自定义一个阳春白雪的Collection

Back to series (/series/advanced-collections)

字号
字号
默认主题
金色主题
暗色主题

简单来说，`Collection` 就是一个包含有限个元素，并可以反复遍历的 `Sequence` 。除此之外，它还意味着你可以通过某种形式的下标来访问集合内的元素。由于，它抽象了我们对数据集合的绝大部分操作，因此，在Swift标准库里， `Collection` 得到了极为广泛的应用，我们已经用过的 `Array` ， `Dictionary` ， `Set` ， `String` 的4中不同的character view，甚至Foundation中的 `Data` ，它们都是 `Collection` 。

因此，我们有必要深入了解一下这个protocol，而了解它最好的方式，就是自己动手写一个遵从 `Collection` 的类型。

在Swift标准库里，最应该添加但却还没有实现的一类集合，就是队列。尽管，我们可以直接用 `Array` 模拟，但是，每次用它的 `remove` 方法删除数组第一个元素时，都会把剩下的所有元素移动位置，作为一个常用操作，O(n)的算法性能并不能让我们满意。因此，我们不妨自己来实现一个队列类型。

从一个自定义的protocol开始

仿照Swift的方式，首先，我们要定义一个 `protocol` 来约束队列的行为：

```
protocol Queue {
    /// The type of elements in `self`
    associatedtype Element


    /// Push an `element` into the queue
    mutating func push(_ element: Element)

    /// Pop and return an `element` out of the queue.
    /// Return `nil` if the queue is empty.
    mutating func pop() -> Element?
}
```

在 `Queue` 的定义里，我们通过 `Element` 定义了队列中存储元素的类型，并约束了仅属于队列操作的两个接口。

为了让 `Queue` 有尽可能广泛的适应性，我们甚至没有在注释中约定每个接口的算法性能，以及队列元素的访问顺序。这是一个我们在设计 `protocol` 的时候，需要不断体会和思考的事情。有时候，定义不约束什么和定义约束什么同样重要。

Queue最简单的实现

当然，提起队列，我们默认还是会想起执行先进先出逻辑的数据结构。接下来，我们就自定义一个实现 `Queue` 的集合类型：

```
struct FIFOQueue<Element>: Queue {
    fileprivate var storage: [Element] = []

    /// Push an `element` into the queue
    /// - Complexity: O(1).
    mutating func push(_ element: Element) {
        storage.append(element)
    }
}
```

在上面的代码里，为了可以仅在定义 `FIFOQueue` 的文件中访问 `storage` ，我们把它定义为了 `fileprivate` ，然后，向队列中添加元素也很简单，直接调用 `Array` 的 `append` 方法就好了。

另外，我们并不需要在 FIFOQueue 中明确定义 Queue 中的 Element，编译器可以根据 FIFOQueue 的泛型参数，来推导出 Queue.Element 的类型。但是，如果你这样定义： FIFOQueue<T>，就需要明确在 FIFOQueue<T> 的定义中，使用 typealias Element = T 了。

接下来，为了让 pop 和 push 一样，有近乎O(1)的性能，我们不能直接调用 Array.remove(at: 0)，这是个O(n)方法。一个用空间换时间的办法，就是我们再创建一个专门用于 pop 元素的数组，它是 storage 的逆序存储，这样，我们就可以通过 Array.popLast 方法直接获取最先加入队列的对象了：

```
struct FIFOQueue<Element>: Queue {
    fileprivate var operation: [Element] = []
    fileprivate var storage: [Element] = []

    /// Pop and return an `element` out of the queue.
    /// Return `nil` if the queue is empty.
    /// - Complexity: Amortized O(1)
    mutating func pop() -> Element? {
        if operation.isEmpty {
            operation = storage.reversed()
            storage.removeAll()
        }

        return operation.popLast()
    }
}
```

在 pop 的实现里，尽管 reversed() 是一个O(n)方法，但是，当我们把 reversed 的执行时间分摊到足够多次的 popLast 之后，可以认为 pop 的执行时间是个常量，并不随着数组规模的增大线性增长。因此，这是一个amortized O(1)的算法。

如何让FIFOQueue适配Collection

理解Collection protocol

实现了队列的两个核心操作之后，关于一个集合类型的其他行为，我们就可以统统交给 Collection 了。一旦让我们的 FIFOQueue 适配了 Collection，它就会立即拥有Swift为集合类型定义的数十种方法和属性。

这很好对不对？但是，当我们把 Collection 的代码刨出来看看，就会发现，似乎遵从 Collection 并不是个容易的事情。在这里，我们去掉了所有的注释，并把相关类型的约束进行了整理之后，来看看这个 protocol：

```
public protocol Collection : Indexable, Sequence {
    // Associated types
    associatedtype IndexDistance : SignedInteger = Int
    associatedtype Iterator : IteratorProtocol = IndexingIterator<Self>
    associatedtype SubSequence : IndexableBase, Sequence = Slice<Self>
    associatedtype Indices : IndexableBase, Sequence = DefaultIndices<Self
>

    // Computed properties
    public var first: Self.Iterator.Element? { get }
    public var indices: Self.Indices { get }

    public var isEmpty: Bool { get }
    public var count: Self.IndexDistance { get }

    // Subscription operator
    public subscript(position: Self.Index) -> Self.Iterator.Element { get
 }
    public subscript(bounds: Range<Self.Index>) -> Self.SubSequence { get
 }

    // Instance methods
    public func makeIterator() -> Self.Iterator

    public func prefix(upTo end: Self.Index) -> Self.SubSequence
    public func prefix(through position: Self.Index) -> Self.SubSequence

    public func suffix(from start: Self.Index) -> Self.SubSequence

    public func index(_ i: Self.Index,
        offsetBy n: Self.IndexDistance) -> Self.Inde
    public func index(_ i: Self.Index,
        offsetBy n: Self.IndexDistance,
        limitedBy limit: Self.Index) -> Self.Index?

    public func distance(from start: Self.Index,
        to end: Self.Index) -> Self.IndexDistance
}
```

是不是光看着都有点儿眼晕，粗算一下，我们需要定义4个 `associatedtype` ，4个computed properties，2个下标操作符并实现7个instance methods。

呵呵，似乎我们的to-do list太长了一点儿。但是别着急，仔细观察一下，就会发现， `Collection` 的4个 `associatedtype` 都是有默认值的，而更好的消息是基于这些特定的默认值，Swift为 `Collection` 中约束的绝大多数方法都提供了默认的实现。

例如，当 `Collection.Iterator` 的类型为默认类型时，Swift就定义了默认的 `makeIterator` 方法：

```
extension Collection where Iterator == IndexingIterator<Self> {
    public func makeIterator() -> IndexingIterator<Self> {
        return IndexingIterator(_elements: self)
    }
}
```

因此，似乎我们直接让 `FIFOQueue` 遵从 `Collection` ，然后根据编译器的报错不断补充还欠缺的内容就好了。但是，这显然不是一个让人愉快的工作，毕竟编译器提供的错误信息可不止你有哪些方法没实现。

动手完成适配

好在，Collection的官方文档 (https://developer.apple.com/reference/swift/collection)里，为我们提供了兼容指南，为了让一个类型适配 `Collection` ，我们最少做三件事情就好了：

- 定义 `startIndex` 和 `endIndex` 属性，表示集合起始和结束位置；
- 定义一个只读的下标操作符；
- 实现一个 `index(after:)` 方法用于在集合中移动索引位置；

跟着官方指引，我们来试一下：

```
extension FIFOQueue: Collection {
    public var startIndex: Int { return 0 }

    public var endIndex: Int {
        return operation.count + storage.count
    }

    public func index(after i: Int) -> Int {
        precondition(i < endIndex)

        return i + 1
    }

    public subscript(pos: Int) -> Element {
        precondition((startIndex..<endIndex).contains(pos),
                     "Out of range")

        if pos < operation.endIndex {
            return operation[operation.count - 1 - pos]
        }

        return storage[pos - operation.count]
    }
}
```

这样就可以了，相比之前 Collection 的复杂定义，我们要完成的事情实在是简单的很。现在，FIFOQueue 就已经坐拥了标准库为 Collection 类型提供的数十个API方法和属性。

我们先定义一个 FIFOQueue 对象：

```
var numberQueue = FIFOQueue<Int>()

for i in 1...10 { numberQueue.push(i) }
```

首先，我们可以直接用 for...in 遍历 FIFOQueue：

```
for i in numberQueue {
    print(i)
}
// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

其次，任何接受 Sequence 参数的方法，都可以接受 FIFOQueue 对象了：

```
var numberArray = Array<Int>()
numberArray.append(contentsOf: numberQueue)
// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

第三， FIFOQueue 有了获取集合基本信息的属性：

```
numberQueue.isEmpty // false
numberQueue.count   // 10
numberQueue.first   // Optional(1)
```

最后，我们的 FIFOQueue 也支持了之前我们在讨论集合类型的系列里提到过的各种变换函数：

```
numberQueue.map { $0 * 2 }
numberQueue.filter { $0 % 2 == 0 }
numberQueue.reduce(0, +)
```

一点小改进

至此，一切都看似很完美。但如果你回头去看看我们初始化 FIFOQueue 的方法，就会觉得先定义一个空的队列，然后用 for 循环初始化的方式有点儿土。为什么不能这样来定义队列呢?

```
var numberQueue: FIFOQueue = [1, 2, 3, 4, 5, 6]
```

当然没问题，只是，我们需要在 FIFOQueue 上多做一些工作。让它实现 ExpressibleByArrayLiteral protocol：

```
extension FIFOQueue: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        self.init(operation: elements.reversed(),
                  storage: [])
    }
}
```

很简单，我们只要把初始化的值直接倒过来放在 operation 里。然后，之前的初始化方法就可以正常工作了。这里要特别说明的是，我们用于初始化 numberQueue 的 [1, 2, 3, 4, 5, 6] 并不是一个Array，仅仅是一个Array Literal。也就是说，下面的代码，是无法通过编译的：

```
var numbers = [1, 2, 3, 4, 5, 6]
var numberQueue: FIFOQueue = numbers // !!! Error !!!
```

What's next?

通过这一节的内容，我们可以看到，让一个自定义类型适配 Collection ，要比理解 Collection 约定的各种细节容易的多。除了要明确指定 Collection 中的 Index 类型之外，我们并不需要过多关注 Collection 中的 associatedtype 以及基于这些类型要实现的方法。

但是，既然我们已经动手实现过一个完整的 Collection 了，为什么不更进一步去了解下这些被定义好的细节呢？其实，它们只是看上去有点儿复杂而已，而理解起来并不麻烦。

◀ Sequence和Iterator究竟是什么关系?
(https://www.boxueio.com/series/advanced-collections/ebook/162)

理解Collection中的associatedtype ▶
(https://www.boxueio.com/series/advanced-collections/ebook/164)