一、Swift预热

1. **If**

**var** optionalName: String? = "John Appleseed"

**if** let name = optionalName**{**

**}**如果可选的optionalName不为空，那么将其赋值给name，然后在接下来的语句中可以使用name

1. **switch case**

switch 支持任意类型的数据以及各种比较操作——不仅仅是整数以及测试相等。每个case执行完了之后不会执行下一个case，如果需要继续执行下一个case可以使用fallthrough

**case "celery":**

**case "cucumber", "watercress":**

**case let x where x.hasSuffix("pepper"):**

**default://必须要有default语句，并且每一个case都要有执行语句**

另外可以使用switch查找键值对：

**let interestingNumbers = [**

**"Prime": [2, 3, 5, 7, 11, 13],**

**"Fibonacci": [1, 1, 2, 3, 5, 8],**

**"Square": [1, 4, 9, 16, 25],**

**]**

**var largest = 0**

**for (kind, numbers) in interestingNumbers {**

**for number in numbers {**

**if number > largest {**

**largest = number**

**}**

**}**

**}**

1. **for**

**for i in 0..3 {}//不包含上界**

**for i in 0...3 {}//包含上界**

1. **函数**

使用 func 来声明一个函数，使用名字和参数来调用函数。使用->来指定函数返回值。

**func greet(name: String, day: String) -> String {**

**return "Hello \(name), today is \(day)."**

**}**

使用一个元组来返回多个值。

**func getGasPrices() -> (Double, Double, Double) {**

**return (3.59, 3.69, 3.79)  
}**

函数的参数数量是可变的，用一个数组来获取它们：

**func sumOf(numbers: Int...) -> Int {**

**var sum = 0**

**for number in numbers {  
 sum += number  
}  
 return sum**

**}**

**sumOf()  
sumOf(42, 597, 12)**

函数可以嵌套。被嵌套的函数可以访问外侧函数的变量，你可以使用嵌套函数来重构一个太长或者太复杂的函数。

func returnFifteen() -> Int {

**var** y = 10

func add() {

y += 5

}

add()

**return** y

}  
returnFifteen()

函数可以作为另一个函数的返回值。

//此函数返回的是一个函数，这个返回函数的参数是int，返回值是int

func makeIncrementer() -> (Int -> Int) {

func addOne(number: Int) -> Int {

**return** 1 + number  
}

**return** addOne

}  
**var** increment = makeIncrementer()//返回一个函数int参数  
increment(7)//8

函数也可以当做参数传入另一个函数。

**func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {**

**for item in list {**

**if condition(item) {//此处实际是在调用lessTanTen函数**

**return true**

**}**

**}**

**return false**

**}**

**func lessThanTen(number: Int) -> Bool {**

**return number < 10**

**}**

**var numbers = [20, 19, 7, 12]**

**hasAnyMatches(numbers, lessThanTen)//传入函数**

1. **对象和类**

使用 class 和类名来创建一个类。类中属性的声明和常量、变量声明一样，唯一的区别就是它们的上下文是类。同样，方法和函数声明也一样。

**class Shape {**

**var numberOfSides = 0**

**func simpleDescription() -> String {**

**return "A shape with \(numberOfSides) sides."**

**}**

**}**

**var shape = Shape()**

**shape.numberOfSides = 7**

**var shapeDescription = shape.simpleDescription()**

这个版本的 Shape 类缺少了一些重要的东西：一个构造函数来初始化类实例。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**

**}**

**func simpleDescription() -> String {**

**return "A shape with \(numberOfSides) sides."**

**}**

**}**

如果你需要在删除对象之前进行一些清理工作，使用 deinit 创建一个析构函数。

1. **枚举**

使用 enum 来创建一个枚举。就像类和其他所有命名类型一样，枚举可以包含方法。枚举原始值的类型是 Int，所以你只需要设置第一个原始值。剩下的原始值会按照顺序赋值。你也可以使用字符串或者浮点数作为枚举的原始值。使用 toRaw 和 fromRaw 函数来在原始值和枚举值之间进行转换。

**if let convertedRank = Rank.fromRaw(3) {**

**let threeDescription = convertedRank.simpleDescription()**

**}**

1. **结构体**

使用 struct 来创建一个结构体。结构体和类有很多相同的地方，比如方法和构造器。它们结构体之间最大的一个区别就是 结构体是传值，类是传引用。

1. **接口**

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 {}**

**struct SimpleStructure: ExampleProtocol {**

**var simpleDescription: String = "A simple structure"**

**mutating func adjust() {**

**simpleDescription += " (adjusted)"**

**}**

**}**

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。

1. **扩展**

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 //49未测试**

1. **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[] {}**

也可以创建泛型类、枚举和结构体。

**enum OptionalValue<T> {}**

二、Swift基础

1. **整数**

可以访问不同整数类型的 min 和 max 属性来获取对应类型的最大值和最小值：

**let minValue = UInt8.min // minValue 为 0，是 UInt8 类型的最小值**

**let maxValue = UInt8.max // maxValue 为 255，是 UInt8 类型的最大值**

1. **浮点数**

Double 精确度很高，至少有 15 位数字，而 Float 最少只有 6 位数字。选择哪个类型取决于你的代码需要处理的值的范围。

1. **类型安全和类型推测**

如果你给一个新常量赋值 42 并且没有标明类型， Swift 可以推测出常量类型是Int

如果你没有给浮点字面量标明类型， Swift 会推测你想要的是 Double, 当推测浮点数的类型时， Swift 总是会选择 Double 而不是 Float。如果表达式中同时出现了整数和浮点数，会被推测为 Double 类型。

1. **数值型字面量**

**let decimalInteger = 17  
let binaryInteger = 0b10001 // 二进制的 17  
let octalInteger = 0o21 // 八进制的 17  
let hexadecimalInteger = 0x11 // 十六进制的 17**

在十进制浮点数中通过大写或者小写的 e 来指定，在十六进制浮点数中通过  
大写或者小写的 p 来指定。

**1.25e2 表示 $1.25 × 10^{2}$，等于 125.0**

**0xFp2 表示 $15 × 2^{2}$，等于 60.0**

**let decimalDouble = 12.1875 //整数  
let exponentDouble = 1.21875e1 //1.21875 \* 10  
let hexadecimalDouble = 0xC.3p0 // 12 + 3 \* (1 / 16) \* 1(2的0次方)**

数值类字面量可以包括额外的格式来增强可读性。整数和浮点数都可以添加额外的零并且包含下划线，并不会影响字面量：

**let paddedDouble = 000123.456  
let oneMillion = 1\_000\_000  
let justOverOneMillion = 1\_000\_000.000\_000\_1**

1. **类型别名**

类型别名(type aliases)就是给现有类型定义另一个名字。你可以使用 typealias 关键字来定义类型别名。

**typealias AudioSample = UInt16**

1. **元组**

元组（ tuples）把多个值组合成一个复合值。元组内的值可以使任意类型，并不要求是相同类型。

**let http404Error = (404, "Not Found")**

**let (statusCode, statusMessage) = http404Error**

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

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

如果你只需要一部分元组值，分解的时候可以把要忽略的部分用下划线（ \_）标记：

**let (justTheStatusCode, \_) = http404Error**

你还可以通过下标来访问元组中的单个元素，下标从零开始：

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

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

可以在定义元组的时候给单个元素命名：

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

给元组中的元素命名后，你可以通过名字来获取这些元素的值：

**println("The status code is \(http200Status.statusCode)")**

**println("The status message is \(http200Status.description)")**

1. **可选**

使用可选（ optionals）来处理值可能缺失的情况。

if 语句以及强制解析:

你可以使用 if 语句来判断一个可选是否包含值。如果可选有值，结果是 true；如果没有值，结果是 false。当你确定可选包确实含值之后，你可以在可选的名字后面加一个感叹号(!)来获取值。这个惊叹号表示“我知道这个可选有值，请使用它。”这被称为可选值的强制解析（forced unwrapping）:

**let possibleNumber = "123"**

**let convertedNumber = possibleNumber.toInt()**

**if convertedNumber {**

**println("\(possibleNumber) has an integer value of \(convertedNumber!)")**

**} else {**

**println("\(possibleNumber) could not be converted to an integer")**

**}**

可选绑定:

使用可选绑定（ optional binding）来判断可选是否包含值，如果包含就把值赋给一个临时  
常量或者变量。

**if let constantName = someOptional {**

**statements**

**}**

nil:

可以给可选变量赋值为 nil 来表示它没有值：

**var serverResponseCode: Int? = 404  
// serverResponseCode 包含一个可选的 I  
serverResponseCode = nil  
// serverResponseCode 现在不包含值**

注意： nil 不能用于非可选的常量和变量。如果你的代码中有常量或者变量需要处理值缺失的情况，请把它们声明成对应的可选类型。

如果你声明一个可选常量或者变量但是没有赋值，它们会自动被设置为 nil。

隐式解析可选：

有时候在程序架构中，第一次被赋值之后，可以确定一个可选总会有值。在这种情况下，每次都要判断和解析可选值是非常低效的，因为可以确定它总会有值。  
这种类型的可选被定义为隐式解析可选（ implicitly unwrapped optionals）。把想要用作可选的类型的后面的问号（ String?）改成感叹号（ String!）来声明一个隐式解析可选。

一个隐式解析可选其实就是一个普通的可选，但是可以被当做非可选来使用，并不需要每次都使用解析来获取可选值。下面的例子展示了可选 String 和隐式解析可选 String 之间的区别：

**let possibleString: String? = "An optional string."  
println(possibleString!) // 需要惊叹号来获取值  
// 输出 "An optional string."  
let assumedString: String! = "An implicitly unwrapp"  
println(assumedString) // 不需要感叹号  
// 输出 "An implicitly unwrapped optional string."**

1. **断言**

**let age = -3**

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

**// 因为 age < 0，所以断言会触发**

1. **求余运算**

在对负数 -b 求余时, -b 的符号会被忽略. 这意味着 a % b 和 a % -b 的结果是相同的.不同于 C 和 Objective-C, Swift 中是可以对浮点数进行求余的.

**8 % 2.5 // 等于 0.5**

1. **区间运算符**

闭区间运算符 a...b 定义一个包含从 a 到 b (包括 a 和 b)的所有值的区间. 闭区间运算符在迭代一个区间的所有值时是非常有用的, 如在 for-in 循环中:

**for index in 1...5 {**

**println("\(index) \* 5 = \(index \* 5)")  
}**

半闭区间：

半闭区间 a..b 定义一个从 a 到 b 但不包括 b 的区间. 之所以称为半闭区间, 是因为该区间包含第一个值而不包括最后的值。

1. **字符串和字符**

使用字符(Characters)：

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。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：

**let yenSign: 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。

比较字符串：

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返回 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。

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。

**let normal = "Could you help me, please?"  
let shouty = normal.uppercaseString  
// shouty 值为 "COULD YOU HELP ME, PLEASE?"  
let whispered = normal.lowercaseString  
// whispered 值为 "could you help me, please?"**

1. 集合类型
2. **数组**

数组使用有序列表存储相同类型的多重数据。相同的值可以多次出现在一个数组的不同位置中。

**var shoppingList: String[] = ["Eggs", "Milk"]**

访问和修改数组

我们可以通过数组的方法和属性来访问和修改数组，或者下标语法。 还可以使用数组的只读属性 count 来获取数组中的数据项数量。

**println("The shopping list contains \(shoppingList.count) items.")**

**// 打印出"The shopping list contains 2 items."（这个数组有 2 个项）**

使用布尔项 isEmpty 来作为检查 count 属性的值是否为 0 的捷径。

**if shoppingList.isEmpty {}**

也可以使用 append 方法在数组后面添加新的数据项：

**shoppingList.append("Flour")**

除此之外，使用加法赋值运算符（ +=）也可以直接在数组后面添加数据项：

**shoppingList += "Baking Powder"**

我们也可以使用加法赋值运算符（ +=）直接添加拥有相同类型数据的数组。

**shoppingList += ["Chocolate Spread", "Cheese", "Butter"]**

可以直接使用下标语法来获取数组中的数据项，把我们需要的数据项的索引值放在直接放在数组名称的方括号中：

**var firstItem = shoppingList[0]**

我们也可以用下标来改变某个已有索引值对应的数据值：

**shoppingList[0] = "Six eggs"**

还可以利用下标来一次改变一系列数据值，即使新数据和原有数据的数量是不一样的。

**shoppingList[4...6] = ["Bananas", "Apples"]**

注意： 我们不能使用下标语法在数组尾部添加新项。如果我们试着用这种方法对索引越界的数据进行检索或者设置新值的操作，我们会引发一个运行期错误。

数组插值、删值

调用数组的 insert(atIndex:)方法来在某个具体索引值之前添加数据项：

**shoppingList.insert("Maple Syrup", atIndex: 0)  
// shoppingList 现在有 7 项  
// "Maple Syrup" 现在是这个列表中的第一项**

类似的我们可以使用 removeAtIndex 方法来移除数组中的某一项。这个方法把数组在特定索引值中存储的数据项移除并且返回这个被移除的数据项（我们不需要的时候就可以无视它）;

**let mapleSyrup = shoppingList.removeAtIndex(0)**

**//索引值为 0 的数据项被移除**

**// shoppingList 现在只有 6 项，而且不包括 Maple Syrup**

**// mapleSyrup 常量的值等于被移除数据项的值 "Maple Syrup"**

数据项被移除后数组中的空出项会被自动填补

如果我们只想把数组中的最后一项移除，可以使用 removeLast 方法而不是 removeAtIndex方法来避免我们需要获取数组的 count 属性。就像后者一样，前者也会返回被移除的数据项：

**let apples = shoppingList.removeLast()  
// 数组的最后一项被移除了  
// shoppingList 现在只有 5 项，不包括 cheese  
// apples 常量的值现在等于"Apples" 字符串**

数组的遍历

可以使用 for-in 循环来遍历所有数组中的数据项：

**for item in shoppingList {**

**println(item)**

**}**

如果我们同时需要每个数据项的值和索引值，可以使用全局 enumerate 函数来进行数组遍历。 enumerate 返回一个由每一个数据项索引值和数据值组成的键值对组。我们可以把这个键值对组分解成临时常量或者变量来进行遍历：

**for (index, value) in enumerate(**shoppingList**) {**

**println("Item \(index + 1): \(value)")**

**}**

创建并且构造一个数组

可以使用构造语法来创建一个由特定数据类型构成的空数组：

**var someInts = Int[]()  
println("someInts is of type Int[] with \(someInts.count) items。 ")  
// 打印 "someInts is of type Int[] with 0 items。 "（ someInts 是 0 数据项的Int[]数组）**

可以使用空数组语句创建一个空数组，它的写法很简单： []（一对空方括号）：

**someInts.append(3)**

**someInts = []**

Swift 中的 Array 类型还提供一个可以创建特定大小并且所有数据都被默认的构造方法。我们可以把准备加入新数组的数据项数量（ count）和适当类型的初始值（ repeatedValue）传入数组构造函数：

**var threeDoubles = Double[](count: 3, repeatedValue:0.0)**

因为类型推断的存在，我们使用这种构造方法的时候不需要特别指定数组中存储的数据类型，因为类型可以从默认值推断出来：

**var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)**

最后，我们可以使用加法操作符（ +）来组合两种已存在的相同类型数组。新数组的数据类型会被从两个数组的数据类型中推断出来：

**var sixDoubles = threeDoubles + anotherThreeDoubles**

1. **字典**

字典是一种存储相同类型多重数据的存储器。每个值（ value）都关联独特的键（ key），键作为字典中的这个值数据的标识符。

Swift 的字典使用时需要具体规定可 以存储 键和值 类型。

Swift 的字典使用 Dictionary<KeyType, ValueType>定义,其中 KeyType 是字典中键的数据类型， ValueType 是字典中对应于这些键所存储值的数据类型。

在这个字典中键是三个字母的国际航空运输相关代码，值是机场名称：

**var airports: Dictionary<String, String> = ["TYO": "Tokyo", "DUB": "Dublin"]**

和数组一样，如果我们使用字面语句构造字典就不用把类型定义清楚。 airports 的也可以用这种方法简短定义：

**var airports = ["TYO": "Tokyo", "DUB": "Dublin"]**

读取和修改字典

可以通过字典的只读属性 count 来获取某个字典的数据项数量：

**println("The dictionary of airports contains \(airports.count) items.")**

我们也可以在字典中使用下标语法来添加新的数据项。可以使用一个合适类型的 key 作为  
下标索引，并且分配新的合适类型的值：

**airports["LHR"] = "London"**

我们也可以使用下标语法来改变特定键对应的值：

**airports["LHR"] = "London Heathrow"**

作为另一种下标方法，字典的updateValue(forKey:)方法可以设置或者更新特定键对应的值。updateValue(forKey:)函数会返回包含一个字典值类型的可选值。举例来说：对于存储 String值的字典，这个函数会返回一个 String?或者“可选 String”类型的值。如果值存在，则这个可选值等于被替换的值，否则将会是 nil。

**if let oldValue = airports.updateValue("Dublin Internation",**

**forKey: "DUB") {**

**println("The old value for DUB was \(oldValue).")  
}  
// 打印出 "The old value for DUB was Dublin."（ dub 原值是 dublin）**

我们也可以使用下标语法来在字典中检索特定键对应的值。由于使用一个没有值的键这种情况是有可能发生的，可选 类型返回这个键存在的相关值，否则就返回 nil：

**if let airportName = airports["DUB"] {**

**println("The name of the airport is \(airportName).")**

**} else {**

**println("That airport is not in the airports dictionary.")**

**}**

我们还可以使用下标语法来通过给某个键的对应值赋值为 nil 来从字典里移除一个键值对：

**airports["APL"] = "Apple Internation"**

**airports["APL"] = nil // APL 现在被移除了**

另外， removeValueForKey 方法也可以用来在字典中移除键值对。这个方法在键值对存在的情况下会移除该键值对并且返回被移除的 value 或者在没有值的情况下返回 nil：

**if let removedValue = airports.removeValueForKey("DUB") {**

**println("The removed airport's name is \(removedValue).")**

**} else {**

**println("The airports dictionary does not contain a value for DUB.")**

**}**

**// 打印 "The removed airport's name is Dublin International."（被移除的机场名字是都柏林国际）**

字典遍历

我们可以使用 for-in 循环来遍历某个字典中的键值对。每一个字典中的数据项都由(key,value)元组形式返回，并且我们可以使用暂时性常量或者变量来分解这些元组：

**for (airportCode, airportName) in airports {**

**println("\(airportCode): \(airportName)")**

**}**

**// TYO: Tokyo**

**// LHR: London Heathrow**

也可以通过访问他的 keys 或者 values 属性（都是可遍历集合）检索一个字典的键或者值：

**for airportCode in airports.keys {**

**println("Airport code: \(airportCode)")**

**}**

**for airportName in airports.values {**

**println("Airport name: \(airportName)")**

**}**

如果只是需要使用某个字典的键集合或者值集合来作为某个接受 Array 实例 API 的参数，可以直接使用 keys 或者 values 属性直接构造一个新数组：

**let airportCodes = Array(airports.keys)**

**// airportCodes is ["TYO", "LHR"]**

**let airportNames = Array(airports.values)**

**// airportNames is ["Tokyo", "London Heathrow"]**

创建一个空字典

**var namesOfIntegers = Dictionary<Int, String>()**

**namesOfIntegers[16] = "sixteen"**

**namesOfIntegers = [:]**

**// namesOfIntegers 又成为了一个 Int, String 类型的空字典**

1. **控制流**

switch case

case 块的模式可以使用 where 语句来判断额外的条件。当且仅当 where  
语句的条件为真时，匹配到的 case 块才会被执行。

**let anotherPoint = (2, 0)**

**switch anotherPoint {**

**case (let x, 0)://元组作为case**

**println("on the x-axis with an x value of \(x)")**

**case let (x, y) where x == y: //where返回boolean**

**println("(\(x), \(y)) is on the line x == y")**

**default:**

**println("(\(somePoint.0), \(somePoint.1)) is outside of the box")**

**}**

**//数值区间作为case**

**let count = 3\_000\_000\_000\_000**

**switch count {**

**case 1...3:**

**var naturalCount = "a few"**

**default:**

**naturalCount = "millions and millions of"**

**}**

**//字符系列作为case**

**let someCharacter: Character = "e"**

**switch someCharacter {**

**case "a", "e", "i", "o", "u":**

**println("\(someCharacter) is a vowel")**

**default:**

**println("\(someCharacter) is not a vowel or a consonant")**

**}**

NOTE： fallthrough 关键字不会检查它下一个将会落入执行的 case 中的匹配 条件。fallthrough 简单地使代码执行继续连接到下一个 case 中的执行代码，这和 C 语言标准中的switch 语句特性是一样的。

Label标签

一般格式是：

**label name: while condition {**

**statements  
}**

**let flags = 5**

**label test: while flags > 0{**

**switch flags{**

**case 0…2:**

**break test //停止标签所在的循环**

**default:**

**continue test**

**}  
}**

1. **函数**

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，来返回一个由多个值组成的复合返回值。

**func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {**

**return (23, 12, 23)**

**}**

默认形参值

你可以为任何形参定义默认值以作为函数定义的一部分。如果已经定义了默认值，那么调用函数时就可以省略该行参。

**func join(string s1: String, toString s2: String,withJoiner joiner: String = " ") -> String {**

**return s1 + joiner + s2**

**}**

调用该函数的时候，如果给jniner传递参数，那么joiner赋值为传递的参数，否则等于默认的参数值

常量形参和变量形参

函数的形参默认是常量。试图在函数体内改变函数形参的值会引发一个编译时错误。这意味着你不能错误地改变形参的值。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 {}**

In-Out 形参

变量形参只能在函数本身内改变。如果你想让函数改变形参值，并想要在函数  
调用结束后保持形参值的改变，那你可以把形参定义为 in-out 形参。

只能传递一个变量作为 in-out 形参对应的实参。你不能传递一个常量或者字面量作为实参，因为常量和字面量不能被修改。

把变量作为实参传递给 in out 形参时，需要在直接在变量前添加 & 符号，以表明它可以被函数修改。

**func swapTwoInts(inout a: Int, inout b: Int) {**

**let temporaryA = a**

**a = b**

**b = temporary**

**}**

**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"**

使用函数类型

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，并为变量指定一个对应的函数：

**func addTwoInts(a: Int, b: Int) -> Int {**

**return a + b**

**}**

**func multiplyTwoInts(a: Int, b: Int) -> Int {**

**return a \* b**

**}**

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

**println("Result: \(mathFunction(2, 3))")**

**// prints "Result: 5"**

具有相同匹配类型的不同函数可以被赋给同一个变量，和非函数类型一样：

**mathFunction = multiplyTwoInts**

**println("Result: \(mathFunction(2, 3))")**

**// prints "Result: 6"**

与其他类型一样,当你给函数赋一个常量或者变量时，你可以让 Swift 去推断函数的类型。

**let anotherMathFunction = addTwoInts**

**// anotherMathFunction is inferred to be of type (Int, Int) -> Int**

作为形参类型的函数类型

可以使用一个函数类型，如(Int, Int)->Int 作为另一个函数的形参类型。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。

**func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int){**

**println("Result: \(mathFunction(a, b))")**

**}**

**printMathResult(addTwoInts, 3, 5)**

**// prints "Result: 8"**

作为返回类型的函数类型

可以将一个函数类型作为另一个函数的返回类型。你可以在返回函数的返回箭头(->) 后立即编写一个完整的函数类型来实现。

**func stepForward(input: Int) -> Int {**

**return input + 1**

**}**

**func stepBackward(input: Int) -> Int {**

**return input – 1**

**}**

chooseStepFunction 函数，它的返回类型是"函数类型(Int) -> Int"。  
chooseStepFunction 基于名为 backwards 的布尔形参返回 stepBackward 或 stepForward  
函数:

**func chooseStepFunction(backwards: Bool) -> (Int) -> Int {**

**return backwards ? stepBackward : stepForward**

**}**

可以使用 chooseStepFunction 获取一个函数,可能是递增函数或递减函数:

**var currentValue = 3**

**let moveNearerToZero = chooseStepFunction(currentValue > 0)**

**// moveNearerToZero now refers to the stepBackward() function**

嵌套函数

在全局作用域中定义。其实你还可以在其他函数体中定义函数，被称为嵌套函数。

嵌套函数默认对外界是隐藏的，但仍然可以通过它们包裹的函数调用和使用它。

**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**

**}**

**var currentValue = -4**

**let moveNearerToZero = 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!**

1. **闭包**

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。

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：  
1. 全局函数是一个有名字但不会捕获任何值的闭包  
2. 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  
3. 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

闭包表达式

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，但不提供默认值。 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。

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

**func backwards(s1: String, s2: String) -> Bool {**

**return s1 > s2**

**}**

**var reversed = sort(names, backwards)**

**// reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]**

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**

**})**

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 } )**

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 } )**

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 } )**

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。

参数名简写

如果您在闭包表达式中使用参数名称简写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称简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 } )**

运算符函数

Swift 的 String 类型定义了关于大于号 (>) 的字符串实现，让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。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, >)**

Trailing 闭包

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。

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 }**

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"]:

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

**let numbers = [16, 58, 510]**

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。 需要注意的时调用 numbers.map 不需要在 map 后面包含任何括号，因为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：

**let strings = 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。

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 (Caputure)

Swift 最简单的闭包形式是嵌套函数，也就是定义在其他函数体内的函数。 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。

**func makeIncrementor(forIncrement amount: Int) -> () -> Int {**

**var runningTotal = 0**

**func incrementor() -> Int {**

**runningTotal += amount**

**return runningTotal**

**}**

**return incrementor**

**}**

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。

**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。

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。

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

此时incrementByTen和incrementBySeven分别捕获着不同的引用

闭包是引用类型

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。

1. **枚举**

枚举语法（ Enumeration Syntax）

使用 enum 关键词并且把它们的整个定义放在一对大括号内：

**enum SomeEumeration {**

**// enumeration definition goes here**

**}**

下面是一个例子：

**enum CompassPoint {**

**case North**

**case South**

**case East**

**case West**

**}**

Swift 的枚举成员在被创建时不会被赋予一个默认的  
整数值。在上面的 CompassPoints 例子中， North， South， East 和 West 不是隐式得等于  
0， 1， 2 和 3。相反的，这些不同的枚举成员在 CompassPoint 的一种显示定义中拥有各自  
不同的值。

多个成员值可以出现在同一行上，用逗号隔开：

**enum Planet {**

**case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Nepturn**

**}**

**var directionToHead = CompassPoint.West**

directionToHead 的类型被推断当它被 CompassPoint 的一个可能值初始化。一 旦directionToHead 被声明为一个 CompassPoint，你可以使用更短的点（ .）语法将其设置为另一个 CompassPoint 的值：

**directionToHead = .East**

匹配枚举值和 Switch 语句

可以匹配单个枚举值和 switch 语句：

**directionToHead = .South**

**switch directionToHead {**

**case .North:**

**println("Lots of planets have a north")**

**case .South:**

**println("Watch out for penguins")**

**case .East:**

**println("Where the sun rises")**

**case .West:**

**println("Where the skies are blue")**

**default:**

**println(“no match case”);**

**}**

**// prints "Watch out for penguins”**

关联值（ Associated Values）

可以定义 Swift 的枚举存储任何类型的关联值，如果需要的话，每个成员的数据类型可以是各不相同的。

在 Swift 中，用来定义两种商品条码的枚举是这样子的：

**enum Barcode {**

**case UPCA(Int, Int, Int)**

**case QRCode(String)**

**}**

可以使用任何一种条码类型创建新的条码，如：

**var productBarcode = Barcode.UPCA(8, 85909\_51226, 3)**

同一个商品可以被分配给一个不同类型的条形码，如：

**productBarcode = .QRCode("ABCDEFGHIJKLMNOP")**

不同的条形码类型可以使用一个 switch 语句来检查，然而这次关联值可以被  
提取作为 switch 语句的一部分。你可以在 switch 的 case 分支代码中提取每个关联值作为一个常量（用 let 前缀）或者作为一个变量（用 var 前缀）来使用：

**switch productBarcode {**

**case .UPCA(let numberSystem, let identifier, let check):**

**println("\(numberSystem),\(identifier),\(check).")**

**case .QRCode(let productCode):**

**println("QR code with value of \(productCode).")**

**}**

**// prints "QR code with value of ABCDEFGHIJKLMNOP.”**

如果一个枚举成员的所有关联值被提取为常量，或者它们全部被提取为变量，为了简洁，你可以只放置一个 var 或者 let 标注在成员名称前：

**switch productBarcode {**

**case let .UPCA(numberSystem, identifier, check):**

**println("\(numberSystem),\(identifier),\(check).")**

**case let .QRCode(productCode):**

**println("QR code with value of \(productCode).")**

**}**

**// prints "QR code with value of ABCDEFGHIJKLMNOP."**

原始值（ Raw Values）

**enum Planet: Int {**

**case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune**

**}**

自动递增意味着 Planet.Venus 的原始值是 2，依次类推。

使用枚举成员的 toRaw 方法可以访问该枚举成员的原始值：

**let earthsOrder = Planet.Earth.toRaw()**

**// earthsOrder is 3**

使用枚举的 fromRaw 方法来试图找到具有特定原始值的枚举成员。

**let possiblePlanet = Planet.fromRaw(7)**

**// possiblePlanet is of type Planet? and equals Planet.Uranus**

然而，并非所有可能的 Int 值都可以找到一个匹配的行星。正因为如此， fromRaw 方法可以返回一个可选的枚举成员。在上面的例子中， possiblePlanet 是 Planet?类型，或“可选的Planet”。

**let positionToFind = 9**

**if let somePlanet = Planet.fromRaw(positionToFind) {**

**}else{**

**}**

在这个范例中，无法检索到位置为 9 的行星，所以 else 分支被执行。

1. **类和结构体**

类和结构体对比

定义属性用于储存值  
定义方法用于提供功能  
定义下标用于通过下标语法访问值  
定义初始化器用于生成初始化值  
通过扩展以增加默认实现的功能  
符合协议以对某类提供标准功能

与结构体相比，类还有如下的附加功能：  
继承允许一个类继承另一个类的特征  
类型转换允许在运行时检查和解释一个类实例的类型  
取消初始化器允许一个类实例释放任何其所被分配的资源  
引用计数允许对一个类的多次引用

定义

类和结构体有着类似的定义方式。我们通过关键字 class 和 struct 来分别表示类和结构体，并在一对大括号中定义它们的具体内容：

**class SomeClass {**

**// class definition goes here**

**}**

**struct SomeStructure {**

**// structure definition goes here**

**}**

以下是定义结构体和定义类的示例：

**struct Resolution {**

**var width = 0**

**var heigth = 0**

**}**

**class VideoMode {**

**var resolution = Resolution()**

**var interlaced = false**

**var frameRate = 0.0**

**var name: String?**

**}**

类和结构体实例

生成结构体和类实例的语法非常相似：

**let someResolution = Resolution()**

**let someVideoMode = VideoMode()**

结构体类型的成员逐一初始化器

所有结构体都有一个自动生成的成员逐一初始化器，用于初始化新结构体实例中成员的属性。新实例中各个属性的初始值可以通过属性的名称传递到成员逐一初始化器之中：

**let vga = resolution（ width:640, heigth: 480）**

结构体和枚举是值类型,类是引用类型

实际上，在 Swift 中，所有的基本类型：整数(Integer)、浮点数(floating-point)、布尔值(Booleans)、字符串(string)、数组(array)和字典(dictionaries)，都是值类型，并且都是以结构体的形式在后台所实现。

恒等运算符

如果能够判定两个常量或者变量是否引用同一个类实例将会很有帮助。为了达到这个目的， Swift 内建了两个恒等运算符：

等价于 ( === )

不等价于 ( !== )

以下是运用这两个运算符检测两个常量或者变量是否引用同一个实例：

**if tenEighty === alsoTenTighty {**

**}**

“等价于”表示两个类类型(class type)的常量或者变量引用同一个类实例。  
“等于”表示两个实例的值“相等”或“相同”

类和结构体的选择

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

结构体的主要目的是用来封装少量相关简单数据值。有理由预计一个结构体实例在赋值或传递时，封装的数据将会被拷贝而不是被引用。任何在结构体中储存的值类型属性，也将会被拷贝，而不是被引用。结构体不需要去继承另一个已存在类型的属性或者行为。

合适的结构体候选者包括：  
几何形状的大小，封装一个 width 属性和 height 属性，两者均为 Double 类型。  
一定范围内的路径，封装一个 start 属性和 length 属性，两者均为 Int 类型。  
三维坐标系内一点，封装 x， y 和 z 属性，三者均为 Double 类型。

在所有其它案例中，定义一个类，生成一个它的实例，并通过引用来管理和传递。实际中，这意味着绝大部分的自定义数据构造都应该是类，而非结构体。

字典类型的赋值和拷贝行为

无论何时将一个字典实例赋给一个常量或变量，或者传递给一个函数或方法，这个字典会在赋值或调用发生时被拷贝。

如果字典实例中所储存的键(keys)和/或值(values)是值类型(结构体或枚举)，当赋值或调用发生时，它们都会被拷贝。相反，如果键(keys)和/或值(values)是引用类型，被拷贝的将会是引用，而不是被它们引用的类实例或函数。字典的键和值的拷贝行为与结构体所储存的属性的拷贝行为相同。

数组的赋值和拷贝行为

如果你将一个数组(Array)实例赋给一个变量或常量，或者将其作为参数传递给函数或方法调用，在事件发生时数组的内容不会被拷贝。相反，数组公用相同的元素序列。当你在一个数组内修改某一元素，修改结果也会在另一数组显示。

对数组来说，拷贝行为仅仅当操作有可能修改数组长度时才会发生。这种行为包括了附加(appending),插入(inserting),删除(removing)或者使用范围下标(ranged subscript)去替换这一范围内的元素。只有当数组拷贝确要发生时，数组内容的行为规则与字典中键值的相同。

确保数组的唯一性

在操作一个数组，或将其传递给函数以及方法调用之前是很有必要先确定这个数组是有一个唯一拷贝的。通过在数组变量上调用 unshare 方法来确定数组引用的唯一性。 (当数组赋给常量时，不能调用 unshare 方法)

**var a = [1, 2, 3]**

**var b = a**

**var c = a**

如果一个数组被多个变量引用，在其中的一个变量上调用 unshare 方法，则会拷贝此数组，此时这个变量将会有属于它自己的独立数组拷贝。

b 和 c 都引用了同一个数组。此时在 b 上调用 unshare 方法则会将  
b 变成一个唯一个拷贝：

**b.unshare()**

在 unshare 方法调用后再修改 b 中第一个元素的值，这三个数组(a,b,c)会返回不同的三个值。

判定两个数组是否共用相同元素

我们通过使用恒等运算符(identity operators)( === and !==)来判定两个数组或子数组共用  
相同的储存空间或元素。

还可以使用恒等运算符来判定两个子数组是否共用相同的元素。

强制复制数组

通过调用数组的 copy 方法进行强制显性复制。这个方法对数组进行了浅拷贝(shallow opy),并且返回一个包含此拷贝的新数组。

**var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]**

**var copiedNames = names.copy**

可以通过修改一个数组中某元素，并且检查另一个数组中对应元素的方法来判定  
names 数组确已被复制。如果你将 copiedNames 中第一个元素从"Mohsen"修改为"Mo",则names 数组返回的仍是拷贝发生前的"Mohsen"。

1. **属性**

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。在属性声明前使用@lazy 来标示一个延迟存储属性。

注意：必须将延迟存储属性声明成变量（使用 var 关键字），因为属性的值在实例构造完成之前可能无法得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

**class DataImporter {**

**/\***

**DataImporter 是一个将外部文件中的数据导入的类。**

**这个类的初始化会消耗不少时间。**

**\*/**

**var fileName = "data.txt"**

**// 这是提供数据导入功能**

**}**

**class DataManager {**

**@lazy var importer = DataImporter()**

**var data = String[]()**

**// 这是提供数据管理功能**

**}**

当 DataManager 的实例被创建时，没必要创建一个 DataImporter 的实例，更明智的是当用到 DataImporter 的时候才去创建它。

计算属性

除存储属性外，类、结构体和枚举可以定义计算属性，计算属性不直接存储值，而是提供一个 getter 来获取值，一个可选的 setter 来间接设置其他属性或变量的值。

**struct Point {**

**var x = 0.0, y = 0.0**

**}**

**struct Size {**

**var width = 0.0, height = 0.0**

**}**

**struct Rect {**

**var origin = Point()**

**var size = Size()**

**var center: Point {**

**get {//获取值的时候调用**

**let centerX = origin.x + (size.width / 2)**

**let centerY = origin.y + (size.height / 2)**

**return Point(x: centerX, y: centerY)**

**}**

**//括号里面是参数名**

**set(newCenter) {//设置值，初始化或者设置值得时候调用**

**origin.x = newCenter.x - (size.width / 2)**

**origin.y = newCenter.y - (size.height / 2)**

**}**

**}**

**}**

**var square = Rect(origin: Point(x: 0.0, y: 0.0),**

**size: Size(width: 10.0, height: 10.0))**

**let initialSquareCenter = square.center**

**square.center = Point(x: 15.0, y: 15.0)**

**println(" (\(square.origin.x),\(square.origin.y))")**

**// 输出 "square.origin is now at (10.0, 10.0)”**

便捷 setter 声明

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 newValue。下面是使用了便捷 setter 声明的 Rect 结构体代码：

**struct AlternativeRect {**

**var origin = Point()**

**var size = Size()**

**var center: Point {**

**get {**

**let centerX = origin.x + (size.width / 2)**

**let centerY = origin.y + (size.height / 2)**

**return Point(x: centerX, y: centerY)**

**}**

**set {//此处set后面没有括号**

**origin.x = newValue.x - (size.width / 2)**

**origin.y = newValue.y - (size.height / 2)**

**}**

**}**

**}**

只读计算属性

只有 getter 没有 setter 的计算属性就是只读计算属性。只读计算属性总是返回一个值，可以通过点运算符访问，但不能设置新的值。

注意：必须使用 var 关键字定义计算属性，包括只读计算属性，因为他们的值不是固定的。  
let 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

只读计算属性的声明可以去掉 get 关键字和花括号：

**struct Cuboid {**

**var width = 0.0, height = 0.0, depth = 0.0**

**var volume: Double {**

**return width \* height \* depth**

**}**

**}**

**let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)**

**println(fourByFiveByTwo is \(fourByFiveByTwo.volume)")**

**// 输出 "the volume of fourByFiveByTwo is 40.0"**

属性监视器

可以为除了延迟存储属性之外的其他存储属性添加属性监视器，也可以通过重载属性的方式为继承的属性（包括存储属性和计算属性）添加属性监视器。

可以为属性添加如下的一个或全部监视器：

willSet 在设置新的值之前调用  
didSet 在新的值被设置之后立即调用  
willSet 监视器会将新的属性值作为固定参数传入，在 willSet 的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称 newValue 表示。类似地， didSet 监视器会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 oldValue。

注意： willSet 和 didSet 监视器在属性初始化过程中不会被调用，他们只会当属性的值在初始化之外的地方被设置时被调用。

**class StepCounter {**

**var totalSteps: Int = 0 {**

**willSet(newTotalSteps) {**

**println("About to set totalSteps to \(newTotalSteps)")**

**}**

**didSet {**

**if totalSteps > oldValue {**

**println("Added \(totalSteps - oldValue) steps")**

**}**

**}**

**}**

**}**

**let stepCounter = StepCounter()**

**stepCounter.totalSteps = 200**

**// About to set totalSteps to 200**

**// Added 200 steps**

**stepCounter.totalSteps = 360**

**// About to set totalSteps to 360**

**// Added 160 steps**

**stepCounter.totalSteps = 896**

**// About to set totalSteps to 896**

**// Added 536 steps**

计算属性和属性监视器所描述的模式也可以用于全局变量和局部变量，全局变量是在函数、方法、闭包或任何类型之外定义的变量，局部变量是在函数、方法或闭包内部定义的变量。