Swift 烧脑体操（一） - Optional 的嵌套

2017-11-26 唐巧 唐巧

前言

Swift 其实比 Objective-C 复杂很多，相对于出生于上世纪 80 年代的 Objective-C 来说，Swift 融入了大量新特性。这也使得我们学习掌握这门语言变得相对来说更加困难。不过一切都是值得的，Swift 相比 Objective-C，写出来的程序更安全、更简洁，最终能够提高我们的工作效率和质量。

Swift 相关的学习资料已经很多，我想从另外一个角度来介绍它的一些特性，我把这个角度叫做「烧脑体操」。什么意思呢？就是我们专门挑一些比较费脑子的语言细节来学习。通过「烧脑」地思考，来达到对 Swift 语言的更加深入的理解。

这是本体操的第一节，练习前请做好准备运动，保持头脑清醒。

准备运动：Optional 的介绍

王巍的《Swifter》(http://swifter.tips/buy)一书中，介绍了一个有用的命令：在 LLDB 中输入 `fr v -R foo`，可以查看 `foo` 这个变量的内存构成。我们稍后的分析将用到这个命令。

在 Swift 的世界里，一切皆对象，包括 Int Float 这些基本数据类型，所以我们可以这么写：`print(1.description)`。

而对象一般都是存储在指针中，Swift 也不例外，这就造成了一个问题，指针为空的情况需要处理。在 Objective-C 中，向一个 nil 的对象发消息是默认不产生任何效果的行为，但是在 Swift 中，这种行为被严格地禁止了。

Swift 是一个强类型语言，它希望在编译期做更多的安全检查，所以引入了类型推断。而类型推断上如果要做到足够的安全，避免空指针调用是一个最基本的要求。于是，Optional 这种类型出现了。Optional 在 Swift 语言中其实是一个枚举类型：

```
public enum Optional<Wrapped> : _Reflectable, NilLiteralConvertible {
    case None
    case Some(Wrapped)
}
```

Optional 的嵌套

Optional 类型的变量，在使用时，大多需要用 `if let` 的方式来解包。如果你没有解包而直接使用，编辑器通过类型推断会提示你，所以看起来这套机制工作得很好。但是，如果 Optional 嵌套层次太多，就会造成一些麻烦，下面我们来看一个例子。

```
let a: Int? = 1
let b: Int?? = a
let c: Int??? = b
```

在这个机制中，1 这个 Int 值被层层 Optional 包裹，我们用刚刚提到的 `fr v -R`，可以很好的看出来内部结构。如下图：

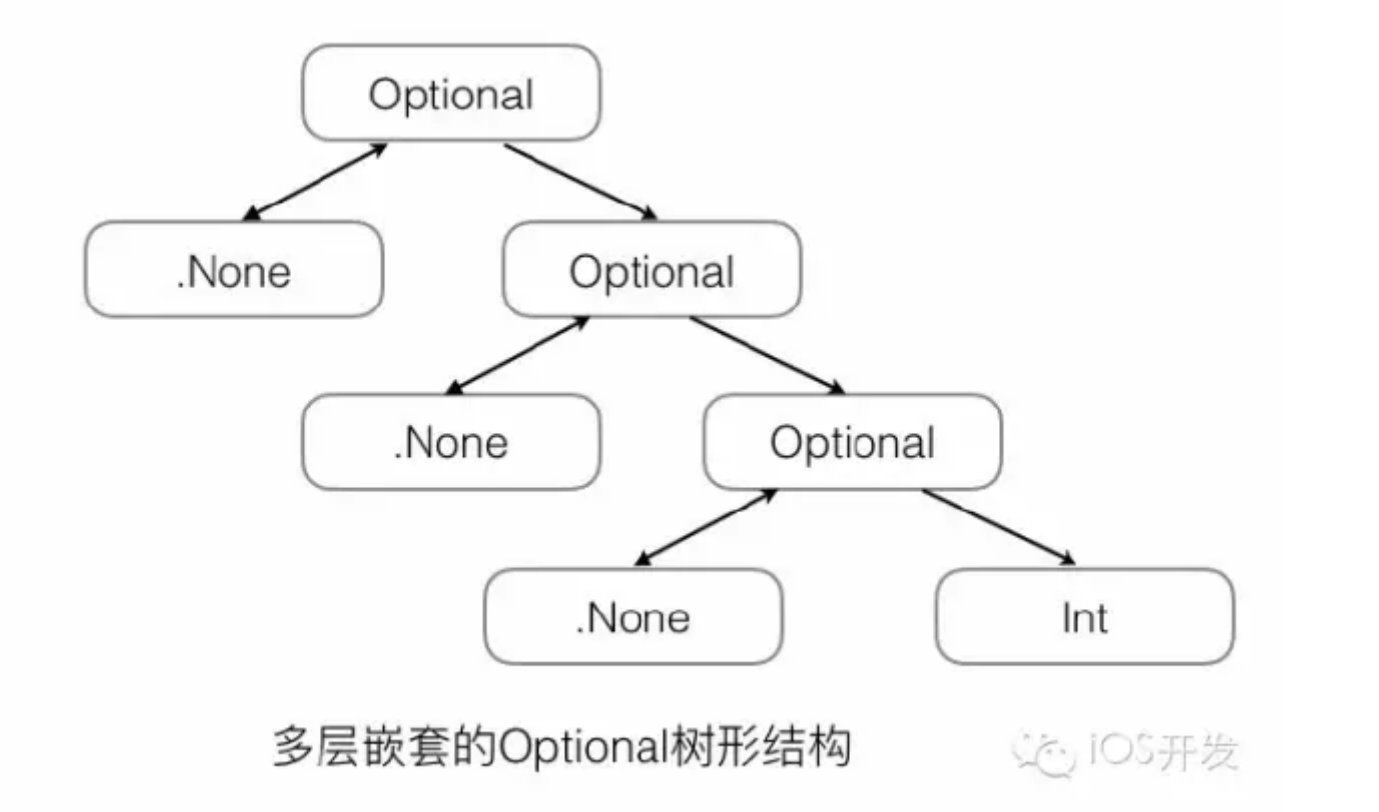
```
(lldb) fr v -R a
(Swift.Optional<Swift.Int>) a = Some {
  Some = {
    value = 1
  }
}

(lldb) fr v -R b
(Swift.Optional<Swift.Optional<Swift.Int>>) b = Some {
  Some = Some {
    Some = {
      value = 1
    }
  }
}

(lldb) fr v -R c
(Swift.Optional<Swift.Optional<Swift.Optional<Swift.Int>>>) c = Some {
```

```
  Some = Some {
    Some = Some {
      Some = {
        value = 1
      }
    }
  }
}
```

从这个示例代码中，我们能看出来多层嵌套的 Optional 的具体内存结构。这个内存结构其实是一个类似二叉树一样的形状，如下图所示：



- 第一层二叉树有两个可选的值，一个值是 .None，另一个值类型是 `Optional<Optional<Int>>` 。
- 第二层二叉树有两个可选的值，一个值是 .None，另一个值类型是 `Optional<Int>` 。
- 第三层二叉树有两个可选的值，一个值是 .None，另一个值类型是 `Int` 。

那么问题来了，看起来这个 Optional.None 可以出现在每一层，那么在每一层的效果一样吗？我做了如下实验：

```
let a: Int? = nil
let b: Int?? = a
let c: Int??? = b
let d: Int??? = nil
```

如果你在 playground 上看，它们的值都是 nil，但是它们的内存布局却不一样，特别是变量 c 和变量 d：

```
(lldb) fr v -R a
(Swift.Optional<Swift.Int>) a = None {
  Some = {
    value = 0
  }
}

(lldb) fr v -R b
(Swift.Optional<Swift.Optional<Swift.Int>>) b = Some {
  Some = None {
    Some = {
      value = 0
    }
  }
}

(lldb) fr v -R c
(Swift.Optional<Swift.Optional<Swift.Optional<Swift.Int>>>) c = Some {
  Some = Some {
    Some = None {
      Some = {
        value = 0
      }
    }
  }
}

(lldb) fr v -R d
(Swift.Optional<Swift.Optional<Swift.Optional<Swift.Int>>>) d = None {
  Some = Some {
    Some = Some {
      Some = {
        value = 0
      }
    }
  }
}
```

- 变量 c 因为是多层嵌套的 nil，所以它在最外层的二叉树上的值，是一个 `Optional<Optional<Int>>` 。
- 变量 d 因为是直接赋值成 nil，所以它在最外层的二叉树上的值，是一个 `Optional.None` 。

麻烦的事情来了，以上原因会造成用 if let 来判断变量 c 是否为 nil 失效了。如下代码最终会输出 `c is not none` 。

```
let a: Int? = nil
let b: Int?? = a
let c: Int??? = b
let d: Int??? = nil

if let _ = c {
```

```
    print("c is not none")
}
```

解释

在我看来，这个问题的根源是：一个 Optional 类型的变量可以接受一个非 Optional 的值。拿上面的代码举例，a 的类型是 Int?，b 的类型是 Int??，但是 a 的值却可以赋值给 b。所以，变量 b（类型为 Int??），它可以接受以下几种类型的赋值：

1. nil 类型
2. Int? 类型
3. Int?? 类型

按理说，Swift 是强类型，等号左右两边的类型不完全一样，为什么能够赋值成功呢？我查了一下 Optional 的源码，原来是对于上面第 1，2 种类型不一样的情况，Optional 定义了构造函数来构造出一个 Int?? 类型的值，这样构造之后，等号左右两边就一样了。源码来自 https://github.com/apple/swift/blob/master/stdlib/public/core/Optional.swift，我摘录如下：

```
public enum Optional<Wrapped> : _Reflectable, NilLiteralConvertible {
  case None
  case Some(Wrapped)

  @available(*, unavailable, renamed="Wrapped")
  public typealias T = Wrapped

  /// Construct a `nil` instance.
  @_transparent
  public init() { self = .None }

  /// Construct a non-`nil` instance that stores `some`.
  @_transparent
  public init(_ some: Wrapped) { self = .Some(some) }

}
```

以上代码中，Optional 提供了两种构造函数，完成了刚刚提到的类型转换工作。

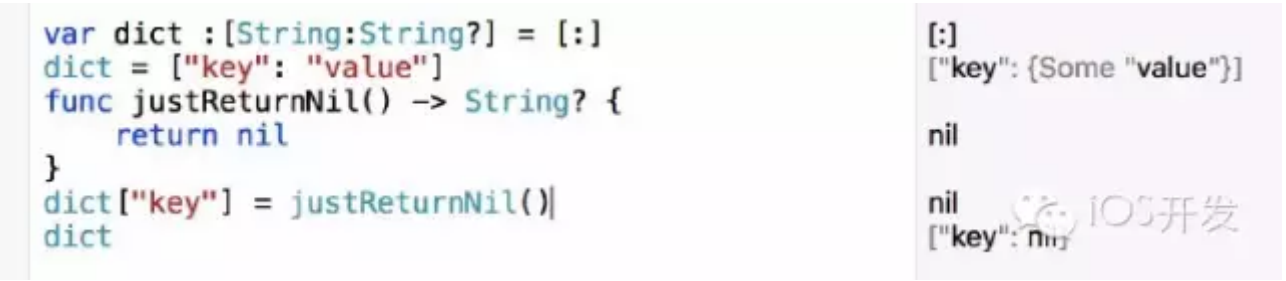
烧脑体操

好了，说了这么多，我们下面开始烧脑了，以下代码来自傅若愚（https://github.com/lingoer）在不久前 Swift 大会（http://atswift.io/#speaker）上的一段分享：

```
var dict :[String:String?] = [:]
dict = ["key": "value"]
func justReturnNil() -> String? {
    return nil
```

```
}
dict["key"] = justReturnNil()
dict
```

以下是代码执行结果：



我们可以看到，我们想通过给这个 Dictionary 设置一个 nil，来删除掉这个 key-value 对。但是从 playground 的执行结果上看，key 并没有被删掉。

为了测试到底设置什么样的值，才能正常地删掉这个 key-value 键值对，我做了如下实验：

```
var dict :[String:String?] = [:]
// first try
dict = ["key": "value"]
dict["key"] = Optional<Optional<String>>.None
dict

// second try
dict = ["key": "value"]
dict["key"] = Optional<String>.None
dict

// third try
dict = ["key": "value"]
dict["key"] = nil
dict

// forth try
dict = ["key": "value"]
let nilValue:String? = nil
dict["key"] = nilValue
dict

// fifth try
dict = ["key": "value"]
let nilValue2:String?? = nil
dict["key"] = nilValue2
dict
```

执行结果如下：

```
var dict :[String:String?] = [:]
// first try
dict = ["key": "value"]
dict["key"] = Optional<Optional<String>>.None
dict
// second try
dict = ["key": "value"]
dict["key"] = Optional<String>.None
dict
// third try
dict = ["key": "value"]
dict["key"] = nil
dict
// forth try
dict = ["key": "value"]
let nilValue:String? = nil;
dict["key"] = nilValue
dict
// fifth try
dict = ["key": "value"]
let nilValue2:String?? = nil;
dict["key"] = nilValue2
dict
```

```
[:]

["key": {Some "value"}]
nil
[:]

["key": {Some "value"}]
nil
["key": nil]

["key": {Some "value"}]
nil
[:]

["key": {Some "value"}]
nil
nil
["key": nil]

["key": {Some "value"}]
nil
nil
[:]
```


我们可以看到，以下三种方式可以成功删除 key-value 键值对：

- `dict["key"] = Optional<Optional<String>>.None`
- `dict["key"] = nil`
- `let nilValue2:String?? = nil;  dict["key"] = nilValue2`

所以，在这个烧脑之旅中，我们发现，一个 [String: String?] 的 Dictionary，可以接受以下类型的赋值：

- nil
- String
- String?
- String??

如果要删除这个 Dictionary 中的元素，必须传入 nil 或 `Optional<Optional<String>>.None`，而如果传入 `Optional<String>.None`，则不能正常删除元素。

好吧，实验出现象了，那这种现象的原因是什么呢?

还好苹果把它的实现开源了，那我们来一起看看吧，源文件来自：https://github.com/apple/swift/blob/master/stdlib/public/core/HashedCollections.swift.gyb，以下是关键代码。

```
public subscript(key: Key) -> Value? {
  get {
    return _variantStorage.maybeGet(key)
  }
  set(newValue) {
    if let x = newValue {
      // FIXME(performance): this loads and discards the old value.
      _variantStorage.updateValue(x, forKey: key)
    }
    else {
      // FIXME(performance): this loads and discards the old value.
      removeValueForKey(key)
    }
  }
}
```

所以，当 Dictionary 的 value 类型为 String 时，如果你要设置它的值，它接受的是一个 String? 类型的参数。而因为我们刚刚例子中的 value 类型为 String?，所以正常情况下它需要的是一个 String?? 类型的参数。在上面的失败的例子中，我们传递的是一个 String? 类型的值，具体值为 `Optional<String>.None`，于是在执行时就会按以下的步骤来进行：

1. 我们传递一个值为 `Optional<String>.None`，类型为 String? 的参数。
2. 因为传的参数类型是 String?，而函数需要的是 String??，所以会执行 Optional 的构造函数，构造一个两层的 Optional。
3. 这个两层 Optional 的值为 `Optional.Some(<Optional<String>.None>)`
4. 进入到 Dictionary 的实现时，会用 if let 进行是否为 nil 的判断，因为两层的 Optional，所以 if let 判断它不是 nil。
5. 所以代码执行到 `_variantStorage.updateValue(x, forKey: key)`，把 Optional .None 当成值，设置给了相应的 key。

如果你没理解，可以再翻翻最初我们对多层嵌套 nil 变量的实验和分析。

我们再看看传递参数是 `Optional<Optional<String>>.None` 的情况，步骤如下：

1. 我们传递一个值为 `Optional<Optional<String>>.None`，类型为 String?? 的参数。
2. 因为参数类型是 String??，函数需要的类型也是 String??，所以参数不经变换，直接进入函数调用中。
3. 这个时候参数的值不变，还是 `Optional<Optional<String>>.None`。
4. 进入到 Dictionary 的实现时，会用 if let 进行是否为 nil 的判断，`Optional<Optional<String>>.None` 用 if let 判断，得到它是 nil。
5. 所以代码执行到 `removeValueForKey(key)`，Dictionary 删除了相应的 key-value 键值对。

总结

好了，「烧脑体操」第一节就做完了，运动一下是不是感觉神清气爽？

总结一下本次烧脑锻炼到的脑细胞：

- Optional 可以多层嵌套。
- 因为 Optional 的构造函数支持，所以可以将一个类型为 T 的值，赋值给一个类型为 T? 的变量。
- 因为 Optional 的构造函数支持，所以可以将 nil 赋值给一个任意嵌套层数的 Optional 变量。
- 将 Optional 嵌套的内容是 nil 时，大家要小心 if let 操作失效问题。
- 多层 Optional 嵌套容易烧脑细胞，尽量避免在工程中使用或触发。
- 遇到问题可以翻翻苹果在 Github 开源的 Swift 源码。

愿大家玩得开心！

阅读原文