The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（二）基本运算符

运算符是一种特定的符号或表达式，用来检验、修改或合并变量。例如，用求和运算符 + 可以对两个数字进行求和 (如 let i = 1 + 2)；稍微复杂一点的例子有逻辑与操作符 && (如 if enteredDoorCode && passedRetinaScan) ，自增长运算符 ++i （这是 i=i+1 的简写方式)

Swift 支持 C 标准库中的大多数运算符并提升了各自的兼容性，从而可以排除常见的编码错误。赋值操作符 (=) 不会返回一个值，这样可以防止你因粗心将赋值运算符 (=) 写成 (==) 而引起错误。算术符 (+、-、*、/、% 等) 会检查与驳回值溢出，这样可以避免值类型的数据在超过值类型所允许的存储范围时，出现意想不到的数据。你可以选择使用 Swift 所提供的值溢出运算符进行量化溢出的行为，详细见溢出操作符。

与 C 语言不同，Swift 允许你对浮点数执行取余运算。同时，Swift 提供两个范围的运算符 (a..b 和 a...b)，作为表示一个数值范围的简写方式，这点 C 不支持。

本章节描述了 Swift 常见运算符。高级运算符覆盖了 Swift 的高级操作符，并且对自定义操作符，对自定义类型操作符的实现进行了描述。

1.1 术语

操作符都是一元、二元或三元:

- 一元操作符操作单个对象 (如 -a)。一元前缀操作符出现在对象前 (如!b)，一元后缀操作符在对象后出现 (如 i++)。
- 二元操作符操作两个对象 (如 2 + 3)，并且操作符位于两个元素中间。
- 三元操作符对两个对象进行操作。与 C 一样，Swift 仅支持一个三元操作符：三元条件操作符 (a ? b : c).

操作符所影响的值被称为操作数。表达式 1 + 2 中，符号 + 是一个二元运算符并且两个操作数分别为 1 和 2。

1.2 赋值运算符

赋值运算符 (a = b) 用 b 的值去初始化或更新 a 的值

```
let b = 10
var a = 5
```

```
a = b
// □ □a□ □ □10
```

假如右边赋值的数据为多个数据的元组，它的元素可以是一次性赋给的多个常量或变量

```
let (x, y) = (1, 2)
// x□ □1, □ □y□ □ 2
```

与 C 及 Objective-C 不同，Swift 中赋值运算符并不将自身作为一个值进行返回。所以以下的代码是不合法的：

```
if x = y {
    // □ □, □ □x = y□ □ □ □ □ □ □ □
}
```

此特性帮助你避免因粗心将赋值运算符 (=) 写成 (==) 而引起的错误。因为 if x = y 这样写是无效的。

1.3 数学运算符

Swift 支持所有数字类型的四个标注运算符:

- 加法 (+) * 减法 (-)
- 乘法 (*)
- 除法 (/)

例如：

```
1 + 2 // equals 3
5 - 3 // equals 2
2 * 3 // equals 6
10.0 / 2.5 // equals 4.0
```

不同于 C 和 Objective-C，默认情况下 Swift 的算术运算符不允许值溢出。你可以通过 Swift 的溢出运算符来选择值的溢出情况 (例如 a & + b)。详见 Overflow Operators

加法运算符对字符串连接也一样适用，例如：

```
"hello, " + "world" // equals "hello, world"
```

两个字符，或者一个字符一个字符串，能组合成一个新的字符串：

```
let dog: Character = "dog"( □ □ □ □ □ □ □ □ □ □ □ □unicode □ □ , □ □ □ □ □ □ □ □ … … )
let cow: Character = "cow" ( □ □ … … )
let dogCow = dog + cow
// dogCow is equal to "dogcow"
```

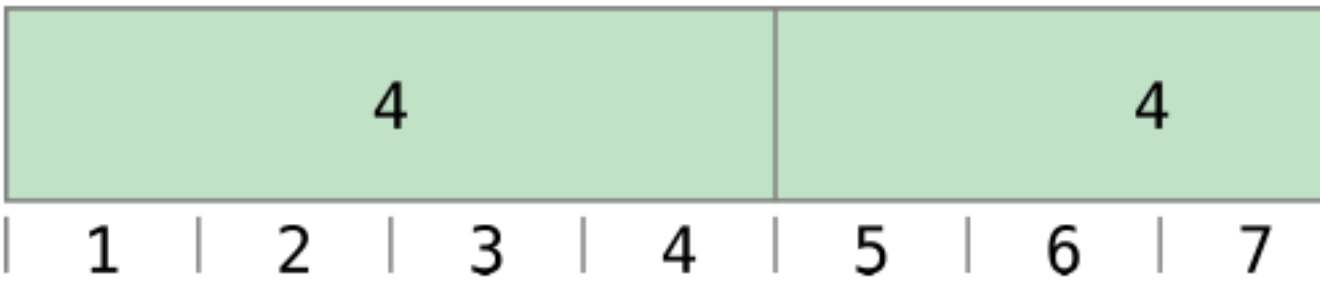
详见Concatenating Strings and Characters

1.3.1 取余运算符

取余运算符 (a % b) 计算出 a 是 b 的几倍然后返回被留下的值 (余数)。

> 注：余数运算符 (%) 亦称是其他语言的一个取模运算符。然而，其在 Swift 里意味着如果对负数操作，严格上讲，得到的是余数而不是模数。

这是余数运算符如何工作。要计算 9% 4，你首先得求出 9 是 4 的几倍：



9 能去除两个 4，并且余数是 1 (显示在橙色)。

在 Swift 中，这个将被写成：

```
9 % 4 // equals
```

确定 a % b 的答案，运算符% 计算下列等式并且返回余数作为其输出：

```
a = (b × some multiplier) + remainder
```

some multiplier 是 a 里面能包含 b 的最多倍数。

将 9 和 4 插入到公式：

```
9 = (4 × 2) + 1
```

同一个方法是应用的，当计算 a 时的一个负值的余数：

```
-9 % 4 // equals -1
```

将 -9 和 4 插入到公式：

```
-9 = (4 × -2) + -1
```

产生余数值为 -1。

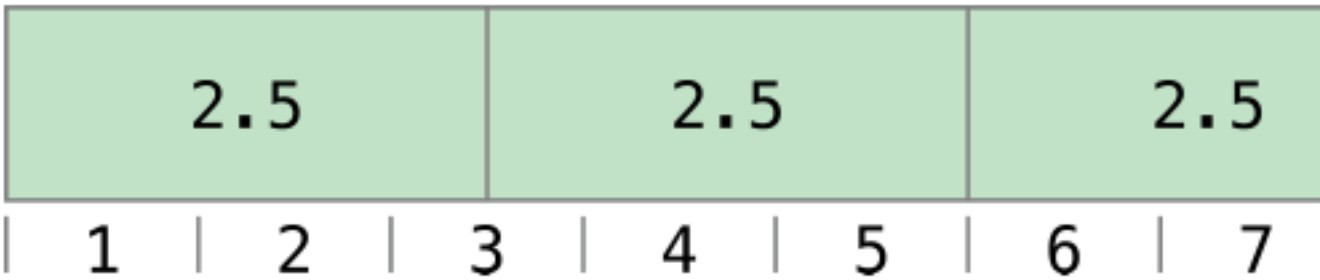
b 为负值时的 b 的符号被忽略，这意味着%b 和%-b 的结果是一样的。、

1.3.2 浮点余数计算

不同于 C 和 Objective-C，Swift 的余数运算符也能运用于浮点数：

```
8 % 2.5 // equals 0.5
```

在本例中，8 用 2.5 来分等于 3，余数是 0.5，因此余数为 0.5。



1.4 自增和自减运算符

像 C 一样,Swift 提供一个自增运算符 (++) 和自减运算符 (–) 作为增加或减少一个数值的一种快捷方式，增减量为 1。您能对任何整数或浮点类型的变量使用这些运算符。

```
var i = 0
++i // i now equals 1
```

每当你使用 ++i ，i 的值增加 1，本质上 ++i 可以看做是 i=i+1，同样 –i 可以看做是 i=i-1。

++ 和 –符号可以使用作为前缀算符或作为后缀运算符。++i 和 i++ 是两个有效的方式给 i 的值增加 1，同样, –i 和 i–如是。

注意这些运算符修改 i 并且返回值。如果你只想要增加或减值 i，您可以忽略返回值。然而，如果你使用返回值，根据下列规则将是不同的根据的您是否使用了运算符的前缀或后缀版本，它:

- 如果运算符在变量之前被写，它在返回其值之前增加变量。
- 如果运算符在变量之后被写，它在返回其值之后增加变量。

例如:

```
var a = 0
let b = ++a
// a and b are now both equal to 1
let c = a++
// a is now equal to 2, but c has been set to the pre-increment value of 1
```

在上面的例子中，let b = ++a 中 a 在返回其值之前增加，这就是为什么 a 和 b 的新值是等于 1。

然而，let c = a++ 中 a 在返回其值之后增加，这意味着 c 获得 a 的原值 1，然后 a 自增，a 等于 2。

除非你需要特定工作情况下才使用 i++，否则在所有的情况下建议你使用 ++i 和 –i， 因为他们修改 i 并返回值的行为符合我们的预期。

1.4.1 一元减运算符

一个数值前加了符号 -，叫作一元减运算符:

```
let three = 3
let minusThree = -three // minusThree equals -3
let plusThree = -minusThree // plusThree equals 3, or "minus minus three"
```

一元减运算符 (-) 直接地被加在前面，在它起作用的值之前，不用任何空白空间。

1.4.2 一元加运算符

一元加运算符 (+) 返回它起作用的值，不做任何变动：

```
let minusSix = -6
let alsoMinusSix = +minusSix // alsoMinusSix equals -6
```

虽然一元加上运算符实际上不执行什么，当你也使用一元减负数的运算符时，你能使用它提供对称的正数。

1.5 复合赋值操作符

Swift 提供类似C 语言的复合赋值操作符，即把赋值和另一个运算合并起来。举个例子，像加法赋值运算符 (+ =)：

```
var a = 1
a += 2
// a is now equal to 3
```

表达式 a += 2 比 a = a + 2 更精炼。加法赋值运算符能够有效地把加法和赋值组合到一个运算, 同时执行这两个任务。

> 要注意的是，复合赋值操作符不返回值。例如，你不能写让成 let b = + = 2, 这种行为不同于上面提到的递增和递减运算符。

复合赋值运算符的完整列表可以在 [Expressions] 那一章节找到

1.6 比较运算符

Swift 支持所有标准 c 的比较运算符

- 等于 (a == b)
- 不等于 (a != b)
- 大于 (a > b)
- 小于 (a < b)
- 大于等于 (a >= b)
- 小于等于 (a <= b)

注：Swift 提供两个恒等运算符 (=== and !==)，用它来测试两个对象引用是否来自于同一个对象实例。详见Classes and Structures。每个比较操作符返回一个 Bool 值来表示语句是否为真：

```
1 == 1 // true, because 1 is equal to 1
2 != 1 // true, because 2 is not equal to 1
2 > 1 // true, because 2 is greater than 1
1 < 2 // true, because 1 is less than 2
1 >= 1 // true, because 1 is greater than or equal to 1
2 <= 1 // false, because 2 is not less than or equal to 1
```

比较操作符通常用在条件语句，如 if 语句：

```
let name = "world"
if name == "world" {
    println("hello, world")
} else {
    println("I'm sorry \(name), but I don't recognize you")
}
// prints "hello, world", because name is indeed equal to "world"
```

想要了解更多有关的 if 语句，请参阅控制流。

1.7 三元条件运算符

三元条件运算符是一种特殊的运算符，有三个部分，其形式为 question? answer1：answer2. 这是一个用来测试两种表达式基于输入是真或是假的快捷方式。如果 question? 为真时, 它评估 answer1 并返回其值; 否则, 它评估 answer2 并返回其值。三元条件运算符是下面的代码的简化：

```
if question {
    answer1
} else {
    answer2
}
```

这里举一个列子，计算一个表行像素的高度，如果行有一个头，行高应该是 50 像素，比内容要高度要高。如果行没有头是 20 像素：

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

前面的例子也可以用下面的的代码:

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// rowHeight is equal to 90
```

第一个例子使用的三元条件运算符，意味着 rowHeight 可以在一行代码被设置为正确的值。这比第二个示例更简洁， 不需要课外的 rowHeight 变量， 因为它的价值不需要在一个 if 语句中修改。

三元条件运算符提供了一个高效的写法来决定哪个表达式会被执行。不过还是请小心使用三元条件运算符，其简洁性如果过度使用会导致阅读代码的困难。要避免多个实例的三元条件运算符组合成一个复合语句。

范围运算符

Swift 包含两个范围运算符，能快捷的表达一系列的值

1.7.1 封闭范围运算符

封闭范围运算符 (a...b) 定义了一个范围, 从 a 到 b, 并包括 a 和 b 的值。

当要在一个范围内迭代所有可能的值的时候，范围运算符是非常有用的, 例如 for-in 循环

```
for index in 1...5 {
println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

欲了解更多 for-in 循环，请参阅控制流。

1.7.2 半封闭的区域运算符

半封闭的区域运算符 (a..b) 定义了从 a 到 b 的范围，但不包括 b。它被认为是半封闭的，因为它包含第一个值，而不包含最终值。

半封闭的范围使用明确，当你使用从零开始的列表，如数组，它是有用的数到 (但不包括) 列表的长度:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..count {
    println("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

请注意, 该数组包含四个项目, 但 0 . . 数只数到 3 (数组中的最后一个项目的索引)，因为它是一个半封闭的范围。欲了解更多有关数组的信息，请参阅数组

1.8 逻辑运算符

逻辑运算符修改或结合布尔逻辑值 true 和 false。Swift 支持这三个标准逻辑运算符基于 c 语言:

- Logical NOT (!a)
- Logical AND (a && b)
- Logical OR (a || b)

1.8.1 逻辑非运算符

逻辑非运算符 (!a) 转化一个 Bollean 值，true 变成 false，false 变成 true。

逻辑操作符是一个前缀操作符， 并立即出现在它修饰的值之前，没有任何空白，它被解读为" 不是"， 见下面的例子:

```
let allowedEntry = false
if !allowedEntry {
    println("ACCESS DENIED")
}
// prints "ACCESS DENIED"
```

这句话 if !allowedEntry 能理解为 "if not allowedEntry." 只执行后续的行，如果 "not allowedEntry" 是 true; 那就是说 if allowedEntry 是 false.

在这个例子中，精心挑选的布尔常量和变量名可以帮助保持代码的可读性和简洁，同时避免双重否定或混乱的逻辑语句。

1.8.2 逻辑与运算符

逻辑与运算符：(A && B) 创建的表达式中，A 和 B 两个值必须同时为 true 时表达式才正确。

其中 A 或者 B 有任一值是 false 时，逻辑与算符表示不成立，必须两者同时为 true 时才成立。事实上，如果第一个值是 false，第二个值甚至不会再进行判断，因为必须是两个值皆为 true，已经有一方 false、则没必要再往下面进行判断了。这被称作短路条件。

以下这个例子判断两个 Bool 类型的值，并只有这两个值都为真的时候会输出：Welcome。失败则输出 "ACCESS DENIED"：

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "ACCESS DENIED ”
```

1.8.3 逻辑或运算符

表达式（a || b）运算符中、只要 a 或者 b 有一个为 true，表达式就成立。

与上面的逻辑与运算符相似，逻辑或运算符使用短路条件判断，如果左边是 true，那么右边不会被判断，因为整体结果不会改变了。

在下面的例子中，第一个布尔值（hasDoorKey）为 false，但第二个值 (knowsOverridePassword）为 true。因为两者有一个值是 true，整个表达式的计算结果也为 true，正确输出：Welcome!

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

1.8.4 复合逻辑表达式

你可以将多个逻辑运算符复合来创建更长的复合表达式：

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

相比于之前两个单独分开的运算符，本次通过多重嵌套、将我们上面的 &&、|| 运算符相结合组合成一个较长的复合表达式。看起来有点饶人、其实本质还是两两相比较、可以简单地看成 A && B || C || D、从左往右根据运算符优先级进行判断、注意区分开 &&、||、只要牢记运算逻辑 && 需要两者都为 true、|| 则只需要一方为 true 则运算符正确即可解析整个复合表达式、透过现象看本质。

1.8.5 明确地括号 (翻译成中文语句不连贯太特么饶人了、怒了自己理解。)

复合表达式中，我们可以添加进 () 使确逻辑意图更加明确，上面的例子中，我们可以在第一部分上加括号来使意义更明确。

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

在复合逻辑表达式中、我们可以使用括号明确地表示我们需要将几个值放在一个单独的逻辑运算中去判断得出结果、最后根据 () 内的结果再去与后面的值进行判断、看上面的例子、就像我们小学学加减乘除一样、如果没有括号 () 我们肯定是按照运算符的优先级去判断、但此时有了括号、我们需要先运算其中的逻辑运算符得到它们的值. 使用括号 () 在符合逻辑表达式中可以更明确的你的意图。

参考文献