Arduino 编程参考手册

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)

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

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 板起动时 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);
}
```

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。

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。

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);
    }
}
```

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

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++;
 }
```

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);
```

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);
}
```

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);
}
```

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
}
```

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:
```

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。

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)
}
```

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。

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

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};

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
senVal = 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。

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。

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。

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。

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
```

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。

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]
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: 0000000001011100

int b = 101;    // in binary: 0000000001100101
int c = a & b;// result:      0000000001000100, 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：

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: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a | b;// result:      0000000001111101, 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 == x
// 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);
    }
}
```

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);
}

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，请参阅维基百科条目：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。

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

*variable - (byte, int, long) number_of_bits integer <= 32

示例：

<pre style="color:green">
int a = 5; // binary: 0000000000000101

int b = a << 3; // binary: 0000000000101000, or 40 in decimal
int c = b >> 3; // binary: 0000000000000101, 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: 0000000000000101
int b = a << 14; // binary: 0100000000000000 - 10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: 1111111111111110

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: 0001111111111110

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.

复合运算符
自加++
i++; /相当于 i = i + 1;
自减--
i--; /相当于 i = i - 1;
</pre>

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;

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。

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
                PI.
#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
```

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

浮点数常量

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

2.34E5 2.34 * 10^5 234000

67E-12 67.0 * 10^-120.000000000067

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

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
```



```
    }
  }
```

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
```

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;
```

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）
```

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 ~（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。
```

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 0~65,535 (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;
```