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

返回视频 (/series/advanced-collections)

集合和集合切片为什么不是同一个类型?

Back to series (/series/advanced-collections)

字号
字号
默认主题
金色主题
暗色主题

之前我们也提到过，当我们访问集合中的某个区间时，得到的类型不是之前的集合类型。用我们自己创建的 `List` 举例：

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

let theSecondIndex = list1.index(after: list1.startIndex)
let slice1 = list1[theSecondIndex ..< list1.endIndex]
```

当我们通过 `theSecondIndex ..< list1.endIndex` 读取 `List` 的时候，就会发现得到的 `slice1` 的类型，并不是一个 `List` ，而是一个 `Slice<List<Int>>` 。

希望你还记得，在 `Collection` 里，有一个叫做 `SubSequence` 的 `associatedtype` ，它的默认值，就是 `Slice<Self>` 。一旦我们获取了一个 `Slice` ，就可以用它来创建各种类型的 `Collection` ，例如：

```
let array1 = Array(slice1)
// [2, 3, 4, 5]
```

这样，我们就截取了一段 `List` ，比创建了一个 `Array` 。然而，这是怎么实现的呢？为了理解这个过程，我们只能诉诸于 `Slice` 的代码了。

了解Slice的实现

为了看到 `Slice` 的实现，我们得借助于Swift源代码中的一个Python脚本，它位于： `utils/gyb.py` 。这是什么呢?

`Slice` 的源代码在 `stdlib/public/core/Slice.swift.gyb` ，其中 `gyb` 的含义是generate your boilerplate，简单来说， `.gyb` 是一份源代码的模板，我们得自己生成一个 `Slice` 的实现。

当然，这不是什么难事儿，在Swift源代码的根目录执行下面的命令：

```
python utils/gyb.py --line-directive= -o Slice.swift stdlib/public/core/Sl
ice.swift.gyb
```

就能在根目录下，生成一个叫 `Slice.swift` 的文件，我们就能进去一看究竟了。简单来说，它包含以下几个重要的部分（在这里，我们省略了注释以及一些移动 `Index` 位置的方法）：

```
public struct Slice<Base : _Indexable>
    : Collection {

    public typealias Index = Base.Index
    public typealias IndexDistance = Base.IndexDistance

    public var _startIndex: Index
    public var _endIndex: Index

    internal let _base: Base

    public init(base: Base, bounds: Range<Index>) {
        self._base = base
        self._startIndex = bounds.lowerBound
        self._endIndex = bounds.upperBound
    }

    public var startIndex: Index {
        return _startIndex
    }

    public var endIndex: Index {
        return _endIndex
    }

    /// ...
}
```

从上面的代码里，我们能读出些什么呢？其实，绝大部分内容，都是我们已经提及过了。

首先，它包含 `startIndex` 和 `endIndex`，表示 `Slice` 代表的区间位置；其次，它还有一个属性，表示它的起始和结束位置索引的原始集合。所以，这么看， `Slice` 似乎还会比它“截取”的集合对象更大一些。当然，我们可以用下面的代码验证一下：

```
MemoryLayout.size(ofValue: list1)  // 32
MemoryLayout.size(ofValue: slice1) // 64
```

从注释的结果里，可以看到， `Slice` 整整比它截取的集合大了一倍。我们简单来分析一下：对于 `List` 来说，它的大小由它的两个 `ListIndex` 属性（ `startIndex` 和 `endIndex` ）决定。

而一个 `ListIndex` 对象包含了两个属性，一个是 `tag`，它的大小是8字节，另一个是 `Node`，对于 `Node` 来说， `indirect case` 保存的是它的地址， `end case` 保存的是 `enum` 的值，它们也都需要8字节存储。所以，一个 `ListIndex` 的大小是16字节，两个 `ListIndex` 自然就是32字节了。

那么，对于 `Slice` 对象来说呢？它除了有两个 `ListIndex` 对象，还有了一个 `_base`，这是它截取的原始集合对象，也就是 `List`，而这个 `List` 刚才我们已经分析过了，大小是32字节。所以，一个 `Slice` 的大小就是两个 `ListIndex` 加上一个 `List`，也就是64字节了。

也许看到这里，这个结果会和你想象的有点儿出入。我们要为区间类型多付出一倍的空间代价，有改进的办法么？

当然有。

如何为集合类型自定义Slice

思路很简单。当我们通过一个区间“截取”集合类型时，不要让它返回 `Slice`，而是让它返回截取后的 `List` 就好了：

```
extension List {
    public subscript(
        bounds: Range<Index>) -> List<Element> {
        return List(startIndex: bounds.lowerBound,
                    endIndex: bounds.upperBound)
    }
}
```

其实这样的实现似乎更符合我们对截取集合的直觉，截取后，直接得到了一个新的自身集合类型。然后，再回头看下之前统计内存大小的代码：

```
MemoryLayout.size(ofValue: list1)  // 32
MemoryLayout.size(ofValue: slice1) // 32
```

你看，这简直是个一箭双雕的技法。我们不仅省掉了一半的空间，还可以暂时忘掉那个叫做 `Slice` 类型的存在。

不要让Slice太长时间保持集合类型的存储

在 List 的实现里，我们发现 SubSequence 和 List 是共享底层的数据存储的。实际上，Swift标准库中的很多集合类型也采取了类似的方法，例如： Array 和 String 。

但这带来了一个问题，一个范围很小的 Slice ，也会把整个集合类型保持在内存里。这是怎么发生的呢？我们看下面的例子：

首先，把之前自定义的 Slice 操作注释掉，我们用回系统默认的 Slice<List<Int>> ：

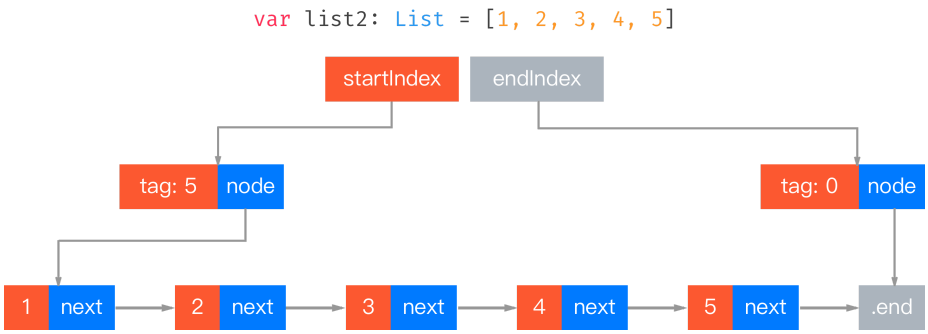
```
//extension List {
//    public subscript(bounds: Range<Index>) -> List<Element> {
//        return List(startIndex: bounds.lowerBound,
//                    endIndex: bounds.upperBound)
//    }
//}
```

然后，编写下面的代码：

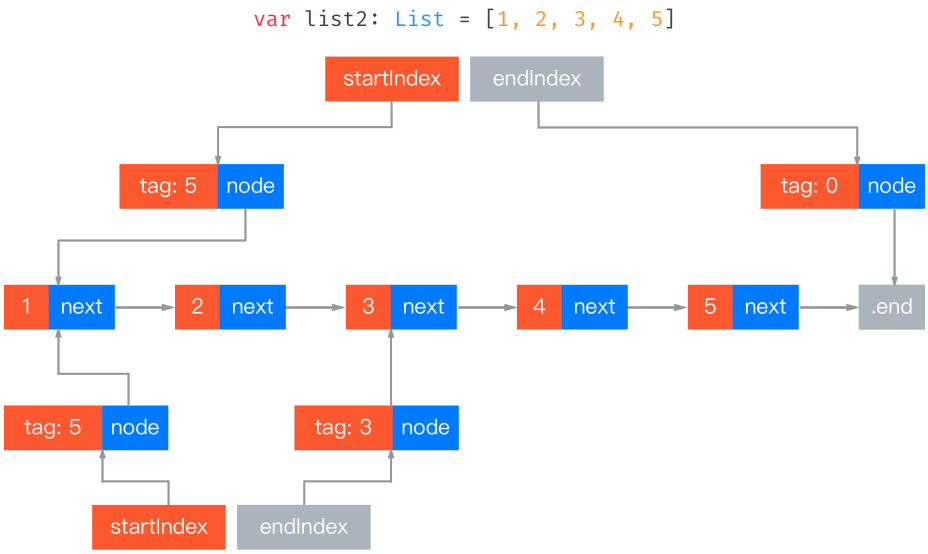
```
// Initialize
var slice12: Slice<List<Int>>!

if true {
    var list1: List = [1, 2, 3, 4, 5]
    slice12 = list1.prefix(2)
}

slice12
```

你能想象会发生什么么？我们一步步来分析一下：

首先，当创建好 list2 之后，它在内存里是这样的：



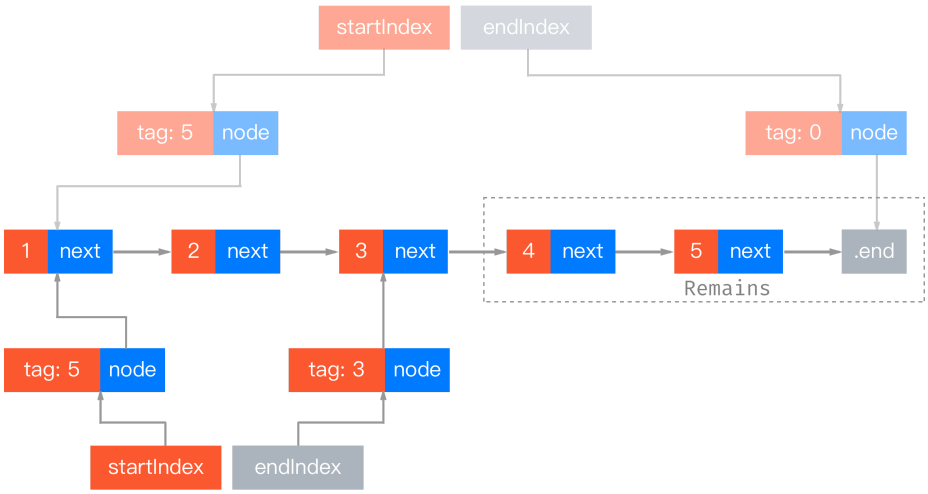
其次，我们截取了 list2 的前两个元素：




```
slice2 = list2.prefix(2)
```

第三，当离开 if 作用域之后， list2 就被回收了。但这时，截取 list2 的 Slice 还在，它在内存中，保持了集合中的前三个元素，进而， list2 中个后三个元素也不会被释放了。因此，整个 list2 的底层存储，都被 Slice 保持在了内存里。想象一下，如果你要截取1GB数据中的前1MB，这样的方式明显就是个不划算的操作。

```
var list2: List = [1, 2, 3, 4, 5]
```




```
slice2 = list2.prefix(2)
```

正是由于这个问题，连Swift在官方文档中，都特别强调了Slice的推荐用法 (https://developer.apple.com/reference/swift/slice)：

> Use slices only for transient computation. A slice may hold a reference to the entire storage of a larger collection, not just to the portion it presents, even after the base collection's lifetime ends. Long-term storage of a slice may therefore prolong the lifetime of elements that are no longer otherwise accessible, which can erroneously appear to be memory leakage.

简单来说，截取集合相关的代码应该尽可能的短，不要让 `Slice` 对象太长时间的存活。

Conclusion

以上，就是我们关于 `Sequence` 和 `Collection` 的内容，应该说，这两个 `protocol` 奠定了Swift集合类型的基础，它们为实现各种集合类型提供了灵活性和便利性。当然，这也是有代价的，就是理解这些错综复杂的约束并不容易，你的确需要自己按照它们的要求一步步的实践一下。希望这个系列的内容，是个不错的开始。

当你寻着蛛丝马迹开始一点点的掌握这套复杂的类型系统时，这些扮演轮子角色的 `protocol` 就会极大提高你的生产力。面对序列或集合类型的代码设计，当你把遵从这些 `protocol` 变成一种思考习惯时，无论是你，还是使用你代码的开发者，都会尽快上手，从中受益。

⏮ 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)

返回视频 ⏭ (/series/advanced-collections)

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/)