翻译：numbbbbb, lyuka, JaySurplus　校对：lslxdx

基础部分

本页包含内容：

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

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

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

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

值时，你可以用一个元组可以返回多个值。

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

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

常量和变量

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

声明常量和变量

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

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

这两行代码可以被理解为：

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

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

的时候都需要增加这个值。

你可以在一行中声明多个常量或者多个变量，用逗号隔开：

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

注意：

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

类型标注

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

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

```
var welcomeMessage: String
```

声明中的冒号代表着“是...类型”，所以这行代码可以被理解为：

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

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

`welcomeMessage`变量现在可以被设置成任意字符串：

```
welcomeMessage = "Hello"
```

注意：

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

常量和变量的命名

你可以用任何你喜欢的字符作为常量和变量名，包括 Unicode 字符：

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

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

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

注意：

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

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

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

与变量不同，常量的值一旦被确定就不能更改了。尝试这样做会导致编译时报错：

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

输出常量和变量

你可以用`println`函数来输出当前常量或变量的值：

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

`println`是一个用来输出的全局函数，输出的内容会在最后换行。如果你

用 Xcode，`println`将会输出内容到“console”面板上。(另一种函数叫`print`，唯一区别是在输出内容最后不会换行。)

`println`函数输出传入的`String`值:

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

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

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

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

注意:

字符串插值所有可用的选项，请参考字符串插值。

注释

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

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

```
// 这是一个注释
```

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

```
/* 这是一个，
```

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

```
/* 这是第一个多行注释的开头
/* 这是第二个被嵌套的多行注释 */
这是第一个多行注释的结尾 */
```
通过运用嵌套多行注释，你可以快速方便的注释掉一大段代码，即使这段代码之中已经含有了多行注释块。

分号

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

```
let cat = "🐱"; println(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、负、零）或者`无符号`（正、零）。

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

整数范围

你可以访问不同整数类型的`min`和`max`属性来获取对应类型的最大值和最小值：

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

Int

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

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

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

UInt

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

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

注意：

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

浮点数

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

浮点类型比整数类型表示的范围更大，可以存储比`Int`类型更大或者更小

的数字。Swift 提供了两种有符号浮点数类型：

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

注意：

`Double`精确度很高，至少有15位数字，而`Float`最少只有6位数字。选择哪个类型取决于你的代码需要处理的值的范围。

类型安全和类型推测

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

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

当你要处理不同类型的值时，类型检查可以帮你避免错误。然而，这并不是说你每次声明常量和变量的时候都需要显式指定类型。如果你没有显式指定类型，Swift 会使用类型推测*(type inference)*来选择合适的类型。有了类型推测，编译器可以在编译代码的时候自动推测出表达式的类型。原理很简单，只要检查你赋的值即可。

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

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

例如，如果你给一个新常量赋值`42`并且没有标明类型，Swift 可以推测出

常量类型是`Int`，因为你给它赋的初始值看起来像一个整数：

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

同理，如果你没有给浮点字面量标明类型，Swift 会推测你想要的是`Double`：

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

当推测浮点数的类型时，Swift 总是会选择`Double`而不是`Float`。

如果表达式中同时出现了整数和浮点数，会被推测为`Double`类型：

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

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

数值型字面量

整数字面量可以被写作：

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

下面的所有整数字面量的十进制值都是`17`：

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 二进制的17
let octalInteger = 0o21           // 八进制的17
let hexadecimalInteger = 0x11     // 十六进制的17
```

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）。浮点字面量还有一个可选的指数*(exponent)*，在十进制浮点数中通过大写或者小

写的`e`来指定，在十六进制浮点数中通过大写或者小写的`p`来指定。

如果一个十进制数的指数为`exp`，那这个数相当于基数和

10

exp

的乘积：

- `1.25e2` 表示 1.25×10^{2}，等于 `125.0`。
- `1.25e-2` 表示 1.25×10^{-2}，等于 `0.0125`。

如果一个十六进制数的指数为`exp`，那这个数相当于基数和

2

exp

的乘积：

- `0xFp2` 表示 15×2^{2}，等于 `60.0`。
- `0xFp-2` 表示 15×2^{-2}，等于 `3.75`。

下面的这些浮点字面量都等于十进制的`12.1875`：

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

数值类字面量可以包括额外的格式来增强可读性。整数和浮点数都可以添加额外的零并且包含下划线，并不会影响字面量：

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

数值型类型转换

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

整数转换

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

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

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

要将一种数字类型转换成另一种，你要用当前值来初始化一个期望类型的新数字，这个数字的类型就是你的目标类型。在下面的例子中，常量`twoThousand`是`UInt16`类型，然而常量`one`是`Uint8`类型。它们不能直接相加，因为它们类型不同。所以要调用`UInt16(one)`来创建一个新的

UInt16数字并用one的值来初始化，然后使用这个新数字来计算：

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

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

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

整数和浮点数转换

整数和浮点数的转换必须显式指定类型：

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

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

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

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

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

注意：

结合数字类常量和变量不同于结合数字类字面量。字面量3可以直接和字

面量`0.14159`相加，因为数字字面量本身没有明确的类型。它们的类型只在编译器需要求值的时候被推测。

类型别名

类型别名*(type aliases)*就是给现有类型定义另一个名字。你可以使用`typealias`关键字来定义类型别名。

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

```
typealias AudioSample = UInt16
```

定义了一个类型别名之后，你可以在任何使用原始名的地方使用别名：

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

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

布尔值

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

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

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

当你编写条件语句比如`if`语句的时候，布尔值非常有用：

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

条件语句，例如`if`，请参考控制流。

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

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

然而，下面的例子是合法的：

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

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

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

元组

元组（*tuples*）把多个值组合成一个复合值。元组内的值可以使任意类型，并不要求是相同类型。

下面这个例子中，`(404, "Not Found")`是一个描述 *HTTP* 状态码

（*HTTP status code*）的元组。HTTP 状态码是当你请求网页的时候 web 服务器返回的一个特殊值。如果你请求的网页不存在就会返回一个`404 Not Found`状态码。

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

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

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

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

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

如果你只需要一部分元组值，分解的时候可以把要忽略的部分用下划线（`_`）标记：

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

此外，你还可以通过下标来访问元组中的单个元素，下标从零开始：

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

你可以在定义元组的时候给单个元素命名：

```
let http200Status = (statusCode: 200, description:
```

```
"OK")
```
给元组中的元素命名后，你可以通过名字来获取这些元素的值：

```
println("The status code is \
(http200Status.statusCode)")
// 输出 "The status code is 200"
println("The status message is \
(http200Status.description)")
// 输出 "The status message is OK"
```
作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个`(Int, String)`元组来描述是否获取成功。和只能返回一个类型的值比较起来，一个包含两个不同类型值的元组可以让函数的返回信息更有用。请参考[函数参数与返回值(06_Functions.html#Function_Parameters_and_Return_Values)。

注意：

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

可选

使用可选（*optionals*）来处理值可能缺失的情况。可选表示：

- 有值，等于 x

或者

- 没有值

注意：

C 和 Objective-C 中并没有可选这个概念。最接近的是 Objective-C 中的一个特性，一个方法要不返回一个对象要不返回`nil`，`nil`表示“缺少一个合法的对象”。然而，这只对对象起作用——对于结构体，基本的 C 类型或者枚举类型不起作用。对于这些类型，Objective-C 方法一般会返回一个特殊值（比如`NSNotFound`）来暗示值缺失。这种方法假设方法的调用者知

道并记得对特殊值进行判断。然而，Swift 的可选可以让你暗示任意类型的值缺失，并不需要一个特殊值。

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

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

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

因为`toInt`方法可能会失败，所以它返回一个可选的（*optional*）`Int`，而不是一个`Int`。一个可选的`Int`被写作`Int?`而不是`Int`。问号暗示包含的值是可选，也就是说可能包含`Int`值也可能不包含值。（不能包含其他任何值比如`Bool`值或者`String`值。只能是`Int`或者什么都没有。）

if 语句以及强制解析

你可以使用`if`语句来判断一个可选是否包含值。如果可选有值，结果是`true`；如果没有值，结果是`false`。

当你确定可选包确实含值之后，你可以在可选的名字后面加一个感叹号(`!`)来获取值。这个惊叹号表示“我知道这个可选有值，请使用它。”这被称为可选值的强制解析（*forced unwrapping*）：

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

更多关于`if`语句的内容，请参考控制流。

注意：

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

可选绑定

使用可选绑定（*optional binding*）来判断可选是否包含值，如果包含就把值赋给一个临时常量或者变量。可选绑定可以用在`if`和`while`语句中来对可选的值进行判断并把值赋给一个常量或者变量。`if`和`while`语句，请参考控制流。

像下面这样在`if`语句中写一个可选绑定：

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

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

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

这段代码可以被理解为：

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

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

你可以在可选绑定中使用常量和变量。如果你想在`if`语句的第一个分支中操作`actualNumber`的值，你可以改成`if var actualNumber`，这样可

选包含的值就会被赋给一个变量而非常量。

nil

你可以给可选变量赋值为`nil`来表示它没有值：

```
var serverResponseCode: Int? = 404
// serverResponseCode 包含一个可选的 Int 值 404
serverResponseCode = nil
// serverResponseCode 现在不包含值
```

注意：

`nil`不能用于非可选的常量和变量。如果你的代码中有常量或者变量需要处理值缺失的情况，请把它们声明成对应的可选类型。
如果你声明一个可选常量或者变量但是没有赋值，它们会自动被设置为`nil`：

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

注意：

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

隐式解析可选

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

有时候在程序架构中，第一次被赋值之后，可以确定一个可选总会有值。在这种情况下，每次都要判断和解析可选值是非常低效的，因为可以确定它总会有值。

这种类型的可选被定义为隐式解析可选（*implicitly unwrapped optionals*）。把想要用作可选的类型的后面的问号（`String?`）改成感叹号（`String!`）来声明一个隐式解析可选。

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

一个隐式解析可选其实就是一个普通的可选，但是可以被当做非可选来使用，并不需要每次都使用解析来获取可选值。下面的例子展示了可选`String`和隐式解析可选`String`之间的区别：

```
let possibleString: String? = "An optional string."
println(possibleString!) // 需要惊叹号来获取值
// 输出 "An optional string."

let assumedString: String! = "An implicitly unwrapped optional string."
println(assumedString)  // 不需要感叹号
// 输出 "An implicitly unwrapped optional string."
```

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

注意：

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

你仍然可以把隐式解析可选当做普通可选来判断它是否包含值：

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

你也可以在可选绑定中使用隐式解析可选来检查并解析它的值：

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

注意：

如果一个变量之后可能变成`nil`的话请不要使用隐式解析可选。如果你需

要在变量的生命周期中判断是否是`nil`的话，请使用普通可选类型。

断言

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

使用断言进行调试

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

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

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

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

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

断言信息不能使用字符串插值。断言信息可以省略，就像这样：

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

何时使用断言

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

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

请参考附属脚本和函数。

注意：

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