泛型历史和概述

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程

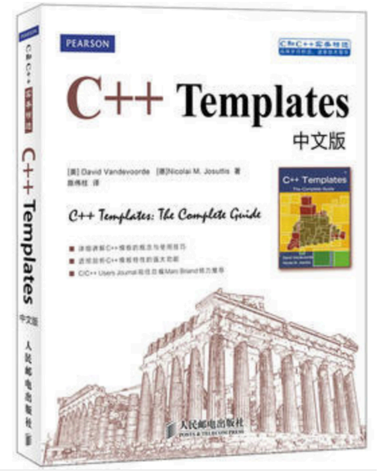
泛型发展

- 泛型程序最早出现1970年代的CLU和Ada语言中，后来被许多基于对象和面向对象的语言所采用，包括BETA、C++、D和Eiffel等。1993年C++在3.0版中引入的模版技术就属于泛型编程，1994年7月ANSI/ISO C++标准委员会通过的STL更是泛型编程的集大成者，它已被纳入1998年9月的C++标准之中。Java于2004年9月在J2SE 5.0（JDK 1.5）中开始使用泛型技术；C# 2.0和Visual Basic .NET 2005也于2005年11月采用了在微软.NET框架2.0版中所引入的泛型方法。

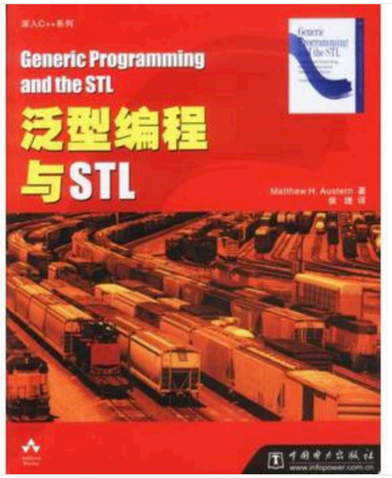
- 1971年，Dave Musser首先提出并推广了泛型编程的理论，但是主要局限于软件开发和计算机代数领域。1979年，Alexander Stepanov开始研究泛型编程，认识到泛型编程的巨大潜力，提出了STL的体系结构。

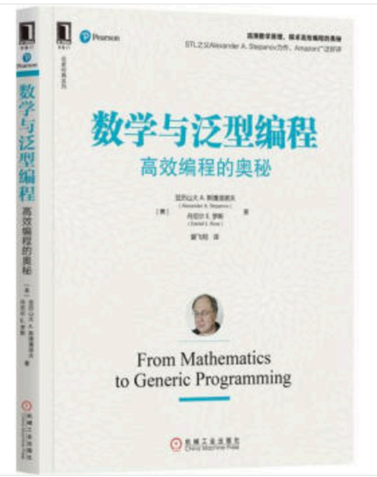
- 1993年11月，受贝尔实验室的Andrew Koening 的邀请，Stepanov在ANSI/ISO C++标准委员会的会议上，介绍了泛型编程的理论和他们的工作。Stepanov和Meng Lee按委员会的要求，于1994年3月提出了STL的草案。委员会提出了一些修改意见，其中最重要的是对关联容器的扩充，由Musser完成了扩充部分的实现工作。1994年7月，ANSI/ISO C++标准委员会终于通过了修改后的STL方案。

C++泛型


PEARSON
C++ Templates
中文版
C++ Templates: The Complete Guide
人民邮电出版社


深入实践
C++模板编程


Generic Programming
and the STL
泛型编程
与STL
Matthew H. Austern


数学与泛型编程
高效编程的奥秘
From Mathematics
to Generic Programming

C++泛型

- 面对对象库
  - MFC
- 模板库
  - STL
  - Boost

template vs generic

- 模板是C++泛型编程的基础。
- 泛型更来指一种编程思想。

为什么需要泛型

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

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

为什么需要泛型

- 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：

```
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
}
```

为什么需要泛型

- 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 ）。

泛型函数

泛型函数定义

- 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(_:_:)

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

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 。

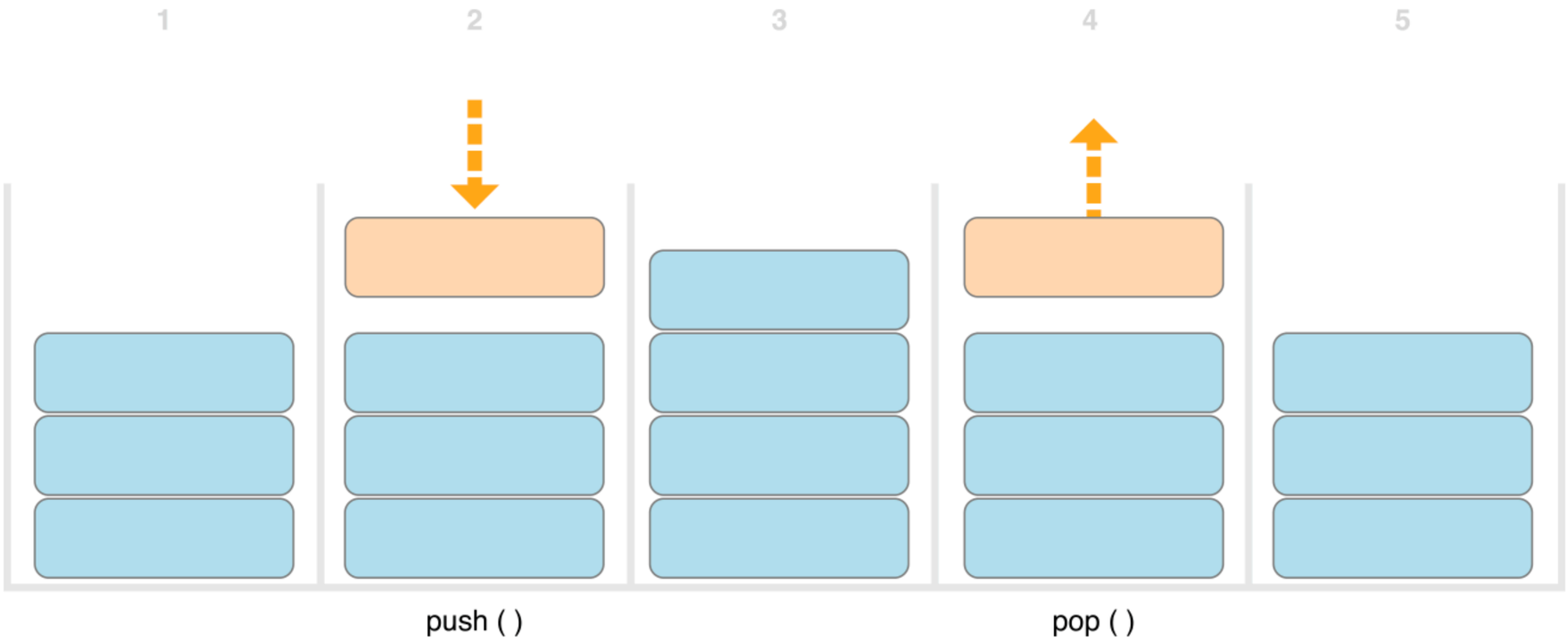
- 类型形式参数永远用大写开头的驼峰命名法（比如 T 和 MyTypeParameter ）命名，以指明它们是一个类型的占位符，不是一个值。

泛型类型

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。

泛型类型



IntStack

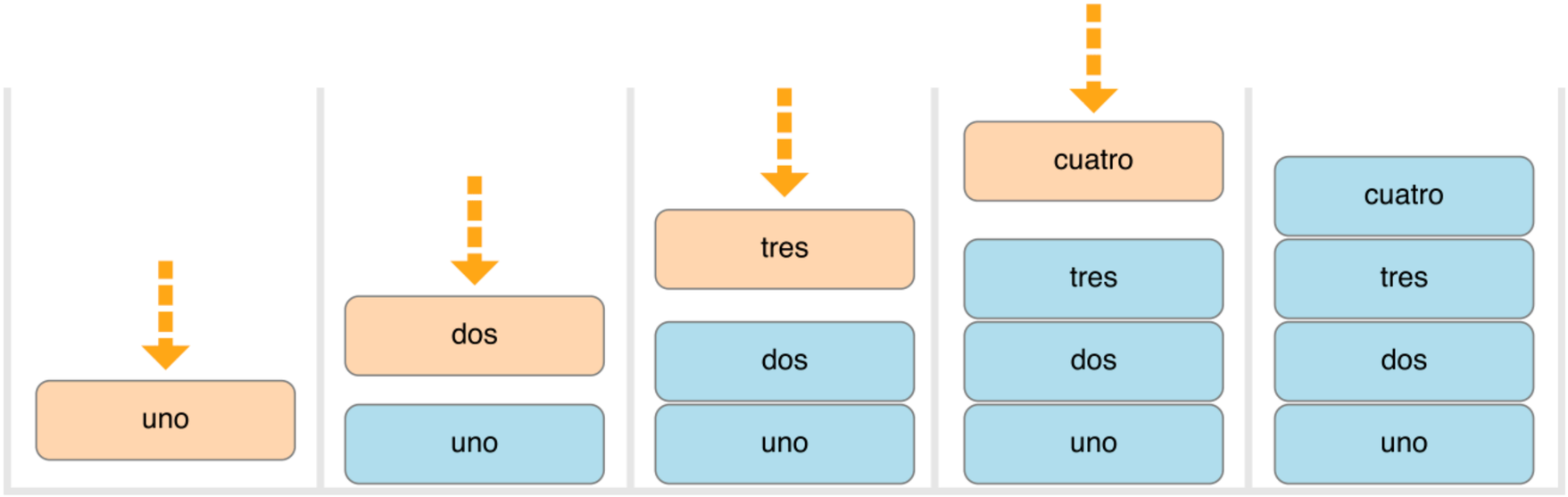
```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

泛型Stack

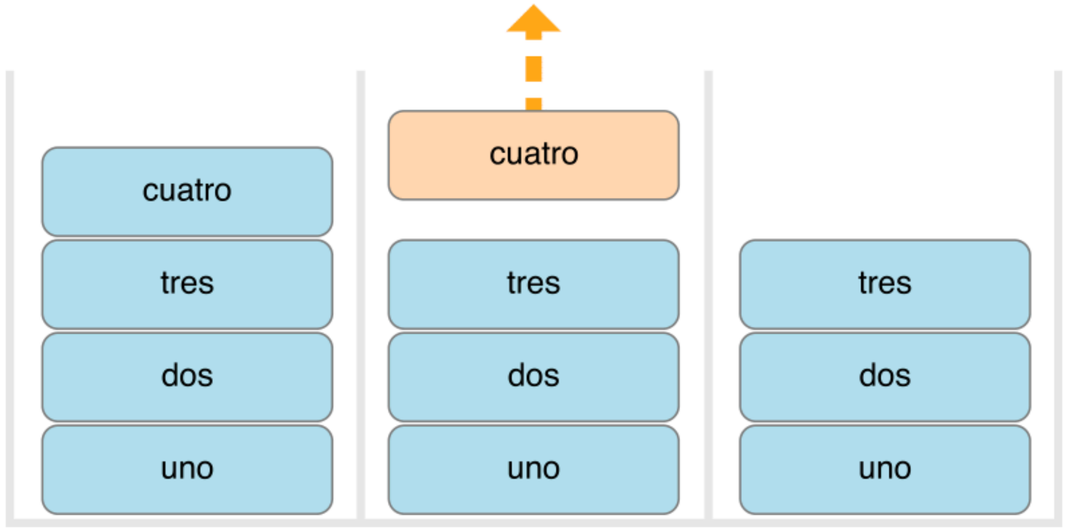
```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

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

泛型Stack


uno
dos
uno
tres
dos
uno
cuatro
tres
dos
uno
cuatro
tres
dos
uno

泛型Stack



扩展泛型类型

- 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。

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

类型约束

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。

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）：

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

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：

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

类型约束的应用

- 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

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

类型约束的应用

- 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：

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

关联类型

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。

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

关联类型的应用

```
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]
    }
}
```

关联类型的应用

```
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]
    }
}
```

关联类型的约束

- 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。

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

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。

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

泛型 where 子句

where 子句

- 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 子句来实现。泛型 Where 子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 子句以 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 子句写在一个类型或函数体的左半个大括号前面。

where 子句

- 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 ）。

```
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
}
```

where 子句

- 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。

带有泛型 Where 子句的扩展

- 你同时也可以使用泛型的 where 子句来作为扩展的一部分。

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

带有泛型 Where 子句的扩展

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

带有泛型 Where 子句的扩展

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

```
extension Container where Item == Double {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
```

关联类型的泛型 Where 子句

- 你可以在关联类型中包含一个泛型 where 子句。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。

```
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
}
```

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

泛型下标

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。

泛型下标

- 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。

```
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
    }
}
```

极客时间

泛型编程思维

泛型思维

- 面向过程的编程，可以将常用代码段封装在一个函数中，然后通过函数调用来达到目标代码重用的目的。面向对象的方法，则可以通过类的继承来实现（对象的目标）代码的重用。

- 如果需要写一个可用于不同数据类型的算法，可以采用的方法有：

  - 面向过程——对源代码进行复制和修改，生成不同数据类型版本的算法函数，调用时需要对数据类型进行手工的判断；

  - 面向对象——可以在一个类中，编写多个同名函数，它们的算法一致，但是所处理数据的类型不同，当然函数的输入参数类型也不同，可通过函数重载来自动调用对应数据类型版本的函数。

泛型思维

- 前面两种方法都需编写了多个相同算法的不同函数，不能做到代码重用。它们二者之间的主要差别，只是调用的方便与否。
- 如果采用泛型编程（例如可采用以类型作为参数的传统C++的模板技术），就可以做到源代码级的重用：
  - 泛型编程——编写以类型作为参数的一个模板函数，在调用时再将参数实例化为具体的数据类型。

泛型思维

- 狮子会跑、轮船会跑、汽车会跑、博尔特也会跑，我需要把会跑的事物集合起来？

- Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

泛型思维

- 泛型编程是一种面向算法的多态技术。
- 在计算机科学中，泛型（generic）是一种允许一个值取不同数据类型（所谓多态）的技术，强调使用这种技术的编程风格被称为泛型编程（generic programming通用编程/类属编程）。
- 泛型编程研究对软件组件的系统化组织。目标是推出一种针对算法、数据结构和内存分配机制的分类方法，以及其他能够带来高度可重用性、模块化和可用性的软件工具。

泛型思维

- 与针对问题和数据的面向对象的方法不同，泛型编程中强调的是算法。是一类通用的参数化算法，它们对各种数据类型和各种数据结构都能以相同的方式进行工作，从而实现源代码级的软件重用。

- 例如，不管（容器）是数组、队列、链表、还是堆栈，不管里面的元素（类型）是字符、整数、浮点数、还是对象，都可以使用同样的（迭代器）方法来遍历容器内的所有元素、获取指定元素的值、添加或删除元素，从而实现排序、检索、复制、合并等各种操作和算法。

- 泛型编程的通用化算法，是建立在各种抽象化基础之上的：利用参数化模版来达到数据类型的抽象化、利用容器和迭代器来达到数据结构的抽象化、利用分配器和适配器来达到存储分配和界面接口的抽象化。

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程