基本操作员

一个*运营商*是一个特殊的符号，或者你使用来检查，更改或合并值的短语。例如，加法运算符（+）添加两个数字，如in `let i = 1 + 2`，逻辑AND运算符（&&）组合两个布尔值，如in `if enteredDoorCode && passedRetinaScan`。

Swift支持大多数标准C运算符，并改进了几种消除常见编码错误的功能。赋值运算符（=）不会返回值，以防止在等于运算符（==）时错误地使用该值。算术运算符（+，-，*，/，%等等）检测和禁止值溢出，与变得比存储它们的类型的所允许的值的范围更大或更小数目的工作时避免意外的结果。您可以通过使用雨燕的溢出运营商选择在价值溢出行为，如在溢出运营商。

Swift还提供了在C中没有的范围运算符，比如`a..<b`和`a...b`，作为表达一系列值的快捷方式。

本章介绍Swift中的常用操作符。高级操作符涵盖了Swift的高级操作符，并且描述了如何定义自己的自定义操作符并为您自己的自定义类型实现标准操作符。

术语

运算符是一元，二元或三元运算符：

- *一元*运算符在单个目标上运行（例如`-a`）。一元*前缀*运算符紧挨着它们的目标（例如`!b`）出现，而一元*后缀*运算符出现在它们的目标之后（例如`c!`）。
- *二元*操作符运行在两个目标上（如`2 + 3`），并且是中*缀*，因为它们出现在两个目标之间。
- *三元*运营商在三个目标上运营。像C一样，Swift只有一个三元运算符，即三元条件运算符（`a ? b : c`）。

操作符影响的值是*操作数*。在表达式中`1 + 2`，+符号是二元运算符，它的两个操作数是值1和2。

作业操作员

该*赋值运算符*（`a = b`）初始化或更新的价值a与价值b：

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

如果赋值的右边是一个具有多个值的元组，则其元素可以一次分解为多个常量或变量：

```
let (x, y) = (1, 2)
// x is equal to 1, and y is equal to 2
```

与C和Objective-C中的赋值运算符不同，Swift中的赋值运算符本身不返回值。以下声明无效：

```
if x = y {
    // This is not valid, because x = y does not return a value.
}
```

此功能可以防止赋值运算=符（==）在实际意图等于运算符（）时被意外使用。通过使`if x = y`无效，Swift可以帮助您避免代码中出现这些类型的错误。

算术运算符

Swift支持所有数字类型的四个标准*算术运算符*：

- 加法（+）
- 减法（-）
- 乘法（*）

- 司（/）

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

与C和Objective-C中的算术运算符不同，Swift算术运算符默认情况下不允许值溢出。你可以通过使用Swift的溢出操作符（比如a &+ b）来选择赋值溢出行为。请参阅溢出操作符。

String连接 操作还支持连接：

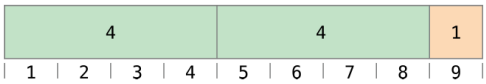
```
"hello, " + "world"  // equals "hello, world"
```

剩余操作员

该*余运算符*（a % b）的作品如何的好几倍b将适合的内部a，并返回剩下值（被称为*剩余部分*）。

> 注意
>
> 余数运算符（%）在其他语言中也称为*模运算符*。然而，它在Swift中用于负数的行为意味着严格来说，它是一种余数而不是模数运算。

以下是剩余操作符的工作方式。为了计算9 % 4，你首先要计算出有多少个4s可以放入里面9：



你可以放入两个4s 9，剩下的就是1（以橙色显示）。

在Swift中，这会写成：

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

为了确定答案a % b，%操作员计算下面的公式并remainder作为其输出返回：

a= （bx some multiplier） +remainder

里面some multiplier最大的倍数是 哪里？ ba

插入9和4这个公式得到：

9= （4x 2） +1

计算余数为负值时应用同样的方法a：

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

插入–9和4代入公式得到：

–9= （4x –2） +–1

给出一个余数值–1。

b负号的值将忽略 符号b。这意味着a % b并a % –b始终给出相同的答案。

一元减法运算符

数值的符号可以使用前缀–（称为*一元减运算符*）来切换：

```
let three = 3
```

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

一元减号运算符

一元加运算符

在*一元加运算*（+）只返回其所操作的价值，没有任何变化：

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

虽然一元加运算符实际上并没有做任何事情，但是在使用一元减运算符来表示负数时，也可以使用它来在代码中提供正数的对称性。

复合分配算子

和C一样，Swift提供了*复合赋值操作符*，它将assignment（=）与另一个操作结合在一起。一个例子是*附加赋值运算符*（+=）：

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

这个表达方式`a += 2`是速记的`a = a + 2`。实际上，添加和分配组合为一个运算符，可同时执行两个任务。

> 注意
>
> 复合赋值运算符不返回值。例如，你不能写`let b = a += 2`。

有关Swift标准库提供的运算符的信息，请参阅运算符声明。

比较运算符

Swift支持所有标准的C *比较运算符*：

- 等于（`a == b`）
- 不等于（`a != b`）
- 大于（`a > b`）
- 小于（`a < b`）
- 大于或等于（`a >= b`）
- 小于或等于（`a <= b`）

> 注意
>
> Swift还提供了两个*身份运算符*（===和!==），用于测试两个对象引用是否都引用同一个对象实例。有关更多信息，请参阅类和结构。

每个比较运算符都会返回一个`Bool`值来指示该语句是否为真：

```
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
```

比较运算符通常用于条件语句中，如if语句：

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

有关该if声明的更多信息，请参阅控制流程。

你可以比较两个元组，如果它们具有相同的类型和相同数量的值。元组从左到右进行比较，一次一个值，直到比较发现两个不相等的值。这两个值进行比较，比较的结果决定了元组比较的总体结果。如果所有的元素都相等，那么元组本身是相等的。例如：

```
(1, "zebra") < (2, "apple")   // true because 1 is less than 2; "zebra" and "apple" are not compared
(3, "apple") < (3, "bird")    // true because 3 is equal to 3, and "apple" is less than "bird"
(4, "dog") == (4, "dog")      // true because 4 is equal to 4, and "dog" is equal to "dog"
```

在上面的示例中，您可以在第一行看到从左到右的比较行为。因为1小于2，(1, "zebra")被认为小于(2, "apple")，不管元组中的任何其他值如何。没关系，"zebra"不小于"apple"，因为比较已经由元组的第一个元素决定了。然而，当元组的第一个元素是相同的，他们的第二个元素*进行*比较 - 这是发生在第二和第三行。

只有当操作符可以应用于各个元组中的每个值时，元组才可以与给定的操作符进行比较。例如，这表现在下面的代码，就可以比较类型的两个元组(String, Int)，因为这两个String和Int值可以使用进行比较<运算符。相比之下，两个元组类型(String, Bool)不能与<运算符进行比较，因为<运算符不能应用于Bool值。

```
("blue", -1) < ("purple", 1)        // OK, evaluates to true
("blue", false) < ("purple", true)  // Error because < can't compare Boolean values
```

> 注意
>
> Swift标准库包含元组少于7个元组的元组比较运算符。要比较具有七个或更多元素的元组，您必须自己实现比较运算符。

三元条件运算符

所述*三元条件算子*是由三个部分组成，这需要形式的特殊操作question ? answer1 : answer2。这是根据是否question为真来评估两个表达式之一的捷径。如果question属实，则评估answer1并返回其值; 否则，它会评估answer2并返回其值。

三元条件运算符是下面代码的简写：

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

以下是一个计算表格行高度的示例。如果该行有标题，则行高应比内容高度高50个点;如果该行没有标题，则行高应高20个点：

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

上面的例子是下面代码的简写：

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

第一个例子使用三元条件运算符意味着rowHeight可以在单行代码上设置正确的值，这比第二个例子中使用的代码更简洁。

三元条件运算符为决定要考虑哪两个表达式提供了有效的速记。但是，请谨慎使用三元条件运算符。如果过度使用，它的简洁性会导致难以阅读的代码。避免将三元条件运算符的多个实例组合成一个复合语句。

无合并操作员

的*零-合并运算符*（a ?? b）进行解包的可选a，如果它包含一个值，或者返回一个默认值b，如果a是nil。表达式a总是可选的类型。表达式b必须与存储在里面的类型匹配a。

nil-coalescing运算符是下面代码的简写：

```
a != nil ? a! : b
```

上面的代码使用三元条件操作和强制解包（a!）来访问内部包裹的值a时a是不nil，并返回b，否则。nil-coalescing运算符提供了一种更简洁的方式来以简明易懂的形式封装这个条件检查和解包。

> 注意
> 如果值为a非nil值，b则不评估值。这就是所谓*的短路评估*。

下面的示例使用nil-coalescing运算符在默认颜色名称和可选的用户定义颜色名称之间进行选择：

```
let defaultColorName = "red"
var userDefinedColorName: String?   // defaults to nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to the default of "red"
```

该userDefinedColorName变量被定义为可选的String，默认值为nil。由于userDefinedColorName是可选类型，因此您可以使用nil-coalescing运算符来考虑其值。在上面的示例中，运算符用于确定String调用变量的初始值colorNameToUse。因为userDefinedColorName是nil，表达式userDefinedColorName ?? defaultColorName返回值defaultColorName或"red"。

如果您将一个非nil值赋予userDefinedColorName并再次执行nil-coalescing操作符检查，则将userDefinedColorName使用内部包装的值而不是缺省值：

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is not nil, so colorNameToUse is set to "green"
```

范围运算符

Swift包含几个*范围运算符*，它们是表示一系列值的捷径。

关闭范围操作员

的*封闭范围操作符*（a...b）限定了从运行范围a来b，并且包括这些值a和b。值a不得大于b。

在遍历希望使用所有值的范围（例如使用`for-in`循环）时，闭范围运算符很有用：

```
for index in 1...5 {
    prj
}
// 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
```

有关`for-in`循环的更多信息，请参阅控制流程。

半开放范围运算符

所述*半开区间运算符*（`a..<b`）限定了从运行范围`a`到`b`，但不包括`b`。据说它是*半开放的*，因为它包含了它的第一个值，但不包含它的最终值。与封闭范围操作符一样，值`a`不得大于`b`。如果值`a`等于`b`，则结果范围将为空。

当您使用基于零的列表（例如数组）时，半开范围特别有用，可以计算（但不包括）列表长度：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("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..<count`只计算`3`（数组中最后一项的索引），因为它是半开范围。有关数组的更多信息，请参阅数组。

单面范围

闭范围运算符有一种替代形式，用于在一个方向上尽可能延续的范围 - 例如，范围包括从索引`2`到数组末尾的所有数组元素。在这些情况下，可以省略范围运算符一侧的值。这种范围被称为*单边范围*，因为操作员只有一方的价值。例如：

```
for name in names[2...] {
    print(name)
}
// Brian
// Jack

for name in names[...2] {
    print(name)
}
// Anna
// Alex
// Brian
```

半开范围的操作符也具有只写入其最终值的单面形式。就像在双方包含一个值时一样，最终的值不是该范围的一部分。例如：

```
for name in names[..<2] {
    print(name)
}
// Anna
// Alex
```

片面范围可以用在其他情况下，而不仅仅用于下标。您不能迭代忽略第一个值的单侧范围，因为不清楚应该从哪里开始迭代。您*可以*遍历一个忽略其最终值的单侧范围; 但是，由于范围无限期地继续，因此请确保为循环添加明确的结束条件。您还可以检查单侧范围是否包含特定值，如下面的代码所示。

```
let rar
range.contains(7)   // false
range.contains(4)   // true
range.contains(-1)  // true
```

逻辑运算符

*逻辑运算符*修改或组合布尔逻辑值`true`和`false`。Swift支持基于C语言中的三种标准逻辑运算符：

- 逻辑NOT（`!a`）
- 逻辑与（`a && b`）
- 逻辑或（`a || b`）

逻辑NOT运算符

的*逻辑非运算符*（`!a`）反转一个布尔值，使得`true`成为`false`，和`false`变`true`。

逻辑NOT运算符是一个前缀运算符，并且在其操作的值之前立即出现，没有任何空格。它可以被解读为“不a”，如下例所示：

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

该短语`if !allowedEntry`可以被理解为“如果不允许进入”。后续行仅在“不允许进入”为真时才被执行; 那就是，如果`allowedEntry`是`false`。

正如在这个例子中，仔细选择布尔常量和变量名称可以帮助保持代码的可读性和简洁性，同时避免双重否定或混淆逻辑语句。

逻辑AND运算符

的*逻辑AND运算符*（`a && b`）创建逻辑表达式，其中这两个值必须`true`为整体表达也有`true`。

如果任何一个值都是`false`，则整体表达式也将是`false`。事实上，如果第一个值是`false`第二个值，那么甚至不会评估第二个值，因为它不可能使整体表达式等于`true`。这就是所谓*的短路评估*。

这个例子考虑两个`Bool`值，并且只允许访问如果价值观`true`：

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

逻辑OR运算符

的*逻辑OR运算符*（`a || b`）是来自两个相邻管字符制成中缀运算符。您可以使用它来创建逻辑表达式，其中只有一个值必须`true`用于整体表达式`true`。

像上面的逻辑与运算符一样，逻辑或运算符使用短路评估来考虑其表达式。如果逻辑或表达式的左侧是true，右侧不计算，因为它不能改变整个表达式的结果。

在下面的例子中，第一个Bool值（hasDoorKey）是false，但第二个值（knowsOverridePassword）是true。因为有一个值true

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

逻辑运算符的组合

您可以组合多个逻辑运算符来创建更长的复合表达式：

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

此示例使用多个&&和||运算符来创建较长的复合表达式。然而，&&和||运算符仍然只能运行两个值，所以这实际上是三个较小的表达式链接在一起。该示例可以解读为：

如果我们输入了正确的门码并通过了视网膜扫描，或者如果我们有一个有效的门钥匙，或者我们知道紧急改写密码，则允许进入。

基于，和的值enteredDoorCode，前两个子表达式是。但是，紧急重写密码是已知的，因此整体复合表达式仍然会评估为。 passedRetinaScanhasDoorKeyfalsetrue

> 注意
>
> 迅捷逻辑运算符&&和||是左结合，这意味着与多个逻辑运算符复合表达式首先评估最左边的子表达式。

在本页

显式括号

在不严格需要的情况下包含括号有时很有用，可以使复杂表达式的意图更容易阅读。在上面的门访问示例中，在复合表达式的第一部分周围添加圆括号以使其意图明确是有用的：

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

括号清楚地表明前两个值在总体逻辑中被认为是单独的可能状态的一部分。复合表达式的输出不变，但整体意图对读者更清楚。可读性始终优于简洁性; 在他们帮助明确你的意图的地方使用括号。