# 第三章 流程控制

程序中的语句通常是顺序执行，即从上至下，依次执行，直至结束。顺序结构通常无法满足复杂的现实需求，所以语言为开发者提供了流程控制语句，利用流程控制语句，可以让计算机重复执行某段代码块，或是让程序跳转执行不同的分支等等。在Swift程序中，主要包含循环语句，条件语句和控制转移语句。

## 3.1循环语句

### 3.1.1 for-in

Swift中提供两种for循环，一种是和其他语言类似的传统for循环，也就是for-condition-increment循环，另一种是for-in循环，常用于遍历一个集合里的所有元素。传统的for循环在Swift3.0中被移除了，所以这里介绍for-in循环。swift中for-in循环的基本语法如下：

for 索引in 集合类型{

    循环体

}

实例8.1 打印区间0...5之间的数字。

for index in 0...5{

    print("当前index为\(index)")

}

程序输出：

当前index为0

当前index为1

当前index为2

当前index为3

当前index为4

当前index为5

实例8.1中，index是每次循环遍历开始时自动被赋值的常量。起初，index被赋值为0...5这个区间中的第一个元素0，并且执行打印语句，接下来index被赋值为区间中的下一个元素也就是1，再次打印，以此类推，直至遍历完整个区间。当然在这种情况下，index不需要使用let关键字进行声明，在循环体内，不可以修改index的值。

如果并不需要index这个索引，比如用for-in循环来计数，就可以使用\_代替index。

实例8.2 \_代替index。

for \_ in 0..<3{

    print("Swift")

}

程序输出：

Swift

Swift

Swift

显然打印3次Swift并不需要index，所以用\_代替，这是一种更加经济的方式。

for-in循环也常常被用来遍历集合类型，如数组，字典等等。实例8.3 用for-in来遍历数组。

let strs = ["Swift", "iOS", "HDU"]

for str in strs{

    print(str)

}

程序输出：

Swift

iOS

HDU

### 3.1.2 while

while循环会一直运行一段语句直到条件变成假（false）。while循环非常适合使用在第一次迭代前，迭代次数未知的情况下。

Swift提供两种while循环的形式：

while循环，在每次执行循环体之前判断条件是否符合。

repeat-while循环，在每次执行循环体之后判断条件是否符合。

* while

while循环基本语法：

while 条件 {

循环体

}

while循环先判断条件再执行语句。如果条件为真（true），会重复运行一段语句，直到条件变为false。实例8.4 使用while打印0...5之间的数字。

var index = 0

while index <= 5 {

    print("当前index为\(index)")

    index += 1

}

程序输出：

当前index为0

当前index为1

当前index为2

当前index为3

当前index为4

当前index为5

index初始时被赋值为0，第一执行循环体之前，判断index的值是否小于等于5，条件成立，执行print语句，随后执行+1，index的值变为1。在下一次执行循环体之前，再次判断，之后执行循环体。直到不再满足判断条件，while循环结束。

* repeat-while

repeat-while循环基本语法：

repeat{

循环体

}while 条件

repeat-while循环不像for-in和while循环在循环体开始执行之前判断条件语句，而是在循环体执行结束之后判断条件是否符合。实例8.5 使用repeat-while打印0...5之间的数字。

var index2 = 0

repeat{

    print("当前index2为\(index2)")

    index2 += 1

}while(index2<=5)

程序输出：

当前index2为0

当前index2为1

当前index2为2

当前index2为3

当前index2为4

当前index2为5

执行完一次循环体之后，如果条件为true，控制流会跳转回上面的repeat，然后重新执行循环体，直到条件变为false为止。

请注意，由于条件表达式出现在循环的尾部，所以循环体在条件被检测之前至少被执行一次。实例8.6。

var index3 = 0

repeat {

    print("当前index3为\(index3)")

} while (index3 < 0)

程序输出：

当前index3为0

尽管index初始值被设置为0，不满足小于0的条件，但是循环体还是会执行一次，打印出0。

在使用while循环的时候，在循环体中要对判定条件进行一定的修改，使判定条件在有限的循环次数之后变为false，不然会出现死循环。

## 3.3 条件语句

条件语句通过判断条件的真假来控制程序执行不同的分支，在条件为真时执行指定的语句，在条件为 false 时执行另外的语句。Swift 提供两种类型的条件语句：if语句和switch语句。通常，当条件较为简单且可能的情况很少时，使用if语句。而switch语句更适用于条件较复杂、有更多排列组合的时候。

### 3.3.1 if-else

if-else基本语法：

if 条件 {

语句

} else {  
 语句

}

if语句最简单的形式就是只包含一个条件，只有条件为true时，才执行相关代码。实例8.7。

var varA:Int = 10

if varA < 20 {

    print("varA 小于 20")

}

程序输出：

varA 小于 20

通过配合else从句，实现二选一执行。也就是当条件为false的时候，执行else语句。实例8.8。

var varB:Int = 30

if varB < 20 {

    print("varB 小于 20")

} else {

    print("varB 大于等于 20")

}

程序输出：

varB 大于等于 20

还可以通过多个if-else配合的方式，来实现更复杂的分支。实例8.9。

var varC:Int = 20

if varC < 20 {

    print("varC 小于 20")

} else if varA == 20{

    print("varC 等于 20")

} else {

    print("varC大于 20")

}

程序输出：

varC 大于等于 20

### 3.3.2 switch

switch被称作多分支选择语句或是开关选择语句，通过匹配的方式来选择程序执行的代码块。

switch基本语法：

switch 表达式 {

case 表达式：

语句

case 表达式:

语句

default:

语句

}

switch语句会尝试把某个值与若干个模式（pattern）进行匹配。根据第一个匹配成功的模式，switch语句会执行相应的代码。当有可能的情况较多时，通常用switch语句替换if语句以简化代码，提高可读性。实例8.10。

let someCharacter : Character = "A"

switch someCharacter {

case "a","e","i","o","u","A","E","I","O","U":

    print("\(someCharacter)是一个元音字母")

default:

    print("\(someCharacter)不是一个元音字母")

}

程序输出：

A是一个元音字母

与if语句类似，每一个 case 都是代码执行的一条分支。switch语句必须是完备的。就是说，每一个可能的值都必须至少有一个 case 分支与之对应，必要的时候需要使用默认（default）分支来涵盖case分支无法匹配的情况，并且这个默认分支必须在switch语句的最后面。switch的case提供了多种功能更加强大的复杂匹配模式。实例8.11。

let time = 11

switch time {

    case let x where x >= 0 && x <= 7:

        print("睡觉时间")

    case 8:

        print("起床时间")

    case 9...12:

        print("上课时间")

    default:

        print("自由活动")

}

程序输出：

上课时间

在swift中，当匹配的case分支中的代码被执行完毕后，会自动终止switch语句，不会继续执行下一个case分支。这也就是说，不需要像其他语言一样，在 case 分支中显式地使用break语句。如果想要实现贯穿的效果，可以使用fallthrough语句。

## 3.4 控制转移语句

控制转移语句可以改变代码的执行顺序，比如提前终止循环结构，也可以认为的控制选择结构的跳转，使代码的执行更加的灵活。本节介绍continue、break、fallthrough和guard。

### 3.4.1 continue

continue语句的作用是告诉一个循环体立刻停止本次循环，重新开始下次循环。结束的仅仅是一次循环，而不是整个循环。实例8.12。

var index = 10

repeat{

    index = index + 1

    if( index == 12 ){ // index 等于 12 时跳过

        continue

    }

    print(index)

}while index < 15

程序输出：

11 13 14 15

当index等于12的时候，print语句被跳过了，而13 14 15还是照常打印，说明循环并未结束。

### 3.4.2 break

break语句的作用是中断当前循环，并结束整个循环。实例8.13。

var index = 10

repeat{

    index = index + 1

    if( index == 12 ){ // index 等于 12 时跳过

        break

    }

    print(index)

}while index < 15

程序输出：

11

将continue改为break，程序就仅仅打印了11是因为当index等于12的时候，整个循环都被终止了，所以12到15都不会被打印。

### 3.4.3 fallthrough

fallthrough 语句可以让 case 之后的语句按顺序继续运行，且不论条件是否满足都会执行。

在大多数语言中的switch结构中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift中，默认是不会执行下去的，switch会终止。如果想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。实例8.14。

如果switch中没有fallthrough语句：

let index = 10

switch index {

case 100  :

    print( "index 的值为 100")

case 10,15  :

    print( "index 的值为 10 或 15")

case 5  :

    print( "index 的值为 5")

default :

    print( "默认 case")

}

程序输出：

index 的值为 10 或 15

在某些case中添加fallthrough语句：

let index = 10

switch index {

case 100  :

    print( "index 的值为 100")

    fallthrough

case 10,15  :

    print( "index 的值为 10 或 15")

    fallthrough

case 5  :

    print( "index 的值为 5")

default :

    print( "默认 case")

}

程序输出：

index 的值为 10 或 15

index 的值为 5

从输出中可以看到，当case 10,15对应的语句块被执行以后，case 5对应的语句块也被执行了。

### 3.4.4 guard

guard，顾名思义，就是守护。guard语句判断其后的表达式布尔值为false时，才会执行之后代码块里的代码，如果为true，则跳过整个guard语句，guard像守卫一样，在代码中设置检查点，当不满足条件时，程序执行就会被拦截下来。示例代码：

func check(age : Int){

    guard age <= 18 else {

      print("未满18岁")

      //执行return语句直接退出

return

    }

    print("年满18岁")

}

check(age: 9)

//输出：未满18岁

check(age: 20)

//输出：年满18岁

如果guard语句条件被满足，不执行else语句块。将变量或常量的可选绑定作为guard语句的条件，可以保护guard花括号后面的代码，这也就是guard的含义的由来。如果条件不满足，则会执else语句块，其中必须包含控制转移语句或者调用一个不返回的方法或函数，例如fatalError()。相比与实现同样功能的if语句，guard语句可以提高代码的可读性。

## 本章小结

本章向读者介绍了swift中的流程控制语句，包括用于遍历集合类型的for-in语句，控制代码重复执行的while循环以及选择代码不同分支执行的条件语句if和switch。随后介绍了控制代码跳转的控制转移语句，包括continue、break、fallthrough、guard。

计算机的强大之处就在于它能高效率的做重复的事情，开发者可以利用这些简单的流程控制语句组合出复杂的程序，实现强大的功能。