反初始化

一个*deinitializer*一个类的实例被释放之前立即调用。您可以使用`deinit`关键字编写deinitializers，与使用关键字编写初始值设定项相似`init`。取消初始化器仅适用于类类型。

如何去初始化工作

Swift在不再需要的时候会自动释放你的实例，释放资源。迅速通过处理实例的存储器管理*自动引用计数*（*ARC*），如在自动引用计数。通常，当您的实例被释放时，您不需要执行手动清理。但是，当您使用自己的资源时，可能需要自己执行一些额外的清理。例如，如果您创建自定义类来打开文件并向其中写入一些数据，则可能需要在释放类实例之前关闭该文件。

类定义每个类最多可以有一个deinitializer。取消初始化程序不采用任何参数，并且不带圆括号写入：

```
deinit {
    // perform the deinitialization
}
```

在实例取消分配之前，取消初始化程序会自动调用。你不能自己打电话给一个deinitializer。超类取消初始化程序由它们的子类继承，超类取消初始化程序在子类取消初始化程序实现的末尾自动调用。即使子类不提供它自己的去初始化器，也会始终调用超类去初始化器。

由于直到调用deinitializer之后才会释放实例，deinitializer可以访问所调用实例的所有属性，并可以根据这些属性修改其行为（例如查找需要关闭的文件的名称）。

取消行动者

这是一个实例中的deinitializer。这个例子定义了两种新类型，`Bank`并且`Player`，对于一个简单的游戏。这个`Bank`班级管理着一种制造的货币，这种货币的流通量不能超过10000个。`Bank`在游戏中只能有一个，所以它`Bank`被实现为一个具有类型属性和方法的类来存储和管理其当前状态：

```
class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank`跟踪其持有的`coinsInBank`财产的当前数量。它还提供了两个方法-`distribute(coins:)`和`receive(coins:)`-to处理硬币的分配和收集。

该`distribute(coins:)`方法在分发之前检查银行中是否有足够的硬币。如果没有足够的硬币，`Bank`返回一个比请求的数字更小的数字（如果没有硬币留在银行，则返回零）。它返回一个整数值来表示实际提供的硬币数量。

该`receive(coins:)`方法只是将收到的硬币数量重新加入银行的硬币商店。

该`Player`课程描述了游戏中的玩家。每个玩家随时都有一定数量的硬币储存在他们的钱包中。这由玩家的`coinsInPurse`财产表示：

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
```

```
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    dei
        Bank.receive(coins: coinsInPurse)
    }
}
```

Player在初始化过程中， 每个实例都初始化为具有来自银行的指定数量的硬币的起始许可，但是Player如果没有足够的硬币可用，则实例可能接收到的数量少于该数量。

本Player类定义了一个win(coins:)方法，它获取一定数量从银行硬币，并将它们添加到玩家的钱包。该Player班还实现了deinitializer，这被称为之前Player实例被释放。在这里，取消初始化者只需将所有玩家的硬币退还给银行：

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

Player创建 一个新实例，如果有可用的话，请求100个硬币。这个Player实例存储在一个Player名为的可选变量中playerOne。这里使用了一个可选变量，因为玩家可以随时离开游戏。该选项可让您跟踪游戏当前是否有玩家。

因为playerOne它是可选的，所以!当它的coinsInPurse属性被访问以打印它的默认数量的硬币并且每当它的win(coins:)方法被调用 时，它被限定感叹号（）。

```
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"
```

在本页

这里，玩家赢得了2000个硬币。玩家的钱包现在包含2,100枚硬币，银行只剩下7,900枚硬币。

```
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"
```

玩家现在已经离开了游戏。这通过设置可选playerOne变量来表示nil，意思是“没有Player实例”。在发生这种情况时，playerOne变量对Player实例的引用被打破。没有其他属性或变量仍然指向该Player实例，因此它将被释放以释放其内存。在此之前，它的deinitializer被自动调用，并且它的硬币被返回到银行。