第 1 章

Swift 入门

本章内容

- 了解 Swift
- 常量和变量的声明以及 Swift 数据类型的使用
- 使用运算符转换值
- 使用条件语句和循环控制代码的执行
- 定义和使用 Swift 的枚举数据类型
- 理解、声明和使用函数、匿名函数和闭包

本章介绍 Swift 编程语言的关键概念以及该语言的新语法和数据类型。本章并不是 Swift 的入门指南，它的目的是让已用过 Swift 的程序员复习一下该语言的知识。之前在 iOS 和 OS X 平台上使用 C 和 Objective-C 编程的知识对于理解本章仍然有用，而且本章中的知识对于没有 C 和 Objective-C 经验的人也有意义。

如果读者已经熟悉 Swift 编程语言的基础知识，可直接跳转到本书的第 2 章。第 2 章还介绍了如何使用 Xcode 的 playground 新特性，使用该特性可在编写 Swift 代码时直接看到其效果。本章的例子可直接输入到 playground，从而可立即看到执行结果，而不用创建 Xcode 项目和编译代码。

1.1 什么是 Swift

Swift 是一门全新的语言，由 Apple 公司开发，用来为开发 iOS 和 OS X 的 Objective-C 提供一种备选方案。尽管该语言被设计为可和 Objective-C 以及 C 和 C++无缝交互，但它并不是 Objective-C 的进化，而是一门和其血统大不相同的全新语言。它抛弃了很多经典的

Objective-C 语言特性，同时引入了大量使 iOS 和 OS X 程序开发更快、更安全的新特性，使得开发过程变得更便捷。

Swift 是一个开发了几年的产品，在其研发过程中参考了很多现有语言。Swift 远远不是 Objective-C 的简单改进，它的语言特性源自诸多编程语言，如 Haskell、C#、Ruby、Python 和 Rust 等。Swift 还吸收了 Cocoa 和 Cocoa Touch 框架的很多特性，如键-值观察。Swift 编译器还利用了很多在创建 LLVM 和 Objective-C 的 clang 编译器时获得的研究成果和经验。

与 Objective-C 不同，Swift 不是动态类型的语言。相反，它使用静态类型来帮助确保程序的完整性和安全性。Swift 还避免了许多 C 语言(包括 Objective-C)的固有问题，尤其是在确保内存完整性方面。尽管有经验的 Objective-C 开发者可能会认为这会丧失程序设计和构造的灵活性，但 Swift 的新特性会让程序的编写更简单和更容易，而且仍为程序员保留了很大的自由度。Apple 想让 Swift 既适合应用开发又适合系统编程，该语言的设计也反映了这一点。

幸运的是，Apple 已经非常注意将 Swift 整合到现有的生态系统。Swift 可以无缝地操作现有的 Objective-C 代码、库和框架，对于用 C 和 C++编写的代码也是如此。通过一些额外的设置甚至可以在程序中混合使用这些语言。Xcode 完全支持 Swift 代码，从而开发者可继续使用其在使用 Objective-C 进行开发时已经熟悉的开发工具，如编译器和调试器等。

实际上，Swift 已经为 Xcode 生态系统引入了一个新工具：playground。playground 是一个交互式开发环境，它会实时反馈代码块的执行结果。Swift 还支持“读取-求值-输出”循环(read-eval-print loop，REPL)，该过程可用于在控制台中测试 Swift 代码片段是否正确。有关 playground 的更多细节可参见本书的第 2 章。

Swift 是 iOS 和 OS X 开发的巨大飞跃。尽管 Objective-C 会在相当长的时间内被继续支持，但 Swift 是 iOS 和 OS X 开发的未来发展方向，掌握 Swift 引入的新技术是非常重要的。所幸开发者目前已掌握的 iOS 和 OS X 开发知识仍然十分重要，即使是完全使用 Swift 编写程序也是如此。最重要的是，Swift 的新特性使得 iOS 和 OS X 应用的编写变得比以前更刺激有趣。

在深入到 Swift 开发的细节前，首先应该熟悉一下 Swift 的基本概念。如果读者已经牢牢掌握了这些概念，可以跳转到本书的第 2 章。

1.2　为什么要学习 Swift

这里有一个更大的问题，那就是为什么不使用 Objective-C 编写 iOS 和 OS X 应用而是去学习 Swift？事实是 Objective-C 不会在短时间内消失。iOS 和 OS X 中的大多数框架都是用 Objective-C 编写的，另外还有大量可供选择的第三方库和框架，以及大多数 iOS 和 OS X 开发教程也都是用 Objective-C 编写的。透彻掌握 Objective-C 的知识对于 iOS 和 OS X 这

两个平台的开发者来说都是十分重要的。然而，Apple 已经分出了一大部分资源来开发 Swift，清楚地表明了 Swift 是 iOS 和 OS X 开发的未来方向。Apple 将 Swift 同时设计为应用和系统编程语言，并且确定将来很多操作系统组件、框架和库都会用 Swift 编写。在将来 Swift 将变得越来越重要。

此外，Swift 还扩展和改进了 Objective-C 的很多思想，并且添加了如闭包这样大量的新特性，同时使现有的如枚举类型这样的特性变得更丰富。Swift 的语法简洁，比 Objective-C 更容易学习和使用，并且其 API 更灵活。Swift 还允许 iOS 和 OS X 开发者探索与 Objective-C 固有的面向对象开发方法不同的其他开发方法，如函数式编程。最后，Swift 很有趣，它为 Apple 开发领域添加了一个新的探索方向。

1.3　使用常量和变量

Swift 具有与 C 和 Objective-C 这样的语言同样的数据类型，如 Int、Float 和 Double 这样的数值类型；Bool 这样的布尔类型；以及 String 和 Char 这样的字符类型。Swift 还在语言层面支持如 Array 和 Dictionary 这样更复杂的类型，并提供了优雅的语法来声明和使用这些容器类型。

Swift 使用名称来引用值。这些名称被称为变量。在 Swift 中，新变量由 var 关键字引入。与 C 和 Objective-C 一样，变量引用的实际值可被改变。在下面的例子中，变量 x 首先引用 10，然后引用 31：

```
var x = 10
x = 31
```

Swift 还支持值不能改变的变量，在此语境中，“变量”这个用词是不恰当的，这些标识符更应该被称为常量。常量使用 let 关键字引入：

```
let x = 10
```

在 x 声明后其值将不能再被改变——否则会导致编译错误。下面的代码是不被允许的：

```
let x = 10
x = 31   // This line will generate an error
```

因为常量值不能被改变，所以常量的状态比变量更容易推测。在 Swift 中，推荐尽量使用常量，只有值在程序执行期间确实需要改变时才使用变量。好的 Swift 程序总是多用常量少用变量。

注意：技术上而言，常量并不是变量，因为它们的值不会改变。然而在编程语言的上下文中，术语“变量”通常不遵循它的严格数学定义，而是表示一个名称，用于引用程序代码中的一个变量。在本书中，术语“变量”经常会在讨论变量和常量时用到，因为在 Swift 中所有适用于变量的概念也都适用于常量。当这两者有差异时会在本书中用文字明确表示出来。

Swift 还允许在变量名称中使用 Unicode 字符，而且不仅是英语字母表中的字符。下面的代码在 Swift 中是允许的：

```
let π = 3.14159
let r = 10.0
let area =π* r * r
```

几乎所有 Unicode 字符都可用在变量名中。然而仍然存在一些限制：变量名不能包含数学符号、箭头、私有和无效 Unicode 代码点，以及用于绘制线条和表的字符。变量名不能以数字开头，但在第一个字符之后可包含数字。即使存在这些限制，Swift 的变量名也比 C 和 Objective-C 的灵活很多。

1.3.1　理解 Swift 数据类型

对于 C 和 Objective-C 中的所有数据类型，Swift 都有自己的版本，包括各种数值类型、布尔类型以及字符类型等。Swift 还自行支持 Array 和 Dictionary 容器类型，并提供了简单直接的语法来声明和使用这些类型。Swift 还引入了新的数据类型 tuple，它可以将多个值组合到一个单独的复合类型中，并且这些值的类型可以不同。

1. 使用数值类型

Swift 提供三个基本数值数据类型：Int、Float 和 Double。这些类型对应 C 和 Objective-C 中的同样的数据类型。Int 表示无小数部分的数字，如 1304 和 10000。Float 和 Double 各自表示具有小数部分的数字，如 0.1、3.14159 和 70.0。和 C 语言一样，这两个类型的区别在于它们的精度不同：Float 是 32 位类型，而 Double 是 64 位类型。因为 Double 具有比 Float 更多的位数，所以它能提供比 Float 更高的精度。

注意：通常推荐使用 Double 而不是 Float，除非确实不需要 Double 这么高的精度并且不能牺牲更多的空间来存储 Double 数值。

Swift 中数值类型的写法与它们在 C 和 Objective-C 中的写法一样。然而为了增加可读性，可以在数值中包含下划线，这些下划线通常用于将数值分成几块。例如，值 1 000 000

可以写成这样：

```
let n = 1_000_000
```

下划线可用于 Int，也可用于 Float 和 Double。

Swift 数值类型具有所有数值运算符，包括+、−、/、*和%，以及和 C 一样的自增(++)和自减(−−)运算符。这些运算符将在本章后面详细介绍。

与 Objective-C 不同，在 Swift 中“基本类型”和“对象”之间没有差别，数值数据类型有它自己的方法。Swift 标准库提供了一系列处理数值类型的方法。

Swift 不提供如 short 和 long 这样的更特定的整数数据类型。与 C 和 Objective-C 不同，Swift 值提供单一的基本整数数据类型，即 Int。Int 的大小或位宽依赖于底层平台：在 32 位处理器中为 32 位，在 64 位处理器中为 64 位。

Swift 还提供指定宽度的整数数据类型，以及它们各自对应的无符号版本。这些数据类型的名称与 C 中的对应类型类似，即 Int8、Int16、Int32 和 Int64。UInt 是 Int 的无符号版本，并且和 Int 一样，其大小与底层平台匹配。若需指定宽度的无符号整数，可使用 UInt8、UInt16、UInt32 和 UInt64。

然而实际上并不需要过多考虑指定大小的整数数据类型。Swift 的类型推导系统可用来处理 Int 数据类型，该系统将在本章的后面讲述。一般而言，在使用 Swift 编程时不会用到指定位宽的整数，只有在极少数情况下才会用到它们。

2. 布尔类型

Swift 只提供了一个布尔类型 Bool，它可以有两个值：true 或 false。

Swift 的 Bool 类型与 C 和 Objective-C 中的类似类型有一点不同。C 实际上没有布尔类型，它将任何非 0 值视为 true 并将任何 0 值视为 false。在 C 中，int、char 甚至指针都依据它们各自的值而被视为 true 或 false。Objective-C 提供了 BOOL 布尔类型，该类型有两个值 YES 和 NO，但在 Objective-C 中，BOOL 实际上只是一个无符号字符，而 YES 和 NO 分别只是值 1 和值 0。Objective-C 遵循和 C 一样的真值规则，所以 Objective-C 中的所有非 0 值在布尔语境(如 if 语句)中都为 true，而所有 0 值都为 false，甚至对于指针也是如此处理。

Swift 废除了这些规则。除了 true 和 false 外任何值都不能赋给 Swift 的 Bool 类型，而且只有 Bool 类型可用在布尔语境中，如 if 语句。比较运算符，如==和!=，它们的运算结果为 Bool 类型。这种改变为 Swift 程序增加了类型安全级别，确保了在布尔语境中只能使用布尔值，除此之外不能再使用其他类型的值。

3. 使用字符类型

Swift 中有两种基于字符的数据类型：String 和 Character。Character 表示一个 Unicode 字符。String 表示 0 个或多个 Unicode 字符的集合。这两种类型都用双引号标记：

```
let c: Character = "a"
let s: String = "apples"
```

尽管 Swift 的 String 和 Character 类型从概念上讲类似于 C 和 Objective-C 中的字符串和字符类型，但实际上它们之间大不相同。Character 和 String 类型都可被视为对象并且具有相关的方法，这一点和 Swift 的其他类型一样。更重要的是，它们被设计用来处理兼容 Unicode 的文本。反之，C 字符串从本质上讲和字节值集合没有什么不同，而 String 字符串可被视为和编码无关的 Unicode 代码点的抽象。

这些不同之处存在某些重要的意义。一个主要问题是 Swift 如何处理基本多文种平面(Basic Multilingual Plane，BMP)之外的字符。这其中包括表情字符，它最初出现在移动消息传递中，并且其用途正在变得越来越广泛。Cocoa 的 NSString 类假设所有的 Unicode 字符都可用一个 16 位整数表示(具体为 unichar，它是 unsigned short 整数的类型别名)。对于 BMP 中的所有字符来说，这都是正确的，它们都可用 UTF-16 编码表示为一个 16 位整数(UTF-16 是 Unicode 文本的一种编码方式，其他比较流行的编码方式还有 UTF-8)，但并不是所有 Unicode 代码点都可用 16 位表示，表情字符就是其中之一。

因此，如果 NSString 对象包含基本多文种平面之外的 Unicode 代码点，它将不能返回直观的长度值，并且也不能直观地枚举它包含的字符。实际上，length 方法将返回用于编码字符的 unichar 的个数。请考虑下面这个简单的 Objective-C 程序：

```
NSString *s = @"\U0001F30D";
NSLog(@"s is %@", s);
NSLog(@"[s length] is %lu", [s length]);
NSLog(@"[s characterAtIndex:0] is %c", [s characterAtIndex:0]);
NSLog(@"[s characterAtIndex:1] is %c", [s characterAtIndex:1]);
```

该程序内的字符串只包含一个字符，即地球表情字符或🌍。然而程序的第 3 行输出的字符串长度将为“2”，因为这是 NSString 编码🌍所用的 unichar 的数量。第 3 行和第 5 行代码还会输出奇怪的字符，这是因为它们输出的是用来编码🌍字符的两个字节中的某个字节。

Swift 没有此问题。一个等价的 Swift 程序可以正确报告该字符串的长度为 1。它还能正确输出索引 0 处的字符，如果尝试输出索引 1 处的字符，将导致程序崩溃(该字符不存在)。等价的 Swift 程序如下所示：

```
let s = "🌍"
print("s is \(s)")
print("s.length is \(countElements(s))")
print("s[0] is \(s[advance(s.startIndex, 0)])")
print("s[1] is \(s[advance(s.startIndex, 1)])")
```

在程序中处理 Unicode 文本正变得越来越重要。相对于 NSString 来说，Swift 能正确处理 Unicode 的事实代表了一次巨大的成功。

String 可使用+运算符连接：

```
let s1 = "hello, "
let s2 = "world"
```

```
let s3 = s1 + s2
// s3 is equal to "hello, world"
```

另外还可以使用==运算符来比较两个字符串是否相等：

```
let s1 = "a string"
let s2 = "a string"
let areEqual = s1 == s2
// areEqual is true
```

理所当然，可以使用!=运算符判断两个字符串是否不等：

```
let s1 = "a string"
let s2 = "a string"
let areNotEqual = s1 != s2
// areNotEqual is false
```

注意：在本章的后面将更详细地讨论如+、==和!=这样的运算符。

Swift 还避免了 Cocoa 中的对可修改和不可修改字符串的区分，如 NSString 和 NSMutableString 类型的区分。Swift 只有一个字符串类型，它使用 Swift 的常量和变量功能来声明一个实例是可变的还是不可变的：

```
let s1 = "an immutable string"
var s2 = "a mutable string"
s2 += " can have a string added to it"
```

不可变字符串，从另一方面讲，不能把字符串附加到不可变字符串。如果尝试附加字符串到不可变字符串，Swift 编译器将报告错误。

使用字符串插值(string interpolation)还可以从其他数据类型建立新字符串。在建立字符串字面量时，\()结构中引用的变量将被转换为字符串：

```
let n = 100
let s = "n is equal to \(n)"
// s is "n is equal to 100"
```

字符串中还可以插入表达式：

```
let n = 5
let s = "n is equal to \(n * 2)"
// s is "n is equal to 10"
```

4. 使用数组

数组是 Swift 语言的一个主要组成部分。它们和 Foundation 框架中的 NSArray 和 NSMutableArray 类相似，但 Swift 可在语法层面方便地建立和操作数组数据类型。Swift

的 Array 类型和 Objective-C 的 NSArray 也有些不同。最重要的是：Swift 的 Array 只能包含类型相同的值，而 NSArray 可包含任何类实例。与 String 一样，Swift 数组的可变和不可变实例都使用 Array 类型，这取决于声明变量时是使用 var 还是 let 来区分它们。

Swift 还提供了初始化数组的语法，可通过在中括号中写入数组元素来初始化数组，如下所示：

```
let shoppingList = ["bananas", "bread", "milk"]
```

数组元素也可以通过中括号访问：

```
let firstItem = shoppingList[0]
// firstItem is "bananas"
```

如果数组使用 let 关键字声明，则它是不可改变的，不能在该数组上添加或删除元素。另一方面，如果数组使用 var 关键字声明，则可以随意在其上添加或删除元素，以及将元素存储到数组指定索引处。例如，在下面的代码中，数组的第二个元素将从"bread"变为"cookies"：

```
var shoppingList = ["bananas", "bread", "milk"]
shoppingList[1] = "cookies"
```

另外，Swift 还可以同时改变数组一个范围内的所有元素：

```
var shoppingList = ["candy", "bananas", "bread", "milk", "cookies"]
shoppingList[1...3] = ["ice cream", "fudge", "pie"]
```

上述代码运行后，shoppingList 将包含["candy","ice cream","fudge","pie","cookies"]。

注意：范围运算符将在本章后面讨论。

此外还可以向数组附加元素：

```
var a = ["one", "two", "three"]
a += ["four"]
print(a)   // Will print ["one", "two", "three", "four"]
```

要尽量使用不可变数组，除非确实需要修改数组的内容。

5. 使用字典

字典也是 Swift 的基础数据类型。它们扮演与 Foundation 框架中的 NSDictionary 和 NSMutableDictionary 同样的角色，只是在实现上有些不同。和数组一样，Swift 还在语法层面上支持创建和操作字典实例。

类似 Swift 数组，但与 Cocoa 的 NSDictionary 不同，Swift 字典的键必须使用同样的类

型，并且它的值也必须使用同样的类型。另外，它的键还必须是可散列的(hashable)。和数组一样，对于可变和不可变字典，Swift 也不提供单独的类型。其中可变字典使用 let 关键字声明，不可变字典使用 var 声明。

字典初始化的语法和数组类似。从语法上看，字典像是一个包含在中括号中的键-值对列表；键和值之间使用冒号分隔，如下所示：

```
let colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
```

单独的字典键值可通过将键放入变量名后面的中括号中来访问：

```
let colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
let redCode = colorCodes["red"]
```

如果字典是可变的，还可以使用用来访问元素的中括号语法来修改该元素：

```
var colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
colorCodes["blue"] = "000099"
```

6. 使用元组

Swift 具有一个新的基础数据类型：元组。元组将多个值组合到一个单一复合类型中。它类似于数组，但和数组不同的是，元组中的元素类型可以不同。例如，可使用下面的元组来表示 HTTP 状态码：

```
let status = (404, "Not Found")
```

在前面的例子中，status 由两个值组成，分别是 404 和"Not Found"。这两个值的类型不同：404 是 Int 类型，"Not Found"是 String 类型。

元组可分解为构成它的单独的元素：

```
let status = (404, "Not Found")
let (code, message) = status
// code equals 404
// message equals "Not Found"
```

在分解语句中可使用下划线(_)来忽略一个或多个元素：

```
let status = (404, "Not Found")
let (code, _) = status
```

注意：在分解元组时，如果想要修改分解出的元素，可用 var 关键字替代 let 关键字。

元组特别适用于在函数或方法中返回多个值的情况。函数或方法只能返回单个值，由于元组就是单个值(虽然它由多个值组成)，因此可用它绕过此限制。

1.3.2　使用类型注解

Swift 是一门类型安全的语言，这意味着每个变量都有一个类型，并且编译器会确保赋给变量的值的类型和传递给函数或方法的值或变量的类型是正确的。这样就确保了代码总是在使用期望范围内的值，并且代码不会尝试在一个值上执行该值不支持的操作(如调用一个方法)。

类型注解是程序员用来告知编译器变量的类型的方法。用 C 或 Objective-C 编过程序的人会对注解比较熟悉。请看下面的代码：

```
NSString *s = @"this is a string";
int x = 10;
```

在上面的例子中，s 和 x 都是变量。在每个变量名的前面都有一个类型注解：注解 s 的类型是 NSString，注解 x 的类型是 int。这些注解是必需的，它们告知编译器期望 s 和 x 保存的值的类型。因此，如果将其他类型的值赋给这些变量，编译器将报告错误，如下所示：

```
int x = 10;
x = @"this is a string";
```

类型通过确保使用的值范围是期望的范围来辅助保证代码的正确性。如果函数期望一个整数，则不能错误地向它传递一个字符串。

Swift 可以注解变量，尽管其语法与 C 和 Objective-C 有些不同。在 Swift 中，可通过在变量名后写入冒号和类型名来注解类型。在下面的例子中，ch 被声明为 Character 类型：

```
let ch: Character = "!"
```

从本质上讲，上面的声明和下面 C 中的声明一样：

```
char ch = '!'
```

类型注解还可以用在函数和方法中以注解它们的参数类型：

```
func multiply(x: Int, y: Int) -> Int {
   return x * y;
}
```

在此处，参数 x 和 y 都被声明为 Int 类型。

注意：在本章 1.7 节“使用函数”中将详细介绍函数。

当然，为每个变量注解类型将会很快变得难以处理，并且确保所有变量都被正确注解变成了程序员的负担。C 和 Objective-C 这样的语言还允许将值从一种类型转换为另一种类型，而这会进一步削弱这些语言提供的本就脆弱的类型安全机制。总之，显式的类型注解

既烦琐又容易出错。幸运的是，Swift 提供了解决该问题的方案：类型推导引擎。

1.3.3　使用类型推导简化类型注解

很多程序员喜欢 Python 和 JavaScript 这样的动态类型语言，因为这些语言不需要程序员注解和管理每个变量的类型。在大多数动态类型的语言中，变量可以是任何类型，从而要么类型注解是可选的，要么根本不允许使用类型注解。Objective-C 采用了一种混合方案：需要类型注解，但是任何指向 Objective-C 类(包括所有派生自 NSObject 的类，但不包括 int、float 这样的基础类型)实例的变量都可被简单地声明为 id 类型，从而使其可以指向任何类型的 Objective-C 实例。即使是在使用更严格的注解时，Objective-C 编译器也不会严格保证 Objective-C 变量的类型。

尽管动态类型语言经常被认为比静态类型语言更好用，但缺乏严格的类型安全机制意味着程序的正确性无法保证，并且它们通常更难以推理，尤其是处理第三方代码或多年前写的代码时。

然而，静态类型语言并不能保证比动态类型语言更安全。C 具有静态类型并且需要类型注解，但使用类型转换、void 指针和类似的语言支持的欺骗机制可很容易绕过 C 的有限的类型安全机制。

Swift 采用了两全其美的方法：它的编译器严格保证每个变量类型是正确的，而它的类型推导机制可避免程序员手工注解每个变量的类型。

类型推导机制使得编译器可根据变量在代码中如何使用来推断出该变量的类型。实际上，这意味着大多数变量都不用添加类型注解，而是应该让编译器来推断出这些变量的类型。即使没有注解，Swift 的类型系统也仍然会保证程序的类型安全。

请思考下面的代码：

```
let s = "string"
let isEmpty = s.isEmpty
```

上述代码中的所有常量都没有注解类型，但是 Swift 能够推断出 s 的类型为 String，而 isEmpty 的类型为 Bool。如果尝试将这些常量之一传递给需要 Int 参数的函数，编译器将报告错误，代码如下：

```
func max(a: Int, b: Int) -> Int {
   return a > b ? a : b
}
max(s, isEmpty)
```

上面的代码将产生错误信息“'String'不能转换为'int'”，这说明 Swift 知道 s 和 isEmpty 不是 Int 类型。

尽管 Swift 的类型推导系统大大减少了代码中手工注解的数量，但有时仍然需要声明类型。当编写函数时，就必须声明所有参数的类型以及函数返回值的类型。

另外，当变量的类型有歧义时也需要为其注解类型。例如，String 和 Character 都使用

双引号括起来的文本表示。那么 Swift 编译器会把双引号括起来的单个字符推导为 String 而不是 Character。下面的代码将产生编译错误，因为 c 的类型是 String：

```
func cId(ch: Character) -> Character { return ch; }
let c = "X"
cId(c)
```

如果希望 c 是 Character 类型，则需要将它显式声明为一个字符。下面的代码不会出错：

```
func cId(ch: Character) -> Character { return ch; }
let c: Character = "X"
cId(c)
```

然而，这种出现歧义的情况在 Swift 中是很罕见的。通常编译器都能够正确推导出正确的变量类型，从而不需要经常进行显式类型注解。

类型理论

在计算机科学中，类型推导是一个巨大课题，在知识和研究领域它被称为类型理论。类型理论的一个重要元素是 Hindley-Milner 类型系统，它是很多类型推导引擎的基础。该系统中使用的类型推导算法主要用来描述简单类型λ演算的类型，该计算方法是很多现代编程语言的基础。

对类型理论的全面讨论远远超出了本书的范围，有兴趣的读者可以在网上查找和该主题有关的资料。

1.3.4　使用类型别名简化代码

在某些情况下，为类型推导使用更特定的术语可让代码更清晰。假设有如下计算速度的函数：

```
func speed(distance: Double, time: Double) -> Double {
   return distance / time
}
```

当然这个函数可正常工作，但 distance 是什么？它的单位是英尺还是米？另外 time 是什么？它的单位是秒还是分钟或小时？这个函数不能清晰地表达出它期望的类型。如果像下面这样编写该函数将更好一些：

```
func speed(distance: Feet, time: Seconds) -> FeetPerSecond {
   return distance / time
}
```

当然，Feet、Seconds 和 FeetPerSecond 不是 Swift 的数据类型。然而，Swift 具有能提供更富有表现力的类型名称的机制：类型别名。对于已经熟悉 C 和 Objective-C 中的 typedef 的人来说，类型别名具有同样的功能：它们允许为现有类型指定别名。

使用类型别名可以将前面的代码修改为可实际编译的代码：

```
typealias Feet = Double
typealias Seconds = Double
typealias FeetPerSecond = Double

func speed(distance: Feet, time: Seconds) -> FeetPerSecond {
   return distance / time
}
```

编写函数时使用类型别名可使其代码更清晰易读。类型别名是一项强大的特性，可在自己的代码中愉悦地使用它。

1.4　使用运算符

Swift 包含大量用于处理数据类型的内建运算符。它们大部分是数学运算符，用来处理数值数据类型(Int、Float 和 Double)，另外 Swift 还提供一个字符串连接运算符以及逻辑和比较运算符。Swift 还引入了表示和处理范围值的运算符。这些运算符中的大多数，除了范围运算符，也是 C 和 Objective-C 的一部分，所以读者肯定非常熟悉它们，它们的表现也和 C 和 Objective-C 中的对应运算符非常接近。

1.4.1　使用基本运算符

Swift 支持 4 个标准数学运算符：

- 加(+)
- 减(−)
- 乘(*)
- 除(/)

这些运算符的行为完全与 C 和 Objective-C 中的一样，除了有一点不同：它们不允许溢出。也就是说，如果一个变量是其类型的最大值(例如 Int 的最大值)，并且想要在其上加一个值，则会发生错误。如果运算符的溢出机制和 C 一致，则必须使用这些运算符的溢出变体。溢出运算符将在下一节中讨论。

和 C 中一样，Int 除法运算符(/)的结果为第二个 Int 除第一个 Int 的整数最大值：

```
let m = 11 / 5
// m is equal to 2
```

加法运算符(+)还支持字符串。将两个字符串"加"在一起会连接它们：

```
lets1 = "hello"
let s2 = ", world"
let s3 = s1 + s2
// s3 is now "hello, world"
```

Swift 还有一个求余运算符%。它返回除法运算的余数：

```
let rem = 11 % 5
```

```
// rem is equal to 1
```

求余运算符还支持 Float 和 Double，这一点和 C 不同，在 C 中只允许求余运算符计算 Int：

```
let rem = 5.0 % 2.3
// rem is equal to 0.4
```

注意：在 C 和 Objective-C 中，%运算符通常被称为求模运算符。负数的求余运算结果虽然在数学上是正确的，但许多程序员在直觉上有疑惑。Swift 在该运算符计算负数时具有更符合直觉的行为，从而将%称为求余运算符。当操作数为正数时，求模运算和求余运算会产生完全相同的结果。

最后，Swift 还支持递增(++)和递减(−−)运算符，这些运算符将值增加或减少 1：

```
var i = 10
i++
// i is now equal to 11
```

这些运算符都提供前缀和后缀两个变种。前缀形式返回运算后的变量值，而后缀形式返回运算前的变量值。参见下面的代码例子：

```
var i = 10
var j = i++
// j is now equal to 10, and i is equal to 11
var k = ++i
// i and k are both equal to 12
```

1.4.2　使用复合赋值运算符

使用复合赋值运算符可以将数学运算符和赋值合并在一起来完成：

- 加法赋值(+=)
- 减法赋值(−=)
- 乘法赋值(*=)
- 除法赋值(/=)
- 求余赋值(%=)

这些运算符在单个表达式中执行和其相关的运算并将结果设置给一个变量：

```
var x = 20
x += 10
// x is now equal to 30
```

1.4.3　使用溢出运算符

Swift 的数学运算符不允许值溢出。在其他如 C 这样的语言中，如果变量为该变量类

型的最大值并且又为它加上了另一个值，该变量将绕回为其变量类型的最小值。而在 Swift 中这会发生运行时错误。可是在某些情况下可能需要这样的溢出行为。所以 Swift 提供了数学运算符的溢出变体：

- 溢出加(&+)
- 溢出减(&−)
- 溢出乘(&*)
- 溢出除(&/)
- 溢出求余(&%)

这些运算符的用法和基本的数学运算符完全一样。唯一的不同是它们允许溢出(或下溢出)。例如：

```
var num1: Int8 = 100;
var num2: Int8 = num1 &+ 100;
print(num2);  // Prints -56
```

1.4.4　使用范围运算符

在 Swift 中可使用范围运算符很容易地表示范围。Swift 提供两种表示范围的方法：闭范围运算符(…　)和开范围运算符(..<)。它们都接受整数操作数并产生这两个整数范围内的一系列的值。闭范围运算符的两个操作数包含在范围内，而开范围运算符只在其范围内包含第一个操作数的值，但不包含第二个操作数的值。这两个运算符经常用作循环计数器：

```
for i in 1...5 {
  // i will contain the values 1, 2, 3, 4, 5
  print("\(i)")
}
for i in 1..<5) {
  // i will contain the values 1, 2, 3, 4
  print("\(i)")
}
```

1.4.5　使用逻辑运算符

Swift 提供三个逻辑运算符：逻辑非(!)、逻辑与(&&)和逻辑或(||)。!是一个前缀运算符，用于求一个变量的相反值：

```
let b1 = true
let b2 = !b1
// b2 is false
```

&&和||都需要两个操作数。如果两个操作数都为 true，则&&返回 true；如果两个操作数中至少有一个为 true，则||返回 true。

```
let b1 = true
let b2 = false
let b3 = b1 || b2
```

```
// b3 is true
let b4 = b1 && b2
// b4 is false
```

Swift 还有一个继承自 C 的三元条件运算符。该运算符需要三个参数：

- 一个条件
- 一个条件为 true 时返回的值
- 一个条件为 false 时返回的值

本质上它是内联的 if 语句，如下所示：

```
let flag = true
let res = if flag ? 1 : 0
// res is equal to 1
```

与 C 和 Objective-C 不同的是，这些逻辑运算符只使用 Bool 值，不能将它们用于非布尔值，如 Int。

1.4.6　使用比较运算符

在 Swift 中存在 6 个比较运算符：

- 等于(==)
- 不等于(!=)
- 大于(>)
- 小于(<)
- 大于等于(>=)
- 小于等于(<=)

和逻辑运算符一样，比较运算符只返回 Bool 类型，尽管它们的操作数可以为大多数类型，包括数值类型甚至可为类(如果类定义了自定义运算符的话)。每个比较运算符都需要两个操作数，并返回运算结果。它们的行为与 C 和 Objective-C 中的对应运算符一样。

参见下面的例子：

```
let a = 10
let b = 20
if (a > b) {
  print("a is greater than b")
} else if (a == b) {
  print("a is equal to b")
} else if (a < b) {
  print("a is less than b")
}
```

注意：在 1.5.1 节“使用条件语句”中将详细讨论条件语句。

1.4.7　使用自定义运算符

在 Swift 中还可以自己定义运算符。在第 8 章中将详细讨论自定义运算符。

1.5　使用控制流进行判断

如果程序设计语言无法基于确定条件进行判断，则在大多数情况下其不是一门好的语言。Swift 提供所有人们都熟悉的控制流语句来让程序在其执行期间进行判断。

1.5.1　使用条件语句

Swift 具有两个基本的条件语句：if 语句和 switch 语句。如果条件为真，则 if 语句执行其语句体：

```
let flag = true
if flag {
    print("This statement is executed")
}
```

if 语句还可以有 else 块，如果条件为 false，则执行该块：

```
let flag = false
if flag {
    print("This statement is not executed")
} else {
    print("This statement is executed")
}
```

if 语句可链接在一起。如果条件为 true，则第一个块会被执行：

```
let x = 10
if x < 5 {
  print("This statement is not executed")
} else if x < 10 {
  print("This statement is not executed, either")
} else if x < 20 {
  print("This statement is executed")
} else {
  print("This statement is not executed")
}
```

和基本的 if 语句一样，链接在一起的 if 语句也不一定需要 else 块。

Swift 还有 switch 语句。switch 语句计算一个值并将它和多种情况进行比较。首个比较结果为 true 的 case 语句将会被执行。switch 语句本质上相当于链接在一起的 if 语句，但其形式更紧凑、更易读。

switch 语句的最基本形式和 C 非常相似，只是语法上有一些不同：

```
let n = 20
switch n {
case 0:
  print("This statement is not executed")
case 10:
  print("Neither is this statement")
case 20:
  print("But this one is!")
default:
  print("And this one isn't")
}
```

除了语法外，Swift 的 switch 语句有几个关键方面和 C 不同。首先，Swift 的 switch 语句操作的值并不一定是整数，甚至不一定是数值类型；字符串、元组、枚举(本章后面讨论)、可选类型(在第 8 章中讨论)甚至是自定义类都可作为 switch 语句操作的值。

每个 switch 的 case 语句都可以有多个匹配，匹配用逗号分隔：

```
let ch: Character = "a"
switch ch {
case "a", "e", "i", "o", "u":
  print("\(ch) is a vowel")
case "y":
  print("\(ch) may be a vowel or a consonant")
default:
  print("\(ch) is a consonant")
}
```

switch 的 case 语句必须有一个语句体。然而可以使用 break 语句代替 case 语句的可执行代码。如果该 case 语句被匹配，则因为此处没有可执行的代码，所以将跳出 switch 语句去执行：

```
let n = 10
switch n {
case 10:
  break
default:
  print("\(n) is not 10")
}
```

在 switch 语句中还可以匹配范围：

```
let n = 23
switch n {
case 0...10:
  print("\(n) is between 0 and 10")
case 11...100:
  print("\(n) is between 11 and 100")
case 101...1000:
  print("\(n) is between 101 and 1000")
```

```
default:
  print("\(n) is a big number")
}
```

甚至可以匹配元组：

```
let color = (255, 0, 0)
switch color {
case (0, 0, 0):
  print("\(color) is black")
case (255, 255, 255):
  print("\(color) is white")
case (255, 0, 0):
  print("\(color) is red")
case (0, 255, 0):
  print("\(color) is green")
case (0, 0, 255):
  print("\(color) is blue")
default:
  print("\(color) is a mixture of primary colors")
}
```

在使用元组或其他如枚举、自定义类这样的复合数据类型时，如能够将变量绑定到这些数据类型中的元素，将是非常有用的。Swift 的值绑定技术可用来将变量和常量关联到另外的变量：

```
let color = (255, 0, 0)
switch color {
case (let red, 0, 0):
  print("\(color) contains \(red) red")
case (0, let green, 0):
  print("\(color) contains \(green) green")
case (0, 0, let blue):
  print("\(color) contains \(blue) blue")
default:
  print("\(color) is white")
}
```

第一个 case 语句(let red, 0, 0)匹配任何第二个和第三个元素为 0 的元组，而第一个元素可以匹配任意值并被绑定到 red 常量。

在绑定中还可以使用变量，只需要将 let 关键字替换为 var 关键字即可。如果绑定的是一个变量，则在 case 语句体中修改该变量，就像任何其他变量一样。

case 语句还可以使用 where 子句来匹配值之外的附加约束：

```
let color = (255, 0, 0)
switch color {
case let (r, g, b) where r == g && g == b:
  print("\(color) has the same value for red, green, and blue")
default:
```

```
  print("\(color)'s RGB values vary")
}
```

第一个 case 语句分解元组并将常量 r、g、b 快速绑定到颜色分量。

Swift 的 switch 语句不会隐式进入到 switch 块中的下一个语句。一旦 case 语句匹配并且其语句体执行后，执行会跳出 switch 语句。使用 fallthrough 关键字可显式进入另一个 case 语句：

```
let ch: Character = "y"
print("\(ch) is a ")
switch ch {
case "y":
  print("consonant, and also a ")
  fallthrough
case "a", "e", "i", "o", "u":
 print("vowel")
default:
  print("consonant")
}
```

最后，switch 语句的所有 case 语句加起来必须是完整的；也就是说，条件值的整个范围都必须被考虑进去，否则将发生编译器错误。default 语句可用来覆盖值的整个范围，如果默认值没有需要执行的动作，则应该使用 break 语句。

1.5.2　使用循环

Swift 提供两种循环：用来枚举值集合的 for-in 循环以及更常见的 for 条件循环，该循环会一直遍历值，直到达到某个确定条件为止。

对 C 程序员来说可能不熟悉 for-in 循环，甚至 Objective-C 也只是在 2006 年发布的 Objective-C 2.0 中提供了该特性。for-in 循环在 Objective-C 文档中也被称为“快速枚举”，它遍历集合类型中的每个元素，这些集合类型包括数组、字典，甚至 Range 对象。在最基本的情况下，for-in 循环可以利用范围运算符来遍历单调递增的值的集合：

```
for i in 1...5 {
  // iterates over the values 1, 2, 3, 4, 5
}
```

除了范围外，还可以使用容器类型(比如数组或字典)来遍历元素的集合：

```
let fruits = ["lemon", "pear", "watermelon", "apple", "breadfruit"]
for fruit in fruits {
  // iterates over the values lemon, pear, watermelon, apple, breadfruit
}
```

for-in 循环可以遍历字典。遍历字典会返回包含各个键值的二元组。这个二元组可分解为其组成部分：

```
let nums = [0: "zero", 10: "ten", 100: "one hundred"]
```

```
for (num, word) in nums {
  // num will be 0, 10, 100
  // word will be zero, ten, one hundred
}
```

Swift 的基本容器类型，如数组、字典和范围，都可以在 for-in 循环中被遍历。其他适配了 Sequence 协议的对象也可以在 for-in 循环中被遍历。在第 8 章中将深入学习如何实现该协议。

Swift 也提供了 for 条件循环，有 C 和 Objective-C 背景的人更熟悉这种循环。该循环本质上等同于 C 的 for 循环，尽管语法有点不同：

```
for var i = 0; i < 5; ++i {
  // loops over 0, 1, 2, 3, 4
}
```

在 for 条件循环中，变量只在循环作用域内有效；在前面的例子中，i 不能在循环外被访问。如果需要在循环外使用 i，则必须在循环之前声明它。下面的代码给出了相关例子：

```
var i = 0;
for i = 0; i < 5; ++i {
  // loop body
}
```

如果不想在函数中使用 for 循环的变量，可使用下划线(_)作为变量名：

```
for _ in 1...10 {
  print("\n")
}
```

for 条件循环有一点多余，因为使用 for-in 循环结合范围可以编写出完全相同的构造，所以在实践中不会经常见到它们。然而，如果决定使用它们，则它们也是可用的。

注意：某些编程语言，比如 Python，只有 for-in 循环。它们完全放弃了 for 条件循环，它们依赖范围来模拟 for 条件循环。某些语言的设计者认为结合范围的 for-in 循环更安全，因为这样避免了 for 条件循环中的边界条件问题(通常指的是差一(off-by-one)错误)。Swift 仍然支持 for 条件循环，但是在代码中使用 for-in 循环将更方便且更不易出错。

for-in 循环和 for 条件循环被视为确定循环(determinate loops)：它们具有定义明确的结束条件。Swift 还提供两种形式的非确定循环(indeterminate loops)，或者说没有明确定义结束条件的循环：while 循环和 do-while 循环。

while 循环的语句体会被简单地执行直到其条件变为 false:

```
var flag = true
var i = 1
```

```
while flag {
  print("i is \(i++)")
  if i > 10 {
    flag = false
  }
}
```

do-while 循环也是类似的，不同之处是其语句体总是至少会被执行一次，以及条件的检查总是在循环的结束处而不是开始处：

```
var flag = false
var i = 1
do {
  print("i is \(i++)")
  flag = i <= 10
} while flag
```

Swift 提供了几个循环变体，虽然使用简单的 while 循环就可以模仿它们，然而使用它们可以让代码更清晰。例如，当遍历已知范围或元素集合时，for-in 循环能比基本的 while 循环表达更清晰的意图。

1.5.3　控制转移语句

循环可以包含将代码的执行控制转移给代码的另外部分的语句。这些语句与它们在 C 和 Objective-C 中的行为类似，并可用于所有类型的循环。

continue 语句会导致执行跳回循环的开始处。下面的代码示例只输出奇数：

```
for i in 1...100 {
  if i % 2 == 0 {
    continue
  }
  print("\(i)")
}
```

break 语句会导致循环立即终止。以下循环将一直执行直到用户输入 q：

```
while true {
  print("Enter 'q' to end: ")
  let input = getUserInput()
  if input == "q" {
    break
  }
}
```

break 和 continue 语句都是跳出最内层的循环。然而通过给循环打标签可实现跳出更外层的循环：

```
let data = [[3, 9, 44], [52, 78, 6], [22, 91, 35]]
let searchFor = 78
var foundVal = false
```

```
outer: for ints in data {
  inner: for val in ints {
    if val == searchFor {
      foundVal = true
      break outer
    }
  }
}
if foundVal {
  print("Found \(searchFor) in \(data)")
} else {
  print("Could not find \(searchFor) in \(data)")
}
```

1.6　使用枚举组织类型

枚举可将相关类型组织在一起。在 C 和 Objective-C 中，枚举被设计成与 enum 关键字一起使用，它无非是组织在一起的常量。在将枚举类型视为专属于它自己的“类型”这方面，编译器只提供了很少的支持。另一方面，Swift 枚举有专属于它自己的类型，从而具有如结构和类这样的类型的所有功能。

与 C 和 Objective-C 一样，Swift 枚举由 enum 关键字引入：

```
enum Direction {
  case North
  case South
  case East
  case West
}
```

枚举成员还可写在同一行，中间用逗号分隔：

```
enum Direction {
   case North, South, East, West
}
```

在 switch 语句中使用枚举很方便，其中的每个 case 语句都可处理枚举类型之一：

```
let dir = Direction.North
switch dir {
case .North:
  print("Heading to the North Pole")
case .South:
  print("Heading to the South Pole")
case .East:
  print("Heading to the Far East")
case .West:
  print("Heading to Europe")
}
```

注意：因为 Direction 枚举的成员只有 North、South、East 和 West，而且 switch 语句处理了所有这些情况，所以默认 case 语句将不再是必要的。

与类和结构一样，可将枚举类型成员视为构造函数并可向其关联值。被关联值类似于类和结构(尽管即使使用该特性，枚举也仍然不像类那样强大)中的实例变量(或属性)。可在建立枚举成员时指定关联值。例如此处是一种用枚举描述 JSON 的方式：

```
enum JSValue {
  case JSNumber(Double)
  case JSString(String)
  case JSBool(Bool)
  case JSArray([JSValue])
  case JSDictionary([String: JSValue])
  case JSNull
}
```

JSNumber 不仅是一个数据类型，它还关联了一个 Double 值；JSString 关联了一个 String 值，除了 JSNull(由其性质可知，它没有与之关联的唯一值)之外，其余依此类推。

和实例化类或结构一样，可以通过关联值的方式来建立枚举：

```
let val = JSValue.JSNumber(2.0)
```

在 switch 语句中可通过分解枚举值来获得其关联值，这就和使用元组一样：

```
let val = JSValue.JSNumber(2.0)
switch val {
case .JSNumber(let n):
  print("JSON number with value \(n)")
default:
  print("\(val) is not a JSON number")
}
```

与 C 和 Objective-C 相比，Swift 中的枚举类型非常强大。在很多方面，可按使用类或结构的方式使用枚举。在第 3 章将详细介绍枚举类型。

1.7　使用函数

Swift 函数是单独的代码块，可以调用它来将输入变换为输出，或只是执行如写入到文件或打印输出到屏幕这样的动作。从功能看它们与 C 和 Objective-C 中的函数一样，尽管它们的语法有本质上的区别。函数可处于源代码的顶层，也可被嵌套在其他函数中(C 和 Objective-C 不提供该特性)。函数还可以关联到类、结构和枚举，在此情况下其被称为方法，尽管声明方法的语法大体上是相同的(在第 4 章将详细学习类和结构)。

1.7.1　声明函数

每个函数都有一个名称，可供用来称呼这个函数。函数还可以接受参数。函数使用 func 关键字声明，之后为该函数的名称。名称后跟一对圆括号；参数可以在这对圆括号中声明。最后，如果函数返回一个值，则可以在返回箭头(->)后指定返回值类型；如果函数没有返回值，则可省略该部分(没有返回值的函数被称为 void 函数)。函数体在大括号中指定。此处是一个简单的函数定义，该函数名为 multiplyByTwo，其接受一个 Int 类型的参数 x 作为输入，并返回参数乘以 2 的结果：

```
func multiplyByTwo(x: Int) -> Int {
  return x * 2
}
```

函数的调用看起来与 C 和 Objective-C 一样：

```
let n = multiplyByTwo(2)
// n is equal to 4
```

如果函数接受多个参数，在参数之间使用逗号分隔：

```
func multiply(x: Int, y: Int) -> Int {
  return x * y
}
let n = multiply(2, 4)
// n is equal to 8
```

没有返回值的函数可以省略返回类型：

```
func printInt(x: Int) {
  print("x is \(x)")
}
printInt(10)
// will print "x is 10" to the console
```

函数只能返回一个值。然而通过将返回值封装在元组中可返回多个值：

```
func makeColor(red: Int, green: Int, blue: Int) -> (Int, Int, Int) {
  return (red, green, blue)
}
let color = makeColor(255, 12, 63)
let (red, green, blue) = color
```

1.7.2　指定参数名称

在函数中，可通过引用参数列表中指定的名称来使用该变量。当调用函数时，参数传入的顺序和它们在函数中声明的顺序一致，在调用函数时调用者不需要指定参数的名称。

有时可能需要强制调用者指定参数名称。这可以提高使用函数代码的可读性。例如在下面的代码中，可以在内部(internal)参数名称前指定外部(external)参数名称：

```
func makeColor(red r: Int, green g: Int, blue b: Int) -> (Int, Int, Int) {
  return (r, g, b)
}
```

在 makeColor 函数内部，参数为 r、g、b。而在函数外部，它们被指定为 red、green 和 blue。当调用带有命名参数的函数时，必须指定参数名：

```
let color = makeColor(red: 255, green: 14, blue: 78)
```

很多情况下都希望内部参数名和外部参数名是一致的。这可通过在参数名前加一个井号(#)来实现：

```
func makeColor(#red: Int, #green: Int, #blue: Int) -> (Int, Int, Int) {
  return (red, green, blue)
}
let color = makeColor(red: 255, green: 14, blue: 78)
```

1.7.3　定义默认参数

函数参数还可以指定默认值。如果在调用函数时没有为这样的参数指定值，将使用该参数的默认值。另外在调用该函数时也可以覆盖该参数的默认值：

```
func multiply(x: Int, by: Int = 2) -> Int {
  return x * by
}
let x = multiply(4)
// x is equal to 8
let y = multiply(4, by: 4)
// y is equal to 16
```

如果没有为默认参数指定外部名称，Swift 将使用内部名称作为外部名称。无论如何，在调用函数时都必须使用外部名称。

如果确实不想让调用者使用默认参数的外部名称，可使用下划线(_)作为该参数的外部名称：

```
func multiply(x: Int, _ by: Int = 2) -> Int {
  return x * by
}
let x = multiply(4, 4)
// x is equal to 16
```

然而，为默认参数指定外部名称会让代码更易读，这是最好的做法。

1.7.4　指定可变参数

函数可接受可变数量的参数。可变参数通过在一个参数类型名后放置三个点(…　)来指定：

```
func multiply(ns: Int...) -> Int {
```

```
  var product = 1
  for n in ns {
     product *= n
}
return product
```

可变参数被作为一个数组传入函数；在前面代码的 multiply 函数中，ns 参数的类型为[Int]或 Int 数组。调用该函数时使用逗号分隔的变量或值列表作为参数：

```
let n = multiply(4, 6, 10)
// n is equal to 240
```

函数只能有一个可变参数，并且必须在参数列表的最后指定。

1.7.5　指定常量、变量和输入-输出参数

就像使用 let 或 var 指定常量或变量一样，还可在函数参数列表中使用 let 或 var 来指定一个参数是常量还是变量。默认情况下，所有参数都是常量，所以不能在函数中改变它们的值。下面的代码是不被允许的：

```
func multiply(x: Int, y: Int) -> Int {
  x *= y
  return x
}
```

然而，如果使用 var 关键字来声明参数，则可在函数体中修改它：

```
func multiply(var x: Int, y: Int) -> Int {
  x *= y
  return x
}
```

不管参数是变量还是常量，参数的改变只能在函数体中看到。函数不能改变函数以外的变量的值：

```
let x = 10
let res = multiply(x, 2)
// res is equal to 20, but x is still equal to 10
```

有时可能需要函数能够影响其外部的变量。这种情况下，可将参数指定为 inout 参数：

```
func multiply(inout x: Int, y: Int) {
  x *= y
}
```

当调用带有 inout 参数的函数时，需要在 inout 参数前加上&符号前缀。如果用 C 或 Objective-C 编过程序，则会对该语法感到熟悉：inout 参数类似于 C 和 Objective-C 中的指针参数。在这两门语言中，&也用于获得变量的内存地址，其会返回该变量的指针。Swift 在很大程度上免除了指针的概念，但由于人们对此比较熟悉，所以仍采用了相似的语法。

尽管前面的 multiply 函数不返回值，但仍会改变传入的值：

```
var x = 10
multiply(&x, 2)
// x is now equal to 20
```

对于 inout 参数只能为其传递变量，传递常量或值将导致编译器错误。

1.7.6　函数类型

函数也是类型，就像 Swift 中的 Int、Float、Array 和任何其他类型一样。然而并不存在一个统一的 Function 类型，每个唯一的函数签名(函数参数类型和返回类型的组合)都表示一个唯一不同的类型。请看如下函数：

```
func multiply(x: Int, y: Int) -> Int {
  return x * y
}
```

此处的类型是一个“接受两个 Int 参数并返回一个 Int 的函数”，并使用类型签名(Int, Int) -> Int 表示。

如果函数没有参数，则其参数类型用一对空括号()指定。尽管其被称为 void，但实际上它是一个没有元素的元组。同样，没有返回值的函数其返回类型也是()。因此如下函数的类型签名为( ) –> ( )：

```
func printDash() {
  print("-")
}
```

与其他数据类型一样，可声明引用函数的常量或变量：

```
func multiply(x: Int, y: Int) -> Int {
  return x * y
}
let m: (Int, Int) -> Int = multiply
let n = m(2, 4)
```

如前面例子中的代码所示，可以像调用函数那样调用引用函数的变量或常量。

注意：和其他变量一样，只有不立即初始化指向函数的变量时，才需要为该变量指定类型，因为 Swift 可以推导该变量的类型，就像它可以推导如 Int、Float 这样的其他数据类型一样。

因为函数是对象，就像 Swift 中任何其他数据类型一样，所以可以将函数作为其他函数的参数或者从一个函数中返回另一个函数。例如，下面的 map 函数接受一个函数和一个 Int 数组，然后将每个数组元素都乘以 2：

```
func multiplyByTwo(x: Int) -> Int {
  return x * 2
}
func map(fn: (Int) -> Int, ns: [Int]) -> [Int] {
  var res: [Int] = []
  for n in ns {
    let n1 = fn(n)
    res.append(n1)
  }
  return res
}
let nums = map(multiplyByTwo, [1, 2, 3])
// nums is now [2, 4, 6]
```

一个函数还可以返回另一个函数：

```
func subtract(x: Int, y: Int) -> Int {
  return x - y
}
func add(x: Int, y: Int) -> Int {
  return x + y
}
func addOrSubtract(flag: Bool) -> (Int, Int) -> Int {
  if flag {
    return add
  } else {
    return subtract
  }
}
let fn = addOrSubtract(false)
let res = fn(4, 2)
// res is equal to 2
```

1.7.7　使用闭包

在和 Swift 的闭包一起使用时，函数的将其他函数作为参数的功能和返回函数的功能将更加强大。和函数一样，闭包也是一块代码，可以被作为对象传递。它们之所以被这样命名，是因为它们封装了声明它们的作用域，这意味着它们可以引用它们的作用域内当前存在的变量和常量。然而和函数不同的是，闭包不需要被命名和声明；在需要它们时可直接定义。

事实上，Swift 函数只是闭包的特例，这意味着它们可被该语言(及其配套基础设施，如 Swift 编译器)以同样的方式对待。

不同于命名函数，闭包通常用在闭包表达式中，它是一小段用来定义匿名闭包的代码。闭包表达式写在大括号中。和函数一样，它们可以接受在括号中指定的参数列表以及可以在返回箭头(->)后指定返回值类型。闭包的语句体在 in 关键字之后开始。

之前定义的 addOrSubtract 函数可被重写为使用闭包而非命名函数：

```
func addOrSubtract(flag: Bool) -> (Int, Int) -> Int {
  if flag {
    return { (x: Int, y: Int) -> Int in return x + y }
  } else {
    return { (x: Int, y: Int) -> Int in return x - y }
  }
}

let fn = addOrSubtract(false)
let res = fn(4, 2)
// res is equal to 2
```

很多情况下，闭包的参数类型可从上下文中推导出来，从而可从其声明中省略。此外，如果闭包只有一条语句，则 return 关键字也可被省略。这让闭包的使用更加简洁，而这种简洁的方式非常重要，因为它们常在 Swift 编程中使用。之前的 map 函数可按如下方式编写和使用：

```
func map(fn: (Int) -> Int, ns: [Int]) -> [Int] {
  var res: [Int] = []
  for n in ns {
    let n1 = fn(n)
    res.append(n1)
  }
  return res
}

let nums = map({ x in x * 2 }, [1, 2, 3,])
// nums is now [2, 4, 6]
```

为让代码的编写更方便，闭包还可以在其语句体内使用简写参数名。该参数名的前缀是一个美元符号($)，参数名是从 0 开始的数字。第一个参数为$0，第二个为$1，等等。之前声明的 map 函数可以按如下方式调用：

```
let nums = map({ $0 * 2 }, [1, 2, 3])
```

Swift 为最后一个传递给函数的闭包提供附加语法：这种情况下，该闭包可在函数调用中的括号外部指定。为能使用这种优雅的语法，如果函数的参数中只有一个函数类型，则会将其指定为最后一个参数。可重写 map 函数，将函数参数作为最后一个参数：

```
func map(ns: [Int], fn: (Int) -> Int) -> [Int] {
  var res: [Int] = []
  for n in ns {
      let n1 = fn(n)
      res.append(n1)
    }
    return res
}
```

对 map 函数的调用可写为如下形式：

```
let nums = map([1, 2, 3]) { $0 * 2 }
```

闭包是 Swift 的难以置信的强大功能的一部分。它类似于 Objective-C 中的块代码，但比其更灵活，并且更重要，它比 Objective-C 的块代码的语法更好、更容易记忆。如何发挥闭包的全部功能和潜力本身就是一个话题，在第 8 章中将详细讨论它。

1.8　本章小结

本章提供了 Apple 的新语言 Swift 的速成课程。虽然本章的目的并不是完整介绍 Swift，但仍希望读者通过它能够重新温习一下 Swift 语言的基础知识。如果读者此时仍然不熟悉 Swift，那么在继续学习本书后面的更高级的概念前，可能需要查看和该语言基础知识有关的更全面指导。