第 5 章

枚举与结构体

在 Swift 语言中，数据类型包括基本类型、组合类型和自定义类型三种，其中，基本类型包括整型、字符型、字符串型、单精度浮点型、双精度浮点型、布尔型等；组合类型包括元组、集合、数组和字典等；自定义类型包括枚举、结构体和类等。在自定义类型中，枚举和结构体属于值类型，而类属于引用类型。此外，在 Swift 语言中所有基本类型和集类型均属于值类型，这种类型的量在复制或传递给函数的参数时，将其值复制一个副本给新的量，原来的量和新量间不再有关系。然而，引用类型的量在赋值给另一个量时，原来的量和新量均指向同一个实例。本章将介绍枚举类型和结构体的概念与用法，第 6 章介绍类与实例的概念与用法。

5.1 枚举

枚举是一种自定义类型，也是一种值类型。枚举用于定义一组相互关联且可列举的量，以增加程序的可读性。枚举类型的基本语法为

```
enum 枚举类型名
{
   case 枚举值 1, 枚举值 2, …, 枚举值 n
}
```

这里，enum 为定义枚举类型的关键字；枚举类型名一般使用“大骆驼命名法”，即首字母大小、名称中包含的每个英文单词的首字母均大写；各个枚举值的名称不能相同。

例如，下面的枚举类型 Week 定义了一周的情况：

```
enum Week
{
case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}
```

注意：枚举值一般使用“小骆驼命名法”，即首字母小写、名称中包含的每个英文单词的首字母大写。但是，这里表示一星期中的每天的英文单词都是首字母大写，为了和真实的英语单词表示统一。

给定枚举类型后，定义枚举类型的变量或常量的方法与使用基本类型定义变量或常量的方法相同，即分别借助 var 和 let 关键字。

程序段 5-1 展示了枚举的基本用法。


视频讲解

程序段 5-1 枚举基本用法实例

```
1    import Foundation
2
```

```
3    enum Week
4    {
5        case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
6    }
7    var day = Week.Thursday
8    switch day
9    {
10   case .Monday, .Tuesday, .Wednesday, .Thursday, .Friday:
11       print("We work at \(day).")
12   case .Saturday, .Sunday:
13       print("We have a rest at \(day).")
14   }
```

在程序段 5-1 中，第 3～6 行定义了枚举类型 Week。第 7 行等价于“var day：Week＝Week. Thursday”或“var day：Week＝. Thursday”，如果可从上下文中知道枚举类型名，可省略枚举类型名，这里表示定义变量 day，赋初值为“Week. Thursday”。

第 8～14 行为一个 switch 结构，根据 day 的值选择相应的分支执行。由于 day 为“Week. Thursday”，故匹配了第 10 行的 case，从而第 11 行“print("We work at \(day).")”被执行，输出“We work at Thursday.”。每个枚举类型的量，在默认情况下为它本身表示的字符串，这里的 day 为“Week. Thursday”，故 day 的值为“Thursday”。

5.1.1　枚举量原始值

在 Swift 语言中，枚举类型中的枚举量可具有同一类型的值，称为原始值，原始值可为字符串型、整型、字符型、单精度或双精度浮点型等，有以下几种情况。

(1) 字符串类型的枚举类型，原始值为各枚举量的文本，例如：

```
enum Polarity : String
{
case negative, positive
}
```

此时，Polarity. negative 具有默认的原始值 negative，Polarity. positive 具有默认的原始值 positive。在程序段 5-1 中，枚举类型 Week 的各个枚举量也属于这类情况。

(2) 整数类型的枚举类型，第一个枚举量的原始值为 0，后续的枚举量的原始值依次加 1，例如：

```
enum DNACode : Int
{
case A, G, C, T
}
```

此时，DNACode. A 具有默认的原始值 0，DNACode. G 的原始值为 1，DNACode. C 的原始值为 2，DNACode. T 的原始值为 3。

(3) 整数类型的枚举类型，可以为每个枚举量指定整数值，此时指定的值即为这些枚举量的原始值；也可以只为第一个枚举量指定整数值作为其原始值，后续的各个枚举量的原始值依次加 1，例如：

```
enum Season : Int
{
case spring = 1
case summer
```

```
case autumn
case winter
}
```

这里，枚举类型的各个枚举量可以使用多个 case，上述 Season. spring 的原始值被指定为 1，则 Season. summer 的原始值为 2，Season. autumn 的原始值为 3，Season. winter 的原始值为 4。

(4) 除上述三种情况，当指定枚举类型的变量类型时，必须指定相同的类型，可单独为每个枚举量设定原始值，例如：

```
enum Constant : Double
{
case pi = 3.14159
case e = 2.71828
}
```

这里枚举类型 Constant 的每个枚举量都被指定了原始值。

当为枚举类型的枚举量指定了原始值后，枚举类型将隐式包含一个“初始化器”，所谓的“初始化器”是指由枚举类型名作为函数名、rawValue(即原始值)作为参数的函数，用于生成一个枚举类型的变量或常量。例如，对于上述 Constant 枚举类型，可以使用它的初始化器定义一个枚举变量 c，即“var c＝Constant(rawValue:3. 14159)”。注意，初始化器返回的类型为可选类型，对于 Constant 枚举类型，返回的类型为可选双精度浮点类型。


视频讲解

程序段 5-2 展示了带有原始值的枚举类型的用法。

程序段 5-2 带原始值的枚举类型用法实例

```
1   import Foundation
2
3   enum Polarity : String
4   {
5   case negative, positive
6   }
7   var p = Polarity(rawValue: "positive")
8   print("\(p!), \(p!.rawValue)")
9   enum DNACode : Int
10  {
11  case A, G, C, T
12  }
13  var dna = DNACode(rawValue: 3)
14  print("\(dna!),\(dna!.rawValue)")
15  enum Season : Int
16  {
17  case spring = 1
18  case summer
19  case autumn
20  case winter
21  }
22  var s = Season.summer
23  print("\(s),\(s.rawValue)")
24  enum Constant : Double
25  {
26  case pi = 3.14159
27  case e = 2.71828
28  }
```

```
29  var c1 = Constant(rawValue: 3.14159)
30  print("\(c1!),\(c1!.rawValue)")
31  var c2 = Constant.pi
32  print("\(c2),\(c2.rawValue)")
33  if let c3 = Constant(rawValue: 2.71828)
34  {
35      print("\(c3.rawValue)")
36  }
```

在程序段 5-2 中，第 3～6 行定义了枚举类型 Polarity，为字符串类型。第 7 行“var p＝Polarity(rawValue:"positive")”使用枚举类型 Polarity 的初始化器定义枚举变量 p，此时的 p 为可选枚举类型。第 8 行“print("\(p!), \(p!.rawValue)")”输出 p 表示的枚举量和 p 表示的枚举量的原始值，得到“positive, positive”。

第 9～12 行定义枚举类型 DNACode，为整数类型。第 13 行“var dna＝DNACode(rawValue:3)”调用枚举类型 DNACode 的初始化器定义枚举变量 dna，此时的 dna 为可选枚举类型。第 14 行“print("\(dna!),\(dna!.rawValue)")”输出枚举变量 dna 的枚举量和它的原始值，得到“T, 3”。

第 15～21 行定义了枚举类型 Season。第 22 行“var s＝Season.summer”定义枚举变量 s，赋值为 Season.summer。第 23 行“print("\(s),\(s.rawValue)")”输出 s 的枚举量和它的原始值，得到“summer, 2”。

第 24～28 行定义了枚举类型 Constant。第 29 行“var c1＝Constant(rawValue:3.14159)”借助 Constant 的初始化器为变量 c1 赋值；第 30 行“print("\(c1!),\(c1!.rawValue)")”输出 c1 的枚举量和它的原始值，得到“pi, 3.14159”。第 31 行“var c2＝Constant.pi”定义变量 c2，赋值为枚举量 Constant.pi；第 32 行“print("\(c2),\(c2.rawValue)")”输出 c2 的枚举量和它的原始值，得到“pi, 3.14159”。第 33～36 行为一个 if 结构，第 33 行“if let c3＝Constant(rawValue:2.71828)”使用可选绑定方法将 Constant(rawValue:2.71828)的值赋给常量 c3，如果 c3 不为 nil，则执行第 35 行“print("\(c3.rawValue)")”，输出 c3 的原始值，得到“2.71828”。

5.1.2　枚举量关联值

枚举类型的枚举量可以关联值，其语法为

```
case 枚举量(值的类型)
```

当有多个值时，需要为每个值指定类型，例如：

```
enum  Computer
{
case CPU(Int, Int)
case memory(String)
case display(Int, String)
}
```

上述代码为一个枚举类型 Computer，其中，枚举量 CPU 关联了两个整型值；memory 关联了一个字符串；display 关联了一个整型值和一个字符串。

此外，枚举类型中可以包括“方法”，所谓的方法是指包含在枚举类型中的函数。需要注意的是，枚举类型为值类型，枚举类型中的方法要改变枚举本身的枚举量时，需要使用关键字 mutating 修改方法。在枚举类型的方法中，使用 self 指代枚举类型定义的变量或常量(一般

称为实例)本身。

枚举类型定义的变量仅能保存一个枚举值,借助枚举类型的这一特点和枚举量关联值可以实现联合体,即多种类型的量共用同一个存储空间,但是任一时刻只能存储一种类型。


视频讲解

程序段 5-3 介绍了枚举类型用作联合体的方法,同时,介绍了枚举类型中的方法。

程序段 5-3 枚举类型实现联合体和枚举类型方法实例

```
import Foundation

enum ASCIICode
{
    case ascii(Int)
    case code(Character)
    mutating func change()
    {
        switch self
        {
        case .ascii(let t):
            self = .code(Character(UnicodeScalar(t)!))
        case .code(let c):
            self = .ascii(Int(c.asciiValue!))
        }
    }
}

var a1 = ASCIICode.ascii(65)
switch a1
{
case .ascii(let t):
    print("\(t)")
case .code(let c):
    print("\(c)")
}

var a2 = ASCIICode.ascii(65)
a2.change()
switch a2
{
case .ascii(let t):
    print("\(t)")
case .code(let c):
    print("\(c)")
}

var a3 = ASCIICode.code("C")
a3.change()
switch a3
{
case .ascii(let t):
    print("\(t)")
case .code(let c):
    print("\(c)")
}
```

在程序段 5-3 中，第 3～17 行定义了枚举类型 ASCIICode，包含两个枚举量，即第 5 行的“case ascii(Int)”，关联一个整型值；第 6 行的“case code(Character)”，关联一个字符值。还包含一个方法，即第 7～16 行的 change 方法“mutating func change()”。在方法 change 内部，第 9～15 行为一个 switch 结构，第 9 行“switch self”根据枚举类型定义的实例做出选择，如果匹配第 11 行“case . ascii(let t):”（这里根据上下文可省略枚举类型名，完整的形式为“ASCIICode. ascii(let t)”），则执行第 12 行“self=. code(Character(UnicodeScalar(t)!))”，将整数值作为 ASCII 值转换为字符；如果匹配第 13 行“case . code(let c):”，则执行第 14 行“self=. ascii(Int(c. asciiValue!))”，将字符转换为对应的 ASCII 整数值。

第 19 行“var a1=ASCIICode. ascii(65)”定义枚举变量 a1，赋值为枚举量 ascii，关联值为 65。第 20～26 行为一个 switch 结构，第 20 行“switch a1”根据 a1 的值选择执行后续操作，如果 a1 匹配第 22 行“case . ascii(let t):”，则执行第 23 行“print("\(t)")”。由第 19 行可知，a1 匹配第 22 行，这里第 23 行执行得到“65”。

第 28 行“var a2=ASCIICode. ascii(65)”定义枚举变量 a2，赋值为枚举量 ascii，关联值为 65。第 29 行“a2. change()”调用 change 方法将变量 a2 的关联整型值转换为字符。第 30～36 行为一个 switch 结构，这里第 30 行“switch a2”根据 a2 的值进行匹配，将与第 34 行“case . code(let c):”匹配成功，执行第 35 行“print("\(c)")”输出“A”。

第 38 行“var a3=ASCIICode. code("C")”定义枚举变量 a3，赋值为枚举量 code，关联值为字符 C。第 39 行“a3. change()”调用 change 方法将变量 a3 的关联字符值转换为其 ASCII 码整型值。第 40～46 行为一个 switch 结构，第 40 行“switch a3”根据 a3 的值选择支路执行，这里 a3 与第 42 行“case . ascii(let t):”相匹配，第 43 行“print("\(t)")”输出“67”。

5.1.3　遍历枚举量

将枚举类型定义为 CaseIterable，使用枚举类型的属性 allCases 可将枚举类型转换为数组，此时可以遍历枚举类型中的各个枚举量。

程序段 5-4 介绍了遍历枚举类型中枚举量的方法。


视频讲解

程序段 5-4　遍历枚举量实例

```
1    import Foundation
2
3    enum Vehicle : CaseIterable
4    {
5        case plane, train, bus, ship
6    }
7    for e in Vehicle.allCases
8    {
9        print(e)
10   }
```

在程序段 5-4 中，第 3～6 行定义枚举类型 Vehicle，使用 CaseIterable 类型将其转换可数特性。第 7～10 行为一个 for-in 结构，第 7 行“for e in Vehicle. allCases”中“Vehicle. allCases”为由枚举类型 Vehicle 中各枚举量组成的数组，这里遍历这个数组，输出各个枚举量，执行程序将依次输出 plane、train、bus、ship。

枚举类型定义了包含原始值的情况下可以输出每个枚举量的原始值，如程序段 5-5 所示。


视频讲解

程序段 5-5 遍历枚举量的原始值实例

```
1    import Foundation
2
3    enum Vehicle : Int, CaseIterable
4    {
5        case plane = 1, train, bus, ship
6    }
7    for e in Vehicle.allCases
8    {
9        print(e.rawValue)
10   }
```

对比程序段 5-4 和程序段 5-5 可知，在程序段 5-5 中第 3 行定义枚举变量时指定了类型为“Int, CaseIterable”，表示为枚举量指定整型原始值。在第 5 行中指定 plane 的原始值为 1，则 train、bus 和 ship 的原始值依次为 2、3 和 4。在第 7～10 行的 for-in 结构中，第 9 行“print(e.rawValue)”输出每个枚举量的原始值，将依次输出 1、2、3、4。

5.1.4 递归枚举

Swift 语言支持递归枚举结构，即在一个枚举类型中可以使用其定义了的实例作为关联值，此时需要用 indirect 关键字修饰该枚举类型，或者用 indirect 关键字修饰使用了自身实例作为关联值的枚举情况，例如：

```
indirect enum Fibonacci
{
case number(Int)
case next(Fibonacci, Fibonacci)
}
```

或

```
enum Fibonacci
{
case number(Int)
indirect case next(Fibonacci, Fibonacci)
}
```

上述定义了一个枚举类型 Fibonacci，其中包含了将枚举类型 Fibonacci 的实例作为关联值的枚举量，这类枚举类型称为递归枚举。


视频讲解

程序段 5-6 展示了递归枚举的用法。

程序 5-6 递归枚举类型用法实例

```
1    import Foundation
2
3    enum Fibonacci
4    {
5    case number(Int)
6    indirect case next(Fibonacci, Fibonacci)
7    }
8    var fab : [Int] = [1,1]
9    for i in 2...10
10   {
11       var k1=Fibonacci.number(fab[i-2])
12       var k2=Fibonacci.number(fab[i-1])
```

```
13      var k3=Fibonacci.next(k1, k2)
14      fab.append(calc(next: k3))
15  }
16  print(fab)
17  func calc(next: Fibonacci)->Int
18  {
19      switch next
20      {
21      case.number(let v):
22          return v
23      case let.next(f1,f2):
24          return calc(next:f1) + calc(next:f2)
25      }
26  }
```

在程序段 5-6 中，第 3～7 行定义了递归枚举类型 Fibonacci，具有两个枚举量，其中，number 关联了一个整型值；next 关联了两个 Fibonacci 类型值。

第 8 行“var fab:[Int]=[1,1]”定义整型数组 fab，赋初始数组为“[1，1]”。第 9～15 行为一个 for-in 结构，循环变量 i 从 2 递增到 10，对于每个 i，循环执行第 11～14 行，第 11 行“var k1=Fibonacci. number(fab[i-2])”定义枚举变量 k1，赋值为枚举量 number，关联值为 fab[i-2]；第 12 行“var k2=Fibonacci. number(fab[i-1])”定义枚举变量 k2，赋值为枚举量 number，关联值为 fab[i-1]；第 13 行“var k3=Fibonacci. next(k1，k2)”定义枚举变量 k3，赋值为枚举量 next，关联值为 k1 和 k2。第 14 行“fab. append(calc(next:k3))”先调用 calc 函数，根据其参数 next 计算下一个 Fibonacci 数，然后，调用 append 方法将该 Fibonacci 数添加到数组 fab 中。

第 16 行“print(fab)”输出数组 fab，得到“[1，1，2，3，5，8，13，21，34，55，89]”。

第 17～26 行为 calc 函数，具有一个 Fibonacci 枚举类型的参数，返回整型值。第 19～25 行为一个 switch 结构，第 19 行 switch next 根据 next 的值选择执行后续程序，当 next 匹配了枚举量 number 时，执行第 22 行 return v 返回 number 关联的整数值；当 next 匹配了枚举量 next 时，执行第 24 行“return calc(next:f1)＋calc(next:f2)”递归执行 calc 函数返回 f1 和 f2 的和。

在第 21 行和第 23 行使用枚举量 number 和 next 时，均无须指出其所在的枚举类型 Fibonacci，因为从上下文环境中(即从第 19 行的 next 上)可推断其枚举类型。

程序段 5-6 中的 calc 函数可以移到枚举类型 Fibonacci 中，作为它的一个方法，这样程序更加简洁易读。在程序段 5-7 中作了这种调整。

程序段 5-7　带有内部方法的递归枚举类型用法实例

```
1   import Foundation
2
3   enum Fibonacci
4   {
5       case number(Int)
6       indirect case next(Fibonacci, Fibonacci)
7       func calc(next: Fibonacci)->Int
8       {
9           switch next
10          {
11          case.number(let v):
```

```
            return v
        case let.next(f1,f2):
            return calc(next:f1) + calc(next:f2)
        }
    }
}
var fab : [Int] = [1,1]
for i in 2...10
{
    var k1=Fibonacci.number(fab[i-2])
    var k2=Fibonacci.number(fab[i-1])
    var k3=Fibonacci.next(k1, k2)
    fab.append(k3.calc(next: k3))
}
print(fab)
```

对比程序 5-6 和程序段 5-7 可知，在程序段 5-7 中，函数 calc 移至枚举类型 Fibonacci 内部，作为它的一个方法，注意，该方法没有修改 Fibonacci 类型的内部枚举量，所以不使用 mutating 关键字修改该方法。

注意，在第 24 行“fab.append(k3.calc(next:k3))”调用 calc 函数时需要使用 Fibonacci 类型定义的实例，这里使用了实例 k3。

5.1.5 枚举初始化器

枚举类型属于值类型，在定义一个枚举类型的变量（称为实例）时，枚举类型具有一个默认的初始化器。典型情况为带有原始值的枚举类型，其默认的初始化器为“枚举类型名(rawValue：枚举值)”。除此之外，还有两种情况：①枚举类型可自定义初始化器，初始化器方法名为 init，可以带有参数，但没有返回值；②枚举类型可自定义带容错能力的初始化器，这类初始化器的方法名为“init?”，如果初始化器执行不成功，将把空值 nil 赋给实例。

注意：①一旦自定义了初始化器，枚举类型默认的初始化器将不能再使用，因为自定义的初始化器“覆盖”了默认的初始化器；②初始化器方法 init 或“init?”不能被直接调用。


视频讲解

下面借助程序段 5-8 介绍枚举类型的初始化器。

程序段 5-8 枚举初始化器实例

```
import Foundation

enum Score : Int
{
    case perfect = 100, good = 80, bad = 60
}
enum Weather
{
    case sunny, cloudy, rainy, snowy, windy
    init(state v : Character)
    {
        switch v
        {
        case "A":
            self = .sunny
        case "B":
            self = .cloudy
```

```
18          case "C":
19              self = .rainy
20          case "D":
21              self = .snowy
22          case "E":
23              self = .windy
24          case _:
25              self = .sunny
26          }
27      }
28      init?(condition w : Character)
29      {
30          switch w
31          {
32          case "A":
33              self = .sunny
34          case "B":
35              self = .cloudy
36          case "C":
37              self = .rainy
38          case "D":
39              self = .snowy
40          case "E":
41              self = .windy
42          case _:
43              return nil
44          }
45      }
46  }
47  if let s = Score(rawValue: 60)
48  {
49      print(s)
50  }
51  let w1 = Weather(state: "B")
52  print(w1)
53  if let w2 = Weather(condition: "C")
54  {
55      print(w2)
56  }
```

在程序段 5-8 中，第 3～6 行定义了一个带原始值的枚举类型 Score，其原始值类型为整型。在第 5 行“case perfect＝100，good＝80，bad＝60”中为各个枚举量指定了原始值。

第 7～46 行定义了枚举类型 Weather，第 9 行定义了其枚举量“case sunny，cloudy，rainy，snowy，windy”。第 10～27 行定义了其初始化器“init(state v:Character)”，具有一个字符型的参数。第 12～26 行为一个 switch 结构，其中，self 表示枚举类型创建的实例本身，如果第 12 行“switch v”的 v 为字符 A，则第 14 行“case "A":”匹配成功，第 15 行“self＝.sunny”表示将枚举量 sunny 赋给新创建的实例。

第 28～45 行为带有容错能力的初始化器“init?（condition w:Character)”，带有一字符型的参数。与第 10～27 行的初始化器不同的是，这里带容错能力的初始化当输入字符不合法时，可以返回空值 nil，如第 42～43 行所示。

第 47～50 行为一个 if 结构，第 47 行“if let s＝Score(rawValue:60)”调用 Score 类型的默认初始化器，将原始值 60 对应的枚举量赋给常量 s，由于默认初始化器返回可选类型，故这里

使用了可选绑定技术，如果 s 不为空值 nil，则执行第 49 行“print(s)”，输出 s，得到“bad”。

第 51 行“let w1＝Weather(state:"B")”将自动调用 Weather 枚举类型的初始化器 init，将参数“B”对应的枚举量 cloudy 赋给 w1。第 52 行“print(w1)”将输出“cloudy”。

第 53～56 行为一个 if 结构，使用可选绑定技术判断第 53 行“if let w2＝Weather(condition:"C")”中 w2 是否为空值 nil，这里自动调用 Weather 枚举类型的带容错能力的初始化器“init?”，如果 w2 不为空值 nil，则执行第 55 行“print(w2)”，输出 w2 的值。此处，第 53 行调用初始化器“init?”将字符 C 对应的枚举量赋给 w2，因此，第 55 行执行得到“rainy”。

5.2 结构体

Swift 语言开发者建议程序设计者多用结构体开发应用程序。在 Swift 语言中，结构体具有了很多类的特性(除类的与继承相关的特性外)，具有属性和方法，且为值类型。所谓的属性是指结构体中的变量或常量，所谓的方法是指结构体中的函数。在结构体中使用属性和方法是因为：①匹别于结构体外部定义的变量和常量；②从面向对象程序设计的角度，结构体对应着现实世界的一个客观物体，描述这个物体的性质需要用到它的属性和方法；③结构体定义的常量或变量称为实例，实例内部的变量或常量成员称为属性，实例内部的函数成员称为方法，这种称谓比常规含义的变量、常量和函数更具有意义。

定义结构体需使用关键字 struct，结构体名建议使用“大骆驼命名法”，即首字母大写且其中的完整英文单词的首字母也大写。结构体定义的实例名建议使用“小骆驼命名法”，即首字母小写且其中的完整英文单词的首字母大写。结构体是一种类型，其中定义的属性和方法的位置可任意放置，可以先定义属性，再定义方法；也可以先定义方法，再定义属性。典型的结构体定义形式如下：

```
1    struct Circle
2    {
3      var radius = 1.0
4      func area() -> Double
5        {
6            return 3.14*radius*radius
7        }
8    }
```

上述代码定义了一个结构体类型，名称为 Circle，具有一个属性 radius 和一个方法 area。一般地，在结构体中定义属性时需要为属性赋初始值，或者使用初始化器为结构体的各个属性赋初始值。属性分为两种，一种为存储类型的属性，如上述的 radius 属性；另一种为计算类型的属性，这种属性的行为类似于方法，但是形式是属性，将在 5.2.2 节讨论。结构体为值类型，如果结构体中定义的方法修改了其中的属性，需要使用关键字 mutating 修饰该方法，这种方法将为属性创建临时存储空间，待方法执行完后，将临时存储空间的值赋回给需要修改的属性。

结构体定义的变量或常量称为实例，实例调用属性和方法时使用“实例.属性”或“实例.方法(实际参数列表)”的形式。除了属于实例的属性和方法外，还可以定义属于结构体的属性和方法，借助 static 关键字定义，使用形式“结构体名.属性”或“结构体名.方法(实际参数列表)”访问这类属性和方法。

5.2.1　结构体用法

结构体的基本用法包括以下几点：

(1) 借助关键字 struct 定义结构体，结构体名建议首字母大写。

(2) 使用与定义变量和常量相同的方法在结构体内部定义属性，这类属性称为存储属性，也称实例属性，即通过结构体定义的实例才能访问的属性。

(3) 使用与定义函数相同的方法在结构体中定义方法，这种方法称为实例方法，表示通过结构体定义的实例访问的方法。如果方法中需要修改结构体的属性，则在定义方法时要使用关键字 mutating。

(4) 使用关键字 static 定义的属性，称为结构体属性，或称静态属性，表示通过结构体名才能访问的属性。

(5) 使用关键字 static 定义的方法，称为结构体方法，或称静态方法，只能通过结构体名才能访问结构体方法。结构体方法只能访问其他的结构体方法和结构体属性，而不能访问实例方法，因为实例方法只有创建了实例后才存在。

(6) 结构体具有默认的初始化器，称为面向属性的初始化器，借助该初始化器，可初始化结构体中的全部存储属性。

程序段 5-9 展示了结构体的基本用法。


视频讲解

程序段 5-9　结构体基本用法实例

```
1    import Foundation
2
3    struct Point
4    {
5        var x,y : Double
6    }
7    struct Circle
8    {
9        static var name = ""
10       static func getName()->String
11       {
12           return name
13       }
14       var radius = 1.0
15       var center = Point(x:0,y:0)
16       func area()->Double
17       {
18           return 3.14 *radius *radius
19       }
20       mutating func moveTo(point: Point)
21       {
22           center.x = point.x
23           center.y = point.y
24       }
25   }
26   Circle.name = "A Circle."
27   var str = Circle.getName()
28   print(str)
29   var cir = Circle(radius: 3.0,center: Point(x: 4.0, y: 5.0))
30   print("Circle at (\(cir.center.x),\(cir.center.y)) with ",terminator: "")
```

```
31  print("area = " + String(format: "%5.2f", cir.area()))
32  cir.radius = 5.0
33  cir.center = Point(x: 12.0, y: 8.0)
34  print("Circle at (\(cir.center.x),\(cir.center.y)) with ",terminator: "")
35  print("area = " + String(format: "%5.2f", cir.area()))
36  let p = Point(x: 2.0, y: 2.0)
37  cir.moveTo(point: p)
38  print("Circle at (\(cir.center.x),\(cir.center.y)) with ",terminator: "")
39  print("area = " + String(format: "%5.2f", cir.area()))
```

程序段 5-9 的执行结果如图 5-1 所示。

```
A Circle.
Circle at (4.0,5.0) with area = 28.26
Circle at (12.0,8.0) with area = 78.50
Circle at (2.0,2.0) with area = 78.50
Program ended with exit code: 0
All Output
```

图 5-1　程序段 5-9 的执行结果

结合图 5-1 分析程序段 5-9 的代码执行过程。在程序段 5-9 中，第 3～6 行定义了结构体 Point，其代码再次罗列如下：

```
3   struct Point
4   {
5       var x,y : Double
6   }
```

可见，结构体 Point 只具有两个存储属性 x 和 y，因此 Point 具有默认的面向属性的初始化器，其调用格式为“Point(x：传递给 x 的值，y：传递给 y 的值)”。例如，在第 15 行“var center＝Point(x:0,y:0)”中定义变量 center，使用 Point 的初始化器将 center 设为 Point 结构体类型的实例。同样地，在第 36 行“let p＝Point(x:2.0，y:2.0)”定义常量 p，使用 Point 的初始化器将 p 设为 Point 结构体类型的实例。在面向属性的默认初始化器中，属性名自动被当作参数。

第 7～25 行定义了结构体 Circle。第 9 行“static var name＝""”定义了结构体属性或静态属性 name，该属性借助结构体名访问。第 10～13 行定义了结构体方法或静态方法 getName，该方法只能借助结构体名访问。注意，结构体的静态方法只能访问其静态属性。

第 14 行“var radius＝1.0”定义了实例属性 radius，该属性也可称为动态属性，这里的 radius 为存储属性。第 15 行“var center＝Point(x:0,y:0)”定义了实例属性 center，赋值为“Point(x:0,y:0)”，将调用 Point 结构体默认的面向属性的初始化器将 center 设为坐标点(0,0)。

第 16～19 行定义了实例方法 area，返回圆的面积。第 20～24 行定义了实例方法 moveTo，由于在 moveTo 方法中修改了结构体 Circle 的存储属性 center，故使用了 mutating 关键字。moveTo 方法将圆心移动到参数 point 指示的坐标点处。

第 26 行“Circle.name＝"A Circle."”将结构体 Circle 的静态属性 name 赋为“A Circle.”。第 27 行“var str＝Circle.getName()”定义变量 str，调用结构体 Circle 的静态方法 getName 将获取的结构体静态属性 name 的值赋给 str。第 28 行“print(str)”输出“ACircle.”。

第 29 行“var cir＝Circle(radius:3.0,center:Point(x:4.0，y:5.0))”定义实例 cir，使用了结构体 Circle 的面向属性的默认初始化器以及结构体 Point 的面向属性的默认初始化器，这

里表示定义了一个圆心在(4.0，5.0)半径为3.0的圆。第30行“print("Circle at (\(cir.center.x),\(cir.center.y)) with ",terminator:"")”和第31行“print("area＝" ＋ String(format:"%5.2f", cir.area()))”输出“Circle at (4.0,5.0) with area＝28.26”，这里使用“cir.center.x”访问圆心的x坐标，使用cir.center.y访问圆心的y坐标，使用cir.area()获取圆的面积。

第32行“cir.radius＝5.0”将圆的半径设为5.0，这里采用“实例.存储属性”的方式向存储属性赋值。第33行“cir.center＝Point(x:12.0，y:8.0)”将圆心设为坐标点“(12.0，8.0)”。第34～35行输出“Circle at (12.0,8.0) with area＝78.50”。

第36行“let p＝Point(x:2.0，y:2.0)”定义结构体Point的常量实例p。第37行“cir.moveTo(point:p)”调用实例cir的moveTo方法将圆心移动到p处。第38～39行输出“Circle at (2.0,2.0) with area＝78.50”。

5.2.2　存储属性与计算属性

在结构体中，属性有以下三种类型：

(1) 使用static定义的静态属性，称为结构体属性，这类属性采用“结构体名.结构体属性名”的方式访问，如程序段5-9的第26行所示；

(2) 存储属性，又可称为动态属性，在结构体中定义的常量和变量均属于存储属性，存储属性为实例属性，采用“实例名.存储属性名”的方式访问；

(3) 计算属性，也属于实例属性，采用“实例名.计算属性名”的方式访问，但是计算属性本身不保存属性值，与存储属性不同之处在于在定义计算属性后添加一对“{ }”，在其中添加get方法和set方法(至少要添加get方法)，依次用于获取存储属性值或设置存储属性值。

对于存储属性：①当使用private修饰时将变成私有存储属性，这类私有存储属性只能借助计算属性间接访问，不能借助实例访问，从而实现了数据“封装”特性；②当使用lazy修饰时将变成惰性存储属性，这类惰性存储属性必须为变量类型，当它被首次使用时才初始化，即如果惰性存储属性只定义了但没有使用，则该属性不会占用存储空间。

程序段5-10详细介绍了存储属性、计算属性和私有存储属性的用法。


视频讲解

程序段5-10　结构体的属性用法实例

```
1    import Foundation
2
3    struct Point
4    {
5        var x,y : Double
6    }
7    struct Circle
8    {
9        var radius = 1.0
10       var r : Double
11       {
12           get
13           {
14               return radius
15           }
16           set
17           {
```

```
            radius = newValue
        }
    }
    private var center = Point(x: 0, y: 0)
    var c : Point
    {
        return center
    }
    mutating func moveTo(point: Point)
    {
        center = point
    }
    func area()->Double
    {
        return 3.14*r*r
    }
}

var circle = Circle()
circle.r = 5.0
print("radius = \(circle.r)")
circle.moveTo(point: Point(x: 3.0, y: 4.0))
print("Circle at (\(circle.c.x),\(circle.c.y))",terminator: " ")
print("with area: \(String(format:"%5.2f",circle.area()))")
```

程序段 5-10 的执行结果如图 5-2 所示。

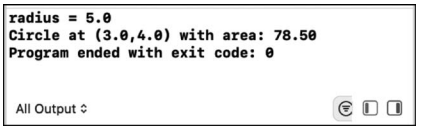

图 5-2　程序段 5-10 的执行结果

下面结合图 5-2 介绍程序段 5-10。在程序段 5-10 中，第 3～6 行定义了结构体 Point，具有两个存储属性 x 和 y。

第 7～34 行定义结构体 Circle。其中第 9 行“var radius＝1.0”定义存储属性 radius，并赋初值为 1.0。一般地，在结构体内定义存储属性时需给它们赋初值。第 10～20 行定义了计算属性 r，将其代码再次罗列如下：

```
var r : Double
{
    get
    {
        return radius
    }
    set
    {
        radius = newValue
    }
}
```

计算属性 r 更像是一个方法，具有“{ }”括起来的可执行代码。其中，第 12～15 行为 get 方法，这是一种程序员约定的名称，因为这种方法形如“get{ }”，get 方法用于返回结构体中某个设定的存储属性的值，这里第 14 行返回存储属性 radius 的值，即读取 r 相当于读取 radius。

第 16～19 行为 set 方法，这也是一种程序员约定的名称，因为这种方法形式“set{ }”，set 方法用于向某个选定的存储属性赋值，这里第 18 行向 radius 赋值，其中，newValue 为关键字，自动保存赋给 r 的值，因此，向 r 赋值(即写 r)就是向 radius 赋值(即写 radius)。注意：①如果计算属性只有 get 方法，则去掉 set 部分，同时，get 和“{ }”也可省略，如第 22～25 行的计算属性 c，此时的计算属性为只读属性；②计算属性可以只有 get 方法，称为只读属性；可以同时有 get 方法和 set 方法，称为可读可写属性；但是不能只有 set 方法，即没有只写属性；③在 set 方法中可以使用 newValue 关键字，用于表示赋给计算属性的值，也可以自定义该量，例如第 16～19 行的代码可以替换为下述代码：

```
16   set(val)
17   {
18       radius = val
19   }
```

这里使用了自定义的 val 变量，此处无须指定数据类型，而是根据计算属性 r 的类型自动推断数据类型。

第 21 行“private var center＝Point(x:0, y:0)”定义私有存储属性 center，这类属性只能借助计算属性访问，同时这类属性可被结构内的其余属性和方法使用，但不能被结构体外部的方法调用，结构体定义的实例也不能访问这类属性。

第 22～25 行定义一个计算属性 c，其代码再次罗列如下：

```
22   var c : Point
23   {
24       return center
25   }
```

这里计算属性 c 为一个只读属性，省略了 get 和“{ }”，返回 center 的值，即读取 c 相当于读取 center。

第 26～29 行定义了函数 moveTo，使用了 mutating 关键字，因为该函数将修改私有存储属性 center 的值。第 28 行“center＝point”将参数 point 赋给私有存储属性 center。

第 30～33 行定义了函数 area，返回值为 Double 类型，函数返回圆的面积。第 32 行“return 3.14 * r * r”使用了计算属性 r，因为计算属性 r 为可读可写属性，这里读取 r 相当于读取 radius。

第 36 行“var circle＝Circle()”定义结构体实例 circle。第 37 行“circle.r＝5.0”将 5.0 赋给实例 circle 的计算属性 r，计算属性 r 并不保存值，其内部将 5.0 赋给存储属性 radius。第 38 行“print("radius＝\(circle.r)")”输出存储属性 radius 的值，得到“radius＝5.0”。第 39 行“circle.moveTo(point:Point(x:3.0, y:4.0))”调用 moveTo 方法将圆心移动到点(3.0, 4.0)处。第 40 行“print("Circle at (\(circle.c.x),\(circle.c.y))",terminator:" ")”和第 41 行“print("with area:\(String(format:"%5.2f",circle.area()))")”联合输出“Circle at (3.0,4.0) with area:78.50”，其中，第 40 行通过只读的计算属性 c，获取圆心坐标。

5.2.3　结构体初始化器

结构体的初始化器包括两类。

(1) 普通初始化器。

普通初始化器由 init()方法实现，在定义结构体实例时自动调用初始化器。初始化器可

以重载，当定义了初始化器后，结构体默认的面向属性的初始化器将被“覆盖”而不能再使用。

(2) 带容错能力的初始化器。

普通初始化器可以带参数，也可编写程序代码处理参数的有效性，但是普通初始化器无返回值。带容错能力的初始化器，当参数无效时，返回空值 nil，使用“init? ()”方法实现，借助带容错能力的初始化器生成的结构体实例为可选类型。


视频讲解

程序段 5-11 展示了上述两种结构体初始化器的用法。

程序 5-11 结构体初始化器用法实例

```
1    import Foundation
2
3    struct Point
4    {
5        var x,y : Double
6        init()
7        {
8            x=0
9            y=0
10       }
```

第 6～10 行定义了一个无参数初始化器，将属性 x 和 y 均赋为 0。

```
11     init(x:Double,y:Double)
12     {
13       self.x = x
14       self.y = y
15     }
```

第 11～15 行定义了一个带参数的初始化器，第 13 行“self. x”中的 self 表示结构体定义实例本身，这里“self. x＝x”表示将参数 x 赋给结构体的属性 x，第 14 行表示将参数 y 赋给结构体的属性 y。

```
16   }
17   struct Circle
18   {
19       var radius = 1.0
20       var center = Point()
21       init()
22       {
23           radius = 1.0
24           center = Point()
25       }
```

第 21～25 行定义了一个无参数的初始化器，将属性 radius 赋为 1.0，将属性 center 赋为 Point()，即调用结构体 Point 的无参数初始化器，将 center 赋为点(0,0)。

```
26   init(r:Double)
27   {
28       radius = r
29       center = Point(x:1.0,y:1.0)
30   }
```

第 26～30 行定义了带有一个参数的初始化器，将参数 r 赋给属性 radius，调用 Point 的带参数的初始化器将(1.0,1.0)赋给 center。

```
31   init(r:Double, p:Point)
```

```
32  {
33      radius = r
34      center = p
35  }
```

第31～35行为带有两个参数的初始化器，将参数r赋给属性radius，将参数p赋给属性center。

```
36  init?(radius r:Double,point p:Point)
37  {
38      if r<0
39      {
40          return nil
41      }
42      radius = r
43      center = p
44  }
```

第36～44行为带有容错能力的初始化器，如果参数r小于0，则执行第40行返回空值nil；否则，将参数r赋给属性radius，将参数p赋给属性center。

```
45      mutating func moveTo(point: Point)
46      {
47          center = point
48      }
49      func area()->Double
50      {
51          return 3.14*radius*radius
52      }
53  }
54
55  var circle = Circle(r:3.0,p:Point(x:2.0,y:2.0))
56  print("radius = \(circle.radius)")
57  circle.moveTo(point: Point(x: 5.0, y: 4.0))
58  print("Circle at (\(circle.center.x),\(circle.center.y))",terminator: " ")
59  print("with area: \(String(format:"%5.2f",circle.area()))")
60
61  if let c = Circle(radius: 2.5, point: Point(x: 8.0, y: 6.5))
62  {
63      print("Circle at (\(c.center.x),\(c.center.y))",terminator: " ")
64      print("with area: \(String(format:"%5.2f",c.area()))")
65  }
```

在程序段5-11中，第3～16行定义了结构体Point，在结构体Point中定义了两个普通初始化器。第17～53行定义了结构体Circle，在结构体Circle中定义了三个普通初始化器和一个带容错能力的初始化器。

第55行调用Circle的普通初始化器(第31～35行的初始化器)创建实例circle。第56行输出radius的值，得到“radius=3.0”。第57行将圆心移至点(5.0，4.0)。第58～59行输出“Circle at (5.0,4.0) with area:28.26”。

第61～65行为一个if结构，第61行使用可选绑定技术，调用Circle结构体的带容错能力的初始化器，如果c不为空值nil(此处，c不为空值)，则执行第63～64行，输出“Circle at(8.0，6.5) with area:19.62”。

5.2.4 实例方法与静态方法

结构体中的方法分为以下两种。

(1) 实例方法。

结构体中的方法与普通的函数类似(以 func 关键字开头),主要的区别在于结构体中的方法一般不具有参数,而是直接使用结构体中的属性,而普通的函数往往带有很多参数。结构体中的方法一般不具有参数或带有少量必要参数,是因为结构体中的方法主要为结构体中的属性服务。那些具有通用功能的方法不应作为某个结构体中的方法,而应作为全局意义上的函数。结构体中的方法通过结构体定义的实例访问,故称为实例方法。

(2) 静态方法,也称结构体方法。

定义在结构体中的以 static 开头的方法,称为静态方法或结构体方法。这类方法通过结构体名调用。静态方法只能访问结构体中的静态属性。某些情况下,使用静态方法比实例方法更方便,例如,对于一些通用算法实现的功能,如数学函数运算等,可以作为静态方法集中在一个结构体中,使用结构体名调用。静态方法一般具有参数。

注意:结构体中的实例方法可以调用静态方法,在调用静态方法时必须使用"结构体名.静态方法名"的形式,即使调用同一个结构体内部的静态方法,在调用时也要为静态方法指定结构体名。然而,静态方法不能调用动态方法,因为动态方法只有在创建了结构体实例后才能使用,而静态方法在定义结构体后就可以使用了。类似地,实例方法可以使用静态属性(或结构体属性),但需要为静态属性指定结构体名;然而,静态方法不能使用动态的存储属性和计算属性。

下面的工程 MyCh0512 中的程序文件展示了上述两种方法的用法。工程 MyCh0512 包括两个程序文件,即 mystruct.swift 和 main.swift,分别如程序段 5-12 和程序段 5-13 所示。


视频讲解

程序段 5-12 程序文件 mystruct.swift

```
import Foundation

struct Math
{
    static let pi = 3.14159
    static func gcd(first op1:Int,second op2:Int)->Int
    {
        var a,b :Int
        a=max(op1,op2)
        b=min(op1,op2)
        while b>0
        {
            (a,b) = (b,a % b)
        }
        return a
    }
    static func lcm(first op1:Int, second op2:Int)->Int
    {
        return op1*op2/gcd(first:op1,second:op2)
    }
}

struct Circle
```

```
24  {
25      static var count : Int = 0
26      var r = 1.0
27      init(radius r:Double)
28      {
29          Circle.count += 1
30          self.r = r
31      }
32      func area()->Double
33      {
34          return Math.pi *r *r
35      }
36  }
```

程序段 5-13　程序文件 main.swift


```
1   import Foundation
2
3   print("GCD(120,90) = ",Math.gcd(first: 120, second: 90))
4   print("LCM(120,90) = ",Math.lcm(first: 120, second: 90))
5
6   var c1 = Circle(radius: 3.5)
7   print("We have created \(Circle.count) instance(s).")
8   print("Circle's area = "+String(format:"%5.2f",c1.area()))
9   var c2 = Circle(radius: 3.5)
10  print("We have created \(Circle.count) instance(s).")
11  print("Circle's area = "+String(format:"%5.2f",c2.area()))
```

工程 MyCh0512 的执行结果如图 5-3 所示。

```
GCD(120,90) =  30
LCM(120,90) =  360
We have created 1 instance(s).
Circle's area = 38.48
We have created 2 instance(s).
Circle's area = 38.48
Program ended with exit code: 0

All Output ⇕
```

图 5-3　工程 MyCh0512 的执行结果

下面结合图 5-3 介绍程序段 5-12 和程序段 5-13。

在程序段 5-12 中，第 3～21 行定义了结构体 Math，其中，第 5 行“static let pi＝3.14159”定义了静态属性 pi，表示圆周率。第 6～16 行定义了静态方法 gcd，用于计算两个整数的最大公约数，使用欧几里得算法；第 17～20 行定义了静态方法 lcm，用于计算两个整数的最小公倍数，其中调用了静态方法 gcd。注意，在同一个结构体内部，一个静态方法可以直接调用另一个静态方法，无须指定结构体名。

第 23～36 行定义了结构体 Circle，其中，第 25 行“static var count：Int＝0”定义了静态属性 count，用于统计结构体 Circle 创建实例的次数。第 26 行“var r＝1.0”定义了存储属性 r，赋初值为 1.0。第 27～31 行为初始化器 init，其中，第 29 行“Circle.count ＋＝ 1”表示每调用一次初始化器，count 的值累加 1，由于 count 为静态属性，故其值在工程的生命期内一直有效；第 30 行“self.r＝r”将参数 r 的值赋给存储属性 r。第 32～35 行定义函数 area，返回圆的面积，其中使用静态属性“Math.pi”。

在程序段 5-13 中，第 3 行“print("GCD(120,90)＝",Math.gcd(first：120， second：90))”调用静态方法“Math.gcd”计算 120 和 90 的最大公约数，输出“GCD(120,90)＝30”。第 4 行“print("LCM(120,90)＝",Math.lcm(first：120， second：90))”调用静态方法“Math.lcm”计算 120 和 90 的最小公倍数，输出“LCM(120,90)＝360”。第 6 行“var c1＝Circle(radius：3.5)”定义实例 c1。第 7 行“print("We have created \(Circle.count) instance(s).")”输出“We have created 1 instance(s).”表示已创建了一个结构体 Circle 类型的实例。第 8 行“print("Circle's

area＝"＋String(format:"%5.2f",c1.area()))"调用实例方法 area 输出圆的面积,得到"Circle's area＝38.48"。第 9 行"var c2＝Circle(radius:3.5)"创建了结构体 Circle 的另一个实例 c2。第 10 行"print("We have created \(Circle.count) instance(s).")"输出"We have created 2 instance(s).",表示已创建了 2 个实例。第 11 行"print("Circle's area＝"＋String(format:"%5.2f",c2.area()))"调用实例方法 area 输出新实例 c2 的面积,得到"Circle's area＝38.48"。

5.2.5 结构体索引器

结构体支持自定义索引器方法,索引器也称下标,例如,在访问数组元素时,使用的"数组名[元素下标]"为一种索引器的形式;在访问字典的元素值时,使用的"字典名[键名]"也是一种索引器的形式。在结构体中,自定义索引器使用关键字 subscript,其语法为

```
subscript(参数列表) ->返回值类型
{
    get
    {
        语句组
        return 语句
    }
    set(参数列表)
    {
        语句组
    }
}
```

在上述索引器语法中,可只有 get 方法,称为只读索引器;或同时具有 get 方法和 set 方法,称为可读可写索引器;但不能只有 set 方法,即没有只写索引器。如果一个索引器为只读索引器,get 和其相关的"{ }"可以省略。对于 set 方法,如果其参数省略,则使用默认参数 newValue,也可以在 set 的"参数列表"中指定参数,无须指定类型。索引器的"参数列表"可以为空,但是必须有返回值。


视频讲解

程序段 5-14 演示了索引器的用法。

程序段 5-14 索引器用法实例

```
1    import Foundation
2
3    struct Matrix
4    {
5        var row : Int
6        var col : Int
7        private var val : [Int]
8        init(row:Int, col:Int)
9        {
10           self.row = row
11           self.col = col
12           val = Array(repeating: 0, count: row *col)
13       }
14       subscript(row:Int,col:Int)->Int
15       {
16           get
17           {
```

```
18              return val[row * self.col+col]
19          }
20          set(v)
21          {
22              val[row * self.col+col] = v
23          }
24      }
25  }
26  var m = Matrix(row: 3, col: 4)
27  for i in 0...3-1
28  {
29      for j in 0...4-1
30      {
31          m[i,j] = i * 4+j+1
32      }
33  }
34  for i in 0...3-1
35  {
36      for j in 0...4-1
37      {
38          print(String(format: "%6d", m[i,j]), terminator: "  ")
39      }
40      print()
41  }
```

在程序段 5-14 中，第 3～25 行定义了一个结构体 Matrix。第 5 行“var row:Int”定义存储属性 row，用于保存矩阵的行数；第 6 行“var col:Int”定义存储属性 col，用于保存矩阵的列数；第 7 行“private var val:[Int]”定义私有属性 val，为一个整型数组。第 8～13 行为初始化器 init，具有两个参数 row 和 col，表示矩阵的行数和列数；在初始化器中，将参数 row 赋给属性 row，将参数 col 赋给属性 col，生成一个长度为“row * col”的全 0 数组赋给私有属性 val。

第 14～24 行为索引器，具有两个参数 row 和 col，指定矩阵的行和列，返回整型值。在第 16～19 行的 get 方法中，第 18 行“return val[row * self. col＋col]”返回数组的第“row * self. col＋col”号元素，相当于返回矩阵的第 row 行第 col 列的元素。在第 20～23 行的 set 方法中，将赋给索引器的值 v 赋给“val[row * self. col＋col]”，相当于赋给矩阵的第 row 行第 col 列的元素。

第 26 行“var m＝Matrix(row:3, col:4)”定义了一个 Matrix 实例 m，初始值为一个长度为 12 的全 0 数组。尽管实例 m 本身只包含了一个长度为 12 的数组，但是通过它的索引器，可使它的元素操作表现为读写一个 3 行 4 列的矩阵。

第 27～33 行为嵌套的 for-in 结构，执行第 31 行“m[i,j]＝i * 4＋j＋1”，向矩阵 m 的第 i 行第 j 列写入值“i * 4＋j＋1”。

第 34～41 行为嵌套的 for-in 结构，执行第 38 行“print(String(format:"%6d", m[i,j]), terminator:" ")”输出 m[i,j]的值，对于每个 i 还将执行第 40 行“print()”输出一个回车换行符，最终的输出结果如图 5-4 所示。

除了程序段 5-14 所示的使用整数值的索引器外，索引器的参数可以为任意基本类型和集类型（例如数组和字典等），甚至可为空参数。程序段 5-15 中使用了双精度浮点型和空类型作为索引器的参

```
     1      2      3      4
     5      6      7      8
     9     10     11     12
Program ended with exit code: 0

All Output ⇕
```

图 5-4　程序段 5-14 的输出结果


视频讲解

数，进一步说明索引器的用法。

程序段 5-15 参数为双精度浮点型和空类型的索引器用法实例

```
1    import Foundation
2
3    struct Circle
4    {
5        var r : Double = 1.0
6        subscript(radius : Double) -> Double
7        {
8            get
9            {
10               return 3.14*radius*radius
11           }
12       }
13       subscript()->Double
14       {
15           get
16           {
17               return 2*3.14*r
18           }
19           set
20           {
21               r = newValue
22           }
23       }
24   }
25   var c = Circle(r: 3.0)
26   print("Area =",String(format: "%6.2f", c[10.0]))
27   c[]=20
28   print("Perimeter =",String(format: "%6.2f", c[]))
```

在程序段 5-15 中，第 3～24 行定义了结构体 Circle。第 5 行“var r:Double=1.0”定义存储属性 r，赋初始值为 1.0，表示圆的半径。

第 6～12 行定义了一个只读索引器，具有一个双精度浮点型参数 radius，其中，第 8、9、11 行的内容可以省略，第 10 行“return 3.14 * radius * radius”返回以索引器的参数为半径的圆面积。

第 13～23 行定义了一个空参数的索引器，在第 15～18 行的 get 方法中返回以存储属性 r 为半径的圆周长；在第 19～22 行的 set 方法中，将赋给索引器的值(这里用 newValue 关键字表示)赋给属性 r。

第 25 行“var c=Circle(r:3.0)”定义结构体 Circle 的实例 c，将其属性 r 设为 3.0。第 26 行“print("Area =",String(format:"%6.2f", c[10.0]))”调用索引器 c[10.0]输出半径为 10.0 的圆面积，得到“Area=314.00”。第 27 行“c[]=20”向参数为空的索引器赋值，即将实例 c 的属性 r 赋为 20。第 28 行“print("Perimeter =",String(format:"%6.2f", c[]))”调用参数为空的索引器 c[]得到圆的周长，输出“Perimeter=125.60”。

5.3 本章小结

在 Swift 语言中，枚举类型和结构体类型都是值类型，并且结构体类型是程序设计常用的自定义类型。在类和结构体之间选择时，Swift 语言设计者鼓励使用结构体类型。本章详细

介绍了枚举类型和结构体类型的定义与用法，讨论了这两种类型的初始化器，详细阐述了结构体的属性和方法，介绍了结构体的索引器用法。在 Swift 语言中，面向结构体的程序设计，已经可以体现面向对象编程的特性。结构体类型除了为值类型外，它的其他特性，如属性、方法、初始化器、索引器等，也体现在类类型上，但是类为引用类型。第 6 章将深入介绍类与其实例。

习题

1. 简述 Swift 语言枚举类型中枚举量的表示方法。

2. 简述 Swift 语言中结构体的定义方法。

3. 给定一个日期，例如 2023 年 12 月 28 日，计算该日期对应的星期，并输出该星期。

4. 给定两个日期，例如 2011 年 5 月 12 日和 2023 年 10 月 10 日，计算这两个日期间的天数。

5. (实践题型)创建一个表示学生信息的结构体，具有属性：姓名、学号、出生日期、性别等，具有输出学生信息的方法 display。编写一个应用，定义上述结构体数组，统计班上同学的出生日期的分布率(按月份统计，即每月出生人数除以总人数的比例)，输出生日相同的同学。

6. 在第 5 题基础上，定义一个班级结构体，将全部学生作为班级结构体的属性，将学号作为索引器，通过学号可访问班级中对应的学生，并输出该学生的信息。