# Arduino 语法手册

Arduino 的程序可以划分为三个主要部分：结构、变量（变量与常量）、函数。

## 结构部分

**一、结构**

1.1 setup()

1.2 loop()

**二、结构控制**

2.1 if

2.2 if...else

2.3 for

2.4 switch case

2.5 while

2.6 do... while

2.7 break

2.8 continue

2.9 return

2.10 goto

**三、扩展语法**

3.1 ;（分号）

3.2 {}（花括号）

3.3 //（单行注释）

3.4 /* */（多行注释）

3.5 #define

3.6 #include

**四、算数运算符**

4.1 =（赋值运算符）

4.2 +（加）

4.3 -（减）

4.4 *（乘）

4.5 /（除）

4.6 %（模）

**五、比较运算符**

5.1==（等于）

5.2 !=（不等于）

5.3 <（小于）

5.4 >（大于）

5.5 <=（小于等于）

5.6 >=（大于等于）

**六、布尔运算符**

6.1 &&（与）

6.2 ||（或）

6.3 !（非）

**七、指针运算符**

7.1 * 取消引用运算符

7.2 &引用运算符

**八、位运算符**

8.1 & (bitwise and)

8.2 | (bitwise or)

8.3 ^ (bitwise xor)

8.4 ~ (bitwise not)

8.5 << (bitshift left)

8.6 >> (bitshift right)

**九、复合运算符**

9.1 ++ (increment)

9.2 -- (decrement)

9.3 += (compound addition)

9.4 -= (compound subtraction)

9.5 *= (compound multiplication)

9.6 /= (compound division)

9.6 &= (compound bitwise and)

9.8 |= (compound bitwise or)

## 变量部分

**十、常量**

10.1 HIGH|LOW（引脚电压定义）

10.2 INPUT|OUTPUT（数字引脚（Digital pins）定义）

10.3 true | false（逻辑层定义）

10.4 integer constants（整数常量）

10.5 floating point constants（浮点常量）

**十一、数据类型**

11.1 void

11.2 boolean（布尔）

11.3 char（有号数据类型）

11.4 unsigned char（无符号数据类型）

11.5 byte（无符号数）

11.6 int（整型）

## 函数部分

# 结构部分

## 一、结构

### 1.1 setup()

在 Arduino 中程序运行时将首先调用 setup() 函数。用于初始化变量、设置针脚的输出\输入类型、配置串口、引入类库文件等等。每次 Arduino 上电或重启后，setup 函数只运行一次。

**示例**

```
int buttonPin = 3;
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

void loop()
{
  // ...
}
```

### 1.2 loop()

在 setup() 函数中初始化和定义了变量，然后执行 loop() 函数。顾名思义,该函数在程序运行过程中不断的循环，根据一些反馈,相应改变执行情况。通过该函数动态控制 Arduino 主控板。

**示例**

```
int buttonPin = 3;// setup 中初始化串口和按键针脚.
void setup()
{
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}

// loop 中每次都检查按钮,如果按钮被按下,就发送信息到串口
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');

  delay(1000);
}
```

# 二、结构控制

## 2.1 if

if（条件判断语句）和 ==、!=、<、>（比较运算符）

if 语句与比较运算符一起用于检测某个条件是否达成，如某输入值是否在特定值之上等。if 语句的语法是：

```
if (someVariable > 50)
{
  // 执行某些语句
}
```

本程序测试 someVariable 变量的值是否大于 50。当大于 50 时，执行一些语句。换句话说，只要 if 后面括号里的结果（称之为测试表达式）为真，则执行大括号中的语句（称之为执行语句块）；若为假，则跳过大括号中的语句。 if 语句后的大括号可以省略。若省略大括号，则只有一条语句（以分号结尾）成为执行语句。

```
if (x > 120) digitalWrite(LEDpin, HIGH);
if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}                                          // 以上所有书写方式都正确
```

在小括号里求值的表达式，需要以下操作符：

**比较运算操作符：**

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

**警告：**

注意使用赋值运算符的情况（如 if (x = 10)）。一个“=”表示的是赋值运算符，作用是将 x 的值设为 10（将值 10 放入 x 变量的内存中）。两个“=”表示的是比较运算符（如 if (x == 10)），用于测试 x 和 10 是否相等。后面这个语句只有 x 是 10 时才为真，而前面赋值的那个语句则永远为真。

这是因为 C 语言按以下规则进行运算 if (x=10)：10 赋值给 x（只要非 0 的数赋值的语句，其赋值表达式的值永远为真），因此 x 现在值为 10。此时 if 的测试表达式值为 10，该值永远为真，因为非 0 值永远为真。所以，if (x = 10) 将永远为真，这就不是我们运行 if 所期待的结果。另外，x 被赋值为 10，这也不是我们所期待的结果。

if 的另外一种分支条件控制结构是 if...else 形式。

## 2.2if...else

if/else 是比 if 更为高级的流程控制语句，它可以进行多次条件测试。比如，检测模拟输入的值，当它小于 500 时该执行哪些操作，大于或等于 500 时执行另外的操作。代码如下：

```
if (pinFiveInput < 500)
{
  // 执行 A 操作
}
else
{
  // 执行 B 操作
}
```

else 可以进行额外的 if 检测，所以多个互斥的条件可以同时进行检测。

测试将一个一个进行下去，直到某个测试结果为真，此时该测试相关的执行语句块将被运行，然后程序就跳过剩下的检测，直接执行到 if/else 的下一条语句。当所有检测都为假时，若存在 else 语句块，将执行默认的 else 语句块。

**注意** else if 语句块可以没有 else 语句块。else if 分支语句的数量无限制。

```
if (pinFiveInput < 500)
{
  // 执行 A 操作
}
else if (pinFiveInput >= 1000)
{
  // 执行 B 操作
}
else
{
  // 执行 C 操作
}
```

另外一种进行多种条件分支判断的语句是 switch case 语句。

## 2.3 for

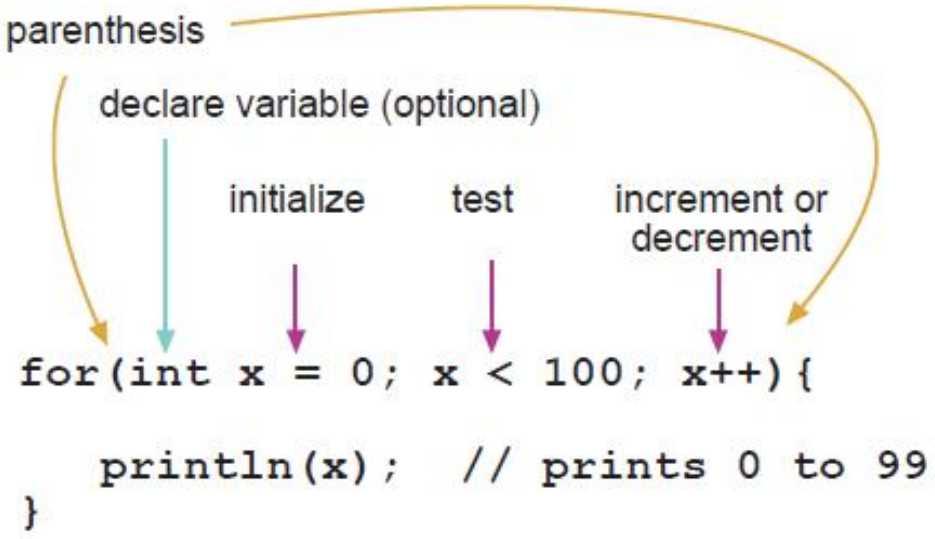
**for 语句**

**描述**

for 语句用于重复执行一段在花括号之内的代码。通常使用一个增量计数器计数并终止循环。for 语句用于重复性的操作非常有效，通常与数组结合起来使用来操作数据、引脚。

for 循环开头有 3 个部分：

```
（初始化;条件;增量计数）{
//语句
}
```




“初始化”只在循环开始执行一次。每次循环，都会检测一次条件；如果条件为真，则执行语句和“增量计数”，之后再检测条件。当条件为假时，循环终止。

**例子**

```
//用 PWM 引脚将 LED 变暗
int PWMpin = 10; //将一个 LED 与 47Ω 电阻串联接在 10 脚

void setup()
{
//无需设置
}

void loop()
{
    for (int i=0; i <= 255; i++)
        {
        analogWrite(PWMpin, i);
        delay(10);
        }
}
```

**编程提示**

C 语言的 for 循环语句比 BASIC 和其他电脑编程语言的 for 语句更灵活。除了分号以外，其他 3 个元素都能省略。同时，初始化，条件，增量计算可以是任何包括无关变量的有效 C 语句，任何 C 数据类型包括 float。这些不寻常的 for 语句可能会解决一些困难的编程问题。

例如，在增量计数中使用乘法可以得到一个等比数列：

```
for(int x = 2; x < 100; x = x * 1.5){
println(x);
}
```

生成：2,3,4,6,9,13,19,28,42,63,94

另一个**例子**，使用 for 循环使 LED 产生渐亮渐灭的效果：

```
void loop()
{
    int x = 1;
    for (int i = 0; i > -1; i = i + x)
      {
        analogWrite(PWMpin, i);
        if (i == 255) x = -1;                    // 在峰值转变方向
      delay(10);
      }
}
```

## 2.4 switch case

switch / case 语句

和 if 语句相同，switch…case 通过程序员设定的在不同条件下执行的代码控制程序的流程。特别地，switch 语句将变量值和 case 语句中设定的值进行比较。当一个 case 语句中的设定值与变量值相同时，这条 case 语句将被执行。

关键字 break 可用于退出 switch 语句，通常每条 case 语句都以 break 结尾。如果没有 break 语句，switch 语句将会一直执行接下来的语句（一直向下）直到遇见一个 break，或者直到 switch 语句结尾。

**例子**

```
switch (var) {
case 1:
  //当 var 等于 1 时，执行一些语句
  break;
case 2
  //当 var 等于 2 时，执行一些语句
  break;
default:
  //如果没有任何匹配，执行 default
  //default 可有可不有
}
```

**语法**

```
switch (var) {
case label:
  // 声明
  break;
case label:
  // 声明
  break;
default:
```

```
  // 声明
}
```

**参数**

var: 用于与下面的 case 中的标签进行比较的变量值

label: 与变量进行比较的值

## 2.5 while

**while 循环**

**描述**

while 循环会无限的循环，直到括号内的判断语句变为假。必须要有能改变判断语句的东西，要不然 while 循环将永远不会结束。这在您的代码表现为一个递增的变量，或一个外部条件，如传感器的返回值。

**语法**

```
while(表达){
    //语句
}
```

**参数**

表达：为真或为假的一个计算结果

**例子**

```
var = 0;
while(var < 200){
//重复一件事 200 遍
var++
}
```

## 2.6 do...while

do…while 循环与 while 循环运行的方式是相近的，不过它的条件判断是在每个循环的最后，所以这个语句至少会被运行一次，然后才被结束。

```
do
{
//语句
}while（测试条件）;
```

**例子**

```
do
{
delay（50）; //等待传感器稳定
X = readSensors（）; //检查传感器取值
}while（X <100）;  //当 x 小于 100 时，继续运行
```

## 2.7 break

break 用于退出 do，for，while 循环，能绕过一般的判断条件。它也能够用于退出 switch 语句。

**例子**

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){          // 超出探测范围
        x = 0;
        break;
    }
    delay(50);
}
```

## 2.8 continue

continue 语句跳过当前循环中剩余的迭代部分（ do，for 或 while ）。它通过检查循环条件表达式，并继续进行任何后续迭代。

**例子**

```
for (x = 0; x < 255; x ++)
{
    if (x > 40 && x < 120){          // 当 x 在 40 与 120 之间时，跳过后面两句，即迭代。
        continue;
}
     digitalWrite(PWMpin, x);
    delay(50);
}
```

## 2.9 return

终止一个函数，如有返回值，将从此函数返回给调用函数。

**语法**

```
return;
return value; // 两种形式均可
```

**参数**

value：任何变量或常量的类型

**例子**

一个比较传感器输入阈值的函数

```
int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;}
```

```
        else{
            return 0;
        }
    }
```

return 关键字可以很方便的测试一段代码，而无需“comment out(注释掉)”大段的可能存在 bug 的代码。

```
    void loop(){
    //写入漂亮的代码来测试这里。
     return;
    //剩下的功能异常的程序
    //return 后的代码永远不会被执行
    }
```

## 2.10 goto

程序将会从程序中已有的标记点开始运行

**语法**

```
label:
goto label;       //从 label 处开始运行
```

**提示**

不要在 C 语言中使用 goto 编程，某些 C 编程作者认为 goto 语句永远是不必要的，但用得好，它可以简化某些特定的程序。许多程序员不同意使用 goto 的原因是， 通过毫无节制地使用 goto 语句，很容易创建一个程序，这种程序拥有不确定的运行流程，因而无法进行调试。

的确在有的实例中 goto 语句可以派上用场，并简化代码。例如在一定的条件用 if 语句来跳出高度嵌入的 for 循环。

**例子**

```
for(byte r = 0; r < 255; r++){
   for(byte g = 255; g > -1; g--){
      for(byte b = 0; b < 255; b++){
         if (analogRead(0) > 250){
            goto bailout;
         }
         //更多的语句...
      }
   }
}
bailout:
```

# 三、扩展语法

## 3.1 ；（分号）

用于表示一句代码的结束。

**例子**

```
int a = 13;
```

**提示**

在每一行忘记使用分号作为结尾，将导致一个编译错误。错误提示可能会清晰的指向缺少分号的那行，也可能不会。如果弹出一个令人费解或看似不合逻辑的编译器错误，第一件事就是在错误附近检查是否缺少分号。

## 3.2{}（花括号）

大括号（也称为“括号”或“大括号”）是 C 编程语言中的一个重要组成部分。它们被用来区分几个不同的结构，下面列出的，有时可能使初学者混乱。

左大括号“{”必须与一个右大括号“}”形成闭合。这是一个常常被称为括号平衡的条件。在 Arduino IDE（集成开发环境）中有一个方便的功能来检查大括号是否平衡。只需选择一个括号，甚至单击紧接括号的插入点，就能知道这个括号的“伴侣括号”。

目前此功能稍微有些错误，因为 IDE 会经常会认为在注释中的括号是不正确的。

对于初学者，以及由 BASIC 语言转向学习 C 语言的程序员，经常不清楚如何使用括号。毕竟，大括号还会在”return 函数”、“endif 条件句”以及“loop 函数”中被使用到。

由于大括号被用在不同的地方，这有一种很好的编程习惯以避免错误：输入一个大括号后，同时也输入另一个大括号以达到平衡。然后在你的括号之间输入回车，然后再插入语句。这样一来，你的括号就不会变得不平衡了。

不平衡的括号常可导致许多错误，比如令人费解的编译器错误，有时很难在一个程序找到这个错误。由于其不同的用法，括号也是一个程序中非常重要的语法，如果括号发生错误，往往会极大地影响了程序的意义。

**大括号中的主要用途**

**功能**

```
void myfunction(datatype argument){
  statements(s)
  }
```

**循环**

```
while (boolean expression)
{
  statement(s)
  }

  do
{
  statement(s)
  }
  while (boolean expression);

for (initialisation; termination condition; incrementing expr)
```

```
{
   statement(s)
   }
```

**条件语句**

```
if (boolean expression)
{
   statement(s)
   }

else if (boolean expression)
{
   statement(s)
   }
else
{
   statement(s)
   }
```

## 3.3 //（单行注释）

Comments（注释）

注释用于提醒自己或他人程序是如何工作的。它们会被编译器忽略掉，也不会传送给处理器，所以它们在 Atmega 芯片上不占用体积。注释的唯一作用就是使你自己理解或帮你回忆你的程序是怎么工作的或提醒他人你的程序是如何工作的。编写注释有两种写法：

**例子**

```
 x = 5;   // 这是一条注释斜杠后面本行内的所有东西是注释
/* 这是多行注释-用于注释一段代码
if (gwb == 0){     // 在多行注释内可使用单行注释
x = 3;                /* 但不允许使用新的多行注释-这是无效的
}
// 别忘了注释的结尾符号-它们是成对出现的!
*/
```

**小提示**

当测试代码的时候，注释掉一段可能有问题的代码是非常有效的方法。这能使这段代码成为注释而保留在程序中，而编译器能忽略它们。这个方法用于寻找问题代码或当编译器提示出错或错误很隐蔽时很有效。

## 3.4 /* */（多行注释）

Comments（注释）

注释用于提醒自己或他人程序是如何工作的。它们会被编译器忽略掉，也不会传送给处理器，所以它们在 Atmega 芯片上不占用体积。注释的唯一作用就是使你自己理解或帮你回忆你的程序是怎么工作的或提醒他人你的程序是如何工作的。编写注释有两种写法：

**例子**

```
 x = 5;   // 这是一条注释斜杠后面本行内的所有东西是注释
/* 这是多行注释-用于注释一段代码
```

```
if (gwb == 0){     // 在多行注释内可使用单行注释
x = 3;                  /* 但不允许使用新的多行注释-这是无效的
}
// 别忘了注释的结尾符号-它们是成对出现的!
*/
```

**小提示**

当测试代码的时候，注释掉一段可能有问题的代码是非常有效的方法。这能使这段代码成为注释而保留在程序中，而编译器能忽略它们。这个方法用于寻找问题代码或当编译器提示出错或错误很隐蔽时很有效。

## 3.5 #define

#define 是一个很有用的 C 语法，它允许程序员在程序编译之前给常量命名。在 Arduino 中，定义的常量不会占用芯片上的任何程序内存空间。在编译时编译器会用事先定义的值来取代这些常量。

然而这样做会产生一些副作用，例如，一个已被定义的常量名已经包含在了其他常量名或者变量名中。在这种情况下，文本将被＃defined 定义的数字或文本所取代。

通常情况下，优先考虑使用 const 关键字替代 #define 来定义常量。

Arduino 拥有和 C 相同的语法规范。

**语法**

#define 常量名常量值注意，＃是必须的。

**例子**

```
#define ledPin 3
//在编译时，编译器将使用数值 3 取代任何用到 ledPin 的地方。
```

**提示**

在#define 声明后不能有分号。如果存在分号，编译器会抛出语义不明的错误，甚至关闭页面。

```
#define ledPin 3; //这是一种错误写法
```

类似的，在#define 声明中包含等号也会产生语义不明的编译错误从而导致关闭页面。

```
#define ledPin = 3 //这是一种错误写法
```

## 3.6 #include

#include 用于调用程序以外的库。这使得程序能够访问大量标准 C 库，也能访问用于 arduino 的库。 AVR C 库（Arduino 基于 AVR 标准语法）语法手册请点击这里。注意#include 和#define 一样，不能在结尾加分号，如果你加了分号编译器将会报错。

**例子**

此例包含了一个库，用于将数据存放在 flash 空间内而不是 ram 内。这为动态内存节约了空间，大型表格查表更容易实现。

```
#include <avr/pgmspace.h>
prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702    , 9128,    0, 25764, 8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

# 四、算数运算符

## 4.1 =（赋值运算符）

= 赋值运算符（单等号）

赋值运算符（单等号）

将等号右边的数值赋值给等号左边的变量

在 C 语言中，单等号被称为赋值运算符，它与数学上的等号含义不同，赋值运算符告诉单片机，将等号的右边的数值或计算表达式的结果，存储在等号左边的变量中。

**例子**

```
int sensVal;   //声明一个名为 sensVal 的整型变量
senVal = analogRead（0）;  //将模拟引脚 0 的输入电压存储在 SensVal 变量中
```

**编程技巧**

要确保赋值运算符（=符号）左侧的变量能够储存右边的数值。如果没有大到足以容纳右边的值，存储在变量中的值将会发生错误。

不要混淆赋值运算符[=]（单等号）与比较运算符[==]（双等号），认为这两个表达式是相等的。

## 4.2 +（加）

加,减,乘,除

**描述**

这些运算符返回两个操作数的和,差,乘积,商。这些运算是根据操作数的数据类型来计算的,比如 9 和 4 都是 int 类型,所以 9 / 4 结果是 2.这也就代表如果运算结果比数据类型所能容纳的范围要大的话，就会出现溢出(例如. 1 加上一个整数 int 类型 32,767 结果变成 -32,768)。如果操作数是不同类型的,结果是”更大”的那种数据类型。如果操作数中的其中一个是 float 类型或者 double 类型, 就变成了浮点数运算。

**例子**

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Syntax

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

Parameters:

value1: 任何常量或者变量，value2: 任何常量或者变量

**编程小提示**

整型常量的默认值是 int 类型,所以一些整型常量(定义中)的计算会导致溢出.(比如: 60 * 1000 会得到一个负数结果.那么 if(60*1000 > 0) ,if 得到的是一个 false 值。

在选择变量的数据类型时,一定要保证变量类型的范围要足够大,以至于能容纳下你的运算结果。

要知道你的变量在哪个点会”翻身”,两个方向上都得注意.如: (0 - 1) 或 (0 - - 32768)

一些数学上的分数处理,要用浮点数,但其缺点是:占用字节长度大,运算速度慢。

使用类型转换符,例如 (int)myFloat 将一个变量强制转换为 int 类型。

## 4.3 -（减）

详见 4.2+（加）

## 4.4 *（乘）

详见 4.2 +（加）

## 4.5 /（除）

详见 4.2 +（加）

## 4.6 %（取模）

**描述**

一个整数除以另一个数，其余数称为模。它有助于保持一个变量在一个特定的范围(例如数组的大小)。

**语法**

结果=被除数%除数

**参数**

被除数：一个被除的数字

除数：一个数字用于除以其他数

**返回**

余数（模）

**举例**

```
X = 7%5; // X 为 2
X = 9% 5；// X 为 4
X = 5% 5；// X 为 0
X = 4％5；  // X 为 4
```

**示例代码**

```
/*通过循环计算 1 到 10 的模*/
  int values[10];
int i = 0;

void setup () {
}

void loop()
{
   values [i] = analogRead（0）;
   i =（i + 1）％10; //取模运算
}
```

**提示**

模运算符对浮点数不起作用。

# 五、比较运算符

## 5.1 ==（等于）

if（条件判断语句）和 ==、!=、<、>（比较运算符）

if 语句与比较运算符一起用于检测某个条件是否达成，如某输入值是否在特定值之上等。if 语句的语法是：

```
if (someVariable > 50)
{
  // 执行某些语句
}
```

本程序测试 someVariable 变量的值是否大于 50。当大于 50 时，执行一些语句。换句话说，只要 if 后面括号里的结果（称之为测试表达式）为真，则执行大括号中的语句（称之为执行语句块）；若为假，则跳过大括号中的语句。 if 语句后的大括号可以省略。若省略大括号，则只有一条语句（以分号结尾）成为执行语句。

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}                                          // 以上所有书写方式都正确
```

在小括号里求值的表达式，需要以下操作符：

比较运算操作符：

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

**警告**

注意使用赋值运算符的情况（如 if (x = 10)）。一个“=”表示的是赋值运算符，作用是将 x 的值设为 10（将值 10 放入 x 变量的内存中）。两个“=”表示的是比较运算符（如 if (x == 10)），用于测试 x 和 10 是否相等。后面这个语句只有 x 是 10 时才为真，而前面赋值的那个语句则永远为真。

这是因为 C 语言按以下规则进行运算 if (x=10)：10 赋值给 x（只要非 0 的数赋值的语句，其赋值表达式的值永远为真），因此 x 现在值为 10。此时 if 的测试表达式值为 10，该值永远为真，因为非 0 值永远为真。所以，if (x = 10) 将永远为真，这就不是我们运行 if 所期待的结果。另外，x 被赋值为 10，这也不是我们所期待的结果。

if 的另外一种分支条件控制结构是 if...else 形式。

### 5.2 !=（不等于）

详见 5.1 ==（等于）

### 5.3 <（小于）

详见 5.1 ==（等于）

### 5.4 >（大于）

详见 5.1 ==（等于）

### 5.5 <=（小于等于）

详见 5.1 ==（等于）

### 5.6 >=（大于等于）

详见 5.1 ==（等于）

## 六、布尔运算符

### 6.1 &&（与）

**布尔运算符**

这些运算符可以用于 if 条件句中。

**&&（逻辑与）**

只有两个运算对象为“真”，才为“真”，如：

```
if (digitalRead(2) == HIGH   && digitalRead(3) == HIGH) { // 读取两个开关的电平
// …
}
```

如果当两个输入都为高电平，则为“真”。

**||（逻辑或）**

只要一个运算对象为“真”，就为“真”，如：

```
if (x > 0 || y > 0) {
  // …
}
```

如果 x 或 y 是大于 0，则为“真”。

**!（逻辑非）**

如果运算对象为“假”，则为“真”，例如

```
if (!x) {
  // …
}
```

如果 x 为“假”，则为真（即如果 x 等于 0）。

**警告**

千万不要误以为，符号为&(单符号)的位运算符”与”就是布尔运算符的“与”符号为&&（双符号）。他们是完全不同的符号。

同样，不要混淆布尔运算符||（双竖）与位运算符“或”符号为| （单竖）。

位运算符～（波浪号）看起来与布尔运算符 not 有很大的差别！（正如程序员说：“惊叹号”或“bang”），但你还是要确定哪一个运算符是你想要的。

**举例**

```
if (a >= 10 && a <= 20){}     // 如果 a 的值在 10 至 20 之间，则为“真”
```

### 6.2 ||（或）

详见 6.1 &&（与）

### 6.3 ！（非）

详见 6.1 &&（与）

## 七、指针运算符

### 7.1 * 取消引用运算符

**指针运算符**

**& (取地址) 和 * (取地址所指的值)**

指针对 C 语言初学者来说是一个比较复杂的内容，但是编写大部分 arduino 代码时可以不用涉及到指针。然而，操作某些数据结构时，使用指针能够简化代码，但是指针的操作知识很难在工具书中找到，可以参考 C 语言相关工具书。

### 7.2 & 引用运算符

详见 7.1 *取消引用运算符

## 八、位运算符

### 8.1 &（按位与）

**按位与（&）**

按位操作符对变量进行位级别的计算。它们能解决很多常见的编程问题。下面的材料大多来自这个非常棒的按位运算指导。

**说明和语法**

下面是所有的运算符的说明和语法。进一步的详细资料，可参考教程。

按位与(&)

位操作符与在 C + +中是一个&符，用在两个整型变量之间。按位与运算符对两侧的变量的每一位都进行运算，规则是：如果两个运算元都是 1，则结果为 1，否则输出 0.另一种表达方式：

```
0 0 1 1  运算元 1
0 1 0 1  运算元 2
----------
0 0 0 1（运算元 1&运算元 2）-返回结果
```

在 Arduino 中，int 类型为 16 位，所以在两个 int 表达式之间使用&会进行 16 个并行按

位与计算。代码片段就像这样：

```
int a =  92;      //二进制: 0000000001011100
int b = 101;      // 二进制: 0000000001100101
int c = a & b;  // 结果:      0000000001000100, 或 10 进制的 68
```

a 和 b 的 16 位每位都进行按位与计算，计算结果存在 c 中，二进制结果是 01000100，十进制结果是 68.

按位与最常见的作用是从整型变量中选取特定的位，也就是屏蔽。见下方的例子。

**按位或（|）**

按位或操作符在 C++中是|。和&操作符类似，|操作符对两个变量的为一位都进行运算，只是运算规则不同。按位或规则：只要两个位有一个为 1 则结果为 1，否则为 0。换句话说：

0 0 1 1 运算元 1

0 1 0 1 运算元 2

---

0 1 1 1（运算元 1 | 运算元 2） - 返回的结果

这里是一个按位或运算在 C + +代码片段：

```
int a =  92;      // 二进制: 0000000001011100
int b = 101;      //二进制: 0000000001100101
int c = a | b;  // 结果:      0000000001111101, 或十进制的 125
```

示例程序

按位与和按位或运算常用于端口的读取-修改-写入。在微控制器中，一个端口是一个 8 位数字，它用于表示引脚状态。对端口进行写入能同时操作所有引脚。

PORTD 是一个内置的常数，是指 0,1,2,3,4,5,6,7 数字引脚的输出状态。如果某一位为 1，着对应管脚为 HIGH。（此引脚需要先用 pinMode()命令设置为输出）因此如果我们这样写，PORTD=B00110001；则引脚 2、3、7 状态为 HIGH。这里有个小陷阱，我们可能同时更改了引脚 0、1 的状态，引脚 0、1 是 Arduino 串行通信端口，因此我们可能会干扰通信。

我们的算法的程序是：

读取 PORT 并用按位与清除我们想要控制的引脚

用按位或对 PORTD 和新的值进行运算

```
int i;      // 计数器
int j;

void setup()
DDRD = DDRD | B11111100; //设置引脚 2~7 的方向，0、1 脚不变(xx|00==xx)
//效果和 pinMode(pin,OUTPUT)设置 2~7 脚为输出一样
serial.begin（9600）;
}

void loop ()    {
for (i=0; i<64; i++){

PORTD = PORTD & B00000011;  // 清除 2~7 位，0、1 保持不变（xx & 11 == xx）
j = (i << 2);                    //将变量左移为 • 2~7 脚，避免 0、1 脚
PORTD = PORTD | j;               //将新状态和原端口状态结合以控制 LED 脚
Serial.println(PORTD, BIN); // 输出掩盖以便调试
```

```
delay(100);
}
}
```

**按位异或（^）**

C++中有一个不常见的操作符叫按位异或，也叫做 XOR（通常读作”eks-or“）。按位异或操作符用‘^'表示。此操作符和按位或（|）很相似，区别是如果两个位都为 1 则结果为 0：

```
0 0 1 1  运算元 1
0 1 0 1  运算元 2
----------
0 1 1 0（运算元 1 ^运算元 2） - 返回的结果
```

按位异或的另一种解释是如果两个位值相同则结果为 0，否则为 1。

下面是一个简单的代码**示例**：

```
int x = 12;         // 二进制: 1100
int y = 10;         // 二进制: 1010
int z = x ^ y;    // 二进制: 0110, 或十进制  6
// Blink_Pin_5
//演示“异或”
void setup(){
DDRD = DDRD | B00100000; / /设置数字脚 5 设置为输出
serial.begin（9600）;
}

void loop ()     {
PORTD = PORTD ^ B00100000;    // 反转第 5 位（数字脚 5），其他保持不变
delay(100);
}
```

## 8.2 |（按位或）

详见 8.1 &（按位与）

## 8.3 ^（按位异或）

详见 8.1 &（按位与）

## 8.4 ~（按位非）

**按位取反 (~)**

按位取反在 C+ +语言中是波浪号~。与&（按位与）和|（按位或）不同，按位取反使用在一个操作数的右侧。按位取反将操作数改变为它的“反面”：0 变为 1，1 变成 0。例如：

```
0   1       operand1
----------
1   0     ~ operand1
int a = 103;       // 二进制:   0000000001100111
int b = ~a;        // 二进制:   1111111110011000 = -104
```

你可能会惊讶地看到结果为像-104 这样的数字。这是因为整数型变量的最高位，即所谓的符号位。如果最高位是 1，这个数字将变为负数。这个正数和负数的编码被称为补。想

了解更多信息，请参考 Wikipedia 文章 two's complement.

顺便说一句，有趣的是，要注意对于任何整数型操作数 X，～X 和-X-1 是相同的。

有时，对带有符号的整数型操作数进行位操作可以造成一些不必要的意外。

## 8.5 <<（左移位运算符）

**bitshift left (<<), bitshift right (>>)**

**描述**

出自 Playground 的 The Bitmath Tutorial 在 C++语言中有两个移位运算符：左移位运算符（«）和右移运算符（»）。这些操作符可使左运算元中的某些位移动右运算元中指定的位数。

想了解有关位的更多信息可以点击这里。

**语法**

variable « number_of_bits variable » number_of_bits

**参数**

variable - (byte, int, long) number_of_bits integer ⇐ 32

**例子**

```
int a = 5;              // 二进制数: 0000000000000101
int b = a << 3;     // 二进制数: 0000000000101000, 或十进制数：40
int c = b >> 3;     // 二进制数: 0000000000000101, 或者说回到开始时的 5
//当你将 x 左移 y 位时（x«y），x 中最左边的 y 位会逐个逐个的丢失：
int a = 5;              // 二进制: 0000000000000101
int b = a << 14;   // 二进制: 0100000000000000 - 101 中最左边的 1 被丢弃
```

如果你确定位移不会引起数据溢出，你可以简单的把左移运算当做对左运算元进行 2 的右运算元次方的操作。例如，要产生 2 的次方，可使用下面的方式：

```
1 << 0 == 1
1 << 1 == 2
1 << 2 == 4
1 << 3 == 8
...
1 << 8 == 256
1 << 9 == 512
10 << 1 == 1024
...
```

当你将 x 右移 y 位(x»y)，如果 x 最高位是 1，位移结果将取决于 x 的数据类型。如果 x 是 int 类型，最高位为符号位，确定是否 x 是负数或不是，正如我们上面的讨论。如果 x 类型为 int，则最高位是符号位，正如我们以前讨论过，符号位表示 x 是正还是负。在这种情况下，由于深奥的历史原因，符号位被复制到较低位：

```
X = -16; //二进制：1111111111110000
Y = X >> 3 //二进制：1111111111111110
```

这种结果，被称为符号扩展，往往不是你想要的行为。你可能希望左边被移入的数是 0。右移操作对无符号整型来说会有不同结果，你可以通过数据强制转换改变从左边移入的数

据：

X = -16; //二进制：1111111111110000

int y = (unsigned int)x >> 3;  // 二进制: 0001111111111110

如果你能小心的避免符号扩展问题，你可以将右移操作当做对数据除 2 运算。例如：

INT = 1000;

Y = X >> 3; 8 1000  //1000 整除 8，使 y=125

### 8.6 >>（右移位运算符）

详见 8.5 <<（左移位运算符）

## 九、复合运算符

### 9.1 ++（递增）

++ (递增) / -- (递减)

**描述**

递增或递减一个变量

**语法**

x++;  //x 自增 1 返回 x 的旧值

++x;  // x 自增 1 返回 x 的新值

x--;  // x 自减 1 返回 x 的旧值

--x;  //x 自减 1 返回 x 的新值

**参数**

x: int 或 long（可能是 unsigned）

**返回**

变量进行自增/自减操作后的原值或新值。

**例子**

x = 2;

y = ++x;  // 现在 x=3，y=3

y = x--;  // 现在 x=2，y 还是 3

### 9.2 --（递减）

详见 9.1 ++ (递增)

### 9.3 +=（复合加）

**+= , -= , *= , /=**

**描述**

执行常量或变量与另一个变量的数学运算。+= 等运算符是以下扩展语法的速记。

**语法**

X += Y;  //相当于表达式 X = X + Y;

X -= Y;  //相当于表达式 X = X - Y;

X *= Y;  //相当于表达式 X = X * Y;

X /= Y;  //相当于表达式 X = X / Y;

**参数**

X：任何变量类型

Y：任何变量类型或常数

**例子**

```
x = 2;
x += 4;      // x 现在等于 6
x -= 3;      // x 现在等于 3
x *= 10;     // x 现在等于 30
x /= 2;      // x 现在等于 15
```

## 9.4 -=（复合减）

详见 9.3 += (复合加)

## 9.5 *=（复合乘）

详见 9.3 += (复合加)

## 9.6 /=（复合除）

详见 9.3 += (复合加)

## 9.6 &=（复合运算按位与）

**描述**

复合运算按位与运算符（&=）经常被用来将一个变量和常量进行运算使变量某些位变为 0。这通常被称为“清算”或“复位”位编程指南。

**语法**

x &= y;  // 等价于 x = x & y;

**参数**

x:char，int 或 long 类型变量

Y:char，int 或 long 类型常量

**例如**

首先，回顾一下按位与（&）运算符

0 0 1 1 运算元 1

0 1 0 1 运算元 2

---

0 0 0 1（运算元 1&运算元 2） - 返回的结果

任何位与 0 进行按位与操作后被清零，如果 myBite 是变量

myByte＆B00000000 = 0;

因此，任何位与 1 进行“按位与运算”后保持不变

myByte B11111111 = myByte;

**注意**：因为我们用位操作符来操作位，所以使用二进制的变量会很方便。如果这些数值是其他值将会得到同样结果，只是不容易理解。同样，B00000000 是为了标示清楚，0 在任何进制中都是 0（恩。。有些哲学的味道）因此 - 清除（置零）变量的任意位 0 和 1，而保持其余的位不变，可与常量 B11111100 进行复合运算按位与（&=）

1 0 1 0 1 0 1 0 变量

1 1 1 1 1 1 0 0 mask

---

1 0 1 0 1 0 0 0

变量不变位清零

将变量替换为 x 可得到同样结果

X X X X X X X X 变量

1 1 1 1 1 1 0 0 mask

---

X X X X X X 0 0

变量不变位清零

同理

myByte = 10101010;

myByte＆= B1111100 == B10101000;

## 9.8 |=（复合运算按位或）

**描述**

复合按位或操作符（| =）经常用于变量和常量“设置”（设置为 1），尤其是变量中的某一位。

**语法**

x |= y;　　//等价于 x = x | y;

**参数**

x: char，int 或 long 类型

y:整数，int 或 long 类型

**例如**

首先，回顾一下 OR（|）运算符

0　0　1　1　　运算元 1

0　1　0　1　　运算元 2

---

0 1 1 1（运算元 1 | 运算元 2） - 返回的结果

如果变量 myByte 中某一位与 0 经过按位或运算后不变。

myByte | B00000000 = myByte;

与 1 经过或运算的位将变为 1.

myByte | B11111111 B11111111;

因此 - 设置变量的某些位为 0 和 1，而变量的其他位不变，可与常量 B00000011 进行按位与运算（| =）

1 0 1 0 1 0 1 0 变量

0 0 0 0 0 0 1 1

---

1 0 1 0 1 0 1 1

变量保持不变位设置

接下来的操作相同，只是将变量用 x 代替

X X X X X X X X 变量

0 0 0 0 0 0 1 1 mask

---

X X X X X X 1 1

变量保持不变位设置

同上：

myByte = B10101010;

myByte | = B00000011 == B10101011;

# 变量部分

## 十、常量

### 10.1 HIGH|LOW（引脚电压定义）

**引脚电压定义，HIGH 和 LOW**

当读取（read）或写入（write）数字引脚时只有两个可能的值： HIGH 和 LOW 。

**HIGH**

HIGH（参考引脚）的含义取决于引脚（pin）的设置，引脚定义为 INPUT 或 OUTPUT 时含义有所不同。当一个引脚通过 pinMode 被设置为 INPUT，并通过 digitalRead 读取（read）时。如果当前引脚的电压大于等于 3V，微控制器将会返回为 HIGH。引脚也可以通过 pinMode 被设置为 INPUT，并通过 digitalWrite 设置为 HIGH。输入引脚的值将被一个内在的 20K 上拉电阻控制在 HIGH 上，除非一个外部电路将其拉低到 LOW。当一个引脚通过 pinMode 被设置为 OUTPUT，并 digitalWrite 设置为 HIGH 时，引脚的电压应在 5V。在这种状态下，它可以输出电流。例如，点亮一个通过一串电阻接地或设置为 LOW 的 OUTPUT 属性引脚的 LED。

**LOW**

LOW 的含义同样取决于引脚设置，引脚定义为 INPUT 或 OUTPUT 时含义有所不同。当一个引脚通过 pinMode 配置为 INPUT，通过 digitalRead 设置为读取（read）时，如果当前引脚的电压小于等于 2V，微控制器将返回为 LOW。当一个引脚通过 pinMode 配置为 OUTPUT，并通过 digitalWrite 设置为 LOW 时，引脚为 0V。在这种状态下，它可以倒灌电流。例如，点亮一个通过串联电阻连接到+5V，或到另一个引脚配置为 OUTPUT、HIGH 的的 LED。

### 10.2 INPUT|OUTPUT（数字引脚（Digital pins）定义）

**数字引脚（Digital pins）定义，INPUT 和 OUTPUT**

数字引脚当作 INPUT 或 OUTPUT 都可以。用 pinMode()方法使一个数字引脚从 INPUT 到 OUTPUT 变化。

引脚（Pins）配置为输入（Inputs）

Arduino（Atmega）引脚通过 pinMode()配置为输入（INPUT）即是将其配置在一个高阻抗的状态。配置为 INPUT 的引脚可以理解为引脚取样时对电路有极小的需求，即等效于在引脚前串联一个 100 兆欧姆(Megohms)的电阻。这使得它们非常利于读取传感器，而不是为 LED 供电。

引脚（Pins）配置为输出（Outputs）

引脚通过 pinMode()配置为输出（OUTPUT）即是将其配置在一个低阻抗的状态。

这意味着它们可以为电路提供充足的电流。Atmega 引脚可以向其他设备/电路提供（提供正电流 positive current）或倒灌（提供负电流 negative current）达 40 毫安（mA）的电流。这使得它们利于给 LED 供电，而不是读取传感器。输出（OUTPUT）引脚被短路的接地或 5V 电路上会受到损坏甚至烧毁。Atmega 引脚在为继电器或电机供电时，由于电流不足，将需

要一些外接电路来实现供电。

## 10.3 true | false（逻辑层定义）

**逻辑层定义，true 与 false（布尔 Boolean 常量）**

在 Arduino 内有两个常量用来表示真和假：true 和 false。

**false**

在这两个常量中 false 更容易被定义。false 被定义为 0（零）。

**true**

true 通常被定义为 1，这是正确的，但 true 具有更广泛的定义。在布尔含义（Boolean sense）里任何非零整数为 true。所以在布尔含义内-1，2 和-200 都定义为 ture。需要注意的是 true 和 false 常量，不同于 HIGH，LOW，INPUT 和 OUTPUT，需要全部小写。

## 10.4 integer constants（整数常量）

整数常量是直接在程序中使用的数字，如 123。默认情况下，这些数字被视为 int，但你可以通过 U 和 L 修饰符进行更多的限制（见下文）。通常情况下，整数常量默认为十进制，但可以加上特殊前缀表示为其他进制。

| 进制 | 例子 | 格式 | 备注 |
|---|---|---|---|
| 10（十进制） | 123 | 无 | |
| 2（二进制） | B1111011 | 前缀'B' | 只适用于 8 位的值（0 到 255）字符 0-1 有效 |
| 8（八进制） | 0173 | 前缀”0” | 字符 0-7 有效 |
| 16（十六进制） | 0x7B | 前缀”0x” | 字符 0-9，A-F，A-F 有效 |

小数是十进制数。这是数学常识。如果一个数没有特定的前缀，则默认为十进制。

二进制以 2 为基底，只有数字 0 和 1 是有效的。

**示例**

```
101   //和十进制 5 等价（1*2^2 + 0*2^1 + 1*2^0）
```

二进制格式只能是 8 位的，即只能表示 0-255 之间的数。如果输入二进制数更方便的话，你可以用以下的方式：

```
myInt = (B11001100 * 256) + B10101010;       // B11001100 作为高位。
```

八进制是以 8 为基底，只有 0-7 是有效的字符。前缀“0”（数字 0）表示该值为八进制。

```
0101      // 等同于十进制数 65     ((1 * 8^2) + (0 * 8^1) + 1)
```

警告:八进制数 0 前缀很可能无意产生很难发现的错误，因为你可能不小心在常量前加了个“0”，结果就悲剧了。

十六进制以 16 为基底，有效的字符为 0-9 和 A-F。十六进制数用前缀“0x”（数字 0，字母爱克斯）表示。请注意，A-F 不区分大小写，就是说你也可以用 a-f。

**示例**

```
0x101     // 等同于十进制 257     ((1 * 16^2) + (0 * 16^1) + 1)
```

**U & L 格式**

默认情况下，整型常量被视作 int 型。要将整型常量转换为其他类型时，请遵循以下规则：

'u' or 'U' 指定一个常量为无符号型。（只能表示正数和 0）例如: 33u

'l' or 'L' 指定一个常量为长整型。（表示数的范围更广）例如: 100000L

'ul' or 'UL' 这个你懂的，就是上面两种类型，称作无符号长整型。例如：32767ul

### 10.5 floating point constants（浮点常量）

和整型常量类似，浮点常量可以使得代码更具可读性。浮点常量在编译时被转换为其表达式所取的值。

**例子**

n = .005; 浮点数可以用科学记数法表示。'E'和'e'都可以作为有效的指数标志。

| 浮点数 | 被转换为 | 被转换为 |
|---|---|---|
| 10.0 | | 10 |
| 2.34E5 | 2.34 * 10^5 | 234000 |
| 67E-12 | 67.0 * 10^-12 | 0.000000000067 |

## 十一、数据类型

### 11.1 void

void 只用在函数声明中。它表示该函数将不会被返回任何数据到它被调用的函数中。

**例子**

```
//功能在“setup”和“loop”被执行
//但没有数据被返回到高一级的程序中
void setup()
{
// ...
}

void loop()
{
// ...
}
```

### 11.2 boolean （布尔）

一个布尔变量拥有两个值，true 或 false。（每个布尔变量占用一个字节的内存。）

**例子**

```
int LEDpin = 5;            // LED 与引脚 5 相连
int switchPin = 13;     // 开关的一个引脚连接引脚 13，另一个引脚接地。
boolean running = false;

void setup()
{
   pinMode(LEDpin, OUTPUT);
   pinMode(switchPin, INPUT);
   digitalWrite(switchPin, HIGH);          // 打开上拉电阻
}
```

```
void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // 按下开关 - 使引脚拉向高电势
    delay(100);                                  // 通过延迟，以滤去开关抖动产生的杂波
    running = !running;                          // 触发 running 变量
    digitalWrite(LEDpin, running)          //点亮 LED
    }
  }
```

## 11.3 char（字符或字符串）

一个数据类型，占用 1 个字节的内存存储一个字符值。字符都写在单引号，如'A'；（多个字符（字符串）使用双引号，如“ABC”）。

字符以编号的形式存储。你可以在 ASCII 表中看到对应的编码。这意味着字符的 ASCII 值可以用来作数学计算。（例如'A'+ 1，因为大写 A 的 ASCII 值是 65，所以结果为 66）。如何将字符转换成数字参考 serial.println 命令。

char 数据类型是有符号的类型，这意味着它的编码为-128 到 127。对于一个无符号一个字节（8 位）的数据类型，使用 byte 数据类型。

**例如**

```
char myChar = 'A';
char myChar = 65;          // both are equivalent
```

## 11.4 unsigned char（无符号数据类型）

一个无符号数据类型占用 1 个字节的内存。与 byte 的数据类型相同。

无符号的 char 数据类型能编码 0 到 255 的数字。

为了保持 Arduino 的编程风格的一致性，byte 数据类型是首选。

**例子**

```
unsigned char myChar = 240;
```

## 11.5 byte（无符号数）

一个字节存储 8 位无符号数，从 0 到 255。

**例子**

```
byte b = B10010;   // "B" 是二进制格式（B10010 等于十进制 18）
```

## 11.6 int（整型）

整数是基本数据类型，占用 2 字节。整数的范围为-32,768 到 32,767（ -2^15 ~（2^15)-1）。

整数类型使用 2 的补码方式存储负数。最高位通常为符号位，表示数的正负。其余位被“取反加 1”（此处请参考补码相关资料，不再赘述）。

Arduino 为您处理负数计算问题，所以数学计算对您是透明的（术语：实际存在，但不可操作。相当于“黑盒”）。但是，当处理右移位运算符（»）时，可能有未预期的编译过程。

**示例**

```
int ledPin = 13;
```

**语法**

int var = val;

var - 变量名

val - 赋给变量的值

**提示**

当变量数值过大而超过整数类型所能表示的范围时（-32,768 到 32,767），变量值会“回滚”（详情见示例）。

```
int x
x = -32,768;
x = x - 1;          // x 现在是 32,767。
x = 32,767;
x = x + 1;          // x 现在是 -32,768。
```

## 11.7 unsigned int（无符号整型）

**描述**

无符号整型变量扩充了变量容量以存储更大的数据，它能存储 32 位(4 字节)数据。与标准长整型不同无符号长整型无法存储负数，其范围从 0 到 4,294,967,295（2 ^ 32 - 1）。

**例子**

```
unsigned long time;
void setup()
{
        Serial.begin(9600);
}

void loop()
{
   Serial.print("Time: ");
   time = millis();
//程序开始后一直打印时间
   Serial.println(time);
//等待一秒钟，以免发送大量的数据
        delay(1000);
}
```

**语法**

unsigned long var = val;

var - 你所定义的变量名

val - 给变量所赋的值

## 11.8 word

**描述**

一个存储一个 16 字节无符号数的字符，取值范围从 0 到 65535，与 unsigned int 相同。

**例子**

```
word w = 10000;
```

## 11.9 long(长整数型)

**描述**

长整数型变量是扩展的数字存储变量，它可以存储 32 位（4 字节）大小的变量，从 -2,147,483,648 到 2,147,483,647。

**例子**

long speedOfLight = 186000L; //参见整数常量‘L’的说明

**语法**

long var = val;

var - 长整型变量名

var - 赋给变量的值

## 11.10 unsigned long(无符号长整数型)

**描述**

无符号长整型变量扩充了变量容量以存储更大的数据，它能存储 32 位(4 字节)数据。与标准长整型不同无符号长整型无法存储负数，其范围从 0 到 4,294,967,295（2 ^ 32 - 1）。

**例子**

```
unsigned long time;
void setup()
{
        Serial.begin(9600);
}

void loop()
{
   Serial.print("Time: ");
   time = millis();
//程序开始后一直打印时间
   Serial.println(time);
//等待一秒钟，以免发送大量的数据
        delay(1000);
}
```

**语法**

unsigned long var = val;

var - 你所定义的变量名

val - 给变量所赋的值

## 11.11 float（浮点型数）

**描述**

float，浮点型数据，就是有一个小数点的数字。浮点数经常被用来近似的模拟连续值，因为他们比整数更大的精确度。浮点数的取值范围在 3.4028235 E+38 ~ -3.4028235E +38。它被存储为 32 位（4 字节）的信息。

float 只有 6-7 位有效数字。这指的是总位数，而不是小数点右边的数字。与其他平台不同的是，在那里你可以使用 double 型得到更精确的结果（如 15 位），在 Arduino 上，double 型与 float 型的大小相同。

浮点数字在有些情况下是不准确的，在数据大小比较时，可能会产生奇怪的结果。例如

6.0 / 3.0 可能不等于 2.0。你应该使两个数字之间的差额的绝对值小于一些小的数字，这样就可以近似的得到这两个数字相等这样的结果。

浮点运算速度远远慢于执行整数运算，例如，如果这个循环有一个关键的计时功能，并需要以最快的速度运行，就应该避免浮点运算。程序员经常使用较长的程式把浮点运算转换成整数运算来提高速度。

**举例**

```
float myfloat;
float sensorCalbrate = 1.117;
```

**语法**

```
float var = val;
```

var——您的 float 型变量名称

val——分配给该变量的值

**示例代码**

```
int x;
int y;
float z;
x = 1;
y = x / 2;              // Y 为 0，因为整数不能容纳分数
z = (float)x / 2.0;     // Z 为 0.5（你必须使用 2.0 做除数，而不是 2）
```

## 11.12 double（双精度浮点数）

**描述**

双精度浮点数。占用 4 个字节。

目前的 arduino 上的 double 实现和 float 相同，精度并未提高。

**提示**

如果你从其他地方得到的代码中包含了 double 类变量，最好检查一遍代码以确认其中的变量的精确度能否在 arduino 上达到。

## 11.13 string（char array/字符串）

**string（字符串）**

**描述**

文本字符串可以有两种表现形式。你可以使用字符串数据类型（这是 0019 版本的核心部分），或者你可以做一个字符串，由 char 类型的数组和空终止字符('\0')构成。（求助，待润色-Leo）本节描述了后一种方法。而字符串对象（String object）将让你拥有更多的功能，同时也消耗更多的内存资源，关于它的详细信息，请参阅页面（String object）[超链接]

**举例**

以下所有字符串都是有效的声明。

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

声明字符串的解释

在 Str1 中声明一个没有初始化的字符数组

在 Str2 中声明一个字符数组（包括一个附加字符），编译器会自动添加所需的空字符

在 Str3 中明确加入空字符

在 Str4 中用引号分隔初始化的字符串常数，编译器将调整数组的大小，以适应字符串常量和终止空字符

在 Str5 中初始化一个包括明确的尺寸和字符串常量的数组

在 Str6 中初始化数组，预留额外的空间用于一个较大的字符串

**空终止字符**

一般来说，字符串的结尾有一个空终止字符（ASCII 代码 0）。以此让功能函数（例如 Serial.pring()）知道一个字符串的结束。否则，他们将从内存继续读取后续字节，而这些并不属于所需字符串的一部分。

这意味着，你的字符串比你想要的文字包含更多的个字符空间。这就是为什么 Str2 和 Str5 需要八个字符，即使“Arduino”只有七个字符 - 最后一个位置会自动填充空字符。str4 将自动调整为八个字符，包括一个额外的空。在 Str3 的，我们自己已经明确地包含了空字符(写入'\ 0')。

需要注意的是，字符串可能没有一个最后的空字符（例如在 Str2 中您已定义字符长度为 7，而不是 8）。这会破坏大部分使用字符串的功能，所以不要故意而为之。如果你注意到一些奇怪的现象（在字符串中操作字符），基本就是这个原因导致的了。

**单引号？还是双引号？**

定义字符串时使用双引号（例如“ABC”），而定义一个单独的字符时使用单引号（例如'A'）

**包装长字符串**

你可以像这样打包长字符串： char myString[] = “This is the first line”” this is the second line”” etcetera”；

**字符串数组**

当你的应用包含大量的文字，如带有液晶显示屏的一个项目，建立一个字符串数组是非常便利的。因为字符串本身就是数组，它实际上是一个两维数组的典型。

在下面的代码，”char*”在字符数据类型 char 后跟了一个星号'*'表示这是一个“指针”数组。所有的数组名实际上是指针，所以这需要一个数组的数组。指针对于 C 语言初学者而言是非常深奥的部分之一，但我们没有必要了解详细指针，就可以有效地应用它。

**样例**

```
char* myStrings[]={
   "This is string 1", "This is string 2", "This is string 3",
   "This is string 4", "This is string 5","This is string 6"};

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

void loop(){
   for (int i = 0; i < 6; i++){
      Serial.println(myStrings[i]);
      delay(500);
   }
```

}

## 11.14 String object（String 类）

**描述**

String 类，是 0019 版的核心的一部分，允许你实现比运用字符数组更复杂的文字操作。你可以连接字符串，增加字符串，寻找和替换子字符串以及其他操作。它比使用一个简单的字符数组需要更多的内存，但它更方便。

仅供参考，字符串数组都用小写的 string 表示而 String 类的实例通常用大写的 String 表示。注意，在“双引号”内指定的字符常量通常被作为字符数组，并非 String 类实例。

**函数**

String
charAt()
compareTo()
concat()
endsWith()
equals()
equalsIgnoreCase()
GetBytes()
indexOf()
lastIndexOf
length
replace()
setCharAt()
startsWith()
substring()
toCharArray()
toLowerCase()
toUpperCase()
trim()

**操作符**

[]（元素访问）
+（串连）
==（比较）

**举例**

StringConstructors
StringAdditionOperator
StringIndexOf
StringAppendOperator
StringLengthTrim
StringCaseChanges
StringReplace
StringCharacters
StringStartsWithEndsWith
StringComparisonOperators

StringSubstring

## 11.15 array （数组）

数组是一种可访问的变量的集合。Arduino 的数组是基于 C 语言的，因此这会变得很复杂，但使用简单的数组是比较简单的。

**创建（声明）一个数组**

下面的方法都可以用来创建（声明）数组。

```
myInts [6];
myPins [] = {2，4，8，3，6};
mySensVals [6] = {2，4，-8，3，2};
char message[6] = "hello";
```

你声明一个未初始化数组，例如 myPins。

在 myPins 中，我们声明了一个没有明确大小的数组。编译器将会计算元素的大小，并创建一个适当大小的数组。

当然，你也可以初始化数组的大小，例如在 mySensVals 中。请注意，当声明一个 char 类型的数组时，你初始化的大小必须大于元素的个数，以容纳所需的空字符。

**访问数组**

数组是从零开始索引的，也就说，上面所提到的数组初始化，数组第一个元素是为索引 0，因此：

mySensVals [0] == 2，mySensVals [1] == 4，

依此类推。

这也意味着，在包含十个元素的数组中，索引九是最后一个元素。因此，

```
int myArray[10] = {9,3,2,4,3,2,7,8,9,11};
// myArray[9]的数值为 11
// myArray[10]，该索引是无效的，它将会是任意的随机信息（内存地址）
```

出于这个原因，你在访问数组应该小心。若访问的数据超出数组的末尾（即索引数大于你声明的数组的大小- 1），则将从其他内存中读取数据。从这些地方读取的数据，除了产生无效的数据外，没有任何作用。向随机存储器中写入数据绝对是一个坏主意，通常会导致不愉快的结果，如导致系统崩溃或程序故障。要排查这样的错误是也是一件难事。不同于 Basic 或 JAVA，C 语言编译器不会检查你访问的数组是否大于你声明的数组。

**指定一个数组的值**

mySensVals [0] = 10;

从数组中访问一个值：

X = mySensVals [4];

**数组和循环**

数组往往在 for 循环中进行操作，循环计数器可用于访问每个数组元素。例如，将数组中的元素通过串口打印，你可以这样做：

```
int i;
for (i = 0; i < 5; i = i + 1) {
Serial.println(myPins[i]);
}
```

**例子**

如果你需要一个演示数组的完整程序，请参考 Knight Rider exampel。

# 十二、数据类型转换

## 12.1 char()

**描述**

将一个变量的类型变为 char。

**语法**

char(x)

**参数**

x：任何类型的值

**返回**

char

## 12.2 byte()

**描述**

将一个值转换为字节型数值。

**语法**

byte(x)

**参数**

X：任何类型的值

**返回**

字节

## 12.3int()

**描述**

将一个值转换为 int 类型。

**语法**

int(x)

**参数**

x:一个任何类型的值

**返回值**

int 类型的值

## 12.4 word()

**描述**

把一个值转换为 word 数据类型的值，或由两个字节创建一个字符。

**语法**

word(x)

word(h, l)

**参数**

X：任何类型的值

H：高阶（最左边）字节

L：低序（最右边）字节

**返回值**

字符

### 12.5 long()

**描述**

将一个值转换为长整型数据类型。

**语法**

long(x)

**参数**

x:任意类型的数值

**返回值**

长整型数

### 12.6 float()

**描述**

将一个值转换为 float 型数值。

**语法**

float(x)

**参数**

X：任何类型的值

**返回值**

float 型数

**注释**

见 float 中关于 Arduino 浮点数的精度和限制的详细信息。

## 十三、变量作用域&修饰符

### 13.1 variable scope（变量的作用域）

**变量的作用域**

在 Arduino 使用的 C 编程语言的变量，有一个名为作用域(scope) 的属性。这一点与类似 BASIC 的语言形成了对比，在 BASIC 语言中所有变量都是全局(global) 变量。

在一个程序内的全局变量是可以被所有函数所调用的。局部变量只在声明它们的函数内可见。在 Arduino 的环境中，任何在函数（例如，setup(),loop()等）外声明的变量，都是全局变量。

当程序变得更大更复杂时，局部变量是一个有效确定每个函数只能访问其自己变量的途径。这可以防止，当一个函数无意中修改另一个函数使用的变量的程序错误。

有时在一个 for 循环内声明并初始化一个变量也是很方便的选择。这将创建一个只能从 for 循环的括号内访问的变量。

**例子**

```
int gPWMval;   // 任何函数都可以调用此变量
void setup()
{
  // ...
}
```

```
void loop()
{
  int i;      // "i" 只在 "loop" 函数内可用
  float f;    // "f" 只在 "loop" 函数内可用
  // ...

  for (int j = 0; j <100; j++){
    //变量 j 只能在循环括号内访问
  }
}
```

## 13.2 static（静态变量）

static 关键字用于创建只对某一函数可见的变量。然而，和局部变量不同的是，局部变量在每次调用函数时都会被创建和销毁，静态变量在函数调用后仍然保持着原来的数据。

静态变量只会在函数第一次调用的时候被创建和初始化。

**例子**

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk 函数在两个终点间随机的上下移动
*  在一个循环中最大的移动由参数“stepsize”决定
*一个静态变量向上和向下移动一个随机量
*这种技术也被叫做“粉红噪声”或“醉步”
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20

int stepsize;
int thisTime;
int total;

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

void loop()
{          //   测试 randomWalk  函数
  stepsize = 5;
  thisTime = randomWalk(stepsize);
serial.println（thisTime）;
   delay(10);
}
```

```
int randomWalk(int moveSize){
  static int    place;        // 在 randomwalk 中存储变量
                              // 声明为静态因此它在函数调用之间能保持数据，但其他函数无法改变它的值
    place = place + (random(-moveSize, moveSize + 1));
  if (place < randomWalkLowRange){                    //检查上下限
    place = place + (randomWalkLowRange - place);     // 将数字变为正方向
}
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange);    // 将数字变为负方向
}
    return place;
}
```

## 13.3 volatile

volatile 这个关键字是变量修饰符，常用在变量类型的前面，以告诉编译器和接下来的程序怎么对待这个变量。

声明一个 volatile 变量是编译器的一个指令。编译器是一个将你的 C/C++代码转换成机器码的软件，机器码是 arduino 上的 Atmega 芯片能识别的真正指令。

具体来说，它指示编译器编译器从 RAM 而非存储寄存器中读取变量，存储寄存器是程序存储和操作变量的一个临时地方。在某些情况下，存储在寄存器中的变量值可能是不准确的。

如果一个变量所在的代码段可能会意外地导致变量值改变那此变量应声明为 volatile，比如并行多线程等。在 arduino 中，唯一可能发生这种现象的地方就是和中断有关的代码段，成为中断服务程序。

**例子**

```
//当中断引脚改变状态时，开闭 LED
int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
```

```
    state = !state;
  }
```

## 13.4 const

const 关键字代表常量。它是一个变量限定符，用于修改变量的性质，使其变为只读状态。这意味着该变量，就像任何相同类型的其他变量一样使用，但不能改变其值。如果尝试为一个 const 变量赋值，编译时将会报错。

const 关键字定义的常量，遵守 variable scoping 管辖的其他变量的规则。这一点加上使用 #define 的缺陷，使 const 关键字成为定义常量的一个的首选方法。

**例子**

```
const float pi = 3.14;
float x;

// ....

x = pi * 2;     // 在数学表达式中使用常量不会报错
pi = 7;         // 错误的用法 - 你不能修改常量值，或给常量赋值。
```

**#define 或 const**

您可以使用 const 或＃define 创建数字或字符串常量。但 arrays，你只能使用 const。一般 const 相对的＃define 是首选的定义常量语法。

# 十四、辅助工具

## 14.1 sizeof()

**描述**

sizeof 操作符返回一个变量类型的字节数，或者该数在数组中占有的字节数。

**语法**

```
sizeof(variable)
```

**参数**

variable：任何变量类型或数组（如 int，float，byte）

**示例代码**

sizeof 操作符用来处理数组非常有效，它能很方便的改变数组的大小而不用破坏程序的其他部分。

这个程序一次打印出一个字符串文本的字符。尝试改变一下字符串。

```
char myStr[] = "this is a test";
int i;

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

{0}void{/0}{1} {/1}{2}loop{/2}{1}() {{/1}
```

```
    for (i = 0; i < sizeof(myStr) - 1; i++){
      Serial.print(i, DEC);
      Serial.print(" = ");
      Serial.println(myStr[i], BYTE);
    }
}
```

请注意 sizeof 返回字节数总数。因此，较大的变量类型，如整数，for 循环看起来应该像这样。

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
  //用 myInts[i]来做些事
}
```

# 函数部分

## 十五、数字 I/O

### 15.1 pinMode()

**描述**

将指定的引脚配置成输出或输入。详情请见 digital pins。

**语法**

pinMode(pin, mode)

**参数**

pin:要设置模式的引脚

mode:INPUT 或 OUTPUT

**返回**

无

**例子**

```
ledPin = 13 // LED 连接到数字脚 13

void setup()
{
pinMode（ledPin，OUTPUT）; //设置数字脚为输出
}

void loop()
{
digitalWrite（ledPin，HIGH）; //点亮 LED
  delay(1000);                    // 等待一秒
  digitalWrite(ledPin, LOW);      // 灭掉 LED
延迟（1000）; //等待第二个
}
```

**注意**

模拟输入脚也能当做数字脚使用，参见 A0，A1 等

## 15.2 digitalWrite()

**描述**

给一个数字引脚写入 HIGH 或者 LOW。

如果一个引脚已经使用 pinMode()配置为 OUTPUT 模式，其电压将被设置为相应的值，HIGH 为 5V（3.3V 控制板上为 3.3V），LOW 为 0V。

如果引脚配置为 INPUT 模式，使用 digitalWrite()写入 HIGH 值，将使内部 20K 上拉电阻（详见数字引脚教程）。写入 LOW 将会禁用上拉。上拉电阻可以点亮一个 LED 让其微微亮，如果 LED 工作，但是亮度很低，可能是因为这个原因引起的。补救的办法是使用 pinMode()函数设置为输出引脚。

注意：数字 13 号引脚难以作为数字输入使用，因为大部分的控制板上使用了一颗 LED 与一个电阻连接到他。如果启动了内部的 20K 上拉电阻，他的电压将在 1.7V 左右，而不是正常的 5V，因为板载 LED 串联的电阻把他使他降了下来，这意味着他返回的值总是 LOW。如果必须使用数字 13 号引脚的输入模式，需要使用外部上拉下拉电阻。

语法

digitalWrite(pin, value)

**参数**

pin: 引脚编号（如 1,5,10，A0，A3）

value: HIGH or LOW

**返回**

无

**例子**

```
int ledPin = 13;                    // LED 连接到数字 13 号端口

void setup()
{
  pinMode(ledPin, OUTPUT);          // 设置数字端口为输入模式
}

void loop()
{
  digitalWrite(ledPin, HIGH);       // 使 LED 亮
  delay(1000);                      // 延迟一秒
  digitalWrite(ledPin, LOW);        // 使 LED 灭
  delay(1000);                      // 延迟一秒
}
```

13 号端口设置为高电平，延迟一秒，然后设置为低电平。

**注释**

模拟引脚也可以当做数字引脚使用，使用方法是输入端口 A0，A1，A2 等。

## 15.3 digitalRead()

**描述**

读取指定引脚的值，HIGH 或 LOW。

**语法**

digitalRead（PIN）

**参数**

pin：你想读取的引脚号（int）

**返回**

HIGH 或 LOW

**例子**

```
ledPin = 13 // LED 连接到 13 脚
int inPin = 7;     // 按钮连接到数字引脚 7
int val = 0;   //定义变量存以储读值

void setup()
{
   pinMode(ledPin, OUTPUT);          // 将 13 脚设置为输出
   pinMode(inPin, INPUT);          // 将 7 脚设置为输入
}

void loop()
{
   val = digitalRead(inPin);     // 读取输入脚
   digitalWrite(ledPin, val);       //将 LED 值设置为按钮的值
}
```

将 13 脚设置为输入脚 7 脚的值。

**注意**

如果引脚悬空，digitalRead()会返回 HIGH 或 LOW（随机变化）。

模拟输入脚能当做数字脚使用，参见 A0，A1 等。

# 十六、模拟 I/O

## 16.1 analogReference()

**描述**

配置用于模拟输入的基准电压（即输入范围的最大值）。选项有:

DEFAULT：默认 5V（Arduino 板为 5V）或 3.3 伏特（Arduino 板为 3.3V）为基准电压。

INTERNAL：在 ATmega168 和 ATmega328 上以 1.1V 为基准电压，以及在 ATmega8 上以 2.56V 为基准电压（Arduino Mega 无此选项）

INTERNAL1V1：以 1.1V 为基准电压（此选项仅针对 Arduino Mega）

INTERNAL2V56：以 2.56V 为基准电压（此选项仅针对 Arduino Mega）

EXTERNAL：以 AREF 引脚（0 至 5V）的电压作为基准电压。

**参数**

type：使用哪种参考类型（DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, 或者 EXTERNAL）。

**返回**

无

**注意事项**

改变基准电压后，之前从 analogRead()读取的数据可能不准确。

**警告**

不要在 AREF 引脚上使用使用任何小于 0V 或超过 5V 的外部电压。如果你使用 AREF 引脚上的电压作为基准电压，你在调用 analogRead()前必须设置参考类型为 EXTERNAL。否则，你将会削短有效的基准电压（内部产生）和 AREF 引脚，这可能会损坏您 Arduino 板上的单片机。

另外，您可以在外部基准电压和 AREF 引脚之间连接一个 5K 电阻，使你可以在外部和内部基准电压之间切换。请注意，总阻值将会发生改变，因为 AREF 引脚内部有一个 32K 电阻。这两个电阻都有分压作用。所以，例如，如果输入 2.5V 的电压，最终在在 AREF 引脚上的电压将为 2.5 * 32 /（32 + 5）= 2.2V。

## 16.2 analogRead()

**描述**

从指定的模拟引脚读取数据值。Arduino 板包含一个 6 通道（Mini 和 Nano 有 8 个通道，Mega 有 16 个通道），10 位模拟数字转换器。这意味着它将 0 至 5 伏特之间的输入电压映射到 0 至 1023 之间的整数值。这将产生读数之间的关系：5 伏特/ 1024 单位，或 0.0049 伏特（4.9 mV）每单位。输入范围和精度可以使用 analogReference()改变。它需要大约 100 微秒（0.0001）来读取模拟输入，所以最大的阅读速度是每秒 10000 次。

**语法**

analogRead（PIN）

**数值的读取**

引脚：从输入引脚（大部分板子从 0 到 5，Mini 和 Nano 从 0 到 7，Mega 从 0 到 15）读取数值

**返回**

从 0 到 1023 的整数值

**注意事项**

如果模拟输入引脚没有连入电路，由 analogRead()返回的值将根据多项因素（例如其他模拟输入引脚，你的手靠近板子等）产生波动。

**例子**

```
int analogPin = 3;      //电位器（中间的引脚）连接到模拟输入引脚 3
                              //另外两个引脚分别接地和+5 V
int val = 0;   //定义变量来存储读取的数值

void setup()

{
serial.begin（9600）; //设置波特率（9600）
}

void loop()
{
val = analogRead（analogPin）; //从输入引脚读取数值
serial.println（val）; //显示读取的数值
```

}

## 16.3 analogWrite() PWM

**描述**

从一个引脚输出模拟值（PWM）。可用于让 LED 以不同的亮度点亮或驱动电机以不同的速度旋转。analogWrite()输出结束后，该引脚将产生一个稳定的特殊占空比方波，直到下次调用 analogWrite()（或在同一引脚调用 digitalRead()或 digitalWrite()）。PWM 信号的频率大约是 490 赫兹。

在大多数 arduino 板（ATmega168 或 ATmega328），只有引脚 3，5，6，9，10 和 11 可以实现该功能。在 aduino Mega 上，引脚 2 到 13 可以实现该功能。老的 Arduino 板（ATmega8）的只有引脚 9、10、11 可以使用 analogWrite()。在使用 analogWrite()前，你不需要调用 pinMode()来设置引脚为输出引脚。

analogWrite 函数与模拟引脚、analogRead 函数没有直接关系。

**语法**

analogWrite（pin,value）

**参数**

pin：用于输入数值的引脚。

value： 占空比：0（完全关闭）到 255（完全打开）之间。

**返回**

无

**说明和已知问题**

引脚 5 和 6 的 PWM 输出将高于预期的占空比（输出的数值偏高）。这是因为 millis()和 delay()功能，和 PWM 输出共享相同的内部定时器。这将导致大多时候处于低占空比状态（如：0 - 10），并可能导致在数值为 0 时，没有完全关闭引脚 5 和 6。

**例子**

通过读取电位器的阻值控制 LED 的亮度

```
int ledPin = 9;   // LED 连接到数字引脚 9
int analogPin = 3;   //电位器连接到模拟引脚 3
int val = 0;   //定义变量存以储读值

void setup()

{
pinMode（ledPin,OUTPUT）;   //设置引脚为输出引脚
}

void loop()
{
val = analogRead（analogPin）;   //从输入引脚读取数值
analogWrite（ledPin，val / 4）;   //  以 val / 4 的数值点亮 LED（因为 analogRead 读取的数值从 0 到 1023，而 analogWrite 输出的数值从 0 到 255）
}
```

# 十七、高级 I/O

## 17.1tone()

**描述**

在一个引脚上产生一个特定频率的方波（50%占空比）。持续时间可以设定，否则波形会一直产生直到调用 noTone()函数。该引脚可以连接压电蜂鸣器或其他喇叭播放声音。

在同一时刻只能产生一个声音。如果一个引脚已经在播放音乐，那调用 tone()将不会有任何效果。如果音乐在同一个引脚上播放，它会自动调整频率。

使用 tone()函数会与 3 脚和 11 脚的 PWM 产生干扰（Mega 板除外）。

注意：如果你要在多个引脚上产生不同的音调，你要在对下一个引脚使用 tone()函数前对此引脚调用 noTone()函数。

**语法**

tone(pin, frequency)

tone(pin, frequency, duration)

**参数**

pin：要产生声音的引脚

frequency: 产生声音的频率，单位 Hz，类型 unsigned int

duration：声音持续的时间，单位毫秒（可选），类型 unsigned long

## 17.2 noTone()

**描述**

停止由 tone()产生的方波。如果没有使用 tone()将不会有效果。

注意：如果你想在多个引脚上产生不同的声音，你要在对下个引脚使用 tone()前对刚才的引脚调用 noTone().

**语法**

noTone(pin)

**参数**

pin: 所要停止产生声音的引脚

## 17.3 shiftOut()

**描述**

将一个数据的一个字节一位一位的移出。从最高有效位（最左边）或最低有效位（最右边）开始。依次向数据脚写入每一位，之后时钟脚被拉高或拉低，指示刚才的数据有效。

注意：如果你所连接的设备时钟类型为上升沿，你要确定在调用 shiftOut()前时钟脚为低电平，如调用 digitalWrite(clockPin, LOW)。

注意：这是一个软件实现；Arduino 提供了一个硬件实现的 SPI 库，它速度更快但只在特定脚有效。

**语法**

shiftOut(dataPin, clockPin, bitOrder, value)

**参数**

dataPin：输出每一位数据的引脚(int)

clockPin：时钟脚，当 dataPin 有值时此引脚电平变化(int)

bitOrder：输出位的顺序，最高位优先或最低位优先

value: 要移位输出的数据(byte)

**返回**

无

**注意**

dataPin 和 clockPin 要用 pinMode()配置为输出。shiftOut 目前只能输出 1 个字节（8 位），所以如果输出值大于 255 需要分两步。

```
//最高有效位优先串行输出
int  数据= 500;
//移位输出高字节
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
//移位输出低字节
shiftOut(data, clock, MSBFIRST, data);

//最低有效位优先串行输出
data = 500;
//移位输出低字节
shiftOut(dataPin, clock, LSBFIRST, data);
//移位输出高字节
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

例子

相应电路，查看 tutorial on controlling a 74HC595 shift register

```
// ********************************************** ************* //
//   Name       : shiftOut 代码, Hello World                                  //
//   Author    : Carlyn Maw,Tom Igoe                                          //
//   Date       : 25 Oct, 2006                                                //
// 版本 : 1.0                                                              //
//   注释：使用 74HC595 移位寄存器从 0 到 255 计数                  //
//
// ********************************************** ***************

//引脚连接到 74HC595 的 ST_CP
int latchPin = 8;
//引脚连接到 74HC595 的 SH_CP
int clockPin = 12;
// //引脚连接到 74HC595 的 DS
int dataPin = 11;

void setup() {
//设置引脚为输出
   pinMode(latchPin, OUTPUT);
   pinMode(clockPin, OUTPUT);
   pinMode(dataPin, OUTPUT);
}
```

```
void loop() {
  //向上计数程序
（J = 0; J <256; J + +）{
      //传输数据的时候将 latchPin 拉低
digitalWrite(latchpin, LOW);
      shiftOut 的（dataPin，clockPin，LSBFIRST，J）;
     //之后将 latchPin 拉高以告诉芯片
      //它不需要再接受信息了
digitalWrite(latchpin, HIGH);
      delay(1000);

}
}
```

## 17.4 shiftIn()

**描述**

将一个数据的一个字节一位一位的移入。从最高有效位（最左边）或最低有效位（最右边）开始。对于每个位，先拉高时钟电平，再从数据传输线中读取一位，再将时钟线拉低。

注意：这是一个软件实现；Arduino 提供了一个硬件实现的 SPI 库，它速度更快但只在特定脚有效。

**语法**

shiftIn(dataPin,clockPin,bitOrder)

**参数**

dataPin：输出每一位数据的引脚(int)

clockPin：时钟脚，当 dataPin 有值时此引脚电平变化(int)

bitOrder：输出位的顺序，最高位优先或最低位优先

## 17.5 pulseIn()

**描述**

读取一个引脚的脉冲（HIGH 或 LOW）。例如，如果 value 是 HIGH，pulseIn()会等待引脚变为 HIGH，开始计时，再等待引脚变为 LOW 并停止计时。返回脉冲的长度，单位微秒。如果在指定的时间内无脉冲函数返回。

此函数的计时功能由经验决定，长时间的脉冲计时可能会出错。计时范围从 10 微秒至 3 分钟。（1 秒=1000 毫秒=1000000 微秒）

**语法**

pulseIn(pin, value)

pulseIn(pin, value, timeout)

**参数**

pin:你要进行脉冲计时的引脚号（int）。

value:要读取的脉冲类型，HIGH 或 LOW（int）。

timeout (可选)：指定脉冲计数的等待时间，单位为微秒，默认值是 1 秒（unsigned long）

**返回**

脉冲长度（微秒），如果等待超时返回 0（unsigned long）

**例子**

```
int pin = 7;
unsigned long duration;

void setup()
{
   pinMode(pin, INPUT);
}

void loop()
{
duration = pulseIn(pin, HIGH);;
}
```

# 十八、时间

## 18.1 millis()

**描述**

返回 Arduino 开发板从运行当前程序开始的毫秒数。这个数字将在约 50 天后溢出（归零）。

**参数**

无

**返回**

返回从运行当前程序开始的毫秒数（无符号长整数）。

**例子**

```
unsigned long time;

void setup(){
        Serial.begin(9600);
}
void loop(){
serial.print("Time:");
time = millis();
//打印从程序开始到现在的时间
serial.println(time);
//等待一秒钟，以免发送大量的数据
        delay(1000);
}
```

**提示**

注意，参数 millis 是一个无符号长整数，试图和其他数据类型（如整型数）做数学运算可能会产生错误。

当中断函数发生时，millis()的数值将不会继续变化。

## 18.2 micros()

**描述**

返回 Arduino 开发板从运行当前程序开始的微秒数。这个数字将在约 70 分钟后溢出(归零)。在 16MHz 的 Arduino 开发板上（比如 Duemilanove 和 Nano），这个函数的分辨率为四微秒（即返回值总是四的倍数）。在 8MHz 的 Arduino 开发板上（比如 LilyPad），这个函数的分辨率为八微秒。

注意：每毫秒是 1,000 微秒，每秒是 1,000,000 微秒。

**参数**

无

**返回**

返回从运行当前程序开始的微秒数（无符号长整数）。

**例子**

```
unsigned long time;

void setup(){
        Serial.begin(9600);
}
void loop(){
Serial.print（“Time:”）;
time = micros();
//打印从程序开始的时间
Serial.println(time);
//等待一秒钟，以免发送大量的数据
        delay(1000);
}
```

## 18.3 delay()

**描述**

使程序暂定设定的时间（单位毫秒）。（一秒等于 1000 毫秒）

**语法**

delay(ms)

**参数**

ms：暂停的毫秒数（unsigned long）

**返回**

无

**例子**

```
ledPin = 13 / / LED 连接到数字 13 脚

void setup()
{
   pinMode(ledPin, OUTPUT);          // 设置引脚为输出
}

void loop()
{
```

```
    digitalWrite(ledPin, HIGH);    // 点亮 LED
    delay(1000);                   // 等待 1 秒
    digitalWrite(ledPin, LOW);     // 灭掉 LED
    delay(1000);                   // 等待一秒
}
```

**警告**

虽然创建一个使用 delay()的闪烁 LED 很简单，并且许多例子将很短的 delay 用于消除开关抖动，delay()确实拥有很多显著的缺点。在 delay 函数使用的过程中，读取传感器值、计算、引脚操作均无法执行，因此，它所带来的后果就是使其他大多数活动暂停。其他操作定时的方法请参加 millis()函数和它下面的例子。大多数熟练的程序员通常避免超过 10 毫秒的 delay(),除非 arduino 程序非常简单。

但某些操作在 delay()执行时任然能够运行，因为 delay 函数不会使中断失效。通信端口 RX 接收到得数据会被记录，PWM(analogWrite)值和引脚状态会保持，中断也会按设定的执行。

## 18.4 delayMicroseconds()

**描述**

使程序暂停指定的一段时间（单位：微秒）。一秒等于 1000000 微秒。目前，能够产生的最大的延时准确值是 16383。这可能会在未来的 Arduino 版本中改变。对于超过几千微秒的延迟，你应该使用 delay()代替。

**语法**

delayMicroseconds（us)

**参数**

us:暂停的时间，单位微秒（unsigned int）

**返回**

无

**例子**

```
int outPin = 8;                    // digital pin 8

void setup()
{
pinMode（outPin，OUTPUT）; //设置为输出的数字管脚
}

void loop()
{
digitalWrite（outPin，HIGH）; //设置引脚高电平
   delayMicroseconds(50);          // 暂停 50 微秒
   digitalWrite(outPin, LOW);      // 设置引脚低电平
   delayMicroseconds(50);          // 暂停 50 微秒
}
```

将 8 号引脚配置为输出脚。它会发出一系列周期 100 微秒的方波。

警告和已知问题

此函数在 3 微秒以以上工作的非常准确。我们不能保证，delayMicroseconds 在更小的

时间内延时准确。

Arduino0018 版本后，delayMicroseconds()不再会使中断失效。

# 十九、数学运算

## 19.1 min()

**min(x, y)**

**描述**

计算两个数字中的最小值。

**参数**

X：第一个数字，任何数据类型

Y：第二个数字，任何数据类型

**返回**

两个数字中的较小者。

**举例**

```
sensVal = min(sensVal，100); //将 sensVal 或 100 中较小者赋值给 sensVal
//确保它永远不会大于 100。
```

**注释**

直观的比较，max() 方法常被用来约束变量的下限，而 min() 常被用来约束变量的上限。

**警告**

由于 min() 函数的实现方式，应避免在括号内出现其他函数，这将导致不正确的结果。

```
min(a++, 100); //避免这种情况 - 会产生不正确的结果
a++;
min(a, 100); //使用这种形式替代 - 将其他数学运算放在函数之外
```

## 19.2 max()

**max(x,y)**

**描述**

计算两个数的最大值。

**参数**

X：第一个数字，任何数据类型

Y：第二个数字，任何数据类型

**返回**

两个参数中较大的一个。

**例子**

```
sensVal = max(senVal, 20); // 将 20 或更大值赋给 sensVal
//（有效保障它的值至少为 20）
```

**注意**

和直观相反，max()通常用来约束变量最小值，而 min()通常用来约束变量的最大值。

**警告**

由于 max()函数的实现方法，要避免在括号内嵌套其他函数，这可能会导致不正确的结果。

```
max(a--, 0);     //避免此用法，这会导致不正确结果
```

```
a--;                // 用此方法代替
max(a, 0);          // 将其他计算放在函数外
```

## 19.3 abs()

**ABS（X）**

**描述**

计算一个数的绝对值。

**参数**

X：一个数

**返回**

如果 x 大于或等于 0，则返回它本身。如果 x 小于 0，则返回它的相反数。

**警告**

由于实现 ABS（）函数的方法，避免在括号内使用任何函数（括号内只能是数字），否则将导致不正确的结果。

```
ABS（a+ +）；  //避免这种情况，否则它将产生不正确的结果
a ++;  //使用这段代码代替上述的错误代码
ABS（a）；  //保证其他函数放在括号的外部
```

## 19.4 constrain()

**描述**

将一个数约束在一个范围内

**参数**

x：要被约束的数字，所有的数据类型适用。

a：该范围的最小值，所有的数据类型适用。

b：该范围的最大值，所有的数据类型适用。

**返回值**

x：如果 x 是介于 a 和 b 之间

a：如果 x 小于 a

b：如果 x 大于 b

**例子**

```
sensVal = constrain(sensVal, 10, 150);
//传感器返回值的范围限制在 10 到 150 之间
```

## 19.5 map()

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

**描述**

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。

不限制值的范围，因为范围外的值有时是刻意的和有用的。如果需要限制的范围，constrain() 函数可以用于此函数之前或之后。

注意，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，例如:

```
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);

是有效的并且可以很好的运行。

map() 函数使用整型数进行运算因此不会产生分数，这时运算应该表明它需要这样做。小数的余数部分会被舍去，不会四舍五入或者平均。

**参数**

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

**返回值**

被映射的值。

**例子**

```
/*映射一个模拟值到 8 位（0 到 255）*/
void setup(){}

void loop()
{
int val = analogRead(0);
val = map(val, 0, 1023, 0, 255);
analogWrite(9, val);
}
```

**附录**

关于数学的实现，这里是完整函数

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

## 19.6 pow()

**pow(base, exponent)**

**描述**

计算一个数的幂次方。Pow()可以用来计算一个数的分数幂。这用来产生指数幂的数或曲线非常方便。

**参数**

base：底数（float）

exponent：幂（float）

**返回**

一个数的幂次方值（double）

**例子**

详情见库代码中的 fscale 函数。

## 19.7 sqrt()

**sqrt(x)**

**描述**

计算一个数的平方根。

**参数**

x：被开方数，任何类型

**返回值**

此数的平方根，类型 double

# 二十、三角函数

## 20.1 sin()

**sin(rad)**

**描述**

计算角度的正弦（弧度）。其结果在-1 和 1 之间。

**参数**

rad: 弧度制的角度（float）

**返回**

角度的正弦值（double）

## 20.2 cos()

**cos(rad)**

**描述**

计算一个角度的余弦值（用弧度表示）。返回值在 -1 和 1 之间。

**参数**

rad：用弧度表示的角度 (浮点数)

**返回**

角度的余弦值 (双精度浮点数)

## 20.3 tan()

**tan(rad)**

**描述**

计算角度的正切（弧度）。结果在负无穷大和无穷大之间。

**参数**

rad：弧度制的角度（float）

**返回值**

角度的正切值

# 二十一、随机数

## 21.1 randomSeed()

**randomSeed(seed)**

**描述**

使用 randomSeed()初始化伪随机数生成器，使生成器在随机序列中的任意点开始。这个

序列，虽然很长，并且是随机的，但始终是同一序列。

如需要在一个 random()序列上生成真正意义的随机数，在执行其子序列时使用 randomSeed()函数预设一个绝对的随机输入，例如在一个断开引脚上的 analogRead()函数的返回值。

反之，有些时候伪随机数的精确重复也是有用的。这可以在一个随机系列开始前，通过调用一个使用固定数值的 randomSeed()函数来完成。

**参数**

long,int - 通过数字生成种子。

**返回**

没有返回值

**例子**

```
long randNumber;
  void setup(){
    Serial.begin(9600);
    randomSeed(analogRead(0));
}

void loop(){
    randNumber = random(300);
    Serial.println(randNumber);

    delay(50);
}
```

## 21.2 random()

**random()**

**描述**

使用 random()函数将生成伪随机数。

**语法**

random(max)

random(min, max)

**参数**

min - 随机数的最小值，随机数将包含此值。（此参数可选）

max - 随机数的最大值，随机数不包含此值。

**返回**

min 和 max-1 之间的随机数（数据类型为 long ）

**注意**

如需要在一个 random()序列上生成真正意义的随机数，在执行其子序列时使用 randomSeed()函数预设一个绝对的随机输入，例如在一个断开引脚上的 analogRead()函数的返回值。

反之，有些时候伪随机数的精确重复也是有用的。这可以在一个随机系列开始前，通过调用一个使用固定数值的 randomSeed()函数来完成。

**例子**

```
long randNumber;

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

//如果模拟输入引脚 0 为断开，随机的模拟噪声
//将会调用 randomSeed()函数在每次代码运行时生成
//不同的种子数值。
//randomSeed()将随机打乱 random 函数。
  randomSeed(analogRead(0));
}

void loop() {
//打印一个 0 到 299 之间的随机数
  randNumber = random(300);
  Serial.println(randNumber);

//打印一个 10 到 19 之间的随机数
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

# 二十二、位操作

## 22.1 lowByte()

**描述**

提取一个变量（例如一个字）的低位（最右边）字节。

**语法**

lowByte(x)

**参数**

x:任何类型的值

**返回**

字节

## 22.2 highByte()

**描述**

提取一个字节的高位（最左边的），或一个更长的字节的第二低位。

**语法**

highByte(x)

**参数**

x：任何类型的值

**返回**

byte

## 22.3 bitRead()

**描述**

读取一个数的位。

**语法**

bitRead(x, n)

**参数**

X：想要被读取的数 N：被读取的位，0 是最低有效位（最右边）

**返回**

该位的值（0 或 1）。

## 22.4 bitWrite()

**描述**

在位上写入数字变量。

**语法**

bitWrite(x, n, b)

**参数**

X：要写入的数值变量

N：要写入的数值变量的位，从 0 开始是最低（最右边）的位

B：写入位的数值（0 或 1）

**返回**

无

## 22.5 bitSet()

**描述**

为一个数字变量设置一个位。

**语句**

bitSet(x, n)

**语法**

X：想要设置的数字变量

N：想要设置的位，0 是最重要（最右边）的位

**返回**

无

## 22.6 bitClear()

**描述**

清除一个数值型数值的指定位(将此位设置成 0)

**语法**

bitClear(x, n)

**参数**

X：指定要清除位的数值 N：指定要清除位的位置，从 0 开始，0 表示最右端位

**返回值**

无

### 22.7 bit()

**描述**

计算指定位的值（0 位是 1，1 位是 2，2 位 4，以此类推）。

**语法**

bit(n)

**参数**

n：需要计算的位

**返回值**

位值

## 二十三、设置中断函数

### 23.1 attachInterrupt()

**attachInterrupt(interrupt, function, mode)**

**描述**

当发生外部中断时，调用一个指定函数。当中断发生时，该函数会取代正在执行的程序。大多数的 Arduino 板有两个外部中断：0（数字引脚 2）和 1（数字引脚 3）。

arduino Mege 有四个外部中断：数字 2（引脚 21），3（20 针），4（引脚 19），5（引脚 18）。

**语法**

interrupt：中断引脚数

function：中断发生时调用的函数，此函数必须不带参数和不返回任何值。该函数有时被称为中断服务程序。

mode：定义何时发生中断以下四个 contstants 预定有效值：

LOW 当引脚为低电平时，触发中断

CHANGE 当引脚电平发生改变时，触发中断

RISING 当引脚由低电平变为高电平时，触发中断

FALLING 当引脚由高电平变为低电平时，触发中断.

**返回**

无

**注意事项**

当中断函数发生时，delay()和 millis()的数值将不会继续变化。当中断发生时，串口收到的数据可能会丢失。你应该声明一个变量来在未发生中断时储存变量。

**使用中断**

在单片机自动化程序中当突发事件发生时，中断是非常有用的，它可以帮助解决时序问题。一个使用中断的任务可能会读一个旋转编码器，监视用户的输入。

如果你想以确保程序始终抓住一个旋转编码器的脉冲，从来不缺少一个脉冲，它将使写一个程序做任何事情都要非常棘手，因为该计划将需要不断轮询的传感器线编码器，为了赶上脉冲发生时。其他传感器也是如此，如试图读取一个声音传感器正试图赶上一按，或红外线槽传感器（照片灭弧室），试图抓住一个硬币下降。在所有这些情况下，使用一个中断可以释放的微控制器来完成其他一些工作。

**程序示例**

```
int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
  state = !state;
}
```

### 23.2 detachInterrupt()

**detachInterrupt(interrupt)**

**描述**

关闭给定的中断。

**参数**

interrupt: 中断禁用的数(0 或者 1).

## 二十四、开关中断

### 24.1 interrupts()（中断）

**描述**

重新启用中断（使用 noInterrupts()命令后将被禁用）。中断允许一些重要任务在后台运行，默认状态是启用的。禁用中断后一些函数可能无法工作，并传入信息可能会被忽略。中断会稍微打乱代码的时间，但是在关键部分可以禁用中断。

**参数**

无

**返回**

无

**例子**

```
void setup() {
}

void loop()
```

```
{
  noInterrupts（）;
  //重要、时间敏感的代码
  interrupts();
  //其他代码写在这里
}
```

## 24.2 noInterrupts()（禁止中断）

**描述**

禁止中断（重新使能中断 interrupts()）。中断允许在后台运行一些重要任务，默认使能中断。禁止中断时部分函数会无法工作，通信中接收到的信息也可能会丢失。

中断会稍影响计时代码，在某些特定的代码中也会失效。

**参数**

无

**返回**

无

**例子**

```
void setup()

void loop()
{
noInterrupts（）;
//关键的、时间敏感的代码放在这
  interrupts();
//其他代码放在这
}
```

# 二十五、通讯

## 25.1 Serial

用于 Arduino 控制板和一台计算机或其他设备之间的通信。所有的 Arduino 控制板有至少一个串口(又称作为 UART 或 USART)。它通过 0(RX)和 1(TX)数字引脚经过串口转换芯片连接计算机 USB 端口与计算机进行通信。因此，如果你使用这些功能的同时你不能使用引脚 0 和 1 作为输入或输出。

您可以使用 Arduino IDE 内置的串口监视器与 Arduino 板通信。点击工具栏上的串口监视器按钮，调用 begin()函数（选择相同的波特率）。

Arduino Mega 有三个额外的串口：Serial 1 使用 19(RX)和 18(TX)，Serial 2 使用 17(RX)和 16(TX)，Serial3 使用 15(RX)和 14(TX)。若要使用这三个引脚与您的个人电脑通信，你需要一个额外的 USB 转串口适配器，因为这三个引脚没有连接到 Mega 上的 USB 转串口适配器。若要用它们来与外部的 TTL 串口设备进行通信，将 TX 引脚连接到您的设备的 RX 引脚，将 RX 引脚连接到您的设备的 TX 引脚，将 GND 连接到您的设备的 GND。(不要直接将这些引脚直接连接到 RS232 串口;他们的工作电压在+/- 12V，可能会损坏您的 Arduino 控制板。)

Arduino Leonardo 板使用 Serial 1 通过 0(RX)和 1(TX)与 viaRS-232 通信，。Serial 预留给使

用 Mouse and Keyboard libarariies 的 USB CDC 通信。更多信息，请参考 Leonardo 开始使用页和硬件页。

**函数**



### 25.1.1 if (Serial)

**说明**

表示指定的串口是否准备好。

在 Leonardo 上，if(Serial)表示不论有无 USB CDC，串行连接都是开放的。对于所有其他的情况，包括 Leonardo 上的 if(Serial1)，将一直返回 true。这来自于 Arduino 1.0.1 版本的介绍.

**语法**

对于所有的 arduino 板:

if (Serial)

Arduino Leonardo 特有:

if (Serial1)

Arduino Mega 特有:

if (Serial1)

if (Serial2)

if (Serial3)

**参数**

无

**返回**

布尔值：如果指定的串行端口是可用的，则返回 true。如果查询 Leonardo 的 USB CDC 串行连接之前，它是准备好的，将只返回 false。

**例子**

```
void setup() {
```

```
  //初始化串口和等待端口打开：
  Serial.begin(9600);
  while (!Serial) {
//等待串口连接。只有 Leonardo 需要。
  }
}

void loop() {
  //正常进行
}
```

**25.1.2 Serial.available()**

**说明**

获取从串口读取有效的字节数（字符）。这是已经传输到，并存储在串行接收缓冲区（能够存储 64 个字节）的数据。 available()继承了 Stream 类。

**语法**

Serial.available()

**仅适用于 Arduino Mega ：**

Serial1.available()

Serial2.available()

Serial3.available()

**参数**

无

**返回**

可读取的字节数

**例子**

```
incomingByte = 0; //传入的串行数据
 void setup() {
  Serial.begin(9600);       // 打开串行端口，设置传输波特率为 9600 bps
}

void loop() {

  //只有当你接收到数据时才会发送数据，：
  if (Serial.available() > 0) {
    //读取传入的字节：
    incomingByte = Serial.read();

    //显示你得到的数据：
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

**Arduino Mega 的例子:**

```
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);

}

void loop() {
  //读取端口 0，发送到端口 1：
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, BYTE);

  }
  //读端口 1，发送到端口 0：
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}
```

**25.1.3 Serial.begin()**

**说明**

将串行数据传输速率设置为位/秒（波特）。与计算机进行通信时，可以使用这些波特率：300，1200，2400，4800，9600，14400，19200，28800，38400，57600 或 115200。当然，您也可以指定其他波特率 - 例如，引脚 0 和 1 和一个元件进行通信，它需要一个特定的波特率。

**语法**

Serial.begin(speed) 仅适用于 Arduino Mega ： Serial1.begin(speed) Serial2.begin(speed) Serial3.begin(speed)

**参数**

speed: 位/秒 (波特) - long

**返回**

无

**例子**

```
void setup() {
    Serial.begin(9600); // 打开串口，设置数据传输速率为 9600bps
}

void loop() {
```

**Arduino Mega 的例子:**

```
// Arduino Mega 可以使用四个串口
// (Serial, Serial1, Serial2, Serial3),
// 从而设置四个不同的波特率：
```

```
void setup(){
  Serial.begin(9600);
  Serial1.begin(38400);
  Serial2.begin(19200);
  Serial3.begin(4800);

  Serial.println("Hello Computer");
  Serial1.println("Hello Serial 1");
  Serial2.println("Hello Serial 2");
  Serial3.println("Hello Serial 3");
}

void loop() {}
```

### 25.1.4 Serial.end()

**说明**

停用串行通信，使 RX 和 TX 引脚用于一般输入和输出。要重新使用串行通信，需要 Serial.begin()语句。

**语法**

Serial.end()

仅适用于 Arduino Mega: Serial1.end() Serial2.end() Serial3.end()

**参数**

无

**返回**

无

### 25.1.5 Serial.find()

**说明**

Serial.find() 从串行缓冲器中读取数据，直到发现给定长度的目标字符串。如果找到目标字符串，该函数返回 true，如果超时则返回 false。

Serial.flush() 继承了 Stream 类.

**语法**

Serial.find(target)

**参数**

target：要搜索的字符串（字符）

**返回**

布尔型

### 25.1.6 Serial.findUntil()

**说明**

Serial.findUntil()从串行缓冲区读取数据，直到找到一个给定的长度或字符串终止位。

如果目标字符串被发现，该函数返回 true，如果超时则返回 false。

Serial.findUntil()继承了 Stream 类。

**语法**

Serial.findUntil(target, terminal)

**参数**

target： 要搜索的字符串(char) terminal： 在搜索中的字符串终止位 (char)

**返回**

布尔型

### 25.1.7 Serial.flush()

**说明**

等待超出的串行数据完成传输。(在 1.0 及以上的版本中，flush()语句的功能不再是丢弃所有进入缓存器的串行数据。)

flush()继承了 Stream 类.

**语法**

Serial.flush()

**仅 Arduino Mega 可以使用的语法：**

Serial1.flush()

Serial2.flush()

Serial3.flush()

**参数**

无

**返回**

无

### 25.1.8 Serial.parseFloat()

**描述**

Serial.parseFloat()命令从串口缓冲区返回第一个有效的浮点数. Characters that are not digits (or the minus sign) are skipped. parseFloat() is terminated by the first character that is not a floating point number.

Serial.parseFloat()继承了 Stream 类。

**语法**

Serial.parseFloat()

**参数**

无

**返回**

float

### 25.1.9 Serial.parseInt()

**说明**

查找传入的串行数据流中的下一个有效的整数。 parseInt()继承了 Stream 类。

**语法**

Serial.parseInt()

**下面三个命令仅适用于 Arduino Mega：**

Serial1.parseInt()

Serial2.parseInt()

Serial3.parseInt()

**参数**

无

**返回**

int： 下一个有效的整数

### 25.1.10 Serial.peek()

**说明**

返回传入的串行数据的下一个字节（字符），而不是进入内部串行缓冲器调取。也就是说，连续调用 peek()将返回相同的字符，与调用 read()方法相同。peek()继承自 Stream 类。

**语法**

Serial.peek()

**仅适用于 Arduino Mega :**

Serial1.peek()

Serial2.peek()

Serial3.peek()

**参数**

无

**返回**

传入的串行数据的第一个字节（或-1，如果没有可用的数据的话）- int

### 25.1.11 Serial.print()

**说明**

以人们可读的 ASCII 文本形式打印数据到串口输出。此命令可以采取多种形式。每个数字的打印输出使用的是 ASCII 字符。浮点型同样打印输出的是 ASCII 字符，保留到小数点后两位。Bytes 型则打印输出单个字符。字符和字符串原样打印输出。Serial.print()打印输出数据不换行，Serial.println()打印输出数据自动换行处理。例如

Serial.print(78)输出为“78”

Serial.print(1.23456)输出为“1.23”

Serial.print(“N”)输出为“N”

Serial.print(“Hello world.”)输出为“Hello world.”

也可以自己定义输出为几进制（格式）；可以是 BIN（二进制，或以 2 为基数），OCT（八进制，或以 8 为基数），DEC（十进制，或以 10 为基数），HEX（十六进制，或以 16 为基数）。对于浮点型数字，可以指定输出的小数数位。例如

Serial.print(78,BIN)输出为“1001110”

Serial.print(78,OCT)输出为“116”

Serial.print(78,DEC)输出为“78”

Serial.print(78,HEX)输出为“4E”

Serial.println(1.23456,0)输出为“1”

Serial.println(1.23456,2)输出为“1.23”

Serial.println(1.23456,4)输出为“1.2346”

你可以通过基于闪存的字符串来进行打印输出，将数据放入 F()中，再放入 Serial.print()。例如 Serial.print(F(“Hello world”)) 若要发送一个字节，则使用 Serial.write()。

**语法**

Serial.print(val)

Serial.print(val，格式)

**参数**

val：打印输出的值 - 任何数据类型

格式：指定进制（整数数据类型）或小数位数（浮点类型）

**返回**

字节 print()将返回写入的字节数，但是否使用（或读出）这个数字是可设定的

**例子**

```
/ *
使用 for 循环打印一个数字的各种格式。
* /
int x = 0;       // 定义一个变量并赋值

void setup() {
  Serial.begin(9600);          // 打开串口传输，并设置波特率为 9600
}

void loop() {
  / /打印标签
  Serial.print("NO FORMAT");           // 打印一个标签
  Serial.print("\t");                  // 打印一个转义字符
  Serial.print("DEC");
  Serial.print("\t");
  Serial.print("HEX");
  Serial.print("\t");
  Serial.print("OCT");
  Serial.print("\t");
  Serial.print("BIN");
  Serial.print("\t");
  for(x=0; x< 64; x++){       // 打印 ASCII 码表的一部分，修改它的格式得到需要的内容
    / /打印多种格式：
      Serial.print(x);           // 以十进制格式将 x 打印输出 - 与 "DEC"相同
    Serial.print("\t");       // 横向跳格
    Serial.print(x, DEC);    // 以十进制格式将 x 打印输出
    Serial.print("\t");       // 横向跳格
    Serial.print(x, HEX);    // 以十六进制格式打印输出
    Serial.print("\t");       // 横向跳格
    Serial.print(x, OCT);    // 以八进制格式打印输出
    Serial.print("\t");       // 横向跳格
    Serial.println(x, BIN);   // 以二进制格式打印输出
  //然后用 "println"打印一个回车
    delay(200);                  // 延时 200ms
  }
  Serial.println("");          // 打印一个空字符，并自动换行
}
```

编程技巧作为 1.0 版本，串行传输是异步的; Serial.print()将返回之前接收到的任何字符。

### 25.1.12 Serial.println()

**说明**

打印数据到串行端口，输出人们可识别的 ASCII 码文本并回车 (ASCII 13, 或 '\r') 及换行 (ASCII 10, 或 '\n')。此命令采用的形式与 Serial.print ()相同。

**语法**

Serial.println(val)

Serial.println(val, format)

**参数**

val: 打印的内容 - 任何数据类型都可以

format: 指定基数（整数数据类型）或小数位数（浮点类型）

**返回**

字节（byte）

println()将返回写入的字节数，但可以选择是否使用它。

**例子**

```
/*
模拟输入信号
读取模拟口 0 的模拟输入，打印输出读取的值。
由 Tom Igoe 创建于 2006 年 3 月 24 日
 */
int analogValue = 0;      // 定义一个变量来保存模拟值

void setup() {
  //设置串口波特率为 9600 bps：
  Serial.begin(9600);
}

void loop() {
  analogValue = analogRead（0）;  //读取引脚 0 的模拟输入：

  //打印 g 各种格式：
  Serial.println(analogValue);          //打印 ASCII 编码的十进制
  Serial.println(analogValue, DEC);   //打印 ASCII 编码的十进制
  Serial.println(analogValue, HEX);   //打印 ASCII 编码的十六进制
  Serial.println(analogValue, OCT);   //打印 ASCII 编码的八进制
  Serial.println(analogValue, BIN);  //打印一个 ASCII 编码的二进制
  delay(10);                                 // 延时 10 毫秒：
}
```

### 25.1.13 Serial.read()

**说明**

读取传入的串口的数据。read() 继承自 Stream 类。

**语法**

serial.read()

**Arduino Mega 独有:**

serial1.read()

serial2.read()

serial3.read()

**参数**

无

**返回**

传入的串口数据的第一个字节（或-1，如果没有可用的数据）- int

**例子**

```
int incomingByte = 0;    // 传入的串行数据
void setup() {
  Serial.begin(9600);       // 打开串口，设置数据传输速率 9600
}

void loop() {

  // 当你接收数据时发送数据
  if (Serial.available() > 0) {
    // 读取传入的数据:
    incomingByte = Serial.read();

    //打印你得到的：
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

### 25.1.14 Serial.readBytes()

**说明**

Serial.readBytes()从串口读字符到一个缓冲区。如果预设的长度读取完毕或者时间到了(参见 Serial.setTimeout())，函数将终止.

Serial.readBytes()返回放置在缓冲区的字符数。返回 0 意味着没有发现有效的数据。

Serial.readBytes()继承自 Stream 类.

**语法**

Serial.readBytes(buffer, length)

**参数**

buffer：用来存储字节（char[]或 byte[]）的缓冲区

length：读取的字节数（int）

**返回**

byte

### 25.1.15Serial.readBytesUntil()

**说明**

Serial.readBytesUntil()将字符从串行缓冲区读取到一个数组。如果检测到终止字符，或预设的读取长度读取完毕，或者时间到了 (参见 Serial.setTimeout())函数将终止。

Serial.readBytesUntil()返回读入数组的字符数。返回 0 意味着没有发现有效的数据。

Serial.readBytesUntil()继承自 Stream 类。

**语法**

Serial.readBytesUntil(character, buffer, length)

**参数**

character ：要搜索的字符（char）

buffer ：缓冲区来存储字节（char[]或 byte[]）

length:读的字节数（int）

**返回**

byte

### 25.1.16 Serial.setTimeout()

**说明**

Serial.setTimeout()设置使用 Serial.readBytesUntil() 或 Serial.readBytes()时等待串口数据的最大毫秒值. 默认为 1000 毫秒。

Serial.setTimeout()继承自 Stream 类。

**语法**

Serial.setTimeout(time)

**参数**

time ：以毫秒为单位的超时时间（long）。

**返回**

无

### 25.1.17 Serial.write()

**说明**

写入二级制数据到串口。发送的数据以一个字节或者一系列的字节为单位。如果写入的数字为字符，需使用 print()命令进行代替。

**语法**

Serial.write(val)

Serial.write(str)

Serial.write(buf, len)

Arduino Mega 还支持：Serial1，Serial2，Serial3 （替代 Serial）

**参数**

val: 以单个字节形式发的值

str: 以一串字节的形式发送的字符串

buf: 以一串字节的形式发送的数组

len: 数组的长度

**返回**

byte

write() 将返回写入的字节数，但是否使用这个数字是可选的

**例子**

```
void setup(){
```

```
  Serial.begin(9600);
}

void loop(){
  Serial.write(45); // 发送一个值为 45 的字节
 int bytesSent = Serial.write(“hello”); //发送字符串“hello”，返回该字符串的长度.
}
```

#### 25.1.18 Serial.SerialEvent()

暂无说明。

### 25.2 Stream

暂无说明。

## 二十六、USB（仅适用于 Leonardo 和 Due）

### 26.1 Mouse（键盘）

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

### 26.2 Keyboard（鼠标）

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

本文内容来自：

http://wiki.geek-workshop.com/doku.php?id=arduino:arduino_language_reference