



触控科技

The Swift Programming Language 中文版 (CocoaChina 精校)

CocoaChina

2014-6-12



Introducing Swift.

本教程系列由 [CocoaChina](#) 整理自 [CocoaChina 的 github](#) 和 [numbbbbb 的 github](#)，禁止商用，特此声明。

微博： [@CocoaChina](#)

微信：CocoaChina

整理人：齐楠楠



目录

1. 欢迎使用Swift	8
1.1. 关于 Swift	8
1.2. Swift 初见.....	9
简单值	9
控制流	11
函数和闭包	14
对象和类.....	16
枚举和结构体.....	20
接口和扩展	23
泛型.....	25
2. Swift 教程	26
2.1 基础部分	26
常量和变量	27
注释.....	30
分号.....	31
整数.....	31
浮点数	32
类型安全和类型推测	32
数值型字面量.....	33
数值型类型转换	35
类型别名.....	36
布尔值	37
元组.....	38
可选.....	40
断言.....	44
2.2. 基本运算符	45
术语.....	46
赋值运算符	47
数值运算.....	47
复合赋值.....	51
比较运算.....	51
三目条件运算.....	52
区间运算符	53
逻辑运算.....	54
2.3. 字符串和字符.....	57
字符串字面量.....	58

初始化空字符串	59
字符串可变性	59
字符串是值类型	60
使用字符(Characters)	60
计算字符数量	61
连接字符串和字符	61
字符串插值	62
比较字符串	63
大写和小写字符串	65
Unicode	65
2.4. 集合类型	68
数组	68
字典	74
集合的可变性	79
2.5. 控制流	79
For 循环	80
While 循环	84
条件语句	88
控制转移语句	97
2.6. 函数	103
定义和调用函数	104
函数的形参和返回值	105
函数形参名	108
函数类型	115
嵌套函数	119
2.7. 闭包	119
闭包表达式	120
Trailing 闭包	124
捕获 (Capture)	127
闭包是引用类型	129
2.8. 枚举	130
枚举语法 (Enumeration Syntax)	130
匹配枚举值和 Switch 语句	132
关联值 (Associated Values)	133
原始值 (Raw Values)	135
2.9. 类和结构体	137
类和结构体对比	138
结构体和枚举是值类型	141
类是引用类型	143
类和结构体的选择	145

集合(Collection)类型的赋值和拷贝行为.....	146
2.10. 属性.....	151
存储属性.....	152
计算属性.....	155
属性监视器	158
全局变量和局部变量	160
类型属性.....	160
2.11. 方法.....	165
实例方法(Instance Methods).....	165
类型方法(Type Methods)	171
2.12. 附属脚本.....	174
附属脚本语法.....	174
附属脚本用法.....	176
附属脚本选项.....	176
2.13. 继承.....	178
定义一个基类 (Base class)	179
子类生成 (Subclassing)	180
重写 (Overriding)	182
防止重写.....	186
2.14. 构造过程.....	187
存储型属性的初始赋值.....	187
定制化构造过程	189
默认构造器	192
值类型的构造器代理	193
类的继承和构造过程	196
通过闭包和函数来设置属性的默认值.....	207
2.15. 反初始化.....	209
反初始化.....	209
反初始化原理.....	209
2.16. 自动引用计数.....	212
ARC 如何工作	213
ARC 实践	214
类实例间的强引用环	215
解决实例间的强引用环.....	218
解决闭包产生的强引用环	228
2.17. 自判断链接	230
自判断链接可替代强制拆包.....	231
为自判断链接定义模型类	233
通过自判断链接调用属性	235

通过自判断链接调用方法	235
使用自判断链接调用子脚本.....	236
连接多层链接.....	237
链接自判断返回值的方法	238
2.18. 类型转换.....	239
定义一个类层次作为例子	239
检查类型.....	241
向下转型（简称下转）	242
Any 和 AnyObject 的转换.....	243
2.19 类型嵌套.....	246
类型嵌套实例.....	246
类型嵌套的引用	248
2.20 扩展.....	249
扩展语法（Extension Syntax）	249
计算型属性（Computed Properties）	250
构造器（Initializers）	251
方法（Methods）	252
下标（Subscripts）	254
嵌套类型（Nested Types）	255
2.21 协议.....	256
协议的语法	256
属性要求.....	257
方法要求.....	258
突变方法要求.....	259
协议类型.....	261
委托(代理)模式	262
在扩展中添加协议成员.....	265
通过延展补充协议声明.....	266
集合中的协议类型.....	267
协议的继承	267
协议合成.....	269
检验协议的一致性.....	270
可选协议要求.....	271
2.22 泛型.....	274
泛型所解决的问题.....	274
泛型函数.....	276
类型参数.....	277
命名类型参数.....	277
泛型类型.....	278
类型约束.....	281

关联类型.....	284
Where 语句	287
2.23 高级运算符	290
位运算符.....	291
溢出运算符	296
优先级和结合性	299
运算符函数	300
自定义运算符.....	304
3 语言附注.....	305
3.1 关于语言附注	305
如何阅读语法.....	306
3.2 词法结构	307
空白与注释	307
标识符	307
关键字	309
字面量	309
运算符	313
3.3 类型	314
类型注解.....	315
类型标识符	316
元组类型.....	316
函数类型.....	317
数组类型.....	319
可选类型.....	319
隐式解析可选类型.....	320
协议合成类型.....	321
元类型	322
类型继承子句.....	323
类型推断.....	323
3.4 表达式.....	324
前缀表达式（Prefix Expressions）	325
二元表达式（Binary Expressions）	325
赋值表达式（Assignment Operator）	328
三元条件运算符（Ternary Conditional Operator）	328
类型转换运算符（Type-Casting Operators）	329
主要表达式（Primary Expressions）	330
后缀表达式（Postfix Expressions）	336
3.5 语句	342
循环语句.....	343
分支语句.....	346

带标签的语句.....	350
3.6 声明	354
模块范围.....	355
代码块	356
引入声明.....	356
常量声明.....	357
变量声明.....	358
类型的别名声明	362
函数声明.....	363
枚举声明.....	367
结构体声明	369
协议声明.....	372
构造器声明	376
析构声明.....	377
扩展声明.....	378
附属脚本声明.....	378
运算符声明	379
3.7 特性	381
声明特性.....	382
类型特性.....	385
3.8 模式	385
通配符模式（Wildcard Pattern）	386
标识符模式（Identifier Pattern）	387
值绑定模式（Value-Binding Pattern）	387
元组模式（Tuple Pattern）	388
枚举案例模式（Enumeration Case Pattern）	389
类型转换模式（Type-Casting Patterns）	389
表达式模式（Expression Pattern）	390
3.9 泛型参数	391
泛型形参语句.....	391
泛型实参语句.....	393
3.10 语法总结.....	394
语句.....	395
泛型参数.....	396
声明.....	397
模式.....	402
特性.....	403
表达式	403
词法结构.....	406
类型.....	409

1. 欢迎使用Swift

1.1.关于 Swift

本篇译者：葛布林大帝（[博客](#)）

重要：

本教程系列是针对开发过程中的 API 或者技术的初步文档。苹果提供该文档旨在为开发者使用该技术和苹果产品上的编程接口提供帮助。这些信息可能会发生变化，依据该文档开发的软件应该使用最终操作系统软件和文档进行测试。文档的新版本可能会提供新的 API 和技术。

Swift 是一种适用于 iOS 和 OS X 应用的全新编程语言，它建立在最好的 C 和 Objective-C 语言之上，并且没有 C 语言的兼容性限制。Swift 采用安全的编程模式，增加了现代功能使编程更容易、更灵活、更有趣。Swift 以成熟且备受宠爱的 Cocoa 和 Cocoa Touch 框架为支撑，这是一个重新构想软件开发的机会。

Swift 经过了数年的酝酿，Apple 通过改进现有的编译器、调试器和框架结构奠定了 Swift 的基础，并通过 Automatic Reference Counting（ARC）简化了内存管理。Swift 的 framework stack 建立在坚实的 Foundation 和 Cocoa 基础之上，实现了现代化和彻底标准化。Objective-C 本身已经逐步发展到支持 blocks、collection literals 和 modules，使框架不间断地采用现代语言技术。基于这个基础，我们可以为今后的 Apple 软件开发引入一种新语言。

对 Objective-C 开发者来说，Swift 看起来会很熟悉。它具有 Objective-C 命名参数的易读性和动态对象模型的强大之处，提供了无缝接入现有 Cocoa 框架和与 Objective-C 混编的能力。基于这个共同点，Swift 引入了许多新的特性、统一的程序和面向对象的语言部分。

Swift 对于新程序员是友好的，作为一个具有表现力和令人愉悦的脚本语言，它是第一个获得工业品质的系统编程语言。它支持 playgrounds，一个允许程序员调试 Swift 代码并立即

看到结果的革新性功能，而无需反复编译运行程序。

Swift 从广阔的苹果工程师文化中结合了最好的现代语言思维和智慧。编译器针对性能进行了优化，语言也针对开发进行了优化，无需任何妥协。它的设计从“Hello World”扩展到整个操作系统。这一切使得 Swift 对于开发者和 Apple 是一个有意义的投资。

Swift 以极好的方式来编写的 iOS 和 OS X 应用，并将使用新特性和功能继续演进。我们对 Swift 充满期待，已经迫不及待看你能用它创造什么了。

1.2. Swift 初见

本篇译者：葛布林大帝（[博客](#)）和 @numbbbbb（[博客](#)），校对：[ChildhoodAndy](#)

通常来说，编程语言教程中的第一个程序应该在屏幕上打印“Hello, world”。在 Swift 中，可以用一行代码实现：

```
1. println("hello, world")
```

如果你写过 C 或者 Objective-C 代码，那你应该很熟悉这种形式——在 Swift 中，这行代码就是一个完整的程序。你不需要为了输入输出或者字符串处理导入一个单独的库。全局作用域中的代码会被自动当做程序的入口点，所以你也不需要 main 函数。你同样不需要在每个语句结尾写上分号。

这个教程会通过一系列编程例子来让你对 Swift 有初步了解，如果你有什么不理解的地方也不用担心——任何本章介绍的内容都会后面的章节中详细讲解。

注意：为了获得最好的体验，在 Xcode 当中使用代码预览功能。代码预览功能可以让你编辑代码并实时看到运行结果。

简单值

使用 **let** 来声明常量，使用 **var** 来声明变量。一个常量的值在编译时并不需要获取，但是你能只能为它赋值一次。也就是说你可以用常量来表示这样一个值：你只需要决定一次，但是需要使用很多次。

```
1. var myVariable = 42
2. myVariable = 50
3. let myConstant = 42
```

常量或者变量的类型必须和你赋给它们的值一样。然而，声明时类型是可选的，声明的同时赋值的话，编译器会自动推断类型。在上面的例子中，编译器推断出 `myVariable` 是一个整数（`integer`）因为它的初始值是整数。

如果初始值没有提供足够的信息（或者没有初始值），那你需要在变量后面声明类型，用冒号分割。

```
1. let implicitInteger = 70
2. let implicitDouble = 70.0
3. let explicitDouble: Double = 70
```

练习：创建一个常量，显式指定类型为 `Float` 并指定初始值为 4。

值永远不会被隐式转换为其他类型。如果你需要把一个值转换成其他类型，请显式转换。

```
1. let label = "The width is"
2. let width = 94
3. let widthLabel = label + String(width)
```

练习：删除最后一行中的 `String`，错误提示是什么？

有一种更简单的把值转换成字符串的方法：把值写到括号中，并且在括号之前写一个反斜杠。例如：

```
1. let apples = 3
2. let oranges = 5
3. let appleSummary = "I have \(apples) apples."
4. let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

练习：使用 `\()` 来把一个浮点计算转换成字符串，并加上某人的名字，和他打个招呼。

使用方括号 `[]` 来创建数组和字典，并使用下标或者键（`key`）来访问元素。

```
1. var shoppingList = ["catfish", "water", "tulips", "blue paint"]
```

```
2. shoppingList[1] = "bottle of water"
3.
4. var occupations = [
5.     "Malcolm": "Captain",
6.     "Kaylee": "Mechanic",
7. ]
8. occupations["Jayne"] = "Public Relations"
```

要创建一个空数组或者字典，使用初始化语法。

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

如果类型信息可以被推断出来，你可以用`[]`和`[:]`来创建空数组和空字典——就像你声明变量或者给函数传参数的时候一样。

```
1. shoppingList = []    // 去逛街并买点东西
```

控制流

使用 `if` 和 `switch` 来进行条件操作，使用 `for-in`、`for`、`while` 和 `do-while` 来进行循环。包裹条件和循环变量括号可以省略，但是语句体的大括号是必须的。

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

在 `if` 语句中，条件必须是一个布尔表达式——像 `if score { ... }` 这样的代码是错误的。

你可以一起使用 `if` 和 `let` 来处理值缺失的情况。有些变量的值是可选的。一个可选的值可能是一个具体的值或者是 `nil`，表示值缺失。在类型后面加一个问号来标记这个变量的值是可

选的。

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

练习: 把 `optionalName` 改成 `nil`, `greeting` 会是什么? 添加一个 `else` 语句, 当 `optionalName` 是 `nil` 时给 `greeting` 赋一个不同的值。

如果变量的可选值是 `nil`, 条件会判断为 `false`, 大括号中的代码会被跳过。如果不是 `nil`, 会将值赋给 `let` 后面的常量, 这样代码块中就可以使用这个值了。

`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。

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

练习: 删除 `default` 语句, 看看会有什么错误?

运行 `switch` 中匹配到的子句之后, 程序会退出 `switch` 语句, 并不会继续向下运行, 所以不需要在每个子句结尾写 `break`。

你可以使用 `for-in` 来遍历字典, 需要两个变量来表示每个键值对。

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

练习：添加另一个变量来记录哪种类型的数字是最大的。

使用 **while** 来重复运行一段代码直到不满足条件。循环条件可以在开头也可以在结尾。

```
1. var n = 2
2. while n < 100 {
3.     n = n * 2
4. }
5. n
6.
7. var m = 2
8. do {
9.     m = m * 2
10. } while m < 100
11. m
```

你可以在循环中使用..**来表示范围**，也可以使用传统的写法，两者是等价的：

```
1. var firstForLoop = 0
2. for i in 0..3 {
3.     firstForLoop += i
4. }
5. firstForLoop
6.
7. var secondForLoop = 0
```

```
8. for var i = 0; i < 3; ++i {
9.     secondForLoop += 1
10. }
11. secondForLoop
```

使用..`创建的范围不包含上界，如果想包含的话需要使用...`

函数和闭包

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。

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

练习：删除 `day` 参数，添加一个参数来表示今天吃了什么午饭。

使用一个元组来返回多个值。

```
1. func getGasPrices() -> (Double, Double, Double) {
2.     return (3.59, 3.69, 3.79)
3. }
4. getGasPrices()
```

函数的参数数量是可变的，用一个数组来获取它们：

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

练习：写一个计算参数平均值的函数。

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。

```
1. func returnFifteen() -> Int {
2.     var y = 10
3.     func add() {
4.         y += 5
5.     }
6.     add()
7.     return y
8. }
9. returnFifteen()
```

函数是一等公民，这意味着函数可以作为另一个函数的返回值。

```
1. func makeIncrementer() -> (Int -> Int) {
2.     func addOne(number: Int) -> Int {
3.         return 1 + number
4.     }
5.     return addOne
6. }
7. var increment = makeIncrementer()
8. increment(7)
```

函数也可以当做参数传入另一个函数。

```
1. func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
2.     for item in list {
3.         if condition(item) {
4.             return true
5.         }
6.     }
7.     return false
8. }
9. func lessThanTen(number: Int) -> Bool {
10.    return number < 10
11. }
12. var numbers = [20, 19, 7, 12]
13. hasAnyMatches(numbers, lessThanTen)
```

函数实际上是一种特殊的闭包，你可以使用`{}`来创建一个匿名闭包。使用 `in` 来分割参数并返回类型。

```
1. numbers.map({
2.     (number: Int) -> Int in
3.     let result = 3 * number
4.     return result
5. })
```

练习：重写闭包，对所有奇数返回 0。

有很多种创建闭包的方法。如果一个闭包的类型已知，比如作为一个回调函数，你可以忽略参数的类型和返回值。单个语句闭包会把它语句的值当做结果返回。

你可以通过参数位置而不是参数名字来引用参数——这个方法在非常短的闭包中非常有用。当一个闭包作为最后一个参数传给一个函数的时候，它可以直接跟在括号后面。

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

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

练习：使用 `let` 添加一个常量属性，再添加一个接收一个参数的方法。

要创建一个类的实例，在类名后面加上括号。使用点语法来访问实例的属性和方法。

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

```
3. 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。

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

注意：**self** 被用来区别实例变量。当你创建实例的时候，像传入函数参数一样给类传入构造器的参数。每个属性都需要赋值——无论是通过声明（就像 **numberOfSides**）还是通过构造器（就像 **name**）。

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。

子类的定义方法是在它们的类名后面加上父类的名字，用冒号分割。创建类的时候并不需要标准的根类，所以你可以忽略父类。

子类如果要重写父类的方法的话，需要用 **override** 标记——如果没有添加 **override** 就重写父类方法的话编译器会报错。编译器同样会检测 **override** 标记的方法是否确实在父类中。

```
1. class Square: NamedShape {
2.     var sideLength: Double
3.
4.     init(sideLength: Double, name: String) {
5.         self.sideLength = sideLength
6.         super.init(name: name)
7.         numberOfSides = 4
8.     }
```

```
9.
10.     func area() -> Double {
11.         return sideLength * sideLength
12.     }
13.
14.     override func simpleDescription() -> String {
15.         return "A square with sides of length \(sideLength)."
16.     }
17. }
18. let test = Square(sideLength: 5.2, name: "my test square")
19. test.area()
20. test.simpleDescription()
```

练习：创建 `NamedShape` 的另一个子类 `Circle`，构造器接收两个参数，一个是半径一个是名称，实现 `area` 和 `describe` 方法。

属性可以有 `getter` 和 `setter`。

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

```
22. }
23. var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle"
    )
24. triangle.perimeter
25. triangle.perimeter = 9.9
26. triangle.sideLength
```

在 `perimeter` 的 `setter` 中，新值的名字是 `newValue`。你可以在 `set` 之后显示的设置一个名字。

注意 `EquilateralTriangle` 类的构造器执行了三步：

1. 设置子类声明的属性值
2. 调用父类的构造器
3. 改变父类定义的属性值。其他的工作比如调用方法、`getters` 和 `setters` 也可以在这个阶段完成。

如果你不需要计算属性但是需要在设置一个新值之前运行一些代码，使用 `willSet` 和 `didSet`。

比如，下面的类确保三角形的边长总是和正方形的边长相同。

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

```
18. triangleAndSquare.square.sideLength
19. triangleAndSquare.triangle.sideLength
20. triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
21. triangleAndSquare.triangle.sideLength
```

类中的方法和一般的函数有一个重要的区别，函数的参数名只在函数内部使用，但是方法的参数名需要在调用的时候显式说明（除了第一个参数）。默认情况下，方法的参数名和它在方法内部的名字一样，不过你也可以定义第二个名字，这个名字被用在方法内部。

```
1. class Counter {
2.     var count: Int = 0
3.     func incrementBy(amount: Int, numberOfTimes times: Int) {
4.         count += amount * times
5.     }
6. }
7. var counter = Counter()
8. counter.incrementBy(2, numberOfTimes: 7)
```

处理变量的可选值时，你可以在操作（比如方法、属性和子脚本）之前加`?`。如果`?`之前的值是`nil`，`?`后面的东西都会被忽略，并且整个表达式返回`nil`。否则，`?`之后的东西都会被运行。在这两种情况下，整个表达式的值也是一个可选值。

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

枚举和结构体

使用 `enum` 来创建一个枚举。就像类和其他所有命名类型一样，枚举可以包含方法。

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

```
9.         case .Jack:
10.             return "jack"
11.         case .Queen:
12.             return "queen"
13.         case .King:
14.             return "king"
15.         default:
16.             return String(self.rawValue)
17.     }
18. }
19. }
20. let ace = Rank.Ace
21. let aceRawValue = ace.rawValue
```

练习：写一个函数，通过比较它们的原始值来比较两个 `Rank` 值。

在上面的例子中，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。

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

枚举的成员值是实际值，并不是原始值的另一种表达方法。实际上，如果原始值没有意义，你不需要设置。

```
1. enum Suit {
2.     case Spades, Hearts, Diamonds, Clubs
3.     func simpleDescription() -> String {
4.         switch self {
5.             case .Spades:
6.                 return "spades"
7.             case .Hearts:
8.                 return "hearts"
9.             case .Diamonds:
10.                return "diamonds"
11.             case .Clubs:
```

```
12.         return "clubs"
13.     }
14. }
15.
16. }
17. let hearts = Suit.Hearts
18. let heartsDescription = hearts.simpleDescription()
```

练习：给 `Suit` 添加一个 `color` 方法，对 `spades` 和 `clubs` 返回“black”，对 `hearts` 和 `diamonds` 返回“red”。

注意：有两种方式可以引用 `Hearts` 成员：给 `hearts` 常量赋值时，枚举成员 `Suit.Hearts` 需要用全名来引用，因为常量没有显式指定类型。在 `switch` 里，枚举成员使用缩写 `Hearts` 来引用，因为 `self` 的值已经知道是一个 `suit`。已知变量类型的情况下你可以使用缩写。

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. struct Card {
2.     var rank: Rank
3.     var suit: Suit
4.     func simpleDescription() -> String {
5.         return "The \(rank.simpleDescription()) of \(
6.             (suit.simpleDescription()))"
7.     }
8. }
9. let threeOfSpades = Card(rank: .Three, suit: .Spades)
10. let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

练习：给 `Card` 添加一个方法，创建一副完整的扑克牌并把每张牌的 `rank` 和 `suit` 对应起来。

一个枚举成员的实例可以有实例值。相同枚举成员的实例可以有不同的值。创建实例的时候传入值即可。实例值和原始值是不同的：枚举成员的原始值对于所有实例都是相同的，而且你是在定义枚举的时候设置原始值。

例如，考虑从服务器获取日出和日落的时间。服务器会返回正常结果或者错误信息。

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

练习：给 `ServerResponse` 和 `switch` 添加第三种情况。

注意：如何从 `ServerResponse` 中提取日升和日落时间。

接口和扩展

使用 `protocol` 来声明一个接口。

```
1. protocol ExampleProtocol {
2.     var simpleDescription: String { get }
3.     mutating func adjust()
4. }
```

类、枚举和结构体都可以实现接口。

```
1. class SimpleClass: ExampleProtocol {
2.     var simpleDescription: String = "A very simple class."
3.     var anotherProperty: Int = 69105
4.     func adjust() {
5.         simpleDescription += " Now 100% adjusted."
6.     }
7. }
```

```
8. var a = SimpleClass()
9. a.adjust()
10. let aDescription = a.simpleDescription
11.
12. struct SimpleStructure: ExampleProtocol {
13.     var simpleDescription: String = "A simple structure"
14.     mutating func adjust() {
15.         simpleDescription += " (adjusted) "
16.     }
17. }
18. var b = SimpleStructure()
19. b.adjust()
20. 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。

```
1. extension Int: ExampleProtocol {
2.     var simpleDescription: String {
3.         return "The number \(self)"
4.     }
5.     mutating func adjust() {
6.         self += 42
7.     }
8. }
9.
10. 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。

```
1. let protocolValue: ExampleProtocol = a
2. protocolValue.simpleDescription
3. // 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。

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。

```
1. func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {
2.     var result = ItemType[]()
3.     for i in 0..times {
4.         result += item
5.     }
6.     return result
7. }
8. repeat("knock", 4)
```

你也可以创建泛型类、枚举和结构体。

```
1. // Reimplement the Swift standard library's optional type
2. enum OptionalValue<T> {
3.     case None
4.     case Some(T)
5. }
6. var possibleInteger: OptionalValue<Int> = .None
7. 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。

```
1. func anyCommonElements <T, U where T: Sequence, U: Sequence, T.Element == U.Element> (lhs: T, rhs: U) -> Bool {
2.     for lhsItem in lhs {
3.         for rhsItem in rhs {
4.             if lhsItem == rhsItem {
```

```
5.         return true
6.     }
7. }
8. }
9.     return false
10. }
11. 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。

2. Swift 教程

2.1 基础部分

本篇译者：numbbbbb ([Github 主页](#))，lyuka ([Github 主页](#))，JaySurplus ([Github 主页](#))，敬请勘误。

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

Swift 的类型是在 C 和 Objective-C 的基础上提出的，`Int` 是整型；`Double` 和 `Float` 是浮点型；`Bool` 是布尔型；`String` 是字符串。Swift 还有两个有用的集合类型，`Array` 和 `Dictionary`，请参考[集合类型](#)。

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

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

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

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

常量和变量

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

声明常量和变量

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

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

这两行代码可以被理解为：

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

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

你可以在一行中声明多个常量或者多个变量，用逗号隔开：

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

注意：如果你的代码中有不需要改变的值，请使用 **let** 关键字将它声明为常量。只将需要改变的值声明为变量。

类型标注

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

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

```
1. var welcomeMessage: String
```

声明中的冒号代表着“是...类型”，所以这行代码可以被理解为：

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

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

`welcomeMessage` 变量现在可以被设置成任意字符串：

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

注意：一般来说你很少需要写类型标注。如果你在声明常量或者变量的时候赋了一个初始值，**Swift** 可以推断出这个常量或者变量的类型，请参考类型安全和类型推断。在上面的例子中，没有给 `welcomeMessage` 赋初始值，所以变量 `welcomeMessage` 的类型是通过一个类型标注指定的，而不是通过初始值推断的。

常量和变量的命名

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

```
1. let π = 3.14159
```

```
2. let 你好 = "你好世界"
3. let ???????? = "dogcow"
```

常量与变量名不能包含数学符号，箭头，保留的（或者非法的）**Unicode** 码位，连线与制表符。也不能以数字开头，但是可以在常量与变量名的其他地方包含数字。

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

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

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

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

与变量不同，常量的值一旦被确定就不能更改了。尝试这样做会导致编译时报错：

```
1. let languageName = "Swift"
2. languageName = "Swift++"
3. // 这会报编译时错误 - languageName 不可改变
```

输出常量和变量

你可以用 **println** 函数来输出当前常量或变量的值：

```
1. println(friendlyWelcome)
2. // 输出 "Bonjour!"
```

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

println 函数输出传入的 **String** 值：

```
1. println("This is a string")
2. // 输出 "This is a string"
```

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

Swift 用字符串插值（string interpolation）的方式把常量名或者变量名当做占位符加入到长字符串中，Swift 会用当前常量或变量的值替换这些占位符。将常量或变量名放入圆括号中，并在开括号前使用反斜杠将其转义：

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

注意：字符串插值所有可用的选项，请参考[字符串插值](#)。

注释

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

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

```
1. // 这是一个注释
```

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

```
1. /* 这是一个，
2. 多行注释 */
```

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

```
1. /* 这是第一个多行注释的开头
2. /* 这是第二个被嵌套的多行注释 */
3. 这是第一个多行注释的结尾 */
```

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

分号

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

```
1. let cat = "????"; println(cat)
2. // 输出 "????"
```

整数

整数就是没有小数部分的数字，比如 **42** 和 **-23**。整数可以是有符号（正、负、零）或者无符号（正、零）。

Swift 提供了 **8**，**16**，**32** 和 **64** 位的有符号和无符号整数类型。这些整数类型和 **C** 语言的命名方式很像，比如 **8** 位无符号整数类型是 **UInt8**，**32** 位有符号整数类型是 **Int32**。就像 **Swift** 的其他类型一样，整数类型采用大写命名法。

整数范围

你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：

```
1. let minValue = UInt8.min // minValue 为 0, 是 UInt8 类型的最小值
2. let maxValue = UInt8.max // maxValue 为 255, 是 UInt8 类型的最大值
```

Int

一般来说，你不需要专门指定整数的长度。**Swift** 提供了一个特殊的整数类型 **Int**，长度与当前平台的原生字长相同：

- 在 **32** 位平台上，**Int** 和 **Int32** 长度相同。
- 在 **64** 位平台上，**Int** 和 **Int64** 长度相同。

除非你需要特定长度的整数，一般来说使用 **Int** 就够了。这可以提高代码一致性和可复用性。即使是在 **32** 位平台上，**Int** 可以存储的整数范围也可以达到 **-2147483648~2147483647**，大多数时候这已经足够大了。

UInt

Swift 也提供了一个特殊的无符号类型 **UInt**，长度与当前平台的原生字长相同：

- 在 32 位平台上，**UInt** 和 **UInt32** 长度相同。
- 在 64 位平台上，**UInt** 和 **UInt64** 长度相同。

注意：尽量不要使用 **UInt**，除非你真的需要存储一个和当前平台原生字长相同的无符号整数。除了这种情况，最好使用 **Int**，即使你要存储的值已知是非负的。统一使用 **Int** 可以提高代码的可复用性，避免不同类型数字之间的转换，并且匹配数字的类型推测，请参考类型安全和类型推测。

浮点数

浮点数是有小数部分的数字，比如 **3.14159**，**0.1** 和 **-273.15**。

浮点类型比整数类型表示的范围更大，可以存储比 **Int** 类型更大或者更小的数字。**Swift** 提供了两种有符号浮点数类型：

- **Double** 表示 64 位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。
- **Float** 表示 32 位浮点数。精度要求不高的话可以使用此类型。

注意：**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。

类型安全和类型推测

Swift 是一个类型安全(**type safe**)的语言。类型安全的语言可以让你清楚地知道代码要处理的值的类型。如果你的代码需要一个 **String**，你绝对不可能不小心传进去一个 **Int**。

由于 **Swift** 是类型安全的，所以它会在编译你的代码时进行类型检查(**type checks**)，并把不匹配的类型标记为错误。这可以让你在开发的时候尽早发现并修复错误。

当你要处理不同类型的值时，类型检查可以帮你避免错误。然而，这并不是说你每次声明常量和变量的时候都需要显式指定类型。如果你没有显式指定类型，**Swift** 会使用类型推

测(type inference)来选择合适的类型。有了类型推测，编译器可以在编译代码的时候自动推测出表达式的类型。原理很简单，只要检查你赋的值即可。

因为有类型推测，和 C 或者 Objective-C 比起来 Swift 很少需要声明类型。常量和变量虽然需要明确类型，但是大部分工作并不需要你自己来完成。

当你声明常量或者变量并赋初值的时候类型推测非常有用。当你在声明常量或者变量的时候赋给它们一个字面量(literal value 或 literal)即可触发类型推测。（字面量就是会直接出现在你代码中的值，比如 42 和 3.14159。）

例如，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，因为你给它赋的初始值看起来像一个整数：

```
1. let meaningOfLife = 42
2. // meaningOfLife 会被推测为 Int 类型
```

同理，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：

```
1. let pi = 3.14159
2. // pi 会被推测为 Double 类型
```

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. let anotherPi = 3 + 0.14159
2. // anotherPi 会被推测为 Double 类型
```

原始值 3 没有显式声明类型，而表达式中出现了一个浮点字面量，所以表达式会被推测为 Double 类型。

数值型字面量

整数字面量可以被写作：

- 一个十进制数，没有前缀

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

下面的所有整数字面量的十进制值都是 17:

```
1. let decimalInteger = 17
2. let binaryInteger = 0b10001           // 二进制的 17
3. let octalInteger = 0o21               // 八进制的 17
4. let hexadecimalInteger = 0x11         // 十六进制的 17
```

浮点数字面量可以是十进制（没有前缀）或者是十六进制（前缀是 `0x`）。小数点两边必须有至少一个十进制数字（或者是十六进制的数字）。浮点数字面量还有一个可选的指数 (**exponent**)，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。

如果一个十进制数的指数为 **exp**，那这个数相当于基数和 10^{exp} 的乘积：

`1.25e2` 表示 1.25×10^2 ，等于 125.0。

`1.25e-2` 表示 1.25×10^{-2} ，等于 0.0125。

如果一个十六进制数的指数为 **exp**，那这个数相当于基数和 2^{exp} 的乘积：

`0xFp2` 表示 15×2^2 ，等于 60.0。

`0xFp-2` 表示 15×2^{-2} ，等于 3.75。

下面的这些浮点数字面量都等于十进制的 12.1875:

```
1. let decimalDouble = 12.1875
2. let exponentDouble = 1.21875e1
3. let hexadecimalDouble = 0xC.3p0
```

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：

```
1. let paddedDouble = 000123.456
2. let oneMillion = 1_000_000
3. let justOverOneMillion = 1_000_000.000_000_1
```

数值型类型转换

通常来讲，即使代码中的整数常量和变量已知非负，也请使用 `Int` 类型。总是使用默认的整数类型可以保证你的整数常量和变量可以直接被复用并且可以匹配整数类字面量的类型推测。只有在必要的时候才使用其他整数类型，比如要处理外部的长度明确的数据或者为了优化性能、内存占用等等。使用显式指定长度的类型可以及时发现值溢出并且可以暗示正在处理特殊数据。

整数转换

不同整数类型的变量和常量可以存储不同范围的数字。`Int8` 类型的常量或者变量可以存储的数字范围是 `-128~127`，而 `UInt8` 类型的常量或者变量能存储的数字范围是 `0~255`。如果数字超出了常量或者变量可存储的范围，编译的时候会报错：

```
1. let cannotBeNegative: UInt8 = -1
2. // UInt8 类型不能存储负数，所以会报错
3. let tooBig: Int8 = Int8.max + 1
4. // Int8 类型不能存储超过最大值的数，所以会报错
```

由于每中整数类型都可以存储不同范围的值，所以你必须根据不同情况选择性使用数值型类型转换。这种选择性使用的方式，可以预防隐式转换的错误并让你的代码中的类型转换意图变得清晰。

要将一种数字类型转换成另一种，你要用当前值来初始化一个期望类型的新数字，这个数字的类型就是你的目标类型。在下面的例子中，常量 `twoThousand` 是 `UInt16` 类型，然而常量 `one` 是 `UInt8` 类型。它们不能直接相加，因为它们类型不同。所以要调用 `UInt16(one)` 来创建一个新的 `UInt16` 数字并用 `one` 的值来初始化，然后使用这个新数字来计算：

```
1. let twoThousand: UInt16 = 2_000
2. let one: UInt8 = 1
3. let twoThousandAndOne = twoThousand + UInt16(one)
```

现在两个数字的类型都是 `UInt16`，可以进行相加。目标常量 `twoThousandAndOne` 的类型被推测为 `UInt16`，因为它是两个 `UInt16` 值的和。

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

整数和浮点数转换

整数和浮点数的转换必须显式指定类型:

```
1. let three = 3
2. let pointOneFourOneFiveNine = 0.14159
3. let pi = Double(three) + pointOneFourOneFiveNine
4. // pi 等于 3.14159, 所以被推测为 Double 类型
```

这个例子中, 常量 `three` 的值被用来创建一个 `Double` 类型的值, 所以加号两边的数类型相同。如果不进行转换, 两者无法相加。

浮点数到整数的反向转换同样行, 整数类型可以用 `Double` 或者 `Float` 类型来初始化:

```
1. let integerPi = Int(pi)
2. // integerPi 等于 3, 所以被推测为 Int 类型
```

当用这种方式来初始化一个新的整数值时, 浮点值会被截断。也就是说 `4.75` 会变成 `4`, `-3.9` 会变成 `-3`。

注意: 结合数字类常量和变量不同于结合数字类字面量。字面量 `3` 可以直接和字面量 `0.14159` 相加, 因为数字字面量本身没有明确的类型。它们的类型只在编译器需要求值的时候被推测。

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。

当你想要给现有类型起一个更有意义的名字时, 类型别名非常有用。假设你正在处理特定长度的外部资源的数据:

```
1. typealias AudioSample = UInt16
```

定义了一个类型别名之后，你可以在任何使用原始名的地方使用别名：

```
1. var maxAmplitudeFound = AudioSample.min
2. // maxAmplitudeFound 现在是 0
```

本例中，`AudioSample` 被定义为 `UInt16` 的一个别名。因为它是别名，`AudioSample.min` 实际上是 `UInt16.min`，所以会给 `maxAmplitudeFound` 赋一个初值 `0`。

布尔值

Swift 有一个基本的布尔(**Boolean**)类型，叫做 **Bool**。布尔值指逻辑上的(logical)，因为它们只能是真或者假。**Swift** 有两个布尔常量，`true` 和 `false`：

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

`orangesAreOrange` 和 `turnipsAreDelicious` 的类型会被推测为 **Bool**，因为它们的初值是布尔字面量。就像之前提到的 **Int** 和 **Double** 一样，如果你创建变量的时候给它们赋值 `true` 或者 `false`，那你不需要将常量或者变量声明为 **Bool** 类型。初始化常量或者变量的时候如果所赋的值类型已知，就可以触发类型推测，这让 **Swift** 代码更加简洁并且可读性更高。

当你编写条件语句比如 `if` 语句的时候，布尔值非常有用：

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

条件语句，例如 `if`，请参考控制流。

如果你在需要使用 **Bool** 类型的地方使用了非布尔值，**Swift** 的类型安全机制会报错。下面的例子会报告一个编译时错误：

```
1. let i = 1
2. if i {
3.     // 这个例子不会通过编译，会报错
4. }
```

然而，下面的例子是合法的：

```
1. let i = 1
2. if i == 1 {
3.     // 这个例子会编译成功
4. }
```

`i == 1` 的比较结果是 `Bool` 类型，所以第二个例子可以通过类型检查。类似 `i == 1` 这样的比较，请参考基本操作符。

和 `Swift` 中的其他类型安全的例子一样，这个方法可以避免错误并保证这块代码的意图总是清晰的。

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。

下面这个例子中，`(404, "Not Found")` 是一个描述 `HTTP` 状态码（`HTTP status code`）的元组。`HTTP` 状态码是当你请求网页的时候 `web` 服务器返回的一个特殊值。如果你请求的网页不存在就会返回一个 `404 Not Found` 状态码。

```
1. let http404Error = (404, "Not Found")
2. // http404Error 的类型是 (Int, String)，值是 (404, "Not Found")
```

`(404, "Not Found")` 元组把一个 `Int` 值和一个 `String` 值组合起来表示 `HTTP` 状态码的两个部分：一个数字和一个人类可读的描述。这个元组可以被描述为“一个类型为 `(Int, String)` 的元组”。

你可以把任意顺序的类型组合成一个元组，这个元组可以包含所有类型。只要你想，你可以创建一个类型为 `(Int, Int, Int)` 或者 `(String, Bool)` 或者其他任何你想要的组合的元组。

你可以将一个元组的内容分解（`decompose`）成单独的常量和变量，然后你就可以正常使用它们了：

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

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：

```
1. let (justTheStatusCode, _) = http404Error
2. println("The status code is \(justTheStatusCode)")
3. // 输出 "The status code is 404"
```

此外，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：

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

你可以在定义元组的时候给单个元素命名：

```
1. 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：

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

作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个`(Int, String)`元组来描述是否获取成功。和只能返回一个类型的值比较起来，一个包含两个不同类型值的元组可以让函数的返回信息更有用。请参考[\[函数参数与返回值 06_Functions.html#Function_Parameters_and_Return_Values\]](#)。

注意：元组在临时组织值的时候很有用，但是并不适合创建复杂的数据结构。如果你的数据结构并不是临时使用，请使用类或者结构体而不是元组。请参考类和结构体。

可选

使用可选（optionals）来处理值可能缺失的情况。可选表示：

- 有值，等于 `x`

或者

没有值

注意：C 和 Objective-C 中并没有可选这个概念。最接近的是 Objective-C 中的一个特性，一个方法要不返回一个对象要不返回 `nil`，`nil` 表示“缺少一个合法的对象”。然而，这只对对象起作用——对于结构体，基本的 C 类型或者枚举类型不起作用。对于这些类型，Objective-C 方法一般会返回一个特殊值（比如 `NSNotFound`）来暗示值缺失。这种方法假设方法的调用者知道并记得对特殊值进行判断。然而，Swift 的可选可以让你暗示任意类型的值缺失，并不需要一个特殊值。

来看一个例子。Swift 的 `String` 类型有一个叫做 `toInt` 的方法，作用是将一个 `String` 值转换成一个 `Int` 值。然而，并不是所有的字符串都可以转换成一个整数。字符串 `"123"` 可以被转换成数字 `123`，但是字符串 `"hello, world"` 不行。

下面的例子使用 `toInt` 方法来尝试将一个 `String` 转换成 `Int`：

```
1. let possibleNumber = "123"
2. let convertedNumber = possibleNumber.toInt()
3. // convertedNumber 被推测为类型 "Int?", 或者类型 "optional Int"
```

因为 `toInt` 方法可能会失败，所以它返回一个可选的（optional）`Int`，而不是一个 `Int`。一个可选的 `Int` 被写作 `Int?` 而不是 `Int`。问号暗示包含的值是可选，也就是说可能包含 `Int` 值也可

能不包含值。（不能包含其他任何值比如 `Bool` 值或者 `String` 值。只能是 `Int` 或者什么都没有。）

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。这个感叹号表示“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`）：

```
1. if convertedNumber {
2.     println("\(possibleNumber) has an integer value of \(convertedNumber!)" )
3. } else {
4.     println("\(possibleNumber) could not be converted to an integer")
5. }
6. // 输出 "123 has an integer value of 123"
```

更多关于 `if` 语句的内容，请参考控制流。

注意：使用!来获取一个不存在的可选值会导致运行时错误。使用!来强制解析值之前，一定要确定可选包含一个非 `nil` 的值。

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。可选绑定可以用在 `if` 和 `while` 语句中来对可选的值进行判断并把值赋给一个常量或者变量。`if` 和 `while` 语句，请参考控制流。

像下面这样在 `if` 语句中写一个可选绑定：

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

你可以像上面这样使用可选绑定来重写 `possibleNumber` 这个例子：

```
1. if let actualNumber = possibleNumber.toInt() {
2.     println("\(possibleNumber) has an integer value of \(actualNumber)
3. } else {
4.     println("\(possibleNumber) could not be converted to an integer")
5. }
6. // 输出 "123 has an integer value of 123"
```

这段代码可以被理解为：

“如果 `possibleNumber.toInt` 返回的可选 `Int` 包含一个值，创建一个叫做 `actualNumber` 的新常量并将可选包含的值赋给它。”

如果转换成功，`actualNumber` 常量可以在 `if` 语句的第一个分支中使用。它已经被可选包含的值初始化过，所以不需要再使用 `!` 后缀来获取它的值。在这个例子中，`actualNumber` 只被用来输出转换结果。

你可以在可选绑定中使用常量和变量。如果你想在 `if` 语句的第一个分支中操作 `actualNumber` 的值，你可以改成 `if var actualNumber`，这样可选包含的值就会被赋给一个变量而非常量。

nil

你可以给可选变量赋值为 `nil` 来表示它没有值：

```
1. var serverResponseCode: Int? = 404
2. // serverResponseCode 包含一个可选的 Int 值 404
3. serverResponseCode = nil
4. // 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`：

```
1. var surveyAnswer: String?
2. // surveyAnswer 被自动设置为 nil
```

注意：Swift 的 nil 和 Objective-C 中的 nil 并不一样。在 Objective-C 中，nil 是一个指向不存在对象的指针。在 Swift 中，nil 不是指针——它是一个确定的值，用来表示值缺失。任何类型的可选都可以被设置为 nil，不只是对象类型。

隐式解析可选

如上所述，可选暗示了常量或者变量可以“没有值”。可选可以通过 if 语句来判断是否有值，如果有值的话可以通过可选绑定来解析值。

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。

当可选被第一次赋值之后就可以确定之后一直有值的时候，隐式解析可选非常有用。隐式解析可选主要被用在 Swift 中类的构造过程中，请参考类实例之间的循环强引用。

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：

```
1. let possibleString: String? = "An optional string."
2. println(possibleString!) // 需要惊叹号来获取值
3. // 输出 "An optional string."
4.
5. let assumedString: String! = "An implicitly unwrapped optional string."
6. println(assumedString) // 不需要感叹号
7. // 输出 "An implicitly unwrapped optional string."
```

你可以把隐式解析可选当做一个可以自动解析的可选。你要做的只是声明的时候把感叹号放到类型的结尾，而不是每次取值的可选名字的结尾。

注意：如果你在隐式解析可选没有值的时候尝试取值，会触发运行时错误。和你在没有值的普通可选后面加一个惊叹号一样。

你仍然可以把隐式解析可选当做普通可选来判断它是否包含值：

```
1. if assumedString {
2.     println(assumedString)
3. }
4. // 输出 "An implicitly unwrapped optional string."
```

你也可以在可选绑定中使用隐式解析可选来检查并解析它的值：

```
1. if let definiteString = assumedString {
2.     println(definiteString)
3. }
4. // 输出 "An implicitly unwrapped optional string."
```

注意：如果一个变量之后可能变成 `nil` 的话请不要使用隐式解析可选。如果你需要在变量的生命周期中判断是否是 `nil` 的话，请使用普通可选类型。

断言

可选可以让你判断值是否存在，你可以在代码中优雅地处理值缺失的情况。然而，在某些情况下，如果值缺失或者值并不满足特定的条件，你的代码可能并不需要继续执行。这时，你可以在你的代码中触发一个断言（**assertion**）来结束代码运行并通过调试来找到值缺失的原因。

使用断言进行调试

断言会在运行时判断一个逻辑条件是否为 `true`。从字面意思来说，断言“断言”一个条件是否为真。你可以使用断言来保证在运行其他代码之前，某些重要的条件已经被满足。如果条件判断为 `true`，代码运行会继续进行；如果条件判断为 `false`，代码运行停止，你的应用被终止。

如果你的代码在调试环境下触发了一个断言，比如你在 **Xcode** 中构建并运行一个应用，你可以清楚地看到不合法的状态发生在哪里并检查断言被触发时你的应用的状态。此外，断言允许你附加一条调试信息。

你可以使用全局 `assert` 函数来写一个断言。向 `assert` 函数传入一个结果为 `true` 或者 `false` 的表达式以及一条信息，当表达式为 `false` 的时候这条信息会被显示：

```
1. let age = -3
2. assert(age >= 0, "A person's age cannot be less than zero")
3. // 因为 age < 0, 所以断言会触发
```

在这个例子中，只有 `age >= 0` 为 `true` 的时候代码运行才会继续，也就是说，当 `age` 的值非负的时候。如果 `age` 的值是负数，就像代码中那样，`age >= 0` 为 `false`，断言被触发，结束应用。

断言信息不能使用字符串插值。断言信息可以省略，就像这样：

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

何时使用断言

当条件可能为假时使用断言，但是最终一定要保证条件为真，这样你的代码才能继续运行。断言的适用情景：

- 整数的附属脚本索引被传入一个自定义附属脚本实现，但是下标索引值可能太小或者太大。
- 需要给函数传入一个值，但是非法的值可能导致函数不能正常执行。
- 一个可选值现在是 `nil`，但是后面的代码运行需要一个非 `nil` 值。

请参考附属脚本和[函数](#)。

注意：断言可能导致你的应用终止运行，所以你应当仔细设计你的代码来让非法条件不会出现。然而，在你的应用发布之前，有时候非法条件可能出现，这时使用断言可以快速发现问题。

2.2. 基本运算符

本篇译者：stanzhai ([Github 主页](#))，敬请勘误。

运算符是检查, 改变, 合并值的特殊符号或短语. 例如, 加号 `+` 把计算两个数的和(如 `let i = 1 + 2`). 复杂些的运行算包括逻辑与`&&`(如 `if enteredDoorCode && passedRetinaScan`), 还有自增运算符 `++i` 这样让自身加一的便捷运算.

Swift 支持大部分标准 **C** 语言的运算符, 且改进许多特性来减少常规编码错误. 如, 赋值符 `=` 不返回值, 以防止错把等号 `==` 写成赋值号 `=` 而导致 **Bug**. 数值运算符(`+`, `-`, `*`, `/`, `%`等)会检测并不允许值溢出, 以此来避免保存变量时由于变量大于或小于其类型所能承载的范围时导致的异常结果. 当然允许你选择使用 **Swift** 的溢出运算符来玩溢出. 具体使用请移步溢出运算符.

区别于 **C** 语言, 在 **Swift** 中你可以对浮点数进行取余运算(`%`), 还提供了 **C** 语言没有的表达两数之间的值的区间运算符, (`a..b` 和 `a...b`), 这方便我们表达一个区间内的数值.

本章节只描述了 **Swift** 中的基本运算符, 高级运算符包含了高级运算符, 及如何自定义运算符, 及如何进行自定义类型的运算符重载.

术语

运算符有一目, 双目和三目运算符.

一目运算符对单一操作对象操作, 如 `-a`.

一目运算符分前置符和后置运算符, 前置运算符需紧排操作对象之前, 如 `!b`, 后置运算符需紧跟操作对象之后, 如 `i++`,

双目运算符操作两个操作对象, 如 `2 + 3`. 是中置的, 因为它们出现在两个操作对象之间.

三目运算符操作三个操作对象, 和 **C** 语言一样, **Swift** 只有一个三目运算符, 就是三目条件运算符 `a ? b : c`.

受运算符影响的值叫操作数, 在表达式 `1 + 2` 中, 加号 `+` 是双目运算符, 它的两个操作数 是值 `1` 和 `2`.

赋值运算符

赋值运算 `a = b`, 表示用 `b` 的值来初始化或更新 `a` 的值.

```
1. let b = 10
2. var a = 5
3. a = b
4. // a 现在等于 10
```

如果赋值的右边是一个多元组, 它的元素可以马上被分解多个变量或变量

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

与 C 语言和 Objective-C 不同, Swift 的赋值操作并不返回任何值. 所以以下代码是错误的:

```
1. if x = y {
2.     // 此句错误, 因为 x = y 并不返回任何值
3. }
```

这个特性使得你无法把 `==` 错写成 `=` 了, 由于 `if x = y` 是错误代码, Swift 从底层帮你避免了这些代码错误.

数值运算

Swift 让所有数值类型都支持了基本的四则运算:

加法 +

减法 -

乘法 *

除法 /

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

与 C 语言和 Objective-C 不同的是, Swift 默认不允许在数值运算中出现溢出情况. 但你可以使用 Swift 的溢出运算符来达到你有目的的溢出, (如 `a &+ b`). 详情请移步: 溢出运算符.

加法操作 `+` 也用于字符串的拼接:

```
1. "hello, " + "world" // 等于 "hello, world"
```

两个字符类型或一个字符类型和一个字符串类型, 相加会生成一个新的字符串类型:

```
1. let dog: Character = "d"
2. let cow: Character = "c"
3. let dogCow = dog + cow
4. // 译者注: 原来的引号内是很可爱的小狗和小牛, 但 win os 下不支持表情字符, 所以改成了普通字符
5. // dogCow 现在是 "dc"
```

详细请点击 [字符,字符串的拼接](#).

求余运算

求余运算 `a % b` 是计算 `b` 的多少倍刚刚好可以容入 `a`, 多出来的那部分叫余数.

注意: 求余运算(`%`)在其他语言也叫取模运算. 然而严格说来, 我们看该运算符对负数的操作结果, 求余 比 取模 更合适些.

我们来谈谈取余是怎么回事, 计算 `9 % 4`, 你先计算出 `4` 的多少倍会刚好可以容入 `9` 中.

2 倍, 非常好, 那余数是 1 (用`*`标出)

1	2	3	4	5	6	7	8	9
4				4				1*

在 Swift 中这么来表达

```
1. 9 % 4 // 等于 1
```

为了得到 `a % b` 的结果, `%`计算了以下等式, 并输出余数作为结果:

```
1. a = (b × 倍数) + 余数
```

当倍数取最大值的时候, 就会刚好可以容入 a 中.

把 9 和 4 代入等式中, 我们得 1:

```
1. 9 = (4 × 2) + 1
```

同样的方法, 我们来计算 $-9 \% 4$:

```
1. -9 % 4 // 等于 -1
```

把 -9 和 4 代入等式, -2 是取到的最大整数:

```
1. -9 = (4 × -2) + -1
```

余数是 -1.

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.

浮点数求余计算

不同于 C 和 Objective-C, Swift 中是可以对浮点数进行求余的.

```
1. 8 % 2.5 // 等于 0.5
```

这个例子中, 8 除以 2.5 等于 3 余 0.5, 所以结果是 0.5.

自增和自增运算

和 C 一样, Swift 也提供了方便对变量本身加 1 或减 1 的自增 `++` 和自减 `--` 的运算符. 其操作对象可以是整形和浮点型.

```
1. var i = 0
2. ++i // 现在 i = 1
```

每调用一次 `++i`, i 的值就会加 1. 实际上, `++i` 是 $i = i + 1$ 的简写, 而 `--i` 是 $i = i - 1$ 的简写.

`++` 和 `--` 既是前置又是后置运算. `++i`, `i++`, `--i` 和 `i--` 都是有效的写法.

我们需要注意的是这些运算符修改了 `i` 后有一个返回值。如果你只想修改 `i` 的值，那你可以忽略这个返回值。但如果你想使用返回值，你就需要留意前置和后置操作的返回值是不同的。

当 `++` 前置的时候，先自增再返回。

当 `++` 后置的时候，先返回再自增。

不懂？我们看例子：

```
1. var a = 0
2. let b = ++a // a 和 b 现在都是 1
3. let c = a++ // a 现在 2, 但 c 是 a 自增前的值 1
```

上述例子，`let b = ++a`，先把 `a` 加 1 了再返回 `a` 的值。所以 `a` 和 `b` 都是新值 1。

而 `let c = a++`，是先返回了 `a` 的值，然后 `a` 才加 1。所以 `c` 得到了 `a` 的旧值 1，而 `a` 加 1 后变成 2。

除非你需要使用 `i++` 的特性，不然推荐你使用 `++i` 和 `--i`，因为先修改后返回这样的行为更符合我们的逻辑。

单目负号

数值的正负号可以使用前缀 `-` (即单目负号) 来切换：

```
1. let three = 3
2. let minusThree = -three // minusThree 等于 -3
3. let plusThree = -minusThree // plusThree 等于 3, 或 "负负 3"
```

单目负号写在操作数之前，中间没有空格。

单目正号

单目正号 `+` 不做任何改变地返回操作数的值。

```
1. let minusSix = -6
2. let alsoMinusSix = +minusSix // alsoMinusSix 等于 -6
```

虽然单目 `+` 做无用功, 但当你在使用单目负号来表达负数时, 你可以使用单目正号来表达正数, 如此你的代码会具有对称美.

复合赋值

如同强大的 C 语言, Swift 也提供把其他运算符和赋值运算 `=` 组合的复合赋值运算符, 加赋运算 `+=` 是其中一个例子:

```
1. var a = 1
2. a += 2 // a 现在是 3
```

表达式 `a += 2` 是 `a = a + 2` 的简写, 一个加赋运算就把加法和赋值两件事完成了.

注意: 复合赋值运算没有返回值, `let b = a += 2` 这类代码是错误的. 这不同于上面提到的自增和自减运算符.

表达式里有复合运算符的完整列表.

比较运算

所有标准 C 中的比较运算都可以在 Swift 中使用.

等于 `a == b`

不等于 `a != b`

大于 `a > b`

小于 `a < b`

大于等于 `a >= b`

小于等于 `a <= b`

注意: Swift 也提供恒等 `===` 和不恒等 `!==` 这两个比较符来判断两个对象是否引用同一个对象实例. 更多细节在 类与结构.

每个比较运算都返回了一个标识表达式是否成立的布尔值:

```
1. 1 == 1 // true, 因为 1 等于 1
```

```
2. 2 != 1    // true, 因为 2 不等于 1
3. 2 > 1     // true, 因为 2 大于 1
4. 1 < 2     // true, 因为 1 小于 2
5. 1 >= 1    // true, 因为 1 大于等于 1
6. 2 <= 1    // false, 因为 2 并不小于等于 1
```

比较运算多用于条件语句, 如 `if` 条件:

```
1. let name = "world"
2. if name == "world" {
3.     println("hello, world")
4. } else {
5.     println("对不起, \(name), 我不认识你!")
6. }
7. // 输出 "hello, world", 因为 `name` 就是等于 "world"
```

关于 `if` 语句, 请看 控制流.

三目条件运算

三目条件运算的特殊在于它是三个操作数的运算符, 它的原型是 `问题 ? 答案1 : 答案2`. 它简洁地表达根据 `问题` 成立与否作出二选一的操作. 如果 `问题` 成立, 返回 `答案1` 的结果; 如果不成立, 返回 `答案2` 的结果.

使用三目条件运算简化了以下代码:

```
1. if question: {
2.     answer1
3. }
4. else {
5.     answer2
6. }
```

这里有个计算表格行高的例子. 如果有表头, 那行高应比内容高度要高出 50 像素; 如果没有表头, 只需高出 20 像素.

```
1. let contentHeight = 40
2. let hasHeader = true
```

```
3. let rowHeight = contentHeight + (hasHeader ? 50 : 20)
4. // rowHeight 现在是 90
```

这样写会比下面的代码简洁:

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

第一段代码例子使用了三目条件运算, 所以一行代码就能让我们得到正确答案. 这比第二段代码简洁得多, 无需将 `rowHeight` 定义成变量, 因为它的值无需在 `if` 语句中改变.

三目条件运算提供有效率且便捷的方式来表达二选一的选择. 需要注意的事, 过度使用三目条件运算就会由简洁的代码变成难懂的代码. 我们应避免在一个组合语句使用多个三目条件运算符.

区间运算符

Swift 提供了两个方便表达一个区间的值的运算符.

闭区间运算符

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:

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

关于 `for-in`, 请看 控制流.

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.

半闭区间的实用性在于当你使用一个 `0` 始的列表(如数组)时, 非常方便地从 `0` 数到列表的长度.

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. let count = names.count
3. for i in 0..
```

注意: 数组有 `4` 个元素, 但 `0.. 只数到 3 (最后一个元素的下标), 因为它是半闭区间. 关于数组, 请查阅 数组.`

逻辑运算

逻辑运算的操作对象是逻辑布尔值. **Swift** 支持基于 **C** 语言的三个标准逻辑运算.

逻辑非 `!a`

逻辑与 `a && b`

逻辑或 `a || b`

逻辑非

逻辑非运算 `!a` 对一个布尔值取反, 使得 `true` 变 `false`, `false` 变 `true`.

它是一个前置运算符, 需出现在操作数之前, 且不加空格. 读作 非 `a`, 然后我们看以下例子:

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

`if !allowedEntry` 语句可以读作 "如果 非 `allowedEntry`.", 接下一行代码只有在如果 "非 `allowedEntry`" 为 `true`, 即 `allowedEntry` 为 `false` 时被执行.

在示例代码中, 小心地选择布尔常量或变量有助于代码的可读性, 并且避免使用双重逻辑非运算, 或混乱的逻辑语句.

逻辑与

逻辑与 `a && b` 表达了只有 `a` 和 `b` 的值都为 `true` 时, 整个表达式的值才会是 `true`.

只要任意一个值为 `false`, 整个表达式的值就为 `false`. 事实上, 如果第一个值为 `false`, 那么是不去计算第二个值的, 因为它已经不可能影响整个表达式的结果了. 这被称做 "短路计算".

以下例子, 只有两个值都为值的时候才允许进入:

```
1. let enteredDoorCode = true
2. let passedRetinaScan = false
3. if enteredDoorCode && passedRetinaScan {
4.     println("Welcome!")
5. } else {
6.     println("ACCESS DENIED")
7. }
8. // 输出 "ACCESS DENIED"
```

逻辑或

逻辑或 `a || b` 是一个由两个连续的 `|` 组成的中置运算符. 它表示了两个逻辑表达式的其中一个为 `true`, 整个表达式就为 `true`.

同逻辑与运算类似, 逻辑或也是"短路计算"的, 当左端的表达式为 `true` 时, 将不计算右边的表达式了, 因为它不可能改变整个表达式的值了.

以下示例代码中，第一个布尔值 `hasDoorKey` 为 `false`，但第二个值 `knowsOverridePassword` 为 `true`，所以整个表达是 `true`，于是允许进入：

```
1. let hasDoorKey = false
2. let knowsOverridePassword = true
3. if hasDoorKey || knowsOverridePassword {
4.     println("Welcome!")
5. } else {
6.     println("ACCESS DENIED")
7. }
8. // 输出 "Welcome!"
```

组合逻辑

我们可以组合多个逻辑运算来表达一个复合逻辑：

```
1. if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverrideP
   assword {
2.     println("Welcome!")
3. } else {
4.     println("ACCESS DENIED")
5. }
6. // 输出 "Welcome!"
```

这个例子使用了含多个 `&&` 和 `||` 的复合逻辑。但无论怎样，`&&` 和 `||` 始终只能操作两个值。所以这实际是三个简单逻辑连续操作的结果。我们来解读一下：

如果我们输入了正确的密码并通过了视网膜扫描；或者我们有一把有效的钥匙；又或者我们知道紧急情况下重置的密码，我们就能把门打开进入。

前两种情况，我们都不满足，所以前两个简单逻辑的结果是 `false`，但是我们是知道紧急情况下重置的密码的，所以整个复杂表达式的值还是 `true`。

使用括号来明确优先级

为了一个复杂表达式更容易读懂，在合适的地方使用括号来明确优先级是很有效的，虽然它并非必要的。在上个关于门的权限的例子中，我们给第一个部分加个括号，使用它看起来逻辑更明确。

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

这括号使得前两个值被看成整个逻辑表达中独立的一个部分。虽然有括号和没括号的输出结果是一样的，但对于读代码的人来说有括号的代码更清晰。

可读性比简洁性更重要，请在可以让你代码变清晰地地方加个括号吧！

2.3. 字符串和字符

本篇译者：wh1100717 ([Github 主页](#))，敬请勘误。

`String` 是一个有序的字符集合，例如 `"hello, world"`, `"albatross"`。Swift 字符串通过 `String` 类型来表示，也可以表示为 `Character` 类型值的集合。

Swift 的 `String` 和 `Character` 类型提供了一个快速的，兼容 `Unicode` 的方式来处理代码中的文本信息。创建和操作字符串的语法与 C 的操作方式相似，轻量并且易读。字符串连接操作只需要简单地通过 `+` 号将两个字符串相连即可。与 Swift 中其他值一样，能否更改字符串的值，取决于其被定义为常量还是变量。

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

`String` 也可以用来在长字符串中插入常量、变量、字面量和表达式，这一过程称为字符串插值。这使得创建用于展示、存储和打印的自定义字符串变得轻松自如。

注意：Swift 的 `String` 类型与 `Foundation NSString` 类进行了无缝桥接。如果您利用 `Cocoa` 或 `Cocoa Touch` 中的 `Foundation` 框架进行工作，整个 `NSString API` 都可以调用您创建的任意 `String` 类型的值，除了本章介绍的 `String` 特性。您也可以在任何要求传

入 `NSString` 实例作为参数的 API 中使用 `String` 类型的值进行替换。

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

字符串字面量

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

字符串字面量可以用于为常量和变量提供初始值。

```
1. let someString = "Some string literal value"
```

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

字符串字面量可以包含以下特殊字符:

1. 转义特殊字符 `\0` (空字符)、`\\` (反斜线)、`\t` (水平制表符)、`\n` (换行符)、`\r` (回车符)、`\"` (双引号)、`\'` (单引号)。
2. 单字节 Unicode 标量, 写成 `\xnn`, 其中 `nn` 为两位十六进制数。
3. 双字节 Unicode 标量, 写成 `\unnnn`, 其中 `nnnn` 为四位十六进制数。
4. 四字节 Unicode 标量, 写成 `\Unnnnnnnn`, 其中 `nnnnnnnn` 为八位十六进制数。

下面的代码为各种特殊字符的使用示例。

`wiseWords` 常量包含了两个转移特殊字符 (双括号); `dollarSign`、`blackHeart` 和 `sparklingHeart` 常量演示了三种不同格式的 Unicode 标量:

```
1. let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
2. // "Imagination is more important than knowledge" - Einstein
3. let dollarSign = "\x24"           // $, Unicode scalar U+0024
4. let blackHeart = "\u2665"        // ♥, Unicode scalar U+2665
5. let sparklingHeart = "\U0001F496" // ????, Unicode scalar U+1F496
```

初始化空字符串

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

```
1. var emptyString = ""           // empty string literal
2. var anotherEmptyString = String() // initializer syntax
3. // 这两个字符串都为空，并且两者等价
```

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

```
1. if emptyString.isEmpty {
2.     println("Nothing to see here")
3. }
4. // 打印 "Nothing to see here"
```

字符串可变性

您可以通过将一个特定字符串分配给一个变量（对其进行修改）或者常量（保证其不会被修改）来指定该字符串是否可以被修改：

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

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

字符串是值类型

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

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

Swift 默认字符串拷贝的方式保证了在函数/方法中传递的是字符串的值，其明确您独有该字符串的值，无论它来自哪里。您可以放心您传递的字符串本身不会被更改，除非是您自己更改它。

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

使用字符(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。您可利用 `for-in` 循环来遍历字符串中的每一个字符：

```
1. for character in "Dog!???" {
2.     println(character)
3. }
4. // D
5. // o
6. // g
7. // !
8. // ???
```

`for-in` 循环在 [For Loops](#) 中进行了详细描述。

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

```
1. 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。

```
1. let unusualMenagerie = "Koala ????, Snail ????, Penguin ????, Dromedary ????"
2. println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
3. // prints "unusualMenagerie has 40 characters"
```

注意：

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

2.另外需要注意的是通过 `countElements` 返回的字符数量并不总是与包含相同字符的 `NSString` 的 `length` 属性相同。`NSString` 的 `length` 属性是基于利用 `UTF-16` 表示的十六位 `code units` 数目，而不是基于 `Unicode` 字符。为了解决这个问题，`NSString` 的 `length` 属性在被 `Swift` 的 `String` 值访问时会被称为 `utf16count`。

连接字符串和字符

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

```
1. let string1 = "hello"
2. let string2 = " there"
3. let character1: Character = "!"
4. let character2: Character = "?"
5.
6. let stringPlusCharacter = string1 + character1           // 等于 "hello!"
7. let stringPlusString = string1 + string2                 // 等于 "hello there"
8. let characterPlusString = character1 + string1           // 等于 "!hello"
9. let characterPlusCharacter = character1 + character2     // 等于 "!"
```

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

```
1. var instruction = "look over"
2. instruction += string2
3. // instruction 现在等于 "look over there"
4.
5. var welcome = "good morning"
6. welcome += character1
7. // welcome 现在等于 "good morning!"
```

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

字符串插值

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

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

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

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

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

比较字符串

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

字符串相等

如果两个字符串以同一顺序包含完全相同的字符，则认为两者字符串相等：

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

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

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

您可以利用 `hasPrefix` 方法使用 `romeoAndJuliet` 数组来计算话剧中第一幕的场景数：

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

同样，可使用 `hasSuffix` 方法来计算发生在 **Capulet** 公馆和 **Lawrence** 牢房内以及周围的场景数。

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

大写和小写字符串

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

```
1. let normal = "Could you help me, please?"
2. let shouty = normal.uppercaseString
3. // shouty 值为 "COULD YOU HELP ME, PLEASE?"
4. let whispered = normal.lowercaseString
5. // whispered 值为 "could you help me, please?"
```

Unicode

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

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

Unicode 术语(Terminology)

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

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


字符串的 Unicode 表示

Swift 提供了几种不同的方式来访问字符串的 **Unicode** 表示。

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

另外，能够以其他三种 `Unicode` 兼容的方式访问字符串的值：

1. UTF-8 代码单元集合 (利用字符串的 `utf8` 属性进行访问)
2. UTF-16 代码单元集合 (利用字符串的 `utf16` 属性进行访问)
3. 21 位的 `Unicode` 标量值集合 (利用字符串的 `unicodeScalars` 属性进行访问)

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

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

UTF-8

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

```
1. for codeUnit in dogString.utf8 {
2.     print("\(codeUnit) ")
3. }
4. print("\n")
5. // 68 111 103 33 240 159 144 182
```

上面的例子中，前四个 10 进制 `codeUnit` 值 (68, 111, 103, 33) 代表了字符 `D o g` 和 `!`，他们的 `UTF-8` 表示与其 `ASCII` 表示相同。后四个 `codeUnit` 值 (240, 159, 144, 182) 是 `DOG FACE` 字符的 4 位 `UTF-8` 表示。

UTF-16

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

UTF-16 表示:

```
1. for codeUnit in dogString.utf16 {
2.     print("\(codeUnit) ")
3. }
4. print("\n")
5. // 68 111 103 33 55357 56374
```

同样，前四个 `codeUnit` 值 (68, 111, 103, 33) 代表了字符 `D o g` 和 `!`，他们的 UTF-16 `code units` 值和 UTF-8 表示完全相同。

第五和第六个 `codeUnit` 值 (55357 and 56374) 是 `DOG FACE` 字符的 UTF-16 表示。第一个值为 `U+D83D` (十进制值为 55357)，第二个值为 `U+DC36` (十进制值为 56374)。

Unicode 标量 (Unicode Scalars)

您可以通过遍历字符串的 `unicodeScalars` 属性来访问它的 Unicode 标量表示。其为 `UnicodeScalarView` 类型的属性，`UnicodeScalarView` 是 `UnicodeScalar` 的集合。一个 `Unicode Scalar` 是任意的 21 位的 Unicode 代码点。

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

```
1. for scalar in dogString.unicodeScalars {
2.     print("\(scalar.value) ")
3. }
4. print("\n")
5. // 68 111 103 33 128054
```

同样，前四个 `UnicodeScalar` 值 (68, 111, 103, 33) 再次代表了字符 `D, o, g` 和 `!`。第五个也是最后一个 `UnicodeScalar` 的值属性为 128054，是一个十六进制 `1F436` 的十进制表示。其等同于 `DOG FACE` 字符的 Unicode 标量 `U+1F436`。

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

```
1. for scalar in dogString.unicodeScalars {
2.     println("\(scalar) ")
3. }
```

```
4. // D
5. // o
6. // g
7. // !
8. // ????
```

2.4. 集合类型

本篇译者：bzszy ([Github 主页](#))，敬请勘误。

Swift 语言提供经典的数组和字典两种集合类型来存储集合数据。数组用来按顺序存储相同类型的数据。字典虽然无序存储相同类型数据值但是需要由独有的标识符引用和寻址（就是键值对）。

Swift 语言里的数组和字典中存储的数据值类型必须明确。这意味着我们不能把不正确的数据类型插入其中。同时这也说明我们完全可以对获取出的值类型非常自信。**Swift** 对显式类型集合的使用确保了我们的代码对工作所需要的类型非常清楚，也让我们在开发中可以早早地找到任何的类型不匹配错误。

注意：**Swift** 的数组结构在被声明成常量和变量或者被传入函数与方法中时会相对于其他类型展现出不同的特性。获取更多信息请参见 [see Mutability of Collections and Assignment and Copy Behavior for Collection Types](#)。（集合的可变性与集合在赋值和复制中的行为章节）

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。

Swift 数组对存储数据有具体要求。不同于 **Objective-C** 的 `NSArray` 和 `NSMutableArray` 类，他们可以存储任何类型的实例而且不提供他们返回对象的任何本质信息。在 **Swift** 中，数据值在被存储进入某个数组之前类型必须明确，方法是通过显式的类型标注或类型推断，而且不是必须是 `class` 类型。例如：如果我们创建了一个 `Int` 值类型的数组，我们不能往其

中插入任何不是 `Int` 类型的数据。Swift 中的数组是类型安全的，并且它们中包含的类型必须明确。

数组的简单语法

写 Swift 数组应该遵循像 `Array<SomeType>` 这样的形式，其中 `someType` 是这个数组中唯一允许存在的数据类型。我们也可以使用像 `SomeType[]` 这样的简单语法。尽管两种形式在功能上是一样的，但是我们推荐较短的那种，而且在本文中都会使用这种形式来使用数组。

数组构造语句

我们可以使用字面语句来进行数组构造，这是一种用一个或者多个数值构造数组的简单方法。字面语句是一系列由逗号分割并由方括号包含的数值。 `[value 1, value 2, value 3]`。

下面这个例子创建了一个叫做 `shoppingList` 并且存储字符串的数组：

```
1. var shoppingList: String[] = ["Eggs", "Milk"]
2. // shoppingList 已经被构造并且拥有两个初始项。
```

`shoppingList` 变量被声明为“字符串值类型的数组”，记作 `String[]`。因为这个数组被规定只有 `String` 一种数据结构，所以只有 `String` 类型可以在其中被存取。在这里，`shoppinglist` 数组由两个 `String` 值（"Eggs" 和 "Milk"）构造，并且由字面语句定义。

注意：`Shoppinglist` 数组被声明为变量（`var` 关键字创建）而不是常量（`let` 创建）是因为以后可能会有更多的数据项被插入其中。

在这个例子中，字面语句仅仅包含两个 `String` 值。匹配了该数组的变量声明（只能包含 `String` 的数组），所以这个字面语句的分配过程就是允许用两个初始项来构造 `shoppinglist`。

由于 Swift 的类型推断机制，当我们用字面语句构造只拥有相同类型值数组的时候，我们不必把数组的类型定义清楚。`shoppinglist` 的构造也可以这样写：

```
1. var shoppingList = ["Eggs", "Milk"]
```

因为所有字面语句中的值都是相同的类型，Swift 可以推断出 `String[]` 是 `shoppinglist` 中变量的正确类型。

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。

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

```
1. if shoppingList.isEmpty {
2.     println("The shopping list is empty.")
3. } else {
4.     println("The shopping list is not empty.")
5. }
6. // 打印 "The shopping list is not empty." (shoppinglist 不是空的)
```

也可以使用 `append` 方法在数组后面添加新的数据项：

```
1. shoppingList.append("Flour")
2. // shoppingList 现在有 3 个数据项，有人在摊煎饼
```

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：

```
1. shoppingList += "Baking Powder"
2. // shoppingList 现在有四项了
```

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。

```
1. shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
2. // shoppingList 现在有 7 项了
```

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：

```
1. var firstItem = shoppingList[0]
2. // 第一项是 "Eggs"
```

注意第一项在数组中的索引值是 0 而不是 1。Swift 中的数组索引总是从零开始。

我们也可以用下标来改变某个已有索引值对应的数据值：

```
1. shoppingList[0] = "Six eggs"
2. // 其中的第一项现在是 "Six eggs" 而不是 "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。下面的例子把"Chocolate Spread", "Cheese", 和"Butter"替换为"Bananas"和 "Apples"：

```
1. shoppingList[4...6] = ["Bananas", "Apples"]
2. // shoppingList 现在有六项
```

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。我们可以使用索引值和数组的 `count` 属性进行比较来在使用某个索引之前先检验是否有效。除了当 `count` 等于 0 时（说明这是个空数组），最大索引值一直是 `count - 1`，因为数组都是零起索引。

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：

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

这次 `insert` 函数调用把值为"Maple Syrup"的新数据项插入 `shopping` 列表的最开始位置，并且使用 0 作为索引值。

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）：

```
1. let mapleSyrup = shoppingList.removeAtIndex(0)
```

```
2. //索引值为 0 的数据项被移除
3. // shoppingList 现在只有 6 项, 而且不包括 Maple Syrup
4. // 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, 所以现在索引值为 0 的数据项的值再次等于 "Six eggs":

```
1. firstItem = shoppingList[0]
2. // firstItem 现在等于 "Six eggs"
```

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:

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

数组的遍历

我们可以使用 `for-in` 循环来遍历所有数组中的数据项:

```
1. for item in shoppingList {
2.     println(item)
3. }
4. // Six eggs
5. // Milk
6. // Flour
7. // Baking Powder
8. // Bananas
```

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:

```
1. for (index, value) in enumerate(shoppingList) {
2.     println("Item \(index + 1): \(value)")
}
```

```
3. }
4. // Item 1: Six eggs
5. // Item 2: Milk
6. // Item 3: Flour
7. // Item 4: Baking Powder
8. // Item 5: Bananas
```

更多关于 **for-in** 循环的介绍请参见 **for** 循环。

创建并且构造一个数组

我们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：

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

注意 **someInts** 被设置为一个 **Int[]** 构造函数的输出所以它的变量类型被定义为 **Int[]**。

除此之外，如果代码上下文中提供了类型信息， 例如一个函数参数或者一个已经定义好类型的常量或者变量，我们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）：

```
1. someInts.append(3)
2. // someInts 现在包含一个 INT 值
3. someInts = []
4. // someInts 现在是空数组，但是仍然是 Int[] 类型的。
```

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：

```
1. var threeDoubles = Double[](count: 3, repeatedValue: 0.0)
2. // threeDoubles 是一种 Double[] 数组，等于 [0.0, 0.0, 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：

```
1. var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
2. // anotherThreeDoubles is inferred as Double[], and equals [2.5, 2.5, 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：

```
1. var sixDoubles = threeDoubles + anotherThreeDoubles
2. // sixDoubles 被推断为 Double[], 等于 [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

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。和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符（键）访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

Swift 的字典使用时需要具体规定可以存储键和值类型。不同于 Objective-C 的 `NSDictionary` 和 `NSMutableDictionary` 类可以使用任何类型的对象来作键和值并且不提供任何关于这些对象的本质信息。在 Swift 中，在某个特定字典中可以存储的键和值必须提前定义清楚，方法是通过显性类型标注或者类型推断。

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。

`KeyType` 的唯一限制就是可哈希的，这样可以保证它是独一无二的，所有的 Swift 基本类型（例如 `String`，`Int`，`Double` 和 `Bool`）都是默认可哈希的，并且所有这些类型都可以在字典中当做键使用。未关联值的枚举成员（参见枚举）也是默认可哈希的。

字典字面语句

我们可以使用字典字面语句来构造字典，他们和我们刚才介绍过的数组字面语句拥有相似语法。一个字典字面语句是一个定义拥有一个或者多个键值对的字典集合的简单语句。

一个键值对是一个 **key** 和一个 **value** 的结合体。在字典字面语句中，每一个键值对的键和值都由冒号分割。这些键值对构成一个列表，其中这些键值对由方括号包含并且由逗号分割：

```
1. [key 1: value 1, key 2: value 2, key 3: value 3]
```

下面的例子创建了一个存储国际机场名称的字典。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：

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

airports 字典被定义为一种 `Dictionary<String, String>`，它意味着这个字典的键和值都是 `String` 类型。

注意：**airports** 字典被声明为变量（用 `var` 关键字）而不是常量（`let` 关键字）因为后来更多的机场信息会被添加到这个示例字典中。

airports 字典使用字典字面语句初始化，包含两个键值对。第一对的键是 **TYO**，值是 **Tokyo**。第二对的键是 **DUB**，值是 **Dublin**。

这个字典语句包含了两个 `String: String` 类型的键值对。他们对应 **airports** 变量声明的类型（一个只有 `String` 键和 `String` 值的字典）所以这个字典字面语句是构造两个初始数据项的 **airport** 字典。

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：

```
1. var airports = ["TYO": "Tokyo", "DUB": "Dublin"]
```

因为这个语句中所有的键和值都分别是相同的数据类型，**Swift** 可以推断出 `Dictionary<String, String>` 是 **airports** 字典的正确类型。

读取和修改字典

我们可以通过字典的方法和属性来读取和修改字典，或者使用下标语法。和数组一样，我们

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：

```
1. println("The dictionary of airports contains \(airports.count) items."
)
2. // 打印 "The dictionary of airports contains 2 items." (这个字典有两个数据项)
```

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：

```
1. airports["LHR"] = "London"
2. // airports 字典现在有三个数据项
```

我们也可以使用下标语法来改变特定键对应的值：

```
1. airports["LHR"] = "London Heathrow"
2. // "LHR"对应的值 被改为 "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。就像上面所示的示例，`updateValue(forKey:)` 方法在这个键不存在对应值的时候设置值或者在存在时更新已存在的值。和上面的下标方法不一样，这个方法返回更新值之前的原值。这样方便我们检查更新是否成功。

`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`。

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

```
1. if let airportName = airports["DUB"] {
    println("The name of the airport is \(airportName).")
} else {
    println("That airport is not in the airports dictionary.")
}
```

```
5. }
6. // 打印 "The name of the airport is Dublin INTernation." (机场的名字是都柏林国际)
```

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：

```
1. airports["APL"] = "Apple Internation"
2. // "Apple Internation"不是真的 APL 机场, 删除它
3. airports["APL"] = nil
4. // 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`：

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

```
1. for (airportCode, airportName) in airports {
2.     prINTln("\(airportCode): \(airportName)")
3. }
4. // TYO: Tokyo
5. // LHR: London Heathrow
```

`for-in` 循环请参见 `For` 循环。

我们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：

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

如果我们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：

```
1. let airportCodes = Array(airports.keys)
2. // airportCodes is ["TYO", "LHR"]
3.
4. let airportNames = Array(airports.values)
5. // airportNames is ["Tokyo", "London Heathrow"]
```

注意：**Swift** 的字典类型是无序集合类型。其中字典键，值，键值对在遍历的时候会重新排列，而且其中顺序是不固定的。

创建一个空字典

我们可以像数组一样使用构造语法创建一个空字典：

```
1. var namesOfIntegers = Dictionary<Int, String>()
2. // namesOfIntegers 是一个空的 Dictionary<Int, String>
```

这个例子创建了一个 **Int, String** 类型的空字典来储存英语对整数的命名。他的键是 **Int** 型，值是 **String** 型。

如果上下文已经提供了信息类型，我们可以使用空字典字面语句来创建一个空字典，记作`[:]`（中括号中放一个冒号）：

```
1. namesOfIntegers[16] = "sixteen"
```

```
2. // namesOfIntegers 现在包含一个键值对
3. namesOfIntegers = [:]
4. // namesOfIntegers 又成为了一个 Int, String 类型的空字典
```

注意：在后台，**Swift** 的数组和字典都是由泛型集合来实现的，想了解更多泛型和集合信息请参见泛型。

集合的可变性

数组和字典都是在单个集合中存储可变值。如果我们创建一个数组或者字典并且把它分配成一个变量，这个集合将会是可变的。这意味着我们可以在创建之后添加更多或移除已存在的数据项来改变这个集合的大小。与此相反，如果我们把数组或字典分配成常量，那么他就是不可变的，它的大小不能被改变。

对字典来说，不可变性也意味着我们不能替换其中任何现有键所对应的值。不可变字典的内容在被首次设定之后不能更改。不可变行对数组来说有一点不同，当然我们不能试着改变任何不可变数组的大小，但是我们可以重新设定相对现存索引所对应的值。这使得 **Swift** 数组在大小被固定的时候依然可以做的很棒。

Swift 数组的可变性行为同时影响了数组实例如何被分配和修改，想获取更多信息，请参见 **Assignment and Copy Behavior for Collection Types**。

注意：在我们不需要改变数组大小的时候创建不可变数组是很好的习惯。如此 **Swift** 编译器可以优化我们创建的集合。

2.5. 控制流

本篇译者：numbbbbb ([Github 主页](#))，敬请勘误。

Swift 提供了类似 **C** 语言的流程控制结构，包括可以多次执行任务的 **for** 和 **while** 循环，基于特定条件选择执行不同代码分支的 **if** 和 **switch** 语句，还有控制流程跳转到其他代码的 **break** 和 **continue** 语句。

除了 C 里面传统的 `for` 条件递增循环，Swift 还增加了 `for-in` 循环，用来更简单地遍历数组(array)，字典(dictionary)，范围(range)，字符串(string)和其他序列类型。

Swift 的 `switch` 语句比 C 语言中更加强大。在 C 语言中，如果某个 `case` 不小心漏写了 `break`，这个 `case` 就会“掉入”下一个 `case`，Swift 无需写 `break`，所以不会发生这种“掉入”的情况。`Case` 还可以匹配更多的类型模式，包括范围(range)匹配，元组(tuple)和特定类型的描述。`switch case` 语句中匹配的值可以是由 `case` 体内部临时的常量或者变量决定，也可以由 `where` 分句描述更复杂的匹配条件。

For 循环

`for` 循环用来按照指定的次数多次执行一系列语句。Swift 提供两种 `for` 循环形式：

`for-in` 用来遍历一个范围(range)，队列(sequence)，集合(collection)，系列(progression)里面所有的元素执行一系列语句。

`for` 条件递增语句(for-condition-increment)，用来重复执行一系列语句直到特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

For-In

你可以使用 `for-in` 循环来遍历一个集合里面的所有元素，例如由数字表示的范围、数组中的元素、字符串中的字符。

下面的例子用来输出乘 5 乘法表前面一部分内容：

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

例子中用来进行遍历的元素是一组使用闭区间操作符(...)表示的从 1 到 5 的闭区间数字。`index` 被赋值为闭区间范围中的第一个数字(1),然后循环中的语句被执行一次。在本例中,这个循环只包含一个语句,用来输出当前 `index` 值所对应的乘 5 乘法表结果。该语句执行后,`index` 的值被更新为闭区间范围中的第二个数字(2),之后 `println` 方法会再执行一次。整个过程会进行到闭区间范围结尾为止。

上面的例子中,`index` 是一个每次循环遍历开始时被自动赋值的常量。这种情况下,`index` 在使用前不需要声明,只需要将它包含在循环的声明中,就可以对其进行隐式声明,而无需使用 `let` 关键字声明。

注意: `index` 常量只存在于循环的生命周期里。如果你想在循环完成后访问 `index` 的值,又或者想让 `index` 成为一个变量而不是常量,你必须在循环之前自己进行声明。

如果你不需要知道范围内每一项的值,你可以使用下划线 (`_`) 替代变量名来忽略对值的访问:

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

这个例子计算 `base` 这个数的 `power` 次幂(本例中,是 3 的 10 次幂),从 1 (3 的 0 次幂)开始做 3 的乘法,进行 10 次,使用 0 到 9 的半闭区间循环。这个计算并不需要知道每一次循环中计数器具体的值,只需要执行了正确的循环次数即可。下划线符号 `_` (替代循环中的变量)能够忽略具体的值,并且不提供循环遍历时对值的访问。

使用 `for-in` 遍历一个数组所有元素:

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. for name in names {
3.     println("Hello, \(name)!")
}
```

```
4. }  
5. // Hello, Anna!  
6. // Hello, Alex!  
7. // Hello, Brian!  
8. // Hello, Jack!
```

你也可以通过遍历一个字典来访问它的键值对(key-value pairs)。遍历字典时，字典的每项元素会以 (key, value) 元组的形式返回，你可以在 **for-in** 循环中使用显式的常量名称来解读 (key, value) 元组。下面的例子中，字典的键(key)解读为常量 **animalName**，字典的值会被解读为常量 **legCount**：

```
1. let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]  
2. for (animalName, legCount) in numberOfLegs {  
3.     println("\(animalName)s have \(legCount) legs")  
4. }  
5. // spiders have 8 legs  
6. // ants have 6 legs  
7. // cats have 4 legs
```

字典元素的遍历顺序和插入顺序可能不同，字典的内容在内部是无序的，所以遍历元素时不能保证顺序。更多数组和字典相关内容，查看集合类型章节。

除了数组和字典，你也可以使用 **for-in** 循环来遍历字符串中的字符：

```
1. for character in "Hello" {  
2.     println(character)  
3. }  
4. // H  
5. // e  
6. // l  
7. // l  
8. // o
```

For 条件递增 (for-condition-increment)

除了 **for-in** 循环，Swift 提供使用条件判断和递增方法的标准 C 样式 **for** 循环：

```
1. for var index = 0; index < 3; ++index {
```

```
2.     println("index is \(index)")
3. }
4. // index is 0
5. // index is 1
6. // index is 2
```

下面是一般情况下这种循环方式的格式：

```
1. for initialization; condition; increment {
2.     statements
3. }
```

和 C 语言中一样，分号将循环的定义分为 3 个部分，不同的是，Swift 不需要使用圆括号将“initialization; condition; increment”包括起来。

这个循环执行流程如下：

- 1、循环首次启动时，初始化表达式（**initialization expression**）被调用一次，用来初始化循环所需的所有常量和变量。
- 2、条件表达式（**condition expression**）被调用，如果表达式调用结果为 **false**，循环结束，继续执行 **for** 循环关闭大括号(**}**)之后的代码。如果表达式调用结果为 **true**，则会执行大括号内部的代码（**statements**）。
- 3、执行所有语句（**statements**）之后，执行递增表达式（**increment expression**）。通常会增加或减少计数器的值，或者根据语句（**statements**）输出来修改某一个初始化的变量。当递增表达式运行完成后，重复执行第 2 步，条件表达式会再次执行。

上述描述和循环格式等同于：

```
1. initialization
2. while condition {
3.     statements
4.     increment
5. }
```

在初始化表达式中声明的常量和变量(比如 `var index = 0`)只在 **for** 循环的生命周期里有效。如果想在循环结束后访问 `index` 的值，你必须要在循环生命周期开始前声明 `index`。

```
1. var index: Int
2. for index = 0; index < 3; ++index {
3.     println("index is \(index)")
4. }
5. // index is 0
6. // index is 1
7. // index is 2
8. println("The loop statements were executed \(index) times")
9. // prints "The loop statements were executed 3 times"
```

注意 `index` 在循环结束后最终的值是 `3` 而不是 `2`。最后一次调用递增表达式 `++index` 会将 `index` 设置为 `3`，从而导致 `index < 3` 条件为 `false`，并终止循环。

While 循环

While 循环运行一系列语句直到条件变成 `false`。这类循环适合使用在第一次迭代前迭代次数未知的情况下。**Swift** 提供两种 **while** 循环形式：

- **while** 循环，每次在循环开始时计算条件是否符合；
- **do-while** 循环，每次在循环结束时计算条件是否符合。

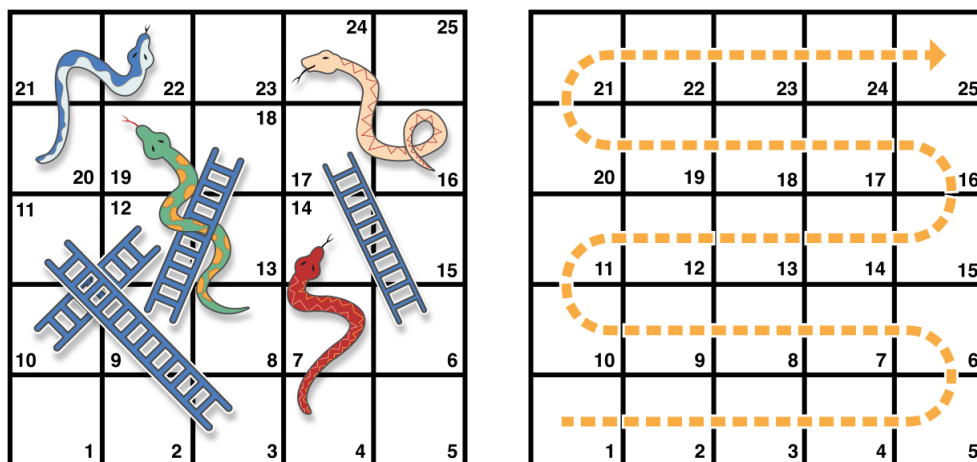
While

While 循环从计算单一条件开始。如果条件为 `true`，会重复运行一系列语句，直到条件变为 `false`。

下面是一般情况下 **while** 循环格式：

```
1. while condition {
2.     statements
3. }
```

下面的例子来玩一个叫做 蛇和梯子 的小游戏（也叫做 滑道和梯子）：



游戏的规则如下：

- 游戏盘面包括 25 个方格，游戏目标是达到或者超过第 25 个方格；
- 每一轮，你通过掷一个 6 边的骰子来确定你移动方块的步数，移动的路线由上图中横向的虚线所示；
- 如果在某轮结束，你移动到了梯子的底部，可以顺着梯子爬上去；
- 如果在某轮结束，你移动到了蛇的头部，你会顺着蛇的身体滑下去。

游戏盘面可以使用一个 `Int` 数组来表达。数组的长度由一个 `finalSquare` 常量储存，用来初始化数组和检测最终胜利条件。游戏盘面由 26 个 `Int 0` 值初始化，而不是 25 个（由 0 到 25，一共 26 个）：

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

一些方块被设置成有蛇或者梯子的指定值。梯子底部的方块是一个正值，是你可以向上移动，蛇头处的方块是一个负值，会让你向下移动：

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

3 号方块是梯子的底部，会让你向上移动到 11 号方格，我们使用 `board[03]` 等于 `+08` 来表示（11 和 3 之间的差值）。使用一元加运算符（`+`）是为了和一元减运算符（`-`）对称，为了让盘面代码整齐，小于 10 的数字都使用 0 补齐（这些风格上的调整都不是必须的，只是为了让代码看起来更加整洁）。

玩家由左下角编号为 0 的方格开始游戏。一般来说玩家第一次掷骰子后才会进入游戏盘面：

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

本例中使用了最简单的方法来模拟掷骰子。`diceRoll` 的值并不是一个随机数，而是以 0 为初始值，之后每一次 `while` 循环，`diceRoll` 的值使用前置自增操作符(`++i`)来自增 1，然后检测是否超出了最大值。`++diceRoll` 调用完成后，返回值等于 `diceRoll` 自增后的值。任何时候如果 `diceRoll` 的值等于 7 时，就超过了骰子的最大值，会被重置为 1。所以 `diceRoll` 的取值顺序会一直是 1, 2, 3, 4, 5, 6, 1, 2。

掷完骰子后，玩家向前移动 `diceRoll` 个方格，如果玩家移动超过了第 25 个方格，这个时候游戏结束，相应的，代码会在 `square` 增加 `board[square]` 的值向前或向后移动（遇到了梯子或者蛇）之前，检测 `square` 的值是否小于 `board` 的 `count` 属性。

如果没有这个检测（`square < board.count`），`board[square]` 可能会越界访问 `board` 数组，导致错误。例如如果 `square` 等于 26，代码会去尝试访问 `board[26]`，超过数组的长度。

当本轮 `while` 循环运行完毕，会再检测循环条件是否需要再运行一次循环。如果玩家移动到或者超过第 25 个方格，循环条件结果为 `false`，此时游戏结束。

`while` 循环比较适合本例中的这种情况，因为在 `while` 循环开始时，我们并不知道游戏的长度或者循环的次数，只有在达成指定条件时循环才会结束。

Do-While

while 循环的另外一种形式是 **do-while**，它和 **while** 的区别是在判断循环条件之前，先执行一次循环的代码块，然后重复循环直到条件为 **false**。

下面是一般情况下 **do-while** 循环的格式：

```
1. do {  
2.     statements  
3. } while condition
```

还是蛇和梯子的游戏，使用 **do-while** 循环来替代 **while** 循环。**finalSquare**，**board**，**square** 和 **diceRoll** 的值初始化同 **while** 循环一样：

```
1. let finalSquare = 25  
2. var board = Int[](count: finalSquare + 1, repeatedValue: 0)  
3. board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02  
4. board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08  
5. var square = 0  
6. var diceRoll = 0
```

do-while 的循环版本，循环中第一步就需要去检测是否在梯子或者蛇的方块上。没有梯子会让玩家直接上到第 25 个方格，所以玩家不会通过梯子直接赢得游戏。这样在循环开始时先检测是否踩在梯子或者蛇上是安全的。

游戏开始时，玩家在第 0 个方格上，**board[0]** 一直等于 0，不会有什么影响：

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

检测完玩家是否踩在梯子或者蛇上之后，开始掷骰子，然后玩家向前移动 `diceRoll` 个方格，本轮循环结束。

循环条件（`while square < finalSquare`）和 `while` 方式相同，但是只会在循环结束后进行计算。在这个游戏中，`do-while` 表现得比 `while` 循环更好。`do-while` 方式会在条件判断 `square` 没有超出后直接运行 `square += board[square]`，这种方式可以去掉 `while` 版本中的数组越界判断。

条件语句

根据特定的条件执行特定的代码通常是十分有用的，例如：当错误发生时，你可能想运行额外的代码；或者，当输入的值太大或太小时，向用户显示一条消息等。要实现这些功能，你就需要使用条件语句。

Swift 提供两种类型的条件语句：`if` 语句和 `switch` 语句。通常，当条件较为简单且可能的情况很少时，使用 `if` 语句。而 `switch` 语句更适用于复杂的条件、可能的情况很多且需要用到模式匹配(pattern-matching)的情境。

If

`if` 语句最简单的形式就是只包含一个条件，当且仅当该条件为真时，才执行相关代码：

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

上面的例子会判断温度是否小于等于 32 华氏度（水的冰点）。如果是，则打印一条消息；否则，不打印任何消息，继续执行 `if` 块后面的代码。

当然，`if` 语句允许二选一，也就是当条件为假时，执行 `else` 语句：

```
1. temperatureInFahrenheit = 40
2. if temperatureInFahrenheit <= 32 {
```

```
3.     println("It's very cold. Consider wearing a scarf.")
4. } else {
5.     println("It's not that cold. Wear a t-shirt.")
6. }
7. // prints "It's not that cold. Wear a t-shirt."
```

显然，这两条分支中总有一条会被执行。由于温度已升至 40 华氏度，不算太冷，没必要再围围巾——因此，**else** 分支就被触发了。

你可以把多个 **if** 语句链接在一起，像下面这样：

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

在上面的例子中，额外的 **if** 语句用于判断是不是特别热。而最后的 **else** 语句被保留了下来，用于打印既不冷也不热时的消息。

实际上，最后的 **else** 语句是可选的：

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

在这个例子中，由于既不冷也不热，所以不会触发 **if** 或 **else if** 分支，也就不会打印任何消息。

Switch

switch 语句会尝试把某个值与若干个模式(pattern)进行匹配。根据第一个匹配成功的模式，**switch** 语句会执行对应的代码。当有可能的情况较多时，通常用 **switch** 语句替换 **if** 语句。**switch** 语句最简单的形式就是把某个值与一个或若干个相同类型的值作比较：

```
1. switch `some value to consider` {
2.   case `value 1`:
3.     `respond to value 1`
4.   case `value 2`,
5.     `value 3`:
6.     `respond to value 2 or 3`
7.   default:
8.     `otherwise, do something else`
9. }
```

switch 语句都由多个 **case** 构成。为了匹配某些更特定的值，**Swift** 提供了几种更复杂的匹配模式，这些模式将在本节的稍后部分提到。

每一个 **case** 都是代码执行的一条分支，这与 **if** 语句类似。与之不同的是，**switch** 语句会决定哪一条分支应该被执行。

switch 语句必须是完备的。这就是说，每一个可能的值都必须至少有一个 **case** 块与之对应。在某些不可能涵盖所有值的情况下，你可以使用默认(**default**)块满足该要求，这个默认块必须在 **switch** 语句的最后面。

下面的例子使用 **switch** 语句来匹配一个名为 **someCharacter** 的小写字符：

```
1. let someCharacter: Character = "e"
2. switch someCharacter {
3.   case "a", "e", "i", "o", "u":
4.     println("\(someCharacter) is a vowel")
5.   case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
6.     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
7.     println("\(someCharacter) is a consonant")
8.   default:
```

```
9.     println("\(someCharacter) is not a vowel or a consonant")
10. }
11. // prints "e is a vowel"
```

在这个例子中，第一个 **case** 块用于匹配五个元音，第二个 **case** 块用于匹配所有的辅音。

由于为其它可能的字符写 **case** 块没有实际的意义，因此在这个例子中使用了默认块来处理剩下的既不是元音也不是辅音的字符——这就保证了 **switch** 语句的完备性。

不存在隐式的贯穿(Fallthrough)

与 C 语言和 Objective-C 中的 **switch** 语句不同，在 Swift 中，当匹配的 **case** 块中的代码执行完毕后，程序会终止 **switch** 语句，而不会继续执行下一个 **case** 块。这也就是说，不需要在 **case** 块中显式地使用 **break** 语句。这使得 **switch** 语句更安全、更易用，也避免了因忘记写 **break** 语句而产生的错误。

注意：你依然可以在 **case** 块中的代码执行完毕前跳出，详情请参考 Switch 语句中的 Break 待添加链接

每一个 **case** 块都必须包含至少一条语句。像下面这样书写代码是无效的，因为第一个 **case** 块是空的：

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

不像 C 语言里的 **switch** 语句，在 Swift 中，**switch** 语句不会同时匹配 "a" 和 "A"。相反的，上面的代码会引起编译期错误：case "a": does not contain any executable statements——这就避免了意外地从一个 **case** 块贯穿到另外一个，使得代码更安全、也更直观。

一个 **case** 也可以包含多个模式，用逗号把它们分开（如果太长了也可以分行写）：

```
1. switch `some value to consider` {
2. case `value 1`,
3.   `value 2`:
4.     `statements`
5. }
```

注意：如果想要贯穿特定的 **case** 块中，请使用 **fallthrough** 语句，详情请参考贯穿 (Fallthrough)待添加链接

范围匹配

case 块的模式也可以是一个值的范围。下面的例子展示了如何使用范围匹配来输出任意数字对应的自然语言格式：

```
1. let count = 3_000_000_000_000
2. let countedThings = "stars in the Milky Way"
3. var naturalCount: String
4. switch count {
5. case 0:
6.     naturalCount = "no"
7. case 1...3:
8.     naturalCount = "a few"
9. case 4...9:
10.    naturalCount = "several"
11. case 10...99:
12.    naturalCount = "tens of"
13. case 100...999:
14.    naturalCount = "hundreds of"
15. case 1000...999_999:
16.    naturalCount = "thousands of"
17. default:
18.    naturalCount = "millions and millions of"
19. }
20. println("There are \(naturalCount) \(countedThings).")
```

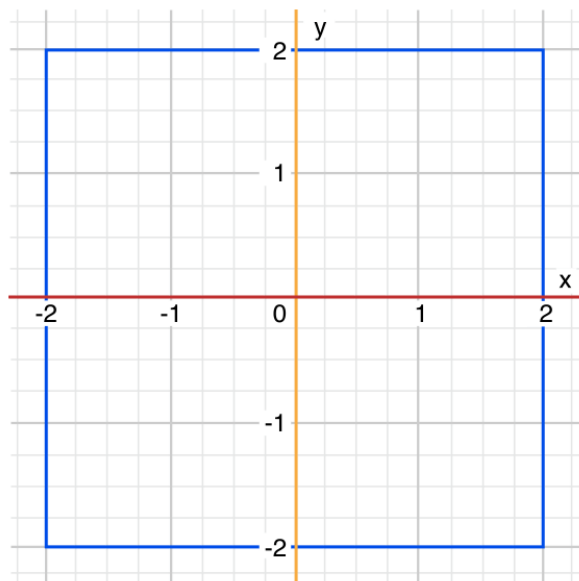
```
21. // prints "There are millions and millions of stars in the Milky Way."
```

元组 (Tuple)

你可以使用元组在同一个 **switch** 语句中测试多个值。元组中的元素可以是值，也可以是范围。另外，使用下划线(_)来匹配所有可能的值。

下面的例子展示了如何使用一个(Int, Int)类型的元组来分类下图中的点(x, y):

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



在上面的例子中，**switch** 语句会判断某个点是否是原点(0, 0)，是否在红色的 **x** 轴上，是否在黄色 **y** 轴上，是否在一个以原点为中心的 4x4 的矩形里，或者在这个矩形外面。

不像 C 语言，Swift 允许多个 **case** 匹配同一个值。实际上，在这个例子中，点(0,0)可以匹配所有四个 **case**。但是，如果存在多个匹配，那么只会执行第一个被匹配到的 **case** 块。考虑点(0,0)会首先匹配 **case** (0, 0)，因此剩下的能够匹配(0,0)的 **case** 块都会被忽视掉。

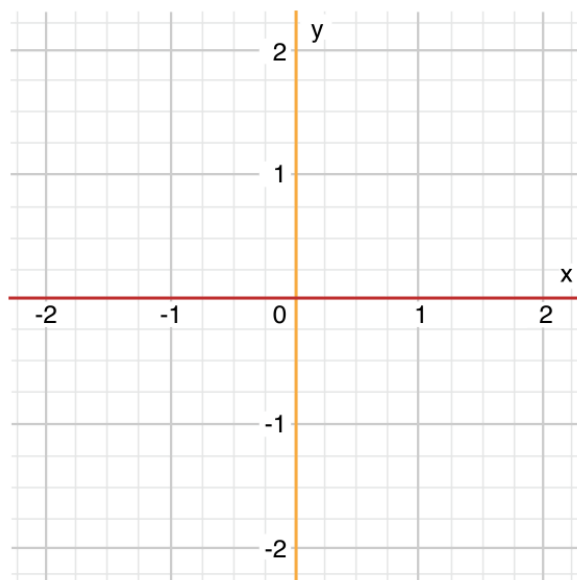
值绑定 (Value Bindings)

case 块的模式允许将匹配的值绑定到一个临时的常量或变量，这些常量或变量在该 **case** 块里就可以被引用了——这种行为被称为值绑定。

下面的例子展示了如何在一个(Int, Int)类型的元组中使用值绑定来分类下图中的点(x, y)：

```
1. let anotherPoint = (2, 0)
2. switch anotherPoint {
3. case (let x, 0):
4.     println("on the x-axis with an x value of \(x)")
5. case (0, let y):
6.     println("on the y-axis with a y value of \(y)")
7. case let (x, y):
8.     println("somewhere else at (\(x), \(y))")
```

```
9. }  
10. // prints "on the x-axis with an x value of 2"
```



在上面的例子中，**switch** 语句会判断某个点是否在红色的 **x** 轴上，是否在黄色 **y** 轴上，或者不在坐标轴上。

这三个 **case** 都声明了常量 **x** 和 **y** 的占位符，用于临时获取元组 **anotherPoint** 的一个或两个值。第一个 **case**——**case (let x, 0)** 将匹配一个纵坐标为 **0** 的点，并把这个点的横坐标赋给临时的常量 **x**。类似的，第二个 **case**——**case (0, let y)** 将匹配一个横坐标为 **0** 的点，并把这个点的纵坐标赋给临时的常量 **y**。

一旦声明了这些临时的常量，它们就可以在其对应的 **case** 块里引用。在这个例子中，它们用于简化 **println** 的书写。

请注意，这个 **switch** 语句不包含默认块。这是因为最后一个 **case**——**case let(x, y)** 声明了一个可以匹配余下所有值的元组。这使得 **switch** 语句已经完备了，因此不需要再书写默认块。

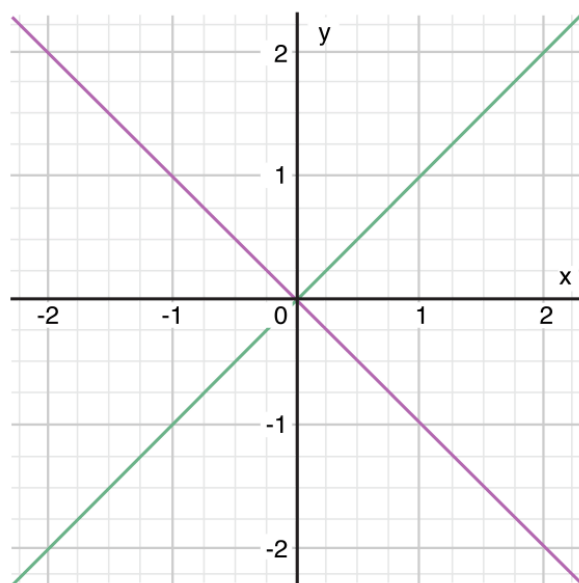
在上面的例子中，**x** 和 **y** 是常量，这是因为没有必要在其对应的 **case** 块中修改它们的值。然而，它们也可以是变量——程序将会创建临时变量，并用相应的值初始化它。修改这些变量只会影响其对应的 **case** 块。

Where

`case` 块的模式可以使用 `where` 语句来判断额外的条件。

下面的例子把下图中的点(x, y)进行了分类:

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



在上面的例子中，`switch` 语句会判断某个点是否在绿色的对角线 `x == y` 上，是否在紫色的对角线 `x == -y` 上，或者不在对角线上。

这三个 `case` 都声明了常量 `x` 和 `y` 的占位符，用于临时获取元组 `yetAnotherPoint` 的两个值。这些常量被用作 `where` 语句的一部分，从而创建一个动态的过滤器(filter)。当且仅当 `where` 语句的条件为真时，匹配到的 `case` 块才会被执行。

就像是值绑定中的例子，由于最后一个 **case** 块匹配了余下所有可能的值，**switch** 语句就已经完备了，因此不需要再书写默认块。

控制转移语句

控制转移语句改变你代码的执行顺序，通过它你可以实现代码的跳转。**Swift** 有四种控制转移语句。

- **continue**
- **break**
- **fallthrough**
- **return**

我们将会在下面讨论 **continue**, **break**, 和 **fallthrough** 语句。**return** 语句将会在函数章节讨论。

Continue

continue 告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。就好像在说“本次循环迭代我已经执行完了”，但是并不会离开整个循环体。

NOTE: 在一个 **for-condition-increment** 循环体中，在调用 **continue** 语句后，迭代增量仍然会被计算求值。循环体继续像往常一样工作，仅仅只是循环体中的执行代码会被跳过。

下面的例子把一个小写字串中的元音字母和空格字符移除，生成了一个含义模糊的短句：

```
1. let puzzleInput = "great minds think alike"
2. var puzzleOutput = ""
3. for character in puzzleInput {
4.     switch character {
5.     case "a", "e", "i", "o", "u", " ":
6.         continue
7.     default:
8.         puzzleOutput += character
9.     }
```

```
10. }  
11. println (puzzleOutput)  
12. // prints "grtmndsthnlk"
```

在上面的代码中，只要匹配到元音字母或者空格字符，就调用 **continue** 语句，使本次循环迭代结束，从新开始下次循环迭代。这种行为使 **switch** 匹配到元音字母和空格字符时不做处理，而不是让每一个匹配到的字符都被打印。

Break

break 语句会立刻结束整个控制流的执行。当你想要更早的结束一个 **switch** 代码块或者一个循环体时，你都可以使用 **break** 语句。

在循环体中使用 Break

当在一个循环体中使用 **break** 时，会立刻中断该循环体的执行，然后跳转到表示循环体结束的大括号(**}**)后的第一行代码。不会再有本次循环迭代的代码被执行，也不会有下次的循环迭代产生。

在 Switch 代码块中使用 Break

当在一个 **switch** 代码块中使用 **break** 时，会立即中断该 **switch** 代码块的执行，并且跳转到表示 **switch** 代码块结束的大括号(**}**)后的第一行代码。

这种特性可以被用来匹配或者忽略一个或多个分支。因为 **Swift** 语言的 **switch** 需要包含所有的分支而且不允许有为空的分支，有时为了使你的意图更明显，需要特意匹配或者忽略某个分支。那么当你想忽略某个分支时，可以在该分支内写上 **break** 语句。当那个分支被匹配到时，分支内的 **break** 语句立即结束 **switch** 代码块。

NOTE: 当一个 **switch** 分支仅仅包含注释时，会被报编译时错误。注释不是代码语句而且也不能让 **switch** 分支达到被忽略的效果。你总是可以使用 **break** 来忽略某个分支。

下面的例子通过 **switch** 来判断一个 **Character** 值是否代表下面四种语言之一。为了简洁，多个值被包含在了同一个分支情况中。

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

这个例子检查 `numberSymbol` 是否是拉丁，阿拉伯，中文或者泰语中的 1...4 之一。如果被匹配到，该 `switch` 分支语句给 `Int?` 类型变量 `possibleIntegerValue` 设置一个整数值。

当 `switch` 代码块执行完后，接下来的代码通过使用可选绑定来判断 '`possibleIntegerValue`' 是否曾经被设置过值。因为是可选类型的缘故，'`possibleIntegerValue`' 有一个隐式的初始值 `nil`，所以仅仅当 `possibleIntegerValue` 曾被 `switch` 代码块的前四个分支中的某个设置过一个值时，可选的绑定将会被判定为成功。

在上面的例子中，想要把 `Character` 所有的可能性都枚举出来是不现实的，所以使用 `default` 分支来包含所有上面没有匹配到字符的情况。由于这个 `default` 分支不需要执行任何动作，所以它只写了一条 `break` 语句。一旦落入到 `default` 分支中后，`break` 语句就完成了该分支的所有代码操作，代码继续向下，开始执行 `if let` 语句。

Fallthrough

Swift 语言中的 `switch` 不会从上一个 `case` 分支落入到下一个 `case` 分支中。相反，只要第一个匹配到的 `case` 分支完成了它需要执行的语句，整个 `switch` 代码块完成了它的执行。相比之下，C 语言要求你显示的插入 `break` 语句到每个 `switch` 分支的末尾来阻止自动落入到下一个 `case` 分支中。Swift 语言的这种避免默认落入到下一个分支中的特性意味着它的 `switch` 功能要比 C 语言的更加清晰和可预测，可以避免无意识地执行多个 `case` 分支从而引发的错误。

如果你确实需要 C 风格的落入(fallthrough)的特性，你可以在每个需要该特性的 `case` 分支中使用 `fallthrough` 关键字。下面的例子使用 `fallthrough` 来创建一个数字的描述语句。

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

这个例子定义了一个 `String` 类型的变量 `description` 并且给它设置了一个初始值。函数使用 `switch` 逻辑来判断 `integerToDescribe` 变量的值。当 `integerToDescribe` 的值属于列表中的质数之一时，该函数添加一段文字在 `description` 后，来表明这个数是数字是一个质数。然后它使用 `fallthrough` 关键字来"落入"到 `default` 分支中。`default` 分支添加一段额外的文字在 `description` 的最后,至此 `switch` 代码块执行完了。

如果 `integerToDescribe` 的值不属于列表中的任何质数，那么它不会匹配到第一个 `switch` 分支。而这里没有其他特别的分支情况，所以 `integerToDescribe` 匹配到包含所有的 `default` 分支中。

当 `switch` 代码块执行完后，使用 `println` 函数打印该数字的描述。在这个例子中，数字 5 被

准确的识别为了一个质数。

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。

Labeled Statements

在 **Swift** 语言中，你可以在循环体和 `switch` 代码块中嵌套循环体和 `switch` 代码块来创造复杂的控制流结构。然而，循环体和 `switch` 代码块两者都可以使用 `break` 语句来提前结束整个方法体。因此，显式地指明 `break` 语句想要终止的是哪个循环体或者 `switch` 代码块，会很有用。类似地，如果你有许多嵌套的循环体，显式指明 `continue` 语句想要影响哪一个循环体也会非常有用。

为了实现这个目的，你可以使用标签来标记一个循环体或者 `switch` 代码块，当使用 `break` 或者 `continue` 时，带上这个标签，可以控制该标签代表对象的中断或者执行。

产生一个带标签的语句是通过在该语句的关键词的同一行前面放置一个标签，并且该标签后面还需带着一个冒号。下面是一个 `while` 循环体的语法，同样的规则适用于所有的循环体和 `switch` 代码块。

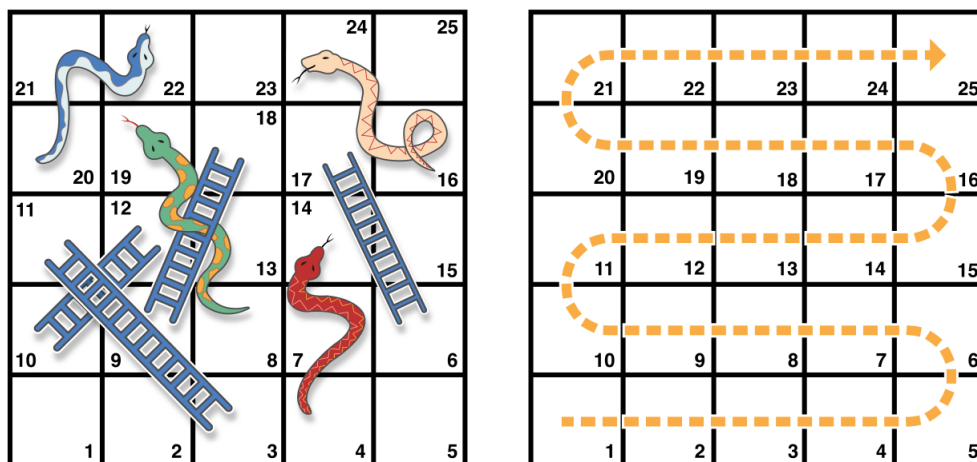
```
1. label name: while condition {  
2.     statements  
3. }
```

下面的例子是在一个带有标签的 `while` 循环体中调用 `break` 和 `continue` 语句，该循环体是上述章节中蛇梯棋游戏的改编版本。这次，游戏增加了一条额外的规则：

- 为了获胜，你必须刚好落在方格 25 中。

如果某次掷骰子使你的移动超出方格 25，你必须重新掷骰子，直到你掷出的骰子数刚好使你能落在方格 25 中。

游戏的棋盘和之前一样：



值 `finalSquare`, `board`, `square` 和 `diceRoll` 的初始化也和之前一样:

```
1. let finalSquare = 25
2. var board = Int[] (count: finalSquare + 1, repeatedValue: 0)
3. board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
4. board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
5. var square = 0
6. var diceRoll = 0
```

这个版本的游戏使用 `while` 循环体和 `switch` 方法块来实现游戏的逻辑。`while` 循环体有一个标签名 `gameLoop`, 来表明它是蛇梯棋游戏的主循环。

该 `while` 循环体的条件判断语句是 `while square != finalSquare`, 这表明你必须刚好落在方格 25 中。

```
1. gameLoop: while square != finalSquare {
2.     if ++diceRoll == 7 { diceRoll = 1 }
3.     switch square + diceRoll {
4.     case finalSquare:
5.         // diceRoll will move us to the final square, so the game is over
6.         break gameLoop
7.     case let newSquare where newSquare > finalSquare:
8.         // diceRoll will move us beyond the final square, so roll again
9.         continue gameLoop
10.    default:
```

```
11.         // this is a valid move, so find out its effect
12.         square += diceRoll
13.         square += board[square]
14.     }
15. }
16. println("Game over!")
```

每次循环迭代开始时掷骰子。与之前玩家掷完骰子就立即移动不同，这里使用了 **switch** 来考虑每次移动可能产生的结果，从而决定玩家本次是否能够移动。

- 如果骰子数刚好使玩家移动到最终的方格里，游戏结束。**break gameLoop** 语句跳转控制去执行 **while** 循环体后的第一行代码，游戏结束。
- 如果骰子数将会使玩家的移动超出最后的方格，那么这种移动是不合法的，玩家需要重新掷骰子。**continue gameLoop** 语句结束本次 **while** 循环的迭代，开始下一次循环迭代。
- 在剩余的所有情况中，骰子数产生的都是合法的移动。玩家向前移动骰子数个方格，然后游戏逻辑再处理玩家当前是否处于蛇头或者梯子的底部。本次循环迭代结束，控制跳转到 **while** 循环体的条件判断语句处，再决定是否能够继续执行下次循环迭代。

NOTE: 如果上述的 **break** 语句没有使用 **gameLoop** 标签，那么它将会中断 **switch** 代码块而不是 **while** 循环体。使用 **gameLoop** 标签清晰的表明了 **break** 想要中断的是哪个代码块。同时请注意，当调用 **continue gameLoop** 去跳转到下一次循环迭代时，这里使用 **gameLoop** 标签并不是严格必须的。因为在这个游戏中，只有一个循环体，所以 **continue** 语句会影响到哪个循环体是没有歧义的。然而，**continue** 语句使用 **gameLoop** 标签也是没有危害的。这样做符合标签的使用规则，同时参照旁边的 **break gameLoop**，能够使游戏的逻辑更加清晰和易于理解。

2.6. 函数

本篇译者：Da 杯柠檬水（[微博](#)），敬请勘误。

函数

函数是执行特定任务的代码自包含块。通过给定一个函数名称标识它是什么，并在需要的时候使用该名称来调用函数以执行任务。

Swift 的统一的功能语法足够灵活的，可表达任何东西，无论是不带参数名称的简单的样式函数，还是带本地和外部参数名称的复杂的 Objective-C 样式方法。参数可为简单函数调用提供默认值，并且可以被作为输入/输出参数传递，在函数执行完成时修改传递来的变量。

Swift 中的每个函数都有一个类型，包括函数的参数类型和返回类型。你可以像使用 Swift 中其他类型一样使用该类型,这使得它很容易将函数作为参数传递给其他函数,甚至从函数中返回函数类型。•函数也可以被写入其他函数中以在函数作用于中封装有用的功能。

定义和调用函数

当你定义一个函数时，你可以选择性地定义一个或多个名称，类型值作为函数的输入（称为形参），或者定义一个函数结束后返回值的类型（称之为返回型）。每一个函数都有一个函数名，用来描述了函数执行的任务。要使用一个函数时，可使用它的名称进行“调用”，并通过它的输入值（称为实参--argument）来匹配函数的参数类型。一个函数的实参（arguments）必须始终和函数形参（parameter）顺序一致。

例如在下面的例子中被调用的函数 `greetingForPerson`，像它描述的那样 -- 它需要一个人的名字作为输入并返回一句针对那个人的问候语。为了实现该功能，你定义了一个输出参数--一个名为 `personName` 的字符串值，以及一个 `String` 返回类型，包含一个针对那个人的问候语：

```
1. func sayHello(personName: String) -> String {
2.     let greeting = "Hello, " + personName + "!"
3.     return greeting
4. }
```

所有这些信息都汇总到以 `func` 关键字为前缀的函数定义中。使用箭头 `->` 来指明函数的返回类型（一个连字符后跟一个向右的箭头），后边跟着返回的类型名称。•

该定义描述了函数的作用是什么，它期望接收什么，以及完成后返回的结果。该定义可轻易地让你在代码中的其他地方清晰明确地调用该函数：

```
1. println(sayHello("Anna"))
2. // prints "Hello, Anna!"
3. println(sayHello("Brian"))
4. // prints "Hello, Brian!"
```

你可以通过给它传递一个圆括号内 **String** 实参值来调用 **sayHello** 函数，例如 **sayHello** ("Anna")。由于该函数返回一个 **String** 值，**sayHello** 可以被包裹在一个 **println** 函数调用中来打印字符串，看看它的返回值，如上图所示。

sayHello 的函数主体首先定义了一个新的名为 **greeting** 的 **String** 常量，并将其设置加上 **personName** 组成一句简单的问候消息。然后这个 **greeting** 以关键字 **return** 来传回到函数外部。只要 **return greeting** 被调用，函数执行完毕后就会返回 **greeting** 的当前值。

你可以通过不同的输入值多次调用 **sayHello** 的函数。上面的例子显示了如果使用 "Anna" 输入值调用它会发生什么，以及以 "Brian" 输入值调用时会发生什么。函数为每种情况量身定制了问候语。

为了简化这个函数的主体，可把消息创建和 **return** 语句合并成一行：

```
1. func sayHello(personName: String) -> String {
2.     return "Hello again, " + personName + "!"
3. }
4. println(sayHello("Anna"))
5. // prints "Hello again, Anna!"
```

函数的形参和返回值

在 **swift** 中，函数的形参和返回值是非常具有灵活性的。你可以定义任何事情，无论是一个简单的仅有一个未命名形参的工具函数，还是那种具有丰富的参数名称和不同的形参选项的复杂函数。

多输入形参

函数可以有多个输入形参，把它们写到函数的括号内，并用逗号加以分隔。下面这个函数设置了一个半开区间的开始和结束索引，用来计算在范围内有多少元素：

```
1. func halfOpenRangeLength(start: Int, end: Int) -> Int {
2.     return end - start
3. }
4. println(halfOpenRangeLength(1, 10))
5. // prints "9"
```

无形参函数

函数并没有要求一定要定义的输入形参。下面就是一个没有输入形参的函数，任何时候调用时它总是返回相同的 **String** 消息：

```
1. func sayHelloWorld() -> String {
2.     return "hello, world"
3. }
4. println(sayHelloWorld())
5. // prints "hello, world"
```

该函数的定义还需要在函数的名称后跟一对儿圆括号，即使它不带任何形参。当函数被调用时函数名称也要跟着一对儿空括号。

无返回值的函数

函数不需要定义一个返回类型。这里有一个版本的 **sayHello** 函数，称为 **waveGoodbye**，它会打印自己的 **String** 值而不是返回它：

```
1. func sayGoodbye(personName: String) {
2.     println("Goodbye, \(personName)!")
3. }
4. sayGoodbye("Dave")
5. // prints "Goodbye, Dave!"
```

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

提示：严格地说，`sayGoodbye` 函数确实还返回一个值，即使没有定义返回值。没有定义返回类型的函数返回了一个 `Void` 类型的特殊值。这仅是一个空元组，这里边没有元素，可以被写成`()`。

当一个函数调用时它的返回值可以忽略不计：

```
1. func printAndCount(stringToPrint: String) -> Int {
2.     println(stringToPrint)
3.     return countElements(stringToPrint)
4. }
5. func printWithoutCounting(stringToPrint: String) {
6.     printAndCount(stringToPrint)
7. }
8. printAndCount("hello, world")
9. // prints "hello, world" and returns a value of 12
10. printWithoutCounting("hello, world")
11. // prints "hello, world" but does not return a value
```

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

提示：返回值可以忽略，但一个定义了返回值的函数则必须有返回值。对于一个定义了返回类型的函数来说，如果没有返回值，那么将不允许控制流离开函数的底部。如果试图这样做将出现一个编译时错误。

多返回值函数

你可以使用一个元组类型作为函数的返回类型，来返回一个由多个值组成的复合返回值。

下面的例子定义了一个名为 `count` 函数，用来计算字符串中基于标准美式英语的元音、辅音以及字符的数量：

```
1. func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {
2.     var vowels = 0, consonants = 0, others = 0
3.     for character in string {
4.         switch String(character).lowercaseString {
```

```
5.         case "a", "e", "i", "o", "u":
6.             ++vowels
7.         case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
8.             "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
9.             ++consonants
10.        default:
11.            ++others
12.        }
13.    }
14.    return (vowels, consonants, others)
15. }
```

您可以使用此计数函数来对任意字符串进行字符计数，以检索一个包含三个指定 `Int` 值的元素统计总数：

```
1. let total = count("some arbitrary string!")
2. println("\(total.vowels) vowels and \(total.consonants) consonants")
3. // prints "6 vowels and 13 consonants"
```

注意：这一点上元组的成员不需要被命名，元组是从函数中返回的，因为它们的名字已经被指定为函数的返回类型的一部分

函数形参名

所有上面的函数都为其形参定义了形参名：

```
1. func someFunction(parameterName: Int) {
2.     // function body goes here, and can use parameterName
3.     // to refer to the argument value for that parameter
4. }
```

然而，这些参数名的仅能在函数本身的主体内使用，不能在调用函数时使用。这种形参类型名称被称之为本地形参名（`local parameter name`），因为它们只能在函数的主体中使用。

外部形参名

有时当你调用一个函数将每个形参进行命名是非常有用的，以表明你把每个实参传递给函数的目的。

如果你希望使用你函数的人在调用函数时提供形参名称，那除了本地形参名外，你还要为每个形参定义一个外部形参名称。你写一个外部形参名称在它支持的本地形参名称之前，之间用一个空格来分隔：

```
1. func someFunction(externalParameterName localParameterName: Int) {  
2.     // function body goes here, and can use localParameterName  
3.     // to refer to the argument value for that parameter  
4. }
```

提示：如果您为形参提供一个外部形参名称，那么外部形参名必须在调用时使用。

举一个例子，考虑下面的函数，通过在它们之间插入第三个"joiner"字符串来连接两个字符串：

```
1. func join(s1: String, s2: String, joiner: String) -> String {  
2.     return s1 + joiner + s2  
3. }
```

当你调用这个函数，你传递给函数的三个字符串的目的就不是很清楚了：

```
1. join("hello", "world", ", ")  
2. // returns "hello, world"
```

为了使这些字符串值的目的更为清晰，为每个 join 函数形参定义外部形参名称：

```
1. func join(string s1: String, toString s2: String, withJoiner joiner: String)  
2.     -> String {  
3.     return s1 + joiner + s2  
4. }
```

在这个版本的 join 函数中，第一个形参的外部名称 **string**，本地名称 **s1**；第二个形参的外部名称 **toString**，本地名称 **s2**；第三个形参的外部名称是 **withJoiner**，本地名称为 **joiner**。

现在, 您可以使用这些外部形参名称清楚地调用该函数:

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

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

注意: 在别人第一次阅读你的代码不知道你函数形参目的时候, 就要考虑到使用外部形参名称了。在调用函数的时候, 如果每个形参的目的清晰明确的话, 那你就无需指定外部形参名。

外部参数名称速记

如果你想为一个函数提供一个外部形参名, 然而本地形参名已经使用了一个合适的名称了, 那你就不需要两次书写该形参的名称。相反, 你可以写一次名字, 并用一个 **hash** 符号 (`#`) 作为名称的前缀。这就告诉 **Swift** 使用名称相同的本地行参名称和外部形参名称。

这个例子定义了一个名为 `containsCharacter` 的函数, 通过在本地形参名前添加 **hash** 符号 (`#`) 来定义外部形参名称。

```
1. func containsCharacter(#string: String, #characterToFind: Character) -
   > Bool {
2.     for character in string {
3.         if character == characterToFind {
4.             return true
5.         }
6.     }
7.     return false
8. }
```

该函数对形参名的选择使得其函数主题更加清晰易读, 并且在调用该函数时也不会有歧义:

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

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。

注意：请在函数形参列表的末尾放置带默认值的形参。这将确保所有函数调用都使用顺序相同的无默认值实参，并让在每种情况下清晰地调用相同的函数。

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

```
1. func join(string s1: String, toString s2: String,
2.     withJoiner joiner: String = " ") -> String {
3.     return s1 + joiner + s2
4. }
```

如果在 `join` 函数调用时为 `joiner` 提供了字符串值，那么该字符串值可以用来连接两个字符串，就跟以前一样：

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

但是，如果函数调用时没有为 `joiner` 提供值，就会使用单个空格（" "）的默认值：

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

有默认值的外部形参名

在大多数情况下,为所有形参提供一个带默认值的外部名是非常有用的(因此要求)。如果在调用函数的时候提供了一个值，那么这将确保形参对应的实参有着明确的目的。

为了使这个过程更容易，当你自己没有提供外部名称时，**Swift** 将为你定义的任何默认形参提供一个自动外部名。这个自动外部名和本地名一样，就像你已经在本地名前添加了 `hash` 符号（`#`）一样。

这里有一个早期 `join` 函数版本，没有为任何外部形参提供外部名，但仍然提供了 `joiner` 形参的默认值：

```
1. func join(s1: String, s2: String, joiner: String = " ") -> String {
2.     return s1 + joiner + s2
3. }
```

在这种情况下，**Swift** 为带默认值的形参提供了外部形参名，当调用该函数的时候，外部形参名必须让形参的目的明确无歧义：

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

注意：•在定义形参时，你可以通过使用下划线(_)来代替显示外部名称。不过在适当的情况下，带有默认值形参的外部名通常是优先推荐的。

可变形参

一个可变形参可接受零个或多个指定类型的值。当函数被调用时，你可以使用可变形参来指定--形参可以用来传递任意数量的输入值。可通过在形参的类型名后边插入三个点符号(...)来编写可变形参。

传递至可变形参的值在函数主体内是以适当类型的数组存在的。例如,一个可变参数的名称为 `numbers` 和类型为 `Double...`在函数体内就作为名为 `numbers` 类型为 `Double[]`的常量数组。

下边示例为任何长度的数字列表计算算术平均值：

```
1. func arithmeticMean(numbers: Double...) -> Double {
2.     var total: Double = 0
3.     for number in numbers {
4.         total += number
5.     }
6.     return total / Double(numbers.count)
7. }
8. arithmeticMean(1, 2, 3, 4, 5)
9. // returns 3.0, which is the arithmetic mean of these five numbers
```

```
10. arithmeticMean(3, 8, 19)
11. // returns 10.0, which is the arithmetic mean of these three numbers
```

注意：函数最多可以有一个可变形参，而且它必须出现在参数列表的最后，以避免使用多个形参调用函数引发歧义。如果你的函数有一个或多个带有默认值的形参，并且还有可变形参，请将可变形参放在所有默认形参之后，也就是列表的最末尾。

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。

但是有时候，函数有一个形参值的变量副本是非常有用的。您可以指定一个或多个形参作为变量形参，从而避免在函数内部为自己定义一个新的变量。变量参数是变量而非常量，并给函数一个可修改的形参值副本。

在参数名称前用关键字 **var** 定义变量参数：

```
1. func alignRight(var string: String, count: Int, pad: Character) -> String {
2.     let amountToPad = count - countElements(string)
3.     for _ in 1...amountToPad {
4.         string = pad + string
5.     }
6.     return string
7. }
8. let originalString = "hello"
9. let paddedString = alignRight(originalString, 10, "-")
10. // paddedString is equal to "-----hello"
11. // originalString is still equal to "hello"
```

这个例子定义了一个新函数叫做 **alignRight**，用于将一个输入字符串和更长的输出字符串右边缘对齐。所有左侧的空白使用指定的占位符来填充。在这个例子中，字符串 **"hello"** 被转化为字符串 **"-----hello"**。

`alignRight` 函数把输入的形参字符串定义成一个变量形参。这意味着字符串现在可以作为一个本地变量，用传入的字符串值初始化，并且可以在函数体中进行相应操作。

函数首先要找出有多少字符需要被添加到字符串的左侧，从而在整个字符串中靠右对齐。这个值存储在本地常量 `amountToPad` 中。该函数然后将填充字符的 `amountToPad` 个字符拷贝到现有的字符串的左边，并返回结果。

注意：你对变量形参所做的改变不会比调用函数更持久，并且在函数体外是不可见的。变量形参仅存在于函数调用的声明周期中。

In-Out 形参

如上描述，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。

通过在形参定义的开始添加 `inout` 关键字来编写 `in-out` 形参。`In-Out` 形参有一个传递至函数的值，由函数修改，并从函数返回来替换原来的值。

你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。当你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。

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

这里的一个叫做 `swapTwoInts` 函数，它有两个称为 `a` 和 `b` 的 `in-out` 整型形参：

```
1. func swapTwoInts(inout a: Int, inout b: Int) {  
2.     let temporaryA = a  
3.     a = b  
4.     b = temporaryA  
5. }
```

`swapTwoInts` 函数只是简单地交换 `a`、`b` 的值。该函数通过存储一个名为 `temporaryA` 临时常量的值，指定 `b` 的值到 `a`，然后分配 `temporaryA` 到 `b` 执行该交换。

你可以通过两个 `Int` 类型的变量调用 `swapTwoInts` 函数，从而交换它们的值。需要注意的是当它们被传递给 `swapTwoInts` 函数时，`someInt` 和 `anotherInt` 名称前要加上前缀符号 `&`:

```
1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // prints "someInt is now 107, and anotherInt is now 3"
```

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

注意：In-out 形参不同于从函数返回一个值。上边 `swapTwoInts` 例子没有定义返回类型或者返回值，但它仍然会修改 `someInt` 和 `anotherInt` 的值。对函数来说，In-out 形参是一个影响函数主体范围外的可选方式。

函数类型

每一个函数都有特定的函数类型，由函数的形参类型和返回类型组成。例如：

```
1. func addTwoInts(a: Int, b: Int) -> Int {
2.     return a + b
3. }
4. func multiplyTwoInts(a: Int, b: Int) -> Int {
5.     return a * b
6. }
```

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

这两个函数的类型都是 `(Int, Int)->Int`。可以解读为："这个函数类型，它有两个 `Int` 类型形参，并返回一个 `Int` 类型的值。"

下面是另一个例子，该函数没有形参或返回值：

```
1. func printHelloWorld() {  
2.     println("hello, world")  
3. }
```

这个函数的类型是 `() -> ()`，或者"没有形参的函数，并返回 `void`。"没有指明返回值的函数通常会返回 `void`，在 `swift` 中相当于一个空元组，显示为 `()`。

使用函数类型

在 `swift` 中您可以像任何其他类型一样的使用函数类型。例如，你可以定义一个常量或变量为一个函数类型，并为变量指定一个对应的函数：

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

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

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

现在你可以使用 `mathFunction` 来调用指定的函数：

```
1. println("Result: \(mathFunction(2, 3))")  
2. // 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：

```
1. mathFunction = multiplyTwoInts  
2. println("Result: \(mathFunction(2, 3))")  
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。

```
1. let anotherMathFunction = addTwoInts
```

```
2. // anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

作为形参类型的函数类型

您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。

下边的例子打印了上边的数学函数的结果：

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

这个例子中定义了一个名为 `printMathResult` 函数，它有三个形参。第一个形参名为 `mathFunction`，类型为`(Int, Int)->Int`。您可以传递任何同类型的函数作为第一个形参的实参。第二和第三个参数 `a`、`b` 都是 `int` 类型。被用来作为数学函数的两个输入值。

当 `printMathResult` 被调用时，它传递 `addTwoInt` 函数，以及整数值 `3` 和 `5`。它使用 `3` 和 `5` 调用了提供的函数，打印的结果是 `8`。

`printMathResult` 的作用是打印调用适当类型的数学函数的结果。该函数真正实现了什么并不重要--它只关心函数的类型是正确的。这使得 `printMathResult` 以一种安全类型的方式把自身的功能转换至函数的调用者。

作为返回类型的函数类型

你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。

下面的例子定义了两个简单的函数调用 `stepForward` 和 `stepBackward`。`stepForward` 函数返回一个输入值+1 的结果，而 `stepBackward` 函数返回一个输入值-1 的结果。这两个函数都有一个相同的`(Int) -> Int` 类型：

```
1. func stepForward(input: Int) -> Int {
2.     return input + 1
3. }
4. func stepBackward(input: Int) -> Int {
5.     return input - 1
6. }
```

这里有一个 `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：

```
1. func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
2.     return backwards ? stepBackward : stepForward
3. }
```

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

```
1. var currentValue = 3
2. let moveNearerToZero = chooseStepFunction(currentValue > 0)
3. // moveNearerToZero now refers to the stepBackward() function
```

前面的例子可以计算出是否需要通过递增或者递减来让 `currentValue` 变量趋于零。
`currentValue` 的初始值为 3，这意味着 `currentValue > 0` 返回为真，并且
`chooseStepFunction` 返回 `stepBackward` 函数。返回函数的引用存储在一个名为
`moveNearerToZero` 的常量里。

如今 `moveNearerToZero` 执行了正确的功能，就可以用来计数到零：

```
1. println("Counting to zero:")
2. // Counting to zero:
3. while currentValue != 0 {
4.     println("\(currentValue)... ")
5.     currentValue = moveNearerToZero(currentValue)
6. }
7. println("zero!")
8. // 3...
9. // 2...
10. // 1...
11. // zero!
```

嵌套函数

迄今为止所有你在本章中遇到函数都是全局函数，在全局作用域中定义。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。`enclosing function` 也可以返回一个嵌套函数，以便在其他作用域中使用嵌套函数。

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

```
1. func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
2.     func stepForward(input: Int) -> Int { return input + 1 }
3.     func stepBackward(input: Int) -> Int { return input - 1 }
4.     return backwards ? stepBackward : stepForward
5. }
6. var currentValue = -4
7. let moveNearerToZero = chooseStepFunction(currentValue > 0)
8. // moveNearerToZero now refers to the nested stepForward() function
9. while currentValue != 0 {
10.     println("\(currentValue)... ")
11.     currentValue = moveNearerToZero(currentValue)
12. }
13. println("zero!")
14. // -4...
15. // -3...
16. // -2...
17. // -1...
18. // zero!
```

2.7. 闭包

本篇译者：Da 杯柠檬水（[微博](#)），敬请勘误。

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

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

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

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

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

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

- * 利用上下文推断参数和返回值类型
- * 单表达式（single-expression）闭包可以省略 return 关键字
- * 参数名称简写
- * Trailing 闭包语法

闭包表达式

嵌套函数是一种在较复杂函数中方便进行命名和定义自包含代码模块的方式。当然，有时候撰写小巧的没有完整定义和命名的类函数结构也是很有用处的，尤其是在处理一些函数并将需要将另外一些函数作为该函数的参数时。

闭包表达式是一种利用简洁语法构建内联闭包的方式。闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。下面闭包表达式的例子通过使用几次迭代展示了 sort 函数定义和语法优化的方式。每一次迭代都用更简洁的方式描述了相同的功能。

sort 函数

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

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

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

排序函数有两个参数：

1. 已知类型值的数组。
2. 一个闭包，采用相同类型的数组的内容的两个参数，并返回一个布尔值来表示是否将第一个值在排序时放到第二个值的前面或是后面。如果第一个值应该出现第二个值之前，闭包需要返回 `true`，否则返回 `false`。

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

提供排序闭包的一种方式是一个符合其类型要求的普通函数，并将其作为 `sort` 函数的第二个参数传入：

```
1. func backwards(s1: String, s2: String) -> Bool {  
2.     return s1 > s2  
3. }  
4. var reversed = sort(names, backwards)  
5. // reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串 (`s1`) 大于第二个字符串 (`s2`)，`backwards` 函数则返回 `true`，表示在新的数组中 `s1` 应该出现在 `s2` 前。字符串中的 "大于" 表示 "按照字母顺序后出现"。这意味着字母 "B" 大于字母 "A"，字符串 "Tom" 大于字符串 "Tim"。其将进行字母逆序排序，"Barry" 将会排在 "Alex" 之后，一次类推。

然而，这是一个相当冗长的方式，本质上只是写了一个单表达式函数 (`a > b`)。在下面的例

子中，利用闭合表达式语法可以更好的构造一个内联排序闭包。

闭包表达式语法

闭包表达式语法有如下一般形式：

```
1. { (parameters) -> returnType in
2.     statements
3. }
```

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。

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：

```
1. reversed = sort(names, { (s1: String, s2: String) -> Bool in
2.     return s1 > s2
3. })
```

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

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

因为这个闭包的函数体部分如此短以至于可以将其改写成一行代码：

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

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

根据上下文推断类型

因为排序闭包是作为函数的参数进行传入的，Swift 可以推断其参数和返回值的类型。 `sort` 期望第二个参数是类型为 `(String, String) -> Bool` 的函数，因此实际上 `String, String` 和 `Bool` 类型并不需要作为闭包表达式定义中的一部分。 因为所有的类型都可以被正确推断，返回箭头 `(->)` 和 围绕在参数周围的括号也可以被省略：

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

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

然而，你也可以使用明确的类型，如果你想它避免读者阅读可能存在的歧义，这样还是值得鼓励的。这个排序函数例子，闭包的目的是很明确的，即排序被替换，而且对读者来说可以安全的假设闭包可能会使用字符串值，因为它正协助一个字符串数组进行排序。

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：

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

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

参数名简写

Swift 自动为内联函数提供了参数名称简写功能，您可以直接通过 `$0,$1,$2` 等名字来引用的闭包的参数的值。

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：

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

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

运算符函数

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。 `Swift` 的 `String` 类型定义了关于大于号 (`>`) 的字符串实现，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。因此，您可以简单地传递一个大于号，`Swift` 可以自动推断出您想使用大于号的字符串函数实现：

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

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

Trailing 闭包

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

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

```
1. func someFunctionThatTakesAClosure(closure: () -> ()) {  
2.     // 函数体部分  
3. }  
4.  
5. // 以下是不使用 trailing 闭包进行函数调用  
6.  
7. someFunctionThatTakesAClosure({  
8.     // 闭包主体部分  
9. })  
10.
```

```
11. // 以下是使用 trailing 闭包进行函数调用
12.
13. someFunctionThatTakesAClosure() {
14.     // 闭包主体部分
15. }
```

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

在上例中作为 **sort** 函数参数的字符串排序闭包可以改写为：

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

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

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

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

```
1. let digitNames = [
2.     0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
3.     5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
4. ]
5. let numbers = [16, 58, 510]
```

上面的代码创建了整数数字到他们的英文名字之间映射字典。同时定义了一个准备转换为字符串的整型数组。

您现在可以通过传递一个 **trailing** 闭包给 **numbers** 的 **map** 方法来创建对应的字符串版本数组。需要注意的时调用 **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：

```
1. let strings = numbers.map {
```

```
2.     (var number) -> String in
3.     var output = ""
4.     while number > 0 {
5.         output = digitNames[number % 10]! + output
6.         number /= 10
7.     }
8.     return output
9. }
10. // strings 常量被推断为字符串类型数组, 即 String[]
11. // 其值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

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

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

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

注意:

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

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

`number` 变量之后除以 10。 因为其是整数, 在计算过程中未除尽部分被忽略。 因此 16 变成了 1, 58 变成了 5, 510 变成了 51。

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

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

捕获 (Capture)

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

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

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

```
1. func makeIncrementor(forIncrement amount: Int) -> () -> Int {
2.     var runningTotal = 0
3.     func incrementor() -> Int {
4.         runningTotal += amount
5.         return runningTotal
6.     }
7.     return incrementor
8. }
```

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

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

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

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

如果我们单独看这个函数，会发现看上去不同寻常：

```
1. func incrementor() -> Int {
2.     runningTotal += amount
3.     return runningTotal
4. }
```

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

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

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

注意：

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

下面为一个使用 `makeIncrementor` 的例子：

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

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

```
1. incrementByTen()
```

```
2. // 返回的值为 10
3. incrementByTen()
4. // 返回的值为 20
5. incrementByTen()
6. // 返回的值为 30
```

如果您创建了另一个 `incrementor`，其会有一个属于自己的独立的 `runningTotal` 变量的引用。下面的例子中，`incrementBySeven` 捕获了一个新的 `runningTotal` 变量，该变量和 `incrementByTen` 中捕获的变量没有任何联系：

```
1. let incrementBySeven = makeIncrementor(forIncrement: 7)
2. incrementBySeven()
3. // 返回的值为 7
4. incrementByTen()
5. // 返回的值为 40
```

注意：

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

闭包是引用类型

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

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

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

```
1. let alsoIncrementByTen = incrementByTen
2. alsoIncrementByTen()
```

```
3. // 返回的值为 50
```

2.8. 枚举

本篇译者：yanguangshi ([Github 主页](#))，敬请勘误。

枚举定义了一个通用类型的一组相关的值，使你可以在你的代码中以一个安全的方式来使用这些值。

如果你熟悉 C 语言，你就会知道，在 C 语言中枚举指定相关名称为一组整型值。Swift 中的枚举更加灵活，不必给每一个枚举成员（**enumeration member**）提供一个值。如果一个值（被认为是“原始”值）被提供给每个枚举成员，则该值可以是一个字符串，一个字符，或是一个整型值或浮点值。

此外，枚举成员可以指定任何类型的关联值存储到枚举成员值中，就像其他语言中的联合体（**unions**）和变体（**variants**）。你可以定义一组通用的相关成员作为枚举的一部分，每一组都有不同的一组与它相关的适当类型的数值。

在 Swift 中，枚举类型是一等（**first-class**）类型。它们采用了很多传统上只被类所支持的特征，例如计算型属性（**computed properties**），用于提供关于枚举当前值的附加信息，实例方法（**instance methods**），用于提供和枚举所代表的值相关联的功能。枚举也可以定义构造器（**initializers**）来提供一个初始成员值；可以在原始的实现基础上扩展它们的功能；可以遵守协议（**protocols**）来提供标准的功能。

欲了解更多相关功能，请参考属性，方法，构造过程，扩展，和协议。

枚举语法（Enumeration Syntax）

使用 **enum** 关键词并且把它们整个定义放在一对大括号内：

```
1. enum SomeEnumeration {  
2.     // enumeration definition goes here  
3. }
```

以下是指南针四个方向的一个例子：

```
1. enum CompassPoint {  
2.     case North  
3.     case South  
4.     case East  
5.     case West  
6. }
```

一个枚举中被定义的值（例如 **North**, **South**, **East** 和 **West**）是枚举的成员值（或者成员）。**case** 关键词表明新的一行成员值将被定义。

注意：不像 **C** 和 **Objective-C** 一样，**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：

```
1. enum Planet {  
2.     case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptun  
3. }
```

每个枚举定义了一个全新的类型。像 **Swift** 中其他类型一样，它们的名字（例如 **CompassPoint** 和 **Planet**）必须以一个大写字母开头。给枚举类型起一个单数名字而不是复数名字，以便于读起来更加容易理解：

```
1. 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：

```
1. directionToHead = .East
```

directionToHead 的类型已知时，当设定它的值时，你可以不再写类型名。使用显示类型的

枚举值可以让代码具有更好的可读性。

匹配枚举值和 Switch 语句

你可以匹配单个枚举值和 `switch` 语句：

```
1. directionToHead = .South
2. switch directionToHead {
3. case .North:
4.     println("Lots of planets have a north")
5. case .South:
6.     println("Watch out for penguins")
7. case .East:
8.     println("Where the sun rises")
9. case .West:
10.    println("Where the skies are blue")
11. }
12. // prints "Watch out for penguins"
```

你可以如此理解这段代码：

“考虑 `directionToHead` 的值。当它等于 `.North`，打印“Lots of planets have a north”。当它等于 `.South`，打印“Watch out for penguins”。”

等等依次类推。

正如在控制流中介绍，当考虑一个枚举的成员们时，一个 `switch` 语句必须全面。如果忽略了 `.West` 这种情况，上面那段代码将无法通过编译，因为它没有考虑到 `CompassPoint` 的全部成员。全面性的要求确保了枚举成员不会被意外遗漏。

当不需要匹配每个枚举成员的时候，你可以提供一个默认 `default` 分支来涵盖所有未明确被提出的任何成员：

```
1. let somePlanet = Planet.Earth
2. switch somePlanet {
3. case .Earth:
```

```
4.     println("Mostly harmless")
5.     default:
6.         println("Not a safe place for humans")
7. }
8. // prints "Mostly harmless"
```

关联值（Associated Values）

上一小节的例子演示了一个枚举的成员是如何被定义（分类）的。你可以为 `Planet.Earth` 设置一个常量或则变量，并且在之后查看这个值。然而，有时候会很有用如果能够把其他类型的关联值和成员值一起存储起来。这能让你随着成员值存储额外的自定义信息，并且当每次你在代码中利用该成员时允许这个信息产生变化。

你可以定义 `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。枚举的这种特性跟其他语言中的可辨识联合（`discriminated unions`），标签联合（`tagged unions`），或者变体（`variants`）相似。

例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有 `UPC-A` 格式的一维码，它使用数字 0 到 9。每一个条形码都有一个代表“数字系统”的数字，该数字后接 10 个代表“标识符”的数字。最后一个数字是“检查”位，用来验证代码是否被正确扫描：



其他商品上标有 `QR` 码格式的二维码，它可以使用任何 `ISO8859-1` 字符，并且可以编码一个最多拥有 2,953 字符的字符串：



对于库存跟踪系统来说，能够把 **UPC-A** 码作为三个整型值的元组，和把 **QR** 码作为一个任何长度的字符串存储起来是方便的。

在 **Swift** 中，用来定义两种商品条码的枚举是这样子的：

```
1. enum Barcode {
2.     case UPCA(Int, Int, Int)
3.     case QRCode(String)
4. }
```

以上代码可以这么理解：

“定义一个名为 **Barcode** 的枚举类型，它可以是 **UPCA** 的一个关联值 (**Int**, **Int**, **Int**)，或者 **QRCode** 的一个字符串类型 (**String**) 关联值。”

这个定义不提供任何 **Int** 或 **String** 的实际值，它只是定义了，当 **Barcode** 常量和变量等于 **Barcode.UPCA** 或 **Barcode.QRCode** 时，关联值的类型。

然后可以使用任何一种条码类型创建新的条码，如：

```
1. var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

以上例子创建了一个名为 **productBarcode** 的新变量，并且赋给它一个 **Barcode.UPCA** 的关联元组值(8,8590951226,3)。提供的“标识符”值在整数字中有一个下划线，使其便于阅读条形码。

同一个商品可以被分配给一个不同类型的条形码，如：

```
1. productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

这时，原始的 `Barcode.UPCA` 和其整数值被新的 `Barcode.QRCode` 和其字符串值所替代。条形码的常量和变量可以存储一个 `.UPCA` 或者一个 `.QRCode`（连同它的关联值），但是在任何指定时间只能存储其中之一。

像以前那样，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：

```
1. switch productBarcode {
2. case .UPCA(let numberSystem, let identifier, let check):
3.     println("UPC-
4.         A with value of \(numberSystem), \(identifier), \(check).")
5. case .QRCode(let productCode):
6.     println("QR code with value of \(productCode).")
7. }
```

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：

```
1. switch productBarcode {
2. case let .UPCA(numberSystem, identifier, check):
3.     println("UPC-
4.         A with value of \(numberSystem), \(identifier), \(check).")
5. case let .QRCode(productCode):
6.     println("QR code with value of \(productCode).")
7. }
```

原始值（Raw Values）

在关联值小节的条形码例子中演示了一个枚举的成员如何声明它们存储不同类型的关联值。作为关联值的替代，枚举成员可以被默认值（称为原始值）预先填充，其中这些原始值具有

相同的类型。

这里是一个枚举成员存储原始 `ASCII` 值的例子：

```
1. enum ASCIIControlCharacter: Character {  
2.     case Tab = "\t"  
3.     case LineFeed = "\n"  
4.     case CarriageReturn = "\r"  
5. }
```

在这里，称为 `ASCIIControlCharacter` 的枚举的原始值类型被定义为字符型 `Character`，并被设置了一些比较常见的 `ASCII` 控制字符。字符值的描述请详见字符串和字符章节。

注意，原始值和关联值是不相同的。当你开始在你的代码中定义枚举的时候原始值是被预先填充的值，像上述三个 `ASCII` 码。对于一个特定的枚举成员，它的原始值始终是相同的。关联值是当你在创建一个基于枚举成员的新常量或变量时才会被设置，并且每次当你这么做的时候，它的值可以是不同的。

原始值可以是字符串，字符，或者任何整型值或浮点型值。每个原始值在它的枚举声明中必须是唯一的。当整型值被用于原始值，如果其他枚举成员没有值时，它们会自动递增。

下面的枚举是对之前 `Planet` 这个枚举的一个细化，利用原始整型值来表示每个 `planet` 在太阳系中的顺序：

```
1. enum Planet: Int {  
2.     case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune  
3. }
```

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：

```
1. let earthsOrder = Planet.Earth.toRaw()  
2. // 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。这个例子通过原始值 7

识别 Uranus:

```
1. let possiblePlanet = Planet.fromRaw(7)
2. // 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`”。

如果你试图寻找一个位置为 9 的行星，通过 `fromRaw` 返回的可选 `Planet` 值将是 `nil`:

```
1. let positionToFind = 9
2. if let somePlanet = Planet.fromRaw(positionToFind) {
3.     switch somePlanet {
4.     case .Earth:
5.         println("Mostly harmless")
6.     default:
7.         println("Not a safe place for humans")
8.     }
9. } else {
10.    println("There isn't a planet at position \(positionToFind)")
11. }
12. // prints "There isn't a planet at position 9"
```

这个范例使用可选绑定（optional binding），通过原始值 9 试图访问一个行星。`if let somePlanet = Planet.fromRaw(9)` 语句获得一个可选 `Planet`，如果可选 `Planet` 可以被获得，把 `somePlanet` 设置成该可选 `Planet` 的内容。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。

2.9. 类和结构体

本篇译者：JaySurplus（[Github 主页](#)），敬请勘误。

类和结构体是人们构建代码所用的一种通用且灵活的构造体。为了在类和结构体中实现各种功能，我们必须严格按照对于常量，变量以及函数所规定的语法规则来定义属性和添加方法。

与其他编程语言所不同的是，**Swift** 并不要求你为自定义类和结构去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类或者结构体，系统将会自动生成面向其它代码的外部接口。

注意：通常一个类的实例被称为对象。然而在 **Swift** 中，类和结构体的关系要比在其他语言中更加的密切，本章中所讨论的大部分功能都可以用在类和结构体上。因此，我们会主要使用实例而不是对象。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 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

更多信息请参见 属性，方法，下标，初始过程，扩展，和协议。

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

更多信息请参见继承，类型转换，初始化，和自动引用计数。

注意：结构体总是通过被复制的方式在代码中传递，因此请不要使用引用计数。

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：

```
1. class SomeClass {
2.     // class definition goes here
3. }
4. struct SomeStructure {
5.     // structure definition goes here
6. }
```

注意：在你每次定义一个新类或者结构体的时候，实际上你是有效地定义了一个新的 **Swift** 类型。因此请使用 **UpperCamelCase** 这种方式来命名（如 **SomeClass** 和 **SomeStructure** 等），以便符合标准 **Swift** 类型的大写命名风格（如 **String**，**Int** 和 **Bool**）。相反的，请使用 **lowerCamelCase** 这种方式为属性和方法命名（如 **framerate** 和 **incrementCount**），以便和类区分。

以下是定义结构体和定义类的示例：

```
1. struct Resolution {
2.     var width = 0
3.     var height = 0
4. }
5. class VideoMode {
6.     var resolution = Resolution()
7.     var interlaced = false
8.     var frameRate = 0.0
9.     var name: String?
10. }
```

在上面的示例中我们定义了一个名为 **Resolution** 的结构体，用来描述一个显示器的像素分辨率。这个结构体包含了两个名为 **width** 和 **height** 的储存属性。储存属性是捆绑和储存在

类或结构体中的常量或变量。当这两个属性被初始化为整数 0 的时候，它们会被推断为 `Int` 类型。

在上面的示例中我们还定义了一个名为 `VideoMode` 的类，用来描述一个视频显示器的特定模式。这个类包含了四个储存属性变量。第一个是分辨率，它被初始化为一个新的 `Resolution` 结构体的实例，具有 `Resolution` 的属性类型。新 `VideoMode` 实例同时还会初始化其它三个属性，它们分别是，初始值为 `false` (意为“non-interlaced video”) 的 `inteflaced`，回放帧率初始值为 0.0 的 `frameRate` 和值为可选 `String` 的 `name`。`name` 属性会被自动赋予一个默认值 `nil`，意为“没有 `name` 值”，因它是一个可选类型。

类和结构体实例

`Resolution` 结构体和 `VideoMode` 类的定义仅描述了什么是 `Resolution` 和 `VideoMode`。它们并没有描述一个特定的分辨率（`resolution`）或者视频模式（`video mode`）。为了描述一个特定的分辨率或者视频模式，我们需要生成一个它们的实例。

生成结构体和类实例的语法非常相似：

```
1. let someResolution = Resolution()
2. let someVideoMode = VideoMode()
```

结构体和类都使用初始化器语法来生成新的实例。初始化器语法的最简单形式是在结构体或者类的类型名称后跟随一个空括弧，如 `Resolution()` 或 `VideoMode()`。通过这种方式所创建的类或者结构体实例，其属性均会被初始化为默认值。构造过程章节会对类和结构体的初始化进行更详细的讨论。

属性访问

通过使用点语法（`dot syntax`），你可以访问实例中所含有的属性。其语法规则是，实例名后面紧跟属性名，两者通过点号（`.`）连接：

```
1. println("The width of someResolution is \(someResolution.width)")
2. // 输出 "The width of someResolution is 0"
```

在上面的例子中，`someResolution.width` 引用 `someResolution` 的 `width` 属性，返回 `width` 的初始值 0。

你也可以访问子属性，如何 `VideoMode` 中 `Resolution` 属性的 `width` 属性：

```
1. println("The width of someVideoMode is \(someVideoMode.resolution.width) ")
2. // 输出 "The width of someVideoMode is 0"
```

你也可以使用点语法为属性变量赋值：

```
1. someVideoMode.resolution.width = 1280
2. println("The width of someVideoMode is now \(someVideoMode.resolution.width) ")
3. // 输出 "The width of someVideoMode is now 1280"
```

注意：与 `Objective-C` 语言不同的是，`Swift` 允许直接设置结构体属性的子属性。上面的最后一个例子，就是直接设置了 `someVideoMode` 中 `resolution` 属性的 `width` 这个子属性，以上操作并不需要从新设置 `resolution` 属性。

结构体类型的成员逐一初始化器

//Memberwise Initializers for structure Types

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：

```
1. let vga = resolution (width:640, height: 480)
```

与结构体不同，类实例没有默认的成员逐一初始化器。构造过程章节会对初始化器进行更详细的讨论。

结构体和枚举是值类型

值类型被赋予给一个变量，常数或者本身被传递给一个函数的时候，实际上操作的是其的拷贝。

在之前的章节中，我们已经大量使用了值类型。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。

在 **Swift** 中，所有的结构体和枚举都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型属性，在代码中传递的时候都会被复制。

请看下面这个示例，其使用了前一个示例中 **Resolution** 结构体：

```
1. let hd = Resolution(width: 1920, height: 1080)
2. var cinema = hd
```

在以上示例中，声明了一个名为 **hd** 的常量，其值为一个初始化为全高清视频分辨率(1920 像素宽，1080 像素高)的 **Resolution** 实例。

然后示例中又声明了一个名为 **cinema** 的变量，其值为之前声明的 **hd**。因为 **Resolution** 是一个结构体，所以 **cinema** 的值其实是 **hd** 的一个拷贝副本，而不是 **hd** 本身。尽管 **hd** 和 **cinema** 有着相同的宽(width)和高(height)属性，但是在后台中，它们是两个完全不同的实例。

下面，为了符合数码影院放映的需求(2048 像素宽，1080 像素高)，**cinema** 的 **width** 属性需要作如下修改：

```
1. cinema.width = 2048
```

这里，将会显示 **cinema** 的 **width** 属性确已改为了 2048：

```
1. println("cinema is now \(cinema.width) pixels wide")
2. // 输出 "cinema is now 2048 pixels wide"
```

然而，初始的 **hd** 实例中 **width** 属性还是 1920：

```
1. println("hd is still \(hd.width) pixels wide")
2. // 输出 "hd is still 1920 pixels wide"
```

在将 `hd` 赋予给 `cinema` 的时候，实际上是将 `hd` 中所储存的值(values)进行拷贝，然后将拷贝的数据储存到新的 `cinema` 实例中。结果就是两个完全独立的实例碰巧包含有相同的数值。由于两者相互独立，因此将 `cinema` 的 `width` 修改为 2048 并不会影响 `hd` 中的宽(width)。

枚举也遵循相同的行为准则：

```
1. enum CompassPoint {
2.     case North, South, East, West
3. }
4. var currentDirection = CompassPoint.West
5. let rememberedDirection = currentDirection
6. currentDirection = .East
7. if rememberedDirection == .West {
8.     println("The remembered direction is still .West")
9. }
10. // 输出 "The remembered direction is still .West"
```

上例中 `rememberedDirection` 被赋予了 `currentDirection` 的值(value)，实际上它被赋予的是值(value)的一个拷贝。赋值过程结束后再修改 `currentDirection` 的值并不影响 `rememberedDirection` 所储存的原始值(value)的拷贝。

类是引用类型

与值类型不同，引用类型在被赋予到一个变量，常量或者被传递到一个函数时，操作的并不是其拷贝。因此，引用的是已存在的实例本身而不是其拷贝。

请看下面这个示例，其使用了之前定义的 `VideoMode` 类：

```
1. let tenEighty = VideoMode()
2. tenEighty.resolution = hd
3. tenEighty.interlaced = true
4. tenEighty.name = "1080i"
5. tenEighty.frameRate = 25.0
```

以上示例中，声明了一个名为 `tenEighty` 的常量，其引用了一个 `VideoMode` 类的新实例。在之前的示例中，这个视频模式(video mode)被赋予了 HD 分辨率(1920*1080)的一个拷贝(hd)。同时设置为交错(interlaced),命名为“1080i”。最后，其帧率是 25.0 帧每秒。

然后，`tenEighty` 被赋予名为 `alsoTenEighty` 的新常量，同时对 `alsoTenEighty` 的帧率进行修改：

```
1. let alsoTenEighty = tenEighty
2. alsoTenEighty.frameRate = 30.0
```

因为类是引用类型，所以 `tenEighty` 和 `alsoTenEighty` 实际上引用的是相同的 `VideoMode` 实例。换句话说，它们只是同一个实例的两种叫法。

下面，通过查看 `tenEighty` 的 `frameRate` 属性，我们会发现它正确的显示了基本 `VideoMode` 实例的新帧率，其值为 30.0：

```
1. println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
2. // 输出 "The frameRate property of theEighty is now 30.0"
```

需要注意的是 `tenEighty` 和 `alsoTenEighty` 被声明为常量(constants)而不是变量。然而你依然可以改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate`,因为这两个常量本身不会改变。它们并不储存这个 `VideoMode` 实例，在后台仅仅是对 `VideoMode` 实例的引用。所以，改变的是被引用的基础 `VideoMode` 的 `frameRate` 参数，而不改变常量的值。

恒等运算符

因为类是引用类型，有可能有多个常量和变量在后台同时引用某一个类实例。(对于结构体和枚举来说，这并不成立。因为它们作值类型，在被赋予到常量，变量或者传递到函数时，总是会被拷贝。)

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：

```
1. if tenEighty === alsoTenTighty {
2.     println("tenTighty and alsoTenEighty refer to the same Resolution
   instance.")
3. }
4. //输出
   出 "tenEighty and alsoTenEighty refer to the same Resolution instance.
   "
```

请注意“等价于”(用三个等号表示，===) 与“等于”(用两个等号表示，==)的不同：

“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”，判定时要遵照类设计者定义定义的评判标准，因此相比于“相等”，这是一种更加合适的叫法。

当你在定义你的自定义类和结构体的时候，你有义务来决定判定两个实例“相等”的标准。

在章节运算符函数(Operator Functions)中将会详细介绍实现自定义“等于”和“不等于”运算符的流程。

指针

如果你有 C, C++ 或者 Objective-C 语言的经验，那么你也许会知道这些语言使用指针来引用内存中的地址。一个 Swift 常量或者变量引用一个引用类型的实例与 C 语言中的指针类似，不同的是并不直接指向内存中的某个地址，而且也不要求你使用星号(*)来表明你在创建一个引用。Swift 中这些引用与其它的常量或变量的定义方式相同。

类和结构体的选择

在你的代码中，你可以使用类和结构体来定义你的自定义数据类型。

然而，结构体实例总是通过值传递，类实例总是通过引用传递。这意味两者适用不同的任务。当你的在考虑一个工程项目的数据构造和功能的时候，你需要决定每个数据构造是定义成类还是结构体。

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。

集合(Collection)类型的赋值和拷贝行为

Swift 中数组(**Array**)和字典(**Dictionary**)类型均以结构体的形式实现。然而当数组被赋予一个常量或变量，或被传递给一个函数或方法时，其拷贝行为与字典和其它结构体有些许不同。

以下对数组和结构体的行为描述与对 **NSArray** 和 **NSDictionary** 的行为描述在本质上不同，后者是以类的形式实现，前者是以结构体的形式实现。**NSArray** 和 **NSDictionary** 实例总是以对已有实例引用,而不是拷贝的方式被赋值和传递。

注意：以下是对于数组，字典，字符串和其它值的拷贝的描述。在你的代码中，拷贝好像是确实是在有拷贝行为的地方产生过。然而，在 **Swift** 的后台中，只有确有必要，实际(**actual**)拷贝才会被执行。**Swift** 管理所有的值拷贝以确保性能最优化的性能，所以你也没有必要去避免赋值以保证最优性能。(实际赋值由系统管理优化)

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，这个字典会即会在赋值或调用发生时被拷贝。在章节结构体和枚举是值类型中将会对此过程进行详细介绍。

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。

下面的示例定义了一个名为 `ages` 的字典，其中储存了四个人的名字和年龄。`ages` 字典被赋予了一个名为 `copiedAges` 的新变量，同时 `ages` 在赋值的过程中被拷贝。赋值结束后，`ages` 和 `copiedAges` 成为两个相互独立的字典。

```
1. var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]
2. var copiedAges = ages
```

这个字典的键(keys)是字符串(String)类型，值(values)是整(Int)类型。这两种类型在 Swift 中都是值类型(value types)，所以当字典被拷贝时，两者都会被拷贝。

我们可以通过改变一个字典中的年龄值(age value)，检查另一个字典中所对应的值，来证明 `ages` 字典确实是被拷贝了。如果在 `copiedAges` 字典中将 `Peter` 的值设为 24，那么 `ages` 字典仍然会返回修改前的值 23：

```
1. copiedAges["Peter"] = 24
2. println(ages["Peter"])
3. // 输出 "23"
```

数组的赋值和拷贝行为

在 Swift 中，数组(Arrays)类型的赋值和拷贝行为要比字典(Dictionary)类型的复杂的多。当操作数组内容时，数组(Array)能提供接近 C 语言的的性能，并且拷贝行为只有在必要时才会发生。

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(append),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,参见章节[集合(collection)类型的赋值与复制行为](#assignment_and_copy_behavior_for_collection_types)。

下面的示例将一个整数(Int)数组赋给了一个名为 **a** 的变量,继而又被赋给了变量 **b** 和 **c**:

```
1. var a = [1, 2, 3]
2. var b = a
3. var c = a
```

我们可以在 **a,b,c** 上使用下标语法以得到数组的第一个元素:

```
1. println(a[0])
2. // 1
3. println(b[0])
4. // 1
5. println(c[0])
6. // 1
```

如果通过下标语法修改数组中某一元素的值,那么 **a,b,c** 中的相应值都会发生改变。请注意当你用下标语法修改某一值时,并没有拷贝行为伴随发生,因为下表语法修改值时没有改变数组长度的可能:

```
1. a[0] = 42
2. println(a[0])
3. // 42
4. println(b[0])
5. // 42
6. println(c[0])
7. // 42
```

然而，当你给 **a** 附加新元素时，数组的长度会改变。当附加元素这一事件发生时，**Swift** 会创建这个数组的一个拷贝。从此以后，**a** 将会是原数组的一个独立拷贝。

拷贝发生后，如果再修改 **a** 中元素值的话，**a** 将会返回与 **b**、**c** 不同的结果，因为后两者引用的是原来的数组：

```
1. a.append(4)
2. a[0] = 777
3. println(a[0])
4. // 777
5. println(b[0])
6. // 42
7. println(c[0])
8. // 42
```

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）

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。当数组仅被一个变量引用时，则不会有拷贝发生。

在上一个示例的最后，**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** 变成一个唯一拷贝：

```
1. 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：

```
1. b[0] = -105
2. println(a[0])
3. // 77
4. println(b[0])
5. // -105
```

```
6. println(c[0])
7. // 42
```

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。

下面这个示例使用了“恒等于(identical to)”运算符(`===`) 来判定 **b** 和 **c** 是否共用相同的数组元素：

```
1. if b === c {
2.     println("b and c still share the same array elements.")
3. } else {
4.     println("b and c now refer to two independent sets of array elements.")
5. }
6.
7. // 输出 "b and c now refer to two independent sets of array elements."
```

此外，我们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。下面这个示例中，比较了 **b** 的两个相等的子数组，并且确定了这两个子数组都引用相同的元素：

```
1. if b[0...1] === b[0...1] {
2.     println("These two subarrays share the same elements.")
3. } else {
4.     println("These two subarrays do not share the same elements.")
5. }
6. // 输出 "These two subarrays share the same elements."
```

强制复制数组

我们通过调用数组的 `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 copy),并且返回一个包含此拷贝的新数组。

下面这个示例中定义了一个 `names` 数组，其包含了七个人名。还定义了一个 `copiedNames` 变量，用以储存在 `names` 上调用 `copy` 方法所返回的结果：

```
1. var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]
2. var copiedNames = names.copy
```

我们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断 `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. copiedName[0] = "Mo"
2. println(name[0])
3. // 输出 "Mohsen"
```

注意：如果你仅需要确保你对数组的引用是唯一引用，请调用 `unshare` 方法，而不是 `copy` 方法。`unshare` 方法仅会在确有必要时才会创建数组拷贝。`copy` 方法会在任何时候都创建一个新的拷贝，即使引用已经是唯一引用。

2.10. 属性

本篇译者：shinyzhu，敬请勘误。

属性将值跟特定的类、结构或枚举关联。存储属性存储常量或变量作为实例的一部分，计算属性计算（而不是存储）一个值。计算属性可以用于类、结构体和枚举里，存储属性只能用于类和结构体。

存储属性和计算属性通常用于特定类型的实例，但是，属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性监视器来监控属性值的变化，以此来触发一个自定义的操作。属性监视器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量，存储属性可以是变量存储属性（用关键字 **var** 定义），也可以是常量存储属性（用关键字 **let** 定义）。

可以在定义存储属性的时候指定默认值，请参考构造过程一章的默认属性值一节。也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值，请参考构造过程一章的在初始化阶段修改常量存储属性一节。

下面的例子定义了一个名为 **FixedLengthRange** 的结构体，他描述了一个在创建后无法修改值域宽度的区间：

```
1. struct FixedLengthRange {
2.     var firstValue: Int
3.     let length: Int
4. }
5. var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
6. // 该区间表示整数 0, 1, 2
7. rangeOfThreeItems.firstValue = 6
8. // 该区间现在表示整数 6, 7, 8
```

FixedLengthRange 的实例包含一个名为 **firstValue** 的变量存储属性和一个名为 **length** 的常量存储属性。在上面的例子中，**length** 在创建实例的时候被赋值，因为它是一个常量存储属性，所以之后无法修改它的值。

常量和存储属性

如果创建了一个结构体的实例并赋值给一个常量，则无法修改实例的任何属性，即使定义了变量存储属性：

```
1. let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
2. // 该区间表示整数 0, 1, 2, 3
3. rangeOfFourItems.firstValue = 6
4. // 尽管 firstValue 是变量属性，这里还是会报错
```

因为 **rangeOfFourItems** 声明成了常量（用 **let** 关键字），即使 **firstValue** 是一个变量属性，也无法再修改它了。

这种行为是由于结构体（**struct**）属于值类型。当值类型的实例被声明为常量的时候，它的所有属性也就成了常量。

属于引用类型的类（**class**）则不一样，把一个引用类型的实例赋给一个常量后，仍然可以修改实例的变量属性。

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。

延迟属性很有用，当属性的值依赖于在实例的构造过程结束前无法知道具体值的外部因素时，或者当属性的值需要复杂或大量计算时，可以只在需要的时候来计算它。

下面的例子使用了延迟存储属性来避免复杂类的不必要的初始化。例子中定义了 **DataImporter** 和 **DataManager** 两个类，下面是部分代码：

```
1. class DataImporter {
2.     /*
3.     DataImporter 是一个将外部文件中的数据导入的类。
4.     这个类的初始化会消耗不少时间。
5.     */
6.     var fileName = "data.txt"
7.     // 这是提供数据导入功能
8. }
9.
10. class DataManager {
11.     @lazy var importer = DataImporter()
12.     var data = String[]()
13.     // 这是提供数据管理功能
14. }
```

```
15.  
16. let manager = DataManager()  
17. manager.data += "Some data"  
18. manager.data += "Some more data"  
19. // DataImporter 实例的 importer 属性还没有被创建
```

DataManager 类包含一个名为 **data** 的存储属性，初始值是一个空的字符串（**String**）数组。虽然没有写出全部代码，**DataManager** 类的目的是管理和提供对这个字符串数组的访问。

DataManager 的一个功能是从文件导入数据，该功能由 **DataImporter** 类提供，**DataImporter** 需要消耗不少时间完成初始化：因为它的实例在初始化时可能要打开文件，还要读取文件内容到内存。

DataManager 也可能不从文件中导入数据。所以当 **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。

由于使用了 **@lazy**，**importer** 属性只有在第一次被访问的时候才被创建。比如访问它的属性 **fileName** 时：

```
1. println(manager.importer.fileName)  
2. // DataImporter 实例的 importer 属性现在被创建了  
3. // 输出 "data.txt"
```

存储属性和实例变量

如果您有过 **Objective-C** 经验，应该知道有两种方式在类实例存储值和引用。对于属性来说，也可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。**Swift** 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。一个类型中属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

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。

```
1. struct Point {
2.     var x = 0.0, y = 0.0
3. }
4. struct Size {
5.     var width = 0.0, height = 0.0
6. }
7. struct Rect {
8.     var origin = Point()
9.     var size = Size()
10.    var center: Point {
11.        get {
12.            let centerX = origin.x + (size.width / 2)
13.            let centerY = origin.y + (size.height / 2)
14.            return Point(x: centerX, y: centerY)
15.        }
16.        set(newCenter) {
17.            origin.x = newCenter.x - (size.width / 2)
18.            origin.y = newCenter.y - (size.height / 2)
19.        }
20.    }
21. }
22. var square = Rect(origin: Point(x: 0.0, y: 0.0),
23.    size: Size(width: 10.0, height: 10.0))
24. let initialSquareCenter = square.center
25. square.center = Point(x: 15.0, y: 15.0)
26. println("square.origin is now at \(square.origin.x), \(square.origin.y)")
27. // 输出 "square.origin is now at (10.0, 10.0)"
```

这个例子定义了 3 个几何形状的结构体：

Point 封装了一个(x, y)的坐标

Size 封装了一个 width 和 height

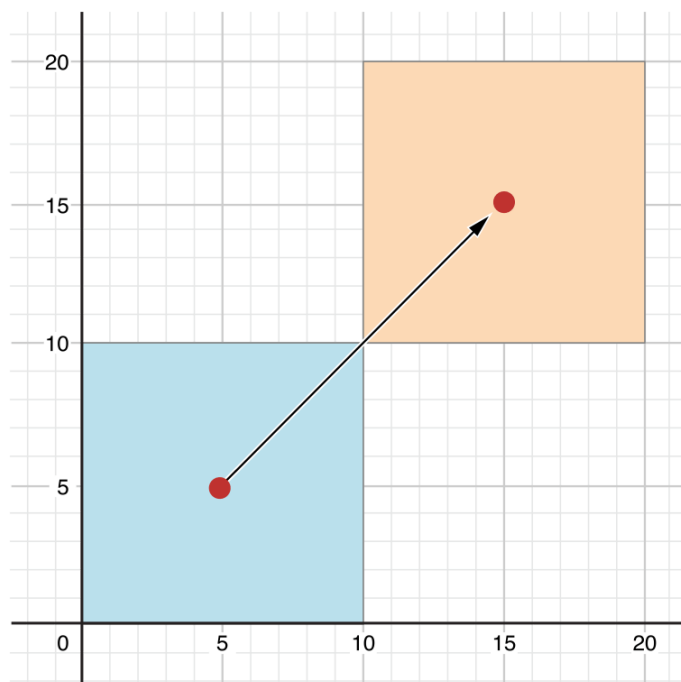
Rect 表示一个有原点和尺寸的矩形

Rect 也提供了一个名为 **center** 的计算属性。一个矩形的中心点可以从原点和尺寸来算出，所以不需要将它以显式声明的 **Point** 来保存。**Rect** 的计算属性 **center** 提供了自定义的 **getter** 和 **setter** 来获取和设置矩形的中心点，就像它有一个存储属性一样。

例子中接下来创建了一个名为 **square** 的 **Rect** 实例，初始值原点是(0,0)，宽度高度都是 10。如图所示蓝色正方形。

square 的 **center** 属性可以通过点运算符 (**square.center**) 来访问，这会调用 **getter** 来获取属性的值。跟直接返回已经存在的值不同，**getter** 实际上通过计算然后返回一个新的 **Point** 来表示 **square** 的中心点。如代码所示，它正确返回了中心点(5, 5)。

center 属性之后被设置了一个新的值(15, 15)，表示向右上方移动正方形到如图所示橙色正方形的位置。设置属性 **center** 的值会调用 **setter** 来修改属性 **origin** 的 **x** 和 **y** 的值，从而实现移动正方形到新的位置。



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：

```
1. struct AlternativeRect {
2.     var origin = Point()
3.     var size = Size()
4.     var center: Point {
5.         get {
6.             let centerX = origin.x + (size.width / 2)
7.             let centerY = origin.y + (size.height / 2)
8.             return Point(x: centerX, y: centerY)
9.         }
10.        set {
11.            origin.x = newValue.x - (size.width / 2)
12.            origin.y = newValue.y - (size.height / 2)
13.        }
14.    }
15. }
```

只读计算属性

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：

```
1. struct Cuboid {
2.     var width = 0.0, height = 0.0, depth = 0.0
3.     var volume: Double {
4.         return width * height * depth
5.     }
6. }
```

```
5.     }
6. }
7. let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
8. println("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
9. // 输出 "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个名为 **Cuboid** 的结构体，表示三维空间的立方体，包含 **width**、**height** 和 **depth** 属性，还有一个名为 **volume** 的只读计算属性用来返回立方体的体积。设置 **volume** 的值毫无意义，因为通过 **width**、**height** 和 **depth** 就能算出 **volume**。然而，**Cuboid** 提供一个只读计算属性来让外部用户直接获取体积是很有用的。

属性监视器

属性监视器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性监视器，甚至新的值和现在的值相同的时候也不例外。

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。属性重载请参考继承一章的重载。

注意：不需要为无法重载的计算属性添加属性监视器，因为可以通过 **setter** 直接监控和响应值的变化。

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。

这里是一个 `willSet` 和 `didSet` 的实际例子，其中定义了一个名为 `StepCounter` 的类，用来统计当人步行时的总步数，可以跟计步器或其他日常锻炼的统计装置的输入数据配合使用。

```
1. class StepCounter {
2.     var totalSteps: Int = 0 {
3.         willSet(newTotalSteps) {
4.             println("About to set totalSteps to \(newTotalSteps)")
5.         }
6.         didSet {
7.             if totalSteps > oldValue {
8.                 println("Added \(totalSteps - oldValue) steps")
9.             }
10.        }
11.    }
12. }
13. let stepCounter = StepCounter()
14. stepCounter.totalSteps = 200
15. // About to set totalSteps to 200
16. // Added 200 steps
17. stepCounter.totalSteps = 360
18. // About to set totalSteps to 360
19. // Added 160 steps
20. stepCounter.totalSteps = 896
21. // About to set totalSteps to 896
22. // Added 536 steps
```

`StepCounter` 类定义了一个 `Int` 类型的属性 `totalSteps`，它是一个存储属性，包含 `willSet` 和 `didSet` 监视器。

当 `totalSteps` 设置新值的时候，它的 `willSet` 和 `didSet` 监视器都会被调用，甚至当新的值和现在的值完全相同也会调用。

例子中的 `willSet` 监视器将表示新值的参数自定义为 `newTotalSteps`，这个监视器只是简单的将新的值输出。

`didSet` 监视器在 `totalSteps` 的值改变后被调用，它把新的值和旧的值进行对比，如果总的步

数增加了，就输出一个消息表示增加了多少步。`didSet` 没有提供自定义名称，所以默认值 `oldValue` 表示旧值的参数名。

注意：如果在 `didSet` 监视器里为属性赋值，这个值会替换监视器之前设置的值。

全局变量和局部变量

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。

前面章节提到的全局或局部变量都属于存储型变量，跟存储属性类似，它提供特定类型的存储空间，并允许读取和写入。

另外，在全局或局部范围都可以定义计算型变量和为存储型变量定义监视器，计算型变量跟计算属性一样，返回一个计算的值而不是存储值，声明格式也完全一样。

注意：全局的常量或变量都是延迟计算的，跟延迟存储属性相似，不同的地方在于，全局的常量或变量不需要标记 `@lazy` 特性；局部范围的常量或变量不会延迟计算。

类型属性

实例的属性属于一个特定类型实例，每次类型实例化后都拥有自己的一套属性值，实例之间的属性相互独立。

也可以为类型本身定义属性，不管类型有多少个实例，这些属性都只有唯一一份。这种属性就是类型属性。

类型属性用于定义特定类型所有实例共享的数据，比如所有实例都能用的一个常量（就像 C 语言中的静态常量），或者所有实例都能访问的一个变量（就像 C 语言中的静态变量）。

对于值类型（指结构体和枚举）可以定义存储型和计算型类型属性，对于类（`class`）则只能定义计算型类型属性。

值类型的存储型类型属性可以是变量或常量，计算型类型属性跟实例的计算属性一样定义成变量属性。

注意：跟实例的存储属性不同，必须给存储型类型属性指定默认值，因为类型本身无法在初始化过程中使用构造器给类型属性赋值。

类型属性语法

在 **C** 或 **Objective-C** 中，静态常量和静态变量的定义是通过特定类型加上 **global** 关键字。在 **Swift** 编程语言中，类型属性是作为类型定义的一部分写在类型最外层的花括号内，因此它的作用范围也就在类型支持的范围内。

使用关键字 **static** 来定义值类型的类型属性，关键字 **class** 来为类（**class**）定义类型属性。下面的例子演示了存储型和计算型类型属性的语法：

```
1. struct SomeStructure {
2.     static var storedTypeProperty = "Some value."
3.     static var computedTypeProperty: Int {
4.         // 这里返回一个 Int 值
5.     }
6. }
7. enum SomeEnumeration {
8.     static var storedTypeProperty = "Some value."
9.     static var computedTypeProperty: Int {
10.        // 这里返回一个 Int 值
11.    }
12. }
13. class SomeClass {
14.     class var computedTypeProperty: Int {
15.         // 这里返回一个 Int 值
16.     }
17. }
```

注意：例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟实例计算属性的语法类似。

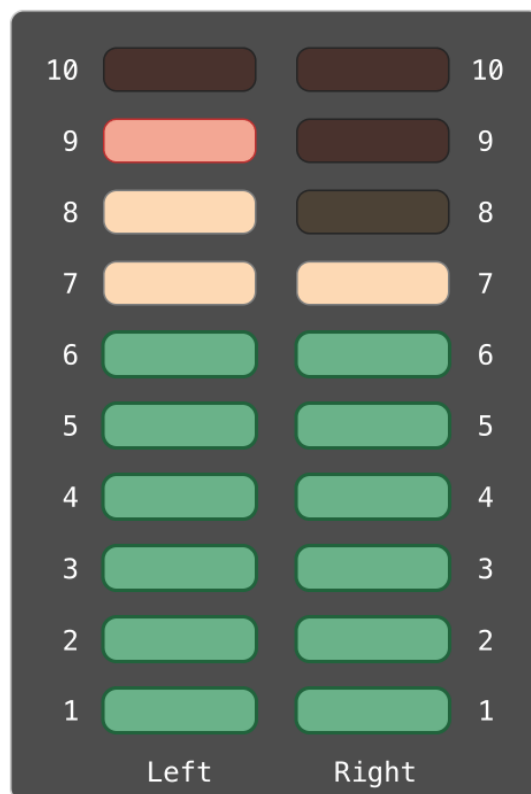
获取和设置类型属性的值

跟实例的属性一样，类型属性的访问也是通过点运算符来进行，但是，类型属性是通过类型本身来获取和设置，而不是通过实例。比如：

```
1. println(SomeClass.computedTypeProperty)
2. // 输出 "42"
3.
4. println(SomeStructure.storedTypeProperty)
5. // 输出 "Some value."
6. SomeStructure.storedTypeProperty = "Another value."
7. println(SomeStructure.storedTypeProperty)
8. // 输出 "Another value."
```

下面的例子定义了一个结构体，使用两个存储型类型属性来表示多个声道的声音电平值，每个声道有一个 0 到 10 之间的整数表示声音电平值。

后面的图表展示了如何联合使用两个声道来表示一个立体声的声音电平值。当声道的电平值是 0，没有一个灯会亮；当声道的电平值是 10，所有灯点亮。本图中，左声道的电平是 9，右声道的电平是 7。



上面所描述的声道模型使用 `AudioChannel` 结构体来表示：

```

1. struct AudioChannel {
2.     static let thresholdLevel = 10
3.     static var maxInputLevelForAllChannels = 0
4.     var currentLevel: Int = 0 {
5.     didSet {
6.         if currentLevel > AudioChannel.thresholdLevel {
7.             // 将新电平值设置为阈值
8.             currentLevel = AudioChannel.thresholdLevel
9.         }
10.        if currentLevel > AudioChannel.maxInputLevelForAllChannels {
11.            // 存储当前电平值作为新的最大输入电平
12.            AudioChannel.maxInputLevelForAllChannels = currentLevel
13.        }
14.    }
15. }
16. }
```

结构 `AudioChannel` 定义了 2 个存储型类型属性来实现上述功能。第一个是 `thresholdLevel`，

表示声音电平的最大上限阈值，它是一个取值为 10 的常量，对所有实例都可见，如果声音电平高于 10，则取最大上限值 10（见后面描述）。

第二个类型属性是变量存储型属性 `maxInputLevelForAllChannels`，它用来表示所有 `AudioChannel` 实例的电平值的最大值，初始值是 0。

`AudioChannel` 也定义了一个名为 `currentLevel` 的实例存储属性，表示当前声道现在的电平值，取值为 0 到 10。

属性 `currentLevel` 包含 `didSet` 属性监视器来检查每次新设置后的属性值，有如下两个检查：

如果 `currentLevel` 的新值大于允许的阈值 `thresholdLevel`，属性监视器将 `currentLevel` 的值限定为阈值 `thresholdLevel`。

如果修正后的 `currentLevel` 值大于任何之前任意 `AudioChannel` 实例中的值，属性监视器将新值保存在静态属性 `maxInputLevelForAllChannels` 中。

注意：在第一个检查过程中，`didSet` 属性监视器将 `currentLevel` 设置成了不同的值，但这时不会再次调用属性监视器。

可以使用结构体 `AudioChannel` 来创建表示立体声系统的两个声道 `leftChannel` 和 `rightChannel`：

```
1. var leftChannel = AudioChannel()
2. var rightChannel = AudioChannel()
```

如果将左声道的电平设置成 7，类型属性 `maxInputLevelForAllChannels` 也会更新成 7：

```
1. leftChannel.currentLevel = 7
2. println(leftChannel.currentLevel)
3. // 输出 "7"
4. println(AudioChannel.maxInputLevelForAllChannels)
5. // 输出 "7"
```

如果试图将右声道的电平设置成 11，则会将右声道的 `currentLevel` 修正到最大值 10，同时 `maxInputLevelForAllChannels` 的值也会更新到 10：

```
1. rightChannel.currentLevel = 11
2. println(rightChannel.currentLevel)
3. // 输出 "10"
4. println(AudioChannel.maxInputLevelForAllChannels)
5. // 输出 "10"
```

2.11. 方法

本篇译者：pp-prog ([Github 主页](#))，敬请勘误。

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。类型方法与 **Objective-C** 中的类方法（**class methods**）相似。

结构体和枚举能够定义方法是 **Swift** 与 **C/Objective-C** 的主要区别之一。在 **Objective-C** 中，类是唯一能定义方法的类型。但在 **Swift** 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活的在你创建的类型（类/结构体/枚举）上定义方法。

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。实例方法的语法与函数完全一致，详情参见函数。

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。

下面的例子，定义一个很简单的类 **Counter**，**Counter** 能被用来对一个动作发生的次数进行计数：

```
1. class Counter {
2.     var count = 0
3.     func increment() {
4.         count++
5.     }
6.     func incrementBy(amount: Int) {
7.         count += amount
8.     }
9.     func reset() {
10.        count = 0
11.    }
12. }
```

Counter 类定义了三个实例方法：

increment 让计数器按一递增；

incrementBy(amount: Int) 让计数器按一个指定的整数值递增；

reset 将计数器重置为 0。

Counter 这个类还声明了一个可变属性 **count**，用它来保持对当前计数器值的追踪。

和调用属性一样，用点语法（**dot syntax**）调用实例方法：

```
1. let counter = Counter()
2. // 初始计数值是 0
3. counter.increment()
4. // 计数值现在是 1
5. counter.incrementBy(5)
6. // 计数值现在是 6
7. counter.reset()
8. // 计数值现在是 0
```

方法的局部参数名称和外部参数名称(Local and External Parameter Names for Methods)

函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用），详情参见函数的外部参数名。方法参数也一样（因为方法就是函数，只是这个函数与某个类型相关联了）。但是，方法和函数的局部名称和外部名称的默认行为是不一样的。

Swift 中的方法和 Objective-C 中的方法极其相似。像在 Objective-C 中一样，Swift 中方法的名称通常用一个介词指向方法的第一个参数，比如：with, for, by 等等。前面的 Counter 类的例子中 incrementBy 方法就是这样的。介词的使用让方法在被调用时能像一个句子一样被解读。和函数参数不同，对于方法的参数，Swift 使用不同的默认处理方式，这可以让方法命名规范更容易写。

具体来说，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。这个约定与典型的命名和调用约定相适应，与你在写 Objective-C 的方法时很相似。这个约定还让表达式方法在调用时不需要再限定参数名称。

看看下面这个 Counter 的另一个版本（它定义了一个更复杂的 incrementBy 方法）：

```
1. class Counter {
2.     var count: Int = 0
3.     func incrementBy(amount: Int, numberOfTimes: Int) {
4.         count += amount * numberOfTimes
5.     }
6. }
```

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：

```
1. let counter = Counter()
2. counter.incrementBy(5, numberOfTimes: 3)
3. // counter value is now 15
```

你不必为第一个参数值再定义一个外部变量名：因为从函数名 incrementBy 已经能很清楚地看出它的作用。但是第二个参数，就要被一个外部参数名称所限定，以便在方法被调用时明确它的作用。

这种默认的行为能够有效的处理方法（method），类似于在参数 numberOfTimes 前写一个井号（#）：

```
1. func incrementBy(amount: Int, #numberOfTimes: Int) {
```

```
2. count += amount * numberOfTimes
3. }
```

这种默认行为使上面代码意味着：在 **Swift** 中定义方法使用了与 **Objective-C** 同样的语法风格，并且方法将以自然表达式的方式被调用。

修改方法的外部参数名称(Modifying External Parameter Name Behavior for Methods)

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。

上面例子中的 **increment** 方法还可以这样写：

```
1. func increment() {
2.     self.count++
3. }
```

实际上，你不必在你的代码里面经常写 **self**。不论何时，只要在一个方法中使用一个已知的属性或者方法名称，如果你没有明确的写 **self**，**Swift** 假定你是指当前实例的属性或者方法。这种假定在上面的 **Counter** 中已经示范了：**Counter** 中的三个实例方法中都使用的是 **count**（而不是 **self.count**）。

使用这条规则的主要场景是实例方法的某个参数名称与实例的某个属性名称相同的时候。在这种情况下，参数名称享有优先权，并且在引用属性时必须使用一种更严格的方式。这时你可以使用 **self** 属性来区分参数名称和属性名称。

下面的例子中，**self** 消除方法参数 **x** 和实例属性 **x** 之间的歧义：

```
1. struct Point {
2.     var x = 0.0, y = 0.0
3.     func isToTheRightOfX(x: Double) -> Bool {
4.         return self.x > x
5.     }
6. }
7. let somePoint = Point(x: 4.0, y: 5.0)
8. if somePoint.isToTheRightOfX(1.0) {
9.     println("This point is to the right of the line where x == 1.0")
10. }
11. // 输出 "This point is to the right of the line where x == 1.0" (这个点
    在 x 等于 1.0 这条线的右边)
```

如果不使用 `self` 前缀，Swift 就认为两次使用的 `x` 都指的是名称为 `x` 的函数参数。

在实例方法中修改值类型(Modifying Value Types from Within Instance Methods)

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。

但是，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：

```
1. struct Point {
2.     var x = 0.0, y = 0.0
3.     mutating func moveByX(deltaX: Double, y deltaY: Double) {
4.         x += deltaX
5.         y += deltaY
6.     }
7. }
8. var somePoint = Point(x: 1.0, y: 1.0)
9. somePoint.moveByX(2.0, y: 3.0)
10. println("The point is now at \(somePoint.x), \(somePoint.y)")
11. // 输出 "The point is now at (3.0, 4.0)"
```

上面的 `Point` 结构体定义了一个变异方法 (mutating method) `moveByX`，`moveByX` 用来移

动点。`moveByX` 方法在被调用时修改了这个点，而不是返回一个新的点。方法定义时加上 `mutating` 关键字,这才让方法可以修改值类型的属性。

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

```
1. let fixedPoint = Point(x: 3.0, y: 3.0)
2. fixedPoint.moveByX(2.0, y: 3.0)
3. // this will report an error
```

在变异方法中给 `self` 赋值(Assigning to `self` Within a Mutating Method)

变异方法能够赋给隐含属性 `self` 一个全新的实例。上面 `Point` 的例子可以用下面的方式改写：

```
1. struct Point {
2.     var x = 0.0, y = 0.0
3.     mutating func moveByX(deltaX: Double, y deltaY: Double) {
4.         self = Point(x: x + deltaX, y: y + deltaY)
5.     }
6. }
```

新版的变异方法 `moveByX` 创建了一个新的结构（它的 `x` 和 `y` 的值都被设定为目标值）。调用这个版本的方法和调用上个版本的最终结果是一样的。

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：

```
1. enum TriStateSwitch {
2.     case Off, Low, High
3.     mutating func next() {
4.         switch self {
5.             case Off:
6.                 self = Low
7.             case Low:
8.                 self = High
9.             case High:
10.                self = Off
11.         }
12.     }
```

```
13. }
14. var ovenLight = TriStateSwitch.Low
15. ovenLight.next()
16. // ovenLight 现在等于 .High
17. ovenLight.next()
18. // ovenLight 现在等于 .Off
```

上面的例子中定义了一个三态开关的枚举。每次调用 `next` 方法时，开关在不同的电源状态（Off, Low, High）之前循环切换。

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`。

注意：在 **Objective-C** 里面，你只能为 **Objective-C** 的类定义类型方法（**type-level methods**）。在 **Swift** 中，你可以为所有的类、结构体和枚举定义类型方法：每一个类型方法都被它所支持的类型显式包含。

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：

```
1. class SomeClass {
2.     class func someTypeMethod() {
3.         // type method implementation goes here
4.     }
5. }
6. SomeClass.someTypeMethod()
```

在类型方法的方法体（**body**）中，`self` 指向这个类型本身，而不是类型的某个实例。对于结构体和枚举来说，这意味着你可以用 `self` 来消除静态属性和静态方法参数之间的歧义（类似于我们在前面处理实例属性和实例方法参数时做的那样）。

一般来说，任何未限定的方法和属性名称，将会来自于本类中另外的类型级别的方法和属性。

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。

下面的例子定义了一个名为 **LevelTracker** 结构体。它监测玩家的游戏发展情况（游戏的不同层次或阶段）。这是一个单人游戏，但也可以存储多个玩家在同一设备上的游戏信息。

游戏初始时，所有的游戏等级（除了等级 1）都被锁定。每次有玩家完成一个等级，这个等级就对这个设备上的所有玩家解锁。**LevelTracker** 结构体用静态属性和方法监测游戏的哪个等级已经被解锁。它还监测每个玩家的当前等级。

```
1. struct LevelTracker {
2.     static var highestUnlockedLevel = 1
3.     static func unlockLevel(level: Int) {
4.         if level > highestUnlockedLevel { highestUnlockedLevel = level }
5.     }
6.     static func levelIsUnlocked(level: Int) -> Bool {
7.         return level <= highestUnlockedLevel
8.     }
9.     var currentLevel = 1
10.    mutating func advanceToLevel(level: Int) -> Bool {
11.        if LevelTracker.levelIsUnlocked(level) {
12.            currentLevel = level
13.            return true
14.        } else {
15.            return false
16.        }
17.    }
18. }
```

LevelTracker 监测玩家的已解锁的最高等级。这个值被存储在静态属性 **highestUnlockedLevel** 中。

LevelTracker 还定义了两个类型方法与 **highestUnlockedLevel** 配合工作。第一个类型方法是 **unlockLevel**：一旦新等级被解锁，它会更新 **highestUnlockedLevel** 的值。第二个类型方法是 **levelIsUnlocked**：如果某个给定的等级已经被解锁，它将返回 **true**。（注意：尽管我们没有使用类似 **LevelTracker.highestUnlockedLevel** 的写法，这个类型方法还是能够访问静

态属性 `highestUnlockedLevel`)

除了静态属性和类型方法, `LevelTracker` 还监测每个玩家的进度。它用实例属性 `currentLevel` 来监测玩家当前的等级。

为了便于管理 `currentLevel` 属性, `LevelTracker` 定义了实例方法 `advanceToLevel`。这个方法会在更新 `currentLevel` 之前检查所请求的新等级是否已经解锁。`advanceToLevel` 方法返回布尔值以指示是否能够设置 `currentLevel`。

下面, `Player` 类使用 `LevelTracker` 来监测和更新每个玩家的发展进度:

```
1. class Player {
2.     var tracker = LevelTracker()
3.     let playerName: String
4.     func completedLevel(level: Int) {
5.         LevelTracker.unlockLevel(level + 1)
6.         tracker.advanceToLevel(level + 1)
7.     }
8.     init(name: String) {
9.         playerName = name
10.    }
11. }
```

`Player` 类创建一个新的 `LevelTracker` 实例来监测这个用户的发展进度。他提供了 `completedLevel` 方法: 一旦玩家完成某个指定等级就调用它。这个方法为所有玩家解锁下一等级, 并且将当前玩家的进度更新为下一等级。(我们忽略了 `advanceToLevel` 返回的布尔值, 因为之前调用 `LevelTracker.unlockLevel` 时就知道了这个等级已经被解锁了)。

你还可以为一个新的玩家创建一个 `Player` 的实例, 然后看这个玩家完成等级一时发生了什么:

```
1. var player = Player(name: "Argyrios")
2. player.completedLevel(1)
3. println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
4. // 输出 "highest unlocked level is now 2" (最高等级现在是 2)
```

如果你创建了第二个玩家，并尝试让他开始一个没有被任何玩家解锁的等级，那么这次设置玩家当前等级的尝试将会失败：

```
1. player = Player(name: "Beto")
2. if player.tracker.advanceToLevel(6) {
3.     println("player is now on level 6")
4. } else {
5.     println("level 6 has not yet been unlocked")
6. }
7. // 输出 "level 6 has not yet been unlocked" (等级 6 还没被解锁)
```

2.12. 附属脚本

本篇译者：siemenliu ([Github 主页](#))，敬请勘误。

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。举例来说，用附属脚本访问一个数组 (Array) 实例中的元素可以这样写 `someArray[index]`，访问字典 (Dictionary) 实例中的元素可以这样写 `someDictionary[key]`。

对于同一个目标可以定义多个附属脚本，通过索引值类型的不同来进行重载，而且索引值的个数可以是多个。

译者：这里附属脚本重载在本小节中原文并没有任何演示

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`：

```
1. subscript(index: Int) -> Int {
2.     get {
```

```
3.         // 返回与入参匹配的 Int 类型的值
4.     }
5.
6.     set(newValue) {
7.         // 执行赋值操作
8.     }
9. }
```

`newValue` 的类型必须和附属脚本定义的返回类型相同。与计算型属性相同的是 `set` 的入参声明 `newValue` 就算不写，在 `set` 代码块中依然可以使用默认的 `newValue` 这个变量来访问新赋的值。

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：

```
1. subscript(index: Int) -> Int {
2.     // 返回与入参匹配的 Int 类型的值
3. }
```

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。

```
1. struct TimesTable {
2.     let multiplier: Int
3.     subscript(index: Int) -> Int {
4.         return multiplier * index
5.     }
6. }
7. let threeTimesTable = TimesTable(multiplier: 3)
8. println("3 的 6 倍是 \(threeTimesTable[6])")
9. // 输出 "3 的 6 倍是 18"
```

在上例中，通过 `TimesTable` 结构体创建了一个用来表示索引值三倍的实例。数值 `3` 作为结构体构造函数入参初始化实例成员 `multiplier`。

你可以通过附属脚本来得到结果，比如 `threeTimesTable[6]`。这句话访问了 `threeTimesTable` 的第六个元素，返回 `18` 或者 `6` 的 `3` 倍。

注意：`TimesTable` 例子是基于一个固定的数学公式。它并不适合开放写权限来对 `threeTimesTable[someIndex]` 进行赋值操作，这也是为什么附属脚本只定义为只读的原因。

附属脚本用法

根据使用场景不同附属脚本也具有不同的含义。通常附属脚本是用来访问集合（**collection**），列表（**list**）或序列（**sequence**）中元素的快捷方式。你可以在你自己特定的类或结构体中自由的实现附属脚本来提供合适的功能。

例如，**Swift** 的字典（**Dictionary**）实现了通过附属脚本来对其实例中存放的值进行存取操作。在附属脚本中使用和字典索引相同类型的值，并且把一个字典值类型的值赋值给这个附属脚本来为字典设置：

```
1. var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2. numberOfLegs["bird"] = 2
```

上例定义一个名为 **numberOfLegs** 的变量并用一个字典字面量初始化出了包含三对键值的字典实例。**numberOfLegs** 的字典存放值类型推断为 **Dictionary<String, Int>**。字典实例创建完成之后通过附属脚本的方式将整型值 **2** 赋值到字典实例的索引为 **bird** 的位置中。

更多关于字典（**Dictionary**）附属脚本的信息请参考读取和修改字典

注意：**Swift** 中字典的附属脚本实现中，在 **get** 部分返回值是 **Int?**，上例中的 **numberOfLegs** 字典通过下边返回的是一个 **Int?**或者说“可选的 **int**”，不是每个字典的索引都能得到一个整型值，对于没有设过值的索引的访问返回的结果就是 **nil**；同样想要从字典实例中删除某个索引下的值也只需要给这个索引赋值为 **nil** 即可。

附属脚本选项

附属脚本允许任意数量的入参索引，并且每个入参类型也没有限制。附属脚本的返回值也可以是任何类型。附属脚本可以使用变量参数和可变参数，但使用写入读出（**in-out**）参数或给参数设置默认值都是不允许的。

一个类或结构体可以根据自身需要提供多个附属脚本实现，在定义附属脚本时通过入参个类型进行区分，使用附属脚本时会自动匹配合适的附属脚本实现运行，这就是附属脚本的重载。

一个附属脚本入参是最常见的情况，但只要有合适的场景也可以定义多个附属脚本入参。如

下例定义了一个 **Matrix** 结构体，将呈现一个 **Double** 类型的二维矩阵。**Matrix** 结构体的附属脚本需要两个整型参数：

```
1. struct Matrix {
2.     let rows: Int, columns: Int
3.     var grid: Double[]
4.     init(rows: Int, columns: Int) {
5.         self.rows = rows
6.         self.columns = columns
7.         grid = Array(count: rows * columns, repeatedValue: 0.0)
8.     }
9.     func isValidForRow(row: Int, column: Int) -> Bool {
10.        return row >= 0 && row < rows && column >= 0 && column < columns
11.    }
12.    subscript(row: Int, column: Int) -> Double {
13.        get {
14.            assert(isValidForRow(row, column: column), "Index out of range")
15.            return grid[(row * columns) + column]
16.        }
17.        set {
18.            assert(isValidForRow(row, column: column), "Index out of range")
19.            grid[(row * columns) + column] = newValue
20.        }
21.    }
22. }
```

Matrix 提供了一个两个入参的构造方法，入参分别是 **rows** 和 **columns**，创建了一个足够容纳 **rows * columns** 个数的 **Double** 类型数组。为了存储，将数组的大小和数组每个元素初始值 **0.0**，都传入数组的构造方法中来创建一个正确大小的新数组。关于数组的构造方法和析构方法请参考创建并且构造一个数组。

你可以通过传入合适的 **row** 和 **column** 的数量来构造一个新的 **Matrix** 实例：

```
1. var matrix = Matrix(rows: 2, columns: 2)
```

上例中创建了一个新的两行两列的 **Matrix** 实例。在阅读顺序从左上到右下的 **Matrix** 实例中

的数组实例 `grid` 是矩阵二维数组的扁平化存储:

```
1. // 示意图
2. grid = [0.0, 0.0, 0.0, 0.0]
3.
4.      col0    col1
5. row0  [0.0,   0.0,
6. row1   0.0,   0.0]
```

将值赋给带有 `row` 和 `column` 附属脚本的 `matrix` 实例表达式可以完成赋值操作, 附属脚本入参使用逗号分割

```
1. matrix[0, 1] = 1.5
2. matrix[1, 0] = 3.2
```

上面两条语句分别让 `matrix` 的右上值为 1.5, 坐下值为 3.2:

```
1. [0.0, 1.5,
2.   3.2, 0.0]
```

`Matrix` 附属脚本的 `getter` 和 `setter` 中同时调用了附属脚本入参的 `row` 和 `column` 是否有效的判断。为了方便进行断言, `Matrix` 包含了一个名为 `indexIsValid` 的成员方法, 用来确认入参的 `row` 或 `column` 值是否会造成数组越界:

```
1. func indexIsValidForRow(row: Int, column: Int) -> Bool {
2.     return row >= 0 && row < rows && column >= 0 && column < columns
3. }
```

断言在附属脚本越界时触发:

```
1. let someValue = matrix[2, 2]
2. // 断言将会触发, 因为 [2, 2] 已经超过了 matrix 的最大长度
```

2.13. 继承

本篇译者: Hawstein ([Github 主页](#)), 敬请勘误。

一个类可以**继承**（inherit）另一个类的方法（methods），属性（property）和其它特性。当一个类继承其它类时，继承类叫子类（subclass），被继承类叫超类（或父类，superclass）。在 Swift 中，继承是区分「类」与其它类型的一个基本特征。

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。

可以为类中继承来的属性添加属性观察器（property observer），这样一来，当属性值改变时，类就会被通知到。可以为任何属性添加属性观察器，无论它原本被定义为存储型属性（stored property）还是计算型属性（computed property）。

定义一个基类（Base class）

不继承于其它类的类，称之为基类（base class）。

注意：Swift 中的类并不是从一个通用的基类继承而来。如果你不为你定义的类指定一个超类的话，这个类就自动成为基类。

下面的例子定义了一个叫 **Vehicle** 的基类。这个基类声明了两个对所有车辆都通用的属性（numberOfWheels 和 maxPassengers）。这些属性在 description 方法中使用，这个方法返回一个 String 类型的，对车辆特征的描述：

```
1. class Vehicle {
2.     var numberOfWheels: Int
3.     var maxPassengers: Int
4.     func description() -> String {
5.         return "\(numberOfWheels) wheels; up to \(maxPassengers) passengers"
6.     }
7.     init() {
8.         numberOfWheels = 0
9.         maxPassengers = 1
10.    }
11. }
```

Vehicle 类定义了构造器（**initializer**）来设置属性的值。构造器会在构造过程一节中详细介绍，这里我们做一下简单介绍，以便于讲解子类中继承来的属性如何被修改。

构造器用于创建某个类型的一个新实例。尽管构造器并不是方法，但在语法上，两者很相似。构造器的工作是准备新实例以供使用，并确保实例中的所有属性都拥有有效的初始化值。

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：

```
1. init() {  
2.     // 执行构造过程  
3. }
```

如果要创建一个 **Vehicle** 类的新实例，使用构造器语法调用上面的初始化器，即类名后面跟一个空的小括号：

```
1. let someVehicle = Vehicle()
```

这个 **Vehicle** 类的构造器为任意的一辆车设置一些初始化属性值（**numberOfWheels = 0** 和 **maxPassengers = 1**）。

Vehicle 类定义了车辆的共同特性，但这个类本身并没太大用处。为了使它更为实用，你需要进一步细化它来描述更具体的车辆。

子类生成（Subclassing）

子类生成（**Subclassing**）指的是在一个已有类的基础上创建一个新的类。子类继承超类的特性，并且可以优化或改变它。你还可以为子类添加新的特性。

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：

```
1. class SomeClass: SomeSuperclass {  
2.     // 类的定义  
3. }
```

下一个例子，定义一个更具体的车辆类叫 **Bicycle**。这个新类是在 **Vehicle** 类的基础上创建起来。因此你需要将 **Vehicle** 类放在 **Bicycle** 类后面，用冒号分隔。

我们可以将这读作：

“定义一个新的类叫 **Bicycle**，它继承了 **Vehicle** 的特性”；

```
1. class Bicycle: Vehicle {
2.     init() {
3.         super.init()
4.         numberOfWheels = 2
5.     }
6. }
```

Bicycle 是 **Vehicle** 的子类，**Vehicle** 是 **Bicycle** 的超类。新的 **Bicycle** 类自动获得 **Vehicle** 类的特性，比如 **maxPassengers** 和 **numberOfWheels** 属性。你可以在子类中定制这些特性，或添加新的特性来更好地描述 **Bicycle** 类。

Bicycle 类定义了一个构造器来设置它定制的特性（自行车只有 2 个轮子）。**Bicycle** 的构造器调用了它父类 **Vehicle** 的构造器 **super.init()**，以此确保在 **Bicycle** 类试图修改那些继承来的属性前 **Vehicle** 类已经初始化过它们了。

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

Vehicle 类中 **maxPassengers** 的默认值对自行车来说已经是正确的，因此在 **Bicycle** 的构造器中并没有改变它。而 **numberOfWheels** 原来的值对自行车来说是不正确的，因此在初始化器中将它更改为 2。

Bicycle 不仅可以继承 **Vehicle** 的属性，还可以继承它的方法。如果你创建了一个 **Bicycle** 类的实例，你就可以调用它继承来的 **description** 方法，并且可以看到，它输出的属性值已经发生了变化：

```
1. let bicycle = Bicycle()
2. println("Bicycle: \(bicycle.description())")
3. // Bicycle: 2 wheels; up to 1 passengers
```

子类还可以继续被其它类继承：

```
1. class Tandem: Bicycle {
2.     init() {
3.         super.init()
4.         maxPassengers = 2
5.     }
6. }
```

上面的例子创建了 **Bicycle** 的一个子类：双人自行车（**tandem**）。**Tandem** 从 **Bicycle** 继承了两个属性，而这两个属性是 **Bicycle** 从 **Vehicle** 继承而来的。**Tandem** 并不修改轮子的数量，因为它仍是一辆自行车，有 2 个轮子。但它需要修改 **maxPassengers** 的值，因为双人自行车可以坐两个人。

注意：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。

创建一个 **Tandem** 类的实例，打印它的描述，即可看到它的属性已被更新：

```
1. let tandem = Tandem()
2. println("Tandem: \(tandem.description())")
3. // Tandem: 2 wheels; up to 2 passengers
```

注意，**Tandem** 类也继承了 **description** 方法。一个类的实例方法会被这个类的所有子类继承。

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。

override 关键字会提醒 **Swift** 编译器去检查该类的超类（或其中一个父类）是否有匹配重写版本的声明。这个检查可以确保你的重写定义是正确的。

访问超类的方法，属性及附属脚本

当你在子类中重写超类的方法，属性或附属脚本时，有时在你的重写版本中使用已经存在的超类实现会大有裨益。比如，你可以优化已有实现的行为，或在一个继承来的变量中存储一个修改过的值。

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

在子类中，你可以重写继承来的实例方法或类方法，提供一个定制或替代的方法实现。

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：

```
1. class Car: Vehicle {
2.     var speed: Double = 0.0
3.     init() {
4.         super.init()
5.         maxPassengers = 5
6.         numberOfWheels = 4
7.     }
8.     override func description() -> String {
9.         return super.description() + "; "
10.            + "traveling at \(speed) mph"
11.     }
12. }
```

`Car` 声明了一个新的存储型属性 `speed`，它是 `Double` 类型的，默认值是 `0.0`，表示“时速是 0 英里”。`Car` 有自己的初始化器，它将乘客的最大数量设为 5，轮子数量设为 4。

`Car` 重写了继承来的 `description` 方法，它的声明与 `Vehicle` 中的 `description` 方法一致，声明前面加上了 `override` 关键字。

`Car` 中的 `description` 方法并非完全自定义，而是通过 `super.description` 使用了超类 `Vehicle` 中的 `description` 方法，然后再追加一些额外的信息，比如汽车的当前速度。

如果你创建一个 `Car` 的新实例，并打印 `description` 方法的输出，你就会发现描述信息已经发生了改变：

```
1. let car = Car()
2. println("Car: \(car.description())")
3. // Car: 4 wheels; up to 5 passengers; traveling at 0.0 mph
```

重写属性

你可以重写继承来的实例属性或类属性，提供自己定制的 `getter` 和 `setter`，或添加属性观察者使重写的属性观察属性值什么时候发生改变。

重写属性的 Getters 和 Setters

你可以提供定制的 `getter`（或 `setter`）来重写任意继承来的属性，无论继承来的属性是存储型的还是计算型的属性。子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名字和类型。你在重写一个属性时，必需将它的名字和类型都写出来。这样才能使编译器去检查你重写的属性是与超类中同名同类型的属性相匹配的。

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。

以下的例子定义了一个新类，叫 `SpeedLimitedCar`，它是 `Car` 的子类。类 `SpeedLimitedCar` 表示安装了限速装置的车，它的最高速度只能达到 40mph。你可以通过重写继承来的 `speed` 属性来实现这个速度限制：

```
1. class SpeedLimitedCar: Car {
2.     override var speed: Double {
3.         get {
4.             return super.speed
5.         }
6.         set {
7.             super.speed = min(newValue, 40.0)
8.         }
9.     }
10. }
```

当你设置一个 `SpeedLimitedCar` 实例的 `speed` 属性时，属性 `setter` 的实现会去检查新值与限制值 40mph 的大小，它会将超类的 `speed` 设置为 `newValue` 和 40.0 中较小的那个。这两个值哪个较小由 `min` 函数决定，它是 `Swift` 标准库中的一个全局函数。`min` 函数接收两个或更多的数，返回其中最小的那个。

如果你尝试将 `SpeedLimitedCar` 实例的 `speed` 属性设置为一个大于 40mph 的数，然后打印 `description` 函数的输出，你会发现速度被限制在 40mph：

```
1. let limitedCar = SpeedLimitedCar()
2. limitedCar.speed = 60.0
3. println("SpeedLimitedCar: \(limitedCar.description())")
4. // SpeedLimitedCar: 4wheels; up to 5 passengers; traveling at 40.0 mph
```

重写属性观察器（Property Observer）

你可以在属性重写中为一个继承来的属性添加属性观察器。这样一来，当继承来的属性值发生改变时，你就会被通知到，无论那个属性原本是如何实现的。关于属性观察器的更多内容，请看属性观察器。

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。

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。

```
1. class AutomaticCar: Car {
2.     var gear = 1
3.     override var speed: Double {
4.         didSet {
5.             gear = Int(speed / 10.0) + 1
6.         }
7.     }
8.     override func description() -> String {
9.         return super.description() + " in gear \(gear)"
10.    }
11. }
```

当你设置 `AutomaticCar` 的 `speed` 属性，属性的 `didSet` 观察器就会自动地设置 `gear` 属性，为新的速度选择一个合适的挡位。具体来说就是，属性观察器将新的速度值除以 10，然后向下取得最接近的整数值，最后加 1 来得到档位 `gear` 的值。例如，速度为 10.0 时，档位为 1；速度为 35.0 时，档位为 4：

```
1. let automatic = AutomaticCar()
2. automatic.speed = 35.0
3. println("AutomaticCar: \(automatic.description())")
4. // AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph i
   n gear 4
```

防止重写

你可以通过把方法，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。（例如： `@final var`， `@final func`， `@final class func`，以及 `@final`

subscript)

如果你重写了 `final` 方法，属性或附属脚本，在编译时会报错。在扩展中，你添加到类里的方法，属性或附属脚本也可以在扩展的定义里标记为 `final`。

你可以通过在关键字 `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。

2.14. 构造过程

本篇译者：lifedim，敬请勘误。

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。

构造过程是通过定义构造器 (**Initializers**) 来实现的，这些构造器可以看做是用来创建特定类型实例的特殊方法。与 **Objective-C** 中的构造器不同，**Swift** 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器 (**deinitializer**) 在类实例释放之前执行特定的清除工作。想了解更多关于析构器的内容，请参考析构过程。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。存储型属性的值不能处于一个未知的状态。

你可以在构造器中为存储型属性赋初值，也可以在定义属性时为其设置默认值。以下章节将详细介绍这两种方法。

注意：当你为存储型属性设置默认值或者在构造器中为其赋值时，它们的值是被直接设置的，不会触发任何属性观测器 (**property observers**)。

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。

下面例子中定义了一个用来保存华氏温度的结构体 **Fahrenheit**，它拥有一个 **Double** 类型的存储型属性 **temperature**：

```
1. struct Fahrenheit {
2.     var temperature: Double
3.     init() {
4.         temperature = 32.0
5.     }
6. }
7.
8. var f = Fahrenheit()
9. println("The default temperature is \(f.temperature)° Fahrenheit")
10. // 输出 "The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个不带参数的构造器 `init`，并在里面将存储型属性 **temperature** 的值初始化为 **32.0**（华摄氏度下水的冰点）。

默认属性值

如前所述，你可以在构造器中为存储型属性设置初始值；同样，你也可以在属性声明时为其设置默认值。

注意：如果一个属性总是使用同一个初始值，可以为其设置一个默认值。无论定义默认值还是在构造器中赋值，最终它们实现的效果是一样的，只不过默认值跟属性构造过程结合的更紧密。使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型；同时，它也能让你充分利用默认构造器、构造器继承（后续章节将讲到）等特性。

你可以使用更简单的方式在定义结构体 **Fahrenheit** 时为属性 **temperature** 设置默认值：

```
1. struct Fahrenheit {
2.     var temperature = 32.0
3. }
```

定制化构造过程

你可以通过输入参数和可选属性类型来定制构造过程，也可以在构造过程中修改常量属性。这些都将在后面章节中提到。

构造参数

你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。

下面例子中定义了一个包含摄氏度温度的结构体 **Celsius**。它定义了两个不同的构造器：**init(fromFahrenheit:)**和 **init(fromKelvin:)**，二者分别通过接受不同刻度表示的温度值来创建新的实例：

```
1. struct Celsius {
2.     var temperatureInCelsius: Double = 0.0
3.     init(fromFahrenheit fahrenheit: Double) {
4.         temperatureInCelsius = (fahrenheit - 32.0) / 1.8
5.     }
6.     init(fromKelvin kelvin: Double) {
7.         temperatureInCelsius = kelvin - 273.15
8.     }
9. }
10.
11. let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
12. // boilingPointOfWater.temperatureInCelsius 是 100.0
13. let freezingPointOfWater = Celsius(fromKelvin: 273.15)
14. // freezingPointOfWater.temperatureInCelsius 是 0.0"
```

第一个构造器拥有一个构造参数，其外部名字为 **fromFahrenheit**，内部名字为 **fahrenheit**；第二个构造器也拥有一个构造参数，其外部名字为 **fromKelvin**，内部名字为

kelvin。这两个构造器都将唯一的参数值转换成摄氏温度值，并保存在属性 `temperatureInCelsius` 中。

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，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。

注意：如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线_来显示描述它的外部名，以此覆盖上面所说的默认行为。

以下例子中定义了一个结构体 **Color**，它包含了三个常量：**red**、**green** 和 **blue**。这些属性可以存储 0.0 到 1.0 之间的值，用来指示颜色中红、绿、蓝成分的含量。

Color 提供了一个构造器，其中包含三个 **Double** 类型的构造参数：

```
1. struct Color {
2.     let red = 0.0, green = 0.0, blue = 0.0
3.     init(red: Double, green: Double, blue: Double) {
4.         self.red    = red
5.         self.green  = green
6.         self.blue   = blue
7.     }
8. }
```

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。

```
1. let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

注意，如果不通过外部参数名字传值，你是没法调用这个构造器的。只要构造器定义了某个外部参数名，你就必须使用它，忽略它将导致编译错误：

```
1. let veryGreen = Color(0.0, 1.0, 0.0)
2. // 报编译时错误，需要外部名称
```

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。

下面例子中定义了类 **SurveyQuestion**，它包含一个可选字符串属性 **response**：

```
1. class SurveyQuestion {
2.     var text: String
3.     var response: String?
4.     init(text: String) {
5.         self.text = text
6.     }
7.     func ask() {
8.         println(text)
9.     }
10. }
11. let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
12. cheeseQuestion.ask()
13. // 输出 "Do you like cheese?"
14. cheeseQuestion.response = "Yes, I do like cheese."
```

调查问题在问题提出之后，我们才能得到回答。所以我们将属性回答 **response** 声明为 **String?** 类型，或者说是可选字符串类型 **optional String**。当 **SurveyQuestion** 实例化时，它将自动赋值为空 **nil**，表明暂时还不存在此字符串。

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。

注意：对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

你可以修改上面的 `SurveyQuestion` 示例，用常量属性替代变量属性 `text`，指明问题内容 `text` 在其创建之后不会再被修改。尽管 `text` 属性现在是常量，我们仍然可以在其类的构造器中修改它的值：

```
1. class SurveyQuestion {
2.     let text: String
3.     var response: String?
4.     init(text: String) {
5.         self.text = text
6.     }
7.     func ask() {
8.         println(text)
9.     }
10. }
11. let beetsQuestion = SurveyQuestion(text: "How about beets?")
12. beetsQuestion.ask()
13. // 输出 "How about beets?"
14. 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。

下面例子中创建了一个类 `ShoppingListItem`，它封装了购物清单中的某一项的属性：名字（`name`）、数量（`quantity`）和购买状态 `purchase state`。

```
1. class ShoppingListItem {
```

```
2.     var name: String?
3.     var quantity = 1
4.     var purchased = false
5. }
6. 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`）。上面例子中使用默认构造器创建了一个 `ShoppingListItem` 类的实例（使用 `ShoppingListItem()` 形式的构造器语法），并将其赋值给变量 `item`。

结构体的逐一成员构造器

除上面提到的默认构造器，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。

下面例子中定义了一个结构体 `Size`，它包含两个属性 `width` 和 `height`。`Swift` 可以根据这两个属性的初始赋值 `0.0` 自动推导出它们的类型 `Double`。

由于这两个存储型属性都有默认值，结构体 `Size` 自动获得了一个逐一成员构造器 `init(width:height:)`。你可以用它来为 `Size` 创建新的实例：

```
1. struct Size {
2.     var width = 0.0, height = 0.0
3. }
4. 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。这些责任将在后续章节类的继承和构造过程中介绍。

对于值类型，你可以使用 `self.init` 在自定义的构造器中引用其它的属于相同值类型的构造器。并且你只能在构造器内部调用 `self.init`。

注意，如果你为某个值类型定义了一个定制的构造器，你将无法访问到默认构造器（如果是结构体，则无法访问逐一对象构造器）。这个限制可以防止你在为值类型定义了一个更复杂的，完成了重要准备构造器之后，别人还是错误的使用了那个自动生成的构造器。

注意：假如你想通过默认构造器、逐一对象构造器以及你自己定制的构造器为值类型创建实例，我们建议你将自己定制的构造器写到扩展（**extension**）中，而不是跟值类型定义混在一起。想查看更多内容，请查看扩展章节。

下面例子将定义一个结构体 **Rect**，用来展现几何矩形。这个例子需要两个辅助的结构体 **Size** 和 **Point**，它们各自为其所有的属性提供了初始值 **0.0**。

```
1. struct Size {
2.     var width = 0.0, height = 0.0
3. }
4. struct Point {
5.     var x = 0.0, y = 0.0
6. }
```

你可以通过以下三种方式为 **Rect** 创建实例--使用默认的 **0** 值来初始化 **origin** 和 **size** 属性；使用特定的 **origin** 和 **size** 实例来初始化；使用特定的 **center** 和 **size** 来初始化。在下面 **Rect** 结构体定义中，我们为着三种方式提供了三个自定义的构造器：

```
1. struct Rect {
2.     var origin = Point()
3.     var size = Size()
4.     init() {}
```

```
5.     init(origin: Point, size: Size) {
6.         self.origin = origin
7.         self.size = size
8.     }
9.     init(center: Point, size: Size) {
10.        let originX = center.x - (size.width / 2)
11.        let originY = center.y - (size.height / 2)
12.        self.init(origin: Point(x: originX, y: originY), size: size)
13.    }
14. }
```

第一个 **Rect** 构造器 **init()**，在功能上跟没有自定义构造器时自动获得的默认构造器是一样的。这个构造器是一个空函数，使用一对大括号`{}`来描述，它没有执行任何定制的构造过程。调用这个构造器将返回一个 **Rect** 实例，它的 **origin** 和 **size** 属性都使用定义时的默认值 **Point(x: 0.0, y: 0.0)**和 **Size(width: 0.0, height: 0.0)**：

```
1. let basicRect = Rect()
2. // basicRect 的原点是 (0.0, 0.0)，尺寸是 (0.0, 0.0)
```

第二个 **Rect** 构造器 **init(origin:size:)**，在功能上跟结构体在没有自定义构造器时获得的逐一成员构造器是一样的。这个构造器只是简单的将 **origin** 和 **size** 的参数值赋给对应的存储型属性：

```
1. let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
2.    size: Size(width: 5.0, height: 5.0))
3. // originRect 的原点是 (2.0, 2.0)，尺寸是 (5.0, 5.0)
```

第三个 **Rect** 构造器 **init(center:size:)**稍微复杂一点。它先通过 **center** 和 **size** 的值计算出 **origin** 的坐标。然后再调用（或代理给）**init(origin:size:)**构造器来将新的 **origin** 和 **size** 值赋值到对应的属性中：

```
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)，尺寸是 (3.0, 3.0)
```

构造器 **init(center:size:)**可以自己将 **origin** 和 **size** 的新值赋值到对应的属性中。然而尽量利用现有的构造器和它所提供的功能来实现 **init(center:size:)**的功能，是更方便、更清晰和更直观的方法。

注意：如果你想用另外一种不需要自己定义 `init()` 和 `init(origin:size:)` 的方式来实现这个例子，请参考扩展。

类的继承和构造过程

类里面的所有存储型属性--包括所有继承自父类的属性--都必须在构造过程中设置初始值。

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。

指定构造器和便利构造器

指定构造器是类中最主要的构造器。一个指定构造器将初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。

每一个类都必须拥有至少一个指定构造器。在某些情况下，许多类通过继承了父类中的指定构造器而满足了这个条件。具体内容请参考后续章节自动构造器的继承。

便利构造器是类中比较次要的、辅助型的构造器。你可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入的实例。

你应当只在必要的时候为类提供便利构造器，比方说某种情况下通过使用便利构造器来快捷调用某个指定构造器，能够节省更多开发时间并让类的构造过程更清、晰明。

构造器链

为了简化指定构造器和便利构造器之间的调用关系，Swift 采用以下三条规则来限制构造器之间的代理调用：

规则 1

指定构造器必须调用其直接父类的指定构造器。

规则 2

便利构造器必须调用同一类中定义的其他构造器。

规则 3

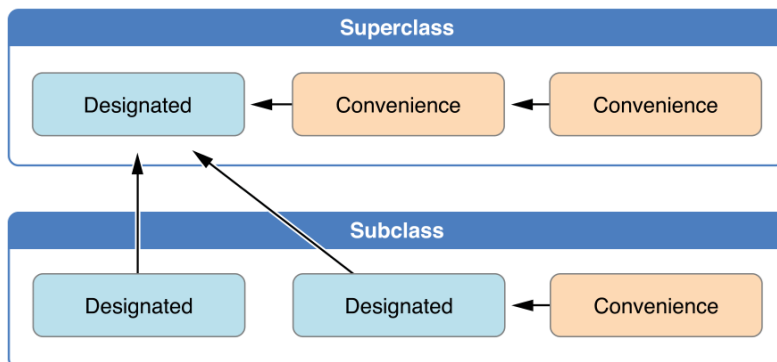
便利构造器必须最终以调用一个指定构造器结束。

一个更方便记忆的方法是：

指定构造器必须总是向上代理

便利构造器必须总是横向代理

这些规则可以通过下面图例来说明：

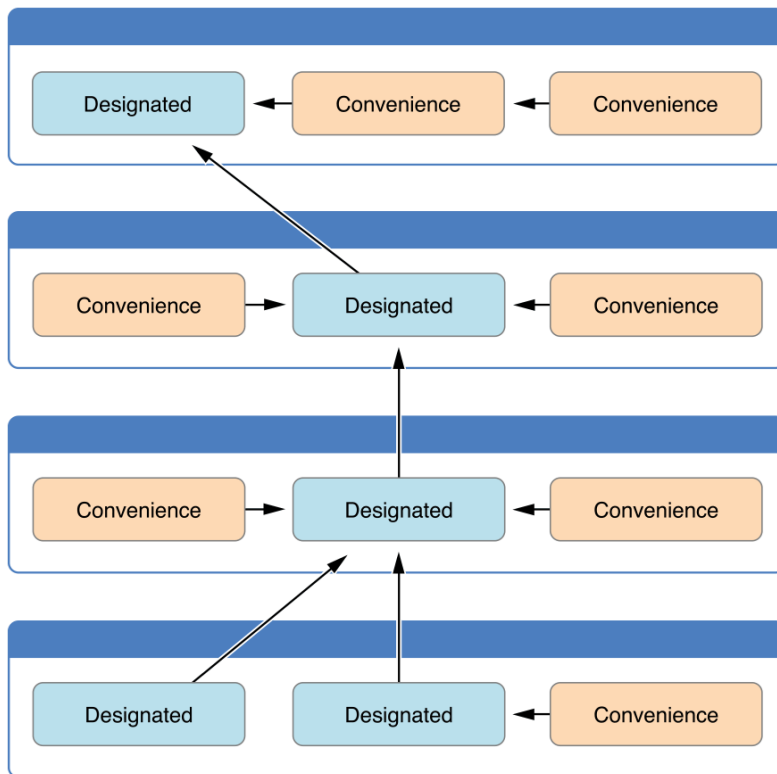


如图所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则 2 和 3。这个父类没有自己的父类，所以规则 1 没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则 2 和 3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则 1。

注意：这些规则不会影响使用时，如何用类去创建实例。任何上图中展示的构造器都可以用来完整创建对应类的实例。这些规则只在实现类的定义时有影响。

下面图例中展示了一种更复杂的类层级结构。它演示了指定构造器是如果在类层级中充当“管道”的作用，在类的构造器链上简化了类之间的内部关系。



两段式构造过程

Swift 中类的构造过程包含两个阶段。第一个阶段，每个存储型属性通过引入它们的类的构造器来设置初始值。当每一个存储型属性值被确定后，第二阶段开始，它给每个类一次机会在新实例准备使用之前进一步定制它们的存储型属性。

两段式构造过程的使用让构造过程更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化之前被访问；也可以防止属性被另外一个构造器意外地赋予不同的值。

注意：**Swift** 的两段式构造过程跟 **Objective-C** 中的构造过程类似。最主要的区别在于阶段 1，**Objective-C** 给每一个属性赋值 0 或空值（比如说 0 或 nil）。**Swift** 的构造流程则更加灵活，它允许你设置定制的初始值，并自如应对某些属性不能以 0 或 nil 作为合法默认值的情况。

Swift 编译器将执行 4 种有效的安全检查，以确保两段式构造过程能顺利完成：

安全检查 1

指定构造器必须保证它所在类引入的所有属性都必须先初始化完成，之后才能将其它构造任务向上代理给父类中的构造器。

如上所述，一个对象的内存只有在其所有存储型属性确定之后才能完全初始化。为了满足这一规则，指定构造器必须保证它所在类引入的属性在它往上代理之前先完成初始化。

安全检查 2

指定构造器必须先向上代理调用父类构造器，然后再为继承的属性设置新值。如果没这么做，指定构造器赋予的新值将被父类中的构造器所覆盖。

安全检查 3

便利构造器必须先代理调用同一类中的其它构造器，然后再为任意属性赋新值。如果没这么做，便利构造器赋予的新值将被同一类中其它指定构造器所覆盖。

安全检查 4

构造器在第一阶段构造完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用 `self` 的值。

以下是两段式构造过程中基于上述安全检查的构造流程展示：

阶段 1

某个指定构造器或便利构造器被调用；

完成新实例内存的分配，但此时内存还没有被初始化；

指定构造器确保其所在类引入的所有存储型属性都已赋初值。存储型属性所属的内存完成初始化；

指定构造器将调用父类的构造器，完成父类属性的初始化；

这个调用父类构造器的过程沿着构造器链一直往上执行，直到到达构造器链的最顶部；

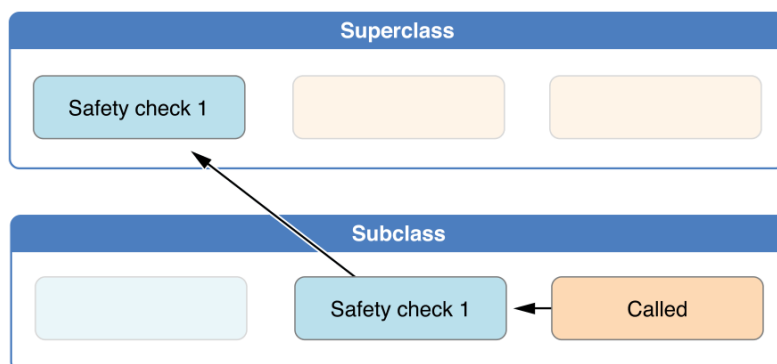
当到达了构造器链最顶部，且已确保所有实例包含的存储型属性都已经赋值，这个实例的内存被认为已经完全初始化。此时阶段 1 完成。

阶段 2

从顶部构造器链一直往下，每个构造器链中类的指定构造器都有机会进一步定制实例。构造器此时可以访问 `self`、修改它的属性并调用实例方法等等。

最终，任意构造器链中的便利构造器可以有机会定制实例和使用 `self`。

下图展示了在假定的子类和父类之间构造的阶段 1：



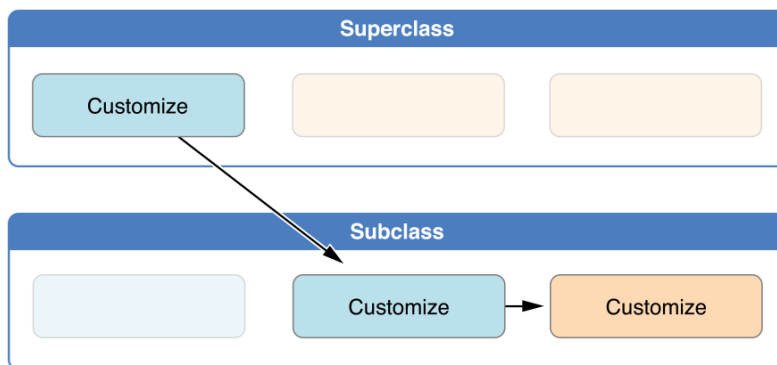
在这个例子中，构造过程从对子类中一个便利构造器的调用开始。这个便利构造器此时没法修改任何属性，它把构造任务代理给同一类中的指定构造器。

如安全检查 1 所示，指定构造器将确保所有子类的属性都有值。然后它将调用父类的指定构造器，并沿着造器链一直往上完成父类的构建过程。

父类中的指定构造器确保所有父类的属性都有值。由于没有更多的父类需要构建，也就无需继续向上做构建代理。

一旦父类中所有属性都有了初始值，实例的内存被认为是完全初始化，而阶段 1 也已完成。

以下展示了相同构造过程的阶段 2：



父类中的指定构造器现在有机会进一步来定制实例（尽管它没有这种必要）。

一旦父类中的指定构造器完成调用，子类的构造指定构造器可以执行更多的定制操作（同样，它也没有这种必要）。

最终，一旦子类的指定构造器完成调用，最开始被调用的便利构造器可以执行更多的定制操作。

构造器的继承和重载

跟 **Objective-C** 中的子类不同，**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。

假如你希望自定义的子类中能够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。

如果你重载的构造器是一个指定构造器，你可以在子类里重载它的实现，并在自定义版本的构造器中调用父类版本的构造器。

如果你重载的构造器是一个便利构造器，你的重载过程必须通过调用同一类中提供的其它指定构造器来实现。这一规则的详细内容请参考构造器链。

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

如上所述，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。

即使你在子类中添加了更多的便利构造器，这两条规则仍然适用。

注意：子类可以通过部分满足规则 2 的方式，使用子类便利构造器来实现父类的指定构造器。

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：

```
1. init(parameters) {  
2.     statements  
3. }
```

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：

```
1. convenience init(parameters) {  
2.     statements  
3. }
```

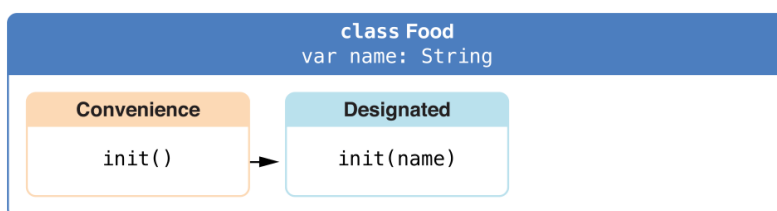
指定构造器和便利构造器实战

接下来的例子将在实战中展示指定构造器、便利构造器和自动构造器的继承。它定义了包含三个类 **Food**、**RecipeIngredient** 以及 **ShoppingListItem** 的类层次结构，并将演示它们的构造器是如何相互作用的。

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：

```
1. class Food {
2.     var name: String
3.     init(name: String) {
4.         self.name = name
5.     }
6.     convenience init() {
7.         self.init(name: "[Unnamed]")
8.     }
9. }
```

下图中展示了 **Food** 的构造器链：



类没有提供一个默认的逐一成员构造器，所以 **Food** 类提供了一个接受单一参数 **name** 的指定构造器。这个构造器可以使用一个特定的名字来创建新的 **Food** 实例：

```
1. let namedMeat = Food(name: "Bacon")
2. // namedMeat 的名字是 "Bacon"
```

Food 类中的构造器 **init(name: String)** 被定义为一个指定构造器，因为它能确保所有新 **Food** 实例的中存储型属性都被初始化。**Food** 类没有父类，所以 **init(name: String)** 构造器不需要调用 **super.init()** 来完成构造。

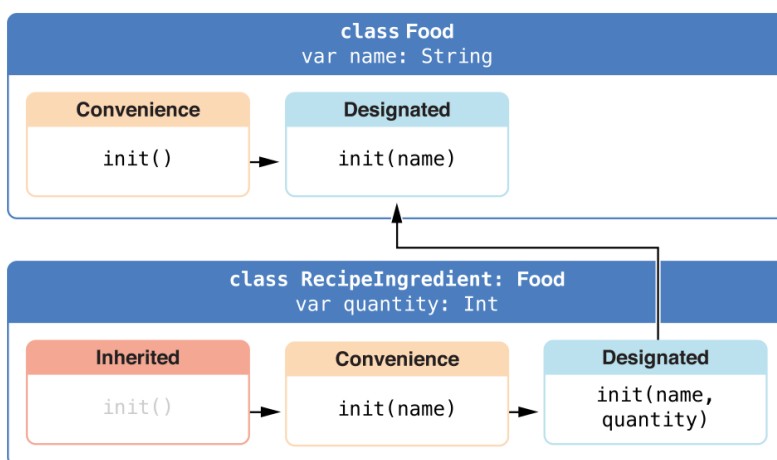
Food 类同样提供了一个没有参数的便利构造器 **init()**。这个 **init()** 构造器为新食物提供了一个默认的占位名字，通过代理调用同一类中定义的指定构造器 **init(name: String)** 并给参数 **name** 传值 **[Unnamed]** 来实现：

```
1. let mysteryMeat = Food()
2. // mysteryMeat 的名字是 [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：

```
1. class RecipeIngredient: Food {
2.     var quantity: Int
3.     init(name: String, quantity: Int) {
4.         self.quantity = quantity
5.         super.init(name: name)
6.     }
7.     convenience init(name: String) {
8.         self.init(name: name, quantity: 1)
9.     }
10. }
```

下图中展示了 `RecipeIngredient` 类的构造器链：



`RecipeIngredient` 类拥有一个指定构造器 `init(name: String, quantity: Int)`，它可以用来产生新 `RecipeIngredient` 实例的所有属性值。这个构造器一开始先将传入的 `quantity` 参数赋值给 `quantity` 属性，这个属性也是唯一在 `RecipeIngredient` 中新引入的属性。随后，构造器将任务向上代理给父类 `Food` 的 `init(name: String)`。这个过程满足两段式构造过程中的安全检查 1。

`RecipeIngredient` 也定义了一个便利构造器 `init(name: String)`，它只通过 `name` 来创建 `RecipeIngredient` 的实例。这个便利构造器假设任意 `RecipeIngredient` 实例的 `quantity` 为 1，所以不需要显示指明数量即可创建出实例。这个便利构造器的定义可以让创建实例更加方便和快捷，并且避免了使用重复的代码来创建多个 `quantity` 为 1 的 `RecipeIngredient` 实例。这个便利构造器只是简单的将任务代理给了同一类里提供的指定构造器。

注意，`RecipeIngredient` 的便利构造器 `init(name: String)` 使用了跟 `Food` 中指定构造器 `init(name: String)` 相同的参数。尽管 `RecipeIngredient` 这个构造器是便利构造器，`RecipeIngredient` 依然提供了对所有父类指定构造器的实现。因此，`RecipeIngredient` 也能自动继承了所有父类的便利构造器。

在这个例子中，`RecipeIngredient` 的父类是 `Food`，它有一个便利构造器 `init()`。这个构造器因此也被 `RecipeIngredient` 继承。这个继承的 `init()` 函数版本跟 `Food` 提供的版本是一样的，除了它是将任务代理给 `RecipeIngredient` 版本的 `init(name: String)` 而不是 `Food` 提供的版本。

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

```
1. let oneMysteryItem = RecipeIngredient()
2. let oneBacon = RecipeIngredient(name: "Bacon")
3. let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

类层级中第三个也是最后一个类是 `RecipeIngredient` 的子类，叫做 `ShoppingListItem`。这个类构建了购物单中出现的某一种调味料。

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：

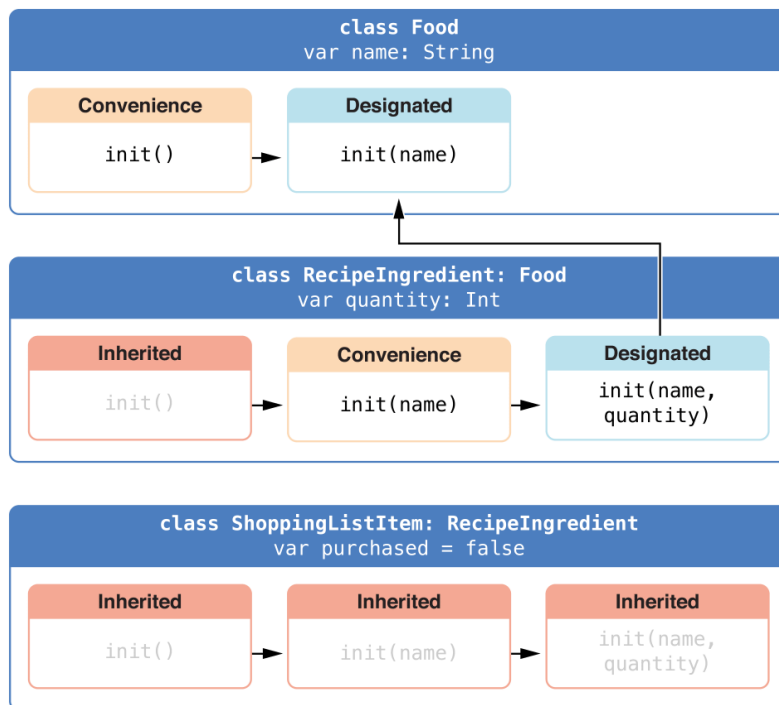
```
1. class ShoppingListItem: RecipeIngredient {
2.     var purchased = false
3.     var description: String {
4.         var output = "\(quantity) x \(name.lowercaseString)"
5.         output += purchased ? " ?" : " ?"
6.         return output
    }
```

```
7.     }
8. }
```

注意：ShoppingListItem 没有定义构造器来为 purchased 提供初始化值，这是因为任何添加到购物单的项的初始状态总是未购买。

由于它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造器，ShoppingListItem 将自动继承所有父类中的指定构造器和便利构造器。

下图种展示了所有三个类的构造器链：



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

```
1. var breakfastList = [
2.     ShoppingListItem(),
3.     ShoppingListItem(name: "Bacon"),
4.     ShoppingListItem(name: "Eggs", quantity: 6),
5. ]
6. breakfastList[0].name = "Orange juice"
7. breakfastList[0].purchased = true
8. for item in breakfastList {
9.     println(item.description)
```

```
10. }  
11. // 1 x orange juice ?  
12. // 1 x bacon ?  
13. // 6 x eggs ?
```

如上所述，例子中通过字面量方式创建了一个新数组 `breakfastList`，它包含了三个新的 `ShoppingListItem` 实例，因此数组的类型也能自动推导为 `ShoppingListItem[]`。在数组创建完之后，数组中第一个 `ShoppingListItem` 实例的名字从 `[Unnamed]` 修改为 `Orange juice`，并标记为已购买。接下来通过遍历数组每个元素并打印它们的描述值，展示了所有项当前的默认状态都已按照预期完成了赋值。

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。

这种类型的闭包或函数一般会创建一个跟属性类型相同的临时变量，然后修改它的值以满足预期的初始状态，最后将这个临时变量的值作为属性的默认值进行返回。

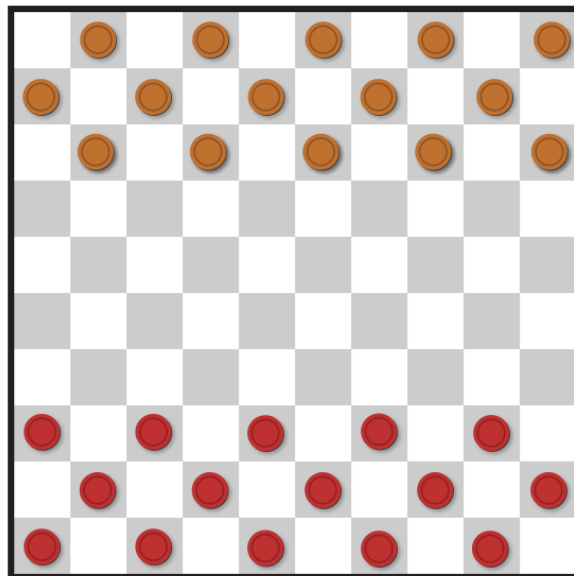
下面列举了闭包如何提供默认值的代码概要：

```
1. class SomeClass {  
2.     let someProperty: SomeType = {  
3.         // 在这个闭包中给 someProperty 创建一个默认值  
4.         // someValue 必须和 SomeType 类型相同  
5.         return someValue  
6.     }()  
7. }
```

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。

注意：如果你使用闭包来初始化属性的值，请记住在闭包执行时，实例的其它部分都还没有初始化。这意味着你不能够在闭包里访问其它的属性，就算这个属性有默认值也不允许。同样，你也不能使用隐式的 **self** 属性，或者调用其它的实例方法。

下面例子中定义了一个结构体 **Checkerboard**，它构建了西洋跳棋游戏的棋盘：



西洋跳棋游戏在一副黑白格交替的 **10x10** 的棋盘中进行。为了呈现这副游戏棋盘，**Checkerboard** 结构体定义了一个属性 **boardColors**，它是一个包含 **100** 个布尔值的数组。数组中的某元素布尔值为 **true** 表示对应的是一个黑格，布尔值为 **false** 表示对应的是一个白格。数组中第一个元素代表棋盘上左上角的格子，最后一个元素代表棋盘上右下角的格子。

boardColor 数组是通过一个闭包来初始化和组装颜色值的：

```
1. struct Checkerboard {
2.     let boardColors: Bool[] = {
3.         var temporaryBoard = Bool[]()
4.         var isBlack = false
5.         for i in 1...10 {
6.             for j in 1...10 {
7.                 temporaryBoard.append(isBlack)
8.                 isBlack = !isBlack
9.             }
10.            isBlack = !isBlack
11.        }
12.    }
13. }
```

```
11.     }
12.     return temporaryBoard
13.     } ()
14.     func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
15.         return boardColors[(row * 10) + column]
16.     }
17. }
```

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**。上面例子中描述的闭包将计算出棋盘中每个格子合适的颜色，将这些颜色值保存到一个临时数组 **temporaryBoard** 中，并在构建完成时将此数组作为闭包返回值返回。这个返回的值将保存到 **boardColors** 中，并可以通过 **squareIsBlackAtRow** 这个工具函数来查询。

```
1. let board = Checkerboard()
2. println(board.squareIsBlackAtRow(0, column: 1))
3. // 输出 "true"
4. println(board.squareIsBlackAtRow(9, column: 9))
5. // 输出 "false"
```

2.15. 反初始化

本篇译者：[bruce0505](#)（Github 主页），校对：[ChildhoodAndy](#)（[博客](#)），敬请勘误。

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地去清理。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：

```
1. deinit {  
2.     // 执行反初始化  
3. }
```

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

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

反初始化函数操作

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

```
1. struct Bank {  
2.     static var coinsInBank = 10_000  
3.     static func vendCoins(var numberOfCoinsToVend: Int) -> Int {  
4.         numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)  
5.         coinsInBank -= numberOfCoinsToVend  
6.         return numberOfCoinsToVend  
7.     }  
8.     static func receiveCoins(coins: Int) {  
9.         coinsInBank += coins  
10.    }  
11. }
```

Bank 根据它的 `coinsInBank` 属性来跟踪当前它拥有的硬币数量。银行还提供两个方法——

`vendCoins` 和 `receiveCoins`，用来处理硬币的分发和收集。

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

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

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

```
1. class Player {
2.     var coinsInPurse: Int
3.     init(coins: Int) {
4.         coinsInPurse = Bank.vendCoins(coins)
5.     }
6.     func winCoins(coins: Int) {
7.         coinsInPurse += Bank.vendCoins(coins)
8.     }
9.     deinit {
10.        Bank.receiveCoins(coinsInPurse)
11.    }
12. }
```

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

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

```
1. var playerOne: Player? = Player(coins: 100)
2. println("A new player has joined the game with \ (playerOne!.coinsInPurse) coins")
3. // 输出 "A new player has joined the game with 100      coins"
```

```
4. println("There are now \(Bank.coinsInBank) coins left    in the bank")
5. // 输出 "There are now 9900 coins left in the bank"
```

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

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

```
1. playerOne!.winCoins(2 000)
2. println("PlayerOne won 2000 coins & now has \    (playerOne!.coinsInPurse) coins")
3. // 输出 "PlayerOne won 2000 coins & now has 2100 coins"
4. println("The bank now only has \(Bank.coinsInBank) coins left")
5. // 输出 "The bank now only has 7900 coins left"
```

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

```
1. playerOne = nil
2. println("PlayerOne has left the game")
3. // 输出 "PlayerOne has left the game"
4. println("The bank now has \(Bank.coinsInBank) coins")
5. // 输出 "The bank now has 10000 coins"
```

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

2.16. 自动引用计数

本篇译者: aquaporcus ([Github 主页](#))，校对: ChildhoodAndy ([博客](#))，敬请勘误。

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

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

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

ARC 如何工作

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

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

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

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

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

ARC 实践

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

```
1. class Person {
2.     let name: String
3.     init(name: String) {
4.         self.name = name
5.         println("\(name) is being initialized")
6.     }
7.
8.     deinit {
9.         println("\(name) is being deinitialized")
10.    }
11. }
```

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

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

```
1. var reference1: Person?
2. var reference2: Person?
3. var reference3: Person?
```

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

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

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

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

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

```
1. reference2 = reference1
2. reference3 = reference2
```

现在，这一个 **Person** 实例有三个强引用。

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

```
1. reference1 = nil
2. 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：

```
1. referenece3 = nil
2. // 打印 "John Appleseed is being deinitialized"
```

类实例间的强引用环

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

然而，我们有可能会写出这样的代码，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。在开始学习如何解决这个问题之前，理解它产生的原因会很有帮助。

下面的例子展示了一个强引用环是如何在不经意之间产生的。例子定义了两个类，分别叫

Person 和 **Apartment**，这两个类建模了一座公寓以及它的居民：

```
1. class Person {
2.     let name: String
3.     init(name: String) { self.name = name }
4.     var apartment: Apartment?
5.     deinit { println("\(name) is being deinitialized") }
6. }
7.
8. class Apartment {
9.     let number: Int
10.    init(number: Int) { self.number = number }
11.    var tenant: Person?
12.    deinit { println("Apartment #\(number) is being deinitialized") }
13. }
```

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

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

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

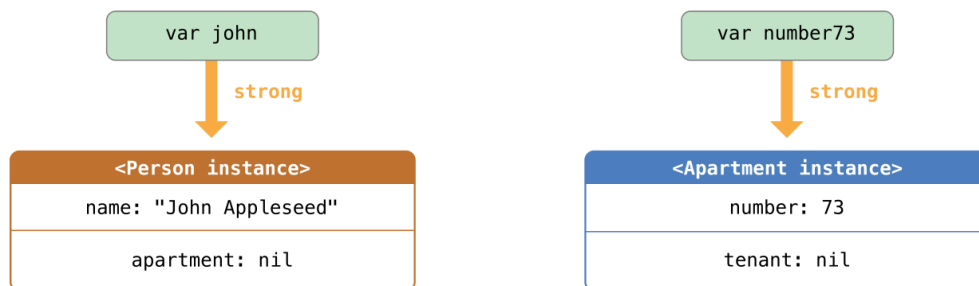
下面的代码片段定义了两个可选类型变量，**john** 和 **number73**，分别被赋值为特定的 **Apartment** 和 **Person** 的实例。得益于可选类型的优点，这两个变量初始值均为 **nil**：

```
1. var john: Person?
2. var number73: Apartment?
```

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

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

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

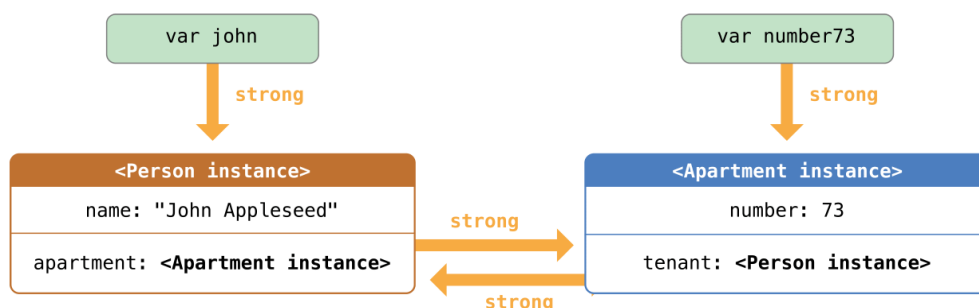


现在你可以将两个实例关联起来，一个人拥有一所公寓，一个公寓也拥有一个房客。注意：用感叹号（!）来展开并访问可选类型的变量，只有这样这些变量才能被赋值：

```

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

两个实例关联起来后，强引用关系如下图所示：



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

```

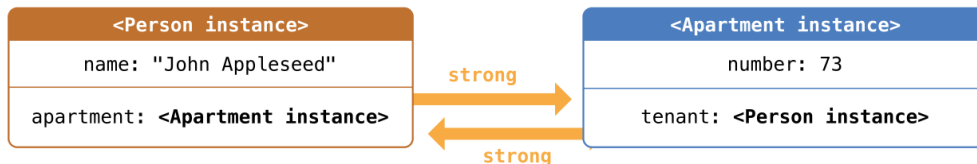
1. john = nil
2. nuber73 = nil
  
```

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

此时强引用关系如下图所示：

var john

var number73



Person 和 Apartment 实例之间的强引用依然存在。

解决实例间的强引用环

Swift 提供两种方法来解决强引用环：弱引用和无主引用。

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

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

弱引用

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

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

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

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

如前所述，弱引用不会保持实例，因此即使实例的弱引用依然存在，ARC 也有可能会销毁

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

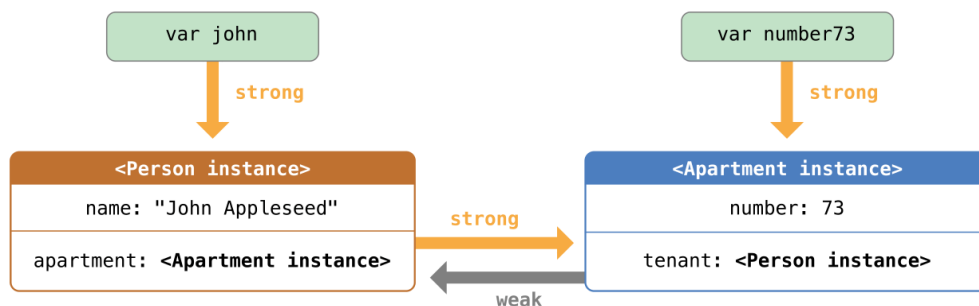
下面的例子和之前的 `Person` 和 `Apartment` 例子相似，除了一个重要的区别。这一次，我们声明 `Apartment` 的 `tenant` 属性为弱引用：

```
1. class Person {
2.     let name: String
3.     init(name: String) { self.name = name }
4.     var apartment: Apartment?
5.     deinit { println("\(name) is being deinitialized") }
6. }
7.
8. class Apartment {
9.     let number: Int
10.    init(number: Int) { self.number = number }
11.    weak var tenant: Person?
12.    deinit { println("Apartment #\(number) is being deinitialized") }
13. }
```

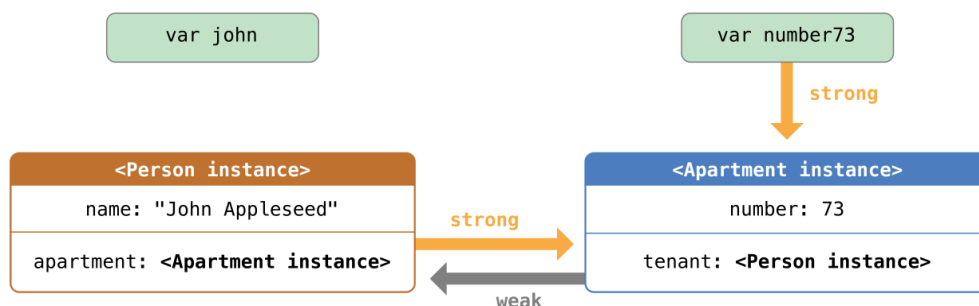
然后创建两个变量(`john` 和 `number73`)的强引用，并关联这两个实例：

```
1. var john: Person?
2. var number73: Apartment?
3.
4. john = Person(name: "John Appleseed")
5. number73 = Apartment(number: 73)
6.
7. john!.apartment = number73
8. number73!.tenant = john
```

下面是引用的关系图：



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

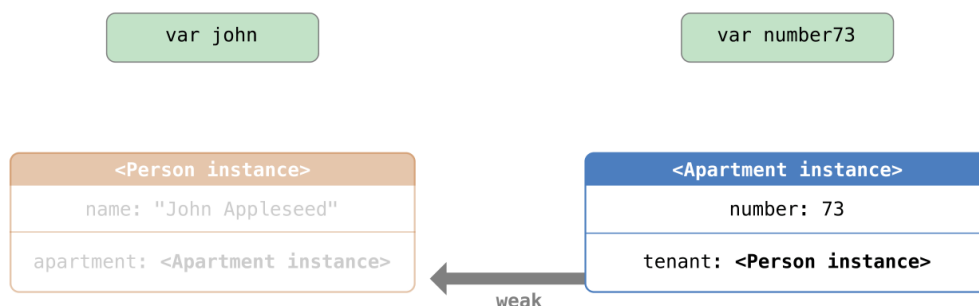


既然不存在 Person 实例的强引用，那么该实例就会被销毁：

```

1. john = nil
2. // 打印"John Appleseed is being deinitialized"
  
```

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



这时，Apartment 实例也被销毁：

```

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

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

无主引用

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

因为是非可选类型,因此当使用无主引用的时候,不需要展开,可以直接访问。不过非可选类型变量不能赋值为 `nil`,因此当实例被销毁的时候,ARC 无法将引用赋值为 `nil`。

注意:

当实例被销毁后,试图访问该实例的无主引用会触发运行时错误。使用无主引用时请确保引用始终指向一个未销毁的实例。上面的非法操作会百分百让应用崩溃,不会发生无法预期的行为。因此,你应该避免这种情况。

接下来的例子定义了两个类, `Customer` 和 `CreditCard`,模拟了银行客户和客户的信用卡。每个类都有一个属性,存储另外一个类的实例。这样的关系可能会产生强引用环。

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

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

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

```
1. class Customer {
2.     let name: String
3.     var card: CreditCard?
```

```

4.     init(name: String) {
5.         self.name = name
6.     }
7.
8.     deinit { println("\(name) is being deinitialized")
9. }
10.
11. class CreditCard {
12.     let number: Int
13.     unowned let customer: Customer
14.     init(number: Int, customer: Customer) {
15.         self.number = number
16.         self.customer = customer
17.     }
18.
19.     deinit { println("Card #\(number) is being deinitialized")
20. }

```

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

```

1. var john: Customer?

```

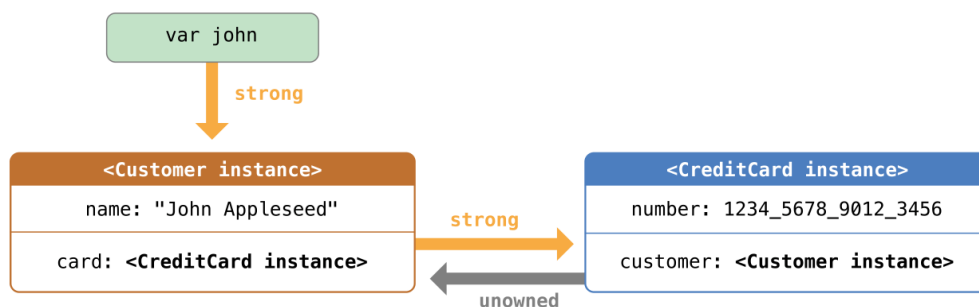
现在创建一个 `Customer` 实例，然后用它来初始化 `CreditCard` 实例，并把刚创建出来的 `CreditCard` 实例赋值给 `Customer` 的 `card` 属性：

```

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

```

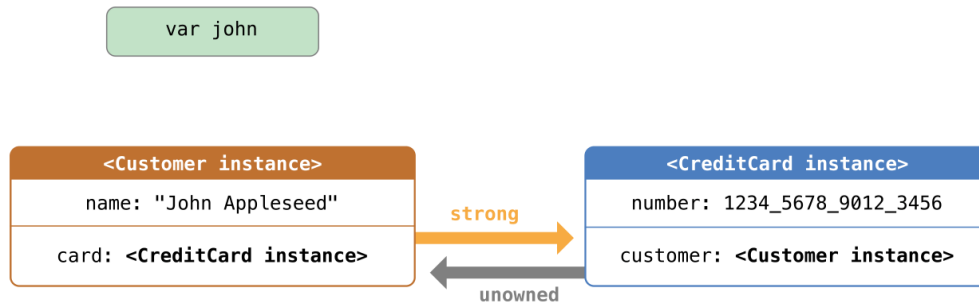
我们来看看此时的引用关系：



`Customer` 实例持有 `CreditCard` 实例的强引用，而 `CreditCard` 实例则持有 `Customer` 实例

的无主引用。

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



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

```
1. john = nil
2. // 打印"John Appleseed is being deinitialized"
3. // 打印"Card #1234567890123456 is being deinitialized"
```

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

无主引用以及隐式展开的可选属性

上述的弱引用和无主引用的例子覆盖了两种常用的需要打破强引用环的应用场景。

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

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

但是，存在第三种场景：两个属性都必须有值，且初始化完成后不能为 `nil`。这种场景下，则需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。

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

下面的例子定义了两个类，**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：

```
1. class Country {
2.     let name: String
3.     let capitalCity: City!
4.     init(name: String, capitalName: String) {
5.         self.name = name
6.         self.capitalCity = City(name: capitalName, country: self)
7.     }
8. }
9.
10. class City {
11.     let name: String
12.     unowned let country: Country
13.     init(name: String, country: Country) {
14.         self.name = name
15.         self.country = country
16.     }
17. }
```

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

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

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

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

参数传递给 `City` 的初始化函数。

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

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

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

闭包产生的强引用环

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

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

因为诸如类这样的闭包是引用类型，导致了强引用环。当你把一个闭包赋值给某个属性时，你也把一个引用赋值给了这个闭包。实质上，这个之前描述的问题是一样的一两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

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

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

```
1. class HTMLElement {
2.
3.     let name: String
```

```
4.     let text: String?
5.
6.     @lazy var asHTML: () -> String = {
7.         if let text = self.text {
8.             return "<\(self.name)>\(text)</\(self.name)>"
9.         } else {
10.            return "<\(self.name) />"
11.        }
12.    }
13.
14.    init(name: String, text: String? = nil) {
15.        self.name = name
16.        self.text = text
17.    }
18.
19.    deinit {
20.        println("\(name) is being deinitialized")
21.    }
22.
23. }
```

类 `HTMLElement` 定义了一个 `name` 属性来表示这个元素的名称，例如代表段落的“`p`”，或者代表换行的“`br`”；以及一个可选属性 `text`，用来设置 `HTML` 元素的文本。

除了上面的两个属性，`HTMLElement` 还定义了一个 `lazy` 属性 `asHTML`。这个属性引用了一个闭包，将 `name` 和 `text` 组合成 `HTML` 字符串片段。该属性是 `() -> String` 类型，就是“没有参数，返回 `String` 的函数”。

默认将闭包赋值给了 `asHTML` 属性，这个闭包返回一个代表 `HTML` 标签的字符串。如果 `text` 值存在，该标签就包含可选值 `text`；或者不包含文本。对于段落，根据 `text` 是“`some text`”还是 `nil`，闭包会返回“`<p>some text</p>`”或者“`<p />`”。

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

注意：`asHTML` 声明为 `lazy` 属性，因为只有当元素确实需要处理为 `HTML` 输出的字符串时，才需要使用 `asHTML`。也就是说，在默认的闭包中可以使用 `self`，因为只有当初始化完成以

及 **self** 确实存在后，才能访问 **lazy** 属性。

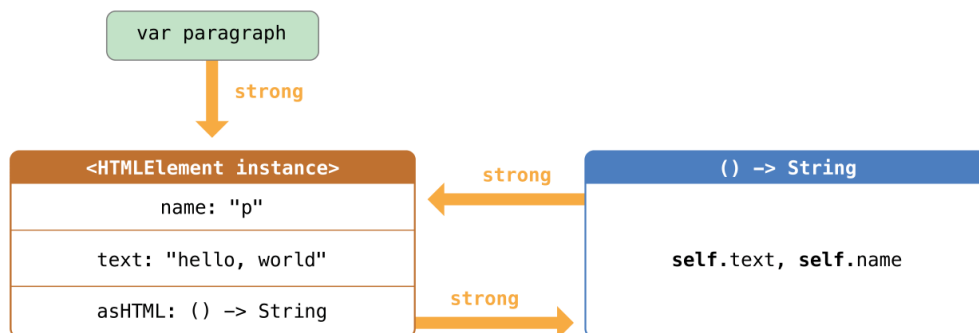
`HTMLElement` 只有一个初始化函数，根据 **name** 和 **text**(如果有的话)参数来初始化一个元素。该类也定义了一个 **deinitializer**，当 `HTMLElement` 实例被销毁时，打印一条消息。

下面的代码创建一个 `HTMLElement` 实例并打印消息。

```
1. var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2. println(paragraph!.asHTML())
3. // 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。如下图所示：



实例的 **asHTML** 属性持有闭包的强引用。

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

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

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

```
1. paragraph = nil
```

注意，`HTMLElementdeinitializer` 中的消息并没有别打印，印证了 `HTMLElement` 实例并没有被销毁。

解决闭包产生的强引用环

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

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

定义占有列表

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

占有列表放置在闭包参数列表和返回类型之前：

```
1. @lazy var someClosure: (Int, String) -> String = {
2.     [unowned self] (index: Int, stringToProcess: String) -> String in
3.     // closure body goes here
4. }
```

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`：

```
1. @lazy var someClosure: () -> String = {
2.     [unowned self] in
3.     // closure body goes here
4.
5. }
```

弱引用和无主引用

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

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

注意：如果占有的引用绝对不会置为 `nil`，应该用无主引用，而不是弱引用。

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

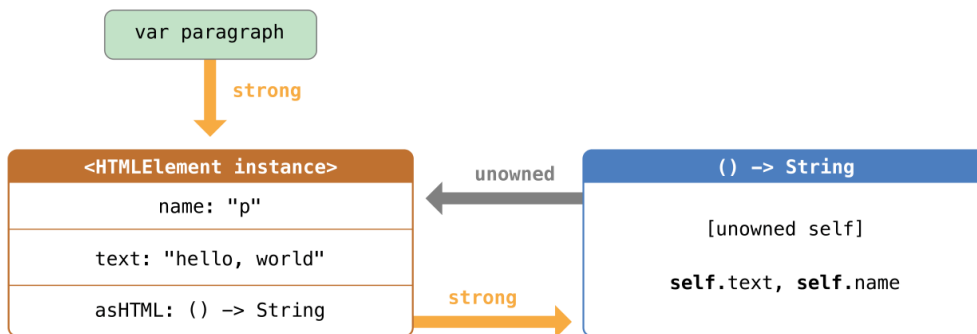
```
1. class HTMLElement {
2.
3.     let name: String
4.     let text: String?
5.
6.     @lazy var asHTML: () -> String = {
7.         [unowned self] in
8.         if let text = self.text {
9.             return "<\(self.name)>\(text)</\(\self.name)>"
10.        } else {
11.            return "<\(self.name) />"
12.        }
13.    }
14.
15.    init(name: String, text: String? = nil) {
16.        self.name = name
17.        self.text = text
18.    }
19.
20.    deinit {
21.        println("\(name) is being deinitialized")
22.    }
23.
24. }
```

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

和之前一样，我们可以创建并打印 `HTMLElement` 实例：

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

使用占有列表后引用关系如下图所示：



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

```
1. paragraph = nil
2. // 打印"p is being deinitialized"
```

2.17. 自判断链接

本篇译者：Jasonbroker ([Github 主页](#))，校对：ChildhoodAndy ([博客](#))，敬请勘误。

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。

注意： **Swift** 的自判断链和 **Objective-C** 中的消息为空有些相像，但是 **Swift** 可以使用在任意类型中，并且失败与否可以被检测到。

自判断链接可替代强制拆包

通过在想调用的属性、方法、或子脚本的自判断值（optional value）（非空）后面放一个问号，可以定义一个自判断链接。这一点很像在自判断值后面放一个声明符号来强制拆得其封包内的值。他们的主要的区别在于当自判断值为空时自判断链接即刻失败，然而一般的强制拆包将会引发运行时错误。

为了反映自判断链接可以调用空（nil），不论你调用的属性、方法、子脚本等返回的值是不是自判断值，它的返回结果都是一个自判断值。你可以利用这个返回值来检测你的自判断链接是否调用成功，有返回值即成功，返回 nil 则失败。

调用自判断链接的返回结果与原本的返回结果具有相同的类型，但是原本的返回结果被包装成了一个自判断值，当自判断链接调用成功时，一个应该返回 `Int` 的属性将会返回 `Int?`。

下面几段代码将解释自判断链接和强制拆包的不同。

首先定义两个类 **Person** 和 **Residence**。

```
1. class Person {
2.     var residence: Residence?
3. }
4.
5. class Residence {
6.     var numberOfRooms = 1
7. }
```

Residence 具有一个 `Int` 类型的 `numberOfRooms`，其值为 1。**Person** 具有一个自判断 `residence` 属性，它的类型是 `Residence?`。

如果你创建一个新的 **Person** 实例，它的 `residence` 属性由于是被定义为自判断型的，此属性将默认初始化为空：

```
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。

```
1. let roomCount = john.residence!.numberOfRooms
2. // this triggers a runtime error
3. //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。然而，如上所述，当 `residence` 为空时，这个代码将会导致运行时错误。

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：

```
1. if let roomCount = john.residence?.numberOfRooms {
2.     println("John's residence has \(roomCount) room(s).")
3. } else {
4.     println("Unable to retrieve the number of rooms.")
5. }
6. // 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。

因为这种尝试获得 `numberOfRooms` 的操作有可能失败，自判断链接会返回 `Int?` 类型值，或者称作“自判断 `Int`”。当 `residence` 是空的时候（上例），选择 `Int` 将会为空，因此会出现无法访问 `numberOfRooms` 的情况。

要注意的是，即使 `numberOfRooms` 是非自判断 `Int`（`Int?`）时这一点也成立。只要是通过自判断链接的请求就意味着最后 `numberOfRooms` 总是返回一个 `Int?` 而不是 `Int`。

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

```
1. john.residence = Residence()
2. 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 numberOfRooms，它将返回一个包含默认值 1 的 Int? :
```

```
3.
4. if let roomCount = john.residence?.numberOfRooms {
5.     println("John's residence has \(roomCount) room(s).")
6. } else {
7.     println("Unable to retrieve the number of rooms.")
8. }
9. // 打印 "John's residence has 1 room(s)".
```

为自判断链接定义模型类

你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。

后面的代码定义了四个将在后面使用的模型类，其中包括多层自判断链接。这些类是由上面的 **Person** 和 **Residence** 模型通过添加一个 **Room** 和一个 **Address** 类拓展来。

Person 类定义与之前相同。

```
1. class Person {
2.     var residence: Residence?
3. }
```

Residence 类比之前复杂些。这次，它定义了一个变量 **rooms**，它被初始化为一个 **Room[]** 类型的空数组：

```
1. class Residence {
2.     var rooms = Room[]()
3.     var numberOfRooms: Int {
4.         return rooms.count
5.     }
6.     subscript(i: Int) -> Room {
7.         return rooms[i]
8.     }
9.     func printNumberOfRooms() {
10.        println("The number of rooms is \(numberOfRooms)")
11.    }
12.    var address: Address?
13. }
```

因为 **Residence** 存储了一个 **Room** 实例的数组，它的 **numberOfRooms** 属性值不是一个固定的存储值，而是通过计算而来的。**numberOfRooms** 属性值是由返回 **rooms** 数组的 **count** 属性值得到的。

为了能快速访问 **rooms** 数组，**Residence** 定义了一个只读的子脚本，通过插入数组的元素角标就可以成功调用。如果该角标存在，子脚本则将该元素返回。

Residence 中也提供了一个 **printNumberOfRooms** 的方法，即简单的打印房间个数。

最后，**Residence** 定义了一个自判断属性叫 **address** (**address?**)。**Address** 类的属性将在后面定义。用于 **rooms** 数组的 **Room** 类是一个很简单的类，它只有一个 **name** 属性和一个设定 **room** 名的初始化器。

```
1. class Room {
2.     let name: String
3.     init(name: String) { self.name = name }
4. }
```

这个模型中的最终类叫做 **Address**。它有三个自判断属性他们额类型是 **String?**。前面两个自判断属性 **buildingName** 和 **buildingNumber** 作为地址的一部分，是定义某个建筑物的两种方式。第三个属性 **street**，用于命名地址的街道名：

```
1. class Address {
2.     var buildingName: String?
3.     var buildingNumber: String?
4.     var street: String?
5.     func buildingIdentifier() -> String? {
6.         if buildingName {
7.             return buildingName
8.         } else if buildingNumber {
9.             return buildingNumber
10.        } else {
11.            return nil
12.        }
13.    }
14. }
```

`Address` 类还提供了一个 `buildingIdentifier` 的方法，它的返回值类型为 `String?`。这个方法检查 `buildingName` 和 `buildingNumber` 的属性，如果 `buildingName` 有值则将其返回，或者如果 `buildingNumber` 有值则将其返回，再或如果没有一个属性有值，返回空。

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。

使用上述定义的类来创建一个人实例，并再次尝试后去它的 `numberOfRooms` 属性：

```
1. let john = Person()
2. if let roomCount = john.residence?.numberOfRooms {
3.     println("John's residence has \(roomCount) room(s).")
4. } else {
5.     println("Unable to retrieve the number of rooms.")
6. }
7. // 打印 "Unable to retrieve the number of rooms."
```

由于 `john.residence` 是空，所以这个自判断链接和之前一样失败了，但是没有运行时错误。

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。

`Residence` 的 `printNumberOfRooms` 方法会打印 `numberOfRooms` 的当前值。方法如下：

```
1. func printNumberOfRooms() {
2.     println("The number of rooms is \(numberOfRooms)")
3. }
```

这个方法没有返回值。但是，没有返回值类型的函数和方法有一个隐式的返回值类型 `Void`（参见 [Function Without Return Values](#)）。

如果你利用自判断链接调用此方法，这个方法的返回值类型将是 `Void?`，而不是 `Void`，因

为当通过自判断链接调用方法时返回值总是自判断类型（`optional type`）。即使是这个方法本是没有定义返回值，你也可以使用 `if` 语句来检查是否能成功调用 `printNumberOfRooms` 方法：如果方法通过自判断链接调用成功，`printNumberOfRooms` 的隐式返回值将会是 `Void`，如果没有成功，将返回 `nil`：

```
1. if john.residence?.printNumberOfRooms() {
2.     println("It was possible to print the number of rooms.")
3. } else {
4.     println("It was not possible to print the number of rooms.")
5. }
6. // 打印 "It was not possible to print the number of rooms."。
```

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。

下面这个例子用在 `Residence` 类中定义的子脚本来获取 `john.residence` 数组中第一个房间的名字。因为 `john.residence` 现在是 `nil`，子脚本的调用失败了。

```
1. if let firstRoomName = john.residence?[0].name {
2.     println("The first room name is \(firstRoomName).")
3. } else {
4.     println("Unable to retrieve the first room name.")
5. }
6. // 打印 "Unable to retrieve the first room name."。
```

在子代码调用中自判断链接的问号直接跟在 `john.residence` 的后面，在子脚本括号的前面，因为 `john.residence` 是自判断链接试图获得的自判断值。

如果你创建一个 `Residence` 实例给 `john.residence`，且在他的 `rooms` 数组中有一个或多个 `Room` 实例，那么你可以使用自判断链接通过 `Residence` 子脚本来获取在 `rooms` 数组中的实例了：

```
1. let johnsHouse = Residence()
2. johnsHouse.rooms += Room(name: "Living Room")
3. johnsHouse.rooms += Room(name: "Kitchen")
4. john.residence = johnsHouse
5.
6. if let firstRoomName = john.residence?[0].name {
7.     println("The first room name is \(firstRoomName).")
8. } else {
9.     println("Unable to retrieve the first room name.")
10. }
11. // 打印 "The first room name is Living Room."
```

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?`。

下面的例子试图获取 `john` 的 `residence` 属性里的 `address` 的 `street` 属性。这里使用了两层自判断链接来联系 `residence` 和 `address` 属性，他们两者都是自判断类型：

```
1. if let johnsStreet = john.residence?.address?.street {
2.     println("John's street name is \(johnsStreet).")
3. } else {
4.     println("Unable to retrieve the address.")
5. }
6. // 打印 "Unable to retrieve the address."
```

`john.residence` 的值现在包含一个 `Residence` 实例，然而 `john.residence.address` 现在是 `nil`，因此 `john.residence?.address?.street` 调用失败。

从上面的例子发现，你试图获得 `street` 属性值。这个属性的类型是 `String?`。因此尽管在自判断类型属性前使用了两层自判断链接，`john.residence?.address?.street` 的返回值类型也

是 `String?` 。

如果你为 `Address` 设定一个实例来作为 `john.residence.address` 的值，并为 `address` 的 `street` 属性设定一个实际值，你可以通过多层自判断链接来得到这个属性值。

```
1. let johnsAddress = Address()
2. johnsAddress.buildingName = "The Larches"
3. johnsAddress.street = "Laurel Street"
4. john.residence!.address = johnsAddress
5.
6. if let johnsStreet = john.residence?.address?.street {
7.     println("John's street name is \(johnsStreet).")
8. } else {
9.     println("Unable to retrieve the address.")
10. }
11. // 打印 "John's street name is Laurel Street."。
```

值得注意的是，“!”符的在定义 `address` 实例时的使用（`john.residence.address`）。`john.residence` 属性是一个自判断类型，因此你需要在它获取 `address` 属性之前使用 `!` 拆包以获得它的实际值。

链接自判断返回值的方法

前面的例子解释了如何通过自判断链接来获得自判断类型属性值。你也可以通过调用返回自判断类型值的方法并按需链接方法的返回值。

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?`：

```
1. if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
2.     println("John's building identifier is \(buildingIdentifier).")
3. }
4. // 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：

```
1. if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
2.     println("John's uppercase building identifier is \(upper).")
3. }
4. // 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。

2.18. 类型转换

本篇译者：xiehurricane ([Github 主页](#))，校对：ChildhoodAndy ([博客](#))，敬请勘误。

类型转换是一种检查类实例的方式，并且哦或者也是让实例作为它的父类或者子类的一种方式。

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。

你也可以用来检查一个类是否实现了某个协议，详细内容请查阅《[Protocols](#)》

定义一个类层次作为例子

你可以将它用在类和子类的层次结构上，检查特定类实例的类型并且转换这个类实例的类型成为这个层次结构中的其他类型。这下面的三个代码段定义了一个类层次和一个 `array` 包含了几个这些类的实例，作为类型转换的例子。

第一个代码片段定义了一个新的基础类 `MediaItem`。这个类为任何出现在数字媒体库的项提供基础功能。特别的，它声明了一个 `String` 类型的 `name` 属性，和一个 `initname` 初始化器。（它假定所有的媒体项都有个名称。）

```
1. class MediaItem {
2.     var name: String
3.     init(name: String) {
4.         self.name = name
5.     }
6. }
```

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：

```
1. class Movie: MediaItem {
2.     var director: String
3.     init(name: String, director: String) {
4.         self.director = director
5.         super.init(name: name)
6.     }
7. }
8.
9. class Song: MediaItem {
10.    var artist: String
11.    init(name: String, artist: String) {
12.        self.artist = artist
13.        super.init(name: name)
14.    }
15. }
```

最后一个代码段创建了一个 **array** 常量 **library**，包含两个 **Movie** 实例和三个 **Song** 实例。**library** 的类型是在它被初始化时根据它的 **array** 标记符和里面的内容（ps: **literal**：标记符其实就是指“[”和“]”，虽然苹果官方的翻译里翻译为字面当总感觉不好理解，有点奇怪。不如翻译为标记符）推断来的。**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。

```
1. let library = [
2.    Movie(name: "Casablanca", director: "Michael Curtiz"),
3.    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
4.    Movie(name: "Citizen Kane", director: "Orson Welles"),
5.    Song(name: "The One And Only", artist: "Chesney Hawkes"),
6.    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
```

```
7. ]  
8. // 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。为了让它们作为它们本来的类型工作，你需要检查它们的类型或者向下转换它们的类型到其它类型，就像下面描述的一样。

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。

```
1. var movieCount = 0  
2. var songCount = 0  
3.  
4. for item in library {  
5.     if item is Movie {  
6.         ++movieCount  
7.     } else if item is Song {  
8.         ++songCount  
9.     }  
10. }  
11.  
12. println("Media library contains \(movieCount) movies and \(songCount)  
    songs")  
13. // prints "Media library contains 2 movies and 3 songs"
```

示例迭代了数组 `library` 中的所有项。每一次，`for-in` 循环设置 `item` 常量的值为数组中的下一个 `MediaItem`。

若当前 `MediaItem` 是一个 `Movie` 类型的实例，`item is Movie` 返回 `true`，相反返回 `false`。同样的，`item is Song` 检查 `item` 是否为 `Song` 类型的实例。在循环末尾，`movieCount` 和 `songCount` 的值就是被找到属于各自的类型的实例数量。

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。你可以相信，上面就是这种情况。你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**。

下面的例子，迭代了 **library** 里的每一个 **MediaItem**，并打印出适当的描述。要这样做，**item** 需要真正作为 **Movie** 或 **Song** 的类型来使用。不仅仅是作为 **MediaItem**。为了能够使用 **Movie** 或 **Song** 的 **director** 或 **artist** 属性，这是必要的。

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：

```
1. for item in library {
2.     if let movie = item as? Movie {
3.         println("Movie: '\(movie.name)', dir. \(movie.director)")
4.     } else if let song = item as? Song {
5.         println("Song: '\(song.name)', by \(song.artist)")
6.     }
7. }
8.
9. // Movie: 'Casablanca', dir. Michael Curtiz
10. // Song: 'Blue Suede Shoes', by Elvis Presley
11. // Movie: 'Citizen Kane', dir. Orson Welles
12. // Song: 'The One And Only', by Chesney Hawkes
```

```
13. // Song: 'Never Gonna Give You Up', by Rick Astley
```

示例首先试图将 `item` 下转为 `Movie`。因为 `item` 是一个 `MediaItem` 类型的实例，它可能是一个 `Movie`；同样，它可能是一个 `Song`，或者仅仅是基类 `MediaItem`。因为不确定，`as?` 形式试图下转时返还一个可选值。`item as Movie` 的返回值是 `Movie?` 类型或“optional `Movie`”。

当应用在两个 `Song` 实例时，下转为 `Movie` 失败。为了处理这种情况，上面的实例使用了可选绑定（optional binding）来检查 optional `Movie` 真的包含一个值（这个是为了判断下转是否成功。）可选绑定是这样写的“if let `movie` = `item as? Movie`”，可以这样解读：

“尝试将 `item` 转为 `Movie` 类型。若成功，设置一个新的临时常量 `movie` 来存储返回的 optional `Movie`”

若下转成功，然后 `movie` 的属性将用于打印一个 `Movie` 实例的描述，包括它的导演的名字 `director`。当 `Song` 被找到时，一个相近的原理被用来检测 `Song` 实例和打印它的描述。

注意：转换没有真的改变实例或它的值。潜在的根本上实例保持不变；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。
2. `Any` 可以表示任何类型，除了方法类型（function types）。

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：

```
1. let someObjects: AnyObject[] = [  
2.     Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),  
  
3.     Movie(name: "Moon", director: "Duncan Jones"),  
4.     Movie(name: "Alien", director: "Ridley Scott")  
5. ]
```

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）：

```
1. for object in someObjects {  
2.     let movie = object as Movie  
3.     println("Movie: '\(movie.name)', dir. \(movie.director)")  
4. }  
5. // Movie: '2001: A Space Odyssey', dir. Stanley Kubrick  
6. // Movie: 'Moon', dir. Duncan Jones  
7. // 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：

```
1. for movie in someObjects as Movie[] {  
2.     println("Movie: '\(movie.name)', dir. \(movie.director)")  
3. }  
4. // Movie: '2001: A Space Odyssey', dir. Stanley Kubrick  
5. // Movie: 'Moon', dir. Duncan Jones  
6. // 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**：

```
1. var things = Any[]()  
2. things.append(0)  
3. things.append(0.0)
```

```
4. things.append(42)
5. things.append(3.14159)
6. things.append("hello")
7. things.append((3.0, 5.0))
8. things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
```

`things` 数组包含两个 `Int` 值, 2 个 `Double` 值, 1 个 `String` 值, 一个元组 (`Double, Double`) , `Ivan Reitman` 导演的电影“`Ghostbusters`”。

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:

```
1. for thing in things {
2.     switch thing {
3.         case 0 as Int:
4.             println("zero as an Int")
5.         case 0 as Double:
6.             println("zero as a Double")
7.         case let someInt as Int:
8.             println("an integer value of \(someInt)")
9.         case let someDouble as Double where someDouble > 0:
10.            println("a positive double value of \(someDouble)")
11.        case is Double:
12.            println("some other double value that I don't want to print")
13.        case let someString as String:
14.            println("a string value of \"\(someString)\"")
15.        case let (x, y) as (Double, Double):
16.            println("an (x, y) point at \(x), \(y)")
17.        case let movie as Movie:
18.            println("a movie called \"\(movie.name)\", dir. \(movie.director)")
19.        default:
20.            println("something else")
21.    }
22. }
23. // zero as an Int
24. // zero as a Double
```

```
25. // an integer value of 42
26. // a positive double value of 3.14159
27. // a string value of "hello"
28. // an (x, y) point at 3.0, 5.0
29. // 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。

2.19 类型嵌套

本篇译者：LIN-H ([Github 主页](#))，敬请勘误。

枚举类型常被用于实现特定类或结构体的功能。同样地，也能够有多种变量类型的环境中方便地定义通用类或结构体。为了实现这种功能，**Swift** 允许你定义类型嵌套，可以在枚举类型、类和结构体中定义支持嵌套的类型。

要在一个类型中嵌套另一个类型，将需要嵌套的类型的定义写在被嵌套类型的区域`{}`内，而且可以根据需要定义多级嵌套。

类型嵌套实例

下面这个例子定义了一个结构体 **BlackjackCard**，用来模拟 **BlackjackCard**(游戏：二十一点)中的扑克牌点数。**BlackjackCard** 结构体包含 2 个嵌套定义的枚举类型 **Suit** 和 **Rank**。

在 **BlackjackCard** 规则中，**Ace** 牌可以表示 1 或者 11，**Ace** 牌的这一特征用一个嵌套在枚举类型 **Rank** 中的结构体 **Values** 来表示。

```
1. struct BlackjackCard {
2.
3.     // 嵌套定义枚举型 Suit
4.     enum Suit: Character {
5.         case Spades = "♠", Hearts = "♥", Diamonds = "♦", Clubs = "♣"
```

```
6.     }
7.
8.     // 嵌套定义枚举型 Rank
9.     enum Rank: Int {
10.         case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
11.         case Jack, Queen, King, Ace
12.         struct Values {
13.
14.             let first: Int, second: Int?
15.         }
16.         var values: Values {
17.             switch self {
18.             case .Ace:
19.                 return Values(first: 1, second: 11)
20.             case .Jack, .Queen, .King:
21.                 return Values(first: 10, second: nil)
22.             default:
23.                 return Values(first: self.toRaw(), second: nil)
24.             }
25.         }
26.     }
27.
28.     // BlackjackCard 的属性和方法
29.     let rank: Rank, suit: Suit
30.     var description: String {
31.         var output = "suit is \(suit.toRaw()),"
32.         output += " value is \(rank.values.first)"
33.         if let second = rank.values.second {
34.             output += " or \(second) "
35.         }
36.         return output
37.     }
38. }
```

枚举型的 **Suit** 用来描述扑克牌的四种花色，并分别用一个 **Character** 类型的值代表花色符号。

枚举型的 **Rank** 用来描述扑克牌从 **Ace~10,J,Q,K,13** 张牌，并分别用一个 **Int** 类型的值表示牌的面值(这个 **Int** 类型的值不适用于 **Ace,J,Q,K** 的牌)。

如上文所提到的，枚举型 **Rank** 在自己内部定义了一个嵌套结构体 **Values**。这个结构体包含两个变量，只有 **Ace** 有两个数值，其余牌都只有一个数值。结构体 **Values** 中定义了两个属性：

first, 为 **Int** ；

second, 为 **Int?**, 或 “optional **Int**”；

Rank 还定义了一个计算属性 **values**，这个计算属性会根据牌的面值，用适当的数值去初始化 **Values** 实例，并赋值给 **values**。对于 **J,Q,K,Ace** 会使用特殊数值，对于数字面值的牌使用 **Int** 类型的值。

BlackjackCard 结构体自身有两个属性——**rank** 与 **suit**，它还定义了一个计算属性 **description**，**description** 属性使用 **rank** 和 **suit** 中的内容来构建对这张扑克牌名字和数值的描述，并且使用可选类型来检查是否存在第二个值，若存在，则在原有的描述中增加对第二数值的描述。

因为 **BlackjackCard** 是一个没有自定义构造函数的结构体，正如《[Memberwise Initializers for Structure Types](#)》中所描述的，**BlackjackCard** 结构体有默认的成员构造函数，所以你可以使用默认的 **initializer** 去初始化新的常量 **theAceOfSpades**：

```
1. let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
2. println("theAceOfSpades: \(theAceOfSpades.description)")
3. // 打印出 "theAceOfSpades: suit is ♠, value is 1 or 11"
```

尽管 **Rank** 和 **Suit** 嵌套在 **BlackjackCard** 中，但仍可被引用，所以在初始化实例时能够通过枚举类型中的成员名称（**.Ace** 和 **.Spades**）单独引用。在上面的例子中，**description** 属性能正确地输出 **theAceOfSpades** 有 1 和 11 两个值。

类型嵌套的引用

在外部对嵌套类型的引用，是以被嵌套类型的名字为前缀，加上所要引用的属性名：

```
1. let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
2.
3. // 红心的符号 为 "?"
```

对于上面这个例子，这样做可以使 **Suit**, **Rank**, 和 **Values** 的名字尽可能的简短，因为它们的名字会自然地由被定义的上下文来限定。

2.20 扩展

本篇译者：lyuka ([Github 主页](#))，敬请勘误。

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**：

```
1. extension SomeType {  
2.     // 加到 SomeType 的新功能写到这里  
3. }
```

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：

```
1. extension SomeType: SomeProtocol, AnotherProctocol {
2.     // 协议实现写到这里
3. }
```

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。

```
1. extension Double {
2.     var km: Double { return self * 1_000.0 }
3.     var m : Double { return self }
4.     var cm: Double { return self / 100.0 }
5.     var mm: Double { return self / 1_000.0 }
6.     var ft: Double { return self / 3.28084 }
7. }
8. let oneInch = 25.4.mm
9. println("One inch is \(oneInch) meters")
10. // 打印输出: "One inch is 0.0254 meters"
11. let threeFeet = 3.ft
12. println("Three feet is \(threeFeet) meters")
13. // 打印输出: "Three feet is 0.914399970739201 meters"
```

这些计算属性表达的含义是把一个 **Double** 型的值看作是某单位下的长度值。即使它们被实现为计算型属性，但这些属性仍可以接一个带有 **dot** 语法的浮点型字面值，而这恰恰是使用这些浮点型字面量实现距离转换的方式。

在上述例子中，一个 **Double** 型的值 **1.0** 被用来表示“1 米”。这就是为什么 **m** 计算型属性返回 **self**——表达式 **1.m** 被认为是计算 **1.0** 的 **Double** 值。

其它单位则需要一些转换来表示在米下测量的值。1 千米等于 1,000 米，所以 **km** 计算型属性要把值乘以 **1_000.00** 来转化成单位米下的数值。类似地，1 米有 **3.28024** 英尺，所以 **ft** 计算型属性要把对应的 **Double** 值除以 **3.28024** 来实现英尺到米的单位换算。

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：

```
1. let aMarathon = 42.km + 195.m
2. println("A marathon is \(aMarathon) meters long")
3. // 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。

注意：如果你使用扩展向一个值类型添加一个构造器，该构造器向所有的存储属性提供默认值，而且没有定义任何定制构造器 (**custom initializers**)，那么对于来自你的扩展构造器中的值类型，你可以调用默认构造器 (**default initializers**) 和成员级构造器 (**memberwise initializers**)。正如在值类型的构造器授权中描述的，如果你已经把构造器写成值类型原始实现的一部分，上述规则不再适用。

下面的例子定义了一个用于描述几何矩形的定制结构体 `Rect`。这个例子同时定义了两个辅助结构体 `Size` 和 `Point`，它们都把 `0.0` 作为所有属性的默认值：

```
1. struct Size {
2.     var width = 0.0, height = 0.0
3. }
4. struct Point {
5.     var x = 0.0, y = 0.0
6. }
7. struct Rect {
8.     var origin = Point()
9.     var size = Size()
10. }
```

因为结构体 `Rect` 提供了其所有属性的默认值，所以正如默认构造器中描述的，它可以自动接受一个默认的构造器和一个成员级构造器。这些构造器可以用于构造新的 `Rect` 实例：

```
1. let defaultRect = Rect()
2. let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
3.     size: Size(width: 5.0, height: 5.0))
```

你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:

```
1. extension Rect {
2.     init(center: Point, size: Size) {
3.         let originX = center.x - (size.width / 2)
4.         let originY = center.y - (size.height / 2)
5.         self.init(origin: Point(x: originX, y: originY), size: size)
6.     }
7. }
```

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:

```
1. let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
2.     size: Size(width: 3.0, height: 3.0))
3. // 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。

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:

```
1. extension Int {
2.     func repetitions(task: () -> ()) {
3.         for i in 0..self {
4.             task()
5.         }
6.     }
7. }
```

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：

```
1. 3.repetitions({
2.     println("Hello!")
3. })
4. // Hello!
5. // Hello!
6. // Hello!
```

可以使用 `trailing` 闭包使调用更加简洁：

```
1. 3.repetitions{
2.     println("Goodbye!")
3. }
4. // Goodbye!
5. // Goodbye!
6. // 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：

```
1. extension Int {
2.     mutating func square() {
3.         self = self * self
4.     }
5. }
6. var someInt = 3
7. someInt.square()
8. // 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

```
1. 123456789[0] 返回 9
2. 123456789[1] 返回 8
```

...等等

```
1. extension Int {
2.     subscript(digitIndex: Int) -> Int {
3.         var decimalBase = 1
4.         for _ in 1...digitIndex {
5.             decimalBase *= 10
6.         }
7.         return (self / decimalBase) % 10
8.     }
9. }
10. 746381295[0]
11. // returns 5
12. 746381295[1]
13. // returns 9
14. 746381295[2]
15. // returns 2
16. 746381295[8]
17. // 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**：

```
1. 746381295[9]
2. //returns 0,
```

即等同于：

```
1. 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：

```
1. extension Character {
2.     enum Kind {
3.         case Vowel, Consonant, Other
4.     }
5.     var kind: Kind {
6.         switch String(self).lowercaseString {
7.         case "a", "e", "i", "o", "u":
8.             return .Vowel
9.         case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
10.            "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
11.             return .Consonant
12.         default:
13.             return .Other
14.         }
15.     }
16. }
```

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。

这个类子还向 **Character** 添加了一个新的计算实例属性,即 **kind**,用来返回合适的 **Kind** 枚举成员。

现在,这个嵌套枚举可以和一个 **Character** 值联合使用了:

```
1. func printLetterKinds(word: String) {
2.     println("'\\(word)' is made up of the following kinds of letters:"
3. )
4.     for character in word {
5.         switch character.kind {
6.         case .Vowel:
7.             print("vowel ")
8.         case .Consonant:
```

```
8.         print("consonant ")
9.         case .Other:
10.            print("other ")
11.        }
12.    }
13.    print("\n")
14.}
15.printLetterKinds("Hello")
16.// 'Hello' is made up of the following kinds of letters:
17.// consonant vowel consonant consonant vowel
```

函数 `printLetterKinds` 的输入是一个 `String` 值并对其字符进行迭代。在每次迭代过程中，考虑当前字符的 `kind` 计算属性，并打印出合适的类别描述。所以 `printLetterKinds` 就可以用来打印一个完整单词中所有字母的类型，正如上述单词“hello”所展示的。

注意：由于已知 `character.kind` 是 `Character.Kind` 型，所以 `Character.Kind` 中的所有成员值都可以使用 `switch` 语句里的形式简写，比如使用 `.Vowel` 代替 `Character.Kind.Vowel`

2.21 协议

本篇译者：geek5nan ([Github 主页](#))，敬请勘误。

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。

遵循者需要提供协议指定的成员，如属性，方法，操作符，下标等。

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：

```
1. protocol SomeProtocol {
2.     // 协议内容
3. }
```

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:

```
1. struct SomeStructure: FirstProtocol, AnotherProtocol {
2.     // 结构体内容
3. }
```

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:

```
1. class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
2.     // 类的内容
3. }
```

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。

```
1. protocol SomeProtocol {
2.     var musBeSettable : Int { get set }
3.     var doesNotNeedToBeSettable: Int { get }
4. }
```

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:

```
1. protocol AnotherProtocol {
2.     class var someTypeProperty: Int { get set }
3. }
4.
5. protocol FullyNamed {
6.     var fullName: String { get }
7. }
```

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。

```
1. struct Person: FullyNamed{
2.     var fullName: String
3. }
4. let john = Person(fullName: "John Appleseed")
5. //john.fullName 为 "John Appleseed"
```

`Person` 结构体含有一个名为 `fullName` 的存储型属性，完整的遵循了协议。(若协议未被完整遵循，编译时则会报错)。

如下所示，`Starship` 类遵循了 `FullyNamed` 协议：

```
1. class Starship: FullyNamed {
2.     var prefix: String?
3.     var name: String
4.     init(name: String, prefix: String? = nil ) {
5.         self.name = name
6.         self.prefix = prefix
7.     }
8.     var fullName: String {
9.         return (prefix ? prefix ! + " " : " ") + name
10.    }
11. }
12. var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
13. // ncc1701.fullName == "USS Enterprise"
```

`Starship` 类将 `fullName` 实现为可读的计算型属性。它的每一个实例都有一个名为 `name` 的必备属性和一个名为 `prefix` 的可选属性。当 `prefix` 存在时，将 `prefix` 插入到 `name` 之前来为 `Starship` 构建 `fullName`。

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：

```
1. protocol SomeProtocol {
2.     class func someTypeMethod()
3. }
4.
5. protocol RandomNumberGenerator {
6.     func random() -> Double
7. }
```

RandomNumberGenerator 协议要求其遵循者必须拥有一个名为 **random**，返回值类型为 **Double** 的实例方法。(我们假设随机数在[0, 1]区间内)。

LinearCongruentialGenerator 类遵循了 **RandomNumberGenerator** 协议，并提供了一个叫做线性同余生成器(**linear congruential generator**)的伪随机数算法。

```
1. class LinearCongruentialGenerator: RandomNumberGenerator {
2.     var lastRandom = 42.0
3.     let m = 139968.0
4.     let a = 3877.0
5.     let c = 29573.0
6.     func random() -> Double {
7.         lastRandom = ((lastRandom * a + c) % m)
8.         return lastRandom / m
9.     }
10. }
11. let generator = LinearCongruentialGenerator()
12. println("Here's a random number: \(generator.random())")
13. // 输出 : "Here's a random number: 0.37464991998171"
14. println("And another one: \(generator.random())")
15. // 输出 : "And another one: 0.729023776863283"
```

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**)(译者注：特

指结构体和枚举)中的函数前缀加上 `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。

(译者注: 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, `Toggable` 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:

```
1. protocol Toggable {
2.     mutating func toggle()
3. }
```

当使用枚举或结构体来实现 `Toggable` 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 `Toggable` 协议, `On`, `Off` 两个成员用于表示当前状态

```
1. enum OnOffSwitch: Toggable {
2.     case Off, On
3.     mutating func toggle() {
4.         switch self {
5.             case Off:
6.                 self = On
7.             case On:
8.                 self = Off
9.         }
10.    }
11. }
12. var lightSwitch = OnOffSwitch.Off
13. lightSwitch.toggle()
14. //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

```
1. class Dice {
2.     let sides: Int
3.     let generator: RandomNumberGenerator
4.     init(sides: Int, generator: RandomNumberGenerator) {
5.         self.sides = sides
6.         self.generator = generator
7.     }
8.     func roll() -> Int {
9.         return Int(generator.random() * Double(sides)) + 1
10.    }
11. }
```

这里定义了一个名为 `Dice` 的类，用来代表桌游中的 `N` 个面的骰子。

`Dice` 含有 `sides` 和 `generator` 两个属性，前者用来表示骰子有几个面，后者为骰子提供一个随机数生成器。由于后者为 `RandomNumberGenerator` 的协议类型。所以它能够被赋值为任意遵循该协议的类型。

此外，使用构造器(`init`)来代替之前版本中的 `setup` 操作。构造器中含有一个名为 `generator`，类型为 `RandomNumberGenerator` 的形参，使得它可以接收任意遵循 `RandomNumberGenerator` 协议的类型。

`roll` 方法用来模拟骰子的面值。它先使用 `generator` 的 `random` 方法来创建一个[0-1]区间内的随机数种子，然后加工这个随机数种子生成骰子的面值。

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

```
1. var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
2. for _ in 1...5 {
3.     println("Random dice roll is \(d6.roll())")
4. }
5. //输出结果
6. //Random dice roll is 3
7. //Random dice roll is 5
8. //Random dice roll is 4
9. //Random dice roll is 5
10. //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：

```
1. protocol DiceGame {
2.     var dice: Dice { get }
3.     func play()
4. }
5. protocol DiceGameDelegate {
6.     func gameDidStart(game: DiceGame)
7.     func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
8.     func gameDidEnd(game: DiceGame)
9. }
```

DiceGame 协议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。

如下所示, SnakesAndLadders 是 Snakes and Ladders(译者注: 控制流章节有该游戏的详细介绍)游戏的新版本。新版本使用 Dice 作为骰子, 并且实现了 DiceGame 和 DiceGameDelegate 协议

```
1. class SnakesAndLadders: DiceGame {
2.     let finalSquare = 25
3.     let dic = Dice(sides: 6, generator: LinearCongruentialGenerator())
4.
5.     var square = 0
6.     var board: Int[]
7.     init() {
8.         board = Int[] (count: finalSquare + 1, repeatedValue: 0)
9.         board[03] = +08; board[06] = +11; borad[09] = +09; board[10] =
+02
10.        borad[14] = -10; board[19] = -11; borad[22] = -
02; board[24] = -08
11.    }
12.    var delegate: DiceGameDelegate?
13.    func play() {
14.        square = 0
15.        delegate?.gameDidStart(self)
16.        gameLoop: while square != finalSquare {
17.            let diceRoll = dice.roll()
18.            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
19.
20.            switch square + diceRoll {
21.            case finalSquare:
22.                break gameLoop
23.            case let newSquare where newSquare > finalSquare:
24.                continue gameLoop
25.            default:
26.                square += diceRoll
27.                square += board[square]
28.            }
29.        }
30.        delegate?.gameDIdEnd(self)
```

```
29.     }  
30. }
```

游戏的初始化设置(setup)被 `SnakesAndLadders` 类的构造器(initializer)实现。所有的游戏逻辑被转移到了 `play` 方法中。

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

```
1. class DiceGameTracker: DiceGameDelegate {  
2.     var numberOfTurns = 0  
3.     func gameDidStart(game: DiceGame) {  
4.         numberOfTurns = 0  
5.         if game is SnakesAndLadders {  
6.             println("Started a new game of Snakes and Ladders")  
7.         }  
8.         println("The game is using a \(game.dice.sides)-sided dice")  
9.     }  
10.    func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)  
11.        t) {  
12.        ++numberOfTurns  
13.        println("Rolled a \(diceRoll)")  
14.    }  
15.    func gameDidEnd(game: DiceGame) {  
16.        println("The game lasted for \(numberOfTurns) turns")  
17.    }
```

`DiceGameTracker` 实现了 `DiceGameDelegate` 协议的方法要求, 用来记录游戏已经进行的

轮数。当游戏开始时，`numberOfTurns` 属性被赋值为 0；在每新一轮中递增；游戏结束后，输出打印游戏的总轮数。

`gameDidStart` 方法从 `game` 参数获取游戏信息并输出。`game` 在方法中被当做 `DiceGame` 类型而不是 `SnakeAndLadders` 类型，所以方法中只能访问 `DiceGame` 协议中的成员。

`DiceGameTracker` 的运行情况，如下所示：

```
1. "let tracker = DiceGameTracker()
2. let game = SnakesAndLadders()
3. game.delegate = tracker
4. game.play()
5. // Started a new game of Snakes and Ladders
6. // The game is using a 6-sided dice
7. // Rolled a 3
8. // Rolled a 5
9. // Rolled a 4
10. // Rolled a 5
11. // The game lasted for 4 turns"
```

在扩展中添加协议成员

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。详情请见扩展章节中查看。

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：

```
1. protocol TextRepresentable {
2.     func asText() -> String
3. }
```

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

```
1. extension Dice: TextRepresentable {
2.     cun asText() -> String {
3.         return "A \(sides)-sided dice"
4.     }
5. }
```

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

```
1. let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
2. println(d12.asText())
3. // 输出 "A 12-
   sided dice"let d12 = Dice(sides: 12, generator: LinearCongruentialGener
   ator())
4. println(d12.asText())
5. // 输出 "A 12-
   sided dice"let d12 = Dice(sides: 12, generator: LinearCongruentialGener
   ator())
6. println(d12.asText())
7. // 输出 "A 12-sided dice"
```

SnakesAndLadders 类也可以通过扩展的方式来遵循协议：

```
1. extension SnakeAndLadders: TextRepresentable {
2.     func asText() -> String {
3.         return "A game of Snakes and Ladders with \(finalSquare) squar
   es"
4.     }
5. }
6. println(game.asText())
7. // 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：

```
1. struct Hamster {
2.     var name: String
3.     func asText() -> String {
4.         return "A hamster named \(name)"
5.     }
6. }
```

```
5.     }
6. }
7. extension Hamster: TextRepresentable {}
```

从现在起，**Hamster** 的实例可以作为 **TextRepresentable** 类型使用

```
1. let simonTheHamster = Hamster(name: "Simon")
2. let somethingTextRepresentable: TextRepresentable = simonTheHamster
3. println(somethingTextRepresentable.asText())
4. // 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：

```
1. let things: TextRepresentable[] = [game, d12, simonTheHamster]
```

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：

```
1. for thing in things {
2.     println(thing.asText())
3. }
4. // A game of Snakes and Ladders with 25 squares
5. // A 12-sided dice
6. // 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

```
1. protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
2.     // 协议定义
```

```

3. }
4. 如下所示, PrettyTextRepresentable 协议继承了 TextRepresentable 协议
5.
6. protocol PrettyTextRepresentable: TextRepresentable {
7.     func asPrettyText() -> String
8. }

```

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:

```

1. extension SnakesAndLadders: PrettyTextRepresentable {
2.     func asPrettyText() -> String {
3.         var output = asText() + ":\n"
4.         for index in 1...finalSquare {
5.             switch board[index] {
6.                 case let ladder where ladder > 0:
7.                     output += "▲ "
8.                 case let snake where snake < 0:
9.                     output += "▼ "
10.                default:
11.                    output += "○ "
12.            }
13.        }
14.        return output
15.    }
16. }

```

在 `for in` 中迭代出了 `board` 数组中的每一个元素:

当从数组中迭代出的元素的值大于 0 时, 用 ▲ 表示。

当从数组中迭代出的元素的值小于 0 时, 用 ▼ 表示。

当从数组中迭代出的元素的值等于 0 时, 用 ○ 表示。

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

```

1. println(game.asPrettyText())
2. // A game of Snakes and Ladders with 25 squares:
3. // ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○

```

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：

```
1. protocol Named {
2.     var name: String { get }
3. }
4. protocol Aged {
5.     var age: Int { get }
6. }
7. struct Person: Named, Aged {
8.     var name: String
9.     var age: Int
10.}
11. func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
12.     println("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")
13.}
14. let birthdayPerson = Person(name: "Malcolm", age: 21)
15. wishHappyBirthday(birthdayPerson)
16. // 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。

```
1. @objc protocol HasArea {
2.     var area: Double { get }
3. }
```

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。详情查看 [Using Swift with Cocoa and Objective-C](#)。

```
1. class Circle: HasArea {
2.     let pi = 3.1415927
3.     var radius: Double
4.     var area: ~radius }
5.     init(radius: Double) { self.radius = radius }
6. }
7. class Country: HasArea {
8.     var area: Double
9.     init(area: Double) { self.area = area }
10. }
```

`Circle` 和 `Country` 都遵循了 `HasArea` 协议，前者把 `area` 写为计算型属性 (computed property)，后者则把 `area` 写为存储型属性 (stored property)。

如下所示，`Animal` 类没有实现任何协议

```
1. class Animal {
2.     var legs: Int
3.     init(legs: Int) { self.legs = legs }
4. }
```

`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:

```
1. let objects: AnyObject[] = [  
2.     Circle(radius: 2.0),  
3.     Country(area: 243_610),  
4.     Animal(legs: 4)  
5. ]
```

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

```
1. for object in objects {  
2.     if let objectWithArea = object as? HasArea {  
3.         println("Area is \(objectWithArea.area)")  
4.     } else {  
5.         println("Something that doesn't have an area")  
6.     }  
7. }  
8. // Area is 12.5663708  
9. // Area is 243610.0  
10. // Something that doesn't have an area
```

当数组中的元素遵循 `HasArea` 协议时, 通过 `as?` 操作符将其可选绑定(optional binding)到 `objectWithArea` 常量上。

`objects` 数组中元素的类型并不会因为向下转型而改变, 当它们被赋值给 `objectWithArea` 时只被视为 `HasArea` 类型, 因此只有 `area` 属性能够被访问。

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。

可选协议在调用时使用可选链, 详细内容在可选链章节中查看。

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 类使用 CounterDataSource 类型的外部数据源来提供增量值(increment amount)，如下所示：

```
1. @objc protocol CounterDataSource {
2.     @optional func incrementForCount(count: Int) -> Int
3.     @optional var fixedIncrement: Int { get }
4. }
```

CounterDataSource 含有 incrementForCount 的可选方法和 fixedIncrement 的可选属性。

注意：CounterDataSource 中的属性和方法都是可选的，因此可以在类中声明但不实现这些成员，尽管技术上允许这样做，不过最好不要这样写。

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

```
1. @objc class Counter {
2.     var count = 0
3.     var dataSource: CounterDataSource?
4.     func increment() {
5.         if let amount = dataSource?.incrementForCount?(count) {
6.             count += amount
7.         } else if let amount = dataSource?.fixedIncrement? {
8.             count += amount
9.         }
10.    }
11. }
```

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：

```
1. class ThreeSource: CounterDataSource {
2.     let fixedIncrement = 3
3. }
```

使用 `ThreeSource` 作为数据源开实例化一个 `Counter`：

```
1. var counter = Counter()
2. counter.dataSource = ThreeSource()
3. for _ in 1...4 {
4.     counter.increment()
5.     println(counter.count)
6. }
7. // 3
8. // 6
9. // 9
10. // 12
```

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

```
1. class TowardsZeroSource: CounterDataSource {
2.     func incrementForCount(count: Int) -> Int {
3.         if count == 0 {
4.             return 0
5.         } else if count < 0 {
6.             return 1
7.         } else {
8.             return -1
9.         }
10.     }
```

```
11. }
```

下边是执行的代码：

```
1. counter.count = -4
2. counter.dataSource = TowardsZeroSource()
3. for _ in 1...5 {
4.     counter.increment()
5.     println(counter.count)
6. }
7. // -3
8. // -2
9. // -1
10. // 0
11. // 0
```

2.22 泛型

本篇译者：takalard，敬请勘误。

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。

泛型是 **Swift** 强大特征中的其中一个，许多 **Swift** 标准库是通过泛型代码构建出来的。事实上，泛型的使用贯穿了整本语言手册，只是你没有发现而已。例如，**Swift** 的数组和字典类型都是泛型集。你可以创建一个 **Int** 数组，也可创建一个 **String** 数组，或者甚至于可以是任何其他 **Swift** 的类型数据数组。同样的，你也可以创建存储任何指定类型的字典（dictionary），而且这些类型可以是没有限制的。

泛型所解决的问题

这里是一个标准的，非泛型函数 `swapTwoInts`，用来交换两个 **Int** 值：

```
1. func swapTwoInts(inout a: Int, inout b: Int)
2.     let temporaryA = a
3.     a = b
```

```
4.     b = temporaryA
5. }
```

这个函数使用写入读出（in-out）参数来交换 **a** 和 **b** 的值，请参考写入读出参数。

`swapTwoInts` 函数可以交换 **b** 的原始值到 **a**，也可以交换 **a** 的原始值到 **b**，你可以调用这个函数交换两个 `Int` 变量值：

```
1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. println("someInt is now \(someInt), and anotherInt is now \(anotherInt
   )")
5. // 输出 "someInt is now 107, and anotherInt is now 3"
```

`swapTwoInts` 函数是非常有用的，但是它只能交换 `Int` 值，如果你想要交换两个 `String` 或者 `Double`，就不得不写更多的函数，如 `swapTwoStrings` 和 `swapTwoDoubles`，如同如下所示：

```
1. func swapTwoStrings(inout a: String, inout b: String) {
2.     let temporaryA = a
3.     a = b
4.     b = temporaryA
5. }
6.
7. func swapTwoDoubles(inout a: Double, inout b: Double) {
8.     let temporaryA = a
9.     a = b
10.    b = temporaryA
11. }
```

你可能注意到 `swapTwoInts`、`swapTwoStrings` 和 `swapTwoDoubles` 函数功能都是相同的，唯一不同之处就在于传入的变量类型不同，分别是 `Int`、`String` 和 `Double`。

但实际应用中通常需要一个用处更强大并且尽可能的考虑到更多的灵活性单个函数，可以用来交换两个任何类型值，很幸运的是，泛型代码帮你解决了这种问题。（一个这种泛型函数后面已经定义好了。）

注意： 在所有三个函数中，**a** 和 **b** 的类型是一样的。如果 **a** 和 **b** 不是相同的类型，那它们俩就不能互换值。**Swift** 是类型安全的语言，所以它不允许一个 **String** 类型的变量和一个 **Double** 类型的变量互相交换值。如果一定要做，**Swift** 将报编译错误。

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：

```
1. func swapTwoValues<T>(inout a: T, inout b: T) {
2.     let temporaryA = a
3.     a = b
4.     b = temporaryA
5. }
```

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不同于 **swapTwoInts**，如下所示：

```
1. func swapTwoInts(inout a: Int, inout b: Int)
2. func swapTwoValues<T>(inout a: T, inout b: T)
```

这个函数的泛型版本使用了占位类型名字（通常此情况下用字母 **T** 来表示）来代替实际类型名（如 **Int**、**String** 或 **Double**）。占位类型名没有提示 **T** 必须是什么类型，但是它提示了 **a** 和 **b** 必须是同一类型 **T**，而不管 **T** 表示什么类型。只有 **swapTwoValues** 函数在每次调用时所传入的实际类型才能决定 **T** 所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的展位类型名字（**T**）是用尖括号括起来的（**<>**）。这个尖括号告诉 **Swift** 那个 **T** 是 **swapTwoValues** 函数所定义的一个类型。因为 **T** 是一个占位命名类型，**Swift** 不会去查找命名为 **T** 的实际类型。

swapTwoValues 函数除了要求传入的两个任何类型值是同一类型外，也可以作为 **swapTwoInts** 函数被调用。每次 **swapTwoValues** 被调用，**T** 所代表的类型值都会传给函数。

在下面的两个例子中，**T** 分别代表 **Int** 和 **String**：

```
1. var someInt = 3
2. var anotherInt = 107
3. swapTwoValues(&someInt, &anotherInt)
4. // someInt is now 107, and anotherInt is now 3
5.
6. var someString = "hello"
7. var anotherString = "world"
8. swapTwoValues(&someString, &anotherString)
9. // someString is now "world", and anotherString is now "hello"
```

注意：上面定义的函数 `swapTwoValues` 是受 `swap` 函数启发而实现的。`swap` 函数存在于 Swift 标准库，并可以在其它类中任意使用。如果你在自己代码中需要类似 `swapTwoValues` 函数的功能，你可以使用已存在的交换函数 `swap` 函数。

类型参数

在上面的 `swapTwoValues` 例子中，占位类型 `T` 是一种类型参数的示例。类型参数指定并命名为一个占位类型，并且紧随在函数名后面，使用一对尖括号括起来（如）。

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。

如果你使用多个参数定义更复杂的泛型函数或泛型类型，那么使用更多的描述类型参数是非常有用的。例如，Swift 字典（`Dictionary`）类型有两个类型参数，一个是键，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。

注意：请始终使用大写字母开头的驼峰式命名法（例如 `T` 和 `KeyType`）来给类型参数命名，以表明它们是类型的占位符，而非类型值。

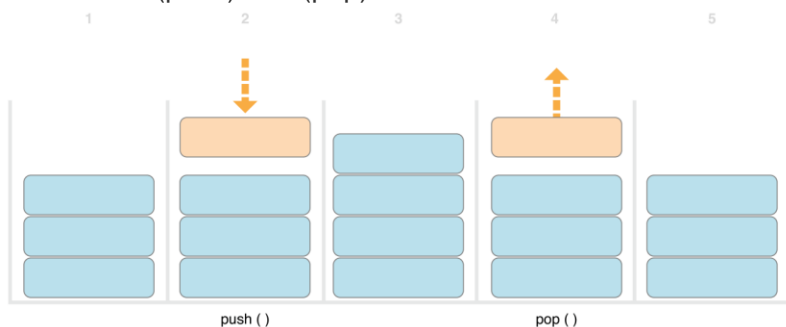
泛型类型

通常在泛型函数中，`Swift` 允许你定义你自己的泛型类型。这些自定义类、结构体和枚举作用于任何类型，如同 `Array` 和 `Dictionary` 的用法。

这部分向你展示如何写一个泛型集类型--`Stack`（栈）。一个栈是一系列值域的集合，和 `Array`（数组）类似，但其是一个比 `Swift` 的 `Array` 类型更多限制的集合。一个数组可以允许其里面任何位置的插入/删除操作，而栈，只允许在集合的末端添加新的项（如同 `push` 一个新值进栈）。同样的一个栈也只能从末端移除项（如同 `pop` 一个值出栈）。

注意：栈的概念已被 `UINavigationController` 类使用来模拟试图控制器的导航结构。你通过调用 `UINavigationController` 的 `pushViewController:animated:` 方法来为导航栈添加（`add`）新的试图控制器；而通过 `popViewControllerAnimated:` 的方法来从导航栈中移除（`pop`）某个试图控制器。每当你需要一个严格的后进先出方式来管理集合，堆栈都是最实用的模型。

下图展示了一个栈的压栈(`push`)/出栈(`pop`)的行为：



- 1、现在有三个值在栈中；
- 2、第四个值“pushed”到栈的顶部；
- 3、现在有四个值在栈中，最近的那个在顶部；

- 4、栈中最顶部的那个项被移除，或称之为“popped”；
- 5、移除掉一个值后，现在栈又重新只有三个值。

这里展示了如何写一个非泛型版本的栈，`Int` 值型的栈：

```
1. struct IntStack {
2.     var items = Int[] ()
3.     mutating func push(item: Int) {
4.         items.append(item)
5.     }
6.     mutating func pop() -> Int {
7.         return items.removeLast()
8.     }
9. }
```

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：

```
1. struct Stack<T> {
2.     var items = T[] ()
3.     mutating func push(item: T) {
4.         items.append(item)
5.     }
6.     mutating func pop() -> T {
7.         return items.removeLast()
8.     }
9. }
```

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。

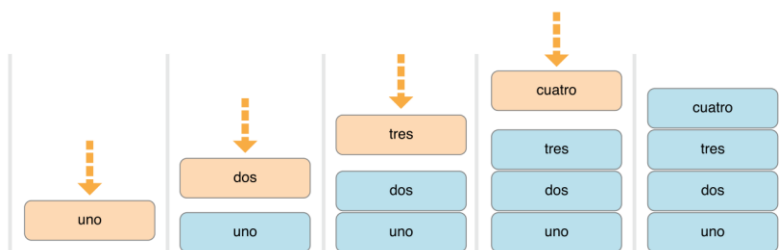
T 定义了一个名为“某种类型 **T**”的节点提供给后来用。这种将来类型可以在结构体的定义里任何地方表示为“**T**”。在这种情况下，**T** 在如下三个地方被用作节点：

- 创建一个名为 **items** 的属性，使用空的 **T** 类型值数组对其进行初始化；
- 指定一个包含一个参数名为 **item** 的 **push** 方法，该参数必须是 **T** 类型；
- 指定一个 **pop** 方法的返回值，该返回值将是一个 **T** 类型值。

当创建一个新单例并初始化时，通过用一对紧随在类型名后的尖括号里写出实际指定栈用到类型，创建一个 **Stack** 实例，同创建 **Array** 和 **Dictionary** 一样：

```
1. var stackOfStrings = Stack<String>()
2. stackOfStrings.push("uno")
3. stackOfStrings.push("dos")
4. stackOfStrings.push("tres")
5. stackOfStrings.push("cuatro")
6. // 现在栈已经有 4 个 string 了
```

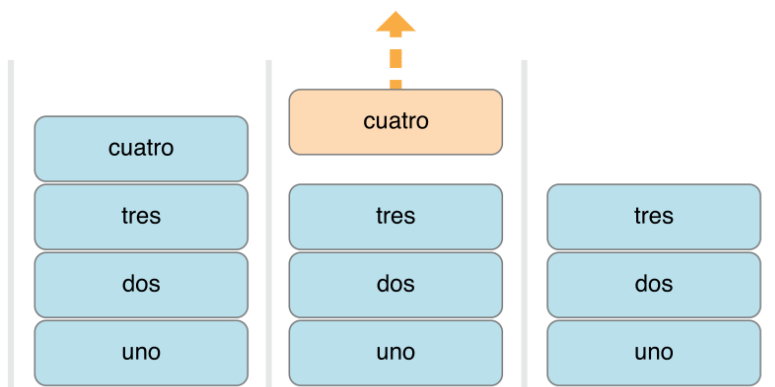
下图将展示 **stackOfStrings** 如何 **push** 这四个值进栈的过程：



从栈中 **pop** 并移除值“cuatro”：

```
1. let fromTheTop = stackOfStrings.pop()
2. // fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

下图展示了如何从栈中 **pop** 一个值的过程：



由于 **Stack** 是泛型类型，所以在 **Swift** 中其可以用来创建任何有效类型的栈，这种方式如同 **Array** 和 **Dictionary**。

类型约束

swapTwoValues 函数和 **Stack** 类型可以作用于任何类型，不过，有的时候对使用在泛型函数和泛型类型上的类型强制约束为某种特定类型是非常有用的。类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

例如，**Swift** 的 **Dictionary** 类型对作用于其键的类型做了些限制。在字典的描述中，字典的键类型必须是可哈希，也就是说，必须有一种方法可以使其是唯一的表示。**Dictionary** 之所以需要其键是可哈希是为了以便于其检查其是否包含某个特定键的值。如无此需求，**Dictionary** 即不会告诉是否插入或者替换了某个特定键的值，也不能查找到已经存储在字典里面的给定键值。

这个需求强制加上一个类型约束作用于 **Dictionary** 的键上，当然其键类型必须遵循 **Hashable** 协议（**Swift** 标准库中定义的一个特定协议）。所有的 **Swift** 基本类型（如 **String**，**Int**，**Double** 和 **Bool**）默认都是可哈希。

当你创建自定义泛型类型时，你可以定义你自己的类型约束，当然，这些约束要支持泛型编程的强力特征中的多数。抽象概念如可哈希具有的类型特征是根据他们概念特征来界定的，而不是他们的直接类型特征。

类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
1. func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {  
2.     // function body goes here  
3. }
```

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**：

```
1. func findStringIndex(array: String[], valueToFind: String) -> Int? {  
2.     for (index, value) in enumerate(array) {  
3.         if value == valueToFind {  
4.             return index  
5.         }  
6.     }  
7.     return nil  
8. }
```

findStringIndex 函数可以作用于查找一字符串数组中的某个字符串：

```
1. let strings = ["cat", "dog", "llama", "parakeet", "terrapiin"]  
2. if let foundIndex = findStringIndex(strings, "llama") {  
3.     println("The index of llama is \(foundIndex)")  
4. }  
5. // 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。

这里展示如何写一个你或许期望的 `findStringIndex` 的泛型版本 `findIndex`。请注意这个函数仍然返回 `Int`，是不是有点迷惑呢，而不是泛型类型？那是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数不会编译，原因在例子后面会说明：

```
1. func findIndex<T>(array: T[], valueToFind: T) -> Int? {
2.     for (index, value) in enumerate(array) {
3.         if value == valueToFind {
4.             return index
5.         }
6.     }
7.     return nil
8. }
```

上面所写的函数不会编译。这个问题的位置在等式的检查上，“`if value == valueToFind`”。不是所有的 **Swift** 中的类型都可以用等式符（`==`）进行比较。例如，如果你创建一个你自己的类或结构体来表示一个复杂的数据模型，那么 **Swift** 没法猜到对于这个类或结构体而言“等于”的意思。正因如此，这部分代码不能可能保证工作于每个可能的类型 `T`，当你试图编译这部分代码时估计会出现相应的错误。

不过，所有的这些并不会让我们无从下手。**Swift** 标准库中定义了一个 **Equatable** 协议，该协议要求任何遵循的类型实现等式符（`==`）和不等符（`!=`）对任何两个该类型进行比较。所有的 **Swift** 标准类型自动支持 **Equatable** 协议。

任何 **Equatable** 类型都可以安全的使用在 `findIndex` 函数中，因为其保证支持等式操作。为了说明这个事实，当你定义一个函数时，你可以写一个 **Equatable** 类型约束作为类型参数定义的一部分：

```
1. func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
2.     for (index, value) in enumerate(array) {
3.         if value == valueToFind {
4.             return index
5.         }
6.     }
7.     return nil
8. }
```

```
6.     }
7.     return nil
8. }
```

`findIndex` 中这个单个类型参数写做：`T: Equatable`，也就意味着“任何 `T` 类型都遵循 `Equatable` 协议”。

`findIndex` 函数现在则可以成功的编译过，并且作用于任何遵循 `Equatable` 的类型，如 `Double` 或 `String`：

```
1. let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
2. // doubleIndex is an optional Int with no value, because 9.3 is not in
   the array
3. let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
4. // stringIndex is an optional Int containing a value of 2
```

关联类型

当定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型给定作用于协议部分的类型一个节点名（或别名）。作用于关联类型上实际类型是不需要指定的，直到该协议接受。关联类型被指定为 `typealias` 关键字。

关联类型行为

这里是一个 `Container` 协议的例子，定义了一个 `ItemType` 关联类型：

```
1. protocol Container {
2.     typealias ItemType
3.     mutating func append(item: ItemType)
4.     var count: Int { get }
5.     subscript(i: Int) -> ItemType { get }
6. }
```

`Container` 协议定义了三个任何容器必须支持的兼容要求：

必须可能通过 `append` 方法添加一个新 `item` 到容器里；

必须可能通过使用 `count` 属性获取容器里 `items` 的数量，并返回一个 `Int` 值；

必须可能通过容器的 `Int` 索引值下标可以检索到每一个 `item`。

这个协议没有指定容器里 `item` 是如何存储的或何种类型是允许的。这个协议只指定三个任何遵循 `Container` 类型所必须支持的功能点。一个遵循的类型也可以提供其他额外的功能，只要满足这三个条件。

任何遵循 `Container` 协议的类型必须指定存储在其里面的值类型，必须保证只有正确类型的 `items` 可以加进容器里，必须明确可以通过其下标返回 `item` 类型。

为了定义这三个条件，`Container` 协议需要一个方法指定容器里的元素将会保留，而不需要知道特定容器的类型。`Container` 协议需要指定任何通过 `append` 方法添加到容器里的值和容器里元素是相同类型，并且通过容器下标返回的容器元素类型的值的类型是相同类型。

为了达到此目的，`Container` 协议声明了一个 `ItemType` 的关联类型，写作 `typealias ItemType`。The protocol does not define what `ItemType` is an alias for—that information is left for any conforming type to provide（这个协议不会定义 `ItemType` 是遵循类型所提供的何种信息的别名）。尽管如此，`ItemType` 别名支持一种方法识别在一个容器里的 `items` 类型，以及定义一种使用在 `append` 方法和下标中的类型，以便保证任何期望的 `Container` 的行为是强制性的。

这里是一个早前 `IntStack` 类型的非泛型版本，适用于遵循 `Container` 协议：

```
1. struct IntStack: Container {
2.     // original IntStack implementation
3.     var items = Int[] ()
4.     mutating func push(item: Int) {
5.         items.append(item)
6.     }
7.     mutating func pop() -> Int {
8.         return items.removeLast()
9.     }
10.    // conformance to the Container protocol
```

```
11.     typealias ItemType = Int
12.     mutating func append(item: Int) {
13.         self.push(item)
14.     }
15.     var count: Int {
16.         return items.count
17.     }
18.     subscript(i: Int) -> Int {
19.         return items[i]
20.     }
21. }
```

`IntStack` 类型实现了 `Container` 协议的所有三个要求，在 `IntStack` 类型的每个包含部分的功能都满足这些要求。

此外，`IntStack` 指定了 `Container` 的实现，适用的 `ItemType` 被用作 `Int` 类型。对于这个 `Container` 协议实现而言，定义 `typealias ItemType = Int`，将抽象的 `ItemType` 类型转换为具体的 `Int` 类型。

感谢 `Swift` 类型参考，你不用在 `IntStack` 定义部分声明一个具体的 `Int` 的 `ItemType`。由于 `IntStack` 遵循 `Container` 协议的所有要求，只要通过简单的查找 `append` 方法的 `item` 参数类型和下标返回的类型，`Swift` 就可以推断出合适的 `ItemType` 来使用。确实，如果上面的代码中你删除了 `typealias ItemType = Int` 这一行，一切仍旧可以工作，因为它清楚的知道 `ItemType` 使用的是何种类型。

你也可以生成遵循 `Container` 协议的泛型 `Stack` 类型：

```
1. struct Stack<T>: Container {
2.     // original Stack<T> implementation
3.     var items = T[]()
4.     mutating func push(item: T) {
5.         items.append(item)
6.     }
7.     mutating func pop() -> T {
8.         return items.removeLast()
9.     }
10.    // conformance to the Container protocol
11.    mutating func append(item: T) {
```

```
12.         self.push(item)
13.     }
14.     var count: Int {
15.         return items.count
16.     }
17.     subscript(i: Int) -> T {
18.         return items[i]
19.     }
20. }
```

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

在使用扩展来添加协议兼容性中有描述扩展一个存在的类型添加遵循一个协议。这个类型包含一个关联类型的协议。

`Swift` 的 `Array` 已经提供 `append` 方法，一个 `count` 属性和通过下标来查找一个自己的元素。这三个功能都达到 `Container` 协议的要求。也就意味着你可以扩展 `Array` 去遵循 `Container` 协议，只要通过简单声明 `Array` 适用于该协议而已。如何实践这样一个空扩展，在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：

```
1. 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。你可写一个 `where` 语句，通过

紧随放置 **where** 关键字在类型参数队列后面，其后跟着一个或者多个针对关联类型的约束，以及（或）一个或多个类型和关联类型的等于关系。

下面的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：

```
1. func allItemsMatch<
2.     C1: Container, C2: Container
3.     where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
4.     (someContainer: C1, anotherContainer: C2) -> Bool {
5.
6.         // check that both containers contain the same number of items
7.
8.         if someContainer.count != anotherContainer.count {
9.             return false
10.        }
11.
12.        // check each pair of items to see if they are equivalent
13.        for i in 0..someContainer.count {
14.            if someContainer[i] != anotherContainer[i] {
15.                return false
16.            }
17.        }
18.
19.        // all items match, so return true
20.        return true
21. }
```

这个函数用了两个参数：**someContainer** 和 **anotherContainer**。**someContainer** 参数是类型 **C1**，**anotherContainer** 参数是类型 **C2**。**C1** 和 **C2** 是容器的两个占位类型参数，决定了这个函数何时被调用。

这个函数的类型参数列紧随在两个类型参数需求的后面：

C1 必须遵循 Container 协议 (写作 C1: Container)。

C2 必须遵循 Container 协议 (写作 C2: Container)。

C1 的 ItemType 同样是 C2 的 ItemType (写作 C1.ItemType == C2.ItemType)。

C1 的 ItemType 必须遵循 Equatable 协议 (写作 C1.ItemType: Equatable)。

第三个和第四个要求被定义为一个 **where** 语句的一部分，写在关键字 **where** 后面，作为函数类型参数链的一部分。

这些要求意思是：

someContainer 是一个 C1 类型的容器。**anotherContainer** 是一个 C2 类型的容器。

someContainer 和 **anotherContainer** 包含相同的元素类型。**someContainer** 中的元素可以通过不等于操作(**!=**)来检查它们是否彼此不同。

第三个和第四个要求结合起来的意思是 **anotherContainer** 中的元素也可以通过 **!=** 操作来检查，因为他们在 **someContainer** 中元素确实是相同的类型。

这些要求能够使 **allItemsMatch** 函数比较两个容器，即便他们是不同的容器类型。

allItemsMatch 首先检查两个容器是否拥有同样数目的 **items**，如果他们的元素数目不同，没有办法进行匹配，函数就会 **false**。

检查完之后，函数通过 **for-in** 循环和半闭区间操作 (**..**) 来迭代 **someContainer** 中的所有元素。对于每个元素，函数检查是否 **someContainer** 中的元素不等于对应的 **anotherContainer** 中的元素，如果这两个元素不等，则这两个容器不匹配，返回 **false**。

如果循环体结束后未发现没有任何的不匹配，那表明两个容器匹配，函数返回 **true**。

这里演示了 **allItemsMatch** 函数运算的过程：

```
1. var stackOfStrings = Stack<String>()
2. stackOfStrings.push("uno")
3. stackOfStrings.push("dos")
4. stackOfStrings.push("tres")
```

```
5.
6. var arrayOfStrings = ["uno", "dos", "tres"]
7.
8. if allItemsMatch(stackOfStrings, arrayOfStrings) {
9.     println("All items match.")
10. } else {
11.     println("Not all items match.")
12. }
13. // 输出 "All items match."
```

上面的例子创建一个 **Stack** 单例来存储 **String**，然后压了三个字符串进栈。这个例子也创建了一个 **Array** 单例，并初始化包含三个同栈里一样的原始字符串。即便栈和数组否是不同的类型，但他们都遵循 **Container** 协议，而且他们都包含同样的类型值。你因此可以调用 **allItemsMatch** 函数，用这两个容器作为它的参数。在上面的例子中，**allItemsMatch** 函数正确的显示了所有的这两个容器的 **items** 匹配。

2.23 高级运算符

本篇译者：xielingwang ([Github 主页](#))，敬请勘误。

除了[基本操作符](#)中所讲的运算符，**Swift** 还有许多复杂的高级运算符，包括了 **C** 语和 **Objective-C** 中的位运算符和移位运算。

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，也可以运用之前定制的运算符。

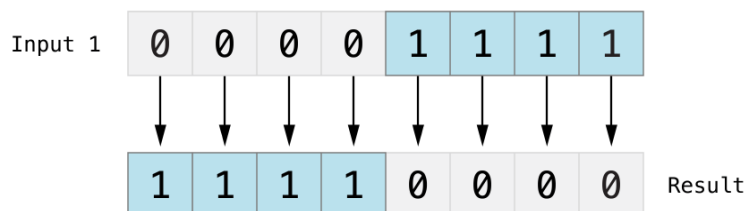
位运算符

位操作符通常在诸如图像处理和创建设备驱动等底层开发中使用，使用它可以单独操作数据结构中原始数据的比特位。在使用一个自定义的协议进行通信的时候，运用位运算符来对原始数据进行编码和解码也是非常有效的。

Swift 支持如下所有 C 语言的位运算符：

按位取反运算符

按位取反运算符~对一个操作数的每一位都取反。



这个运算符是前置的，所以请不添加任何空格地写着操作数之前。

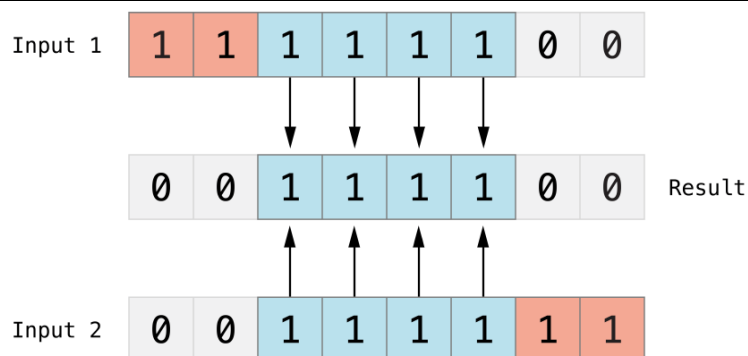
```
1. let initialBits: UInt8 = 0b00001111
2. let invertedBits = ~initialBits // 等于 0b11110000
```

UInt8 是 8 位无符整型，可以存储 0~255 之间的任意数。这个例子初始化一个整型为二进制值 00001111(前 4 位为 0，后 4 位为 1)，它的十进制值为 15。

使用按位取反运算~对 initialBits 操作，然后赋值给 invertedBits 这个新常量。这个新常量的值等于所有位都取反的 initialBits，即 1 变成 0，0 变成 1，变成了 11110000，十进制值为 240。

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。

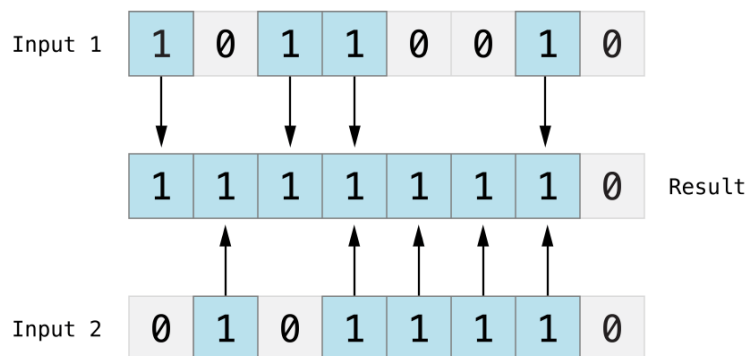


以下代码，`firstSixBits` 和 `lastSixBits` 中间 4 个位都为 1。对它俩进行按位与运算后，就得到了 00111100，即十进制的 60。

```
1. let firstSixBits: UInt8 = 0b11111100
2. let lastSixBits: UInt8 = 0b00111111
3. let middleFourBits = firstSixBits & lastSixBits // 等于 00111100
```

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)。

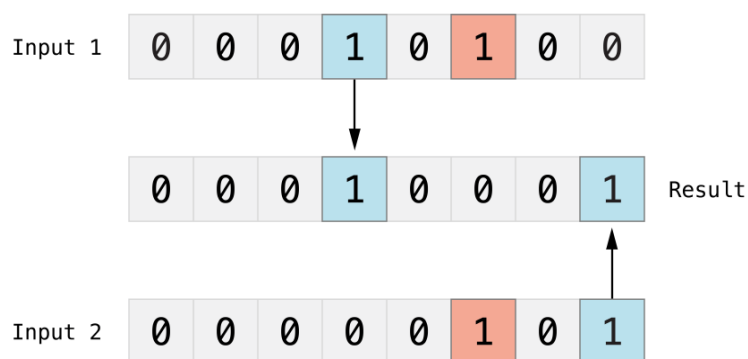


如下代码，`someBits` 和 `moreBits` 在不同位上有 1。按位或运行的结果是 11111110，即十进制的 254。

```
1. let someBits: UInt8 = 0b10110010
2. let moreBits: UInt8 = 0b01011110
3. 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。



以下代码，`firstBits` 和 `otherBits` 都有一个 1 跟另一个数不同的。所以按位异或的结果是把它这些位置为 1，其他都置为 0。

```
1. let firstBits: UInt8 = 0b00010100
2. let otherBits: UInt8 = 0b00000101
3. 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。

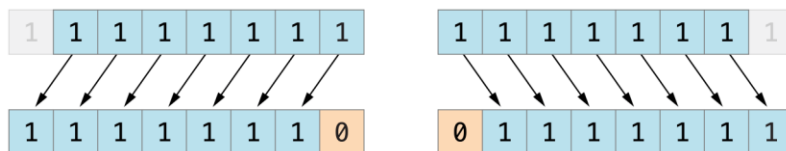
按位左移和按位右移的效果相当把一个整数乘于或除于一个因子为 2 的整数。向左移动一个整型的比特位相当于把这个数乘于 2，向右移一位就是除于 2。

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。

以下这张把展示了 $11111111 \ll 1$ (11111111 向左移 1 位), 和 $11111111 \gg 1$ (11111111 向右移 1 位)。蓝色的是被移位的, 灰色是被抛弃的, 橙色的 0 是被填充进来的。



```
1. let shiftBits: UInt8 = 4    // 即二进制的 00000100
2. shiftBits << 1              // 00001000
3. shiftBits << 2              // 00010000
4. shiftBits << 5              // 10000000
5. shiftBits << 6              // 00000000
6. shiftBits >> 2              // 00000001
```

你可以使用移位操作进行其他数据类型的编码和解码。

```
1. let pink: UInt32 = 0xCC6699
2. let redComponent = (pink & 0xFF0000) >> 16    // redComponent 是 0xCC,
   即 204
3. let greenComponent = (pink & 0x00FF00) >> 8    // greenComponent 是 0x66,
   即 102
4. 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。

对 `0xCC6699` 和 `0xFF0000` 进行按位与`&`操作就可以得到红色部分。`0xFF0000` 中的 0 了遮盖了 `0xCC6699` 的第二和第三个字节, 这样 `6699` 被忽略了, 只留下 `0xCC0000`。

然后, 按向右移动 16 位, 即 `>> 16`。十六进制中每两个字符是 8 比特位, 所以移动 16 位的结果是把 `0xCC0000` 变成 `0x0000CC`。这和 `0xCC` 是相等的, 都是十进制的 204。

同样的, 绿色部分来自于 `0xCC6699` 和 `0x00FF00` 的按位操作得到 `0x006600`。然后向右移动 8 們, 得到 `0x66`, 即十进制的 102。

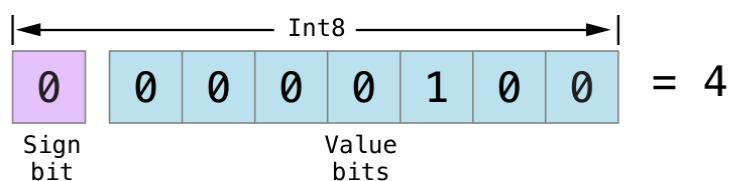
最后，蓝色部分对 `0xCC6699` 和 `0x0000FF` 进行按位与运算，得到 `0x000099`，无需向右移位了，所以结果就是 `0x99`，即十进制的 153。

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。

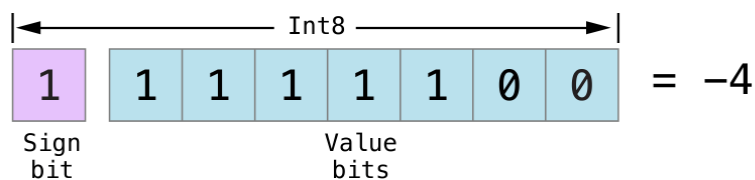
其余的比特位(称为数值位)存储其实值。有符正整数和无符正整数在计算机里的存储结果是一样的，下面我们来看+4 内部的二进制结构。



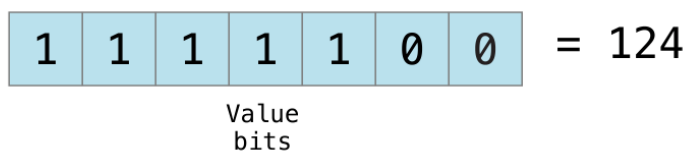
符号位为 0，代表正数，另外 7 比特位二进制表示的实际值就刚好是 4。

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。

我们来看-4 存储的二进制结构。

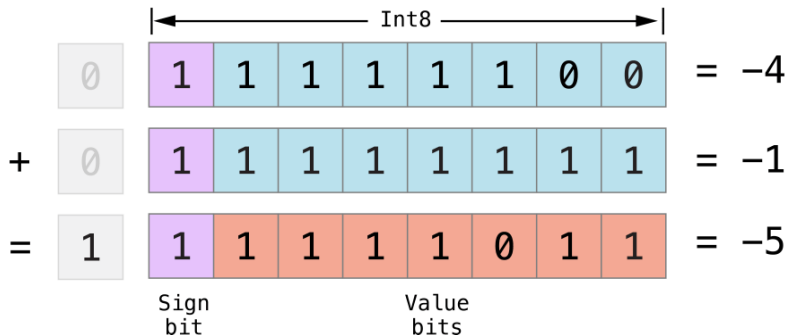


现在符号位为 1，代表负数，7 个数值位要表达的二进制值是 124，即 $128 - 4$ 。



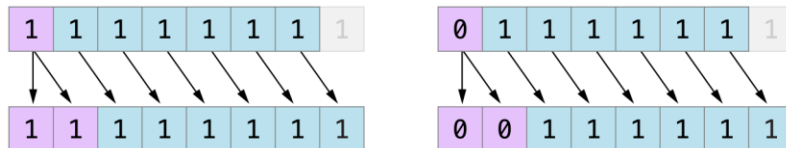
负数的编码方式称为二进制补码表示。这种表示方式看起来很奇怪，但它有几个优点。

首先，只需要对全部 8 个比特位(包括符号)做标准的二进制加法就可以完成 $-1 + -4$ 的操作，忽略加法过程产生的超过 8 个比特位表达的任何信息。



第二，由于使用二进制补码表示，我们可以和正数一样对负数进行按位左移右移的，同样也是左移 1 位时乘以 2，右移 1 位时除以 2。要达到此目的，对有符整型的右移有一个特别的要求：

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。

例如，Int16 整型能承载的整数范围是 -32768 到 32767，如果给它赋上超过这个范围的数，就会报错：

```
1. var potentialOverflow = Int16.max
```

```
2. // potentialOverflow 等于 32767, 这是 Int16 能承载的最大整数
3. potentialOverflow += 1
4. // 噢, 出错了
```

对过大或过小的数值进行错误处理让你的数值边界条件更灵活。

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。**Swift** 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 **&/**

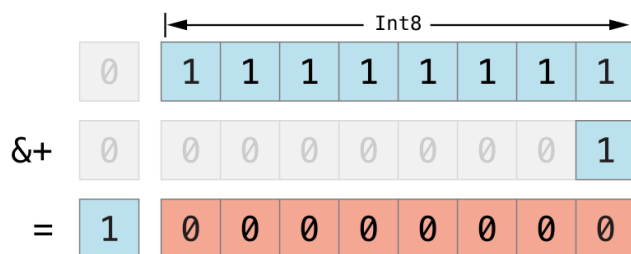
溢出求余 **&%**

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

```
1. var willOverflow = UInt8.max
2. // willOverflow 等于 UInt8 的最大整数 255
3. willOverflow = willOverflow &+ 1
4. // 这时候 willOverflow 等于 0
```

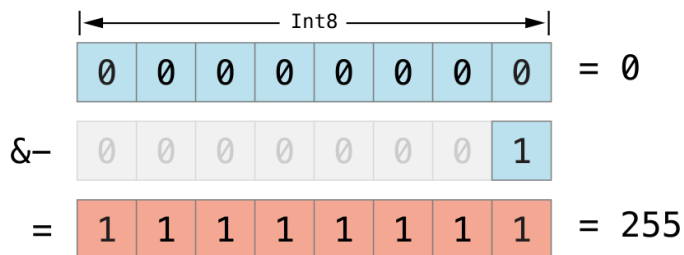
willOverflow 用 **Int8** 所能承载的最大值 255(二进制 11111111)，然后用**&+**加 1。然后 **UInt8** 就无法表达这个新值的二进制了，也就导致了这个新值上溢出了，大家可以看下图。溢出后，新值在 **UInt8** 的承载范围内的那部分是 00000000，也就是 0。



值的下溢出

数值也有可能因为太小而越界。举个例子：

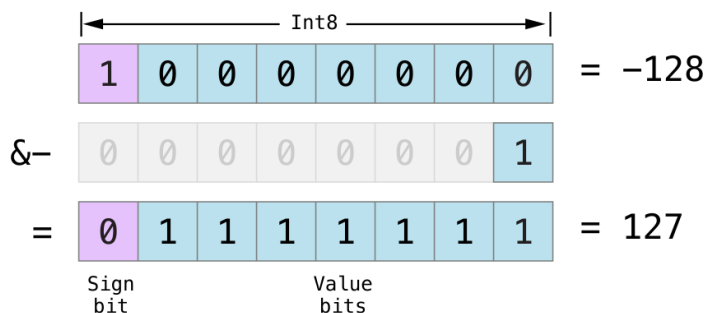
UInt8 的最小值是 0(二进制为 00000000)。使用&-进行溢出减 1，就会得到二进制的 11111111 即十进制的 255。



Swift 代码是这样的：

```
1. var willUnderflow = UInt8.min
2. // willUnderflow 等于 UInt8 的最小值 0
3. willUnderflow = willUnderflow &- 1
4. // 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 1 后，变成了 01111111，即 UInt8 所能承载的最大整数 127。



来看看 Swift 代码：

```
1. var signedUnderflow = Int8.min
2. // signedUnderflow 等于最小的有符整数 -128
3. signedUnderflow = signedUnderflow &- 1
4. // 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。

```
1. let x = 1
2. 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。

```
1. let x = 1
2. let y = x &/ 0
3. // 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。

结合性定义相同优先级的运算符在一起时是怎么组合或关联的，是和左边的一组呢，还是和右边的一组。意思就是，到底是和左边的表达式结合呢，还是和右边的表达式结合？

在混合表达式中，运算符的优先级和结合性是非常重要的。举个例子，为什么下列表达式的结果为 4？

```
1. 2 + 3 * 4 % 5
2. // 结果是 4
```

如果严格地从左计算到右，计算过程会是这样：

2 plus 3 equals 5;

$2 + 3 = 5$

5 times 4 equals 20;

$5 * 4 = 20$

20 remainder 5 equals 0

$20 / 5 = 4$ 余 0

但是正确答案是 **4** 而不是 **0**。优先级高的运算符要先计算，在 **Swift** 和 **C** 语言中，都是先乘除后加减的。所以，执行完乘法和求余运算才能执行加减运算。

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。

```
1. 2 + ((3 * 4) % 5)
```

$(3 * 4)$ is 12, so this is equivalent to: $3 * 4 = 12$ ，所以这相当于：

```
1. 2 + (12 % 5)
```

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

```
1. 2 + 2
```

计算结果为 **4**。

查阅 **Swift** 运算符的优先级和结合性的完整列表，请看表达式。

注意：**Swift** 的运算符较 **C** 语言和 **Objective-C** 来得更简单和保守，这意味着跟基于 **C** 的语言可能不一样。所以，在移植已有代码到 **Swift** 时，注意去确保代码按你想的那样去执行。

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。

```
1. struct Vector2D {  
2.     var x = 0.0, y = 0.0
```

```
3. }
4. @infix func + (left: Vector2D, right: Vector2D) -> Vector2D {
5.     return Vector2D(x: left.x + right.x, y: left.y + right.y)
6. }
```

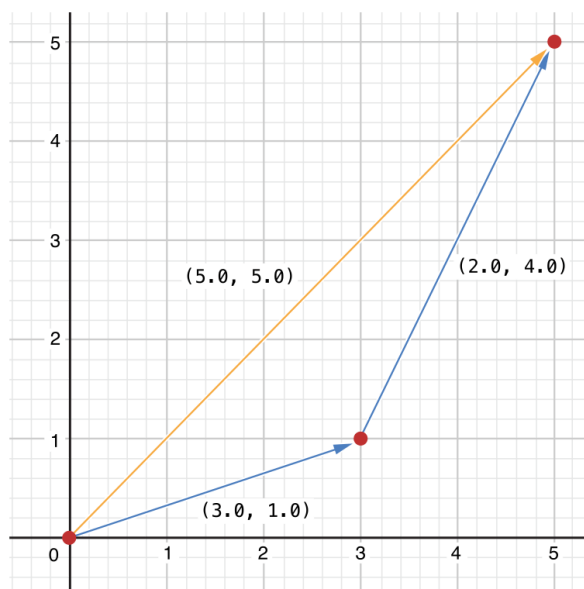
该运算符函数定义了一个全局的+函数，这个函数需要两个 **Vector2D** 类型的参数，返回值也是 **Vector2D** 类型。需要定义和实现一个中置运算的时候，在关键字 **func** 之前写上属性 **@infix** 就可以了。

在这个代码实现中，参数被命名为了 **left** 和 **right**，代表+左边和右边的两个 **Vector2D** 对象。函数返回了一个新的 **Vector2D** 的对象，这个对象的 **x** 和 **y** 分别等于两个参数对象的 **x** 和 **y** 的和。

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。

```
1. let vector = Vector2D(x: 3.0, y: 1.0)
2. let anotherVector = Vector2D(x: 2.0, y: 4.0)
3. let combinedVector = vector + anotherVector
4. // combinedVector 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。如下图所示：



前置和后置运算符

上个例子演示了一个双目中置运算符的自定义实现，同样我们也可以玩标准单目运算符的实现。单目运算符只有一个操作数，在操作数之前就是前置的，如 `-a`；在操作数之后就是后置的，如 `i++`。

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。

```
1. @prefix func - (vector: Vector2D) -> Vector2D {
2.     return Vector2D(x: -vector.x, y: -vector.y)
3. }
```

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` 都进行单目减运算。

```
1. let positive = Vector2D(x: 3.0, y: 4.0)
2. let negative = -positive
3. // negative 为 (-3.0, -4.0)
4. let alsoPositive = -negative
5. // 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。

```
1. @assignment func += (inout left: Vector2D, right: Vector2D) {
2.     left = left + right
3. }
```

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。

```
1. var original = Vector2D(x: 1.0, y: 2.0)
2. let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
3. original += vectorToAdd
4. // original 现在为 (4.0, 6.0)
```

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

```
1. @prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
2.     vector += Vector2D(x: 1.0, y: 1.0)
3.     return vector
4. }
```

这个前置使用了已经定义好的高级加赋运算，将自己加上一个值为 `(1.0, 1.0)` 的对象然后赋给自己，然后再将自己返回。

```
1. var toIncrement = Vector2D(x: 3.0, y: 4.0)
2. let afterIncrement = ++toIncrement
3. // toIncrement 现在是 (4.0, 5.0)
4. // 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。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：

```
1. @infix func == (left: Vector2D, right: Vector2D) -> Bool {
2.     return (left.x == right.x) && (left.y == right.y)
3. }
4.
5. @infix func != (left: Vector2D, right: Vector2D) -> Bool {
6.     return !(left == right)
7. }
```

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。

```
1. let twoThree = Vector2D(x: 2.0, y: 3.0)
2. let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
3. if twoThree == anotherTwoThree {
4.     println("这两个向量是相等的.")
5. }
6. // 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。

```
1. 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己加上自己然后返回的运算。

```
1. @prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
2.     vector += vector
3.     return vector
4. }
```

`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。

```
1. var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
2. let afterDoubling = +++toBeDoubled
3. // toBeDoubled 现在是 (2.0, 8.0)
```

```
4. // afterDoubling 现在也是 (2.0, 8.0)
```

自定义中置运算符的优先级和结合性

可以为自定义的中置运算符指定优先级和结合性。可以回头看看优先级和结合性解释这两个因素是如何影响多种中置运算符混合的表达式的计算的。

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**。

```
1. operator infix +- { associativity left precedence 140 }
2. func +- (left: Vector2D, right: Vector2D) -> Vector2D {
3.     return Vector2D(x: left.x + right.x, y: left.y - right.y)
4. }
5. let firstVector = Vector2D(x: 1.0, y: 2.0)
6. let secondVector = Vector2D(x: 3.0, y: 4.0)
7. let plusMinusVector = firstVector +- secondVector
8. // 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**)。查阅完整的 **Swift** 默认结合性和优先级的设置, 请移步表达式;

3 语言附注

3.1 关于语言附注

本篇译者: ChildhoodAndy ([博客](#)), 敬请勘误。

关于语言附注

本书的这一节描述了 **Swift** 编程语言的形式语法。这里描述的语法是为了帮您更详细地了解该语言，而不是让您直接实现一个解析器或编译器。

Swift 语言相对较小，这是由于在 **Swift** 代码中几乎无处不在的许多常见的类型、函数以及运算符都由 **Swift** 标准库来定义。虽然这些类型、函数和运算符不是 **Swift** 语言本身的一部分，但是它们广泛地被用于本书的讨论和代码范例中。

如何阅读语法

用来描述 **Swift** 编程语言形式语法的符号遵循下面几个约定：

1. 箭头（→）用来标记语法产式，可以被理解为“可以包含”。
2. 句法范畴由斜体文字表示，并出现在一个语法产式规则两侧。
3. 义词和标点符号由粗体固定宽度的文本显示和只出现在一个语法产式规则的右边。
4. 选择性的语法产式由竖线（|）分隔。当可选用的语法产式太长难以阅读时，它们将被拆分为多行语法产式规则。
5. 在少数情况下，常规字体文字用来描述语法产式规则的右边。
6. 可选的句法范畴和字面量用尾标 **opt** 来标记。

•

举个例子，**getter-setter** 的语法块的定义如下：

GRAMMAR OF A GETTER-SETTER BLOCK

`getter-setter-block` → { `getter-clause` `setter-clause` `opt` }- | { `setter-clause` `getter-clause` }-

这个定义表明，一个 **getter-setter** 方法块可以由一个 **getter** 子句后跟一个可选的 **setter** 子句构成，用大括号括起来，或者由一个 **setter** 子句后跟一个 **getter** 子句构成，用大括号括起来。上述的文法产生等价于下面的两个产生，明确阐明如何二择一：

GRAMMAR OF A GETTER-SETTER BLOCK

`getter-setter-block` → { `getter-clause` `setter-clause` `opt` }--

`getter-setter-block` → { `setter-clause` `getter-clause` }-

3.2 词法结构

本篇译者：superkam ([git 主页](#))，敬请勘误。

Swift 的“词法结构 (lexical structure)”描述了如何在该语言中用字符序列构建合法标记，组成该语言中最底层的代码块，并在之后的章节中用于描述语言的其他部分。

通常，标记在随后介绍的语法约束下，由 Swift 源文件的输入文本中提取可能的最长子串生成。这种方法称为“最长匹配项 (longest match)”，或者“最大适合” (maximal munch)。

空白与注释

空白 (whitespace) 有两个用途：分隔源文件中的标记和区分运算符属于前缀还是后缀，(参见 运算符) 在其他情况下则会被忽略。以下的字符会被当作空白：空格 (space) (U+0020)、换行符 (line feed) (U+000A)、回车符 (carriage return) (U+000D)、水平 tab (horizontal tab) (U+0009)、垂直 tab (vertical tab) (U+000B)、换页符 (form feed) (U+000C) 以及空 (null) (U+0000)。

注释 (comments) 被编译器当作空白处理。单行注释由 `//` 开始直到该行结束。多行注释由 `/*` 开始，以 `*/` 结束。可以嵌套注释，但注意注释标记必须匹配。

标识符

标识符 (identifiers) 可以由以下的字符开始：大写或小写的字母 **A** 到 **Z**、下划线 `_`、基本多语言面 (Basic Multilingual Plane) 中的 Unicode 非组合字符以及基本多语言面以外的非专用区 (Private Use Area) 字符。首字符之后，标识符允许使用数字和 Unicode 字符组合。

使用保留字 (reserved word) 作为标识符，需要在其前后增加反引号 ```。例如，`class` 不是合法的标识符，但可以使用 ``class``。反引号不属于标识符的一部分，``x`` 和 `x` 表示同一标识符。

闭包（closure）中如果没有明确指定参数名称，参数将被隐式命名为 \$0、\$1、\$2... 这些命名在闭包作用域内是合法的标识符。

标识符语法

identifier → [identifier-headidentifier-characters_{opt}](#)

identifier → ` [identifier-headidentifier-characters_{opt}](#) `

identifier → [implicit-parameter-name](#)

identifier-list → [identifier](#) [identifier](#) , *identifier-list*

identifier-head → **Upper- or lowercase letter A through Z**

identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA

identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF

identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF

identifier-head → U+1E00–U+1FFF

identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F

identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793

identifier-head → U+2C00–U+2DFF or U+2E80–U+2FFF

identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF

identifier-head → U+F900–U+FD3D, U+FD40–U+FDCE, U+FE00–U+FE1F, or U+FE30–U+FE44

identifier-head → U+FE47–U+FFFF

identifier-head → U+10000–U+1FFFF, U+20000–U+2FFFF, U+30000–U+3FFFF, or U+40000–U+4FFFF

identifier-head → U+50000–U+5FFFF, U+60000–U+6FFFF, U+70000–U+7FFFF, or U+80000–U+8FFFF

identifier-head → U+90000–U+9FFFF, U+A0000–U+AFFFD, U+B0000–U+BFFFF, or U+C0000–U+CFFFF

identifier-head → U+D0000–U+DFFFF or U+E0000–U+EFFFF

identifier-character → Digit 0 through 9

identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F

identifier-character → [identifier-head](#)

identifier-characters → [identifier-characteridentifier-characters_{opt}](#)

implicit-parameter-name → \$[decimal-digits](#)

关键字

被保留的关键字（**keywords**）不允许用作标识符，除非被反引号转义，参见 标识符。

1. 用作声明的关键字：`class`、`deinit`、`enum`、`extension`、`func`、`import`、`init`、`let`、`protocol`、`static`、`struct`、`subscript`、`typealias`、`var`
2. 用作语句的关键字：`break`、`case`、`continue`、`default`、`do`、`else`、`fallthrough`、`if`、`in`、`for`、`return`、`switch`、`where`、`while`
3. 用作表达和类型的关键字：`as`、`dynamicType`、`is`、`new`、`super`、`self`、`Self`、`Type`、`__COLUMN__`、`__FILE__`、`__FUNCTION__`、`__LINE__`
4. 特定上下文中被保留的关键字：`associativity`、`didSet`、`get`、`infix`、`inout`、`left`、`mutating`、`none`、`nonmutating`、`operator`、`override`、`postfix`、`precedence`、`prefix`、`right`、`set`、`unowned`、`unowned(safe)`、`unowned(unsafe)`、`weak`、`willSet`，这些关键字在特定上下文之外可以用于标识符。

字面量

字面值表示整型、浮点型数字或文本类型的值，举例如下：

```
1. 42                // 整型字面量
2. 3.14159           // 浮点型字面量
3. "Hello, world!"   // 文本型字面量
```

字面量语法： *literal* → [integer-literal](#) [floating-point-literal](#) [string-literal](#)

整型字面量

整型字面量（**integer literals**）表示未指定精度整型数的值。整型字面量默认用十进制表示，可以加前缀来指定其他的进制，二进制字面量加 `0b`，八进制字面量加 `0o`，十六进制字面量加 `0x`。

十进制字面量包含数字 `0` 至 `9`。二进制字面量只包含 `0` 或 `1`，八进制字面量包含数字 `0` 至 `7`，十六进制字面量包含数字 `0` 至 `9` 以及字母 `A` 至 `F`（大小写均可）。

负整数的字面量在数字前加减号 `-`，比如 `-42`。

允许使用下划线 `_` 来增加数字的可读性，下划线不会影响字面量的值。整型字面量也可以在数字前加 `0`，同样不会影响字面量的值。

```
1. 1000_000    // 等于 1000000
2. 005         // 等于 5
```

除非特殊指定，整型字面量的默认类型为 **Swift** 标准库类型中的 `Int`。**Swift** 标准库还定义了其他不同长度以及是否带符号的整数类型，请参考 [整数类型](#)。

整型字面量语法

integer-literal → [binary-literal](#)

integer-literal → [octal-literal](#)

integer-literal → [decimal-literal](#)

integer-literal → [hexadecimal-literal](#)

binary-literal → `0b`[binary-digitbinary-literal-characters](#)_{opt}

binary-digit → Digit 0 or 1

binary-literal-character → [binary-digit](#)

binary-literal-characters → [binary-literal-characterbinary-literal-characters](#)_{opt}

octal-literal → `0o`[octal-digitoctal-literal-characters](#)_{opt}

octal-digit → Digit 0 through 7

octal-literal-character → [octal-digit](#)

octal-literal-characters → [octal-literal-characteroctal-literal-characters](#)_{opt}

decimal-literal → [decimal-digitdecimal-literal-characters](#)_{opt}

decimal-digit → Digit 0 through 9

decimal-digits → [decimal-digitdecimal-digits](#)_{opt}

decimal-literal-character → [decimal-digit](#)

decimal-literal-characters → [decimal-literal-characterdecimal-literal-characters](#)_{opt}

hexadecimal-literal → `0x`[hexadecimal-digithexadecimal-literal-characters](#)_{opt}

hexadecimal-digit → Digit 0 through 9, a through f, or A through F

hexadecimal-literal-character → [hexadecimal-digit](#)

hexadecimal-literal-characters → [hexadecimal-literal-characterhexadecimal-literal-characters](#)_{opt}

浮点型字面量

浮点型字面量（floating-point literals）表示未指定精度浮点数的值。

浮点型字面量默认用十进制表示（无前缀），也可以用十六进制表示（加前缀 `0x`）。

十进制浮点型字面量（**decimal floating-point literals**）由十进制数字串后跟小数部分或指数部分（或两者皆有）组成。十进制小数部分由小数点 `.` 后跟十进制数字串组成。指数部分由大写或小写字母 `e` 后跟十进制数字串组成，这串数字表示 `e` 之前的数量乘以 `10` 的几次方。例如：`1.25e2` 表示 1.25×10^2 ，也就是 `125.0`；同样，`1.25e-2` 表示 1.25×10^{-2} ，也就是 `0.0125`。

十六进制浮点型字面量（**hexadecimal floating-point literals**）由前缀 `0x` 后跟可选的十六进制小数部分以及十六进制指数部分组成。十六进制小数部分由小数点后跟十六进制数字串组成。指数部分由大写或小写字母 `p` 后跟十进制数字串组成，这串数字表示 `p` 之前的数量乘以 `2` 的几次方。例如：`0xFp2` 表示 15×2^2 ，也就是 `60`；同样，`0xFp-2` 表示 15×2^{-2} ，也就是 `3.75`。

与整型字面量不同，负的浮点型字面量由一元运算符减号 `-` 和浮点型字面量组成，例如 `-42.0`。这代表一个表达式，而不是一个浮点整型字面量。

允许使用下划线 `_` 来增强可读性，下划线不会影响字面量的值。浮点型字面量也可以在数字前加 `0`，同样不会影响字面量的值。

```
1. 10_000.56      // 等于 10000.56
2. 005000.76      // 等于 5000.76
```

除非特殊指定，浮点型字面量的默认类型为 **Swift** 标准库类型中的 `Double`，表示 64 位浮点数。**Swift** 标准库也定义 `Float` 类型，表示 32 位浮点数。

浮点型字面量语法

floating-point-literal → [decimal-literal](#)[decimal-fraction](#)_{opt}[decimal-exponent](#)_{opt}

floating-point-literal → [hexadecimal-literal](#)[hexadecimal-fraction](#)_{opt}[hexadecimal-exponent](#)

decimal-fraction → [.decimal-literal](#)

decimal-exponent → [floating-point-esign](#)_{opt}[decimal-literal](#)

hexadecimal-fraction → [.hexadecimal-literal](#)_{opt}

hexadecimal-exponent → [floating-point-psign](#)_{opt}[hexadecimal-literal](#)

floating-point-e → `e` `E`

floating-point-p → `p` `P`

sign → `+` `-`

文本型字面量

文本型字面量（**string literal**）由双引号中的字符串组成，形式如下：

```
1. "characters"
```

文本型字面量中不能包含未转义的双引号 "、未转义的反斜线\、回车符（**carriage return**）或换行符（**line feed**）。

可以在文本型字面量中使用的转义特殊符号如下：

1. 空字符（**Null Character**）\0
2. 反斜线（**Backslash**）\\
3. 水平 Tab（**Horizontal Tab**）\t
4. 换行符（**Line Feed**）\n
5. 回车符（**Carriage Return**）\r
6. 双引号（**Double Quote**）\"
7. 单引号（**Single Quote**）\'

字符也可以用以下方式表示：

1. \x 后跟两位十六进制数字
2. \u 后跟四位十六进制数字
3. \U 后跟八位十六进制数字

后跟的数字表示一个 **Unicode** 码点。

文本型字面量允许在反斜线小括号 \() 中插入表达式的值。插入表达式（**interpolated expression**）不能包含未转义的双引号 "、反斜线 \、回车符或者换行符。表达式值的类型必须在 **String** 类中有对应的初始化方法。

例如，以下所有文本型字面量的值相同：

```
1. "1 2 3"
2. "1 2 \ (3) "
3. "1 2 \ (1 + 2) "
4. var x = 3; "1 2 \ (x) "
```

文本型字面量的默认类型为 `String`。组成字符串的字符类型为 `Character`。更多有关 `String` 和 `Character` 的信息请参照 字符串和字符。

文本型字面量语法

string-literal → "[quoted-text](#)"

quoted-text → [quoted-text-item](#)[quoted-text](#)_{opt}

quoted-text-item → [escaped-character](#)

quoted-text-item → \ ([expression](#))

quoted-text-item → Any Unicode extended grapheme cluster except `"`, `\`, `U+000A`, or `U+000D`

escaped-character → `\0` `\\` `\t` `\n` `\r` `\"` `\'`

escaped-character → `\x`[hexadecimal-digit](#)[hexadecimal-digit](#)

escaped-character → `\u`[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)

escaped-character → `\v`[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)[hexadecimal-digit](#)

运算符

`Swift` 标准库定义了许多可供使用的运算符，其中大部分在 基础运算符 和 高级运算符 中进行了阐述。这里将描述哪些字符能用作运算符。

运算符由一个或多个以下字符组成：`/`、`=`、`-`、`+`、`!`、`*`、`%`、`<`、`>`、`&`、`|`、`^`、`~`、`.`。也就是说，标记 `=`、`->`、`//`、`/*`、`*/`、`.` 以及一元前缀运算符 `&` 属于保留字，这些标记不能被重写或用于自定义运算符。

运算符两侧的空白被用来区分该运算符是否为前缀运算符（`prefix operator`）、后缀运算符（`postfix operator`）或二元运算符（`binary operator`）。规则总结如下：

- 1.如果运算符两侧都有空白或两侧都无空白，将被看作二元运算符。例如：`a+b` 和 `a + b` 中的运算符 `+` 被看作二元运算符。
- 2.如果运算符只有左侧空白，将被看作前缀一元运算符。例如 `a ++b` 中的 `++` 被看作前缀一元运算符。
- 3.如果运算符只有右侧空白，将被看作后缀一元运算符。例如 `a++ b` 中的 `++` 被看作后缀一元运算符。

一元运算符。

4.如果运算符左侧没有空白并紧跟 `.`，将被看作后缀一元运算符。例如 `a++b` 中的 `++` 被看作后缀一元运算符（同理，`a++ . b` 中的 `++` 是后缀一元运算符而 `a ++ . b` 中的 `++` 不是）。

鉴于这些规则，运算符前的字符 `(`、`[` 和 `{`；运算符后的字符 `)`、`]` 和 `}` 以及字符 `,`、`;` 和 `:` 都将用于空白检测。

以上规则需注意一点，如果运算符 `!` 或 `?` 左侧没有空白，则不管右侧是否有空白都将被看作后缀运算符。如果将 `?` 用作可选类型（`optional type`）修饰，左侧必须无空白。如果用于条件运算符 `?:`，必须两侧都有空白。

在特定构成中，以 `<` 或 `>` 开头的运算符会被分离成两个或多个标记，剩余部分以同样的方式会被再次分离。因此，在 `Dictionary<String, Array<Int>>` 中没有必要添加空白来消除闭合字符 `>` 的歧义。在这个例子中，闭合字符 `>` 被看作单字符标记，而不会被误解为移位运算符 `>>`。

要学习如何自定义新的运算符，请参考 [自定义操作符](#) 和 [运算符声明](#)。学习如何重写现有运算符，请参考 [运算符方法](#)。

运算符语法

`operator` → [operator-character](#)[operator](#)_{opt}

`operator-character` → `/` `=` `-` `+` `!` `*` `%` `<` `>` `&` `|` `^` `~` `.`

`binary-operator` → [operator](#)

`prefix-operator` → [operator](#)

`postfix-operator` → [operator](#)

3.3 类型

本篇译者：翻译：lyuka ([git 主页](#)) 校对：numbbbbb ([git 主页](#))，敬请勘误。

Swift 语言存在两种类型：命名型类型和复合型类型。命名型类型是指定义时可以给定名字的类型。命名型类型包括类、结构体、枚举和协议。比如，一个用户定义的类 `MyClass` 的实例拥有类型 `MyClass`。除了用户定义的命名型类型，Swift 标准库也定义了很多常用的命名型类型，包括那些表示数组、字典和可选值的类型。

那些通常被其它语言认为是基本或初级的数据型类型（Data types）——比如表示数字、字符和字符串——实际上就是命名型类型，Swift 标准库是使用结构体定义和实现它们的。因为它们是命名型类型，因此你可以按照[扩展](#)和[扩展声明](#)章节里讨论的那样，声明一个扩展来增加它们的行为以适应你程序的需求。

复合型类型是没有名字的类型，它由 Swift 本身定义。Swift 存在两种复合型类型：函数类型和元组类型。一个复合型类型可以包含命名型类型和其它复合型类型。例如，元组类型 `(Int, (Int, Int))` 包含两个元素：第一个是命名型类型 `Int`，第二个是另一个复合型类型 `(Int, Int)`。

本节讨论 Swift 语言本身定义的类型，并描述 Swift 中的类型推断行为。

类型的语法：

`type` → [array-type](#) | [function-type](#) | [type-identifier](#) | [tuple-type](#) | [optional-type](#) | [implicitly-unwrapped-optional-type](#) | [protocol-composition-type](#) | [metatype-type](#)

类型注解

类型注解显式地指定一个变量或表达式的值。类型注解始于冒号:终于类型，比如下面两个例子：

```
1. let someTuple: (Double, Double) = (3.14159, 2.71828)
2. func someFunction(a: Int){ /* ... */ }
```

在第一个例子中，表达式 `someTuple` 的类型被指定为 `(Double, Double)`。在第二个例子中，函数 `someFunction` 的参数 `a` 的类型被指定为 `Int`。

类型注解可以在类型之前包含一个类型特性（type attributes）的可选列表。

类型注解的语法: *type-annotation* → **:** [attributes](#)_{opt} *type*

类型标识符

类型标识符引用命名型类型或者是命名型/复合型类型的别名。

大多数情况下，类型标识符引用的是同名的命名型类型。例如类型标识符 `Int` 引用命名型类型 `Int`，同样，类型标识符 `Dictionary<String, Int>` 引用命名型类型 `Dictionary<String, Int>`。

在两种情况下类型标识符引用的不是同名的类型。情况一，类型标识符引用的是命名型/复合型类型的类型别名。比如，在下面的例子中，类型标识符使用 `Point` 来引用元组(`Int`, `Int`):

```
1. typealias Point = (Int, Int)
2. let origin: Point = (0, 0)
```

情况二，类型标识符使用 `dot(.)` 语法来表示在其它模块（`modules`）或其它类型嵌套内声明的命名型类型。例如，下面例子中的类型标识符引用在 `ExampleModule` 模块中声明的命名型类型 `MyType`:

```
1. var someValue: ExampleModule.MyType
```

类型标识符的语法:

type-identifier → [type-name](#)[generic-argument-clause](#)_{opt} [type-name](#)[generic-argument-clause](#)_{opt} **.** *type-identifier*

type-name → [identifier](#)

元组类型

元组类型使用逗号隔开并使用括号括起来的 0 个或多个类型组成的列表。

你可以使用元组类型作为一个函数的返回类型，这样就可以使函数返回多个值。你也可以命名元组类型中的元素，然后用这些名字来引用每个元素的值。元素的名字由一个标识符和:组成。“[函数和多返回值](#)”章节里有一个展示上述特性的例子。

`void` 是空元组类型`()`的别名。如果括号内只有一个元素，那么该类型就是括号内元素的类型。比如，`(Int)`的类型是 `Int` 而不是`(Int)`。所以，只有当元组类型包含两个元素以上时才可以标记元组元素。

元组类型语法:

`tuple-type` → `(tuple-type-bodyopt)`

`tuple-type-body` → `tuple-type-element-list`..._{opt}

`tuple-type-element-list` → `tuple-type-element tuple-type-element, tuple-type-element-list`

`tuple-type-element` → `attributesopt inoutopt typeinoutopt element-nametype-annotation`

`element-name` → `identifier`

函数类型

函数类型表示一个函数、方法或闭包的类型，它由一个参数类型和返回值类型组成，中间用箭头`->`隔开:

```
1. parameter type -> return type
```

由于 参数类型 和 返回值类型 可以是元组类型，所以函数类型可以让函数与方法支持多参数与多返回值。

你可以对函数类型应用带有参数类型`()`并返回表达式类型的 `auto_closure` 属性（见类型属性章节）。一个自动闭包函数捕获特定表达式上的隐式闭包而非表达式本身。下面的例子使用 `auto_closure` 属性来定义一个很简单的 `assert` 函数:

```
1. func simpleAssert(condition: @auto_closure () -> Bool, message: String) {
2.     if !condition() {
3.         println(message)
4.     }
5. }
6. let testNumber = 5
```

```
7. simpleAssert(testNumber % 2 == 0, "testNumber isn't an even number.")
8. // prints "testNumber isn't an even number."
```

函数类型可以拥有一个可变长参数作为参数类型中的最后一个参数。从语法角度上讲，可变长参数由一个基础类型名字和...组成，如 `Int...`。可变长参数被认为是一个包含了基础类型元素的数组。即 `Int...` 就是 `Int[]`。关于使用可变长参数的例子，见章节“[可变长参数](#)”。

为了指定一个 in-out 参数，可以在参数类型前加 `inout` 前缀。但是你不可以对可变长参数或返回值类型使用 `inout`。关于 In-Out 参数的讨论见章节 [In-Out 参数](#) 部分。

柯里化函数（curried function）的类型相当于一个嵌套函数类型。例如，下面的柯里化函数 `addTwoNumber()` 的类型是 `Int -> Int -> Int`：

```
1. func addTwoNumbers(a: Int) (b: Int) -> Int {
2.     return a + b
3. }
4. addTwoNumbers(4)(5) // returns 9
```

柯里化函数的函数类型从右向左组成一组。例如，函数类型 `Int -> Int -> Int` 可以被理解为 `Int -> (Int -> Int)`——也就是说，一个函数传入一个 `Int` 然后输出作为另一个函数的输入，然后又返回一个 `Int`。例如，你可以使用如下嵌套函数来重写柯里化函数

`addTwoNumbers()`：

```
1. func addTwoNumbers(a: Int) -> (Int -> Int) {
2.     func addTheSecondNumber(b: Int) -> Int {
3.         return a + b
4.     }
5.     return addTheSecondNumber
6. }
7. addTwoNumbers(4)(5) // Returns 9
```

函数类型的语法：

function-type → [type](#) -> [type](#)

数组类型

Swift 语言使用类型名紧接中括号`[]`来简化标准库中定义的命名型类型 `Array<T>`。换句话说，下面两个声明是等价的：

```
1. let someArray: String[] = ["Alex", "Brian", "Dave"]
2. let someArray: Array<String> = ["Alex", "Brian", "Dave"]
```

上面两种情况下，常量 `someArray` 都被声明为字符串数组。数组的元素也可以通过`[]`获取访问：`someArray[0]`是指第 0 个元素“Alex”。

上面的例子同时显示，你可以使用`[]`作为初始值构造数组，空的`[]`则用来构造指定类型的空数组。

```
1. var emptyArray: Double[] = []
```

你也可以使用链接起来的多个`[]`集合来构造多维数组。例如，下例使用三个`[]`集合来构造三维整型数组：

```
1. var array3D: Int[][][] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

访问一个多维数组的元素时，最左边的下标指向最外层数组的相应位置元素。接下来往右的下标指向第一层嵌入的相应位置元素，依次类推。这就意味着，在上面的例子中，`array3D[0]`是指`[[1, 2], [3, 4]]`，`array3D[0][1]`是指`[3, 4]`，`array3D[0][1][1]`则是指值 4。

关于 Swift 标准库中 `Array` 类型的细节讨论，见章节 [Arrays](#)。

数组类型的语法： `array-type` \rightarrow `type[]` `array-type[]`

可选类型

Swift 定义后缀`?`来作为标准库中的定义的命名型类型 `Optional<T>`的简写。换句话说，下面两个声明是等价的：

```
1. var optionalInteger: Int?
2. var optionalInteger: Optional<Int>
```

在上述两种情况下，变量 `optionalInteger` 都被声明为可选整型类型。注意在类型和`?`之间没有空格。

类型 `Optional<T>` 是一个枚举，有两种形式，`None` 和 `Some(T)`，又来代表可能出现或可能不出现的值。任意类型都可以被显式的声明（或隐式的转换）为可选类型。当声明一个可选类型时，确保使用括号给`?`提供合适的作用范围。比如说，声明一个整型的可选数组，应写作 `(Int[])?`，写成 `Int[]?` 的话则会出错。

如果你在声明或定义可选变量或特性的时候没有提供初始值，它的值则会自动赋成缺省值 `nil`。

可选符合 `LogicValue` 协议，因此可以出现在布尔值环境下。此时，如果一个可选类型 `T?` 实例包含有类型为 `T` 的值（也就是说值为 `Optional.Some(T)`），那么此可选类型就为 `true`，否则为 `false`。

如果一个可选类型的实例包含一个值，那么你就可以使用后缀操作符`!`来获取该值，正如下面描述的：

```
1. optionalInteger = 42
2. optionalInteger! // 42
```

使用`!`操作符获取值为 `nil` 的可选项会导致运行错误（runtime error）。

你也可以使用可选链和可选绑定来选择性的执行可选表达式上的操作。如果值为 `nil`，不会执行任何操作因此也就没有运行错误产生。

更多细节以及更多如何使用可选类型的例子，见章节[“可选”](#)。

可选类型语法： `optional-type` \rightarrow `type?`

隐式解析可选类型

Swift 语言定义后缀`!`作为标准库中命名类型 `ImplicitlyUnwrappedOptional<T>` 的简写。换句话说，下面两个声明等价：

```
1. var implicitlyUnwrappedString: String!  
2. var implicitlyUnwrappedString: ImplicitlyUnwrappedOptional<String>
```

上述两种情况下，变量 `implicitlyUnwrappedString` 被声明为一个隐式解析可选类型的字符串。注意类型与`!`之间没有空格。

你可以在使用可选的地方同样使用隐式解析可选。比如，你可以将隐式解析可选的值赋给变量、常量和可选特性，反之亦然。

有了可选，你在声明隐式解析可选变量或特性的时候就不用指定初始值，因为它有缺省值 `nil`。

由于隐式解析可选的值会在使用时自动解析，所以没必要使用操作符`!`来解析它。也就是说，如果你使用值为 `nil` 的隐式解析可选，就会导致运行错误。

使用可选链会选择性的执行隐式解析可选表达式上的某一个操作。如果值为 `nil`，就不会执行任何操作，因此也不会产生运行错误。

关于隐式解析可选的更多细节，见章节[“隐式解析可选”](#)。

隐式解析可选的语法： *implicitly-unwrapped-optional-type* → [type](#)!

协议合成类型

协议合成类型是一种符合每个协议的指定协议列表类型。协议合成类型可能会用在类型注解和泛型参数中。

协议合成类型的形式如下：

```
1. protocol<Protocol 1, Protocol 2>
```

协议合成类型允许你指定一个值，其类型可以适配多个协议的条件，而且不需要定义一个新的命名型协议来继承其它想要适配的各个协议。比如，协议合成类型 `protocol<Protocol A, Protocol B, Protocol C>`等效于一个从 `Protocol A`，`Protocol B`，`Protocol C` 继承而来的新协议 `Protocol D`，很显然这样做有效率的多，甚至不需引入一个新名字。

协议合成列表中的每项必须是协议名或协议合成类型的类型别名。如果列表为空，它就会指定一个空协议合成列表，这样每个类型都能适配。

协议合成类型的语法：

protocol-composition-type → protocol<**protocol-identifier-list**_{opt}>

protocol-identifier-list → **protocol-identifier** **protocol-identifier**, **protocol-identifier-list**

protocol-identifier → **type-identifier**

元类型

元类型是指所有类型的类型，包括类、结构体、枚举和协议。

类、结构体或枚举类型的元类型是相应的类型名紧跟 **Type**。协议类型的元类型——并不是运行时适配该协议的具体类型——是该协议名字紧跟 **Protocol**。比如，类 **SomeClass** 的元类型就是 **SomeClass.Type**，协议 **SomeProtocol** 的元类型就是 **SomeProtocol.Protocol**。

你可以使用后缀 **self** 表达式来获取类型。比如，**SomeClass.self** 返回 **SomeClass** 本身，而不是 **SomeClass** 的一个实例。同样，**SomeProtocol.self** 返回 **SomeProtocol** 本身，而不是运行时适配 **SomeProtocol** 的某个类型的实例。还可以对类型的实例使用 **dynamicType** 表达式来获取该实例在运行阶段的类型，如下所示：

```
1. class SomeBaseClass {
2.     class func printClassName() {
3.         println("SomeBaseClass")
4.     }
5. }
6. class SomeSubClass: SomeBaseClass {
7.     override class func printClassName() {
8.         println("SomeSubClass")
9.     }
10. }
11. let someInstance: SomeBaseClass = SomeSubClass()
12. // someInstance is of type SomeBaseClass at compile time, but
13. // someInstance is of type SomeSubClass at runtime
```

```
14. someInstance.dynamicType.printClassName()
15. // prints "SomeSubClass"
```

元类型的语法: *metatype-type* → *type.Type* *type.Protocol*

类型继承子句

类型继承子句被用来指定一个命名型类型继承哪个类且适配哪些协议。类型继承子句开始于冒号:，紧跟由,隔开的类型标识符列表。

类可以继承单个超类，适配任意数量的协议。当定义一个类时，超类的名字必须出现在类型标识符列表首位，然后跟上该类需要适配的任意数量的协议。如果一个类不是从其它类继承而来，那么列表可以以协议开头。关于类继承更多的讨论和例子，见章节“[继承](#)”。

其它命名型类型可能只继承或适配一个协议列表。协议类型可能继承于其它任意数量的协议。当一个协议类型继承于其它协议时，其它协议的条件集合会被集成在一起，然后其它从当前协议继承的任意类型必须适配所有这些条件。

枚举定义中的类型继承子句可以是一个协议列表，或是指定原始值的枚举，一个单独的指定原始值类型的命名型类型。使用类型继承子句来指定原始值类型的枚举定义的例子，见章节“[原始值](#)”。

类型继承子句的语法:

type-inheritance-clause → *: type-identifier*

type-identifier → *type-identifier* *type-identifier* , *type-identifier*

类型推断

Swift 广泛的使用类型推断，从而允许你可以忽略很多变量和表达式的类型或部分类型。比如，对于 `var x: Int = 0`，你可以完全忽略类型而简写成 `var x = 0`——编译器会正确的推断出 `x` 的类型 `Int`。类似的，当完整的类型可以从上下文推断出来时，你也可以忽略类型的一

部分。比如，如果你写了 `let dict: Dictionary = ["A": 1]`，编译器也能推断出 `dict` 的类型是 `Dictionary<String, Int>`。

在上面的两个例子中，类型信息从表达式树（**expression tree**）的叶子节点传向根节点。也就是说，`var x: Int = 0` 中 `x` 的类型首先根据 `0` 的类型进行推断，然后将该类型信息传递到根节点（变量 `x`）。

在 **Swift** 中，类型信息也可以反方向流动——从根节点传向叶子节点。在下面的例子中，常量 `eFloat` 上的显式类型注解（`:Float`）导致数字字面量 `2.71828` 的类型是 `Float` 而非 `Double`。

```
1. let e = 2.71828 // The type of e is inferred to be Double.
2. let eFloat: Float = 2.71828 // The type of eFloat is Float.
```

Swift 中的类型推断在单独的表达式或语句水平上进行。这意味着所有用于推断类型的信息必须可以从表达式或其某个子表达式的类型检查中获取。

3.4 表达式

本篇译者：sg552（[Github 主页](#)），敬请勘误。

Swift 中存在四种表达式：前缀（**prefix**）表达式，二元（**binary**）表达式，主要（**primary**）表达式和后缀（**postfix**）表达式。表达式可以返回一个值，以及运行某些逻辑（**causes a side effect**）。

前缀表达式和二元表达式就是对某些表达式使用各种运算符（**operators**）。主要表达式是最短小的表达式，它提供了获取（变量的）值的一种途径。后缀表达式则允许你建立复杂的表达式，例如配合函数调用和成员访问。每种表达式都在下面有详细论述～

表达式的语法：

expression → [prefix-expression](#)[binary-expressions](#)_{opt}

expression-list → [expression](#) [expression](#), [expression-list](#)

前缀表达式（Prefix Expressions）

前缀表达式由 前缀符号和表达式组成。（这个前缀符号只能接收一个参数）

Swift 标准库支持如下的前缀操作符：

- ++ 自增 1（increment）
- 自减 1（decrement）
- ! 逻辑否（Logical NOT）
- ~ 按位否（Bitwise NOT）
- + 加（Unary plus）
- 减（Unary minus）

对于这些操作符的使用，请参见：[Basic Operators](#) 和 [Advanced Operators](#)

作为对上面标准库运算符的补充，你也可以对 某个函数的参数使用 '&'运算符。更多信息，请参见：["In-Out parameters"](#)。

前缀表达式的语法：

prefix-expression → [prefix-operator](#)_{opt}[postfix-expression](#)

prefix-expression → [in-out-expression](#)

in-out-expression → [&identifier](#)

二元表达式（Binary Expressions）

二元表达式由 "左边参数" + "二元运算符" + "右边参数" 组成，它有如下的形式：

`left-hand argument` `operator` `right-hand argument`

Swift 标准库提供了如下的二元运算符：

求幂相关（无结合，优先级 160）

- << 按位左移（Bitwise left shift）
- >> 按位右移（Bitwise right shift）

乘除法相关（左结合，优先级 150）

* 乘

/ 除

% 求余

&* 乘法，忽略溢出（Multiply, ignoring overflow）

&/ 除法，忽略溢出（Divide, ignoring overflow）

&% 求余，忽略溢出（Remainder, ignoring overflow）

& 位与（Bitwise AND）

加减法相关（左结合，优先级 140）

+ 加

- 减

&+ Add with overflow

&- Subtract with overflow

| 按位或（Bitwise OR）

^ 按位异或（Bitwise XOR）

Range（无结合，优先级 135）

.. 半闭值域 Half-closed range

... 全闭值域 Closed range

类型转换（无结合，优先级 132）

is 类型检查（type check）

as 类型转换（type cast）

Comparative（无结合，优先级 130）

< 小于

<= 小于等于

> 大于

>= 大于等于

== 等于

!= 不等

=== 恒等于

!== 不恒等

~= 模式匹配 (Pattern match)

合取 (**Conjunctive**) (左结合, 优先级 120)

&& 逻辑与 (Logical AND)

析取 (**Disjunctive**) (左结合, 优先级 110)

|| 逻辑或 (Logical OR)

三元条件 (**Ternary Conditional**) (右结合, 优先级 100)

?: 三元条件 Ternary conditional

赋值 (**Assignment**) (右结合, 优先级 90)

= 赋值 (Assign)

*= Multiply and assign

/= Divide and assign

%= Remainder and assign

+= Add and assign

-= Subtract and assign

<<= Left bit shift and assign

=>= Right bit shift and assign

&= Bitwise AND and assign

^= Bitwise XOR and assign

|= Bitwise OR and assign

&&= Logical AND and assign

||= Logical OR and assign

关于这些运算符 (operators) 的更多信息, 请参见: [Basic Operators](#) 和 [Advanced Operators](#).

注意: 在解析时, 一个二元表达式表示为一个一级数组 (a flat list), 这个数组 (List) 根据运算符的先后顺序, 被转换成了一个 tree. 例如: `2+3*5` 首先被认为是: `2, +, 3, *, 5`. 随后它被转换成 tree (`2 + (3 * 5)`)

二元表达式的语法

binary-expression → [binary-operatorprefix-expression](#)

binary-expression → [assignment-operatorprefix-expression](#)

binary-expression → [conditional-operatorprefix-expression](#)

binary-expression → [type-casting-operator](#)

binary-expressions → [binary-expressionbinary-expressions_{opt}](#)

赋值表达式（Assignment Operator）

赋值表达式会对某个给定的表达式赋值。它有如下的形式：

`expression = value`

就是把右边的 `value` 赋值给左边的 `expression`。如果左边的 `expression` 需要接收多个参数（是一个 `tuple`），那么右边必须也是一个具有同样数量参数的 `tuple`。（允许嵌套的 `tuple`）

```
1. (a, _, (b, c)) = ("test", 9.45, (12, 3))
2. // a is "test", b is 12, c is 3, and 9.45 is ignored
```

赋值运算符不返回任何值。

赋值表达式的语法：assignment-operator → =

三元条件运算符（Ternary Conditional Operator）

三元条件运算符 是根据条件来获取值。形式如下：

`condition ? expression used if true : expression used if false`

如果 `condition` 是 `true`，那么返回 第一个表达式的值（此时不会调用第二个表达式），否则返回第二个表达式的值（此时不会调用第一个表达式）。

想看三元条件运算符的例子，请参见：[Ternary Conditional Operator](#)。

三元条件表达式: *conditional-operator* → ?[expression](#)

类型转换运算符（Type-Casting Operators）

有两种类型转换操作符: `as` 和 `is`. 它们有如下的形式:

`expression as type`

`expression as? type`

`expression is type`

`as` 运算符会把目标表达式转换成指定的类型（*specified type*），过程如下:

1. 如果类型转换成功，那么目标表达式就会返回指定类型的实例（*instance*）。例如：把子类（*subclass*）变成父类（*superclass*）时。
2. 如果转换失败，则会抛出编译错误（*compile-time error*）。
3. 如果上述两个情况都不是（也就是说，编译器在编译时期无法确定转换能否成功，）那么目标表达式就会变成指定的类型的 *optional*.（*is an optional of the specified type*）然后在运行时，如果转换成功，目标表达式就会作为 *optional* 的一部分来返回，否则，目标表达式返回 `nil`. 对应的例子是：把一个 *superclass* 转换成一个 *subclass*.

```
1. class SomeSuperType {}
2. class SomeType: SomeSuperType {}
3. class SomeChildType: SomeType {}
4. let s = SomeType()
5.
6. let x = s as SomeSuperType // known to succeed; type is SomeSuperType
7. let y = s as Int           // known to fail; compile-time error
8. let z = s as SomeChildType // might fail at runtime; type is SomeChildType?
```

使用 `'as'` 做类型转换跟正常的类型声明，对于编译器来说是一样的。例如:

```
1. let y1 = x as SomeType // Type information from 'as'
```

```
2. let y2: SomeType = x    // Type information from an annotation
```

'is' 运算符在“运行时（runtime）”会做检查。 成功会返回 `true`, 否则 `false`

上述检查在“编译时（compile time）”不能使用。 例如下面的使用是错误的：

```
1. "hello" is String
2. "hello" is Int
```

关于类型转换的更多内容和例子，请参见：[Type Casting](#).

类型转换的语法：*type-casting-operator* → `is`[type](#) *as?*_{opt}[type](#)

主要表达式（Primary Expressions）

主要表达式是最基本的表达式。 它们可以跟 前缀表达式，二元表达式，后缀表达式以及其他主要表达式组合使用。

主要表达式的语法

primary-expression → [identifiergeneric-argument-clause](#)_{opt}

primary-expression → [literal-expression](#)

primary-expression → [self-expression](#)

primary-expression → [superclass-expression](#)

primary-expression → [closure-expression](#)

primary-expression → [parenthesized-expression](#)

primary-expression → [implicit-member-expression](#)

primary-expression → [wildcard-expression](#)

字符型表达式（Literal Expression）

由这些内容组成：普通的字符（`string, number`），一个字符的字典或者数组，或者下面列表中的特殊字符。

字符 (Literal)	类型 (Type)	值 (Value)
<code>__FILE__</code>	String	所在的文件名
<code>__LINE__</code>	Int	所在的行数
<code>__COLUMN__</code>	Int	所在的列数
<code>__FUNCTION__</code>	String	所在的function 的名字

在某个函数(function)中, `__FUNCTION__` 会返回当前函数的名字。在某个方法(method)中, 它会返回当前方法的名字。在某个 property 的 getter/setter 中会返回这个属性的名字。在 init/subscript 中 只有的特殊成员 (member) 中会返回这个 keyword 的名字, 在某个文件的顶端 (the top level of a file), 它返回的是当前 module 的名字。

一个 array literal, 是一个有序的值的集合。 它的形式是:

```
[ value 1, value 2, ... ]
```

数组中的最后一个表达式可以紧跟一个逗号 (','), [] 表示空数组。 array literal 的 type 是 T[], 这个 T 就是数组中元素的 type. 如果该数组中有多种 type, T 则是跟这些 type 的公共 supertype 最接近的 type. (closest common supertype)

一个 dictionary literal 是一个包含无序的键值对 (key-value pairs) 的集合, 它的形式是:

```
[ key 1 : value 1, key 2 : value 2, ... ]
```

dictionary 的最后一个表达式可以是一个逗号 (','), [:] 表示一个空的 dictionary. 它的 type 是 Dictionary (这里 KeyType 表示 key 的 type, ValueType 表示 value 的 type) 如果这个 dictionary 中包含多种 types, 那么 KeyType, Value 则对应着它们的公共 supertype 最接近的 type (closest common supertype)。

字符型表达式的语法:

literal-expression → [literal](#)

literal-expression → [array-literal](#) [dictionary-literal](#)

literal-expression → `__FILE__` `__LINE__` `__COLUMN__` `__FUNCTION__`

array-literal → [[array-literal-items](#)_{opt}]

array-literal-items → [array-literal-item](#), _{opt} [array-literal-item](#), [array-literal-items](#)

array-literal-item → [expression](#)

`dictionary-literal` → [\[dictionary-literal-items\]](#) `[:]`
`dictionary-literal-items` → [dictionary-literal-item](#), *opt* [dictionary-literal-item](#), [dictionary-literal-items](#)
`dictionary-literal-item` → [expression](#):[expression](#)

self 表达式 (Self Expression)

`self` 表达式是对 当前 `type` 或者当前 `instance` 的引用。它的形式如下：

```
self
self. member name
self[ subscript index ]
self( initializer arguments )
self.init( initializer arguments )
```

如果在 `initializer`, `subscript`, `instance method` 中, `self` 等同于当前 `type` 的 `instance`. 在一个静态方法 (`static method`), 类方法 (`class method`) 中, `self` 等同于当前的 `type`.

当访问 `member` (成员变量时), `self` 用来区分重名变量 (例如函数的参数). 例如, (下面的 `self.greeting` 指的是 `var greeting: String`, 而不是 `init(greeting: String)`)

```
1. class SomeClass {
2.     var greeting: String
3.     init(greeting: String) {
4.         self.greeting = greeting
5.     }
6. }
```

在 `mutating` 方法中, 你可以使用 `self` 对 该 `instance` 进行赋值。

```
1. struct Point {
2.     var x = 0.0, y = 0.0
3.     mutating func moveByX(deltaX: Double, y deltaY: Double) {
4.         self = Point(x: x + deltaX, y: y + deltaY)
5.     }
6. }
```

self 表达式的语法:

self-expression → `self`

self-expression → `self.identifier`

self-expression → `self[expression]`

self-expression → `self.init`

超类表达式 (Superclass Expression)

超类表达式可以使我们在某个 `class` 中访问它的超类. 它有如下形式:

`super.member name`

`super[subscript index]`

`super.init(initializer arguments)`

形式 1 用来访问超类的某个成员 (`member`). 形式 2 用来访问该超类的 `subscript` 实现。

形式 3 用来访问该超类的 `initializer`.

子类 (`subclass`) 可以通过超类 (`superclass`) 表达式在它们的 `member`, `subscripting` 和 `initializers` 中来利用它们超类中的某些实现 (既有的方法或者逻辑)。

超类表达式语法:

superclass-expression → [superclass-method-expression](#) [superclass-subscript-expression](#) -
[superclass-initializer-expression](#)

superclass-method-expression → `super.identifier`

superclass-subscript-expression → `super[expression]`

superclass-initializer-expression → `super.init`

闭包表达式 (Closure Expression)

闭包 (`closure`) 表达式可以建立一个闭包 (在其他语言中也叫 `lambda`, 或者 匿名函数 (`anonymous function`)). 跟函数 (`function`) 的声明一样, 闭包 (`closure`) 包含了可执行的代码 (跟方法主体 (`statement`) 类似) 以及接收 (`capture`) 的参数. 它的形式如下:

```
{ (parameters) -> return type in  
    statements  
}
```

闭包的参数声明形式跟方法中的声明一样，请参见：[Function Declaration](#)。

闭包还有几种特殊的形式，让使用更加简洁：

1. 闭包可以省略 它的参数的 **type** 和返回值的 **type**。如果省略了参数和参数类型，就也要省略 **'in'**关键字。 如果被省略的 **type** 无法被编译器获知（**inferred**），那么就会抛出编译错误。
2. 闭包可以省略参数，转而在方法体（**statement**）中使用 **0,1,\$2** 来引用出现的第一个，第二个，第三个参数。
3. 如果闭包中只包含了一个表达式，那么该表达式就会自动成为该闭包的返回值。 在执行 **'type inference'**时，该表达式也会返回。

下面几个 闭包表达式是 等价的：

```
1. myFunction {  
2.     (x: Int, y: Int) -> Int in  
3.     return x + y  
4. }  
5.  
6. myFunction {  
7.     (x, y) in  
8.     return x + y  
9. }  
10.  
11. myFunction { return $0 + $1 }  
12.  
13. myFunction { $0 + $1 }
```

关于 向闭包中传递参数的内容，参见：[Function Call Expression](#)。

闭包表达式可以通过一个参数列表（**capture list**）来显式指定它需要的参数。参数列表 由中括号 **[]** 括起来，里面的参数由逗号','分隔。一旦使用了参数列表，就必须使用**'in'**关键字（在任何情况下都得这样做，包括忽略参数的名字，**type**，返回值时等等）。

在闭包的参数列表（capture list）中，参数可以声明为 'weak' 或者 'unowned'.

```
1. myFunction { print(self.title) } // strong capture
2. myFunction { [weak self] in print(self!.title) } // weak capture
3. myFunction { [unowned self] in print(self.title) } // unowned capture
```

在参数列表中，也可以使用任意表达式来赋值. 该表达式会在闭包被执行时赋值，然后按照不同的力度来获取（这句话请慎重理解），例如：

```
1. // Weak capture of "self.parent" as "parent"
2. myFunction { [weak parent = self.parent] in print(parent!.title) }
```

关于闭包表达式的更多信息和例子，请参见：[Closure Expressions](#).

闭包表达式的语法：

```
closure-expression → {closure-signatureopt statements}
closure-signature → parameter-clauseopt function-resultopt in
closure-signature → identifier-listopt function-resultopt in
closure-signature → capture-listopt parameter-clauseopt function-resultopt in
closure-signature → capture-listopt identifier-listopt function-resultopt in
closure-signature → capture-listopt in
capture-list → [capture-specifieropt expression]
capture-specifier → weak | unowned | unowned(safe) | unowned(unsafe)
```

隐式成员表达式（Implicit Member Expression）

在可以判断出类型(type)的上下文(context)中, 隐式成员表达式是访问某个type的member（例如 class method, enumeration case）的简洁方法。它的形式是：

. member name

例子：

```
1. var x = MyEnumeration.SomeValue
```

```
2. x = .AnotherValue
```

隐式成员表达式的语法: *implicit-member-expression* → [.\[identifier\]\(#\)](#)

圆括号表达式（Parenthesized Expression）

圆括号表达式由多个子表达式和逗号','组成。每个子表达式前面可以有 `identifier x`: 这样的可选前缀。形式如下:

```
( identifier 1 : expression 1 , identifier 2 : expression 2 , ... )
```

圆括号表达式用来建立 `tuples` , 然后把它做为参数传递给 `function`. 如果某个圆括号表达式中只有一个子表达式, 那么它的 `type` 就是子表达式的 `type`。例如: `(1)` 的 `type` 是 `Int`, 而不是 `(Int)`

圆括号表达式的语法:

parenthesized-expression → [\(expression-element-list_{opt}\)](#)

expression-element-list → [expression-element](#) | [expression-element](#) , [expression-element-list](#)

expression-element → [expression](#) | [identifier](#) : [expression](#)

通配符表达式（Wildcard Expression）

通配符表达式用来忽略传递进来的某个参数。例如: 下面的代码中, `10` 被传递给 `x`, `20` 被忽略 (译注: 好奇葩的语法。。。)

```
1. (x, _) = (10, 20)
2. // x is 10, 20 is ignored
```

通配符表达式的语法: *wildcard-expression* → `_`

后缀表达式（Postfix Expressions）

后缀表达式就是在某个表达式的后面加上 操作符。 严格的讲, 每个主要表达式 (`primary`

expression) 都是一个后缀表达式。

Swift 标准库提供了下列后缀表达式：

++ Increment

-- Decrement

对于这些操作符的使用，请参见：[Basic Operators](#) 和 [Advanced Operators](#)

后缀表达式的语法：

postfix-expression → [primary-expression](#)
postfix-expression → [postfix-expressionpostfix-operator](#)
postfix-expression → [function-call-expression](#)
postfix-expression → [initializer-expression](#)
postfix-expression → [explicit-member-expression](#)
postfix-expression → [postfix-self-expression](#)
postfix-expression → [dynamic-type-expression](#)
postfix-expression → [subscript-expression](#)
postfix-expression → [forced-value-expression](#)
postfix-expression → [optional-chaining-expression](#)

函数调用表达式（Function Call Expression）

函数调用表达式由函数名和参数列表组成。它的形式如下：

```
function name ( argument value 1 , argument value 2 )
```

如果该 `function` 的声明中指定了参数的名字，那么在调用的时候也必须得写出来。例如：

```
function name ( argument name 1 : argument value 1 , argument name 2 : argument value 2 )
```

可以在 函数调用表达式的尾部（最后一个参数之后）加上 一个闭包（closure），该闭包会被目标函数理解并执行。它具有如下两种写法：

```
1. // someFunction takes an integer and a closure as its arguments
2. someFunction(x, {$0 == 13})
3. someFunction(x) {$0 == 13}
```

如果闭包是该函数的唯一参数，那么圆括号可以省略。

```
1. // someFunction takes a closure as its only argument
2. myData.someMethod() {$0 == 13}
3. myData.someMethod {$0 == 13}
```

函数调用表达式语法：

function-call-expression → [postfix-expressionparenthesized-expression](#)

function-call-expression → [postfix-expressionparenthesized-expression](#)_{opt}[trailing-closure](#)

trailing-closure → [closure-expression](#)

初始化函数表达式（Initializer Expression）

Initializer 表达式用来给某个 Type 初始化。它的形式如下：

```
expression .init( initializer arguments )
```

（Initializer 表达式用来给某个 Type 初始化。）跟函数（function）不同，initializer 不能返回值。

```
1. var x = SomeClass.someClassFunction // ok
2. var y = SomeClass.init                // error
```

你也可以使用初始化函数表达式来委托给父类的 initializer

```
1. class SomeSubClass: SomeSuperClass {
2.     init() {
3.         // subclass initialization goes here
4.         super.init()
5.     }
6. }
```

initializer 表达式的语法： *initializer-expression* → [postfix-expression](#).init

显式成员表达式（Explicit Member Expression）

显示成员表达式允许我们访问 `type`, `tuple`, `module` 的成员变量。它的形式如下：

`expression . member name`

该 `member` 就是某个 `type` 在声明时候所定义（`declaration or extension`）的变量，例如：

```
1. class SomeClass {
2.     var someProperty = 42
3. }
4. let c = SomeClass()
5. let y = c.someProperty // Member access
```

对于 `tuple`，要根据它们出现的顺序（0, 1, 2...）来使用：

```
1. var t = (10, 20, 30)
2. t.0 = t.1
3. // Now t is (20, 20, 30)
```

显示成员表达式的语法：

explicit-member-expression → [postfix-expression](#) . [decimal-digit](#)

explicit-member-expression → [postfix-expression](#) . [identifiergeneric-argument-clause](#)_{opt}

后缀 self 表达式（Postfix Self Expression）

后缀表达式由 某个表达式 + `'.self'` 组成。形式如下：

`expression . self`

`type . self`

形式 1 表示会返回 `expression` 的值。例如：`x.self` 返回 `x`

形式 2 返回对应的 `type`。我们可以用它来动态的获取某个 `instance` 的 `type`。

后缀 `self` 表达式的语法：*postfix-self-expression* → [postfix-expression](#) . `self`

Dynamic Type 表达式 (Dynamic Type Expression)

(因为 `dynamicType` 是一个独有的方法, 所以这里保留了英文单词, 未作翻译, --- 类似与 `self expression`)

Dynamic Type 表达式由 某个表达式 + `'.dynamicType'` 组成。

`expression.dynamicType`

上面的形式中, `expression` 不能是某 `type` 的名字 (当然了, 如果我都知道它的名字了还需要动态来获取它吗)。动态类型表达式会返回"运行时"某个 `instance` 的 `type`, 具体请看下面的例子:

```
1. class SomeBaseClass {
2.     class func printClassName() {
3.         println("SomeBaseClass")
4.     }
5. }
6. class SomeSubClass: SomeBaseClass {
7.     override class func printClassName() {
8.         println("SomeSubClass")
9.     }
10. }
11. let someInstance: SomeBaseClass = SomeSubClass()
12. // someInstance is of type SomeBaseClass at compile
    time, but
13. // someInstance is of type SomeSubClass at runtime
14. someInstance.dynamicType.printClassName()
15. // prints "SomeSubClass"
```

Dynamic Type 表达式语法: *dynamic-type-expression* → [postfix-expression](#).`dynamicType`

附属脚本表达式 (Subscript Expression)

附属脚本表达式提供了通过附属脚本访问 `getter/setter` 的方法。它的形式是:

`expression [index expressions]`

可以通过附属脚本表达式通过 `getter` 获取某个值, 或者通过 `setter` 赋予某个值。

关于 subscript 的声明，请参见：[Protocol Subscript Declaration](#).

附属脚本表达式的语法：*subscript-expression* → [postfix-expression](#) [[expression-list](#)]

强制取值表达式（Forced-Value Expression）

强制取值表达式用来获取某个目标表达式的值（该目标表达式的值必须不是 `nil`）。它的形式如下：

`expression !`

如果该表达式的值不是 `nil`，则返回对应的值。否则，抛出运行时错误（runtime error）。

强制取值表达式的语法：*forced-value-expression* → [postfix-expression](#) !

可选链表达式（Optional-Chaining Expression）

可选链表达式由目标表达式 + '?' 组成，形式如下：

`expression ?`

后缀 '?' 返回目标表达式的值，把它做为可选的参数传递给后续的表达式

如果某个后缀表达式包含了可选链表达式，那么它的执行过程就比较特殊：首先先判断该可选链表达式的值，如果是 `nil`，整个后缀表达式都返回 `nil`，如果该可选链的值不是 `nil`，则正常返回该后缀表达式的值（依次执行它的各个子表达式）。在这两种情况下，该后缀表达式仍然是一个 optional type（In either case, the value of the postfix expression is still of an optional type）

如果某个"后缀表达式"的"子表达式"中包含了"可选链表达式"，那么只有最外层的表达式返回的才是一个 optional type. 例如，在下面的例子中，如果 `c` 不是 `nil`，那么 `c?.property.performAction()` 这句代码在执行时，就会先获得 `c` 的 `property` 方法，然后调用 `performAction()` 方法。然后对于 `"c?.property.performAction()"` 这个整体，它的

返回值是一个 optional type.

```
1. var c: SomeClass?
2. var result: Bool? = c?.property.performAction()
```

如果不使用可选链表达式, 那么 上面例子的代码跟下面例子等价:

```
1. if let unwrappedC = c {
2.     result = unwrappedC.property.performAction()
3. }
```

可选链表达式的语法: *optional-chaining-expression* → [postfix-expression](#)

3.5 语句

本篇译者: coverxit ([Github 主页](#)), 敬请勘误。

在 **Swift** 中, 有两种类型的语句: 简单语句和控制流语句。简单语句是最常见的, 用于构造表达式和声明。控制流语句则用于控制程序执行的流程, **Swift** 中有三种类型的控制流语句: 循环语句、分支语句和控制传递语句。

循环语句用于重复执行代码块; 分支语句用于执行满足特定条件的代码块; 控制传递语句则用于修改代码的执行顺序。在稍后的叙述中, 将会详细地介绍每一种类型的控制流语句。

是否将分号 (;) 添加到语句的结尾处是可选的。但若要在同一行内写多条独立语句, 请务必使用分号。

GRAMMAR OF A STATEMENT

statement → expression; opt

statement → declaration; opt

statement → loop-statement; opt

statement → branch-statement; opt

statement → labeled-statement

statement → control-transfer-statement; opt

statement → statment statements; opt

循环语句

取决于特定的循环条件，循环语句允许重复执行代码块。**Swift** 提供四种类型的循环语句：**for** 语句、**for-in** 语句、**while** 语句和 **do-while** 语句。

通过 **break** 语句和 **continue** 语句可以改变循环语句的控制流。有关这两条语句，详情参见 **Break** 语句和 **Continue** 语句。

GRAMMAR OF A LOOP STATEMENT

loop-statement → for-statement

loop-statement → for-in-statement

loop-statement → while-statement

loop-statement → do-while-statement

For 语句

for 语句允许在重复执行代码块的同时，递增一个计数器。

for 语句的形式如下：

```
1. for `initialization`; `condition`; `increment` {
```

```
2.     `statements`  
3. }
```

`initialization`、`condition` 和 `increment` 之间的分号，以及包围循环体 `statements` 的大括号都是不可省略的。

`for` 语句的执行流程如下：

- 1、`initialization` 只会被执行一次，通常用于声明和初始化在接下来的循环中需要使用的变量。
- 2、计算 `condition` 表达式： 如果为 `true`，`statements` 将会被执行，然后转到第 3 步。如果为 `false`，`statements` 和 `increment` 都不会被执行，`for` 至此执行完毕。
- 3、计算 `increment` 表达式，然后转到第 2 步。

定义在 `initialization` 中的变量仅在 `for` 语句的作用域以内有效。`condition` 表达式的值的类型必须遵循 `LogicValue` 协议。

GRAMMAR OF A FOR STATEMENT

`for-statement` → `for` `for-init opt` ; `expression opt` ; `expression opt code-block`

`for-statement` → `for` (`for-init opt` ; `expression opt` ; `expression opt`) `code-block`

`for-statement` → `variable-declaration` | `expression-list`

For-In 语句

`for-in` 语句允许在重复执行代码块的同时，迭代集合（或遵循 `Sequence` 协议的任意类型）中的每一项。

`for-in` 语句的形式如下：

```
1. for `item` in `collection` {  
2.     `statements`  
3. }
```

`for-in` 语句在循环开始前会调用 `collection` 表达式的 `generate` 方法来获取一个生成器类型（这是一个遵循 `Generator` 协议的类型）的值。接下来循环开始，调用 `collection` 表达式的 `next` 方法。如果其返回值不是 `None`，它将会被赋给 `item`，然后执行 `statements`，执行完毕后回到循环开始处；否则，将不会赋值给 `item` 也不会执行 `statements`，`for-in` 至此执行完毕。

GRAMMAR OF A FOR-IN STATEMENT

`for-in-statement` → `for pattern in expression code-block`

While 语句

`while` 语句允许重复执行代码块。

`while` 语句的形式如下：

```
1. while `condition` {  
2.     `statements`  
3. }
```

`while` 语句的执行流程如下：

- 1、计算 `condition` 表达式： 如果为真 `true`，转到第 2 步。如果为 `false`，`while` 至此执行完毕。
- 2、执行 `statements`，然后转到第 1 步。

由于 `condition` 的值在 `statements` 执行前就已计算出，因此 `while` 语句中的 `statements` 可能会被执行若干次，也可能不会被执行。

`condition` 表达式的值的类型必须遵循 `LogicValue` 协议。同时，`condition` 表达式也可以使用可选绑定，详情参见可选绑定。

GRAMMAR OF A WHILE STATEMENT

`while-statement` → `while while-condition code-block`

`while-condition` → `expression` | `declaration`

Do-While 语句

`do-while` 语句允许代码块被执行一次或多次。

`do-while` 语句的形式如下：

```
1. do {  
2.     `statements`  
3. } while `condition`
```

`do-while` 语句的执行流程如下：

- 1、执行 `statements`，然后转到第 2 步。
- 2、计算 `condition` 表达式： 如果为 `true`，转到第 1 步。如果为 `false`，`do-while` 至此执行完毕。

由于 `condition` 表达式的值是在 `statements` 执行后才计算出，因此 `do-while` 语句中的 `statements` 至少会被执行一次。

`condition` 表达式的值的类型必须遵循 `LogicValue` 协议。同时，`condition` 表达式也可以使用可选绑定，详情参见可选绑定。

GRAMMAR OF A DO-WHILE STATEMENT

`do-while-statement` → `do code-block while while-condition`

分支语句

取决于一个或者多个条件的值，分支语句允许程序执行指定部分的代码。显然，分支语句中条件的值将会决定如何分支以及执行哪一块代码。**Swift** 提供两种类型的分支语句：`if` 语句

和 `switch` 语句。

`switch` 语句中的控制流可以用 `break` 语句修改，详情请见 `Break` 语句。

GRAMMAR OF A BRANCH STATEMENT

branch-statement \rightarrow if-statement

branch-statement \rightarrow switch-statement

If 语句

取决于一个或多个条件的值，`if` 语句将决定执行哪一块代码。

`if` 语句有两种标准形式，在这两种形式里都必须有大括号。

第一种形式是当且仅当条件为真时执行代码，像下面这样：

```
1. if `condition` {  
2.     `statements`  
3. }
```

第二种形式是在第一种形式的基础上添加 `else` 语句，当只有一个 `else` 语句时，像下面这样：

```
1. if `condition` {  
2.     `statements to execute if condition is true`  
3. } else {  
4.     `statements to execute if condition is false`  
5. }
```

同时，`else` 语句也可包含 `if` 语句，从而形成一条链来测试更多的条件，像下面这样：

```
1. if `condition 1` {  
2.     `statements to execute if condition 1 is true`  
3. } else if `condition 2` {
```

```
4.     `statements to execute if condition 2 is true`
5. }
6. else {
7.     `statements to execute if both conditions are false`
8. }
```

if 语句中条件的值的类型必须遵循 `LogicValue` 协议。同时，条件也可以使用可选绑定，详情参见可选绑定。

GRAMMAR OF AN IF STATEMENT

if-statement → if if-condition code-block else-clause opt

if-condition → expression | declaration

else-clause → else code-block | else if-statement opt

Switch 语句

取决于 `switch` 语句的控制表达式（`control expression`），`switch` 语句将决定执行哪一块代码。

`switch` 语句的形式如下：

```
1. switch `control expression` {
2.     case `pattern 1`:
3.         `statements`
4.     case `pattern 2` where `condition`:
5.         `statements`
6.     case `pattern 3` where `condition`,
7.         `pattern 4` where `condition`:
8.         `statements`
9.     default:
10.         `statements`
11. }
```

`switch` 语句的控制表达式（`control expression`）会首先被计算，然后与每一个 `case` 的模

式(pattern)进行匹配。如果匹配成功,程序将会执行对应的 **case** 分支里的 **statements**。另外,每一个 **case** 分支都不能为空,也就是说在每一个 **case** 分支中至少有一条语句。如果你不想在匹配到的 **case** 分支中执行代码,只需在该分支里写一条 **break** 语句即可。

可以用作控制表达式的值是十分灵活的,除了标量类型(scalar types, 如 **Int**、**Character**)外,你可以使用任何类型的值,包括浮点数、字符串、元组、自定义类的实例和可选(optional)类型,甚至是枚举类型中的成员值和指定的范围(range)等。关于在 **switch** 语句中使用这些类型,详情参见控制流一章的 **Switch**。

你可以在模式后面添加一个起保护作用的表达式(guard expression)。起保护作用的表达式是这样构成的:关键字 **where** 后面跟着一个作为额外测试条件的表达式。因此,当且仅当控制表达式匹配一个 **case** 的某个模式且起保护作用的表达式为真时,对应 **case** 分支中的 **statements** 才会被执行。在下面的例子中,控制表达式只会匹配含两个相等元素的元组,如(1, 1):

```
1. case let (x, y) where x == y:
```

正如上面这个例子,也可以在模式中使用 **let** (或 **var**) 语句来绑定常量 (或变量)。这些常量 (或变量) 可以在其对应的起保护作用的表达式和其对应的 **case** 块里的代码中引用。但是,如果 **case** 中有多个模式匹配控制表达式,那么这些模式都不能绑定常量 (或变量)。

switch 语句也可以包含默认(default)分支,只有其它 **case** 分支都无法匹配控制表达式时,默认分支中的代码才会被执行。一个 **switch** 语句只能有一个默认分支,而且必须在 **switch** 语句的最后面。

尽管模式匹配操作实际的执行顺序,特别是模式的计算顺序是不可知的,但是 **Swift** 规定 **switch** 语句中的模式匹配的顺序和书写源代码的顺序保持一致。因此,当多个模式含有相同的值且能够匹配控制表达式时,程序只会执行源代码中第一个匹配的 **case** 分支中的代码。

Switch 语句必须是完备的

在 **Swift** 中, **switch** 语句中控制表达式的每一个可能的值都必须至少有一个 **case** 分支与之对应。在某些情况下 (例如,表达式的类型是 **Int**), 你可以使用默认块满足该要求。

不存在隐式的贯穿(fall through)

当匹配的 `case` 分支中的代码执行完毕后，程序会终止 `switch` 语句，而不会继续执行下一个 `case` 分支。这意味着，如果你想执行下一个 `case` 分支，需要显式地在你需要的 `case` 分支里使用 `fallthrough` 语句。关于 `fallthrough` 语句的更多信息，详情参见 `Fallthrough` 语句。

GRAMMAR OF A SWITCH STATEMENT

`switch-statement` → `switch expression { switch-cases opt }`

`switch-cases` → `switch-case switch-cases opt`

`switch-case` → `case-label statement | default-label statements`

`switch-case` → `case-label ; | default-label ;`

`case-label` → `case case-item-list :`

`case-item-list` → `pattern guard-clause opt | pattern guard-clause opt, case-item-list`

`default-label` → `default :`

`guard-clause` → `where guard-expression`

`guard-expression` → `expression`

带标签的语句

你可以在循环语句或 `switch` 语句前面加上标签，它由标签名和紧随其后的冒号(:)组成。在 `break` 和 `continue` 后面跟上标签名可以显式地在循环语句或 `switch` 语句中更改控制流，把控制权传递给指定标签标记的语句。关于这两条语句用法，详情参见 `Break` 语句和 `Continue` 语句。

标签的作用域是该标签所标记的语句之后的所有语句。你可以不使用带标签的语句，但只要

使用它，标签名就必唯一。

关于使用带标签的语句的例子，详情参见控制流一章的带标签的语句。

GRAMMAR OF A LABELED STATEMENT

labeled-statement \rightarrow statement-label loop-statement | statement-label switch-statement

statement-label \rightarrow label-name :

label-name \rightarrow identifier

控制传递语句

通过无条件地把控制权从一片代码传递到另一片代码，控制传递语句能够改变代码执行的顺序。Swift 提供四种类型的控制传递语句：**break** 语句、**continue** 语句、**fallthrough** 语句和 **return** 语句。

GRAMMAR OF A CONTROL TRANSFER STATEMENT

control-transfer-statement \rightarrow break-statement

control-transfer-statement \rightarrow continue-statement

control-transfer-statement \rightarrow fallthrough-statement

control-transfer-statement \rightarrow return-statement

Break 语句

break 语句用于终止循环或 **switch** 语句的执行。使用 **break** 语句时，可以只写 **break** 这个关键词，也可以在 **break** 后面跟上标签名（label name），像下面这样：

1. `break`
2. `break`label name``

当 `break` 语句后面带标签名时，可用于终止由这个标签标记的循环或 `switch` 语句的执行。

而当只写 `break` 时，则会终止 `switch` 语句或上下文中包含 `break` 语句的最内层循环的执行。

在这两种情况下，控制权都会被传递给循环或 `switch` 语句外面的第一行语句。

关于使用 `break` 语句的例子，详情参见控制流一章的 **Break** 和带标签的语句。

GRAMMAR OF A BREAK STATEMENT

`break-statement` → `break label-name opt`

Continue 语句

`continue` 语句用于终止循环中当前迭代的执行，但不会终止该循环的执行。使用 `continue` 语句时，可以只写 `continue` 这个关键词，也可以在 `continue` 后面跟上标签名(`label name`)，像下面这样：

1. `continue`
2. `continue`label name``

当 `continue` 语句后面带标签名时，可用于终止由这个标签标记的循环中当前迭代的执行。

而当只写 `break` 时，可用于终止上下文中包含 `continue` 语句的最内层循环中当前迭代的执行。

在这两种情况下，控制权都会被传递给循环外面的第一行语句。

在 `for` 语句中，`continue` 语句执行后，`increment` 表达式还是会被计算，这是因为每次循环体执行完毕后 `increment` 表达式都会被计算。

关于使用 `continue` 语句的例子，详情参见控制流一章的 `Continue` 和带标签的语句。

GRAMMAR OF A CONTINUE STATEMENT

`continue-statement` → `continue label-name opt`

Fallthrough 语句

`fallthrough` 语句用于在 `switch` 语句中传递控制权。`fallthrough` 语句会把控制权从 `switch` 语句中的一个 `case` 传递给下一个 `case`。这种传递是无条件的，即使下一个 `case` 的模式与 `switch` 语句的控制表达式的值不匹配。

`fallthrough` 语句可出现在 `switch` 语句中的任意 `case` 里，但不能出现在最后一个 `case` 分支中。同时，`fallthrough` 语句也不能把控制权传递给使用了可选绑定的 `case` 分支。

关于在 `switch` 语句中使用 `fallthrough` 语句的例子，详情参见控制流一章的控制传递语句。

GRAMMAR OF A FALLTHROUGH STATEMENT

`continue-statement` → `fallthrough`

Return 语句

`return` 语句用于在函数或方法的实现中将控制权传递给调用者，接着程序将会从调用者的位置继续向下执行。

使用 `return` 语句时，可以只写 `return` 这个关键词，也可以在 `return` 后面跟上表达式，像下面这样：

```
1. return
2. return `expression`
```

当 `return` 语句后面带表达式时，表达式的值将会返回给调用者。如果表达式值的类型与调用者期望的类型不匹配，`Swift` 则会在返回表达式的值之前将表达式值的类型转换为调用者

期望的类型。

而当只写 `return` 时，仅仅是将控制权从该函数或方法传递给调用者，而不返回一个值。（这就是说，该函数或方法的返回类型为 `Void` 或 `()`）

GRAMMAR OF A RETURN STATEMENT

`return-statement` → `return expression opt`

3.6 声明

本篇译者：marsprince（[Github 主页](#)），敬请勘误。

一条声明可以在你的程序里引入新的名字和构造。举例来说，你可以使用声明来引入函数和方法，变量和常量，或者来定义新的命名好的枚举，结构，类和协议类型。你也可以使用一条声明来延长一个已经存在的命名好的类型的行为。或者在你的程序里引入在其他地方声明的符号。

在 `swift` 中，大多数声明在某种意义上讲也是执行或同事声明它们的初始化定义。这意味着，因为协议和他们的成员不匹配，大多数协议成员需要单独的声明。为了方便起见，也因为这些区别在 `swift` 里不是很重要，声明语句同时包含了声明和定义。

GRAMMAR OF A DECLARATION

`declaration` → `import-declaration`

`declaration` → `constant-declaration`

`declaration` → `variable-declaration`

`declaration` → `typealias-declaration`

declaration → function-declaration

declaration → enum-declaration

declaration → struct-declaration

declaration → class-declaration

declaration → protocol-declaration

declaration → initializer-declaration

declaration → deinitializer-declaration

declaration → extension-declaration

declaration → subscript-declaration

declaration → operator-declaration

declarations → declarationdeclarationsopt

declaration-specifiers → declaration-specifierdeclaration-specifiersopt

declaration-specifier → class | mutating | nonmutating | override | static | unowned |

模块范围

模块范围定义了对模块中其他源文件可见的代码。（注：待改进）在 **swift** 的源文件中，最高级别的代码由零个或多个语句，声明和表达组成。变量，常量和其他的声明语句在一个源文件的最顶级被声明，使得他们对同一模块中的每个源文件都是可见的。

GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → statements opt

代码块

代码块用来将一些声明和控制结构的语句组织在一起。它有如下的形式：

```
1. {  
2.     `statements`  
3. }
```

代码块中的语句包括声明，表达式和各种其他类型的语句，它们按照在源码中的出现顺序被依次执行。

GRAMMAR OF A CODE BLOCK

code-block → statements opt

引入声明

引入声明使你可以使用在其他文件中声明的内容。引入语句的基本形式是引入整个代码模块；它由 `import` 关键字开始，后面紧跟一个模块名：

```
1. import module
```

你可以提供更多的细节来限制引入的符号，如声明一个特殊的子模块或者在一个模块或子模块中做特殊的声明。（待改进） 当你使用了这些细节后，在当前的程序汇总只有引入的符号是可用的（并不是声明的整个模块）。

```
1. import import kind module.symbol name  
2. import module.submodule
```

GRAMMAR OF AN IMPORT DECLARATION

import-declaration → attributes opt import import-kind opt import-path import-kind → typealias | struct | class | enum | protocol | var | func import-path → import-path-identifier

import-path-identifier.import-path import-path-identifier → identifier operator

常量声明

常量声明可以在你的程序里命名一个常量。常量以关键词 **let** 来声明，遵循如下的格式：

```
1. let constant name: type = expression
```

当常量的值被给定后，常量就将常量名称和表达式初始值不变的结合在了一起，而且不能更改。这意味着如果常量以类的形式被初始化，类本身的内容是可以改变的，但是常量和类之间的结合关系是不能改变的。当一个常量被声明为全局变量，它必须被给定一个初始值。当一个常量在类或者结构体中被声明时，他被认为是一个常量属性。常量并不是可计算的属性，因此不包含 **getters** 和 **setters**。（译者注：**getters** 和 **setters** 不知道怎么翻译，待改进）

如果常量名是一个元祖形式，元祖中的每一项初始化表达式中都要有对应的值

```
1. let (firstNumber, secondNumber) = (10, 42)
```

在上例中，**firstNumber** 是一个值为 10 的常量，**secondNumber** 是一个值为 42 的常量。所有常量都可以独立的使用：

```
1. 1 println("The first number is \(firstNumber).")
2. 2 // prints "The first number is 10."
3. 3 println("The second number is \(secondNumber).")
4. 4 // prints "The second number is 42."
```

类型注释(**:type**)在常量声明中是一个可选项，它可以用来描述在类型接口(**type inference**)中找到的类型。

声明一个静态常量要使用关键字 **static**。静态属性在类型属性(**type properties**)中有介绍。

如果还想获得更多关于常量的信息或者想在使用中获得帮助，请查看常量和变量(**constants and variables**)，存储属性(**stored properties**)等节。

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → attributes opt declaration-specifiers opt letpattern-initializer-list
pattern-initializer-list → pattern-initializer | pattern-initializer , pattern-initializer-list pattern-
initializer → pattern initializer opt initializer → =expression

变量声明

变量声明可以在你的程序里声明一个变量，它以关键字 **var** 来声明。根据声明变量类型和值的不同，如存储和计算 变量和属性，存储变量和属性监视，和静态变量属性，有着不同的声明形式。(待改进) 所使用的声明形式取决于变量所声明的范围和你打算声明的变量类型。

注意：你也可以在协议声明的上下文声明属性，详情参见类型属性声明。

存储型变量和存储型属性

下面的形式声明了一个存储型变量或存储型变量属性

```
1. var variable name: type = expression
```

你可以在全局，函数内，或者在类和结构体的声明(context)中使用这种形式来声明一个变量。当变量以这种形式 在全局或者一个函数内被声明时，它代表一个存储型变量。当他在类或者结构体中被声明时，他代表一个存储型变量属性。

构造器表达式可以被

和常量声明相比，如果变量名是一个元祖类型，元祖的每一项的名字都要和初始化表达式一致。

正如名字一样，存储型变量的值或存储型变量属性存储在内存中。

计算型变量和计算型属性

如下形式声明一个一个存储型变量或存储型属性：

```
1. var variable name: type {  
2.   get {  
3.       statements  
4.   }  
5.   set(setter name) {  
6.       statements  
7.   }  
8. }
```

你可以在全局，函数体内或者类，结构体，枚举，扩展声明的上下文中使用这种形式的声明。当变量以这种形式在全局或者一个函数内被声明时，它代表一个计算型变量。当他在类，结构体，枚举，扩展声明的上下文 中被声明时，他代表一个计算型变量属性。

getter 用来读取变量值，**setter** 用来写入变量值。**setter** 子句是可选的，只有 **getter** 是必需的，你可以将这些语句 都省略，只是简单的直接返回请求值，正如在只读计算属性(**read-only computed properties**)中描述的那样。但是如果你提供了一个 **setter** 语句，你也必需提供一个 **getter** 语句。

setter 的名字和圆括号内的语句是可选的。如果你写了一个 **setter** 名，它就会作为 **setter** 的参数被使用。如果你不写 **setter** 名，**setter** 的初始名为 **newValue**，正如在 **setter** 声明速记 (**shorthand setter declaration**)中提到的那样。

不像存储型变量和存储型属性那样，计算型属性和计算型变量的值不存储在内存中。

获得更多信息，查看更多关于计算型属性的例子，请查看计算型属性(**computed properties**)一节。

存储型变量监视器和属性监视器

你可以用 **willset** 和 **didset** 监视器来声明一个存储型变量或属性。一个包含监视器的存储型

变量或属性按如下的形式声明：

```
1. var variable name: type = expression {
2.   willSet(setter name) {
3.     statements
4.   }
5.   didSet(setter name {
6.     statements
7.   }
8. }
```

你可以在全局，函数体内或者类，结构体，枚举，扩展声明的上下文中使用这种形式的声明。当变量以这种形式在全局或者一个函数内被声明时，监视器代表一个存储型变量监视器；当他在类，结构体，枚举，扩展声明的上下文中被声明时，监视器代表属性监视器。

你可以为适合的监视器添加任何存储型属性。你也可以通过重写子类属性的方式为适合的监视器添加任何继承的属性（无论是存储型还是计算型的），参见重写属性监视器(overriding property observers)。

初始化表达式在类或者结构体的声明中是可选的，但是在其他地方是必需的。无论在什么地方声明，所有包含监视器的变量声明都必须有类型注释(type annotation)。

当变量或属性的值被改变时，**willset** 和 **didset** 监视器提供了一个监视方法（适当的回应）。监视器不会在变量或属性第一次初始化时不会被运行，他们只有在值被外部初始化语句改变时才会被运行。

willset 监视器只有在变量或属性值被改变之前运行。新的值作为一个常量经过 **willset** 监视器，因此不可以在 **willset** 语句中改变它。**didset** 监视器在变量或属性值被改变后立即运行。和 **willset** 监视器相反，为了以防止你仍然需要获得旧的数据，旧变量值或者属性会经过 **didset** 监视器。这意味着，如果你在变量或属性自身的 **didset** 监视器语句中设置了一个值，你设置的新值会取代刚刚在 **willset** 监视器中经过的那个值。

在 **willset** 和 **didset** 语句中，**setter** 名和圆括号的语句是可选的。如果你写了一个 **setter** 名，它就会作为 **willset** 和 **didset** 的参数被使用。如果你不写 **setter** 名，**willset** 监视器初始名为 **newValue**，**didset** 监视器初始名为 **oldvalue**。

当你提供一个 `willset` 语句时，`didset` 语句是可选的。同样的，在你提供了一个 `didset` 语句时，`willset` 语句是可选的。

获得更多信息，查看如何使用属性监视器的例子，请查看属性监视器(property observers)一节。

类和静态变量属性

`class` 关键字用来声明类的计算型属性。`static` 关键字用来声明类的静态变量属性。类和静态变量在类型属性(type properties)中有详细讨论。

GRAMMAR OF A VARIABLE DECLARATION

`variable-declaration` → `variable-declaration-head``pattern-initializer-list`

`variable-declaration` → `variable-declaration-head` `variable-name` `type-annotation` `code-block`

`variable-declaration` → `variable-declaration-head` `variable-name` `type-annotation` `getter-setter-block`

`variable-declaration` → `variable-declaration-head` `variable-name` `type-annotation` `getter-setter-keyword-block`

`variable-declaration` → `variable-declaration-head` `variable-name` `type-annotation``initializer`
`opt willSet-didSet-block`

`variable-declaration-head` → `attributes` `opt` `declaration-specifiers` `opt` `var` `variable-name`
→ `identifier`

`getter-setter-block` → {`getter-clause` `setter-clause` `opt`}

`getter-setter-block` → {setter-clause getter-clause}

`getter-clause` → attributes optgetcode-block

`setter-clause` → attributes opt set setter-name opt code-block

`setter-name` → (identifier)

`getter-setter-keyword-block` → {getter-keyword-clause setter-keyword-clause opt} getter-

`setter-keyword-block` → {setter-keyword-clause getter-keyword-clause}

`getter-keyword-clause` → attributes opt get

`setter-keyword-clause` → attributes opt set

`willSet-didSet-block` → {willSet-clause didSet-clause opt}

`willSet-didSet-block` → {didSet-clause willSet-clause}

`willSet-clause` → attributes opt willSet setter-name opt code-block

`didSet-clause` → attributes opt didSet setter-name opt code-block

类型的别名声明

类型别名的声明可以在你的程序里为一个已存在的类型声明一个别名。类型的别名声明以关键字 `typealias` 开始，遵循如下的形式：

```
1. typealias name = existing type
```

当一个类型被别名被声明后，你可以在你程序的任何地方使用别名来代替已存在的类型。已存在的类型可以是已经被命名的 类型或者是混合类型。类型的别名不产生新的类型，它只是简单的和已存在的类型做名称替换。

查看更多 Protocol Associated Type Declaration.

GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → typealias-head typealias-assignment
typealias-head → typealias-name
typealias-name → identifier
typealias-assignment → =type

函数声明

你可以使用函数声明在你的程序里引入新的函数。函数可以在类的上下文，结构体，枚举，或者作为方法的协议中被声明。函数声明使用关键字 **func**，遵循如下的形式：

```
1. func function name(parameters) -> return type {  
2.     statements  
3. }
```

如果函数不返回任何值，返回类型可以被忽略，如下所示：

```
1. func function name(parameters) {  
2.     statements  
3. }
```

每个参数的类型都要标明，它们不能被推断出来。初始时函数的参数是常值。在这些参数前面添加 **var** 使它们成为变量，作用域内任何对变量的改变只在函数体内有效，或者用 **inout** 使的这些改变可以在调用域内生效。更多关于 **in-out** 参数的讨论，参见 **in-out 参数(in-out parameters)**

函数可以使用元组类型作为返回值来返回多个变量。

函数定义可以出现在另一个函数声明内。这种函数被称作 **nested** 函数。更多关于 **nested** 函数的讨论，参见 **nested functions**。

参数名

函数的参数是一个以逗号分隔的列表。函数调用时的变量顺序必须和函数声明时的参数顺序一致。最简单的参数列表有着如下的形式：

```
1. parameter name: parameter type
```

对于函数参数来讲，参数名在函数体内被使用，而不是在函数调用时使用。对于方法参数，参数名在函数体内被使用，同时也在方法被调用时作为标签被使用。该方法的第一个参数名仅仅在函数体内被使用，就像函数的参数一样，举例来讲：

```
1. func f(x: Int, y: String) -> String {
2.     return y + String(x)
3. }
4. f(7, "hello") // x and y have no name
5.
6. class C {
7.     func f(x: Int, y: String) -> String {
8.         return y + String(x)
9.     }
10. }
11. let c = C()
12. c.f(7, y: "hello") // x 没有名称, y 有名称
```

你可以按如下的形式，重写参数名被使用的过程：

```
1. external parameter name local parameter name: parameter type
2. #parameter name: parameter type
3. _ local parameter name: parameter type
```

在本地参数前命名的第二名称(second name)使得参数有一个扩展名。且不同于本地的参数名。扩展参数名在函数被调用时必须被使用。对应的参数在方法或函数被调用时必须有扩展名。

在参数名前所写的哈希符号(#)代表着这个参数名可以同时作为外部或本体参数名来使用。等同于书写两次本地参数名。在函数或方法调用时，与其对应的语句必须包含这个名字。

本地参数名前的强调字符(`_`)使参数在函数被调用时没有名称。在函数或方法调用时，与其对应的语句必须没有名字。

特殊类型的参数

参数可以被忽略，值可以是变化的，并且提供一个初始值，这种方法有着如下的形式：

```
1.      : <#parameter type#>.
2. parameter name: parameter type...
3. parameter name: parameter type = default argument value
```

以强调符(`_`)命名的参数明确的在函数体内不能被访问。

一个以基础类型名的参数，如果紧跟着三个点(`...`)，被理解为是可变参数。一个函数至多可以拥有一个可变参数，且必须是最后一个参数。可变参数被作为该基本类型名的数组来看待。举例来讲，可变参数 `int...`被看做是 `int[]`。查看可变参数的使用例子，详见可变参数 (variadic parameters)一节。

在参数的类型后面有一个以等号(=)连接的表达式，这样的参数被看做有着给定表达式的初试值。如果参数在函数调用时被省略了，就会使用初始值。如果参数没有胜率，那么它在函数调用是必须有自己的名字.举例来讲，`f()`和 `f(x:7)`都是只有一个变量 `x` 的函数的有效调用，但是 `f(7)`是非法的，因为它提供了一个值而不是名称。

特殊方法

以 `self` 修饰的枚举或结构体方法必须以 `mutating` 关键字作为函数声明头。

子类重写的方法必须以 `override` 关键字作为函数声明头。不用 `override` 关键字重写的方法，使用了 `override` 关键字 却并没有重写父类方法都会报错。

和类型相关而不是和类型实例相关的方法必须在 `static` 声明的结构以或枚举内，亦或是以 `class` 关键字定义的类内。

柯里化函数和方法

柯里化函数或方法有着如下的形式：

```
1. func function name(parameters) (parameters) -> return type {  
2.     statements  
3. }
```

以这种形式定义的函数的返回值是另一个函数。举例来说，下面的两个声明时等价的：

```
1. func addTwoNumbers(a: Int) (b: Int) -> Int {  
2.     return a + b  
3. }  
4. func addTwoNumbers(a: Int) -> (Int -> Int) {  
5.     func addTheSecondNumber(b: Int) -> Int {  
6.         return a + b  
7.     }  
8.     return addTheSecondNumber  
9. }  
10.  
11. addTwoNumbers(4)(5) // Returns 9
```

多级柯里化应用如下

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → function-head function-name generic-parameter-clause opt-
function-signature function-body function-head → attributes opt declaration-specifiers opt
func function-name → identifier operator function-signature → parameter-clauses function-
result opt function-result → ->attributes opt type function-body → code-block parameter-
clauses → parameter-clause parameter-clauses opt parameter-clause → () (parameter-
list...opt) parameter-list → parameter parameter,parameter-list parameter → inout opt let
opt#optparameter-name local-parameter-name opt type-annotation default-argument-
clause opt parameter → inoutoptvar#optparameter-namelocal-parameter-name opt type-
annotationdefault-argument-clause opt parameter → attributes opt type parameter-name
→ identifier local-parameter-name → identifier default-argument-clause → =-
expression:

枚举声明

在你的程序里使用枚举声明来引入一个枚举类型。

枚举声明有两种基本的形式，使用关键字 `enum` 来声明。枚举声明体使用从零开始的变量——叫做枚举事件，和任意数量的 声明，包括计算型属性，实例方法，静态方法，构造器，类型别名，甚至其他枚举，结构体，和类。枚举声明不能 包含析构器或者协议声明。

不像类或者结构体。枚举类型并不提供隐式的初始构造器，所有构造器必须显式的声明。构造器可以委托枚举中的其他 构造器，但是构造过程仅当构造器将一个枚举时间完成后才全部完成。

和结构体类似但是和类不同，枚举是值类型：枚举实例在赋予变量或常量时，或者被函数调用时被复制。 更多关于值类型的信息，参见结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以扩展枚举类型，正如在扩展名声明(Extension Declaration)中讨论的一样。

任意事件类型的枚举

如下的形式声明了一个包含任意类型枚举时间的枚举变量

```
1. enum enumeration name {  
2.     case enumeration case 1  
3.     case enumeration case 2(associated value types)  
4. }
```

这种形式的枚举声明在其他语言中有时被叫做可识别联合(discriminated)。

这种形式中，每一个事件块由关键字 `case` 开始，后面紧接着一个或多个以逗号分隔的枚举事件。每一个事件名必须是 独一无二的。每一个事件也可以指定它所存储的指定类型的值，这些类型在关联值类型的元祖里被指定，立即书写在事件 名后。获得更多关于关联值类型的信息和例子，请查看关联值(associated values)一节。

使用原始事件值的枚举

以下的形式声明了一个包含相同基础类型的枚举事件的枚举：

```
1. enum enumeration name: raw value type {  
2.     case enumeration case 1 = raw value 1  
3.     case enumeration case 2 = raw value 2  
4. }
```

在这种形式中，每一个事件块由 **case** 关键字开始，后面紧接着一个或多个以逗号分隔的枚举事件。和第一种形式的枚举事件不同，这种形式的枚举事件包含一个同类型的基础值，叫做原始值(**raw value**)。这些值的类型在原始值类型(**raw value type**) 中被指定，必须是面上的整数，浮点数，字符或者字符串。

每一个事件必须有唯一的名字，必须有一个唯一的初始值。如果初始值类型被指定为 **int**，则不必为事件显式的指定值，它们会隐式的被标为值 0,1,2 等。每一个没有被赋值的 **Int** 类型时间会隐式的赋予一个初始值，它们是自动递增的。

```
1. enum ExampleEnum: Int {  
2.     case A, B, C = 5, D  
3. }
```

在上面的例子中，**ExampleEnum.A** 的值是 0，**ExampleEnum.B** 的值是 。因为 **ExampleEnum.C** 的值被显式的设定为 5，因此 **ExampleEnum.D** 的值会自动增长为 6。

枚举事件的初始值可以调用方法 **toRaw** 获得，如 **ExampleEnum.B.toRaw()**。你也可以通过调用 **fromRaw** 方法来使用初始值找到 其对应的事件，并返回一个可选的事件。查看更多信息和获取初始值类型事件的信息，参阅初始值(**raw values**)。

获得枚举事件

使用点(.)来引用枚举类型的事件，如 **EnumerationType.EnumerationCase**。当枚举类型可以上下文推断出时，你可以 省略它(.仍然需要)，参照枚举语法(**Enumeration Syntax**)和显式成员表达(**Implicit Member Expression**)。

使用 **switch** 语句来检验枚举事件的值，正如使用 **switch** 语句匹配枚举值 (Matching

Enumeration Values with a Switch Statement) 一节描述的那样。

枚举类型是模式匹配(pattern-matched)的, 和其相反的是 `switch` 语句 `case` 块中枚举事件匹配, 在枚举事件类型(Enumeration Case Pattern)中有描述。

GRAMMAR OF AN ENUMERATION DECLARATION

```
enum-declaration → attributesopt union-style-enum attributesopt raw-value-style-enum
union-style-enum → enum-name generic-parameter-clauseopt { union-style-enum-
membersopt } union-style-enum-members → union-style-enum-member union-style-enum-
membersopt union-style-enum-member → declaration union-style-enum-case-clause
union-style-enum-case-clause → attributesopt case union-style-enum-case-list union-style-
enum-case-list → union-style-enum-case union-style-enum-case, union-style-enum-case-
list union-style-enum-case → enum-case-name tuple-typeopt enum-name → identifier
enum-case-name → identifier raw-value-style-enum → enum-name generic-parameter-
clauseopt : type-identifier { raw-value-style-enum-membersopt } raw-value-style-enum-
members → raw-value-style-enum-member raw-value-style-enum-membersopt raw-value-
style-enum-member → declaration raw-value-style-enum-case-clause raw-value-style-
enum-case-clause → attributesopt case raw-value-style-enum-case-list raw-value-style-
enum-case-list → raw-value-style-enum-case raw-value-style-enum-case, raw-value-style-
enum-case-list raw-value-style-enum-case → enum-case-name raw-value-assignmentopt
raw-value-assignment → = literal
```

结构体声明

使用结构体声明可以在你的程序里引入一个结构体类型。结构体声明使用 `struct` 关键字, 遵循如下的形式:

```
1. struct structure name: adopted protocols {
2.     declarations
3. }
```

结构体内包含零或多个声明。这些声明可以包括存储型和计算型属性, 静态属性, 实例方法, 静态方法, 构造器, 类型别名, 甚至其他结构体, 类, 和枚举声明。结构体声明不能包含

析构器或者协议声明。详细讨论和包含多种结构体 声明的实例，参见类和结构体一节。

结构体可以包含任意数量的协议，但是不能继承自类，枚举或者其他结构体。

有三种方法可以创建一个声明过的结构体实例：

- 调用结构体内声明的构造器，参照构造器(initializers)一节。

- 如果没有声明构造器，调用结构体的逐个构造器，详情参见 **Memberwise Initializers for Structure Types**.

- 如果没有声明析构器，结构体的所有属性都有初始值，调用结构体的默认构造器，详情参见默认构造器(Default Initializers).

结构体的构造过程参见初始化(initialization)一节。

结构体实例属性可以用点(.)来获得，详情参见获得属性(Accessing Properties)一节。

结构体是值类型；结构体的实例在被赋予变量或常量，被函数调用时被复制。获得关于值类型更多信息，参见 结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以使用扩展声明来扩展结构体类型的行为，参见扩展声明(Extension Declaration).

GRAMMAR OF A STRUCTURE DECLARATION

```
struct-declaration → attributesoptstructstruct-namegeneric-parameter-clauseopttype-  
inheritance-clauseoptstruct-body struct-name → identifier struct-body → {declarationsopt}
```

类声明

你可以在你的程序中使用类声明来引入一个类。类声明使用关键字 **class**，遵循如下的形式：

```
1. class class name: superclass, adopted protocols {  
2.     declarations  
3. }
```

一个类内包含零或多个声明。这些声明可以包括存储型和计算型属性，实例方法，类方法，构造器，单独的析构器方法，类型别名，甚至其他结构体，类，和枚举声明。类声明不能包含协议声明。详细讨论和包含多种类声明的实例，参见类和结构体一节。

一个类只能继承一个父类，超类，但是可以包含任意数量的协议。这些超类第一次在 `type-inheritance-clause` 出现，遵循任意协议。

正如在初始化声明(Initializer Declaration)谈及的那样，类可以有指定和方便的构造器。当你声明任一构造器时，你可以使用 `required` 变量来标记构造器，要求任意子类来重写它。指定类的构造器必须初始化类所有的已声明的属性，它必须在子类构造器调用前被执行。

类可以重写属性，方法和它的超类的构造器。重写的方法和属性必须以 `override` 标注。

虽然超类的属性和方法声明可以被当前类继承，但是超类声明的指定构造器却不能。这意味着，如果当前类重写了超类的所有指定构造器，它就继承了超类的方便构造器。**Swift** 的类并不是继承自一个全局基础类。

有两种方法来创建已声明的类的实例：

- 调用类的一个构造器，参见构造器(initializers)。

- 如果没有声明构造器，而且类的所有属性都被赋予了初始值，调用类的默认构造器，参见默认构造器(default initializers)。

类实例属性可以用点(.)来获得，详情参见获得属性(Accessing Properties)一节。

类是引用类型；当被赋予常量或变量，函数调用时，类的实例是被引用，而不是复制。获得更多关于引用类型的信息，结构体和枚举都是值类型(Structures and Enumerations Are Value Types)一节。

你可以使用扩展声明来扩展类的行为，参见扩展声明(Extension Declaration)。

GRAMMAR OF A CLASS DECLARATION

class-declaration → attributesoptclassclass-namegeneric-parameter-clauseopttype-inheritance-clauseoptclass-body class-name → identifier class-body → {declarationsopt}

协议声明

一个协议声明为你的程序引入一个命名了的协议类型。协议声明使用 **protocol** 关键词来进行声明并有下面这样的形式：

```
1. protocol protocol name: inherited protocols {
2.     protocol member declarations
3. }
```

协议的主体包含零或多个协议成员声明，这些成员描述了任何采用该协议必须满足的一致性要求。特别的，一个协议可以声明必须实现某些属性、方法、初始化程序及附属脚本的一致性类型。协议也可以声明专用种类的类型别名，叫做关联类型，它可以指定协议的不同声明之间的关系。协议成员声明会在下面的详情里进行讨论。

协议类型可以从很多其它协议那继承。当一个协议类型从其它协议那继承的时候，来自其它协议的所有要求就集合了，而且从当前协议继承的任何类型必须符合所有的这些要求。对于如何使用协议继承的例子，查看协议继承

注意：你也可以使用协议合成类型集合多个协议的一致性要求，详情参见协议合成类型和协议合成

你可以通过采用在类型的扩展声明中的协议来为之前声明的类型添加协议一致性。在扩展中你必须实现所有采用协议的要求。如果该类型已经实现了所有的要求，你可以让这个扩展声明的主题留空。

默认地，符合某一个协议的类型必须实现所有声明在协议中的属性、方法和附属脚本。也就是说，你可以用 **optional** 属性标注这些协议成员声明以指定它们的一致性类型实现是可选

的。`optional` 属性仅仅可以用于使用 `objc` 属性标记过的协议。这样的结果就是仅仅类类型可以采用并符合包含可选成员要求的协议。更多关于如何使用 `optional` 属性的信息及如何访问可选协议成员的指导——比如当你不能肯定是否一致性的类型实现了它们——参见可选协议要求

为了限制协议的采用仅仅针对类类型，需要使用 `class_protocol` 属性标记整个协议声明。任意继承自标记有 `class_protocol` 属性协议的协议都可以智能地仅能被类类型采用。

注意：如果协议已经用 `object` 属性标记了，`class_protocol` 属性就隐性地应用于该协议；没有必要再明确地使用 `class_protocol` 属性来标记该协议了。

协议是命名的类型，因此它们可以以另一个命名类型出现在你代码的所有地方，就像协议类型里讨论的那样。然而你不能构造一个协议的实例，因为协议实际上不提供它们指定的要求的实现。

你可以使用协议来声明一个类的代理的方法或者应该实现的结构，就像委托(代理)模式描述的那样。

协议声明的语法 `protocol-declaration` \rightarrow `attributesoptprotocolprotocol-name`
`type-inheritance-clauseoptprotocol-body` `protocol-name` \rightarrow `identifier` `protocol-body` \rightarrow `{protocol-member-declarationsopt}` `protocol-member-declaration` \rightarrow `protocol-property-declaration`
`protocol-member-declaration` \rightarrow `protocol-method-declaration` `protocol-member-declaration` \rightarrow `protocol-initializer-declaration` `protocol-member-declaration` \rightarrow `protocol-subscript-declaration` `protocol-member-declaration` \rightarrow `protocol-associated-type-declaration` `protocol-member-declarations` \rightarrow `protocol-member-declarationprotocol-member-declarationsopt`

协议属性声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议属性声明来实现一个属性。协议属性声明有一种特殊的类型声明形式：

```
1. var property name: type { get set }
```

同其它协议成员声明一样，这些属性声明仅仅针对符合该协议的类型声明了 `getter` 和 `setter` 要求。结果就是你不需要在协议里它被声明的地方实现 `getter` 和 `setter`。

`getter` 和 `setter` 要求可以通过一致性类型以各种方式满足。如果属性声明包含 `get` 和 `set` 关键词，一致性类型就可以用可读写（实现了 `getter` 和 `setter`）的存储型变量属性或计算型属性，但是属性不能以常量属性或只读计算型属性实现。如果属性声明仅仅包含 `get` 关键词的话，它可以作为任意类型的属性被实现。比如说实现了协议的属性要求的一致性类型，参见属性要求

更多参见变量声明

协议属性声明语法 `protocol-property-declaration` → `variable-declaration-head`
`variable-nametype-annotationgetter-setter-keyword-block`

协议方法声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议方法声明来实现一个方法。协议方法声明和函数方法声明有着相同的形式，包含如下两条规则：他们不包括函数体，你不能在类的声明内为他们的参数提供初始值。举例来说，符合的类型执行协议必需的方法。参见必需方法一节。

使用关键字 `class` 可以在协议声明中声明一个类或必需的静态方法。执行这些方法的类也用关键字 `class` 声明。相反的，执行这些方法的结构体必须以关键字 `static` 声明。如果你想使用扩展方法，在扩展类时使用 `class` 关键字，在扩展结构体时使用 `static` 关键字。

更多请参阅函数声明。

GRAMMAR OF A PROTOCOL METHOD DECLARATION

`protocol-method-declaration` → `function-head`
`function-namegeneric-parameter-clauseopt-function-signature`

协议构造器声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议构造器声明来实现一个

构造器。协议构造器声明 除了不包含构造器体外，和构造器声明有着相同的形式，

更多请参阅构造器声明。

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

`protocol-initializer-declaration` → `initializer-headgeneric-parameter-clauseoptparameter-clause`

协议附属脚本声明

协议声明了一致性类型必须在协议声明的主体里通过引入一个协议附属脚本声明 来实现一个附属脚本。协议属性声明 对附属脚本声明有一个特殊的形式：

`subscript (parameters) -> return type { get set }`

附属脚本声明只为和协议一致的类型声明了必需的最小数量的 `getter` 和 `setter`。如果附属脚本申明包含 `get` 和 `set` 关键字，一致的类型也必须有一个 `getter` 和 `setter` 语句。如果附属脚本声明值包含 `get` 关键字，一致的类型必须至少包含一个 `getter` 语句，可以选择是否包含 `setter` 语句。

更多参阅附属脚本声明。

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

`protocol-subscript-declaration` → `subscript-headsubscript-resultgetter-setter-keyword-block`

协议相关类型声明

协议声明相关类型使用关键字 `typealias`。相关类型为作为协议声明的一部分的类型提供了一个别名。相关类型和参数 语句中的类型参数很相似，但是它们在声明的协议中包含 `self` 关键字。在这些语句中，`self` 指代和协议一致的可能的类型。 获得更多信息和例子，查看 相关类型或类型别名声明。

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

`protocol-associated-type-declaration` → `typealias-headtype-inheritance-clauseopt-typealias-assignmentopt`

构造器声明

构造器声明会为程序内的类，结构体或枚举引入构造器。构造器使用关键字 `init` 来声明，遵循两条基本形式。

结构体，枚举，类可以有任意数量的构造器，但是类的构造器的规则和行为是不一样的。不像结构体和枚举那样，类 有两种结构体，`designed initializers` 和 `convenience initializers`，参见构造器一节。

如下的形式声明了结构体，枚举和类的指定构造器：

```
1. init(parameters) {  
2.     statements  
3. }
```

类的指定构造器将类的所有属性直接初始化。如果类有超类，它不能调用该类的其他构造器，它只能调用超类的一个 指定构造器。如果该类从它的超类处继承了任何属性，这些属性在当前类内被赋值或修饰时，必须带哦用一个超类的 指定构造器。

指定构造器可以在类声明的上下文中声明，因此它不能用扩展声明的方法加入一个类中。

结构体和枚举的构造器可以带哦用其他的已声明的构造器，来委托其中一个火全部进行初始化过程。

以关键字 `convenience` 来声明一个类的便利构造器：

```
1. convenience init(parameters) {  
2.     statements  
3. }
```

便利构造器可以将初始化过程委托给另一个便利构造器或类的一个指定构造器。这意味着，类的初始化过程必须 以一个将所有类属性完全初始化的指定构造器的调用作为结束。便利构造器不能调用超类的构造器。

你可以使用 `required` 关键字，将便利构造器和指定构造器标记为每个子类的构造器都必须

拥有的。因为指定构造器 不被子类继承，他们必须被立即执行。当子类直接执行所有超类的指定构造器(或使用便利构造器重写指定构造器)时， 必需的便利构造器可以被隐式的执行，亦可以被继承。不像方法，附属脚本那样，你不需要为这些重写的构造器标注 **override** 关键字。

查看更多关于不同声明方法的构造器的例子，参阅构造过程一节。

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → initializer-headgeneric-parameter-clauseoptparameter-clause-initializer-body
initializer-head → attributesoptconvenienceoptinit
initializer-body → code-block

析构声明

析构声明为类声明了一个析构器。析构器没有参数，遵循如下的格式：

```
1. deinit {  
2.     statements  
3. }
```

当类没有任何语句时将要被释放时，析构器会自动的被调用。析构器在类的声明体内只能被声明一次——但是不能在 类的扩展声明内，每个类最多只能有一个。

子类继承了它的超类的析构器，在子类将要被释放时隐式的调用。子类在所有析构器被执行完毕前不会被释放。

析构器不会被直接调用。

查看例子和如何在类的声明中使用析构器，参见析构过程一节 。

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → attributesoptdeinitcode-block

扩展声明

扩展声明用于扩展一个现存的类，结构体，枚举的行为。扩展声明以关键字 **extension** 开始，遵循如下的规则：

```
1. extension type: adopted protocols {  
2.     declarations  
3. }
```

一个扩展声明体包括零个或多个声明。这些声明可以包括计算型属性，计算型静态属性，实例方法，静态和类方法，构造器，附属脚本声明，甚至其他结构体，类，和枚举声明。扩展声明不能包含析构器，协议声明，存储型属性，属性监测器或其他 的扩展属性。详细讨论和查看包含多种扩展声明的实例，参见扩展一节。

扩展声明可以向现存的类，结构体，枚举内添加一致的协议。扩展声明不能向一个类中添加继承的类，因此 **type-inheritance-clause** 是一个只包含协议列表的扩展声明。

属性，方法，现存类型的构造器不能被它们类型的扩展所重写。

扩展声明可以包含构造器声明，这意味着，如果你扩展的类型在其他模块中定义，构造器声明必须委托另一个在 那个模块里声明的构造器来恰当的初始化。

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → **extension** **type-identifier** **type-inheritance-clause** **opt** **extension-body**
extension-body → { **declarations** **opt** }

附属脚本声明

附属脚本用于向特定类型添加附属脚本支持，通常为访问集合，列表和序列的元素时提供语法便利。附属脚本声明使用关键字 **subscript**，声明形式如下：

```
subscript (parameter) -> (return type){ get{ statements } set(setter name){ statements } }
```

附属脚本声明只能在类，结构体，枚举，扩展和协议声明的上下文进行声明。

变量(parameters)指定一个或多个用于在相关类型的附属脚本中访问元素的索引（例如，表达式 `object[i]` 中的 `i`）。尽管用于元素访问的索引可以是任意类型的，但是每个变量必须包含一个用于指定每种索引类型的类型标注。返回类型(return type)指定被访问的元素的类型。

和计算性属性一样，附属脚本声明支持对访问元素的读写操作。`getter` 用于读取值，`setter` 用于写入值。`setter` 子句是可选的，当仅需要一个 `getter` 子句时，可以将二者都忽略且直接返回请求的值即可。也就是说，如果使用了 `setter` 子句，就必须使用 `getter` 子句。

`setter` 的名字和封闭的括号是可选的。如果使用了 `setter` 名称，它会被当做传给 `setter` 的变量的名称。如果不使用 `setter` 名称，那么传给 `setter` 的变量的名称默认是 `value`。`setter` 名称的类型必须与返回类型(return type)的类型相同。

可以在附属脚本声明的类型中，可以重载附属脚本，只要变量(parameters)或返回类型(return type)与先前的不同即可。此时，必须使用 `override` 关键字声明那个被覆盖的附属脚本。(注：好乱啊！到底是重载还是覆盖？！)

同样可以在协议声明的上下文中声明附属脚本，Protocol Subscript Declaration 中有所描述。

更多关于附属脚本和附属脚本声明的例子，请参考 Subscripts。

GRAMMAR OF A SUBSCRIPT DECLARATION

```
subscript-declaration → subscript-head subscript-result code-block subscript-declaration →  
subscript-head subscript-result getter-setter-block subscript-declaration → subscript-head-  
subscript-result getter-setter-keyword-block subscript-head → attributes opt subscript-  
parameter-clause subscript-result → -> attributes opt type
```

运算符声明

运算符声明会向程序中引入中缀、前缀或后缀运算符，它使用上下文关键字 `operator` 声明。可以声明三种不同的缀性：中缀、前缀和后缀。操作符的缀性描述了操作符与它的操作数的相对位置。运算符声明有三种基本形式，每种缀性各一种。运算符的缀性通过在 `operator`

和运算符之间添加上下文关键字 `infix`, `prefix` 或 `postfix` 来指定。每种形式中, 运算符的名字只能包含 `Operators` 中定义的运算符字符。

下面的这种形式声明了一个新的中缀运算符:

```
operator infix operator name{ precedence precedence level associativity associativity }
```

中缀运算符是二元运算符, 它可以被置于两个操作数之间, 比如表达式 `1 + 2` 中的加法运算符(+).

中缀运算符可以可选地指定优先级, 结合性, 或两者同时指定。

运算符的优先级可以指定在没有括号包围的情况下, 运算符与它的操作数如何紧密绑定的。可以使用上下文关键字 `precedence` 并优先级(`precedence level`)一起来指定一个运算符的优先级。优先级可以是 `0` 到 `255` 之间的任何一个数字(十进制整数); 与十进制整数字面量不同的是, 它不可以包含任何下划线字符。尽管优先级是一个特定的数字, 但它仅用作与另一个运算符比较(大小)。也就是说, 一个操作数可以同时被两个运算符使用时, 例如 `2 + 3 * 5`, 优先级更高的运算符将优先与操作数绑定。

运算符的结合性可以指定在没有括号包围的情况下, 优先级相同的运算符以何种顺序被分组的。可以使用上下文关键字 `associativity` 并结合性(`associativity`)一起来指定一个运算符的结合性, 其中结合性可以说是上下文关键字 `left`, `right` 或 `none` 中的任何一个。左结合运算符以从左到右的形式分组。例如, 减法运算符(-)具有左结合性, 因此 `4 - 5 - 6` 被以 `(4 - 5) - 6` 的形式分组, 其结果为-7。右结合运算符以从右到左的形式分组, 对于设置为 `none` 的非结合运算符, 它们不以任何形式分组。具有相同优先级的非结合运算符, 不可以互相邻接。例如, 表达式 `1 < 2 < 3` 非法的。

声明时不指定任何优先级或结合性的中缀运算符, 它们的优先级会被初始化为 `100`, 结合性被初始化为 `none`。

下面的这种形式声明了一个新的前缀运算符:

```
operator prefix operator name{ }
```

紧跟在操作数前边的前缀运算符(`prefix operator`)是一元运算符, 例如表达式 `++i` 中的前缀递

增运算符(++).

前缀运算符的声明中不指定优先级。前缀运算符是非结合的。

下面的这种形式声明了一个新的后缀运算符:

```
operator postfix operator name{}
```

紧跟在操作数后边的后缀运算符(postfix operator)是一元运算符, 例如表达式 `i++` 中的前缀递增运算符(++).

和前缀运算符一样, 后缀运算符的声明中不指定优先级。后缀运算符是非结合的。

声明了一个新的运算符以后, 需要声明一个跟这个运算符同名的函数来实现这个运算符。如何实现一个新的运算符, 请参考 [Custom Operators](#)。

GRAMMAR OF AN OPERATOR DECLARATION

```
operator-declaration → prefix-operator-declaration postfix-operator-declaration >infix-  
operator-declaration prefix-operator-declaration → operator prefix operator{} postfix-  
operator-declaration → operator postfix operator{} infix-operator-declaration → operator-  
infixoperator{infix-operator-attributesopt} infix-operator-attributes → precedence-clause-  
optassociativity-clauseopt precedence-clause → precedenceprecedence-level  
precedence-level → Digit 0 through 255 associativity-clause → associativityassociativity  
associativity → left right none
```

3.7 特性

本篇译者: Hawstein ([Github 主页](#)), 敬请勘误。

特性提供了关于声明和类型的更多信息。在 **Swift** 中有两类特性, 用于修饰声明的以及用于修饰类型的。例如, **required** 特性, 当应用于一个类的指定或便利初始化器声明时, 表明它的每个子类都必须实现那个初始化器。再比如 **noreturn** 特性, 当应用于函数或方法类型时,

表明该函数或方法不会返回到它的调用者。

通过以下方式指定一个特性：符号@后面跟特性名，如果包含参数，则把参数带上：

```
@ attribute name
```

```
@ attribute name ( attribute arguments )
```

有些声明特性通过接收参数来指定特性的更多信息以及它是如何修饰一个特定的声明的。这些特性的参数写在小括号内，它们的格式由它们所属的特性来定义。

声明特性

声明特性只能应用于声明。然而，你也可以将 `noreturn` 特性应用于函数或方法类型。

assignment

该特性用于修饰重载了复合赋值运算符的函数。重载了复合赋值运算符的函数必需将它们的初始输入参数标记为 `inout`。如何使用 `assignment` 特性的一个例子，请见：复合赋值运算符。

class_protocol

该特性用于修饰一个协议表明该协议只能被类类型采用[待改：adopted]。

如果你用 `objc` 特性修饰一个协议，`class_protocol` 特性就会隐式地应用到该协议，因此无需显式地用 `class_protocol` 特性标记该协议。

exported

该特性用于修饰导入声明，以此来导出已导入的模块，子模块，或当前模块的声明。如果另一个模块导入了当前模块，那么那个模块可以访问当前模块的导出项。

final

该特性用于修饰一个类或类中的属性，方法，以及下标成员。如果用它修饰一个类，那么这个类则不能被继承。如果用它修饰类中的属性，方法或下标，则表示在子类中，它们不能被重写。

lazy

该特性用于修饰类或结构体中的存储型变量属性，表示该属性的初始值最多只被计算和存储一次，且发生在第一次访问它时。如何使用 **lazy** 特性的一个例子，请见：惰性存储型属性。

noreturn

该特性用于修饰函数或方法声明，表明该函数或方法的对应类型，**T**，是 **@noreturn T**。你可以用这个特性修饰函数或方法的类型，这样一来，函数或方法就不会返回到它的调用者中去。

对于一个没有用 **noreturn** 特性标记的函数或方法，你可以将它重写(**override**)为用该特性标记的。相反，对于一个已经用 **noreturn** 特性标记的函数或方法，你则不可以将它重写为没使用该特性标记的。相同的规则试用于当你在一个 **conforming** 类型中实现一个协议方法时。

NSCopying

该特性用于修饰一个类的存储型变量属性。该特性将使属性的 **setter** 与属性值的一个副本合成，由 **copyWithZone** 方法返回，而不是属性本身的值。该属性的类型必需遵循 **NSCopying** 协议。

NSCopying 特性的行为与 **Objective-C** 中的 **copy** 特性相似。

NSManaged

该特性用于修饰 **NSManagedObject** 子类中的存储型变量属性，表明属性的存储和实现由 **Core Data** 在运行时基于相关实体描述动态提供。

objc

该特性用于修饰任意可以在 **Objective-C** 中表示的声明，比如，非嵌套类，协议，类和协议中的属性和方法（包含 **getter** 和 **setter**），初始化器，析构器，以下下标。**objc** 特性告诉编译器该声明可以在 **Objective-C** 代码中使用。

如果你将 **objc** 特性应用于一个类或协议，它也会隐式地应用于那个类或协议的成员。对于标记了 **objc** 特性的类，编译器会隐式地为它的子类添加 **objc** 特性。标记了 **objc** 特性的协议不能继承自没有标记 **objc** 的协议。

`objc` 特性有一个可选的参数，由标记符组成。当你想把 `objc` 所修饰的实体以一个不同的名字暴露给 Objective-C，你就可以使用这个特性参数。你可以使用这个参数来命名类，协议，方法，`getters`，`setters`，以及初始化器。下面的例子把 `ExampleClass` 中 `enabled` 属性的 `getter` 暴露给 Objective-C，名字是 `isEnabled`，而不是它原来的属性名。

```
1. @objc
2. class ExampleClass {
3.     var enabled: Bool {
4.         @objc(isEnabled) get {
5.             // Return the appropriate value
6.         }
7.     }
8. }
```

optional

用该特性修饰协议的属性，方法或下标成员，表示实现这些成员并不需要一致性类型（conforming type）。

你只能用 `optional` 特性修饰那些标记了 `objc` 特性的协议。因此，只有类类型可以 `adopt` 和 `conform to` 那些包含可选成员需求的协议。更多关于如何使用 `optional` 特性以及如何访问可选协议成员的指导，例如，当你不确定一个 `conforming` 类型是否实现了它们，请见：可选协议需求。

required

用该特性修饰一个类的指定或便利初始化器，表示该类的所有子类都必需实现该初始化器。

加了该特性的指定初始化器必需显式地实现，而便利初始化器既可显式地实现，也可以在子类实现了超类所有指定初始化器后继承而来（或者当子类使用便利初始化器重写了指定初始化器）。

Interface Builder 使用的声明特性

Interface Builder 特性是 Interface Builder 用来与 Xcode 同步的声明特性。Swift 提供了以下的 Interface Builder 特性：`IBAction`，`IBDesignable`，`IBInspectable`，以及 `IBOutlet`。这些特性与 Objective-C 中对应的特性在概念上是相同的。

`IBOutlet` 和 `IBInspectable` 用于修饰一个类的属性声明；`IBAction` 特性用于修饰一个类的方法声明；`IBDesignable` 用于修饰类的声明。

类型特性

类型特性只能用于修饰类型。然而，你也可以用 `noreturn` 特性去修饰函数或方法声明。

`auto_closure`

这个特性通过自动地将表达式封闭到一个无参数闭包中来延迟表达式的求值。使用该特性修饰无参的函数或方法类型，返回表达式的类型。一个如何使用 `auto_closure` 特性的例子，见函数类型

`noreturn`

该特性用于修饰函数或方法的类型，表明该函数或方法不会返回到它的调用者中去。你也可以用它标记函数或方法的声明，表示函数或方法的相应类型，`T`，是 `@noreturn T`。

特性的语法：

attribute → `@attribute-nameattribute-argument-clauseopt`

attribute-name → [identifier](#)

attribute-argument-clause → `(balanced-tokensopt)`

attributes → `attributeattributesopt`

balanced-tokens → `balanced-tokenbalanced-tokensopt`

balanced-token → `(balanced-tokensopt)`

balanced-token → `[balanced-tokensopt]`

balanced-token → `{balanced-tokensopt}`

balanced-token → Any identifier, keyword, literal, or operator

balanced-token → Any punctuation except `(,), [,], {, or }`

3.8 模式

本篇译者：honghaoz ([Github 主页](#))，敬请勘误。

模式（**pattern**）代表了单个值或者复合值的结构。例如，元组(1, 2)的结构是逗号分隔的，包含两个元素的列表。因为模式代表一种值的结构，而不是特定的某个值，你可以把模式和各种同类型的值匹配起来。比如，(x, y)可以匹配元组(1, 2)，以及任何含两个元素的元组。除了将模式与一个值匹配外，你可以从合成值中提取出部分或全部，然后分别把各个部分和一个常量或变量绑定起来。

在 **Swift** 中，模式出现在变量和常量的声明（在它们的左侧），**for-in** 语句和 **switch** 语句（在他们的 **case** 标签）中。尽管任何模式都可以出现在 **switch** 语句的 **case** 标签中，但在其他情况下，只有通配符模式（**wildcard pattern**），标识符模式（**identifier pattern**）和包含这两种模式的模式才能出现。

你可以为通配符模式（**wildcard pattern**），标识符模式（**identifier pattern**）和元组模式（**tuple pattern**）指定类型注释，用来限制这种模式只匹配某种类型的值。

模式的语法：

`pattern → wildcard-pattern type-annotationopt`

`pattern → identifier-pattern type-annotationopt`

`pattern → value-binding-pattern`

`pattern → tuple-pattern type-annotationopt`

`pattern → enum-case-pattern`

`pattern → type-casting-pattern`

`pattern → expression-pattern`

通配符模式（**Wildcard Pattern**）

通配符模式匹配并忽略任何值，包含一个下划线（`_`）。当你不关心被匹配的值时，可以使用

此模式。例如，下面这段代码进行了 1...3 的循环，并忽略了每次循环的值：

```
1. for _ in 1...3 {  
2.     // Do something three times.  
3. }
```

通配符模式的语法：wildcard-pattern → _

标识符模式（Identifier Pattern）

标识符模式匹配任何值，并将匹配的值和一个变量或常量绑定起来。例如，在下面的常量申明中，`someValue` 是一个标识符模式，匹配了类型是 `Int` 的 42。

```
1. let someValue = 42
```

当匹配成功时，42 被绑定（赋值）给常量 `someValue`。

当一个变量或常量申明的左边是标识符模式时，此时，标识符模式是隐式的值绑定模式（value-binding pattern）。

标识符模式的语法：identifier-pattern → identifier

值绑定模式（Value-Binding Pattern）

值绑定模式绑定匹配的值到一个变量或常量。当绑定匹配值给常量时，用关键字 `let`，绑定给变量时，用关键字 `var`。

标识符模式包含在值绑定模式中，绑定新的变量或常量到匹配的值。例如，你可以分解一个元组的元素，并把每个元素绑定到相应的标识符模式中。

```
1. let point = (3, 2)  
2. switch point {  
3.     // Bind x and y to the elements of point.  
4. case let (x, y):  
5.     println("The point is at \(x), \(y).")  
6. }
```

```
7. // prints "The point is at (3, 2)."
```

在上面这个例子中，`let` 将元组模式`(x, y)`分配到各个标识符模式。因为这种行为，`switch` 语句中 `case let (x, y):`和 `case (let x, let y):`匹配的值是一样的。

值绑定模式的语法：`value-binding-pattern` \rightarrow `var pattern` | `let pattern`

元组模式（Tuple Pattern）

元组模式是逗号分隔的列表，包含一个或多个模式，并包含在一对圆括号中。元组模式匹配相应元组类型的值。

你可以使用类型注释来限制一个元组模式来匹配某种元组类型。例如，在常量申明 `let (x, y): (Int, Int) = (1, 2)`中的元组模式`(x, y): (Int, Int)`，只匹配两个元素都是 `Int` 这种类型的元组。如果仅需要限制一个元组模式中的某几个元素，只需要直接对这几个元素提供类型注释即可。例如，在 `let (x: String, y)`中的元组模式，只要某个元组类型是包含两个元素，且第一个元素类型是 `String`，则被匹配。

当元组模式被用在 `for-in` 语句或者变量或常量申明时，它可以包含通配符模式，标识符模式或者其他包含这两种模式的模式。例如，下面这段代码是不正确的，因为`(x, 0)`中的元素 `0` 是一个表达式模式：

```
1. let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
2.           // This code isn't valid.
3.     for (x, 0) in points {
4.         /* ... */
5.     }
```

对于只包含一个元素的元组，括号是不起作用的。模式匹配那个单个元素的类型。例如，下面是等效的：

```
1. let a = 2           // a: Int = 2
2. let (a) = 2          // a: Int = 2
3. let (a): Int = 2     // a: Int = 2
```

元组模式的语法：

`tuple-pattern` → `(tuple-pattern-element-list opt)`

`tuple-pattern-element-list` → `tuple-pattern-element` | `tuple-pattern-element, tuple-pattern-element-list`

`tuple-pattern-element` → `pattern`

枚举案例模式（Enumeration Case Pattern）

枚举案例模式匹配现有的枚举类型的某种案例。枚举案例模式仅在 `switch` 语句中的 `case` 标签中出现。

如果你准备匹配的枚举案例有任何关联的值，则相应的枚举案例模式必须指定一个包含每个关联值元素的元组模式。关于使用 `switch` 语句来匹配包含关联值枚举案例的例子，请参阅 [Associated Values](#)。

枚举案例模式的语法：

`enum-case-pattern` → `type-identifier opt . enum-case-name tuple-pattern op`

类型转换模式（Type-Casting Patterns）

有两种类型转换模式，`is` 模式和 `as` 模式。这两种模式均只出现在 `switch` 语句中的 `case` 标签中。`is` 模式和 `as` 模式有以下形式：

1. `is type`
2. `pattern as type`

`is` 模式匹配一个值，如果这个值的类型在运行时（`runtime`）和 `is` 模式右边的指定类型（或者那个类型的子类）是一致的。`is` 模式和 `is` 操作符一样，他们都进行类型转换，但是抛弃了返回的类型。

`as` 模式匹配一个值，如果这个值的类型在运行时（`runtime`）和 `as` 模式右边的指定类型（或者那个类型的子类）是一致的。一旦匹配成功，匹配的值的类型被转换成 `as` 模式左边指定

的模式。

关于使用 `switch` 语句来匹配 `is` 模式和 `as` 模式值的例子，请参阅 [Type Casting for Any and AnyObject](#)。

类型转换模式的语法：

`type-casting-pattern` → `is-pattern as-pattern`

`is-pattern` → `istype`

`as-pattern` → `patternastype`

表达式模式（Expression Pattern）

表达式模式代表了一个表达式的值。这个模式只出现在 `switch` 语句中的 `case` 标签中。

由表达式模式所代表的表达式用 **Swift** 标准库中的 `~=` 操作符与输入表达式的值进行比较。如果 `~=` 操作符返回 `true`，则匹配成功。默认情况下，`~=` 操作符使用 `==` 操作符来比较两个相同类型的值。它也可以匹配一个整数值与一个 `Range` 对象中的整数范围，正如下面这个例子所示：

```
1. let point = (1, 2)
2. switch point {
3.     case (0, 0):
4.         println("(0, 0) is at the origin.")
5.     case (-2...2, -2...2):
6.         println("\(point.0), \(point.1) is near the origin.")
7.     default:
8.         println("The point is at \(point.0), \(point.1).")
9. }
10. // prints "(1, 2) is near the origin."
```

你可以重载 `~=` 操作符来提供自定义的表达式行为。例如，你可以重写上面的例子，以实现用字符串表达的点来比较 `point` 表达式。

```
1. // Overload the ~= operator to match a string with an integer
2. func ~= (pattern: String, value: Int) -> Bool {
3.     return pattern == "\(value)"
4. }
5. switch point {
6. case ("0", "0"):
7.     println("(0, 0) is at the origin.")
8. case ("-2...2", "-2...2"):
9.     println("\(point.0), \(point.1) is near the origin.")
10. default:
11.     println("The point is at \(point.0), \(point.1).")
12. }
13. // prints "(1, 2) is near the origin."
```

表达式模式的语法: `expression-pattern` \rightarrow `expression`

3.9 泛型参数

本篇译者: fd5788 ([Github 主页](#)), 敬请勘误。

本节涉及泛型类型、泛型函数以及泛型构造器的参数, 包括形参和实参。声明泛型类型、函数或构造器时, 须指定相应的类型参数。类型参数相当于一个占位符, 当实例化泛型类型、调用泛型函数或泛型构造器时, 就用具体的类型实参替代之。

关于 Swift 语言的泛型概述, 见泛型(第二部分第 22 章)。

泛型形参语句

泛型形参语句指定泛型类型或函数的类型形参, 以及这些参数的关联约束和要求。泛型形参语句用尖括号 ($\langle \rangle$) 包住, 并且有以下两种形式:

`< generic parameter list >`

`< generic parameter list where requirements >`

泛型形参列表中泛型形参用逗号分开, 每一个采用以下形式:

`type parameter` : `constraint`

泛型形参由两部分组成：类型形参及其后的可选约束。类型形参只是占位符类型（如 `T`，`U`，`V`，`KeyType`，`ValueType` 等）的名字而已。你可以在泛型类型、函数的其余部分或者构造器声明，以及函数或构造器的签名中使用它。

约束用于指明该类型形参继承自某个类或者遵守某个协议或协议的一部分。例如，在下面的泛型中，泛型形参 `T: Comparable` 表示任何用于替代类型形参 `T` 的类型实参必须满足 `Comparable` 协议。

```
1. func simpleMin<T: Comparable>(x: T, y: T) -> T {
2.     if x < y {
3.         return y
4.     }
5.     return x
6. }
```

如，`Int` 和 `Double` 均满足 `Comparable` 协议，该函数接受任何一种类型。与泛型类型相反，调用泛型函数或构造器时不需要指定泛型实参语句。类型实参由传递给函数或构造器的实参推断而出。

```
1. simpleMin(17, 42) // T is inferred to be Int
2. simpleMin(3.14159, 2.71828) // T is inferred to be Double
```

Where 语句

要想对类型形参及其关联类型指定额外要求，可以在泛型形参列表之后添加 `where` 语句。`where` 语句由关键字 `where` 及其后的用逗号分割的多个要求组成。

`where` 语句中的要求用于指明该类型形参继承自某个类或遵守某个协议或协议的一部分。尽管 `where` 语句有助于表达类型形参上的简单约束（如 `T: Comparable` 等同于 `T where T: Comparable`，等等），但是依然可以用来对类型形参及其关联约束提供更复杂的约束。如，`<T where T: C, T: P>` 表示泛型类型 `T` 继承自类 `C` 且遵守协议 `P`。

如上所述，可以强制约束类型形参的关联类型遵守某个协议。<T: Generator where T.Element: Equatable>表示 T 遵守 Generator 协议，而且 T 的关联类型 T.Element 遵守 Equatable 协议（T 有关联类型是因为 Generator 声明了 Element，而 T 遵守 Generator 协议）。

也可以用操作符==来指定两个类型等效的要求。例如，有这样一个约束：T 和 U 遵守 Generator 协议，同时要求它们的关联类型等同，可以这样来表达：<T: Generator, U: Generator where T.Element == U.Element>。

当然，替代类型形参的类型实参必须满足所有类型形参所要求的约束和要求。

泛型函数或构造器可以重载，但在泛型形参语句中的类型形参必须有不同的约束或要求，抑或二者皆不同。当调用重载的泛型函数或构造器时，编译器会用这些约束来决定调用哪个重载函数或构造器。

泛型类可以生成一个子类，但是这个子类也必须是泛型类。

GRAMMAR OF A GENERIC PARAMETER CLAUSE

```
generic-parameter-clause → <generic-parameter-listrequirement-clauseopt>
generic-parameter-list → generic-parameter generic-parameter, generic-parameter-list
generic-parameter → type-name
generic-parameter → type-name : type-identifier
generic-parameter → type-name : protocol-composition-type
requirement-clause → where requirement-list
requirement-list → requirement requirement, requirement-list
requirement → conformance-requirement same-type-requirement
conformance-requirement → type-identifier : type-identifier
conformance-requirement → type-identifier : protocol-composition-type
same-type-requirement → type-identifier == type-identifier
```

泛型实参语句

泛型实参语句指定泛型类型的类型实参。泛型实参语句用尖括号（<>）包住，形式如下：

```
< generic argument list >
```

泛型实参列表中类型实参有逗号分开。类型实参是实际具体类型的名字，用来替代泛型类型的泛型形参语句中的相应的类型形参。从而得到泛型类型的一个特化版本。如，Swift 标准库的泛型字典类型定义如下：

```
1. struct Dictionary<KeyType: Hashable, ValueType>: Collection, DictionaryLiteralConvertible {
    yLiteralConvertible {
2.     /* ... */
3. }
```

泛型 `Dictionary` 类型的特化版本，`Dictionary<String, Int>` 就是用具体的 `String` 和 `Int` 类型替代泛型类型 `KeyType: Hashable` 和 `ValueType` 产生的。每一个类型实参必须满足它所替代的泛型形参的所有约束，包括任何 `where` 语句所指定的额外的要求。上面的例子中，类型形参 `KeyType` 要求满足 `Hashable` 协议，因此 `String` 也必须满足 `Hashable` 协议。

可以用本身就是泛型类型的特化版本的类型实参替代类型形参（假设已满足合适的约束和要求）。例如，为了生成一个元素类型是整型数组的数组，可以用数组的特化版本 `Array<Int>` 替代泛型类型 `Array<T>` 的类型形参 `T` 来实现。

```
1. let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

如泛型形参语句所述，不能用泛型实参语句来指定泛型函数或构造器的类型实参。

泛型实参语法：

generic-argument-clause → [<generic-argument-list>](#)

generic-argument-list → [generic-argument](#) [generic-argument](#), [generic-argument-list](#)

generic-argument → [type](#)

3.10 语法总结

本篇译者：StanZhai ([Github 主页](#))，敬请勘误。

语句

语句语法

statement → [expression](#); *opt*

statement → [declaration](#); *opt*

statement → [loop-statement](#); *opt*

statement → [branch-statement](#); *opt*

statement → [labeled-statement](#)

statement → [control-transfer-statement](#); *opt*

statements → [statement](#) [statements](#) *opt*

循环语句语法

loop-statement → [for-statement](#)

loop-statement → [for-in-statement](#)

loop-statement → [while-statement](#)

loop-statement → [do-while-statement](#)

For 循环语法

for-statement → **for** [for-init](#) *opt*; [expression](#) *opt*; [expression](#) *opt* [code-block](#)

for-statement → **for** ([for-init](#) *opt*; [expression](#) *opt*; [expression](#) *opt*) [code-block](#)

for-init → [variable-declaration](#) | [expression-list](#)

For-In 循环语法

for-in-statement → **for** [pattern](#) **in** [expression](#) [code-block](#)

While 循环语法

while-statement → **while** [while-condition](#) [code-block](#)

while-condition → [expression](#) | [declaration](#)

Do-While 循环语法

do-while-statement → **do** [code-block](#) **while** [while-condition](#)

分支语句语法

branch-statement → [if-statement](#)

branch-statement → [switch-statement](#)

If 语句语法

if-statement → **if** [if-condition](#) [code-block](#) [else-clause](#) *opt*

if-condition → [expression](#) | [declaration](#)

else-clause → **else** [code-block](#) | **else** [if-statement](#)

Switch 语句语法

switch-statement → **switch** [expression](#) { [switch-cases](#) *opt* }

switch-cases → [switch-case](#) [switch-cases](#) *opt*

switch-case → [case-label](#) [statements](#) | [default-label](#) [statements](#)

switch-case → [case-label](#); | [default-label](#);

case-label → **case** [case-item-list](#):

case-item-list → [pattern guard-clause](#) *opt* | [pattern guard-clause](#) *opt*, [case-item-list](#)

default-label → **default**:

guard-clause → **where** [guard-expression](#)

guard-expression → [expression](#)

带标签的语句语法

labeled-statement → [statement-label](#) [loop-statement](#) | [statement-label](#) [switch-statement](#)

statement-label → [label-name](#):

label-name → [identifier](#)

控制传递语句语法

control-transfer-statement → [break-statement](#)

control-transfer-statement → [continue-statement](#)

control-transfer-statement → [fallthrough-statement](#)

control-transfer-statement → [return-statement](#)

Break 语句语法

break-statement → **break** [label-name](#) *opt*

Continue 语句语法

continue-statement → **continue** [label-name](#) *opt*

Fallthrough 语句语法

fallthrough-statement → **fallthrough**

Return 语句语法

return-statement → **return** [expression](#) *opt*

泛型参数

泛型形参子句语法

generic-parameter-clause → < [generic-parameter-list](#) [requirement-clause](#) *opt* >

generic-parameter-list → [generic-parameter](#) | [generic-parameter](#), [generic-parameter-list](#)

generic-parameter → [type-name](#)

generic-parameter → [type-name](#): [type-identifier](#)

generic-parameter → [type-name](#): [protocol-composition-type](#)

requirement-clause → **where** [requirement-list](#)

requirement-list → [requirement](#) | [requirement](#), [requirement-list](#)

requirement → [conformance-requirement](#) | [same-type-requirement](#)

conformance-requirement → [type-identifier](#): [type-identifier](#)

conformance-requirement → [type-identifier](#): [protocol-composition-type](#)

same-type-requirement → [type-identifier](#) == [type-identifier](#)

泛型实参子句语法

generic-argument-clause → < [generic-argument-list](#) >

generic-argument-list → [generic-argument](#) | [generic-argument](#), [generic-argument-list](#)

generic-argument → [type](#)

声明

声明语法

declaration → [import-declaration](#)

declaration → [constant-declaration](#)

declaration → [variable-declaration](#)

declaration → [typealias-declaration](#)

declaration → [function-declaration](#)

declaration → [enum-declaration](#)

declaration → [struct-declaration](#)

declaration → [class-declaration](#)

declaration → [protocol-declaration](#)

declaration → [initializer-declaration](#)

declaration → [deinitializer-declaration](#)

declaration → [extension-declaration](#)

declaration → [subscript-declaration](#)

declaration → [operator-declaration](#)

declarations → [declaration](#) [declarations](#) opt

declaration-specifiers → [declaration-specifier](#) *declaration-specifiers* *opt*

declaration-

specifier → **class** | **mutating** | **nonmutating** | **override** | **static** | **unowned** | **unowned(safe)** | **unowned(unsafe)** | **weak**

顶级声明语法

top-level-declaration → [statements](#) *opt*

代码块语法

code-block → { [statements](#) *opt* }

Import 声明语法

import-declaration → [attributes](#) *opt* **import** [import-kind](#) *opt* [import-path](#)

import-kind → **typealias** | **struct** | **class** | **enum** | **protocol** | **var** | **func**

import-path → [import-path-identifier](#) | [import-path-identifier](#) . [import-path](#)

import-path-identifier → [identifier](#) | [operator](#)

常数声明语法

constant-declaration → [attributes](#) *opt* [declaration-specifiers](#) *opt* **let** [pattern-initializer-list](#)

pattern-initializer-list → [pattern-initializer](#) | [pattern-initializer](#) , [pattern-initializer-list](#)

pattern-initializer → [pattern](#) [initializer](#) *opt*

initializer → = [expression](#)

变量声明语法

variable-declaration → [variable-declaration-head](#) [pattern-initializer-list](#)

variable-declaration → [variable-declaration-head](#) [variable-name](#) [type-annotation](#) [code-block](#)

variable-declaration → [variable-declaration-head](#) [variable-name](#) [type-annotation](#) [getter-setter-block](#)

variable-declaration → [variable-declaration-head](#) [variable-name](#) [type-annotation](#) [getter-setter-keyword-block](#)

variable-declaration → [variable-declaration-head](#) [variable-name](#) [type-annotation](#) [initializer](#) *opt* [willSet-didSet-block](#)

variable-declaration-head → [attributes](#) *opt* [declaration-specifiers](#) *opt* **var**

variable-name → [identifier](#)

getter-setter-block → { [getter-clause](#) [setter-clause](#) *opt* }

getter-setter-block → { [setter-clause](#) [getter-clause](#) }

getter-clause → [attributes](#) *opt* **get** [code-block](#)

setter-clause → [attributes](#) *opt* **set** [setter-name](#) *opt* [code-block](#)

setter-name → ([identifier](#))

getter-setter-keyword-block → { [getter-keyword-clause](#) [setter-keyword-clause](#) opt }

getter-setter-keyword-block → { [setter-keyword-clause](#) [getter-keyword-clause](#) }

getter-keyword-clause → [attributes](#) opt **get**

setter-keyword-clause → [attributes](#) opt **set**

willSet-didSet-block → { [willSet-clause](#) [didSet-clause](#) opt }

willSet-didSet-block → { [didSet-clause](#) [willSet-clause](#) }

willSet-clause → [attributes](#) opt **willSet** [setter-name](#) opt [code-block](#)

didSet-clause → [attributes](#) opt **didSet** [setter-name](#) opt [code-block](#)

类型别名声明语法

typealias-declaration → [typealias-head](#) [typealias-assignment](#)

typealias-head → **typealias** [typealias-name](#)

typealias-name → [identifier](#)

typealias-assignment → = [type](#)

函数声明语法

function-declaration → [function-head](#) [function-name](#) [generic-parameter-clause](#) opt [function-signature](#)[function-body](#)

function-head → [attributes](#) opt [declaration-specifiers](#) opt **func**

function-name → [identifier](#) | [operator](#)

function-signature → [parameter-clauses](#) [function-result](#) opt

function-result → -> [attributes](#) opt [type](#)

function-body → [code-block](#)

parameter-clauses → [parameter-clause](#) [parameter-clauses](#) opt

parameter-clause → () | ([parameter-list](#)... opt)

parameter-list → [parameter](#) | [parameter](#) , [parameter-list](#)

parameter → **inout** opt **let** opt # opt [parameter-name](#) [local-parameter-name](#) opt [type-annotation](#) [default-argument-clause](#) opt

parameter → **inout** opt **var** # opt [parameter-name](#) [local-parameter-name](#) opt [type-annotation](#) [default-argument-clause](#) opt

parameter → [attributes](#) opt [type](#)

parameter-name → [identifier](#) | _

local-parameter-name → [identifier](#) | _

default-argument-clause → = [expression](#)

枚举声明语法

`enum-declaration` → [attributes](#) opt [union-style-enum](#) | [attributes](#) opt [raw-value-style-enum](#)

`union-style-enum` → [enum-name](#) [generic-parameter-clause](#) opt { [union-style-enum-members](#) opt }

`union-style-enum-members` → [union-style-enum-member](#) [union-style-enum-members](#) opt

`union-style-enum-member` → [declaration](#) | [union-style-enum-case-clause](#)

`union-style-enum-case-clause` → [attributes](#) opt **case** [union-style-enum-case-list](#)

`union-style-enum-case-list` → [union-style-enum-case](#) | [union-style-enum-case](#) , [union-style-enum-case-list](#)

`union-style-enum-case` → [enum-case-name](#) [tuple-type](#) opt

`enum-name` → [identifier](#)

`enum-case-name` → [identifier](#)

`raw-value-style-enum` → [enum-name](#) [generic-parameter-clause](#) opt : [type-identifier](#) { [raw-value-style-enum-members](#) opt }

`raw-value-style-enum-members` → [raw-value-style-enum-member](#) [raw-value-style-enum-members](#) opt

`raw-value-style-enum-member` → [declaration](#) | [raw-value-style-enum-case-clause](#)

`raw-value-style-enum-case-clause` → [attributes](#) opt **case** [raw-value-style-enum-case-list](#)

`raw-value-style-enum-case-list` → [raw-value-style-enum-case](#) | [raw-value-style-enum-case](#) , [raw-value-style-enum-case-list](#)

`raw-value-style-enum-case` → [enum-case-name](#) [raw-value-assignment](#) opt

`raw-value-assignment` → = [literal](#)

结构体声明语法

`struct-declaration` → [attributes](#) opt **struct** [struct-name](#) [generic-parameter-clause](#) opt [type-inheritance-clause](#) opt [struct-body](#)

`struct-name` → [identifier](#)

`struct-body` → { [declarations](#) opt }

类声明语法

`class-declaration` → [attributes](#) opt **class** [class-name](#) [generic-parameter-clause](#) opt [type-inheritance-clause](#) opt [class-body](#)

`class-name` → [identifier](#)

`class-body` → { [declarations](#) opt }

协议声明语法

`protocol-declaration` → [attributes](#) opt **protocol** [protocol-name](#) [type-inheritance-clause](#) opt [protocol-body](#)

protocol-name → [identifier](#)

protocol-body → { [protocol-member-declarations](#) opt }

protocol-member-declaration → [protocol-property-declaration](#)

protocol-member-declaration → [protocol-method-declaration](#)

protocol-member-declaration → [protocol-initializer-declaration](#)

protocol-member-declaration → [protocol-subscript-declaration](#)

protocol-member-declaration → [protocol-associated-type-declaration](#)

protocol-member-declarations → [protocol-member-declaration](#) [protocol-member-declarations](#) opt

协议属性声明语法

protocol-property-declaration → [variable-declaration-head](#) [variable-name](#) [type-annotation](#) [getter-setter-keyword-block](#)

协议方法声明语法

protocol-method-declaration → [function-head](#) [function-name](#) [generic-parameter-clause](#) opt [function-signature](#)

协议构造函数声明语法

protocol-initializer-declaration → [initializer-head](#) [generic-parameter-clause](#) opt [parameter-clause](#)

协议附属脚本声明语法

protocol-subscript-declaration → [subscript-head](#) [subscript-result](#) [getter-setter-keyword-block](#)

协议关联类型声明语法

protocol-associated-type-declaration → [typealias-head](#) [type-inheritance-clause](#) opt [typealias-assignment](#) opt

构造函数声明语法

initializer-declaration → [initializer-head](#) [generic-parameter-clause](#) opt [parameter-clause](#) [initializer-body](#)

initializer-head → [attributes](#) opt **convenience** opt **init**

initializer-body → [code-block](#)

析构函数声明语法

deinitializer-declaration → [attributes](#) opt **deinit** [code-block](#)

扩展声明语法

extension-declaration → **extension** [type-identifier](#) [type-inheritance-clause](#) opt [extension-body](#)

extension-body → { [declarations](#) opt }

附属脚本声明语法

subscript-declaration → [subscript-head](#) [subscript-result](#) [code-block](#)

subscript-declaration → [subscript-head](#) [subscript-result](#) [getter-setter-block](#)

subscript-declaration → [subscript-head](#) [subscript-result](#) [getter-setter-keyword-block](#)

subscript-head → [attributes](#) *opt* **subscript** [parameter-clause](#)

subscript-result → **->** [attributes](#) *opt* [type](#)

运算符声明语法

operator-declaration → [prefix-operator-declaration](#) | [postfix-operator-declaration](#) | [infix-operator-declaration](#)

prefix-operator-declaration → **operator** **prefix** [operator](#) { }

postfix-operator-declaration → **operator** **postfix** [operator](#) { }

infix-operator-declaration → **operator** **infix** [operator](#) { [infix-operator-attributes](#) *opt* }

infix-operator-attributes → [precedence-clause](#) *opt* [associativity-clause](#) *opt*

precedence-clause → **precedence** [precedence-level](#)

precedence-level → Digit 0 through 255

associativity-clause → **associativity** [associativity](#)

associativity → **left** | **right** | **none**

模式

模式语法

pattern → [wildcard-pattern](#) [type-annotation](#) *opt*

pattern → [identifier-pattern](#) [type-annotation](#) *opt*

pattern → [value-binding-pattern](#)

pattern → [tuple-pattern](#) [type-annotation](#) *opt*

pattern → [enum-case-pattern](#)

pattern → [type-casting-pattern](#)

pattern → [expression-pattern](#)

通配符模式语法

wildcard-pattern → **_**

标识符模式语法

identifier-pattern → [identifier](#)

值绑定模式语法

value-binding-pattern → **var** [pattern](#) | **let** [pattern](#)

元组模式语法

tuple-pattern → ([tuple-pattern-element-list](#) *opt*)

tuple-pattern-element-list → [tuple-pattern-element](#) | [tuple-pattern-element](#) , [tuple-pattern-element-list](#)

tuple-pattern-element → [pattern](#)

枚举用例模式语法

enum-case-pattern → [type-identifier](#) *opt* . [enum-case-name](#) [tuple-pattern](#) *opt*

类型转换模式语法

type-casting-pattern → [is-pattern](#) | [as-pattern](#)

is-pattern → **is** [type](#)

as-pattern → [pattern](#) **as** [type](#)

表达式模式语法

expression-pattern → [expression](#)

特性

特性语法

attribute → @ [attribute-name](#) [attribute-argument-clause](#) *opt*

attribute-name → [identifier](#)

attribute-argument-clause → ([balanced-tokens](#) *opt*)

attributes → [attribute](#) [attributes](#) *opt*

balanced-tokens → [balanced-token](#) [balanced-tokens](#) *opt*

balanced-token → ([balanced-tokens](#) *opt*)

balanced-token → [[balanced-tokens](#) *opt*]

balanced-token → { [balanced-tokens](#) *opt* }

balanced-token → Any identifier, keyword, literal, or operator

balanced-token → Any punctuation except (,), [,], {, or }

表达式

表达式语法

expression → [prefix-expression](#) [binary-expressions](#) *opt*

expression-list → [expression](#) | [expression](#) , [expression-list](#)

前缀表达式语法

prefix-expression → [prefix-operator](#) opt [postfix-expression](#)

prefix-expression → [in-out-expression](#)

in-out-expression → & [identifier](#)

二进制表达式语法

binary-expression → [binary-operator](#) [prefix-expression](#)

binary-expression → [assignment-operator](#) [prefix-expression](#)

binary-expression → [conditional-operator](#) [prefix-expression](#)

binary-expression → [type-casting-operator](#)

binary-expressions → [binary-expression](#) [binary-expressions](#) opt

赋值运算符语法

assignment-operator → =

条件运算符语法

conditional-operator → ? [expression](#) :

类型转换运算符语法

type-casting-operator → is [type](#) | as ? opt [type](#)

主表达式语法

primary-expression → [identifier](#) [generic-argument-clause](#) opt

primary-expression → [literal-expression](#)

primary-expression → [self-expression](#)

primary-expression → [superclass-expression](#)

primary-expression → [closure-expression](#)

primary-expression → [parenthesized-expression](#)

primary-expression → [implicit-member-expression](#)

primary-expression → [wildcard-expression](#)

字面量表达式语法

literal-expression → [literal](#)

literal-expression → [array-literal](#) | [dictionary-literal](#)

literal-expression → __FILE__ | __LINE__ | __COLUMN__ | __FUNCTION__

array-literal → [[array-literal-items](#) opt]

array-literal-items → [array-literal-item](#) , opt | [array-literal-item](#) , [array-literal-items](#)

array-literal-item → [expression](#)

dictionary-literal → [[dictionary-literal-items](#)] | [:]

dictionary-literal-items → [dictionary-literal-item](#), opt | [dictionary-literal-item](#), [dictionary-literal-items](#)

dictionary-literal-item → [expression](#) : [expression](#)

Self 表达式语法

self-expression → **self**

self-expression → **self** . [identifier](#)

self-expression → **self** [[expression](#)]

self-expression → **self** . **init**

超类表达式语法

superclass-expression → [superclass-method-expression](#) | [superclass-subscript-expression](#) | [superclass-initializer-expression](#)

superclass-method-expression → **super** . [identifier](#)

superclass-subscript-expression → **super** [[expression](#)]

superclass-initializer-expression → **super** . **init**

闭包表达式语法

closure-expression → { [closure-signature](#) opt [statements](#) }

closure-signature → [parameter-clause](#) [function-result](#) opt **in**

closure-signature → [identifier-list](#) [function-result](#) opt **in**

closure-signature → [capture-list](#) [parameter-clause](#) [function-result](#) opt **in**

closure-signature → [capture-list](#) [identifier-list](#) [function-result](#) opt **in**

closure-signature → [capture-list](#) **in**

capture-list → [[capture-specifier](#) [expression](#)]

capture-specifier → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

隐式成员表达式语法

implicit-member-expression → . [identifier](#)

带圆括号的表达式语法

parenthesized-expression → ([expression-element-list](#) opt)

expression-element-list → [expression-element](#) | [expression-element](#), [expression-element-list](#)

expression-element → [expression](#) | [identifier](#) : [expression](#)

通配符表达式语法

wildcard-expression → _

后缀表达式语法

postfix-expression → [primary-expression](#)

postfix-expression → [postfix-expression postfix-operator](#)

postfix-expression → [function-call-expression](#)

postfix-expression → [initializer-expression](#)

postfix-expression → [explicit-member-expression](#)

postfix-expression → [postfix-self-expression](#)

postfix-expression → [dynamic-type-expression](#)

postfix-expression → [subscript-expression](#)

postfix-expression → [forced-value-expression](#)

postfix-expression → [optional-chaining-expression](#)

函数调用表达式语法

function-call-expression → [postfix-expression parenthesized-expression](#)

function-call-expression → [postfix-expression parenthesized-expression](#) opt [trailing-closure](#)

trailing-closure → [closure-expression](#)

初始化表达式语法

initializer-expression → [postfix-expression](#) . **init**

显式成员表达式语法

explicit-member-expression → [postfix-expression](#) . [decimal-digit](#)

explicit-member-expression → [postfix-expression](#) . [identifier generic-argument-clause](#) opt

Self 表达式语法

postfix-self-expression → [postfix-expression](#) . **self**

动态类型表达式语法

dynamic-type-expression → [postfix-expression](#) . **dynamicType**

附属脚本表达式语法

subscript-expression → [postfix-expression](#) [[expression-list](#)]

Forced-value 表达式语法

forced-value-expression → [postfix-expression](#) !

可选链表达式语法

optional-chaining-expression → [postfix-expression](#) ?

词法结构

标识符语法

identifier → [identifier-head identifier-characters opt](#)

identifier → ` [identifier-head identifier-characters opt](#) `

identifier → [implicit-parameter-name](#)

identifier-list → [identifier](#) | [identifier](#) , [identifier-list](#)

identifier-head → Upper- or lowercase letter A through Z

identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA

identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF

identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF

identifier-head → U+1E00–U+1FFF

identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F

identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793

identifier-head → U+2C00–U+2DFF or U+2E80–U+2FFF

identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF

identifier-head → U+F900–U+FD3D, U+FD40–U+FDCE, U+FDFO–U+FE1F, or U+FE30–U+FE44

identifier-head → U+FE47–U+FFFD

identifier-head → U+10000–U+1FFFFD, U+20000–U+2FFFFD, U+30000–U+3FFFFD, or U+40000–U+4FFFFD

identifier-head → U+50000–U+5FFFFD, U+60000–U+6FFFFD, U+70000–U+7FFFFD, or U+80000–U+8FFFFD

identifier-head → U+90000–U+9FFFFD, U+A0000–U+AFFFFD, U+B0000–U+BFFFFD, or U+C0000–U+CFFFFD

identifier-head → U+D0000–U+DFFFFD or U+E0000–U+EFFFFD

identifier-character → Digit 0 through 9

identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F

identifier-character → [identifier-head](#)

identifier-characters → [identifier-character](#) [identifier-characters](#) *opt*

implicit-parameter-name → \$ [decimal-digits](#)

字面量语法

literal → [integer-literal](#) | [floating-point-literal](#) | [string-literal](#)

整形字面量语法

integer-literal → [binary-literal](#)

integer-literal → [octal-literal](#)

integer-literal → [decimal-literal](#)

integer-literal → [hexadecimal-literal](#)

binary-literal → **0b** [binary-digit](#) [binary-literal-characters](#) *opt*

binary-digit → Digit 0 or 1

binary-literal-character → [binary-digit](#) | `_`

binary-literal-characters → [binary-literal-character](#) [binary-literal-characters](#) *opt*

octal-literal → **0o** [octal-digit](#) [octal-literal-characters](#) *opt*

octal-digit → Digit 0 through 7

octal-literal-character → [octal-digit](#) | `_`

octal-literal-characters → [octal-literal-character](#) [octal-literal-characters](#) *opt*

decimal-literal → [decimal-digit](#) [decimal-literal-characters](#) *opt*

decimal-digit → Digit 0 through 9

decimal-digits → [decimal-digit](#) [decimal-digits](#) *opt*

decimal-literal-character → [decimal-digit](#) | `_`

decimal-literal-characters → [decimal-literal-character](#) [decimal-literal-characters](#) *opt*

hexadecimal-literal → **0x** [hexadecimal-digit](#) [hexadecimal-literal-characters](#) *opt*

hexadecimal-digit → Digit 0 through 9, a through f, or A through F

hexadecimal-literal-character → [hexadecimal-digit](#) | `_`

hexadecimal-literal-characters → [hexadecimal-literal-character](#) [hexadecimal-literal-characters](#) *opt*

浮点型字面量语法

floating-point-literal → [decimal-literal](#) [decimal-fraction](#) *opt* [decimal-exponent](#) *opt*

floating-point-literal → [hexadecimal-literal](#) [hexadecimal-fraction](#) *opt* [hexadecimal-exponent](#)

decimal-fraction → . [decimal-literal](#)

decimal-exponent → [floating-point-e sign](#) *opt* [decimal-literal](#)

hexadecimal-fraction → . [hexadecimal-literal](#) *opt*

hexadecimal-exponent → [floating-point-p sign](#) opt [hexadecimal-literal](#)

floating-point-e → **e** | **E**

floating-point-p → **p** | **P**

sign → + | -

字符型字面量语法

string-literal → " [quoted-text](#) "

quoted-text → [quoted-text-item](#) [quoted-text](#) opt

quoted-text-item → [escaped-character](#)

quoted-text-item → ([expression](#))

quoted-text-item → Any Unicode extended grapheme cluster except ", \, U+000A, or U+000D

escaped-character → \ **o** | \ | \ **t** | \ **n** | \ **r** | \" | \'

escaped-character → \ **x** [hexadecimal-digit](#) [hexadecimal-digit](#)

escaped-character → \ **u** [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#)

escaped-character → \ **U** [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#) [hexadecimal-digit](#)

运算符语法语法

operator → [operator-character](#) [operator](#) opt

operator-character → / | = | - | + | ! | * | % | < | > | & | | | ^ | ~ | .

binary-operator → [operator](#)

prefix-operator → [operator](#)

postfix-operator → [operator](#)

类型

类型语法

type → [array-type](#) | [function-type](#) | [type-identifier](#) | [tuple-type](#) | [optional-type](#) | [implicitly-unwrapped-optional-type](#) | [protocol-composition-type](#) | [metatype-type](#)

类型标注语法

type-annotation → : [attributes](#) opt [type](#)

类型标识语法

type-identifier → [type-name](#) [generic-argument-clause](#) opt | [type-name](#) [generic-argument-clause](#) opt . [type-identifier](#)

type-name → [identifier](#)

元组类型语法

tuple-type → ([tuple-type-body](#) *opt*)

tuple-type-body → [tuple-type-element-list](#)... *opt*

tuple-type-element-list → [tuple-type-element](#) | [tuple-type-element](#) , [tuple-type-element-list](#)

tuple-type-element → [attributes](#) *opt* **inout** *opt* [type](#) | **inout** *opt* [element-name](#) [type-annotation](#)

element-name → [identifier](#)

函数类型语法

function-type → [type](#) -> [type](#)

数组类型语法

array-type → [type](#) [] | [array-type](#) []

可选类型语法

optional-type → [type](#) ?

隐式解析可选类型语法

implicitly-unwrapped-optional-type → [type](#) !

协议合成类型语法

protocol-composition-type → **protocol** < [protocol-identifier-list](#) *opt* >

protocol-identifier-list → [protocol-identifier](#) | [protocol-identifier](#) , [protocol-identifier-list](#)

protocol-identifier → [type-identifier](#)

元类型语法

metatype-type → [type](#) . **Type** | [type](#) . **Protocol**

类型继承子句语法

type-inheritance-clause → : [type-inheritance-list](#)

type-inheritance-list → [type-identifier](#) | [type-identifier](#) , [type-inheritance-list](#)