集合类型背后的“轮子”

如何为内存不连续的集合设计索引类型-I (https://www.boxueio.com/series/advanced-collections/ebook/166)

集合和集合切片为什么不是同一个类型? (https://www.boxueio.com/series/advanced-collections/ebook/168)

如何为内存不连续的集合设计索引类型-II

Back to series (/series/advanced-collections)

字号
字号
默认主题
金色主题
暗色主题

欢迎回来，在这一节，我们继续完成让 List 适配 Collection 的收尾工作。虽说是收尾工作，但也有不少细节是需要我们注意的地方。

ExpressibleByArrayLiteral

首先，是通过“数组字面值”来初始化 List：

```
extension List: ExpressibleByArrayLiteral {
    init(arrayLiteral elements: Element...) {
        startIndex = Index(node:
            elements.reversed().reduce(.end) {
                $0.insert($1)
            }, tag: elements.count)

        endIndex = Index(node: .end, tag: 0)
    }
}
```

和之前不同的是，我们要做的不再仅仅是把 elements 里的内容逆向连接起来，而是要设置 Collection 里约定的两个 Index 对象。在它的实现里可以看到：

- endIndex 表示一个 tag 为0，值为 .end 的节点；
- startIndex 则是把 elements 逆序后，所有元素串联到一起之后的表头元素，它的 tag 是 elements.count；

push和pop

接下来要修改的，是 List 在表头的两个动作，添加和删除元素，简单来说，我们只是正确移动 startIndex 的位置就好了：

```
extension List {
    mutating func push(_ value: Element) {
        startIndex = Index(
            node: startIndex.node.insert(value),
            tag: startIndex.tag + 1)
    }

    mutating func pop() -> Index {
        let ret = startIndex
        startIndex = index(after: startIndex)

        return ret
    }
}
```

CustomStringConvertible

为了稍后方便查看 List 的内容，我们还可以给 List 也添加一个 CustomStringConvertible 实现：

```
extension List: CustomStringConvertible {
    var description: String {
        let values = self.map {
                String(describing: $0)
            }.joined(separator: ", ")

        return "List: \(values)"
    }
}
```

在上面的代码里，我们先把集合中的 Element 元素变成 String ，得到了一个 Array<String> ，然后调用 Array.joined 把数组中的每个元素用 separator 分隔开，并拼接成一个完整的字符串。

修改 next 方法

由于我们让 pop 方法返回了被弹出节点的 Index ，所以，为了得到被弹出的对象，我们得把之前实现的 next 方法修改一下：

```
extension List: IteratorProtocol, Sequence {
    mutating public func next() -> Element? {
        switch pop().node {
        case .end:
            return nil
        case let .node(value, _):
            return value
        }
    }
}
```

没什么特别的，只是把 pop 的返回值解析了一下而已。

count和==

接下来，我们改进一个方法的实现： count ，虽然在适配了 Collection 之后， List 已经获得了一个免费的 count 实现，依据链表的特性，这是一个 O(n) 性能的方法。但是，由于我们在 Index 里加了 tag ，可以把它改造成一个 O(1) 的算法：

```
extension List {
    var count: Int {
        return startIndex.tag - endIndex.tag
    }
}
```

这要比从头遍历到尾计数快多了。

最后，我们来实现两个 List 对象的比较：

```
func == <T: Equatable>(
    lhs: List<T>,
    rhs: List<T>) -> Bool {
    return lhs.elementsEqual(rhs)
}
```

只要 List 中的元素实现了 Equatable protocol，我们直接调用 Sequence 中的 elementsEqual 方法比较两个 List 对象就好了。

测试List Collection

接下来，我们用一些测试代码，测试 List 的行为，在注释里，我们可以看到对应的执行结果：

```
/// Initialize
var list1: List = [1, 2, 3, 4, 5]
// List: 1, 2, 3, 4, 5

/// Iterator
var begin = list1.makeIterator()
begin.next() // Optional(1)

/// Common properties
list1.count        // 5
list1.first        // Optional(1)
list1.index(of: 5) // ListIndex.tag = 1

/// Common operations
list1.push(11)
// List: 11, 1, 2, 3, 4, 5
list1.pop()
// List: 1, 2, 3, 4, 5

/// Equality
let list2: List = [1, 2, 3, 4, 5]
list1 == list2 // true
```

最后，我们的 List 实现还有一个小缺陷，尽管我们可以比较两个 List 对象，但是如果一个泛型函数需要接受一个遵从 Equatable 的类型， List 却不适用，例如：

```
func demo<T: Equatable>(_ l: T) { }
demo(list1) // ERROR: argument type 'List<Int>' does not conform to expect
ed type 'Equatable'
```

这是因为，我们并没有让 List 遵从 Equatable ，因为一旦如此，我们就没办法要求 List.Element 也遵从 Equatable protocol了：

```
extension List: Equatable where Element: Equatable {
    // Do not support in Swift 3
}
```

我们只能期待Swift 4中，可以加入这个语言特性了。

What's next?

至此，我们就从0开始，完成了一个自定义类型适配 Collection 的过程。其实，在整个过程里，最重要的就是理解 Collection.Index 的设计，当集合类型的底层存储空间不连续的时候，我们就要花费一番心思，以保证通过 Index 索引集合内容时的性能。

现在，是时候休息一下了。在下一节中，我们将详细讨论之前提到过的一个问题：为什么集合类型的区间要使用一个独立的类型呢?

◀ 如何为内存不连续的集合设计索引类型-I
(https://www.boxueio.com/series/advanced-collections/ebook/166)

集合和集合切片为什么不是同一个类型? ▶
(https://www.boxueio.com/series/advanced-collections/ebook/168)