**反初始化**

在一个类的实例被释放之前，反初始化函数被立即调用。用关键字deinit来标示反初始化函数，类似于初始化函数用init来标示。反初始化函数只适用于类类型。

**反初始化原理**

Swift会自动释放不再需要的实例以释放资源。如自动引用计数那一章描述，Swift通过自动引用计数（ARC）处理实例的内存管理。通常当你的实例被释放时不需要手动的去清理。但是，当使用自己的资源时，你可能需要进行一些额外的清理。例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，你可能需要在类实例被释放之前关闭该文件。

在类的定义中，每个类最多只能有一个反初始化函数。反初始化函数不带任何参数，在写法上不带括号:

1. deinit {
2. // 执行反初始化
3. }

反初始化函数是在实例释放发生前一步被自动调用。不允许主动调用自己的反初始化函数。子类继承了父类的反初始化函数，并且在子类反初始化函数实现的最后，父类的反初始化函数被自动调用。即使子类没有提供自己的反初始化函数，父类的反初始化函数也总是被调用。

因为直到实例的反初始化函数被调用时，实例才会被释放，所以反初始化函数可以访问所有请求实例的属性，并且根据那些属性可以修改它的行为(比如查找一个需要被关闭的文件的名称)。

反初始化函数操作

这里是一个反初始化函数操作的例子。这个例子是一个简单的游戏，定义了两种新类型，Bank和Player。Bank结构体管理一个虚拟货币的流通，在这个流通中Bank永远不可能拥有超过10,000的硬币。在这个游戏中有且只能有一个Bank存在，因此Bank由带有静态属性和静态方法的结构体实现，从而存储和管理其当前的状态。

1. struct Bank {
2. **static** **var** coinsInBank = 10\_000
3. **static** func vendCoins(**var** numberOfCoinsToVend: Int) -> Int {
4. numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)
5. coinsInBank -= numberOfCoinsToVend
6. **return** numberOfCoinsToVend
7. }
8. **static** func receiveCoins(coins: Int) {
9. coinsInBank += coins
10. }
11. }

Bank根据它的coinsInBank属性来跟踪当前它拥有的硬币数量。银行还提供两个方法—vendCoins和receiveCoins，用来处理硬币的分发和收集。

vendCoins方法在bank分发硬币之前检查是否有足够的硬币。如果没有足够多的硬币，bank返回一个比请求时小的数字(如果没有硬币留在bank中就返回0)。vendCoins方法声明numberOfCoinsToVend为一个变量参数，这样就可以在方法体的内部修改数字，而不需要定义一个新的变量。vendCoins方法返回一个整型值，表明了提供的硬币的实际数目。

receiveCoins方法只是将bank的硬币存储和接收到的硬币数目相加，再保存回bank。

Player类描述了游戏中的一个玩家。每一个player在任何时刻都有一定数量的硬币存储在他们的钱包中。这通过player的coinsInPurse属性来体现:

1. **class** Player {
2. **var** coinsInPurse: Int
3. init(coins: Int) {
4. coinsInPurse = Bank.vendCoins(coins)
5. }
6. func winCoins(coins: Int) {
7. coinsInPurse += Bank.vendCoins(coins)
8. }
9. deinit {
10. Bank.receiveCoins(coinsInPurse)
11. }
12. }

每个Player实例都由一个指定数目硬币组成的启动额度初始化，这些硬币在bank初始化的过程中得到。如果没有足够的硬币可用，Player实例可能收到比指定数目少的硬币。

Player类定义了一个winCoins方法，该方法从bank获取一定数量的硬币，并把它们添加到player的钱包。Player类还实现了一个反初始化函数，这个反初始化函数在Player实例释放前一步被调用。这里反初始化函数只是将player的所有硬币都返回给bank:

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

一个新的Player实例随着一个100个硬币(如果有)的请求而被创建。这个Player实例存储在一个名为playerOne的可选Player变量中。这里使用一个可选变量，是因为players可以随时离开游戏。设置为可选使得你可以跟踪当前是否有player在游戏中。

因为playerOne是可选的，所以由一个感叹号(!)来修饰，每当其winCoins方法被调用时，coinsInPurse属性被访问并打印出它的默认硬币数目。

1. playerOne!.winCoins(2\_000)
2. println("PlayerOne won 2000 coins & now has \    (playerOne!.coinsInPurse) coins")
3. // 输出 "PlayerOne won 2000 coins & now has 2100 coins"
4. println("The bank now only has \(Bank.coinsInBank) coins left")
5. // 输出 "The bank now only has 7900 coins left"

这里，player已经赢得了2,000硬币。player的钱包现在有2,100硬币，bank只剩余7,900硬币。

1. playerOne = nil
2. println("PlayerOne has left the game")
3. // 输出 "PlayerOne has left the game"
4. println("The bank now has \(Bank.coinsInBank) coins")
5. // 输出 "The bank now has 10000 coins"

player现在已经离开了游戏。这表明是要将可选的playerOne变量设置为nil，意思是"没有Player实例"。当这种情况发生的时候，playerOne变量对Player实例的引用被破坏了。没有其它属性或者变量引用Player实例，因此为了清空它占用的内存从而释放它。在这发生前一步，其反初始化函数被自动调用，其硬币被返回到bank。