Swift 继承 Swift 析构过程

Swift 构造过程

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

Swift 构造函数使用 init() 方法。

与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器（deinitializer）在类实例释放之前执行清理内存的工作。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。

存储属性在构造器中赋值时，它们的值是被直接设置的，不会触发任何属性观测器。

存储属性在构造器中赋值流程：

- 创建初始值。
- 在属性定义中指定默认属性值。
- 初始化实例，并调用 init() 方法。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字init命名。

语法

```
init()
{
    // 实例化后执行的代码
}
```

实例

以下结构体定义了一个不带参数的构造器 init，并在里面将存储型属性 length 和 breadth 的值初始化为 6 和 12：

```
struct rectangle {
    var length: Double
    var breadth: Double
    init() {
        length = 6
        breadth = 12
    }
```

```
}
var area = rectangle()
print("矩形面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72.0
```

默认属性值

我们可以在构造器中为存储型属性设置初始值；同样，也可以在属性声明时为其设置默认值。

> *使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型。*

以下实例我们在属性声明时为其设置默认值：

```
struct rectangle {
    // 设置默认值
    var length = 6
    var breadth = 12
}
var area = rectangle()
print("矩形的面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72
```

构造参数

你可以在定义构造器 init() 时提供构造参数，如下所示：

```
struct Rectangle {
    var length: Double
    var breadth: Double
    var area: Double

    init(fromLength length: Double, fromBreadth breadth: Double) {
        self.length = length
        self.breadth = breadth
        area = length * breadth
    }

    init(fromLeng leng: Double, fromBread bread: Double) {
```

```
        self.length = leng
        self.breadth = bread
        area = leng * bread
    }
}

let ar = Rectangle(fromLength: 6, fromBreadth: 12)
print("面积为: \(ar.area)")

let are = Rectangle(fromLeng: 36, fromBread: 12)
print("面积为: \(are.area)")
```

以上程序执行输出结果为：

```
面积为: 72.0
面积为: 432.0
```

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器时，主要通过构造器中的参数名和类型来确定需要调用的构造器。

如果你在定义构造器时没有提供参数的外部名字，Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名。

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)

print("red 值为: \(magenta.red)")
print("green 值为: \(magenta.green)")
print("blue 值为: \(magenta.blue)")

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
```

```
let halfGray = Color(white: 0.5)
print("red 值为: \(halfGray.red)")
print("green 值为: \(halfGray.green)")
print("blue 值为: \(halfGray.blue)")
```

以上程序执行输出结果为：

```
red 值为: 1.0
green 值为: 0.0
blue 值为: 1.0
red 值为: 0.5
green 值为: 0.5
blue 值为: 0.5
```

没有外部名称参数

如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线_来显示描述它的外部名。

```
struct Rectangle {
    var length: Double

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }
    //不提供外部名字
    init(_ area: Double) {
        length = area
    }
}

// 调用不提供外部名字
let rectarea = Rectangle(180.0)
print("面积为: \(rectarea.length)")

// 调用不提供外部名字
let rearea = Rectangle(370.0)
print("面积为: \(rearea.length)")

// 调用不提供外部名字
let recarea = Rectangle(110.0)
print("面积为: \(recarea.length)")
```

以上程序执行输出结果为：

```
面积为: 180.0
面积为: 370.0
面积为: 110.0
```

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性，你都需要将它定义为可选类型optional type（可选属性类型）。

当存储属性声明为可选时，将自动初始化为空 nil。

```
struct Rectangle {
    var length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为: \(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为: \(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为: \(recarea.length)")
```

以上程序执行输出结果为：

```
面积为: Optional(180.0)
面积为: Optional(370.0)
面积为: Optional(110.0)
```

构造过程中修改常量属性

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

尽管 length 属性现在是常量，我们仍然可以在其类的构造器中设置它的值：

```
struct Rectangle {
    let length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为: \(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为: \(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为: \(recarea.length)")
```

以上程序执行输出结果为：

```
面积为: Optional(180.0)
面积为: Optional(370.0)
面积为: Optional(110.0)
```

默认构造器

默认构造器将简单的创建一个所有属性值都设置为默认值的实例:

以下实例中，ShoppingListItem类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
```

```
var item = ShoppingListItem()


print("名字为: \(item.name)")
print("数理为: \(item.quantity)")
print("是否付款: \(item.purchased)")
```

以上程序执行输出结果为：

```
名字为: nil
数理为: 1
是否付款: false
```

结构体的逐一成员构造器

如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。

我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

下面例子中定义了一个结构体 Rectangle，它包含两个属性 length 和 breadth。Swift 可以根据这两个属性的初始赋值100.0 、200.0自动推导出它们的类型Double。

```
struct Rectangle {
    var length = 100.0, breadth = 200.0
}
let area = Rectangle(length: 24.0, breadth: 32.0)

print("矩形的面积: \(area.length)")
print("矩形的面积: \(area.breadth)")
```

由于这两个存储型属性都有默认值，结构体 Rectangle 自动获得了一个逐一成员构造器 init(width:height:)。 你可以用它来为 Rectangle 创建新的实例。

以上程序执行输出结果为：

```
名字为: nil
矩形的面积: 24.0
矩形的面积: 32.0
```

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。

以下实例中，Rect 结构体调用了 Size 和 Point 的构造过程：

```
struct Size {
    var width = 0.0, height = 0.0
}
```

```
struct Point {
    var x = 0.0, y = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}


// origin和size属性都使用定义时的默认值Point(x: 0.0, y: 0.0)和Size(width: 0.0, height: 0.0):
let basicRect = Rect()
print("Size 结构体初始值: \(basicRect.size.width, basicRect.size.height) ")
print("Rect 结构体初始值: \(basicRect.origin.x, basicRect.origin.y) ")

// 将origin和size的参数值赋给对应的存储型属性
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))

print("Size 结构体初始值: \(originRect.size.width, originRect.size.height) ")
print("Rect 结构体初始值: \(originRect.origin.x, originRect.origin.y) ")


//先通过center和size的值计算出origin的坐标。
//然后再调用（或代理给）init(origin:size:)构造器来将新的origin和size值赋值到对应的属性中
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))

print("Size 结构体初始值: \(centerRect.size.width, centerRect.size.height) ")
print("Rect 结构体初始值: \(centerRect.origin.x, centerRect.origin.y) ")
```

以上程序执行输出结果为：

```
Size 结构体初始值: (0.0, 0.0)
Rect 结构体初始值: (0.0, 0.0)
Size 结构体初始值: (5.0, 5.0)
Rect 结构体初始值: (2.0, 2.0)
```

```
Size 结构体初始值: (3.0, 3.0)
Rect 结构体初始值: (2.5, 2.5)
```

构造器代理规则

值类型	类类型
不支持继承，所以构造器代理的过程相对简单，因为它们只能代理给本身提供的其它构造器。 你可以使用self.init在自定义的构造器中引用其它的属于相同值类型的构造器。	它可以继承自其它类,这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。

类的继承和构造过程

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

指定构造器	便利构造器
类中最主要的构造器	类中比较次要的、辅助型的构造器
初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。	可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入的实例。
每一个类都必须拥有至少一个指定构造器	只在必要的时候为类提供便利构造器

`Init(parameters) {`
`  statements`
`}`
	`convenience init(parameters) {`
	`  statements`
	`}`

指定构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}
class subClass : mainClass {
    var no2 : Int // 新的子类存储变量
    init(no1 : Int, no2 : Int) {
        self.no2 = no2 // 初始化
        super.init(no1:no1) // 初始化超类
    }
}

let res = mainClass(no1: 10)
```

```
let res2 = subClass(no1: 10, no2: 20)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为: 10
res 为: 10
res 为: 20
```

便利构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass : mainClass {
    var no2 : Int
    init(no1 : Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 便利方法只需要一个参数
    override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let res2 = subClass(no1: 30, no2: 50)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为: 20
res2 为: 30
res2 为: 50
```

构造器的继承和重载

Swift 中的子类不会默认继承父类的构造器。

父类的构造器仅在确定和安全的情况下被继承。

当你重写一个父类指定构造器时，你需要写override修饰符。

```
class SuperClass {
    var corners = 4
    var description: String {
        return "\(corners) 边"
    }
}
let rectangle = SuperClass()
print("矩形: \(rectangle.description)")

class SubClass: SuperClass {
    override init() {  //重载构造器
        super.init()
        corners = 5
    }
}

let subClass = SubClass()
print("五角型: \(subClass.description)")
```

以上程序执行输出结果为：

```
矩形: 4 边
五角型: 5 边
```

指定构造器和便利构造器实例

接下来的例子将在操作中展示指定构造器、便利构造器和自动构造器的继承。

它定义了包含两个个类MainClass、SubClass的类层次结构，并将演示它们的构造器是如何相互作用的。

```
class MainClass {
    var name: String

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

    convenience init() {
        self.init(name: "[匿名]")
    }
}
```

```
let main = MainClass(name: "Runoob")
print("MainClass 名字为: \(main.name)")

let main2 = MainClass()
print("没有对应名字: \(main2.name)")

class SubClass: MainClass {
    var count: Int
    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}

let sub = SubClass(name: "Runoob")
print("MainClass 名字为: \(sub.name)")

let sub2 = SubClass(name: "Runoob", count: 3)
print("count 变量: \(sub2.count)")
```

以上程序执行输出结果为：

```
MainClass 名字为: Runoob
没有对应名字: [匿名]
MainClass 名字为: Runoob
count 变量: 3
```

类的可失败构造器

如果一个类，结构体或枚举类型的对象，在构造自身的过程中有可能失败，则为其定义一个可失败构造器。

变量初始化失败可能的原因有：

- 传入无效的参数值。
- 缺少某种所需的外部资源。
- 没有满足特定条件。

为了妥善处理这种构造过程中可能会失败的情况。

你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在init关键字后面加添问号(init?)。

实例

下例中，定义了一个名为Animal的结构体，其中有一个名为species的，String类型的常量属性。

同时该结构体还定义了一个，带一个String类型参数species的,可失败构造器。这个可失败构造器，被用来检查传入的参数是否为一个空字符串，如果为空字符串，则该可失败构造器，构建对象失败，否则成功。

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

//通过该可失败构造器来构建一个Animal的对象，并检查其构建过程是否成功
// someCreature 的类型是 Animal? 而不是 Animal
let someCreature = Animal(species: "长颈鹿")

// 打印 "动物初始化为长颈鹿"
if let giraffe = someCreature {
    print("动物初始化为\(giraffe.species)")
}
```

以上程序执行输出结果为：

```
动物初始化为长颈鹿
```

枚举类型的可失败构造器

你可以通过构造一个带一个或多个参数的可失败构造器来获取枚举类型中特定的枚举成员。

实例

下例中，定义了一个名为TemperatureUnit的枚举类型。其中包含了三个可能的枚举成员(Kelvin，Celsius，和 Fahrenheit)和一个被用来找到Character值所对应的枚举成员的可失败构造器：

```
enum TemperatureUnit {
    // 开尔文，摄氏，华氏
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
```

```
            return nil
        }
    }
}


let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("这是一个已定义的温度单位，所以初始化成功。")
}

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("这不是一个已定义的温度单位，所以初始化失败。")
}
```

以上程序执行输出结果为：

```
这是一个已定义的温度单位，所以初始化成功。
这不是一个已定义的温度单位，所以初始化失败。
```

类的可失败构造器

值类型（如结构体或枚举类型）的可失败构造器，对何时何地触发构造失败这个行为没有任何的限制。

但是，类的可失败构造器只能在所有的类属性被初始化后和所有类之间的构造器之间的代理调用发生完后触发失败行为。

实例

下例子中，定义了一个名为 StudRecord 的类，因为 studname 属性是一个常量，所以一旦 StudRecord 类构造成功，studname 属性肯定有一个非nil的值。

```
class StudRecord {
    let studname: String!
    init?(studname: String) {
        self.studname = studname
        if studname.isEmpty { return nil }
    }
}
if let stname = StudRecord(studname: "失败构造器") {
    print("模块为 \(stname.studname)")
}
```

以上程序执行输出结果为：

```
模块为 失败构造器
```

覆盖一个可失败构造器

就如同其它构造器一样，你也可以用子类的可失败构造器覆盖基类的可失败构造器。

者你也可以用子类的非可失败构造器覆盖一个基类的可失败构造器。

你可以用一个非可失败构造器覆盖一个可失败构造器，但反过来却行不通。

一个非可失败的构造器永远也不能代理调用一个可失败构造器。

实例

以下实例描述了可失败与非可失败构造器：

```
class Planet {
    var name: String

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

    convenience init() {
        self.init(name: "[No Planets]")
    }
}
let plName = Planet(name: "Mercury")
print("行星的名字是: \(plName.name)")

let noplName = Planet()
print("没有这个名字的行星: \(noplName.name)")

class planets: Planet {
    var count: Int

    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}
```

以上程序执行输出结果为：

```
行星的名字是: Mercury
没有这个名字的行星: [No Planets]
```

可失败构造器 init!

通常来说我们通过在init关键字后添加问号的方式（init?）来定义一个可失败构造器，但你也可以使用通过在init后面添加惊叹号的方式来定义一个可失败构造器(init!)。实例如下：

```
struct StudRecord {
    let stname: String

    init!(stname: String) {
        if stname.isEmpty {return nil }
        self.stname = stname
    }
}

let stmark = StudRecord(stname: "Runoob")
if let name = stmark {
    print("指定了学生名")
}

let blankname = StudRecord(stname: "")
if blankname == nil {
    print("学生名为空")
}
```

以上程序执行输出结果为：

```
指定了学生名
学生名为空
```

Swift 继承 Swift 析构过程

```
class Tiger: Cat {
    let power: Int
    override init() {
        power = 10
        super.init()
        name = "tiger"
    }
}
```

倔强的小胖墩 1年前 (2018-01-15)