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。

1. **方法**

方法是与某些特定类型相关联的函数。类、结构体、枚举都可以定义实例方法；实例方法为给定类型的实例封装了具体的任务与功能。类、结构体、枚举也可以定义类型方法；类型方法与类型本身相关联。

实例方法(Instance Methods)

实例方法是属于某个特定类、结构体或者枚举类型实例的方法。实例方法提供访问和修改实例属性的方法或提供与实例目的相关的功能，并以此来支撑实例的功能。

实例方法要写在它所属的类型的前后大括号之间。实例方法能够隐式访问它所属类型的所有的其他实例方法和属性。实例方法只能被它所属的类的某个特定实例调用。实例方法不能脱离于现存的实例而被调用。

方法的局部参数名称和外部参数名称

Swift 默认仅给方法的第一个参数名称一个局部参数名称;默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。

**class Counter {**

**var count: Int = 0**

**func incrementBy(amount: Int, numberOfTimes: Int) {**

**count += amount \* numberOfTimes**

**}**

**}**

incrementBy方法有两个参数： amount和 numberOfTimes。默认情况下，Swift 只把 amount当作一个局部名称，但是把 numberOfTimes 即看作局部名称又看作外部名称。下面调用这个方法：

**let counter = Counter()**

**counter.incrementBy(5, numberOfTimes: 3)**

**// counter value is now 15**

修改方法的外部参数名称

有时为方法的第一个参数提供一个外部参数名称是非常有用的，尽管这不是默认的行为。你可以自己添加一个显式的外部名称或者用一个井号（ #）作为第一个参数的前缀来把这个局部名称当作外部名称使用。  
相反，如果你不想为方法的第二个及后续的参数提供一个外部名称，可以通过使用下划线（ \_）作为该参数的显式外部名称，这样做将覆盖默认行为。

self 属性(The self Property)

类型的每一个实例都有一个隐含属性叫做 self， self 完全等同于该实例本身。你可以在一个实例的实例方法中使用这个隐含的 self 属性来引用当前实例。

在实例方法中修改值类型

结构体和枚举是值类型。一般情况下，值类型的属性不能在它的实例方法中被修改。如果你确实需要在某个具体的方法中修改结构体或者枚举的属性，你可以选择变异(mutating)这个方法，然后方法就可以从方法内部改变它的属性；并且它做的任何改变在方法结束时还会保留在原始结构中。方法还可以给它隐含的 self 属性赋值一个全新的实例，这个新实例在方法结束后将替换原来的实例。

要使用变异方法， 将关键字 mutating 放到方法的 func 关键字之前就可以了

**struct Point {**

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

**mutating func moveByX(deltaX: Double, y deltaY: Double) {**

**x += deltaX**

**y += deltaY**

**}**

**}**

**var somePoint = Point(x: 1.0, y: 1.0)**

**somePoint.moveByX(2.0, y: 3.0)**

**println(" (\(somePoint.x), \(somePoint.y))")**

**// 输出 "The point is now at (3.0, 4.0)"**

注意：不能在结构体类型常量上调用变异方法，因为常量的属性不能被改变，即使想改变的是常量的变量属性也不行，详情参见存储属性和实例变量

**let fixedPoint = Point(x: 3.0, y: 3.0)**

**fixedPoint.moveByX(2.0, y: 3.0)**

**// this will report an error**

在变异方法中给 self 赋值

变异方法能够赋给隐含属性 self 一个全新的实例。

**struct Point {**

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

**mutating func moveByX(deltaX: Double, y deltaY: Double) {**

**self = Point(x: x + deltaX, y: y + deltaY)**

**}**

**}**

枚举的变异方法可以把 self 设置为相同的枚举类型中不同的成员：

**enum TriStateSwitch {**

**case Off, Low, High**

**mutating func next() {**

**switch self {**

**case Off:**

**self = Low**

**case Low:**

**self = High**

**case High:**

**self = Off**

**}**

**}**

**}**

**var ovenLight = TriStateSwitch.Low**

**ovenLight.next()**

**// ovenLight 现在等于 .High**

**ovenLight.next()**

**// ovenLight 现在等于 .Off**

1. **类型方法(Type Methods)**

实例方法是被类型的某个实例调用的方法。你也可以定义类型本身调用的方法，这种方法就叫做类型方法。

声明类的类型方法，在方法的 func 关键字之前加上关键字 class；声明结构体和枚举的类型方法，在方法的 func 关键字之前加上关键字 static。

类型方法和实例方法一样用点语法调用。

但是，你是在类型层面上调用这个方法，而不是在实例层面上调用。下面是如何在 SomeClass 类上调用类型方法的例子：

**class SomeClass {**

**class func someTypeMethod() {**

**// type method implementation goes here**

**}**

**}**

**SomeClass.someTypeMethod()**

一个类型方法可以调用本类中另一个类型方法的名称，而无需在方法名称前面加上类型名称的前缀。同样，结构体和枚举的类型方法也能够直接通过静态属性的名称访问静态属性，而不需要类型名称前缀。

1. **附属脚本**

附属脚本可以定义在类（ Class）、结构体（ structure）和枚举（ enumeration）这些目标中，可以认为是访问对象、集合或序列的快捷方式，不需要再调用实例的特定的赋值和访问方法。

附属脚本语法

附属脚本允许你通过在实例后面的方括号中传入一个或者多个的索引值来对实例进行访问和赋值。语法类似于实例方法和计算型属性的混合。与定义实例方法类似，定义附属脚本使用 subscript 关键字，显式声明入参（一个或多个）和返回类型。与实例方法不同的是附属脚本可以设定为读写或只读。这种方式又有点像计算型属性的 getter 和 setter：

**subscript(index: Int) -> Int {**

**get {**

**// 返回与入参匹配的 Int 类型的值**

**}**

**set(newValue) {**

**// 执行赋值操作**

**}**

**}**

与只读计算型属性一样，可以直接将原本应该写在 get 代码块中的代码写在 subscript 中：

**subscript(index: Int) -> Int {**

**// 返回与入参匹配的 Int 类型的值**

**}**

下面代码演示了一个在 TimesTable 结构体中使用只读附属脚本的用法，该结构体用来展示传入整数的 n 倍。

**struct TimesTable {**

**let multiplier: Int**

**subscript(index: Int) -> Int {**

**return multiplier \* index**

**}**

**}**

**let threeTimesTable = TimesTable(multiplier: 3)**

**println("3 的 6 倍是\(threeTimesTable[6])")**

**// 输出 "3 的 6 倍是 18"**

1. **继承**

在 Swift 中，类可以调用和访问超类的方法，属性和附属脚本（ subscripts），并且可以重写（ override）这些方法，属性和附属脚本来优化或修改它们的行为。 Swift 会检查你的重写定义在超类中是否有匹配的定义，以此确保你的重写行为是正确的。

构造器的最简单形式就像一个没有参数的实例方法，使用 init 关键字：

**init() {**

**// 执行构造过程**

**}**

子类生成（ Subclassing）

为了指明某个类的超类，将超类名写在子类名的后面，用冒号分隔：

**class SomeClass: SomeSuperclass {**

**// 类的定义**

**}**

注意：不像 Objective-C，在 Swift 中，初始化器默认是不继承的，见初始化器的继承与重写, 子类只允许修改从超类继承来的变量属性，而不能修改继承来的常量属性。

重写（ Overriding）

子类可以为继承来的实例方法（ instance method），类方法（ class method），实例属性（ instance property），或附属脚本（ subscript）提供自己定制的实现（ implementation）。我们把这种行为叫重写（ overriding）。

如果要重写某个特性，你需要在重写定义的前面加上 override 关键字。这么做，你就表明了你是想提供一个重写版本，而非错误地提供了一个相同的定义。意外的重写行为可能会导致不可预知的错误，任何缺少 override 关键字的重写都会在编译时被诊断为错误。

访问超类的方法，属性及附属脚本

在合适的地方，你可以通过使用 super 前缀来访问超类版本的方法，属性或附属脚本：在方法 someMethod 的重写实现中，可以通过 super.someMethod()来调用超类版本的someMethod 方法。  
在属性 someProperty 的 getter 或 setter 的重写实现中，可以通过 super.someProperty来访问超类版本的 someProperty 属性。  
在附属脚本的重写实现中，可以通过 super[someIndex]来访问超类版本中的相同附属脚本。

重写方法

**class Vehicle {**

**var numberOfWheels: Int**

**var maxPassengers: Int**

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

**return "\(numberOfWheels); \(maxPassengers)"**

**}**

**init() {**

**numberOfWheels = 0**

**maxPassengers = 1**

**}**

**}**

下面的例子定义了 Vehicle 的一个新的子类，叫 Car，它重写了从 Vehicle 类继承来的description 方法：

**class Car: Vehicle {**

**var speed: Double = 0.0**

**init() {**

**super.init()**

**maxPassengers = 5**

**numberOfWheels = 4**

**}**

**override func description() -> String {**

**return super.description() + "; "+ "\(speed) mph"**

**}**

**}**

重写属性

重写属性的 Getters 和 Setters

你可以将一个继承来的只读属性重写为一个读写属性，只需要你在重写版本的属性里提供getter 和 setter 即可。但是，你不可以将一个继承来的读写属性重写为一个只读属性。

注意：如果你在重写属性中提供了 setter，那么你也一定要提供 getter。如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接返回 super.someProperty 来返回继承来的值。正如下面的 SpeedLimitedCar 的例子所示。

**class SpeedLimitedCar: Car {**

**override var speed: Double {**

**get {**

**return super.speed**

**}**

**set {**

**super.speed = min(newValue, 40.0)**

**}**

**}**

**}**

重写属性观察器（ Property Observer）

注意：你不可以为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察器。这些属性的值是不可以被设置的，所以，为它们提供 willSet 或 didSet 实现是不恰当。此外还要注意，你不可以同时提供重写的 setter 和重写的属性观察器。如果你想观察属性值的变化，并且你已经为那个属性提供了定制的 setter，那么你在 setter 中就可以观察到任何值变化了。

**class Car: Vehicle {**

**var speed: Double = 0.0**

**init() {**

**super.init()**

**maxPassengers = 5**

**numberOfWheels = 4**

**}**

**override func description() -> String {**

**return super.description() + "at \(speed) mph"**

**}**

**}**

下面的例子定义了一个新类叫 AutomaticCar，它是 Car 的子类。 AutomaticCar 表示自动挡汽车，它可以根据当前的速度自动选择合适的挡位。 AutomaticCar 也提供了 定制的description 方法，可以输出当前挡位。

**class AutomaticCar: Car {**

**var gear = 1**

**override var speed: Double {**

**didSet {**

**gear = Int(speed / 10.0) + 1**

**}**

**}**

**override func description() -> String {**

**return super.description() + " in gear \(gear)"**

**}**

**}**

**let automatic = AutomaticCar()**

**automatic.speed = 35.0**

**println("AutomaticCar: \(automatic.description())")**

**// AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph in gear 4**

防止重写

通过把方法，属性或附属脚本标记为 final 来防止它们被重写，只需要在声明关键字前加上@final 特性即可。

可以通过在关键字 class 前添加@final 特性（ @final class）来将整个类标记为 final 的，这样的类是不可被继承的，否则会报编译错误。

1. **构造过程**

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字 init 命名。

定制化构造过程

可以在定义构造器时提供构造参数，为其提供定制化构造所需值的类型和名字。构造器参数的功能和语法跟函数和方法参数相同。

内部和外部参数名

构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器  
时，主要通过构造器中的参数名和类型来确定需要调用的构造器。正因为参数如此重要，如果你在定义构造器时没有提供参数的外部名字， Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名，就相当于在每个构造参数之前加了一个哈希符号。

**struct Color {**

**let red = 0.0, green = 0.0, blue = 0.0**

**init(red: Double, green: Double, blue: Double) {**

**self.red = red**

**self.green = green**

**self.blue = blue**

**}**

**}**

每当你创建一个新的 Color 实例，你都需要通过三种颜色的外部参数名来传值，并调用构造器。

**let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)**

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性--不管是因为它无法在初始化时赋值，还是因为它可以在之后某个时间点可以赋值为空--你都需要将它定义为可选类型 optional type。可选类型的属性将自动初始化为空 nil，表示这个属性是故意在初始化时设置为空的。

**class SurveyQuestion {**

**var text: String**

**var response: String?**

**init(text: String) {**

**self.text = text**

**}**

**func ask() {**

**println(text)**

**}**

**}**

**let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")**

**cheeseQuestion.ask()**

**// 输出 "Do you like cheese?"**

**cheeseQuestion.response = "Yes, I do like cheese.**

构造过程中常量属性的修改

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

**class SurveyQuestion {**

**let text: String**

**var response: String?**

**init(text: String) {**

**self.text = text**

**}**

**func ask() {**

**println(text)**

**}**

**}**

**let beetsQuestion = SurveyQuestion(text: "How about beets?")**

**beetsQuestion.ask()**

**// 输出 "How about beets?"**

**beetsQuestion.response = "I also like beets. (But not with cheese.)**

默认构造器

Swift 将为所有属性已提供默认值的且自身没有定义任何构造器的结构体或基类，提供一个默认的构造器。这个默认构造器将简单的创建一个所有属性值都设置为默认值的实例。

**class ShoppingListItem {**

**var name: String?**

**var quantity = 1**

**var purchased = false**

**}**

**var item = ShoppingListItem()**

由于 ShoppingListItem 类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器（尽管代码中没有显式为 name 属性设置默认值，但由于 name 是可选字符串类型，它将默认设置为 nil）。

结构体的逐一成员构造器

如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。逐一成员构造器是用来初始化结构体新实例里成员属性的快捷方法。我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

**struct Size {**

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

**}**

**let twoByTwo = Size(width: 2.0, height: 2.0)**

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。构造器代理的实现规则和形式在值类型和类类型中有所不同。值类型（结构体和枚举类型）不支持继承，所以构造器代理的过程相对简单，因为它们只能代理任务给本身提供的其它构造器。类则不同，它可以继承自其它类（请参考继承），这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。

**struct Size {**

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

**}**

**struct Point {**

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

**}**

**struct Rect {**

**var origin = Point()**

**var size = Size()**

**init() {}**

**init(origin: Point, size: Size) {**

**self.origin = origin**

**self.size = size**

**}**

**init(center: Point, size: Size) {**

**let originX = center.x - (size.width / 2)**

**let originY = center.y - (size.height / 2)**

**self.init(origin: Point(x: originX, y: originY), size: size)**

**}**

**}**

类的继承和构造过程

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

构造器的继承和重载

Swift 中的子类不会默认继承父类的构造器。 Swift 的这种机制可以防止一个父类的简单构造器被一个更专业的子类继承，并被错误的用来创建子  
类的实例。希望自定义的子类中能实现一个或多个跟父类相同的构造器--也许是为了完成一些定制的构造过程--你可以在你定制的子类中提供和重载与父类相同的构造器。

注意：与方法、属性和下标不同，在重载构造器时你没有必要使用关键字 override。

自动构造器的继承

子类不会默认继承父类的构造器。但是如果特定条件可以满足，父类构造器是  
可以被自动继承的。在实践中，这意味着对于许多常见场景你不必重载父类的构造器，并且在尽可能安全的情况下以最小的代价来继承父类的构造器。

假设要为子类中引入的任意新属性提供默认值，请遵守以下 2 个规则：

规则 1  
如果子类没有定义任何指定构造器，它将自动继承所有父类的指定构造器。  
规则 2  
如果子类提供了所有父类指定构造器的实现--不管是通过规则 1 继承过来的，还是通过自定义实现的--它将自动继承所有父类的便利构造器。

指定构造器和便利构造器的语法  
类的指定构造器的写法跟值类型简单构造器一样：

**init(parameters) {**

**statements**

**}**  
便利构造器也采用相同样式的写法，但需要在 init 关键字之前放置 convenience 关键字，并使用空格将它们俩分开：

**convenience init(parameters) {**

**statements**

**}**  
指定构造器和便利构造器实战

类层次中的基类是 Food，它是一个简单的用来封装食物名字的类。 Food 类引入了一个叫做 name 的 String 类型属性，并且提供了两个构造器来创建 Food 实例：

**class Food {**

**var name: String**

**init(name: String) {**

**self.name = name**

**}**

**convenience init() {**

**self.init(name: "[Unnamed]")**

**}**

**}**

类层级中的第二个类是 Food 的子类 RecipeIngredient。 RecipeIngredient 类构建了食谱中的一味调味剂。它引入了 Int 类型的数量属性 quantity（以及从 Food 继承过来的 name 属性），并且定义了两个构造器来创建 RecipeIngredient 实例：

**class RecipeIngredient: Food {**

**var quantity: Int**

**init(name: String, quantity: Int) {**

**self.quantity = quantity**

**super.init(name: name)**

**}**

**convenience init(name: String) {**

**self.init(name: name, quantity: 1)**

**}**

**}**

所有的这三种构造器都可以用来创建新的 RecipeIngredient 实例：

**let oneMysteryItem = RecipeIngredient()**

**let oneBacon = RecipeIngredient(name: "Bacon")**

**let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)**

购物单中的每一项总是从 unpurchased 未购买状态开始的。为了展现这一事实，ShoppingListItem 引入了一个布尔类型的属性 purchased，它的默认值是 false。ShoppingListItem 还添加了一个计算型属性 description，它提供了关于 ShoppingListItem实例的一些文字描述：

**class ShoppingListItem: RecipeIngredient {**

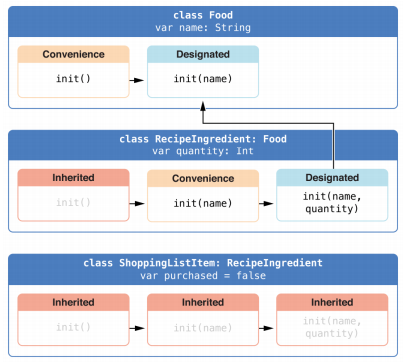
**var purchased = false**

**var description: String {**

**var output = "\(quantity) x \(name.lowercaseString)"**

**output += purchased ? " ?" : " ?"return output}**

**}**

****

你可以使用全部三个继承来的构造器来创建 ShoppingListItem 的新实例：

**var breakfastList = [**

**ShoppingListItem(),**

**ShoppingListItem(name: "Bacon"),**

**ShoppingListItem(name: "Eggs", quantity: 6),**

**]**

**breakfastList[0].name = "Orange juice"**

**breakfastList[0].purchased = true**

**for item in breakfastList {**

**println(item.description)**

**}**

**// 1 x orange juice ?**

**// 1 x bacon ?**

**// 6 x eggs ?**

通过闭包和函数来设置属性的默认值

如果某个存储型属性的默认值需要特别的定制或准备，你就可以使用闭包或全局函数来为其属性提供定制的默认值。每当某个属性所属的新类型实例创建时，对应的闭包或函数会被调用，而它们的返回值会当做默认值赋值给这个属性。

下面列举了闭包如何提供默认值的代码概要：  
**class SomeClass {**

**let someProperty: SomeType = {**

**// 在这个闭包中给 someProperty 创建一个默认值**

**// someValue 必须和 SomeType 类型相同**

**return someValue**

**}()**

**}**

注意闭包结尾的大括号后面接了一对空的小括号。这是用来告诉 Swift 需要立刻执行此闭包。如果你忽略了这对括号，相当于是将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。

**struct Checkerboard {**

**let boardColors: Bool[] = {**

**var temporaryBoard = Bool[]()**

**var isBlack = false**

**for i in 1...10 {**

**for j in 1...10 {**

**temporaryBoard.append(isBlack)**

**isBlack = !isBlack**

**}**

**isBlack = !isBlack**

**}**

**return temporaryBoard**

**}()**

**func squareIsBlackAtRow(row: Int, column: Int) -> Bool {**

**return boardColors[(row \* 10) + column]**

**}**

**}**

每当一个新的 Checkerboard 实例创建时，对应的赋值闭包会执行，一系列颜色值会被计算出来作为默认值赋值给 boardColors。

1. **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，你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，你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

1. **自动引用计数**

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。

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。

**class Person {**

**let name: String**

**init(name: String) {**

**self.name = name**

**println("\(name) is being initialized")**

**}**

**deinit {**

**println("\(name) is being deinitialized")**

**}**

**}**

**var reference1: Person?  
var reference2: Person?  
var reference3: Person?**  
现在创建一个新的 Person 实例，并且将它赋值给上述三个变量重的一个：  
**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。

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**

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

然而，我们有可能会写出这样的代码，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。

**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 }**

**var tenant: Person?**

**deinit {**

**println("Apartment #\(number) is being deinitialized")**

**}**

**}**

此时Person和Apartment互相持有对方的强引用

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。

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。

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

**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")**

**}**

**}**

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。

**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")**

**}**

**}**

**john = Customer(name: "John Appleseed")**

**john!.card = CreditCard(number: 1234\_5678\_9012\_3456, customer:john!)**

**john = nil**

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

**// 打印"Card #1234567890123456 is being deinitialized"**

无主引用以及隐式展开的可选属性

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，另一个类用隐式展开的可选属性。

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：

**class Country {**

**let name: String**

**let capitalCity: City!**

**init(name: String, capitalName: String) {**

**self.name = name**

**self.capitalCity = City(name: capitalName, country: self)**

**}**

**}**

**class City {**

**let name: String**

**unowned let country: Country**

**init(name: String, country: Country) {**

**self.name = name**

**self.country = country**

**}**

**}**

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：  
**var country = Country(name: "Canada", capitalName: "Ottawa")**

**println("\(country.name)\(country.capitalCity.name)")  
// 打印"Canada's capital city is called Ottawa"**

闭包产生的强引用环

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，从而产生了抢引用环。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(closuer capturelist)。

**class HTMLElement {**

**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")**

**}**

**}**

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"<p>hello, world</p>"**  
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。

注意：虽然闭包多次使用了 self，它只占有 HTMLElement 实例的一个强引用。

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。

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。

**class HTMLElement {**

**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")**

**}**

**}**

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。

**var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")**

**println(paragraph!.asTHML())**

**// 打印"<p>hello, world</p>"**

**paragraph = nil**

**// 打印"p is being deinitialized"**

1. **自判断链接**

自判断链接（ Optional Chaining）是一种可以请求和调用属性、方法及子脚本的过程，它的自判断性体现于请求或调用的目标当前可能为空（ nil）。如果自判断的目标有值，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。

自判断链接可替代强制拆包

**class Person {**

**var residence: Residence?**

**}**

**class Residence {**

**var numberOfRooms = 1**

**}**

**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，会运行通过，且会将 roomCount 设置为一个 int 类型的合理值。

自判断链接提供了一种另一种获得 numberOfRooms 的方法。利用自判断链接，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 来链接自判断 residence？属性，如果 residence 存在则取回 numberOfRooms的值。

你可以自己定义一个 Residence 实例给 john.residence，这样它就不再为空了：

**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自判  
//断链接来获得 numberOfRoooms，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)"。**

为自判断链接定义模型类

可以使用自判断链接来多层调用属性，方法，和子脚本。这让你可以利用它们之间的复杂模型来获取更底层的属性，并检查是否可以成功获取此类底层属性。

通过自判断链接调用属性

正如上面“ 自判断链接可替代强制拆包”中所述，你可以利用自判断链接的自判断值获取属性，并且检查属性是否获取成功。然而，你不能使用自判断链接为属性赋值。

通过自判断链接调用方法

你可以使用自判断链接的来调用自判断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，你依然可以使用自判断链接来达成这一目的。

使用自判断链接调用子脚本

你可以使用自判断链接来尝试从子脚本获取值并检查子脚本的调用是否成功，然而，你不能通过自判断链接来设置子代码。

注意： 当你使用自判断链接来获取子脚本的时候，你应该将问号放在子脚本括号的前面而不是后面。自判断链接的问号一般直接跟在自判断表达语句的后面。

**class Residence {**

**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?**

**}**

**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 "Unable to retrieve the first room name."。**

连接多层链接

你可以将多层自判断链接连接在一起，可以掘取模型内更下层的属性方法和子脚本。然而多层自判断链接不能再添加比已经返回的自判断值更多的层。 也就是说：

如果你试图获得的类型不是自判断类型，由于使用了自判断链接它将变成自判断类型。

如果你试图获得的类型已经是自判断类型，由于自判断链接它也不会提高自判断性。  
因此，如果你试图通过自判断链接获得 Int 值，不论使用了多少层链接返回的总是 Int？。相似的，如果你试图通过自判断链接获得Int？值，不论使用了多少层链接返回的总是Int？。

链接自判断返回值的方法

下面的例子通过自判断链接调用了 Address 类中的 buildingIdentifier 方法。这个方法的返回值类型是 String?。如上所述，这个方法在自判断链接调用后最终的返回值类型依然是String？：

**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**

**}**

**}**

**}**

**If let buildingIdentifier = john.residence?.address?.buildingIdentifier() {**

**println("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自判断链接，将自判断链接问号符放在方法括号的后面：

**if let upper = john.residence?.address?.buildingIdentifier()?.uppercas  
eString {**

**println("uppercase building identifier is \(upper).")**

**}**

**// 打印 "John's uppercase building identifier is THE LARCHES."。**

注意： 在上面的例子中，你将自判断链接问号符放在括号后面是因为你想要链接的自判断值是 buildingIdentifier 方法的返回值，不是 buildingIdentifier 方法本身。

1. **类型转换**

类型转换在 Swift 中使用 is 和 as 操作符实现。这两个操作符提供了一种简单达意的方式去检查值的类型或者转换它的类型。

定义一个类层次作为例子

**class MediaItem {**

**var name: String**

**init(name: String) {**

**self.name = name**

**}**

**}**

下一个代码段定义了 MediaItem 的两个子类。第一个子类 Movie，在父类（或者说基类）的基础上增加了一个 director（导演） 属性，和相应的初始化器。第二个类在父类的基础上增加了一个 artist（艺术家） 属性，和相应的初始化器：

**class Movie: MediaItem {**

**var director: String**

**init(name: String, director: String) {**

**self.director = director**

**super.init(name: name)**

**}**

**}**

**class Song: MediaItem {**

**var artist: String**

**init(name: String, artist: String) {**

**self.artist = artist**

**super.init(name: name)**

**}**

**}**

Swift 的类型检测器能够演绎出 Movie 和 Song 有共同的父类  
MediaItem ，所以它推断出 MediaItem[] 类作为 library 的类型。

**let library = [**

**Movie(name: "Casablanca", director: "Michael Curtiz"),**

**Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),**

**Movie(name: "Citizen Kane", director: "Orson Welles"),**

**Song(name: "The One And Only", artist: "Chesney Hawkes"),**

**Song(name: "Never Gonna Give You Up", artist: "Rick Astley")**

**]**

**// the type of "library" is inferred to be MediaItem[]**

在幕后 library 里存储的项依然是 Movie 和 Song 类型的，但是，若你迭代它，取出的实例会是 MediaItem 类型的，而不是 Movie 和 Song 类型的。

检查类型

用类型检查操作符(is)来检查一个实例是否属于特定子类型。类型检查操作符返回 true 若实例属于那个子类型，若不属于返回 false 。

下面的例子定义了连个变量， movieCount 和 songCount，用来计算数组 library 中 Movie和 Song 类型的实例数量。

**var movieCount = 0**

**var songCount = 0**

**for item in library {**

**if item is Movie {**

**++movieCount**

**} else if item is Song {**

**++songCount**

**}**

**}**

**println("Media library contains \(movieCount) movies and \(songCount)songs")**

**// prints "Media library contains 2 movies and 3 songs"**

向下转型（简称下转）

某类型的一个常量或变量可能在幕后实际上属于一个子类。可以尝试向下转到它的子类型，用类型转换操作符(as)

因为向下转型可能会失败，类型转换操作符带有两种不同形式。可选形式（optional form）as? 返回一个你试图下转成的类型的可选值（ optional value）。强制形式 as 把试图向下转型和强制解包（ force-unwraps）结果作为一个混合动作。

当你不确定下转可以成功时，用类型转换的可选形式(as?)。可选形式的类型转换总是返回一个可选值（ optional value），并且若下转是不可能的，可选值将是 nil 。这使你能够检查下转是否成功。

只有你可以确定下转一定会成功时，才使用强制形式。当你试图下转为一个不正确的类型时，强制形式的类型转换会触发一个 runtime error。

在这个示例中，数组中的每一个 item 可能是 Movie 或 Song。 事前你不知道每个 item 的真实类型，所以这里使用可选形式的类型转换 (as?)去检查循环里的每次下转：

**for item in library {**

**if let movie = item as? Movie {**

**println("'\(movie.name)', dir. \(movie.director)")**

**} else if let song = item as? Song {**

**println("Song: '\(song.name)', by \(song.artist)")**

**}**

**}**

**// Movie: 'Casablanca', dir. Michael Curtiz**

**// Song: 'Blue Suede Shoes', by Elvis Presley**

**// Movie: 'Citizen Kane', dir. Orson Welles**

**// Song: 'The One And Only', by Chesney Hawkes**

**// Song: 'Never Gonna Give You Up', by Rick Astley**

注意：转换没有真的改变实例或它的值。潜在的根本的实例保持不变；只是简单地把它作为它被转换成的类来使用。

Any 和 AnyObject 的转换

Swift 为不确定类型提供了两种特殊类型别名：  
1. AnyObject 可以代表任何 class 类型的实例。(例如UILabel labelInstance)  
2. Any 可以表示任何类型，除了方法类型（ function types）。(例如UIButton)  
注意：只有当你明确的需要它的行为和功能时才使用 Any 和 AnyObject。在你的代码里使用你期望的明确的类型总是更好的。

AnyObject 类型  
当需要在工作中使用 Cocoa APIs，它一般接收一个 AnyObject[]类型的数组，或者说“一个任何对象类型的数组”。这是因为 OC 没有明确的类型化数组。但是，你常常可以确定包含在仅从你知道的 API 信息提供的这样一个数组中的对象的类型。

在这些情况下，你可以使用强制形式的类型转换(as)来下转在数组中的每一项到比AnyObject 更明确的类型，不需要可选解包（optional unwrapping）。

下面的示例定义了一个 AnyObject[] 类型的数组并填入三个 Movie 类型的实例：

**let someObjects: AnyObject[] = [**

**Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),**

**Movie(name: "Moon", director: "Duncan Jones"),**

**Movie(name: "Alien", director: "Ridley Scott"**

**]**

因为知道这个数组只包含 Movie 实例，你可以直接用(as)下转并解包到不可选的 Movie 类型（ ps：其实就是我们常用的正常类型，这里是为了和可选类型相对比）：

**for object in someObjects {**

**let movie = object as Movie**

**println("'\(movie.name)', dir. \(movie.director)")**

**}**

**// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick**

**// Movie: 'Moon', dir. Duncan Jones**

**// Movie: 'Alien', dir. Ridley Scott**  
为了变为一个更短的形式，下转 someObjects 类型成功 Movie[] 类型代替下转每一项：

**for movie in someObjects as Movie[] {**

**println("'\(movie.name)', dir. \(movie.director)")**

**}**

**// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick**

**// Movie: 'Moon', dir. Duncan Jones**

**// Movie: 'Alien', dir. Ridley Scott**

Any 类型

这里有个示例，使用 Any 类型来和混合的不同类型一起工作，包括非 class 类型。它创建了一个可以存储 Any 类型的数组 things：

**var things = Any[]()**

**things.append(0)**

**things.append(0.0)**

**things.append(42)**

**things.append(3.14159)**

**things.append("hello")**

**things.append((3.0, 5.0))**

**things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))**

你可以在 switch cases 里用 is 和 as 操作符来发觉只知道是 Any 或 AnyObject 的常量或变量的类型。 下面的示例迭代 things 数组中的每一项的并用 switch 语句查找每一项的类型。这几种 switch 语句的情形绑定它们匹配的值到一个规定类型的常量，让它们可以打印它们的值：

**for thing in things {**

**switch thing {**

**case 0 as Int:**

**println("zero as an Int")**

**case 0 as Double:**

**println("zero as a Double")**

**case let someInt as Int:**

**println("an integer value of \(someInt)")**

**case let someDouble as Double where someDouble > 0:**

**println("a positive double value of \(someDouble)")**

**case is Double:**

**println("some other double value that I don't want to print")**

**case let someString as String:**

**println("a string value of \"\(someString)\"")**

**case let (x, y) as (Double, Double):**

**println("an (x, y) point at \(x), \(y)")**

**case let movie as Movie:**

**println("'\(movie.name)'. \(movie.director)")**

**default:**

**println("something else")**

**}**

**}**

**// zero as an Int**

**// zero as a Double**

**// an integer value of 42**

**// a positive double value of 3.14159**

**// a string value of "hello"**

**// an (x, y) point at 3.0, 5.0**

**// a movie called 'Ghostbusters', dir. Ivan Reitman。**  
注意：在一个 switch 语句的 case 中使用强制形式的类型转换操作符(as, 而不是 as?) 来检查和转换到一个规定的类型。在 switch case 语句的内容中这种检查总是安全的。

1. **类型嵌套**

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。

1. **扩展**

扩展就是向一个已有的类、结构体或枚举类型添加新功能（ functionality）。这包括在没有权限获取原始源代码的情况下扩展类型的能力（即逆向建模）。扩展和 Objective-C 中的分类（ categories）类似。（不过与 Objective-C 不同的是， Swift 的扩展没有名字。）

Swift 中的扩展可以：  
1. 添加计算型属性和计算静态属性  
2. 定义实例方法和类型方法  
3. 提供新的构造器  
4. 定义下标  
5. 定义和使用新的嵌套类型  
6. 使一个已有类型符合某个接口  
注意：如果你定义了一个扩展向一个已有类型添加新功能，那么这个新功能对该类型的所有已有实例中都是可用的，即使它们是在你的这个扩展的前面定义的。

扩展语法（ Extension Syntax）

声明一个扩展使用关键字 extension：

**extension SomeType {**

**// 加到 SomeType 的新功能写到这里**

**}**  
一个扩展可以扩展一个已有类型，使其能够适配一个或多个协议（ protocol）。当这种情况发生时，接口的名字应该完全按照类或结构体的名字的方式进行书写：

**extension SomeType: SomeProtocol, AnotherProctocol {**

**// 协议实现写到这里**

**}**  
按照这种方式添加的协议遵循者（ protocol conformance）被称之为在扩展中添加协议遵循者

计算型属性（ Computed Properties）  
扩展可以向已有类型添加计算型实例属性和计算型类型属性。下面的例子向 Swift 的内建Double 类型添加了 5 个计算型实例属性，从而提供与距离单位协作的基本支持。

**extension Double {**

**var km: Double { return self \* 1\_000.0 }**

**var m : Double { return self }**

**var cm: Double { return self / 100.0 }**

**var mm: Double { return self / 1\_000.0 }**

**var ft: Double { return self / 3.28084 }**

**}**

**let oneInch = 25.4.mm**

**println("One inch is \(oneInch) meters")**

**// 打印输出： "One inch is 0.0254 meters"**

**let threeFeet = 3.ft**

**println("Three feet is \(threeFeet) meters")**

**// 打印输出： "Three feet is 0.914399970739201 meters"**

这些属性是只读的计算型属性，所有从简考虑它们不用 get 关键字表示。它们的返回值是Double 型，而且可以用于所有接受 Double 的数学计算中：

**let aMarathon = 42.km + 195.m**

**println("A marathon is \(aMarathon) meters long")**

**// 打印输出： "A marathon is 42495.0 meters long"**  
注意：扩展可以添加新的计算属性，但是不可以添加存储属性，也不可以向已有属性添加属性观测器(property observers)。

构造器（ Initializers）

扩展可以向已有类型添加新的构造器。这可以让你扩展其它类型，将你自己的定制类型作为构造器参数，或者提供该类型的原始实现中没有包含的额外初始化选项。

**struct Size {**

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

**}**

**struct Point {**

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

**}**

**struct Rect {**

**var origin = Point()**

**var size = Size()**

**}**

可以提供一个额外的使用特殊中心点和大小的构造器来扩展 Rect 结构体：

**extension Rect {**

**init(center: Point, size: Size) {**

**let originX = center.x - (size.width / 2)**

**let originY = center.y - (size.height / 2)**

**self.init(origin: Point(x: originX, y: originY), size: size)**

**}**

**}**

这个新的构造器首先根据提供的 center 和 size 值计算一个合适的原点。然后调用该结构体自动的成员构造器 init(origin:size:)，该构造器将新的原点和大小存到了合适的属性中：

**let centerRect = Rect(center: Point(x: 4.0, y: 4.0),**

**size: Size(width: 3.0, height: 3.0))**

**// centerRect 的原点是 (2.5, 2.5)，大小是 (3.0, 3.0)**  
注意：如果你使用扩展提供了一个新的构造器，你依旧有责任保证构造过程能够让所有实例完全初始化。

扩展方法（ Methods）

扩展可以向已有类型添加新的实例方法和类型方法。下面的例子向 Int 类型添加一个名为repetitions 的新实例方法：

**extension Int {**

**func repetitions(task: () -> ()) {**

**for i in 0..self {**

**task()**

**}**

**}**

**}**

这个 repetitions 方法使用了一个() -> ()类型的单参数（ single argument），表明函数没有参数而且没有返回值。

定义该扩展之后，你就可以对任意整数调用 repetitions 方法,实现的功能则是多次执行某任务：

**3.repetitions({**

**println("Hello!")**

**})**

**// Hello!**

**// Hello!**

**// Hello!**  
可以使用 trailing 闭包使调用更加简洁：

**3.repetitions{**

**println("Goodbye!")**

**}**

**// Goodbye!**

**// Goodbye!**

**// Goodbye!**

修改实例方法（ Mutating Instance Methods）

通过扩展添加的实例方法也可以修改该实例本身。结构体和枚举类型中修改 self 或其属性的方法必须将该实例方法标注为 mutating，正如来自原始实现的修改方法一样。  
下面的例子向 Swift 的 Int 类型添加了一个新的名为 square 的修改方法，来实现一个原始值的平方计算：

**extension Int {**

**mutating func square() {**

**self = self \* self**

**}**

**}**

**var someInt = 3**

**someInt.square()  
// someInt 现在值是 9**

下标（ Subscripts）

扩展可以向一个已有类型添加新下标。这个例子向 Swift 内建类型 Int 添加了一个整型下标。  
该下标[n]返回十进制数字从右向左数的第 n 个数字

**extension Int {**

**subscript(digitIndex: Int) -> Int {**

**var decimalBase = 1**

**for \_ in 1...digitIndex {**

**decimalBase \*= 10**

**}**

**return (self / decimalBase) % 10**

**}**

**}**

**746381295[0]**

**// returns 5**

**746381295[1]**

**// returns 9**

**746381295[2]**

**// returns 2**

**746381295[8]**

**// returns 7**

如果该 Int 值没有足够的位数，即下标越界，那么上述实现的下标会返回 0， 因为它会在数字左边自动补 0：

**746381295[9]**

**//returns 0,**  
即等同于：

**0746381295[9]**

嵌套类型（ Nested Types）

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

**extension Character {**

**enum Kind {**

**case Vowel, Consonant, Other  
}**

**var kind: Kind {**

**switch String(self).lowercaseString {**

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

**return .Vowel**

**case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",**

**"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":**

**return .Consonant**

**default:**

**return .Other**

**}**

**}**

**}**  
该例子向 Character 添加了新的嵌套枚举。这个名为 Kind 的枚举表示特定字符的类型。具体来说，就是表示一个标准的拉丁脚本中的字符是元音还是辅音（不考虑口语和地方变种），或者是其它类型。

1. **协议**

Protocol(协议)用于统一方法和属性的名称，而不实现任何功能。协议能够被类，枚举，结构体实现，满足协议要求的类，枚举，结构体被称为协议的遵循者。

协议的语法

协议的定义与类，结构体，枚举的定义非常相似，如下所示：

protocol SomeProtocol {

// 协议内容

}

在类，结构体，枚举的名称后加上协议名称，中间以冒号:分隔即可实现协议；实现多个协议时，各协议之间用逗号,分隔，如下所示：

struct SomeStructure: FirstProtocol, AnotherProtocol {

// 结构体内容

}  
当某个类含有父类的同时并实现了协议，应当把父类放在所有的协议之前，如下所示：

class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {

// 类的内容

}

属性要求

协议能够要求其遵循者必须含有一些特定名称和类型的实例属性(instance property)或类属性 (type property)，也能够要求属性的(设置权限)settable 和(访问权限)gettable，但它不要求属性是存储型属性(stored property)还是计算型属性(calculate property)。  
通常前置 var 关键字将属性声明为变量。在属性声明后写上{ get set }表示属性为可读写的。{ get }用来表示属性为可读的。即使你为可读的属性实现了 setter 方法，它也不会出错。

protocol SomeProtocol {

var musBeSettable : Int { get set }

var doesNotNeedToBeSettable: Int { get }

}

用类来实现协议时，使用class关键字来表示该属性为类成员；用结构体或枚举实现协议时，则使用 static 关键字来表示：

用类来实现协议时，使用class关键字来表示该属性为类成员；用结构体或枚举实现协议时，则使用 static 关键字来表示：

protocol AnotherProtocol {

class var someTypeProperty: Int { get set }

}

protocol FullyNamed {

var fullName: String { get }

}

FullyNamed 协议含有 fullName 属性。因此其遵循者必须含有一个名为 fullName，类型为String 的可读属性。

方法要求

协议能够要求其遵循者必备某些特定的实例方法和类方法。协议方法的声明与普通方法声明相似，但它不需要方法内容。

注意： 协议方法支持变长参数(variadic parameter)，不支持默认参数(default parameter)。前置 class 关键字表示协议中的成员为类成员；当协议用于被枚举或结构体遵循时，则使用static 关键字。如下所示：

protocol SomeProtocol {

class func someTypeMethod()

}

protocol RandomNumberGenerator {

func random() -> Double

}

突变方法要求

能在方法或函数内部改变实例类型的方法称为突变方法。在值类型(Value Type) 中的的函数前缀加上 mutating 关键字来表示该函数允许改变该实例和其属  
性的类型。 这一变换过程在实例方法(Instance Methods)章节中有详细描述。

类中的成员为引用类型(Reference Type)，可以方便的修改实例及其属性的值而无  
需改变类型；而结构体和枚举中的成员均为值类型(Value Type)，修改变量的值就相当于修改变量的类型，而 Swift 默认不允许修改类型，因此需要前置 mutating 关键字用来表示该函数中能够修改类型

注意：用 class 实现协议中的 mutating 方法时，不用写 mutating 关键字；用结构体，枚举实现协议中的 mutating 方法时，必须写 mutating 关键字。

如下所示， Togglable 协议含有 toggle 函数。根据函数名称推测， toggle 可能用于切换或恢复某个属性的状态。 mutating 关键字表示它为突变方法：

protocol Togglable {

mutating func toggle()

}  
当使用枚举或结构体来实现 Togglabl 协议时，必须在 toggle 方法前加上 mutating 关键字。  
如下所示， OnOffSwitch 枚举遵循了 Togglable 协议， On， Off 两个成员用于表示当前状态

enum OnOffSwitch: Togglable {

case Off, On

mutating func toggle() {

switch self {

case Off:

self = On

case On:

self = Off

}

}

}

var lightSwitch = OnOffSwitch.Off

lightSwitch.toggle()

//lightSwitch 现在的值为 .On

协议类型  
协议本身不实现任何功能，但你可以将它当做类型来使用。  
使用场景：  
1. 作为函数，方法或构造器中的参数类型，返回值类型  
2. 作为常量，变量，属性的类型  
3. 作为数组，字典或其他容器中的元素类型  
注意：协议类型应与其他类型(Int， Double， String)的写法相同，使用驼峰式

class Dice {

let sides: Int

let generator: RandomNumberGenerator

init(sides: Int, generator: RandomNumberGenerator) {

self.sides = sides

self.generator = generator

}

func roll() -> Int {

return Int(generator.random() \* Double(sides)) +1

}

}

如下所示， LinearCongruentialGenerator 的实例作为随机数生成器传入 Dice 的构造器

var d6 = Dice(sides: 6,generator: LinearCongruentialGenerator())

for \_ in 1...5 {

println("Random dice roll is \(d6.roll())")

}

//输出结果

//Random dice roll is 3

//Random dice roll is 5

//Random dice roll is 4

//Random dice roll is 5

//Random dice roll is 4

委托(代理)模式  
委托是一种设计模式，它允许类或结构体将一些需要它们负责的功能交由(委托)给其他的类型。  
委托模式的实现很简单： 定义协议来封装那些需要被委托的函数和方法， 使其遵循者拥有这些被委托的函数和方法。  
委托模式可以用来响应特定的动作或接收外部数据源提供的数据，而无需要知道外部数据源的类型。

下文是两个基于骰子游戏的协议：

protocol DiceGame {

var dice: Dice { get }

func play()

}

protocol DiceGameDelegate {

func gameDidStart(game: DiceGame)

func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:Int)

func gameDidEnd(game: DiceGame)

}

DiceGame 协议可以在任意含有骰子的游戏中实现， DiceGameDelegate 协议可以用来追踪DiceGame 的游戏过程。

class SnakesAndLadders: DiceGame {

let finalSquare = 25

let dic = Dice(sides: 6, generator: LinearCongruentialGenerator())

var square = 0

var board: Int[]

init() {

board = Int[](count: finalSquare + 1, repeatedValue: 0)

board[03] = +08; board[06] = +11;

borad[09] = +09; board[10] =+02

borad[14] = -10; board[19] = -11;

borad[22] = -02; board[24] = -08

}

var delegate: DiceGameDelegate?

func play() {

square = 0

delegate?.gameDidStart(self)

gameLoop: while square != finalSquare {

let diceRoll = dice.roll()

delegate?.game(self,didStartNewTurnWithDiceRoll: diceRoll)

switch square + diceRoll {

case finalSquare:

break gameLoop

case let newSquare where newSquare > finalSquare:

continue gameLoop

default:

square += diceRoll

square += board[square]

}

}

delegate?.gameDIdEnd(self)

}

}

注意：因为 delegate 并不是该游戏的必备条件， delegate 被定义为遵循 DiceGameDelegate协议的可选属性DicegameDelegate 协议提供了三个方法用来追踪游戏过程。被放置于游戏的逻辑中，即play()方法内。分别在游戏开始时，新一轮开始时，游戏结束时被调用。  
因为 delegate 是一个遵循 DiceGameDelegate 的可选属性，因此在 play()方法中使用了可选链来调用委托方法。 若 delegate 属性为 nil， 则委托调用优雅地失效。若 delegate 不为nil，则委托方法被调用

如下所示， DiceGameTracker 遵循了 DiceGameDelegate 协议

class DiceGameTracker: DiceGameDelegate {

var numberOfTurns = 0

func gameDidStart(game: DiceGame) {

numberOfTurns = 0

if game is SnakesAndLadders {

println("Started a new game of Snakes and Ladders")

}

println("The game is using a \(game.dice.sides)-sided dice")

}

func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {

++numberOfTurns

println("Rolled a \(diceRoll)")

}

func gameDidEnd(game: DiceGame) {

println("The game lasted for \(numberOfTurns) turns")

}

}  
DiceGameTracker 的运行情况，如下所示：

“let tracker = DiceGameTracker()

let game = SnakesAndLadders()

game.delegate = tracker

game.play()

// Started a new game of Snakes and Ladders

// The game is using a 6-sided dice

// Rolled a 3

// Rolled a 5

// Rolled a 4

// Rolled a 5

// The game lasted for 4 turns”

在扩展中添加协议成员

即便无法修改源代码，依然可以通过扩展(Extension)来扩充已存在类型(译者注： 类，结构体，枚举等)。扩展可以为已存在的类型添加属性，方法，下标，协议等成员。

注意： 通过扩展为已存在的类型遵循协议时，该类型的所有实例也会随之添加协议中的方法

TextRepresentable 协议含有一个 asText，如下所示：

protocol TextRepresentable {

func asText() -> String

}  
通过扩展为上一节中提到的 Dice 类遵循 TextRepresentable 协议

extension Dice: TextRepresentable {

func asText() -> String {

return "A \(sides)-sided dice"

}  
}

从现在起， Dice 类型的实例可被当作 TextRepresentable 类型：

let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())

println(d12.asText())

// 输出 "A 12-  
sided dice"let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())

println(d12.asText())

// 输出 "A 12-  
sided dice"let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())

println(d12.asText())

// 输出 "A 12-sided dice"  
SnakesAndLadders 类也可以通过扩展的方式来遵循协议：

extension SnakeAndLadders: TextRepresentable {

func asText() -> String {

return "A game of Snakes and Ladders with \(finalSquare) squar  
es"

}

}

}

println(game.asText())

// 输出 "A game of Snakes and Ladders with 25 squares”

通过延展补充协议声明

当一个类型已经实现了协议中的所有要求，却没有声明时，可以通过扩展来补充协议声明：  
struct Hamster {

var name: String

func asText() -> String {

return "A hamster named \(name)"

}

}

extension Hamster: TextRepresentabl {}

从现在起， Hamster 的实例可以作为 TextRepresentable 类型使用  
let simonTheHamster = Hamster(name: "Simon")

let somethingTextRepresentable: TextRepresentabl = simonTheHamester

println(somethingTextRepresentable.asText())

// 输出 "A hamster named Simon"  
注意：即时满足了协议的所有要求，类型也不会自动转变，因此你必须为它做出明显的协议声明

集合中的协议类型

协议类型可以被集合使用，表示集合中的元素均为协议类型：  
let things: TextRepresentable[] = [game,d12,simoTheHamster]  
如下所示， things 数组可以被直接遍历，并调用其中元素的 asText()函数：

for thing in things {

println(thing.asText())

}

// A game of Snakes and Ladders with 25 squares

// A 12-sided dice

// A hamster named Simon  
thing 被当做是 TextRepresentable 类型而不是 Dice， DiceGame， Hamster 等类型。因此能且仅能调用 asText 方法

协议的继承

协议能够继承一到多个其他协议。语法与类的继承相似，多个协议间用逗号,分隔  
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {  
// 协议定义

}

如下所示， PrettyTextRepresentable 协议继承了 TextRepresentable 协议

protocol PrettyTextRepresentable: TextRepresentable {

func asPrettyText() -> String

}

遵循 PrettyTextRepresentable 协议的同时，也需要遵循 TextRepresentable`协议。  
如下所示，用扩展为 SnakesAndLadders 遵循 PrettyTextRepresentable 协议：

extension SnakesAndLadders: PrettyTextRepresentable {

func asPrettyText() -> String {

var output = asText() + ":\n"

for index in 1...finalSquare {

switch board[index] {

case let ladder where ladder > 0:

output += "▲ "

case let snake where snake < 0:

output += "▼ "

default:

output += "○ "

}

}

return output

}

}

任意 SankesAndLadders 的实例都可以使用 asPrettyText()方法。

println(game.asPrettyText())

// A game of Snakes and Ladders with 25 squares:

// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○

协议合成

一个协议可由多个协议采用 protocol<SomeProtocol, AnotherProtocol>这样的格式进行组合，称为协议合成(protocol composition)。

举个例子：

protocol Named {

var name: String { get }

}

protocol Aged {

var age: Int { get }

}

struct Person: Named, Aged {

var name: String

var age: Int

}

func wishHappyBirthday(celebrator: protocol<Named, Aged>) {

println("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")

}

let birthdayPerson = Person(name: "Malcolm", age: 21)

wishHappyBirthday(birthdayPerson)

// 输出 "Happy birthday Malcolm - you're 21!  
Named 协议包含 String 类型的 name 属性； Aged 协议包含 Int 类型的 age 属性。 Person结构体遵循了这两个协议。

wishHappyBirthday 函数的形参 celebrator 的类型为 protocol<Named,Aged>。可以传入任意遵循这两个协议的类型的实例  
注意：协议合成并不会生成一个新协议类型，而是将多个协议合成为一个临时的协议，超出范围后立即失效。

检验协议的一致性

使用 is 检验协议一致性，使用 as 将协议类型向下转换(downcast)为的其他协议类型。检验与转换的语法和之前相同(详情查看类型检查)：  
1. is 操作符用来检查实例是否遵循了某个协议。  
2. as?返回一个可选值，当实例遵循协议时，返回该协议类型；否则返回 nil  
3. as 用以强制向下转换型。

@objc protocol HasArea {

var area: Double { get }

}

注意： @objc 用来表示协议是可选的，也可以用来表示暴露给 Objective-C 的代码，此外，@objc 型协议只对类有效，因此只能在类中检查协议的一致性。

Circle,Country,Animal 并没有一个相同的基类，所以采用 AnyObject 类型的数组来装载在他们的实例，如下所示：

let objects: AnyObject[] = [

Circle(radius: 2.0),

Country(area: 243\_610),

Animal(legs: 4)

]  
如下所示，在迭代时检查 object 数组的元素是否遵循了 HasArea 协议：

for object in objects {

if let objectWithArea = object as? HasArea {

println("Area is \(objectWithArea.area)")  
} else {

println("Something that doesn't have an area")

}

}

// Area is 12.5663708

// Area is 243610.0  
// Something that doesn't have an area  
当数组中的元素遵循 HasArea 协议时，通过 as?操作符将其可选绑定(optional binding)到objectWithArea 常量上。

可选协议要求

可选协议含有可选成员，其遵循者可以选择是否实现这些成员。在协议中使用@optional 关键字作为前缀来定义可选成员。

像 someOptionalMethod?(someArgument)一样，你可以在可选方法名称后加上?来检查该方法是否被实现。可选方法和可选属性都会返回一个可选值(optional value)，当其不可访问时， ?之后语句不会执行，并返回 nil。  
注意：可选协议只能在含有@objc 前缀的协议中生效。且@objc 的协议只能被类遵循。

Counter 类含有 CounterDataSource?类型的可选属性 dataSource，如下所示：

@objc class Counter {

var count = 0

var dataSource: CounterDataSource?

func increment() {

if let amount = dataSource?.incrementForCount?(count) {

count += amount

} else if let amount = dataSource?.fixedIncrement? {

count += amount

}

}

}  
count 属性用于存储当前的值， increment 方法用来为 count 赋值。

increment 方法通过可选链，尝试从两种可选成员中获取 count。  
由于 dataSource 可能为 nil，因此在 dataSource 后边加上了?标记来表明只在 dataSource非空时才去调用 incrementForCount`方法。

即使 dataSource 存在，但是也无法保证其是否实现了 incrementForCount 方法，因此在incrementForCount 方法后边也加有?标记。  
在调用 incrementForCount 方法后， Int 型可选值通过可选绑定(optional binding)自动拆包并赋值给常量 amount。  
当 incrementForCount 不能被调用时，尝试使用可选属性 fixedIncrement 来代替。  
ThreeSource 实现了 CounterDataSource 协议，如下所示：  
class ThreeSource: CounterDataSource {

let fixedIncrement = 3

}

使用 ThreeSource 作为数据源开实例化一个 Counter：

var counter = Counter()

counter.dataSource = ThreeSource()

for \_ in 1...4 {

counter.increment()

println(counter.count)

}

// 3

// 6

// 9

// 12

TowardsZeroSource 实现了 CounterDataSource 协议中的 incrementForCount 方法，如下所示：

class TowardsZeroSource: CounterDataSource {

func incrementForCount(count: Int) -> Int {

if count == 0 {

return 0

} else if count < 0 {

return 1

} else {

return -1

}

}

}

下边是执行的代码：

counter.count = -4

counter.dataSource = TowardsZeroSource()

**for** \_ **in** 1...5 {

counter.increment()

println(counter.count)

}

// -3

// -2

// -1

// 0

// 0

1. **泛型**

泛型代码可以让你写出根据自我需求定义、适用于任何类型的，灵活且可重用的函数和类型。它可以让你避免重复的代码，用一种清晰和抽象的方式来表达代码的意图。

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 = temporary

}  
swapTwoValues 函数主体和 swapTwoInts 函数是一样的，它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)

类型参数

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占位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占位类型（如上面的 swapTwoValues泛型函数，或一个存储单一类型的泛型集，如数组），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，一个是键，另外一个是值。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。

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，但是泛型版本的占位类型参数为 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。

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

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。

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:

let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]

if let foundIndex = findStringIndex(strings, "llama") {

println("The index of llama is \(foundIndex)")

}

// 输出 "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。

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。

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”。

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。

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，占位类型参数 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

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，在使用扩展来声明协议的采纳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

类型约束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。

下面的列子定义了一个名为 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元素。如果匹配到所有的元素， 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元素。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)。

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.")

}

// 输出 "All items match."

1. **高级运算符**

不同于 C 语言中的数值计算， Swift 的数值计算默认是不可溢出的。溢出行为会被捕获并报告为错误。你是故意的？好吧，你可以使用 Swift 为你准备的另一套默认允许溢出的数值运算符，如可溢出加&+。所有允许溢出的运算符都是以&开始的。  
自定义的结构，类和枚举，是否可以使用标准的运算符来定义操作？当然可以！在 Swift中，你可以为你创建的所有类型定制运算符的操作。  
可定制的运算符并不限于那些预设的运算符，自定义有个性的中置，前置，后置及赋值运算符，当然还有优先级和结合性。这些运算符的实现可以运用预设的运算符，也可以运用

位运算符

Swift 支持如下所有 C 语言的位运算符：

按位取反运算符

这个运算符是前置的，所以请不加任何空格地写着操作数之前。

let initialBits: UInt8 = 0b00001111

let invertedBits = ~initialBits // 等于 0b11110000

按位与运算符

按位与运算符对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为 1 时才为 1。

按位或运算

按位或运算符|比较两个数，然后返回一个新的数，这个数的每一位设置 1 的条件是两个输入数的同一位都不为 0(即任意一个为 1，或都为 1)。

let someBits: UInt8 = 0b10110010

let moreBits: UInt8 = 0b01011110

let combinedbits = someBits | moreBits // 等于 11111110

按位异或运算符

按位异或运算符^比较两个数，然后返回一个数，这个数的每个位设为 1 的条件是两个输入数的同一位不同，如果相同就设为 0。

let firstBits: UInt8 = 0b00010100

let otherBits: UInt8 = 0b00000101

let outputBits = firstBits ^ otherBits // 等于 00010001

按位左移/右移运算符  
左移运算符<<和右移运算符>>会把一个数的所有比特位按以下定义的规则向左或向右移动指定位数。  
按位左移和按位右移的效果相当把一个整数乘于或除于一个因子为 2 的整数。向左移动一个整型的比特位相当于把这个数乘于 2，向右移一位就是除于 2。

无符整型的移位操作

对无符整型的移位的效果如下：  
已经存在的比特位向左或向右移动指定的位数。被移出整型存储边界的的位数直接抛弃，移动留下的空白位用零 0 来填充。这种方法称为逻辑移位。

let pink: UInt32 = 0xCC6699

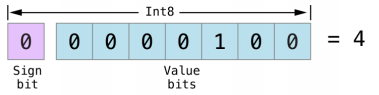
let redComponent = (pink & 0xFF0000) >> 16 // redComponent 是 0xCC,即 204

let greenComponent = (pink & 0x00FF00) >> 8 // greenComponent 是 0x66, 即 102

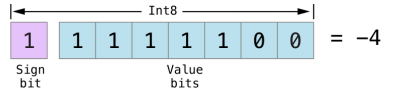
let blueComponent = pink & 0x0000FF // blueComponent 是 0x99, 即 153  
这个例子使用了一个 UInt32 的命名为 pink 的常量来存储层叠样式表 CSS 中粉色的颜色值， CSS 颜色#CC6699 在 Swift 用十六进制 0xCC6699 来表示。然后使用按位与(&)和按位右移就可以从这个颜色值中解析出红(CC)，绿(66)，蓝(99)三个部分。

有符整型的移位操作

有符整型的移位操作相对复杂得多，因为正负号也是用二进制位表示的。 (这里举的例子虽然都是 8 位的，但它的原理是通用的.)  
有符整型通过第 1 个比特位(称为符号位)来表达这个整数是正数还是负数。 0 代表正数， 1代表负数。  
其余的比特位(称为数值位)存储其实值。有符正整数和无符正整数在计算机里的存储结果是一样的。



符号位为 0，代表正数，另外 7 比特位二进制表示的实际值就刚好是 4。  
负数呢，跟正数不同。负数存储的是 2 的 n 次方减去它的绝对值， n 为数值位的位数。一个 8 比特的数有 7 个数值位，所以是 2 的 7 次方，即 128。



现在符号位为 1，代表负数， 7 个数值位要表达的二进制值是 124，即 128 - 4。

对有符整型按位右移时，使用符号位(正数为 0，负数为 1)填充空白位。

这就确保了在右移的过程中，有符整型的符号不会发生变化。这称为算术移位。  
正因为正数和负数特殊的存储方式，向右移位使它接近于 0。移位过程中保持符号会不变，负数在接近 0 的过程中一直是负数。

溢出运算符

默认情况下，当你往一个整型常量或变量赋于一个它不能承载的大数时， Swift 不会让你这么干的，它会报错。这样，在操作过大或过小的数的时候就很安全了。

当然，你有意在溢出时对有效位进行截断，你可采用溢出运算，而非错误处理。 Swfit 为整型计算提供了 5 个&符号开头的溢出运算符。  
溢出加法 &+  
溢出减法 &-  
溢出乘法 &\*  
溢出除法 &/  
溢出求余 &%

值的上溢出

下面例子使用了溢出加法&+来解剖的无符整数的上溢出

var willOverflow = UInt8.max

// willOverflow 等于 UInt8 的最大整数 255

willOverflow = willOverflow &+ 1

// 这时候 willOverflow 等于 0  
willOverflow 用 Int8 所能承载的最大值 255(二进制 11111111)，然后用&+加 1。然后UInt8 就无法表达这个新值的二进制了，也就导致了这个新值上溢出了。溢出后，新值在 UInt8 的承载范围内的那部分是 00000000，也就是 0。

值的下溢出

数值也有可能因为太小而越界。举个例子：  
UInt8 的最小值是 0(二进制为 00000000)。使用&-进行溢出减 1，就会得到二进制的11111111 即十进制的 255。

Swift 代码是这样的:

var willUnderflow = UInt8.min

// willUnderflow 等于 UInt8 的最小值 0

willUnderflow = willUnderflow &- 1

// 此时 willUnderflow 等于 255

有符整型也有类似的下溢出，有符整型所有的减法也都是对包括在符号位在内的二进制数进行二进制减法的，这在 "按位左移/右移运算符" 一节提到过。最小的有符整数是-128，即二进制的 10000000。用溢出减法减去去 1 后，变成了 01111111，即 UInt8 所能承载的最大整数 127。

var signedUnderflow = Int8.min

// signedUnderflow 等于最小的有符整数 -128

signedUnderflow = signedUnderflow &- 1

// 如今 signedUnderflow 等于 127

除零溢出

一个数除于 0 i / 0，或者对 0 求余数 i % 0，就会产生一个错误。

let x = 1

let y = x / 0  
使用它们对应的可溢出的版本的运算符&/和&%进行除 0 操作时就会得到 0 值。

let x = 1

let y = x &/ 0

// y 等于 0

优先级和结合性

运算符的优先级使得一些运算符优先于其他运算符，高优先级的运算符会先被计算。

乘法和求余拥有相同的优先级，在运算过程中，我们还需要结合性，乘法和求余运算都是左结合的。这相当于在表达式中有隐藏的括号让运算从左开始。  
2 + ((3 \* 4) % 5)  
(3 \* 4) is 12, so this is equivalent to: 3 \* 4 = 12，所以这相当于：

2 + (12 % 5)  
(12 % 5) is 2, so this is equivalent to: 12 % 5 = 2，所这又相当于

2 + 2  
计算结果为 4。

运算符函数

让已有的运算符也可以对自定义的类和结构进行运算，这称为运算符重载。

这个例子展示了如何用+让一个自定义的结构做加法。算术运算符+是一个两目运算符，因为它有两个操作数，而且它必须出现在两个操作数之间。

例子中定义了一个名为 Vector2D 的二维坐标向量 (x， y) 的结构，然后定义了让两个Vector2D 的对象相加的运算符函数。

struct Vector2D {

var x = 0.0, y = 0.0

}

@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {

return Vector2D(x: left.x + right.x, y: left.y + right.y)

}  
该运算符函数定义了一个全局的+函数，这个函数需要两个 Vector2D 类型的参数，返回值也是 Vector2D 类型。需要定义和实现一个中置运算的时候，在关键字 func 之前写上属性@infix 就可以了。

这个函数是全局的，而不是 Vector2D 结构的成员方法，所以任意两个 Vector2D 对象都可以使用这个中置运算符。

let vector = Vector2D(x: 3.0, y: 1.0)

let anotherVector = Vector2D(x: 2.0, y: 4.0)

let combinedVector = vector + anotherVector

// combinedVector 是一个新的 Vector2D, 值为 (5.0, 5.0)  
这个例子实现两个向量 (3.0， 1.0) 和 (2.0， 4.0) 相加，得到向量 (5.0， 5.0) 的过程。

前置和后置运算符

实现一个前置或后置运算符时，在定义该运算符的时候于关键字 func 之前标注 @prefix或 @postfix 属性。

@prefix func - (vector: Vector2D) -> Vector2D {

return Vector2D(x: -vector.x, y: -vector.y)

}  
这段代码为 Vector2D 类型提供了单目减运算-a， @prefix 属性表明这是个前置运算符。对于数值，单目减运算符可以把正数变负数，把负数变正数。对于 Vector2D，单目减运算将其 x 和 y 都进进行单目减运算。

let positive = Vector2D(x: 3.0, y: 4.0)

let negative = -positive

// negative 为 (-3.0, -4.0)

let alsoPositive = -negative

// alsoPositive 为 (3.0, 4.0)

组合赋值运算符

组合赋值是其他运算符和赋值运算符一起执行的运算。如+=把加运算和赋值运算组合成一个操作。实现一个组合赋值符号需要使用@assignment 属性，还需要把运算符的左参数设置成 inout，因为这个参数会在运算符函数内直接修改它的值。

@assignment func += (inout left: Vector2D, right: Vector2D) {

left = left + right

}  
因为加法运算在之前定义过了，这里无需重新定义。所以，加赋运算符函数使用已经存在的高级加法运算符函数来执行左值加右值的运算。

var original = Vector2D(x: 1.0, y: 2.0)

let vectorToAdd = Vector2D(x: 3.0, y: 4.0)

original += vectorToAdd

// original 现在为 (4.0, 6.0)

可以将 @assignment 属性和 @prefix 或 @postfix 属性起来组合，实现一个  
Vector2D 的前置运算符。

@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {

vector += Vector2D(x: 1.0, y: 1.0)

return vector

}

var toIncrement = Vector2D(x: 3.0, y: 4.0)

let afterIncrement = ++toIncrement

// toIncrement 现在是 (4.0, 5.0)

// afterIncrement 现在也是 (4.0, 5.0)

注意：默认的赋值符是不可重载的。只有组合赋值符可以重载。三目条件运算符 a？ b： c也是不可重载。

比较运算符

Swift 无所知道自定义类型是否相等或不等，因为等于或者不等于由你的代码说了算了。所以自定义的类和结构要使用比较符==或!=就需要重载。  
定义相等运算符函数跟定义其他中置运算符雷同：

@infix func == (left: Vector2D, right: Vector2D) -> Bool {

return (left.x == right.x) && (left.y == right.y)

}

@infix func != (left: Vector2D, right: Vector2D) -> Bool {

return !(left == right)

}

上述代码实现了相等运算符==来判断两个 Vector2D 对象是否有相等的值，相等的概念就是他们有相同的 x 值和相同的 y 值，我们就用这个逻辑来实现。接着使用==的结果实现了不相等运算符!=。

现在我们可以使用这两个运算符来判断两个 Vector2D 对象是否相等。

let twoThree = Vector2D(x: 2.0, y: 3.0)

let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)

if twoThree == anotherTwoThree {

println("这两个向量是相等的.")

}

// prints "这两个向量是相等的."

自定义运算符

标准的运算符不够玩，那你可以声明一些个性的运算符，但个性的运算符只能使用这些字符 / = - + \* % < >！ & | ^。 ~。  
新的运算符声明需在全局域使用 operator 关键字声明，可以声明为前置，中置或后置的。  
operator prefix +++ {}  
这段代码定义了一个新的前置运算符叫+++，此前 Swift 并不存在这个运算符。此处为了演示，我们让+++对 Vector2D 对象的操作定义为 双自增 这样一个独有的操作，这个操作使用了之前定义的加赋运算实现了自已加上自己然后返回的运算。

@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {

vector += vector

return vector

}

Vector2D 的 +++ 的实现和 ++ 的实现很接近, 唯一不同的前者是加自己, 后者是加值为(1.0, 1.0) 的向量.

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)

let afterDoubling = +++toBeDoubled

// toBeDoubled 现在是 (2.0, 8.0)

自定义中置运算符的优先级和结合性

结合性(associativity)的值可取的值有 left， right 和 none。左结合运算符跟其他优先级相同的左结合运算符写在一起时，会跟左边的操作数结合。同理，右结合运算符会跟右边的操作数结合。而非结合运算符不能跟其他相同优先级的运算符写在一起。

结合性(associativity)的值默认为 none，优先级(precedence)默认为 100。

以下例子定义了一个新的中置符+-，是左结合的 left，优先级为 140。

operator infix +- { associativity left precedence 140 }

func +- (left: Vector2D, right: Vector2D) -> Vector2D {

return Vector2D(x: left.x + right.x, y: left.y - right.y)

}

let firstVector = Vector2D(x: 1.0, y: 2.0)

let secondVector = Vector2D(x: 3.0, y: 4.0)

let plusMinusVector = firstVector +- secondVector

// plusMinusVector 此时的值为 (4.0, -2.0)

这个运算符把两个向量的 x 相加，把向量的 y 相减。因为他实际是属于加减运算，所以让它保持了和加法一样的结合性和优先级(left 和 140)。