Swift 语言快速入门

（内部资料）

大学霸
www.daxueba.net

前　言

Swift 是苹果公司在 2014 年 WWDC 大会上推出的新的编程语言，用于取代原有的苹果官方语言 Objective-C。Swift 主要用于编写 iOS 和 Mac OS 应用程序。在此编程语言推出以后，就有将近 37 万的开发者在苹果官网下载 Swift 手册进行学习。同时，《Swift 中文版》也由国内自发的翻译小组花费 9 天时间完成。

通过《Swift 中文版》，读者可以了解 Swift 的一些语法特性。但由于它只是对语法手册的翻译，存在以下缺陷：

第一，它仅是一个语法词典，只是简单地、逐条地对语法进行解释。类似于学英语，靠一本英语词典学会英语，难度是相当的高。

第二，它所具备的实例有限。地球人都知道，学习编程语言的最佳手段是通过实例代码学习。没有足够的实例，只靠零星的代码片段或语法就能理解 Swift，那是高手所为。

基于以上不可忽略的事实，本书决定着眼于讲解 Swift 语言的入门，将苹果官方提供的 Swift 手册内容重新进行系统的划分，并针对每一个知识点配套相应实例，帮助读者可以快速学习和掌握 Swift 语言。

1.学习所需的系统和软件

- Mac OS 10.9.4 以上的操作系统
- 安装 Xcode 6

2.学习建议

大家学习之前，可以致信到 swift@daxueba.net，获取相关的资料和软件。如果大家在学习过程遇到问题，也可以将问题发送到该邮箱。我们将尽力给大家解决。

目　录

第 8 章　类

虽然函数可以简化代码，但是当一个程序中出现成百上千的函数和变量时，代码还是会显得很混乱。为此，人们又引入了新的类型——类。它是人们构建代码所用的一种通用、灵活的构造方式。本章将主要详细讲解类的使用。

8.1　类与对象

类是一种新的数据类型，类似于生活中犬类、猫类等等。而对象则是将这个抽象的类进行了具体化。例如，在犬类中，有哈士奇，金毛等等，这些就是犬类的具体化，即对象。本节将讲解类的创建以及如何将类进行具体化（即实例化）为对象。

8.1.1　类的组成

在一个类中通常可以包含如图 8.1 所示的内容。

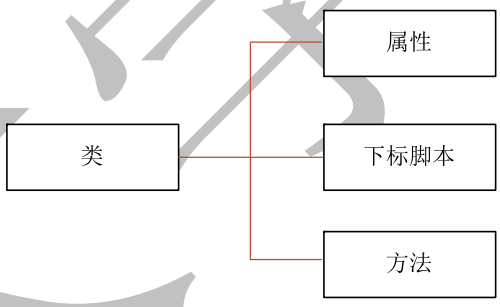

图 8.1　类的构成

其中，这些内容的功能如下：

- 属性：它将值和特定的类关联。
- 下标脚本：访问对象、集合等的快捷方式。
- 方法：实现某一特定的功能，类似于函数。

8.1.2　创建类

在 Swift 中类的创建要比在 Objective-C 中简单的多。在 Objecttive-C 中，需要使用需要@interface 和@end 对类中的内容进行声明，还需要使用@implementation 和@end 对类声明的内容进行实现。在 Xcode 6 之前，它们需要放置在不同的文件中。虽然在 Xcode 6 中，它们可以放置在一个文件中，但是也相当的麻烦。Swift 语言推出了自己创建类的方式，只使用一个 class 关键字，其一般的创建形式如下：

```
class 类名{
  \\具体内容
}
```

注意：在类中可以定义属性和方法，这些内容会在后面做详细的介绍。类名可以使用“大骆驼拼写法”方式来命名（如 SomeClass），以便符合标准 Swift 类型的大写命名风格（如 String，Int 和 Bool）。对于后面所讲的对象、属性以及方法等可以使用“小骆驼拼写法”来命名。

【示例 8-1】以下创建一个名为 NewClass 的类。代码如下：

```
class NewClass{

}
```

该类名称为 NewClass。由于其中没有属性和方法，所以它只是一个空类。

8.1.3 实例化对象

实例化对象也可以称为类的实例，其语法形式如下：

```
var/let 对象名=类名()
```

【示例 8-2】以下会创建一个类名为 NewClass 的类，然后再进行实例化。代码如下：

```
import Foundation
class NewClass{

}
let newClass=NewClass ()
```

注意：在进行实例化时，类名后一定要加上()。否则程序就会错误，如以下的代码：

```
var newClass =NewClass
```

由于在实例化时缺少了()，导致程序出现以下的错误信息：

```
Expected member name or constructor call after type name
```

以上所讲的这些只是简单的实例化对象。它使用了最简单的构造器来生成一个对象。在后面的章节中我们会为开发者讲解构造器的具体用法。

8.2 属性

在 Objective-C 中，属性是使用关键字@property 关键字进行声明的内容。在 Swift 中，属性可以将值跟特定的类、结构或枚举关联。属性一般分为存储属性、计算属性和类型属性。本节将讲解对这些属性做详细的讲解。

8.2.1 存储属性

存储属性就是存储特定类中的一个常量或者变量。根据数据是否可变，分为常量存储属性和变量存储属性。

1.定义存储属性

常量存储属性使用 let 关键字定义（声明定义在一起进行，为了方便称为定义），其语法形式如下：

```
let 常量存储属性名:数据类型=初始值
```

变量存储属性可以使用 var 关键字定义，其语法形式如下：

```
var 变量存储属性名:数据类型=初始值
```

【示例 8-3】以下代码定义类 NewClass1，其中包含两个属性 value1 和 value2，代码如下：

```
class NewClass1 {
    let value1=20
    var value2:Int=10
}
```

其中，value1 使用 let 定义为常量存储属性，value2 使用 var 定义为变量存储属性。在定义存储属性时，初始值是必不可少的，否则，就会出现错误。例如，以下的代码：

```
class NewClass1 {
    let value1=20
    var value2:Int
}
```

在此代码中，由于 value2 后面未加初始值，导致程序出现以下的错误信息：

```
Class 'NewClass1' has no initializers
```

2.访问存储属性

对于这些存储属性的访问，需要使用“.”点运算符。其语法形式如下：

```
对象名.常量存储属性名/变量存储属性名
```

【示例 8-4】以下定义了 3 个存储属性 firstValue、secondValue、thirdValue，然后进行访问。代码如下：

```
import Foundation
class NewClass{
    let firstValue:Int = 0
    let secondValue=200
    var thirdValue:String="Hello"
}
let newclass=NewClass()
//存储属性的访问
println("firstValue=\(newclass.firstValue)")
println("secondValue=\(newclass.secondValue)")
println("thirdValue=\(newclass.threeValue)")
```

运行结果如下所示：

```
firstValue=0
secondValue=200
thirdValue=Hello
Program ended with exit code: 0
```

注意：对存储属性进行访问时，只可以对在自己类中定义的存储属性进行访问，否则就会出现错误，代码如下：

```
import Foundation
class NewClass1 {
   var class1Value=10
}
class NewClass2 {
    var class2Value=10
}
let newclass1=NewClass1()
println(newclass1.class1Value)
println(newclass1.class2Value)
```

在此代码中，由于 class2Value 存储属性是在 NewClass2 类中定义的，而不是 NewClass1 中定义的，所以程序就会出现以下的错误信息：

```
'NewClass1' does not have a member named 'class2Value'
```

存储属性除了可以使用“.”点运算符进行读取外，还可以对其进行修改。修存储改属性的一般语法形式如下：

```
对象名.存储属性=修改的内容
```

【示例 8-5】以下代码就将 secondValue 的属性值"Hello"修改为了"Swift"，代码如下：

```
import Foundation
class NewClass{
    var secondValue:String="Hello"
}
let newclass=NewClass()
println("修改前：secondValue=\(newclass.secondValue)")
newclass.secondValue="Swift"                                    //修改存储实现
println("修改后：secondValue=\(newclass.secondValue)")
```

运行结果如下所示：

```
修改前：secondValue=Hello
修改后：secondValue=Swift
Program ended with exit code: 0
```

注意：只有变量存储属性才可以进行属性修改，常量存储属性不可以进行属性修改。如以下的代码：

```
import Foundation
class NewClass{
    let firstValue:Int = 0
}
let newclass=NewClass()
println("修改前：firstValue=\(newclass.firstValue)")
newclass.firstValue=100                                    //试图对属性 firstValue 的值进行修改
println("修改后：\(newclass.firstValue)")
```

由于在类中使用了 let 对存储属性进行了定义，其值是不可以进行修改的，所以出现了以下的错误信息：

```
Cannot assign to 'firstValue' in 'newclass'
```

3.延迟存储属性

如果开发者只有在第一次调用存储属性时才能确定初始值，这时需要使用延迟存储属性实现。它的定义一般需要使用关键字@lazy 实现的，其语法形式如下：

```
@lazy var  属性名:数据类型=初始内容
```

注意：在延迟存储属性中初始内容是不可以省去的。数据类型也是可以省去的，因为 swift 会根据初始内容自行判断数据类型。

【示例 8-6】以下将使用@lazy 来定义一个延迟存储属性 importer，代码如下：

```
import Foundation
class DataImporter {
    var fileName = 123456
}
class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
}
let manager = DataManager()
manager.data += "Some more data"
```

```
println(manager.data)
println(manager.importer.fileName)
```

在没有调用 manager.importer.fileName 时，实例的 importer 属性还没有被创建。运行结果如下所示：

```
[Some more data]
123456
Program ended with exit code: 0
```

我们可以使用断点调试的方法对此代码进行调试，来查看它的运行结果。具体步骤如下：

（1）为几行关键代码添加断点，如图 8.2 所示。

```
import Foundation
class DataImporter {
    var fileName = 123456
}
class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
}
let manager = DataManager()
manager.data += "Some more data"
println(manager.data)
println(manager.importer.fileName)
```

图 8.2　添加断点

（2）单击运行按钮，此时会在第一个断点处出现一个蓝色的箭头，表示此行代码在运行。然后选择 Debug|Continue 命令，按下调试窗口工具栏中的 Continue program execution 按钮，查看程序的执行，其中，可以使用查看器来观察属性值的变化。属性查看器位于调试信息窗口左半部分，如图 8.3 所示。程序的执行，如图 8.4 所示。其中，看到的 self、data、importer.storage（因为 importer 是延迟属性，为了和其他属性区分，所以在查看器上看到是 importer.storage）等都是属性。它们会随程序的执行为改变。

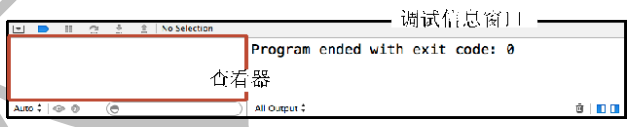

图 8.3　查看器位置

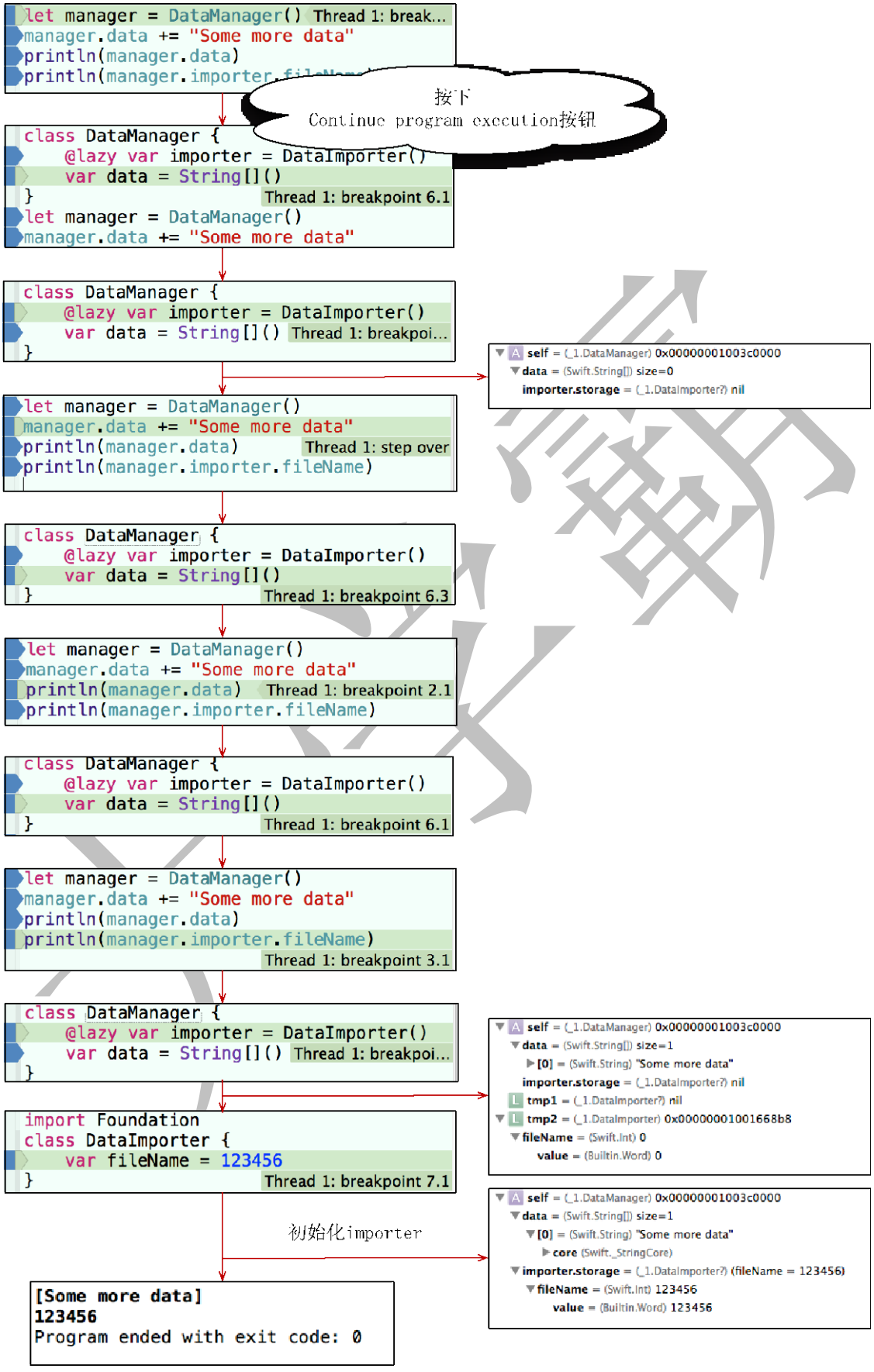
let manager = DataManager() Thread 1: break...
manager.data += "Some more data"
println(manager.data)
println(manager.importer
按下
Continue program execution按钮
class DataManager {
@lazy var importer = DataImporter()
var data = String[]()
}
Thread 1: breakpoint 6.1
let manager = DataManager()
manager.data += "Some more data"
class DataManager {
@lazy var importer = DataImporter()
var data = String[]() Thread 1: breakpoi...
}
self = (_1.DataManager) 0x00000001003c0000
data = (Swift.String[]) size=0
importer.storage = (_1.DataImporter?) nil
let manager = DataManager()
manager.data += "Some more data"
println(manager.data) Thread 1: step over
println(manager.importer.fileName)
class DataManager {
@lazy var importer = DataImporter()
var data = String[]()
}
Thread 1: breakpoint 6.3
let manager = DataManager()
manager.data += "Some more data"
println(manager.data) Thread 1: breakpoint 2.1
println(manager.importer.fileName)
class DataManager {
@lazy var importer = DataImporter()
var data = String[]()
}
Thread 1: breakpoint 6.1
let manager = DataManager()
manager.data += "Some more data"
println(manager.data)
println(manager.importer.fileName)
Thread 1: breakpoint 3.1
class DataManager {
@lazy var importer = DataImporter()
var data = String[]() Thread 1: breakpoi...
}
self = (_1.DataManager) 0x00000001003c0000
data = (Swift.String[]) size=1
[0] = (Swift.String) "Some more data"
importer.storage = (_1.DataImporter?) nil
tmp1 = (_1.DataImporter?) nil
tmp2 = (_1.DataImporter) 0x00000001001668b8
fileName = (Swift.Int) 0
value = (Builtin.Word) 0
import Foundation
class DataImporter {
var fileName = 123456
}
Thread 1: breakpoint 7.1
初始化importer
self = (_1.DataManager) 0x00000001003c0000
data = (Swift.String[]) size=1
[0] = (Swift.String) "Some more data"
core (Swift._StringCore)
importer.storage = (_1.DataImporter?) (fileName = 123456)
fileName = (Swift.Int) 123456
value = (Builtin.Word) 123456
[Some more data]
123456
Program ended with exit code: 0

图 8.4　调试

在此图中程序执行到第 3 步，也就是第三个图时，才为类 DataManager 的属性 data 初始化，但没有给 importer.storage 属性进行初始化，一直为 nil，直到执行到第 9 步，即第 9 个图时，才为 importer.storage 属性的属性初始化。

在定义一个延迟存储属性时需要注意以下 2 点，

（1）定义一个延迟存储属性时除了@lazy 外，还需要使用 var 关键字，但是不能使用 let 关键字，否则程序就会出现错误，如以下代码：

```
class DataImporter {
    var fileName = 123456
}
class DataManager {
    @lazy let importer = DataImporter()
    var data = String[]()
}
```

由于在定义延迟属性时使用了 let 关键字，所以导致程序出现了以下的错误：

```
'lazy' attribute cannot be used on a let
```

（2）初始内容是不可以省去的，否则程序就会出现错误，如以下的代码，定义了一个没有初始值的延迟属性。代码如下:

```
class DataManager {
    @lazy var value:Int
}
```

由于在此代码中没有为 value 指定初始值，导致程序出现了以下的错误：

```
@lazy properties must have an initializer
```

8.2.2 计算属性

除了存储属性外，类中还可以定义计算属性。计算属性不存储值，而是提供了一个 getter 和 setter 来分别进行获取值和设置其他属性的值。getter 使用 get 关键字进行定义，其一般形式如下：

```
get{
  …
  return 某一属性值
}
```

setter 使用 set 关键字进行定义，其一般语法形式如下：

```
set(参数名称){
  …
  属性值=某一个值
  …
}
```

当然，它也可以没有参数名称。这种情况会后面的内容中讲解。在计算属性中同时包含了 getter 和 setter，其一般定义形式如下：

```
var 属性名:数据类型{
    get{
        …
        return 某一属性值
    }
    set(参数名称){
        …
        属性值=某一个值
```

```
            …
    }
}
```

【示例 8-7】以下代码定义了一个类 WalletClass，用来保存钱包中的金额，其默认单位为美元。为了方便用户以人民币为单位进行访问值和设置值，所以使用了计算属性 cal。代码如下：

```
import Foundation
class WalletClass{
    var money=0.0
    var cal:Double{                                    //定义计算属性 cal
        get{                                           //定义 getter
            var RMB=money*6.1
            return RMB                                 //返回以人民币为单位的金额
        }
        set(RMB){                                      //定义 setter
            money=RMB/6.1                              //返回以美元为单位的金额
        }
    }
}
var mywallet=WalletClass()
mywallet.cal=(20)
//输出
println(mywallet.cal)
println(mywallet.money)
```

运行结果如下所示：

```
20.0
3.27868852459016
Program ended with exit code: 0
```

注意：在使用计算属性时需要注意以下三点：

1.定义计算属性的关键字

在定义一个计算属性时，必须且只能使用 var 关键字，否则就会出现错误。以下的代码，将示例 8-7 的代码做了一些修改，代码如下：

```
class WalletClass{
    var money=0.0
    let cal:Double{
        get{
            var RMB=money*6.1
            return RMB
        }
        set(RMB){
            money=RMB/6.1
        }
    }
}
```

在此代码中定义一个计算属性，但是使用了 let 关键字，导致程序出现了以下的错误：

```
'let' declarations cannot be a computed property
```

2.数据类型

在定义计算属性时，一定要为属性指定一个明确的数据类型，否则就会出现错误提示。例如以下的代码，是将示例 8-7 中的代码做了一些修改。代码如下：

```
class WalletClass{
```

```
    var money=0.0
    var cal{                                                    //没有设置 cal 的数据类型
        get{
            var RMB=money*6.1
            return RMB
        }
        set(RMB){
            money=RMB/6.1
        }
    }
}
```

在此代码中由于没有为计算属性指定一个明确的数据类型，导致程序出现了以下的错误信息：

```
Variable with getter/setter must have an explicit type
```

3.set 后面的参数类型

在使用计算属性时，set 后面的参数类型要和返回值的类型相同，不需要再指定类型。否则，程序就会出现错误。如以下的代码，此代码是将示例 8-7 中的代码做了一下修改，代码如下：

```
class WalletClass{
    var money=0.0
    var cal:Double{                                             //没有设置 cal 的数据类型
        get{
            var RMB=money*6.1
            return RMB
        }
        set(RMB:String){                                        //为参数定义了数据类型
            money=RMB/6.1
        }
    }
}
```

在此代码中，对 set 后面的参数 RMB 指定了数据类型，导致程序出现了以下的错误：

```
Expected ')' after setter value name
Expected '{' to start setter definition
```

3.没有定义参数名称

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 newValue。

【示例 8-8】以下代码就使用了 newValue 来实现了华氏温度和摄氏温度的转换。代码如下：

```
import Foundation
class DegreeClass{
    var degree=0.0
    var cal :Double{
        get{
            var centigradedegree=(degree-32)/1.8
            return centigradedegree
        }
        set{
            degree=1.8*newValue+32                              //没有定义参数名称，可以使用默认的
        }
    }
```

```
}
var degreeClass=DegreeClass()
degreeClass.cal=(10.0)
println(degreeClass.cal)
println(degreeClass.degree)
```

运行结果如下所示：

```
10.0
50.0
Program ended with exit code: 0
```

4.定义参数名后不能使用默认参数名

在 set 后面如果定义了参数名称，就不能再使用 Swift 默认的参数名称 newValue。否则，就会导致程序出现错误。如以下的代码，将示例 8-8 做了一些修改，代码如下：

```
import Foundation
class DegreeClass{
    var degree=0.0
    var cal :Double{
        get{
            var centigradedegree=(degree-32)/1.8
            return centigradedegree
        }
        set(aaa){                                    //定义参数名称后，
使用默认参数
            degree=1.8*newValue+32
        }
    }
}
...
```

在此代码中，set 后面定义了参数名称，但是又使用了默认的参数名称，导致程序出现了以下的错误：

```
Use of unresolved identifier 'newValue'
```

5.setter 和 getter 的省略

在计算属性中，如果只有一个 getter，则称为只读计算属性。只读计算属性可以返回一个值，但不能设置新的值。

【示例 8-9】以下将通过 getter 来获取名称的字符串。代码如下：

```
import Foundation
class PersonName{
    var name:String=""
    var returnName :String{
        if (name.isEmpty) {
            return "NULL"
        }else{
            return name
        }
    }
}
var personClass=PersonName()
println("没有名字时\(personClass.returnName)")
personClass.name=("Tom")
println("有名字时\(personClass.returnName)")
```

在此代码中，当刚创建实例 PersonName 后，就去访问 returnName。由于 name 默认为

空，所以会返回字符串"NULL"。再对 name 赋值后，再一次访问 returnName。由于 name 不为空，所以会返回 name 的内容。运行结果如下所示：

```
没有名字时 NULL
有名字时 Tom
Program ended with exit code: 0
```

注意：1.在只读计算属性中，可以将 get 关键字和花括号去掉。2.在 C#等其他语言中可以将属性分为只读属性（只有 getter）、只写属性（只有 setter）和可读可写属性。但是在 Swift 中就不同了，只有只读计算属性和可读可写计算属性两个。没有只写计算属性，否则程序就会出现错误，如以下的代码：

```
class PersonName{
    var name:String=""
    var setName :String{
        set{
          …
        }
}
```

在此代码中定义了一个只写计算属性，导致程序出现了以下的错误：

```
Variable with a setter must also have a getter
```

8.2.3 类型属性

类型属性就是不需要对类进行实例化就可以使用的属性。它需要使用关键字 class 进行定义，其定义形式如下：

```
class var 类型属性名:数据类型{
    …
    返回一个值
}
```

例如下面代码定义了一个类型属性 count，代码如下：

```
class var count:Int{
     return 20
}
```

类型属性也是可以被访问的，其访问类型属性的一般形式如下：

```
类名.类型属性
```

【示例 8-10】以下代码定义了一个类型属性 newvalue，然后进行遍历访问该属性的每一个字符，并输出。代码如下：

```
import Foundation
class NewClass {
    class var newvalue:String{                          //定义类型属性 newvalue
        return "Hello"
    }
}
println(NewClass.newvalue)
println("遍历 NewClass.newvalue：")
//遍历类型属性 newvalue 的值
for index in NewClass.newvalue {
    println(index)
}
```

运行结果如下所示：

```
Hello
```

```
遍历 NewClass.newvalue：
H
e
l
l
o
Program ended with exit code: 0
```

在使用类型属性时需要注意以下 2 点：

1.let 关键字不能声明类型属性

定义类型属性时除了有关键字 class 外，还需要使用 var 关键字，但不能使用 let 关键字，否则程序提示错误，如以下代码，此代码定义了一个类型属性 newvalue。代码如下：

```
import Foundation
class NewClass {
    class let newvalue:Int{
        return 20
    }
}
println(NewClass.newvalue)
```

在此代码中使用了关键字 let 进行了类型属性的声明，导致程序出现了以下的错误：

```
'let' declarations cannot be a computed property
```

2.存储属性

在类型方法中不能使用存储属性，否则程序就会出现错误，如以下的代码，此代码实现的是输出字符串"Hello"。

```
import Foundation
class NewClass {
    var count:Int=20
    class var newvalue:Int{
        return count
    }
}
println(NewClass.newvalue)
```

其中，count 是存储属性，newvalue 是类型属性。在代码中，将 str 用在 newvalue 中，导致程序出现了以下的错误：

```
'NewClass.Type' does not have a member named 'count'
```

3.对象不能访问类型属性

类型属性只可以使用类去访问，而不可以使用对象进行访问。否则，就会出现错误，如以下的代码：

```
import Foundation
class NewClass {
    class var newvalue:Int{
        return 20
    }
}
var newClass=NewClass()
println(newClass.newvalue)
```

在此代码中，定义了一个类型属性 newvalue，但在访问它时使用了对象，导致程序出现了以下错误：

```
'NewClass' does not have a member named 'newvalue'
```

类型属性和存储属性一样，除了可以进行访问外，还可以进行修改，其语法形式如下：

```
类名.类型属性=修改的内容
```

【示例 8-11】以下程序将类型属性 0 变为 200，并输出。代码如下所示：

```
import Foundation
var value:Int=0
class NewClass{
    class var count :Int{
        get{
            var newvalue=value
            return newvalue
        }
        set{
            value=newValue
        }
    }
}
println("修改前：\(NewClass.count)")
NewClass.count=200
println("修改后：\(NewClass.count)")
```

运行结果如下所示：

```
修改前：0
修改后：200
Program ended with exit code: 0
```

8.2.4 属性监视器

属性监视器用来监控和响应属性值的变化。每次属性被设置值的时候，都会调用属性监视器，哪怕是新的值和原先的值相同。一个属性监视器由 willSet 和 didSet 组成，其定义形式如下：

```
var 属性名:数据类型=初始值{
    willSet(参数名){
        …
    }
    didSet(参数名){
        …
    }
}
```

其中，willSet 在设置新的值之前被调用，它会将新的属性值作为固定参数传入。didSet 在新的值被设置之后被调用，会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 oldValue。

【示例 8-12】以下将使用属性监视器监视 totalSteps 属性值的变化。代码如下：

```
import Foundation
class StepCounter {
var totalSteps: Int = 0 {
    //完整的属性监视器
        willSet(newTotalSteps) {
            println("新的值为 \(newTotalSteps)")
        }
        didSet(old) {
```

```
                if totalSteps > old {
                        println("与原来相比增减了 \(totalSteps - old) 个值")
                }
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 0
stepCounter.totalSteps = 200
stepCounter.totalSteps = 400
stepCounter.totalSteps = 800
```

运行结果如下所示：

```
新的值为 0
新的值为 200
与原来相比增减了 200 个值
新的值为 400
与原来相比增减了 200 个值
新的值为 800
与原来相比增减了 400 个值
Program ended with exit code: 0
```

注意：在使用属性监视器时，需要使用注意以下 4 点：

1.不指定参数名

在 willSet 后面是可以不指定参数的，这时 Swift 会使用默认 newValue 表示新值。例如以下的代码在没有指定 willSet 参数的情况下，直接使用 newValue 来输出新的值。代码如下：

```
import Foundation
import Foundation
class StepCounter {
    var totalSteps: Int=0   {
        willSet {
            println("新的值为 \(newValue)")
        }
        didSet (old){
            if totalSteps > old   {
                println("与原来相比增减了 \(totalSteps - old) 个值")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 0
stepCounter.totalSteps = 200
```

运行结果如下所示：

```
新的值为 0
新的值为 200
与原来相比增减了 200 个值
Program ended with exit code: 0
```

同样在 didSet 后面也可以不指定参数名，此时 Swift 会使用默认参数名 oldValue。如以下的代码，此是示例 8-12 的代码做了一个修改，代码如下：

```
import Foundation
class StepCounter {
var totalSteps: Int = 0 {
```

```
    //完整的属性监视器
        willSet(newTotalSteps) {
            println("新的值为 \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
println("与原来相比增减了 \(totalSteps - oldValue) 个值")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 0
…
```

2.默认参数不可以交换使用

在使用 willSet 和 didSet 时，它们默认的参数可以是不可以交换使用的。例如在 willSet 中使用的 newValue 不可以使用在 didSet 中，在 didSet 中使用的 oldValue 不可以使用在 willSet 中，否则程序就会出现错误。例如以下的代码，将示例 8-12 做了一些修改，代码如下：

```
import Foundation
class StepCounter {
var totalSteps: Int = 0 {
    //完整的属性监视器
        willSet {
            println("新的值为 \(newValue)")
        }
        didSet {
            if newValue > oldValue  {
                println("与原来相比增减了 \(newValue - oldValue) 个值")    //输出新值和旧值
之间的差值
            }
        }
    }
}
let stepCounter = StepCounter()
…
```

在此代码中，由于在 didSet 中使用了 willSet 中的默认参数，导致程序出现了以下的错误：

```
Use of unresolved identifier 'newValue'
```

3.延迟属性不能使用属性监视器

在延迟属性中不可以使用属性监视器，否则程序会出现错误。如以下的代码：

```
class StepCounter {
    @lazy var totalSteps: Int=0  {
        willSet {
            println("新的值为 \(newValue)")
        }
        didSet {
            if totalSteps > oldValue  {
                println("与原来相比增减了 \(totalSteps - oldValue) 个值")
            }
        }
    }
```

```
}
```

此代码中延迟属性中添加了属性监视器，导致程序出现了如下的错误：

```
@lazy properties may not have observers
```

4.分开使用 willSet 和 didSet

一个完整的属性监视器由 willSet 和 didSet 组成，但是 willSet 和 didSet 也可以单独使用。例如以下的代码就只使用了 willSet 输出了新值的信息。代码如下：

```
import Foundation
class StepCounter {
    var totalSteps: Int=0   {
          willSet {
               println("新的值为  \(newValue)")
          }
     }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 0
stepCounter.totalSteps = 200
stepCounter.totalSteps = 600
stepCounter.totalSteps = 1200
```

这里属性监视器 total 只使用了 willset，而没有使用 didset。运作结果如下所示：

```
新的值为  0
新的值为  200
新的值为  600
新的值为  1200
Program ended with exit code: 0
```

8.3 方法

方法其实就是函数，只不过它被定义在了类中。在 Swift 中，根据被使用的方式不同，方法分为了实例方法和类型方法两种。这两种方法的定义也和 Objective-C 是不同的。本节依次讲解这两种方法。

8.3.1 实例方法

实例方法被定义在类中，但是由类的实例调用。所以，这类方法被称为实例方法。实例方法和函数一样，分为了不带参数和带参数两种。以下依次讲解这两种方法的使用。

1.不带参数的实例方法

不带参数的实例方法定义和函数的是一样的，其语法形式如下：

```
func 方法名()->返回值类型{
   …
}
```

但它的调用形式和函数的有所不同，其调用形式如下：

```
对象名.方法名()
```

其中，对象名必须代表的是方法所属类的实例。

【示例 8-13】以下将使用方法输出字符串"Hello"。代码如下：

```
import Foundation
class NewClass{
    var str="Hello"
    //方法
    func printHello(){
        println(str)
    }
}
let newClass=NewClass()
newClass.printHello()
```

运行结果如下所示：

```
Hello
Program ended with exit code: 0
```

2.具有参数的实例方法

具有参数的实例方法就是在方法名后面的括号中添加了参数列表。它的定义也和函数一样，定义形式如下：

```
func 方法名(参数 1:数据类型,参数 2:数据类型,…)->返回值类型{
  …
}
```

它的调用形式如下：

```
对象名.方法名(参数 1,参数 2,…)
```

下面依次讲解分别具有一个参数和具有多个参数实例方法的使用方式。

（1）具有一个参数的实例方法

具有一个参数的实例方法是指在参数列表中只有一个参数以及类型。

【示例 8-14】下面定义一个类 Counter。其中包含了 increment()、incrementBy()、reset()方法。代码如下：

```
import Foundation
class Counter {
    var count = 0
    //让 count 加 1
    func increment() {
        count++
        println(count)
    }
    //让 count 加一个指定的数值
    func incrementBy(amount: Int) {
        count += amount
        println(count)
    }
    //将 count 设置为 0
    func reset() {
        count = 0
        println(count)
    }
}
let counter = Counter()
//加 1
counter.increment()
counter.increment()
counter.increment()
//加指定的值
```

```
counter.incrementBy(10)
//设置为 0
counter.reset()
```

其中，increment()方法的功能是自动加 1，在每调用一次此方法时，都会在原来值的基础上增加 1；incrementBy()方法是将当前的值和指定的数相加。运行结果如下所示：

```
//加 1 的结果
1
2
3
//加 10 的结果
13
//重新进行设置后的结果
0
Program ended with exit code: 0
```

（2）具有多个参数的实例方法

当实例方法的参数列表中包含多个参数时，参数和参数之间需要使用“,”逗号分隔。

【示例 8-15】以下将实现两个数和的计算。代码如下：

```
import Foundation
class AddClass {
    var count:Int=0
    //计算两个数的和
    func add(amount1:Int,amount2:Int){
        count=amount1+amount2
        println(count)
    }
}
let counter = AddClass()
counter.add(5,amount2: 10)
```

其中，方法 add()包含两个参数 amount1 和 amount2。调用的时候，分别赋值为 5 和 10。运行结果如下所示：

```
15
Program ended with exit code: 0
```

注意：1.当方法具有两个或者两个以上的参数时，Swift 默认仅给方法的第一个参数名称一个局部参数名称；默认同时给第二个和后续的参数名称一个局部参数名称和外部参数名称，其中局部参数名称和外部参数名称的名称都是一样的。如果在调用时，第二个参数以及后续的参数不使用外部参数名，程序就会出现错误。如下，

```
import Foundation
class AddClass {
var count:Int=0
//计算两个数的和
    func add(amount1:Int,amount2:Int){
        count=amount1+amount2
        println(count)
    }
}
let counter = AddClass()
counter.add(5,10)
```

在此代码中，实例方法在调用时第二个参数没有使用外部参数名，导致程序出现如下的错误：

```
Missing argument label 'amount2:' in call
```

2.如果开发者不想为方法的第二个及后续的参数提供外部名称，可以通过使用“_” 下划线作为该参数的显式外部名称，这样做将覆盖默认行为。

【示例 8-16】以下将实现两个数相乘的结果。代码如下：

```
import Foundation
class MultiplyClass {
    var count:Int=0
    func multiply(amount1:Int, _ amount2:Int){
        count=amount1*amount2
        println(count)
    }
}
let counter = MultiplyClass()
counter.multiply(5, 10)
```

在定义时，第二个参数前加上了“_”下划线。所以调用的时候，第二个参数可以省略外部参数名。运行结果如下所示：

```
50
Program ended with exit code: 0
```

3.对于实例方法的对象，都必须是通过实例去调用的，它不可以和函数的调用方法一样去调用，如以下的代码，此代码实现的功能是在类中定义一个输出字符串的方法，然后去调用，代码如下：

```
import Foundation
class NewClass {
    func printHello(){
        println("Hello")
    }
}
printHello()
```

```
Use of unresolved identifier 'printHello'
```

8.3.2 类型方法

实例方法是被类的某个实例调用的方法。开发者还可以定义类自身调用的方法，这种方法被称为类型方法（因为类被认为是一种类型。所以，属于类的方法，被称为类型方法）。定义类型方法需要在 func 关键字之前加上关键字 class，其一般定义形式如下：

```
class 类名 {
  class func 方法名(参数 1:数据类型, 参数 1:数据类型,…) {
    …
  }
}
```

在定义好类型方法后，就可以进行调用了，其调用的一般形式如下：

```
类名.方法名(参数 1,参数 2,…)
```

注意：这里的方法名为类型方法名。

1.不带参数列表的类型方法

不带参数列表的类型方法就是方法名后面的参数列表中没有参数。以下定义了一个输出字符串"Hello"中字符的类型方法。代码如下：

```
import Foundation
```

```
class NewClass {
    class var str:String{
        return "Hello"
    }
    //类型方法
    class func printHello(){
        for index in str{
            println(index)
        }
    }
}
NewClass.printHello()
```

运行结果如下所示：

```
H
e
l
l
o
Program ended with exit code: 0
```

2.具有参数的类型方法

具有参数的类型方法是具有参数列表方法。随着参数的不同，可以将具有参数的类型方法分为具有一个参数的类型方法和具有多个参数的类型方法。以下就是具有一个参数的类型方法，该方法实现输出任意字符串的字符。代码如下：

```
import Foundation
class NewClass {
    class func printString(str:String){
        for index in str{
            println(index)
        }
    }
}
NewClass.printString("Swift")
```

运行结果如下所示：

```
S
w
i
f
t
Program ended with exit code: 0
```

具有多个参数的类型方法说明在参数列表中有多个参数以及类型，参数和参数之间要使用“,”逗号分隔。以下就是定义了多个参数的类型方法，此方法实现了字符串和字符串的连接功能。代码如下：

```
import Foundation
class NewClass {
   //具有多个参数的类型方法
    class func joinerString(string:String,toString:String,withjoiner:String){
        println("str1、str2、str3 实现关联为：\(string)\(withjoiner)\(toString)")
     }
}
var str1="Hello"
var str2="Swift"
```

```
var str3="——"
println("str1=\(str1)")
println("str2=\(str2)")
println("str3=\(str3)")
NewClass.joinerString(str1,toString: str2,withjoiner: str3)
```

运行结果如下所示：

```
str1=Hello
str2=Swift
str3=——
str1、str2、str3 实现关联为：Hello——Swift
Program ended with exit code: 0
```

注意：在使用类方法时需要注意以下 3 点：

（1）存储属性

在一个类型方法中不可以使用存储属性，否则程序就会出现错误，如以下的代码，此代码实现的是输出字符串"Hello"。

```
import Foundation
class NewClass {
    var str="Hello"
    class func printHello(){
        for index in str{
            println(index)
        }
    }
}
NewClass.printHello()
```

由于 str 是存储属性，而 printHello()是一个类型方法，在此代码中将 str 用在了 printHello()方法中，导致程序出现了以下的错误：

```
'NewClass.Type' does not have a member named 'str'
```

（2）类调用方法

所有的类型方法必须要使用类去调用，而非对象，否则就会出现错误，如以下的代码：

```
import Foundation
class NewClass {
    class var str:String{
        return "Hello"
    }
    class func printHello(){
        for index in str{
            println(index)
        }
    }
}
let newClass=NewClass()
newClass.printHello()
```

由于在此代码中，方法 printHello()是一个实例方法，但是在调用时，使用对象进行了调用，导致程序出现了以下错误：

```
'NewClass' does not have a member named 'printHello'
```

（3）外部参数名

和实例方法相同， Swift 默认给类型方法的第一个参数名称一个局部参数名称；默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。在调用时，一定不要忘记外部参数名。

8.3.3 存储属性、局部变量和全局变量的区别

存储属性可以理解为变量的一种。所以随着变量使用地方的不同，可以将变量分为存储属性、局部变量和全局变量。这三种变量的不同如表 8-1 所示。

表 8-1 三种变量的不同

变量名称	定义范围
存储属性	定义在类中
局部变量	函数、方法或闭包内部
全局变量	函数、方法、闭包或任何类型之外定义的变量

注意：这里提到的方法会在后面进行讲解。

【示例 8-17】在以下定义了三个类型的变量分别为 str、str1、str2，将它们使用在不同的地方。代码如下：

```
import Foundation
let str="Hello"
class NewClass {
    let str1="Swift"
    func printstring(){
        let str2="World"
        println(str)
        println(str1)
        println(str2)
    }
}
let newclass=NewClass()
newclass.printstring()
```

在此代码中提到的方法会在以一节中讲解。运行结果如下所示：

```
Hello
Swift
World
Program ended with exit code: 0
```

注意：在使用存储属性、局部变量、和全局变量时一定要注意它们的作用域，所谓作用域就是指这些变量的有效范围，图 8.5 就是以上代码中变量的有效范围。

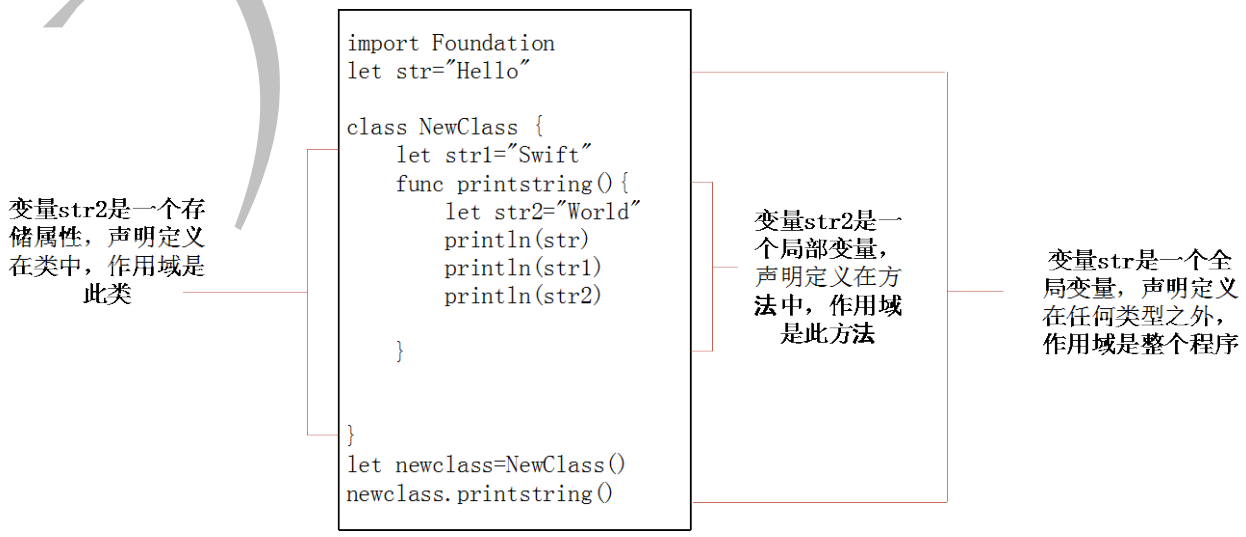

图 8.5 作用域

如果一个变量超出了它的有效范围，程序就会出现错误，如以下的代码：

```
import Foundation
let str="Hello"
class NewClass {
    let str1="Swift"
    func printstring(){
        let str2="World"
        println(str)
        println(str1)
        println(str2)
    }
    func printstr(){
        println(str)
        println(str1)
        println(str2)
    }
}
let newclass=NewClass()
newclass.printstring()
```

在此代码中 str2 是一个定义在 printstring()方法中的局部变量，所以它的有效范围就是在此方法中。但是在此方法之外使用会出现错误，在此代码中将 str2 又使用在了方法 printstr()中。导致程序出现了以下的出现：

```
Use of unresolved identifier 'str2'
```

8.3.4 局部变量和存储属性同名的解决方法——self 属性

在一个类中，方法（方法会在以下节中讲到）中的局部变量可能和存储属性同名，这是，不可以直接使用相同的名称去访问存储属性，为了解决这一问题，Swift 引入了一个 self 属性。如以下代码，此代码的功能是使用 self 对存储属性进行访问。代码如下：

```
import Foundation
class NewClass{
    var count:Int=100
    func printcount(){
        var count:Int=50
        println(count)
        println(self.count)
    }
}
let newClass=NewClass()
newClass.printcount()
```

count 表示的是局部变量的值 50，self.count 表示存储属性的值 100。运行结果如下所示：

```
50
100
Program ended with exit code: 0
```

8.4 下标脚本

下标脚本是访问对象、集合或者序列的快速方式。开发者不需要调用实例特定的赋值和访问方法，就可以直接访问所需要的数值。例如在数组中，可以直接使用下标去访问或者修改数组中的某一个元素。代码如下：

```
import Foundation
var array=["One","Two","Three","Four","Five"]
println("访问元素：\(array[2])")
array[2]="Third"
println("访问修改后的元素：\(array[2])")
```

运行结果如下：

```
访问元素：Three
访问修改后的元素：Third
Program ended with exit code: 0
```

在 Swift 中，下标脚本也可以定义在类中。这样，开发者就可以像数组一样，快速访问类中的属性。本节将主要讲解类中如何使用下标脚本。

8.4.1 定义下标脚本

下标脚本通过 subscript 关键字进行定义，其定义形式如下：

```
subscript(参数名称 1:数据类型,参数名称 2:数据类型,…) ->返回值的数据类型 {
    get {
      // 返回与参数类型匹配的类型的值
    }
    set(参数名称) {
      // 执行赋值操作
    }
}
```

注意：set 参数名称必须和下标脚本定义的返回值类型相同，所以不为它指定数据类型。与计算属性相同，set 后面如果没有声明参数，那么就使用默认的 newValue。

【示例 8-18】以下将在类中定义一个下标脚本，实现通过下标脚本获取某一属性的值。代码如下：

```
class NewClass{
    var english:Int=0
     var chinese:Int=0
     var math:Int=0
     //定义下标脚本
     subscript(index:Int)->Int{
          get{
               switch index{
                    case 0:
                         return english
                    case 1:
                         return chinese
                    case 2:
                         return math
                    default:
```

```
                    return 0
                }
            }
            set{
                english=newValue
                chinese=newValue
                math=newValue
            }
        }
}
```

8.4.2 调用下标脚本

定义下标脚本后，就可以进行调用了，其调用形式如下：

```
实例对象[参数 1,参数 2,…]
```

其中，[]和它里面的内容就代表了在类中定义的下标脚本。

8.4.3 使用下标脚本

下标脚本可以根据传入参数的不同，分为具有一个入参参数的下标脚本和具有多个入参参数的下标脚本。以下就是对这两个下标脚本在类中的使用。

1.具有一个传入参数的下标脚本

具有一个入参参数的下标脚本是最常见的。在集合以及字符串中使用的下标就是具有一个传入参数的下标脚本。

【示例 8-19】以下程序通过使用下标脚本计算 3 门成绩的和。代码如下：

```
import Foundation
class Score{
    var english:Int=0
    var chinese:Int=0
    var math:Int=0
    //定义下标脚本
    subscript(index:Int)->Int{
        get{
            switch index{
                case 0:
                    return english
                case 1:
                    return chinese
                case 2:
                    return math
                default:
                    return 0
            }
        }
        set{
            english=newValue
            chinese=newValue
            math=newValue
```

```
        }
    }
}
var myscore=Score()
var sum:Int=0
var i:Int=0
//遍历
for i=0;i<3;++i{
    sum+=myscore[i]
}
println(sum)
//修改属性值
myscore[0]=100
myscore[1]=90
myscore[2]=80
//遍历求和
for i=0;i<3;++i{
    sum+=myscore[i]
}
println(sum)
```

运行结果如下所示：

```
0
240
Program ended with exit code: 0
```

注意：下标脚本可以和计算属性一样设置为读写或只读。以上的代码是读写的形式。只读的一般语法形式如下：

```
subscript(参数名称:数据类型) -> Int {
    get{
        //返回与参数匹配的 Int 类型的值
    }
}
```

可以简写为以下的形式：

```
subscript(参数名称:数据类型) -> Int {
    // 返回与参数匹配的 Int 类型的值
}
```

【示例 8-20】以下就使用只读的形式实现使用下标访问属性值的功能。代码如下：

```
import Foundation
class Score{
    var english:Int=50
    var chinese:Int=100
    var math:Int=30
    //定义下标脚本
    subscript(index:Int)->Int{
            switch index{
                case 0:
                    return english
                case 1:
                    return chinese
                case 2:
                    return math
                default:
                    return 0
```

```
            }
        }
}
var myscore=Score()
var sum:Int=0
var i:Int=0
//遍历输出属性值
for i=0;i<3;++i{
    println(myscore[i])
}
```

运行结果如下所示：

```
50
100
30
Program ended with exit code: 0
```

2.具有多个参数的下标脚本

具有一个入参参数的下标脚本一般使用在多维维数组中。以下就是使用具有两个参数的下标为二维数组赋值。代码如下：

```
import Foundation
var value:Int=0
class NewClass{
    let rows: Int = 0, columns: Int=0
    var grid: Double[]
    //初始化方法
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    //下标脚本
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
var matrix = NewClass(rows: 2, columns: 2)
println("没有赋值前")
println(matrix[0,0])
println(matrix[0,1])
println(matrix[1,0])
println(matrix[1,1])
println("赋值后")
matrix[0,0]=1.0
```

```
matrix[0,1]=5.6
matrix[1,0]=2.4
matrix[1,1]=3.2
println(matrix[0,0])
println(matrix[0,1])
println(matrix[1,0])
println(matrix[1,1])
```

运行结果如下所示：

```
没有赋值前
0.0
0.0
0.0
0.0
赋值后
1.0
5.6
2.4
3.2
Program ended with exit code: 0
```

当然，下标脚本除了可以对访问对象以及对象中的属性外，还可以实现一些自定义的功能，如以下的代码，此代码实现的功能是计算下标值和 10 的乘积。代码如下：

```
import Foundation
class NewClass{
    var count1: Int=10
    //定义下标脚本
    subscript(index:Int) -> Int {
        get {
            var count=index*count1
            return count
        }
        set(newvalue){
            //执行赋值操作
            count1=newvalue
        }
    }
}
let newClass=NewClass()
println(newClass.count1)
println(newClass[6])
```

运行结果如下：

```
10
60
Program ended with exit code: 0
```

8.5 类的嵌套

在一个类中可以嵌套一个或者多个类。它们的嵌套形式也是不同的，大致分为了两种：直接嵌套和多次嵌套。下面依次讲解这两种方式。

8.5.1 直接嵌套

当一个类或者多个类直接嵌套在另外一个类，这时就构成直接嵌套，如图 8.6 所示。

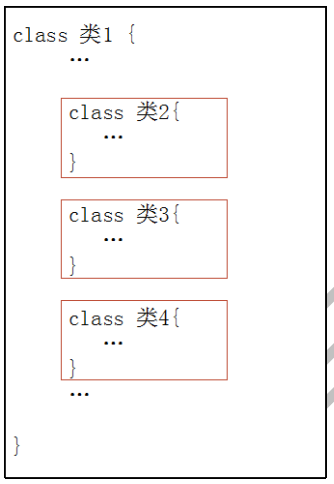

图 8.6 类的嵌套

在图 8.6 中，类 2、类 3 和类 4 都是直接嵌套在类 1 中。对于这种情况，使用类 1 的实例属性和方法，语法形式如下：

```
类 1().属性
类 1().方法
```

使用类 1 的类型属性和方法的形式如下：

```
类 1.属性
类 1.方法
```

使用类 2 的实例属性和方法，语法形式如下：

```
类 1.类 2().属性
类 1.类 2().方法
```

使用类 2 的类型属性和方法的形式如下：

```
类 1.类 2.属性
类 1.类 2.方法
```

类 3 和类 4 的使用方法类似。

【示例 8-21】以下将定义一个直接嵌套的类 NewClass，在此类中嵌套了 Str1Class、Str2Class、Str3Class 这 3 个类，和一个可以输出这 3 个类中属性内容的方法。在 Str1Class、Str2Class、Str3Class 这三个类中又定义了类型属性，它们都会返回一个字符串。代码如下：

```
import Foundation
class NewClass {
    class func printstr(str:String){
        println(str)
    }
    //Str1Class 类
    class Str1Class{
        class var str:String{
            return "Swift"
        }
    }
    //Str2Class 类
    class Str2Class{
```

```
        class var str:String{
            return "Hello"
        }
    }
    //Str3Class 类
    class Str3Class{
        class var str:String{
            return "World"
        }
    }
}
//调用
NewClass.printstr(NewClass.Str1Class.str)
NewClass.printstr(NewClass.Str2Class.str)
NewClass.printstr(NewClass.Str3Class.str)
```

在此代码中，在一个 NewCllass 类中有包含了 3 个类，分别为 Str1Class、Str2Class、Str3Class。运行结果如下所示：

```
Swift
Hello
World
Program ended with exit code: 0
```

注意：它的调用形式如下

8.5.2 多次嵌套

Swift 中，类的嵌套不仅允许一次嵌套，还允许多次嵌套。这时的嵌套形式如图 8.7 所示。

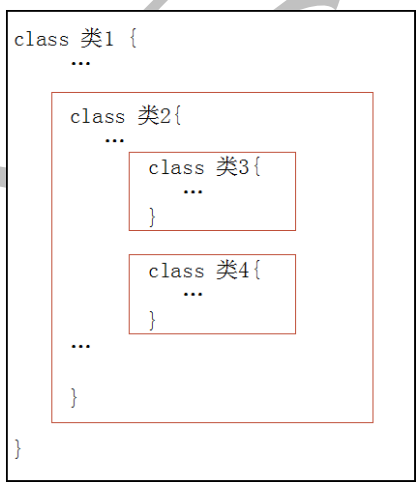

图 8.7 类的嵌套 2

类 3 和类 4 是直接嵌套在类 2 中，而类 2 又直接嵌套在类 1。这样形成了多层嵌套。这时，如果访问类 1 的实例属性和方法，其语法形式如下：

```
类 1().属性
类 1().方法
```

访问类 1 的类型属性和方法，其语法形式如下：

```
类 1.属性
```

```
类 1.方法
```

如果要访问类 2 的实例属性和方法，对应的语法形式如下：

```
类 1.类 2().属性
类 1.类 2().方法
```

访问类 2 的类型属性和方法，对应的语法形式如下：

```
类 1.类 2.属性
类 1.类 2.方法
```

如果要访问类 3 的实例属性和方法，对应的语法形式如下：

```
类 1.类 2.类 3().属性
类 1.类 2.类 3().方法
```

如果要访问类 3 的类型属性和方法，对应的语法形式如下：

```
类 1.类 2.类 3.属性
类 1.类 2.类 3.方法
```

【示例 8-22】以下将定义一个多次嵌套的类 NewClass，在此类中嵌套了 StrClass 类，和一个可以输出属性内容的方法。在 StrClass 类中又嵌套了 Str1Class、Str2Class、Str3Class 这 3 个类，它们都会返回一个字符串。代码如下：

```
import Foundation
class NewClass {
    class func printstr(str:String){
        println(str)
    }
    // StrClass 类
    class StrClass{
        // StrClass1 类
        class Str1Class{
            class var str:String{
            return "Hello"
            }
        }
        // StrClass2 类
        class Str2Class{
            class var str:String{
            return "Swift"
            }
        }
        // StrClass3 类
        class Str3Class{
            class var str:String{
            return "World"
            }
        }
    }
}
//调用
NewClass.printstr(NewClass.StrClass.Str1Class.str)
NewClass.printstr(NewClass.StrClass.Str2Class.str)
NewClass.printstr(NewClass.StrClass.Str3Class.str)
```

在此代码中，在一个 NewCllass 类中有包含了 1 个类 StrClass，在 StrClass 类中又包含了 3 个类，分别为 Str1Class、Str2Class、Str3Class。运行结果如下所示：

```
Hello
Swift
```

```
World
Program ended with exit code: 0
```

8.6 可选链接

在类或者其他的类型中，声明的属性和变量/常量，都不可以为空。为了解决这一问题，Swift 提出了可选类型。通过可选类型定义的元素可以为空或者是不为空，但是如何要使用这些可选类型的值又成为一大难题。Swift 接着就提出了可选链接。可选链接可以判断请求或调用的目标（属性、方法、下标脚本等）是否为空。如果目标有值，那么调用就会成功；相反，则返回空（nil）。对于多次请求或调用的可以被链接在一起形成一个链条。Swift 中的可选链接和 Objective-C 中的消息为空类似，但是 Swift 可以使用在任意的类型中使用，并且失败与否可以被检测到。以下将详细讲解可选链接的内容。

8.6.1 可选链接的实现方式

可选链接其实就是使用“?”问号操作符对可选类型实现的一种运算。开发者可以在想要调用的属性、下标脚本和方法的可选值后面添加一个“?”问号来进行可选链接的定义。以下就是对于这些可选链接的定义形式：

```
属性名?                                    //属性的可选链接
下标脚本?                                  //下标脚本的可选链接
方法名?                                    //方法的可选链接
```

对象可选链接的调用形式如下：

```
对象名.可选链接
```

【示例 8-23】以下将定义属性的可选链接，其代码如下：

```
import Foundation
class Residence {
    //定义一个可选类型的类型属性 numberOfRooms
    class var numberOfRooms:Int?{
        return 100
    }
    var number:Int?
}
let newClass=Residence()
if Residence.numberOfRooms? {                          //调用可选链接
    println("目标有值")
}else{
    println("目标为空")
}
if newClass.number? {                                  //可选链接
    println("目标有值")
}else{
    println("目标为空")
}
```

在此代码中，由于 umberOfRooms 的属性值不为空，所以会出现"目标有值"，但是对于 number 属性来说，没有赋初值即 number 属性为 nil，所以会出现"目标为空"运行结果如下所示：

```
目标为空
目标有值
Program ended with exit code: 0
Program ended with exit code: 0
```

注意：在定属性、下标脚本以及方法定义为可选链接时，这些属性、下标脚本和方法都必须是可选类型，否则程序就会出现以下的错误。如以下的代码，是对属性进行的可选链接：

```
import Foundation
class NewClass{
    var value:Int=10
}
let newClass=NewClass()
let newValue=newClass.value?
println(newValue)
```

由于在此代码中对一个不是可选类型的属性进行了可选链接定义，导致程序出现了以下的错误：

```
Operand of postfix '?' should have optional type; type is 'Int'
```

8.6.2 通过可选链接调用属性、下标脚本、方法

开发者可以使用可选链接的可选值来调用属性、下标脚本和方法，并检查这些内容调用是否成功。以下就是通过可选链接调用属性、下标脚本、方法的详细讲解。

1.通过可选链接调用属性

通过可选链接调用属性的语法形式如下：

```
可选链接.属性名
```

【示例 8-24】以下将通过自判断可选链接来调用属性值，并获取这个属性值。代码如下：

```
import Foundation
class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms = 10
}
let john = Person()
if let roomCount = john.residence?.numberOfRooms {                    //通过可选链接调用属性
    println("John 在房子中有 \(roomCount)个房间")
} else {
    println("无法检索房间数")
}
```

由于 john.residence 是空，所以这个可选链接就会失败，但是不会出现错误，会返回一个 nil。运行结果如下：

```
无法检索房间数
Program ended with exit code: 0
```

如果不行返回 nil，开发者需要将 john.residence 设置为不为空。如以下的代码，将 john.residence 设置为 johnResidence。代码如下：

```
let john = Person()
let johnResidence = Residence()
```

```
john.residence=johnResidence
if let roomCount = john.residence?.numberOfRooms {                    //通过可选链接调用属性
    println("John 在房子中有 \(roomCount)个房间")
} else {
    println("无法检索房间数")
}
```

运行结果如下：

```
John 在房子中有 10 个房间
Program ended with exit code: 0
```

2. 通过可选链接调用下标脚本

通过可选链接调用下标脚本的语法形式如下：

```
可选链接.[下标]
```

【示例 8-25】以下将通过自判断可选链接来调用下标脚本。代码如下：

```
import Foundation
class Person {
    var residence: Residence?
}
class Residence {
    subscript(i: Int) -> Int {
        return i
    }
}
let john = Person()
if let firstRoomName = john.residence?[5] {                    //通过可选链接调用下标脚本
    println("John 在房子中有 \(firstRoomName)个房子")
} else {
    println("无法检索房间数")
}
```

运行结果如下所示：

```
无法检索房间数
Program ended with exit code: 0
```

注意：当开发者使用可选链来调用子脚本的时候，你应该将“?”问号放在下标脚本括号的前面而不是后面。可选链的问号一般直接跟在自判断表达语句的后面。否则程序就会出现错误。如以下的代码就将上面的代码做了一下修改，代码如下：

```
let john = Person()
if let firstRoomName = john.residence[5]? {                    //通过可选链接调用下标脚本
    println("John 在房子中有 \(firstRoomName)个房子")
} else {
    println("无法检索房间数")
}
```

在此代码中就“?”问号放在了下标脚本括号得后面，导致程序出现了以下的错误：

```
'Residence?' does not have a member named 'subscript'
```

3.通过可选链接调用方法

通过可选链接调用方法的语法形式如下：

```
可选链接.方法
```

【示例 8-26】以下将通过自判断可选链接来调用方法 printNumberOfRooms()，此方法的功能是输出 numberOfRooms 的值。代码如下：

```
import Foundation
class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms=10
    func printNumberOfRooms() {
        println("The number of rooms is \(numberOfRooms)")
    }
}
let john = Person()
if john.residence?.printNumberOfRooms() {                    ////通过可选链接调用方法
    println("打印房间数")
} else {
    println("无法打印房间数")
}
```

运行结果如下：

```
无法打印房间数
Program ended with exit code: 0
```

8.6.3 连接多个链接

开发者可以将多个可选链接放在一起，如以下的代码，此代码实现的功能是将获取属性 street 的内容。代码如下：

```
import Foundation
// Person 类，定义了属性 residence
class Person {
    var residence: Residence?
}
// Residence 类，定义了属性 address
class Residence {
    var address: Address?
}
// Address 类定义了属性 street
class Address {
    var street: String?
}
//实例化对象
let john = Person()
let johnsHouse = Residence()
john.residence = johnsHouse
let johnsAddress = Address()
//赋值
johnsHouse.address=johnsAddress
johnsAddress.street = "Laurel Street"
if let johnsStreet = john.residence?.address?.street {          //链接了两个可选链
接
    println("John 的地址为: \(johnsStreet)")
} else {
    println("无法检索地址")
}
```

在此代码中，john.residence 现在存在一个实例 johnsHouse，而不是 nil，而 john.residence?.address 现实也存在一个实例 johnsAddress，并为 street 设置了实例的值。所以在执行程序后，会返回 Street 的值。运行结果如下：

```
John 的地址为: Laurel Street
Program ended with exit code: 0
```