类型

cnswift.org/types

在 Swift 中，有两种类型：命名类型和复合类型。*命名类型*是在定义时给定特定名字的类型。命名类型包括类、结构体、枚举和协议。例如，自定义的类 MyClass 的实例拥有类型 MyClass 。另外，Swift 标准库定义了许多常用的命名类型，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类型——例如表示数字、字符和字符串的类型——实际上就是命名类型，在 Swift 标准库中用结构体来定义和实现。因为它们是命名类型，你可以通过使用扩展增加它们的行为来符合你程序的需求，在扩展和 扩展声明（此处应有链接）里有相关讨论。

*复合类型*是没有名字的类型，定义在 Swift 语言本身中。复合类型有两种：函数型类型和元组型类型。复合类型可以包含命名类型和其他复合类型。例如，元组型类型 (Int,(Int,Int)) 包含两个元素：第一个是命名类型 Int ，第二个是另一个复合类型 (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
> *类型* → *数组类型* | *字典类型* | *函数类型* | *类型标识符* | 元组类型 | *可选类型* | *隐式展开可选类型* | *协议组合类型* | 元类型

类型标注

*类型标注*显式地指定一个变量或表达式的类型。类型标注以冒号（:）开始以类型结束。

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

在第一个例子中，表达式 someTuple 的类型被指定为元组类型 (Double, Double) 。在第二个例子中，函数 someFunction 的形式参数 a 的类型被显式地指定为类型 Int 。

类型标注可以在类型之前包含一个类型特性的列表。

> 类型标注的语法
> *type-annotation* → : *attributes*$_{opt}$ **inout**$_{opt}$ *type*

类型标识符

类型标识符指的是命名类型或者是命名型和复合类型的别名。

大多数情况下，类型标识符指的是与类型标识符同名的命名类型。例如 Int 类型标识符指的是命名类型 Int ，类型标识符 Dictionary<String, Int> 指的是命名类型 Dictionary<String, Int> 。

有两种情况类型标识符不是指的是和类型标识符同名的类型。第一种情况，类型标识指的是某个命名或复合类型的类型别名。例如，在下面的例子中，类型标注中 Point 指的是元组类型 (Int, Int) ：

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

第二种情况，类型标识符使用点(.)语法来表示声明在其它模块中或内嵌在其它类型中的命名类型。例如，下面代码中的类型标识符指的是在 ExampleModule 模块中声明的命名类型 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

元组类型是是使用逗号分隔的零个或多个类型的列表，用括号括起来。

你可以使用元组类型作为函数的返回值使函数返回一个包含多个值的元组。你也能命名元组的元素并且使用那些名字来表示每个元素的值。一个元素的名字由一个标识符与紧随其后的冒号（：）组成。例如在函数和多返回值中所展示的这些特点，见 多返回值的函数。

Void 是空元组类型 () 的别名。如果括号中只有一个元素，类型就是那个元素的类型。例如，类型 (Int) 就是 Int ，而不是 (Int) 。因此，你只能在元组中的元素当元组有2个或多个元素时给元素命名。

> 元组类型的语法
> *tuple-type* → (*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* | *element-nametype-annotation*
> *element-name* → *identifier*

函数类型

函数类型表示一个类别函数、方法或闭包的类型，由形式参数类型和返回值类型组成，用箭头(->)隔开：

(parameter type) -> return type
*形式参数类型*是逗号分隔的一系列类型。由于*返回值类型*可以是元组类型，函数类型支持多形式参数与多返回值的函数与方法。

函数类型的形式参数 () - > T （其中 T 是任何类型）可以应用 autoclosure 特性在其调用时隐式创建闭包。这提供了语法上方便的方式来推迟表达式的执行而不需要在调用函数时写一个显式的闭包。例如自动闭包函数类型形式参数，参见自动闭包。

函数类型可以在它的形式参数类型中有可变形式参数。在语法上，一个可变参数由一个紧跟三个点（ ... ）的基本类型名组成，如 Int... 。一个可变形式参数被视为包含基本类型元素的数组。例如，一个可变形式参数 Int... 被视为 [Int] 。使用可变形式参数的示例，请参阅可变形式参数。

要指定输入输出形式参数，对形式参数类型使用 inout 前缀关键字。你不能用 inout 关键字标记可变形式参数或返回类型。输入输出形式参数请参阅 输入输出形式参数。

如果一个函数类型只有一个形式参数而且形式参数的类型是元组类型，那么元组类型在写函数类型的时候必须用圆括号括起来。比如说， ((Int, Int)) -> Void 是接收一个元组 (Int, Int) 作为形式参数的函数的类型。通常来京，不加括号， (Int, Int) -> Void 时一个接收两个 Int 形式参数并且不返回任何值的函数的类型。

函数和方法的实际参数名与函数类型无关，比如说：

```
1   func someFunction(left: Int, right: Int) {}
2   func anotherFunction(left: Int, right: Int) {}
3   func functionWithDifferentLabels(top: Int, bottom: Int) {}
4   var f = someFunction // The type of f is (Int, Int) -> Void, not (left: Int, right: Int)
5   -> Void.
6   f = anotherFunction            // OK
7   f = functionWithDifferentLabels  // OK
8   func functionWithDifferentArgumentTypes(left: Int, right: String) {}
9   f = functionWithDifferentArgumentTypes     // Error
10  func functionWithDifferentNumberOfArguments(left: Int, right: Int, top: Int) {}
11  f = functionWithDifferentNumberOfArguments // Error
12
13
```

由于实际参数不是函数类型的一部分，你可以在写函数类型的时候省略它们。

```
1   var operation: (lhs: Int, rhs: Int) -> Int     // Error
2   var operation: (_ lhs: Int, _ rhs: Int) -> Int // OK
3   var operation: (Int, Int) -> Int               // OK
```

如果函数类型包括不止一个箭头（ -> ），函数类型从右到左进行分组。例如，函数类型 (Int) -> (Int) -> (Int) 被理解为 (Int) -> ((Int) -> (Int)) ——也就是说，一个接收 Int 返回类型另一个接收和返回 Int 函数的函数。

函数类型如果要抛出错误就必须使用 throws 关键字标记，而且能重抛错误的函数类型必须使用 rethrows 关键字标记。 throws 关键字是函数类型的一部分，不抛出函数是抛出函数的子类型。所以，你可以在使用抛出函数的位置使用不抛出函数。抛出和重抛函数在抛出函数与方法（此处应有链接）和重抛函数与方法（此处应有链接）。

> 函数类型的语法
>
> unction-type → attributes$_{opt}$function-type-argument-clause `throws` $_{opt}$ `->` type
>
> function-type → attributes$_{opt}$function-type-argument-clause `rethrows` `->` type
>
> function-type-argument-clause → `(` `)`
>
> function-type-argument-clause → `(` function-type-argument-list `...` $_{opt}$ `)`
>
> function-type-argument-list → function-type-argument function-type-argument `,` function-type-argument-list
>
> function-type-argument → attributes$_{opt}$ `inout` $_{opt}$type argument-labeltype-annotation
>
> argument-label → identifier

数组类型

Swift 语言为 Swift 标准库中 Array<Element> 类型提供下列语法糖：

[type]

换句话说，下面两个声明是等价的：

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

上面两种情况中，常量 someArray 声明为字符串数组。数组的元素可以通过下标在方括号中指定一个有效的索引值访问： someArray[0] 是指第 0 个元素 "Alex" 。

你可以通过内嵌多对方括号创建多维数组，元素的基本类型名包含在最里面的方括号。例如，下你可以用三对方括号创建三维整数数组：

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

访问一个多维数组的元素时，最左边的下标索引值指的是最外层数组的相应位置元素。接下来的下标索引值指的是第一层嵌套的相应位置元素。依次类推。这意味着在上面的例子中， array3D[0] 指的是 [[1, 2], [3, 4]] ， array3D[0][1] 是指 [3, 4] ， array3D[0][1][1] 则是指的是值 4。

关于 Swift 标准库中 Array 类型的详细讨论，参阅 数组。

> 数组类型的语法
> *array-type* → [*type*]

字典类型

Swift 语言为 Swift 标准库中的 Dictionary<Key,Value> 类型提供下列语法糖：

[key type : value type]

换句话说，下面两个声明是等价的：

```
1  let someDictionary: [String: Int] = ["Alex": 31, "Paul": 39]
2  let someDictionary: Dictionary<String, Int> = ["Alex": 31, "Paul": 39]
```

上面两种情况中，常量 someDictionary 声明为字符串类型的键和值为整数类型的值的字典

字典中的值可以通过下标指定方括号中的相应键来访问： someDictionary["Alex"] 指的是和键 "Alex" 关联的值。下标返回一个字典中键的类型的可选项。如果被指定的键在字典中不存在的话，下标返回 nil 。

字典中键的类型必须遵循 Swift 标准库中的 Hashable 协议。

关于 Swift 标准库中 Dictionary 类型的详细讨论可，参阅字典。

> 字典类型的语法
> *dictionary-type* → [*type* : *type*]

可选类型

Swift 为命名类型 Optional<Wrapped> 定义后缀 ? 作为语法糖 ，其定义在 Swift 标准库中。换句话说，下列两种声明是等价的:

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

在上述两种情况下，变量 optionalInteger 都声明为可选整数类型。注意在类型和 ? 之间没有空格。

类型 Optional<Wrapped> 是有两种情况的枚举， none 和 Some(Wrapped) ，它代表可能没有值或可能有值。任何类型都可以被显式的声明（或隐式的转换）为可选类型。如果你在声明可选的变量或属性时没有提供初始值，它的值则会默认为 nil 。

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，如下描述：

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

使用 ! 操作符获解析一个值为 nil 的可选项会导致运行时错误。

你也可以使用可选链和可选绑定来有条件地执行对可选表达式的操作。如果值为 nil ，不会执行任何操作并且不会因此产生运行时错误。

关于更多的信息以及查看如何使用可选类型的示例，参阅可选项。

> 可选类型的语法
> *optional-type → type* **?**

隐式展开可选类型

Swift 为命名类型 Optional<Wrapped> 定义后缀 ! 作为语法糖 ，其定义在 Swift 标准库中，作为它被访问时自动解析的附加行为。如果你试图使用一个值为 nil 的隐式解析，你会得到一个运行时错误。除了隐式展开的行为之外，下面两个声明是等价的：

```
var implicitlyUnwrappedString: String!
var explicitlyUnwrappedString: Optional<String>
```

注意类型与 ! 之间没有空格。

因为隐式展开改变了包含该类型的声明的含义，内嵌在元组类型或泛型类型中的可选类型——如字典或数组中元素的类型——不能被标记为隐式展开。例如:

```
let tupleOfImplicitlyUnwrappedElements: (Int!, Int!)  // Error
let implicitlyUnwrappedTuple: (Int, Int)!             // OK
let arrayOfImplicitlyUnwrappedElements: [Int!]        // Error
let implicitlyUnwrappedArray: [Int]!                  // OK

```

因为隐式展开可选项有相同的 Optional<Wrapped> 类型作为可选值，你可以使用隐式展开可选项在代码中所有你使用可选项的地方。例如，你可以把隐式展开可选项赋值给变量、常量、以及可选属性，反之亦然。

有了可选项，如果在声明隐式展开可选变量或属性时你不用提供初始值，它的值会默认为 nil 。

使用可选链有条件地对隐式展开可选项的表达式进行操作。如果值为 nil ，就不执行任何操作，因此也不会产生运行错误。

关于隐式解析可选的更多信息，参阅隐式展开可选项。

> 隐式展开可选类型的语法
> *implicitly-unwrapped-optional-type → type* **!**

协议组合类型

协议组合类型表述的是遵循指定的协议列表中的每个协议的类型，或者某类型的子类且遵循指定的协议列表中的每个协议的类型。协议组合类型可以被用在仅当在类型注解中标明类型，泛型参数分句以及泛型 where 分句中。

协议组合类型有下列形式:

Protocol 1 & Protocol 2

协议组合类型允许你指定一个值，该值的类型遵循多个协议的要求而不必显式定义一个新的命名型的继承自每个你想要该类型遵循的协议的协议。比如，指定一个协议组合类型 Protocol A & Protocol B & Protocol C 实际上是和定义一个新的继承自 Protocol A ，Protocol B ， Protocol C 的协议 Protocol D 是完全一样的，但不需要引入一个新名字。同理，标明一个协议组合类型 SuperClass & ProtocolA 与声明一个新类型 SubClass 继承自 SuperClass 并遵循 ProtocolA 是一样的，但不需要引入新名字。

协议组合列表中的每项元素必须是类名，协议名或协议组合类型、协议、类的类型别名。列表可以最多包含一个类。

当协议组合类型包含类型别名，就有可能同一个协议在定义中出现不止一次——重复会被忽略。比如说，下面的 PQR 定义等价于 P & Q & R 。

```
1  typealias PQ = P & Q
2  typealias PQR = PQ & Q & R
```

> 协议组合类型的语法
>
> protocol-composition-type → type-identifier `&` protocol-composition-continuation
>
> protocol-composition-continuation → type-identifier protocol-composition-type

元类型

元类型指的是所有类型的类型，包括类类型、结构体类型、枚举类型和协议类型

类、结构体或枚举类型的元类型是类型名字后紧跟 .Type 。协议类型的元类型——并不是运行时遵循该协议的具体类型——是的该协议名字紧跟 .Protocol 。例如，类类型 SomeClass 的元类型就是 SomeClass.Type ，协议类型 SomeProtocol 的元类型就是 SomeProtocal.Protocol 。

你可以使用后缀 self 表达式来获取类型作为一个值。比如说， SomeClass.self 返回 SomeClass 本身，而不是 SomeClas 的一个实例。并且 SomeProtocol.self 返回 SomeProtocol 本身，而不是运行时遵循 SomeProtocol 的某个类型的实例。你可以对类型的实例使用 dynamicType 表达式来获取该实例的动态运行时的类型，如下例所示：

```
class SomeBaseClass {
    class func printClassName() {
        print("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        print("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass()
// The compile-time type of someInstance is SomeBaseClass,
// and the runtime type of someInstance is SomeSubClass
someInstance.dynamicType.printClassName()
// Prints "SomeSubClass"
```

可以使用特征运算符(=== 和 !==)来测试一个实例的运行时类型和它的编译时类型是否一致。

```
if someInstance.dynamicType === someInstance.self {
    print("The dynamic and static type of someInstance are the same")
} else {
    print("The dynamic and static type of someInstance are different")
}
// Prints "The dynamic and static type of someInstance are different"
```

使用初始化器表达式从类型的元类型的值构造出类型的实例。对于类实例，必须用 required 关键字标记被调用的构造器或者使用 final 关键字标记整个类。

```
class AnotherSubClass: SomeBaseClass {
    let string: String
    required init(string: String) {
        self.string = string
    }
    override class func printClassName() {
        print("AnotherSubClass")
    }
}
let metatype: AnotherSubClass.Type = AnotherSubClass.self
let anotherInstance = metatype.init(string: "some string")
```

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

Self 类型

Self 类型不是具体的类型，但能让你方便地引用当前类型而不需要重复已知的类型名称。

在协议声明或者协议成员声明中， Self 类型引用自最终遵循协议的类型。

在结构体、类或者枚举声明中， Self 类型引用自通过声明引入的类型。在类型成员声明中， Self 类型引用自那个类型。在类成员声明中， Self 仅可在以下情况中使用：

- 作为方法返回类型；
- 作为只读下标的返回类型；
- 作为只读计算属性的类型；
- 在方法的代码块内。

比如，下面的代码显示了实例方法 f ，它返回的类型是 Self 。

```
class Superclass {
    func f() -> Self { return self }
}
let x = Superclass()
print(type(of: x.f()))
// Prints "Superclass"
class Subclass: Superclass { }
let y = Subclass()
print(type(of: y.f()))
// Prints "Subclass"
let z: Superclass = Subclass()
print(type(of: z.f()))
// Prints "Subclass"


```

上面例子的最后一部分显示了 Self 引用自 z 的值的运行时 Subclass ，而不是编译时类型 Superclass 自己的类型。

在内嵌类型声明中， Self 类型引用自最内层类型声明引入的类型。

Self 类型引用自 Swift 标准库函数 type(of:) 相同的类型。使用 Self.someStaticMember 来访问当前类型的成员，与 type(of: self).someStaticMember 没区别。

> GRAMMAR OF A SELF TYPE
>
> self-type → `Self`

类型继承分句

类型继承分句用来指定一个命名类型继承哪个类和遵循的哪些协议。类型继承分句也用来指定协议的 class 要求。类型继承分句开始于冒号（：），其后是 class 要求或类型标识符列表或者两者均有。

类类型可以继承单个父类还能遵循任意数量的协议。当定义一个类时，父类的名字必须出现在类型标识符列表第一位，其后跟随该类必须遵循的任意个协议。如果一个类不继承自其他类，列表能够以协议开头。关于类继承更多的讨论和示例，参阅继承。

其它只继承或遵循协议列表的命名类型。协议类型可以继承于任意个其它协议。当一个协议类型继承于其它协议时，那些其它协议的要求集合会被整合在一起，任何继承自当前协议的类型必须遵循所有这些要求。正如协议声明（此处应有链接）中讨论的那样，可以将 class 关键字作为第一项包含在类型继承语句中来标记一个附有 class 要求的协议声明。

类型继承分句在枚举定义中可以是协议列表或者是当枚举赋值原始值给它的情况时的一个指定那些原始值的类型的一个单个命名类型。使用类型继承分句来指定其原始值类型的枚举定义的例子，参阅原始值。

> 类型继承分句的语法
> *type-inheritance-clause* → : *class-requirement* ,*t ype-inheritance-list*
> *type-inheritance-clause* → : *class-requirement*
> *type-inheritance-clause* → : *type-inheritance-list*
> *type-inheritance-list* → *type-identifier* | *type-identifier* , *type-inheritance-list*
> *class-requirement* → **class**

类型推断

Swift 广泛的使用类型推断，允许你在你的代码中忽略很多变量和表达式的类型或部分类型。比如，你可以写成 var x = 0 ，完全忽略类型，而不是写 var x: Int = 0 ——编译器会正确的推断出 x 是一个类型为 Int 的值的名字 。类似的，当整个类型可以从上下文推断出来时你可以忽略类型的一部分。例如，如果你写 let dict: Dictionary = ["A":1] ，编译器推断 dict 的类型是 Dictionary<String, Int> 。

上面的两个例子中，类型信息从表达式树的叶节点向上传向根节点。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——从根节点向下传向叶节点。下面的示例中，例如，常量 eFloat 的显式类型标注 （ :Float ）导致数字字面量 2.71828 拥有类型是 Float 而不是 Double 。

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

Swift 中的类型推断作用于单独的表达式或语句的级别。这意味所有用于推断省略的类型或表达式中的类型所必需的信息必须可以从表达式或其子表达式中的某个表达式的类型检查中获取。