初始化

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

码拉松

初始化器

- 类、结构体、枚举都可以定义初始化器
- 类有2种初始化器：指定初始化器（designated initializer）、便捷初始化器（convenience initializer）

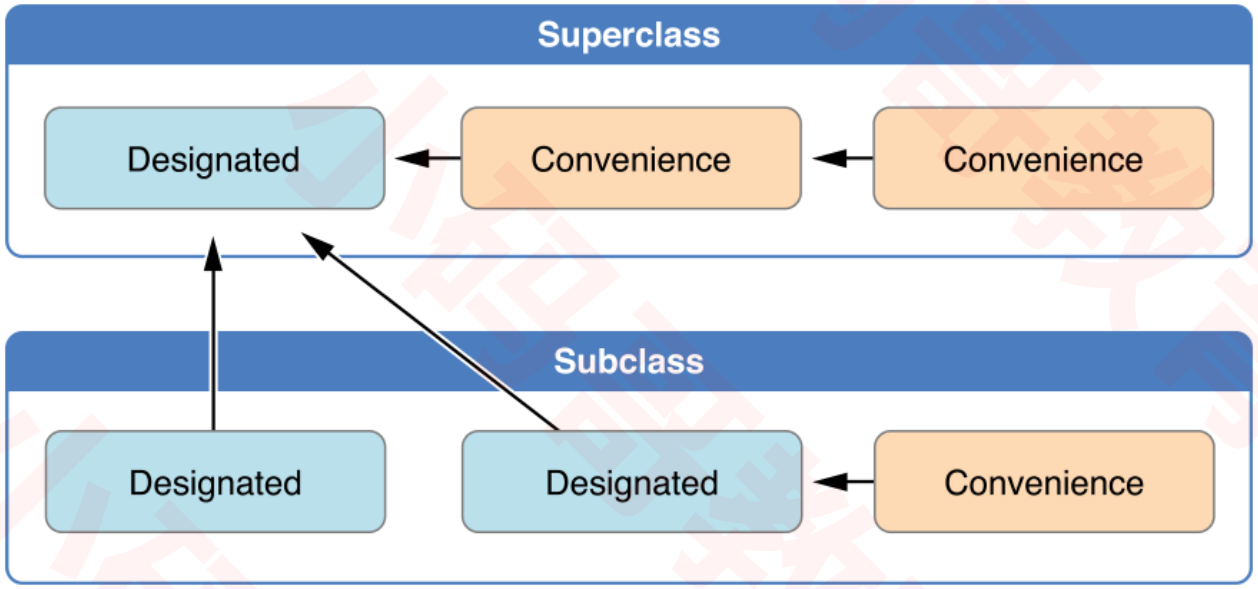
```
// 指定初始化器
init(parameters) {
    statements
}

// 便捷初始化器
convenience init(parameters) {
    statements
}
```

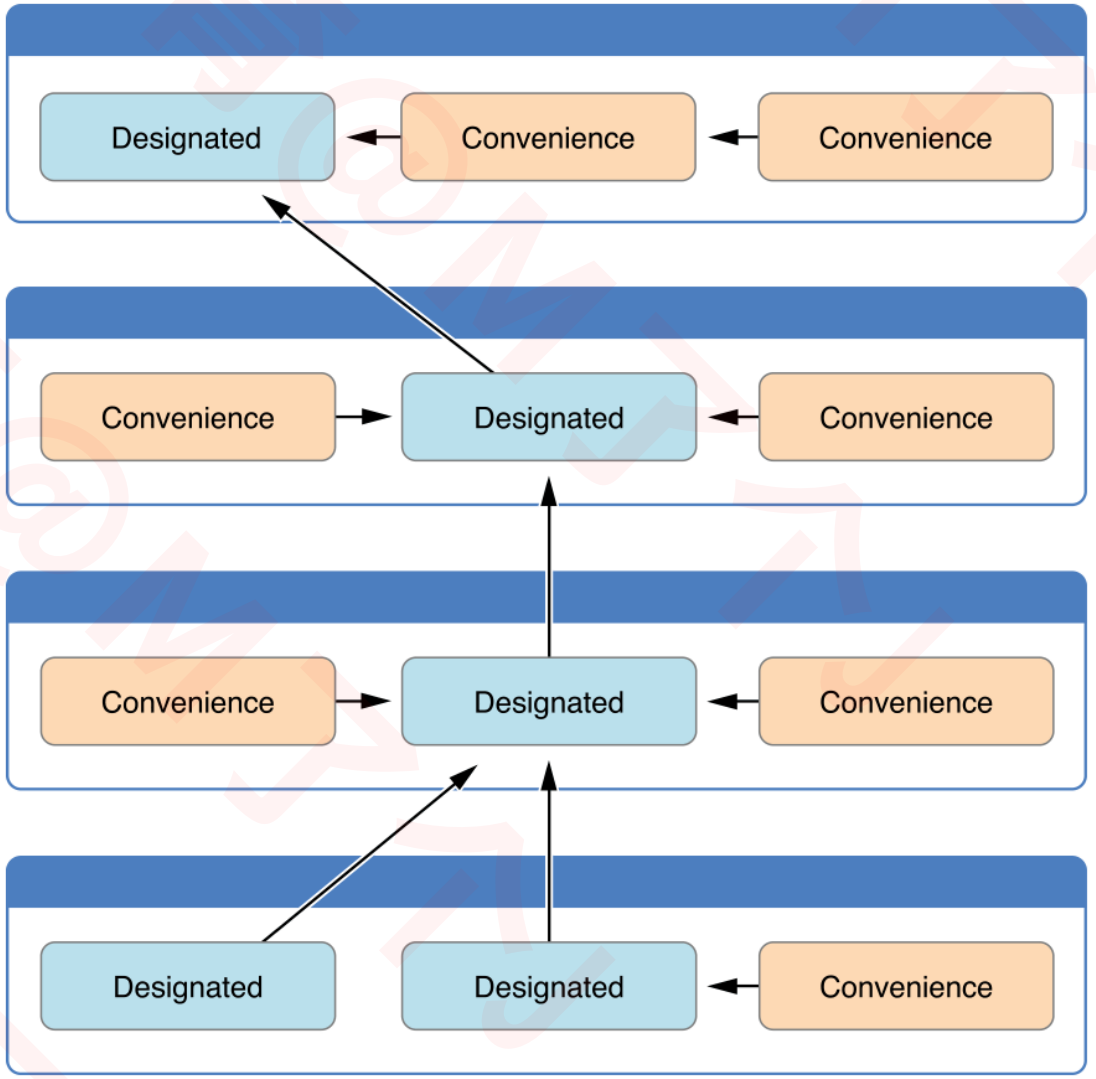
- 每个类至少有一个指定初始化器，指定初始化器是类的主要初始化器
- 默认初始化器总是类的指定初始化器
- 类偏向于少量指定初始化器，一个类通常只有一个指定初始化器

- 初始化器的相互调用规则
  - 指定初始化器必须从它的直系父类调用指定初始化器
  - 便捷初始化器必须从相同的类里调用另一个初始化器
  - 便捷初始化器最终必须调用一个指定初始化器

初始化器的相互调用



- 这一套规则保证了
  - 使用任意初始化器，都可以完整地初始化实例



两段式初始化

- Swift在编码安全方面是煞费苦心，为了保证初始化过程的安全，设定了两段式初始化、 安全检查
- 两段式初始化
  - 第1阶段：初始化所有存储属性
    1. 外层调用指定\便捷初始化器
    2. 分配内存给实例，但未初始化
    3. 指定初始化器确保当前类定义的存储属性都初始化
    4. 指定初始化器调用父类的初始化器，不断向上调用，形成初始化器链
  - 第2阶段：设置新的存储属性值
    1. 从顶部初始化器往下，链中的每一个指定初始化器都有机会进一步定制实例
    2. 初始化器现在能够使用`self`（访问、修改它的属性，调用它的实例方法等等）
    3. 最终，链中任何便捷初始化器都有机会定制实例以及使用`self`

小码哥教育 SEEMYGO

安全检查

- 指定初始化器必须保证在调用父类初始化器之前，其所在类定义的所有存储属性都要初始化完成
- 指定初始化器必须先调用父类初始化器，然后才能为继承的属性设置新值
- 便捷初始化器必须先调用同类中的其它初始化器，然后再为任意属性设置新值
- 初始化器在第1阶段初始化完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用`self`
- 直到第1阶段结束，实例才算完全合法

小码哥教育 SEEMYGO

重写

- 当重写父类的指定初始化器时，必须加上override（即使子类的实现是便捷初始化器）

- 如果子类写了一个匹配父类便捷初始化器的初始化器，不用加上override
  - 因为父类的便捷初始化器永远不会通过子类直接调用，因此，严格来说，子类无法重写父类的便捷初始化器

自动继承

① 如果子类没有自定义任何指定初始化器，它会自动继承父类所有的指定初始化器

② 如果子类提供了父类所有指定初始化器的实现（要么通过方式①继承，要么重写）

- 子类自动继承所有的父类便捷初始化器

③ 就算子类添加了更多的便捷初始化器，这些规则仍然适用

④ 子类以便捷初始化器的形式重写父类的指定初始化器，也可以作为满足规则②的一部分

required

- 用required修饰指定初始化器，表明其所有子类都必须实现该初始化器（通过继承或者重写实现）
- 如果子类重写了required初始化器，也必须加上required，不用加override

```
class Person {
    required init() { }
    init(age: Int) { }
}

class Student : Person {
    required init() {
        super.init()
    }
}
```

小码哥教育 SEEMYGO

属性观察器

- 父类的属性在它自己的初始化器中赋值不会触发属性观察器，但在子类的初始化器中赋值会触发属性观察器

```
class Person {
    var age: Int {
        willSet {
            print("willSet", newValue)
        }
        didSet {
            print("didSet", oldValue, age)
        }
    }
    init() {
        self.age = 0
    }
}
```

```
class Student : Person {
    override init() {
        super.init()
        self.age = 1
    }
}


// willSet 1
// didSet 0 1
var stu = Student()
```

可失败初始化器

- 类、结构体、枚举都可以使用init?定义可失败初始化器

```
class Person {
    var name: String
    init?(name: String) {
        if name.isEmpty  {
            return nil
        }
        self.name = name
    }
}
```

- 之前接触过的可失败初始化器

```
var num = Int("123")
public init?(_ description: String)
```

```
enum Answer : Int {
    case wrong, right
}
var an = Answer(rawValue: 1)
```

- 不允许同时定义参数标签、参数个数、参数类型相同的可失败初始化器和非可失败初始化器
- 可以用init!定义隐式解包的可失败初始化器
- 可失败初始化器可以调用非可失败初始化器，非可失败初始化器调用可失败初始化器需要进行解包
- 如果初始化器调用一个可失败初始化器导致初始化失败，那么整个初始化过程都失败，并且之后的代码都停止执行
- 可以用一个非可失败初始化器重写一个可失败初始化器，但反过来是不行的

反初始化器（deinit）

- deinit叫做反初始化器，类似于C++的析构函数、OC中的dealloc方法
  - 当类的实例对象被释放内存时，就会调用实例对象的deinit方法

```
class Person {
    deinit {
        print("Person对象销毁了")
    }
}
```

- deinit不接受任何参数，不能写小括号，不能自行调用
- 父类的deinit能被子类继承
- 子类的deinit实现执行完毕后会调用父类的deinit