swift4 新功能

内容

开区间

字符串

同文件内的扩展, 私有声明可见

智能Key path

编码和解码

协议相关类型的约束

字典(Dictionary)和集合(Set)的增强

MutableCollection.swapAt 方法

reduce 和 inout

泛型下标

NSNumber 桥接

类和协议的组合

开区间

SE-0172 带来一种新的 RangeExpression 协议和一组前缀/后缀操作符给开区间. 比如现在区间无论是上界还是下界都可以不指定.

无限序列

你可以用开区间来造一个无限序列, 对长期使用 enumerated() 方法的同学来说,这是一个福音,尤

其是当你不想序号从0开始的时候:

let 字母表 = ["a","b","c","d"]

let 加序号的字母表 = zip(1..., 字母表)

Array(加序号的字母表)
集合的下标

在集合的下标中用开区间的话, 集合的 startIndex or endIndex 会"智能填充" 缺失的那一边.

let numbers = [1,2,3,4,5,6,7,8,9,10]
numbers[5...] 取代 numbers[5..<numbers.endIndex]
方式匹配

开区间可用于方式匹配, 比如一个 switch 语句中case 表达式 . 不过, 编译器好像还不能(暂时?) 判定switch 已被穷尽.

```
let value = 5
switch value {
case 1...:
   print("大于0")
case 0:
   print("0")
case ..<0:
   print("小于0")
default:
   fatalError("不可到达")
}
```

字符串

多行字符串字面量

SE-0168 带来一种简洁定义多行字符串的语法,使用 ("""). 在一个多行字符串里并不需要写转义字符, 也就是说大多数文本格式 (如JSON 或 HTML) 就可以直接粘贴而无须任何转义. 结尾三引号的缩进,决定了每一行头部被裁剪的空格多少. Python:致敬我吗 Kotlin:我也早有这功能了

let 多行字符串 = """

```
    这是一个多行字符串.
    不需要在这里转义 "引号".
    结尾三引号的位置,
    控制空格的裁剪数.
    """
print(多行字符串)
```
可以打开控制台 (菜单View > Debug Area > Activate Console) 来看 print 的输出.

字符串"又双"变回一个 Collection了, 没错, 天地暂停,时光倒流

SE-0163 是 Swift 4 字符串模型的第一部分修正. 最大变化 String 再度是一个 Collection (因为在 Swift 1.x中是这样的), 比如 String.CharacterView 已经被并入其父类型中. (其他view, UnicodeScalarView, UTF8View, 和 UTF16View, 依旧存在.)

注意SE-0163还没完全实现并且这条建议中还有很多字符串相关的提议(意思是还有的折腾).

```
let 欢迎语 = "侬好Bobo, !"
 不需要再钻到 .characters 属性里面去了
欢迎语.count
for 字 in 欢迎语 {
    print(字)
}
```
Substring 是字符串切片后的新类型

字符串切片现在是 Substring类型的实例. String 和 Substring 两者都遵从 StringProtocol. 几乎所有字符串API都在 StringProtocol 所以 String 和 Substring 行为很大程度是一样的.

```
let 逗号的位置 = 欢迎语.index(of: ",")!
let substring = 欢迎语[..<逗号的位置]
type(of: substring)
 Substring 可以调用 String 的 API
print(substring.uppercased())
```
Unicode 9

Swift 4 即将支持 Unicode 9, 当前正在修正 一些时髦emoji适当的语义问题. 下面的所有字符计数是 1, 和实际的对比:

```
"".count  人 + 肤色
"".count  有4个成员的家庭
```

"\u{200D}\u{200D}\u{200D}".count 家庭 + 肤色
"".count 人 + 肤色 + 职业
Character.unicodeScalars 属性

现在可以直接访问一个 Character 的unicode编码值,而不用先转成String (SE-0178):

let c: Character = ""
Array(c.unicodeScalars)
同文件内的扩展, 私有声明可见

SE-0169 更改了访问控制规则,比如在同文件内的扩展中,原类型的private声明也是可见的. 这种改进可让同文件内保持使用private分割类型定义成为可能 , 减少不受欢迎的 fileprivate 关键词的使用.

```
struct SortedArray {
    private var storage: [Element] = []
    init(unsorted: [Element]) {
        storage = unsorted.sorted()
    }
}

extension SortedArray {
    mutating func insert(_ element: Element) {
         storage 此处可见
        storage.append(element)
        storage.sort()
    }
}

let array = SortedArray(unsorted: [3,1,2])
 storage 此处不可见 (不像 fileprivate)
array.storage  error: 'storage' is inaccessible due to 'private' protection level
```

智能key path

SE-0161描述的新式key path有可能搞了个Swift 4的大新闻. 不像Cocoa中基于字符串的那样too simple, Swift中的可是强类型的, 你们要认真学习.

```
struct 人 {
    var 名字: String
```

```
}

struct 书 {
    var 标题: String
    var 作者: [人]
    var 第一作者: 人 {
        return 作者.first!
    }
}

let Vergil = 人(名字: "Vergil")
let Xernaga = 人(名字: "Xernaga")
let Kotlin快速入门书 = 书(标题: "Kotlin快速入门", 作者: [Vergil, Xernaga])
```

Key path由一个根类型开始,和其下任意深度的属性链和下标名组成.

你可以写一个key path由一个反斜杠开始: \书.标题. 每个类型自动获取一个 [keyPath: …] 下标可以设置或获取指定key path的值.

```
Kotlin快速入门书[keyPath: \书.标题]
 Key path 可深入并支持计算属性
Kotlin快速入门书[keyPath: \书.第一作者.名字]
```

Key path 是可被存储和操作的对象. 比如, 你可以给一个key path加上额外字段深入到作者.

```
let 作者KeyPath = \书.第一作者
type(of: 作者KeyPath)
let 名字KeyPath = 作者KeyPath.appending(path: \.名字)  可以省略类型名, 如果编译器能推断的话
Kotlin快速入门书[keyPath: 名字KeyPath]
下标Key path
```

Key paths 也支持下标. 如此一来可以非常便捷的深入到数组或字典这些集合类型中. 不过这功能在当前snapshot还未实现.

压缩化 和 序列化

SE-0166: Swift Archival & Serialization 定义了一种为任意Swift类型 (class, struct, 和 enum) 来描述自身如何压缩和序列化的方法. 类型可遵从 Codable 协议让自身可(解)压缩.

大多数情况下添加 Codable 协议就可以让你的自定义类型完美解压缩, 因为编译器可以生成一个

默认的实现,前提是所有成员类型都是Codable的. 当然你可以覆盖默认方法如果需要优化自定义类型的编码. 这个说来话长 – 还请研读SE-0166.

```
 遵从Codable协议, 让一个自定义类型 (和其所有成员) 可压缩
struct 扑克: Codable {
   enum 全部花色: String, Codable {
      case 黑桃, 梅花, 红心, 方片
   }

   enum 全部点数: Int, Codable {
      case 尖 = 1, 二, 三, 四, 五, 六, 七, 八, 九, 十, 金钩, 皮蛋, 老K
   }

   var 花色: 全部花色
   var 点数: 全部点数
}

let 我的牌 = [扑克(花色: .黑桃, 点数: .尖), 扑克(花色: .红心, 点数: .皮蛋)]
编码
```

一旦有一个Codable值, 你要把它传递给一个编码器以便压缩 .

利用Codable协议的基础设施可以写自己的编解码器, 不过Swift同时为JSON提供一个内置的编解码器 (JSONEncoder 和 JSONDecoder) 和属性列表 (PropertyListEncoder 和 PropertyListDecoder). 这些是在 SE-0167 中定义的. NSKeyedArchiver 同样支持所有的 Codable 类型.

```
import Foundation

var encoder = JSONEncoder()

 JSONEncoder提供的可定制化属性
encoder.dataEncodingStrategy
encoder.dateEncodingStrategy
encoder.nonConformingFloatEncodingStrategy
encoder.outputFormatting = .prettyPrinted 格式化的json字符串
encoder.userInfo

let jsonData = try encoder.encode(我的牌)
```

String(data: jsonData, encoding: .utf8)
解码

let decoder = JSONDecoder()
let decoded = try decoder.decode([扑克].self, from: jsonData)
协议相关类型的约束

SE-0142: 协议的相关类型可以用where语句约束. 看似一小步,却是类型系统表达能力的一大步,让标准库可以大幅简化. 喜大普奔的是, Sequence 和 Collection 在Swift 4中用上这个就更直观了.

Sequence.Element

Sequence 现在有了自己的相关类型 Element . 原先Swift 3中到处露脸的 Iterator.Element , 现在瘦身成Element:

```
extension Sequence where Element: Numeric {
   var 求和: Element {
      var 结果: Element = 0
      for 单个元素 in self {
         结果 += 单个元素
      }
      return 结果
   }
}
```

[1,2,3,4].求和
当扩展 Sequence 和 Collection 时所需约束更少

在Swift 3时代, 这种扩展需要很多的约束:

```
extension Collection where Iterator.Element: Equatable,
   SubSequence: Sequence,
   SubSequence.Iterator.Element == Iterator.Element
```

而在Swift 4, 编译器已经提前知道了上述3个约束中的2个, 因为可以用相关类型的where语句来表达它们.

```
extension Collection where Element: Equatable {
   func 头尾镜像(_ n: Int) ➡ Bool {
      let 头 = prefix(n)
      let 尾 = suffix(n).reversed()
```

```
        return 头.elementsEqual(尾)
    }
}
```

[1,2,3,4,2,1].头尾镜像(2)
字典(Dictionary) 和 集合(Set) 的增强

SE-0165 加了一些很奶死的 Dictionary 和 Set增强.

基于序列(Sequence)的构造器

从一个键值对序列构造字典.

```
let 热门编程语言 = ["Swift", "Python", "Kotlin"]
let 热门编程语言排行 = Dictionary(uniqueKeysWithValues: zip(1..., 热门编程语言))
热门编程语言排行[2]
```

合并(merge)构造器 & merge 方法

当从一个序列构造字典,或把一个序列合并到字典中,描述如何处理重复的键.

```
let 热门技术 = [("苹果", "Swift"), ("谷歌", "TensorFlow"), ("苹果", "Swift Playgrouds"),
                ("苹果", "ARKit"), ("谷歌", "TensorFlowLite"),("谷歌", "Kotlin"),("苹果", "Core ML")]
let 厂商 = Dictionary(热门技术, uniquingKeysWith: { (第一, 最后) in 最后 })
厂商
```

合并构造器或 merge 方法遇到一个字典时就没那么舒服了. 因为字典的元素类型是一个 带标签的元组型 (key: Key, value: Value) 但上述2个方法却要求一个 无标签的 元组型 (Key, Value), 不得已要手工转换. 希望这个今后能完善. 见 SR-922 和 SR-4969.

```
let 默认设置 = ["自动登录": false, "已绑定手机": false, "蓝牙开启": false]
var 用户设置 = ["自动登录": true, "已绑定手机": false]

 会产生一个烦人的类型转换警告
let 合并的设置 = 用户设置.merge(默认设置) { (old, _) in old }

 只能使用以下替代:
用户设置.merge(默认设置.map { $0 }) { (old, _) in old }
用户设置
```

下标的默认值

你现在可以给下标中加一个默认值参数, 当key不存在时会返回这个值, 这样便可让返回类型非Optional.

热门编程语言排行[4, default: "(未知)"]

```
/*:
 在你想通过下标更新一个值时,这个功能就非常有用:
 */
import Foundation
var 词组 = """
   天姥连天向天横 势拔五岳掩赤城
   天台四万八千丈 对此欲倒东南倾
   我欲因之梦吴越 一夜飞度镜湖月
   湖月照我影 送我至剡溪
   """
var 出现频率: [Character: Int] = [:]
for 词 in 词组.components(separatedBy: .whitespacesAndNewlines).joined() {
   出现频率[词, default: 0] += 1
}
for (词, 次数) in 出现频率 {
   if 次数 > 1 {
      print(词,次数)
   }
}
```

Dictionary相关的 map 和 filter

filter 返回一个 Dictionary 而非 Array. 相似的, 新方法mapValues转换值的同时保持字典结构.

```
let filtered = 热门编程语言排行.filter {
   $0.key % 2 == 0
}
type(of: filtered)

let mapped = 热门编程语言排行.mapValues { value in
   value.uppercased()
}
mapped
```

Set.filter 现在同样返回一个 Set 而不是 Array.

let set: Set = [1,2,3,4,5]
let filteredSet = set.filter { $0 % 2 == 0 }
type(of: filteredSet)
分组一个序列

把一个序列分成几组, 比如联系人按姓分组.

let 联系人 = ["张三丰", "李思思", "张素芳", "李白", "王飞飞", "张小军"]
let 通讯录 = Dictionary(grouping: 联系人, by: { $0.first! })
通讯录

SE-0173 介绍了一种交换一个集合中两个元素的新方法. 与既有的 swap(::) 方法不同, swapAt(::) 接受一个要交换的元素切片, 而不是整个元素本身 (通过 inout 参数).

加这个的目的是 swap 方法带2个 inout 参数 不再兼容新的独占式内存访问规则,见 SE-0176. 既有的 swap(::) 方法不能再交换同一个集合中的两个元素.

var numbers = [1,2,3,4,5]
numbers.swapAt(0,1)

Swift 4中非法
swap(&numbers[3], &numbers[4])
numbers
reduce 和 inout

SE-0171 新增 reduce 的一个变体, 让部分结果以 inout 传递给组合函数. 如此一来可以通过消除中间结果的副本来递增一个序列,大幅提升reduce算法的性能.

SE-0171 为实现.

尚未实现
```
extension Sequence where Element: Equatable {
    func uniq() → [Element] {
        return reduce(into: []) { (result: inout [Element], element) in
            if result.last != element {
                result.append(element)
            }
```

```
        }
    }
}

[1,1,1,2,3,3,4].uniq()
```

泛型下标

托 SE-0148 的福, 下标现在可以有泛型参数和返回类型.

最权威的例子莫过于表示 JSON 数据: 你可以定义一个泛型下标来保持调用者期望类型的内容.

```
struct JSON {
    fileprivate var storage: [String:Any]

    init(dictionary: [String:Any]) {
        self.storage = dictionary
    }

    subscript(key: String) -> T? {
        return storage[key] as? T
    }
}

let json = JSON(dictionary: [
    "城市名": "北京",
    "国家代码": "cn",
    "人口": 21_710_000
    ])

 没必要用 as? Int
let population: Int? = json["人口"]
```

另一个例子: Collection 的一个下标接受一个泛型索引序列, 并返回一个这些索引所在的数组:

```
extension Collection {
    subscript(indices: Indices) -> [Element] where Indices.Element == Index {
        var result: [Element] = []
        for index in indices {
            result.append(self[index])
        }
```

```
        return result
    }
}

let words = "我 思 故 我 在".split(separator: " ")
words[[1,2]]
```

NSNumber 桥接

SE-0170 修正部分危险行为当桥接Swift原生数字类型和NSNumber的时候.

```
import Foundation

let n = NSNumber(value: UInt32(301))
let v = n as? Int8  nil(Swift 4). Swift 3会是45 (试试看!).
```

类和协议的组合

SE-0156: 你现在能写出OC这段 UIViewController * 在Swift中的等价代码, 比如声明这样一个变量,这个变量拥有实体类型并同时遵守若干协议. 语法 let 变量: 某个类 & 协议1 & 协议2.

```
import Cocoa

protocol HeaderView {}

class ViewController: NSViewController {
    let header: NSView & HeaderView

    init(header: NSView & HeaderView) {
        self.header = header
        super.init(nibName: nil, bundle: nil)!
    }

    required init(coder decoder: NSCoder) {
        fatalError("not implemented")
    }
}
```

不能传一个简单的NSView进去因为不遵守协议

```
ViewController(header: NSView())
```

错误: argument type 'NSView' does not conform to expected type 'NSView & HeaderView'

必须穿一个 NSView (子类) 同时遵守协议
extension NSImageView: HeaderView {}

ViewController(header: NSImageView()) 有用
https://oleb.net/blog/2017/05/whats-new-in-swift-4-playground/

http://www.xiaoboswift.com
原文出處