

Swift 概览

 cnswift.org/a-swift-tour

依照传统，使用新语言写的第一个程序都应该是屏幕上打印“Hello,world!”，使用 Swift 语言，你可以在一行中完成。

```
1 print("Hello, world!")
```

如果你曾使用 C 或者 Objective-C 写代码，那么 Swift 的语法不会让你感到陌生——在 Swift 语言当中，这一行代码就是一个完整的程序！你不需要为每一个功能导入单独的库比如输入输出和字符串处理功能。写在全局范围的代码已被用来作为程序的入口，所以你不再需要 main() 函数。同样，你也不再需要在每句代码后边写分号。

通过向你展示各种编程任务，这个概览会给你足够的信息来开始使用 Swift 进行开发。如果觉得这个概览不够详细，不要担心——这个概览所介绍的内容都会在本书的余下章节里进行详细解释。

为了更好的阅读体验，我们推荐你使用 Xcode 里的 Playground 打开本章内容，Playground 允许你编辑代码并立即看到代码的运算结果。

下载本章的 [Playground](#) (官网链接)

简单值

使用 let 来声明一个常量，用 var 来声明一个变量。常量的值在编译时并不要求已知，但是你必须为其赋值一次。这意味着你可以使用常量来给一个值命名，然后一次定义多次使用。

```
1 varmyVariable=42
2 myVariable=50
3 letmyConstant=42
```

常量或者变量必须拥有和你赋给它们的值相同的类型。不过，你并不需要总是显式地写出类型。在声明一个常量或者变量的时候直接给它们赋值就可以让编译器推断它们的类型。比如下面的例子，编译器就会推断 myVariable 是一个整型，因为它的初始值是一个整型。

如果初始值并不能提供足够的信息（或者根本没有提供初始值），就需要在变量的后边写出来了，用冒号分隔。

```
1 letimplicitInteger=70
2 letimplicitDouble=70.0
3 letexplicitDouble:Double=70
```

实验

创建一个常量并显式声明类型为 Float，赋值为 4

值绝对不会隐式地转换为其他类型。如果你需要将一个值转换为不同的类型，需要使用对应的类型显示地声明。

```
1 letlabel="The width is "
2 letwidth=94
3 letwidthLabel=label+String(width)
```

实验

试试去掉最后一行的转换标记 String ，看看会有怎样的报错 ?

其实还有一种更简单的方法来把值加入字符串 : 将值写在圆括号里 , 然后再在圆括号的前边写一个反斜杠 (\) , 举个栗子 :

```
1 letapples=3
2 letoranges=5
3 letappleSummary="I have \(apples) apples."
4 letfruitSummary="I have \(apples+oranges) pieces of fruit."
```

实验

使用 \() 来把一个浮点计算包含进字符串 , 然后再在一个欢迎语句中插入某人的名字。

使用方括号 ([]) 来创建数组或者字典 , 并且使用方括号来按照序号或者键访问它们的元素。

```
1 varshoppingList=["catfish","water","tulips","blue paint"]
2 shoppingList[1]="bottle of water"
3 varoccupations=[
4 "Malcolm":"Captain",
5 "Kaylee":"Mechanic",
6 ]
7 occupations["Jayne"]="Public Relations"
8
```

使用初始化器语法来创建一个空的数组或者字典。

```
1 letemptyArray=[String]()
2 letemptyDictionary=[String:Float]()
```

如果类型信息能被推断 , 那么你就可以用 [] 来表示空数组 , 用 [:] 来表示空字典。举个栗子 , 当你给变量设置新的值或者传参数给函数的时候。

```
1 shoppingList=[]
2 occupations=[:]
```

控制流

使用 if 和 switch 来做逻辑判断 , 使用 for-in , for , while , 以及 repeat-while 来做循环。使用圆括号把条件或者循环变量括起来不再是强制的了 , 不过仍旧需要使用花括号来括住代码块。

```
1 let individualScores=[75,43,103,87,12]
2 var teamScore=0
3 for score in individualScores{
4     if score>50{
5         teamScore+=3
6     }else{
7         teamScore+=1
8     }
9 }
10 print(teamScore)
```

在一个 `if` 语句当中，条件必须是布尔表达式——这意味着比如说 `if score{...}` 将会报错，不再隐式地与零做计算了。

你可以一起使用 `if` 和 `let` 来操作那些可能会丢失的值。这些值使用可选项表示。可选的值包括了一个值或者一个 `nil` 来表示值不存在。在一个值的类型后边使用问号（`?`）来把某个值标记为可选的。

```
1 var optionalString:String?="Hello"
2 print(optionalString==nil)
3 var optionalName:String?="John Appleseed"
4 var greeting="Hello!"
5 if let name=optionalName{
6     greeting="Hello, \(name)"
7 }
8
```

实验

把 `optionalName` 的值改为 `nil`。你得到的 `greeting` 是什么内容？添加 `else` 分句，如果 `optionalName` 为 `nil` 就设置不同的内容给 `greeting`。

如果可选项的值为 `nil`，则条件为 `false` 并且花括号里的代码将会被跳过。否则，可选项的值就会被展开且赋给 `let` 后边声明的常量，这样会让展开的值对花括号内的代码可用。

另一种处理可选值的方法是使用 `??` 运算符提供默认值。如果可选值丢失，默认值就会使用。

```
1 let nickName:String?=nil
2 let fullName:String="John Appleseed"
3 let informalGreeting="Hi \(nickName ?? fullName)"
```

`Switch` 选择语句支持任意类型的数据和各种类型的比较操作——它不再限制于整型和测试相等上。

```
1 let vegetable="red pepper"
2 switch vegetable{
3     case "celery":
4         print("Add some raisins and make ants on a log. ")
5     case "cucumber", "watercress":
6         print("That would make a good tea sandwich. ")
7     case let x where x.hasSuffix("pepper"):
8         print("Is it a spicy \(x)?")
9     default:
10        print("Everything tastes good in soup.")
11 }
```

实验

尝试去掉 default选项。会得到什么样的报错？

注意 let 可以用在模式里来指定匹配的值到一个常量当中。

在执行完 switch 语句里匹配到的 case 之后，程序就会从 switch 语句中退出。执行并不会继续跳到下一个 case 里，所以完全没有必要显式地在每一个 case 后都标记 break。

你可以使用 for-in 来遍历字典中的项目，这需要提供一对变量名来储存键值对。字典使用无序集合，所以键值的遍历也是无序的。

```
1 let interestingNumbers=[  
2   "Prime":[2,3,5,7,11,13],  
3   "Fibonacci":[1,1,2,3,5,8],  
4   "Square": [1,4,9,16,25],  
5 ]  
6 var largest=0  
7 for(kind,numbers)in interestingNumbers{  
8   for number in numbers{  
9     if number>largest{  
10       largest=number  
11     }  
12   }  
13 }  
14 print(largest)
```

实验

添加另一个变量来追踪哪一类的数字是最大的，同时那个最大的数是多少。

使用 while 来重复代码快直到条件改变。循环的条件可以放在末尾，这样可以保证循环至少运行了一次。

```
1 var n=2  
2 while n<100{  
3   n=n*2  
4 }  
5 print(n)  
6 var m=2  
7 repeat{  
8   m=m*2  
9 }while m<100  
10 print(m)  
11
```

你可以使用 ..< 来创造一个序列区间：

```
1 var total=0  
2 for i in 0..<4{  
3   total+=i  
4 }  
5 print(total)
```

使用 ..< 来创建一个不包含最大值的区间，使用 ... 来创造一个包含最大值和最小值的区间。

函数和闭包

使用 func 来声明一个函数。通过在名字之后在圆括号内添加一系列参数来调用这个方法。使用 -> 来分隔形式参数名字类型和函数返回的类型。

```
1 funcgreet(person:String,day:String)->String{  
2   return "Hello \$(person), today is \$(day)."  
3 }  
4 greet(person:"Bob",day:"Tuesday")
```

实验

移除 day 形式参数。添加一个参数来包含今日午餐吃了什么。然后在欢迎语句里显示出来。

默认情况下，函数使用他们的形式参数名来作为实际参数标签。在形式参数前可以写自定义的实际参数标签，或者使用 _ 来避免使用实际参数标签。

```
1 funcgreet(_person:String,on day:String)->String{  
2   return "Hello \$(person), today is \$(day)."  
3 }  
4 greet("John",on:"Wednesday")
```

使用元组来创建复合值——比如，为了从函数中返回多个值。元组中的元素可以通过名字或者数字调用。

```
1 funccalculateStatistics(scores:[Int])->(min:Int,max:Int,sum:Int){  
2   varmin=scores[0]  
3   varmax=scores[0]  
4   varsum=0  
5   for score in scores{  
6     if score > max{  
7       max = score  
8     } else if score < min{  
9       min = score  
10    }  
11    sum += score  
12  }  
13  return(min,max,sum)  
14 }  
15 let statistics = calculateStatistics(scores:[5,3,100,3,9])  
16 print(statistics.sum)  
17 print(statistics.2)  
18  
19
```

函数同样可以接受多个参数，然后把它们存放进数组当中。

```
1 funcsumOf(numbers:Int...)->Int{  
2   varsum=0  
3   for number in numbers{  
4     sum+=number  
5   }  
6   return sum  
7 }  
8 sumOf()  
9 sumOf(numbers:42,597,12)
```

实验

写一个计算它接受到参数的平均数的函数

函数可以内嵌。内嵌的函数可以访问外部函数里的变量。你可以通过使用内嵌函数来组织代码，以避免某个函数太长或者太过复杂。

```
1 funcreturnFifteen()->Int{  
2     vary=10  
3     funcadd(){  
4         y+=5  
5     }  
6     add()  
7     returny  
8 }  
9 returnFifteen()
```

函数是一等类型，这意味着函数可以把函数作为值来返回。

```
1 funcmakeIncrementer()->((Int)->Int){  
2     funcaddOne(number:Int)->Int{  
3         return1+number  
4     }  
5     returnaddOne  
6 }  
7 varincrement=makeIncrementer()  
8 increment(7)
```

函数也可以把另外一个函数作为其自身的参数。

```
1 funchasAnyMatches(list:[Int],condition:(Int)->Bool)->Bool{  
2     foriteminlist{  
3         ifcondition(item){  
4             returntrue  
5         }  
6     }  
7     returnfalse  
8 }  
9 funclessThanTen(number:Int)->Bool{  
10    returnnumber<10  
11 }  
12 varnumbers=[20,19,7,12]  
13 hasAnyMatches(list:numbers,condition:lessThanTen)
```

函数其实就是闭包的一种特殊形式：一段可以被随后调用的代码块。闭包中的代码可以访问其生效范围内的变量和函数，就算是闭包在它声明的范围之外被执行——你已经在内嵌函数的栗子上感受过了。你可以使用花括号（{}）括起一个没有名字的闭包。在闭包中使用 in 来分隔实际参数和返回类型。

```
1 numbers.map({  
2     (number:Int)->Intin  
3     letresult=3*number  
4     returnresult  
5 })
```

实验

重写这个闭包来为所有奇数返回零。

你有更多的选择来把闭包写的更加简洁。当一个闭包的类型已经可知，比如说某个委托的回调，你可以去掉它的参数类型，它的返回类型，或者都去掉。

单语句闭包隐式地返回语句执行的结果。

```
1 let mappedNumbers=numbers.map({number in 3*number})
2 print(mappedNumbers)
```

你可以调用参数通过数字而非名字——这个特性在非常简短的闭包当中尤其有用。当一个闭包作为函数最后一个参数出入时，可以直接跟在圆括号后边。如果闭包是函数的唯一参数，你可以去掉圆括号直接写闭包。

```
1 let sortedNumbers=numbers.sorted{$0>$1}
2 print(sortedNumbers)
```

对象和类

通过在 class 后接类名称来创建一个类。在类里边声明属性与声明常量或者变量的方法是相同的，唯一的区别的它们在类环境下。同样的，方法和函数的声明也是相同的写法。

```
1 class Shape{
2     var numberOfSides=0
3     func simpleDescription()->String{
4         return "A shape with \(numberOfSides) sides."
5     }
6 }
```

实验

使用 let 添加一个常量属性，添加另一个方法接收一个参数。

通过在类名字后边添加一对圆括号来创建一个类的实例。使用点语法来访问实例里的属性和方法。

```
1 var shape=Shape()
2 shape.numberOfSides=7
3 var shapeDescription=shape.simpleDescription()
```

这个 Shape 类的版本缺失了一些重要的东西：一个用在创建实例的时候来设置类的初始化器。使用 init 来创建一个初始化器。

```
1 class NamedShape{
2     var numberOfSides:Int=0
3     var name:String
4     init(name:String){
5         self.name=name
6     }
7     func simpleDescription()->String{
8         return "A shape with \(numberOfSides) sides."
9     }
10 }
11
12
```

注意使用 `self` 来区分 `name` 属性还是初始化器里的 `name` 参数。创建类实例的时候给初始化器传参就好像是调用方法一样。每一个属性都需要赋值——要么在声明的时候（比如说 `numberOfSides`），要么就要在初始化器里赋值（比如说 `name`）。使用 `deinit` 来创建一个反初始化器，如果你需要在释放对象之前执行一些清理工作的话。

声明子类就在它名字后面跟上父类的名字，用冒号分隔。创建类不需要从什么标准根类来继承，所以你可以按需包含或者去掉父类声明。

子类的方法如果要重写父类的实现，则需要使用 `override`——不使用 `override` 关键字来标记则会导致编译器报错。编译器同样也会检测使用 `override` 的方法是否存在于父类当中。

```
1 class Square: NamedShape{  
2     var sideLength: Double  
3     init(sideLength: Double, name: String){  
4         self.sideLength = sideLength  
5         super.init(name: name)  
6         numberOfSides = 4  
7     }  
8     func area() -> Double{  
9         return sideLength * sideLength  
10    }  
11    override func simpleDescription() -> String{  
12        return "A square with sides of length \(sideLength)."  
13    }  
14 }  
15 let test = Square(sideLength: 5.2, name: "my test square")  
16 test.area()  
17 test.simpleDescription()  
18  
19  
20
```

实验

创建另一个 `NamedShape` 的子类，名为 `Circle`，它接收半径和名称作为其初始化器的参数。并在 `Circle` 类里实现一个 `area()` 和一个 `simpleDescription()` 方法。

除了存储属性，你也可以拥有带有 `getter` 和 `setter` 的计算属性。

```

1 class EquilateralTriangle: NamedShape{
2     var sideLength: Double = 0.0
3     init(sideLength: Double, name: String) {
4         self.sideLength = sideLength
5         super.init(name: name)
6         numberOfSides = 3
7     }
8     var perimeter: Double {
9         get {
10            return 3.0 * sideLength
11        }
12        set {
13            sideLength = newValue / 3.0
14        }
15    }
16    override func simpleDescription() -> String {
17        return "An equilateral triangle with sides of length \(sideLength)."
18    }
19}
20 var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
21 print(triangle.perimeter)
22 triangle.perimeter = 9.9
23 print(triangle.sideLength)
24
25
26

```

在 `perimeter` 的 `setter` 中，新值被隐式地命名为 `newValue`。你可以提供一个显式的名称放在 `set` 后边的圆括号里。

注意 `EquilateralTriangle` 类的初始化器有三个不同的步骤：

1. 设定子类声明的属性的值；
2. 调用父类的初始化器；
3. 改变父类定义的属性中的值，以及其他任何使用方法，`getter` 或者 `setter` 等需要在这时候完成的内容。

如果你不需要计算属性但仍然需要在设置一个新值的前后执行代码，使用 `willSet` 和 `didSet`。比如说，下面的类确保三角形的边长始终和正方形的边长相同。

```

1 class TriangleAndSquare {
2     var triangle: EquilateralTriangle {
3         willSet {
4             square.sideLength = newValue.sideLength
5         }
6     }
7     var square: Square {
8         willSet {
9             triangle.sideLength = newValue.sideLength
10    }
11   }
12   init(size: Double, name: String) {
13       square = Square(sideLength: size, name: name)
14       triangle = EquilateralTriangle(sideLength: size, name: name)
15   }
16 }
17 var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
18 print(triangleAndSquare.square.sideLength)
19 print(triangleAndSquare.triangle.sideLength)
20 triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
21 print(triangleAndSquare.triangle.sideLength)

```

当你操作可选项的值的时候，你可以在可选项前边使用 ?比如方法，属性和下标脚本。如果 ?前的值是 nil，那 ?后的所有内容都会被忽略并且整个表达式的值都是 nil。否则，可选项的值将被展开，然后 ?后边的代码根据展开的值执行。在这两种情况当中，表达式的值是一个可选的值。

```
1 let optionalSquare:Square?=Square(sideLength:2.5,name:"optional square")
2 let sideLength=optionalSquare?.sideLength
```

枚举和结构体

使用 enum 来创建枚举，类似于类和其他所有的命名类型，枚举也能够包含方法。

```
1 enumRank: Int{
2     case ace=1
3     case two,three,four,five,six,seven,eight,nine,ten
4     case jack,queen,king
5     func simpleDescription() -> String{
6         switch self{
7             case .ace:
8                 return "ace"
9             case .jack:
10                return "jack"
11            case .queen:
12                return "queen"
13            case .king:
14                return "king"
15            default:
16                return String(self.rawValue)
17         }
18     }
19 }
20 let ace=Rank.ace
21 let aceRawValue=ace.rawValue
```

实验

写一个函数通过对比它们的原始值来对比两个 Rank 值

默认情况下，Swift 从零开始给原始值赋值后边递增，但你可以通过指定特定的值来改变这一行为。在上边的栗子当中，原始值的枚举类型是 Int，所以你只需要确定第一个原始值。剩下的原始值是按照顺序指定的。你同样可以使用字符串或者浮点数作为枚举的原始值。使用 rawValue 属性来访问枚举成员的原始值。

使用 init?(rawValue:) 初始化器来从一个原始值创建枚举的实例。

```
1 if let convertedRank=Rank(rawValue:3){
2     let threeDescription=convertedRank.simpleDescription()
3 }
```

枚举成员的值是实际的值，不是原始值的另一种写法。事实上，在这种情况下没有一个有意义的原始值，你根本没有必要提供一个。

```
1 enumSuit{
2     case spades, hearts, diamonds, clubs
3     func simpleDescription() -> String{
4         switch self{
5             case .spades:
6                 return "spades"
7             case .hearts:
8                 return "hearts"
9             case .diamonds:
10                return "diamonds"
11            case .clubs:
12                return "clubs"
13        }
14    }
15 }
16 let hearts = Suit.hearts
17 let heartsDescription = hearts.simpleDescription()
```

实验

添加一个 color()方法到 Suit，为黑桃和梅花返回“black”，为红桃和方片返回“red”。

注意有两种方法可以调用枚举的 hearts 成员：当给 hearts 指定一个常量时，枚举成员 Suit.Hearts 会被以全名的方式调用因为常量并没有显式地指定类型。在 Switch 语句当中，枚举成员可以通过缩写的方式 .hearts 被调用，因为 self 已经明确了是 suit。你可以在任何值的类型已经明确的场景下使用使用缩写。

如果枚举拥有原始值，这些值在声明时确定，就是说每一个这个枚举的实例都将拥有相同的原始值。另一个选择是让 case 与值关联——这些值在你初始化实例的时候确定，这样它们就可以在每个实例中不同了。比如说，考虑在服务器上请求日出和日落时间的 case，服务器要么返回请求的信息，要么返回错误信息。

```
1 enum ServerResponse{
2     case result(String, String)
3     case failure(String)
4 }
5 let success = ServerResponse(result("6:00 am", "8:09 pm"))
6 let failure = ServerResponse.failure("Out of cheese.")
7 switch success{
8     case let .result(sunrise, sunset):
9         print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
10    case let .failure(message):
11        print("Failure... \(message)")
12 }
13
14
```

实验

添加第三个 case 到 ServerResponse 和 switch。

注意现在日出和日落时间是从 ServerResponse 值中以 switch case 匹配的形式取出的。

使用 struct 来创建结构体。结构体提供很多类似与类的行为，包括方法和初始化器。其中最重要的一点区别就是结构体总是会在传递的时候拷贝其自身，而类则会传递引用。

```
1 structCard{
2   varrank:Rank
3   varsuit:Suit
4   funcsimpleDescription()->String{
5     return"The \$(rank.simpleDescription()) of \$(suit.simpleDescription())"
6   }
7 }
8 letthreeOfSpades=Card(rank:.three,suit:.spades)
9 letthreeOfSpadesDescription=threeOfSpades.simpleDescription()
```

实验

给 Card 添加一个方法来创建一整副扑克牌，并且把每张牌的 rank 和 suit 对应起来。

协议和扩展

使用 protocol 来声明协议。

```
1 protocolExampleProtocol{
2   varsimpleDescription: String{get}
3   mutatingfuncadjust()
4 }
```

类，枚举以及结构体都兼容协议。

```
1 classSimpleClass: ExampleProtocol{
2   varsimpleDescription:String="A very simple class."
3   varanotherProperty:Int=69105
4   funcadjust(){
5     simpleDescription+=" Now 100% adjusted."
6   }
7 }
8 vara=SimpleClass()
9 a.adjust()
10 letaDescription=a.simpleDescription
11 structSimpleStructure: ExampleProtocol{
12   varsimpleDescription:String="A simple structure"
13   mutatingfuncadjust(){
14     simpleDescription+=" (adjusted)"
15   }
16 }
17 varb=SimpleStructure()
18 b.adjust()
19 letbDescription=b.simpleDescription
20
```

实验

写一个枚举来遵循这个协议。

注意使用 mutating 关键字来声明在 SimpleStructure 中使方法可以修改结构体。在 SimpleClass 中则不需要这样声明，因为类里的方法总是可以修改其自身属性的。

使用 extension 来给现存的类型增加功能，比如说新的方法和计算属性。你可以使用扩展来使协议来别处定义的类型，或者你导入的其他库或框架。

```
1 extensionInt: ExampleProtocol{
2     var simpleDescription: String{
3         return "The number \u002f(self)"
4     }
5     mutating func adjust(){
6         self += 42
7     }
8 }
9 print(7.simpleDescription)

1 let protocolValue: ExampleProtocol = a
2 print(protocolValue.simpleDescription)
3 // print(protocolValue.anotherProperty) // Uncomment to see the error
```

尽管变量 `protocolValue` 有 `SimpleClass` 的运行时类型，但编译器还是把它看做 `ExampleProtocol`。这意味着你不能访问类在这个协议中扩展的方法或者属性。

错误处理

你可以用任何遵循 `Error` 协议的类型来表示错误。

```
1 enum PrinterError: Error{
2     case outOfPaper
3     case noToner
4     case onFire
5 }
```

使用 `throw` 来抛出一个错误并且用 `throws` 来标记一个可以抛出错误的函数。如果你在函数里抛出一个错误，函数会立即返回并且调用函数的代码会处理错误。

```
1 func send(job: Int, toPrinter printerName: String) throws -> String{
2     if printerName == "Never Has Toner"{
3         throw PrinterError.noToner
4     }
5     return "Job sent"
6 }
```

有好几种方法来处理错误。一种是使用 `do-catch`。在 `do` 代码块里，你用 `try` 来在能抛出错误的函数前标记。在 `catch` 代码块，错误会自动赋予名字 `error`，如果你不给定其他名字的话。

```
1 do{
2     let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
3     print(printerResponse)
4 } catch {
5     print(error)
6 }
```

实验

改变 `printer` 的名字为 "Never Has Toner"，好让 `send(job:toPrinter:)` 函数抛出一个错误。

你可以提供多个 `catch` 代码块来处理特定的错误。你可以在 `catch` 后写一个模式，用法和 `switch` 语句里的 `case` 一样。

```
1 do{
2   let printerResponse=trysend(job:1440,toPrinter:"Gutenberg")
3   print(printerResponse)
4 }catchPrinterError.onFire{
5   print("I'll just put this over here, with the rest of the fire." )
6 }catchletprinterErrorasPrinterError{
7   print("Printer error: \printerError")
8 }catch{
9   print(error)
10 }
```

实验

添加一些代码来在 do 代码块里抛出一个错误。你要抛出什么样的错误才能让第一个 catch 代码块处理到？第二个，第三个呢？

另一种处理错误的方法是使用 try? 来转换结果为可选项。如果函数抛出了错误，那么错误被忽略并且结果为 nil 。否则，结果是一个包含了函数返回值的可选项。

```
1 let printerSuccess=try?send(job:1884,toPrinter:"Mergenthaler")
2 let printerFailure=try?send(job:1885,toPrinter:"Never Has Toner")
```

1.

使用 defer 来写在函数返回后也会被执行的代码块，无论是否错误被抛出。你甚至可以在没有错误处理的时候使用 defer ，来简化需要在多处地方返回的函数。

```
1 var fridgelsOpen=false
2 let fridgeContent=["milk","eggs","leftovers"]
3 func fridgeContains(_ food:String)->Bool{
4   fridgelsOpen=true
5   defer{
6     fridgelsOpen=false
7   }
8   let result=fridgeContent.contains(food)
9   return result
10 }
11 fridgeContains("banana")
12 print(fridgelsOpen)
13
14
```

泛型

把名字写在尖括号里来创建一个泛型方法或者类型。

```
1 func makeArray<Item>(repeating item:Item,numberoftimes:Int)->[Item]{
2   var result=[Item]()
3   for _ in 0..<numberoftimes{
4     result.append(item)
5   }
6   return result
7 }
8 makeArray(repeating:"knock",numberoftimes:4)
```

你可以从函数和方法同时还有类，枚举以及结构体创建泛型。

```
1 // Reimplement the Swift standard library's optional type
2 enumOptionalValue<Wrapped>{
3     casenone
4     casesome(Wrapped)
5 }
6 varpossibleInteger:OptionalValue<Int>=.none
7 possibleInteger=.some(100)
```

在类型名称后紧接 `where` 来明确一系列需求——比如说，来要求类型实现一个协议，要求两个类型必须相同，或者要求类必须继承自特定的父类。

```
1 funcanyCommonElements<T:Sequence,U:Sequence>(_lhs:T,_rhs:U)->Bool
2 where T.Iterator.Element:Equatable,T.Iterator.Element==U.Iterator.Element{
3     forlhsIteminlhs{
4         forrhsIteminrhs{
5             iflhsItem==rhsItem{
6                 returntrue
7             }
8         }
9     }
10    returnfalse
11 }
12 anyCommonElements([1,2,3],[3])
```

实验

修改 `anyCommonElements(_:_)` 函数来返回一个 两个数组中共有元素 的数组。

写 `<T:Equatable>` 和 `<T where T:Equatable>` 是同一回事。

基础内容

 cnswift.org/the-basics

Swift 是一门全新的用于开发 iOS, OS X 以及 watchOS 应用的编程语言。不过，如果你有 C 或者 Objective-C 语言开发经验的话，Swift 的许多地方都会让你感到熟悉。

Swift 为所有 C 和 Objective-C 的类型提供了自己的版本，包括整型值的 Int，浮点数值的 Double 和 Float，布尔量值的 Bool，字符串值的 String。如同集合类型中描述的那样，Swift 同样也为三个主要的集合类型提供了更高效的版本，Array，Set 和 Dictionary。

和 C 一样，Swift 用变量存储和调用值，通过变量名来做区分。Swift 中也大量采用了值不可变的变量。它们就是所谓的常量，但是它们比 C 中的常量更加给力。当你所处理的值不需要更改时，使用常量会让你的代码更加安全、简洁地表达你的意图。

除了我们熟悉的类型以外，Swift 还增加了 Objective-C 中没有的类型，比如元组。元组允许你来创建和传递一组数据。你可以利用元组在一个函数中以单个复合值的形式返回多个值。

Swift 还增加了可选项，用来处理没有值的情况。可选项意味着要么“这里有一个值，它等于 x”要么“这里根本没有值”。可选项类似于 Objective-C 中的 nil 指针，但是不只是类，可选项也可以用在所有的类型上。可选项比 Objective-C 中的 nil 指针更安全、更易读，他也是 Swift 语言中许多重要功能的核心。

可选项充分证明了 Swift 是一门类型安全的语言。Swift 帮助你明确代码可以操作值的类型。如果你的一段代码预期得到一个 String，类型会安全地阻止你不小心传入 Int。在开发过程中，这个限制能帮助你在开发过程中更早地发现并修复错误。

常量和变量

常量和变量把名字（例如 maximumNumberOfLoginAttempts 或者 welcomeMessage）和一个特定类型的值（例如数字 10 或者字符串 “Hello”）关联起来。常量的值一旦设置好便不能再被更改，然而变量可以在将来被设置为不同的值。

声明常量和变量

常量和变量必须在使用前被声明，使用关键字 let 来声明常量，使用关键字 var 来声明变量。这里有一个如何利用常量和变量记录用户登录次数的栗子：

```
1 let maximumNumberOfLoginAttempts=10  
2 var currentLoginAttempt=0
```

这段代码可以读作：

“声明一个叫做 maximumNumberOfLoginAttempts 的新常量，并设置值为 10。然后声明一个叫做 currentLoginAttempt 的新变量，并且给他一个初始值 0。”

在这个栗子中，登录次数允许的最大值被声明为一个常量，因为最大值永远不会更改。当前尝试登录的次数被声明为一个变量，因为这个值在每次登录尝试失败之后会递增。

你可以在一行中声明多个变量或常量，用逗号分隔：

```
1 varx=0.0,y=0.0,z=0.0
```

注意

在你的代码中，如果存储的值不会改变，请用 let 关键字将之声明为一个常量。只有储存会改变的值时才使用变量。

类型标注

你可以在声明一个变量或常量的时候提供 **类型标注**，来明确变量或常量能够储存值的类型。添加类型标注的方法是在变量或常量的名字后边加一个冒号，再跟一个空格，最后加上要使用的类型名称。

下面这个栗子给一个叫做 welcomeMessage 的变量添加了类型标注，明确这个变量可以存储 String 类型的值。

```
1 varwelcomeMessage:String
```

声明中的冒号的意思是“是...类型”，所以上面的代码可以读作：

“声明一个叫做 welcomeMessage 的变量，他的类型是 String ”

我们说“类型是 String ”就意味着“可以存储任何 String 值”。也可以理解为“这类东西”（或者“这种东西”）可以被存储进去。

现在这个 welcomeMessage 变量就可以被设置到任何字符串中而不会报错了：

```
1 welcomeMessage="Hello"
```

你可以在一行中定义多个相关的变量为相同的类型，用逗号分隔，只要在最后的变量名字后边加上类型标注。

```
1 varred,green,blue:Double
```

注意

实际上，你并不需要经常使用类型标注。如果你在定义一个常量或者变量的时候就给他设定一个初始值，那么 Swift 就像**类型安全和类型推断**中描述的那样，几乎都可以推断出这个常量或变量的类型。在上面 welcomeMessage 的栗子中，没有提供初始值，所以 welcomeMessage 这个变量使用了类型标注来明确它的类型而不是通过初始值的类型推断出来的。

命名常量和变量

常量和变量的名字几乎可以使用任何字符，甚至包括 Unicode 字符：

```
1 letπ=3.14159
2 let你好="你好世界"
3 let🐮="dogcow"
```

常量和变量的名字不能包含空白字符、数学符号、箭头、保留的（或者无效的）Unicode 码

位、连线和制表符。也不能以数字开头，尽管数字几乎可以使用在名字其他的任何地方。

一旦你声明了一个确定类型的常量或者变量，就不能使用相同的名字再次进行声明，也不能让它改存其他类型的值。常量和变量之间也不能互换。

注意

如果你需要使用 Swift 保留的关键字来给常量或变量命名，可以使用反引号（`）包围它来作为名称。总之，除非别无选择，避免使用关键字作为名字除非你确实别无选择。

你可以把现有变量的值更改为其他相同类型的值。在这个栗子中 friendlyWelcome 的值从 “Hello!” 改变为 “Bonjour!”

```
1 varfriendlyWelcome="Hello!"  
2 friendlyWelcome="Bonjour!"  
3 // friendlyWelcome 现在是 "Bonjour!"
```

不同于变量，常量的值一旦设定则不能再被改变。尝试这么做将会在你代码编译时导致报错：

```
1 letlanguageName="Swift"  
2 languageName="Swift++"  
3 // this is a compile-time error - languageName cannot be changed
```

输出常量和变量

你可以使用 `print(_:separator:terminator:)` 函数来打印当前常量和变量中的值。

```
1 print(friendlyWelcome)  
2 // 输出 "Bonjour!"
```

`print(_:separator:terminator:)` 是一个用来把一个或者多个值用合适的方式输出的全局函数。比如说，在 Xcode 中 `print(_:separator:terminator:)` 函数输出的内容会显示在 Xcode 的 “console” 面板上。`separator` 和 `terminator` 形式参数有默认值，所以你可以在调用这个函数的时候忽略它们。默认来说，函数通过在行末尾添加换行符来结束输出。要想输出不带换行符的值，那就传一个空的换行符作为结束——比如说，`print(someValue,terminator:"")`。更多关于带有默认值的形式参数信息，见[默认形式参数值](#)。

Swift 使用字符串插值的方式来把常量名或者变量名当做占位符加入到更长的字符串中，然后让 Swift 用常量或变量的当前值替换这些占位符。将常量或变量名放入圆括号中并在括号前使用反斜杠将其转义：

```
1 print("The current value of friendlyWelcome is \(friendlyWelcome)")  
2 // 输出 "The current value of friendlyWelcome is Bonjour!"
```

注意

[字符串插值](#)中描述了你可以使用字符串插值的所有选项。

注释

使用注释来将不需要执行的文本放入的代码当中，作为标记或者你自己的提醒。当 Swift 编译器在编译代码的时候会忽略掉你的注释。

Swift 中的注释和 C 的注释基本相同。单行注释用两个斜杠开头 (//) :

```
1 // 这是一个注释
```

多行的注释以一个斜杠加一个星号开头 (/*) , 以一个星号加斜杠结尾(*/)。

```
1 /* this is also a comment,
2 but written over multiple lines */
```

和 C 中的多行注释不同的是， Swift 语言中的多行的注释可以内嵌在其它的多行注释之中，你可以在多行注释中先开启一个注释块，接着再开启另一个注释块。然后关闭第二个注释块，再关闭第一个注释块。

```
1 /* 这是第一个多行注释的开头
2 /* 这是第二个嵌套在内的注释块 */
3 这是第一个注释块的结尾*/
```

内嵌多行注释，可以便捷地注释掉一大段代码块，即使这段代码块中已经有了多行注释。

分号

和许多其他的语言不同，Swift 并不要求你在每一句代码结尾写分号 (;) ，当然如果你想写的话也没问题。总之，如果你想在一行里写多句代码，分号还是需要的。

```
1 letcat="🐱";print(cat)
2 // 输出 "🐱"
```

整数

整数就是没有小数部分的数字，比如 42 和 -23 。整数可以是有符号（正，零或者负），或者无符号（正数或零）。

Swift 提供了 8 , 16 , 32 和 64 位编码的有符号和无符号整数，这些整数类型的命名方式和 C 相似，例如 8 位无符号整数的类型是 UInt8 , 32 位有符号整数的类型是 Int32 。与 Swift 中的其他类型相同，这些整数类型也用开头大写命名法。

整数范围

你可以通过 min 和 max 属性来访问每个整数类型的最小值和最大值：

```
1 let minValue=UInt8.min// 最小值是 0, 值的类型是 UInt8
2 let maxValue=UInt8.max// 最大值是 255, 值得类型是 UInt8
```

这些属性的值都是自适应大小的数字类型（比如说上边栗子里的 UInt8 ）并且因此可以在表达式中与在其他相同类型值同用。

Int

在大多数情况下，你不需要在你的代码中为整数设置一个特定的长度。Swift 提供了一个额外的整数类型：`Int`，它拥有与当前平台的原生字相同的长度。

- 在32位平台上，`Int` 的长度和 `Int32` 相同。
- 在64位平台上，`Int` 的长度和 `Int64` 相同。

除非你需操作特定长度的整数，否则请尽量在代码中使用 `Int` 作为你的整数的值类型。这样能提高代码的统一性和兼容性，即使在 32 位的平台上，`Int` 也可以存 -2,147,483,648 到 2,147,483,647 之间的任意值，对于大多数整数区间来说完全够用了。

UInt

Swift 也提供了一种无符号的整数类型，`UInt`，它和当前平台的原生字长度相同。

- 在32位平台上，`UInt` 长度和 `UInt32` 长度相同。
- 在64位平台上，`UInt` 长度和 `UInt64` 长度相同。

注意

只在的确需要存储一个和当前平台原生字长度相同的无符号整数的时候才使用 `UInt`。其他情况下，推荐使用 `Int`，即使已经知道存储的值都是非负的。如同 [类型安全和类型推断](#) 中描述的那样，统一使用 `Int` 会提高代码的兼容性，同时可以避免不同数字类型之间的转换问题，也符合整数的类型推断。

浮点数

浮点数是有小数的数字，比如 `3.14159`, `0.1`, 和 `-273.15`。

浮点类型相比整数类型来说能表示更大范围的值，可以存储比 `Int` 类型更大或者更小的数字。Swift 提供了两种有符号的浮点数类型。

- `Double` 代表 64 位的浮点数。
- `Float` 代表 32 位的浮点数。

注意

`Double` 有至少 15 位数字的精度，而 `Float` 的精度只有 6 位。具体使用哪种浮点类型取决于你代码需要处理的值范围。在两种类型都可以的情况下，推荐使用 `Double` 类型。

类型安全和类型推断

Swift 是一门类型安全的语言。类型安全的语言可以让你清楚地知道代码可以处理的值的类型。如果你的一部分代码期望获得 `String`，你就不能错误的传给它一个 `Int`。

因为 Swift 是类型安全的，他在编译代码的时候会进行类型检查，任何不匹配的类型都会被标记为错误。这会帮助你在开发阶段更早的发现并修复错误。

当你操作不同类型的值时，类型检查能帮助你避免错误。当然，这并不意味着你得为每一个常量或变量声明一个特定的类型。如果你没有为所需要的值进行类型声明，Swift 会使用类型推断的功能推断出合适的类型。通过检查你给变量赋的值，类型推断能够在编译阶段自动的推断出值的类型。

因为有了类型推断，Swift 和 C 以及 Objective-C 相比，只需要少量的类型声明。其实常量和变量仍然需要明确的类型，但是大部分的声明工作 Swift 会帮你做。

在你为一个变量或常量设定一个初始值的时候，类型推断就显得更加有用。它通常在你声明一个变量或常量同时设置一个初始的字面量（文本）时就已经完成。（字面量就是会直接出现在你代码中的值，比如下边代码中的 42 和 3.14159。）

举个栗子，如果你给一个新的常量设定一个 42 的字面量，而且没有说它的类型是什么，Swift 会推断这个常量的类型是 Int，因为你给这个常量初始化为一个看起来像是一个整数的数字。

```
1 letmeaningOfLife=42
2 // meaningOfLife is inferred to be of type Int
```

同样，如果你没有为一个浮点值的字面量设定类型，Swift 会推断你想创建一个 Double。

```
1 letpi=3.14159
2 // pi is inferred to be of type Double
```

Swift 在推断浮点值的时候始终会选择 Double（而不是 Float）。

如果你在一个表达式中将整数和浮点数结合起来，Double 会从内容中被推断出来。

```
1 letanotherPi=3+0.14159
2 // anotherPi is also inferred to be of type Double
```

这字面量 3 没有显式的声明它的类型，但因为后边有一个浮点类型的字面量，所以这个类型就被推断为 Double。

数值型字面量

整数型字面量可以写作：

- 一个十进制数，没有前缀
- 一个二进制数，前缀是 0b
- 一个八进制数，前缀是 0o
- 一个十六进制数，前缀是 0x

下面的这些所有整数字面量的十进制值都是 17：

```
1 letdecimalInteger=17
2 letbinaryInteger=0b10001// 17 in binary notation
3 letoctalInteger=0o21// 17 in octal notation
4 lethexadecimalInteger=0x11// 17 in hexadecimal notation
```

浮点字面量可以是十进制（没有前缀）或者是十六进制（前缀是 0x）。小数点两边必须有至少一个十进制数字（或者是十六进制的数字）。十进制的浮点字面量还有一个可选的指数，用大写或小写的 e 表示；十六进制的浮点字面量必须有指数，用大写或小写的 p 来表示。

十进制数与 exp 的指数，结果就等于基数乘以 10^{exp} ：

- 1.25e2 意味着 1.25×10^2 , 或者 125.0 .
- 1.25e-2 意味着 1.25×10^{-2} , 或者 0.0125 .

十六进制数与 exp 指数 , 结果就等于基数乘以 2^{exp} :

- 0xFp2 意味着 15×2^2 , 或者 60.0 .
- 0xFp-2 意味着 15×2^{-2} , 或者 3.75 .

下面的这些浮点字面量的值都是十进制的 12.1875 :

```
1 letdecimalDouble=12.1875
2 letexponentDouble=1.21875e1
3 lethexadecimalDouble=0xC.3p0
```

数值型字面量也可以增加额外的格式使代码更加易读。整数和浮点数都可以添加额外的零或者添加下划线来增加代码的可读性。下面的这些格式都不会影响字面量的值。

```
1 letpaddedDouble=000123.456
2 letoneMillion=1_000_000
3 letjustOverOneMillion=1_000_000.000_000_1
```

数值类型转换

通常来讲，即使我们知道代码中的整数变量和常量是非负的，我们也会使用 Int 类型。经常使用默认的整数类型可以确保你的整数常量和变量可以直接被复用并且符合整数字面量的类型推测。

只有在特殊情况下才会使用整数的其他类型，例如需要处理外部长度明确的数据或者为了优化性能、内存占用等其他必要情况。在这些情况下，使用指定长度的类型可以帮助你及时发现意外的值溢出和隐式记录正在使用数据的本质。

整数转换

不同整数的类型在变量和常量中存储的数字范围是不同的。 Int8 类型的常量或变量可以存储的数字范围是 -128~127，而 UInt8 类型的常量或者变量能存储的数字范围是 0~255。如果数字超出了常量或者变量可存储的范围，编译的时候就会报错：

```
1 letcannotBeNegative:UInt8=-1
2 // UInt8 cannot store negative numbers, and so this will report an error
3 lettooBig:Int8=Int8.max+1
4 // Int8 cannot store a number larger than its maximum value,
5 // and so this will also report an error
```

因为每个数值类型可存储的值的范围不同，你必须根据不同的情况进行数值类型的转换。这种选择性使用的方式可以避免隐式转换的错误并使你代码中的类型转换意图更加清晰。

要将一种数字类型转换成另外一种类型，你需要用当前值来初始化一个期望的类型。在下面的栗子中，常量 twoThousand 的类型是 UInt16，而常量 one 的类型是 UInt8。他们不能直接被相加在一起，因为他们的类型不同。所以，这里让 UInt16 (one) 创建一个新的 UInt16 类型并用 one 的值初始化，这样就可以在原来的地方使用了。

```
1 lettwoThousand:UInt16=2_000
2 letone:UInt8=1
3 lettwoThousandAndOne=twoThousand+UInt16(one)
```

因为加号两边的类型现在都是 UInt16，所以现在是可以相加的。输出的常量 (twoThousandAndOne) 被推断为 UInt16 类型，因为他是两个 UInt16 类型的和。

SomeType(ofInitialValue) 是调用 Swift 类型初始化器并传入一个初始值的默认方法。在语言的内部，UInt16 有一个初始化器，可以接受一个 UInt8 类型的值，所以这个初始化器可以用现有的 UInt8 来创建一个新的 UInt16。这里需要注意的是并不能传入任意类型的值，只能传入 UInt16 内部有对应初始化器的值。不过你可以扩展现有的类型来让它可以接收其他类型的值（包括自定义类型），请参考扩展。

整数和浮点数转换

整数和浮点数类型的转换必须显式地指定类型：

```
1 letthree=3
2 letpointOneFourOneFiveNine=0.14159
3 letpi=Double(three)+pointOneFourOneFiveNine
4 // pi equals 3.14159, and is inferred to be of type Double
```

在这里，常量 three 的值被用来创建一个类型为 Double 的新的值，所以加号两边的值的类型是相同的。没有这个转换，加法就无法进行。

浮点转换为整数也必须显式地指定类型。一个整数类型可以用一个 Double 或者 Float 值初始化。

```
1 letintegerPi=Int(pi)
2 // integerPi equals 3, and is inferred to be of type Int
```

在用浮点数初始化一个新的整数类型的时候，数值会被截断。也就是说 4.75 会变成 4，-3.9 会变为 -3。

注意

结合数字常量和变量的规则与结合数字字面量的规则不同，字面量 3 可以直接和字面量 0.14159 相加，因为数字字面量本身没有明确的类型。它们的类型只有在编译器需要计算的时候才会被推测出来。

类型别名

类型别名可以为已经存在的类型定义了一个新的可选名字。用 typealias 关键字定义类型别名。

当你根据上下文的语境想要给类型一个更有意义的名字的时候，类型别名会非常高效，例如处理外部资源中特定长度的数据时：

```
1 typealiasAudioSample=UInt16
```

一旦为类型创建了一个别名，你就可以在任何使用原始名字的地方使用这个别名。

```
1 var maxAmplitudeFound = AudioSample.min
2 // maxAmplitudeFound is now 0
```

在这个栗子中，`AudioSample` 就是 `UInt16` 的别名，因为这个别名的存在，我们调用 `AudioSample.min` 其实就是在调用 `Int16.min`，在这里变量 `maxAmplitudeFound` 被提供了一个初始值 0。

布尔值

Swift 有一个基础的布尔量类型，就是 `Bool`，布尔量被作为逻辑值来引用，因为他的值只能是真或者假。Swift 为布尔量提供了两个常量值，`true` 和 `false`。

```
1 let orangesAreOrange = true
2 let turnipsAreDelicious = false
```

上面的两个类型 `orangesAreOrange` 和 `turnipsAreDelicious`，被推断为 `Bool`，因为它们使用布尔量来初始化。对于上文中的 `Int` 和 `Double`，当你在创建他们的的时候设置为 `true` 或 `false`，那么就不必给这个常量或者变量声明为 `Bool` 类型。初始化常量或者变量的时候，如果值的类型已知，类型推断会把 Swift 代码变的更加整洁和易读。

当你处理条件语句的时候例如 `if` 语句时，布尔值就会变得非常有用：

```
1 if turnipsAreDelicious {
2     print("Mmm, tasty turnips!")
3 } else {
4     print("Eww, turnips are horrible.")
5 }
6 // prints "Eww, turnips are horrible."
```

关于条件判断语句例如 `if` 语句，请参考控制流。

Swift 的类型安全机制会阻止你用一个非布尔量的值替换掉 `Bool`。下面的栗子中报告了一个发生在编译时的错误：

```
1 let i = 1
2 if i {
3     // this example will not compile, and will report an error
4 }
```

然而，下边的这个例子就是可行的：

```
1 let i = 1
2 if i == 1 {
3     // this example will compile successfully
4 }
```

这里 `i == 1` 的比较结果是一个 `Bool` 类型，所以第二个栗子可以通过类型检查。类似 `i == 1` 这样的比较请参考基本运算符。

与 Swift 中其他的类型安全示例一样，这个方法可以避免错误的发生并确保这块代码的意图清晰。

元组

元组把多个值合并成单一的复合型的值。元组内的值可以是任何类型，而且可以不必是同一类型。

在下面的示例中，(404,"Not Found")是一个描述了HTTP状态代码的元组。HTTP状态代码是当你请求网页的时候web服务器返回的一个特殊值。当你请求不存在的网页时，就会返回404NotFound

```
1 let http404Error=(404,"Not Found")
2 // http404Error is of type (Int, String), and equals (404, "Not Found")
```

(404,"Not Found")元组把一个Int和一个String组合起来表示HTTP状态代码的两种不同的值：数字和人类可读的描述。他可以被描述为“一个类型为(Int, String)的元组”

任何类型的排列都可以被用来创建一个元组，他可以包含任意多的类型。例如(Int, Int, Int)或者(String, Bool)，实际上，任何类型的组合都是可以的。

你也可以将一个元组的内容分解成单独的常量或变量，这样你就可以正常的使用它们了：

```
1 let(statusCode,statusMessage)=http404Error
2 print("The status code is \$(statusCode)")
3 // prints "The status code is 404"
4 print("The status message is \$(statusMessage)")
5 // prints "The status message is Not Found"
```

当你分解元组的时候，如果只需要使用其中的一部分数据，不需要的数据可以用下滑线(_)代替：

```
1 let(justTheStatusCode,_)=http404Error
2 print("The status code is \$(justTheStatusCode)")
3 // prints "The status code is 404"
```

另外一种方法就是利用从零开始的索引数字访问元组中的单独元素：

```
1 print("The status code is \$(http404Error.0)")
2 // prints "The status code is 404"
3 print("The status message is \$(http404Error.1)")
4 // prints "The status message is Not Found"
```

你可以在定义元组的时候给其中的单个元素命名：

```
1 let http200Status=(statusCode:200,description:"OK")
```

在命名之后，你就可以通过访问名字来获取元素的值了：

```
1 print("The status code is \$(http200Status.statusCode)")
2 // prints "The status code is 200"
3 print("The status message is \$(http200Status.description)")
4 // prints "The status message is OK"
```

作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个(Int, String)元组来描述是否获取成功。相比只能返回一个类型的值，元组能包含两个不同类型值，他可以让函数的返回信息更有用。更多内容请参考[多返回值的函数](#)。

注意

元组在临时的值组合中很有用，但是它们不适合创建复杂的数据结构。如果你的数据结构超出了临时使用的范围，那么请建立一个类或结构体来代替元组。更多信息请参考类和结构体。

可选项

可以利用可选项来处理值可能缺失的情况。可选项意味着：

这里有一个值，他等于 x

或者

这里根本没有值

注意

在 C 和 Objective-C 中，没有可选项的概念。在 Objective-C 中有一个近似的特性，一个方法可以返回一个对象或者返回 nil。nil 的意思是“缺少一个可用对象”。然而，他只能用在对象上，却不能作用在结构体，基础的 C 类型和枚举值上。对于这些类型，Objective-C 会返回一个特殊的值（例如 `NSNotFound`）来表示值的缺失。这种方法是建立在假设调用者知道这个特殊的值并记得去检查他。然而，Swift 中的可选项就可以让你知道任何类型的值的缺失，他并不需要一个特殊的值。

下面的栗子演示了可选项如何作用于值的缺失，Swift 的 Int 类型中有一个初始化器，可以将 String 值转换为一个 Int 值。然而并不是所有的字符串都可以转换成整数。字符串 “123” 可以被转换为数字值 123，但是字符串 "hello, world" 就显然不能转换为一个数字值。

在下面的栗子中，试图利用初始化器将一个 String 转换为 Int：

```
1 let possibleNumber = "123"
2 let convertedNumber = Int(possibleNumber)
3 // convertedNumber is inferred to be of type "Int?", or "optional Int"
```

因为这个初始化器可能会失败，所以他会返回一个可选的 Int，而不是 Int。可选的 Int 写做 `Int?`，而不是 `Int`。问号明确了它储存的值是一个可选项，意思就是说它可能包含某些 Int 值，或者可能根本不包含值。（他不能包含其他的值，例如 Bool 值或者 String 值。它要么是 Int 要么什么都没有。）

nil

你可以通过给可选变量赋值一个 nil 来将之设置为没有值：

```
1 var serverResponseCode: Int? = 404
2 // serverResponseCode contains an actual Int value of 404
3 serverResponseCode = nil
4 // serverResponseCode now contains no value
```

注意

nil 不能用于非可选的常量或者变量，如果你的代码中变量或常量需要作用于特定条件下的值缺失，可以给他声明为相应类型的可选项。

如果你定义的可选变量没有提供一个默认值，变量会被自动设置成 nil。

```
1 var surveyAnswer:String?  
2 // surveyAnswer is automatically set to nil
```

注意

Swift 中的 nil 和 Objective-C 中的 nil 不同，在 Objective-C 中 nil 是一个指向不存在对象的指针。在 Swift 中，nil 不是指针，他是值缺失的一种特殊类型，任何类型的可选项都可以设置成 nil 而不仅仅是对象类型。

If 语句以及强制展开

你可以利用 if 语句通过比较 nil 来判断一个可选中是否包含值。利用相等运算符（==）和不等运算符（!=）。

如果一个可选有值，他就“不等于” nil：

```
1 if convertedNumber != nil{  
2     print("convertedNumber contains some integer value.")  
3 }  
4 // prints "convertedNumber contains some integer value."
```

一旦你确定可选中包含值，你可以在可选的名字后面加一个感叹号（!）来获取值，感叹号的意思就是说“我知道这个可选项里边有值，展开吧。”这就是所谓的可选值的强制展开。

```
1 if convertedNumber != nil{  
2     print("convertedNumber has an integer value of \(convertedNumber!).")  
3 }  
4 // prints "convertedNumber has an integer value of 123."
```

如需了解更多有关 if 语句的内容，请参考[控制流](#)。

注意

使用 ! 来获取一个不存在的可选值会导致运行错误，在使用 ! 强制展开之前必须确保可选项中包含一个非 nil 的值。

可选项绑定

可以使用可选项绑定来判断可选项是否包含值，如果包含就把值赋给一个临时的常量或者变量。可选绑定可以与 if 和 while 的语句使用来检查可选项内部的值，并赋值给一个变量或常量。if 和 while 语句的更多详细描述，请参考[控制流](#)。

在 if 语句中，这样书写可选绑定：

```
1 if let constantName = someOptional{  
2     statements  
3 }
```

你可以像上面这样使用可选绑定而不是强制展开来重写 `possibleNumber` 这个例子：

```
1 if let actualNumber = Int(possibleNumber){  
2     print("\(possibleNumber)' has an integer value of \(actualNumber)")  
3 } else{  
4     print("\(possibleNumber)' could not be converted to an integer")  
5 }  
6 // prints "123' has an integer value of 123"
```

代码可以读作：

“如果 `Int(possibleNumber)` 返回的可选 `Int` 包含一个值，将这个可选项中的值赋予一个叫做 `actualNumber` 的新常量。”

如果转换成功，常量 `actualNumber` 就可以用在 `if` 语句的第一个分支中，他早已被可选内部的值进行了初始化，所以这时就没有必要用 `!?` 后缀来获取里边的值。在这个栗子中 `actualNumber` 被用来输出转换后的值。

常量和变量都可以使用可选项绑定，如果你想操作 `if` 语句中第一个分支的 `actualNumber` 的值，你可以写 `if var actualNumber` 来代替，可选项内部包含的值就会被设置为一个变量而不是常量。

你可以在同一个 `if` 语句中包含多可选项绑定，用逗号分隔即可。如果任一可选绑定结果是 `nil` 或者布尔值为 `false`，那么整个 `if` 判断会被看作 `false`。下面的两个 `if` 语句是等价的：

```
1 if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < secondNumber && secondNumber < 100{  
2     print("\(firstNumber) < \(secondNumber) < 100")  
3 }  
4 // Prints "4 < 42 < 100"  
5 if let firstNumber = Int("4"){  
6     if let secondNumber = Int("42"){  
7         if firstNumber < secondNumber && secondNumber < 100{  
8             print("\(firstNumber) < \(secondNumber) < 100")  
9         }  
10    }  
11 }  
12 // Prints "4 < 42 < 100"  
13
```

隐式展开可选项

如上所述，可选项明确了常量或者变量可以“没有值”。可选项可以通过 `if` 语句来判断是否有值，如果有值的话可以通过可选项绑定来获取里边的值。

有时在一些程序结构中可选项一旦被设定值之后，就会一直拥有值。在这种情况下，就可以去掉检查的需求，也不必每次访问的时候都进行展开，因为它可以安全的确认每次访问的时候都有一个值。

这种类型的可选项被定义为隐式展开可选项。通过在声明的类型后边添加一个叹号（`String!`）而非问号（`String?`）来书写隐式展开可选项。

在可选项被定义的时候就能立即确认其中有值的情况下，隐式展开可选项非常有用。如同无主引用和隐式展开的可选属性中描述的那样，隐式展开可选项主要被用在 Swift 类的初始化过程中。

隐式展开可选项是后台场景中通用的可选项，但是同样可以像非可选值那样来使用，每次访问的时候都不需要展开。下面的栗子展示了在访问被明确为 String 的可选项展开值时，可选字符串和隐式展开可选字符串的行为区别：

```
1 letpossibleString:String?="An optional string."
2 letforcedString:String=possibleString!// requires an exclamation mark
3 letassumedString:String!="An implicitly unwrapped optional string."
4 letimplicitString:String=assumedString// no need for an exclamation mark
5
```

你可以把隐式展开可选项当做在每次访问它的的时候被给予了自动进行展开的权限，你可以在声明可选项的时候添加一个叹号而不是每次调用的时候在可选项后边添加一个叹号。

注意

如果你在隐式展开可选项没有值的时候还尝试获取值，会导致运行错误。结果和在没有值的普通可选项后面加一个叹号一样。

你可以像对待普通可选一样对待隐式展开可选项来检查里边是否包含一个值：

```
1 ifassumedString!=nil{
2   print(assumedString)
3 }
4 // prints "An implicitly unwrapped optional string."
```

你也可以使用隐式展开可选项通过可选项绑定在一句话中检查和展开值：

```
1 ifletdefiniteString=assumedString{
2   print(definiteString)
3 }
4 // prints "An implicitly unwrapped optional string."
```

注意

不要在一个变量将来会变为 nil 的情况下使用隐式展开可选项。如果你需要检查一个变量在生存期内是否会变为 nil，就使用普通的可选项。

错误处理

在程序执行阶段，你可以使用错误处理机制来为错误状况负责。

相比于可选项的通过值是否缺失来判断程序的执行正确与否，而错误处理机制能允许你判断错误的形成原因，在必要的情况下，还能将你的代码中的错误传递到程序的其他地方。

当一个函数遇到错误情况，他会抛出一个错误，这个函数的访问者会捕捉到这个错误，并作出合适的反应。

```
1 funccanThrowAnError()throws{
2   // this function may or may not throw an error
3 }
```

通过在函数声明过程当中加入 throws 关键字来表明这个函数会抛出一个错误。当你调用了一个可以抛出错误的函数时，需要在表达式前预置 try 关键字。

Swift 会自动将错误传递到它们的生效范围之外，直到它们被 catch 分句处理。

```
1 do{
2     trycanThrowAnError()
3     // no error was thrown
4 }catch{
5     // an error was thrown
6 }
```

do语句创建了一个新的容器范围，可以让错误被传递到不止一个的 catch 分句里。

下面的栗子演示了如何利用错误处理机制处理不同的错误情况：

```
1 funcmakeASandwich()throws{
2     // ...
3 }
4 do{
5     trymakeASandwich()
6     eatASandwich()
7 }catchError.OutOfCleanDishes{
8     washDishes()
9 }catchError.MissingIngredients(letingredients){
10    buyGroceries(ingredients)
11 }
12 }
```

在上面的栗子中，在没有干净的盘子或者缺少原料的情况下，方法 makeASandwich() 就会抛出一个错误。由于 makeASandwich() 的抛出，方法的调用被包裹在了一个 try 的表达式中。通过将方法的调用包裹在 do 语句中，任何抛出来的错误都会被传递到预先提供的 catch 分句中。

如果没有错误抛出，方法 eatASandwich() 就会被调用，如果有错误抛出且满足 Error.OutOfCleanDishes 这个条件，方法 washDishes() 就会被执行。如果一个错误被抛出，而它又满足 Error.MissingIngredients 的条件，那么 buyGroceries(_:) 就会协同被 catch 模式捕获的 [String] 值一起调用。

有关抛出，捕获和错误传递的更详细信息请参考[错误处理](#)。

断言和先决条件

断言和先决条件用来检测运行时发生的事情。你可以使用它们来保证在执行后续代码前某必要条件是满足的。如果布尔条件在断言或先决条件中计算为 true，代码就正常继续执行。如果条件计算为 false，那么程序当前的状态就是非法的；代码执行结束，然后你的 app 终止。

你可以使用断言和先决条件来验证那些你在写代码时候的期望和假定，所以你可以包含它们作为你代码的一部分。断能够帮助你在开发的过程中找到错误和不正确的假定，先决条件帮助你探测产品的问题。在运行时帮助你额外验证你的期望，断言和先决条件同样是代码中好用的证明形式。不同于在上文[错误处理](#)中讨论的，断言和先决条件并不用于可回复或者期望的错误。由于错误断言或先决条件显示非法的程序状态，所以没办法来抓取错误断言。

使用断言和先决条件不能代替你代码中小概率非法情况的处理设计。总之，使用他们来强制数据和状态正确会让你的 app 在有非法状态时终止的更可预料，并帮助你更好的 debug。在检测到异常状态时尽可能快地停止执行同样能够帮助你减小由于异常状态造成的损失。

断言和先决条件的不同之处在于他们什么时候做检查：断言只在 debug 构建的时候检查，但先决条件则在 debug 和生产构建中生效。在生产构建中，断言中的条件不会被计算。这就是说你可以在开发的过程当中随便使用断言而无需担心影响生产性能。

使用断言进行调试

断言会在运行的时候检查一个逻辑条件是否为 true。顾名思义，断言可以“断言”一个条件是否为真。你可以使用断言确保在运行其他代码之前必要的条件已经被满足。如果条件判断为 true，代码运行会继续进行；如果条件判断为 false，代码运行结束，你的应用也就中止了。

如果你的代码在调试环境下触发了一个断言，例如你在 Xcode 中创建并运行一个应用，你可以明确的知道不可用的状态发生在什么地方，还能检查断言被触发时你的应用的状态。另外，断言还允许你附加一条调试的信息。

你可以使用全局函数 `assert(_:_)` 函数来写断言。向 `assert(_:_)` 函数传入一个结果为 true 或者 false 的表达式以及一条会在结果为 false 的时候显式的信息：

```
1 let age=-3
2 assert(age>=0,"A person's age cannot be less than zero")
3 // this causes the assertion to trigger, because age is not >= 0
```

在这个例子当中，代码执行只要在 `if age>=0` 评定为 true 时才会继续，就是说，如果 `age` 的值非负。如果 `age` 的值是负数，在上文的代码当中，`age>=0` 评定为 false，断言就会被触发，终止应用。

断言信息可以删掉如果你想的话，就像下边的栗子：

```
1 assert(age>=0)
```

如果代码已经检查了条件，你可以使用 `assertionFailure(_:file:line:)` 函数来标明断言失败，比如：

```
1 if age>10{
2     print("You can ride the roller-coaster or the ferris wheel." )
3 }elseif age>0{
4     print("You can ride the ferris wheel." )
5 }else{
6     assertionFailure("A person's age can't be less than zero." )
7 }
```

强制先决条件

在你代码中任何条件可能潜在为假但必须肯定为真才能继续执行的地方使用先决条件。比如说，使用先决条件来检测下标没有越界，或者检测函数是否收到了一个合法的值。

你可以通过调用 `precondition(_:_:file:line:)` 函数来写先决条件。给这个函数传入表达式计算为 true 或 false，如果条件的结果是 false 信息就会显示出来。比如说：

```
1 // In the implementation of a subscript...
2 precondition(index>0,"Index must be greater than zero.")
```

你可以调用 `preconditionFailure(_:_:file:line:)` 函数来标明错误发生了——比如说，如果 `switch` 的默认情况被选中，但所有的合法输入数据应该被其他 `switch` 的情况处理。

注意

如果你在不检查模式编译（`-Ounchecked`），先决条件不会检查。编译器假定先决条件永远为真，并且它根据你的代码进行优化。总之，`fatalError(_ :file:line:)` 函数一定会终止执行，无论你优化设定如何。

你可以在草拟和早期开发过程中使用 `fatalError(_ :file:line:)` 函数标记那些还没实现的功能，通过使用 `fatalError("Unimplemented")` 来作为代替。由于致命错误永远不会被优化，不同于断言和先决条件，你可以确定执行遇到这些临时占位永远会停止。

基本运算符

 cn.swift.org/basic-operators

运算符是一种用来检查、改变或者合并值的特殊符号或组合符号。举例来说，加运算符（`+`）能够把两个数字相加（比如 `let i=1+2`）。更复杂的栗子包括逻辑与运算 `&&` 比如 `if enteredDoorCode && passedRetinaScan`。

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。如同 [溢出操作符](#) 中描述的那样，你可以通过使用 Swift 的溢出操作符来选择进入值溢出行为模式。

Swift 提供了两种 C 中没有的区间运算符（`a..<b` 和 `a...b`），来让你便捷表达某个范围的值。

这个章节叙述了 Swift 语言当中常见的运算符。[高级运算符](#) 则涵盖了 Swift 中的高级运算符，同时描述了如何定义你自己的运算符以及在你自己的类当中实现标准运算符。

专门用语

运算符包括一元、二元、三元：

- **一元运算符**对一个目标进行操作（比如 `-a`）。一元前缀运算符在目标之前直接添加（比如 `!b`），同时一元后缀运算符直接在目标末尾添加（比如 `c!`）。
- **二元运算符**对两个目标进行操作（比如 `2+3`）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。如同 C，Swift 语言也仅有一个三元运算符，三元条件运算符（`a?b:c`）。

受到运算符影响的值叫做操作数。在表达式 `1+2` 中，`+` 符号是一个二元运算符，其中的两个值 `1` 和 `2` 就是操作数。

赋值运算符

赋值运算符（`a=b`）可以初始化或者更新 `a` 为 `b` 的值：

```
1 let b=10
2 var a=5
3 a=b
4 // a 的值现在是 10
```

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：

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

与 Objective-C 和 C 不同，Swift 的赋值符号自身不会返回值。下面的语句是不合法的：

```
1 ifx=y{  
2 // 这是不合法的, 因为 x = y 并不会返回任何值。  
3 }
```

这个特性避免了赋值符号 (`=`) 被意外地用于等于符号 (`==`) 的实际意图上。Swift 通过让 `ifx=y` 非法来帮助你避免这类的错误在你的代码中出现。

算术运算符

Swift 对所有的数字类型支持四种标准算术运算符：

- 加 (`+`)
- 减 (`-`)
- 乘 (`*`)
- 除 (`/`)

```
1 1+2// equals 3  
2 5-3// equals 2  
3 2*3// equals 6  
4 10.0/2.5// equals 4.0
```

与 C 和 Objective-C 中的算术运算符不同，Swift 算术运算符默认不允许值溢出。你可以选择使用 Swift 的溢出操作符 (比如 `a&+b`) 来行使溢出行为。参见 [溢出操作符](#) 加法运算符同时也支持 `String` 的拼接：

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

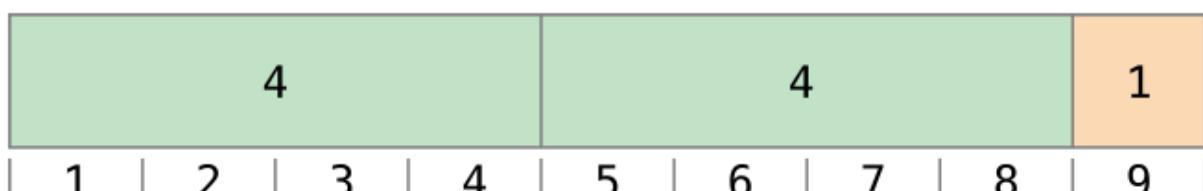
余数运算符

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) 。

注意

余数运算符 (`%`) 同样会在别的语言中称作取模运算符。总之，严格来讲的话这个行为对应着 Swift 中对负数的操作，所以余数要比模取更合适。

现在我们展示余数运算符如何生效。要计算 `9%4`，你首先要求出多少个 4 能够放到 9 里边：



你可以给 9 当中放进两个 4 去，这样就得到了 1 这个余数 (橘黄色的部分) 。

在 Swift 中，这将写作：

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

决定 $a \% b$ 的结果， $\%$ 按照如下等式运算然后返回 remainder 作为它的输出：

$a = (b \times \text{some multiplier}) + \text{remainder}$

此时 some multiplier 是 b 能放进 a 的最大倍数。

把 9 和 4 插入到等式当中去：

```
9 = (4 \times 2) + 1
```

当 a 是负数时也使用相同的方法来进行计算：

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

把 -9 和 4 插入到等式当中：

```
-9 = (4 \times -2) + -1
```

得到余数 -1。

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。

一元减号运算符

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**：

```
1 letthree=3
2 letminusThree=-three// minusThree equals -3
3 letplusThree=-minusThree// plusThree equals 3, or "minus minus three"
```

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

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：

```
1 letminusSix=-6
2 letalsoMinusSix=+minusSix// alsoMinusSix equals -6
```

尽管一元加号运算符实际上什么也不做，你还是可以对正数使用它来让你的代码对一元减号运算符来说显得更加对称。

组合赋值符号

如同 C，Swift 提供了由赋值符号（ $=$ ）和其他符号组成的 **组合赋值符号**。一个加赋值符号的栗子（ $+=$ ）：

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

表达式 **a+=2** 其实就是 **a=a+2** 的简写。效率上来讲，加号和赋值符号组合成的一个运算符能够同时进行这两个操作。

注意

组合运算符不会返回任何值。举例来说，你不能写成这样 **let b=a+=2**。这个与前边提到的增量和减量符号的行为不同。

你可以在 表达式 (此处应有链接) 章节找到组合赋值符号的完整列表。

比较运算符

Swift 支持所有 C 的标准比较运算符：

- 相等 (**a==b**)
- 不相等 (**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 同时也提供两个等价运算符 (**==** 和 **!=**)，你可以使用它们来判断两个对象的引用是否相同。参考 [类和结构体](#) 章节来了解更多。

每个比较运算符都会返回一个 **Bool** 值来表示语句是否为真：

```
1 1==1// true, because 1 is equal to 1
2 2!=1// true, because 2 is not equal to 1
3 2>1// true, because 2 is greater than 1
4 1<2// true, because 1 is less than 2
5 1>=1// true, because 1 is greater than or equal to 1
6 2<=1// false, because 2 is not less than or equal to 1
```

比较运算符通常被用在条件语句当中，比如说 **if** 语句：

```
1 let name="world"
2 if name=="world"{
3     print("hello, world")
4 }else{
5     print("I'm sorry \\\(name), but I don't recognize you" )
6 }
7 // prints "hello, world", because name is indeed equal to "world"
```

更多关于 **if** 语句的信息，参见 [控制流](#)。

你同样可以比较拥有同样数量值的元组，只要元组中的每个值都是可比较的。比如说，**Int** 和 **String** 都可以用来比较大小，也就是说 **(Int,String)** 类型的元组就可以比较。一般来说，**Bool** 不能比较，这意味着包含布尔值的元组不能用来比较大小。

元组以从左到右的顺序比较大小，一次一个值，直到找到两个不相等的值为止。如果所有的值都是相等的，那么就认为元组本身是相等的。比如说：

```
1 (1,"zebra")<(2,"apple")// true because 1 is less than 2
2 (3,"apple")<(3,"bird")// true because 3 is equal to 3, and "apple" is less than "bird"
3 (4,"dog")==(4,"dog")// true because 4 is equal to 4, and "dog" is equal to "dog"
```

注意

Swift 标准库包含的元组比较运算符仅支持小于七个元素的元组。要比较拥有七个或者更多元素的元组，你必须自己实现比较运算符。

三元条件运算符

三元条件运算符是一种有三部分的特殊运算，它看起来是这样的：

question?answer1:answer2。这是一种基于 question 是真还是假来选择两个表达式之一的便捷写法。如果 question 是真，则会判断为 answer1 并且返回它的值；否则，它判断为 answer2 并且返回它的值。

三元条件运算符就是下边代码的简写：

```
1 if question{
2   answer1
3 }else{
4   answer2
5 }
```

这里有一个栗子，它计算一个表格的行高。行高应该是比内容的高度高50点，如果行有标题的话。要是没有标题，就比内容高20点：

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

先前的栗子是下边代码的简写：

```
1 let contentHeight=40
2 let hasHeader=true
3 var rowHeight=contentHeight
4 if hasHeader{
5   rowHeight=rowHeight+50
6 }else{
7   rowHeight=rowHeight+20
8 }
9 // rowHeight is equal to 90
```

第一个栗子使用了三元条件运算符意味着 rowHeight 可以在单行代码中就能被设置为正确的值。它要比第二个栗子简洁不少，并且不再需要 rowHeight 为变量，因为它的值不需要在 if 中做改变了。

三元条件运算符提供了一个非常有效的简写来决策要两个表达式之间选哪个。总之，使用三元条件运算符要小心。它的简洁性会导致你代码重用的时候失去易读的特性。避免把多个三元条件运算符组合到一句代码当中。

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。

合并空值运算符是下边代码的缩写：

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

上边的代码中，三元条件运算符强制展开（`a!`）储存在 `a` 中的值，如果 `a` 不是 `nil` 的话，否则就返回 `b` 的值。合并空值运算符提供了更加优雅的方式来封装这个条件选择和展开操作，让它更加简洁易读。

注意

如果 `a` 的值是非空的，`b` 的值将不会被考虑。这就是所谓的 短路计算。

下边的栗子使用了合并空值运算符来在默认颜色名和可选的用户定义颜色名之间做选择：

```
1 letdefaultColorName="red"
2 varuserDefinedColorName:String?? defaults to nil
3 varcolorNameToUse=userDefinedColorName??defaultColorName
4 // userDefinedColorName is nil, so colorNameToUse is set to the default of "red"
```

`userDefinedColorName` 变量被定义为可选的 `String`，默认为 `nil`。由于 `userDefinedColorName` 是一个可选类型，你可以使用合并空值运算符来控制它的值。在上边的栗子当中，这个运算符被用来决定 `String` 类型的变量 `colorNameToUse` 的初始值。因为 `userDefinedColorName` 是 `nil`，表达式 `userDefinedColorName??defaultColorName` 返回了 `defaultColorName` 的值，"red"。

如果你给 `userDefinedColorName` 指定一个非空的值然后让合并空值运算符在检查一次，那么 `userDefinedColorName` 中封装的值将会替换掉默认值：

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

区间运算符

Swift 包含了两个 区间运算符，他们是表示一个范围的值的便捷方式。

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`。

在遍历你需要用到的所有数字时，使用闭区间运算符是个不错的选择，比如说在 `for-in` 循环

当中：

```
1 for index in 1...5{
2     print("\\"(index) times 5 is \\"(index*5)")
3 }
4 // 1 times 5 is 5
5 // 2 times 5 is 10
6 // 3 times 5 is 15
7 // 4 times 5 is 20
8 // 5 times 5 is 25
```

更多关于 for-in 循环的内容，参见[控制流](#)。

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，即 半开，因为它只包含起始值但并不包含结束值。 (十奶注：其实这就是左闭右开区间。) 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。

半开区间在遍历基于零开始序列比如说数组的时候非常有用，它从零开始遍历到数组长度（但是不包含）：

```
1 let names=["Anna", "Alex", "Brian", "Jack"]
2 let count=names.count
3 for i in 0..<count{
4     print("Person \\"(i+1) is called \\"(names[i])")
5 }
6 // Person 1 is called Anna
7 // Person 2 is called Alex
8 // Person 3 is called Brian
9 // Person 4 is called Jack
```

注意数组包含四个元素，但是 0..<count 只遍历到 3 (元素序号的最大值) ，因为这是一个半开区间。关于数组的更多内容，参见[数组](#)。

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——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。因为运算符只有一侧有值，所以这种区间叫做单侧区间。比如说：

```
1 for name in names[2...]{ 
2     print(name)
3 }
4 // Brian
5 // Jack
6 for name in names[...2]{ 
7     print(name)
8 }
9 // Anna
10 // Alex
11 // Brian
12
```

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。和你两侧都包含值一样，最终的值不是区间的一部分。举例来说：

```
1 forname in names[..<2]{  
2     print(name)  
3 }  
4 // Anna  
5 // Alex
```

单侧区间可以在其他上下文中使用，不仅仅是下标。你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；总之，由于区间无限连续，你要确保给循环添加一个显式的条件。你同样可以检测单侧区间是否包含特定的值，就如下面的代码所述。

```
1 let range=...5  
2 range.contains(7)// false  
3 range.contains(4)// true  
4 range.contains(-1)// true
```

逻辑运算符

逻辑运算符可以修改或者合并布尔逻辑值 `true` 和 `false`。Swift 支持三种其他基于 C 的语言也包含的标准逻辑运算符

- 逻辑 非 (`!a`)
- 逻辑 与 (`a&&b`)
- 逻辑 或 (`a||b`)

逻辑非运算符

逻辑非运算符 (`!a`) 会转换布尔值，把 `true` 变成 `false`，把 `false` 变成 `true`。

逻辑非运算符是一个前缀运算符，它直接写在要进行运算的值前边，不加空格。读作“非 `a`”，如同下边的栗子：

```
1 let allowedEntry=false  
2 if!allowedEntry{  
3     print("ACCESS DENIED")  
4 }  
5 // prints "ACCESS DENIED"
```

这句 `if!allowedEntry` 可以读作“如果不允进入。”后边的代码只有“不允许进入”为真才会执行；比如说现在 `allowedEntry` 为 `false`。

在这个栗子当中，要注意布尔量的常量和变量名能够帮助你保持代码的可读和简洁，同时也避免双重否定或者其他奇奇怪怪的逻辑语句。

逻辑与运算符

逻辑与运算符 (`a&&b`) 需要逻辑表达式的两个值都为 `true`，整个表达式的值才为 `true`。

如果任意一个值是 `false`，那么整个表达式的结果会是 `false`。事实上，如果第一个值是 `false`，那么第二个值就会被忽略掉了，因为它已经无法让整个表达式再成为 `true`。这就是所谓的 短路计算。

这个栗子依据两个 `Bool` 值判断只有它们都为 `true` 时才允许访问：

```
1 letenteredDoorCode=true
2 letpassedRetinaScan=false
3 ifenteredDoorCode&&passedRetinaScan{
4     print("Welcome!")
5 }else{
6     print("ACCESS DENIED")
7 }
8 // prints "ACCESS DENIED"
```

逻辑或运算符

逻辑或运算符（`a||b`）是一个中缀运算符，它由两个相邻的管道字符组成。你可以使用它来创建两个值之间只要有一个为 `true` 那么整个表达式就是 `true` 的逻辑表达式。

如同上文中的逻辑与运算符，逻辑或运算符也使用短路计算来判断表达式。如果逻辑或运算符左侧的表达式为 `true`，那么右侧则不予考虑了，因为它不会影响到整个逻辑表达式的結果。

在下边的栗子当中，第一个 Bool 值（`hasDoorKey`）是 `false`，但是第二个值（`knowsOverridePassword`）是 `true`。由于有一个值是 `true`，这整个逻辑表达式的值同样被判断为 `true`，所以访问被允许：

```
1 lethasDoorKey=false
2 letknowsOverridePassword=true
3 ifhasDoorKey||knowsOverridePassword{
4     print("Welcome!")
5 }else{
6     print("ACCESS DENIED")
7 }
8 // prints "Welcome!"
```

混合逻辑运算

你可以组合多个逻辑运算符来创建一个更长的组合表达式：

```
1 ifenteredDoorCode&&passedRetinaScan||hasDoorKey||knowsOverridePassword{
2     print("Welcome!")
3 }else{
4     print("ACCESS DENIED")
5 }
6 // prints "Welcome!"
```

这个栗子使用了多个 `&&` 和 `||` 运算符来创建组合表达式。不过，`&&` 和 `||` 仍旧只能够操作两个值，它实际上是三个更小的表达式链接而成。这个栗子可以读作：

如果我们输入了正确的密码并通过了视网膜扫描，或者如果我们有合法的钥匙或者我们知道紧急超驰密码，就允许进入。

基于 `enteredDoorCode`，`passedRetinaScan`，和 `hasDoorKey` 的值，前两个子表达式都是 `false`。总之，紧急超驰密码是知道的，所以整个组合的表达式仍然被评定为 `true`。

注意^[1]

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。

译注

[1]：优先级问题：在Swift 编程语言全文当中并没有提到逻辑运算符的优先级问题（即默认相等），总之，它们是有优先级的，在标准库引用文档中提及。

显式括号

很多时候虽然不被要求，但使用括号还是很有用的，这能让复杂的表达式更容易阅读。在上文当中的门禁栗子里，把前边部分的表达式用圆括号括起来就会让整个组合表达式的意图更加明显：

```
1 if(enteredDoorCode&&passedRetinaScan)||hasDoorKey||knowsOverridePassword{
2     print("Welcome!")
3 }else{
4     print("ACCESS DENIED")
5 }
6 // prints "Welcome!"
```

圆括号把前边的两个值单独作为一部分来考虑，这样使整个表达式的意图清晰明显。组合表达式的输出并没有改变，但是整个意图变得清晰易读。可读性永远是第一位的；当需要的时候，使用圆括号让你的意图更加明确。

字符串和字符

 [cnswift.org/strings-and-characters](https://cn.swift.org/strings-and-characters)

字符串是一系列的字符，比如说 "hello, world" 或者 "albatross"。Swift 的字符串用 `String` 类型来表示。`String` 的内容可以通过各种方法来访问到，包括作为 `Character` 值的集合。

Swift 的 `String` 和 `Character` 类型提供了一种快速的符合 Unicode 的方式操作你的代码。字符串的创建和修改语法非常轻量易读，使用与 C 类似的字符串字面量语法。字符串串联只需要使用 `+` 运算符即可，字符串的可修改能力通过选择常量和变量来进行管理，就如同 Swift 语言中的其他值。你同样可以使用字符串来插入常量、变量、字面量以及表达式到更长的字符串当中，这就是所谓的字符串插值。这样让创建自定义字符串值来显示、储存和打印值变得更加简单。

别看语法简单，Swift 的 `String` 类型仍旧是快速和现代的字符串实现。每一个字符串都是由 Unicode 字符的独立编码组成，并且提供了多种 Unicode 表示下访问这些字符的支持。

注意

Swift 的 `String` 类型桥接到了基础库中的 `NSString` 类。Foundation 同时也扩展了所有 `NSString` 定义的方法给 `String`。也就是说，如果你导入 Foundation，就可以在 `String` 中访问所有的 `NSString` 方法，无需转换格式。

更多在 Foundation 和 Cocoa 框架中使用 `String` 的内容，参见 [与 Cocoa 和 Objective-C 一起使用 Swift \(Swift 4\) \(官网链接\)](#)。

字符串字面量

你可以在你的代码中插入预先写好的 `String` 值作为字符串字面量。字符串字面量是被双引号 ("") 包裹的固定顺序文本字符。

使用字符串字面量作为常量或者变量的初始值：

注意

更多关于在字符串字面量中使用特殊字符的信息请参考 [字符串字面量中的特殊字符](#)。

如果你需要很多行的字符串，使用多行字符串字面量。多行字符串字面量是用三个双引号引起来的一系列字符：

```
1 let quotation="""\n2 The White Rabbit put on his spectacles. "WhereshallIbegin,\n3 please your Majesty?" he asked.\n4 "Begin at the beginning," the King said gravely, "andgo on\n5 till you come tothe end;thenstop."\n6 """\n7
```

如同上面展示的那样，由于多行用了三个双引号而不是一个，你可以在多行字面量中使用单个双引号 "。要在多行字符串中包含 ""”，你必须用反斜杠 (\) 转义至少其中一个双引号。举例来说：

```
1 letthreeDoubleQuotes="""
2 Escaping the first quote \'"""
3 Escaping all three quotes \"\"\""
4 """
```

更多关于使用反斜杠转义特殊字符，见[8.2 字符串字面量中的特殊字符](#)。

在这个多行格式中，字符串字面量包含了双引号包括的所有行。字符串起始于三个双引号 (""") 之后的第一行，结束于三个双引号 (""") 之前的一行，也就是说双引号不会开始或结束带有换行。下面的两个字符串是一样的：

```
1 letsingleLineString="These are the same."
2 letmultilineString=""""
3 These are the same.
4 """
```

要让多行字符串字面量开始或结束带有换行，写一个空行作为第一行或者是最后一行。比如：

```
1 """
2 This string starts with a line feed.
3 It also ends with a line feed.
4 """
5
6
```

多行字符串可以缩进以匹配周围的代码。双引号 (""") 前的空格会告诉 Swift 其他行前应该有多少空白是需要忽略的。比如说，尽管下面函数中多行字符串字面量缩进了，但实际上字符串不会以任何空白开头。

```
1 funcgenerateQuotation()->String{
2     letquotation=""""
3         The White Rabbit put on his spectacles. " WhereshallIbegin,
4     please your Majesty?" he asked.
5         "Begin at the beginning," the King said gravely, " andgo on
6     till you come tothe end;thenstop."
7         """
8     returnquotation
9 }
10 print(quotation==generateQuotation())
11 // Prints "true"
12
```

总而言之，如果你在某行的空格超过了结束的双引号 (""") ，那么这些空格会被包含。

```

let linesWithIndentation = """
    This line doesn't begin with whitespace.
Space ignored ─────────────────────────────────────────────────────────────────
    This line begins with four spaces.
Appears in string ─────────────────────────────────────────────────────────
    This line doesn't begin with whitespace.
    """

```

初始化一个空字符串

为了绑定一个更长的字符串，要在一开始创建一个空的 String 值，要么赋值一个空的字符串面量给变量，要么使用初始化器语法来初始化一个新的 String 实例：

```

1 varemptyString=""// empty string literal
2 varanotherEmptyString=String()// initializer syntax
3 // these two strings are both empty, and are equivalent to each other

```

通过检查布尔量 isEmpty 属性来确认一个 String 值是否为空：

```

1 ifemptyString.isEmpty{
2     print("Nothing to see here")
3 }
4 // prints "Nothing to see here"

```

字符串可变性

你可以通过把一个 String 设置为变量（这里指可被修改），或者为常量（不能被修改）来指定它是否可以被修改（或者改变）：

```

1 varvariableString="Horse"
2 variableString+=" and carriage"
3 // variableString is now "Horse and carriage"
4 letconstantString="Highlander"
5 constantString+=" and another Highlander"
6 // this reports a compile-time error - a constant string cannot be modified
7

```

注意

这个功能与 Objective-C 和 Cocoa 中的字符串改变不同，通过选择不同的类（`NSString` 和 `NSMutableString`）来明确字符串是否可被改变。

字符串是值类型

Swift 的 String 类型是一种值类型。如果你创建了一个新的 String 值，String 值在传递给方法或者函数的时候会被复制过去，还有赋值给常量或者变量的时候也是一样。每一次赋值和传递，现存的 String 值都会被复制一次，传递走的是拷贝而不是原本。值类型在 [结构体和枚举是值类型](#) 一章当中有详细描述。

Swift 的默认拷贝 String 行为保证了当一个方法或者函数传给你一个 String 值，你就绝对拥有了这个 String 值，无需关心它从哪里来。你可以确定你传走的这个字符串除了你自己就不会有别人改变它。

另一方面，Swift 编译器优化了字符串使用的资源，实际上拷贝只会在确实需要的时候才进行。这意味着当你把字符串当做值类型来操作的时候总是能够有很棒的性能。

操作字符

你可以通过 for-in 循环遍历 String 中的每一个独立的 Character 值：

```
1 for character in "Dog!%@", {
2     print(character)
3 }
4 // D
5 // o
6 // g
7 // !
8 // %@",
```

在 [For-In 循环](#) 一节中有 for-in 循环的详细叙述。

另外，你可以通过提供 Character 类型标注来从单个字符的字符串字面量创建一个独立的 Character 常量或者变量：

```
1 let exclamationMark: Character = "!"
```

String 值可以通过传入 Character 值的字符串作为实际参数到它的初始化器来构造：

```
1 let catCharacters: [Character] = ["C", "a", "t", "!", "😺"]
2 let catString = String(catCharacters)
3 print(catString)
4 // prints "Cat!😺"
```

连接字符串和字符

String 值能够被加起来（或者说连接），使用加运算符（+）来创建新的 String 值：

```
1 let string1 = "hello"
2 let string2 = " there"
3 var welcome = string1 + string2
4 // welcome now equals "hello there"
```

你同样也可以使用加赋值符号（+=）在已经存在的 String 值末尾追加一个 String 值：

```
1 var instruction = "look over"
2 instruction += string2
3 // instruction now equals "look over there"
```

你使用 String 类型的 append() 方法来可以给一个 String 变量的末尾追加 Character 值：

```
1 let exclamationMark: Character = "!"  
2 welcome.append(exclamationMark)  
3 // welcome now equals "hello there!"
```

注意

你不能把 String 或者 Character 追加到已经存在的 Character 变量当中，因为 Character 值能且只能包含一个字符。

字符串插值

字符串插值是一种从混合常量、变量、字面量和表达式的字符串字面量构造新 String 值的方法。每一个你插入到字符串字面量的元素都要被一对圆括号包裹，然后使用反斜杠前缀：

```
1 let multiplier = 3  
2 let message = "\((multiplier) times 2.5 is \(Double(multiplier) * 2.5))"  
3 // message is "3 times 2.5 is 7.5"
```

在上边的栗子当中，multiplier 的值以 \((multiplier) 的形式插入到了字符串字面量当中。当字符串插值需要被用来创建真的字符串的时候，这个占位符就会被 multiplier 的真实值代替。multiplier 的值同时也是字符串后边更大一点表达式的一部分。这个表达式计算了 Double(multiplier)*2.5 的值并且插入结果 (7.5) 到字符串当中。在这个栗子当中，表达式在字符串字面量中被写作 \((Double(multiplier)*2.5) 的形式。

注意

你作为插值写在圆括号中的表达式不能包含非转义的双引号 ("") 或者反斜杠 (\)，并且不能包含回车或者换行符。总之，它们可以包含其他字符串字面量。

Unicode

Unicode 是一种在不同书写系统中编码、表示和处理文本的国际标准。它允许你表示几乎标准化格式的任何语言中的任何字符，并且为外部源比如文本文档或者网页读写这些字符。如同这节中描述的那样，Swift 的 String 和 Character 类型是完全 Unicode 兼容的。

Unicode 标量

面板之下，Swift 的原生 String 类型建立于 Unicode 标量值之上。一个 Unicode 标量是一个为字符或者修饰符创建的独一无二的 21 位数字，比如 LATIN SMALL LETTER A ("a") 的 U+0061，或者 FRONT-FACING BABY CHICK ("🐥") 的 U+1F425。

注意

Unicode 标量码位位于 U+0000 到 U+D7FF 或者 U+E000 到 U+10FFFF 之间。Unicode 标量码位不包括从 U+D800 到 U+DFFF 的 16 位码元码位。

注意不是所有的 21 位 Unicode 标量都指定了字符——有些标量是为将来所保留的。那些有了字符的标量通常来说也会有个名字，比如上边栗子中的 LATIN SMALL LETTER A 和 FRONT-FACING BABY CHICK。

字符串字面量中的特殊字符

字符串字面量能包含以下特殊字符：

- 转义特殊字符 \0 (空字符) , \\ (反斜杠) , \t (水平制表符) , \n (换行符) , \r(回车符) , \" (双引号) 以及 ' (单引号) ;
- 任意的 Unicode 标量，写作 \u{n}，里边的 n 是一个 1-8 个与合法 Unicode 码位相等的 16 进制数字。

下边的代码展示了这些特殊字符的四个栗子。 wiseWords 常量包含了两个转义双引号字符。 dollarSign , blackHeart 和 sparklingHeart 常量展示了 Unicode 标量格式：

```
1 letwiseWords = "\"Imagination is more important than knowledge\" - Einstein"
2 // "Imagination is more important than knowledge" - Einstein
3 letdollarSign = "\u{24}"// $, Unicode scalar U+0024
4 letblackHeart = "\u{2665}"// ♥, Unicode scalar U+2665
5 letsparklingHeart = "\u{1F496}"// 💫, Unicode scalar U+1F496
```

扩展字形集群

每一个 Swift 的 Character 类型实例都表示了单一的 扩展字形集群。 扩展字形集群是一个或者多个有序的 Unicode 标量（当组合起来时）产生的单个人类可读字符。

这有个栗子。字母 é 以单个 Unicode 标量 é (LATIN SMALL LETTER E WITH ACUTE, 或者 U+00E9) 表示。总之，同样的字母也可以用一对标量——一个标准的字母 e (LATIN SMALL LETTER E, 或者说 U+0065) ，以及 COMBINING ACUTE ACCENT 标量 (U+0301) 表示。 COMBINING ACUTE ACCENT 标量会以图形方式应用到它前边的标量上，当 Unicode 文本渲染系统渲染时，就会把 e 转换为 é 来输出。

在这两种情况中，字母 é 都会作为单独的 Swift Character 值以扩展字形集群来表示。在前者当中，集群包含了一个单独的标量；后者，则是两个标量的集群：

```
1 leteAcute:Character = "\u{E9}"// é
2 letcombinedEAcute:Character = "\u{65}\u{301}"// e followed by
3 // eAcute is é, combinedEAcute is é
```

扩展字形集群是一种非常灵活的把各种复杂脚本字符作为单一 Character 值来表示的方法。比如说韩文字母中的音节能被表示为复合和分解序列两种。这两种表示在 Swift 中都完全合格于单一 Character 值：

```
1 letprecomposed:Character = "\u{D55C}"// 한
2 letdecomposed:Character = "\u{1112}\u{1161}\u{11AB}"// ㄱ, ㅏ, ㄴ
3 // precomposed is 한, decomposed is 한
```

扩展字形集群允许封闭标记的标量（比如 COMBINING ENCLOSING CIRCLE, 或者说 U+20DD）作为单一 Character 值来圈住其他 Unicode 标量：

```
1 letenclosedEAcute:Character = "\u{E9}\u{20DD}"
2 // enclosedEAcute is ⓘ
```

区域指示符号的 Unicode 标量可以成对组合来成为单一的 Character 值，比如说这个 REGIONAL INDICATOR SYMBOL LETTERU (U+1F1FA) 和 REGIONAL INDICATOR SYMBOL LETTERS (U+1F1F8) :

```
1 let regionalIndicatorForUS:Character = "\u{1F1FA}\u{1F1F8}"
2 // regionalIndicatorForUS is U+S
```

字符统计

要在字符串中取回 Character 值的总数，使用字符串的 count 属性：

```
1 let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐕"
2 print("unusualMenagerie has \(unusualMenagerie.count) characters")
3 // Prints "unusualMenagerie has 40 characters"
```

注意 Swift 为 Character 值使用的扩展字形集群意味着字符串的创建和修改可能不会总是影响字符串的字符统计数。

比如说，如果你使用四个字符的 cafe 来初始化一个新的字符串，然后追加一个 COMBINING ACUTE ACCENT (U+0301) 到字符串的末尾，字符串的字符统计结果将仍旧是 4，但第四个字符是 é 而不是 e：

```
1 var word = "cafe"
2 print("the number of characters in \(word) is \(word.count)")
3 // Prints "the number of characters in cafe is 4"
4 word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301
5 print("the number of characters in \(word) is \(word.count)")
6 // Prints "the number of characters in café is 4"
7
8
```

注意

扩展字形集群能够组合一个或者多个 Unicode 标量。这意味着不同的字符——以及相同字符的不同表示——能够获得不同大小的内存来储存。因此，Swift 中的字符并不会在字符串中获得相同的内存空间。所以说，字符串中字符的数量如果不遍历它的扩展字形集群边界的话，是不能被计算出来的。如果你在操作特殊的长字符串值，要注意 count 属性为了确定字符串中的字符要遍历整个字符串的 Unicode 标量。

通过 count 属性返回的字符统计并不会总是与包含相同字符的 NSString 中 length 属性相同。NSString 中的长度是基于在字符串的 UTF-16 表示中 16 位码元的数量来表示的，而不是字符串中 Unicode 扩展字形集群的数量。

访问和修改字符串

你可以通过下标脚本语法或者它自身的属性和方法来访问和修改字符串。

字符串索引

每一个 String 值都有相关的索引类型，String.Index，它相当于每个 Character 在字符串中的位置。

如上文中提到的那样，不同的字符会获得不同的内存空间来储存，所以为了明确哪个 Character 在哪个特定的位置，你必须从 String 的开头或结尾遍历每一个 Unicode 标量。因此，Swift 的字符串不能通过整数值索引。

使用 startIndex 属性来访问 String 中第一个 Character 的位置。 endIndex 属性就是 String 中最后一个字符后的位置。所以说， endIndex 属性并不是字符串下标脚本的合法实际参数。如果 String 为空，则 startIndex 与 endIndex 相等。

使用 index(before:) 和 index(after:) 方法来访问给定索引的前后。要访问给定索引更远的索引，你可以使用 index(_:offsetBy:) 方法而不是多次调用这两个方法。

你可以使用下标脚本语法来访问 String 索引中的特定 Character。

```
1 let greeting="Guten Tag!"  
2 greeting[greeting.startIndex]  
3 // G  
4 greeting[greeting.index(before:greeting.endIndex)]  
5 // !  
6 greeting[greeting.index(after:greeting.startIndex)]  
7 // u  
8 let index=greeting.index(greeting.startIndex,offsetBy:7)  
9 greeting[index]  
10 // a
```

尝试访问的 Character 如果索引位置在字符串范围之外，就会触发运行时错误。

```
1 greeting[greeting.endIndex]// error  
2 greeting.index(after:endIndex)// error
```

使用 characters 属性的 indices 属性来创建所有能够用来访问字符串中独立字符的索引范围 Range。

```
1 for index in greeting.characters.indices{  
2 print("\(greeting[index]) ", terminator:"")  
3 }  
4 // Prints "G u t e n  T a g !"
```

插入和删除

要给字符串的特定索引位置插入字符，使用 insert(_:at:) 方法，另外要冲入另一个字符串的内容到特定的索引，使用 insert(contentsOf:at:) 方法。

```
1 var welcome="hello"  
2 welcome.insert("!", at:welcome.endIndex)  
3 // welcome now equals "hello!"  
4 welcome.insert(contentsOf:" there".characters, at:welcome.index(before:welcome.endIndex))  
5 // welcome now equals "hello there!"  
6
```

要从字符串的特定索引位置移除字符，使用 remove(at:) 方法，另外要移除一小段特定范围的字符串，使用 removeSubrange(_:) 方法：

```
1 welcome.remove(at:welcome.index(before:welcome.endIndex))
2 // welcome now equals "hello there"
3 letrange=welcome.index(welcome.endIndex,offsetBy:-6)..<welcome.endIndex
4 welcome.removeSubrange(range)
5 // welcome now equals "hello"
6
```

注意

你可以在任何遵循了 RangeReplaceableIndexable 协议的类型中使用 insert(_:at:) , insert(contentsOf:at:) , remove(at:) 方法。这包括了这里使用的 String , 同样还有集合类型比如 Array , Dictionary 和 Set 。

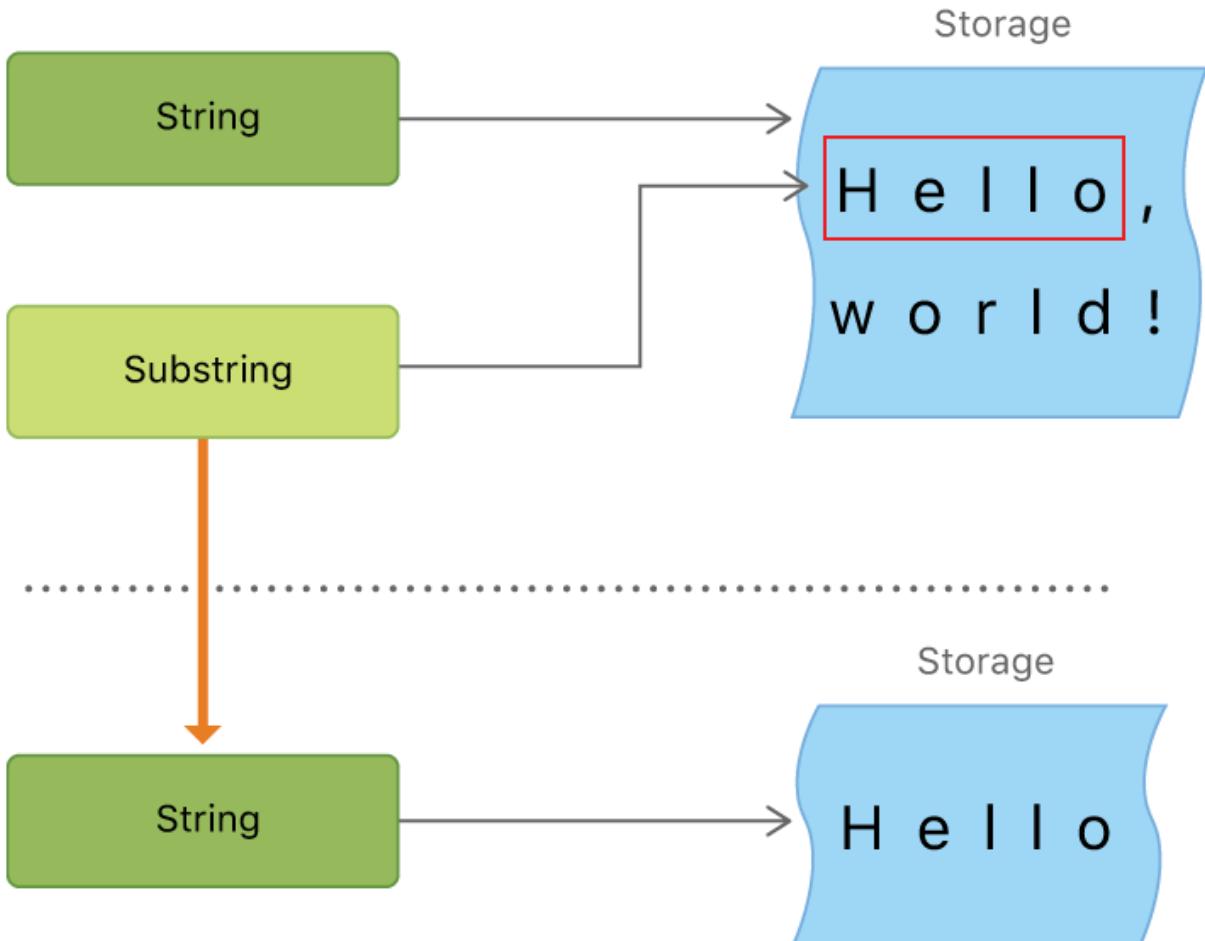
子字符串

当你获得了一个字符串的子字符串——比如说，使用下标或者类似 prefix(_) 的方法——结果是一个 Substring 的实例，不是另外一个字符串。Swift 中的子字符串拥有绝大部分字符串所拥有的方法，也就是说你可以用操作字符串相同的方法来操作子字符串。总之，与字符串不同，在字符串上执行动作的话你应该使用子字符串执行短期处理。当你想要把结果保存得长久一点时，你需要把子字符串转换为 String 实例。比如说：

```
1 letgreeting="Hello, world!"
2 letindex=greeting.index(of:",")??greeting.endIndex
3 letbeginning=greeting[..
```

与字符串类似，每一个子字符串都有一块内存区域用来保存组成子字符串的字符。字符串与子字符串的不同之处在于，作为性能上的优化，子字符串可以重用一部分用来保存原字符串的内存，或者是用来保存其他子字符串的内存。（字符串也拥有类似的优化，但是如果两个字符串使用相同的内存，他们就是等价的。）这个性能优化意味着在你修改字符串或者子字符串之前都不需要花费拷贝内存的代价。如同上面所说的，子字符串并不适合长期保存——因为它们重用了原字符串的内存，只要这个字符串有子字符串在使用中，那么这个字符串就必须一直保存在内存里。

在上面的例子中， greeting 是一个字符串，也就是说它拥有一块内存保存着组成这个字符串的字符。由于 beginning 是 greeting 的子字符串，它重用了 greeting 所用的内存。不同的是， newString 是字符串——当它从子字符串创建时，它就有了自己的内存。下面的图例显示了这些关系：



注意

String 和 Substring 都遵循 [StringProtocol](#) 协议，也就是说它基本上能很方便地兼容所有接受 StringProtocol 值的字符串操作函数。你可以无差别使用 String 或 Substring 值来调用这些函数。

字符串比较

Swift 提供了三种方法来比较文本值：字符串和字符相等性，前缀相等性以及后缀相等性。

字符串和字符相等性

如同[比较运算符](#)中所描述的那样，字符串和字符相等使用“等于”运算符 (==) 和“不等”运算符 (!=) 进行检查：

```

1 let quotation="We're a lot alike, you and I."
2 let sameQuotation="We're a lot alike, you and I."
3 if quotation==sameQuotation{
4   print("These two strings are considered equal")
5 }
6 // prints "These two strings are considered equal"
  
```

两个 String 值（或者两个 Character 值）如果它们的扩展字形集群是规范化相等，则被认为是相等的。如果扩展字形集群拥有相同的语言意义和外形，我们就说它规范化相等，就算它们实际上是由不同的 Unicode 标量组合而成。

比如说， LATIN SMALL LETTER E WITH ACUTE (U+00E9) 是规范化相等于 LATIN SMALL

LETTERE(U+0065)加 COMBINING ACUTE ACCENT (U+0301)的。这两个扩展字形集群都是表示字符é的合法方式，所以它们被看做规范化相等：

```
1 // "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
2 letacuteQuestion="Voulez-vous un caé{E9}?"
3 // "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
4 letcombinedEAcuteQuestion="Voulez-vous un caé{65}\u{301}?"
5 ifeAcuteQuestion==combinedEAcuteQuestion{
6     print("These two strings are considered equal")
7 }
8 // prints "These two strings are considered equal"
9
10
```

反而，LATIN CAPITAL LETTER A (U+0041,或者说 "A")在英语当中是不同于俄语的 CYRILLIC CAPITAL LETTER A (U+0410,或者说 "A")的。字符看上去差不多，但是它们拥有不同的语言意义：

```
1 letlatinCapitalLetterA:Character="\u{41}"
2 letcyrillicCapitalLetterA:Character="\u{0410}"
3 iflatinCapitalLetterA!=cyrillicCapitalLetterA{
4     print("These two characters are not equivalent")
5 }
6 // prints "These two characters are not equivalent"
7
8
```

注意

字符串和字符的比较在 Swift 中并不区分区域设置。

前缀和后缀相等性

要检查一个字符串是否拥有特定的字符串前缀或者后缀，调用字符串的 hasPrefix(_:) 和 hasSuffix(_:) 方法，它们两个都会接受一个 String 类型的实际参数并且返回一个布尔量值。

下边的栗子假设一个表示莎士比亚的《罗密欧与朱丽叶》前两场场景位置的字符串数组：

```
1 letromeoAndJuliet=[
2     "Act 1 Scene 1: Verona, A public place",
3     "Act 1 Scene 2: Capulet's mansion",
4     "Act 1 Scene 3: A room in Capulet's mansion",
5     "Act 1 Scene 4: A street outside Capulet's mansion",
6     "Act 1 Scene 5: The Great Hall in Capulet's mansion",
7     "Act 2 Scene 1: Outside Capulet's mansion",
8     "Act 2 Scene 2: Capulet's orchard",
9     "Act 2 Scene 3: Outside Friar Lawrence's cell",
10    "Act 2 Scene 4: A street in Verona",
11    "Act 2 Scene 5: Capulet's mansion",
12    "Act 2 Scene 6: Friar Lawrence's cell"
13 ]
```

你可以使用 hasPrefix(_:) 方法操作 romeoAndJuliet 数组来计算第一场场景的数量：

```
1 varact1SceneCount=0
2 for scene in romeoAndJuliet{
3     if scene.hasPrefix("Act 1"){
4         act1SceneCount+=1
5     }
6 }
7 print("There are \(act1SceneCount) scenes in Act 1")
8 // Prints "There are 5 scenes in Act 1"
```

同样的，使用 `hasSuffix(_)`方法来计算与 Capulet's mansion 和 Friar Lawrence's cell 两个地方相关的场景数量：

```
1 varmansionCount=0
2 varcellCount=0
3 for scene in romeoAndJuliet{
4     if scene.hasSuffix("Capulet's mansion"){
5         mansionCount+=1
6     }else if scene.hasSuffix("Friar Lawrence's cell"){
7         cellCount+=1
8     }
9 }
10 print("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
11 // Prints "6 mansion scenes; 2 cell scenes"
```

注意

如同字符串和字符相等性一节所描述的那样，`hasPrefix(_)`和`hasSuffix(_)`方法只对字符串当中的每一个扩展字形集群之间进行了一个逐字符的规范化相等比较。

字符串的 Unicode 表示法

当一个 Unicode 字符串写入文本文档或者其他储存里边的时候，这个字符串的 Unicode 标量会被编码为一个或者一系列 Unicode 定义的编码格式。每一种格式都把字符串编码成所谓码元的小块。这些包括 UTF-8 编码格式（它把字符串以8 码元编码），UTF-16 编码格式（它把字符串按照 16位 码元 编码），以及 UTF-32 编码格式（它把字符串以32位码元编码）。

Swift 提供了几种不同的方法来访问字符串的 Unicode 表示。你可以使用 `for-in`语句来遍历整个字符串，来访以 Unicode 扩展字形集群的方式访问单独的 `Character`值。这个过程在操作字符串章节有着详细的描述。

或者，你也可以用以下三者之一的其他 Unicode 兼容表示法来访问 `String`值：

- UTF-8 码元的集合（关联于字符串的 `utf8` 属性）
- UTF-16 码元的集合（关联于字符串的 `utf16` 属性）
- 21位 Unicode 标量值的集合，等同于字符串的 UTF-32 编码格式（关联于字符串的 `unicodeScalars` 属性）

下边的每一个栗子都展示了接下来的字符串的不同表示方法，这个字符串由字符 D ， o ， g ， !! (DOUBLEEXCLAMATION MARK, 或者说 Unicode 标量 U+203C)以及 ? 字符(DOG FACE ,或者说 Unicode 标量 U+1F436)组成：

```
1 let dogString="Dog!!@?"
```

UTF-8 表示法

你可以通过遍历 `utf8` 属性来访问一个 `String` 的 UTF-8 表示法。这个属性的类型是 `String.UTF8View`，它是非负 8 位 (`UInt8`) 值，在字符串的 UTF-8 表示法中每一个字节的内容：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	🐶 U+1F436					
UTF-8 Code Unit	68	111	103	226	128	188	240	159	144	182
Position	0	1	2	3	4	5	6	7	8	9

```
1 for codeUnit in dogString.utf8{  
2   print("(codeUnit) ", terminator: "")  
3 }  
4 print("")  
5 // 68 111 103 226 128 188 240 159 144 182
```

上文中的栗子，前三个十进制 `codeUnit` 值 (68, 111, 103) 表示了字符 D, o, 和 g，它们的 UTF-8 表示法与它们的 ASCII 表示法相同。接下来的三个十进制 `codeUnit` 值 (226, 128, 188) 是 DOUBLEEXCLAMATION MARK 字符的三字节 UTF-8 表示法。最后四个 `codeUnit` 值 (240, 159, 144, 182) 是 DOG FACE 字符的四字节 UTF-8 表示法。

UTF-16 表示法

你可以通过遍历 `utf16` 属性来访问 `String` 的 UTF-16 表示法。这个属性的类型是 `String.UTF16View`，它是非负 16 位 (`UInt16`) 值，在字符串 UTF-16 表示法中每一个 16 位 的内容：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	🐶 U+1F436		
UTF-16 Code Unit	68	111	103	8252	55357	56374	
Position	0	1	2	3	4	5	

```

1 forcodeUnit indogString.utf16{
2   print("\\"(codeUnit) ",terminator:"")
3 }
4 print("")
5 // Prints "68 111 103 8252 55357 56374 "

```

再一次，前三个 codeUnit值 (68, 111, 103) 表示了字符 D , o , 和 g ，它们的 UTF-16 码元与字符串 UTF-8 表示法中的值相同 (因为这些 Unicode 标量表示 ASCII 字符) 。

第四个 codeUnit值(8252)是与十六进制值 203C相等的十进制数字，它表示了 DOUBLEEXCLAMATION MARK字符的 Unicode 标量 U+203C。这个字符可以在 UTF-16 中表示为单个码元了。

第五和第六个 codeUnit值 (55357和 56374)是 UTF-16 16位码元对表示的 DOG FACE字符。这些值是高16位码元值 U+D83D (十进制值为 55357) 和低16位码元值 U+DC36 (十进制值为 56374) 。

Unicode 标量表示法

你可以通过遍历 `unicodeScalars`属性来访问 String值的 Unicode 标量表示法。这个属性的类型是 `UnicodeScalarView`，它是 `UnicodeScalar`类型值的合集。

每一个 `UnicodeScalar`都有值属性可以返回一个标量的21位值，用 `UInt32`值表示：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	 U+1F436
Unicode Scalar Code Unit	68	111	103	8252	128054
Position	0	1	2	3	4

```

1 forscalar indogString.unicodeScalars{
2   print("\\"(scalar.value) ",terminator:"")
3 }
4 print("")
5 // Prints "68 111 103 8252 128054 "

```

前三个 `UnicodeScalar`值的 `value`属性 (68, 111, 103) 还是表示了字符 D, o, 和 g 。

第四个 codeUnit值 (8252)还是等于十六进制值 203C的十进制值，它表示了 DOUBLEEXCLAMATION MARK字符的 Unicode 标量 U+203C。

第五个和最后一个 `UnicodeScalar`的 `value`属性， 128054，是一个等于十六进制值 1F436的十进制数字，它表示了 DOG FACE字符的 Unicode 标量 U+1F436。

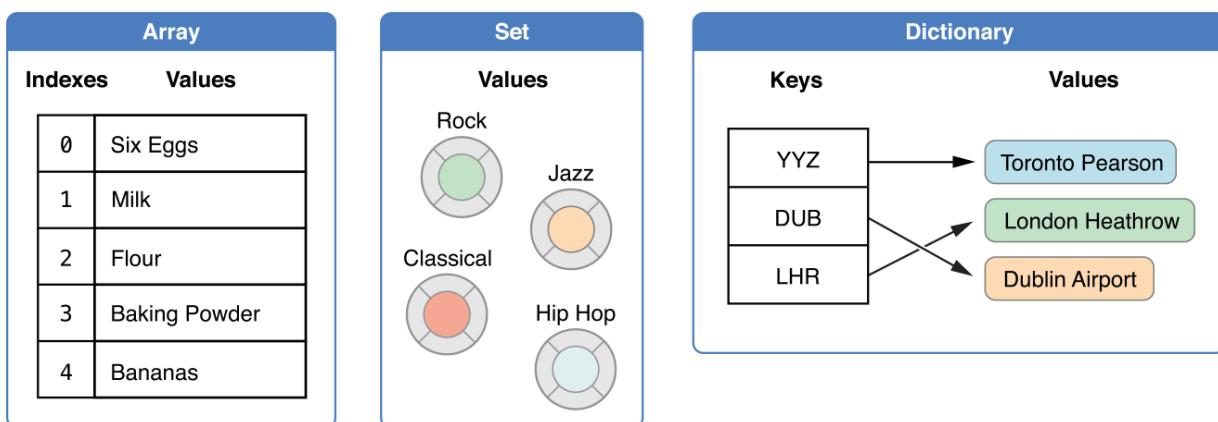
作为查询它们 value 属性的替代方法，每一个 UnicodeScalar 值同样可以用来构造新的 String 值，比如说使用字符串插值：

```
1 for scalar in dogString.unicodeScalars{  
2     print("\(scalar) ")  
3 }  
4 // D  
5 // o  
6 // g  
7 // !!  
8 // ☺
```

集合类型

 cnswift.org/collection-types

Swift 提供了三种主要的集合类型，所谓的数组、合集还有字典，用来储存值的集合。数组是有序的值的集合。合集是唯一值的无序集合。字典是无序的键值对集合。



Swift 中的数组、合集和字典总是明确能储存的值的类型以及它们能储存的键。就是说你不会意外地插入一个错误类型的值到集合中去。它同样意味着你可以从集合当中取回确定类型的值。

注意

Swift 的数组、合集和字典是以泛型集合实现的。要了解更多关于泛型类型和集合，参见[泛型](#)。

集合的可变性

如果你创建一个数组、合集或者一个字典，并且赋值给一个变量，那么创建的集合就是可变的。这意味着你随后可以通过添加、移除、或者改变集合中的元素来改变（或者说异变）集合。如果你把数组、合集或者字典赋值给一个常量，则集合就成了不可变的，它的大小和内容都不能被改变。

注意

在集合不需要改变的情况下创建不可变集合是个不错的选择。这样做可以允许 Swift 编译器优化你创建的集合的性能。

数组

数组以有序的方式来储存相同类型的值。相同类型的值可以在数组的不同地方多次出现。

注意

Swift 的 Array类型被桥接到了基础框架的 NSArray类上。

更多关于与基础框架和 Cocoa 一同使用 Array的信息，参考与 Cocoa 和 Objective-C 一起使用 Swift (Swift 3) (官方链接) 。

数组类型简写语法

Swift 数组的类型完整写法是 `Array<Element>`，`Element`是数组允许存入的值的类型。你同样可以简写数组的类型为 `[Element]`。尽管两种格式功能上相同，我们更推荐简写并且全书涉及到数组类型的时候都会使用简写。

创建一个空数组

你可以使用确定类型通过初始化器语法来创建一个空数组：

```
1 var someInts=[Int]()
2 print("someInts is of type [Int] with \(someInts.count) items.")
3 // prints "someInts is of type [Int] with 0 items."
```

注意 `someInts`变量的类型通过初始化器的类型推断为 `[Int]`。

相反，如果内容已经提供了类型信息，比如说作为函数的实际参数或者已经分类了的变量或常量，你可以通过空数组字面量来创建一个空数组，它写作`[]`（一对空方括号）：

```
1 someInts.append(3)
2 // someInts now contains 1 value of type Int
3 someInts=[]
4 // someInts is now an empty array, but is still of type [Int]
5
```

使用默认值创建数组

Swift 的 Array类型提供了初始化器来创建确定大小且元素都设定为相同默认值的数组。你可以传给初始化器对应类型的默认值（叫做 `repeating`）和新数组元素的数量（叫做 `count`）：

```
1 var threeDoubles=Array(repeating:0.0,count:3)
2 // threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

通过连接两个数组来创建数组

你可以通过把两个兼容类型的现存数组用加运算符（`+`）加在一起创建一个新数组。新数组的类型将从你相加的数组里推断出来：

```
1 var anotherThreeDoubles=Array(repeating:2.5,count:3)
2 // anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]
3 var sixDoubles=threeDoubles+anotherThreeDoubles
4 // sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
5
```

使用数组字面量创建数组

你同样可以使用数组字面量来初始化一个数组，它是一种以数组集合来写一个或者多个值的

简写方式。数组字面量写做一系列的值，用逗号分隔，用方括号括起来：

```
1 [value1,value2,value3]
```

下边的示例创建了一个叫做 shoppingList的数组来储存 String值：

```
1 varshoppingList:[String]=["Eggs","Milk"]
2 // shoppingList has been initialized with two initial items
```

shoppingList变量被声明为“字符串值的数组”写做 [String]。由于这个特定的数组拥有特定的 String值类型，它就只能储存 String值。这里， shoppingList被两个 String值（ "Eggs"和 "Milk"）初始化，写在字符串字面量里。

注意

数组 shoppingList被声明为变量（用 var提示符）而不是常量（用 let提示符）因为更多的元素会在下边的栗子中添加到数组当中。

在这种情况下，数组的字面量只包含两个 String值。这与 shoppingList变量声明类型一致（一个只能储存 String值的数组），因此用数组字面量作为 shoppingList以两个初始元素初始化的方式是被允许的。

依托于 Swift 的类型推断，如果你用包含相同类型值的数组字面量初始化数组，就不需要写明数组的类型。 shoppingList的初始化可以写得更短：

```
1 varshoppingList=["Eggs","Milk"]
```

因为数组字面量中的值都是相同的类型，Swift 就能够推断 [String]是 shoppingList变量最合适的数据类型。

访问和修改数组

你可以通过数组的方法和属性来修改数组，或者使用下标脚本语法。

要得出数组中元素的数量，检查只读的 count属性：

```
1 print("The shopping list contains \(shoppingList.count) items.")
2 // prints "The shopping list contains 2 items."
```

使用布尔量 isEmpty属性来作为检查 count属性是否等于 0的快捷方式：

```
1 ifshoppingList.isEmpty{
2     print("The shopping list is empty.")
3 }else{
4     print("The shopping list is not empty.")
5 }
6 // prints "The shopping list is not empty."
```

你可以通过 append(:)方法给数组末尾添加新的元素：

```
1 shoppingList.append("Flour")
2 // shoppingList now contains 3 items, and someone is making pancakes
```

另外，可以使用加赋值运算符 (+=) 来在数组末尾添加一个或者多个同类型元素：

```
1 shoppingList+=["Baking Powder"]
2 // shoppingList now contains 4 items
3 shoppingList+=["Chocolate Spread","Cheese","Butter"]
4 // shoppingList now contains 7 items
```

通过下标脚本语法来从数组当中取回一个值，在紧跟数组名后的方括号内传入你想要取回的值的索引：

```
1 var firstItem=shoppingList[0]
2 // firstItem is equal to "Eggs"
```

注意

数组中的第一个元素的索引为 0，不是 1。Swift 中的数组都是零开头的。

你可以使用下标脚本语法来改变给定索引中已经存在的值：

```
1 shoppingList[0]="Six eggs"
2 // the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

你同样可以使用下标脚本语法来一次改变一个范围的值，就算替换与范围长度不同的值的合集也行。下面的栗子替换用 "Bananas" 和 "Apples" 替换 "Chocolate Spread", "Cheese", and "Butter"：

```
1 shoppingList[4...6]=["Bananas","Apples"]
2 // shoppingList now contains 6 items
```

注意

你不能用下标脚本语法来追加一个新元素到数组的末尾。

要把元素插入到特定的索引位置，调用数组的 `insert(_:at:)` 方法：

```
1 shoppingList.insert("Maple Syrup",at:0)
2 // shoppingList now contains 7 items
3 // "Maple Syrup" is now the first item in the list
```

调用 `insert(_:at:)` 方法插入了一个新元素值为 "Maple Syrup" 到 shopping list 的最前面，通过明确索引位置为 0。

类似地，你可以使用 `remove(at:)` 方法来移除一个元素。这个方法移除特定索引的元素并且返回它（尽管你不需要的话可以无视返回的值）：

```
1 letmapleSyrup=shoppingList.remove(at:0)
2 // the item that was at index 0 has just been removed
3 // shoppingList now contains 6 items, and no Maple Syrup
4 // the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

注意

如果你访问或者修改一个超出数组边界索引的值，你将会触发运行时错误。你可以在使用索引前通过对比数组的 count 属性来检查它。除非当 count 为 0（就是说数组为空），否则最大的合法索引永远都是 count-1，因为数组的索引从零开始。

当数组中元素被移除，任何留下的空白都会被封闭，所以索引 0 的值再一次等于 "Six eggs"：

```
1 firstItem=shoppingList[0]
2 // firstItem is now equal to "Six eggs"
```

如果你想要移除数组最后一个元素，使用 removeLast() 方法而不是 remove(at:) 方法以避免查询数组的 count 属性。与 remove(at:) 方法相同，removeLast() 返回删除了的元素：

```
1 letapples=shoppingList.removeLast()
2 // the last item in the array has just been removed
3 // shoppingList now contains 5 items, and no apples
4 // the apples constant is now equal to the removed "Apples" string
```

遍历一个数组

你可以用 `for-in` 循环来遍历整个数组中值的合集：

```
1 foriteminshoppingList{
2   print(item)
3 }
4 // Six eggs
5 // Milk
6 // Flour
7 // Baking Powder
8 // Bananas
```

如果你需要每个元素以及值的整数索引，使用 `enumerated()` 方法来遍历数组。

`enumerated()` 方法返回数组中每一个元素的元组，包含了这个元素的索引和值。你可以分解元组为临时的常量或者变量作为遍历的一部分：

```
1 for(index,value)inshoppingList.enumerated(){
2   print("Item \u207b(index+1): \u207b(value)")
3 }
4 // Item 1: Six eggs
5 // Item 2: Milk
6 // Item 3: Flour
7 // Item 4: Baking Powder
8 // Item 5: Bananas
```

关于 `for-in` 循环的更多内容，见 [For-in 循环](#)。

合集^[1]

合集将同一类型且不重复的值无序地储存在一个集合当中。当元素的顺序不那么重要的时候你就可以使用合集来代替数组，或者你需要确保元素不会重复的时候。

注意

Swift 的 Set 类型桥接到了基础框架的 NSSet 类上。

更多关于与基础框架和 Cocoa 一起使用 Set 的信息，见 [与 Cocoa 和 Objective-C 一起使用 Swift \(Swift 3\)](#)。

Set 类型的哈希值

为了能让类型储存在合集当中，它必须是可哈希的——就是说类型必须提供计算它自身哈希值的方法。哈希值是 Int 值且所有的对比起来相等的对象都相同，比如 `a==b`，它遵循 `a.hashValue==b.hashValue`。

所有 Swift 的基础类型（比如 String, Int, Double, 和 Bool）默认都是可哈希的，并且可以用于合集或者字典的键。没有关联值的枚举成员值（如同枚举当中描述的那样）同样默认可哈希。

注意

你可以使用你自己自定义的类型作为合集的值类型或者字典的键类型，只要让它们遵循 Swift 基础库的 Hashable 协议即可。遵循 Hashable 协议的类型必须提供可获取的叫做 `hashValue` 的 Int 属性。通过 `hashValue` 属性返回的值不需要在同一个程序的不同的执行当中都相同，或者不同程序。

因为 Hashable 协议遵循 Equatable，遵循的类型必须同时一个“等于”运算符（`==`）的实现。Equatable 协议需要任何遵循 `==` 的实现都具有等价关系。就是说，`==` 的实现必须满足以下三个条件，其中 `a`, `b`, 和 `c` 是任意值：

- `a==a` (自反性)
- `a==b` 意味着 `b==a` (对称性)
- `a==b & b==c` 意味着 `a==c` (传递性)

更多对协议的遵循信息，见 [协议](#)。

合集类型语法

Swift 的合集类型写做 `Set<Element>`，这里的 `Element` 是合集要储存的类型。不同与数组，合集没有等价的简写。

创建并初始化一个空合集

你可以使用初始化器语法来创建一个确定类型的空合集：

```
1 var letters=Set<Character>()
2 print("letters is of type Set<Character> with \(letters.count) items.")
3 // prints "letters is of type Set<Character> with 0 items."
```

注意

letters变量的类型被推断为 Set<Character>，基于初始化器的类型。

另外，如果内容已经提供了类型信息，比如函数的实际参数或者已经分类的变量常量，你就可以用空的数组字面量来创建一个空合集：

```
1 letters.insert("a")
2 // letters now contains 1 value of type Character
3 letters=[]
4 // letters is now an empty set, but is still of type Set<Character>
```

使用数组字面量创建合集

你同样可以使用数组字面量来初始化一个合集，算是一种写一个或者多个合集值的快捷方式。

下边的栗子创建了一个叫做 favoriteGenres的合集来储存 String值：

```
1 varfavoriteGenres:Set<String>=["Rock","Classical","Hip hop"]
2 // favoriteGenres has been initialized with three initial items
```

favoriteGenres变量被声明为“ String值的合集”，写做 Set<String>。由于这个合集已经被明确值类型为 String，它只允许储存 String值。这时，合集 favoriteGenres用三个写在数组字面量中的 String值 ("Rock", "Classical", 和 "Hip hop")初始化。

注意

合集 favoriteGenres作为变量（用 var标记）而不是常量（用 let标记）是因为元素会在下边的栗子中添加和移除。

合集类型不能从数组字面量推断出来，所以 Set类型必须被显式地声明。总之，由于 Swift 的类型推断，你不需要在使用包含相同类型值的数组字面量初始化合集的时候写合集的类型。 favoriteGenres 的初始化可以写的更短一些：

```
1 varfavoriteGenres:Set=["Rock","Classical","Hip hop"]
```

由于数组字面量中所有的值都是相同类型的，Swift 就可以推断 Set<String>是 favoriteGenres变量的正确类型。

访问和修改合集

你可以通过合集的方法和属性来访问和修改合集。

要得出合集中元素的数量，检查它的只读 count属性：

```
1 print("I have \(favoriteGenres.count) favorite music genres.")
2 // prints "I have 3 favorite music genres."
```

使用布尔量 isEmpty属性作为检查 count属性是否等于 0的快捷方式：

```
1 if favoriteGenres.isEmpty{
2     print("As far as music goes, I'm not picky. ")
3 }else{
4     print("I have particular music preferences.")
5 }
6 // prints "I have particular music preferences."
```

你可通过调用 `insert(_)`方法来添加一个新的元素到合集：

```
1 favoriteGenres.insert("Jazz")
2 // favoriteGenres now contains 4 items
```

你可以通过调用合集的 `remove(_)`方法来从合集当中移除一个元素，如果元素是合集的成员就移除它，并且返回移除的值，如果合集没有这个成员就返回 nil。另外，合集当中所有的元素可以用 `removeAll()`一次移除。

```
1 if let removedGenre = favoriteGenres.remove("Rock"){
2     print("\(removedGenre)? I'm over it.")
3 }else{
4     print("I never much cared for that.")
5 }
6 // prints "Rock? I'm over it."
```

要检查合集是否包含了特定的元素，使用 `contains(_)`方法。

```
1 if favoriteGenres.contains("Funk"){
2     print("I get up on the good foot.")
3 }else{
4     print("It's too funky in here.")
5 }
6 // prints "It's too funky in here."
```

遍历合集

你可以在 `for-in`循环里遍历合集的值。

```
1 for genre in favoriteGenres{
2     print("\(genre)")
3 }
4 // Classical
5 // Jazz
6 // Hip hop
```

更多关于 `for-in`循环，见 [For-in 循环](#)。

Swift 的 `Set`类型是无序的。要以特定的顺序遍历合集的值，使用 `sorted()`方法，它把合集的元素作为使用 `<` 运算符排序了的数组返回。

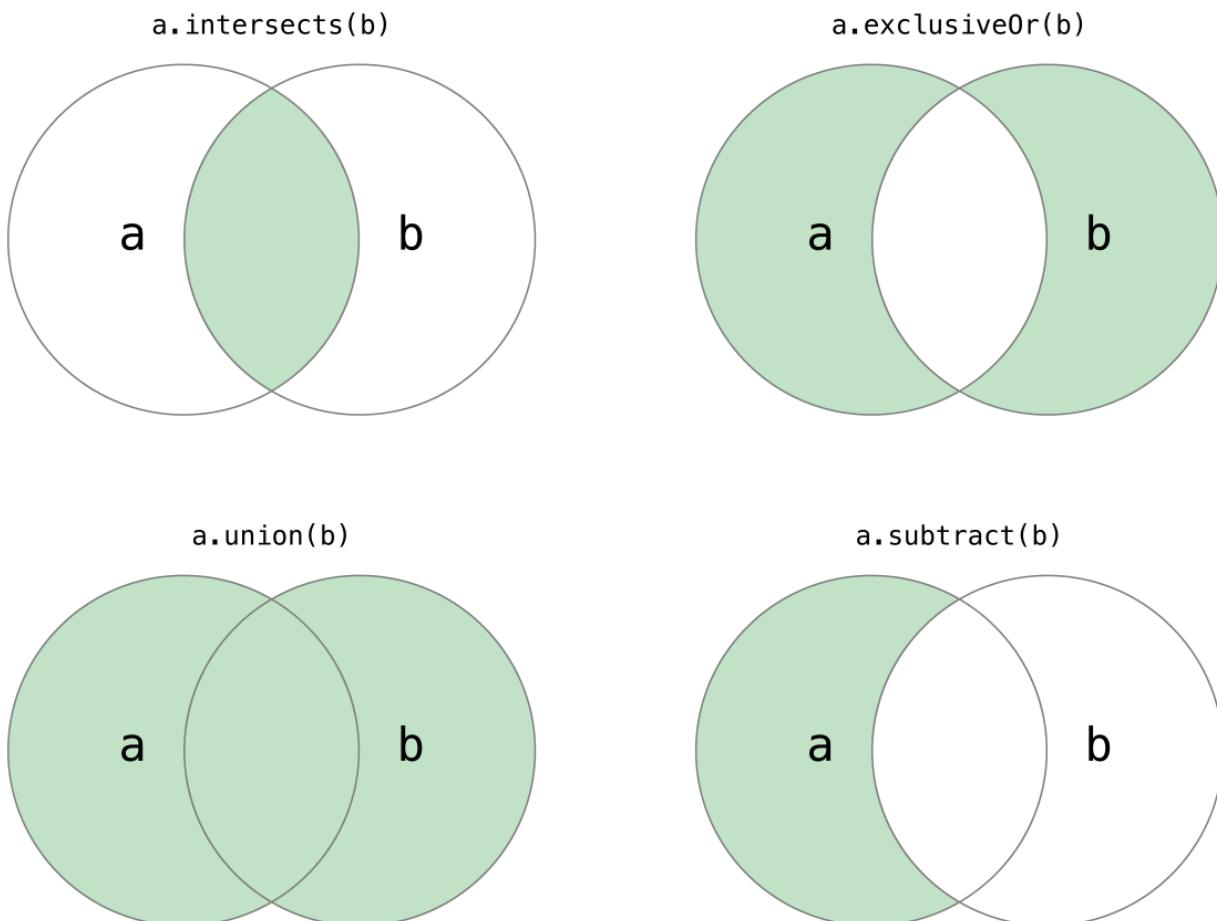
```
1 for genre in favoriteGenres.sorted(){
2     print("\(genre)")
3 }
4 // Classical
5 // Hip hop
6 // Jazz
```

执行合集操作

你可以高效地执行基本的合集操作，比如合并两个合集，确定两个合集共有哪个值，或者确定两个合集是否包含所有、某些或没有相同的值。

基本合集操作

下边的示例描述了两个合集——`a`和`b`——在各种合集操作下的结果，用阴影部分表示。

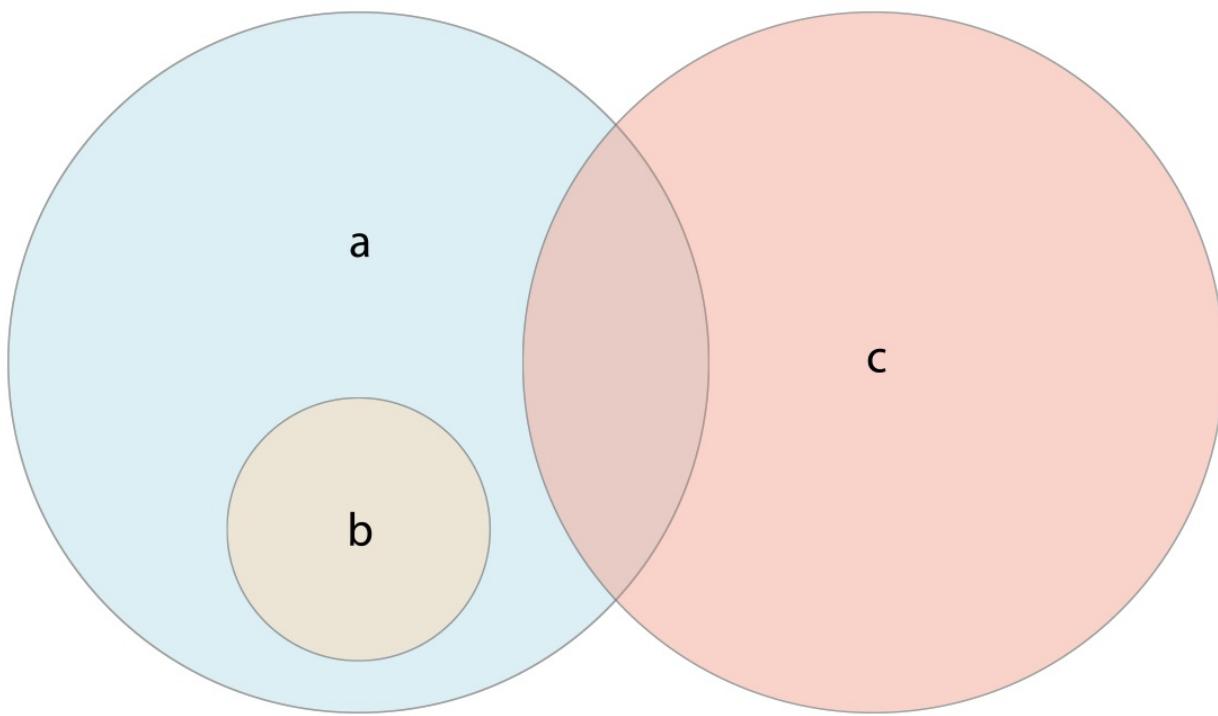


- 使用 `intersection(_)`方法来创建一个只包含两个合集共有值的新合集；
- 使用 `symmetricDifference(_)`方法来创建一个只包含两个合集各自有的非共有值的新合集；
- 使用 `union(_)`方法来创建一个包含两个合集所有值的新合集；
- 使用 `subtracting(_)`方法来创建一个两个合集当中不包含某个合集值的新合集。

```
1 letoddDigits:Set=[1,3,5,7,9]
2 letevenDigits:Set=[0,2,4,6,8]
3 letsingleDigitPrimeNumbers:Set=[2,3,5,7]
4 oddDigits.union(evenDigits).sorted()
5 // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
6 oddDigits.intersection(evenDigits).sorted()
7 // []
8 oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
9 // [1, 9]
10 oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
11 // [1, 2, 9]
12
```

合集成员关系和相等性

下面的示例描述了三个合集—— a， b和 c——用重叠区域代表合集之间值共享。合集 a是合集 b的超集，因为 a包含 b的所有元素。相反地，合集 b是合集 a的子集，因为 b的所有元素被 a包含。合集 b和合集 c是不相交的，因为他们的元素没有相同的。



- 使用“相等”运算符 (==)来判断两个合集是否包含有相同的值；
- 使用 isSubset(of:) 方法来确定一个合集的所有值是被某合集包含；
- 使用 isSuperset(of:)方法来确定一个合集是否包含某个合集的所有值；
- 使用 isStrictSubset(of:) 或者 isStrictSuperset(of:)方法来确定是个合集是否为某一个合集的子集或者超集，但并不相等；
- 使用 isDisjoint(with:)方法来判断两个合集是否拥有完全不同的值。

```
1 let houseAnimals:Set=["?", "?"]
2 let farmAnimals:Set=["?", "?", "?", "?", "?"]
3 let cityAnimals:Set=["?", "?"]
4 houseAnimals.isSubset(of:farmAnimals)
5 // true
6 farmAnimals.isSuperset(of:houseAnimals)
7 // true
8 farmAnimals.isDisjoint(with:cityAnimals)
9 // true
10
```

字典

字典储存无序的互相关联的同一类型的键和同一类型的值的集合。每一个值都与唯一的键相关联，它就好像这个值的身份标记一样。不同于数组中的元素，字典中的元素没有特定的顺序。当你需要查找基于特定标记的值的时候使用字典，很类似现实生活中字典用来查找特定字的定义。

注意

Swift 的 Dictionary桥接到了基础框架的 NSDictionary类。

更多关于与基础框架和 Cocoa 一起使用 Dictionary的信息，见与 Cocoa 和 Objective-C 一起使用 Swift (Swift 3) (官方链接)。

字典类型简写语法

Swift 的字典类型写全了是这样的：Dictionary<Key,Value>，其中的 Key是用来作为字典键的值类型，Value就是字典为这些键储存的值的类型。

注意

字典的 Key类型必须遵循 Hashable协议，就像合集的值类型。

你同样可以用简写的形式来写字典的类型为 [Key:Value]。尽管两种写法是完全相同的，但本书所有提及字典的地方都会使用简写形式。

创建一个空字典

就像数组，你可以用初始化器语法来创建一个空 Dictionary：

```
1 varnamesOfIntegers=[Int:String]()
2 // namesOfIntegers is an empty [Int: String] dictionary
```

这个栗子创建了类型为 [Int:String]的空字典来储存整数的可读名称。它的键是 Int类型，值是 String类型。

如果内容已经提供了信息，你就可以用字典字面量创建空字典了，它写做 [:]（在一对方括号里写一个冒号）：

```
1 namesOfIntegers[16]="sixteen"
2 // namesOfIntegers now contains 1 key-value pair
3 namesOfIntegers=[:]
4 // namesOfIntegers is once again an empty dictionary of type [Int: String]
```

用字典字面量创建字典

你同样可以使用 字典字面量 来初始化一个字典，它与数组字面量看起来差不多。字典字面量是写一个或者多个键值对为 Dictionary集合的快捷方式。

键值对由一个键和一个值组合而成，每个键值对里的键和值用冒号分隔。键值对写做一个列表，用逗号分隔，并且最终用方括号把它们括起来：

[key1:value1,key2:value2,key3:value3]

下边的栗子创建了一个储存国际机场名称的字典。这个字典中，键是三个字母的国际航空运输协会代码，值是机场的名字：

```
1 varairports:[String:String]=["YYZ":"Toronto Pearson","DUB":"Dublin"]
```

airports字典被声明为 [String:String]类型，它意思是“一个键和值都是 String的 Dictionary”。

注意

airports字典被声明为变量（使用 var标记），而不是常量（使用 let标记），因为下边的栗子还有给它添加更多机场。

字典 airports 用包含两个键值对的字典字面量初始化。第一对有 "YYZ"的键和 "Toronto Pearson"的值。第二对时 "DUB" 的键和 "Dublin"的值。

这个字典字面量包含两个 String:String对。这个键值对类型与 airports变量的声明类型相同（一个使用 String键并且只储存 String值的字典），所以赋值的这个字典字面量让 airports字典用两个初始元素初始化。

与数组一样，如果你用一致类型的字典字面量初始化字典，就不需要写出字典的类型了。airports的初始化就能写的更短：

```
1 varairports=["YYZ":"Toronto Pearson","DUB":"Dublin"]
```

由于字面量中所有的键都有相同的类型，同时所有的值也是相同的类型，Swift 可以推断 [String:String]就是 airports字典的正确类型。

访问和修改字典

你可以通过字典自身的方法和属性来访问和修改它，或者通过使用下标脚本语法。

如同数组，你可以使用 count只读属性来找出 Dictionary拥有多少元素：

```
1 print("The airports dictionary contains \(airports.count) items.")  
2 // prints "The airports dictionary contains 2 items."
```

使用布尔量 isEmpty属性作为检查 count属性是否等于 0的快捷方式：

```
1 ifairports.isEmpty{  
2 print("The airports dictionary is empty.")  
3 }else{  
4 print("The airports dictionary is not empty.")  
5 }  
6 // prints "The airports dictionary is not empty."
```

你可以用下标脚本给字典添加新元素。使用正确类型的新键作为下标脚本的索引，然后赋值一个正确类型的值：

```
1 airports["LHR"]="London"  
2 // the airports dictionary now contains 3 items
```

你同样可以使用下标脚本语法来改变特定键关联的值：

```
1 airports["LHR"]="London Heathrow"  
2 // the value for "LHR" has been changed to "London Heathrow"
```

作为下标脚本的代替，使用字典的 `updateValue(_:forKey:)`方法来设置或者更新特点键的值。就像上边下标脚本的栗子，`updateValue(_:forKey:)`方法会在键没有值的时候设置一个值，或者在键已经存在的时候更新它。总之，不同于下标脚本，`updateValue(_:forKey:)`方法在执行更新之后返回旧的值。这允许你检查更新是否成功。

`updateValue(_:forKey:)`方法返回一个字典值类型的可选项值。比如对于储存 String 值的字典来说，方法会返回 `String?`类型的值，或者说“可选的 String”。这个可选项包含了键的旧值如果更新前存在的话，否则就是 `nil`：

```
1 if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB"){
2     print("The old value for DUB was \(oldValue).")
3 }
4 // prints "The old value for DUB was Dublin."
```

你同样可以使用下标脚本语法来从字典的特点键中取回值。由于可能请求的键没有值，字典的下标脚本返回可选的字典值类型。如果字典包含了请求的键的值，下标脚本就返回一个包含这个键的值的可选项。否则，下标脚本返回 `nil`：

```
1 if let airportName = airports["DUB"]{
2     print("The name of the airport is \(airportName).")
3 } else{
4     print("That airport is not in the airports dictionary. ")
5 }
6 // prints "The name of the airport is Dublin Airport."
```

你可以使用下标脚本语法给一个键赋值 `nil` 来从字典当中移除一个键值对：

```
1 airports["APL"] = "Apple International"
2 // "Apple International" is not the real airport for APL, so delete it
3 airports["APL"] = nil
4 // APL has now been removed from the dictionary
```

另外，使用 `removeValue(forKey:)` 来从字典里移除键值对。这个方法移除键值对如果他们存在的话，并且返回移除的值，如果值不存在则返回 `nil`：

```
1 if let removedValue = airports.removeValue(forKey: "DUB"){
2     print("The removed airport's name is \(removedValue).")
3 } else{
4     print("The airports dictionary does not contain a value for DUB. ")
5 }
6 // Prints "The removed airport's name is Dublin Airport."
```

遍历字典

你可以用 `for-in` 循环来遍历字典的键值对。字典中的每一个元素返回为 `(key,value)` 元组，你可以解元组成员到临时的常量或者变量作为遍历的一部分：

```
1 for(airportCode, airportName) in airports{
2     print("\(airportCode): \(airportName)")
3 }
4 // YYZ: Toronto Pearson
5 // LHR: London Heathrow
```

更多关于 `for-in` 循环，见 [For-in 循环](#)。

你同样可以通过访问字典的 keys 和 values 属性来取回可遍历的字典的键或值的集合：

```
1 for airportCode in airports.keys{  
2     print("Airport code: \(airportCode)")  
3 }  
4 // Airport code: YYZ  
5 // Airport code: LHR  
6 for airportName in airports.values{  
7     print("Airport name: \(airportName)")  
8 }  
9 // Airport name: Toronto Pearson  
10 // Airport name: London Heathrow  
11
```

如果你需要和接收 Array 实例的 API 一起使用字典的键或值，就用 keys 或 values 属性来初始化一个新数组：

```
1 let airportCodes=[String](airports.keys)  
2 // airportCodes is ["YYZ", "LHR"]  
3 let airportNames=[String](airports.values)  
4 // airportNames is ["Toronto Pearson", "London Heathrow"]
```

Swift 的 Dictionary 类型是无序的。要以特定的顺序遍历字典的键或值，使用键或值的 sorted() 方法。

译注：

[1] 合集：此处 Set 应为“集合”，但为了与文章标题“Collection”做区别，故改写为合集。

控制流

 cn.swift.org/control-flow

Swift 提供所有多样化的控制流语句。包括 while 循环来多次执行任务； if , guard 和 switch 语句来基于特定的条件执行不同的代码分支；还有比如 break 和 continue 语句来传递执行流到你代码的另一个点上。

Swift 同样添加了 for-in 循环，它让你更简便地遍历数组、字典、范围和其他序列。

Swift 的 switch 语句同样比 C 中的对应语句多了不少新功能。比如说 Swift 中的 switch 语句不再“贯穿”到下一个情况当中，这就避免了 C 中常见的 break 语句丢失问题。情况可以匹配多种模式，包括间隔匹配，元组和特定的类型。switch 中匹配的值还能绑定到临时的常量和变量上供情况下代码使用，并且可以为每一个情况写 where 分句表达式来应用复杂条件匹配。

For-in 循环

使用 for-in 循环来遍历序列，比如一个范围的数字，数组中的元素或者字符串中的字符。

这个例子使用 for-in 循环来遍历数组中的元素：

```
1 letnames=["Anna","Alex","Brian","Jack"]
2 fornameinnames{
3   print("Hello, \(name)!")
4 }
5 // Hello, Anna!
6 // Hello, Alex!
7 // Hello, Brian!
8 // Hello, Jack!
```

你同样可以遍历字典来访问它的键值对。当字典遍历时，每一个元素都返回一个 (key,value) 元组，你可以在 for-in 循环体中使用显式命名常量来分解 (key,value) 元组成员。这时，字典的键就分解到了叫做 animalName 的常量中，而字典的值被分解到了 legCount 的常量中：

```
1 letnumberOfLegs=[{"spider":8,"ant":6,"cat":4}
2 for(animalName,legCount)innumberOfLegs{
3   print("\(animalName)s have \(legCount) legs")
4 }
5 // ants have 6 legs
6 // cats have 4 legs
7 // spiders have 8 legs
```

Dictionary 中的元素没有必要按照它们写入的顺序遍历出来。Dictionary 的内容内在无序，并且不在取回遍历时保证有序。需要注意的是，你给 Dictionary 插入元素的次序并不能代表你遍历时候的顺序。更多关于数组和字典，见集合类型。

for-in 循环同样能遍历数字区间。这个栗子打印了乘五表格的前几行：

```
1 for index in 1...5{  
2   print("\(index) times 5 is \(index*5)")  
3 }  
4 // 1 times 5 is 5  
5 // 2 times 5 is 10  
6 // 3 times 5 is 15  
7 // 4 times 5 is 20  
8 // 5 times 5 is 25
```

被遍历的序列是 1 到 5 的数字范围，包含这两个数，使用闭区间运算符 (...)。index 的值被设置为范围中的第一个数字 (1)，并且循环内的语句被执行了。这个栗子里，循环只包含了一个语句，它打印乘五表格中 index 的当前值。在语句执行之后，index 的值就更新到了范围中的第二个值 (2)，并且 print(_:separator:terminator:) 函数被再一次调用。这个过程会一直持续到范围结束。

在上面的栗子当中，index 是一个常量，它的值在每次遍历循环开始的时候被自动地设置。因此，它不需要在使用之前声明。它隐式地在循环的声明中声明了，不需要再用 let 声明关键字。

如果你不需要序列的每一个值，你可以使用下划线来取代遍历名以忽略值。

```
1 let base=3  
2 let power=10  
3 var answer=1  
4 for _ in 1...power{  
5   answer*=base  
6 }  
7 print("\(base) to the power of \(power) is \(answer)")  
8 // prints "3 to the power of 10 is 59049"
```

这个栗子计算一个数字的指数幂（这里为 3 的 10 次方）值。它以 1（就是说，3 的 0 次幂）乘 3 开始，十次，使用闭区间，从 1 开始到 10 为止。这样计算不需要计数器记录每次循环的值——它只需要以正确的次数执行循环就行了。下划线字符 _（在循环变量那里使用的那个）导致单个值被忽略并且不需要在每次遍历循环中提供当前值的访问。

在某些情况下，你可能不想要一个闭区间，它包含了区间两端的值。比如说给表盘上画分钟标记。你得画 60 个标记，从 0 分钟开始，使用半开区间运算符 (..<) 来包含最小值但不包含最大值。更多关于区间的内容，见[区间运算符](#)。

```
1 let minutes=60  
2 for tickMark in 0..<minutes{  
3   // render the tick mark each minute (60 times)  
4 }
```

有些用户可能想要在他们的 UI 上少来点分钟标记。比如说每 5 分钟一个标记吧。使用 stride(from:to:by:) 函数来跳过不想要的标记。

```
1 let minuteInterval=5  
2 for tickMark in stride(from:0,to:minutes,by:minuteInterval){  
3   // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)  
4 }
```

闭区间也同样适用，使用 stride(from:through:by:) 即可：

```

1 lethours=12
2 lethourInterval=3
3 for tickMark instride(from:3,through:hours,by:hourInterval){
4 // render the tick mark every 3 hours (3, 6, 9, 12)
5 }

```

While 循环

`while` 循环执行一个合集的语句指导条件变成 `false`。这种循环最好在第一次循环之后还有未知数量的遍历时使用。Swift 提供了两种 `while` 循环：

- `while` 在每次循环开始的时候计算它自己的条件；
- `repeat-while` 在每次循环结束的时候计算它自己的条件。

While

`while` 循环通过判断单一的条件开始。如果条件为 `true`，语句的合集就会重复执行直到条件变为 `false`。

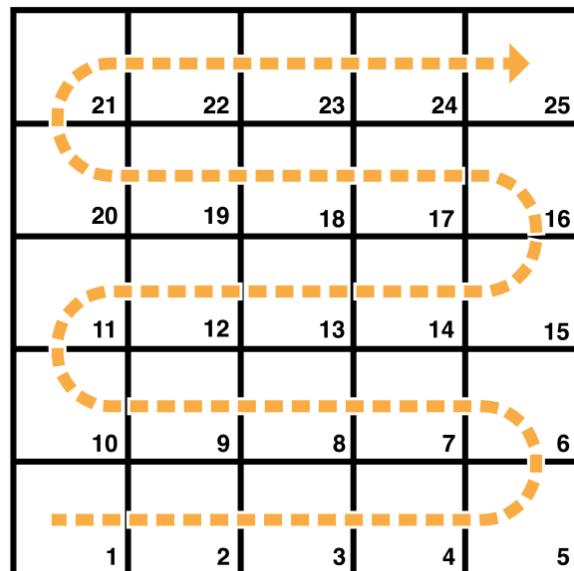
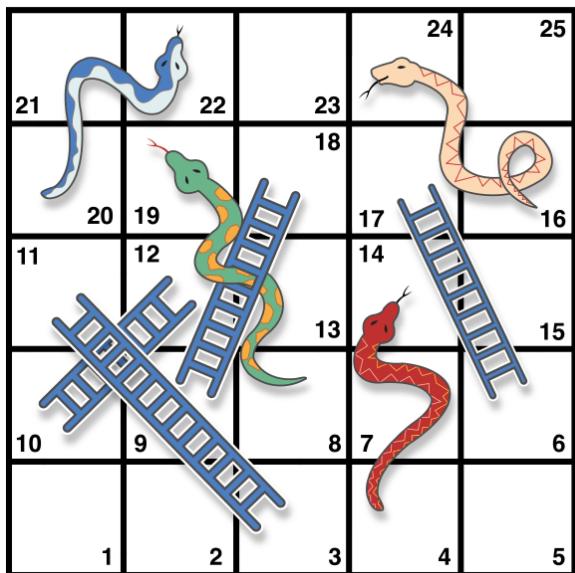
这里是一个 `while` 循环的通用格式：

```

1 while condition{
2 statements
3 }
4
5

```

这是一个玩蛇与梯子（也叫滑梯与梯子）的简单栗子：



下边是游戏的规则：

- 棋盘拥有 25 个方格，目标就是到达或者超过第 25 号方格；
- 每一次，你扔一个六面色子，安装方格的数字移动，依据水平的线路，如图安装上边虚线箭头标注的路线；
- 如果你停留在了梯子的下边，你就可以顺着梯子爬上去；
- 如果你停留在了蛇的头上，你就要顺着蛇滑下来。

游戏棋盘用 Int 值的数组来表现。它的大小基于一个叫做 finalSquare 的常量，它被用来初始化数组同样用来检测稍后的胜利条件。棋盘使用 26 个零 Int 值初始化，而不是 25 个（从 0 到 25）：

```
1 let finalSquare=25
2 var board=[Int](repeating:0,count:finalSquare+1)
```

有些方格随后设置为拥有更多特定的值比如蛇和梯子。有梯子的方格有一个正数来让你移动到棋盘的上方，因此有蛇的方格有一个负数来让你从棋盘上倒退：

```
1 board[03]=+08;board[06]=+11;board[09]=+09;board[10]=+02
2 board[14]=-10;board[19]=-11;board[22]=-02;board[24]=-08
```

方格 3 包含了一个梯子的底部，它把你移动到 11 号方格。要表达这个， board[03] 等于 +08 ，它等价于整数值 8 （如同 3 和 11 ）。一元加运算符（ +i ）是为了与一元减运算符（ -i ）保持一致，并且所有小于 10 的数字都要用零补齐（这两种做法都不是强制必须的，但它们让代码更加整洁，方便你开启处女座模式。）

玩家从“零格”开始，它正好是棋盘的左下角。第一次扔色子总会让玩家上到棋盘上去：

```
1 var square=0
2 var diceRoll=0
3 while square<finalSquare{
4     // roll the dice
5     diceRoll+=1
6     if diceRoll==7{diceRoll=1}
7     // move by the rolled amount
8     square+=diceRoll
9     if square<board.count{
10        // if we're still on the board, move up or down for a snake or a ladder
11        square+=board[square]
12    }
13 }
14 print("Game over!")
```

这个栗子使用了一个非常简单的算法来扔色子。比起随机生成数，它以 diceRoll 的 0 开始。每次通过 while 循环， diceRoll 增加一，然后检查看是否过大。无论何时这个返回的值等于 7 ，它变得过大，就把它重置到 1 .这给我们一个有序的 diceRoll 值，它永远是 1 ， 2 ， 3 ， 4 ， 5 ， 6 ， 1 ， 2 等等。

在扔色子之后，玩家根据 diceRoll 来在棋盘上移动。色子的值是有可能让玩家超出 25 号方格的，这时游戏结束。为了应付这种情况，代码在加储存在 board[square] 中的值到当前 square 值以让玩家移动之前检查 square 是否比 board 数组的 count 属性小。

如果这个检查没有执行， board[square] 就有可能尝试访问到超出 board 数组边界的值，这会触发错误。如果 square 现在等于 26 ，代码就会去尝试检查 board[26] 的值，它比数组的长度还大。

注意

这个检查还没有被执行，`board[square]` 可能会尝试访问超过 `board` 数组边界的值，这就会触发一个错误。如果 `square` 值为 26，代码将会尝试检查 `board[26]` 的值，这就会比数组的长度大一点。

当前的 `while` 循环执行结束，并且循环条件已经检查来看循环是否应该再次执行。如果玩家已经移到或者超出了第 25 号方格，循环评定为 `false`，游戏就结束了。

`while` 循环在这个情况当中合适是因为开始 `while` 循环之后游戏的长度并不确定。循环会一直执行下去直到特定的条件不满足。

Repeat-While

`while` 循环的另一种形式，就是所谓的 `repeat-while` 循环，在判断循环条件之前会执行一次循环代码块。然后会继续重复循环直到条件为 `false`。

注意

Swift 的 `repeat-while` 循环是与其他语言中的 `do-while` 循环类似的。

这里是 `repeat-while` 循环的通用形式：

```
1 repeat{  
2 statements  
3 }whilecondition
```

再次回顾蛇与梯子的栗子，使用 `repeat-while` 循环而不是 `while` 循环。`finalSquare`, `board`, `square`, 和 `diceRoll` 的值初始化的方式与 `while` 循环完全相同：

```
1 letfinalSquare=25  
2 varboard=[Int](repeating:0,count:finalSquare+1)  
3 board[03]=+08;board[06]=+11;board[09]=+09;board[10]=+02  
4 board[14]=-10;board[19]=-11;board[22]=-02;board[24]=-08  
5 varsquare=0  
6 vardiceRoll=0
```

在这个版本的游戏中，第一次循环中的动作是用来检查梯子或者蛇的。没有梯子能直接把玩家带到 25 格，因此不可能通过梯子赢得游戏。也就是说，在循环一开始就检查蛇还是梯子是安全的。

游戏一开始，玩家在“零格”。`board[0]` 总是等于 0 的，并且没有效果：

```
1 repeat{  
2 // move up or down for a snake or ladder  
3 square+=board[square]  
4 // roll the dice  
5 diceRoll+=1  
6 ifdiceRoll==7{diceRoll=1}  
7 // move by the rolled amount  
8 square+=diceRoll  
9 }whilesquare<finalSquare  
10 print("Game over!")
```

在检查是蛇还是梯子的代码之后，就是要色子了，玩家按照 diceRoll 数量的格数前进。当前循环执行结束。

循环条件 (`while square < finalSquare`) 与之前的相同，但是这次它会在第一次循环结束之后才会被判定。 repeat-while 循环的结构要比前边栗子里的 while 循环更适合这个游戏。在上边的 repeat-while 循环中， `square+=board[square]` 总是会在循环循环的 while 条件确定 `square` 仍在棋盘上之后立即执行。这个行为就去掉了早期游戏版本中对数组边界检查的需要。

条件语句

很多时候根据特定的条件来执行不同的代码是很有用的。你可能想要在错误发生时运行额外的代码，或者当值变得太高或者太低的时候显示一条信息。要达成这个目的，你可以让你的那部分代码有条件地执行。

Swift 提供了两种方法来给你的代码添加条件分支，就是所谓的 if 语句和 switch 语句。总的来说，你可以使用 if 语句来判定简单的条件，比如少量的可能性。 switch 语句则适合更复杂的条件，比如多个可能的组合，并且在模式匹配的情况下更加有用，可以帮你选择一段合适的代码分支来执行。

If

最简单的形式中， if 语句有着一个单一的 if 条件。它只会在条件为 true 的情况下才会执行语句的集合：

```
1 var temperatureInFahrenheit=30
2 if temperatureInFahrenheit<=32{
3     print("It's very cold. Consider wearing a scarf.")
4 }
5 // prints "It's very cold. Consider wearing a scarf."
```

先前的栗子检测了温度是否小于等于 32 华氏温度（水的冰点）。如果是，就打印一个信息。否则，没有信息打印，并且执行 if 语句的大括号后边的代码。

if 语句可以提供一个可选语句集，就是所谓的 else 分句，用来在 if 条件为 false 的时候使用。这些语句用 else 关键字明确：

```
1 temperatureInFahrenheit=40
2 if temperatureInFahrenheit<=32{
3     print("It's very cold. Consider wearing a scarf.")
4 }else{
5     print("It's not that cold. Wear a t-shirt.")
6 }
7 // prints "It's not that cold. Wear a t-shirt."
```

这两个分支至少会有一个被执行。因为温度增加到 40 华氏度，就不再冷的围围巾了，所以 else 分支就被激活了。

你可以链接多个 if 语句，来考虑额外的条件：

```
1 temperatureInFahrenheit=90
2 if temperatureInFahrenheit<=32{
3     print("It's very cold. Consider wearing a scarf.")
4 }elseif temperatureInFahrenheit>=86{
5     print("It's really warm. Don't forget to wear sunscreen.")
6 }else{
7     print("It's not that cold. Wear a t-shirt.")
8 }
9 // prints "It's really warm. Don't forget to wear sunscreen."
```

在这个栗子中，添加了一个额外的if语句来响应特定的温暖温度。最终的else分句保留并打印对其他任何不冷也不热的温度的响应。

最后的else分句是可选的，总之，如果条件集合不需要完成的话它可以被排除。

```
1 temperatureInFahrenheit=72
2 if temperatureInFahrenheit<=32{
3     print("It's very cold. Consider wearing a scarf.")
4 }elseif temperatureInFahrenheit>=86{
5     print("It's really warm. Don't forget to wear sunscreen.")
6 }
```

这个栗子当中，温度既不太冷也补太热，所以没有触发if或者else条件，最后就没有信息被打印出来。

Switch

switch 语句会将一个值与多个可能的模式匹配。然后基于第一个成功匹配的模式来执行合适的代码块。 switch 语句代替 if 语句提供了对多个潜在状态的响应。

在其自身最简单的格式中， switch 语句把一个值与一个或多个相同类型的值比较：

```
1 switch someValueToConsider{
2     case value1:
3         respond to value1
4     case value2,
5         value3:
6         respond to value2 or 3
7     default:
8         otherwise, do something else
9 }
```

每一个 switch 语句都由多个可能的情况组成，每一个情况都以 case 关键字开始。对于对比额外特定的值来说，Swift 提供了多种方法给每个情况来区别更复杂的匹配模式。这些选项会在本小节稍后的内容中详述。

每一个 switch 情况函数体都是独立的代码执行分支，与 if 语句的分支差不多。 switch 语句决定那个分支应该被选取。这就是所谓的在给定的值之间选择。

switch 语句一定得使全面的。就是说，给定类型里每一个值都得被考虑到并且匹配到一个 switch 情况。如果无法提供一个switch情况给所有可能的值，你可以定义一个默认匹配所有的情况来匹配所有未明确出来的值。这个匹配所有的情况用关键字 default 标记，并且必须在所有情况的最后出现。

这个示例使用了一个 switch 语句来考虑一个叫做 someCharacter 的单一小写字符：

```
1 let someCharacter:Character="z"
2 switch someCharacter{
3 case "a":
4     print("The first letter of the alphabet")
5 case "z":
6     print("The last letter of the alphabet")
7 default:
8     print("Some other character")
9 }
10 // Prints "The last letter of the alphabet"
```

switch 语句的第一个情况匹配英语字母表里的第一个字母，a，并且它的第二个情况匹配最后一个字母，z，由于 switch 必须拥有所有可能的字母的情况，而不是仅仅英语字母表里的字符，这个 switch 语句使用一个 default 情况来匹配所有其他非 a 和 z 的字符。这使得 switch 语句一定是全面的。

没有隐式贯穿

相比 C 和 Objective-C 里的 switch 语句来说，Swift 里的 switch 语句不会默认从每个情况的末尾贯穿到下一个情况里。相反，整个 switch 语句会在匹配到第一个 switch 情况执行完毕之后退出，不再需要显式的 break 语句。这使得 switch 语句比 C 的更安全和易用，并且避免了意外地执行多个 switch 情况。

注意

尽管 break 在 Swift 里不是必须的，你仍然可以使用 break 语句来匹配和忽略特定的情况，或者在某个情况执行完成之前就打断它。移步 [Switch 语句中的 Break](#) 来了解更多。

每一个情况的函数体必须包含至少一个可执行的语句。下面的代码就是不正确的，因为第一个情况是空的：

```
1 let anotherCharacter:Character="a"
2 switch anotherCharacter{
3 case "a":
4 case "A":
5     print("The letter A")
6 default:
7     print("Not the letter A")
8 }
9 // this will report a compile-time error
```

与 C 中的 switch 语句不同，这个 switch 语句没有同时匹配 "a" 和 "A"。相反它会导致一个编译时错误 case "a": 没有包含任何可执行语句。这可以避免意外地从一个情况贯穿到另一个情况中，并且让代码更加安全和易读。

在一个 switch 情况中匹配多个值可以用逗号分隔，并且可以写成多行，如果列表太长的话：

```
1 let anotherCharacter:Character="a"
2 switch anotherCharacter{
3 case "a", "A":
4     print("The letter A")
5 default:
6     print("Not the letter A")
7 }
8 // Prints "The letter A"
```

为了可读性，复合的情况同样可以写成多行。更多关于符合情况的信息，见复合情况。

注意

如同在贯穿中描述的那样，要在特定的 switch 情况中使用贯穿行为，使用 fallthrough 关键字。

区间匹配

switch情况的值可以在一个区间中匹配。这个栗子使用了数字区间来为语言中的数字区间进行转换：

```
1 let approximateCount=62
2 let countedThings="moons orbiting Saturn"
3 var naturalCount:String
4 switch approximateCount{
5 case 0:
6   naturalCount="no"
7 case 1..<5:
8   naturalCount="a few"
9 case 5..<12:
10  naturalCount="several"
11 case 12..<100:
12  naturalCount="dozens of"
13 case 100..<1000:
14  naturalCount="hundreds of"
15 default:
16  naturalCount="many"
17 }
18 print("There are \$(naturalCount) \$(countedThings).")
19 // prints "There are dozens of moons orbiting Saturn."
```

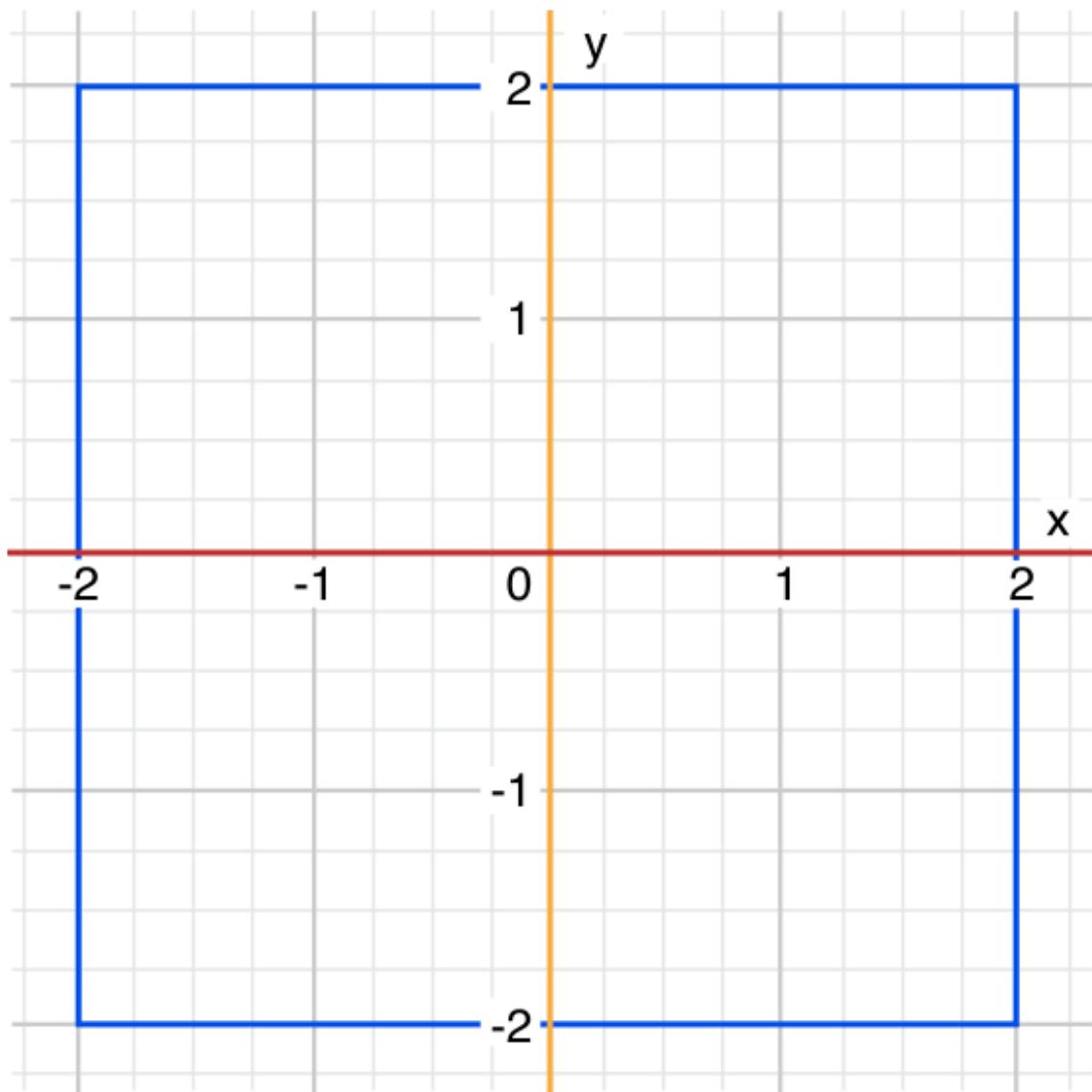
在上面的栗子中， approximateCount 在 switch 语句中进行评定。每个 case 都与数字或者区间进行对比。由于 approximateCount 的值在12和100之间， naturalCount 被赋值“dozens of”，并且执行结果传递出了 switch 语句。

元组

你可以使用元组来在一个 switch 语句中测试多个值。每个元组中的元素都可以与不同的值或者区间进行匹配。另外，使用下划线（_）来表明匹配所有可能的值。

下边的例子接收一个 (x,y) 点坐标，用一个简单的元组类型 (Int,Int) ，并且在后边显示在图片中：

```
1 let somePoint=(1,1)
2 switch somePoint{
3 case (0,0):
4   print("(0, 0) is at the origin")
5 case (_,0):
6   print("(\\(somePoint.0), 0) is on the x-axis")
7 case (0,_):
8   print("(0, \\(somePoint.1)) is on the y-axis")
9 case (-2...2,-2...2):
10  print("(\\(somePoint.0), \\(somePoint.1)) is inside the box")
11 default:
12  print("(\\(somePoint.0), \\(somePoint.1)) is outside of the box")
13 }
14 // prints "(1, 1) is inside the box"
```



switch 语句决定坐标是否在原点 (0,0)；在红色的 x 坐标轴；在橘黄色的 y 坐标轴；在蓝色的4乘4以原点为中心的方格里；或者在方格外边。

与 C 不同，Swift 允许多个 switch 情况来判断相同的值。事实上，坐标 (0,0) 可能匹配这个例子中所有四个情况，第一个匹配到的情况会被使用。坐标 (0,0) 将会最先匹配 case(0,0)，所以接下来的所有再匹配到的情况将被忽略。

值绑定

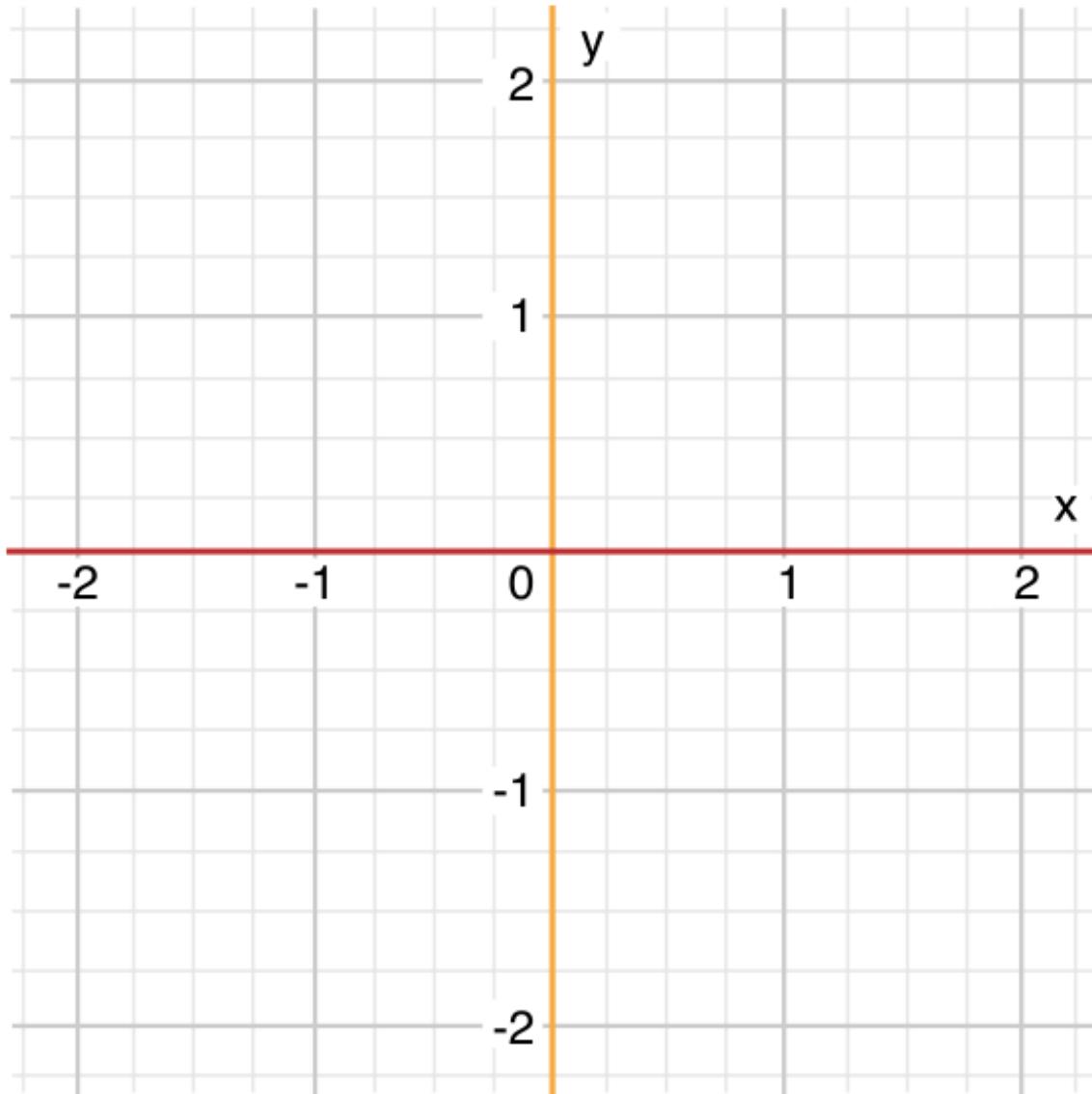
switch 情况可以将匹配到的值临时绑定为一个常量或者变量，来给情况的函数体使用。这就是所谓的值绑定，因为值是在情况的函数体里“绑定”到临时的常量或者变量的。

下边的栗子接收一个 (x,y) 坐标，使用 (Int,Int) 元组类型并且在下边的图片里显示：

```

1 letanotherPoint=(2,0)
2 switchanotherPoint{
3     case(letx,0):
4         print("on the x-axis with an x value of \(\(x)\)")
5     case(0,lety):
6         print("on the y-axis with a y value of \(\(y)\)")
7     caselet(x,y):
8         print("somewhere else at (\(\(x), \(\(y))")
9 }
10 // prints "on the x-axis with an x value of 2"

```



switch 语句决定坐标是否在在红色的x坐标轴，在橘黄色的y坐标轴；还是其他地方；或不在坐标轴上。

三个 switch 情况都使用了常量占位符 `x` 和 `y`，它会从临时 `anotherPoint` 获取一个或者两个元组值。第一个情况，`case(letx,0)`，匹配任何 `y` 的值是 0 并且赋值坐标的 `x` 到临时常量 `x` 里。类似地，第二个情况，`case(0,lety)`，匹配让后 `x` 值是 0 并且把 `y` 的值赋值给临时常量 `y`。

在临时常量被声明后，它们就可以在情况的代码块里使用。这里，它们用来输出点的分类。

注意这个 switch 语句没有任何的 default 情况。最后的情况，`caselet(x,y)`，声明了一个带

有两个占位符常量的元组，它可以匹配所有的值。结果，它匹配了所有剩下的值，然后就不需要 default 情况来让 switch 语句穷尽了。

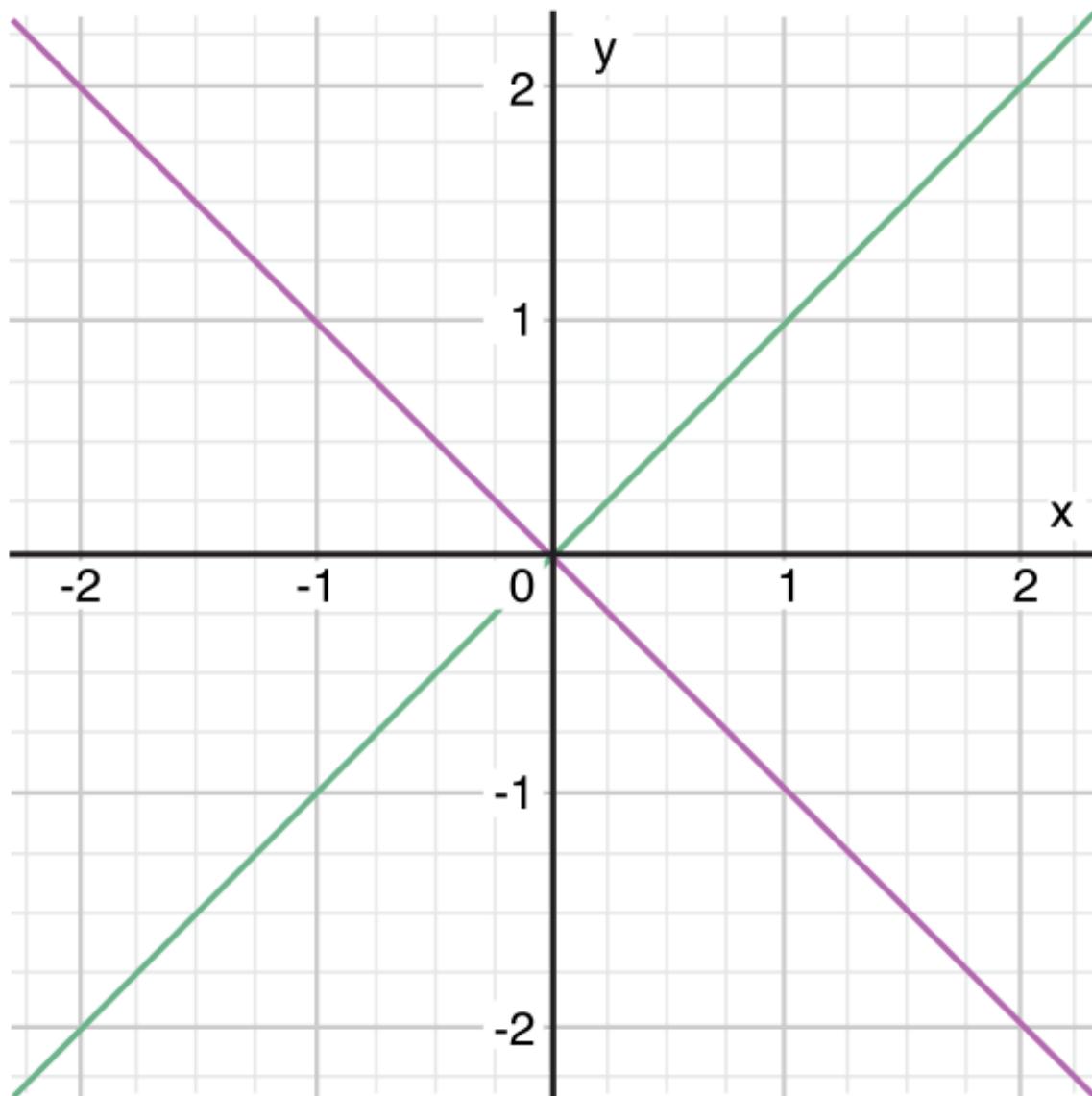
在上边的栗子中，`x` 和 `y` 被 `let` 关键字声明为常量，因为它们没有必要在情况体内被修改。总之，它们也可以用变量来声明，使用 `var` 关键字。如果这么做，临时的变量就会以合适的值来创建并初始化。对这个变量的任何改变都只会在情况函数体内有效。

Where

switch 情况可以使用 where 分句来检查额外的情况。

下边的栗子划分 (x,y) 坐标到下边的图例中：

```
1 let yetAnotherPoint=(1,-1)
2 switch yetAnotherPoint{
3     case let(x,y) where x==y:
4         print("(\\(x), \\(y)) is on the line x == y")
5     case let(x,y) where x== -y:
6         print("(\\(x), \\(y)) is on the line x == -y")
7     case let(x,y):
8         print("(\\(x), \\(y)) is just some arbitrary point")
9     }
10 // prints "(1, -1) is on the line x == -y"
```



switch 语句决定坐标在绿色的斜线 $x==y$, 还是在紫色的斜线 $x==\text{-}y$, 或者都不是。

三个 switch 情况声明了占位符常量 x 和 y , 它从 `yetAnotherPoint` 临时接收两个元组值。这个常量使用 `where` 分句 , 来创建动态过滤。 switch 情况只有 `where` 分句情况评定等于 `true` 时才会匹配这个值。

和前边的栗子一样 , 最后的情况匹配了余下所有可能的值 , 所以不需要 `default` 情况这个 switch 也是全面的。

复合情况

多个 switch 共享同一个函数体的多个情况可以在 `case` 后写多个模式来复合 , 在每个模式之间用逗号分隔。如果任何一个模式匹配了 , 那么这个情况都会被认为是匹配的。如果模式太长 , 可以把它们写成多行 , 比如说 :

```
1 letsomeCharacter:Character="e"
2 switchsomeCharacter{
3     case"a","e","i","o","u":
4         print("(someCharacter) is a vowel")
5     case"b","c","d","f","g","h","j","k","l","m",
6     "n","p","q","r","s","t","v","w","x","y","z":
7         print("\(someCharacter) is a consonant")
8     default:
9         print("\(someCharacter) is not a vowel or a consonant")
10    }
11 // Prints "e is a vowel"
```

这个 switch 语句的第一个情况匹配了英语语言里所有五个小写的元音。类似的 , 第二个情况匹配了英语语言里所有的辅音。最终 , `default` 情况匹配其他任意字符。

复合情况同样可以包含值绑定。所有复合情况的模式都必须包含相同的值绑定集合 , 并且复合情况中的每一个绑定都得有相同的类型格式。这才能确保无论复合情况的那部分匹配了 , 接下来的函数体中的代码都能访问到绑定的值并且值的类型也都相同。

```
1 letstillAnotherPoint=(9,0)
2 switchstillAnotherPoint{
3     case(letdistance,0),(0,letdistance):
4         print("On an axis, \(distance) from the origin")
5     default:
6         print("Not on an axis")
7    }
8 // Prints "On an axis, 9 from the origin"
```

上边的 `case` 拥有两个模式 : `(letdistance,0)` 匹配 x 轴的点以及 `(0,letdistance)` 匹配 y 轴的点。两个模式都包含一个 `distance` 的绑定并且 `distance` 在两个模式中都是整形——也就是说这个 `case` 函数体的代码一定可以访问 `distance` 的值。

控制转移语句

控制转移语句在你代码执行期间改变代码的执行顺序 , 通过从一段代码转移控制到另一段。 Swift 拥有五种控制转移语句 :

- `continue`
- `break`

- `fallthrough`

- `return`

- `throw`

`continue` , `break` , 和 `fallthrough` 语句在下边有详细描述。 `return` 语句在函数中描述，还有 `throw` 语句在使用抛出函数传递错误中描述。

Continue

`continue` 语句告诉循环停止正在做的事情并且再次从头开始循环的下一次遍历。它是说“我不再继续当前的循环遍历了”而不是离开整个的循环。

注意

在一个包含条件和自增器的 `for` 循环中，循环的自增器仍然会在调用 `continue` 语句后评定。循环自身还是会和往常一样工作；只有循环体中的代码被跳过。

下面的栗子移除了所有小写字符串中的元音和空格来创建一个谜之语句：

```
1 letpuzzleInput="great minds think alike"
2 varpuzzleOutput=""
3 forcharacter inpuzzleInput.characters{
4     switchcharacter{
5         case"a","e","i","o","u"," ":
6             continue
7         default:
8             puzzleOutput.append(character)
9     }
10 }
11 print(puzzleOutput)
12 // prints "grtmndsthnklk"
```

上面的代码在匹配到元音或者空格的时候调用了 `continue` 关键字，导致遍历的当前循环立即结束并直接跳到了下一次遍历的开始。这个行为使得 `switch` 代码块匹配（和忽略）只有元音和空格的字符，而不是请求匹配每一个要打印的字符。

Break

`break` 语句会立即结束整个控制流语句。当你想要提前结束 `switch` 或者循环语句或者其他情况时可以在 `switch` 语句或者循环语句中使用 `break` 语句。

循环语句中的 Break

当在循环语句中使用时，`break` 会立即结束循环的执行，并且转移控制到循环结束花括号（`}`）后的第一行代码上。当前遍历循环里的其他代码都不会被执行，并且余下的遍历循环也不会开始了。

Switch 语句里的 Break

当在 `switch` 语句里使用时，`break` 导致 `switch` 语句立即结束它的执行，并且转移控制到 `switch` 语句结束花括号（`}`）之后的第一行代码上。

这可以用来在一个 switch 语句中匹配和忽略一个或者多个情况。因为 Swift 的 switch 语句是穷尽且不允许空情况的，所以有时候有必要故意匹配和忽略一个匹配到的情况以让你的意图更加明确。要这样做的话你可以通过把 break 语句作为情况的整个函数体来忽略某个情况。当这个情况通过 switch 语句匹配到了，情况中的 break 语句会立即结束 switch 语句的执行。

注意

switch 的情况如果只包含注释的话会导致编译时错误。注释不是语句，并且不会导致 switch 情况被忽略。要使用 break 语句来忽略 switch 情况。

下面的栗子匹配一个 Character 值并且决定它表示四种语言中哪种语言的数字符号。简明起见，多个值被覆盖在了一个 switch 情况中：

```
1 let numberSymbol:Character="☰"// Simplified Chinese for the number 3
2 var possibleIntegerValue:Int?
3 switch numberSymbol{
4 case "1", "່", "一", "၁":
5     possibleIntegerValue=1
6 case "2", "້", "二", "၂":
7     possibleIntegerValue=2
8 case "3", "໌", "三", "၃":
9     possibleIntegerValue=3
10 case "4", "ໍ", "四", "၄":
11     possibleIntegerValue=4
12 default:
13     break
14 }
15 if let integerValue=possibleIntegerValue{
16     print("The integer value of \(numberSymbol) is \(integerValue).")
17 }else{
18     print("An integer value could not be found for \(numberSymbol).")
19 }
20 // prints "The integer value of ☰ is 3."
```

这个栗子检测 numberSymbol 来确定它是拉丁语，阿拉伯语，中文还是泰语的 1 到 4。如果匹配到，其中一个 switch 语句的情况赋值一个可选的 Int? 变量叫做 possibleIntegerValue 为一个合适的整数值。

在 switch 语句完成其执行后，栗子使用了可选绑定来确定是否有值。

possibleIntegerValue 变量作为可选类型一开始拥有一个隐式初始值 nil，所以因此可选绑定只有在 possibleIntegerValue 被 switch 语句前四个情况之一设定了实际存在的值之后才会成功。

在上边的例子中，列举所有可能的 Character 值是不实际的，所以 default 情况就提供了一个匹配所有没有匹配到的字符的功能。这个 default 情况不需要执行任何动作，所以因此就写了一个 break 语句作为函数体。一旦 default 情况匹配到了，break 语句结束 switch 语句的执行，然后代码从 iflet 语句继续执行。

Fallthrough

Swift 中的 Switch 语句不会从每个情况的末尾贯穿到下一个情况中。相反，整个 switch 语句会在第一个匹配到的情况执行完毕之后就直接结束执行。比较而言，C 你在每一个 switch 情况末尾插入显式的 break 语句来阻止贯穿。避免默认贯穿意味着 Swift 的 switch 语句比 C 更加清晰和可预料，并且因此它们避免了意外执行多个 switch 情况。

如果你确实需要 C 风格的贯穿行为，你可以选择在每个情况末尾使用 `fallthrough` 关键字。下面的栗子使用了 `fallthrough` 来创建一个数字的文字描述：

```
1 letintegerToDescribe=5
2 vardescription="The number \(integerToDescribe) is"
3 switchintegerToDescribe{
4 case2,3,5,7,11,13,17,19:
5 description+=" a prime number, and also"
6 fallthrough
7 default:
8 description+=" an integer."
9 }
10 print(description)
11 // prints "The number 5 is a prime number, and also an integer."
```

这个栗子声明了一个新的 `String` 变量叫做 `description` 并且赋值给它一个初始值。然后函数使用一个 `switch` 语句来判断 `integerToDescribe`。如果 `integerToDescribe` 是一个列表中的质数，函数就在 `description` 的末尾追加文字，来标记这个数字是质数。然后它使用 `fallthrough` 关键字来“贯穿到” `default` 情况。`default` 情况添加额外的文字到描述的末尾，接着 `switch` 语句结束。

如果 `integerToDescribe` 不在已知质数列表中，它就不会匹配第一个 `switch` 情况。然后也没用其他特定的情况，所以 `integerToDescribe` 匹配了默认的 `default` 情况。

在 `switch` 语句完成执行之后，数字的描述使用 `print(_:separator:terminator:)` 函数打印出来。在这个例子中，数字 5 被正确地分辨为一个质数。

注意

`fallthrough` 关键字不会为 `switch` 情况检查贯穿入情况的条件。`fallthrough` 关键字只是使代码执行直接移动到下一个情况（或者 `default` 情况）的代码块中，就像 C 的标准 `switch` 语句行为一样。

给语句打标签

你可以内嵌循环和条件语句到其他循环和条件语句当中以在 Swift 语言中创建一个复杂的控制流结构。总之，循环和条件语句都可以使用 `break` 语句来提前结束它们的执行。因此，显式地标记那个循环或者条件语句是你想用 `break` 语句结束的就很有必要。同样的，如果你有多个内嵌循环，显式地标记你想让 `continue` 语句生效的是哪个循环就很有必要了。

要达到这些目的，你可以用语句标签来给循环语句或者条件语句做标记。在一个条件语句中，你可以使用一个语句标签配合 `break` 语句来结束被标记的语句。在循环语句中，你可以使用语句标签来配合 `break` 或者 `continue` 语句来结束或者继续执行被标记的语句。

通过把标签作为关键字放到语句开头来用标签标记一段语句，后跟冒号。这里是一个对 `while` 循环使用标签的栗子，这个原则对所有的循环和 `switch` 语句来说都相同：

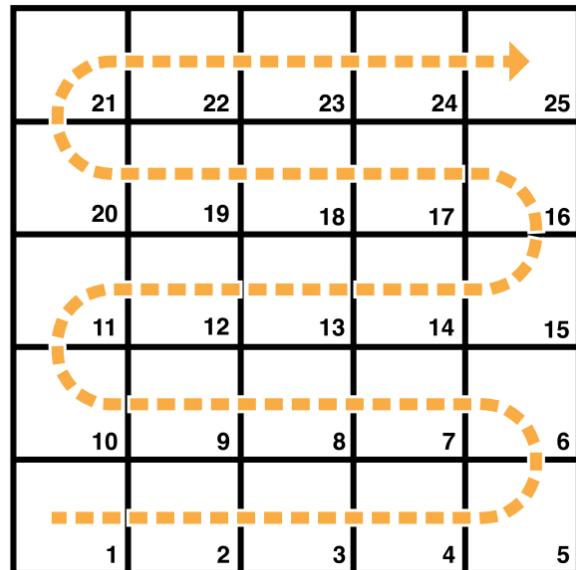
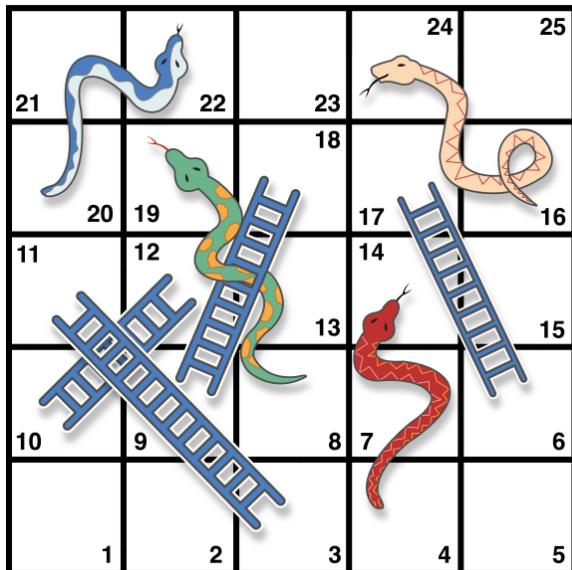
```
1 label name:whilecondition{
2 statements
3 }
```

下边的栗子为你之前章节看过的蛇与梯子游戏做了修改，在 `while` 循环中使用了标签来配合 `break` 和 `continue` 语句。这次，这个游戏有一个额外的规则：

要赢得游戏，你必须精确地落在第25格上。

如果特定的点数带你超过了第25格，你必须再次掷色子直到你恰好得到了落到第25格的点数。

游戏棋盘与之前的一样：



finalSquare，board，square 和 diceRoll 的值也用和之前一样的方式来初始化：

```
1 letfinalSquare=25
2 varboard=[Int](count:finalSquare+1,repeatedValue:0)
3 board[03]=+08;board[06]=+11;board[09]=+09;board[10]=+02
4 board[14]=-10;board[19]=-11;board[22]=-02;board[24]=-08
5 varsquare=0
6 vardiceRoll=0
```

这个版本的游戏使用了一个 while 循环和一个 switch 语句来实现游戏的逻辑。while 循环有一个叫做 gameLoop 的标签，来表明它是蛇与梯子游戏的主题循环。

while 循环条件是 while square!=finalSquare，用来反映你必须精确地落在第25格上：

```
1 gameLoop:whilesquare!=finalSquare{
2   diceRoll+=1
3   ifdiceRoll==7{diceRoll=1}
4   switchsquare+diceRoll{
5     casefinalSquare:
6       // diceRoll will move us to the final square, so the game is over
7       breakgameLoop
8     caseletnewSquare wherenewSquare>finalSquare:
9       // diceRoll will move us beyond the final square, so roll again
10      continuegameLoop
11    default:
12      // this is a valid move, so find out its effect
13      square+=diceRoll
14      square+=board[square]
15    }
16  }
17  print("Game over!")
```

每次循环，都会扔色子。使用一个 switch 语句来考虑移动的结果而不是立即移动玩家，然后如果移动允许的话就工作：

- 如果扔的色子将把玩家移动到最后的方格，游戏就结束。 breakgameLoop 语句转移控制到 while 循环外的第一行代码上，它会结束游戏。
- 如果扔的色子点数将会把玩家移动超过最终的方格，那么移动就是不合法的，玩家就需要再次扔色子。 continuegameLoop 语句就会结束当前的 while 循环遍历并且开始下一次循环的遍历。
- 在其他所有的情况下，色子是合法的。玩家根据 diceRoll 的方格数前进，并且游戏的逻辑会检查蛇和梯子。然后循环结束，控制返回到 while 条件来决定是否要再次循环。

注意

如果上边的 break 语句不使用 gameLoop 标签，它就会中断 switch 语句而不是 while 语句。使用 gameLoop 标签使得要结束那个控制语句变得清晰明了。

同时注意当调用 continuegameLoop 来跳入下一次循环并不是强制必须使用 gameLoop 标签的。游戏里只有一个循环，所以 continue 对谁生效是不会有关的。总之，配合 continue 使用 gameLoop 也无伤大雅。一直在 break 语句里写标签会让游戏的逻辑更加清晰和易读。

提前退出

guard 语句，类似于 if 语句，基于布尔值表达式来执行语句。使用 guard 语句来要求一个条件必须是真才能执行 guard 之后的语句。与 if 语句不同， guard 语句总是有一个 else 分句—— else 分句里的代码会在条件不为真的时候执行。

```

1 funcgreet(person:[String:String]){
2   guard letname=person["name"]else{
3     return
4   }
5   print("Hello \((name)!")
6   guard letlocation=person["location"]else{
7     print("I hope the weather is nice near you.")
8     return
9   }
10  print("I hope the weather is nice in \(location).")
11 }
12 greet(["name":"John"])
13 // prints "Hello John!"
14 // prints "I hope the weather is nice near you."
15 greet(["name":"Jane","location":"Cupertino"])
16 // prints "Hello Jane!"
17 // prints "I hope the weather is nice in Cupertino."
18
19
20
21

```

如果 guard 语句的条件被满足，代码会继续执行直到 guard 语句后的花括号。任何在条件中使用可选项绑定而赋值的变量或者常量在 guard 所在的代码块中随后的代码里都是可用的。

如果这个条件没有被满足，那么在 else 分支里的代码就会被执行。这个分支必须转移控制结束 guard 所在的代码块。要这么做可以使用控制转移语句比如 return ， break ， continue 或者 throw ，或者它可以调用一个不带有返回值的函数或者方法，比如

`fatalError()`。

相对于使用 `if` 语句来做同样的事情，为需求使用 `guard` 语句来提升你代码的稳定性。它会让正常地写代码而不用把它们包裹进 `else` 代码块，并且它允许你保留在需求之后处理危险的需求。

检查API的可用性

Swift 拥有内置的对 API 可用性的检查功能，它能够确保你不会悲剧地使用了对部属目标不可用的 API。

编译器在 SDK 中使用可用性信息来确保在你项目中明确的 API 都是可用的。如果你尝试使用一个不可用的 API 的话，Swift 会在编译时报告一个错误。

你可以在 `if` 或者 `guard` 语句中使用一个 **可用性条件** 来有条件地执行代码，基于在运行时你想用的那个 API 是可用的。当验证在代码块中的 API 可用性时，编译器使用来自可用性条件里的信息来检查。

```
1 if#available(iOS10,macOS10.12,*){  
2 // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS  
3 }else{  
4 // Fall back to earlier iOS and macOS APIs  
5 }
```

上边的可用性条件确定了在 iOS 平台，`if` 函数体只在 iOS 10 及以上版本才会执行；对于 macOS 平台，在只有在 macOS 10.12 及以上版本才会运行。最后一个实际参数，`*`，它需求并表明在其他所有平台，`if` 函数体执行你在目标里明确的最小部属。

在这个通用的格式中，可用性条件接收平台的名称和版本列表。你可以使用 iOS，macOS 和 watchOS 来作为平台的名字。要说明额外的特定主版本号则使用类似 iOS 8 这样的名字，你可以明确更小一点的版本号比如 iOS 8.3 和 macOS 10.10.3.

```
1 if#available(platform name version,...,*){  
2 statements to execute if the APIs are available  
3 }else{  
4 fallback statements to execute if the APIs are unavailable  
5 }
```

函数

 cnswift.org/functions

函数是一个独立的代码块，用来执行特定的任务。通过给函数一个名字来定义它的功能，并且在需要的时候，通过这个名字来“调用”函数执行它的任务。

Swift 统一的函数语法十分灵活，可以表达从简单的无形式参数的 C 风格函数到复杂的每一个形式参数都带有局部和外部形式参数名的 Objective-C 风格方法的任何内容。形式参数能提供一个默认的值来简化函数的调用，也可以被当作输入输出形式参数被传递，它在函数执行完成时修改传递来的变量。

Swift 中的每一个函数都有类型，由函数的形式参数类型和返回类型组成。你可以像 Swift 中其他类型那样来使用它，这使得你能够方便的将一个函数当作一个形式参数传递到另外的一个函数中，也可以在一个函数中返回另一个函数。函数同时也可以写在其他函数内部来在内嵌范围封装有用的功能。

定义和调用函数

当你定义了一个函数的时候，你可以选择定义一个或者多个命名的分类的值作为函数的输入（所谓的形式参数），并且/或者定义函数完成后将要传回作为输出的值的类型（所谓它的返回类型）。

每一个函数都有一个函数名，它描述函数执行的任务。要使用一个函数，你可以通过“调用”函数的名字并且传入一个符合函数形式参数类型的输入值（所谓实际参数）来调用这个函数。给函数提供的实际参数的顺序必须符合函数的形式参数列表顺序。

下边示例中的函数叫做 `greet(person:)`，跟它的功能一致——它接收一个人的名字作为输入然后返回对这个人的问题。要完成它，你需要定义一个输入形式参数——一个叫做 `person` 的 `String` 类型值——并且返回一个 `String` 类型，它将会包含对这个人的问题：

```
1 funcgreet(person:String)->String{  
2     letgreeting="Hello, "+person+"!"  
3     returngreeting  
4 }
```

这些信息都被包含在了函数的定义中，它使用一个 `func` 的关键字前缀。你可以用一个返回箭头 `->` (一个连字符后面跟一个向右的尖括号) 来明确函数返回的类型。

定义描述了函数会做什么，接收什么和它结束的时候会返回什么。定义能够帮助你更容易的从你代码的其他地方准确的调用到函数：

```
1 print(greet(person:"Anna"))  
2 // Prints "Hello, Anna!"  
3 print(greet(person:"Brian"))  
4 // Prints "Hello, Brian!"
```

你可以通过在 person 标签后边给函数 greet(person:) 传入一个用圆括号包裹的 String 实际参数值来调用它，例如 greet(person:"Anna")。如同上边示例中展示的一样，因为函数返回一个 String 值，所以 greet(person:) 可以包裹在 print(_:separator:terminator:) 函数中来打印字符串并查看它的返回值。

注意

函数 print(_:separator:terminator:) 的第一个实际参数并没有标签，并且它的其他实际参数是可选的，是因为他们都有默认值。这些函数语法的变化在下边函数实际参数标签和形式参数名以及默认形式参数值（此处应有链接）小节中讨论。

函数 greet(person:) 的主体从定义一个新的叫做 greeting 的 String 常量开始，它被设置成简单问候信息。之后这个问候被 return 关键字传递出函数。一旦执行到 return greeting 这句代码，函数就会结束执行并返回 greeting 的当前值。

你可以多次调用 greet(person:) 并给它传入不同的值，上面的栗子演示了我们用 "Anna" 值和 "Brian" 值作为输入值调用函数会发生什么。函数为每种情况定制了一个问候语。

为了简化这个函数的主体，我们将创建信息和返回语句组合到一行：

```
1 funcgreetAgain(person:String)->String{  
2     return"Hello again, "+person+"!"  
3 }  
4 print(greetAgain(person:"Anna"))  
5 // Prints "Hello again, Anna!"
```

函数的形式参数和返回值

在 Swift 中，函数的形式参数和返回值非常灵活。你可以定义从一个简单的只有一个未命名形式参数的工具函数到那种具有形象的参数名称和不同的形式参数选项的复杂函数之间的任何函数。

无形式参数的函数

函数没有要求必须输入一个参数，这里有一个没有输入形式参数的函数，无论何时它被调用永远会返回相同的 String 信息：

```
1 funcsayHelloWorld()->String{  
2     return"hello, world"  
3 }  
4 print(sayHelloWorld())  
5 // prints "hello, world"
```

函数的定义仍然需要在名字后边加一个圆括号，即使它不接受形式参数也得这样做。当函数被调用的时候也要在函数的名字后边加一个空的圆括号。

多形式参数的函数

函数可以输入多个形式参数，可以写在函数后边的圆括号内，用逗号分隔。

这个函数以一个人的名字以及是否被问候过为输入，并返回对这个人的相应的问候：

```
1 funcgreet(person:String,alreadyGreeted:Bool)->String{
2 ifalreadyGreeted{
3 returngreetAgain(person:person)
4 }else{
5 returngreet(person:person)
6 }
7 }
8 print(greet(person:"Tim",alreadyGreeted:true))
9 // Prints "Hello again, Tim!"
```

通过在圆括号中传入带有 person 标签的 String 实际参数值和带有 alreadyGreeted 标签的 Bool 实际参数值来调用 greet(person:alreadyGreeted:) 这个函数，实际参数之间用逗号分隔。注意这个函数与之前展示的函数 greet(person:) 是明显不同的。尽管两个函数都叫做 greet，greet(person:alreadyGreeted:) 接收两个实际参数但 greet(person:) 函数只接收一个。

无返回值的函数

函数定义中没有要求必须有一个返回类型。下面是另一个版本的 greet(person:) 函数，叫做 sayGoodbye(_:)，它将自己的 String 值打印了出来而不是返回它：

```
1 funcgreet(person:String){
2 print("Hello, \$(person)!")
3 }
4 greet(person:"Dave")
5 // Prints "Hello, Dave!"
```

正因为它不需要返回值，函数在定义的时候就没有包含返回箭头（->）或者返回类型。

注意

严格来讲，函数 greet(person:) 还是有一个返回值的，尽管没有定义返回值。没有定义返回类型的函数实际上会返回一个特殊的类型 Void。它其实是一个空的元组，作用相当于没有元素的元组，可以写作 ()。

当函数被调用时，函数的返回值可以被忽略：

```
1 funcprintAndCount(string:String)->Int{
2 print(string)
3 returnstring.characters.count
4 }
5 funcprintWithoutCounting(string:String){
6 let_=printAndCount(string:string)
7 }
8 printAndCount(string:"hello, world")
9 // prints "hello, world" and returns a value of 12
10 printWithoutCounting(string:"hello, world")
11 // prints "hello, world" but does not return a value
```

在第一个函数 printAndCount(_) 中，打印了一个字符串，然后返回了一个 Int 类型的字符数统计。在第二个函数 printWithoutCounting 中，调用了第一个函数，但却忽略了它的返回值。当调用第二个函数时，第一个函数仍然会打印出信息，但是返回的值却没有被使用。

注意

返回值可以被忽略，但是如果一个函数需要返回值的时候就必须返回。如果一个函数有定义的返回类型，没有返回值的话就不会继续运行到函数的末尾，尝试这么做的话会得到编译时错误。

多返回值的函数

为了让函数返回多个值作为一个复合的返回值，你可以使用元组类型作为返回类型。

下面的栗子定义了一个叫做 `minMax(array:)` 的函数，它可以找到类型为 `Int` 的数组中最大数字和最小数字。

```
1 funcminMax(array:[Int])->(min:Int,max:Int){  
2     varcurrentMin=array[0]  
3     varcurrentMax=array[0]  
4     forvalue inarray[1..<array.count]{  
5         ifvalue<currentMin{  
6             currentMin=value  
7         }elseifvalue>currentMax{  
8             currentMax=value  
9         }  
10    }  
11    return(currentMin,currentMax)  
12 }
```

函数 `minMax(array:)` 返回了一个包含两个 `Int` 值的元组。这两个值被 `min` 和 `max` 标记，这样当查询函数返回值的时候就可以通过名字访问了。

函数 `minMax(array:)` 的主体在给数组中的第一个整数的值设置两个名为 `currentMin` 和 `currentMax` 的操作变量的时候开始。然后函数遍历数组中剩下的值，并检查它们会不会比 `currentMin` 小或者比 `currentMax` 大。最后最终的最大值和最小值被当作一个包含两个 `Int` 值的元组被返回。

因为元组的成员值在函数的返回类型部分被命名，所以它们可以通过使用点语法取出最大值和最小值：

```
1 letbounds=minMax(array:[8,-6,2,109,3,71])  
2 print("min is \(bounds.min) and max is \(bounds.max)")  
3 // Prints "min is -6 and max is 109"
```

需要注意的是，元组的成员值不必在函数返回元组的时候命名，因为它们的名字早已经在函数的返回类型部分被明确。

可选元组返回类型

如果元组在函数的返回类型中有可能“没有值”，你可以用一个可选元组返回类型来说明整个元组的可能是 `nil`。书法是在可选元组类型的圆括号后边添加一个问号（`?`）例如 `(Int,Int)?` 或者 `(String,Int,Bool)?`。

注意

类似 `(Int,Int)?` 的可选元组类型和包含可选类型的元组 `(Int?,Int?)` 是不同的。对于可选元组类型，整个元组是可选的，而不仅仅是元组里边的单个值。

上面的函数 `minMax(array:)` 返回了一个包含两个 Int 值的元组。总之，函数不会对传入的数组进行安全性检查。如果 `array` 的实际参数包含了一个空的数组，上面定义的函数 `minMax(array:)` 在尝试调用 `array[0]` 的时候就会触发一个运行时错误。

为了安全的处理这种“空数组”的情景，就需要把 `minMax(array:)` 函数的返回类型写做可选元组，当数组是空的时候返回一个 `nil` 值：

```
1 funcminMax(array:[Int])->(min:Int,max:Int)?{
2 ifarray.isEmpty{returnnil}
3 varcurrentMin=array[0]
4 varcurrentMax=array[0]
5 forvalue inarray[1..<array.count]{
6 ifvalue<currentMin{
7 currentMin=value
8 }elseifvalue>currentMax{
9 currentMax=value
10 }
11 }
12 return(currentMin,currentMax)
13 }
```

你可以利用可选项绑定去检查这个版本的 `minMax(array:)` 函数返回了一个实际的元组值还是 `nil`。

```
1 ifletbounds=minMax(array:[8,-6,2,109,3,71]){
2 print("min is \(bounds.min) and max is \(bounds.max)")
3 }
4 // Prints "min is -6 and max is 109"
```

函数实际参数标签和形式参数名

每一个函数的形式参数都包含 **实际参数标签** 和 **形式参数名**。实际参数标签用在调用函数的时候；在调用函数的时候每一个实际参数前边都要写实际参数标签。形式参数名用在函数的实现当中。默认情况下，形式参数使用它们的形式参数名作为实际参数标签。

```
1 funcsomeFunction(firstParameterName:Int,secondParameterName:Int){
2 // In the function body, firstParameterName and secondParameterName
3 // refer to the argument values for the first and second parameters.
4 }
5 someFunction(firstParameterName:1,secondParameterName:2)
```

左右的形式参数必须有唯一的名字。尽管有可能多个形式参数拥有相同的真实参数标签，唯一的真实参数标签有助于让你的代码更加易读。

指定实际参数标签

在提供形式参数名之前写实际参数标签，用空格分隔：

```
1 funcsomeFunction(argumentLabel parameterName:Int){
2 // In the function body, parameterName refers to the argument value
3 // for that parameter.
4 }
```

注意

如果你为一个形式参数提供了实际参数标签，那么这个实际参数就必须在调用函数的时候使用标签。

这里有另一个函数 greet(person:) 的版本，接收一个人名字和家乡然后返回对这个的问候：

```
1 funcgreet(person:String,from hometown:String)->String{
2     return "Hello \$(person)! Glad you could visit from \$(hometown)."
3 }
4 print(greet(person:"Bill",from:"Cupertino"))
5 // Prints "Hello Bill! Glad you could visit from Cupertino."
```

实际参数标签的使用能够让函数的调用更加明确，更像是自然语句，同时还能提供更可读的函数体并更清晰地表达你的意图。

省略实际参数标签

如果对于函数的形式参数不想使用实际参数标签的话，可以利用下划线（_）来为这个形式参数代替显式的真实参数标签。

```
1 funcsomeFunction(_firstParameterName:Int,secondParameterName:Int){
2     // In the function body, firstParameterName and secondParameterName
3     // refer to the argument values for the first and second parameters.
4 }
5 someFunction(1,secondParameterName:2)
```

默认形式参数值

你可以通过在形式参数类型后给形式参数赋一个值来给函数的任意形式参数定义一个默认值。如果定义了默认值，你就可以在调用函数时候省略这个形式参数。

```
1 funcsomeFunction(parameterWithDefault:Int=12){
2     // In the function body, if no arguments are passed to the function
3     // call, the value of parameterWithDefault is 12.
4 }
5 someFunction(parameterWithDefault:6)// parameterWithDefault is 6
6 someFunction()// parameterWithDefault is 12
```

把不带有默认值的形式参数放到函数的形式参数列表中带有默认值的形式参数前边，不带有默认值的形式参数通常对函数有着重要的意义——把它们写在前边可以便于让人们看出来无论是否省略带默认值的形式参数，调用的都是同一个函数。

可变形式参数

一个可变形式参数可以接受零或者多个特定类型的值。当调用函数的时候你可以利用可变形式参数来声明形式参数可以被传入值的数量是可变的。可以通过在形式参数的类型名称后边插入三个点符号（...）来书写可变形式参数。

传入到可变参数中的值在函数的主体中被当作是对应类型的数组。举个栗子，一个可变参数的名字是 numbers 类型是 Double... 在函数的主体中它会被当作名字是 numbers 类型是 [Double] 的常量数组。

下面的栗子计算了一组任意长度的数字的算术平均值（也叫做平均数）。

```
1 func arithmeticMean(_numbers:Double...)->Double{
2     var total:Double=0
3     for number in numbers{
4         total+=number
5     }
6     return total/Double(numbers.count)
7 }
8 arithmeticMean(1,2,3,4,5)
9 // returns 3.0, which is the arithmetic mean of these five numbers
10 arithmeticMean(3,8,25,18,75)
11 // returns 10.0, which is the arithmetic mean of these three numbers
```

注意

一个函数最多只能有一个可变形式参数。

输入输出形式参数

就像上面描述的，可变形式参数只能在函数的内部做改变。如果你想函数能够修改一个形式参数的值，而且你想这些改变在函数结束之后依然生效，那么就需要将形式参数定义为输入输出形式参数。

在形式参数定义开始的时候在前边添加一个 `inout` 关键字可以定义一个输入输出形式参数。输入输出形式参数有一个能输入给函数的值，函数能对其进行修改，还能输出到函数外边替换原来的值。

你只能把变量作为输入输出形式参数的实际参数。你不能用常量或者字面量作为实际参数，因为常量和字面量不能修改。在将变量作为实际参数传递给输入输出形式参数的时候，直接在它前边添加一个和符号 (`&`) 来明确可以被函数修改。

注意

输入输出形式参数不能有默认值，可变形式参数不能标记为 `inout`，如果你给一个形式参数标记了 `inout`，那么它们也不能标记 `var` 和 `let` 了。

这里有一个 `swapTwoInts(_:_)` 函数，它有两个输入输出整数形式参数 `a` 和 `b`：

```
1 func swapTwoInts(_a:inoutInt,_b:inoutInt){
2     let temporaryA=a
3     a=b
4     b=temporaryA
5 }
```

函数 `swapTwoInts(_:_)` 只是简单的将 `b` 和 `a` 的值进行了调换。函数将 `a` 的值储存在临时常量 `temporaryA` 中，将 `b` 的值赋给 `a`，然后再将 `temporaryA` 的值赋给 `b`。

你可以通过两个 `Int` 类型的变量来调用函数 `swapTwoInts(_:_)` 去调换它们两个的值，需要注意的是 `someInt` 的值和 `anotherInt` 的值在传入函数 `swapTwoInts(_:_)` 时都添加了和符号。

```
1 var someInt=3
2 var anotherInt=107
3 swapTwoInts(&someInt,&anotherInt)
4 print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5 // prints "someInt is now 107, and anotherInt is now 3"
```

上边的栗子显示了 someInt 和 anotherInt 的原始值即使是在函数的外部定义的，也可被函数 swapTwoInts(_:_:) 修改。

注意

输入输出形式参数与函数的返回值不同。上边的 swapTwoInts 没有定义返回类型和返回值，但它仍然能修改 someInt 和 anotherInt 的值。输入输出形式参数是函数能影响到函数范围外的另一种替代方式。

函数类型

每一个函数都有一个特定的 **函数类型**，它由形式参数类型，返回类型组成。

举个栗子：

```
1 funcaddTwoInts(_a:Int,_b:Int)->Int{  
2     return a+b  
3 }  
4 funcmultiplyTwoInts(_a:Int,_b:Int)->Int{  
5     return a*b  
6 }
```

上边的栗子定义了两个简单的数学函数 addTwoInts 和 multiplyTwoInts。这两个函数每个传入两个 Int 值，返回一个 Int 值，就是函数经过数学运算得出的结果。

这两个函数的类型都是 (Int,Int)->Int。也读作：

“有两个形式参数的函数类型，它们都是 Int 类型，并且返回一个 Int 类型的值。”

下边的另外一个栗子，一个没有形式参数和返回值的函数。

```
1 funcprintHelloWorld(){  
2     print("hello, world")  
3 }
```

这个函数的类型是 ()->Void，或者“一个没有形式参数的函数，返回 Void。”

使用函数类型

你可以像使用 Swift 中的其他类型一样使用函数类型。例如，你可以给一个常量或变量定义一个函数类型，并且为变量指定一个相应的函数。

```
1 varmathFunction:(Int,Int)->Int=addTwoInts
```

这可以读作：

“定义一个叫做 mathFunction 的变量，它的类型是‘一个能接受两个 Int 值的函数，并返回一个 Int 值。’将这个新的变量指向 addTwoInts 函数。”

这个 addTwoInts(_:_:) 函数和 mathFunction 函数有相同的类型，所以这个赋值是可以通过 Swift 的类型检查的。

你可以利用名字 mathFunction 来调用指定的函数。

```
1 print("Result: \mathFunction(2,3)")  
2 // prints "Result: 5"
```

不同的函数如果有相同的匹配的类型的话，就可以指定相同的变量，和非函数的类型一样：

```
1 mathFunction=multiplyTwoInts  
2 print("Result: \\"(mathFunction(2,3))\"")  
3 // prints "Result: 6"
```

和其他的类型一样，当你指定一个函数为常量或者变量的时候，可以将它留给 Swift 来对类型进行推断：

```
1 let anotherMathFunction=addTwoInts  
2 // anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

函数类型作为形式参数类型

你可以利用使用一个函数的类型例如 `(Int,Int)->Int` 作为其他函数的形式参数类型。这允许你预留函数的部分实现从而让函数的调用者在调用函数的时候提供。

下面的栗子打印出了上文中数学函数执行后的结果：

```
1 funcprintMathResult(_mathFunction:(Int,Int)->Int,_a:Int,_b:Int){  
2     print("Result: \(mathFunction(a,b))")  
3 }  
4 printMathResult(addTwoInts,3,5)  
5 // Prints "Result: 8"
```

这个栗子定义了一个叫做 `printMathResult(_:_:_:)` 的函数，它有三个形式参数。第一个形式参数叫做 `mathFunction`，类型是 `(Int,Int)->Int`。你可以传入任何这个类型的函数作为第一个形式参数的实例参数。第二个和第三个形式参数叫做 `a` 和 `b`，它们都是 `Int` 类型。它们被用作提供的数学函数的两个传入值。

函数类型作为返回类型

你可以利用函数的类型作为另一个函数的返回类型。写法是在函数的返回箭头 (`->`) 后立即写一个完整的函数类型。

下边的栗子定义了两个简单函数叫做 `stepForward(_)` 和 `stepBackward(_)`。函数 `stepForward(_)` 返回一个大于输入值的值，而 `stepBackward(_)` 返回一个小于输入值的值。这两个函数的类型都是 `(Int)->Int`：

```
1 funcstepForward(_input:Int)->Int{  
2     return input+1  
3 }  
4 funcstepBackward(_input:Int)->Int{  
5     return input-1  
6 }
```

这里有一个函数 `chooseStepFunction(backward:)`，它的返回类型是“一个函数的类型 (`Int`)-`>Int`”。函数 `chooseStepFunction(backward:)` 返回了 `stepForward(_:)` 函数或者一个基于叫做 `backwards` 的布尔量形式参数的函数 `stepBackward(_:)`：

```
1 funcchooseStepFunction(backwards:Bool)->(Int)->Int{  
2     return backwards?stepBackward:stepForward  
3 }
```

现在你可以用 `chooseStepFunction(backward:)` 来得到一个向某方向前进或者其他的函数：

```
1 varcurrentValue=3  
2 letmoveNearerToZero=chooseStepFunction(backward:currentValue>0)  
3 // moveNearerToZero now refers to the stepBackward() function
```

上面的栗子显示了使变量趋近于零需要一个整数还是一个负数。`currentValue` 有一个 3 的初始值，也就是说 `currentValue>0` 返回 `true`，导致 `chooseStepFunction(backward:)` 返回 `stepBackward(_:)` 函数。一个返回函数的引用存储在名为 `moveNearerToZero` 的常量里。

现在这个 `moveNearerToZero` 指向了正确的函数，它可以用来进行到零的计算了：

```
1 print("Counting to zero:")  
2 // Counting to zero:  
3 whilecurrentValue!=0{  
4     print("\(currentValue)... ")  
5     currentValue=moveNearerToZero(currentValue)  
6 }  
7 print("zero!")  
8 // 3...  
9 // 2...  
10 // 1...  
11 // zero!
```

内嵌函数

到目前为止，你在本章中遇到的所有函数都是全局函数，都是在全局的范围内进行定义的。你也可以在函数的内部定义另外一个函数。这就是内嵌函数。

内嵌函数在默认情况下在外部是被隐藏起来的，但却仍然可以通过包裹它们的函数来调用它们。包裹的函数也可以返回它内部的一个内嵌函数来在另外的范围里使用。

你可以重写上边的栗子 `chooseStepFunction(backward:)` 来使用和返回内嵌函数：

```
1 funcchooseStepFunction(backward:Bool)->(Int)->Int{
2 funcstepForward(input:Int)->Int{returninput+1}
3 funcstepBackward(input:Int)->Int{returninput-1}
4 returnbackward?stepBackward:stepForward
5 }
6 varcurrentValue=-4
7 letmoveNearerToZero=chooseStepFunction(backward:currentValue>0)
8 // moveNearerToZero now refers to the nested stepForward() function
9 whilecurrentValue!=0{
10 print("\\"(currentValue)... ")
11 currentValue=moveNearerToZero(currentValue)
12 }
13 print("zero!")
14 // -4...
15 // -3...
16 // -2...
17 // -1...
18 // zero!
```

闭包

 cnswift.org/closures

闭包是可以在你的代码中被传递和引用的功能性独立模块。Swift 中的闭包和 C 以及 Objective-C 中的 blocks 很像，还有其他语言中的匿名函数也类似。

闭包能够捕获和存储定义在其上下文中的任何常量和变量的引用，这也就是所谓的`闭合`并包裹那些常量和变量，因此被称为“闭包”，Swift 能够为你处理所有关于捕获的内存管理的操作。

注意

不必担心你不熟悉“捕获”这个概念，在后面的[捕获值](#)中会对其进行详细介绍。

在[函数](#)章节中有介绍的全局和内嵌函数，实际上是特殊的闭包。闭包符合如下三种形式中的一种：

- 全局函数是一个有名字但不会捕获任何值的闭包；
- 内嵌函数是一个有名字且能从其上层函数捕获值的闭包；
- 闭包表达式是一个轻量级语法所写的可以捕获其上下文中常量或变量值的没有名字的闭包。

Swift 的闭包表达式拥有简洁的风格，鼓励在常见场景中实现简洁，无累赘的语法。常见的优化包括：

- 利用上下文推断形式参数和返回值的类型；
- 单表达式的闭包可以隐式返回；
- 简写实际参数名；
- 尾随闭包语法。

闭包表达式

内嵌函数，在[内嵌函数](#)中有介绍，一种在较复杂的函数中方便命名和定义独立代码块的手段。总之，有时候对于写更简短的没有完整定义和命名的类函数构造非常有用，尤其是在你处理一些函数时调用其他函数作为该函数的参数时。

闭包表达式是一种在简短行内就能写完闭包的语法。闭包表达式为了缩减书写长度又不失易读明晰而提供了一系列的语法优化。下边的闭包表达式栗子通过使用几次迭代展示 `sorted(by:)` 方法的精简来展示这些优化，每一次都让相同的功能性更加简明扼要。

Sorted 方法

Swift 的标准库提供了一个叫做 `sorted(by:)` 的方法，会根据你提供的排序闭包将已知类型的数组的值进行排序。一旦它排序完成，`sorted(by:)` 方法会返回与原数组类型大小完全相同的一个新数组，该数组的元素是已排序好的。原始数组不会被 `sorted(by:)` 方法修改。

下面这个闭包表达式的栗子使用 `sorted(by:)` 方法按字母排序顺序来排序一个 `String` 类型的数组。这是将被排序的初始数组：

```
1 letnames=["Chris","Alex","Ewa","Barry","Daniella"]
```

sorted(by:) 方法接收一个接收两个与数组内容相同类型的实际参数的闭包，然后返回一个 Bool 值来说明第一个值在排序后应该出现在第二个值的前边还是后边。如果第一个值应该出现在第二个值之前，排序闭包需要返回 true ，否则返回 false

这个栗子对一个 String 类型的数组进行排序，因此排序闭包需为一个 (String,String)->Bool 的类型函数。

提供排序闭包的一个方法是写一个符合其类型需求的普通函数，并将它作为 sorted(by:) 方法的形式参数传入：

```
1 funcbackward(_s1:String,_s2:String)->Bool{  
2     returns1>s2  
3 }  
4 varreversedNames=names.sorted(by:backward)  
5 //reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串 s1 大于第二个字符串 s2， backwards(_:_:) 函数将返回 true，这意味着 s1 应该在排序数组中排在 s2 的前面。对于在字符串中的字符，“比谁大”意思是“比较谁排在字母顺序的后面”。这意味着字母 “B” 是“大于”字母 “A”的，并且字符串 “Tom” 大于字符串 “Tim”。如果按照相反的字母顺序表的话，“Barry” 应该处于 “Alex”的前面，依次类推。

总之，这样来写本质上只是一个单一表达函数(a>b)是非常啰嗦的。在这个栗子中，我们更倾向于使用闭包表达式在行内写一个简单的闭包。

闭包表达式语法

闭包表达式语法有如下的一般形式：

```
1 {(parameters)->(returntype)in  
2 statements  
3 }
```

闭包表达式语法能够使用常量形式参数、变量形式参数和输入输出形式参数，但不能提供默认值。可变形式参数也能使用，但需要在形式参数列表的最后面使用。元组也可被用来作为形式参数和返回类型。

下面这个栗子展示一个之前 backward(_:_:) 函数的闭包表达版本：

```
1 reversedNames=names.sorted(by:{(s1:String,s2:String)->Bool in  
2     returns1>s2  
3 })
```

需要注意的是行内闭包的形式参数类型和返回类型的声明与 backwards(_:_:) 函数的申明相同。在这两个方式中，都书写成 (s1:String,s2:String)->Bool。总之对于行内闭包表达式来说，形式参数类型和返回类型都应写在花括号内而不是花括号外面。

闭包的函数整体部分由关键字 in 导入，这个关键字表示闭包的形式参数类型和返回类型定义已经完成，并且闭包的函数体即将开始。

闭包的函数体特别短以至于能够只用一行来书写：

```
1 reversedNames=names.sorted(by:{(s1:String,s2:String)->Boolinreturns1>s2})
```

示例中 `sorted(by:)` 方法的整体部分调用保持不变，一对圆括号仍然包裹函数的所有实际参数。然而，这些实际参数中的一个变成了现在的行内闭包。

从语境中推断类型

因排序闭包为实际参数来传递给函数，故 Swift 能推断它的形式参数类型和返回类型。

`sorted(by:)` 方法期望它的第二个形式参数是一个 `(String,String)->Bool`类型的函数。这意味着 `(String,String)` 和 `Bool` 类型不需要被写成闭包表达式定义中的一部分，因为所有的类型都能被推断，返回箭头 (`->`) 和围绕在形式参数名周围的括号也能被省略：

```
1 reversedNames=names.sorted(by:{s1,s2 inreturns1>s2})
```

当把闭包作为行内闭包表达式传递给函数，形式参数类型和返回类型都可以被推断出来。所以说，当闭包被用作函数的实际参数时你都不需要用完整格式来书写行内闭包。

然而，如果你希望的话仍然可以明确类型，并且在读者阅读你的代码时如果它能避免可能存在的歧义的话还是值得的。在这个 `sorted(by:)` 方法的栗子中，闭包的目的很明确，即排序被替换。对读者来说可以放心的假设闭包可能会使用 `String` 值，因为它正帮一个字符串数组进行排序。

从单表达式闭包隐式返回

单表达式闭包能够通过从它们的声明中删掉 `return` 关键字来隐式返回它们单个表达式的结

果，前面的栗子可以写作：

```
1 reversedNames=names.sorted(by:{s1,s2 ins1>s2})
```

这里，`sorted(by:)` 方法的第二个实际参数的函数类型已经明确必须通过闭包返回一个 `Bool` 值。因为闭包的结构包涵返回 `Bool` 值的单一表达式 (`s1>s2`)，因此没有歧义，并且 `return` 关键字能够被省略。

简写实际参数名

Swift 自动对行内闭包提供简写实际参数名，你也可以通过 `$0`, `$1`, `$2` 等名字来引用闭包的实际参数值。

如果你在闭包表达式中使用这些简写实际参数名，那么你可以在闭包的实际参数列表中忽略对其的定义，并且简写实际参数名的数字和类型将会从期望的函数类型中推断出来。`in` 关键字也能被省略，因为闭包表达式完全由它的函数体组成：

```
1 reversedNames=names.sorted(by:{$0>$1})
```

这里，`$0` 和 `$1` 分别是闭包的第一个和第二个 `String` 实际参数。

运算符函数

实际上还有一种更简短的方式来撰写上述闭包表达式。Swift 的 String 类型定义了关于大于号 (>) 的特定字符串实现，让其作为一个有两个 String 类型形式参数的函数并返回一个 Bool 类型的值。这正好与 sorted(by:) 方法的第二个形式参数需要的函数相匹配。因此，你能简单地传递一个大于号，并且 Swift 将推断你想使用大于号特殊字符串函数实现：

```
1 reversedNames=names.sorted(by:>)
```

想要了解更多有关运算符函数，请阅览运算符函数

尾随闭包

如果你需要将一个很长的闭包表达式作为函数最后一个实际参数传递给函数，使用尾随闭包将增强函数的可读性。尾随闭包是一个被书写在函数形式参数的括号外面（后面）的闭包表达式：

```
1 func someFunctionThatTakesAClosure(closure:()>Void){  
2 //function body goes here  
3 }  
4 //here's how you call this function without using a trailing closure  
5 someFunctionThatTakesAClosure({  
6 //closure's body goes here  
7 })  
8 //here's how you call this function with a trailing closure instead  
9 someFunctionThatTakesAClosure(){  
10 // trailing closure's body goes here  
11 }  
12  
13  
14  
15
```

注意

如果闭包表达式被用作函数唯一的实际参数并且你把闭包表达式用作尾随闭包，那么调用这个函数的时候你就不需要在函数的名字后面写一对圆括号 ()。

来自于上文的 闭包表达式一节的字符串排列闭包也可以作为一个尾随闭包被书写在 sorted(by:) 方法的括号外面：

```
1 reversedNames=names.sorted(){$0>$1}
```

如果闭包表达式作为函数的唯一实际参数传入，而你又使用了尾随闭包的语法，那你就不需要在函数名后边写圆括号了：

```
1 reversedNames=names.sorted{$0>$1}
```

当闭包很长以至于不能被写成一行时尾随闭包就显得非常有用了。举个栗子，Swift 的 Array 类型中有一个以闭包表达式为唯一的实际参数的 map(_:) 方法。数组中每一个元素调用一次该闭包，并且返回该元素所映射的值（有可能是其他类型）。具体的映射方式和返回值的类型有闭包来指定。

在给数组中每一个成员提供闭包时，map(_:) 方法返回一个新的数组，数组中包含与原数组一一对应的映射值。

使用一个带有尾随闭包的 `map(_)` 方法将一个包含 Int 值的数组转换成一个包含 String 值的数组。这个数组 [16,58,510] 被转换成一个新的数组

`["OneSix","FiveEight","FiveOneZero"]` :

```
1 letdigitNames=[  
2 0:"Zero",1:"One",2:"Two",3:"Three",4:"Four",  
3 5:"Five",6:"Six",7:"Seven",8:"Eight",9:"Nine"  
4 ]  
5 letnumbers=[16,58,510]  
6
```

上面的代码创建了一个整数数字到它们英文名字之间的映射字典，同时定义了一个将被转换成字符串的整数数组。

你现在可以通过传递一个闭包表达式到数组的 `map(_)` 方法中作为尾随闭包，利用 `number` 数组来创建一个 `String` 类型的数组。值得注意的是 `number.map` 不需要在 `map` 后面加任何圆括号，因为 `map(_)` 方法仅仅只有一个形式参数，这个形式参数将以尾随闭包的形式来书写：

```
1 letstrings=numbers.map{  
2 (number)->Stringin  
3 varnumber=number  
4 varoutput=""  
5 repeat{  
6 output=digitNames[number%10]!+output  
7 number/=10  
8 }whilenumber>0  
9 returnoutput  
10 }  
11 // strings is inferred to be of type [String]  
12 // its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_)` 方法在数组中为每一个元素调用了一次闭包表达式。你不需要指定闭包的输入形式参数 `number` 的类型，因为它能从数组中将被映射的值来做推断。

这个栗子中，变量 `number` 以闭包的 `number` 形式参数初始化，这样它就可以在闭包结构内部直接被修改。（函数和闭包的形式参数是常量。）闭包表达式同样指定返回 `String` 类型，来表明存储映射值的 `output` 数组也为 `String` 类型。

闭包表达式建立一个命名为 `output` 的字符串，每调用一次就命名一次。它通过利用余数运算符 (`number%10`) 计算了 `number` 的最后一位数字，并且用这个数字在 `digitNames` 字典中找到相对应的字符串。这个闭包能够被用来创建一个任何一个大于零的整数表示的字符串。

注意

`digitNames` 的下标紧跟着一个感叹号(!)，因为字典下标返回一个可选值，表明即使该 key 不存在也不会查找失败。上述这个栗子中，它保证了 `number%10` 可以总是作为字典 `digitNames` 的下标 key，因此一个感叹号可以被用作强制展开 (force-unwrap) 存储在可选返回值下标项的 `String` 值。

从字典 `digitNames` 获取的字符串被添加到 `output` 的前面，有效的逆序建立了一个字符串版本的数字。（表达式 `number%10` 得出 16 中的 6，58 中的 8，510 中的 ）

`number` 变量随后被 10 分开，因为他是一个整数，未除尽部分被忽略。因此 16 便得 1，58 得 5，510 得 51。

整个过程反复运行，直到 `number/=10` 是等于 0，这时闭包会将字符串 `output` 输出，并且通过 `map(_)` 方法添加到输出数组中。

在上面这个例子中尾随闭包语法在函数后整洁地封装了具体的闭包功能，而不再需要将整个闭包包裹在 `map(_)` 方法的括号内。

捕获值

一个闭包能够从上下文捕获已被定义的常量和变量。即使定义这些常量和变量的原作用域已经不存在，闭包仍能够在其函数体内引用和修改这些值。

在 Swift 中，一个能够捕获值的闭包最简单的模型是内嵌函数，即被书写在另一个函数的内部。一个内嵌函数能够捕获外部函数的实际参数并且能够捕获任何在外部函数的内部定义了的常量与变量。

这里有个命名为 `makeIncrement` 的函数栗子，其中包含了一个名叫 `incrementer` 一个内嵌函数。这个内嵌 `incrementer()` 函数能在它的上下文捕获两个值，`runningTotal` 和 `amount`。在捕获这些值后，通过 `makeIncrement` 将 `incrementer` 作为一个闭包返回，每一次调用 `incrementer` 时，将以 `amount` 作为增量来增加 `runningTotal`：

```
1 func makeIncrementer(forIncrement amount:Int)->()->Int{
2     var runningTotal=0
3     func incrementer()->Int{
4         runningTotal+=amount
5         return runningTotal
6     }
7     return incrementer
8 }
```

`makeIncrementer` 的返回类型是 `()->Int`，意思就是比起返回一个单一的值，它返回的是一个函数。这个函数没有返回任何形式参数，每调用一次就返回一个 `Int` 值。想要了解更多关于函数是如何返回另一个函数的，请参照[函数类型作为返回类型](#)。

`makeIncrementer(forIncrement:)` 函数定义了一个叫 `runningTotal` 的整数变量，用来存储当前增加的总量，该值通过 `Incrementer` 返回。这个变量用 0 初始化。

`makeIncrementer(forIncrement:)` 函数只有一个在外部命名为 `Incrementer`，局部命名为 `amount` 的 `Int` 形式参数。在每次调用 `incrementer` 函数时，实际参数值通过形式参数指定 `runningTotal` 增加多少。

`makeIncrementer` 定义了一个名叫 `incrementer` 的内嵌函数，表明实际增加量，这个函数直接把 `amount` 增加到 `runningTotal`，并且返回结果。

当我们单看这个函数时，会发现内嵌函数 `incrementer()` 不同寻常：

```
1 func incrementer()->Int{
2     runningTotal+=amount
3     return runningTotal
4 }
```

`incrementer()` 函数是没有任何形式参数，`runningTotal` 和 `amount` 不是来自于函数体的内部，而是通过捕获主函数的 `runningTotal` 和 `amount` 把它们内嵌在自身函数内部供使用。当调用 `makeIncrementer` 结束时通过引用捕获来确保不会消失，并确保了在下次再次调用

incrementer 时，runningTotal 将继续增加。

注意

作为一种优化，如果一个值没有改变或者在闭包的外面，Swift 可能会使用这个值的拷贝而不是捕获。

Swift 也处理了变量的内存管理操作，当变量不再需要时会被释放。

这有个使用 makeIncrementer 的栗子：

```
1 let incrementByTen = makeIncrementer(forIncrement: 10)
```

这个例子定义了一个叫 incrementByTen 的常量，该常量指向一个每次调用会加 10 的函数。调用这个函数多次得到以下结果：

```
1 incrementByTen()
2 //return a value of 10
3 incrementByTen()
4 //return a value of 20
5 incrementByTen()
6 //return a value of 30
```

如果你建立了第二个 incrementer，它将会有一个新的、独立的 runningTotal 变量的引用：

```
1 let incrementBySeven = makeIncrementer(forIncrement: 7)
2 incrementBySeven()
3 // returns a value of 7
```

再次调用原来增量器 (incrementByTen) 继续增加它自己的变量 runningTotal 的值，并且不影响 incrementBySeven 捕获的变量 runningTotal 值：

```
1 incrementByTen()
2 // returns a value of 40
```

注意

如果你分配了一个闭包给类实例的属性，并且闭包通过引用该实例或者它的成员来捕获实例，你将在闭包和实例间建立一个强引用环。

Swift 将使用捕获列表来打破这种强引用环。更多信息请参考[闭包的强引用环](#)。

闭包是引用类型

在上面例子中，incrementBySeven 和 incrementByTen 是常量，但是这些常量指向的闭包仍可以增加已捕获的变量 runningTotal 的值。这是因为函数和闭包都是引用类型。

无论你什么时候安赋值一个函数或者闭包给常量或者变量，你实际上都是将常量和变量设置为对函数和闭包的引用。这上面这个例子中，闭包选择 incrementByTen 指向一个常量，而不是闭包它自身的内容。

这也意味着你赋值一个闭包到两个不同的常量或变量中，这两个常量或变量都将指向相同的闭包：

```
1 let alsoIncrementByTen=incrementByTen
2 alsoIncrementByTen()
3 //return a value of 50
```

逃逸闭包

当闭包作为一个实际参数传递给一个函数的时候，我们就说这个闭包逃逸了，因为它可以在函数返回之后被调用。当你声明一个接受闭包作为形式参数的函数时，你可以在形式参数前写 `@escaping` 来明确闭包是允许逃逸的。

闭包可以逃逸的一种方法是被储存在定义于函数外的变量里。比如说，很多函数接收闭包实际参数来作为启动异步任务的回调。函数在启动任务后返回，但是闭包要直到任务完成——闭包需要逃逸，以便于稍后调用。举例来说：

```
1 var completionHandlers:[()>Void]=[]
2 func someFunctionWithEscapingClosure(completionHandler:@escaping()>Void){
3     completionHandlers.append(completionHandler)
4 }
```

函数 `someFunctionWithEscapingClosure(_)` 接收一个闭包作为实际参数并且添加它到声明在函数外部的数组里。如果你不标记函数的形式参数为 `@escaping`，你就会遇到编译时错误。

让闭包 `@escaping` 意味着你必须在闭包中显式地引用 `self`，比如说，下面的代码中，传给 `someFunctionWithEscapingClosure(_)` 的闭包是一个逃逸闭包，也就是说它需要显式地引用 `self`。相反，传给 `someFunctionWithNonescapingClosure(_)` 的闭包是非逃逸闭包，也就是说它可以隐式地引用 `self`。

```
1 func someFunctionWithNonescapingClosure(closure:()>Void){
2     closure()
3 }
4 class SomeClass{
5     var x=10
6     func doSomething(){
7         someFunctionWithEscapingClosure{self.x=100}
8         someFunctionWithNonescapingClosure{x=200}
9     }
10 }
11 let instance=SomeClass()
12 instance.doSomething()
13 print(instance.x)
14 // Prints "200"
15 completionHandlers.first?()
16 print(instance.x)
17 // Prints "100"
18
19
20
```

自动闭包

自动闭包是一种自动创建的用来把作为实际参数传递给函数的表达式打包的闭包。它不接受任何实际参数，并且当它被调用时，它会返回内部打包的表达式的值。这个语法的好处在于通过写普通表达式代替显式闭包而使你省略包围函数形式参数的括号。

调用一个带有自动闭包的函数是很常见的，但实现这类函数就不那么常见了。比如说，`assert(condition:message:file:line:)` 函数为它的 `condition` 和 `message` 形式参数接收一个自动闭包；它的 `condition` 形式参数只有在调试构建时才评判，而且 `message` 形式参数只有在 `condition` 是 `false` 时才评判。

自动闭包允许你延迟处理，因此闭包内部的代码直到你调用它的时候才会运行。对于有副作用或者占用资源的代码来说很有用，因为它可以允许你控制代码何时才进行求值。下面的代码展示了闭包如何延迟求值。

```
1 var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
2 print(customersInLine.count)
3 // Prints "5"
4 let customerProvider = {customersInLine.remove(at:0)}
5 print(customersInLine.count)
6 // Prints "5"
7 print("Now serving \(customerProvider())!")
8 // Prints "Now serving Chris!"
9 print(customersInLine.count)
10 // Prints "4"
11
12
```

尽管 `customersInLine` 数组的第一个元素以闭包的一部分被移除了，但任务并没有执行直到闭包被实际调用。如果闭包永远不被调用，那么闭包里边的表达式就永远不会求值。注意 `customerProvider` 的类型不是 `String` 而是 `()->String` ——一个不接受实际参数并且返回一个字符串的函数。

当你传一个闭包作为实际参数到函数的时候，你会得到与延迟处理相同的行为。

```
1 // customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
2 func serve(customer customerProvider: ()->String) {
3     print("Now serving \(customerProvider())!")
4 }
5 serve(customer: {customersInLine.remove(at:0)})
6 // Prints "Now serving Alex!"
```

上边的函数 `serve(customer:)` 接收一个明确的返回下一个客户名称的闭包。下边的另一个版本的 `serve(customer:)` 执行相同任务但是不使用明确的闭包而是通过 `@autoclosure` 标志标记它的形式参数使用了自动闭包。现在你可以调用函数就像它接收了一个 `String` 实际参数而不是闭包。实际参数自动地转换为闭包，因为 `customerProvider` 形式参数的类型被标记为 `@autoclosure` 标记。

```
1 // customersInLine is ["Ewa", "Barry", "Daniella"]
2 func serve(customer customerProvider: @autoclosure()->String) {
3     print("Now serving \(customerProvider())!")
4 }
5 serve(customer: customersInLine.remove(at:0))
6 // Prints "Now serving Ewa!"
```

注意

滥用自动闭包会导致你的代码难以读懂。上下文和函数名应该写清楚求值是延迟的。

如果你想要自动闭包允许逃逸，就同时使用 `@autoclosure` 和 `@escaping` 标志。`@escaping` 标志在上边的逃逸闭包里有详细的解释。

```
1 // customersInLine is ["Barry", "Daniella"]
2 varcustomerProviders:[()>String]=[]
3 funccollectCustomerProviders(_customerProvider:@autoclosure@escaping()->String){
4     customerProviders.append(customerProvider)
5 }
6 collectCustomerProviders(customersInLine.remove(at:0))
7 collectCustomerProviders(customersInLine.remove(at:0))
8 print("Collected \$(customerProviders.count) closures.")
9 // Prints "Collected 2 closures."
10 forcustomerProviderincustomerProviders{
11     print("Now serving \$(customerProvider())!")
12 }
13 // Prints "Now serving Barry!"
14 // Prints "Now serving Daniella!"
15
```

上边的代码中，不是调用传入后作为 customerProvider 实际参数的闭包，
collectCustomerProviders(_:) 函数把闭包追加到了 customerProviders 数组的末尾。数组声明在函数的生效范围之外，也就是说数组里的闭包有可能在函数返回之后执行。结果，customerProvider 实际参数的值必须能够逃逸出函数的生效范围。

枚举

 cnswift.org/enumerations

枚举为一组相关值定义了一个通用类型，从而可以让你在代码中类型安全地操作这些值。

如果你熟悉 C，那么你可能知道 C 中的枚举会给一组整数值分配相关的名称。Swift 中的枚举则更加灵活，并且不需给枚举中的每一个成员都提供值。如果一个值（所谓“原始”值）要被提供给每一个枚举成员，那么这个值可以是字符串、字符、任意的整数值，或者是浮点类型。

而且，枚举成员可以指定任意类型的值来与不同的成员值关联储存，这更像是其他语言中的 union 或 variant 的效果。你可以定义一组相关成员的合集作为枚举的一部分，每一个成员都可以有不同类型的值的合集与其关联。

Swift 中的枚举是具有自己权限的一类类型。它们使用了许多一般只被类所支持的特性，例如计算属性用来提供关于枚举当前值的额外信息，并且实例方法用来提供与枚举表示的值相关的能力。枚举同样也能够定义初始化器来初始化成员值；而且能够遵循协议来提供标准功能。

要了解更多，请参阅属性, 方法, 初始化, 扩展, 和协议。

枚举语法

你可以用 enum关键字来定义一个枚举，然后将其所有的定义内容放在一个大括号（{}）中：

```
1 enumSomeEnumeration{  
2 // enumeration definition goes here  
3 }
```

这是一个指南针的四个主要方向的例子：

```
1 enumCompassPoint{  
2 casenorth  
3 casesouth  
4 caseeast  
5 casewest  
6 }
```

在一个枚举中定义的值（比如： north , south , east 和 west）就是枚举的成员值（或成员） case关键字则明确了要定义成员值。

注意

不像 C 和 Objective-C 那样，Swift 的枚举成员在被创建时不会分配一个默认的整数值。在上文的 CompassPoint例子中， north , south , east和 west并不代表 0 , 1 , 2和 3。而相反，不同的枚举成员在它们自己的权限中都是完全合格的值，并且是一个在 CompassPoint中被显式定义的类型。

多个成员值可以出现在同一行中，要用逗号隔开：

```
1 enumPlanet{  
2     case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune  
3 }
```

每个枚举都定义了一个全新的类型。正如 Swift 中其它的类型那样，它们的名称（例如：CompassPoint和 Planet）需要首字母大写。给枚举类型起一个单数的而不是复数的名字，从而使得它们能够顾名思义：

```
1 var directionToHead = CompassPoint.west
```

当与 `CompassPoint` 中可用的某一值一同初始化时 `directionToHead` 的类型会被推断出来。一旦 `directionToHead` 以 `CompassPoint` 类型被声明，你就可以用一个点语法把它设定成不同的 `CompassPoint` 值：

```
1 directionToHead = .east
```

`directionToHead` 的类型是已知的，所以当设定它的值时你可以不用写类型。这样做可以使得你在操作确定类型的枚举时让代码非常易读。

使用 Switch 语句来匹配枚举值

你可以用 `switch` 语句来匹配每一个单独的枚举值：

```
1 directionToHead = .south  
2 switch directionToHead {  
3     case .north:  
4         print("Lots of planets have a north")  
5     case .south:  
6         print("Watch out for penguins")  
7     case .east:  
8         print("Where the sun rises")  
9     case .west:  
10        print("Where the skies are blue")  
11    }  
12 // prints "Watch out for penguins"
```

你可以将上述代码读作：

“判断 `directionToHead` 的值。在等于 `.north` 的 `case` 中，则打印 `"Lots of planets have a north"`。在等于 `.south` 的 `case` 中，则显示 `"Watch out for penguins"`”

.....以此类推。

就像在控制流中所描述的那样，当判断一个枚举成员时，`switch` 语句应该是全覆盖的。如果 `.west` 的 `case` 被省略了，那么代码将不能编译，因为这时表明它并没有覆盖 `CompassPoint` 的所有成员。要求覆盖所有枚举成员是因为这样可以保证枚举成员不会意外的被漏掉。

如果不能为所有枚举成员都提供一个 `case`，那你也可以提供一个 `default` 情况来包含那些不能被明确写出的成员：

```
1 let somePlanet=Planet.earth
2 switch somePlanet{
3 case.earth:
4 print("Mostly harmless")
5 default:
6 print("Not a safe place for humans" )
7 }
8 // Prints "Mostly harmless"
```

关联值

之前几节中的栗子展示了枚举成员是怎样在他们各自的权限中被定义（和被分类）的。你可以给 `Planet.earth` 设定常量或变量，然后再使用这个值。总之，有时将其它类型的关联值与这些成员值一起存储是很有用的。这样你就可以将额外的自定义信息和成员值一起储存，并且允许你在代码中使用每次调用这个成员时都能使用它。

你可以定义 Swift 枚举来存储任意给定类型的关联值，如果需要的话不同枚举成员关联值的类型可以不同。枚举其他语言中的 *discriminated unions*, *tagged unions*, 或者 *variants* 类似。

举个栗子，假设库存跟踪系统需要按两个不同类型的条形码跟踪产品，一些产品贴的是用数字 0~9 的 UPC-A 格式一维条形码。每一个条码数字都含有一个“数字系统”位，之后是五个“制造商代码”数字和五个“产品代码”数字。而最后则是一个“检测”位来验证代码已经被正确扫描：



其它的产品则贴着二维码，它可以使用任何 ISO 8859-1 字符并且编码最长有 2953 个字符的字符串：

这样可以让库存跟踪系统很方便的以一个由 4 个整数组成的元组来储存 UPC-A 条形码，然而二维码则可以被存储为一个任意长度的字符串中。

在 Swift 中，为不同类型产品条码定义枚举大概是这种姿势：

```
1 enum Barcode{
2     case upc(Int, Int, Int, Int)
3     case qrCode(String)
4 }
```

这可以读作：

“定义一个叫做 `Barcode` 的枚举类型，它要么用 `(Int, Int, Int, Int)` 类型的关联值获取 `upc` 值，要么用 `String` 类型的关联值获取一个 `qrCode` 的值。”



这个定义并不提供任何实际的 `Int` 或者 `String` 的值——它只定义当 `Barcode` 常量和变量与 `Barcode.upc` 或 `Barcode.qrCode` 相同时可以存储的关联值的类型。

然后，新的条码就可以用任意一个类型来创建了：

```
1 var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

这个栗子创建了一个叫做 `productBarcode` 的新变量而且给它赋值了一个 `Barcode.upc` 的值关联了值为 `(8, 85909, 51226, 3)` 的元组值。

同样的产品可以被分配一个不同类型的条码：

```
1 productBarcode = .qrCode("ABCDEFGHIJKLMNP")
```

这时，最初的 `Barcode.upc` 和它的整数值将被新的 `Barcode.qrCode` 和它的字符串值代替。 `Barcode` 类型的常量和变量可以存储一个 `.upc` 或一个 `.qrCode` (和它们的相关值一起存储) 中的任意一个，但是它们只可以在给定的时间内存储它们其中之一。

和以往一样，不同的条码类型可以用 `switch` 语句来检查。这一次，总之，相关值可以被提取为 `switch` 语句的一部分。你提取的每一个相关值都可以作为常量 (用 `let` 前缀) 或者变量 (用 `var` 前缀) 在 `switch` 的 `case` 中使用：

```
1 switch productBarcode{
2     case .upc(let numberSystem, let manufacturer, let product, let check):
3         print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
4     case .qrCode(let productCode):
5         print("QR code: \(productCode).")
6     }
7 // Prints "QR code: ABCDEFGHIJKLMNOP."
```

如果对于一个枚举成员的所有相关值都被提取为常量，或如果都被提取为变量，为了简洁，你可以用一个单独的 `var` 或 `let` 在成员名称前标注：

```
1 switch productBarcode{  
2 case let upc(numberSystem, manufacturer, product, check):  
3 print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")  
4 case let qrCode(productCode):  
5 print("QR code: \(productCode).")  
6 }  
7 // Prints "QR code: ABCDEFGHIJKLMNOP."
```

原始值

关联值中条形码的栗子展示了枚举成员是如何声明它们存储不同类型的相关值的。作为相关值的另一种选择，枚举成员可以用相同类型的默认值预先填充（称为原始值）。

这里有一个和已命名的枚举成员一起存储的原始 ASCII 码的例子：

```
1 enum ASCIIControlCharacter: Character{  
2 case tab = "\t"  
3 case lineFeed = "\n"  
4 case carriageReturn = "\r"  
5 }
```

这里，一个叫做 `ASCIIControlCharacter` 的枚举原始值被定义为类型 `Character`，并且被放置在了更多的一些 ASCII 控制字符中，`Character` 值的描述见[字符串和字符](#)。

注意

原始值与关联值不同。原始值是当你第一次定义枚举的时候，它们用来预先填充的值，正如上面的三个 ASCII 码。特定枚举成员的原始值是始终相同的。关联值在你基于枚举成员的其中之一创建新的常量或变量时设定，并且在你每次这么做的时候这些关联值可以是不同的。

隐式指定的原始值

当你在操作存储整数或字符串原始值枚举的时候，你不必显式地给每一个成员都分配一个原始值。当你没有分配时，Swift 将会自动为你分配值。

实际上，当整数值被用于作为原始值时，每成员的隐式值都比前一个大一。如果第一个成员没有值，那么它的值是 0。

下面的枚举是先前的 `Planet` 枚举的简化，用整数原始值来代表从太阳到每一个行星的顺序：

```
1 enum Planet: Int{  
2 case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune  
3 }
```

在上面的例子中，`Planet.mercury` 有一个明确的原始值 1，`Planet.venus` 的隐式原始值是 2，以此类推。

当字符串被用于原始值，那么每一个成员的隐式原始值则是那个成员的名称。

下面的枚举是先前 `CompassPoint` 枚举的简化，有字符串的原始值来代表每一个方位的名字：

```
1 enumCompassPoint: String{  
2   case north, south, east, west  
3 }
```

在上面的例子中， CompassPoint.south有一个隐式原始值 "south"，以此类推。

你可以用 rawValue属性来访问一个枚举成员的原始值：

```
1 letearthsOrder=Planet.Earth.rawValue  
2 // earthsOrder is 3  
3 letsunsetDirection=CompassPoint.west.rawValue  
4 // sunsetDirection is "west"  
5
```

从原始值初始化

如果你用原始值类型来定义一个枚举，那么枚举就会自动收到一个可以接受原始值类型的值的初始化器（叫做 rawValue的形式参数）然后返回一个枚举成员或者 nil。你可以使用这个初始化器来尝试创建一个枚举的新实例。

这个例子从它的原始值 7来辨认出 Uranus：

```
1 letpossiblePlanet=Planet(rawValue:7)  
2 // possiblePlanet is of type Planet? and equals Planet.Uranus
```

总之，不是所有可能的 Int 值都会对应一个行星。因此原始值的初始化器总是返回可选的枚举成员。在上面的例子中， possiblePlanet的类型是 Planet?，或者“可选项 Planet”

注意

原始值初始化器是一个可失败初始化器，因为不是所有原始值都将返回一个枚举成员。要获取更多信息，请参阅[可失败初始化器](#)。

如果你尝试寻找有位置 11的行星，那么被原始值初始化器返回的可选项 Planet值将会是 nil:

```
1 letpositionToFind=11  
2 ifletsomePlanet=Planet(rawValue:positionToFind){  
3   switchsomePlanet{  
4     case.earth:  
5       print("Mostly harmless")  
6     default:  
7       print("Not a safe place for humans")  
8     }  
9   }else{  
10   print("There isn't a planet at position \ \(positionToFind)")  
11 }  
12 // Prints "There isn't a planet at position 11"
```

这个例子使用可选项绑定来尝试访问一个原始值是 11 的行星。其中的 `ifletsomePlanet=Planet(rawValue:11)` 语句创建了一个可选项 Planet，而且如果 Planet 的值可被取回的话，就将它赋给 somePlanet。在这种情况下，取回一个位置为 11 的行星是不可能的，所以执行 else分支会被执行。

递归枚举

枚举在对序号考虑固定数量可能性的数据建模时表现良好，比如用来做简单整数运算的运算符。这些运算符允许你组合简单的整数数学运算表达式比如5到更复杂的比如5+4.

数学表达式的一大特征就是它们可以内嵌。比如说表达式 $(5 + 4) * 2$ 在乘法右手侧有一个数但其他表达式在乘法的左手侧。因为数据被内嵌了，用来储存数据的枚举同样需要支持内嵌——这意味着枚举需要被递归。

递归枚举是拥有另一个枚举作为枚举成员关联值的枚举。当编译器操作递归枚举时必须插入间接寻址层。你可以在声明枚举成员之前使用 `indirect` 关键字来明确它是递归的。

举例来讲，这里有一个储存简单数学运算表达式的枚举：

```
1 enum ArithmeticExpression{
2     casenumber(Int)
3     indirect caseaddition(ArithmeticExpression,ArithmeticExpression)
4     indirect casemultiplication(ArithmeticExpression,ArithmeticExpression)
5 }
```

你同样可以在枚举之前写 `indirect` 来让整个枚举成员在需要时可以递归：

```
1 indirect enum ArithmeticExpression{
2     casenumber(Int)
3     caseaddition(ArithmeticExpression,ArithmeticExpression)
4     casemultiplication(ArithmeticExpression,ArithmeticExpression)
5 }
```

这个枚举可以储存三种数学运算表达式：单一的数字，两个两个表达式的加法，以及两个表达式的乘法。`addition` 和 `multiplication` 成员拥有同样是数学表达式的关联值——这些关联值让嵌套表达式成为可能。比如说，表达式 $(5+4)*2$ 乘号右侧有一个数字左侧有其他表达式。由于数据是内嵌的，用来储存数据的枚举同样需要支持内嵌——这就是说枚举需要递归。下边的代码展示了为 $(5+4)*2$ 创建的递归枚举 `ArithmeticExpression`：

```
1 let five=ArithmeticExpression.number(5)
2 let four=ArithmeticExpression.number(4)
3 let sum=ArithmeticExpression.addition(five,four)
4 let product=ArithmeticExpression.multiplication(sum,ArithmeticExpression.number(2))
```

递归函数是一种操作递归结构数据的简单方法。比如说，这里有一个判断数学表达式的函数：

```
1 func evaluate(_expression:ArithmeticExpression)->Int{
2     switch expression{
3         case let .number(value):
4             return value
5         case let .addition(left,right):
6             return evaluate(left)+evaluate(right)
7         case let .multiplication(left,right):
8             return evaluate(left)*evaluate(right)
9     }
10 }
11 print(evaluate(product))
12 // Prints "18"
13
```

这个函数通过直接返回关联值来判断普通数字。它通过衡量表达式左手侧和右手侧判断是加法还是乘法，然后对它们加或者乘。

类和结构体

 cnswift.org/classes-and-structures

作为你程序代码的构建基础，类和结构体是一种多功能且灵活的构造体。通过使用与现存常量、变量、函数完全相同的语法来在类和结构体当中定义属性和方法以添加功能。

不像其他的程序语言，Swift不需要你为自定义类和结构体创建独立的接口和实现文件。在Swift中，你在一个文件中定义一个类或者结构体，则系统将会自动生成面向其他代码的外部接口。

注意

一个类的实例通常被称为对象。总之，Swift的类和结构体在功能上要比其他语言中的更加相近，并且本章节所讨论的大部分功能都可以同时用在类和结构体的实例上。因此，我们使用更加通用的术语实例。

类与结构体的对比

在Swift中类和结构体有很多共同之处，它们都能：

- 定义属性用来存储值；
- 定义方法用于提供功能；
- 定义下标脚本用来允许使用下标语法访问值；
- 定义初始化器用于初始化状态；
- 可以被扩展来默认所没有的功能；
- 遵循协议来针对特定类型提供标准功能。

更多信息，请阅览[属性](#)，[方法](#)，[下标脚本](#)，[初始化](#)，[扩展](#)和[协议](#)。

类有而结构体没有的额外功能：

- 继承允许一个类继承另一个类的特征；
- 类型转换允许你在运行检查和解释一个类实例的类型；
- 反初始化器允许一个类实例释放任何其所被分配的资源；
- 引用计数允许不止一个对类实例的引用。

更多信息，请阅览[继承](#)，[类型转换](#)，[反初始化](#)和[自动引用计数](#)。

注意

结构体在你的代码中通过复制来传递，并且并不会使用引用计数。

定义语法

类与结构体有着相似的定义语法，你可以通过使用关键词class来定义类使用struct来定义结构体。并在一对大括号内定义它们的具体内容。

```
1 classSomeClass{  
2 // class definition goes here  
3 }  
4 structSomeStructure{  
5 // structure definition goes here  
6 }
```

注意

无论你在何时定义了一个新的类或者结构体，实际上你定义了一个全新的 Swift 类型。请用 UpperCamelCase 命名法¹¹命名（比如这里我们说到的 SomeClass 和 SomeStructure）以符合 Swift 的字母大写风格（比如说 String，Int 以及 Bool）。相反，对于属性和方法使用 lowerCamelCase 命名法¹¹（比如 frameRate 和 incrementCount），以此来区别于类型名称。

这里有个类定义和结构体定义的例子：

```
1 structResolution{  
2 varwidth=0  
3 varheight=0  
4 }  
5 classVideoMode{  
6 varresolution=Resolution()  
7 varinterlaced=false  
8 varframeRate=0.0  
9 varname:String?  
10 }
```

上面这个例子定义了一个名叫 Resolution的新结构体，用来描述一个基于像素的显示器分辨率。这个结构体拥有两个存储属性名叫 width 和 height，存储属性是绑定并储存在类或者结构体中的常量或者变量。这两个属性因以值 0 来初始化，所以它们的类型被推断为 Int。

上面这个例子也定义了一个名叫 VideoMode的新类，用来描述一个视频显示的特定视频模式。这个类有四个变量存储属性。第一个， resolution, 用 Resolution结构体实例来初始化，它使属性的类型被推断为 Resolution。对于其他三个属性来说，新的 VideoMode实例将会以 interlaced 为 false（意思是“非隔行扫描视频”），回放帧率为 0.0，和一个名叫 name 的可选项 String 值来初始化。name 属性会自动被赋予一个空值 nil，或“无 name 值”，因为它是一个可选项。

类与结构体实例

Resolution结构体的定义和 VideoMode类的定义仅仅描述了什么是 Resolution 或 VideoMode。它们自己并没有描述一个特定的分辨率或视频 模式。对此，你需要创建一个结构体或类的实例。

创建结构体和类的实例的语法是非常相似的：

```
1 letsomeResolution=Resolution()  
2 letsomeVideoMode=VideoMode()
```

结构体和类两者都能使用初始化器语法来生成新的实例。初始化器语法最简单的是在类或结构体名字后面接一个空的圆括号，例如 Resolution() 或者 VideoMode()。这样就创建了一个新的类或者结构体的实例，任何属性都被初始化为它们的默认值。在 [初始化](#)一章有对类和结

构体的初始化更详尽的描述。

访问属性

你可以用点语法来访问一个实例的属性。在点语法中，你只需在实例名后面书写属性名，用(.)来分开，无需空格：

```
1 print("The width of someResolution is \(someResolution.width)")  
2 // prints "The width of someResolution is 0"
```

在这个栗子中，someResolution.width就是someResolution中的width属性，返回一个它的默认初始值0。

你可以继续下去来访问子属性，如VideoMode中resolution属性中的width属性：

```
1 print("The width of someVideoMode is \(someVideoMode.resolution.width)")  
2 // prints "The width of someVideoMode is 0"
```

你亦可以用点语法来指定一个新值到一个变量属性中：

```
1 someVideoMode.resolution.width=1280  
2 print("The width of someVideoMode is now \(someVideoMode.resolution.width)")  
3 // prints "The width of someVideoMode is now 1280"
```

注意

不同于Objective-C，Swift允许你直接设置一个结构体属性中的子属性。在上述最后一个栗子中，someVideoMode的resolution属性中的width这个属性可以直接设置，不用你重新设置整个resolution属性到一个新值。

结构体类型的成员初始化器

所有的结构体都有一个自动生成的成员初始化器，你可以使用它来初始化新结构体实例的成员属性。新实例属性的初始化值可以通过属性名称传递到成员初始化器中：

```
1 let vga=Resolution(width:640,height:480)
```

与结构体不同，类实例不会接收默认的成员初始化器，初始化器的更多细节在初始化章节。

结构体和枚举是值类型

值类型是一种当它被指定到常量或者变量，或者被传递给函数时会被拷贝的类型。

其实，在之前的章节中我们已经大量使用了值类型。实际上，Swift中所有的基本类型——整数，浮点数，布尔量，字符串，数组和字典——都是值类型，并且都以结构体的形式在后台实现。

Swift中所有的结构体和枚举都是值类型，这意味着你所创建的任何结构体和枚举实例——和实例作为属性所包含的任意值类型——在代码传递中总是被拷贝的。

看这个栗子，其使用了前面例子中的Resolution结构体：

```
1 let hd=Resolution(width:1920,height:1080)
2 var cinema=hd
```

这个栗子声明了一个叫 `hd` 的常量，并且赋予它一个以全高清视频(1920像素宽乘以 1080像素高)宽和高初始化的 `Resolution`实例。

之后声明了一个叫 `cinema`的变量并且以当前 `hd` 的值进行初始化。因为 `Resolution`是一个结构体，现有实例的拷贝会被制作出来，然后这份新的拷贝就赋值给了 `cinema`。尽管 `hd`和 `cinema`有相同的像素宽和像素高，但是在后台中他们是两个完全不同的实例。

接下来，为了适应数字影院的放映需求 (2048像素宽和 1080像素高)，我们把 `cinema`的属性 `width`修改为稍宽一点的 2K 标准：

```
1 cinema.width=2048
```

检查 `cinema`的 `width`属性发现已被改成 2048：

```
1 println("cinema is now \$(cinema.width) pixels wide")
2 //println "cinema is now 2048 pixels wide"
```

总之，原始 `hd`实例中的 `width`属性依旧是 1920：

```
1 print("hd is still \$(hd.width) pixels wide")
2 // prints "hd is still 1920 pixels wide"
```

当 `cinema`被赋予 `hd`的当前值，存储在 `hd`中的值就被拷贝给了新的 `cinema`实例。这最终的结果是两个完全不同的实例，它们只是碰巧包含了相同的数字值。由于它们是完全不同的实例， `cinema`的宽度被设置 2048并不影响 `hd`中 `width`存储的值。

这种行为规则同样适用于枚举：

```
1 enum CompassPoint{
2     case North,South,East,West
3 }
4 var currentDirection=CompassPoint.West
5 let rememberedDirection=currentDirection
6 currentDirection=.East
7 if rememberedDirection==.West{
8     print("The remembered direction is still .West")
9 }
10 // prints "The remembered direction is still .West"
```

当 `rememberedDirection`被赋予了 `currentDirection`中的值，实际上是值的拷贝。之后再改变 `currentDirection`的值并不影响 `rememberedDirection`所存储的原版值。

类是引用类型

不同于值类型，在引用类型被赋值到一个常量，变量或者本身被传递到一个函数的时候它是不会被拷贝的。相对于拷贝，这里使用的是同一个对现存实例的引用。

这里有个栗子，使用上面定义的 `VideoMode`类：

```
1 lettenEighty=VideoMode()
2 tenEighty.resolution=hd
3 tenEighty.interlaced=true
4 tenEighty.name="1080i"
5 tenEighty.frameRate=25.0
```

这个栗子声明了一个新的名叫 `tenEighty` 的常量并且设置它引用一个 `VideoMode` 类的新实例，这个视频模式复制了之前的 1920 乘 1080 的 HD 分辨率。同时设置为隔行扫描，并且给予了一个名字“1080i”。最后，设置了 25.0 帧每秒的帧率。

然后，`tenEighty` 是赋给了一个名叫 `alsoEighty` 的新常量，并且将其帧率修改：

```
1 letalsoTenEighty=tenEighty
2 alsoTenEighty.frameRate=30.0
```

因为类是引用类型，`tenEighty` 和 `alsoTenEighty` 其实都是引用了相同的 `VideoMode` 实例。实际上，它们只是相同实例的两个不同命名罢了。

下面，`tenEighty` 的 `frameRate` 属性展示了它正确地显示了来自于 `VideoMode` 实例的新帧率：

```
1 print("The frameRate property of tenEighty is now \tenEighty.frameRate")
2 // prints "The frameRate property of tenEighty is now 30.0"
```

注意 `tenEighty` 和 `alsoTenEighty` 都被声明为常量。然而，你仍然能改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate` 因为 `tenEighty` 和 `alsoTenEighty` 常量本身值不会改变。`tenEighty` 和 `alsoTenEighty` 本身是并没有存储 `VideoMode` 实例—相反，它们两者都在后台指向了 `VideoMode` 实例。这是 `VideoMode` 的 `frameRate` 参数在改变而不是引用 `VideoMode` 的常量的值在改变。

特征运算符

因为类是引用类型，在后台有可能有很多常量和变量都是引用到了同一个类的实例。(相同这个词对结构体和枚举来说并不是真的相同，因为它们在赋予给常量，变量或者被传递给一个函数时总是被拷贝过去的。)

有时候找出两个常量或者变量是否引用自同一个类实例非常有用，为了允许这样，Swift 提供了两个特点运算符：

- 相同于 (`==`)
- 不相同于 (`!=`)

利用这两个恒等运算符来检查两个常量或者变量是否引用相同的实例：

```
1 iftenEighty==alsoTenEighty{
2     print("tenEighty and alsoTenEighty refer to the same VideoMode instance." )
3 }
4 // prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

注意“相同于”(用三个等于号表示，或者说 `==`) 这与“等于”的意义不同(用两个等于号表示，或者说 `==`)。

- “相同于”意味着两个类类型常量或者变量引用自相同的实例；

- “等于”意味着两个实例的在值上被视作“相等”或者“等价”，某种意义上的“相等”，就如同类设计者定义的那样。

当你定义了你自己的自定义类和结构体，你有义务来判定两个实例“相等”的标准。这个定义在你自己的“等于”和“不等于”实现的过程在相等运算符(此处应有链接)中有详细的介绍。

指针

如果你有过 C , C++ 或者 Objective-C 的经验，你可能知道这些语言使用可指针来引用内存中的地址。一个 Swift 的常量或者变量指向某个实例的引用类型和 C 中的指针类似，但是这并不是直接指向内存地址的指针，也不需要你书写星号(*)来明确你建立了一个引用。相反，这些引用被定义得就像 Swift 中其他常量或者变量一样。

类和结构体之间的选择

类和结构体都可以用来定义自定义的数据类型，作为你的程序代码构建块。

总之，结构体实例总是通过值来传递，而类实例总是通过引用传递。这意味着他们分别适用于不同类型的任务。当你考虑你的工程项目中数据结构和功能的时候，你需要决定把每个数据结构定义成类还是结构体。

按照通用准则，当符合以下一条或多条情形时应考虑创建一个结构体：

- 结构体的主要目的是为了封装一些相关的简单数据值；
- 当你在赋予或者传递结构实例时，有理由需要封装的数据值被拷贝而不是引用；
- 任何存储在结构体中的属性是值类型，也将被拷贝而不是被引用；
- 结构体不需要从一个已存在类型继承属性或者行为。

合适的结构体候选者包括：

- 几何形状的大小，可能封装了一个 width 属性和 height 属性，两者都为 double 类型；
- 一定范围的路径，可能封装了一个 start 属性和 length 属性，两者为 Int 类型；
- 三维坐标系的一个点，可能封装了 x, y 和 z 属性，他们都是 double 类型。

在其他的情况下，定义一个类，并创建这个类的实例通过引用传递。事实上，大部分的自定义的数据结构应该是类，而不是结构体。

字符串，数组和字典的赋值与拷贝行为

Swift 的 String , Array 和 Dictionary 类型是作为结构体来实现的，这意味着字符串，数组和字典在它们被赋值到一个新的常量或者变量，亦或者它们本身被传递到一个函数或方法中的时候，其实是传递了拷贝。

这种行为不同于基础库中的 NSString, NSArray 和 NSDictionary，它们是作为类来实现的，而不是结构体。 NSString , NSArray 和 NSDictionary 实例总是作为一个已存在实例的引用而不是拷贝来赋值和传递。

注意

在上述有关字符串，数组和字典“拷贝”的描述中。你在代码中所见到的行为好像总是拷贝。然而在后台 Swift 只有在需要这么做时才会实际去拷贝。Swift 能够管理所有的值的拷贝来确保最佳的性能，所有你也没必要为了保证最佳性能来避免赋值。

译注

[1] CamelCase names : 在给储存器或者函数命名时我们习惯上把多个有意义的单词以开头大写的形式拼接在一起组成一个单一的长单词。这种方法被称为“驼峰式命名法”，又分为开头大写和开头小写两种。比如说 SomeClass 、 frameRate 等。

属性

 cnswift.org/properties

属性可以将值与特定的类、结构体或者是枚举联系起来。存储属性会存储常量或变量作为实例的一部分，反之计算属性会计算（而不是存储）值。计算属性可以由类、结构体和枚举定义。存储属性只能由类和结构体定义。

存储属性和计算属性通常和特定类型的实例相关联。总之，属性也可以与类型本身相关联。这种属性就是所谓的类型属性。

另外，你也可以定义属性观察器来检查属性中值的变化，这样你就可以用自定义的行为来响应。属性观察器可以被添加到你自己定义的存储属性中，也可以添加到子类从他的父类那里所继承来的属性中。

存储属性

在其最简单的形式下，存储属性是一个作为特定类和结构体实例一部分的常量或变量。存储属性要么是变量存储属性（由 var 关键字引入）要么是常量存储属性（由 let 关键字引入）。

正如默认属性值中所述，你可以为存储属性提供一个默认值作为它定义的一部分。你也可以在初始化的过程中设置和修改存储属性的初始值。正如在初始化中分配常量属性所述，这一点对于常量存储属性也成立。

下面的例子定义了一个名为 FixedLengthRange 的结构体，它描述了一个一旦被创建长度就不能改变的整型值域：

```
1 structFixedLengthRange{  
2     varfirstValue:Int  
3     letlength:Int  
4 }  
5 varrangeOfThreeItems=FixedLengthRange(firstValue:0,length:3)  
6 // the range represents integer values 0, 1, and 2  
7 rangeOfThreeItems.firstValue=6  
8 // the range now represents integer values 6, 7, and 8
```

FixedLengthRange 的实例有一个名为 firstValue 的变量存储属性和一个名为 length 的常量存储属性。在上面的例子中，当新的值域创建时 length 已经被创建并且不能再修改，因为这是一个常量属性。

常量结构体实例的存储属性

如果你创建了一个结构体的实例并且把这个实例赋给常量，你不能修改这个实例的属性，即使是声明为变量的属性：

```
1 letrangeOfFourItems=FixedLengthRange(firstValue:0,length:4)  
2 // this range represents integer values 0, 1, 2, and 3  
3 rangeOfFourItems.firstValue=6  
4 // this will report an error, even though firstValue is a variable property
```

由于 `rangeOfFourItems` 被声明为常量（用 `let` 关键字），我们不能改变其 `firstValue` 属性，即使 `firstValue` 是一个变量属性。

这是由于结构体是值类型。当一个值类型的实例被标记为常量时，该实例的其他属性也均为常量。

对于类来说则不同，它是引用类型。如果你给一个常量赋值引用类型实例，你仍然可以修改那个实例的变量属性。

延迟存储属性

延迟存储属性的初始值在其第一次使用时才进行计算。你可以通过在其声明前标注 `lazy` 修饰语来表示一个延迟存储属性。

注意

你必须把延迟存储属性声明为变量（使用 `var` 关键字），因为它的初始值可能在实例初始化完成之前无法取得。常量属性则必须在初始化完成之前有值，因此不能声明为延迟。

一个属性的初始值可能依赖于某些外部因素，当这些外部因素的值只有在实例的初始化完成后才能得到时，延迟属性就可以发挥作用了。而当属性的初始值需要执行复杂或代价高昂的配置才能获得，你又想要在需要时才执行，延迟属性就能够派上用场了。

下面这个栗子使用了一个延迟存储属性来避免复杂类不必要的初始化。这个例子定义了两个名为 `DartImporter` 和 `DartManager` 的类，他们都没有完整显示：

```
1 class DataImporter{  
2   //DataImporter is a class to import data from an external file.  
3   //The class is assumed to take a non-trivial amount of time to initialize.  
4   var fileName="data.txt"  
5   // the DataImporter class would provide data importing functionality here  
6 }  
7 class DataManager{  
8   lazy var importer=DataImporter()  
9   var data=[String]()  
10  // the DataManager class would provide data management functionality here  
11 }  
12 let manager=DataManager()  
13 manager.data.append("Some data")  
14 manager.data.append("Some more data")  
15 // the DataImporter instance for the importer property has not yet been created  
16  
17  
18  
19
```

类 `DataManager` 有一个名为 `data` 的存储属性，它被初始化为一个空的新 `String` 数组。尽管它的其余功能没有展示出来，还是可以知道类 `DataManager` 的目的是管理并提供访问这个 `String` 数组的方法。

`DataManager` 类的功能之一是从文件导入数据。此功能由 `DataImporter` 类提供，它假定为需要一定时间来进行初始化。这大概是因为 `DataImporter` 实例在进行初始化的时候需要打开文件并读取其内容到内存中。

`DataManager` 实例并不要从文件导入数据就可以管理其数据的情况是有可能发生的，所以当

DataManager 本身创建的时候没有必要去再创建一个新的 DataImporter 实例。反之，在 DataImporter 第一次被使用的时候再创建它才更有意义。

因为它被 `lazy` 修饰符所标记，只有在 `importer` 属性第一次被访问时才会创建 DataManager 实例，比如当查询它的 `fileName` 属性时：

```
1 print(manager.importer.fileName)
2 // the DataImporter instance for the importer property has now been created
3 // prints "data.txt"
```

存储属性与实例变量

如果你有 Objective-C 的开发经验，那你应该知道在类实例里有两种方法来存储值和引用。另外，你还可以使用实例变量作为属性中所储存的值的备份存储。

Swift 把这些概念都统一到了属性声明里。Swift 属性没有与之相对应的实例变量，并且属性的后备存储不能被直接访问。这避免了不同环境中对值的访问的混淆并且将属性的声明简化为一条单一的、限定的语句。所有关于属性的信息——包括它的名字，类型和内存管理特征——都作为类的定义放在了同一个地方。

计算属性

除了存储属性，类、结构体和枚举也能够定义计算属性，而它实际并不存储值。相反，他们提供一个读取器和一个可选的设置器来间接得到和设置其他的属性和值。

```
1 structPoint{
2     varx=0.0,y=0.0
3 }
4 structSize{
5     varwidth=0.0,height=0.0
6 }
7 structRect{
8     varorigin=Point()
9     varsize=Size()
10    varcenter: Point{
11        get{
12            letcenterX=origin.x+(size.width/2)
13            letcenterY=origin.y+(size.height/2)
14            returnPoint(x:centerX,y:centerY)
15        }
16        set(newCenter){
17            origin.x=newCenter.x-(size.width/2)
18            origin.y=newCenter.y-(size.height/2)
19        }
20    }
21 }
22 varsquare=Rect(origin:Point(x:0.0,y:0.0),
23 size:Size(width:10.0,height:10.0))
24 letinitialSquareCenter=square.center
25 square.center=Point(x:15.0,y:15.0)
26 print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
27 // prints "square.origin is now at (10.0, 10.0)"
```

这个例子定义了三个结构体来处理几何图形：

- `Point` 封装了一个 `(x,y)` 坐标；
- `Size` 封装了一个 `width` 和 `height`；

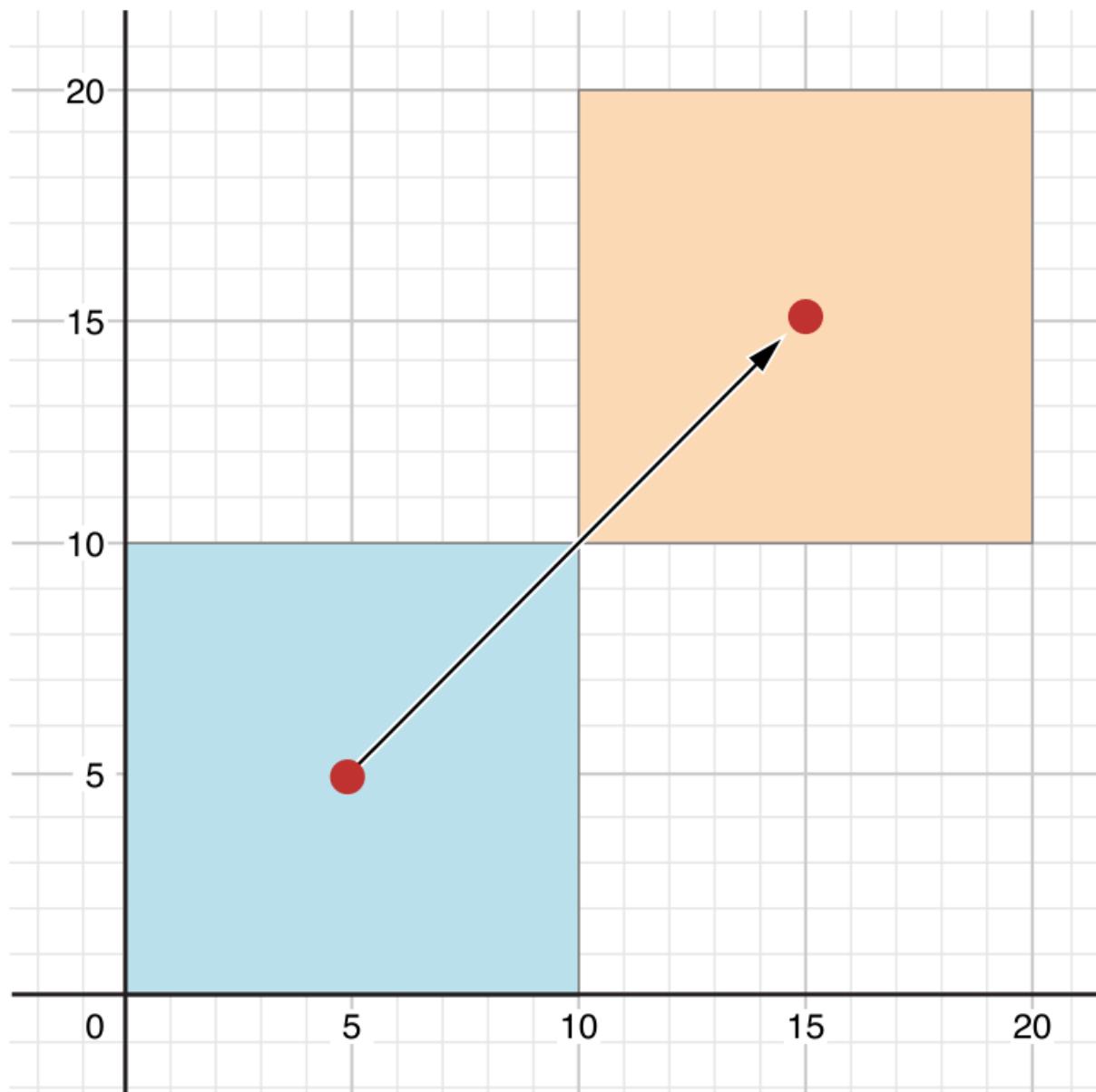
- Rect 封装了一个长方形包括原点和大小。

Rect 结构体还提供了一个名为 center 的计算属性。Rect 的当前中心位置由它的 origin 和 size 决定，所以你不需要把中心点作为一个明确的 Point 值来存储。相反，Rect 为名为 center 的计算变量定义了一个定制的读取器（getter）和设置器（setter），来允许你使用该长方形的 center，就好像它是一个真正的存储属性一样。

前面的例子创建了一个名为 square 的新 Rect 变量。square 变量以 (0,0) 为重心，宽和高为 10 初始化。这个方形在下面的图像中以蓝色方形区域表示。

square 变量的 center 属性通过点操作符（square.center）来访问，通过调用 center 的读取器，来得到当前的属性值。而读取器实际上是计算并返回一个新的 Point 来表示这个方形区域的中心，而不是返回一个已存在的值。如上边你看到的，getter 正确返回了一个值为 (5, 5) 的中心点。

然后 center 属性被赋予新值 (15, 15)，使得该方形区域被移动到了右上方，到达下图中如黄色区域所在的新位置。设置 center 属性调用 center 的设置器方法，通过修改 origin 存储属性中 x 和 y 的值，将该正方形移动到新位置。



简写设置器 (setter) 声明

如果一个计算属性的设置器没有为将要被设置的值定义一个名字，那么他将被默认命名为 newValue。下面是结构体 Rect 的另一种写法，其中利用了简写设置器声明的特性。

```
1 structAlternativeRect{  
2     varorigin=Point()  
3     varszie=Size()  
4     varcenter: Point{  
5         get{  
6             letcenterX=origin.x+(size.width/2)  
7             letcenterY=origin.y+(size.height/2)  
8             returnPoint(x:centerX,y:centerY)  
9         }  
10        set{  
11            origin.x=newValue.x-(size.width/2)  
12            origin.y=newValue.y-(size.height/2)  
13        }  
14    }  
15 }
```

只读计算属性

一个有读取器但是没有设置器的计算属性就是所谓的只读计算属性。只读计算属性返回一个值，也可以通过点语法访问，但是不能被修改为另一个值。

注意

你必须用 var 关键字定义计算属性——包括只读计算属性——为变量属性，因为它们的值不是固定的。let 关键字只用于常量属性，用于明确那些值一旦作为实例初始化就不能更改。

你可以通过去掉 get 关键字和他的大扩号来简化只读计算属性的声明：

```
1 structCuboid{  
2     varwidth=0.0,height=0.0,depth=0.0  
3     varvolume: Double{  
4         returnwidth*height*depth  
5     }  
6 }  
7 letfourByFiveByTwo=Cuboid(width:4.0,height:5.0,depth:2.0)  
8 print("the volume of fourByFiveByTwo is \$(fourByFiveByTwo.volume)")  
9 // prints "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个名为 Cuboid 的新结构体，它代表了一个有 width，height 和 depth 属性的三维长方形结构。这个结构体还有一个名为 volume 的只读计算属性，它计算并返回长方体的当前体积。对于 volume 属性来说可被设置并没有意义，因为它会明确 width，height 和 depth 中哪个值用在特定的 volume 值中，对 Cuboid 来说提供一个只读计算属性来让外部用户来发现它的当前计算体积就显得很有用了。

属性观察者

属性观察者会观察并对属性值的变化做出回应。每当一个属性的值被设置时，属性观察者都会被调用，即使这个值与该属性当前的值相同。

你可以为你定义的任意存储属性添加属性观察者，除了延迟存储属性。你也可以通过在子类里重写属性来为任何继承属性（无论是存储属性还是计算属性）添加属性观察者。属性重载将会在重写中详细描述。

注意

你不需要为非重写的计算属性定义属性观察者，因为你在计算属性的设置器里直接观察和相应它们值的改变。

你可以选择将这些观察者或其中之一定义在属性上：

- willSet 会在该值被存储之前被调用。
- didSet 会在一个新值被存储后被调用。

如果你实现了一个 willSet 观察者，新的属性值会以常量形式参数传递。你可以在你的 willSet 实现中为这个参数定义名字。如果你没有为它命名，那么它会使用默认的名字 newValue 。

同样，如果你实现了一个 didSet 观察者，一个包含旧属性值的常量形式参数将会被传递。你可以为它命名，也可以使用默认的形式参数名 oldValue 。如果你在属性自己的 didSet 观察者里给自己赋值，你赋值的新值就会取代刚刚设置的值。

注意

父类属性的 willSet 和 didSet 观察者会在子类初始化器中设置时被调用。它们不会在类的父类初始化器调用中设置其自身属性时被调用。

更多关于初始化器委托的信息，看[值类型的初始化器委托](#)和[类类型的初始化器委托](#)。

这里有一个关于 willSet 和 didSet 的使用栗子。下面的栗子定义了一个名为 StepCounter 的新类，它追踪人散步的总数量。这个类可能会用于从计步器或者其他计步工具导入数据来追踪人日常的锻炼情况。

```
1 class StepCounter{  
2     var totalSteps:Int=0{  
3         willSet(newTotalSteps){  
4             print("About to set totalSteps to \(newTotalSteps)")  
5         }  
6         didSet{  
7             if totalSteps>oldValue{  
8                 print("Added \(totalSteps-oldValue) steps")  
9             }  
10        }  
11    }  
12 }  
13 let stepCounter=StepCounter()  
14 stepCounter.totalSteps=200  
15 // About to set totalSteps to 200  
16 // Added 200 steps  
17 stepCounter.totalSteps=360  
18 // About to set totalSteps to 360  
19 // Added 160 steps  
20 stepCounter.totalSteps=896  
21 // About to set totalSteps to 896  
22 // Added 536 steps
```

StepCounter 类声明了一个 Int 类型的 totalSteps 属性。这是一个包含了 willSet 和 didSet 观察者的储存属性。

totalSteps 的 willSet 和 didSet 观察者会在每次属性被赋新值的时候调用。就算新值与当前值完全相同也会如此。

栗子中的 willSet 观察者为增量的新值使用自定义的形式参数名 newTotalSteps，它只是简单的打印出将要设置的值。

didSet 观察者在 totalSteps 的值更新后调用。它用旧值对比 totalSteps 的新值。如果总步数增加了，就打印一条信息来表示接收了多少新的步数。 didSet 观察者不会提供自定义的形式参数名给旧值，而是使用 oldValue 这个默认的名字。

注意

如果你以输入输出形式参数传一个拥有观察者的属性给函数， willSet 和 didSet 观察者一定会被调用。这是由于输入输出形式参数的拷贝入拷贝出存储模型导致的：值一定会在函数结束后写回属性。更多关于输入输出形式参数行为的讨论，参见[输入输出形式参数](#)。

全局和局部变量

上边描述的计算属性和观察属性的能力同样对全局变量和局部变量有效。全局变量是定义在任何函数、方法、闭包或者类型环境之外的变量。局部变量是定义在函数、方法或者闭包环境之中的变量。

你在之前章节中所遇到的全局和局部变量都是存储变量。存储变量，类似于存储属性，为特定类型的值提供存储并且允许这个值被设置和取回。

总之，你同样可以定义计算属性以及给存储变量定义观察者，无论是全局还是局部环境。计算变量计算而不是存储值，并且与计算属性的写法一致。

注意

全局常量和变量永远是延迟计算的，与[延迟存储属性](#)有着相同的行为。不同于延迟存储属性，全局常量和变量不需要标记 lazy 修饰符。

类型属性

实例属性是属于特定类型实例的属性。每次你创建这个类型的新实例，它就拥有一堆属性值，与其他实例不同。

你同样可以定义属于类型本身的属性，不是这个类型的某一个实例的属性。这个属性只有一个拷贝，无论你创建了多少个类对应的实例。这样的属性叫做类型属性。

类型属性在定义那些对特定类型的所有实例都通用的值的时候很有用，比如实例要使用的常量属性（类似 C 里的静态常量），或者储存对这个类型的所有实例全局可见的值的存储属性（类似 C 里的静态变量）。

存储类型属性可以是变量或者常量。计算类型属性总要被声明为变量属性，与计算实例属性一致。

注意

不同于存储实例属性，你必须总是给存储类型属性一个默认值。这是因为类型本身不能拥有能够在初始化时给存储类型属性赋值的初始化器。

存储类型属性是在它们第一次访问时延迟初始化的。它们保证只会初始化一次，就算被多个线程同时访问，他们也不需要使用 `lazy` 修饰符标记。

类型属性语法

在 C 和 Objective-C 中，你使用全局静态变量来定义一个与类型关联的静态常量和变量。在 Swift 中，总之，类型属性是写在类型的定义之中的，在类型的花括号里，并且每一个类型属性都显式地放在它支持的类型范围内。

使用 `static` 关键字来开一类型属性。对于类类型的计算类型属性，你可以使用 `class` 关键字来允许子类重写父类的实现。下面的栗子展示了存储和计算类型属性的语法：

```
1 structSomeStructure{
2     staticvarstoredTypeProperty="Some value."
3     staticvarcomputedTypeProperty: Int{
4         return1
5     }
6 }
7 enumSomeEnumeration{
8     staticvarstoredTypeProperty="Some value."
9     staticvarcomputedTypeProperty: Int{
10        return6
11    }
12 }
13 classSomeClass{
14     staticvarstoredTypeProperty="Some value."
15     staticvarcomputedTypeProperty: Int{
16         return27
17     }
18     classvaroverrideableComputedTypeProperty: Int{
19         return107
20     }
21 }
```

注意

上边的计算类型属性示例时对于只读计算类型属性的，但你还是可以使用与计算实例属性相同的语法定义可读写计算类型属性。

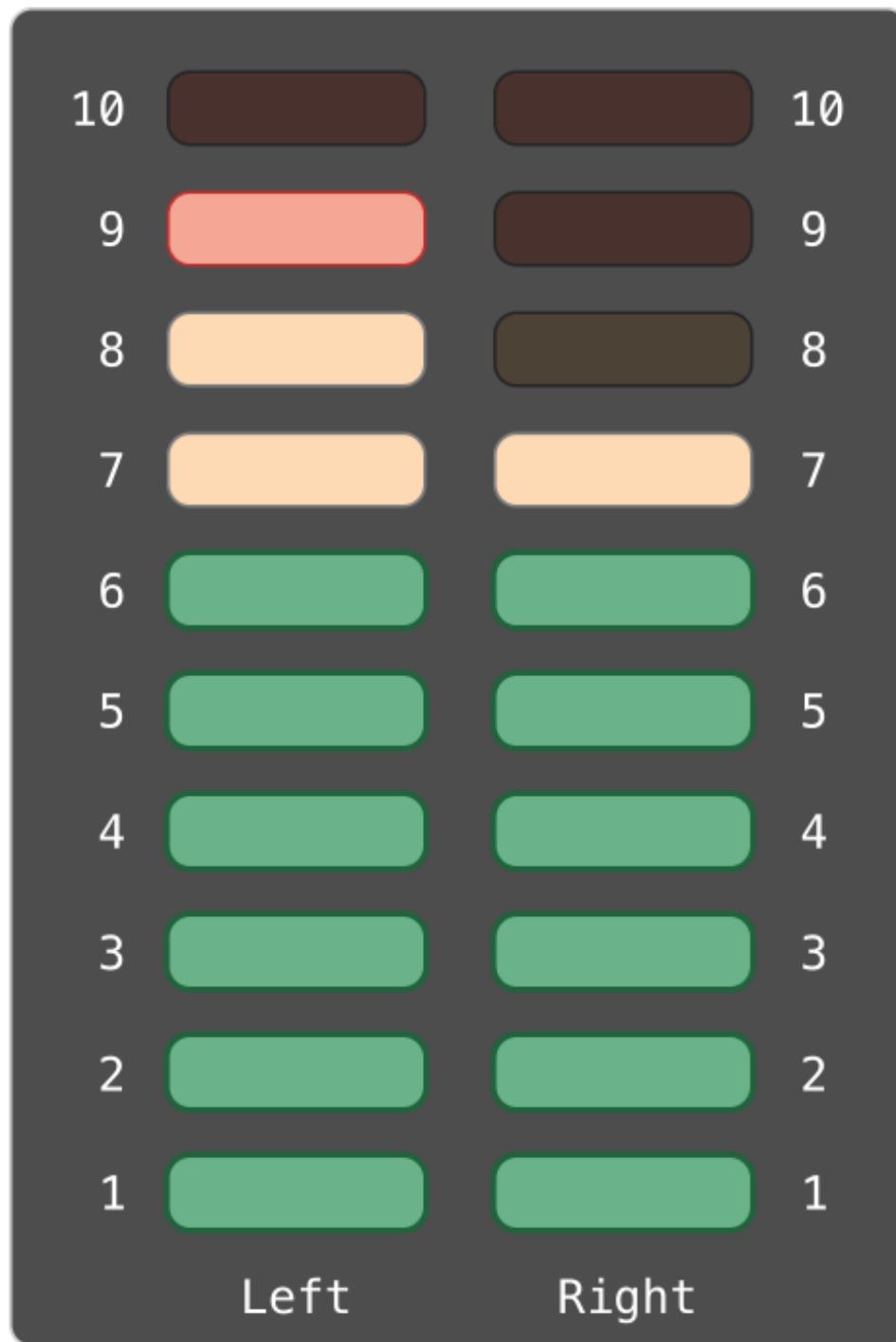
查询和设置类型属性

类型属性使用点语法来查询和设置，与类型属性一致。总之，类型属性在类里查询和设置，而不是这个类型的实例。举例来说：

```
1 print(SomeStructure.storedTypeProperty)
2 // prints "Some value."
3 SomeStructure.storedTypeProperty="Another value."
4 print(SomeStructure.storedTypeProperty)
5 // prints "Another value."
6 print(SomeEnumeration.computedTypeProperty)
7 // prints "6"
8 print(SomeClass.computedTypeProperty)
9 // prints "27"
```

接下来的栗子使用了两个存储类型属性作为建模一个为数字音频信道音频测量表的结构体的一部分。每一个频道都有一个介于 0 到 10 之间的数字音频等级。

下边的图例展示了这个音频频道如何组合建模一个立体声音频测量表。当频道的音频电平为 0，那个对应频道的灯就不会亮。当电平是 10，所有这个频道的灯都会亮。在这个图例里，左声道当前电平是 9，右声道的当前电平是 7：



上边描述的音频声道使用 `AudioChannel` 结构体来表示：

```

1 structAudioChannel{
2     staticlethresholdLevel=10
3     staticvarmaxInputLevelForAllChannels=0
4     varcurrentLevel:Int=0{
5         didSet{
6             ifcurrentLevel>AudioChannel.thresholdLevel{
7                 // cap the new audio level to the threshold level
8                 currentLevel=AudioChannel.thresholdLevel
9             }
10            ifcurrentLevel>AudioChannel.maxInputLevelForAllChannels{
11                // store this as the new overall maximum input level
12                AudioChannel.maxInputLevelForAllChannels=currentLevel
13            }
14        }
15    }
16 }

```

`AudioChannel` 结构体定义了两个存储类型属性来支持它的功能。第一个，`thresholdLevel`，定义了最大限度电平。它是对所有 `AudioChannel` 实例来说是一个常量值 10。（如下方描述的那样）如果传入音频信号值大于 10，则只会被限定在这个限定值上。

第二个类型属性是一个变量存储属性叫做 `maxInputLevelForAllChannels`。它保持追踪任意 `AudioChannel` 实例接收到的最大输入值。它以 0 初始值开始。

`AudioChannel` 结构体同样定义了存储实力属性叫做 `currentLevel`，它表示声道的当前音频电平标量从 0 到 10。

`currentLevel` 属性有一个 `didSet` 属性观察者来检查每次 `currentLevel` 设定的值，这个观察者执行两个检查：

- 如果 `currentLevel` 的新值比允许的限定值要大，属性观察者就限定 `currentLevel` 为 `thresholdLevel`；
- 如果 `currentLevel` 的新值（任何限定之后）比之前任何 `AudioChannel` 实例接收的都高，属性观察者就在 `maxInputLevelForAllChannels` 类型属性里储存 `currentLevel` 的新值。

注意

在这两个检查的第一个中，`didSet` 观察者设置 `currentLevel` 为不同的值。总之，它不会导致观察者的再次调用。

你可以使用 `AudioChannel` 结构体来创建两个新的音频声道 `leftChannel` 和 `rightChannel`，来表示双声道系统的音频等级：

```

1 varleftChannel=AudioChannel()
2 varrightChannel=AudioChannel()

```

1.

如果你设置左声道的 `currentLevel` 为 7，你就会看到 `maxInputLevelForAllChannels` 类型属性被更新到了 7：

```
1 leftChannel.currentLevel=7
2 print(leftChannel.currentLevel)
3 // prints "7"
4 print(AudioChannel.maxInputLevelForAllChannels)
5 // prints "7"
```

1.

如果你尝试去设置右声道的 currentLevel 到 11 ，你就会看到右声道的 currentLevel 属性被上限限制到了最大值 10 ，并且 maxInputLevelForAllChannels 类型属性被更新到了 10 :

1.

```
1 rightChannel.currentLevel=11
2 print(rightChannel.currentLevel)
3 // prints "10"
4 print(AudioChannel.maxInputLevelForAllChannels)
5 // prints "10"
```

方法

 cnswift.org/methods

方法 是关联了特定类型的函数。类，结构体以及枚举都能定义实例方法，方法封装了给定类型特定的任务和功能。类，结构体和枚举同样可以定义类型方法，这是与类型本身关联的方法。类型方法与 Objective-C 中的类方法相似。

事实上在 结构体和枚举中定义方法是 Swift 语言与 C 语言和 Objective-C 的主要区别。在 Objective-C 中，类是唯一能定义方法的类型。但是在 Swift，你可以选择无论类，结构体还是方法，它们都拥有强大的灵活性来在你创建的类型中定义方法。

实例方法

实例方法 是属于特定类实例、结构体实例或者枚举实例的函数。他们为这些实例提供功能性，要么通过提供访问和修改实例属性的方法，要么通过提供与实例目的相关的功能。实例方法与函数的语法完全相同，就如同函数章节里描述的那样。

要写一个实例方法，你需要把它放在对应类的花括号之间。实例方法默认可以访问同类下所有其他实例方法和属性。实例方法只能在类型的具体实例里被调用。它不能在独立于实例而被调用。

这里有个定义 Counter类的栗子，它可以用来计算动作发生的次数：

```
1 classCounter{  
2     var count=0  
3     func increment(){  
4         count+=1  
5     }  
6     func increment(by amount:Int){  
7         count+=amount  
8     }  
9     func reset(){  
10        count=0  
11    }  
12 }
```

Counter 类定义了三个实例方法：

- `increment`每次给计数器增加 1；
- `increment(by:Int)`按照特定的整型数量来增加计数器；
- `reset`会把计数器重置为零。

Counter类同样声明了一个变量属性 `count`来追踪当前计数器的值。

调用实例方法与属性一样都是用点语法：

```
1 letcounter=Counter()
2 // the initial counter value is 0
3 counter.increment()
4 // the counter's value is now 1
5 counter.increment(by:5)
6 // the counter's value is now 6
7 counter.reset()
8 // the counter's value is now 0
```

如同函数实际参数标签和形式参数名中描述的那样，函数的形式参数可以同时拥有一个局部名称（用于函数体）和一个实际参数标签（用于调用函数的时候）。同样的，对于方法的形式参数来说也可以，因为方法就是与类型关联的函数。

self 属性

每一个类的实例都隐含一个叫做 `self` 的属性，它完完全全与实例本身相等。你可以使用 `self` 属性来在当前实例当中调用它自身的方法。

在上边的例子中，`increment()` 方法可以写成这样：

```
1 funcincrement(){
2     self.count+=1
3 }
```

实际上，你不需要经常在代码中写 `self`。如果你没有显式地写出 `self`，Swift 会在你于方法中使用已知属性或者方法的时候假定你是调用了当前实例中的属性或者方法。这个假定通过在 `Counter` 的三个实例中使用 `count`（而不是 `self.count`）来做了示范。

对于这个规则的一个重要例外就是当一个实例方法的形式参数名与实例中某个属性拥有相同的名字的时候。在这种情况下，形式参数名具有优先权，并且调用属性的时候使用更加严谨的方式就很有必要了。你可以使用 `self` 属性来区分形式参数名和属性名。

这时，`self` 就避免了叫做 `x` 的方法形式参数还是同样叫做 `x` 的实例属性这样的歧义。

```
1 structPoint{
2     varx=0.0,y=0.0
3     funcisToTheRightOf(x:Double)->Bool{
4         returnself.x>x
5     }
6 }
7 letsomePoint=Point(x:4.0,y:5.0)
8 ifsomePoint.isToTheRightOf(x:1.0){
9     print("This point is to the right of the line where x == 1.0" )
10 }
11 // Prints "This point is to the right of the line where x == 1.0"
```

除去 `self` 前缀，Swift 将会假定两个 `x` 都是叫做 `x` 的方法形式参数。

在实例方法中修改值类型

结构体和枚举是值类型。默认情况下，值类型属性不能被自身的实例方法修改。

总之，如果你需要在特定的方法中修改结构体或者枚举的属性，你可以选择将这个方法异变。然后这个方法就可以在方法中异变（嗯，改变）它的属性了，并且任何改变在方法结束的时候都会写入到原始的结构体中。方法同样可以指定一个全新的实例给它隐含的 `self` 属性。

性，并且这个新的实例将会在方法结束的时候替换掉现存的这个实例。

你可以选择在 func关键字前放一个 mutating关键字来使用这个行为：

```
1 structPoint{  
2     varx=0.0,y=0.0  
3     mutatingfuncmoveBy(xdeltaX:Double,ydeltaY:Double){  
4         x+=deltaX  
5         y+=deltaY  
6     }  
7 }  
8 varsomePoint=Point(x:1.0,y:1.0)  
9 somePoint.moveBy(x:2.0,y:3.0)  
10 print("The point is now at (\(somePoint.x), \(somePoint.y))")  
11 // prints "The point is now at (3.0, 4.0)"
```

上文中的 Point 结构体定义了一个异变方法 moveBy(x:y:)，它以特定的数值移动一个 Point实例。相比于返回一个新的点，这个方法实际上修改了调用它的点。被添加到定义中的 mutating关键字允许它修改自身的属性。

注意，如同 常量结构体实例的存储属性 里描述的那样，你不能在常量结构体类型里调用异变方法，因为自身属性不能被改变，就算它们是变量属性：

```
1 letfixedPoint=Point(x:3.0,y:3.0)  
2 fixedPoint.moveBy(x:2.0,y:3.0)  
3 // this will report an error
```

在异变方法里指定自身

异变方法可以指定整个实例给隐含的 self属性。上文中那个 Point的栗子可以用下边的代码代替：

```
1 structPoint{  
2     varx=0.0,y=0.0  
3     mutatingfuncmoveBy(xdeltaX:Double,ydeltaY:Double){  
4         self=Point(x:x+deltaX,y:y+deltaY)  
5     }  
6 }
```

这次的异变方法 moveBy(x:y:)创建了一个 x和 y设置在目的坐标的全新的结构体。调用这个版本的方法和的结果会和之前那个完全一样。

枚举的异变方法可以设置隐含的 self属性为相同枚举里的不同成员：

```
1 enumTriStateSwitch{
2     case off, low, high
3     mutating func next(){
4         switch self{
5             case .off:
6                 self = .low
7             case .low:
8                 self = .high
9             case .high:
10                self = .off
11        }
12    }
13 }
14 var ovenLight = TriStateSwitch.low
15 ovenLight.next()
16 // ovenLight is now equal to .high
17 ovenLight.next()
18 // ovenLight is now equal to .off
```

这个栗子定义了一个三种开关状态的枚举。每次调用 `next()`方法时，这个开关就会在三种不同的电力状态（`Off`, `low`和 `high`）下切换。

类型方法

如上文描述的那样，实例方法是特定类型实例中调用的方法。你同样可以定义在类型本身调用的方法。这类方法被称作类型方法。你可以通过在 `func`关键字之前使用 `static`关键字来明确一个类型方法。类同样可以使用 `class`关键字来允许子类重写父类对类型方法的实现。

注意

在 Objective-C 中，你只能在 Objective-C 的类中定义类级别的方法。但是在 Swift 里，你可以在所有的类里定义类级别的方法，还有结构体和枚举。每一个类方法都能够对它自身的类范围显式生效。

类型方法和实例方法一样使用点语法调用。不过，你得在类上调用类型方法，而不是这个类的实例。接下来是一个在 `SomeClass`类里调用类型方法的栗子：

```
1 class SomeClass{
2     class func someTypeMethod(){}
3     // type method implementation goes here
4 }
5 }
6 SomeClass.someTypeMethod()
```

在类型方法的函数体中，隐含的 `self`属性指向了类本身而不是这个类的实例。对于结构体和枚举，这意味着你可以使用 `self`来消除类型属性和类型方法形式参数之间的歧义，用法和实例属性与实例方法形式参数之间的用法完全相同。

一般来说，你在类型方法函数体内书写的任何非完全标准的方法和属性名 都将会指向另一个类级别的方法和属性。一个类型方法可以使用方法名调用另一个类型方法，并不需要使用类型名字作为前缀。同样的，结构体和枚举中的类型方法也可以通过直接使用类型属性名而不需要写类型名称前缀来访问类型属性。

下边的栗子定义了一个叫做 `LevelTracker`的结构体，它通过不同的等级或者阶层来追踪玩家的游戏进度。这是一个单人游戏，但是可以在一个设备上储存多个玩家的信息。

当游戏第一次开始的时候所有的的游戏等级（除了第一级）都是锁定的。每当一个玩家完成一个等级，那这个等级就对设备上的所有玩家解锁。LevelTracker结构体使用类型属性和方法来追踪解锁的游戏等级。它同样追踪每一个独立玩家的当前等级。

```
1 structLevelTracker{  
2     staticvarhighestUnlockedLevel=1  
3     varcurrentLevel=1  
4     staticfuncunlock(_level:Int){  
5         iflevel>highestUnlockedLevel{highestUnlockedLevel=level}  
6     }  
7     staticfuncisUnlocked(_level:Int)->Bool{  
8         returnlevel<=highestUnlockedLevel  
9     }  
10    @discardableResult  
11    mutatingfuncadvance(tolevel:Int)->Bool{  
12        ifLevelTracker.isUnlocked(level){  
13            currentLevel=level  
14            returntrue  
15        }else{  
16            returnfalse  
17        }  
18    }  
19 }  
20  
21  
22
```

LevelTracker结构体持续追踪任意玩家解锁的最高等级。这个值被储存在叫做 highestUnlockedLevel的类型属性里边。

LevelTracker同时还定义了两个类型函数来操作 highestUnlockedLevel属性。第一个类型函数叫做 unlock(_:)，它在新等级解锁的时候更新 highestUnlockedLevel。第二个是叫做 isUnlocked(_:)的便捷类型方法，如果特定等级已经解锁，则返回 true。（注意这些类型方法可以访问 highestUnlockedLevel类型属性而并不需要写作 LevelTracker.highestUnlockedLevel。）

除了其自身的类型属性和类型方法，LevelTracker还追踪每一个玩家在游戏中的进度。它使用一个实例属性 currentLevel来追踪当前游戏中的游戏等级。

为了帮助管理 currentLevel属性，LevelTracker定义了一个叫做 advance(to:)的实例方法。在更新 currentLevel之前，这个方法会检查请求的新等级是否解锁。advance(to:)方法返回一个布尔值来明确它是否真的可以设置 currentLevel。由于调用时忽略 advance(to:) 的返回值并不是什么大不了的问题，这个函数用 @discardableResult 特性。更多关于这个特性的信息，见特性。

看下边的栗子，LevelTracker结构体与 Player类共同使用来追踪和更新每一个玩家的进度：

```
1 classPlayer{  
2     vartracker=LevelTracker()  
3     letplayerName:String  
4     funccomplete(level:Int){  
5         LevelTracker.unlock(level+1)  
6         tracker.advance(to:level+1)  
7     }  
8     init(name:String){  
9         playerName=name  
10    }  
11 }
```

Player类创建了一个新的 LevelTracker实例 来追踪玩家的进度。它同事提供了一个叫做 complete(level:)的方法，这个方法在玩家完成一个特定等级的时候会被调用。这个方法会为所有的玩家解锁下一个等级并且更新玩家的进度到下一个等级。（ advance(to:)返回的布尔值被忽略掉了，因为等级已经在先前调用 LevelTracker.unlock(_:)时已知解锁。）你可以通过为 Player类创建一个实例来新建一个玩家，然后看看当玩家达成等级1时会发生什么：

```
1 varplayer=Player(name:"Argyrios")
2 player.complete(level:1)
3 print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
4 // Prints "highest unlocked level is now 2"
```

如果你创建了第二个玩家，当你让他尝试进入尚未被任何玩家在游戏中解锁的等级时，设置玩家当前等级的尝试将会失败：

```
1 player=Player(name:"Beto")
2 ifplayer.tracker.advance(to:6){
3     print("player is now on level 6")
4 }else{
5     print("level 6 has not yet been unlocked")
6 }
7 // Prints "level 6 has not yet been unlocked"
```

下标

 cnswift.org/subscripts

类、结构体和枚举可以定义下标，它可以作为访问集合、列表或序列成员元素的快捷方式。你可使用下标通过索引值来设置或检索值而不需要为设置和检索分别使用实例方法。比如说，用 `someArray[index]` 来访问 `Array` 实例中的元素以及用 `someDictionary[key]` 访问 `Dictionary` 实例中的元素。

你可以为一个类型定义多个下标，并且下标会基于传入的索引值的类型选择合适的下标重载使用。下标没有限制单个维度，你可以使用多个输入形参来定义下标以满足自定义类型的需求。

下标的语法

下表脚本允许你通过在实例名后面方括号内写一个或多个值对该类的实例进行查询。它的语法类似于实例方法和计算属性。使用关键字 `subscript` 来定义下标，并且指定一个或多个输入形式参数和返回类型，与实例方法一样。与实例方法不同的是，下标可以是读写也可以是只读的。这个行为通过与计算属性中相同的 `getter` 和 `setter` 传达：

```
1 subscript(index:Int)->Int{
2     get{
3         // return an appropriate subscript value here
4     }
5     set(newValue){
6         // perform a suitable setting action here
7     }
8 }
```

`newValue` 的类型和下标的返回值一样。与计算属性一样，你可以选择不去指定 `setter` 的(`newValue`)形式参数。`setter` 默认提供形式参数 `newValue`，如果你自己没有提供的话。

与只读计算属性一样，你可以给只读下标省略 `get` 关键字：

```
1 subscript(index:Int)->Int{
2     // return an appropriate subscript value here
3 }
```

下面是一个只读下标实现的栗子，它定义了一个 `TimeTable` 结构体来表示整数的 n 倍表：

```
1 struct TimesTable{
2     let multiplier:Int
3     subscript(index:Int)->Int{
4         return multiplier*index
5     }
6 }
7 let threeTimesTable=TimesTable(multiplier:3)
8 print("six times three is \(threeTimesTable[6])")
9 // prints "six times three is 18"
```

在这个栗子中，创建了一个 `TimeTable` 的新实例来表示三倍表。它表示通过给结构体的 `initializer` 转入值 3 来作为用于实例的 `multiplier` 形式参数。

你可以通过下标来查询 `threeTimesTable`，比如说调用 `threeTimesTable[6]`。这条获取了三倍表的第六条结果，它返回了值 18 或者 6 的 3 倍。

注意

n倍表是基于固定的数学公式。并不适合对 `threeTimesTable[someIndex]` 进行赋值操作，所以 `TimesTable` 的下标定义为只读下标。

下标用法

“下标”确切的意思取决于它使用的上下文。通常下标是用来访问集合、列表或序列中元素的快捷方式。你可以在你自己特定的类或结构体中自由实现下标来提供合适的功能。

例如，Swift 的 `Dictionary` 类型实现了下标来对 `Dictionary` 实例中存放的值进行设置和读取操作。你可以在下标的方括号中通过提供字典键类型相同的键来设置字典里的值，并且把一个与字典值类型相同的值赋给这个下标：

```
1 varnumberOfLegs = ["spider":8, "ant":6, "cat":4]
2 numberOfLegs["bird"] = 2
```

上面的栗子定义了一个名为 `numberOfLegs` 的变量并用一个字典字面量初始化出了包含三个键值对的字典实例。`numberOfLegs` 字典的类型推断为 `[String:Int]`。字典实例创之后，这个栗子下标赋值来添加一个 `String` 键 "bird" 和 `Int` 值 2 到字典中。

更多关于 `Dictionary` 下标的信息，请参考[访问和修改字典](#)。

注意

Swift 的 `Dictionary` 类型实现它的下标为接收和返回可选类型的下标。对于上例中的 `numberOfLegs` 字典，键值下标接收和返回一个 `Int?` 类型的值，或者说“可选的 `Int`”。`Dictionary` 类型使用可选的下标类型来建模不是所有键都会有值的事实，并且提供了一种通过给键赋值为 `nil` 来删除对应键的值的方法。不

下标选项

下标可以接收任意数量的输入形式参数，并且这些输入形式参数可以是任意类型。下标也可以返回任意类型。下标可以使用变量形式参数和可变形式参数，但是不能使用输入输出形式参数或提供默认形式参数值。

类或结构体可以根据自身需要提供多个下标实现，合适被使用的下标会基于值类型或者使用下标时下标方括号里包含的值来推断。这个对多下标的定义就是所谓的 [下标重载](#)。

通常来讲下标接收一个形式参数，但只要你的类型需要也可以为下标定义多个参数。如下例定义了一个 `Matrix` 结构体，它呈现一个 `Double` 类型的二维矩阵。`Matrix` 结构体的下标接收两个整数形式参数：

```

1 structMatrix{
2     letrows:Int,columns:Int
3     vargrid:[Double]
4     init(rows:Int,columns:Int){
5         self.rows=rows
6         self.columns=columns
7         grid=Array(repeating:0.0,count:rows*columns)
8     }
9     funcindexIsValid(row:Int,column:Int)->Bool{
10        return row>=0 && row<rows && column>=0 && column<columns
11    }
12    subscript(row:Int,column:Int)->Double{
13        get{
14            assert(indexIsValid(row:row,column:column),"Index out of range")
15            return grid[(row*columns)+column]
16        }
17        set{
18            assert(indexIsValid(row:row,column:column),"Index out of range")
19            grid[(row*columns)+column]=newValue
20        }
21    }
22}

```

Matrix 提供了一个接收 rows 和 columns 两个形式参数的初始化器，创建了一个足够容纳 rows*columns 个数的 Double 类型数组。矩阵里的每个位置都用 0.0 初始化。要这么做，数组的长度，每一格初始化为 0.0，都传入数组初始化器来创建和初始化一个正确长度的数组。这个初始化器在使用默认值创建数组里有更详细的描述。

你可以通过传入合适的行和列的数量来构造一个新的 Matrix 实例：

```
1 varmatrix=Matrix(rows:2,columns:2)
```

上例中创建了一个新的两行两列的 Matrix 实例。Matrix 实例中的数组 `grid` 是 矩阵的高效扁平化版本，阅读顺序从左上到右下：

$$\text{grid} = \begin{bmatrix} 0.0, & 0.0, & 0.0, & 0.0 \end{bmatrix}$$

$$\begin{array}{c}
 & \text{column} \\
 & 0 \quad 1 \\
 \text{row} & \left[\begin{array}{cc} 0.0, & 0.0, \\ 0.0, & 0.0 \end{array} \right]
 \end{array}$$

矩阵里的值可以通过传行和列给下标来设置，用逗号分隔：

```
1 matrix[0,1]=1.5  
2 matrix[1,0]=3.2
```

上面两条语句调用的下标的设置器来给矩阵的右上角（row是0，column是1）赋值为1.5，给左下角（row是1，column是0）赋值为3.2：

Matrix下标的设置器和读取器都包含了一个断言来检查下标的row和column是否有效。为了方便进行断言，Matrix包含了一个名为indexIsValidForRow(_:column:)的成员方法，它用来确认请求的row或column值是否会造数组越界：

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

```
1 func indexIsValidForRow(row:Int,column:Int)->Bool{  
2   return row>=0 && row<rows && column>=0 && column<columns  
3 }
```

断言在下标越界时触发：

```
1 let someValue=matrix[2,2]  
2 // this triggers an assert, because [2, 2] is outside of the matrix bounds
```

继承

 cnswift.org/inheritance

一个类可以从另一个类继承方法、属性和其他的特性。当一个类从另一个类继承的时候，继承的类就是所谓的子类，而这个类继承的类被称为父类。在 Swift 中，继承与其他类型不同的基础分类行为。

在 Swift 中类可以调用和访问属于它们父类的方法、属性和下标脚本，并且可以提供它们自己重写的方法，属性和下标脚本来定义或修改它们的行为。Swift 会通过检查重写定义都只有一个与之匹配的父类定义来确保你的重写是正确的。

类也可以向继承的属性添加属性观察器，以便在属性的值改变时得到通知。可以添加任何属性监视到属性中，不管它是被定义为存储还是计算属性。

定义一个基类

任何不从另一个类继承的类都是所谓的基类。

注意

Swift 类不会从一个通用基类继承。你没有指定特定父类的类都会以基类的形式创建。

下面的栗子定义了一个叫做 Vehicle 的基类。这个基类定义了一个称为 currentSpeed 的存储属性，使用默认值 0.0 (推断为一个 Double 类型的属性)。currentSpeed 属性的值被用在一个称为 description 的 String 只读计算属性来创建一个 vehicle 的描述。

Vehicle 基类也定义了一个称为 makeNoise 的方法。这个方法实际上不会为这个 Vehicle 基类的实例做任何事，但是稍后它可以被 Vehicle 的子类自定义：

```
1 class Vehicle{  
2     var currentSpeed = 0.0  
3     var description: String{  
4         return "traveling at \(currentSpeed) miles per hour"  
5     }  
6     func makeNoise(){  
7         // do nothing - an arbitrary vehicle doesn't necessarily make a noise  
8     }  
9 }
```

你使用初始化语法创建了一个新的 Vehicle 实例，写为 类型名 后面跟着一个空括号：

```
1 let someVehicle = Vehicle()
```

在创建了一个新的 Vehicle 实例之后，你可以访问它的 description 属性来输出一个人类可读的汽车当前速度的描述：

```
1 print("Vehicle: \(someVehicle.description)")  
2 // Vehicle: traveling at 0.0 miles per hour  
3
```

Vehicle 类为任意的车辆定义了共同的特征，但是对它本身没有太大用处。为了让它更有用，你需要重定义它来描述更具体的车辆种类。

子类

子类是基于现有类创建新类的行为。子类从现有的类继承了一些特征，你可以重新定义它们。你也可以为子类添加新的特征。

为了表明子类有父类，要把子类写在父类的前面，用冒号分隔：

```
1 class SomeSubclass: SomeSuperclass{  
2 // subclass definition goes here  
3 }
```

下面的例子定义了一个称为 Bicycle 的子类，继承自 Vehicle：

```
1 class Bicycle: Vehicle{  
2 var hasBasket=false  
3 }
```

新的 Bicycle 类自动获得了 Vehicle 的所有特征，例如它的 currentSpeed 和 description 属性以及 makeNoise() 方法。

除了继承的特征，Bicycle 类定义了一个新的存储属性 hasBasket，并且默认值为 false (属性的类型被推断为 Bool)。

默认情况下，任何你新建的 Bicycle 实例都不会有篮子。在 Bicycle 类的实例创建之后，你可以将它的 hasBasket 属性设置为 true：

```
1 let bicycle=Bicycle()  
2 bicycle.hasBasket=true  
3
```

你也可以在 Bicycle 类实例中修改继承而来的 currentSpeed 属性，或是查询实例中继承的 description 属性：

```
1 bicycle.currentSpeed=15.0  
2 print("Bicycle: \$(bicycle.description)")  
3 // Bicycle: traveling at 15.0 miles per hour  
4  
5
```

子类本身也可以被继承。下个栗子创建了一个 Bicycle 的子类，称为“tandem”的两座自行车：

```
1 class Tandem: Bicycle{  
2 var currentNumberOfPassengers=0  
3 }
```

Tandem 继承了 Bicycle 中所有的属性和方法，也继承了 Vehicle 的所有属性和方法。Tandem 子类也添加了一个新的称为 currentNumberOfPassengers 的存储属性，并且有一个默认值 0：

```
1 lettandem=Tandem()
2 tandem.hasBasket=true
3 tandem.currentNumberOfPassengers=2
4 tandem.currentSpeed=22.0
5 print("Tandem: \"(tandem.description)")
6 // Tandem: traveling at 22.0 miles per hour
7
8
9
10
11
```

重写

子类可以提供它自己的实例方法、类型方法、实例属性，类型属性或下标脚本的自定义实现，否则它将会从父类继承。这就所谓的重写。

要重写而不是继承一个特征，你需要在你的重写定义前面加上 `override` 关键字。这样做说明你打算提供一个重写而不是意外提供了一个相同定义。意外的重写可能导致意想不到的行为，并且任何没有使用 `override` 关键字的重写都会在编译时被诊断为错误。

`override` 关键字会执行 Swift 编译器检查你重写的类的父类(或者父类的父类)是否有与之匹配的声明来供你重写。这个检查确保你重写的定义是正确的。

访问父类的方法、属性和下标脚本

当你为子类提供了一个方法、属性或者下标脚本时，有时使用现有的父类实现作为你重写的一部分是很有用的。比如说，你可以重新定义现有实现的行为，或者在现有继承的变量中存储一个修改过的值。

你可以通过使用 `super` 前缀访问父类的方法、属性或下标脚本，这是合适的：

- 一个命名为 `someMethod()` 的重写方法可以通过 `super.someMethod()` 在重写方法的实现中调用父类版本的 `someMethod()` 方法；
- 一个命名为 `someProperty` 的重写属性可以通过 `super.someProperty` 在重写的 `getter` 或 `setter` 实现中访问父类版本的 `someProperty` 属性；
- 一个命名为 `someIndex` 的重写下标脚本可以使用 `super[someIndex]` 在重写的下标脚本实现中访问父类版本中相同的下标脚本。

重写方法

你可以在你的子类中重写一个继承的实例或类型方法来提供定制的或替代的方法实现。

下面的栗子定义了一个新的 `Vehicle` 子类，称为 `Train`，它重写了 `Train` 继承自 `Vehicle` 的 `makeNoise()` 方法：

```
1 class Train: Vehicle{
2     override func makeNoise(){
3         print("Choo Choo")
4     }
5 }
```

如果你创建了一个新的 `Train` 实例并且调用它的 `makeNoise()` 方法，你可以看到 `Train` 子类版本的方法被调用了：

```
1 let train=Train()  
2 train.makeNoise()  
3 // prints "Choo Choo"  
4  
5
```

重写属性

你可以重写一个继承的实例或类型属性来为自己的属性提供你自己自定义的 getter 和 setter，或者添加属性观察器确保当底层属性值改变时来监听重写的属性。

重写属性的Getter和Setter

你可以提供一个自定义的 Getter(和 Setter，如果合适的话)来重写任意继承的属性，无论在最开始继承的属性实现为储属性还是计算属性。继承的属性是存储还是计算属性不对子类透明——它仅仅知道继承的属性有个特定名字和类型。你必须声明你重写的属性名字和类型，以确保编译器可以检查你的重写是否匹配了父类中有相同名字和类型的属性。

你可以通过在你的子类重写里为继承而来的只读属性添加 Getter 和 Setter 来把它用作可读写属性。总之，你不能把一个继承而来的可读写属性表示为只读属性。

注意

如果你提供了一个 setter 作为属性重写的一部分，你也就必须为重写提供一个 getter。如果你不想在重写 getter 时修改继承属性的值，那么你可以简单通过从 getter 返回 super.someProperty 来传递继承的值，someProperty 就是你重写那个属性的名字。

下面的栗子定义了一个叫做 Car 的新类，它是 Vehicle 的子类。Car 类引入了一个新的存储属性 gear，并且有一个默认的整数值 1。Car 类也重写了继承自 Vehicle 的 description 属性，来提供自定义的描述，介绍当前的档位：

```
1 class Car: Vehicle{  
2     var gear=1  
3     override var description: String{  
4         return super.description + " in gear \\" + (gear)  
5     }  
6 }
```

description 属性的重写以调用 super.description 开始，它返回了 Vehicle 类的 description 属性。Car 类的 description 随后就添加了一些额外的文本到描述的末尾以提供关于当前档位的信息。

如果你创建一个 Car 类的实例并且设置它的 gear 和 currentSpeed 属性，你就可以看到它的 description 属性在 Car 类的定义里返回了定制的描述：

```
1 let car=Car()  
2 car.currentSpeed=25.0  
3 car.gear=3  
4 print("Car: \\" + car.description + "")  
5 // Car: traveling at 25.0 miles per hour in gear 3  
6  
7  
8  
9
```

重写属性观察器

你可以使用属性重写来为继承的属性添加属性观察器。这就可以让你在继承属性的值改变时得到通知，无论这个属性最初如何实现。关于属性观察器的更多信息，移步属性观察器（此处应有链接）。

注意

你不能给继承而来的常量存储属性或者只读的计算属性添加属性观察器。这些属性的值不能被设置，所以提供 willSet 或 didSet 实现作为重写的一部分也是不合适的。

也要注意你不能为同一个属性同时提供重写的setter和重写的属性观察器。如果你想要监听属性值的改变，并且你已经为那个属性提供了一个自定义的setter，那么你从自定义的setter里就可以监听任意值的改变。

下面的例子定义了一个叫做 AutomaticCar 的新类，它是 Car 的子类。AutomaticCar 类代表一辆车有一个自动的变速箱，可以根据当前的速度自动地选择一个合适的档位：

```
1 class AutomaticCar: Car{  
2     override var currentSpeed: Double{  
3         didSet{  
4             gear = Int(currentSpeed / 10.0) + 1  
5         }  
6     }  
7 }
```

无论你在什么时候设置了 AutomaticCar 实例的 currentSpeed 属性，属性的 didSet 观察器都会设置实例的 gear 属性为新速度设置一个合适的档位。具体地说，属性观察器选择的档位就是新的 currentSpeed 值除以 10，四舍五入到最近整数，加 1。速度是 35.0 就对应 4：

```
1 let automatic = AutomaticCar()  
2 automatic.currentSpeed = 35.0  
3 print("AutomaticCar: \(automatic.description)")  
4 // AutomaticCar: traveling at 35.0 miles per hour in gear 4  
5  
6  
7
```

阻止重写

你可以通过标记为终点来阻止一个方法、属性或者下标脚本被重写。通过在方法、属性或者下标脚本的关键字前写 final 修饰符(比如 final var , final func , final classfunc , final subscript)。

任何在子类里重写终点方法、属性或下标脚本的尝试都会被报告为编译时错误。你在扩展中添加到类的方法、属性或下标脚本也可以在扩展的定义里被标记为终点。

你可以通过在类定义中在 class 关键字前面写 final 修饰符(final class)标记一整个类为终点。任何想要从终点类创建子类的行为都会被报告一个编译时错误。

初始化

 cnswift.org/initialization

初始化是为类、结构体或者枚举准备实例的过程。这个过程需要给实例里的每一个存储属性设置一个初始值并且在新实例可以使用之前执行任何其他所必须的配置或初始化。

你通过定义 **初始化器** 来实现这个初始化过程，它更像是一个用来创建特定类型新实例的特殊的方法。不同于 Objective-C 的初始化器，Swift 初始化器不返回值。这些初始化器主要的角色就是确保在第一次使用之前某类型的新实例能够正确初始化。

类类型的实例同样可以实现一个 **反初始化器**，它会在这个类的实例被释放之前执行任意的自定义清理。更多关于反初始化器的信息，请看[反初始化](#)。

为存储属性设置初始化值

在创建类和结构体的实例时必须为所有的存储属性设置一个合适的初始值。存储属性不能遗留在不确定的状态中。

你可以在初始化器里为存储属性设置一个初始值，或者通过分配一个默认的属性值作为属性定义的一部分。在下面的小节中会描述这些动作。

注意

当你给一个存储属性分配默认值，或者在一个初始化器里设置它的初始值的时候，属性的值就会被直接设置，不会调用任何属性监听器。

初始化器

初始化器在创建特定类型的实例时被调用。在这个简单的形式中，初始化器就像一个没有形式参数的实例方法，使用 `init` 关键字来写：

```
1 init(){  
2 // perform some initialization here  
3 }
```

下面的栗子定义了一个名为 `Fahrenheit` 结构体以储存华氏度表示的温度。`Fahrenheit` 结构体有一个 `Double` 类型的存储属性 `temperature`：

```
1 structFahrenheit{  
2 vartemperature:Double  
3 init(){  
4 temperature=32.0  
5 }  
6 }  
7 varf=Fahrenheit()  
8 print("The default temperature is \(f.temperature)° Fahrenheit")  
9 // prints "The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个初始化器，`init`，没有形式参数，它初始化储存温度的值为 32.0 (在华氏温度下水的冰点)。

默认的属性值

如上所述，你可以在初始化器里为存储属性设置初始值。另外，指定一个默认属性值作为属性声明的一部分。当属性被定义的时候你可以通过为这个属性分配一个初始值来指定默认的属性值。

注意

如果一个属性一直保持相同的初始值，可以提供一个默认值而不是在初始化器里设置这个值。最终结果是一样的，但是默认值将属性的初始化与声明更紧密地联系到一起。它使得你的初始化器更短更清晰，并且可以让你属性根据默认值推断类型。如后边的章节所述，默认值也让你使用默认初始化器和初始化器继承更加容易。

通过提供 `temperature` 属性的默认值，你可以把上面的 `Fahrenheit` 结构体写的更简单：

```
1 structFahrenheit{  
2     vartemperature=32.0  
3 }
```

自定义初始化

如同下面章节所述，你可以通过输入形式参数和可选类型来自定义初始化过程，或者在初始化的时候分配常量属性。

初始化形式参数

你可以提供初始化形式参数作为初始化器的一部分，来定义初始化过程中的类型和值的名称。初始化形式参数与函数和方法的形式参数具有相同的功能和语法。

下面的栗子定义了一个名为 `Celsius` 的结构体，它用摄氏度表示储存温度。`Celsius` 结构体实现了两个自定义的初始化器 `init(fromFahrenheit:)` 和 `init(fromKelvin:)`，它们用不同的温度单位初始化新的结构体实例：

```
1 structCelsius{  
2     vartemperatureInCelsius:Double  
3     init(fromFahrenheit fahrenheit:Double){  
4         temperatureInCelsius=(fahrenheit-32.0)/1.8  
5     }  
6     init(fromKelvin kelvin:Double){  
7         temperatureInCelsius=kelvin-273.15  
8     }  
9 }  
10 letboilingPointOfWater=Celsius(fromFahrenheit:212.0)  
11 // boilingPointOfWater.temperatureInCelsius is 100.0  
12 letfreezingPointOfWater=Celsius(fromKelvin:273.15)  
13 // freezingPointOfWater.temperatureInCelsius is 0.0
```

第一个初始化器只有一个外部变量名 `fromFahrenheit` 和一个局部变量名 `fahrenheit` 的初始化形式参数。第二个初始化器只有一个外部变量名 `fromKelvin` 和一个局部变量名 `kelvin` 的初始化形式参数。这两个初始化器都把它们的实际参数转换为了摄氏度并且把这个值储存到了名为 `temperatureInCelsius` 的属性里。

形式参数名和实际参数标签

与函数和方法的形式参数一样，初始化形式参数也可以在初始化器内部有一个局部变量名以及实际参数标签供调用的时候使用。

总之，初始化器并不能像函数和方法那样在圆括号前面有一个用来区分的函数名。因此，一个初始化器的参数名称和类型在识别该调用哪个初始化器的时候就扮演了一个非常重要的角色。因此，如果你没有提供外部名 Swift 会自动为每一个形式参数提供一个外部名称。

下面的栗子定义了一个名为 Color 的结构体，它有三个常量属性，分别为 red , green 和 blue 。这些属性储存了一个介于 0.0 到 1.0 之间的值来表示颜色里的红、绿、蓝。

Color 给它的红绿蓝组合提供了一个初始化器，它带有三个合适名称的 Double 类型形式参数。 Color 同样提供了第二个只有一个 white 形式参数的初始化器，它用来给三个颜色组合设置相同的值：

```
1 structColor{  
2     letred,green,blue:Double  
3     init(red:Double,green:Double,blue:Double){  
4         self.red=red  
5         self.green=green  
6         self.blue=blue  
7     }  
8     init(white:Double){  
9         red=white  
10        green=white  
11        blue=white  
12    }  
13 }
```

通过为每一个初始化器的形式参数提供一个初始值，初始化器可以用来创建新的 Color 实例：

```
1 letmagenta=Color(red:1.0,green:0.0,blue:1.0)  
2 lethalfGray=Color(white:0.5)
```

注意不使用外部名称是不能调用这些初始化器的。如果定义了外部参数名就必须用在初始化器里，省略的话会报一个编译时错误：

```
1 letveryGreen=Color(0.0,1.0,0.0)  
2 // this reports a compile-time error - external names are required
```

无实际参数标签的初始化器形式参数

如果你不想为初始化器形式参数使用实际参数标签，可以写一个下划线(_)替代明确的实际参数标签以重写默认行为。

这里有一个之前 Celsius 类的扩展版本栗子，它有一个额外的初始化器来从已经是摄氏度的 Double 值创建一个新的 Celsius 类实例：

```
1 struct Celsius{  
2     var temperatureInCelsius: Double  
3     init(fromFahrenheit fahrenheit: Double){  
4         temperatureInCelsius = (fahrenheit - 32.0) / 1.8  
5     }  
6     init(fromKelvin kelvin: Double){  
7         temperatureInCelsius = kelvin - 273.15  
8     }  
9     init(_celsius: Double){  
10        temperatureInCelsius = _celsius  
11    }  
12 }  
13 let bodyTemperature = Celsius(37.0)  
14 // bodyTemperature.temperatureInCelsius is 37.0
```

调用初始化器 `Celsius(37.0)` 有着清楚的意图而不需要外部形式参数名。因此，把初始化器写为 `init(_celsius:Double)` 是合适的，它也就可以通过提供未命名的 `Double` 值被调用了。

可选属性类型

如果你的自定义类型有一个逻辑上是允许“无值”的存储属性——大概因为它的值在初始化期间不能被设置，或者因为它在稍后允许设置为“无值”——声明属性为 **可选** 类型。可选类型的属性自动地初始化为 `nil`，表示该属性在初始化期间故意设为“还没有值”。

下面的栗子定义了一个名为 `SurveyQuestion` 的类，有一个可选 `String` 属性，名为 `response`：

```
1 class SurveyQuestion{  
2     var text: String  
3     var response: String?  
4     init(text: String){  
5         self.text = text  
6     }  
7     func ask(){  
8         print(text)  
9     }  
10 }  
11 let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")  
12 cheeseQuestion.ask()  
13 // prints "Do you like cheese?"  
14 cheeseQuestion.response = "Yes, I do like cheese."
```

对调查问题的回答直到被问的时候才能知道，所以 `response` 属性被声明为 `String?` 类型，或者是“可选 `String`”。当新的 `SurveyQuestion` 实例被初始化的时候，它会自动分配一个为 `nil` 的默认值，意味着“还没有字符串”。

在初始化中分配常量属性

在初始化的任意时刻，你都可以给常量属性赋值，只要它在初始化结束时设置了确定的值即可。一旦为常量属性被赋值，它就不能再被修改了。

注意

对于类实例来说，常量属性在初始化中只能通过引用的类来修改。它不能被子类修改。

你可以修改上面 SurveyQuestion 的例子，给 text 使用常量属性而不是变量属性来表示问题，来明确一旦 SurveyQuestion 的实例被创建，那个问题将不会改变。尽管现在 text 属性是一个常量，但是它依然可以在类的初始化器里设置：

```
1 class SurveyQuestion{  
2     let text: String  
3     var response: String?  
4     init(text: String){  
5         self.text = text  
6     }  
7     func ask(){  
8         print(text)  
9     }  
10    }  
11    let beetsQuestion = SurveyQuestion(text: "How about beets?")  
12    beetsQuestion.ask()  
13    // prints "How about beets?"  
14    beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

默认初始化器

Swift 为所有没有提供初始化器的结构体或类提供了一个默认的初始化器来给所有的属性提供了默认值。这个默认的初始化器只是简单地创建了一个所有属性都有默认值的新实例。

这个栗子定义了一个名为 ShoppingListItem 的类，它在购物列表的物品里封装了名称，数量和购买状态属性：

```
1 class ShoppingListItem{  
2     var name: String?  
3     var quantity = 1  
4     var purchased = false  
5 }  
6 var item = ShoppingListItem()
```

由于 ShoppingListItem 类的所有属性都有默认值，又由于它是一个没有父类的基类， ShoppingListItem 类自动地获得了一个默认的初始化器，使用默认值设置了它的所有属性然后创建了新的实例。(name 属性是一个可选 String 属性，所以它会自动设置为 nil 默认值，尽管这个值没有写在代码里。)上面的栗子给 ShoppingListItem 类使用默认初始化器以及初始化器语法创建新的实例，写作 ShoppingListItem()，并且给这个新实例赋了一个名为 item 的变量。

结构体类型的成员初始化器

如果结构体类型中没有定义任何自定义初始化器，它会自动获得一个成员初始化器。不同于默认初始化器，结构体会接收成员初始化器即使它的存储属性没有默认值。

这个成员初始化器是一个快速初始化新结构体实例成员属性的方式。新实例的属性初始值可以通过名称传递到成员初始化器里。

下面的栗子定义了一个名为 Size 有两个属性分别是 width 和 height 的结构体，这两个属性通过分配默认值 0.0，从而被推断为 Double 类型。

Size 结构体自动接收一个 init(width:height:) 成员初始化器，你可以使用它来初始化一个新的 Size 实例：

```
1 structSize{  
2     varwidth=0.0,height=0.0  
3 }  
4 lettwoByTwo=Size(width:2.0,height:2.0)
```

值类型的初始化器委托

初始化器可以调用其他初始化器来执行部分实例的初始化。这个过程，就是所谓的 **初始化器委托**，避免了多个初始化器里冗余代码。

初始化器委托的运作，以及允许那些形式的委托，这些规则对于值类型和类类型是不同的。值类型(结构体和枚举)不支持继承，所以他它们的初始化器委托的过程相当简单，因为它们只能提供它们自己为另一个初始化器委托。如同继承里描述的那样，总之，类可以从其他类继承。这就意味着类有额外的责任来确保它们继承的所有存储属性在初始化期间都分配了一个合适的值。这些责任在下边的类的继承和初始化里做详述。

对于值类型，当你写自己自定义的初始化器时可以使用 `self.init` 从相同的值类型里引用其他初始化器。你只能从初始化器里调用 `self.init`。

注意如果你为值类型定义了自定义初始化器，你就不能访问那个类型的默认初始化器(或者是成员初始化器，如果是结构体的话)。这个限制防止别人意外地使用自动初始化器而把复杂初始化器里提供的额外必要配置给覆盖掉的情况发生。

注意

如果你想要你自己的自定义值类型能够使用默认初始化器和成员初始化器初始化，以及你的自定义初始化器来初始化，把你的自定义初始化器写在扩展里而不是作为值类型原始实的一部分。想要了解更多的信息，请看[扩展](#)。

下面的栗子定义了一个自定义的 `Rect` 结构体代表几何矩形。这个栗子需要两个结构体，分别是 `Size` 和 `Point`，都为他们各自的属性默认值都是 0.0：

```
1 structSize{  
2     varwidth=0.0,height=0.0  
3 }  
4 structPoint{  
5     varx=0.0,y=0.0  
6 }
```

你可以用三个方式中的任意一个来初始化下面的 `Rect` 结构体——通过使用默认赋零初始化 `origin` 和 `size` 属性值，通过提供一个具体的原点坐标和大小，或者提供一个具体的中心点和大小。这三种初始化选项通过三个写在 `Rect` 结构体定义里的自定义初始化器代表：

```
1 structRect{
2     var origin=Point()
3     var size=Size()
4     init(){}
5     init(origin:Point,size:Size){
6         self.origin=origin
7         self.size=size
8     }
9     init(center:Point,size:Size){
10        let originX=center.x-(size.width/2)
11        let originY=center.y-(size.height/2)
12        self.init(origin:Point(x:originX,y:originY),size:size)
13    }
14 }
```

第一个 Rect 初始化器，`init()`，和默认初始化器有一样的功能，就是那个如果 Rect 没有自定义初始化器，它将会使用的那个默认初始化器。这个初始化器是空的，用一个大括号 {} 来表示，并且不会执行任何初始化。调用这个初始化器会返回一个 origin 和 size 属性都被初始化为默认值 0.0 的 Rect 实例：

```
1 let basicRect=Rect()
2 // basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

第二个 Rect 初始化器，`init(origin:size)`，和成员初始化器功能相同，就是如果 Rect 没有自定义的初始化器，它将使用的那个初始化器。这个初始化器只是把 origin 和 size 实际参数值赋值给合适的存储属性：

```
1 let originRect=Rect(origin:Point(x:2.0,y:2.0),
2 size:Size(width:5.0,height:5.0))
3 // originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

第三个 Rect 的初始化器，`init(center:size:)`，略显复杂。它以计算一个基于 center 和 size 值的原点开始。然后调用(或是委托) `init(origin:size:)` 初始化器，它在合适的属性里储存了新的原点和大小值：

```
1 let centerRect=Rect(center:Point(x:4.0,y:4.0),
2 size:Size(width:3.0,height:3.0))
3 // centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

`init(center:size:)` 初始化器可能自己已经为 origin 和 size 属性赋值了新值。总之，对于 `init(center:size:)` 初始化器来说，可以更方便(更清楚)地利用现有已经提供了准确功能的初始化器。

注意

另一种方法是不需要你自己定义 `init()` 和 `init(origin:size:)` 初始化器，请看[扩展](#)。

类的继承和初始化

所有类的存储属性——包括从它的父类继承的所有属性——都必须在初始化期间分配初始值。

Swift 为类类型定义了两种初始化器以确保所有的存储属性接收一个初始值。这些就是所谓的指定初始化器和便捷初始化器。

指定初始化器和便捷初始化器

指定初始化器是类的主要初始化器。指定的初始化器可以初始化所有那个类引用的属性并且调用合适的父类初始化器来继续这个初始化过程给父类链。

类偏向于少量指定初始化器，并且一个类通常只有一个指定初始化器。指定初始化器是初始化开始并持续初始化过程到父类链的“传送”点。

每个类至少得有一个指定初始化器。如同在[自动初始化器的继承](#)里描述的那样，在某些情况下，这些需求通过从父类继承一个或多个指定初始化器来满足。

便捷初始化器是次要的，为一个类支持初始化器。你可以在相同的类里定义一个便捷初始化器来调用一个指定的初始化器作为便捷初始化器来给指定初始化器设置默认形式参数。你也可以为具体的使用情况或输入的值类型定义一个便捷初始化器从而创建这个类的实例。

如果你的类不需要便捷初始化器你可以不提供它。在为通用的初始化模式创建快捷方式以节省时间或者类的初始化更加清晰明了的时候时候便捷初始化器。

指定初始化器和便捷初始化器语法

用与值类型的简单初始化器相同的方式来写类的指定初始化器：

```
1 init(parameters){  
2 statements  
3 }
```

便捷初始化器有着相同的书写方式，但是要用 `convenience` 修饰符放到 `init` 关键字前，用空格隔开：

```
1 convenience init(parameters){  
2 statements  
3 }
```

类类型的初始化器委托

为了简化指定和便捷初始化器之间的调用关系，Swift 在初始化器之间的委托调用有下面的三个规则：

规则 1

指定初始化器必须从它的直系父类调用指定初始化器。

规则 2

便捷初始化器必须从相同的类里调用另一个初始化器。

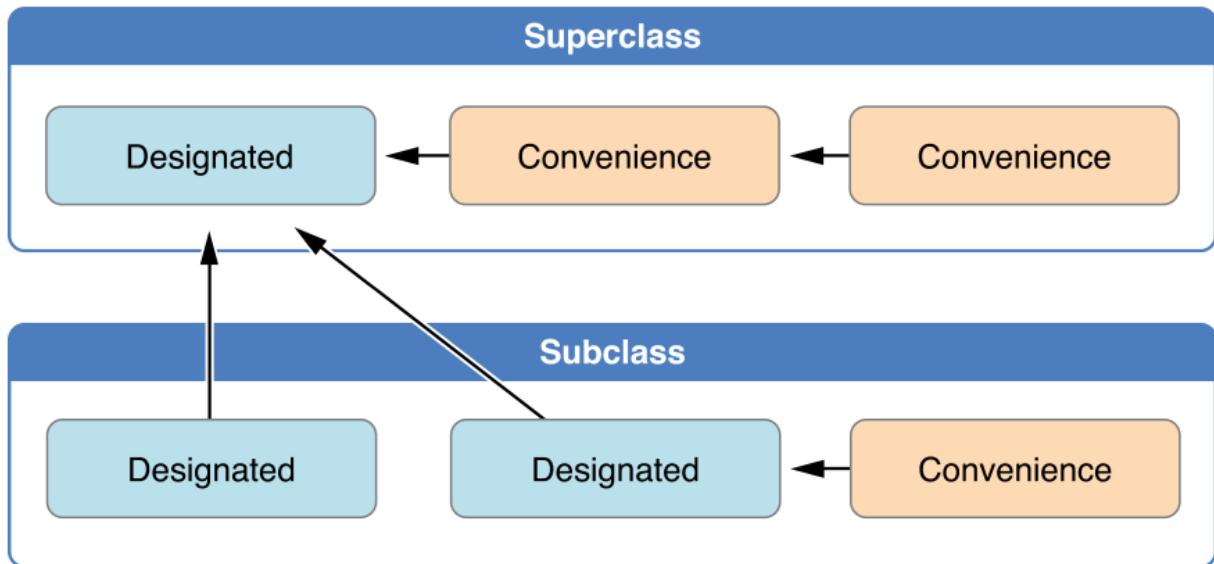
规则 3

便捷初始化器最终必须调用一个指定初始化器。

简单记忆的这些规则的方法如下：

- 指定初始化器必须总是向上委托。
- 便捷初始化器必须总是横向委托。

下面图中表示了这些规则：



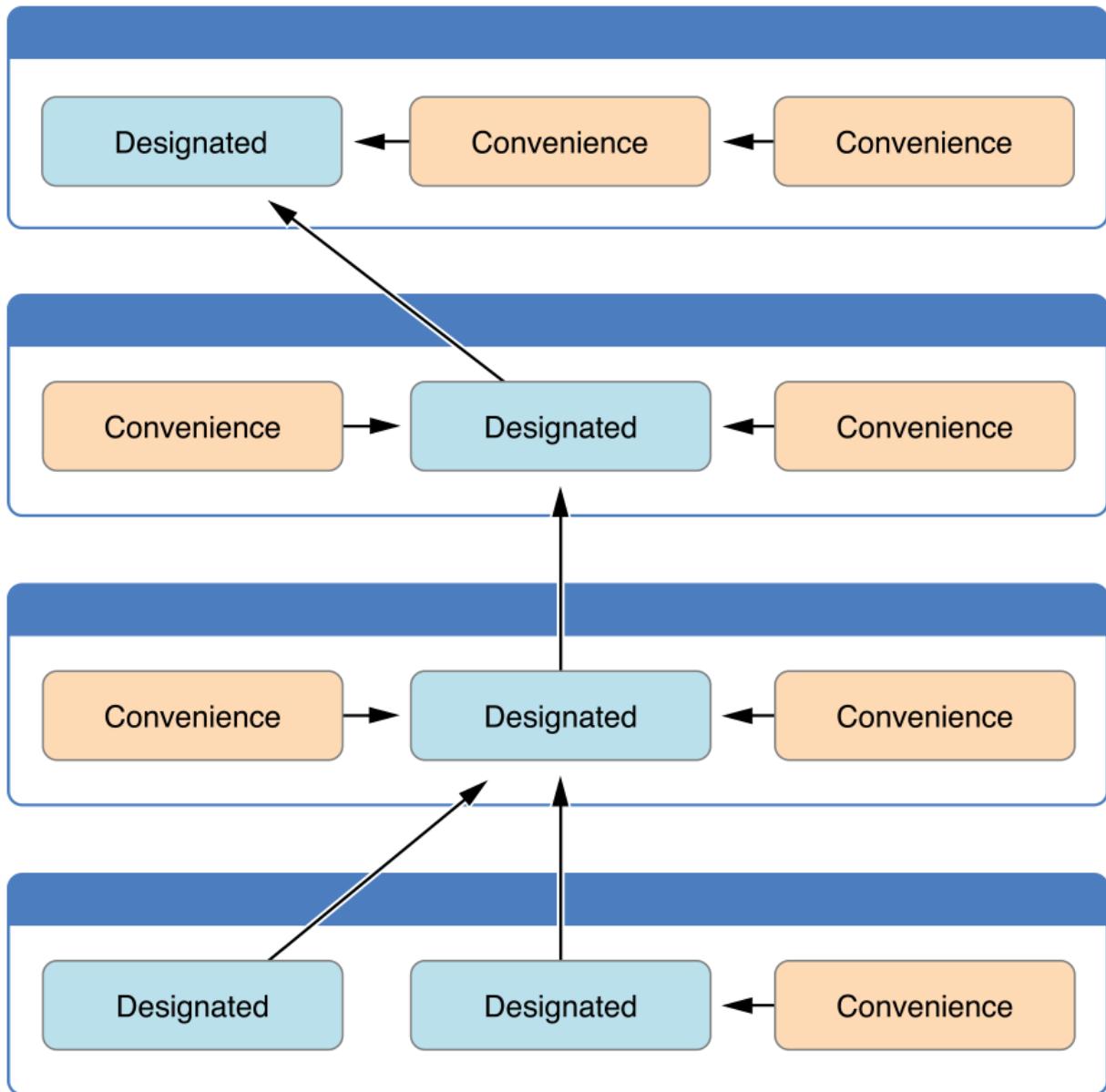
如图所示，父类包含一个指定初始化器和两个便捷初始化器。一个便捷初始化器调用另一个便捷初始化器，而后者又调用了指定初始化器。这满足了上边的规则2和规则3。父类本身没有其他父类，所以规则1不适用。

这个图中的子类有两个指定初始化器和一个便捷初始化器。便捷初始化器必须调用这两个指定初始化器之一，因为它只能从同一个类中调用初始化器。这满足了上边的规则2和规则3。两个指定初始化器又必须从父类调用一个指定初始化器，这满足了上边所说的规则1。

注意

这些规则不会影响你的类的使用者为每个类创建实例。任何上图的初始化器都可以用来完整创建对应类的实例。这个规则只在类的实现时有影响。

下图展示了四个类之间更复杂的层级结构。它演示了指定初始化器是如何在此层级结构中充当“管道”作用，在类的初始化链上简化了类之间的内部关系：



两段式初始化

Swift 的类初始化是一个两段式过程。在第一个阶段，每一个存储属性被引入类为分配了一个初始值。一旦每个存储属性的初始状态被确定，第二个阶段就开始了，每个类都有机会在新的实例准备使用之前来定制它的存储属性。

两段式初始化过程的使用让初始化更加安全，同时在每个类的层级结构给与了完备的灵活性。两段式初始化过程可以防止属性值在初始化之前被访问，还可以防止属性值被另一个初始化器意外地赋予不同的值。

注意

Swift 的两段式初始化过程与 Objective-C 的初始化类似。主要的不同点是在第一阶段，Objective-C 为每一个属性分配零或空值(例如 0 或 nil)。Swift 的初始化流程更加灵活，它允许你设置自定义的初始值，并可以自如应对 0 或 nil 不为合法值的情况。

Swift 编译器执行四种有效安全检查来确保两段式初始化过程能够顺利完成：

安全检查 1

指定初始化器必须保证在向上委托给父类初始化器之前，其所在类引入的所有属性都要初始

化完成。

如上所述，一个对象的内存只有在其所有储存型属性确定之后才能完全初始化。为了满足这一规则，指定初始化器必须保证它自己的属性在它上交委托之前先完成初始化。

安全检查 2

指定初始化器必须先向上委托父类初始化器，然后才能为继承的属性设置新值。如果不这样做，指定初始化器赋予的新值将被父类中的初始化器所覆盖。

安全检查 3

便捷初始化器必须先委托同类中的其它初始化器，然后再为任意属性赋新值（包括同类里定义的属性）。如果没这么做，便捷构初始化器赋予的新值将被自己类中其它指定初始化器所覆盖。

安全检查 4

初始化器在第一阶段初始化完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用 `self` 作为值。

直到第一阶段结束类实例才完全合法。属性只能被读取，方法也只能被调用，直到第一阶段结束的时候，这个类实例才被看做是合法的。

以下是两段初始化过程，基于上述四种检查的流程：

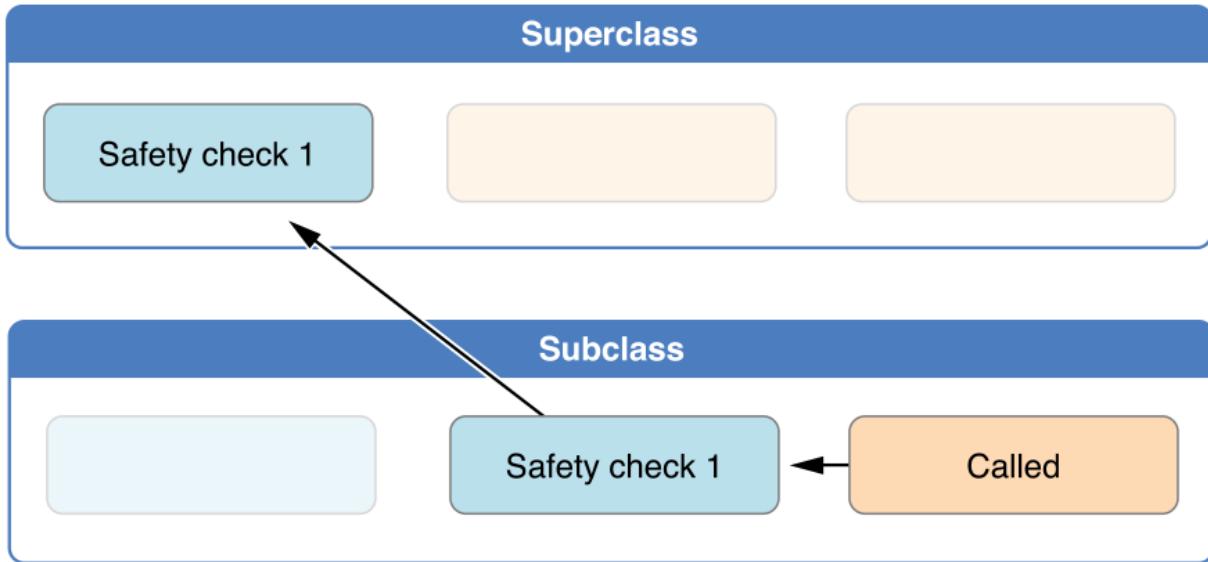
阶段 1

- 指定或便捷初始化器在类中被调用；
- 为这个类的新实例分配内存。内存还没有被初始化；
- 这个类的指定初始化器确保所有由此类引入的存储属性都有一个值。现在这些存储属性的内存被初始化了；
- 指定初始化器上交父类的初始化器为其存储属性执行相同任务；
- 这个调用父类初始化器的过程将沿着初始化器链一直向上进行，直到到达初始化器链的最顶部；
- 一旦达了初始化器链的最顶部，在链顶部的类确保所有的存储属性都有一个值，此实例的内存被认为完全初始化了，此时第一阶段完成。

阶段 2

- 从顶部初始化器往下，链中的每一个指定初始化器都有机会进一步定制实例。初始化器现在能够访问 `self` 并且可以修改它的属性，调用它的实例方法等等；
- 最终，链中任何便捷初始化器都有机会定制实例以及使用 `self`。

下面展示了第一阶段假定的子类和父类之间的初始化调用关系：



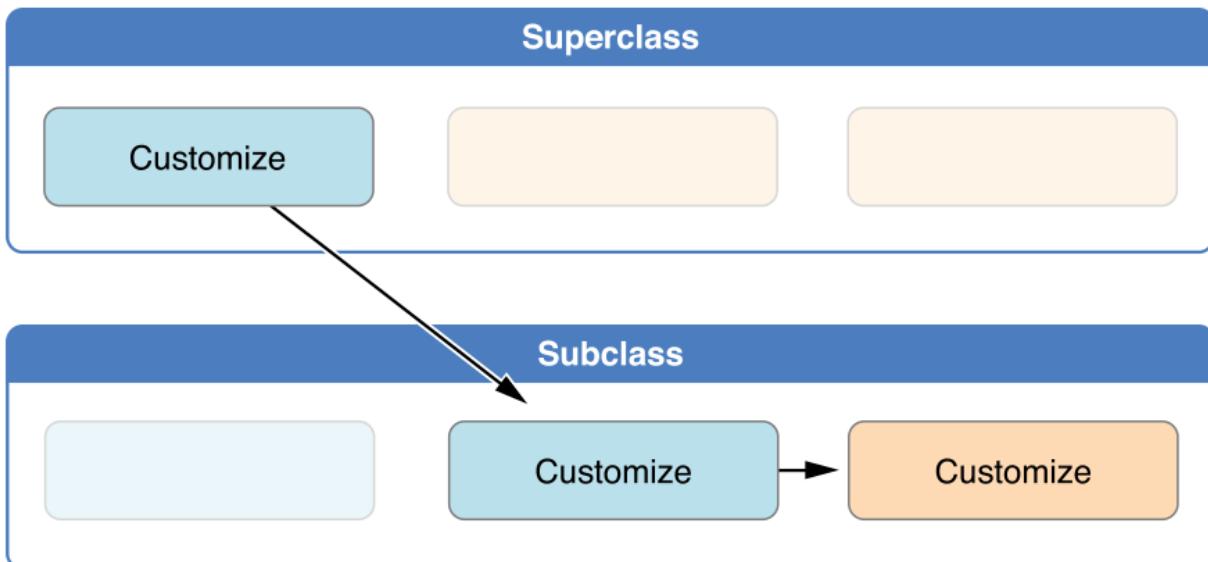
在这个栗子中，初始化过程从一个子类的便捷初始化器开始。这个便捷初始化器还不能修改任何属性。它委托给了同一类里的指定初始化器。

指定初始化器确保所有的子类属性都有值，如安全检查1。然后它调用父类的指定初始化器来沿着初始化器链一直往上完成父类的初始化过程。

父类的指定初始化器确保所有的父类属性都有值。由于没有更多的父类来初始化，也就不需要更多的委托。

一旦父类中所有属性都有初始值，它的内存就被认为完全初始化了，第阶段完成。

下图是相同的初始化过程在第二阶段的样子：



现在父类的指定初始化器有机会来定制更多实例(尽管没有这种必要)。

一旦父类的指定初始化器完成了调用，子类的指定初始化器就可以执行额外的定制(同样，尽管没有这种必要)。

最后，一旦子类的指定初始化器完成，最初调用的便捷初始化器将会执行额外的定制操作。

初始化器的继承和重写

不像在 Objective-C 中的子类，Swift 的子类不会默认继承父类的初始化器。Swift 的这种机制防止父类的简单初始化器被一个更专用的子类继承并被用来创建一个没有完全或错误初始化的新实例的情况发生。

注意

只有在特定情况下才会继承父类的初始化器，但只有这样是安全且合适的时候。想要了解更多信息，请看下边的[自动初始化器的继承](#)。

如果你想自定义子类来实现一个或多个和父类相同的初始化器，你可以在子类中为那些初始化器提供定制的实现。

当你写的子类初始化器匹配父类指定初始化器的时候，你实际上可以重写那个初始化器。因此，在子类的初始化器定义之前你必须写 `override` 修饰符。如同[默认初始化器](#)所描述的那样，即使是自动提供的默认初始化器你也可以重写。

作为一个重写的属性，方法或下标脚本，`override` 修饰符的出现会让 Swift 来检查父类是否有一个匹配的指定初始化器来重写，并且验证你重写的初始化器已经按照意图指定了形式参数。

注意

当重写父类指定初始化器时，你必须写 `override` 修饰符，就算你子类初始化器的实现是一个便捷初始化器。

相反，如同[上边类类型的初始化器委托](#)所描述的规则那样，如果你写了一个匹配父类便捷初始化器的子类初始化器，父类的便捷初始化器将永远不会通过你的子类直接调用。因此，你的子类不能(严格来讲)提供父类初始化器的重写。当提供一个匹配的父类便捷初始化器的实现时，你不用写 `override` 修饰符。

下面的栗子定义了一个名为 `Vehicle` 的基类。基类声明了一个名为 `numberOfWheels` 的存储属性，类型为 `Int` 的默认值 0。`numberOfWheels` 属性通过一个名为 `description` 的计算属性来创建一个 `String` 类型的车辆特征字符串描述：

```
1 class Vehicle{  
2     var numberOfWheels = 0  
3     var description: String {  
4         return "\((numberOfWheels)) wheel(s)"  
5     }  
6 }
```

`Vehicle` 类只为它的存储属性提供了默认值，并且没有提供自定义的初始化器。因此，如同[默认初始化器](#)中描述的那样，它会自动收到一个默认初始化器。默认初始化器(如果可用的话)总是类的指定初始化器，也可以用来创建一个新的 `Vehicle` 实例，`numberOfWheels` 默认为 0：

```
1 let vehicle = Vehicle()  
2 print("Vehicle: \(vehicle.description)")  
3 // Vehicle: 0 wheel(s)
```

下面的例子定义了一个名为 `Bicycle` 继承自 `Vehicle` 的子类：

```
1 class Bicycle: Vehicle{  
2     override init(){  
3         super.init()  
4         numberOfWheels=2  
5     }  
6 }
```

子类 `Bicycle` 定义了一个自定义初始化器 `init()`。这个指定初始化器和 `Bicycle` 的父类的指定初始化器相匹配，所以 `Bicycle` 中的指定初始化器需要带上 `override` 修饰符。

`Bicycle` 类的 `init()` 初始化器以调用 `super.init()` 开始，这个方法作用是调用父类的初始化器。这样可以确保 `Bicycle` 在修改属性之前它所继承的属性 `numberOfWheels` 能被 `Vehicle` 类初始化。在调用 `super.init()` 之后，一开始的 `numberOfWheels` 值会被新值 2 替换。

如果你创建一个 `Bicycle` 实例，你可以调用继承的 `description` 计算属性来查看属性 `numberOfWheels` 是否有改变。

```
1 let bicycle=Bicycle()  
2 print("Bicycle: \(bicycle.description)")  
3 // Bicycle: 2 wheel(s)
```

注意

子类可以在初始化时修改继承的变量属性，但是不能修改继承过来的常量属性。

自动初始化器的继承

如上所述，子类默认不会继承父类初始化器。总之，在特定的情况下父类初始化器是可以被自动继承的。实际上，这意味着在许多场景中你不必重写父类初始化器，只要可以安全操作，你就可以毫不费力地继承父类的初始化器。

假设你为你子类引入的任何新的属性都提供了默认值，请遵守以下2个规则：

规则1

如果你的子类没有定义任何指定初始化器，它会自动继承父类所有的指定初始化器。

规则2

如果你的子类提供了所有父类指定初始化器的实现——要么是通过规则1继承来的，要么通过在定义中提供自定义实现的——那么它自动继承所有的父类便捷初始化器。

就算你的子类添加了更多的便捷初始化器，这些规则仍然适用。

注意

子类能够以便捷初始化器的形式实现父类指定初始化器来作为满足规则2的一部分。

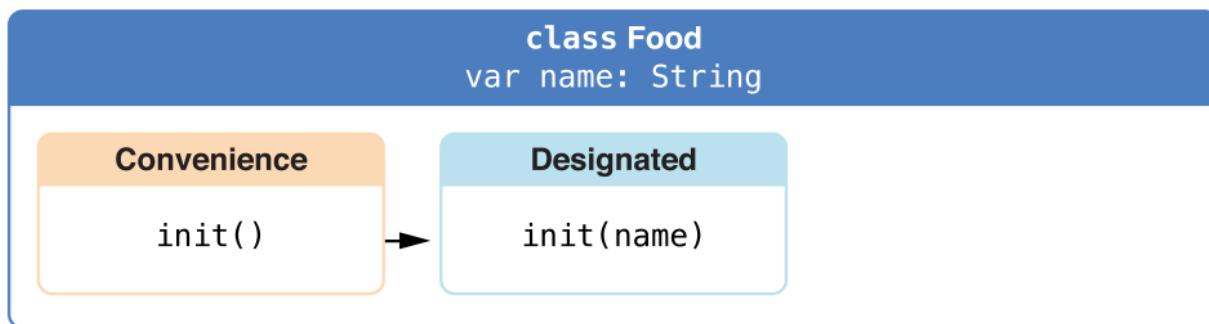
指定和便捷初始化器的操作

下面的栗子展示了在操作中指定初始化器，便捷初始化器和自动初始化器的继承。这个栗子定义了 `Food`，`RecipeIngredient` 和 `ShoppingListItem` 三个类的层级关系，并演示了它们的继承关系是如何相互作用的。

在层级关系中的基类称为 Food，它是一个简单的类用来封装了食品的名称。Food 类引入了一个名为 name 的 String 属性还提供了两个创建 Food 实例的初始化器：

```
1 class Food{  
2     var name: String  
3     init(name: String){  
4         self.name = name  
5     }  
6     convenience init(){  
7         self.init(name: "[Unnamed]")  
8     }  
9 }
```

下面的图表展示了 Food 类的初始化链：



类没有默认成员初始化器，所以 Food 类提供了一个接受单一实际参数的指定初始化器叫做 name。这个初始化器可以使用一个具体的名称来创建新的 Food 实例：

```
1 let namedMeat = Food(name: "Bacon")  
2 // namedMeat's name is "Bacon"
```

Food 类的 `init(name: String)` 初始化器是一个指定初始化器，因为它确保 Food 类新实例的所有存储属性都被完全初始化。Food 类没有父类，所以 `init(name: String)` 初始化器不用调用 `super.init()` 来完成它的初始化。

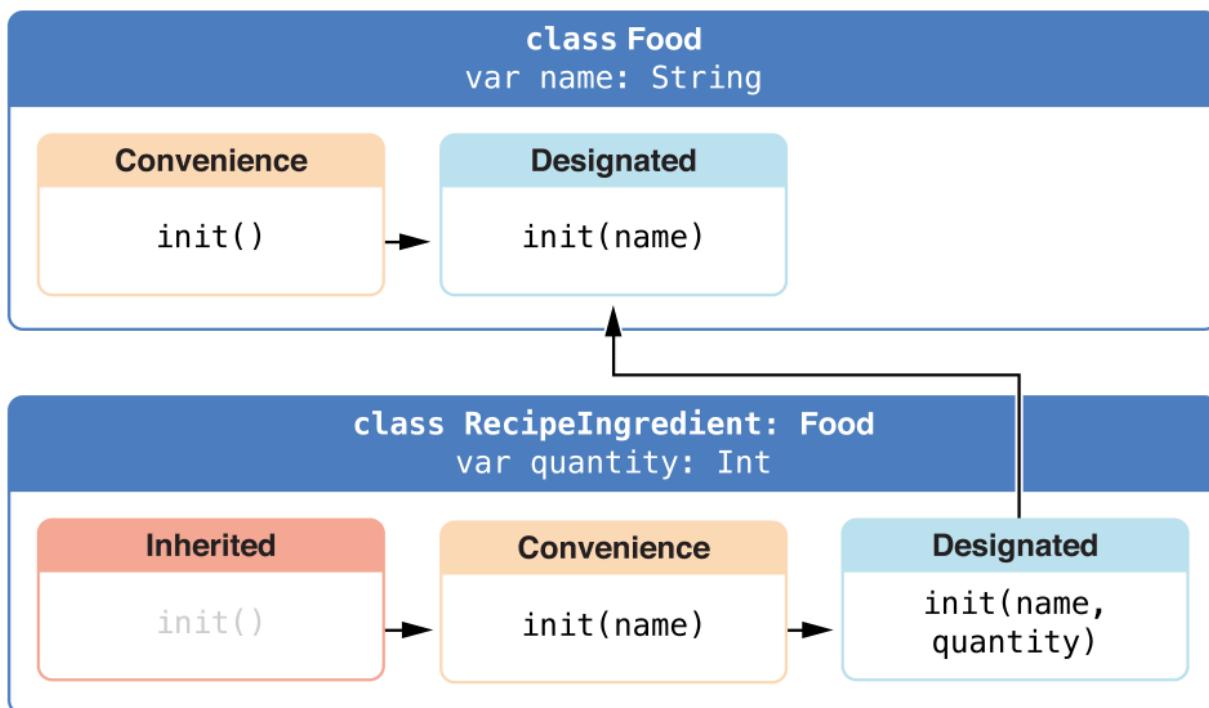
Food 类也提供了没有实际参数的便捷初始化器 `init()`。`init()` 初始化器通过委托调用同一类中定义的指定初始化器 `init(name: String)` 并给参数 `name` 传值 `[Unnamed]` 来实现提供默认的名称占位符：

```
1 let mysteryMeat = Food()  
2 // mysteryMeat's name is "[Unnamed]"
```

类层级中的第二个类是 Food 的子类 RecipeIngredient。RecipeIngredient 类模型构建了食谱中一个调味剂。它引入了 Int 类型的数量属性 `quantity` (以及从 Food 继承过来的 `name` 属性)并且定义了两个初始化器来创建 RecipeIngredient 实例：

```
1 class RecipeIngredient: Food{  
2     var quantity: Int  
3     init(name: String, quantity: Int){  
4         self.quantity = quantity  
5         super.init(name: name)  
6     }  
7     override convenience init(name: String){  
8         self.init(name: name, quantity: 1)  
9     }  
10 }
```

下面的图标展示了 RecipeIngredient 了的初始化链:



RecipeIngredient 类只有一个指定初始化器，`init(name:String,quantity:Int)`，它可以用来填充新的 RecipeIngredient 实例中所有的属性。这个初始化器一开始先将传入的 quantity 实际参数赋值给 quantity 属性，这个属性也是唯一一个通过 RecipeIngredient 引入的新属性。随后，初始化器将向上委托给父类 Food 的 `init(name:String)` 初始化器。这个过程满足上边所述的两段式初始化的安全检查1。

RecipeIngredient 同样定义了一个便捷初始化器，`init(name:String)`，它只通过name来创建 RecipeIngredient 的实例。这个便捷初始化器假设任意 RecipeIngredient 没有明确数量的实例的 quantity 值都为 1。便捷初始化器的定义让 RecipeIngredient 实例创建的方便更快捷，并且当创建多个单数实例时可以避免代码冗余。这个便捷初始化器只是简单的委托了类的指定初始化器，传递了值为 1 的 quantity。

RecipeIngredient 类提供的 `init(name:String)` 便捷初始化器接收与 Food 中指定初始化器 `init(name:String)` 相同的形式参数。因为这个便捷初始化器从它的父类重写了一个指定初始化器，它必须使用 `override` 修饰符(如同在初始化器的继承和重写中描述的那样)。

尽管 RecipeIngredient 提供了 `init(name:String)` 初始化器作为一个便捷初始化器，然而 RecipeIngredient 类为所有的父类指定初始化器提供了实现。因此， RecipeIngredient 类也自动继承了父类所有的便捷初始化器。

在这个栗子中， RecipeIngredient 的父类是 Food ，它只有一个 `init()` 便捷初始化器。因此这个初始化器也被 RecipeIngredient 类继承。这个继承的 `init()` 函数和 Food 提供的是一样的，除了它是委托给 RecipeIngredient 版本的 `init(name:String)` 而不是 Food 版本。

所有的这三种初始化器都可以用来创建新的 RecipeIngredient 实例：

- 1 letoneMysteryItem=RecipeIngredient()
- 2 letoneBacon=RecipeIngredient(name:"Bacon")
- 3 letsixEggs=RecipeIngredient(name:"Eggs",quantity:6)

类层级中第三个也是最后一个类是 RecipeIngredient 的子类，叫做 ShoppingListItem。这个类构建了购物单中出现的某一种调味料。

在购物表里每一项都是从未购买状态开始的。为了展现这一事实，ShoppingListItem 引入了一个布尔类型的属性 purchased，默认值 false。ShoppingListItem 也添加了一个计算属性 description 属性，它提供了关于 ShoppingListItem 实例的一些文字描述：

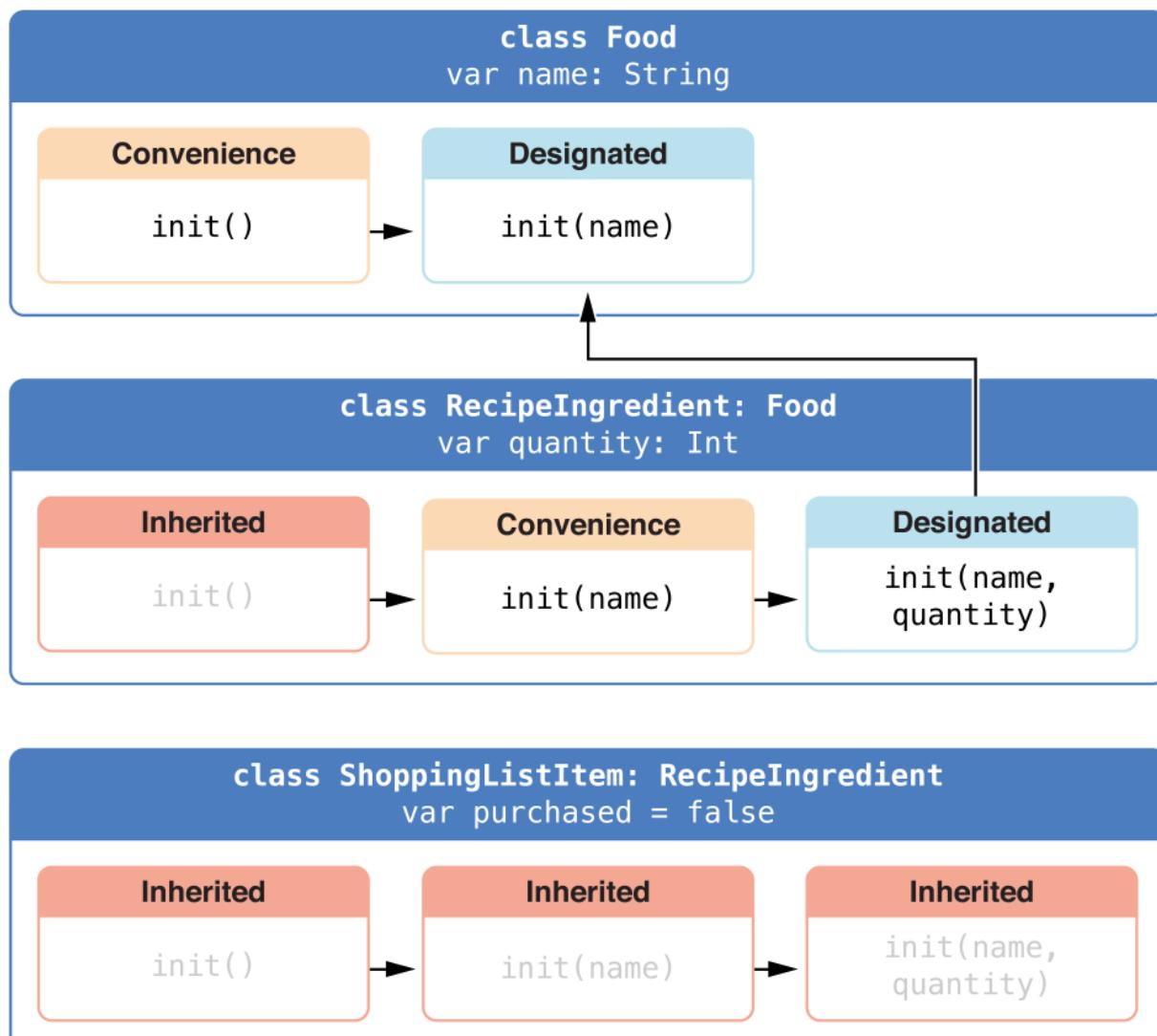
```
1 class ShoppingListItem: RecipeIngredient{
2     var purchased = false
3     var description: String{
4         var output = "(quantity) x (name)"
5         output += purchased ? " ✓" : " ✗"
6         return output
7     }
8 }
```

注意

ShoppingListItem 没有定义初始化器来给 purchased 一个初始值，这是因为任何添加到购物单（这里的模型）的项的初始状态总是未购买。

由于它为自己引入的所有属性提供了一个默认值并且自己没有定义任何初始化器，ShoppingListItem 会自动从父类继承所有的指定和便捷初始化器。

下图展示了三个类的初始化链：



你可以使用全部三个继承来的初始化器来创建 `ShoppingListItem` 的新实例：

```
1 varbreakfastList=[  
2 ShoppingListItem(),  
3 ShoppingListItem(name:"Bacon"),  
4 ShoppingListItem(name:"Eggs",quantity:6),  
5 ]  
6 breakfastList[0].name="Orange juice"  
7 breakfastList[0].purchased=true  
8 foriteminbreakfastList{  
9     print(item.description)  
10 }  
11 // 1 x Orange juice ✓  
12 // 1 x Bacon ✗  
13 // 6 x Eggs ✗
```

这里，名为 `breakfastList` 的数组通过包含三个 `ShoppingListItem` 实例的数组字面量创建。数组的类型被推断为 `[ShoppingListItem]`。数组创建之后，数组第一个 `ShoppingListItem` 的 `name` 从 `"[Unnamed]"` 修改为 `"Orange juice"`，并且它的 `purchased` 也标记为了 `true`。然后通过遍历打印每个数组的描述，展示了它们的默认状态都按照预期被设置了。

可失败初始化器

定义类、结构体或枚举初始化时可以失败在某些情况下会管大用。这个失败可能由以下几种方式触发，包括给初始化传入无效的形式参数值，或缺少某种外部所需的资源，又或是其他阻止初始化的情况。

为了妥善处理这种可能失败的情况，在类、结构体或枚举中定义一个或多个 **可失败的初始化器**。通过在 `init` 关键字后面添加问号(`init?`)来写。

注意

你不能定义可失败和非可失败的初始化器为相同的形式参数类型和名称。

可失败的初始化器创建了一个初始化类型的**可选值**。你通过在可失败初始化器写 `returnnil` 语句，来表明可失败初始化器在何种情况下会触发初始化失败。

注意

严格来讲，初始化器不会有返回值。相反，它们的角色是确保在初始化结束时，`self` 能够被正确初始化。虽然你写了 `returnnil` 来触发初始化失败，但是你不能使用 `return` 关键字来表示初始化成功了。

比如说，可失败初始化器为数字类型转换器做实现。为了确保数字类型之间的转换保持值不变，使用 `init(exactly:)` 初始化器。如果类型转换不能保持值不变，初始化器失败。

```

1 letwholeNumber:Double=12345.0
2 letpi=3.14159
3 ifletvalueMaintained=Int(exactly:wholeNumber){
4     print("\(wholeNumber) conversion to int maintains value" )
5 }
6 // Prints "12345.0 conversion to int maintains value"
7 letvalueChanged=Int(exactly:pi)
8 // valueChanged is of type Int?, not Int
9 ifvalueChanged==nil{
10     print("\(pi) conversion to int does not maintain value" )
11 }
12 // Prints "3.14159 conversion to int does not maintain value"
13
14
15

```

下面的栗子定义了一个名为 Animal 的结构体，有一个名为 species 的 String 类型常量属性。Animal 结构体也定义了一个可失败初始化器，有一个形式参数 species 。这个初始化器用来检查传入 species 的字符串是否为空。如果发现了一个空字符串，初始化失败被触发。否则，就设置 species 属性的值，然后初始化成功：

```

1 structAnimal{
2     letspecies:String
3     init?(species:String){
4         ifspecies.isEmpty{returnnil}
5         self.species=species
6     }
7 }

```

你可以通过可失败初始化器来尝试初始化一个新的 Animal 实例并且检查初始化是否成功：

```

1 letsomeCreature=Animal(species:"Giraffe")
2 // someCreature is of type Animal?, not Animal
3 ifletgiraffe=someCreature{
4     print("An animal was initialized with a species of \(giraffe.species)")
5 }
6 // prints "An animal was initialized with a species of Giraffe"
7

```

如果你给可失败初始化器的 species 形式参数传了一个字符串值，初始化器触发初始化失败：

```

1 letanonymousCreature=Animal(species:"")
2 // anonymousCreature is of type Animal?, not Animal
3 ifanonymousCreature==nil{
4     print("The anonymous creature could not be initialized")
5 }
6 // prints "The anonymous creature could not be initialized"
7

```

注意

检查空字符串值(比如是 "" 而不是 "Giraffe")和检查值是否为 nil 来表明可选项String是不是没有值是两个不一样的的概念。上面的栗子中，空字符串("")是合法的，非可选的 String 。总之，对于 Animal 来说让它的species属性有一个空的字符串作为值是不合适的。要模式化这个限制，可失败的初始化器就会在发现空字符串时触发初始化失败。

枚举的可失败初始化器

你可以使用一个可失败初始化器来在带一个或多个形式参数的枚举中选择合适的情况。如果提供的形式参数没有匹配合适的情况初始化器就可能失败。

下面的栗子定义一个名为 TemperatureUnit 的枚举，有三种可能的状态(Kelvin ， Celsius 和 Fahrenheit)。使用一个可失败初始化器来找一个合适用来表示气温符号的 Character 值：

```
1 enumTemperatureUnit{  
2     caseKelvin,Celsius,Fahrenheit  
3     init?(symbol:Character){  
4         switchsymbol{  
5             case "K":  
6                 self=.Kelvin  
7             case "C":  
8                 self=.Celsius  
9             case "F":  
10                self=.Fahrenheit  
11         default:  
12             returnnil  
13     }  
14 }  
15 }
```

你可以使用可失败初始化器为可能的三种状态来选择一个合适的枚举情况，当参数的值不能与任意一枚举成员相匹配时，该枚举类型初始化失败：

```
1 letfahrenheitUnit=TemperatureUnit(symbol:"F")  
2 iffahrenheitUnit!=nil{  
3     print("This is a defined temperature unit, so initialization succeeded." )  
4 }  
5 // prints "This is a defined temperature unit, so initialization succeeded."  
6 letunknownUnit=TemperatureUnit(symbol:"X")  
7 ifunknownUnit==nil{  
8     print("This is not a defined temperature unit, so initialization failed." )  
9 }  
10 // prints "This is not a defined temperature unit, so initialization failed."  
11
```

带有原始值枚举的可失败初始化器

带有原始值的枚举会自动获得一个可失败初始化器 `init?(rawValue:)`，该可失败初始化器接收一个名为 `rawValue` 的合适的原始值类型形式参数如果找到了匹配的枚举情况就选择其一，或者没有找到匹配的值就触发初始化失败。

你可以把上面的 TemperatureUnit 的栗子可以重写为使用 `Character` 原始值并带有改过的 `init?(rawValue:)` 初始化器：

```
1 enumTemperatureUnit: Character{
2     caseKelvin="K", Celsius="C", Fahrenheit="F"
3 }
4 letfahrenheitUnit=TemperatureUnit(rawValue:"F")
5 iffahrenheitUnit!=nil{
6     print("This is a defined temperature unit, so initialization succeeded." )
7 }
8 // prints "This is a defined temperature unit, so initialization succeeded."
9 letunknownUnit=TemperatureUnit(rawValue:"X")
10 ifunknownUnit==nil{
11     print("This is not a defined temperature unit, so initialization failed." )
12 }
13 // prints "This is not a defined temperature unit, so initialization failed."
14
15
```

初始化失败的传递

类，结构体或枚举的可失败初始化器可以横向委托到同一个类，结构体或枚举里的另一个可失败初始化器。类似地，子类的可失败初始化器可以向上委托到父类的可失败初始化器。

无论哪种情况，如果你委托到另一个初始化器导致了初始化失败，那么整个初始化过程也会立即失败，并且之后任何初始化代码都不会执行。

注意

可失败初始化器也可以委托其他的非可失败初始化器。通过这个方法，你可以为已有的初始化过程添加初始化失败的条件。

下面的栗子定义了 Product 类的子类 CartItem 。 CartItem 类建立了一个在线购物车中物品的模型。 CartItem 引入了一个名为 quantity 常量存储属性，并且确保了这个属性至少是 1 :

```
1 classProduct{
2     letname:String
3     init?(name:String){
4         ifname.isEmpty{returnnil}
5         self.name=name
6     }
7 }
8 classCartItem: Product{
9     letquantity:Int
10    init?(name:String,quantity:Int){
11        ifquantity<1{returnnil}
12        self.quantity=quantity
13        super.init(name:name)
14    }
15 }
16
```

CartItem 的可失败初始化器以检测它是否接受了一个 quantity 值为 1 或者更多开始。如果 quantity 不合法，整个初始化过程会立即失败并且后来的初始化代码都不会执行。同样地， Product 的可失败初始化器检查 name 的值，初始化器进程会在 name 为空字符串时直接失败。

如果你用不能为空 name 属性和数量为 1 或者更多来创建 CartItem 实例，则初始化成功：

```
1 iflettwoSocks=CartItem(name:"sock",quantity:2){  
2   print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")  
3 }  
4 // prints "Item: sock, quantity: 2"
```

如果你创建了一个 `CartItem` 实例，`quantity` 的值为 0，`CartItem` 初始化器会导致初始化失败：

```
1 ifletzeroShirts=CartItem(name:"shirt",quantity:0){  
2   print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")  
3 }else{  
4   print("Unable to initialize zero shirts")  
5 }  
6 // prints "Unable to initialize zero shirts"
```

类似地，如果你尝试创建一个 `CartItem` 实例，并且令 `name` 为空值，那么父类 `Product` 的初始化器就会导致初始化失败：

```
1 ifletoneUnnamed=CartItem(name:"",quantity:1){  
2   print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")  
3 }else{  
4   print("Unable to initialize one unnamed product")  
5 }  
6 // prints "Unable to initialize one unnamed product"
```

重写可失败初始化器

你可以在子类里重写父类的可失败初始化器。就好比其他的初始化器。或者，你可以用子类的非可失败初始化器来重写父类可失败初始化器。这样允许你定义一个初始化不会失败的子类，尽管父类的初始化允许失败。

注意如果你用非可失败的子类初始化器重写了一个可失败初始化器，向上委托到父类初始化器的唯一办法是强制展开父类可失败初始化器的结果。

注意

你可以用一个非可失败初始化器重写一个可失败初始化器，但反过来是不行的。

下面的栗子定义了一个名为 `Document` 的类。这个类建模了一个文档，其中的 `name` 属性要么是一个非空的字符串值要么为 `nil`，但不能是一个空字符串：

```
1 classDocument{  
2   varname:String?  
3   // this initializer creates a document with a nil name value  
4   init(){  
5     // this initializer creates a document with a non-empty name value  
6     init?(name:String){  
7       self.name=name  
8       ifname.isEmpty{returnnil}  
9     }  
10 }
```

下面这个栗子定义了一个名为 `AutomaticallyNamedDocument` 的 `Document` 类的子类。这个子类重写了 `Document` 类引入的两个指定初始化器。这些重写确保了 `AutomaticallyNamedDocument` 实例在初始化时没有名字或者传给 `init(name:)` 初始化器一个空字符串时 `name` 的初始值为 "[Untitled]"：

```
1 class AutomaticallyNamedDocument: Document{
2     override init(){
3         super.init()
4         self.name = "[Untitled]"
5     }
6     override init(name: String){
7         super.init()
8         if name.isEmpty{
9             self.name = "[Untitled]"
10        }else{
11            self.name = name
12        }
13    }
14 }
```

AutomaticallyNamedDocument 类用非可失败的 init(name:) 初始化器重写了父类的可失败 init?(name:) 初始化器。因为 AutomaticallyNamedDocument 类用不同的方式处理了空字符串的情况，它的初始化器不会失败，所以它提供了非可失败初始化器来代替。

你可以在初始化器里使用强制展开来从父类调用一个可失败初始化器作为子类非可失败初始化器的一部分。例如，下边的 UntitledDocument 子类将总是被命名为 "[Untitled]"，并且在初始化期间它使用了父类的可失败 init(name:) 初始化器：

```
1 class UntitledDocument: Document{
2     override init(){
3         super.init(name: "[Untitled]!")
4     }
5 }
```

这种情况下，如果父类的 init(name:) 初始化器曾以空字符串做名字调用，强制展开操作会导致运行时错误。总之，由于它调用了一个字符串常量，那么你可以看到初始化器不会失败，所以这时不会有运行时错误发生。

可失败初始化器 init!

通常来讲我们通过在 init 关键字后添加问号 (init?) 的方式来定义一个可失败初始化器以创建一个合适类型的可选项实例。另外，你也可以使用可失败初始化器创建一个隐式展开具有合适类型的可选项实例。通过在 init 后面添加惊叹号(init!)是不是问号。

你可以在 init? 初始化器中委托调用 init! 初始化器，反之亦然。你也可以用 init! 重写 init?，反之亦然。你还可以用 init 委托调用 init!，尽管当 init! 初始化器导致初始化失败时会触发断言。

必要初始化器

在类的初始化器前添加 required 修饰符来表明所有该类的子类都必须实现该初始化器：

```
1 class SomeClass{
2     required init(){
3         // initializer implementation goes here
4     }
5 }
```

当子类重写父类的必要初始化器时，必须在子类的初始化器前同样添加 `required` 修饰符以确保当其它类继承该子类时，该初始化器同为必要初始化器。在重写父类的必要初始化器时，不需要添加 `override` 修饰符：

```
1 class SomeSubclass: SomeClass{  
2     required init(){  
3         // subclass implementation of the required initializer goes here  
4     }  
5 }
```

注意

如果子类继承的初始化器能够满足需求，则你无需显式地在子类中提供必要初始化器的实现。

通过闭包和函数来设置属性的默认值

如果某个存储属性的默认值需要自定义或设置，你可以使用闭包或全局函数来为属性提供默认值。当这个属性属于的实例初始化时，闭包或函数就会被调用，并且它的返回值就会作为属性的默认值。

这种闭包或函数通常会创建一个和属性相同的临时值，处理这个值以表示初始的状态，并且把这个临时值返回作为属性的默认值。

下面的代码框架展示了闭包是如何提供默认值给属性的：

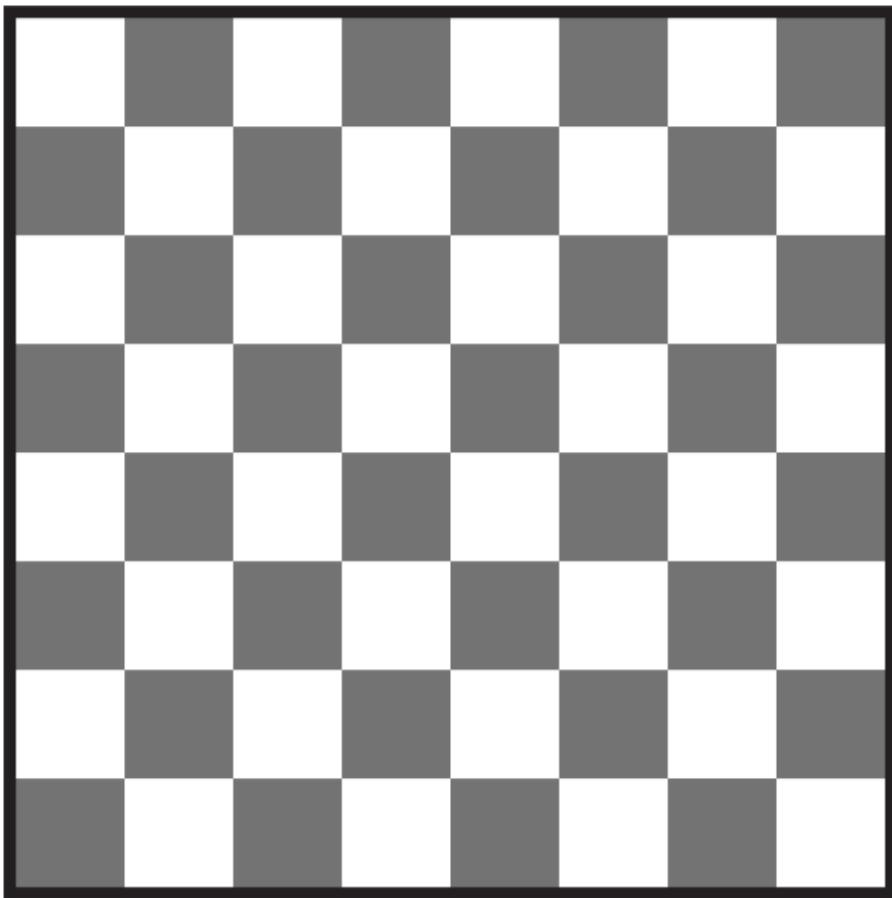
```
1 class SomeClass{  
2     let someProperty: SomeType = {  
3         // create a default value for someProperty inside this closure  
4         // someValue must be of the same type as SomeType  
5         return someValue  
6     }()  
7 }
```

注意闭包花括号的结尾跟一个没有参数的圆括号。这是告诉 Swift 立即执行闭包。如果你忽略了这对圆括号，你就会把闭包作为值赋给了属性，并且不会返回闭包的值。

注意

如果你使用了闭包来初始化属性，请记住闭包执行的时候，实例的其他部分还没有被初始化。这就意味着你不能在闭包里读取任何其他的属性值，即使这些属性有默认值。你也不能使用隐式 `self` 属性，或者调用实例的方法。

下面的栗子定义了一个名为 `Checkerboard` 结构体，建模了一个国际象棋的棋盘。国际象棋在一个8×8的棋盘上进行，这里我们用黑白色块来代替。



为了呈现游戏的棋盘， Checkerboard 结构体只有一个名为 boardColors 的属性，它是一个包含100个 Bool 值的数组。数组中的 true 代表黑色的格子， false 代表白色的格子。数组中第一项代表棋盘的左上角方格，数组最后一项代表棋盘的右下角方格。

boardColors 数组在一个闭包里初始化，来设置它的颜色值：

```
1 structChessboard{  
2     letboardColors:[Bool]={  
3         vartemporaryBoard=[Bool]()  
4         varisBlack=false  
5         foriin1...8{  
6             forjin1...8{  
7                 temporaryBoard.append(isBlack)  
8                 isBlack=!isBlack  
9             }  
10            isBlack=!isBlack  
11        }  
12        returntemporaryBoard  
13    }()  
14    funcsquareIsBlackAt(row:Int,column:Int)->Bool{  
15        returnboardColors[(row*8)+column]  
16    }  
17 }
```

无论何时，创建一个新的 Checkboard ，闭包就会执行，并且 boardColors 的默认值就会计算和返回。上面栗子中的闭包在一个名为 temporaryBoard 的临时数组中为每个方格计算并且设置合适的颜色，然后一旦设置完毕，就把这个临时数组作为闭包的返回值返回。返回的数组值储存在 boardColors 中，并且可以通过 squareIsBlackAtRow 工具函数来查询：

```
1 letboard=Chessboard()
2 print(board.squareIsBlackAt(row:0,column:1))
3 // Prints "true"
4 print(board.squareIsBlackAt(row:7,column:7))
5 // Prints "false"
```

反初始化

 cnswift.org/deinitialization

在类实例被释放的时候，反初始化器就会立即被调用。你可以是用 `deinit` 关键字来写反初始化器，就如同写初始化器要用 `init` 关键字一样。反初始化器只在类类型中有效。

反初始化器原理

当实例不再被需要的时候 Swift 会自动将其释放掉，以节省资源。如同自动引用计数中描述的那样，Swift 通过自动引用计数 (ARC) 来处理实例的内存管理。基本上，当你的实例被释放时，你不需要手动清除它们。总之，当你在操作自己的资源时，你可能还是需要在释放实例时执行一些额外的清理工作。比如说，如果你创建了一个自定义类来打开某文件写点数据进去，你就得在实例释放之前关闭这个文件。

每个类当中只能有一个反初始化器。反初始化器不接收任何形式参数，并且不需要写圆括号：

```
1 deinit{  
2 // perform the deinitialization  
3 }
```

反初始化器会在实例被释放之前自动被调用。你不能自行调用反初始化器。父类的反初始化器可以被子类继承，并且子类的反初始化器实现结束之后父类的反初始化器会被调用。父类的反初始化器总会被调用，就算子类没有反初始化器。

由于实例在反初始化器被调用之前都不会被释放，反初始化器可以访问实例中的所有属性并且可以基于这些属性修改自身行为（比如说查找需要被关闭的那个文件的文件名）。

应用反初始化器

这里有一个应用反初始化器的栗子。这里栗子给一个简单的游戏定义了两个新的类型，`Bank` 和 `Player`。`Bank` 类用来管理虚拟货币，它在流通过程中永远都不能拥有超过 10000 金币。游戏当中只能有一个 `Bank`，所以 `Bank` 以具有类型属性和方法的类来实现当前状态的储存和管理：

```
1 class Bank{  
2     static var coinsInBank = 10_000  
3     static func distribute(coins: Int) -> Int {  
4         let number_of_coins_to_vend = min(coins, coinsInBank)  
5         coinsInBank -= number_of_coins_to_vend  
6         return number_of_coins_to_vend  
7     }  
8     static func receive(coins: Int) {  
9         coinsInBank += coins  
10    }  
11 }
```

`Bank` 会一直用 `CoinsInBank` 属性来追踪当前金币数量。它同样也提供了两个方法——

distribute(coins:)和 receive(coins:)——来处理金币的收集和分发。

distribute(coins:)在分发金币之前检查银行当中是否有足够的金币。如果金币不足，Bank返回一个比需要的数小一些的数值（并且零如果银行里没有金币的话）。distribute(coins:)声明了一个numberOfCoinsToVend的变量形式参数，所以数值可以在方法体内修改而不需要再声明一个新的变量。它返回一个整数值来明确提供的金币的实际数量。

receive(coins:)方法只是添加了接受的金币数量到银行的金币储存里去。

Player类描述了游戏中的一个玩家。每个玩家都有确定数量的金币储存在它们的钱包中。这个以玩家的coinsInPurse属性表示：

```
1 classPlayer{  
2     varcoinsInPurse:Int  
3     init(coins:Int){  
4         coinsInPurse=Bank.distribute(coins:coins)  
5     }  
6     funcwin(coins:Int){  
7         coinsInPurse+=Bank.distribute(coins:coins)  
8     }  
9     deinit{  
10        Bank.receive(coins:coinsInPurse)  
11    }  
12 }
```

每一个Player实例都会用银行指定的金币数量来作为一开始的限定来初始化，尽管Player实例可能会在没有足够多金币的时候收到更少的数量。

Player类定义了一个win(coins:)方法，它从银行取回确定数量的金币并且把它们添加到玩家的钱包当中。Player类同样实现了一个反初始化器，它会在Player实例释放之前被调用。这时，反初始化器会把玩家多有的金币返回到银行当中：

```
1 varplayerOne:Player?=Player(coins:100)  
2 print("A new player has joined the game with \$(playerOne!.coinsInPurse) coins")  
3 // Prints "A new player has joined the game with 100 coins"  
4 print("There are now \$(Bank.coinsInBank) coins left in the bank")  
5 // Prints "There are now 9900 coins left in the bank"
```

新的Player实例创建出来了，同时如果可以的话会获取100个金币。这个Player实例储存了一个可选的Player变量叫做playerOne。这里使用了一个可选变量，是因为玩家可以在任何时候离开游戏。可选项允许你追踪当前游戏中是否有玩家。

因为playerOne是可选项，当它的coinsInPurse属性被访问来打印默认金币时，必须使用感叹号(!)才能完全符合，并且无论win(coins:)方法是否被调用：

```
1 playerOne!.win(coins:2_000)  
2 print("PlayerOne won 2000 coins & now has \$(playerOne!.coinsInPurse) coins")  
3 // Prints "PlayerOne won 2000 coins & now has 2100 coins"  
4 print("The bank now only has \$(Bank.coinsInBank) coins left")  
5 // Prints "The bank now only has 7900 coins left"
```

这时，玩家拥有了2000个金币。玩家的钱包当中保存了2100个金币，并且银行只剩下7900个金币。

```
1 playerOne=nil
2 print("PlayerOne has left the game")
3 // prints "PlayerOne has left the game"
4 print("The bank now has \$(Bank.coinsInBank) coins")
5 // prints "The bank now has 10000 coins"
6
```

现在玩家离开了游戏。这通过设置 playerOne变量为 nil来明确，意味着“无 Player实例。”当这个时候， playerOne变量到 Player实例的引用被破坏掉了。没有其他的属性或者变量仍在引用 Player实例，所以它将会被释放掉以节约内存。在释放掉的瞬间，它的反初始化器会自动被调用，然后它的金币被送回给了银行。

自动引用计数

 cnswift.org/automatic-reference-counting

Swift 使用自动引用计数(ARC)机制来追踪和管理你的 App 的内存。在大多数情况下，这意味着 Swift 的内存管理机制会一直起作用，你不需要自己考虑内存管理。当这些实例不在需要时，ARC会自动释放类实例所占用的内存。

总之，在少数情况下，为了能帮助你管理内存，ARC需要更多关于你代码之间的关系信息。本章描述了这些情况并向你展示如何启用ARC来管理你 App 的内存。在 Swift 中使用 ARC 与 [Transitioning to ARC Release Notes](#) 中描述的在 Objective-C 中使用 ARC 十分相似。

注意

引用计数只应用于类的实例。结构体和枚举是值类型，不是引用类型，没有通过引用存储和传递。

ARC的工作机制

每次你创建一个类的实例，ARC 会分配一大块内存来存储这个实例的信息。这些内存中保留有实例的类型信息，以及该实例所有存储属性值的信息。

此外，当实例不需要时，ARC 会释放该实例所占用的内存，释放的内存用于其他用途。这确保类实例当它不在需要时，不会一直占用内存。

然而，如果 ARC 释放了正在使用的实例内存，那么它将不会访问实例的属性，或者调用实例的方法。确实，如果你试图访问该实例，你的app很可能会崩溃。

为了确保使用中的实例不会消失，ARC 会跟踪和计算当前实例被多少属性，常量和变量所引用。只要存在对该类实例的引用，ARC 将不会释放该实例。

为了使这些成为可能，无论你将实例分配给属性，常量或变量，它们都会创建该实例的强引用。之所以称之为“强”引用，是因为它会将实例保持住，只要强引用还在，实例是不允许被销毁的。

ARC

下面的例子展示了自动引用计数的工作机制。这个例子由一个简单的 Person 类开始，定义了一个名为 name 的存储常量属性：

```
1 classPerson{  
2     letname:String  
3     init(name:String){  
4         self.name=name  
5         print("\(name) is being initialized")  
6     }  
7     deinit{  
8         print("\(name) is being deinitialized")  
9     }  
10 }
```

Person 类有一个初始化器，它设置了实例的 name 属性并且输出一条信息表明初始化器生效。 Person 类也有一个反初始化器，会在类的实例被销毁的时候打印一条信息。

下面的代码片段定义了三个 Person? 类型的变量，用来按照代码中的顺序，为新的 Person 实例设置多个引用。由于可选类型的变量会被自动初始化为一个 nil 值，目前还不会引用到 Person 类的实例。

```
1 varreference1:Person?  
2 varreference2:Person?  
3 varreference3:Person?
```

你可以创建一个新的 Person 实例并且将它赋值给三个变量中的一个：

```
1 reference1=Person(name:"John Appleseed")  
2 // prints "John Appleseed is being initialized"
```

注意，当调用 person 类的出初始化器的时候，会输出 "John Appleseed is being initialized" 信息。这就说明初始化执行了。

因为 Person 实例已经赋值给了 reference1 变量，现在就有了一个从 reference1 到该实例的强引用。因为至少有一个强引用，ARC 可以确保 Person 一直保持在内存中不被销毁。

如果你将同一个 Person 实例分配给了两个变量，则该实例又会多出两个强引用：

```
1 reference2=reference1  
2 reference3=reference1
```

现在这一个 Person 实例就有了三个强引用。

如果你通过给其中两个变量赋值 nil 的方式断开两个强引用（包括最先的那个强引用），只留下一个强引用，Person 实例不会被销毁：

```
1 reference1=nil  
2 reference2=nil
```

在你清楚地表明不再使用这个 Person 实例时，直到第三个也就是最后一个强引用被断开时 ARC 会销毁它。

```
1 reference3=nil  
2 // prints "John Appleseed is being deinitialized"
```

类实例之间的循环强引用

在上面的例子中，ARC 能够追踪你所创建的 Person 实例的引用数量，并且会在 Person 实例不在使用时销毁。

总之，写出某个类永远不会变成零强引用的代码是可能的。如果两个类实例彼此持有对方的强引用，因而每个实例都让对方一直存在，就会发生这种情况。这就是所谓的循环强引用。

解决循环强引用问题，可以通过定义类之间的关系为弱引用(weak)或无主引用(unowned)来代替强引用。这个过程在解决类实例之间的循环强引用中有描述。总之，在你学习如何解决循环强引用问题前，了解一下它是如何产生的也是很有意义的事情。

下面的例子展示了一个如何意外地创建循环强引用的例子。这个例子定义了两个类，分别是 Person 和 Apartment ，用来建模公寓和它其中的居民：

```
1 classPerson{  
2     letname:String  
3     init(name:String){self.name=name}  
4     varapartment:Apartment?  
5     deinit{print("\(name) is being deinitialized")}  
6 }  
7 classApartment{  
8     letunit:String  
9     init(unit:String){self.unit=unit}  
10    vartenant:Person?  
11    deinit{print("Apartment \(unit) is being deinitialized")}  
12 }  
13
```

每一个 Person 实例有一个类型为 String ，名字为 name 的属性，并有一个可选的初始化为 nil 的 apartment 属性。 apartment 属性是可选项，因为一个人并不总是拥有公寓。

类似的，每个 Apartment 实例都有一个叫 unit ，类型为 String 的属性，并有一个可选的初始化为 nil 的 tenant 属性。 tenant 属性是可选的，因为一栋公寓并不总是有居民。

这两个类都定义了反初始化器，用以在类实例被反初始化时输出信息。这让你能够知晓 Person 和 Apartment 的实例是否像预期的那样被释放。

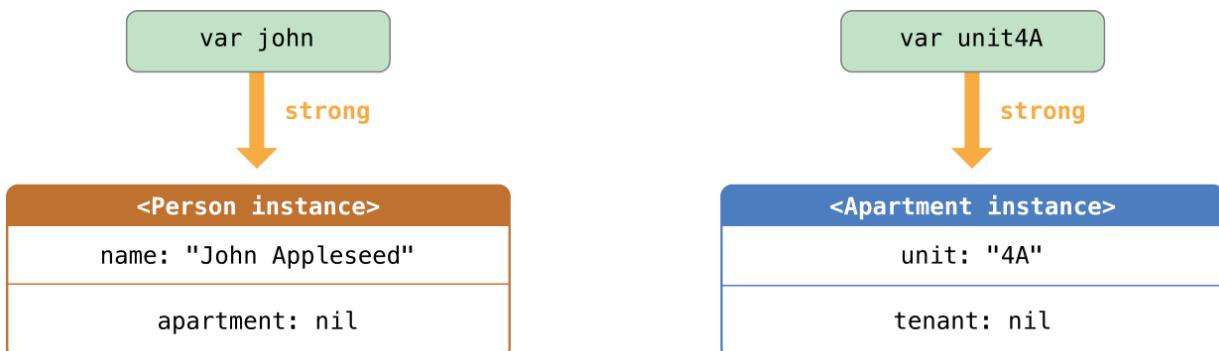
接下来的代码片段定义了两个可选项变量 john 和 unit4A ，它们分别被赋值为下面的 Apartment 和 Person 的实例。这两个变量都被初始化为 nil ，这正是可选项的优点：

```
1 varjohn:Person?  
2 varunit4A:Apartment?
```

现在你可以创建特定的 Person 和 Apartment 实例并将其赋值给 john 和 unit4A 变量：

```
1 john=Person(name:"John Appleseed")  
2 unit4A=Apartment(unit:"4A")
```

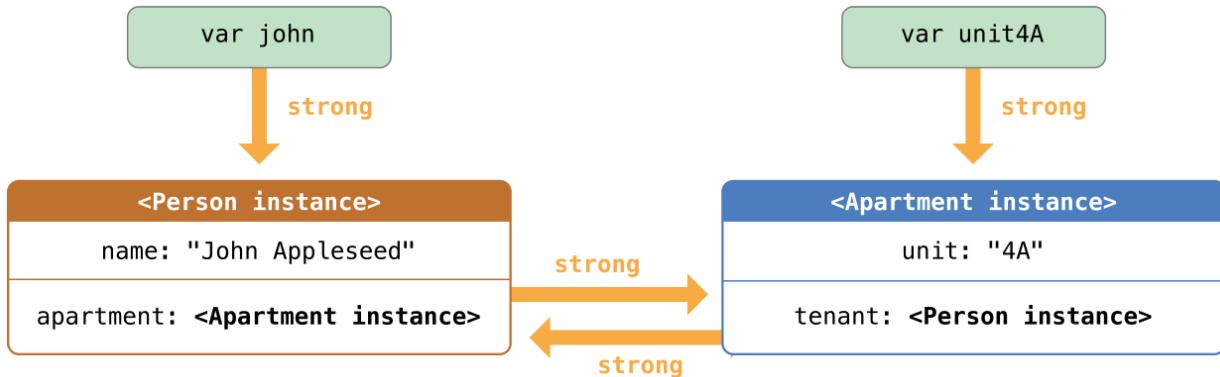
在两个实例的强引用创建和分配之后，下图表现了强引用的关系。 John 变量对 Person 实例有一个强引用， unit4A 变量对 Apartment 实例有一个强引用：



现在你就可以把这两个实例关联在一起，这样人就有公寓了，而且公寓有房间号。注意，感叹号(!)是用来展开和访问可选变量 john 和 unit4A 里的实例的，所以这些实例的属性可以设置：

```
1 john!.apartment=unit4A  
2 unit4A!.tenant=john
```

在将两个实例联系在一起之后，强引用的关系如图所示：

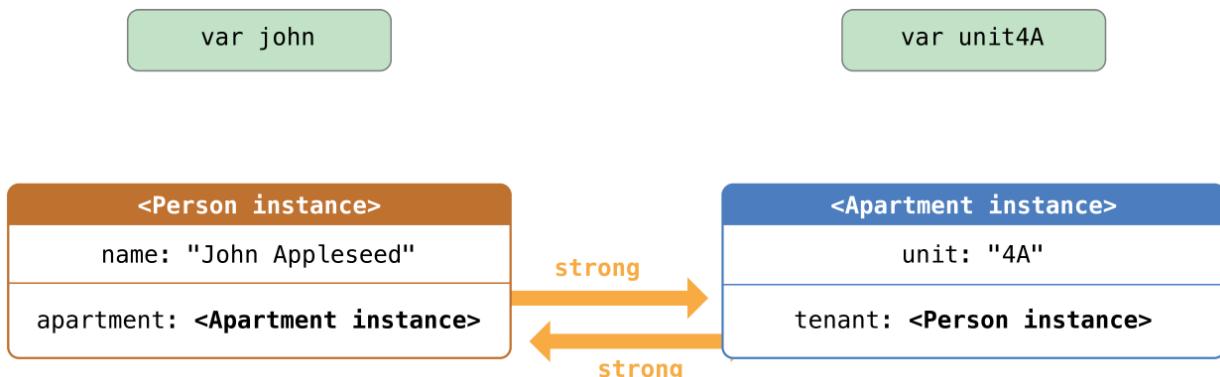


不幸的是，这两个实例关联后会产生一个循环强引用。Person 实例现在有了一个指向 Apartment 实例的强引用，而 Apartment 实例也有了一个指向 Person 实例的强引用。因此，当你断开 john 和 unit4A 变量所持有的强引用时，引用计数并不会降零，实例也不会被 ARC 释放：

```
1 john=nil  
2 unit4A=nil
```

注意，当你把这两个变量设为 nil 时，没有任何一个反初始化器被调用。循环强引用会一直阻止 Person 和 Apartment 类实例的释放，这就在你的应用程序中造成了内存泄漏。

在你将 john 和 unit4A 赋值为 nil 后，强引用关系如下图：



Person 和 Apartment 实例之间的强引用关系保留了下来并且不会被断开。

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：弱引用（weakreference）和无主引用（unownedreference）。

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为 nil 的实例使用弱引用。相反，对于初始化赋值后再也不会被赋值为 nil 的实例，使用无主引用。在实例的生命周期中，当引用可能“没有值”的时候，就使用弱引用避免循环引用。如同在无主引用中描述的那样，如果引用始终有值，则可以使用无主引

用来代替。上面的 Apartment 例子中，在它的声明周期中，有时是“没有居民”的，因此适合使用弱引用来解决循环强引用。

弱引用

弱引用不会对其引用的实例保持强引用，因而不会阻止 ARC 释放被引用的实例。这个特性阻止了引用变为循环强引用。声明属性或者变量时，在前面加上 weak 关键字表明这是一个弱引用。

由于弱引用不会强保持对实例的引用，所以说实例被释放了弱引用仍旧引用着这个实例也是有可能的。因此，ARC 会在被引用的实例被释放时自动地设置弱引用为 nil。由于弱引用需要允许它们的值为 nil，它们一定得是可选类型。

你可以检查弱引用的值是否存在，就像其他可选项的值一样，并且你将永远不会遇到“野指针”。

注意

在 ARC 给弱引用设置 nil 时不会调用属性观察者。

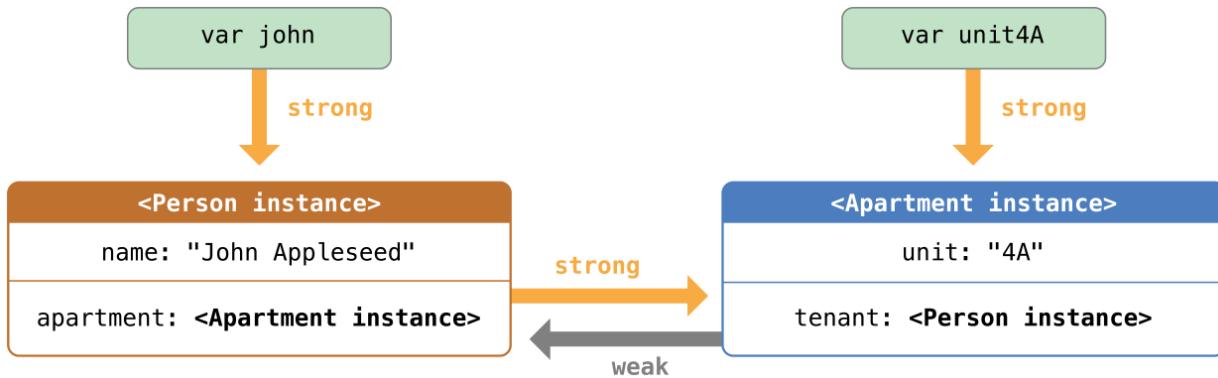
下面的例子跟上面 Person 和 Apartment 的例子一致，但是有一个重要的区别。这次，Apartment 的 tenant 属性被声明为弱引用：

```
1 classPerson{
2     letname:String
3     init(name:String){self.name=name}
4     varapartment:Apartment?
5     deinit{print("\(name) is being deinitialized")}
6 }
7 classApartment{
8     letunit:String
9     init(unit:String){self.unit=unit}
10    weakvartenant:Person?
11    deinit{print("Apartment \(unit) is being deinitialized")}
12 }
13
```

两个变量（john 和 unit4A）之间的强引用和关联创建得与上次相同：

```
1 varjohn:Person?
2 varunit4A:Apartment?
3 john=Person(name:"John Appleseed")
4 unit4A=Apartment(unit:"4A")
5 john!.apartment=unit4A
6 unit4A!.tenant=john
7
8
```

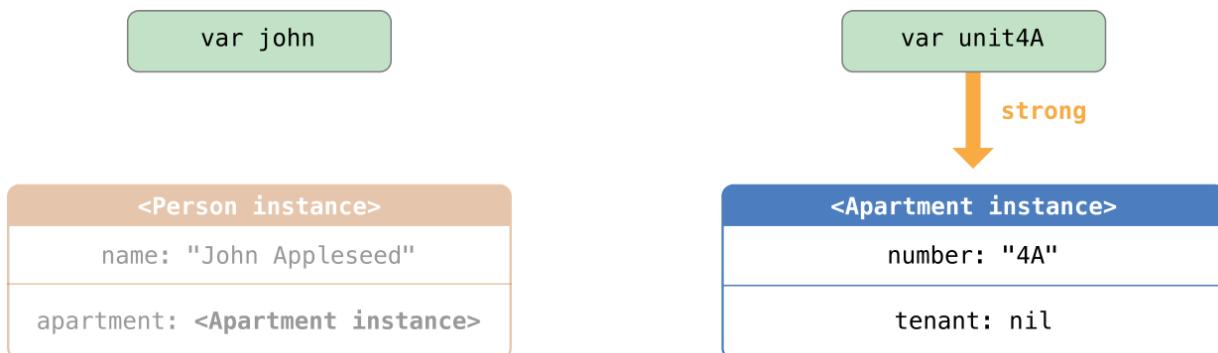
现在，两个关联在一起的实例的引用关系如下图所示：



Person 实例依然保持对 Apartment 实例的强引用，但是 Apartment 实例现在对 Person 实例是弱引用。这意味着当你断开 john 变量所保持的强引用时，再也没有指向 Person 实例的强引用了，由于再也没有指向 Person 实例的强引用，该实例会被释放：

```
1 john=nil
2 // prints "John Appleseed is being deinitialized"
```

由于再也没有强引用到 Person 它被释放掉了并且 tenant 属性被设置为 nil：



现在只剩下来自 unit4A 变量对 Apartment 实例的强引用。如果你打断这个强引用，那么 Apartment 实例就再也没有强引用了：

```
1 unit4A=nil
2 // prints "Apartment 4A is being deinitialized"
```

无主引用

和弱引用类似，无主引用不会牢牢保持住引用的实例。但是不像弱引用，总之，无主引用假定是永远有值的。因此，无主引用总是被定义为非可选类型。你可以在声明属性或者变量时，在前面加上关键字 `unowned` 表示这是一个无主引用。

由于无主引用是非可选类型，你不需要在使用它的时候将它展开。无主引用总是可以直接访问。不过 ARC 无法在实例被释放后将无主引用设为 `nil`，因为非可选类型的变量不允许被赋值为 `nil`。

注意

如果你试图在实例的被释放后访问无主引用，那么你将触发运行时错误。只有在你确保引用会一直引用实例的时候才使用无主引用。

还要注意的是，如果你试图访问引用的实例已经被释放了的无主引用，Swift 会确保程序直接崩溃。你不会因此而遭遇无法预期的行为。所以你应当避免这样的事情发生。

下面的例子定义了两个类，`Customer` 和 `CreditCard`，模拟了银行客户和客户的信用卡。这两个类中，每一个都将另外一个类的实例作为自身的属性。这种关系可能会造成循环强引用。

`Customer` 和 `CreditCard` 之间的关系与前面弱引用例子中 `Apartment` 和 `Person` 的关系略微不同。在这个数据模型中，一个客户可能有或者没有信用卡，但是一张信用卡总是关联着一个客户。为了表示这种关系，`Customer` 类有一个可选类型的 `card` 属性，但是 `CreditCard` 类有一个非可选类型的 `customer` 属性。

另外，新的 `CreditCard` 实例只有通过传送 `number` 值和一个 `customer` 实例到 `CreditCard` 的初始化器才能创建。这就确保了 `CreditCard` 实例在创建时总是有与之关联的 `customer` 实例。

由于信用卡总是关联着一个客户，因此将 `customer` 属性定义为无主引用，以避免循环强引用：

```
1 class Customer{
2     let name: String
3     var card: CreditCard?
4     init(name: String) {
5         self.name = name
6     }
7     deinit { print("\(name) is being deinitialized") }
8 }
9 class CreditCard {
10    let number: UInt64
11    unowned let customer: Customer
12    init(number: UInt64, customer: Customer) {
13        self.number = number
14        self.customer = customer
15    }
16    deinit { print("Card #\(number) is being deinitialized") }
17 }
18
```

注意：`CreditCard` 类的 `number` 属性定义为 `UInt64` 类型而不是 `Int`，以确保 `number` 属性的存储量在32位和64位系统上都能足够容纳16位的卡号。

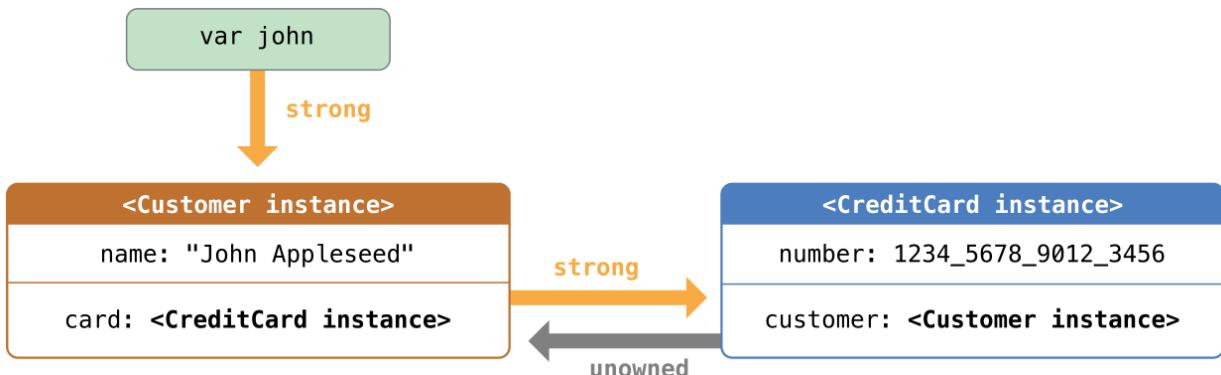
下面的代码片段定义了一个叫 `john` 的可选 `Customer` 变量，用来保存某个特定客户的引用。由于是可选项，所以变量被初始化为 `nil`。

```
1 var john: Customer?
```

现在你可以创建一个 `Customer` 实例，用它初始化和分配一个新的 `CreditCard` 实例作为 `customer` 的 `card` 属性：

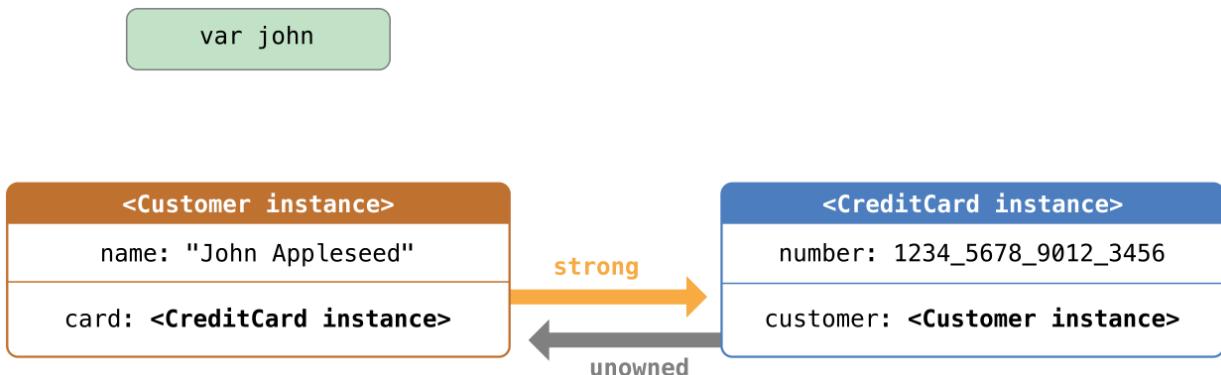
```
1 john=Customer(name:"John Appleseed")
2 john!.card=CreditCard(number:1234_5678_9012_3456 , customer:john!)
```

如下图，是你关联了两个实例后的图示关系：



现在 Customer 实例对 CreditCard 实例有一个强引用，并且 CreditCard 实例对 Customer 实例有一个无主引用。

由于 Customer 的无主引用，当你断开 john 变量持有的强引用时，那么就再也没有指向 Customer 实例的强引用了。



因为不再有 Customer 的强引用，该实例被释放了。其后，再也没有指向 CreditCard 实例的强引用，该实例也随之被释放了：

```
1 john=nil
2 // prints "John Appleseed is being deinitialized"
3 // prints "Card #1234567890123456 is being deinitialized"
```

最后的代码片段展示了在 `john` 变量被设为 `nil` 后 Customer 实例和 CreditCard 实例的反初始化器都打印出了“被释放”的信息。

注意

上边的例子展示了如何使用安全无主引用。Swift 还为你需要关闭运行时安全检查的情况提供了不安全无主引用——举例来说，性能优化的时候。对于所有的不安全操作，你要自己负责检查代码安全性。

使用 `unowned(unsafe)` 来明确使用了一个不安全无主引用。如果你在实例的引用被释放后访问这个不安全无主引用，你的程序就会尝试访问这个实例曾今存在过的内存地址，这就是不安全操作。

无主引用和隐式展开的可选属性

上面弱引用和无主引用例子涵盖了两种常用的需要打破循环强引用的场景。

Person 和 Apartment 的例子展示了两个属性的值都允许为 nil，并会潜在的产生循环强引用。这种场景最适合用弱引用来解决。

Customer 和 CreditCard 的例子展示了一个属性的值允许为 nil，而另一个属性的值不允许为 nil，这也可能导致循环强引用。这种场景最好使用无主引用来解决。

总之，还有第三种场景，在这种场景中，两个属性都必须有值，并且初始化完成后永远不会为 nil。在这种场景中，需要一个类使用无主属性，而另外一个类使用隐式展开的可选属性。

一旦初始化完成，这两个属性能被直接访问(不需要可选展开)，同时避免了循环引用。这一节将为你展示如何建立这种关系。

下面的例子定义了两个类，Country 和 City，每个类将另外一个类的实例保存为属性。在这个数据模型中，每个国家必须有首都，每个城市必须属于一个国家。为了实现这种关系，Country 类拥有一个 capitalCity 属性，而 City 类有一个 country 属性：

```
1 classCountry{
2     letname:String
3     varcapitalCity:City!
4     init(name:String,capitalName:String){
5         self.name=name
6         self.capitalCity=City(name:capitalName,country:self)
7     }
8 }
9 classCity{
10    letname:String
11    unownedletcountry:Country
12    init(name:String,country:Country){
13        self.name=name
14        self.country=country
15    }
16 }
17
```

为了建立两个类的依赖关系，City 的初始化器接收一个 Country 实例，并且将实例保存到 country 属性。

Country 的初始化器调用了 City 的初始化器。总之，如同在两段式初始化中描述的那样，只有 Country 的实例完全初始化完后，Country 的初始化器才能把 self 传给 City 的初始化器。

为了满足这种需求，通过在类型结尾处加上感叹号（City!）的方式，以声明 Country 的 capitalCity 属性为一个隐式展开的可选属性。如同在隐式展开可选项中描述的那样，这意味着像其他可选项一样，capitalCity 属性有一个默认值 nil，但是不需要展开它的值就能访问它。

由于 capitalCity 默认值为 nil，一旦 Country 的实例在初始化器中给 name 属性赋值后，整个初始化过程就完成了。这意味着一旦 name 属性被赋值后，Country 的初始化器就能引用并传递隐式的 self。Country 的初始化器在赋值 capitalCity 时，就能将 self 作为参数传递给 City 的初始化器。

以上的意义在于你可以通过一条语句同时创建 Country 和 City 的实例，而不产生循环强引用，并且 capitalCity 的属性能被直接访问，而不需要通过感叹号来展开它的可选值：

```
1 varcountry=Country(name:"Canada",capitalName:"Ottawa")
2 print("\(country.name)'s capital city is called \(country.capitalCity.name)")
3 // prints "Canada's capital city is called Ottawa"
```

在上面的例子中，使用隐式展开的可选属性的意义在于满足了两段式类初始化器的需求。capitalCity 属性在初始化完成后，就能像非可选项一样使用和存取同时还避免了循环强引用。

闭包的循环强引用

上面我们看到了循环强引用是如何在两个实例属性互相保持对方的强引用时产生的，还知道了如何用弱引用和无主引用来打破这些循环强引用。

循环强引用还会出现在你把一个闭包分配给类实例属性的时候，并且这个闭包中又捕获了这个实例。捕获可能发生于这个闭包函数体中访问了实例的某个属性，比如 self.someProperty，或者这个闭包调用了一个实例的方法，例如 self.someMethod()。这两种情况都导致了闭包“捕获”了 self，从而产生了循环强引用。

循环强引用的产生，是因为闭包和类相似，都是引用类型。当你把闭包赋值给了一个属性，你实际上是把一个引用赋值给了这个闭包。实质上，这跟之前上面的问题是一样的——两个强引用让彼此一直有效。总之，和两个类实例不同，这次一个是类实例和一个闭包互相引用。

Swift 提供了一种优雅的方法来解决这个问题，称之为闭包捕获列表（closuer capture list）。不过，在学习如何用闭包捕获列表打破循环强引用之前，我们还是先来了解一下这个循环强引用是如何产生的，这对我们很有帮助。

下面的例子为你展示了当一个闭包引用了 self 后是如何产生一个循环强引用的。例子中定义了一个叫 HTMLElement 的类，用一种简单的模型表示 HTML 中的一个单独的元素：

```
1 classHTMLElement{
2     letname:String
3     lettext:String?
4     lazy varasHTML:()->String={
5         iflettext=self.text{
6             return"<\(self.name)>\(text)</\(&(self.name)>""
7         }else{
8             return"<\(self.name) />"
9         }
10    }
11    init(name:String,text:String?=nil){
12        self.name=name
13        self.text=text
14    }
15    deinit{
16        print("\(name) is being deinitialized")
17    }
18}
19
20
21
22
23
```

`HTMLElement` 类定义了一个 `name` 属性来表示这个元素的名称，例如表示标题元素的 "`h1`"、代表段落元素的 "`p`"、或者代表换行元素的 "`br`"。 `HTMLElement` 还定义了一个可选的属性 `text`，它可以用来设置和展现 HTML 元素的文本。

除了上面的两个属性，`HTMLElement` 还定义了一个 `lazy` 属性 `asHTML`。这个属性引用了一个将 `name` 和 `text` 组合成 HTML 字符串片段的闭包。该属性是 `Void->String` 类型，或者可以理解为“一个没有参数，但返回 `String` 的函数”。

默认情况下，闭包赋值给了 `asHTML` 属性，这个闭包返回一个代表 HTML 标签的字符串。如果 `text` 值存在，该标签就包含可选值 `text`；如果 `text` 不存在，该标签就不包含文本。对于段落元素，根据 `text` 是 "`some text`" 还是 `nil`，闭包会返回 "`<p>some text</p>`" 或者 "`<p />`"。

可以像实例方法那样去命名、使用 `asHTML` 属性。总之，由于 `asHTML` 是闭包而不是实例方法，如果你想改变特定元素的 HTML 处理的话，可以用自定义的闭包来取代默认值。

```
1 let heading=HTMLElement{name:"h1"}  
2 let defaultText="some default text"  
3 heading.asHTML={  
4   return"<\(heading.name)>\(heading.text??defaultText)<\\(heading.name)>"  
5 }  
6 print(heading.asHTML())  
7 // prints "<h1>some default text</h1>"
```

注意：

`asHTML` 声明为 `lazy` 属性，因为只有当元素确实需要处理为 HTML 输出的字符串时，才需要使用 `asHTML`。实际上 `asHTML` 是延迟加载属性意味着你在默认的闭包中可以使用 `self`，因为只有当初始化完成以及 `self` 确实存在后，才能访问延迟加载属性。

`HTMLElement` 类只提供一个初始化器，通过 `name` 和 `text`（如果有的话）参数来初始化一个元素。该类也定义了一个初始化器，当 `HTMLElement` 实例被释放时打印一条消息。

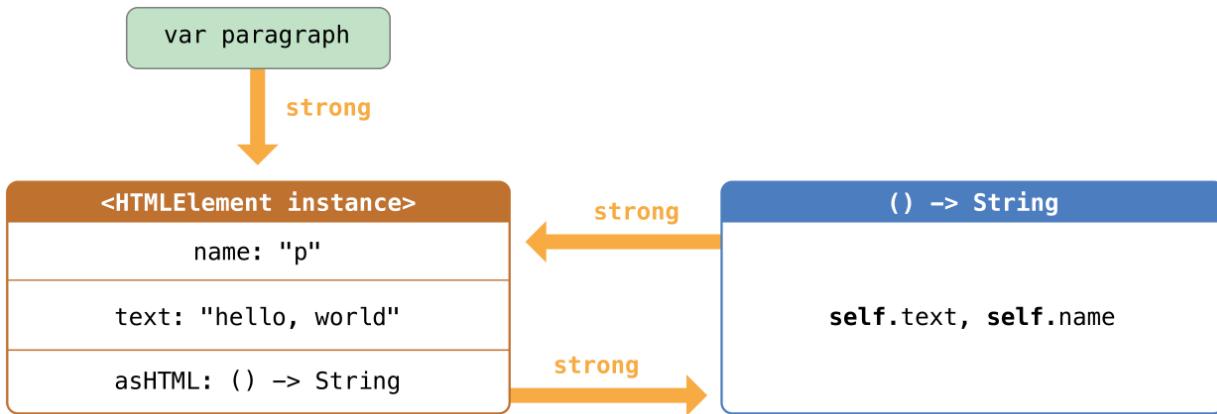
下面的代码展示了如何用 `HTMLElement` 类创建实例并打印消息。

```
1 var paragraph:HTMLElement?=HTMLElement{name:"p",text:"hello, world"}  
2 print(paragraph!.asHTML())  
3 // prints "hello, world"
```

注意：

上面的 `paragraph` 变量定义为可选 `HTMLElement`，因此我们接下来可以赋值 `nil` 给它来演示循环强引用。

不幸的是，上面写的 `HTMLElement` 类产生了类实例和 `asHTML` 默认值的闭包之间的循环强引用。循环强引用如下图所示：



实例的 `asHTML` 属性持有闭包的强引用。但是，闭包在其闭包体内使用了 `self`（引用了 `self.name` 和 `self.text`），因此闭包捕获了 `self`，这意味着闭包又反过来持有了 `HTMLElement` 实例的强引用。这样两个对象就产生了循环强引用。（更多关于闭包捕获值的信息，请参考[值捕获](#)）。

注意

尽管闭包多次引用了 `self`，它只捕获 `HTMLElement` 实例的一个强引用。

如果设置 `paragraph` 变量为 `nil`，打破它持有的 `HTMLElement` 实例的强引用，`HTMLElement` 实例和它的闭包都不会被释放，也是因为循环强引用：

```
1 paragraph=nil
```

注意 `HTMLElement` 的反初始化器中的消息并没有被打印，证明了 `HTMLElement` 实例并没有被销毁。

解决闭包的循环强引用

你可以通过定义捕获列表作为闭包的定义来解决在闭包和类实例之间的循环强引用。捕获列表定义了当在闭包体里捕获一个或多个引用类型的规则。正如在两个类实例之间的循环强引用，声明每个捕获的引用为引用或无主引用而不是强引用。应当根据代码关系来决定使用弱引用还是无主引用。

注意

Swift 要求你在闭包中引用 `self` 成员时使用 `self.someProperty` 或者 `self.someMethod`（而不仅仅是 `someProperty` 或 `someMethod`）。这有助于提醒你可能会一不小心就捕获了 `self`。

定义捕获列表

捕获列表中的每一项都由 `weak` 或 `unowned` 关键字与类实例的引用（如 `self`）或初始化过的变量（如 `delegate=self.delegate!`）成对组成。这些项写在方括号中用逗号分开。

把捕获列表放在形式参数和返回类型前边，如果它们存在的话：

```
1 lazy varsomeClosure:(Int,String)->String={  
2 [unownedself,weakdelegate=self.delegate!]in  
3 // closure body goes here  
4 }
```

如果闭包没有指明形式参数列表或者返回类型，是因为它们会通过上下文推断，那么就把捕获列表放在关键字 `in` 前边，闭包最开始的地方：

```
1 lazy varsomeClosure:()->String={  
2 [unownedself,weakdelegate=self.delegate!]in  
3 // closure body goes here  
4 }
```

弱引用和无主引用

在闭包和捕获的实例总是互相引用并且总是同时释放时，将闭包内的捕获定义为无主引用。

相反，在被捕获的引用可能会变为 `nil` 时，定义一个弱引用的捕获。弱引用总是可选项，当实例的引用释放时会自动变为 `nil`。这使我们可以在闭包体内检查它们是否存在。

注意

如果被捕获的引用永远不会变为 `nil`，应该用无主引用而不是弱引用。

前面的 `HTMLElement` 例子中，无主引用是正确的解决循环强引用的方法。这样编写 `HTMLElement` 类来避免循环强引用：

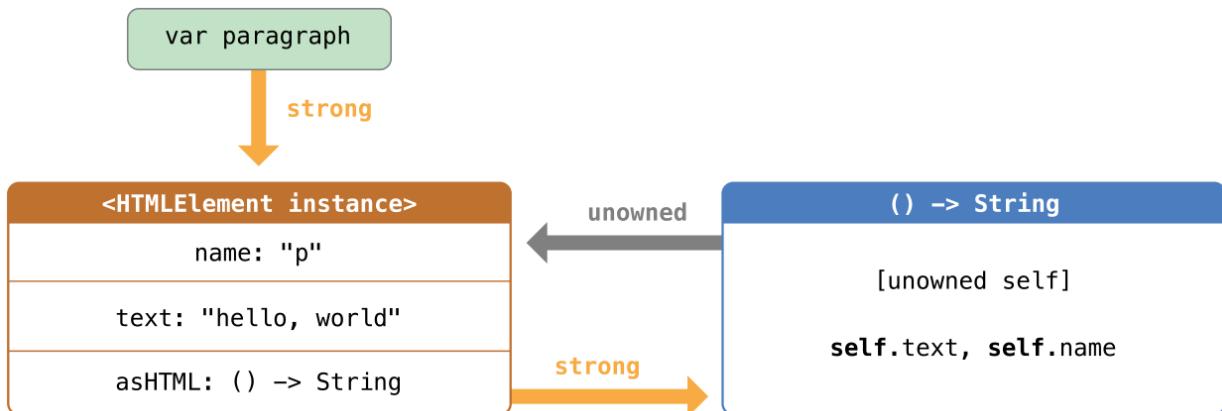
```
1 class HTMLElement{  
2 letname:String  
3 lettext:String?  
4 lazy varasHTML:()->String={  
5 [unownedself]in  
6 iflettext=self.text{  
7 return"<\(self.name)>\(text)</\((self.name)>"  
8 }else{  
9 return"<\(self.name) />"  
10 }  
11 }  
12 init(name:String,text:String?=nil){  
13 self.name=name  
14 self.text=text  
15 }  
16 deinit{  
17 print("\(name) is being deinitialized")  
18 }  
19 }  
20  
21  
22  
23  
24
```

上面的 `HTMLElement` 实现和之前的实现一致，除了在 `asHTML` 闭包中多了一个捕获列表。这里，捕获列表是 `[unownedself]`，表示“用无主引用而不是强引用来捕获 `self`”。

和之前一样，我们可以创建并打印 `HTMLElement` 实例：

```
1 varparagraph:HTMLElement?=HTMLElement(name:"p",text:"hello, world")
2 print(paragraph!.asHTML())
3 // prints "<p>hello, world</p>"
```

使用捕获列表后引用关系如下图所示：



这次，闭包以无主引用的形式捕获 `self`，并不会持有 `HTMLElement` 实例的强引用。如果将 `paragraph` 赋值为 `nil`，`HTMLElement` 实例将会被释放，并能看到它的反初始化器打印出的消息。

```
1 paragraph=nil
2 // prints "p is being deinitialized"
```

了解更多关于捕获列表，请看[捕获列表](#)。

可选链

 cnswift.org/optional-chaining

可选链是一个调用和查询可选属性、方法和下标的过程，它可能为 nil。如果可选项包含值，属性、方法或者下标的调用成功；如果可选项是 nil，属性、方法或者下标的调用会返回 nil。多个查询可以链接在一起，如果链中任何一个节点是 nil，那么整个链就会得体地失败。

注意

Swift 中的可选链与 Objective-C 中的 nil 信息类似，但是它却工作在任意类型上，而且它能检测成功还是失败。

可选链代替强制展开

你可以通过在你希望如果可选项为非 nil 就调用属性、方法或者脚本的可选值后边使用问号（?）来明确可选链。这和在可选值后放感叹号（!）来强制展开它的值非常类似。主要的区别在于可选链会在可选项为 nil 时得体地失败，而强制展开则在可选项为 nil 时触发运行时错误。

为了显示出可选链可以在 nil 值上调用，可选链调用的结果一定是一个可选值，就算你查询的属性、方法或者下标返回的是非可选值。你可以使用这个可选项返回值来检查可选链调用是成功（返回的可选项包含值），还是由于链中出现了 nil 而导致没有成功（返回的可选值是 nil）。

另外，可选链调用的结果与期望的返回值类型相同，只是包装成了可选项。通常返回 Int 的属性通过可选链后会返回一个 Int?。

接下来的一些代码片段演示了可选链与强制展开的不同并允许你检查是否成功。

首先，定义两个类， Person 和 Residence：

```
1 classPerson{  
2     varresidence:Residence?  
3 }  
4 classResidence{  
5     varnumberOfRooms=1  
6 }  
7
```

Residence 实例有一个 Int 属性叫做 numberOfRooms，它带有默认值 1。Person 实例有一个 Residence? 类型的可选 residence 属性。

如果你创建一个新的 Person 实例，得益于可选项的特性，它的 residence 属性会默认初始化为 nil。下面的代码中， john 拥有值为 nil 的 residence 属性：

```
1 letjohn=Person()
```

如果你尝试访问这个人的 residence 里的 numberOfRooms 属性，通过在 residence 后放一个叹号来强制展开它的值，你会触发一个运行时错误，因为 residence 根本没有值可以展开：

```
1 if let roomCount = john.residence!.numberOfRooms
2 // this triggers a runtime error
```

上边的代码会在 john.residence 有一个非 nil 值时成功并且给 roomCount 赋值一个包含合适房间号的 Int 值。总之，这段代码一定会在 residence 为 nil 时触发运行时错误，如同上边展示的那样。

可选链提供另一种访问 numberOfRooms 的方法。要使用可选链，使用问号而不是叹号：

```
1 if let roomCount = john.residence?.numberOfRooms{
2 print("John's residence has \(roomCount) room(s).")
3 }else{
4 print("Unable to retrieve the number of rooms.")
5 }
6 // Prints "Unable to retrieve the number of rooms."
```

这将会告诉 Swift 把可选 residence 属性“链接”起来并且取回 numberOfRooms 的值，如果 residence 存在的话。

由于尝试访问 numberOfRooms 有失败的潜在可能，可选链尝试返回一个 Int? 类型的值，或者说“可选 Int”。当 residence 为 nil，就如同上边的栗子，这个可选 Int 将也会是 nil，来反映出不能访问 numberOfRooms 这个事实。可选 Int 通过可选绑定来展开整数并赋值非可选值给 roomCount 变量。

注意就算 numberOfRooms 是非可选的 Int 也是适用的。事实上通过可选链查询就意味着对 numberOfRooms 的调用一定会返回 Int? 而不是 Int。

你可以赋值一个 Residence 实例给 john.residence，这样它就不会再有 nil 值：

```
1 john.residence = Residence()
```

john.residence 现在包含了实际的 Residence 实例，而不是 nil。如果你尝试用与之前相同的可选链访问 numberOfRooms，它就会返回一个 Int? 包含默认 numberOfRooms 值 1：

```
1 if let roomCount = john.residence?.numberOfRooms{
2 print("John's residence has \(roomCount) room(s).")
3 }else{
4 print("Unable to retrieve the number of rooms.")
5 }
6 // Prints "John's residence has 1 room(s.)"
```

为可选链定义模型类

你可以使用可选链来调用属性、方法和下标不止一个层级。这允许你在相关类型的复杂模型中深入到子属性，并检查是否可以在这些自属性里访问属性、方法和下标。

下边的代码段定义了四个模型类用于随后的栗子，包括多层可选链的栗子。这些栗子通过添加 Room 和 Address 类扩展了上边的 Person 和 Residence 模型，以及相关的属性、方法和

下标。

Person 类与之前的定义方式相同：

```
1 classPerson{  
2 varresidence:Residence?  
3 }
```

Residence 类比以前要复杂一些。这次， Residence 类定义了一个叫做 rooms 的变量属性，它使用一个空的 [Room] 类型空数组初始化：

```
1 classResidence{  
2 varrooms=[Room]()  
3 varnumberOfRooms: Int{  
4 returnrooms.count  
5 }  
6 subscript(i:Int)->Room{  
7 get{  
8 returnrooms[i]  
9 }  
10 set{  
11 rooms[i]=newValue  
12 }  
13 }  
14 funcprintNumberOfRooms(){  
15 print("The number of rooms is \$(numberOfRooms)")  
16 }  
17 varaddress:Address?  
18 }
```

由于这个版本的 Residence 储存了 Room 实例的数组，它的 numberOfRooms 属性使用计算属性来实现，而不是储存属性。计算属性 numberOfRooms 只是返回 rooms 数组的 count 属性值。

作为给它的 rooms 数组赋值的快捷方式，这个版本的 Residence 提供了一个可读写的下标来访问 rooms 数组的索引位置。

这个版本的 Residence 同样提供了一个叫做 printNumberOfRooms 的方法，它打印住所中的房间号。

最终， Residence 定义了一个可选属性叫做 address ，它是一个 Address? 类型，这个属性的 Address 类类型在下面定义。

rooms 数组使用的 Room 类型仅有一个属性叫做 name ，还有一个初始化器来给这个属性设置合适的房间名：

```
1 classRoom{  
2 letname:String  
3 init(name:String){self.name=name}  
4 }
```

这个模型的最后一个类型叫做 Address 。这个类型有三个 String? 类型可选属性。前两个属性， buildingName 和 buildingNumber ，是定义地址中特定建筑部分的代替方式。第三个属性， street ，是给地址里街道命名的：

```

1 classAddress{
2     varbuildingName:String?
3     varbuildingNumber:String?
4     varstreet:String?
5     funcbuildingIdentifier()->String?{
6         ifbuildingName!=nil{
7             returnbuildingName
8         }elseifbuildingNumber!=nil&&street!=nil{
9             return"\(buildingNumber) \\(street)"
10        }else{
11            returnnil
12        }
13    }
14 }
```

Address 类同样提供了一个方法叫做 buildingIdentifier()，它有一个 String? 类型的返回值。这个方法检查地址的属性并返回 buildingName 如果它有值的话，或者把 buildingNumber 与 street 串联起来，如果它们都有值的话，或者就是 nil。

通过可选链访问属性

如同可选链代替强制展开中展示的那样，你可以使用可选链来访问可选值里的属性，并且检查这个属性的访问是否成功。

使用上边定义的类来创建一个新得 Person 实例，并且尝试如之前一样访问它的 numberOfRooms 属性：

```

1 letjohn=Person()
2 ifletroomCount=john.residence?.numberOfRooms{
3     print("John's residence has \(roomCount) room(s).")
4 }else{
5     print("Unable to retrieve the number of rooms.")
6 }
7 // Prints "Unable to retrieve the number of rooms."
```

由于 john.residence 是 nil，这个可选链调用与之前一样失败了。

你同样可以尝试通过可选链来给属性赋值：

```

1 letsomeAddress=Address()
2 someAddress.buildingNumber="29"
3 someAddress.street="Acacia Road"
4 john.residence?.address=someAddress
```

在这个栗子中，给 john.residence 的 address 属性赋值会失败，因为 john.residence 目前是 nil。

这个赋值是可选链的一部分，也就是说 = 运算符右手侧的代码都不会被评判。在先前的栗子中，不容易看出 someAddress 没有被评判，因为赋值一个常量不会有任何副作用。下边的栗子做同样的赋值，但它使用一个函数来创建地址。函数会在返回值之前打印“函数被调用了”，这可以让你看到 = 运算符右手侧是否被评判。

```
1 func createAddress() -> Address{  
2     print("Function was called.")  
3     let someAddress = Address()  
4     someAddress.buildingNumber = "29"  
5     someAddress.street = "Acacia Road"  
6     return someAddress  
7 }  
8 john.residence?.address = createAddress()  
9  
10
```

你可以看到 `createAddress()` 函数没有被调用，因为没有任何东西打印出来。

通过可选链调用方法

你可以使用可选链来调用可选项里的方法，并且检查调用是否成功。你甚至可以在没有定义返回值的方法上这么做。

`Residence` 类中的 `printNumberOfRooms()` 方法打印了当前 `numberOfRooms` 的值。方法看起来长这样：

```
1 func printNumberOfRooms(){  
2     print("The number of rooms is \(numberOfRooms)")  
3 }
```

这个方法没有指定返回类型。总之，如没有返回值的函数中描述的那样，函数和方法没有返回类型就隐式地指明为 `Void` 类型。意思是说它们返回一个 `()` 的值或者是一个空的元组。

如果你用可选链在可选项里调用这个方法，方法的返回类型将会是 `Void?`，而不是 `Void`，因为当你通过可选链调用的时候返回值一定会是一个可选类型。这允许你使用 `if` 语句来检查是否能调用 `printNumberOfRooms()` 方法，就算是方法自身没有定义返回值也可以。通过对比调用 `printNumberOfRooms` 返回的值是否为 `nil` 来确定方法的调用是否成功：

```
1 if john.residence?.printNumberOfRooms() != nil{  
2     print("It was possible to print the number of rooms.")  
3 } else{  
4     print("It was not possible to print the number of rooms.")  
5 }  
6 // Prints "It was not possible to print the number of rooms."
```

如果你尝试通过可选链来设置属性也是一样的。上边通过可选链访问属性中的例子尝试设置 `address` 值给 `john.residence`，就算是 `residence` 属性是 `nil` 也行。任何通过可选链设置属性的尝试都会返回一个 `Void?` 类型值，它允许你与 `nil` 比较来检查属性是否设置成功：

```
1 if(john.residence?.address = someAddress) != nil{  
2     print("It was possible to set the address.")  
3 } else{  
4     print("It was not possible to set the address.")  
5 }  
6 // Prints "It was not possible to set the address."
```

通过可选链访问下标

你可以使用可选链来给可选项下标取回或设置值，并且检查下标的调用是否成功。

注意

当你通过可选链访问一个可选项的下标时，你需要把问号放在下标括号的前边，而不是后边。可选链的问号一定是紧跟在可选项表达式的后边的。

下边的栗子尝试使用下标取回 Residence 类里 john.residence 属性的数组 rooms 里第一个房间的名字。由于 john.residence 目前是 nil，下标的调用失败了：

```
1 if let firstRoomName = john.residence?[0].name{  
2     print("The first room name is \(firstRoomName).")  
3 } else{  
4     print("Unable to retrieve the first room name.")  
5 }  
6 // Prints "Unable to retrieve the first room name."
```

可选链问号在下标的调用中紧跟 john.residence，在下标的方括号之前，因为 john.residence 在可选链被访问时是可选值。

同样的，你可以尝试通过在可选链里用下标来设置一个新值：

```
1 john.residence?[0] = Room(name: "Bathroom")
```

这个下标设置的尝试同样失败了，因为 residence 目前还是 nil。

如果你创建并且赋值一个实际的 Residence 实例给 john.residence，在 rooms 数组里添加一个或者多个 Room 实例，你就可以通过可选链使用 Residence 下标来访问 rooms 数组里的实际元素了：

```
1 let johnsHouse = Residence()  
2 johnsHouse.rooms.append(Room(name: "Living Room"))  
3 johnsHouse.rooms.append(Room(name: "Kitchen"))  
4 john.residence = johnsHouse  
5 if let firstRoomName = john.residence?[0].name{  
6     print("The first room name is \(firstRoomName).")  
7 } else{  
8     print("Unable to retrieve the first room name.")  
9 }  
10 // Prints "The first room name is Living Room."  
11
```

访问可选类型的下标

如果下标返回一个可选类型的值——比如说 Swift 的 Dictionary 类型的键下标——放一个问号在下标的方括号后边来链接它的可选返回值：

```
1 var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]  
2 testScores["Dave"]?[0] = 91  
3 testScores["Bev"]?[0] += 1  
4 testScores["Brian"]?[0] = 72  
5 // the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]
```

上面的栗子中定义了一个叫做 testScores 的字典，它包含两个键值对把 String 类型的键映射到一个整型值的数组。这个栗子用可选链把 "Dave" 数组中第一个元素设为 91；把 "Bev" 数组的第一个元素增加 1；然后尝试设置 "Brian" 数组中的第一个元素。前两个调用是成功了，因为 testScores 字典包含了 "Dave" 和 "Bev" 这两个键。第三个调用失败了，因为字典 testScores 并没有包含 "Brian" 键。

链的多层连接

你可以通过连接多个可选链来在模型中深入访问属性、方法以及下标。总之，多层可选链不会给返回的值添加多层的可选性。

也就是说：

- 如果你访问的值不是可选项，它会因为可选链而变成可选项；
- 如果你访问的值已经是可选的，它不会因为可选链而变得更加可选。

因此：

- 如果你尝试通过可选链取回一个 Int 值，就一定会返回 Int?，不论通过了多少层的可选链；
- 类似地，如果你尝试通过可选链访问 Int? 值，Int? 一定就是返回的类型，无论通过了多少层的可选链。

下边的栗子尝试访问 john 的 residence 属性里的 address 属性里的 street 属性。这里一共使用了两层可选链，以链接 residence 和 address 属性，它们都是可选类型：

```
1 if let johnsStreet = john.residence?.address?.street{  
2     print("John's street name is \(johnsStreet).")  
3 } else{  
4     print("Unable to retrieve the address.")  
5 }  
6 // Prints "Unable to retrieve the address."
```

john.residence 的值当前包含合法的 Residence 实例。总之，john.residence.address 的值目前为 nil。因此，john.residence?.address?.street 的调用失败了。

需要注意的是上面的栗子中，你尝试取回的 street 属性。它的类型为 String?。

john.residence?.address?.street 的返回值自然也是 String?，即使对属性的可选项来说已经通过了两层可选链。

如果你设置一个 Address 实例作为 john.residence.address 的值，并且为地址的 street 属性设置一个实际的值，你就可以通过多层可选链访问 street 属性的值了：

```
1 let johnsAddress=Address()
2 johnsAddress.buildingName="The Larches"
3 johnsAddress.street="Laurel Street"
4 john.residence?.address=johnsAddress
5 if let johnsStreet=john.residence?.address?.street{
6     print("John's street name is \(johnsStreet).")
7 }else{
8     print("Unable to retrieve the address.")
9 }
10 // Prints "John's street name is Laurel Street."
11
```

在上面的栗子中，对 `john.residence` 的 `address` 属性赋值能够成功，是因为 `john.residence` 的值目前包含了一个可用的 `Residence` 实例。

用可选返回值链接方法

先前的例子说明了如何通过可选链来获取可选类型属性的值。你还可以通过可选链来调用返回可选类型的方法，并且如果需要的话可以继续对方法的返回值进行链接。

在下面的栗子通过可选链来调用 `Address` 的 `buildingIdentifier()` 方法。这个方法返回 `String?` 类型值。正如上面所说，通过可选链调用的方法的最终返回的类型还是 `String?`：

```
1 if let buildingIdentifier=john.residence?.address?.buildingIdentifier(){
2     print("John's building identifier is \(buildingIdentifier).")
3 }
4 // Prints "John's building identifier is The Larches."
```

如果你要进一步对方法的返回值进行可选链，在方法 `buildingIdentifier()` 的圆括号后面加上可选链问号：

```
1 if let beginsWithThe=
2     john.residence?.address?.buildingIdentifier()?.hasPrefix("The"){
3         if beginsWithThe{
4             print("John's building identifier begins with \"The\". ")
5         }else{
6             print("John's building identifier does not begin with \"The\". ")
7         }
8     }
9 // Prints "John's building identifier begins with \"The\"."
```

注意

在上面的例子中，在方法的圆括号后面加上可选链问号，是因为链中的可选项是 `buildingIdentifier()` 的返回值，而不是 `buildingIdentifier()` 方法本身。

错误处理

 cnswift.org/error-handling

错误处理是相应和接收来自你程序中错误条件的过程。Swift 给运行时可恢复错误的抛出、捕获、传递和操纵提供了一类支持。

有些函数和方法不能保证总能完全执行或者产生有用的输出。可选项用来表示不存在值，但是当函数错误，能够了解到什么导致了错误将会变得很有用处，这样你的代码就能根据错误来响应了。

举例来说，假设一个阅读和处理来自硬盘上文件数据的任务。这种情况下有很多种导致任务失败的方法，目录中文件不存在，文件没有读权限，或者文件没有以兼容格式编码。从这些错误中区分不同的状况将能够让程序解决和从这些错误中恢复，并且把不能解决的错误通知给用户。

注意

在 Swift 中的错误处理表示法兼容于 Cocoa 和 Objective-C 中的 NSError 类错误处理模式，参考与 [Cocoa 和 Objective-C 一起使用 Swift \(Swift 3\)](#) (官方链接) 中的 [错误处理](#) (官方链接)。

表示和抛出错误

在 Swift 中，错误表示为遵循 Error 协议类型的值。这个空的协议明确了一个类型可以用于错误处理。

Swift 枚举是典型的为一组相关错误条件建模的完美配适类型，关联值还允许错误错误通讯携带额外的信息。比如说，这是你可能会想到的游戏里自动售货机会遇到的错误条件：

```
1 enum VendingMachineError: Error{  
2     case invalidSelection  
3     case insufficientFunds(coinsNeeded:Int)  
4     case outOfStock  
5 }
```

抛出一个错误允许你明确某些意外的事情发生了并且正常的执行流不能继续下去。你可以使用 throw 语句来抛出一个错误。比如说，下面的代码通过抛出一个错误来明确自动售货机需要五个额外的金币：

```
1 throw VendingMachineError.insufficientFunds(coinsNeeded:5)
```

处理错误

当一个错误被抛出，周围的某些代码必须为处理错误响应——比如说，为了纠正错误，尝试替代方案，或者把错误通知用户。

在 Swift 中有四种方式来处理错误。你可以将来自函数的错误传递给调用函数的代码中，使用 do-catch 语句来处理错误，把错误作为可选项的值，或者错误不会发生的断言。每一种方法都在下边的章节中有详细叙述。

当函数抛出一个错误，它就改变了你程序的流，所以能够快速定位错误就显得格外重要。要定位你代码中的这些位置，使用 try 关键字——或者 try? 或 try! 变体——放在调用函数、方法或者会抛出错误的初始化器代码之前。这些关键字在下面的章节中有详细的描述。

注意

Swift 中的错误处理，try, catch 和 throw 的使用与其他语言中的异常处理很相仿。不同于许多语言中的异常处理——包括 Objective-C——Swift 中的错误处理并不涉及调用堆栈展开，一个高占用过程。因此，throw 语句的性能特征与 return 比不差多少。

使用抛出函数传递错误

为了明确一个函数或者方法可以抛出错误，你要在它的声明当中的形式参数后边写上 throws 关键字。使用 throws 标记的函数叫做 **抛出函数**。如果它明确了一个返回类型，那么 throws 关键字要在返回箭头 (->) 之前。

```
1 func canThrowErrors() throws -> String  
2 func cannotThrowErrors() -> String  
3
```

抛出函数可以把它内部抛出的错误传递到它被调用的生效范围之内。

注意

只有抛出函数可以传递错误。任何在非抛出函数中抛出的错误都必须在该函数内部处理。

在抛出函数体内的任何地方，你都可以用 throw 语句来抛出错误。

在下边的栗子中，VendingMachine 类拥有一个如果请求的物品不存在、卖光了或者比押金贵了就会抛出对应的 VendingMachineError 错误的 vend(itemNamed:) 方法：

```

1 structItem{
2     varprice:Int
3     varcount:Int
4 }
5 classVendingMachine{
6     varinventory=[]
7     "Candy Bar":Item(price:12,count:7),
8     "Chips":Item(price:10,count:4),
9     "Pretzels":Item(price:7,count:11)
10 ]
11 varcoinsDeposited=0
12 funcvend(itemNamed name:String)throws{
13     guard letitem=inventory[name]else{
14         throwVendingMachineError.invalidSelection
15     }
16     guard item.count>0else{
17         throwVendingMachineError.outOfStock
18     }
19     guard item.price<=coinsDepositedelse{
20         throwVendingMachineError.insufficientFunds(coinsNeeded:item.price-coinsDeposited)
21     }
22     coinsDeposited-=item.price
23     varnewItem=item
24     newItem.count-=1
25     inventory[name]=newItem
26     print("Dispensing \\"name")"
27 }
28 }
29
30
31
32
33
34
35

```

vend(itemNamed:)方法的实现使用了 guard语句来提前退出并抛出错误，如果购买零食的条件不符合的话。因为 throw语句立即传递程序控制，所以只有所有条件都达到，物品才会售出。

由于 vend(itemNamed:)方法传递它抛出的任何错误，所以你调用它的代码要么直接处理错误——使用 `do-catch` 语句，`try?`或者 `try!`——要么继续传递它们。比如说，下边栗子中的 `buyFavoriteSnack(person:vendingMachine:)`同样是一个抛出函数，任何 vend(itemNamed:)方法抛出的函数都会向上传递给调用 `buyFavoriteSnack(person:vendingMachine:)`函数的地方。

```

1 letfavoriteSnacks=[
2     "Alice":"Chips",
3     "Bob":"Licorice",
4     "Eve":"Pretzels",
5 ]
6 funcbuyFavoriteSnack(person:String,vendingMachine:VendingMachine)throws{
7     letsnackName=favoriteSnacks[person]?"Candy Bar"
8     tryvendingMachine.vend(itemNamed:snackName)
9 }
10 // Dispensing Chips

```

在这个栗子中， `buyFavoriteSnack(person:vendingMachine:)`函数查找给定人的最爱零食并且尝试通过调用 vend(itemNamed:)方法来购买它们。由于 vend(itemNamed:) 方法会抛出错误，调用的时候要在前边用 try关键字。

使用 Do-Catch 处理错误

使用 do-catch语句来通过运行一段代码处理错误。如果do分句中抛出了一个错误，它就会与 catch分句匹配，以确定其中之一可以处理错误。

这是 do-catch语句的通常使用姿势：

```
1 do{
2   tryexpression
3   statements
4 }catchpattern1{
5   statements
6 }catchpattern2wherecondition{
7   statements
8 }
```

在 catch后写一个模式来明确分句可以处理哪个错误。如果一个 catch分句没有模式，这个分句就可以匹配所有错误并且绑定这个错误到本地常量 error上。更多关于模式匹配的信息，见模式。

catch分句没有处理 do分句可能抛出的所有错误。如果没有 catch分句能处理这个错误，那错误就会传递到周围的生效范围当中。总之，错误总得在周围某个范围内被处理。举例来说，接下来的代码处理了 VendingMachineError枚举里的所有三个错误，但其他所有错误得通过范围内其他代码处理：

```
1 varvendingMachine=VendingMachine()
2 vendingMachine.coinsDeposited=8
3 do{
4   trybuyFavoriteSnack("Alice",vendingMachine:vendingMachine)
5   // Enjoy delicious snack
6 }catchVendingMachineError.invalidSelection{
7   print("Invalid Selection.")
8 }catchVendingMachineError.outOfStock{
9   print("Out of Stock.")
10 }catchVendingMachineError.insufficientFunds(letcoinsNeeded){
11   print("Insufficient funds. Please insert an additional \$(coinsNeeded) coins.")
12 }
13 // prints "Insufficient funds. Please insert an additional 2 coins."
```

在上面的栗子当中，函数 buyFavoriteSnack(person:vendingMachine:)在 try表达式中被调用，因为它会抛出错误。如果抛出错误，执行会立即切换到 catch分句，它决定是否传递来继续。如果没有错误抛出， do语句中剩下的语句将被执行。

转换错误为可选项

使用 try?通过将错误转换为可选项来处理一个错误。如果一个错误在 try?表达式中抛出，则表达式的值为 nil。比如说下面的代码x和y拥有同样的值和行为：

```
1 funcsomeThrowingFunction()throws->Int{
2 // ...
3 }
4 letx=try?someThrowingFunction()
5 lety:Int?
6 do{
7 y=trysomeThrowingFunction()
8 }catch{
9 y=nil
10 }
11
12
```

如果 `someThrowingFunction()` 抛出一个错误，`x` 和 `y` 的值就是 `nil`。另一方面，`x` 和 `y` 的值是函数返回的值。注意 `x` 和 `y` 是可选的无论 `someThrowingFunction()` 返回什么类型，这里函数返回了一个整数，所以 `x` 和 `y` 是可选整数。

当你想要在同一句里处理所有错误时，使用 `try?` 能让你的错误处理代码更加简洁。比如，下边的代码使用了一些方法来获取数据，或者在所有方式都失败后返回 `nil`。

```
1 funcfetchData()->Data?{
2 ifletdata=try?fetchDataFromDisk(){returndata}
3 ifletdata=try?fetchDataFromServer(){returndata}
4 returnnil
5 }
```

取消错误传递

事实上有时你已经知道一个抛出错误或者方法不会在运行时抛出错误。在这种情况下，你可以在表达式前写 `try!` 来取消错误传递并且把调用放进不会有错误抛出的运行时断言当中。如果错误真的抛出了，你会得到一个运行时错误。

比如说，下面的代码使用了 `loadImage(_)` 函数，它在给定路径下加载图像资源，如果图像不能被加载则抛出一个错误。在这种情况下，由于图像跟着应用走，运行时不会有错误抛出，所以取消错误传递是合适的。

```
1 letphoto=try!loadImage("./Resources/John Appleseed.jpg")
```

指定清理操作

使用 `defer` 语句来在代码离开当前代码块前执行语句合集。这个语句允许你在以任何方式离开当前代码块前执行必须要的清理工作——无论是因为抛出了错误还是因为 `return` 或者 `break` 这样的语句。比如，你可以使用 `defer` 语句来保证文件描述符都关闭并且手动指定的内存到被释放。

`defer` 语句延迟执行直到当前范围退出。这个语句由 `defer` 关键字和需要稍后执行的语句组成。被延迟执行的语句可能不会包含任何会切换控制出语句的代码，比如 `break` 或 `return` 语句，或者通过抛出一个错误。延迟的操作与其指定的顺序相反执行——就是说，第一个 `defer` 语句中的代码会在第二个中代码执行完毕后执行，以此类推。

```
1 funcprocessFile(filename:String) throws{  
2     ifexists(filename){  
3         letfile=open(filename)  
4         defer{  
5             close(file)  
6         }  
7         whileletline=tryfile.readline(){  
8             // Work with the file.  
9         }  
10        // close(file) is called here, at the end of the scope.  
11    }  
12 }
```

上面的离子使用 `defer` 语句来保证 `open(_)` 函数能调用 `close(_)`。

注意

就算没有涉及错误处理代码，你也可以使用 `defer` 语句。

类型转换

 cnswift.org/type-casting

类型转换可以判断实例的类型，也可以将该实例在其所在的类层次中视为其父类或子类的实例。

Swift 中类型转换的实现为 `is` 和 `as` 操作符。这两个操作符使用了一种简单传神的方式来检查一个值的类型或将某个值转换为另一种类型。

如同协议实现的检查（此处应有链接）中描述的那样，你还可以使用类型转换来检查类型是否遵循某个协议。

为类型转换定义类层次

你可以在类及其子类层次中使用类型转换来判断特定类实例的类型并且在同一类层次中将该实例类型转换为另一个类。下面的三段代码定义了一个类层次以及一个包含了这些类实例的数组，作为类型转换的例子。

第一个代码片段定义了一个叫做 `Medialtem` 的新基类。这个类为出现在数字媒体库中的所有成员提供了基本的功能。它声明了一个 `String` 类型的 `name` 和一个叫做 `init` 的 `name` 初始化器。（这里假设所有的媒体项目，包括所有电影和音乐，都有一个名字。）

```
1 classMedialtem{  
2     varname:String  
3     init(name:String){  
4         self.name=name  
5     }  
6 }
```

下一个片段定义了两个 `Medialtem` 的子类。第一个子类，`Movie`，封装了额外的电影的信息。他在 `Medialtem` 的基础上添加了名为 `director` 的属性及其初始化器。第二个子类，`Song`，增加了名为 `artist` 的属性及其初始化器。

```
1 classMovie: Medialtem{  
2     vardirector:String  
3     init(name:String,director:String){  
4         self.director=director  
5         super.init(name:name)  
6     }  
7 }  
8 classSong: Medialtem{  
9     varartist:String  
10    init(name:String,artist:String){  
11        self.artist=artist  
12        super.init(name:name)  
13    }  
14 }  
15
```

最后一个代码段创建了名为 `library` 的常量数组，它包含了两个 `Movie` 实例和三个 `Song` 实例。`library` 数组的类型是在初始化时根据常量字面量推断出来的。Swift 的类型检查器能够推断 `Movie` 和 `Song` 有一个共同的父类 `Medialtem`，因此 `library` 的类型推断为

[Medialtem] :

```
1 letlibrary=[  
2   Movie(name:"Casablanca",director:"Michael Curtiz"),  
3   Song(name:"Blue Suede Shoes",artist:"Elvis Presley"),  
4   Movie(name:"Citizen Kane",director:"Orson Welles"),  
5   Song(name:"The One And Only",artist:"Chesney Hawkes"),  
6   Song(name:"Never Gonna Give You Up",artist:"Rick Astley")  
7 ]  
8 // "library" 的类型被推断为[Medialtem]
```

事实上 library 储存的项目在后台依旧是 Movie 和 Song 实例。总之，如果你遍历这个数组的内容，你取出的项目将会是 Medialtem 类型而非 Movie 或 Song 类型。为了使用他们原生的类型，你需要检查他们的类型或将他们向下转换为不同的类型，如下所述。

类型检查

使用 **类型检查操作符** (is) 来检查一个实例是否属于一个特定的子类。如果实例是该子类类型，类型检查操作符返回 true ，否则返回 false 。

下面的例子定义了两个变量， movieCount 和 songCount ，用来计算数组 library 中 Movie 和 Song 实例的个数：

```
1 varmovieCount=0  
2 varsongCount=0  
3 foriteminlibrary{  
4   ifitemisMovie{  
5     movieCount+=1  
6   }elseifitemisSong{  
7     songCount+=1  
8   }  
9 }  
10 print("Media library contains \$(movieCount) movies and \$(songCount) songs")  
11 // Prints "Media library contains 2 movies and 3 songs"  
12  
13
```

这个例子遍历了 library 数组中的每个元素。每一轮中， for-in 的循环都将 item 常量设置为数组中的下一个 Medialtem 。

如果当前 Medialtem 是 Movie 类型的实例， item isMovie 返回 true ，反之返回 false 。同样的， item isSong 检查了该对象是否为 Song 类型的实例。在 for-in 循环的最后， movieCount 和 songCount 的值就是数组中对应类型实例的数量。

向下类型转换

某个类类型的常量或变量可能实际上在后台引用自一个子类的实例。当你遇到这种情况时你可以尝试使用 **类型转换操作符** (as? 或 as!) 将它向下类型转换至其子类类型。

由于向下类型转换能失败，类型转换操作符就有了两个不同形式。条件形式， as? ，返回了一个你将要向下类型转换的值的可选项。强制形式， as! ，则将向下类型转换和强制展开结合为一个步骤。

如果你不确定你向下转换类型是否能够成功，请使用条件形式的类型转换操作符 (as?) 。使用条件形式的类型转换操作符总是返回一个可选项，如果向下转换失败，可选值为 nil 。

这允许你检查向下类型转换是否成功。

当你确信向下转换类型会成功时，使用强制形式的类型转换操作符（`as!`）。当你向下转换至一个错误的类型时，强制形式的类型转换操作符会触发一个运行错误。

下面的例子遍历了 `library` 中的每个 `MediaItem`，并打印出相应的描述信息。要这样的话，每个项目均需要被当做 `Movie` 或 `Song` 来访问，而不仅仅是 `MediaItem`。为了在描述信息中访问 `Movie` 或 `Song` 的 `director` 和 `artist` 属性，这样做是必要的。

在这个例子中，数组中每一个项目的类型可能是 `Movie` 也可能是 `Song`。你不知道遍历时项目的确切类型是什么，所以这时使用条件形式的类型转换符（`as?`）来检查遍历中每次向下类型转换：

```
1 for item in library{
2     if let movie = item as? Movie{
3         print("Movie: '\(movie.name)', dir. \(movie.director)")
4     } else if let song = item as? Song{
5         print("Song: '\(song.name)', by \(song.artist)")
6     }
7 }
8 // Movie: 'Casablanca', dir. Michael Curtiz
9 // Song: 'Blue Suede Shoes', by Elvis Presley
10 // Movie: 'Citizen Kane', dir. Orson Welles
11 // Song: 'The One And Only', by Chesney Hawkes
12 // Song: 'Never Gonna Give You Up', by Rick Astley
13
```

例子开头尝试将当前 `item` 当做 `Movie` 向下类型转换。由于 `item` 是一个 `MediaItem` 的实例，它有可能是 `Movie` 类型；同样的，也有可能是 `Song` 或者仅仅是 `MediaItem` 基类。介于这种不确定性，类型转换符 `as?` 在向下类型转换到子类时返回了一个可选项。`item as? Movie` 的结果是 `Movie?` 类型，也就是“可选 `Movie` 类型”。

当数组中的 `Song` 实例使用向下转换至 `Movie` 类型时会失败。为了处理这种情况，上面的例子使用了可选绑定来检查可选 `Movie` 类型是否包含了一个值（或者说检查向下类型转换是否成功）。这个可选绑定写作“`if let movie = item as? Movie`”，它可以被读作：

尝试以 `Movie` 类型访问 `item`。如果成功，设置一个新的临时常量 `movie` 储存返回的可选 `Movie` 类型。

如果向下类型转换成功，`movie` 的属性将用于输出 `Movie` 实例的描述信息，包括 `director` 的名字。同理，无论是否在数组中找到 `Song`，均可以检查 `Song` 实例然后输出合适的描述（包括 `artist` 的名字）。

注意

类型转换实际上不会改变实例及修改其值。实例不会改变；它只是将它当做要转换的类型来访问。

Any 和 AnyObject 的类型转换

Swift 为不确定的类型提供了两种特殊的类型别名：

- `AnyObject` 可以表示任何类类型的实例。
- `Any` 可以表示任何类型，包括函数类型。

只有当你确切需要使用它们的功能和行为时再使用 Any 和 AnyObject。在写代码时使用更加明确的类型表达总要好一些。

这里有一个使用 Any 类型来对不同类型进行操作的例子，包含了函数类型以及非类类型。这个例子定义了一个名为 things 的数组，它用于储存 Any 类型的值：

```
1 varthings=[Any]()
2 things.append(0)
3 things.append(0.0)
4 things.append(42)
5 things.append(3.14159)
6 things.append("hello")
7 things.append((3.0,5.0))
8 things.append(Movie(name:"Ghostbusters",director:"Ivan Reitman"))
9 things.append({(name:String)->Stringin"Hello, \$(name)"})
10
```

这个 things 数组包含了两个 Int 值、两个 Double 值、一个 String 值、一个 (Double,Double) 的元组、 Movie 实例“Ghostbusters”、以及一个接收 String 值并返回 String 值的闭包表达式。

你可以在 switch 结构的 case 中使用 is 和 as 操作符找出已知 Any 或 AnyObject 类型的常量或变量的具体类型。下面的例子使用 switch 语句遍历了 things 数组并查询每一项的类型。其中几个 switch 的 case 将确定的值和确定类型的常量绑定在一起，使其值可以被输出：

```
1 forthinginthings{
2     switchthing{
3         case0asInt:
4             print("zero as an Int")
5         case0asDouble:
6             print("zero as a Double")
7         caseletsomeInt asInt:
8             print("an integer value of \$(someInt)")
9         caseletsomeDouble asDoublewheresomeDouble>0:
10            print("a positive double value of \$(someDouble)")
11        caseisDouble:
12            print("some other double value that I don't want to print" )
13        caseletsomeString asString:
14            print("a string value of \"\$(someString)\"")
15        caselet(x,y)as(Double,Double):
16            print("an (x, y) point at \$(x), \$(y)")
17        caseletmovie asMovie:
18            print("a movie called \$(movie.name), dir. \$(movie.director)")
19        caseletstringConverter as(String)->String:
20            print(stringConverter("Michael"))
21        default:
22            print("something else")
23    }
24 }
25 // zero as an Int
26 // zero as a Double
27 // an integer value of 42
28 // a positive double value of 3.14159
29 // a string value of "hello"
30 // an (x, y) point at 3.0, 5.0
31 // a movie called Ghostbusters, dir. Ivan Reitman
32 // Hello, Michael
33
```

注意

Any类型表示了任意类型的值，包括可选类型。如果你给显式声明的Any类型使用可选项，Swift 就会发出警告。如果你真心需要在Any值中使用可选项，如下所示，你可以使用as运算符来显式地转换可选项为Any。

```
1 let optionalNumber:Int?=3
2 things.append(optionalNumber)// Warning
3 things.append(optionalNumber as Any)// No warning
```

内嵌类型

 cnswift.org/nested-types

枚举通常用于实现特定类或结构体的功能。类似的，它也可以在更加复杂的类型环境中方便的定义通用类和结构体。为实现这种功能，Swift 允许你定义 **内嵌类型**，借此在支持类型的定义中嵌套枚举、类、或结构体。

若要在一种类型中嵌套另一种类型，在其支持类型的大括号内定义即可。可以根据需求多级嵌套数个类型。

内嵌类型的使用

下方的例子定义了一个名为 BlackJackCard 的结构体，模拟了21点游戏中的扑克牌。

BlackjackCard 结构体包含两个内嵌的枚举类型 Suit 和 Rank 。

在21点游戏中，Ace 可以表示一或十一两个值，这通过 Rank 枚举中内嵌的结构体 Values 决定：

```
1 structBlackjackCard{  
2 // nested Suit enumeration  
3 enumSuit: Character{  
4 caseSpades="♠",Hearts="♥",Diamonds="♦",Clubs="♣"  
5 }  
6 // nested Rank enumeration  
7 enumRank: Int{  
8 caseTwo=2,Three,Four,Five,Six,Seven,Eight,Nine,Ten  
9 caseJack,Queen,King,Ace  
10 structValues{  
11 letfirst:Int,second:Int?  
12 }  
13 varvalues: Values{  
14 switchself{  
15 case.Ace:  
16 returnValues(first:1,second:11)  
17 case.Jack,, Queen,, King:  
18 returnValues(first:10,second:nil)  
19 default:  
20 returnValues(first:self.rawValue,second:nil)  
21 }  
22 }  
23 }  
24 // BlackjackCard properties and methods  
25 lerank:Rank,suit:Suit  
26 vardescription: String{  
27 varoutput="suit is \(suit.rawValue),"  
28 output+=" value is \(rank.values.first)"  
29 ifletsecond=rank.values.second{  
30 output+=" or \(second)"  
31 }  
32 returnoutput  
33 }  
34 }  
35  
36  
37
```

Suit 枚举用于描述扑克牌的四种花色，并用原始值 Character 来代表各自的花色。

Rank 枚举用于描述扑克牌可能出现的十三种点数，并用原始值 Int 来代表各自的点数值（这里的 Int 并不会用于 J、Q、K、Ace 的表示）。

如上所述，Rank 枚举中定义了一个内嵌结构体 Values。这个结构体封装了大多牌只有一个值，而 Ace 可以有两个值这一事实。Values 结构体定义了两个属性来表示这些：

- Int 类型的 first
- Int? 类型的 second，或者说“可选 Int”

Rank 还定义了一个计算属性，values，它用于返回 Values 结构体的实例。这个计算属性会根据牌的点数，用适当的值初始化新的 Values 实例。对于 Jack、Queen、King、和 Ace 使用特殊的值。而对于数值的牌，则使用它本身的 Int 原始值。

BlackjackCard 结构体本身有两个属性——rank 和 suit。还定义了一个名为 description 的计算属性，用 rank 和 suit 储存的值构建对扑克牌花色和值的描述。description 属性使用可选绑定来检查是否有第二个值要描述，若有，则添加对第二个值的描述。

由于 BlackjackCard 是一个没有自定义初始化器的结构体，如结构体类型的成员初始化器所述，它有一个隐式的成员初始化器。你可以使用这个初始化器去初始化新的常量 theAceOfSpades：

```
1 lettheAceOfSpades=BlackjackCard(rank:.Ace,suit:.Spades)
2 print("theAceOfSpades: \(theAceOfSpades.description)")
3 // Prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

尽管 Rank 和 Suit 被嵌套在 BlackjackCard 中，但其类型仍可从上下文中推断出来，因此，该实例的初始化器可以单独通过成员名称（.Ace 和 .Spades）引用枚举类型。在上面的例子中，description 属性正确的反馈了黑桃 Ace 拥有 1 或 11 两个值。

引用内嵌类型

要在定义外部使用内嵌类型，只需在其前缀加上内嵌了它的类的类型名即可：

```
1 letheartsSymbol=BlackjackCard.Suit.Hearts.rawValue
2 // heartsSymbol is "♥"
```

对于上面的栗子来说，可以使 Suit、Rank 和 Values 的名字尽可能的短，因为它们的名字由定义时的上下文自然限定。

扩展

 cn.swift.org/extensions

扩展为现有的类、结构体、枚举类型、或协议添加了新功能。这也包括了为无访问权限的源代码扩展类型的能力（即所谓的逆向建模）。扩展和 Objective-C 中的分类类似。（与 Objective-C 的分类不同的是，Swift 的扩展没有名字。）

Swift 中的扩展可以：

- 添加计算实例属性和计算类型属性；
- 定义实例方法和类型方法；
- 提供新初始化器；
- 定义下标；
- 定义和使用新内嵌类型；
- 使现有的类型遵循某协议

在 Swift 中，你甚至可以扩展一个协议，以提供其要求的实现或添加符合类型的附加功能。
详见协议扩展。

注意

扩展可以向一个类型添加新的方法，但是不能重写已有的方法。

扩展的语法

使用 `extension` 关键字来声明扩展：

```
1 extension SomeType{  
2     // new functionality to add to SomeType goes here  
3 }
```

扩展可以使已有的类型遵循一个或多个协议。在这种情况下，协议名的书写方式与类或结构体完全一样：

```
1 extension SomeType: SomeProtocol, AnotherProtocol{  
2     // implementation of protocol requirements goes here  
3 }
```

用这种方式添加协议一致性详见在扩展里添加协议遵循。

如同扩展一个泛型类型中描述的那样，扩展可以用于丰富现有泛型类型。如同带有泛型 Where 分句的扩展中描述的那样，你也可以可选地给泛型添加功能。

注意

如果你向已存在的类型添加新功能，新功能会在该类型的所有实例中可用，即使实例在该扩展定义之前就已经创建。

计算属性

扩展可以向已有的类型添加计算实例属性和计算类型属性。下面的例子向 Swift 内建的 Double 类型添加了五个计算实例属性，以提供对距离单位的基本支持：

```
1 extension Double{  
2     var km: Double{return self*1_000.0}  
3     var m: Double{return self}  
4     var cm: Double{return self/100.0}  
5     var mm: Double{return self/1_000.0}  
6     var ft: Double{return self/3.28084}  
7 }  
8 let oneInch=25.4.mm  
9 print("One inch is \(oneInch) meters")  
10 // Prints "One inch is 0.0254 meters"  
11 let threeFeet=3.ft  
12 print("Three feet is \(threeFeet) meters")  
13 // Prints "Three feet is 0.914399970739201 meters"
```

这些计算属性表述了 Double 值应被看作是确定的长度单位。尽管它们被实现为计算属性，这些属性的名字仍可使用点符号添加在浮点型的字面量之后，作为一种使用该字面量来执行距离转换的方法。

在这个例子中，一个 1.0 的 Double 值表示“一米”。这就是为什么 m 计算属性要返回 self —— 表达式 1.m 表示计算 1.0 的 Double 值。

其他的单位则在以米作为计量值的基础上加以转换表示。一千米表示1000米，所以 km 计算属性将值乘 1_000.00 以用米来表示。类似的，一米有3.28084英尺，所以 ft 计算属性用 Double 值除以3.28084，将英尺转换为米。

上述属性为只读计算属性，为了简洁没有使用 get 关键字。他们都返回 Double 类型的值，可用于所有使用 Double 值的数学计算中：

```
1 let aMarathon=42.km+195.m  
2 print("A marathon is \(aMarathon) meters long")  
3 // Prints "A marathon is 42195.0 meters long"
```

注意

扩展可以添加新的计算属性，但是不能添加存储属性，也不能向已有的属性添加属性观察者。

初始化器

扩展可向已有的类型添加新的初始化器。这允许你扩展其他类型以使初始化器接收你的自定义类型作为形式参数，或提供该类型的原始实现中未包含的额外初始化选项。

扩展能为类添加新的便捷初始化器，但是不能为类添加指定初始化器或反初始化器。指定初始化器和反初始化器 必须由原来类的实现提供。

注意

如果你使用扩展为一个值类型添加初始化器，且该值类型为其所有储存的属性提供默认值，而又不定义任何自定义初始化器时，你可以在你扩展的初始化器中调用该类型默认的初始化器和成员初始化器。

如同在值类型的初始化器委托中所述，如果你在值类型的原始实现中写过它的初始化器了，上述规则就不再适用了。

下面的例子定义了一个自定义的 Rect 结构体用于描述几何矩形。这个例子也定义了两个辅助结构体 Size 和 Point，二者的默认值都是 0.0：

```
1 structSize{  
2     varwidth=0.0,height=0.0  
3 }  
4 structPoint{  
5     varx=0.0,y=0.0  
6 }  
7 structRect{  
8     varorigin=Point()  
9     varszie=Size()  
10 }
```

如同默认初始化器中描述的那样，由于 Rect 结构体为其所有属性提供了默认值，它将自动接收一个默认的初始化器和一个成员初始化器。这些初始化器能用于创建新的 Rect 实例：

```
1 letdefaultRect=Rect()  
2 letmemberwiseRect=Rect(origin:Point(x:2.0,y:2.0),  
3 size:Size(width:5.0,height:5.0))
```

你可以扩展 Rect 结构体以额外提供一个接收特定原点和大小的初始化器：

```
1 extensionRect{  
2     init(center:Point,size:Size){  
3         letoriginX=center.x-(size.width/2)  
4         letoriginY=center.y-(size.height/2)  
5         self.init(origin:Point(x:originX,y:originY),size:size)  
6     }  
7 }
```

这个初始化器首先基于提供的 center 点和 size 值计算合适的原点。然后初始化器调用该结构体的自动成员初始化器 init(origin:size:)，这样就将新的原点和大小值保存在了对应属性中：

```
1 letcenterRect=Rect(center:Point(x:4.0,y:4.0),  
2 size:Size(width:3.0,height:3.0))  
3 // centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

注意

如果你使用扩展提供了一个新的初始化器，你仍应确保每一个实例都在初始化完成时完全初始化。

方法

扩展可以为已有的类型添加新的实例方法和类型方法。下面的例子为 Int 类型添加了一个名为 repetitions 的新实例方法：

```
1 extensionInt{
2     func repetitions(task:@escaping ()->Void){
3         for _ in 0..
```

repetitions(task:) 方法接收一个 ()->Void 类型的单一实际参数，它表示一个没有参数且无返回值的函数。

在这个扩展定义之后，你可以在任何整型数字处调用 repetitions(task:) 方法，以执行相应次数的操作：

```
1 3.repetitions{
2     print("Hello!")
3 }
4 // Hello!
5 // Hello!
6 // Hello!
```

异变实例方法

增加了扩展的实例方法仍可以修改（或异变）实例本身。结构体和枚举类型方法在修改 self 或本身的属性时必须标记实例方法为 mutating，和原本实现的异变方法一样。

下面的例子为 Swift 的 Int 类型添加了一个新的异变方法 square，以表示原值的平方：

```
1 extensionInt{
2     mutating func square(){
3         self = self * self
4     }
5 }
6 var someInt = 3
7 someInt.square()
8 // someInt is now 9
```

下标

扩展能为已有的类型添加新的下标。下面的例子为 Swift 内建的 Int 类型添加了一个整型下标。这个下标 [n] 返回了从右开始第 n 位的十进制数字：

- 123456789[0] 返回 9
- 123456789[1] 返回 8

.....以此类推：

```
1 extension Int{
2     subscript(digitIndex:Int) -> Int{
3         var decimalBase = 1
4         for _ in 0..
```

若 Int 值没有所需索引的那么多数字，下标实现返回 0，就像是这个数左边用零填充：

```
1 746381295[9]
2 // returns 0, as if you had requested:
3 0746381295[9]
```

内嵌类型

扩展可以为已有的类、结构体和枚举类型添加新的内嵌类型：

```
1 extension Int{
2     enum Kind {
3         case negative, zero, positive
4     }
5     var kind: Kind {
6         switch self {
7             case 0:
8                 return .zero
9             case let x where x > 0:
10                return .positive
11            default:
12                return .negative
13        }
14    }
15 }
```

这个例子为 Int 添加了新的内嵌枚举类型。这个名为 Kind 的枚举类型表示一个特定整数的类型。具体表示了这个数字是负数、零还是正数。

这个例还向 Int 中添加了新的计算实例属性 kind，以返回该整数的合适 Kind 枚举情况。

这个内嵌的枚举类型可以和任意 Int 一起使用：

```
1 funcprintIntegerKinds(_numbers:[Int]){
2     for number in numbers{
3         switch number.kind{
4             case .negative:
5                 print("- ", terminator: "")
6             case .zero:
7                 print("0 ", terminator: "")
8             case .positive:
9                 print("+ ", terminator: "")
10        }
11    }
12    print("")
13 }
14 printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
15 // Prints "+ + - 0 - 0 + "
```

这里 `printIntegerKinds(_:)` 函数接收一个 `Int` 的数组并对这些值进行遍历。对数组的每一个数字，函数考虑这个整数的 `kind` 计算属性，并输出合适的描述。

注意

已知 `number.kind` 是 `Int.Kind` 类型。因此，`switch` 语句中的所有 `Int.Kind` 情况值都可以简写，例如用 `.Negative` 而不是 `Int.Kind.Negative`。

协议

 cnswift.org/protocols

协认为方法、属性、以及其他特定的任务需求或功能定义蓝图。协议可被类、结构体、或枚举类型采纳以提供所需功能的具体实现。满足了协议中需求的任意类型都叫做遵循了该协议。

除了指定遵循类型必须实现的要求外，你可以扩展一个协议以实现其中的一些需求或实现一个符合类型的可以利用的附加功能。

协议的语法

定义协议的方式与类、结构体、枚举类型非常相似：

```
1 protocolSomeProtocol{  
2 // protocol definition goes here  
3 }
```

在自定义类型声明时，将协议名放在类型名的冒号之后来表示该类型采纳一个特定的协议。多个协议可以用逗号分开列出：

```
1 structSomeStructure:FirstProtocol,AnotherProtocol{  
2 // structure definition goes here  
3 }
```

若一个类拥有父类，将这个父类名放在其采纳的协议名之前，并用逗号分隔：

```
1 classSomeClass:SomeSuperclass,FirstProtocol,AnotherProtocol{  
2 // class definition goes here  
3 }
```

属性要求

协议可以要求所有遵循该协议的类型提供特定名字和类型的实例属性或类型属性。协议并不会具体说明属性是储存型属性还是计算型属性——它只具体要求属性有特定的名称和类型。协议同时要求一个属性必须明确是可读的或可读的和可写的。

若协议要求一个属性为可读和可写的，那么该属性要求不能用常量存储属性或只读计算属性来满足。若协议只要求属性为可读的，那么任何种类的属性都能满足这个要求，而且如果你的代码需要的话，该属性也可以是可写的。

属性要求定义为变量属性，在名称前面使用 var 关键字。可读写的属性使用 {getset} 来写在声明后面来明确，使用 {get} 来明确可读的属性。

```
1 protocolSomeProtocol{  
2 varmustBeSettable: Int{getset}  
3 vardoesNotNeedToBeSettable: Int{get}  
4 }
```

在协议中定义类型属性时在前面添加 static 关键字。当类的实现使用 class 或 static 关键字前缀声明类型属性要求时，这个规则仍然适用：

```
1 protocolAnotherProtocol{  
2     staticvarsomeTypeProperty: Int{getset}  
3 }
```

这里是一个只有一个实例属性要求的协议：

```
1 protocolFullyNamed{  
2     varfullName: String{get}  
3 }
```

上面 FullyNamed 协议要求遵循的类型提供一个完全符合的名字。这个协议并未指定遵循类型的其他任何性质——它只要求这个属性必须为其自身提供一个全名。协议申明了所有 FullyNamed 类型必须有一个可读实例属性 fullName，且为 String 类型。

这里是一个采纳并遵循 FullyNamed 协议的结构体的例子：

```
1 structPerson: FullyNamed{  
2     varfullName:String  
3 }  
4 letjohn=Person(fullName:"John Appleseed")  
5 // john.fullName is "John Appleseed"
```

这个例子定义了一个名为 Person 的结构体，它表示一个有名字的人。它在其第一行定义中表明了它采纳 FullyNamed 协议作为它自身的一部分。

每一个 Person 的实例都有一个名为 fullName 的 String 类型储存属性。这符合了 FullyNamed 协议的单一要求，并且表示 Person 已经正确地遵循了该协议。（若协议的要求没有完全达标，Swift 在编译时会报错。）

这里是一个更加复杂的类，采纳并遵循了 FullyNamed 协议：

```
1 classStarship: FullyNamed{  
2     varprefix:String?  
3     varname:String  
4     init(name:String,prefix:String?=nil){  
5         self.name=name  
6         self.prefix=prefix  
7     }  
8     varfullName: String{  
9         return(prefix!=nil?prefix!+" "+name:  
10    )  
11    }  
12    varncc1701=Starship(name:"Enterprise",prefix:"USS")  
13    // ncc1701.fullName is "USS Enterprise"
```

这个类为一艘星舰实现了 fullName 计算型只读属性的要求。每一个 Starship 类的实例储存了一个不可选的 name 属性以及一个可选的 prefix 属性。当 prefix 值存在时，fullName 将 prefix 放在 name 之前以创建星舰的全名。

方法要求

协议可以要求采纳的类型实现指定的实例方法和类方法。这些方法作为协议定义的一部分，书写方式与正常实例和类方法的方式完全相同，但是不需要大括号和方法的主体。允许变量拥有参数，与正常的方法使用同样的规则。但在协议的定义中，方法参数不能定义默认值。

正如类型属性要求的那样，当协议中定义类型方法时，你总要在其之前添加 `static` 关键字。即使在类实现时，类型方法要求使用 `class` 或 `static` 作为关键字前缀，前面的规则仍然适用：

```
1 protocolSomeProtocol{  
2     staticfuncsomeTypeMethod()  
3 }
```

下面的例子定义了一个只有一个实例方法要求的协议：

```
1 protocolRandomNumberGenerator{  
2     funcrandom()->Double  
3 }
```

这里 `RandomNumberGenerator` 协议要求所有采用该协议的类型都必须有一个实例方法 `random`，而且要返回一个 `Double` 的值，无论这个值叫什么。尽管协议没有明确定义，这里默认这个值在 0.0 到 1.0（不包括）之间。

`RandomNumberGenerator` 协议并不为随机值的生成过程做任何定义，它只要求生成器提供一个生成随机数的标准过程。

这里有一个采用并遵循 `RandomNumberGenerator` 协议的类的实现。这个类实现了著名的 *linear congruential generator* 伪随机数发生器算法：

```
1 classLinearCongruentialGenerator: RandomNumberGenerator{  
2     varlastRandom=42.0  
3     letm=139968.0  
4     leta=3877.0  
5     letc=29573.0  
6     funcrandom()->Double{  
7         lastRandom=((lastRandom*a+c).truncatingRemainder(dividingBy:m))  
8         returnlastRandom/m  
9     }  
10 }  
11 letgenerator=LinearCongruentialGenerator()  
12 print("Here's a random number: \(generator.random())")  
13 // Prints "Here's a random number: 0.37464991998171"  
14 print("And another one: \(generator.random())")  
15 // Prints "And another one: 0.729023776863283"
```

异变方法要求

有时一个方法需要改变（或异变）其所属的实例。例如值类型的实例方法（即结构体或枚举），在方法的 `func` 关键字之前使用 `mutating` 关键字，来表示在该方法可以改变其所属的实例，以及该实例的所有属性。这一过程写在了在实例方法中修改值类型中。

若你定义了一个协议的实例方法需求，想要异变任何采用了该协议的类型实例，只需在协议里方法的定义当中使用 `mutating` 关键字。这允许结构体和枚举类型能采用相应协议并满足方法要求。

注意

如果你在协议中标记实例方法需求为 mutating，在为类实现该方法的时候不需要写 mutating 关键字。mutating 关键字只在结构体和枚举类型中需要书写。

下面的例子定义了一个名为 Toggable 的协议，协议只定义了一个实例方法要求叫做 toggle。顾名思义，toggle() 方法将切换或转换任何遵循该协议的类型的状态，典型地，修改该类型的属性。

在 Toggable 协议的定义中，toggle() 方法使用 mutating 关键字标记，来表明该方法在调用时会改变遵循该协议的实例的状态：

```
1 protocolToggable{  
2   mutatingfunctoggle()  
3 }
```

若使用结构体或枚举实现 Toggable 协议，这个结构体或枚举可以通过使用 mutating 标记这个 toggle() 方法，来保证该实现符合协议要求。

下面的例子定义了一个名为 OnOffSwitch 的枚举。这个枚举在两种状态间改变，即枚举成员 On 和 Off。该枚举的 toggle 实现使用了 mutating 关键字，以满足 Toggable 协议需求：

```
1 enumOnOffSwitch: Toggable{  
2   caseoff,on  
3   mutatingfunctoggle(){  
4     switchself{  
5       case.off:  
6         self=.on  
7       case.on:  
8         self=.off  
9     }  
10   }  
11 }  
12 varlightSwitch=OnOffSwitch.off  
13 lightSwitch.toggle()  
14 // lightSwitch is now equal to .on
```

初始化器要求

协议可以要求遵循协议的类型实现指定的初始化器。和一般的初始化器一样，只用将初始化器写在协议的定义当中，只是不用写大括号也就是初始化器的实体：

```
1 protocolSomeProtocol{  
2   init(someParameter:Int)  
3 }
```

协议初始化器要求的类实现

你可以通过实现指定初始化器或便捷初始化器来使遵循该协议的类满足协议的初始化器要求。在这两种情况下，你都必须使用 required 关键字修饰初始化器的实现：

```
1 classSomeClass: SomeProtocol{  
2   required init(someParameter:Int){  
3     // initializer implementation goes here  
4   }  
5 }
```

在遵循协议的类的所有子类中，`required` 修饰的使用保证了你为协议初始化器要求提供了一个明确的继承实现。

详见必要初始化器。

注意

由于 `final` 的类不会有子类，如果协议初始化器实现的类使用了 `final` 标记，你就不需要使用 `required` 来修饰了。因为这样的类不能被继承子类。详见阻止重写了解更多 `final` 修饰符的信息。

如果一个子类重写了父类指定的初始化器，并且遵循协议实现了初始化器要求，那么就要为这个初始化器的实现添加 `required` 和 `override` 两个修饰符：

```
1 protocolSomeProtocol{
2     init()
3 }
4 classSomeSuperClass{
5     init(){
6         // initializer implementation goes here
7     }
8 }
9 classSomeSubClass:SomeSuperClass,SomeProtocol{
10    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
11    required override init(){
12        // initializer implementation goes here
13    }
14 }
15
16
```

可失败初始化器要求

如同可失败初始化器定义的一样，协议可以为遵循该协议的类型定义可失败的初始化器。

遵循协议的类型可以使用一个可失败的或不可失败的初始化器满足一个可失败的初始化器要求。不可失败初始化器要求可以使用一个不可失败初始化器或隐式展开的可失败初始化器满足。

将协议作为类型

实际上协议自身并不实现功能。不过你创建的任意协议都可以变为一个功能完备的类型在代码中使用。

由于它是一个类型，你可以在很多其他类型可以使用的地方使用协议，包括：

- 在函数、方法或者初始化器里作为形式参数类型或者返回类型；
- 作为常量、变量或者属性的类型；
- 作为数组、字典或者其他存储器的元素的类型。

注意

由于协议是类型，要开头大写（比如说 `FullyNamed` 和 `RandomNumberGenerator`）来匹配 Swift 里其他类型名称格式（比如说 `Int`、`String` 还有 `Double`）。

这里有一个把协议用作类型的例子：

```
1 class Dice{  
2     let sides:Int  
3     let generator:RandomNumberGenerator  
4     init(sides:Int,generator:RandomNumberGenerator){  
5         self.sides=sides  
6         self.generator=generator  
7     }  
8     func roll()->Int{  
9         return Int(generator.random()*Double(sides))+1  
10    }  
11 }
```

这个例子定义了一个叫做 `Dice` 的新类，它表示一个用于棋盘游戏的 n 面骰子。 `Dice` 实例有一个叫做 `sides` 的整数属性，它表示了骰子有多少个面，还有个叫做 `generator` 的属性，它提供了随机数的生成器来生成骰子的值。

`generator` 属性是 `RandomNumberGenerator` 类型。因此，你可以把它放到任何采纳了 `RandomNumberGenerator` 协议的类型的实例里。除了这个实例必须采纳 `RandomNumberGenerator` 协议以外，没有其他任何要求了。

`Dice` 也有一个初始化器，来设置初始状态。这个初始化器有一个形式参数叫做 `generator`，它同样也是 `RandomNumberGenerator` 类型。你可以在初始化新的 `Dice` 实例的时候传入一个任意遵循这个协议的类型的值到这个形式参数里。

`Dice` 提供了一个形式参数方法，`roll`，它返回一个介于 1 和骰子面数之间的整数值。这个方法调用生成器的 `random()` 方法来创建一个新的介于 0.0 和 1.0 之间的随机数，然后使用这个随机数来在正确的范围创建一个骰子的值。由于 `generator` 已知采纳了 `RandomNumberGenerator`，它保证了会有 `random()` 方法以供调用。

这里是 `Dice` 类使用 `LinearCongruentialGenerator` 实例作为用于创建一个六面骰子的随机数生成器来创建一个六面骰子的过程：

```
1 var d6=Dice(sides:6,generator:LinearCongruentialGenerator())  
2 for _ in 1...5{  
3     print("Random dice roll is \(d6.roll())")  
4 }  
5 // Random dice roll is 3  
6 // Random dice roll is 5  
7 // Random dice roll is 4  
8 // Random dice roll is 5  
9 // Random dice roll is 4
```

委托

委托^[1]是一个允许类或者结构体放手（或者说委托）它们自身的某些责任给另外类型实例的设计模式。这个设计模式通过定义一个封装了委托责任的协议来实现，比如遵循了协议的类型（所谓的委托）来保证提供被委托的功能。委托可以用来响应一个特定的行为，或者从外部资源取回数据而不需要了解资源具体的类型。

译注

[1] , Delegation 委托 , 可能也以“代理”而为人熟知 , 这里我们选择译为“委托”是为了更好的理解避免混淆。

下面的例子定义了两个协议以用于基于骰子的棋盘游戏 :

```
1 protocolDiceGame{
2     var dice: Dice{get}
3     func play()
4 }
5 protocol DiceGameDelegate{
6     func gameDidStart(_ game: DiceGame)
7     func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
8     func gameDidEnd(_ game: DiceGame)
9 }
```

DiceGame 协议是一个给任何与骰子有关的游戏采纳的协议。 DiceGameDelegate 协议可以被任何追踪 DiceGame 进度的类型采纳。

这里有一个原本在控制流中介绍的蛇与梯子游戏的一个版本。这个版本采纳了协议以使用 Dice 实例来让它使用骰子 ; 采用 DiceGame 协议 ; 然后通知一个 DiceGameDelegate 关于进度的信息 :

```
1 class SnakesAndLadders: DiceGame{
2     let finalSquare = 25
3     let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
4     var square = 0
5     var board: [Int]
6     init() {
7         board = Array(repeating: 0, count: finalSquare + 1)
8         board[0] = +08; board[6] = +11; board[9] = +09; board[10] = +02
9         board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
10    }
11    var delegate: DiceGameDelegate?
12    func play() {
13        square = 0
14        delegate?.gameDidStart(self)
15        gameLoop: while square != finalSquare {
16            let diceRoll = dice.roll()
17            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
18            switch square + diceRoll {
19                case finalSquare:
20                    break gameLoop
21                case let newSquare where newSquare > finalSquare:
22                    continue gameLoop
23                default:
24                    square += diceRoll
25                    square += board[square]
26            }
27        }
28        delegate?.gameDidEnd(self)
29    }
30 }
```

要找到蛇与梯子游戏的介绍 , 查看控制流章节的Break小节。

这个版本的游戏使用了叫做 SnakesAndLadders 类包装 , 它采纳了 DiceGame 协议。它提供了可读的 dice 属性和一个 play() 方法来遵循协议。 (dice 属性声明为常量属性是因为它不需要在初始化后再改变了 , 而且协议只需要它是可读的。)

蛇与梯子游戏棋盘设置都写在了类的 `init()` 初始化器中。所有的游戏逻辑都移动到了协议的 `play` 方法里，它使用了协议要求的 `dice` 属性来提供它的骰子值。

注意 `delegate` 属性被定义为可选的 `DiceGameDelegate`，是因为玩游戏并不是必须要有委托。由于它是一个可选类型，`delegate` 属性自动地初始化为 `nil`。此后，游戏的实例化者可以选择给属性赋值一个合适的委托。

`DiceGameDelegate` 提供了三个追踪游戏进度的方法。这三个方法在游戏逻辑的 `play()` 方法中协作，并且在游戏开始时调用，新一局开始，或者游戏结束。

由于 `delegate` 属性是可选的 `DiceGameDelegate`，`play()` 方法在每次调用委托的时候都使用可选链。如果 `delegate` 属性是空的，这些委托调用会优雅地失败并且没有错误。如果 `delegate` 属性非空，委托的方法就被调用了，并且把 `SnakesAndLadders` 实例作为形式参数传入。

接下来的例子展示了叫做 `DiceGameTracker` 的类，它次那了 `DiceGameDelegate` 协议：

```
1 class DiceGameTracker: DiceGameDelegate{
2     var numberOfTurns=0
3     func gameDidStart(_ game: DiceGame){
4         numberOfTurns=0
5         if game is SnakesAndLadders{
6             print("Started a new game of Snakes and Ladders")
7         }
8         print("The game is using a \(game.dice.sides)-sided dice")
9     }
10    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int){
11        numberOfTurns+=1
12        print("Rolled a \(diceRoll)")
13    }
14    func gameDidEnd(_ game: DiceGame){
15        print("The game lasted for \(numberOfTurns) turns")
16    }
17 }
```

`DiceGameTracker` 实现了 `DiceGameDelegate` 要求的所有方法。它使用这些方法来对游戏开了多少局保持追踪。它在游戏开始的时候重置 `numberOfTurns` 属性为零，在每次新一轮游戏开始的时候增加，并且一旦游戏结束，打印出游戏一共开了多少轮。

上边展示的 `gameDidStart(_)` 的实现使用了 `game` 形式参数来打印某些关于游戏的信息。`game` 形式参数是 `DiceGame` 类型，不是 `SnakesAndLadders`，所以 `gameDidStart(_)` 只能访问和使用 `DiceGame` 协议实现的那部分方法和属性。总之，转换类型之后方法还是可以使用的。在这个例子中，它检查 `game` 在后台是否就是 `SnakesAndLadders` 实例，如果是，打印合适的信息。

`gameDidStart(_)` 方法同样访问传入的 `game` 形式参数里的 `dice` 属性。由于 `game` 已经遵循 `DiceGame` 协议，这就保证了 `dice` 属性的存在，并且 `gameDidStart(_)` 方法能够访问和打印骰子的 `sides` 属性，无论玩的是什么类型的游戏。

这里是 `DiceGameTracker` 的运行结果：

```
1 lettracker=DiceGameTracker()
2 letgame=SnakesAndLadders()
3 game.delegate=tracker
4 game.play()
5 // Started a new game of Snakes and Ladders
6 // The game is using a 6-sided dice
7 // Rolled a 3
8 // Rolled a 5
9 // Rolled a 4
10 // Rolled a 5
11 // The game lasted for 4 turns
```

在扩展里添加协议遵循

你可以扩展一个已经存在的类型来采纳和遵循一个新的协议，就算是你无法访问现有类型的源代码也行。扩展可以添加新的属性、方法和下标到已经存在的类型，并且因此允许你添加协议需要的任何需要。要了解更多关于扩展的信息，见[扩展](#)。

注意

类型已经存在的实例会在给它的类型扩展中添加遵循协议时自动地采纳和遵循这个协议。

举例来说，这个协议，叫做 `TextRepresentable`，可以被任何可以用文本表达的类型实现。这可能是它自身的描述，或者是它当前状态的文字版显示：

```
1 protocolTextRepresentable{
2     var textualDescription: String{get}
3 }
```

先前的 `Dice` 类可以扩展来采纳和遵循 `TextRepresentable`：

```
1 extension Dice: TextRepresentable{
2     var textualDescription: String{
3         return "A \$(sides)-sided dice"
4     }
5 }
```

这个扩展使用了与 `Dice` 提供它原本实现完全相同的方式来采纳了新的协议。协议名写在类型的名字之后，用冒号隔开，并且在扩展的花括号里实现了所有协议的需要。

任何 `Dice` 实例现在都可以被视作 `TextRepresentable`：

```
1 let d12=Dice(sides:12,generator:LinearCongruentialGenerator())
2 print(d12.textualDescription)
3 // Prints "A 12-sided dice"
```

类似地，`SnakesAndLadders` 游戏类可以扩展来采纳和遵循 `TextRepresentable` 协议：

```
1 extension SnakesAndLadders: TextRepresentable{
2     var textualDescription: String{
3         return "A game of Snakes and Ladders with \$(finalSquare) squares"
4     }
5 }
6 print(game.textualDescription)
7 // Prints "A game of Snakes and Ladders with 25 squares"
```

有条件地遵循协议

泛型类型可能只在某些情况下满足一个协议的要求，比如当类型的泛型形式参数遵循对应协议时。你可以通过在扩展类型时列出限制让泛型类型有条件地遵循某协议。在你采纳协议的名字后面写泛型 where 分句。更多关于泛型 where 分句，见泛型Where分句。

下面的扩展让 Array 类型在存储遵循 TextRepresentable 协议的元素时遵循 TextRepresentable 协议。

```
1 extensionArray:TextRepresentable whereElement: TextRepresentable{
2     var textualDescription: String{
3         let itemsAsText=self.map{$0.textualDescription}
4         return "["+itemsAsText.joined(separator:", ")+"]"
5     }
6 }
7 let myDice=[d6,d12]
8 print(myDice.textualDescription)
9 // Prints "[A 6-sided dice, A 12-sided dice]"
```

使用扩展声明采纳协议

如果一个类型已经遵循了协议的所有需求，但是还没有声明它采纳了这个协议，你可以让通过一个空的扩展来让它采纳这个协议：

```
1 struct Hamster{
2     var name:String
3     var textualDescription: String{
4         return "A hamster named \(name)"
5     }
6 }
7 extension Hamster: TextRepresentable{}
```

Hamster 实例现在可以用在任何 TextRepresentable 类型可用的地方了：

```
1 let simonTheHamster=Hamster(name:"Simon")
2 let somethingTextRepresentable:TextRepresentable=simonTheHamster
3 print(somethingTextRepresentable.textualDescription)
4 // Prints "A hamster named Simon"
```

协议类型的集合

协议可以用作储存在集合比如数组或者字典中的类型，如同在协议作为类型（此处应有链接）。这个例子创建了一个 TextRepresentable 实例的数组：

```
1 let things:[TextRepresentable]=[game,d12,simonTheHamster]
```

现在可以遍历数组中的元素了，并且打印每一个元素的文本化描述：

```
1 for thing in things{
2     print(thing.textualDescription)
3 }
4 // A game of Snakes and Ladders with 25 squares
5 // A 12-sided dice
6 // A hamster named Simon
```

注意 thing 常量是 TextRepresentable 类型。它不是 Dice 类型，抑或 DiceGame 还是 Hamster ，就算后台实际类型是它们之一。总之，由于它是 TextRepresentable ，并且 TextRepresentable 唯一已知的信息就是包含了 textualDescription 属性，所以每次循环来访问 thing.textualDescription 是安全的。

协议继承

协议可以继承一个或者多个其他协议并且可以在它继承的基础之上添加更多要求。协议继承的语法与类继承的语法相似，只不过可以选择列出多个继承的协议，使用逗号分隔：

```
1 protocolInheritingProtocol:SomeProtocol,AnotherProtocol{
2 // protocol definition goes here
3 }
```

这里是一个继承了上边 TextRepresentable 协议的例子：

```
1 protocolPrettyTextRepresentable: TextRepresentable{
2 varprettyTextualDescription: String{get}
3 }
```

这个例子定义了一个新的协议 PrettyTextRepresentable ，它继承自 TextRepresentable 。任何采用了 PrettyTextRepresentable 的类型都必须满足所有 TextRepresentable 强制的需要，另外还有 PrettyTextRepresentable 强制的要求。在这个例子中， PrettyTextRepresentable 添加了一个叫做 prettyTextualDescription 的可读属性，它返回一个 String 。

SnakesAndLadders 类可以通过扩展来采纳和遵循 PrettyTextRepresentable :

```
1 extensionSnakesAndLadders: PrettyTextRepresentable{
2 varprettyTextualDescription: String{
3 varoutput=textualDescription+":\n"
4 forindex in1...finalSquare{
5 switchboard[index]{
6 caseletladder whereladder>0:
7 output+="▲ "
8 caseletsnake wheresnake<0:
9 output+="▼ "
10 default:
11 output+="○ "
12 }
13 }
14 returnoutput
15 }
16 }
```

这个扩展声明了它采纳了 PrettyTextRepresentable 协议并且为 SnakesAndLadders 类提供了 prettyTextualDescription 属性的实现。任何 PrettyTextRepresentable 必须同时是 TextRepresentable ，并且 prettyTextualDescription 起始于访问 TextRepresentable 协议的 textualDescription 属性来开始输出字符串。它追加了一个冒号和一个换行符，并且使用这个作为友好文本输出的开始。它随后遍历棋盘数组，追加几何图形来表示每个方格的内容：

- 如果方格的值大于 0 ，它是梯子的底部，就表示为 ▲ ；
- 如果方格的值小于 0 ，它是蛇的头部，就表示为 ▼ ；
- 否则，方格的值为 0 ，是“自由”方格，表示为 ○ 。

现在 PrettyTextRepresentable 属性可以用来输出任何 SnakesAndLadders 实例的友好文本描述了：

```
1 print(game.prettyTextualDescription)
2 // A game of Snakes and Ladders with 25 squares:
3 // ○○▲○○▲○○▲▲○○○▼○○○○▼○○▼○▼○
```

类专用的协议

通过添加 AnyObject 关键字到协议的继承列表，你就可以限制协议只能被类类型采纳（并且不是结构体或者枚举）。

```
1 protocolSomeClassOnlyProtocol:AnyObject,SomeInheritedProtocol{
2 // class-only protocol definition goes here
3 }
```

在上边的例子当中， SomeClassOnlyProtocol 只能被类类型采纳。如果在结构体或者枚举中尝试采纳 SomeClassOnlyProtocol 就会出发编译时错误。

注意在协议的要求假定或需要遵循的类型拥有引用语意的时候使用类专用的协议而不是值语意。要了解更多关于引用和值语意，见[结构体和枚举是值类型和类是引用类型](#)。

协议组合

要求一个类型一次遵循多个协议是很有用的。你可以使用协议组合来复合多个协议到一个要求里。协议组合行为就和你定义的临时局部协议一样拥有构成中所有协议的需求。协议组合不定义任何新的协议类型。

协议组合使用 SomeProtocol&AnotherProtocol 的形式。你可以列举任意数量的协议，用和符号连接（&），使用逗号分隔。除了协议列表，协议组合也能包含类类型，这允许你标明一个需要的父类。

这里是一个复合两个叫做 Named 和 Aged 的协议到函数形式参数里一个协议组合要求的例子：

```
1 protocolNamed{
2 varname: String{get}
3 }
4 protocolAged{
5 varage: Int{get}
6 }
7 structPerson:Named,Aged{
8 varname:String
9 varage:Int
10 }
11 funcwishHappyBirthday(tocelebrator:Named&Aged){
12 print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
13 }
14 letbirthdayPerson=Person(name:"Malcolm",age:21)
15 wishHappyBirthday(to:birthdayPerson)
16 // Prints "Happy birthday, Malcolm, you're 21!"
```

这个例子定义了一个叫做 Named 的协议，它只有一个叫做 name 的可读 String 属性要求。它同样定义了一个叫做 Aged 的协议，只有一个叫做 age 的 Int 属性要求。两个协议都被叫做 Person 的结构体采纳。

例子中同样定义了一个叫做 wishHappyBirthday(to:) 的函数， celebrator 形式参数的类型是 Named&Aged ，这意味着“任何同时遵循 Named 和 Aged 的协议。”它不关心具体是什么样的类型传入函数，只要它遵循这两个要求的协议即可。

然后例子中又创建了一个新的叫做 birthdayPerson 的 Person 实例并且把这个新的实例传入 wishHappyBirthday(to:) 函数。由于 Person 同时遵循两个协议，所以这是合法调用，并且 wishHappyBirthday(to:) 函数能够打印出生日祝福。

这里是一个包含了先前例子中 Named 协议以及一个 Location 类的例子：

```
1 classLocation{
2     varlatitude:Double
3     varlongitude:Double
4     init(latitude:Double,longitude:Double){
5         self.latitude=latitude
6         self.longitude=longitude
7     }
8 }
9 classCity:Location,Named{
10    varname:String
11    init(name:String,latitude:Double,longitude:Double){
12        self.name=name
13        super.init(latitude:latitude,longitude:longitude)
14    }
15 }
16 funcbeginConcert(inlocation:Location&Named){
17     print("Hello, \(location.name)!")
18 }
19 letseattle=City(name:"Seattle",latitude:47.6,longitude:-122.3)
20 beginConcert(in:seattle)
21 // Prints "Hello, Seattle!"
22
```

函数 beginConcert(in:) 接收一个 Location&Named 类型的形式参数，也就是说任何 Location 的子类且遵循 Named 协议的类型。具体到这里， City 同事满足两者需求。

如果你尝试过传 birthdayPerson 给 beginConcert(in:) 函数，这是非法的，由于 Person 不是 Location 的子类。同理，如果你创建了一个 Location 的子类但并不遵循 Named 协议，用这个类型调用 beginConcert(in:) 也是非法的。

协议遵循的检查

你可以使用类型转换中描述的 is 和 as 运算符来检查协议遵循，还能转换为特定的协议。检查和转换协议的语法与检查和转换类型是完全一样的：

- 如果实例遵循协议 is 运算符返回 true 否则返回 false ；
- as? 版本的向下转换运算符返回协议的可选项，如果实例不遵循这个协议的话值就是 nil ；
- as! 版本的向下转换运算符强制转换协议类型并且在失败时触发运行时错误。

这个例子定义了一个叫做 HasArea 的协议，只有一个叫做 area 的可读 Double 属性要求：

```
1 protocolHasArea{  
2     vararea: Double{get}  
3 }
```

这里有两个类， Circle 和 Country ，这两个类都遵循 HasArea 协议：

```
1 classCircle: HasArea{  
2     letpi=3.1415927  
3     varradius:Double  
4     vararea: Double{returnpi*radius*radius}  
5     init(radius:Double){self.radius=radius}  
6 }  
7 classCountry: HasArea{  
8     vararea:Double  
9     init(area:Double){self.area=area}  
10 }
```

Circle 类基于存储属性 radius 用计算属性实现了 area 属性要求。 Country 类则直接使用存储属性实现了 area 要求。这两个类都正确地遵循了 HasArea 协议。

这里是一个叫做 Animal 的类，它不遵循 HasArea 协议：

```
1 classAnimal{  
2     varlegs:Int  
3     init(legs:Int){self.legs=legs}  
4 }
```

Circle 、 Country 和 Animal 类并不基于相同的基类。不过它们都是类，所以它们三个类型的实例都可以用于初始化储存类型为 AnyObject 的数组：

```
1 letobjects:[AnyObject]=[  
2     Circle(radius:2.0),  
3     Country(area:243_610),  
4     Animal(legs:4)  
5 ]
```

objects 数组使用包含 Circle 两个单位半径的实例、 Country 以平方公里为单位英国面积实例、 Animal 有四条腿实例的数组字面量初始化。

objects 数组现在可以遍历了，而且数组中每一个对象都能检查是否遵循 HasArea 协议：

```
1 forobjectinobjects{  
2     ifletobjectWithArea=objectas?HasArea{  
3         print("Area is \$(objectWithArea.area)")  
4     }else{  
5         print("Something that doesn't have an area")  
6     }  
7 }  
8 // Area is 12.5663708  
9 // Area is 243610.0  
10 // Something that doesn't have an area
```

当数组中的对象遵循 HasArea 协议， as? 运算符返回的可选项就通过可选绑定赋值到一个叫做 objectWithArea 的常量当中。 objectWithArea 已知类型为 HasArea ，所以它的 area 属性可以通过类型安全的方式访问和打印。

注意使用的对象并没有通过转换过程而改变。他们仍然是 Circle 、 Country 和 Animal 。总之，在储存在 objectWithArea 常量中的那一刻，他们仅被所知为 HasArea ，所以只有 area 属性可以访问。

可选协议要求

你可以给协议定义 可选要求，这些要求不需要强制遵循协议的类型实现。可选要求使用 optional 修饰符作为前缀放在协议的定义中。可选要求允许你的代码与 Objective-C 操作。协议和可选要求必须使用 @objc 标志标记。注意 @objc 协议只能被继承自 Objective-C 类或其他 @objc 类采纳。它们不能被结构体或者枚举采纳。

当你在可选要求中使用方法或属性是，它的类型自动变成可选项。比如说，一个 (Int)->String 类型的方法会变成 ((Int)->String)? 。注意是这个函数类型变成可选项，不是方法的返回值。

可选协议要求可以在可选链中调用，来说明要求没有被遵循协议的类型实现的概率。你可以通过在调用方法的时候在方法名后边写一个问号来检查它是否被实现，比如 someOptionalMethod?(someArgument) 。更多关于可选链的信息，见[可选链](#)。

下面的例子定义了一个叫做 Counter 的整数计数的类，它使用一个外部数据源来提供它的增量。这个数据源通过 CounterDataSource 协议定义，它有两个可选要求：

```
1 @objcprotocolCounterDataSource{
2     @objc optional funcincrement(forCount count:Int)->Int
3     @objc optional varfixedIncrement: Int{get}
4 }
```

CounterDataSource 协议定义了一个叫做 increment(forCount:) 的可选方法要求以及一个叫做 fixedIncrement 的可选属性要求。这些要求给 Counter 实例定义了两个不同的提供合适增量的方法。

注意

严格来讲，你可以写一个遵循 CounterDataSource 的自定义类而不实现任何协议要求。反正它们都是可选的。尽管技术上来讲是可以的，但这样的话就不能做一个好的数据源了。

```
1 classCounter{
2     varcount=0
3     vardataSource:CounterDataSource?
4     funcincrement(){
5         ifletamount=dataSource?.increment?(forCount:count){
6             count+=amount
7         }elseifletamount=dataSource?.fixedIncrement{
8             count+=amount
9         }
10    }
11 }
```

Counter 类在一个叫做 count 的变量属性里储存当前值。 Counter 类同样定义了一个叫做 increment 的方法，它在每次被调用的时候增加 count 属性。

`increment()` 方法首先通过查找自身数据源的 `increment(forCount:)` 的实现来尝试获取增量。`increment()` 方法使用可选链来尝试调用 `increment(forCount:)`，同时传入当前 `count` 值作为方法的唯一实际参数。

注意这里有两层可选链。首先，`dataSource` 有可能是 `nil`，所以 `dataSource` 名字后边有一个问号以表示只有 `dataSource` 不是 `nil` 的时候才能调用 `incrementForCount(forCount:)`。其次，就算 `dataSource` 确实存在，也没有人能保证它实现了 `incrementForCount(forCount:)`，因为它是可选要求。所以，`incrementForCount(forCount:)` 没有被实现的可能也被可选链处理。`incrementForCount(forCount:)` 的调用只发生在 `incrementForCount(forCount:)` 存在的情况下——也就是说，不是 `nil`。这就是为什么 `incrementForCount(forCount:)` 也在名字后边写一个问号。

由于对 `incrementForCount(forCount:)` 的调用可以两个理由中的任何一个而失败，调用返回一个可选的 `Int` 值。这就算 `incrementForCount(forCount:)` 在 `CounterDataSource` 中定义为返回一个非可选的 `Int` 值也生效。尽管这里有两个可选链运算，但结果仍然封装在一个可选项中。要了解更多关于多个可选链运算的信息，见 [链的多层连接](#)。

在 `increment(forCount:)` 调用之后，返回的可选的 `Int` 使用可选绑定展开到一个叫做 `amount` 的常量中。如果可选 `Int` 包含值——也就是说，如果委托和方法都存在，并且方法返回值——展开的 `amount` 添加到 `count` 存储属性，增加完成。

如果不能从 `increment(forCount:)` 方法取回值——无论是由于 `dataSource` 为空还是由于数据源没有实现 `increment(forCount:)`——那么 `increment()` 方法就尝试从数据源的 `fixedIncrement` 属性取回值。`fixedIncrement` 属性同样是一个可选要求，所以值是一个可选的 `Int` 值，就算 `fixedIncrement` 在 `CounterDataSource` 协议的定义中被定义为非可选 `Int` 属性。

这里是 `CounterDataSource` 的简单实现，数据源在每次查询时返回固定值 3。它通过实现可选 `fixedIncrement` 属性要求来实现这一点：

```
1 class ThreeSource: NSObject, CounterDataSource{
2     let fixedIncrement = 3
3 }
```

你可以使用 `ThreeSource` 的实例作为新 `Counter` 实例的数据源：

```
1 var counter = Counter()
2 counter.dataSource = ThreeSource()
3 for _ in 1...4{
4     counter.increment()
5     print(counter.count)
6 }
7 // 3
8 // 6
9 // 9
10 // 12
```

下边的代码创建了一个新的 `Counter` 实例；设置它的数据源为新的 `ThreeSource` 实例；并且调用计数器的 `increment()` 方法四次。按照预期，计数器的 `count` 属性在每次 `increment()` 调用是增加三。

这里有一个更加复杂一点的 `TowardsZeroSource`，它使 `Counter` 实例依照它当前的

count 值往上或往下朝着零计数：

```
1 @objc class TowardsZeroSource: NSObject, CounterDataSource{
2     func increment(forCount count: Int) -> Int{
3         if count == 0{
4             return 0
5         } else if count < 0{
6             return 1
7         } else{
8             return -1
9         }
10    }
11 }
```

TowardsZeroSource 类实现了 CounterDataSource 协议的可选 increment(forCount:) 方法并且使用 count 实际参数来计算改朝哪个方向计数。如果 count 已经是零，方法返回 0 来表示无需继续计数。

你可以使用 TowardsZeroSource 给现存的 Counter 实例来从 -4 到零。一旦计数器到零，就不会再变化：

```
1 counter.count = -4
2 counter.dataSource = TowardsZeroSource()
3 for _ in 1...5{
4     counter.increment()
5     print(counter.count)
6 }
7 // -3
8 // -2
9 // -1
10 // 0
11 // 0
```

协议扩展

协议可以通过扩展来提供方法和属性的实现以遵循类型。这就允许你在协议自身定义行为，而不是在每一个遵循或者在全局函数里定义。比如说， RandomNumberGenerator 协议可以扩展来提供 randomBool() 方法，它使用要求的 random() 方法来返回随机的 Bool 值：

```
1 extension RandomNumberGenerator{
2     func randomBool() -> Bool{
3         return random() > 0.5
4     }
5 }
```

通过给协议创建扩展，所有的遵循类型自动获得这个方法的实现而不需要任何额外的修改。

```
1 let generator = LinearCongruentialGenerator()
2 print("Here's a random number: \(generator.random())")
3 // Prints "Here's a random number: 0.37464991998171"
4 print("And here's a random Boolean: \(generator.randomBool())")
5 // Prints "And here's a random Boolean: true"
```

提供默认实现

你可以使用协议扩展来给协议的任意方法或者计算属性要求提供默认实现。如果遵循类型给这个协议的要求提供了它自己的实现，那么它就会替代扩展中提供的默认实现。

注意

通过扩展给协议要求提供默认实现与可选协议要求的区别是明确的。尽管遵循协议都不需要提供它们自己的实现。有默认实现的要求不需要使用可选链就能调用。

举例来说，继承自 `TextRepresentable` 的 `PrettyTextRepresentable` 协议可以给它要求的 `prettyTextualDescription` 属性提供一个默认实现来简单的返回访问 `textualDescription` 属性的结果：

```
1 extensionPrettyTextRepresentable{
2     var prettyTextualDescription: String{
3         return textualDescription
4     }
5 }
```

给协议扩展添加限制

当你定义一个协议扩展，你可以明确遵循类型必须在扩展的方法和属性可用之前满足的限制。如同 `Where` 分句（此处应有链接）中描述的那样，在扩展协议名字后边使用 `where` 分句来写这些限制。比如说，你可以给 `Collection` 定义一个扩展来应用于任意元素遵循上面 `TextRepresentable` 协议的集合。

```
1 extensionCollection whereIterator.Element: TextRepresentable{
2     var textualDescription: String{
3         let itemsAsText = self.map{$0.textualDescription}
4         return "[" + itemsAsText.joined(separator: ", ") + "]"
5     }
6 }
```

`textualDescription` 属性返回整个集合写在花括号里通过用逗号组合集合中每个元素的文本化表示的文本化描述。

考虑之前的 `Hamster` 结构体，它遵循 `TextRepresentable` 协议，`Hamster` 值的数组：

```
1 let murrayTheHamster = Hamster(name: "Murray")
2 let morganTheHamster = Hamster(name: "Morgan")
3 let mauriceTheHamster = Hamster(name: "Maurice")
4 let hamsters = [murrayTheHamster, morganTheHamster, mauriceTheHamster]
```

由于 `Array` 遵循 `Collection` 并且数组的元素遵循 `TextRepresentable` 协议，数组可以使用 `textualDescription` 属性来获取它内容的文本化表示：

```
1 print(hamsters.textualDescription)
2 // Prints "[A hamster named Murray, A hamster named Morgan, A hamster named Maurice]"
```

注意

如果遵循类型满足了为相同方法或者属性提供实现的多限制扩展的要求，Swift 会使用最匹配限制的实现。

泛型

 cnswift.org/generics

泛型代码让你能根据你所定义的要求写出可以用于任何类型的灵活的、可复用的函数。你可以编写出可复用、意图表达清晰、抽象的代码。

泛型是 Swift 最强大的特性之一，很多 Swift 标准库是基于泛型代码构建的。实际上，甚至你都没有意识到在语言指南中一直在使用泛型。例如，Swift 的 `Array` 和 `Dictionary` 类型都是泛型集合。你可以创建一个容纳 `Int` 值的数组，或者容纳 `String` 值的数组，甚至容纳任何 Swift 可以创建的其他类型的数组。同样，你可以创建一个存储任何指定类型值的字典，而且类型没有限制。

泛型解决的问题

下面的 `swapTwoInts(_:_:)` 是一个标准的非泛型函数，用于交换两个 `Int` 值：

```
1 func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

如输入输出形式参数中描述的一样，这个函数用输入输出形式参数来交换 `a` 和 `b` 的值。

`swapTwoInts(_:_:)` 函数把 `b` 原本的值给 `a`，把 `a` 原本的值给 `b`。你可以调用这个函数来交换两个 `Int` 变量的值。

```
1 var someInt = 3  
2 var anotherInt = 107  
3 swapTwoInts(&someInt, &anotherInt)  
4 print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")  
5 // Prints "someInt is now 107, and anotherInt is now 3"
```

`swapTwoInts(_:_:)` 函数很实用，但是它只能用于 `Int` 值。如果你想交换两个 `String` 值，或者两个 `Double` 值，你只能再写更多的函数，比如下面的 `swapTwoStrings(_:_:)` 和 `swapTwoDoubles(_:_:)` 函数：

```
1 func swapTwoStrings(_ a: inout String, _ b: inout String) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

```
1 func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

你可能已经注意到了，`swapTwoInts(_:_)`、`swapTwoStrings(_:_)`、`swapTwoDoubles(_:_)` 函数体是一样的。唯一的区别是它们接收值类型不同（`Int`、`String` 和 `Double`）。

写一个可以交换任意类型值的函数会更实用、更灵活。泛型代码让你能写出这样的函数。
(下文中定义了这些函数的泛型版本。)

三个函数中，`a` 和 `b` 被定义为了相同的类型，这一点很重要。如果 `a` 和 `b` 类型不一样，不能交换它们的值。Swift 是类型安全的语言，不允许（例如）一个 `String` 类型的变量和一个 `Double` 类型的变量交换值。尝试这样做会引发一个编译错误。

泛型函数

泛型函数可以用于任何类型。这里是上面提到的 `swapTwoInts(_:_)` 函数的泛型版本，叫做 `swapTwoValues(_:_)`：

```
1 func swapTwoValues<T>(_a:inout T,_b:inout T){  
2     let temporaryA=a  
3     a=b  
4     b=temporaryA  
5 }
```

`swapTwoValues(_:_)` 和 `swapTwoInts(_:_)` 函数体是一样的。但是，`swapTwoValues(_:_)` 和 `swapTwoInts(_:_)` 的第一行有点不一样。下面是首行的对比：

```
1 func swapTwoInts(_a:inout Int,_b:inout Int)  
2 func swapTwoValues<T>(_a:inout T,_b:inout T)
```

泛型版本的函数用了一个占位符类型名（这里叫做 `T`），而不是一个实际的类型名（比如 `Int`、`String` 或 `Double`）。占位符类型名没有声明 `T` 必须是什么样的，但是它确实说了 `a` 和 `b` 必须都是同一个类型 `T`，或者说都是 `T` 所表示的类型。替代 `T` 实际使用的类型将在每次调用 `swapTwoValues(_:_)` 函数时决定。

其他的区别是泛型函数名（`swapTwoValues(_:_)`）后面有包在尖括号（`<T>`）里的占位符类型名（`T`）。尖括号告诉 Swift，`T` 是一个 `swapTwoValues(_:_)` 函数定义里的占位符类型名。因为 `T` 是一个占位符，Swift 不会查找真的叫 `T` 的类型。

现在，可以用调用 `swapTwoInts` 的方式来调用 `swapTwoValues(_:_)` 函数，除此之外，可以给函数传递两个任意类型的值，只要两个实参的类型一致即可。每次调用 `swapTwoValues(_:_)`，用于 `T` 的类型会根据传入函数的值类型自动推断。

在下面的两个例子中，`T` 分别被推断为 `Int` 和 `String`：

```
1 var someInt=3  
2 var anotherInt=107  
3 swapTwoValues(&someInt,&anotherInt)  
4 // someInt is now 107, and anotherInt is now 3  
5 var someString="hello"  
6 var anotherString="world"  
7 swapTwoValues(&someString,&anotherString)  
8 // someString is now "world", and anotherString is now "hello"  
9
```

上面定义的 `swapTwoValues(_:_)` 函数受一个名为 `swap` 的泛型函数启发，`swap` 函数是 Swift 标准库的一部分，可以用于你的应用中。如果你需要在你自己的代码中用 `swapTwoValues(_:_)` 函数的功能，可以直接用 Swift 提供的 `swap(_:_)` 函数，不需要自己实现。

类型形式参数

上面的 `swapTwoValues(_:_)` 中，占位符类型 `T` 就是一个类型形式参数的例子。类型形式参数指定并且命名一个占位符类型，紧挨着写在函数名后面的一对尖括号里（比如 `<T>`）。

一旦你指定了一个类型形式参数，你就可以用它定义一个函数形式参数（比如 `swapTwoValues(_:_)` 函数中的形式参数 `a` 和 `b`）的类型，或者用它做函数返回值类型，或者做函数体中类型标注。在不同情况下，用调用函数时的实际类型来替换类型形式参数。

（上面的 `swapTwoValues(_:_)` 例子中，第一次调用函数的时候用 `Int` 替换了 `T`，第二次调用是用 `String` 替换的。）

你可以通过在尖括号里写多个用逗号隔开的类型形式参数名，来提供更多类型形式参数。

命名类型形式参数

大多数情况下，类型形式参数的名字要有描述性，比如 `Dictionary<Key,Value>` 中的 `Key` 和 `Value`，借此告知读者类型形式参数和泛型类型、泛型用到的函数之间的关系。但是，他们之间的关系没有意义时，一般按惯例用单个字母命名，比如 `T`、`U`、`V`，比如上面的 `swapTwoValues(_:_)` 函数中的 `T`。

类型形式参数永远用大写开头的驼峰命名法（比如 `T` 和 `MyTypeParameter`）命名，以指明它们是一个类型的占位符，不是一个值。

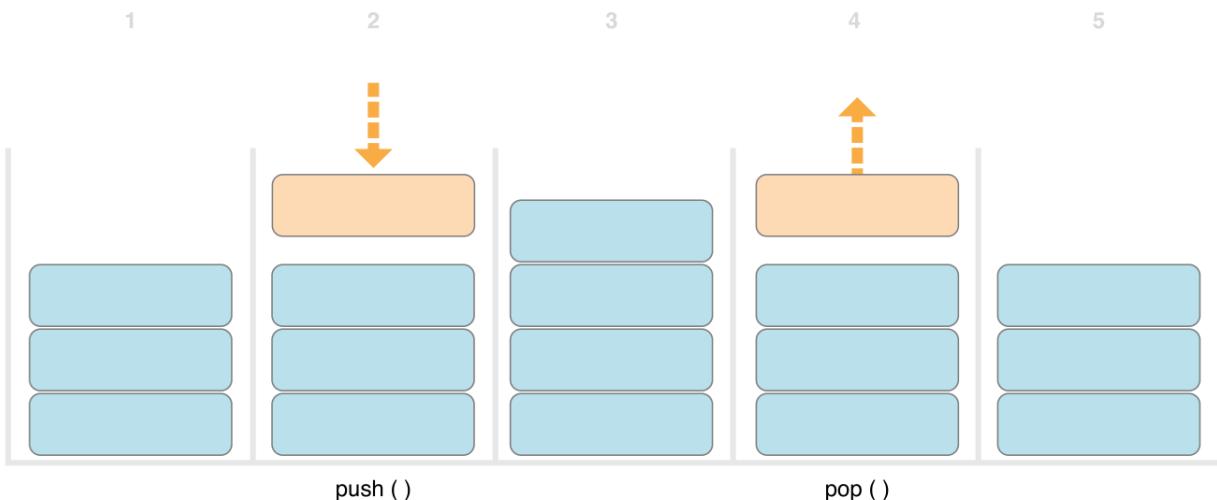
泛型类型

除了泛型函数，Swift 允许你定义自己的泛型类型。它们是可以用于任意类型的自定义类、结构体、枚举，和 `Array`、`Dictionary` 方式类似。

本章将向你展示如何写出一个叫做 `Stack` 的泛型集合类型。栈是值的有序集合，和数组类似，但是比 Swift 的 `Array` 类型有更严格的操作限制。数组允许在其中任何位置插入和移除元素。但是，栈的新元素只能添加到集合的末尾（这就是所谓的压栈）。同样，栈只允许从集合的末尾移除元素（这就是所谓的出栈）。

`UINavigationController` 类在它的导航层级关系中管理视图控制器就是用的栈的思想。你可以调用 `UINavigationController` 类的 `pushViewController(_:animated:)` 方法添加（或者说 `push`）一个视图控制器到导航栈里，用 `popViewControllerAnimated(_:)` 方法从导航栈移除（或者说 `pop`）一个视图控制器。当你需要用严格的“后进，先出”方式管理一个集合时，栈是一个很有用的集合模型。

下面的图示展示了压栈和出栈的行为：



1. 现在栈里有三个值；
2. 第四个值压到栈顶；
3. 栈里现在有四个值，最近添加的那个在顶部；
4. 栈中顶部的元素被移除，或者说叫“出栈”；
5. 移除一个元素之后，栈里又有三个元素了。

这里是如何写一个非泛型版本的栈，这种情况是一个 Int 值的栈：

```

1 structIntStack{
2     varitems=[Int]()
3     mutatingfuncpush(_item:Int){
4         items.append(item)
5     }
6     mutatingfuncpop()->Int{
7         return items.removeLast()
8     }
9 }
```

这个结构体用了一个叫做 `items` 的 `Array` 属性去存储栈中的值。 `Stack` 提供两个方法，`push` 和 `pop`，用于添加和移除栈中的值。这两个方法被标记为 `mutating`，是因为他们需要修改（或者说改变）结构体的 `items` 数组。

上面展示的 `IntStack` 类型只能用于 `Int` 值。但是定义一个泛型 `Stack` 会更实用，这样可以管理任何类型值的栈。

这里有一个相同代码的泛型版本：

```

1 structStack<Element>{
2     varitems=[Element]()
3     mutatingfuncpush(_item:Element){
4         items.append(item)
5     }
6     mutatingfuncpop()->Element{
7         return items.removeLast()
8     }
9 }
```

注意，这个泛型的 `Stack` 和非泛型版本的本质上是一样的，只是用一个叫做 `Element` 的类型形式参数代替了实际的 `Int` 类型。这个类型形式参数写在一对尖括号 (`<Element>`) 里，紧跟在结构体名字后面。

`Element` 为稍后提供的“某类型 `Element`”定义了一个占位符名称。这个未来的类型可以在结

构体定义内部任何位置以” Element “引用。在这个例子中，有三个地方将 Element 作为一个占位符使用：

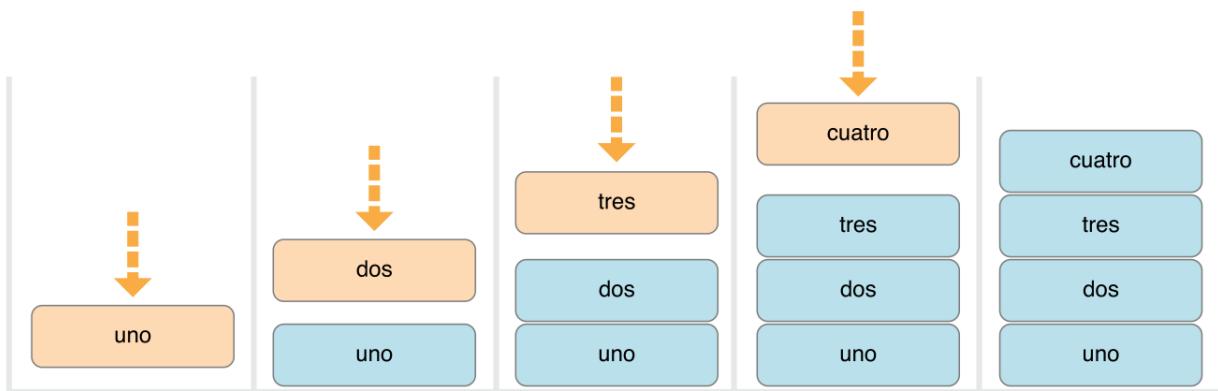
- 创建一个名为 items 的属性，用一个 Element 类型值的空数组初始化这个属性；
- 指定 push(_:) 方法有一个叫做 item 的形式参数，其必须是 Element 类型；
- 指定 pop() 方法的返回值是一个 Element 类型的值。

因为它是泛型，因此能以 Array 和 Dictionary 相似的方式，用 Stack 创建一个 Swift 中有效的任意类型的栈。

通过在尖括号中写出存储在栈里的类型，来创建一个新的 Stack 实例。例如，创建一个新的字符串栈，可以写 Stack<String>() :

```
1 var stackOfStrings=Stack<String>()
2 stackOfStrings.push("uno")
3 stackOfStrings.push("dos")
4 stackOfStrings.push("tres")
5 stackOfStrings.push("cuatro")
6 // the stack now contains 4 strings
```

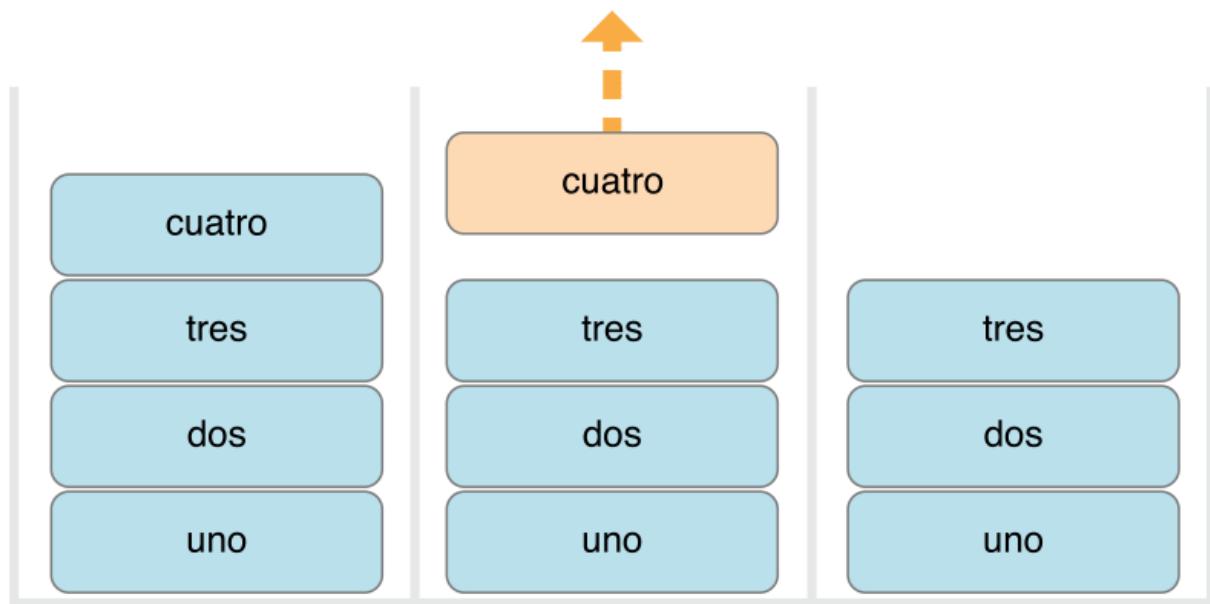
这是往栈里压入四个值之后， stackOfStrings 的图示：



从栈中移除并返回顶部的值， "cuatro" :

```
1 let fromTheTop=stackOfStrings.pop()
2 // fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

这是栈顶部的值出栈后的栈图示：



扩展一个泛型类型

当你扩展一个泛型类型时，不需要在扩展的定义中提供类型形式参数列表。原始类型定义的类型形式参数列表在扩展体里仍然有效，并且原始类型形式参数列表名称也用于扩展类型形式参数。

下面的例子扩展了泛型 Stack 类型，向其中添加一个叫做 `topItem` 的只读计算属性，不需要从栈里移除就能返回顶部的元素：

```

1 extension Stack{
2     var topItem: Element? {
3         return items.isEmpty ? nil : items[items.count - 1]
4     }
5 }
```

`topItem` 属性返回一个 `Element` 类型的可选值。如果栈是空的，`topItem` 返回 `nil`；如果栈非空，`topItem` 返回 `items` 数组的最后一个元素。

注意，这个扩展没有定义类型形式参数列表。相反，扩展中用 `Stack` 已有的类型形式参数名称，`Element`，来指明计算属性 `topItem` 的可选项类型。

现在，不用移除元素，就可以用任何 `Stack` 实例的 `topItem` 计算属性来访问和查询它顶部的元素：

```

1 if let topItem = stackOfStrings.topItem {
2     print("The top item on the stack is \(topItem).")
3 }
4 // Prints "The top item on the stack is tres."
```

类型约束

`swapTwoValues(_:_:)` 函数和 `Stack` 类型可以用于任意类型。但是，有时在用于泛型函数的类型和泛型类型上，强制其遵循特定的类型约束很有用。类型约束指出一个类型形式参数必须继承自特定类，或者遵循一个特定的协议、组合协议。

例如，Swift的 Dictionary 类型在可以用于字典中键的类型上设置了一个限制。如字典中描述的一样，字典键的类型必须是可哈希的。也就是说，它必须提供一种使其可以唯一表示的方法。Dictionary 需要它的键是可哈希的，以便它可以检查字典中是否包含一个特定键的值。没有了这个要求，Dictionary 不能区分该插入还是替换一个指定键的值，也不能在字典中查找已经给定的键的值。

这个要求通过 Dictionary 键类型上的类型约束实现，它指明了键类型必须遵循 Swift 标准库中定义的 Hashable 协议。所有 Swift 基本类型（比如 String、Int、Double 和 Bool）默认都是可哈希的。

创建自定义泛型类型时，你可以定义你自己的类型约束，这些约束可以提供强大的泛型编程能力。像 Hashable 这样的抽象概念，根据概念上的特征，而不是确切的类型来表征类型。

类型约束语法

在一个类型形式参数名称后面放置一个类或者协议作为形式参数列表的一部分，并用冒号隔开，以写出一个类型约束。下面展示了一个泛型函数类型约束的基本语法（和泛型类型的语法相同）：

```
1 func someFunction<T:SomeClass,U:SomeProtocol>(someT:T,someU:U){  
2 // function body goes here  
3 }
```

上面的假想函数有两个形式参数。第一个类型形式参数，T，有一个类型约束要求 T 是 SomeClass 的子类。第二个类型形式参数，U，有一个类型约束要求 U 遵循 SomeProtocol 协议。

类型约束的应用

这是一个叫做 `findIndex(ofString:in:)` 的非泛型函数，在给定的 String 值数组中查找给定的 String 值。`findIndex(ofString:in:)` 函数返回一个可选的 Int 值，如果找到了给定字符串，它会返回数组中第一个匹配的字符串的索引值，如果找不到给定字符串就返回 nil：

```
1 func findIndex(ofString valueToFind:String,inarray:[String])->Int?{  
2 for(index,value)inarray.enumerated(){  
3 if value==valueToFind{  
4 return index  
5 }  
6 }  
7 return nil  
8 }
```

`findIndex(ofString:in:)` 函数可以用于字符串数组中查找字符串值：

```
1 let strings=["cat","dog","llama","parakeet","terrapin"]  
2 if let foundIndex=findIndex(ofString:"llama",in:strings){  
3 print("The index of llama is \(foundIndex)")  
4 }  
5 // Prints "The index of llama is 2"
```

在数组中查找值的索引的原理只能用于字符串。但是，通过某种 T 类型的值代替所有用到的字符串，你可以用泛型函数写一个相同的功能。

这里写出一个叫做 `findIndex(of:in:)` 的函数，可能是你期望的 `findIndex(ofString:in:)` 函数

的一个泛型版本。注意，函数的返回值仍然是 Int?，因为函数返回一个可选的索引数字，而不是数组里的一个可选的值。这个函数没有编译，例子后面会解释原因：

```
1 funcfindIndex<T>(of valueToFind:T,inarray:[T])->Int?{
2 for(index,value)inarray.enumerated(){
3 ifvalue==valueToFind{
4 returnindex
5 }
6 }
7 returnnil
8 }
```

这个函数没有像上面写的那样编译。问题在于相等检查，“ifvalue==valueToFind”。Swift 中的类型不是每种都能用相等操作符（==）来比较的。如果你创建自己的类或者结构体去描述一个复杂的数据模型，比如说，对于那个类或结构体来说，“相等”的意义不是 Swift 能替你猜出来的。因此，不能保证这份代码可以用于所有 T 可以表示的类型，当你尝试编译这份代码时会提示一个相应的错误。

并非无路可走，总之，Swift 标准库中定义了一个叫做 Equatable 的协议，要求遵循其协议的类型要实现相等操作符（==）和不等操作符（!=），用于比较该类型的任意两个值。所有 Swift 标准库中的类型自动支持 Equatable 协议。

任何 Equatable 的类型都能安全地用于 findIndex(of:) 函数，因为可以保证那些类型支持相等操作符。为了表达这个事实，当你定义函数时将 Equatable 类型约束作为类型形式参数定义的一部分书写：

```
1 funcfindIndex<T:Equatable>(of valueToFind:T,inarray:[T])->Int?{
2 for(index,value)inarray.enumerated(){
3 ifvalue==valueToFind{
4 returnindex
5 }
6 }
7 returnnil
8 }
```

findIndex(of:) 的类型形式参数写作 T:Equatable，表示“任何遵循 Equatable 协议的类型 T”。

findIndex(of:) 函数现在可以成功编译，并且可以用于任何 Equatable 的类型，比如 Double 或者 String：

```
1 letdoubleIndex=findIndex(of:9.3,in:[3.14159,0.1,0.25])
2 // doubleIndex is an optional Int with no value, because 9.3 is not in the array
3 letstringIndex=findIndex(of:"Andrea",in:["Mike","Malcolm","Andrea"])
4 // stringIndex is an optional Int containing a value of 2
```

关联类型

定义一个协议时，有时在协议定义里声明一个或多个关联类型是很有用的。关联类型给协议中用到的类型一个占位符名称。直到采纳协议时，才指定用于该关联类型的实际类型。关联类型通过 associatedtype 关键字指定。

关联类型的应用

这里是一个叫做 Container 的示例协议，声明了一个叫做 ItemType 的关联类型：

```
1 protocolContainer{  
2     associatedtype ItemType  
3     mutatingfuncappend(_item:ItemType)  
4     varcount: Int{get}  
5     subscript(i:Int)->ItemType{get}  
6 }
```

Container 协议定义了三个所有容器必须提供的功能：

- 必须能够通过 append(_) 方法向容器中添加新元素；
- 必须能够通过一个返回 Int 值的 count 属性获取容器中的元素数量；
- 必须能够通过 Int 索引值的下标取出容器中每个元素。

这个协议没有指定元素如何储存在容器中，也没指定允许存入容器的元素类型。协议仅仅指定了想成为一个 Container 的类型，必须提供的三种功能。遵循该协议的类型可以提供其他功能，只要满足这三个要求即可。

任何遵循 Container 协议的类型必须能指定其存储值的类型。尤其是它必须保证只有正确类型的元素才能添加到容器中，而且该类型下标返回的元素类型必须是正确的。

为了定义这些要求，Container 协议需要一种在不知道容器具体类型的情况下，引用该容器将存储的元素类型的方法。Container 协议需要指定所有传给 append(_) 方法的值必须和容器里元素的值类型是一样的，而且容器下标返回的值也是和容器里元素的值类型相同。

为了实现这些要求，Container 协议声明了一个叫做 ItemType 的关联类型，写作 associatedtype ItemType。协议没有定义 ItemType 是什么类型，这个信息留给遵循协议的类型去提供。但是，ItemType 这个别名，提供了一种引用 Container 中元素类型的方式，定义了一种用于 Container 方法和下标的类型，确保了任何 Container 期待的行为都得到满足。

这是前面非泛型版本的 IntStack ，使其遵循 Container 协议：

```
1 structIntStack: Container{  
2     // original IntStack implementation  
3     varitems=[Int]()  
4     mutatingfuncpush(_item:Int){  
5         items.append(item)  
6     }  
7     mutatingfuncpop()->Int{  
8         returnitems.removeLast()  
9     }  
10    // conformance to the Container protocol  
11    typealias ItemType=Int  
12    mutatingfuncappend(_item:Int){  
13        self.push(item)  
14    }  
15    varcount: Int{  
16        returnitems.count  
17    }  
18    subscript(i:Int)->Int{  
19        returnitems[i]  
20    }  
21 }
```

IntStack 实现了 Container 协议所有的要求，为满足这些要求，封装了 IntStack 里现有的方法。

此外，IntStack 为了实现 Container 协议，指定了适用于 ItemType 的类型是 Int 类型。
typealias ItemType = Int 把 ItemType 抽象类型转换为了具体的 Int 类型。

感谢 Swift 的类型推断功能，你不用真的在 IntStack 定义中声明一个具体的 Int 类型 ItemType。因为 IntStack 遵循 Container 协议的所有要求，通过简单查看 append(_:) 方法的 item 形式参数和下标的返回类型，Swift 可以推断出合适的 ItemType。如果你真的从上面的代码中删除 typealias ItemType = Int，一切都会正常运行，因为 ItemType 该用什么类型是非常明确的。

你也可以做一个遵循 Container 协议的泛型 Stack 类型：

```
1 struct Stack<Element>: Container{
2     // original Stack<Element> implementation
3     var items = [Element]()
4     mutating func push(_ item: Element) {
5         items.append(item)
6     }
7     mutating func pop() -> Element {
8         return items.removeLast()
9     }
10    // conformance to the Container protocol
11    mutating func append(_ item: Element) {
12        self.push(item)
13    }
14    var count: Int {
15        return items.count
16    }
17    subscript(i: Int) -> Element {
18        return items[i]
19    }
20 }
```

这次，类型形式参数 Element 用于 append(_) 方法的 item 形式参数和下标的返回类型。因此，对于这个容器，Swift 可以推断出 Element 是适用于 ItemType 的类型。

给关联类型添加约束

你可以在协议里给关联类型添加约束来要求遵循的类型满足约束。比如说，下面的代码定义了一个版本的 Container，它要求容器中的元素都是可判等的。

```
1 protocol Container {
2     associatedtype Item: Equatable
3     mutating func append(_ item: Item)
4     var count: Int { get }
5     subscript(i: Int) -> Item { get }
6 }
```

要遵循这个版本的 Container，容器的 Item 必须遵循 Equatable 协议。

在关联类型约束里使用协议

协议可以作为它自身的要求出现。比如说，这里有一个协议细化了 Container 协议，添加了一个 suffix(_:) 方法。suffix(_:) 方法返回容器中从后往前给定数量的元素，把它们存储在一

个 Suffix 类型的实例里。

```
1 protocol SuffixableContainer: Container{  
2     associatedtype Suffix: SuffixableContainer where Suffix.Item == Item  
3     func suffix(_ size: Int) -> Suffix  
4 }
```

在这个协议里，Suffix 是一个关联类型，就像上边例子中 Container 的 Item 类型一样。Suffix 拥有两个约束：它必须遵循 SuffixableContainer 协议（就是当前定义的协议），以及它的 Item 类型必须是和容器里的 Item 类型相同。Item 的约束是一个 where 分句，它在下面带有泛型 Where 分句的扩展中有讨论。

这里有一个来自闭包的循环强引用的 Stack 类型的扩展，它添加了对 SuffixableContainer 协议的遵循：

```
1 extension Stack: SuffixableContainer{  
2     func suffix(_ size: Int) -> Stack{  
3         var result = Stack()  
4         for index in (count - size)..< count{  
5             result.append(self[index])  
6         }  
7         return result  
8     }  
9     // Inferred that Suffix is Stack.  
10 }  
11 var stackOfInts = Stack<Int>()  
12 stackOfInts.append(10)  
13 stackOfInts.append(20)  
14 stackOfInts.append(30)  
15 let suffix = stackOfInts.suffix(2)  
16 // suffix contains 20 and 30
```

在上面的例子中，Suffix 是 Stack 的关联类型，也就是 Stack，所以 Stack 的后缀运算返回另一个 Stack。另外，遵循 SuffixableContainer 的类型可以拥有一个与它自己不同的 Suffix 类型——也就是说后缀运算可以返回不同的类型。比如说，这里有一个非泛型 IntStack 类型的扩展，它添加了 SuffixableContainer 遵循，使用 Stack<Int> 作为它的后缀类型而不是 IntStack：

```
1 extension IntStack: SuffixableContainer{  
2     func suffix(_ size: Int) -> Stack<Int>{  
3         var result = Stack<Int>()  
4         for index in (count - size)..< count{  
5             result.append(self[index])  
6         }  
7         return result  
8     }  
9     // Inferred that Suffix is Stack<Int>.  
10 }
```

扩展现有类型来指定关联类型

你可以扩展一个现有类型使其遵循一个协议，如在扩展里添加协议遵循描述的一样。这包括一个带关联类型的协议。

Swift 的 Array 类型已经提供了 append(_) 方法、count 属性、用 Int 索引取出其元素的下标。这三个功能满足了 Container 协议的要求。这意味着你可以通过简单地声明 Array 采纳协议，扩展 Array 使其遵循 Container 协议。通过一个空的扩展实现，如使用扩展声明采纳

协议：

```
1 extensionArray: Container{}
```

数组已有的 `append(_)` 方法和下标使得 Swift 能为 `ItemType` 推断出合适的类型，就像上面的泛型 `Stack` 类型一样。定义这个扩展之后，你可以把任何 `Array` 当做一个 `Container` 使用。

泛型 Where 分句

如类型约束中描述的一样，类型约束允许你在泛型函数或泛型类型相关的类型形式参数上定义要求。

类型约束在为关联类型定义要求时也很有用。通过定义一个泛型 `Where` 分句来实现。泛型 `Where` 分句让你能够要求一个关联类型必须遵循指定的协议，或者指定的类型形式参数和关联类型必须相同。泛型 `Where` 分句以 `Where` 关键字开头，后接关联类型的约束或类型和关联类型一致的关系。泛型 `Where` 分句写在一个类型或函数体的左半个大括号前面。

下面的例子定义了一个叫做 `allItemsMatch` 的泛型函数，用来检查两个 `Container` 实例是否包含相同顺序的相同元素。如果所有元素都匹配，函数返回布尔值 `true`，否则返回 `false`。

被检查的两个容器不一定是相同类型的（尽管它们可以是），但是它们的元素类型必须相同。这个要求通过类型约束和泛型 `Where` 分句一起体现：

```
1 func allItemsMatch<C1:Container,C2:Container>
2     (_someContainer:C1,_anotherContainer:C2)->Bool
3     where C1.ItemType==C2.ItemType,C1.ItemType: Equatable{
4         // Check that both containers contain the same number of items.
5         if someContainer.count!=anotherContainer.count{
6             return false
7         }
8         // Check each pair of items to see if they are equivalent.
9         for i in 0..
```

这个函数有两个形式参数，`someContainer` 和 `anotherContainer`。`someContainer` 形式参数是 `C1` 类型，`anotherContainer` 形式参数是 `C2` 类型。`C1` 和 `C2` 是两个容器类型的类型形式参数，它们的类型在调用函数时决定。

下面是函数的两个类型形式参数上设置的要求：

- `C1` 必须遵循 `Container` 协议（写作 `C1:Container`）；
- `C2` 也必须遵循 `Container` 协议（写作 `C2:Container`）；
- `C1` 的 `ItemType` 必须和 `C2` 的 `ItemType` 相同（写作 `C1.ItemType==C2.ItemType`）；
- `C1` 的 `ItemType` 必须遵循 `Equatable` 协议（写作 `C1.ItemType:Equatable`）。

前两个要求定义在了函数的类型形式参数列表里，后两个要求定义在了函数的泛型 Where 分句中。

这些要求意味着：

- someContainer 是一个 C1 类型的容器；
- anotherContainer 是一个 C2 类型的容器；
- someContainer 和 anotherContainer 中的元素类型相同；
- someContainer 中的元素可以通过不等操作符 (!=) 检查它们是否不一样。

后两个要求放到一起意味着，anotherContainer 中的元素也可以通过 != 操作符检查，因为它们和 someContainer 中的元素类型完全相同。

这些要求使得 allItemsMatch(_:_:) 函数可以比较两个容器，即使它们是不同类型的容器。

allItemsMatch(_:_:) 函数开始会先检查两个容器中的元素数量是否相同。如果它们的元素数量不同，它们不可能匹配，函数就会返回 false。

检查完数量之后，用一个 for-in 循环和半开区间操作符 (..<) 遍历 someContainer 中的所有元素。函数会检查 someContainer 中的每个元素，是否和 anotherContainer 中对应的元素不相等。如果两个元素不相等，则两个容器不匹配，函数返回 false。

如果循环完成都没有出现不匹配的情况，两个容器就是匹配的，则函数返回 true。

这是 allItemsMatch(_:_:) 函数使用的示例：

```
1 var stackOfStrings=Stack<String>()
2 stackOfStrings.push("uno")
3 stackOfStrings.push("dos")
4 stackOfStrings.push("tres")
5 var arrayOfStrings=["uno","dos","tres"]
6 if allItemsMatch(stackOfStrings,arrayOfStrings){
7     print("All items match.")
8 } else{
9     print("Not all items match.")
10 }
11 // Prints "All items match."
12
13
```

上面的例子创建了一个 Stack 实例来存储 String 值，压到栈中三个字符串。还创建了一个 Array 实例，用三个同样字符串的字面量初始化该数组。虽然栈和数组的类型不一样，但它们都遵循 Container 协议，并且它们包含的值类型一样。因此，你可以调用 allItemsMatch(_:_:) 函数，用那两个容器做函数的形式参数。上面的例子中，allItemsMatch(_:_:) 函数正确地报告了两个容器中所有元素匹配。

带有泛型 Where 分句的扩展

你同时也可以使用泛型的 where 分句来作为扩展的一部分。下面的泛型 Stack 结构体的扩展了先前的栗子，添加了一个 isTop(_:) 方法。

```

1 extensionStack whereElement: Equatable{
2 func isTop(_ item: Element) -> Bool{
3 guard let topItem = items.last else{
4 return false
5 }
6 return topItem == item
7 }
8 }

```

这个新的 `isTop(_)` 方法首先校验栈不为空，然后对比给定的元素与栈顶元素。如果你尝试不使用泛型 `where` 分句来做这个，你可能会遇到一个问题：`isTop(_)` 的实现要使用 `==` 运算符，但 `Stack` 的定义并不需要其元素可相等，所以使用 `==` 运算符会导致运行时错误。使用泛型 `where` 分句则允许你给扩展添加一个新的要求，这样扩展只会在栈内元素可判等的时候才给栈添加 `isTop(_)` 方法。

这是用法：

```

1 if stackOfStrings.isTop("tres"){
2 print("Top element is tres.")
3 }else{
4 print("Top element is something else.")
5 }
6 // Prints "Top element is tres."

```

如果尝试在元素不能判等的栈调用 `isTop(_)` 方法，你就会出发运行时错误。

```

1 struct NotEquatable{}
2 var notEquatableStack = Stack<NotEquatable>()
3 let notEquatableValue = NotEquatable()
4 notEquatableStack.push(notEquatableValue)
5 notEquatableStack.isTop(notEquatableValue) // Error

```

你可以使用泛型 `where` 分句来扩展到一个协议。下面的栗子把先前的 `Container` 协议扩展添加了一个 `startsWith(_)` 方法。

```

1 extension Container where Item: Equatable{
2 func startsWith(_ item: Item) -> Bool{
3 return count >= 1 && self[0] == item
4 }
5 }

```

`startsWith(_)` 方法首先确保容器拥有至少一个元素，然后它检查第一个元素是否与给定元素相同。这个新的 `startsWith(_)` 方法可以应用到任何遵循 `Container` 协议的类型上，包括前面我们用的栈和数组，只要容器的元素可以判等。

```

1 if [9, 9, 9].startsWith(42){
2 print("Starts with 42.")
3 }else{
4 print("Starts with something else.")
5 }
6 // Prints "Starts with something else."

```

上边例子中的泛型 `where` 分句要求 `Item` 遵循协议，但你同样可以写一个泛型 `where` 分句来要求 `Item` 为特定类型。比如：

```
1 extensionContainer whereItem==Double{
2     func average()->Double{
3         var sum=0.0
4         for index in 0..<count{
5             sum+=self[index]
6         }
7         return sum/Double(count)
8     }
9 }
10 print([1260.0,1200.0,98.6,37.0].average())
11 // Prints "648.9"
```

这个栗子在 Item 是 Double 时给容器添加了 average() 方法。它遍历容器中的元素来把它们相加，然后除以容器的总数来计算平均值。它显式地把总数从 Int 转为 Double 来允许浮点除法。

你可以在一个泛型 where 分句中包含多个要求来作为扩展的一部分，就如同你在其它地方写的泛型 where 分句一样。每一个需求用逗号分隔。

关联类型的泛型 Where 分句

你可以在关联类型中包含一个泛型 where 分句。比如说，假定你想要做一个包含遍历器的 Container，比如标准库中 Sequence 协议那样。那么你会这么写：

```
1 protocol Container{
2     associatedtype Item
3     mutating func append(_ item: Item)
4     var count: Int {get}
5     subscript(i: Int) -> Item {get}
6     associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
7     func makeIterator() -> Iterator
8 }
9
```

Iterator 中的泛型 where 分句要求遍历器以相同的类型遍历容器内的所有元素，无论遍历器是什么类型。makeIterator() 函数提供了容器的遍历器的访问。

对于一个继承自其他协议的协议来说，你可以通过在协议的声明中包含泛型 where 分句来给继承的协议中关联类型添加限定。比如说，下面的代码声明了一个 ComparableContainer 协议，它要求 Item 遵循 Comparable：

```
1 protocol ComparableContainer: Container where Item: Comparable {}
```

泛型下标

下标可以是泛型，它们可以包含泛型 where 分句。你可以在 subscript 后用尖括号来写类型占位符，你还可以在下标代码块花括号前写泛型 where 分句。举例来说：

```
1 extensionContainer{
2     subscript<Indices:Sequence>(indices:Indices)->[Item]
3     where Indices.Iterator.Element==Int{
4         var result=[Item]()
5         for index in indices{
6             result.append(self[index])
7         }
8         return result
9     }
10 }
```

这个 Container 协议的扩展添加了一个接收一系列索引并返回包含给定索引元素的数组。这个泛型下标有如下限定：

- 在尖括号中的泛型形式参数 Indices 必须是遵循标准库中 Sequence 协议的某类型；
- 下标接收单个形式参数， indices ，它是一个 Indices 类型的实例；
- 泛型 where 分句要求序列的遍历器必须遍历 Int 类型的元素。这就保证了序列中的索引都是作为容器索引的相同类型。

合在一起，这些限定意味着传入的 indices 形式参数是一个整数的序列。

内存安全性

 cnsnift.org/memory-safety

默认情况下，Swift 会阻止你代码中发生的不安全行为。比如说，Swift 会保证在使用前就初始化，内存在变量释放后这块内存就不能再访问了，以及数组会检查越界错误。

Swift 还通过要求标记内存位置来确保代码对内存有独占访问权，以确保了同一内存多访问时不会冲突。由于 Swift 自动管理内存，大部份情况下你根本不需要考虑访问内存的事情。总之，了解一下什么情况下会潜在导致冲突是一件很重要的事情，这样你就可以避免写出对内存访问冲突的代码了。如果你的代码确实包含冲突，你就会得到编译时或运行时错误。

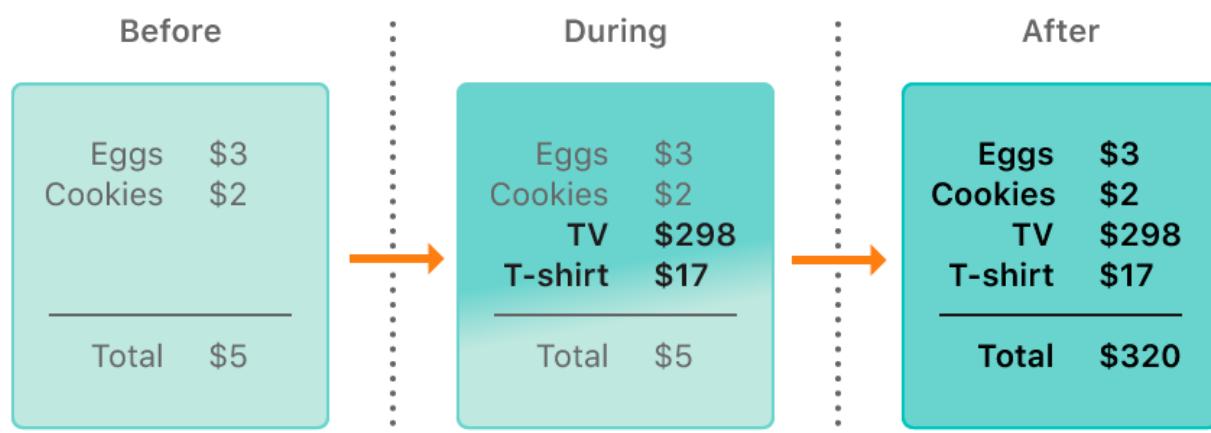
了解内存访问冲突

内存访问会在你做一些比如设置变量的值或者传递一个实际参数给函数的时候发生。比如说，下面的代码同时包含了读取访问和写入访问：

```
1 // A write access to the memory where one is stored.  
2 varone=1  
3 // A read access from the memory where one is stored.  
4 print("We're number \(one)!")  
5
```

内存访问冲突会在你的代码不同地方同一时间尝试访问同一块内存时发生。在同一时间多处访问同一块内存会产生不可预料或者说不一致的行为。在 Swift 中，有好几种方式来修改跨多行代码的值，从而可以在访问值的中间进行它自身的修改。

你可以想象一个类似的问题比如你是如何更新一张写在纸上的预算。更新预算是一个两步过程：首先你添加项目的名字和价格，然后你改变总价来显示当前列表中的变化。在更新的前后，你可以读取任何预算信息并且得到正确的结果，图例如下：



当你添加项目到预算时，它处在一个临时的状态，这是一个不可用的状态因为总价还没有更新来显示最新的价格。在添加新项目的过程中读取总价就会得到错误的信息。

这个例子也展示了你在修复内存访问冲突时可能会遭遇的挑战：有时多种修复冲突的方式会产生不同的结果，并且通常也不会明显哪个结果就是正确的。在这个例子中，基于你想要原本的总价还是更新后的总价，要么是 \$5 要么是 \$320 都可能是正确的。在你修复内存访问冲

突之前，你必须决定想要哪一种。

注意

如果你写并发或者多线程代码，内存访问冲突可能会是一个常见问题。总之，我们这里讨论的访问冲突也可以发生在单线程并且不涉及并发和多线程代码。

如果你在单线程遇到内存访问冲突，Swift 会保证你在要么编译时要么运行时得到错误。对于多线程代码，使用Thread Sanitizer 来帮助探测线程之间的访问冲突。

典型的内存访问

在访问冲突上下文中有三种典型的内存访问需要考虑：不论访问是读取还是写入，在访问过程中，以及内存地址被访问。具体来说，冲突会在你用两个访问并满足下列条件时发生：

- 至少一个是写入访问；
- 它们访问的是同一块内存；
- 它们的访问时间重叠。

读和写的区别通常显而易见：写入访问改变了内存，但读取访问不会。内存地址则指向被访问的东西——比如说，变量、常量或者属性。访问内存的时间要么是即时的，要么是长时间的。

如果一个访问在启动后其他代码不能执行直到它结束后才能，那么这个访问就是*即时的*。基于它们的特性，两个即时访问不能同时发生。大多数内存访问都是即时的。比如，下面列出的所有读写访问都是即时的：

```
1 func oneMore(than number:Int) -> Int{  
2     return number + 1  
3 }  
4 var myNumber = 1  
5 myNumber = oneMore(than:myNumber)  
6 print(myNumber)  
7 // Prints "2"  
8
```

总之，还有有很多访问内存的方法，比如被称作*长时访问*的，跨越其他代码执行过程。长时访问和即时访问的不同之处在于长时访开始后在它结束之前其他代码依旧可以运行，这就是所谓的*重叠*。长时访问可以与其他长时访问以及即时访问重叠。

重叠访问主要是出现在使用了输入输出形式参数的函数以及方法或者结构体中的异变方法。特定种类的使用长时访问的 Swift 代码在下文详述。

输入输出形式参数的访问冲突

拥有长时写入访问到所有自身输入输出形式参数的函数。对输入输出形式参数的写入访问会在所有非输入输出形式参数计算之后开始，并持续到整个函数调用结束。如果有多个输入输出形式参数，那么写入访问会以形式参数出现的顺序开始。

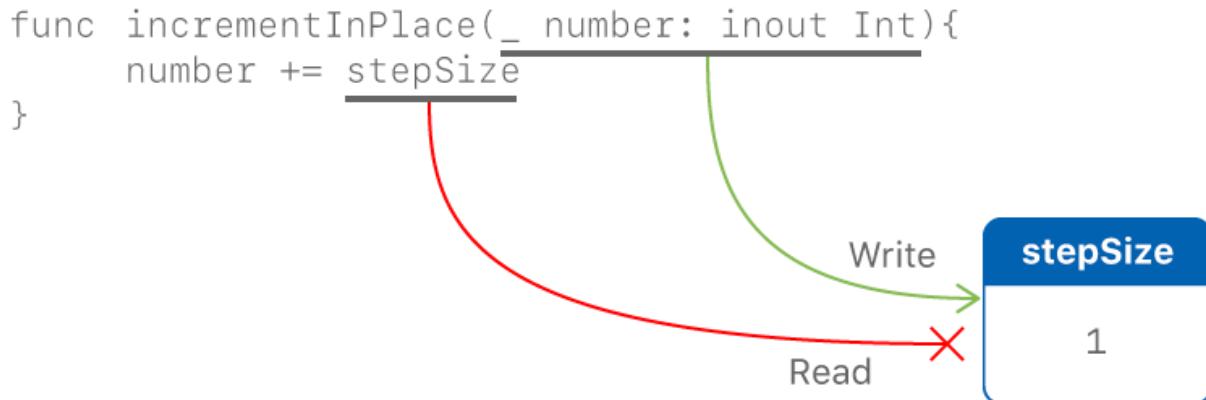
这种长时写入访问的一个后果就是你不能访问作为输入输出传递的原本变量，就算生效范围和访问控制可能会允许你这么做——任何对原变量的访问都会造成冲突，比如说：

```

1 var stepSize=1
2 func increment(_ number: inout Int){
3     number+=stepSize
4 }
5 increment(&stepSize)
6 // Error: conflicting accesses to stepSize
7
8

```

在上面的代码中，`stepSize` 是一个全局变量，它通常可以在 `increment(_)` 中可以访问。总之，`stepSize` 的读取访问与 `number` 的写入访问重叠了。如下图所示，`number` 和 `stepSize` 引用的是同一内存地址。读取和写入访问引用同一内存并且重叠，产生了冲突。



一种解决这个冲突的办法是显式地做一个 `stepSize` 的拷贝：

```

1 // Make an explicit copy.
2 var copyOfStepSize=stepSize
3 increment(&copyOfStepSize)
4 // Update the original.
5 stepSize=copyOfStepSize
6 // stepSize is now 2
7 // stepSize is now 2
8

```

当你在调用 `increment(_)` 之前给 `stepSize` 了一份之后，显然 `copyOfStepSize` 基于当前的步长增加了。读取访问在写入访问开始前结束，所以不会再有冲突。

输入输出形式参数的长时写入访问的另一个后果是传入一个单独的变量作为实际形式参数给同一个函数的多个输入输出形式参数产生冲突。比如：

```

1 func balance(_ x: inout Int, _ y: inout Int){
2     let sum=x+y
3     x=sum/2
4     y=sum-x
5 }
6 var playerOneScore=42
7 var playerTwoScore=30
8 balance(&playerOneScore, &playerTwoScore)// OK
9 balance(&playerOneScore, &playerOneScore)
10 // Error: Conflicting accesses to playerOneScore

```

上边 `balance(_:_)` 修改它的两个形式参数将它们的总数进行平均分配。用 `playerOneScore` 和 `playerTwoScore` 作为实际参数不会产生冲突——一共有两个写入访问在同一时间重叠，但它们访问的是不同的内存地址。相反，传入 `playerOneScore` 作为两个形式参数的值则产生冲突，因为它尝试执行两个写入访问到同一个内存地址且是在同一时间执

行。

注意

由于操作是函数，它们同样也可以对其输入输出形式参数进行长时访问，比如，如果 `balance(_:_)` 是一个名为 `<^>` 的操作符函数，写 `playerOneScore<^>playerOneScore` 就会造成和 `balance(&playerOneScore,&playerOneScore)` 一样的冲突。

在方法中对 `self` 的访问冲突

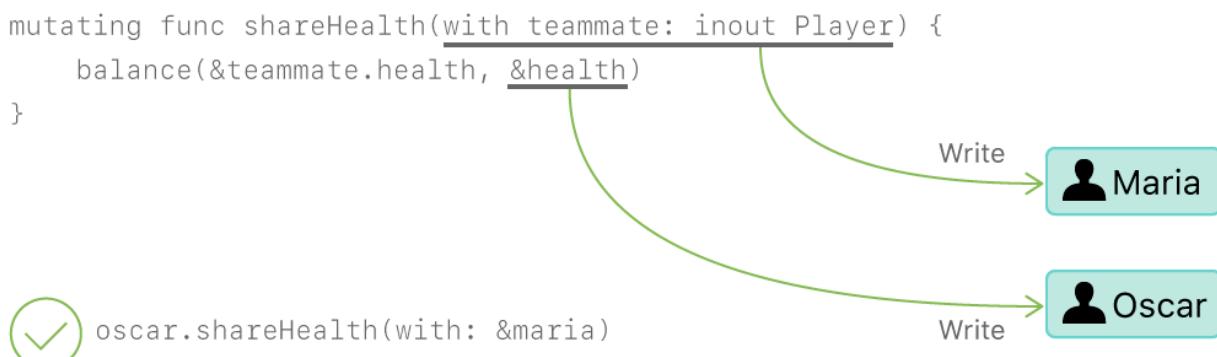
结构体中的异变方法可以在方法调用时对 `self` 进行写入访问。比如说想象一个每个玩家都有生命值的游戏，当玩家受伤时降低生命值，以及一个能量值，它在使用技能时降低。

```
1 structPlayer{  
2     varname:String  
3     varhealth:Int  
4     varenergy:Int  
5     staticletmaxHealth=10  
6     mutatingfuncrestoreHealth(){  
7         health=Player.maxHealth  
8     }  
9 }  
10
```

在上面 `restoreHealth()` 方法中，对 `self` 的写入访问在方法一开始就启动然后结束于方法返回。在这种情况下，在 `restoreHealth()` 中没有其他代码可能会重叠访问 `Player` 实例中的属性。下面的 `shareHealth(with:)` 方法则接收另一个 `Player` 实例作为输入输出形式参数，为重叠访问创建了可能性：

```
1 extensionPlayer{  
2     mutatingfuncshareHealth(with teammate:inoutPlayer){  
3         balance(&teammate.health,&health)  
4     }  
5 }  
6 varoscar=Player(name:"Oscar",health:10,energy:10)  
7 varmaria=Player(name:"Maria",health:5,energy:10)  
8 oscar.shareHealth(with:&maria)// OK  
9
```

在上面的例子中，调用 Oscar 玩家的 `shareHealth(with:)` 方法分享血条给 Maria 玩家不会造成冲突。在方法调用的过程中只有一个写入访问到 `oscar` 因为 `oscar` 是异变方法中 `self` 值，并且在同一时间内只有一个写入访问到 `maria` 因为 `maria` 是以输入输出形式参数传递的。如下面的图例所示，他们访问了内存中不同的地址。就算他们的写入访问是同时发生的，但不会冲突。

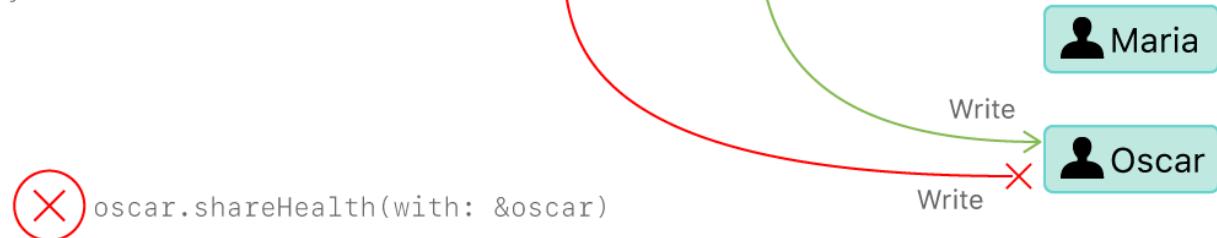


总之，如果你把 oscar 作为实际参数传递给 shareHealth(with:)，就出现了冲突：

```
1 oscar.shareHealth(with:&oscar)
2 // Error: conflicting accesses to oscar
```

异变方法在方法的自行过程中需要写入访问到 self，并且输入输出形式参数同时也需要写入访问到 teammate。在方法中，self 和 teammate 实际上引用自同一内存地址——如图所示。这两个写入访问引用到了同一个地址并重叠，产生冲突。

```
mutating func shareHealth(with teammate: inout Player) {
    balance(&teammate.health, &health)
}
```



属性的访问冲突

像是结构体、元组以及枚举这些类型都是由独立值构成，比如结构体的属性或者元组的元素。由于这些都是值类型，改变任何一个值都会改变整个类型，意味着读或者写访问到这些属性就需要对整个值进行读写访问。比如说，对元组的元素进行重叠写入访问就会产生冲突：

```
1 varplayerInformation=(health:10,energy:20)
2 balance(&playerInformation.health,&playerInformation.energy)
3 // Error: conflicting access to properties of playerInformation
```

在上边例子中，在同一元组的元素上调用 `balance(_:_)` 产生冲突是因为对 `playerInformation` 产生了重叠写入访问。`playerInformation.health` 和 `playerInformation.energy` 作为输入输出形式参数传入，这就意味着 `balance(_:_)` 在函数调用的过程中对他们产生写入访问。在这两种情况下，对元组元素的写入访问需要整个元组的写入访问。也就是说 `playerInformation` 在调用过程中有两个写入访问重叠，导致冲突。

下面的代码显示了对全局变量结构体属性的重叠写入访问，导致同样的错误。

```
1 varholly=Player(name:"Holly",health:10,energy:10)
2 balance(&holly.health,&holly.energy)// Error
```

实际上，大多数对结构体属性的访问可以安全的重叠。比如如果上边变量 holly 变成局部变量而不是全局变量，那么编译器就可以保证重叠访问结构体的存储属性是安全的：

```
1 funcsomeFunction(){
2     varoscar=Player(name:"Oscar",health:10,energy:10)
3     balance(&oscar.health,&oscar.energy)// OK
4 }
```

在上边的例子中，Oscar 的血条和能量作为两个输入输出形式参数传递给了 `balance(_:_)`。编译器可以证明内存安全得以保证是因为两个存储属性不会以任何形式交互。

对重叠访问结构体的属性进行限制并不总是必要才能保证内存安全性。内存安全性是一个需要的保证，但独占访问是比内存安全更严格的要求——也就是说某些代码保证了内存安全性，尽管它违反了内存的独占访问。如果编译器可以保证非独占访问内存仍然是安全的 Swift 就允许这些内存安全的代码。具体来说，如果下面的条件可以满足就说明重叠访问结构体的属性是安全的：

- 你只访问实例的存储属性，不是计算属性或者类属性；
- 结构体是局部变量而非全局变量；
- 结构体要么没有被闭包捕获要么只被非逃逸闭包捕获。

如果编译器不能保证访问是安全的，它就不允许访问。

访问控制

 cnswift.org/access-control

访问控制限制其他源文件和模块对你的代码的访问。这个特性允许你隐藏代码的实现细节，并指定一个偏好的接口让其他代码可以访问和使用。

你可以给特定的单个类型 (类，结构体和枚举)设置访问级别，比如说属性、方法、初始化器以及属于那些类型的下标。协议可以限制在一定的范围内使用，就像全局常量，变量，函数那样。

除了提供各种级别的访问控制，Swift 为典型场景提供默认的访问级别，减少了显式指定访问控制级别的需求。事实上，如果你编写单目标应用程序，你可能根本不需要显式指定访问控制级别。

注意

简洁起见，代码中可以设置访问级别的部分(属性，类型，函数等)在下面的章节称为“实体”。

模块和源文件

Swift 的访问控制模型基于模块和源文件的概念。

模块是单一的代码分配单元——一个框架或应用程序会作为的独立的单元构建和发布并且可以使用 Swift 的 `import` 关键字导入到另一个模块。

Xcode 中的每个构建目标 (例如应用程序包或框架) 在 Swift 中被视为一个独立的模块。如果你将应用程序的代码作为独立的框架组合在一起——或许可以在多个应用程序中封装和重用该代码——那么当在一个应用程序中导入和使用时，在该框架中定义的所有内容都将作为独立模块的一部分，或是当它在另一个框架中使用时。

源文件是一个模块中的单个 Swift 源代码文件 (实际上，是一个应用程序或是框架中的单个文件)。虽然通常在单独源文件中定义单个类型，但是一个源文件可以包含多个类型。函数等的定义。

访问级别

Swift 为代码的实体提供个五个不同的访问级别。这些访问级别和定义实体的源文件相关，并且也和源文件所属的模块相关。

- *Open 访问* 和 *public 访问* 允许实体被定义模块中的任意源文件访问，同样可以被另一模块的源文件通过导入该定义模块来访问。在指定框架的公共接口时，通常使用 `open` 或 `public` 访问。`open` 和 `public` 访问之间的区别将在之后给出；
- *Internal 访问* 允许实体被定义模块中的任意源文件访问，但不能被该模块之外的任何源文件访问。通常在定义应用程序或是框架的内部结构时使用。
- *File-private 访问* 将实体的使用限制于当前定义源文件中。当一些细节在整个文件中使用时，使用 `file-private` 访问隐藏特定功能的实现细节。

- `private` 访问将实体的使用限制于封闭声明中。当一些细节仅在单独的声明中使用时，使用 `private` 访问隐藏特定功能的实现细节。

`open` 访问是最高的（限制最少）访问级别，`private` 是最低的（限制最多）访问级别。

`open` 访问仅适用于类和类成员，它与 `public` 访问区别如下：

- `public` 访问，或任何更严格的访问级别的类，只能在其定义模块中被继承。
- `public` 访问，或任何更严格访问级别的类成员，只能被其定义模块的子类重写。
- `open` 类可以在其定义模块中被继承，也可在任何导入定义模块的其他模块中被继承。
- `open` 类成员可以被其定义模块的子类重写，也可以被导入其定义模块的任何模块重写。

显式地标记类为 `open` 意味着你考虑过其他模块使用该类作为父类对代码的影响，并且相应地设计了类的代码。

访问级别的指导准则

Swift 中的访问级别遵循一个总体指导准则：实体不可以被更低（限制更多）访问级别的实体定义。

例如：

- 一个 `public` 的变量其类型的访问级别不能是 `internal`, `file-private` 或是 `private`，因为在使用 `public` 变量的地方可能没有这些类型的访问权限。
- 一个函数不能比它的参数类型和返回类型访问级别高，因为函数可以使用的环境而其参数和返回类型却不能使用。

这个准则对此语言其他方面的明确含义已经详细的列在下面。

默认访问级别

如果你不指明访问级别的话，你的代码中的所有实体（以及本章后续提及的少数例外）都会默认为 `internal` 级别。因此，大多数情况下你不需要明确指定访问级别。

单目标应用的访问级别

当你编写一个简单的单目标应用时，你的应用中的代码都是在本应用中使用的并且不会在应用模块之外使用。默认的 `internal` 访问级别已经匹配了这种需求。因此，你不需要明确自定访问级别。但你可能会将代码的一些部分标注为 `file private` 或 `private` 以对模块中的其他代码隐藏它们的实现细节。

框架的访问级别

当你开发一个框架时，将该框架的面向公众的接口标注为 `open` 或 `public`，这样它就能被其他的模块看到或访问，比如导入该框架的应用。这个面向公众的接口就是该框架的应用编程接口（API）。

注意

你框架的任何内部实现细节仍可以使用 `internal` 默认访问级别，如果你想从框架的其他部分隐藏细节也可以将它们标注为 `private` 或 `file private`。仅当你想将它设为框架的 API 时你才能将实体标注为 `open` 或 `public`。

单元测试目标的访问级别

当你在写一个有单元测试目标的应用时，你的代码应该能被模块访问到以进行测试。默认情况下只有标注为 `open` 或 `public` 的才可以被其他模块访问。但是，如果你使用 `@testable` 属性标注了导入的生产模块并且用使能测试的方式编译了这个模块，单元测试目标就能访问任何 `internal` 的实体。

访问控制语法

通过在实体的引入之前添加 `open`，`public`，`internal`，`fileprivate`，或 `private` 修饰符来定义访问级别。

```
1 public class SomePublicClass{}  
2 internal class SomeInternalClass{}  
3 fileprivate class SomeFilePrivateClass{}  
4 private class SomePrivateClass{}  
5 public var somePublicVariable=0  
6 internal let someInternalConstant=0  
7 fileprivate func someFilePrivateFunction(){}
8 private func somePrivateFunction(){}
9
```

除非已经标注，否则都会使用默认的 `internal` 访问级别，这一点在[默认访问级别](#)一节已经说明。这意味着 `SomeInternalClass` 和 `someInternalConstant` 不需要指明访问级别也会是 `internal` 级别。

```
1 class SomeInternalClass{} // implicitly internal  
2 let someInternalConstant=0 // implicitly internal  
3
```

自定类型

如果你想给自定类型指明访问级别，那就在定义时指明。只要访问级别允许，新类型就可以被使用。例如，你定义了一个 `file-private` 的类，它就只能在定义文件中被当作属性类型、函数参数或返回类型使用。

类型的访问控制级别也会影响它的成员的默认访问级别（它的属性，方法，初始化方法，下标）。如果你将类型定义为 `private` 或 `file private` 级别，那么它的成员的默认访问级别也会是 `private` 或 `file private`。如果你将类型定义为 `internal` 或 `public` 级别（或直接使用默认级别而不显式指出），那么它的成员的默认访问级别会是 `internal`。

重要

`public` 的类型默认拥有 `internal` 级别的成员，而不是 `public`。如果你想让其中的一个类型成员是 `public` 的，你必须按实示例代码指明。这个要求确保类型的面向公众的 API 是你选择的，并且可以避免将类型的内部工作细节公开成 API 的失误。

```
1 public class SomePublicClass{// explicitly public class
2     public var somePublicProperty=0// explicitly public class member
3     var someInternalProperty=0// implicitly internal class member
4     fileprivate func someFilePrivateMethod(){}// explicitly file-private class member
5     private func somePrivateMethod(){}// explicitly private class member
6 }
7 class SomeInternalClass{// implicitly internal class
8     var someInternalProperty=0// implicitly internal class member
9     fileprivate func someFilePrivateMethod(){}// explicitly file-private class member
10    private func somePrivateMethod(){}// explicitly private class member
11 }
12 fileprivate class SomeFilePrivateClass{// explicitly file-private class
13     func someFilePrivateMethod(){}// implicitly file-private class member
14     private func somePrivateMethod(){}// explicitly private class member
15 }
16 private class SomePrivateClass{// explicitly private class
17     func somePrivateMethod(){}// implicitly private class member
18 }
19
20
21
```

元组类型

元组类型的访问级别是所有类型里最严格的。例如，如果你将两个不同类型的元素组成一个元组，一个元素的访问级别是 `internal`，另一个是 `private`，那么这个元组类型是 `private` 级别的。

注意

元组类型不像类、结构体、枚举和函数那样有一个单独的定义。元组类型的访问级别会在使用的时候被自动推断出来，不需要显式指明。

函数类型

函数类型的访问级别由函数成员类型和返回类型中的最严格访问级别决定。如果函数的计算访问级别与上下文环境默认级别不匹配，你必须在函数定义时显式指出。

下面的例子定义了一个称为 `someFunction()` 的全局函数，而没有指明它的访问级别。你或许以为它会是默认的“`internal`”级别，但事实不是这样。这样的 `someFunction()` 是无法通过编译的：

```
1 func someFunction()->(SomeInternalClass, SomePrivateClass){
2     // function implementation goes here
3 }
```

这个函数的返回类型是一个由两个在自定义类型里定义的类组成的元组。其中一个类是“`internal`”级别的，另一个是“`private`”。因此，这个元组的访问级别是“`private`”（元组成员的最严级别）。

由于返回类型是 `private` 级别的，你必须使用 `private` 修饰符使其合法：

```
1 private func someFunction()->(SomeInternalClass, SomePrivateClass){
2     // function implementation goes here
3 }
```

使用 public 或 internal 标注 someFunction() 的定义是无效的，使用默认的 internal 也是无效的，7的函数可能无法访问到 private 的函数返回值。

枚举类型

枚举中的独立成员自动使用该枚举类型的访问级别。你不能给独立的成员指明一个不同的访问级别。

在下面的例子中 CompassPoint 有一个指明的“public”级别。里面的成员 north , south , east , 和 west 因此是“public”：

```
1 public enum CompassPoint{  
2     case north  
3     case south  
4     case east  
5     case west  
6 }
```

原始值和关联值

枚举定义中的原始值和关联值使用的类型必须有一个不低于枚举的访问级别。例如，你不能使用一个 private 类型作为一个 internal 级别的枚举类型中的原始值类型。

嵌套类型

private 级别的类型中定义的嵌套类型自动为 private 级别。fileprivate 级别的类型中定义的嵌套类型自动为 fileprivate 级别。public 或 internal 级别的类型中定义的嵌套类型自动为 internal 级别。如果你想让嵌套类型是 public 级别的，你必须将其显式指明为 public。

子类

你可以继承任何类只要是在当前可以访问的上下文环境中。但子类不能高于父类的访问级别，例如，你不能写一个 internal 父类的 public 子类。

而且，你可以重写任何类成员（方法，属性，初始化器或下标），只要是在确定的访问域中是可见的。

重写可以让一个继承类成员比它的父类中的更容易访问。在下例中，public 级别的类 A 有一个 fileprivate 级别的 someMethod() 函数。B 是 A 的子类，有一个降低的“internal”级别。但是，类 B 对 someMethod() 函数进行了重写即改为“internal”级别，这比 someMethod() 的原本实现级别更高：

```
1 public class A{  
2     fileprivate func someMethod(){}
3 }
4 internal class B: A{
5     override internal func someMethod(){}
6 }
```

子类成员调用父类中比子类更低访问级别的成员，只要这个调用发生在一个允许的访问级别上下文中（即对 fileprivate 成员的调用要求父类在同一个源文件中，对 internal 成员的调用要求父类在同一个模块中）：

```
1 public class A{  
2     fileprivate func someMethod(){  
3 }  
4     internal class B: A{  
5         override internal func someMethod(){  
6             super.someMethod()  
7         }  
8     }  
9 }
```

因为父类 A 和子类 B 定义在同一个源文件中，那么 B 类可以在 someMethod() 中调用父类的 someMethod()。

常量，变量，属性和下标

常量、变量、属性不能拥有比它们类型更高的访问级别。例如，你不能写一个 `public` 的属性而它的类型是 `private` 的。类似的，下标也不能拥有比它的索引类型和返回类型更高的访问级别。

如果常量、变量、属性或下标由 `private` 类型组成，那么常量、变量、属性或下标也要被标注为 `private`：

```
1 private var privateInstance=SomePrivateClass()
```

Getters 和 Setters

常量、变量、属性和下标的 `getter` 和 `setter` 自动接收它们所属常量、变量、属性和下标的访问级别。

你可以给 `setter` 函数一个比相对应 `getter` 函数更低的访问级别以限制变量、属性、下标的读写权限。你可以通过在 `var` 和 `subscript` 的置入器之前书写 `fileprivate(set)`，`private(set)`，或 `internal(set)` 来声明更低的访问级别。

注意

这个规则应用于存储属性和计算属性。即使你没有给一个存储属性书写一个明确的 `getter` 和 `setter`，Swift 会为你合成一个 `getter` 和 `setter` 以访问到存储属性的隐式存储。使用 `fileprivate(set)`，`private(set)` 和 `internal(set)` 可以改变这个合成的 `setter` 的访问级别，同样也可以改变计算属性的访问级别。

下面的例子定义了一个称为 `TrackedString` 的结构体，它保持追踪一个字符串属性的修改次数：

```
1 struct TrackedString{  
2     private(set) var numberOfEdits=0  
3     var value:String=""  
4     didSet{  
5         numberOfEdits+=1  
6     }  
7 }  
8 }
```

TrackedString 结构体定义了一个可存储字符串的属性 value，它又一个初始值 ""（空字符串）。这个结构图同样定义了一个可存储整数的属性 numberOfEdits，它被用于记录 value 的修改次数。这个记录由 value 属性中的 didset 属性实现，它会增加 numberOfEdits 的值一旦 value 被设为一个新值。

TrackedString 结构体和 value 属性都没有显式指出访问级别修饰符，因此它们都遵循默认的 internal 级别。numberOfEdits 属性的访问级别已经标注为 private(set) 以说明这个属性的 getter 是默认的 internal 级别，但是这个属性只能被 TrackedString 内的代码设置。这允许 TrackedString 在内部修改 numberOfEdits 属性，而且可以展示这个属性作为一个只读属性当在结构体定义之外使用时——包括 TrackedString 的扩展。

如果你创建了一个 TrackedString 的实例并修改了几次字符串的值，你可以看到 numberOfEdits 属性的值更新到匹配修改的次数：

```
1 var stringToEdit=TrackedString()  
2 stringToEdit.value="This string will be tracked."  
3 stringToEdit.value+=" This edit will increment numberOfEdits."  
4 stringToEdit.value+=" So will this one."  
5 print("The number of edits is \(stringToEdit.numberOfEdits)")  
6 // Prints "The number of edits is 3"
```

尽管你可以从别的源文件中询问到 numberOfEdits 属性的当前值，但你不能从别的源文件中修改该属性的值。这个限制保护了 TrackedString 编辑追踪功能的实现细节，并同时为该功能的一个方面提供方便的访问。

你若有必要也可以显式指明 getter 和 setter 方法。下面的例子提供了一个定义为 public 级别的 TrackedString 结构体。结构体成员（包括 numberOfEdits 属性）因此有一个默认的 internal 级别。你可以设置 numberOfEdits 属性的 getter 方法为 public，setter 方法为 private 级别，通过结合 public 和 private(set) 访问级别修饰符：

```
1 public struct TrackedString{  
2     public private(set) var numberOfEdits = 0  
3     public var value: String = ""{  
4         didSet{  
5             numberOfEdits += 1  
6         }  
7     }  
8     public init(){  
9 }
```

初始化器

我们可以给自定义初始化方法设置一个低于或等于它的所属的类的访问级别。唯一的例外是必要初始化器（定义在必要初始化器）。必要初始化器必须和它所属类的访问级别一致。

就像函数和方法的参数一样，初始化器的参数类型不能比初始化方法的访问级别还低。

默认初始化器

正如默认初始化器中描述的那样，Swift 自动为任何结构体和类提供一个无参数的默认初始化方法，以给它的属性提供默认值但不会提供给初始化器自身。

默认初始化方法与所属类的访问级别一致，除非该类型定义为 `public`。如果一个类定义为 `public`，那么默认初始化方法为 `internal` 级别。如果你想一个 `public` 类可以被一个无参初始化器初始化当在另一个模块中使用时，你必须显式提供一个 `public` 的无参初始化方法。

结构体的默认成员初始化器

如果结构体的存储属性时 `private` 的，那么它的默认成员初始化方法就是 `private` 级别。如果结构体的存储属性时 `file private` 的，那么它的默认成员初始化方法就是 `file private` 级别。否则就是默认的 `internal` 级别。正如以上默认初始化的描述，如果你想在另一个模块中使用结构体的成员初始化方法，你必须提供在定义中提供一个 `public` 的成员初始化方法。

协议

如果你想给一个协议类型分配一个显式的访问级别，那就在定义时指明。这让你创建的协议可以在一个明确的访问上下文中被接受。

协议定义中的每一个要求的访问级别都自动设为与该协议相同。你不能将一个协议要求的访问级别设为与协议不同。这保证协议的所有要求都能被接受该协议的类型所见。

注意如果你定义了一个 `public` 的协议，该协议的规定要求在被实现时拥有一个 `public` 的访问级别。这个行为不同于其他类型，一个 `public` 的类型的成员时 `internal` 访问级别。

协议继承

如果你定义了一个继承已有协议的协议，这个新协议最高与它继承的协议访问级别一致。例如你不能写一个 `public` 的协议继承一个 `internal` 的协议。

协议遵循

类型可以遵循更低访问级别的协议。例如，你可以定义一个可在其他模块使用的 `public` 类型，但它就只能在定义模块中使用如果遵循一个 `internal` 的协议。

遵循了协议的类的访问级别取这个协议和该类的访问级别的最小者。如果这个类型是 `public` 级别的，它所遵循的协议是 `internal` 级别，这个类型就是 `internal` 级别的。

当你写或是扩张一个类型以遵循协议时，你必须确保该类按协议要求的实现方法与该协议的访问级别一致。例如，一个 `public` 的类遵循一个 `internal` 协议，该类的方法实现至少是“`internal`”的。

注意

在 Swift 和 Objective-C 中协议遵循是全局的——一个类不可能在一个程序中用不同方法遵循一个协议。

扩展

你可以在任何可访问的上下文环境中对类、结构体、或枚举进行扩展。在扩展中添加的任何类型成员都有着被扩展类型相同的访问权限。如果你扩展一个公开或者内部类型，你添加的任何新类型成员都拥有默认的内部访问权限。如果你扩展一个文件内私有的类型，你添加的

任何新类型成员都拥有默认的私有访问权限。如果你扩展一个私有类型，你添加的任何新类型成员都拥有默认的私有访问权限。

或者，你可以显式标注扩展的访问级别（例如，`private extension`）已给扩展中的成员设置新的默认访问级别。这个默认同样可以在扩展中为单个类型成员重写。

你不能给用于协议遵循的扩展显式标注访问权限修饰符。相反，在扩展中使用协议自身的访问权限作为协议实现的默认访问权限。

扩展中的私有成员

在同一文件中的扩展比如类、结构体或者枚举，可以写成类似多个部分的类型声明。你可以：

- 在原本的声明中声明一个私有成员，然后在同一文件的扩展中访问它；
- 在扩展中声明一个私有成员，然后在同一文件的其他扩展中访问它；
- 在扩展中声明一个私有成员，然后在同一文件的原本声明中访问它。

这样的行为意味着你可以和组织代码一样使用扩展，无论你的类型是否拥有私有成员。比如说，假设下面这样的简单协议：

```
1 protocolSomeProtocol{  
2 funcdoSomething()  
3 }
```

你可以使用扩展来添加协议遵循，比如这样：

```
1 structSomeStruct{  
2 private varprivateVariable=12  
3 }  
4 extensionSomeStruct: SomeProtocol{  
5 funcdoSomething(){  
6 print(privateVariable)  
7 }  
8 }  
9
```

泛型

泛指类型和泛指函数的访问级别取泛指类型或函数以及泛型类型参数的访问级别的最小值。

类型别名

任何你定义的类型同义名都被视为不同的类型以进行访问控制。一个类型同义名的访问级别不高于原类型。例如，一个 `private` 的类型同义名可联系到 `private`，`file-private`，`internal`，`public` 或 `open` 的类型，但 `public` 的类型同义名不可联系到 `internal`，`file-private` 或 `private` 类型。

注意

这条规则适用于为满足协议遵循而给类型别名关联值的情况。

高级运算符

 cnswift.org/advanced-operators

作为基本运算符的补充，Swift 提供了一些对值进行更加复杂操作的高级运算符。这些运算包括你在 C 或 Objective-C 所熟悉的所有按位和移位运算符。

与 C 的算术运算符不同，Swift 中算术运算符默认不会溢出。溢出行为都会作为错误被捕获。要允许溢出行为，可以使用 Swift 中另一套默认支持的溢出运算符，比如溢出加法运算符（`&+`）。所有这些溢出运算符都是以（`&`）符号开始的。

当你定义了自己的结构体、类以及枚举的时候，那么为这些自定义类型也提供 Swift 标准的运算符就很有必要。Swift 简化了这些运算符的定制实现并且精确地确定了你创建的每个类型的运算符所具有的行为。

你不会被限制在预定义的运算符里。Swift 允许你自由地定义你自己的中缀、前缀、后缀和赋值运算符，以及相对应的优先级和结合性。这些运算符可以像预先定义的运算符一样在你的代码里使用和采纳，甚至你可以扩展已存在的类型来支持你自己定义的运算符。

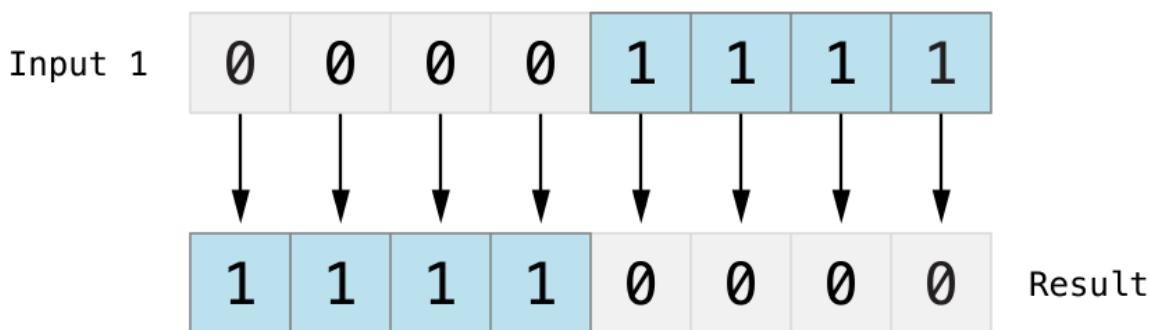
位运算符

位运算符可以操作数据结构中每一个独立的位。它们通常被用在底层开发中，比如图形编程和创建设备驱动。位运算符在处理外部资源的原始数据时也非常有用，比如为自定义的通信协议的数据进行编码和解码。

Swift 支持 C 里面所有的位运算符，具体如下：

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：



位取反运算符是一个前缀运算符，需要直接放在运算符的前面，并且不能有空格：

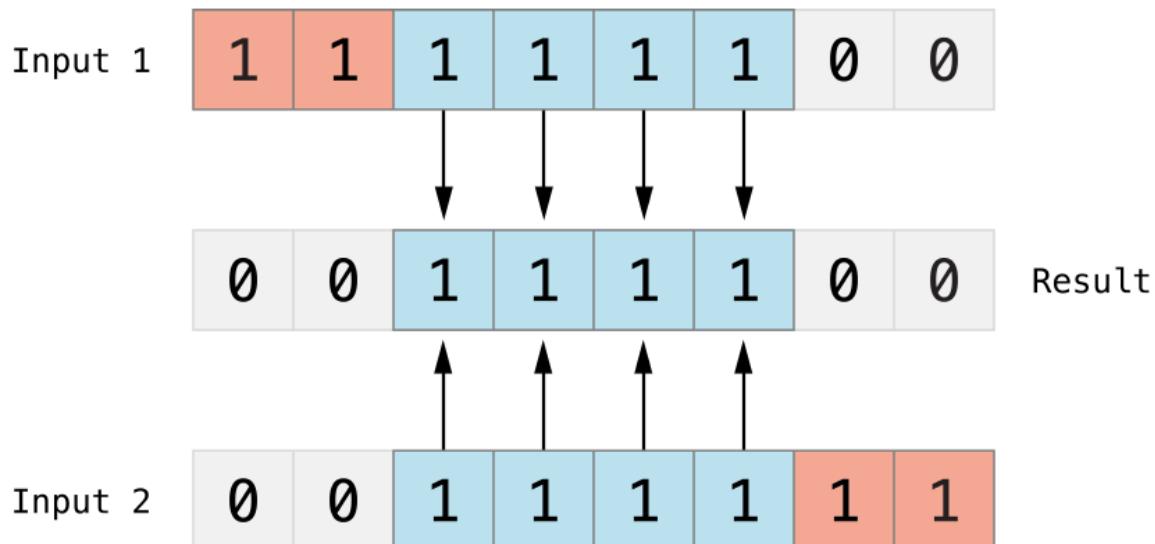
```
1 let initialBits: UInt8=0b00001111
2 let invertedBits=~initialBits// equals 11110000
```

UInt8 类型的整数有八位，可以存储 0 到 255 之间的任意值。这个例子使用二进制值 00001111 初始化了一个 UInt8 类型的整数，前四位全是 0，后四位都是 1。这和十进制的 15 是相等的。

然后使用位取反运算符创建一个新的常量名为 invertedBits，它和 initialBits 相等，但是所有位都被取反了。0 变为了 1，1 变为了 。 invertedBits 的值是 11110000，和无符号十进制整数 240 相等。

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。

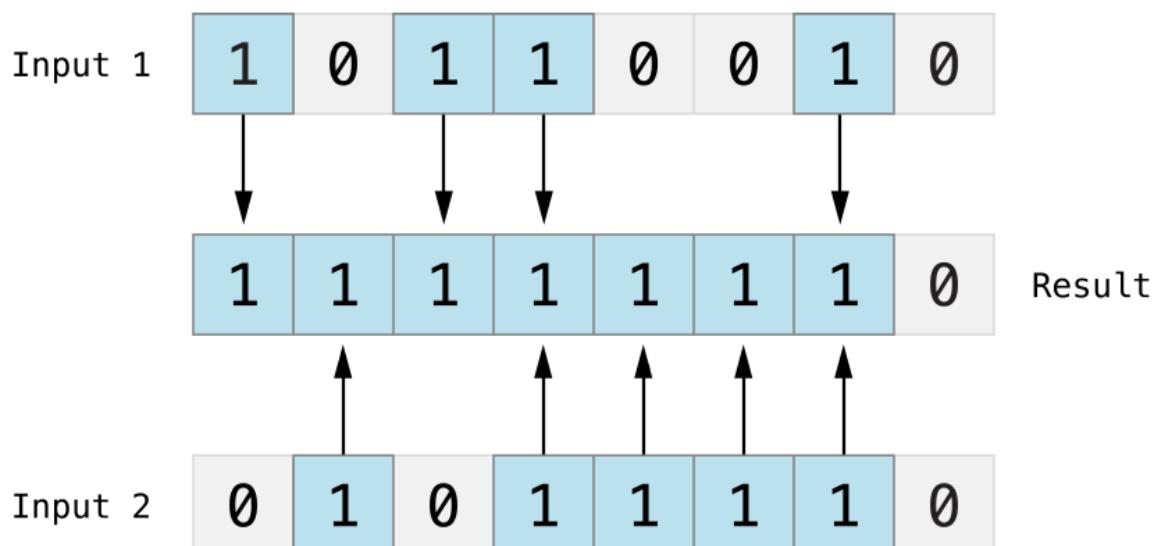


在下面的例子中，firstSixBits 和 lastSixBits 的中间四个位都为 1。按位与可以把它们合并为一个新值 00111100，对应十进制的值为 60。

```
1 let firstSixBits: UInt8=0b11111100
2 let lastSixBits: UInt8=0b00111111
3 let middleFourBits=firstSixBits&lastSixBits// equals 00111100
```

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：

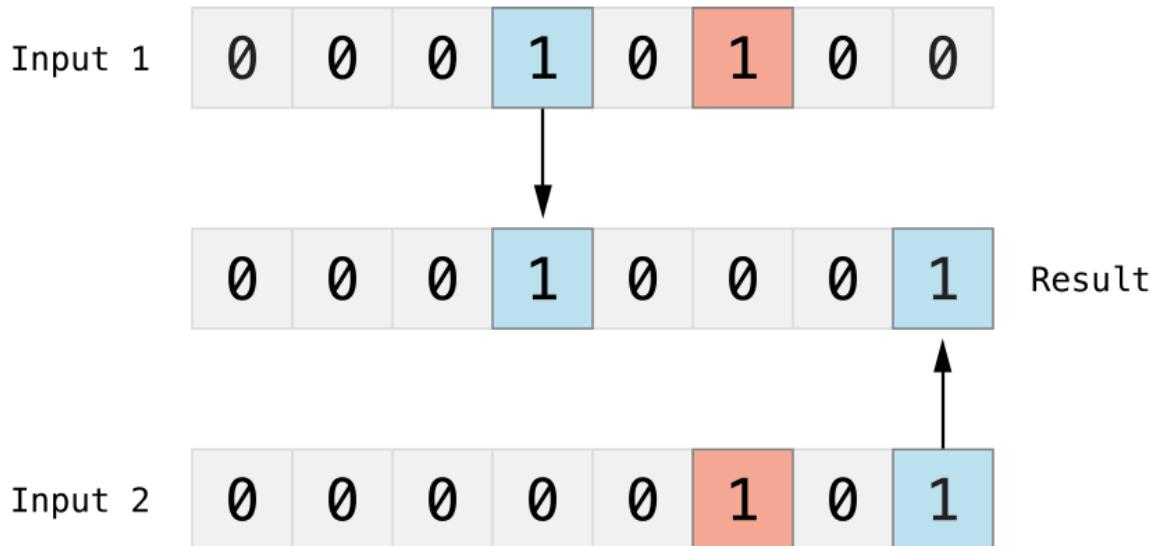


在下面的例子中，someBits 和 moreBits 在不同的位设置了 1。位或运算符把它们合并为 11111110，等于十进制无符号整数 154。

```
1 let someBits: UInt8=0b10110010
2 let moreBits: UInt8=0b01011110
3 let combinedBits=someBits|moreBits// equals 11111110
```

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：



在下面的例子中，firstBits 和 otherBits 的值有一位设置为 1，而对方设置为 。位异或运算符会将这两个位上的值设置为 1，firstBits 和 otherBits 其他位都设置为了 。

```
1 let firstBits: UInt8=0b00010100
2 let otherBits: UInt8=0b00000101
3 let outputBits=firstBits^otherBits// equals 00010001
```

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，但是需要遵守下面定义的规则。

位左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。

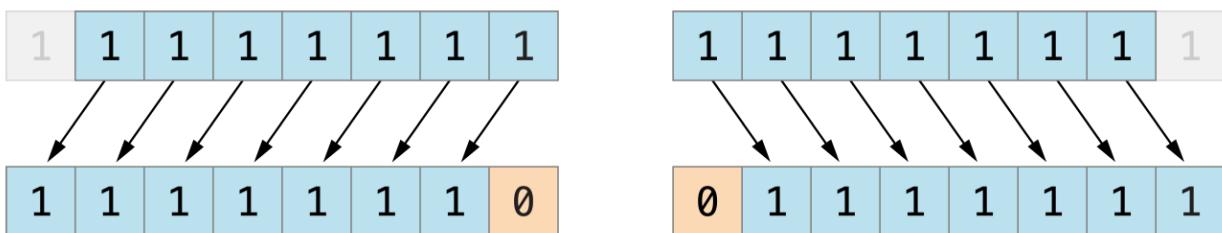
无符号整数的移位操作

对无符号整数的移位规则如下：

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

这种方法称就是所谓的逻辑移位。

下图展示了 $11111111 \ll 1$ (即把 11111111 向左移 1 位) , $11111111 \gg 1$ (即把 11111111 向右移 1 位) , 蓝色的数字是被移位的, 灰色的数字被舍弃, 橙色的数字 0 是新插入的 :



下面的代码展示了 Swift 的移位操作 :

```
1 letshiftBits:UInt8=4// 00000100 in binary
2 shiftBits<<1// 00001000
3 shiftBits<<2// 00010000
4 shiftBits<<5// 10000000
5 shiftBits<<6// 00000000
6 shiftBits>>2// 00000001
```

可以使用移位操作对其他的数据类型进行编码和解码 :

```
1 letpink:UInt32=0xCC6699letredComponent=(pink&0xFF0000)>>16// redComponent is 0xCC, or
204 let greenComponent = (pink & 0x00FF00) >> 8 // greenComponent is 0x66, or 102 let
blueComponent = pink & 0x0000FF // blueComponent is 0x99, or 153
```

这个示例使用了一个命名为 pink 的 UInt32 常量来存储层叠样式表^[1]中粉色的颜色值。该 CSS 的颜色值 #CC6699 , 在 Swift 中表示为十六进制 0xCC6699 。然后利用位与运算符 (&) 和位右移运算符 ($>>$) 从这个颜色值中分解出红 (CC) 、绿 (66) 以及蓝 (99) 三个部分。

红色部分是通过对 0xCC6699 和 0xFF0000 进行按位与运算后得到的。 0xFF0000 中的 0 部分作为掩码，掩盖了 0xCC6699 中的第二和第三个字节，使得数值中的 6699 被忽略，只留下 0xCC0000 。

然后，再将这个数按向右移动 16 位 ($>>16$) 。十六进制中每两个字符表示 8 个比特位，所以移动 16 位后 0xCC0000 就变为 0x0000CC 。这个数和 0xCC 是等同的，也就是十进制数值的 204 。

同样的，绿色部分通过对 0xCC6699 和 0x00FF00 进行按位与运算得到 0x006600 。然后将这个数向右移动 8 位，得到 0x66 ，也就是十进制数值的 102 。

最后，蓝色部分通过对 0xCC6699 和 0x0000FF 进行按位与运算得到 0x000099 。并且不需要进行向右移位，所以结果为 0x99 ，也就是十进制数值的 153 。

译注

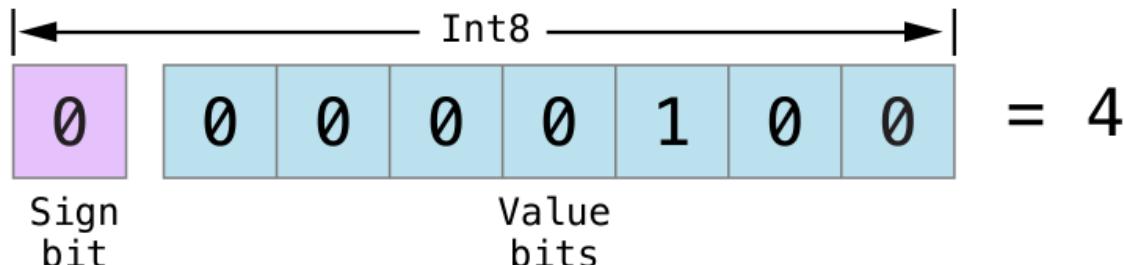
[1] 层叠样式表 (Cascading Style Sheets) , 即 CSS 。

有符号整型的位移操作

对比无符号整型来说有符整型的移位操作相对复杂得多，这种复杂性源于有符号整数的二进制表现形式。（为了简单起见，以下的示例都是基于 8 位有符号整数的，但是其中的原理对大小的有符号整数都是一样的。）

有符号整型使用它的第一位（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。

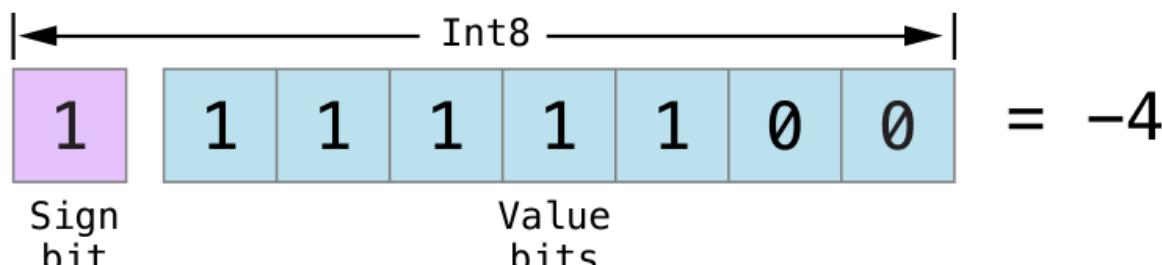
其余的位数（所谓的数值位）存储了实际的值。有符号正整数和无符号数的存储方式是一样的，都是从 0 开始算起。这是值为 4 的 Int8 型整数的二进制位表现形式：



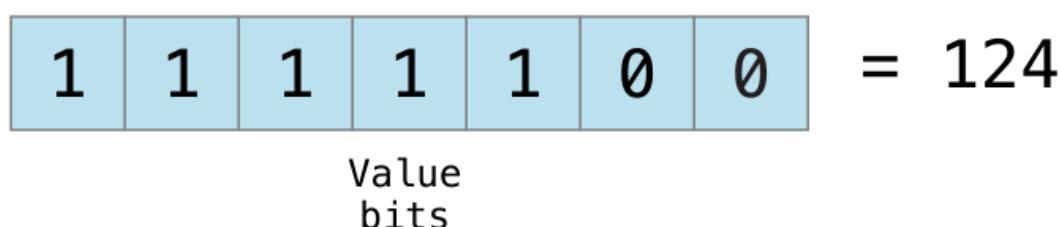
符号位是 0（意味着是一个正数），另外七位则代表了十进制数值 4 的二进制表示。

但是负数的存储方式略有不同。它存储的是 2^n 次方减去它的绝对值，这里的 n 为数值位的位数。一个 8 位的数有七个数值位，所以是 2^7 次方，或者说 128。

这是值为 -4 的 Int8 型整数的二进制位表现形式：

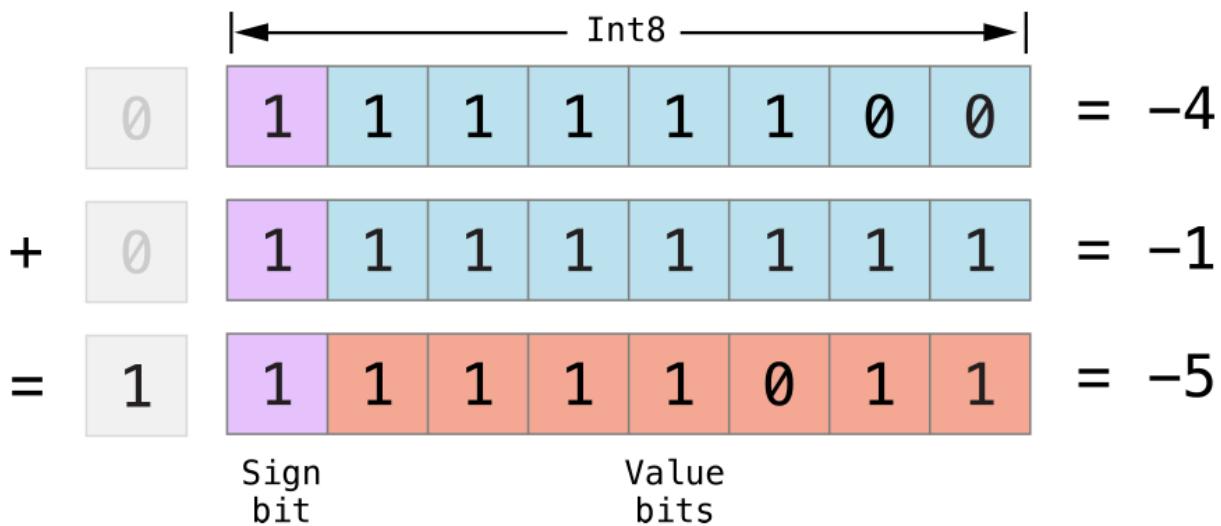


这次，符号位为 1（说明是负数），另外七个位则代表了数值 124（即 $128-4$ ）的二进制表示：

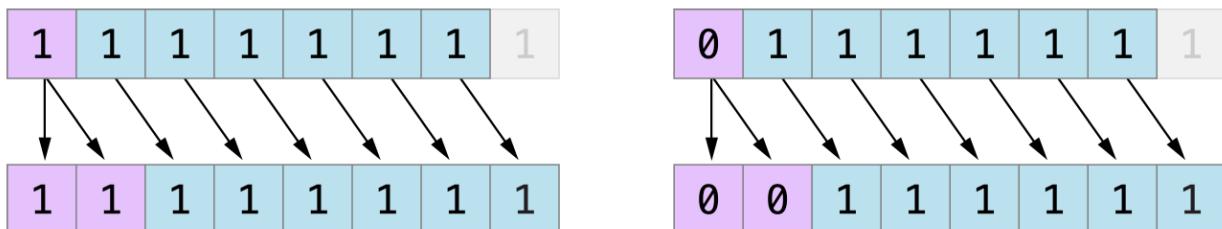


负数的编码就是所谓的二进制补码表示。用这种方法来表示负数乍看起来有点奇怪，但它有几个优点。

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：



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。



这个行为可以确保有符号整数的符号位不会因为右移操作而改变，这就是所谓的算术移位。

由于正数和负数的特殊存储方式，在对它们进行右移的时候，会使它们越来越接近零。在移位的过程中保持符号位不变，意味着负整数在接近零的过程中会一直保持为负。

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。这个行为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。

例如，Int16 整数能容纳的有符号整数范围是 -32768 到 32767，当为一个 Int16 型变量赋的值超过这个范围时，系统就会报错：

```

1 var potentialOverflow=Int16.max
2 // potentialOverflow equals 32767, which is the maximum value an Int16 can hold
3 potentialOverflow+=1
4 // this causes an error

```

为过大或者过小的数值提供错误处理，能让我们在处理边界值时更加灵活。

总之，当你故意想要溢出来截断可用位的数字时，也可以选择这么做而非报错。Swift 提供三个算数溢出运算符来让系统支持整数溢出运算。这些运算符都是以 & 开头的：

- 溢出加法 (&+)
- 溢出减法 (&-)

- 溢出乘法 (&*)

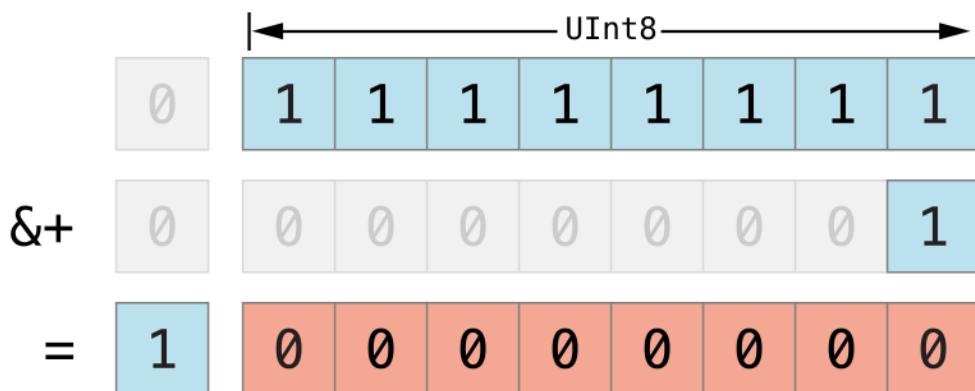
值溢出

数值可能出现向上溢出或向下溢出。

这个示例演示了当对一个无符号整数使用溢出加法 (&+) 进行上溢运算时会发生什么：

```
1 var unsignedOverflow=UInt8.max
2 // unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
3 unsignedOverflow=unsignedOverflow&+1
4 // unsignedOverflow is now equal to 0
```

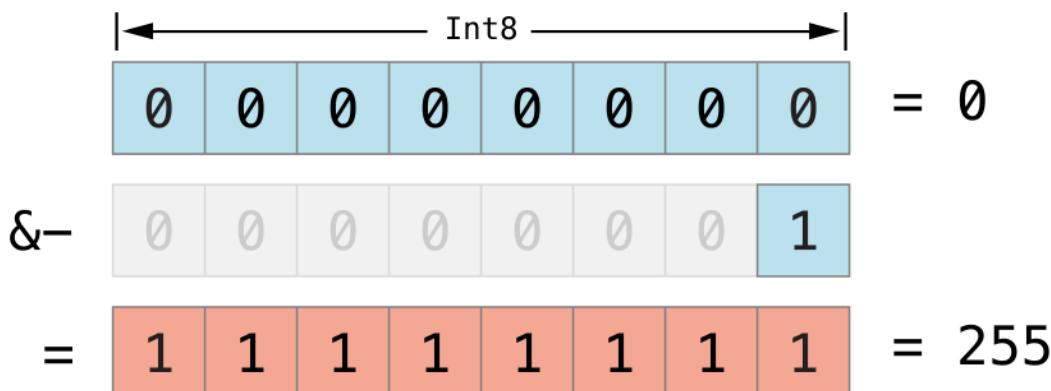
`unsignedOverflow` 初始化为 `UInt8` 所能容纳的最大整数 (255 , 二进制为 11111111) 。溢出加法运算符 (&+) 对其进行加 1 操作。这使得它的二进制表示正好超出 `UInt8` 所能容纳的位数，也就导致它溢出了边界，如下图所示。溢出后，留在 `UInt8` 边界内的值是 00000000 ，也就是十进制数值的 0 。



同样地，当我们对一个无符号整数使用溢出减法 (&-) 进行下溢运算时也会产生类似的现象：

```
1 var unsignedOverflow=UInt8.min
2 // unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
3 unsignedOverflow=unsignedOverflow&-1
4 // unsignedOverflow is now equal to 255
```

`UInt8` 型整数能容纳的最小值是 00000000 ，以二进制表示即 00000000 。当使用溢出减法运算符 (&-) 对其进行减 1 操作时，数值会产生下溢并被截断为 11111111 ，也就是十进制数值的 255 。

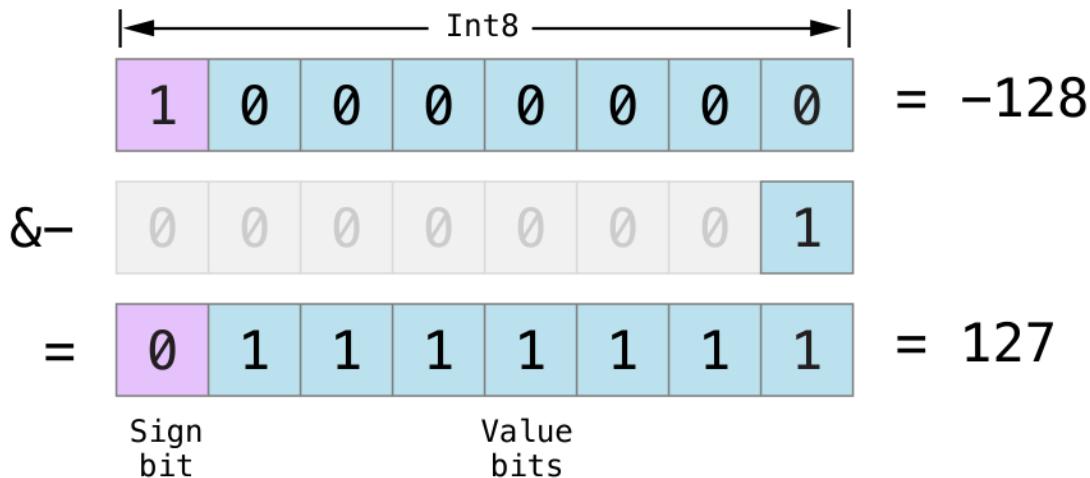


溢出也会发生在有符号整型数值上。正如按位左移/右移运算符所描述的，在对有符号整型数

值进行溢出加法或溢出减法运算时，符号位也需要参与计算。

```
1 var signedOverflow=Int8.min  
2 // signedOverflow equals -128, which is the minimum value an Int8 can hold  
3 signedOverflow=signedOverflow&-1  
4 // signedOverflow is now equal to 127
```

Int8 整数能容纳的最小值是 -128，以二进制表示即 10000000。当使用溢出减法运算符对其进行减 1 操作时，符号位翻转，得到二进制数值 01111111，也就是十进制数值的 127，这个值也是 Int8 型整数所能容纳的最大值。



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。同样地，当发生下溢时，它们会从所能容纳的最小数变成最大的数。

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”。

在复合表达式的运算顺序中，运算符的优先级和结合性是非常重要的。举例来说，为什么下面这个表达式的运算结果是 17？

```
1 2+3%4*5  
2 // this equals 17
```

如果严格地从左到右进行运算，则运算的过程是这样的：

- $2 + 3 = 5$
- $5 \% 4 = 1$
- $1 * 5 = 5$

然而正确的答案是 17，而不是 5。优先级高的运算符要先于优先级低的运算符进行计算。与 C 语言类似，在 Swift 中，取余运算符（%）和乘法运算符（*）的优先级高于加法运算符（+）。因此，它们的计算顺序要先于加法运算。

但是，取余和乘法具有相同的优先级。这时为了得到正确的运算顺序，还需要考虑结合性，乘法与取余运算都是左结合的。可以将这考虑成为这两部分表达式都隐式地加上了括号：

```
1 2+((3%4)*5)
```

(3%4) 是 3，所以表达式等价于：

```
1 2+(3*5)
```

(3*5) 是 15，所以表达式等价于：

```
1 2+15
```

此时可以容易地看出计算的结果为 17。

如果想查看完整的 Swift 运算符优先级和结合性规则，请参考[表达式](#)。以及[Swift 标准库中的运算符](#)。

注意

对于 C 和 Objective-C 来说，Swift 的运算符优先级和结合性规则是更加简洁和可预测的。但是，这也意味着它们于那些基于 C 的语言不是完全一致的。在对现有的代码进行移植的时候，要注意确保运算符的行为仍然是按照你所想的那样去执行。

运算符函数

类和结构体可以为现有的运算符提供自定义的实现，这通常被称为运算符重载。

下面的例子展示了如何为自定义的结构实现加法运算符(+)。算术加法运算符是一个二元运算符，因为它可以对两个目标进行操作，同时它还是中缀运算符，因为它出现在两个目标中间。

例子中定义了一个名为 Vector2D 的结构体用来表示二维坐标向量 (x,y)，紧接着定义了一个可以对两个 Vector2D 结构体进行相加的运算符方法：

```
1 structVector2D{  
2     varx=0.0,y=0.0  
3 }  
4 extensionVector2D{  
5     staticfunc+(left:Vector2D,right:Vector2D)->Vector2D{  
6         returnVector2D(x:left.x+right.x,y:left.y+right.y)  
7     }  
8 }  
9
```

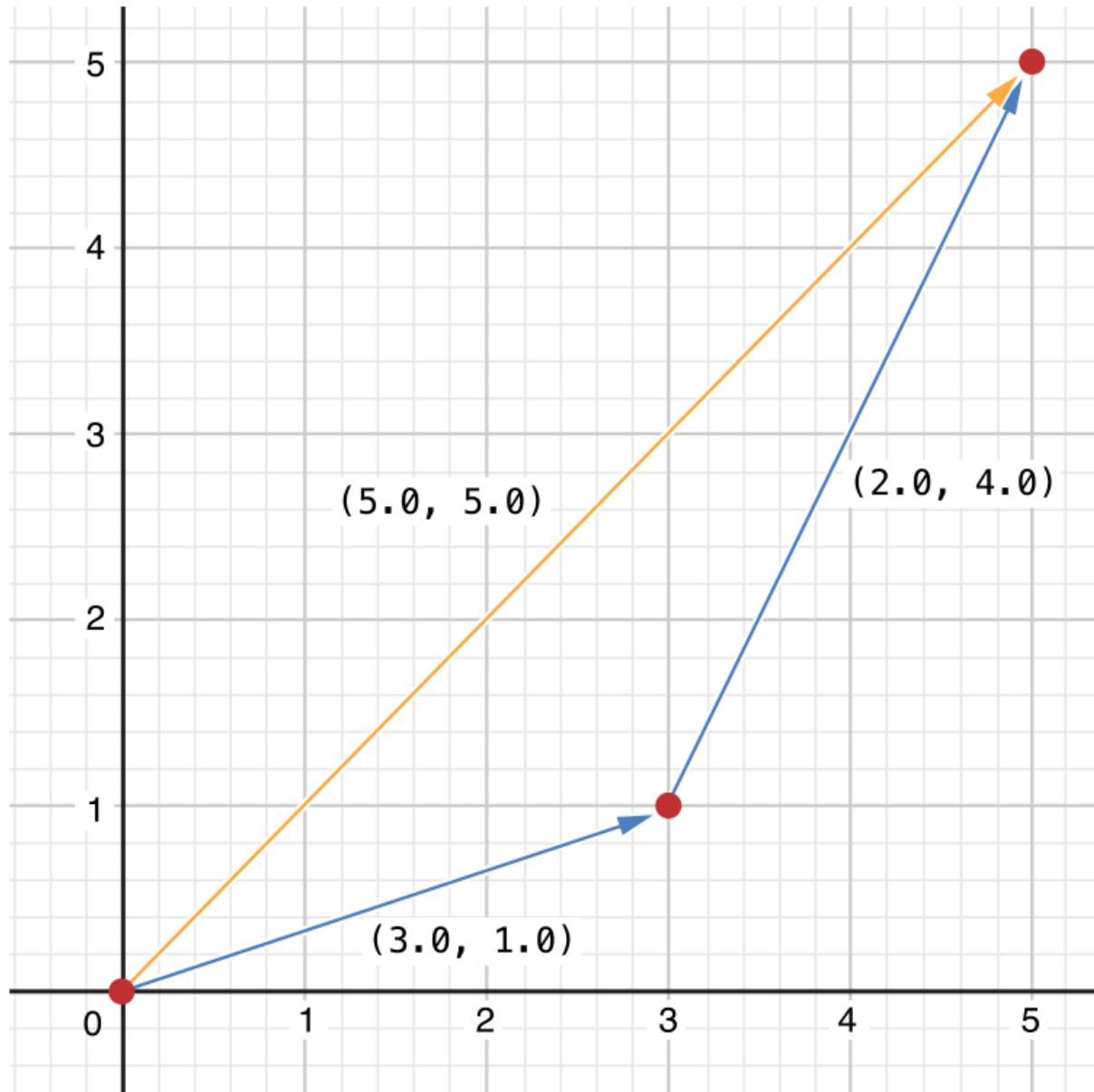
该运算符函数被定义为一个全局函数，并且函数的名字与它要进行重载的 + 名字一致。因为算术加法运算符是双目运算符，所以这个运算符函数接收两个类型为 Vector2D 的输入参数，同时有一个 Vector2D 类型的返回值。

在这个实现中，输入参数分别被命名为 left 和 right，代表在 + 运算符左边和右边的两个 Vector2D 对象。函数返回了一个新的 Vector2D 的对象，这个对象的 x 和 y 分别等于两个参数对象的 x 和 y 的值之和。

这个函数被定义成全局的，而不是 Vector2D 结构的成员方法，所以任意两个 Vector2D 对象都可以使用这个中缀运算符：

```
1 let vector=Vector2D(x:3.0,y:1.0)
2 let anotherVector=Vector2D(x:2.0,y:4.0)
3 let combinedVector=vector+anotherVector
4 // combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

这个例子实现两个向量 (3.0 , 1.0) 和 (2.0 , 4.0) 的相加，并得到新的向量 (5.0 , 5.0)。这个过程如下图示：



前缀和后缀运算符

上个例子演示了一个二元中缀运算符的自定义实现。类与结构体也能提供标准一元运算符的实现。单目运算符只有一个操作目标。当运算符出现在目标之前，它就是前缀(比如 `-a`)，当它出现在操作目标之后时，它就是后缀运算符(比如 `b!`)。

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：

```
1 extensionVector2D{  
2     staticprefixfunc-(vector:Vector2D)->Vector2D{  
3         returnVector2D(x:-vector.x,y:-vector.y)  
4     }  
5 }
```

这段代码为 Vector2D 类型实现了单目减运算符 (-a)。由于单目减运算符是前缀运算符，所以这个函数需要加上 prefix 限定符。

对于简单数值，单目减运算符可以对它们的正负性进行改变。对于 Vector2D 来说，单目减运算将其 x 和 y 属性的正负性都进行了改变。

```
1 letpositive=Vector2D(x:3.0,y:4.0)  
2 letnegative=-positive  
3 // negative is a Vector2D instance with values of (-3.0, -4.0)  
4 letalsoPositive=-negative  
5 // alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

组合赋值运算符

组合赋值运算符将赋值运算符(=)与其它运算符进行结合。比如，将加法与赋值结合成加法赋值运算符 (+=)。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。

下面的例子实现了一个 Vector2D 的加赋值运算符：

```
1 extensionVector2D{  
2     staticfunc+=(left:inoutVector2D,right:Vector2D){  
3         left=left+right  
4     }  
5 }
```

因为加法运算在之前已经定义过了，所以在里无需重新定义。在这里可以直接利用现有的加法运算符函数，用它来对左值和右值进行相加，并再次赋值给左值：

```
1 varoriginal=Vector2D(x:1.0,y:2.0)  
2 letvectorToAdd=Vector2D(x:3.0,y:4.0)  
3 original+=vectorToAdd  
4 // original now has values of (4.0, 6.0)
```

注意

不能对默认的赋值运算符 (=) 进行重载。只有组合赋值运算符可以被重载。同样地，也无法对三元条件运算符 a?b:c 进行重载。

等价运算符

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：

```
1 extensionVector2D: Equatable{  
2     staticfunc==(left:Vector2D,right:Vector2D)->Bool{  
3         return(left.x==right.x)&&(left.y==right.y)  
4     }  
5 }
```

上面的例子实现了一个“等于”运算符（`==`）来检查两个 `Vector2D` 实例是否拥有相同的值。在 `Vector2D` 上下文中，“等于”作为“两个实例都具有相同的 x 值和 y 值”是有意义的，因此这个逻辑用作运算符的实现。标准库提供了一个关于“不等于”运算符（`!=`）的默认实现，它仅仅返回“等于”运算符的相反值。

现在你就可以用这些运算符来检查 `Vector2D` 实例是否等价了：

```
1 lettwoThree=Vector2D(x:2.0,y:3.0)  
2 letanotherTwoThree=Vector2D(x:2.0,y:3.0)  
3 iftwoThree==anotherTwoThree{  
4     print("These two vectors are equivalent.")  
5 }  
6 // Prints "These two vectors are equivalent."
```

Swift 为以下自定义类型提等价运算符供合成实现：

- 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。

在类型原本的声明中声明遵循 `Equatable` 来接收这些默认实现。

下面为三维位置向量 (`x,y,z`) 定义的 `Vector3D` 结构体，与 `Vector2D` 类似，由于 `x`，`y` 和 `z` 属性都是 `Equatable` 类型，`Vector3D` 就收到默认的等价运算符实现了。

```
1 structVector3D: Equatable{  
2     varx=0.0,y=0.0,z=0.0  
3 }  
4 lettwoThreeFour=Vector3D(x:2.0,y:3.0,z:4.0)  
5 letanotherTwoThreeFour=Vector3D(x:2.0,y:3.0,z:4.0)  
6 iftwoThreeFour==anotherTwoThreeFour{  
7     print("These two vectors are also equivalent.")  
8 }  
9 // Prints "These two vectors are also equivalent."  
10
```

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）。可以用来自定义运算符的字符列表请参考运算符

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：

```
1 prefixoperator+++{}
```

上面的代码定义了一个新的名为 `+++` 的前缀运算符。这个运算符在 Swift 中并没有意义，我们针对 `Vector2D` 的实例来赋予它意义。对这个例子来讲，`+++` 作为“前缀翻倍”运算符。它让 `Vector2D` 实例的 `x` 属性和 `y` 属性的值翻倍，使用前面定义的复合加法运算符来让向量对自身进行相加。要实现`+++`运算符，添加一个叫做 `+++` 的类型方法到 `Vector2D`：

```
1 extension Vector2D{  
2     staticprefixfunc +++(vector:inout Vector2D)->Vector2D{  
3         vector+=vector  
4         return vector  
5     }  
6 }  
7 var toBeDoubled=Vector2D(x:1.0,y:4.0)  
8 let afterDoubling=+++toBeDoubled  
9 // toBeDoubled now has values of (2.0, 8.0)  
10 // afterDoubling also has values of (2.0, 8.0)  
11
```

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。优先级和结合性中详细阐述了这两个特性是如何对中缀运算符的运算产生影响的。

结合性（ associativity ）可取的值有 `left` , `right` 和 `none` 。当左结合运算符跟其他相同优先级的左结合运算符写在一起时，会跟左边的操作数进行结合。同理，当右结合运算符跟其他相同优先级的右结合运算符写在一起时，会跟右边的操作数进行结合。而非结合运算符不能跟其他相同优先级的运算符写在一起。

`associativity` 的默认值是 `none` , `precedence` 默认为 100 。

下面例子定义了一个新的自定义中缀运算符 `+-` , 此运算符是 `left` 结合的，优先级为 140 :

```
1 infixoperator+-{associativity left precedence 140}  
2 extension Vector2D{  
3     staticfunc +- (left:Vector2D,right:Vector2D)->Vector2D{  
4         return Vector2D(x:left.x+right.x,y:left.y-right.y)  
5     }  
6 }  
7 let firstVector=Vector2D(x:1.0,y:2.0)  
8 let secondVector=Vector2D(x:3.0,y:4.0)  
9 let plusMinusVector=firstVector+-secondVector  
10 // plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

这个运算符把两个向量的 `x` 值相加，同时用第一个向量的 `y` 值减去第二个向量的 `y` 值。因为它本质上是属于“加”运算符，所以将它的结合性和优先级被设置与 `+` 和 `-` 等默认的中缀加型运算符是相同的（ `left` 和 140 ）。完整的 Swift 运算符默认结合性与优先级请参考[Swift 标准库运算符引用](#)。

注意

当定义前缀与后缀运算符的时候，我们并没有指定优先级。然而，如果对同一个操作数同时使用前缀与后缀运算符，则后缀运算符会先被应用。