Swift 协议 Swift 访问控制

Swift 泛型

Swift 提供了泛型让你写出灵活且可重用的函数和类型。

Swift 标准库是通过泛型代码构建出来的。

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。

以下实例是一个非泛型函数 exchange 用来交换两个 Int 值：

实例

```
// 定义一个交换两个变量的函数
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
let temporaryA = a
a = b
b = temporaryA
}
var numb1 = 100
var numb2 = 200
print("交换前数据: \(numb1) 和 \(numb2)")
swapTwoInts(&numb1, &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")
```

以上程序执行输出结果为：

```
交换前数据: 100 和 200
交换后数据: 200 和 100
```

以上实例只试用与交换整数 Int 类型的变量。如果你想要交换两个 String 值或者 Double 值，就得重新写个对应的函数，例如 swapTwoStrings(_:_:) 和 swapTwoDoubles(_:_:)，如下所示：

String 和 Double 值交换函数

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

从以上代码来看，它们功能代码是相同的，只是类型上不一样，这时我们可以使用泛型，从而避免重复编写代码。

泛型使用了占位类型名（在这里用字母 T 来表示）来代替实际类型名（例如 Int、String 或 Double）。

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

swapTwoValues 后面跟着占位类型名（T），并用尖括号括起来（<T>）。这个尖括号告诉 Swift 那个 T 是 swapTwoValues(_:_:) 函数定义内的一个占位类型名，因此 Swift 不会去查找名为 T 的实际类型。

以下实例是一个泛型函数 exchange 用来交换两个 Int 和 String 值：

实例

```
// 定义一个交换两个变量的函数
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
let temporaryA = a
a = b
b = temporaryA
}
var numb1 = 100
var numb2 = 200
print("交换前数据: \(numb1) 和 \(numb2)")
swapTwoValues(&numb1, &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")
var str1 = "A"
var str2 = "B"
print("交换前数据: \(str1) 和 \(str2)")
swapTwoValues(&str1, &str2)
print("交换后数据: \(str1) 和 \(str2)")
```

以上程序执行输出结果为：

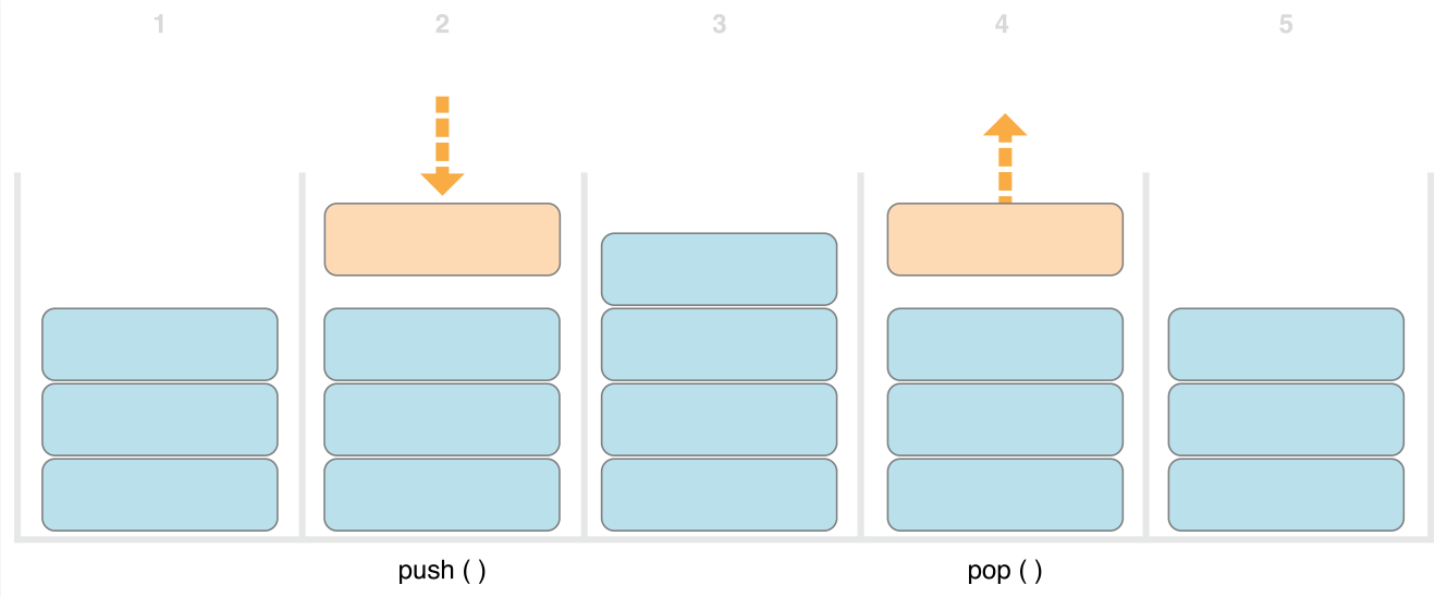
```
交换前数据:  100 和 200
交换后数据: 200 和 100
交换前数据:  A 和 B
交换后数据: B 和 A
```

泛型类型

Swift 允许你定义你自己的泛型类型。

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。

接下来我们来编写一个名为 Stack （栈）的泛型集合类型，栈只允许在集合的末端添加新的元素（称之为入栈），且也只能从末端移除元素（称之为出栈）。



图片中从左到右解析如下：

- 三个值在栈中。
- 第四个值被压入到栈的顶部。
- 现在有四个值在栈中，最近入栈的那个值在顶部。
- 栈中最顶部的那个值被移除，或称之为出栈。
- 移除掉一个值后，现在栈又只有三个值了。

以下实例是一个非泛型版本的栈，以 Int 型的栈为例：

Int 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名为 items 的 Array 属性来存储值。Stack 提供了两个方法：push(_:) 和 pop()，用来向栈中压入值以及从栈中移除值。这些方法被标记为 mutating，因为它们需要修改结构体的 items 数组。

上面的 IntStack 结构体只能用于 Int 类型。不过，可以定义一个泛型 Stack 结构体，从而能够处理任意类型的值。

下面是相同代码的泛型版本：

泛型的栈

```
struct Stack<Element> {
var items = [Element]()
mutating func push(_ item: Element) {
items.append(item)
```

```
}
mutating func pop() -> Element {
return items.removeLast()
}
}
var stackOfStrings = Stack<String>()
print("字符串元素入栈: ")
stackOfStrings.push("google")
stackOfStrings.push("runoob")
print(stackOfStrings.items);
let deletetos = stackOfStrings.pop()
print("出栈元素: " + deletetos)
var stackOfInts = Stack<Int>()
print("整数元素入栈: ")
stackOfInts.push(1)
stackOfInts.push(2)
print(stackOfInts.items);
```

实例执行结果为：

```
字符串元素入栈:
["google", "runoob"]
出栈元素: runoob
整数元素入栈:
[1, 2]
```

Stack 基本上和 IntStack 相同，占位类型参数 Element 代替了实际的 Int 类型。

以上实例中 Element 在如下三个地方被用作占位符：

- 创建 **items** 属性，使用 **Element** 类型的空数组对其进行初始化。
- 指定 **push(_:)** 方法的唯一参数 **item** 的类型必须是 **Element** 类型。
- 指定 **pop()** 方法的返回值类型必须是 **Element** 类型。

扩展泛型类型

当你扩展一个泛型类型的时候（使用 extension 关键字），你并不需要在扩展的定义中提供类型参数列表。更加方便的是，原始类型定义中声明的类型参数列表在扩展里是可以使用的，并且这些来自原始类型中的参数名称会被用作原始定义中类型参数的引用。

下面的例子扩展了泛型类型 Stack，为其添加了一个名为 topItem 的只读计算型属性，它将会返回当前栈顶端的元素而不会将其从栈中移除：

泛型

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

```
return items.removeLast()
}
}
extension Stack {
var topItem: Element? {
return items.isEmpty ? nil : items[items.count - 1]
}
}
var stackOfStrings = Stack<String>()
print("字符串元素入栈: ")
stackOfStrings.push("google")
stackOfStrings.push("runoob")
if let topItem = stackOfStrings.topItem {
print("栈中的顶部元素是: \(topItem).")
}
print(stackOfStrings.items)
```

实例中 topItem 属性会返回一个 Element 类型的可选值。当栈为空的时候，topItem 会返回 nil；当栈不为空的时候，topItem 会返回 items 数组中的最后一个元素。

以上程序执行输出结果为：

```
字符串元素入栈:
栈中的顶部元素是: runoob.
["google", "runoob"]
```

我们也可以通过扩展一个存在的类型来指定关联类型。

例如 Swift 的 Array 类型已经提供 append(_:) 方法，一个 count 属性，以及一个接受 Int 类型索引值的下标用以检索其元素。这三个功能都符合 Container 协议的要求，所以你只需简单地声明 Array 采纳该协议就可以扩展 Array。

以下实例创建一个空扩展即可:

```
extension Array: Container {}
```

类型约束

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。

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) {
    // 这里是泛型函数的函数体部分
}
```

上面这个函数有两个类型参数。第一个类型参数 T，有一个要求 T 必须是 SomeClass 子类的类型约束；第二个类型参数 U，有一个要求 U 必须符合 SomeProtocol 协议的类型约束。

实例

泛型

```
// 非泛型函数，查找指定字符串在数组中的索引
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
for (index, value) in array.enumerated() {
if value == valueToFind {
// 找到返回索引值
return index
}
}
return nil
}
let strings = ["google", "weibo", "taobao", "runoob", "facebook"]
if let foundIndex = findIndex(ofString: "runoob", in: strings) {
print("runoob 的索引为 \(foundIndex)")
}
```

索引下标从 0 开始。

以上程序执行输出结果为：

```
runoob 的索引为 3
```

关联类

Swift 中使用 associatedtype 关键字来设置关联类型实例。

下面例子定义了一个 Container 协议，该协议定义了一个关联类型 ItemType。

Container 协议只指定了三个任何遵从 Container 协议的类型必须提供的功能。遵从协议的类型在满足这三个条件的情况下也可以提供其他额外的功能。

```
// Container 协议
protocol Container {
    associatedtype ItemType
    // 添加一个新元素到容器里
    mutating func append(_ item: ItemType)
    // 获取容器中元素的数
    var count: Int { get }
    // 通过索引值类型为 Int 的下标检索到容器中的每一个元素
    subscript(i: Int) -> ItemType { get }
}

// Stack 结构体遵从 Container 协议
struct Stack<Element>: Container {
```

```
    // Stack<Element> 的原始实现部分
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // Container 协议的实现部分
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}

var tos = Stack<String>()
tos.push("google")
tos.push("runoob")
tos.push("taobao")
// 元素列表
print(tos.items)
// 元素个数
print( tos.count)
```

以上程序执行输出结果为：

```
["google", "runoob", "taobao"]
3
```

Where 语句

类型约束能够确保类型符合泛型函数或类的定义约束。

你可以在参数列表中通过where语句定义参数的约束。

你可以写一个where语句，紧跟在在类型参数列表后面，where语句后跟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间的等价(equality)关系。

实例

下面的例子定义了一个名为allItemsMatch的泛型函数，用来检查两个Container实例是否包含相同顺序的相同元素。

如果所有的元素能够匹配，那么返回 true，反之则返回 false。

泛型

```
// Container 协议
protocol Container {
associatedtype ItemType
// 添加一个新元素到容器里
mutating func append(_ item: ItemType)
// 获取容器中元素的数
var count: Int { get }
// 通过索引值类型为 Int 的下标检索到容器中的每一个元素
subscript(i: Int) -> ItemType { get }
}
// // 遵循Container协议的泛型TOS类型
struct Stack<Element>: Container {
// Stack<Element> 的原始实现部分
var items = [Element]()
mutating func push(_ item: Element) {
items.append(item)
}
mutating func pop() -> Element {
return items.removeLast()
}
// Container 协议的实现部分
mutating func append(_ item: Element) {
self.push(item)
}
var count: Int {
return items.count
}
subscript(i: Int) -> Element {
return items[i]
}
}
// 扩展，将 Array 当作 Container 来使用
extension Array: Container {}
func allItemsMatch<C1: Container, C2: Container>
(_ someContainer: C1, _ anotherContainer: C2) -> Bool
where C1.ItemType == C2.ItemType, C1.ItemType: Equatable {
// 检查两个容器含有相同数量的元素
if someContainer.count != anotherContainer.count {
return false
}
// 检查每一对元素是否相等
for i in 0..<someContainer.count {
if someContainer[i] != anotherContainer[i] {
return false
}
}
// 所有元素都匹配，返回 true
return true
}
var tos = Stack<String>()
tos.push("google")
tos.push("runoob")
```

```
tos.push("taobao")
var aos = ["google", "runoob", "taobao"]
if allItemsMatch(tos, aos) {
print("匹配所有元素")
} else {
print("元素不匹配")
}
```

以上程序执行输出结果为：

```
匹配所有元素
```

Swift 协议 Swift 访问控制