泛型

*通用代码*使您能够编写灵活的，可重用的函数和类型，它们可以与任何类型一起工作，并且符合您定义的要求。您可以编写避免重复的代码，并以清晰抽象的方式表达其意图。

泛型是Swift最强大的特性之一，Swift标准库的大部分都是用泛型代码构建的。实际上，即使您没有意识到，您在整个“ *语言指南”中*都一直在使用泛型。例如，Swift `Array`和`Dictionary`类型都是泛型集合。你可以创建一个包含`Int`值的数组，或者一个包含值的数组`String`，或者确实可以在Swift中创建任何其他类型的数组。同样，您可以创建一个字典来存储任何指定类型的值，并且对该类型可以是什么没有限制。

泛型求解的问题

这是一个标准的非泛型函数`swapTwoInts(_:_:)`，它可以交换两个`Int`值：

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

该功能利用了在出参数交换的值a和b，如在描述的In-Out参数。

该`swapTwoInts(_:_:)`函数将原始值交换b为a，并将原始值交换a为b。你可以调用这个函数来交换两个`Int`变量的值：

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

该`swapTwoInts(_:_:)`功能很有用，但它只能与`Int`值一起使用。如果你想交换两个`String`值或两个`Double`值，你必须编写更多的函数，比如下面的函数`swapTwoStrings(_:_:)`和`swapTwoDoubles(_:_:)`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
}
```

您可能已经注意到的尸体`swapTwoInts(_:_:)`，`swapTwoStrings(_:_:)`和`swapTwoDoubles(_:_:)`功能是相同的。唯一的区别是该值的，他们接受的类型（`Int`，`String`，和`Double`）。

编写一个交换*任何*类型的两个值的单个函数更有用，而且更灵活。通用代码使您能够编写这样的功能。（这些函数的通用版本定义如下。）

> 注意
>
> 在所有三个函数中，类型a和b必须是相同的。如果a和b不是同一类型，则不可能交换它们的值。Swift是一种类型安全的语言，不允许（例如）类型`String`的变量和类型的变量`Double`相互交换值。试图这样做会导致编译时错误。

通用函数

*通用函数*可以处理任何类型。这是swapTwoInts(_:_:)上面函数的一个通用版本，叫做swapTwoValues(_:_:)：

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

所述的主体swapTwoValues(_:_:)的功能是相同的身体swapTwoInts(_:_:)功能。但是，第一行与第一行swapTwoValues(_:_:)稍有不同swapTwoInts(_:_:)。以下是第一行比较的方式：

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

该函数的通用版本使用*占位符*的类型名（称为T，在这种情况下），而不是一个*实际的*类型名称（例如Int，String或Double）。占位符类型名字就不说了什么什么T必须的，但它*确实*说，双方a并b必须是同一类型的T，不管T代表。T每次swapTwoValues(_:_:)调用函数时都会确定要使用的实际类型。

泛型函数和非泛型函数之间的另一个区别在于泛型函数的名称（swapTwoValues(_:_:)）后面跟着T尖括号（<T>）中的占位符类型名称（）。括号告诉Swift，它T是swapTwoValues(_:_:)函数定义中的占位符类型名称。因为T是一个占位符，所以Swift不会寻找一个实际的类型T。

swapTwoValues(_:_:)现在可以按照相同的方式调用 该函数swapTwoInts，只要它可以传递*任何*类型的两个值，只要这两个值的类型相同即可。每次swapTwoValues(_:_:)调用时，要使用的类型T都是从传递给该函数的值的类型推断出来的。

在下面的两个例子中，T被推断为Int和String分别为：

```
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
>
> swapTwoValues(_:_:)上面定义的函数受名为的通用函数的启发，该函数swap是Swift标准库的一部分，并且会自动提供给您在您的应用程序中使用。如果您需要swapTwoValues(_:_:)在自己的代码中使用该函数的行为，则可以使用Swift的现有swap(_:_:)函数，而不是提供自己的实现。

类型参数

在swapTwoValues(_:_:)上面的例子中，占位符类型T是一个*类型参数*的例子。类型参数指定并命名一个占位符类型，并在一对匹配的尖括号（如<T>）之间立即写入函数名称后面。

一旦您指定一个类型参数，你可以用它来定义一个函数的参数（如类型a，并b在参数swapTwoValues(_:_:)功能），或作为函数的返回类型，或者作为函数体中的一个类型的注释。在每种情况下，只要函数被调用，类型参数就会被*实际*类型替换。（在swapTwoValues(_:_:)上面的例子中，T被替换Int的第一次调用函数，并与被替换String，它被称为第二时间）。

您可以通过在尖括号内写入多个类型参数名称来提供多个类型参数，并用逗号分隔。

命名类型参数

在大多数情况下，类型参数具有描述性的名称，如Key和Value中Dictionary<Key, Value>和Element中Array<Element>，其中讲述的类型参数和泛型类型或功能它在使用之间的关系的读者。但是，如果没有它们之间建立有意义的关系，这是传统的给它们命名使用单个字母，例如T，U和V，如T在swapTwoValues(_:_:)上述功能。

> 注意
>
> 总是给出类型参数上面的驼峰大小写名称（例如T和MyTypeParameter），以表明它们是*类型*的占位符，而不是值。

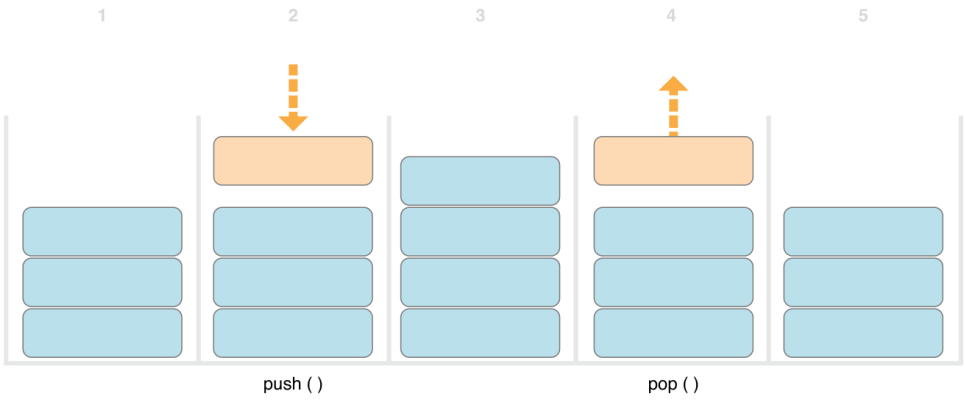
泛型类型

除了泛型函数外，Swift还允许您定义自己的*泛型类型*。这是自定义类，结构和枚举，可以一起工作*的任何*类型，以类似的方式来Array和Dictionary。

本节介绍如何编写一个名为的通用集合类型Stack。堆栈是一组有序的值，类似于数组，但具有比Swift Array类型更有限的一组操作。数组允许在数组中的任何位置插入和移除新项目。但是，堆栈允许将新项目仅附加到集合的末尾（称为*将*新值*推*入堆栈）。同样，一个堆栈允许只从集合的末尾删除项目（称为从堆栈中*弹出*一个值）。

> 注意
>
> UINavigationController该类的概念用于在其导航层次结构中对视图控制器建模。您可以调用UINavigationController类pushViewController(_:animated:)方法将视图控制器添加（或推送）到导航堆栈，以及从导航堆栈popViewControllerAnimated(_:)中删除（或弹出）视图控制器的方法。无论何时需要严格的“先进先出”方法来管理集合，堆栈都是一个有用的集合模型。

下图显示了堆栈的推送和弹出行为：



1. 目前在堆栈中有三个值。
2. 第四个值被压入栈顶。
3. 堆栈现在包含四个值，最近的一个在顶部。
4. 弹出堆栈中的顶层项目。
5. 弹出一个值后，堆栈再次保存三个值。

以下是如何编写一个非通用版本的堆栈，在这种情况下，为一堆Int值：

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

该结构使用一个Array调用的属性items将值存储在堆栈中。Stack提供了两种方法，push并pop在堆栈上和堆栈之间推送和弹出值。这些方法被标记为mutating，因为它们需要修改（或*变异*）结构的items数组。

IntStack上面显示 的类型只能与Int值一起使用，但是，定义一个*通用* Stack类可以管理*任何*类型的值的堆栈会更有用。

以下是相同代码的通用版本：

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

请注意，泛型版本Stack与非泛型版本基本相同，但是调用类型参数Element而不是实际类型Int。此类型参数写在<Element>结构名称后面的一对尖括号（）中。

Element为稍后提供的类型定义一个占位符名称。这种未来类型可以被称为Element结构定义中的任何地方。在这种情况下，Element在三个地方用作占位符：

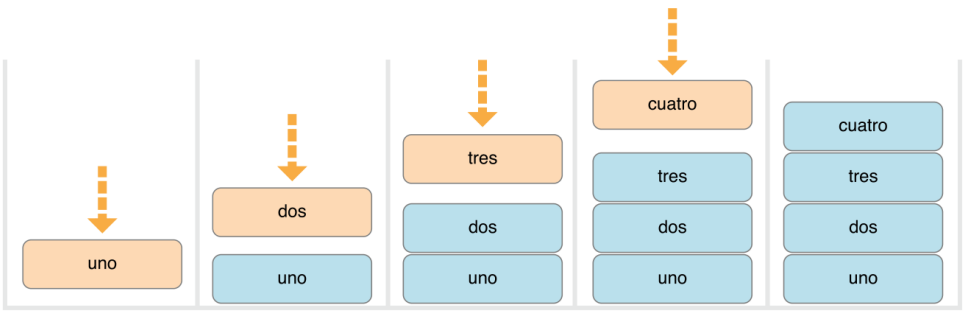
- 创建一个名为的属性items，该属性使用空的数组类型进行初始化Element
- 指定该push(_:)方法具有一个调用的单个参数item，该参数必须是类型的Element
- 指定该pop()方法返回的值将是一个类型的值Element

因为它是一个通用型，Stack可用于创建一叠*任何*斯威夫特有效的类型，以类似的方式来Array和Dictionary。

Stack通过在尖括号内写入要存储在堆栈中的类型来 创建一个新实例。例如，要创建一个新的字符串堆栈，可以这样写Stack<String>()：

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

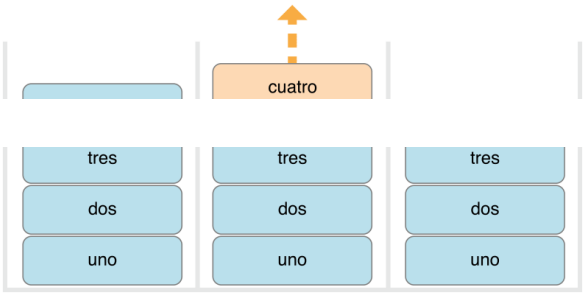
下面是stackOfStrings将这四个值推入堆栈后的外观：



从堆栈中弹出一个值将删除并返回顶部值"cuatro"：

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

以下是弹出最高值后堆栈的外观：



扩展一个通用类型

扩展泛型时，不提供类型参数列表作为扩展定义的一部分。相反，*原始*类型定义中的类型参数列表在扩展的主体中可用，并且原始类型参数名称用于引用原始定义中的类型参数。

以下示例扩展泛型`Stack`类型以添加名为的只读计算属性`topItem`，该属性返回堆栈中的顶层项目，而不弹出堆栈中的顶层项目：

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

该`topItem`属性返回一个可选的类型值`Element`。如果堆栈为空，则`topItem`返回`nil`; 如果堆栈不为空，则`topItem`返回`items`数组中的最后一项。

请注意，此扩展未定义类型参数列表。而是在扩展中使用`Stack`类型的现有类型参数名称，`Element`以指示`topItem`计算属性的可选类型。

在`topItem`计算性能，现在可以与任何使用`Stack`实例来访问，而没有删除它查询其顶端的项目。

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

通用类型的扩展还可以包括扩展类型的实例必须满足以获得新功能的要求，正如在下面的通用Where子句的扩展中所讨论的。

类型约束

该`swapTwoValues(_:_:)`功能和`Stack`类型可以与任何类型的工作。但是，对可以与泛型函数和泛型类型一起使用的类型强制执行某些*类型约束*有时很有用。类型约束指定类型参数必须从特定的类继承，或者符合特定的协议或协议组合。

例如，Swift的`Dictionary`类型对可用作字典键的类型进行了限制。如字典所述，字典键的类型必须是可*散列的*。也就是说，它必须提供一种使自己唯一可代表的方式。`Dictionary`需要它的密钥是可散列的，以便它可以检查它是否已经包含特定密钥的值。如果没有这个要求，`Dictionary`无法判断它是否应该插入或替换某个特定键的值，也不能找到字典中已有键的值。

这个要求是通过对key类型的类型约束来实施的`Dictionary`，它指定了密钥类型必须符合`Hashable`协议，这是一个在Swift标准库中定义的特殊协议。所有斯威夫特的基本类型（例如`String`，`Int`，`Double`，和`Bool`）默认情况下可哈希。

您可以在创建自定义泛型时定义自己的类型约束，这些约束提供了泛型编程的很多功能。抽象概念喜欢`Hashable`根据其概念特征来表征类型，而不是其具体类型。

类型约束语法

通过在类型参数的名称之后放置单个类或协议约束（用冒号分隔）作为类型参数列表的一部分来编写类型约束。泛型函数的类型约束的基本语法如下所示（尽管泛型类型的语法是相同的）：

```
func sc
    // 
}
```

上面的假设函数有两个类型参数。第一个类型参数，T有一个类型约束，需要T成为的子类SomeClass。第二个类型参数，U具有需要U符合协议的类型约束SomeProtocol。

类型约束在行动

这是一个非泛型函数findIndex(ofString:in:)，它被赋予一个String值来查找和String找到它的值的数组。该findIndex(ofString:in:)函数返回一个可选Int值，该值是数组中第一个匹配字符串的索引（如果找到nil该字符串），或者该字符串不可找到：

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

该findIndex(ofString:in:)函数可用于在字符串数组中找到字符串值：

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

然而，查找数组中的值的索引的原理并不仅适用于字符串。您可以通过用某种类型的值替换任何字符串来编写与通用函数相同的功能T。

以下是您可能期望写入的findIndex(ofString:in:)所谓的通用版本findIndex(of:in:)。请注意，此函数的返回类型仍然是Int?，因为函数返回一个可选的索引号，而不是数组中的可选值。不过要注意的是，这个函数不能编译，因为在这个例子之后解释的原因：

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

这个函数不像上面写的那样编译。问题在于平等检查，“ if value == valueToFind”。并非Swift中的每个类型都可以与等号运算符（==）进行比较。例如，如果您创建自己的类或结构来表示复杂的数据模型，那么对于该类或结构而言，“等于”的含义不是Swift可以为您猜测的。因此，不可能保证此代码适用于*所有*可能的类型T，并且在尝试编译代码时会报告相应的错误。

然而，所有的东西都不会丢失。Swift标准库定义了一个调用的协议Equatable，它要求任何符合类型实现等于运算符（==）和不等于运算符（!=）来比较该类型的任何两个值。所有Swift的标准类型都自动支持该Equatable协议。

任何类型Equatable的findIndex(of:in:)函数都可以安全地使用，因为它保证支持等于操作符。为了表达这个事实，Equatable当你定义函数时，你写了一个类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
```

```
            return index
        }
    }
    ret
}
```

单一类型参数`findIndex(of:in:)`写成`T: Equatable`，意思是“`T`符合`Equatable`协议的任何类型”。

该`findIndex(of:in:)`函数现在编译成功，可以用于任何类型`Equatable`，如`Double`或`String`：

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

相关类型

在定义协议时，将一个或多个关联类型声明为协议定义的一部分有时很有用。一个*相关联的类型*给出了一个占位符名称到被用作协议的一部分的类型。在采用该协议之前，不会指定用于该关联类型的实际类型。关联的类型由`associatedtype`关键字指定。

关联的类型在行动

以下是一个调用协议的示例`Container`，它声明了一个名为的关联类型`Item`：

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

该`Container`协议定义了任何容器必须提供的三种必需的功能：

- 使用方法必须可以将新项目添加到容器`append(_:)`。
- 必须可以通过`count`返回`Int`值的属性访问容器中的项目数。
- 必须可以使用带有`Int`索引值的下标来检索容器中的每个项目。

该协议并未指定容器中的项目应如何存储或允许的类型。协议只规定了任何类型为了被认为是必须提供的三个功能位`Container`。符合类型可以提供额外的功能，只要满足这三个要求即可。

任何符合`Container`协议的类型都必须能够指定它存储的值的类型。具体来说，它必须确保只有正确类型的项目才会添加到容器中，并且必须清楚其下标所返回项目的类型。

为了定义这些需求，`Container`协议需要一种方法来引用容器将容纳的元素的类型，而不知道特定容器的类型。该`Container`协议需要指定传递给该`append(_:)`方法的任何值必须具有与该容器的元素类型相同的类型，并且该容器的下标返回的值将与该容器的元素类型具有相同的类型。

为了达到这个目的，`Container`协议声明了一个名为的关联类型`Item`，写为`associatedtype Item`。该协议没有定义什么`Item`是 - 该信息留给任何符合类型提供。尽管如此，`Item`别名提供了一种方法来引用a中的项目类型`Container`，并定义一种用于`append(_:)`方法和下标的类型，以确保`Container`强制执行任何预期的行为。

以下是`IntStack`来自上述泛型类型的非泛型类型的一个版本，适用于符合`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 Item = Int
    mut
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

该IntStack类型实现了Container协议所有三个要求，并且在每种情况下都包装了该IntStack类型现有功能的一部分以满足这些要求。

而且，IntStack指定对于这个实现Container，适当Item使用是一种类型的Int。该定义typealias Item = Int将抽象类型Item转换Int为该Container协议实现的具体类型。

感谢Swift的类型推断，你实际上不需要声明具体Item的Int作为定义的一部分IntStack。因为IntStack符合Container协议的所有要求，所以Swift可以Item简单地通过查看append(_:)方法item参数的类型和下标的返回类型来推断恰当的使用。的确，如果你typealias Item = Int从上面的代码中删除了这一行，所有东西仍然有效，因为很清楚应该使用哪种类型Item。

您也可以使通用Stack类型符合Container协议：

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

这次，类型参数Element被用作append(_:)方法item参数的类型和下标的返回类型。因此，Swift可以推断这Element是Item用于这个特定容器的适当类型。

扩展现有类型以指定关联类型

您可以扩展现有类型以添加协议，如添加扩展协议一致性中所述。这包括一个关联类型的协议。

Swift的Array类型已经提供了一个append(_:)方法，一个count属性和一个带Int索引的下标来检索它的元素。这三个功能符合Container协议的要求。这意味着您可以简单地通过声明采用协议来扩展Array以符合协议。您可以使用空白扩展名来执行此操作，如使用扩展声明协议采用中所述： ContainerArray

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

Array的现有append(_:)方法和下标使Swift能够推断出适用的类型Item，就像Stack上面的泛型一样。定义这个扩展后，你可以使用任何Array一个Container。

将约束添加到关联的类型

您可以将类型约束添加到协议中的关联类型，以要求符合类型满足这些约束。例如，下面的代码定义了一个Container需要符

```
protocol Container {
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

为了符合这个版本Container，容器的Item类型必须符合Equatable协议。

在其关联类型的约束中使用协议

协议可以作为其自身要求的一部分出现。例如，这是一个改进Container协议的协议，增加了suffix(_:)方法的要求。该suffix(_:)方法从容器的末尾返回给定数量的元素，将它们存储在Suffix类型的实例中。

```
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

在这个协议中，Suffix是一个关联类型，就像上面例子中的Item类型Container。Suffix有两个约束：它必须符合SuffixableContainer协议（当前正在定义的协议），其Item类型必须与容器的Item类型相同。约束Item是一个通用的where子句，在下面的关联类型中使用通用的子句讨论。

这是上面关闭的强参考周期Stack类型的一个扩展，它增加了协议的一致性： SuffixableContainer

```
extension Stack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack {
        var result = Stack()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack.
}
var stackOfInts = Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
let suffix = stackOfInts.suffix(2)
// suffix contains 20 and 30
```

在上面的例子中，Suffix关联的类型Stack也是Stack，所以后缀操作Stack返回另一个Stack。或者，符合的类型SuffixableContainer可以具有Suffix与自身不同的类型 - 这意味着后缀操作可以返回不同的类型。例如，下面是扩展符合性的非泛型IntStack类型的扩展SuffixableContainer，使用Stack<Int>后缀类型代替IntStack：

```
extension IntStack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack<Int> {
        var result = Stack<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack<Int>.
```

```
}
```

通用条款

如类型约束中所述，类型约束使您能够定义与通用函数，下标或类型关联的类型参数的需求。

定义关联类型的需求也很有用。你通过定义一个*通用的where子句来*做到这一点。泛型where子句使您能够要求相关类型必须符合特定协议，或者某些类型参数和相关类型必须相同。通用where子句从where关键字开始，随后是关联类型的约束或类型和关联类型之间的相等关系。您where在类型或函数的主体的开始大括号之前编写通用子句。

下面的示例定义了一个名为的泛型函数allItemsMatch，它检查两个Container实例是否以相同的顺序包含相同的项目。true如果所有项目匹配，该函数将返回布尔值，如果不匹配，则返回值false。

要检查的两个容器不必是相同类型的容器（尽管它们可以），但它们必须保持相同类型的物品。此要求通过类型约束和通用where子句的组合来表达：

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

        // 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're equivalent.
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // All items match, so return true.
        return true
}
```

这个函数有两个参数叫做someContainerand anotherContainer。该someContainer参数是类型C1，以及anotherContainer参数的类型的C2。既C1和C2当调用该函数时要确定了两个容器类型是类型参数。

以下要求放在函数的两个类型参数上：

- C1必须符合Container协议（写为C1: Container）。
- C2还必须符合Container协议（写为C2: Container）。
- 该Item对C1必须相同Item的C2（写成C1.Item == C2.Item）。
- 在Item用于C1必须符合Equatable协议（写为C1.Item: Equatable）。

第一个和第二个需求在函数的类型参数列表中定义，第三个和第四个需求在函数的通用where子句中定义。

这些要求意味着：

- someContainer是一种类型的容器C1。
- anotherContainer是一种类型的容器C2。
- someContainer并anotherContainer包含相同类型的项目。
- someContainer可以使用不相等的运算符（!=）来检查 项目，看它们是否彼此不同。

第三和第四的要求相结合，意味着中的项目anotherContainer可以*也*可以与检查!=经营者，因为他们是完全一样的类型中的项目someContainer。

这些要求使allItemsMatch(_:_:)函数能够比较两个容器，即使它们是不同的容器类型。

该allItemsMatch(_:_:)功能首先检查两个容器是否包含相同数量的项目。如果它们包含不同数量的项目，则它们无法匹配，并且函数返回false。

在做这个检查后，函数someContainer用for-in循环和半开范围运算符（..<）遍历所有项。对于每个项目，函数检查项目from someContainer是否不等于in中的相应项目anotherContainer。如果两个项目不相等，则两个容器不匹配，并且函数返回false。

如果循环完成而

以下是该allItemsMatch(_:_:)功能如何起作用的功能：

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

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

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

上面的例子创建一个Stack存储String值的实例，并将三个字符串推入堆栈。该示例还创建了一个Array使用包含与堆栈相同的三个字符串的数组字面值初始化的实例。即使堆栈和阵列属于不同的类型，它们都符合Container协议，并且都包含相同类型的值。因此可以allItemsMatch(_:_:)用这两个容器作为参数来调用函数。在上面的示例中，该allItemsMatch(_:_:)函数可以正确报告两个容器中的所有项目都匹配。

扩展与通用的where子句

您也可以使用通用where子句作为扩展的一部分。下面的例子扩展了Stack前面例子的通用结构以添加一个isTop(_:)方法。

```
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```

这个新isTop(_:)方法首先检查堆栈是否为空，然后比较给定的项目和堆栈的最上面的项目。如果您尝试在没有通用where子句的情况下执行isTop(_:)此==操作，则会遇到问题：实现使用运算符，但定义Stack不要求其项目可以相等，因此使用==运算符会导致编译时错误。使用通用where子句可让您向扩展添加新的需求，以便isTop(_:)只有在堆栈中的项目可以相等时扩展才会添加该方法。

以下是该isTop(_:)方法看起来如何运作的方法：

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

如果您尝试isTop(_:)在其元素不相等的堆栈上调用该方法，则会出现编译时错误。

```
struct NotEquatable { }
var notEquatableStack = Stack<NotEquatable>()
let notEquatableValue = NotEquatable()
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue)  // Error
```

您可以使用通用where子句以及对协议的扩展。下面的例子Container从前面的例子扩展了协议来添加一个startsWith(_:)方法。

```
extensi
    func startsWith(_ item: Item) -> Bool {
        return count >= 1 && self[0] == item
    }
}
```

该startsWith(_:)方法首先确保容器至少有一个项目，然后检查容器中的第一个项目是否与给定的项目相匹配。只要容器的物品是可以平衡的，这种新startsWith(_:)方法就可以用于任何符合Container协议的类型，包括上面使用的堆栈和数组。

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

where上面示例中 的通用子句要求Item符合协议，但您也可以编写一个通用where子句，它们需要Item是特定的类型。例如：

```
extension Container where Item == Double {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
print([1260.0, 1200.0, 98.6, 37.0].average())
// Prints "648.9"
```

此示例将average()方法添加到Item类型为的容器Double。它迭代容器中的项目以将它们相加，然后除以容器的计数来计算平均值。它明确地将count从Intto Double转换为能够进行浮点除法。

您可以将多个需求包含在where作为扩展的一部分的通用子句中，就像您可以为where其他地方编写的通用子句一样。用逗号分隔列表中的每个要求。

与通用条款相关联的类型

您可以where在关联的类型中包含通用子句。例如，假设你想制作一个Container包含迭代器的版本，就像Sequence协议在标准库中使用的那样。以下是你如何写的：

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

    associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
    func makeIterator() -> Iterator
}
```

通用where子句on Iterator要求迭代器必须遍历与容器项目相同的项目类型的元素，而不管迭代器的类型如何。该makeIterator()函数提供对容器迭代器的访问。

对于从另一个协议继承的协议，通过where在协议声明中包含通用子句，可以为继承的关联类型添加约束。例如，以下代码声明了ComparableContainer需要Item符合的协议Comparable：

```
protocol ComparableContainer: Container where Item: Comparable { }
```

通用下标

在本页

下标可以是通用的，并且可以包括通用的where子句。您在后面的尖括号内写入占位符类型名称subscript，然后where在下标的正文的开始大括号之前写下一个通用子句。例如：

```
extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
```

该Container协议的扩展添加了一个下标，它接收一系列索引并返回一个包含每个给定索引处的项的数组。这个通用下标的约束如下：

- Indices尖括号中 的通用参数必须是符合Sequence标准库协议的类型。
- 下标采用单个参数，indices它是该Indices类型的一个实例。
- 通用where子句要求序列的迭代器必须遍历类型的元素Int。这确保序列中的索引与用于容器的索引相同。

总之，这些约束条件意味着为indices参数传递的值是一个整数序列。