西湖大学创新工坊第一期课程

在线仿真：https://wokwi.com/projects/new/arduino-uno

作者：高建生

时间：2024/9/18

目录

第一章 Arduino 编程基础 1

1.1 什么是 Arduino 开发板 . . . 1
1.2 与 Arduino 的初次互动 . . . 1
1.2.1 硬件连接 . . . 1
1.2.2 软件基础 . . . 2
1.2.3 像搭积木一样写代码 . . . 2
1.2.4 上传代码 . . . 3
1.2.5 几点提醒 . . . 3
1.3 数据类型，常量和变量 . . . 4
1.3.1 二进制和十六进制数 . . . 4
1.3.2 数据是如何存储起来的 . . . 5
1.3.3 数据为什么有类型上的区别 . . . 6
1.3.4 常量和变量 . . . 6
1.4 串口监视器 . . . 8
1.4.1 通过串口向串口监视器发送数据时调用的函数 . . . 8
1.4.2 char 型数据和 ASCII 码 . . . 9
1.5 符号和小数点如何存储的 . . . 10
1.6 Arduino 基础之运算符 . . . 12
1.6.1 位运算的常见用途 . . . 12
1.7 Arduino 基础之控制语句 . . . 16
1.7.1 if 语句 . . . 16
1.7.2 for 语句 . . . 19
1.7.3 while 语句 . . . 20
1.7.4 continue 语句 . . . 21
1.7.5 return 语句 . . . 22
1.8 Arduino 基础之函数 . . . 24
1.9 Arduino 基础之变量的作用域 . . . 25
1.9.1 局部变量 . . . 25
1.9.2 全局变量 . . . 25

第二章 Arduino 硬件基础 27

2.1 输入、输出接口 . . . 27
2.2 脉冲和时序 . . . 29
2.2.1 脉冲 . . . 29
2.2.2 时序 . . . 29

第三章 电子秤的制作 31

3.1 外形设计 . . . 31
3.2 安装和连接 . . . 31
3.3 到写代码的时间了 . . . 32
3.3.1 1602 显示模块 . . . 32
3.3.2 初次认识库函数 . . . 32

3.3.3 称重传感器和 HX711 模块的使用 34
3.3.4 将读出的数据转换为质量 37
3.3.5 滤波 39

第一章 Arduino 编程基础

1.1 什么是 Arduino 开发板

指用 Arduino IDE 开发的开发板的统称，不同的 Arduino 开发板搭载的芯片可能是不同的，在我们的项目中，使用得最多的 Arduino UNO(图1.1) 和 Arduino Pro Mini(图1.2) 都是使用 Atmega328p 芯片。

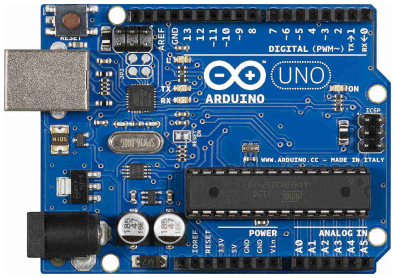


图 1.1

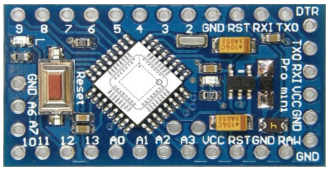


图 1.2

刚接触 Arduino 开发时，学习者最想知道：能用 Arduino 开发板开发什么？如何开发？这两个问题的答案。用 Arduino 开发板，我们可以:

- 学习基于 C/C++ 的 Arduino 语言；
- 了解硬件方面的知识；
- 使用传感器；
- 熟悉常见的通讯协议，让遵守相同协议的不同的设备展开“对话”；
- 开发具有控制、数据收发功能的产品。

1.2 与 Arduino 的初次互动

Arduino 开发涉及硬件、软件两个方面的内容，每个方面又分成不同层次，对于初学者而言，先在软件方面打下扎实的基础，硬件方面的提高由于需要多个学科知识的支撑，因此要不急不躁，循序渐进。我们先通过一个例子了解一下 Arduino 的开发过程，用 Arduino 开发板控制 LED 的亮灭。

1.2.1 硬件连接

如图1.3所示，Arduino UNO 开发板上有 D0-D13、A0-A5 共 20 个 I/O 接口，所谓 I/O 接口指既可以输出，也可以输入，因此使用这些接口时需要先设置好输出还是输入。

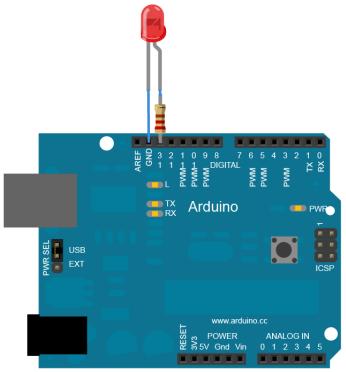


图 1.3

输出、输入什么呢？输出稳定的 5V(当然在允许的功率范围内的话）或 0V 电压，输入指接收外部的电压信号，并将其识别为高电平或低电平，具体什么范围内的电压会被识别为高（低）电平，不同的开发板会有所不同，可以通过实验来确定，接收输入时，输入的功率也有所限制。

1.2.2 软件基础

Arduino 开发能使用的软件比较多，Arduino IDE 为官方提供的软件，运行界面如图1.4所示，创建的文件叫做 sketch，文件类型为 *.ino，新建文件时，软件自动为文件命名，比如“*sketch_dec29a*”，dec29a 表示 12 月 29 号创建的第一个 sketch。

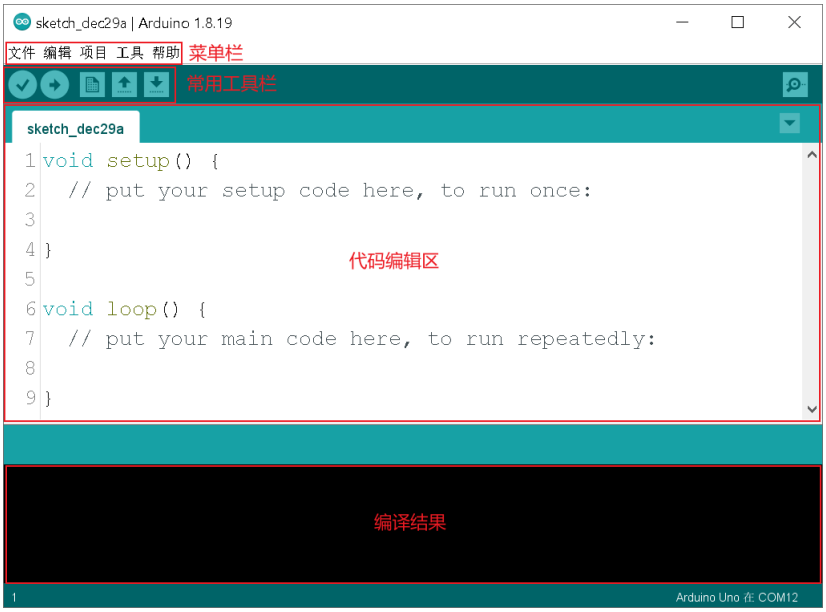


图 1.4: Arduino UNO 开发板

Arduino IDE 的代码包含三个部分：

- void setup()
  每次 Arduino 上电或重启后，void setup() 内的代码只运行一次，一般用来设置 Arduino 开发板引脚的工作模式（输入还是输出）、配置并开启开发板与计算机的串口通信等等。
- void loop()
  loop 有“循环”的意思，程序运行时，loop() 部分的代码不断循环执行，除非人为地让程序运行停下来。
- void setup() 和 void loop() 之外的区域
  往往包括：引入的库文件、定义的全局变量、宏定义以及自定义的函数等等。

1.2.3 像搭积木一样写代码

Arduino 编程比较容易上手，因为有大量的函数可以使用，函数其实就是一些事先写好了的实现特定功能的代码块，下面我们调用函数让 LED 灯亮一秒，熄灭一秒，如此反复，电路连接如图1.3。
实现方案非常简单：设置 13 引脚为输出，让 13 引脚输出高电平（5V），则 LED 发光，延时 1 秒后让 13 引脚输出低电平（0V），则 LED 熄灭，再延时 1 秒，这些操作需要调用的函数分别如下：

- 调用 pinMode() 函数设置 13 引脚的工作模式为输出
  pinMode(13,OUTPUT);
- 调用 digitalWrite() 函数让 13 引脚输出高电平
  digitalWrite(13,HIGH);
- 用 delay() 函数让 LED 的状态持续 1s

delay(1000);

- 调用 digitalWrite() 函数让 13 引脚输出低电平
  digitalWrite(13, LOW);
- 用 delay() 函数让 LED 的状态持续 1s
  delay(1000);

完整代码如下：

```
/*
  调用函数让LED灯闪烁
*/
void setup()
{
  pinMode(13, OUTPUT); //设置pin13为输出模式
}

void loop()
{
  digitalWrite(13, HIGH); //调用digitalWrite()函数让pin13输出高电平
  delay(1000); //调用delay()函数延时1000ms，即1s
  digitalWrite(13, LOW); //调用digitalWrite()函数让pin13输出低电平
  delay(1000); //调用delay()函数延时1s
}
```

1.2.4 上传代码

上传代码分以下几步完成：

- 连接计算机与 Arduino UNO
- 选择开发板型号
  Arduino IDE 中，展开” 工具 > 开发板 >Arduino AVR Boards”，如图1.5所示。
  选择开发板的型号为：Arduino/Genuino UNO。
- 选择与计算机连接的端口
  Arduino IDE 中，展开“工具 > 端口”，选择计算机与 Arduino UNO 连接的端口，如图1.6所示。
  如果显示多个端口号，可以拔掉 USB 连接线，再次查看端口号，消失了的那个就是计算机连接开发板时使用的端口号。
- 编译并上传代码
  点击常用工具中的“上传”，Arduino IDE 将完成代码的编译和上传，如果编译失败，软件会在编译结果中显示出错的原因。
  注：通过 USB 数据线上传代码时，不用选择工具菜单下的“编程器”，编程器下的选项是使用外部编程器设置熔丝位，烧写 bootloader 或者代码时使用的。

代码上传成功的话，可以看到，板载 LED 灯亮 1 秒，熄灭 1 秒，并不断重复。

1.2.5 几点提醒

- 区分大小写、中英文符号
  初学者很容易混淆中英文标点符号和英文字符的大小写，图1.7为使用了中文标点符号的出错提示。
- 养成写注释的习惯

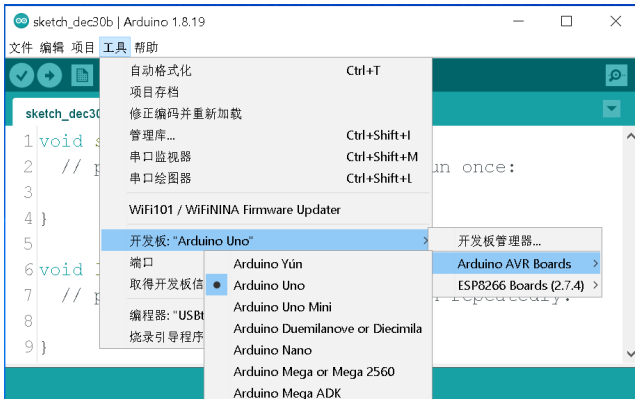


图 1.5: 选择开发板的型号

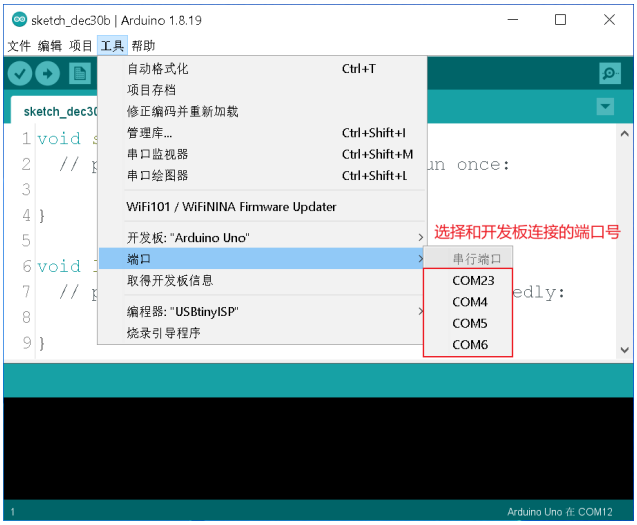


图 1.6: 选择连接开发板的端口号

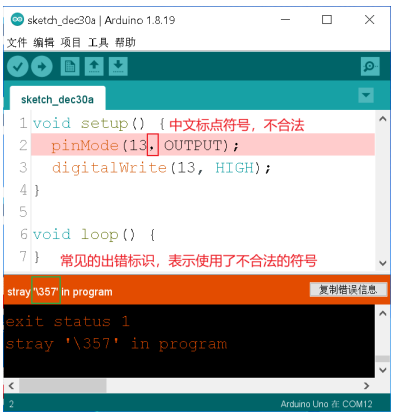


图 1.7: 常见的错误标识“\357”

代码中，”//” 后面的内容为代码解释，编译代码时，注释部分会被忽视掉。初学者最好写详尽的注释，逐渐过渡到对代码块或重要的代码行写注释。如果注释比较短，一行就可以写下，用“//注释内容”的方式，否则用“/* 注释内容 */”这种方式，注释分成三个层次：

- 程序信息的注释
- 代码块的注释
- 某行代码的注释

1.3 数据类型，常量和变量

1.3.1 二进制和十六进制数

二进制数

二进制数，即“满 2 进 1 位”，只用到了两个数字，0 和 1，表1.1列出了 0-4 所对应的二进制数。

为了区别二进制数和其它进制数，在二进制数的左边添加 B（或 0b，但不能仅用小写 b），比如 B100100 或 0b100100。

二进制数中，各位的权重如图1.2所示。

注：二进制数中，权重最高的位称为 MSD(most significant digit)，权重最低的位称为 LSD(least significant digit)。

十进制数	二进制数
0	0
1	1
2	10
3	11
4	100

表 1.1: 二进制和十进制数的对应

权重	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}
位	*	*	*	*	*	.	*	*
	MSD							LSD

表 1.2: 二进制数各位的权重

十六进制数

顾名思义，“十六进制”就是指“满 16 进 1 位”，十六进制数由 16 个数字构成，0、1、2、3、4、5、6、7、8 和 9 是一位数，更大的数字不能用 10、11、12、13、14 和 15 来表示，因为十六进制数中，这些数代表的是其它数，比 9 大比 16 小的其它数字分别用字符 A、B、C、D、E 和 F 来代替，图1.3列出了十进制数所对应的十六进制数。

十进制数	十六进制数
0	0
1	1
...	...
9	9
10	A
...	...
15	F
16	10
17	11

表 1.3: 十进制和十六进制数的对应

十六进制数的优点是表示比较大的数和二进制数特别方便，比如二进制数 B1111 用 F 表示就可以了，B10000 用 10 表示，为了和十进制数区别开，十六进制数 10 写成 0x10，x 可以大写。
十六进制数中，各位的权重如表1.4所示。
注：十进制、二进制和十六进制只是数的不同表示方法，完全可以把十进制数的运算法则推广到二进制、十六进制数。

1.3.2 数据是如何存储起来的

数据在存储器里都是用高低电平的排列的方式存储起来的，能实现高低电平两种状态的器件就是电容器，充电后为高电平状态，放电后为低电平状态，高电平用 1，低电平用 0 来表示的话，存储起来的指令和数据就可以用二进制数来表示了。比如：存储数值 8，其二进制数为 B1000，表面上 4 个电容器就 OK 了，但实际上以 8 个为存储的最小单元，称为 1 个 byte（字节），byte 中的每一位称为一个 bit（位），因此存储数值 8 时，要么用 1 个 byte 的空间，要么 2 个 bytes 的空间，甚至更多 bytes 的空间。
如果是 1 个 byte 的空间的话，用二进制数表示为

$$(MSB)00001000(LSB)$$

如果是 2 个 bytes 的空间的话，用二进制数表示为

$$(MSB)0000000000001000(LSB)$$

权重	16^4	16^3	16^2	16^1	16^0		16^{-1}	16^{-2}
位	*	*	*	*	*	.	*	*
	MSD							LSD

表 1.4: 十六进制数各位的权重

无论数据占据多少个字节，数据中的最高位都称为 MSB-most significant bit，数据中的最低位都称为 LSB-least significant bit。

1.3.3 数据为什么有类型上的区别

数据在存储器里到底是怎样的二进制数，这取决于数据的类型？数据还有类型上的区别？举个例子吧！当你看到“1”的时候，既可以把它当成数值，也可以看成符号，反应截然不同，当成数值时，你的反应可能是：好小，看成符号时，你的反应可能是：好直！数据存在多个方面的区别，列举如下：

- 数值还是符号
- 非负数还是可正可负
- 什么范围内的数
- 整数还是小数

等等。

数据类型其实就是为了回答上面这些问题，常见的数据类型如图1.8所示，假设数据为数值、不可能为负、只能在 0-100 之间变化、整数，那就定义类型为 unsigned char 就 OK 了。

Data Type	Size (Bytes)	Range of Values
void	0	null
bool/boolean	1	True/False
char	1	-128 to +127
unsigned char	1	0 to 255
byte	1	0 to 255
int	2	-32,768 to 32,767
unsigned int	2	0 to 65,535
word	2	0 to 65,535
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
float	4	-3.4028235E+38 to 3.4028235E+38
double	4	-3.4028235E+38 to 3.4028235E+38
string	-	character array

图 1.8

1.3.4 常量和变量

数据在存储器里有自己的地址，我们可以通过地址找到数据，但前提是得知道数据存储在什么地方。常用的做法是给数据取个名字，作为数据的全权代表，数据按照值是否改变分为：变量和常量，变量的值可以改变，常量的值不可以改变，比如温度传感器输出的数据是不断变化的，因此是一个变量，假设该变量取名为 tempVal，则 tempVal 代表的就是温度的值了。π 是一个固定值，因此是一个常量，用 piVal 来取名的话，piVal 代表的就是该常量的值了。

变量和常量命名遵循：

- 用字母，数字或下滑线；

- 第一个字符不能是数字；
- 必须以英文字母或下划线 _ 开头。

常量的定义方式

有两种：

#define 常量名 常量值

或者

const 数据类型 常量名 常量值;

注：用 #define 这种方式定义常量时，称为“宏定义”，不以“;”结束，比如：

#define PI 3.1415926

在编译代码前，代码中有 PI 出现的地方统统换成 3.1415926，因此我们写的代码是：

```
#define ledPin 13
void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(500);
  digitalWrite(ledPin, LOW);
  delay(500);
}
```

而编译器实际编译的是：

```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(500);
  digitalWrite(13, LOW);
  delay(500);
}
```

由于将 ledPin 替换成 13 是在编译前完成的，因此常量 ledPin 并不需要占据存储空间。
Arduino IDE 用 #define 这种方式定义了一些常量，可以直接用宏定义的名称代替具体的值，常见的几个是：

- false,true 分别代表 0 和 1
- HIGH,LOW 分别代表 1 和 0

- INPUT,OUTPUT 和 INPUT_PULLUP 分别表示 0，1 和 2
- LED_BUILTIN 也是一个宏定义的常量，但具体代表的值与开发板的型号有关

为什么要宏定义这些常量，一个原因是英文单词表意，一看这些单词就知道是什么意思；另一个原因是便于代码的移植，比如：不同型号的 Arduino 开发板，板载 LED 灯所连接的输出引脚可能不一样，Arduino UNO 的板载 LED 灯连接的是 D13，而 Arduino MKR1000 的板载 LED 灯则连接的是 D6，因此在 Arduino UNO 上，digitalWrite(13,HIGH) 可以点亮 LED，在 Arduino MKR1000 上，digitalWrite(6,HIGH) 才能点亮 LED。但不管是哪种 Arduino 平台，如果我们把 LED 连接的引脚都宏定义为 LED_BUILTIN，那么无论代码移植到什么 Arduino 平台上，digitalWrite(LED_BUILTIN,HIGH) 都可以点亮板载 LED。

变量的定义方式

变量的命名格式如下：

```
数据类型 变量名（=变量初值）;
```

若不给初值，默认为 0，比如：

```
int val;
```

则 val 的初值为 0。

1.4 串口监视器

串口是 Arduino UNO 与电脑或别的模块通信时的一种接口，开发板上串口通信使用的引脚是 D0(RX，接收) 和 D1(TX，发送)，用 USB 数据线连接开发板和电脑进行的就是串口通信，串口监视器是 Arduino IDE 工具菜单下的一个小工具，如图1.9所示，既可以显示开发板通过串口发送给电脑的的数据，比如运算的结果、文字信息等，也可以通过串口向开发板发送数据。

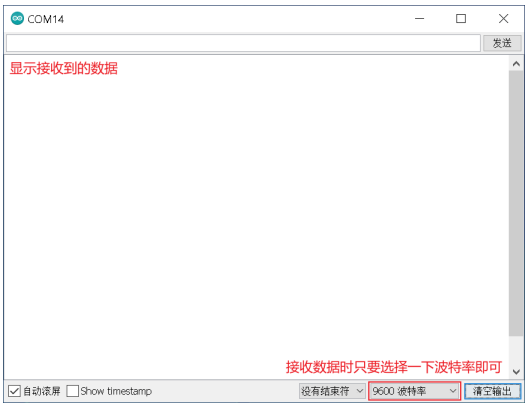


图 1.9: 串口监视器

注：上传代码前选择的端口号就是串口号，目前不用管串口的硬件结构和通信原理。

1.4.1 通过串口向串口监视器发送数据时调用的函数

向串口监视器发送数据需要调用的函数是：

- Serial.begin()
  格式：Serial.begin(波特率)
  作用：设置开发板向电脑发送数据时，每秒传输数据的位数（波特率），并同时开启开发板与电脑的串口通信，比如：

```
Serial.begin(115200); //设置波特率为115200
```

由于串口的设置不需要反复执行，因此该代码写在 void setup() 部分。
串口通信时的波特率不能随意设定，能设置哪些波特率，可以展开串口监视器的波特率列表查看。开发板发送和电脑接收数据的波特率要一致，因此串口监视器的波特率也要设置为 115200。波特率太高，发送或接受数据时，可能会出错。
注意：Serial.begin() 函数的写法和前面使用过的 pinMode()、digitalWrite() 和 delay() 不一样，函数名称包含两个部分，用“.”隔开，其中“.”右边的部分为函数名，左边的部分为对象名，连起来就是 Serial 对象调用 begin() 函数。

- Serial.print() 或 Serial.println() 或 Serial.write()
  格式：Serial.print(数据)，Serial.println(数据)，Serial.write(数据)
  作用：通过串口向串口监视器发送数据

1.4.2 char 型数据和 ASCII 码

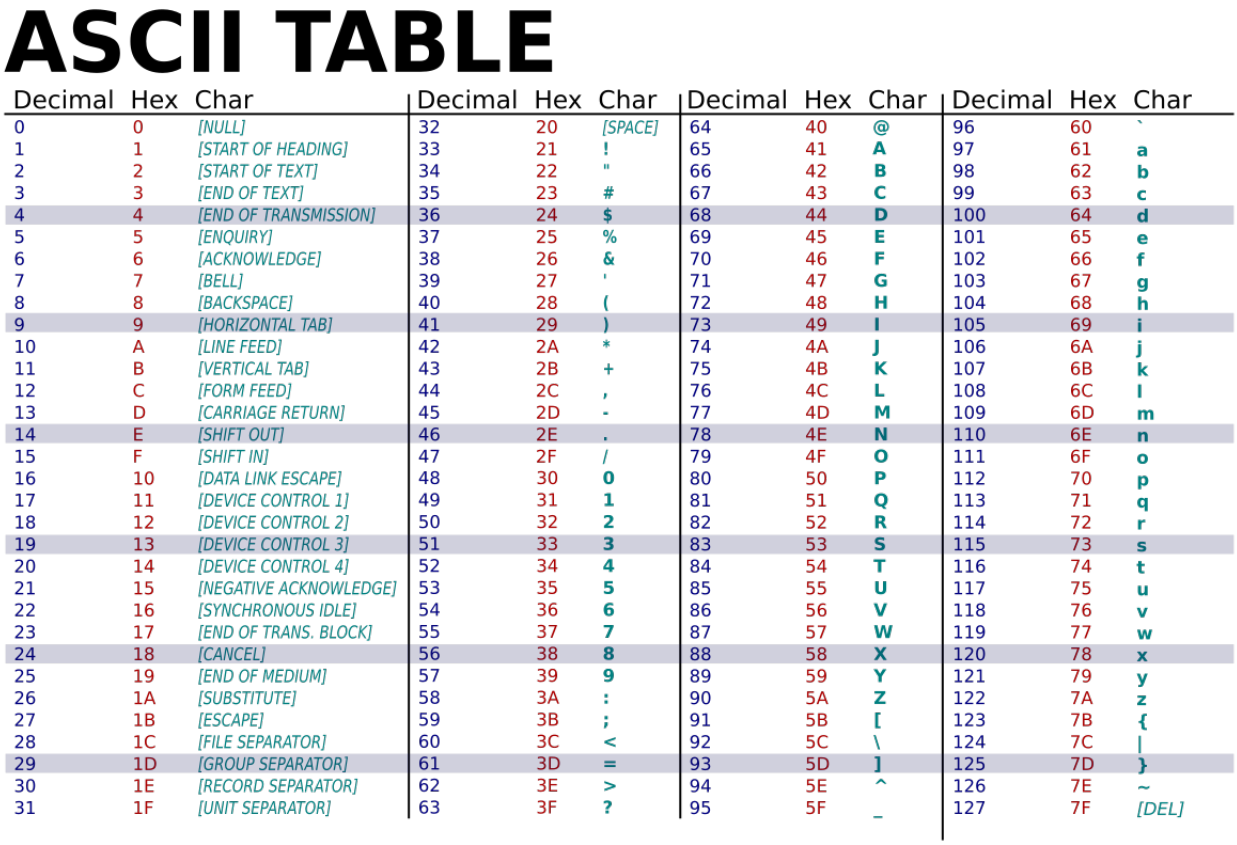
所有的数据类型中需要强调下 char 型，c 和 c++ 对 char 型的定义是不同的，Arduino 语言中 char 和 signed char 的区别只是出现在串口打印时，比如：

```
void setup()
{
  Serial.begin(9600);
  char A =129;
  signed char B=129;
  signed char C=-65;
  Serial.println(A+C);
  Serial.println(B+C);
}

void loop()
{
}
```

进行运算的时候，char 型和 signed char 型没有什么区别，打印结果都是-192。但如果打印 char 型和其它类型的数据时，则会出现区别，比如：

```
void setup()
{
   Serial.begin(9600);
   char A =65;
   signed char B=65;
  Serial.println(A);
  Serial.println(B);
}

void loop()
{
}
```

打印结果分别为：A 和 65。串口打印时，当编译器看到该数据类型为 char 型时，就把该数据当成某个字符的 ASCII 码（国家上通用的一套字符编码，如图1.10所示）并直接将 65 这个 ASCII 码值发送出去，而串口监视器始终将接收到的数据当成 ASCII 码值来对待并将对应的字符显示出来，因此显示字符“A”，而对于别的类型的数据，则是将数据中每位的 ASCII 码发送出去，因此发送 B 的值时，分别发送的是 6 和 5 的 ASCII 码出去，串口监视器接收到 6 和 5 的 ASCII 码当然显示 65 了。



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

图 1.10

1.5 符号和小数点如何存储的

非负整数不存在正负的区别，也没有小数点，其存储非常直接了当，比如：非负整数 8，定义为 unsigned char 型，对应的二进制数为 B00001000，存储起来的就是 B00001000，而可正可负的整数则是按照补码存储起来的，小数又是按照不同的方式存储起来的。下面这段代码的作用是将整个存储器里的数据“倒”出来，看看各变量在存储器里的数据到底是什么（显示为十六进制数）？

```
/*
读出-56在存储器中的数据
*/
volatile unsigned char val1;
volatile signed char val2;
volatile int val3;
volatile float val4;

int main()
{
```

```
  val1 = 17;
12  val2 = -56;
  val3 = 51;
14  val4 = 3.2;
  UCSR0B = (1 << TXEN0);
16  UBRR0L = 103;
  memory_dump();
18  while (1);
}
20
void memory_dump()
22 {
  uint16_t address = 0x0100; //SRAM起始地址
24  uint8_t byte_at_address, new_line;
  new_line = 1;
26  while (address <= 0x08FF) //0x08FF为SRAM结束地址
  {
28    byte_at_address = *(byte *)address;
    if (((byte_at_address >> 4) & 0x0F) > 9)
30      UDR0 = 55 + (byte_at_address >> 4 & 0x0F);
    else
32      UDR0 = 48 + ((byte_at_address >> 4) & 0x0F);
    while ( !( UCSR0A & (1 << UDRE0)) );
34    if ((byte_at_address & 0x0F) > 9)
    UDR0 = 55 + (byte_at_address & 0x0F);
36    else UDR0 = 48 + (byte_at_address & 0x0F);
    while ( !( UCSR0A & (1 << UDRE0)) );
38    UDR0 = 0x20;
    while ( !( UCSR0A & (1 << UDRE0)) );
40    if (new_line == 64) //每行显示64个数据
    {
42      new_line = 0;
      UDR0 = 0x0A;
44      while ( !( UCSR0A & (1 << UDRE0)) );
      UDR0 = 0x0D;
46      while ( !( UCSR0A & (1 << UDRE0)) );
    }
48    address ++;
    new_line ++;
50  }
}
```

打印结果为：CD CC 4C 40 33 00 C8 11 ...；17 的十六进制数为 0x11,51 的十六进制数为 0x0033（因为定义为 int 型，占 2 个字节）,0xC8 是-56 的存储结果，0x404CCCCD 是 3.2 存储起来的结果。“带符号的数和小数是如何存储起来”作为大家课下的研究内容。

1.6 Arduino 基础之运算符

大家很熟悉“+、-、×、÷”这些运算符号，Arduino 语言中运算符的面更广，有以下大类：

- 赋值运算符
- 算术运算符
- 自加、自减运算符
- 比较运算符
- 逻辑运算符
- 复合运算符
- 位运算符
- 杂项运算符

这里我们只是学习下位运算符，其余的留给大家自学了，位运算符当然对数据的某（些）位进行运算的运算符，包括1.5：

位运算符	符号	作用
按位与	&	返回两个二进制数按位进行与运算后的结果
按位或	|	返回两个二进制数按位进行或运算后的结果
按位取反	~	返回一个二进制数按位取反后的结果
异或	∧	返回两个二进制数按位进行异或运算后的结果
左移	$<< n$	返回二进制数向左移动 n 位后的结果，空出来的位为 0
右移	$>> n$	返回二进制数向右移动 n 位后的结果，空出来的位为 0

表 1.5

- & 运算符
  按位与，其作用是将前后两个数据 (二进制数) 的对应位分别进行“与”运算，比如：

$$B00001110 \& B01001100 = B00001100 = 12$$

- | 运算符
  按位或，其作用是将前后两个数据 (二进制数) 的对应位分别进行“或”运算，比如：

$$B00001110 | B01001100 = B01001110 = 78$$

- ∧ 运算符
  异或，其作用是将前后两个数据 (二进制数) 的对应位分别进行“异或”运算（只要一个为 1，一个为 0，则“异或”运算结果为 1，否则为 0），比如：

$$B00001110 B01001100 = B01000010 = 66$$

- ~ 运算符
  按位取反，不是针对两个数据的运算符，”$\sim a$” 表示将 a 的每一位取反后的结果，比如：a = 14(B00001110)。则

$$\sim a = B11110001 = 241$$

- ”$<< n$” 和”$>> n$” 运算符
  移位运算符，将数据的各位向左或向右移动 n 位，移位后空出的位全部为 0。

1.6.1 位运算的常见用途

- 读出某一位的值
  读出某一位的值的思路很多，比如：读出 a = 14 中 bit3 位的值（bit0 为第一位）。我们可以拿 a 和 8（B00001000）进行按位与运算，如果 a 中 bit3 的值为 1，则 a&8=8；如果 a 中 bit3 的值为 0（1），则 a&8=0（8），因此按位与运算的结果是否为 0 可以作为判断 a 中 bit3 位的值的依据，比如：

```
#define num 9
unsigned char bit3;
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  bit3 = 8 & num; //将8和num按位与
  if (bit3 != 0)
  {
    Serial.print("bit3是");
    Serial.println(1);
    delay(1000);
  }
  else
  {
    Serial.print("bit3是");
    Serial.println(0);
    delay(1000);
  }
  while (1);
}
```

打印结果:bit3 是 1
解析：9 的二进制数是 B00001001，bit3 位的确为 1。
也可以将高位上的数据移到 LSB 位上和 1 去进行与运算，比如：

```
const unsigned char val = 60;
void setup()
{
  Serial.begin(9600);
  for (int i = 0; i < 8; i++)
  {
    Serial.print("bit");
    Serial.print(i);
    Serial.print("的值为：");
    Serial.println((val >> i) & 1);
  }
}
```

打印结果如图1.11所示。
解析：60 的二进制数为 B00111100。
当然也可以不移动数据，左移参与与运算的 1，比如：

```
const unsigned char val = 72;
```

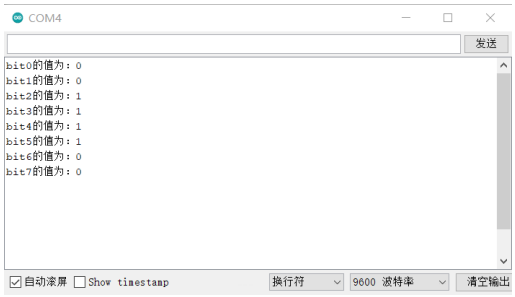


图 1.11

```
char bitVal;
void setup()
{
  Serial.begin(9600);
  Serial.print(val & (1 << 3));
}
void loop()
{
}
```

打印结果：8

解析：1«3 的结果为 B00001000，显然

$$val\&(1 << 3) = B00001000 = 8$$

- 给一位或多位写 1

将某一位置 1，将数据中的该位与 1 做或运算，比如：

```
byte val1 = B00101100;
val1 = (val1|B00000010); //将val1中的bit1位置1
```

也可以用移位运算符

```
val1 = val1|(1<<1);
```

用同样的方法，我们还可以同时修改几位的值，比如将 val1 的 bit1，bit3 和 bit7 位同时置 1。

```
val1 = val1|(1<<1)|(1<<3)|(1<<7);
```

- 给一位或多位写 0

```
unsigned char val1 = B11111111;
void setup()
{
  Serial.begin(9600);
  val1 = val1 & (~1 << 2);
  val1 = val1 & (~((1 << 1) | (1 << 3) | (1 << 7)));
  Serial.print(val1, BIN); //打印二进制数
}
void loop()
{
}
```

打印结果：1110000。

解析：先进行按位或运算，再按位取反得到 B10001010，然后再和 val1 进行与运算。

- 取出一个字节的数据

取出数据中的一位，我们用一位 1 去做与运算，如果取出一个字节的数据就用 8 位 1 去做与运算，比如：

```
int val = 1035;
```

val 变量定义成 int 型，因此占两个字节，二进制数为

B00010000 00110101

其中高字节为 B00010000，低字节为 B00110101。取出低字节的代码为：

```
val&0xff;
```

为了取出高字节数据，我们可以将高字节用移位的方式移到低字节位置，然后用上面的方法取出高字节，代码为：

```
(val >> 8) & 0xff;
```

```
1  int val = 1035;
   int lsbByte, msbByte;
3
   void setup()
5  {
     Serial.begin(9600);
7    lsbByte = val & 0xff;
     msbByte = (val >> 8) & 0xff;
9    Serial.print("1035的低字节为：");
     Serial.println(lsbByte);
11   Serial.print("1035的高字节为：");
     Serial.println(msbByte);
13 }

15 void loop()
   {
17 }
```

打印结果如图1.12所示。

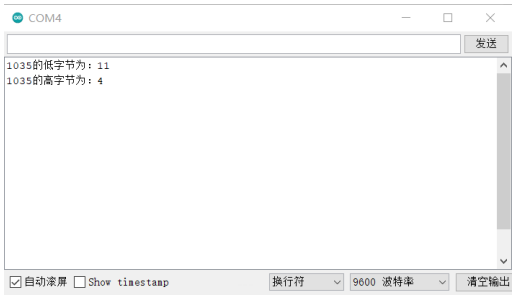


图 1.12

读出的低字节为 11，高字节为 4，显然 $4 \times 256 + 11 = 1035$。

- 将两个数据合并
  比如 B10010011 和 B00011010 分别为某个数据的高字节和低字节，合并成一个数据的方法是：将高字节左移 8 位，然后和低字节进行或运算。

  ```
  highByte = B00111001;
  lowByte = B10100010;
  int val = (highByte<<8)|lowByte;
  ```

  合并后，数据所占的字节数发生改变了，因此要注意匹配变量的类型。
  注意：因此 a»2 只是取出变量 a 的值并右移两位，并没有改变 a 在存储器中的值，但如果是 a=a»2，则 a 的值改变了。

```
1 int val1 = B11111100, val2 = B11111100, val3;

3 void setup()
  {
5   Serial.begin(9600);
    val1 = val1 >> 2;
7   val3 = val2 >> 2;
    Serial.println(val1);
9   Serial.println(val2);
  }
11
  void loop()
13 {
  }
```

  打印结果：
  63
  252
  val1 的值变为 B0011111111=63，而 val2 的值并没有改变。

1.7 Arduino 基础之控制语句

程序运行时，有时需要根据条件来决定程序运行的方向，控制语句就起到这样的作用，使用控制语句时最好搭配流程图，即反映程序执行流程的图形，流程图有三种基本形式：顺序结构 (图1.13)、选择结构 (图1.14) 和循环结构 (图1.15)。
顺序结构即按照代码的先后依次执行，选择结构根据条件是否满足分别执行不同的代码，循环结构则根据条件是否满足决定是否重复执行某段代码。

1.7.1 if 语句

if 语句为条件判断语句，if 表示“如果”。if 语句为程序的运行生成两个可能的方向，条件满足时程序运行的方向和条件不满足时程序运行的方向。

if 语句的基本结构

最基本的 if 语句的结构为：

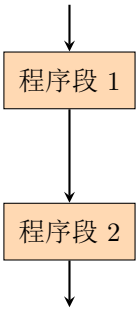


图 1.13: 顺序结构

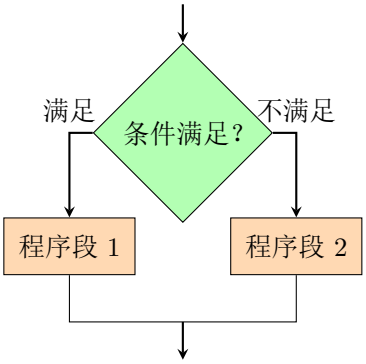


图 1.14: 选择结构

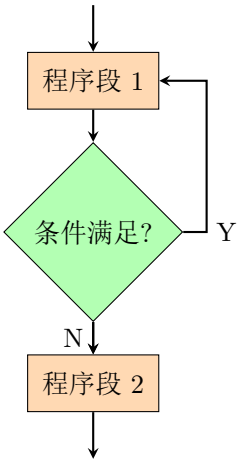


图 1.15: 循环结构

```
if(条件)
{
  程序段（条件满足时执行的代码）;
}
```

图1.16为 if 语句的流程图。

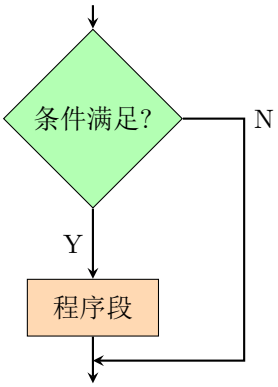


图 1.16: 基本 if 语句的流程图

若表达式成立则执行程序段，否则跳过程序段执行 if 语句下面的代码。比如显示两个数中较大的那个数。

```
/*
  打印两个数中较大的那个
*/
const int val1 = 13, val2 = 8; //定义待比较的整型常量val1,val2

void setup()
{
  Serial.begin(9600); //开启串口通信
  Serial.print("较大的数是:");
  if (val1 >= val2) //如果val1>=val2，打印val1的值
  {
    Serial.println(val1);
  }
   if (val1 < val2) //如果val1<val2，打印val2的值
```

```
  {
16    Serial.println(val2);
  }
18 }

20 void loop()
  {
22 }
```

打印结果：较大的数是：13
下面几种 if 语句的衍生结构也是经常见到的。

if 语句的衍生结构 1

语句结构为

```
if(表达式)
{
  程序段1;
}
else
{
  程序段2;
}
```

图1.17为流程图。

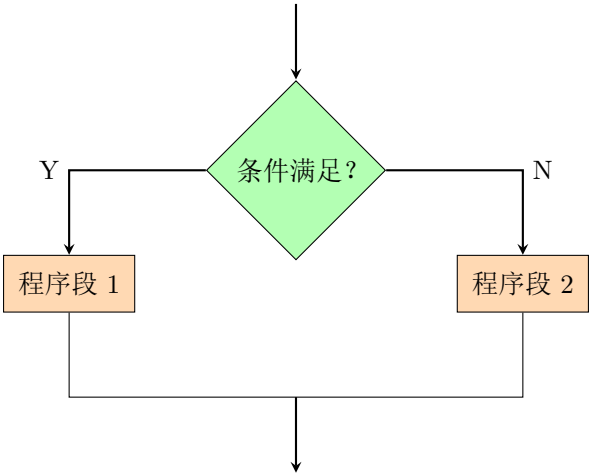


图 1.17: if...else... 语句

if...else... 结构给出了条件满足和不满足时，程序分别执行的代码。

if 语句变形结构 2

语句结构为

```
if(表达式1)
{
```

```
  程序段1;
}
else  if(表达式2)
{
  程序段2;
}
else  if(表达式3)
{
  程序段3;
}
...
else
{
  程序段n;
}
```

if...else if...else... 语句的流程图如图1.21所示，以三个条件为例。

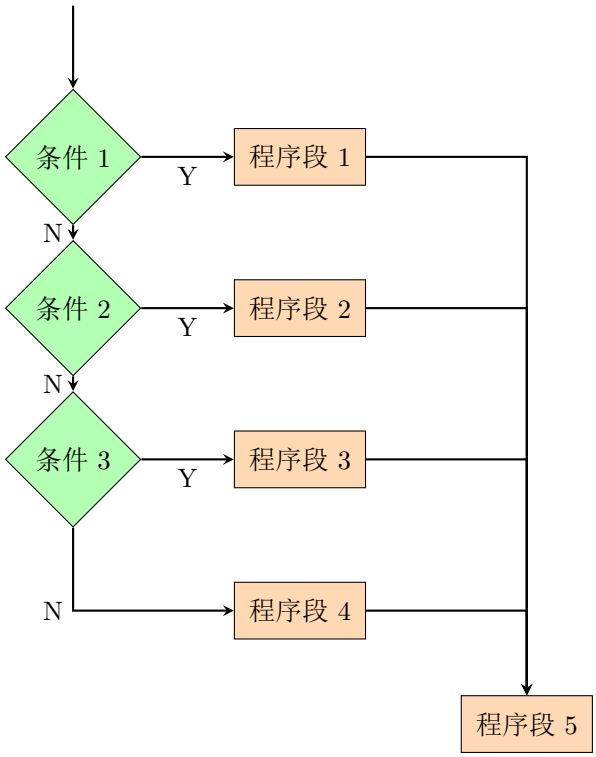


图 1.18: if...else if...else... 语句

1.7.2 for 语句

for 语句实现特定次数的循环，代码结构为：

```
for(表达式1; 表达式2; 表达式3)
{
  程序段;
}
```

for 语句需要定义一个循环变量来控制循环的次数，比如：int i，让循环变量从一个值（初值）按某种方式（比如每次增加 1）逐步变化到另一个值（终值）。表达式 1 给出循环变量的初值，表达式 2 给出循环结束的条件，表达式 3 给出一次循环完成后，如何改变循环变量的值，i++ 表示 i 增加 1，i- -表示 i 减少 1。
比如求 1 到 100 的整数的和，我们可以用 for 循环实现，每次进入循环后，将上一次得到的和与下一个整数加起来，下一个整数可以借用循环控制变量来获得。
代码

```
int sum = 0; //记录累积和

void setup()
{
  Serial.begin(9600);
  for (int num = 1; num <= 10000; num++)
  {
    sum += num;
  }
  Serial.println(sum);
  delay(1000);
}

void loop()
{
}
```

打印结果：5050。

1.7.3 while 语句

while 语句的流程图如图1.19所示。

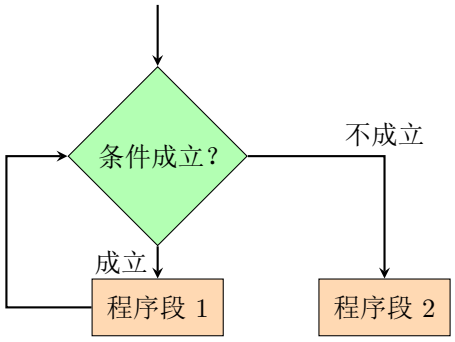


图 1.19: while 语句

代码结构为：

```
while(表达式)
{
  语句;
}
```

表达式成立，值为 1，语句执行；否则不执行。
while 循环也可以用具体的值来控制，即

```
while(value)
{
  代码;//value的值不为0时，执行的代码
}
```

分析下面代码执行的结果。

```
unsigned char i = 0;
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  while (0)
  {
    Serial.print("Jason Gao");
  }
  while (i < 10)
  {
    Serial.print("amazing!");
    Serial.print(i);
    Serial.println("次");
    i = i + 1;
  }
  while (1);
  while (4 % 2 == 0 )
    Serial.print("number is an even number");
}
```

1.7.4 continue 语句

continue 语句的流程图如图1.20所示

continue 语句要在循环体中使用，当执行到 continue 语句时，重新开始循环，循环体中 continue 语句后的代码不会被执行。

例如：

```
int a = 2, b = 5;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  for (int i = 0; i < 5; i++)
```

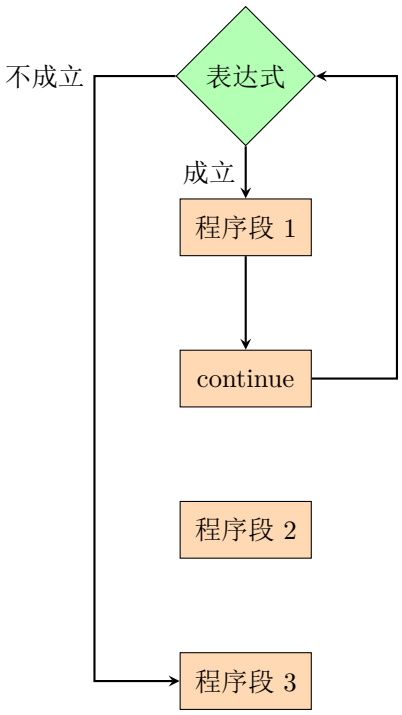


图 1.20: continue 语句的流程图

```
    {
12      Serial.print("a=");
        Serial.println(a);
14      continue; //让我们开始下一次循环吧
        Serial.print("b=␣");
16      Serial.println(b);
    }
18  while (1);
}
```

打印结果如图1.21所示。

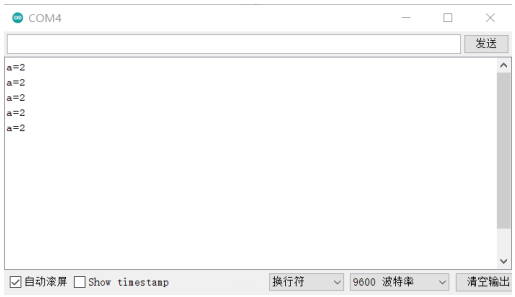


图 1.21

continue1.png 当执行到 continue 时，剩下的代码被忽视，总共循环了 5 次，说明跳出当前循环的时候，i++ 被执行了，这和后面将要学习的 return 语句有所区别。

1.7.5 return 语句

return 语句的流程如图 1.22所示。

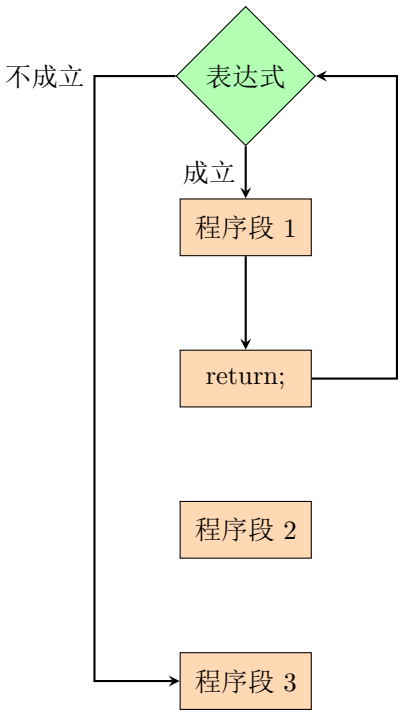


图 1.22: return 语句的流程图

return 的含义是返回，有两层含义：

- 跳过 return 语句后的代码
  用在循环中，表示跳过 return 语句后面的代码，重新回到循环的初始位置，比如：

```
/*
循环语句中，return语句的作用
*/
int a = 2, b = 5;
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.print("a=");
  Serial.println(a);
  return; //循环体中，return语句后面的代码会被跳过
  Serial.print("b=");
  Serial.println(b);
}
```

  打印结果如图1.23所示。
  return 后面的语句被忽略了，因此不断打印 a=2。
- 函数中用于返回一个值
  参看函数部分。

图 1.23: 循环语句中使用 return 语句

1.8 Arduino 基础之函数

函数是能实现一定功能的一段代码，比如：延时函数 delay()，要延时 1 秒，我们只要调用这个函数，并传递参数 (传递到 () 中的值) 给这个函数即可，有些函数执行后，会有一个值留下来，称为函数的返回值。
根据调用函数时是否需要传递参数、是否有返回值，可以把函数分成以下几类：

- 无参数，无返回值；
- 无参数，有返回值；
- 有参数，无返回值；
- 有参数，有返回值。

定义函数的格式为：

```
返回值类型 函数名 (参数类型 参数1,参数类型 参数2,...)
{
  函数体;//调用函数时运行的代码
}
```

比如我们定义一个计算圆面积的函数 circArea()，参数为圆的半径 r，float 型，返回圆的面积 A，float 型，代码如下：

```
void setup()
{
  Serial.begin(9600);
  Serial.print(circArea(3));
}

void loop()
{
}

float circArea(float r)
{
  float A = 3.14159*pow(r,2); //面积的计算公式
  return A; //将计算结果返回
```

}

函数中的变量为局部变量，一旦函数执行完，局部变量就会被销毁，因此需要用 return 语句将变量的值传递出去，传递出去的值找谁要呢？就是函数本身。

1.9 Arduino 基础之变量的作用域

变量除了有类型上的区别，在不同位置定义的变量也有不同的作用范围，分成局部变量和全局变量。

1.9.1 局部变量

在函数中，比如 setup()、loop() 或者自定义的函数中声明的变量是局部变量，它们只能在该函数中使用，例如：

```
void setup()
{
  int x =10; //x在setup()函数中定义，为局部变量
  Serial.begin(9600);
}
void loop()
{
  Serial.println(x); //在loop()函数中调用setup()函数中定义的变量x
}
```

编译代码时出错，如图1.24。

图 1.24

x 定义在 setup() 函数内，因此只能在 setup() 函数中调用，一旦 setup() 函数执行完，该变量就不复存在了，因此在 loop() 函数中调用 x 就会出现变量未被定义的错误。

1.9.2 全局变量

全局变量在所有函数之外定义，可以在定义之后的任何位置处调用，比如：

```
int x=10; //x为全局变量，可以在任何位置调用

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    Serial.println(x);
}
```

打印结果：10。

第二章 Arduino 硬件基础

这里我们不去学习芯片的整体结构，只是围绕如何读写寄存器展开，寄存器其实就是高速存储器，可以存储硬件的配置参数，要发送或接收到的数据等？以串口通信来说吧，之前我们是调用 Serial 库函数来实现的，在芯片内部有许多与串口通信有关的寄存器，也可以直接写寄存器实现：配置通信的波特率，发送的数据包的格式，写要发送的数据，读出接收到的数据。

2.1 输入、输出接口

Arduino 开发板上的 I/O 口即可以向外发送数据，也可以接收外部传来的数据，几个问题：

- 如何设置成 OUTPUT 接口或 INPUT 接口？
- 设置成 OUTPUT 接口的话，在哪里写要发送的电平信号？
- 设置成 INPUT 接口的话，从哪里读取出接收到的电平信号？

这些都是通过操作寄存器来实现的。

图2.1为 Atmega 328p 的引脚说明图。

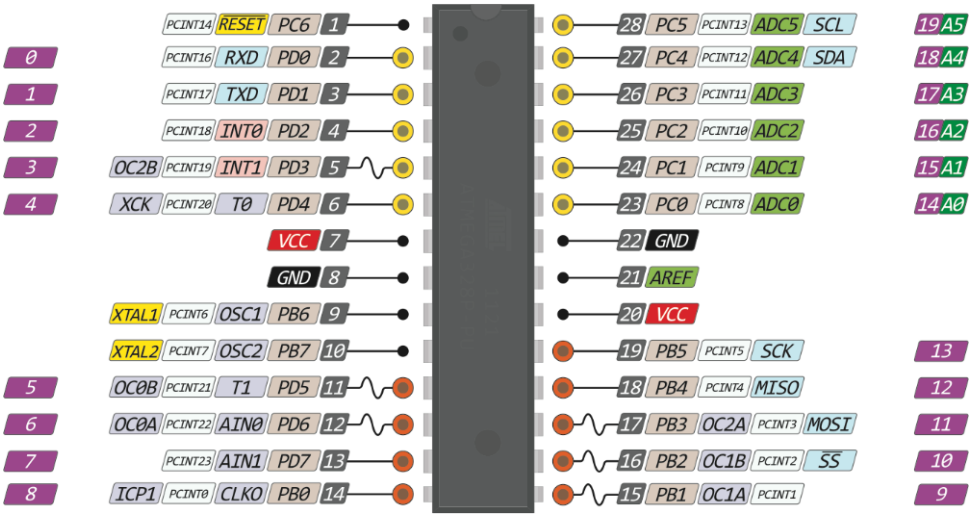


图 2.1

注意图中标 PB*，PC* 和 PD* 的这些引脚，都是芯片的输入输出引脚，其中的 PB6,PB7,PC6 这三个因为另作他用，因此开发板上标出来的 I/O 口就只有 20 个了，Atmega328p 的 I/O 口分成 B、C 和 D 三组。

- B 组: pin8 到 D13，OSC1 和 OSC2(接外部晶振了);
- C 组: A0 到 A5 和 Reset[1];
- D 组: pin0 到 pin7。

每一组引脚都有三个寄存器与之对应：DDRx，PORTx 和 PINx（x 代表 B、C 或 D），图2.2为 B 组引脚所关联的三个寄存器。

[1]如果禁用 Reset 功能的话，Reset 引脚可以作数字输入/出引脚使用

PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PINB – The Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

图 2.2

DDRx

方向寄存器，设置某个或某些引脚为输入或输出，写 1（0）表示输出（入），比如：设定 D13 为输出，根据图2.1，则需要设置 DDRB 中 DDB5（bit5）位为 1。

DDRB |= (1 << 5);
或
DDRB |= (1 << DDB5);

设定 D13 为输入，则需要设置 DDRB 中 DDB5（bit5）位为 0。

DDRB &= ~(1 << DDB5);

相当于 pinMode(13,OUTPUT)。
每个寄存器和寄存器中的每一位都有名称，寄存器的名称相当于变量名，寄存器中的位名称相当于常量名，因此可以直接用。

PORTx

方输出寄存器，当某个或某些引脚被设置为输出时，输出的电平需要在 PORTx 中的对应位设置（1 为高电平，0 为低电平），比如设定 D13 为输出且输出高电平的代码为：

DDRB |= (1 << DDB5); //设置D13为输出
PORTB |= (1<<PORTB5); //D13输出高电平

如果引脚被设置为输入，则 PORTx 的值为 1 表示启用该引脚的上拉电阻（即该引脚被悬空时默认输入高电平），0 表示不启用上拉电阻。

PINx

输入寄存器，在某个或某些引脚被设置为数字输入时，输入的电平存储在 PINx 中的对应位，输入高电平，对应位置 1，否则置 0，比如设定 D13 为输入，读取出 PINB 中 PINB5 位的值即输入的电平。

```
void setup()
{
```

```
  Serial.begin(9600);
4  DDRB &= ~(1 << DDB5); //设置D13为输入
}
6
void loop()
8 {
  int pinValue = (PINB &= (1<<PINB5)); //读取D13引脚的电平
10  if(pinValue == 32)//如果PINB5位为1
  {
12    Serial.println(1);
  }
14  else
    Serial.println(0);
16 }
```

连接 D13 和 5V 引脚，打印 1，连接 D13 和 GND 引脚，打印 0。

2.2 脉冲和时序

脉冲和时序是通信中非常重要的两个概念，

2.2.1 脉冲

脉冲其实就是一系列高低电平间隔着的信号，在数据的传送过程中，脉冲信号起着“指挥棒”的作用，比如 A 将篮子里的鸡蛋取出递给 B，并放在另一个篮子里，为了整个过程能顺利运作，我们需要 C 来发号施令，C 起的就是脉冲信号的作用。
脉冲信号可以由硬件产生，也可以通过代码产生。Atmega328p 内部有三个定时/计数器，可以硬件产生多种类型的脉冲信号，软件方式的话，比如将 I/O 口设置为输出模式，依次输出高低电平。

图 2.3

2.2.2 时序

简单而言，时序就是什么时候做什么事情的标志图，“电子秤”项目中，我们要使用 HX711 模块（如图??），该模块输出端有两个引脚 DT 和 SCK。
其中 DT 引脚为数据输出引脚，SCK 为脉冲信号输入引脚，假设 DT，SCK 分别和开发板的 D3,D2 相连接，D2 输出脉冲信号给模块，然后数据从模块的 DT 引脚一位一位地“吐”出来，具体的时序如图2.4。

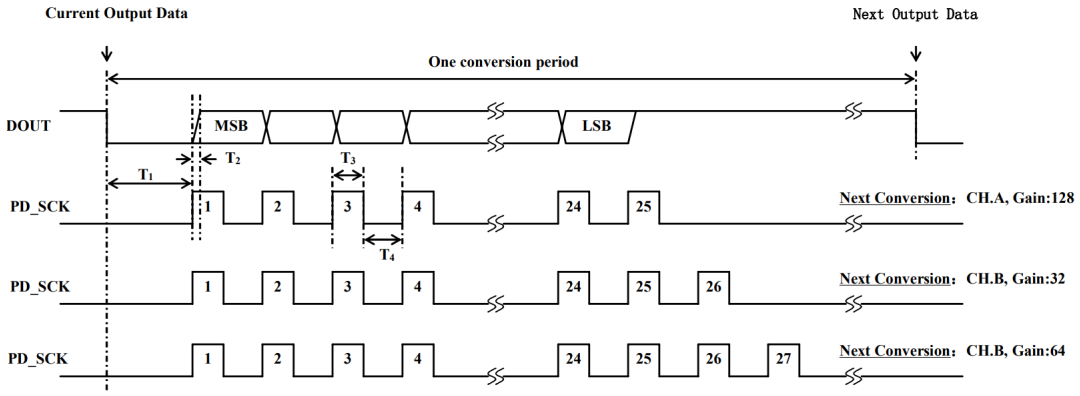


图 2.4

整个时序图分成 3 个阶段：起始，传送中和结束。

当我们需要 HX711 读取数据时，是不是应该先告诉模块，从图中可以看出告诉的方法就是：给 DT 引脚输出一个高电平，SCK 输出一个低电平。

模块收到读取数据的请求后不是马上就能提供数据，总得要准备一下，此时接收端就只能耐心的等待，什么时候可以接收数据只能等模块的通知了，通知的方法就是模块将 DT 引脚拉低，因此开发板发送起始信号后需要马上将 D3 引脚设置为输入模式来监测 DT 引脚的电平变化。

模块将 DT 引脚拉低只是告诉开发板，数据准备好了，你来取吧！为了使模块“递”数据和开发板“接”数据能顺利的对接，此时就需要一个“指挥棒”，即脉冲信号。

从图中可以看出：脉冲信号发送到 SCK 引脚，具体的过程是，先将 SCK 引脚拉高，此时不要急着去读 DT 引脚，因为模块把数据中的一位从内部的寄存器里取出送到 DT 引脚上总需要一点时间吧。待我们把 DT 引脚拉低的时候再去读，如此反复 24 次就可以将 24 位数据全部读出了。

从图上还可以看出拉高再拉低 24 次后，还需要再拉高拉低 1 到 3 次才结束一次数据的传输，这是怎么回事呢，这就需要仔细读 HX711 的数据手册了，模块有两组输入信号输入，A 组和 B 组，输入的信号在内部放大的时候有 3 种增益可选，选择哪组？哪种增益？就通过读取完数据后继续发送的脉冲个数来定。

第三章 电子秤的制作

我们将制作一台低成本的电子秤，秤的精度取决于好的算法的加持，这一版本的电子秤功能非常单一，称量并将结果显示在 1602 液晶屏上，主要是巩固 Arduino 开发中的软、硬件知识，随着学习的深入，我们会不断地对电子秤进行改进，实现一些炫酷的功能，比如无线传输数据、甚至通过物联网实现远程的数据传输等。

3.1 外形设计

进行外形设计的时候，需要确定各模块的外形尺寸并制作出 3D 模型，使用的模块包括：

- Arduino UNO 开发板
- 1602 显示屏
- 称重传感器
- HX711 模块
- 按键
- 开关
- 18650 电池盒（2 节，选用）

各模块的尺寸可以参考淘宝上的商品介绍，但有些模块只是给出了平面图形的尺寸，没有考虑厚度，实际安装的时候，为了使各模块能更加紧凑地组合在一起，我们还是需要进行实际的测量，并制作出模块的 3D 模型和电子秤的框架模型。图3.1为整个电子秤的 3D 模型，放入了各个模块的 3D 模型便于确定内部空间是否合适，图3.2为只显示了轮廓的 3D 模型，务必在 3D 设计软件中向各个方向转动模型检查是否有干涉的地方。

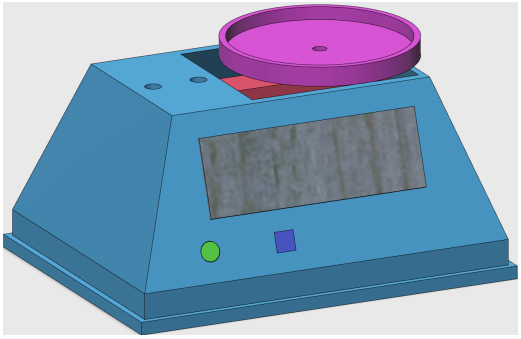


图 3.1

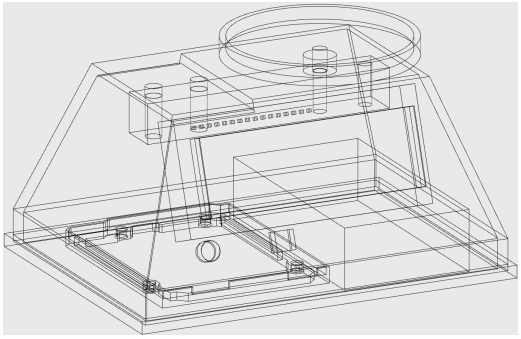


图 3.2

3.2 安装和连接

各模块的连接如图3.3所示，我们的作品中使用了一个按键，一端接 GND，另一端接 D8 引脚 (也可以用别的 I/O 口)。

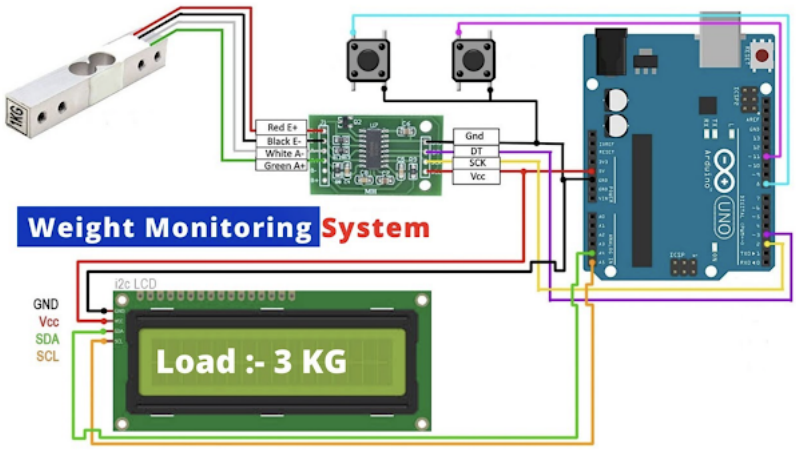


图 3.3: from:https://electronoobs.com/eng_arduino_tut115_sch1.php

将各模块安装在电子秤的框架上，然后确定各连接线的长度，接下来制作带端子的连接线（连接线用硅胶线制作，还需要用到如图3.4所示的端子和胶壳）。

图 3.4

注：有些连接可以不需要端子，直接焊接，需不需要端子由大家自己决定。

3.3 到写代码的时间了

3.3.1 1602 显示模块

单独的 1602 模块共有 16 个引脚，使用比较复杂，也不是我们这个项目的重点，为了简化该模块的使用，

- 使用带 I2C 转接板的 1602 模块
  只需要连接 4 个引脚，I2C 是一种数据传输协议，通过两个接口实现数据的传输，SDA（传输数据）和 SCL（传输脉冲信号）。Arduino UNO 上 I2C 通信的两个接口分别是 A4 和 A5。
- 调用库函数
  使用的库函数为：LiquidCrystal_I2C.h，这个库函数不是 Arduino IDE 自带的库函数，因此需要下载并将解压缩后的文件夹复制到 Arduino IDE 安装目录的 libraries 文件夹下。

3.3.2 初次认识库函数

库函数中的对象构造函数

往往一个库函数包含很多的成员函数，不需要搞清楚每个成员函数的具体用法，但其中的构造函数一定是要掌握的，构造函数和库的名称相同，下面列出 LiquidCrystal_I2C.h 库中的一些成员函数：

- LiquidCrystal_I2C

- begin();
- clear();
- acklight();
- print();
- setCursor(*,*);
- write();

很容易辨别出来 LiquidCrystal_I2C 就是构造函数，构造函数的作用是构造一个或多个对象，构造出的对象可以独立地调用其它成员函数，如何区分不同的对象呢？在构造对象的时候，需要传递一些参数，就是通过这些参数来区分的。比如使用 LiquidCrystal_I2C 来构造对象的格式为

LiquidCrystal_I2C 对象名(参数1，参数2，参数3);

其中：

- 参数 1 为 I2C 设备的地址
- 参数 2 为每行显示的半角字符数
- 参数 3 为需要显示的行数

参数 2 和 3 好说，1602 的含义就是显示 2 行，每行 16 个半角字符，至于设备的地址，可以从模块的介绍中找到，也可以运行扫描 I2C 设备地址的程序，I2C 设备的地址是可以更改的。
仔细观察转接板，如图3.5，将 A0，A1，A2 这三组焊点短接或断开，可以有 7 种组合，分别对应不同的地址。

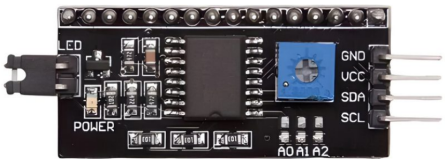


图 3.5

图3.6是从网上截取的关于 I2C 转换模块地址的描述。

I2C Address

The default address is basically 0x27, in a few cases it may be 0x3F.

Taking the default address of 0x27 as an example, the device address can be modified by shorting the A0/A1/A2 pads; in the default state, A0/A1/A2 is 1, and if the pad is shorted, A0/A1/A2 is 0.

Slave Address

0	0	1	0	0	A2	A1	A0

0	0	1	0	0	1	1	1	0x27
0	0	1	0	0	1	1	0	0x26
0	0	1	0	0	1	0	1	0x25
0	0	1	0	0	0	1	1	0x23
					.....			
0	0	1	0	0	0	0	0	0x20

图 3.6

构造的对象调用其它成员函数

假设 A,B 两块带 I2C 转接板的 1602 显示屏的地址分别为 0x20 和 0x21，构造对象时

LiquidCrystal_I2C 1602A(0x20,16,2);
LiquidCrystal_I2C 1602B(0x21,16,2);

当对象 1602A 调用成员函数时将实现显示屏 A 的显示。
对象调用其它成员函数的格式为:

对象名.成员函数名(参数) //成员函数不一定都带参数

由于上面列出的几个成员函数都比较好理解，这里就不一一介绍了，直接通过一个例子就可以掌握了。

```
//在特定的位置显示字符的代码

#include <Wire.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

void setup()
{
  lcd.begin();
}

void loop()
{
  lcd.clear(); //清除原来的显示内容
  lcd.setCursor(0, 0);//设置显示的起始位置（列，行）
  lcd.print("reading:"); //在第一列，第一行显示的内容
  lcd.setCursor(0,1);
  lcd.print(3); //在第一列，第二行显示的内容
  delay(100); //延时的长短根据实际来定
}
```

3.3.3 称重传感器和 HX711 模块的使用

称重传感器的工作原理

称重传感器的结构如图3.7所示。

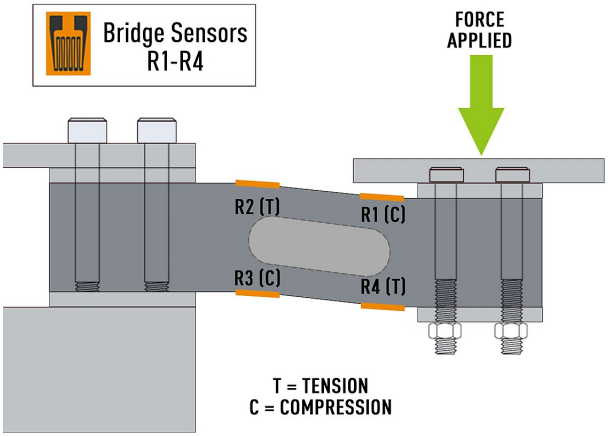


图 3.7

图中的 R1-R4 为 4 个应变片，应变片在受到拉伸或者压缩时，其电阻值发生变化，压缩时电阻变小，拉伸时电阻变大，其原理就是导体电阻的计算公式

$$R = \frac{\rho L}{S}$$

沿着箭头方向施加压力（该箭头方向在称重传感器上标示出来了），应变片 R1 和 R3 处于 compression 状态（应变片变得粗短，阻值减小），R2 和 R4 处于 tension 状态（应变片变得细长，阻值增大）。
4 个应变片的连接如图??所示。

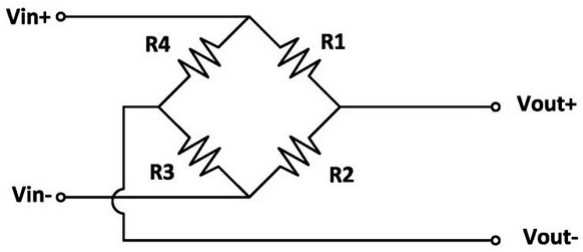


图 3.8

图中的 Vin+ 和 Vin-对应模块的红线和黑线，负责给应变片供电，接下来需要一点物理知识来分析 Vout+，Vout-分别对应什么颜色的线了，

$$V_3 = \frac{R3 \times \Delta V}{R_3 + R_4}, V_2 = \frac{R2 \times \Delta V}{R_1 + R_2}$$

向下挤压时，R_1R_3 减小，R_2R_4 增大，因此 $V_2 > V_3$，不难理解 Vout+ 和 Vout-了。
仔细观察图3.3，可以发现绿色的线和连接 A+，因此图??中的 Vout+ 对应称重传感器的绿色线。

HX711 的使用

Vout+ 和 Vout-之间的电压是非常微弱的，因此需要进行放大，并转换为数字信号输出，这两个任务都是由 HX711 来完成的，由于输出的数字信号与被称质量呈线性关系，因此我们只要放两个已知质量的物体上去，根据读数，就可以建立起读数和质量之间的关系，假设质量为 $m_1(m_2)$ 时输出的数字信号为 $x_1(x_2)$，则

$$x = x_1 + \frac{x_2 - x_1}{m_2 - m_1}(m - m_1)$$

$$m = m_1 + \frac{x - x_1}{x_2 - x_1}(m_2 - m_1)$$

对于 HX711 的使用，这里我们就不使用库函数了，为了巩固所学的关于寄存器和脉冲信号的知识，仔细阅读数据手册，找出读取数据的时序图（如图2.4），从图中可以看出，读数据时，先给 DT 引脚高电平，SCK 引脚低电平，相当于告诉模块，我要准备读数了，当模块收到这个起始信号后，开始处理数据，一旦数据处理完成，便在 DT 引脚上输出一个低电平，因此我们发出起始信号后，马上监视 DT 引脚上的电平信号，一旦变成低电平，就表示，可以读取数据了，读取数据时，给 SCK 引脚脉冲信号，拉高再拉低，从图上看出，拉低时，数据处于稳定状态，此时读取到可靠的数据，脉冲的个数取决于连接 HX711 时所使用的通道以及增益，比如通道 A，增益 64，则需要 27 个脉冲。
DT 引脚输出的数据是补码，如果是正数，最大值为 01111111 11111111 11111111, 对应的十进制数为 8388607 如果是负数，最大值为 10000000000000000000000，对应的十进制数为-8388608。如果不希望结果是负数的形式，可以将-8388608 到 8388607 这个范围转换到 0(0x000000) 到 1677215(0xFFFFFF) 这个范围。实现的方式就是：

$$val = val \text{^} 0x800000$$

与 0x800000 进行异或运算可以起到这样的作用

```
#define DT 3
#define SCK 2
```

```
void setup()
{
  Serial.begin(9600);
  unsigned long sum = 0,averageVal;
  for(unsigned char i=0;i<10;i++)
{
  sum = sum + dataOut();
}
 averageVal = sum/10;
 Serial.println(averageVal);
}

void loop()
{



  delay(100);
}

unsigned long dataOut(void)
{
  unsigned long val=0;
  pinMode(DT, OUTPUT);
  pinMode(SCK,OUTPUT);
  digitalWrite(DT,HIGH);
  digitalWrite(SCK,LOW);
  pinMode(DT, INPUT);
  while(digitalRead(DT));
  for (unsigned char i=0;i<24;i++)
  {
    digitalWrite(SCK,HIGH);
    val=val<<1;
    digitalWrite(SCK,LOW);
    if(digitalRead(DT))
    val++;
  }
  digitalWrite(SCK,HIGH);
  val=val^0x800000; //输出的是二进制补码
  digitalWrite(SCK,LOW);
  return(val);
}
```

读数如图3.9所示。

图 3.9

3.3.4 将读出的数据转换为质量

输出的数据和质量之间呈线性关系，因此根据对已知质量的两次测量，就可以建立起质量和输出数据之间的关系，假设第一次我们不放任何重物，得到读数 x_1，然后再放一个标准质量 m_2 得到读数 x_2，则

$$m = m_1 + \frac{x - x_1}{x_2 - x_1}(m_2 - m_1)$$

可以简化为

$$m = \frac{m_2(x - x_1)}{x_2 - x_1}$$

其中 $\frac{m_2}{x_2 - x_1}$ 表示输出数据每增加 1 所对应的质量的增加量，用 c 表示，则

$$m = c(x - x_1)$$

因此，为了得到质量，我们需要事先确定出 c 和 x_1 的值。

将下面代码上传给开发板，按照屏幕的提示进行操作，如果没有已知质量的物体，我们可以用 1 元硬币，比如我手头上的硬币的质量为 6.1g，8 枚的质量为 48.8g。

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

#define DT 3
#define SCK 2

unsigned long init_Val;
float  scale_factor;

void setup()
{
  unsigned long sum=0;
```

```
  pinMode(A0,INPUT_PULLUP);
  lcd.begin();
  Serial.begin(9600);
  lcd.clear(); //清除原来的显示内容
  lcd.setCursor(0, 0);//设置显示的起始位置（列，行）
  lcd.print("remove load!"); //在第一列，第一行显示的内容
  for(unsigned char i=0;i<20;i++)
  {
    sum = sum + dataOut();
  }
  init_Val = sum/20;
  sum = 0;
  lcd.clear(); //清除原来的显示内容
  lcd.setCursor(0, 0);//设置显示的起始位置（列，行）
  lcd.print("load then button!"); //在第一列，第一行显示的内容
  while(digitalRead(A0)==1);
  for(unsigned char i=0;i<20;i++)
  {
    sum = sum + dataOut();
  }
  scale_factor = 48.8/(sum/20-init_Val); //如果用别的质量做标准，将48.8改成相应的值，以g为单位
}

//loop里进行任意质量的测量并将结果显示出来。

void loop()
{
  float reading=0;
  unsigned long sum=0;
  for(unsigned char i=0;i<20;i++)
  {
    sum = sum + dataOut();
  }
  reading = (sum/20-init_Val)*scale_factor;
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("reading:");
  lcd.setCursor(0,1);
  lcd.print(reading);
}

unsigned long dataOut(void)
{
  unsigned long val = 0;
  pinMode(DT, OUTPUT);
```

```
  pinMode(SCK, OUTPUT);
  digitalWrite(DT, HIGH);
  digitalWrite(SCK, LOW);
  pinMode(DT, INPUT);
  while (digitalRead(DT));
  for (unsigned char i = 0; i < 24; i++)
  {
    digitalWrite(SCK, HIGH);
    val = val << 1;
    digitalWrite(SCK, LOW);
    if (digitalRead(DT))
      val++;
  }
  digitalWrite(SCK, HIGH);
  val = val ^ 0x800000; //输出的是二进制补码
  digitalWrite(SCK, LOW);
  return (val);
}
```

3.3.5 滤波

所谓滤波指除去干扰，得到更稳定的测量结果，HX711 模块输出的数据受电压的影响明显，提供稳定的电压因此就很重要了，我们可以在模块上 VCC 和 GND 之间连接一个电容器来稳定电压，这也是一种滤波，硬件层面的滤波，不过接下来，我们将通过算法来滤波，前面的代码中，其实我们已经采用了一种算法-求平均值，不过比较简单粗暴，这里我们优化下平均值算法，当 HX711 受到突然的干扰时，往往会使得输出的数据特别大或特别小，因此需要去除掉，为此我们再添加一个滤波函数 Filter()，这个滤波函数的作用，大家可以分析代码得出。

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

#define DT 3
#define SCK 2

unsigned long init_Val;
float  scale_factor;

void setup()
{
  unsigned long sum = 0;
  pinMode(A0, INPUT_PULLUP);
  lcd.begin();
  Serial.begin(9600);
  lcd.clear(); //清除原来的显示内容
  lcd.setCursor(0, 0);//设置显示的起始位置（列，行）
```

```
  lcd.print("remove load!"); //在第一列，第一行显示的内容
  for (unsigned char i = 0; i < 20; i++)
  {
    sum = sum + Filter();
  }
  init_Val = sum / 20;
  sum = 0;
  lcd.clear(); //清除原来的显示内容
  lcd.setCursor(0, 0);//设置显示的起始位置（列，行）
  lcd.print("load then button!"); //在第一列，第一行显示的内容
  while (digitalRead(A0) == 1);
  for (unsigned char i = 0; i < 20; i++)
  {
    sum = sum + Filter();
  }
  scale_factor = 48.8 / (sum / 20 - init_Val);
}

//loop里进行任意质量的测量并将结果显示出来。

void loop()
{
  float reading = 0;
  unsigned long sum = 0;
  for (unsigned char i = 0; i < 20; i++)
  {
    sum = sum + Filter();
  }
  reading = (sum / 20 - init_Val) * scale_factor;
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("reading:");
  lcd.setCursor(0, 1);
  lcd.print(reading);
}

unsigned long dataOut(void)
{
  unsigned long val = 0;
  pinMode(DT, OUTPUT);
  pinMode(SCK, OUTPUT);
  digitalWrite(DT, HIGH);
  digitalWrite(SCK, LOW);
  pinMode(DT, INPUT);
  while (digitalRead(DT));
```

```
  for (unsigned char i = 0; i < 24; i++)
  {
    digitalWrite(SCK, HIGH);
    val = val << 1;
    digitalWrite(SCK, LOW);
    if (digitalRead(DT))
      val++;
  }
  digitalWrite(SCK, HIGH);
  val = val ^ 0x800000; //输出的是二进制补码
  digitalWrite(SCK, LOW);
  return (val);
}

unsigned long Filter() //中位值平均滤波法
{
  unsigned char FILTER_N = 5;
  unsigned long filter_temp;
  unsigned long filter_sum = 0;
  unsigned long filter_buf[FILTER_N];
  for (unsigned char i = 0; i < FILTER_N; i++)
  {
    filter_buf[i] = dataOut();
    delay(1);
  }
  for (unsigned char j = 0; j < FILTER_N - 1; j++)
  {
    for (unsigned char i = 0; i < FILTER_N - 1 - j; i++)
    {
      if (filter_buf[i] > filter_buf[i + 1])
      {
        filter_temp = filter_buf[i];
        filter_buf[i] = filter_buf[i + 1];
        filter_buf[i + 1] = filter_temp;
      }
    }
  }
  for (int i = 1; i < FILTER_N - 1; i++)
    filter_sum += filter_buf[i];
  return filter_sum / (FILTER_N - 2);
}
```

改良滤波算法后的结果更加的稳定和准确了，不过得到结果需要的时间更久了。