赋值和算术运算符

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程

基本概念

- 一元运算符对一个目标进行操作。一元前缀运算符（如 !b），一元后缀运算符（b!）。
- 二元运算符对两个目标进行操作（比如 a + b ）同时因为它们出现在两个目标之间，所以是中缀。
- 三元运算符操作三个目标。Swift 语言也仅有一个三元运算符，三元条件运算符（ a ? b : c ）。

Swift 运算符的改进

- Swift 在支持 C 中的大多数标准运算符的同时也增加了一些排除常见代码错误的能力：
  - 赋值符号（ = ）不会返回值，以防它被误用于等于符号（ == ）的意图上。
  - 算数符号（ + , - , * , / , % 以及其他）可以检测并阻止值溢出，以避免你在操作比储存类型允许的范围更大或者更小的数字时得到各种奇奇怪怪的结果。

赋值运算符

- 赋值运算符将一个值赋给另外一个值。
- 如果赋值符号右侧是拥有多个值的元组，它的元素将会一次性地拆分成常量或者变量。
- Swift 的赋值符号自身不会返回值。

赋值运算符不会返回值

OC

```objc
NSInteger a = 1;
NSInteger b = 0;
if (b == a) {
    //do something
}
if (b = a) {
    //do something
}
```

⚠ Using the result of an assignment as a condition without parentheses

```objc
- (instancetype)init
{
    if (self = [super init]) {
        //do something
    }
    return self;
}
```

Swift

```swift
3  var a = 1
4  var b = 0
5  if b = a {
6      //do something
   }
8
```

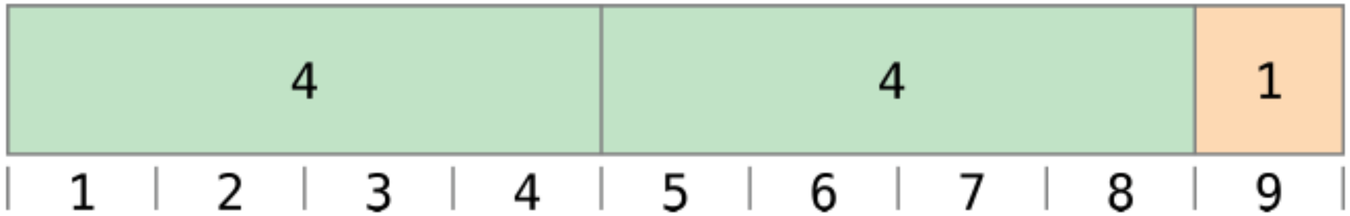
```
error: Operator.playground:5:6: error: use of '=' in a boolean context, did you mean '=='?
if b = a {
   ~ ^ ~
     ==
```

算术运算符 - 标准运算符

- 标准算术运算符 + - * /
- 加法运算符同时也支持 String 的拼接
- Swift 算术运算符默认不允许值溢出

算术运算符 - 余数运算符

- 余数运算符（ a % b ）可以求出多少个 b 的倍数能够刚好放进 a 中并且返回剩下的值（就是我们所谓的余数）。
- 当 a 是负数时也使用相同的方法来进行计算。
- 当 b 为负数时它的正负号被忽略掉了。这意味着 a % b 与 a % -b 能够获得相同的答案。



算术运算符 - 一元

- 数字值的正负号可以用前缀 - 来切换，我们称之为一元减号运算符。
- 一元减号运算符（ - ）直接在要进行操作的值前边放置，不加任何空格。
- 一元加号运算符 （ + ）直接返回它操作的值，不会对其进行任何的修改。

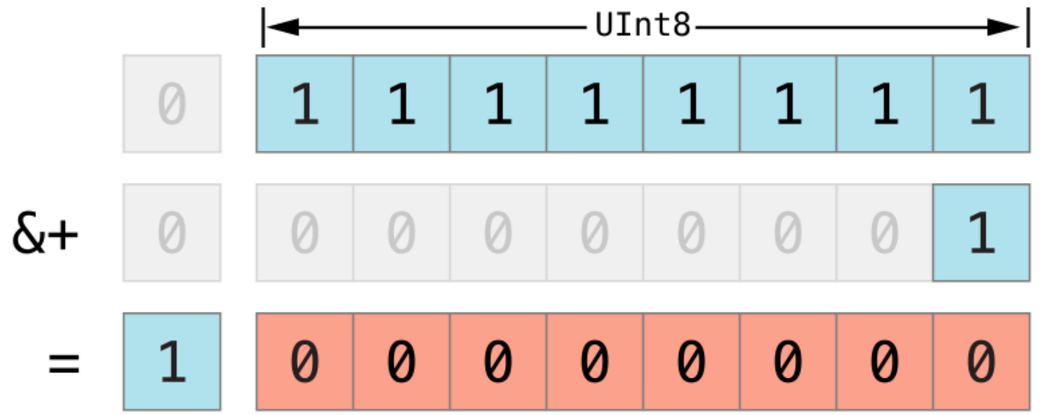
溢出运算符

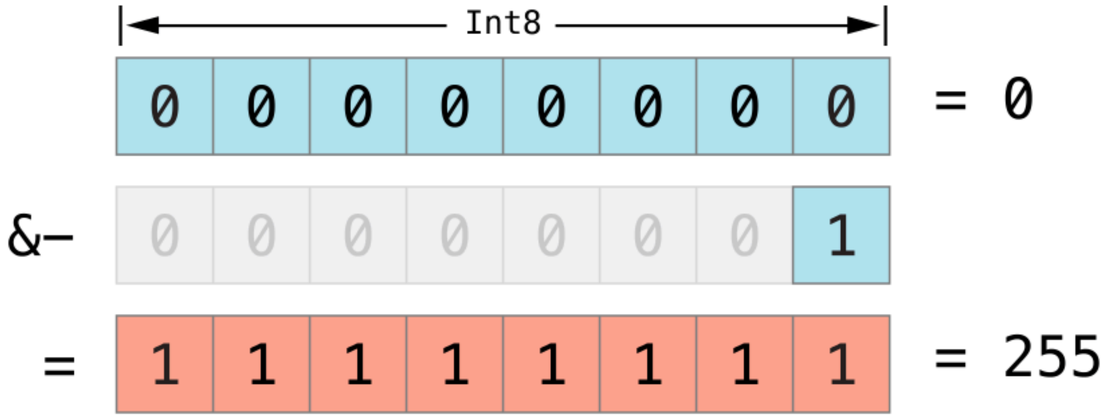
溢出运算符

- 在默认情况下，当向一个整数赋超过它容量的值时，Swift 会报错而不是生成一个无效的数，给我们操作过大或者过小的数的时候提供了额外的安全性。
- 同时提供三个算数溢出运算符来让系统支持整数溢出运算：
  - 溢出加法 （ &+ ）
  - 溢出减法 （ &- ）
  - 溢出乘法 （ &* ）

值溢出

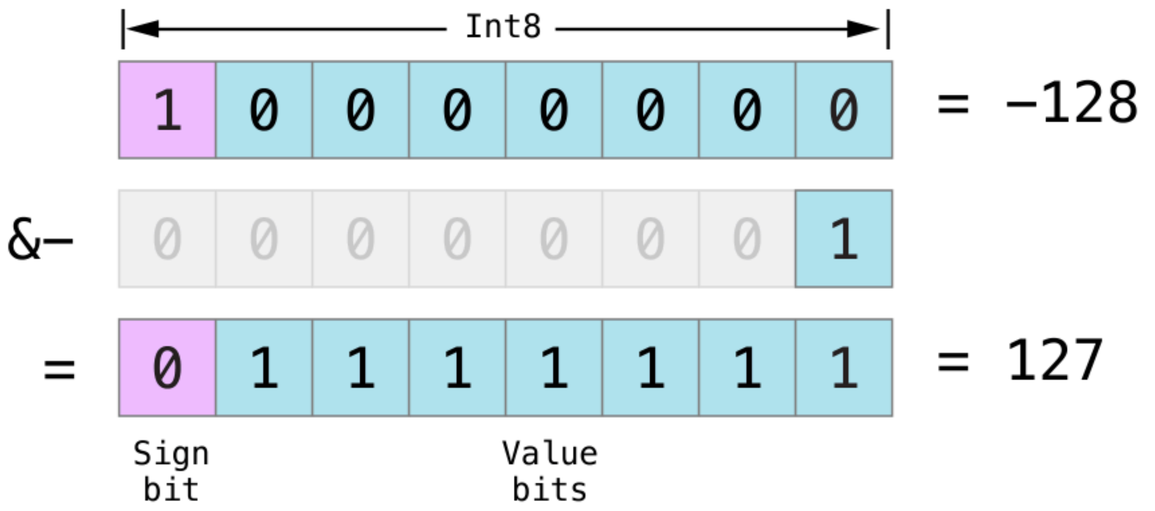
- 数值可以出现向上溢出或向下溢出





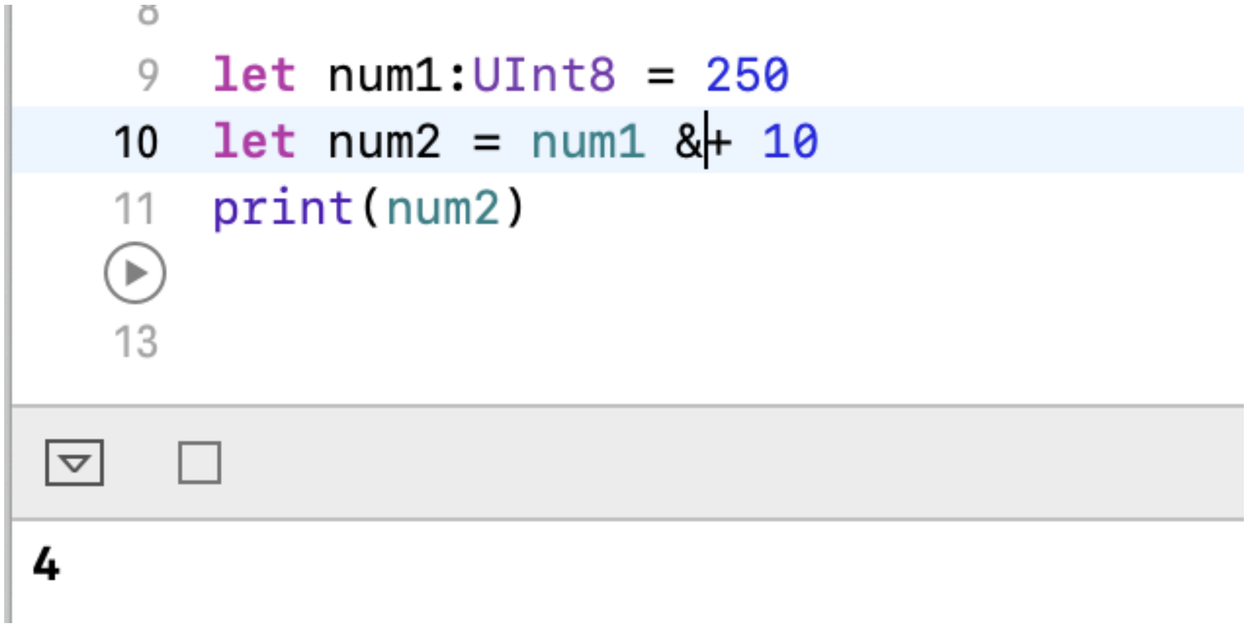
值溢出

- 溢出也会发生在有符号整型数值上。
- 对于无符号与有符号整型数值来说，当出现上溢时，它们会从数值所能容纳的最大数变成最小的数。同样的，当发生下溢时，它们会从所能容纳的最小数变成最大的数。



值溢出

```
8
9  let num1:UInt8 = 250
10 let num2 = num1 + 10   error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=...
   print(num2)
12
```



合并空值运算符

合并空值运算符

- 合并空值运算符（ a ?? b ）如果可选项 a 有值则展开，如果没有值，是 nil ，则返回默认值 b 。
- 表达式 a 必须是一个可选类型。表达式 b 必须与 a 的储存类型相同。

合并空值运算符

- 实际上是三元运算符作用到 Optional 上的缩写（a != nil ? a! : b）。
- 如果 a 的值是非空的，b 的值将不会被考虑，也就是合并空值运算符是短路的。

合并空值运算符

```swift
func addTowNumber(num1: Int?, num2: Int?) -> Int {
    // 1.强制解包有风险,如果 x 或者 y 有为 nil 会崩
    //return num1! + num2!
    // 2.使用 if 判断，但是如果直接使用if，参数很多的时候，会使代码很丑
    if num1 != nil && num1 != nil {
        return num1! + num1!
    } else {
        return 0
    }
    // 使用 运算符 ??
    return (num1 ?? 0) + (num1 ?? 0)

}
```

区间运算符

闭区间运算符

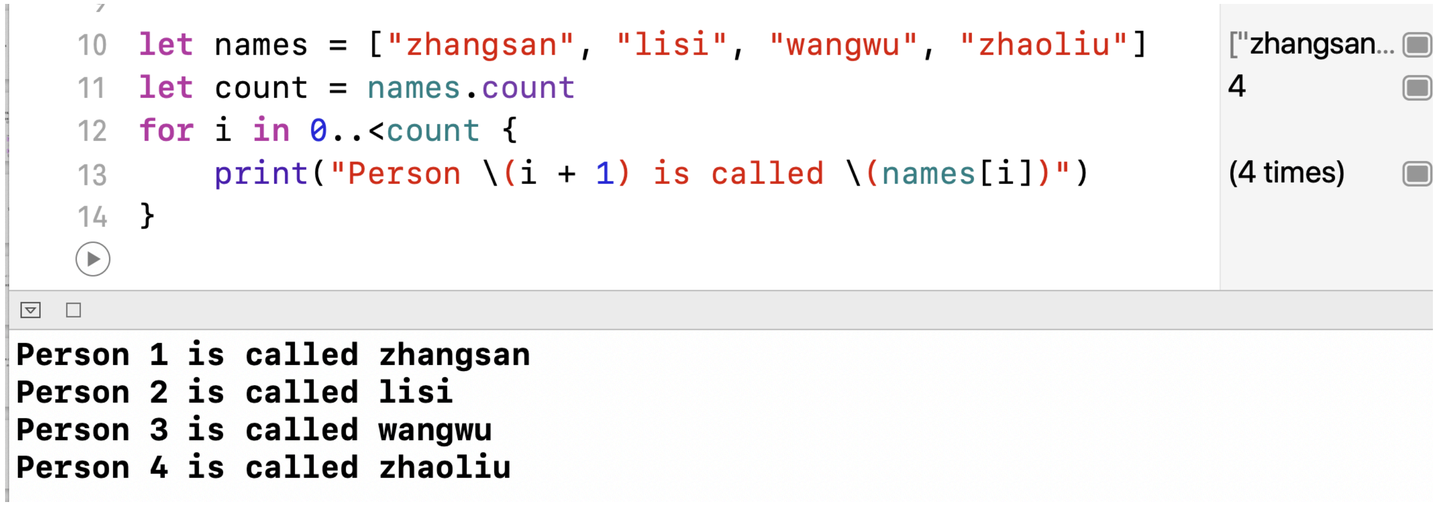
- 闭区间运算符（ a...b ）定义了从 a 到 b 的一组范围，并且包含 a 和 b 。a 的值不能大于 b 。

```
5
6 for index in 1...5 {
7     print("\(index) times 5 is \(index * 5)")
8 }
```

```
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
```

半开区间运算符

- 半开区间运算符（ a..<b ）定义了从 a 到 b 但不包括 b 的区间。
- 如同闭区间运算符，a 的值也不能大于 b ，如果 a 与 b 的值相等，那返回的区间将会是空的。



单侧区间

- 闭区间有另外一种形式来让区间朝一个方向尽可能的远，这种区间叫做单侧区间。
- 半开区间运算符同样可以有单侧形式，只需要写它最终的值。
- 比如说，一个包含数组所有元素的区间，从索引 2 到数组的结束。在这种情况下，你可以省略区间运算符一侧的值。

```
17 for name in names[2...] {
18     print(name)
19 }
20 print("----------")
21 for name in names[...2] {
22     print(name)
23 }
```

```
wangwu
zhaoliu
----------
zhangsan
lisi
wangwu
```

```
26 for name in names[..<2] {
27     print(name)
28 }
```

```
zhangsan
lisi
```

单侧区间

- 单侧区间可以在其他上下文中使用，不仅仅是下标。
- 不能遍历省略了第一个值的单侧区间，因为遍历根本不知道该从哪里开始。你可以遍历省略了最终值的单侧区间。

```
30 let range = ...5                    PartialRangeThrough<Int>
31 range.contains(7)    // false       false
32 range.contains(4)    // true        true
33 range.contains(-1)   // true        true
```

字符串索引区间

- 字符串范围也可以使用区间运算符

```
var welcome = "hello,world"

let range = welcome.index(welcome.endIndex, offsetBy:
    -6)..<welcome.endIndex
welcome.removeSubrange(range)
```

倒序索引

- 通过 reversed() 方法，我们可以将一个正序循环变成逆序循环。

```
var welcome = "hello,world"

let range = welcome.index(welcome.endIndex, offsetBy:
    -6)..<welcome.endIndex
welcome.removeSubrange(range)
```

Comparable 区间

- 区间运算符可以作用在 Comparable 类型上，返回闭区间和半闭区间。

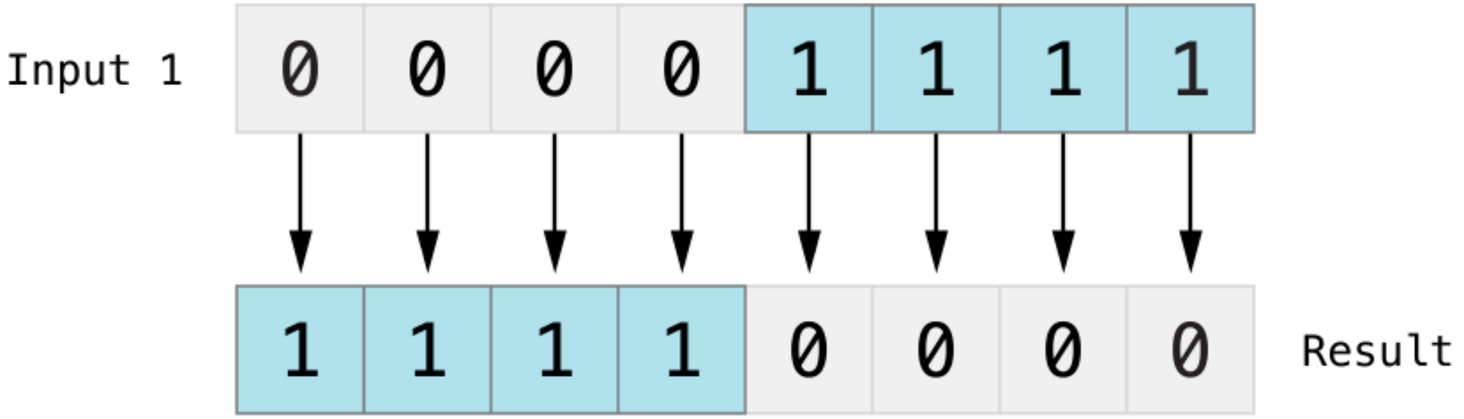
```
39  let welcome = "hello,world"
40  let interval = "a"..."z"
41  for c in welcome {
42      if !interval.contains(String(c)) {
43          print("\(c)不是小写字母")
44      }
45
46  }
```

,不是小写字母

位运算符

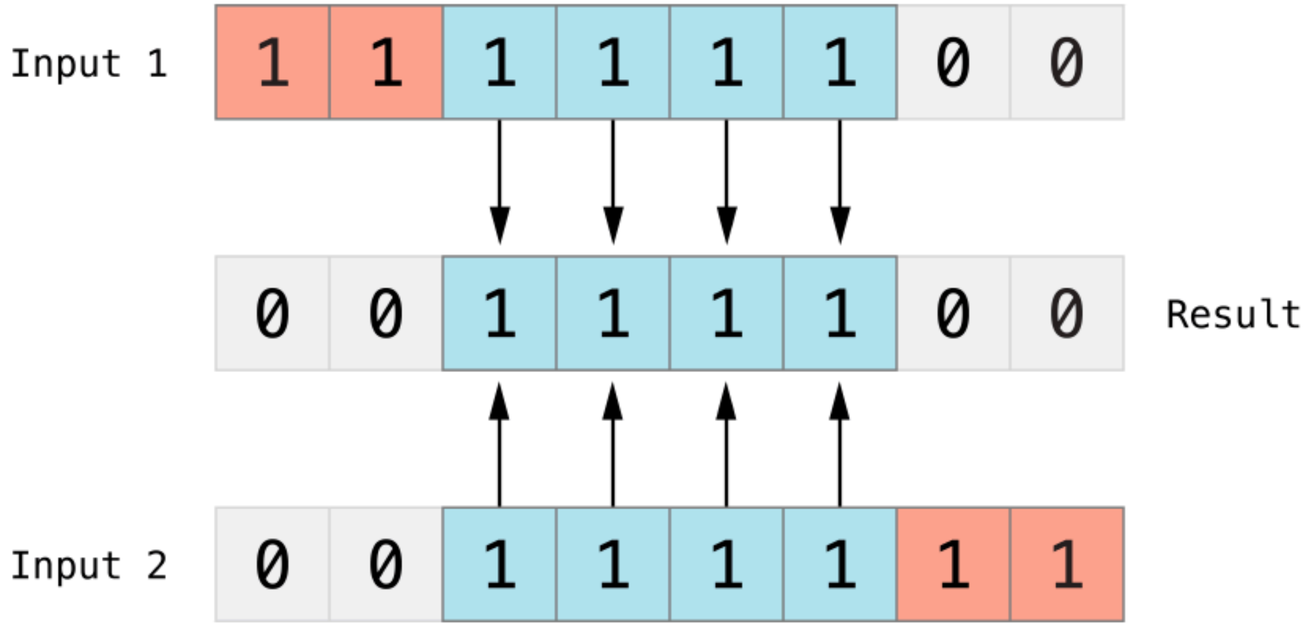
位取反运算符

- 位取反运算符（ ~ ）是对所有位的数字进行取反操作



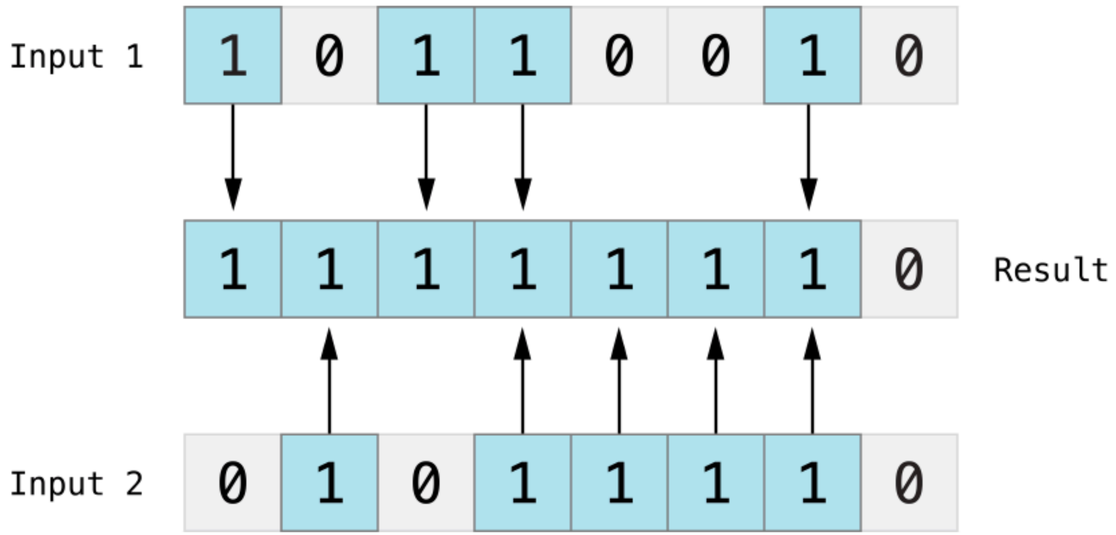
位与运算符

- 位与运算符（ & ）可以对两个数的比特位进行合并。它会返回一个新的数，只有当这两个数都是 1 的时候才能返回 1 。



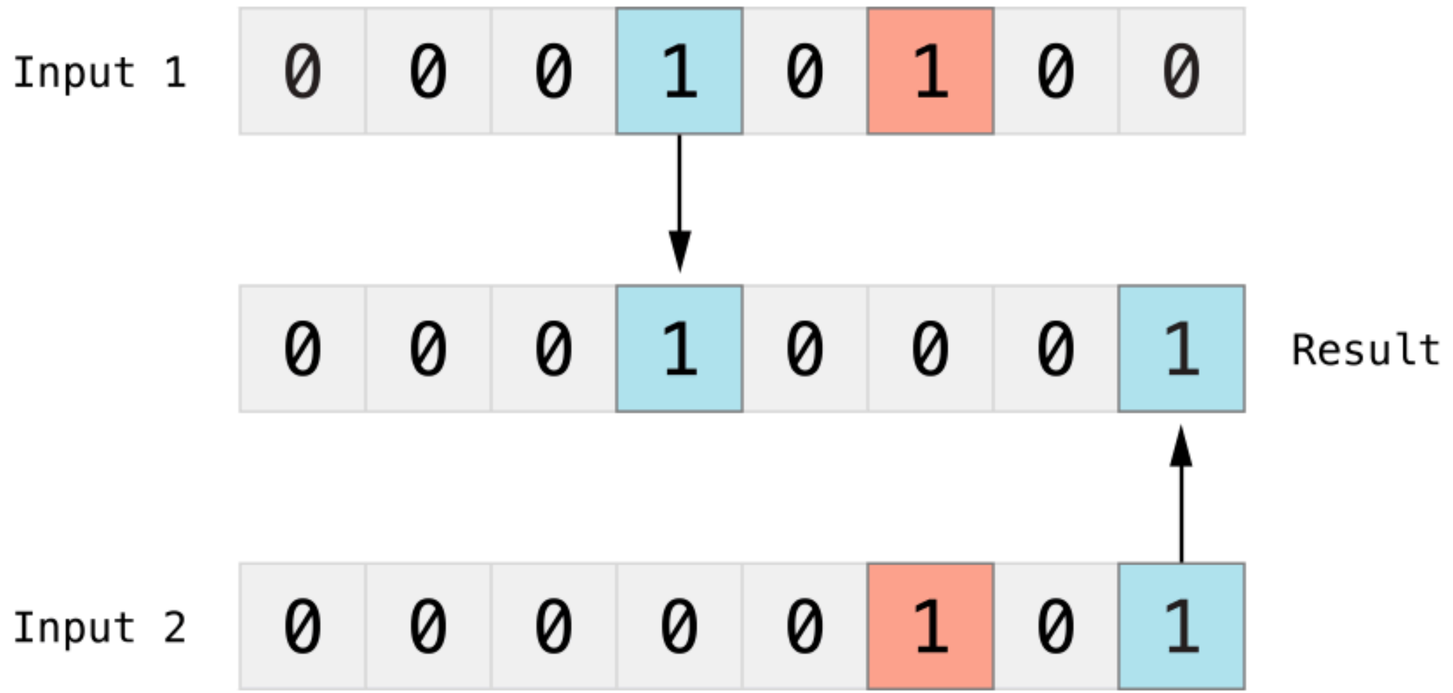
位或运算符

- 位或运算符（ | ）可以对两个比特位进行比较，然后返回一个新的数，只要两个操作位任意一个为 1 时，那么对应的位数就为 1 。



位异或运算符

- 位异或运算符，或者说“互斥或”（ ^ ）可以对两个数的比特位进行比较。它返回一个新的数，当两个操作数的对应位不相同时，该数的对应位就为 1 。



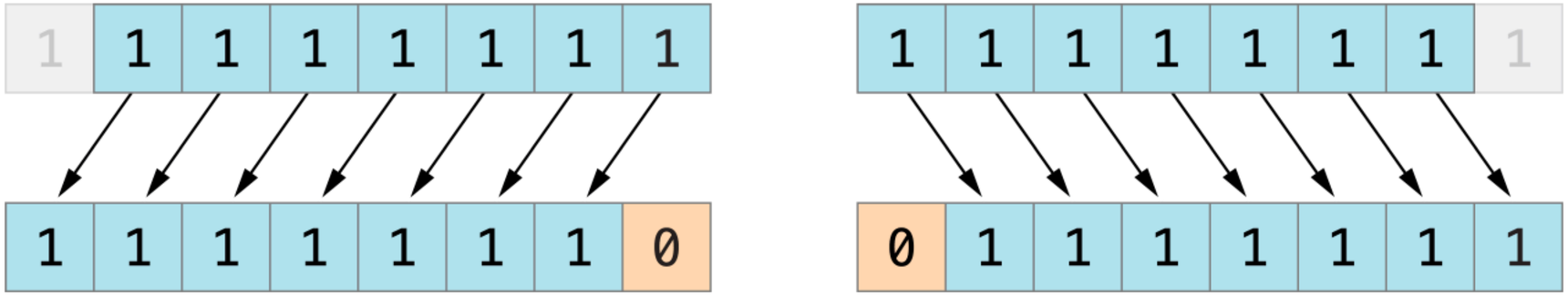
位左移和右移运算符

- 位左移运算符（ << ）和位右移运算符（ >> ）可以把所有位数的数字向左或向右移动一个确定的位数。

- 位左移和右移具有给整数乘以或除以二的效果。将一个数左移一位相当于把这个数翻倍，将一个数右移一位相当于把这个数减半。

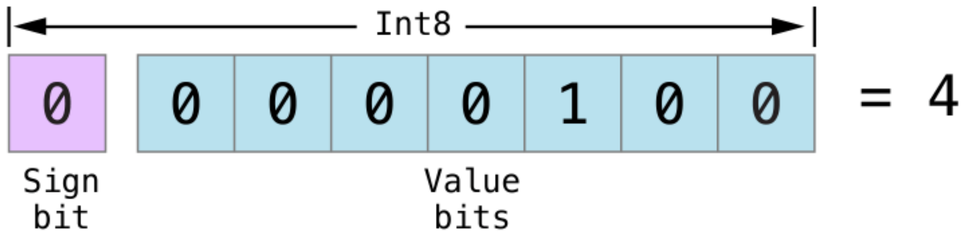
无符号整数的移位操作

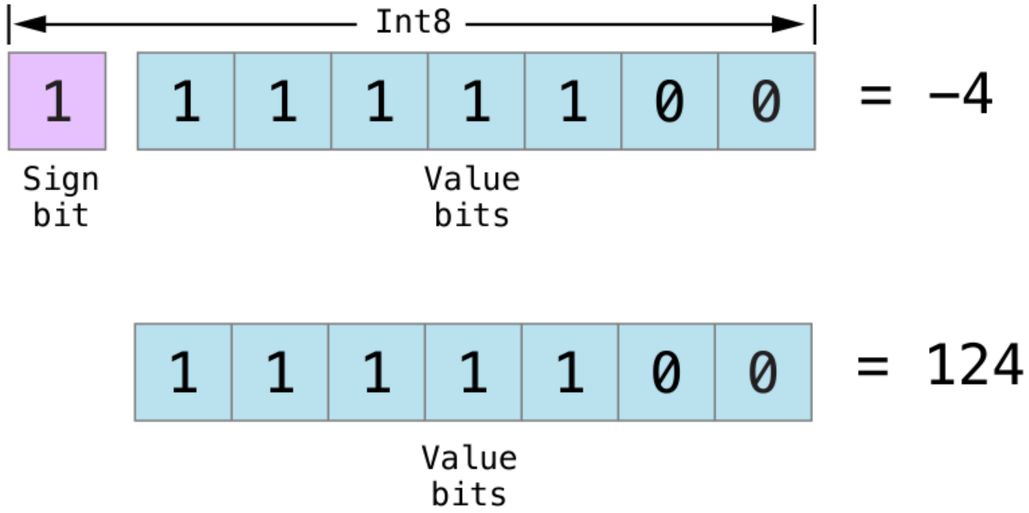
- 已经存在的比特位按指定的位数进行左移和右移
- 任何移动超出整型存储边界的位都会被丢弃
- 用 0 来填充向左或向右移动后产生的空白位



有符号整数的移位操作

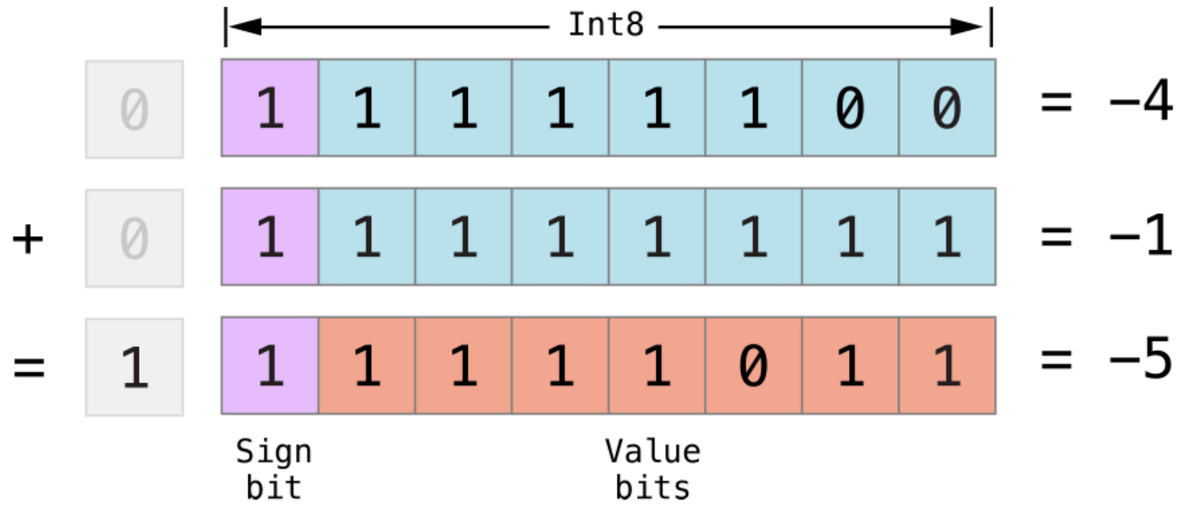
- 有符号整数使用它的第一位（所谓的符号位）来表示这个整数是正数还是负数。符号位为 0 表示为正数，1 表示为负数。
- 其余的位数（所谓的数值位）存储了实际的值。有符号正整数和无符号数的存储方式是一样的，都是从 0 开始算起。
- 但是负数的存储方式略有不同。它存储的是 2 的 n 次方减去它的绝对值，这里的 n 为数值位的位数。





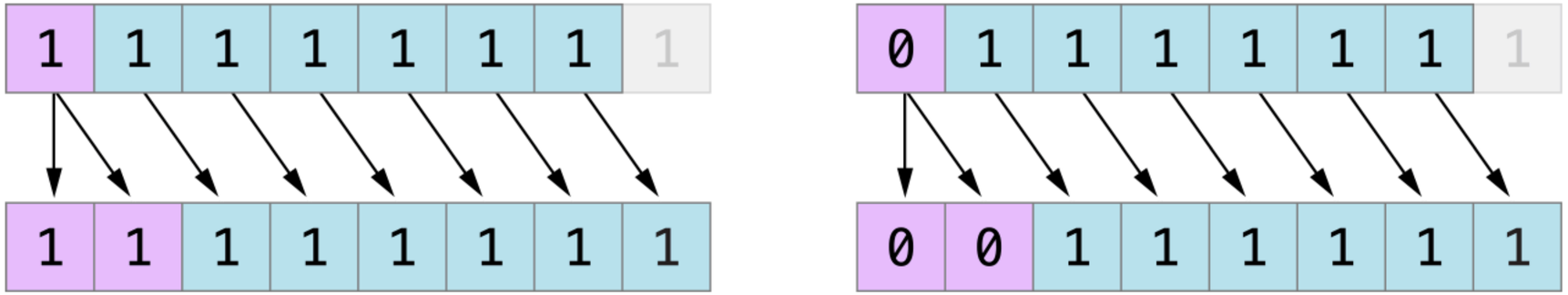
补码表示的优点

- 首先，如果想给 -4 加个 -1 ，只需要将这两个数的全部八个比特位相加（包括符号位），并且将计算结果中超出的部分丢弃。



补码表示的优点

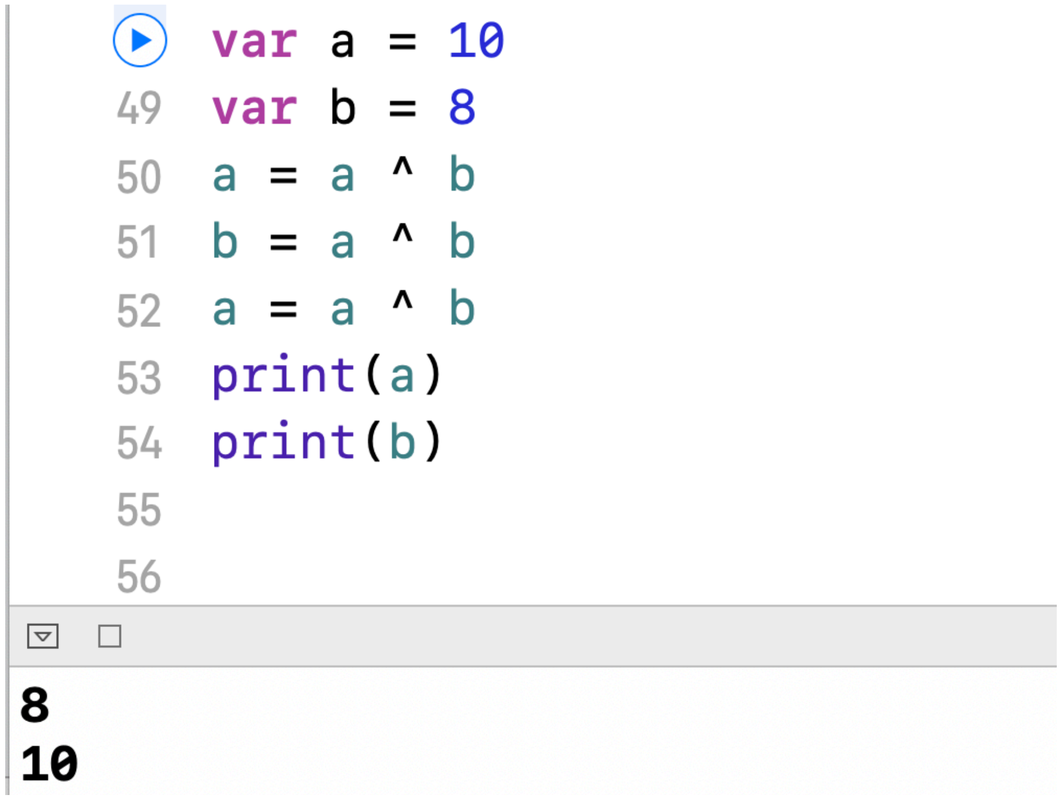
- 其次，使用二进制补码可以使负数的位左移和右移操作得到跟正数同样的效果，即每向左移一位就将自身的数值乘以 2 ，每向右移一位就将自身的数值除以 2 。要达到此目的，对有符号整数的右移有一个额外的规则：当对正整数进行位右移操作时，遵循与无符号整数相同的规则，但是对于移位产生的空白位使用符号位进行填充，而不是 0 。



位运算符经典算法

两个数字交换

- 不借助临时变量，交换两个变量的值。



求无符号整数二进制中 1 的个数

- 给定一个无符号整型（UInt）变量，求其二进制表示中 “1” 的个数，要求算法的执行效率尽可能的高。

求无符号整数二进制中 1 的个数

- 思路：看一个八位整数 10 100 001 ，先判断最后一位是否为 1 ，而“与”操作可以达到目的。可以把这个八位的数字与 00000001 进行“与”操作。如果结果为 1 ，则表示当前八位数的最后一位为 1 ，否则为 0 。怎么判断第二位呢？向右移位，再延续前面的判断即可。

```
func countOfOnes(num: UInt) -> UInt {
    var count: UInt = 0
    var temp = num
    while temp != 0 {
        count += temp & 1
        temp = temp >> 1
    }
    return count
}
```

求无符号整数二进制中 1 的个数

- 如果整数的二进制中有较多的 0，那么我们每一次右移一位做判断会很浪费，怎么改进前面的算法呢？有没有办法让算法的复杂度只与“1”的个数有关？

求无符号整数二进制中 1 的个数

- 思路：为了简化这个问题，我们考虑只有高位有 1 的情况。例如：11 000 000，如何跳过前面低位的 6 个 0，而直接判断第七位的 1？我们可以设计 11 000 000 和 10 111 111（也就是 11 000 000 - 1）做“与”操作，消去最低位的 1 。如果得到的结果为 0，说明我们已经找到/消去里面最后一个 1 。如果不为 0，那么说明我们消去了最低位的 1，但是二进制中还有其他的 1，我们的计数器需要加 1，然后继续上面的操作。

计数器 count = 0

步骤一： 整数不为 0，说明二进制里肯定有 1，count = 1

11 000 000 & 10 111 111 = 10 000 000（消去第七位的 1 ）。

步骤二：结果不为 0，说明二进制里还有 1， count = 2

10 000 000 & 01 111 111 = 0（消去第八位的 1 ）。

步骤三：结果为 0，终止，返回 count 为 2 。

```swift
func countOfOnes2(num: UInt) -> UInt {
    var count: UInt = 0
    var temp = num
    while temp != 0 {
        count += 1
        temp = temp & (temp - 1)
    }
    return count
}

print(countOfOnes2(num: 0x100010))
```

引申：如何判断一个整数位为 2 的整数次幂

- 给定一个无符号整型（UInt）变量，判断是否为 2 的整数次幂。

- 思路：一个整数如果是 2 的整数次方，那么它的二进制表示中有且只有一位是 1 ，而其它所有位都是 0 。 根据前面的分析，把这个整数减去 1 后再和它自己做与运算，这个整数中唯一的 1 就变成 0 了，也就是得到的结尾为 0 。

```
func isPowerOfTwo(num: UInt) -> Bool {
    return (num & (num - 1)) == 0
}

print(isPowerOfTwo(num: 512))
```

位运算符经典算法 2

缺失的数字

- 很多成对出现的正整数保存在磁盘文件中，注意成对的数字不一定是相邻的，如 2, 3, 4, 3, 4, 2……，由于意外有一个数字消失了，如何尽快找到是哪个数字消失了？

缺失的数字

- 思路：考虑“异或”操作的定义，当两个操作数的对应位不相同时，该数的对应位就为 1 。也就是说如果是相等的两个数“异或”，得到的结果为 0 ，而 0 与任何数字“异或”，得到的是那个数字本身。所以我们考虑将所有的数字做“异或”操作，因为只有一个数字消失，那么其他两两出现的数字“异或”后为 0 ，0 与仅有的一个的数字做“异或”，我们就得到了消失的数字是哪个。

```
func findLostNum(nums: [UInt]) -> UInt {
    var lostNum:UInt = 0
    for num in nums {
        lostNum = lostNum ^ num
    }
    return lostNum
}

print(findLostNum(nums: [1, 2, 3, 4, 3, 2, 1]))
```

缺失的数字 2

- 如果有两个数字意外丢失了（丢失的不是相等的数字），该如何找到丢失的两个数字?

缺失的数字 2

- 思路：设题目中这两个只出现 1 次的数字分别为 A 和 B，如果能将 A，B 分开到二个数组中，那显然符合“异或”解法的关键点了。因此这个题目的关键点就是将 A，B 分开到二个数组中。由于 A，B 肯定是不相等的，因此在二进制上必定有一位是不同的。根据这一位是 0 还是 1 可以将 A 和 B 分开到 A组 和 B组。而这个数组中其它数字要么就属于 A 组，要么就属于 B 组。再对 A组 和 B组 分别执行“异或”解法就可以得到 A，B 了。而要判断 A，B 在哪一位上不相同，只要根据 “A 异或 B” 的结果就可以知道了，这个结果在二进制上为 1 的位都说明 A，B 在这一位上是不相同的。

缺失的数字 2

```swift
func findTwoLostNums(nums: [UInt]) ->(UInt, UInt) {
    var lostNum1:UInt = 0
    var lostNum2: UInt = 0
    var temp: UInt = 0
    //计算两个数的异或结果
    for num in nums {
        temp = temp ^ num
    }
    //找到第一个不为1的位
    var flag: UInt = 1;
    while ((flag & temp) == 0) {
        flag = flag << 1
    }
    //找两个丢失的数字
    for num in nums {
        if (num & flag) == 0 {
            lostNum1 = lostNum1 ^ num
        } else {
            lostNum2 = lostNum2 ^ num
        }
    }
    return (lostNum1, lostNum2)
}

print(findTwoLostNums(nums: [1, 2, 3, 4, 5, 3, 2, 1]))
```

思考题 - 缺失的数字 3

- 数组中，只有一个数出现一次，剩下都出现三次，找出出现一次的数字。

运算符的优先级和结合性

运算符优先级和结合性

- 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。
- 结合性定义了具有相同优先级的运算符是如何结合（或关联）的——是与左边结合为一组，还是与右边结合为一组。可以这样理解：“它们是与左边的表达式结合的”或者“它们是与右边的表达式结合的”。

运算符优先级和结合性

- 2 + 3 % 4 * 5

- 如果严格按照从左到右计算
  - 2 + 3 = 5
  - 5 % 4 = 1
  - 1 * 5 = 5
- 按照优先级优先级
  - 3 % 4 = 3
  - 3 * 5 = 15
  - 2 * 15 = 17

运算符优先级 - 显式括号

- 2 + 3 % 4 * 5 等价于 2 + ((3 % 4) * 5)

运算符优先级 - 显式括号

- Swift 语言中逻辑运算符 && 和 || 是左相关的，这意味着多个逻辑运算符组合的表达式会首先计算最左边的子表达式。

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



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

运算符方法

运算符重载

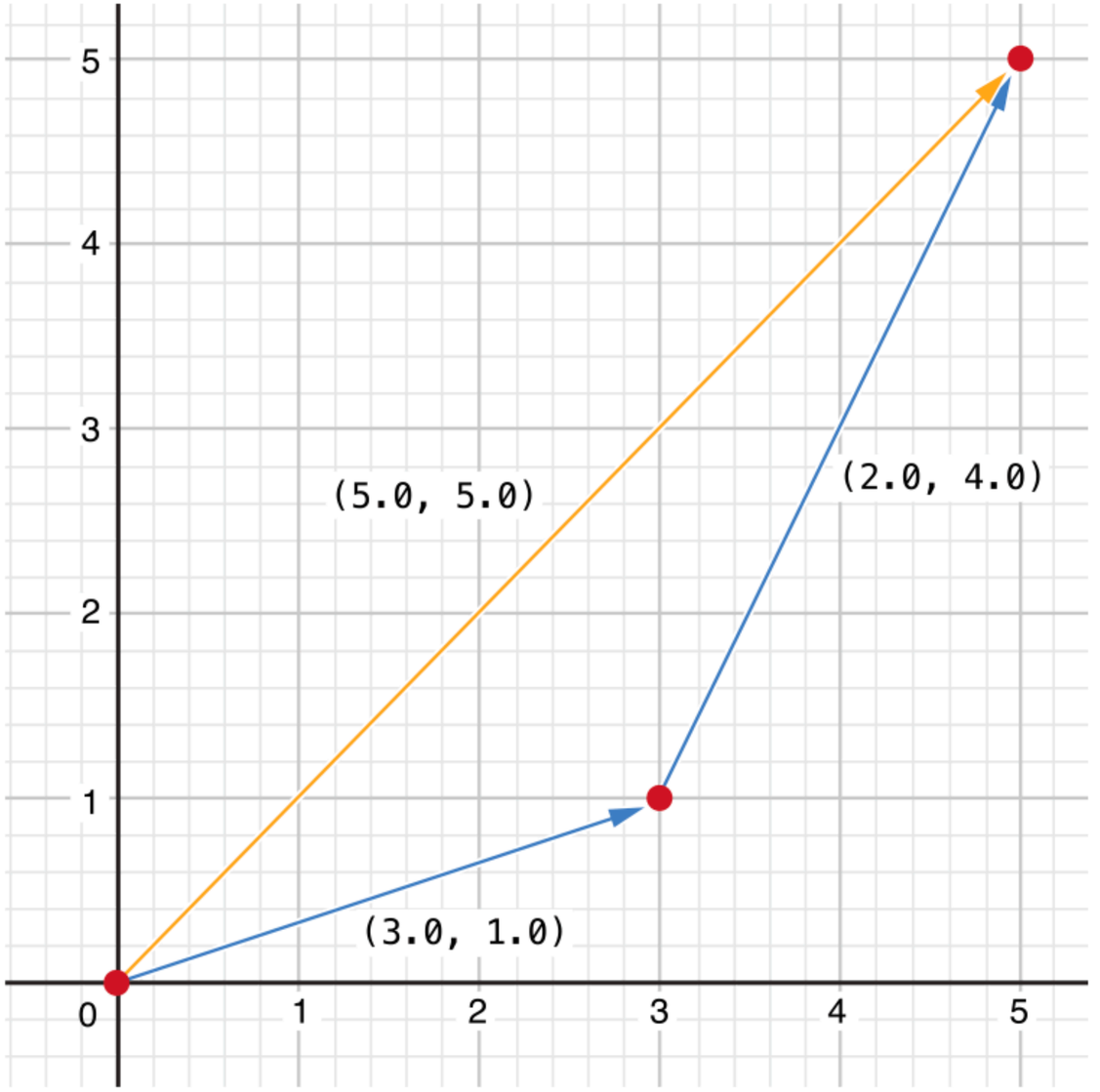
- 类和结构体可以为现有的运算符提供自定义的实现，称为运算符重载。

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
}

let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
```

运算符重载



一元运算符重载

- 类与结构体也能提供标准一元运算符的实现。
- 要实现前缀或者后缀运算符，需要在声明运算符函数的时候在 func 关键字之前指定 prefix 或者 postfix 限定符。

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

let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
```

组合赋值运算符重载

- 组合赋值运算符将赋值运算符(=)与其它运算符进行结合。
- 在实现的时候，需要把运算符的左参数设置成 inout 类型，因为这个参数的值会在运算符函数内直接被修改。

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}

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

等价运算符重载

- 自定义类和结构体不接收等价运算符的默认实现，也就是所谓的“等于”运算符（ == ）和“不等于”运算符（ != ）。

- 要使用等价运算符来检查你自己类型的等价，需要和其他中缀运算符一样提供一个“等于”运算符，并且遵循标准库的 Equatable 协议

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}

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

等价运算符重载

- Swift 为以下自定义类型提供等价运算符合成实现：
  - 只拥有遵循 Equatable 协议存储属性的结构体
  - 只拥有遵循 Equatable 协议关联类型的枚举
  - 没有关联类型的枚举

```
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")
}
```

自定义运算符

自定义运算符

- 除了实现标准运算符，在 Swift 当中还可以声明和实现自定义运算符（custom operators）。
- 新的运算符要在全局作用域内，使用 operator 关键字进行声明，同时还要指定 prefix 、infix 或者 postfix 限定符。

```
prefix operator +++ {}

extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
```

自定义中缀运算符的优先级和结合性

- 自定义的中缀（ infix ）运算符也可以指定优先级和结合性
- 每一个自定义的中缀运算符都属于一个优先级组
- 优先级组指定了自定义中缀运算符和其他中缀运算符的关系

```
precedencegroup MyPrecedence {
    associativity: left
    lowerThan: AdditionPrecedence
}
```

自定义中缀运算符的优先级和结合性

```
204 infix operator +-: AdditionPrecedence
205 extension Vector2D {
206     static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
207         return Vector2D(x: left.x + right.x, y: left.y - right.y)
208     }
209 }
210
211 infix operator *^: MultiplicationPrecedence
212 precedencegroup MyPrecedence {
213     associativity: left
214     lowerThan: AdditionPrecedence
215 }
216 extension Vector2D {
217     static func *^ (left: Vector2D, right: Vector2D) -> Vector2D {
218         return Vector2D(x: left.x * right.x, y: left.y * left
219             .y + right.y * right.y)
220     }
221 }
222
223 let firstVector = Vector2D(x: 1.0, y: 2.0)
224 let secondVector = Vector2D(x: 3.0, y: 7.0)
225 let plusMinusVector = firstVector +- secondVector
226 let thirdVector = Vector2D(x: 2.0, y: 2.0)
227 let vector = firstVector +- secondVector *^ thirdVector
228 print(vector.x)
229 print(vector.y)
```

```
7.0
-51.0
```

自定义中缀运算符的优先级和结合性

```
204 infix operator +-: AdditionPrecedence
205 extension Vector2D {
206     static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
207         return Vector2D(x: left.x + right.x, y: left.y - right.y)
208     }
209 }
210
211 infix operator *^: MyPrecedence
212 precedencegroup MyPrecedence {
213     associativity: left
214     lowerThan: AdditionPrecedence
215 }
216 extension Vector2D {
217     static func *^ (left: Vector2D, right: Vector2D) -> Vector2D {
218         return Vector2D(x: left.x * right.x, y: left.y * left
219             .y + right.y * right.y)
220     }
221 }
222
223 let firstVector = Vector2D(x: 1.0, y: 2.0)
224 let secondVector = Vector2D(x: 3.0, y: 7.0)
225 let plusMinusVector = firstVector +- secondVector
226 let thirdVector = Vector2D(x: 2.0, y: 2.0)
227 let vector = firstVector +- secondVector *^ thirdVector
228 print(vector.x)
229 print(vector.y)
```

```
8.0
29.0
```

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程