翻译：bruce0505 校对：fd5788

析构过程（Deinitialization）

本页包含内容：

- 析构过程原理
- 析构函数操作

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

析构过程原理

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地去清理。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析构函数。析构函数不带任何参数，在写法上不带括号：

```
deinit {
    // 执行析构过程
}
```

析构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析构函数。子类继承了父类的析构函数，并且在子类析构函数实现的最后，父类的析构函数被自动调用。即使子类没有提供自己的析构函数，父类的析构函数也总是被调用。

因为直到实例的析构函数被调用时，实例才会被释放，所以析构函数可以

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）。

析构函数操作

这里是一个析构函数操作的例子。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。

```
struct Bank {
    static var coinsInBank = 10_000
    static func vendCoins(var numberOfCoinsToVend:
Int) -> Int {
          numberOfCoinsToVend =
min(numberOfCoinsToVend, coinsInBank)
          coinsInBank -= numberOfCoinsToVend
         return numberOfCoinsToVend
    }
    static func receiveCoins(coins: Int) {
    coinsInBank += coins
    }
}
```

`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：

```
    class Player {
      var coinsInPurse: Int
      init(coins: Int) {
          coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
          coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
          Bank.receiveCoins(coinsInPurse)
    }
    }
```

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，该方法从银行获取一定数量的硬币，并把它们添加到玩家的钱包。`Player`类还实现了一个析构函数，这个析构函数在`Player`实例释放前一步被调用。这里析构函数只是将玩家的所有硬币都返回给银行：

```
var playerOne: Player? = Player(coins: 100)
println("A new player has joined the game with \
(playerOne!.coinsInPurse) coins")
// 输出 "A new player has joined the game with 100
coins"
println("There are now \(Bank.coinsInBank) coins left
in the bank")
// 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，是因为玩家可以随时离开游戏。设置为可选使得你可以跟踪当前是否有玩家在游戏中。

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。

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

```
playerOne = nil
println("PlayerOne has left the game")
// 输出 "PlayerOne has left the game"
println("The bank now has \(Bank.coinsInBank) coins")
// 输出 "The bank now has 10000 coins"
```

玩家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，其析构函数被自动调用，其硬币被返回到银行。