反初始化

cnswift.org/deinitialization

在类实例被释放的时候，*反初始化器*就会立即被调用。你可以是用 deinit 关键字来写反初始化器，就如同写初始化器要用 init 关键字一样。反初始化器只在类类型中有效。

反初始化器原理

当实例不再被需要的时候 Swift会自动将其释放掉，以节省资源。如同自动引用计数中描述的那样，Swift 通过*自动引用计数（ARC）*来处理实例的内存管理。基本上，当你的实例被释放时，你不需要手动清除它们。总之，当你在操作自己的资源时，你可能还是需要在释放实例时执行一些额外的清理工作。比如说，如果你创建了一个自定义类来打开某文件写点数据进去，你就得在实例释放之前关闭这个文件。

每个类当中只能有一个反初始化器。反初始化器不接收任何形式参数，并且不需要写圆括号：

```
1   deinit {
2       // perform the deinitialization
3   }
```

反初始化器会在实例被释放之前自动被调用。你不能自行调用反初始化器。父类的反初始化器可以被子类继承，并且子类的反初始化器实现结束之后父类的反初始化器会被调用。父类的反初始化器总会被调用，就算子类没有反初始化器。

由于实例在反初始化器被调用之前都不会被释放，反初始化器可以访问实例中的所有属性并且可以基于这些属性修改自身行为（比如说查找需要被关闭的那个文件的文件名）。

应用反初始化器

这里有一个应用反初始化器的栗子。这里栗子给一个简单的游戏定义了两个新的类型， Bank 和 Player。 Bank类用来管理虚拟货币，它在流通过程中永远都不能拥有超过10000金币。游戏当中只能有一个 Bank，所以 Bank以具有类型属性和方法的类来实现当前状态的储存和管理：

```
1    class Bank {
2        static var coinsInBank = 10_000
3        static func distribute(coins numberOfCoinsRequested: Int) -> Int {
4            let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
5            coinsInBank -= numberOfCoinsToVend
6            return numberOfCoinsToVend
7        }
8        static func receive(coins: Int) {
9            coinsInBank += coins
10       }
11   }
```

Bank会一直用 CoinsInBank属性来追踪当前金币数量。它同样也提供了两个方法——distribute(coins:)和 receive(coins:)——来处理金币的收集和分发。
distribute(coins:)在分发金币之前检查银行当中是否有足够的金币。如果金币不足， Bank返回一个比需要的数小一些的数值（并且零如果银行里没有金币的话）。 distribute(coins:)声明了一个 numberOfCoinsToVend的变量形式参数，所以数值可以在方法体内修改而不需要再声明一个新的变量。它返回一个整数值来明确提供的金币的实际数量。

receive(coins:)方法只是添加了接受的金币数量到银行的金币储存里去。

Player类描述了游戏中的一个玩家。每个玩家都有确定数量的金币储存在它们的钱包中。这个以玩家的 coinsInPurse属性表示：

```
1    class Player {
2        var coinsInPurse: Int
3        init(coins: Int) {
4            coinsInPurse = Bank.distribute(coins: coins)
5        }
6        func win(coins: Int) {
7            coinsInPurse += Bank.distribute(coins: coins)
8        }
9        deinit {
10           Bank.receive(coins: coinsInPurse)
11       }
12   }
```

每一个 Player实例都会用银行指定的金币数量来作为一开始的限定来初始化，尽管 Player实例可能会在没有足够多金币的时候收到更少的数量。

Player类定义了一个 win(coins:)方法，它从银行取回确定数量的金币并且把它们添加到玩家的钱包当中。 Player类同样实现了一个反初始化器，它会在 Player实例释放之前被调用。这时，反初始化器会把玩家多有的金币返回到银行当中：

```
1   var playerOne: Player? = Player(coins: 100)
2   print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
3   // Prints "A new player has joined the game with 100 coins"
4   print("There are now \(Bank.coinsInBank) coins left in the bank")
5   // Prints "There are now 9900 coins left in the bank"
```

新的 Player实例创建出来了，同时如果可以的话会获取100个金币。这个 Player实例储存了一个可选的 Player变量叫做 playerOne。这里使用了一个可选变量，是因为玩家可以在任何时候离开游戏。可选项允许你追踪当前游戏中是否有玩家。

因为 playerOne是可选项，当它的 coinsInPurse属相被访问来打印默认金币时，必须使用叹号 (!)才能完全符合，并且无论 win(coins:)方法是否被调用：

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// Prints "PlayerOne won 2000 coins & now has 2100 coins"
print("The bank now only has \(Bank.coinsInBank) coins left")
// Prints "The bank now only has 7900 coins left"
```

这时，玩家拥有了2000个金币。玩家的钱包当中保存了2100个金币，并且银行只剩下7900个金币。

```
playerOne = nil
print("PlayerOne has left the game")
// prints "PlayerOne has left the game"

print("The bank now has \(Bank.coinsInBank) coins")
// prints "The bank now has 10000 coins"
```

现在玩家离开了游戏。这通过设置 playerOne变量为 nil来明确，意味着“无 Player实例。”当这个时候， playerOne变量到 Player实例的引用被破坏掉了。没有其他的属性或者变量仍在引用 Player实例，所以它将会被释放掉以节约内存。在释放掉的瞬间，它的反初始化器会自动被调用，然后它的金币被送回给了银行。