翻译：xielingwang　校对：numbbbbb

高级运算符

本页内容包括：

除了基本操作符中所讲的运算符，Swift还有许多复杂的高级运算符，包括了C语和Objective-C中的位运算符和移位运算。

不同于C语言中的数值计算，Swift的数值计算默认是不可溢出的。溢出行为会被捕获并报告为错误。你是故意的？好吧，你可以使用Swift为你准备的另一套默认允许溢出的数值运算符，如可溢出加&+。所有允许溢出的运算符都是以&开始的。

自定义的结构，类和枚举，是否可以使用标准的运算符来定义操作？当然可以！在Swift中，你可以为你创建的所有类型定制运算符的操作。

可定制的运算符并不限于那些预设的运算符，自定义有个性的中置，前置，后置及赋值运算符，当然还有优先级和结合性。这些运算符的实现可以运用预设的运算符，也可以运用之前定制的运算符。

位运算符

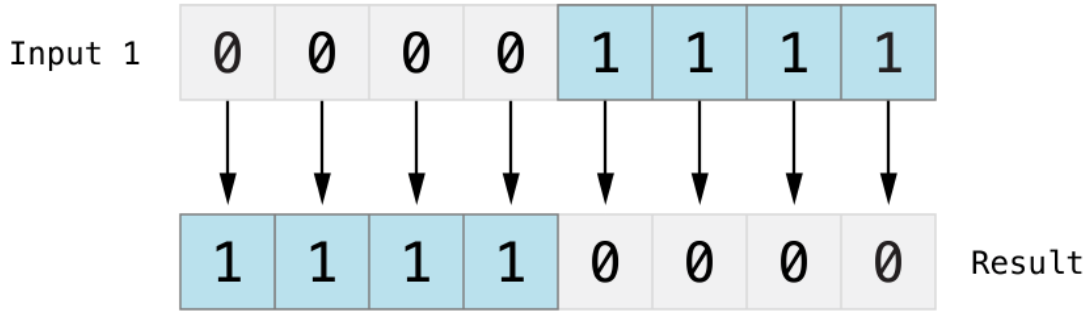
位操作符通常在诸如图像处理和创建设备驱动等底层开发中使用，使用它可以单独操作数据结构中原始数据的比特位。在使用一个自定义的协议进行通信的时候，运用位运算符来对原始数据进行编码和解码也是非常有效

的。

Swift支持如下所有C语言的位运算符：

按位取反运算符

按位取反运算符~对一个操作数的每一位都取反。



这个运算符是前置的，所以请不加任何空格地写着操作数之前。

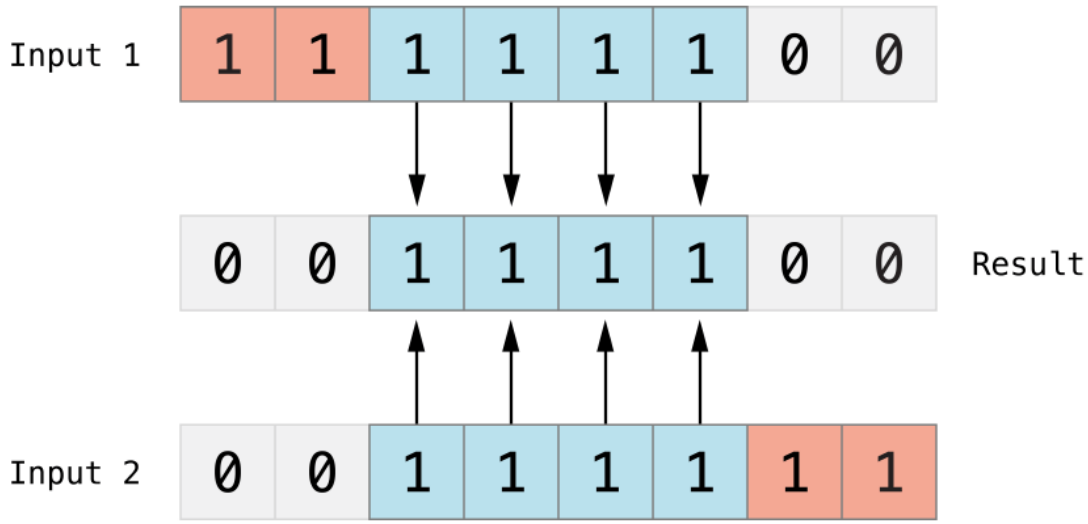
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // 等于 0b11110000
```

UInt8是8位无符整型，可以存储0~255之间的任意数。这个例子初始化一个整型为二进制值00001111(前4位为0，后4位为1)，它的十进制值为15。

使用按位取反运算~对initialBits操作，然后赋值给invertedBits这个新常量。这个新常量的值等于所有位都取反的initialBits，即1变成0，0变成1，变成了11110000，十进制值为240。

按位与运算符

按位与运算符对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1时才为1。

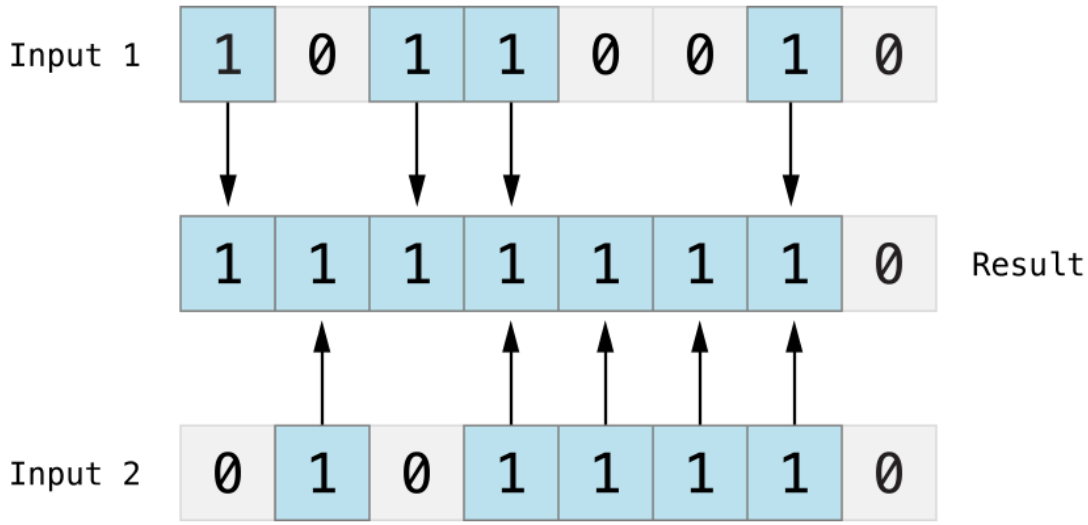


以下代码，`firstSixBits`和`lastSixBits`中间4个位都为1。对它俩进行按位与运算后，就得到了`00111100`，即十进制的`60`。

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

按位或运算

按位或运算符`|`比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是两个输入数的同一位都不为0(即任意一个为1，或都为1)。

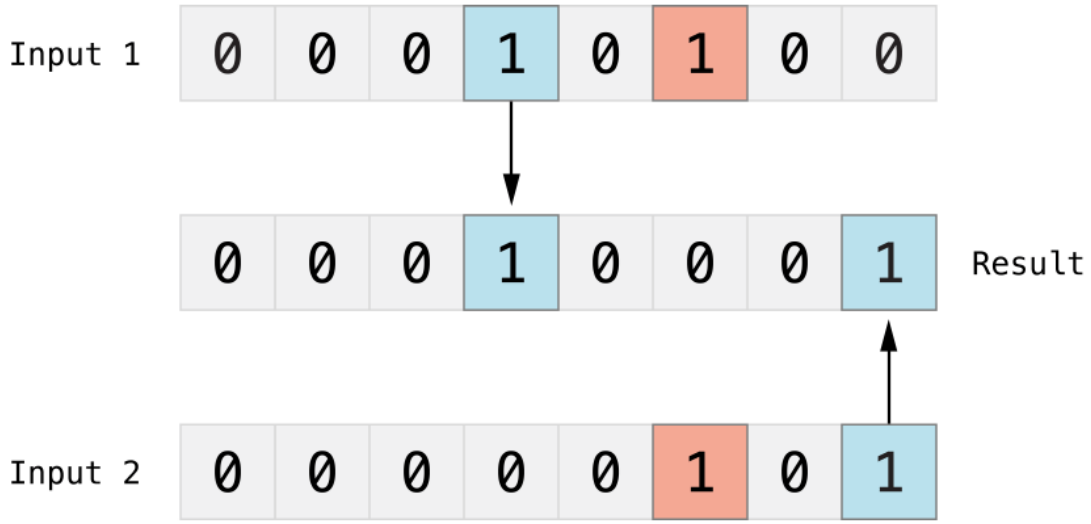


如下代码，`someBits`和`moreBits`在不同位上有`1`。按位或运行的结果是`11111110`，即十进制的`254`。

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

按位异或运算符

按位异或运算符^比较两个数，然后返回一个数，这个数的每个位设为1的条件是两个输入数的同一位不同，如果相同就设为0。



以下代码，firstBits和otherBits都有一个1跟另一个数不同的。所以按位异或的结果是把它这些位置为1，其他都置为0。

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

按位左移/右移运算符

左移运算符<<和右移运算符>>会把一个数的所有比特位按以下定义的规则向左或向右移动指定位数。

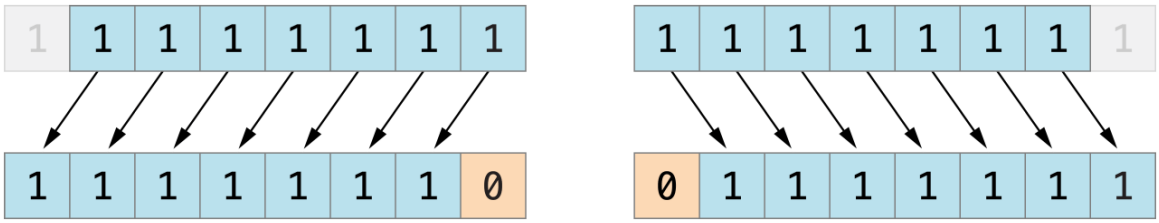
按位左移和按位右移的效果相当把一个整数乘于或除于一个因子为2的整数。向左移动一个整型的比特位相当于把这个数乘于2，向右移一位就是除于2。

无符整型的移位操作

对无符整型的移位的效果如下：

已经存在的比特位向左或向右移动指定的位数。被移出整型存储边界的的位数直接抛弃，移动留下的空白位用零0来填充。这种方法称为逻辑移位。

以下这张把展示了 11111111 << 1(11111111向左移1位)，和 11111111 >> 1(11111111向右移1位)。蓝色的是被移位的，灰色是被抛弃的，橙色的0是被填充进来的。




```
let shiftBits: UInt8 = 4   // 即二进制的00000100
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

你可以使用移位操作进行其他数据类型的编码和解码。

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

这个例子使用了一个UInt32的命名为pink的常量来存储层叠样式表CSS中粉色的颜色值，CSS颜色#CC6699在Swift用十六进制0xCC6699来表示。然后使用按位与(&)和按位右移就可以从这个颜色值中解析出红(CC)，绿(66)，蓝(99)三个部分。

对0xCC6699和0xFF0000进行按位与&操作就可以得到红色部分。0xFF0000中的0了遮盖了0xCC6699的第二和第三个字节，这样6699被忽

略了，只留下`0xCC0000`。

然后，按向右移动16位，即`>> 16`。十六进制中每两个字符是8比特位，所以移动16位的结果是把`0xCC0000`变成`0x0000CC`。这和`0xCC`是相等的，都是十进制的`204`。

同样的，绿色部分来自于`0xCC6699`和`0x00FF00`的按位操作得到`0x006600`。然后向右移动8們，得到`0x66`，即十进制的`102`。

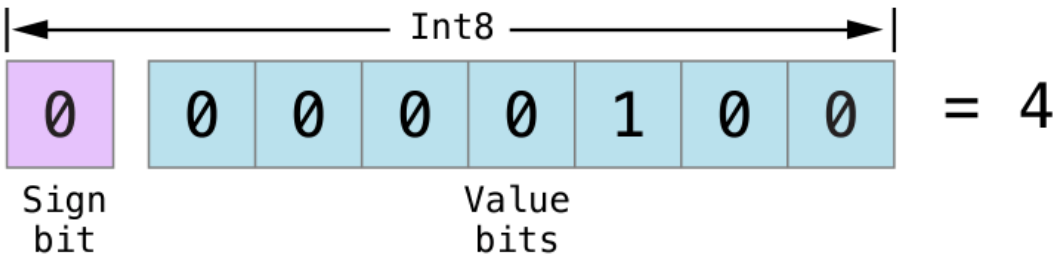
最后，蓝色部分对`0xCC6699`和`0x0000FF`进行按位与运算，得到`0x000099`，无需向右移位了，所以结果就是`0x99`，即十进制的`153`。

有符整型的移位操作

有符整型的移位操作相对复杂得多，因为正负号也是用二进制位表示的。(这里举的例子虽然都是8位的，但它的原理是通用的。)

有符整型通过第1个比特位(称为符号位)来表达这个整数是正数还是负数。`0`代表正数，`1`代表负数。

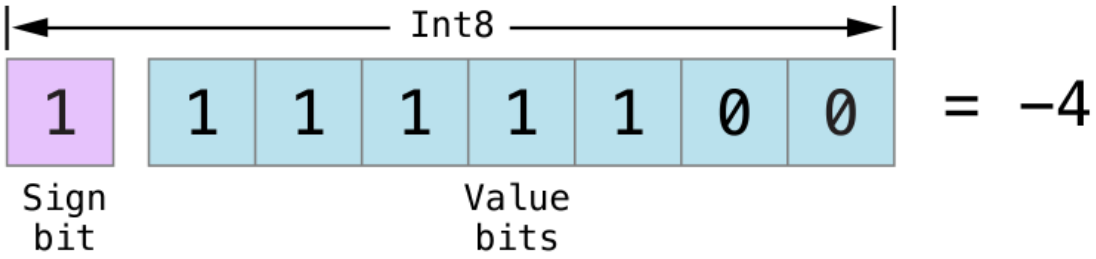
其余的比特位(称为数值位)存储其实值。有符正整数和无符正整数在计算机里的存储结果是一样的，下来我们来看`+4`内部的二进制结构。



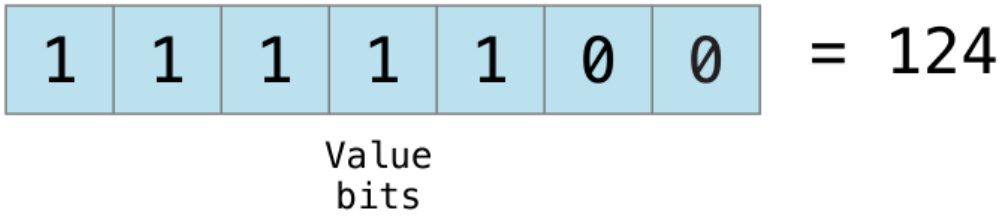
符号位为`0`，代表正数，另外7比特位二进制表示的实际值就刚好是`4`。

负数呢，跟正数不同。负数存储的是2的n次方减去它的绝对值，n为数值位的位数。一个8比特的数有7个数值位，所以是2的7次方，即128。

我们来看`-4`存储的二进制结构。

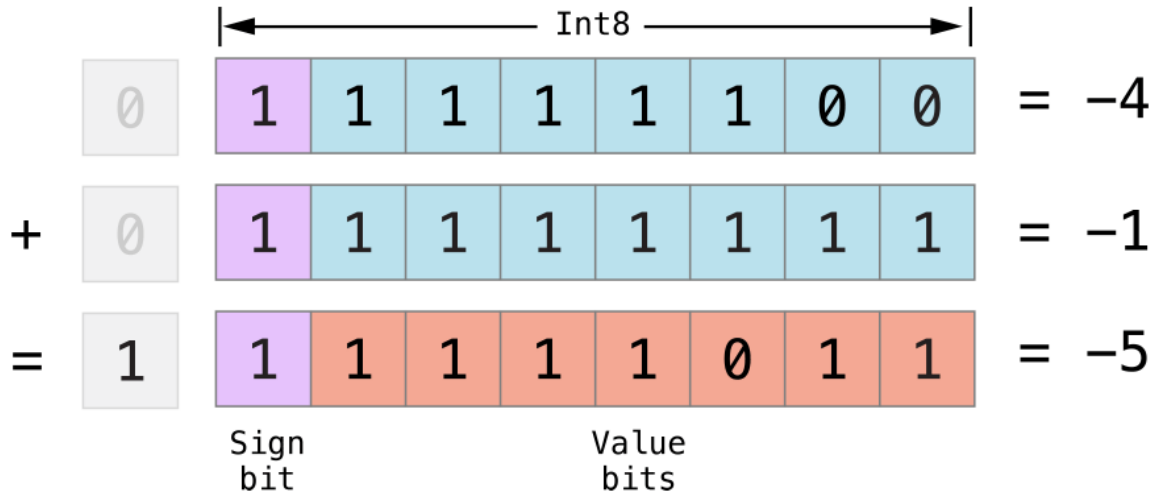


现在符号位为1，代表负数，7个数值位要表达的二进制值是124，即128 - 4。



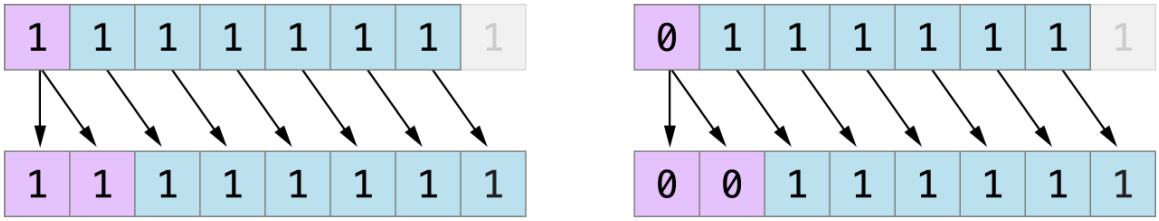
负数的编码方式称为二进制补码表示。这种表示方式看起来很奇怪，但它有几个优点。

首先，只需要对全部8个比特位(包括符号)做标准的二进制加法就可以完成 `-1 + -4` 的操作，忽略加法过程产生的超过8个比特位表达的任何信息。



第二，由于使用二进制补码表示，我们可以和正数一样对负数进行按位左移右移的，同样也是左移1位时乘于2，右移1位时除于2。要达到此目的，对有符整型的右移有一个特别的要求：

对有符整型按位右移时，使用符号位(正数为0，负数为1)填充空白位。



这就确保了在右移的过程中，有符整型的符号不会发生变化。这称为算术移位。

正因为正数和负数特殊的存储方式，向右移位使它接近于`0`。移位过程中保持符号会不变，负数在接近`0`的过程中一直是负数。

溢出运算符

默认情况下，当你往一个整型常量或变量赋于一个它不能承载的大数时，Swift不会让你这么干的，它会报错。这样，在操作过大或过小的数的时候就很安全了。

例如，`Int16`整型能承载的整数范围是`-32768`到`32767`，如果给它赋上超过这个范围的数，就会报错：

```
var potentialOverflow = Int16.max
// potentialOverflow 等于 32767, 这是 Int16 能承载的最大整数
potentialOverflow += 1
// 噢, 出错了
```

对过大或过小的数值进行错误处理让你的数值边界条件更灵活。

当然，你有意在溢出时对有效位进行截断，你可采用溢出运算，而非错误处理。Swfit为整型计算提供了5个`&`符号开头的溢出运算符。

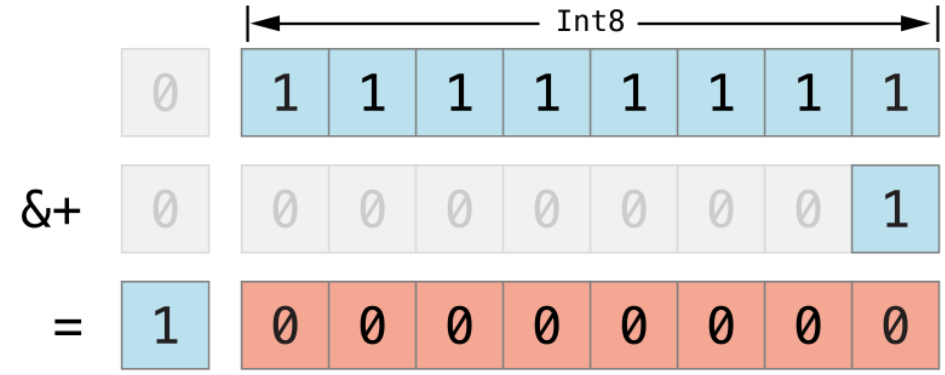
- 溢出加法 `&+`
- 溢出减法 `&-`
- 溢出乘法 `&*`
- 溢出除法 `&/`
- 溢出求余 `&%`

值的上溢出

下面例子使用了溢出加法&+来解剖的无符整数的上溢出

```
var willOverflow = UInt8.max
// willOverflow 等于UInt8的最大整数 255
willOverflow = willOverflow &+ 1
// 这时候 willOverflow 等于 0
```

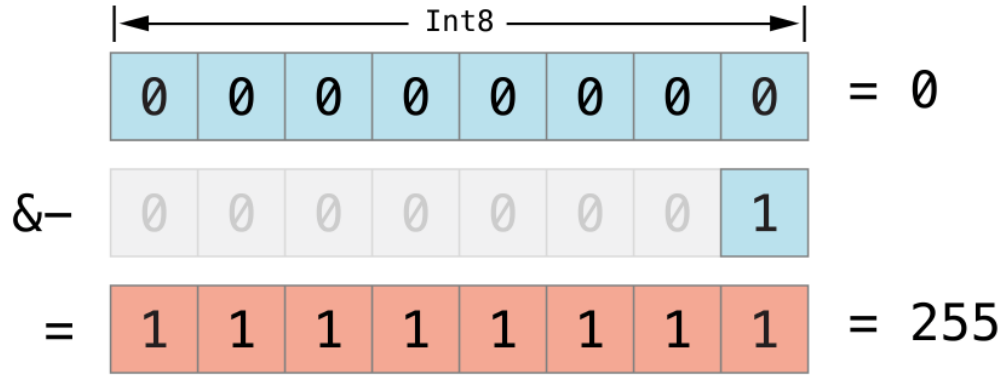
willOverflow用Int8所能承载的最大值255(二进制11111111)，然后用&+加1。然后UInt8就无法表达这个新值的二进制了，也就导致了这个新值上溢出了，大家可以看下图。溢出后，新值在UInt8的承载范围内的那部分是00000000，也就是0。



值的下溢出

数值也有可能因为太小而越界。举个例子:

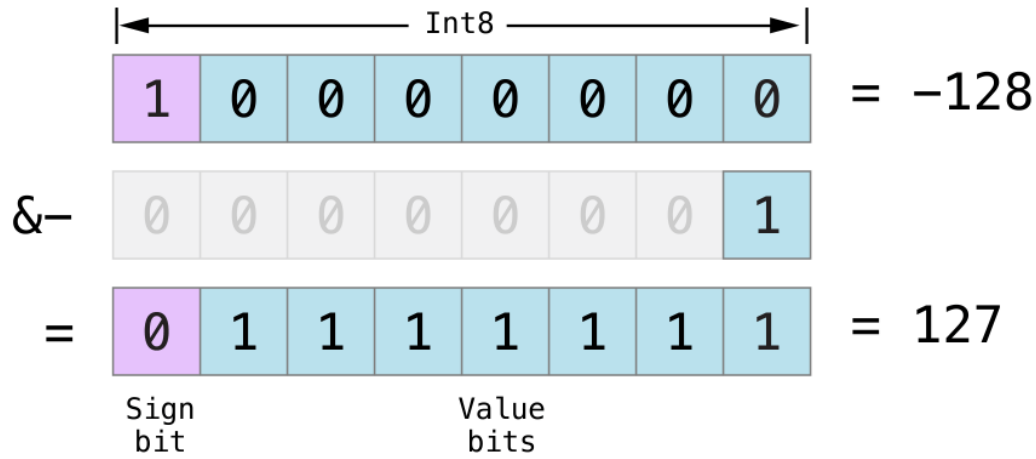
UInt8的最小值是0(二进制为00000000)。使用&-进行溢出减1，就会得到二进制的11111111即十进制的255。



Swift代码是这样的:

```
var willUnderflow = UInt8.min
// willUnderflow 等于UInt8的最小值0
willUnderflow = willUnderflow &- 1
// 此时 willUnderflow 等于 255
```

有符整型也有类似的下溢出，有符整型所有的减法也都是对包括在符号位在内的二进制数进行二进制减法的，这在 "按位左移/右移运算符" 一节提到过。最小的有符整数是`-128`，即二进制的`10000000`。用溢出减法减去去1后，变成了`01111111`，即UInt8所能承载的最大整数`127`。



来看看Swift代码:

```
var signedUnderflow = Int8.min
// signedUnderflow 等于最小的有符整数 -128
signedUnderflow = signedUnderflow &- 1
// 如今 signedUnderflow 等于 127
```

除零溢出

一个数除于0 `i / 0`，或者对0求余数 `i % 0`，就会产生一个错误。

```
let x = 1
let y = x / 0
```

使用它们对应的可溢出的版本的运算符`&/`和`&%`进行除0操作时就会得到`0`值。

```
let x = 1
let y = x &/ 0
```

```
// y 等于 0
```

优先级和结合性

运算符的优先级使得一些运算符优先于其他运算符，高优先级的运算符会先被计算。

结合性定义相同优先级的运算符在一起时是怎么组合或关联的，是和左边的一组呢，还是和右边的一组。意思就是，到底是和左边的表达式结合呢，还是和右边的表达式结合?

在混合表达式中，运算符的优先级和结合性是非常重要的。举个例子，为什么下列表达式的结果为`4`?

```
2 + 3 * 4 % 5
// 结果是 4
```

如果严格地从左计算到右，计算过程会是这样：

- 2 plus 3 equals 5;
- 2 + 3 = 5
- 5 times 4 equals 20;
- 5 * 4 = 20
- 20 remainder 5 equals 0
- 20 / 5 = 4 余 0

但是正确答案是`4`而不是`0`。优先级高的运算符要先计算，在Swift和C语言中，都是先乘除后加减的。所以，执行完乘法和求余运算才能执行加减运算。

乘法和求余拥有相同的优先级，在运算过程中，我们还需要结合性，乘法和求余运算都是左结合的。这相当于在表达式中有隐藏的括号让运算从左开始。

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

(3 4) is 12, so this is equivalent to: 3 4 = 12，所以这相当于：

```
2 + (12 % 5)
```

(12 % 5) is 2, so this is equivalent to: 12 % 5 = 2，所这又相当于

```
2 + 2
```

计算结果为 4。

查阅Swift运算符的优先级和结合性的完整列表，请看表达式。

注意：

Swift的运算符较C语言和Objective-C来得更简单和保守，这意味着跟基于C的语言可能不一样。所以，在移植已有代码到Swift时，注意去确保代码按你想的那样去执行。

运算符函数

让已有的运算符也可以对自定义的类和结构进行运算，这称为运算符重载。

这个例子展示了如何用`+`让一个自定义的结构做加法。算术运算符`+`是一个两目运算符，因为它有两个操作数，而且它必须出现在两个操作数之间。

例子中定义了一个名为`Vector2D`的二维坐标向量`(x, y)`的结构，然后定义了让两个`Vector2D`的对象相加的运算符函数。

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
@infix func + (left: Vector2D, right: Vector2D) ->
Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y +
right.y)
}
```

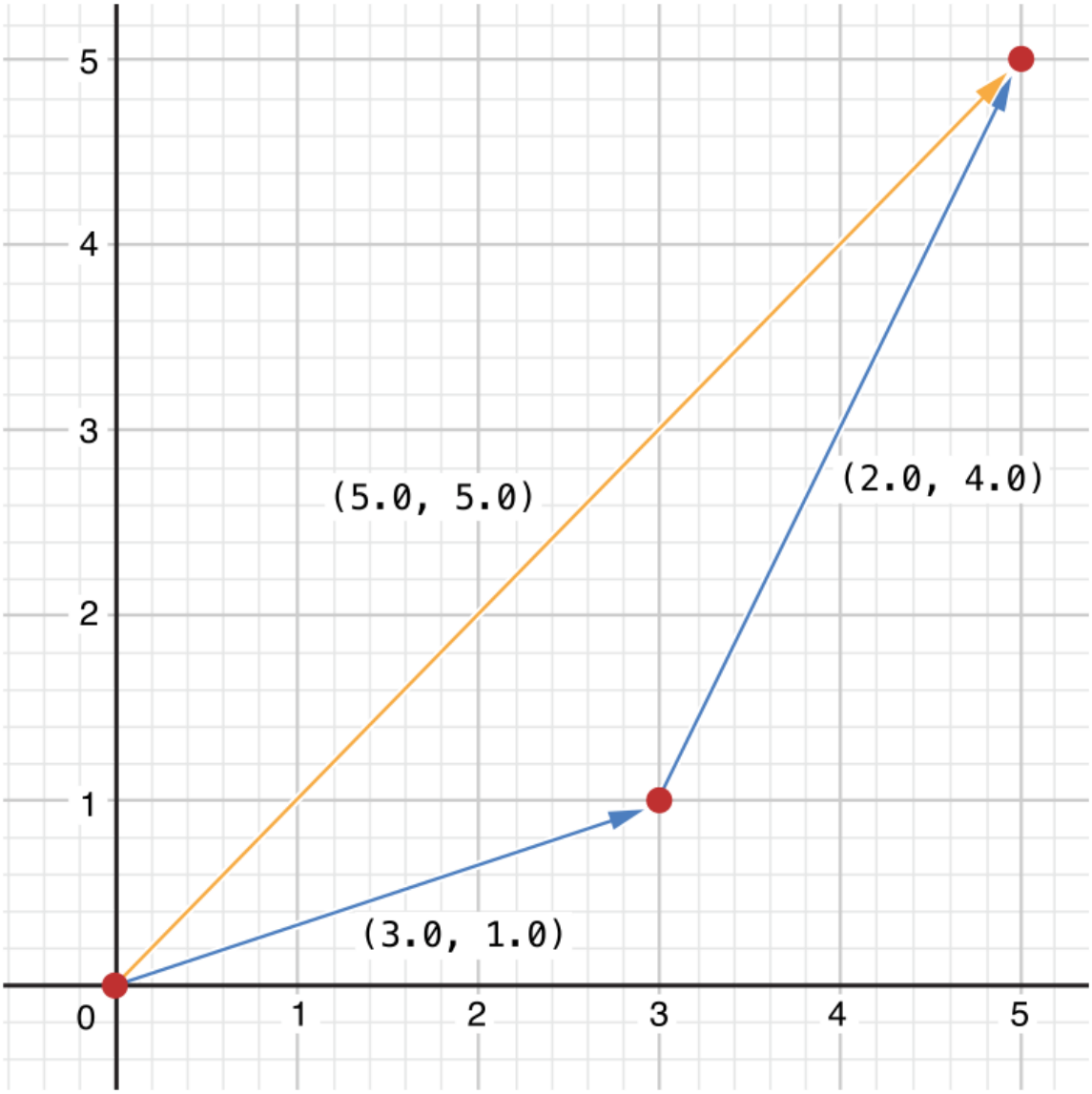
该运算符函数定义了一个全局的`+`函数，这个函数需要两个`Vector2D`类型的参数，返回值也是`Vector2D`类型。需要定义和实现一个中置运算的时候，在关键字`func`之前写上属性`@infix`就可以了。

在这个代码实现中，参数被命名为了`left`和`right`，代表`+`左边和右边的两个`Vector2D`对象。函数返回了一个新的`Vector2D`的对象，这个对象的`x`和`y`分别等于两个参数对象的`x`和`y`的和。

这个函数是全局的，而不是`Vector2D`结构的成员方法，所以任意两个`Vector2D`对象都可以使用这个中置运算符。

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector 是一个新的Vector2D, 值为 (5.0, 5.0)
```

这个例子实现两个向量`(3.0, 1.0)`和`(2.0, 4.0)`相加，得到向量`(5.0, 5.0)`的过程。如下图示：



前置和后置运算符

上个例子演示了一个双目中置运算符的自定义实现，同样我们也可以玩标准单目运算符的实现。单目运算符只有一个操作数，在操作数之前就是前置的，如`-a`; 在操作数之后就是后置的，如`i++`。

实现一个前置或后置运算符时，在定义该运算符的时候于关键字`func`之前标注 `@prefix` 或 `@postfix` 属性。

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

这段代码为`Vector2D`类型提供了单目减运算`-a`，`@prefix`属性表明这是个前置运算符。

对于数值，单目减运算符可以把正数变负数，把负数变正数。对于`Vector2D`，单目减运算将其`x`和`y`都进进行单目减运算。

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative 为 (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive 为 (3.0, 4.0)
```

组合赋值运算符

组合赋值是其他运算符和赋值运算符一起执行的运算。如`+=`把加运算和赋值运算组合成一个操作。实现一个组合赋值符号需要使用`@assignment`属性，还需要把运算符的左参数设置成`inout`，因为这个参数会在运算符函数内直接修改它的值。

```
@assignment func += (inout left: Vector2D, right:
Vector2D) {
    left = left + right
}
```

因为加法运算在之前定义过了，这里无需重新定义。所以，加赋运算符函数使用已经存在的高级加法运算符函数来执行左值加右值的运算。

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original 现在为 (4.0, 6.0)
```

你可以将`@assignment`属性和`@prefix`或`@postfix`属性起来组合，实现一个`Vector2D`的前置运算符。

```
@prefix @assignment func ++ (inout vector: Vector2D)
-> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

这个前置使用了已经定义好的高级加赋运算，将自己加上一个值为`(1.0, 1.0)`的对象然后赋给自己，然后再将自己返回。

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement 现在是 (4.0, 5.0)
// afterIncrement 现在也是 (4.0, 5.0)
```

注意:

默认的赋值符是不可重载的。只有组合赋值符可以重载。三目条件运算符`a? b: c`也是不可重载。

比较运算符

Swift无所知道自定义类型是否相等或不等，因为等于或者不等于由你的代码说了算了。所以自定义的类和结构要使用比较符`==`或`!=`就需要重载。

定义相等运算符函数跟定义其他中置运算符雷同:

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

@infix func != (left: Vector2D, right: Vector2D) ->
Bool {
    return !(left == right)
```

```
}
```
上述代码实现了相等运算符==来判断两个Vector2D对象是否有相等的值，相等的概念就是他们有相同的x值和相同的y值，我们就用这个逻辑来实现。接着使用==的结果实现了不相等运算符!=。

现在我们可以使用这两个运算符来判断两个Vector2D对象是否相等。

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("这两个向量是相等的.")
}
// prints "这两个向量是相等的."
```

自定义运算符

标准的运算符不够玩，那你可以声明一些个性的运算符，但个性的运算符只能使用这些字符 / = - + * % < >! & | ^。~。

新的运算符声明需在全局域使用operator关键字声明，可以声明为前置，中置或后置的。

```
operator prefix +++ {}
```
这段代码定义了一个新的前置运算符叫+++，此前Swift并不存在这个运算符。此处为了演示，我们让+++对Vector2D对象的操作定义为 双自增 这样一个独有的操作，这个操作使用了之前定义的加赋运算实现了自己加上自己然后返回的运算。

```
@prefix @assignment func +++ (inout vector: Vector2D)
-> Vector2D {
    vector += vector
    return vector
}
```
Vector2D 的 +++ 的实现和 ++ 的实现很接近，唯一不同的前者是加自己，后者是加值为 (1.0, 1.0) 的向量.

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled 现在是 (2.0, 8.0)
```

```
// afterDoubling 现在也是 (2.0, 8.0)
```

自定义中置运算符的优先级和结合性

可以为自定义的中置运算符指定优先级和结合性。可以回头看看优先级和结合性解释这两个因素是如何影响多种中置运算符混合的表达式的计算的。

结合性(associativity)的值可取的值有`left`，`right`和`none`。左结合运算符跟其他优先级相同的左结合运算符写在一起时，会跟左边的操作数结合。同理，右结合运算符会跟右边的操作数结合。而非结合运算符不能跟其他相同优先级的运算符写在一起。

结合性(associativity)的值默认为`none`，优先级(precedence)默认为`100`。

以下例子定义了一个新的中置符`+-`，是左结合的`left`，优先级为`140`。

```
operator infix +- { associativity left precedence
140 }
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 此时的值为 (4.0, -2.0)
```

这个运算符把两个向量的`x`相加，把向量的`y`相减。因为他实际是属于加减运算，所以让它保持了和加法一样的结合性和优先级(`left`和`140`)。查阅完整的Swift默认结合性和优先级的设置，请移步表达式;