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 1

1.1 常量和变量 1

1.1.1 常量和变量的声明 1

1.1.2 类型注解 2

1.1.3 常量和变量的命名 2

1.1.4 输出常量和变量 3

1.2 注释 4

1.3 分号 4

1.4 整数 5

1.4.1 整数边界 5

1.4.2 Int 类型 5

1.4.3 UInt 类型 5

1.5 数值量表达 7

1.6 数值类型转换 8

1.6.1 整数转换 8

1.6.2 整数和浮点数转换 9

1.7 类型别名 10

1.8 布尔类型 10

1.9 元组类型 11

1.10 可选类型 12

1.10.1 if 语句和强制解包 13

1.10.2 选择绑定 14

1.10.3 nil 14

1.10.4 隐式解包可选类型 15

1.11 断言 16

1.11.1 使用断言调试 16

1.11.2 使用断言的时机 17

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。

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)