翻译：xielingwang　校对：Evilcome

基本运算符

本页包含内容：

运算符是检查，改变，合并值的特殊符号或短语。例如，加号`+`将两个数相加（如`let i = 1 + 2`）。复杂些的运行算例如逻辑与运算符`&&`（如`if enteredDoorCode && passedRetinaScan`），又或让 i 值加1的便捷运算符自增运算符`++i`等。

Swift 支持大部分标准 C 语言的运算符，且改进许多特性来减少常规编码错误。如，赋值符（`=`）不返回值，以防止把想要判断相等运算符（`==`）的地方写成赋值符导致的错误。数值运算符（`+`，`-`，`*`，`/`，`%`等）会检测并不允许值溢出，以此来避免保存变量时由于变量大于或小于其类型所能承载的范围时导致的异常结果。当然允许你使用 Swift 的溢出运算符来实现溢出。详情参见溢出运算符。

区别于 C 语言，在 Swift 中你可以对浮点数进行取余运算（`%`），Swift 还提供了 C 语言没有的表达两数之间的值的区间运算符，（`a..b`和`a...b`），这方便我们表达一个区间内的数值。

本章节只描述了 Swift 中的基本运算符，高级运算符包含了高级运算符，及如何自定义运算符，及如何进行自定义类型的运算符重载。

术语

运算符有一元，二元和三元运算符。

- 一元运算符对单一操作对象操作（如`-a`）。一元运算符分前置符和后置运算符，前置运算符需紧排操作对象之前（如`!b`），后置运算符需紧跟操作对象之后（如`i++`）。
- 二元运算符操作两个操作对象（如`2 + 3`），是中置的，因为它们出现在两个操作对象之间。
- 三元运算符操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三元条件运算符（`a ? b : c`）。

受运算符影响的值叫操作数，在表达式`1 + 2`中，加号`+`是二元运算符，它的两个操作数是值`1`和`2`。

赋值运算符

赋值运算（`a = b`），表示用`b`的值来初始化或更新`a`的值：

```
let b = 10
var a = 5
a = b
// a 现在等于 10
```

如果赋值的右边是一个多元组，它的元素可以马上被分解多个变量或变量：

```
let (x, y) = (1, 2)
// 现在 x 等于 1, y 等于 2
```

与 C 语言和 Objective-C 不同，Swift 的赋值操作并不返回任何值。所以以下代码是错误的：

```
if x = y {
    // 此句错误，因为 x = y 并不返回任何值
}
```

这个特性使得你无法把（==）错写成（=）了，由于if x = y是错误代码，Swift 从底层帮你避免了这些代码错误。

数值运算

Swift 让所有数值类型都支持了基本的四则运算：

- 加法（+）
- 减法（-）
- 乘法（*）
- 除法（/）

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

与 C 语言和 Objective-C 不同的是，Swift 默认不允许在数值运算中出现溢出情况。但你可以使用 Swift 的溢出运算符来达到你有目的的溢出（如a &+ b）。详情参见溢出运算符。

加法运算符也用于String的拼接：

```
"hello, " + "world"  // 等于 "hello, world"
```

两个Character值或一个String和一个Character值，相加会生成一个新的String值：

```
let dog: Character = "d"
let cow: Character = "c"
let dogCow = dog + cow
// 译者注：原来的引号内是很可爱的小狗和小牛，但win os下不支持表情字符，所以改成了普通字符
// dogCow 现在是 "dc"
```

详情参见字符，字符串的拼接。

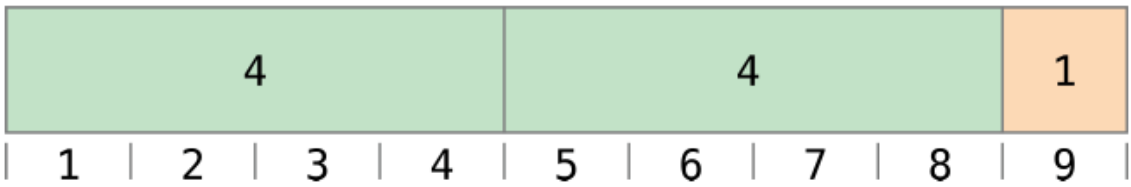
求余运算

求余运算（`a % b`）是计算`b`的多少倍刚刚好可以容入`a`，返回多出来的那部分（余数）。

注意：

求余运算（`%`）在其他语言也叫取模运算。然而严格说来，我们看该运算符对负数的操作结果，"求余"比"取模"更合适些。

我们来谈谈取余是怎么回事，计算`9 % 4`，你先计算出`4`的多少倍会刚好可以容入`9`中：



2倍，非常好，那余数是1（用橙色标出）

在 Swift 中这么来表达：

```
9 % 4    // 等于 1
```

为了得到`a % b`的结果，`%`计算了以下等式，并输出余数作为结果：

$a = (b \times 倍数) + 余数$

当倍数取最大值的时候，就会刚好可以容入`a`中。

把`9`和`4`代入等式中，我们得`1`：

```
9 = (4 × 2) + 1
```

同样的方法，我来们计算 `-9 % 4`：

```
-9 % 4   // 等于 -1
```

把`-9`和`4`代入等式，`-2`是取到的最大整数：

```
-9 = (4 × -2) + -1
```

余数是`-1`。

在对负数`b`求余时，`b`的符号会被忽略。这意味着 `a % b` 和 `a % -b`的结果

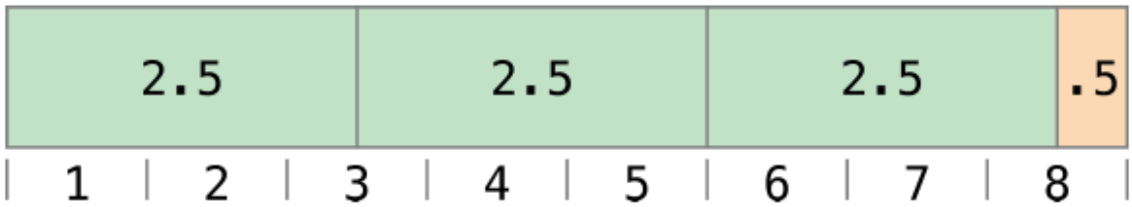
是相同的。

浮点数求余计算

不同于 C 语言和 Objective-C，Swift 中是可以对浮点数进行求余的。

```
8 % 2.5 // 等于 0.5
```

这个例子中，8除于2.5等于3余0.5，所以结果是一个Double值0.5。



自增和自增运算

和 C 语言一样，Swift 也提供了方便对变量本身加1或减1的自增（++）和自减（--）的运算符。其操作对象可以是整形和浮点型。

```
var i = 0
++i       // 现在 i = 1
```

每调用一次++i，i的值就会加1。实际上，++i是i = i + 1的简写，而--i是i = i - 1的简写。

++和--既是前置又是后置运算。++i，i++，--i和i--都是有效的写法。

我们需要注意的是这些运算符修改了i后有一个返回值。如果你只想修改i的值，那你就可以忽略这个返回值。但如果你想使用返回值，你就需要留意前置和后置操作的返回值是不同的。

- 当++前置的时候，先自增再返回。
- 当++后置的时候，先返回再自增。

例如：

```
var a = 0
let b = ++a // a 和 b 现在都是 1
let c = a++ // a 现在 2，但 c 是 a 自增前的值 1
```

上述例子，let b = ++a先把a加1了再返回a的值。所以a和b都是新值

1。

而let c = a++，是先返回了a的值，然后a才加1。所以c得到了a的旧值1，而a加1后变成2。

除非你需要使用i++的特性，不然推荐你使用++i和--i，因为先修改后返回这样的行为更符合我们的逻辑。

一元负号

数值的正负号可以使用前缀-（即一元负号）来切换：

```
let three = 3
let minusThree = -three       // minusThree 等于 -3
let plusThree = -minusThree   // plusThree 等于 3, 或
"负负3"
```

一元负号（-）写在操作数之前，中间没有空格。

一元正号

一元正号（+）不做任何改变地返回操作数的值。

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

虽然一元+做无用功，但当你在使用一元负号来表达负数时，你可以使用一元正号来表达正数，如此你的代码会具有对称美。

复合赋值（Compound Assignment Operators）

如同强大的 C 语言，Swift 也提供把其他运算符和赋值运算（=）组合的复合赋值运算符，加赋运算（+=）是其中一个例子：

```
var a = 1
a += 2 // a 现在是 3
```

表达式`a += 2`是`a = a + 2`的简写，一个加赋运算就把加法和赋值两件事完成了。

注意:

复合赋值运算没有返回值，`let b = a += 2`这类代码是错误。这不同于上面提到的自增和自减运算符。
在表达式章节里有复合运算符的完整列表。

比较运算

所有标准 C 语言中的比较运算都可以在 Swift 中使用。

- 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`）

注意:

Swift 也提供恒等`===`和不恒等`!==`这两个比较符来判断两个对象是否引用同一个对象实例。更多细节在类与结构。
每个比较运算都返回了一个标识表达式是否成立的布尔值:

```
1 == 1   // true, 因为 1 等于 1
2 != 1   // true, 因为 2 不等于 1
2 > 1    // true, 因为 2 大于 1
1 < 2    // true, 因为 1 小于2
1 >= 1   // true, 因为 1 大于等于 1
2 <= 1   // false, 因为 2 并不小于等于 1
```

比较运算多用于条件语句，如`if`条件:

```
let name = "world"
if name == "world" {
    println("hello, world")
} else {
```

```
    println("I'm sorry \(name), but I don't recognize you")
}
// 输出 "hello, world", 因为 `name` 就是等于 "world"
```

关于if语句，请看控制流。

三元条件运算(Ternary Conditional Operator)

三元条件运算的特殊在于它是有三个操作数的运算符，它的原型是问题? 答案1: 答案2。它简洁地表达根据问题成立与否作出二选一的操作。如果问题成立，返回答案1的结果; 如果不成立，返回答案2的结果。

使用三元条件运算简化了以下代码:

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

这里有个计算表格行高的例子。如果有表头，那行高应比内容高度要高出50像素; 如果没有表头，只需高出20像素。

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight 现在是 90
```

这样写会比下面的代码简洁:

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
```

```
} else {
    rowHeight = rowHeight + 20
}
// rowHeight 现在是 90
```

第一段代码例子使用了三元条件运算，所以一行代码就能让我们得到正确答案。这比第二段代码简洁得多，无需将`rowHeight`定义成变量，因为它的值无需在`if`语句中改变。

三元条件运算提供有效率且便捷的方式来表达二选一的选择。需要注意的事，过度使用三元条件运算就会由简洁的代码变成难懂的代码。我们应避免在一个组合语句使用多个三元条件运算符。

区间运算符

Swift 提供了两个方便表达一个区间的值的运算符。

闭区间运算符

闭区间运算符（`a...b`）定义一个包含从`a`到`b`(包括`a`和`b`)的所有值的区间。 闭区间运算符在迭代一个区间的所有值时是非常有用的，如在`for-in`循环中：

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

关于`for-in`，请看控制流。

半闭区间

半闭区间（`a..b`）定义一个从`a`到`b`但不包括`b`的区间。 之所以称为半闭区

间，是因为该区间包含第一个值而不包括最后的值。

半闭区间的实用性在于当你使用一个0始的列表(如数组)时，非常方便地从0数到列表的长度。

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

数组有4个元素，但`0..count`只数到3(最后一个元素的下标)，因为它是半闭区间。关于数组，请查阅数组。

逻辑运算

逻辑运算的操作对象是逻辑布尔值。Swift 支持基于 C 语言的三个标准逻辑运算。

- 逻辑非（`!a`）
- 逻辑与（`a && b`）
- 逻辑或（`a || b`）

逻辑非

逻辑非运算（`!a`）对一个布尔值取反，使得`true`变`false`，`false`变`true`。

它是一个前置运算符，需出现在操作数之前，且不加空格。读作`非 a`，然后我们看以下例子：

```
let allowedEntry = false
if !allowedEntry {
```

```
    println("ACCESS DENIED")
}
// 输出 "ACCESS DENIED"
```

`if! allowedEntry`语句可以读作 "如果 非 alowed entry。"，接下一行代码只有在如果 "非 allow entry" 为`true`，即`allowEntry`为`false`时被执行。

在示例代码中，小心地选择布尔常量或变量有助于代码的可读性，并且避免使用双重逻辑非运算，或混乱的逻辑语句。

逻辑与

逻辑与（`a && b`）表达了只有`a`和`b`的值都为`true`时，整个表达式的值才会是`true`。

只要任意一个值为`false`，整个表达式的值就为`false`。事实上，如果第一个值为`false`，那么是不去计算第二个值的，因为它已经不可能影响整个表达式的结果了。这被称做 "短路计算（short-circuit evaluation）"。

以下例子，只有两个`Bool`值都为`true`值的时候才允许进入：

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

逻辑或

逻辑或（`a || b`）是一个由两个连续的`|`组成的中置运算符。它表示了两个逻辑表达式的其中一个为`true`，整个表达式就为`true`。

同逻辑与运算类似，逻辑或也是"短路计算"的，当左端的表达式为`true`时，将不计算右边的表达式了，因为它不可能改变整个表达式的值了。

以下示例代码中，第一个布尔值（`hasDoorKey`）为`false`，但第二个值

（knowsOverridePassword）为true，所以整个表达是true，于是允许进入：

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

组合逻辑

我们可以组合多个逻辑运算来表达一个复合逻辑：

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

这个例子使用了含多个&&和||的复合逻辑。但无论怎样，&&和||始终只能操作两个值。所以这实际是三个简单逻辑连续操作的结果。我们来解读一下：

如果我们输入了正确的密码并通过了视网膜扫描；或者我们有一把有效的钥匙；又或者我们知道紧急情况下重置的密码，我们就能把门打开进入。

前两种情况，我们都不满足，所以前两个简单逻辑的结果是false，但是我们是知道紧急情况下重置的密码的，所以整个复杂表达式的值还是true。

使用括号来明确优先级

为了一个复杂表达式更容易读懂，在合适的地方使用括号来明确优先级是很有效的，虽然它并非必要的。在上个关于门的权限的例子中，我们给第

一个部分加个括号，使用它看起来逻辑更明确：

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

这括号使得前两个值被看成整个逻辑表达中独立的一个部分。虽然有括号和没括号的输出结果是一样的，但对于读代码的人来说有括号的代码更清晰。可读性比简洁性更重要，请在可以让你代码变清晰地地方加个括号吧！