集合类型背后的“轮子”

实现一个Swift“风味”的链表集合

(https://www.boxueio.com/series/advanced-collections/ebook/165)

如何为内存不连续的集合设计索引类型-II

(https://www.boxueio.com/series/advanced-collections/ebook/167)

如何为内存不连续的集合设计索引类型-I

Back to series (/series/advanced-collections)

为了能让我们实现的 List 更像是一个集合类型，最简单的方式，当然就是让它适配Swift标准库中的 Collection protocol，而不是自己再动手写一遍所有的“轮子”方法。对于一个无法用 Int 来计算 Index 的集合类型，这个适配的过程有着诸多细节问题需要考量。这一节里，我们就来逐一实现这个过程。

让List是一个Sequence

在让 List 成为合格的 Collection 之前，我们得先让它成为一个合格的 Sequence ，这是万里长征的第一步。希望你还记得下面的规则：

要让一个类型适配 Sequence ，我们得做两件事情：第一，为序列实现一个遵从 IteratorProtocol 的类型，它用来向前遍历整个序列；第二，实现一个 makeIterator 方法，它返回我们第一步创建的 Iterator 。

当然，希望你还记得，如果 IteratorProtocol 自身也是一个 Sequence ，那么 Sequence 还实现了一个默认的 makeIterator 方法，当然，作用就是返回它自己。

那么，如何为我们的 List 类型选定 Iterator 呢？作为一个理论上的无限元素集合，最简单的方式就是把遍历的状态保存在链表头部，也就是说，让一个 List 是它自己的 Iterator：

```
extension List: IteratorProtocol, Sequence {
    mutating func next() -> Element? {
        return pop()
    }
}
```

怎么样，是不是比想象的简单很多？只要我们让 List 同时遵从 IteratorProtocol 和 Sequence ，那么我们连那个默认的 makeIterator 方法都可以省略掉。

然后，用下面的方法试一下：

```
let list1: List = [1, 2, 3, 4, 5]

// 1 2 3 4 5
for i in list1 {
    print(i)
}

// 1 2 3 4 5
list1.forEach { print($0) }

// true
list1.contains(1)
// true
list1.elementsEqual([1, 2, 3, 4, 5])
```

接下来，就可以动手让 List 适配 Collection 了。而这个过程里，最麻烦的事情，就是为 List 找到一个合适的 Index 类型。

理解Collection中的Index

简单来说， Collection.Index 用于表示一个位置，通过这个位置可以访问到集合中的元素。例如，我们在 FIFOQueue 和 Array 中使用的 Int ，应该说，这是我们最熟悉的Index类型。

但是，当我们访问一个 Dictionary 的时候，现在想一下，它的 Index 类型是什么呢？也许你马上就会觉得，就是它的 Key 类型啊，我们难道不是用 Dictionary[Key] 这样的形式来访问 Value 么？

实际上并不是，因为 **Collection 要求它的 Index 类型是可以移动的，通过移动这个 Index 对象，我们可以不断访问集合中后续的元素。**

但想象一下，如果我们用一个 String 作为 Index 类型，我们该如何理解移动到这个 String 对象的下一个位置呢?

所以，对于 Dictionary 来说， Index 的类型并不是 Key ，而是一个我们不会用到的内部类型，这个类型和数组的索引是类似的，它直接指向 Dictionary 存储空间的内部，可以让我们直接访问到 Dictionary 的内容。如果你不信，可以来看一点 Dictionary 的代码（我们去掉了一些和 Index 无关的内容）：

```
public struct Dictionary<Key, Value>: Collection {
    /// The index type of a dictionary.
    public typealias Index = DictionaryIndex<Key, Value>

    /// ...
}
```

看到了吧， Dictionary.Index 是一个叫做 DictionaryIndex 的类型。当然，我们的重点不是 DictionaryIndex 的实现，我们只要知道它不是 Key 也就好了。

如果你继续往下看，还会发现， Dictionary 还为 DictionaryIndex 定义了一个特别的 [] 实现：

```
public subscript(
    position: DictionaryIndex<Key, Value>
) -> (key: Key, value: Value) { get }
```

它直接返回一个tuple，包含了特定位置的key和value。而 Dictionary 为 Key 实现的 [] 是这样的：

```
public subscript(key: Key) -> Value?
```

它只是返回一个 Optional<Value> 。为什么要有这种差异呢?

除了刚才我们提到的要求 Index 可以移动之外， Collection 还要求在下标操作符中使用 Index 访问集合值的时候，要满足一个条件：既 subscript 的实现应该有O(1)的性能，因为 Dictionary 中，还有很多算法的 O(1) 性能是依赖于 [] 的实现的；

这也就是说，通过 Index 获取集合中保存的值的时候，并不需要遍历查找，也不需要计算 Index 的哈希值等，它必须是一个可以直接访问到集合中，某个特定位置值的操作；

所以，尽管我们没有看到 Dictionary.Index 版本的实现，但我们可以相信，这是一个可以快速完成的动作。

看到这里，把Swift对 Index 类型的要求总结一下，就是：

- 可以移动和计算位置；
- 要通过这个位置可以直接读取集合中的值；
- 在[]中，通过Index访问集合中的值要有O(1)的性能；

Ah...现在你该理解为什么我们会在开始提到，设计一个 Index 类型是 List 实现最麻烦的环节了吧。

设计List.Index

但有时，要求多并不是一个坏事儿，因为限制多了，也就意味着我们的选择少了。为了让 subscript 可以以O(1)的性能访问到 List 的内容，和 Iterator 一样，似乎 Index 只能让 List 自己来担当。这样，通过 Index 和通过 Iterator 访问集合的方式就很类似了。

但很可惜，这样不太行。为什么呢?

因为 Collection 和 Index 这两个类型，它们对于比较相等这个动作的理解，是完全不同的：

- 对于 Collection 来说，相等，意味着比较其中的每一个Element对象，他要求Element支持相等比较操作；
- 对于 Index 来说，相等意味着一个 Collection 对象中的同一个位置，它并不关心 Collection 中的 Element 是否支持比较操作；

于是，我们别无选择了。只能为 List 再单独设计一个 Index 类型。为此，我们还要单独设计一个 List 的节点类型 Node ，让每一个 Index 对象，都包含一个 Node ，以达到可以直接通过 Index 访问到链表节点值的效果。这听着有点儿复杂，别着急，我们一个个的来。

Node

在我们之前的 List 实现里，是没有单独的节点概念的，只有链表的概念。我们可以把一个节点，看成是只包含一个节点的链表。现在，我们要把节点的概念单独拿出来了，不过很简单，把之前的 List 改成 Node 就好了：

```
enum Node<Element> {
    case end
    indirect case node(Element, next: Node<Element>)
}

extension Node {
    func insert(_ value: Element) -> Node {
        return .node(value, next: self)
    }
}
```

ListIndex

然后，我们定义一个 ListIndex 类型，让它包含 Node ，这样就可以直接通过 ListIndex 访问链表中的元素了：

```
public struct ListIndex<Element> {
    fileprivate let node: Node<Element>
}
```

这里，我们使用了一个技巧。对于 ListIndex 来说，它有两个性质：

- 我们应该允许开发者在程序中使用这个类型来声明变量；
- 我们不应该允许开发者自己直接生成一个 ListIndex 对象，这应该是链表的私有行为。开发者不应该关心链表的内部存储形式；

所以，对于 ListIndex ，我们使用了 public 修饰，表示这个类型可以在任意范围内使用；对于它的属性，我们使用了 fileprivate 修饰。这样，我们就实现了只能看到它的类型，而无法创建 ListIndex 对象的效果了。

接下来，我们需要给 ListIndex 添加比较的功能，以确定两个 ListIndex 对象是否指向链表的同一个位置。怎么办呢?

我们可以给 ListIndex 添加一个 Int 类型的属性 tag ，然后让 endIndex 的 tag 等于0，让 startIndex 的tag等于链表中当前所有元素个数。

当我们遍历链表的时候，就可以根据表头的 tag 来计算出特定位置 ListIndex 对象的 tag 。稍后，我们会看到这个实现，现在我们先把 tag 添加到 ListIndex：

```
public struct ListIndex<Element>: CustomStringConvertible {
    fileprivate let node: Node<Element>
    fileprivate let tag: Int

    public var description: String {
        return "IndexTag: \(tag)"
    }
}
```

为了稍后方便观察 ListIndex 的 tag 值，我们还让它实现了 CustomStringConvertible protocol。

接下来，就该实现 ListIndex 的比较了，我们让 ListIndex 实现 Comparable protocol即可。它要根据 tag 的值，比较 ListIndex 对象的大小：

```
extension ListIndex: Comparable {
    public static func ==<T> (
        lhs: ListIndex<T>,
        rhs: ListIndex<T>) -> Bool {
        return lhs.tag == rhs.tag
    }

    public static func < <T>(
        lhs: ListIndex<T>,
        rhs: ListIndex<T>) -> Bool {
        return lhs.tag > rhs.tag
    }
}
```

对于 Comparable protocol来说，最低的要求是实现 == 和 < 就好了，其它的比较操作符可以根据这两个操作符推导出来。在 < 的实现里，刚才我们说过 endIndex.tag 为0， startIndex.tag 是链表的元素个数，所以，两个节点 ListIndex 的小于关系，应该是两个 tag 的大于关系。

实现List

解决了 ListIndex 的设计之后，我们终于可以着手实现 List 并让它遵从 Collection protocol了。这一步反倒是非常简单的，和我们实现 FIFOQueue 的过程如出一辙。

首先，自定义 List.Index 类型：

```
struct List<Element>: Collection {
    typealias Index = ListIndex<Element>
}
```

其次，定义 List 的 startIndex 和 endIndex 属性：

```
struct List<Element>: Collection {
    typealias Index = ListIndex<Element>

    var startIndex: Index
    let endIndex: Index
}
```

第三，实现 [] 操作符，由于我们可以通过 Index 对象直接访问其表示的节点，所以，这是一个 O(1) 操作：

```
struct List<Element>: Collection {
    /// ...

    subscript(position: Index) -> Element {
        switch position.node {
        case .end:
            fatalError("out of range")
        case let .node(value, _):
            return value
        }
    }
}
```

这里，再次强调下一个细节，我们通过 Index 作为索引得到的，应该是一个 Element 对象，而不是一个optional。

最后，实现移动 Index 的方法：

```
struct List<Element>: Collection {
    /// ...

    func index(after idx: Index) -> Index {
        switch idx.node {
        case .end:
            fatalError("out of range")
        case let .node(_, next):
            return Index(node: next, tag: idx.tag - 1)
        }
    }
}
```

在 index 的实现里，当我们向后移动移动一个位置时，新位置 Index 对象的 tag 是前一个索引的 tag-1 。

What's next ?

这样，让 List 适配 Collection 的主体工作就完成了。但还有一些之前实现的辅助方法，在下一节中，我们将逐一把它们修正过来。但现在，是时候休息一下了。

◀ 实现一个Swift"风味"的链表集合 (https://www.boxueio.com/series/advanced-collections/ebook/165)

如何为内存不连续的集合设计索引类型-II ▶ (https://www.boxueio.com/series/advanced-collections/ebook/167)

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)

QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)