可选链

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程

可选值的缺点

- 使用可选值有时会让人感到有点笨拙，所有的解包和检查会变得如此繁重，以至于会让你想要丢几个感叹号上去强制解包，好让你能继续工作下去。但是请小心：如果你强制解包一个没有值的可选值，你的代码就崩了。为了解决这个缺点，Swift 引入两个特性，一是合并空值运算符，二是可选链。

可选链

- 可选链是一个调用和查询可选属性、方法和下标的过程，它可能为 nil 。如果可选项包含值，属性、方法或者下标的调用成功；如果可选项是 nil ，属性、方法或者下标的调用会返回 nil 。多个查询可以链接在一起，如果链中任何一个节点是 nil ，那么整个链就会得体地失败。

可选链代替强制展开

- 你可以通过在你希望如果可选项为非 nil 就调用属性、方法或者脚本的可选值后边使用问号（？）来明确可选链。这和在可选值后放叹号（！）来强制展开它的值非常类似。主要的区别在于可选链会在可选项为 nil 时得体地失败，而强制展开则在可选项为 nil 时触发运行时错误。

- 为了显示出可选链可以在 nil 值上调用，可选链调用的结果一定是一个可选值，就算你查询的属性、方法或者下标返回的是非可选值。你可以使用这个可选项返回值来检查可选链调用是成功（返回的可选项包含值），还是由于链中出现了 nil 而导致没有成功（返回的可选值是 nil ）。

- 另外，可选链调用的结果与期望的返回值类型相同，只是包装成了可选项。通常返回 Int 的属性通过可选链后会返回一个 Int? 。

可选链代替强制展开

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

```
let john = Person()

let roomCount = john.residence!.numberOfRooms
```

可选链代替强制展开

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
```

为可选链定义模型类

```
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName != nil {
            return buildingName
        } else if buildingNumber != nil && street != nil {
            return "\(buildingNumber) \(street)"
        } else {
            return nil
        }
    }
}
```

通过可选链访问属性

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
```

```
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

通过可选链访问属性

```
func createAddress() -> Address {
    print("Function was called.")

    let someAddress = Address()
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}
john.residence?.address = createAddress()
```

不会被打印

通过可选链调用方法

- 函数和方法没有返回类型就隐式地指明为 Void 类型。意思是说它们返回一个 () 的值或者是一个空的元组。

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
```

通过可选链调用方法

- 如果你尝试通过可选链来设置属性也是一样的。上边通过可选链访问属性中的例子尝试设置 address 值给 john.residence，就算是 residence 属性是 nil 也行。任何通过可选链设置属性的尝试都会返回一个 Void? 类型值，它允许你与 nil 比较来检查属性是否设置成功：

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
```

通过可选链访问下标

- 通过可选链访问下标你可以使用可选链来给可选项下标取回或设置值，并且检查下标的调用是否成功。

```
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
```

链的多层连接

- 你可以通过连接多个可选链来在模型中深入访问属性、方法以及下标。总之，多层可选链不会给返回的值添加多层的可选性。

也就是说：

- 如果你访问的值不是可选项，它会因为可选链而变成可选项；
- 如果你访问的值已经是可选的，它不会因为可选链而变得更加可选。

因此：

- 如果你尝试通过可选链取回一个 Int 值，就一定会返回 Int? ，不论通过了多少层的可选链；
- 类似地，如果你尝试通过可选链访问 Int? 值， Int? 一定就是返回的类型，无论通过了多少层的可选链。

用可选返回值链接方法

- 可以通过可选链来调用返回可选类型的方法，并且如果需要的话可以继续对方法的返回值进行链接。

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
```

```
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
        if beginsWithThe {
            print("John's building identifier begins with \"The\".")
        } else {
            print("John's building identifier does not begin with \"The\".")
        }
}
```

KVC/KVO

KVC

- 从 Swift4 开始，类和 struct 都支持 KVC。
- 继承自 NSObject 的类，标记为 @objc 的属性可以使用 setValue(_:forKey)
- 非继承自 NSObject 的类和结构体，使用索引+参数值

KVC-索引+参数名

```swift
struct ValueType {
    var name:String
}
```

```swift
var object = ValueType(name: "Objective-C")
let name = \ValueType.name

// set
object[keyPath: name] = "swift"
// get
let valueOfName = object[keyPath:name]
print(valueOfName)
```

KVC-索引+参数名

```swift
class Teacher {
    var age = 0
    var name = ""
}
```

```swift
let teacher = Teacher()
teacher[keyPath: \Teacher.name] = "zhangsan"
let teacherName = teacher[keyPath: \Teacher.name]
print(teacherName)
```

KVO

- 只有 NSObject 才能支持 KVO。
- 要观察的属性必须使用 @objc dynamic 修饰。

KVO

```
class Person: NSObject {
    @objc dynamic var age = 0
}
```

```
@objc dynamic let p = Person()

var ob : NSKeyValueObservation?
```

```
p.addObserver(self, forKeyPath: "age", options: [.new, .old], context: nil)

self.ob = observe(\.p.age, options: [.new,.old]) { (_, change) in
    print(change.newValue as Any)
}
```

```
override func observeValue(forKeyPath keyPath: String?, of object: Any?, change:
    [NSKeyValueChangeKey : Any]?, context: UnsafeMutableRawPointer?) {
    if keyPath == "age" {
        print("age被改变了")
        print(change as Any)
    }
}
```

聊聊面试

面试看什么

- 你现在的能力
- 你将来的潜力

√

- Array & Set
  - 区别
  - 怎么实现一个Array -> 存储 -> 添加删除怎么做 -> 怎么优化 -> Ring buffer
  - 怎么实现一个Set -> Hash存储 -> Hash算法 -> Hash冲突解决
  - Array算法 -> 排序/查找/乱序/LCS/.....
  - Set算法 -> 子集/位运算/OrderedSet

极客时间

扫码试看/订阅

《Swift核心技术与实战》视频课程