理解引用语义的自定义类型

差异于语法之外的struct和class
(https://www.boxueio.com/series/understand-ref-types/ebook/174)

什么是two-phase initialization
(https://www.boxueio.com/series/understand-ref-types/ebook/176)

理解class类型的各种init方法

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

由于 class 之间可以存在继承关系，因此它的初始化过程要比 struct 复杂的多，为了保证一个 class 中的所有属性都可以被正确的初始化，Swift引入了一系列特定的规则。在这一节中，我们就来了解 class 中的 init 方法家族。

为了演示各种 init 方法，我们定义了一个 class Point2D，表示平面中一个点的坐标：

```
class Point2D {
    var x: Double
    var y: Double
}
```

当然，这样是不行的。上一节我们已经说过，Swift不会为 class 生成默认的 init 方法，我们必须明确定义 class 类型的对象的创建过程。因此，我们第一个要介绍的，就是 class 的默认 init 方法。

默认init

为了可以像这样构建一个 Point2D 对象：

```
let origin = Point2D()
```

我们得让 Point2D 有一个默认的 init 方法，而我们，可以通过两种方式定义默认 init 。第一种，就是给每一个属性都添加默认值：

```
class Point2D {
    var x: Double = 0
    var y: Double = 0
}
```

这种方式，只适合表意简单并且初始值固定的 class 。因为，此时我们只能创建原点位置的 Point2D 对象，它连一个memberwise init方法都没有。下面的代码会导致编译错误：

```
// Compile time error
let point11 = Point2D(x: 11, y: 11)
```

因此，通常，我们还是至少会为 class 添加一个memberwise init方法。哪怕它就是一个逐个属性赋值的方法：

```
class Point2D {
    var x: Double = 0
    var y: Double = 0

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

但这样，即便 x 和 y 都有了默认值，我们之前定义 orgin 的代码也会导致编译错误。编译器认为我们接手了 init 方法的定义之后，就不会再插手 init 的工作。因此，如果我们定义了memberwise init，那么最好还是把属性的默认值写成memberwise init 方法的默认参数：

```
class Point2D {
    init(x: Double = 0, y: Double = 0) {
        self.x = x
        self.y = y
    }
}
```

这样一来，之前的 origin 和 point11 就可以顺利通过编译了。总之一句话，为了让一个对象可以默认构造， class 必须提供一个不需要参数的 init 方法，并且，这个方法必须初始化 class 的每一个属性。

在Swift里，这种真正初始化 class 属性的 init 方法，叫designated init，它们必须定义在 class 内部，而不能定义在 extension 里，否则会导致编译错误。

另外，除了designated init方法之外，还有一类不真正初始化 class 属性的方法，那它们是做什么的呢？

Convenience init

在实际编程中，除了使用memberwise的方式创建 Point2D ，我们还可能使用一些语义上更好的方式。例如：

```
let point22 = Point2D(at: (2.0, 2.0))
```

这时，我们就需要把作为参数的 (2.0, 2.0) 拆开成 Point2D 的每一个属性，然后调用designated init。对于完成这类任务的 init 方法，就叫做convenience init。

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

可以看到，对于convenience init来说，它有两个要素：

- 使用 convienience 关键字修饰；
- 必须最终调用designated init完成对象的初始化；如果我们直接在convenience init中设置 self.x 或 self.y ，会导致编译错误；

以上我们讨论的 init 方法有一个共性，就是它们的参数一定可以用来初始化 class 的属性，但事实并不总是如此。例如，对于convenience init来说，如果参数拆分后无法传递给designated init方法，这个 init 方法就会执行失败。为了处理这样的情况，Swift中还有一类 init 方法，叫做failable initiazlizer。

Failable init

例如，我们希望用一个 String tuple初始化 Point2D：

```
let point44 = Point2D(at: ("4.0", "4.0"))
```

参考之前的convenience init，我们可以如法炮制一个：

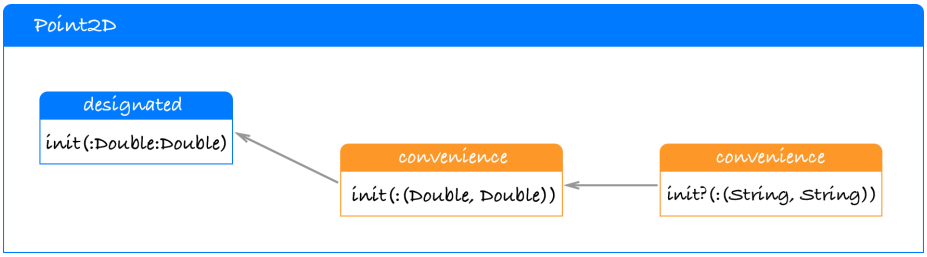
```
class Point2D {
    // ...

    convenience init?(at: (String, String)) {
        guard let x = Double(at.0),
            let y = Double(at.1) else {
                return nil
            }

        self.init(at: (x, y))
    }
}
```

这次，由于 String tuple版本的 init 有可能失败，我们需要用 init? 的形式来定义它。在它的实现里，如果参数中的 String 无法转换成 Double ，我们就返回 nil ，表示构建失败。否则，就调用 Double tuple版本的convenience init最终完成对象的创建。

> 这里，我们只要保证最终可以调用到designated init方法就好了，而不一定要在convenience init方法中，直接调用deignated init方法。

另外要说明的一点是，一个failable designated init方法不能被non failable convenience init调用。但是，一个普通的designated init方法，却可以被failable convenience init调用。

最后，我们用一张图，来表示 class init 方法家族之间的关系：



What's next?

以上，就是单个 `class` 的各种 `init` 的用法和关系。但事情还远未结束，在下一节里，我们将会看到，当类之间存在继承关系的时候，为了保证派生类和基类的属性都可以被正确初始化，Swift定义了一套严格的two phase initialization机制。

◀ 差异于语法之外的struct和class

(https://www.boxueio.com/series/understand-ref-types/ebook/174)

什么是two-phase initialization ▶

(https://www.boxueio.com/series/understand-ref-types/ebook/176)