翻译：takalard 校对：lifedim

泛型

本页包含内容：

泛型代码可以让你写出根据自我需求定义、适用于任何类型的，灵活且可重用的函数和类型。它的可以让你避免重复的代码，用一种清晰和抽象的方式来表达代码的意图。

泛型是 Swift 强大特征中的其中一个，许多 Swift 标准库是通过泛型代码构建出来的。事实上，泛型的使用贯穿了整本语言手册，只是你没有发现而已。例如，Swift 的数组和字典类型都是泛型集。你可以创建一个`Int`数组，也可创建一个`String`数组，或者甚至于可以是任何其他 Swift 的类型数据数组。同样的，你也可以创建存储任何指定类型的字典（dictionary），而且这些类型可以是没有限制的。

泛型所解决的问题

这里是一个标准的，非泛型函数`swapTwoInts`,用来交换两个Int值：

```
func swapTwoInts(inout a: Int, inout b: Int)
    let temporaryA = a
    a = b
```

```
    b = temporaryA
}
```
这个函数使用写入读出（in-out）参数来交换a和b的值，请参考[写入读出参数][1]。

swapTwoInts函数可以交换b的原始值到a，也可以交换a的原始值到b，你可以调用这个函数交换两个Int变量值：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// 输出 "someInt is now 107, and anotherInt is now 3"
```
swapTwoInts函数是非常有用的，但是它只能交换Int值，如果你想要交换两个String或者Double，就不得不写更多的函数，如swapTwoStrings和swapTwoDoublesfunctions，如同如下所示：

```
func swapTwoStrings(inout a: String, inout b: String)
{
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(inout a: Double, inout b: Double)
{
    let temporaryA = a
    a = b
    b = temporaryA
}
```
你可能注意到 swapTwoInts、 swapTwoStrings和swapTwoDoubles函数功能都是相同的，唯一不同之处就在于传入的变量类型不同，分别是Int、String和Double。

但实际应用中通常需要一个用处更强大并且尽可能的考虑到更多的灵活性单个函数，可以用来交换两个任何类型值，很幸运的是，泛型代码帮你解

决了这种问题。（一个这种泛型函数后面已经定义好了。）

注意： 在所有三个函数中，a和b的类型是一样的。如果a和b不是相同的类型，那它们俩就不能互换值。Swift 是类型安全的语言，所以它不允许一个String类型的变量和一个Double类型的变量互相交换值。如果一定要做，Swift 将报编译错误。

泛型函数

泛型函数可以工作于任何类型，这里是一个上面swapTwoInts函数的泛型版本，用于交换两个值：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

swapTwoValues函数主体和swapTwoInts函数是一样的，它只在第一行稍微有那么一点点不同于swapTwoInts，如下所示：

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

这个函数的泛型版本使用了占位类型名字（通常此情况下用字母T来表示）来代替实际类型名（如In、String或Doubl）。占位类型名没有提示T必须是什么类型，但是它提示了a和b必须是同一类型T，而不管T表示什么类型。只有swapTwoValues函数在每次调用时所传入的实际类型才能决定T所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的展位类型名字（T）是用尖括号括起来的（）。这个尖括号告诉 Swift 那个T是swapTwoValues函数所定义的一个类型。因为T是一个占位命名类型，Swift 不会去查找命名为T的实际类型。

swapTwoValues函数除了要求传入的两个任何类型值是同一类型外，也可以作为swapTwoInts函数被调用。每次swapTwoValues被调用，T所代

表的类型值都会传给函数。

在下面的两个例子中,T分别代表Int和String:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is
now "hello"
```

注意 上面定义的函数swapTwoValues是受swap函数启发而实现的。swap函数存在于 Swift 标准库，并可以在其它类中任意使用。如果你在自己代码中需要类似swapTwoValues函数的功能，你可以使用已存在的交换函数swap函数。

类型参数

在上面的swapTwoValues例子中，占位类型T是一种类型参数的示例。类型参数指定并命名为一个占位类型，并且紧随在函数名后面，使用一对尖括号括起来（如）。

一旦一个类型参数被指定，那么其可以被使用来定义一个函数的参数类型（如swapTwoValues函数中的参数a和b），或作为一个函数返回类型，或用作函数主体中的注释类型。在这种情况下，被类型参数所代表的占位类型不管函数任何时候被调用，都会被实际类型所替换（在上面swapTwoValues例子中，当函数第一次被调用时，T被Int替换，第二次调用时，被String替换。）。

你可支持多个类型参数，命名在尖括号中，用逗号分开。

命名类型参数

在简单的情况下，泛型函数或泛型类型需要指定一个占位类型（如上面的`swapTwoValues`泛型函数，或一个存储单一类型的泛型集，如数组），通常用一单个字母`T`来命名类型参数。不过，你可以使用任何有效的标识符来作为类型参数名。

如果你使用多个参数定义更复杂的泛型函数或泛型类型，那么使用更多的描述类型参数是非常有用的。例如，Swift 字典（Dictionary）类型有两个类型参数，一个是键，另外一个是值。如果你自己写字典，你或许会定义这两个类型参数为`KeyType`和`ValueType`，用来记住它们在你的泛型代码中的作用。

注意 请始终使用大写字母开头的驼峰式命名法（例如`T`和`KeyType`）来给类型参数命名，以表明它们是类型的占位符，而非类型值。

泛型类型

通常在泛型函数中，Swift 允许你定义你自己的泛型类型。这些自定义类、结构体和枚举作用于任何类型，如同`Array`和`Dictionary`的用法。

这部分向你展示如何写一个泛型集类型--`Stack`（栈）。一个栈是一系列值域的集合，和`Array`（数组）类似，但其是一个比 Swift 的`Array`类型更多限制的集合。一个数组可以允许其里面任何位置的插入/删除操作，而栈，只允许在集合的末端添加新的项（如同*push*一个新值进栈）。同样的一个栈也只能从末端移除项（如同*pop*一个值出栈）。

注意 栈的概念已被`UINavigationController`类使用来模拟试图控制器的导航结构。你通过调用`UINavigationController`的`pushViewController:animated:`方法来为导航栈添加（add）新的试图控制器；而通过`popViewControllerAnimated:`的方法来从导航栈中移除（pop）某个试图控制器。每当你需要一个严格的`后进先出`方式来管理集合，堆栈都是最实用的模型。
下图展示了一个栈的压栈(push)/出栈(pop)的行为：

![此处输入图片的描述][2]

1. 现在有三个值在栈中；
2. 第四个值“pushed”到栈的顶部；
3. 现在有四个值在栈中，最近的那个在顶部；
4. 栈中最顶部的那个项被移除，或称之为“popped”；
5. 移除掉一个值后，现在栈又重新只有三个值。

这里展示了如何写一个非泛型版本的栈，`Int`值型的栈：

```
struct IntStack {
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

这个结构体在栈中使用一个`Array`性质的`items`存储值。`Stack`提供两个方法：`push`和`pop`，从栈中压进一个值和移除一个值。这些方法标记为可变的，因为他们需要修改（或转换）结构体的`items`数组。

上面所展现的`IntStack`类型只能用于`Int`值，不过，其对于定义一个泛型`Stack`类（可以处理任何类型值的栈）是非常有用的。

这里是一个相同代码的泛型版本：

```
struct Stack<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

注意到`Stack`的泛型版本基本上和非泛型版本相同，但是泛型版本的占位

类型参数为T代替了实际Int类型。这种类型参数包含在一对尖括号里（<T>），紧随在结构体名字后面。

T定义了一个名为“某种类型T”的节点提供给后来用。这种将来类型可以在结构体的定义里任何地方表示为“T”。在这种情况下，T在如下三个地方被用作节点：

- 创建一个名为items的属性，使用空的T类型值数组对其进行初始化；
- 指定一个包含一个参数名为item的push方法，该参数必须是T类型；
- 指定一个pop方法的返回值，该返回值将是一个T类型值。

当创建一个新单例并初始化时， 通过用一对紧随在类型名后的尖括号里写出实际指定栈用到类型，创建一个Stack实例，同创建Array和Dictionary一样：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// 现在栈已经有4个string了
```

下图将展示stackOfStrings如何push这四个值进栈的过程：

从栈中pop并移除值"cuatro"：

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now
contains 3 strings
```

下图展示了如何从栈中pop一个值的过程：

由于Stack是泛型类型，所以在 Swift 中其可以用来创建任何有效类型的栈，这种方式如同Array和Dictionary。

类型约束

swapTwoValues函数和Stack类型可以作用于任何类型，不过，有的时候对使用在泛型函数和泛型类型上的类型强制约束为某种特定类型是非常有用的。类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

例如，Swift 的Dictionary类型对作用于其键的类型做了些限制。在[字典][5]的描述中，字典的键类型必须是可哈希，也就是说，必须有一种方法可以使其是唯一的表示。Dictionary之所以需要其键是可哈希是为了以便于其检查其是否包含某个特定键的值。如无此需求，Dictionary即不会告诉是否插入或者替换了某个特定键的值，也不能查找到已经存储在字典里面的给定键值。

这个需求强制加上一个类型约束作用于Dictionary的键上，当然其键类型必须遵循Hashable协议（Swift 标准库中定义的一个特定协议）。所有的 Swift 基本类型（如String，Int，Double和Bool）默认都是可哈希。

当你创建自定义泛型类型时，你可以定义你自己的类型约束，当然，这些约束要支持泛型编程的强力特征中的多数。抽象概念如可哈希具有的类型特征是根据他们概念特征来界定的，而不是他们的直接类型特征。

类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U:
SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

上面这个假定函数有两个类型参数。第一个类型参数T，有一个需要T必须是SomeClass子类的类型约束；第二个类型参数U，有一个需要U必须遵循

SomeProtocol协议的类型约束。

类型约束行为

这里有个名为findStringIndex的非泛型函数，该函数功能是去查找包含一给定String值的数组。若查找到匹配的字符串，findStringIndex函数返回该字符串在数组中的索引值（Int），反之则返回nil：

```
func findStringIndex(array: String[], valueToFind:
String) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

findStringIndex函数可以作用于查找一字符串数组中的某个字符串：

```
let strings = ["cat", "dog", "llama", "parakeet",
"terrapin"]
if let foundIndex = findStringIndex(strings, "llama")
{
    println("The index of llama is \(foundIndex)")
}
// 输出 "The index of llama is 2"
```

如果只是针对字符串而言查找在数组中的某个值的索引，用处不是很大，不过，你可以写出相同功能的泛型函数findIndex，用某个类型T值替换掉提到的字符串。

这里展示如何写一个你或许期望的findStringIndex的泛型版本findIndex。请注意这个函数仍然返回Int，是不是有点迷惑呢，而不是泛型类型?那是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数不会编译，原因在例子后面会说明：

```
func findIndex<T>(array: T[], valueToFind: T) -> Int?
```

```
{
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所写的函数不会编译。这个问题的位置在等式的检查上，“`if value == valueToFind`”。不是所有的 Swift 中的类型都可以用等式符（==）进行比较。例如，如果你创建一个你自己的类或结构体来表示一个复杂的数据模型，那么 Swift 没法猜到对于这个类或结构体而言“等于”的意思。正因如此，这部分代码不能可能保证工作于每个可能的类型`T`，当你试图编译这部分代码时估计会出现相应的错误。

不过，所有的这些并不会让我们无从下手。Swift 标准库中定义了一个`Equatable`协议，该协议要求任何遵循的类型实现等式符（==）和不等符（!=）对任何两个该类型进行比较。所有的 Swift 标准类型自动支持`Equatable`协议。

任何`Equatable`类型都可以安全的使用在`findIndex`函数中，因为其保证支持等式操作。为了说明这个事实，当你定义一个函数时，你可以写一个`Equatable`类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(array: T[], valueToFind:
T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

`findIndex`中这个单个类型参数写做：`T: Equatable`，也就意味着“任何T类型都遵循`Equatable`协议”。

`findIndex`函数现在则可以成功的编译过，并且作用于任何遵循

Equatable的类型，如Double或String:

```
let doubleIndex = findIndex([3.14159, 0.1, 0.25],
9.3)
// doubleIndex is an optional Int with no value,
because 9.3 is not in the array
let stringIndex = findIndex(["Mike", "Malcolm",
"Andrea"], "Andrea")
// stringIndex is an optional Int containing a value
of 2
```

关联类型

当定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型给定作用于协议部分的类型一个节点名（或别名）。作用于关联类型上实际类型是不需要指定的，直到该协议接受。关联类型被指定为typealias关键字。

关联类型行为

这里是一个Container协议的例子，定义了一个ItemType关联类型：

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

Container协议定义了三个任何容器必须支持的兼容要求：

- 必须可能通过append方法添加一个新item到容器里；
- 必须可能通过使用count属性获取容器里items的数量，并返回一个Int值；
- 必须可能通过容器的Int索引值下标可以检索到每一个item。

这个协议没有指定容器里item是如何存储的或何种类型是允许的。这个协议只指定三个任何遵循Container类型所必须支持的功能点。一个遵循的

类型也可以提供其他额外的功能，只要满足这三个条件。

任何遵循`Container`协议的类型必须指定存储在其里面的值类型，必须保证只有正确类型的items可以加进容器里，必须明确可以通过其下标返回item类型。

为了定义这三个条件，`Container`协议需要一个方法指定容器里的元素将会保留，而不需要知道特定容器的类型。`Container`协议需要指定任何通过`append`方法添加到容器里的值和容器里元素是相同类型，并且通过容器下标返回的容器元素类型的值的类型是相同类型。

为了达到此目的，`Container`协议声明了一个ItemType的关联类型，写作`typealias ItemType`。The protocol does not define what ItemType is an alias for—that information is left for any conforming type to provide（这个协议不会定义`ItemType`是遵循类型所提供的何种信息的别名）。尽管如此，`ItemType`别名支持一种方法识别在一个容器里的items类型，以及定义一种使用在`append`方法和下标中的类型，以便保证任何期望的`Container`的行为是强制性的。

这里是一个早前IntStack类型的非泛型版本，适用于遵循Container协议：

```
struct IntStack: Container {
    // original IntStack implementation
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias ItemType = Int
    mutating func append(item: Int) {
        self.push(item)
    }
    var count: Int {
    return items.count
```

```
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

IntStack类型实现了Container协议的所有三个要求，在IntStack类型的每个包含部分的功能都满足这些要求。

此外，IntStack指定了Container的实现，适用的ItemType被用作Int类型。对于这个Container协议实现而言，定义 typealias ItemType = Int，将抽象的ItemType类型转换为具体的Int类型。

感谢Swift类型参考，你不用在IntStack定义部分声明一个具体的Int的ItemType。由于IntStack遵循Container协议的所有要求，只要通过简单的查找append方法的item参数类型和下标返回的类型，Swift就可以推断出合适的ItemType来使用。确实，如果上面的代码中你删除了typealias ItemType = Int这一行，一切仍旧可以工作，因为它清楚的知道ItemType使用的是何种类型。

你也可以生成遵循Container协议的泛型Stack类型：

```
struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }
    var count: Int {
    return items.count
    }
```

```
    subscript(i: Int) -> T {
        return items[i]
    }
}
```
这个时候，占位类型参数T被用作append方法的item参数和下标的返回类型。Swift 因此可以推断出被用作这个特定容器的ItemType的T的合适类型。

扩展一个存在的类型为一指定关联类型

在[使用扩展来添加协议兼容性][6]中有描述扩展一个存在的类型添加遵循一个协议。这个类型包含一个关联类型的协议。

Swift的Array已经提供append方法，一个count属性和通过下标来查找一个自己的元素。这三个功能都达到Container协议的要求。也就意味着你可以扩展Array去遵循Container协议，只要通过简单声明Array适用于该协议而已。如何实践这样一个空扩展，在[使用扩展来声明协议的采纳][7]中有描述这样一个实现一个空扩展的行为：

```
extension Array: Container {}
```
如同上面的泛型Stack类型一样，Array的append方法和下标保证Swift可以推断出ItemType所使用的适用的类型。定义了这个扩展后，你可以将任何Array当作Container来使用。

Where 语句

[类型约束][8]中描述的类型约束确保你定义关于类型参数的需求和一泛型函数或类型有关联。

对于关联类型的定义需求也是非常有用的。你可以通过这样去定义*where*语句作为一个类型参数队列的一部分。一个where语句使你能够要求一个关联类型遵循一个特定的协议，以及（或）那个特定的类型参数和关联类型可以是相同的。你可写一个where语句，通过紧随放置where关键字在类型参数队列后面，其后跟着一个或者多个针对关联类型的约束，以及

（或）一个或多个类型和关联类型的等于关系。

下面的列子定义了一个名为`allItemsMatch`的泛型函数，用来检查是否两个`Container`单例包含具有相同顺序的相同元素。如果匹配到所有的元素，那么返回一个为`true`的`Boolean`值，反之，则相反。

这两个容器可以被检查出是否是相同类型的容器（虽然它们可以是），但他们确实拥有相同类型的元素。这个需求通过一个类型约束和`where`语句结合来表示：

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType:
Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool
{

        // check that both containers contain the
same number of items
        if someContainer.count !=
anotherContainer.count {
            return false
        }

        // check each pair of items to see if they
are equivalent
        for i in 0..someContainer.count {
            if someContainer[i] !=
anotherContainer[i] {
                return false
            }
        }

        // all items match, so return true
        return true

}
```

这个函数用了两个参数：`someContainer`和`anotherContainer`。`someContainer`参数是类型`C1`，`anotherContainer`参数是类型`C2`。`C1`和`C2`是容器的两个占位类型参数，决定了这个函数何时被调用。

这个函数的类型参数列紧随在两个类型参数需求的后面：

- `C1`必须遵循`Container`协议 (写作 `C1: Container`)。
- `C2`必须遵循`Container`协议 (写作 `C2: Container`)。
- `C1`的`ItemType`同样是C2的`ItemType` （写作 `C1.ItemType == C2.ItemType`） 。
- `C1`的`ItemType`必须遵循`Equatable`协议 (写作 `C1.ItemType: Equatable`)。

第三个和第四个要求被定义为一个`where`语句的一部分，写在关键字`where`后面，作为函数类型参数链的一部分。

这些要求意思是：

`someContainer`是一个`C1`类型的容器。 `anotherContainer`是一个`C2`类型的容器。 `someContainer`和`anotherContainer`包含相同的元素类型。 `someContainer`中的元素可以通过不等于操作(`!=`)来检查它们是否彼此不同。

第三个和第四个要求结合起来的意思是`anotherContainer`中的元素也可以通过 `!=` 操作来检查，因为他们在`someContainer`中元素确实是相同的类型。

这些要求能够使`allItemsMatch`函数比较两个容器，即便他们是不同的容器类型。

`allItemsMatch`首先检查两个容器是否拥有同样数目的items，如果他们的元素数目不同，没有办法进行匹配，函数就会`false`。

检查完之后，函数通过`for-in`循环和半闭区间操作（..）来迭代`someContainer`中的所有元素。对于每个元素，函数检查是否`someContainer`中的元素不等于对应的`anotherContainer`中的元素，如果这两个元素不等，则这两个容器不匹配，返回`false`。

如果循环体结束后未发现没有任何的不匹配，那表明两个容器匹配，函数返回`true`。

这里演示了allItemsMatch函数运算的过程：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    println("All items match.")
} else {
    println("Not all items match.")
}
// 输出 "All items match."
```

上面的例子创建一个`Stack`单例来存储`String`，然后压了三个字符串进栈。这个例子也创建了一个`Array`单例，并初始化包含三个同栈里一样的原始字符串。即便栈和数组否是不同的类型，但他们都遵循`Container`协议，而且他们都包含同样的类型值。你因此可以调用`allItemsMatch`函数，用这两个容器作为它的参数。在上面的例子中，`allItemsMatch`函数正确的显示了所有的这两个容器的`items`匹配。