The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（一）基础类型

虽然 Swift 是一个为开发 iOS 和 OS X app 设计的全新编程语言，但是 Swift 的很多特性还是跟 C 和 Objective-C 相似。

Swift 也提供了与 C 和 Objective-C 类似的基础数据类型，包括整形 Int、浮点数 Double 和 Float、布尔类型 Bool 以及字符串类型 String。Swift 还提供了两种更强大的基本集合数据类型，Array 和 Dictionary，更详细的内容可以参考：Collection Types。

跟 C 语言一样，Swift 使用特定的名称来定义和使用变量。同样，Swift 中也可以定义常量，与 C 语言不同的是，Swift 中的常量更加强大，在编程时使用常量能够让代码看起来更加安全和简洁。

除了常见的数据类型之外，Swift 还集成了 Objective-C 中所没有的 “元组” 类型，可以作为一个整体被传递。元组也可以成为一个函数的返回值，从而允许函数一次返回多个值。

Swift 还提供了可选类型，用来处理一些未知的不存在的值。可选类型的意思是：这个值要么存在，并且等于 x，要么根本不存在。可选类型类似于 Objective-C 中指针的 nil 值，但是 nil 只对类 (class) 有用，而可选类型对所有的类型都可用，并且更安全。可选类型是大部分 Swift 新特性的核心。

可选性类型只是 Swift 作为类型安全的编程语言的一个例子。Swift 可以帮助你更快地发现编码中的类型错误。如果你的代码期望传递的参数类型是 String 的，那么类型安全就会防止你错误地传递一个 Int 值。这样就可以让编程人员在开发期更快地发现和修复问题。

1.1 常量和变量

常量和变量由一个特定名称来表示，如 maximumNumberOfLoginAttempt 或者 welcomeMessage。常量所指向的是一个特定类型的值，如数字 10 或者字符 “hello”。变量的值可以根据需要不断修改，而常量的值是不能够被二次修改的。

1.1.1 常量和变量的声明

常量和变量在使用前都需要声明，在 Swift 中使用 let 关键词来声明一个常量，var 关键词声明一个变量。如下面例子

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

以上代码可以理解为：

声明一个叫 maximumNumberOfLoginAttempts 的值为 10 的常量。然后声明一个变量 currentLoginAttempt 初始值为 0。

在这个例子中，最大的登录尝试次数 10 是不变的，因此声明为常量。而已经登录的尝试次数是可变的，因此定义为变量。也可以在一行中声明多个变量或常量，用, 号分隔：

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

> 注：如果一个值在之后的代码中不会再变化，应该用 let 关键词将它声明为常量。变量只用来存储会更改的值。

1.1.2 类型注解

在声明常量和变量时，可以使用注解来注明该变量或常量的类型。使用: 号加空格加类型名在变量或常量名之后就可以完成类型注解。下面的例子就是声明了一个变量叫 welcomeMessage，注解类型为字符串 String：

```
var welcomeMessage: String
```

分号 “:” 在这的作用就像是在说：...是...类型的，因此上述代码可以理解为：

声明一个叫 welcomeMessage 的变量，它的类型是 String

这个类型注解表明 welcomeMessage 变量能无误地存储任何字符串类型的值，比如 welcomeMessage = “hello”

> 注：实际编程中很少需要使用类型注解，定义常量或者变量的时候 Swift 已经根据初始化的值确定了类型信息。Swift 几乎都可以隐式的确定变量或常量的类型，详见：Type Safety and Type Inference。而上面的 welcomeMessage 的例子中，初始化值没有被给出，所以更好的办法是指定 welcomeMessage 变量的类型而不是让 Swift 隐式推导类型。

1.1.3 常量和变量的命名

Swift 中可以使用几乎任何字符来作为常量和变量名，包括 Unicode，比如：

```
let   = 3.14159
let □ □ = "□ □ □ □ "
let = "dogcow"
```

但是名称中不能含有数学符号，箭头，无效的 Unicode，横线 -和制表符，且不能以数字开头，尽管数字可以包含在名称里。一旦完成了声明，就不能再次声明相同名称的变量或常量，或者改变它的类型。变量和常量也不能互换。

> 注：如果你想用 Swift 保留字命名一个常量或者变量，你可以用 ‘ 符号把命名包围起来。尽管如此，除非处于特别的意图，尽量不要使用保留字作为变量/常量名。

可以改变变量的值为它声明的类型的其它值，如下的例子里，变量 friendlyWelcome 的值从 “Hello!” 被修改为”Bonjour¡‘:

```
var friendlyWelcome = “ hello! ”
friendlyWelcome = “ Bonjour! ”
// friendlyWelcome is now “ Bonjour! ”
```

与变量不同的是，常量的值一旦确定就不能修改。如果想尝试改变一个常量的值，编译代码时就会报错

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

1.1.4 输出常量和变量

Swift 使用 println 来输出变量或者常量:

```
println(friendlyWelcome)
// prints “ Bonjour! ”
```

println 是一个全局函数，用来输出一个值，最后输出一个换行。在 Xcode 中，println 输出在控制台中。print 函数也类似，只不过最后不会输出换行。

println 函数一般输出一个字符串

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

println 函数还可以格式化输出一些日志信息，就像是 Cocoa 中 NSLog 函数的行为一样，可以包括一些常量和变量本身。Swift 在字符串中插入变量名作为占位符，使用反斜杠 \ 和小括号 () 来提示 Swift 替换变量/常量名为其实际的值，如：

println(“The current value of friendlyWelcome is (friendlyWelcome)”) // prints “The current value of friendlyWelcome is Bonjour!”

注：关于格式化字符的详见 String Interpolation

1.2 注释

不参与编译的语句称为注释，注释可以提示你代码的意图。Swift 中的注释和 C 语言中的一样，有单行注释

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

和多行注释，使用/*和*/分隔

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

和 C 语言不同的是，多行注释可以嵌套，你需要先开始一个多行注释，然后开始第二个多行注释，关闭注释的时候先关闭第二个，然后是第一个。如下

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

这样可以方便地在大段已注释的代码块中继续添加注释

1.3 分号

和其它一些编程语言不同，Swift 不需要使用分号; 来分隔每一个语句。当然你也可以选择使用分号，或者你想在一行中书写多个语句。

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

1.4 整数

整数就是像 42 和 -23 这样不带分数的数字，包括有符号（正数，负数，0）和无符号（正数，0）。Swift 提供了 8、16、32 和 64 位的数字形式，和 C 语言类似，可以使用 8 位的无符号整数 UInt8，或者 32 位的整数 Int32. 像其他 Swift 类型一样，这些类型名的首字母大写。

1.4.1 整数边界

使用 min 或 max 值来获取该类型的最大最小值，如：

```
let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8
```

这些值边界值区分了整数的类型（比如 UInt8），所以可以像该类型的其他值一样被用在表达式中而不用考虑益处的问题。

1.4.2 Int 类型

一般来说，编程人员在写代码时不需要选择整数的位数，Swift 提供了一种额外的整数类型 Int，是和当前机器环境的字长相同的整数位数

- 在 32 位机器上，Int 和 Int32 一样大小
- 在 64 位机器上，Int 和 Int64 一样大小

除非你确实需要使用特定字长的正数，尽量使用 Int 类型。这保证了代码的可移植性。即使在 32 位的平台上，Int 也可以存储 -2,147,483,648 到 2,147,483,647 范围内的值，这对大部分正数来讲已经足够了。

1.4.3 UInt 类型

Swift 还提供了一种无符号类型 UInt，同理也是和当前机器环境的字长相等。

- 在 32 位机器上，UInt 和 UInt32 一样大小
- 在 64 位机器上，UInt 和 UInt64 一样大小

> 注：只有显式的需要指定一个长度跟机器字长相等的无符号数的时候才需要使用 UInt，其他的情况，尽量使用 Int，即使这个变量确定是无符号的。都使用 Int 保证了代码的可移植性，避免了不同数字类型之间的转换。详见Type Safety and Type Inference.

5、浮点数

浮点数就是像 3.14159，0.1，-273.15 这样带分数的数字。浮点数可以表达比 Int 范围更广（更大或更小）的数值。swift 支持两种带符号浮点数类型：

- Double 类型能表示 64 位的有符号浮点数。当需要表的数字非常大或者精度要求非常高的时候可以使用 Double 类型。
- Float 类型能表示 32 为的有符号浮点数。当需要表达的数字不需要 64 位精度的时候可以使用 Float 类型。
  注 Double 至少有 15 位小数，Float 至少有 6 位小数。合适的浮点数小数位数取决于你代码里需要处理的浮点数范围。

6、类型安全和类型推导

Swift 是一种类型安全的语言。类型安全就是说在编程的时候需要弄清楚变量的类型。如果您的代码部分需要一个字符串，你不能错误地传递一个整数类型。

因为 Swift 是类型安全的，它会在编译的时候就检查你的代码，任何类型不匹配时都会报错。这使得编程人员能够尽快捕获并尽可能早地在开发过程中修正错误。

类型检查可以在使用不同类型的值时帮助避免错误。但是，这并不意味着你必须指定每一个常量和变量所声明的类型。如果不指定你需要的类型，Swift 使用类型推导来指定出相应的类型。类型推导使编译器在编译的时候通过你提供的初始化值自动推导出特定的表达式的类型。

类型推导使 Swift 比起 C 或 Objective-C 只需要更少的类型声明语句。常量和变量仍然显式类型，但大部分指定其类型的工作 Swift 已经为你完成了。

当你声明一个常量或变量并给出初始值类型的时候，类型推导就显得特别有用。这通常是通过给所声明的常量或变量赋常值来完成的。(常值是直接出现在源代码中的值，如下面的例子 42 和 3.14159 。)

例如，如果您指定 42 到一个新的常数变量，而不用说它是什么类型，Swift 推断出你想要的常量是一个整数，因为你已经初始化它为一个整数

```
let meaningOfLife= 42
// meaningOfLife is inferred to be of typeInt
```

同样，如果你不指定浮点值的类型，Swift 推断出你想要创建一个 Double：

```
let pi = 3.14159
// pi is inferred to be of type Double
```

Swift 总是选择 Double（而非 Float）当它需要浮点数类型时。如果你在一个表达式中把整数和浮点数相加，会推导一个 Double 类型：

```
let anotherPi= 3 + 0.14159
// anotherPi is also inferred to be of typeDouble
```

常值 3 没有显示指明类型，所以 Swift 根据表达式中的浮点值推出输出类型 Double。

1.5 数值量表达

整型常量可以写成：

- 一个十进制数，不带前缀
- 一个二进制数，用前缀 0b
- 一个八进制数，用 0o 前缀
- 一个十六进制数，以 0x 前缀

所有如下用这些整型常量都可以来表达十进制值的 17：

```
let decimalInteger= 17
let binaryInteger = 0b10001 // 17 in binary notation
let octalInteger = 0o21 // 17 in octal notation
let hexadecimalInteger = 0x11 // 17 inhexadecimal notation
```

浮点可以是十进制（不带前缀）或十六进制（以 0x 前缀），小数点的两侧必须始终有一个十进制数（或十六进制数）。他们也可以有一个可选的指数，由一个大写或小写 e 表示十进制浮点数表示，或大写/小写 p 表示十六进制浮点数

带指数 exp 的十进制数，实际值等于基数乘以 10 的 exp 次方，如：

- 1.25e2 表示 1.25×102, 或者 125.0.
- 1.25e-2 表示 1.25×10-2, 或者 0.0125.

带指数 exp 的十六进制数，实际值等于基部数乘以 2 的 exp 次方，如：

- 0xFp2 表示 15×22, 或者 60.0.
- 0xFp-2 表示 15×2-2, 或者 3.75.

所有如下这些浮点常量都表示十进制的 12.1875：

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

数字值可以包含额外的格式，使它们更容易阅读。整数和浮点数都可以被额外的零填充，并且可以包含下划线，以增加可读性。以上格式都不影响变量的值:

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

1.6 数值类型转换

为代码中所有通用的数值型整型常量和变量使用 Int 类型，即使它们已知是非负的。这意味着代码中的数值常量和变量能够相互兼容并且能够与自动推导出的类型相互匹配。

只有因为某些原因（性能，内存占用或者其他必须的优化）确实需要使用其他数值类型的时候，才应该使用这些数值类型。这些情况下使用显式指定长度的类型有助于发现值范围溢出，同时应该留下文档。

1.6.1 整数转换

可以存储在一个整数常量或变量的范围根据每个数值类型是不同的。一个 Int8 常量或变量可以存储范围 -128 到 127 之间的数，而一个 UInt8 常量或变量可以存储 0 到 255 之间的数字。错误的赋值会让编译器报错:

```
let cannotBeNegative: UInt8 = -1
// UInt8 cannot store negative numbers, and so this will report an error
let tooBig: Int8 = Int8.max + 1
// Int8 cannot store a number larger thanits maximum value,
// and so this will also report an error
```

因为每个数字类型可以存储不同范围的值，你必须在基础数值类型上逐步做转换。这种可以防止隐藏的转换错误，并帮助明确你的代码中类型转换的意图。

要转换一个特定的数字类型到另一个，你需要定义一个所需类型的新变量，并用当前值初始化它。在下面的例子中，常量 twoThousand 是 UInt16 类型的，

而常量 one 是 UINT8 类型的。它们不能被直接相加的，因为类型不同。相反的，该 示例调用 UInt16(one) 来创建一个用变量 one 的值初始化的 UInt16 类型的新变量，并且使用这个值来代替原来的值参与运算：

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

可以由于相加双方的类型都是 UInt16 的，现在可以做加法运算了。输出常量（twoThousandAndOne）被推断为 UInt16 类型的，因为它是两个 UInt16 的值的总和。

SomeType(ofInitialValue) 是 Swift 默认的类型转换方式。实现上看，UInt16 的有一个接受 UINT8 值的构造器，这个构造器用于从现有 UInt8 构造出一个新的 UInt16 的变量。你不能传入任意类型的参数，它必须是一个类型的 UInt16 初始化能接受的类型。如何扩展现有类型，规定接受新的类型（包括你自己的类型定义）可以参见 Extensions。

1.6.2 整数和浮点数转换

整数和浮点类型之间的转化必须显式声明：

```
let three = 3
let pointOneFourOneFiveNine= 0.14159
let pi = Double(three) +pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to beof typde Double
```

这里，常量 three 的值被用来创建 Double 类型的新变量，从而使表达式两侧是相同的类型。如果没有这个转换，加法操作不会被允许。反之亦然，一个整数类型可以用 double 或 float 值进行初始化：

```
let integerPi= Int(pi)
// integerPi equals 3, and is inferred tobe of type Int
```

当使用这种方式初始化一个新的整数值的时候，浮点值总是被截断。这意味着，4.75 变为 4，和 -3.9 变为 -3。

> 注：数值类型常量/变量的类型转换规则和数字类型常值的转换规则不同。常值 3 可以直接与常值 0.14159 相加，因为常值没有一个明确的类型。他们的类型是被编译器推导出来的。

1.7 类型别名

类型别名为现有类型定义的可替代名称。你可以使用 typealias 关键字定义类型别名。类型别名可以帮助你使用更符合上下文语境的名字来指代一个已存在的类型，比如处理一个外来的有指定长度的类型的时候：

```
typealias AudioSample = UInt16
```

一旦你定义了一个类型别名，你可以在任何可能使用原来的名称地方使用别名：

```
var maxAmplitudeFound= AudioSample.min
// maxAmplitudeFound is now 0
```

这里，AudioSample 被定义为一个 UInt16 的别名。因为它是一个别名，调用 AudioSample.min 实际上是调用 UInt16.min，从而给 maxAmplitudeFound 变量赋初始值 0。

1.8 布尔类型

Swift 中的布尔类型使用 Bool 定义，也被称为 Logical (逻辑) 类型，可选值是 true 和 false：

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

这里 orangesAreOrange 和 turnipsAreDelicious 的类型被推导为 Bool 因为他们被初始化被 Bool 类型的常值。跟 Int 和 Double 类型一样，在定义布尔类型的时候不需要显式的给出数据类型，只需要直接赋值为 true 或 false 即可。当使用确定类型的常值初始化一个常量/变量的时候，类型推导使 Swift 代码更精确和可读。布尔类型在条件语句中特别适用，比如在 if 语句中

```
if turnipsAreDelicious {
    println("Mmm, tasty turnips!")
} else {
    println("Eww, turnips are horrible.")
}
// prints "Eww, turnips are horrible."
```

像 if 语句这样的条件语句，我们会在之后的章节ControlFlow有详细介绍。Swift 的类型安全策略会防止其他非布尔类型转换为布尔类型使用，比如

```
let i = 1
if i {
    // this example will not compile, and will report an error
}
```

就会报错，但这在其他编程语言中是可行的。但是如下的定义是正确的：

```
let i = 1
if i == 1 {
    // this example will compile successfully
}
```

i == 1 的结果就是一个布尔类型，所以可以通过类型检查。像 i==1 这种比较将会在章节 [Basic Operators] 中讨论。上面的例子也是一个 Swift 类型安全的例子。类型安全避免了偶然的类型错误，保证了代码的意图是明确的。

1.9 元组类型

元组类型可以将一些不同的数据类型组装成一个元素，这些数据类型可以是任意类型，并且不需要是同样的类型。

在下面的例子中，(404, “Not Found”) 是一个 HTTP 状态码。HTTP 状态码是请求网页的时候返回的一种特定的状态编码。404 错误的具体含义是页面未找到。

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

这个元组由一个 Int 和一个字符串 String 组成，这样的组合即包含了数字，也包含了便于人们认知的字符串描述。这个元组可以描述为类型 (Int,String) 的元组。

编程人员可以随意地创建自己需要的元组类型，比如 (Int, Int, Int), 或者 (String, Bool) 等。同时组成元组的类型数量也是不限的。可以通过如下方式分别访问一个元组的值：

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// prints "The status code is 404"
println("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

如果仅需要元组中的个别值，可以使用 (_) 来忽略不需要的值

```
let (justTheStatusCode, _) = http404Error
println("The status code is \(justTheStatusCode)")
// prints "The status code is 404"
```

另外，也可以使用元素序号来选择元组中的值，注意序号是从 0 开始的

```
println("The status code is \(http404Error.0)")
// prints "The status code is 404"
println("The status message is \(http404Error.1)")
// prints "The status message is Not Found"
```

在创建一个元组的时候，也可以直接指定每个元素的名称，然后直接使用元组名. 元素名访问，如:

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

元组类型在作为函数返回值的时候特别适用，可以为函数返回更多的用户需要的信息。比如一个请求 web 页面的函数可以返回 (Int,String) 类型的元组来表征页面获取的成功或者失败。返回两个不同类型组成的元组可以比只返回一个类型的一个值提供更多的返回信息。详见Functions with Multiple Return Values

1.10 可选类型

在一个值可能不存在的时候，可以使用可选类型。这种类型的定义是：要么存在这个值，且等于 x，要么在这个值不存在。

> 注：这种类型在 C 和 Objective-C 中是不存在的，但是 Objective-C 中有一个相似的类型，叫 nil，但是仅仅对对象有用。对其他的情况，Object-C 方法返回一个特殊值（比如 NSNotFound）来表明这个值不存在。这种方式假设方法调用者知道这个特殊值的存在和含义。Swift 的可选类型帮助你定义任意的值不存在的情况。

下面给出一个例子，在 Swift 中 String 类型有一个叫 toInt 的方法，能够将一个字符串转换为一个 Int 类型。但是需要注意的是，不是所有的字符串都可以转换为整数。比如字符串 “123” 可以转换为 123，但是 “hello, world” 就不能被转换。

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt()
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

由于 toInt 方法可能会失败，因此它会返回一个可选的 Int 类型，而不同于 Int 类型。一个可选的 Int 类型被记为 Int?，不是 Int。问号表明它的值是可选的，可能返回的是一个 Int，或者返回的值不存在。

1.10.1 if 语句和强制解包

编程人员可以使用 if 语句来检测一个可选类型是否包含一个特定的值，如果一个可选类型确实包含一个值，在 if 语句中它将返回 true，否则返回 false。如果你已经检测确认该值存在，那么可以使用或者输出它，在输出的时候只需要在名称后面加上感叹号 (!) 即可，意思是告诉编译器：我已经检测好这个值了，可以使用它了。如：

```
if convertedNumber {
   println("\(possibleNumber) has an integer value of \(convertedNumber!)")
} else {
    println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

像 if 语句这样的条件语句，我们会在之后的章节ControlFlow有详细介绍。

1.10.2 选择绑定

选择绑定帮助确定一个可选值是不是包含了一个值，如果包含，把该值转化成一个临时常量或者变量。选择绑定可以用在 if 或 while 语句中，用来在可选类型外部检查是否有值并提取可能的值。if 和 while 语句详见ControlFlow。

方法如下：

```
if let constantName = someOptional {
    statements
}
```

那么上一个例子也可以改写为：

```
if let actualNumber = possibleNumber.toInt() {
   println("\(possibleNumber) has an integer value of \(actualNumber)")
} else {
    println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

上述代码理解起来不难：如果 possibleNumber.toInt 返回的这个可选 Int 类型包含一个值，那么定义一个常量 actualNumber 来等于这个值，并在后续代码中直接使用。

如果转换是成功的，那么 actualNumber 常量在 if 的第一个分支可用，并且被初始化为可选类型包含的值，同时也不需要使用！前缀。这个例子里，actualNumber 只是简单的被用来打印结果。

常量和变量都可以用来做可选绑定。如果你想要在 if 第一个分支修改 actualNumber 的值，可以写成 if var actualNumber，actualNumber 就成为一个变量从而可以被修改。

1.10.3 nil

可以给可选类型指定一个特殊的值 nil：

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

如果你定义了一个可选类型并且没有给予初始值的时候，会默认设置为 nil

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

> 注: Swift 的 nil 不同于 Object-C 中的 nil. Object-C 中，nil 是一个指针指向不存在的对象。Swift 中，nil 不是指针而是一个特定类型的空值。任何类型的可选变量都可以被设为 nil，不光是指针。

1.10.4 隐式解包可选类型

在上面的例子中，可选类型表示一个常量/变量可以没有值。可选类型可以被 if 语句检测是否有值，并且可以被可选绑定解包。

但是在一些情况下，可选类型是一直有效的，那么可以通过定义来隐式地去掉类型检查，强制使用可选类型。这些可选类型被成为隐式解包的可选类型。你可以直接在类型后面加! 而不是？来指定。

隐式解包的可选类型主要用在一个变量/常量在定义瞬间完成之后值一定会存在的情况。这主要用在类的初始化过程中，详见Unowned References and Implicitly Unwrapped Optional Properties.

隐式解包的可选类型本质是可选类型，但是可以被当成一般类型来使用，不需要每次验证值是否存在。如下的例子展示了可选类型和解包可选类型之间的区别。

```
let possibleString: String? = "An optional string."
println(possibleString!) // requires an exclamation mark to access its value
// prints "An optional string."

let assumedString: String! = "An implicitly unwrapped optional string."
println(assumedString) // no exclamation mark is needed to access its value
// prints "An implicitly unwrapped optional string."
```

你可以把隐式解包可选类型当成对每次使用的时候自动解包的可选类型。即不是每次使用的时候在变量/常量后面加！而是直接在定义的时候加。

> 注：如果一个隐式解包的可选类型不包含一个实际值，那么对它的访问会抛出一个运行时错误。在变量/常量名后面加！的情况也是一样的。

你依然可以把解包可选类型当成正常的可选类型来探测是否有值。

```
if assumedString {
    println(assumedString)
}
// prints "An implicitly unwrapped optional string."
```

或者通过选择绑定检查

```
if let definiteString = assumedString {
    println(definiteString)
}
// prints "An implicitly unwrapped optional string."
```

> 注：如果一个可选类型存在没有值的可能的话，不应该使用解包可选类型。这种情况下，一定要使用正常的可选类型。

1.11 断言

可选类型让编程人员可以在运行期检测一个值是否存在，然后使用代码来处理不存在的情况。但是有些情况下，如果一个值不存在或者值不满足条件会直接影响代码的执行，这个时候就需要使用断言。这种情况下，断言结束程序的执行从而提供调试的依据。

1.11.1 使用断言调试

断言是一种实时检测条件是否为 true 的方法，也就是说，断言假定条件为 true。断言保证了后续代码的执行依赖于条件的成立。如果条件满足，那么代码继续执行，如果这个条件为 false，那么代码将会中断执行。

在 Xcode 中，在调试的时候如果中断，可以通过查看调试语句来查看断言失败时的程序状态。断言也能提供适合的 debug 信息。使用全局函数 assert 来使用断言调试，assert 函数接受一个布尔表达式和一个断言失败时显示的消息，如：

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// this causes the assertion to trigger, because age is not >= 0
```

当前一个条件返回 false 的时候，后面的错误日志将会输出。

在这个例子中，只有当 age >= 0 的时候，条件被判定为 true，但是 age = -3，所以条件判定为 false，输出错误日志 “A person’s age cannot be less than zero”。

当然错误日志也可以省略，但是这样不利于调试，如

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

1.11.2 使用断言的时机

当需要检测一个条件可能是 false，但是代码运行必须返回 true 的时候使用。下面给出了一些常用场景，可能会用到断言检测：

- 传递一个整数类型下标的时候，比如作为数组的 Index，这个值可能太小或者太大，从而造成数组越界；
- 传递给函数的参数，但是一个无效的参数将不能在该函数中执行
- 一个可选类型现在是 nil，但是在接下来的代码中，需要是非 nil 的值才能够继续运行。

详见 Subscripts和Functions

> 注：断言会导致程序运行的中止，所以如果异常是预期可能发生的，那么断言是不合适的。这种情况下，异常是更合适的。断言保证错误在开发过程中会被发现，发布的应用里最好不要使用。

2 Swift 中文教程（二）基本运算符

运算符是一种特定的符号或表达式，用来检验、修改或合并变量。例如，用求和运算符 + 可以对两个数字进行求和 (如 let i = 1 + 2)；稍微复杂一点的例子有逻辑与操作符 && (如 if enteredDoorCode && passedRetinaScan) ，自增长运算符 ++i （这是 i=i+1 的简写方式）

Swift 支持 C 标准库中的大多数运算符并提升了各自的兼容性，从而可以排除常见的编码错误。赋值操作符 (=) 不会返回一个值，这样可以防止你因粗心将赋值运算符 (=) 写成 (==) 而引起错误。算术符 (+、-、*、/、% 等) 会检查与驳回值溢出，这样可以避免值类型的数据在超过值类型所允许的存储范围时，出现意想不到的数据。你可以选择使用 Swift 所提供的值溢出运算符进行量化溢出的行为，详细见溢出操作符。

与 C 语言不同，Swift 允许你对浮点数执行取余运算。同时，Swift 提供两个范围的运算符 (a..b 和 a...b)，作为表示一个数值范围的简写方式，这点 C 不支持。

本章节描述了 Swift 常见运算符。高级运算符覆盖了 Swift 的高级操作符，并且对自定义操作符，对自定义类型操作符的实现进行了描述。

2.1 术语

操作符都是一元、二元或三元:

- 一元操作符操作单个对象 (如 -a)。一元前缀操作符出现在对象前 (如!b)，一元后缀操作符在对象后出现 (如 i++)。
- 二元操作符操作两个对象 (如 2 + 3)，并且操作符位于两个元素中间。
- 三元操作符对两个对象进行操作。与 C 一样，Swift 仅支持一个三元操作符：三元条件操作符 (a ? b : c).

操作符所影响的值被称为操作数。表达式 1 + 2 中，符号 + 是一个二元运算符并且两个操作数分别为 1 和 2。

2.2 赋值运算符

赋值运算符 (a = b) 用 b 的值去初始化或更新 a 的值

```
let b = 10
var a = 5
a = b
// □ □a□ □ □10
```

假如右边赋值的数据为多个数据的元组，它的元素可以是一次性赋给的多个常量或变量

```
let (x, y) = (1, 2)
// x□ □1, □ □y□ □ 2
```

与 C 及 Objective-C 不同，Swift 中赋值运算符并不将自身作为一个值进行返回。所以以下的代码是不合法的：

```
if x = y {
    // □ □ , □ □x = y□ □ □ □ □ □ □ □
}
```

此特性帮助你避免因粗心将赋值运算符 (=) 写成 (==) 而引起的错误。因为 if x = y 这样写是无效的。

2.3 数学运算符

Swift 支持所有数字类型的四个标注运算符:

- 加法 (+) * 减法 (-)
- 乘法 (*)
- 除法 (/)

例如:

```
1 + 2 // equals 3
5 - 3 // equals 2
2 * 3 // equals 6
10.0 / 2.5 // equals 4.0
```

不同于 C 和 Objective-C，默认情况下 Swift 的算术运算符不允许值溢出。你可以通过 Swift 的溢出运算符来选择值的溢出情况 (例如 a & + b)。详见 Overflow Operators

加法运算符对字符串连接也一样适用，例如:

```
"hello, " + "world" // equals "hello, world"
```

两个字符，或者一个字符一个字符串，能组合成一个新的字符串:

```
let dog: Character = "dog"( □ □ □ □ □ □ □ □ □ □ □ □unicode □ □ , □ □ □ □ □ □ □ □ … … )
let cow: Character = "cow" ( □ □ … … )
let dogCow = dog + cow
// dogCow is equal to "dogcow"
```

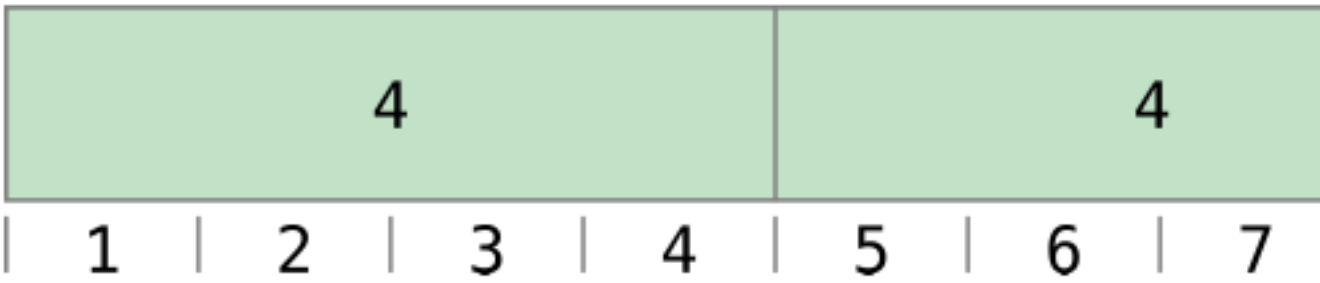
详见Concatenating Strings and Characters

2.3.1 取余运算符

取余运算符 (a % b) 计算出 a 是 b 的几倍然后返回被留下的值 (余数)。

> 注：余数运算符 (%) 亦称是其他语言的一个取模运算符。然而，其在 Swift 里意味着如果对负数操作，严格上讲，得到的是余数而不是模数。

这是余数运算符如何工作。要计算 9% 4，你首先得求出 9 是 4 的几倍：



9 能去除两个 4，并且余数是 1 (显示在橙色)。

在 Swift 中，这个将被写成：

```
9 % 4 // equals
```

确定 a % b 的答案，运算符% 计算下列等式并且返回余数作为其输出：

```
a = (b × some multiplier) + remainder
```

some multiplier 是 a 里面能包含 b 的最多倍数。

将 9 和 4 插入到公式：

```
9 = (4 × 2) + 1
```

同一个方法是应用的，当计算 a 时的一个负值的余数：

```
 -9 % 4 // equals -1
```

将 -9 和 4 插入到公式：

```
-9 = (4 × -2) + -1
```

产生余数值为 -1。

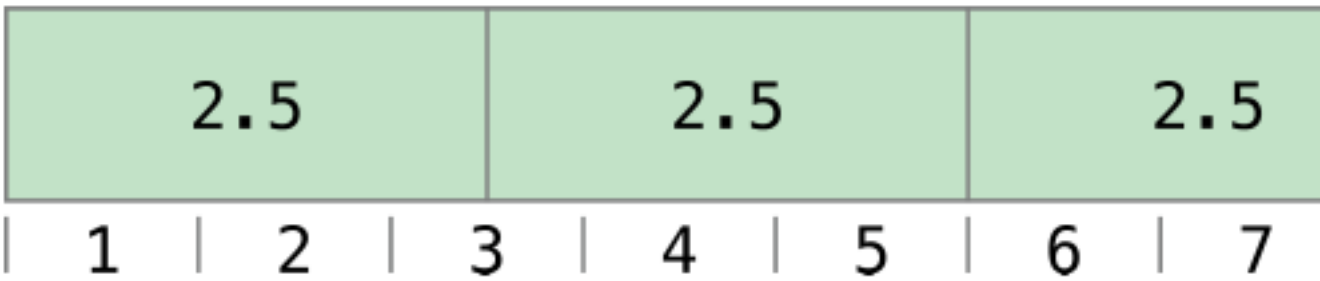
b 为负值时的 b 的符号被忽略，这意味着%b 和%-b 的结果是一样的。、

2.3.2 浮点余数计算

不同于 C 和 Objective-C，Swift 的余数运算符也能运用于浮点数：

```
8 % 2.5 // equals 0.5
```

在本例中，8 用 2.5 来分等于 3，余数是 0.5，因此余数为 0.5。



2.4 自增和自减运算符

像 C 一样,Swift 提供一个自增运算符 (++) 和自减运算符 (–) 作为增加或减少一个数值的一种快捷方式，增减量为 1。您能对任何整数或浮点类型的变量使用这些运算符。

```
var i = 0
++i // i now equals 1
```

每当你使用 ++i ，i 的值增加 1，本质上 ++i 可以看做是 i=i+1，同样 –i 可以看做是 i=i-1。

++ 和 –符号可以使用作为前缀算符或作为后缀运算符。++i 和 i++ 是两个有效的方式给 i 的值增加 1，同样, –i 和 i–如是。

注意这些运算符修改 i 并且返回值。如果你只想要增加或减值 i，您可以忽略返回值。然而，如果你使用返回值，根据下列规则将是不同的根据的您是否使用了运算符的前缀或后缀版本，它：

- 如果运算符在变量之前被写，它在返回其值之前增加变量。
- 如果运算符在变量之后被写，它在返回其值之后增加变量。

例如：

```
var a = 0
let b = ++a
// a and b are now both equal to 1
let c = a++
// a is now equal to 2, but c has been set to the pre-increment value of 1
```

在上面的例子中，let b = ++a 中 a 在返回其值之前增加，这就是为什么 a 和 b 的新值是等于 1。

然而，let c = a++ 中 a 在返回其值之后增加，这意味着 c 获得 a 的原值 1，然后 a 自增，a 等于 2。

除非你需要特定工作情况下才使用 i++，否则在所有的情况下建议你使用 ++i 和 –i， 因为他们修改 i 并返回值的行为符合我们的预期。

2.4.1 一元减运算符

一个数值前加了符号 -，叫作一元减运算符：

```
let three = 3
let minusThree = -three // minusThree equals -3
let plusThree = -minusThree // plusThree equals 3, or "minus minus three"
```

一元减运算符 (-) 直接地被加在前面，在它起作用的值之前，不用任何空白空间。

2.4.2 一元加运算符

一元加运算符 (+) 返回它起作用的值，不做任何变动：

```
let minusSix = -6
let alsoMinusSix = +minusSix // alsoMinusSix equals -6
```

虽然一元加上运算符实际上不执行什么，当你也使用一元减负数的运算符时，你能使用它提供对称的正数。

2.5 复合赋值操作符

Swift 提供类似C 语言的复合赋值操作符，即把赋值和另一个运算合并起来。举个例子，像加法赋值运算符 (+ =):

```
var a = 1
a += 2
// a is now equal to 3
```

表达式 a += 2 比 a = a + 2 更精炼。加法赋值运算符能够有效地把加法和赋值组合到一个运算, 同时执行这两个任务。

> 要注意的是，复合赋值操作符不返回值。例如，你不能写让成 let b = + = 2, 这种行为不同于上面提到的递增和递减运算符。

复合赋值运算符的完整列表可以在 [Expressions] 那一章节找到

2.6 比较运算符

Swift 支持所有标准 c 的比较运算符

- 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：Swift 提供两个恒等运算符 (=== and !==)，用它来测试两个对象引用是否来自于同一个对象实例。详见Classes and Structures。每个比较操作符返回一个 Bool 值来表示语句是否为真：

```
1 == 1 // true, because 1 is equal to 1
2 != 1 // true, because 2 is not equal to 1
2 > 1 // true, because 2 is greater than 1
1 < 2 // true, because 1 is less than 2
1 >= 1 // true, because 1 is greater than or equal to 1
2 <= 1 // false, because 2 is not less than or equal to 1
```

比较操作符通常用在条件语句，如 if 语句：

```
let name = "world"
if name == "world" {
    println("hello, world")
} else {
    println("I'm sorry \(name), but I don't recognize you")
}
// prints "hello, world", because name is indeed equal to "world"
```

想要了解更多有关的 if 语句，请参阅控制流。

2.7 三元条件运算符

三元条件运算符是一种特殊的运算符，有三个部分，其形式为 question? answer1：answer2. 这是一个用来测试两种表达式基于输入是真或是假的快捷方式。如果 question? 为真时, 它评估 answer1 并返回其值; 否则, 它评估 answer2 并返回其值。三元条件运算符是下面的代码的简化:

```
if question {
    answer1
} else {
    answer2
}
```

这里举一个列子，计算一个表行像素的高度，如果行有一个头，行高应该是 50 像素，比内容要高度要高。如果行没有头是 20 像素：

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

前面的例子也可以用下面的的代码:

```
let contentHeight = 40
let hasHeader = true
```

```
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// rowHeight is equal to 90
```

第一个例子使用的三元条件运算符，意味着 rowHeight 可以在一行代码被设置为正确的值。这比第二个示例更简洁， 不需要课外的 rowHeight 变量， 因为它的价值不需要在一个 if 语句中修改。

三元条件运算符提供了一个高效的写法来决定哪个表达式会被执行。不过还是请小心使用三元条件运算符，其简洁性如果过度使用会导致阅读代码的困难。要避免多个实例的三元条件运算符组合成一个复合语句。

范围运算符

Swift 包含两个范围运算符，能快捷的表达一系列的值

2.7.1 封闭范围运算符

封闭范围运算符 (a...b) 定义了一个范围, 从 a 到 b, 并包括 a 和 b 的值。

当要在一个范围内迭代所有可能的值的时候，范围运算符是非常有用的, 例如 for-in 循环

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

欲了解更多 for-in 循环，请参阅控制流。

2.7.2 半封闭的区域运算符

半封闭的区域运算符 (a..b) 定义了从 a 到 b 的范围，但不包括 b。它被认为是半封闭的，因为它包含第一个值，而不包含最终值。

半封闭的范围使用明确，当你使用从零开始的列表，如数组，它是有用的数到 (但不包括) 列表的长度:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..count {
    println("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

请注意, 该数组包含四个项目, 但 0 . . 数只数到 3 (数组中的最后一个项目的索引)，因为它是一个半封闭的范围。欲了解更多有关数组的信息，请参阅数组

2.8 逻辑运算符

逻辑运算符修改或结合布尔逻辑值 true 和 false。Swift 支持这三个标准逻辑运算符基于 c 语言:

- Logical NOT (!a)
- Logical AND (a && b)
- Logical OR (a || b)

2.8.1 逻辑非运算符

逻辑非运算符 (!a) 转化一个 Bollean 值，true 变成 false，false 变成 true。

逻辑操作符是一个前缀操作符， 并立即出现在它修饰的值之前，没有任何空白，它被解读为” 不是”， 见下面的例子:

```
let allowedEntry = false
if !allowedEntry {
    println("ACCESS DENIED")
```

```
}
// prints "ACCESS DENIED"
```

这句话 if !allowedEntry 能理解为 “if not allowedEntry.” 只执行后续的行，如果 “not allowedEntry” 是 true; 那就是说 if allowedEntry 是 false.

在这个例子中，精心挑选的布尔常量和变量名可以帮助保持代码的可读性和简洁，同时避免双重否定或混乱的逻辑语句。

2.8.2 逻辑与运算符

逻辑与运算符：（A && B）创建的表达式中，A 和 B 两个值必须同时为 true 时表达式才正确。

其中 A 或者 B 有任一值是 false 时，逻辑与算符表示不成立，必须两者同时为 true 时才成立。事实上，如果第一个值是 false，第二个值甚至不会再进行判断，因为必须是两个值皆为 true，已经有一方 false、则没必要再往下面进行判断了。这被称作短路条件。

以下这个例子判断两个 Bool 类型的值，并只有这两个值都为真的时候会输出：Welcome。失败则输出 “ACCESS DENIED”：

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "ACCESS DENIED ”
```

2.8.3 逻辑或运算符

表达式（a || b）运算符中、只要 a 或者 b 有一个为 true，表达式就成立。

与上面的逻辑与运算符相似，逻辑或运算符使用短路条件判断，如果左边是 true，那么右边不会被判断，因为整体结果不会改变了。

在下面的例子中，第一个布尔值（hasDoorKey）为 false，但第二个值 (knowsOverridePassword) 为 true。因为两者有一个值是 true，整个表达式的计算结果也为 true，正确输出：Welcome!

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

2.8.4 复合逻辑表达式

你可以将多个逻辑运算符复合来创建更长的复合表达式:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

相比于之前两个单独分开的运算符，本次通过多重嵌套、将我们上面的 &&、|| 运算符相结合组合成一个较长的复合表达式。看起来有点饶人、其实本质还是两两相比较、可以简单地看成 A && B || C || D、从左往右根据运算符优先级进行判断、注意区分开 &&、||、只要牢记运算逻辑 && 需要两者都为 true、|| 则只需要一方为 true 则运算符正确即可解析整个复合表达式、透过现象看本质。

2.8.5 明确地括号 (翻译成中文语句不连贯太特么饶人了、怒了自己理解。)

复合表达式中，我们可以添加进 () 使确逻辑意图更加明确，上面的例子中，我们可以在第一部分上加括号来使意义更明确。

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

在复合逻辑表达式中、我们可以使用括号明确地表示我们需要将几个值放在一个单独的逻辑运算中去判断得出结果、最后根据 () 内的结果再去与后面的值进行判断、看上面的例子、就像我们小学学加减乘除一样、如果没有括号 () 我们肯定是按照运算符的优先级去判断、但此时有了括号、我们需要先运算其中的逻辑运算符得到它们的值. 使用括号 () 在符合逻辑表达式中可以更明确的你的意图。

3 Swift 中文教程（三）字符串和字符

一个字符串 String 就是一个字符序列，像"hello,world","albatross" 这样的。Swift 中的字符串是用 String 关键词来定义的，同时它也是一些字符的集合，用 Character 定义。

Swift 的 String 和 Character 类型为代码提供了一个快速的，兼容 Unicode 的字符解决方案。String 类型的初始化和使用都是可读的，并且和 C 中的 strings 类似。同时 String 也可以通过使用 + 运算符来组合，使用字符串就像使用 Swift 中的其他基本类型一样简单。

3.1 字符串常量

在代码中可以使用由 String 预先定义的字符串常量，定义方式非常简单：

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

字符串常量可以包括下面这些特殊字符：

- 空字符 \0，反斜杠 \，制表符,换行符 \n，回车符 \r，双引号 \" 和单引号 \'
- 单字节 Unicode 字符，\xnn，其中 nn 是两个十六进制数
- 双字节 Unicode 字符，\unnnn，其中 nnnn 是四个十六进制数
- 四字节 Unicode 字符，\Unnnnnnnn，其中 nnnnnnnn 是八个十六进制数

下面的代码给出了这四种字符串的例子：

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

3.2 初始化一个空串

初始化一个空串时有两种形式，但是两种初始化方法的结果都一样，表示空串

```
var emptyString = "" // empty string literal
var anotherEmptyString = String() // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

通过 isEmpty 属性可以检查一个字符串是否为空

```
if emptyString.isEmpty {
    println("Nothing to see here")
}
// prints "Nothing to see here"
```

3.3 变长字符串

如果使用 var 关键词定义的字符串即为可修改的变长字符串，而 let 关键词定义的字符串是常量字符串，不可修改。

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
```

3.4 字符串不是指针，而是实际的值

在 Swift 中，一个 String 类型就是一个实际的值，当定义一个新的 String，并且将之前的 String 值拷贝过来的时候，是实际创建了一个相等的新值，而不是仅仅像指针那样指向过去。

同样在函数传递参数的时候，也是传递的实际值，并且创建了一个新的字符串，后续的操作都不会改变原有的 String 字符串。

3.5 字符

Swift 的字符串 String 就是由字符 Character 组成的，每一个 Character 都代表了一个特定的 Unicode 字符。通过 for-in 循环，可以遍历字符串中的每一个字符：

```
for character in "Dog!" {
    println(character)
}
// D
// o
// g
// !
//
```

你也可以仅仅定义一个单独的字符：

```
let yenSign: Character = "￥"
```

3.6 字符计数

使用全局函数 countElements 可以计算一个字符串中字符的数量：

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

3.7 组合使用字符和字符串

String 和 Character 类型可以通过使用 + 号相加来组合成一个新的字符串

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"
let stringPlusCharacter = string1 + character1 // equals "hello!"
let stringPlusString = string1 + string2 // equals "hello there"
let characterPlusString = character1 + string1 // equals "!hello"
let characterPlusCharacter = character1 + character2 // equals "!?"
```

也可以使用 += 号来组合：

```
var instruction = "look over"
instruction += string2
// instruction now equals "look over there"
var welcome = "good morning"
welcome += character1
// welcome now equals "good morning!"
```

3.8 使用字符串生成新串

通过现有的字符串，可以使用如下方法来生成新的字符串：

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

在上面这个例子中，首先使用 multiplier 这个字符串 3，来作为新串的一部分，用 (multiplier) 添加，同时上面的例子还用到了类型转换 Double(multiplier)，将计算结果和字符串本身都作为元素添加到了新的字符串中。

3.9 字符串比较

Swift 提供三种方法比较字符串的值：字符串相等，前缀相等，和后缀相等

3.9.1 字符串相等

当两个字符串的包含完全相同的字符时，他们被判断为相等。

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

3.9.2 前缀（prefix）相等和后缀 (hasSuffix) 相等

使用 string 类的两个方法 hasPrefix 和 hasSuffix, 来检查一个字符串的前缀或者后缀是否包含另外一个字符串，它需要一个 String 类型型的参数以及返回一个布尔类型的值。两个方法都会在原始字符串和前缀字符串或者后缀字符串之间做字符与字符之间的。

下面一个例子中, 用一个字符串数组再现了莎士比亚的罗密欧与朱丽叶前两幕的场景。

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

你可以使用 hasPrefix 方法和 romeoAndJuliet 数组计算出第一幕要表演多少个场景。

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        ++act1SceneCount
    }
}
println("There are \(act1SceneCount) scenes in Act 1")
// □ □ "There are 5 scenes in Act 1"
```

同理，使用 hasSuffix 方法去计算有多少个场景发生在 Capulet 公馆和 Friar Lawrence 牢房

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

3.10 大小写字符串

你可以从一个 String 类型的 uppercaseString 和 lowercaseString 中获得一个字符串的大写或小写。

```
let normal = "Could you help me, please?"
let shouty = normal.uppercaseString
// shouty is equal to "COULD YOU HELP ME, PLEASE?"
let whispered = normal.lowercaseString
// whispered is equal to "could you help me, please?"
```

3.11 Unicode

Unicode 是编码和表示文本的国际标准。它几乎可以显示所有语言的所有字符的标准形态。还可以从类似于文本文件或者网页这样的外部源文件中读取和修改他们的字符。

3.11.1 Unicode 术语

每一个 Unicode 字符都能被编码为一个或多个 unicode scalar。一个 unicode scalar 是一个唯一的 21 位数（或者名称），对应着一个字符或者标识。例如 U+0061 是一个小写的 A (“a”), 或者 U+1F425 是一个面向我们的黄色小鸡

当一个 Unicode 字符串写入文本或者其他储存时，unicode scalar 会根据 Unicode 定义的格式来编码。每一个格式化编码字符都是小的代码块，称成为

code units. 他包含 UTF-8 格式 (每一个字符串由 8 位的 code units 组成)。和 UTF-16 格式（每一个字符串由 16 位的 code units 组成)

3.11.2 Unicode 字符串

Swift 支持多种不同的方式取得 Unicode 字符串.

你可以使用 for-in 语句遍历字符串，来获得每一个字符的 Unicode 编码值。这个过程已经在字符（Working with Characters）描述过了。

或者，下面三个描述中使用合适的一个来获得一个字符串的值

- UTF-8 字符编码单元集合使用 String 类型的 utf-8 属性
- UTF-16 字符编码单元集合使用 String 类型的 utf-16 属性
- 21 位 Unicode 标量集合使用 String 类型的 unicodeScalars 属性

下面的每一个例子展示了不同编码显示由 D , o , g , !

(DOG FACE, 或者 Unicode 标量 U+1F436）字符组成的字符串

3.11.3 UTF-8

你可以使用 String 类型的 utf8 属性遍历一个 UTF-8 编码的字符串。这个属性是 UTF8View 类型，UTF8View 是一个 8 位无符号整形 (UInt8) 的集合, 集合中的每一个字节都是 UTF-8 编码。

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

在上面的例子中，前 4 个十进制 codeunit 值（68,111,103,33）显示为字符串 D , o ,g 和! ，和他们的 ASCII 编码相同一样。后面 4 个 codeunit 的值 (240,159,144,182）是 DOG FACE 字符的 4 字节 UTF-8 编码。

3.11.4 UTF-16

你可以使用 String 类型的 utf16 属性遍历一个 UTF-16 编码的字符串。这个属性是 UTF16View 类型，UTF16View 是一个 16 位无符号整形 (UInt16) 的集合, 集合中的每一个字节都是 UTF-16 编码。

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

同理，前 4 个十进制 codeunit 值（68,111,103,33）显示为字符串 D , o ,g 和! ，他们的 UTF-16 的 codeunit 和他们 UTF-8 的编码值相同。

第 5 和第 6 个 codeunit 值（55357 和 56374）是 DOG FACE 字符的 UTF-16 的代理对编码。他们的值是由值为 U+D83D（十进制 55357）的高位代理 (lead surrogate) 和值为 U+DC36 (十进制 56374) 的低位代理（trail surrogate）组成。

3.11.5 Unicode 标量

你可以使用 String 类型的 unicodeScalars 属性遍历一个 Unicode 标量编码的字符串。这个属性是 UnicodeScalarsView 类型，UnicodeScalarsView 是一个 UnicodeScalar 类型的集合。每一个 Unicode 标量都是一个任意 21 位 Unicode 码位，没有高位代理，也没有低位代理。

每一个 UnicodeScalar 使用 value 属性，返回标量的 21 位值，每一位都是 32 位无符号整形 (UInt32) 的值:

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

value 属性在前 4 个 UnicodeScalar 值（68,111,103,33）再一次展示编码了字符 D , o , g 和！。第五个也是最后一个 UnicodeScalar 是 DOG FACE 字符，十进制为 128054，等价于 16 进制的 1F436，相当于 Unicode 标量的 U+1F436。

每一个 UnicodeScalar 可以被构造成一个新的字符串来代替读取他们的 value 属性，类似于插入字符串。

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

```
// !
//
```

4 Swift 中文教程（四）集合类型

Swift 提供两种集合类型来存储集合，数组和字典。数组是一个同类型的序列化列表集合。字典是一个能够使用类似于键的唯一标识符来获取值的非序列化集合。

在 Swift 中，数组和字典里的键和值都必须是明确的某个特定类型。这意味这数组和字典不会插入一个错误的类型的值, 以致于出错。这也意味着当你在数组和字典中取回数值的时候能够确定它的类型。

Swift 使用确定的集合类型可以保证代码工作是不会出错，和让你在开发阶段就能更早的捕获错误。

> 注意：Swift 的数组储存不同的类型会展示出不同的行为，例如变量，常量或者函数和方法。更多的信息参见 Mutability of Collections and Assignment 和 Copy Behavior for Collection Types.

4.1 数组

数组是储存同类型多数值的序列化列表。同样的值可以在数组的不同位置出现多次。

Swift 数组只能存储特定的某一类值，这和 Objective-C 中的 NSArray 和 NSMutableArray 类是有区别的。因为它们是储存各种的对象，而且并不提供返回任何有关对象的具体信息。在 Swift 中，无论是确定的声明，还是隐式的声明，数组是非常确定它自身是储存什么样的类型，而且，它并不一定要求储存的是类对象。如果你创建一个存储 int 值的数组，那么这个数组中只能出现 int 类型的值。Swift 数组是类型安全的，而且它一直都清楚它自身所能包含的值的类型。

4.1.1 数组的简略语法

定义数组的完整写法是 Array，其中 SomeType 是你想要包含的类型。你也可以使用类似于 SomeType[] 这样的简略语法。虽然这两种方法在功能上是相同的。但是我们更推荐后者，而且它会一直贯穿于本书。

4.1.2 数组实量（**Array Literals**）

你可以用一个数组实量（Array Literals）来初始化一个数组，它是用简略写法来创建一个包含一个或多个的值的数组。一个数组实量（Array Literals）是由它包含的值，“,” 分隔符已经包括以上内容的中括号对 “[]” 组成：

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

下面的例子是创建一个叫 shoppinglist，储存字符串（String）类型的数组。

```
var shoppingList: String[] = ["Eggs", "Milk"]
// □ □ □ □ □ □ □ □ □ □ □ □ □ □shoppingList
```

shoppinglist 变量被定义为字符串（String）类型的数组，写作 String[]。因为这个数组被确定为字符串类型（String），所以它只能储存字符串（String）类型的值。在这里，我们用两个字符串类型的值（”Eggs” and “Milk”）和数组实量（Array Literals）的写法来初始化 shoppingList 数组。

> 注意 shoppingList 数组是被定义为一个变量（使用 var 标识符）而不是常量（使用 let 标识符），所以在下面的例子可以直接添加元素。

在这个例子中，数组实量（Array Literals）只包含两个字符串类型的值，这符合了 shoppingList 变量的定义（只能包含字符串（String）类型的数组），所以被分配的数组实量（Array Literals）被允许用两个字符串类型的值来初始化。

得益于 Swift 的类型推断，当你用相同类型的值来初始化时，你可以不写明类型。初始化 shoppingList 可以用下面这个方法来代替。

```
var shoppingList = ["Eggs", “ Milk"]
```

因为数组实量（Array Literals）中所有的值都是同类型的，所以 Swift 能够推断 shoppingList 的类型为字符串数组（String[]）。

4.1.3 读取和修改数组

你可以通过方法和属性，或者下标来读取和修改数组。

通过只读属性 count 来读取数组的项数；

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

通过一个返回布尔类型的 isEmpty 属性检查数组的项数是否为 0

```
if shoppingList.isEmpty {
    println("The shopping list is empty.")
} else {
    println("The shopping list is not empty.")
}
// □ □ □ "The shopping list is not empty."
```

在数组末尾增加一项可以通过 append 方法

```
shoppingList.append("Flour")
// shoppingList □ □ □ □3□
```

同理，也可以用（+=）操作符来把一个元素添加到数组末尾

```
shoppingList += "Baking Powder"
// shoppingList □ □ □ □4□
```

你也可以用（+=）操作符来把一个数组添加到另一个数组的末尾

```
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList □ □ □ □7□ □ □
```

从数组中取出一个值可以使用下标语法。如果你知道一个元素的索引值，你可以数组名后面的中括号中填写索引值来获取这个元素

```
var firstItem = shoppingList[0]
// firstItem □ □ “ Eggs"
```

注意，数组的第一个元素的索引值为 0，不为 1，Swift 的数组的索引总是从 0 开始;

你可以使用下标语法通过索引修改已经存在的值。

```
shoppingList[0] = "Six eggs"
// □ □ □ □ □ □ □ □ □ □ "Six eggs" □ □ □ □ “ Eggs"
```

你可以使用下标语法一次性改变一系列的值，尽管修改的区域远远大于要修改的值。在下面的雷子中，把 “Chocolate Spread”, “Cheese” 和”Butter” 替换为”Bananas” 和”Apples”:

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList □ □ □ □6□ □ □
```

> 注意，你不能使用下标语法在数组中添加一个元素，如果你尝试使用下标语法来获取或者设置一个元素，你将得到一个运行时的错误。尽管如此，你可以通过 count 属性验证索引是否正确再使用它。除非 count 等于 0（也就是说数组是空的），最大的索引都是 count-1，因为数组的索引从 0 开始计算。

在一个特定的索引位置插入一个值，可以使用 insert(atIndex:) 方法

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList □ □ □ □7□ □ □
// "Maple Syrup" □ □ □ □ □ □ □
```

这里调用 insert 方法指明在 shoppingList 的索引为 0 的位置中插入一个新元素 “Maple Syrup”

同理，你可以调用 removeAtIndex 方法移除特定的元素。这个方法移除特定索引位置的元素并返回这个被移除的元素（尽管你可能并不关心这个返回值)。

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// □ □ □ □ □0□ □ □ □ □ □
// shoppingList □ □ □ □6□ □ □ , □ □ □ Maple Syrup
// mapleSyrup □ □ □ □ □ □ □ □ "Maple Syrup" □ □ □
```

当元素被移除的，数组空缺的位置将会被填补，所以现在索引位置为 0 的元素再一次等于”Six eggs”:

```
firstItem = shoppingList[0]
// firstItem □ □ □ □ "Six eggs ”
```

如果你想从数组中移除最后一个元素，使用 removeLast 方法比 removeAtIndex 更方便，因为后者需要通过 count 属性计算数组的长度。和 removeAtIndex 方法一样，removeLast 会返回被移除的元素。

```
let apples = shoppingList.removeLast()
//□ □ □ □ □ □ □ □ □ □ □ □
// shoppingList □ □ □ □5□ □ □ , □ □ □ cheese
// apples □ □ □ □ □ □ □ □ □ "Apples" string
```

4.1.4 遍历数组

可以使用 for − in 循环来遍历数组中的值

```
for item in shoppingList {
    println(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

如果既需要每个元素的值，又需要每个元素的索引值，使用 enumerate 函数代替会更方便，enumerate 函数对于每一个元素都会返回一个包含元素的索引和值的元组（tuple)。你可以在遍历部分分解元素并储存在临时变量或者常量中。

```
for (index, value) in enumerate(shoppingList) {
    println("Item \(index + 1): \(value)")
}
// □ □ 1: Six eggs
// □ □ 2: Milk
// □ □ 3: Flour
// □ □ 4: Baking Powder
// □ □ 5: Bananas
```

如需更多 for-in 循环信息, 参见 For Loops.

4.1.5 创建和初始化数组

创建一个空的数组和确定的类型（不包含初始化值）使用的初始化语法：

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts.count) items.")
// □ □ "someInts is of type Int[] with 0 items. ”
```

注意，someInt 的变量类型被确定为 Int[]，因为它使用生成 Int[] 的初始化方法。

或者，如果上下文（context）已经提供类型信息，例如函数参数或者已经确定类型的常量和变量，你可以从空的数组实量（Array Literals）创建一个空数组，写作 []（空的中括号对）。

```
someInts.append(3)
// someInts □ □ □ □1□Int□ □ □ □
someInts = []
// someInts □ □ □ □ □ □ □ □ □ □, □ □ □ □ □ □ □Int[];
```

Swift 数组类型也提供初始化方法来创建确定长度和提供默认数值的数组。你可以通过这个初始化方法增加一个新的数组，元素的数量成为 count，合适的默认值为 repeatedValue

```
var threeDoubles = Double[](count: 3, repeatedValue: 0.0)
// threeDoubles □ □ □ □ Double[], □ □ □ [0.0, 0.0, 0.0]
```

得益于类型推断，你并不需要指明这个数组储存的类型就能使用这个初始化方法，因为它从默认值中就能推断出来。

```
var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
// anotherThreeDoubles □ □ □ □ Double[], □ □ □ □ [2.5, 2.5, 2.5]
```

最后，你可以使用（+）操作符就能创建一个新的数组，把两个存在的数组添加进来这个新的数组类型从你添加的两个数组中推断出来

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles □ □ □ □ Double[], □ □ □ [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

4.2 字典

字典是储存同一类型多个值的容器。每一个值都对应这一个唯一的键 (Key)，就像是字典内的每一个值都有一个标识符。和数组内的元素是有区别的，字典内的元素是没有特殊的序列的。当你需要根据标识符来查找批量的值时，就可以使用字典，和在真实世界的字典中寻找某个字的解释相似。

Swift 字典储存一个特定类型的键和值，与 Objective-C 的 NSDictionary 和 NSMutableDictionary 不同，因为它们是使用各种的对象来作为它们的键和值，而且并不提供任何有关对象的具体信息。在 Swift 中，对于一个特定的字典，它所能储存的键和值的类型都是确定的，无论是明确声明的类型还是隐式推断的类型。

Swift 的字典写法是 Dictionary,KeyType 是你想要储存的键的类型，Value-Type 是你想要储存的值的类型。

唯一的限制就是 KeyType 必须是可哈希的（hashable）——就是提供一个形式让它们自身是独立识别的。Swift 的所有基础类型（例如字符串（String），整形（Int），双精度（Double）和布尔（Bool））在默认是可哈希的（hashable），和这些类型都常常被当作字典的键。没有协助值（associated values）的枚举成员 (具体描述在 Enumerations）默认也是可哈希的（hashable）。

4.2.1 字典实量（Dictionary Literals）

你可以直接用一个字典实量（Dictionary Literals）初始化一个字典。和前面定义一个数组实量（Array Literals）的语法一样。字典实量（Dictionary Literals）就是使用简略写法直接写一个或者多个对应的键和值对来定义一个字典。

一个键值对是一个键和值的组合。在字典实量（Dictionary Literals）里面，每一个键值对总是用一个冒号把键和值分割。键值对的写法就想是一个列表，使用逗号分割，并被一对中括号 □ 包含着：

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

在下面的例子，将会创建一个字典来储存国际机场的名字。在这个字典里面，键是三个字的国际航空运送协会代码，以及它的值是机场的名称：

```
var airport :Dictionary<String, String> = ["TYO": "Tokyo", "DUB": “ Dublin"]
```

airport 字典被定义为一个类型为 Dictionary，这意味这，这个字典的键类型是字符串 String，和它的值的类型也是 String。

> 注意 airport 字典是被定义为一个变量（使用 var 标识符）而不是常量（使用 let 标识符），所以在下面的例子可以直接添加元素。

airport 字典使用一个包含两个键值对的字典实量（Dictionary Literals）来初始化。第一对有一个叫 “TYO” 的键和一个叫 “Tokyo” 的值，第二对有一个叫 “DUB” 的键和一个叫 “Dublin” 的值。

这个字典实量（Dictionary Literals）包含两个字符串（String）：字符串对。这符合 airport 变量定义的类型（一个字典只包括字符串（String）键和字符串（String）值），所以在分配字典实量（Dictionary Literals）的时候被允许作为 airport 字典的两个初始化元素。

和数组一样，如果你初始化一个字典的时候使用相同的类型，你可以不指明字典的类型。airport 初始化可以用下面这个简略写法来代替：

```
var airports = ["TYO": "Tokyo", "DUB": “Dublin”]
```

因为所有的键在字面上都是相同的类型，同样，所有的值也是同样的类型，所以 Swift 可以推断为 Dictionary 是 airports 字典的正确类型。

4.2.2 读取和修改字典

你可以通过属性，方法或者下标来读取和修改字典。和数组一样，你使用只读的 count 属性来检查字典（Dictionary）包含多少个元素。

```
println("The dictionary of airports contains \(airports.count) items.")
// □□ "The dictionary of airports contains 2 items."
```

你可以使用下标语法给一个字典添加一个元素。使用合适类型作为新的键，并分配给它一个合适类型的值

```
airports["LHR"] = "London"
//  airports dictionary □□□ 3 items
```

你也可以使用下标语法去改变一个特定键所关联的值。

```
airports["LHR"] = "London Heathrow"
//"LHR" □□□□□□□□ "London Heathrow"
```

同样, 使用字典的 updateValue(forKey:) 方法去设置或者更新一个特定键的值. 和上面的下标例子一样, updateValue(forKey:) 方法如果键不存在则会设置它的值，如果键存在则会更新它的值, 和下标不一样是, updateValue(forKey:) 方法如果更新时，会返回原来旧的值，意味着你可以使用这个来判断数据是否发生了更新。

updateValue(forKey:) 方法返回一个和字典的值相同类型的可选值. 例如，如果字典的值的类型时 String，则会返回 String? 或者叫 “可选 String“，这个可选值包含一个如果值发生更新的旧值和如果值不存在的 nil 值。

```
if let oldValue = airports.updateValue("Dublin International", forKey: "DUB") {
    println("The old value for DUB was \(oldValue).")
}
// prints "The old value for DUB was Dublin."
```

你也可以使用下标语法通过特定的键去读取一个值。因为如果他的值不存在的时候，字典的下标语法会返回一个字典的值的类型的可选值。如果字典中的键包含对应的值，这字典下标语法会返回这个键所对应的值，否则返回 nil

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

你可以使用下标语法把他的值分配为 nil，来移除这个键值对。

```
airports["APL"] = "Apple International"
// "Apple International" □ □APL□ □ □ □ □ □,□ □ □ □ □
airports["APL"] = nil
// APL□ □ □ □ □ □ □ □ □
```

同样，从一个字典中移除一个键值对可以使用 removeValueForKey 方法，这个方法如果存在键所对应的值，则移除一个键值对，并返回被移除的值，否则返回 nil。

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

```
    println("The removed airport's name is \(removedValue).")
} else {
   println("The airports dictionary does not contain a value for DUB.")
}
// prints "The removed airport's name is Dublin International."
```

4.2.3 遍历字典

你可以使用一个 for-in 循环来遍历字典的键值对。字典中的每一个元素都会返回一个元祖（tuple），你可以在循环部分分解这个元祖，并用临时变量或者常量来储存它。

```
for (airportCode, airportName) in airports {
    println("\(airportCode): \(airportName)")
}
// TYO: Tokyo
// LHR: London Heathrow
```

更多有关 for-in 循环的信息, 参见 For Loops.

你也可以读取字典的 keys 属性或者 values 属性来遍历这个字典的键或值的集合。

```
for airportCode in airports.keys {
    println("Airport code: \(airportCode)")
}
// Airport code: TYO
// Airport code: LHR
for airportName in airports.values {
    println("Airport name: \(airportName)")
}
// Airport name: Tokyo
// Airport name: London Heathrow
```

如果你需要一个接口来创建一个字典的键或者值的数组实例，你可以使用 keys 或者 values 属性来初始化一个数组。

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

```
// airportCodes is ["TYO", "LHR"]
let airportNames = Array(airports.values)
// airportNames is ["Tokyo", "London Heathrow"]
```

> 注意 Swift 中的字典类型是非序列化集合，序列化取回键，值，或者键值对的顺序是不明确的。

4.2.4 创建一个空字典

和字典一样，你可以使用确定类型的语法创建一个空的字典。

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers □ □ □ □ □ Dictionary<Int, String> □ □ □ □ □
```

这个例子创建一个 Int，String 类型的字典来储存可读性较好的整数值。它的键是 Int 类型，它的值是 String 类型。

如果上下文（context）中已经提供类型信息，可用一个字典实量（Dictionary Literal）创建一个空的字典，写作 [:]（由一对 □ 包含一个冒号:）

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers□ □ □ □1 □ □ □ □
namesOfIntegers = [:]
// namesOfIntegers □ □ □ □ □ □Int, String□ □ □ □ □
```

> 注意在这个场景，Swift 数组和字典类型是一个内置的集合。更多的内置类型和集合参见 Generics

4.3 可变集合类型

数组和字典都是在一个集合中一起储存多个变量. 如果你创建一个数组或者字典，再包含一个变量，创建的这个变量被称为可变的（mutable）这意味这，你可以在创建之后增加更多的元素来改变这个集合的长度，或者移除已经包含的。相反的, 如果你把一个数组或者字典定义为常量，则这个数组或者字典不是可变的，他们包含的项数并不能被改变。

在字典中，不可变也意味着你不能替换已经存在的键的值。一个不可变字典，一旦被设置就不能改变。

数组的不可变有一点点的不同。然而，你仍然不能做任何改变项数的操作。但是你可以重新设置一个已经存在的索引，这使得当 Swift 的数组的长度确定时，能更好地优化数组的性能。

拥有可变行为的数组也影响着数组实例的分配和修改，更多内容参见 Assignment and Copy Behavior for Collection Types.

> 注意在一个集合的项数不需要被改变时，创建不可变集合是非常好的尝试。这样的话 Swift 编译器就能充分利用你所创造的集合的性能。# Swift 中文教程（五）控制流

Swift 提供了所有 C 语言中相似的控制流结构。包括 for 和 while 循环；if 和 switch 条件语句；break 和 continue 跳转语句等。

Swift 还加入了 for-in 循环语句，让编程人员可以在遍历数组，字典，范围，字符串或者其它序列时更加便捷。

相对于 C 语言，Swift 中 switch 语句的 case 语句后，不会自动跳转到下一个语句，这样就避免了 C 语言中因为忘记 break 而造成的错误。另外 case 语句可以匹配多种类型，包括数据范围，元组，或者特定的类型等。switch 语句中已匹配的数值也可以被用在后续的 case 语句体中，where 关键词还能被加入任意的 case 语句中，来增加匹配的方式。

4.4 for 循环

for 循环可以根据设置，重复执行一个代码块多次。Swift 中提供了两种 for 循环方式:

for-in 循环，对于数据范围，序列，集合等中的每一个元素，都执行一次

for-condition-increment，一直执行，知道一个特定的条件满足，每一次循环执行，都会增加一次计数

4.4.1 for-in 循环

下面的例子打印出了 5 的倍数序列的前 5 项

```
for index in 1...5 {
    println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
```

```
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

迭代的项目是一个数字序列，从 1 到 5 的闭区间，通过使用 (...) 来表示序列。index 被赋值为 1，然后执行循环体中的代码。在这种情况下，循环只有一条语句，也就是打印 5 的 index 倍数。在这条语句执行完毕后，index 的值被更新为序列中的下一个数值 2，println 函数再次被调用，一次循环直到这个序列的结尾。

在上面的例子中，index 在每一次循环开始前都已经被赋值，因此不需要在每次使用前对它进行定义。每次它都隐式地被定义，就像是使用了 let 关键词一样。注意 index 是一个常量。

> 注意：index 只在循环中存在，在循环完成之后如果需要继续使用，需要重新定义才可以。

如果你不需要序列中的每一个值，可以使用 _ 来忽略它，仅仅只是使用循环体本身：

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

这个例子计算了一个数的特定次方（在这个例子中是 3 的 10 次方）。连续的乘法从 1（实际上是 3 的 0 次方）开始，依次累乘以 3，由于使用的是半闭区间，从 0 开始到 9 的左闭右开区间，所以是执行 10 次。在循环的时候不需要知道实际执行到第一次了，而是要保证执行了正确的次数，因此这里不需要 index 的值。

同理我们可以使用 for-in 来循环遍历一个数组的元素

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

在遍历字典的时候，可以使用 key-value 对来进行遍历。每一个字典中的元素都是一个 (key, value) 元组，当遍历的时候，可以指定字段的 key 和 value 为一个特定的名称，这样在遍历的时候就可以更好地理解和使用它们，比如下面例子中的 animalName 和 legCount：

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

字典中的元素在遍历的时候一般不需要按照插入的顺序，因此不能保证遍历字典的时候，元素是有序的。更多跟数组和字典相关的内容可以参考：Collection Types

另外在数组和字典中也可以使用类似的遍历方式，如可以使用 for-in 循环来遍历字符串中的每一个字符：

```
for character in "Hello" {
    println(character)
}
// H
// e
// l
// l
// o
```

4.4.2 For-Condition-Increment 条件循环

Swift 同样支持 C 语言样式的 for 循环，它也包括了一个条件语句和一个增量语句：

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

下面是这种 for 循环的一般结构：

```
for initialization; condition; increment {
    statements
}
```

分号在这里用来分隔 for 循环的三个结构，和 C 语言一样，但是不需要用括号来包裹它们。

这种 for 循环的执行方式是：

1. 当进入循环的时候，初始化语句首先被执行，设定好循环需要的变量或常量
2. 测试条件语句，看是否满足继续循环的条件，只有在条件语句是 true 的时候才会继续执行，如果是 false 则会停止循环。
3. 在所有的循环体语句执行完毕后，增量语句执行，可能是对计数器的增加或者是减少，或者是其它的一些语句。然后返回步骤 2 继续执行。

这种循环方式还可以被描述为下面的形式：

```
initialization
while condition {
    statements
    increment
}
```

在初始化语句中被定义（比如 var index = 0）的常量和变量，只在 for 循环语句范围内有效。如果想要在循环执行之后继续使用，需要在循环开始之前就定义好：

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

需要注意的是，在循环执行完毕之后，index 的值是 3，而不是 2。因为是在 index 增 1 之后，条件语句 index < 3 返回 false，循环才终止，而这时，index 已经为 3 了。

4.5 while 循环

while 循环执行一系列代码块，直到某个条件为 false 为止。这种循环最长用于循环的次数不确定的情况。Swift 提供了两种 while 循环方式:

while 循环，在每次循环开始前测试循环条件是否成立

do-while 循环，在每次循环之后测试循环条件是否成立

4.5.1 while 循环

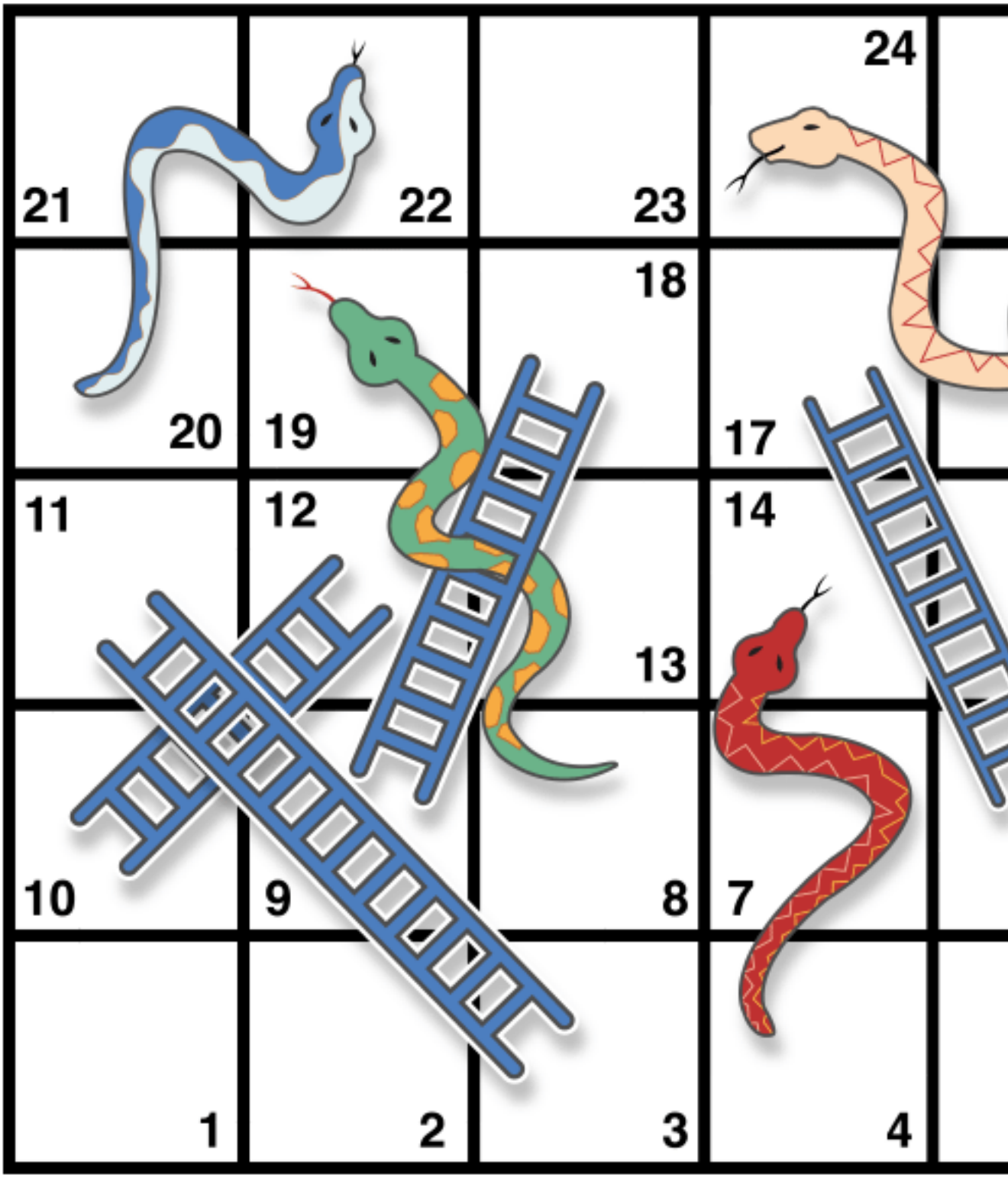
while 循环由一个条件语句开始，如果条件语句为 true，一直执行，直到条件语句变为 false。下面是一个 while 循环的一般形式:

```
while condition {
    statements
}
```

下面的例子是一个简单的游戏，Snakes and Ladders，蛇和梯子

游戏的规则是这样的:

- 游戏面板上有 25 个格子，游戏的目标是到达第 25 个格子;
- 每个回合通过一个 6 面的骰子来决定行走的步数，行走的路线按右图所示;
- 如果落在梯子的底部，那么就爬上那个梯子到达另外一个格子;
- 如果落到蛇的头部，就会滑到蛇尾部所在的格子。


24
21
22
23
18
20
19
17
11
12
14
13
10
9
8
7
1
2
3
4

游戏面板由一个 Int 数组组成，大小由一个常量设置 finalSquare，同时用来检测是否到达了胜利的格子。游戏面板由 26 个 Int 数字 0 初始化（不是 25 个，因为从 0 到 25 有 26 个数字）

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

其中一些格子被设置为一些特定的值用来表示蛇或者梯子。有梯子的地方是整数，而有蛇的地方是负数：

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

第三个格子是一个梯子的底部，表示玩家可以通过梯子到达第 11 格，因此设置 board3为 +08，表示前进 8 步。同理蛇的位置设置为负数，表示后退 i 步。

玩家从为 0 的格子开始游戏。

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

这个例子用到了一个非常简单的掷骰子的方式，就是每次加 1，而不是使用一个随机数。diceRoll 用来表示每次行走的步数，需要注意的是，每次执行前，++diceRoll 都会先执行加 1，然后再与 7 比较，如果等于 7 的话，就设置为 1，因此可以看出 diceRoll 的变化是 1,2,3,4,5,6,1……

在掷骰子之后，玩家移动 diceRoll 指示的步数，这时可能已经超过了 finalSquare，因此需要进行 if 判断，如果为 true 的话，执行该格子上的事件：如

果是普通格子就不动，如果是梯子或者蛇就移动相应的步数，这里只需要直接使用 square += board[square] 就可以了。

在 while 循环执行完毕之后，重新检查条件 square < finalSquare 是否成立，继续游戏直到游戏结束。

4.5.2 Do-while 循环

另一种 while 循环是 do-while 循环。在这种循环中，循环体中的语句会先被执行一次，然后才开始检测循环条件是否满足，下面是 do-while 循环的一般形式：

```
do {
    statements
} while condition
```

上面的蛇与梯子的游戏使用 do-while 循环来写可以这样完成。初始化语句和 while 循环的类似：

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

在这种循环中，第一个动作就是检测是否落在梯子或者蛇上，因为没有梯子或者蛇可以让玩家直接到达第 25 格，所以游戏不会直接结束，接下来的过程就和上面的 while 循环类似了，循环的条件语句还是检测是否已经达到最终格子。

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

4.6 条件语句

通常情况下我们都需要根据不同条件来执行不同语句。比如当错误发生的时候，执行一些错误信息的语句，告诉编程人员这个值是太大了还是太小了等等。这里就需要用到条件语句。

Swift 提供了两种条件分支语句的方式，if 语句和 switch 语句。一般 if 语句比较常用，但是只能检测少量的条件情况。switch 语句用于大量的条件可能发生时的条件语句。

4.6.1 if 语句

在最基本的 if 语句中，条件语句只有一个，如果条件为 true 时，执行 if 语句块中的语句：

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

上面这个例子检测温度是不是比 32 华氏度（32 华氏度是水的冰点，和摄氏度不一样）低，如果低的话就会输出一行语句。如果不低，则不会输出。if 语句块是用大括号包含的部分。

当条件语句有多种可能时，就会用到 else 语句，当 if 为 false 时，else 语句开始执行：

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a scarf.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// prints "It's not that cold. Wear a t-shirt."
```

在这种情况下，两个分支的其中一个一定会被执行。

同样也可以有多个分支，使用多次 if 和 else

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

上面这个例子中有多个 if 出现，用来判断温度是太低还是太高，最后一个 else 表示的是温度不高不低的时候。

当然 else 也可以被省掉

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

在这个例子中，温度不高不低的时候不会输入任何信息。

4.6.2 switch 语句

switch 语句考察一个值的多种可能性，将它与多个 case 相比较，从而决定执行哪一个分支的代码。switch 语句和 if 语句不同的是，它还可以提供多种情况同时匹配时，执行多个语句块。

switch 语句的一般结构是：

```
switch some value to consider {
    case value 1:
        respond to value 1
    case value 2, value 3:
         respond to value 2 or 3
    default:
         otherwise, do something else
}
```

每个 switch 语句包含有多个 case 语句块，除了直接比较值以外，Swift 还提供了多种更加复杂的模式匹配的方式来选择语句执行的分支，这在后续的小节会继续介绍。

在 switch 中，每一个 case 分支都会被匹配和检测到，所有 case 没有提到的情况都必须使用 default 关键词。注意 default 关键词必须在所有 case 的最后。

下面的例子用 switch 语句来判断一个字符的类型：

```
let someCharacter: Character = "e"
switch someCharacter {
    case "a", "e", "i", "o", "u":
        println("\(someCharacter) is a vowel")
    case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
        "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
        println("\(someCharacter) is a consonant")
    default:
        println("\(someCharacter) is not a vowel or a consonant")
}
// prints "e is a vowel"
```

在这个例子中，首先看这个字符是不是元音字母，再检测是不是辅音字母。其它的情况都用 default 来匹配即可。

4.6.3 不会一直执行

跟 C 和 Objective-C 不同，Swift 中的 switch 语句不会因为在 case 语句的结尾没有 break 就跳转到下一个 case 语句执行。switch 语句只会执行匹配上的 case 里的语句，然后就会直接停止。这样可以让 switch 语句更加安全，因为很多时候编程人员都会忘记写 break。

每一个 case 中都需要有可以执行的语句，下面的例子就是不正确的：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
    case "a":
    case "A":
        println("The letter A")
    default:
        println("Not the letter A")
```

```
}
// this will report a compile-time error
```

跟 C 不同，switch 语句不会同时匹配 a 和 A，它会直接报错。一个 case 中可以有多个条件，用逗号, 分隔即可：

```
switch some value to consider {
    case value 1,
         value 2:
         statements
}
```

4.6.4 范围匹配

switch 语句的 case 中可以匹配一个数值范围，比如：

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
    case 0:
        naturalCount = "no"
    case 1...3:
        naturalCount = "a few"
    case 4...9:
        naturalCount = "several"
    case 10...99:
        naturalCount = "tens of"
    case 100...999:
        naturalCount = "hundreds of"
    case 1000...999_999:
        naturalCount = "thousands of"
    default:
        naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \(countedThings).")
// prints "There are millions and millions of stars in the Milky Way."
```

4.6.5 元组

case 中还可以直接测试元组是否符合相应的条件，_ 可以匹配任意值。

下面的例子是判断 (x,y) 是否在矩形中，元组类型是 (Int,Int)

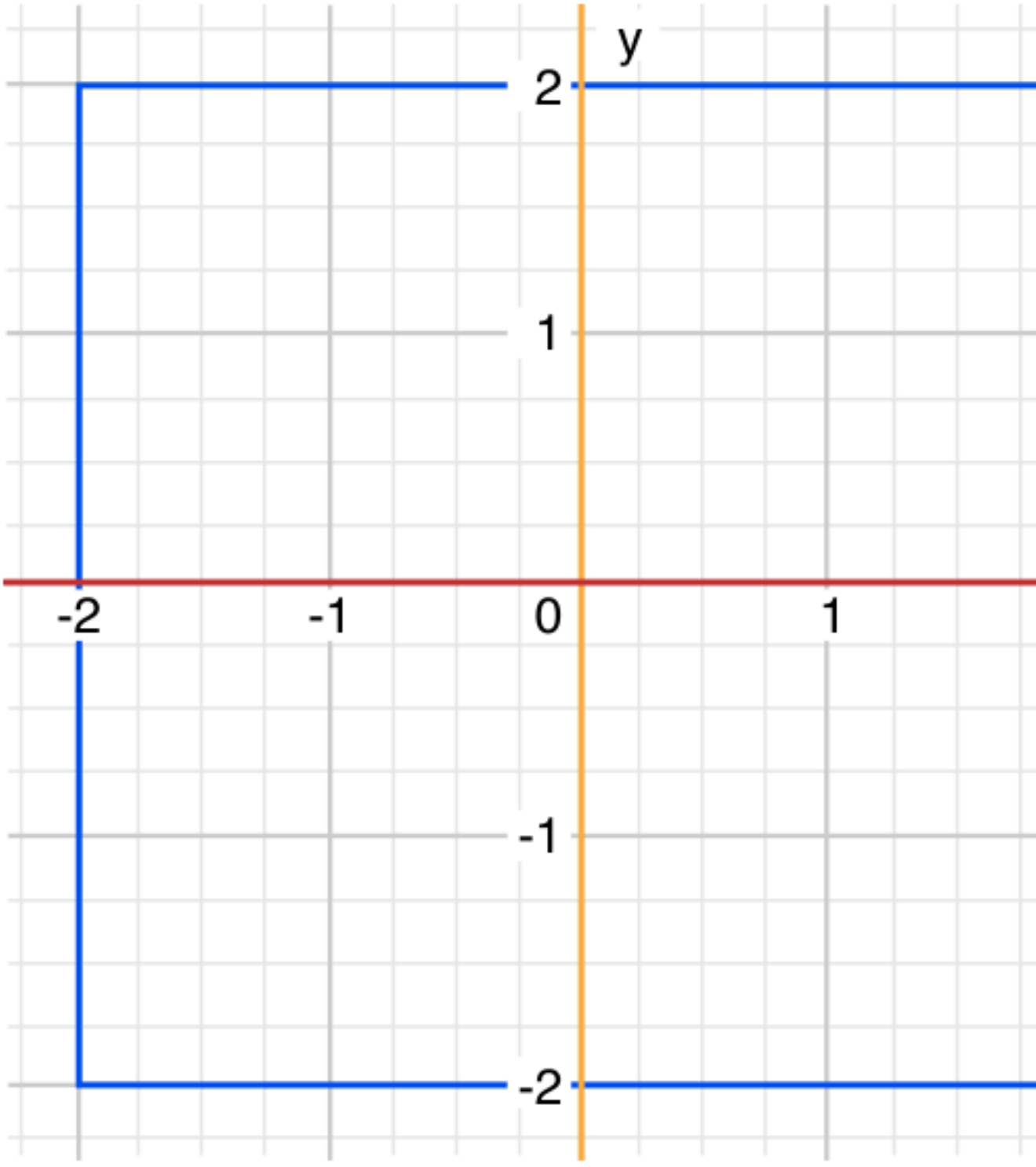
```
let somePoint = (1, 1)
switch somePoint {
    case (0, 0):
        println("(0, 0) is at the origin")
    case (_, 0):
        println("(\(somePoint.0), 0) is on the x-axis")
    case (0, _):
        println("(0, \(somePoint.1)) is on the y-axis")
    case (-2...2, -2...2):
       println("(\(somePoint.0), \(somePoint.1)) is inside the box")
    default:
     println("(\(somePoint.0), \(somePoint.1)) is outside of the box")
}
// prints "(1, 1) is inside the box"
```

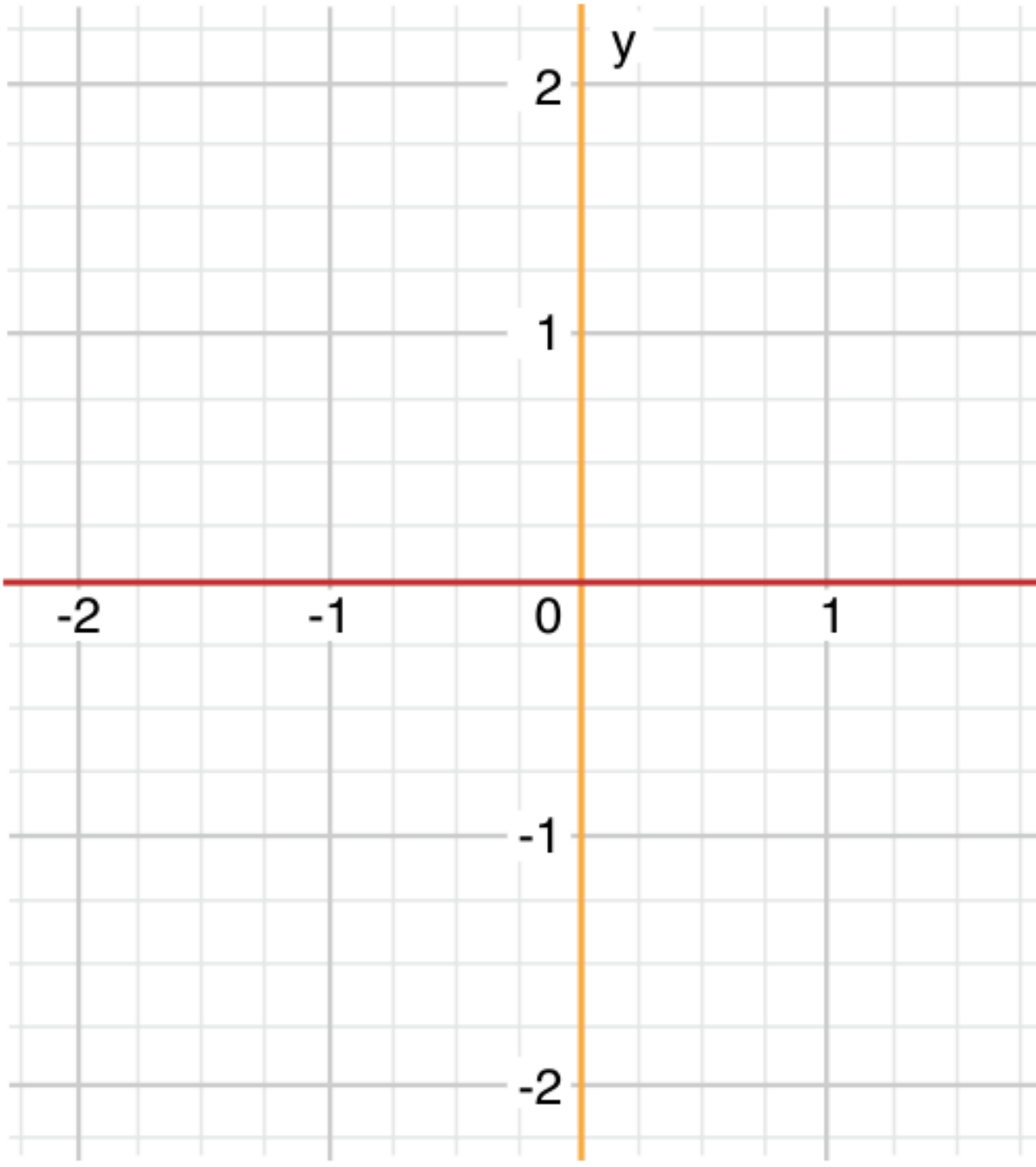
和 C 语言不同，Swift 可以判断元组是否符合条件。

4.6.6 数值绑定

在 case 匹配的同时，可以将 switch 语句中的值绑定给一个特定的常量或者变量，以便在 case 的语句中使用。比如:

```
let anotherPoint = (2, 0)
switch anotherPoint {
    case (let x, 0):
        println("on the x-axis with an x value of \(x)")
    case (0, let y):
        println("on the y-axis with a y value of \(y)")
    case let (x, y):
        println("somewhere else at (\(x), \(y))")
}
// prints "on the x-axis with an x value of 2"
```


y
2
1
-2
-1
0
1
-1
-2


y
2
1
-2
-1
0
1
-1
-2

switch 语句判断一个点是在 x 轴上还是 y 轴上，或者在其他地方。这里用到了匹配和数值绑定。第一种情况，如果点是 (x,0) 模式的，将值绑定到 x 上，这样在 case 语句中可以输出该值。同理如果在 y 轴上，就输出 y 的值。

4.6.7 Where 关键词

switch 语句可以使用 where 关键词来增加判断的条件，在下面的例子中：

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
    case let (x, y) where x == y:
        println("(\(x), \(y)) is on the line x == y")
    case let (x, y) where x == -y:
        println("(\(x), \(y)) is on the line x == -y")
    case let (x, y):
        println("(\(x), \(y)) is just some arbitrary point")
}
// prints "(1, -1) is on the line x == -y"
```

每个 case 都因为有 where 而不同，第一个 case 就是判断 x 是否与 y 相等，表示点在斜线 y=x 上。

4.7 控制跳转语句

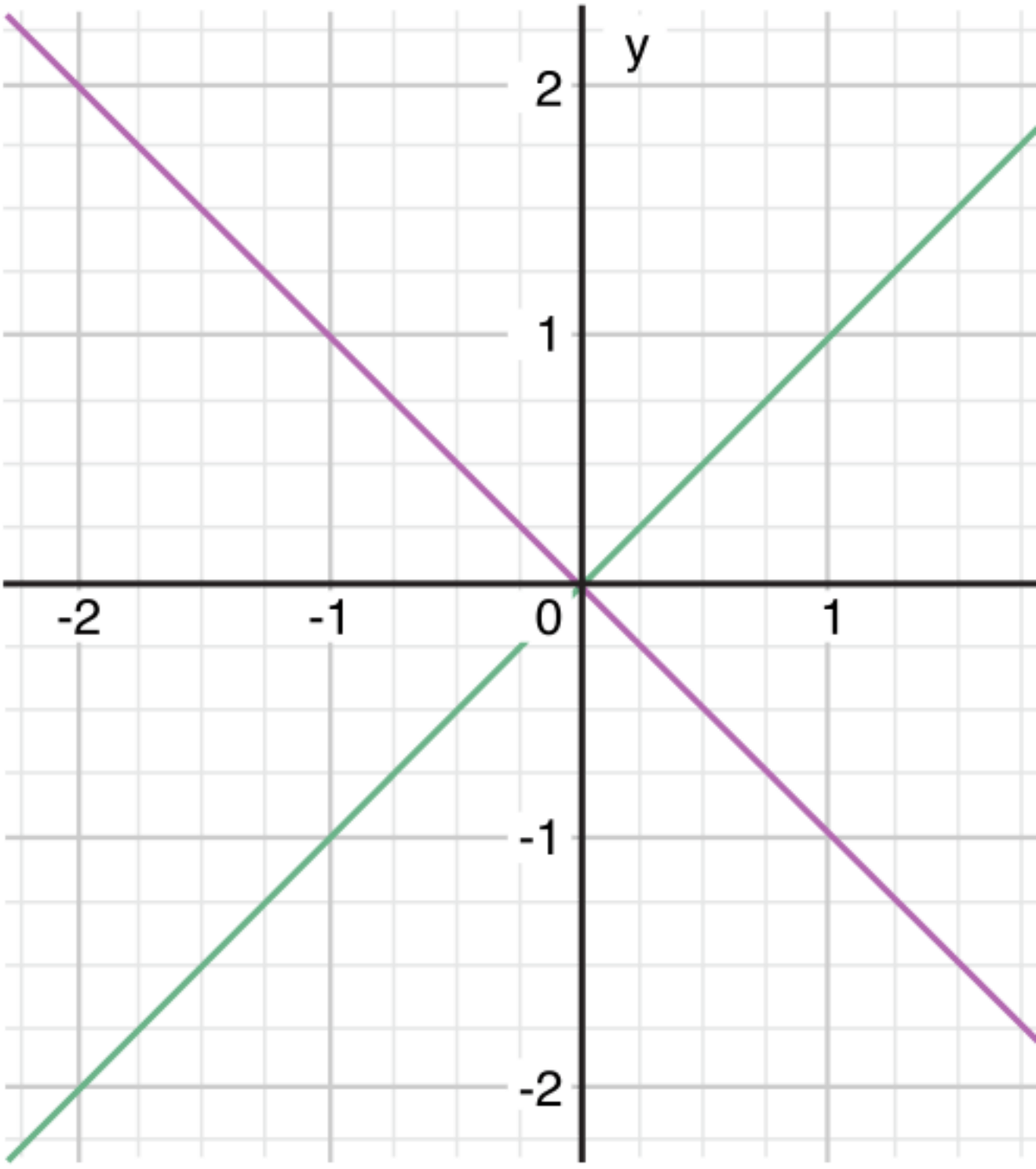
在 Swift 中控制跳转语句有 4 种，让编程人员更好地控制代码的流转，包括：

- continue
- break
- fallthrough
- return

其中 continue，break 和 fallthrough 在下面详细介绍，return 语句将在函数一章介绍。

4.7.1 continue

continue 语句告诉一个循环停止现在在执行的语句，开始下一次循环。


y
2
1
-2
-1
0
1
-1
-2

注意：在 for-condition-increment 循环中，increment 增量语句依然执行，只是略过了一次循环体。

下面的例子实现的是去除一个字符串中的空格和元音字母，从而组成一个字谜：

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
    switch character {
        case "a", "e", "i", "o", "u", " ":
            continue
        default:
            puzzleOutput += character
    }
}
println(puzzleOutput)
// prints "grtmndsthnklk"
```

遍历字符串的每一个字符，当遇到元音字母或者空格时就忽略，进行下一次循环，从而得到了最终的字谜。

4.7.2 break

break 语句将终止整个循环的执行，可以用在循环语句中，也可以用在 switch 语句中。

```
let numberSymbol: Character = "□" // Simplified Chinese for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
    case "1", "□", "□", "□":
        possibleIntegerValue = 1
    case "2", "□", "□", "□":
        possibleIntegerValue = 2
    case "3", "□", "□", "□":
        possibleIntegerValue = 3
    case "4", "□", "□", "□":
        possibleIntegerValue = 4
```

```
    default:
        break
}

if let integerValue = possibleIntegerValue {
   println("The integer value of \(numberSymbol) is \(integerValue).")
} else {
   println("An integer value could not be found for \(numberSymbol).")
}
// prints "The integer value of □ is 3."
```

上面的例子首先检查 numberSymbol 是不是一个数字，阿拉伯数字，汉字，拉丁文或者泰文都可以。如果匹配完成，则将 possibleIntegerValue 赋值。最后在通过 if 语句检测是否已被赋值，并绑定到 integerValue 常量上，最后输出。default 语句用来迎接未能被上述 case 匹配的情况，但是不需要做任何事情，因此直接使用 break 终止即可。

4.7.3 fallthrough

由于 Swift 中的 switch 语句不会自动的因为没有 break 而跳转到下一个 case，因此如果需要想 C 语言中那样，依次执行每个 case 的时候，就需要用到 fallthrough 关键词。

像下面这个例子一样，default 分支最终都会被执行:

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

4.7.4 标签语句

switch 和循环可以互相嵌套，循环之间也可以互相嵌套，因此在使用 break 或者 continue 的时候，需要知道到底是对哪个语句起作用。这就需要用到标签语句。标签语句的一般形式如下：

```
label name: while condition {
    statements
}
```

下面的例子演示了如何使用标签语句以及嵌套的循环和 switch。

依然采用之前的那个梯子与蛇的游戏，第一步依然是设置初始值：

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

然后，使用一个 while 循环与 switch 的嵌套来完成游戏

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
        case finalSquare:
        // diceRoll will move us to the final square, so the game is over
            break gameLoop
        case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so roll again
            continue gameLoop
        default:
            // this is a valid move, so find out its effect
            square += diceRoll
            square += board[square]
    }
}
println("Game over!")
```

在这个代码中，将游戏的循环命名为 gameLoop，然后在每一步移动格子时，判断当前是否到达了游戏终点，在 break 的时候，需要将整个游戏循环终止掉，而不是终止 switch，因此用到了 break gameLoop。同样的，在第二个分支中，continue gameLoop 也指明了需要 continue 的是整个游戏，而不是 switch 语句本身。# Swift 中文教程（六）函数

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

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

Swift 中的每个函数都有一个类型，包括函数的参数类型和返回类型。您可以方便的使用此类型像任何其他类型一样, 这使得它很容易将函数作为参数传递给其他函数, 甚至从函数中返回函数类型。函数也可以写在其他函数中来封装一个嵌套函数用以范围内有用的功能。

4.8 函数的声明与调用

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

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

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

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

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

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

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

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

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

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

为了简化这个函数的主体，结合消息创建和 return 语句用一行来表示：

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

4.9 函数的参数和返回值

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

4.9.1 多输入参数

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

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

4.9.2 无参函数

函数并没有要求一定要定义的输入参数。下面就一个没有输入参数的函数，任何时候调用时它总是返回相同的字符串消息：

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

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

4.9.3 没有返回值的函数

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

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

因为它并不需要返回一个值，该函数的定义不包括返回箭头（- >）和返回类型。

> 提示严格地说，sayGoodbye 功能确实还返回一个值，即使没有返回值定义。函数没有定义返回类型但返回了一个 void 返回类型的特殊值。它是一个简直是空的元组，实际上零个元素的元组，可以写为 ()。当一个函数调用时它的返回值可以忽略不计：

```
func printAndCount(stringToPrint: String) -> Int {
    println(stringToPrint)
    return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting("hello, world")
// prints "hello, world" but does not return a value
```

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

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

4.9.4 多返回值函数

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

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

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

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

```
let total = count("some arbitrary string!")
```

```
println("\(total.vowels) vowels and \(total.consonants) consonants")
// prints "6 vowels and 13 consonants"
```

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

4.10 函数参数名

所有上面的函数都为参数定义了参数名称:

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

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

4.10.1 外部参数名

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

如果你希望用户函数调用你的函数时提供参数名称, 除了设置本地地的参数名称，也要为每个参数定义外部参数名称。你写一个外部参数名称在它所支持的本地参数名称之前, 之间用一个空格来分隔:

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

> 注意如果您为参数提供一个外部参数名称，调用该函数时外部名称必须始终被使用。作为一个例子，考虑下面的函数，它通过插入他们之间的第三个”joiner” 字符串来连接两个字符串:

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

当你调用这个函数，你传递给函数的三个字符串的目的就不是很清楚了：

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

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

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

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

现在，您可以使用这些外部参数名称调用清楚明确的调用该函数：

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

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

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

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

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

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

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

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

4.10.2 参数的默认值

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

> 注意将使用默认值的参数放在函数的参数列表的末尾。这确保了所有调用函数的非默认参数使用相同的顺序, 并明确地表示在每种情况下相同的函数调用。这里有一个版本，是早期的 join 函数，并为参数 joiner 设置了默认值:

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

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

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

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

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

4.10.3 有默认值的外部名称参数

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

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

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

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

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

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

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

4.10.4 可变参数

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

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

下面的示例计算任意长度的数字的算术平均值（也称为平均）:

```
func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
```

```
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}

arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
// returns 10.0, which is the arithmetic mean of these three numbers
```

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

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

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

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

在参数名称前用关键字 var 定义变量参数:

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

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

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

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

> 注意一个变量参数的变化没有超出了每个调用函数, 所以对外部函数体是不可见的。变量参数只能存在于函数调用的生命周期里。

4.10.5 输入 -输出参数

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

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

4.11 函数类型

每个函数都有一个特定的类型，包括参数类型和返回值类型，比如:

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

这个例子定义了两个简单的数学函数 addTwoInts 和 multiplyTwoInts。每个函数接受两个 int 参数，返回一个 int 值，执行相应的数学运算然后返回结果

这两个函数的类型是 (Int, Int)->Int 可以解释为：

这个函数类型它有两个 int 型的参数，并返回一个 int 类型的值

下面这个例子是一个不带任何参数和返回值的函数：

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

这个函数的类型是 ()->()，或者函数没有参数，返回 void。函数没有显式地指定返回类型，默认为 void，在 Swift 中相当于一个空元组，记为 ()。

4.11.1 使用函数类型

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

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

可以解读为：

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

该 mathFunction 函数具有与 addTwoInts 函数相同类型的变量，所以这个赋值能通过 Swift 的类型检查。

现在你可以调用指定的函数名称为 mathFunction：

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

不同的函数相同的匹配类型可以分配给相同的变量, 也同样的适用于非函数性类型:

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

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

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

4.11.2 函数类型的参数

可以使用一个函数类型，如 (Int, Int)->Int 作为另一个函数的参数类型。这使你预留了一个函数的某些方面的实现，让调用者调用函数时提供。

下面就以打印上面的数学函数的结果为例:

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

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

当 printMathResult 被调用时，它传递 addTwoInt 函数，以及整数值 3 和 5。它使用 3 和 5，调用 addTwoInt 函数，并打印函数运行的结果 8。

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

4.12 函数类型的返回值

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

下面的例子定义了两个简单的函数，分别是 stepForward 和 stepBackward。stepForward 函数返回输入值自增 1，而 stepBackward 函数返回输入值自减 1。这两个函数都有一个相同的类型 (Int) -> Int:

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

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

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

您现在可以使用 chooseStepFunction 选择一个函数, 可能是加一函数或另一个:

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

□□□□□□□□□□□□□□□□□□□□□□□□currentValue□□□□□□□□currentV

□□moveNearerToZero□□□□□□□□,□□□□□□□□□:

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

4.13 嵌套函数

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

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

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

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

本文部分内容来源于CocoaChina的翻译小组，感谢他们的辛勤付出 ~

5 Swift 中文教程（七）闭包

闭包（Closures）是独立的函数代码块，能在代码中传递及使用。Swift 中的闭包与 C 和 Objective-C 中的代码块及其它编程语言中的匿名函数相似。

闭包可以在上下文的范围内捕获、存储任何被定义的常量和变量引用。因这些常量和变量的封闭性，而命名为“闭包（Closures）”。Swift 能够对所有你所能捕获到的引用进行内存管理。

NOTE 假如你对 “捕获（capturing）” 不熟悉，请不要担心，具体可以参考 Capturing Values（捕获值）。

- 全局函数和嵌套函数已在 Functions(函数) 中介绍过，实际上这些都是特殊的闭包函数
- 全局函数都是闭包，特点是有函数名但没有捕获任何值。
- 嵌套函数都是闭包，特点是有函数名，并且可以在它封闭的函数中捕获值。
- 闭包表达式都是闭包，特点是没有函数名，可以使用轻量的语法在它所围绕的上下文中捕获值。

Swift 的闭包表达式有着干净，清晰的风格，并常见情况下对于鼓励简短、整洁的语法做出优化。这些优化包括:

- 推理参数及返回值类型源自上下文
- 隐式返回源于单一表达式闭包
- 简约参数名
- 尾随闭包语法

5.1 闭包表达式

嵌套函数已经在 Nested Functions（嵌套函数）中有所介绍，是种方便命名和定义自包含代码块的一种方式，然而，有时候在编写简短函数式的构造器时非常有用，它不需要完整的函数声明及函数名，尤其是在你需要调用一个或多个参数的函数时。

闭包表达式是一种编写内联闭包的方式，它简洁、紧凑。闭包表达式提供了数种语义优化，为的是以最简单的形式编程而不需要大量的声明或意图。以下以同一个 sort 函数进行几次改进，每次函数都更加简洁，以此说明闭包表达式的优化。

5.1.1 Sort 函数

Swift 的标准函数库提供了一个名为 sort 的函数，它通过基于输出类型排序的闭包函数，给已知类型的数组数据的值排序。一旦完成排序工作，会返回一个同先前数组相同大小，相同数据类型，并且的新数组，并且这个数组的元素都在正确排好序的位置上。The closure expression examples below use the sort function to sort an array of String values in reverse alphabetical order. Here's the initial array to be sorted:

以下的闭包表达式通过 sort 函数将 String 值按字母顺序进行排序作说明，这是待排序的初始化数组。

let names = [“Chris”, “Alex”, “Ewa”, “Barry”, “Daniella”]

sort 函数需要两个参数：

- 一个已知值类型的数组
- 一个接收两个参数的闭包函数，这两个参数的数据类型都同于数组元素。并且返回一个 Bool 表明是否第一个参数应排在第二个参数前或后。

这个例子是一组排序的字符串值，因此需要排序的封闭类型的函数（字符串，字符串）-> Bool。

构造排序闭包的一种方式是书写一个符合其类型要求的普通函数:backwards，并将其返回值作为 sort 函数的第二个参数传入：

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

如果 backwards 函数参数 s1 大于 s2，则返回 true 值，表示在新的数组排序中 s1 应该出现在 s2 前。字符中的 “大于” 表示 “按照字母顺序后出现”。这意味着字母 “B” 大于字母 “A”, 字符串 “Tom” 大于字符串 “Tim”。其将进行字母逆序排序，”Barry” 将会排在 “Alex” 之后, 以此类推。

但这是一个相当冗长的方式，本质上只是做了一个简单的单表达式函数：(a > b)。下面的例子中，我们利用闭合表达式可以相比上面的例子更效率的构造一个内联排序闭包。

5.1.2 闭包表达式语法

闭合表达式语法具有以下一般构造形式：

```
{ (parameters) -> return type in
    statements
}
```

闭包表达式语法可以使用常量参数、变量参数和 inout 类型作为参数，但皆不可提供默认值。如果你需要使用一个可变的参数，可将可变参数放在最后，元组类型也可以作为参数和返回值使用。

下面的例子展示了上面的 backwards 函数对应的闭包表达式构造函数代码

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

需要注意的是声明内联闭包的参数和返回值类型与 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非常简约短所以完全可以将上面的 backwards 函数缩写成一行连贯的代码

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

可以看出 sort 函数的整体调用保持不变，还是一对圆括号包含两个参数变成了内联闭包形式、只不过第二个参数的值变成了。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)。

5.1.3 根据上下文推断类型

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

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

实际情况下，通过构造内联闭包表达式的闭包作为函数的参数传递给函数时，都可以判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。

同样，如果你希望避免阅读函数时可能存在的歧义，你可以直接明确参数的类型。

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。

5.1.4 单行表达式闭包可以省略 return

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

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

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

5.1.5 参数名简写

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

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

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

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。

5.1.6 运算符函数

运算符函数实际上是一个更短的方式构造以上的表达式。

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

更多关于运算符表达式的内容请查看 Operator Functions 。

5.2 Trailing 闭包

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

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

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
    // function body goes here
}
// here's how you call this function without using a trailing closure:
someFunctionThatTakesAClosure({
    // closure's body goes here
})
// here's how you call this function with a trailing closure instead:
someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

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

在上例中作为 sort 函数参数的字符串排序闭包可以改写为:

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

当闭包非常长以至于不能在一行中进行书写时，Trailing 闭包就变得非常有用。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”]:

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

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。

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

```
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
    number /= 10
}
return output
}
// strings is inferred to be of type String[]
// its value is ["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。

5.2.1 获取值

闭包可以在其定义的范围内捕捉（引用/得到）常量和变量，闭包可以引用和修改这些值，即使定义的常量和变量已经不复存在了依然可以修改和引用。牛逼吧、

在 Swift 中最简单形式是一个嵌套函数, 写在另一个函数的方法里面。嵌套函数可以捕获任何外部函数的参数，也可以捕获任何常量和变量在外部函数的定义。

看下面这个例子，一个函数方法为 makeIncrementor、这是一个嵌套函数，在这个函数体内嵌套了另一个函数方法：incrementor，在这个 incrementor 函数体内有两个参数：runningTotal 和 amount，实际运作时传进所需的两个参数后，incrementor 函数每次被调用时都会返回一个 runningTotal 值提供给外部的 makeIncrementor 使用：

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

而函数 makeincrementor 的返回类型值我们可以通过函数名后面的 () -> int 得知返回的是一个 Int 类型的值。如需想学习了解更多地函数返回类型，可以参考：Function Types as Return Types.（超链接跳转）

我们可以看见 makeincrementor 这个函数体内首先定义了一个整型变量：runningtotal，初始值为 0 ，而 incrementor() 函数最终运行的出来的返回值会赋值给这个整型变量。

makeincrementor 函数 () 中向外部抛出了一个 forIncrement 参数供外部穿参进来、一旦有值进入函数体内会被函数实例化替代为 amount，而 amount 会被传

递进内嵌的 incrementor 函数体中与整型常量 runningTotal 相加得到一个新的 runningTotal 并返回。而我们这个主函数要返回的值是 Int 类型，runningTotal 直接作为最终值被返回出去、makeincrementor 函数 () 执行完毕。

makeincrementor 函数 () 在其内部又定义了一个新的函数体 incrementor，作用就是将外部传递过来的值 amount 传进 incrementor 函数中与整形常量 runningTotal 相加得到一个新的 runningTotal,

单独的看 incrementor 函数、你会发现这个函数不寻常：

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

因为 incrementor 函数没有任何的参数，但是在它的函数方法体内却指向 runningTotal 和 amount，显而易见、这是 incrementor 函数获取了外部函数的值 amount，incrementor 不能去修改它但是却可以和体内的 runningTotal 相加得出新的 runningTotal 值返回出去。

不过，由于 runningtotal 每次被调用时都会相加改变一次实际值，相应地 incrementor 函数被调用时会去加载最新的 runningtotal 值，而不再是第一次舒适化的 0. 并且需要保证每次 runningTotal 的值在 makeIncrementor 函数体内不会丢失直到函数完全加载完毕。要能确保在函数体内下一次引用时上一次的值依然还在。

> 注意 Swift 中需要明确知道什么时候该引用什么时候该赋值，在 incrementor 函数中你不需要注解 amount 和 runningTotal。Swift 还负责处理当函数不在需要 runningTotal 的时候，内存应该如何去管理。

这里有一个例子 makeIncrementor 函数：

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

5.3 引用类型闭包

在上面的例子中，incrementBySeven 和 incrementByTen 是常量，但是这些常量在闭包的状态下依然可以被修改。为何？很简单，因为函数和闭包是引用类型。当你指定一个函数或一个闭包常量/变量时、实际上是在设置该常量或变量是否为一个引用函数。在上面的例子中，它是闭合的选择，incrementByTen 指的是恒定的，而不是封闭件本身的内容。这也意味着，如果你分配一个封闭两种不同的常量或变量，这两个常量或变量将引用同一个闭包：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50
```

6 Swift 中文教程（八）枚举类型

枚举定义了一个常用的具有相关性的一组数据，并在你的代码中以一个安全的方式使用它们。

如果你熟悉 C 语言，你就会知道，C 语言中的枚举指定相关名称为一组整数值。在 Swift 中枚举更为灵活，不必为枚举的每个成员提供一个值。如果一个值（被称为 “原始” 的值）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，或者任何整数或浮点类型的值。

另外，枚举成员可以指定任何类型，每个成员都可以存储的不同的相关值，就像其他语言中使用集合或变体。你还可以定义一组通用的相关成员为一个枚举，每一种都有不同的一组与它相关的适当类型的值的一部分。

在 Swift 中枚举类型是最重要的类型。它采用了很多以前只有类才具有的特性，如计算性能，以提供有关枚举的当前值的更多信息，方法和实例方法提供的功能相关的枚举表示的值传统上支持的许多功能。枚

举也可以定义初始化，以提供一个初始成员值; 可以在原有基础上扩展扩大它们的功能; 并使用协议来提供标准功能。

欲了解更多有关这些功能，请参见 Properties, Methods, Initialization, Extensions, Protocols

6.1 枚举语法

使用枚举 enum 关键词并把他们的整个定义在一对大括号内:

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

下面是一个指南针的四个点一个例子:

```
enum CompassPoint {
    case North
```

```
    case South
    case East
    case West
}
```

在枚举中定义的值（如 North，South，East 和 West）是枚举的成员值（或成员)。这个例子里 case 关键字表示成员值一条新的分支将被定义。

> Note 不像 C 和 Objective-C，Swift 枚举成员在创建时不分配默认整数值。在上面的例子 CompassPoints 中 North，South，Eath，West 不等于隐含 0，1，2 和 3，而是一种与 CompassPoint 明确被定义的类型却各不相同的值。

多个成员的值可以出现在一行上，用逗号分隔:

```
enum Planet {
   case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每个枚举定义中定义了一个全新的类型。像其他 Swift 的类型，它们的名称(如 CompassPoint 和 Planet）应为大写字母。给枚举类型单数而不是复数的名字，这样理解起来更加容易如:

```
var directionToHead = CompassPoint.West
```

使用 directionToHead 的类型时，用 CompassPoint 的一个可能值初始化的推断。一旦 directionToHead 被声明为一个 CompassPoint，您可以将其设置为使用更短的. 语法而不用再书写枚举 CompassPoint 值本身:

```
directionToHead = .East
```

directionToHead 的类型是已知的，所以你可以在设定它的值时，不写该类型。使用类型明确的枚举值可以让代码具有更好的可读性。

6.2 匹配枚举值与 **switch** 语句

你可以使用单个枚举值匹配 switch 语句:

```
directionToHead = .South
switch directionToHead {
    case .North:
        println("Lots of planets have a north")
    case .South:
        println("Watch out for penguins")
    case .East:
        println("Where the sun rises")
    case .West:
        println("Where the skies are blue")
}
// prints "Watch out for penguins"
```

你可以理解这段代码:

“考虑 directionToHead 的价值。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。

正如控制流所描述，Switch 语句考虑枚举的成员, 如果省略了 West 时，这段代码无法编译，因为它没有考虑 CompassPoint 成员的完整性。Switch 语句要求全面性确保枚举成员，避免不小心漏掉情况发生。

当它不需要为每一个枚举成员都匹配的情况下，你可以提供一个默认 default 分支来涵盖未明确提到的任何成员:

```
let somePlanet = Planet.Earth
switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
}
// prints "Mostly harmless"
```

6.3 关联值

在上一节中的示例延时了一个枚举的成员是如何被定义（分类）的。你可以为 Planet.Earth 设置一个常量或变量，然后在代码中检查这个值。但是，它有时是有用的才能存储其它类型的关联值除了这些成员的值。这让你随着成员值存储额外的自定义信息，并允许在你的代码中来使用该信息。

Swift 的枚举类型可以由一些数据类型相关的组成，如果需要的话，这些数据类型可以是各不相同的。枚举的这种特性跟其它语言中的奇异集合，标签集合或者变体相似

例如，假设一个库存跟踪系统需要由两种不同类型的条形码来跟踪产品。有些产品上标有 UPC-A 代码格式，它使用数字 0 到 9 的一维条码，每一个条码都有一个 “数字系统” 的数字，后跟十 “标识符” 的数字。最后一位是 “检查” 位，以验证代码已被正确扫描：

其他产品都贴有二维条码 QR 码格式，它可以使用任何的 ISO8859-1 字符，并可以编码字符串，最多 2,953 个字符：

这将是方便的库存跟踪系统能够存储 UPC-A 条码作为三个整数的元组，和 QR 代码的条形码的任何长度的字符串。

在 Swift 中可以使用一个枚举来定义两种类型的产品条形码，结构可以是这样的：

```
enum Barcode {
    case UPCA(Int, Int, Int)
```

```
    case QRCode(String)
}
```

这可以被理解为:

“定义一个名为条形码枚举类型，它可以是 UPC-A 的任一值类型的关联值 (Int，Int，Int)，或 QRCode 的一个类型为 String 的关联值。”

这个定义不提供任何实际的 Int 或 String 值，它只是定义了条形码常量和变量当等于 Barcode.UPCA 或 Barcode.QRCode 关联值的类型的时候的存储形式。

然后可以使用任何一种类型来创建新的条码:

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

此示例创建一个名为 productBarcode 新的变量，并与相关联的元组值赋给它 Barcode.UPCA 的值 (8，8590951226，3)。提供的 “标识符” 值都有整数加下划线的文字，85909_51226，使其更易于阅读的条形码。

同一产品可以分配不同类型的条形码:

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

在这一点上，原来 Barcode.UPCA 和其整数值被新的 Barcode.QRCode 及其字符串值代替。*条形码的常量和变量可以存储任何一个* UPCA 或 QRCode 的 (连同其关联值)，但它们只能存储其中之一在任何指定时间。

不同的条码类型像以前一样可以使用一个 switch 语句来检查，但是这一次相关的值可以被提取作为 switch 语句的一部分。您提取每个相关值作为常数 (let 前缀) 或变量 (var 前缀) 不同的情况下，在 switch 语句的 case 代码内使用：

```
switch productBarcode {
    case .UPCA(let numberSystem, let identifier, let check):
     println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
    case .QRCode(let productCode):
        println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

如果所有的枚举成员的关联值的提取为常数，或者当所有被提取为变量，为了简洁起见，可以放置一个 var，或 let 标注在成员名称前：

```
switch productBarcode {
    case let .UPCA(numberSystem, identifier, check):
     println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
    case let .QRCode(productCode):
        println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

6.4 原始值

在关联值的条形码的例子演示了一个枚举的成员如何能声明它们存储不同类型的关联值。作为替代关联值，枚举成员可以拿出预先填入缺省值 (称为原始值)，从而具有相同的类型。

这里是一个存储原始的 ASCII 值命名枚举成员的一个例子：

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

在这里，原始值被定义为字符类型的枚举叫做 ASCIIControlCharacter，并设置了一些比较常见的 ASCII 控制字符。字符值的字符串和字符的描述。

注意，原始值是不相同关联值。原始值设置为预填充的值时，应先在你的代码中定义枚举，像上述三个 ASCII 码。对于一个特定的枚举成员的原始值始终是相同的。当你创建一个基于枚举的常量或变量的新成员的关联值设置，每次当你这样做的时候可以是不同的。

原始值可以是字符串，字符，或任何整数或浮点数类型。每个原始值必须在它的枚举中唯一声明。当整数被用于原始值，如果其他 枚举成员没有值时，它们自动递增。

下面列举的是一个细化的早期 Planet 枚举，使用原始整数值来表示每个 Planet 的太阳系的顺序：

```
enum Planet: Int {
   case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

自动递增意味着 Planet.Venus 具有 2 的原始值，依此类推。

访问其 toRaw 方法枚举成员的原始值：

```
let earthsOrder = Planet.Earth.toRaw()
// earthsOrder is 3
```

使用枚举的 fromRaw 方法来试图找到一个特定的原始值枚举成员。这个例子识别 Uranus 的位置通过原始值为 7：

```
let possiblePlanet = Planet.fromRaw(7)
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

然而，并非所有可能的 Int 值都会找到一个匹配的星球。正因如此，该 fromRaw 方法返回一个可选的枚举成员。在上面的例子中，是 possiblePlanet 类型 Planet？或 “可选的 Planet”。

如果你试图找到一个 Planet 为 9 的位置，通过 fromRaw 返回可选的 Planet 值将是无：

```
let positionToFind = 9
if let somePlanet = Planet.fromRaw(positionToFind) {
```

```
    switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
    }
} else {
    println("There isn't a planet at position \(positionToFind)")
}
// prints "There isn't a planet at position 9"
```

这个范例使用 somePlanet= Planet.fromRaw(9) 来尝试访问可选集合 Planet，在可选 Planet 集合中设置检索条件 somePlanet，在原始值为 9 的情况下，不能检索到位置为 9 的星球，所有 else 分支被执行。

7 Swift 中文教程（九）类与结构

类与结构是编程人员在代码中会经常用到的代码块。在类与结构中可以像定义常量，变量和函数一样，定义相关的属性和方法以此来实现各种功能。

和其它的编程语言不太相同的是，Swift 不需要单独创建接口或者实现文件来使用类或者结构。Swift 中的类或者结构可以在单文件中直接定义，一旦定义完成后，就能够被直接其它代码使用。

注意：一个类的实例一般被视作一个对象，但是在 Swift 中，类与结构更像是一个函数方法，在后续的章节中更多地是讲述类和结构的功能性。

7.1 类和结构的异同

类和结构有一些相似的地方，它们都可以：

- 定义一些可以赋值的属性；
- 定义具有功能性的方法
- 定义下标，使用下标语法
- 定义初始化方法来设置初始状态
- 在原实现方法上的可扩展性
- 根据协议提供某一特定类别的基本功能

更多内容可以阅读：属性，方法，下标，初始化，扩展和协议等章节

类还有一些结构不具备的特性：

- 类的继承性
- 对类实例实时的类型转换
- 析构一个类的实例使之释放空间
- 引用计数，一个类实例可以有多个引用

更多内容可以阅读：继承，类型转换，初始化自动引用计数

> 注意：结构每次在代码中传递时都是复制了一整个，所以不要使用引用计数

7.1.1 定义语法

类和结构拥有相似的定义语法，使用 class 关键词定义一个类，struct 关键词定义结构。每个定义都由一对大括号包含：

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> 注意：在定义类和结构时，一般使用 UpperCamelCase 命名法来定义类和结构的名称，比如 SomeClass 和 SomeStructure，这样也符合 Swift 其它类型的标准。而给属性和方法命名时，一般时候 lowerCamelCase 命名法，比如 frameRate 和 incrementCount 等。

下面是一个结构和一个类的定义示例：

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
```

```
    var interlaced = falsevar
    frameRate = 0.0
    var name: String?
}
```

上面的例子首先定义了一个叫 Resolution 的结构，用来描述一个像素显示的分辨率，它有两个属性分别叫 width 和 height。这两个属性被默认定义为 Int 类型，初始化为 0.

之后定义了一个叫 VideoMode 的类，为视频显示的显示方式。这个类有四个属性，第一个属性 resolution 本身又是一个结构，然后是另外两个属性。最后一个属性用到了可选字符串类型 String?，表示这个属性可以存在，或者不存在为 nil。

7.1.2 类和结构的实例

上面的两个定义仅仅是定义了结构 Resolution 和类 VideoMode 的整体样式，它们本身不是一个特定的分辨率或者显示方式，这时候就需要实例化这个结构和类。

实例化的语法相似:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

类和结构都使用实例语法来完成实例化。最简单的实例语法就是用两个括号 () 完成。在这种情况下定义的实例中的属性都会完成默认初始化。更多内容可以参考初始化一章。

7.1.3 访问属性

使用. 语法就可以方便地访问一个实例的属性。在. 语法中，在实例名之后加上 (.) 再加上属性名即可，不需要空格:

```
println("The width of someResolution is \(someResolution.width)")
// prints "The width of someResolution is 0"
```

在这个例子中，someResolution.width 表示 someResolution 的 width 属性，返回了它的初始值 0

也可以使用. 语法连续地获取属性的属性，比如 VideoMode 中 resolution 属性的 width 属性

```
println("The width of someVideoMode is \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is 0"
```

使用这种方法不仅可以访问，也可以赋值:

```
someVideoMode.resolution.width = 1280
println("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is now 1280"
```

注意: 和 Objective-C 不同，Swift 能够直接设置一个结构属性的子属性，就像上面这个例子一样。

7.1.4 结构类型的成员初始化方法

每个结构都有一个成员初始化方法，可以在初始化的时候通过使用属性名称来指定每一个属性的初始值:

```
let vga = Resolution(width: 640, height: 480)
```

但是和结构不同，类实例不能够使用成员初始化方法，在初始化一章有专门的介绍。

7.2 结构和枚举类型是数值类型

数值类型是说当它被赋值给一个常量或者变量，或者作为参数传递给函数时，是完整地复制了一个新的数值，而不是仅仅改变了引用对象。

事实上读到这里你已经在前面几章见过数值类型了，所有 Swift 中的基础类型 -整型，浮点型，布尔类型，字符串，数组和字典都是数值类型。它们也都是由结构来实现的。

在 Swift 中所有的结构和枚举类型都是数值类型。这意味这你实例化的每个结构和枚举，其包含的所有属性，都会在代码中传递的时候被完整复制。

下面的这个例子可以说明这个特性:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

声明了一个常量 hd，是 Resolution 的实例化，宽度是 1920，高度是 1080，然后声明了一个变量 cinema，和 hd 相同。这个时候表明，cinema 和 hd 是两个实例，虽然他们的宽度都是 1920，高度都是 1080。

如果把 cinema 的宽度更改为 2048，hd 的宽度不会变化，依然是 1920

```
cinema.width = 2048
println("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
println("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

这表明当 hd 被赋值给 cinema 时，是完整地复制了一个全新的 Resolution 结构给 cinema，所以当 cinema 的属性被修改时，hd 的属性不会变化。

下面的例子演示的是枚举类型:

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
    println("The remembered direction is still .West")
}
// prints "The remembered direction is still .West"
```

尽管经过几次赋值，rememberedDirection 依然没有变化，这是因为在每一次赋值过程中，都是将数值类型完整地复制了过来。

7.3 类是引用类型

和数值类型不同引用类型不会复制整个实例，当它被赋值给另外一个常量或者变量的时候，而是会建立一个和已有的实例相关的引用来表示它。

下面是引用的示例，VideoMode 被定义为一个类:

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

分别将这个实例 tenEighty 的四个属性初始化，然后 tenEighty 被赋值给了另外一个叫 alsoTenEighty 的常量，然后 alsoTenEighty 的 frameRate 被修改了

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

由于类是一个引用类型，所以 tenEighty 和 alsoTenEighty 实际上是同一个实例，仅仅只是使用了不同的名称而已，我们通过检查 frameRate 可以证明这个问题:

```
println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// prints "The frameRate property of tenEighty is now 30.0"
```

注意到 tenEighty 和 alsoTenEighty 是被定义为常量的，而不是变量。但是我们还是可以改变他们的属性值，这是因为它们本身实际上没有改变，它们并没有保存这个 VideoMode 的实例，仅仅只是引用了一个 VideoMode 实例，而我们修改的也是它们引用的实例中的属性。

7.3.1 特征操作

因为类是引用类型，那么就可能存在多个常量或者变量只想同一个类的实例 (这对于数值类型的结构和枚举是不成立的)。

可以通过如下两个操作来判断两个常量或者变量是否引用的是同一个类的实例:

- 相同的实例 (===)
- 不同的实例 (!==)

使用这些操作可以检查:

if tenEighty === alsoTenEighty { println(“tenEighty and alsoTenEighty refer to the same Resolution instance.”) } // prints “tenEighty and alsoTenEighty refer to the same Resolution instance.”

注意是相同的实例判断使用三个连续的等号，这和相等（两个等号）是不同的

实例相同表示的是两个变量或者常量所引用的是同一个类的实例

相等是指两个实例在数值上的相等，或者相同。

当你定义一个类的时候，就需要说明什么样的时候是两个类相等，什么时候是两个类不相等。更多内容可以从相等操作一章中获得。

7.3.2 指针

如果你有 C，C++ 或者 Objective-C 的编程经验，你一定知道在这些语言中使用指针来引用一个内存地址。Swift 中引用一个实例的常量或变量跟 C 中的指针类似，但是不是一个直接指向内存地址的指针，也不需要使用 * 记号表示你正在定义一个引用。Swift 中引用和其它变量，常量的定义方法相同。

7.3.3 如何选择使用类还是结构

在代码中可以选择类或者结构来实现你所需要的代码块，完成相应的功能。但是结构实例传递的是值，而类实例传递的是引用。那么对于不同的任务，应该考虑到数据结构和功能的需求不同，从而选择不同的实例。

一般来说，下面的一个或多个条件满足时，应当选择创建一个结构：

- 结构主要是用来封装一些简单的数据值
- 当赋值或者传递的时候更希望这些封装的数据被赋值，而不是被引用过去
- 所有被结构存储的属性本身也是数值类型
- 结构不需要被另外一个类型继承或者完成其它行为

一些比较好的使用结构的例子：

- 一个几何形状的尺寸，可能包括宽度，高度或者其它属性，每个属性都是 Double 类型的
- 一个序列的对应关系，可能包括开始 start 和长度 length 属性，每个属性都是 Int 类型的
- 3D 坐标系中的一个点，包括 x，y 和 z 坐标，都是 Double 类型

在其它情况下，类会是更好的选择。也就是说一般情况下，自定义的一些数据结构一般都会被定义为类。

7.4 集合类型的赋值和复制操作

Swift 中，数组 Array 和字典 Dictionary 是用结构来实现的，但是数组与字典和其它结构在进行赋值或者作为参数传递给函数的时候有一些不同。

并且数组和字典的这些操作，又与 Foundation 中的 NSArray 和 NSDictionary 不同，它们是用类来实现的。

> 注意：下面的小节将会介绍数组，字典，字符串等的复制操作。这些复制操作看起来都已经发生，但是 Swift 只会在确实需要复制的时候才会完整复制，从而达到最优的性能。

7.4.1 字典的赋值和复制操作

每次将一个字典 Dictionary 类型赋值给一个常量或者变量，或者作为参数传递给函数时，字典会在赋值或者函数调用时才会被复制。这个过程在上面的小节：结构和枚举是数值类型中描述了。

如果字典中的键值是数值类型（结构或者枚举），它们在赋值的时候会同时被复制。相反，如果是引用类型（类或者函数），引用本身将会被复制，而不是类实例或者函数本身。字典的这种复制方式和结构相同。

下面的例子演示的是一个叫 ages 的字典，存储了一些人名和年龄的对应关系，当赋值给 copiedAges 的时候，里面的数值同时被完整复制。当改变复制了的数值的时候，原有的数值不会变化，如下例子：

```
var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]
var copiedAges = ages
```

这个字典的键是字符串 String 类型，值是 Int 类型，都是数值类型，那么在赋值的时候都会被完整复制。

```
copiedAges["Peter"] = 24
println(ages["Peter"])
// prints "23"
```

7.4.2 数组的赋值和复制操作

和字典 Dictionary 类型比起来，数组 Array 的赋值和复制操作就更加复杂。Array 类型和 C 语言中的类似，仅仅只会在需要的时候才会完整复制数组的值。

如果将一个数组赋值给一个常量或者变量，或者作为一个参数传递给函数，复制在赋值和函数调用的时候并不会发生。这两个数组将会共享一个元素序列，如果你修改了其中一个，另外一个也将会改变。

对于数组来说，复制只会在你进行了一个可能会修改数组长度操作时才会发生。包括拼接，添加或者移除元素等等。当复制实际发生的时候，才会像字典的赋值和复制操作一样。

下面的例子演示了数组的赋值操作：

```
var a = [1, 2, 3]
var b = a
var c = a
```

数组 a 被赋值给了 b 和 c，然后输出相同的下标会发现：

```
println(a[0])
// 1
println(b[0])
// 1
println(c[0])
// 1
```

如果改变 a 中的某个值，会发现 b 和 c 中的数值也会跟着改变，因为赋值操作没有改变数组的长度：

```
a[0] = 42
println(a[0])
// 42
println(b[0])
// 42
println(c[0])
// 42
```

但是，如果在 a 中添加一个新的元素，那么就改变了数组的长度，这个时候就会发生实际的复制操作。如果再改变 a 中元素的值，b 和 c 中的元素将不会发生改变：

```
a.append(4)
a[0] = 777
println(a[0])
// 777
println(b[0])
// 42
println(c[0])
// 42
```

7.4.3 设置数组是唯一的

如果可以在对数组进行修改前，将它设置为唯一的就最好了。我们可以通过使用 unshare 方法来将数组自行拷贝出来，成为一个唯一的实体。

如果多个变量引用了同一个数组，可以使用 unshare 方法来完成一次“独立”

```
b.unshare()
```

这时候如果再修改 b 的值，c 的值也不会再受影响

```
b[0] = -105
println(a[0])
// 777
println(b[0])
// -105
println(c[0])
// 42
```

检查两个数组时候共用了相同的元素

使用实例相等操作符来判断两个数组是否共用了元素（=== 和!===）

下面这个例子演示的就是判断是否共用元素：

```
if b === c {
    println("b and c still share the same array elements.")
} else {
   println("b and c now refer to two independent sets of array elements.")
}
// prints "b and c now refer to two independent sets of array elements."
```

也可以使用这个操作来判断两个子数组是否有共用的元素：

```
if b[0...1] === b[0...1] {
    println("These two subarrays share the same elements.")
} else {
    println("These two subarrays do not share the same elements.")
}
// prints "These two subarrays share the same elements."
```

7.4.4 强制数组拷贝

通过调用数组的 copy 方法来完成强制拷贝。这个方法将会完整复制一个数组到新的数组中。

下面的例子中这个叫 names 的数组会被完整拷贝到 copiedNames 中去。

```
var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]
var copiedNames = names.copy()
```

通过改变 copiedNames 的值可以验证，数组已经被完整拷贝，不会影响到之前的数组：

```
copiedNames[0] = "Mo"
println(names[0])
// prints "Mohsen"
```

> 注意：如果你不确定你需要的数组是否是独立的，那么仅仅使用 unshare 就可以了。而 copy 方法不管当前是不是独立的，都会完整拷贝一次，哪怕这个数组已经是 unshare 的了。

8 Swift 中文教程（十）属性

属性是描述特定类、结构或者枚举的值。存储属性作为实例的一部分存储常量与变量的值，而计算属性计算他们的值（不只是存储）。计算属性存在于类、结构与枚举中。存储属性仅仅只在类与结构中。

属性通常与特定类型实例联系在一起。但属性也可以与类型本身联系在一起，这样的属性称之为类型属性。

另外，可以定义属性观察者来处理属性值发生改变的情况，这样你就可以对用户操作做出反应。属性观察者可以被加在自己定义的存储属性之上，也可以在从父类继承的子类属性之上。

8.1 存储属性

最简单的情形，作为特定类或结构实例的一部分，存储属性存储着常量或者变量的值。存储属性可分为变量存储属性（关键字 var 描述）和常量存储属性(关键字 let 描述)。

当定义存储属性时，你可以提供一个默认值，这些在“默认属性值”描述。在初始化过程中你也可以设置或改变存储属性的初值。这个准则对常量存储属性也同样适用（在“初始化过程中改变常量属性”描述)

下面的例子定义了一个叫 FixedLengthRange 的结构，它描述了一个一定范围内的整数值，当创建这个结构时，范围长度是不可以被改变的：

```
struct FixedLengthRange {
var firstValue: Int
let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

FixedLengthRange 的实例包含一个名为 firstValue 的变量存储属性和名为 length 的常量存储属性。以上的例子中，当范围确定，length 被初始化之后它的值是不可以被改变的

8.1.1 常量结构实例的存储属性

如果你创建一个结构实例，并将其赋给一个常量，这个实例中的属性将不可以被改变，即使他们被声明为变量属性

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even thought firstValue is a variable property
```

因为 rangeOfFourItems 是一个常量（let），即便 firstValue 是一个变量属性，它的值也是不可以被改变的

这样的特性是因为结构是值类型。当一个值类型实例作为常量而存在，它的所有属性也作为常量而存在。

而这个特性对类并不适用，因为类是引用类型。如果你将引用类型的实例赋值给常量，依然能够改变实例的变量属性。

8.1.2 Lazy Stored Properties（懒惰存储属性?）

懒惰存储属性是当它第一次被使用时才进行初值计算。通过在属性声明前加上 @lazy 来标识一个懒惰存储属性。

> 注意必须声明懒惰存储属性为变量属性（通过 var），因为它的初始值直到实例初始化完成之后才被检索。常量属性在实例初始化完成之前就应该被赋值，因此常量属性不能够被声明为懒惰存储属性。

当属性初始值因为外部原因，在实例初始化完成之前不能够确定时，就要定义成懒惰存储属性。当属性初始值需要复杂或高代价的设置，在它需要时才被赋值时，懒惰存储属性就派上用场了。

下面的例子使用懒惰存储属性来防止类中不必要的初始化操作。它定义了类 DataImporter 和类 DataManager:

```
class DataImporter {
   /*DataImporter is a class to import data from an external file.    The class is assu
    var fileName = "data.txt"
   // the DataImporter class would provide data importing functionality here
}
class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
   // the DataManager class would provide data management functionality here
}
let manager = DataManager()
manager.data += "Some data"
manager.data += "Some more data"
// the DataImporter instance for the importer property has not yet been created
```

类 DataManager 有一个称为 data 的存储属性，它被初始化为一个空的 String 数组。虽然 DataManager 定义的其它部分并没有写出来，但可以看出 DataManager 的目的是管理 String 数据并为其提供访问接口。

DataManager 类的部分功能是从文件中引用数据。这个功能是由 DataImporter 类提供的，这个类需要一定的时间来初始化，因为它的实例需要打开文件并见内容读到内存中。

因为 DataManager 实例可能并不需要立即管理从文件中引用的数据，所以在 DataManager 实例被创建时，并不需要马上就创建一个新的 DataImporter 实例。这就使得当 DataImporter 实例在需要时才被创建理所当然起来。

因为被声明为 @lazy 属性，DataImporter 的实例 importer 只有在当它在第一次被访问时才被创建。例如它的 fileName 属性需要被访问时：

```
println(manager.importer.fileName)
// the DataImporter instance for the importer property has now been created
// prints "data.txt
```

8.1.3 存储属性与实例变量

如果你使用过 Objective − C，你应该知道它提供两种方式来存储作为类实例一部分的值与引用。除了属性，你可以使用实例变量作为属性值的后备存储

Swift 使用一个单一属性声明来统一这些概念。一个 Swift 属性没有与之相符的实例变量，并且属性的后备存储也不能直接访问。这防止了在不通上下文中访问值的混淆，并且简化属性声明成为一个单一的、最终的语句。关于属性的所有信息－包含名称、类型和内存管理等－作为类型定义的一部分而定义。

8.2 计算属性

除了存储属性，类、结构和枚举能够定义计算属性。计算属性并不存储值，它提供 getter 和可选的 setter 来间接地获取和设置其它的属性和值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
```

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
println("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// prints "square.origin is now at (10.0, 10.0)"
```

这个例子定义了三个处理几何图形的结构:

- Point 包含一个（x，y）坐标
- Size 包含宽度 width 和高度 height
- Rect 定义了一个长方形，包含原点和大小 size

Rect 结构包含一个称之为 center 的计算属性。Rect 当前中心点的坐标可以通过 origin 和 size 属性得来，所以并不需要显式地存储中心点的值。取而代之的是，Rect 定义一个称为 center 的计算属性，它包含一个 get 和一个 set 方法，通过它们来操作长方形的中心点，就像它是一个真正的存储属性一样。

例子中定义了一个名为 square 的 Rect 变量，它的中心点初始化为（0，0），高度和宽度初始化为 10，由以下图形中的蓝色正方形部分。

变量 square 的 center 属性通过点操作符访问，它会调用 center 的 getter 方法。不同于直接返回一个存在的值，getter 方法要通过计算才能返回长方形的中心点的值（point）。以上的例子中，getter 方法返回中心点（5，5）。

然后 center 属性被设置成新的值（15，15），这样就把这个正方形向右向上移动到了途中黄色部分所表示的新的位置。通过调用 setter 方法来设置 center，

改变 origin 中坐标 x 和 y 的值，将正方形移动到新的位置。

8.2.1 setter 声明的简略写法

如果计算属性的 setter 方法没有将被设置的值定义一个名称，将会默认地使用 newValue 这个名称来代替。下面的例子采用了这样一种特性，定义了 Rect 结构的新版本：

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

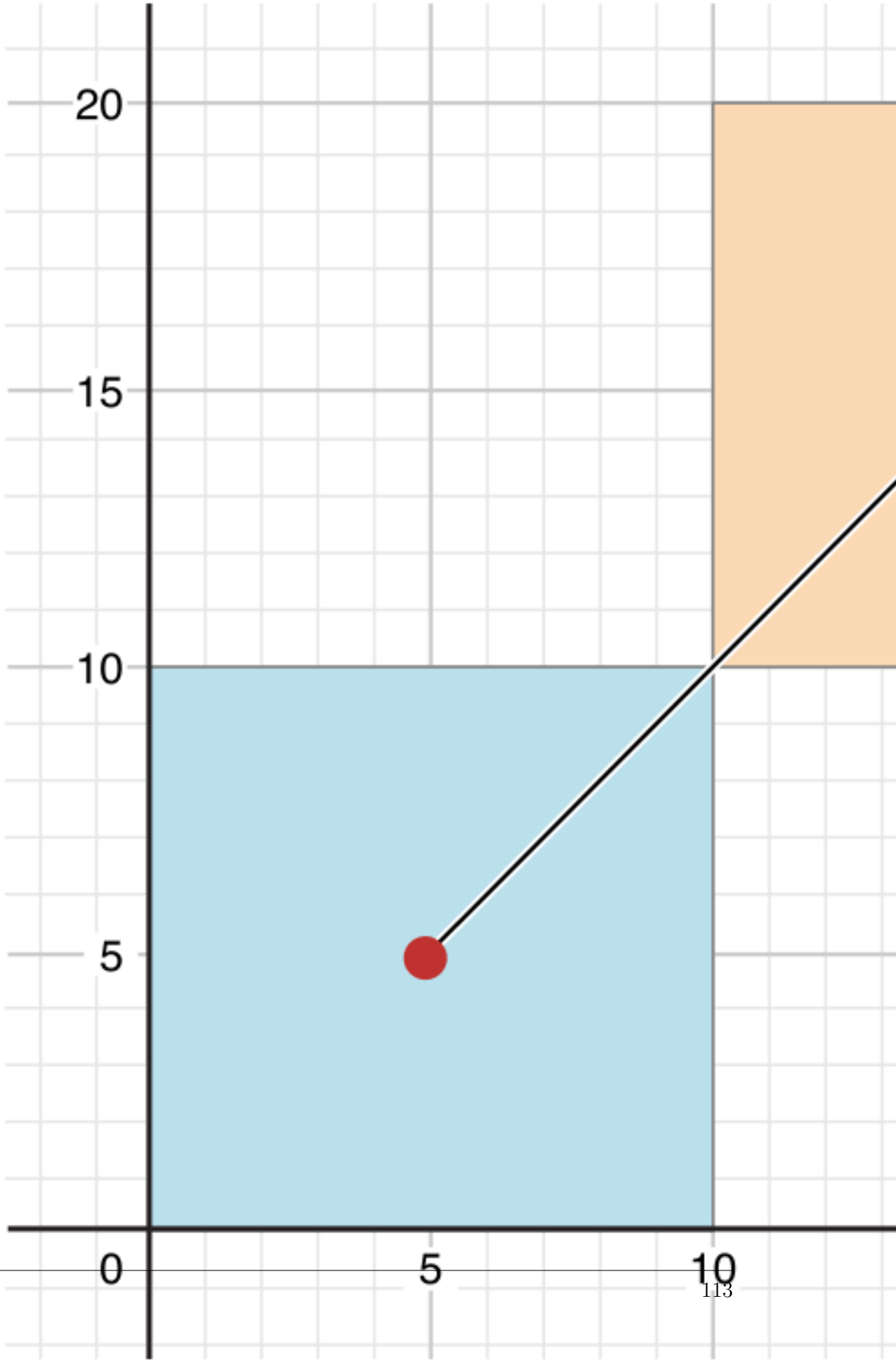
8.2.2 只读计算属性

只读计算属性只带有一个 getter 方法，通过点操作符，可以放回属性值，但是不能修改它的值。

> 注意应该使用 var 关键字将计算属性－包含只读计算属性－定义成变量属性，因为它们的值并不是固定的。let 关键字只被常量属性说使用，以表明一旦被设置它们的值就是不可改变的了

通过移除 get 关键字和它的大括号，可以简化只读计算属性的定义：

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
```


20
15
10
5
0
5
10

```
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
println("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// prints "the volume of fourByFiveByTwo is 40.0
```

这个例子定义了一个三维长方体结构 Cuboid，包含了长宽高三个属性，和一个表示长方体容积的只读计算属性 volume。volume 值是不可被设置的，因为它直接由长宽高三个属性计算而来。通过提供这样一个只读计算属性，Cuboid 使外部用户能够访问到其当前的容积值。

8.3 属性观察者

属性观察者观察属性值的改变并对此做出响应。当设置属性的值时，属性观察者就被调用，即使当新值同原值相同时也会被调用。

除了懒惰存储属性，你可以为任何存储属性加上属性观察者定义。另外，通过重写子类属性，也可以继承属性（存储或计算）加上属性观察者定义。属性重写在“重写”章节定义。

> 注意不必为未重写的计算属性定义属性观察者，因为可以通过它的 setter 方法直接对值的改变做出响应

定义属性的观察者时，你可以单独或同时使用下面的方法：

- willSet：设置值前被调用
- didSet：设置值后立刻被调用

当实现 willSet 观察者时，新的属性值作为常量参数被传递。你可以为这个参数起一个名字，如果不的话，这个参数就默认地被命名成 newValue。

在实现 didSet 观察者时也是一样，只不过传递的产量参数表示的是旧的属性值。

> 注意：属性初始化时，willset 和 didSet 并不会被调用。只有在初始化上下文之外，当设置属性值时才被调用

下面是一个 willSet 和 didSet 用法的实例。定义了一个类 StepCounter，用来统计人走路时的步数。它可以从计步器或其它计数器上获取输入数据，对日常联系锻炼的步数进行追踪。

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            println("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                println("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

类 StepCounter 声明了一个 Int 类型的、含有 willSet 和 didSet 观察者的存储属性 totalSteps。当这个属性被赋予新值时，willSet 和 didSet 将会被调用，即使新值和旧值是相同的。

例子中的 willSet 观察者为参数起了个新的名字 newTotalSteps，它简单地打印了即将被设置的值。

当 totalSteps 值被更新时，didSet 观察者被调用，它比较 totalSteps 的新值和旧值，如果新值比旧值大，就打印所增加的步数。didSet 并没有为旧值参数命名，在本例中，将会使用默认的名字 oldValue 来表示旧的值。

> 注意如果通过 didSet 来设置属性的值，即使属性值刚刚被设置过，起作用的也将会是 didSet，即新值是 didSet 设置的值

8.4 全局和局部变量

以上所写的关于计算与观察属性值的特性同样适用于全局和局部变量。全局变量是在任何函数、方法、闭包、类型上下文外部定义的变量，而局部变量是在函数、方法、闭包中定义的变量。

前面章节所遇到过的全局、局部变量都是存储变量。和存储属性一样，存储变量为特定类型提供存储空间并且可以被访问

但是，你可以在全局或局部范围定义计算变量和存储变量观察者。计算变量并不存储值，只用来计算特定值，它的定义方式与计算属性一样。

> 注意全局常量和变量通常是延迟计算的，跟懒惰存储属性一样，但是不需要加上 @lazy。而局部常量与变量不是延迟计算的。

8.5 类型属性

实例属性是特定类型实例的属性。当创建一个类型的实例时，这个实例有自己的属性值的集合，这将它与其它实例区分开来。

也可以定义属于类型本身的属性，即使创建再多的这个类的实例，这个属性也不属于任何一个，它只属于类型本身，这样的属性就称为类型属性。

类型属性适用于定义那些特定类型实例所通用的属性，例如一个可以被所有实例使用的常量属性（就像 c 中的静态常量），或者变量属性（c 中的静态变量）。

可以为值类型（结构、枚举）定义存储类型属性和计算类型属性。对类而言，只能够定义计算类型属性。

值类型的存储类型属性可以是常量也可以是变量。而计算类型属性通常声明成变量属性，类似于计算实例属性

> 注意不想存储实例属性，你需要给存储类型属性一个初始值。因为类型本身在初始化时不能为存储类型属性设置值

8.5.1 类型属性句法

在 C 和 Objective − C 中，定义静态常量、变量和全局静态变量一样。但是在 swift 中，类型属性的定义要放在类型定义中进行，在类型定义的大括号中，显示地声明它在类型中的作用域。

对值类型而言，定义类型属性使用 static 关键字，而定义类类型的类型属性使用 class 关键字。下面的例子展示了存储和计算类型属性的用法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        // return an Int value here
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
   static var computedTypeProperty: Int {    // return an Int value here
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
        // return an Int value here
    }
}
```

注意上面的例子是针对只读计算类型属性而言的，不过你也可以像计算实例属性一样定义可读可写的计算类型属性

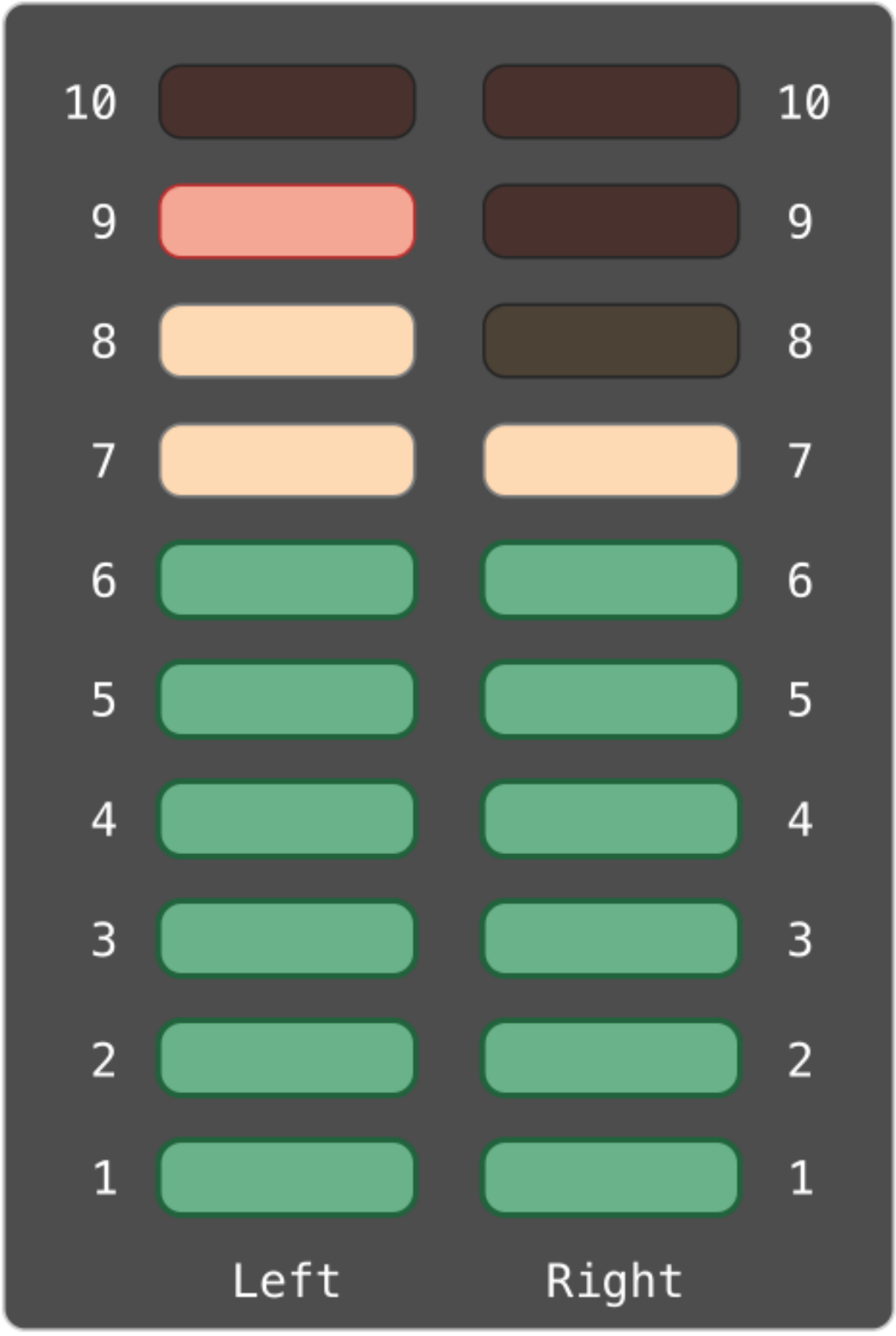
8.5.2 查询与设置类型属性

像实例属性一样，类型属性通过点操作符来查询与设置。但是类型属性的查询与设置是针对类型而言的，并不是针对类型的实例。例如：

```
println(SomeClass.computedTypeProperty)
// prints "42"
println(SomeStructure.storedTypeProperty)
// prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// prints "Another value.
```

下面的例子在一个结构中使用两个存储类型属性来展示一组声音通道的音频等级表。每个通道使用 0 到 10 来表示声音的等级。

从下面的图表中可以看出，使用了两组声音通道来表示一个立体声音频等级表。当一个通道的等级为 0 时，所有的灯都不会亮，当等级为 10 时，所有的灯都会亮。下面的图中，左边的通道表示声音等级为 9，右边的为 7


10
10
9
9
8
8
7
7
6
6
5
5
4
4
3
3
2
2
1
1
Left
Right

上述的声音通道由以下的 AudioChannel 结构实例来表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                //cap the new audio level to the threshold level
                currentLevel = AudioChannel.thresholdLevel
            }
         if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // store this as the new overall maximum input level
             AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

AudioChannel 结构定义了两个存储类型属性。thresholdLevel 定义了音频所能达到的最高等级，对所有的 AudoChannel 实例而言，是个值为 10 的常量。当一个声音信号的值超过 10 时，会被截断为其阈值 10。

第二个类型属性是一个变量存储属性 maxInputLevelForAllChannels。它保存了当前所有 AudioChannel 实例中所接受到声音的最高等级，它被初始化为 0。

结构还定义了一个存储实例属性 currentLevel，表示当前的通道声音等级。这个属性使用 didSet 属性观察者来检测 currentLevel 的改变。这个观察者执行两道检查：

- 如果 currentlevel 的新值比阈值 thresholdLevel 大，currentLevel 将被设置成 thresholdLevel
- 如果 currentLevel 的新值比所有 AudioChannel 实例之前接受到的最大声音等级还要大，那么 maxInputLevelForAllChannles 将会被设置成 cueentLevel 大值。

  注意第一道检查中，didSet 为 currentLevel 设置了新值。这并不会造成观察者再次被调用

可以创建两个 AudioChannel 实例，leftChannel 和 rightChannel，来表示一个立体声系统：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

如果设置左通道的 currentLevel 为 7, 它的类型属性 maxInputLevelForAllChannels 将更新成为 7:

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
// prints "7"
println(AudioChannel.maxInputLevelForAllChannels)
// prints "7 ”
```

如果像设置右通道的 currentlevel 为 11，它的值将被截短成为 10，而且 maxInputLevelForAllChannels 的值也将更新为 10:

```
“ rightChannel.currentLevel = 11
println(rightChannel.currentLevel)
// prints "10"
println(AudioChannel.maxInputLevelForAllChannels)
// prints "10"
```

9 Swift 中文教程（十一）方法

方法是关联到一个特定类型的函数。类、结构、枚举所有可以定义实例方法, 封装特定任务和功能处理给定类型的一个实例。类、结构、枚举类型还可以定义方法, 相关的类型本身。类型方法类似于 objective – c 类方法。

结构和枚举可以定义方法 swift 与 C 和 objective – C 是一个重大的区别。在 objective – c 中, 类是唯一类型可以定义方法。在 swift, 你可以选择是否要定义一个类, 结构, 或枚举, 还有你定义方法类型的灵活性创造。

9.1 实例方法

实例方法是属于一个特定的类，结构或枚举实例的功能。他们支持这些实例的功能，无论是通过提供方法来访问和修改实例属性，或提供的功能与实例的目的。实例方法具有完全相同的语法功能，如功能描述你所属的类型的打开和关闭括号内写一个实例方法。一个实例方法具有隐式访问所有其他实例方法和

该类型的属性。一个实例方法只能在它所属的类的特定实例调用，它不能访问一个不存在的实例。这里，定义了一个简单的计数器类，它可以用来计数一个动作发生的次数的示例：

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

counter 类可以定义三个实例方法：- 增量递增计数器 1。- incrementBy(amount:Int) 由指定的整数金额递增计数器。- 重置将计数器的值重置为零。

计数类也声明了一个变量属性, 统计, 跟踪当前的计数器值。

你调用实例方法具有相同点语法的属性

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.incrementBy(5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

9.1.1 本地和外部参数名称的方法

函数参数可以有一个本地名称 (在函数体内使用) 和外部名称 (在调用函数时使用), 所述外部参数名称。方法参数也是如此, 因为方法与类型相关的函数。然而, 本地名称和外部名称的默认行为是不同的函数和方法。

方法在 Swift 非常类似于 objective – c 的同行。在 objective – c 中, 一个方法的名称在 Swift 通常是指使用 preposition 等方法的第一个参数„ 或者, 就像在 incrementBy 方法从前面的 counter 类的例子。使用可以被解读为一个判断的方法叫做 preposition。Swift 使这个方法建立命名约定易于编写通过使用一个不同的默认方法。

具体来说,Swift 给第一个参数名称方法默认本地参数名称, 并给出第二和后续的参数名称默认本地和外部参数名称。这个约定可以在熟悉的 objective – c 中调用到, 并使得表达方法调用而不需要符合你的参数名称。

考虑这个替代版本的 counter 类, 它定义了一个更复杂的形式的 incrementBy 方法:

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

这有两个 parameters-amount 和 numberOfTimes incrementBy 方法。默认情况下,Swift 将 amount 视为本地名称, 但将 numberOfTimes 视为本地和外部名称。您调用的方法如下:

```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// counter value is now 15
```

你不需要定义一个外部参数名称为第一个参数值, 因为它是明确的函数名 incrementBy。然而, 第二个参数是由外部参数名称进行限定。

这种默认行为有效的外部方法, 如果你有 numberOfTimes 参数之前写了一个 hash 符号（#）:

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
    count += amount * numberOfTimes
}
```

上面描述的默认行为在 Swift 写入相同的方法定义，语法类似于 objective – c，可以更方便地被调用。

9.1.2 修改外部参数名称的行为方法

有时是有用的提供一个外部方法的第一个参数的参数名称, 即使这不是默认行为。你自己可以添加一个显式的外部名称, 或者你可以用一个散列前缀的名字的第一个参数标志使用本地名称作为外部的名字。相反, 如果你不想为第二个提供外部名称或后续参数的方法, 覆盖默认行为通过使用下划线字符 (_) 作为一个明确的外部参数名称参数。

9.1.3 Self 属性

一个类型的每个实例都有所谓的一个隐含 self 属性，它是完全等同于该实例本身。您可以使用这个隐含的 self 属性来引用当前实例中它自己的实例方法。

在上面的例子中，增量方法也可以写成这样:

```
func increment() {
    self.count++
}
```

在实践中，你不需要写 self，这在你的代码会非常频繁。如果你没有明确写 self，Swift 假设你是指当前实例的属性或方法，每当你使用一个方法中一个已知的属性或方法名。这个假设是证明了里边三个实例方法的计数器使用 count (rather than self.count) 的。

主要的例外发生在一个实例方法的参数名称相同的名称作为该实例的属性。在这种情况下，参数名称的优先，有必要参考属性更多合格的方式。您可以使用隐式的自我属性的参数名和属性名来区分。

如果一个方法参数叫 x，还有一个实例属性也叫 x，在 Swift 中可以自动对两个 x 消除歧义，不会混淆。

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
    println("This point is to the right of the line where x == 1.0")
```

```
}
// prints "This point is to the right of the line where x == 1.0"
```

如果没有 self 前缀，Swift 将假定 x 的两种用法称为 X 的方法参数

9.1.4 修改值类型的实例方法

结构和枚举值类型。默认情况下, 一个值类型的属性不能修改它的实例方法

然而, 如果您需要修改的属性结构或枚举在一个特定的方法, 你可以选择该方法的变化行为。但任何更改都会使它得编写的方法结束时回到原来的结构。当该方法结束时还可以分配一个完全新的实例对其隐含的 self 属性, 而这个新的实例将取代现有的。

你可以选择这个行为之前将变异的关键字嵌入函数关键字的方法:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
println("The point is now at (\(somePoint.x), \(somePoint.y))")
// prints "The point is now at (3.0, 4.0)"
```

Point 结构上面定义了一个变异 moveByX 方法，它通过一定量移动一个 Point 实例。而不是返回一个新的起点，这种方法实际上会修改在其上调用点。该变异包含被添加到它的定义，使其能够修改其属性。请注意, 您不能调用变异方法结构类型的常数，因为它的属性不能改变，即使它们是可变的特性，如在固定结构实例存储的属性描述:

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// this will report an error
```

9.1.5 分配中的 self 变异方法

变异的方法可以分配一个全新的实例隐含的 self 属性。上面所示的点的例子也可以写成下面的方式来代替：

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

此版本的突变 moveByX 方法创建一个全新的结构，它的 x 和 y 值被设置到目标位置。调用该方法的结果和早期版本是完全一样的

变异的方法枚举可以设置 self 参数是从同一个枚举不同的成员

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
            case Off:
                self = Low
            case Low:
                self = High
            case High:
                self = Off
        }
    }
}
var ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight is now equal to .High
ovenLight.next()
// ovenLight is now equal to .Off
```

这个例子定义了一个三态开关枚举。三种不同的功率状态之间的切换周期 (关, 低, 高)

9.2 类型方法

如上所述, 实例方法的方法要求一个特定类型的实例。您还可以定义该类型自身的方法，这种方法被称为 type 方法，您显示的 type 方法直接在类结构体里面用 class func 开头，对于枚举和结构来说，类型方法是用 static func 开头。

> 请注意; 在 objective – c 中, 您可以定义 type-level 方法仅为 objective – c 类。在 Swift 可以为所有类定义 type-level 方法, 结构, 和枚举。每种方法的显示局限于它所支持的类型。

类型方法调用 dot syntax，就像实例方法。但是，您调用的是类型的方法，而不是该类型的一个实例。这里是你如何调用一个类调用 SomeClass 的一个类型的方法:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

在类型方法的主体，隐含的 self 属性是指类型本身，而不是该类型的一个实例。对于结构体和枚举，这意味着你可以使用自助静态属性和静态方法的参数消除歧义，就像你做的实例属性和实例方法的参数。

更普遍的是，你一个类型的方法体中使用任何不合格的方法和属性名称会参考其他 type-level 方法和属性。一种方法可以调用另一个类的方法与其他方法的名称，而不需要与类型名称前缀了。同样，结构和枚举类型的方法可以使用静态属性的名称，没有类型名称前缀访问静态属性。

下面的例子定义了一个名为 LevelTracker 结构，它通过游戏的不同层次或阶段跟踪球员的进步。这是一个单人游戏，但可以存储的信息为一个单一的设备上的多个玩家。

所有的游戏的水平 (除了一级) 当游戏第一次玩。每当玩家完成一个级别，该级别解锁设备上的所有玩家。LevelTracker 结构使用静态属性和方法来跟踪哪些级别的比赛已经解锁。它还跟踪当前个别球员水平

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
```

```
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel {
            highestUnlockedLevel = level
        }
    }
    static func levelIsUnlocked(level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
    var currentLevel = 1
    mutating func advanceToLevel(level: Int) -> Bool {
        if LevelTracker.levelIsUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

该 LevelTracker 结构跟踪任何玩家解锁的最高水平。这个值是存储在一个名为 highestUnlockedLevel 的静态属性。

LevelTracker 还定义了两种类型的功能与 highestUnlockedLevel，首先是一种叫做 unlockLevel 功能，每当一个新的水平解锁都会用来更新 highestUnlockedLevel，第二个是 levelIsUnlocked 功能，如果一个特定的水平数已经解锁，就会返回 ture。注意, 这些类型的方法可以访问 highestUnlockedLevel 静态属性但是你需要把它写成 LevelTracker.highestUnlockedLevel)。

除了它的静态属性和类型的方法，LevelTracker 通过游戏追踪每个玩家的进度。它使用被称为 currentLevel 实例属性来跟踪玩家级别。

为了帮助管理 urrentLevel 属性，advanceToLevel LevelTracker 定义一个实例方法。这种方法更新 currentLevel 之前，用来检查是否要求新的水平已经解除锁定。该 advanceToLevel 方法返回一个布尔值来指示它是否能够设置 currentLevel。

该 LevelTracker 结构使用 Player 类, 如下所示, 跟踪和更新单个球员的进步:

```
class Player {
    var tracker = LevelTracker()
```

```
    let playerName: String
    func completedLevel(level: Int) {
        LevelTracker.unlockLevel(level + 1)
        tracker.advanceToLevel(level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

Player 类创建 LevelTracker 的一个新实例来跟踪球员的进步。它也提供了一个名为 completedLevel 方法，每当玩家到达一个特定的级别，这种方法就会解锁一个新的级别和进度并把玩家移到下一个级别。(advanceToLevel 返回的布尔值将被忽略, 因为已知被调用 LevelTracker.unlockLevel。)

您可以创建一个新球员 Player 的实例, 看看当玩家完成一个级别会发生什么:

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// prints "highest unlocked level is now 2"
```

如果你创建第二个球员, 你想尝试移动到尚未被游戏解锁的级别，就会出现当前级别失败

```
player = Player(name: "Beto")
if player.tracker.advanceToLevel(6) {
    println("player is now on level 6")
} else {
    println("level 6 has not yet been unlocked")
}
// prints "level 6 has not yet been unlocked"
```

10 Swift 中文教程（十二）下标

类，结构和枚举类型都可以通过定义下标来访问一组或者一个序列中的成员元素。通过下标索引就可以方便地检索和设置相应的值，而不需要其他

的额外操作。比如你可以通过 someArray[index] 来访问数组中的元素，或者 someDictionary[key] 来对字典进行索引。

你可以为一个类型定义多个下标，以及适当的下标重载用来根据传递给下标的索引来设置相应的值。下标不仅可以定义为一维的，还可以根据需要定义为多维的，多个参数的。

10.1 下标语法

下标可以让你通过实例名后加中括号内一个或多个数值的形式检索一个元素。语法和方法语法和属性语法类似，通过使用 subscript 关键定义，一个或多个输入参数以及一个返回值。不同于实例方法的是，下标可以是可读写的或者只读的。这种行为通过一个 getter 和 setter 语句联通，就像是计算属性一样。

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

newValue 的类型和下标返回的类型一样。和计算属性一样，你可以选择不指定 setter 的参数，因为当你不指定的时候，默认参数 newValue 会被提供给 setter。

和计算属性一样，只读下标可以不需要 get 关键词：

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

下面是一个只读下标的实现，定义了一个 TimesTable 结构来表示一个整数的倍数表：

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
```

```
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
println("six times three is \(threeTimesTable[6])")
// prints "six times three is 18"
```

在这个例子中，实例 TimesTable 被创建为 3 倍数表，这是通过在初始化的时候为 multiplier 参数传入的数值 3 设置的。

注意:

倍数表是根据特定的数学规则设置的,所以不应该为 threeTimeTable[someIndex] 元素设置一个新值，所以 TimesTable 的下标定义为只读。

10.2 下标的使用

下标的具体含义由使用它时的上下文来确定。下标主要用来作为集合，列表和序列的元素快捷方式。你可以自由的为你的类或者结构定义你所需要的下标。

比如说，Swift 中字典类型实现的下标是设置和检索字典实例中的值。可以通过分别给出下标中的关键词和值来设置多个值，也可以通过下标来设置单个字典的值:

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

上面的例子中定义了一个变量 numberOfLegs，然后通过键值对初始化。numberOfLegs 的类型是字典类型 Dictionary。在字典创建之后，例子使用了下标赋值方法添加了一个类型为字符串的键”bird” 和 Int 值 2 到字典中。

更多关于字典的下标可以参考：访问和修改字典这一章节

注意:

Swift 中字典类型实现的键值对下标是可选类型。对于 numberOfLges 字典来说，返回的值是 Int?，也就是可选 Int 值。字典的这种使用可选类型下标的方式说明不是所有的键都有对应的值。同样也可以通过给键赋值 nil 来删除这个键。

10.3 下标选项

下标可以接收任意数量的参数，参数的类型也可以各异。下标还可以返回任何类型的值。下标可以使用变量参数或者可变参数，但是不能够使用输入输出参数或者提供默认参数的值。

类或者结构可以根据需要实现各种下标方式，可以在需要的时候使用合适的下标通过中括号中的参数返回需要的值。这种多下标的定义被称作下标重载。

当然，最常见的下标用法是单个参数，也可以定义多个参数的下标。下面的例子演示了一个矩阵 Matrix 结构，它含有二维的 Double 值。矩阵结构的下标包括两个整形参数:

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
      return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
        assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
        assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

矩阵 Matrix 提供了一个初始化方法，使用两个参数 rows 和 columns，然后建立了一个数组来存储类型为 Double 的值 rows*columns。每个矩阵中的位置都被设置了一个初始值 0.0。通过传递初始值 0.0 和数组长度给数组初始化方法完成上述操作。数组的初始化方法在：创建和初始化数组中有更详细的叙述。

你可以传递两个参数 row 和 column 来完成 Matrix 的初始化：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上面的初始化操作创建了一个两行两列的矩阵 Matrix 实例。这个矩阵实例的 grid 数组看起来是平坦的，但是实际上是矩阵从左上到右下的一维存储形式。

```
![](pic/subscriptMatrix01_2x.png "subscriptMatrix01_2x")
```

矩阵中的值可以通过使用包含 row 和 column 以及逗号的下标来设置：

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

这两个语句调用了下标的 setter 方法为右上和左下角的两个元素分别赋值 1.5 和 3.2

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

矩阵下标的 getter 和 setter 方法都包括了一个断言语句来检查下标 row 和 column 是否有效。通过 indexIsValid 方法来判断 row 和 column 是否在矩阵的范围内：

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

如果访问的矩阵越界的时候，断言就会被触发：

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix bounds
```

11 Swift 中文教程（十三）继承

一个类可以从另外一个类中继承方法，属性或者其它的一些特性。当一个类继承于另外一个类时，这个继承的类叫子类，被继承的类叫父类。继承是 Swift 中类区别于其它类型的一个基本特征。

Swift 中的类可以调用父类的方法，使用父类的属性和下标，还可以根据需要使用重写方法或者属性来重新定义和修改他们的一些特性。Swift 可以帮助你检查重写的方法和父类的方法定义是相符的。

类还可以为它继承的属性添加观察者，这样可以能够让它在一个属性变化的时候得到通知。属性观察者可以被添加给任何属性，不管它之前是存储属性还是计算属性。

11.1 定义一个基类

任何一个不继承于其它类的类被称作基类

> 注意：Swift 的类不是从一个全局基类继承而来。在你编写代码的时，只要是在类的定义中没有继承自父类的类都是基类。

下面的例子定义了一个叫 Vehicle 的基类。基类包含两个所有交通工具通用的属性 numberOfWheels 和 maxPassengers。这两个属性被一个叫 description 的方法使用，通过返回一个 String 描述来作为这个交通工具的特征：

```
class Vehicle {
    var numberOfWheels: Int
    var maxPassengers: Int
    func description() -> String {
     return "\(numberOfWheels) wheels; up to \(maxPassengers) passengers"
    }
    init() {
        numberOfWheels = 0
        maxPassengers = 1
    }
}
```

这个交通工具类 Vehicle 还定义了一个构造函数来设置它的属性。构造函数更多的解释在 Initialization 一章，但是为了说明子类如何修改继承的属性，这里需要简要解释一下什么叫构造函数。

通过构造函数可以创建一个类型的实例。尽管构造函数不是方法，但是它们在编码的时候使用了非常相似的语法。构造函数通过确保所有实例的属性都是有效的来创建一个新的实例。

构造函数最简单的形式是使用 init 关键词的一个类似方法的函数，并且没有任何参数：

```
init() {
    // perform some initialization here
}
```

使用构造函数语法 TypeName 和空的两个小括号来完成一个 Vehicle 实例的创建：

```
let someVehicle = Vehicle()
```

Vehicle 的构造函数为属性设置了一些初始值 (numberOfWheels = 0 然后 maxPassengers = 1)。

Vehicle 类定义的是一个通用的交通工具特性，它本身没有太多意义，所以就需要冲定义它的一些属性或者方法来让它具有实际的意义。

11.2 产生子类

产生子类就是根据一个已有的类产生新类的过程。子类继承了父类的一些可以修改的特性。还可以为子类添加一些新的特性。

为了表明一个类是继承自一个父类，需要将父类的名称写在子类的后面，并且用冒号分隔：

```
class SomeClass: SomeSuperclass {
    // class definition goes here
}
```

下面的例子定义了一种特定叫 Bicycle 的交通工具。这个新类是基于已有的类 Vehicle 产生的。书写方式是在类名 Bicycle 后加冒号加父类 Vehicle 名。

可以理解为：

定义一个新的类叫 Bicycle，它继承了 Vehicle 的特性：

```
class Bicycle: Vehicle {
    init() {
        super.init()
        numberOfWheels = 2
    }
}
```

Bicycle 是 Vehicle 的子类，Vehicle 是 Bicycle 的父类。Bicycle 类继承了 Vehicle 所有的特征，比如 maxPassengers 和 numberOfWheels 属性。你还可以为 Bicycle 类添加心的属性。

Bicycle 类也定义了构造函数，在这个构造函数中调用了父类的构造函数 super.init()，这样可以确保在 Bicycle 修改他们之前，父类已经初始化了。

> 注意：跟 Objective-C 不同的是，Swift 中的构造函数没有默认继承。
> 更多信息可以参考 Initializer Inheritance and Overriding 这一章节。

maxPassengers 属性在继承自父类的时候已经被初始化了，对于 Bicycle 来说是正确的，因此不需要再做更改。然后 numberOfWheels 是不对的，所以被替换成了 2.

不仅属性是继承于 Vehicle 的，Bicycle 还继承了父类的方法。如果你创建一个实例，然后调用了已经继承的 description 方法，可以得到该交通工具的描述并且看到它的属性已经被修改：

```
let bicycle = Bicycle()
println("Bicycle: \(bicycle.description())")
// Bicycle: 2 wheels; up to 1 passengers
```

子类本身也可以作为父类被再次继承：

```
class Tandem: Bicycle {
    init() {
        super.init()
        maxPassengers = 2
    }
}
```

上面的例子创建了 Bicycle 的子类，叫做 tandem，也就可以两个人一起骑的自行车。所以 Tandem 没有修改 numberOfWheels 属性，只是更新了 maxPassengers 属性。

注意：子类只能够在构造的时候修改变量的属性，不能修改常量的属性。

创建一个 Tandem 的实例，然后调用 description 方法检查属性是否被正确修改：

```
let tandem = Tandem()
println("Tandem: \(tandem.description())")
// Tandem: 2 wheels; up to 2 passengers
```

注意到 description 方法也被 Tandem 继承了。

11.3 重写方法

子类可以提供由父类继承来的实例方法，类方法，实例属性或者下标的个性化实现。这个特性被称为重写。

重写一个由继承而来的方法需要在方法定义前标注 override 关键词。通过这样的操作可以确保你所要修改的这个方法确实是继承而来的，而不会出现重写错误。错误的重写会造成一些不可预知的错误，所以如果如果不标记 override 关键词的话，就会被在代码编译时报错。

override 关键词还能够让 Swift 编译器检查该类的父类是否有相符的方法，以确保你的重写是可用的，正确的。

11.3.1 访问父类方法，属性和下标

当在重写子类继承自父类的方法，属性或者下标的时候，需要用到一部分父类已有的实现。比如你可以重定义已知的一个实现或者在继承的变量中存储一个修改的值。

适当的时候，可以通过使用 super 前缀来访问父类的方法，属性或者下标：

- 叫 someMethod 的重写方法可以在实现的时候通过 super.someMethod() 调用父类的 someMethod 方法。
- 叫 someProperty 的重写属性可以在重写实现 getter 或者 setter 的时候通过 super.someProperty 调用父类的 someProperty。
- 叫 someIndex 的重写下标可以在实现下标的时候通过 super[someIndex] 来访问父类的下标。

11.3.2 复写方法

你可以在你的子类中实现定制的继承于父类的实例方法或者类方法。

下面的例子演示的就是一个叫 Car 的 Vehicle 子类，重写了继承自 Vehicle 的 description 方法。

```
class Car: Vehicle {
    var speed: Double = 0.0
    init() {
        super.init()
        maxPassengers = 5
        numberOfWheels = 4
    }
    override func description() -> String {
        return super.description() + "; "
            + "traveling at \(speed) mph"
    }
}
```

Car 中定义了一个新的 Double 类型的存储属性 speed。这个属性默认值是 0.0，意思是每小时 0 英里。Car 还有一个自定义的构造函数，设置了最大乘客数为 5，轮子数量是 4.

Car 重写了继承的 description 方法，并在方法名 description 前标注了 override 关键词。

在 description 中并没有给出了一个全新的描述实现,还是通过 super.description 使用了 Vehicle 提供的部分描述语句，然后加上了自己定义的一些属性，如当前速度。

如果你创建一个 Car 的实例，然后调用 description 方法，会发现描述语句变成了这样:

```
let car = Car()
println("Car: \(car.description())")
// Car: 4 wheels; up to 5 passengers; traveling at 0.0 mph
```

11.3.3 复写属性

你还可以提供继承自父类的实例属性或者类属性的个性化 getter 和 setter 方法，或者是添加属性观察者来实现重写的属性可以观察到继承属性的变动。

11.3.4 重写属性的 Getters 和 Setters

不管在源类中继承的这个属性是存储属性还是计算属性，你都可以提供一个定制的 getter 或者 setter 方法来重写这个继承属性。子类一般不会知道这个继承的属性本来是存储属性还是计算属性，但是它知道这个属性有特定的名字和类型。在重写的时候需要指明属性的类型和名字，好让编译器可以检查你的重写是否与父类的属性相符。

你可以将一个只读的属性通过提那家 getter 和 setter 继承为可读写的，但是反之不可。

> 注意：如果你为一个重写属性提供了 setter 方法，那么也需要提供 getter 方法。如果你不想在 getter 中修改继承的属性的值，可以在 getter 中使用 super.someProperty 即可，在下面 SpeedLimitedCar 例子中也是这样。

下面的例子定义了一个新类 SpeedLimitedCar，是 Car 的一个子类。这个类表示一个显示在 40 码一下的车辆。通过重写继承的 speed 属性来实现：

```
class SpeedLimitedCar: Car {
    override var speed: Double  {
    get {
        return super.speed
    }
    set {
        super.speed = min(newValue, 40.0)
    }
    }
}
```

每当你要设置 speed 属性的时候，setter 都会检查新值是否比 40 大，二者中较小的值会被设置给 SpeedLimitedCar。

如果你尝试为 speed 设置超过 40 的值，description 的输出依然还是 40：

```
let limitedCar = SpeedLimitedCar()
limitedCar.speed = 60.0
println("SpeedLimitedCar: \(limitedCar.description())")
// SpeedLimitedCar: 4 wheels; up to 5 passengers; traveling at 40.0 mph
```

11.3.5 重写属性观察者

你可以使用属性重写为继承的属性添加观察者。这种做法可以让你无论这个属性之前是如何实现的，在继承的这个属性变化的时候都能得到提醒。更多相关的信息可以参考 Property Observers 这章。

> 注意：不能为继承的常量存储属性或者是只读计算属性添加观察者。这些属性值是不能被修改的，因此不适合在重写实现时添加 willSet 或者 didSet 方法。

> 注意：不能同时定义重写 setter 和重写属性观察者，如果想要观察属性值的变化，并且又为该属性给出了定制的 setter，那只需要在 setter 中直接获得属性值的变化就行了。

下面的代码演示的是一个新类 AutomaticCar，也是 Car 的一个子类。这个类表明一个拥有自动变速箱的汽车，可以根据现在的速度自动选择档位，并在 description 中输出当前档位：

```
class AutomaticCar: Car {
    var gear = 1
    override var speed: Double {
    didSet {
        gear = Int(speed / 10.0) + 1
    }
    }
    override func description() -> String {
        return super.description() + " in gear \(gear)"
    }
}
```

这样就可以实现，每次你设置 speed 的值的时候，didSet 方法都会被调用，来看档位是否需要变化。gear 是由 speed 除以 10 加 1 计算得来，所以当速度为 35 的时候，gear 档位为 4：

```
let automatic = AutomaticCar()
automatic.speed = 35.0
println("AutomaticCar: \(automatic.description())")
// AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph in gear 4
```

11.4 禁止重写

你可以通过标记 final 关键词来禁止重写一个类的方法，属性或者下标。在定义的关键词前面标注 @final 属性即可。

在子类中任何尝试重写父类的 final 方法，属性或者下标的行为都会在编译时报错。同样在扩展中为类添加的方法，属性或者下标也可以被标记为 final。

还可以在类关键词 class 前使用 @final 标记一整个类为 final(@final class)。任何子类尝试继承这个父类时都会在编译时报错。# Swift 中文教程（十四）初始化

初始化是类，结构体和枚举类型实例化的准备阶段。这个阶段设置这个实例存储的属性的初始化数值和做一些使用实例之前的准备以及必须要做的其他一些设置工作。

通过定义构造器（initializers）实现这个实例化过程，也就是创建一个新的具体实例的特殊方法。和 Objective-C 不一样的是，Swift 的构造器没有返回值。它们主要充当的角色是确保这个实例在使用之前能正确的初始化。

类实例也能实现一个析构器（deinitializer），在类实例销毁之前做一些清理工作。更多的关于析构器（deinitializer）的内容可以参考 Deinitialization。

11.5 存储属性的初始化

类和结构体必须在它们被创建时把它们所有的属性设置为合理的值。存储属性不能为不确定状态

你可以在构造方法里面给一个属性设置一个初始值，或者在定义的时候给属性设置一个默认值，这个行为将会在接下来的章节描述。

> 注意：当你对给一个属性分配一个默认值的时候，它会调用它相对应的初始化方法，这个值是对属性直接设置的，不会通知它对应的观察者

11.5.1 构造器

构造器是创建一个具体类型实例的方法。最简单的构造器就是一个没有任何参数实例方法，写作 init。

在下面的例子定义了一个叫 Fahrenheit（华氏度）的新结构体，来储存转换成华氏度的温度。Fahrenheit 结构体，有一个属性，叫 temperature（温度），它的类型为 Double（双精度浮点数）：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
println("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个单一的构造方法 init，它没有任何参数，它储存的温度属性初始化为 32.0 度。(水在华氏度的温度情况下的冰点)。

11.5.2 属性的默认值

如上所述，你可以在构造器中设置它自己的储存属性的初始化值。或者在属性声明时，指定属性的默认值，你指定一个默认的属性值，会被分配到它定义的初始值。

> 注意: 如果一个属性常常使用同样的初始化值，提供一个默认值会比在初始化使用一个默认值会更好。

同样的结果，但是默认值与属性的初始化在它定义地时候就紧紧地捆绑在一起。很简单地就能构造器更简洁，和可以让你从默认值中推断出这个属性的类型。默认值也能让你优化默认构造器和继承构造器变得更容易，在本章会稍候描述。

你可以在上面的 Fahrenheit（华氏度）结构体定义时，给 temperature（温度）属性提供默认值。

```
struct Fahrenheit {
    var temperature = 32.0
}
```

11.6 自定义初始化（Customizing Initialization）

你可以根据输入的参数来自定义初始化过程和可选的属性类型，或者在初始化的时候修改静态属性。在这章节将会详细叙述。

11.6.1 初始化参数

你可以在构造器定义的时候提供一部分参数，在自定义初始化过程中定义变量的类型和名称。初始化参和函数或者方法参数一样有着同样的功能。

在下面的例子中，定义了一个结构体 Celsius。储存了转换成摄氏度的温度，Celsius 结构体实现了从不同的温度初始化结构体的两个方法，init(fromFahrenheit:) 和 init(fromKelvin:)。

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

第一个构造器只有一个初始化参数，形参（External Parameter Names）fromFahrenheit，和实参（Local Parameter Names）fahrenheit。第二个构造器有一个单一的初始化参数，形参（External Parameter Names）fromKenvin，和实参（Local Parameter Names）kelvin。两个构造器都把单一的参数转换为摄氏度和储存到一个 temperatureInCelsius 的属性.

11.6.2 实参名（**Local Parameter Names**）和形参名（**External Parameter Names**）

和函数参数和方法参数一样，初始化参数拥有在构造器函数体使用的实参，和在调用时使用的形参.

然而，和函数或者方法不同，构造器在圆括号前面没有一个识别函数名称。因此，构造器参数的名称和类型，在被调用的时候，很大程度上扮演一个被识别的重要角色。为此，在构造器中，当你没有提供形参名时，Swift 就会为每一个参数提供一个自动的形参名。这个形参名和实参名相同，就像和之前你写的每一个初始化参数的 hash 符号一样。

> 注意：如果你在构造器中没有定义形参，提供一个下横线（_）作为区分形参和上面说描述的重写默认行为。

在下面的例子，定义了一个结构体 Color，拥有三个静态属性 red，green 和 blue。这些属性储存了从 0.0 到 1.0 的值，这些值代表红色，绿色和蓝色的深度。

Color 提供了一个构造器，以及三个双精度（Double）类型的参数：

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
}
```

无论什么时候，你创建一个 Color 实例，你必须使用每一个颜色的形参来调用构造器：

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

值得注意的是，不能不通过形参名来调用构造器。在构造器定义之后，形参名必须一致使用。如果漏掉就会在编写时提示错误。

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - external names are required
```

11.6.3 可选类型

如果你储存属性使用的是自定义的类型在逻辑上允许值为空 -或者他们的值并不在构造器中初始化，或者他们被允许为空。可以定义一个可选类型的属性。可选类型属性是一个自动初始化值为 nil，表示这个属性有意在构造器中设置为“空值”（no value yet）。

在下面的例子中，定义了一个 SurveryQuestion 类，拥有一个可选的 String 属性 response。

这个回答在他们调查问题在发布之前是无法知道的，所以 response 定义为类型 String? ，或者叫可选 String（optional String）。说明它会被自动分配一个默认值 nil，意思为当 surverQuestion 初始化时还不存在。

11.6.4 在初始化时修改静态属性

当你在设置静态属性值时，只要在初始化完成之前，你都可以在初始化时随时修改静态属性。

> 注意：对于类的实例化，一个静态属性只能在初始化时被修改，这个初始化在类定义时已经确定。

你可以重写 SurveryQuestion 例子，对于问题的 text 属性，使用静态属性会比动态属性要好，因为 SurveyQuestion 实例被创建之后就无法修改。尽管 text 属性现在是静态的，但是仍然可以在构造器中被设置：

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

11.7 默认构造器

Swift 为每一个结构或者基类提供了默认的构造器，来初始化它们所包含的所有属性。默认构造器将会创建一个新的实例然后将它们的属性设置为默认值。

下面的例子定义了一个叫 ShoppingListItem 的类，包含了名称，数量和是否已购买的属性，将会被用在购物清单中：

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
```

```
    var purchased = false
}
var item = ShoppingListItem()
```

因为 ShoppingListItem 类中所有的属性都有默认值，并且这个类是一个没有父类的基类，所以它默认拥有一个会将所有包含的属性设置为初始值的默认构造器。比如在这个例子中 name 属性是一个可选 String 属性，它会被默认设置为 nil，尽管在代码中没有指明。上面的例子使用默认构造器创建了一个 ShoppingListItem 类，记做 ShoppingListItem()，然后将它赋值给了变量 item。

11.7.1 结构类型的成员逐一构造器

除了上面提到的默认构造器之外，结构类型还有另外一种成员逐一完成初始化的构造器，可以在定义结构的时候直接指定每个属性的初始值。

成员逐一构造器是一种为结构的成员属性进行初始化的简便方法。下面的例子定义了一个叫 Size 的结构，和两个属性分别叫 width 和 height。每个属性都是 Double 类型的并且被初始化为 0.0。

因为每个存储属性都有默认值，在 Size 结构创建一个实例的时候就可以自动调用这个成员逐一构造器 init(width:height:):

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

11.8 数值类型的构造器代理

在实例的初始化过程中，构造器可以调用其他的构造器来完成初始化。这个过程叫构造器代理，可以避免多个构造器的重复代码。

对于数值类型和类来说，构造器代理的工作形式是不一样的。数值类型（结构和枚举）不支持继承，因此他们的构造器代理相对简单，因为它们只能使用自己的构造器代理。但是一个类可以继承自另外一个类，所以类需要确保在初始化的时候将它所有的存储属性都设置为正确的值。这种过程在下一节类的继承和初始化中叙述。

对于数值类型来说，可以使用 self.init 来调用其他构造器，注意只能在这个数值类型内部调用相应的构造器。

需要注意的是如果你为数值类型定义了一个构造器，你就不能再使用默认构造器了。这种特性可以避免当你提供了一个特别复杂的构造器的时候，另外一个人误使用了默认构造器而出错。

> 注意：如果你想要同时使用默认构造器和你自己设置的构造器，不要将这两种构造器写在一起，而是使用扩展形式。更多内容可以参考 Extensions 一章。

下面的示例定义了一个结构 Rect 来表示一个几何中的矩形。这个 Rect 结构需要另外两个结构来组成，包括 Size 和 Point，初始值均为 0.0：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

现在你有三种初始化 Rect 结构的方式：直接使用为 origin 和 size 属性初始化的 0 值，给定一个指定的 origin 和 size，或者使用中心点和大小来初始化。下面的例子包含了这三种初始化方式：

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

init() 构造器和默认构造器的功能相同。这个构造器不需要任何内容，只是用来在已有其他构造器的时候表示默认构造器的依然存在。调用这个构造器创建的 Rect，根据 Point 和 Size 的结构定义，Point(x: 0.0, y: 0.0) ，Size(width: 0.0, height: 0.0) origin 和 size 都会被设置为 0。

```
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

第二个 Rect 构造器 init(origin:size:) 和成员逐一构造器类似，它使用给定的值来初始化结构的属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

第三个构造器 init(center:size) 就更加复杂一些，它首先使用 center 和 size 计算出了 origin 的值，然后调用（或者是使用代理）了 init(origin:size) 构造器，设置 origin 和 size 的值：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

init(center:size:) 构造器同样可以设置 oringin 和 size 的值，而且使用起来也非常方便，代码也比较简洁因为它使用了已有的一些构造器。

注意：可以参考 Extensions 一章，学习怎样省略 init() 和 init(origin:size:)

11.9 类的继承和初始化

译者注：本小节内容 Apple 从底层解释，十分复杂，建议有需要的读者自行阅读英文原文。

本小节主要的意思就是说：

1. 自定义初始化方法要先调用自己类默认初始化方法，自己重写默认初始化方法要先调用父类默认初始化方法
2. 应该要先调用父类的构造器或者自身的默认构造器，以防止先给属性赋值了然后才调用父类或者自身的默认构造器把以前的赋值覆盖了

一个类的所有存储属性 -包括从父类继承而来的属性 -都必须在初始化的时候设置初始值。

Swift 为 class 类型定义了两种构造器来确保它们所有的存储属性都设置了初始值。这两种方式叫做指定构造器和便捷构造器。

11.9.1 指定构造器和便捷构造器

指定构造器是一个类最主要的构造器。指定构造器通过设置所有属性的初值并且调用所有的父类构造器来根据构造链一次初始化所有的属性。

类所拥有的指定构造器很少，一般只有一个，并且是连接这父类的构造链依次完成构造的。

每个类至少有一个指定构造器，在有些情况下，需要使用继承来从父类中得到该指定构造器，更多内容可以查看后面的 Automatic Initializer Inheritance 章节。

便捷构造器是类的第二种常用构造器。你可以调用同一个类中的指定构造器来定义一个便捷构造器，使用指定构造器来设置相关的参数默认值。你还可以定义一个便捷构造器来创建这个类的实例或者是别的特殊用途。

如果你的类不需要它们，也可以不定义便捷构造器。不过对于常见初始化模型需要快捷方式的时候创建一个便捷构造器可以让你的初始化过程变成十分简单便捷。

11.9.2 构造链

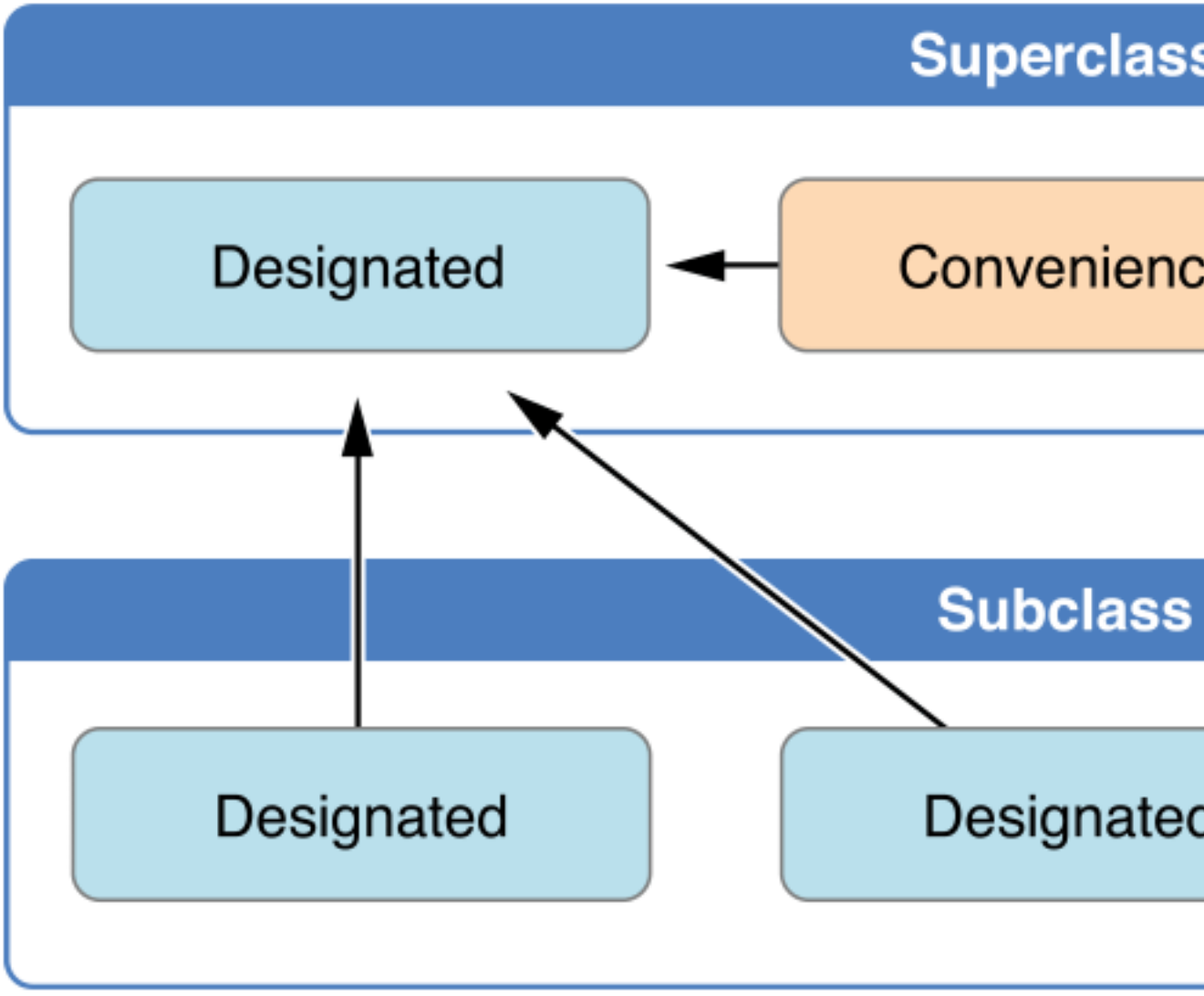
为了简化指定构造器和便捷构造器的关系，Swift 为两种构造器的代理调用设置了三个规则：

- 规则 1 指定构造器必须调用它直接父类的指定构造器
- 规则 2 便捷构造器只能调用同一个类中的其它构造器
- 规则 3 便捷构造器必须以调用一个指定构造器结束

记下这些规则的简单方法是：

- 指定构造器必须向上代理
- 便捷构造器必须横向代理
- 可以使用下面的图来表示：

父类中的两个便捷构造器依次调用直到指定构造器，子类中的指定构造器调用了父类的指定构造器。


Superclass
Designated
Convenienc
Subclass
Designated
Designated

> 注意：这些规则不会影响每个类的实例创建过程。每个构造器都可以用来创建它们各自的类的实例。这些规则只影响你如何编写类实现代码。

下图演示的是另一种更为复杂的具有四个等级的类。这个图展示了指定构造器在类的初始化过程中如何被作为 “漏斗” 节点的。这个构造链简化了类与类之间的交互关系：

11.9.3 两阶段的初始化

在 Swift 中，类的初始化要经过两个阶段。在第一个阶段，每一个存储属性都被设置了一个初始值。一旦每个存储属性的值在初始化阶段被设置了，在第二个阶段，每个类在这个实例被使用之前都会有机会来设置它们相应的存储属性。

两阶段的模式使初始化过程更加安全，还可以让每个类在类的层级关系中具有更多的可能性。两阶段初始化方法可以防止属性在被初始化之前就被使用，或者是被另一个构造器错误地赋值。

> 注意：Swift 的这种两阶段初始化方法跟 Objective-C 中的类似。主要的差别是在第一个过程中，Objective-C 为每个属性赋值 0 或者 null，而在 Swift 中，可以个性化设置这些初始值，还可以处理一些初始值不能是 0 或者 nil 的情况。

Swift 编译器通过四重检查来确保两阶段式的初始化过程是完全正确无误的：

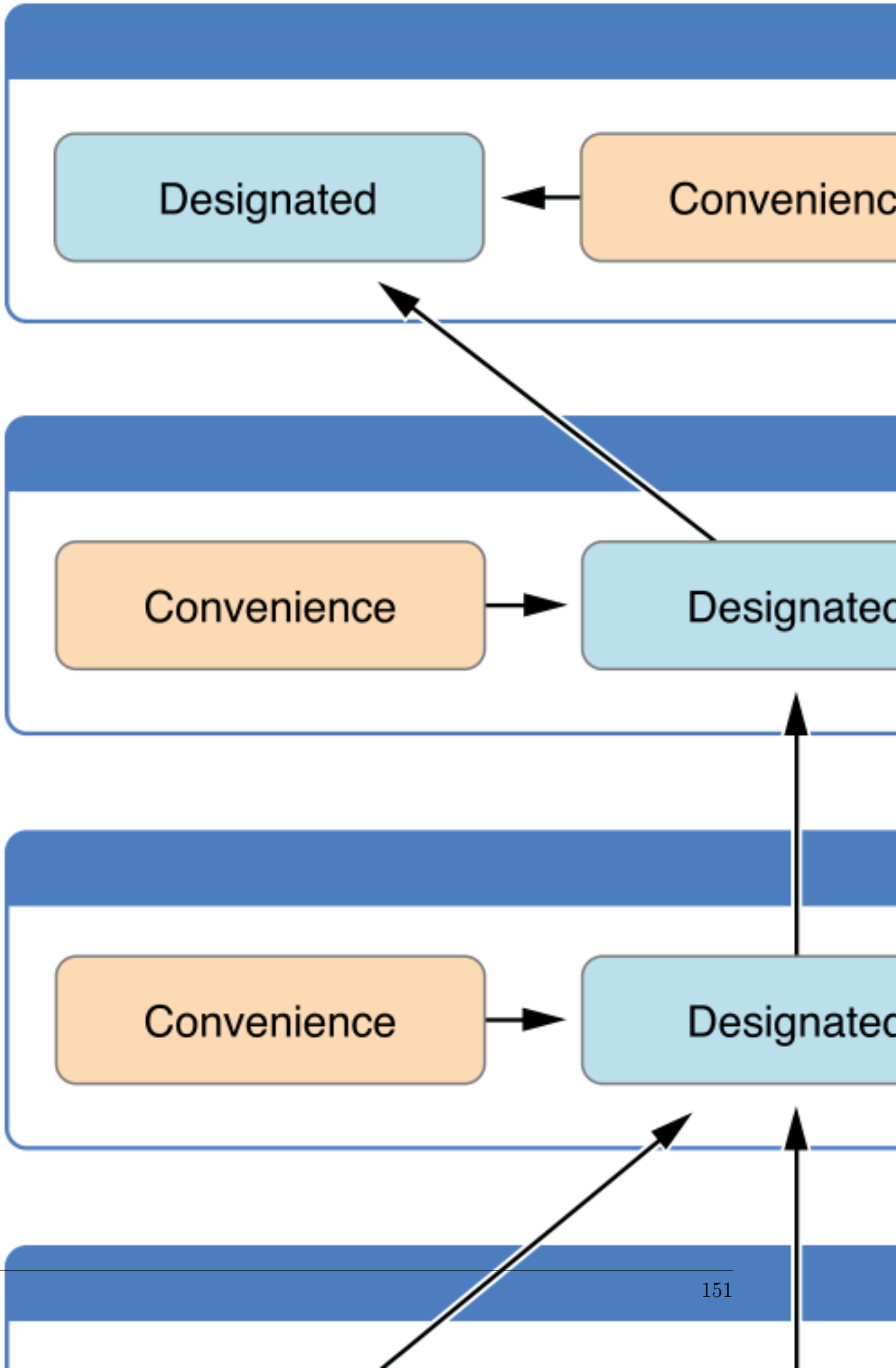
- Safety check 1

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer. As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all its own properties are initialized before it hands off up the chain.

- Safety check 2

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

- Safety check 3


Designated
Convenienc
Convenience
Designated
Convenience
Designated

A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

- Safety check 4

An initializer cannot call any instance methods, read the values of any instance properties, or refer to selfas a value until after the first phase of initialization is complete. The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out, based on the four safety checks above:

- Phase 1

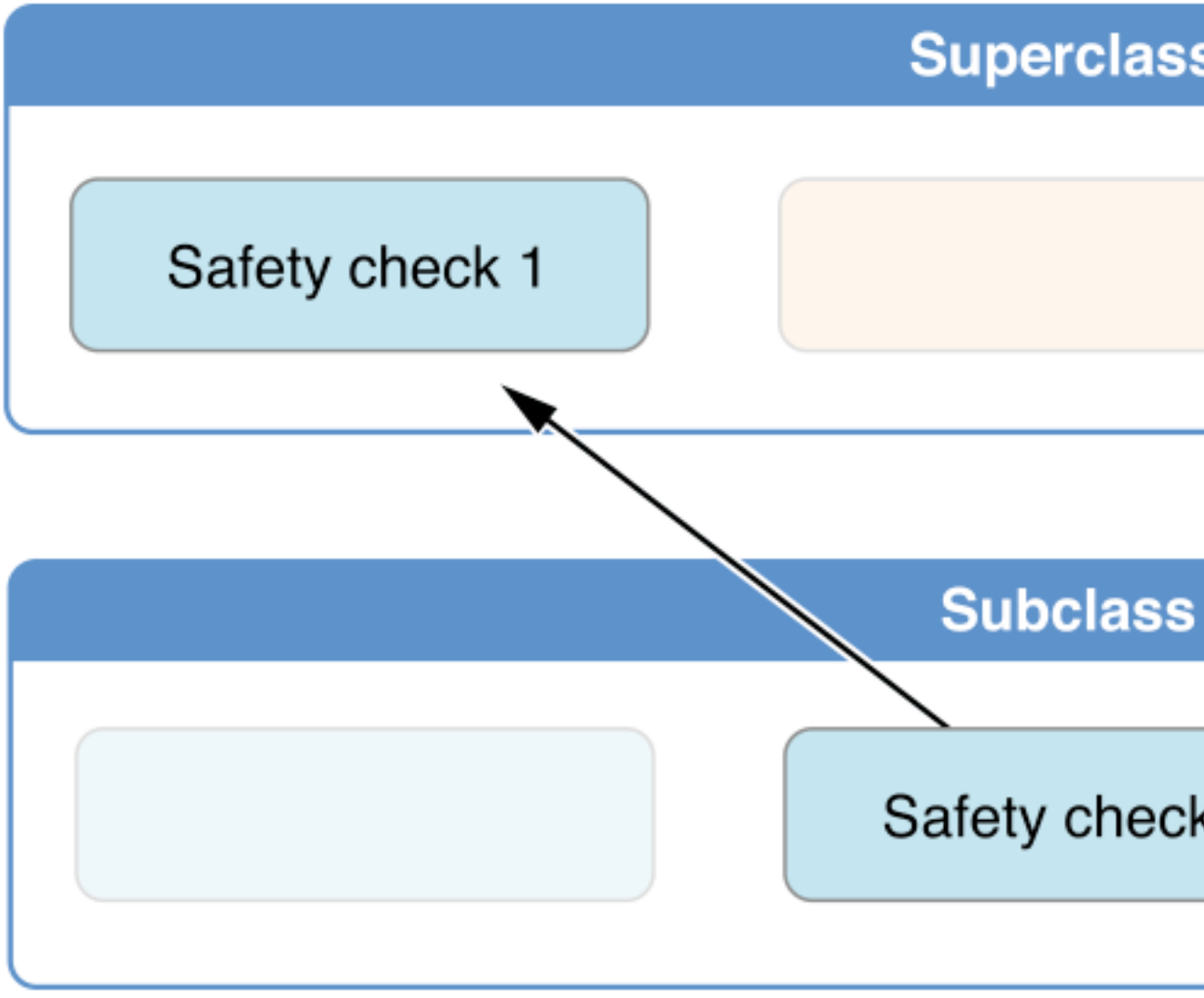
A designated or convenience initializer is called on a class. Memory for a new instance of that class is allocated. The memory is not yet initialized. A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized. The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties. This continues up the class inheritance chain until the top of the chain is reached. Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

- Phase 2

Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access self and can modify its properties, call its instance methods, and so on. Finally, any convenience initializers in the chain have the option to customize the instance and to work with self. Here's how phase 1 looks for an initialization call for a hypothetical subclass and superclass:

In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

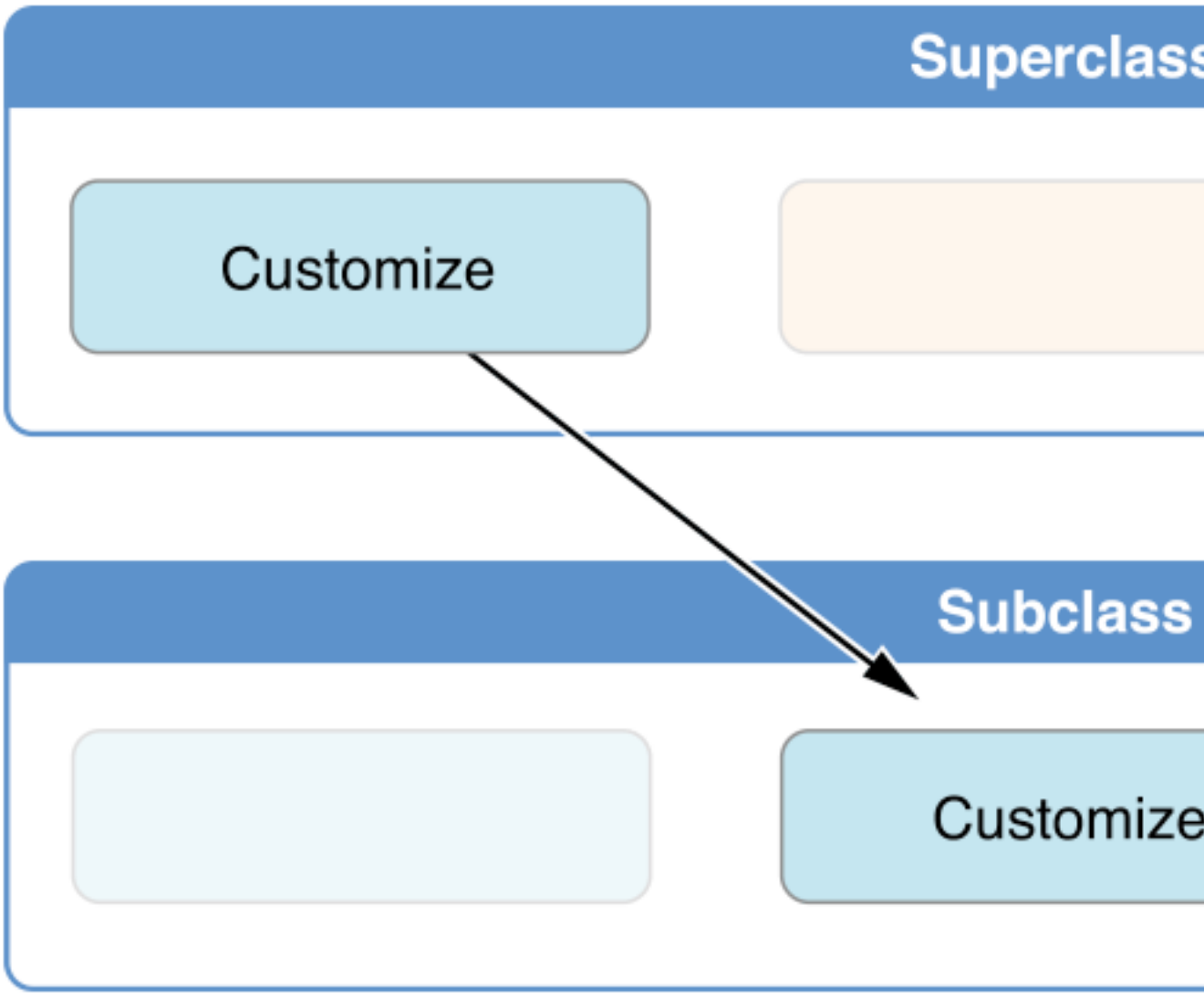
The designated initializer makes sure that all of the subclass's properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.


Superclass
Safety check 1
Subclass
Safety check

The superclass's designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and Phase 1 is complete.

Here's how phase 2 looks for the same initialization call:



The superclass's designated initializer now has an opportunity to customize the instance further (although it does not have to).

Once the superclass's designated initializer is finished, the subclass's designated initializer can perform additional customization (although again, it does not have to).

Finally, once the subclass's designated initializer is finished, the convenience

initializer that was originally called can perform additional customization.

11.9.4 构造器的继承和重写

Unlike subclasses in Objective-C, Swift subclasses do not not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is automatically inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

If you want your custom subclass to present one or more of the same initializers as its superclass—perhaps to perform some customization during initialization—you can provide an overriding implementation of the same initializer within your custom subclass.

If the initializer you are overriding is a designated initializer, you can override its implementation in your subclass and call the superclass version of the initializer from within your overriding version.

If the initializer you are overriding is a convenience initializer, your override must call another designated initializer from its own subclass, as per the rules described above in Initializer Chaining.

> NOTE Unlike methods, properties, and subscripts, you do not need to write the override keyword when overriding an initializer.

11.9.5 构造器自动继承

As mentioned above, subclasses do not not inherit their superclass initializers by default. However, superclass initializers are automatically inherited if certain conditions are met. In practice, this means that you do not need to write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it is safe to do so.

Assuming that you provide default values for any new properties you introduce in a subclass, the following two rules apply:

- Rule 1 If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

- Rule 2 If your subclass provides an implementation of all of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

NOTE

A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

11.9.6 指定初始化和便捷初始化的语法

Designated initializers for classes are written in the same way as simple initializers for value types:

```
init(parameters) {
    statements
}
```

Convenience initializers are written in the same style, but with the convenience keyword placed before the initkeyword, separated by a space:

```
convenience init(parameters) {
    statements
}
```

11.9.7 指定初始化和便捷初始化实战

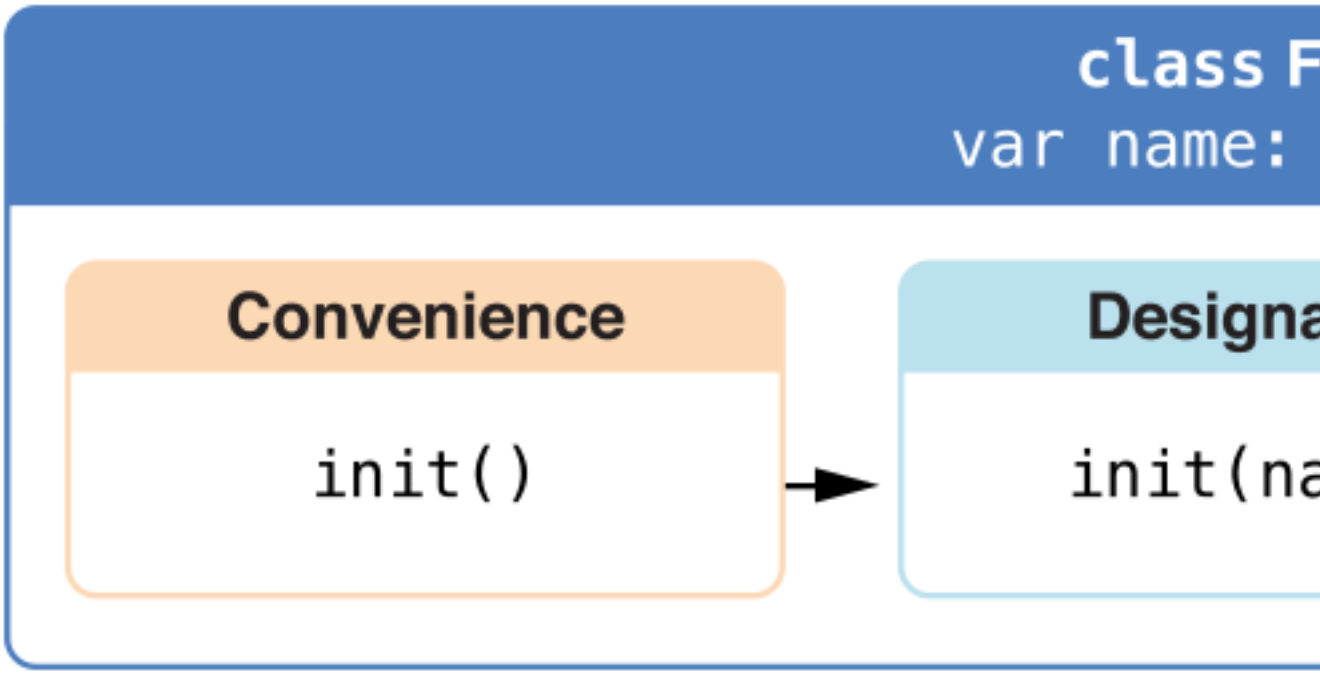
下面的例子演示的是指定构造器，便捷构造器和自动构造器继承的实战。例子中定义了三个类分别叫 Food，RecipeIngredient 和 ShoppingListItem，并给出了他们的继承关系。

基类叫做 Food，是一个简单的类只有一个 name 属性：

```
class Food {
    var name: String
    init(name: String) {
```

```
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下图就是 Food 类的构造链：



类不存在成员逐一构造器，所以 Food 类提供了一个指定构造器，使用参数 name 来完成初始化：

```
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"
```

init(name:String) 构造器就是 Food 类中的指定构造器，因为它保证了每一个 Food 实例的属性都被初始化了。由于它没有父类，所以不需要调用 super.init() 构造器。

Food 类也提供了便捷构造器 init()，这个构造器没有参数，仅仅只是将 name 设置为了 [Unnamed]：

```
let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"
```

下一个类是 Food 的子类，叫做 RecipeIngredient。这个类描述的是做饭时候的配料，包括一个数量属性 Int 类型，然后定义了两个构造器：

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

下图表示这两个类的构造链：

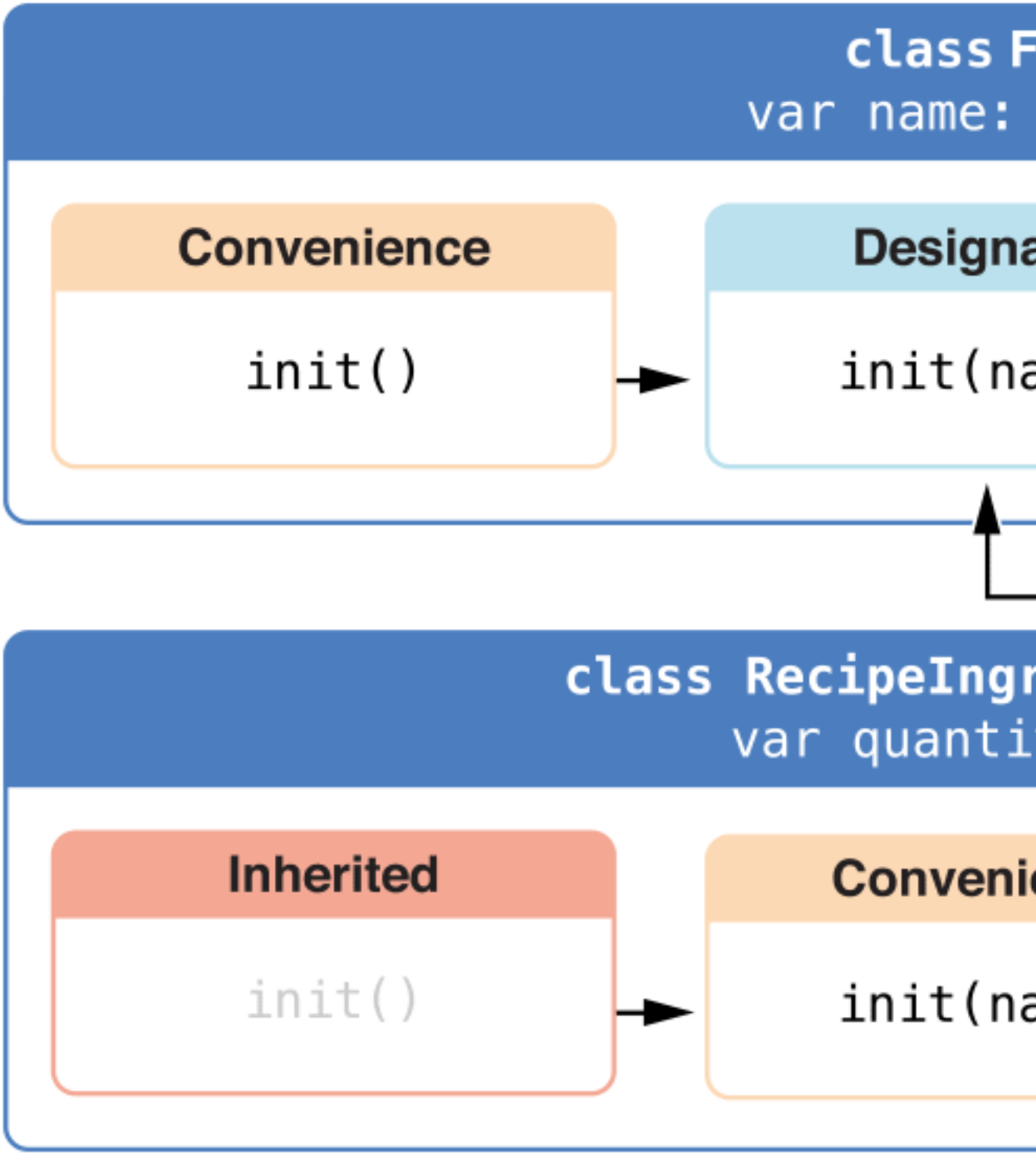
RecipeIngredient 类有它自己的指定构造器 init(name: String, quantity:Int)，用来创建一个新的 RecipeIngredient 实例。在这个指定构造器中它调用了父类的指定构造器 init(name:String)。

然后它还有一个便捷构造器，init(name)，它使用了同一个类中的指定构造器。当然它还包括一个继承来的默认构造器 init()，这个构造器将使用 RecipeIngredient 中的 init(name: String) 构造器。

RecipeIngredient also defines a convenience initializer, init(name: String), which is used to create aRecipeIngredient instance by name alone. This convenience initializer assumes a quantity of 1 for anyRecipeIngredient instance that is created without an explicit quantity. The definition of this convenience initializer makes RecipeIngredient instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity RecipeIngredient instances. This convenience initializer simply delegates across to the class's designated initializer.

Note that the init(name: String) convenience initializer provided by RecipeIngredient takes the same parameters as the init(name: String) designated initializer from Food. Even though RecipeIngredient provides this initializer as a convenience initializer, RecipeIngredient has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, RecipeIngredient automatically inherits all of its superclass's convenience initializers too.

In this example, the superclass for RecipeIngredient is Food, which has a single convenience initializer calledinit(). This initializer is therefore inherited by


class F
var name:
Convenience
init()
Designa
init(na
class RecipeIngr
var quanti
Inherited
init()
Conveni
init(na

RecipeIngredient. The inherited version of init() functions in exactly the same way as the Food version, except that it delegates to the RecipeIngredient version of init(name: String) rather than the Food version.

上述三种构造器都可以用来创建 RecipeIngredient 实例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

最后一个类是 ShoppingListItem 继承自 RecipeIngredient，它又包括了另外两个属性，是否已购买 purchased，描述 description，描述本身还是一个计算属性：

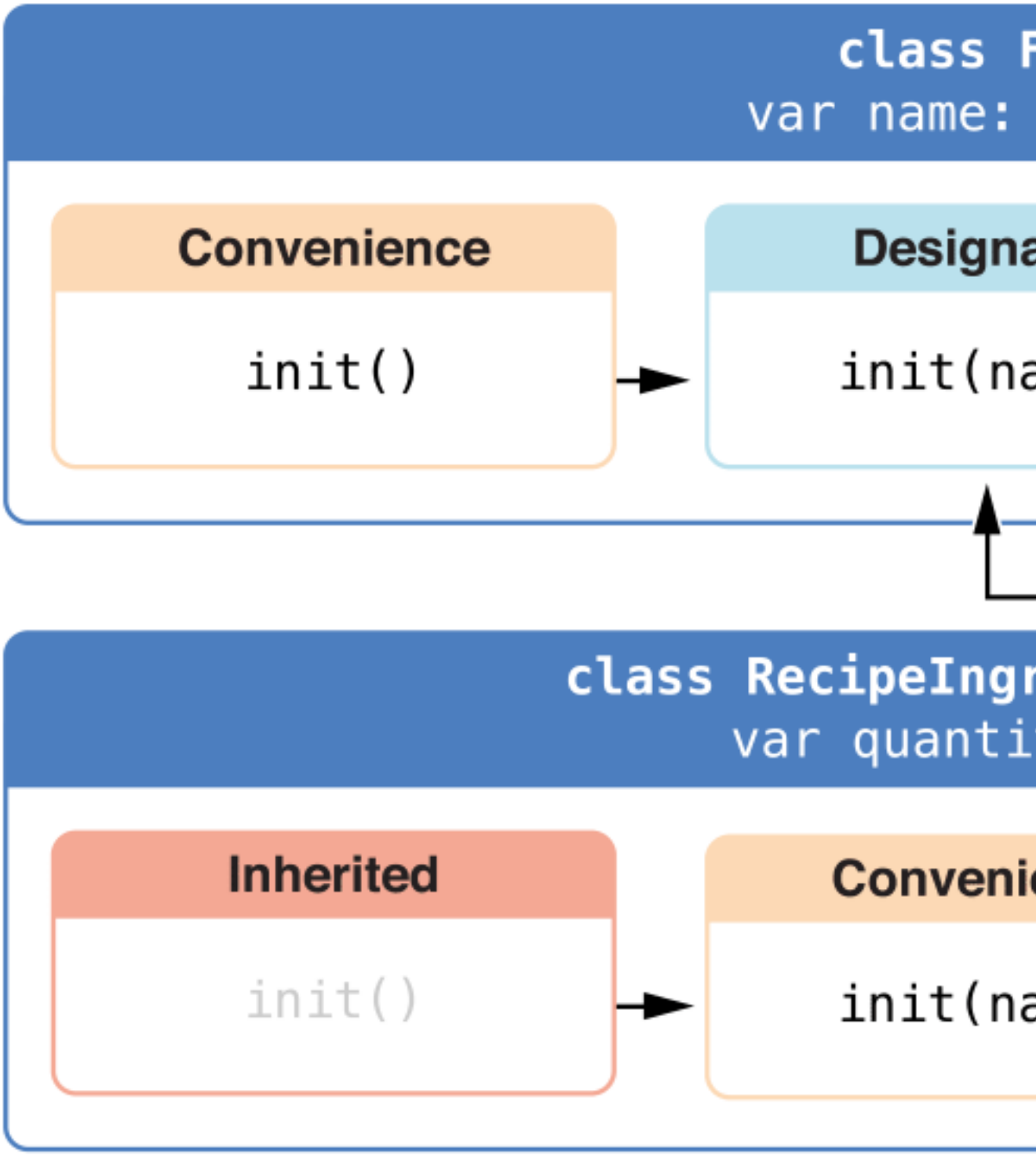
```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
    var output = "\(quantity) x \(name.lowercaseString)"
        output += purchased ? " yes" : " no"
        return output
    }
}
```

> 注意：ShoppingListItem 没有定义构造器来初始化 purchased 的值，因为每个商品在买之前 purchased 都是默认被设置为没有被购买的。

因为 ShoppingListItem 没有提供其他构造器，那么它就完全继承了父类的构造器，用下图可以说明：

你可以在创建 ShoppingListItem 实例时使用所有的继承构造器：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
```


class R
var name:
Convenience
init()
Designa
init(na
class RecipeIngr
var quanti
Inherited
init()
Convenie
init(na
class ShoppingListItem
var purchase
Inherited
Inherit

```
    println(item.description)
}
// 1 x orange juice yes
// 1 x bacon no
// 6 x eggs no
```

通过输出可以看出所有的实例在创建的时候，属性的默认值都被正确的初始化了。

11.10 通过闭包或者函数来设置一个默认属性值

如果存储属性的默认值需要额外的特殊设置，可以使用闭包或者函数来完成。

闭包或者函数会创建一个临时变量来作为返回值为这个属性赋值。下面是如果使用闭包赋值的一个示意代码：

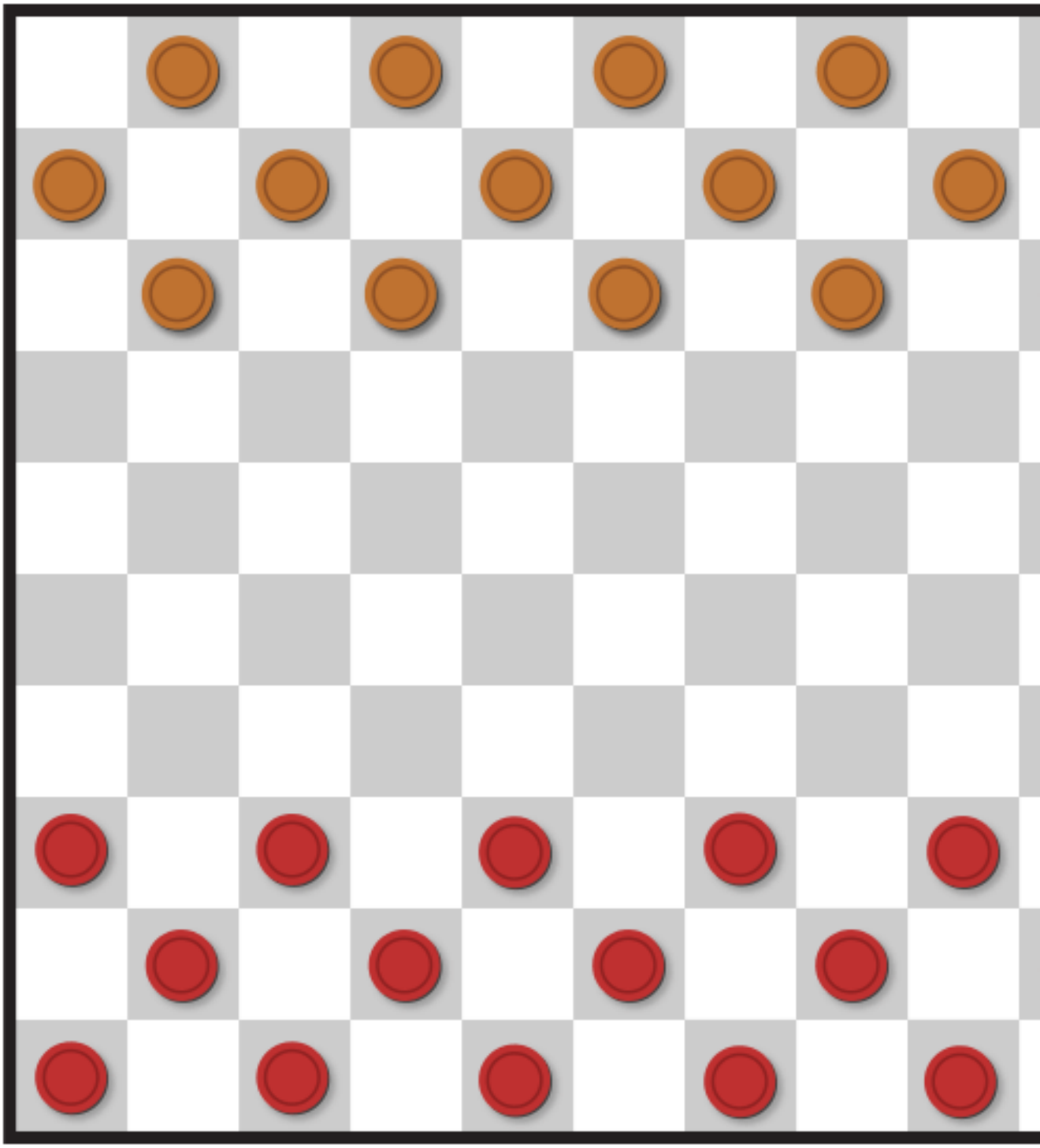
```
class SomeClass {
    let someProperty: SomeType = {
       // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
        }()
}
```

需要注意的是在闭包结尾有两个小括号，告诉 Swift 这个闭包是需要立即执行的。

> 注意：如果你时候闭包来初始化一个属性，在闭包执行的时候，后续的一些属性还没有被初始化。在闭包中不要访问任何后面的属性，一面发生错误，也不能使用 self 属性，或者其它实例方法。

下面的例子是一个叫 Checkerboard 的结构，是由游戏 Checkers 来的

这个游戏是在一个 10×10 的黑白相间的格子上进行的。来表示这个游戏盘，使用了一个叫 Checkerboard 的结构，其中一个属性叫 boardColors，是一个 100 个 Bool 类型的数组。true 表示这个格子是黑色，false 表示是白色。那么在初始化的时候可以通过下面的代码来初始化：



```
struct Checkerboard {
    let boardColors: Bool[] = {
        var temporaryBoard = Bool[]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
        }()
    func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
        return boardColors[(row * 10) + column]
    }
}
```

当一个新的 Checkerboard 实例创建的时候，闭包会执行，然后 boardColor 的默认值将会被依次计算并且返回，然后作为结构的一个属性。通过使用 squareIsBlackAtRow 工具函数可以检测是否被正确设置：

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// prints "true"
println(board.squareIsBlackAtRow(9, column: 9))
// prints "false"
```

12 Swift 中文教程（十五）析构

在一个类的实例被释放之前，析构函数会被调用。用关键字 deinit 来定义析构函数，类似于初始化函数用 init 来定义。析构函数只适用于 class 类型。

12.1 析构过程原理

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析构函数。析构函数不带任何参数，在写法上不带括号:

```
deinit {
// □ □ □ □ □ □
}
```

析构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析构函数。子类继承了父类的析构函数，并且在子类析构函数实现的最后，父类的析构函数被自动调用。即使子类没有提供自己的析构函数，父类的析构函数也总是被调用。

因为直到实例的析构函数被调用时，实例才会被释放，所以析构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)。

12.2 析构器操作

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

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

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

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

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

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

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

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

Player 类定义了一个 winCoins 方法，该方法从银行获取一定数量的硬币，并把它们添加到玩家的钱包。Player 类还实现了一个析构函数，这个析构函数在 Player 实例释放前一步被调用。这里析构函数只是将玩家的所有硬币都返回给银行：

```
var playerOne: Player? = Player(coins: 100)
println(“ A new player has joined the game with (playerOne!.coinsInPurse) coins ”)
// □ □  “ A new player has joined the game with 100     coins ”
```

```
println(“There are now (Bank.coinsInBank) coins left    in the bank”)
// 打印 “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，是因为玩家可以随时离开游戏。设置为可选使得你可以跟踪当前是否有玩家在游戏中。

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

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

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

```
playerOne = nil
println(“PlayerOne has left the game”)
// 打印 “PlayerOne has left the game”
println(“The bank now has (Bank.coinsInBank) coins”)
// 打印 “The bank now has 10000 coins”
```

玩家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，其析构函数被自动调用，其硬币被返回到银行。# Swift 中文教程（十六）自动引用计数

Swift 使用自动引用计数（ARC）来管理应用程序的内存使用。这表示内存管理已经是 Swift 的一部分，在大多数情况下，你并不需要考虑内存的管理。当实例并不再被需要时，ARC 会自动释放这些实例所使用的内存。

但是，少数情况下，你必须提供部分代码的额外信息给 ARC，这样它才能够帮你管理这部分内存。本章阐述了这些情况并且展示如何使用 ARC 来管理应用程序的内存。

> 注意引用计数仅仅作用于类实例上。结构和枚举是值类型，而非引用类型，所以不能被引用存储和传递。

12.3 ARC 怎样工作

每当你创建一个类的实例，ARC 分配一个内存块来存储这个实例的信息，包含了类型信息和实例的属性值信息。

另外当实例不再被使用时，ARC 会释放实例所占用的内存，这些内存可以再次被使用。

但是，如果 ARC 释放了正在被使用的实例，就不能再访问实例属性，或者调用实例的方法了。直接访问这个实例可能造成应用程序的崩溃。

为了保证需要实例时实例是存在的，ARC 对每个类实例，都追踪有多少属性、常量、变量指向这些实例。当有活动引用指向它时，ARC 是不会释放这个实例的。

为实现这点，当你将类实例赋值给属性、常量或变量时，指向实例的一个强引用（strong reference）将会被构造出来。被称为强引用是因为它稳定地持有这个实例，当这个强引用存在是，实例就不能够被释放。

12.4 ARC 实例

下面的例子展示了 ARC 是怎样工作的。定义一个简单的类 Person，包含一个存储常量属性 name:

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

Person 类有一个初始化方法来设置属性 name 并打印一条信息表明这个初始化过程。还有一个析构方法打印实例被释放的信息。

下面的代码定义了是三个 Person? 类型的变量，随后的代码中，这些变量用来设置一个 Person 实例的多重引用。因为这些变量是可选类型（Person?），它们自动被初始化为 nil，并且不应用任何 Person 实例。

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

现在你可以创建一个 Person 实例并赋值给其中一个变量：

reference1 = Person(name: “John Appleseed”) // prints “John Appleseed is being initialized”

注意这条信息：““John Appleseed is being initialized”，指出类 Person 的构造器已经被调用。

因为新的 Person 实例被赋值给变量 reference1，因此这是一个强引用。由于有一个强引用的存在，ARC 保证了 Person 实例在内存中不被释放掉。

如果你将这个 Person 实例赋值给更多的变量，就建立了相应数量的强引用：

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

现在有三个强引用指向这个 Person 实例了。

如果你将 nil 赋值给其中两个变量从而切断了这两个强引用（包含原始引用），还有一个强引用是存在的，因此 Person 实例不被释放。

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

直到第三个强引用被破坏之后，ARC 才释放这个 Person 实例，因此之后你就不能在使用这个实例了：

```
reference3 = nil
```

12.5 类实例间的强引用循环

在上面的例子中，ARC 跟踪指向 Person 实例的引用并保证只在 Person 实例不再被使用后才释放。

但是，写出一个类的实例没有强引用指向它这样的代码是可能的。试想，如果两个类实例都有一个强引用指向对方，这样的情况就是强引用循环。

通过在类之间定义弱的（weak）或无主的（unowned）引用可以解决强引用循环这个问题。这些方法在 “解决类实例间的强引用循环 “（“Resolving Strong

Reference Cycles Between Class Instances”）描述。但是，在学习怎样解决这个问题前，先来理解这样的循环是怎样造成的。

下面的例子描述了强引用循环是怎样无意中被造成的。定义了两个类 Person 和 Apartment，建立了公寓和它的住户间的关系

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

每个 Person 实例有一个 String 类型的属性 name 和一个初值为 nil 的可选属性 apartment。之所以 apartment 是可选属性，因为一个住户可能并没有一个公寓。

同样，每个 Apartment 实例有一个 Int 类型的属性 number 和一个初值为 nil 的可选属性 tenant，一个公寓（apartment）并不总是有人居住，所以 tenant 是可选属性。

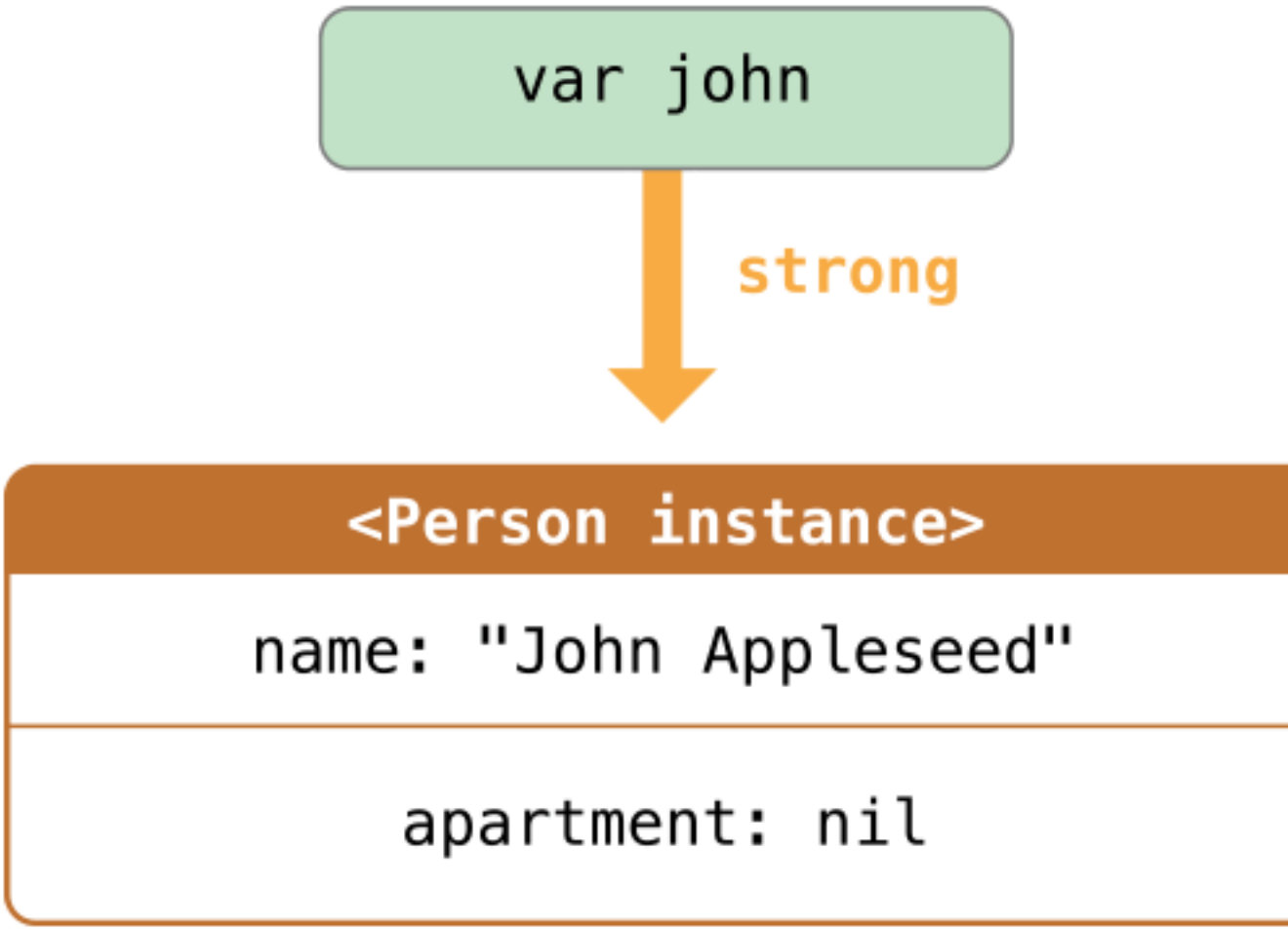
两个类都定义了析构方法，打印表明类实例被析构的语句，这告诉你 Person 和 Apartment 实例是否如愿的被释放掉了。

下面的代码定义了两个可选类型变量 john 和 number73，将用来设置之后的 Apartment 和 Person 实例。两个变量都被初始化为 nil:

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

下面创建两个 Person 实例和 Apartment 实例赋值给上面的变量:

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

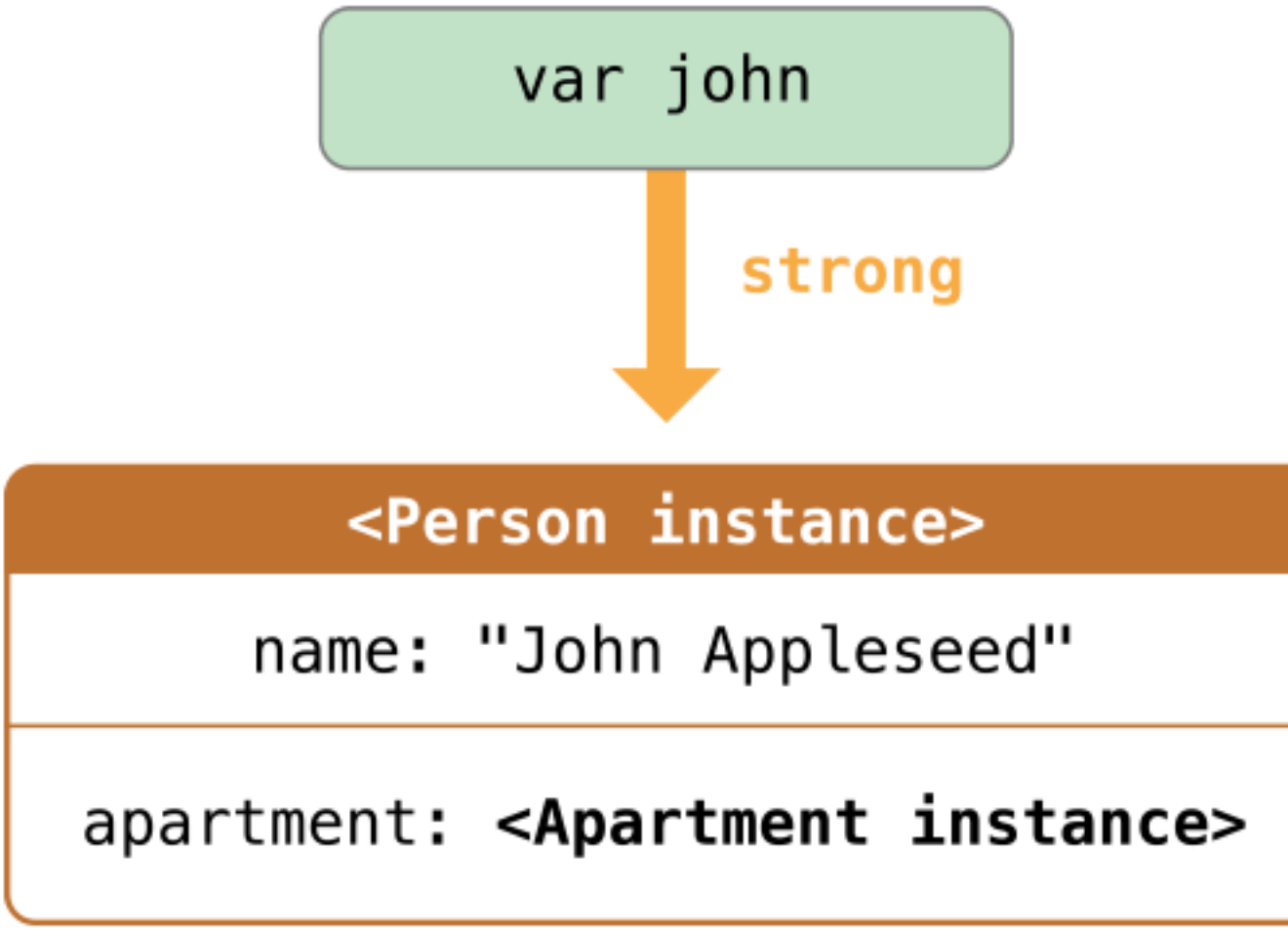

var john
strong
<Person instance>
name: "John Appleseed"
apartment: nil

下面的图表明在创建这两个实例并赋值后的样子，变量 john 有一个指向 Person 实例的强引用，变量 number73 有一个指向 Apartment 实例的强引用：

现在你可以将这两个实例连接起来，使得一个住户与一个公寓一一对应起来，注意感叹号用来解开（? unwrap）并访问 john 和 number73 中的可选变量，因此属性可以被设置：

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

下图是连接后的强引用管理图示：



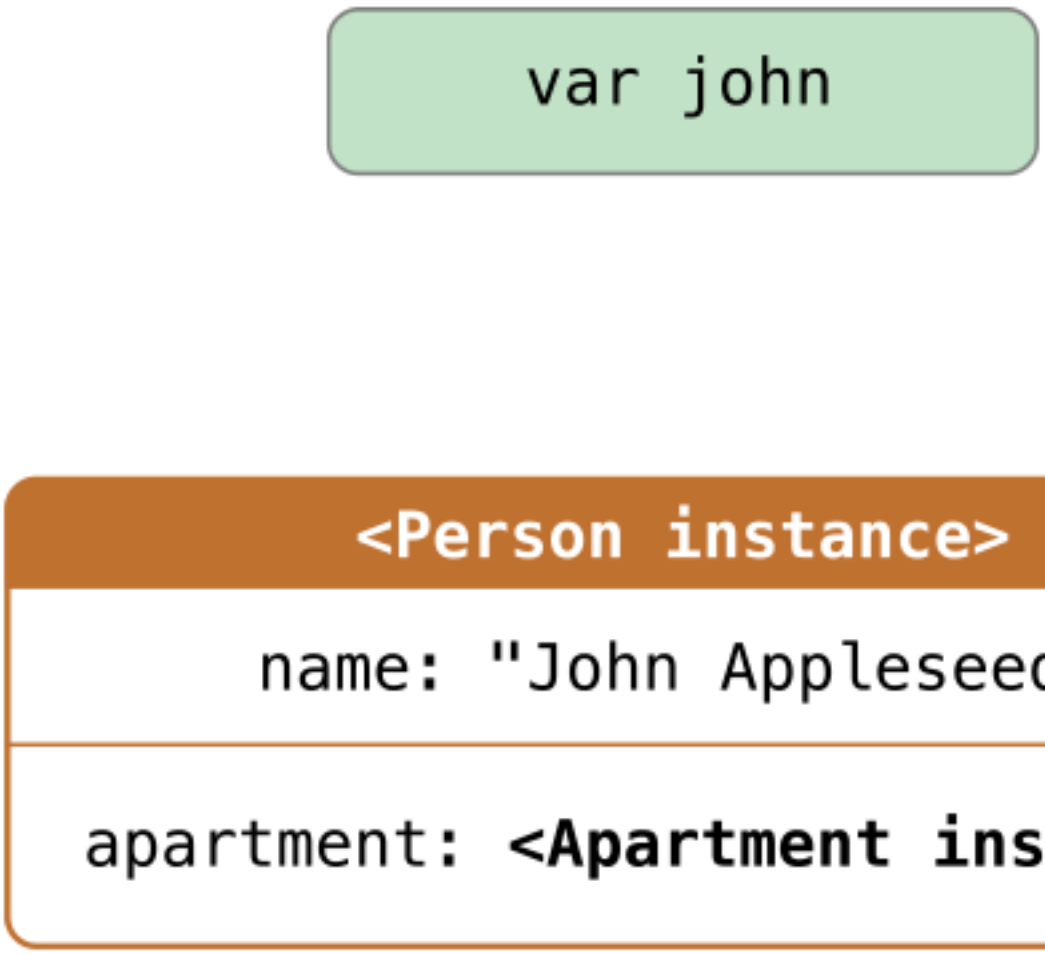
不幸的是，这样做造成了两个实例间的强引用循环。因此，当你破坏 john 和 number73 变量间的强引用、时，引用计数并没有减少到 0，ARC 也不会释放实例：

```
john = nil
```

```
number73 = nil
```

当你将两个变量设置为 nil 时，各自的析构方法都不会被调用到。强引用循环防止了 Person 和 Apartment 实例被释放造成的内存泄漏。

下图是设置 john 和 number73 为 nil 后的情况：



Person 实例和 Apartment 之间的强引用并没有被破坏掉。

12.6 解决类实例之间的强引用循环

Swift 提供了两种方法解决类实例属性间的强引用循环：弱引用和无主（unowned）引用。

弱引用和无主引用使得一个引用循环中实例并不需要强引用就可以指向循环中的其他实例。互相引用的实例就不用形成一个强引用循环。

当在生命周期的某些时刻引用可能变为 nil 时使用弱引用。相反，当引用在初始化期间被设置后不再为 nil 时使用无主引用。

12.6.1 弱引用

弱引用并不保持对所指对象的强烈持有，因此并不阻止 ARC 对引用实例的回收。这个特性保证了引用不成为强引用循环的一部分。指明引用为弱引用是在生命属性或变量时在其前面加上关键字 weak。

使用弱引用不管在生命周期的某时刻是否有值。如果引用一直有值，使用无主引用（见无主引用节)。在上例中，公寓不可能一直都有住户，所以应该使用弱引用，来打破强引用循环。

> 注意弱引用必须声明为变量，指明它们的值在运行期可以改变。弱引用不能被声明为常量。

因为弱引用可以不含有值，所以必须声明弱引用为可选类型。因为可选类型使得 Swift 中的不含有值成为可能。

因为弱引用的这个特性，所以当弱引用指向实例时实例仍然可以被释放。实例释放后，ARC 将弱引用的值设置为 nil。你可以想其他可选类型一样检查弱引用的值，你也不会使用所指向实例不存在的的引用。

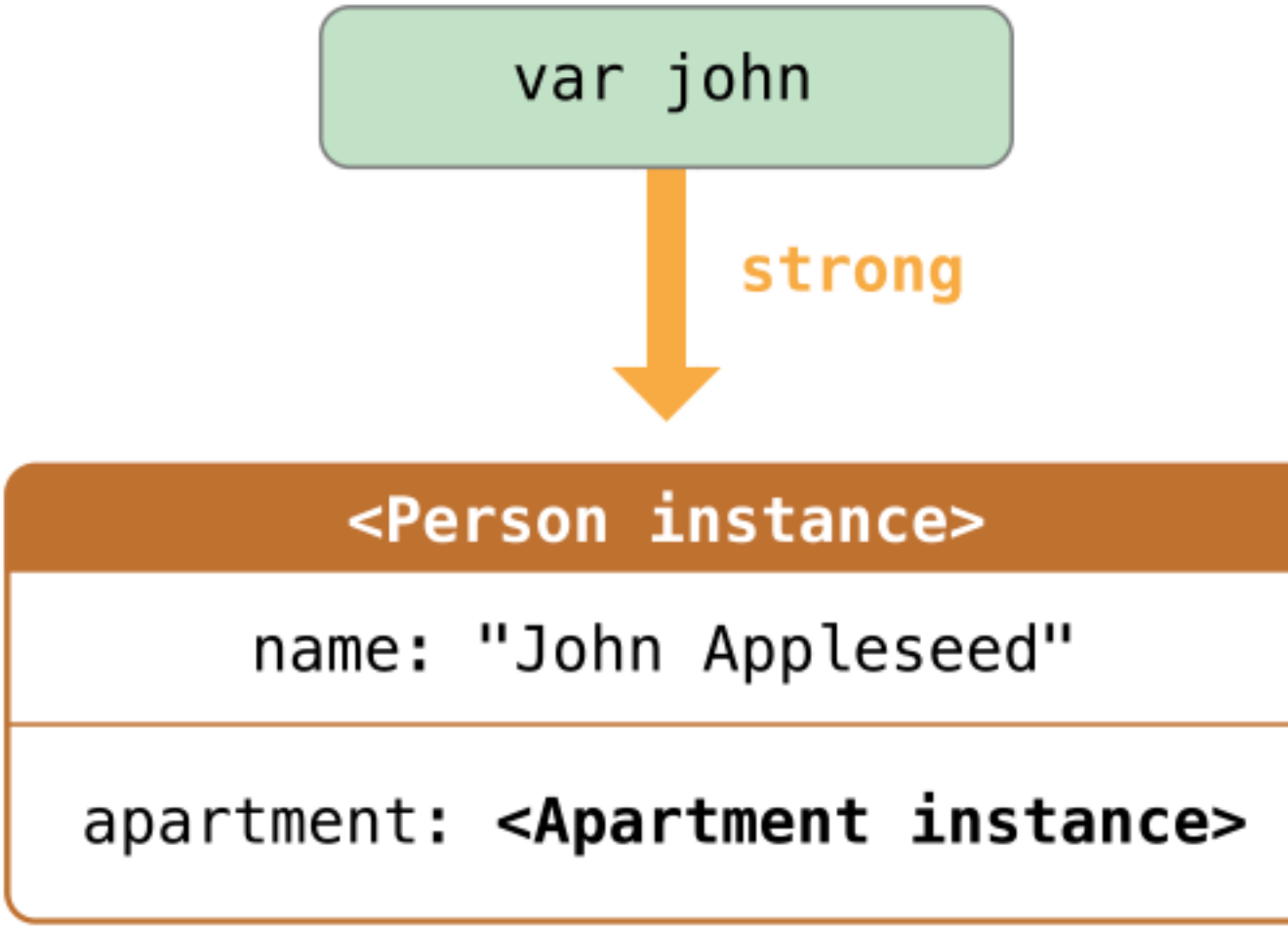
与上面的例子不同，下例声明了 Apartment 的属性 tenant 为弱引用：

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

依然像前例一样创建两个变量（john 和 number73）和它们之间的强引用：

```
var john: Person?
var number73: Apartment?
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
john!.apartment = number73
number73!.tenant = john
```

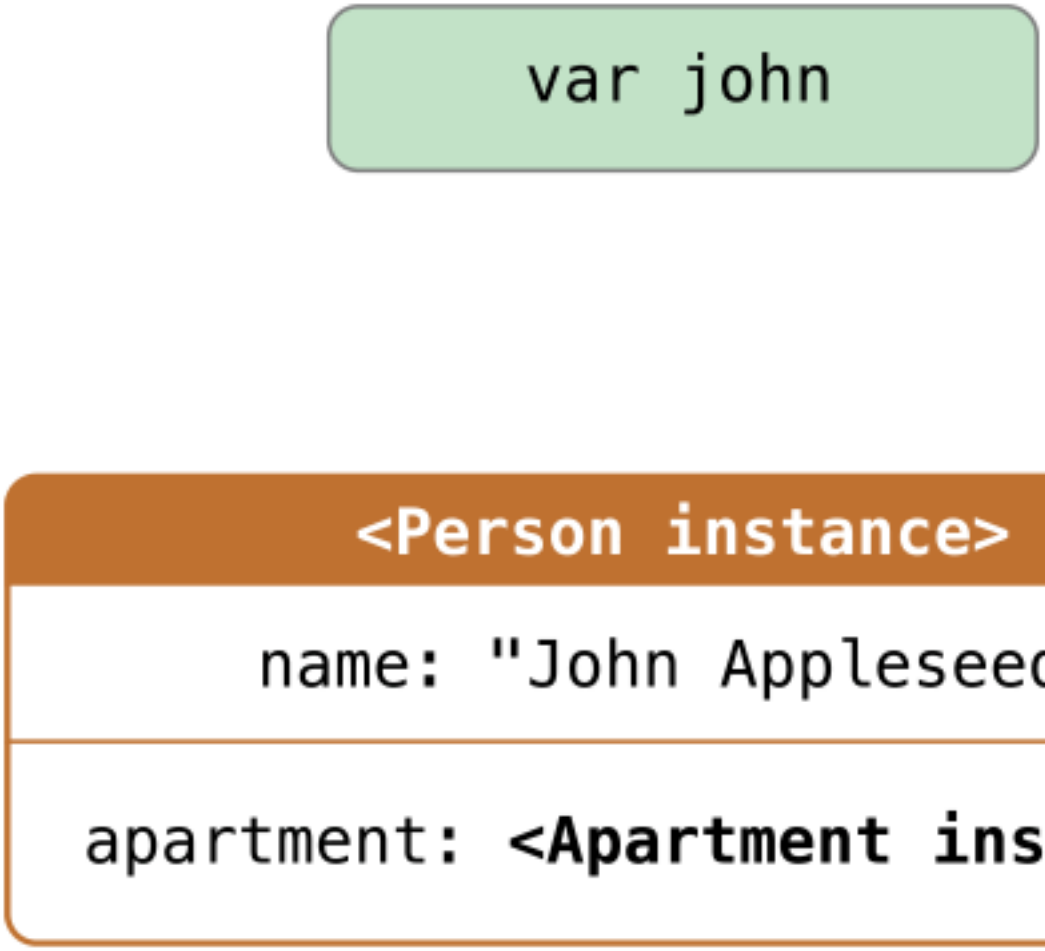
下图是两个实例之间是怎样引用的：



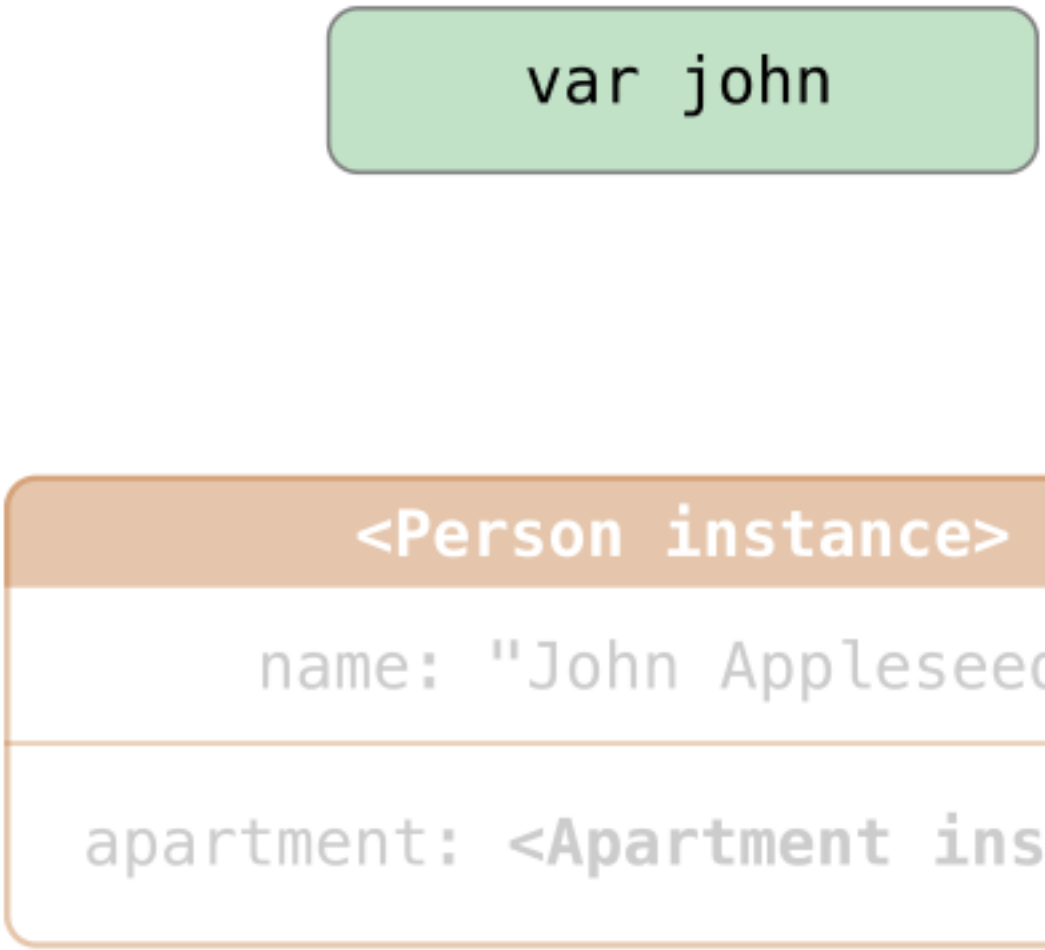
Person 实例仍然有一个到实例 Apartment 的强引用，相反实例 Apartment 实例只有一个到 Person 实例弱引用。意味着当你破坏 john 变量持有的强引用时，到 Person 实例的强引用就不存在了。

因为没有到 Person 实例的强引用，实例可以释放：

```
john = nil
```


var john
<Person instance>
name: "John Appleseed"
apartment: <Apartment instance>

仅存的强引用是从变量 number73 到 Apartment 实例的强引用，当你破坏它时，这个强引用就不存在了：



因此可以释放 Person：

```
number73 = nil
```

上面两段代码展示了 Person 实例和 Aparment 实例的析构，当两个变量分别被设置为 nil 时，就打印各自的析构提示信息，展示了引用循环被打破了。

12.6.2 无主引用

和弱引用一样，无主引用也并不持有实例的强引用。但和弱引用不同的是，无主引用通常都有一个值。因此，无主引用并不定义成可选类型。指明为无主引用是在属性或变量声明的时候在之前加上关键字 unowned。

因为无主引用非可选类型，所以每当使用无主引用时不必解开（unwrap?）它。无主引用通常可以直接访问。但是当无主引用所指实例被释放时，ARC 并不能将引用值设置为 nil，因为非可选类型不能设置为 nil。

> 注意在无主引用指向实例被释放后，如果你像访问这个无主引用，将会触发一个运行期错误（仅当能够确认一个引用一直指向一个实例时才使用无主引用)。在 Swift 中这种情况也会造成应用程序的崩溃，会有一些不可预知的行为发生，尽管你可能已经采取了一些预防措施

接下来的例子定义了两个类，Customer 和 CreditCard，表示一个银行客户和信用卡。这两个类的属性各自互相存储对方类实例。这种关系存在着潜在的强引用循环。

这两个类之间的关系稍微和前面的 Person 和 Apartment 有些不同。在此例中，一个客户可能有也可能没有一个信用卡，但是一个信用卡必须由一个客户持有。因此，类 Customer 有一个可选的 card 熟悉，而类 CreditCard 有一个非可选 customer 属性。

另外，创建 CreditCard 实例时必须必须向其构造器传递一个值 number 和一个 customer 实例。这保证了信用卡实例总有一个客户与之联系在一起。

因为信用卡总由一个用户持有，所以定义 customer 属性为无主引用，来防止强引用循环。

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { println("\(name) is being deinitialized") }
}
class CreditCard {
    let number: Int
    unowned let customer: Customer
    init(number: Int, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { println("Card #\(number) is being deinitialized") }
```

```
}
```

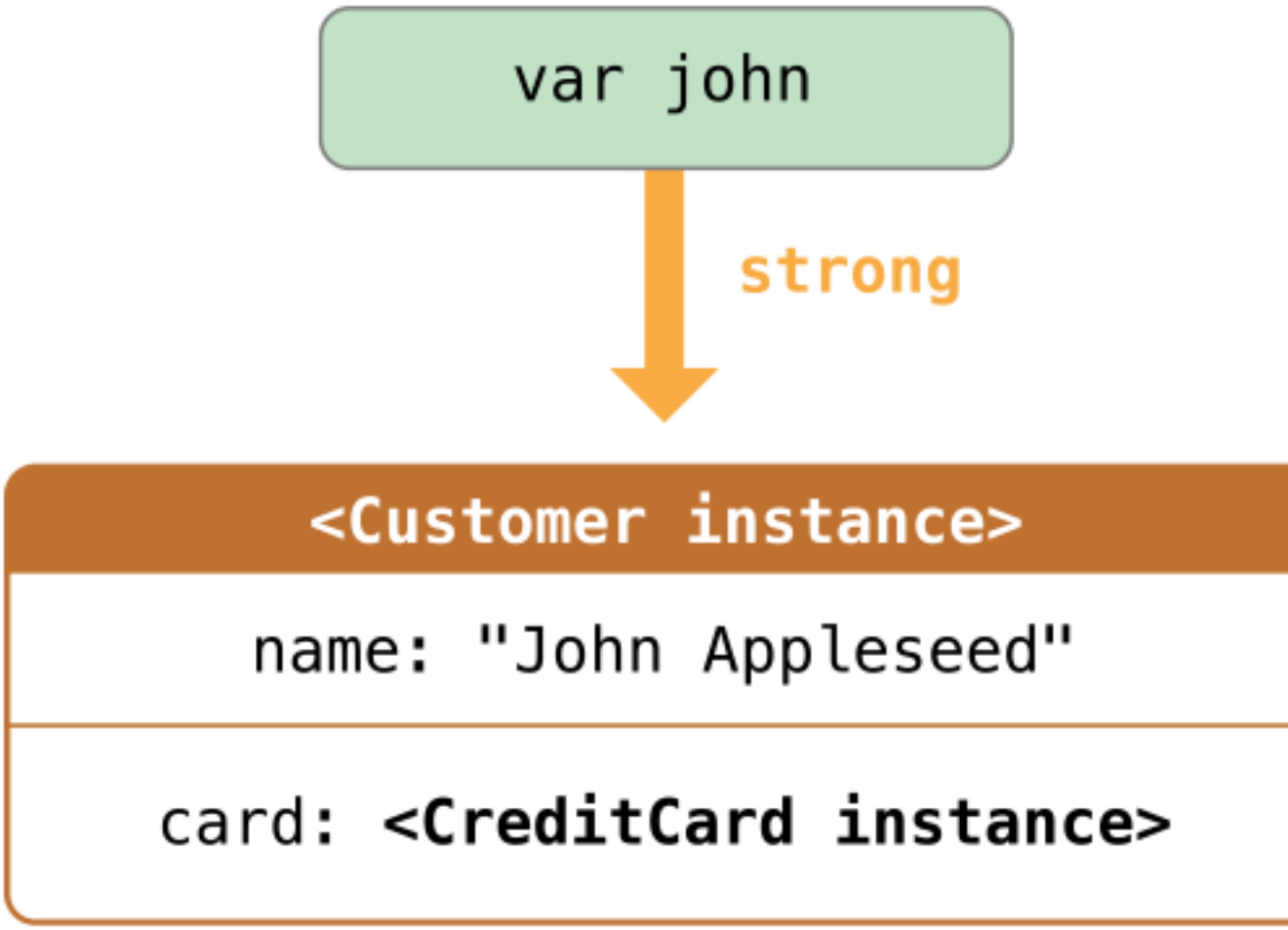
下面的代码段定义了 Customer 类型可选变量 john，用来存储一个特定用户的引用，这个变量初值为 nil：

```
var john: Customer?
```

现在可以创建一个 Customer 实例，并初始化一个新的 CreditCard 实例来设置 customer 实例的 card 属性：

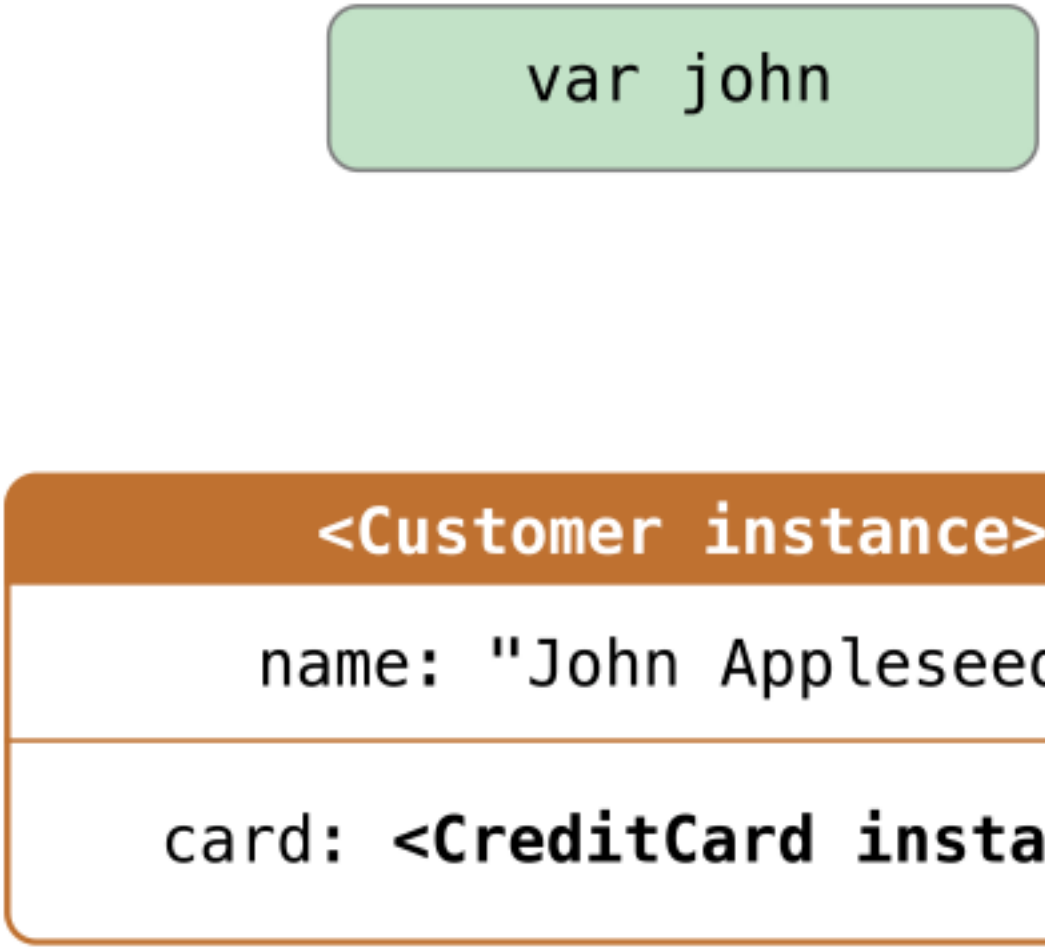
```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

下图是引用间的连接管理：



Customer 实例有一个到 CreditCard 实例的强引用，CreaditCard 实例有一个到 Customer 实例无主引用。

因为无主引用的存在，当你破坏变量 john 持有的强引用时，就再也没有到 Customer 实例的强引用了。



因为没有到 Customer 实例的强引用，实例被释放了。之后，到 CreditCard 实例的强引用也不存在了，因此这个实例也被释放了：

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

上面的代码段显示了变量 john 设置为 nil 后 Customer 实例和 CreditCard 实例被析构的信息。

12.6.3 无主引用和隐式拆箱可选属性

上面的弱引用和无主引用例子是更多常见场景中的两个，表面打破强引用循环是必要的。

例子 Person 和 Apartment 显示了当互相引用的两个属性被设置为 nil 时可能造成强引用循环。这种情况可以使用弱引用来解决。

例子 Customer 和 CreditCard 显示了一个属性可以设置为 nil，而另一个不可以为 nil 时可能造成的强引用循环。这种情况可以使用无主引用解决。

但是，有第三种情况，两个属性都一直有值，并且都不可以被设置为 nil。这种情况下，通常联合一个类种的无主属性和一个类种的隐式装箱可选属性 (implicitly unwrapped optional property)。

这保证了两个属性都可以被直接访问，并且防止了引用循环。这一节展示了如何进行这样的设置。

下面的例子定义了两个类，Coutry 和 City，两者的属性都存放另外一个类的实例。在数据模型中，每个国家都有一个首都，而每个城市都属于一个国家。代码如下：

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

为表达这样的关系，City 的构造器有一个参数为 Country 实例，并将他存为 country 属性。

City 的构造器也在 Country 的构造器中被调用。但是直到一个新的 Country 实例被完整地初始化，Country 构造器不能传递 self 给 City 的构造器（在两阶段初始化 “Two-Phase Initialization” 节描述）

为处理这样的情况，声明 Country 的属性 capitalCity 属性为隐式拆箱可选属性，通过在类型注解后加检测符号实现（City!）。这表明 capitalCity 属性像其他可选属性一样有一个默认值 nil，但是可以不需要对值拆箱就可以访问（隐式拆箱选项（implicitly unwrapped optionals）描述）

因为 capitalCity 有一个默认值 nil，所以当在 Country 的构造器中设置了 name 属性的值后，一个新的 Country 实例就完全地被初始化了，这表明 Country 构造器已经可以传递隐式 self 属性，从而设置 capitalCity 属性值。

这表明你可以在一个单独的语句中创建 Country 和 CIty 实例，不会造成强引用循环，并且可以不用使用检测符号（!）解包可选值来直接访问 capitalCity 属性：

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

上面的例子中，隐式解包选项的使用表示分阶段初始化是可行的。当初始化完成时，capitalCity 属性可以像一个非可选值那样访问，并且不会造成强引用循环。

12.7 闭包的强引用循环

前面你知道了当两个类实例持有对方的强引用时强引用循环是怎样被创建的。你也知道了怎样使用弱引用和无主引用来破坏强引用循环。

当将一个闭包赋值给一个类实例的属性，并且闭包体捕获这个实例时，也可能存在一个强引用循环。捕获实例是因为闭包体访问了实例的属性，就像 self.someProperty，或者调用了实例的方法，就像 self.someMethod ()。不管哪种情况，这都造成闭包捕获 self，造成强引用循环。

这个强引用循环的存在是因为闭包和类一样都是引用类型。当你将闭包赋值给属性时，就给这个闭包赋值了一个引用。本质上和前面的问题相同 – 两个强引用都互相地指向对方。但是，与两个类实例不同，这里是一个类与一个闭包。

Swift 为这个问题提供了一个优美的解决方法，就是闭包捕获列表。但是，在学习怎样通过闭包捕获列表破坏强引用循环以前，有必要了解这样的循环是怎样造成的。

下面的例子展示了当使用闭包引用 self 时强引用循环是怎样造成当。定义了一个名为 HTMLElement 的类，建模了 HTML 文档中的一个单独的元素：

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

这个 HTMLElement 类定义了一个表示元素（例如 “p“，”br“）名称的属性 name，和一个可选属性 text，表示要在页面上渲染的 html 元素的字符串的值

另外，还定义了一个懒惰属性 asHTML。这个属性引用一个闭包，这个闭包结合 name 与 text 形成一个 html 代码字符串。这个属性类型是 () − > String，表示一个函数不需要任何参数，返回一个字符串值。

默认地，asHTML 属性赋值为返回 HTML 标签字符串的闭包。这个标签包含了可选的 text 值。对一个段落而言，闭包返回”

some text

” 或者”

”，取决其中的 text 属性为 “some text” 还是 nil。

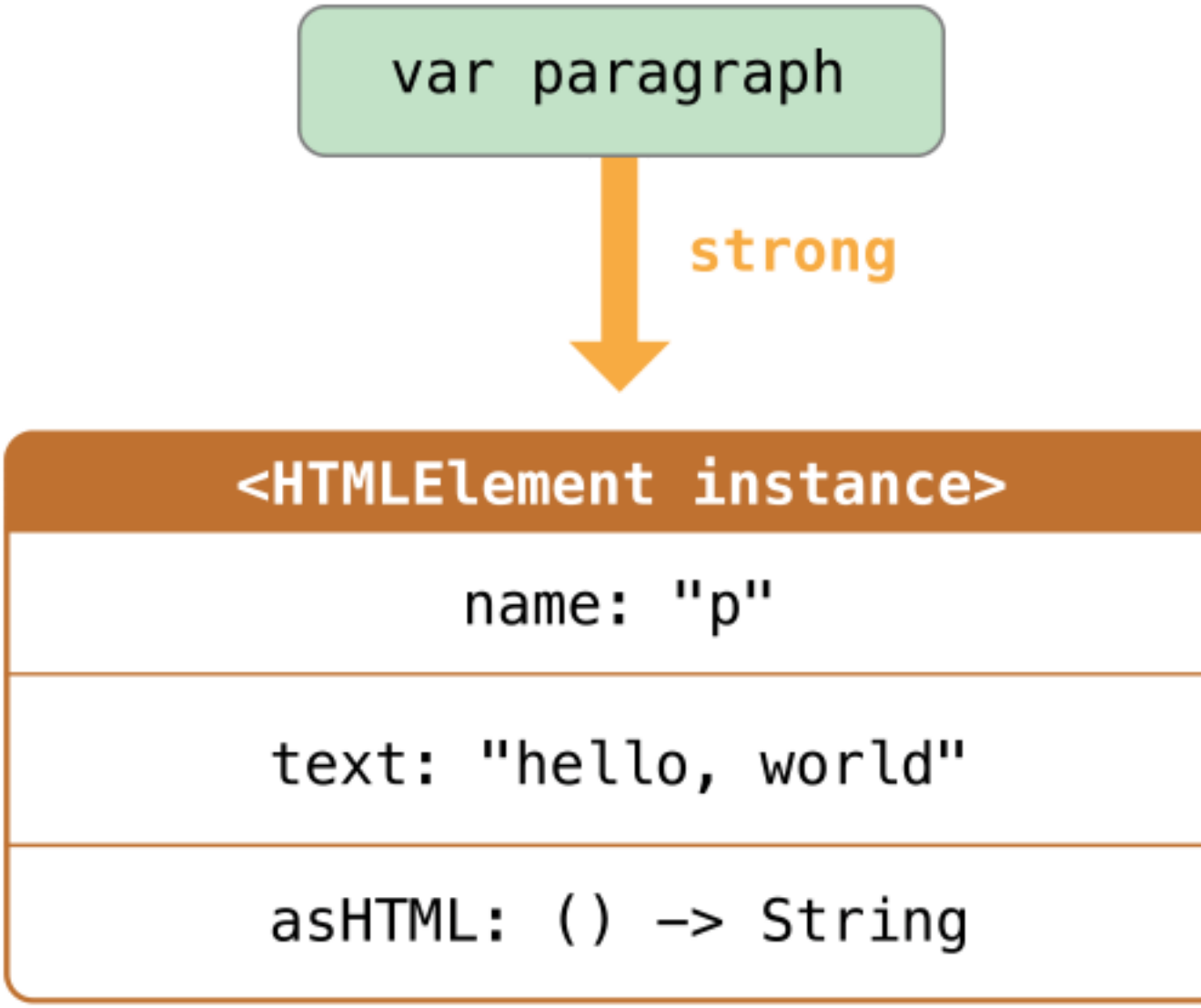
asHTML 属性的命名和使用都和实例方法类似，但是，因为它是一个闭包属性，如果想渲染特定的 html 元素，你可以使用另外一个闭包来代替 asHTML 属性的默认值。

这个 HTMLElement 类提供单一的构造器，传递一个 name 和一个 text 参数。定义了一个析构器，打印 HTMLElement 实例的析构信息。

下面是如何使用 HTMLElement 类来创建和打印一个新的实例：

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

不幸的是，上面所写的 HTMLElemnt 类的实现会在 HTMLElement 实例和闭包所使用的默认 asHTML 值之间造成强引用循环，下面是其图示：



实例的 asHTML 属性持有其闭包的一个强引用，但是因为闭包在其类内引用 self（self.name 和 self.text 方式），闭包捕获类本身，意味着它也持有到 HTMLElement 实例的引用。强引用循环就这样建立了。(关于闭包捕获值的更多信息，参见 CapturingValues)

如果设置 paragraph 变量值为 nil，破坏了到 HTMLElement 实例的强引用，实例和其闭包都不会被析构，因为强引用循环：

```
paragraph = nil
```

注意 HTMLElement 析构器中的提示信息不会被打印，表示 HTMLElement 实例并没有被析构。

12.8 解决闭包的强引用循环

通过定义捕获列表为闭包的一部分可以解决闭包和类实例之间的强引用循环。捕获列表定义了在闭包体内何时捕获一个或多个引用类型的规则。像解决两个类实例之间的强引用循环一样，你声明每个捕获引用为弱引用或者无主引用。究竟选择哪种定义取决于代码中其他部分间的关系

12.8.1 定义捕获列表

捕获列表中的每个元素由一对 weak ／ unowned 关键字和类实例（self 或 someInstance）的引用所组成。这些对由方括号括起来并由都好分隔。

将捕获列表放在闭包参数列表和返回类型（如果提供）的前面：

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

如果闭包没有包含参数列表和返回值，它们可以从上下文中推断出来的话，将捕获列表放在闭包的前面，后面跟着关键字 in：

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

12.8.2 弱引用和无主引用

当闭包和实例之间总是引用对方并且同时释放时，定义闭包捕获列表为无主引用。

当捕获引用可能为 nil，定义捕获列表为弱引用。弱引用通常是可选类型，并且在实例释放后被设置为 nil。这使得你可以在闭包体内检查实例是否存在。

在例子 HTMLElement 中，可以使用无主引用来解决强引用循环问题，下面是其代码：

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

这个 HTMLELement 实现在之前的基础上在 asHTML 闭包中加上了捕获列表。这里，捕获列表是 [unowned self]，表示作为无主引用来捕获自己而不是强引用。

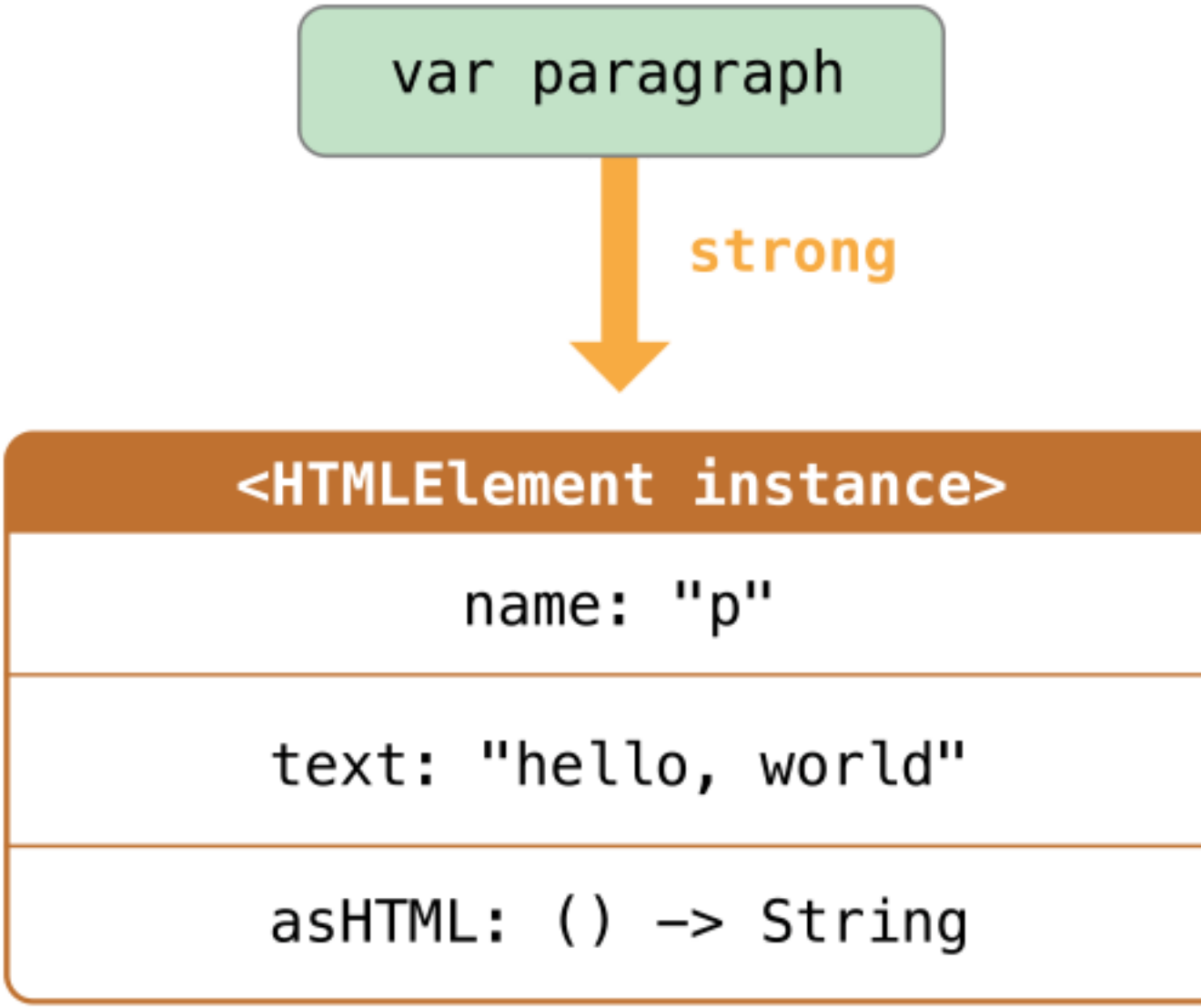
你可以像之间一样创建和打印 HTMLElement 实例：

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

下图是使用捕获列表后的引用图示：

此时，闭包捕获自身是一个无主引用，并不持有捕获 HTMLelement 实例的强引用。如果你设置 paragraph 的强引用为 nil，HTMLElement 实例就被释放了，可以从析构信息中看出来：

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


var paragraph
strong
<HTMLElement instance>
name: "p"
text: "hello, world"
asHTML: () -> String

13 Swift 中文教程（十七）可选链

可选链（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。

13.0.3 可选链可替代强制解析

通过在想调用的属性、方法、或子脚本的可选值（optional value）（非空）后面放一个问号，可以定义一个可选链。这一点很像在可选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可选值为空时可选链即刻失败，然而一般的强制解析将会引发运行时错误。

为了反映可选链可以调用空（nil），不论你调用的属性、方法、子脚本等返回的值是不是可选值，它的返回结果都是一个可选值。你可以利用这个返回值来检测你的可选链是否调用成功，有返回值即成功，返回 nil 则失败。

调用可选链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可选值，当可选链调用成功时，一个应该返回 Int 的属性将会返回 Int?。

下面几段代码将解释可选链和强制解析的不同。

首先定义两个类 Person 和 Residence。

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

Residence 具有一个 Int 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：

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

如果你想使用感叹号(!)强制解析获得这个人 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解析的 residence 值。

```
let roomCount = john.residence!.numberOfRooms
// □ □ □ □ □ □ □ □
```

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。

可选链提供了一种另一种获得 numberOfRooms 的方法。利用可选链，使用问号来代替原来! 的位置：

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// □ □ "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，可选链会返回 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，因此会出先无法访问 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。只要是通过可选链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：

```
john.residence = Residence()
```

john.residence 现在有了实际存在的实例而不是 nil 了。如果你想使用和前面一样的可选链来获得 numberOfRoooms，它将返回一个包含默认值 1 的 Int?:

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// □ □ "John's residence has 1 room(s)" □
```

13.0.4 为可选链定义模型类

你可以使用可选链来多层调用属性，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，其中包括多层可选链。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。

```
class Person {
    var residence: Residence?
}
```

Residence 类比之前复杂些。这次，它定义了一个变量 rooms，它被初始化为一个 Room[] 类型的空数组:

```
class Residence {
    var rooms = Room[]()
    var numberOfRooms: Int {
    return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        println("The number of rooms is \(numberOfRooms)")
    }
```

```
    var address: Address?
}
```

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。

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

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:

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName {
            return buildingName
        } else if buildingNumber {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

Address 类还提供了一个 buildingIdentifier 的方法，它的返回值类型为 String?。这个方法检查 buildingName 和 buildingNumber 的属性，如果 buildingName 有值则将其返回，或者如果 buildingNumber 有值则将其返回，再或如果没有一个属性有值，返回空。

13.0.5 通过可选链调用属性

正如上面 “ 可选链可替代强制解析” 中所述，你可以利用可选链的可选值获取属性，并且检查属性是否获取成功。然而，你不能使用可选链为属性赋值。

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：

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

由于 john.residence 是空，所以这个可选链和之前一样失败了，但是没有运行时错误。

13.0.6 通过可选链调用方法

你可以使用可选链的来调用可选值的方法并检查方法调用是否成功。即使这个方法没有返回值，你依然可以使用可选链来达成这一目的。

Residence 的 printNumberOfRooms 方法会打印 numberOfRooms 的当前值。方法如下：

```
func printNumberOfRooms(){
    println( “ The number of rooms is \(numberOfRooms) ” )
}
```

这个方法没有返回值。但是，没有返回值类型的函数和方法有一个隐式的返回值类型 Void（参见 Function Without Return Values）。

如果你利用可选链调用此方法，这个方法的返回值类型将是 Void?，而不是 Void，因为当通过可选链调用方法时返回值总是可选类型（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：如果方法通过可选链调用成功，printNumberOfRooms 的隐式返回值将会是 Void，如果没有成功，将返回 nil:

```
if john.residence?.printNumberOfRooms() {
    println("It was possible to print the number of rooms.")
} else {
    println("It was not possible to print the number of rooms.")
}
// □ □ "It was not possible to print the number of rooms." □
```

13.0.7 使用可选链调用子脚本

你可以使用可选链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，你不能通过可选链来设置子代码。

> 注意：当你使用可选链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。可选链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。

```
if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
    println("Unable to retrieve the first room name.")
}
// □ □ "Unable to retrieve the first room name." □
```

在子代码调用中可选链的问号直接跟在 john.residence 的后面，在子脚本括号的前面，因为 john.residence 是可选链试图获得的可选值。

如果你创建一个 Residence 实例给 john.residence，且在他的 rooms 数组中有一个或多个 Room 实例，那么你可以使用可选链通过 Residence 子脚本来获取在 rooms 数组中的实例了：

```
let johnsHouse = Residence()
johnsHouse.rooms += Room(name: "Living Room")
johnsHouse.rooms += Room(name: "Kitchen")
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
    println("Unable to retrieve the first room name.")
}
// □ □ "The first room name is Living Room." □
```

13.0.8 连接多层链接

你可以将多层可选链连接在一起，可以掘取模型内更下层的属性方法和子脚本。然而多层可选链不能再添加比已经返回的可选值更多的层。也就是说:

如果你试图获得的类型不是可选类型，由于使用了可选链它将变成可选类型。如果你试图获得的类型已经是可选类型，由于可选链它也不会提高自判断性。

因此:

如果你试图通过可选链获得 Int 值，不论使用了多少层链接返回的总是 Int?。相似的，如果你试图通过可选链获得 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。这里使用了两层可选链来联系 residence 和 address 属性，他们两者都是可选类型:

```
if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// □ □ "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?。因此尽管在可选类型属性前使用了两层可选链，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，你可以通过多层可选链来得到这个属性值。

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence!.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// □ □ "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 属性是一个可选类型，因此你需要在它获取 address 属性之前使用! 解析以获得它的实际值。

13.0.9 链接自判断返回值的方法

前面的例子解释了如何通过可选链来获得可选类型属性值。你也可以通过调用返回可选类型值的方法并按需链接方法的返回值。

下面的例子通过可选链调用了 Address 类中的 buildingIdentifier 方法。这个方法的返回值类型是 String?。如上所述，这个方法在可选链调用后最终的返回值类型依然是 String?:

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    println("John's building identifier is \(buildingIdentifier).")
}
// □ □ "John's building identifier is The Larches." □
```

如果你还想进一步对方法返回值执行可选链，将可选链问号符放在方法括号的后面:

```
if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
    println("John's uppercase building identifier is \(upper).")
}
// □ □ "John's uppercase building identifier is THE LARCHES." □
```

> 注意：在上面的例子中，你将可选链问号符放在括号后面是因为你想要链接的可选值是 buildingIdentifier 方法的返回值，不是 buildingIdentifier 方法本身。

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（十八）类型检查

类型检查是一种检查类实例的方式，并且或者也是让实例作为它的父类或者子类的一种方式。

类型检查在 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，就像在 Protocols Checking for Protocol Conformance 部分讲述的一样。

13.1 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包含了几个这些类实例的数组，作为类型检查的例子。

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媒体项提供基础功能。特别的，它声明了一个 String 类型的 name 属性，和一个 init name 初始化器。(它假定所有的媒体项都有个名称。)

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

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

```
class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

最后一个代码段创建了一个数组常量 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数组中所包含的内容推断来的。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。

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be MediaItem[]
```

在幕后 library 里存储的媒体项依然是 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。

13.2 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 。

下面的例子定义了两个变量，movieCount 和 songCount，用来计算数组 library 中 Movie 和 Song 类型的实例数量。

```
var movieCount = 0
var songCount = 0
```

```
for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}

println("Media library contains \(movieCount) movies and \(songCount) songs")
// prints "Media library contains 2 movies and 3 songs"
```

示例迭代了数组 library 中的所有项。每一次，for-in 循环设置 item 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。在循环结束后，movieCount 和 songCount 的值就是被找到属于各自的类型的实例数量。

13.3 向下转型（Downcasting）

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，用类型检查操作符 (as)

因为向下转型可能会失败，类型检查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，用类型检查的可选形式 (as?)。可选形式的类型检查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，强制形式的类型检查会触发一个运行时错误。

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检查（as?）去检查循环里的每次下转。

```
for item in library {
    if let movie = item as? Movie {
        println("Movie: '\(movie.name)', dir. \(movie.director)")
    } else if let song = item as? Song {
        println("Song: '\(song.name)', by \(song.artist)")
    }
}

// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley
```

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? 形式在试图下转时将返还一个可选值。item as Movie 的返回值是 Movie? 类型或 “optional Movie”。

当下转为 Movie 应用在两个 Song 实例时将会失败。为了处理这种情况，上面的例子使用了可选绑定（optional binding）来检查可选 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 来存储返回的可选 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。潜在的根本的实例保持不变；只是简单地把它作为它被转换成的类来使用。

13.3.1 Any 和 AnyObject 的类型检查

Swift 为不确定类型提供了两种特殊类型别名：

- AnyObject 可以代表任何 class 类型的实例。
- 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。

13.3.2 AnyObject 类型

当需要在工作中使用 Cocoa APIs，它一般接收一个 AnyObject[] 类型的数组，或者说 “一个任何对象类型的数组”。这是因为 Objective-C 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检查 (as) 来下转在数组中的每一项到比 AnyObject 更明确的类型，不需要可选解析（optional unwrapping)。

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

```
let someObjects: AnyObject[] = [
   Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")
]
```

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

```
for object in someObjects {
    let movie = object as Movie
    println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

为了变为一个更短的形式，下转 someObjects 数组为 Movie[] 类型来代替下转每一项方式。

```
for movie in someObjects as Movie[] {
    println("Movie: '\(movie.name)', dir. \(movie.director)")
```

```
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

13.4 Any 类型

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

```
var things = Any[]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
```

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:

```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
    case let someInt as Int:
        println("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        println("a positive double value of \(someDouble)")
```

```
    case is Double:
       println("some other double value that I don't want to print")
    case let someString as String:
        println("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        println("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
     println("a movie called '\(movie.name)', dir. \(movie.director)")
    default:
        println("something else")
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called 'Ghostbusters', dir. Ivan Reitman
□
```

> 注意：在一个 switch 语句的 case 中使用强制形式的类型检查操作符 (as, 而不是 as?) 来检查和转换到一个明确的类型。在 switch case 语句的内容中这种检查总是安全的。

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（十九）类型嵌套

枚举类型常被用于实现特定类或结构体的功能。也能够在有多种变量类型的环境中，方便地定义通用类或结构体来使用，为了实现这种功能，Swift 允许你定义类型嵌套，可以在枚举类型、类和结构体中定义支持嵌套的类型。

要在一个类型中嵌套另一个类型，将需要嵌套的类型的定义写在被嵌套类型的区域 {} 内，而且可以根据需要定义多级嵌套。

13.5 类型嵌套实例

下面这个例子定义了一个结构体 BlackjackCard(二十一点)，用来模拟 BlackjackCard 中的扑克牌点数。BlackjackCard 结构体包含 2 个嵌套定义的枚举类型 Suit 和 Rank。

在 BlackjackCard 规则中，Ace 牌可以表示 1 或者 11，Ace 牌的这一特征用一个嵌套在枚举型 Rank 的结构体 Values 来表示。

```
struct BlackjackCard {
    // □ □ □ □ □ □ □ Suit
    enum Suit: Character {
       case Spades = " ", Hearts = " ", Diamonds = " ", Clubs = " "
   }
    // □ □ □ □ □ □ □ Rank
    enum Rank: Int {
       case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
        case Jack, Queen, King, Ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
         switch self {
         case .Ace:
             return Values(first: 1, second: 11)
         case .Jack, .Queen, .King:
             return Values(first: 10, second: nil)
         default:
             return Values(first: self.toRaw(), second: nil)
             }
       }
    }
    // BlackjackCard □ □ □ □ □ □
    let rank: Rank, suit: Suit
    var description: String {
    var output = "suit is \(suit.toRaw()),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
```

```
            output += " or \(second)"
        }
        return output
    }
}
```

枚举型的 Suit 用来描述扑克牌的四种花色，并分别用一个 Character 类型的值代表花色符号。

枚举型的 Rank 用来描述扑克牌从 Ace~10,J,Q,K,13 张牌，并分别用一个 Int 类型的值表示牌的面值。(这个 Int 类型的值不适用于 Ace,J,Q,K 的牌)。

如上文所提到的，枚举型 Rank 在自己内部定义了一个嵌套结构体 Values。这个结构体包含两个变量，只有 Ace 有两个数值，其余牌都只有一个数值。结构体 Values 中定义的两个属性:

first, 为 Int second, 为 Int?, 或 “optional Int”

Rank 定义了一个计算属性 values，这个计算属性会根据牌的面值，用适当的数值去初始化 Values 实例，并赋值给 values。对于 J,Q,K,Ace 会使用特殊数值，对于数字面值的牌使用 Int 类型的值。

BlackjackCard 结构体自身有两个属性 —rank 与 suit，也同样定义了一个计算属性 description，description 属性用 rank 和 suit 的中内容来构建对这张扑克牌名字和数值的描述，并用可选类型 second 来检查是否存在第二个值，若存在，则在原有的描述中增加对第二数值的描述。

因为 BlackjackCard 是一个没有自定义构造函数的结构体，在 Memberwise Initializers for Structure Types 中知道结构体有默认的成员构造函数，所以你可以用默认的 initializer 去初始化新的常量 theAceOfSpades:

```
let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
println("theAceOfSpades: \(theAceOfSpades.description)")
// □ □ □ "theAceOfSpades: suit is  , value is 1 or 11"
```

尽管 Rank 和 Suit 嵌套在 BlackjackCard 中，但仍可被引用，所以在初始化实例时能够通过枚举类型中的成员名称单独引用。在上面的例子中 description 属性能正确得输出对 Ace 牌有 1 和 11 两个值。

13.5.1 类型嵌套的引用

在外部对嵌套类型的引用，以被嵌套类型的名字为前缀，加上所要引用的属性名:

```
let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
// □ □ □ □ □ □ □ "□"
```

对于上面这个例子，这样可以使 Suit, Rank, 和 Values 的名字尽可能的短，因为它们的名字会自然的由被定义的上下文来限定。

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（二十）扩展

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

Swift 中的扩展可以：

- 添加计算型属性和计算静态属性
- 定义实例方法和类型方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 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。

13.6 扩展语法（Extension Syntax）

声明一个扩展使用关键字 extension：

```
extension SomeType {
    // □ □ SomeType □ □ □ □ □ □ □ □
}
```

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

```
extension SomeType: SomeProtocol, AnotherProctocol {
    // □ □ □ □ □ □ □ □
}
```

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

13.7 计算型属性（Computed Properties）

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

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m : Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
println("One inch is \(oneInch) meters")
// □ □ □ □ : "One inch is 0.0254 meters"
let threeFeet = 3.ft
println("Three feet is \(threeFeet) meters")
// □ □ □ □ : "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：

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// □ □ □ □ : "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)。

13.8 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：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

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：

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
    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：

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

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

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect □ □ □ □ (2.5, 2.5) , □ □ □ (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。

13.8.1 方法（Methods）

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

```
extension Int {
    func repetitions(task: () -> ()) {
        for i in 0..self {
            task()
```

```
        }
    }
}
```

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

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

```
3.repetitions({
    println("Hello!")
    })
// Hello!
// Hello!
// Hello!
```

可以使用 trailing 闭包使调用更加简洁：

```
3.repetitions{
    println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

13.8.2 修改实例方法（Mutating Instance Methods）

通过扩展添加的实例方法也可以修改该实例本身。结构体和枚举类型中修改 self 或其属性的方法必须将该实例方法标注为 mutating，正如来自原始实现的修改方法一样。

下面的例子向 Swift 的 Int 类型添加了一个新的名为 square 的修改方法，来实现一个原始值的平方计算：

```
extension Int {
    mutating func square() {
        self = self * self
    }
```

```
}
var someInt = 3
someInt.square()
// someInt □ □ □ □ 9
```

13.8.3 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

- 123456789[0] 返回 9
- 1234567891返回 8

...等等

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
            for _ in 1...digitIndex {
                decimalBase *= 10
            }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

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

```
746381295[9]
//returns 0, □ □ □ □ :
0746381295[9]
```

13.8.4 嵌套类型（Nested Types）

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

```
extension Character {
    enum Kind {
        case Vowel, Consonant, Other
    }
    var kind: Kind {
        switch String(self).lowercaseString {
        case "a", "e", "i", "o", "u":
            return .Vowel
        case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
             "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
            return .Consonant
        default:
            return .Other
        }
    }
}
```

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

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：

```
func printLetterKinds(word: String) {
   println("'\\(word)' is made up of the following kinds of letters:")
    for character in word {
        switch character.kind {
        case .Vowel:
            print("vowel ")
        case .Consonant:
            print("consonant ")
        case .Other:
```

```
            print("other ")
        }
    }
    print("\n")
}
printLetterKinds("Hello")
// 'Hello' is made up of the following kinds of letters:
// 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

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（二十一）协议

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

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。

13.9 协议的语法

协议的定义与类，结构体，枚举的定义非常相似，如下所示：

```
protocol SomeProtocol {
    // □ □ □ □
}
```

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

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // □ □ □ □ □
}
```

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

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // □ □ □ □
}
```

13.10 属性要求

协议能够要求其遵循者必须含有一些特定名称和类型的实例属性 (instance property) 或类属性 (type property)，也能够要求属性的 (设置权限)settable 和 (访问权限)gettable，但它不要求属性是存储型属性 (stored property) 还是计算型属性 (calculate property)。

通常前置 var 关键字将属性声明为变量。在属性声明后写上 { get set } 表示属性为可读写的。{ get } 用来表示属性为可读的。即使你为可读的属性实现了 setter 方法，它也不会出错。

```
protocol SomeProtocol {
    var musBeSettable : Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

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

```
protocol AnotherProtocol {
    class var someTypeProperty: Int { get set }
}

protocol FullyNamed {
    var fullName: String { get }
}
```

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

```
struct Person: FullyNamed{
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
//john.fullName  "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，Startship 类遵循了 FullyNamed 协议：

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil ) {
        self.anme = name
        self.prefix = prefix
    }
    var fullName: String {
    return (prefix ? prefix ! + " " : " ") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// 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。

13.11 方法要求

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

> 注意：协议方法支持变长参数 (variadic parameter)，不支持默认参数 (default parameter)。

前置 class 关键字表示协议中的成员为类成员；当协议用于被枚举或结构体遵循时，则使用 static 关键字。如下所示：

```
protocol SomeProtocol {
    class func someTypeMethod()
}

protocol RandomNumberGenerator {
    func random() -> Double
}
```

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。

```
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c) % m)
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
println("Here's a random number: \(generator.random())")
// □□ : "Here's a random number: 0.37464991998171"
println("And another one: \(generator.random())")
// □□ : "And another one: 0.729023776863283"
```

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

```
protocol Togglable {
    mutating func toggle()
}
```

当使用枚举或结构体来实现 Togglabl 协议时，必须在 toggle 方法前加上 mutating 关键字。

如下所示，OnOffSwitch 枚举遵循了 Togglable 协议，On，Off 两个成员用于表示当前状态

```
enum OnOffSwitch: Togglable {
    case Off, On
    mutating func toggle() {
        switch self {
        case Off:
            self = On
        case On:
            self = Off
        }
    }
}
var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
//lightSwitch □ □ □ □ □ .On
```

13.13 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：

- 作为函数，方法或构造器中的参数类型，返回值类型
- 作为常量，变量，属性的类型
- 作为数组，字典或其他容器中的元素类型

注意：协议类型应与其他类型 (Int，Double，String) 的写法相同，使用驼峰式

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) +1
    }
}
```

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

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

```
for _ in 1...5 {
    println("Random dice roll is \(d6.roll())")
}
//□ □ □ □
//Random dice roll is 3
//Random dice roll is 5
//Random dice roll is 4
//Random dice roll is 5
//Random dice roll is 4
```

13.14 委托 (代理) 模式

委托是一种设计模式，它允许类或结构体将一些需要它们负责的功能交由 (委托) 给其他的类型。

委托模式的实现很简单：定义协议来封装那些需要被委托的函数和方法，使其遵循者拥有这些被委托的函数和方法。

委托模式可以用来响应特定的动作或接收外部数据源提供的数据，而无需要知道外部数据源的类型。

下文是两个基于骰子游戏的协议：

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate {
    func gameDidStart(game: DiceGame)
   func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:Int)
    func gameDidEnd(game: DiceGame)
}
```

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

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

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
   let dic = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: Int[]
    init() {
        board = Int[](count: finalSquare + 1, repeatedValue: 0)
     board[03] = +08; board[06] = +11; borad[09] = +09; board[10] = +02
     borad[14] = -10; board[19] = -11; borad[22] = -02; board[24] = -08
    }
     var delegate: DiceGameDelegate?
     func play() {
         square = 0
         delegate?.gameDidStart(self)
         gameLoop: while square != finalSquare {
             let diceRoll = dice.roll()
          delegate?.game(self,didStartNewTurnWithDiceRoll: diceRoll)
             switch square + diceRoll {
             case finalSquare:
                 break gameLoop
             case let newSquare where newSquare > finalSquare:
                 continue gameLoop
             default:
             square += diceRoll
             square += board[square]
             }
         }
         delegate?.gameDIdEnd(self)
     }
}
```

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

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            println("Started a new game of Snakes and Ladders")
        }
        println("The game is using a \(game.dice.sides)-sided dice")
    }
   func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        ++numberOfTurns
        println("Rolled a \(diceRoll)")
    }
    func gameDidEnd(game: DiceGame) {
        println("The game lasted for \(numberOfTurns) turns")
    }
}
```

DiceGameTracker 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；在每新一轮中递加；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:

```
“ let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
```

```
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns ”
```

13.14.1 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。详情请在扩展章节中查看。

> 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：

```
protocol TextRepresentable {
    func asText() -> String
}
```

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

```
extension Dice: TextRepresentable {
    cun asText() -> String {
        return "A \(sides)-sided dice"
    }
}
```

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

```
let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())
println(d12.asText())
//  □ □  "A 12-sided dice"
SnakesAndLadders □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ :
```

```
extension SnakeAndLadders: TextRepresentable {
    func asText() -> String {
     return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
println(game.asText())
// □ □ "A game of Snakes and Ladders with 25 squares"
```

13.14.2 通过延展补充协议声明

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

```
struct Hamster {
    var name: String
    func asText() -> String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentabl {}
```

从现在起，Hamster 的实例可以作为 TextRepresentable 类型使用

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentabl = simonTheHamester
println(somethingTextRepresentable.asText())
// □ □ "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

13.14.3 集合中的协议类型

协议类型可以被集合使用，表示集合中的元素均为协议类型:

```
let things: TextRepresentable[] = [game,d12,simoTheHamster]
```

如下所示，things 数组可以被直接遍历，并调用其中元素的 asText() 函数：

```
for thing in things {
    println(thing.asText())
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

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

13.15 协议的继承

协议能够继承一到多个其他协议。语法与类的继承相似，多个协议间用逗号，分隔

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // □ □ □ □
}
```

如下所示，PrettyTextRepresentable 协议继承了 TextRepresentable 协议

```
protocol PrettyTextRepresentable: TextRepresentable {
    func asPrettyText() -> String
}
```

遵循 ‘PrettyTextRepresentable □□□□□,□□□□□ TextRepresentable 协议。

如下所示，用扩展为 SnakesAndLadders 遵循 PrettyTextRepresentable 协议：

```
extension SnakesAndLadders: PrettyTextRepresentable {
    func asPrettyText() -> String {
        var output = asText() + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
                case let ladder where ladder > 0:
                output += "  "
```

```
            case let snake where snake < 0:
                output += "  "
            default:
                output += "  "
            }
        }
        return output
    }
}
```

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

```
println(game.asPrettyText())
// A game of Snakes and Ladders with 25 squares:
//
```

13.15.1 协议合成

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

举个例子：

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
```

```
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
   println("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(birthdayPerson)
// □ □ "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。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。

13.15.2 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)：

is 操作符用来检查实例是否遵循了某个协议。

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

as 用以强制向下转换型。

```
@objc protocol HasArea {
    var area: Double { get }
}
```

> 注意：

@objc 用来表示协议是可选的，也可以用来表示暴露给 Objective-C 的代码，此外，@objc 型协议只对类有效，因此只能在类中检查协议的一致性。详情查看 Using Siwft with Cocoa and Objectivei-c。

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
```

```
    var area: radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

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

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

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

```
let objects: AnyObject[] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

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

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        println("Area is \(objectWithArea.area)")
    } else {
        println("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

当数组中的元素遵循 HasArea 协议时，通过 as? 操作符将其可选绑定 (optional binding) 到 objectWithArea 常量上。

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。

13.15.3 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：

```
@objc protocol CounterDataSource {
    @optional func incrementForCount(count: Int) -> Int
    @optional var fixedIncrement: Int { get }
}
```

CounterDataSource 含有 incrementForCount 的可选方法和 fiexdIncrement 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：

```
@objc class Counter {
    var count = 0
    var dataSource: CounterDataSource?
```

```
    func increment() {
        if let amount = dataSource?.incrementForCount?(count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement? {
            count += amount
        }
    }
}
```

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

increment 方法通过可选链，尝试从两种可选成员中获取 count。

由于 dataSource 可能为 nil，因此在 dataSource 后边加上了? 标记来表明只在 dataSource 非空时才去调用 incrementForCount‘方法。即使 dataSource 存在，但是也无法保证其是否实现了 incrementForCount 方法，因此在 incrementForCount 方法后边也加有? 标记。在调用 incrementForCount 方法后，Int 型可选值通过可选绑定 (optional binding) 自动拆包并赋值给常量 amount。

当 incrementForCount 不能被调用时，尝试使用可选属性 “fixedIncrement 来代替。

ThreeSource 实现了 CounterDataSource 协议，如下所示：

```
class ThreeSource: CounterDataSource {
    let fixedIncrement = 3
}
```

使用 ThreeSource 作为数据源开实例化一个 Counter：

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    println(counter.count)
}
// 3
// 6
// 9
// 12
```

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

```
class TowardsZeroSource: CounterDataSource {
func incrementForCount(count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

下边是执行的代码：

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    println(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（二十二）泛型

泛型代码可以让你写出根据自我需求定义、适用于任何类型的，灵活且可重用的函数和类型。它的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。

13.16 泛型所解决的问题

这里是一个标准的，非泛型函数 swapTwoInts, 用来交换两个 Int 值:

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

这个函数使用写入读出（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:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// □ □ "someInt is now 107, and anotherInt is now 3"
```

swapTwoInts 函数是非常有用的，但是它只能交换 Int 值，如果你想要交换两个 String 或者 Double，就不得不写更多的函数，如 swapTwoStrings 和 swapTwoDoublesfunctions，如同如下所示:

```
func swapTwoStrings(inout a: String, inout b: String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(inout a: Double, inout b: Double) {
    let temporaryA = a
```

```
    a = b
    b = temporaryA
}
```

你可能注意到 swapTwoInts、swapTwoStrings 和 swapTwoDoubles 函数功能都是相同的，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。

13.17 泛型函数

泛型函数可以工作于任何类型，这里是一个上面 swapTwoInts 函数的泛型版本，用于交换两个值：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

swapTwoValues 函数主体和 swapTwoInts 函数是一样的，它只在第一行稍微有那么一点点不同于 swapTwoInts，如下所示：

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

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 In、String 或 Doubl)。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:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

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。

13.18 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。

13.18.1 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。

13.18.2 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）。

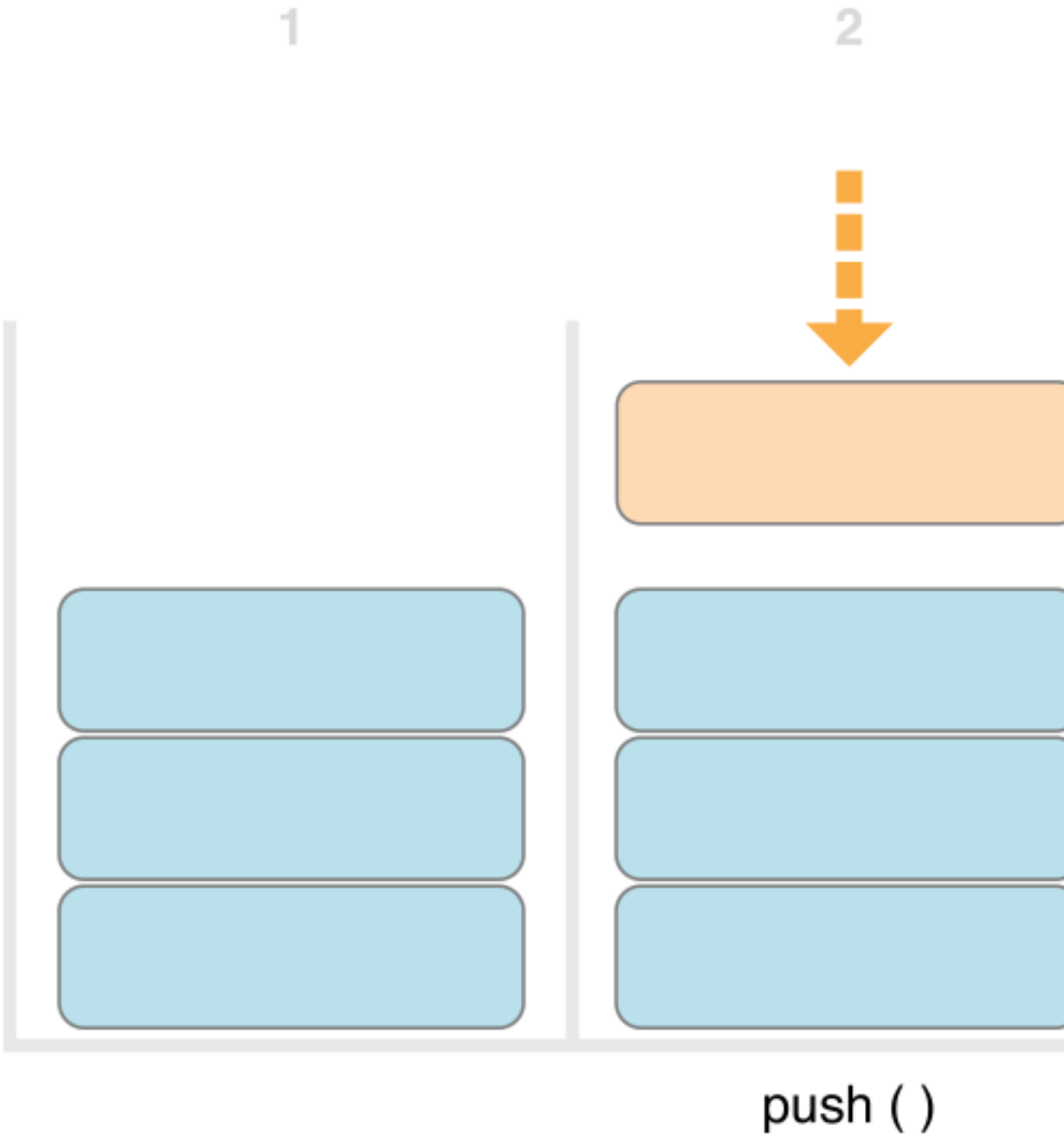
注意栈的概念已被 UINavigationController 类使用来模拟试图控制器的导航结构。你通过调用 UINavigationController 的 pushViewController:animated: 方法来为导航栈添加（add）新的试图控制器；而通过 popViewControllerAnimated: 的方法来从导航栈中移除（pop）某个试图控制器。每当你需要一个严格的后进先出方式来管理集合，堆栈都是最实用的模型。

下图展示了一个栈的压栈 (push)/出栈 (pop) 的行为：

现在有三个值在栈中；第四个值 “pushed” 到栈的顶部；现在有四个值在栈中，最近的那个在顶部；栈中最顶部的那个项被移除，或称之为 “popped”；移除掉一个值后，现在栈又重新只有三个值。

这里展示了如何写一个非泛型版本的栈，Int 值型的栈：

```
struct IntStack {
    var items = Int[]()
```


1
2
push ( )

```
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

这个结构体在栈中使用一个 Array 性质的 items 存储值。Stack 提供两个方法：push 和 pop，从栈中压进一个值和移除一个值。这些方法标记为可变的，因为他们需要修改（或转换）结构体的 items 数组。

上面所展现的 IntStack 类型只能用于 Int 值，不过，其对于定义一个泛型 Stack 类（可以处理任何类型值的栈）是非常有用的。

这里是一个相同代码的泛型版本：

```
struct Stack<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

注意到 Stack 的泛型版本基本上和非泛型版本相同，但是泛型版本的占位类型参数为 T 代替了实际 Int 类型。这种类型参数包含在一对尖括号里 ()，紧随在结构体名字后面。

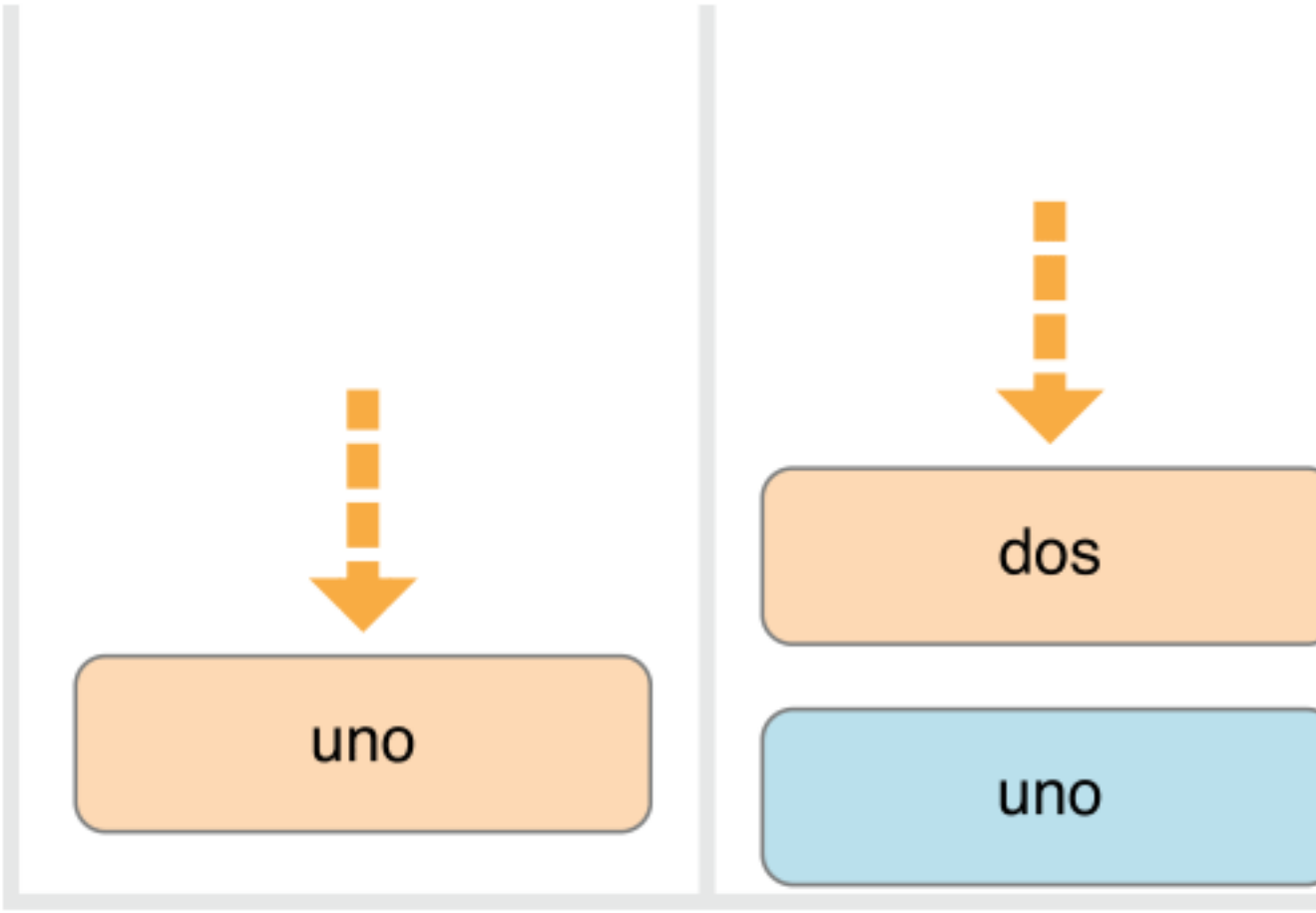
T 定义了一个名为 “某种类型 T” 的节点提供给后来用。这种将来类型可以在结构体的定义里任何地方表示为 “T”。在这种情况下，T 在如下三个地方被用作节点：

- 创建一个名为 items 的属性，使用空的 T 类型值数组对其进行初始化；
- 指定一个包含一个参数名为 item 的 push 方法，该参数必须是 T 类型；
- 指定一个 pop 方法的返回值，该返回值将是一个 T 类型值。

当创建一个新单例并初始化时，通过用一对紧随在类型名后的尖括号里写出实际指定栈用到类型，创建一个 Stack 实例，同创建 Array 和 Dictionary 一样：

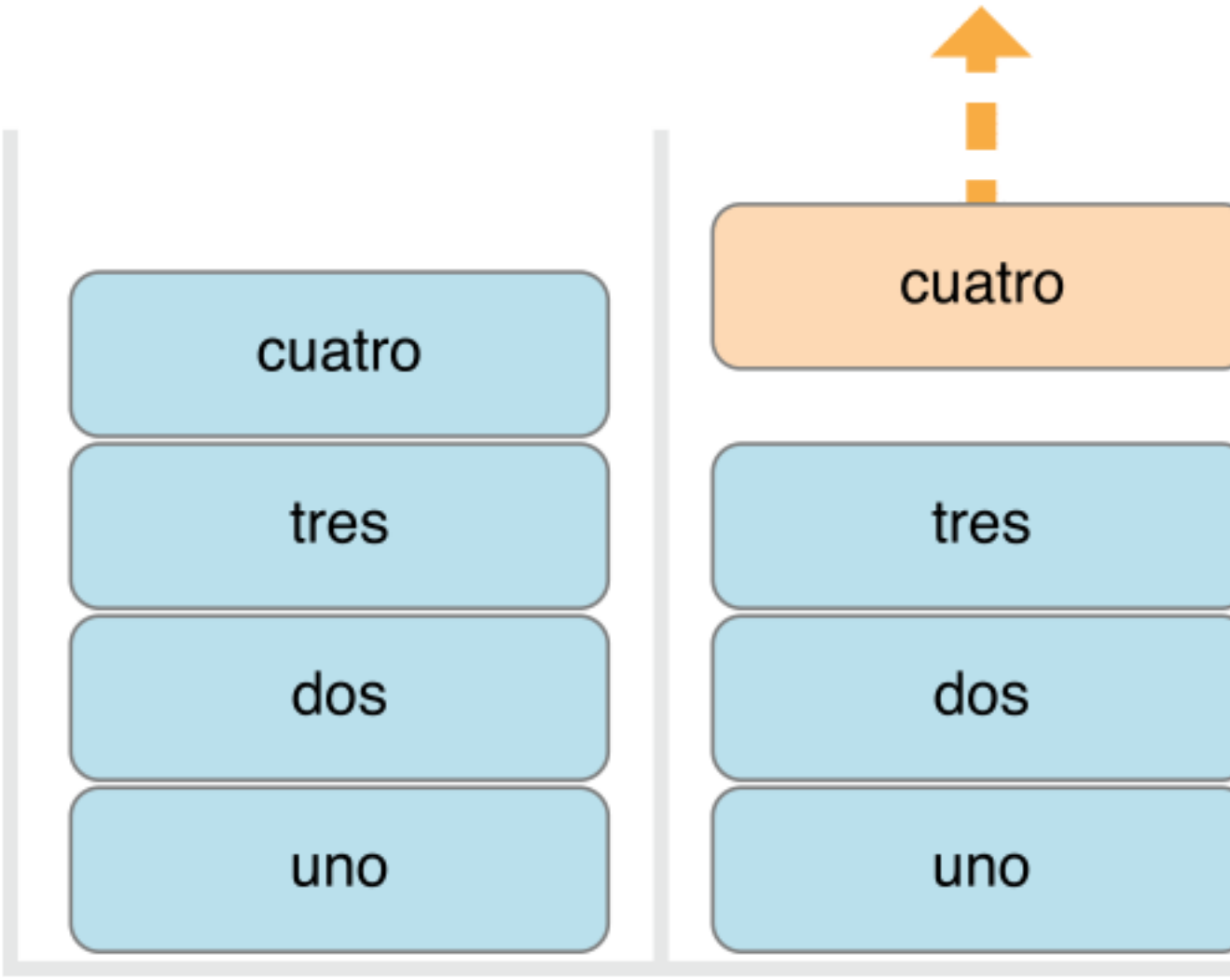
```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// □ □ □ □ □ □4□string□
```

下图将展示 stackOfStrings 如何 push 这四个值进栈的过程：



从栈中 pop 并移除值”cuatro”：

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```


cuatro
tres
dos
uno
cuatro
tres
dos
uno

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。

13.19 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。

13.19.1 类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）:

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

上面这个假定函数有两个类型参数。第一个类型参数 T，有一个需要 T 必须是 SomeClass 子类的类型约束；第二个类型参数 U，有一个需要 U 必须遵循 SomeProtocol 协议的类型约束。

13.19.2 类型约束行为

这里有个名为 findStringIndex 的非泛型函数，该函数功能是去查找包含一给定 String 值的数组。若查找到匹配的字符串，findStringIndex 函数返回该字符串在数组中的索引值（Int），反之则返回 nil:

```
func findStringIndex(array: String[], valueToFind: String) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

findStringIndex 函数可以作用于查找一字符串数组中的某个字符串:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    println("The index of llama is \(foundIndex)")
}
// □ □ "The index of llama is 2"
```

如果只是针对字符串而言查找在数组中的某个值的索引，用处不是很大，不过，你可以写出相同功能的泛型函数 findIndex，用某个类型 T 值替换掉提到的字符串。

这里展示如何写一个你或许期望的 findStringIndex 的泛型版本 findIndex。请注意这个函数仍然返回 Int，是不是有点迷惑呢，而不是泛型类型? 那是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数不会编译，原因在例子后面会说明:

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所写的函数不会编译。这个问题的位置在等式的检查上，“if value == valueToFind”。不是所有的 Swift 中的类型都可以用等式符（==）进行比较。例如，如果你创建一个你自己的类或结构体来表示一个复杂的数据模型，那么 Swift 没法猜到对于这个类或结构体而言 “等于” 的意思。正因如此，这部分代码不能可能保证工作于每个可能的类型 T，当你试图编译这部分代码时估计会出现相应的错误。

不过，所有的这些并不会让我们无从下手。Swift 标准库中定义了一个 Equatable 协议，该协议要求任何遵循的类型实现等式符（==）和不等符（!=）对任何两个该类型进行比较。所有的 Swift 标准类型自动支持 Equatable 协议。

任何 Equatable 类型都可以安全的使用在 findIndex 函数中，因为其保证支持等式操作。为了说明这个事实，当你定义一个函数时，你可以写一个 Equatable 类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

findIndex 中这个单个类型参数写做：T: Equatable，也就意味着 “任何 T 类型都遵循 Equatable 协议”。

findIndex 函数现在则可以成功的编译过，并且作用于任何遵循 Equatable 的类型，如 Double 或 String:

```
let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
// doubleIndex is an optional Int with no value, because 9.3 is not in the array
let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
// stringIndex is an optional Int containing a value of 2
```

13.20 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。

13.20.1 关联类型行为

这里是一个 Container 协议的例子，定义了一个 ItemType 关联类型：

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

Container 协议定义了三个任何容器必须支持的兼容要求：

- 必须可能通过 append 方法添加一个新 item 到容器里；
- 必须可能通过使用 count 属性获取容器里 items 的数量，并返回一个 Int 值；
- 必须可能通过容器的 Int 索引值下标可以检索到每一个 item。

这个协议没有指定容器里 item 是如何存储的或何种类型是允许的。这个协议只指定三个任何遵循 Container 类型所必须支持的功能点。一个遵循的类型也可以提供其他额外的功能，只要满足这三个条件。

任何遵循 Container 协议的类型必须指定存储在其里面的值类型，必须保证只有正确类型的 items 可以加进容器里，必须明确可以通过其下标返回 item 类型。

为了定义这三个条件，Container 协议需要一个方法指定容器里的元素将会保留，而不需要知道特定容器的类型。Container 协议需要指定任何通过 append 方法添加到容器里的值和容器里元素是相同类型，并且通过容器下标返回的容器元素类型的值的类型是相同类型。

为了达到此目的，Container 协议声明了一个 ItemType 的关联类型，写作 typealias ItemType。The protocol does not define what ItemType is an alias for—that information is left for any conforming type to provide（这个协议不会定义 ItemType 是遵循类型所提供的何种信息的别名）。尽管如此，ItemType 别名支持一种方法识别在一个容器里的 items 类型，以及定义一种使用在 append 方法和下标中的类型，以便保证任何期望的 Container 的行为是强制性的。

这里是一个早前 IntStack 类型的非泛型版本，适用于遵循 Container 协议：

```
struct IntStack: Container {
    // original IntStack implementation
```

```
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias ItemType = Int
    mutating func append(item: Int) {
        self.push(item)
    }
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

IntStack 类型实现了 Container 协议的所有三个要求，在 IntStack 类型的每个包含部分的功能都满足这些要求。

此外，IntStack 指定了 Container 的实现，适用的 ItemType 被用作 Int 类型。对于这个 Container 协议实现而言，定义 typealias ItemType = Int，将抽象的 ItemType 类型转换为具体的 Int 类型。

感谢 Swift 类型参考，你不用在 IntStack 定义部分声明一个具体的 Int 的 ItemType。由于 IntStack 遵循 Container 协议的所有要求，只要通过简单的查找 append 方法的 item 参数类型和下标返回的类型，Swift 就可以推断出合适的 ItemType 来使用。确实，如果上面的代码中你删除了 typealias ItemType = Int 这一行，一切仍旧可以工作，因为它清楚的知道 ItemType 使用的是何种类型。

你也可以生成遵循 Container 协议的泛型 Stack 类型：

```
struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
```

```
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> T {
        return items[i]
    }
}
```

这个时候，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。

13.20.2 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：

```
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。

13.20.3 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：

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {

       // check that both containers contain the same number of items
        if someContainer.count != anotherContainer.count {
            return false
        }

        // check each pair of items to see if they are equivalent
        for i in 0..someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // all items match, so return true
        return true

}
```

这个函数用了两个参数：someContainer 和 anotherContainer。someContainer 参数是类型 C1，anotherContainer 参数是类型 C2。C1 和 C2 是容器的两个占

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：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
```

```
    println("All items match.")
} else {
    println("Not all items match.")
}
// □ □ "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。

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。# Swift 中文教程（二十三）高级运算符

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。

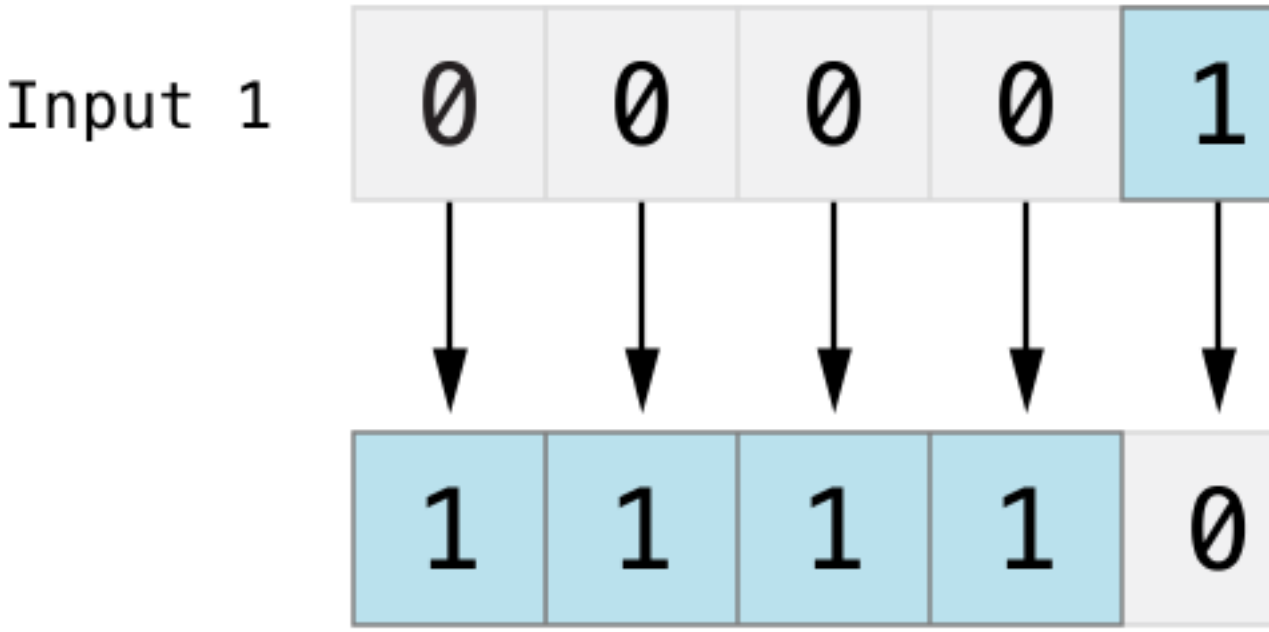
13.21 位运算符

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：

13.21.1 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，所以请不加任何空格地写着操作数之前。

```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // □ □ 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。

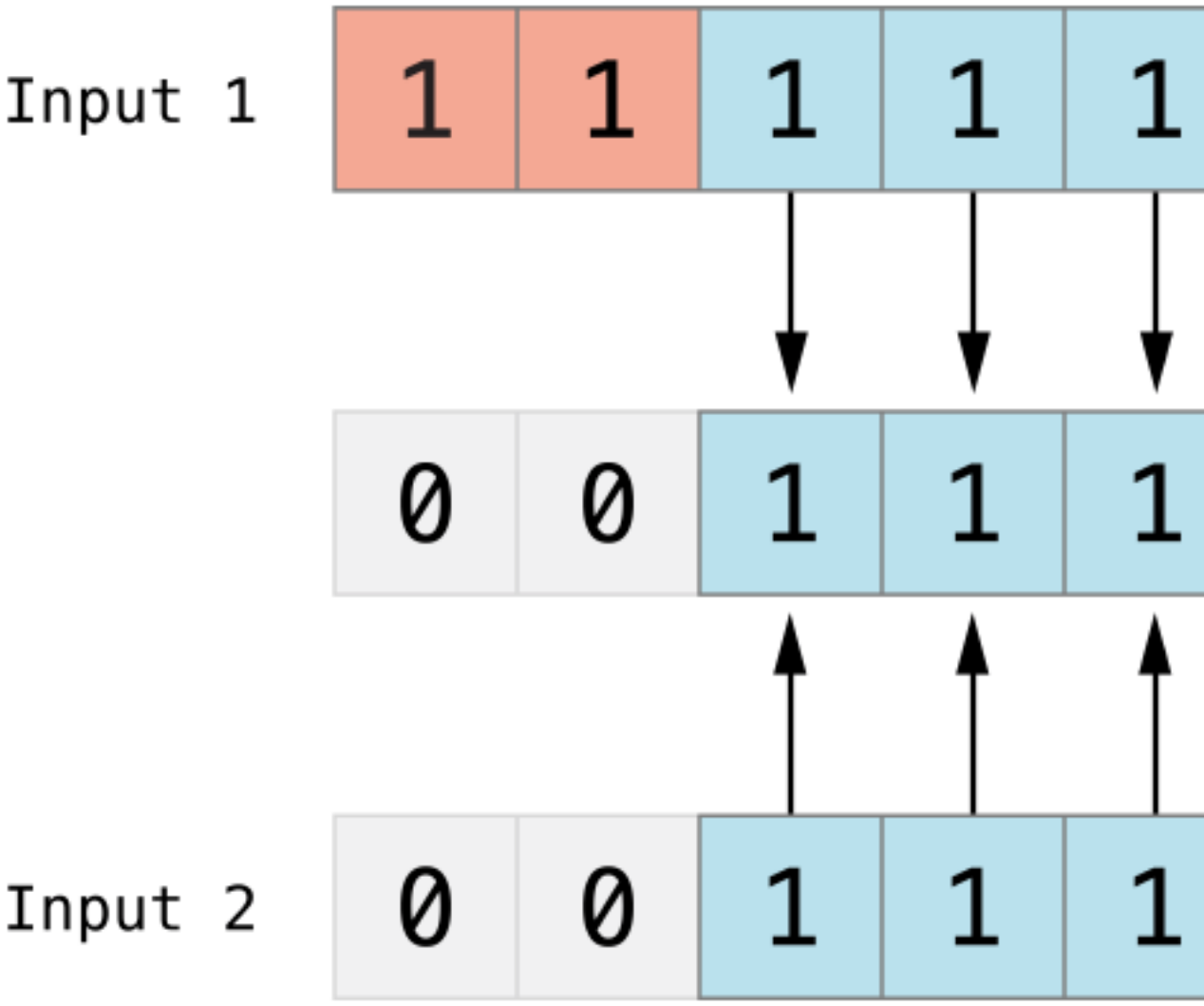
使用按位取反运算 ~ 对 initialBits 操作，然后赋值给 invertedBits 这个新常量。这个新常量的值等于所有位都取反的 initialBits，即 1 变成 0，0 变成 1，变成了 11110000，十进制值为 240。

13.21.2 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。

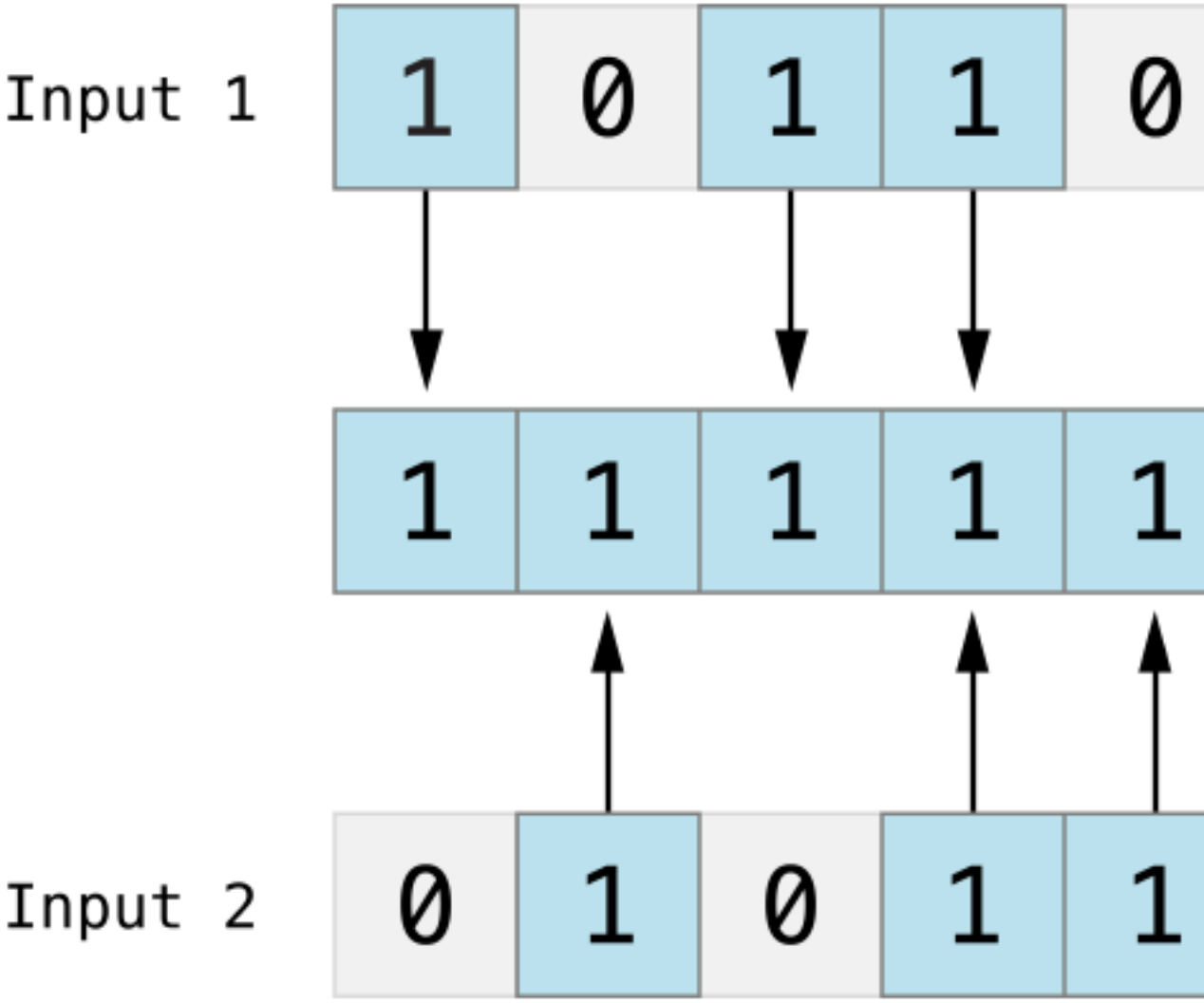
以下代码，firstSixBits 和 lastSixBits 中间 4 个位都为 1。对它俩进行按位与运算后，就得到了 00111100，即十进制的 60。

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // □ □ 00111100
```


Input 1
1 1 1 1 1
0 0 1 1 1
Input 2
0 0 1 1 1

13.21.3 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)。

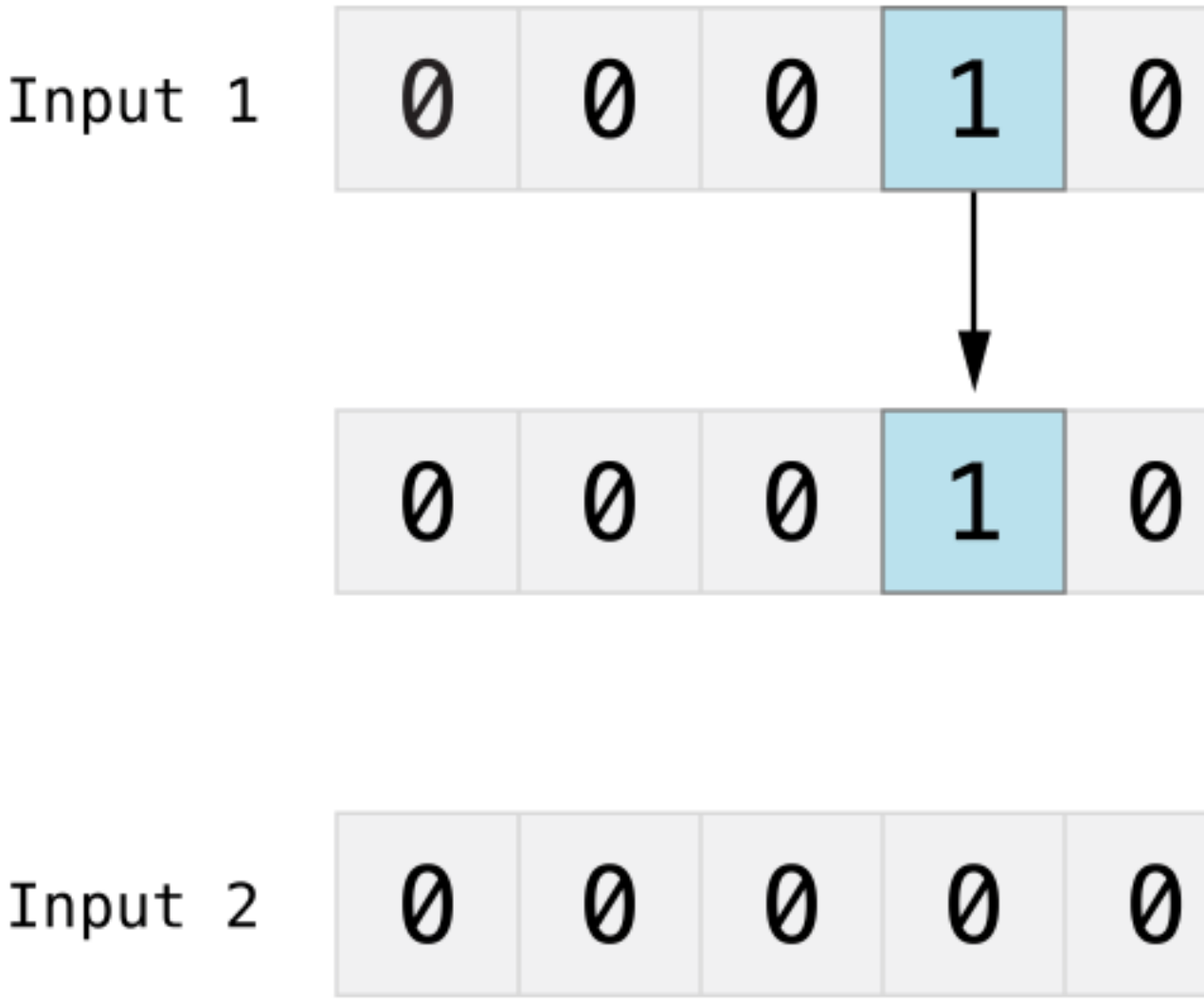


如下代码，someBits 和 moreBits 在不同位上有 1。按位或运行的结果是 11111110，即十进制的 254。

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // □ □ 11111110
```

13.21.4 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。


Input 1
0 0 0 1 0
0 0 0 1 0
Input 2
0 0 0 0 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。

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // □ □ 00010001
```

13.21.5 按位左移/右移运算符

左移运算符 <> 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。

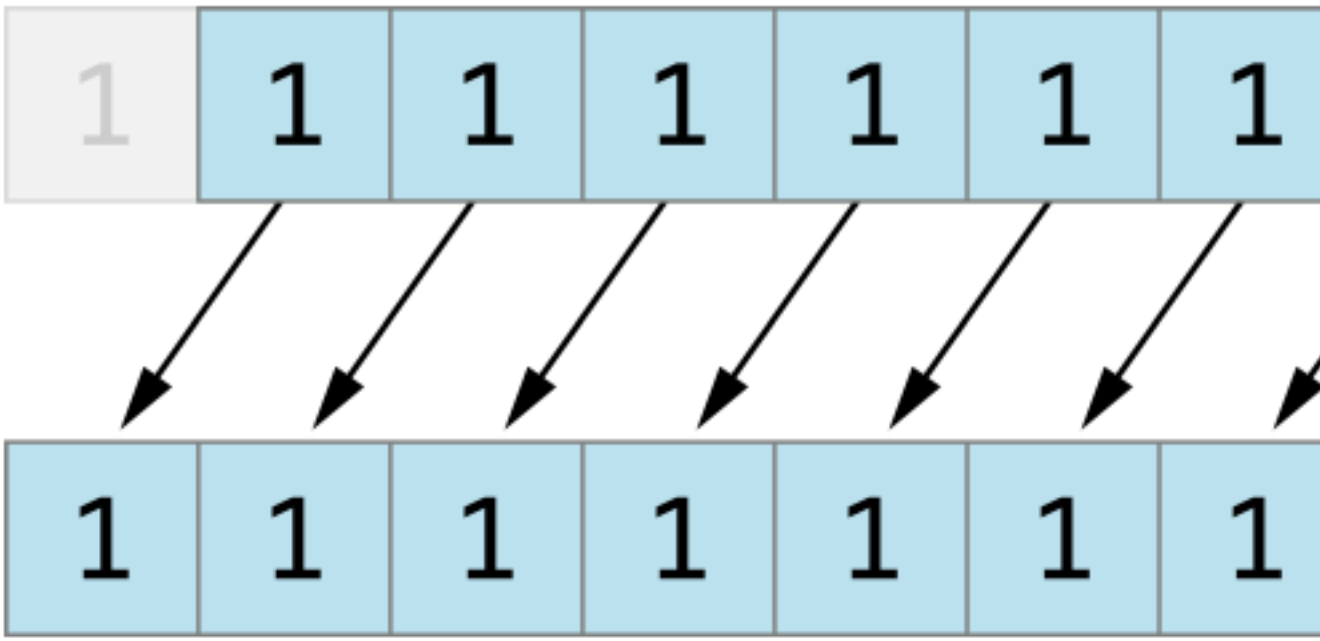
按位左移和按位右移的效果相当把一个整数乘于或除于一个因子为 2 的整数。向左移动一个整型的比特位相当于把这个数乘于 2，向右移一位就是除于 2。

13.21.6 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 << 1(11111111 向左移 1 位)，和 11111111 >> 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。



```
let shiftBits: UInt8 = 4   // □ □ □ □ □00000100
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
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。

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16   // redComponent □ 0xCC, □ 204
let greenComponent = (pink & 0x00FF00) >> 8  // greenComponent □ 0x66, □ 102
let blueComponent = pink & 0x0000FF         // blueComponent □ 0x99, □ 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 OxCC6699 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。

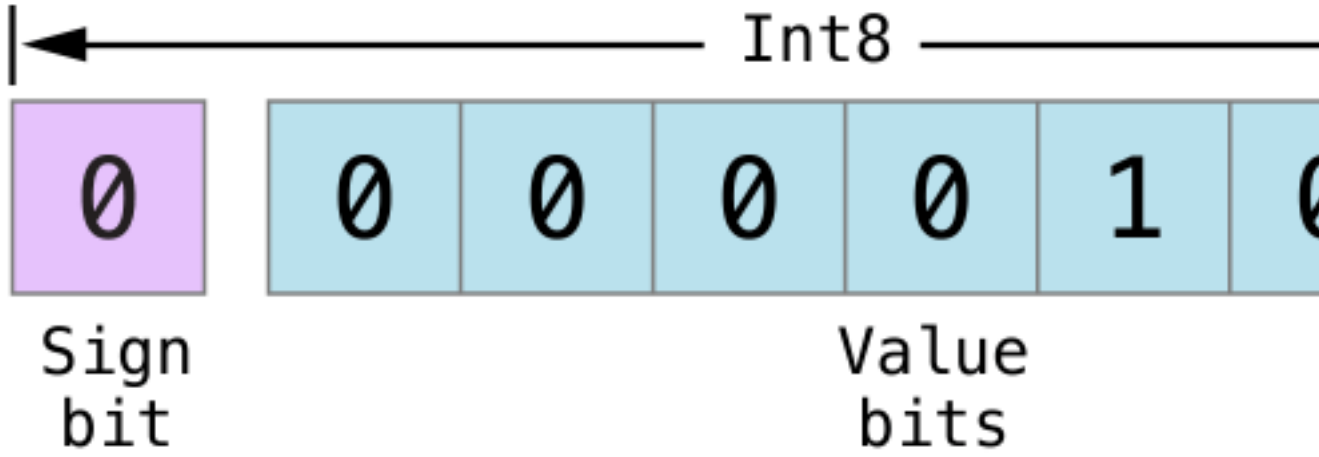
同样的，绿色部分来自于 0xCC6699 和 0x00FF00 的按位操作得到 0x006600。然后向右移动 8 們，得到 0x66，即十进制的 102。

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。

13.21.7 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。

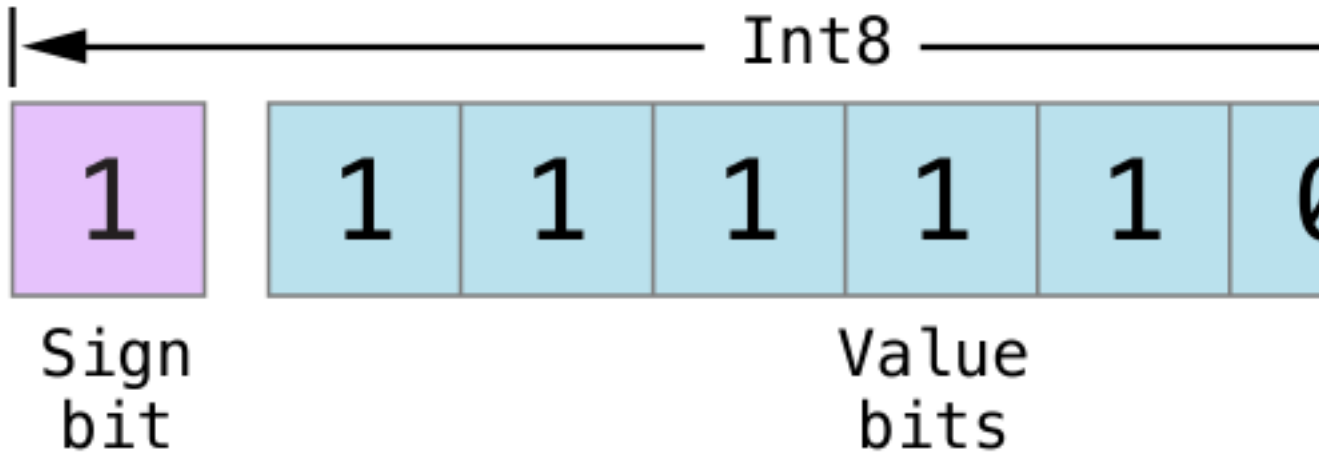


其余的比特位 (称为数值位) 存储其实值。有符正整数和无符正整数在计算机里的存储结果是一样的，下来我们来看 +4 内部的二进制结构。

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。

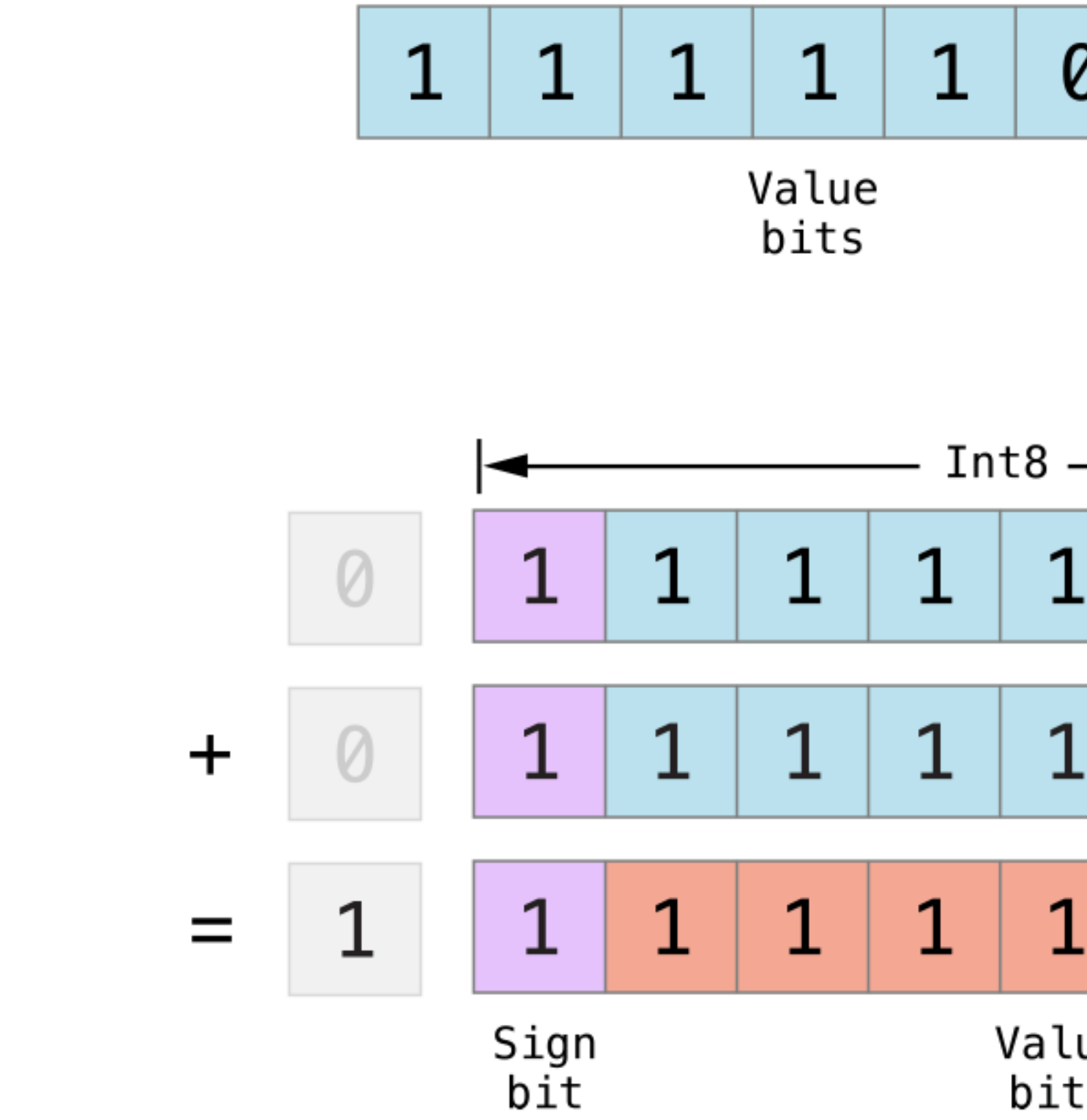
我们来看 -4 存储的二进制结构。



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。

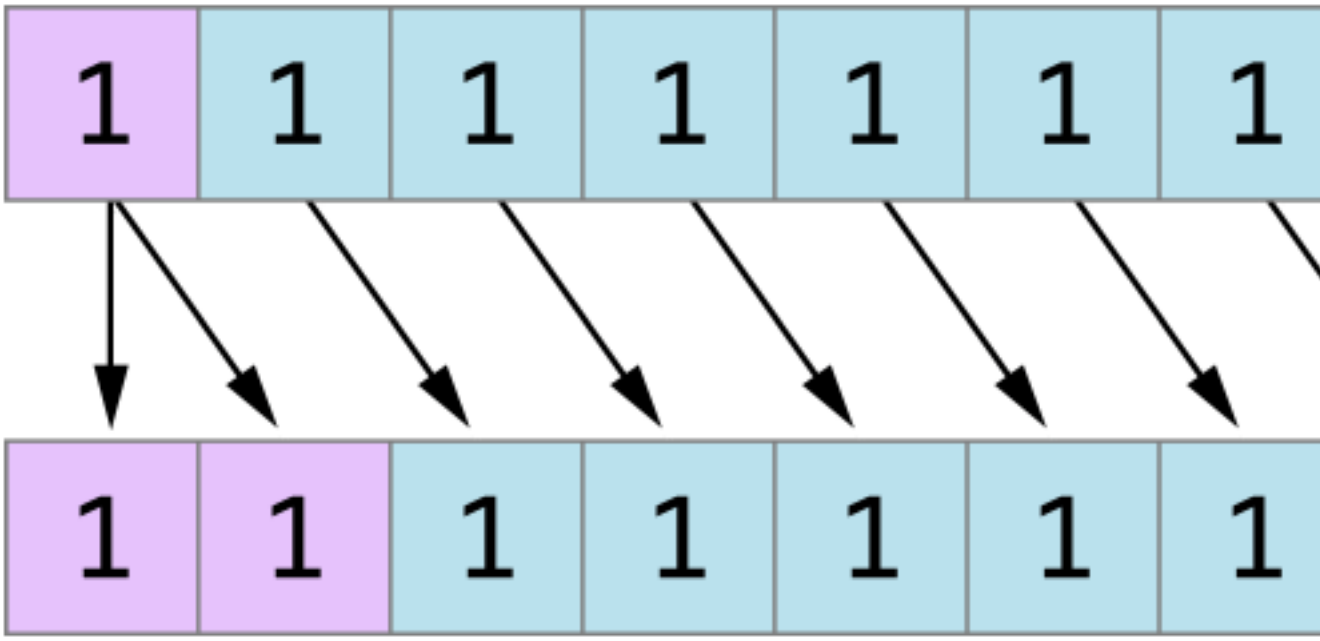
负数的编码方式称为二进制补码表示。这种表示方式看起来很奇怪，但它有几个优点。

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。


1 1 1 1 1 0
Value bits
Int8
0
1 1 1 1 1
+
0
1 1 1 1 1
=
1
1 1 1 1 1
Sign bit
Valu bit

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 位时乘于 2，右移 1 位时除于 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。

13.21.8 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：

```
var potentialOverflow = Int16.max
// potentialOverflow □ □ 32767, □ □ Int16 □ □ □ □ □ □ □ □
potentialOverflow += 1
// □, □ □ □
```

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。Swfit 为整型计算提供了 5 个 & 符号开头的溢出运算符。

- 溢出加法 &+
- 溢出减法 &-
- 溢出乘法 &*
- 溢出除法 &/
- 溢出求余 &%

13.21.9 值的上溢出

下面例子使用了溢出加法 &+ 来解剖的无符整数的上溢出

```
var willOverflow = UInt8.max
// willOverflow □ □UInt8□ □ □ □ □ 255
willOverflow = willOverflow &+ 1
// □ □ □ willOverflow □ □ 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。

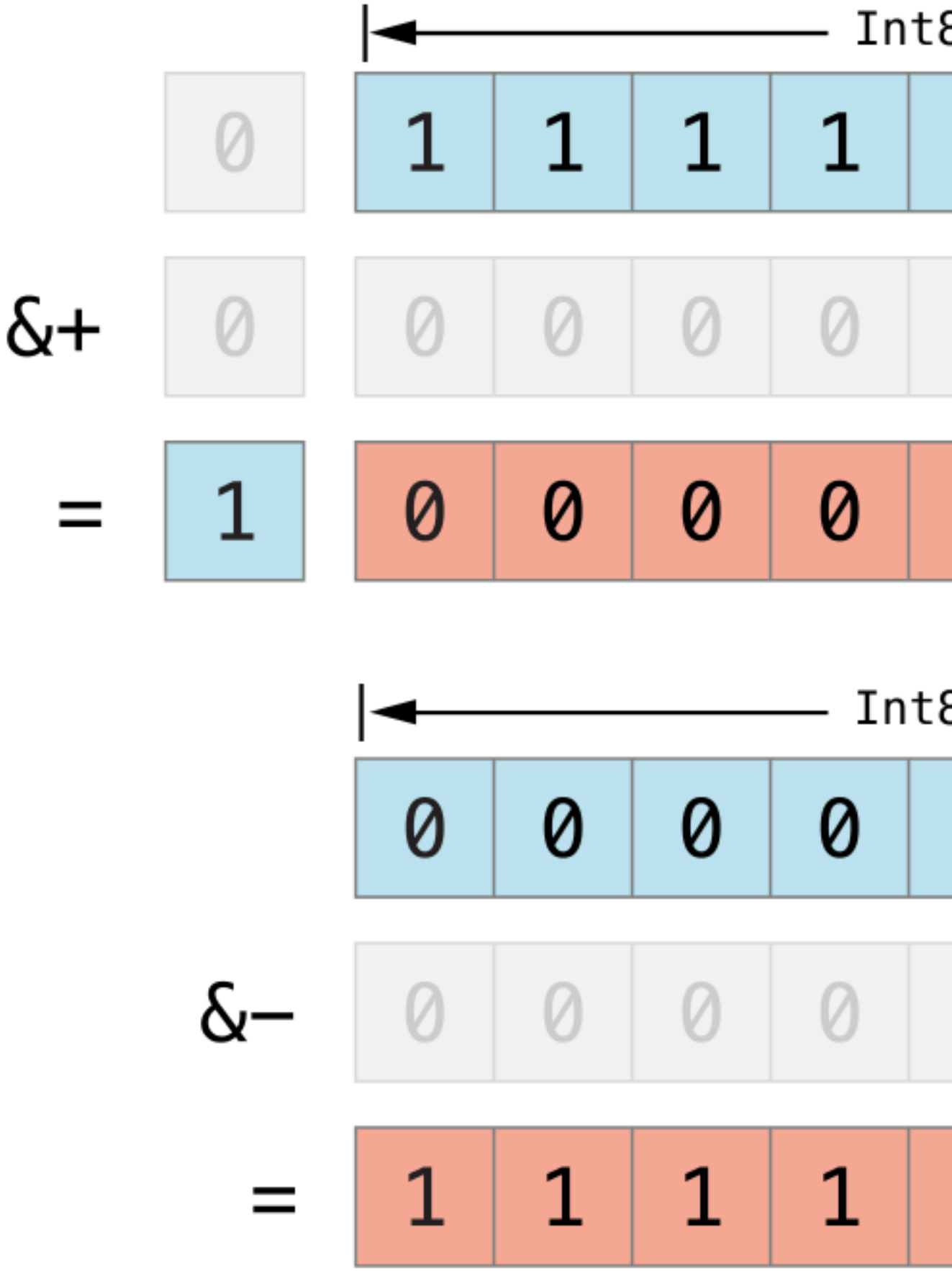
13.21.10 值的下溢出

数值也有可能因为太小而越界。举个例子:

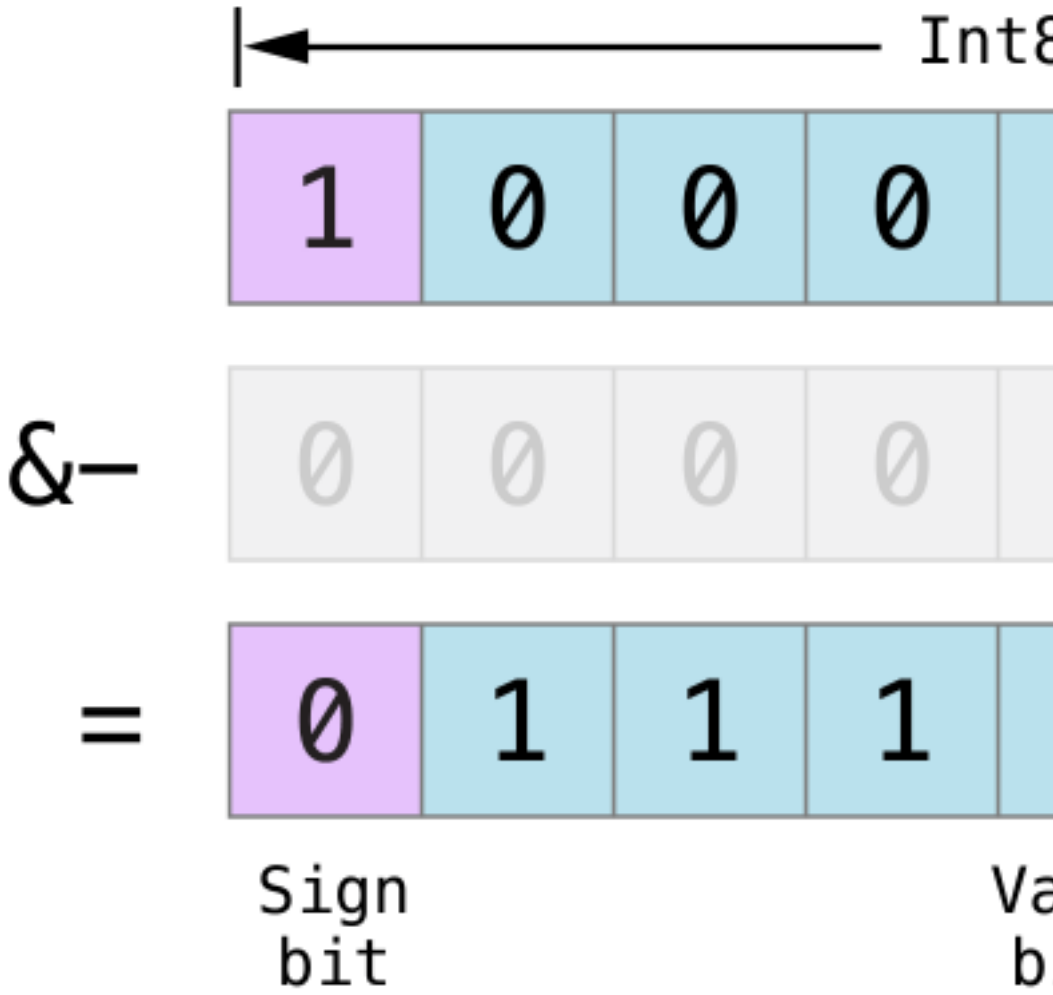
UInt8 的最小值是 0(二进制为 00000000)。使用 &-进行溢出减 1，就会得到二进制的 11111111 即十进制的 255。

Swift 代码是这样的:

```
var willUnderflow = UInt8.min
// willUnderflow □ □UInt8□ □ □ □ 0
willUnderflow = willUnderflow &- 1
// □ □ willUnderflow □ □ 255
```


Int8
0
1 1 1 1
&+
0
0 0 0 0
=
1
0 0 0 0
Int8
0 0 0 0
&-
0 0 0 0
=
1 1 1 1

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



来看看 Swift 代码：

```
var signedUnderflow = Int8.min
// signedUnderflow □ □ □ □ □ □ □ □ -128
signedUnderflow = signedUnderflow &- 1
// □ □ signedUnderflow □ □ 127
```

13.21.11 除零溢出

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

```
let x = 1
let y = x / 0
```

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

```
let x = 1
let y = x &/ 0
// y □ □ 0
```

13.22 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？

```
2 + 3 * 4 % 5
// □ □ □ 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 □ 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。

```
2 + ((3 * 4) % 5)
```

(3 4) is 12, so this is equivalent to: 3 4 = 12，所以这相当于：

```
2 + (12 % 5)
```

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

```
2 + 2
```

计算结果为 4。

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。

13.23 运算符函数

让已有的运算符也可以对自定义的类和结构进行运算，这称为运算符重载。

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

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

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

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

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。

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector □ □ □ □ □Vector2D, □ □ (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。如下图示:

13.24 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++。

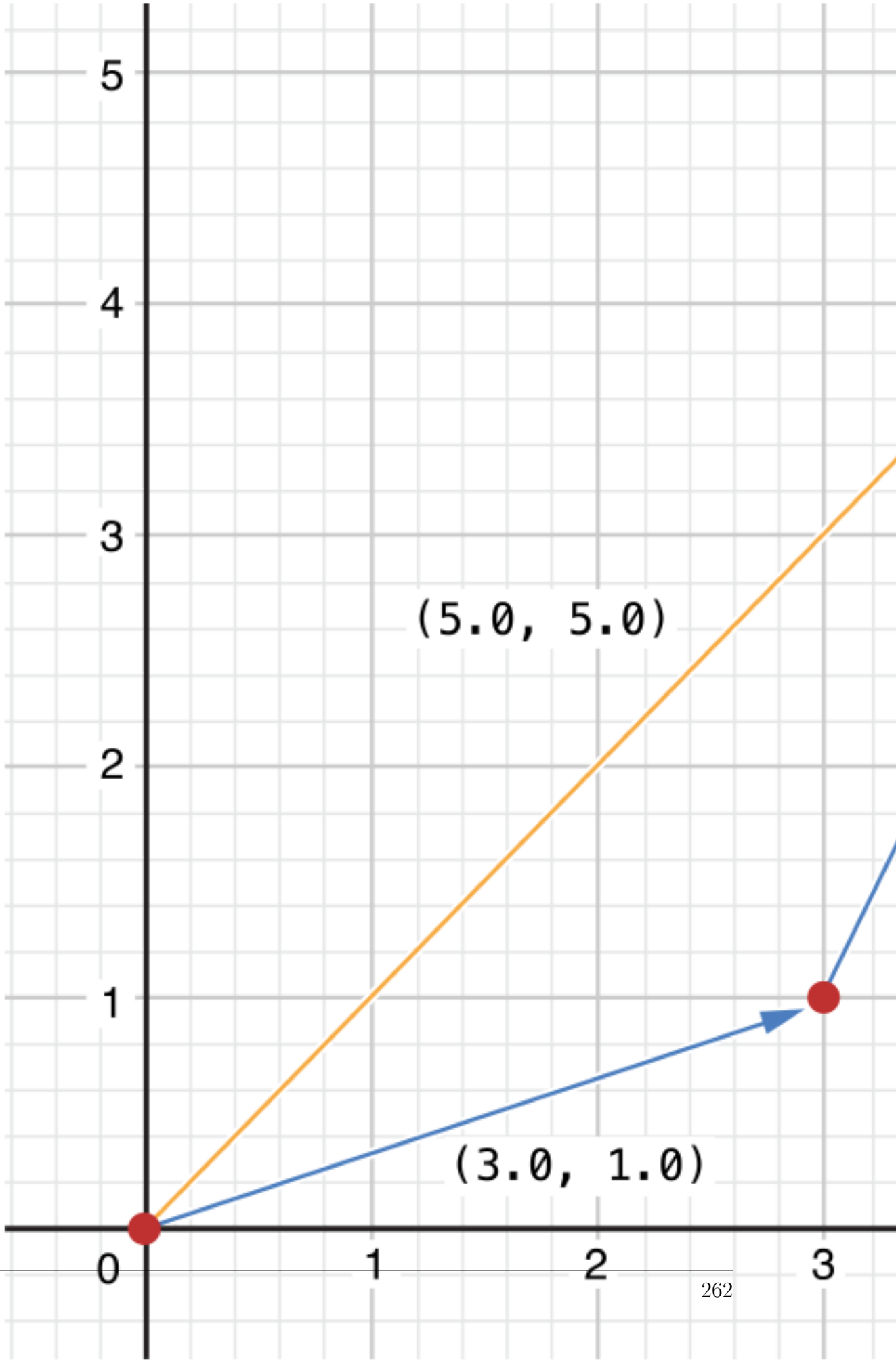
实现一个前置或后置运算符时，在定义该运算符的时候于关键字 func 之前标注 @prefix 或 @postfix 属性。

```
@prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

这段代码为 Vector2D 类型提供了单目减运算 -a，@prefix 属性表明这是个前置运算符。

对于数值，单目减运算符可以把正数变负数，把负数变正数。对于 Vector2D，单目减运算将其 x 和 y 都进进行单目减运算。

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative □ (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive □ (3.0, 4.0)
```


5
4
3
2
1
0
1
2
3
(5.0, 5.0)
(3.0, 1.0)

13.24.1 组合赋值运算符

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

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

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

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original □ □ □ (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。

```
@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

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。

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement □ □ □ (4.0, 5.0)
// afterIncrement □ □ □ □ (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。

13.24.2 比较运算符

Swift 无所知道自定义类型是否相等或不等，因为等于或者不等于由你的代码说了算了。所以自定义的类和结构要使用比较符 == 或!= 就需要重载。

定义相等运算符函数跟定义其他中置运算符雷同:

```
@infix func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}

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

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

现在我们可以使用这两个运算符来判断两个 Vector2D 对象是否相等。

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("□ □ □ □ □ □ □ □ □.")
}
// prints "□ □ □ □ □ □ □ □ □."
```

13.24.3 自定义运算符

标准的运算符不够玩，那你可以声明一些个性的运算符，但个性的运算符只能使用这些字符/ = - + * % < >! & | ^。~。

新的运算符声明需在全局域使用 operator 关键字声明，可以声明为前置，中置或后置的。

```
operator prefix +++ {}
```

这段代码定义了一个新的前置运算符叫 +++，此前 Swift 并不存在这个运算符。此处为了演示，我们让 +++ 对 Vector2D 对象的操作定义为双自增这样一个独有的操作，这个操作使用了之前定义的加赋运算实现了自已加上自已然后返回的运算。

```
@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

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

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled □ □ □ (2.0, 8.0)
// afterDoubling □ □ □ □ (2.0, 8.0)
```

13.24.4 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。

```
operator infix +- { associativity left precedence 140 }
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector □ □ □ □ □ (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;

本文部分原文来自于 http://www.swiftguide.cn/翻译小组的译文，共同校对中。The-Swift-Programming-Language-in-Chinese ==========

The Chinese Translation of The Swift Programming Language.

Created using pandoc and Markdown.

13.25 Usage:

```
build.py [pdf|html|epub|beamer]

directory for output:

    output  - beamer
            - pdf
            - epub
            - html
                - pic
```

the default value of build.py is html. It will use default.html as a template and turn all markdown files in root dir into html in output dir. Then you can preview it in your browser.

Pdf/beamer tranformation needs Latex (actual Xetex). TexLive is suggested.

Tested under Ubuntu and Windows7.

13.26 Markdown Grammar

see this for refrence.

13.27 Tools

- pandoc
- sublime
- vim

感谢翻译小组成员：李起攀 (微博)、若晨 (微博)、YAO、粽子、山有木兮木有枝、渺 -Bessie、墨离、Tiger 大顾 (微博)，校对：CXH_ME(微博),Joshua 孟思拓 (微博)

本文由翻译小组成员原创发布，个人转载请注明出处和原始链接，商业转载请联系我们 ~ 感谢您对我们工作的支持 ~

参考文献