# 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。可以为类定义属性（常量或变量）和方法。

现实世界中的一切事物都可以被看做对象，而类则是对象的抽象。类是抽象的，对象是具体的。可以把类看做对象的模板，根据类创建对象。

举个例子，如果你要买电脑，如何考虑？你可能会考虑品牌、CPU、显卡、内存、价格等因素，此时你心中的电脑就是类，它是抽象的。随后你买了一台电脑，它的CPU是i7，显卡是GTX1070，内存16G。这里的电脑就是对象，它是具体的，是根据”电脑“这个抽象概念制造出来的。电脑的CPU型号，显卡型号，内存大小被称作属性，属性包括存储属性和计算属性，用电脑浏览网页，看电影，玩游戏被称为动作，也就是方法。方法包括实例方法和类型方法。

### 7.1.1 类的定义

在Swift中，定义类的关键字为class，基本语法：

|  |
| --- |
| class ClassName {  //定义属性和方法  } |

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。

|  |
| --- |
| class Student{      var name:String = "Tom"        func sayHi(){          print("hi! I'm \(name)")      }  } |

本例中，创建了一个Student类，并定义了一个属性和一个方法。属性name类型为String，默认值设置为Tom，swift中的存储属性，必须要进行初始化，这里采用赋初值的方式。随后定义了sayHi方法，打印一句话。

|  |
| --- |
| //通过类创建对象  var stu = Student()  stu.sayHi()  stu.name = "Jack"  stu.sayHi() |

程序输出：

|  |
| --- |
| hello! I'm Tom  hello! I'm Jack |

首先创建了一个对象stu，这个对象被称作Student类的实例，然后调用了sayHi方法，由于Student默认的name是Tom，所以打印出hello! I'm Tom。修改stu的name属性为Jack，再次调用sayHi，打印出hello! I'm Jack。

### 7.1.2 构造过程

类的存储属性必须要进行初始化，除了设置默认值的方式外，还可以采用构造器对其进行初始化。基本语法：

|  |
| --- |
| init(){  //初始化  } |

构造器在创建类的实例的时候被调用，在构造器中，完成对存储属性的初始化。

|  |
| --- |
| class Car{      var brand:String      var speed:Int      init(){          brand = "benz"          speed = 60      }  } |

本例中，为Car类定义了两个存储属性brand和speed，并定义了一个无参的构造器init，在其中对存储属性进行初始化。

当然也可以在定义构造器init()时提供构造参数，在上面的例子中，为Car类继续添加一个包含参数的构造器。

|  |
| --- |
| class Car{      var brand:String      var speed:Int      init(){          brand = "benz"          speed = 60      }        init(brand:String, speed:Int){          self.brand = brand          self.speed = speed      }      func drive() {          print("brand is \(brand), speed is \(speed)")      }  } |

这样，Car类就有了两个构造器，使用类名( )的形式可以创建对象。

|  |
| --- |
| var car = Car()  car.drive()  var car2 = Car(brand: "BMW", speed: 70)  car2.drive() |

程序输出：

|  |
| --- |
| brand is benz, speed is 60  brand is BMW, speed is 70 |

如果存储属性在逻辑上可以为空，就需要将它定义为可选类型optional type，当存储属性被声明为可选类型时，将自动被设置成为空nil。

|  |
| --- |
| class SurveyQuestion {      var text: String      var response: String?      init(text: String) {          self.text = text      }      func ask() {  print(text)  print("response:\(response)")      }  } |

本例中，response被声明为可选类型，并且没有初始化。

|  |
| --- |
| var sq = SurveyQuestion(text: "Are you ok?")  sq.ask() |

程序输出：

|  |
| --- |
| text:Are you ok?  response:nil |

在没有初始化response的情况下，response被默认设置成了nil。

### 7.1.3 类的引用特征

在swift中，元组、枚举和结构体属于值类型，而类属于引用类型。

值类型最显著的特征就是深拷贝(Deep Copy)，也就是复制的时候，传递的是整个对象，每次复制都会在栈内存中开辟一块新的独立的空间，存储拷贝后的副本，但是拷贝的动作不是发生在复制的时候，而是发生在副本被修改的时候，被称作copy-on-write。

与值类型不同的是，引用类型，在复制的时候，传递的是栈内存中的引用，可以将其看做指针，原对象和副本都指向堆内存的同一块区域，如果通过其中之一对堆内存进行修改，两者都会受影响。通过一个例子，展示一下引用类型的特征。定义一个简单的类，并打印。

|  |
| --- |
| class Dog{      var name = "haha"  }  var dog1 = Dog()  print("dog1:\(dog1.name)") |

程序输出：

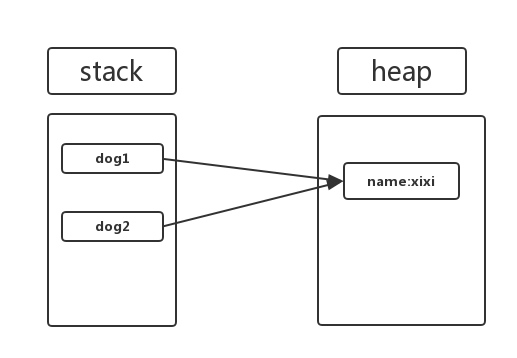
|  |
| --- |
| dog1:haha |

此时，正确答应dog1的名字haha。如果创建dog2，并将dog1的值赋给dog2，然后修改dog2的name为xixi，分别打印dog1和dog2的name。

|  |
| --- |
| var dog2 = dog1  dog2.name = "xixi"  print("dog1:\(dog1.name)")  print("dog2:\(dog2.name)") |

程序输出：

|  |
| --- |
| dog1:xixi  dog2:xixi |

在修改了dog2的name的时候，dog1的name也被修改了。这说明实际上dog1和dog2指向的是同一个对象，换话说，dog2只是dog1的别名。它们的关系可以表示成下图。

### 7.1.4 恒等运算符

上节介绍了类是引用类型，必然出现多个变量或常量引用同一个实例，为了判断两个常量或变量是否引用同一个实例，Swift提供了两个恒等运算符。

|  |  |
| --- | --- |
| 恒等运算符 | 不恒等运算符 |
| ===（三个等号） | ！==（！+两个等号） |
| 如果两个常量或变量引用同一个实例，返回true，否则返回false。 | 如果两个常量或变量不是引用同一个实例，返回true，否则返回false。 |

### 7.1.5 析构过程

Swift通过自动引用计数器(ARC)处理实例的内存管理，Swift会自动释放没有被引用的实例以回收资源，一般情况下不需要手动清理。但是，如果在代码中，引用了外部资源，如读文件，那就需要用到析构方法，在实例被释放之前，做一些额外的工作。

析构方法的基本语法和构造方法类似，但是析构方法不能带任何参数，析构方法由系统调用。

|  |
| --- |
| Deinit {  //执行析构过程  } |

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 |

## 7.2 类的属性

### 7.2.1 存储属性和计算属性

属性可以分为存储属性和计算属性。

存储属性的作用是存储，简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个变量或常量，变量存储属性就用var定义，常量存储属性用let定义。

计算属性计算一个值，计算属性不能直接存储值，而是提供一个getter来取值，一个可选的setter来间接设置其他属性或变量的值。

本章之前的属性都属于存储属性，下面用一个例子介绍计算属性。

|  |
| --- |
| class Square {      var side:Double        var area:Double {          get{              return self.side \* self.side          }          set(newArea){              self.side = sqrt(newArea)          }      }        init(side: Double) {          self.side = side      }  } |

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。

|  |
| --- |
| 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。

### 7.2.2 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，在didSet中原先的值被命名成oldValue。

|  |
| --- |
| 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，首先实例化一个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 |

### 7.2.3 懒加载属性

懒加载属性又称延迟加载属性，指的是当第一次被调用的时候才会计算其初始值的属性。

因为被声明成懒加载的属性，可能在实例构造完成之前无法得到值。而常量属性在构造过程完成之前必须有初始化的值，所以懒加载属性必须声明为变量，并使用lazy关键字标记，以后的开发中，会经常用到懒加载延迟加载的性质。

## 7.3 类的方法

### 7.3.1 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。

下面的例子，定义一个简单的Counter类，实现计数的功能。

|  |
| --- |
| class Counter {      var count = 0      func increment() {          count += 1      }      func increment(by amount: Int) {          count += amount      }      func reset() {          count = 0      }  } |

Counter包含一个存储属性和3个实例方法。count用于计数，无参的increment方法实现计数器加一，而increment（by：Int）方法让计数器一次增加指定数量。reset则可以让计数器清零。

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名.(dot)方法名的方式调用。

|  |
| --- |
| let counter = Counter()  print("count:\(counter.count)")  counter.increment()  print("count:\(counter.count)")  counter.increment(by: 3)  print("count:\(counter.count)")  counter.reset()  print("count:\(counter.count)") |

程序输出：

|  |
| --- |
| count:0  count:1  count:4  count:0 |

实例方法都包含一个隐含的参数self，self代指这个实例本身，多数情况下self是可以省略的，比如本节计数器的例子，实际上increment可以写作：

|  |
| --- |
| func increment() {  self.count += 1  } |

也有某些情况下，self不可以省略，比如第一节包含参数的构造函数的例子：

|  |
| --- |
| init(brand:String, speed:Int){          self.brand = brand          self.speed = speed  } |

这里的方法参数brand和类的属性brand是重名的，所以此时，必须要用self.brand和brand来区分这两者，消除歧义。如果写作brand = brand，Swift会认为brand指的都是函数的参数brand。

### 7.3.2 类型方法

类的另一种方法叫做类型方法或者静态方法，如果某个操作不依赖具体实例，比如求某个数的绝对值，就可以将其定义为类型方法，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。在为类定义类型方法时，可以使用class关键字。

下面定义一个Math类，包含一个求绝对值的方法。

|  |
| --- |
| class Math {      class func abs(\_ num:Int)->Int{          return num >= 0 ? num : -num      }  } |

本例中，利用三元运算符，返回num的绝对值，等价于num大于等于0返回本身，否则返回num的相反数。实现求绝对值这个功能是不依赖于实例的，所以定义成类型方法。事实上，类似Math这种工具类都包含大量类型方法。

通过类名.(dot)方法名来调用类型方法。

|  |
| --- |
| let A = -20  let absA = Math.abs(A)  print("absA:\(absA)") |

程序输出：

|  |
| --- |
| absA:20 |

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。

## 7.4 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 Cat:Animal{      var name: String = "hehe"        override func say(){          print("I'm cat,my name is \(name)")      }  } |

上面的代码中，创建了Cat的类，用：表示这个类继承自Animal，所以Animal是Cat的父类，此时Cat就继承了Animal的say方法和run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”hehe“。

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，需要在重写的方法名称前面加上override关键字，表示这是对父类方法的重写。

在定义好子类以后，就可以定义子类的实例了。

|  |
| --- |
| var cat = Cat()  cat.run()  cat.say() |

程序输出：

|  |
| --- |
| I'm running  I'm cat,my name is hehe |

虽然在Cat中没有定义run方法，但是依旧可以调用run方法，是因为Cat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Cat中调用say方法是输出的Cat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。