

# Arduino编程参考手册

来自DFRobot Product Wiki

首页

## 程序结构

setup()  
loop()  
  
控制结构  
if  
if...else  
for  
switch case  
while  
do...while  
break  
continue  
return  
goto  
  
相关语法  
  ; 分号  
{ 大括号  
// 单行注释  
/\*\*/ 多行注释  
#define 宏定义  
#include 文件包含  
  
算术运算符  
= 赋值  
+ (加)  
- (减)  
\* (乘)  
/ (除)  
% (取模)  
  
比较运算符  
== 等于  
!= (不等于)  
< (小于)  
> (大于)  
<= (小于等于)  
>= (大于等于)  
  
布尔运算符  
&& (逻辑与)  
|| (逻辑或)  
! (逻辑非)  
  
指针运算符  
\* 指针运算符  
& 地址运算符  
  
位运算  
& (位与)  
| (位或)  
^ (位异或)  
~ (位非)  
<< (左移)  
>> (右移)  
  
复合运算符  
++ (自加)  
-- (自减)  
+= (复合加)  
-= (复合减)  
\*= (复合乘)  
/= (复合除)  
&= (复合与)  
|= (复合或)

## 变量

范围  
HIGH | LOW  
INPUT | OUTPUT  
true | false  
整型常量  
浮点数常量  
  
数据类型  
void  
boolean  
char  
unsigned char  
byte  
int  
unsigned int  
word  
long  
unsigned long  
float  
double  
string  
String(c++)  
array  
  
数据类型转换  
char()  
byte()  
int()  
word()  
long()  
float()  
  
变量作用域  
变量作用域  
static (静态变量)  
volatile (易变变量)  
const (不可改变变量)  
  
辅助工具  
sizeof() (sizeof运算符)  
ASCII码表

## 基本函数

数字I/O  
pinMode()  
digitalWrite()  
digitalRead()  
  
模拟I/O  
analogReference()  
analogRead()  
analogWrite()  
  
指高级I/O  
shiftOut()  
pulseIn()  
  
时间  
millis()  
delay(ms)  
delayMicroseconds(us)  
  
数学库  
min()  
max()  
abs()  
constrain()  
map()  
pow()  
sqrt()  
  
三角函数  
sin(rad)  
cos(rad)  
tan(rad)  
  
随机数  
randomSeed()  
random()  
random()  
  
位操作  
lowByte()  
highByte()  
bitRead()  
bitWrite()  
bitSet()  
bitClear()  
bit()  
  
设置中断函数  
attachInterrupt()  
detachInterrupt()  
interrupts()  
noInterrupts()  
  
串口通讯  
begin()  
available()  
read()  
flush  
print()  
println()  
write()  
peak()  
serialEvent()

## 程序结构

(本节直译自Arduino官网最新Reference)

在Arduino中，标准的程序入口main函数在内部被定义，用户只需要关心以下两个函数：

`setup()`

当Arduino板启动时`setup()`函数会被调用。用它来初始化变量，引脚模式，开始使用某个库，等等。该函数在Arduino板的每次上电和复位时只运行一次。

`loop()`

在创建`setup`函数，该函数初始化和设置初始值，`loop()`函数所做事的正如其名，连续循环，允许你的程序改变状态和响应事件。可以用它来实时控制arduino板。

示例：

```
int buttonPin = 3;

void setup()
{
  Serial.begin(9600); //初始化串口
  pinMode(buttonPin, INPUT); //设置3号引脚为输入模式
}

void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');

  delay(1000);
}
```

■ [返回顶部](#)

## 控制语句

`if`

`if`，用于与比较运算符结合使用，测试是否已达到某些条件，例如一个输入数据在某个范围之外。使用格式如下：

```
if (value > 50)
{
  // 这里加入你的代码
}
```

该程序测试`value`是否大于50。如果是，程序将执行特定的动作。换句话说，如果圆括号中的语句为真，大括号中的语句就会执行。如果不是，程序将跳过这段代码。大括号可以被省略，如果这么做，下一行（以分号结尾）将成为唯一的条件语句。

```
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);
  // 都是正确的
}
```

圆括号中要被计算的语句需要一个或多个操作符。

■ [返回顶部](#)

`if...else`

与基本的`if`语句相比，由于允许多个测试组合在一起，`if/else`可以使用更多的控制流。例如，可以测试一个模拟量输入，如果输入值小于500，则采取一个动作，而如果输入值大于或等于500，则采取另一个动作。代码看起来像是这样：

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

else中可以进行另一个if测试，这样多个相互独立的测试就可以同时进行。每一个测试一个接一个地执行直到遇到一个测试为真为止。当发现一个测试条件为真时，与其关联的代码块就会执行，然后程序将跳到完整的if/else结构的下一行。如果没有一个测试被验证为真。缺省的else语句块，如果存在的话，将被设为默认行为，并执行。

注意：一个else if语句块可能有或者没有终止else语句块，同理。每个else if分支允许有无限多个。

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

另外一种表达互斥分支测试的方式，是使用switch case语句。

- [返回顶部](#)

for

for语句

描述

for语句用于重复执行被花括号包围的语句块。一个增量计数器通常被用来递增和终止循环。for语句对于任何需要重复的操作是非常有用的。常常用于与数组联合使用以收集数据/引脚。for循环的头部有三个部分：

```
for (初始化部分; 条件判断部分; 数据递增部分) {
//语句块
  ..
}
```

初始化部分被第一个执行，且只执行一次。每次通过这个循环，条件判断部分将被测试;如果为真，语句块和数据递增部分就会被执行,然后条件判断部分就会被再次测试，当条件测试为假时，结束循环。

示例：

```
//使用一个PWM引脚使LED灯闪烁
int PWMpin = 10; // LED在10号引脚串联一个470欧姆的电阻

void setup()
{
  //这里无需设置
}

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

编码提示：

C中的for循环比在其它计算机语言中发现的for循环要灵活的多，包括BASIC。三个头元素中的任何一个或全部可能被省略，尽管分号是必须的。而且初始化部分、条件判断部分和数据递增部分可以是任何合法的使用任意变量的C语句。且可以使用任何数据类型包括floats。这些不常用的类型用于语句段也许可以为一些罕见的编程问题提供解决方案。

例如，在递增部分中使用一个乘法将形成对数级增长：

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

输出：2, 3, 4, 6, 9, 13, 19, 28, 42, 63, 94

另一个例子，在一个for循环中使一个LED灯渐渐地变亮和变暗：

```
void loop()
{
  int x = 1;
  for (int i = 0; i > -1; i = i + x){
```

```
analogWrite(PWMPin, i);
if (i == 255) x = -1;      // 在峰值切换方向
delay(10);
}
```

▪ 返回顶部

switch case

switch case 语句

就像if语句，switch...case通过允许程序员根据不同的条件指定不同的应被执行的代码来控制程序流。特别地，一个switch语句对一个变量的值与case语句中指定的值进行比较。当一个case语句被发现其值等于该变量的值。就会运行这个case语句下的代码。

break关键字将中止并跳出switch语句段，常常用于每个case语句的最后面。如果没有break语句，switch语句将继续执行下面的表达式（“持续下降”）直到遇到break，或者是到达switch语句的末尾。

示例:

```
switch (var) {
  case 1:
    //当var等于1执行这里
    break;
  case 2:
    //当var等于2执行这里
    break;
  default:
    // 如果没有匹配项，将执行此缺省段
    // default段是可选的
}
```

语法

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

参数

var: 与不同的case中的值进行比较的变量

label: 相应的case的值

▪ 返回顶部

while

while循环

描述:

while循环将会连续地无限地循环，直到圆括号（）中的表达式变为假。被测试的变量必须被改变，否则while循环将永远不会中止。这可以是你的代码，比如一个递增的变量，或者是一个外部条件，比如测试一个传感器。

语法:

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

参数:

expression - 一个（布尔型）C语句，被求值为真或假

示例:

```
var = 0;
while(var < 200){
  // 做两百次重复的事情
  var++;
}
```

- [返回顶部](#)

do...while

do循环

do循环与while循环使用相同方式工作，不同的是条件是在循环的末尾被测试的，所以do循环总是至少会运行一次。

```
do
{
    // 语句块
    while (测试条件);
```

示例:

```
do
{
    delay(50);           // 等待传感器稳定
    x = readSensors();   // 检查传感器的值
    while (x < 100);
```

- [返回顶部](#)

break

break用于中止do，for，或while循环，绕过正常的循环条件。它也用于中止switch语句。

示例:

```
for (x = 0; x < 255; x ++){
{
    digitalWrite(PWMPin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){ // bail out on sensor detect
        x = 0;
        break;
    }
    delay(50);
}
```

- [返回顶部](#)

continue

continue语句跳过一个循环的当前迭代的余下部分。（do，for，或while）。通过检查循环测试条件它将继续进行随后的迭代。

示例:

```
for (x = 0; x < 255; x ++){
{
    if (x > 40 && x < 120){ // create jump in values
        continue;
    }

    digitalWrite(PWMPin, x);
    delay(50);
}
```

- [返回顶部](#)

return

终止一个函数，并向被调用函数并返回一个值，如果你想的话。

语法:

```
return;
return value; // both forms are valid
```

参数:

value: 任何类型的变量或常量

示例:

```
//一个函数，用于对一个传感器输入与一个阈值进行比较

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

return 关键字对测试一段代码很方便，不需“注释掉”大段的可能是错误的代码。

```
void loop() {

//在此测试代码是个好想法

return;

// 这里是功能不正常的代码
// 这里的代码永远也不会执行
}
```

- [返回顶部](#)

goto

在程序中转移程序流到一个标记点

语法:

```
label:
goto label; // sends program flow to the label
```

提示:

在C程序中不建议使用goto，而且一些C编程书的作者主张永远不要使用goto语句，但是明智地使用它可以 简化某些代码。许多程序员不赞成使用goto的原因是，无节制地使用goto语句很容易产生执行流混乱的很难被调试程序。 尽管如是说，仍然有很多使用goto语句而大大简化编码的实例。其中之一就是从 一个很深的循环嵌套中跳出去，或者是if逻辑块，在 某人些条件下。

示例:

```
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;}
            // 其它语句。。。
        }
    }
}
bailout:
```

- [返回顶部](#)

相关语法

分号

用于一个语句的结束

示例

```
int a = 13;
```

提示

忘记在一行的末尾加一个分号将产生一个编译器错误。该错误信息可能是明显的，且会提及丢失分号，但也许不会。如果出现一个不可理喻的或看起来不合逻辑的错误，其中一个首先要做的事就是检查分号丢失。编译器会在前一行的附近发出抱怨。

- [返回顶部](#)

## 大括号

大括号（又称括弧或花括号）是C语言的主要组成部分。它们用在几个不同的结构中，大致如下，这可能会令初学者感到困惑。

一个左大括号必须有一个右大括号跟在后面。这是一个常被称为平衡括号的条件。Arduino IDE（集成开发环境）包含一个方便的特性以检验平衡大括号。只需选择一个大括号，甚至直接在一个大括号后面点击插入点，然后它的逻辑上的同伴就会高亮显示。

目前此功能有些许错误，因为IDE经常在文本中（错误地）发现一个已经被注释掉的大括号。

初级程序员，和从BASIC转到C的程序员常常发现使用大括号令人困惑或畏缩。毕竟，用同样的大括号在子例程（函数）中替换RETURN语句，在条件语句中替换ENDIF语句和在FOR循环中替换NEXT语句。

由于大括号的使用是如此的多样，当插入一个需要大括号的结构时，直接在打出开括号之后打出闭括号是个不错的编程实践。然后在大括号之间插入一些回车符，接着开始插入语句。你的大括号，还有你的态度，将永远不会变得不平衡。

不平衡的大括号常常导致古怪的，难以理解的编译器错误，有时在大型程序中很难查出。因为它们的多样的使用，大括号对于程序的语法也是极其重要的，对一个大括号移动一行或两行常常显著地影响程序的意义。

大括号的主要用法

```
//函数
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)
}
```

### ■ 返回顶部

## 注释

注释是程序中的一些行，用于让自己或他人了解程序的工作方式。他们会被编译器忽略，而不会输出到控制器，所以它们不会占用Atmega芯片上的任何空间。

注释唯一的目的是帮助你理解（或记忆）你的程序是怎样工作的，或者是告知其他人你的程序是怎样工作的。标记一行为注释只有两种方式：

示例

```
x = 5; //这是一个单行注释。此斜线后的任何内容都是注释
      //直到该行的结尾

/* 这是多行注释 - 用它来注释掉整个代码块
if (gwb == 0){ //在多行注释中使用单行注释是没有问题的
x = 3;        /* 但是其中不可以使用另一个多行注释 - 这是不合法的 */
}

//别忘了加上“关闭”注释符 - 它们必须是平衡的
*/
```

### 提示

当实验代码时，“注释掉”你的程序的一部分来移除可能是错误的行是一种方便的方法。这不是把这些行从程序中移除，而是把它们放到注释中，所以编译器就会忽略它们。这在定位问题时，或者当程序无法编译通过且编译错误信息很古怪或没有帮助时特别有用。

- [返回顶部](#)

## define

### #define 宏定义

宏定义是一个有用的C组件，它允许程序员在程序编译前给常量取一个名字。在arduino中定义的常量不会在芯片中占用任何程序空间。编译器在编译时会将这些常量引用替换为定义的值。

这虽然可能有些有害的副作用，举例来说，一个已被定义的常量名被包含在一些其它的常量或变量名中。那样的话该文本将被替换成被定义的数字（或文本）。

通常，用const关键字定义常量是更受欢迎的且用来代替#define会很有用。

Arduino宏定义与C宏定义有同样的语法

### 语法

```
#define constantName value
```

注意‘#’是必须的

示例:

```
#define ledPin 3
// 编译器在编译时会将任何提及ledPin的地方替换成数值3。
```

### 提示

#define语句的后面分号。如果你加了一个，编译器将会在进一步的页面引发奇怪的错误。

```
#define ledPin 3; // this is an error
```

类似地，包含一个等号通常也会在进一步的页面引发奇怪的编译错误。

```
#define ledPin = 3 // this is also an error
```

- [返回顶部](#)

## include

### #include 包含

#include用于在你的sketch中包含外部的库。这使程序员可以访问一个巨大的标准C库（预定义函数集合）的集合。

AVR C库（AVR是Atmel芯片的一个基准，Arduino正是基于它）的主参考手册页在这里。

注意#include和#define相似，没有分号终止符，且如果你加了，编译器会产生奇怪的错误信息。

示例

该示例包含一个用于输出数据到程序空间闪存的库，而不是内存。这会为动态内存需求节省存储空间且使需要创建巨大的查找表变得更实际。

```
#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};
```

- [返回顶部](#)



## 算术运算符

### 赋值

=赋值运算符（单个等号）

把等号右边的值存储到等号左边的变量中。

在C语言中单个等号被称为赋值运算符。它与在代数课中的意义不同，后者象征等式或相等。赋值运算符告诉微控制器求值等号右边的变量或表达式，然后把结果存入等号左边的变量中。

### 示例

```
int sensVal;           //声明一个名为sensVal的整型变量
sensVal = analogRead(0); //存储（数字的）0号模拟引脚的输入电压值到sensVal
```

### 编程技巧

赋值运算符（=号）左边的变量需要能够保存存储在其中的值。如果它不足以大到容纳一个值，那个存储在该变量中的值将是错误的。

不要混淆赋值运算符[ = ]（单个等号）和比较运算符[ == ]（双等号），后者求值两个表达式是否相等。

- [返回顶部](#)

## 加，减，乘，除

### 描述

这些运算符（分别）返回两人运算对象的和，差，积，商。这些操作受运算对象的数据类型的影响。所以，例如，9 / 4 结果是2，如果9和2是整型数。这也意味着运算会溢出，如果结果超出其在相应的数据类型下所能表示的数。（例如，给整型数值32767加1结果是-32768）。如果运算对象是不同的类型，会用那个较大的类型进行计算。

如果其中一个数字（运算符）是float类型或double类型，将采用浮点数进行计算。

### 示例

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

### 语法

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

### 参数:

value1:任何变量或常量

value2:任何变量或常量

### 编程技巧:

要知道整型常量默认为int型，因此一些常量计算可能会溢出（例如：60 \* 1000将产生负的结果）

选择一个大小足够大的变量以容纳你的最大的计算结果。

要知道你的变量在哪一点将会“翻转”且要知道在另一个方向上会发生什么，例如：（0 - 1）或（0 - 32768）。

对于数学需要分数，就使用浮点变量，但是要注意它们的缺点：占用空间大，计算速度慢。

使用强制类型转换符例如：（int）myFloat以在运行中转换一个变量到另一个类型。

- [返回顶部](#)

## 取模

% (取模)

### 描述

计算一个数除以另一个数的余数。这对于保持一个变量在一个特定的范围很有用（例如：数组的大小）。

### 语法

```
result = dividend % divisor
```

### 参数

dividend: 被除数

divisor: 除数

结果: 余数

### 示例

```
x = 7 % 5; // x now contains 2
x = 9 % 5; // x now contains 4
x = 5 % 5; // x now contains 0
x = 4 % 5; // x now contains 4
```

### 示例代码

```
/* update one value in an array each time through a loop */
int values[10];
int i = 0;

void setup() {}

void loop()
{
  values[i] = analogRead(0);
  i = (i + 1) % 10; // modulo operator rolls over variable
}
```

### 提示:

取模运算符不能用于浮点型数。

- [返回顶部](#)

## 比较运算符

if(条件) and ==, !=, <, > (比较运算符)

if, 用于和比较运算符联合使用, 测试某一条件是否到达, 例如一个输入超出某一数值。if条件测试的格式:

```
if (someVariable > 50)
{
  // do something here
}
```

该程序测试someVariable是否大于50。如果是, 程序执行特定的动作。换句话说, 如果圆括号中的语句为真, 花括号中的语句就会运行。否则, 程序跳过该代码。

if语句后的花括号可能被省略。如果这么做了, 下一行(由分号定义的行)就会变成唯一的条件语句。

```
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);
    // all are correct
}
```

圆括号中被求值的语句需要使用一个或多个运算符：

比较运算符：

```
x == y (x is equal to y)
x != y (x is not equal to y)
x < y (x is less than y)
x > y (x is greater than y)
x <= y (x is less than or equal to y)
x >= y (x is greater than or equal to y)
```

警告：

小心偶然地使用单个等号（例如if(x = 10)）。单个等号是赋值运算符，这里设置x为10（将值10存入变量x）。改用双等号（例如if (x == 10)），这个是比较运算符，用于测试x是否等于10。后者只在x等于10时返回真，但是前者将总是为真。

这是因为C如下求值语句if(x=10)：10分配给x（切记单个等号是赋值运算符），因此x现在为10。然后'if'条件求值10，其总是为真，由于任何非零数值都为真值。由此，if (x = 10)将总是求值为真，这不是使用if语句所期望的结果。另外，变量x将被设置为10，这也不是期望的操作。

if也可以是使用[if...else]的分支控制结构的一部分。

- [返回顶部](#)

## 布尔运算符

它们可用于if语句中的条件

&& （逻辑与）

只有在两个操作数都为真时才返回真，例如：

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) { // read two switches
    // ...
}
```

只在两个输入都为高时返回真

|| （逻辑或）

任意一个为真时返回真，例如：

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

x或y任意一个大于0时返回真

! （非）

当操作数为假时返回真，例如：

```
if (!x) {
    // ...
}
```

若x为假返回真（即如果x等于0）

警告

确保你没有把布尔与运算符，&&（两个与符号）错认为按位与运算符&（单个与符号）。它们是完全不同的概念。

同样，不要混淆布尔或运算符||（双竖杠）与按位或运算符|（单竖杠）。

按位取反~（波浪号）看起来与布尔非!有很大不同（感叹号或程序员口中的“棒”），但是你仍然必须确保在什么地方用哪一个。

例如

```
if (a >= 10 && a <= 20) {} // true if a is between 10 and 20
```

- 返回顶部

指针运算符

&（引用）和 \*（间接引用）

指针对于 C 初学者来说是更复杂的对象之一。并且可能写大量的Arduino程序甚至都不会遇到指针。无论如何，巧妙地控制特定的数据结构，使用指针可以简化代码，而且在自己工具箱中拥有熟练控制指针的知识是很方便的。

- 返回顶部

位运算

位与

按位与（&）

按位操作符在变量的位级执行运算。它们帮助解决各种常见的编程问题。以下大部分资料来自一个有关位数学的优秀教程，或许可以在这里找到。[1] (<http://playground.arduino.cc/Code/BitMath>)

描述和语法

以下是所有这些运算符的描述和语法。更详细的资料或许可以在参考指南中找到。

按位与（&）

在C++中按位与运算符是单个与符号，用于其它两个整型表达式之间使用。按位与运算独立地在周围的表达式的每一位上执行操作。根据这一规则：如果两个输入位都是1，结果输出1，否则输出0。表达这一思想的另一个方法是：

```
0 0 1 1  operand1
0 1 0 1  operand2
-----
0 0 0 1  (operand1 & operand2) - returned result
```

在Arduino中，int型是16位的。所以在两个整型表达式之间使用&将会导致16个与运算同时发生。代码片断就像这样：

```
int a = 92;    // in binary: 000000001011100
int b = 101;   // in binary: 000000001100101
int c = a & b; // result:    000000001000100, or 68 in decimal.
```

在a和b的16位的每一位将使用按位与处理。且所有16位结果存入C中，以二进制存入的结果值01000100，即十进制的68。

按位与的其中一个最常用的用途是从一个整型数中选择特定的位，常被称为掩码屏蔽。看如下示例：

- 返回顶部

位或

按位或（|）

在C++中按位或运算符是垂直的条杆符号，|。就像&运算符，|独立地计算它周围的两个整型表达式的每一位。（当然）它所做的是不同的（操作）。两个输入位其中一个或都是1按位或将得到1，否则为0。换句话说：

```

0 0 1 1   operand1
0 1 0 1   operand2
-----
0 1 1 1   (operand1 | operand2) - returned result

```

这是一个使用一小断C++代码描述的按位或（运算）的例子：

```

int a = 92;    // in binary: 000000001011100
int b = 101;   // in binary: 000000001100101
int c = a | b; // result:    000000001111101, or 125 in decimal.

```

按位与和按位或的一个共同的工作是在端口上进行程序员称之为读-改-写的操作。在微控制器中，每个端口是一个8位数字，每一位表示一个引脚的状态。写一个端口可以同时控制所有的引脚。

PORTD是内建的参照数字口0, 1, 2, 3, 4, 5, 6, 7的输出状态的常量。如果一个比特位是1，那么该引脚置高。（引脚总是需要用pinMode()指令设置为输出模式）。所以如果我们写入PORTD = B00110001；我们就会让引脚2, 3和7输出高。一个小小的问题是，我们同时也改变了某些引脚的0, 1状态。这用于Arduino与串口通讯，所以我们可能会干扰串口通讯。

我们的程序规则是：

仅仅获取和清除我们想控制的与相应引脚对应的位（使用按位与）。  
合并要修改的PORTD值与所控制的引脚的新值（使用按位或）。

```

int i;    // counter variable
int j;

void setup() {
  DDRD = DDRD | B11111100; // set direction bits for pins 2 to 7, leave 0 and 1 untouched (xx | 00 == xx)
  // same as pinMode(pin, OUTPUT) for pins 2 to 7
  Serial.begin(9600);
}

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

    PORTD = PORTD & B00000011; // clear out bits 2 - 7, leave pins 0 and 1 untouched (xx & 11 == xx)
    j = (i << 2);              // shift variable up to pins 2 - 7 - to avoid pins 0 and 1
    PORTD = PORTD | j;          // combine the port information with the new information for LED pins
    Serial.println(PORTD, BIN); // debug to show masking
    delay(100);
  }
}

```

- [返回顶部](#)

## 位异或

### 按位异或（^）

在C++中有一个有点不寻常的操作，它被称为按位异或，或者XOR（在英语中，通常读作“eks-or”）。按位异或运算符使用符号^。该运算符与按位或运算符“|”非常相似，唯一的不同是当输入位都为1时它返回0。

```

0 0 1 1   operand1
0 1 0 1   operand2
-----
0 1 1 0   (operand1 ^ operand2) - returned result

```

看待XOR的另一个视角是，当输入不同时结果为1，当输入相同时结果为0。

这里是一个简单的示例代码：

```

int x = 12;    // binary: 1100
int y = 10;    // binary: 1010

```

```
int z = x ^ y; // binary: 0110, or decimal 6
```

“^”运算符常用于翻转整数表达式的某些位（例如从0变为1，或从1变为0）。在一个按位异或操作中，如果相应的掩码位为1，该位将翻转，如果为0，该位不变。以下是一个闪烁引脚5的程序。

```
// Blink_Pin_5
// demo for Exclusive OR
void setup(){
  DDRD = DDRD | B00100000; // set digital pin five as OUTPUT
  Serial.begin(9600);
}

void loop(){
  PORTD = PORTD ^ B00100000; // invert bit 5 (digital pin 5), leave others untouched
  delay(100);
}
```

▪ [返回顶部](#)

位非

按位取反（~）

在C++中按位取反运算符为波浪符“~”。不像“&”和“|”，按位取反运算符应用于其右侧的单个操作数。按位取反操作会翻转其每一位。0变为1，1变为0。例如：

```
0 1 operand1
-----
1 0 ~ operand1
```

```
int a = 103; // binary: 0000000001100111
int b = ~a; // binary: 1111111110011000 = -104
```

看到此操作的结果为一个负数：-104，你可能会感到惊讶，这是因为一个整型变量的最高位是所谓的符号位。如果最高位为1，该整数被解释为负数。这里正数和负数的编码被称为二进制补码。欲了解更多信息，请参阅百科条目：补码。

顺便说一句，值得注意的是，对于任何整数x，~x 与 -x-1 相等。

有时候，符号位在有符号整数表达式中能引起一些不期的意外。

▪ [返回顶部](#)

左移、右移

左移运算（<<），右移运算（>>）

描述

From The Bitmath Tutorial in The Playground

在C++中有两个移位运算符：左移运算符<<和右移运算符>>。这些运算符将使左边操作数的每一位左移或右移其右边指定的位数。

语法

```
variable << number_of_bits
variable >> number_of_bits

参数<br>
*variable - (byte, int, long) number_of_bits integer <= 32 <br>
```

```
示例: <br>
<pre style="color:green">
int a = 5;           // binary: 000000000000101
int b = a << 3;      // binary: 000000000101000, or 40 in decimal
int c = b >> 3;      // binary: 000000000000101, or back to 5 like we started with
```

当把x左移y位（x << y），x中最左边的y位将会丢失。

```
int a = 5;           // binary: 000000000000101
int b = a << 14;     // binary: 0100000000000000 - 101中的第一个1被丢弃
```

如果您确信没有值被移出，理解左移位运算符一个简单的办法是，把它的左操作数乘2将提高其幂值。例如，要生成2的乘方，可以使用以下表达式：

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

当把x右移y位，x的最高位为1，该行为依赖于x的确切的数据类型。如果x的类型是int，最高位为符号位，决定x是不是负数，正如我们在上面已经讨论过的。在这种情况下，符号位会复制到较低的位：

```
int x = -16;         // binary: 1111111111110000
int y = x >> 3;      // binary: 111111111111110
```

该行为，被称为符号扩展，常常不是你所期待的。反而，你可能希望移入左边的是0。事实上右移规则对于无符合整型表达式是不同的。所以你可以使用强制类型转换来避免左边移入1。

```
int x = -16;         // binary: 1111111111110000
int y = (unsigned int)x >> 3; // binary: 000111111111110
```

如果你可以很小心地避免符号扩展，你可以使用右移位运算符>>，作为除以2的幂的一种方法。例如

```
int x = 1000;
int y = x >> 3;     // 1000除以8，得y = 125.
```

- [返回顶部](#)

复合运算符

```
自加++
i++; //相当于 i = i + 1;
自减--
i--; //相当于 i = i - 1;
复合加+=
i+=5; //相当于 i = i + 5;
复合减-=
i-=5; //相当于 i = i - 5;
复合乘*=
i*=5; //相当于 i = i * 5;
复合除/=
i/=5; //相当于 i = i / 5;
复合与&=
i&=5; //相当于 i = i & 5;
复合或|=
i|=5; //相当于 i = i | 5;
```

- [返回顶部](#)

变量

(本节转自极客工坊)

常量

constants是在Arduino语言里预定义的变量。它们被用来使程序更易阅读。我们按组将常量分类。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。

——这里引申一下题外话arduino是大小写敏感语言（case sensitive）。  
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。  
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。

■ 返回顶部

宏定义

```
#define HIGH 0x1 高电平
#define LOW 0x0 低电平
#define INPUT 0x0 输入
#define OUTPUT 0x1 输出
#define true 0x1 真
#define false 0x0 假
#define PI 3.14159265
#define HALF_PI 1.57079 二分之一PI
#define TWO_PI 6.283185 二倍PI
#define DEG_TO_RAD 0.01745329 弧度转角度
#define RAD_TO_DEG 57.2957786 角度转弧度
```

■ 返回顶部



整型常量

整数常量

整数常量是直接程序中使用的数字，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

- [返回顶部](#)

浮点常量

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

- [返回顶部](#)

数据类型

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()
{
  // ...
}
```

- 返回顶部

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
  }
}
```

- 返回顶部

char

char

描述

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
```

- 返回顶部

unsigned char

无符号字符型  
描述

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;
```

- [返回顶部](#)

byte

字节型  
描述

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）
```

- [返回顶部](#)

int

整型  
简介

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（ $-2^{15} \sim (2^{15}-1)$ ）。

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。
```

- [返回顶部](#)

unsigned int

无符号整型  
描述

unsigned int（无符号整型）与整型数据同样大小，占据2字节。它只能用于存储正数而不能存储负数，范围 $0 \sim 65,535$  ( $2^{16} - 1$ )。

无符号整型和整型最重要的区别是它们的最高位不同，既符号位。在Arduino整型类型中，如果最高位是1，则此数被认为是负数，剩下的15位为按2的补码计算所得值。

例子

```
unsigned int ledPin = 13;
```

语法

```
unsigned int var = val;
```

- var - 无符号变量名称
- val - 给变量所赋予的值

编程提示

当变量的值超过它能表示的最大值时它会“滚回”最小值，反向也会出现这种现象。

```
unsigned int x
  x = 0;
x = x - 1;      //x现在等于65535—向负数方向滚回
x = x + 1;      //x现在等于0—滚回
```

- [返回顶部](#)

word

字  
描述

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

例子

```
word w = 10000;
```

- [返回顶部](#)

long

长整型  
描述

长整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

- 返回顶部

unsigned long

无符号长整型  
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

- 返回顶部

float

单精度浮点型  
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 $3.4028235 \times 10^{+38}$  ~  $-3.4028235 \times 10^{+38}$ 。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)
```

- 返回顶部

double

双精度浮点型  
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。

- 返回顶部

string

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。

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，它实际上是一个二维数组的典型。

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);
  }
}
```

- [返回顶部](#)

## String(c++)

### 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）

- [返回顶部](#)

## array

### Arrays（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，也就是说，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]);
}
```

- [返回顶部](#)

## 数据类型转换

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



- [返回顶部](#)

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

- [返回顶部](#)

int()

简介

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

- [返回顶部](#)

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

- 字符

- [返回顶部](#)

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

- 长整型数

- [返回顶部](#)

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

- float型数

- [返回顶部](#)

变量作用域&修饰符

变量作用域

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 gPWVal; // 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
  }
}
```

■ 返回顶部

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;
}
```

■ 返回顶部

volatile（易变变量）

volatile关键字

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;
}
```

▪ 返回顶部

const（不可改变变量）

const关键字

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。

▪ 返回顶部

辅助工具

sizeof()（sizeof运算符）

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
}
```

- [返回顶部](#)

ASCII码表

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

- [返回顶部](#)

基本函数

（本节由柴树杉[翻译整理]（chaishushan@gmail.com））

数字I/O

pinMode()

```
void pinMode (uint8_t pin, uint8_t mode)
```

## 设置引脚模式

配置引脚为输出或输出模式.

参数:

- pin 引脚编号
- mode: INPUT, OUTPUT, 或 INPUT\_PULLUP.

例子:

```
int ledPin = 13;           // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH); // sets the LED on
  delay(1000);                // waits for a second
  digitalWrite(ledPin, LOW);  // sets the LED off
  delay(1000);                // waits for a second
}
```

注解:

- 模拟引脚也可以当作数字引脚使用, 编号为14(对应模拟引脚0)到19(对应模拟引脚5).
- [返回顶部](#)

## digitalWrite()

```
void digitalWrite (uint8_t pin, uint8_t value)
```

### 写数字引脚

写数字引脚, 对应引脚的高低电平. 在写引脚之前, 需要将引脚设置为OUTPUT模式.

参数:

- pin 引脚编号
- value HIGH 或 LOW

用法:

```
int ledPin = 13;           // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH); // 点亮LED
  delay(1000);                // 等待1秒
  digitalWrite(ledPin, LOW);  // 关
  delay(1000);                // waits for a second
}
```

注解:

- 模拟引脚也可以当作数字引脚使用, 编号为14(对应模拟引脚0)到19(对应模拟引脚5).
- [返回顶部](#)

## digitalRead()

```
int digitalRead (uint8_t pin)
```

### 读数字引脚

读数字引脚, 返回引脚的高低电平. 在读引脚之前, 需要将引脚设置为INPUT模式.

参数:

- pin 引脚编号

返回:

HIGH或LOW

```
int ledPin = 13; // LED connected to digital pin 13
int inPin = 7;   // pushbutton connected to digital pin 7
int val = 0;     // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the digital pin 13 as output
  pinMode(inPin, INPUT);   // sets the digital pin 7 as input
}

void loop()
{
  val = digitalRead(inPin); // read the input pin
  digitalWrite(ledPin, val); // sets the LED to the button's value
}
```

注解:

如果引脚没有链接到任何地方，那么将随机返回 HIGH 或 LOW.

- 返回顶部

## 模拟I/O

analogReference()

```
void analogReference (uint8_t type)
```

配置参考电压

配置模式引脚的参考电压。函数 analogRead 在读取模拟值之后，将根据参考电压将 模拟值转换到[0, 1023]区间。有以下类型：

DEFAULT : 默认5V。 INTERNAL：低功耗模式。 ATmega168和ATmega8对应1.1V到2.56V。 EXTERNAL：扩展模式。通过AREF引脚获取参考电压。

参数:

- type 参考类型 (DEFAULT/INTERNAL/EXTERNAL)

- 返回顶部

analogRead()

```
int analogRead (uint8_t pin)
```

读模拟引脚

读模拟引脚，返回[0-1023]之间的值。每读一次需要花1微妙的时间。

参数:

- pin 引脚编号

返回:

- 0到1023之间的值

例子:

```
int analogPin = 3; // potentiometer wiper (middle terminal) connected to analog pin 3
                  // outside leads to ground and +5V
int val = 0;       // variable to store the value read

void setup()
```

```
{
  Serial.begin(9600);          //  setup serial
}

void loop()
{
  val = analogRead(analogPin); // read the input pin
  Serial.println(val);         // debug value
}
```

- [返回顶部](#)

## analogWrite()

```
void analogWrite (uint8_t pin, int value)
```

### 写模拟引脚

#### 参数:

- pin 引脚编号
- value 0到255之间的值, 0对应off, 255对应on

写一个模拟值(PWM)到引脚. 可以用来控制LED的亮度, 或者控制电机的转速. 在执行该操作后, 应该等待一定时间后才能对该引脚进行下一次的读或写操作. PWM的频率大约为490Hz.

在一些基于ATmega168的新的Arduino控制板(如Mini 和BT)中, 该函数支持以下引脚: 3, 5, 6, 9, 10, 11. 在基于ATmega8的型号中支持9, 10, 11引脚.

#### 例子:

```
int ledPin = 9;          // LED connected to digital pin 9
int analogPin = 3;       // potentiometer connected to analog pin 3
int val = 0;             // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin); // read the input pin
  analogWrite(ledPin, val / 4); // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
}
```

- [返回顶部](#)

## 高级I/O

### shiftOut()

```
void shiftOut (uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, byte val)
```

#### 位移输出函数

输入value数据后Arduino会自动把数据移动分配到8个并行输出端. 其中dataPin为连接DS的引脚号, clockPin为连接SH\_CP的引脚号, bitOrder为设置数据位移顺序, 分别为高位先入MSBFIRST或者低位先入LSBFIRST.

#### 参数:

- dataPin 数据引脚
- clockPin 时钟引脚
- bitOrder 移位顺序 ( MSBFIRST 或 LSBFIRST)
- val 数据

```
// Do this for MSBFIRST serial
int data = 500;
// shift out highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// shift out lowbyte
shiftOut(dataPin, clock, MSBFIRST, data);

// Or do this for LSBFIRST serial
data = 500;
```



```
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// shift out highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

- [返回顶部](#)

## pulseIn()

```
unsigned long pulseIn (uint8_t pin, uint8_t state, unsigned long timeout)
```

### 读脉冲

读引脚的脉冲，脉冲可以是 HIGH 或 LOW. 如果是 HIGH, 函数将先等引脚变为高电平，然后 开始计时，一直到变为低电平为止。返回脉冲持续的时间长短，单位为微秒。如果超时还没有 读到的话，将返回0。

参数：

1. pin 引脚编号
2. state 脉冲状态
3. timeout 超时时间(us)

下面的例子演示了统计高电平的继续时间：

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

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

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

- [返回顶部](#)

## 时间

### millis()

```
unsigned long millis (void)
```

### 毫秒时间

获取机器运行的时间长度，单位毫秒。系统最长的记录时间接近50天，如果超出时间将从0开始。

警告：

时间为 unsigned long类型，如果用 int 保存时间将得到错误结果：

### delay(ms)

```
void delay (unsigned long ms)
```

### 延时(毫秒)

延时，单位毫秒(1秒有1000毫秒)。

警告：

参数为unsigned long，因此在延时参数超过32767(int型最大值)时，需要用“UL”后缀表示为无符号 长整型，例如：delay(60000UL)；。同样在参数表达式，切表达式中有int类型时，需要强制转换为 unsigned long类型，例如：delay((unsigned long)tdelay \* 100UL)；。

一下例子设置13引脚对应的LED等以1秒频率闪烁：

```
int ledPin = 13;           // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                  // wait for 1 second
  digitalWrite(ledPin, LOW);    // turn the LED off (LOW is the voltage level)
  delay(1000);                  // wait for 1 second
}
```

```
digitalWrite(ledPin, HIGH); // sets the LED on
delay(1000);                // waits for a second
digitalWrite(ledPin, LOW);  // sets the LED off
delay(1000);                // waits for a second
}
```

▪ [返回顶部](#)

delayMicroseconds(us)

```
void delayMicroseconds (unsigned int us)
```

延时(微秒)

延时，单位为微秒(1毫秒有1000微妙)。如果延时的时间有几千微妙，那么建议使用 delay 函数。目前参数最大支持16383微妙(不过以后的版本中可能会变化)。

以下代码向第8号引脚发送脉冲，每次脉冲持续50微妙的时间。

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

void setup()
{
  pinMode(outPin, OUTPUT); // sets the digital pin as output
}

void loop()
{
  digitalWrite(outPin, HIGH); // sets the pin on
  delayMicroseconds(50);      // pauses for 50 microseconds
  digitalWrite(outPin, LOW);  // sets the pin off
  delayMicroseconds(50);      // pauses for 50 microseconds
}
```

▪ [返回顶部](#)

数学库

min()

```
#define min(a, b) ((a)<(b)?(a):(b))
```

最小值

取两者之间最小值。例如：

```
sensVal = min(sensVal, 100); // assigns sensVal to the smaller of sensVal or 100
                             // ensuring that it never gets above 100.
```

▪ [返回顶部](#)

max()

```
#define max(a, b) ((a)>(b)?(a):(b))
```

最大值

取两者之间最大值。例如：

```
sensVal = max(sensVal, 20); // assigns sensVal to the larger of sensVal or 20
                             // (effectively ensuring that it is at least 20)
```

▪ [返回顶部](#)

abs()

```
abs(x)  ((x)>0?(x):-x)
```

求绝对值

- [返回顶部](#)

constrain()

```
#define constrain(amt, low, high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
```

调整到区间

如果值 amt 小于 low, 则返回 low; 如果 amt 大于 high, 则返回 high; 否则, 返回 amt . 一般可以用于将值归一化到某个区间.

例如:

```
sensVal = constrain(sensVal, 10, 150);
// limits range of sensor values to between 10 and 150
```

- [返回顶部](#)

map()

```
long map ( long x,
long in_min,
long in_max,
long out_min,
long out_max
)
```

等比映射

将位于[in\_min, in\_max]之间的x映射到[out\_min, out\_max].

参数:

- x 要映射的值
- in\_min 映射前区间
- in\_max 映射前区间
- out\_min 映射后区间
- out\_max 映射后区间

例如下面的代码中用 map 将模拟量从[0, 1023]映射到[0, 255]区间:

```
// Map an analog value to 8 bits (0 to 255)
void setup() {}

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

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;
}
```

- [返回顶部](#)

pow()

```
double pow (float base, float exponent)
指数函数 <br>
```

- [返回顶部](#)

sqrt()

```
double sqrt (double x)
//开平方
```

- [返回顶部](#)

三角函数

sin(),cos(),tan()

```
float sin (float rad)
//正弦函数
float cos (float rad)
//余弦函数
float tan (float rad)
//正切函数
```

- [返回顶部](#)

随机数

randomSeed()

```
void randomSeed ( unsigned int seed )
```

设置随机种子

可以用当前时间作为随机种子。随机种子的设置对产生的随机序列有影响。

参数:

- seed 随机种子

- [返回顶部](#)

random()

```
long random (long howbig)
```

生成随机数

生成[0, howbig-1]范围的随机数。

参数:

- howbig 最大值

```
long random (long howsmall, long howbig)
```

生成随机数

生成[howsmall, howbig-1]范围的随机数.

参数:

- howsmall 最小值
- howbig 最大值

- [返回顶部](#)

## 位操作

位操作

```
#define lowByte(w)    ((w) & 0xff)
//取低字节

#define highByte(w)   ((w) >> 8)
//取高字节

#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
//读一个bit

#define bitWrite(value, bit, bitvalue) (bitvalue ? bitSet(value, bit) : bitClear(value, bit))
//写一个bit

#define bitSet(value, bit) ((value) |= (1UL << (bit)))
//设置一个bit

#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
//清空一个bit

#define bit(b)        (1 << (b))
//生成相应bit
```

- [返回顶部](#)

## 设置中断函数

attachInterrupt()

```
void attachInterrupt (uint8_t interruptNum, void(*)(void)userFunc, int mode)
```

设置中断

指定中断函数. 外部中断有0和1两种, 一般对应2号和3号数字引脚.

参数:

- interrupt 中断类型, 0或1
- fun 对应函数
- mode 触发方式. 有以下几种:
  - LOW 低电平触发中断
  - CHANGE 变化时触发中断
  - RISING 低电平变为高电平触发中断
  - FALLING 高电平变为低电平触发中断

注解:

在中断函数中 delay 函数不能使用, millis 始终返回进入中断前的值. 读串口数据的话, 可能会丢失. 中断函数中使用的变量需要定义为 volatile 类型.

下面的例子如果通过外部引脚触发中断函数, 然后控制LED的闪烁.

```
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;
}
```

- [返回顶部](#)

detachInterrupt()

```
void detachInterrupt (uint8_t interruptNum)
```

取消中断

取消指定类型的中断.

参数:

- interrupt 中断的类型.

- [返回顶部](#)

interrupts()

```
#define interrupts() sei()
```

开中断

例子:

```
void setup() {}

void loop()
{
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

noInterrupts()

```
#define noInterrupts() cli()
```

关中断

例子:

```
void setup() {}

void loop()
{
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

- [返回顶部](#)

## 串口通讯

（该小节为最新翻译）

void begin (long) 打开串口

uint8\_t available (void) 有串口数据返回真

int read (void) //读串口

void flush (void) //刷新串口数据

virtual void write (uint8\_t) //写串口 </pre>

- 返回顶部

begin()

```
void HardwareSerial::begin (long speed)
```

打开串口

参数:

- speed 波特率

- 返回顶部

available()

获取串口上可读取的数据的字节数。该数据是指已经到达并存储在接收缓存（共有64字节）中。available()继承自Stream实用类。

语法:

```
Serial.available()
```

Arduino Mega only: Serial1.available() Serial2.available() Serial3.available()

参数:

无

返回值:

- 返回可读取的字节数

示例:

```
int incomingByte = 0;  // for incoming serial data

void setup() {
    Serial.begin(9600);  // opens serial port, sets data rate to 9600 bps
}

void loop() {
    // send data only when you receive data:
    if (Serial.available() > 0) {
        // read the incoming byte:
        incomingByte = Serial.read();

        // say what you got:
        Serial.print("I received: ");
        Serial.println(incomingByte, DEC);
    }
}
```

Arduino Mega example:

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

```

}

void loop() {
  // read from port 0, send to port 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, BYTE);
  }
  // read from port 1, send to port 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}

```

■ [返回顶部](#)

## read()

读串口数据，read()继承自Stream实用类。 语法：

```
Serial.read()
```

Arduino Mega only: Serial1.read() Serial2.read() Serial3.read()

参数：

无

返回值：

串口上第一个可读取的字节（如果没有可读取的数据则返回-1）- int型。

示例：

```

int incomingByte = 0;  // 用于存储从串口读到的数据

void setup() {
  Serial.begin(9600);    // 打开串口，设置速率为9600 bps
}

void loop() {
  // 只在收到数据时发送数据
  if (Serial.available() > 0) {
    // 读取传入的字节
    incomingByte = Serial.read();

    // 指示你收到的数据
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}

```

■ [返回顶部](#)

## flush()

刷新串口数据

## print()

往串口发数据，无换行 描述

以人类可读的ASCII码形式向串口发送数据，该函数有多种格式。整数的每一位将以ASCII码形式发送。浮点数同样以ASCII码形式发送，默认保留小数点后两位。字节型数据将以单个字符形式发送。字符和字符串会以其相应的形式发送。例如：

```

Serial.print(78) 发送 "78"
Serial.print(1.23456) 发送 "1.23"
Serial.print('N') 发送 "N"
Serial.print("Hello world.") 发送 "Hello world."

```

可选的第二个参数用于指定数据的格式。允许的值为：BIN（binary二进制），OCT（octal八进制），DEC（decimal十进制），HEX（hexadecimal十六进制）。对于浮点数，该参数指定小数点的位数。例如：



```
Serial.print(78, BIN) gives "1001110"
Serial.print(78, OCT) gives "116"
Serial.print(78, DEC) gives "78"
Serial.print(78, HEX) gives "4E"
Serial.println(1.23456, 0) gives "1"
Serial.println(1.23456, 2) gives "1.23"
Serial.println(1.23456, 4) gives "1.2346"
```

你可以用F()把待发送的字符串包装到flash存储器。例如：

```
Serial.print(F("Hello World"))
```

要发送单个字节数据，请使用Serial.write()。

语法：

```
Serial.print(val)
Serial.print(val, format)
```

参数：

- val: 要发送的数据（任何数据类型）
- format: 指定数字的基数（用于整型数）或者小数的位数（用于浮点数）。

返回值：<>

- size\_t (long): print() 返回发送的字节数（可丢弃该返回值）。

示例：

```
/*
 * Uses a FOR loop for data and prints a number in various formats.
 */
int x = 0;    // variable

void setup() {
  Serial.begin(9600);    // open the serial port at 9600 bps:
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");    // prints a label
  Serial.print("\t");    // prints a tab

  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++){    // only part of the ASCII chart, change to suit

    // print it out in many formats:
    Serial.print(x);    // print as an ASCII-encoded decimal - same as "DEC"
    Serial.print("\t");    // prints a tab

    Serial.print(x, DEC);    // print as an ASCII-encoded decimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, HEX);    // print as an ASCII-encoded hexadecimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, OCT);    // print as an ASCII-encoded octal
    Serial.print("\t");    // prints a tab

    Serial.println(x, BIN);    // print as an ASCII-encoded binary
    // then adds the carriage return with "println"
    delay(200);    // delay 200 milliseconds
  }
  Serial.println("");    // prints another carriage return
```

编程技巧：

在版本1.0时，串口传输是异步的，Serial.print()会在数据发送完成前返回。

- [返回顶部](#)

println()

往串口发数据，类似Serial.print()，但有换行

write()

写二进制数据到串口，数据是一个字节一个字节地发送的，若以字符形式发送数字请使用print()代替。  
语法：

```
Serial.write(val)
Serial.write(str)
Serial.write(buf, len)
```

Arduino Mega也支持：Serial1, Serial2, Serial3(在Serial的位置)

参数：

- val: 作为单个字节发送的数据
- str: 由一系列字节组成的字符串
- buf: 同一系列字节组成的数组
- len: 要发送的数组的长度

返回：

- byte

write() 会返回发送的字节数，所以读取该返回值是可选的。

示例：

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

void loop() {
  Serial.write(45); //以二进制形式发送数字45

  int bytesSent = Serial.write("hello"); //发送字符串“hello” 并返回该字符串的长度。
}
```

- [返回顶部](#)

peak()

描述：

返回收到的串口数据的下一个字节（字符），但是并不把该数据从串口数据缓存中清除。就是说，每次成功调用peak()将返回相同的字符。与read()一样，peak()继承自Stream实用类。 语法：  
可参照Serial.read()

- [返回顶部](#)

serialEvent()

描述：

当串口有数据到达时调用该函数（然后使用Serial.read()捕获该数据）。

注意：目前serialEvent()并不兼容于Esplora, Leonardo, 或 Micro。

语法：

```
void serialEvent() {
  //statements

  Arduino Mega only:
  void serialEvent1() {
    //statements
  }

  void serialEvent2() {
    //statements
  }

  void serialEvent3() {
    //statements
  }
}
```

statements可以是任何有效的语句。

- [返回顶部](#)

来自“<http://wiki.dfrobot.com.cn/index.php?title=Arduino编程参考手册&oldid=88237>”

- 
- 本页面最后修改于2015年8月26日（星期三）11:59。
  - 此页面已被浏览过78,143次。