的基本原理保持不变，但我们需要利用三个不同的数字端口

来分别控制这三个 LED 灯。在编程时，关键在于通过调整这些数字端口的高低电平信号，来独立控制每个 LED 灯的亮灭状态。



项目 - 交通信号灯

有没有试着做【SOS 求救信号器】的课后作业呢？做出来的话，说明你已经基本掌握之前所学的东西了，如果不会也没关系，我相信，看完这个章节，前面那个问题就不攻自破了！我们这回就基于课后作业的交通灯来进行一个拓展，增加一种行人按键请求通过马路的功能。当按键被按下时，Arduino 会自动反应，改变交通灯的状态，让车停下，允许行人通过。

这个项目中，我们开始要实现 Arduino 的互动了，也会学习在代码中如何创建自己的函数。这次的代码相对长一点，耐下心来，等看完这一章，相信你能收获不少！

元件清单

	5MM LED 5MM LED灯	x5		Resistor 220R 220欧电阻	x6
	Pushbutton 按键开关	x1			
	Jumper Cables M/M 跳线（公公头）	x13			

*这里 5 个 LED 灯，为什么会用到了 6 个电阻呢？我们知道 5 个电阻是 LED 的限流电阻。还有一个电阻是给按键的，它叫做下拉电阻（我们后面会解释）。

硬件连接

按图 1 的连线图连接你的电路。特别要注意的是，这次连线比较多，注意不要插错。下图中，面包板上标出淡绿色的不是跳线，只是为了说明纵向的孔导通，避免你插错。给 Arduino 上电前认真检查你的接线是否正确。在连线时，保持电源是断开的状态，也就是没有插 USB 线。

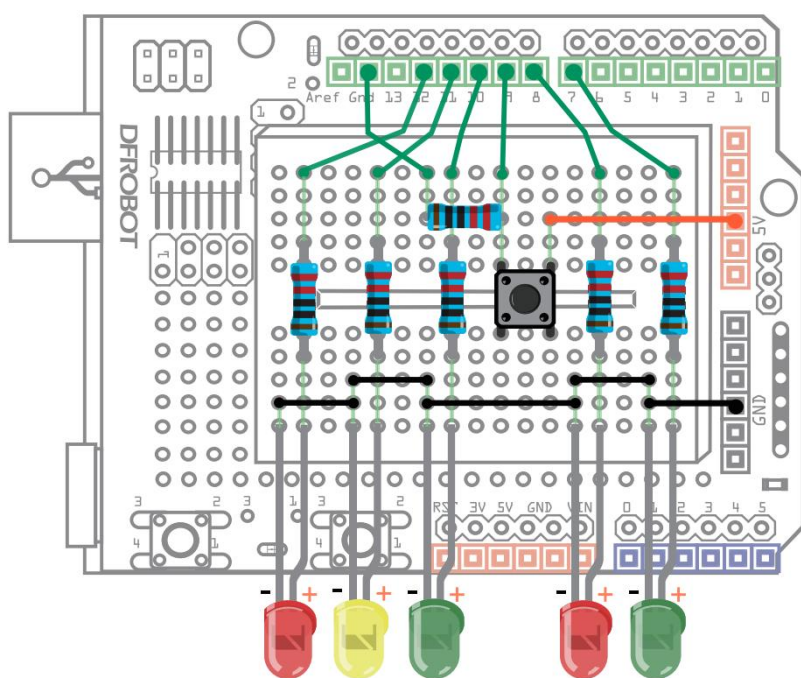


图 1 交通信号灯连接图

示例代码

输入下面的样例代码，这段代码引自《beginning-arduino》一书。

样例代码：

```
// 项目 - 交通信号灯
```

```
int carRed = 12;    // 汽车红灯引脚
```

```
int carYellow = 11; // 汽车黄灯引脚

int carGreen = 10;  // 汽车绿灯引脚

int button = 9;     // 按钮引脚

int pedRed = 8;     // 行人红灯引脚

int pedGreen = 7;   // 行人绿灯引脚

int crossTime = 5000; // 允许行人通过的时间（毫秒）

unsigned long changeTime; // 按钮按下后的时间

void setup() {
    // 初始化引脚模式

    pinMode(carRed, OUTPUT);

    pinMode(carYellow, OUTPUT);

    pinMode(carGreen, OUTPUT);

    pinMode(pedRed, OUTPUT);

    pinMode(pedGreen, OUTPUT);

    pinMode(button, INPUT); // 按钮设置为输入模式

    // 初始状态设置

    digitalWrite(carGreen, HIGH); // 车行绿灯亮

    digitalWrite(pedRed, HIGH);   // 人行红灯亮
}

void loop() {
```

```
int state = digitalRead(button);

// 检测按钮状态及时间间隔

if (state == HIGH && (millis() - changeTime) > 5000) {

    // 调用变灯函数

    changeLights();

}

}

void changeLights() {

    // 汽车绿灯变黄灯

    digitalWrite(carGreen, LOW);

    digitalWrite(carYellow, HIGH);

    delay(2000); // 等待 2 秒

    // 汽车黄灯变红灯

    digitalWrite(carYellow, LOW);

    digitalWrite(carRed, HIGH);

    delay(1000); // 为安全考虑等待 1 秒

    // 行人红灯变绿灯

    digitalWrite(pedRed, LOW);

    digitalWrite(pedGreen, HIGH);
```

```
// 行人绿灯亮，持续 crossTime 毫秒  
  
delay(crossTime);  
  
// 闪烁行人绿灯，提示时间快到  
  
for (int x = 0; x < 10; x++) {  
    digitalWrite(pedGreen, HIGH);  
    delay(250);  
    digitalWrite(pedGreen, LOW);  
    delay(250);  
}  
  
// 行人红灯再次亮起  
  
digitalWrite(pedRed, HIGH);  
delay(500);  
  
// 恢复到汽车绿灯亮状态  
  
digitalWrite(carRed, LOW);  
digitalWrite(carYellow, HIGH);  
delay(1000);  
digitalWrite(carYellow, LOW);  
digitalWrite(carGreen, HIGH);  
  
// 更新 changeTime
```

```
    changeTime = millis();  
}
```

下载完成后，可以尝试按下按键。看看是个什么的效果？我们可以看到整个变化过程是这样的——开始时，汽车灯为绿灯，行人灯为红灯，代表车行人停。一旦行人按键被按下，请求过马路，那么汽车灯由绿变黄，变红，接着行人灯就由红变绿。在行人通行的过程中，设置了一个过马路的时间 `crossTime`，一旦到点，行人绿灯开始闪烁，提醒行人快速过马路。闪烁完毕，最终，又回到了开始的状态，汽车灯为绿灯，行人灯为红灯。

整段代码看起来很复杂，其实理清一下思路并不难。如果你还是没有办法理不清里面变化关系的话，可以试着画一个示意图，像项目【SOS 求救信号器】的课后作业示例图那样，这样一来可能会方便你理解程序。

代码回顾

通过前面两个项目，你应该能够理解这个代码的大部分内容。代码开始是一串的变量的声明，在声明中，出现了一个新名词 `unsigned long`。这里就解释一下这个新名词：

```
unsigned long changeTime;
```

这是一个新的变量类型。我们之前，只创建过 int 整型变量，它可以存放一个 -32768 到 32767 之间的整数。这次要创建的是一个 long 的变量类型，它可以存放一个 -2147483648 到 2147483647 之间的整数。而 unsigned long 数据类型不存储负数，所以存储的范围就变成了 0 到 4294967295。

如果我们使用一个 int 型的话，信号灯状态变化的时间，只能存储最大 32 秒（32768 毫秒约为 32 秒），一旦出现变量溢出就会造成程序运行出现错误，所以为了避免这样的情况，要选用能存储更大数的一个变量类型，并且不为负，我们就可以考虑使用 unsigned long 型。你可以用笔计算下，这个变量最大能存储的数，时间可达 49 天。

随即进入 setup() 函数，对 LED 和按钮键进行一些设置，在设置时需要注意的是：

```
pinMode(button, INPUT);
```

pinMode() 函数我们已经很熟悉了，在项目一的时候就介绍过，只是和 LED 有所不同的是，按键要设置为 INPUT。

在 setup() 函数中，先给定行人灯和汽车灯的一个初始状态：

```
digitalWrite(carGreen, HIGH); //开始时，汽车灯绿灯
```

```
digitalWrite(pedRed, LOW); //行人灯为红灯
```

进入到的主程序中的第一句，就是来检测 button（引脚 9）的状态的：

```
int state = digitalRead(button);
```

变量这个盒子无限大吗？

有人会问为什么有些变量类型可以存储很大的数，而有些变量类型不行呢？这是由变量类型所占的存储空间决定的。就拿我们前面讲变量的时候举过的例子，变量好比用来放东西的盒子，把不同类型的变量想象成不同大小的盒子，int 的盒子比 unsigned long 的盒子小，所以放的东西当然少啦！这样解释是不是比较容易理解这个概念呢？

那又有人问，设置不同大小的盒子干嘛呢？一样大不就行啦，都设置的大一点。理论上没有什么不可以的，可是我们不能忽略一个问题，那就是微控制器的内部存储容量是有限定的。电脑有内存，我们的微控制器同样有内存。像 Arduino UNO 板上用的主芯片 Atmega328 最大内存是 32K。所以，我们要尽量少的用存储空间，能不用则不用。

下表列出了程序中可能用到的变量数据类型：

数据类型	RAM	范围
boolean (布尔型)	1 byte	0 ~ 1 (True 或 False)
char (字符型)	1 byte	-128 ~ 127
unsigned char (无符号字符型)	1 byte	0 ~ 255
int (整型)	2 byte	-32768 ~ 32768

unsigned int (无符号整型)	2 byte	0 ~ 65535
long (长整型)	4 byte	-2147483648 ~ 2147483647
unsigned long (无符号长整型)	4 byte	0 ~ 4294967295
float (单精度浮点型)	4 byte	-3.4028235E38 ~ 3.4028235E38
double (双精度浮点型)	8 byte	-1.7976931348623157E+308 ~ 1.7976931348623157E+308

从上面表格可以看到，变量的类型有很多，不同的类型数对应不同的变量，int 和 long 是针对整数变量，char 是针对字符型变量，而 float，double 是针对含有小数点的变量。

此时，一个新函数出现了——digitalRead()!

digitalRead(pin)

引脚号

这个函数是用来读取数字引脚状态的，HIGH 还是 LOW（其实 HIGH 还有一种表达就是“1”，LOW 是“0”，只是 HIGH/LOW 更直观）。函数需要一个传递参数给对应的——pin，pin 代表对应的引脚号。这里需要读取的是按键信号，按键所在引脚是数字引脚 9，由于前面做了声明，所以这里用 button 来代替。并且把读到的信号传递给变量

state, 用于后面进行判断。state 为 HIGH 或者说为 1 时, 说明按键被按下了。state 为 LOW 或者 0, 表明按键没被按下。

所以, 可以直接检查 state 的值来判断按键是否被按下:

```
if(state == HIGH && (millis() -  
changeTime)> 5000) {  
    //调用变灯函数  
    changeLights();  
}
```

这里涉及新的语句-- if 语句:

```
if(表达式){  
    语句;  
}
```

if 语句是一种条件判断的语句, 判断是否满足括号内的条件, 如满足则执行花括号内的语句, 如不满足则跳出 if 语句。

表达式是指我们的判断条件, 通常为一些关系式或逻辑式, 也可是直接表示某一数值。如果 if 表达式条件为真则执行 if 中的语句。表达式条件为假, 则跳出 if 语句。

我们代码中, 第一个条件是 state 变量为 HIGH。如果按键被按下, state 就会

变为 HIGH。第二个条件是 millis() 的值减 changeTime 的值大于 5000。这两个条件之间有个 “&&” 符号。这是一种逻辑运算符，表示的含义是两者同时满足。

```
(millis() - changeTime) > 5000)
```

millis() 是一个函数，该函数是 Arduino 语言自有的函数，它返回值是一个时间，是 Arduino 开始运行到执行到当前的时间，也称之为机器时间，就像一个隐形时钟，从控制器开始运行的那一刻起开始计时（单位为毫秒）。

变量 changeTime 初始化时，不存储任何数值，只有在 Arduino 运行之后，将 millis() 值赋给它，它才开始有数值，并且随着 millis() 值变化而变化。通过 millis() 函数不断记录时间，判断两次按键之间的时间是不是大于 5 秒，大于五秒则执行 if 花括号内的语句，如果在 5 秒之内则不执行。这样做的目的是，防止重复按键而导致的运行错误。

if 语句内只有一个函数：

```
changeLights();
```

这是一个函数调用的例子。该函数单独写在了 loop() 函数之外。我们需要使用时，直接写出函数名就可以实现调用了。该函数是 void 型，所以是无返回值、无传递参数的函数。当函数被调用时，程序也就自动跳到它的函数中运行。运行完之后，再跳回主函数。需要特别注意的：函数调用时，函数名后面的括号

不能省，要和所写的函数保持一致。changeLights() 函数内部就不做说明了。

逻辑运算符

前面说到的&&是一个逻辑运算符，常用的逻辑运算符有：

&& —— 逻辑与（两者同时满足）
—— 逻辑或（两者其中一个满足）
! —— 逻辑非（取反，相反的情况）

硬件回顾

按键开关

按键一共有 4 个引脚，图 2 分别显示了正面与背面。而图 3 则说明了按键的工作原理。一旦按下后，左右两侧就被导通了，而上下两端始终导通。

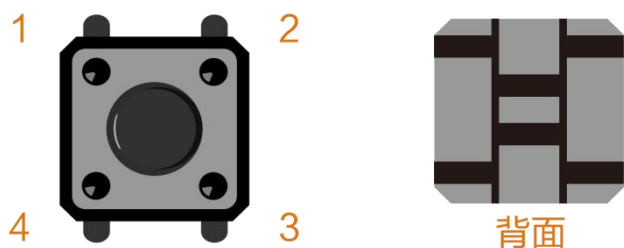


图 2 按键结构图



图 3 按键原理图

在图 4 中，按键仅就是起到一个通断的作用。但在本项目中，按键要能被读取状态，所以除了要将按键的 1、2（或者 3、4；1、4；2、3）引脚连入电路中，还需要接入信号口。按下的话，数字引脚 9 就能检测到为高电平。否则就是保持一个低电平的状态。在我们这个项目中，按钮控制数字引脚是否接高（接 5V）。按下的话，数字引脚 9 就能检测到为高电平。否则就是保持一个低电平的状态（接 GND）。

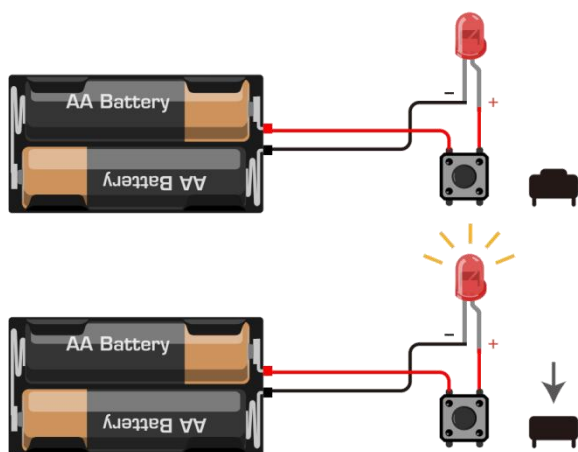


图 4 按键示意图

下拉电阻

下拉电阻这个名词可能比较抽象，从字的含义着手，“下拉”我们就理解为把电压往下拉，降低电压。按键作为开关，当输入电路状态为高电平（high）的时候，电压要尽可能接近 5V；输入电路状态为低电平（low）的时候，电压要尽可能接近 0V。为了避免状态偏离所需电压造成的电压浮动现象如果不能确保

状态接近所需电压，这部分电路就会产生电压浮动。所以，在按键电路中，我们将一个电阻在按钮串联在输入（input）引脚和接地（ground）之间，将按键未被按下的输入状态固定在低电平那里接了一个电阻来确保一定达到 LOW，这个电阻就是所谓下拉电阻。

可以从下面两张图看到，图 5 五是未接下拉电阻的电路，按键未没被按下时，input 输入引脚就处于一个悬空状态。空气会使该引脚电压产生浮动，不能确保是 0V。然而图 6 六是接了下拉电阻的电路，当按键未没被按下时，输入引脚通过电阻接地，确保为 0V，不会产生电压浮动现象。

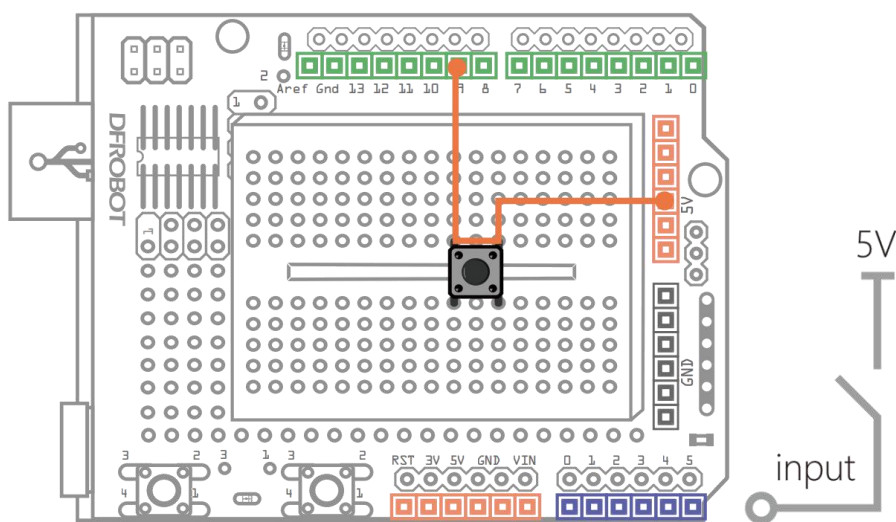


图 55 未接下拉电阻

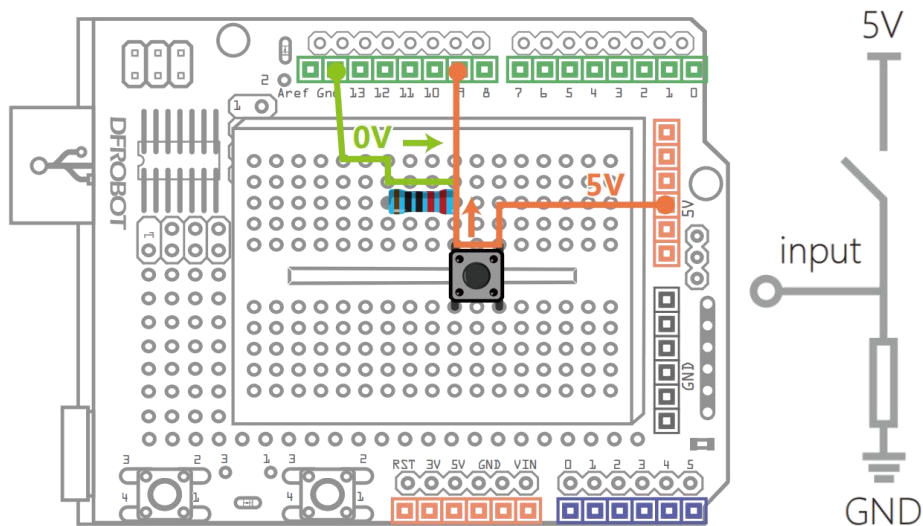
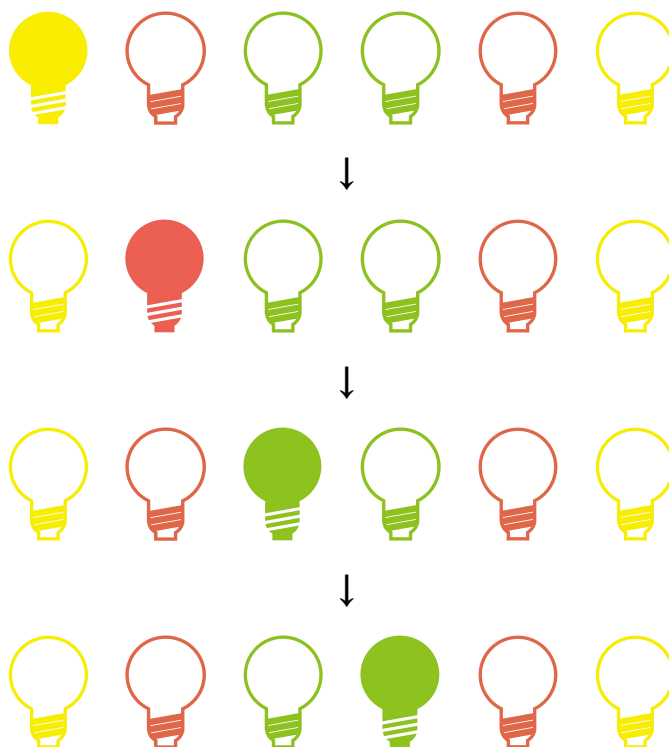
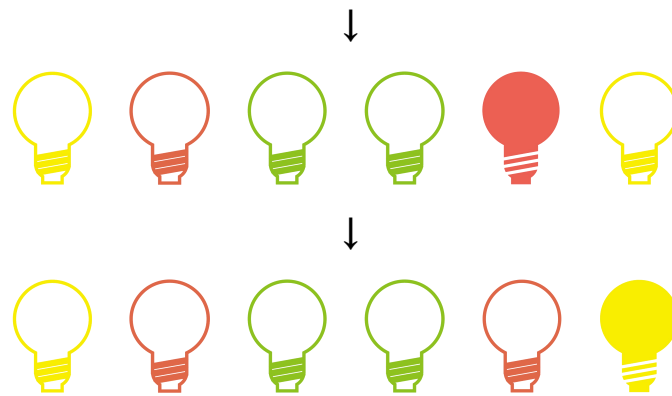


图 66 有下拉电阻

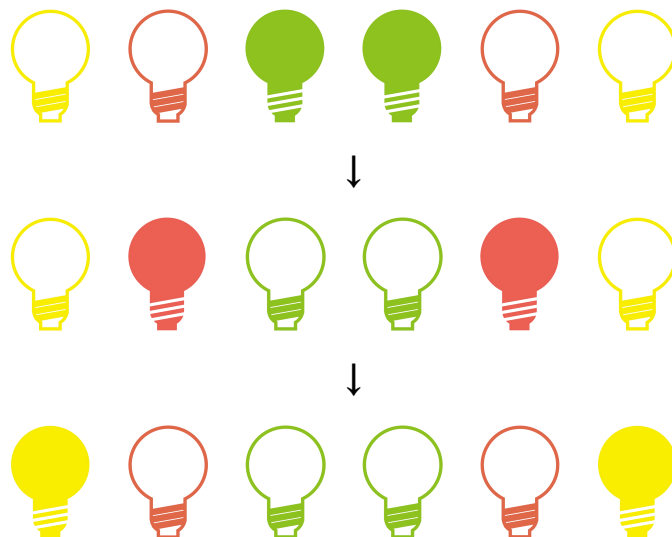
课后练习

1、选择任意颜色 LED 6 个，做一个流水灯的效果，6 盏灯从左至右依次点亮，重复循环这个过程。

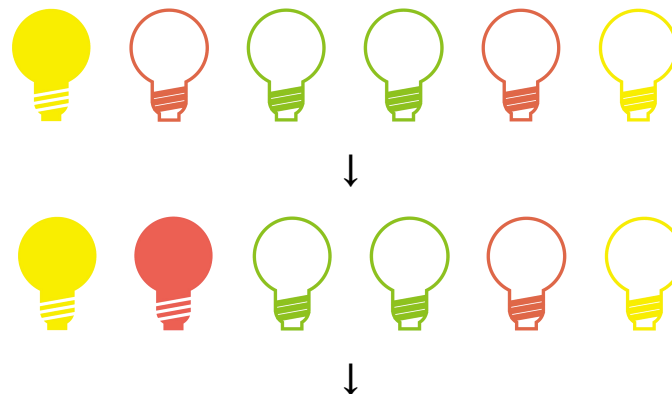


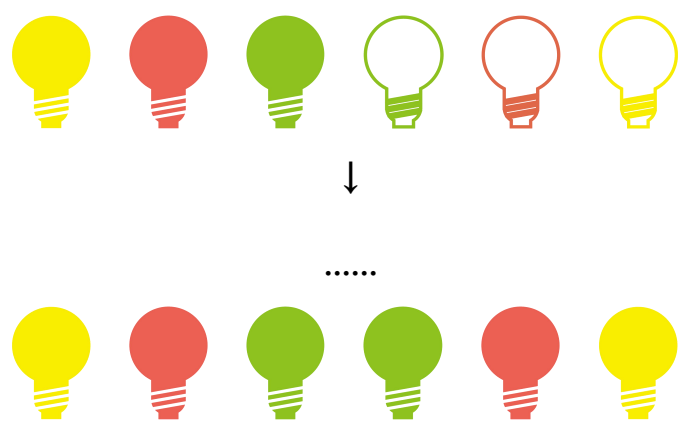


2、如果上面那个你已经完成了的话，可以尝试一下，先从中间的灯开始亮起，依次向两边扩开。下图是个变换过程的示意图。



3、再比如，从左至右，依次亮起 1 个，2 个，3 个.....





项目 - 呼吸灯

在前面几章中，我们知道了如何通过程序来控制 LED 亮灭。但 Arduino 还有个很强大的功能，就是通过程序来控制 LED 的明亮度。Arduino UNO 数字引脚中有六个引脚标有“~”，这个符号就说明该口具有 PWM 功能。我们动手做一个呼吸灯，在做的过程中体会 PWM 的神奇力量！所谓呼吸灯，就是让灯有一个由亮到暗，再到亮的逐渐变化的过程，感觉像是在均匀的呼吸。

元件清单

	5MM LED 5MM LED灯	x1
	Resistor 220R 220欧电阻	x1
	Jumper Cables M/M 跳线（公公头）	x2

硬件连接

这个项目的硬件连接与项目【闪烁第一个 LED】类似。但需要注意连接的引脚有所区别，请参考下面的连线图进行连接。

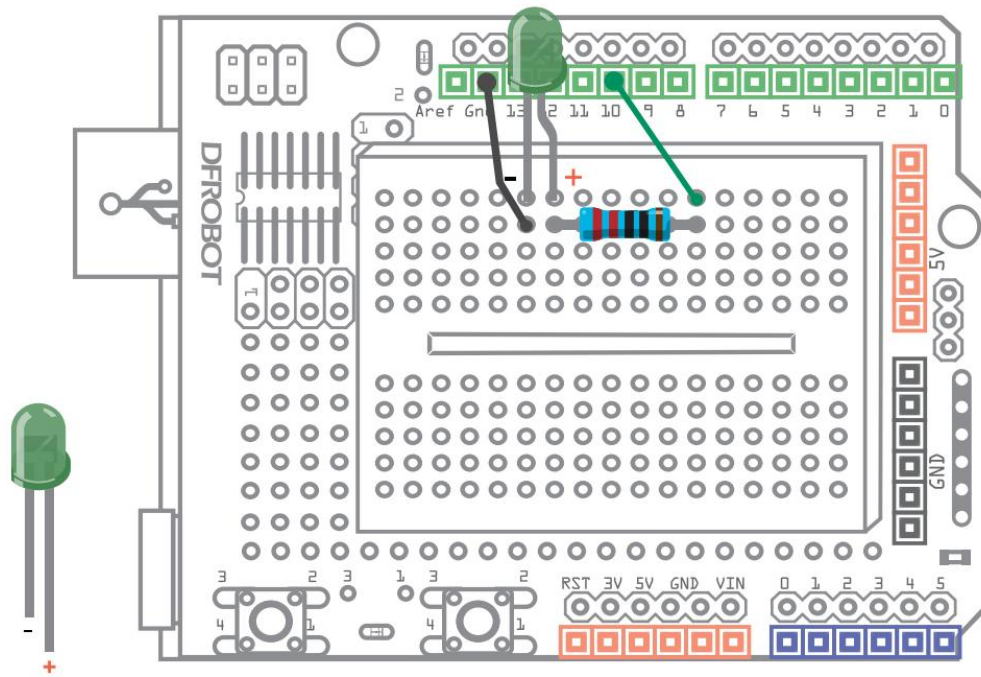


图 1 呼吸灯连线图

示例代码

样例代码：

//项目 - 呼吸灯

```
int ledPin = 10;
```

```
void setup() {  
    pinMode(ledPin,OUTPUT);  
}
```

```
void loop(){  
    fadeOn(1000,5);  
    fadeOff(1000,5);  
}
```

```
}

void fadeOn(unsigned int time,int increament){
    for (byte value = 0 ; value < 255; value+=increament){
        analogWrite(ledPin, value);
        delay(time/(255/5));
    }
}

void fadeOff(unsigned int time,int decreament){
    for (byte value = 255; value >0; value-=decreament){
        analogWrite(ledPin, value);
        delay(time/(255/5));
    }
}
```

代码上传完成后，我们可以看到 LED 会有个逐渐由暗到亮的一个缓慢过程，如同呼吸一般，均匀变化，而不是直接的亮灭。

代码回顾

大部分代码我们已经很熟悉了，比如初始化变量声明、引脚设置、for 循环、以

及函数调用。

在主函数中，只有两个调用函数，先看其中一个就能明白了。

```
void fadeOn(unsigned int time,int increament){  
    for (byte value = 0 ; value < 255; value+=increament){  
        analogWrite(ledPin, value);  
        delay(time/(255/5));  
    }  
}
```

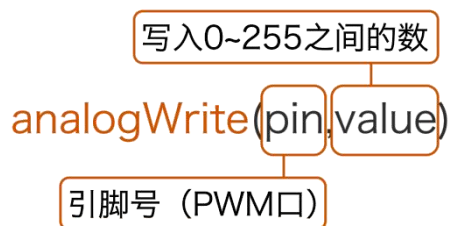
fadeOn()函数有两个传递参数，从参数名称中就可以简单看出，int time 指的是时间，int increament 指的是增量。函数中包含了一个 for 循环，循环条件是 value<255，变量的增量由 increament 决定。

for 语句中涉及了一个新函数：

```
analogWrite(ledPin, value);
```

要想将一个模拟值发送到数字引脚，我们需要使用到这个函数，使用这个函数是要具备特定条件的——该数字引脚需具有 PWM 功能。观察一下 Arduino 板，查看数字引脚，你会发现其中 6 个引脚（3、5、6、9、10、11）旁标有“~”，这些引脚不同于其他引脚，因为它们可以输出 PWM 信号。

函数格式如下：

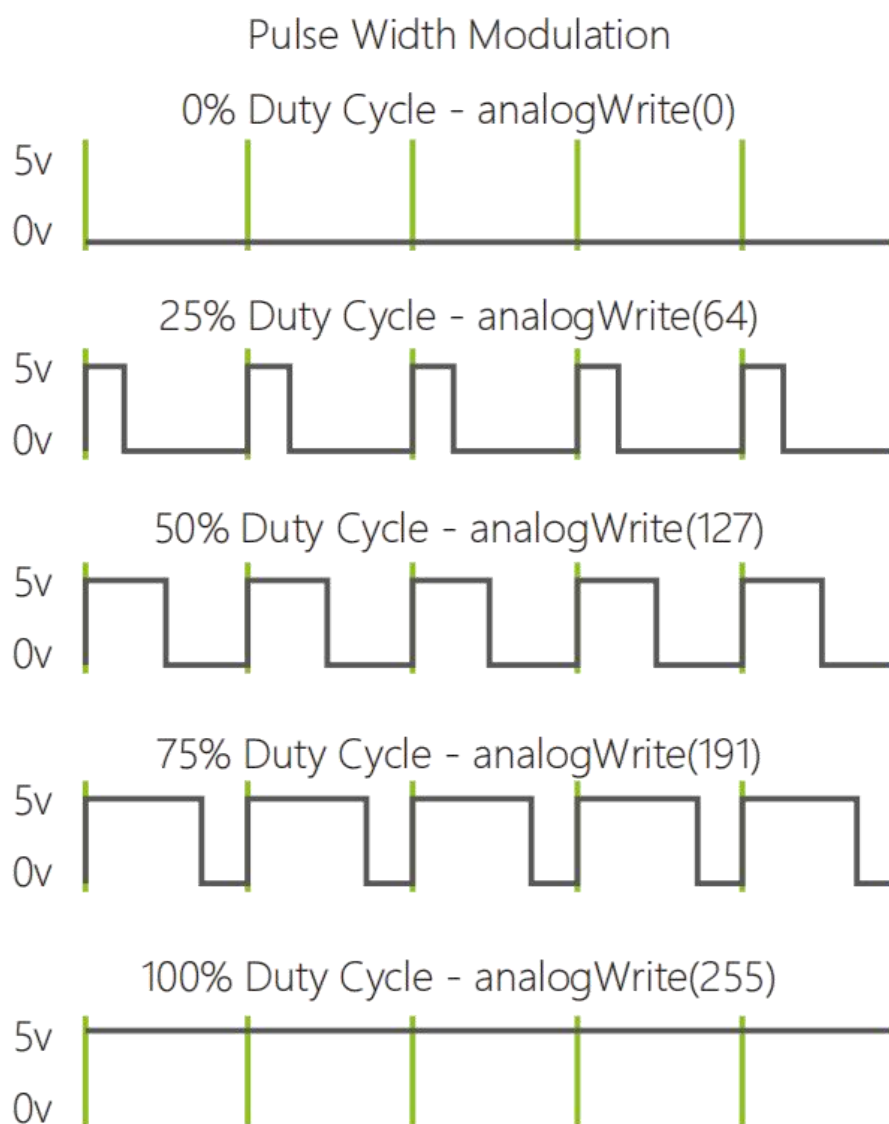


`analogWrite()`函数用于给 PWM 口写入一个 0~255 的模拟值。特别注意的是，`analogWrite()`函数只能对具有 PWM 功能的数字引脚进行模拟写入。

PWM

PWM 是一项通过数字方法来获得模拟量的技术。数字控制来形成一个方波，方波信号只有开关两种状态（也就是我们数字引脚的高低）。通过控制开与关所持续时间的比值就能模拟到一个 0 到 5V 之间变化的电压。开（学术上称为高电平）所占用的时间就叫做脉冲宽度，所以 PWM 也叫做脉冲宽度调制。

通过下面五个方波来更形象的了解一下 PWM。



上图绿色竖线代表方波的一个周期。每个 `analogWrite(value)` 中写入的 `value` 都能对应一个百分比，这个百分比也称为占空比(Duty Cycle)，指的是一个周期内高电平持续时间比上低电平持续时间得到的百分比。

图中，从上往下，第一个方波，占空比为 0%，对应的 `value` 为 0。LED 亮度最低，也就是灭的状态。

高电平持续时间越长，也就越亮。所以，最后一个占空比为 100% 的对应 `value` 是 255，LED 最亮。50% 就是最亮的一半了，25% 则更暗一些。

PWM 比较多的用于调节 LED 灯的亮度，或者是电机的转动速度。在玩一些 Arduino 小车时，更能体现 PWM 的好处，通过控制电机转动的速度来控制车速，还能通过控制左右电机的速度来控制转向。

这一章介绍结束了！同样的硬件连接，通过软件的变化，可以呈现出完全不同的效果，是不是觉得 Arduino 很神奇！

课后练习

1、用 LED 能否做个火焰的效果，通过 PWM 使 LED 产生随机的亮度变化，来模拟一个火焰闪烁的效果。用个浅色罩子盖住效果更佳，可以放在家中作为小夜灯。

主要材料：一个红色 LED、两个黄色 LED 以及三个 220 欧电阻。在这个练习中，推荐使用 random()函数来实现火焰不规则变化的效果。

random()函数的功能是产生一定范围内的随机数。

提示：可以先给 LED 设定一个基础亮度，在基础亮度值附近产生一个随机数，比如 random(-60, 60)+135，让其值稳定在 135 附近，并产生小幅变化，就更具有火焰跳跃感。不妨写程序来尝试一下。

具体用法可以查看下面链接的编程参考手册，里面会详细介绍这个函数的用法。

之后的学习中，如果遇到了新的函数或者想尝试使用没有讲解过的函数，你可以通过编程参考手册来学习某个新函数。

点击查看：DFRobot 中文版 Arduino 编程参考手册(位置：入门教程- Arduino 编程参考手册)

<http://wiki.dfrobot.com.cn/index.php/%E9%A6%96%E9%A1%B5#.E5.85.A5.E9.97.A8.E6.95.99.E7.A8.8B>

点击查看：Arduino 官方编程参考手册：

<http://arduino.cc/en/Reference/HomePage>

2、再尝试一个稍微有点难度的，通过两个按键，一个按键控制 LED 逐次变亮，另一个按键控制 LED 逐次变暗。

项目 - 炫彩 LED

这个项目介绍一种新的 LED——RGB LED。

之所以叫 RGB，是因为这个 LED 是由红（Red）、绿（Green）和蓝（Blue）三色组成。我们电脑的显示器也是由一个个小的红、绿、蓝点组成的。可以通过调整三个 LED 中每个灯的亮度就能产生不同的颜色。这个项目就是教你通过一个 RGB 小灯随机产生不同的炫彩颜色。找出所需元件，按下图连接硬件并输入代码。

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

连接之前，先判别 RGB 是共阴还是共阳，如果不是很清楚的，可以先跳到这个项目的硬件部分介绍。连接时，还需注意一点，引脚的顺序，可参照左边的引

脚图。

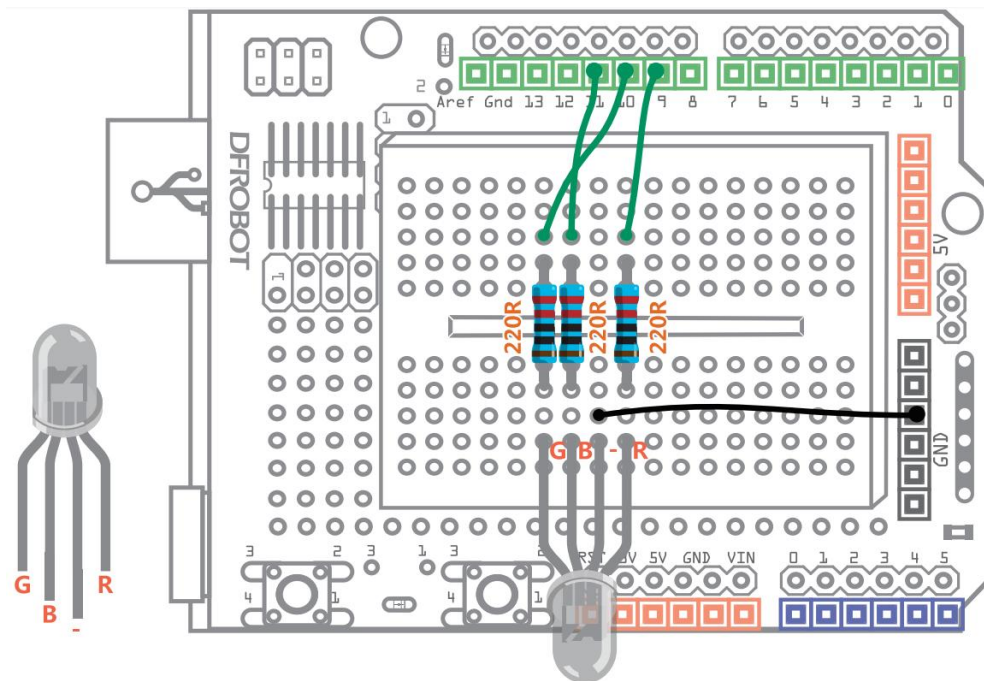


图 1 炫彩 LED 硬件连接图

示例代码

样例代码：

//项目 - 炫彩 RGB

```
int redPin = 9;
```

```
int greenPin = 10;
```

```
int bluePin = 11;
```

```
void setup(){
```

```
    pinMode(redPin, OUTPUT);
```

```
    pinMode(greenPin, OUTPUT);

    pinMode(bluePin, OUTPUT);

}

void loop(){

    colorRGB(random(0,255),random(0,255),random(0,255));
//R:0-255 G:0-255 B:0-255

    delay(1000);

}

void colorRGB(int red, int green, int blue){

    analogWrite(redPin,constrain(red,0,255));

    analogWrite(greenPin,constrain(green,0,255));

    analogWrite(bluePin,constrain(blue,0,255));

}
```

代码下载完成后，我们可以看到 LED 颜色呈现随机的变化，不只是单一的一种颜色。

代码回顾

来分析一下，其实一个 RGB 灯，就是我们前面讲的单色 LED 的结合体，内部集

成了三个 LED，也就需要用三个数字 PWM 口来控制。在我们程序开头部分可以看到定义了三个引脚，并设置为输出模式。

主函数中调用了一个自己创建的函数 `colorRGB()`，函数有三个传递参数，用于写入 Red、Green、Blue 的值，也就是 0~255 的值。

使用函数的好处在于，之后我们想调到某个颜色的时候，只有直接给这三个参数赋值就可以了。不需要重复写 `analogWrite()` 函数，使程序变得冗长。

这段函数中，我们比较陌生的就是 `constrain()` 和 `random()` 这两个函数。

函数格式如下：



`constrain()` 函数需要 3 个参数：x、a 和 b。这里 x 是一个被约束的数，a 是最小值，b 是最大值。如果值小于 a，则返回 a。如果大于 b，则返回 b。

回到我们的程序，red、green、blue 值是被约束数，约束范围在 0~255，也就是我们 PWM 值的范围。它们的值来源于 `random()` 函数随机产生。

函数格式如下：



`random()`函数用于生成一个随机数，`min` 是随机数的最小值，`max` 是随机数的最大值。如果想知道 `random()`函数的其他用法，可以参看手册。

<https://wiki.dfrobot.com.cn/Arduino%E7%BC%96%E7%A8%8B%E5%8F%82%E8%80%83%E6%89%8B%E5%86%8C#random>

硬件回顾

RGB 灯

RGB 灯有 4 个引脚，R、G、B 三个引脚连接到信号传输口上，还有一个引脚是共用的正极（阳）或者共用的负极（阴）。我们这里选用的是共阴 RGB。图 2 展示了三个 LED 如何华丽蜕变为一个 RGB LED 的过程，R、G、B 其实就是三个 LED 的正极，把它们的负极拉到一个公共引脚上了，它们公共引脚是负极，所以称之为共阴 RGB。

RGB 灯如何使用？如何实现变色？

RGB 只是简单的把三个颜色的 LED 灯封装在一个 LED 中，只需要将这三个颜色的 LED 分别进行控制即可。我们都知道红色、绿色、蓝色是三原色，Arduino 通过 PWM 口对三种颜色明暗的调节，即 `analogWrite(value)` 语句，就能让 RGB

LED 调出任何你想要的颜色。

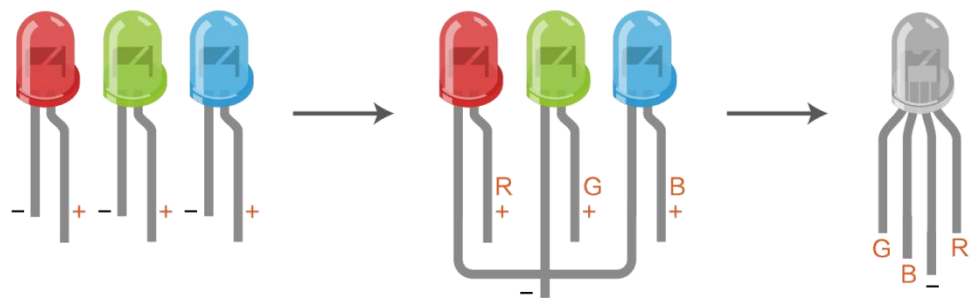


图 2 三个 LED 蜕变成一个 RGB 的过程

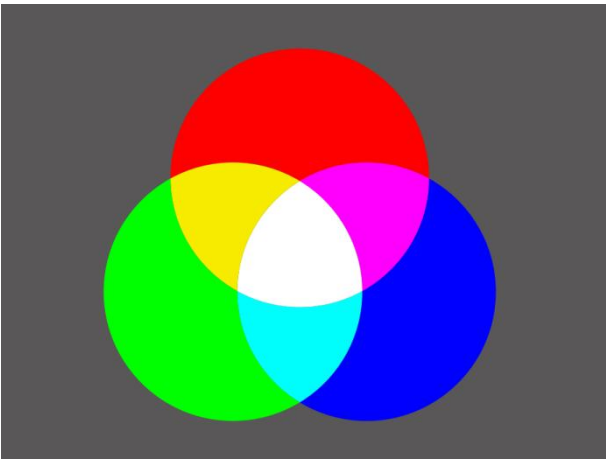


图 3 混合 RGB 产生不同颜色

表 1 只是展示了几种典型的混合颜色，可调的色彩远多于下表所示的，使用 PWM 可以产生 0~255 之间的全部颜色,共 16777216 种颜色(256×256×256)。不妨动手尝试一下，设置三个 LED 的 PWM 值来随意切换颜色吧！

红色	绿色	蓝色	颜色
255	0	0	红色
0	255	0	绿色
0	0	255	蓝色
255	255	0	黄色
0	255	255	蓝绿色
255	0	255	紫红色
255	255	255	白色

表 1 不同 LED 的 PWM 值所组合产生的颜色

共阳 RGB 与共阴 RGB 的区别

上面我们还遗留一个问题——共阴与共阳在使用上有什么区别？共阳 RGB 就是把正极拉到一个公共引脚上，其他三个端则是负极。下图是可以看出，外表上共阴共阳没有任何区别。然而在使用上是有区别的，区别分为以下两点：

- (1) 接线中的改变。共阳的话，共用端需要接 5V，而不是 GND，否则 LED 不能被点亮。
- (2) 第二点就是，在颜色的调配上，与共阴是完全相反的。

举个例子：共阴 RGB 显示红色为 R-255, G-0, B-0。然而共阳则完全相反，RGB 数值是 R-0, G-255, B-255。

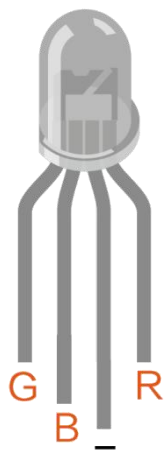


图 4 共阴 RGB 示意图

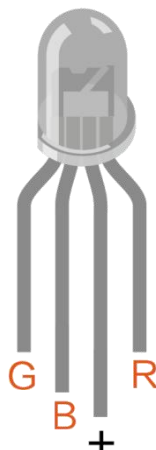


图 5 共阳 RGB 示意图

如何判断共阴还是共阳

其实非常简单，甚至都不需要编写程序来验证。前面讲过 RGB LED 其实是三个颜色的 LED。共阴和共阳的电路图如下：

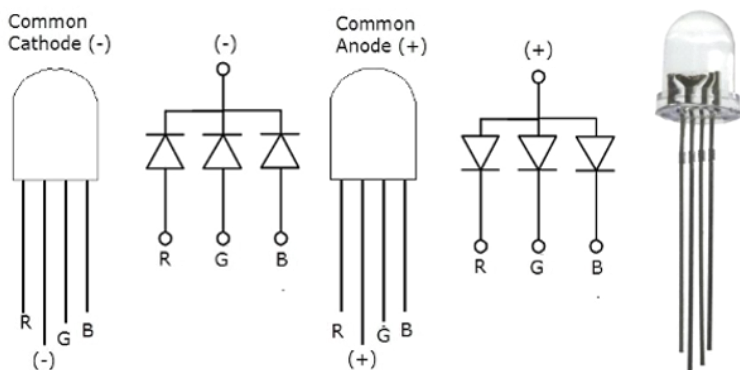


图 6 RGB LED 共阴共阳对比图

测试方法如下：

1. 在 R、G、B 三个引脚各串联一个 220Ω 电阻。
2. 首先将公共端接入负极（接地）。
3. 逐一尝试将 R、G、B 引脚接入正极（如 3V 电源）。
4. 观察 LED 是否亮起对应颜色：若某引脚接入正极时亮起相应颜色（红、绿、蓝），则该 RGB LED 为共阴极；若均不亮，则判定为共阳极。
5. 为确保准确性，可按相同方法测试共阳极情况（即公共端接正极，逐一测试 R、G、B 引脚接地）

课后练习

1、基于我们上面的炫彩 RGB 项目，改变程序做一个沿着彩虹色变化的 RGB 灯，而不是我们这样随机产生颜色。这里比较困难的应该是颜色的调制，通过改变 Red、Green、Blue 的值 0~255，组合出你想要的颜色。

提示：只要在原有代码基础上做修改就可以了，直接调用 `colorRGB()` 函数，将函数中 3 个参数写入所对应颜色的值即可。

2、在作业 1 的基础上，能否结合我们上面说的呼吸灯，将彩虹色以呼吸灯渐变形式变化。这样的变换会显得更加柔和。

3、Arduino 是个开源的平台，从 Arduino IDE 中就可以加载一些别人已经写好了的库，不需要自己从头写，难度也比较大，所以我们只需调用别人写好的

库，来达到我们想要的效果就可以了。

比如我们今天使用的 RGB LED，就已经有现成的用户库了。

下面就提供一个加载 RGB LED 库和使用库的案例。

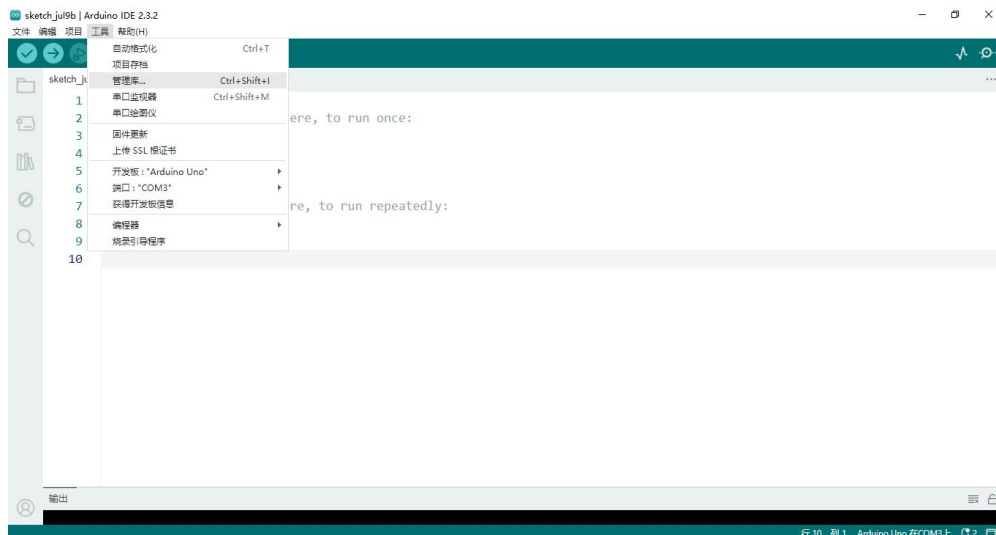


图 7 工具->管理库

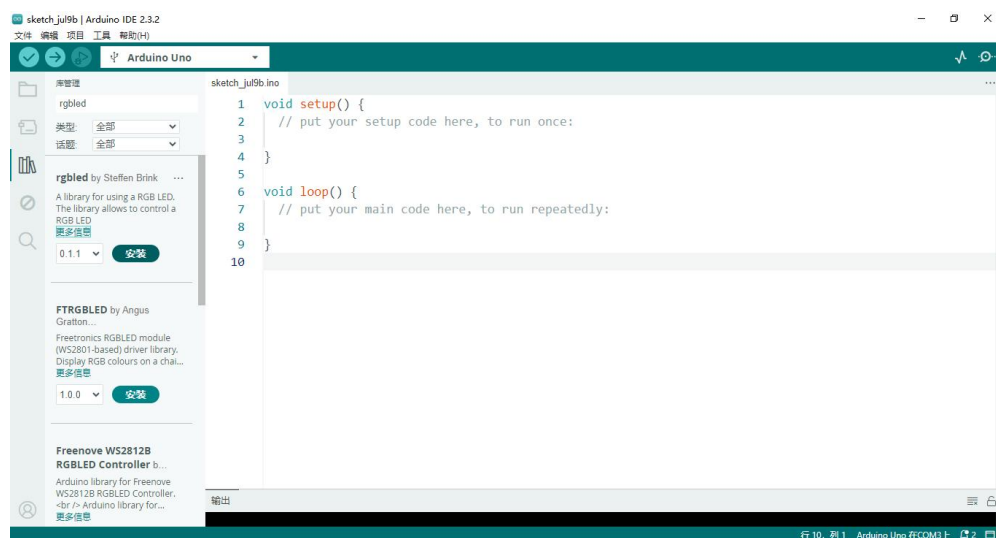


图 8 搜索 rgbled->安装

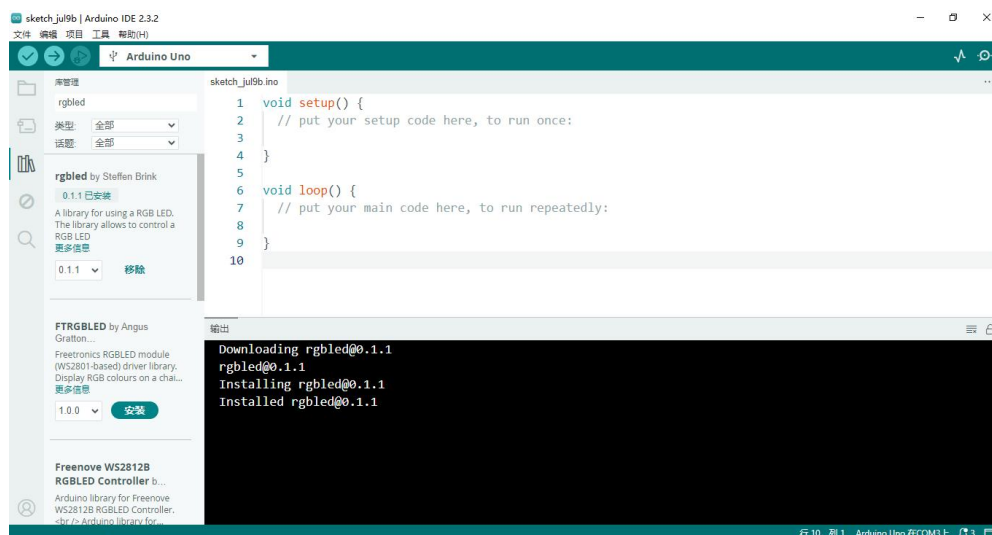


图 9 安装成功

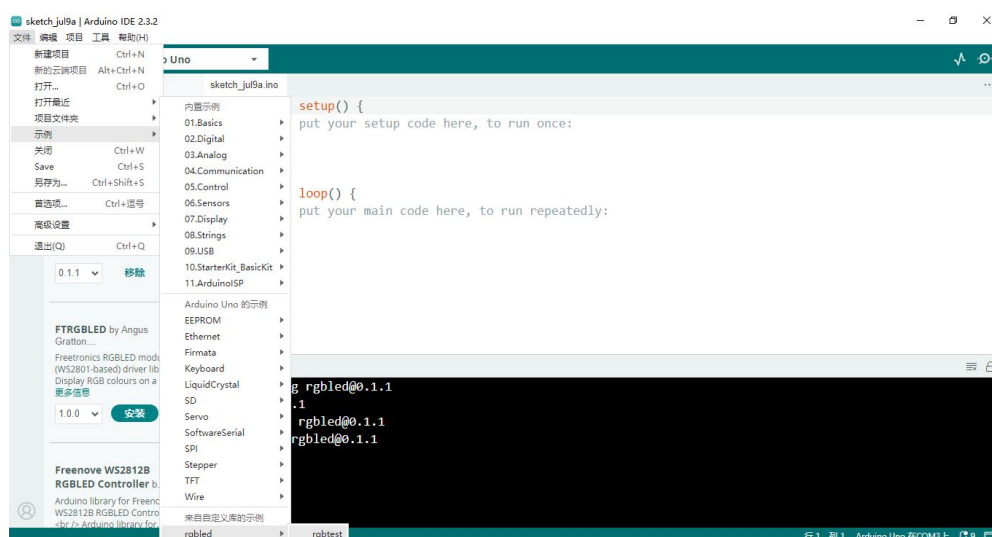


图 10 文件->示例->rgbled

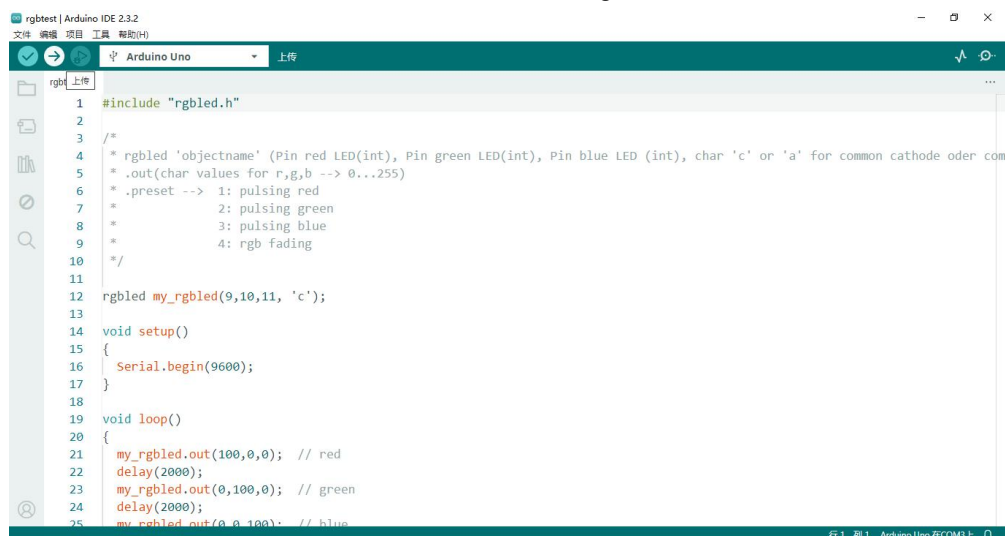


图 11 验证和上传

项目 - 报警器

这里我们要接触一个新的电子元件——蜂鸣器，从字面意思就可以知道，这是一个会发声的元件。这次做一个报警器，通过连接蜂鸣器到 Arduino 数字输出引脚，并配合相应的程序就可以产生报警器的声音。其原理是利用正弦波产生不同频率的声音。如果再结合一个 LED，配合同样的正弦波产生灯光的话，就是一个完整的声光报警器了。

元件清单



Buzzer
蜂鸣器

x1



Jumper Cables M/M
跳线（公公头）

x2

硬件连接

按下图连接图连接，注意本项目蜂鸣器为无源蜂鸣器无正负之分。蜂鸣器的两个引脚分别接到 GND 和数字口 8。

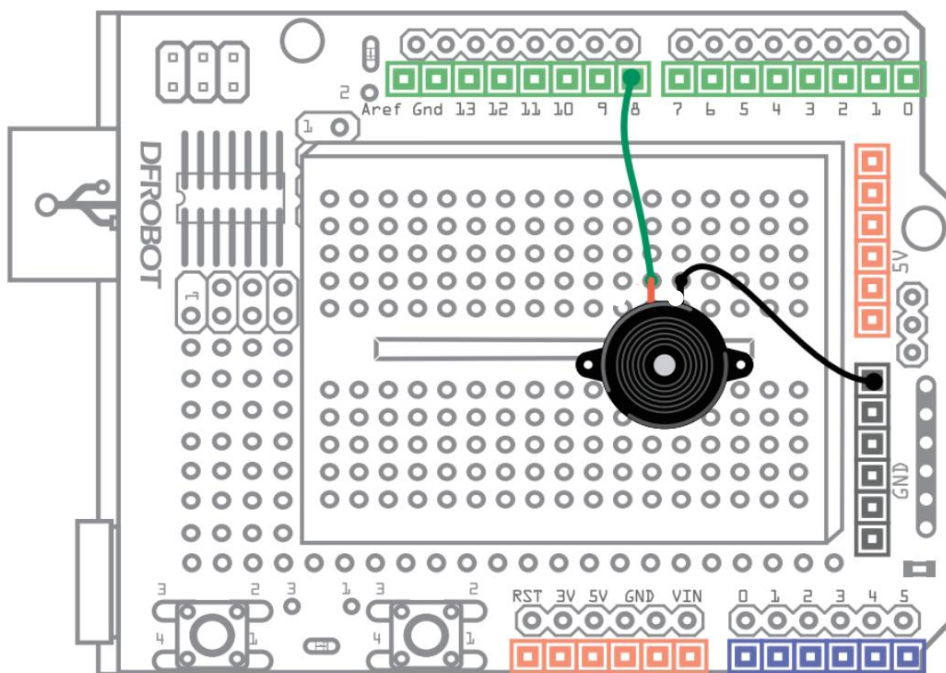


图 1 报警器连线图

示例代码

输入下面的样例代码，这段代码引自《beginning-arduino》一书。

样例代码：

```
//项目 - 报警器
```

```
float sinVal;
```

```
int toneVal;
```

```
void setup(){
```

```
    pinMode(8, OUTPUT);
```

```
}
```

```
void loop(){
```

```
for(int x=0; x<180; x++){  
    //将 sin 函数角度转化为弧度  
    sinVal = (sin(x*(3.1412/180)));  
    //将 sin 函数角度转化为弧度  
    toneVal = 2000+(int(sinVal*1000));  
    //用 sin 函数值产生声音的频率  
    tone(8, toneVal);  
    //将 toneVal 给引脚 8  
    delay(2);  
}  
}
```

下载程序完成后，你会听到平滑的一高一低的报警声，如同汽车报警器。

代码回顾

首先，定义两个变量：

```
float sinVal;  
int toneVal;
```

浮点型变量 sinVal 用来存储正弦值，正弦波呈现一个波浪形的变化，变化比较

均匀，所以我们选用正弦波的变化来作为我们声音频率的变换，toneVal 从 sinVal 变量中获得数值，并把它转换为所需要的频率。

这里用的是 $\sin()$ 函数，一个数学函数，可以算出一个角度的正弦值，这个函数采用弧度单位。如图 2，当弧度位于区间 $0 \sim \pi$ ，也就是角度位于区间 $0 \sim 180$ 度时， $\sin()$ 值为正。因为我们不想让函数值出现负数，所以设置 for 循环在 $0 \sim 179$ 之间，也就是 $0 \leq x < 180$ 。

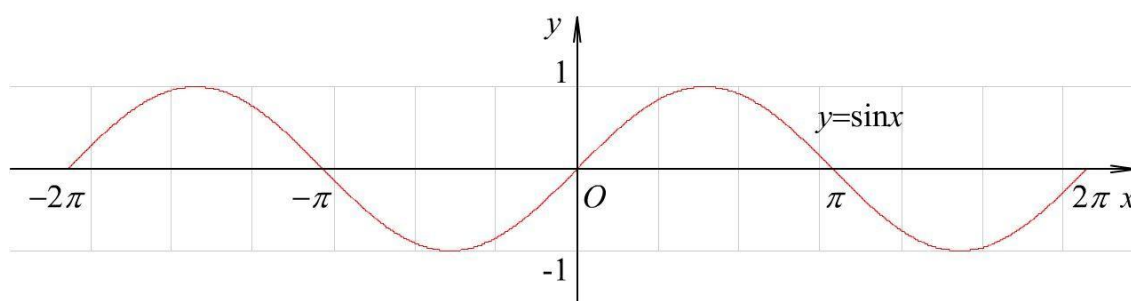


图 2 $\sin()$ 函数

```
for(int x=0; x<180; x++)
```

函数 $\sin()$ 用的弧度单位，不是角度单位。要通过公式 $(3.1412/180)$ 将角度转为弧度：

```
sinVal = (sin(x*(3.1412/180)));
```

这样 sinVal 的值将会根据角度的变化产生像波浪一样的起伏，之后，将这个值转变成相应的报警声音的频率：

```
toneVal = 2000+(int(sinVal*1000));
```

这里有个知识点——浮点型值转换为整型。sinVal 是个浮点型变量，也就是含小数点的值，而我們不希望频率出现小数点的，所以需要有一个浮点值转换为整型值得过程，也就是下面这条语句就完成了这件事：

```
int toneVal;
```

把 sinVal 乘以 1000，转换为整型后再加上 2000 赋值给变量 toneVal，现在 toneVal 就是一个适合声音频率了。之后，我们用 tone()函数把生成的这个频率给我们的蜂鸣器。

```
tone(8, toneVal);
```

下面我们来介绍一下 tone 相关的三个函数：

1. tone(pin, frequency)

Pin 都是指连接到蜂鸣器的数字引脚，frequency 是以 Hz 为单位的频率值。

2. tone(pin, frequency, duration)

第二个函数，有个 duration 参数，它是以毫秒为单位，表示声音长度的参数。

像第一个函数，如果没有指定 duration，声音将一直持续直到输出一个不同频率的声音产生。

3. noTone(pin)

noTone(pin)函数，结束该指定引脚上产生的声音。

硬件回顾

蜂鸣器

蜂鸣器其实就是一种会发声的电子元件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

压电式蜂鸣器和电磁式蜂鸣器区别

压电式蜂鸣器是以压电陶瓷的压电效应，来带动金属片的振动而发声。当受到外力导致压电材料发生形变时压电材料会产生电荷。电磁式的蜂鸣器，则是利用通电导体会产生磁场的特性，通电时将金属振动膜吸下，不通电时依振动膜的弹力弹回。不太明白也没太大关系，不影响我们使用。压电式蜂鸣器需要比较高的电压才能有足够的音压，一般建议为 9V 以上。电磁式蜂鸣器用 1.5V 就可以发出 85dB 以上的音压了，但消耗电流会大大的高于压电式蜂鸣器。所以还是建议初学者使用电磁式蜂鸣器。

有源蜂鸣器和无源蜂鸣器区别

无论是压电式蜂鸣器还是电磁式蜂鸣器，都有有源蜂鸣器和无源蜂鸣器两种区分。

有源蜂鸣器和无源蜂鸣器的根本区别是输入信号的要求不一样。这里的“源”不是指电源，而是指振荡源，有源蜂鸣器内部带振荡源，说白了就是只要一通电就会响。而无源内部不带振荡源，所以如果仅用直流信号无法使其响，必须用 2K-5K 的方波去驱动它。

从外观上看，有源无源的区别在于，有源蜂鸣器有长短脚，也就是所谓正负极，长脚为正极，短脚为负极。而无源蜂鸣器则没有正负极，两个引脚长度相同。

在套件中，我们为初学者选用的蜂鸣器类型是无源蜂鸣器，可以演奏出不同的音乐效果。

蜂鸣器的应用有很多，我们也可以就蜂鸣器做一些好玩的东西，比如常见的会结合蜂鸣器的有，红外传感器，超声波传感器，用于监测物体靠近报警。温度传感器，测到温度过高报警。气体传感器，有气体泄漏报警等等。除了报警，还可以用来作为乐器，通过不同频率，调成乐谱的不同调式，是不是很 amazing?

课后练习

1、结合红色 LED 灯做一个完整的报警器。

提示：可以让 LED 也随着 sin 函数变化，使声音与灯光节奏保持一致。

2、结合项目【交通信号灯】中介绍的按钮，做个简易门铃的效果，每次按下按钮，蜂鸣器发出提示音。

项目 - 温度报警器

在项目【报警器】中，我们认识了一个发声元件——蜂鸣器，也做了一个简单的小报警器。是不是还不过瘾呢？这次我们要做一个更贴近生活的应用——温度报警器。当温度到达我们设定的限定值时，报警器就会响。我们可以用于厨房温度检测报警等各种需要检测温度的场合。这个项目中，除了要用到蜂鸣器，还需要一个 LM35 温度传感器。

我们这里是第一次接触传感器，传感器是什么？简单的从字面上的理解就是，一种能感知周围环境，并把感知到的信号转换为电信号的感应元件。感应元件再把电信号传递给控制器。就好比人的各个感官，感知周围环境后，再将信息传递给大脑是一样的道理。

元件清单



Tem.Sensor
温度传感器

x1



Buzzer
蜂鸣器

x1



Jumper Cables M/M
跳线（公公头）

x5

硬件连接

这个项目中的蜂鸣器和项目【报警器】的接法相同。在接 LM35 温度传感器时，注意三个引脚的位置，有 LM35 字样的一面面向自己，从左至右依次接 5V、Analog 0 和 GND，如下图所示。

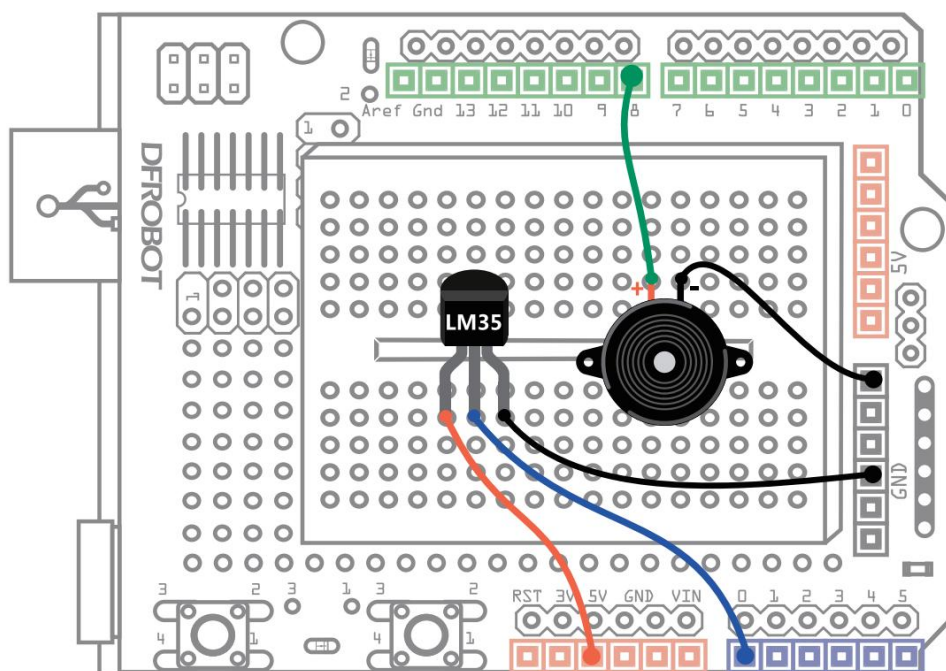


图 1 温度报警器连线图

代码示例

样例代码：

```
//项目 - 温度报警器
```

```
float sinVal;
```

```
int toneVal;
```

```
unsigned long tepTimer ;
```

```
void setup(){

    pinMode(8, OUTPUT);          // 蜂鸣器引脚设置

    Serial.begin(9600);          //设置波特率为 9600 bps

}

void loop(){

    int val;                      // 用于存储 LM35 读到的值

    double data;                  //用于存储已转换的温度值

    val=analogRead(0);           //LM35 连到模拟口，并从模拟口读值

    data = (double) val * (5/1024)*100; // 得到电压值，通过公式换成
    温度

    if(data>27){                  // 如果温度大于 27，蜂鸣器响

        for(int x=0; x<180; x++){

            //将 sin 函数角度转化为弧度

            sinVal = (sin(x*(3.1412/180)));

            //用 sin 函数值产生声音的频率

            toneVal = 2000+(int(sinVal*1000));

            //给引脚 8 一个

            tone(8, toneVal);

            delay(2);

        }

    } else {                      // 如果温度小于 27，关闭蜂鸣器
```

```
noTone(8);           //关闭蜂鸣器

}

if(millis() - tepTimer > 500){ // 每 500ms, 串口输出一次温度值
    tepTimer = millis();

    Serial.print("temperature: "); // 串口输出“温度”
    Serial.print(data);           // 串口输出温度值
    Serial.println("C");          // 串口输出温度单位
}

}
```

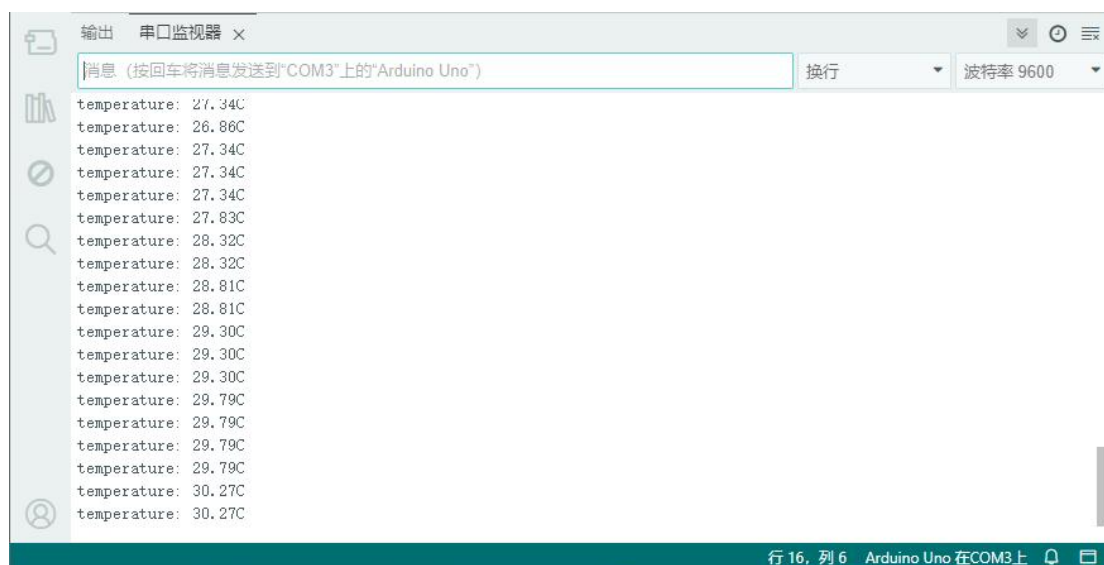
成功下载完程序后，打开 Arduino IDE 的串口监视器。



设置串口监视器的波特率为 9600。



可以直接从串口中读取温度值，并尝试升高周围环境温度，或者用手直接接触 LM35 使其升温，串口可以很直观的看到温度有明显的变化。



蜂鸣器工作的条件是，一旦检测到环境温度大于 27 度，蜂鸣器鸣响，环境温度小于 27 度，则关闭蜂鸣器。

代码回顾

这段代码与我们项目【报警器】的大部分内容是相同的，代码中的大部分语法在前几个项目中已经说过了，现在看起来是不是有点头绪了呢？

程序开始设置了三个变量：

```
float sinVal;  
  
int toneVal;  
  
unsigned long tepTimer ;
```

第一二个变量就不说了，项目【报警器】中代码回顾中已作解释，第三个变量 tepTimer，是一个无符号的长整型（unsigned long）用于存放机器时间，便

于定时在串口输出温度值，由于机器运行时间较长，所以选用一个长整型，又由于时间不为负，则选用无符号长整型，对于变量类型不明确的，可以再回看下项目【交通信号灯】相关解释。

setup()函数的第一句，我们想必已经很熟了，设置蜂鸣器为输出模式，有人可能会问为什么 LM35 不用设置呢？LM35 传输的是个模拟量，模拟量不需要设置引脚模式。pinMode()只用于数字引脚。

串口可用的函数也有好多，可查看语法手册。我们这里就先介绍几个常用的：

```
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。

再到 loop()函数内部，开始部分又声明了两个变量 val 和 data，注释中已对这两个变量进行说明了，这两个变量与前面声明的两个变量不同的是，这两个是局部变量，只在 loop()函数内部起作用。关于全局变量和局部变量的区别，可以参看一下项目【SOS 求救信号器】中说明。

```
val=analogRead(0);
```

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。

我们这里读到的是温度的电压值，是以 0~1023 的方式输出。而我们 LM35 温度传感器每 10mV 对应 1 摄氏度。

```
data = (double) val * (5/1024)*100;
```

val: 这是从温度传感器读取的模拟值，它介于 0 到 1023 之间。

(5/1024): 这个部分是一个比例因子。因为 1024 温度传感器能输出的最大数字值加 1 (从 0 开始计数)，所以 5/1024 表示每增加一个单位的模拟值所增加的电压值。val * (5/1024)代表现在温度对应的电压值，单位为 V (伏特)。

100: 乘以 100 是为了将结果转换为温度。由于每 10mV (即 0.01V) 对应一摄氏度，所以转换后的电压为 val(5/1024)/0.01 (即*100)，就能得到温度。

Arduino 的通信伙伴——串口

串口是 Arduino 与外界进行通信的一个简单的方法。每个 Arduino 都至少有一个串口，UNO 的串口分别与数字引脚 0(RX)和数字引脚 1(TX)相连。所以如果要用到串口通信的，数字 0 和 1 不能用于输入输出功能。

Arduino 下载程序也是通过串口来完成的。所以，当下载程序的时候，USB 将占用了数字引脚 0(RX)和数字引脚 1(TX)。此时，在下载程序的过程中，RX 和 TX 引脚不能接任何东西，否则会产生冲突。可以下载完之后，再接上。

在以后的使用过程中，需要注意，特别是一些无线通信模块，通常会用到 TX、RX。所以下载程序时，需将模块先取下。避免造成程序下载不进去的情况。

后面进入一个 if 语句，对温度值进行判断。这里的 if 语句与之前讲的有所不同。if...else 用于对两种情况进行判断的时候。

if...else 语句格式：

```
if(表达式){  
    语句一;  
}  
else {  
    语句二;  
}
```

表达式结果为真时，执行语句一，放弃语句二的执行，接着跳过 if 语句，执行 if 语句的下一条语句；如果表达式结果为假时，执行语句二，放弃语句一的执行，接着跳过 if 语句，执行 if 语句的下一条语句。无论如何，对于一次条件的判断，语句 1 和语句 2 只能有一个被执行，不能同时被执行。

回到我们的代码，if 中的语句就省略不说了，不明白的可回看项目【报警器】：

```
if(data>27){  
    for(int x=0; x<180; x++){  
        .....  
    }  
} else {  
    .....  
}
```

进入 if 判断，对 data 也就是温度值进行判断，如果大于 27，进入 if 前半段，蜂鸣器鸣响。否则，进入 else 后的语句，关闭蜂鸣器。

除了不断检测温度进行报警，我们还需要代码在串口实时显示温度。这里又用到 millis() 函数（项目三中有说明），利用固定的机器时间，每隔 500ms 定时向串口发出数据。

那串口收到数据后，如何在串口监视器上显示呢？就要用到下面的两句语句：

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

print()的解释是，以我们可读的 ASCII 形式从串口输出。

这条命令有多种形式：

- (1) 数字则是以位形式输出（例 1）
- (2) 浮点型数据输出时只保留小数点后两位（例 2）
- (3) 字符和字符串则原样输出，字符需要加单引号（例 3），字符串需要加双引号（例 4）。

例如：

- (1) `Serial.print(78);`输出 “78” 。
- (2) `Serial.print(1.23456);`输出 “1.23” 。
- (3) `Serial.print(N);`输出 “N” 。
- (4) `Serial.print(“Hello World.”);`输出 “Hello world.” 。

不仅有我们上面这种形式输出，还可以以进制形式输出，可以参看语法手册。

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

串口监视器输出还有一条语句比较常见的是 Serial.write()，它不是以 ASCII 形式输出，而是以字节形式输出，感兴趣的可以查看语法手册。

代码中，可能有一处会不太明白：

```
Serial.print(data);
```

有人会问，data 不是字符串吗？怎么输出是数字呢？不要忘了，这是我们前面定义的变量，它其实就是代表数字，输出当然就是数字啦！

硬件回顾

LM35

LM35 是一种常见的温度传感器，使用简便，不需要额外的校准处理就可以达到 $\pm 0.25^{\circ}\text{C}$ 的准确率。我们看一下 LM35 引脚示意图，Vs 接入电源，Vout 是电压输出，GND 接地。其引脚分布如下图所示：

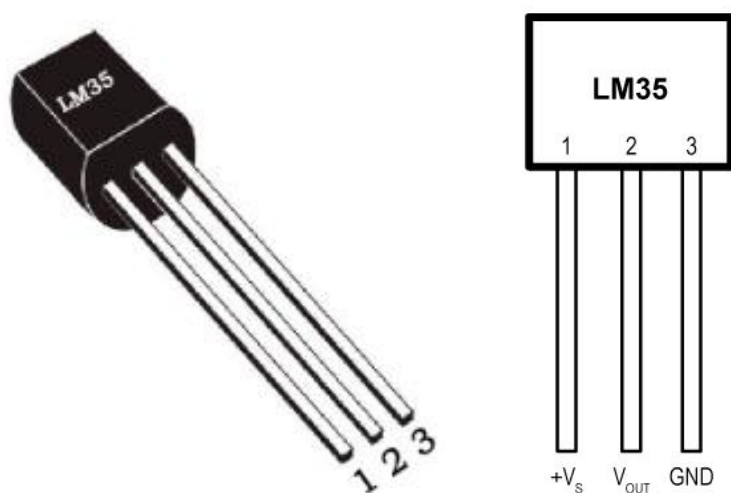


图 2 LM35 引脚示意图

计算公式：

$V_{out} = 10\text{mV}/^{\circ}\text{C} * T^{\circ}\text{C}$ （温度范围在 $+2^{\circ}\text{C} \sim 40^{\circ}\text{C}$ ）这个公式哪里来的呢？如果我们换做其他的温度传感器该怎么换算呢？这里提供一个网址，专门用来查芯片的使用说明书，也叫做 datasheet。Datasheet 会提供出厂芯片所有的性能参数，以及一些简单典型电路的搭建也会告诉你。以后碰到其他的传感器，

不同的芯片就能通过这个方法来得得到计算公式。

ALLDATASHEET:

<http://www.alldatasheet.com/>

我们试一下搜索 LM35，下图公式就是截取自 LM35 的 datasheet 中。图中显示的就是 LM35 的计算公式。

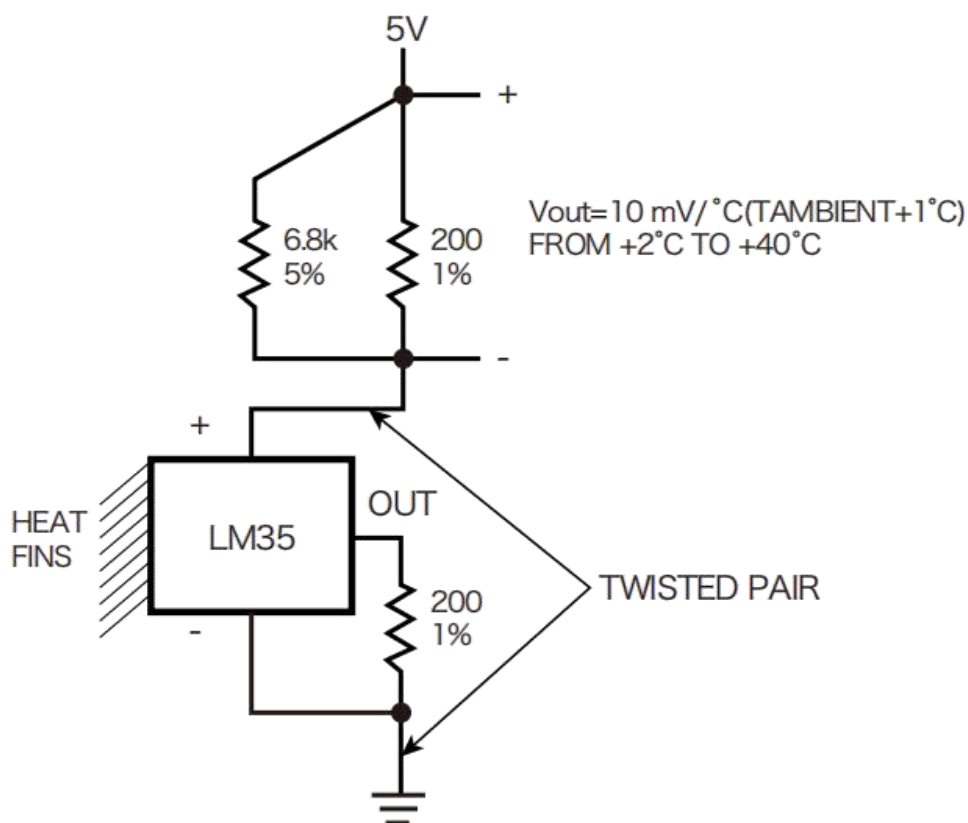


图 3 LM35 计算公式

课后练习

温度报警器可以用于监控对环境温度有要求的一些东西，比如植物。将我们上面的温度报警器再结合 LED 灯。在不同的温度范围设置不同颜色灯，并伴随不同频率的声音。

比如：温度小于 10 或者大于 35，亮红灯，蜂鸣器发出比较急促的声音。

温度在 25~35 之间，亮黄灯，蜂鸣器伴随相对缓和的声音。

温度在 10~25 之间，亮绿灯，关闭蜂鸣器。

发挥你的想象，看看还能玩出什么好玩的东西？

项目 - 震动探测

震动探测，我们从名字中应该就可以判断，该装置能够检测物体是否在震动。

我们用什么来做震动探测呢？那就是倾斜开关。其内部含有导电珠子，器件一旦震动，珠子随之滚动，就能使两端的导针导通。

通过这个原理，我们可以做一些小玩具结合起来。最常见的，比如我们看到一些小孩子穿的一闪一闪的小鞋子！走动的过程，就能使内部珠子滚动。

只要传感器检测到东西震动，就会有信号输出。这里，我们想通过倾斜开关做个简单的震动传感器，并把震动传感器和 LED 的结合，当传感器检测到物体震动时，LED 亮起，停止震动时，LED 熄灭。

元件清单



Resistor 220R
220欧电阻

x2



5MM LED
5MM LED灯

x1



Tilt Switch Sensor
倾斜开关

x1



Jumper Cables M/M
跳线（公公头）

x5

硬件连接

从倾斜开关这个名字，我们可以把它和什么联想在一起？就是按键开关，倾斜开关和我们项目【交通信号灯】中介绍的按钮在硬件连接是完全相同的，原理也相似。只是使用方法不同而已。可以把下图与项目【交通信号灯】一起看，你会发现很多相似之处。倾斜开关也需要一个下拉电阻，LED 需要一个限流电阻。

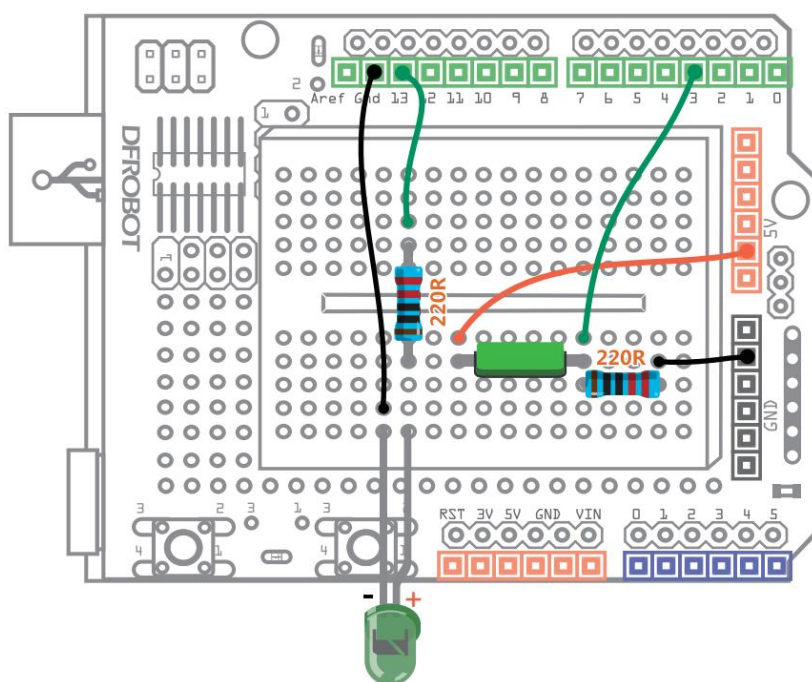


图 1 震动探测连线图

示例代码

样例代码：

//项目 - 震动探测

```
int SensorLED = 13;           //定义 LED 为数字引脚 13
```

```
int SensorINPUT = 3;          //连接倾斜开关到中断 1，也就是数字引脚 3
```

```
unsigned char state = 0;
```



```
void setup() {  
  
    pinMode(SensorLED, OUTPUT);          //LED 为输出模式  
  
    pinMode(SensorINPUT, INPUT);         //倾斜开关为输入模式  
  
    //低电平变高电平的过程中，触发中断 1，调用 blink 函数  
  
    attachInterrupt(1, blink, RISING);  
  
}  
  
void loop(){  
  
    if(state!=0){                        // 如果 state 不是 0 时  
  
        state = 0;                       // state 值赋为 0  
  
        digitalWrite(SensorLED,HIGH);    // 亮灯  
  
        delay(500);                      //延时 500ms  
  
    }  
  
    else  
  
        digitalWrite(SensorLED,LOW);     // 否则，关灯  
  
}  
  
void blink(){                            //中断函数 blink()  
  
state++;                                //一旦中断触发，state 就不断自加  
  
}
```

当我们晃动板子时，LED 灯也会随之亮，一旦停止晃动，LED 灯又恢复到熄灭的状态。

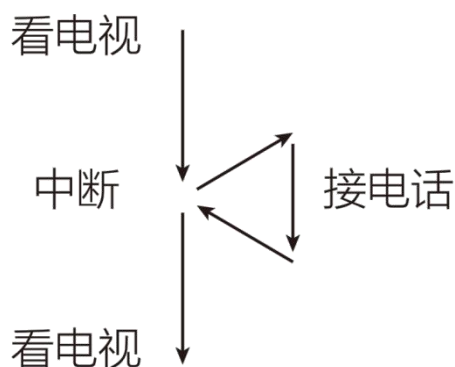
代码回顾

代码虽不长，但还是不太容易理解的。先大致说下代码的运行过程。

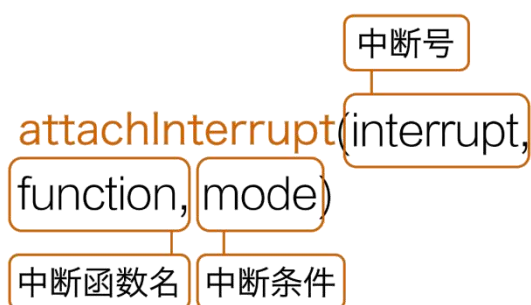
在没有任何打扰的情况下，程序在不断运行着.....，让 LED 一直处于关闭。突然，被人打扰了（也就是晃动板子），就跳到中断函数 `blink()` 中（当然进入中断也是要条件的，我们后面说）。此时，`state` 不断自加导致主函数中 `if` 函数检测到 `state` 不为 0 了，那么就让 LED 亮起了，同时又重新让 `state` 为 0，等待下一次中断。未发生中断，LED 则恢复关闭状态。

重复的知识点不再赘述，下面重点讲解中断函数 `attachInterrupt()`。

什么是中断？打个比方吧，比如你在家看电视，突然家里电话铃响了，那么你不得不停下看电视先去接电话，等接完电话后，你又可以继续看电视啦！在整个过程中接电话就是一个中断过程，电话铃响就是中断的标志，或者说中断条件。



现在知道中断是什么意思了吧，再回到 `attachInterrupt()` 函数，它是一个当外部发生中断时，才被唤醒的函数。区别于其他函数，它依附于中断引脚才发生。大多数板子都有两个外部中断引脚：数字引脚 2（中断 0）和数字引脚 3（中断 1）。中断 0 与中断 1 是中断号，在函数中需要用到。不同板子，中断号对应引脚可能不同，可以查阅 Arduino 官方编程语法手册，网址如下：
<http://arduino.cc/en/Reference/AttachInterrupt>。



`interrupt`: 中断号 0 或者 1。如果选择 0 的话，连接到数字引脚 2 上，选择 1 的话，连接到数字引脚 3 上。

`function`: 调用的中断函数名。写中断函数时，需要特别说明以下三点：

- 1、我们在写中断函数的时候，该函数不能含有参数和返回值。也就是说，要写一个无返回值的函数。
- 2、中断函数中不要使用 `delay()` 和 `millis()` 函数，因为数值不会继续变化。
- 3、中断函数中不要读取串口，串口收到的数据可能会丢失。

`mode`: 中断的条件。只有特定的以下四种情况：

- 1、LOW 当引脚为低电平时，触发中断。

2、CHANGE 当引脚电平发生改变时，触发中断。

3、RISING 当引脚由低电平变为高电平时，触发中断。

4、FALLING 当引脚由高电平变为低电平时，触发中断。

知道了 `attachInterrupt()` 函数的用法，回归到我们的代码中：

```
attachInterrupt(1, blink, RISING);
```

对应上面说明看。1，指中断号 1。所以倾斜开关接到数字引脚 3。blink 是我们下面要调用的中断函数。RISING，指引脚 3 在由低变为高的一瞬间，中断触发。

为什么要选 RISING 呢？由于硬件我们还没提到，我们就把倾斜开关想象成按键。在按键没按下时，是断开的，引脚 3 处于低的状态。一旦被按下，就和 5V 导通，变为高。这个过程是引脚由低电平变高电平的过程，所以选择 RISING 模式。

硬件回顾

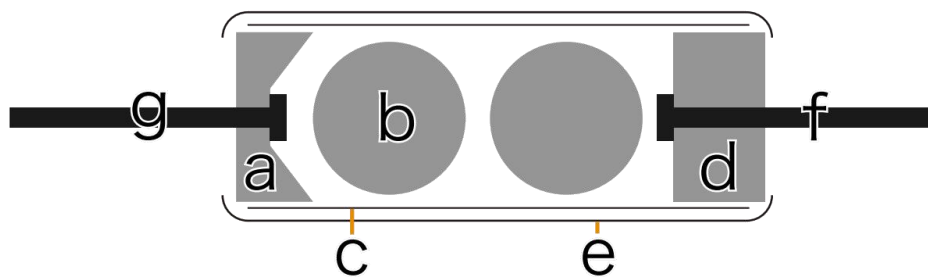
倾斜开关

倾斜开关，也叫做滚珠开关，震动开关等等。虽然叫法不同，不过原理是相同的。就是通过珠子滚动接触导针的原理来控制电路的通断。

看下结构图就明白了。倾斜开关内部两个珠子，通过珠子滚动接触导针的原理来控制电路的接通或者断开。传感器震动或者晃动时，珠子就会接触导针，从而导通。还需要注意的一点是，由于倾斜开关的内部构造，倾斜开关只有一头

是导通的，金色导针一端是导通的，银色导针一端是不导通的。这也就是为什么，往金色一端倾斜，灯会点亮，而偏向银色一端倾斜时，灯不会被点亮的原

因。



- | | | | |
|--------|---------|----------|---------|
| a. 青铜盖 | b. 青铜珠子 | c. 青铜管 | d. PC胶座 |
| e. 热缩管 | f. 青铜导针 | g. 磷铜弹簧夹 | |

图 2 倾斜开关内部结构图

项目 - 感光灯

这个项目中将介绍一个新元件——光敏电阻。从名字可以看出，这个元件是依赖光作用的。在黑暗的环境中，光敏电阻具有非常高阻值的电阻。光线越强，电阻值反而越低。通过读取这个电阻值，就可以检查光线的亮暗了。我们这里选用的是光敏二极管，光敏二极管其实就是光敏电阻中的一种，只是它还具有正负极性。

我们这次做的这个非常好玩，叫做感光灯。它能随着光线明暗而选择是否亮灯。这个光感灯非常适合用做夜晚使用的小夜灯。晚上睡觉的时候，家中灯关掉后，感光灯感觉到周围环境变暗了，就自动亮起。到了白天，天亮后，感光灯就又恢复到关闭的状态了。

元件清单

 5MM LED 5MM LED灯	x1	 Ambient Light Sensor 光敏电阻	x1
 Resistor 10K 10K电阻	x1	 Resistor 220R 220欧电阻	x1
 Jumper Cables M/M 跳线（公公头）	x5		

硬件连接

LED 灯还是和以往一样的接法。而光敏二极管是有正负极的，和 LED 一样，也是遵循长脚（-），短脚（+）的原则（在下方连接图中左侧为负极，右侧为正极）。还需注意的与光敏二极管相连的电阻是 10k，而不是 220Ω。

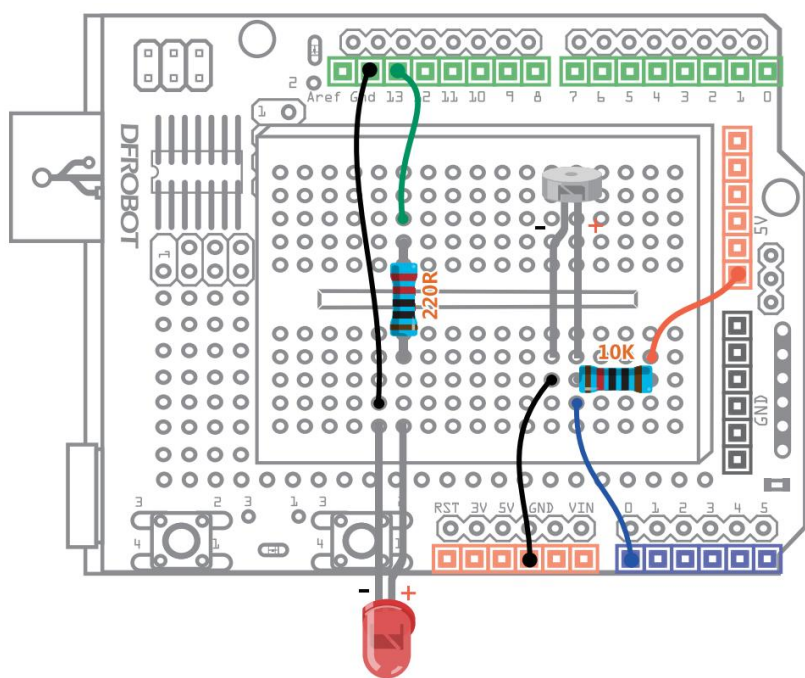


图 1 感光灯连线图

示例代码

样例代码：

//项目 - 感光灯

```
int LED = 13;
```

//设置 LED 灯为数字引脚 13

```
int val = 0;
```

//设置模拟引脚 0 读取光敏二极管的电压值

```
void setup(){  
    pinMode(LED,OUTPUT);          // LED 为输出模式  
    Serial.begin(9600);           // 串口波特率设置为 9600  
}  
  
void loop(){  
    val = analogRead(0);          // 读取电压值 0~1023  
    Serial.println(val);          // 串口查看电压值的变化  
    if(val<1000){                 // 一旦小于设定的值，LED 灯关闭  
        digitalWrite(LED,LOW);  
    }else{                        // 否则 LED 亮起  
        digitalWrite(LED,HIGH);  
    }  
    delay(10);                    // 延时 10ms  
}
```

下载完代码后，LED 灯会亮起，这时，你需要拿一个手电筒照你的光敏二极管（用手机后置摄像头的闪光灯应该也可以），这时你会发现 LED 灯神奇般的自动熄灭。但是，一旦你的手电筒移开，LED 灯又再次亮起。

代码回顾

这段代码想必你一定能看的懂了吧？我就简单说一下可能不明白的地方。我们之前在项目【温度报警器】中讲 LM35 温度传感器的时候，也用到了模拟口读取。强调了模拟量不需要设置 pinMode() 输入输出模式。这里，也是同样用模拟口用来读取光敏二极管的模拟值。

一旦有光照射，读出的模拟值就会减小，这里设定的上限值是 1000。这个值可以按你需要的亮度来选取。选取方法：先把整个装置放在你想让 LED 关闭的一个环境下，然后打开串口，查看串口显示的值，把这个值替换掉代码中的 1000。从串口读值，是调试代码一种很好的方法。

硬件回顾

光敏二极管

这里接触了一种新元件——光敏器件。这类器件都是将光信号变成电信号的特殊电子元件。元件内部有特殊的光导材料，外部用塑料或者玻璃封装。光线照射在这类光导材料上时，光敏器件的电阻值就会迅速变小。光敏元件有很多，光敏电阻，光敏二极管，光敏三极管等等。不过原理是差不多的。我们这里选用的是光敏二极管。光敏二极管其实是光敏电阻中的一种。所谓二极管，就是有正负极的，所以在连线的时候也要注意正负极。

光敏电阻在黑暗的环境中，具有非常高阻值的电阻。光线越强，电阻值反而越低。随着两端电阻值的减小，电压也就相应减小（从模拟口读到的值也就变小，模拟口 0~1023 的值对应是 0~5V 的电压值）。

那电压为什么会减小呢？那就要用到我们初中物理知识——分压原理。让我们看一个典型的分压电路，看看它是如何工作的。（图 2）

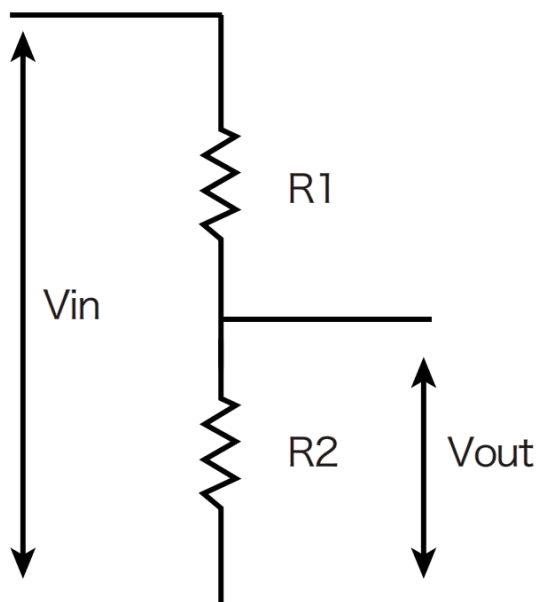


图 2 分压电路图

输入电压 V_{in} （我们这里也就是 5V），连在两个电阻上，只测量通过电阻 R_2 的电压 V_{out} ，其电压将小于输入电压。计算 R_2 两端的 V_{out} 电压公式如下图所示。（图 3）

$$V_{out} = \frac{R_2}{R_1 + R_2} \times V_{in}$$

图 3 分压公式

在我们这项目中， R_1 代表的就是 10k 电阻， R_2 代表的就是光敏二极管。本来 R_2 在黑暗中，值很大很大，所以 V_{out} 也就很大，接近 5V。一旦有光线照射的话， R_2 的值就会迅速减小，所以 V_{out} 也就随之减小了，读取的电压值就小。

通过上面这个公式可以看出，R1 选取不能太小，最好在 1k~10k 左右，否则比值变化不明显。

项目 - 舵机初动

本项目要接触到舵机。舵机是一种电机，通过反馈系统来控制其位置，从而能够精确地掌握电机的角度。大多数舵机最大旋转角度是 180° 。也有一些能转到 270° ，甚至 360° 。舵机常被用于对角度有精确要求的场合，如摄像头、智能小车的前置探测器，以及需要在特定范围内进行监测的移动平台。又或者把舵机放到玩具，让玩具动起来。还可以用多个舵机，做个小型机器人，舵机就可以作为机器人的关节部分。所以，舵机的用处很多。

Arduino 也提供了 `<Servo.h>` 库，让我们使用舵机变得更方便了。

先从简单程序入手，套件这个 9G 小舵机是 180° 的，我们就让它在 $0\sim180^{\circ}$ 之间来回转动。

元件清单



Servo
舵机

x1



Jumper Cables M/M
跳线（公公头）

x3

硬件连接

这个项目的连线很简单，只需按图 1 所示连接舵机三根线就可以了，连的时候注意线序，舵机引出三根线。一根是红色，连到+5V 上。一根棕色（有些是黑的），连到 GND。还有一根是黄色或者橘色，连到数字引脚 9。

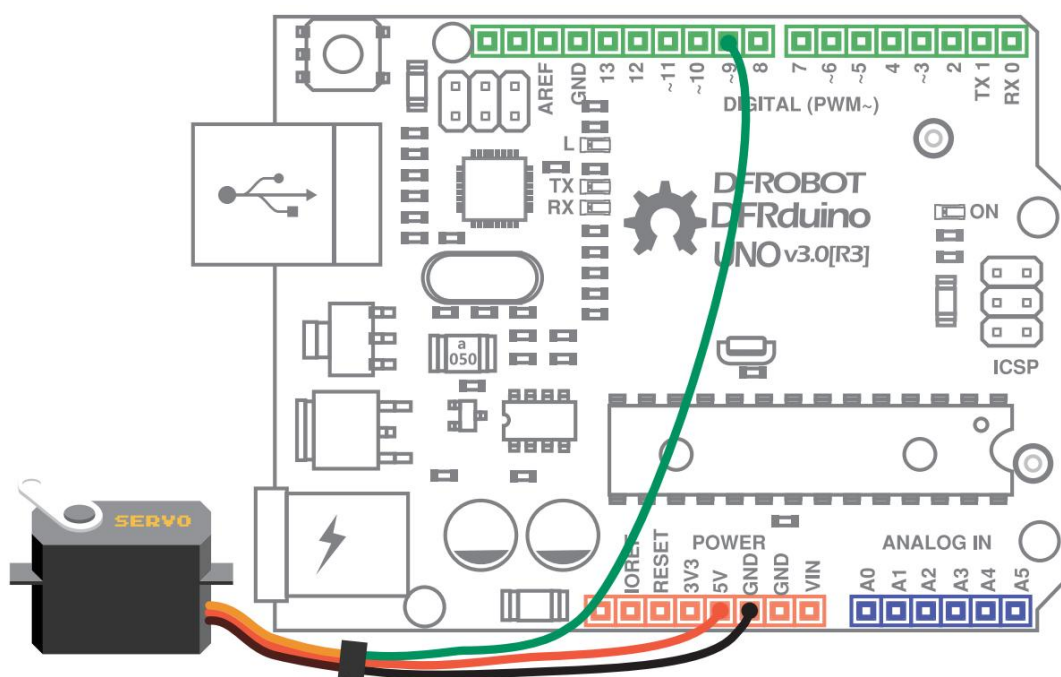


图 1 舵机初动连线图

代码示例

样例代码：

```
//项目 - 舵机初动
```

```
#include <Servo.h>           // 声明调用 Servo.h 库
```

```
Servo myservo;               // 创建一个舵机对象
```

```
int pos = 0;                  // 变量 pos 用来存储舵机位置
```

```
void setup() {  
    myservo.attach(9); // 将引脚 9 上的舵机与声明的舵机对象连接起来  
}  
  
void loop() {  
    for(pos = 0; pos < 180; pos += 1){  
        // 舵机从 0°转到 180°，每次增加 1°  
        myservo.write(pos);           // 给舵机写入角度  
        delay(15);                     // 延时 15ms 让舵机转到指定位置  
    }  
  
    for(pos = 180; pos>=1; pos-=1) {  
        // 舵机从 180°回到 0°，每次减小 1°  
        myservo.write(pos);           // 写角度到舵机  
        delay(15);                     // 延时 15ms 让舵机转到指定位置  
    }  
}
```

下载代码，下载成功后我们可以看到舵机 0~180°来回转动。

代码回顾

代码的开头首先调用了<Servo.h>库

```
#include <Servo.h>
```

这个库已经在 Arduino IDE 中了，可以打开 Arduino-1.0.5/ libraries/Servo/ Servo.h，这就是 Servo 库所在位置。

我们怎么理解库呢？和我们前面讲到的函数意义是差不多的。函数通常按一个个功能来划分的，就像一个个小的储物柜，函数名好比储物柜标签名。我们使用的时候，直接看标签就好了，方便我们使用。那库是什么呢？库则是把多个函数封装打包起来，好比大的储物柜，里面含有一个个小的储物柜。不知道这样说，你是不是能理解库和函数的关系？

同样，大储物柜也需要一个标签，这标签的学术名叫做“对象”。所以这里叫创建一个对象。就是我们接下来的这句语句：

```
Servo myservo;
```

setup()函数中有一条语句：

```
myservo.attach(9);
```

这里就开始调用 Servo 库中的函数了，和我们以前函数调用有点区别。这里，

我们需要先指明这是哪个库中的函数。所以，先指出对象名，再指出函数名。每次要用到储物柜的东西就要先指明这个标签。这样程序才知道要去哪里找东西。库函数调用格式如下：

对象名.函数名();

不要忘了中间的 “.”！myservo 是我们前面设的标签（对象），然后调用的函数是：



```
attach(pin);
```

数字引脚

attach(pin)函数有一个传递参数——pin，任意一个数字引脚（不建议使用数字 0,1）。我们这里选择数字引脚 9。

进入主函数，有两个 for 循环，第一段是从 0 开始，循环到 180，每次增加 1 度。第二个 for 循环则是从 180 开始，每次减小 1 度，一直减到 0。再回到上面那个循环中.....

for 循环中又调用了一个 Servo 库中的函数 write(pos)，变量 pos 用来存放角度值。我们可以不用管函数内部复杂的程序，只要先会使用就可以了。

```
myservo.write(pos);
```


和上面那个函数调用一样，先要指明是哪个库。该函数的功能是将舵机转动到对应的角度，传递参数就是角度，单位为°。

如果还想了解 Servo 库中还有哪些好用的函数的话，可以参看下面的网址，里面会有相关介绍的。

Servo 库：

<http://arduino.cc/en/reference/servo>

我们无需深入了解函数内部的复杂逻辑，只需学会如何正确使用即可。如果还想了解更多的话，可以借助我们的网络资源。

DF 创客社区：

www.dfrobot.com.cn

项目 - 可控舵机

在前面一个项目中，我们知道了如何让舵机动起来，这里将进一步的通过外部信号来让舵机随着输入的改变来相应改变角度，方便做一些可控的转动装置。我们这里通过一个可变电阻——电位器，来控制舵机。当然你也可以通过其他的模拟量或者数字量来控制舵机。模拟量的话，比如改造一下前面的感光灯，变成一个会动的感光灯。数字量的话，比如通过一个按钮，倾斜开关等等，一旦触发开关，就让舵机转动，可以有很多玩儿法。再给舵机加个外壳，让它更具生命力。

元件清单



10K Potentiometer
10K 电位器

x1



Servo
舵机

x1



Jumper Cables M/M
跳线（公公头）

x6

硬件连接

与项目【舵机初动】不同处在于多了一个电位器，电位器相当于一个可变阻值的电阻，两个引脚的一边分别接 5V 与 GND，而另一边只有单独一个引脚的接模拟口 0，用于做输入信号。

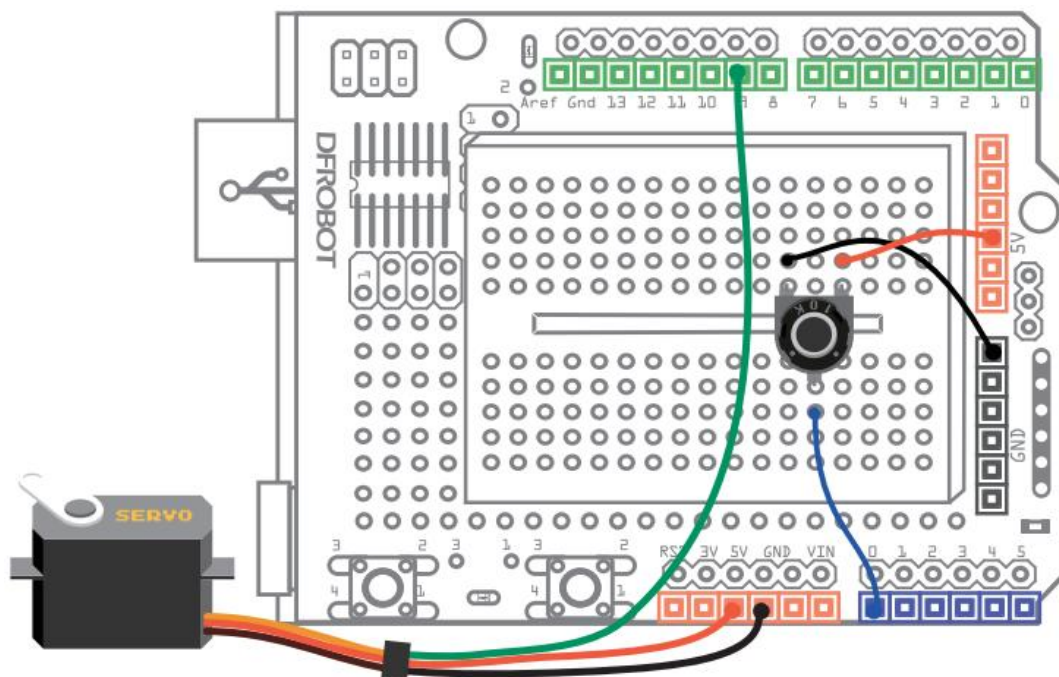


图 1 可控舵机连线图

代码示例

```
//项目 - 可控舵机
```

```
#include <Servo.h>
```

```
// 声明调用 Servo.h 库
```

```
Servo myservo;
```

```
// 创建一个舵机对象
```

```
int potpin = 0;
```

```
// 连接到模拟口 0
```

```
int val;
```

```
//变量 val 用来存储从模拟口 0 读到的值
```

```
void setup() {  
    myservo.attach(9);    // 将引脚 9 上的舵机与声明的舵机对象连接起来  
}  
  
void loop() {  
    val = analogRead(potpin);  
    //从模拟口 0 读值，并通过 val 记录  
    val = map(val, 0, 1023, 0, 179); //通过 map 函数进行数值转换  
    myservo.write(val);              // 给舵机写入角度  
    delay(15);                      // 延时 15ms 让舵机转到指定位置  
}
```

下载代码，成功后，旋转电位器，看看舵机是不是随着电位器转动。

代码回顾

代码的开始部分还是需要调用<Servo.h>库，并创建相应的对象。同时，需要一个模拟口用来读取电位器的值，我们这里用变量 potPin 代表模拟口 0。这里主要讲下 map 函数。

函数格式如下：

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

map 函数的作用是将一个数从一个范围映射到另外一个范围。也就是说，会将 fromLow 到 fromHigh 之间的值映射到 toLow 在 toHigh 之间的值。

map 函数参数含义：

value：需要映射的值

fromLow：当前范围值的下限

fromHigh：当前范围值的上限

toLow：目标范围值的下限

toHigh：目标范围值的上限

map 的神奇之处还在于，两个范围中的“下限”可以比“上限”更大或者更小，因此 map()函数可以用来翻转数值的范围，可以这么写：

```
y = map(x, 1, 50, 50, 1);
```

这个函数同样可以处理负数，请看下面这个例子：

```
y = map(x, 1, 50, 50, -100);  
val = map(val, 0, 1023, 0, 179);
```

所以，回到代码中，我们是想将模拟口读到的 0~1023 的值，转换为舵机的 0~180°。

项目 - 彩灯调光台

在项目【炫彩 LED】的时候，我们已经接触过 RGB LED 了，可以实现变色，这回儿我们需要加入互动元素进去。通过三个电位器来任意变换对应的 R、G、B，组合成任何你想要的颜色，在家做个心情灯吧，随心情任意切换。

元件清单



5MM RGB LED
5MM RGB LED灯 **x1**



Resistor 220R
220欧电阻 **x3**



10K Potentiometer
10K 电位器 **x3**



Jumper Cables M/M
跳线（公公头） **x13**

硬件连接

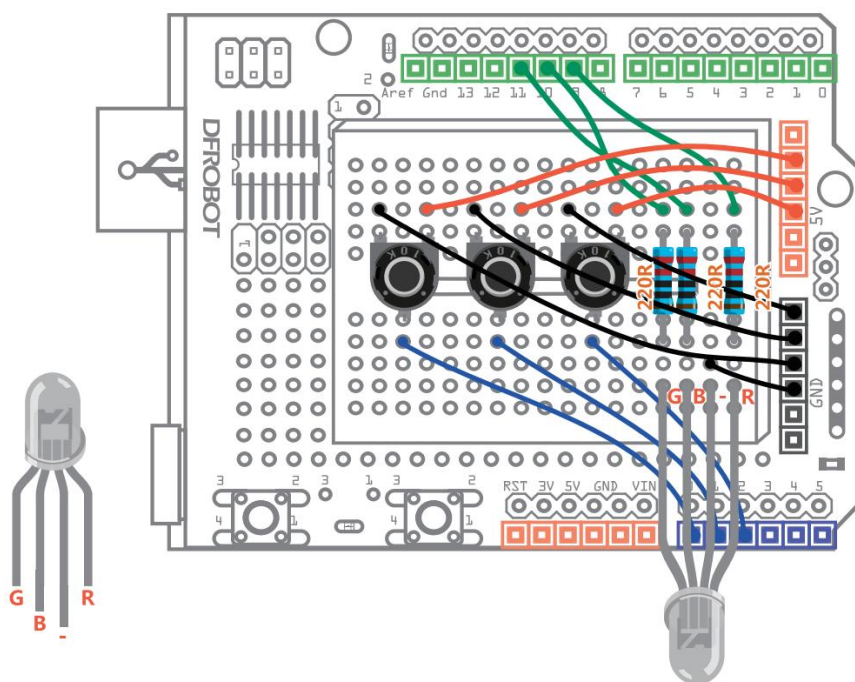


图 1 彩灯调光台连线图

代码示例

样例代码：

//项目 - 互动彩灯

```
int redPin = 9;           // R - digital 9
int greenPin = 10;        // G - digital 10
int bluePin = 11;         // B - digital 11
int potRedPin = 0;         // 电位器 1 - analog 0
int potGreenPin = 1;       // 电位器 2 - analog 1
int potBluePin = 2;        // 电位器 3 - analog 2
```

```
void setup(){

    pinMode(redPin,OUTPUT);

    pinMode(greenPin,OUTPUT);

    pinMode(bluePin,OUTPUT);

    Serial.begin(9600);           // 初始化串口

}


void loop(){

    int potRed = analogRead(potRedPin);
// potRed 存储模拟口 0 读到的值

    int potGreen = analogRead(potGreenPin);
// potGreen 存储模拟口 1 读到的值

    int potBlue = analogRead(potBluePin);
// potBlue 存储模拟口 2 读到的值


    int val1 = map(potRed,0,1023,0,255);
//通过 map 函数转换为 0~255 的值

    int val2 = map(potGreen,0,1023,0,255);

    int val3 = map(potBlue,0,1023,0,255);


    //串口依次输出 Red, Green, Blue 对应值

    Serial.print("Red:");

    Serial.print(val1);
```



```
Serial.print("Green:");  
  
Serial.print(val2);  
  
Serial.print("Blue:");  
  
Serial.println(val3);  
  
  
colorRGB(val1,val2,val3);    // 让 RGB LED 呈现对应颜色  
  
}  
  
  
//该函数用于显示颜色  
void colorRGB(int red, int green, int blue){  
    analogWrite(redPin,constrain(red,0,255));  
    analogWrite(greenPin,constrain(green,0,255));  
    analogWrite(bluePin,constrain(blue,0,255));  
}
```

上传代码，成功后，通过旋转三个电位器，看看 RGB 灯是否会随之变化。

项目 - 自制风扇

这次，我们会做一个用按钮控制小风扇。同时会接触两件新元件——继电器、直流电机。继电器，我们可以理解为是用较小的电流去控制较大电流的一种“自动开关”。在这里，继电器是用来控制电机转动的。

元件清单

 Pushbutton 按键开关	x1	 Relay 小型继电器	x1	 Resistor 220R 220欧电阻	x2
 5MM LED 5MM LED灯	x1	 130Motor 130小马达	x1	 Fan 透明风扇叶	x1
 Jumper Cables M/M 跳线（公公头）	x9				

硬件连接

按下图进行连线，按钮接线与项目【交通信号灯】类似，连接到数字 2。按钮一端连接 5V，另一端连接 GND，并用一个 220Ω 的电阻作为下拉电阻，以防引脚悬空干扰。继电器有 6 个引脚，分别标有序号。1，2 引脚为继电器的输入信号，分别接 Arduino 的数字引脚和 GND。3，4，5，6 为继电器输出的控制引脚，这里只使用 4，6 两个引脚。我们把继电器想成一个开关，开关也只要用到两个

引脚。

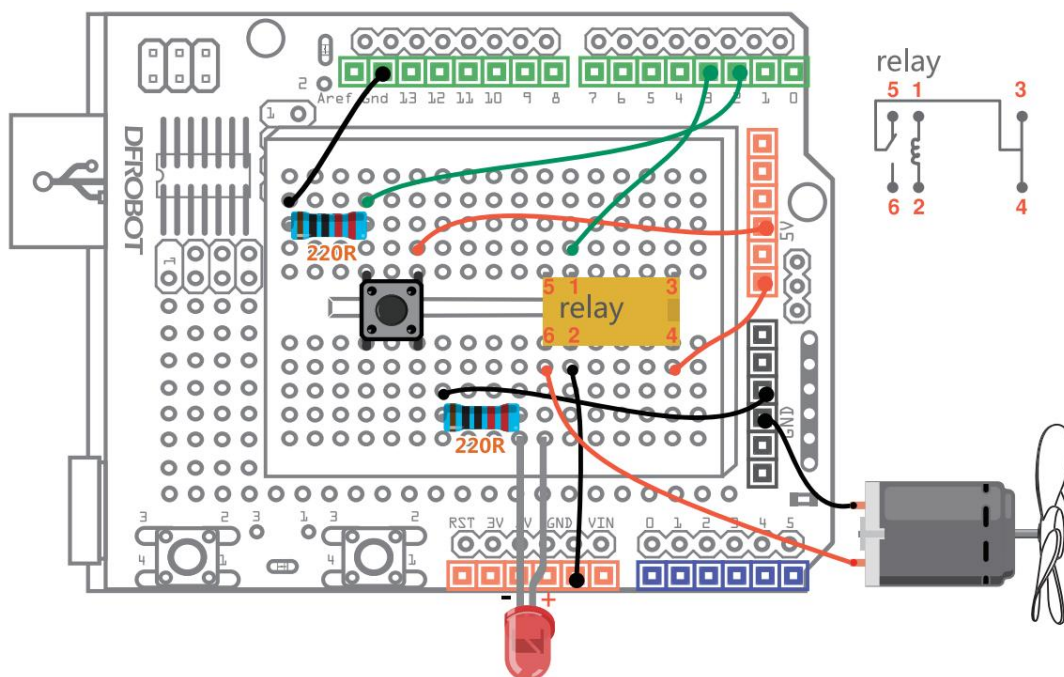


图 1 自制风扇连线图

代码示例

Sample code:

//项目 - 自制风扇

```
int buttonPin = 2;
```

// button 连接到数字 2

```
int relayPin = 3;
```

// 继电器连接到数字 3

```
int relayState = HIGH;
```

// 继电器初始状态为 HIGH

```
int buttonState;
```

// 记录 button 当前状态值

```
int lastButtonState = LOW;
```

// 记录 button 前一个状态值

```
long lastDebounceTime = 0;
```

```
long debounceDelay = 50;           // 去除抖动时间

void setup() {
    pinMode(buttonPin, INPUT);
    pinMode(relayPin, OUTPUT);

    digitalWrite(relayPin, relayState);    // 设置继电器的初始状态
}

void loop() {
    int reading = digitalRead(buttonPin);
    //reading 用来存储 buttonPin 的数据

    // 一旦检测到数据发生变化，记录当前时间
    if (reading != lastButtonState) {
        lastDebounceTime = millis();
    }

    // 等待 50ms，再进行一次判断，是否和当前 button 状态相同
    // 如果和当前状态不相同，改变 button 状态
    // 同时，如果 button 状态为高（也就是被按下），那么就改变继电器的状态
    if ((millis() - lastDebounceTime) > debounceDelay) {
        if (reading != buttonState) {
            buttonState = reading;
```

```
    if (buttonState == HIGH) {  
        relayState = !relayState;  
    }  
}  
  
digitalWrite(relayPin, relayState);  
  
// 改变 button 前一个状态值  
lastButtonState = reading;  
}
```

代码回顾

代码的大部分内容，基本应该没有什么难度了，主要说下按键去抖的代码进行说明。

去抖代码如下：

```
if (reading != lastButtonState) {  
    lastDebounceTime = millis();  
}  
  
if ((millis() - lastDebounceTime) > debounceDelay) {
```

```
    if (reading != buttonState) {  
        ...  
    }  
}
```

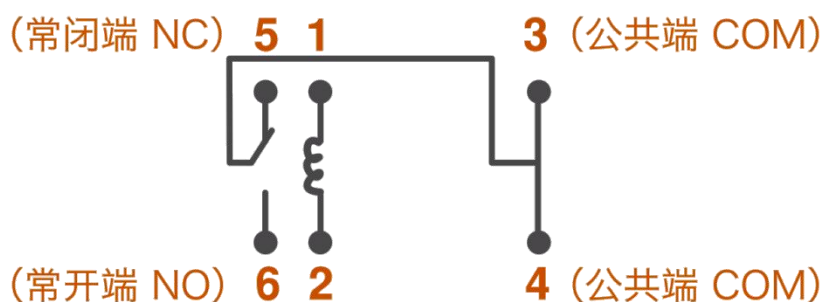
reading 有变化之后，不是立马就采取相应的行动，而是先“按兵不动”，先看看这个信号是不是“错误信号”，所以再等待一阵，（也就是通过 millis 来实现这个等待过程的），发现确实是前方发过来的正确信号，然后执行相关动作。这么做的原因是，按键在被按下时，会有个抖动的过程，而不是立马由低变高，或者由高变低。所以这个过程中，可能会产生错误信号，我们通过程序中的这种方法，来解决硬件上的这个问题。

硬件回顾

继电器

我们可以把继电器理解为一个“开关”，实际上是用比较小的电流去控制较大电流的“开关”。这里只是为了让初学者了解继电器工作原理，所以没有使用较大的电源器件，而是选用需要 5V 就能驱动的直流电机。

我们来看下继电器的内部构造：



这款继电器一共有 6 个引脚。1, 2 引脚是用来接 Arduino 数字引脚和 GND。通过数字引脚来驱动继电器。1, 2 两端为线圈两端。Arduino 给 HIGH 后, 线圈中就有电流, 线圈就会产生磁性 (就像磁铁一样), 吸合中间的触片 (能听到 “哒” 一声), 常开端 (NO) 就与公共端导通。相反, 如果 Arduino 给 LOW, 线圈中没有电流, 常闭端 (NC) 就与公共端导通。所以, 电路中我们接了 4, 6 引脚用于控制电机和 LED 的通断, (当然也可以用引脚 3, 6)。

直流电机、直流减速电机与舵机的区别

普通直流电机是我们接触比较多的电机。一般只有两个引脚, 上电就能转, 正负极反接则反向转动。如你所见, 它做着周而复始的圆周运动, 无法进行角度的控制, 不过可以通过电机驱动板, 可以对转速进行控制, 不过由于普通电机转速过快, 所以, 一般不直接用在智能小车上。

直流减速电机是在普通电机加上了减速箱, 这样便降低了转速, 使得普通电机有更广泛的使用空间, 比如可以用于智能小车上。同样也可以通过 PWM 来进行调速。

舵机也是一种电机，它使用一个反馈系统来控制电机的位置，可以用来控制角度。所以，舵机经常用来控制一些机器人手臂关节的转动。

项目 - 红外遥控灯

这节我们会接触一个新的元件——红外接收管。所谓红外接收管，也就是接收红外光的电子器件。红外接收管，看着离我们很遥远的感觉！其实不然，它就在我们身边。比如我们的电视机，空调这些家电，其实它们都需要用到红外接收管。我们都知道遥控器发射出来的都是红外光，电视机上势必要有红外接收管，才能接收到遥控器发过来的红外信号。

我们这次就用红外接收管做个遥控灯，通过遥控器的红色电源键来控制 LED 的开关。在开始遥控灯之前，我们先来个预热实验，通过串口来了解下如何使用红外接收管和遥控器。

元件清单

	5MM LED 5MM LED灯	x1		Resistor 220R 220欧电阻	x1
	IR Receiver 红外接收管	x1			
	Jumper Cables M/M 跳线（公公头）	x5			

硬件连接

红外接收模块只需要连接三根线就可以了，注意一下正负就可以了（图中表明部分）。红外接收管 Vout 输出接到数字引脚 11。在这个基础上，加一个 LED 和电阻，LED 使用的是数字引脚 10。红外接收管仍然接的是数字引脚 11（LED 灯在该项目的后半部分会用上）。

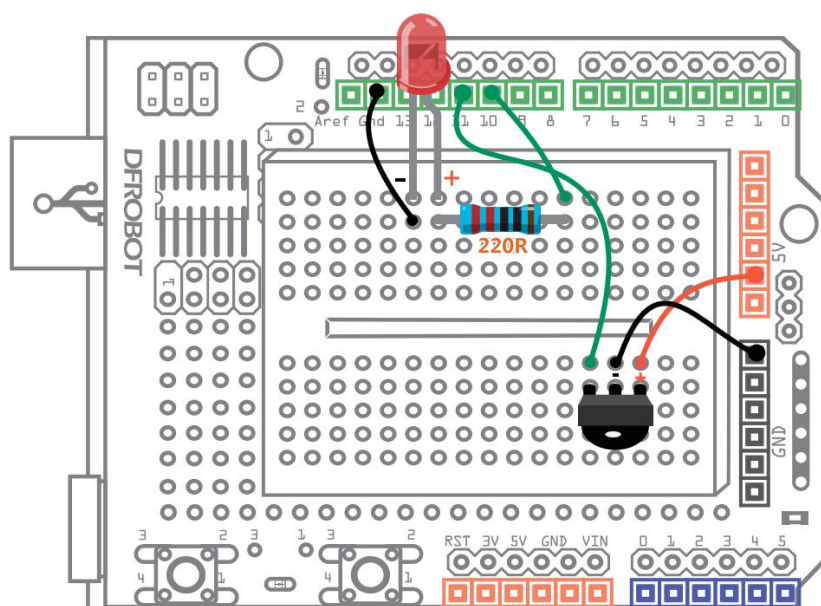
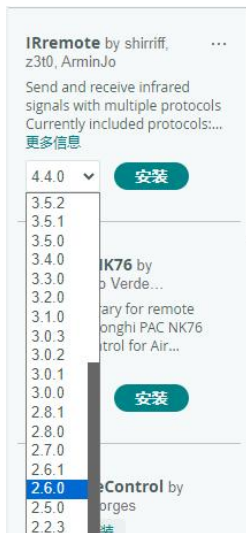


图 1 红外遥控接收器连线图

代码示例

对于遥控红外模块，我们提供现成的 IRremote 库，直接在 Arduino IDE 中安装 IRremote 库（需使用该库的 2.6.0 版本，如图 2），输入样例代码 1 运行即可。如果还是不是很明白如何加载库，可以回看一下项目【炫彩 LED】课后作业部分，对如何加载库做了详细说明。



样例代码 1:

//项目 - 红外遥控灯测试代码

```
#include <IRremote.h> //调用 IRremote.h 库
```

```
int RECV_PIN = 11; //定义 RECV_PIN 变量为 11
```

```
IRrecv irrecv(RECV_PIN); //设置 RECV_PIN（也就是 11 引脚）为红外接收端
```

```
decode_results results; //定义 results 变量为红外结果存放位置
```

```
void setup()
```

```
{
```

```
    Serial.begin(9600); //串口波特率设为 9600
```

```
    irrecv.enableIRIn(); //启动红外解码
```

```
}
```

```
void loop() {
```

```
    //是否接收到解码数据,把接收到的数据存储在变量 results 中
```

```
    if (irrecv.decode(&results)) {
```

```
//接收到的数据以 16 进制的方式在串口输出  
  
Serial.println(results.value, HEX);  
  
irrecv.resume(); // 继续等待接收下一组信号  
  
}  
  
}
```

下载完成后，打开 Arduino IDE 的串口监视器（Serial Monitor），设置波特率为 9600，与代码中 Serial.begin(9600)相匹配。



设置完后，用 Mini 遥控器的按钮对着红外接收管的方向，任意按个按钮，我们都能在串口监视器上看到相对应的代码。如下图所示，按数字“0”，接收到对应 16 进制的代码是“FD30CF”。每个按钮都有一个特定的 16 进制的代码。



如果长按一个键不放就会出现“FFFFFFFF”。



在串口中，正确接收的话，应该收到以 FD-开头的六位数。如果遥控器没有对准红外接收管的话，可能会接收到错误的代码。如下图所示：



上面这段代码我们没有像以前一样一步一步做详细说明，原因就是由于红外解码较为复杂，所幸的是，高手把这些难的工作已经做好了，提供给我们这个 IRremote 库，我们只需要会用就可以了，先不需要弄明白函数内部如何工作的。要用的时候，把代码原样搬过来就好了。照猫画虎，先用起来再说~ 预热完之后，我们言归正传，开始制作遥控灯。

这里不建议一步一步输入代码，可以在原有的代码上进行修改，观察下相对样例代码 1 增加了哪些内容。

样例代码 2:

```
#include <IRremote.h>           //调用 IRremote.h 库

int RECV_PIN = 11;              //定义 RECV_PIN 变量为 11

int ledPin = 10;                // LED - digital 10

boolean ledState = LOW;         // ledstate 用来存储 LED 的状态

IRrecv irrecv(RECV_PIN);

//设置 RECV_PIN（也就是 11 引脚）为红外接收端

decode_results results;         //定义 results 变量为红外结果存放位置
```

```
void setup(){

    Serial.begin(9600);           //串口波特率设为 9600

    irrecv.enableIRIn();          //启动红外解码

    pinMode(ledPin,OUTPUT);       // 设置 LED 为输出状态
}

void loop() {

    //是否接收到解码数据,把接收到的数据存储在变量 results 中

    if (irrecv.decode(&results)) {

        //接收到的数据以 16 进制的方式在串口输出

        Serial.println(results.value, HEX);

        //一旦接收到电源键的代码,LED 翻转状态,HIGH 变 LOW,或者 LOW 变 HIGH

        if(results.value == 0xFD00FF){

            ledState = !ledState;           //取反

            digitalWrite(ledPin,ledState);

            //改变 LED 相应状态

        }

        irrecv.resume(); // 继续等待接收下一组信号

    }

}
```

代码回顾

程序一开始还是对红外接收管的一些常规定义，按原样搬过来就可以了。

```
#include <IRremote.h>           //调用 IRremote.h 库

int RECV_PIN = 11;              //定义 RECV_PIN 变量为 11

int ledPin = 10;                // LED - digital 10

boolean ledState = LOW;         // ledstate 用来存储 LED 的状态

IRrecv irrecv(RECV_PIN);

//设置 RECV_PIN（也就是 11 引脚）为红外接收端

decode_results results;         //定义 results 变量为红外结果存放位置
```

在这里，我们多定义了一个变量 `ledState`，通过名字应该就可以看出来含义了，用来存储 LED 的状态的，由于 LED 状态就两种（1 或者 0），所以我们使用 `boolean` 变量类型，（可回看项目三中，表 3-1 列举出的数据类型）。

`setup()`函数中，对使用串口，启动红外解码，数字引脚模式进行设置。

到了主函数 `loop()`，一开始还是先判断是否接收到红外码，并把接收到的数据存储在变量 `results` 中。

```
if (irrecv.decode(&results))
```

一旦接收到数据后，程序就要做两件事。第一件事，判断是否接收到了电源键的红外码。

```
if(results.value == 0xFD00FF)
```

第二件事，就是让 LED 改变状态。

```
ledState = !ledState;           //取反  
digitalWrite(ledPin,ledState);  //改变 LED 相应状态
```

这里可能对 “!” 比较陌生，“!” 是一个逻辑非的符号，“取反”的意思。我们知道 “!=” 代表的是不等于的意思，也就是相反。这里可以类推为，!ledState 是 ledState 相反的一个状态。“!” 只能用于只有两种状态的变量中，也就是 boolean 型变量。

最后，继续等待下一组信号。

```
irrecv.resume();
```

课后练习

- 1、通过这个遥控项目，再结合项目【自制风扇】的风扇，能不能再给遥控器增加一个功能，既可控灯，还可控风扇。
- 2、DIY 一个你的遥控作品吧！比如简单的会动的小人，结合我们前面的舵机，通过遥控器上不同的按键，让舵机转动不同的角度，感觉随你的控制转动，发

挥你的想象做出更多 Arduino 作品吧!

项目 - 红外遥控数码管

数码管，常见的用来显示数字的，像计算器屏幕上显示的数字就是一种数码管。在本项目之前我们先来了解一下数码管是如何工作的。数码管，其实也算是 LED 中的一种。数码管的每一段，都是一个独立的 LED，通过数字引脚来控制相应段的亮灭就能达到显示数字的效果。现在，让我们通过完成项目的方式来感受一下数码管的神奇之处吧！

元件清单

 **8-Segment LED**
八段数码管 **x1**

 **Resistor 220R**
220欧电阻 **x8**

 **IR Receiver**
红外接收管 **x1**

 **Jumper Cables M/M**
跳线（公公头） **x13**

硬件连接

按下图连线图连接，注意数码管各段所对应的引脚。下边引脚说明图上为什么画这么几个箭头呢？个人觉得，这样看起来更方便。可以给你作为参考。我们从上面一排看，红色箭头的方向，从右往左， $b \rightarrow a \rightarrow f \rightarrow g$ 的顺序正好对

应，下面红色箭头逆时针顺序 $b \rightarrow a \rightarrow f \rightarrow g$ 。蓝色箭头也是表达的同样的意思。

我还特意在连线图上，对数码管所连接的引脚做了标示。这样就能更清楚的知道哪个引脚控制哪一段了。这 8 个电阻同样是起限流的作用。红外接收模块连线如图所示，如果有不清楚的地方请参考项目【红外遥控灯】。

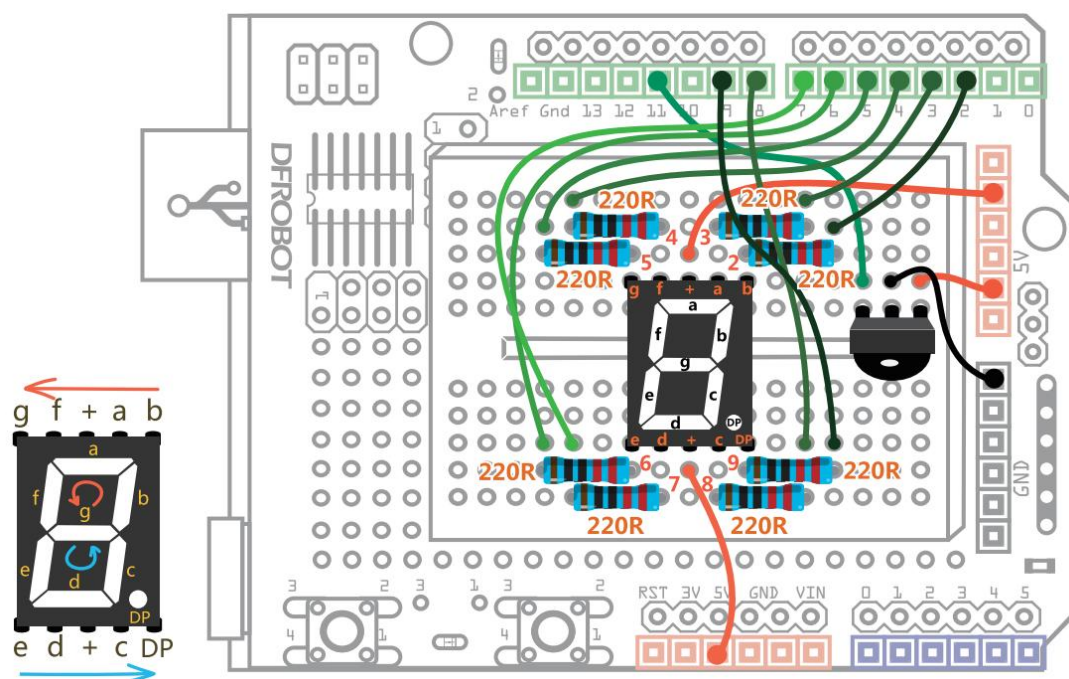


图 1 红外遥控数码管连线图

示例代码_1

样例代码 1:

//项目 - 红外遥控数码管_1

```
void setup(){
```

```
  for(int pin = 2 ; pin <= 9 ; pin++){
```

```
    // 设置数字引脚 2~9 为输出模式
```

```
    pinMode(pin, OUTPUT);

    digitalWrite(pin, HIGH);

}

}

void loop() {

    // 显示数字 0

    int n0[8]={0,0,0,1,0,0,0,1};

    for(int pin = 2; pin <= 9 ; pin++){

        // 数字引脚 2~9 依次按数组 n0[8]中的数据显示

        digitalWrite(pin,n0[pin-2]);

    }

    delay(500);

    // 显示数字 1

    int n1[8]={0,1,1,1,1,1,0,1};

    for(int pin = 2; pin <= 9 ; pin++){

        // 数字引脚 2~9 依次按数组 n1[8]中的数据显示

        digitalWrite(pin,n1[pin-2]);

    }

    delay(500);

    // 显示数字 2
```

```
int n2[8]={0,0,1,0,0,0,1,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n2[8]中的数据显示

    digitalWrite(pin,n2[pin-2]);

}

delay(500);

// 显示数字 3

int n3[8]={0,0,1,0,1,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n3[8]中的数据显示

    digitalWrite(pin,n3[pin-2]);

}

delay(500);


// 显示数字 4

int n4[8]={0,1,0,0,1,1,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n4[8]中的数据显示

    digitalWrite(pin,n4[pin-2]);

}

delay(500);


// 显示数字 5
```

```
int n5[8]={1,0,0,0,1,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n5[8]中的数据显示

    digitalWrite(pin,n5[pin-2]);

}

delay(500);


// 显示数字 6

int n6[8]={1,0,0,0,0,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n6[8]中的数据显示

    digitalWrite(pin,n6[pin-2]);

}

delay(500);


// 显示数字 7

int n7[8]={0,0,1,1,1,1,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n7[8]中的数据显示

    digitalWrite(pin,n7[pin-2]);

}

delay(500);
```

```
// 显示数字 8

int n8[8]={0,0,0,0,0,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){
    // 数字引脚 2~9 依次按数组 n8[8]中的数据显示

    digitalWrite(pin,n8[pin-2]);
}

delay(500);

// 显示数字 9

int n9[8]={0,0,0,0,1,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){
    // 数字引脚 2~9 依次按数组 n9[9]中的数据显示

    digitalWrite(pin,n9[pin-2]);
}

delay(500);
}
```

完成下载后，数码管就会循环显示 0~9 的数字。由于要看懂代码的话，首先需要了解数码管的构造，所以我们这回先说硬件部分。

硬件回顾

数码管其实就是一个前面介绍的 LED 的组合体，这个组合体包含 8 个 LED，所

以也称之为八段数码管。说白了就八个灯。哪八段？不用多说了吧！a 到 g 以及小数点 DP。其实用法和前面说的 LED 也是一样的，每段都是一个发光二极管，分别用 8 个数字口来控制它们的亮灭，通过不同段的显示，就能组成 0~9 的数字。比如，我们让 b, a, f, e, d, c 亮起的话，就能显示一个数字“0”了。

下面图 2 是个引脚说明图，不陌生了吧！在前面硬件连接的时候，已经看到过一次了。

这里, b → a → f → g → e → d → c → DP 分别连接到 Arduino 数字引脚 2~9。

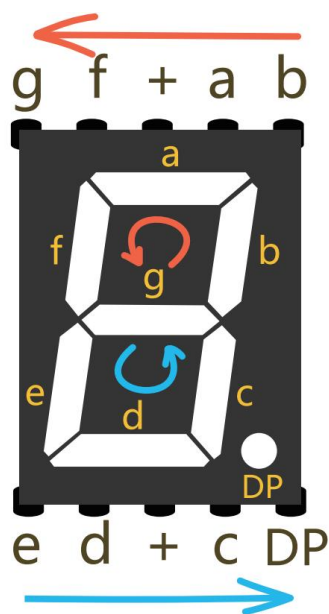


图 2 引脚连接说明图

数码管一共有 10 个引脚。a~DP 这 8 个引脚接到数字口，那还有两个引脚呢？另外两个引脚是公共端，LED 只有一端是不能被点亮的。我们在项目【炫彩 LED】中讲到过共阴共阳的问题，数码管也存在共阴共阳问题。所谓共阳就是公共端接+5V，共阴则是公共端接 GND。

共阴共阳只是在代码上要稍作修改。我们这里选用的是共阳数码管。现在你对

硬件有所了解了，我们来看看软件部分。

数码管的共阴共阳在使用上有什么区别？

共阳数码管，它们公共端接 5V，那在代码中，控制另一端的数字引脚为 LOW，这样才能让数码管点亮。

如果是共阴数码管，公共端接 GND，在代码中，控制另一端数字引脚为 HIGH，才让数码管点亮。

代码回顾_1

硬件部分我们已经说过，数码管需要接到 8 个数字引脚，所以在一开始，需要定义 8 个数字引脚作为输出。这次我们用一个 for 循环来完成这 8 个数字引脚的设置。数码管 b, a, f, g, e, d, c, DP 分别和 Arduino 数字引脚 2~9 对应。

```
for(int pin = 2 ; pin <= 9 ; pin++){ // 设置数字引脚 2~9 为输出模式  
    pinMode(pin, OUTPUT);  
    digitalWrite(pin, HIGH);  
}
```

从引脚 2 开始，一直循环到引脚 9，都设为 OUTPUT 模式，初始化为 HIGH。前面说过，共阳的话，设置 HIGH，不被点亮，所以开始先不点亮数码管。（当然，你一个引脚分开设置输出模式也是不会错的，只是会让代码显得很冗长）

好了，到了主函数，要分别显示 0~9 的数字。是不是觉得代码大部分都是相似的。所以，我们只要看明白如何显示数字 0，那整段代码就都迎刃而解了。

```
int n0[8]={0,0,0,1,0,0,0,1};
```

这里我们要引入一个数组的概念。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。在我们的程序中，声明了一个 int 型的数组。并取名为 n0。之后用 8 个数值来初始化这个数组。那如何获得数组中的元素呢？你只需要简单的指出这个元素的索引号。数组是从 0 开始索引的，这意味着数组中的第一个元素的索引号为 0 而不是 1，因此数组中的 8 个元素的索引号是 0~7。在这里元素 4，对应索引号为 3 (n0[3])，值为 1。元素 8 (索引号 7, n0[7]) 的值为 1。

声明中 n0[8]的方括号中的 8 代表有 8 个元素。

定义完数组后，进入又一个 for 循环。

```
for(int pin = 2; pin <= 9 ; pin++){  
    // 数字引脚 2~9 依次按数组 n0[8]中的数据显示  
    digitalWrite(pin,n0[pin-2]);  
}
```

这个for循环是给 2~9 引脚写入状态值,也就是 HIGH 还是 LOW,digitalWrite()

函数中写入 HIGH 的另一种形式就是写入 “1”，LOW 则可以写为 “0”。我们通过数组索引的方式给 2~9 引脚赋值。

比如当 pin=2，代入 n0[pin-2]中，对应为 n0[0]，n0[0]意思是获得数组的第一个元素，为 0。完成了引脚 2 置低（LOW）。我们前面说了，共阳的数码管，置低（LOW）的话，是被点亮，所以，b 端被点亮了。循环到 pin=3，a 段被点亮。循环到 pin=4，f 段被点亮，依次类推.....。

整个循环过程如下：

pin=2 → n0[0] =0 → digitalWrite(2,0) → b 段点亮

pin=3 → n0[1] =0 → digitalWrite(3,0) → a 段点亮

pin=4 → n0[2] =0 → digitalWrite(4,0) → f 段点亮

pin=5 → n0[3] =1 → digitalWrite(5,1) → g 段不点亮

pin=6 → n0[4] =0 → digitalWrite(6,0) → e 段点亮

pin=7 → n0[5] =0 → digitalWrite(7,0) → d 段点亮

pin=8 → n0[6] =0 → digitalWrite(8,0) → c 段点亮

pin=9 → n0[7] =1 → digitalWrite(9,1) → DP 段不点亮

这样就完成了显示数字 “0” 了。同样用数组的方法显示数字 1~9。自己动手画一下，哪几段亮，哪几段不亮就一目了然了。

如果样例代码 1 弄明白后，我们要教大家一种更简单的方法，继续看下方的样例代码 2。

示例代码_2

我们这里要教大家实现这个数码管 0~9 循环的代码另一种写法，上面我们说到了数组，通过创建 10 个数组用于显示 0~9。这里同样也还是用数组写，区别在于代码 1 中其实准确的说应该叫一维数组，我们这里用到二维数据。这样一来，可以让代码看起来更简洁。动手输一下下面这段代码吧，看看是不是同样的效果！

样例代码 2:

```
//项目 - 红外遥控数码管_2

int number[10][8] =
{
    {0,0,0,1,0,0,0,1},    //显示 0
    {0,1,1,1,1,1,0,1},    //显示 1
    {0,0,1,0,0,0,1,1},    //显示 2
    {0,0,1,0,1,0,0,1},    //显示 3
    {0,1,0,0,1,1,0,1},    //显示 4
    {1,0,0,0,1,0,0,1},    //显示 5
    {1,0,0,0,0,0,0,1},    //显示 6
    {0,0,1,1,1,1,0,1},    //显示 7
    {0,0,0,0,0,0,0,1},    //显示 8
    {0,0,0,0,1,0,0,1}     //显示 9
};
```

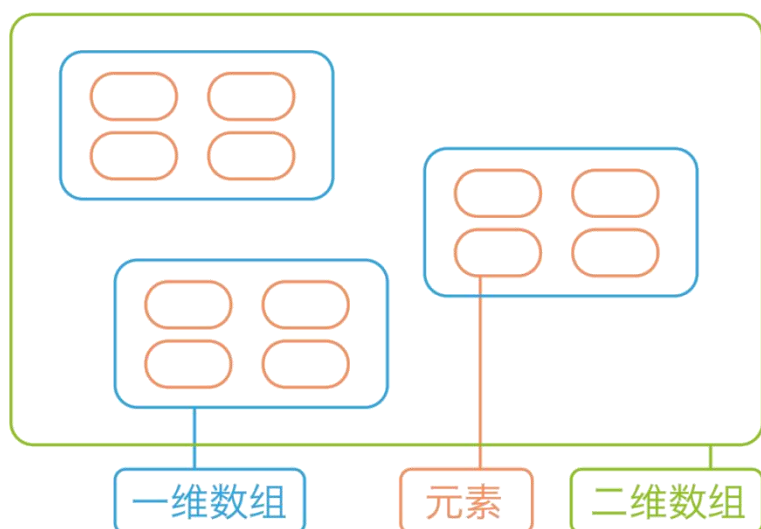
```
void numberShow(int i){           //该函数用来显示数字
    for(int pin = 2; pin <= 9 ; pin++){
        digitalWrite(pin, number[i][pin - 2]);
    }
}

void setup(){
    for(int pin = 2 ; pin <= 9 ; pin++){ // 设置数字引脚 2~9 为输出模式
        pinMode(pin, OUTPUT);
        digitalWrite(pin, HIGH);
    }
}

void loop() {
    for(int j = 0; j <= 9 ; j++){
        numberShow(j);           //调用 numberShow()函数，显示 0~9
        delay(500);
    }
}
```

代码回顾_2

对比一下样例代码 1，能发现明显的区别在哪里了吗？代码 1 中，我们创建了 10 个一维数组，代码 2 只需要创建一个二维数组就全部搞定了。不要被什么一维、二维数组的名字给吓唬到，其实用法一样的。



通过上面这个图，元素、一维数组、二维数组之间的关系就一目了然了。一维数组由元素组成，而二维数组则是由一个个一维数组组成的，关系就是那么简单。

样例代码 1 中，分别用 10 个数组，每个数组有 8 个元素，每个元素依次对应到数码管 b~DP 引脚状态值，这样就能在数码管上反映为 0~9 的数字显示。

而我们现在则是把前面散开的 10 个一维数组整合到一起，变为一个二维数组。同样通过索引的方式来找到这些元素。但还是不要忘了索引号也是从 0 开始的！前面的方括号写入的只是元素个数。

看一下代码：

```
int number[10][8] =
{
    {0,0,0,1,0,0,0,1},    //显示 0
    {0,1,1,1,1,1,0,1},    //显示 1
    {0,0,1,0,0,0,1,1},    //显示 2
    {0,0,1,0,1,0,0,1},    //显示 3
    {0,1,0,0,1,1,0,1},    //显示 4
    {1,0,0,0,1,0,0,1},    //显示 5
    {1,0,0,0,0,0,0,1},    //显示 6
    {0,0,1,1,1,1,0,1},    //显示 7
    {0,0,0,0,0,0,0,1},    //显示 8
    {0,0,0,0,1,0,0,1}    //显示 9
};
```

这就是一个二维数组。索引号从 0 开始，如果让你找 `number[0][0]`，能找到是哪个数吗？就是二维数组中第 1 行的第 1 个数，为 0。`number[9][7]`也就是第 10 行的第 8 个数，为 1。

```
void numberShow(int i){          //该函数用来显示数字
    for(int pin = 2; pin <= 9 ; pin++){
        digitalWrite(pin, number[i][pin - 2]);
    }
}
```

```
}

void loop() {
    for(int j = 0; j <= 9 ; j++){
        numberShow(j);    //调用 numberShow()函数，显示 0~9
        delay(500);
    }
}
```

上面这两段代码我们整合在一起看，loop()主函数中，for 循环让变量 j 在 0~9 循环，j 每赋一次值，numberShow()函数就要运行一次。

numberShow()函数整个运行过程如下：程序一开始 j=0，numberShow(j)为 numberShow(0)，跳回到上面的 numberShow()函数，i 现在的值就为 0 了，pin 初始值为 2，所以 digitalWrite()现在值为 digitalWrite(2, number[0][0])，回到数组 number[10][8]中找到 number[0][0]对应的值，为 0。此时，digitalWrite(2,0)，代表引脚 2 被置 LOW，引脚 2 对应的 b 段点亮（共阳数码管置 LOW 才被点亮）。之后再循环 pin=3，pin=4，.....，一直到 pin=9 整个 for 循环才结束，也代表数组的第一行的 8 个元素全被运行了一遍，最终显示一个数字 “0”。

回顾一下样例代码 1 是如何显示一个数字 “0” 的：


```
// 显示数字 0

int n0[8]={0,0,0,1,0,0,0,1};

for(int pin = 2; pin <= 9 ; pin++){

// 数字引脚 2~9 依次按数组 n0[8]中的数据显示

    digitalWrite(pin,n0[pin-2]);

}
```

原理是一样的，通过给引脚 2~9 循环赋值，控制数码管 b~DP 段亮灭，就能显示出一个我们想要的数字。numberShow(0)循环完后，再次回到 loop()中的 for 函数：

j=1 → numberShow(1) → i =1 → number[1][pin-2] → 显示数字 1
j=2 → numberShow(2) → i=2 → number[2][pin-2] → 显示数字 2
j=3 → numberShow(3) → i=3 → number[3][pin-2] → 显示数字 3
.....
j=9 → numberShow(9) → i=9 → number[9][pin-2] → 显示数字 9

好了，这就是整段代码的分析过程，好好体会一下一维数组和二维数组的区别，以及整段代码如何巧妙运行的。了解了数码管和红外接收管各自的工作原理后，我们需要把这两者结合起来，看看红外接收管和数码管结合能迸发出怎样的火花呢？想到了吗？遥控数码管！Arduino 控制器把红外接收管从 Mini 遥控器那儿接到的信号，经处理传达给数码管。让 Mini 遥控器上 0~9 对应数码管上显示 0~9，除此之外，还有递减，递增的功能。

示例代码_3

样例代码 3:

```
//项目 - 红外遥控数码管_3

#include <IRremote.h>          //调用 IRremote.h 库

int RECV_PIN = 11;             //定义 RECV_PIN 变量为 11

IRrecv irrecv(RECV_PIN);

//设置 RECV_PIN（也就是 11 引脚）为红外接收端

decode_results results;        //定义 results 变量为红外结果存放位置

int currentNumber = 0;         //该变量用于存放当前数字

long codes[12]=                //该数组用来存放红外遥控器发出的红外码
{
    0xFD30CF,0xFD08F7,          // 0 ,1
    0xFD8877,0xFD48B7,          // 2 ,3
    0xFD28D7,0xFDA857,          // 4 ,5
    0xFD6897,0xFD18E7,          // 6 ,7
    0xFD9867,0xFD58A7,          // 8 ,9
    0xFD20DF,0xFD609F,          // 前进 ,后退
};

int number[10][8] =            //该数组用来存放数码管显示的数字
{
```

```
{0,0,0,1,0,0,0,1},//0
{0,1,1,1,1,1,0,1},//1
{0,0,1,0,0,0,1,1},//2
{0,0,1,0,1,0,0,1},//3
{0,1,0,0,1,1,0,1},//4
{1,0,0,0,1,0,0,1},//5
{1,0,0,0,0,0,0,1},//6
{0,0,1,1,1,1,0,1},//7
{0,0,0,0,0,0,0,1},//8
{0,0,0,0,1,0,0,1} //9
};

void numberShow(int i) {           //该函数用来让数码管显示数字
    for(int pin = 2; pin <= 9 ; pin++){
        digitalWrite(pin, number[i][pin - 2]);
    }
}

void setup(){
    Serial.begin(9600);             //设置波特率为 9600
    irrecv.enableIRIn();            //启动红外解码

    for(int pin = 2 ; pin <= 9 ; pin++){
```

```
//设置数字引脚 2~9 为输出模式

    pinMode(pin, OUTPUT);

    digitalWrite(pin, HIGH);

}

}

void loop() {

    //判断是否接收到解码数据,把接收到的数据存储在变量 results 中

    if (irrecv.decode(&results)) {

        for(int i = 0; i <= 11; i++){

            //判断是否接收到 0~9 按键的红外码

            if(results.value == codes[i]&& i <= 9){

                numberShow(i); //在数码管上对应显示 0~9

                currentNumber = i;

                //把当前显示的值赋给变量 currentNumber

                Serial.println(i);

                break;

            }

            //判断是否接收到递减的红外码, 并且当前值不为 0

            else if(results.value == codes[10]&& currentNumber != 0){

                currentNumber--; //当前值递减

                numberShow(currentNumber); //数码管显示递减后的值

            }

        }

    }

}
```

```
Serial.println(currentNumber); //串口输出递减后的值

break;

}

//判断是否接收到递增的红外码，并且当前值不为9

else if(results.value == codes[11] && currentNumber != 9){

    currentNumber++; //当前值递增

    numberShow(currentNumber); //数码管显示递增后的值

    Serial.println(currentNumber); //串口输出递增后的值

    break;

}

}

Serial.println(results.value, HEX); //串口监视器查看红外码

irrecv.resume(); //等待接收下一个信号

}

}
```

上传完代码后，尝试按下图 3 指出部分的按键，看看是数码管是个怎样的变化。



图 3 遥控器按键说明

代码回顾_3

程序一开始还是对红外接收管的一些常规定义，按项目【红外遥控灯】搬过来即可（注意：IRremote 库选择 2.6.0 版本）。

```
#include <IRremote.h>           //调用 IRremote.h 库

int RECV_PIN = 11;               //定义 RECV_PIN 变量为 11

IRrecv irrecv(RECV_PIN);

//设置 RECV_PIN（也就是 11 引脚）为红外接收端

decode_results results;          //定义 results 变量为红外结果存放位置

int currentNumber = 0;           //该变量用于存放当前数字
```

在这里，我们多定义了一个变量 `currentNumber`，通过名字应该就可以看出来含义了吧。这个变量的作用是，用来存储当前的数字，数字递增递减时能找到

对应的参照点。

同样用数组的方式来存放这些红外码, 0x-表示是 16 进制。long 是变量的类型, 如果你还想用遥控器上的其他按键来控制做一些其他事情的话, 把红外码替换掉就好了。

```
long codes[12]=                                //该数组用来存放红外遥控器发出的红外码
{
    0xFD30CF,0xFD08F7,                          // 0 ,1
    0xFD8877,0xFD48B7,                          // 2 ,3
    0xFD28D7,0xFDA857,                          // 4 ,5
    0xFD6897,0xFD18E7,                          // 6 ,7
    0xFD9867,0xFD58A7,                          // 8 ,9
    0xFD20DF,0xFD609F,                          // 前进 ,后退
};
```

紧接着是, 一个二维数组 number[10][8]的定义。我们在代码回顾_2 已有说明了, 通过调用数组的元素, 把这些元素的值依次赋给数码管显示段的控制引脚, 并在 numberShow()函数中得以实现数码管数字显示。setup()函数中, 仍然是波特率设置, 启动红外解码, 数字引脚模式设置等常规设置。

到了主函数 loop(), 一开始还是先判断是否接收到红外码, 并把接收到的数据存储在变量 results 中。

```
if (irrecv.decode(&results))
```

一旦接收到数据后, 程序就要做两件事。第一件事, 判断是哪个红外码, 也就

能对应找到是哪个按键按下的。第二件事，找到对应按键后，让数码管干什么事？让我们接着看看程序是如何完成这两件事的。

第一件事：

会有三种情况需要判断，第一种情况，按遥控器 0~9 时，数码管显示数字 0~9。

第二种情况，每按下后退键，数字在原有基础上递减一位，直到减到 0 为止。

第三种情况，每按下前进键，数字在原有基础上增一位，直到增到 9 为止。

对这三种情况进行判断，这里呢，同样用到了 if 语句，与以往有所不同的，我们选择用 if...else if。if...else 和 if...else if 的区别在哪儿？区别在于 else if 后面需要接判断表达式，else 不需要判断表达式。然而，不管是 else 还是 else if 都是依附于 if 语句存在的，不能独立使用。

回到代码中，这就是以下三种情况：

```
if(results.value == codes[i]&& i <= 9)

if(results.value == codes[10]&& currentNumber != 0)

if(results.value == codes[11]&& currentNumber != 9)
```

第一个 if 判断的是第一种情况，显示数字 0~9。判断条件就是接收到的数据 results.value 的值是不是数组中 codes[0]~codes[9]的红外码。

第二个 if 判断的是第二种情况，是否接到后退键指令，也就是 code[10]=0xFD20DF，并且当前显示数字不为 0。

第三个 if 判断的是第三种情况，是否接到前进键指令，也就是 code[11]=

0xFDA857，并且当前显示数字不为 9。

还有一个问题——如何找到数组中的元素呢？所以，就需要在 if 判断前设置一个 for 循环，让变量 i 一直在 0~11 之间循环。

第一件事判断是哪个红外码完成后，开始执行第二件事。

就是每个 if 语句后，都有相应的执行代码。就不一一详细说了。

整段代码就讲完了，这段代码应该是所有项目中最复杂的代码，可能一开始不能完全看明白，不过没关系，实践出真知，通过一遍遍不断的尝试，相信你总有一天能明白的。

课后练习

学习了如何使用遥控功能，DIY 一个你的遥控作品吧！比如简单的会动的小人，结合我们前面的舵机，通过遥控器上不同的按键，让舵机转动不同的角度，发挥你的想象做出更多 Arduino 作品吧！

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。

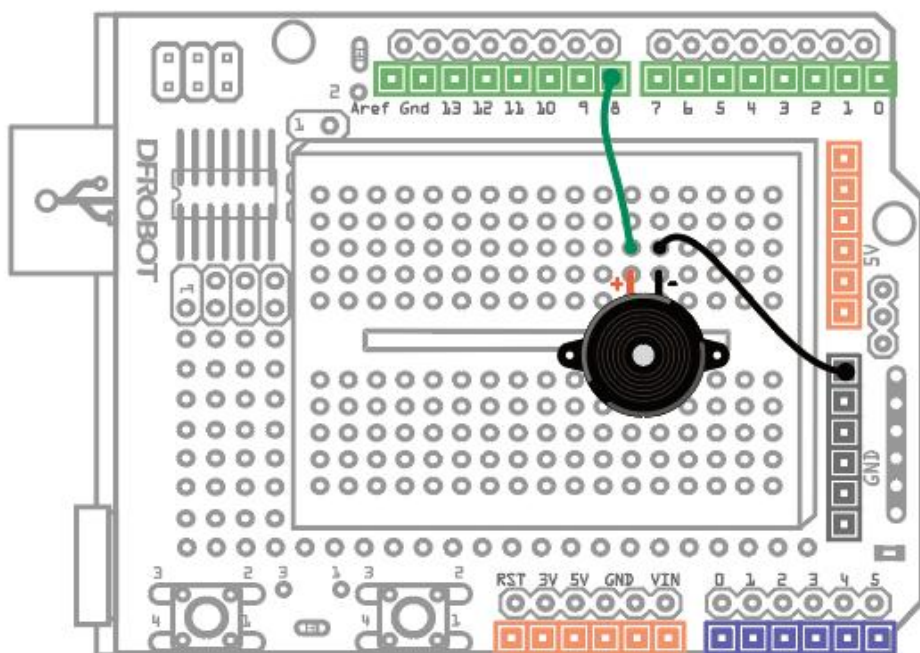


图 1 音乐蜂鸣器连线图

示例代码

样例代码 - Main Sketch:

//项目 - 音乐蜂鸣器

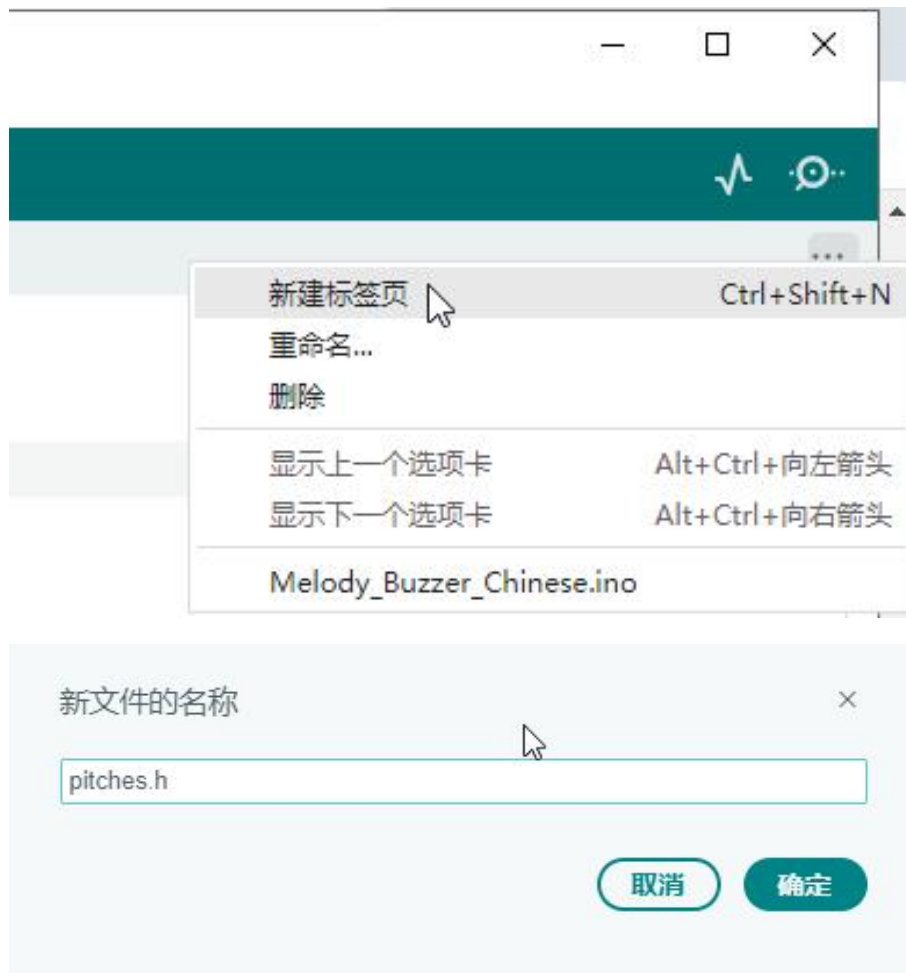
```
#include "pitches.h" // 包含一个头文件
```

```
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。

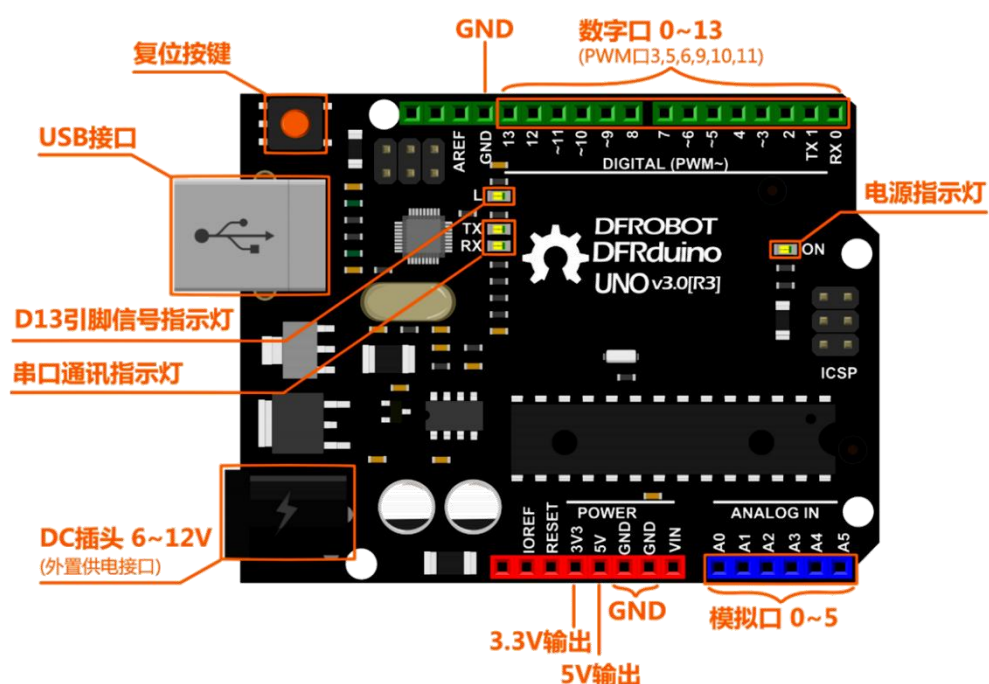


图 2 Arduino UNO 复位键位置

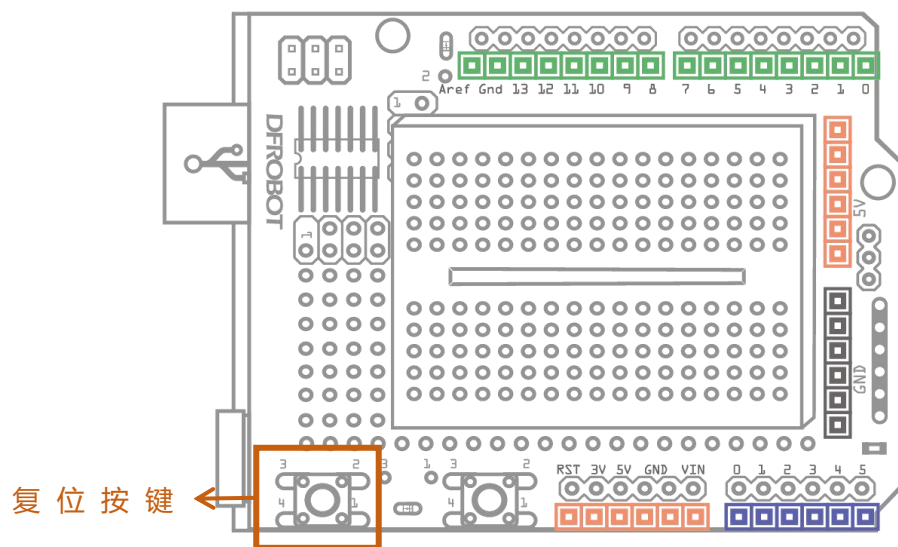


图 3 扩展板复位键位置

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 Potentiometer
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。

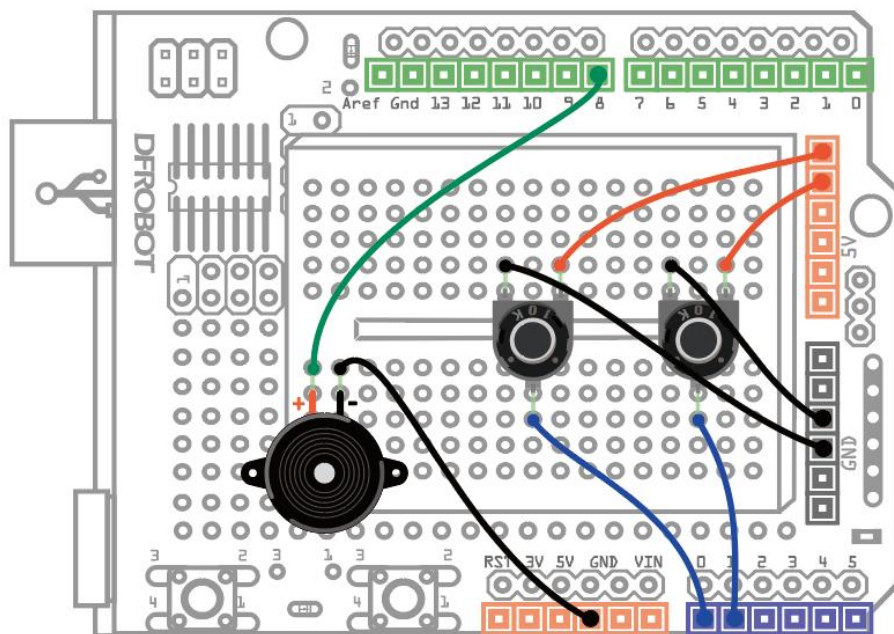


图 1 DJ 调音台连线图

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

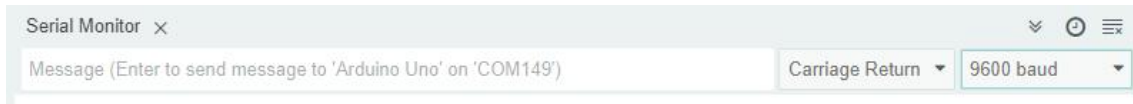
noTone(piezo); // 停止播放音调

}
```

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。

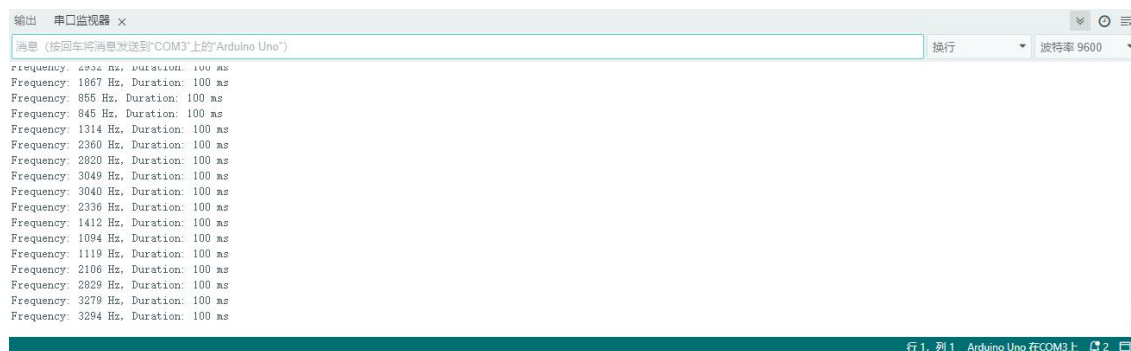
```
Serial.print("Frequency: ");  
Serial.print(frequency);
```

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：



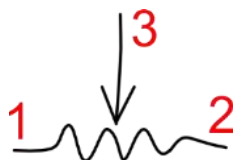
使用与项目【音乐蜂鸣器】相同的程序来播放声音并设置间隔，如有疑问，请参考之前的内容。

硬件回顾

电位器

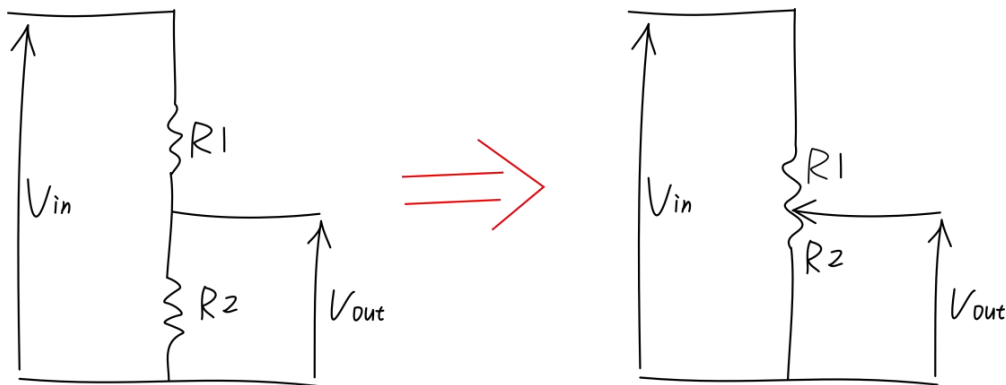
电位器可以理解为是一个电阻，只是这个电阻阻值可变。我们这里可调节的范围是 $0\sim 10K\Omega$ 。电阻两端接电源，通过中间引脚调节阻值，随着电阻值的改变而带动电压变化。我们用模拟口 0 读取到这个变化中的电压值，并转换为对应的音调和节奏。这就是整个的控制过程。

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 V_{out} ）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

	Piezo Disk 压电陶瓷	x1		Zener Diode 稳压二极管	x1
	Buzzer 蜂鸣器	x1		Resistor 1M 1M电阻	x1
	Jumper Cables M/M 跳线（公公头）	x4			

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。

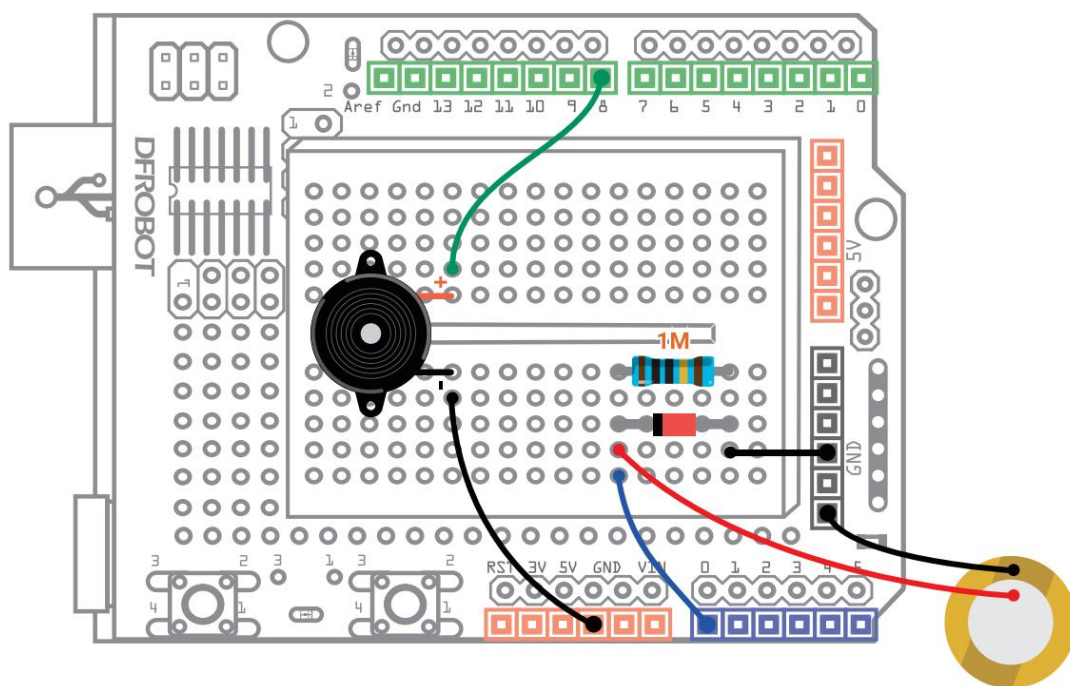


图 2 敲击电子鼓连线图

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模拟输入引脚和蜂鸣器连接的模拟输出引脚。

接着声明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。

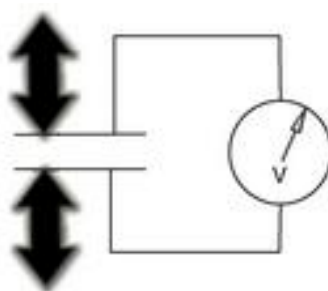


图 2 当压电换能器发生形变时就会产生一个电荷，挤压和松开换能器，就会产生一个变换的电压

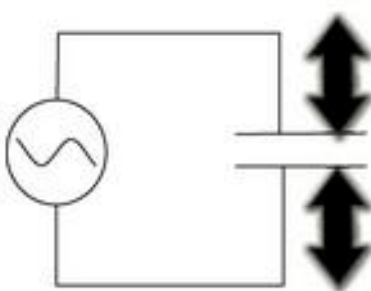


图 3 当对压电换能器施加一个变化的电压时，换能器就会发生形变

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

 5MM LED 5MM LED灯	x1	 Resistor 220R 220欧电阻	x1
 Ambient Light Sensor 光敏电阻	x2	 Resistor 1M 1M电阻	x2
 Jumper Cables M/M 跳线（公公头）	x8		

硬件连接

按下图进行硬件连接。

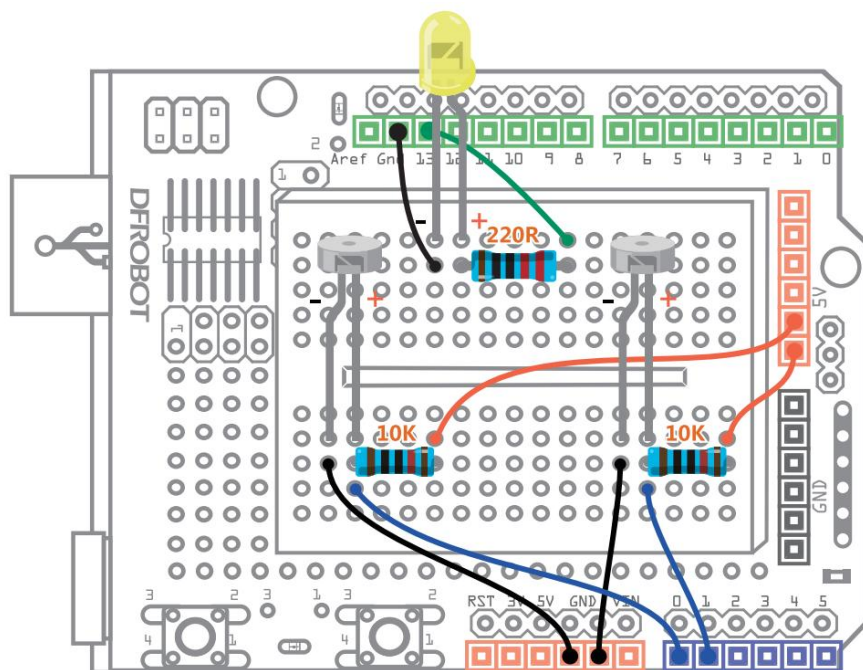


图 1 手势控制灯连线图

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

 5MM LED 5MM LED灯	x3	 Resistor 220R 220欧电阻	x3
 Pushbutton 按键开关	x3		
 Jumper Cables M/M 跳线（公公头）	x12		

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。

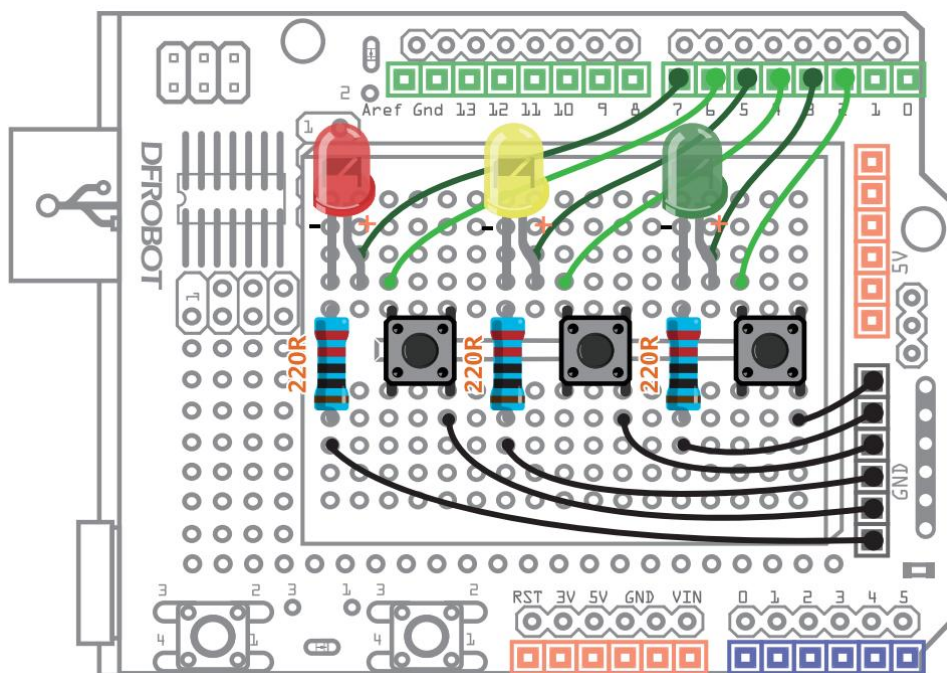


图 1 反应速度测试连线图

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(LED_S[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
```

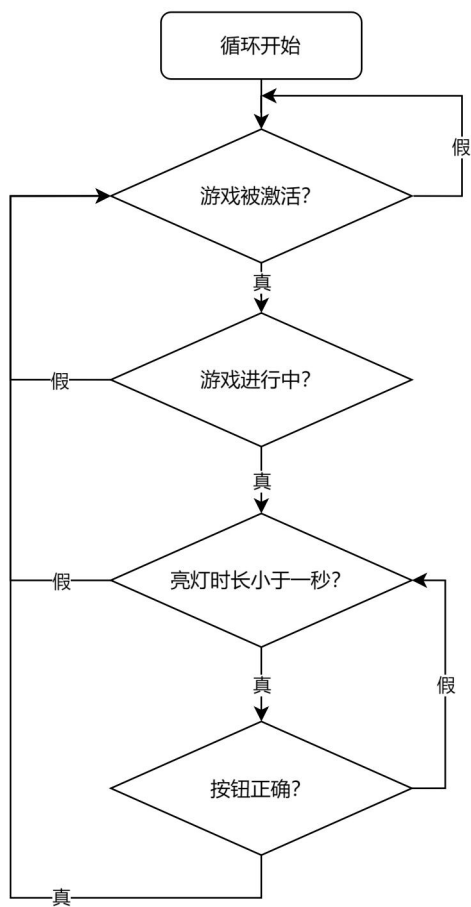
```
if (!gameStarted && digitalRead(BUTTONS[0]) == LOW &&  
digitalRead(BUTTONS[1]) == LOW && digitalRead(BUTTONS[2]) == LOW  
) {  
    // 游戏初始化  
    gameStarted = true;  
}
```

```
// 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。

游戏中进行：

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 `false`（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



8-Segment LED
八段数码管 **x1**



Resistor 220R
220欧电阻 **x9**



Tilt Switch Sensor
倾斜开关 **x1**



Jumper Cables M/M
跳线（公公头） **x12**

硬件连接

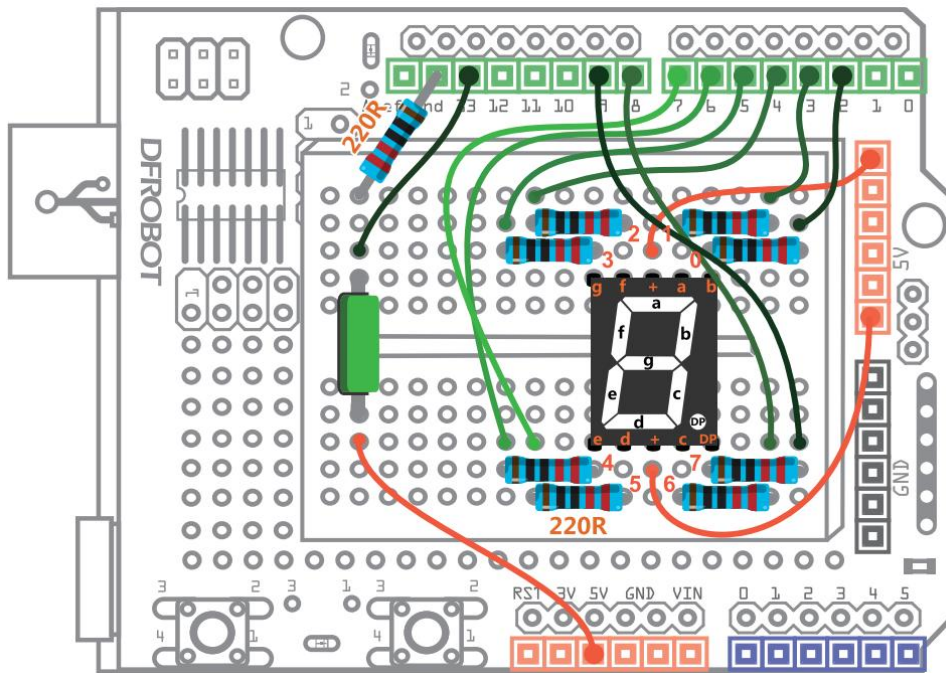


图 1 数码管摇骰子连线图

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

 5MM LED 5MM LED灯	x1	 NPN Transistor NPN三极管	x1
 Resistor 1K 1K电阻	x1	 Resistor 220R 220欧电阻	x2
 Jumper Cables M/M 跳线（公公头）	x7		

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。

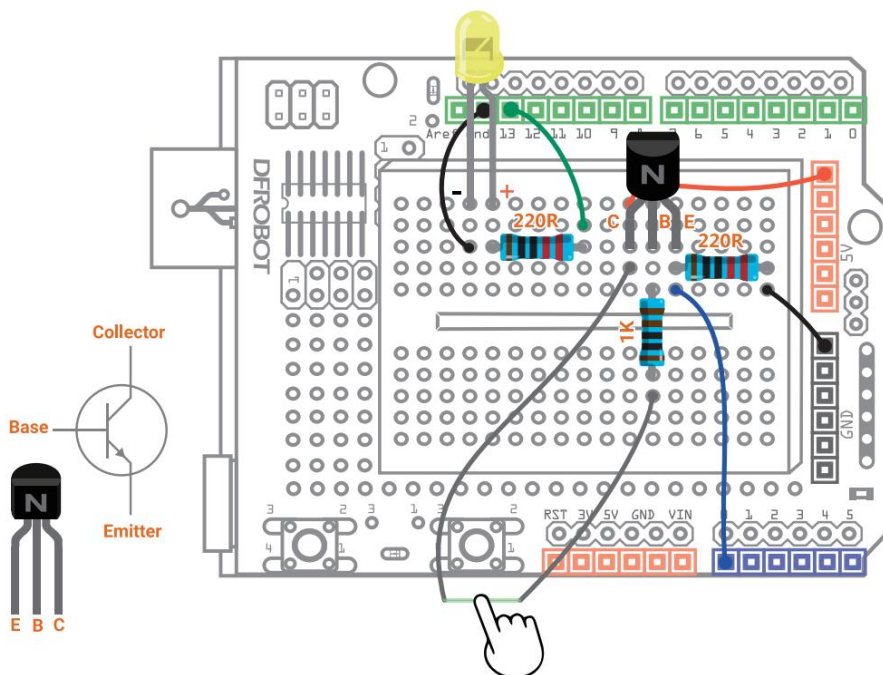


图 1 指尖开关连线图

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。

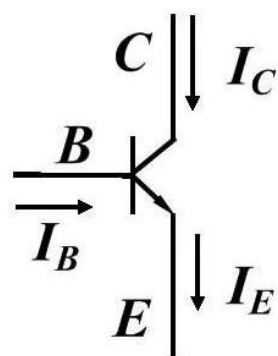


图 2 NPN 型三极管

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 $1M\Omega\sim6M\Omega$ ），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

 **10K Potentiometer**
10K电位器 **x1**

 **NPN Transistor**
NPN三极管 **x1**

 **Resistor 1K**
1K电阻 **x1**

 **Diode**
二极管 **x1**

 **130Motor**
130小马达 **x1**

 **Fan**
透明风扇叶 **x1**

 **Jumper Cables M/M**
跳线 (公公头) **x5**

硬件连接

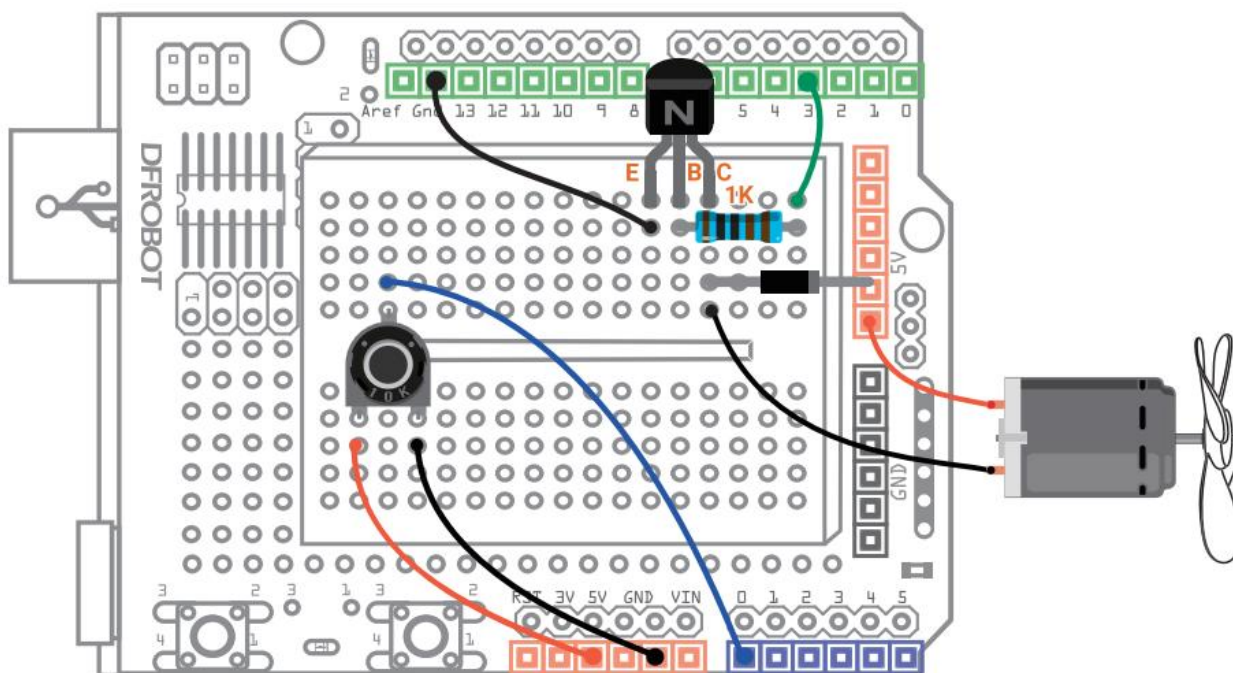


图 1 换挡风扇连线图

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。

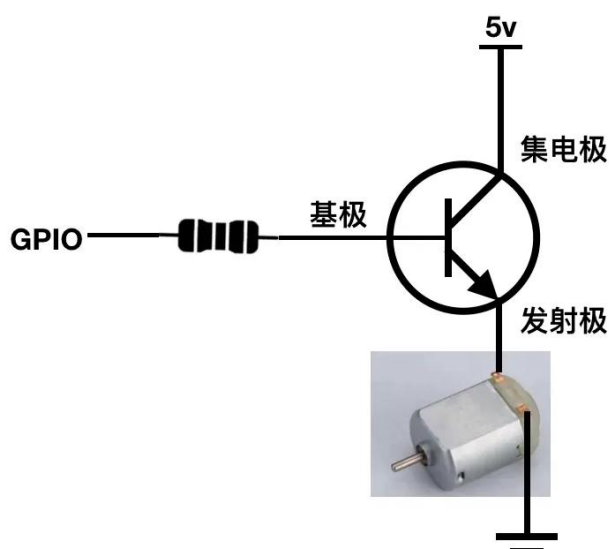


图 2 130 小马达驱动方式

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。

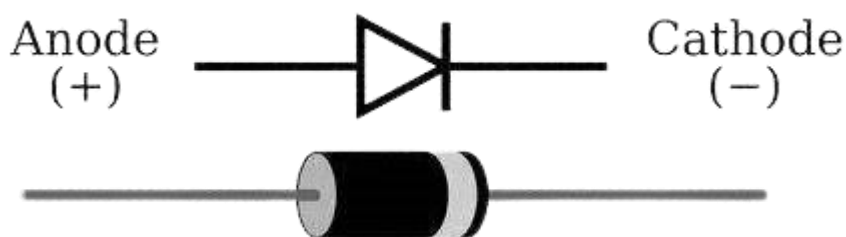


图 3 二极管正负极

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

 NPN Transistor NPN三极管	x2	 PNP Transistor PNP三极管	x2
 Resistor 1K 1K电阻	x2	 Diode 二极管	x4
 Jumper Cables M/M 跳线（公公头）	x16		

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）。

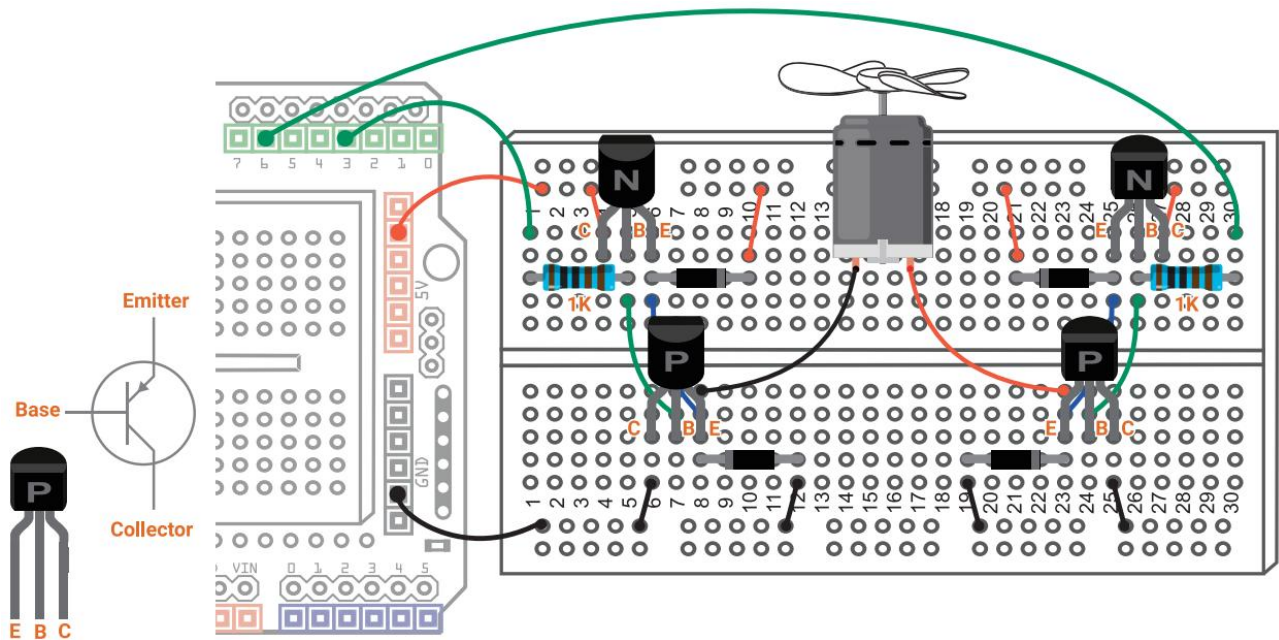


图 1 H 桥电路驱动

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。

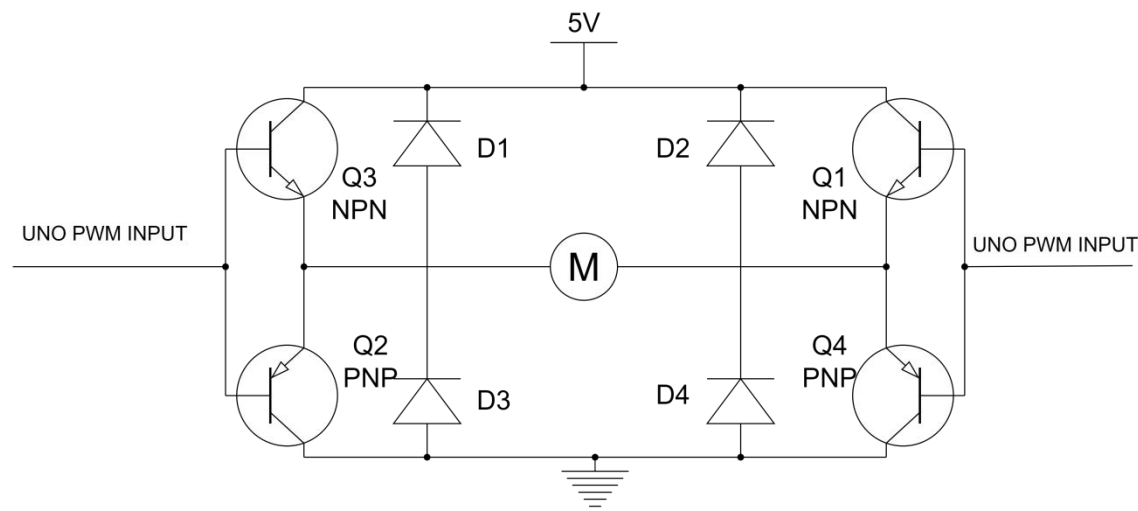


图 2 H 桥电路

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

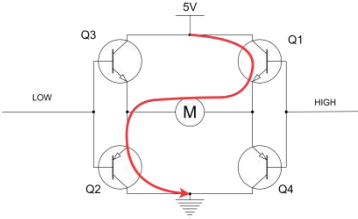
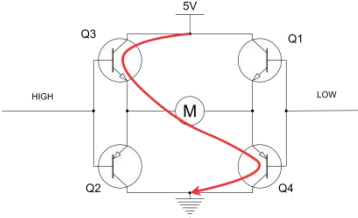
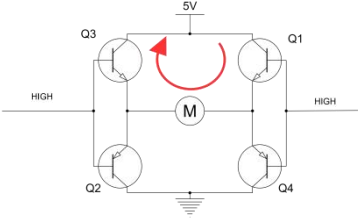
低电平	高电平		正转
高电平	低电平		反转
高电平	高电平		停止

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

 PNP Transistor PNP三极管 x1	 Electret Microphone 麦克风 x1	 Capacitor 0.1μF 0.1μF电容 x1
 Resistor 1K 1K电阻 x1	 5MM LED 5MM LED灯 x1	 Resistor 220R 220欧电阻 x1
 Resistor 4.7K 4.7K电阻 x1	 Resistor 1M 1M电阻 x1	 Jumper Cables M/M 跳线（公公头） x9

硬件连接

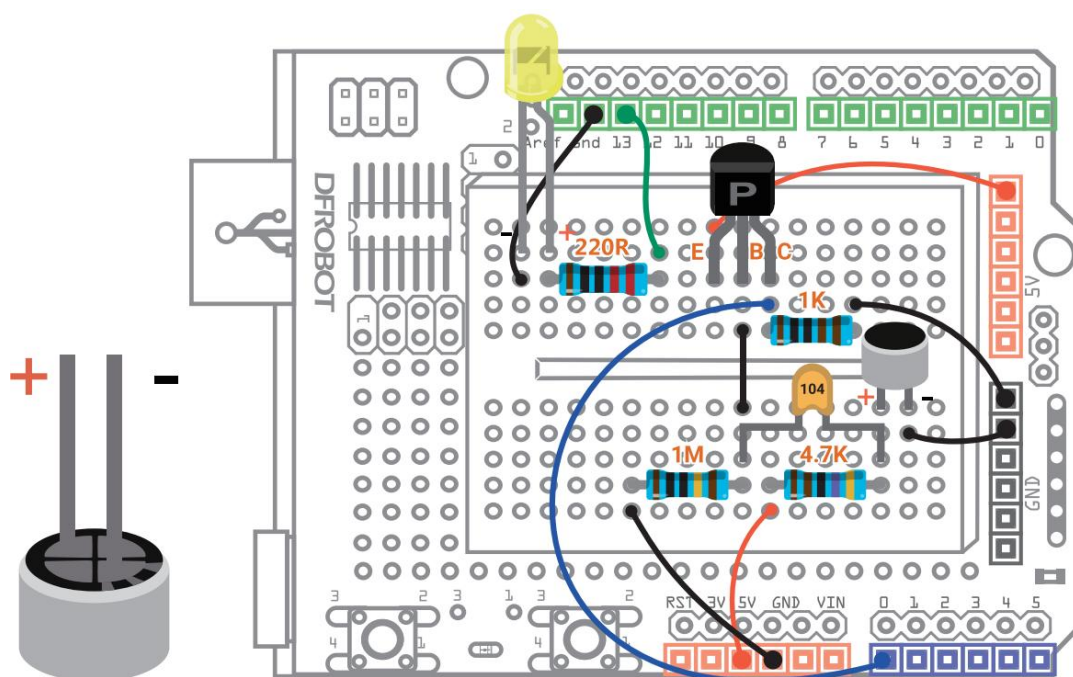


图 1 声控灯连线图

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：

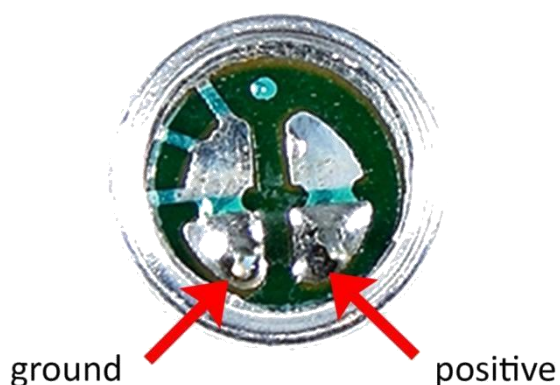


图 2 驻极体麦克风引脚

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：

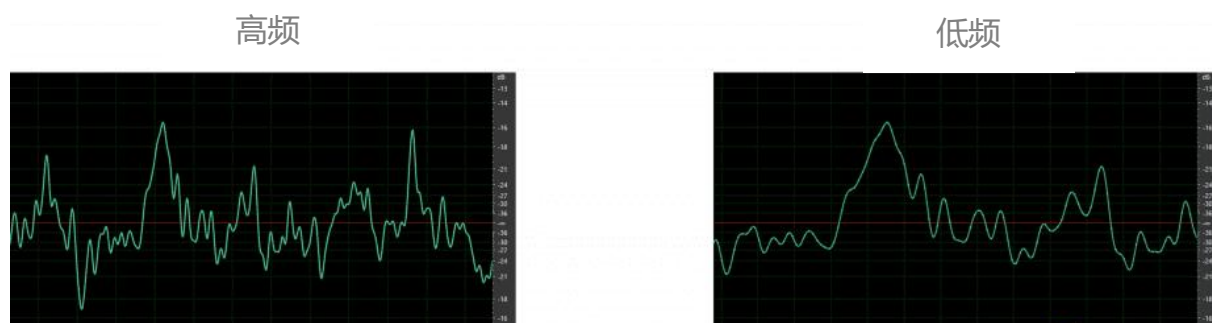


图 3 频率

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。

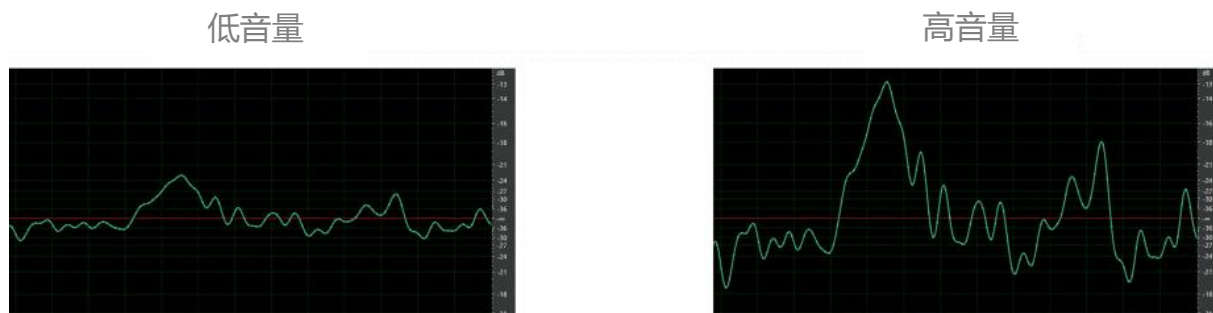


图 4 振幅

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。

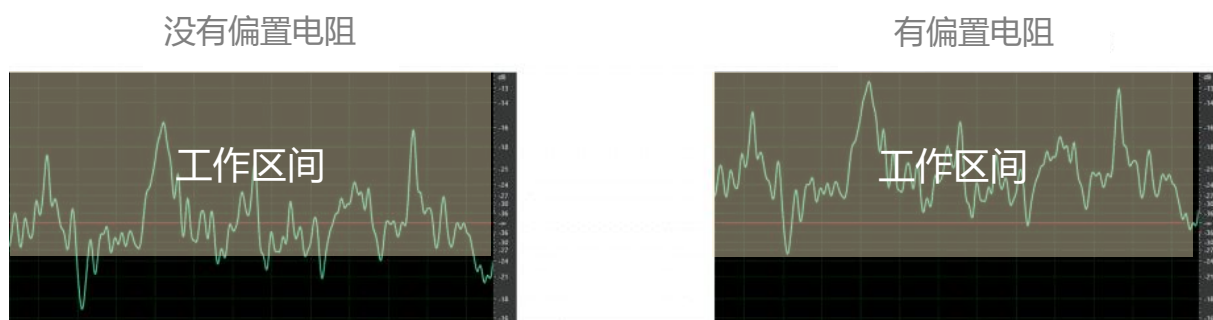


图 5 工作区间

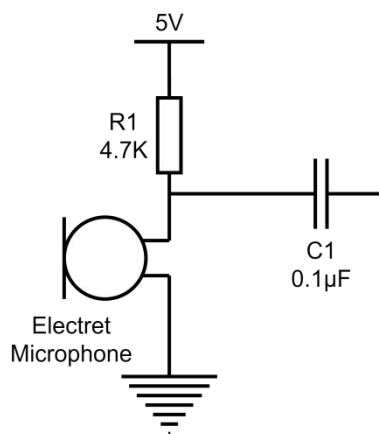


图 6 麦克风部分电路

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：

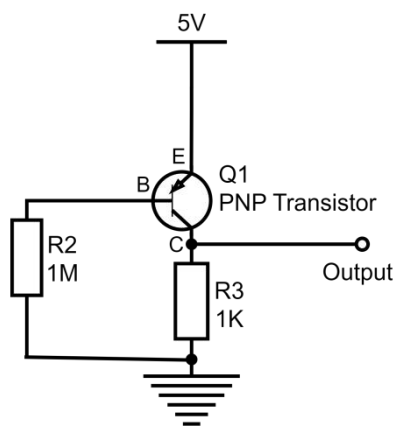


图 7 放大电路部分

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。

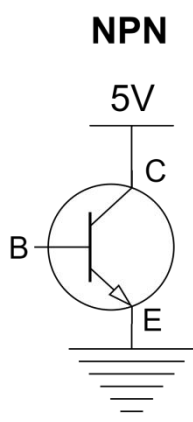


图 8 NPN 型三极管

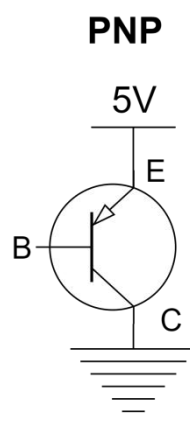


图 9 PNP 型三极管

(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

 PNP Transistor PNP三极管 x1	 Electret Microphone 麦克风 x1	 Capacitor 0.1μF 0.1μF电容 x1
 Resistor 1K 1K电阻 x1	 Resistor 4.7K 4.7K电阻 x1	 Resistor 1M 1M电阻 x1
 Servo 舵机 x1	 Jumper Cables M/M 跳线（公公头） x10	

硬件连接

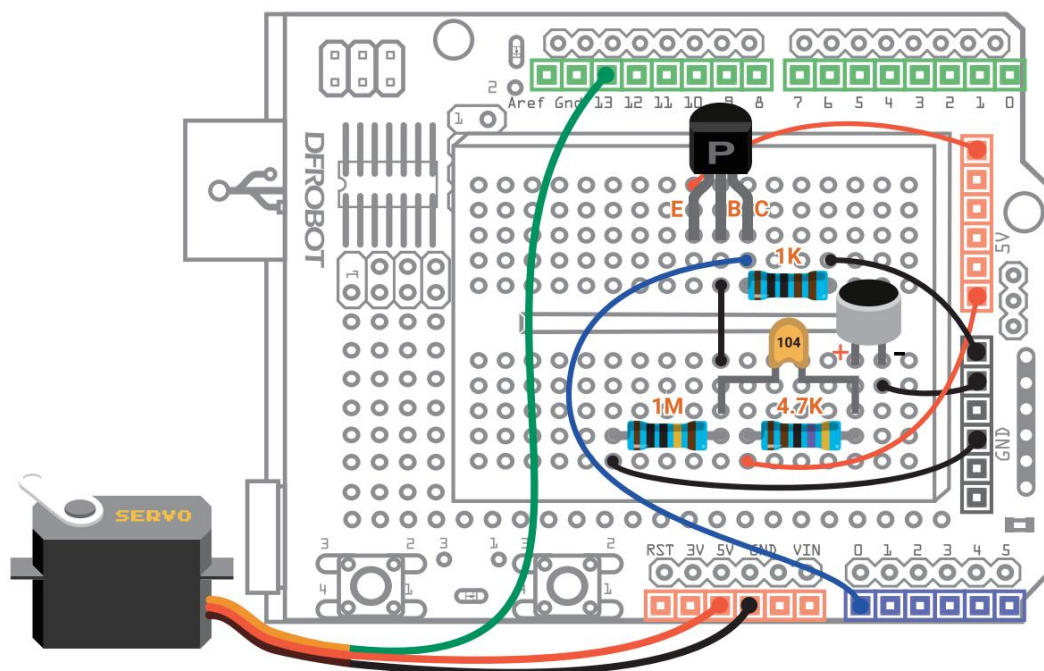


图 1 指针式噪音计

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

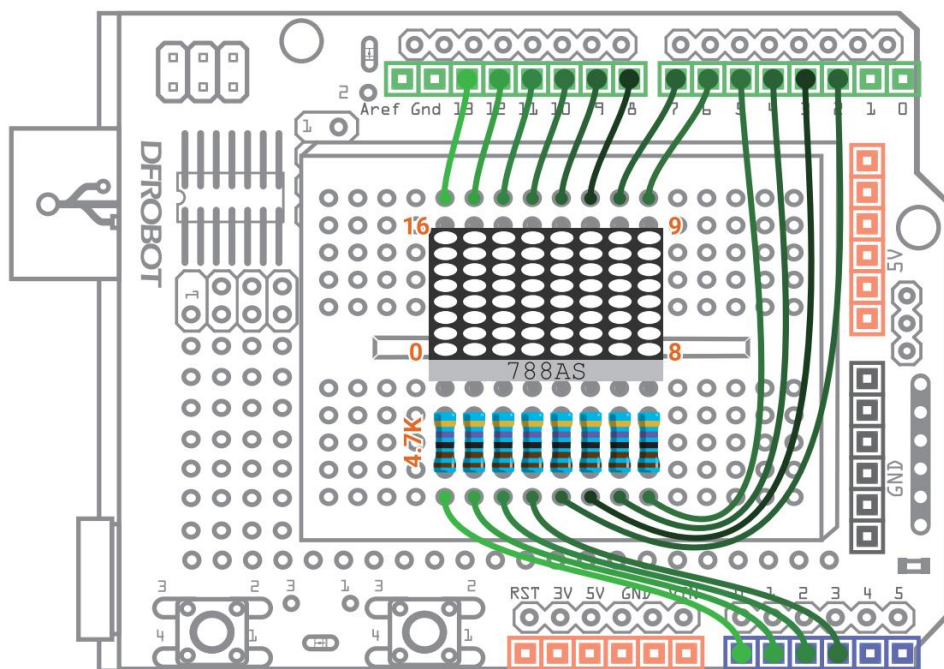


图 1 点亮点阵连线图

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，当对应的某一行置高电平，某一行置低电平，则相应的二极管就亮。

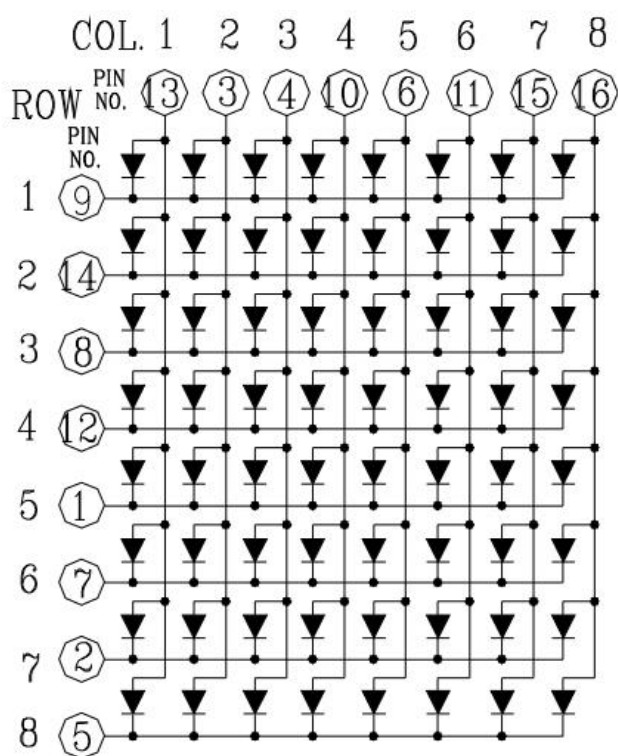


图 2 内部电路图

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

	Tem.Sensor 温度传感器	x1
	Resistor 4.7K 4.7K电阻	x8
	8x8 LED Matrix 8x8 LED点阵	x1
	Jumper Cables M/M 跳线 (公公头)	x19

硬件连接

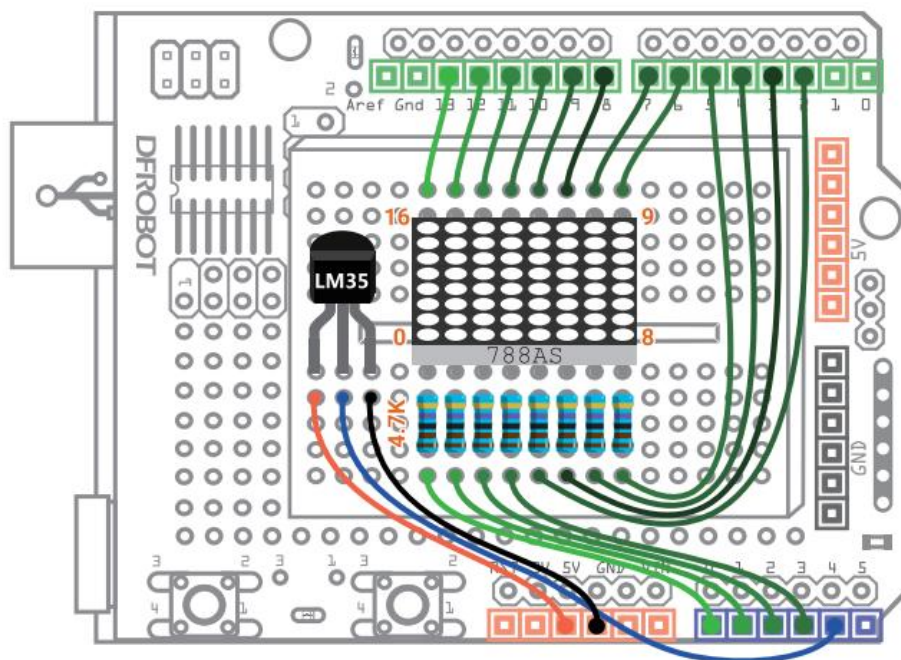


图 1 温度计连线图

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,
0b01100110,
0b00111100},

// 1
{0b00011000,
0b00111000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b11111110},

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

DFRobot

温度计

```
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,  
0b01111110,  
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】。

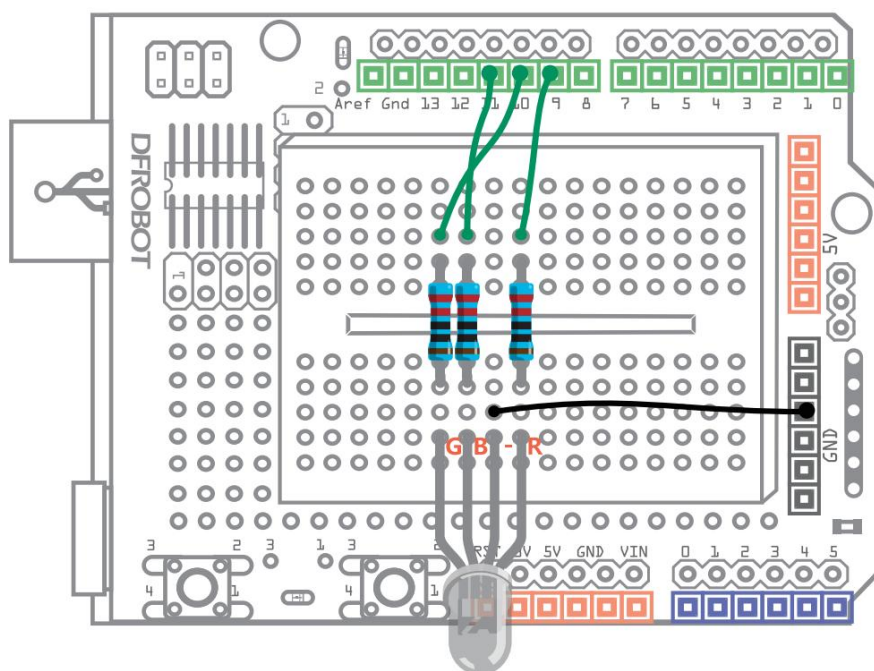


图 1 调光面板连线图

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>

Welcome to Processing!

Processing is a flexible software sketchbook and a language for learning how to code. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. There are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning and prototyping.

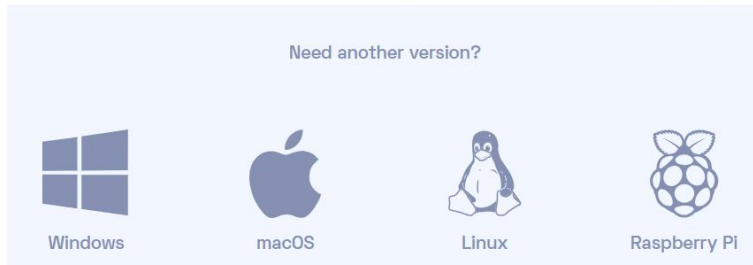
[Download](#)[Reference](#)[Donate](#)

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)

Download Processing 4.3 for Windows

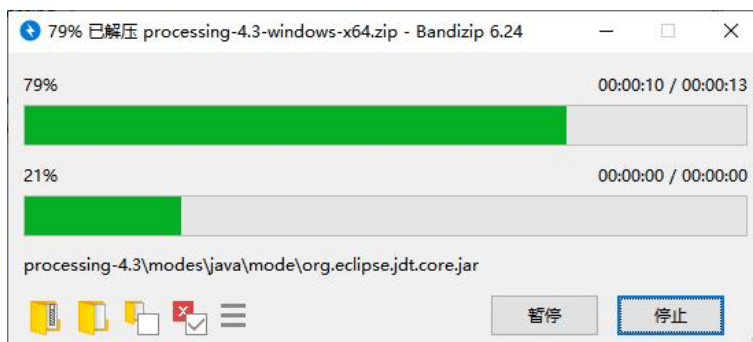
Windows • Intel 64-bit • 214 MB • ©



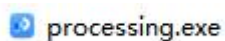
(如果你是用的其他系统的计算机，请参考 <https://processing.org/tutorials/gettingstarted>)

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



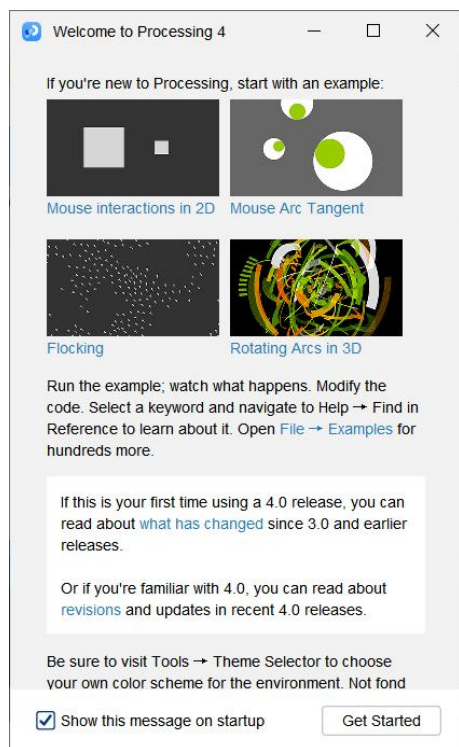
双击解压缩文件中的 processing.exe 文件以运行 processing。



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

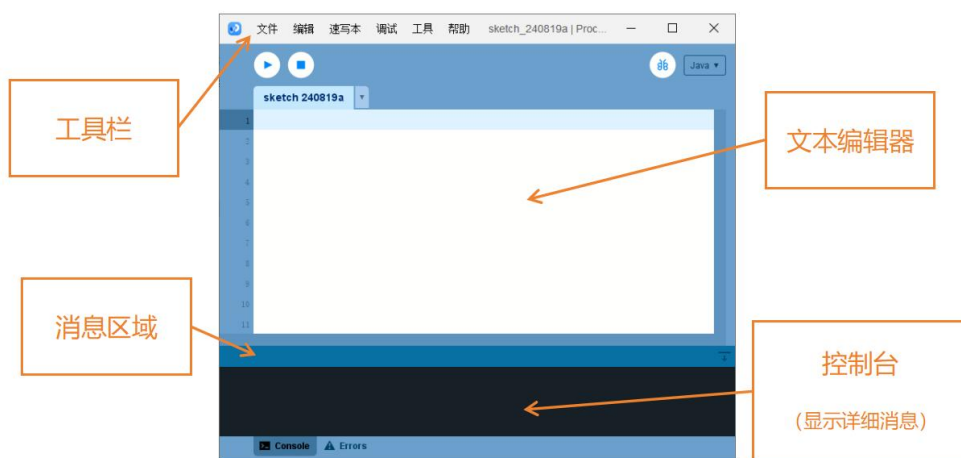
能的解决方案。

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



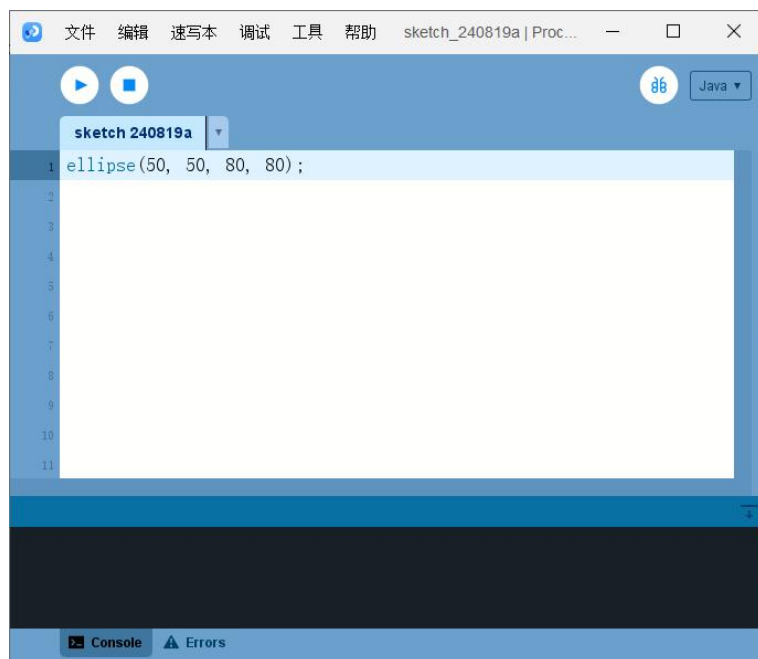
编写你的第一个 Processing 程序

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：

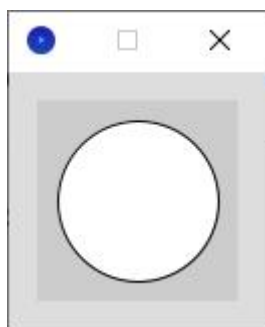
```
ellipse(50, 50, 80, 80);
```

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。

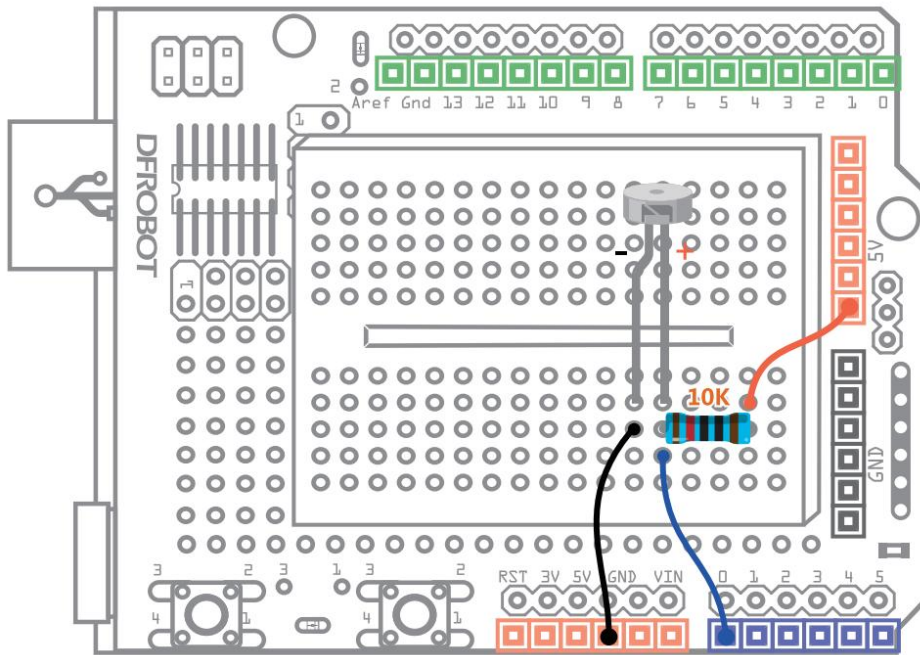


图 1 绘制光线连线图

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）。