Swift 析构过程 Swift 可选链

Swift 自动引用计数（ARC）

Swift 使用自动引用计数（ARC）这一机制来跟踪和管理应用程序的内存

通常情况下我们不需要去手动释放内存，因为 ARC 会在类的实例不再被使用时，自动释放其占用的内存。

但在有些时候我们还是需要在代码中实现内存管理。

ARC 功能

- 当每次使用 init() 方法创建一个类的新的实例的时候，ARC 会分配一大块内存用来储存实例的信息。
- 内存中会包含实例的类型信息，以及这个实例所有相关属性的值。
- 当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。
- 为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。
- 实例赋值给属性、常量或变量，它们都会创建此实例的强引用，只要强引用还在，实例是不允许被销毁的。

ARC 实例

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) 开始初始化")
    }
    deinit {
        print("\(name) 被析构")
    }
}

// 值会被自动初始化为nil，目前还不会引用到Person类的实例
var reference1: Person?
var reference2: Person?
var reference3: Person?

// 创建Person类的新实例
reference1 = Person(name: "Runoob")


//赋值给其他两个变量，该实例又会多出两个强引用
reference2 = reference1
reference3 = reference1

//断开第一个强引用
```

```
reference1 = nil
//断开第二个强引用
reference2 = nil
//断开第三个强引用，并调用析构函数
reference3 = nil
```

以上程序执行输出结果为：

```
Runoob 开始初始化
Runoob 被析构
```

类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 Person 实例的引用数量，并且会在 Person 实例不再被需要时销毁它。

然而，我们可能会写出这样的代码，一个类永远不会有0个强引用。这种情况发生在两个类实例互相保持对方的强引用，并让对方不被销毁。这就是所谓的循环强引用。

实例

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类：Person和Apartment，用来建模公寓和它其中的居民:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) 被析构") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    var tenant: Person?
    deinit { print("Apartment #\(number) 被析构") }
}

// 两个变量都被初始化为nil
var runoob: Person?
var number73: Apartment?

// 赋值
runoob = Person(name: "Runoob")
number73 = Apartment(number: 73)

// 意感叹号是用来展开和访问可选变量 runoob 和 number73 中的实例
// 循环强引用被创建
runoob!.apartment = number73
number73!.tenant = runoob
```

```
// 断开 runoob 和 number73 变量所持有的强引用时，引用计数并不会降为 0，实例也不会被 ARC 销毁
// 注意，当你把这两个变量设为nil时，没有任何一个析构函数被调用。
// 强引用循环阻止了Person和Apartment类实例的销毁，并在你的应用程序中造成了内存泄漏
runoob = nil
number73 = nil
```

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：

- 弱引用
- 无主引用

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为nil的实例使用弱引用。相反的，对于初始化赋值后再也不会被赋值为nil的实例，使用无主引用。

弱引用实例

```
class Module {
    let name: String
    init(name: String) { self.name = name }
    var sub: SubModule?
    deinit { print("\(name) 主模块") }
}

class SubModule {
    let number: Int

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

    weak var topic: Module?

    deinit { print("子模块 topic 数为 \(number)") }
}

var toc: Module?
var list: SubModule?
toc = Module(name: "ARC")
list = SubModule(number: 4)
toc!.sub = list
list!.topic = toc

toc = nil
list = nil
```

以上程序执行输出结果为：

```
ARC 主模块
子模块 topic 数为 4
```

无主引用实例

```
class Student {
    let name: String
    var section: Marks?

    init(name: String) {
        self.name = name
    }

    deinit { print("\(name)") }
}
class Marks {
    let marks: Int
    unowned let stname: Student

    init(marks: Int, stname: Student) {
        self.marks = marks
        self.stname = stname
    }

    deinit { print("学生的分数为 \(marks)") }
}

var module: Student?
module = Student(name: "ARC")
module!.section = Marks(marks: 98, stname: module!)
module = nil
```

以上程序执行输出结果为：

```
ARC
学生的分数为 98
```

闭包引起的循环强引用

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了实例。这个闭包体中可能访问了实例的某个属性，例如self.someProperty，或者闭包中调用了实例的某个方法，例如self.someMethod。这两种情况都导致了闭包 "捕获" self，从而产生了循环强引用。

实例

下面的例子为你展示了当一个闭包引用了self后是如何产生一个循环强引用的。例子中定义了一个叫HTMLElement的类，用一种简单的模型表示 HTML 中的一个单独的元素：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}

// 创建实例并打印信息
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
```

HTMLElement 类产生了类实例和 asHTML 默认值的闭包之间的循环强引用。

实例的 asHTML 属性持有闭包的强引用。但是，闭包在其闭包体内使用了self（引用了self.name和self.text），因此闭包捕获了self，这意味着闭包又反过来持有了HTMLElement实例的强引用。这样两个对象就产生了循环强引用。

解决闭包引起的循环强引用:在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。

弱引用和无主引用

当闭包和捕获的实例总是互相引用时并且总是同时销毁时，将闭包内的捕获定义为无主引用。

相反的，当捕获引用有时可能会是nil时，将闭包内的捕获定义为弱引用。

如果捕获的引用绝对不会置为nil，应该用无主引用，而不是弱引用。

实例

前面的HTMLElement例子中，无主引用是正确的解决循环强引用的方法。这样编写HTMLElement类来避免循环强引用：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) 被析构")
    }

}

//创建并打印HTMLElement实例
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())

// HTMLElement实例将会被销毁，并能看到它的析构函数打印出的消息
paragraph = nil
```

以上程序执行输出结果为：

```
<p>hello, world</p>
p 被析构
```

← Swift 析构过程 Swift 可选链 →