泛型

cnswift.org/generics

*泛型代码*让你能根据你所定义的要求写出可以用于任何类型的灵活的、可复用的函数。你可以编写出可复用、意图表达清晰、抽象的代码。

泛型是 Swift 最强大的特性之一，很多 Swift 标准库是基于泛型代码构建的。实际上，甚至你都没有意识到在*语言指南*中一直在使用泛型。例如，Swift 的 Array 和 Dictionary 类型都是泛型集合。你可以创建一个容纳 Int 值的数组，或者容纳 String 值的数组，甚至容纳任何 Swift 可以创建的其他类型的数组。同样，你可以创建一个存储任何指定类型值的字典，而且类型没有限制。

泛型解决的问题

下面的 swapTwoInts(_:_:) 是一个标准的非泛型函数，用于交换两个 Int 值：

```
funcswapTwoInts(_a:inoutInt,_b:inoutInt){
lettemporaryA=a
a=b
b=temporaryA
}
```

如输入输出形式参数中描述的一样，这个函数用输入输出形式参数来交换 a 和 b 的值。

swapTwoInts(_:_:) 函数把 b 原本的值给 a ，把 a 原本的值给 b 。你可以调用这个函数来交换两个 Int 变量的值。

```
varsomeInt=3
varanotherInt=107
swapTwoInts(&someInt,&anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

swapTwoInts(_:_:) 函数很实用，但是它只能用于 Int 值。如果你想交换两个 String 值，或者两个 Double 值，你只能再写更多的函数，比如下面的 swapTwoStrings(_:_:) 和 swapTwoDoubles(_:_:) 函数：

```
funcswapTwoStrings(_a:inoutString,_b:inoutString){
lettemporaryA=a
a=b
b=temporaryA
}
```

```
funcswapTwoDoubles(_a:inoutDouble,_b:inoutDouble){
lettemporaryA=a
a=b
b=temporaryA
}
```

你可能已经注意到了， swapTwoInts(_:_:) 、 swapTwoStrings(_:_:) 、 swapTwoDoubles(_:_:) 函数体是一样的。唯一的区别是它们接收值类型不同（ Int 、 String 和 Double ）。

写一个可以交换*任意*类型值的函数会更实用、更灵活。泛型代码让你能写出这样的函数。（下文中定义了这些函数的泛型版本。）

> 三个函数中， a 和 b 被定义为了相同的类型，这一点很重要。如果 a 和 b 类型不一样，不能交换它们的值。Swift是类型安全的语言，不允许（例如）一个 String 类型的变量和一个 Double 类型的变量交换值。尝试这样做会引发一个编译错误。

泛型函数

*泛型函数*可以用于任何类型。这里是上面提到的 swapTwoInts(_:_:) 函数的泛型版本，叫做 swapTwoValues(_:_:) ：

```
funcswapTwoValues<T>(_a:inoutT,_b:inoutT){
lettemporaryA=a
a=b
b=temporaryA
}
```

swapTwoValues(_:_:) 和 swapTwoInts(_:_:) 函数体是一样的。但是， swapTwoValues(_:_:) 和 swapTwoInts(_:_:) 的第一行有点不一样。下面是首行的对比：

```
funcswapTwoInts(_a:inoutInt,_b:inoutInt)
funcswapTwoValues<T>(_a:inoutT,_b:inoutT)
```

泛型版本的函数用了一个*占位符*类型名（这里叫做 T ），而不是一个*实际的*类型名（比如 Int 、 String 或 Double ）。占位符类型名没有声明 T 必须是什么样的，但是它*确实*说了 a 和 b 必须都是同一个类型 T ，或者说都是 T 所表示的类型。替代 T 实际使用的类型将在每次调用 swapTwoValues(_:_:) 函数时决定。

其他的区别是泛型函数名（ swapTwoValues(_:_:) ）后面有包在尖括号（ <T> ）里的占位符类型名（ T ）。尖括号告诉Swift， T 是一个 swapTwoValues(_:_:) 函数定义里的占位符类型名。因为 T 是一个占位符，Swift 不会查找真的叫 T 的类型。

现在，可以用调用 swapTwoInts 的方式来调用 swapTwoValues(_:_:) 函数，除此之外，可以给函数传递两个*任意*类型的值，只要两个实参的类型一致即可。每次调用 swapTwoValues(_:_:) ，用于 T 的类型会根据传入函数的值类型自动推断。

在下面的两个例子中， T 分别被推断为 Int 和 String ：

```
varsomeInt=3
varanotherInt=107
swapTwoValues(&someInt,&anotherInt)
// someInt is now 107, and anotherInt is now 3
varsomeString="hello"
varanotherString="world"
swapTwoValues(&someString,&anotherString)
// someString is now "world", and anotherString is now "hello"

```

> 上面定义的 swapTwoValues(_:_:) 函数受一个名为 swap 的泛型函数启发， swap 函数是 Swift 标准库的一部分，可以用于你的应用中。如果你需要在你自己的代码中用 swapTwoValues(_:_:) 函数的功能，可以直接用 Swift 提供的 swap(_:_:) 函数，不需要自己实现。

类型形式参数

上面的 swapTwoValues(_:_:) 中，占位符类型 T 就是一个*类型形式参数*的例子。类型形式参数指定并且命名一个占位符类型，紧挨着写在函数名后面的一对尖括号里（比如 <T> ）。

一旦你指定了一个类型形式参数，你就可以用它定义一个函数形式参数（比如 swapTwoValues(_:_:) 函数中的形式参数 a 和 b ）的类型，或者用它做函数返回值类型，或者做函数体中类型标注。在不同情况下，用调用函数时的*实际*类型来替换类型形式参数。（上面的 swapTwoValues(_:_:) 例子中，第一次调用函数的时候用 Int 替换了 T ，第二次调用是用 String 替换的。）

你可以通过在尖括号里写多个用逗号隔开的类型形式参数名，来提供更多类型形式参数。

命名类型形式参数

大多数情况下，类型形式参数的名字要有描述性，比如 Dictionary<Key,Value> 中的 Key 和 Value ，借此告知读者类型形式参数和泛型类型、泛型用到的函数之间的关系。但是，他们之间的关系没有意义时，一般按惯例用单个字母命名，比如 T 、 U 、 V ，比如上面的 swapTwoValues(_:_:) 函数中的 T 。

> 类型形式参数永远用大写开头的驼峰命名法（比如 T 和 MyTypeParameter ）命名，以指明它们是一个类型的占位符，不是一个值。

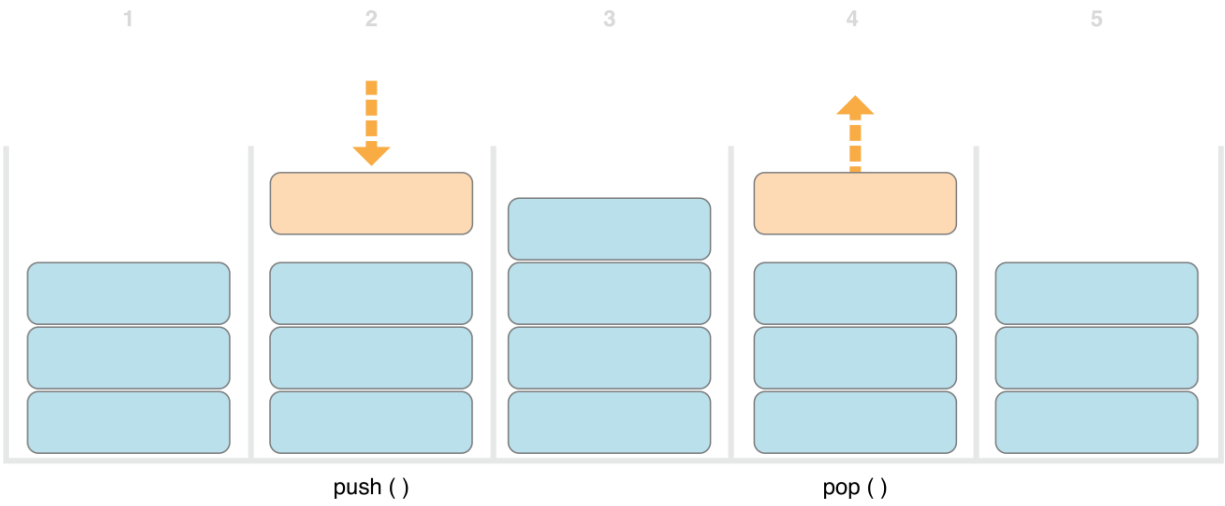
泛型类型

除了泛型函数，Swift允许你定义自己的*泛型类型*。它们是可以用于*任意*类型的自定义类、结构体、枚举，和 Array 、 Dictionary 方式类似。

本章将向你展示如何写出一个叫做 Stack 的泛型集合类型。栈是值的有序集合，和数组类似，但是比 Swift 的 Array 类型有更严格的操作限制。数组允许在其中任何位置插入和移除元素。但是，栈的新元素只能添加到集合的末尾（这就是所谓的*压栈*）。同样，栈只允许从集合的末尾移除元素（这就是所谓的*出栈*）。

> UINavigationController 类在它的导航层级关系中管理视图控制器就是用的栈的思想。你可以调用 UINavigationController 类的 pushViewController(_:animated:) 方法添加（或者说push）一个视图控制器到导航栈里，用 popViewControllerAnimated(_:) 方法从导航栈移除（或者说pop）一个视图控制器。当你需要用严格的”后进，先出”方式管理一个集合时，栈是一个很有用的集合模型。

下面的图示展示了压栈和出栈的行为：



1. 现在栈里有三个值；
2. 第四个值压到栈顶；
3. 栈里现在有四个值，最近添加的那个在顶部；
4. 栈中顶部的元素被移除，或者说叫”出栈”；
5. 移除一个元素之后，栈里又有三个元素了。

这里是如何写一个非泛型版本的栈，这种情况是一个 Int 值的栈：

```
structIntStack{
varitems=[Int]()
mutatingfuncpush(_item:Int){
items.append(item)
}
mutatingfuncpop()->Int{
returnitems.removeLast()
}
}
```

这个结构体用了一个叫做 items 的 Array 属性去存储栈中的值。 Stack 提供两个方法，push 和 pop ，用于添加和移除栈中的值。这两个方法被标记为 mutating ，是因为他们需要修改（或者说改变）结构体的 items 数组。

上面展示的 IntStack 类型只能用于 Int 值。但是定义一个*泛型* Stack 会更实用，这样可以管理*任何*类型值的栈。

这里有一个相同代码的泛型版本：

```
structStack<Element>{
varitems=[Element]()
mutatingfuncpush(_item:Element){
items.append(item)
}
mutatingfuncpop()->Element{
returnitems.removeLast()
}
}
```

注意，这个泛型的 Stack 和非泛型版本的本质上是一样的，只是用一个叫做 Element 的类型形式参数代替了实际的 Int 类型。这个类型形式参数写在一对尖括号（<Element>）里，紧跟在结构体名字后面。

Element 为稍后提供的”某类型 Element “定义了一个占位符名称。这个未来的类型可以在结构体定义内部任何位置以” Element “引用。在这个例子中，有三个地方将 Element 作为一个占位符使用：

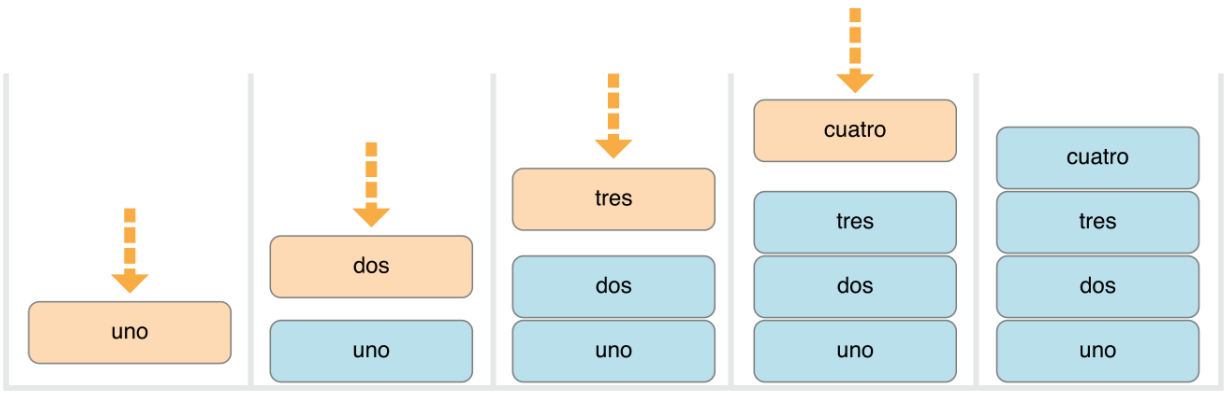
- 创建一个名为 items 的属性，用一个 Element 类型值的空数组初始化这个属性；
- 指定 push(_:) 方法有一个叫做 item 的形式参数，其必须是 Element 类型；
- 指定 pop() 方法的返回值是一个 Element 类型的值。

因为它是泛型，因此能以 Array 和 Dictionary 相似的方式，用 Stack 创建一个Swift中有效的*任意*类型的栈。

通过在尖括号中写出存储在栈里的类型，来创建一个新的 Stack 实例。例如，创建一个新的字符串栈，可以写 Stack<String>() ：

```
varstackOfStrings=Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

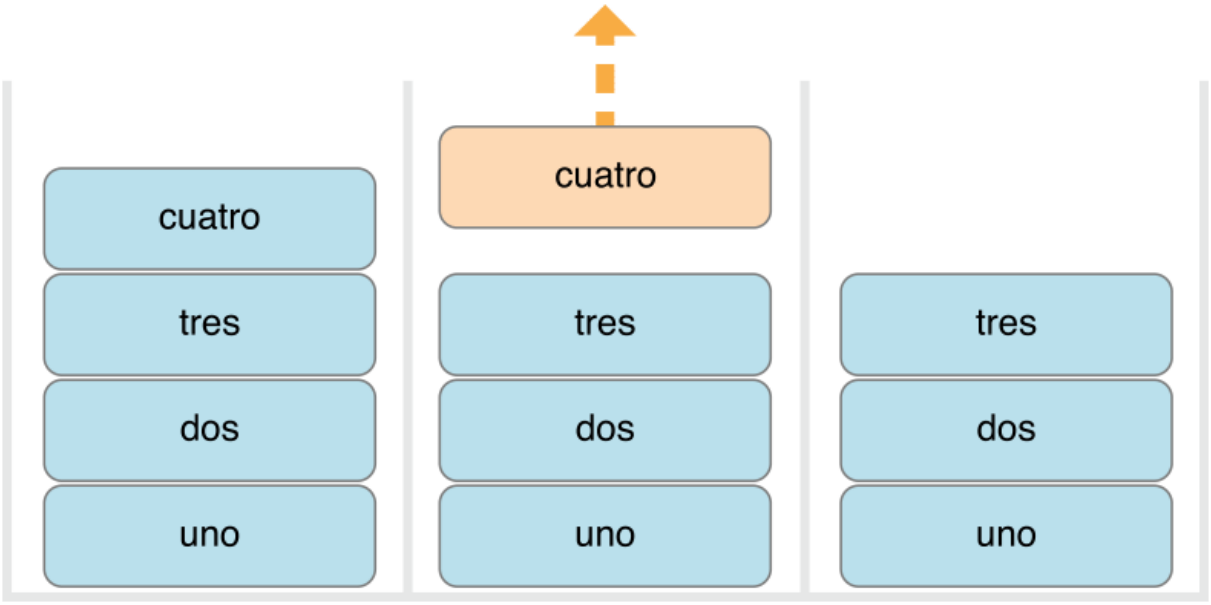
这是往栈里压入四个值之后， stackOfStrings 的图示：



从栈中移除并返回顶部的值， "cuatro" :

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

这是栈顶部的值出栈后的栈图示：



扩展一个泛型类型

当你扩展一个泛型类型时，不需要在扩展的定义中提供类型形式参数列表。*原始*类型定义的类型形式参数列表在扩展体里仍然有效，并且原始类型形式参数列表名称也用于扩展类型形式参数。

下面的例子扩展了泛型 Stack 类型，向其中添加一个叫做 topItem 的只读计算属性，不需要从栈里移除就能返回顶部的元素：

```
extensionStack{
vartopItem:Element?{
returnitems.isEmpty?nil:items[items.count-1]
}
}
```

topItem 属性返回一个 Element 类型的可选值。如果栈是空的， topItem 返回 nil ；如果栈非空， topItem 返回 items 数组的最后一个元素。

注意，这个扩展没有定义类型形式参数列表。相反，扩展中用 Stack 已有的类型形式参数名称， Element ，来指明计算属性 topItem 的可选项类型。

现在，不用移除元素，就可以用任何 Stack 实例的 topItem 计算属性来访问和查询它顶部的元素：

```
iflettopItem=stackOfStrings.topItem{
print("The top item on the stack is \(topItem).")
}
// Prints "The top item on the stack is tres."
```

类型约束

swapTwoValues(_:_:) 函数和 Stack 类型可以用于任意类型。但是，有时在用于泛型函数的类型和泛型类型上，强制其遵循特定的*类型约束*很有用。类型约束指出一个类型形式参数必须继承自特定类，或者遵循一个特定的协议、组合协议。

例如，Swift的 Dictionary 类型在可以用于字典中键的类型上设置了一个限制。如字典中描述的一样，字典键的类型必须是是可哈希的。也就是说，它必须提供一种使其可以唯一表示的方法。 Dictionary 需要它的键是可哈希的，以便它可以检查字典中是否包含一个特定键的值。没有了这个要求， Dictionary 不能区分该插入还是替换一个指定键的值，也不能在字典中查找已经给定的键的值。

这个要求通过 Dictionary 键类型上的类型约束实现，它指明了键类型必须遵循 Swift 标准库中定义的 Hashable 协议。所有 Swift 基本类型（比如 String 、 Int 、 Double 和 Bool ）默认都是可哈希的。

创建自定义泛型类型时，你可以定义你自己的类型约束，这些约束可以提供强大的泛型编程能力。像 Hashable 这样的抽象概念，根据概念上的特征，而不是确切的类型来表征类型。

类型约束语法

在一个类型形式参数名称后面放置一个类或者协议作为形式参数列表的一部分，并用冒号隔开，以写出一个类型约束。下面展示了一个泛型函数类型约束的基本语法（和泛型类型的语法相同）：

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

上面的假想函数有两个形式参数。第一个类型形式参数， T ，有一个类型约束要求 T 是 SomeClass 的子类。第二个类型形式参数， U ，有一个类型约束要求 U 遵循 SomeProtocol 协议。

类型约束的应用

这是一个叫做 findIndex(ofString:in:) 的非泛型函数，在给定的 String 值数组中查找给定的 String 值。 findIndex(ofString:in:) 函数返回一个可选的 Int 值，如果找到了给定字符串，它会返回数组中第一个匹配的字符串的索引值，如果找不到给定字符串就返回 nil ：

```
funcfindIndex(ofString valueToFind:String,inarray:[String])->Int?{
for(index,value)inarray.enumerated(){
ifvalue==valueToFind{
returnindex
}
}
returnnil
}
```

findIndex(ofString:in:) 函数可以用于字符串数组中查找字符串值：

```
letstrings=["cat","dog","llama","parakeet","terrapin"]
ifletfoundIndex=findIndex(ofString:"llama",in:strings){
print("The index of llama is \(foundIndex)")
}
// Prints "The index of llama is 2"
```

在数组中查找值的索引的原理只能用于字符串。但是，通过某种 T 类型的值代替所有用到的字符串，你可以用泛型函数写一个相同的功能。

这里写出了一个叫做 findIndex(of:in:) 的函数，可能是你期望的 findIndex(ofString:in:) 函数的一个泛型版本。注意，函数的返回值仍然是 Int？ ，因为函数返回一个可选的索引数字，而不是数组里的一个可选的值。这个函数没有编译，例子后面会解释原因：

```
funcfindIndex<T>(of valueToFind:T,inarray:[T])->Int?{
for(index,value)inarray.enumerated(){
ifvalue==valueToFind{
returnindex
}
}
returnnil
}
```

这个函数没有像上面写的那样编译。问题在于相等检查，” ifvalue==valueToFind “。Swift 中的类型不是每种都能用相等操作符（ == ）来比较的。如果你创建自己的类或者结构体去描述一个复杂的数据模型，比如说，对于那个类或结构体来说，”相等”的意义不是 Swift 能替你猜出来的。因此，不能保证这份代码可以用于*所有* T 可以表示的类型，当你尝试编译这份代码时会提示一个相应的错误。

并非无路可走，总之，Swift 标准库中定义了一个叫做 Equatable 的协议，要求遵循其协议的类型要实现相等操作符（ == ）和不等操作符（ != ），用于比较该类型的任意两个值。所有Swift标准库中的类型自动支持 Equatable 协议。

任何 Equatable 的类型都能安全地用于 findIndex(of:in:) 函数，因为可以保证那些类型支持相等操作符。为了表达这个事实，当你定义函数时将 Equatable 类型约束作为类型形式参数定义的一部分书写：

```
funcfindIndex<T:Equatable>(of valueToFind:T,inarray:[T])->Int?{
for(index,value)inarray.enumerated(){
ifvalue==valueToFind{
returnindex
}
}
returnnil
}
```

findIndex(of:in:) 的类型形式参数写作 T:Equatable ，表示”任何遵循 Equatable 协议的类型 T “。

findIndex(of:in:) 函数现在可以成功编译，并且可以用于任何 Equatable 的类型，比如 Double 或者 String ：

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

关联类型

定义一个协议时，有时在协议定义里声明一个或多个关联类型是很有用的。*关联类型*给协议中用到的类型一个占位符名称。直到采纳协议时，才指定用于该关联类型的实际类型。关联类型通过 associatedtype 关键字指定。

关联类型的应用

这里是一个叫做 Container 的示例协议，声明了一个叫做 ItemType 的关联类型：

```
protocolContainer{
associatedtype ItemType
mutatingfuncappend(_item:ItemType)
varcount: Int{get}
subscript(i:Int)->ItemType{get}
}
```

Container 协议定义了三个所有容器必须提供的功能：

- 必须能够通过 append(_:) 方法向容器中添加新元素；
- 必须能够通过一个返回 Int 值的 count 属性获取容器中的元素数量；
- 必须能够通过 Int 索引值的下标取出容器中每个元素。

这个协议没有指定元素如何储存在容器中，也没指定允许存入容器的元素类型。协议仅仅指定了想成为一个 Container 的类型，必须提供的三种功能。遵循该协议的类型可以提供其他功能，只要满足这三个要求即可。

任何遵循 Container 协议的类型必须能指定其存储值的类型。尤其是它必须保证只有正确类型的元素才能添加到容器中，而且该类型下标返回的元素类型必须是正确的。

为了定义这些要求， Container 协议需要一种在不知道容器具体类型的情况下，引用该容器将存储的元素类型的方法。 Container 协议需要指定所有传给 append(_:) 方法的值必须和容器里元素的值类型是一样的，而且容器下标返回的值也是和容器里元素的值类型相同。

为了实现这些要求， Container 协议声明了一个叫做 ItemType 的关联类型，写作 associatedtype ItemType 。协议没有定义 ItemType 是什么类型，这个信息留给遵循协议的类型去提供。但是， ItemType 这个别名，提供了一种引用 Container 中元素类型的方式，定义了一种用于 Container 方法和下标的类型，确保了任何 Container 期待的行为都得到满足。

这是前面非泛型版本的 IntStack ，使其遵循 Container 协议：

```
structIntStack: Container{
// original IntStack implementation
varitems=[Int]()
mutatingfuncpush(_item:Int){
items.append(item)
}
mutatingfuncpop()->Int{
returnitems.removeLast()
}
// conformance to the Container protocol
typealiasItemType=Int
mutatingfuncappend(_item:Int){
self.push(item)
}
varcount: Int{
returnitems.count
}
subscript(i:Int)->Int{
returnitems[i]
}
}
```

IntStack 实现了 Container 协议所有的要求，为满足这些要求，封装了 IntStack 里现有的方法。

此外， IntStack 为了实现 Container 协议，指定了适用于 ItemType 的类型是 Int 类型。typealiasItemType=Int 把 ItemType 抽象类型转换为了具体的 Int 类型。

感谢Swift的类型推断功能，你不用真的在 IntStack 定义中声明一个具体的 Int 类型 ItemType 。因为 IntStack 遵循 Container 协议的所有要求，通过简单查看 append(_:) 方法的 item 形式参数和下标的返回类型，Swift可以推断出合适的 ItemType 。如果你真的从上面的代码中删除 typealiasItemType=Int ，一切都会正常运行，因为 ItemType 该用什么类型是非常明确的。

你也可以做一个遵循 Container 协议的泛型 Stack 类型：

```
structStack<Element>: Container{
// original Stack<Element> implementation
varitems=[Element]()
mutatingfuncpush(_item:Element){
items.append(item)
}
mutatingfuncpop()->Element{
returnitems.removeLast()
}
// conformance to the Container protocol
mutatingfuncappend(_item:Element){
self.push(item)
}
varcount: Int{
returnitems.count
}
subscript(i:Int)->Element{
returnitems[i]
}
}
```

这次，类型形式参数 Element 用于 append(_:) 方法的 item 形式参数和下标的返回类型。因此，对于这个容器，Swift可以推断出 Element 是适用于 ItemType 的类型。

给关联类型添加约束

你可以在协议里给关联类型添加约束来要求遵循的类型满足约束。比如说，下面的代码定义了一个版本的 Container ，它要求容器中的元素都是可判等的。

```
protocolContainer{
associatedtype Item:Equatable
mutatingfuncappend(_item:Item)
varcount: Int{get}
subscript(i:Int)->Item{get}
}
```

要遵循这个版本的 Container ，容器的 Item 必须遵循 Equatable 协议。

在关联类型约束里使用协议

协议可以作为它自身的要求出现。比如说，这里有一个协议细化了 Container 协议，添加了一个 suffix(_:) 方法。 suffix(_:) 方法返回容器中从后往前给定数量的元素，把它们存储在一个 Suffix 类型的实例里。

```
protocolSuffixableContainer: Container{
associatedtype Suffix:SuffixableContainer whereSuffix.Item==Item
funcsuffix(_size:Int)->Suffix
}
```

在这个协议里， Suffix 是一个关联类型，就像上边例子中 Container 的 Item 类型一样。 Suffix 拥有两个约束：它必须遵循 SuffixableContainer 协议（就是当前定义的协议），以及它的 Item 类型必须是和容器里的 Item 类型相同。 Item 的约束是一个 where 分句，它在下面带有泛型 Where 分句的扩展中有讨论。

这里有一个来自闭包的循环强引用的 Stack 类型的扩展，它添加了对 SuffixableContainer 协议的遵循：

```
extensionStack: SuffixableContainer{
funcsuffix(_size:Int)->Stack{
varresult=Stack()
forindex in(count-size)..<count{
result.append(self[index])
}
returnresult
}
// Inferred that Suffix is Stack.
}
varstackOfInts=Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
letsuffix=stackOfInts.suffix(2)
// suffix contains 20 and 30
```

在上面的例子中， Suffix 是 Stack 的关联类型，也就是 Stack ，所以 Stack 的后缀运算返回另一个 Stack 。另外，遵循 SuffixableContainer 的类型可以拥有一个与它自己不同的 Suffix 类型——也就是说后缀运算可以返回不同的类型。比如说，这里有一个非泛型 IntStack 类型的扩展，它添加了 SuffixableContainer 遵循，使用 Stack<Int> 作为它的后缀类型而不是 IntStack ：

```
extensionIntStack: SuffixableContainer{
funcsuffix(_size:Int)->Stack<Int>{
varresult=Stack<Int>()
forindex in(count-size)..<count{
result.append(self[index])
}
returnresult
}
// Inferred that Suffix is Stack<Int>.
}
```

扩展现有类型来指定关联类型

你可以扩展一个现有类型使其遵循一个协议，如在扩展里添加协议遵循描述的一样。这包括一个带关联类型的协议。

Swift 的 Array 类型已经提供了 append(_:) 方法、 count 属性、用 Int 索引取出其元素的下标。这三个功能满足了 Container 协议的要求。这意味着你可以通过简单地声明 Array 采纳协议，扩展 Array 使其遵循 Container 协议。通过一个空的扩展实现，如使用扩展声明采纳协议：

```
1  extensionArray: Container{}
```

数组已有的 append(_:) 方法和下标使得Swift能为 ItemType 推断出合适的类型，就像上面的泛型 Stack 类型一样。定义这个扩展之后，你可以把任何 Array 当做一个 Container 使用。

泛型Where分句

如类型约束中描述的一样，类型约束允许你在泛型函数或泛型类型相关的类型形式参数上定义要求。

类型约束在为关联类型定义要求时也很有用。通过定义一个*泛型Where分句*来实现。泛型 Where 分句让你能够要求一个关联类型必须遵循指定的协议，或者指定的类型形式参数和关联类型必须相同。泛型 Where 分句以 Where 关键字开头，后接关联类型的约束或类型和关联类型一致的关系。泛型 Where 分句写在一个类型或函数体的左半个大括号前面。

下面的例子定义了一个叫做 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。如果所有元素都匹配，函数返回布尔值 ture ，否则返回 false 。

被检查的两个容器不一定是相同类型的（尽管它们可以是），但是它们的元素类型必须相同。这个要求通过类型约束和泛型 Where 分句一起体现：

```
1   funcallItemsMatch<C1:Container,C2:Container>
2   (_someContainer:C1,_anotherContainer:C2)->Bool
3   whereC1.ItemType==C2.ItemType,C1.ItemType: Equatable{
4   // Check that both containers contain the same number of items.
5   ifsomeContainer.count!=anotherContainer.count{
6   returnfalse
7   }
8   // Check each pair of items to see if they are equivalent.
9   foriin0..<someContainer.count{
10  ifsomeContainer[i]!=anotherContainer[i]{
11  returnfalse
12  }
13  }
14  // All items match, so return true.
15  returntrue
16  }
17
18
19
```

这个函数有两个形式参数， someContainer 和 anotherContainer 。 someContainer 形式参数是 C1 类型， anotherContainer 形式参数是 C2 类型。 C1 和 C2 是两个容器类型的类型形式参数，它们的类型在调用函数时决定。

下面是函数的两个类型形式参数上设置的要求：

- C1 必须遵循 Container 协议（写作 C1:Container ）；
- C2 也必须遵循 Container 协议（写作 C2:Container ）；
- C1 的 ItemType 必须和 C2 的 ItemType 相同（写作 C1.ItemType==C2.ItemType ）；
- C1 的 ItemType 必须遵循 Equatable 协议（写作 C1.ItemType:Equatable ）。

前两个要求定义在了函数的类型形式参数列表里，后两个要求定义在了函数的泛型 Where 分句中。

这些要求意味着：

- someContainer 是一个 C1 类型的容器；
- anotherContainer 是一个 C2 类型的容器；
- someContainer 和 anotherContainer 中的元素类型相同；
- someContainer 中的元素可以通过不等操作符（ != ）检查它们是否不一样。

后两个要求放到一起意味着， anotherContainer 中的元素*也*可以通过 != 操作符检查，因为它们和 someContainer 中的元素类型完全相同。

这些要求使得 allItemsMatch(_:_:) 函数可以比较两个容器，即使它们是不同类型的容器。

allItemsMatch(_:_:) 函数开始会先检查两个容器中的元素数量是否相同。如果它们的元素数量不同，它们不可能匹配，函数就会返回 false 。

检查完数量之后，用一个 for- in 循环和半开区间操作符（ ..< ）遍历 someContainer 中的所有元素。函数会检查 someContainer 中的每个元素，是否和 anotherContainer 中对应的元素不相等。如果两个元素不相等，则两个容器不匹配，函数返回 false 。

如果循环完成都没有出现不匹配的情况，两个容器就是匹配的，则函数返回 true 。

这是 allItemsMatch(_:_:) 函数使用的示例：

```
varstackOfStrings=Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
vararrayOfStrings=["uno","dos","tres"]
ifallItemsMatch(stackOfStrings,arrayOfStrings){
print("All items match.")
}else{
print("Not all items match.")
}
// Prints "All items match."


```

上面的例子创建了一个 Stack 实例来存储 String 值，压到栈中三个字符串。还创建了一个 Array 实例，用三个同样字符串的字面量初始化该数组。虽然栈和数组的类型不一样，但它们都遵循 Container 协议，并且它们包含的值类型一样。因此，你可以调用 allItemsMatch(_:_:) 函数，用那两个容器做函数的形式参数。上面的例子中，allItemsMatch(_:_:) 函数正确地报告了两个容器中所有元素匹配。

带有泛型 Where 分句的扩展

你同时也可以使用泛型的 where 分句来作为扩展的一部分。下面的泛型 Stack 结构体的扩展了先前的栗子，添加了一个 isTop(_:) 方法。

```
extensionStack whereElement: Equatable{
funcisTop(_item:Element)->Bool{
guard lettopItem=items.lastelse{
returnfalse
}
returntopItem==item
}
}
```

这个新的 isTop(_:) 方法首先校验栈不为空，然后对比给定的元素与栈顶元素。如果你尝试不使用泛型 where 分句来做这个，你可能会遇到一个问题： isTop(_:) 的实现要使用 == 运算符，但 Stack 的定义并不需要其元素可相等，所以使用 == 运算符会导致运行时错误。使用泛型 where 分句则允许你给扩展添加一个新的要求，这样扩展只会在栈内元素可判等的时候才给栈添加 isTop(_:) 方法。

这是用法：

```
ifstackOfStrings.isTop("tres"){
print("Top element is tres.")
}else{
print("Top element is something else.")
}
// Prints "Top element is tres."
```

如果尝试在元素不能判等的栈调用 isTop(_:) 方法，你就会出发运行时错误。

```
structNotEquatable{}
varnotEquatableStack=Stack<NotEquatable>()
letnotEquatableValue=NotEquatable()
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue)// Error
```

你可以使用泛型 where 分句来扩展到一个协议。下面的栗子把先前的 Container 协议扩展添加了一个 startsWith(_:) 方法。

```
extensionContainer whereItem: Equatable{
funcstartsWith(_item:Item)->Bool{
returncount>=1&&self[0]==item
}
}
```

startsWith(_:) 方法首先确保容器拥有至少一个元素，然后它检查第一个元素是否与给定元素相同。这个新的 startsWith(_:) 方法可以应用到任何遵循 Container 协议的类型上，包括前面我们用的栈和数组，只要容器的元素可以判等。

```
if[9,9,9].startsWith(42){
print("Starts with 42.")
}else{
print("Starts with something else.")
}
// Prints "Starts with something else."
```

上边例子中的泛型 where 分句要求 Item 遵循协议，但你同样可以写一个泛型 where 分句来要求 Item 为特定类型。比如：

```
extensionContainer whereItem==Double{
funcaverage()->Double{
varsum=0.0
forindex in0..<count{
sum+=self[index]
}
returnsum/Double(count)
}
}
print([1260.0,1200.0,98.6,37.0].average())
// Prints "648.9"
```

这个栗子在 Item 是 Double 时给容器添加了 average() 方法。它遍历容器中的元素来把它们相加，然后除以容器的总数来计算平均值。它显式地把总数从 Int 转为 Double 来允许浮点除法。

你可以在一个泛型 where 分句中包含多个要求来作为扩展的一部分，就如同你在其它地方写的泛型 where 分句一样。每一个需求用逗号分隔。

关联类型的泛型 Where 分句

你可以在关联类型中包含一个泛型 where 分句。比如说，假定你想要做一个包含遍历器的 Container ，比如标准库中 Sequence 协议那样。那么你会这么写：

```
protocolContainer{
associatedtype Item
mutatingfuncappend(_item:Item)
varcount: Int{get}
subscript(i:Int)->Item{get}
associatedtype Iterator:IteratorProtocol whereIterator.Element==Item
funcmakeIterator()->Iterator
}

```

Iterator 中的泛型 where 分句要求遍历器以相同的类型遍历容器内的所有元素，无论遍历器是什么类型。 makeIterator() 函数提供了容器的遍历器的访问。

对于一个继承自其他协议的协议来说，你可以通过在协议的声明中包含泛型 where 分句来给继承的协议中关联类型添加限定。比如说，下面的代码声明了一个 ComparableContainer 协议，它要求 Item 遵循 Comparable ：

```
protocolComparableContainer:Container whereItem: Comparable{}
```

泛型下标

下标可以是泛型，它们可以包含泛型 where 分句。你可以在 subscript 后用尖括号来写类型占位符，你还可以在下标代码块花括号前写泛型 where 分句。举例来说：

```
extensionContainer{
subscript<Indices:Sequence>(indices:Indices)->[Item]
whereIndices.Iterator.Element==Int{
varresult=[Item]()
forindexinindices{
result.append(self[index])
}
returnresult
}
}
```

这个 Container 协议的扩展添加了一个接收一系列索引并返回包含给定索引元素的数组。这个泛型下标有如下限定：

- 在尖括号中的泛型形式参数 Indices 必须是遵循标准库中 Sequence 协议的某类型；
- 下标接收单个形式参数， indices ，它是一个 Indices 类型的实例；
- 泛型 where 分句要求序列的遍历器必须遍历 Int 类型的元素。这就保证了序列中的索引都是作为容器索引的相同类型。

合在一起，这些限定意味着传入的 indices 形式参数是一个整数的序列。