**基本操作符**

操作符是用来查，改和合并数值的特殊符号或短语。例如，加号（+）将两个数字加到一起（就像 let i =1+2）。更为复杂的例子包括逻辑AND操作符&&（就像 if enteredDoorCode && passedRetinaScan）和递增操作符 ++i， ++i是将i增加1的快捷方法。

Swift支持绝大多数的标准C操作符并且提升了诸多特性并去除了一些常见的编码错误。赋值运算符（=）不返回数值，这样就避免了错误的将之误用于等于（==）操作符情况的发生。运算操作符（+, -, \*, /, % 等等）检测并且不允许数值溢出，这样就避免了使用超出类型允许存储范围外的数子。你也可以使用Swift的溢出操作符来控制溢出的行为。这会再溢出操作符的章节中讲到。

不同于C，Swift允许你对浮点数字使用余（%）计算。Swift也提供了两种范围操作符（a..b and a...b）这是C中没有的，是表达数值范围的快捷方式。

这章描述了Swift中一些常见的操作符，高级操作符章节会讲到Swift中得一些高级操作符，并且描述了如何自定义操作符以及如何将标准操作符用于实现你自定义的类型。

**术语**

运算符是一元，二元或三元的。

一元操作符作用于单一对象（比如-a）一元前缀操作符出现在对象前（比如!b）,而一元后缀操作符则出现在对象之后（比如i++）

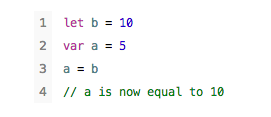
二元操作符作用于两个对象（比如2+3）,它以中缀的形式出现在对象之间。

类似于C，三元操作符是作用于三个对象上得，Swift只有一个三元条件操作符 （a?b:c）

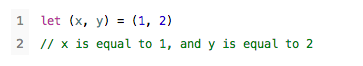
操作符作用的对象被称之为操作数，在表达式1+2中，+号是二元操作符，它的操作数则是1和2.

**赋值操作符**

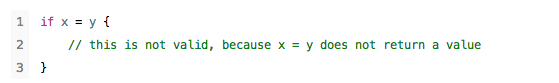
赋值操作符（a=b）将b的值用于初始化或者更新a的值



如果赋值右边是一个元组，那么它的元素可以被立即分解为多个常量或者变量。



与C和Objective-C不同的是，赋值操作符在Swift中不会返回数值，因此下述语句是无效的：



该特性避免了赋值运算符（=）和等于运算符（==）间的混用，通过使得if x=y无效，这样Swift就帮助你在编码中避免了这样类型的错误。

**数学运算符**

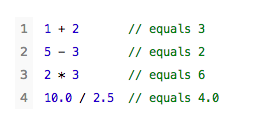
对于所有的数字类型，Swift支持四种数学运算符：

加法（+）

减法（-）

乘法（\*）

除法（/）

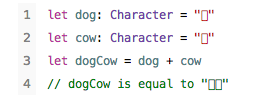


不同于C和Objective-C, Swift 数学运算符默认不允许数值溢出，你可以通过使用溢出操作符来控制溢出行为（比如a &+b）见溢出操作符。

同样的加法运算符也支持String字符串的合并运算。



两个Character值或者一个Character值与一个String值 可以相加形成一个新的String



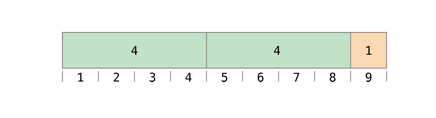
**余操作符**

模操作符（a%b）用于计算有多少b的倍数可以包含在a之内，然后返回剩余的数值（也称之为余）

注意

在其他语言中，余操作符也称之为模操作符，但是对于负数而言，在swift中，严格说，是余而非模。

余操作符是这样工作的，计算9%4，你首先需要先计算出9中可以放下多少个4，如图，你可以放下两个4，所以剩余的值为1，也就是余。



在swift里，你可以这么书写



为了得出a%b的计算结果，余操作符执行下面的公式并且将remainder作为输出返回



这里的 some multipler是a中最多能放下多少b的数量。

将9和4插入公式，我们得到了：

9=（4x2）+1

同样的方法也适用于计算负a的余



在公式中插入 -9 和4 我们得到了：

-9 = （4x-2）+-1

余也就是-1

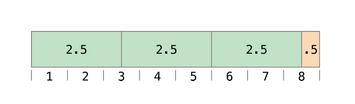
在求余中，b的符号忽略，也就是说a%b 和 a%-b结果相同。

**浮点求余计算**

和C以及Objective-C不同，Swift也支持浮点数求余：



在这个例子中， 8除以2.5等于3，余0.5，因此求余返回0.5的Double值



**递增和递减操作符**

和C相同，Swift提供了递增（++）和递减（--）操作符用于快速将数值增加1或者减少1，你可以将之使用在任意的整数或者浮点数上。



每一次你调用++i的时候，i的值增加1，本质上讲，++i 和 i=i+1是等价的，同样的，--i等价为i=i-1

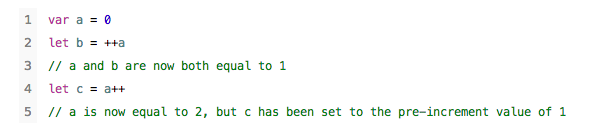
++ 和 – 操作符可以被用于前缀或者后缀操作符， ++i和i++都是使i增加1的有效方法，同样的 --i和i--也是使i减少1的有效方法。

需要注意的是，这些操作符修改i的同时也回返回数值，如果你只是增加或者减少i中存储的数值，那么你可以忽略返回值。但是如果你需要使用到返回值，那么根据如下规则，前缀操作符和后缀操作符将有所不同：

如果操作符在变量之前书写，那么它先增加变量，再返回数值。

如果操作符在变量之后书写，那么它先返回数值，再增加变量。

例如：



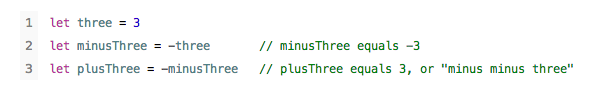
在上面的例子中，let b = ++a 先增加a然后再返回数值，这也是为什么，a和b都等于最新的1

但是， let c = a++ 先返回值后再增加a， 这也意味着， c获得的是旧的1值，而a则更新到值2.

除非你有特定的理由去使用i++那么推荐一直使用 ++i 和 --i 因为修改i然后返回结果是最常用的。

**一元减操作符**

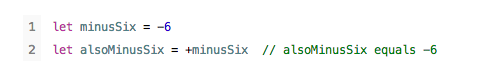
一元减操作符用于切换数值的符号的正负：



一元减操作符直接放在操作数值的前面，没有任何空格。

一元加操作符

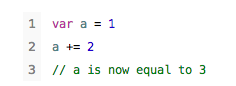
一元加操作符直接返回操作的数值，对其不做任何操作。



尽管一元加操作符不起任何作用，但是你可以使用它实现代码的对称性，特别是当你使用一元减操作符的时候。

**复合赋值操作符**

和C一样， Swift 提供了复合赋值操作符将赋值操作符和其他操作合并在一起，下面是一个相加赋值操作符（+=）的例子：



该表达式a+=2是a=a+2的快捷写法，它非常高效的将赋值和相加的操作合并在一个操作中。

注意：

复合赋值操作符不返回值。 你不能这么书写 let b = a+=2, 这和之前提到的递增和递减操作符是有区别的。

复合操作符的完整列表可以在表达式章节中找到。

**比较操作符**

Swift支持所有C中的标准比较操作符

等于（a==b）

不等于（a!=b）

大于（a>b）

小于（a<b）

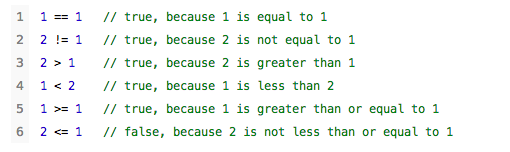
大于等于（a>=b）

小于等于（a<=b）

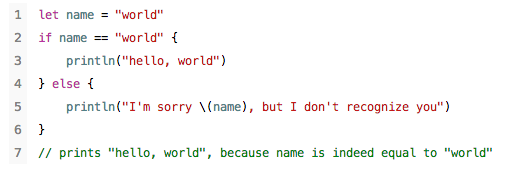
注意：

Swift也提供了身份操作符（===和!==）这用于测试两个对象引用是否引用的时同一对象实例。 可以查看 类和结构章节获取更多的知识

每个比较语句会返回一个Bool值用于指明该语句是真还是假：



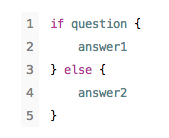
比较操作符也经常在条件语句中使用，比如if



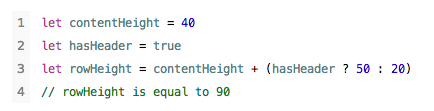
**三元条件操作符**

三元条件操作符是一个特殊的具有三部分的操作符，有着这样的格式 question? answer1 : answer2 如果question为真，那么就执行answer1部分，如果question为假，那么就执行answer2.

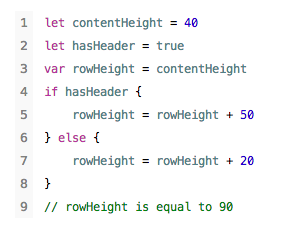
三元条件操作符和下面代码快捷书写方法：



这里有个例子，是计算表行的像素高度的。 如果行有头部分，那么行的高度应该比内容高出50像素，如果没有头则只应该高出20像素：



前面这个例子是下面代码的简写：



第一个例子使用三元条件操作符使得rowHeight的值在一行就能正确的被指定。这比第二个例子更为简明，并且也省去了创建rowHeight变量的需要，因为它的例子不需要在if语句中被修改。

三元条件操作符提供了一个简明的方法来判断两个表达式应该使用哪一个。当然也请谨慎使用它，过多使用三元条件操作符会降低代码可阅读性。请避免将多个三元条件符合并在一个语句中的情况。

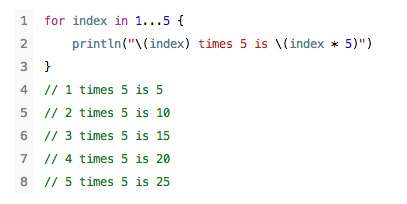
**范围操作符**

Swift提供了两个范围操作符，用于简单表明数值范围。

闭合范围操作符

闭合范围操作符（a…b）定义了a到b的范围，并且包含a和b (以a开始以b结尾)

闭合范围操作符在对诸如 for-in这样对范围中每一个值都“遍历”的循环特别有用：

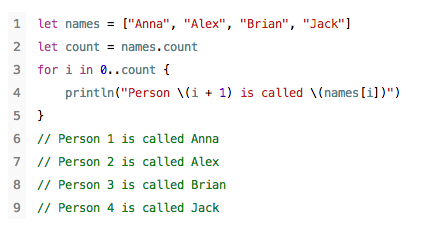


**半闭合范围操作符**

半闭合操作符（a..b）定义了a到b的范围 (以a开始以b结尾)

，但是不包含b，之所以称之为半闭合式因为该范围包含了a但是不包含b。

半闭合范围操作符适用于基于零的列举，例如array 计数到array得长度但是不包含该长度。



需要注意的时， 如果array包含4个元素，但是0..count仅数到3（也即是array中最后一个元素的索引值），因为这是一个半闭合的范围。 关于Arrays 可以查看Arrays章节

**逻辑操作符**

逻辑操作符用于修改和合并布尔逻辑值true和false。Swift支持C中三个标准逻辑操作符：

逻辑非 （!a）

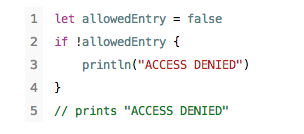
逻辑和 （a&&b）

逻辑或 （a||b）

**逻辑非操作符**

逻辑非操作符（!a）反转布尔值使得true变为false，false变为true.

逻辑非操作符是前缀操作符，直接出现在操作的对象前，没有空格，可以这么阅读“非a”



短语 if !allowedEntry 可以这么阅读“如果不被允许进入”也就是说接下来的代码仅会在不被允许进入为真，也即是allowedEntry为假时执行。

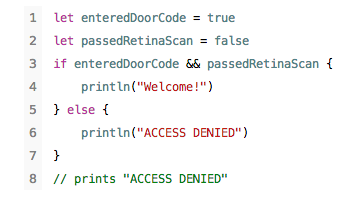
从该例子我们可以看到，恰当选择布尔常量和变量名字有助于代码的可读性和简明性，并且避免了难理解的双重否定和逻辑。

**逻辑和操作符**

逻辑操作符（a&&b）是这样工作的：只有a和b都为真得时候，该表达式才为真。

如果其中任何一个为false，整个表达式则为false。事实上如果第一个值为假，那么第二个值是不会被执行的，因为整个表达式已经不可能再为true。这也被称做“短路计算”

该例子考虑了两个布尔值，仅在两个值都未真得时候才允许访问

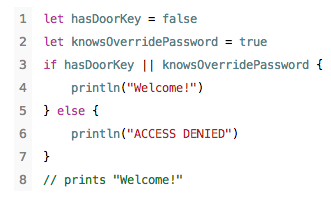


**逻辑或操作符**

逻辑或操作符是一个中缀操作符，只要其中任何一个数值为真，那么整个表达式则为真。

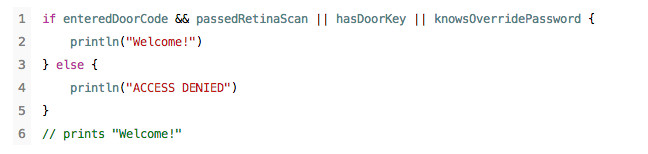
和逻辑和操作符类似，逻辑或操作符使用“短路计算”来执行其表达式，如果左边的表达式为真，右边的表达式不被计算，因为这不会改变表达式最终的结果。

在下面的例子中，第一个Bool值（hasDoorKey）为假，但是第二个值（knowsOverridePassword）为true. 因为其中一个值为真，整个表达式也为真。访问被允许：



**合并逻辑操作符**

你可以合并逻辑操作符来创建较长的复合表达式：



该例子使用了多个&&和||来创建了一个较长的复合表达式，但是&&和||表达式仅操作两个值，因此该表达式其实是三个较小的表达式链接到了一起。

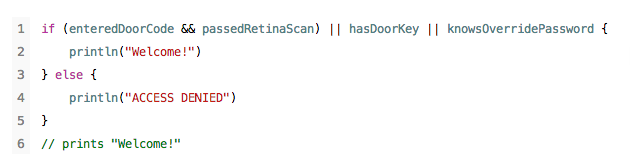
可以这么阅读：

如果我已经输入了正确的房门密码并且通过虹膜扫描； 或者我又一个有效的钥匙或者我拥有紧急复写密码，那么访问被允许。

根据 enteredDoorCode, passedRetinaScan 和 hasDoorKey 的值，前两个小表达式值为false, 但是， 紧急复写密码是知道的，这样整个复合表达式仍然为true.

**明确括号**

有时候故意在代码中添加括号，这样做得目的是使得复杂的表达式变得易读，在上面的例子中，我们可以给表达式添加上括号使得整个表达式更为简明易懂：



明确括号使得前面两个值被看做一体作为整个逻辑的一个可能状态。整个复合表达式的输出结果不变，但是整个表达式的目的性更明确了同时也更加易读。易读性永远是优先于简短的，如果明确括号能使得你得代码更加清晰和易读，那么请使用它。