翻译：lyuka 校对：numbbbbb

类型（Types）

本页包含内容：

Swift 语言存在两种类型：命名型类型和复合型类型。命名型类型是指定义时可以给定名字的类型。命名型类型包括类、结构体、枚举和协议。比如，一个用户定义的类`MyClass`的实例拥有类型`MyClass`。除了用户定义的命名型类型，Swift 标准库也定义了很多常用的命名型类型，包括那些表示数组、字典和可选值的类型。

那些通常被其它语言认为是基本或初级的数据型类型（Data types）——比如表示数字、字符和字符串——实际上就是命名型类型，Swift 标准库是使用结构体定义和实现它们的。因为它们是命名型类型，因此你可以按照“扩展和扩展声明”章节里讨论的那样，声明一个扩展来增加它们的行为以适应你程序的需求。

复合型类型是没有名字的类型，它由 Swift 本身定义。Swift 存在两种复合型类型：函数类型和元组类型。一个复合型类型可以包含命名型类型和其它复合型类型。例如，元组类型`(Int, (Int, Int))`包含两个元素：第一个是命名型类型`Int`，第二个是另一个复合型类型`(Int, Int)`.

本节讨论 Swift 语言本身定义的类型，并描述 Swift 中的类型推断行为。

类型的语法： *type* → *array-type* | *function-type* | *type-identifier* | *tuple-type* | *optional-type* | *implicitly-unwrapped-optional-type* | protocol-composition-type | metatype-type

类型注解

类型注解显式地指定一个变量或表达式的值。类型注解始于冒号`:`终于类型，比如下面两个例子：

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
func someFunction(a: Int){ /* ... */ }
```

在第一个例子中，表达式`someTuple`的类型被指定为`(Double, Double)`。在第二个例子中，函数`someFunction`的参数`a`的类型被指定为`Int`。

类型注解可以在类型之前包含一个类型特性（type attributes）的可选列表。

类型注解的语法： *type-annotation* → :*attributes*[opt] *type*

类型标识符

类型标识符引用命名型类型或者是命名型/复合型类型的别名。

大多数情况下，类型标识符引用的是同名的命名型类型。例如类型标识符`Int`引用命名型类型`Int`，同样，类型标识符`Dictionary<String, Int>`引用命名型类型`Dictionary<String, Int>`。

在两种情况下类型标识符引用的不是同名的类型。情况一，类型标识符引用的是命名型/复合型类型的类型别名。比如，在下面的例子中，类型标识符使用`Point`来引用元组`(Int, Int)`：

```
typealias Point = (Int, Int)
```

```
let origin: Point = (0, 0)
```

情况二，类型标识符使用dot(.)语法来表示在其它模块（modules）或其它类型嵌套内声明的命名型类型。例如，下面例子中的类型标识符引用在`ExampleModule`模块中声明的命名型类型`MyType`：

```
var someValue: ExampleModule.MyType
```

类型标识符的语法： *type-identifier* → *type-name generic-argument-clause*[opt] | *type-name generic-argument-clause*[opt].*type-identifier type-name* → *identifier*

元组类型

元组类型使用逗号隔开并使用括号括起来的0个或多个类型组成的列表。

你可以使用元组类型作为一个函数的返回类型，这样就可以使函数返回多个值。你也可以命名元组类型中的元素，然后用这些名字来引用每个元素的值。元素的名字由一个标识符和`:`组成。“函数和多返回值”章节里有一个展示上述特性的例子。

`void`是空元组类型`()`的别名。如果括号内只有一个元素，那么该类型就是括号内元素的类型。比如，`(Int)`的类型是`Int`而不是`(Int)`。所以，只有当元组类型包含两个元素以上时才可以标记元组元素。

元组类型语法： *tuple* → (*tuple-type-body*[opt]) *tuple-type-body* → *tuple-type-element-list* ...[opt] *tuple-type-element-list* → *tuple-type-element* | *tuple-type-element*, *tuple-type-element-list* *tuple-type-element* → *attributes*[opt] **inout**[opt] *type* | **inout** [opt] *element-name type-annotation element-name* → *identifier*

函数类型

函数类型表示一个函数、方法或闭包的类型，它由一个参数类型和返回值类型组成，中间用箭头`->`隔开：

- `parameter type` -> `return type`

由于 参数类型 和 返回值类型 可以是元组类型，所以函数类型可以让函数与方法支持多参数与多返回值。

你可以对函数类型应用带有参数类型`()`并返回表达式类型的`auto_closure`属性（见类型属性章节）。一个自动闭包函数捕获特定表达式上的隐式闭包而非表达式本身。下面的例子使用`auto_closure`属性来定义一个很简单的assert函数：

```
func simpleAssert(condition: @auto_closure () ->
Bool, message: String){
    if !condition(){
        println(message)
    }
}
let testNumber = 5
simpleAssert(testNumber % 2 == 0, "testNumber isn't
an even number.")
// prints "testNumber isn't an even number."
```

函数类型可以拥有一个可变长参数作为参数类型中的最后一个参数。从语法角度上讲，可变长参数由一个基础类型名字和`...`组成，如`Int...`。可变长参数被认为是一个包含了基础类型元素的数组。即`Int...`就是`Int[]`。关于使用可变长参数的例子，见章节“可变长参数”。

为了指定一个`in-out`参数，可以在参数类型前加`inout`前缀。但是你不可以对可变长参数或返回值类型使用`inout`。关于In-Out参数的讨论见章节In-Out参数部分。

柯里化函数（curried function）的类型相当于一个嵌套函数类型。例如，下面的柯里化函数`addTwoNumber()()`的类型是`Int -> Int -> Int`：

```
func addTwoNumbers(a: Int)(b: Int) -> Int{
    return a + b
}
addTwoNumbers(4)(5)       // returns 9
```

柯里化函数的函数类型从右向左组成一组。例如，函数类型`Int -> Int -> Int`可以被理解为`Int -> (Int -> Int)`——也就是说，一个函数传入一个`Int`然后输出作为另一个函数的输入，然后又返回一个`Int`。例

如，你可以使用如下嵌套函数来重写柯里化函数`addTwoNumbers()()`：

```
func addTwoNumbers(a: Int) -> (Int -> Int){
    func addTheSecondNumber(b: Int) -> Int{
        return a + b
    }
    return addTheSecondNumber
}
addTwoNumbers(4)(5)      // Returns 9
```

函数类型的语法： *function-type → type -> type*

数组类型

Swift语言使用类型名紧接中括号`[]`来简化标准库中定义的命名型类型`Array<T>`。换句话说，下面两个声明是等价的：

```
let someArray: String[] = ["Alex", "Brian", "Dave"]
let someArray: Array<String> = ["Alex", "Brian",
"Dave"]
```

上面两种情况下，常量`someArray`都被声明为字符串数组。数组的元素也可以通过`[]`获取访问：`someArray[0]`是指第0个元素“Alex”。

上面的例子同时显示，你可以使用`[]`作为初始值构造数组，空的`[]`则用来来构造指定类型的空数组。

```
var emptyArray: Double[] = []
```

你也可以使用链接起来的多个`[]`集合来构造多维数组。例如，下例使用三个`[]`集合来构造三维整型数组：

```
var array3D: Int[][][] = [[[1, 2], [3, 4]], [[5, 6],
[7, 8]]]
```

访问一个多维数组的元素时，最左边的下标指向最外层数组的相应位置元素。接下来往右的下标指向第一层嵌入的相应位置元素，依次类推。这就意味着，在上面的例子中，`array3D[0]`是指`[[1, 2], [3, 4]]`，`array3D[0][1]`是指`[3, 4]`，`array3D[0][1][1]`则是指值`4`。

关于Swift标准库中`Array`类型的细节讨论，见章节Arrays。

数组类型的语法： *array-type* → *type*`[ ]` | *array-type*`[ ]`

可选类型

Swift定义后缀`?`来作为标准库中的定义的命名型类型`Optional<T>`的简写。换句话说，下面两个声明是等价的：

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

在上述两种情况下，变量`optionalInteger`都被声明为可选整型类型。注意在类型和`?`之间没有空格。

类型`Optional<T>`是一个枚举，有两种形式，`None`和`Some(T)`，又来代表可能出现或可能不出现的值。任意类型都可以被显式的声明（或隐式的转换）为可选类型。当声明一个可选类型时，确保使用括号给`?`提供合适的作用范围。比如说，声明一个整型的可选数组，应写作`(Int[])?`，写成`Int[]?`的话则会出错。

如果你在声明或定义可选变量或特性的时候没有提供初始值，它的值则会自动赋成缺省值`nil`。

可选符合`LogicValue`协议，因此可以出现在布尔值环境下。此时，如果一个可选类型`T?`实例包含有类型为`T`的值（也就是说值为`Optional.Some(T)`），那么此可选类型就为`true`，否则为`false`。

如果一个可选类型的实例包含一个值，那么你就可以使用后缀操作符`!`来获取该值，正如下面描述的：

```
optionalInteger = 42
optionalInteger!      // 42
```

使用`!`操作符获取值为`nil`的可选项会导致运行错误（runtime error）。

你也可以使用可选链和可选绑定来选择性的执行可选表达式上的操作。如

果值为`nil`，不会执行任何操作因此也就没有运行错误产生。

更多细节以及更多如何使用可选类型的例子，见章节“可选”。

可选类型语法： *optional-type → type?*

隐式解析可选类型

Swift语言定义后缀`!`作为标准库中命名类型`ImplicitlyUnwrappedOptional<T>`的简写。换句话说，下面两个声明等价：

```
var implicitlyUnwrappedString: String!
var implicitlyUnwrappedString:
ImplicitlyUnwrappedOptional<String>
```

上述两种情况下，变量`implicitlyUnwrappedString`被声明为一个隐式解析可选类型的字符串。注意类型与`!`之间没有空格。

你可以在使用可选的地方同样使用隐式解析可选。比如，你可以将隐式解析可选的值赋给变量、常量和可选特性，反之亦然。

有了可选，你在声明隐式解析可选变量或特性的时候就不用指定初始值，因为它有缺省值`nil`。

由于隐式解析可选的值会在使用时自动解析，所以没必要使用操作符`!`来解析它。也就是说，如果你使用值为`nil`的隐式解析可选，就会导致运行错误。

使用可选链会选择性的执行隐式解析可选表达式上的某一个操作。如果值为`nil`，就不会执行任何操作，因此也不会产生运行错误。

关于隐式解析可选的更多细节，见章节“隐式解析可选”。

隐式解析可选的语法： implicitly-unwrapped-optional-type → type!

协议合成类型

协议合成类型是一种符合每个协议的指定协议列表类型。协议合成类型可能会用在类型注解和泛型参数中。

协议合成类型的形式如下：

```
protocol<Protocol 1, Procotol 2>
```

协议合成类型允许你指定一个值，其类型可以适配多个协议的条件，而且不需要定义一个新的命名型协议来继承其它想要适配的各个协议。比如，协议合成类型`protocol<Protocol A, Protocol B, Protocol C>`等效于一个从`Protocol A`，`Protocol B`，`Protocol C`继承而来的新协议`Protocol D`，很显然这样做有效率的多，甚至不需引入一个新名字。

协议合成列表中的每项必须是协议名或协议合成类型的类型别名。如果列表为空，它就会指定一个空协议合成列表，这样每个类型都能适配。

协议合成类型的语法： *protocol-composition-type* → **protocol** <*protocol-identifier-list[opt]*> *protocol-identifier-list* → *protocol-identifier* | *protocol-identifier*, *protocol-identifier-list* *protocol-identifier* → *type-identifier*

元类型

元类型是指所有类型的类型，包括类、结构体、枚举和协议。

类、结构体或枚举类型的元类型是相应的类型名紧跟`.Type`。协议类型的元类型——并不是运行时适配该协议的具体类型——是该协议名字紧跟`.Protocol`。比如，类`SomeClass`的元类型就是`SomeClass.Type`，协议`SomeProtocol`的元类型就是`SomeProtocal.Protocol`。

你可以使用后缀`self`表达式来获取类型。比如，`SomeClass.self`返回`SomeClass`本身，而不是`SomeClass`的一个实例。同样，`SomeProtocol.self`返回`SomeProtocol`本身，而不是运行时适配`SomeProtocol`的某个类型的实例。还可以对类型的实例使用`dynamicType`表达式来获取该实例在运行阶段的类型，如下所示：

```
class SomeBaseClass {
    class func printClassName() {
        println("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        println("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass()
// someInstance is of type SomeBaseClass at compile
time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName()
// prints "SomeSubClass
```

元类型的语法： *metatype-type* → *type*.**Type** | *type*.**Protocol**

类型继承子句

类型继承子句被用来指定一个命名型类型继承哪个类且适配哪些协议。类型继承子句开始于冒号`:`，紧跟由`,`隔开的类型标识符列表。

类可以继承单个超类，适配任意数量的协议。当定义一个类时，超类的名字必须出现在类型标识符列表首位，然后跟上该类需要适配的任意数量的协议。如果一个类不是从其它类继承而来，那么列表可以以协议开头。关于类继承更多的讨论和例子，见章节“继承”。

其它命名型类型可能只继承或适配一个协议列表。协议类型可能继承于其它任意数量的协议。当一个协议类型继承于其它协议时，其它协议的条件集合会被集成在一起，然后其它从当前协议继承的任意类型必须适配所有这些条件。

枚举定义中的类型继承子句可以是一个协议列表，或是指定原始值的枚举，一个单独的指定原始值类型的命名型类型。使用类型继承子句来指定

原始值类型的枚举定义的例子，见章节“原始值”。

类型继承子句的语法： *type-inheritance-clause* → *:type-inheritance-list type-inheritance-list* → *type-identifier* | *type-identifier*, *type-inheritance-list*

类型推断

Swift广泛的使用类型推断，从而允许你可以忽略很多变量和表达式的类型或部分类型。比如，对于`var x: Int = 0`，你可以完全忽略类型而简写成`var x = 0`——编译器会正确的推断出`x`的类型`Int`。类似的，当完整的类型可以从上下文推断出来时，你也可以忽略类型的一部分。比如，如果你写了`let dict: Dictionary = ["A": 1]`，编译提也能推断出`dict`的类型是`Dictionary<String, Int>`。

在上面的两个例子中，类型信息从表达式树（expression tree）的叶子节点传向根节点。也就是说，`var x: Int = 0`中`x`的类型首先根据`0`的类型进行推断，然后将该类型信息传递到根节点（变量`x`）。

在Swift中，类型信息也可以反方向流动——从根节点传向叶子节点。在下面的例子中，常量`eFloat`上的显式类型注解（`:Float`）导致数字字面量`2.71828`的类型是`Float`而非`Double`。

```
let e = 2.71828 // The type of e is inferred to be Double.
let eFloat: Float = 2.71828 // The type of eFloat is Float.
```

Swift中的类型推断在单独的表达式或语句水平上进行。这意味着所有用于推断类型的信息必须可以从表达式或其某个子表达式的类型检查中获取。