基础

Swift是iOS，macOS，watchOS和tvOS应用程序开发的新编程语言。尽管如此，Swift的很多部分都将从您在C和Objective-C中开发的经验中熟悉起来。

雨燕提供了自己的所有基本C和Objective-C类型的版本，包括`Int`为整数，`Double`并`Float`为浮点值，`Bool`布尔值，并`String`为文本数据。雨燕还提供了三种主要类型的集合强大的版本`Array`，`Set`和`Dictionary`，如在集合类型。

像C一样，Swift使用变量来存储和引用一个识别名称的值。Swift还广泛使用其值不能更改的变量。这些被称为常量，并且比C中的常量强大得多。当您使用不需要更改的值时，常量在Swift中用于使代码更安全和更清晰。

除了熟悉的类型之外，Swift还引入了在Objective-C中找不到的高级类型，例如元组。元组使您能够创建和传递值组。您可以使用元组作为单个复合值从函数中返回多个值。

Swift还引入了可选类型，它处理缺少值的情况。选配说要么“有*是*一个值，它等于*X*”或“有*没有*一个价值可言”。使用optionals类似于`nil`在Objective-C中使用指针，但它们适用于任何类型，而不仅仅是类。`nil`Objective-C 中的选项不仅比指针更安全，更有表现力，它们是Swift最强大功能中的核心。

Swift是一种*类型安全的*语言，这意味着该语言可以帮助您清楚您的代码可以使用的值的类型。如果部分代码需要a `String`，那么类型安全性会阻止您`Int`错误地传递它。同样，类型安全性可以防止您意外地将可选`String`代码传递给需要非选项的代码片段`String`。类型安全可以帮助您在开发过程中尽早捕获并修复错误。

常量和变量

常量和变量将名称（如`maximumNumberOfLoginAttempts`或`welcomeMessage`）与特定类型的值（如数字`10`或字符串`"Hello"`）关联。*常量*的值一旦设置就无法更改，而将来可以将*变量*设置为不同的值。

声明常量和变量

常量和变量必须在使用之前声明。你用`let`关键字声明关键字和变量的常量`var`。以下是常量和变量如何用于跟踪用户进行的登录尝试次数的示例：

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

这段代码可以读作：

“声明一个新的常量`maximumNumberOfLoginAttempts`，并给它一个值`10`。然后，声明一个新的变量`currentLoginAttempt`，并给它一个初始值`0`。“

在此示例中，允许的登录尝试的最大数量被声明为常量，因为最大值永远不会更改。当前的登录尝试计数器被声明为变量，因为必须在每次失败的登录尝试后增加此值。

您可以在一行中声明多个常量或多个变量，并用逗号分隔：

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

> 注意
>
> 如果代码中的存储值不会更改，请始终使用`let`关键字将其声明为常量。仅使用变量来存储需要能够更改的值。

键入注释

您可以在声明常量或变量时提供*类型注释*，以清楚常量或变量可以存储的值的种类。通过在常量或变量名称后面加一个冒号，后跟一个空格，然后输入要使用的类型的名称，来编写一个类型注释。

此示例为名为变量的变量提供了一个类型注释`welcomeMessage`，以指示该变量可以存储`String`值：

```
var welcomeMessage: String
```

声明中的冒号意味着“……类型…”，所以上面的代码可以被读为：

“声明一个称为welcomeMessage该类型的变量String。”

“类型String”这
事物”）的含义。

welcomeMessage现在可以 将变量设置为任何字符串值而不会出错：

```
welcomeMessage = "Hello"
```

您可以在一行中定义多个相同类型的相关变量，用逗号分隔，最后一个变量名称后面带有单个类型注释：

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

> 注意
>
> 在实践中你很少需要编写类型注释。如果您在定义点处为常量或变量提供初始值，则Swift几乎总是可以推断出该常量或变量所用的类型，如类型安全和类型推断中所述。在welcomeMessage上面的示例中，没有提供初始值，因此welcomeMessage变量的类型是使用类型注释指定的，而不是从初始值推断的。

命名常量和变量

常量和变量名称几乎可以包含任何字符，包括Unicode字符：

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

常量和变量名称不能包含空白字符，数学符号，箭头，专用（或无效）Unicode代码点或线和框绘图字符。它们也不能以数字开头，尽管数字可能包含在名称的其他地方。

一旦声明了某个特定类型的常量或变量，就不能再用同一个名称声明它，或者将其更改为存储不同类型的值。你也不能将一个常量变成一个变量或一个常量变量。

> 注意
>
> 如果您需要为保留的Swift关键字指定一个常量或变量，请使用反引号（`）作为名称来包围关键字。但是，除非您完全没有选择，否则请避免使用关键字作为名称。

您可以将现有变量的值更改为兼容类型的另一个值。在这个例子中，值friendlyWelcome从"Hello!"变为"Bonjour!"：

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

与变量不同，常量的值在设置后无法更改。试图这样做是在编译代码时报告为错误：

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

打印常量和变量

您可以使用以下print(_:separator:terminator:)函数打印常量或变量的当前值：

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

该print(_:separator:terminator:)函数是一个全局函数，它将一个或多个值打印到适当的输出。例如，在Xcode中，该print(_:separator:terminator:)函数在Xcode的“控制台”窗格中输出其输出。该separator和

`terminator`参数都有默认值，所以当你调用这个函数，你可以忽略它们。默认情况下，该函数通过添加换行符终止它打印的行。要打印一个没有换行符的值，传递一个空字符串作为终止符 - 例如`print(someValue, terminator: "")`。有关使用默认值的参数的信息，请参阅默认参数值。

Swift使用*字符串*
变量的当前值。将名称包装在圆括号中，并在左括号之前用反斜杠进行转义：

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

> 注意
> 你可以用串插中使用的所有选项中描述字符串插值。

注释

使用注释在代码中包含不可执行的文本，作为便笺或提醒自己。在编译代码时，Swift编译器会忽略注释。

Swift中的注释与C中的注释非常相似。单行注释以两个正斜杠（`//`）开头：

```
// This is a comment.
```

多行注释以正斜杠开头，后跟星号（`/*`）并以星号结尾，后跟正斜杠（`*/`）：

```
/* This is also a comment
 but is written over multiple lines. */
```

与C中的多行注释不同，Swift中的多行注释可嵌套在其他多行注释中。您可以通过启动多行注释块并在第一个块内开始第二个多行注释来编写嵌套注释。然后关闭第二个块，然后关闭第一个块：

```
/* 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. */
```

嵌套的多行注释使您可以快速轻松地注释大量代码，即使代码已包含多行注释。

分号

与许多其他语言不同，Swift并不要求您`;`在代码中的每个语句之后都写一个分号（），尽管如果您愿意的话可以这样做。但是，如果要在单行中编写多个单独的语句，*则*需要分号：

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

整型

*整数*是没有小数分量的*整数*，如`42`和`-23`。整数是有*符号的*（正数，零或负数）或*无符号数*（正数或零）。

Swift以8,16,32和64位格式提供有符号和无符号整数。这些整数遵循类似于C的命名约定，其中8位无符号整数是类型的`UInt8`，并且32位有符号整数是类型的`Int32`。像Swift中的所有类型一样，这些整数类型都有大写的名字。

整数界限

你可以用它来访问每个整数类型的最小值和最大值`min`和`max`特性：

```
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
```

这些属性的值具有适当大小的数字类型（UInt8例如上面的示例中），因此可以在表达式中与其他相同类型的值一起使用。

诠释

在大多数情况下，您不需要选择特定的整数大小来在代码中使用。Swift提供了一个额外的整数类型，Int它与当前平台的本地字大小大小相同：

- 在32位平台上，Int尺寸与Int32。
- 在64位平台上，Int尺寸与Int64。

除非需要使用特定大小的整数，否则请始终Int在代码中使用整数值。这有助于代码的一致性和互操作性。即使在32位平台上，Int也可以存储-2,147,483,648和之间的任何值2,147,483,647，并且对于很多整数范围来说足够大。

UINT

Swift还提供了一个无符号整数类型，UInt它与当前平台的本地字大小大小相同：

- 在32位平台上，UInt尺寸与UInt32。
- 在64位平台上，UInt尺寸与UInt64。

> 注意
>
> 使用UInt只有当你特别需要具有相同大小的平台的本地字大小的无符号整型。如果不是这种情况，Int即使要存储的值已知是非负的，也是优选的。如类型安全和类型推断中Int所述，对整数值的一致使用有助于代码互操作性，避免了在不同数字类型之间进行转换以及匹配整数类型推断的需要。

浮点数字

*浮点数*是具有小数部分的数字，例如3.14159，0.1，和-273.15。

浮点类型可以表示比整数类型更广泛的数值范围，并且可以存储比可以存储在数组中更大或更小的数字Int。Swift提供了两个有符号的浮点数类型：

- Double 表示一个64位浮点数。
- Float 表示一个32位浮点数。

> 注意
>
> Double具有至少15位十进制数字的精度，而精度Float可以小至6位十进制数字。要使用的适当的浮点类型取决于代码中需要使用的值的性质和范围。在任何类型都适合的情况下，这Double是首选。

类型安全和类型推断

Swift是一种*类型安全的*语言。类型安全的语言鼓励您清楚您的代码可以使用的值的类型。如果你的部分代码需要一个String，你不能Int错误地传递它。

因为Swift是类型安全的，所以它在编译代码时执行*类型检查*并将任何不匹配的类型标记为错误。这使您能够在开发过程中尽早捕获并修复错误。

类型检查有助于避免在使用不同类型的值时发生错误。但是，这并不意味着您必须指定您声明的每个常量和变量的类型。如果你没有指定你需要的值的类型，Swift使用*类型推断*来计算出合适的类型。类型推断使编译器能够在编译代码时自动推断特定表达式的类型，只需检查您提供的值即可。

由于类型推理，Swift比C或Objective-C等语言所需的类型声明要少得多。常量和变量仍然是明确的类型，但是很多指定类型的工作都是为你完成的。

当您使用初始值声明常量或变量时，类型推断特别有用。这通常是通过在声明它的点处为常量或变量赋予一个*文字值*（或*文字*）来完成的。（A字面值是直接出现在源代码中，如一个值`42`和`3.14159`在下面的例子。）

例如，如果你将一个字面值赋给一个`42`新的常量而不说明它是什么类型的话，Swift推断你希望这个常量是一个常量`Int`，因为你

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

同样，如果您没有为浮点文字指定类型，Swift会推断您想要创建一个`Double`：

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

当推断浮点数的类型时， Swift总是选择`Double`（而不是`Float`）。

如果在表达式中结合整数和浮点文字，`Double`将从上下文中推断出一种类型：

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

字面值`3`没有明确的类型本身，所以`Double`从浮点文字的存在推断出适当的输出类型作为加法的一部分。

数字文字

整数文字可以写成：

- 一个*十进制*数字，不带前缀
- 一个*二进制*数字，带有`0b`前缀
- 一个*八进制*数，有`0o`前缀
- 一个*十六进制*数字，带有`0x`前缀

所有这些整数文字都有一个十进制值`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
```

浮点文字可以是十进制（不带前缀）或十六进制（带`0x`前缀）。它们必须始终在小数点的两侧都有一个数字（或十六进制数字）。十进制浮点数也可以有一个可选的*指数*，用大写或小写表示`e`; 十六进制浮点数必须有一个指数，用大写或小写表示`p`。

对于指数为的十进制数`exp`，基数乘以10^{exp}：

- `1.25e2`意味着1.25×10^{2}，或`125.0`。
- `1.25e-2`意味着1.25×10^{-2}，或`0.0125`。

对于指数为的十六进制数`exp`，基数乘以2^{exp}：

- `0xFp2`指15×2^{2}，或`60.0`。
- `0xFp-2`指15×2^{-2}，或`3.75`。

所有这些浮点文字都有十进制值`12.1875`：

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

数字文字可以包含额外的格式，以便于阅读。整数和浮点数都可以用额外的零填充，并且可以包含下划线以提高可读性。这两种格式都不影响文字的基础值：

```
let pac
let one
let justOverOneMillion = 1_000_000.000_000_1
```

数字类型转换

Int即使已知它们是非负的，也可以在代码中 使用所有通用整数常量和变量的类型。在日常情况下使用默认的整数类型意味着整数常量和变量可以在您的代码中立即互操作，并且将与整数文字值的推断类型匹配。

仅当手头任务特别需要使用其他整数类型时，才能使用其他整数类型，这是因为来自外部源的显式大小的数据，或性能，内存使用情况或其他必要的优化。在这些情况下使用明确大小的类型有助于捕获任何意外的值溢出，并隐含记录正在使用的数据的性质。

整数转换

每个数字类型可以存储在整数常量或变量中的数字范围不同。一个Int8常数或变量可以存储之间的数字-128和127，而UInt8常数或变量可以存储之间的数字0和255。当您的代码编译时，一个不适合大小整数类型的常量或变量的数字会报告为错误：

```
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
```

由于每种数字类型都可以存储不同范围的值，因此您必须逐个选择数字类型转换。这种选择加入方式可以防止隐藏的转换错误，并有助于在代码中明确地显示类型转换意图。

要将一个特定的号码类型转换为另一个号码类型，您需要使用现有的值初始化一个所需类型的新号码。在下面的例子中，常量twoThousand是类型的UInt16，而常量one是类型的UInt8。它们不能直接加在一起，因为它们不是同一类型。相反，这个例子调用UInt16(one)创建一个新UInt16值，初始值为one，并使用这个值代替原来的值：

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

因为添加的两侧都是现在的类型UInt16，所以允许添加。输出常量（twoThousandAndOne）被推断为是类型的UInt16，因为它是两个UInt16值的总和。

SomeType(ofInitialValue)是调用Swift类型的初始值设定项并传入初始值的默认方式。在幕后，UInt16有一个接受一个UInt8值的初始化器，所以这个初始化器被用来UInt16从现有的一个新的UInt8。你不能在这里传入*任何*类型，但是它必须是一个UInt16提供初始值的类型。扩展现有的类型，以提供接受新类型（包括你自己的类型定义）是覆盖在初始化扩展。

整数和浮点转换

整数和浮点数值类型之间的转换必须明确：

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

这里，常量的值three用于创建一个新的类型值Double，以便添加的两侧都是相同的类型。如果没有这种转换，则不允许添加。

浮点到整数转换也必须明确。整数类型可以使用Double或Float值来初始化：

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

用这种方式初始

> 注意
>
> 数字常量和变量的组合规则与数字文字的规则不同。字面值3可以直接添加到文字值`0.14159`，因为数字文字本身没有明确的类型。只有在编译器评估它们的时候才推断它们的类型。

类型别名

*类型别名*为现有类型定义了一个替代名称。您可以使用`typealias`关键字定义类型别名。

如果要通过上下文更合适的名称引用现有类型，例如在从外部源处理特定大小的数据时，类型别名很有用：

```
typealias AudioSample = UInt16
```

一旦你定义了一个类型别名，你可以在任何你可能使用原始名称的地方使用别名：

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

在这里，`AudioSample`被定义为一个别名`UInt16`。因为它是一个别名，`AudioSample.min`实际调用的调用为变量`UInt16.min`提供初始值。 `0maxAmplitudeFound`

布尔

Swift有一个基本的*布尔*类型，叫做`Bool`。布尔值被称为*逻辑*，因为它们只能是真或假。Swift提供了两个布尔常量值，`true`并且`false`：

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

由于它们是用布尔文字值初始化的事实推断`orangesAreOrange`和`turnipsAreDelicious`推断的类型`Bool`。正如`Int`和`Double`上面，你并不需要声明常量或变量`Bool`，如果将其设置为`true`或`false`为您创建它们尽快。当使用类型已知的其他值初始化常量或变量时，类型推断有助于使Swift代码更加简洁和易读。

当您使用条件语句（如`if`语句）时，布尔值特别有用：

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

条例语句（如`if`语句）在控制流程中有更详细的介绍。

Swift的类型安全性可以防止替换非布尔值`Bool`。以下示例报告编译时错误：

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

但是，下面的替代示例是有效的：

```
let i = 1
```

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

`i == 1`比较 的结果是类型的Bool，所以这第二个例子通过类型检查。比较像`1 == 1`在基本操作符中的比。

与Swift中的其他类型安全示例一样，这种方法避免了意外错误，并确保特定代码段的意图始终清晰。

元组

*元*组将多个值组合为一个复合值。元组中的值可以是任何类型，不必是彼此相同的类型。

在这个例子中，`(404, "Not Found")`是一个描述*HTTP状态码*的元组。每当您请求网页时，HTTP状态码都是Web服务器返回的特殊值。`404 Not Found`如果您请求不存在的网页，则会返回状态代码。

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

的`(404, "Not Found")`元组基团一起的`Int`和`String`，得到的HTTP状态代码两个独立的值：一个数字和一个人类可读的描述。它可以被描述为“一个类型的元组`(Int, String)`”。

您可以从任何类型的排列中创建元组，并且可以根据需要包含尽可能多的不同类型。没有什么阻止你有型的元组`(Int, Int, Int)`，或者`(String, Bool)`，或者你确实需要的任何其他排列。

您可以*将*元组的内容*分解*为单独的常量或变量，然后像往常一样访问它们：

```
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"
```

如果您只需要某些元组的值，则_在分解元组时，请使用下划线（）忽略部分元组：

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

或者，使用从零开始的索引号访问元组中的各个元素值：

```
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"
```

定义元组时，可以将元组中的各个元素命名为：

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

如果您命名元组中的元素，则可以使用元素名称来访问这些元素的值：

```
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"
```

元组作为函数的返回值特别有用。试图检索网页的函数可能会返回`(Int, String)`元组类型来描述页面检索的成功或失败。通过返回一个具有两个不同值的元组（每个元素都是不同的类型），该函数提供了有关其结果的更多有用信息，而不是仅返回单个类型的单个值。有关更多信息，请参阅具有多个返回值的函数。

> 注意

> 元组对于相关值的临时组是有用的。它们不适合创建复杂的数据结构。如果您的数据结构可能会持续超出临时范围，请将其建模为类或结构，而不是元组。有关更多信息，请参阅类和结构。

选配

在价值可能不存在的情况下 使用*可选项*。可选的代表两种可能性：要么有*是*一个值，你可以解开可选的访问值，或者有*没有*价值可言。

> 注意
>
> C或Objective-C中不存在可选项的概念。Objective-C中最接近的东西是`nil`从一个方法返回的能力，否则它会返回一个对象，`nil`意思是“缺少一个有效的对象”。但是，这只适用于对象 - 它不适用于结构，基本C类型或枚举值。对于这些类型，Objective-C方法通常会返回一个特殊值（如`NSNotFound`）来指示缺少值。这种方法假定方法的调用者知道有一个特殊的值来测试并记住检查它。Swift的选项允许你指出*任何类型*的值*都不存在*，而不需要特殊的常量。

以下是如何使用可选项来应对价值缺失的一个例子。Swift的`Int`类型有一个初始化器，它试图将一个`String`值转换为一个`Int`值。但是，并非每个字符串都可以转换为整数。该字符串`"123"`可以转换为数值`123`，但该字符串`"hello, world"`没有明显的数值转换。

下面的例子使用初始化器来尝试将a `String`转换为`Int`：

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

由于初始化程序可能失败，它会返回一个*可选项* `Int`，而不是一个`Int`。可选`Int`是写成`Int?`，而不是`Int`。问号表明它所包含的值是可选的，这意味着它可能包含*一些* `Int`值，或者它可能*根本不*包含*任何值*。（它不能包含其他任何东西，比如`Bool`值或`String`值，它可以是一个`Int`，或者什么也没有。）

零

通过将可选变量分配给特殊值，可将其设置为无值状态`nil`：

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

> 注意
>
> 你不能使用`nil`nonoptional常量和变量。如果代码中的常量或变量需要在某些条件下没有值时使用，则始终将其声明为适当类型的可选值。

如果您定义了一个可选变量而不提供默认值，则会自动`nil`为您设置该变量：

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

> 注意
>
> Swift `nil`与`nil`Objective-C中的不一样。在Objective-C中，`nil`是一个指向不存在对象的指针。在Swift中，`nil`不是一个指针 - 它是缺少某种类型的值。选配的*任何*类型可以设置为`nil`，不只是对象类型。

如果陈述和强制解包

您可以使用if语句通过比较可选的against来确定可选是否包含值nil。您使用“等于”运算符（==）或“不等于”运算符（!=）执行此比较。

如果可选项具有值，则认为它“不等于”nil：

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value."
```

一旦你确定了可选*确实*包含一个值，你可以通过添加一个感叹号（访问其基础值!）到可选的名称的末尾。感叹号有效地说：“我知道这个可选项肯定有价值; 请使用它“。这称为*强制展开*可选值：

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

有关该if声明的更多信息，请参阅控制流程。

> 注意
>
> 尝试使用!访问不存在的可选值触发运行时错误。nil在使用!强制解开其值时，务必确保可选包含非值。

可选绑定

您使用*可选绑定*来确定可选是否包含值，如果是，则将该值作为临时常量或变量提供。可选绑定可以用于if和while语句来检查可选内部的值，并将该值作为单个操作的一部分提取到常量或变量中。if并while在控制流程中更详细地描述报表。

为if语句编写一个可选的绑定，如下所示：

```
如果 让 constantName = someOptional {
    声明
}
```

您可以possibleNumber从Optionals部分重写示例以使用可选绑定而不是强制展开：

```
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"
```

这段代码可以读作：

“如果Int返回的可选参数Int(possibleNumber)包含一个值，则设置一个新的常量，以调用actualNumber可选参数中包含的值。”

如果转换成功，则该actualNumber常量可在if语句的第一个分支内使用。它已经使用包含*在*可选项中的值进行了初始化，因此不需要使用!后缀来访问它的值。在本例中，actualNumber仅用于打印转换结果。

您可以使用可选绑定的常量和变量。如果您想操作语句actualNumber第一个分支内的值if，您可以if var actualNumber改为写入，而可选内容中的值将作为变量而不是常量提供。

您可以根据需要在单个if语句中包含尽可能多的可选绑定和布尔条件，并用逗号分隔。如果可选绑定中的nil任何值是或任何布尔条件的计算结果false，则整个if语句的条件被认为是false。以下if声明是等同的：

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

> 注意
>
> 在if语句中使用可选绑定创建的常量和变量仅在if语句的主体内可用。相反，使用guard语句创建的常量和变量在语句后面的代码行中可用guard，如Early Exit中所述。

隐式解包选项

如上所述，可选项表示允许常量或变量具有“无值”。可以使用if语句来检查选项，以查看是否存在值，并且可以使用可选绑定有条件地解包，以访问可选值（如果存在）。

有时候从程序的结构中可以清楚地看到，在第一次设置值之后，可选项将*始终*有一个值。在这些情况下，每次访问时都不必检查和打开可选的值，因为可以安全地假定所有时间都有值。

这些选项被定义为*隐式解包选项*。通过在想要使其可选的类型之后放置感叹号（String!）而不是问号（String?），可以编写一个隐式解包的可选项。

当可选值被确定为在第一次定义可选值后立即存在，并且肯定可以假设其后每一点都存在时，隐式解包的可选值就很有用。Swift中隐式展开option的主要用途是在类初始化期间，如Unowned References和Implicitly Unwrapped Optional Properties中所述。

隐式解包可选是幕后的普通可选，但也可以像非可选值一样使用，而不必在每次访问时解开可选值。以下示例显示了在以显式方式访问它们的包装值时，可选字符串和隐式解包可选字符串之间的行为差异String：

```
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
```

你可以想象一个隐式解包的可选方案，因为只要使用它就可以自动解包可选的权限。每次使用该名称时，不要在可选名称后面放置感叹号，而要在声明它时在可选类型后面放置感叹号。

> 注意
>
> 如果隐式解包可选，nil并且您尝试访问其包装的值，则会触发运行时错误。结果与在不包含值的普通可选项后放置感叹号完全相同。

你仍然可以像一个普通的可选项那样处理隐式解包的可选项，以检查它是否包含值：

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

你也可以使用一个带有可选绑定的隐式解包的可选方法，在一个语句中检查并打开它的值：

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

> 注意
>
> 当变量有可能nil在稍后出现时，不要使用隐式解包的可选项。如果您需要nil在变量的生命周期中检查值，则始终使用正常的可选类型。

错误处理

您可以使用*错误处理*来响应程序在执行过程中可能遇到的错误情况。

与可以使用值的存在或不存在来传递函数的成功或失败的可选项相比，错误处理允许您确定失败的根本原因，并在必要时将错误传播到程序的另一部分。

当函数遇到错误情况时，它*会引发*错误。然后该函数的调用者可以*捕获*错误并做出适当的响应。

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

函数指示它可以通过throws在其声明中包含关键字来引发错误。当你调用一个可能引发错误的函数时，你try需要在表达式中添加关键字。

Swift会自动将错误传播出当前范围，直到它们由一个catch子句处理。

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

一个do语句创建一个新的包含范围，允许误差传播到一个或多个catch条款。

以下是错误处理如何用于响应不同错误条件的示例：

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

在这个例子中，makeASandwich()如果没有干净的菜肴或缺少任何配料，该功能将会报错。因为makeASandwich()可以抛出一个错误，函数调用被包装在一个try表达式中。通过在函数中包装函数调用do，所有抛出的错误都会传播到提供的catch子句中。

如果没有错误发生，eatASandwich()则调用该函数。如果抛出一个错误并且匹配SandwichError.outOfCleanDishes大小写，那么washDishes()函数将被调用。如果抛出一个错误并且它与SandwichError.missingIngredients大小写匹配，那么该buyGroceries(_:)函数将与[String]该catch模式捕获的关联值一起被调用。

错误处理中 详细介绍了投掷，捕捉和传播错误。

断言和先决条件

*断言*和*先决条件*是在运行时发生的检查。在执行任何进一步的代码之前，您可以使用它们来确保满足基本条件。如果断言或先决条件中的布尔条件评估为true，则代码执行将像往常一样继续。如果条件评估为false，程

序的当前状态无效; 代码执行结束，并且您的应用程序被终止。

您使用断言和先决条件来表达您所做的假设和编码时的期望，以便您可以将它们包含在代码中。断言有助于您在开发过程中发现错误和不正确的假设，并且先决条件可帮助您检测生产中的问题。

除了在运行时验证您的期望之外，断言和先决条件也成为代码中有用的文档形式。与上述错误处理中的错误条件不同，断言和先决条件不用于可恢复或预期的错误。由于失败的断言或先决条件表示无效的程序状态，因此无法捕获失败的断言。

使用断言和先决条件不能取代设计代码的方式，使得不可能出现无效条件。但是，使用它们来强制执行有效的数据和状态会导致应用程序在发生无效状态时更可预测地终止，并且有助于使问题更易于调试。一旦检测到无效状态，立即停止执行也有助于限制由该无效状态引起的损害。

断言和先决条件之间的区别在于它们被检查时：断言仅在调试版本中检查，但在调试版本和生产版本中都检查了先决条件。在生产构建中，断言内的条件不被评估。这意味着您可以在开发过程中使用尽可能多的断言，而不会影响生产性能。

使用断言进行调试

你通过调用`assert(_:_:file:line:)`Swift标准库中的函数来编写断言。如果条件结果是，则将此函数传递给一个表达式，该表达式将评估为`true`或`false`显示一条消息以显示`false`。例如：

```
let age = -3
assert(age >= 0, "A person's age can't be less than zero.")
// This assertion fails because -3 is not >= 0.
```

在这个例子中，如果`age >= 0`计算结果为代码执行继续`true`，也就是说，如果值`age`是非负的。如果值为`age`负值（如上面的代码所示），则`age >= 0`评估为`false`，并且断言失败，则终止该应用程序。

你可以省略断言信息 - 例如，当它只是重复作为散文的条件。

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

如果代码已经检查了条件，则使用该`assertionFailure(_:file:line:)`函数来指示断言失败。例如：

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

在本页

执行先决条件

使用时的条件必须是假的潜力的前提条件，但必须*肯定*是真的对你的代码继续执行。例如，使用前提条件检查下标是否超出界限，或者检查函数是否已传递有效值。

你通过调用该`precondition(_:_:file:line:)`函数来编写一个前提条件。如果条件结果是，则将此函数传递给一个表达式，该表达式将评估为`true`或`false`显示一条消息以显示`false`。例如：

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

您也可以调用该`preconditionFailure(_:file:line:)`函数来指示发生故障 - 例如，如果采用了交换机的默认情况，但所有有效的输入数据都应该由交换机的其中一种情况处理。

> 注意
>
> 如果以未检查模式编译（`-Ounchecked`），则不检查前置条件。编译器假定前提条件始终为真，并相应地优化您的代码。但是，`fatalError(_:file:line:)`无论优化设置如何，该函数都会暂停执行。
>
> 您可以`fatalError(_:file:line:)`在原型设计和早期开发过程中使用该函数，通过编写`fatalError("Unimplemented")`作为存根实现来为尚未实现的功能创建存根。由于致命错误永远不会被优化，与断言或前提条件不同，您可以确保执行始终在遇到存根实现时暂停。