C# 常量 C# 判断

C# 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C# 有丰富的内置运算符，分类如下：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

本教程将逐一讲解算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

算术运算符

下表显示了 C# 支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C# 中所有可用的算术运算符：

实例

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int a = 21;
            int b = 10;
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - c 的值是 {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - c 的值是 {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - c 的值是 {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - c 的值是 {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - c 的值是 {0}", c);

            // ++a 先进行自增运算再赋值
            c = ++a;
            Console.WriteLine("Line 6 - c 的值是 {0}", c);

            // 此时 a 的值为 22
            // --a 先进行自减运算再赋值
            c = --a;
            Console.WriteLine("Line 7 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 22
Line 7 - c 的值是 21
```

- **c = a++**: 先将 a 赋值给 c，再对 a 进行自增运算。
- **c = ++a**: 先将 a 进行自增运算，再将 a 赋值给 c 。
- **c = a--**: 先将 a 赋值给 c，再对 a 进行自减运算。
- **c = --a**: 先将 a 进行自减运算，再将 a 赋值给 c 。

实例

```
using System;

namespace OperatorsAppl
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int a = 1;
            int b;

            // a++ 先赋值再进行自增运算
            b = a++;
            Console.WriteLine("a = {0}", a);
            Console.WriteLine("b = {0}", b);
            Console.ReadLine();

            // ++a 先进行自增运算再赋值
            a = 1; // 重新初始化 a
            b = ++a;
            Console.WriteLine("a = {0}", a);
            Console.WriteLine("b = {0}", b);
            Console.ReadLine();

            // a-- 先赋值再进行自减运算
            a = 1;  // 重新初始化 a
            b= a--;
            Console.WriteLine("a = {0}", a);
            Console.WriteLine("b = {0}", b);
            Console.ReadLine();

            // --a 先进行自减运算再赋值
            a = 1;  // 重新初始化 a
            b= --a;
            Console.WriteLine("a = {0}", a);
            Console.WriteLine("b = {0}", b);
            Console.ReadLine();
        }
    }
}
```

运行实例 »

执行以上程序，输出结果为：

```
a = 2
b = 1
a = 2
b = 2
a = 0
b = 1
a = 0
b = 0
```

关系运算符

下表显示了 C# 支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C# 中所有可用的关系运算符：

实例

```
using System;

class Program
{
  static void Main(string[] args)
  {
      int a = 21;
      int b = 10;

      if (a == b)
      {
          Console.WriteLine("Line 1 - a 等于 b");
      }
      else
      {
          Console.WriteLine("Line 1 - a 不等于 b");
      }
      if (a < b)
      {
          Console.WriteLine("Line 2 - a 小于 b");
      }
      else
      {
          Console.WriteLine("Line 2 - a 不小于 b");
      }
      if (a > b)
```

```
        {
            Console.WriteLine("Line 3 - a 大于 b");
        }
        else
        {
            Console.WriteLine("Line 3 - a 不大于 b");
        }
        /* 改变 a 和 b 的值 */
        a = 5;
        b = 20;
        if (a <= b)
        {
           Console.WriteLine("Line 4 - a 小于或等于 b");
        }
        if (b >= a)
        {
           Console.WriteLine("Line 5 - b 大于或等于 a");
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 a
```

逻辑运算符

下表显示了 C# 支持的所有逻辑运算符。假设变量 **A** 为布尔值 true，变量 **B** 为布尔值 false，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C# 中所有可用的逻辑运算符：

实例

```
using System;
```

```
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - 条件为真");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - 条件为真");
            }
            /* 改变 a 和 b 的值 */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - 条件为真");
            }
            else
            {
                Console.WriteLine("Line 3 - 条件不为真");
            }
            if (!(a && b))
            {
                Console.WriteLine("Line 4 - 条件为真");
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p | q	p ^ q
0	0	0	0	0

0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

下表列出了 C# 支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	按位取反运算符是一元运算符，具有"翻转"位效果，即0变成1，1变成0，包括符号位。	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C# 中所有可用的位运算符：

实例

```
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;            /* 60 = 0011 1100 */
            int b = 13;            /* 13 = 0000 1101 */
```

```
            int c = 0;

             c = a & b;           /* 12 = 0000 1100 */
             Console.WriteLine("Line 1 - c 的值是 {0}", c );

             c = a | b;           /* 61 = 0011 1101 */
             Console.WriteLine("Line 2 - c 的值是 {0}", c);

             c = a ^ b;           /* 49 = 0011 0001 */
             Console.WriteLine("Line 3 - c 的值是 {0}", c);

             c = ~a;               /*-61 = 1100 0011 */
             Console.WriteLine("Line 4 - c 的值是 {0}", c);

             c = a << 2;     /* 240 = 1111 0000 */
             Console.WriteLine("Line 5 - c 的值是 {0}", c);

             c = a >> 2;     /* 15 = 0000 1111 */
             Console.WriteLine("Line 6 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

赋值运算符

下表列出了 C# 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A

/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2

实例

请看下面的实例，了解 C# 中所有可用的赋值运算符：

实例

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  c 的值 = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += c 的值 = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  c 的值 = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  c 的值 = {0}", c);

            c /= a;
            Console.WriteLine("Line 5 - /=  c 的值 = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  c 的值 = {0}", c);

            c <<= 2;
```

```
            Console.WriteLine("Line 7 - <<=  c 的值 = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  c 的值 = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  c 的值 = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  c 的值 = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =     c 的值 = 21
Line 2 - +=    c 的值 = 42
Line 3 - -=    c 的值 = 21
Line 4 - *=    c 的值 = 441
Line 5 - /=    c 的值 = 21
Line 6 - %=    c 的值 = 11
Line 7 - <<=    c 的值 = 44
Line 8 - >>=    c 的值 = 11
Line 9 - &=    c 的值 = 2
Line 10 - ^=    c 的值 = 0
Line 11 - |=    c 的值 = 2
```

其他运算符

下表列出了 C# 支持的其他一些重要的运算符，包括 **sizeof**、**typeof** 和 **? :**。

运算符	描述	实例
sizeof()	返回数据类型的大小。	sizeof(int)，将返回 4.
typeof()	返回 class 的类型。	typeof(StreamReader);
&	返回变量的地址。	&a; 将得到变量的实际地址。
*	变量的指针。	*a; 将指向一个变量。
? :	条件表达式	如果条件为真 ? 则为 X : 否则为 Y
is	判断对象是否为某一类型。	If(Ford is Car) // 检查 Ford 是否是 Car 类

		的一个对象。
as	强制转换，即使转换失败也不会抛出异常。	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

实例

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {

         /* sizeof 运算符的实例 */
         Console.WriteLine("int 的大小是 {0}", sizeof(int));
         Console.WriteLine("short 的大小是 {0}", sizeof(short));
         Console.WriteLine("double 的大小是 {0}", sizeof(double));

         /* 三元运算符的实例 */
         int a, b;
         a = 10;
         b = (a == 1) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);

         b = (a == 10) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
int 的大小是 4
short 的大小是 2
double 的大小是 8
b 的值是 30
b 的值是 20
```

C# 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 * 2，在这里，x 被赋值为 13，而不是 20，因为运算符 * 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
```

```
        {
            int a = 20;
            int b = 10;
            int c = 15;
            int d = 5;
            int e;
            e = (a + b) * c / d;      // ( 30 * 15 ) / 5
            Console.WriteLine("(a + b) * c / d 的值是 {0}", e);

            e = ((a + b) * c) / d;   // (30 * 15 ) / 5
            Console.WriteLine("((a + b) * c) / d 的值是 {0}", e);

            e = (a + b) * (c / d);   // (30) * (15/5)
            Console.WriteLine("(a + b) * (c / d) 的值是 {0}", e);

            e = a + (b * c) / d;    //  20 + (150/5)
            Console.WriteLine("a + (b * c) / d 的值是 {0}", e);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

← C# 常量 C# 判断 →

1 篇笔记

写笔记

`&，|，^` 除了用于位运算，还可以用于逻辑运算，分别对应与，或，异或。

^ 运算符针对整型类型和 bool 预定义了二元 ^ 运算符。 对于整型类型，^ 会计算其操作数的按位异或。 对于 bool 操作数，^ 计算其操作数的逻辑异或；即，当且仅当其一个操作数为 true 时，结果才为 true。

```
Console.WriteLine(true ^ false);  // 返回 true
Console.WriteLine(false ^ false);  // 返回 false
Console.WriteLine(true ^ true);  // 返回 false
```

| 运算符针对整型类型和 bool 预定义了二元 | 运算符。 对于整型类型，| 会计算其操作数的按位 OR。 对于 bool 操作数，| 会计算其操作数的逻辑 OR；即，当且仅当其两个操作数皆为 false 时，结果才为 false。

```
Console.WriteLine(true | false);  // 返回 true
Console.WriteLine(false | false); // 返回 false
```

& 运算符为整型类型和 bool 预定义了二元 & 运算符。 对于整型类型，& 计算其操作数的逻辑按位 AND。 对于 bool 操作数，& 计算其操作数的逻辑 AND；即，当且仅当其两个操作数皆为 true 时，结果才为 true。

```
Console.WriteLine(true & false);  // 返回 false
Console.WriteLine(true & true);   // 返回 true
```

其中&，|的运算结果与&&，||完全相同，但&&和||的性能更好。因为&&和||都是检查第一个操作数的值，如果已经能判断结果，就根本不处理第二个操作数。

比如：

```
bool a = true;
bool b = false;
bool c = a || b;
```

检查第一个操作数a时已经得出c为true，就不用再处理第二个操作数b了。

沐水杉 1年前 (2018-01-23)