

# 关于 Swift

Swift 是一门用于开发 iOS 和 OS X 应用程序的新语言，基于 C 和 Objective-C，但是没有 C 兼容性的限制。Swift 采用安全的编程模式，并增加了许多新的现代模式，让编程更加的容易、灵活，让编程更加有乐趣。Swift 被目前成熟并很受欢迎的 Cocoa 和 Cocoa Touch 支持，是重新思考如何做软件开发的时候了。

Swift 已经经过多年的酝酿。苹果公司通过改进现有的编译器，调试器和基本框架奠定了 Swift 的基础。我们通过自动引用计数（ARC）简化了存储管理。我们的框架协议栈，建立在坚实的基础框架和 Cocoa 之上，并且更加新潮和标准化。Objective-C 天生已经实现模块化，允许框架使用新的编程语言。由于这个基础，我们现在可以引入新的语言开发未来的苹果软件。

Objective-C 的程序员对 Swift 会很容易上手，因为它采用 Objective-C 相同的命名规则以及 Objective-C 的动态对象模型。Swift 可以无缝的调用 Cocoa 框架，并且可以和 Objective-C 混编。Swift 引入了许多新的特性，统一了编程语言程序和面向对象部分。

Swift 对于新接触的程序员也相当友好。它是第一个工业品质系统的编程语言，Swift 编写的程序可以及时看到效果，并且是一门很有趣的脚本语言。Swift 无需写完代码后再编译运行。

Swift 结合了现代编程语言的思维，广泛的结合苹果工程师的智慧。Swift 的编译器正对性能进行了优化，扩展等无需其它的开销。它的设计从“hello, world”扩展到整个操作系统。所有的一切值得开发者和苹果公司对 Swift 的投资。

Swift 是一种优雅的方式来编写 iOS 和 OS X 应用程序，会持续加入新的功能和特性。我们对 Swift 充满信心。我们迫不及待的想看到您用 Swift 编写的有趣 App。

## 开始 Swift 之旅

依据传统，一门新语言的第一个程序是在屏幕上打印出“Hello World”。用 Swift 可以在一行代码中实现：

```
println("Hello, world");
```

如果你曾经写过 C 或则 Objective-C，这行 Swift 语法你看起来或许非常熟悉，这行代码已经是一个完整的程序。你不需要导入一个单独的库来处理类似输入/输出、字符串处理问题。全局范围内写的代码被作为整个程序的入口，因此你不需要一个 main 函数。同样不需要在语句结尾写分号 (;)。

本节文章展示 Swift 如何完成各种编程任务，给你足够的信息来学习 Swift。如果在学习本节时候你有不明白的地方不用担心，整个教程的其余部分会有一个详细的说明。

注意：

为获得最佳体验，请下载并在 Xcode 中打开源码。可以编辑的代码，并立即看到结果。

## 目录

[隐藏](#)

- [1 简单的赋值](#)

### [2 流程控制](#)

#### [2.1 if](#)

#### [2.2 switch](#)

#### [2.3 for-in](#)

#### [2.4 while](#)

- [3 函数与闭包](#)
- [4 类和对象](#)
- [5 枚举与结构](#)
- [6 接口和扩展](#)

### [7 泛型](#)

# 简单的赋值

使用 `let` 来定义常量，`var` 定义变量。常量的值无需在编译时指定，但是至少要赋值一次。这意味着你可以使用常量来命名一个值，你发现只需一次定义，多个地方使用。

```
varmyVariable = 42
myVariable = 50
letmyConstant = 42
```

一个常量或变量必须与赋值时拥有相同的类型。因此你不用严格定义类型。提供一个值就可以创建常量或变量，并让编译器推断其类型。在上面例子中，编译其会推断 `myVariable` 是一个整数类型，因为其初始化值就是个整数。

如果初始化值没有提供足够的信息(或没有初始化值)，可以在变量名后写类型，以冒号分隔。

```
letimplicitInteger = 70
letimplicitDouble = 70.0
letexplicitDouble: Double = 70
```

练习：

创建一个常量，类型为 `Float`，值为4

值永远不会隐式转换为另一种类型。如果你需要把一个值转换到不同类型，需要明确的构造一个所需类型的实例。

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

练习：

尝试删除最后一行的 `String` 转换，你会得到什么错误？

还有更简单的方法实现在字符串中包含值：以小括号来写值，并用反斜线(\)放在小括号之前。例如：

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

练习：

字符串中使用 \() 来包含一个浮点数，并包含某人的名字

创建一个数组和字典使用方括号 "[]"，访问其元素则是通过方括号中的索引或键。

```
varshoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
```

```
varoccupations = [
  "Malcolm": "Captain",
  "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

要创建一个空的数组或字典，使用初始化语法：

```
letemptyArray = String[]()
letemptyDictionary = Dictionary<String, Float>()
```

如果类型信息无法推断，你可以写空的数组"[]" 和空的字典"[:]”，例如你设置一个知道变量并传入参数到函数：

```
shoppingList = [] // Went shopping and bought everything
```

## 流程控制

使用 if 和 switch 作为条件控制。使用 for-in、for、while、do-while 作为循环。小括号不是必须的，但主体的大括号是必需的。

```
letindividualScores = [75, 43, 103, 87, 12]
varteamScore = 0
forscore in individualScores {
  if score > 50 {
    teamScore += 3
  } else {
    teamScore += 1
  }
}
teamScore
```

### if

在 if 语句中，条件必须是布尔表达式，这意味着 if score { ... } 是错误的，不能隐含的与0比较。

你可以一起使用 if 和 let 来防止值的丢失。这些值是可选的。可选值可以包含一个值或包含一个 nil 来指定值还不存在。写一个问号 "?" 在类型后表示值是可选的。

```
varoptionalString: String? = "Hello"  
optionalString == nil
```

```
varoptionalName: String? = "John Appleseed"  
vargreeting = "Hello!"  
ifletname = optionalName {  
    greeting = "Hello, \(name)"  
}
```

练习:

改变 optionalName 为 nil 会发生什么? 添加一个 else 子句, 如果 optionalName 为 nil 时设置一个不同的值

如果可选值为 nil, 条件判断为 false, 大括号中的代码会被跳过。否则可选值未赋值, 并赋值给了一个常量, 这样为赋值变量会到代码块中执行。

## switch

switch 支持多种数据以及多种比较, 不限制必须是整数

```
letvegetable = "red pepper"  
switchvegetable {  
case"celery":  
    let vegetableComment = "Add some raisins and make ants on a log."  
case"cucumber", "watercress":  
    let vegetableComment = "That would make a good tea sandwich."  
caseletx where x.hasSuffix("pepper"):  
    let vegetableComment = "Is it a spicy \(x)?"  
default:  
    let vegetableComment = "Everything tastes good in soup."  
}
```

练习:

尝试去掉 default, 看看得到什么错误

在执行匹配的情况后, 程序会从 switch 跳出, 而不是继续执行下一个情况。所以不再需要 break 跳出 switch。

## for-in

可使用 for-in 来迭代字典中的每个元素, 提供一对名字来使用每个键值对。

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

```
}  
}  
largest
```

练习:

添加另一个变量来跟踪哪个种类中的数字最大，也就是最大的数字所在的

## while

使用 `while` 来重复执行代码块直到条件改变。循环的条件可以放在末尾来确保循环至少执行一次。

```
var n = 2  
while n < 100 {  
    n = n * 2  
}  
n
```

```
var m = 2  
do {  
    m = m * 2  
} while m < 100  
m
```

你可以在循环中保持一个索引，通过 `..` 来表示索引范围或明确声明一个初始值、条件、增量。这两个循环做相同的事情:

```
var firstForLoop = 0  
for i in 0..3 {  
    firstForLoop += i  
}  
firstForLoop
```

```
var secondForLoop = 0  
for var i = 0; i < 3; ++i {  
    secondForLoop += 1  
}  
secondForLoop
```

使用 `..` 构造范围忽略最高值，而用 `...` 构造的范围则包含两个值。

## 函数与闭包

使用 `func` 声明一个函数。调用函数使用他的名字加上小括号中的参数列表。使用 `->` 分隔参数的名字和返回值类型。

```
func greet(name: String, day: String) -> String {  
    return "Hello \(name), today is \(day)."  
}  
greet("Bob", "Tuesday")
```

练习:

去掉 day 参数，添加一个参数包含今天的午餐选择

使用元组(tuple)来返回多个值。

```
func getGasPrices() -> (Double, Double, Double) {  
    return (3.59, 3.69, 3.79)  
}  
getGasPrices()
```

函数可以接受可变参数个数，集合到一个数组中。

```
func sumOf(numbers: Int...) -> Int {  
    var sum = 0  
    for number in numbers {  
        sum += number  
    }  
    return sum  
}  
sumOf()  
sumOf(42, 597, 12)
```

练习：

编写一个函数计算其参数的平均值

函数可以嵌套。内嵌函数可以访问其定义所在函数的变量。你可以使用内嵌函数来组织代码，避免函数过长和过于复杂。

```
func returnFifteen() -> Int {  
    var y = 10  
    func add() {  
        y += 5  
    }  
    add()  
    return y  
}  
returnFifteen()
```

函数是第一类型的。这意味着函数可以返回另一个函数

```
func makeIncrementer() -> (Int -> Int) {  
    func addOne(number: Int) -> Int {  
        return 1 + number  
    }  
    return addOne  
}  
var increment = makeIncrementer()  
increment(7)
```

一个函数可以接受其他函数作为参数

```
func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
  for item in list {
    if condition(item) {
      return true
    }
  }
  return false
}

func lessThanTen(number: Int) -> Bool {
  return number < 10
}

var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

函数实际是闭包的特殊情况。你可以写一个闭包而无需名字，只需要放在大括号中即可。使用 `in` 把参数、返回值和函数体分开。

```
numbers.map({
  (number: Int) -> Int in
  let result = 3 * number
  return result
})
```

练习：

重写这个闭包，所有奇数返回0。

写闭包时有多种选项。当一个闭包的类型是已知时，例如代表回调，你可以忽略其参数和返回值，或两者。单一语句的闭包可以直接返回值。

```
numbers.map({ number in 3 * number })
```

你可以通过数字而不是名字来引用一个参数，这对于很短的闭包很有用。一个闭包传递其最后一个参数到函数作为返回值。

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

## 类和对象

使用 `class` 可以创建一个类。一个属性的声明则是在类里作为常量或变量声明的，除了是在类的上下文中。方法和函数也是这么写的。

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

练习：通过“`let`”添加一个常量属性，以及添加另一个能接受参数的方法

通过在类名后加小括号来创建类的实例。使用点语法来访问实例的属性和方法。

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

这里写的 Shape 类缺少一些重要的东西：一个构造器，用来在创建实例时设置类。可以使用 init 来创建一个构造器。

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    fun simpleDescription() -> String {
        return "A shape with \($numberOfSides) sides."
    }
}
```

注意 self 用来区分 name 属性和 name 参数。构造器的生命跟函数一样，除了会创建类的实例。每个属性都需要赋值，无论在声明里还是在构造器里。

使用 deinit 来创建一个析构器，来执行对象销毁时的清理工作。

子类名后加父类的名字，以冒号分隔。在继承根类（类似 java Object 类）时无需声明，所以你可以忽略父类。

子类的方法可以通过标记 override 重载父类中的实现，而没有 override 的会被编译器看作是错误。编译器也会检查那些没有被重载的方法。

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    fun area() -> Double {
        return sideLength * sideLength
    }

    override fun simpleDescription() -> String {
        return "A square with sides of length \($sideLength)."
    }
}

let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

练习:

编写另一个 NamedShape 的子类叫做 Circle , 接受半径和名字到其构造器。实现 area 和 describe 方法。

属性可以有 getter 和 setter 方法

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \(sideLength)."
    }
}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

在 perimeter 的 setter 中, 新的值的名字就是 newValue 。你可以提供一个在 set 之后提供一个不冲突的名字。  
注意 EquilateralTriangle 的构造器有3个不同的步骤:

- 设置属性的值
- 调用超类的构造器
- 改变超类定义的属性的值, 添加调用附加的方法、getter、setter 也可以在这里

如果你不需要计算属性, 但是仍然使用 willSet 和 didSet 提供在设置值之后执行工作 。例如, 下面的类要保证其三角形的边长等于矩形的变长。

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
}
```

```
}
}
var square: Square {
  willSet {
    triangle.sideLength = newValue.sideLength
  }
}
init(size: Double, name: String) {
  square = Square(sideLength: size, name: name)
  triangle = EquilateralTriangle(sideLength: size, name: name)
}
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
triangleAndSquare.triangle.sideLength
```

类的方法与函数有个重要的区别。函数的参数名仅用于函数，但方法的参数名也可以用于调用方法(除了第一个参数)。缺省时，一个方法有一个同名的参数，调用时就是参数本身。你可以指定第二个名字，在方法内部使用。

```
class Counter {
  var count: Int = 0
  func incrementBy(amount: Int, numberOfTimes times: Int) {
    count += amount * times
  }
}
var counter = Counter()
counter.incrementBy(2, numberOfTimes: 7)
```

当与可选值一起工作时，你可以写“?”到操作符之前类似于方法属性。如果值在“?”之前就已经是 nil，所有在“?”之后的都会自动忽略，而整个表达式是 nil。另外，可选值是未包装的，所有“?”之后的都作为未包装的值。在两种情况中，整个表达式的值是可选值。

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

## 枚举与结构

使用 enum 来创建枚举。如同类和其他命名类型，枚举也可以有方法

```
enum Rank: Int {
  case Ace = 1
  case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
  case Jack, Queen, King
  func simpleDescription() -> String {
    switch self {
    case .Ace:
      return "ace"
```

```
    case .Jack:
        return "jack"
    case .Queen:
        return "queen"
    case .King:
        return "king"
    default:
        return String(self.toRaw())
    }
}

let ace = Rank.Ace
let aceRawValue = ace.toRaw()
```

练习:

编写一个函数, 通过比较其原始值, 比较两个 Rank 的值

在如上例子中, 原始值的类型是 Int 所以可以只指定第一个原始值。其后的原始值都是按照顺序赋值的。也可以使用字符串或浮点数作为枚举的原始值。

使用 toRaw 和 fromRaw 函数可以转换原始值和枚举值。

```
if let convertedRank = Rank.fromRaw(3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

枚举的成员值就是实际值, 而不是其他方式写的原始值。实际上, 有些情况是原始值, 就是你不提供的时候。

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
            case .Spades:
                return "spades"
            case .Hearts:
                return "hearts"
            case .Diamonds:
                return "diamonds"
            case .Clubs:
                return "clubs"
        }
    }
}

let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

练习:

添加一个 color 方法到 Suit 并在 spades 和 clubs 时返回 "black", 并且给 hearts 和 diamonds 返回 "red"。

注意上面引用 Hearts 成员的两种方法：当赋值到 hearts 常量时，枚举成员 Suit.Hearts 通过全名引用，因为常量没有明确的类型。在 switch 中，枚举通过 .Hearts 引用，因为 self 的值是已知的。你可以在任何时候使用方便的方法。

使用 struct 创建结构体。结构体支持多个与类相同的行为，包括方法和构造器。一大重要的区别是代码之间的传递总是用拷贝(值传递)，而类则是传递引用。

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}

letthreeOfSpades = Card(rank: .Three, suit: .Spades)
letthreeOfSpadesDescription = threeOfSpades.simpleDescription()
```

练习：

加方法到 Card 类来创建一桌的纸牌，每个纸牌都有合并的 rank 和 suit。

一个枚举的实例成员可以拥有实例的值。相同枚举成员实例可以有不同的值。你在创建实例时赋值。指定值和原始值的区别：枚举的原始值与其实例相同，你在定义枚举时提供原始值。

例如，假设情况需要从服务器获取太阳升起和降落时间。服务器可以响应相同的信息或一些错误信息。

```
enumServerResponse {
    case Result(String, String)
    case Error(String)
}

letsuccess = ServerResponse.Result("6:00 am", "8:09 pm")
letfailure = ServerResponse.Error("Out of cheese.")

switchsuccess {
caselet.Result(sunrise, sunset):
    let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."
caselet.Error(error):
    let serverResponse = "Failure... \(error)"
}
```

练习：

给 ServerResponse 添加第三种情况来选择

注意日出和日落时间实际上来自于对 ServerResponse 的部分匹配来选择的

# 接口和扩展

使用 protocol 来声明一个接口。

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

类、枚举和结构体都可以实现接口。

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}

vara = SimpleClass()
a.adjust()
letaDescription = a.simpleDescription
```

```
struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}

varb = SimpleStructure()
b.adjust()
letbDescription = b.simpleDescription
```

练习：

写一个实现这个接口的枚举

注意声明 SimpleStructure 时候 mutating 关键字用来标记一个会修改结构体的方法。SimpleClass 的声明不需要标记任何方法因为类中的方法经常会修改类。

使用 extension 来为现有的类型添加功能，比如添加一个计算属性的方法。你可以使用扩展来给任意类型添加协议，甚至是你从外部库或者框架中导入的类型。

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
```

```
}  
}  
7. simpleDescription
```

练习:

给 Double 类型写一个扩展，添加 absoluteValue 功能

你可以像使用其他命名类型一样使用接口名——例如，创建一个有不同类型但是都实现一个接口的对象集合。当你处理类型是接口的值时，接口外定义的方法不可用。

```
let protocolValue: ExampleProtocol = a  
protocolValue.simpleDescription  
// protocolValue.anotherProperty // Uncomment to see the error
```

即使 protocolValue 变量运用时的类型是 simpleClass，编译器会把它的类型当做 ExampleProtocol。这表示你不能调用类在它实现的接口之外实现的方法或者属性。

## 泛型

在尖括号里写一个名字来创建一个泛型函数或者类型。

```
func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {  
    var result = ItemType[] ()  
    for i in 0..times {  
        result += item  
    }  
    return result  
}  
repeat("knock", 4)
```

你也可以创建泛型类、枚举和结构体。

```
// Reimplement the Swift standard library's optional type  
enum OptionalValue<T> {  
    case None  
    case Some(T)  
}  
var possibleInteger: OptionalValue<Int> = .None  
possibleInteger = .Some(100)
```

在类型名后面使用 where 来指定一个需求列表——例如，要限定实现一个协议的类型，需要限定两个类型要相同，或者限定一个类必须有一个特定的父类。

```
func anyCommonElements <T, U where T: Sequence, U: Sequence, T.GeneratorType.Element: Equatable,  
T.GeneratorType.Element == U.GeneratorType.Element> (lhs: T, rhs: U) -> Bool {  
    for lhsItem in lhs {  
        for rhsItem in rhs {  
            if lhsItem == rhsItem {  
                return true  
            }  
        }  
    }  
    return false  
}
```

```
    }  
    }  
    }  
    return false  
}  
anyCommonElements([1, 2, 3], [3])
```

练习:

修改 `anyCommonElements` 函数来创建一个函数，返回一个数组，内容是两个序列的共有元素。

简单起见，你可以忽略 `where`，只在冒号后面写接口或者类名。`<T: Equatable>`和`<T where T: Equatable>`是等价的。

## The Basics

---

**导航搜索**Swift 是 iOS 和 OS X 应用开发的一门新语言。然而，如果你有 C 或者 Objective-C 开发经验的话，你会发现 Swift 的很多内容都是你熟悉的。

Swift 的类型是在 C 和 Objective-C 的基础上提出的，`Int` 是整型；`Double` 和 `Float` 是浮点型；`Bool` 是布尔型；`String` 是字符串。Swift 还有两个有用的集合类型，`Array` 和 `Dictionary`，详情参见集合类型(待添加链接)。

就像 C 语言一样，Swift 使用变量来进行存储并通过变量名来关联值。在 Swift 中，值不可变的变量有着广泛的应用，它们就是常量，而且比 C 语言的常量更强大。在 Swift 中，如果你要处理的值不需要改变，那使用常量可以让你的代码更加安全并且更好地表达你的意图。

除了我们熟悉的类型，Swift 还增加了 Objective-C 中没有的类型比如元组 (Tuple)。元组可以让你创建或者传递一组数据，比如作为函数的返回值时，你可以用一个元组可以返回多个值。

Swift 还增加了可选 (Optional) 类型，用于处理值缺失的情况。可选表示“那儿有一个值，并且它等于 `x`”或者“那儿没有值”。可选有点像在 Objective-C 中使用 `nil`，但是它可以用在任何类型上，不仅仅是类。可选类型比 Objective-C 中的 `nil` 指针更加安全也更具表现力，它是 Swift 许多强大特性的重要组成部分。

Swift 是一个类型安全的语言，可选就是一个很好的例子。Swift 可以让你清楚地知道值的类型。如果你的代码期望得到一个 `String`，类型安全会阻止你不小心传入一个 `Int`。你可以在开发阶段尽早发现并修正错误。

## 目录

[隐藏](#)

### [1 常量和变量](#)

#### [1.1 声明常量和变量](#)

#### [1.2 类型标注](#)

#### [1.3 常量和变量的命名](#)

#### [1.4 输出常量和变量](#)

- [2 注释](#)

- [3 分号](#)

### [4 整数](#)

#### [4.1 整数的边界](#)

#### [4.2 Int](#)

#### [4.3 UInt](#)

- [5 浮点数](#)

- [6 类型安全及类型推断](#)

- [7 数字字面量 \(Literals\)](#)

### [8 数字类型转换](#)

#### [8.1 整数转换](#)

#### [8.2 整数与浮点数转换](#)

- [9 类型别名 \(aliases\)](#)

- [10 布尔值](#)

- [11 元组](#)

### [12 可选量](#)

#### [12.1 if 语句与强制拆包](#)

#### [12.2 可选值绑定](#)

#### [12.3 nil](#)

#### [12.4 可选量的隐式拆包](#)

### [13 断言 \(Assertions\)](#)

[13.1 借助断言辅助调试](#)

[13.2 时应使用断言](#)

## 常量和变量

常量和变量把一个名字(比如 `maximumNumberOfLoginAttempts` 或者 `welcomeMessage`)和一个指定类型的值(比如数字`10`或者字符串 `Hello`) 关联起来。常量的值一旦设定就不能改变, 而变量的值可以随意更改。

### 声明常量和变量

常量和变量必须在使用前声明, 用 `let` 来声明常量, 用 `var` 来声明变量。下面的例子展示了如何用常量和变量来记录用户尝试登录的次数:

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

这两行代码可以被理解为 :

“声明一个名字是 `maximumNumberOfLoginAttempts` 的新常量, 并给它一个值`10`。然后, 声明一个名字是 `currentLoginAttempt` 的变量并将它的值初始化为`0`。”

在这个例子中, 允许的最大尝试登录次数被声明为一个常量, 因为这个值不会改变。当前尝试登录次数被声明为一个变量, 因为每次尝试登录失败的时候都需要增加这个值。

你可以在一行中声明多个常量或者多个变量, 用逗号隔开:

```
var x = 0.0, y = 0.0, z = 0.0
```

注意:

如果你的代码中有不需要改变的值, 请将它声明为常量。只将需要改变的值声明为变量。

### 类型标注

当你声明常量或者变量的时候可以加上类型标注, 说明常量或者变量中要存储的值的类型。如果要添加类型标注, 在常量或者变量名后面加上一个冒号和空格, 然后加上类型名称。

这个例子给 `welcomeMessage` 变量添加了类型标注, 表示这个变量可以存储 `String` 类型的值:

```
var welcomeMessage: String
```

声明中的冒号代表着“是...类型”, 所以这行代码可以被理解为:

“声明一个类型为 `String`, 名字为 `welcomeMessage` 的变量。”

“类型为 `String`”的意思是“可以存储任意 `String` 类型的值。”

welcomeMessage 变量现在可以被设置成任意字符串：

```
welcomeMessage = "Hello"
```

注意：

一般来说你很少需要写类型标注。如果你在声明常量或者变量的时候赋了一个初始值，**Swift** 可以推断出这个常量或者变量的类型，详情参见类型安全和类型推断(待添加链接)。在上面的例子中，没有给 **welcomeMessage** 赋初始值，所以添加了一个类型标注。

## 常量和变量的命名

你可以用任何你喜欢的字符作为常量和变量名，包括 **Unicode** 字符：

```
let π = 3.14159
let 你好 = "你好世界"
let      = "dogcow"
```

常量与变量名不能包含数学符号，箭头，保留的(或者非法的)**Unicode** 码位，连线与制表符。尽管常量与变量名中可以包含数字，但是它们不能以数字打头。

一旦你将常量或者变量声明为确定的类型，你就不能使用相同的名字再次进行声明，或者以改变其存储的值为其他类型。同时，你也不能将常量与变量进行互转。

注意：如果你需要使用与 **Swift** 保留关键字相同的名称作为常量或者变量名，你可以使用反引号(`)将关键字围住的方式将其作为名字使用。无论如何，你应当避免使用关键字作为常量或变量名，除非你别无选择。

你可以更改现有的变量值为其他同类型的值，在下面的例子中，**friendlyWelcome** 的值从**"Hello!"**改为了**"Bonjour!"**：

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

和变量不一样，常量的值一旦被确定以后就不能更改了。尝试这样做会在编译时报错：

```
let languageName = "Swift"
languageName = "Swift++"
// this is a compile-time error - languageName cannot be changed
```

## 输出常量和变量

你可以用 **println** 函数来输出当前常量或变量的值：

```
println(friendlyWelcome)
// prints "Bonjour!"
```

`println` 是一个用来输出的全局函数，输出的内容会在最后带换行。如果你用 `Xcode`，`println` 将会输出内容到“console”面板上。(另一种函数叫 `print`，唯一区别是在输出内容最后不会加入换行。)

`println` 函数输出传入的 `String` 值：

```
println("This is a string")
// prints "This is a string"
```

像 `Cocoa` 里的 `NSLog` 函数一样，`println` 函数可以输出更复杂的信息。这些信息可以包含当前常量和变量的值。

`Swift` 用字符串插值 (`string interpolation`) 的方式把常量名或者变量名当做占位符加入到长字符串中，`Swift` 会用当前常量或变量的值替换这些占位符。将常量或变量名放入反斜杠符加一对圆括号中 `"\()"`：

```
println("The current value of friendlyWelcome is \(friendlyWelcome)")
// prints "The current value of friendlyWelcome is Bonjour!"
```

注意：

字符串插值所有可用的选项在 [字符串插值](#) 这章中讲述。

## 注释

请将你的代码中的非执行文本注释成提示或者笔记以方便你将来阅读。`Swift` 的编译器将会在编译代码时自动忽略掉注释部分。

`Swift` 中的注释与 `C` 语言的注释非常相似。单行注释以双正斜杠作 `(//)` 为起始标记：

```
// this is a comment
```

你也可以进行多行注释，其起始标记为单个正斜杠后跟随一个星号 `(/*)`，终止标记为一个星号后跟随单个正斜杠 `(*/)`：

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

与 `C` 语言多行注释不同的是，`Swift` 的多行注释可以嵌套在其它的多行注释之中。你可以先生成一个多行注释块，然后在这个注释块之中再嵌套成第二个多行注释。终止注释时先插入第二个注释块的终止标记，然后再插入第一个注释块的终止标记：

```
/* this is the start of the first multiline comment
/* this is the second, nested multiline comment */
this is the end of the first multiline comment */
```

通过运用嵌套多行注释，你可以快速方便的注释掉一大段代码，即使这段代码之中已经含有了多行注释块。

## 分号

与其他大部分编程语言不同，**Swift** 并不强制要求你在每条语句的结尾处使用分号(;), 当然，你也可以按照你自己的习惯添加分号。有一种情况下必须要用分号，即你打算在同一行内写多条独立的语句：

```
let cat = " "; println(cat)
```

```
// prints " "
```

## 整数

整数（integer）指没有小数部分的整数，如 **42** 或 **-23**。整数既可以是带符号的（**signed**，正数、零、负数）也可以是无符号的（**unsigned**，正数或零）。

**Swift** 提供 **8**、**16**、**32**、**64** 位形式的带符号及无符号整数。这些整数类型遵循 **C** 语言的命名规约，如 **8** 位无符号整数的类型为 **UInt8**，**32** 位带符号整数的类型为 **Int32**。与 **Swift** 中的所有类型一样，这些整数类型的名称以大写字母开头。

### 整数的边界

各整数类型允许的最小值及最大值可通过 **min** 及 **max** 属性获得：

```
let minValue = UInt8.min // minValue 等于 0, 类型为 UInt8
```

```
let maxValue = UInt8.max //maxValue 等于 255, 类型为 UInt8
```

这些属性的值的类型与对应宽度的数据类型一致（如上例为 **UInt8**），因此也可以在表达式中与同类型的其他数值一起使用。

## Int

绝大多数情况下，你并不需要自己决定代码中要使用的整数宽度。**Swift** 还提供了一个整数类型 **Int**，其宽度与当前平台的原生字长（**word size**）一致：

- 在 32 位平台，`Int` 与 `Int32` 宽度一致。
- 在 64 位平台，`Int` 与 `Int64` 宽度一致。

除非你需要处理特定宽度的整数，在代码中应该只使用 `Int` 表示整数。这样可以保证一致性及互运算性。即使是在 32 位平台，`Int` 也能存储 `-2,147,483,648` 到 `2,147,483,647` 的任意数值，对于很多整数区间需求来说已经足够大了。  
译注：信苹果会丢饭碗的

## UInt

`Swift` 还提供了无符号整数类型 `UInt`，其宽度与当前平台的原生字长一致：

- 在 32 位平台，`UInt` 与 `UInt32` 宽度一致。
- 在 64 位平台，`UInt` 与 `UInt64` 宽度一致。

注：只有在特别需要宽度与平台原生字长一致的时才需要使用无整数类型 `UInt`。否则应使用 `Int`，即使要存储的值一定非负。总使用 `Int` 表示整数值有助于保证代码互运算性、避免不同数据类型的转换，并且与整数类型推断相匹配，参见 [类型安全及类型推断](#)。

## 浮点数

浮点数表示有小数部分的数字，例如 `3.14159`、`0.1` 及 `-273.15`。

浮点数类型可以表示的值比整数类型宽广得多，也能存储 `Int` 类型能存放的最大及最小值。`Swift` 提供两种有符号的浮点数类型：

- `Double` 表示一个 64 位浮点数。在浮点数值非常大或非常精确时使用它。
- `Float` 表示一个 32 位浮点数。在浮点数值不需要 64 位精度时使用它。

注意：`Double` 的精度为 15 个十进制有效数字，而 `Float` 的精度只有 6 位十进制有效数字。应根据代码所需数值的特点及值域选用合适的浮点数类型。

## 类型安全及类型推断

`Swift` 是一门类型安全的语言。类型安全要求代码中值的类型非常明确。如果代码中要求提供 `String` 数据，则不会错

误地向它传递 `Int` 数据。

由于 `Swift` 类型安全，它会在编译代码时执行类型检查，并将任何类型不匹配的地方标为错误。这样可以在开发过程中尽可能早地发现并修复问题。

类型检查有助于在操作不同类型值时避免错误。然而，这并不意味着你必须为声明的每个常量与变量指定类型。如果你不指定所需值的类型，`Swift` 会通过类型推断（`typeinference`）求得适当的类型。类型推断允许编译器在编译代码时，根据你提供的值，自动推测出特定表达式的类型。

得益于类型推断，`Swift` 对类型声明的需要比起 `C` 或 `Objective-C` 语言而言要少很多。常量与变量仍然有明确的类型，但明确指定类型的工作已经由编译器代你完成。

类型推断在你声明常量或变量的同时提供初始值时尤其有用。通常通过在声明时赋字面值（`literal value`，或称“字面量” `literal`）实现。（字面值指直接出现在源代码中的值，如下例中的 `42` 与 `3.14159`。）

例如，如果将字面值 `42` 赋给新的常量，而不明确讲它是什么类型，`Swift` 会推断出你希望该常量为 `Int` 类型，因为你初始化时提供的数字像是个整数

```
let meaningOfLife = 42
```

```
// meaningOfLife 被推断属于 Int 类型
```

类似地，如果不为浮点数字面值指定类型，`Swift` 会推断出你希望创建一个 `Double` 变量：

```
let pi = 3.14159
```

```
// pi 被推断属于 Double 类型
```

`Swift` 在推断浮点数类型时总会选用 `Double`（而不用 `Float`）。

如果在表达式中同时使用整数与浮点数字面值，将根据上下文推断得到 `Double` 类型：

```
let anotherPi = 3 + 0.14159
```

```
// anotherPi 也被推断为 Double 类型
```

字面值 `3` 没有明确的类型，自身也不属于某个明确的类型，但由于加法中出现了浮点数字面值，因此推断出合适的输出类型为 `Double`。

# 数字字面量 (Literals)

整数字面量可以以下面的形式书写：

- 十进制数，无需前缀
- 二进制数，以 `0b` 为前缀
- 八进制数，以 `0o` 为前缀
- 十六进制数，以 `0x` 为前缀

下述整数字面量的值均为十进制的 `17`：

```
letdecimalInteger = 17 = 17

letbinaryInteger = 0b10001    // 17 的二进制表示

letoctalInteger= 0o21        // 17 的八进制表示

lethexadecimalInteger = 0x11 // 17 的十六进制表示
```

浮点数字面值可以为十进制（无需前缀），也可以是十六进制（以 `0x` 为前缀）。小数点两侧均必须有数字（或十六进制数字）。还可以有一个可选的幂次（**exponent**），对十进制浮点数为大写或小写的 **e**，对十六进制浮点数为大写或小写的 **p**。

对幂次为 **exp** 的十进制数，基数将乘以 $10^{\text{exp}}$ ：

- `1.25e2` means  $1.25 \times 10^2$ , or 125.0.
- `1.25e-2` means  $1.25 \times 10^{-2}$ , or 0.0125.

对幂次为 **exp** 的十六进制数，基数将乘以 $2^{\text{exp}}$ ：

- `0xFp2` means  $15 \times 2^2$ , or 60.0.
- `0xFp-2` means  $15 \times 2^{-2}$ , or 3.75.

下述所有浮点数字面量的值均为十进制的 `12.1875`：

```
letdecimalDouble = 12.1875
```

```
letexponentDouble = 1.21875e1
```

```
lehexadecimalDouble = 0xC.3p0
```

数字字面量可以包含额外的格式以便于阅读。整数与浮点数均可以添加多余的零或下划线以提高可读性。两种格式均不会影响字面量的实际值：

```
letpaddedDouble = 000123.456
```

```
letoneMillion = 1_000_000
```

```
letjustOverOneMillion = 1_000_000.000_000_1
```

## 数字类型转换

`Int` 类型应作为所有常规用途的整数常量及变量的类型，即使它们确实非负。通常情况下，使用默认的整数类型意味着这些整型常量与变量均可即时互相参与运算，并可与根据整数字面值推断出的类型相匹配。

仅当手中的任务必须使用其他整数类型时才用它们，如外部数据源提供宽度明确的数据，或为了性能、内存占用等其他必需优化考虑。在这些情况下使用宽度明确的类型有助于发现偶然的数值溢出，并还原这些数据实际使用时的特点。

### 整数转换

不同类型的整数常量或变量所能存储的值域不同。`Int8` 常量或变量能存储 `-128` 到 `127`，而 `UInt8` 常量或变量能存储 `0` 到 `255`。无法存放进某常量或变量的数字会报编译时错误：

```
letcannotBeNegative: UInt8 = -1
```

```
// UInt8 不能存储负数，因此会报错
```

```
lettooBig: Int8 = Int8.max + 1
```

```
// Int8 不能存储大于其最大值的数字，
```

```
// 因此这里也会报错
```

由于不同数据类型能存储的值域不同，在进行数据转换时需要具体问题具体对待。这种实际选择的过程可避免隐式转换的问题，并使类型转换的意图在代码中明确地展现出来。

要将一种数据类型转换为另一种，应使用现有值初始化一个所需类型的新数。下例中，常量 `twoThousand` 的类型为 `UInt16`，而常量 `one` 的类型为 `UInt8`。它们无法直接相加，因为类型不同。因此，本例将调用 `UInt16(one)` 新建一个 `UInt16` 数，并以 `one` 的数值初始化，并将新值放在调用处：

```
lettwoThousand: UInt16 = 2_000
```

```
letone: UInt8 = 1
```

```
lettwoThousandAndOne = twoThousand + UInt16(one)
```

现在加号两侧均为 `UInt16` 类型，因此允许相加。输出的常量 (`twoThousandAndOne`) 推断得出的类型为 `UInt16`，因为它是两个 `UInt16` 值的和。

某类型(赋初始值) 是调用 `Swift` 类型构造函数并传递初始值的默认方法。幕后运作情况是，`UInt16` 有一个接受 `UInt8` 值的构造函数，因此该构造函数会被用于根据现有 `UInt8` 创建新的 `UInt16`。不过，在这里并不能传入任意类型——只能传入 `UInt16` 提供有构造函数的类型。扩展现有类型使其提供接受新类型（包括自己定义的类型）的构造函数的方法请见 [扩展](#) 一章。

## 整数与浮点数转换

整数与浮点数类型之间的转换必须显式指定：

```
letthree = 3
```

```
letpointOneFourOneFiveNine = 0.14159
```

```
letpi = Double(three) + pointOneFourOneFiveNine
```

```
// pi 等于 3.14159，推断得到的类型为 Double
```

这段代码中，常量 `three` 被用来创建新的 `Double` 类型值，这样加法两侧的类型才一致。不进行类型转换的话，两侧将不允许相加。

浮点数到整数的逆向转换同样可行，整数类型可以用 `Double` 或 `Float` 值初始化：

```
letintegerPi = Int(pi)
```

```
// integerPi 等于 3，推断得到的类型为 Int
```

这样用浮点数初始化新整数时，浮点数值总会被截断。即，`4.75` 变为 `4`，`-3.9` 变为 `-3`。

提示：多个数字常量或变量的结合规则与数字字面量的结合规则不同。字面量 `3` 可以直接与字面值 `0.14159` 相加，因

为数字字面量没有明确指定类型，它们自身也没有明确的类型。其类型仅当被编译器求值时才推断得出。

## 类型别名（aliases）

类型别名（**type aliases**）为现有类型定义可替代的名称。类型别名通过 **typealias** 关键字定义。

类型别名在需要以上下文中更为合适的名称称呼某个现有类型时非常有用，例如当处理来自外部数据源的特定宽度的数据时：

```
typealias AudioSample = UInt16
```

类型别名定义完成后，即可在可能用到原名的地方使用别名：

```
varmaxAmplitudeFound = AudioSample.min
```

```
// 已发现的最大振幅 现在为 0
```

此处音频采样 作为 **UInt16** 的别名定义。因为它是别名，因此对音频采样.min 的调用实际上会调用 **UInt16.min**，最终为 已发现的最大振幅 变量提供初始值 **0**。

## 布尔值

**Swift** 实现了基本的布尔（**boolean**）类型，称为 **Bool**。布尔值也称为逻辑值（**logical**），因为只能为真（**true**）或假（**false**）。**Swift** 提供了两种布尔常量值：**true** 与 **false**：

```
letorangesAreOrange = true
```

```
letturnipsAreDelicious = false
```

**orangesAreOrange** 与 **turnipsAreDelicious** 很好吃 的类型被推断为 **Bool**，因为它们以布尔字面值初始化。与上文中的 **Int** 及 **Double** 一样，并不需要明确声明为 **Bool**，只要在创建变量的同时用 **true** 或 **false** 初始化。以已知类型的值初始化常量或变量时，类型推断使 **Swift** 的代码更简练、更具可读性。

控制条件语句（如 **if** 语句）时，布尔值尤其有用：

```
ifturnipsAreDelicious {
```

```
    println("Mmm, tastyturnips!")
```

```
}else{  
  
    println("Eww, turnipsare horrible.")  
  
}  
  
// prints "Eww, turnips are horrible."
```

if 等条件语句的详细情况请见“流程控制”

Swift 的类型安全特性可避免非布尔值被当作 **Bool** 使用。下面的例子会报编译时错误：

```
leti = 1  
  
ifi {  
  
    // 本例无法通过编译，报编译错误  
  
}
```

换成下例便可以通过：

```
leti = 1  
  
ifi == 1 {  
  
    // 本例可成功编译  
  
}
```

`i == 1` 比较的结果类型为 **Bool**，因此第二个例子可以通过类型检查。`i == 1` 这类的比较在“基本运算符”一章讨论。

与 Swift 中的其他类型检查规则一样，这些规则可避免偶然错误，并确保各段代码的目的总是明确的。

## 元组

元组将多个值组合为单个值。元组内的值可以是任意类型，各元素不必是相同的类型。

在本例中，`(404, "Not Found")` 是描述一条 HTTP 状态码（HTTPstatus code）的元组。HTTP 状态码 是请求任何网页时，web 服务器返回的特殊值。如果请求了不存在的网页，则会返回状态码 `404 Not Found`。

```
lethttp404Error = (404, "Not Found")  
  
// http404Error is of type (Int, String), and equals (404, "NotFound")
```

(404,"Not Found") 元组将一个 `Int` 值与一个 `String` 值组合起来，表示 HTTP 状态码的两个值：一个数字和一个供人类阅读的描述。它可以这样描述：“类型为(`Int`, `String`) 的元组”。

你可以将类型任意排列来创建元组，它们可以包含任意多种不同的类型。只要你愿意，创建类型为 (`Int`, `Int`, `Int`) 或 (`String`, `Bool`) 的元组也不会有问题，只要这种排列对你有意义。

元组的内容可以还原（`decompose`）为独立的常量或变量，然后便可照常访问：

```
let(statusCode, statusMessage) = http404Error

println("The status code is \(statusCode)")

// prints "The status code is 404"

println("The status message is \(statusMessage)")

// prints "The status message is Not Found"
```

如果你只需要元组的一部分值，可以在还原元组时用下划线（`_`）忽略掉其他部分：

```
let(justTheStatusCode, _) = http404Error

println("The status code is \(justTheStatusCode)")

// prints "The status code is 404"
```

还可以通过以 `0` 开头的索引号访问元组的各个元素值：

```
println("The status code is \(http404Error.0)")

// prints "The status code is 404"

println("The status message is \(http404Error.1)")

// prints "The status message is Not Found"
```

还可以在定义元组时为各元素命名：

```
lethttp200Status = (statusCode: 200, description: "OK")
```

为元组各元素命名后，便可以通过元素名称访问各元素的值了：

```
println("The status code is \(\(http200Status.statusCode)")  
  
// prints "The status code is 200"  
  
println("The status message is\(\(http200Status.description)")  
  
// prints "The status message is OK"
```

元组在作为函数返回值时尤其有用。一个获取网页内容的函数可能会返回 **(Int, String)** 元组类型，来描述网页装取是成功还是失败。函数返回两个类型完全不同的值描述结果，所能提供的信息比只能返回固定类型的单个值要有用得多。详情请参见返回多个返回值的函数。

提示：元组对临时组合相关的多个值非常有用。它们并不适合用来创建复杂的数据结构。如果你的数据结构的生命期超过临时使用的范畴，请将它作为类或结构建模，而不是以元组存储。详情请见类与结构。

## 可选量

在值可能不存在时使用可选量（**optional**）。可选量是指：

- 存在一个值，这个值等于 **x**

或者

- 不存在任何值

注：可选量的概念在 **C** 和 **Objective-C** 中并不存在。**Objective-C** 中最相近的是，一个以对象为返回值的方法，可以返回 **nil**，表示“不存在有效的对象”。不过，该规则只对对象有效——对于结构体、基本的 **C** 类型或枚举值均不起作用。对于这些类型，**Objective-C** 语言的方法通常会返回一个特殊值（如 **NSNotFound**）来表示值不存在。这种策略假定该方法的调用方知道要测试返回值是否等于某个特殊值，并且记得要作此检查。**Swift** 的可选量允许表示任何类型不存在值，无需定义特殊常量。

举例说明。**Swift** 的 **String** 类型有一个名为 **toInt** 的方法，可尝试将 **String** 值转为 **Int** 值。然而，不是所有字符串都可以转换为整数。字符串 **"123"** 可以转换为数值 **123**，而字符串 **"hello, world"** 却显然没有对应的数值。

下面的例子会利用 **toInt** 方法，尝试将 **String** 转换为 **Int**：

```
let possibleNumber = "123"
```

```
letconvertedNumber = possibleNumber.toInt()
```

```
// 转换得到的数字 被推断为 "Int?" 类型，即"可选的 Int"
```

由于 `toInt` 方法可能转换失败，因此它会返回一个可选的 `Int` 型，而不是 `Int` 型。可选的 `Int` 记作 `Int?`，而不是 `Int`。

其中的问号表示该类型包含的值是可选的，即 `Int?` 可能包含某个 `Int` 类型的值，也可能不含任何值。（但不能包含其他类型的值，如 `Bool` 值或 `String` 值。不是 `Int` 就是不存在。）

## if 语句与强制拆包

可以使用 `if` 语句测试可选量是否包含值。如果存在，则求值结果为 `true`；否则为 `false`。

一旦确认可选量的确包含值，便可以通过在变量名末尾添加感叹号 (!) 访问其内部的值。感叹号明确表达：“我知道这个可选量的确存在值；请使用那个值。”这种操作称为对可选值进行强制拆包 (`force-unwrap`):

```
ifconvertedNumber {  
  
    println("\((possibleNumber)has an integer value of \((convertedNumber!))"  
  
}else{  
  
    println("\((possibleNumber) could not be converted to aninteger")  
  
}  
  
//输出 "123 的整数值为 123"
```

注:

尝试用 `!` 访问不存在的可选值时会导致运行时错误。在用 `!` 强制拆包之前，务必确认可选量的确包含非 `nil` 的值。

## 可选值绑定

可以通过可选值绑定 (`optional binding`) 测试可选量是否包含一个值，如果存在，则将该值以临时常量或变量的形式拆包使用。可选值绑定可以与 `if` 或 `while` 语句结合使用，这样只需要一步就可以检查是否存在值、提取该值、并存放于常量或变量中。关于 `if` 与 `while` 语句的更多情况在 [流程控制](#) 一章中讲解。

以 `if` 语句为例，可选值绑定可以这样书写:

```
ifletconstantName = someOptional {  
  
    statements  
  
}
```

上文中可能是数字 一例，可以改写用可选值绑定代替强制拆包：

```
ifletactualNumber = possibleNumber.toInt() {  
  
    println("\(possibleNumber) has an integer value of\\(actualNumber)")  
  
}else{  
  
    println("\(possibleNumber) could not be converted to aninteger")  
  
}  
  
// 输出 "123 的整数值为 123"
```

可以这样理解：

“如果 可能是数字.toInt 返回的 可选的 Int 包含一个值，则新建一个名为 实际值 的常量，并将其值设为可选量中包含的值。”

如果转换成功，常量 实际值 将可供 if 语句的第一段分支使用。该常量已经以可选量内部的值初始化，因此不再需要用后缀 ! 访问其值。本例中，实际值 被直接用来输出转换结果。

常量与变量均可用于可选值绑定。如果需要在第一个分支中修改实际值 的值，可以改写为 if var 实际值，这样可选量的值将作为变量而非常量拆包。

## nil

要将可选变量设为值不存在的状态，可以给它赋特殊值 nil：

```
varserverResponseCode: Int? = 404  
  
//服务器响应码 包含一个实际存在的 Int 值：404  
  
serverResponseCode = nil  
  
//服务器响应码 现在不含任何值
```

注：`nil` 不能用于非可选量。如果代码中的常量或变量需要适配值不存在的特殊情况，务必将它声明为恰当的可选类型。

如果定义的可选量时不提供默认值，该常量或变量将自动设为 `nil`：

```
varsurveyAnswer: String?
```

```
// surveyAnswer 被自动设为 nil
```

注：`Swift` 的 `nil` 与 `Objective-C` 的 `nil` 不同。`Objective-C` 的 `nil` 是指向不存在对象的指针。而 `Swift` 的 `nil` 不是指针——它代表特定类型的值不存在。任何类型的可选量都能赋值为 `nil`，而不仅限于对象类型。

## 可选量的隐式拆包

如上所述，可选量指允许“值不存在”的常量或变量。可选量可以通过 `if` 语句测试是否存在值，也可以通过可选值绑定按条件拆包，并在值存在的情况下才访问可选量的值。

有时根据程序结构可以推断，可选量在首次赋值后，必然存在值。这些情况下，可以不必每次访问时都检测并提取可选量的值，因为可以安全地认为那时一定存在值。

这些可选量可定义为隐式拆包的可选量（`implicitly unwrapped optional`）。隐式拆包的可选量的声明格式为，在希望标为可选的类型名称后面，用感叹号（`String!`）代替问号（`String?`）。

隐式拆包的可选量在可选量首次定义后即确认存在值，在此之后任何时刻都肯定存在的时候有用。`Swift` 中主要应用在类初始化，详见 外部引用与隐式拆包的可选属性。

隐式拆包的可选量在实现级别就是普通的可选量，但能够像非可选量那样使用，无需在每次访问时显式拆包。下例显示了可选的 `String` 与 隐式拆包的可选 `String` 之间的行为差异：

```
letpossibleString: String? = "An optional string."
```

```
println(possibleString!) // // 访问其值时需要添加感叹号
```

```
//输出 "可选的 String。"
```

```
letassumedString: String! = "An implicitly unwrapped optionalstring."
```

```
println(assumedString) //访问其值时无需感叹号
```

```
//输出 "隐式拆包的可选 String。"
```

可以认为，隐式拆包的可选量即授予可选量在被使用时自动拆包的权限。不必每次使用可选量时都在名称后面添加感叹号，只需在定义时在类型后面加上感叹号即可。

注：如果在隐式拆包的可选量存在值之前就尝试访问，会触发运行时错误。结果与在普通可选量尚未赋值时直接加感叹号引用相同。

隐式拆包的可选量也可以当作普通可选量对待，检查是否存在值：

```
ifassumedString {  
  
    println(assumedString)  
  
}  
  
// 输出 "隐式拆包的可选 String。"
```

隐式拆包的可选量同样可以结合可选值绑定使用，单条语句完成检查值并拆包的工作：

```
ifletdefiniteString = assumedString {  
  
    println(definiteString)  
  
}  
  
// 输出 "隐式拆包的可选 String。"
```

注：当变量在后续过程中可能变为 `nil` 时，不应使用隐式拆包的可选量。如果在变量声明周期内都需要检查 `nil` 值，请务必使用普通的可选类型量。

## 断言（Assertions）

可选量允许检查值的存在与否，并允许代码能够适配不存在值的情况。但也有时候，如果值不存在，或不满足特定条件，代码便不可能继续执行下去。对于这些情况，需要在代码中触发断言（`assertion`）译注：“触发”指断言不成立来终止执行，并为找出值不存在或无效的原因创造机会。

### 借助断言辅助调试

断言是一种运行时检查，确认一定为 `true` 的逻辑条件是否成立。即，断言“宣告”某条件一定成立。使用断言，可确保在进一步执行后续代码之前，确保必要条件确实已满足。如果条件的求值结果为 `true`，代码将照常继续运行；如果条件的求值结果为 `false`，则代码不再继续执行，应用程序随之终止。

如果是在调试环境运行时触发断言（如在 **Xcode** 中构建并执行应用程序），你将确切地知道异常情况出现的位置，并能在程序执行到该断言位置的状态下调试程序。断言还允许提供一段合适的调试消息，对该断言加以说明。

断言可通过调用全局函数 **assert** 来实现。将需要求值的表达式（结果为 **true** 或 **false**）传递给 **assert** 函数，如果条件求值结果为 **false**，则应显示错误消息：

```
let age = -3

assert(age >= 0, "A person's age cannot be less than zero")

//断言触发错误，因为 age >= 0 不成立
```

本例中，代码仅在 **age >= 0** 的求值结果为 **true** 时才继续执行，即，**age** 的值为非负数才继续。如果 **age** 的值为负数，即上述代码中的情况，则 **age >= 0** 的求值结果为 **false**，于是断言触发，应用程序终止。

断言的消息内容不能使用字符串内插。如果需要，可省略断言消息，如下例所示：

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

## 时应使用断言

仅当条件可能为假、但必须一定为真代码才能继续执行时才使用断言。适合运用断言检查的场景包括：

- 向自定义下标实现传递了整数下标索引，但该索引号可能太小或太大。
- 值传递给了函数，但如果值无效，函数无法完成其任务。
- 可选值当前为 **nil**，但后续代码要求值非 **nil** 方可成功执行。

参见 [下标](#) 与 [函数](#)。

注：断言可使应用程序终止。断言不适合设计不太可能出现无效条件的场景。尽管如此，在可能出现无效条件的情况下，断言仍不失为在开发过程中确保这些问题在发布以前得到关注与处理的有效途径。

## Strings and Characters

[导航搜索](#) **String** 是一个有序的字符集合，例如 **"hello, world"**, **"albatross"**。**Swift** 字符串通过 **String** 类型来表示，

也可以表示为 **Character** 类型值的集合。

**Swift** 的 **String** 和 **Character** 类型提供了一个快速的，兼容 **Unicode** 的方式来处理代码中的文本信息。创建和操

作字符串的语法与 C 的操作方式相似，轻量并且易读。字符串连接操作只需要简单地通过 + 号将两个字符串相连即可。

与 Swift 中其他值一样，能否更改字符串的值，取决于其被定义为常量还是变量。

尽管语法简易，但 String 类型是一种快速、现代化的字符串实现。每一个字符串都是由独立编码的 Unicode 字符组成，并提供了用于访问这些字符在不同的 Unicode 表示的支持。

String 也可以用于在常量、变量、字面量和表达式中进行字符串插值，这使得创建用于展示、存储和打印的字符串变得轻松自如。

注意：Swift 的 String 类型与 Foundation NSString 类进行了无缝桥接。如果您利用 Cocoa 或 Cocoa Touch 中的 Foundation 框架进行工作，整个 NSString API 都可以调用您创建的任意 String 类型的值，您额外还可以在任意 API 中使用本章介绍的 String 特性。您也可以在任意要求传入 NSString 实例作为参数的 API 中使用 String 类型的值进行替换。

更多关于在 Foundation 和 Cocoa 中使用 String 的信息请查看 [Using Swift with Cocoa and Objective-C](#)。

## 目录

### 隐藏

- [1 字符串字面量](#)
- [2 初始化空字符串](#)
- [3 字符串可变性](#)
- [4 字符串是值类型](#)
- [5 使用字符\(Characters\)](#)
- [6 计算字符数量](#)
- [7 字符串插值](#)

### [8 比较字符串](#)

#### [8.1 字符串相等](#)

#### [8.2 前缀/后缀相等](#)

- [9 大写和小写字符串](#)

### [10 Unicode](#)

#### [10.1 Unicode 术语\(Terminology\)](#)

#### [10.2 字符串的 Unicode 表示](#)

#### [10.3 UTF-8](#)

#### [10.4 UTF-16](#)

#### [10.5 Unicode 标量 \(Scalars\)](#)

# 字符串字面量

您可以在您的代码中包含一段预定义的字符串值作为字符串字面量。字符串字面量是由双引号包裹着的具有固定顺序的文本字符集。

字符串字面量可以用于为常量和变量提供初始值。

```
letsomeString = "Some string literal value"
```

注意: `someString` 变量通过字符串字面量进行初始化, `Swift` 因此推断其为 `String` 类型。

字符串字面量可以包含以下特殊字符:

- 转移特殊字符 `\0` (空字符)、`\\` (反斜线)、`\t` (水平制表符)、`\n` (换行符)、`\r` (回车符)、`\"` (双引号)、`\'` (单引号)。
- 单字节 `Unicode` 标量, 写成 `\xnn`, 其中 `nn` 为两位十六进制数。
- 双字节 `Unicode` 标量, 写成 `\unnnn`, 其中 `nnnn` 为四位十六进制数。
- 四字节 `Unicode` 标量, 写成 `\Unnnnnnnn`, 其

中 `nnnnnnnn` 为八位十六进制数。下面的代码为各种特殊字符的使用示例。`wiseWords` 常量包含了两个转移特殊字符 (双括号); `dollarSign`、`blackHeart` 和 `sparklingHeart` 常量演示了三种不同格式的 Unicode 标量:

```
letwiseWords = "\"Imagination is more important than knowledge\" - Einstein"

// "Imagination is more important than knowledge" - Einstein

letdollarSign = "\x24"           // $, Unicode scalar U+0024

letblackHeart = "\u2665"        // ♥, Unicode scalar U+2665

letsparklingHeart = "\U0001F496" // , Unicode scalar U+1F496
```

## 初始化空字符串

为了构造一个很长的字符串, 可以创建一个空字符串作为初始值。可以将空的字符串字面量赋值给变量, 也可以初始化一个新的 `String` 实例:

```
varempyString = ""                // empty string literal

varanotherEmptyString = String() // initializer syntax

// 这两个字符串都为空, 并且两者等价
```

您可以通过检查其 `Boolean` 类型的 `isEmpty` 属性来判断该字符串是否为空:

```
ifemptyString.isEmpty {

    println("Nothing to see here")

}

// 打印 "Nothing to see here"
```

## 字符串可变性

您可以通过将一个特定字符串分配给一个变量来对其进行修改, 或者分配给一个常量来保证其不会被修改:

```
varvariableString = "Horse"

variableString += " and carriage"
```

```
// variableString 现在为 "Horse and carriage"  
  
let constantString = "Highlander"  
  
constantString += " and another Highlander"  
  
// 这会报告一个编译错误(compile-time error) - 常量不可以被修改。
```

注意：在 Objective-C 和 Cocoa 中，您通过选择两个不同的类( `NSString` 和 `NSMutableString` )来指定该字符串是否可以被修改，Swift 中的字符串是否可以修改仅通过定义的是变量还是常量来决定，实现了多种类型可变性操作的统一。

## 字符串是值类型

Swift 的 `String` 类型是值类型。如果您创建了一个新的字符串，那么当其进行常量、变量赋值操作或在函数/方法中传递时，会进行值拷贝。任何情况下，都会对已有字符串值创建新副本，并对该新副本进行传递或赋值。值类型在 [Structures and Enumerations Are Value Types](#) 中进行了说明。

注意：其 Cocoa 中的 `NSString` 不同，当您在 Cocoa 中创建了一个 `NSString` 实例，并将其传递给一个函数/方法，或者赋值给一个变量，您永远都是传递或赋值同一个 `NSString` 实例的一个引用。除非您特别要求其进行值拷贝，否则字符串不会进行赋值新副本操作。

Swift 默认字符串拷贝的方式保证了在函数/方法中传递的是字符串的值，其明确了无论该值来自于哪里，都是您独自拥有的。您可以放心您传递的字符串本身不会被更改。

在实际编译时，Swift 编译器会优化字符串的使用，使实际的复制只发生在绝对必要的情况下，这意味着您始终可以将字符串作为值类型的同时获得极高的性能。

## 使用字符(Character)

Swift 的 `String` 类型表示特定序列的字符值的集合。每一个字符值代表一个 Unicode 字符。您可利用 `for-in` 循环来遍历字符串中的每一个字符：

```
for character in "Dog! " {  
  
    println(character)
```

```
}  
  
// D  
  
// o  
  
// g  
  
//!  
  
//
```

for-in 循环在 For Loops 中进行了详细描述。

另外，通过标明一个 `Character` 类型注解并通过字符字面量进行赋值，可以建立一个独立的字符常量或变量：

```
letyenSign: Character = "¥"
```

## 计算字符数量

通过调用全局 `countElements` 函数并将字符串作为参数进行传递可以获取该字符串的字符数量。

```
letunusualMenagerie = "Koala , Snail , Penguin , Dromedary "  
  
println("unusualMenagerie has \ \(countElements(unusualMenagerie)) characters")  
  
// prints "unusualMenagerie has 40 characters"
```

注意：

不同的 `Unicode` 字符以及相同 `Unicode` 字符的不同表示方式可能需要不同数量的内存空间来存储，所以 `Swift` 中的字符在一个字符串中并不一定占用相同的内存空间。因此，字符串的长度不得通过迭代字符串中每一个字符的长度来进行计算。如果您正在处理一个长字符串，需要注意 `countElements` 函数必须遍历字符串中的字符以精准计算字符串的长度。

另外需要注意的是通过 `countElements` 返回的字符数量并不总是与包含相同字符的 `NSString` 的 `length` 属性相同。`NSString` 的 `length` 属性是基于利用 `UTF-16` 表示的十六位代码单元数字，而不是基于 `Unicode` 字符。为了解决这个问题，`NSString` 的 `length` 属性在被 `Swift` 的 `String` 访问时会成为 `utf16count`。

```
</syntaxhighlight>
```

==连接字符串和字符==

字符串和字符的值可以通过加法运算符 (+) 相加在一起并创建一个新的字符串值:

```
<syntaxhighlight>
```

```
let string1 = "hello"
```

```
let string2 = " there"
```

```
let character1: Character = "!"
```

```
let character2: Character = "?"
```

```
let stringPlusCharacter = string1 + character1 // 等于 "hello!"
```

```
let stringPlusString = string1 + string2 // 等于 "hello there"
```

```
let characterPlusString = character1 + string1 // 等于 "!hello"
```

```
let characterPlusCharacter = character1 + character2 // 等于 "!"
```

```
</syntaxhighlight>
```

您也可以通过加法赋值运算符 (+=) 将一个字符串或者字符添加到一个已经存在字符串变量上:

```
<syntaxhighlight>
```

```
var instruction = "look over"
```

```
instruction += string2
```

```
// instruction 现在等于 "look over there"  
  
var welcome = "good morning"  
  
welcome += character1  
  
// welcome 现在等于 "good morning!"  
  
</syntaxhighlight>
```

```
<pre>
```

注意：您不能将一个字符串或者字符添加到一个已经存在的字符变量上，因为字符变量只能包含一个字符。

## 字符串插值

字符串插值是一种全新的构建字符串的方式，可以在其中包含常量、变量、字面量和表达式。您插入的字符串字面量的每一项都被包裹在以反斜线为前缀的圆括号中：

```
letmultiplier = 3  
  
letmessage = "\((multiplier) times 2.5 is \((Double(multiplier) * 2.5))"  
  
// message is "3 times 2.5 is 7.5"
```

在上面的例子中，`multiplier` 作为 `\((multiplier)` 被插入到一个字符串字面量中。当创建字符串执行插值计算时此占位符会被替换为 `multiplier` 实际的值。

`multiplier` 的值也作为字符串中后面表达式的一部分。该表达式计算 `Double(multiplier) * 2.5` 的值并将结果 (7.5) 插入到字符串中。在这个例子中，表达式写为 `\((Double(multiplier) * 2.5)` 并包含在字符串字面量中。

注意：您插值字符串中写在括号中的表达式不能包含非转义双引号 (") 和反斜杠 (\)，并且不能包含回车或换行符。

# 比较字符串

Swift 提供了三种方式来比较字符串的值：字符串相等，前缀相等和后缀相等。

## 字符串相等

如果两个字符串以同一顺序包含完全相同的字符，则认为两者字符串相等：

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    println("These two strings are considered equal")
}
// prints "These two strings are considered equal"
```

## 前缀/后缀相等

通过调用字符串的 `hasPrefix/hasSuffix` 方法来检查字符串是否拥有特定前缀/后缀。两个方法均需要以字符串作为参数传入并传出 **Boolean** 值。两个方法均执行基本字符串和前缀/后缀字符串之间逐个字符的比较操作。

下面的例子以一个字符串数组表示莎士比亚话剧《罗密欧与朱丽叶》中前两场的场景位置：

```
let romeoAndJuliet = [
    "Act 1 Scene 1: Verona, A public place",
    "Act 1 Scene 2: Capulet's mansion",
    "Act 1 Scene 3: A room in Capulet's mansion",
    "Act 1 Scene 4: A street outside Capulet's mansion",
    "Act 1 Scene 5: The Great Hall in Capulet's mansion",
    "Act 2 Scene 1: Outside Capulet's mansion",
    "Act 2 Scene 2: Capulet's orchard",
    "Act 2 Scene 3: Outside Friar Lawrence's cell",
    "Act 2 Scene 4: A street in Verona",
    "Act 2 Scene 5: Capulet's mansion",
    "Act 2 Scene 6: Friar Lawrence's cell"
]
```

您可以利用 `hasPrefix` 方法来计算话剧中第一幕的场景数：

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        ++act1SceneCount
    }
}
```

```
}  
println("There are \act1SceneCount) scenes in Act 1")  
// prints "There are 5 scenes in Act 1"
```

同样的，利用 `hasPrefix` 方法来计算 `Capulet's mansion` 和 `Friar Lawrence's cell` 数:

```
varmansionCount = 0  
varcellCount = 0  
forscene in romeoAndJuliet {  
    if scene.hasSuffix("Capulet's mansion") {  
        ++mansionCount  
    } else if scene.hasSuffix("Friar Lawrence's cell") {  
        ++cellCount  
    }  
}  
println("\mansionCount) mansion scenes; \cellCount) cell scenes")  
// prints "6 mansion scenes; 2 cell scenes"
```

## 大写和小写字符串

您可以通过字符串的 `uppercaseString` 和 `lowercaseString` 属性来访问一个字符串的大写/小写版本。

```
letnormal = "Could you help me, please?"  
letshouty = normal.uppercaseString  
// shouty 值为 "COULD YOU HELP ME, PLEASE?"  
letwhispered = normal.lowercaseString  
// whispered 值为 "could you help me, please?"
```

## Unicode

**Unicode** 是文本编码和表示的国际标准。它使您可以用标准格式表示来自任意语言几乎所有的字符，并能够对文本文件或网页这样的外部资源中的字符进行读写操作。

**Swift** 的字符串和字符类型是完全兼容 **Unicode** 的，它支持如下所述的一系列不同的 **Unicode** 编码。

### Unicode 术语(Terminology)

**Unicode** 中每一个字符都可以被解释为一个或多个 **unicode** 标量。字符的 **unicode** 标量是一个唯一的21位数字(和名称)，例如 `U+0061` 表示小写的拉丁字母A ("a")，`U+1F425` 表示正面站立的鸡宝宝 ("🐣")

当 **Unicode** 字符串被写进文本文件或其他存储结构当中，这些 **unicode** 标量将会按照 **Unicode** 定义的集中格式之一进行编码。其包括 **UTF-8** (以8位代码单元进行编码) 和 **UTF-16** (以16位代码单元进行编码)。

# 字符串的 Unicode 表示

Swift 提供了几种不同的方式来访问字符串的 Unicode 表示。

您可以利用 `for-in` 来对字符串进行遍历，从而以 Unicode 字符的方式访问每一个字符值。该过程在 `Working with Characters` 中进行了描述。

另外，能够以其他三种 Unicode 兼容的方式访问字符串的值：

- UTF-8 代码单元集合 (利用字符串的 `utf8` 属性进行访问)
- UTF-16 代码单元集合 (利用字符串的 `utf16` 属性进行访问)
- 21 位的 Unicode 标量值集合 (利用字符串的 `unicodeScalars` 属性进行访问)

下面由 `D o g !` 和 `🐶` (狗脸表情，Unicode 标量为 U+1F436)组成的字符串中的每一个字符代表着一种不同的表示：

```
let dogString = "Dog! 🐶"
```

## UTF-8

您可以通过遍历字符串的 `utf8` 属性来访问它的 UTF-8 表示。其为 `UTF8View` 类型的属性，`UTF8View` 是无符号 8 位 (`UInt8`) 值的集合，每一个 `UInt8` 都是一个字符的 UTF-8 表示：

```
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 240 159 144 182
```

上面的例子中，前四个 10 进制代码单元值 (68, 111, 103, 33) 代表了字符 `D o g` 和 `!`，他们的 UTF-8 表示与 ASCII 表示相同。后四个代码单元值 (240, 159, 144, 182) 是 `🐶` 的 4 位 UTF-8 表示。

## UTF-16

您可以通过遍历字符串的 `utf16` 属性来访问它的 UTF-16 表示。其为 `UTF16View` 类型的属性，`UTF16View` 是无符号 16 位 (`UInt16`) 值的集合，每一个 `UInt16` 都是一个字符的 UTF-16 表示：

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 55357 56374
```

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符 `D o g` 和 `!`，他们的 UTF-16 代码单元和 UTF-8 完全相同。第五和第六个代码单元值 (55357 and 56374) 是狗脸表情 字符的 UTF-16 表示。第一个值为 U+D83D (十进制值为 55357)，第二个值为 U+DC36 (十进制值为 56374)。

## Unicode 标量 (Scalars)

您可以通过遍历字符串的 `unicodeScalars` 属性来访问它的 Unicode 标量表示。其为 `UnicodeScalarView` 类型的属性，`UnicodeScalarView` 是 `UnicodeScalar` 的集合。`UnicodeScalar` 是21位的 Unicode 代码点。

每一个 `UnicodeScalar` 拥有一个值属性，可以返回对应的21位数值，用 `UInt32` 来表示。

```
forscalar in dogString.unicodeScalars {
    print("\(scalar.value) ")
}
print("\n")
// 68 111 103 33 128054
```

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符 `D o g` 和 `!`。第五位数值，128054，是一个十六进制1F436的十进制表示。其等同于狗脸表情 的 Unicode 标量 U+1F436。

作为查询字符值属性的一种替代方法，每个 `UnicodeScalar` 值也可以用来构建一个新的字符串值，比如在字符串插值中使用：

```
forscalar in dogString.unicodeScalars {
    println("\(scalar) ")
}
// D
// o
// g
// !
//
```

# Collection Types

## 目录

### [隐藏](#)

- [1 集合类型](#)
- [2 数组](#)
- [3 数组类型简写](#)

### [语法](#)

### [4 数组体](#)

## 集合类型

Swift 提供了两种集合类型：数组和字典序集合，用来存储一系列的值。数组存储相同类型的有序的值。字典序集合存储相同类型的无序的值，这些值可以通过一个唯一值（就是 **key** 啦）来搜索获取到。

在 Swift 中，存储在数组和字典序集合的 **key** 和值的类型总是很明确的。这意味着你不能插入一个错误的类型到数组或者字典序集合中去。这也意味着在你从数组或者字典序集合遍历值的时候你是很明确这些值的类型的。Swift 的明确类型集合确保了你在编写代码时，值的类型总是明确有效的，而且也确保了在开发环境中就能捕捉到类型错误。

注意：

当 Swift 中的数组是分配给常量或者变脸，或者传递给一个函数或者方法是，它的排列方式和其他的不一样。想要更多信息，请看 [Mutability of Collections and Assignment and Copy Behavior for Collection Types](#)

## 数组

数组存储大量的值，这些值的类型一样而且有序排列。相同的值可以在数组中在不同地方出现多次。

Swift 的数组存储的值是明确的。他们不同于 Objective-C 的 `NSArray` 和 `NSMutableArray` 类，这写类是可以存储各种不同的对象，而且不必提供任何这些对象的性质。在 Swift 中，一个特定数组能存储明确的类型值，无论是显示注解类型，还是接口类型，不一定非得是类类型。举个例子：你创建一个 `Int` 类型的数组，那你就不能插入除 `Int` 类型以外的值。Swift 数组是类型安全的，而且总是很明确的知道它所包含的值的类型。

## 数组类型简写语法

Swift 数组类型全写法这样: `Array<SomeType>`,其中 `SomeType` 就是数组允许被存储的类型 (xuan: 就是 java 中的泛型)。你也可以简写一个数组这样: `SomeType[]`。虽然两种写法功能上是一样的,但是简写是你的首选,当你要指定一个类型的数组时,按指南这样来写是很有用的。

## 数组体

你可以初始化一个数组用一个数组体,它是一种简短的写法可以写入一个或者多个值来形成一个数组集合。一个数组体被要求写入一系列的值,多个用逗号分开,用中括号包含:

```
[value 1, value 2, value 3]
```

下面举个例子来创建一个名字叫 `shoppingList` 的数组,该数组用来存储一些 `String` 值:

```
varshoppingList: String[] = ["Eggs", "Milk"]  
// shoppingList 被初始化了,包含了两个值
```

`shoppingList` 变量被声明成“一个存放 `String` 的数组”,就像这样的写法 `String[]`。因为这个特定的数组被指定了 `String` 类型,所有它只能储存 `String` 类型的值。这里,`shoppingList` 数组被初始化进去了两个值("Eggs" 和 "Milk"),直接写在了数组体内。

注意:

`shoppingList` 数组被申明成了一个变量(用 `var` 申明),而不是一个常量(用 `let` 申明)因为在下面的例子中我们会添加更多的 `items` 到这个 `shoppingList` 数组中去。

## Functions

[导航搜索](#) 函数是执行特定任务的代码自包含块。给定一个函数名称标识,当执行其任务时就可以用这个标识来进行"调用"。

Swift 的统一的语法足够灵活来表达任何东西,无论是甚至没有参数名称的简单的 C 风格的函数表达式,还是需要为每个本地参数和外部参数设置复杂名称的 Objective-C 语言风格的函数。参数提供默认值,以简化函数调用,并通过设置在输入输出参数,在函数执行完成时修改传递的变量。

Swift 中的每个函数都有一个类型,包括函数的参数类型和返回类型。您可以方便的使用此类型像任何其他类型一样,

这使得它很容易将函数作为参数传递给其他函数,甚至从函数中返回函数类型。函数也可以写在其他函数中来封装一个嵌套函数用以范围内有用的功能。

## 目录

隐藏

- [1 函数的声明与调用](#)

- [2 函数的参数和返回值](#)

- [2.1 多输入参数](#)

- [2.2 无参函数](#)

- [2.3 没有返回值的函数](#)

- [2.4 多返回值函数](#)

- [3 函数参数名](#)

- [3.1 外部参数名](#)

- [3.2 外部参数名称速记](#)

- [3.3 参数的默认值](#)

- [3.4 有默认值的外部名称参数](#)

- [3.5 可变参数](#)

- [3.6 常量参数和变量参数](#)

- [3.7 输入-输出参数](#)

- [4 函数类型](#)

- [4.1 使用函数类型](#)

- [4.2 函数类型的参数](#)

- [4.3 函数类型的返回值](#)

- [4.4 嵌套函数](#)

# 函数的声明与调用

当你定义一个函数时,你可以为其定义一个或多个命名,定义类型值作为函数的输入(称为参数),当该函数完成时将传回输出定义的类型(称为作为它的返回类型)。

每一个函数都有一个函数名,用来描述了函数执行的任务。要使用一个函数的功能时,你通过使用它的名称进行“调用”,并通过它的输入值(称为参数)来匹配函数的参数类型。一个函数的提供的参数必须始终以相同的顺序来作为函数参数列表。

例如在下面的例子中被调用的函数 `greetingForPerson`,像它描述的那样 -- 它需要一个人的名字作为输入并返回一句问候给那个人。

```
func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}
```

所有这些信息都汇总到函数的定义中，并以 **func** 关键字为前缀。您指定的函数的返回类型是以箭头->（一个连字符后跟一个右尖括号）以及随后类型的名称作为返回的。

该定义描述了函数的作用是什么，它期望接收什么，以及当它完成返回的结果是什么。该定义很容易让该函数可以让你在代码的其他地方以清晰、明确的方式来调用：

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!"
```

通过括号内 **String** 类型参数值调用 **sayHello** 的函数，如的 **sayHello("Anna")**。由于该函数返回一个字符串值，**sayHello** 的可以包裹在一个 **println** 函数调用中来打印字符串，看看它的返回值，如上图所示。

在 **sayHello** 的函数体开始定义了一个新的名为 **greeting** 的 **String** 常量，并将其设置加上 **personName** 个人姓名组成一句简单的问候消息。然后这个问候函数以关键字 **return** 来传回。只要问候函数被调用时，函数执行完毕是就会返回问候语的当前值。

你可以通过不同的输入值多次调用 **sayHello** 的函数。上面的例子显示了如果它以 **"Anna"** 为输入值，以 **"Brian"** 为输入值会发生什么。函数的返回在每种情况下都是量身定制的问候。

为了简化这个函数的主体，结合消息创建和 **return** 语句用一行来表示：

```
func sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
println(sayHelloAgain("Anna"))
// prints "Hello again, Anna!"
```

## 函数的参数和返回值

在 **swift** 中函数的参数和返回值是非常具有灵活性的。你可以定义任何东西无论是一个简单的仅仅有一个未命名的参数的函数还是那种具有丰富的参数名称和不同的参数选项的复杂函数。

### 多输入参数

函数可以有多个输入参数，把他们写到函数的括号内，并用逗号加以分隔。下面这个函数设置了一个开始和结束索引的一个半开区间，用来计算在范围内有多少元素包含：

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}
```

```
}  
println(halfOpenRangeLength(1, 10))  
// prints "9"
```

## 无参函数

函数并没有要求一定要定义的输入参数。下面就是一个没有输入参数的函数，任何时候调用时它总是返回相同的字符串消息：

```
func sayHelloWorld() -> String {  
    return "hello, world"  
}  
println(sayHelloWorld())  
// prints "hello, world"
```

该函数的定义在函数的名称后还需要括号，即使它不带任何参数。当函数被调用时函数名称也要跟着一对空括号。

## 没有返回值的函数

函数也不需要定义一个返回类型。这里有一个版本的 `sayHello` 的函数，称为 `waveGoodbye`，它会输出自己的字符串值而不是函数返回：

```
func sayGoodbye(personName: String) {  
    println("Goodbye, \ \(personName)!")  
}  
sayGoodbye("Dave")  
// prints "Goodbye, Dave!"
```

因为它并不需要返回一个值，该函数的定义不包括返回箭头（`->`）和返回类型。

### 提示

严格地说，`sayGoodbye` 功能确实还返回一个值，即使没有返回值定义。函数没有定义返回类型但返回了一个 `void` 返回类型的特殊值。它是一个简直是空的元组，实际上零个元素的元组，可以写为 `()`。

当一个函数调用时它的返回值可以忽略不计：

```
func printAndCount(stringToPrint: String) -> Int {  
    println(stringToPrint)  
    return countElements(stringToPrint)  
}  
func printWithoutCounting(stringToPrint: String) {  
    printAndCount(stringToPrint)  
}  
printAndCount("hello, world")  
// prints "hello, world" and returns a value of 12  
printWithoutCounting("hello, world")
```

```
// prints "hello, world" but does not return a value
```

第一个函数 `printAndCount`，打印了一个字符串，然后并以 `Int` 类型返回它的字符数。第二个函数 `printWithoutCounting`，调用的第一个函数，但忽略它的返回值。当第二函数被调用时，字符串消息由第一函数打印了回来，却没有使用其返回值。

#### 提示

返回值可以忽略不计，但对一个函数来说，它的返回值即便不使用还是一定会返回的。在函数体底部 返回时与定义的返回类型的函数不能相容时，如果试图这样做将导致一个编译时错误。

## 多返回值函数

你可以使用一个元组类型作为函数的返回类型返回一个有多个值组成的一个复合作为返回值。

下面的例子定义了一个名为 `count` 函数，用它来计算字符串中基于标准的美式英语中设定使用的元音、辅音以及字符的数量：

```
func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {
    var vowels = 0, consonants = 0, others = 0
    for character in string {
        switch String(character).lowercaseString {
            case "a", "e", "i", "o", "u":
                ++vowels
            case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
                ++consonants
            default:
                ++others
        }
    }
    return (vowels, consonants, others)
}
```

您可以使用此计数函数来对任意字符串进行字符计数，并检索统计总数的元组三个指定 `Int` 值：

```
letttotal = count("some arbitrary string!")
println("\$(total.vowels) vowels and \$(total.consonants) consonants")
// prints "6 vowels and 13 consonants"
```

需要注意的是在这一点上元组的成员不需要被命名在该该函数返回的元组中，因为他们的名字已经被指定为函数的返回类型的一部分。

# 函数参数名

所有上面的函数都为参数定义了参数名称:

```
func someFunction(parameterName: Int) {  
    // function body goes here, and can use parameterName  
    // to refer to the argument value for that parameter  
}
```

然而, 这些参数名的仅能在函数本身的主体内使用, 在调用函数时, 不能使用。这些类型的参数名称被称为本地的参数, 因为它们只适用于函数体中使用。

## 外部参数名

有时当你调用一个函数将每个参数进行命名是非常有用的, 以表明你传递给函数的每个参数的目的。

如果你希望用户函数调用你的函数时提供参数名称, 除了设置本地地的参数名称, 也要为每个参数定义外部参数名称。

你写一个外部参数名称在它所支持的本地参数名称之前, 之间用一个空格来分隔:

```
func someFunction(externalParameterName localParameterName: Int) {  
    // function body goes here, and can use localParameterName  
    // to refer to the argument value for that parameter  
}
```

### 提示

如果您为参数提供一个外部参数名称, 调用该函数时外部名称必须始终被使用。

作为一个例子, 考虑下面的函数, 它通过插入他们之间的第三个"joiner"字符串来连接两个字符串:

```
func join(s1: String, s2: String, joiner: String) -> String {  
    return s1 + joiner + s2  
}
```

当你调用这个函数, 你传递给函数的三个字符串的目的就不是很清楚了:

```
join("hello", "world", ",")  
// returns "hello, world"
```

为了使这些字符串值的目的更为清晰, 为每个 `join` 函数参数定义外部参数名称:

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)  
    -> String {  
    return s1 + joiner + s2
```

```
}
```

在这个版本的 `join` 函数中, 第一个参数有一个外部名称 `string` 和一个本地名称 `s1`; 第二个参数有一个外部名称 `toString` 和一个本地名称 `s2`; 第三个参数有一个外部名称 `withJoiner` 和一个本地名称 `joiner`。

现在, 您可以使用这些外部参数名称调用清楚明确的调用该函数:

```
join(string: "hello", toString: "world", withJoiner: ", ")  
// returns "hello, world"
```

使用外部参数名称使 `join` 函数的第二个版本功能更富有表现力, 用户习惯使用 `sentence-like` 的方式, 同时还提供了一个可读的、意图明确的函数体。

#### 提示

考虑到使用外部参数名称的初衷就是为了在别人第一次阅读你的代码时并不知道你函数参数的目的是什么。但当函数调用时如果每个参数的目的是明确的和毫不含糊的, 你并不需要指定外部参数名称。

## 外部参数名称速记

如果你想为一个函数参数提供一个外部参数名, 然而本地参数名已经使用了一个合适的名称了, 你不需要为该参数写相同的两次名称。取而代之的是, 写一次名字, 并用一个 `hash` 符号 (`#`) 作为名称的前缀。这告诉 `Swift` 使用该名称同时作为本地参数名称和外部参数名称。

这个例子定义了一个名为 `containsCharacter` 的函数, 定义了两个参数的外部参数名称并通过放置一个散列标志在他们本地参数名称之前:

```
func containsCharacter(#string: String, #characterToFind: Character) -> Bool {  
    for character in string {  
        if character == characterToFind {  
            return true  
        }  
    }  
    return false  
}
```

这个函数选择的参数名称清晰的、函数体极具可读性, 使的该函数被调用时没有歧义:

```
let containsAVee = containsCharacter(string: "aardvark", characterToFind: "v")  
// containsAVee equals true, because "aardvark" contains a "v"
```

## 参数的默认值

可以为任何参数设定默认值来作为函数的定义的一部分。如果默认值已经定义, 调用函数时就可以省略该参数的传值。

提示

将使用默认值的参数放在函数的参数列表的末尾。这确保了所有调用函数的非默认参数使用相同的顺序,并明确地表示在每种情况下相同的函数调用。

这里有一个版本,是早期的 `join` 函数,并为参数 `joiner` 设置了默认值:

```
func join(string s1: String, toString s2: String,
  withJoiner joiner: String = " ") -> String {
  return s1 + joiner + s2
}
```

如果在 `join` 函数被调用时提供给 `joiner` 一个字符串值,该字符串是用来连接两个字符串,就跟以前一样:

```
join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world"
```

但是,如果当函数被调用时提供了 `joiner` 的没有值,就会使用单个空格 (" ") 的默认值:

```
join(string: "hello", toString: "world")
// returns "hello world"
```

## 有默认值的外部名称参数

在大多数情况下,为所有参数提供一个外部带有默认值的参数的名称是非常有用的(因此要求)。这将确保如果当函数被调用时提供的值时参数必须具有明确的目的。

为了使这个过程更容易,当你自己没有提供外部名称时,Swift 自动为所有参数定义了缺省的参数外部名称。自动外部名称与本地名称相同,就好像你在你的代码中的本地名称之前写了一个 `hash` 符号。

这里有一个早期 `join` 函数版本,它不为任何参数提供的外部名称,但仍然提供了 `joiner` 参数的默认值:

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
  return s1 + joiner + s2
}
```

在这种情况下,Swift 自动为一个具有默认值的参数提供了外部参数名称。调用函数时,为使得参数的目的明确、毫不含糊,因此必须提供外部名称:

```
join("hello", "world", joiner: "-")
// returns "hello-world"
```

提示

您可以通过编写一个下划线(\_)有选择进行这种行为,而不是一个明确的定义外部参数名称。然而,在适当情况下有默认值的外部名称参数总是优先被使用。

## 可变参数

一个可变参数的参数接受零个或多个指定类型的值。当函数被调用时,您可以使用一个可变参数的参数来指定该参数可以传递不同数量的输入值。写可变参数的参数时,需要参数的类型名称后加上点字符(...)

传递一个可变参数的参数的值时,函数体中是以提供适当类型的数组的形式存在。例如,一个可变参数的名称为 `numbers` 和类型为 `Double...`在函数体内就作为名为 `numbers` 类型为 `Double[]`的常量数组。

下面的示例计算任意长度的数字的算术平均值(也称为平均):

```
func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
// returns 10.0, which is the arithmetic mean of these three numbers
```

### 提示

函数可以最多有一个可变参数的参数,而且它必须出现在参数列表的最后以避免多参数函数调用时出现歧义。

如果函数有一个或多个参数使用默认值,并且还具有可变参数,将可变参数放在列表的最末尾的所有默认值的参数之后

## 常量参数和变量参数

函数参数的默认值都是常量。试图改变一个函数参数的值会让这个函数体内部产生一个编译时错误。这意味着您不能错误地改变参数的值。

但是,有时函数有一个参数的值的变量副本是非常有用的。您可以通过指定一个或多个参数作为变量参数,而不是避免在函数内部为自己定义一个新的变量。变量参数可以是变量而不是常量,并给函数中新修改的参数的值的提供一个副本。

在参数名称前用关键字 `var` 定义变量参数:

```
func alignRight(var string: String, count: Int, pad: Character) -> String {
    let amountToPad = count - countElements(string)
    for _ in 1..amountToPad {
```

```
    string = pad + string
  }
  return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
// paddedString is equal to "-----hello"
// originalString is still equal to "hello"
```

这个例子定义了一个新函数叫做 **alignRight**,它对准一个输入字符串, 以一个较长的输出字符串。在左侧的空间中填充规定的字符。在该示例中, 字符串 **"hello"** 被转换为字符串 **"-----hello"**。

该 **alignRight** 函数把输入参数的字符串定义成了一个变量参数。这意味着字符串现在可以作为一个局部变量, 用传入的字符串值初始化, 并且可以在函数体中进行相应操作。

函数首先找出有多少字符需要被添加到左边让字符串以右对齐在整个字符串中。这个值存储在本地常量 **amountToPad** 中。该函数然后将填充字符的 **amountToPad** 个字符拷贝到现有的字符串的左边, 并返回结果。整个过程使用字符串变量参数进行字符串操作。

#### 提示

一个变量参数的变化没有超出了每个调用函数,所以对外部函数体是不可见的。变量参数只能存在于函数调用 的生命周期里。

## 输入-输出参数

可变参数, 如上所述, 只能在函数本身内改变。如果你想有一个函数来修改参数的值, 并且想让这些变化要坚持在函数调用结束后, 你就可以定义输入-输出参数来代替。

通过在其参数定义的开始添加 **inout** 关键字写用来标明输入-输出参数。一个在输入-输出参数都有一个传递给函数的值, 由函数修改后, 并从函数返回来替换原来的值。

参数列表中只可以传递一个变量作为一个 **in-out** 参数。不能传递一个常数或常值作为参数, 因为常量和文字不能修改。

你直接在变量名前放置一个连字符 (&), 当你把它作为一个参数传递给一个 **in-out** 参数, 表明它可以通过该功能进行修改。

#### 提示

**in-out** 参数不能有默认值, 可变参数的参数也不能被标记为 **inout**。如果您标记参数为 **inout**, 它不能同时被标记为 **var** 或 **let**。

这里的一个叫做 `swapTwoInts` 函数，它有两个称为 `a` 和 `b` 的输入-输出整数参数：

```
func swapTwoInts(inout a: Int, inout b: Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

`swapTwoInts` 函数只是简单地交换 `a`、`b` 的值。该功能通过存储在一个临时常数称为 `temporaryA` 的值，指定 `b` 的值到 `a`，然后分配 `temporaryA` 到 `b` 执行该交换。

您可以调用交换函数 `swapTwoInts` 来交换任何 `int` 类型的变量以交换它们的值。需要注意的是他们传递给

`swapTwoInts` 执行功能时，`someInt` 和 `anotherInt` 名称前需要加上的前缀符号：

```
var someInt = 3  
var anotherInt = 107  
swapTwoInts(&someInt, &anotherInt)  
println("someInt is now \${someInt}, and anotherInt is now \${anotherInt}")  
// prints "someInt is now 107, and anotherInt is now 3"
```

上面的例子表明，`someInt` 和 `anotherInt` 的原始值由 `swapTwoInts` 函数进行了修改，即使它们定义在函数定义之外。

#### 提示

输入输出参数与从函数返回的值是不一样的。上述 `swapTwoInts` 例子没有定义返回类型或返回一个值，但它仍然会修改 `someInt` 和 `anotherInt` 的值。输入输出参数是一个函数的另一个影响函数体范围之外的方式。

## 函数类型

每一个函数都有特定的函数类型，可以充当参数类型和函数的返回类型。

例如：

```
func addTwoInts(a: Int, b: Int) -> Int {  
    return a + b  
}  
func multiplyTwoInts(a: Int, b: Int) -> Int {  
    return a * b  
}
```

这个例子中定义了两个简单的数学函数 `addTwoInts` 和 `multiplyTwoInts`。每个函数接受两个 `int` 值，并返回一个 `int` 值，执行适当的数学运算并返回结果。

这两个函数的类型是 `(Int, Int) -> Int`。可以解读为：

"这个函数类型，它有两个 `int` 型的参数，并返回一个 `int` 类型的值。"

下面是另一个例子，不带任何参数或返回值的函数：

```
func printHelloWorld() {  
    println("hello, world")  
}
```

这个函数的类型是 `() -> ()`，或者"函数没有参数，并返回 `void`。"函数不显式的指出一个返回值类型是 `void`，在 `swift` 中相当于一个空元组，显示为 `()`。

## 使用函数类型

在 `swift` 中您可以像任何其他类型一样的使用函数类型。例如，你可以定义一个常量或变量为一个函数类型，并指定适当的函数给该变量：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

可以解读为：

"定义一个名为 `mathFunction` 变量，该变量的类型为'一个函数，它接受两个 `int` 值，并返回一个 `int` 值。'设置这个新的变量来引用名为 `addTwoInts` 功能。"

该 `addTwoInts` 函数具有与 `mathFunction` 相同类型的变量，所以这个赋值在能通过 `swift` 的类型检查。

现在你可以调用指定的函数名称为 `mathFunction`：

```
println("Result: \(mathFunction(2, 3))")  
// prints "Result: 5"
```

不同的函数相同的匹配类型可以分配给相同的变量,也同样的适用于非函数性类型:

```
mathFunction = multiplyTwoInts  
println("Result: \(mathFunction(2, 3))")  
// prints "Result: 6"
```

与其他类型一样,你可以把它迅速推断成函数类型当你为常量或变量分配一个函数时:

```
let anotherMathFunction = addTwoInts  
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

## 函数类型的参数

您可以使用一个函数类型，如 `(Int, Int) -> Int` 作为另一个函数的参数类型。这使你预留了一个函数的某些方面的函数实

现，让调用者提供的函数时被调用。

下面就以打印上面的数学函数的结果为例：

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {  
    println("Result: \(${mathFunction(a, b)})")  
}  
printMathResult(addTwoInts, 3, 5)  
// prints "Result: 8"
```

这个例子中定义了一个名为 `printMathResult` 函数，它有三个参数。第一个参数名为 `mathFunction`，类型为 `(Int, Int)->Int`。您可以传入符合的任何函数类型作为此函数的第一个参数。第二和第三个参数 `a`、`b` 都是 `int` 类型。被用作用于提供数学函数的两个输入值。

当 `printMathResult` 被调用时，它传递 `addTwoInt` 函数，以及整数值3和5。它调用的值3和5所提供的功能，并打印8的结果。

`printMathResult` 的作用是调用一个适当类型的数学函数并打印相应结果。那是什么功能的实现其实并不重要，你只要给以正确的类型匹配就行。这使 `printMathResult` 以调用者类型安全的方式转换了函数的功能。

## 函数类型的返回值

您可以使用一个函数类型作为另一个函数的返回类型。返回的函数- (>) 即你的返回箭头后，立即写一个完整的函数类型就做到这一点。

下面的例子定义了两个简单的函数调用 `stepForward` 和 `stepBackward`。该 `stepForward` 函数返回一个值高于其输入值，而 `stepBackward` 函数返回一个值低于其输入值。这两个函数都有一个相同的类型 `(Int) -> Int`：

```
func stepForward(input: Int) -> Int {  
    return input + 1  
}  
func stepBackward(input: Int) -> Int {  
    return input - 1  
}
```

这里有一个 `chooseStepFunction` 函数，它的返回类型是"函数类型 `(Int) -> Int`"。 `chooseStepFunction` 返回一个基于布尔参数的 `stepBackward` 或 `stepForward` 函数类型：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {  
    return backwards ? stepBackward : stepForward  
}
```

您现在可以使用 `chooseStepFunction` 获取一个函数,可能是加一函数或另一个:

```
varcurrentValue = 3
letmoveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

`epFunction` 返回 `stepBackward` 函数。返回函数的引用存储在一个称为 `moveNearerToZero` 常量里。

如今 `moveNearerToZero` 执行了正确的功能，就可以用来计数到零：前面的例子可以判断正负的步骤决定是否需要移动一个名为使得 `currentValue` 变量逐步接近零。`currentValue` 初始值是3，这意味着当前值 $>0$ ，则返回 `true`，`chooseSt`

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// 3...
// 2...
// 1...
// zero!
```

## 嵌套函数

迄今为止所有你在本章中遇到函数都是全局函数，在全局范围内定义。其实你还可以在其他函数中定义函数，被称为嵌套函数。

嵌套函数默认对外界是隐藏的，但仍然可以调用和使用其内部的函数。内部函数也可以返回一个嵌套函数，允许在嵌套函数内的另一个范围内使用。

你可以重写上面的 `chooseStepFunction` 例子使用并返回嵌套函数：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
varcurrentValue = -4
letmoveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// -4...
// -3...
// -2...
// -1...
```

```
// zero!
```

## Closures

闭包是功能性自包含模块，可以在代码中被传递和使用。Swift 中的闭包与 C 和 Objective-C 中的 blocks 以及其他一些编程语言中的 lambdas 比较相似。

闭包可以捕获和存储其所在上下文中任意常量和变量的引用。这就是所谓的闭合并包裹着这些常量和变量，俗称闭包。Swift 会为您管理在捕获过程中涉及到的内存操作。

注意：

如果您不熟悉捕获 (capturing) 这个概念也不用担心，后面会详细对其进行介绍。

在函数章节中介绍的全局和嵌套函数实际上也是特殊的闭包，闭包采取如下三种形式之一：

- 全局函数是一个有名字但不会捕获任何值的闭包
- 嵌套函数是一个有名字并可以捕获其封闭函数域内值的闭包
- 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的没有名字的闭包

Swift 的闭包表达式拥有简洁的风格，并鼓励在常见场景中进行语法优化，主要优化如下：

- 利用上下文推断参数和返回值类型
- 单表达式闭包可以省略 return 关键字
- 参数名称缩写
- Trailing 闭包语法

## 目录

[隐藏](#)

- [1 闭包表达式](#)

### [2 sort 函数](#)

#### [2.1 闭包表达式语法](#)

#### [2.2 根据上下文推断类型](#)

#### [2.3 单行表达式闭包可以省略 return](#)

#### [2.4 参数名称缩写](#)

#### [2.5 运算符函数](#)

- [3 Trailing 闭包](#)
- [4 捕获 \(Capture\)](#)

### [5 闭包是引用类型](#)

# 闭包表达式

嵌套函数是一个在较复杂函数中方便进行命名和定义自包含代码模块的方式。当然，有时候撰写小巧的没有完整定义和命名的类函数结构也是很有用处的，尤其是在您处理一些函数并需要将另外一些函数作为该函数的参数时。

闭包表达式是一种利用简洁语法构建内联闭包的方式。闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。

下面闭包表达式的例子通过使用几次迭代展示了 `sort` 函数定义和语法优化的方式。每一次迭代都用更简洁的方式描述了相同的功能。

## sort 函数

`Swift` 标准库提供了 `sort` 函数，会根据您提供的排序闭包将已知类型数组中的值进行排序。一旦排序完成，函数会返回一个与原数组大小相同的新数组，该数组中包含已经正确排序的同类型元素。

下面的闭包表达式示例使用 `sort` 函数对一个 `String` 类型的数组进行字母逆序排序，以下是初始数组值：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

该例子对一个 `String` 类型的数组进行排序，因此排序闭包需为 `(String, String) -> Bool` 类型的函数。

提供排序闭包的一种方式是在撰写一个符合其类型要求的普通函数，并将其作为 `sort` 函数的第二个参数传入：

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
```

```
varreversed = sort(names, backwards)
// reversed 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** 函数则返回 **true**, 表示在新的数组中 **s1** 应该出现在 **s2** 前。 字符中的 "大于" 表示 "按照字母顺序后出现"。 这意味着字母 **"B"** 大于字母 **"A"**, 字符串 **"Tom"** 大于字符串 **"Tim"**。 其将进行字母逆序排序, **"Barry"** 将会排在 **"Alex"** 之后。

然而, 这是一个相当冗长的方式, 本质上只是写了一个单表达式函数 (**a > b**)。 在下面的例子中, 利用闭合表达式语法可以更好的构造一个内联排序闭包。

## 闭包表达式语法

闭包表达式语法有如下一般形式:

```
{(parameters) -> returnType in
  statements
}
```

闭包表达式语法可以使用常量、变量和 **inout** 类型作为参数, 不提供默认值。 也可以在参数列表的最后使用可变参数。 元组也可以作为参数和返回值。

下面的例子展示了之前 **backwards** 函数对应的闭包表达式版本的代码:

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in
  return s1 > s2
})
```

需要注意的是内联闭包参数和返回值类型声明与 **backwards** 函数类型声明相同。 在这两种方式中, 都写成了 **(s1: String, s2: String) -> Bool**。 然而在内联闭包表达式中, 函数和返回值类型都写在大括号内, 而不是大括号外。

闭包的函数体部分由关键字 **in** 引入。 该关键字表示闭包的参数和返回值类型定义已经完成, 闭包函数体即将开始。

因为这个闭包的函数体部分如此短以至于可以将其改写成一行代码:

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in return s1 > s2 })
```

这说明 **sort** 函数的整体调用保持不变, 一对圆括号仍然包裹住了函数中整个参数集合。 而其中一个参数现在变成了内联闭包 (相比于 **backwards** 版本的代码)。

## 根据上下文推断类型

因为排序闭包是作为函数的参数进行传入的, **Swift** 可以推断其参数和返回值的类型。 **sort** 期望第二个参数是类型为

(String, String) -> Bool 的函数，因此实际上 String, String 和 Bool 类型并不需要作为闭包表达式定义中的一部分。

因为所有的类型都可以被正确推断，返回箭头 (->) 和 围绕在参数周围的括号也可以被省略：

```
reversed = sort(names, { s1, s2 in return s1 > s2 })
```

实际上任何情况下，通过内联闭包表达式构造的闭包作为参数传递给函数时，都可以推断出闭包的参数和返回值类型，这意味着您几乎不需要利用完整格式构造任何内联闭包。

## 单行表达式闭包可以省略 return

单行表达式闭包可以通过隐藏 return 关键字来隐式返回单行表达式的结果，如上版本的例子可以改写为：

```
reversed = sort(names, { s1, s2 in s1 > s2 })
```

在这个例子中，sort 函数的第二个参数函数类型明确了闭包必须返回一个 Bool 类型值。因为闭包函数体只包含了一个单一表达式 (s1 > s2)，该表达式返回 Bool 类型值，因此这里没有歧义，return 关键字可以省略。

## 参数名称缩写

Swift 自动为内联函数提供了参数名称缩写功能，您可以通过 \$0,\$1,\$2 来顺序调用闭包的参数。

如果您在闭包表达式中使用参数名称缩写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成：

```
reversed = sort(names, { $0 > $1 })
```

在这个例子中，\$0 和 \$1 表示闭包中第一个和第二个 String 类型的参数。

## 运算符函数

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。Swift 的 String 类型定义了关于大于号 (>) 的字符串实现，其作为一个函数接受两个 String 类型的参数并返回 Bool 类型的值。而这正好与 sort 函数的第二个参数需要的函数类型相符合。因此，您可以简单地传递一个大于号，Swift 可以自动推断出您想使用大于号的字符串函数实现：

```
reversed = sort(names, >)
```

更多关于运算符表达式的内容请查看 [Operator Functions](#) 。

# Trailing 闭包

如果您需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用 **trailing** 闭包来增强函数的可读性。

**Trailing** 闭包是一个书写在函数括号之外(之后)的闭包表达式，函数支持将其作为最后一个参数调用。

```
func someFunctionThatTakesAClosure(closure: () -> ()) {  
    // 函数体部分  
}
```

// 以下是不使用 **trailing** 闭包进行函数调用

```
someFunctionThatTakesAClosure({  
    // 闭包主体部分  
})
```

// 以下是使用 **trailing** 闭包进行函数调用

```
someFunctionThatTakesAClosure() {  
    // 闭包主体部分  
}
```

注意：

如果函数只需要闭包表达式一个参数，当您使用 **trailing** 闭包时，您甚至可以把 `()` 省略掉。

在上例中作为 **sort** 函数参数的字符串排序闭包可以改写为：

```
reversed = sort(names) { $0 > $1 }
```

当闭包非常长以至于不能在一行中进行书写时，**Trailing** 闭包变得非常有用。举例来说，**Swift** 的 **Array** 类型有一个 **map** 方法，其获取一个闭包表达式作为其唯一参数。数组中的每一个元素调用一次该闭包函数，并返回该元素所映射的值(也可以是不同类型的值)。具体的映射方式和返回值类型由闭包来指定。

当提供给数组闭包函数后，**map** 方法将返回一个新的数组，数组中包含了与原数组一一对应的映射后的值。

下例介绍了如何在 **map** 方法中使用 **trailing** 闭包将 **Int** 类型数组 `[16,58,510]` 转换为包含对应 **String** 类型的数组 `["OneSix", "FiveEight", "FiveOneZero"]`：

```
letdigitNames = [  
    0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four",  
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
```

```
]
letnumbers = [16, 58, 510]
```

如上代码创建了一个数字位和他们名字映射的英文版本字典。同时定义了一个准备转换为字符串的整型数组。

您现在可以通过传递一个 **trailing** 闭包给 **numbers** 的 **map** 方法来创建对应的字符串版本数组。需要注意的是调用 **numbers.map** 不需要在 **map** 后面包含任何括号，因为其只需要传递闭包表达式这一个参数，并且该闭包表达式参数通过 **trailing** 方式进行撰写：

```
letstrings = numbers.map {
  (var number) -> String in
  var output = ""
  while number > 0 {
    output = digitNames[number % 10]! + output
    number /= 10
  }
  return output
}
// strings 常量被推断为字符串类型数组，即 String[]
// 其值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

**map** 在数组中为每一个元素调用了闭包表达式。您不需要指定闭包的输入参数 **number** 的类型，因为可以通过要映射的数组类型进行推断。

闭包 **number** 参数被声明为一个变量参数（变量的具体描述请参看 **Constant and Variable Parameters**），因此可以在闭包函数体内对其进行修改。闭包表达式制定了返回类型为 **String**，以表明存储映射值的新数组类型为 **String**。

闭包表达式在每次被调用的时候创建了一个字符串并返回。其使用求余运算符 (**number % 10**) 计算最后一位数字并利用 **digitNames** 字典获取所映射的字符串。

注意：

字典 **digitNames** 下标后跟着一个叹号 (!)，因为字典下标返回一个可选值 (optional value)，表明即使该 **key** 不存在也不会查找失败。在上例中，它保证了 **number % 10** 可以总是作为一个 **digitNames** 字典的有效下标 **key**。因此叹号可以用于强制展开 (force-unwrap) 存储在可选下标项中的 **String** 类型值。

从 **digitNames** 字典中获取的字符串被添加到输出的前部，逆序建立了一个字符串版本的数字。（在表达式 **number % 10** 中，如果 **number** 为16，则返回6，58返回8，510返回0）。

**number** 变量之后除以10。因为其是整数，在计算过程中未除尽部分被忽略。因此 16变成了1，58变成了5，510

变成了51。

整个过程重复进行，直到 `number /= 10` 为0，这时闭包会将字符串输出，而 `map` 函数则会将字符串添加到所映射的数组中。

上例中 `trailing` 闭包语法在函数后整洁封装了具体的闭包功能，而不再需要将整个闭包包裹在 `map` 函数的括号内。

## 捕获 (Capture)

闭包可以在其定义的上下文中捕获常量或变量。即使定义这些常量和变量的原域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

**Swift** 最简单的闭包形式是嵌套函数，也就是定义在其他函数的函数体内的函数。嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

下例为一个叫做 `makeIncrementor` 的函数，其包含了一个叫做 `incrementor` 嵌套函数。嵌套函数 `incrementor` 从上下文中捕获了两个值，`runningTotal` 和 `amount`。之后 `makeIncrementor` 将 `incrementor` 作为闭包返回。每次调用 `incrementor` 时，其会以 `amount` 作为增量增加 `runningTotal` 的值。

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

`makeIncrementor` 返回类型为 `() -> Int`。这意味着其返回的是一个函数，而不是一个简单类型值。该函数在每次调用时不接受参数只返回一个 `Int` 类型的值。关于函数返回其他函数的内容，请查看 [Function Types as Return Types](#)。

`makeIncrementor` 函数定义了一个整型变量 `runningTotal` (初始为0) 用来存储当前跑步总数。该值通过 `incrementor` 返回。

`makeIncrementor` 有一个 `Int` 类型的参数，其外部命名为 `forIncrement`，内部命名为 `amount`，表示每次 `incrementor` 被调用时 `runningTotal` 将要增加的量。

`incrementor` 函数用来执行实际的增加操作。该函数简单地使 `runningTotal` 增加 `amount`，并将其返回。

如果我们单独看这个函数，会发现看上去不同寻常：

```
func incrementor() -> Int {  
    runningTotal += amount  
    return runningTotal  
}
```

`incrementor` 函数并没有获取任何参数，但是在函数体内访问了 `runningTotal` 和 `amount` 变量。这是因为其通过捕获在包含它的函数体内已经存在的 `runningTotal` 和 `amount` 变量而实现。

由于没有修改 `amount` 变量，`incrementor` 实际上捕获并存储了该变量的一个副本，而该副本随着 `incrementor` 一同被存储。

然而，因为每次调用该函数的时候都会修改 `runningTotal` 的值，`incrementor` 捕获了当前 `runningTotal` 变量的引用，而不是仅仅复制该变量的初始值。捕获一个引用保证了当 `makeIncrementor` 结束时候并不会消失，也保证了当下一次执行 `incrementor` 函数时，`runningTotal` 可以继续增加。

注意：

`Swift` 会决定捕获引用还是拷贝值。您不需要标注 `amount` 或者 `runningTotal` 来声明在嵌入的 `incrementor` 函数中的使用方式。`Swift` 同时也处理 `runningTotal` 变量的内存管理操作，如果不再被 `incrementor` 函数使用，则会被清除。

下面为一个使用 `makeIncrementor` 的例子：

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

该例子定义了一个叫做 `incrementByTen` 的常量，该常量指向一个每次调用会加10的 `incrementor` 函数。调用这个函数多次可以得到以下结果：

```
incrementByTen()  
// 返回的值为10  
incrementByTen()  
// 返回的值为20  
incrementByTen()  
// 返回的值为30
```

如果您创建了另一个 `incrementor`，其会有一个属于自己的独立的 `runningTotal` 变量的引用。下面的例子中，

`incrementBySeven` 捕获了一个新的 `runningTotal` 变量，该变量和 `incrementByTen` 中捕获的变量没有任何联系：

```
let incrementBySeven = makeIncrementor(forIncrement: 7)  
incrementBySeven()
```

```
// 返回的值为7
incrementByTen()
// 返回的值为40
```

注意：

如果您闭包分配给一个类实例的属性，并且该闭包通过指向该实例或其成员来捕获了该实例，您将创建一个在闭包和实例间的强引用环。 **Swift** 使用捕获列表来打破这种强引用环。更多信息，请参考 [Strong Reference Cycles for Closures](#)。

## 闭包是引用类型

上面的例子中，`incrementBySeven` 和 `incrementByTen` 是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量值。这是因为函数和闭包都是引用类型。

无论您将函数/闭包赋值给一个常量还是变量，您实际上都是将常量/变量的值设置为对应函数/闭包的引用。上面的例子中，`incrementByTen` 指向闭包的引用是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包：

```
letalsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// 返回的值为50
```

## Deinitialization

### 反初始化

在一个类的实例被释放之前，反初始化函数被立即调用。用关键字 `deinit` 来标示反初始化函数，类似于初始化函数用 `init` 来标示。反初始化函数只适用于类类型。

### 反初始化原理

**Swift** 会自动释放不再需要的实例以释放资源。如自动引用计数那一章描述，**Swift** 通过自动引用计数（ARC）处理实例的内存管理。通常当你的实例被释放时不需要手动的去清理。但是，当使用自己的资源时，您可能需要进行一些额外的清理。例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，您可能需要在类实例被释放之前关闭该文件。

在类的定义中，每个类最多只能有一个反初始化函数。反初始化函数不带任何参数，在写法上不带括号：

```
deinit {  
    // 执行反初始化  
}
```

反初始化函数是在实例释放发生前一步被自动调用。不允许主动调用自己的反初始化函数。子类继承了父类的反初始化函数，并且在子类反初始化函数实现的最后，父类的反初始化函数被自动调用。即使子类没有提供自己的反初始化函数，父类的反初始化函数也总是被调用。

因为直到实例的反初始化函数被调用时，实例才会被释放，所以反初始化函数可以访问所有请求实例的属性，并且根据那些属性可以修改它的行为(比如查找一个需要被关闭的文件的名称)。

## 反初始化函数操作

这里是一个反初始化函数操作的例子。这个例子是一个简单的游戏，定义了两种新类型，**Bank** 和 **Player**。**Bank** 结构体管理一个虚拟货币的流通，在这个流通中 **Bank** 永远不可能拥有超过**10,000**的硬币。在这个游戏中有且只能有一个 **Bank** 存在，因此 **Bank** 由带有静态属性和静态方法的结构体实现，从而存储和管理其当前的状态。

```
struct Bank {  
    static var coinsInBank = 10_000  
    static func vendCoins(var numberOfCoinsToVend: Int) -> Int {  
        numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)  
        coinsInBank -= numberOfCoinsToVend  
        return numberOfCoinsToVend  
    }  
    static func receiveCoins(coins: Int) {  
        coinsInBank += coins  
    }  
}
```

**Bank** 根据它的 **coinsInBank** 属性来跟踪当前它拥有的硬币数量。银行还提供两个方法—**vendCoins** 和 **receiveCoins**，用来处理硬币的分发和收集。

**vendCoins** 方法在 **bank** 分发硬币之前检查是否有足够的硬币。如果没有足够多的硬币，**bank** 返回一个比请求时小的数字(如果没有硬币留在 **bank** 中就返回0)。**vendCoins** 方法声明 **numberOfCoinsToVend** 为一个变量参数，这样就可以在方法体的内部修改数字，而不需要定义一个新的变量。**vendCoins** 方法返回一个整型值，表明了提供的硬币的实际数目。

**receiveCoins** 方法只是将 **bank** 的硬币存储和接收到的硬币数目相加，再保存回 **bank**。

**Player** 类描述了游戏中的一个玩家。每一个 **player** 在任何时刻都有一定数量的硬币存储在他们的钱包中。这通过 **player** 的 **coinsInPurse** 属性来体现:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
        Bank.receiveCoins(coinsInPurse)
    }
}
```

每个 **Player** 实例都由一个指定数目硬币组成的启动额度初始化，这些硬币在 **bank** 初始化的过程中得到。如果没有足够的硬币可用，**Player** 实例可能收到比指定数目少的硬币。

**Player** 类定义了一个 **winCoins** 方法，该方法从 **bank** 获取一定数量的硬币，并把它们添加到 **player** 的钱包。**Player** 类还实现了一个反初始化函数，这个反初始化函数在 **Player** 实例释放前一步被调用。这里反初始化函数只是将 **player** 的所有硬币都返回给 **bank**：

```
var playerOne: Player? = Player(coins: 100)
println("A new player has joined the game with \ (playerOne!.coinsInPurse) coins")
// 输出 "A new player has joined the game with 100    coins"
println("There are now \ (Bank.coinsInBank) coins left    in the bank")
// 输出 "There are now 9900 coins left in the bank"
```

一个新的 **Player** 实例随着一个100个硬币(如果有)的请求而被创建。这个 **Player** 实例存储在一个名为 **playerOne** 的可选 **Player** 变量中。这里使用一个可选变量，是因为 **players** 可以随时离开游戏。设置为可选使得你可以跟踪当前是否有 **player** 在游戏中。

因为 **playerOne** 是可选的，所以由一个感叹号(!)来修饰，每当其 **winCoins** 方法被调用时，**coinsInPurse** 属性被访问并打印出它的默认硬币数目。

```
playerOne!.winCoins(2_000)
println("PlayerOne won 2000 coins & now has \    (playerOne!.coinsInPurse) coins")
// 输出 "PlayerOne won 2000 coins & now has 2100 coins"
println("The bank now only has \ (Bank.coinsInBank) coins left")
// 输出 "The bank now only has 7900 coins left"
```

这里，**player** 已经赢得了2,000硬币。**player** 的钱包现在有2,100硬币，**bank** 只剩余7,900硬币。

```
playerOne = nil
```

```
println("PlayerOne has left the game")  
// 输出 "PlayerOne has left the game"  
println("The bank now has \(Bank.coinsInBank) coins")  
// 输出 "The bank now has 10000 coins"
```

`player` 现在已经离开了游戏。这表明是要将可选的 `playerOne` 变量设置为 `nil`，意思是"没有 `Player` 实例"。当这种情况发生的时候，`playerOne` 变量对 `Player` 实例的引用被破坏了。没有其它属性或者变量引用 `Player` 实例，因此为了清空它占用的内存从而释放它。在这发生前一步，其反初始化函数被自动调用，其硬币被返回到 `bank`。

## Automatic Reference Counting

---

`Swift` 使用自动引用计数(**ARC**)来跟踪并管理应用使用的内存。大部分情况下，这意味着在 `Swift` 语言中，内存管理"仍然工作"，不需要自己去考虑内存管理的事情。当实例不再被使用时，**ARC** 会自动释放这些类的实例所占用的内存。

然而，在少数情况下，为了自动的管理内存空间，**ARC** 需要了解关于你的代码片段之间关系的更多信息。本章描述了这些情况，并向大家展示如何打开 **ARC** 来管理应用的所有内存空间。

注意

引用计数只应用在类的实例。结构体(**Structure**)和枚举类型是值类型，并非引用类型，不是以引用的方式来存储和传递的。

## 目录

隐藏

- [1 ARC 如何工作](#)
- [2 ARC 实践](#)
- [3 类实例间的强引用环](#)

### [4 解决实例间的强引用环](#)

#### [4.1 弱引用](#)

#### [4.2 无主引用](#)

#### [4.3 无主引用以及显式展开的可选属性](#)

- [5 闭包产生的强引用环](#)

### [6 解决闭包产生的强引用环](#)

#### [6.1 定义占有列表](#)

#### [6.2 弱引用和无主引用](#)

# ARC 如何工作

每次创建一个类的实例，**ARC** 就会分配一个内存块，用来存储这个实例的相关信息。这个内存块保存着实例的类型，以及这个实例相关的属性的值。

当实例不再被使用时，**ARC** 释放这个实例使用的内存，使这块内存可作它用。这保证了类实例不再被使用时，它们不会占用内存空间。

但是，如果 **ARC** 释放了仍在使用的实例，那么你就不能再访问这个实例的属性或者调用它的方法。如果你仍然试图访问这个实例，应用极有可能会崩溃。

为了保证不会发生上述的情况，**ARC** 跟踪与类的实例相关的属性、常量以及变量的数量。只要有一个有效的引用，**ARC** 都不会释放这个实例。

为了让这变成现实，只要你将一个类的实例赋值给一个属性或者常量或者变量，这个属性、常量或者变量就是这个实例的强引用(**strong reference**)。之所以称之为“强”引用，是因为它强持有这个实例，并且只要这个强引用还存在，就不允许销毁实例。

# ARC 实践

下面的例子展示了 **ARC** 是如何工作的。本例定义了一个简单的类，类名是 **Person**，并定义了一个名为 **name** 的常量属性：

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        println("\(name) is being initialized")
    }

    deinit {
        println("\(name) is being deinitialized")
    }
}
```

类 **Person** 有一个初始化函数(initializer), 设置这个实例的 **name** 属性, 打印一条消息来指示初始化正在进行。类 **Person** 还有一个 **deinitializer** 方法, 当销毁一个类的实例时, 会打印一条消息。

接下来的代码片段定义了三个 **Person?** 类型的变量, 这些变量用来创建多个引用, 这些引用都引用紧跟着的代码所创建的 **Person** 对象。因为这些变量都是可选类型 (**Person?**, 而非 **Person**), 因此他们都被自动初始化为 **nil**, 并且当前并没有引用一个 **Person** 的实例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

现在我们创建一个新的 **Person** 实例, 并且将它赋值给上述三个变量中的一个:

```
reference1 = Person(name: "John Appleseed")
// prints "Jonh Appleseed is being initialized"
```

注意, 消息 "John Appleseed is being initialized" 在调用 **Person** 类的初始化函数时打印。这印证初始化确实发生了。

因为 **Person** 的实例赋值给了变量 **reference1**, 所以 **reference1** 是 **Person** 实例的强引用。又因为至少有这一个强引用, **ARC** 就保证这个实例会保存在内存中而不会被销毁。

如果将这个 **Person** 实例赋值给另外的两个变量, 那么将建立另外两个指向这个实例的强引用:

```
reference2 = reference1
reference3 = reference2
```

现在, 这一个 **Person** 实例有三个强引用。

如果你通过赋值 **nil** 给两个变量来破坏其中的两个强引用 (包括原始的引用), 只剩下一个强引用, 这个 **Person** 实例也不会被销毁:

```
reference1 = nil  
reference2 = nil
```

直到第三个也是最后一个强引用被破坏,ARC 才会销毁 **Person** 的实例,这时,有一点非常明确,你无法继续使用 **Person** 实例:

```
referenece3 = nil  
// 打印 “John Appleseed is being deinitialized”
```

## 类实例间的强引用环

在上面的例子中,ARC 可以追踪 **Person** 实例的引用数量,并且在它不再被使用时销毁这个实例。

然而,我们有可能写出这样的代码,一个类的实例永远不会有0个强引用。在两个类实例彼此保持对方的强引用,使得每个实例都使对方保持有效时会发生这种情况。我们称之为强引用环。

通过用弱引用或者无主引用来取代强引用,我们可以解决强引用环问题。在开始学习如何解决这个问题之前,理解它产生的原因会很有帮助。

下面的例子展示了一个强引用环是如何在不经意之间产生的。例子定义了两个类,分别叫 **Person** 和 **Apartment**,这两个类建模了一座公寓以及它的居民:

```
classPerson {  
    let name: String  
    init(name: String) { self.name = name }  
    var apartment: Apartment?  
    deinit { println("\(name) is being deinitialized") }  
}  
  
classApartment {  
    let number: Int  
    init(number: Int) { self.number = number }  
    var tenant: Person?  
    deinit { println("Apartment #\(number) is being deinitialized") }  
}
```

每个 **Person** 实例拥有一个 **String** 类型的 **name** 属性以及一个被初始化为 **nil** 的 **apartment** 可选属性。**apartment** 属性是可选的,因为一个人并不一定拥有一座公寓。

类似的,每个 **Apartment** 实例拥有一个 **Int** 类型的 **number** 属性以及一个初始化为 **nil** 的 **tenant** 可选属性。**tenant** 属性是可选的,因为一个公寓并不一定有居民。

这两个类也都定义了初始化函数,打印消息表明这个类的实例正在被初始化。这使你能够看到 **Person** 和 **Apartment** 的实例是否像预期的那样被销毁了。

下面的代码片段定义了两个可选类型变量，`john` 和 `number73`，分别被赋值为特定的 `Apartment` 和 `Person` 的实例。

得益于可选类型的优点，这两个变量初始值均为 `nil`：

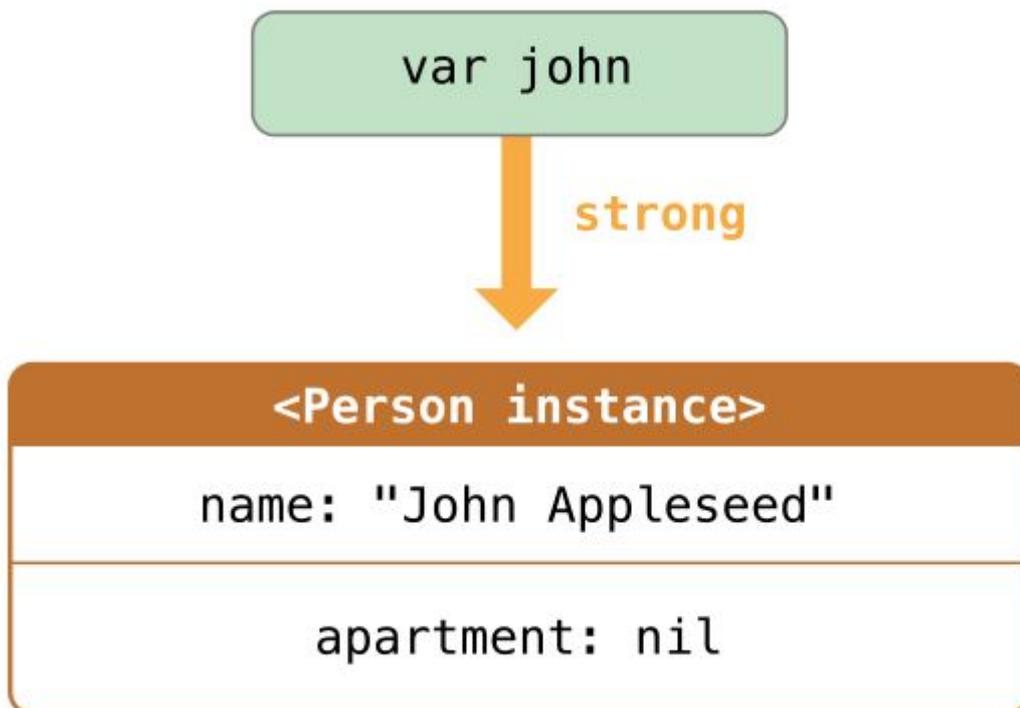
```
var john: Person?
var number73: Apartment?
```

现在，你可以创建特定的 `Person` 实例以及 `Apartment` 实例，并赋值给 `john` 和 `number73`：

```
john = Person(name: "John Appleseed")
number73 = Apartments(number: 73)
```

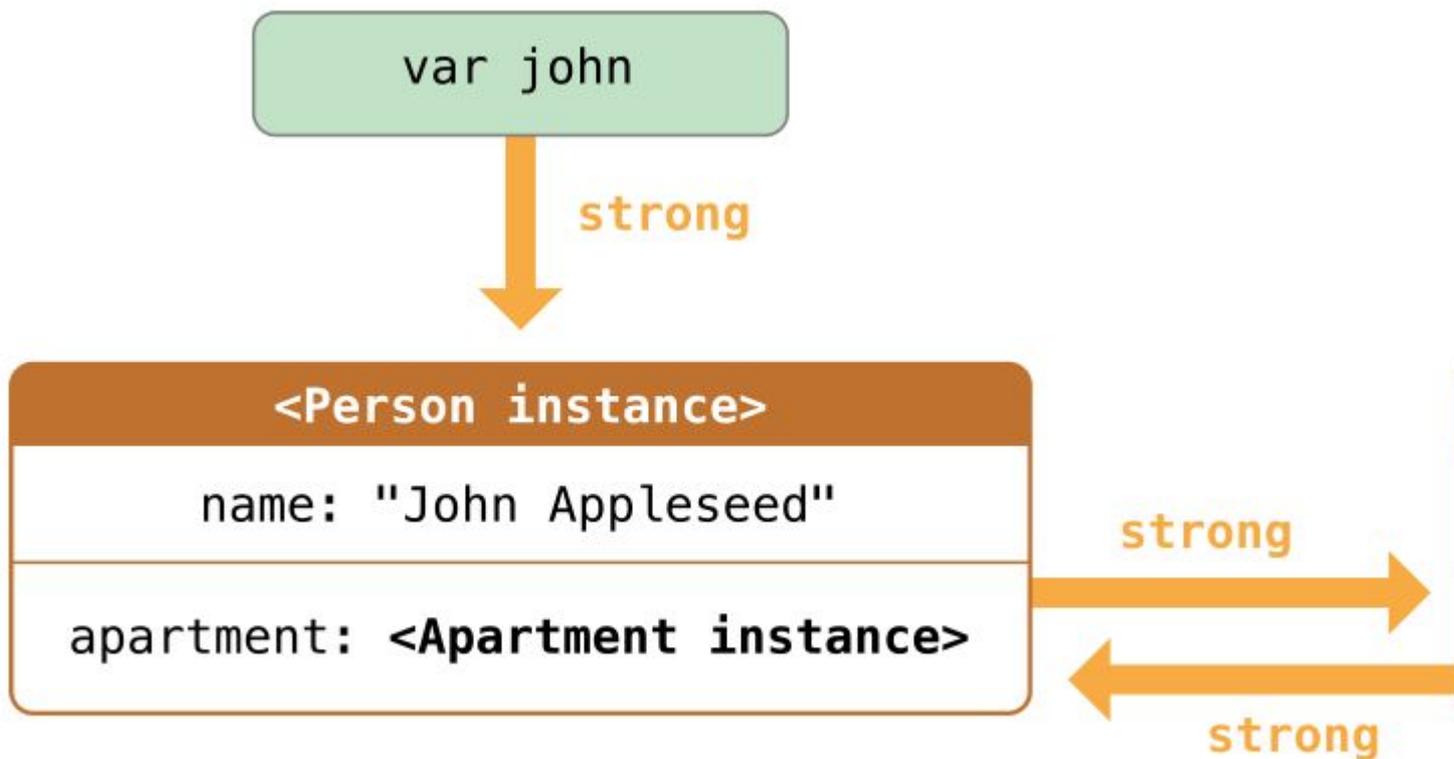
下面的图表明了创建以及赋值这两个实例后强引用的关系。`john` 拥有一个 `Person` 实例的强引用，`number73` 拥有一个 `Apartment` 实例的强引用：

现在你可以将两个实例关联起来，一个人拥有一所公寓，一个公寓也拥有一个房客。注意：用感叹号 (!) 来展开并访问可选类型的变量，只有这样这些变量才能被赋值：



```
john!.apartment = number73
number73!.tenant = john
```

两个实例关联起来后，强引用关系如下图所示：



糟糕的是，关联这俩实例生成了一个强引用环，**Person** 实例和 **Apartment** 实例各持有一个对方的强引用。因此，即使你破坏 **john** 和 **number73**所持有的强引用，引用计数也不会变为0，因此 **ARC** 不会销毁这两个实例：

```
john = nil
nuber73 = nil
```

注意，当上面两个变量赋值为 **nil** 时，没有调用任何一个 **deinitializer**。强引用环阻止了 **Person** 和 **Apartment** 实例的销毁，进一步导致内存泄漏。

此时强引用关系如下图所示：

```
var john
```



Person 和 Apartment 实例之间的强引用依然存在。

## 解决实例间的强引用环

Swift 提供两种方法来解决强引用环：弱引用和无主引用。

弱引用和无主引用允许引用环中的一个实例引用另外一个实例，但不是强引用。因此实例可以互相引用但是不会产生强引用环。

对于生命周期中引用会变为 nil 的实例，使用弱引用；对于初始化时赋值之后引用再也不会赋值为 nil 的实例，使用无主引用。

### 弱引用

弱引用不会增加实例的引用计数，因此不会阻止 ARC 销毁被引用的实例。这种特性使得引用不会变成强引用环。声明属性或者变量的时候，关键字 **weak** 表明引用为弱引用。

在实例的生命周期中，如果某些时候引用没有值，那么弱引用可以阻止强引用环。如果整个生命周期内引用都有值，那么相应的用无主引用，在无主引用这一章中有详细描述。在上面的 Apartment 例子中，有时一个 Apartment 实例可能没有房客，因此此处应该用弱引用。

**注意**  
弱引用只能声明为变量类型，因为运行时它的值可能改变。弱引用绝对不能声明为常量。

因为弱引用可以没有值，所以声明弱引用的时候必须是可选类型的。在 Swift 语言中，推荐用可选类型来作为可能没有值的引用的类型。

如前所述，弱引用不会保持实例，因此即使实例的弱引用依然存在，ARC 也有可能销毁实例，并将弱引用赋值为 nil。

你可以想检查其他的可选值一样检查弱引用是否存在，永远也不会碰到引用了也被销毁的实例的情况。

下面的例子和之前的 Person 和 Apartment 例子相似，除了一个重要的区别。这一次，我们声明 Apartment 的 tenant 属性为弱引用：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    weak var tenant: Person?
    deinit { println("Apartment #\(number) is being deinitialized") }
}
```

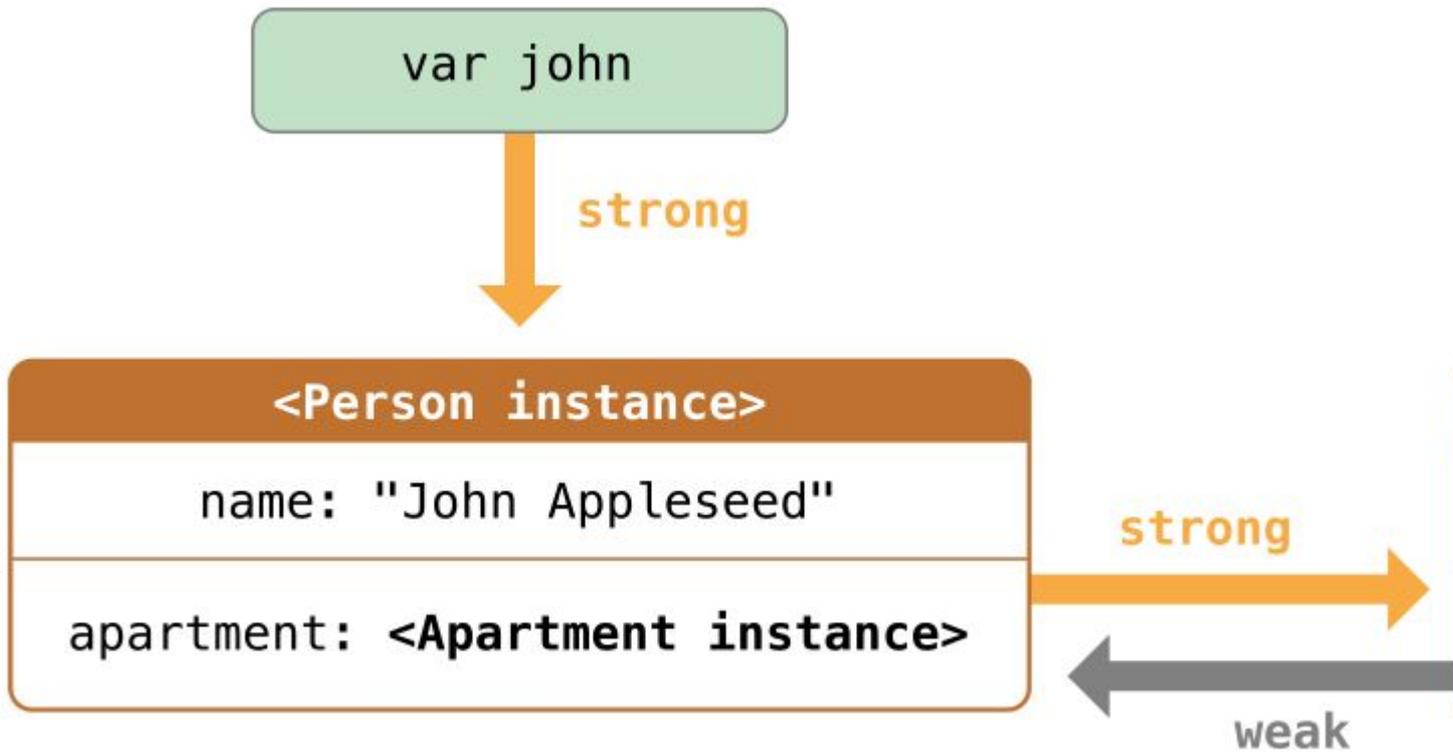
然后创建两个变量(john 和 number73)的强引用，并关联这两个实例：

```
var john: Person?
var number73: Apartment?

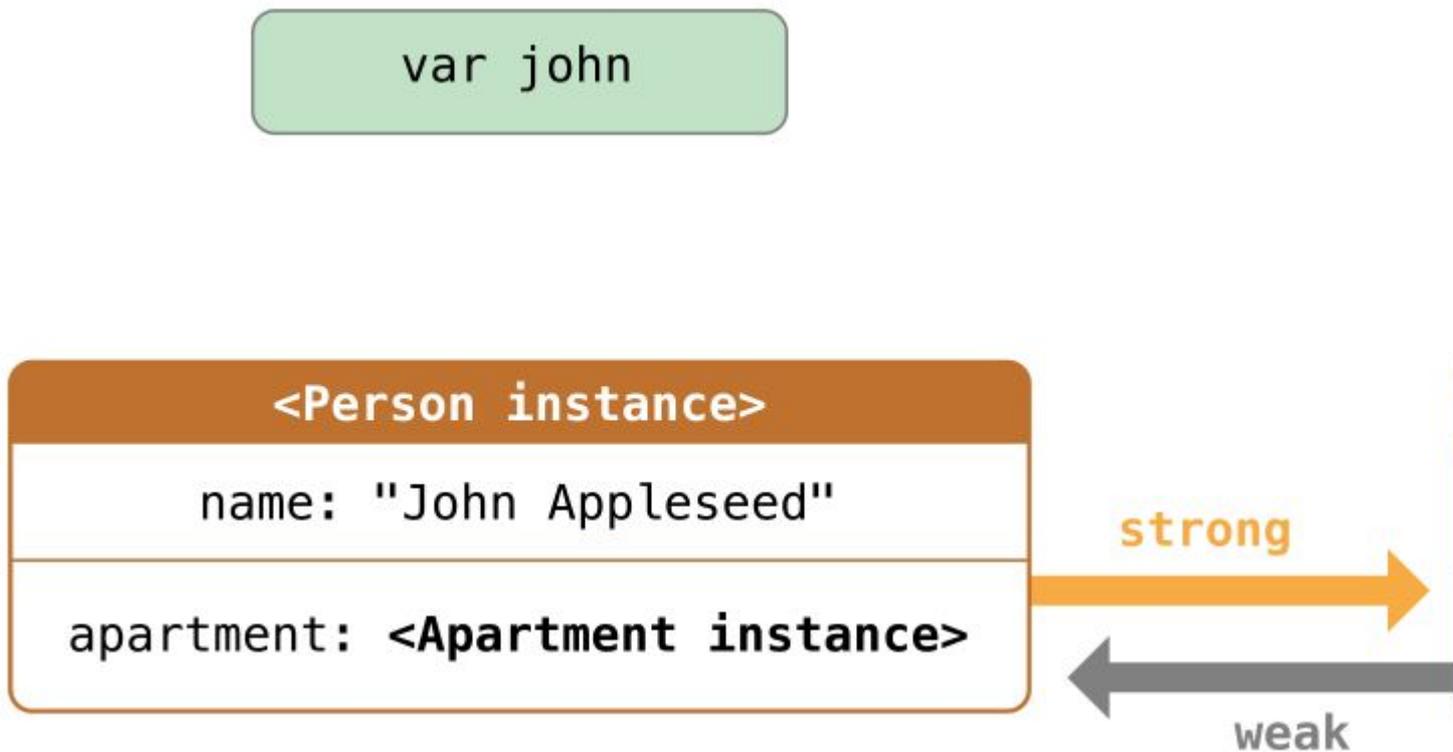
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)

john!.apartment = number73
number73!.tenant = john
```

下面是引用的关系图：



Person 的实例仍然是 Apartment 实例的强引用，但是 Apartment 实例则是 Person 实例的弱引用。这意味着当破坏 john 变量所持有的强引用后，不再存在任何 Person 实例的强引用：



既然不存在 Person 实例的强引用，那么该实例就会被销毁：

```
john = nil
```

```
// 打印"John Appleseed is being deinitialized"
```

只有 `number73` 还持有 `Apartment` 实例的强引用。如果你破坏这个强引用，那么也不存在 `Apartment` 实例的任何强引用：

```
var john
```



这时，`Apartment` 实例也被销毁：

```
number73 = nil
// 打印"Apartment #73 is being deinitialized"
```

上面的两段代码表明在 `john` 和 `number73` 赋值为 `nil` 后，`Person` 和 `Apartment` 实例的 `deinitializer` 都打印了“销毁”的消息。这证明了引用环已经被打破了。

## 无主引用

和弱引用相似，无主引用也不强持有实例。但是和弱引用不同的是，无主引用默认始终有值。因此，无主引用只能定义为非可选类型（`non-optional type`）。在属性、变量前添加 `unowned` 关键字，可以声明一个无主引用。

因为是非可选类型，因此当使用无主引用的时候，不需要展开，可以直接访问。不过非可选类型变量不能赋值为 `nil`，

因此当实例被销毁的时候，ARC 无法将引用赋值为 `nil`。

注意

当实例被销毁后，试图访问该实例的无主引用会触发运行时错误。使用无主引用时请确保引用始终指向一个未销毁的实例。 上面的

非法操作会百分百让应用崩溃，不会发生无法预期的行为。因此，你应该避免这种情况。

接下来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。每个类都有一个属性，存储另外一个类的实例。这样的关系可能会产生强引用环。

**Customer**、**CreditCard** 的关系和之前弱引用例子中的 **Apartment**、**Person** 的关系截然不同。在这个模型中，消费者不一定有信用卡，但是每张信用卡一定对应一个消费者。鉴于这种关系，**Customer** 类有一个可选类型属性 **card**，而 **CreditCard** 类的 **customer** 属性则是非可选类型的。

进一步，要创建一个 **CreditCard** 实例，只能通过传递 **number** 值和 **customer** 实例到定制的 **CreditCard** 初始化函数来完成。这样可以确保当创建 **CreditCard** 实例时总是有一个 **customer** 实例与之关联。

因为信用卡总是对应一个消费者，因此定义 **customer** 属性为无主引用，这样可以避免强引用环：

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }

    deinit { println("\(name) is being deinitialized")
}

class CreditCard {
    let number: Int
    unowned let customer: Customer
    init(number: Int, customer: Customer) {
        self.number = number
        self.customer = customer
    }

    deinit { println("Card #\(number) is being deinitialized")
}
```

下面的代码定义了一个叫 **john** 的可选类型 **Customer** 变量，用来保存某个特定消费者的引用。因为是可变类型，该变量的初始值为 **nil**：

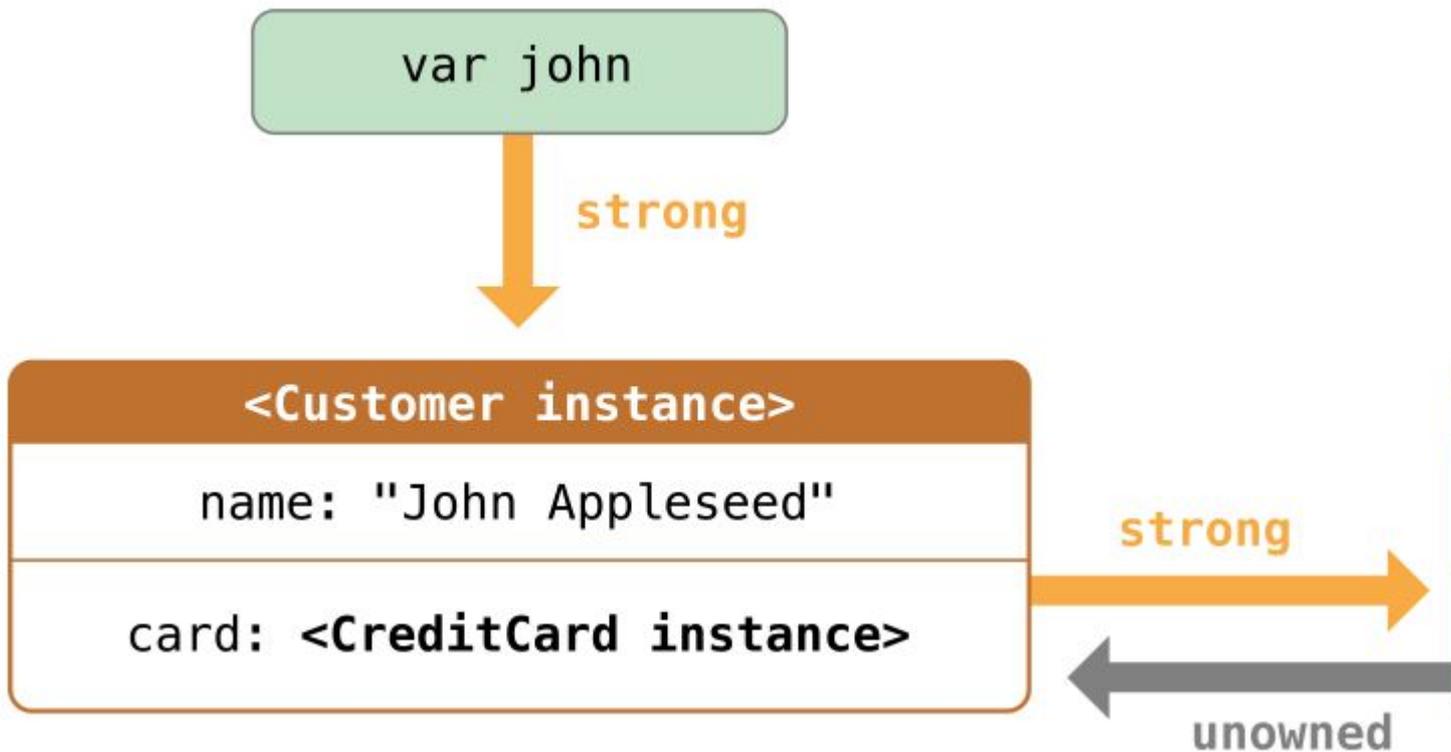
```
var john: Customer?
```

现在创建一个 **Customer** 实例，然后用它来初始化 **CreditCard** 实例，并把刚创建出来的 **CreditCard** 实例赋值给

Customer 的 card 属性:

```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer:john!)
```

我们来看看此时的引用关系:



Customer 实例持有 CreditCard 实例的强引用，而 CreditCard 实例则持有 Customer 实例的无主引用。

因为 customer 的无主引用，当破坏 john 变量持有的强引用时，就没有 Customer 实例的强引用了：

```
var john
```



此时 Customer 实例被销毁。然后，CreditCard 实例的强引用也不复存在，因此 CreditCard 实例也被销毁：

```
john = nil
// 打印"John Appleseed is being deinitialized"
// 打印"Card #1234567890123456 is being deinitialized"
```

上面的代码证明，john 变量赋值为 nil 后，Customer 实例和 CreditCard 实例的 deinitializer 方法都打印了 "deinitialized" 消息。

## 无主引用以及显式展开的可选属性

上述的弱引用和无主引用的例子覆盖了两种常用的需要打破强引用环的应用场景。

Person 和 Apartment 的例子说明了下面的场景：两个属性的值都可能是 nil, 并有可能产生强引用环。这种场景下适合使用弱引用。

Customer 和 CreditCard 的例子则说明了另外的场景：一个属性可以是 nil, 另外一个属性不允许是 nil, 并有可能产生强引用环。这种场景下适合使用无主引用。

但是，存在第三种场景：两个属性都必须有值，且初始化完成后不能为 nil。这种场景下，则要一个类用无主引用属性，另一个类用显式展开的可选属性。

这样，在初始化完成后我们可以立即访问这两个变量（而不需要可选展开），同时又避免了引用环。本节将告诉你应该如何配置这样的关系。

下面的例子顶一个了两个类，Country 和 City，都有一个属性用来保存另外的类的实例。在这个模型里，每个国家都有

首都，每个城市都隶属于一个国家。所以，类 **Country** 有一个 **capitalCity** 属性，类 **City** 有一个 **country** 属性：

```
classCountry {
  let name: String
  let capitalCity: City!
  init(name: String, capitalName: String) {
    self.name = name
    self.capitalCity = City(name: capitalName, country: self)
  }
}

classCity {
  let name: String
  unowned let country: Country
  init(name: String, country: Country) {
    self.name = name
    self.country = country
  }
}
```

**City** 的初始化函数有一个 **Country** 实例参数，并且用 **country** 属性来存储这个实例。这样就实现了上面说的关系。

**Country** 的初始化函数调用了 **City** 的初始化函数。但是，只有 **Country** 的实例完全初始化完后（在 **Two-Phase Initialization**），**Country** 的初始化函数才能把 **self** 传给 **City** 的初始化函数。

为满足这种需求，通过在类型结尾处加感叹号(**City!**)，我们声明 **Country** 的 **capitalCity** 属性为显式展开的可选类型属性。就是说，**capitalCity** 属性的默认值是 **nil**，不需要展开它的值（在 **Implicitly Unwrapped Optionals** 中描述）就可以直接访问。

因为 **capitalCity** 默认值是 **nil**，一旦 **Country** 的实例在初始化时给 **name** 属性赋值后，整个初始化过程就完成了。这代表只要赋值 **name** 属性后，**Country** 的初始化函数就能引用并传递显式的 **self**。所以，当 **Country** 的初始化函数在赋值 **capitalCity** 时，它也可以将 **self** 作为参数传递给 **City** 的初始化函数。

综上所述，你可以在一条语句中同时创建 **Country** 和 **City** 的实例，却不会产生强引用环，并且不需要使用感叹号来展开它的可选值就可以直接访问 **capitalCity**：

```
varcountry = Country(name: "Canada", capitalName: "Ottawa")
println("\(country.name)'s captial city is called \(country.capitalCity.name)")
// 打印"Canada's capital city is called Ottawa"
```

在上面的例子中，使用显式展开的可选项满足了两个类的初始化函数的要求。初始化完成后，`capitalCity` 属性就可以做为非可选值类型使用，却不会产生强引用环。

## 闭包产生的强引用环

前面我们看到了强引用环是如何产生的，还知道了如何引入弱引用和无主引用来打破引用环。

将一个闭包赋值给类实例的某个属性，并且这个闭包使用了实例，这样也会产生强引用环。这个闭包可能访问了实例的某个属性，例如 `self.someProperty`，或者调用了实例的某个方法，例如 `self.someMethod`。这两种情况都导致了闭包使用 `self`，从而产生了强引用环。

因为诸如类这样的闭包是引用类型，导致了强引用环。当你把一个闭包赋值给某个属性时，你也把一个引用赋值给了这个闭包。实质上，这个之前描述的问题是一样的—两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

**Swift** 提供了一种优雅的方法来解决这个问题，我们称之为闭包占用列表(**closure capture list**)。同样的，在学习如何避免因闭包占用列表产生强引用环之前，先来看看这个强引用环是如何产生的。

下面的例子将会告诉你当一个闭包引用了 `self` 后是如何产生一个强引用环的。本例顶一个一个名为 `HTMLElement` 的类，来建模 **HTML** 中的一个单独的元素：

```
classHTMLElement {  
  
    let name: String  
    let text: String?  
  
    @lazy var asHTML: () -> String = {  
        if let text = self.text {  
            return "<\(self.name)>\(text)</\(\self.name)>"  
        } else {  
            return "<\(self.name) />"  
        }  
    }  
  
    init(name: String, text: String? = nil) {  
        self.name = name  
        self.text = text  
    }  
  
    deinit {  
        println("\(name) is being deinitialized")  
    }  
  
}
```

类 `HTMLElement` 定义了一个 `name` 属性来表示这个元素的名称，例如代表段落的“`p`”，或者代表换行的“`br`”；以及一个可选属性 `text`，用来设置 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闭包，将 `name` 和 `text` 组合成 HTML 字符串片段。该属性是 `() -> String` 类型，就是“没有参数，返回 `String` 的函数”。

默认将闭包赋值给了 `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`；或者不包含文本。对于段落，根据 `text` 是“`some text`”还是 `nil`，闭包会返回“

`some text`

”或者“`<p />`”。

可以像实例方法那样去命名、使用 `asHTML`。然而，因为 `asHTML` 终究是闭包而不是实例方法，如果你像改变特定元素的 HTML 处理的话，可以用定制的闭包来取代默认值。

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`。也就是说，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 `lazy` 属性。

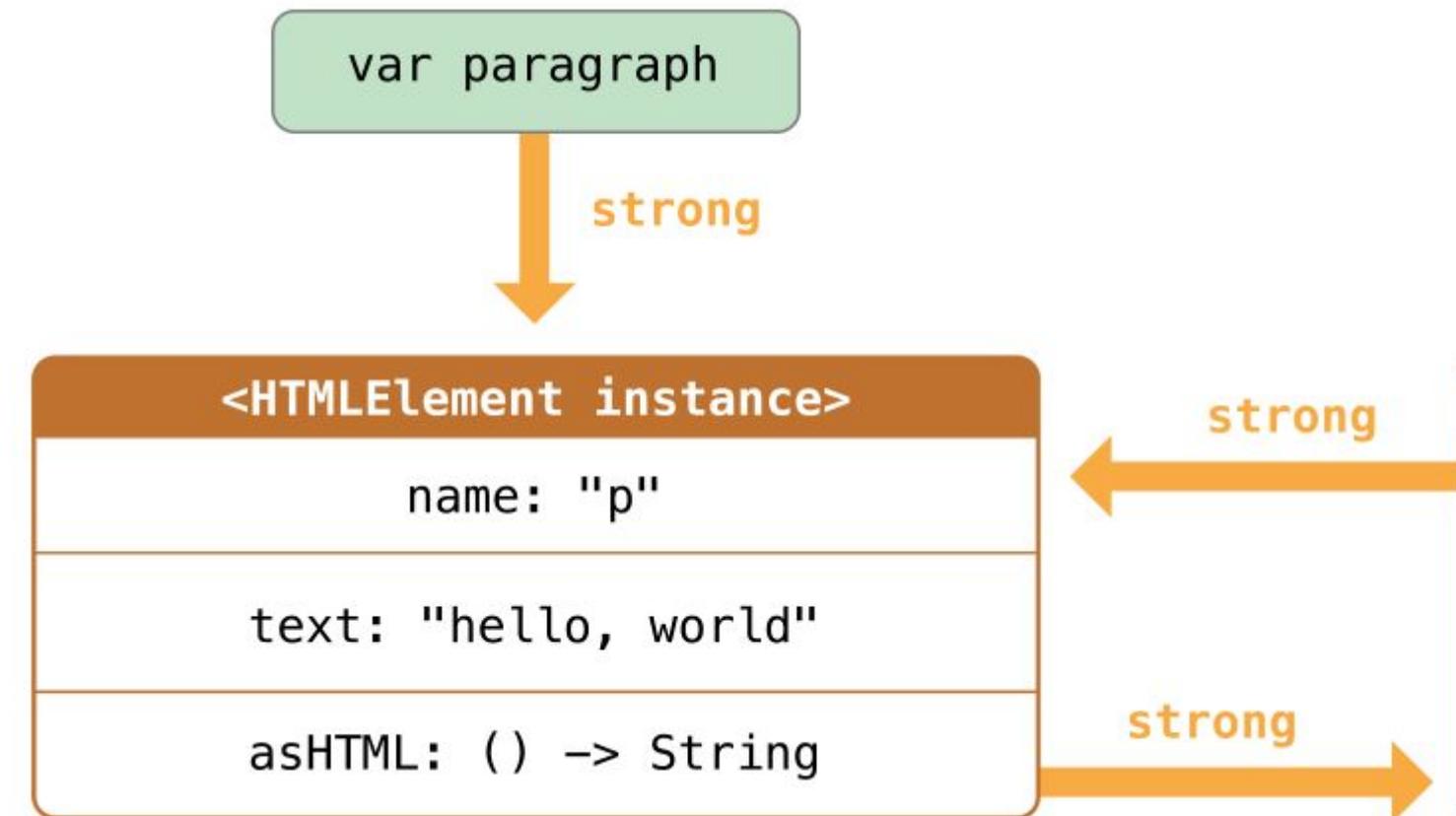
`HTMLElement` 只有一个初始化函数，根据 `name` 和 `text`(如果有的话)参数来初始化一个元素。该类也定义了一个 `deinitializer`，当 `HTMLElement` 实例被销毁时，打印一条消息。

下面的代码创建一个 `HTMLElement` 实例并打印消息。

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world") println(paragraph!.asHTML())  
  
// 打印"  
hello, world  
"
```

注意 上面的 `paragraph` 变量定义为可选 `HTMLElement`，因此我们可以赋值 `nil` 给它来演示强引用环。不幸的是，

`HTMLElement` 类产生了类实例和 `asHTML` 默认值的闭包之间的强引用环。如下图所示：



实例的 `asHTML` 属性持有闭包的强引用。

但是，闭包使用了 `self`（引用了 `self.name` 和 `self.text`），因此闭包占有了 `self`，这意味着闭包又反过来持有了 `HTML<Element` 实例的强引用。这样就产生了强引用环。（更多闭包哪占有值的信息，请参考 [Capturing Values](#)）。

注意

虽然闭包多次使用了 `self`，它只占有 `HTML<Element` 实例的一个强引用。

如果设置 `paragraph` 为 `nil`，打破它持有的 `HTML<Element` 实例的强引用，`HTML<Element` 实例和它的闭包都不会被销毁，就因为强引用环：

```
paragraph = nil
```

注意 `HTML<Elementdeinitializer` 中的消息并没有别打印，印证了 `HTML<Element` 实例并没有被销毁。

## 解决闭包产生的强引用环

在定义闭包时同时定义占有列表作为闭包的一部分，可以解决闭包和类实例之间的强引用环。占有列表定义了闭包内占有一个或者多个引用类型的规则。和解决两个类实例间的强引用环一样，声明每个占有的引用为弱引用或无主引用，而不是强引用。根据代码关系来决定使用弱引用还是无主引用。

注意

Swift 有如下约束：只要在闭包内使用 `self` 的成员，就要用 `self.someProperty` 或者 `self.someMethod`（而非只是 `someProperty` 或 `someMethod`）。这可以提醒你可能会不小心就占有了 `self`。

## 定义占有列表

占有列表中的每个元素都是由 `weak` 或者 `unowned` 关键字和实例的引用(如 `self` 或 `someInstance`)组成。每一对都在花括号中，通过逗号分开。

占有列表放置在闭包参数列表和返回类型之前：

```
@lazy var someClosure: (Int, String) -> String = {
    [unowned self] (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

如果闭包没有指定参数列表或者返回类型（可以通过上下文推断），那么占有列表放在闭包开始的地方，跟着是关键字 `in`：

```
@lazy var someClosure: () -> String = {
    [unowned self] in
    // closure body goes here
}
```

## 弱引用和无主引用

当闭包和占有的实例总是互相引用时并且总是同时销毁时，将闭包内的占有定义为无主引用。

相反的，当占有引用有时可能会是 `nil` 时，将闭包内的占有定义为弱引用。弱引用总是可选类型，并且当引用的实例被销毁后，弱引用的值会自动置为 `nil`。利用这个特性，我们可以在闭包内检查他们是否存在。

注意

如果占有的引用绝对不会置为 `nil`，应该用无主引用，而不是弱引用。

前面提到的 `HTMLElement` 例子中，无主引用是正确的解决强引用的方法。这样编码 `HTMLElement` 类来避免强引用环：

```
classHTMLElement {

    let name: String
    let text: String?

    @lazy var asHTML: () -> String = {
```

```
[unowned self] in
if let text = self.text {
    return "<\(self.name)>\(text)</\(self.name)>"
} else {
    return "<\(self.name) />"
}
}

init(name: String, text: String? = nil) {
    self.name = name
    self.text = text
}

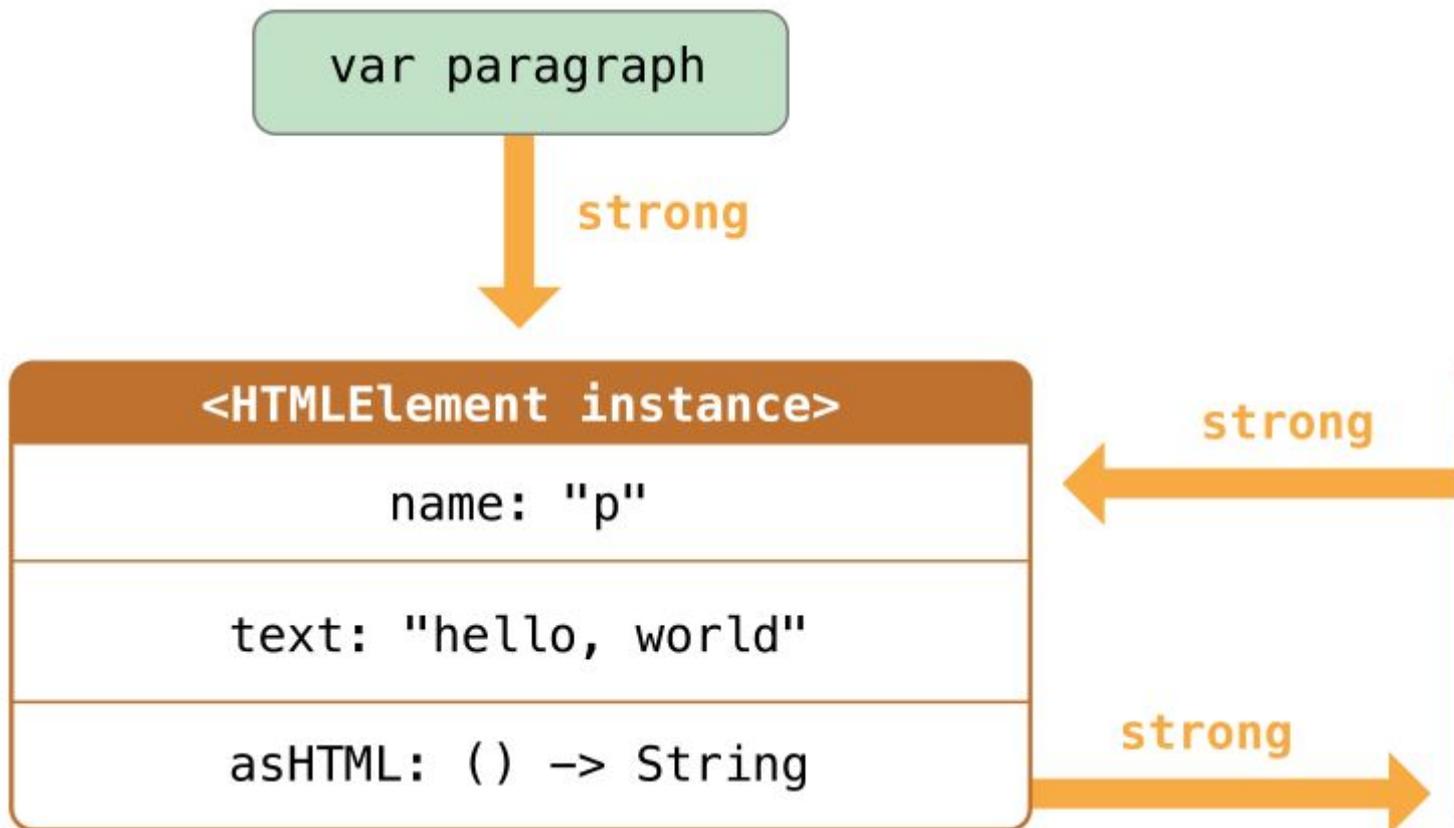
deinit {
    println("\(name) is being deinitialized")
}
}
```

上面的 `HTMLElement` 实现和之前的实现相同，只是多了占有列表。这里，占有列表是 `[unowned self]`，代表“用无主引用而不是强引用来占有 `self`”。

和之前一样，我们可以创建并打印 `HTMLElement` 实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// 打印"<p>hello, world</p>"
```

使用占有列表后引用关系如下图所示：



这一次，闭包以无主引用的形式占有 `self`，并不会持有 `HTMLElement` 实例的强引用。如果赋值 `paragraph` 为 `nil`，`HTMLElement` 实例将会被销毁，并能看到它的 `deinitializer` 打印的消息。

```

paragraph = nil
// 打印"p is being deinitialized"
  
```

## Optional Chaining

供选链接是一种可以请求和调用属性、方法及角标的一种过程，它的供选择性体现于请求或调用的目标当前可能为空（`nil`）。如果选择的目标有值，那么调用就会成功；相反，如果选择的目标为空（`nil`），则这种调用将返回空（`nil`）。多次请求或调用可以被链接在一起形成一个链，如果任何一个节点为空（`nil`）将导致整个链失效。

提示：

`Swift` 的选择链和 `Objective-C` 中的消息为空有些相像，但是 `Swift` 可以使用在任意类型中，并且失败与否可以被检测到。

## 目录

### 隐藏

- [1 供选链接可以作为强制拆包的替代](#)
- [2 为供选链接定义模型类](#)
- [3 通过供选链接调用属性](#)
- [4 通过供选链接调用方法](#)
- [5 使用供选链接调用角标](#)
- [6 连接多层链接](#)

### [7 链接供选返回值的方法](#)

# 供选链接可以作为强制拆包的替代

你可以通过在你想调用属性，方法或下标的供选值（**optional value**）（非空）后面放一个问号来定义一个供选链接。

这一点很像在供选值后面放一个声明符号来强制拆得它的值。他们的主要的区别在于当供选值为空时供选链接即刻失败，然而强制拆包将会引发运行时错误。

为了反映供选链接可以用空（**nil**）调用，供选链接调用的结果通常是一个供选值，即使是你调用的属性方法下标等返回的是非供选值。你可以利用这个返回值来检测你的供选链接是否调用成功，有返回值即成功，返回 **nil** 则失败。

调用供选链接的返回结果与期待的返回结果具有相同的类型，但是被包装成一个供选值，当供选链接调用成功时，一个应该返回 **Int** 的属性将会返回 **Int?**。

下面几段代码将解释供选链接和强制拆包的不同。

首先定义两个类 **Person** 和 **Residence**。 `class Person { var residence: Residence? }`

```
class Residence {  
    var numberOfRooms = 1  
}
```

**Residence** 具有一个 **Int** 属性叫 **numberOfRooms** 其值为1。**Person** 具有一个供选 **residence** 属性，它的类型是 **Residence?**。

如果你创建一个新的 **Person** 实例，它的 **residence** 属性由于是被定义为供选的，此属性将默认初始化为空：

```
let john = Person()
```

如果你想使用声明符！强制拆包获得这个人 **residence** 属性 **numberOfRooms** 属性值，将会引发运行时错误，因为这

时没有可以供拆包的 `residence` 值。

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error”
//将导致运行时错误
```

当 `john.residence` 不是 `nil` 时会通过运行，且会将 `roomCount` 设置为一个 `int` 类型的合理值。然而如上所述，当 `residence` 为空时，这个代码将会导致运行时错误。

供选链接提供了一种另一种获得 `numberOfRooms` 的方法。利用供选链接，使用问号来代替原来 `!` 的位置：

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \$(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// 打印 "Unable to retrieve the number of rooms."。
```

这告诉 `Swift` 来链接供选 `residence` 属性，如果 `residence` 存在则取回 `numberOfRooms` 的值。

因为这种尝试获得 `numberOfRooms` 值得操作有可能失败，供选链接会返回 `Int?` 类型值，或者称作“供选 `Int`”。

当 `residence` 是空的时候（上例），选择 `Int` 将会为空，因此可以反映出无法访问 `numberOfRooms` 的情况。

要注意的是，即使 `numberOfRooms` 是非供选 `Int` (`Int?`) 时这一点也成立。只要是通过供选链接的请求就意味着最后 `numberOfRooms` 总是返回一个 `Int?` 而不是 `Int`。

你可以自己定义一个 `Residence` 实例给 `john.residence`，这样他就不再为空了：

```
john.residence = Residence()
```

`john.residence` 现在有了实际存在的实例而不是 `nil` 了。如果你想使用和前面一样的供选链接来获得 `numberOfRooms`，它将返回一个包含默认值 `1` 的 `Int?`：

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \$(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// 打印 "John's residence has 1 room(s)"。
```

# 为供选链接定义模型类

你可以使用供选链接来多层调用属性，方法，和下标。这让你可以利用相互联系的复杂模型来获取亚层的属性，并检查是否可以成功获取此类亚层属性中的各种属性方法下标等。

后面的代码定义了四个将在后面例子中使用的模型类，其中包括多层供选链接。这些类是由上面的 **Person** 和 **Residence** 模型通过添加一个 **Room** 和一个 **Address** 类拓展来。

**Person** 类定义与之前相同。

```
classPerson {
    var residence: Residence?
}
```

**Residence** 类比之前复杂些。这次，它定义了一个变量 **rooms**，它被初始化为一个 **Room[]**类型的空数组：

```
classResidence {
    var rooms = Room[]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        println("The number of rooms is \(${numberOfRooms}")
    }
    var address: Address?
}
```

因为这个版本的 **Residence** 存储了一个 **Room** 实例的数组，它的 **numberOfRooms** 属性不是一个固定的存储值而是通过计算而来的。**numberOfRooms** 属性是由返回 **rooms** 数组的 **count** 属性值得到的。

为了能快速访问 **rooms** 数组，**Residence** 定义了一个只读的下标，插入数组的元素角标值就可以访问到数组中的对应元素。如果该元素存在，**subscript** 则返回数组中的此元素。

**Residence** 中也提供了一个 **printNumberOfRooms** 的方法，即简单的打印房间个数。

最后，**Residence** 定义了一个供选属性叫 **address** (**address?**)。**Address** 类的属性将在后面定义。用于 **rooms** 数组的 **Room** 类是一个很简单的类，它只有一个 **name** 属性和一个设定 **room** 名的初始化器。

```
classRoom {
    let name: String
```

```
init(name: String) { self.name = name }  
}
```

这个模型中的最终类叫做 **Address** 类。这个类有三个供选属性他们的类型是 **String?**。前面两个供选属性 **buildingName** 和 **buildingNumber** 作为地址的一部分，是定义某个特定的建筑的两种不同方式。第三个属性 **street**，用于命名地址的街道名：

```
class Address {  
    var buildingName: String?  
    var buildingNumber: String?  
    var street: String?  
    func buildingIdentifier() -> String? {  
        if buildingName {  
            return buildingName  
        } else if buildingNumber {  
            return buildingNumber  
        } else {  
            return nil  
        }  
    }  
}
```

**Address** 类还提供了一个 **buildingIdentifier** 的方法，它的返回值类型为 **String?**。这个方法检查 **buildingName** 和 **buildingNumber** 的属性，如果 **buildingName** 有值则将其返回，或者如果 **buildingNumber** 有值则将其返回，再或返回空如果没有一个属性有值。

## 通过供选链接调用属性

正如上面“[Optional Chaining as an Alternative to Forced Unwrapping](#)”中所述，你可以利用供选链接的供选值获取属性，并且检查属性是否获取成功。然而你不能使用供选链接设置属性值。

使用上述定义的类来创建一个人实例，并再次尝试去它的 **numberOfRooms** 属性：

```
let john = Person()  
if let roomCount = john.residence?.numberOfRooms {  
    println("John's residence has \(roomCount) room(s).")  
} else {  
    println("Unable to retrieve the number of rooms.")  
}  
// 打印 "Unable to retrieve the number of rooms."
```

由于 **john.residence** 是空，所以这个供选链接和之前一样编译失败了，但是没有运行时错误。

# 通过供选链接调用方法

你可以使用供选链接的来调用供选值的方法并检查方法调用是否成功。即使这个方法没有返回值，你依然可以使用供选链接来达成这一目的。

`Residence` 的 `printNumberOfRooms` 方法会打印 `numberOfRooms` 的当前值。方法如下：

```
func printNumberOfRooms(){
    println( "The number of rooms is \ \(numberOfRooms)" )
}
```

这个方法没有返回值。但是，没有返回值类型的函数和方法有一个隐式的返回值类型 `Void`（参见 `Function Without Return Values`）。

如果你利用供选链接调用此方法，这个方法的返回值类型将是 `Void?`，而不是 `Void`，因为当通过供选链接调用方法时返回值总是供选类型（optional type）这可以让你使用 `if` 语句来检查是否能成功调用 `printNumberOfRooms` 方法，即使是这个方法本是没有定义返回值；如果方法通过供选链接调用成功，`printNumberOfRooms` 的隐式返回值将会是 `Void`，如果没有成功，将返回 `nil`：

```
if john.residence?.printNumberOfRooms() {
    println("It was possible to print the number of rooms.")
}else{
    println("It was not possible to print the number of rooms.")
}
// 打印 "It was not possible to print the number of rooms."。
```

# 使用供选链接调用角标

你可以使用供选链接来尝试从角标获取值并检查角标的调用是否成功，然而你不能通过供选链接来设置角标。

提示：

当你使用供选链接来获取角标的时候，你应该将问号放在角标括号的前面而不是后面。供选链接的问号一般直接跟着供选表达语句的后面。

下面这个例子试图使用在 `Residence` 类中定义的角标来获取 `john.residence` 数组中第一个房间的名字。因为 `john.residence` 现在是 `nil`，角标的调用失败了。

```
if let firstRoomName = john.residence?[0].name {
    println("The first room name is \ \(firstRoomName).")
}else{
    println("Unable to retrieve the first room name.")
}
```

```
}  
// prints "Unable to retrieve the first room name."
```

在角标调用中供选链接的问号直接跟在 `john.residence` 的后面，在角标括号的前面，因为 `john.residence` 是供选链接试图获得的供选值。

如果你创建一个 `Residence` 实例给 `john.residence`，且在他的 `rooms` 数组中有一个或多个 `Room` 实例，那么你可以使用供选链接通过 `Residence` 角标来获取在 `rooms` 数组中的实例了：

```
let johnsHouse = Residence()  
johnsHouse.rooms += Room(name: "Living Room")  
johnsHouse.rooms += Room(name: "Kitchen")  
john.residence = johnsHouse  
  
if let firstRoomName = john.residence?[0].name {  
    println("The first room name is \(firstRoomName).")  
} else {  
    println("Unable to retrieve the first room name.")  
}  
  
// 打印 "The first room name is Living Room."
```

## 连接多层链接

你可以将多层供选链接连接在一起，可以掘取模型内更下层的属性方法和角标。然而多层供选链接不能再添加比已经返回的供选值更多的层。也就是说：

如果你试图获得类型不是供选类型，由于供选链接它将变成供选类型。

如果你试图获得的类型已经是供选类型，由于供选链接它也不会提高供选性。

因此：

如果你试图通过供选链接获得 `Int` 值，不论使用了多少层链接返回的总是 `Int?`。

相似的，如果你试图通过供选链接获得 `Int?` 值，不论使用了多少层链接返回的总是 `Int?`。

下面的例子试图获取 `john` 的 `residence` 属性里的 `address` 的 `street` 属性。这里使用了两层供选链接来联系 `residence` 和 `address` 属性，他们两者都是供选类型：

```
if let johnsStreet = john.residence?.address?.street {  
    println("John's street name is \(johnsStreet).")  
} else {  
    println("Unable to retrieve the address.")  
}  
  
// 打印 "Unable to retrieve the address."
```

`john.residence` 的值现在包含一个 `Residence` 实例，然而 `john.residence.address` 现在是 `nil`，因此 `john.residence?.address?.street` 调用失败。

从上面的例子发现，你试图获得 `street` 属性值。这个属性的类型是 `String?`。因此尽管在供选类型属性前使用了两层供选链接，`john.residence?.address?.street` 的返回值类型也是 `String?`。

如果你为 `Address` 设定一个实例来作为 `john.residence.address` 的值，并为 `address` 的 `street` 属性设定一个实际值，你可以通过多层供选链接来得到这个属性值。

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence!.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
}else{
    println("Unable to retrieve the address.")
}
// 打印 "John's street name is Laurel Street."。
```

值得注意的是，“!”符的在定义 `address` 实例时的使用（`john.residence.address`）。`john.residence` 属性是一个供选类型，因此你需要在它获取 `address` 属性之前使用! 拆包以获得它的实际值。

## 链接供选返回值的方法

前面的例子解释了如何通过供选链接来获得供选类型属性值。你也可以通过调用返回供选类型值的方法并按需链接次方法的返回值。下面的例子通过供选链接调用了 `Address` 类中的 `buildingIdentifier` 方法。这个方法的返回值类型是 `String?`。如上所述，这个方法在供选链接调用后最终的返回值类型依然是 `String?`：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    println("John's building identifier is \(buildingIdentifier).")
}
// 打印 "John's building identifier is The Larches."。
```

如果你还想进一步对方法返回值执行供选链接，将供选链接问号符放在方法括号的后面：

```
if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
    println("John's uppercase building identifier is \(upper).")
}
// 打印 "John's uppercase building identifier is THE LARCHES."。
```

提示：

在上面的例子中，你将供选链接问号符放在括号后面是因为你想要链接的供选值是 `buildingIdentifier` 方法的返回值，不是 `buildingIdentifier` 方法本身。

## Nested Types

# 类型嵌套

枚举类型常被用于实现特定类或结构体的功能。也能够在有多种变量类型的环境中，方便地定义通用类或结构体来使用，为了实现这种功能，`Swift` 允许你定义类型嵌套，可以在枚举类型、类和结构体中定义支持嵌套的类型。

要在一个类型中嵌套另一个类型，将需要嵌套的类型的定义写在被嵌套类型的区域 `{}` 内，而且可以根据需要定义多级嵌套。

## 类型嵌套实例

下面这个例子定义了一个结构体 `BlackjackCard`(二十一点)，用来模拟 `BlackjackCard` 中的扑克牌点数。`BlackjackCard` 结构体包含2个嵌套定义的枚举类型 `Suit` 和 `Rank`。

在 `BlackjackCard` 规则中，`Ace` 牌可以表示1或者11，`Ace` 牌的这一特征用一个嵌套在枚举型 `Rank` 的结构体 `Values` 来表示。

```
struct BlackjackCard {
    // 嵌套定义枚举型 Suit
    enum Suit: Character {
        case Spades = "♠", Hearts = "♥", Diamonds = "♦", Clubs = "♣"
    }
    // 嵌套定义枚举型 Rank
    enum Rank: Int {
        case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
        case Jack, Queen, King, Ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .Ace:
                return Values(first: 1, second: 11)
            }
        }
    }
}
```

```
case .Jack, .Queen, .King:
    return Values(first: 10, second: nil)
default:
    return Values(first: self.toRaw(), second: nil)
}
}
// BlackjackCard 的属性和方法
let rank: Rank, suit: Suit
var description: String {
var output = "suit is \(suit.toRaw()),"
    output += " value is \(rank.values.first)"
    if let second = rank.values.second {
        output += " or \(second)"
    }
    return output
}
}
```

枚举型的 **Suit** 用来描述扑克牌的四种花色，并分别用一个 **Character** 类型的值代表花色符号。

枚举型的 **Rank** 用来描述扑克牌从 **Ace~10,J,Q,K,13**张牌，并分别用一个 **Int** 类型的值表示牌的面值。(这个 **Int** 类型的值不适用于 **Ace,J,Q,K** 的牌)。

如上文所提到的，枚举型 **Rank** 在自己内部定义了一个嵌套结构体 **Values**。这个结构体包含两个变量，只有 **Ace** 有两个数值，其余牌都只有一个数值。结构体 **Values** 中定义的两个属性：

- **first**, 为 **Int**
- **second**, 为 **Int?**, 或 “optional Int”

**Rank** 定义了一个计算属性 **values**，这个计算属性会根据牌的面值，用适当的数值去初始化 **Values** 实例，并赋值给 **values**。对于 **J,Q,K,Ace** 会使用特殊数值，对于数字面值的牌使用 **Int** 类型的值。

**BlackjackCard** 结构体自身有两个属性—**rank** 与 **suit**，也同样定义了一个计算属性 **description**，**description** 属性用 **rank** 和 **suit** 的中内容来构建对这张扑克牌名字和数值的描述，并用可选类型 **second** 来检查是否存在第二个值，若存在，则在原有的描述中增加对第二数值的描述。

因为 **BlackjackCard** 是一个没有自定义构造函数的结构体，在 **Memberwise Initializers for Structure Types** 中知道结构体有默认的成员构造函数，所以你可以用默认的 **initializer** 去初始化新的常量 **theAceOfSpades**:

```
lettheAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
println("theAceOfSpades: \(${theAceOfSpades.description}")
// 打印出 "theAceOfSpades: suit is ♠, value is 1 or 11"
```

尽管 **Rank** 和 **Suit** 嵌套在 **BlackjackCard** 中，但仍可被引用，所以在初始化实例时能够通过枚举类型中的成员名称单独引用。在上面的例子中 **description** 属性能正确得输出对 **Ace** 牌有**1**和**11**两个值。

## 类型嵌套的引用

在外部对嵌套类型的引用，以被嵌套类型的名字为前缀，加上所要引用的属性名：

```
letheartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
// 红心的符号为 "♥"
```

对于上面这个例子，这样可以使 **Suit**, **Rank**, 和 **Values** 的名字尽可能的短，因为它们的名字会自然的由被定义的上下文来限定。

## Generics

泛型代码可以确保你写出灵活的，可重用的函数和定义出任何你所确定好的需求的类型。你的可以写出避免重复的代码，并且用一种清晰的，抽象的方式表达出来。

泛型是 **Swift** 需要强大特征中的其中一个，许多 **Swift** 标准库是通过泛型代码构建出来的。事实上，你已经使用泛型贯穿整个 **Language Guide**，即便你没有实现它。例如：**Swift** 的 **Array** 和 **Dictionary** 类型都是泛型集。你可以创建一个 **Int** 数组，也可创建一个 **String** 数组，或者甚至于可以是任何其他 **Swift** 的类型数据数组。同样的，你也可以创建存储任何指定类型的字典(dictionary)，而且这些类型可以是没有限制的。

## 目录

### 隐藏

- [1 泛型所解决的问题](#)
- [2 泛型函数](#)
- [3 类型参数](#)
- [4 命名类型参数](#)
- [5 泛型类型](#)

### 6 类型约束

#### [6.1 类型约束语法](#)

#### [6.2 类型约束行为](#)

### 7 关联类型

#### [7.1 关联类型行为](#)

#### [7.2 扩展一个存在的类型为一指定关联类型](#)

### 8 Where 语句

# 泛型所解决的问题

这里是一个标准的，非泛型函数 `swapTwoInts`，用来交换两个 `Int` 值：

```
func swapTwoInts(inout a: Int, inout b: Int)
    let temporaryA = a
    a = b
    b = temporaryA
}
```

这个函数使用 `in-out` 参数交换 `a` 和 `b` 的值，这两个参数被描述为 `In-Out` 类型参数。

`swapTwoInts` 函数可以交换 `b` 的原始值到 `a`，也可以交换 `a` 的原始值到 `b`，你可以调用这个函数交换两个 `Int` 变量值：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "someInt is now 107, and anotherInt is now 3"
```

`swapTwoInts` 函数是非常有用的，但是它只能交换 `Int` 值，如果你想要交换两个 `String` 或者 `Double`，就不得不写更

多的函数，如 `swapTwoStrings` 和 `swapTwoDoublesfunctions`，如同如下所示：

```
func swapTwoStrings(inout a: String, inout b: String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(inout a: Double, inout b: Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

你可能注意到 `swapTwoInts`、`swapTwoStrings` 和 `swapTwoDoubles` 函数主题都是相同的，唯一不同之处就在于传入的变量不同，分别是 `Int`、`String` 和 `Double`。

但实际应用中通常需要一个用处更强大并且尽可能的考虑到更多的灵活性单个函数，可以用来交换两个任何类型值，很幸运的是，泛型代码帮你解决了这种问题。（一个这种泛型函数后面已经定义好了。）

#### NOTE

In all three functions, it is important that the types of `a` and `b` are defined to be the same as each other. If `a` and `b` were not of the same type, it would not be possible to swap their values. Swift is a type-safe language, and does not allow (for example) a variable of type `String` and a variable of type `Double` to swap values with each other. Attempting to do so would be reported as a compile-time error.

## 泛型函数

泛型函数可以工作于任何类型，这里是一个上面 `swapTwoInts` 函数的泛型版本，用于交换两个值：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

`swapTwoValues` 函数主体和 `swapTwoInts` 函数是一样，而且，只在第一行稍微有那么一点点不同于 `swapTwoInts`，如下所示：

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

这个函数的泛型版本使用了节点类型命名（通常此情况下用字母 `T` 来表示）来代替实际类型名（如 `Int`、`String` 或 `Double`）。节点类型名并不是表示 `T` 必须是任何类型，但是其规定 `a` 和 `b` 必须是同一类型的 `T`，而不管 `T` 表示任何类型。

只有 `swapTwoValues` 函数在每次调用时所传入的实际类型决定了 `T` 所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的节点类型名(`T`)是用尖括号括起来的`()`。这个尖括号告诉 `Swift` 那个 `T`

是 `swapTwoValues` 函数所定义的一个节点类型。因为 `T` 是一个节点，`Swift` 不会去查找每一个命名为 `T` 的实际类型。

`swapTwoValues` 函数除了只要传入的两个任何类型值是同一类型外，也可以作为 `swapTwoInts` 函数被调用。每次 `swapTwoValues` 被调用，`T` 所代表的类型值都会传给函数。

在下面的两个例子中，`T` 分别代表 `Int` 和 `String`：

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

#### NOTE

The `swapTwoValues` function defined above is inspired by a generic function called `swap`, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the `swapTwoValues` function in your own code, you can use Swift's existing `swap` function rather than providing your own implementation.

## 类型参数

在上面的 `swapTwoValues` 例子中，节点类型 `T` 是一种类型参数的示例。类型参数指定并命名为一个节点类型，并且紧跟在函数名后面，并用一对尖括号括起来（如 `<T>`）。

一旦一个类型参数被指定，那么其可以被使用来定义一个函数的参数类型（如 `swapTwoValues` 函数中的参数 `a` 和 `b`），或作为一个函数返回类型，或用作函数主体中的注释类型。在这种情况下，被类型参数所代表的节点类型不管函数任何时候被调用，都会被实际类型所替换（在上面 `swapTwoValues` 例子中，当函数第一次被调用时，`T` 被 `Int` 替换，第二次调用时，被 `String` 替换。）。

你可支持多个类型参数，命名在尖括号中，用逗号分开。

## 命名类型参数

在简单的情况下，泛型函数或泛型类型需要指定一个节点类型（如上面的 `swapTwoValues` 泛型函数，或一个存储单一类型的泛型集，如 `Array`），通常用一单个字母 `T` 来命名类型参数。不过，你可以使用任何有效的标识符来作为类型参数名。

如果你使用多个参数定义更复杂的泛型函数或泛型类型，那么使用更多的描述类型参数是非常有用的。例如，**Swift** 字典(**Dictionary**)类型有两个类型参数，一个是 **key**，另外一个值是。如果你自己写字典，你或许会定义这两个类型参数为 **KeyType** 和 **ValueType**，用来记住它们在你的泛型代码中的作用。

#### NOTE

Always give type parameters UpperCamelCase names (such as T and KeyType) to indicate that they are a placeholder for a type, not a value.

## 泛型类型

---

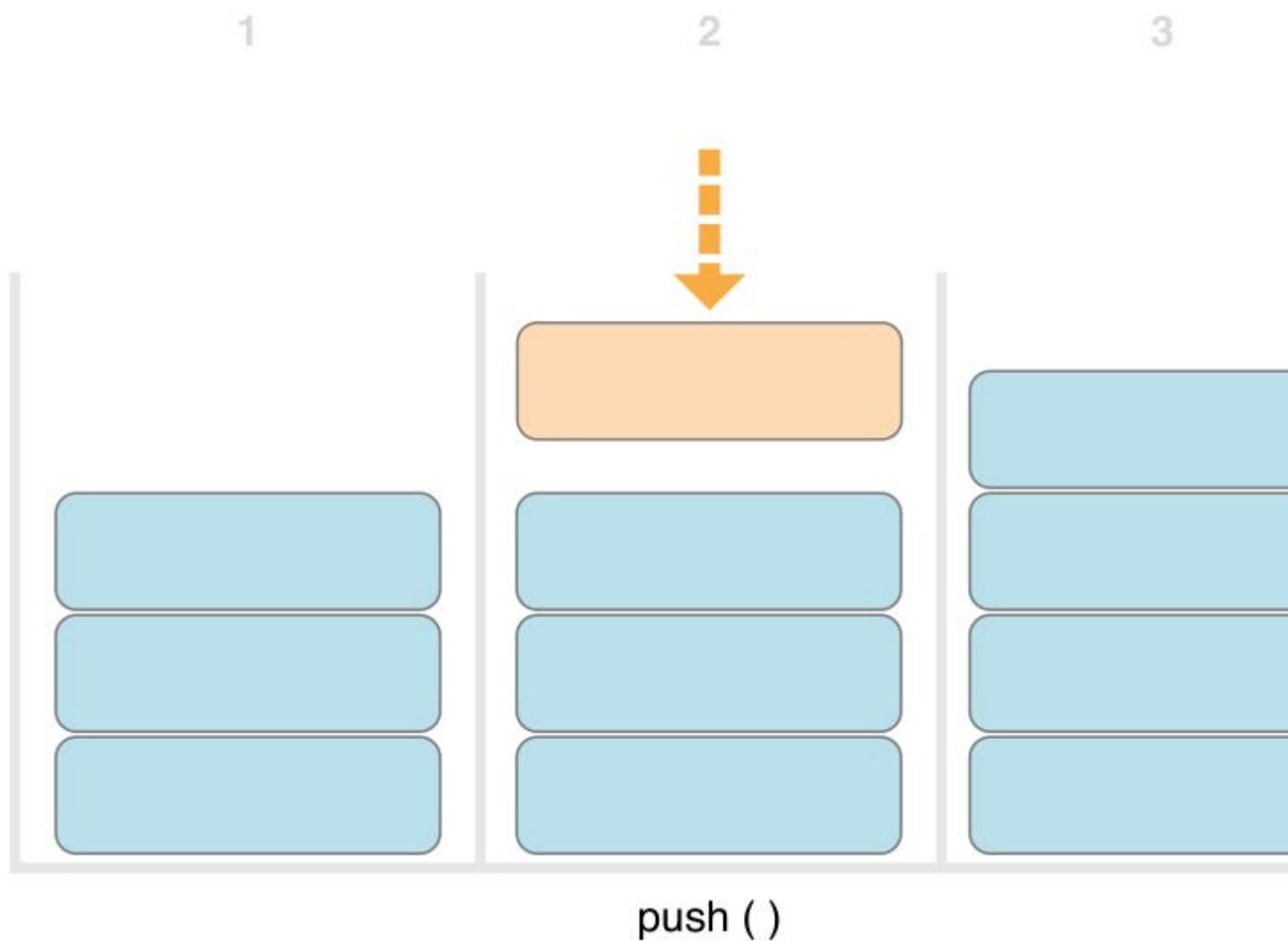
通常在泛型函数中，**Swift** 允许你定义你自己的泛型类型。这些自定义类、结构体和枚举作用于任何类型，如同 **Array** 和 **Dictionary** 的用法。

这部分向你展示如何写一个泛型集类型-**Stack**(栈)。一个栈是一系列值域的集合，和 **array**(数组)相似，但其是一个比 **Swift** 的 **Array** 类型更多限制的集合。一个数组可以允许其里面任何位置的插入/删除操作，而栈，只允许，只允许在集合的末端添加新的项（如同 **push** 一个新值进栈）。同样的一个栈也只能从末端移除项（如同 **pop** 一个值出栈）。

#### NOTE

The concept of a stack is used by the **UINavigationController** class to model the view controllers in its navigation hierarchy. You call the **UINavigationController** class `pushViewController:animated:` method to add (or push) a view controller on to the navigation stack, and its `popViewControllerAnimated:` method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict “last in, first out” approach to managing a collection.

下图展示了一个栈的压栈(push)/出栈(pop)的行为:



- 1 现在有三个值在栈中；
- 2 第四个值“pushed”到栈的顶部；
- 3 现在四个值在栈中，最近的那个在顶部；
- 4 栈中最顶部的那个项被移除，或称之为“popped”；
- 5 移除掉一个值后，现在栈又重新只有三个值。

这里展示了如何写一个非泛型版本的栈，Int 值型的栈：

```
struct IntStack {
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

这个结构体在栈中使用一个 **Array** 性质的 **items** 存储值。**Stack** 提供两个方法：**push** 和 **pop**，从栈中压进一个值和移除一个值。这些方法标记为可变的，因为他们需要修改（或转换）结构体的 **items** 数组。

上面所展现的 **IntStack** 类型只能用于 **Int** 值，不过，其对于定义一个泛型 **Stack** 类（可以处理任何类型值的栈）是非常有用的。

这里是一个相同代码的泛型版本：

```
struct Stack<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

注意到 **Stack** 的泛型版本基本上和非泛型版本相同，但是泛型版本的节点类型参数为 **T** 代替了实际 **Int** 类型。这种类型参数包含在一对尖括号里(<T>)，紧随在结构体名字后面。

**T** 定义了一个名为“某种类型 **T**”的节点提供给后来用。这种将来类型可以在结构体的定义里任何地方表示为“**T**”。

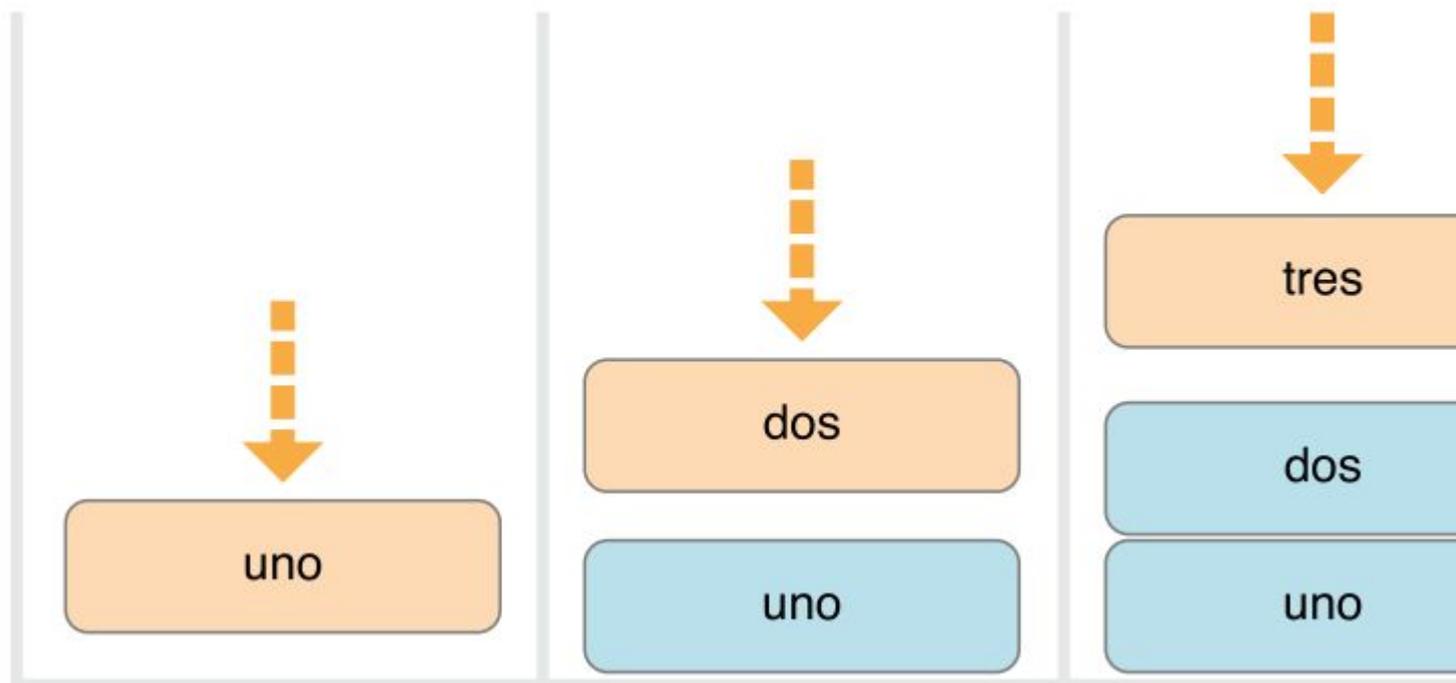
在这种情况下，**T** 在如下三个地方被用作节点：

- 创建一个名为 **items** 的属性，使用空的 **T** 类型值数组对其进行初始化；
- 指定一个包含一个参数名为 **item** 的 **push** 方法，该参数必须是 **T** 类型；
- 指定一个 **pop** 方法的返回值，该返回值将是一个 **T** 类型值。

当创建一个新单例并初始化时，通过用一对紧随在类型名后的尖括号里写出实际指定栈用到类型，创建一个 **Stack** 实例，同创建 **Array** 和 **Dictionary** 一样：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// 现在栈已经有4个 string 了
```

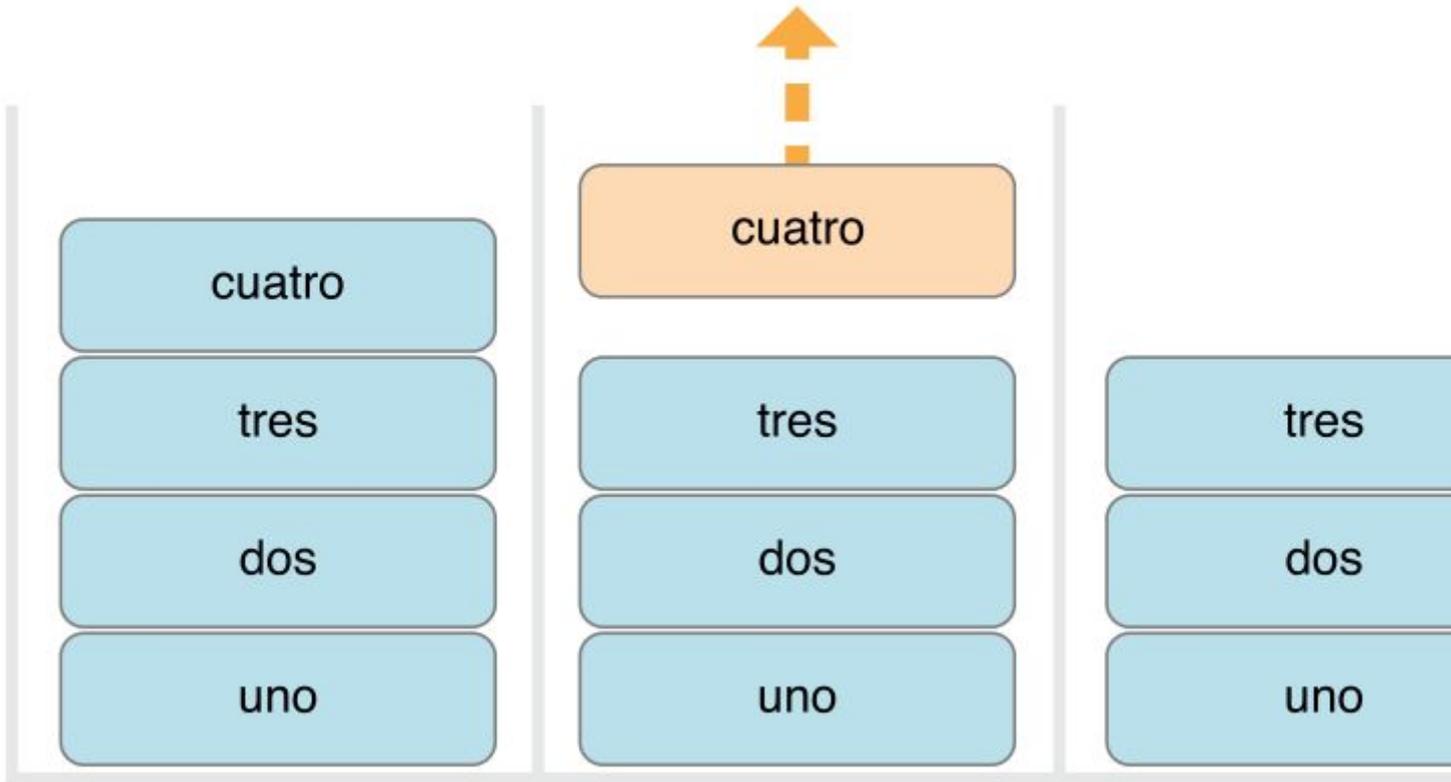
下图将展示 `stackOfStrings` 如何 `push` 这四个值进栈的过程：



从栈中 `pop` 并移除值 "cuatro"：

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

下图展示了如何从栈中 `pop` 一个值的过程：



由于 `Stack` 是泛型类型，所以在 `Swift` 中其可以用来创建任何有效类型的栈，这种方式如同 `Array` 和 `Dictionary`。

## 类型约束

`swapTwoValues` 函数和 `Stack` 类型可以作用于任何类型，不过，有的时候对使用在泛型函数和泛型类型上的类型强制约束为某种特定类型是非常有用的。类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

例如，`Swift` 的 `Dictionary` 类型对作用于其 `keys` 的类型做了些限制。在 `Dictionaries` 的描述中，字典的 `keys` 类型必须是 `hashable`，也就是说，必须有一种方法可以使其是唯一的表示。`Dictionary` 之所以需要其 `keys` 是 `hashable` 是为了以便于其检查其是否包含某个特定 `key` 的值。如无此需求，`Dictionary` 即不会告诉是否插入或者替换了某个特定 `key` 的值，也不能查找到已经存储在字典里面的给定 `key` 值。

这个需求强制加上一个类型约束作用于 `Dictionary` 的 `key` 上，当然其 `key` 类型必须遵循 `Hashable` 协议（`Swift` 标准库中定义的一个特定协议）。所有的 `Swift` 基本类型（如 `String`，`Int`，`Double` 和 `Bool`）默认都是 `hashable`。

当你创建自定义泛型类型时，你可以定义你自己的类型约束，当然，这些约束要支持泛型编程的强力特征中的多数。抽象概念如 `Hashtable` 具有的类型特征是根据他们概念特征来界定的，而不是他们的直接类型特征。

## 类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {  
    // function body goes here  
}
```

上面这个假定函数有两个类型参数。第一个类型参数 **T**，有一个需要 **T** 必须是 **SomeClass** 子类的类型约束；第二个类型参数 **U**，有一个需要 **U** 必须遵循 **SomeProtocol** 协议的类型约束。

## 类型约束行为

这里有个名为 **findStringIndex** 的非泛型函数，该函数功能是去查找包含一给定 **String** 值的数组。若查找到匹配的字符串，**findStringIndex** 函数返回该字符串在数组中的索引值（**Int**），反之则返回 **nil**：

```
func findStringIndex(array: String[], valueToFind: String) -> Int? {  
    for (index, value) in enumerate(array) {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

**findStringIndex** 函数可以作用于查找一字符串数组中的某个字符串：

```
letstrings = ["cat", "dog", "llama", "parakeet", "terrapi"]  
ifletfoundIndex = findStringIndex(strings, "llama") {  
    println("The index of llama is \$(foundIndex)")  
}  
// prints "The index of llama is 2"
```

如果只是针对字符串而言查找在数组中的某个值的索引，用处不是很大，不过，你可以写出相同功能的泛型函数 **findIndex**，用某个类型 **T** 值替换掉提到的字符串。

这里展示如何写一个你或许期望的 **findStringIndex** 的泛型版本 **findIndex**。请注意这个函数仍然返回 **Int**，是不是有点迷惑呢，而不是泛型类型？那是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数不会编译，原因在例子后面会说明：

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {  
    for (index, value) in enumerate(array) {
```

```
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所写的函数不会编译。这个问题的位置在等式的检查上，“if value == valueToFind”。不是所有的 Swift 中的类型都可以用等式符(==)进行比较。例如，如果你创建一个你自己的类或结构体来表示一个复杂的数据模型，那么 Swift 没法猜到对于这个类或结构体而言“等于”的意思。正因如此，这部分代码不能可能保证工作于每个可能的类型 T，当你试图编译这部分代码时估计会出现相应的错误。

不过，所有的这些并不会让我们无从下手。Swift 标准库中定义了一个 **Equatable** 协议，该协议要求任何遵循的类型实现等式符(==)和不等符(!=)对任何两个该类型进行比较。所有的 Swift 标准类型自动支持 **Equatable** 协议。

任何 **Equatable** 类型都可以安全的使用在 **findIndex** 函数中，因为其保证支持等式操作。为了说明这个事实，当你定义一个函数时，你可以写一个 **Equatable** 类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

**findIndex** 中这个单个类型参数写做：**T: Equatable**，也就意味着“任何 T 类型都遵循 **Equatable** 协议”。

**findIndex** 函数现在则可以成功的编译过，并且作用于任何遵循 **Equatable** 的类型，如 **Double** 或 **String**：

```
letdoubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
// doubleIndex is an optional Int with no value, because 9.3 is not in the array
letstringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
// stringIndex is an optional Int containing a value of 2
```

## 关联类型

当定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型给定作用于协议部分的类型一个节点名（或别名）。作用于关联类型上实际类型是不需要指定的，直到该协议接受。关联类型被指定为 **typealias** 关键字。

# 关联类型行为

这里是一个 **Container** 协议的例子，定义了一个 **ItemType** 关联类型：

```
protocol Container {  
    typealias ItemType  
    mutating func append(item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}
```

**Container** 协议定义了三个任何容器必须支持的兼容要求：

- 必须可能通过 **append** 方法添加一个新 **item** 到容器里；
- 必须可能通过使用 **count** 属性获取容器里 **items** 的数量，并返回一个 **Int** 值；
- 必须可能通过容器的 **Int** 索引值下标可以检索到每一个 **item**。

这个协议没有指定容器里 **item** 是如何存储的或何种类型是允许的。这个协议只指定三个任何遵循 **Container** 类型所必须支持的功能点。一个遵循的类型也可以提供其他额外的功能，只要满足这三个条件。

任何遵循 **Container** 协议的类型必须指定存储在其里面的值类型，必须保证只有正确类型的 **items** 可以加进容器里，必须明确可以通过其下标返回 **item** 类型。

为了定义这三个条件，**Container** 协议需要一个方法指定容器里的元素将会保留，而不需要知道特定容器的类型。

**Container** 协议需要指定任何通过 **append** 方法添加到容器里的值和容器里元素是相同类型，并且通过容器下标返回的容器元素类型的值的类型是相同类型。

为了达到此目的，**Container** 协议声明了一个 **ItemType** 的关联类型，写作 `typealias ItemType`。The protocol does not define what **ItemType** is an alias for—that information is left for any conforming type to provide（这个协议不会定义 **ItemType** 是遵循类型所提供的何种信息的别名）。尽管如此，**ItemType** 别名支持一种方法识别在一个容器里的 **items** 类型，以及定义一种使用在 **append** 方法和下标中的类型，以便保证任何期望的 **Container** 的行为是强制性的。

这里是一个早前 **IntStack** 类型的非泛型版本，适用于遵循 **Container** 协议：

```
struct IntStack: Container {
```

```
// original IntStack implementation
var items = Int[]()
mutating func push(item: Int) {
    items.append(item)
}
mutating func pop() -> Int {
    return items.removeLast()
}
// conformance to the Container protocol
typealias ItemType = Int
mutating func append(item: Int) {
    self.push(item)
}
var count: Int {
    return items.count
}
subscript(i: Int) -> Int {
    return items[i]
}
}
```

**IntStack** 类型实现了 **Container** 协议的所有三个要求，在 **IntStack** 类型的每个包含部分的功能都满足这些要求。

此外，**IntStack** 指定了 **Container** 的实现，适用的 **ItemType** 被用作 **Int** 类型。对于这个 **Container** 协议实现而言，定义 `typealias ItemType = Int`，将抽象的 **ItemType** 类型转换为具体的 **Int** 类型。

感谢 **Swift** 类型参考，你不用在 **IntStack** 定义部分声明一个具体的 **Int** 的 **ItemType**。由于 **IntStack** 遵循 **Container** 协议的所有要求，只要通过简单的查找 **append** 方法的 **item** 参数类型和下标返回的类型，**Swift** 就可以推断出合适的 **ItemType** 来使用。确实，如果上面的代码中你删除了 `typealias ItemType = Int` 这一行，一切仍旧可以工作，因为它清楚的知道 **ItemType** 使用的是何种类型。

你也可以生成遵循 **Container** 协议的泛型 **Stack** 类型：

```
struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // conformance to the Container protocol
```

```
mutating func append(item: T) {
    self.push(item)
}
var count: Int {
    return items.count
}
subscript(i: Int) -> T {
    return items[i]
}
}
```

这个时候，节点类型参数 **T** 被用作 **append** 方法的 **item** 参数和下标的返回类型。**Swift** 因此可以推断出被用作这个特定容器的 **ItemType** 的 **T** 的合适类型。

## 扩展一个存在的类型为一指定关联类型

在 **Adding Protocol Conformance with an Extension** 中有描述扩展一个存在的类型添加遵循一个协议。这个类型包含一个关联类型的协议。

**Swift** 的 **Array** 已经提供 **append** 方法，一个 **count** 属性和通过下标来查找一个自己的元素。这三个功能都达到 **Container** 协议的要求。也就意味着你可以扩展 **Array** 去遵循 **Container** 协议，只要通过简单声明 **Array** 适用于该协议而已。如何实践这样一个空扩展，在 **Declaring Protocol Adoption with an Extension** 中有描述这样一个实现一个空扩展的行为：

```
extension Array: Container {}
```

如同上面的泛型 **Stack** 类型一样，**Array** 的 **append** 方法和下标保证 **Swift** 可以推断出 **ItemType** 所使用的适用的类型。

定义了这个扩展后，你可以将任何 **Array** 当作 **Container** 来使用。

## Where 语句

**Type Constraints** 中描述的类型约束确保你定义关于类型参数的需求和一泛型函数或类型有关联。

对于关联类型的定义需求也是非常有用的。你可以通过这样去定义 **where** 语句作为一个类型参数队列的一部分。一个 **where** 语句使你能够要求一个关联类型遵循一个特定的协议，以及（或）那个特定的类型参数和关联类型可以是相同的。

你可写一个 **where** 语句，通过紧随放置 **where** 关键字在类型参数队列后面，其后跟着一个或者多个针对关联类型的约束，以及（或）一个或多个类型和关联类型的等于关系。

下面的例子定义了一个名为 `allItemsMatch` 的泛型函数，用来检查是否两个 `Container` 单例包含具有相同顺序的相同 `items`。如果匹配到所有的 `items`，那么返回一个为 `true` 的 `Boolean` 值，反之，则相反。

这两个容器可以被检查出是否是相同类型的容器（虽然它们可以是），但他们确实拥有相同类型的 `items`。这个需求通过一个类型约束和 `where` 语句结合来表示：

```
func allItemsMatch<
  C1: Container, C2: Container
  where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
  (someContainer: C1, anotherContainer: C2) -> Bool {

  // check that both containers contain the same number of items
  if someContainer.count != anotherContainer.count {
    return false
  }

  // check each pair of items to see if they are equivalent
  for i in 0..someContainer.count {
    if someContainer[i] != anotherContainer[i] {
      return false
    }
  }

  // all items match, so return true
  return true
}
```

这个函数用了两个参数：`someContainer` 和 `anotherContainer`。`someContainer` 参数是类型 `C1`，`anotherContainer` 参数是类型 `C2`。`C1`和 `C2`是容器的两个节点类型参数，决定了这个函数何时被调用。

这个函数的类型参数列紧随在两个类型参数需求的后面：

- `C1`必须遵循 `Container` 协议（写作 `C1: Container`）。
- `C2`必须遵循 `Container` 协议（写作 `C2: Container`）。
- `C1`的 `ItemType` 同样是 `C2`的 `ItemType`（写作 `C1.ItemType == C2.ItemType`）。
- `C1`的 `ItemType` 必须遵循 `Equatable` 协议（写作 `C1.ItemType: Equatable`）。

第三个和第四个要求被定义为一个 **where** 语句的一部分，写在关键字 **where** 后面，作为函数类型参数链的一部分。

这些要求意思是：

- **someContainer** 是一个 **C1**类型的容器。
- **anotherContainer** 是一个 **C2**类型的容器。
- **someContainer** 和 **anotherContainer** 包含相同的 **items** 类型。
- **someContainer** 中的 **items** 可以通过不等于操作(**!=**)来检查它们是否彼此不同。

第三个和第四个要求结合起来的意思是 **anotherContainer** 中的 **items** 也可以通过 **!=** 操作来检查，因为他们在 **someContainer** 中 **items** 确实是相同的类型。

这些要求能够使 **allItemsMatch** 函数比较两个容器，即便他们是不同的容器类型。

**allItemsMatch** 首先检查两个容器是否拥有同样数目的 **items**，如果他们的 **items** 数目不同，没有办法进行匹配，函数就会 **false**。

检查完之后，函数通过 **for-in** 循环和半闭区间操作(**..**)来迭代 **someContainer** 中的所有 **items**。对于每个 **item**，函数检查是否 **someContainer** 中的 **item** 不等于对应的 **anotherContainer** 中的 **item**，如果这两个 **items** 不等，则这两个容器不匹配，返回 **false**。

如果循环体结束后未发现没有任何的不匹配，那表明两个容器匹配，函数返回 **true**。

这里演示了 **allItemsMatch** 函数运算的过程：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    println("All items match.")
}else{
```

```
println("Not all items match.")  
}  
// prints "All items match."
```

上面的例子创建一个 **Stack** 单例来存储 **String**，然后压了三个字符串进栈。这个例子也创建了一个 **Array** 单例，并初始化包含三个同栈里一样的原始字符串。即便栈和数组否是不同的类型，但他们都遵循 **Container** 协议，而且他们都包含同样的类型值。你因此可以调用 **allItemsMatch** 函数，用这两个容器作为它的参数。在上面的例子中，**allItemsMatch** 函数正确的显示了所有的这两个容器的 **items** 匹配。