基础内容

cnswift.org/the-basics

Swift 是一门全新的用于开发 iOS, OS X 以及 watchOS 应用的编程语言。不过，如果你有 C 或者Objective-C 语言开发经验的话，Swift 的许多地方都会让你感到熟悉。

Swift 为所有 C 和 Objective-C 的类型提供了自己的版本，包括整型值的 Int ，浮点数值的 Double 和 Float ，布尔量值的 Bool ，字符串值的 String 。如同集合类型中描述的那样，Swift 同样也为三个主要的集合类型提供了更高效的版本， Array ， Set 和 Dictionary 。

和 C 一样，Swift 用变量存储和调用值，通过变量名来做区分。Swift 中也大量采用了值不可变的变量。它们就是所谓的常量，但是它们比 C 中的常量更加给力。当你所处理的值不需要更改时，使用常量会让你的代码更加安全、简洁地表达你的意图。

除了我们熟悉的类型以外，Swift 还增加了 Objective-C 中没有的类型，比如元组。元组允许你来创建和传递一组数据。你可以利用元组在一个函数中以单个复合值的形式返回多个值。

Swift 还增加了可选项，用来处理没有值的情况。可选项意味着要么“这里*有*一个值，它*等于* x”要么“这里根本*没有*值”。可选项类似于 Objective-C 中的 nil 指针，但是不只是类，可选项也可以用在所有的类型上。可选项比 Objective-C 中的 nil 指针更安全、更易读，他也是 Swift 语言中许多重要功能的核心。

可选项充分证明了 Swift 是一门*类型安全*的语言。Swift 帮助你明确代码可以操作值的类型。如果你的一段代码预期得到一个 String ，类型会安全地阻止你不小心传入 Int 。在开发过程中，这个限制能帮助你在开发过程中更早地发现并修复错误。

常量和变量

常量和变量把名字（例如 maximumNumberOfLoginAttempts 或者 welcomeMessage ）和一个特定类型的值（例如数字 10 或者字符串 “Hello”）关联起来。*常量*的值一旦设置好便不能再被更改，然而*变量*可以在将来被设置为不同的值。

声明常量和变量

常量和变量必须在使用前被声明，使用关键字 let 来声明常量，使用关键字 var 来声明变量。这里有一个如何利用常量和变量记录用户登录次数的栗子：

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

这段代码可以读作：

“声明一个叫做 maximumNumberOfLoginAttempts 的新常量，并设置值为 10 。然后声明一个叫做 currentLoginAttempt 的新变量, 并且给他一个初始值 0。”

在这个栗子中，登录次数允许的最大值被声明为一个常量，因为最大值永远不会更改。当前尝试登录的次数被声明为一个变量，因为这个值在每次登录尝试失败之后会递增。

你可以在一行中声明多个变量或常量，用逗号分隔：

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

> 注意
>
> 在你的代码中，如果存储的值不会改变，请用 let 关键字将之声明为一个常量。只有储存会改变的值时才使用变量。

类型标注

你可以在声明一个变量或常量的时候提供*类型标注*，来明确变量或常量能够储存值的类型。添加类型标注的方法是在变量或常量的名字后边加一个冒号，再跟一个空格，最后加上要使用的类型名称。

下面这个栗子给一个叫做 welcomeMessage 的变量添加了类型标注，明确这个变量可以存储 String 类型的值。

```
1  var welcomeMessage: String
```

声明中的冒号的意思是“是…类型”，所以上面的代码可以读作：

“声明一个叫做 welcomMessage 的变量，他的类型是 String ”

我们说“类型是 String ”就意味着“可以存储任何 String 值”。也可以理解为“这类东西”（或者“这种东西”）可以被存储进去。

现在这个 welcomeMessage 变量就可以被设置到任何字符串中而不会报错了：

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

你可以在一行中定义多个相关的变量为相同的类型，用逗号分隔，只要在最后的变量名字后边加上类型标注。

```
1  var red, green, blue: Double
```

> 注意
> 实际上，你并不需要经常使用类型标注。如果你在定义一个常量或者变量的时候就给他设定一个初始值，那么 Swift 就像类型安全和类型推断中描述的那样，几乎都可以推断出这个常量或变量的类型。在上面 welcomeMessage 的栗子中，没有提供初始值，所以 welcomeMessage 这个变量使用了类型标注来明确它的类型而不是通过初始值的类型推断出来的。

命名常量和变量

常量和变量的名字几乎可以使用任何字符，甚至包括 Unicode 字符：

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

常量和变量的名字不能包含空白字符、数学符号、箭头、保留的（或者无效的）Unicode 码位、连线和制表符。也不能以数字开头，尽管数字几乎可以使用在名字其他的任何地方。

一旦你声明了一个确定类型的常量或者变量，就不能使用相同的名字再次进行声明，也不能让它改存其他类型的值。常量和变量之间也不能互换。

> 注意
> 如果你需要使用 Swift 保留的关键字来给常量或变量命名，可以使用反引号（`）包围它来作为名称。总之，除非别无选择，避免使用关键字作为名字除非你确实别无选择。

你可以把现有变量的值更改为其他相同类型的值。在这个栗子中 friendlyWelcome 的值从“Hello!” 改变为 “Bonjour!”

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome 现在是 "Bonjour!"
```

不同于变量，常量的值一旦设定则不能再被改变。尝试这么做将会在你代码编译时导致报错：

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

输出常量和变量

你可以使用 print(_:separator:terminator:) 函数来打印当前常量和变量中的值。

```
print(friendlyWelcome)
// 输出 "Bonjour!"
```

print(_:separator:terminator:) 是一个用来把一个或者多个值用合适的方式输出的全局函数。比如说，在 Xcode 中 print(_:separator:terminator:) 函数输出的内容会显示在Xcode的“console” 面板上。 separator 和 terminator 形式参数有默认值，所以你可以在调用这个函数的时候忽略它们。默认来说，函数通过在行末尾添加换行符来结束输出。要想输出不带换行符的值，那就传一个空的换行符作为结束——比如说， print(someValue, terminator: "") 。更多关于带有默认值的形式参数信息，见默认形式参数值。

Swift 使用*字符串插值* 的方式来把常量名或者变量名当做占位符加入到更长的字符串中，然后让 Swift 用常量或变量的当前值替换这些占位符。将常量或变量名放入圆括号中并在括号前使用反斜杠将其转义：

```
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// 输出 "The current value of friendlyWelcome is Bonjour!"
```

> 注意
> 字符串插值中描述了你可以使用字符串插值的所有选项。

注释

使用注释来将不需要执行的文本放入的代码当中，作为标记或者你自己的提醒。当 Swift 编译器在编译代码的时候会忽略掉你的注释。

Swift 中的注释和 C 的注释基本相同。单行注释用两个斜杠开头（ // ）：

```
// 这是一个注释
```

多行的注释以一个斜杠加一个星号开头（ /* ），以一个星号加斜杠结尾(*/)。

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

和 C 中的多行注释不同的是， Swift 语言中的多行的注释可以内嵌在其它的多行注释之中，你可以在多行注释中先开启一个注释块，接着再开启另一个注释块。然后关闭第二个注释块，再关闭第一个注释块。

```
/* 这是第一个多行注释的开头
/* 这是第二个嵌套在内的注释块 */
这是第一个注释块的结尾*/
```

内嵌多行注释，可以便捷地注释掉一大段代码块，即使这段代码块中已经有了多行注释。

分号

和许多其他的语言不同，Swift 并不要求你在每一句代码结尾写分号（；），当然如果你想写的话也没问题。总之，如果你想在一行里写多句代码，分号还*是*需要的。

```
let cat = "😺"; print(cat)
// 输出 "😺"
```

整数

*整数*就是没有小数部分的数字，比如 42 和 -23 。整数可以是*有符号*（正，零或者负），或者*无符号*（正数或零）。

Swift 提供了 8，16，32 和 64 位编码的有符号和无符号整数，这些整数类型的命名方式和 C 相似，例如 8 位无符号整数的类型是 UInt8 ，32 位有符号整数的类型是 Int32 。与 Swift 中的其他类型相同，这些整数类型也用开头大写命名法。

整数范围

你可以通过 min 和 max 属性来访问每个整数类型的最小值和最大值：

```
let minValue = UInt8.min // 最小值是 0, 值的类型是 UInt8
let maxValue = UInt8.max // 最大值是 255, 值得类型是 UInt8
```

这些属性的值都是自适应大小的数字类型（比如说上边栗子里的 UInt8 ）并且因此可以在表达式中与在其他相同类型值同用。

Int

在大多数情况下，你不需要在你的代码中为整数设置一个特定的长度。Swift 提供了一个额外的整数类型： Int ，它拥有与当前平台的原生字相同的长度。

- 在32位平台上， Int 的长度和 Int32 相同。
- 在64位平台上， Int 的长度和 Int64 相同。

除非你需操作特定长度的整数，否则请尽量在代码中使用 Int 作为你的整数的值类型。这样能提高代码的统一性和兼容性，即使在 32 位的平台上， Int 也可以存 -2,147,483,648 到 2,147,483,647 之间的任意值，对于大多数整数区间来说完全够用了。

UInt

Swift 也提供了一种无符号的整数类型， UInt ，它和当前平台的原生字长度相同。

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

> 注意
> 只在的确需要存储一个和当前平台原生字长度相同的无符号整数的时候才使用 UInt 。其他情况下，推荐使用 Int ，即使已经知道存储的值都是非负的。如同类型安全和类型推断中描述的那样，统一使用 Int 会提高代码的兼容性，同时可以避免不同数字类型之间的转换问题，也符合整数的类型推断。

浮点数

*浮点数*是有小数的数字，比如 3.14159 , 0.1 , 和 -273.15 。
浮点类型相比整数类型来说能表示更大范围的值，可以存储比 Int 类型更大或者更小的数字。Swift 提供了两种有符号的浮点数类型。

- Double代表 64 位的浮点数。
- Float 代表 32 位的浮点数。

> 注意
> Double 有至少 15 位数字的精度，而 Float 的精度只有 6 位。具体使用哪种浮点类型取决于你代码需要处理的值范围。在两种类型都可以的情况下，推荐使用 Double 类型。

类型安全和类型推断

Swift 是一门类型安全的语言。类型安全的语言可以让你清楚地知道代码可以处理的值的类型。如果你的一部分代码期望获得 String ，你就不能错误的传给它一个 *Int* 。

因为 Swift 是类型安全的，他在编译代码的时候会进行类型检查，任何不匹配的类型都会被标记为错误。这会帮助你在开发阶段更早的发现并修复错误。

当你操作不同类型的值时，类型检查能帮助你避免错误。当然，这并不意味着你得为每一个常量或变量声明一个特定的类型。如果你没有为所需要的值进行类型声明，Swift 会使用类型推断的功能推断出合适的类型。通过检查你给变量赋的值，类型推断能够在编译阶段自动的推断出值的类型。

因为有了类型推断，Swift 和 C 以及 Objective-C 相比，只需要少量的类型声明。其实常量和变量仍然需要明确的类型，但是大部分的声明工作 Swift 会帮你做。

在你为一个变量或常量设定一个初始值的时候，类型推断就显得更加有用。它通常在你声明一个变量或常量同时设置一个初始的*字面量（文本）*时就已经完成。（字面量就是会直接出现在你代码中的值，比如下边代码中的 42 和 3.14159 。）

举个栗子，如果你给一个新的常量设定一个 42 的字面量，而且没有说它的类型是什么，Swift 会推断这个常量的类型是 Int ，因为你给这个常量初始化为一个看起来像是一个整数的数字。

```
1  let meaningOfLife = 42
2  // meaningOfLife is inferred to be of type Int
```

同样，如果你没有为一个浮点值的字面量设定类型，Swift 会推断你想创建一个 Double 。

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

Swift 在推断浮点值的时候始终会选择 Double （而不是 Float ）。

如果你在一个表达式中将整数和浮点数结合起来， Double 会从内容中被推断出来。

```
1  let anotherPi = 3 + 0.14159
2  // anotherPi is also inferred to be of type Double
```

这字面量 3 没有显式的声明它的类型，但因为后边有一个浮点类型的字面量，所以这个类型就被推断为 Double 。

数值型字面量

整数型字面量可以写作：

- 一个十进制数，没有前缀
- 一个二进制数，前缀是 0b
- 一个八进制数，前缀是 0o
- 一个十六进制数，前缀是 0x

下面的这些所有整数字面量的十进制值都是 17 ：

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

浮点字面量可以是十进制（没有前缀）或者是十六进制（前缀是 0x ）。小数点两边必须有至少一个十进制数字（或者是十六进制的数字）。十进制的浮点字面量还有一个可选的*指数*，用大写或小写的 e 表示；十六进制的浮点字面量必须有指数，用大写或小写的 p 来表示。

十进制数与 exp 的指数，结果就等于基数乘以 10^{exp}：

- 1.25e2 意味着 1.25×10^{2}, 或者 125.0 .
- 1.25e-2 意味着 1.25×10^{-2}, 或者 0.0125 .

十六进制数与 exp 指数，结果就等于基数乘以2^{exp}：

- 0xFp2 意味着 15×2^{2}, 或者 60.0 .
- 0xFp-2 意味着 15×2^{-2}, 或者 3.75 .

下面的这些浮点字面量的值都是十进制的 12.1875 ：

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

数值型字面量也可以增加额外的格式使代码更加易读。整数和浮点数都可以添加额外的零或者添加下划线来增加代码的可读性。下面的这些格式都不会影响字面量的值。

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

数值类型转换

通常来讲，即使我们知道代码中的整数变量和常量是非负的，我们也会使用 Int 类型。经常使用默认的整数类型可以确保你的整数常量和变量可以直接被复用并且符合整数字面量的类型推测。

只有在特殊情况下才会使用整数的其他类型，例如需要处理外部长度明确的数据或者为了优化性能、内存占用等其他必要情况。在这些情况下，使用指定长度的类型可以帮助你及时发现意外的值溢出和隐式记录正在使用数据的本质。

整数转换

不同整数的类型在变量和常量中存储的数字范围是不同的。 Int8 类型的常量或变量可以存储的数字范围是 -128~127，而 UInt8 类型的常量或者变量能存储的数字范围是 0~255 。如果数字超出了常量或者变量可存储的范围，编译的时候就会报错：

```
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 than its maximum value,
// and so this will also report an error
```

因为每个数值类型可存储的值的范围不同，你必须根据不同的情况进行数值类型的转换。这种选择性使用的方式可以避免隐式转换的错误并使你代码中的类型转换意图更加清晰。

要将一种数字类型转换成另外一种类型，你需要用当前值来初始化一个期望的类型。在下面的栗子中，常量 twoThousand 的类型是 UInt16 ，而常量 one 的类型是 UInt8 。他们不能直接被相加在一起，因为他们的类型不同。所以，这里让 UInt16 (one) 创建一个新的 UInt16 类型并用 one 的值初始化，这样就可以在原来的地方使用了。

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

因为加号两边的类型现在都是 UInt16 ，所以现在是可以相加的。输出的常量（ twoThousandAndOne ）被推断为 UInt16 类型，因为他是两个 UInt16 类型的和。

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

整数和浮点数转换

整数和浮点数类型的转换必须显式地指定类型：

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

在这里，常量 three 的值被用来创建一个类型为 Double 的新的值，所以加号两边的值的类型是相同的。没有这个转换，加法就无法进行。

浮点转换为整数也必须显式地指定类型。一个整数类型可以用一个 Double 或者 Float 值初始化。

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

在用浮点数初始化一个新的整数类型的时候，数值会被截断。也就是说 4.75 会变成 4 ，-3.9 会变为 -3 。

> 注意
>
> 结合数字常量和变量的规则与结合数字字面量的规则不同，字面量 3 可以直接和字面量 0.14159 相加，因为数字字面量本身没有明确的类型。它们的类型只有在编译器需要计算的时候才会被推测出来。

类型别名

*类型别名*可以为已经存在的类型定义了一个新的可选名字。用 typealias 关键字定义类型别名。

当你根据上下文的语境想要给类型一个更有意义的名字的时候，类型别名会非常高效，例如处理外部资源中特定长度的数据时：

```
typealias AudioSample = UInt16
```

一旦为类型创建了一个别名，你就可以在任何使用原始名字的地方使用这个别名。

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

在这个栗子中， AudioSample 就是 UInt16 的别名，因为这个别名的存在，我们调用 AudioSample.min 其实就是在调用 UInt16.min ，在这里变量 maxAmplitudeFound 被提供了一个初始值 0 。

布尔值

Swift 有一个基础的布尔量类型，就是 Bool ，布尔量被作为逻辑值来引用，因为他的值只能是真或者假。Swift为布尔量提供了两个常量值， true 和 false 。

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

上面的两个类型 orangesAreOrange 和 turnipsAreDelicious ，被推断为 Bool ，因为它们使用布尔量来初始化。对于上文中的 Int 和 Double ，当你在创建的他们的时候设置为 true 或 false ，那么就不必给这个常量或者变量声明为 Bool 类型。初始化常量或者变量的时候，如果值的类型已知，类型推断会把 Swift 代码变的更加整洁和易读。

当你处理条件语句的时候例如 if 语句时，布尔值就会变得非常有用：

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

关于条件判断语句例如 if 语句，请参考控制流。

Swift 的类型安全机制会阻止你用一个非布尔量的值替换掉 Bool 。下面的栗子中报告了一个发生在编译时的错误：

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

然而，下边的这个例子就是可行的：

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

这里 i == 1 的比较结果是一个 Bool 类型，所以第二个栗子可以通过类型检查。类似 i == 1 这样的比较请参考基本运算符。

与 Swift 中其他的类型安全示例一样，这个方法可以避免错误的发生并确保这块代码的意图清晰。

元组

*元组*把多个值合并成单一的复合型的值。元组内的值可以是任何类型，而且可以不必是同一类型。

在下面的示例中， (404, "Not Found") 是一个描述了 *HTTP 状态代码* 的元组。HTTP 状态代码是当你请求网页的时候 web 服务器返回的一个特殊值。当你请求不存在的网页时，就会返回 404 Not Found

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

(404, "Not Found") 元组把一个 Int 和一个 String 组合起来表示 HTTP 状态代码的两种不同的值：数字和人类可读的描述。他可以被描述为“一个类型为 (Int, String) 的元组”

任何类型的排列都可以被用来创建一个元组，他可以包含任意多的类型。例如 (Int, Int, Int) 或者 (String, Bool) ，实际上，任何类型的组合都是可以的。

你也可以将一个元组的内容分解成单独的常量或变量，这样你就可以正常的使用它们了：

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

当你分解元组的时候，如果只需要使用其中的一部分数据，不需要的数据可以用下滑线（_）代替：

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

另外一种方法就是利用从零开始的索引数字访问元组中的单独元素：

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

你可以在定义元组的时候给其中的单个元素命名：

```
let http200Status = (statusCode: 200, description: "OK")
```

在命名之后，你就可以通过访问名字来获取元素的值了：

```
print("The status code is \(http200Status.statusCode)")
// prints "The status code is 200"
print("The status message is \(http200Status.description)")
// prints "The status message is OK"
```

作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个 (Int, String) 元组来描述是否获取成功。相比只能返回一个类型的值，元组能包含两个不同类型值，他可以让函数的返回信息更有用。更多内容请参考多返回值的函数。

> 注意
>
> 元组在临时的值组合中很有用，但是它们不适合创建复杂的数据结构。如果你的数据结构超出了临时使用的范围，那么请建立一个类或结构体来代替元组。更多信息请参考类和结构体。

可选项

可以利用*可选项*来处理值可能缺失的情况。可选项意味着：

这里有一个值，他等于*x*

或者

这里根本没有值

> 注意
>
> 在 C 和 Objective-C 中，没有可选项的概念。在 Objective-C 中有一个近似的特性，一个方法可以返回一个对象或者返回 nil 。 nil 的意思是“缺少一个可用对象”。然而，他只能用在对象上，却不能作用在结构体，基础的 C 类型和枚举值上。对于这些类型，Objective-C 会返回一个特殊的值（例如 *NSNotFound* ）来表示值的缺失。这种方法是建立在假设调用者知道这个特殊的值并记得去检查他。然而，Swift 中的可选项就可以让你知道*任何类型*的值的缺失，他并不需要一个特殊的值。

下面的栗子演示了可选项如何作用于值的缺失，Swift 的 Int 类型中有一个初始化器，可以将 String 值转换为一个 Int 值。然而并不是所有的字符串都可以转换成整数。字符串 “123” 可以被转换为数字值 123 ，但是字符串 "hello, world" 就显然不能转换为一个数字值。

在下面的栗子中，试图利用初始化器将一个 String 转换为 Int ：

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

因为这个初始化器可能会失败，所以他会返回一个*可选的* Int ，而不是 Int 。可选的 Int 写做 Int? ，而不是 Int 。问号明确了它储存的值是一个可选项，意思就是说它可能包含某些 Int 值，或者可能根本*不包含值*。（他不能包含其他的值，例如 Bool 值或者 String 值。它要么是 Int 要么什么都没有。）

nil

你可以通过给可选变量赋值一个 nil 来将之设置为没有值：

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

> 注意
>
> nil 不能用于非可选的常量或者变量，如果你的代码中变量或常量需要作用于特定条件下的值缺失，可以给他声明为相应类型的可选项。

如果你定义的可选变量没有提供一个默认值，变量会被自动设置成 nil 。

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

> 注意
>
> Swift 中的 nil 和Objective-C 中的 nil 不同，在 Objective-C 中 nil 是一个指向不存在对象的指针。在 Swift***中，*** nil 不是指针，他是值缺失的一种特殊类型，任何类型的可选项都可以设置成 nil 而不仅仅是对象类型。

If 语句以及强制展开

你可以利用 if 语句通过比较 nil 来判断一个可选中是否包含值。利用相等运算符 （ == ）和不等运算符（ != ）。

如果一个可选有值，他就“不等于” nil ：

```
1  if convertedNumber != nil {
2      print("convertedNumber contains some integer value.")
3  }
4  // prints "convertedNumber contains some integer value."
```

一旦你确定可选中包含值，你可以在可选的名字后面加一个感叹号 （！） 来获取值，感叹号的意思就是说“我知道这个可选项里边有值，展开吧。”这就是所谓的可选值的强制展开。

```
1  if convertedNumber != nil {
2      print("convertedNumber has an integer value of \(convertedNumber!).")
3  }
4  // prints "convertedNumber has an integer value of 123."
```

如需了解更多有关 *if* 语句的内容，请参考控制流。

> 注意
>
> 使用 ! 来获取一个不存在的可选值会导致运行错误，在使用***!***强制展开之前必须确保可选项中包含一个非 nil 的值。

可选项绑定

可以使用*可选项绑定*来判断可选项是否包含值，如果包含就把值赋给一个临时的常量或者变量。可选绑定可以与 if 和 while 的语句使用来检查可选项内部的值，并赋值给一个变量或常量。 if 和 while 语句的更多详细描述，请参考控制流。

在 if 语句中，这样书写可选绑定：

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

你可以像上面这样使用可选绑定而不是强制展开来重写 possibleNumber 这个例子：

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

代码可以读作：

“如果 Int(possibleNumber) 返回的可选 Int 包含一个值，将这个可选项中的值赋予一个叫做 actualNumber 的新常量。”

如果转换成功，常量 actualNumber 就可以用在 if 语句的第一个分支中，他早已被可选内部的值进行了初始化，所以这时就没有必要用 ! 后缀来获取里边的值。在这个栗子中 actualNumber 被用来输出转换后的值。

常量和变量都可以使用可选项绑定，如果你想操作 if 语句中第一个分支的 actualNumber 的值，你可以写 if var actualNumber 来代替，可选项内部包含的值就会被设置为一个变量而不是常量。

你可以在同一个 if 语句中包含多可选项绑定，用逗号分隔即可。如果任一可选绑定结果是 nil 或者布尔值为 false ，那么整个 if 判断会被看作 false 。下面的两个 if 语句是等价的：

```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber <
secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"
if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100"
```

隐式展开可选项

如上所述，可选项明确了常量或者变量可以“没有值”。可选项可以通过 if 语句来判断是否有值，如果有值的话可以通过可选项绑定来获取里边的值。

有时在一些程序结构中可选项一旦被设定值之后，就会一直拥有值。在这种情况下，就可以去掉检查的需求，也不必每次访问的时候都进行展开，因为它可以安全的确认每次访问的时候都有一个值。

这种类型的可选项被定义为*隐式展开可选项*。通过在声明的类型后边添加一个叹号（ String! ）而非问号（ String? ） 来书写隐式展开可选项。与在使用可选项时在名称后加一个叹号不同的是，声明的时候要把叹号放在类型的后面。

在可选项被定义的时候就能立即确认其中有值的情况下，隐式展开可选项非常有用。如同无主引用和隐式展开的可选属性中描述的那样，隐式展开可选项主要被用在 Swift 类的初始化过程中。

隐式展开可选项是后台场景中通用的可选项，但是同样可以像非可选值那样来使用，每次访问的时候都不需要展开。下面的栗子展示了在访问被明确为 String 的可选项展开值时，可选字符串和隐式展开可选字符串的行为区别：

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark
let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation mark

```

你可以把隐式展开可选项当做在每次访问它的时候被给予了自动进行展开的权限，你可以在声明可选项的时候添加一个叹号而不是每次调用的时候在可选项后边添加一个叹号。当你使用隐式展开可选项值时，Swift 首先尝试将它作为普通可选值来使用；如果它不能作为可选项，Swift 就强制展开值。在上面的代码中，可选值 assumedString 在给 implicitString 赋值前强制展开，因为 implicitString 有显式的非可选 String 类型。在下面的代码中， optionalString 没有显式写明类型所以它是普通可选项。

```
let optionalString = assumedString
// The type of optionalString is "String?" and assumedString isn't force-unwrapped.
```

如果隐式展开可选项结果是 nil ，你还尝试访问它的值，你就会触发运行时错误。结果和在没有值的普通可选项后面加一个叹号一样。

你可以和检查普通可选项一样检查隐式展开可选项是否为 nil ：

```
if assumedString != nil {
    print(assumedString)
}
// prints "An implicitly unwrapped optional string."
```

你也可以使用隐式展开可选项通过可选项绑定在一行代码中检查和展开值：

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

> 注意
>
> 不要在一个变量将来会变为 nil 的情况下使用隐式展开可选项。如果你需要检查一个变量在生存期内是否会变为 nil ，就使用普通的可选项。

错误处理

在程序执行阶段，你可以使用*错误处理*机制来为错误状况负责。

相比于可选项的通过值是否缺失来判断程序的执行正确与否，而错误处理机制能允许你判断错误的形成原因，在必要的情况下，还能将你的代码中的错误传递到程序的其他地方。

当一个函数遇到错误情况，他会*抛出*一个错误，这个函数的访问者会*捕捉*到这个错误，并作出合适的反应。

```
1  func canThrowAnError() throws {
2      // this function may or may not throw an error
3  }
```

通过在函数声明过程当中加入 throws 关键字来表明这个函数会抛出一个错误。当你调用了一个可以抛出错误的函数时，需要在表达式前预置 try 关键字。

Swift 会自动将错误传递到它们的生效范围之外，直到它们被 catch 分句处理。

```
1  do {
2      try canThrowAnError()
3      // no error was thrown
4  } catch {
5      // an error was thrown
6  }
```

do 语句创建了一个新的容器范围，可以让错误被传递到到不止一个的 catch 分句里。

下面的栗子演示了如何利用错误处理机制处理不同的错误情况：

```
1   func makeASandwich() throws {
2       // ...
3   }
4   do {
5       try makeASandwich()
6       eatASandwich()
7   } catch Error.OutOfCleanDishes {
8       washDishes()
9   } catch Error.MissingIngredients(let ingredients) {
10      buyGroceries(ingredients)
11  }
12
```

在上面的栗子中，在没有干净的盘子或者缺少原料的情况下，方法 makeASandwich() 就会抛出一个错误。由于 makeASandwich() 的抛出，方法的调用被包裹在了一个 try 的表达式中。通过将方法的调用包裹在 do 语句中，任何抛出来的错误都会被传递到预先提供的 catch 分句中。

如果没有错误抛出，方法 eatASandwich() 就会被调用，如果有错误抛出且满足 Error.OutOfCleanDishes 这个条件，方法 washDishes() 就会被执行。如果一个错误被抛出，而它又满足 Error.MissingIngredients 的条件，那么 buyGroceries(_:) 就会协同被 catch 模式捕获的 [String] 值一起调用。

有关抛出，捕获和错误传递的更详细信息请参考错误处理。

断言和先决条件

*断言*和*先决条件*用来检测运行时发生的事情。你可以使用它们来保证在执行后续代码前某必要条件是满足的。如果布尔条件在断言或先决条件中计算为 true ，代码就正常继续执行。如果条件计算为 false ，那么程序当前的状态就是非法的；代码执行结束，然后你的 app 终止。

你可以使用断言和先决条件来验证那些你在写代码时候的期望和假定，所以你可以包含它们作为你代码的一部分。断言能够帮助你在开发的过程中找到错误和不正确的假定，先决条件帮助你探测产品的问题。在运行时帮助你额外验证你的期望，断言和先决条件同样是代码中好用的证明形式。不同于在上文错误处理中讨论的，断言和先决条件并不用于可回复或者期望的错误。由于错误断言或先决条件显示非法的程序状态，所以没办法来抓取错误断言。

使用断言和先决条件不能代替你代码中小概率非法情况的处理设计。总之，使用他们来强制数据和状态正确会让你的 app 在有非法状态时终止的更可预料，并帮助你更好的 debug。在检测到异常状态时尽可能快地停止执行同样能够帮助你减小由于异常状态造成的损失。

断言和先决条件的不同之处在于他们什么时候做检查：断言只在 debug 构建的时候检查，但先决条件则在 debug 和生产构建中生效。在生产构建中，断言中的条件不会被计算。这就是说你可以在开发的过程当中随便使用断言而无需担心影响生产性能。

使用断言进行调试

断言会在运行的时候检查一个逻辑条件是否为 true 。顾名思义，断言可以“断言”一个条件是否为真。你可以使用断言确保在运行其他代码之前必要的条件已经被满足。如果条件判断为 true，代码运行会继续进行；如果条件判断为 false，代码运行结束，你的应用也就中止了。

如果你的代码在调试环境下触发了一个断言，例如你在 Xcode 中创建并运行一个应用，你可以明确的知道不可用的状态发生在什么地方，还能检查断言被触发时你的应用的状态。另外，断言还允许你附加一条调试的信息。

你可以使用全局函数 assert(_:_:) 函数来写断言。向 assert(_:_:) 函数传入一个结果为 true 或者 false 的表达式以及一条会在结果为 false 的时候显式的信息：

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

在这个例子当中，代码执行只要在 if age >= 0 评定为 true 时才会继续，就是说，如果 age 的值非负。如果 age 的值*是*负数，在上文的代码当中， age >= 0 评定为 false ，断言就会被触发，终止应用。

断言信息可以删掉如果你想的话，就像下边的栗子：

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

如果代码已经检查了条件，你可以使用 assertionFailure(_:file:line:) 函数来标明断言失败，比如：

```
1   if age > 10 {
2       print("You can ride the roller-coaster or the ferris wheel.")
3   } else if age > 0 {
4       print("You can ride the ferris wheel.")
5   } else {
6       assertionFailure("A person's age can't be less than zero.")
7   }
```

强制先决条件

在你代码中任何条件可能潜在为假但必须*肯定*为真才能继续执行的地方使用先决条件。比如说，使用先决条件来检测下标没有越界，或者检测函数是否收到了一个合法的值。

你可以通过调用 precondition(_:_:file:line:) 函数来写先决条件。给这个函数传入表达式计算为 true 或 false ，如果条件的结果是 false 信息就会显示出来。比如说：

```
1   // In the implementation of a subscript...
2   precondition(index > 0, "Index must be greater than zero.")
```

你可以调用 preconditionFailure(_:file:line:) 函数来标明错误发生了——比如说，如果 switch 的默认情况被选中，但所有的合法输入数据应该被其他 switch 的情况处理。

> 注意
>
> 如果你在不检查模式编译（ -Ounchecked ），先决条件不会检查。编译器假定先决条件永远为真，并且它根据你的代码进行优化。总之， fatalError(_:file:line:) 函数一定会终止执行，无论你优化设定如何。
>
> 你可以在草拟和早期开发过程中使用 fatalError(_:file:line:) 函数标记那些还没实现的功能，通过使用 fatalError("Unimplemented") 来作为代替。由于致命错误永远不会被优化，不同于断言和先决条件，你可以确定执行遇到这些临时占位永远会停止。