# 第七章 类

## 7.1 类的概念

Swift类是构建代码所用的一种通用且灵活的构造体。

在Swift中，用“类”来描述“对象”，所谓的对象是指现实世界中的一切事物，那么类就可以看做是对相似事物的抽象。只说概念可能是很抽象的，举个实际的例子。假如你要买一台电脑，你会考虑哪些因素？按照常见思维，我们会考虑电脑的型号，CPU，显卡，内存，主板，屏幕......。随后你打开电商网站，选了一台这样的电脑，它的型号是ABC，CPU是i7，显卡是 gtx1060，内存是16g等等，2天之后你收到了新的电脑，你很满意。在这次购物过程中，按照面向对象的思维如何理解呢？”类“和”对象“分别是什么？“类”就是你心中的的电脑，你知道一台电脑应该有哪些部件，应该有哪些功能。它是高度抽象的，而你买的型号为ABC的电脑则是一个”对象“，它是按照”电脑“这个抽象概念制造出来的，换句话说对象（型号为ABC的电脑）是类（”电脑“）的一个实例，是具体的。电脑的CPU，显卡等等被称作属性，而电脑能完成的工作称作行为，也就是方法。

### 7.1.1 类和结构体比较

Swift 中类和结构体有很多共同点。共同处在于：

* 定义属性用于存储值
* 定义方法用于提供功能
* 定义下标操作使得可以通过下标语法来访问实例所包含的值
* 定义构造器用于生成初始化值
* 通过扩展以增加默认实现的功能
* 实现协议以提供某种标准功能

当然类有很多结构体所不具备的特性：

* 继承允许一个类继承另一个类的特征
* 类型转换允许在运行时检查和解释一个类实例的类型
* 析构器允许一个类实例释放任何其所被分配的资源
* 引用计数允许对一个类的多次引用

### 7.2.1 类的定义

类和结构体有着类似的定义方式。通过关键字class来定义Swift的类，并在一对大括号中定义它的具体内容，以汽车为例，定义一个Student类。

class Car {

    var brand : String = ""

    var speed : Int = 0

}

在定义类的时候，类名应使用UpperCamelCase（首字母大写的驼峰命名法，如SomeClass）来命名，这是一种规范，Swift中的类都采用这种命名方式。而属性名，方法名应采用lowerCamelCase（首字母小写的驼峰命名法，如someAttribute）来命名，以便和类名更好的区分。

在第一行代码中，使用class关键字定义了名为Car的类，它拥有brand和speed两个属性。Swift要求在定义类的时候，类的属性要进行初始化，所以这里设置brand为“”（空字符串），而speed属性的默认值为0。

也可以为类添加init初始化方法，在init方法中为属性赋初始值。

class Car {

    var brand : String

    var speed : Int

    init() {

        self.brand = ""

        self.speed = 0

    }

}

在原来的代码的基础上又添加了一个init方法，通过该方法对brand和speed属性都进行了初始化操作。

还可以在定义类的时候，为Car添加包含参数的初始化方法。

class Car {

    var brand : String

    var speed : Int

    init() {

        self.brand = ""

        self.speed = 0

    }

    init(brand: String, speed: Int) {

        self.brand = brand

        self.speed = speed

    }

}

有了包含参数的init方法，就可以通过以下方法来实例化Car对象。

let car2 = Car(brand: "Benz",speed: 70)

print(car2.brand)

//打印 Benz

## 7.3 实例方法

实例方法是属于类或结构体的实例的方法。可以通过实例方法设置和修改实例的属性值，或是实现实例的某些功能。实例方法和函数类似，使用func声明，可以定义参数和返回，具体逻辑书写在一堆大括号中。在实例方法中可以调用所属类型的属性和其他实例方法。

要调用实例方法，必须先创建类的实例，然后才能通过类的实例.方法名的方式调用实例方法。

class Car {

    var brand : String

    var speed : Int

    init() {

        self.brand = ""

        self.speed = 0

    }

    func drive(){

        print("Driving \(brand) at \(speed)!")

    }

}

现在Car拥有了名为drive的方法。接着来实例化Car类，以调用drive方法。

var car = Car()

car.brand = "Audi"

car.speed = 80

car.drive()

//打印 Driving Audi at 80!

在第1行，实例化了一个car对象。

接着在第2~3两行，分别设置了car对象的brand和speed属性的值。

然后在第4行的代码中，调用了car对象的drive方法，打印输出。

## 7.4 类属性的set和get方法

上面的例子中，brand和speed是类的存储属性。简单来说，一个存储属性就是存储在特定类或结构体实例里的一个常量或变量。存储属性可以是变量存储属性（用关键字 var 定义），也可以是常量存储属性（用关键字 let 定义）。可以在定义存储属性的时候直接赋值，也可以在构造过程中设置或修改存储属性的值。

除存储属性外，还可以定义计算属性。计算属性不直接存储值，而是提供一个 get方法 和一个可选的 set方法，来间接获取和设置其他属性或变量的值。

class Square {

    var side:Double

    init(side: Double) {

        self.side = side

    }

    var area:Double {

        get{

            return self.side \* self.side

        }

        set(newArea){

            self.side = sqrt(newArea)

        }

    }

}

在这段代码中,定义了一个Square类。它有两个属性，边长和面积。其中，边长（side）是存储属性，而面积（area）是计算属性。在get方法中，将area的值赋值为side \* side，在set方法中，同时设置side的值为sqrt(area)，也就是area开二次方根。

var square = Square(side: 10)

print(square.area)

//打印 100.0

第1行，定义了一个边长为10的正方形，接着打印正方形的面积，自动调用了area的get方法，正确打印出100.0，如果修改一下square的area的值。

square.area = 25

print(square.area)

//打印 25.0

print(square.side)

//打印 5.0

area被设置为25时，会自动调用area的set方法，从而在设置area的同时，把side的值设置为sqrt(25)，为5.0。

## 7.5 属性观察器

通过给属性添加willSet和didSet方法可以给类添加属性观察器。属性观察器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，即使新值和当前值相同的时候也不例外。

见名知意，willSet是在设置属性值之前被调用，而didSet方法则在设置属性值之后被调用。

当实现willSet时，新的属性值作为常量参数被传递。可以为这个参数起一个名，如果不命名，则这个参数就被默认命名为newValue。

class Tower {

    var height : Int = 10 {

        willSet {

            print("willSet：即将设置新值\(newValue)")

        }

        didSet{

            print("didSet：成功将\(oldValue)设置成\(height)")

        }

    }

}

在Tower类中，设置了一个height属性，表示高度，初始化设置成为10。在height的willSet观察器中，打印出height即将被设置的值，而didSet观察器中，我们打印出之前的oldValue和当前的height。编写一个简单的测试代码。

var tower = Tower()

tower.height = 20

测试代码中，首先初始化一个tower，此时height是初始化的10，之后，我们修改height属性为20，观察输出结果：

**willSet：即将设置新值20**

**didSet：成功将10设置成20**

## 7.6 类的静态方法

在移动开发的项目中，在经常会使用到一些工具类定义的大量静态方法。与实例方法调用需要先创建类的实例不同，如果某些方法不依赖具体实例， 那它就是静态的，可以将方法定义为为静态方法。

实例方法为实例所有，而静态方法为类所有，需要通过类来使用。在为结构体和枚举定义静态方法时，可以使用static关键字。在为类定义静态方法时，可以使用class关键字。

class MathTool {

    class func sum(num1: Int, num2: Int) ->Int{

        return num1 + num2

    }

    class func multiply(num1: Int, num2: Int) -> Int{

        return num1 \* num2

    }

}

首先定义了一个名为MathTool的类。

接着在第3~10行的代码中，依次创建了两个静态方法sum和multiply。它们都接受2个整型参数，分别实现了加法和乘法，返回一个整型结果。

使用的时候，使用类名.方法名直接调用。

print(MathTool.sum(num1: 3, num2: 4))

//打印 7

print(MathTool.multiply(num1: 2, num2: 5))

//打印 10

如果一个方法不依赖于实例，建议将它定义为静态方法，可以省去对类进行实例化的步骤。

## 7.7 类的析构

Swift通过自动引用计数（ARC）技术，来实现对实例的内存管理。Swift会自动释放不再需要的实例以释放资源，通常情况下不需要手动清理。但是，当使用自己的资源时，可能需要进行一些额外的清理，就需要用到析构方法（deinit）。例如，如果创建了一个自定义的类来打开一个文件，并写入若干数据，就需要在类的实例被释放前关闭该文件。

class FileManager {

    func loadFile(){

        print("Load file from the resource")

    }

    func writeFile(){

        print("Write the data into the file")

    }

    deinit {

        print("Delete the fileManager")

    }

}

在第一行代码中，定义了一个名为FileManager的类。

接着定义了2个实例方法，分别打印对应的一句话，代表完成了某些操作。

最后实添加了一个deinit析构函数。当FileManager类的实例的引用计数器为0时，系统会自动调用该析构函数。

var fileManager:FileManager? = FileManager()

fileManager!.loadFile()

fileManager!.writeFile()

fileManager = nil

这里，我们创建了一个FileManager类的实例，然后依次调用loadFile和writeFile实例方法，最后将fileManager设置为nil，将该实例的引用计数器设置为0，从而使系统调用析构函数，最终的打印结果为：

Load file from the resource

Write the data into the file

Delete the fileManager

## 7.7 类的继承

一个类可以继承另一个类的方法，属性和其它特性。当一个类继承其它类时，继承类叫子类，被继承类叫超类（或父类）。在 Swift 中，继承是区分”类“与其它类型的一个基本特征。

在 Swift 中，类可以调用和访问父类的方法，属性和下标，并且可以重写这些方法，属性和下标来优化或修改它们的行为。Swift 会检查重写定义在父类中是否有匹配的定义，以此确保重写行为是正确的。

类的继承具有以下优点：

1.继承是在一些通用类的基础上构造，建立和扩充新类的最有效的手段。

2.继承简化了人们对事物的认识和描述，能清晰体现相关类之间的层次结构关系。

3.继承提供软件复用功能。若乙类继承甲类，那么在构建乙类时，只需要再描述与甲类不同的属性和方法即可。这种做法能减小代码的冗余度，增加程序的重用性。

4.继承通过增强一致性，来减少模块间的接口和界面，大大增加了程序的易维护性。

首先，创建一个基类，也就是没有继承其他类的类。

class Animal {

    func say(){

        print("I'm an animal")

    }

    func run() {

        print("I'm running")

    }

}

这里定义了一个名为Animal的基类，它拥有两个方法say和run，接着定义一个继承自Animal的子类。

class Dog :Animal{

    var name: String = "doggy"

    override func say(){

        print("I'm a dog,my name is \(name)")

    }

}

创建了一个名叫Dog的类，用：表示这个类继承自Animal，所以Animal是Dog的父类，此时Dog就获取了Animal的两个方法。

在子类中可以添加父类没有的属性和方法，这里添加了一个名为name的属性，并且初始化为”doggy“。

接着，在子类中重写了父类的say方法。在重写一个父类的方法时，你需要在重写的方法名称前面加上override关键字。如果不使用override关键字的重写，编译时就会被判定为错误。

在定义好子类以后，就可以定义子类的实例了。

var dog = Dog()

dog.run()

//打印 I'm running

dog.say()

//打印 I'm a dog,my name is doggy

虽然在Dog中没有定义run方法，但是依旧可以调用run方法，是因为Dog继承了Animal的run方法。由于对于Animal的say方法进行了重写，所以在Dog中调用say方法是输出的Dog中定义的内容。

如果不希望父类的属性、下标、方法被重写，可以在属性或方法之前，添加final关键字修饰。例如 final var、final func、final class func。当子类试图重写final修饰的方法或属性时，编译时都会报错。

## 7.8 类的引用特征

在Swift环境中，元组、枚举和结构属于值类型，而类属于引用类型。

值类型最基本的特征就是复制在赋值、初始化和传递参数过程中的数据，并为这个数据创建一个新的独立的实例。

引用类型的实例再被赋予到一个变量或者常量，或者作为参数被传递到一个函数时，其操作的不是实例的拷贝，而是已经存在的实例本身，概念类似C++的引用，对引用的任何修改，都会影响内存中的实例对象。使用上一节的Dog类，我们编写一个例子。

let dog1 = Dog()

print("修改前dog1的名字是\(dog1.name)")

//打印 修改前dog1的名字是doggy

let dog2 = dog1

dog2.name = "Swift"

print("修改后dog1的名字\(dog1.name)")

//打印 修改后dog1的名字Swift

print("修改后dog2的名字\(dog2.name)")

//打印 修改后dog2的名字Swift

首先，创建了一个Dog的实例dog1，它的名字使我们初始化设置的”doggy“。

接着，又创建了一个Dog的实例dog2，并把dog1赋值给dog2，随后修改了dog2的名字为”Swift“。随后打印两只狗的名字，输出的都是”Swift“。这充分说明，dog1和dog2实际上指向的是同一个对象。换句话说，它们只是同一个实例的两种叫法。