苹果**Swift**编程语言入门教程【中文版】

目录

1 简介

Swift是供iOS和OS X应用编程的新编程语言，基于C和Objective-C，而却没有C的一些兼容约束。Swift采用了安全的编程模式和添加现代的功能来是的编程更加简单、灵活和有趣。界面则基于广受人民群众爱戴的Cocoa和Cocoa Touch框架，展示了软件开发的新方向。

Swift已经存在了多年。Apple基于已有的编译器、调试器、框架作为其基础架构。通过ARC(Automatic Reference Counting，自动引用计数)来简化内存管理。我们的框架栈则一直基于Cocoa。Objective-C进化支持了块、collection literal和模块，允许现代语言的框架无需深入即可使用。(by gashero)感谢这些基础工作，才使得可以在Apple软件开发中引入新的编程语言。

Objective-C开发者会感到Swift的似曾相识。Swift采用了Objective-C的命名参数和动态对象模型。提供了对Cocoa框架和mix-and-match的互操作性。基于这些基础，Swift引入了很多新功能和结合面向过程和面向对象的功能。

Swift对新的程序员也是友好的。他是工业级品质的系统编程语言，却又像脚本语言一样的友好。他支持playground，允许程序员实验一段Swift代码功能并立即看到结果，而无需麻烦的构建和运行一个应用。

Swift集成了现代编程语言思想，以及Apple工程文化的智慧。编译器是按照性能优化的，而语言是为开发优化的，无需互相折中。(by gashero)可以从"Hello, world"开始学起并过渡到整个系统。所有这些使得Swift成为Apple软件开发者创新的源泉。

Swift是编写iOS和OSX应用的梦幻方式，并且会持续推进新功能的引入。我们迫不及待的看到你用他来做点什么。

2 Swift入门

一个新语言的学习应该从打印"Hello, world"开始。在Swift，就是一行:

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

如果你写过C或Objective-C代码，这个语法看起来很熟悉，在Swift，这就是完整的程序了。你无需导入(import)一个单独的库供输入输出和字符串处理。全局范围的代码就是用于程序的入口，所以你无需编写一个 main() 函数。你也无需在每个语句后写分号。

这个入门会给出足够的信息教你完成一个编程任务。无需担心你还不理解一些东西，所有没解释清楚的，会在本书后续详细讲解。

作为最佳实践，可以将本章在Xcode的playground中打开。Playground允许你编辑代码并立即看到结果。

3 简单值

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

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

gashero注记
这里的常量定义类似于函数式编程语言中的变量，一次赋值后就无法修改。多多使用有益健康。

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

gashero注记
类型与变量名绑定，属于静态类型语言。有助于静态优化。与Python、JavaScript等有所区别。

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

```
let imlicitInteger = 70
let imlicitDouble = 70.0
let explicitDouble: Double = 70
```

练习
创建一个常量，类型为Float，值为4。

值永远不会隐含转换到其他类型。如果你需要转换一个值到不同类型，明确的构造一个所需类型的实例。

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

练习
尝试删除最后一行的String转换，你会得到什么错误?

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

```
let apples = 3
let oranges = 5 //by gashero
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

练习
使用 () 来包含一个浮点数计算到字符串，并包含某人的名字来问候。

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

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
var occupations = [ "Malcolm": "Captain", "Kaylee": "Mechanic", ]
occupations["Jayne"] = "Public Relations"
```

要创建一个空的数组或字典，使用初始化语法:

```
let emptyArray = String[]()
let emptyDictionary = Dictionary<String, Float>()
```

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

```
shoppingList = [] //去购物并买些东西 by gashero
```

4 控制流

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

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
if score > 50{
teamScores += 3
}
else {
teamScores += 1
}
}
teamScore
```

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

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

```
var optionalString: String? = "Hello"
optionalString == nil
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
greeting = "Hello, \(name)"
}
```

练习
改变 optionalName 为 nil 。在问候时会发生什么？添加一个 else 子句在 optionalName 为 nil 时设置一个不同的值。

如果可选值为 nil ，条件就是 false 大括号中的代码会被跳过。否则可选值未包装并赋值为一个常量，会是的未包装值的变量到代码块中。

switch 支持多种数据以及多种比较，不限制必须是整数和测试相等。

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
let vegetableComment = "Add some raisins and make ants on a log."
```

```
case "cucumber", "watercress":
let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
let vegetableComment = "Is it a spicy \(x)?"
default: //by gashero
let vegetableComment = "Everything tastes good in soup."
}
```

练习
尝试去掉 default ，看看得到什么错误。

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

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

```
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
}
}
}
```

练习
添加另一个变量来跟踪哪个种类中的数字最大，也就是最大的数字所在的。

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

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

```
while m < 100
m
```

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

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

使用 .. 构造范围忽略最高值，而用 ... 构造的范围则包含两个值。

5 函数与闭包

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

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

Note
练习
去掉 day 参数，添加一个参数包含今天的午餐选择。

使用元组(tuple)来返回多个值。

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

函数可以接受可变参数个数，收集到一个数组中。

```
func sumOf(numbers: Int...) -> Int {
var sum = 0
```

```
for number in numbers {
sum += number
}
return sum
}
sumOf()
sumOf(42, 597, 12)
```

练习
编写一个函数计算其参数的平均值。

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

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

函数是第一类型的。这意味着函数可以返回另一个函数。

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

一个函数可以接受其他函数作为参数。

```
func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
for item in list {
if condition(item) {
return true
}
}
return false
}
func lessThanTen(number: Int) -> Bool {
return number < 10
}
```

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

函数实际是闭包的特殊情况。你可以写一个闭包而无需名字，只需要放在大括号中即可。使用 in 到特定参数和主体的返回值。

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

练习
重写一个闭包来对所有奇数返回0。

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

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

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

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

6 对象与类

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

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

练习
通过 "let" 添加一个常量属性，以及添加另一个方法能接受参数。

通过在类名后加小括号来创建类的实例。使用点语法来访问实例的属性和方法。

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

这个版本的 Shape 类有些重要的东西不在：一个构造器来在创建实例时设置类。使用

init 来创建一个。

```
class NamedShape {
var numberOfSides: Int = 0
var name: String init(name: String) {
self.name = name
} //by gashero
func simpleDescription() -> String {
return "A Shape with \(numberOfSides) sides."
}
}
```

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

使用 **deinit** 来创建一个析构器，来执行对象销毁时的清理工作。

子类包括其超类的名字，以冒号分隔。在继承标准根类时无需声明，所以你可以忽略超类。

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

class Square: NamedShape {var sideLength: Double init(sideLength: Double, name: String) { self.sideLength = sideLength super.init(name: name) numberOfSides = 4 } func area() -> Double { return sideLength * sideLength } override func simpleDescription() -> String { return "A square with sides of length \(sideLength)." } } let test = Square(sideLength: 5.2, name: "my test square") test.area() test.simpleDescription()

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

属性可以有 getter 和 setter 。

```
class EquilateralTriangle: NamedShape
{
var sideLength: Double = 0.0
init(sideLength: Double, name: String){
self.sideLength = sideLength
super.init(name: name)
numberOfSides = 3
}
var perimeter: Double{
get {
return 3.0 * sideLength
}
set {
```

```
sideLength = newValue / 3.0
}
}
override func simpleDescription() -> String {
return "An equilateral triangle with sides of length \(sideLength)."
}
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

在 perimeter 的 setter 中，新的值的名字就是 newValue 。你可以提供一个在 set 之后提供一个不冲突的名字。

注意 EquilateralTriangle 的构造器有3个不同的步骤：

设置属性的值 调用超类的构造器 改变超类定义的属性的值，添加附加的工作来使用方法、getter、setter也可以在这里

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

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

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

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

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

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

7 枚举与结构

使用 enum 来创建枚举。有如类和其他命名类型，枚举可以有方法。

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

case .Ace: return "ace"
case .Jack: return "jack"
case .Queen: return "queen"
case .King: return "king"
default: return String(self.toRaw())
}
}
}
let ace = Rank.Ace //by gashero
let aceRawValue = ace.toRaw()
```

练习
编写一个函数比较两个 Rank 的值，通过比较其原始值。

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

使用 toRaw 和 fromRaw 函数可以转换原始值和枚举值。

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

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

```
enum Suit {
case Spades, Hearts, Diamonds, Clubs
func simpleDescription() -> String {
switch self {
case .Spades: return "spades"
case .Hearts: return "hearts"
case .Diamonds: return "dismonds"
case .Clubs: return "clubs"
}
}
}
let hearts = Suit.Hearts //by gashero
let heartsDescription = hearts.simpleDescription()
```

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

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

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

```
struct Card {
var rank: Rank
var suit: Suit
func simpleDescription() -> String {
return "The \(rank.simpleDescription()) of \
(suit.simpleDescription())"
}
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

练习

添加方法到 Card 类来创建一桌的纸牌，每个纸牌都有合并的rank和suit。(就是个打字员的活二，by gashero)。

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

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

```
enum ServerResponse {
case Result(String, String)
case Error(String)
}
let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")
switch success {
case let .Result(sunrise, sunset):
let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."
case let .Error(error):
let serverResponse = "Failure... \(error)"
}
```

练习
给 ServerResponse 添加第三种情况来选择。

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

8 接口和扩展

使用`protocol`来声明一个接口。

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

类、枚举和结构体都可以实现接口。

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

```
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

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

练习：写一个实现这个接口的枚举。
注意声明SimpleStructure时候mutating关键字用来标记一个会修改结构体的方法。SimpleClass的声明不需要标记任何方法因为类中的方法经常会修改类。

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

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

练习：给Double类型写一个扩展，添加absoluteValue功能。
你可以像使用其他命名类型一样使用接口名——例如，创建一个有不同类型但是都实现一个接口的对象集合。当你处理类型是接口的值时，接口外定义的方法不可用。

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

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

9 泛型

在尖括号里写一个名字来创建一个泛型函数或者类型。

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

你也可以创建泛型类、枚举和结构体。

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

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

```
func anyCommonElements <T, U where T: Sequence, U: Sequence, T.GeneratorType.Element: Equatable,
T.GeneratorType.Element == U.GeneratorType.Element> (lhs: T, rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])
```

练习：修改`anyCommonElements`函数来创建一个函数，返回一个数组，内容是两个序列的共有元素。
简单起见，你可以忽略`where`，只在冒号后面写接口或者类名。`<T: Equatable>`和`<T where T: Equatable>`是等价的。