Swift 针对iOS和OSX 开发是一门全新的编程语言，但是Swift很多部分对于有经验的C 和 Objective-C 的开发者将会很熟悉。

Swift 提供了C和Objective-C的所有基础数据类型，包含整数Int， 浮点数Double，Float，布尔值 Bool 以及String 字符串。同时Swift也提供了两种强大的集合数据类型，包括 Array和Dictionary。

和C一样，Swift使用变量来储存和访问数值。Swift也同时扩展性的引入了数值不能改变的变量，这在C中我们称为常量，但是在Swift中常量将会更加强大。当变量值不需要改变的时候可以在swift中试用常量，这会使得代码更加安全和简洁。

除了我们熟悉的类型外，Swift也引入了在Objective-C中没有的一些高级数据类型，这包括tuples，这可以使你创建和传递一组数值。Tuples可以从函数以一个单独的复合值的形式返回多个数值。

在Swift中我们也引入了可选的数据类型（Optionals），用于处理变量值“缺席”的情况。可选的意思有两种：一是变量是存在的，它等于X，另一种意思的，该变量值根本不存在。Optionals类似于Objective-C中指向nil的指针，但是适用于所有的数据类型，而非仅仅局限于类，Optionals 相比于Objective-C中nil指针更加安全和简明，并且也是Swift诸多最强大功能的核心。

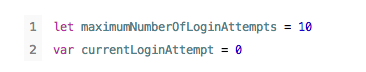
Optionals是Swift做为一个类型安全语言的案例，Swift能帮助你在类型安全的环境下工作，如果你的代码中需要使用String类型，Swift的安全机制能阻止你错误的将Int值传递过来，这能使你再研发阶段就能很早的发现并修正问题。

**常量和变量**

常量和变量将变量名（例如maximumNumberOfLoginAttempts和welcomeMessage）和变量值（例如数值10 和 字符串“Hello”）关联起来，常量当值设定后就不能再改变，而变量则可在未来改变。

声明常量和变量

常量和变量必须在使用前进行声明， 常量用let关键字声明，而变量用var关键字声明，这里有个例子是如何使用常量和变量来跟踪用户尝试登录的次数的：



这段代码可以这么阅读：

“声明一个名字叫maximumNumberOfLoginAttempts的常量，并且将其值设定为10，然后再声明一个新的变量currentLoginAttempt然后将初始值设定为0”

在这个例子中，最大得允许尝试登录数量被声明为常量，因为这永远不会改变，当前尝试登录数被设为变量，因为在登录失败后该变量值会增加。

你也可以在一行中对多个变量或者常量进行声明，中间用分号隔开：



注意：

当你需要储存的变量值不会改变的，请一直使用let将之声明为常量，而变量仅仅是用于储存需要变化的数值。

**类型标注**

你可以在声明常量和变量的时候提供类型标注，用于对变量和常量储存的数据类型进行明确说明。类型标注这么书写：在常量和变量名后添加冒号 然后添加空格 最后再添加变量的类型名。

这个例子提供了一个名叫welcomeMessage的变量类型标注，指明变量可以用来储存字符串String：



上面的冒号意思是“。。。的类型是。。。” 因此我们可以这么来阅读：“声明一个类型是String的变量，取名welcomeMessage”。

这里的意思是可以储存任何String值，可以理解为能够储存的东西的类型。

此时，welcomeMessage就能被赋予任何字符串值了：

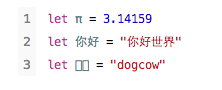


注意：

很少的情况你需要使用类型注释的功能，如果你再声明常量和变量的时候就提供了初始值，那么Swift就能根据提供提供的数值来推断出变量可以储存的类型，在welcomeMessage的例子中，没有任何初始数值提供，所以改变量类型通过变量注释来指定而非通过初始值的推断。

对常量和变量进行命名

你可以使用任何你喜欢的字符来命名你的变量和常量，包含Unicode字符：



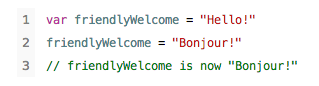
但是常量和变量名不能包含数学符号，箭头，私有或者无效的 Unicode 点 或者线以及边框绘图的字符。并且也不能以数字开头但是数字可以包含在名字中的其他任意地方。

一旦你将变量或者常量声明为某一类型，之后你便不能重新对其再以同样的名字声明了或者将之用来储存其他类型的数据。也不能将变量转化为常量或者将常量转化为变量。

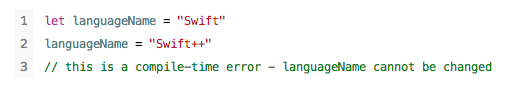
注意：

如果你需要将变量和常量命名为和Swift持有关键字，这时你可以在名字前后加上（`）符号，但是你应该在任何情况下避免这种操作，除非你没有选择。

你可以在相同类型下对同一变量赋予不同的值，例如 friendlyWelcome 值可以从 “Hello!”变为“Bonjour!”:



但是和变量不一样，常量当设定后就不能再更改了，如果这么做在编译的时候会报错：



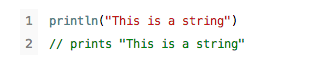
**打印常量和变量**

你可以用println函数来打印当前的变量或者常量值：



println 是一个用来打印的全局函数，在输出的地方会跟一个“断行”如果你再Xcode中使用，结果会输出到Xcode的console面板。（另外一个函数print 会执行同样的功能，只是不会在输出结果后跟一个“断行”）

println会打印你传递进来的字符串值：



除此之外，println也可以打印更为复杂的日志信息，类似于Cocoa的NSLog函数，信息中可以包含当前常量和变量的数值。

Swift使用插入字符串的形式将常量或者变量值打印到信息中，而变量名或者常量名则用来做为插入的一个占位符。格式如下： 在需要打印变量或常量的地方用 \(变量名) 来替代：



注意：

字符串插值的所有选项我们会在“字符串插值”章节讲到

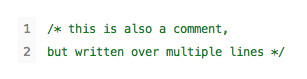
**注释**

可以使用注释来书写不被执行的文档，用于注释或者来提醒你自己。注释会Xcode编译的时候被忽略。

Swift中得注释和C中非常相似，单行的注释以（//）开始：



你也可以书写多行注释，以(/\*)开始，并且以(\*/)结尾。

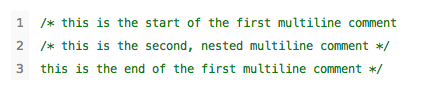


不像C中得多行注释，在Swift中，多行注释是可以内嵌的，如下：

/\* 这是第一个多行注释

/\*这是第二个多行注释\*/

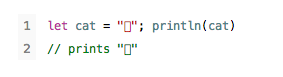
这是第一个多行注释的结尾\*/



内嵌注释可以使你能迅速和简单的注释掉大部分代码，即使代码已经包含了多行注释。

**分号**

和其他语言不一样，Swift不需要你再每个语句后书写分号结尾，尽管你也可以这么做。但是如果你要再一行中书写多个语句，那么分号就是必须的。



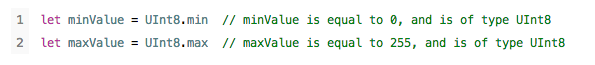
**整数**

整数类型是指的没有小数部分的数字，比如42和-23，整数可以是有符号的（正整数，零或者负整数） 或者无符号的（正整数或者零）

Swift提供的有符号的整数和无符号的整数有8，16，32，64位形式。这些整数命名规则遵循C中惯例，UInt8 代表8位无符号整数，Int32代表32位有符号整数。这些整数有大写的名字。

**整数范围**

你可以通过min和max属性访问整数的最大值和最小值：



这些属性值是数字类型的恰当大小(比如上面UInt8的例子) 也因此可以和其他同类型的值用于一个表达式中。

**Int**

在更多的情况下，你不需要在代码中指定整数的大小，Swift提供了另外一个整数类型Int，该类型会和当前的操作系统平台原生字大小匹配：

对于32位平台，Int大小和Int 32相同。

对于64位平台，Int大小和Int 64相同。

除非你需要使用某一大小的整数，那么请一直在代码中使用Int, 这有助于提升代码的一致性和互通性。即使在32位机器上，Int也可以存储-2,147,483,648 到 2,147,483,647范围类的值，这对于许多整数范围已经足够了。

**UInt**

Swift也提供了无符号的整数类型，该类型会和当前的操作系统平台原生字大小匹配：

对于32位平台，UInt大小和UInt 32相同。

对于64位平台，UInt大小和UInt64相同。

注意：

仅在你有特定需求的时候使用和平台原生字大小相同的无符号类型整数UInt, 其他时候推荐使用Int，即使储存的数值已经知道是非负数的。对Int一致性使用能够使代码更具互通性，并且避免了在不同数值类型间转化的需求，这和整数类型推测是匹配的。

**浮点数值**

浮点数值是具有小数部分的浮点数，比如3.14159和-273.15。

相比整数来说，浮点数类型能表示更广的数值范围，储存更大或者更小数字。Swift提供了两种有符号的浮点数类型：

Double表示64位浮点数，在浮点数需要特别大或者特别精确的时候使用。

Float表示32位浮点数，在不需要64位精度的时候使用。

注意

Double具有15位小数的精度，而Float只有6位，如何选择需要根据你代码中对数值范围和大小的具体需求来决定。

**类型安全和类型推测**

Swift是一个类型安全的编程语言。这鼓励你在代码中明确数值的类型。如果你代码中需要用到String，那么你不能错误的将Int传递进去。

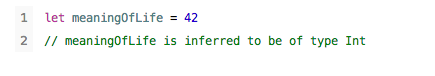
因为Swift是类型安全的，所以它会再编译的时候对类型进行检查，任何错误和不匹配都会报错。这会使得你再研发的时候尽可能早得发现和修正问题。

类型检查能够帮助你在使用不同类型值的过程中避免错误，但是，这并不意味着你必须指定每一个你声明的变量或者常量类型。如果你没有指定变量或者常量类型，Swift会推测出恰当的数据类型。类型推测使得编译器在编译代码的时候简单的通过查看你提供的数值来自动推测出相应的数据类型。

因为类型推测，相比C和Objective-C来说Swift对类型声明需求就非常少了，变量和常量类型仍然是非常明确指定的，但是更多的工作已经为你完成了。

类型推测在你声明变量或者常量并同时赋值的时候特别有用。这通过在声明的时候对变量或者常量赋予字面值来完成。（字面值的意思就是在下面例子中直接出现在源代码中得数值，比如42和3.14159）

例如，如果你再声明一个常量的时候，将42赋给该常量，那么Swift则会推测出该常量的类型为Int, 因为你初始化常量的值看起来像是一个整数。

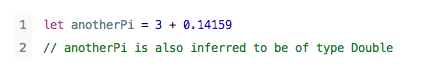


同样的，如果你不指定浮点字面值的类型，Swift会推测出你想创建Double类型：



当推测浮点数值的时候，Swift一直会选择Double而非Float。

如果你再表达式中混合浮点数值和整数值，那么Double会做为推测的结果。



表达式中得3没有任何明确的类型，因此选择Double是源于加法中浮点字面值的出现。

**数字**

整数字可以如下书写：

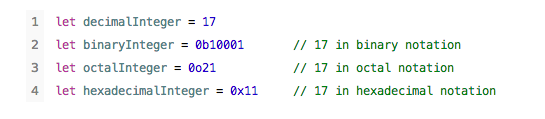
十进制数字，没有前缀

二进制数字，以0b作为前缀

八进制数字，以0o作为前缀

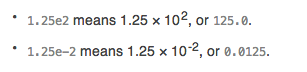
十六进制数字，以0x作为前缀

这些数字表示都和17 十进制等价：

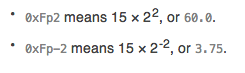


浮点数值可以是十进制（没有前缀）或者是十六进制（0x前缀），他们必须在小数点左右都要由数字，指数是可选的，用大小写e来表示十进制浮点或者大小写p来表示十六进制浮点。

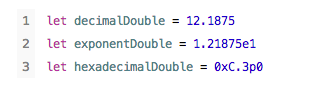
对于十进制数字指数exp, 基数的值则乘以10exp



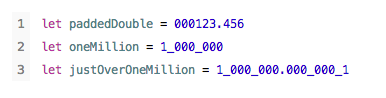
对于十六进制指数exp, 基数则乘以2exp



12.1875十进制形式有如下几种表现方式：



数字也可以包含格式化信息来使得他们变得易读。整数和浮点数都能前面添加额外的零或者下划线来提升可读性。这两种格式化都不会影响实际的数值：



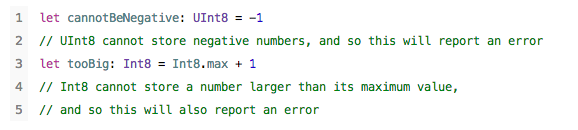
**数字类型转换**

在通常情况下请使用Int来标记变量或者常量的整数类型，即使已知的数值并不为负，使用默认的整数类型好处是可以保证常量和变量类型的一致性，并且和推测出得类型进行匹配。

仅在有特殊需求的时候使用其他的数据类型，比如外来源明确大小的数据，或者是因为性能，内存使用或者其他必要的优化。在这些情况下使用明确的大小的类型能够有助于发现意外数值溢出并潜在记录使用的数据特性。

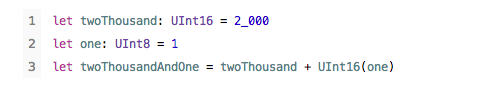
**整数转换**

数字的储存范围根据不同的数字类型有所不同。Int8的常量或者变量可以储存-128到127得整数范围，而UInt8常量或者变量可以储存0到255的数字范围，如果储存的数字超过相应数字类型的范围在编译的时候则会报错。



因为不同的数字类型会储存不同范围的数值，你必须根据具体的情况选择数字类型的转换，这种明确的转换方式避免了隐藏转换的错误并且使得代码中类型转换目的明确化。

用当前数值来初始化目标数字类型数字可以将当前数字类型转换到其他类型。在下面的例子中，常量twoThousand 是 UInt16类型，而常量one是UInt8数字类型，它们不能直接相加，因为它们不是同一类型。我们通过UInt16(one)创建一个新的UInt16，并且用one得值来初始化，这样我们就可以用这个值来替换原来的了：

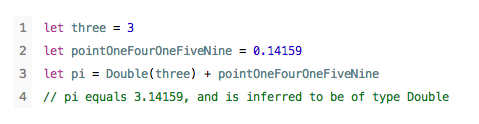


因为现在两边的数字类型都是UInt16了，现在加法就允许了，输出的结果常量twoThousandAndOne被推测为UInt16类型，因为它使两个UInt16相加的结果。

类型(初始值) 是调用Swift类型构造函数并传入相应值的默认方法。事实上，UInt16拥有一个可以接受UInt8值的构造函数，因此该构造函数用于从现有的UInt8来创建一个新的UInt16. 你不能再这里传递任意类型，这必须是UInt16构造函数所支持的。

**整数和浮点数间转换**

整数和浮点数间的转换必须是明确的：



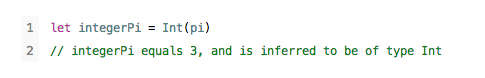
这里常量three用于创建一个新的Double,这样等式两边数字类型相同，否则是不能相加的。

反之亦然，整数类型也可用Double或者Float来初始化：

浮点数会在转化为整数的过程中丢失小数部分，也就是说4.75会变成4，-3.9 会变成-3.

注意：

对数值类型的变量或者常量组合不同于数字简单的组合，数字3可以直接加到0.14159上是因为数字本身不具有明确的数字类型，他们的类型会再编译的时候由编译器推测。



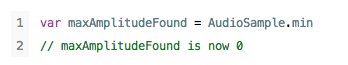
**类型别名**

类型别名对当前的类型定义了另一个名字，类型别名通过使用typealias关键字来定义。

根据特定的情况，你可能会使用其他的名字来引用某一类型，这时候就可以使用类型别名，比如使用来自外部特定大小的数据：



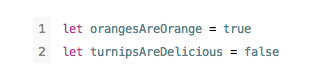
当你定义了类型别名，在代码中都可以使用类型别名对原名进行替代：



这里AudioSample被定义为UInt16的类型别名，因为这是别名，所以调用AudioSample.min和调用UInt16.min是等价的，这里为maxAmplitudeFound提供了初始值0。

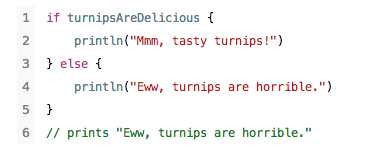
**布尔**

Swift提供了基础的布尔值类型，Bool，布尔值也是逻辑值因为它只能为真或者为假，Swift提供了两个布尔值常量 true 和 false：



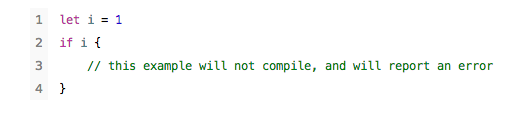
orangesAreOrange 和 turnipsAreDelicious 被推测为布尔类型是因为它们被布尔值初始化。和Int, Double一样只要在最初声明的时候用布尔值进行初始化，是不需要明确的去声明其布尔类型的。类型推测使得代码更加的简洁，易读。

布尔值在条件语句的使用中非常有用，比如if语句：

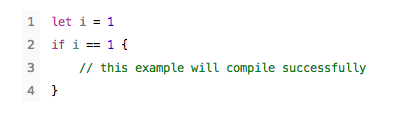


if语句会在Control Flow章节中详细探讨

Swift 类型安全会阻止使用非布尔值类型来替代Bool, 下面的例子中编译会报错：



但是这个例子却是正确的：



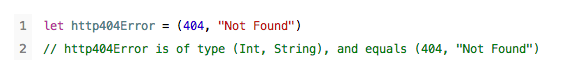
i == 1 的比较结果是Bool类型，因此可以在该例子中使用，类似 i == 1 比较会在Basic Operators章节中讨论。

和其他类型安全的例子一样，这样一来防止了意外的错误发生并且保证了代码目的的清晰性。

**Tuples(元组)**

元组可以将多个数值组合成一个复合值，元组中得值可以是任意类型，并且值与值之间的类型可以不同。

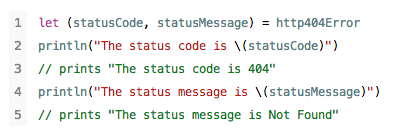
在这里例子中 (404,”Not Found”)是一个描述HTTP状态码的元组。 HTTP状态码是在请求网页的时候由服务器返回的，404 Not Found表示请求的网页不存在。



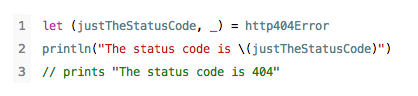
(404,”Not Found”) 将 Int（数字）和String（描述）组合在一起描述HTTP的状态码。可以这么说明：(Int,String)元组类型。

你可以通过元组创建任意类型的排列，并且可以包含任意数量的不同类型，比如你可以创建 (Int,Int,Int) 或者 (String,Bool) 或者任意其他你需要的排列组合。

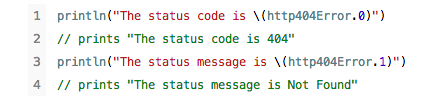
你也可以分解元组的内容到常量或者变量中，然后进行访问。



如果你仅需要元组中得某些数值，那么在分解元组的时候使用下划线来忽略其他的部分



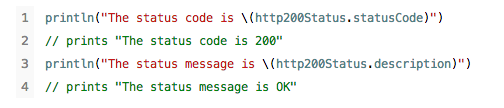
或者使用从0开始的索引来访问某一个元组元素：



你也可以在元组定义的时候对元组中得元素进行命名：



如果你对元组中得元素进行了命名，那么也可以使用元素的名字来访问对应的数值：



元组在函数返回值上特别有用，一个尝试提取网页信息的函数可以返回(Int, String)元组类型来描述网页提取的成功或者失败的信息。相比于函数返回单一的类型值，返回多个类型值的函数能提供更多有用的信息。

注意：

元组仅对临时的相关数据组合有用，他们并不适合于创建复杂的数据结构，如果你需要持续使用数据结构，请使用类或者结构类型而非元组类型。