集合类型背后的“轮子”

两种不同拷贝语义的Iterator
(https://www.boxueio.com/series/advanced-collections/ebook/161)

自定义一个阳春白雪的Collection
(https://www.boxueio.com/series/advanced-collections/ebook/163)

Sequence和Iterator究竟是什么关系?

Back to series (/series/advanced-collections)

当我们理解了 Sequence 和 Iterator 的用法之后，现在，有个问题似乎又让我们搞不清这两个类型的职责边界了。我们完全可以脱离 Sequence ，而单独使用 Iterator 来实现逐个访问序列中每个元素的任务。

例如，对于我们之前定义过的 FiboIterator：

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

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

        return nextValue
    }
}
```

即便我们不把它单独定义在一个遵从 Sequence 的类型里，也完全可以通过下面的代码访问数列中的每个值：

```
var iter = FiboIterator()

for _ in (1...10) {
    print(iter.next()!)
}
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

看到这里，再考虑下我们一开始的问题： iter 表示的究竟是一个用来访问元素的 Iterator ，还是一个表示10个Fibonacci数的 Sequence 呢？是不是它们的界线更模糊了？

Iterator是一个遍历时消费自身的Sequence

实际上，我们的确可以把 Iterator 理解为一个**在遍历时会消费掉自身元素的序列**，我们在上一节中使用的 stdIn 也好，我们刚才使用的 iter 也罢，都属于这类型的序列。它们自身的遍历状态由序列本身维护，元素遍历完就消费掉了，因此，我们的确可以把这种序列的 Iterator 和 Sequence 类型整合起来。

甚至， Sequence 还提供了一个默认的 makeIterator 实现，如果遵从 Sequence 的类型自身就是一个 Iterator ，这个默认的实现就直接返回 Self 作为它的 Iterator 类型。例如，我们可以把 FiboIterator 的声明改成这样：

```
struct FiboIterator: IteratorProtocol, Sequence {
    // The same as before
}
```

然后，我们就直接获得了一个默认的 makeIterator 实现：

```
var iter = FiboIterator().makeIterator()
```

当然，此时 makeIterator 返回的，就是 FiboIterator 创建的对象自身。只不过通过这个例子，我们应该进一步认识到， Iterator 有时就是一个 Sequence 。

但也有一些遍历时不消费自身元素的序列

但情况并不总是如此，还有一类序列遍历它们的时候，是不应该消耗序列元素自身的。例如，对于一个完整的Fibnacci数列来说：

```
let fibs = Fibonacci()
```

遍历不会消耗掉它的成员，无论我们遍历多少次，结果都应该是相同的。因此，我们就不能在这个序列里，维护遍历的状态，它需要一个单独的对象来完成遍历的任务。而这，就是把 Sequence 和 Iterator 类型要分开的原因。

对于 fibs 来说，我们要调用 makeIterator 方法获得一个单独的遍历并保存状态的对象：

```
var fibsIter1 = fibs.makeIterator()
var fibsIter2 = fibs.makeIterator()
```

这样，无论我们遍历多少次，结果都会是一样的：

```
var i = 0
while let value = fibsIter1.next(), i != 10 {
    print("Iter1: \(value)");
    i += 1
}
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

i = 0
while let value = fibsIter2.next(), i != 10 {
    print("Iter2: \(value)")
    i += 1
}
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

了解了 Sequence 和 Iterator 的关系之后，我们来看个和子序列有关的话题。

Sequence和SubSequence

在上一节里，我们还使用了 Sequence 的 prefix(maxLength:) 方法，读取了Fibonacci数列的前10个元素：

```
let fibo1 = Fibonacci.prefix(10)
```

但是我们却没有提及 prefix 返回的类型，直觉上， prefix 应该也返回一个 Fibonacci 对象。但实际上并不是。为了理解这个事情，我们先看一些常用的获取子序列的方法：

首先，是 prefix & suffix ，用来访问序列中的开始或结束的n个元素：

```
let fibo1 = Fibonacci().prefix(10)
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

let fibo2 = Fibonacci().prefix(10).suffix(5)
// [5, 8, 13, 21, 34]
```

其次，是 dropFirst & dropLast ，用来访问序列中除去开始或结束的n个元素后，剩余的部分；

```
let fibo3 = Fibonacci().dropFirst(10).prefix(10)
// [5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

let fibo4 = Fibonacci().prefix(10).dropLast(5)
// [0, 1, 1, 2, 3]
```

最后，是 split ，用于把序列按照特定的条件进行分割，并返回一个数组，数组中的每个元素，都是分割后的子序列：

```
let fiboArray = fibo1.split(whereSeparator: { $0 % 2 == 0 })
// 1 1 | 3 5 | 13 21
```

这些方法的用起来都很简单，在这里，我们要着重讨论的，是它们返回值的类型。除了 split(whereSeparator:) 返回一个 Array 之外， prefix / suffix / dropFirst / dropLast 方法返回的，都不是 Fibonacci 。

在 Sequence 里，有另外一个 associatedtype ，叫 SubSequence ，这些方法返回的类型都是 SubSequence 。

在自定义的序列里，如果我们不明确指定 SubSequence ，编译器会把它设置为 AnySequence<Iterator.Element> 。因此，对于 fibo1/2/3/4 来说，它的类型就是 AnySequence<Int> 。

当然，在实际使用的时候，其实我们很难感受到这些差别，用在 Sequence 的方法，在 SubSequence 也适用。例如，在前面的例子里：

```
let fibo1 = Fibonacci().prefix(10)
```

我们可以通过 `prefix` 得到一个 SubSequence ，然后，可以继续获取这个 SubSequence 的 SubSequence：

```
let fibo2 = Fibonacci().prefix(10).suffix(5)
```

所以，如果不是事先告诉你，你丝毫不会感受到 Sequence 和 SubSequence 在类型上的差别。

既然如此，为什么我们还需要两个独立的类型表示序列呢？答案很简单，当我们需要表达子序列的时候，除了访问序列中的成员，我们还需要记录这个子序列的起始和结束位置。如果我们只是根据截取的区间从原序列生成一个新的序列，子序列表示的区间就不可变了。

当然，由于 SubSequence 默认的类型是 `AnySequence<T>` ，它的 `Iterator` 类型，自然也就是 `AnyIterator<T>` 。当复制这个 `Iterator` 的时候，会执行引用语义，这是我们要注意的地方。

如果你不希望这个行为，就得自己定义一个 SubSequence 类型，然后实现我们上面提到的5个方法。并在序列类型的定义里，自定义 SubSequence 就好了。

What's next?

通过这几节的内容我们可以发现，其实 Sequence 是个非常单调的类型抽象，它只约束了一个我们可以逐个访问的数据序列。而事实上，我们对一个序列的访问需求要比顺序遍历复杂的多，例如：

- 通过下标随机访问；
- 确定序列中元素的个数；
- 确定序列中元素的遍历顺序；
- 对同一个序列多次执行遍历；

等等。而完成这些的方法，都不是遵从 Sequence 的类型可以保障的，我们需要基于 Sequence 开发更复杂的约束。在Swift里，这个约束叫做： `Collection` 。

⏮ 两种不同拷贝语义的Iterator
(https://www.boxueio.com/series/advanced-collections/ebook/161)

自定义一个阳春白雪的Collection ⏭
(https://www.boxueio.com/series/advanced-collections/ebook/163)