Коллекции. Sequence

Протокол Sequence является основой для всех коллекций. Несмотря на большой список методов Sequence предъявляет всего одно требование - метод makeIterator(). Для остальных методов уже написана дефолтная реализация.
Поэтому мы начнем со знакомства с итераторами.

// --- КОД --- //

```
public protocol IteratorProtocol {
    associatedtype Element
    public mutating func next() -> Self.Element?
}
```

В протоколе IteratorProtocol объявлен всего один метод next(). Он возвращает следующий элемент или nil если его нет. Обратите внимание, что тип элементов последовательности объявлен в итераторе. Поэтому в самой Sequence элементы объявляются через него.

// --- КОД --- //

```
public func drop(while predicate: (Self.Iterator.Element) throws -> Bool) rethrows ->
Self.SubSequence
```

Итераторы возвращают только следующий элемент и никогда не сбрасываются в начало. Они могут предоставлять элементы бесконечно или возвращать всегда nil.

Рассмотрим пример простейшего итератора. Он не связан ни с какой последовательностью. Он генерирует числа кратные заданному значению. При этом nil он никогда не вернет. Если вызывать next() в цикле while let element = iterator.next(), то произойдет переполнение переменной base и приложение упадет.

// --- Коллекции. Sequence CODE SNIPPET #1 --- //

```
struct MultiplicityIterator: IteratorProtocol {
    var base = 1

    public mutating func next() -> Int? {
        defer { base += base }
        return base
    }
}

var i = MultiplicityIterator()
i.base = 4
```

```
i.next() // Optional(4)
i.next() // Optional(8)
i.next() // Optional(16)
```

Большинство итераторов имеют value semantics, но это не всегда так. AnyIterator - это итератор который оборачивает другой итератор. Его можно использовать для того, чтобы скрыть детали реализации.

```
// --- Коллекции. Sequence CODE SNIPPET #2 --- //

let sequence = stride(from: 0, to: 10, by: 1) // Последовательность чисел от 0 до 10 с шагом 1

var iterator1 = sequence.makeIterator()
var anyIterator1 = AnyIterator(iterator1)

var iterator2 = iterator1
var anyIterator2 = anyIterator1

iterator1.next() // Optional(0)
iterator1.next() // Optional(1)
iterator2.next() // Optional(0)
iterator2.next() // Optional(1)

anyIterator1.next() // Optional(0)
anyIterator2.next() // Optional(1)
anyIterator1.next() // Optional(2)
anyIterator1.next() // Optional(3)
```

Поэтому не следует передавать итераторы из одной части программы в другую. Они должны использоваться локально и сразу отбрасываться после использования.

У AnyIterator есть еще одно применение. У него есть второй конструктор. Он принимает замыкание, возвращающее следующий элемент последовательности. Вместе с AnySequence это позволяет создать новую последовательность без создания нового типа.

```
// --- Коллекции. Sequence CODE SNIPPET #3 --- //

let randomIterator = AnyIterator() { arc4random_uniform(100) }
let randomSequence = AnySequence(randomIterator)

let arrayOfRandomNumbers = Array(randomSequence.prefix(10)) // Массив случайных чисел
```

Вернемся к самим последовательностям. Скорее всего вам не придется взаимодействовать с итераторами самостоятельно. Обычно для прохода по последовательности используется for in цикл который сам создает новый итератор и вызывает у него next() пока не получит nil.

// --- Коллекции. Sequence CODE SNIPPET #4 --- //

```
for randomNumber in randomSequence.prefix(15) {
    print("Another random number: \(randomNumber)")
}
```

Рассмотрим Sequence аналогичный, показанному ранее MultiplicityIterator. В отличие от итератора он будет генерировать числа в виде последовательности которую мы можем, например, преобразовать в массив чисел. Каждый раз вызывая prefix() мы получаем ту же самую последовательность чисел т.к. создается новый итератор.

// --- Коллекции. Sequence CODE SNIPPET #5 --- //

```
struct MultiplicitySequence: Sequence {
    private let base: Int

    init(base: Int) {
        self.base = base
    }

    public func makeIterator() -> MultiplicityIterator {
        var iterator = MultiplicityIterator()
        iterator.base = base
        return iterator
    }
}

let multiplicitySequence = MultiplicitySequence(base: 4)
Array(multiplicitySequence.prefix(4)) // [4, 8, 16, 32]
Array(multiplicitySequence.prefix(4)) // [4, 8, 16, 32]
```

Важно понимать, что последовательность не всегда можно представить как массив с данными. Например, у нас есть поток данных из интернета. Его можно обернуть в Sequence. Но получив однажды какой-то элемент мы не можем заставить последовательность выдать нам его снова. Даже если создадим новый экземпляр итератора. В отличии от коллекций последовательности при каждом проходе могут генерировать любые значения. Такие последовательности называются destructively consumed sequence.

Возможно вы задумались зачем вообще в последовательности нужен итератор и можно ли обойтись без него. Действительно, в некоторых случаях это возможно. Swift вам в этом поможет. Все что нужно сделать это определить в Sequence метод next().

// --- Коллекции. Sequence CODE SNIPPET #6 --- //

```
struct Countdown: Sequence, IteratorProtocol {

    var count: Int

    mutating func next() -> Int? {
        if count == 0 {
            return nil
        } else {
            defer { count -= 1 }
            return count
        }
    }
}
```

Однако такой подход можно использовать только с разрушающимися последовательностями. Ведь каждый такой экземпляр Sequence можно будет пройти только один раз - метода создающего новый итератор нет. Последовательность сама себе итератор.

Еще одной важной частью Sequence является Subsequence. Он используется как результат выполнения методов у Sequence, возвращающих часть последовательности, например: prefix и suffix - возвращают первые и последние n элементов; dropFirst и dropLast - возвращают последовательность без первых и последних n элементов; split - разделяет последовательность по разделителю и возвращает массив subsequence.

Если вы не предоставите собственную реализацию, то по умолчанию в качестве Subsequence будет использоваться AnySequence<Iterator.Element>. В некоторых случаях будет логично в качестве Subsequence использовать тот же тип, что и сама коллекция т.е. Subsequence == Self, но в данный момент Swift не поддерживает рекурсивное описание associated type. Эта возможность будет реализована в будущем и код ниже будет компилироваться.

// --- Коллекции. Sequence CODE SNIPPET #7 --- //

```
// Не компилируется в Xcode 9 Beta 5
protocol Sequence {
    associatedtype SubSequence: Sequence
    where Iterator.Element == SubSequence.Iterator.Element, SubSequence.SubSequence
== SubSequence
```

```
    func dropFirst(_ n: Int) -> Self.SubSequence
    // ...
}
```

По понятным причинам получить размер Sequence невозможно. Но в протоколе есть свойство underestimatedCount. Оно позволяет получить примерное количество элементов в последовательности. Оно гарантированно будет не больше реального числа объектов, но может быть меньше. Его безопасно использовать с разрушающимися коллекциями.

Еще одно требование Sequence - это соблюдение алгоритмической сложности. Большинство методов должны иметь сложность O(n). Т.е. время их выполнения прямо пропорционально размеру последовательности. Но некоторые должны отрабатывать за константное время. Например, makeIterator(), dropFirst(), dropLast() и свойство underestimatedCount. Обратитесь к документации для уточнений если будете реализовывать собственные последовательности.