集合类型背后的“轮子”

从抽象顺序访问一系列数据开始

(https://www.boxueio.com/series/advanced-collections/ebook/160)

Sequence和Iterator究竟是什么关系?

(https://www.boxueio.com/series/advanced-collections/ebook/162)

两种不同拷贝语义的Iterator

Back to series (/series/advanced-collections)

字号
字号
默认主题
金色主题
暗色主题

了解了 Iterator 的基本用法之后，在这一节，我们来讨论复制 Iterator 对象时，执行的语义问题。

按值语义拷贝的一般Iterator

还是用上个视频中使用的 Fibnacci 举例。

```
struct FiboIterator: IteratorProtocol {
    var state = (0, 1)

    mutating func next() -> Int? {
        let nextValue = state.0
        state = (state.1, state.0 + state.1)

        return nextValue
    }
}

struct Fibonacci: Sequence {

    func makeIterator() -> FiboIterator {
        return FiboIterator()
    }
}
```

由于 FiboIterator 是我们自己实现的，它是一个 struct ，当我们复制多个 Iterator 对象的时候，当然执行的就是值语义，也就是说，复制出来的 Iterator 有其自己的遍历状态。

例如，我们先定义两个 FibIterator：

```
let fibo = Fibonacci()

var fiboIter1 = fibo.makeIterator()
var fiboIter2 = fiboIter1
```

然后，分别多次使用这两个 Iterator：

```
fiboIter1.next() // Optional(0)
fiboIter1.next() // Optional(1)

fiboIter2.next() // Optional(0)
fiboIter2.next() // Optional(1)
fiboIter2.next() // Optional(1)
```

从代码的注释中可以看到， fiboIter1 和 fiboIter2 没有共享它们迭代的状态，而是每个对象各自持有自身的状态。这很简单，但是，Swift里有一个特殊的情况，会让 Iterator 在赋值的时候，执行引用语义，也就是说，多个 Iterator 会共享遍历的状态。

按引用语义拷贝的AnyIterator

有时，我们并不希望对外暴露集合类型使用的 Iterator 的具体类型，它有可能名字很长，有可能未来会被修改，我们只希望对外提供一个 Iterator 的概念，让它对自己的集合类型提供遍历的支持。

这时，我们就可以把 Iterator 通过 AnyIterator 隐藏起来。例如，我们可以把 FiboIterator 改成这样：

```
struct Fibonacci: Sequence {
    typealias Element = Int

    func makeIterator() -> AnyIterator<Element> {
        return AnyIterator(FiboIterator())
    }
}
```

在上面这个实现里，我们使用了 `typealias Element = Int` 明确指定了 `Element` 的类型，这是因为，在 `makeIterator` 的返回值里，我们使用了 `AnyIterator`，它需要我们通过泛型参数，明确指定它返回的元素类型，这个时候，type inference就不好用了。

这样， `Fibonacci.makeIterator` 方法就用 `AnyIterator` 隐藏了它真实使用的 `Iterator` 的类型。而 `AnyIterator` 的用法，和之前是完全一样的。只不过，当我们再查看之前的调用结果时，却会发现，之前复制 `Iterator` 的代码的运行结果，变成了这样：

```
fiboIter1.next() // Optional(0)
fiboIter1.next() // Optional(1)

fiboIter2.next() // Optional(1)
fiboIter2.next() // Optional(2)
fiboIter2.next() // Optional(3)
```

发现差别了么？没错，这次，当我们复制一个 `AnyIterator` 的时候，拷贝行为就从值语义变成了引用语义，它们共享同一个迭代状态。当然，你只要知道这个差异就好了，绝大多数情况下，这种差异不会带来太大的麻烦。毕竟， `Iterator` 的用法并不是在函数之间传来传去。通常，我们获得一个 `Iterator`，用它来遍历，遍历结束之后， `Iterator` 也就作废了。

当然，除了隐藏具体的 `Iterator` 类型之外， `AnyIterator` 还有一个功能，它可以把一个函数直接封装成 `Iterator`，避免了定义 `struct` 的麻烦。

通过函数构造的Iterator

继续我们的 `FiboIterator` 例子，除了给 `AnyIterator` 传递一个遵从 `IteratorProtocol` 的类型之外，它还可以接受一个函数类型的参数，像这样：

```
func fiboIterator() -> AnyIterator<Int> {
    var state = (0, 1)

    return AnyIterator {
        let theNext = state.0
        state = (state.1, state.0 + state.1)

        return theNext
    }
}
```

可以看到，函数版本的实现，和之前的 `struct` 实现逻辑是类似的。只不过这次，我们通过捕获的一个局部变量保存的每次迭代后的状态。然后，我们可以这样来使用 `fiboIterator`：

```
let fiboSequence = AnySequence(fiboIterator)
Array(fiboSequence.prefix(10))
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Swift 3中新增的两个函数

除了使用 `AnySequence` 生成序列之外，Swift 3中还新增了两个函数，方便我们生成各种序列。先来看个简单的：

```
let tenToOne = sequence(first: 10, next: {
    guard $0 != 1 else { return nil }

    return $0 - 1
})

Array(tenToOne)
// [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

`sequence(first:next:)` 方法的第一个参数表示初始值， `next` 根据上一个生成的值，生成序列中的下一个值，直到 `next` 返回 `nil`，生成序列的过程就结束了。于是，上面的代码就生成了10到1的数组。

在上面的例子里， sequence 方法有一个小缺陷，就是用于生成下一个值的状态，只能是 next 的返回值，如果序列中的值的类型和我们保存初始状态的值的类型不同，上面的 sequence 方法就不好用了，为此，Swift 3提供了另外一个版本的 sequence：

```
let fiboSequence = sequence(state: (0, 1), next: {
    (state: inout (Int, Int)) -> Int? in
        let theNext = state.0
        state = (state.1, state.0 + state.1)

        return theNext
    })
```

在这个版本的 sequence 里，每次迭代的状态，和最终生成的序列的值的类型，可以是不同的。并且， next 函数的参数还可以在多次调用之间被修改后，传递给下一次调用。这样，我们就能用和之前同样的逻辑，来生成Fibonacci数列了。

用下面的代码来试一下：

```
Array(fiboSequence.prefix(10))
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

这样，我们就会得到和之前 AnyIterator 一样的结果。了解了 sequence 的概念和用法之后，下次，当你再需要生成一系列有特定属性的值时，就可以试着用它们来替代传统的 for 循环了。

再次理解Sequence抽象的数据类型

在我们上面的两个例子里，无论是元素个数有限的 tenToOne ，还是元素个数无限的 fiboSequence ，它们至少都有一个特征，就是无论我们遍历这两个 Sequence 多少次（当然，要使用不同的 Iterator 对象），结果都是相同的。

但并不是所有的 Sequence 都如此，甚至Swift官方文档上 (https://developer.apple.com/reference/swift/sequence)，对多次遍历 Sequence 这个行为，都有了明确的说明：

> Don’t assume that multiple for-in loops on a sequence will either resume iteration or restart from the beginning.

为什么会这样呢?

因为 Sequence 抽象的行为，仅仅是顺序访问一系列内容。但它并没有约定遍历行为并不会消费掉序列中的元素。来看下面的例子：

```
let stdIn = AnySequence {
    return AnyIterator {
        readLine()
    }
}
```

在上面的代码里， readLine() 执行的是lazy generate。也就是说， AnySequence 是在执行的时候才创建的，而不是在编译时创建的，因此，当我们使用下面的循环，来遍历 stdIn 时：

```
for i in stdIn {
    print(i)
}
```

我们每在控制台输入一行内容， print 就会重复打印一行内容，但从 stdIn 读取出输入的内容之后， stdIn 中就不再包含之前输入的内容了，因此，我们无法多次遍历 stdIn 并得到同样的结果。

所以，当我们遍历 Sequence 类型时，一定要时刻记得自己的遍历行为是否会消费掉序列中的元素。当然，我们也有一个办法来确定是否可以安全的多次遍历序列，就是看这个类型是否遵从 Collection protocol。稍后，我们还会专门提到它。

What's next?

至此，我们应该对 Sequence 和 Iterator 各自承担的角色，以及它们用法，有一定的了解了。但这时，我们再回过头看一下，其实所有的元素最终都是 Iterator 生成的，就算是没有 Sequence ，我们一样可以生成一系列内容出来，并逐个访问它们。所以，现在是时候认真考虑一下这个问题了：到底 Sequence 和 Iterator 是什么关系呢？为什么Swift没有把 IteratorProtocol 中的约束，直接放到 Sequence 中呢？在下一节中，我们就来讨论这个话题。

从抽象顺序访问一系列数据开始

(https://www.boxueio.com/series/advanced-collections/ebook/160)

Sequence和Iterator究竟是什么关系?

(https://www.boxueio.com/series/advanced-collections/ebook/162)

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/)