高级操作员

除了在基本操作符中描述的操作符之外，Swift还提供了几个执行更复杂值操作的高级操作符。这些包括您将从C和Objective-C熟悉的所有按位和位移操作符。

与C中的算术运算符不同，Swift中的算术运算符默认不会溢出。溢出行为被捕获并报告为错误。要选择溢出行为，请使用Swift默认情况下溢出的第二组算术运算符，如溢出添加运算符（&+）。所有这些溢出操作符都以&符号（&）开头。

当您定义自己的结构，类和枚举时，为这些自定义类型提供标准Swift运算符的自己实现会很有用。Swift可以轻松地为这些运算符提供量身定制的实现，并准确确定它们的行为应该针对您创建的每种类型。

您不仅限于预定义的操作员。Swift使您可以自由定义自定义中缀，前缀，后缀和赋值运算符，并具有自定义优先级和关联性值。这些运算符可以像任何预定义的运算符一样在代码中使用和采用，甚至可以扩展现有类型以支持您定义的自定义运算符。

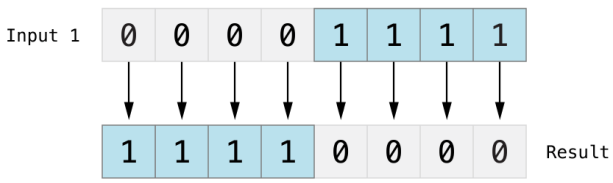
按位运算符

*按位运算符*使您可以操作数据结构中的各个原始数据位。它们通常用于低级编程，如图形编程和设备驱动程序创建。按位运算符在处理来自外部源的原始数据时也很有用，例如编码和解码用于通过自定义协议进行通信的数据。

Swift支持在C中找到的所有按位运算符，如下所述。

按位NOT运算符

该*位NOT运算符*（~）反转数所有位：



按位NOT运算符是一个前缀运算符，它在其运行的值之前立即出现，没有任何空格：

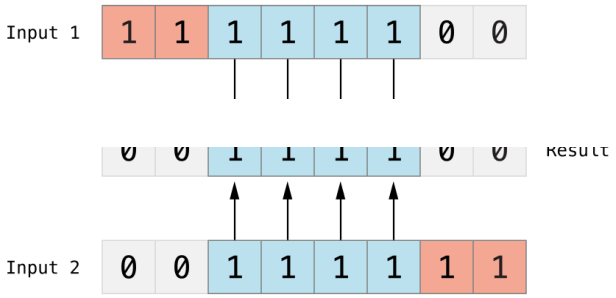
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

`UInt8`整数有八位，可以存储`0`和之间的任何值`255`。本示例`UInt8`使用二进制值初始化一个整数，`00001111`其前四位设置为`0`，其后四位设置为`1`。这相当于一个十进制值`15`。

然后使用按位NOT运算符创建一个新的常数`invertedBits`，该常数等于`initialBits`，但所有位反转。零点成为零点，零点变为零点。`invertedBitsis` 的值`11110000`，它等于一个无符号十进制值`240`。

按位与运算符

该*位AND运算符*（&）结合了两个数字的位数。它返回一个新的数字，其位数`1`只有`1`在*两个*输入数字中的位数都相等时才被设置为：

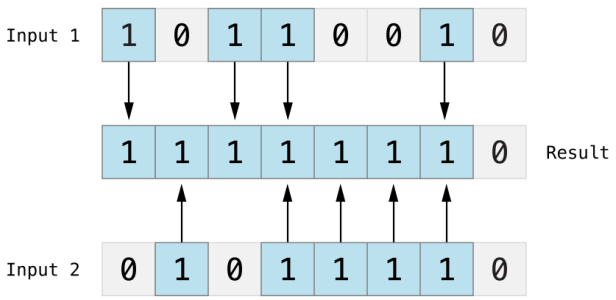


在下面的例子中，其值`firstSixBits`和`lastSixBits`四个中间位等于1。按位AND运算符组合它们以生成数字`00111100`，该数字等于无符号十进制值`60`：

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

按位或运算符

的*位或运算符*（`|`）两个数的比特进行比较。1如果这两个位1在*任*一输入数中都相等，那么操作员将返回一个新位，该位将被设置为：

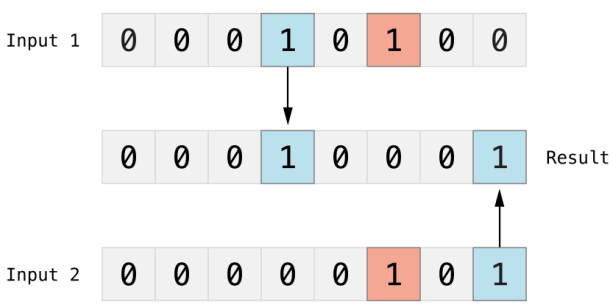


在下面的例子中，的值`someBits`和`moreBits`具有不同的位设置为1。按位或运算符将它们组合成数字`11111110`，该数字等于无符号十进制数`254`：

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```

按位XOR运算符

的*按位XOR运算符*，或“异或运算符”（`^`），比较两个数的位。运算符返回一个新的数字，其位设置为1输入位不同，并设置为`0`输入位相同的位置：



在下面的例子中，`firstBits`和其中的`otherBits`每一个的值都设置1在另一个位置。按位XOR运算符将这两个位设置1为其输出值。所有其他位`firstBits`和`otherBits`匹配并设置为`0`输出值：

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

按位左右移位运算符

在按位左移位运算符（<<）和*逐位向右移位运算符*（>>）中的数向左或由特定数量的地方向右移动的所有位，根据下面定义的规则。

按位左移和右移具有将整数乘以或除以因子2的效果。将一个整数位向左移动一个位置使其值翻倍，而向右移动一个位置则将其值减半。

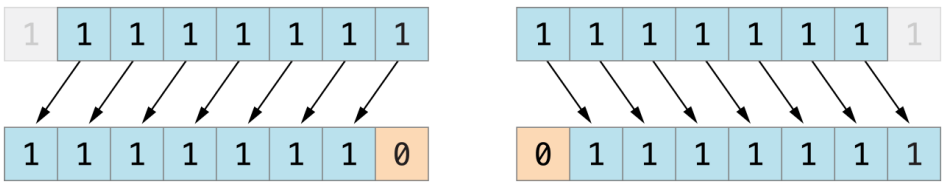
无符号整数的移位行为

无符号整数的位移行为如下所示：

1. 现有的位按照请求的位数向左或向右移动。
2. 丢弃超出整数存储范围的任何位。
3. 将原始位移动到左侧或右侧后，将零插入左侧空间中。

这种方法被称为*逻辑转换*。

下图显示11111111 << 1（11111111按1地点向左移动）和11111111 >> 1（11111111按1地点向右移动）的结果。蓝色的数字被移位，灰色的数字被丢弃，橙色的零被插入：



以下是Swift代码中移位的方式：

```
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

您可以使用位移对其他数据类型中的值进行编码和解码：

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF           // blueComponent is 0x99, or 153
```

此示例使用UInt32调用的常量pink来为粉红色存储层叠样式表颜色值。CSS颜色值#CC6699按照0xCC6699Swift的十六进制数字表示形式写入。然后通过按位AND运算符（）和按位右移运算符（）将该颜色分解为其红色（CC），绿色（66）和蓝色（99）分量。 &>>

红色分量是通过在数字0xCC6699和数字之间执行按位“与”来获得的0xFF0000。零0xFF0000有效地“屏蔽”第二个和第三个字节0xCC6699，导致6699被忽略并离开0xCC0000结果。

然后这个数字向右移动16个位置（>> 16）。十六进制数字中的每对字符使用8位，因此向右移动16位将转换0xCC0000为0x0000CC。这与0xCC小数点的值相同204。

类似地，绿色成分通过执行按位与数字之间获得0xCC6699和0x00FF00，其给出的输出值0x006600。这个输出值然后向右移动八个位置，给出0x66一个十进制值为的值102。

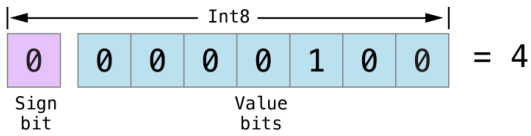
最后，将蓝色成分是通过执行按位与数字之间获得0xCC6699和0x0000FF，其给出的输出值0x000099。没有必要把它转移到右边，因为0x000099已经等于0x99，它的十进制值是153。

签名整数的移位行为

对于有符号整数，移位行为比无符号整数更复杂，因为有符号整数以二进制表示。（为简单起见，以下示例基于8位有符号整数，但相同的原则适用于任何大小的有符号整数。）

带符号的整数使用它们的第一位（称为*符号位*）来指示整数是正数还是负数。符号位0表示正数，符号位1表示负数。

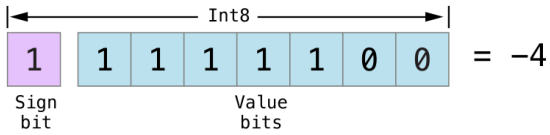
剩余的位（称为*值位*）存储实际值。正数与无符号整数的存储方式完全相同，从上向上计数0。以下是Int8查找数字的位数4：



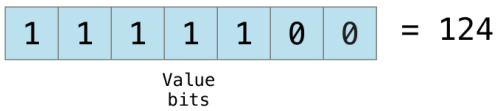
符号位是0（意思是“正”），并且七个值位仅仅是4用二进制表示法写的数字。

但是，负数存储的方式不同。它们通过从减去它们绝对值存储2给力n，其中n是值的位数。一个八位数有七个数值位，所以这意味着2权力7，或128。

以下是Int8查找数字的位数–4：

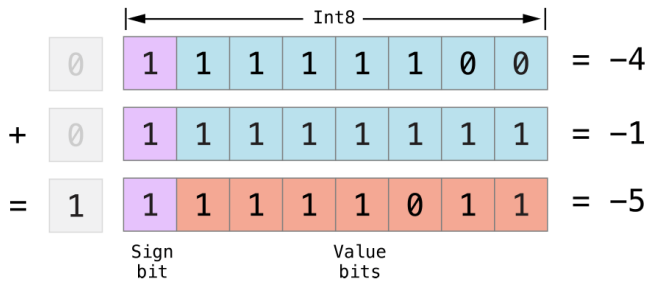


这一次，符号位是1（意思是“否定的”），并且七个值位具有二进制值124（它是128 – 4）：

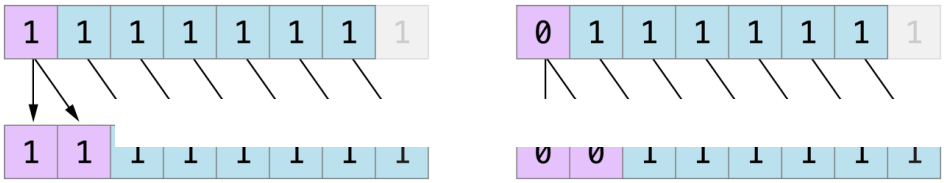


这种用于负数的编码被称为*二进制补码*表示。这似乎是一种不寻常的方式来表示负数，但它有几个优点。

首先，您可以添加–1到–4，简单地通过执行一个标准二进制加法全部八个位（包括符号位），并丢弃任何不适合在八位一旦你完成：



其次，二进制补码表示法还可以让您将负数的位移到正数的左侧和右侧，并且在您向左移动的每个移位时仍然会将其加倍，或者在右移到右侧时将其减半。为了达到这个目的，当有符号整数向右移动时使用额外的规则：当你将有符号整数移到右边时，应用与无符号整数相同的规则，但是用*符号位*填充左边的空白*位*，而是比零。



此操作可确保有符号整数在向右移位后具有相同符号，并称为*算术移位*。

由于存储正数和负数的特殊方式，将它们中的任何一个移到右边都会使它们靠近零。在这种转变过程中保持符号位相同意味着当它们的值接近零时，负整数保持负值。

溢出操作符

如果您尝试将数字插入到不能保存该值的整数常量或变量中，则默认情况下，Swift会报告错误而不是允许创建无效值。当您使用太大或太小的数字时，此行为会提供额外的安全性。

例如，`Int16`整数类型可以包含`-32768`和之间的任何有符号整数`32767`。尝试将`Int16`常量或变量设置为此范围之外的数字会导致错误：

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value an Int16 can hold
potentialOverflow += 1
// this causes an error
```

当值变得太大或者太小时提供错误处理在编码边界值条件时给你更大的灵活性。

但是，当您特别想要溢出条件截断可用位数时，可以选择此行为而不是触发错误。Swift提供了三个算术*溢出运算符*，它们选择整数计算的溢出行为。这些运算符都以“&”符号开头（`&`）：

- 溢出添加（`&+`）
- 溢出减法（`&-`）
- 溢出乘法（`&*`）

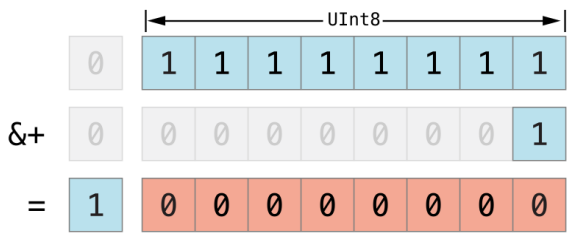
值溢出

数字可以在正向和负向两个方向溢出。

下面是一个例子，说明当一个无符号整数被允许在正方向溢出时使用溢出加法运算符（`&+`）：

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

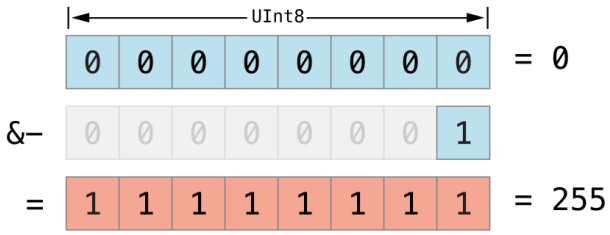
该变量`unsignedOverflow`以`UInt8`可容纳的最大值（`255`或`11111111`以二进制形式）进行初始化。然后通过`1`使用溢出加法运算符（`&+`）来递增。这会将其二进制表示的大小`UInt8`超过其所能容纳的大小，导致它超出其边界，如下图所示。保留在`UInt8`溢出添加之后的`00000000`值为零，或者为零。



当一个无符号整数被允许向负方向溢出时，会发生类似的情况。下面是一个使用溢出减法运算符（`&-`）的例子：

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsi
```

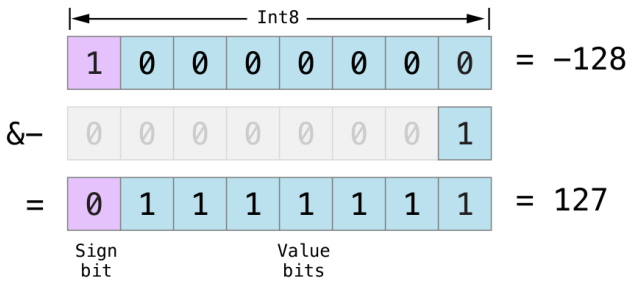
一个UInt8可以容纳的最小值是零，或者00000000二进制。如果1从00000000使用溢出减法运算符（&-）中减去该数字，则该数字将溢出并绕回至11111111或255以十进制形式。



对于有符号整数也会发生溢出。有符号整数的所有加法和减法均按位方式执行，符号位作为数字的一部分被添加或减去，如按位左移和右移操作符中所述。

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

Int8可以容纳 的最小值是-128或者10000000二进制。1用溢出运算符从这个二进制数中减去一个二进制值01111111，它将切换符号位并给出肯定的127，即Int8可以保持的最大正值。



对于有符号整数和无符号整数，正向溢出从最大有效整数值回到最小值，负向溢出从最小值回绕到最大值。

优先和相关性

运营商*优先级*给一些运营商比其他运营商更高的优先级 首先应用这些运营商。

运算符*关联*定义了相同优先级的运算符如何组合在一起 - 从左边分组，或从右边分组。把它看作是“他们与他们左边的表达联系在一起”的意思，或者“他们把这个表达联系到他们的右边”。

在计算复合表达式的计算顺序时，考虑每个运算符的优先级和相关性非常重要。例如，运算符优先级解释了为什么以下表达式等于17。

```
2 + 3 % 4 * 5
// this equals 17
```

如果你严格按照从左到右的顺序阅读，你可能会预期表达式的计算如下：

- 2加上3等于5
- 5余数4相等1
- 1次数5等于5

但是，实际的答案是17，没有5。高优先级运算符在低优先级运算符之前进行评估。在Swift中，与C中一样，余数运算符（%）和乘法运算符（*）具有比加法运算符（+）更高的优先级。因此，在考虑添加之前对它们进行评估。

但是，余数和乘
法都与左边的表达联系在一起。把它看作是从表达式的这些部分开始，从左边开始添加隐式圆括号：

```
2 + ((3 % 4) * 5)
```

(3 % 4)是3，所以这相当于：

```
2 + (3 * 5)
```

(3 * 5)是15，所以这相当于：

```
2 + 15
```

这个计算得出最终答案17。

有关Swift标准库提供的运算符的信息，包括运算符优先级组和关联设置的完整列表，请参阅运算符声明。

> 注意
>
> Swift的运算符优先级和关联性规则比C和Objective-C中的那些规则更简单和更可预测。但是，这意味着它们与基于C的语言不完全相同。注意确保操作符交互仍然按照您将现有代码移植到Swift时的方式运行。

运算符方法

类和结构可以提供他们自己的现有操作符的实现。这被称为*重载*现有的操作员。

下面的例子显示了如何+为自定义结构实现算术加法运算符（）。算术加法运算符是一个*二元运算符*，因为它在两个目标上运行，并且被称为中*缀*，因为它出现在这两个目标之间。

该示例定义了Vector2D一个二维位置向量的结构(x, y)，后面跟着一个*运算符方法*的定义，以将该Vector2D结构的实例添加到一起：

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
}
```

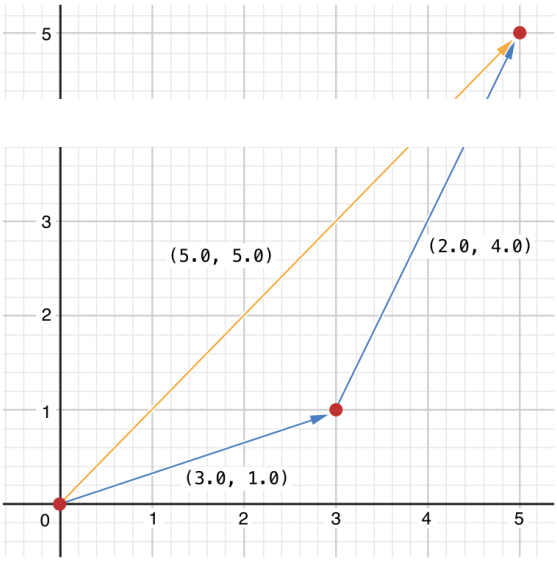
运算符方法被定义为一个类型方法Vector2D，方法名称与运算符重载（+）相匹配。由于添加不是矢量的基本行为的一部分，所以类型方法是在扩展Vector2D而不是在主结构声明中定义的Vector2D。由于算术加法运算符是二元运算符，因此此运算符方法接受两个输入参数类型Vector2D并返回单个输出值，也是类型Vector2D。

在这个实现中，输入参数被命名left并且right表示Vector2D将在+操作符的左侧和右侧的实例。该方法返回一个新Vector2D实例，其x和y性质随的总和被初始化x并且y从两个属性Vector2D，它们加到一起实例。

类型方法可以用作现有Vector2D实例之间的中缀运算符：

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

这个例子将这些向量加起来(3.0, 1.0)并(2.0, 4.0)构成向量(5.0, 5.0)，如下所示。



前缀和后缀运算符

上面显示的示例演示了二进制中缀运算符的自定义实现。类和结构也可以提供标准*一元运算符的实现*。一元操作符在单个目标上操作。他们是*前缀*，如果他们先于他们的目标（如`-a`）和*后缀*运营商，如果他们遵循自己的目标（如`b!`）。

在声明操作符方法时，通过在关键字之前 写入`prefix`或`postfix`修饰符来实现前缀或后缀一元运算`func`符：

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}
```

上面的例子`-a`为`Vector2D`实例实现了一元减运算符（）。一元减号运算符是一个前缀运算符，因此该方法必须使用`prefix`修饰符进行限定。

对于简单的数值，单目减号运算符将正数转换为负数，反之亦然。`Vector2D`实例的相应实现对`x`和`y`属性都执行此操作：

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

复合分配算子

*复合赋值操作符*将assignment（=）与另一个操作组合在一起 例如，添加赋值运算符（+=）将添加和赋值组合到一个单独的操作中。您将复合赋值运算符的左输入参数类型标记为`inout`，因为该参数的值将直接从运算符方法中修改。

下面的例子为`Vector2D`实例实现了一个添加赋值操作符方法：

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}
```

由于添加运算符的定义较早，因此不需要在此重新实现添加过程。相反，添加赋值运算符方法利用了现有的添加运算符方法，并使用它将左值设置为左值和右值：

```
var original = Vector2D(x: 1.0, y: 2.0)
```

```
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

> 注意
>
> 不能使默认赋值运算符（=）过载。只有复合赋值运算符可以被重载。同样，三元条件运算符（a ? b : c）也不能重载。

等价算子

默认情况下，自定义类和结构不会收到*等效运算符*的默认实现，称为“等于”运算符（==）和“不等于”运算符（!=）。

要使用等价运算符来检查自定义类型的等价性，请按照与其他中缀运算符相同的方式提供“等于”运算符的实现，并添加符合标准库的Equatable协议：

```
extension Vector2D: Equatable {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
}
```

上面的例子实现了一个“等于”运算符（==）来检查两个Vector2D实例是否有等价值。在上下文中Vector2D，将“相等”看作“两个实例具有相同的x值和y值” 意义是合理的，因此这是操作员实现所使用的逻辑。标准库提供了“不等于”运算符（!=）的默认实现，它只是返回“等于”运算符结果的逆。

您现在可以使用这些运算符来检查两个Vector2D实例是否相同：

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// Prints "These two vectors are equivalent."
```

Swift为以下类型的自定义类型提供了等价运算符的综合实现：

- 只存储符合Equatable协议的结构
- 仅具有符合Equatable协议的 关联类型的枚举
- 没有关联类型的枚举

声明Equatable符合性是类型原始声明的一部分，以接收这些默认实现。

下面的例子定义了Vector3D三维位置矢量(x, y, z)的Vector2D结构，类似于结构。因为x，y和z性能都是一个的Equatable类型，Vector3D接收等价运营商的默认实现。

```
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")
}
// Prints "These two vectors are also equivalent."
```

自定义操作符

除了Swift提供的标准运算符之外，您还可以声明和实现自己的*自定义运算*符。有关可用于定义自定义运算符的字符列表，请参阅运算符。

新的运营商使用的是在全球范围内声明的operator关键字，并且都标有prefix，infix或postfix修饰：

```
prefix operator +++
```

上面的例子定义了一个新的前缀运算符+++。此运算符在Swift中没有现有含义，因此在使用Vector2D实例的特定上下文中给出它自己的自定义含义。就本例而言，+++被视为新的“前缀加倍”运算符。它通过使用前面定义的附加赋值运算符将向量添加到自身来使实例的值x和y值加倍Vector2D。要实现该+++运算符，需要添加一个名为+++to 的类型方法Vector2D，如下所示：

```
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

在本页

自定义中缀操作符的优先级

自定义中缀运算符都属于一个优先组。优先组指定运算符相对于其他中缀运算符的优先级，以及运算符的关联性。有关这些特性如何影响中缀操作员与其他中缀操作员的交互的说明，请参阅优先级和关联性。

未明确放置到优先组中的自定义中缀运算符被赋予一个优先级高于三元条件运算符优先级的默认优先级组。

以下示例定义了一个名为的新自定义中缀运算符+-，它属于优先级组AdditionPrecedence：

```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

该运算符将x两个向量的值相加，并y从第一个向量中减去第二个向量的值。因为它本质上是一个“加法”运算符，所以它被赋予与加法运算符如+和的相同的优先级组-。有关Swift标准库提供的运算符的信息，包括运算符优先级组和关联设置的完整列表，请参阅运算符声明。有关优先级组的更多信息，并查看定义您自己的运算符和优先级组的语法，请参阅运算符声明。

> 注意
>
> 定义前缀或后缀运算符时，不要指定优先顺序。但是，如果将前缀和后缀运算符应用于相同的操作数，则首先应用后缀运算符。