理解引用语义的自定义类型

理解class类型的各种init方法
(https://www.boxueio.com/series/understand-ref-types/ebook/175)

确定继承关系用于模拟"is a"的关系
(https://www.boxueio.com/series/understand-ref-types/ebook/177)

什么是two-phase initialization

Back to series (/series/understand-ref-types)

字号
字号
默认主题
金色主题
暗色主题

了解了 class 基础的初始化方式之后，我们继续讨论在继承关系中，类对象的初始化过程。

从init的继承说起

如果你有过其他语言的编程经验就会知道，派生类会自动继承其基类的方法，那么 `init` 呢？作为一个特殊的类方法，它的继承规则也有些特别。默认情况下，派生类不从基类继承任何 `init` 方法。基类的 `init` 方法只在有限条件下被派生类自动继承。我们基于上一段视频的 `Point2D` 派生一个 `Point3D`，来了解派生关系中的初始化规则。

```
class Point3D: Point2D {
    var z: Double
}
```

这时 `Point3D` 会从 `Point2D` 获得什么呢？答案有二。

答案一：**如果派生类没有定义任何designated initializer，那么它将自动继承继承所有基类的designated initializer**。

但即便获得了 `Point2D` 的designated initializer， `self.z` 仍旧是无法初始化的。因此， `Point3D` 实际上并不会从 `Point2D` 继承designated initializer，我们会在控制台看到类似这样的错误：*stored property 'z' without initial value prevents synthesized initializers*。

也就是说，由于 `self.z` 无法被初始化，因此， `Point3D` 无法从 `Point2D` 合成自己的 `init` 方法。为了让我们刚才提到的自动继承生效，最简单的办法，就是直接给 `self.z` 提供一个默认值：

```
class Point3D: Point2D {
    var z: Double = 0
}
```

这样，只要 `Point3D` 不定义自己的designated `init`，它就会自动继承 `Point2D` 的 `init` 方法。不信？我们可以用下面这些方式定义 `Point3D` 对象：

```
let origin3D = Point3D() // 0, 0, 0
let point11 = Point3D(x: 1, y: 1) // 1, 1, 0
let point22 = Point3D(xy: 2) // 2, 2, 0
```

除了自动继承designated `init` 之外， `init` 方法的第二条继承规则是：**如果一个派生类定义了所有基类的designated `init`，那么它也将自动继承基类所有的convenience `init`**。

怎么理解"如果一个派生类定义了所有基类的designated `init`"呢？这里的定义，可以是我们自己手动实现的，当然也可以是默认从基类继承而来的。因此，如果一个派生类自动从基类继承了designated `init`，它必然自动继承基类所有的convenience `init`。不信？我们还可以用下面这些方式来定义 `Point3D`：

```
let point33 = Point3D(at: (3, 3))
let point44 = Point3D(at: ("4", "4"))
```

看到了吧，这些都是我们在 `Point2D` 里定义的convenience `init`。从 `Point2D` 可以自动继承各种 `init` 固然方便，但是我们始终都还是在定义平面上的点，如果我们想自定义一个三维空间中的点呢？这时，故事就不同了。

如何重载init方法?

首先，为了可以在初始化时自定义 `self.z`，我们必须自己编写一个designated `init`，这样，我们也就不用在定义的时候给它设置初始值了：

```
class Point3D: Point2D {
    var z: Double

    init(x: Double = 0, y: Double = 0, z: Double = 0) {
        self.z = z
        super.init(x: x, y: y)
    }
}
```

然后，我们之前所有自动从 Point2D 继承而来的 init 就都不存在了。在派生类中自定义designated init，表示我们要明确控制派生类对象的构建过程。这时，Swift就会自动收手，让你自己来。于是，之前这些创建 Point3D 的代码就会导致编译错误：

```
// Designated init
let point22 = Point3D(xy: 2)
// Convenience init
let point33 = Point3D(at: (3, 3))
let point44 = Point3D(at: ("4", "4"))
```

但是，我们仍有办法让 Point3D 从 Point2D 继承所有的convenience init，只要我们在 Point3D 中手工实现所有 Point2D 的designated init方法就好了：

```
class Point3D: Point2D {
    // ...

    override init(x: Double, y: Double) {
        self.z = 0
        super.init(x: x, y: y)
    }

    override init?(xy: Double) {
        if xy <= 0 { return nil }

        self.z = xy
        super.init(x: xy, y: xy)
    }
}
```

这样，之前会引发编译错误的代码就可以正常工作了。**所以，只要派生类拥有基类所有的designated init方法，他就会自动获得所有基类的 convenience init 方法**。这里，还有一点要特别说明下，在派生类中，重载基类的 convenience init 方法，是不需要 override 关键字的，像这样：

```
class Point3D: Point2D {
    // ...
    convenience init(at: (Double, Double)) {
        self.init(x: at.0, y: at.1, z: 0)
    }
}
```

以上，就是在派生关系中， init 方法的继承以及重载规则，我们用一张图来总结一下：

什么是two-phase initialization

了解了继承体系中 init 方法的关系，我们就可以继续了解two-phase initialization了。这是Swift为了保证在一个继承关系中，派生类和基类的属性都可以正确初始化而约定的初始化机制。简单来说，这个机制把派生类的初始化过程分成了两个阶段。

阶段一：从派生类到基类，自下而上让类的每一个属性都有初始值

因此，如果你回头去看 Point3D 定义的designated init方法就会发现，我们一定是先初始化 self.z ，然后再调用 super.init 初始化基类的成员。如果你把这个顺序倒过来，就会发生编译错误。

阶段二：所有属性都有初始值之后，从基类到派生类，自上而下对类的每个属性进行进一步加工

当 class 中所有属性都有初始值之后，我们才可以调用其它方法或库函数进一步对属性的值进行加工。例如，我们给 Point3D 添加下面的方法：

```
class Point3D: Point2D {
    // ...
    func initXYZ(x: Double, y: Double, z: Double) {
        self.x = round(x)
        self.y = round(y)
        self.z = round(z)
    }
}
```

然后，直接在designated init方法中调用 `initXYZ` 就会导致编译错误：

```
class Point3D: Point2D {
    // ...
    init(x: Double = 0, y: Double = 0, z: Double = 0) {
        // Compile time error
        self.initXYZ(x: x, y: y, z: z)
    }
```

在上面的实现里，尽管通过 `initXYZ` 也可以初始化 `Point3D` 的所有属性，但Swift并不允许你这样，因为无法保证在 `init` 中直接调用其它方法总是可以正确初始化所有的类属性。我们只能按照Swift的套路来，把方法的调用放到“阶段一”后面。

```
class Point3D: Point2D {
    // ...
    init(x: Double = 0, y: Double = 0, z: Double = 0) {
        // Phase 1
        self.z = z
        super.init(x: x, y: y)

        // Phase 2
        self.initXYZ(x: x, y: y, z: z)
    }
}
```

最后，我们用一张图来展示一个 `Point3D` 对象的初始化过程：

What's next?

以上，就是在 `class` 继承关系里，对象的初始化规则。在下一节，我们继续讨论和继承有关的话题，毕竟，这是面向对象编程的核心思想。从一个类派生，当然不仅仅是可以重用基类代码这么简单。为了让你的面向对象代码看上去自然易懂，我们要遵循一个设计准则：确定继承关系用于模拟“is a”的关系。

⏮ 理解class类型的各种init方法
(https://www.boxueio.com/series/understand-ref-types/ebook/175)

确定继承关系用于模拟“is a”的关系 ⏭
(https://www.boxueio.com/series/understand-ref-types/ebook/177)

泊 .I HD

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)

Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)

QQ: 2085489246

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (http://www.miibeian.gov.cn/) 京公网安备 11010802020752号 (http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752)

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)