第 3 章

运算符与程序控制

Swift 语言具有众多的运算符，根据运算符作用的操作数的个数，可将运算符分为单目运算符、双目运算符和三目运算符。根据运算符的性质，又可分为算术运算符、关系运算符、逻辑运算符、位运算符和赋值运算符等。运算符和操作数构成表达式，或称语句，是程序执行的基本单元。程序执行主要有三种控制方式，即顺序执行、分支执行（或称选择执行）和循环执行，当考虑硬件触发中断或软件异常处理时，还有一种中断执行方式。本章将介绍 Swift 语言的运算符和主要程序控制方式。

表 3-1 为按优先级排列的全部运算符。注意，在语句中，加圆括号“()”部分运算优先级更高。

表 3-1　Swift 语言运算符

优先级别	运　算　符	含　　义
1（最高）	<<	按位左移
	>>	按位右移
	&<<	按位左移（忽略溢出）
	&>>	按位右移（忽略溢出）
2	*	乘法
	&*	乘法（带溢出）
	/	除法
	%	取余
	&	按位与
3	+	加法
	&+	加法（带溢出）
	−	减法
	&−	减法（带溢出）
	|	按位或
	^	按位异或
4	..<	左闭右开的半开区间
	....	闭区间
5	is	判断一个对象是否属于某个子类实例
	as	将子类对象用作父类实例
	as!	将子类对象强制用作父类实例
	as?	将子类对象用作父类实例（失败返回 nil）
6	??	表达式为 nil 时返回其后的默认值

续表

优先级别	运算符	含义
7	<	小于
	<=	小于或等于
	>	大于
	>=	大于或等于
	==	等于
	!=	不等于
	===	两个对象为同一个实例
	!==	两个对象不为同一个实例
	~=	模式匹配
	.<	向量逐点小于
	.<=	向量逐点小于或等于
	.>	向量逐点大于
	.>=	向量逐点大于或等于
	.==	向量逐点等于
	.!=	向量逐点不等于
8	&&	逻辑与
	.&	向量逐点按位与
9	||	逻辑或
	.|	向量逐点按位或
	.^	向量逐点按位异或
10	?:	三目运算符(或称条件运算符)
11（最低）	=	赋值
	*=	相乘与赋值复合
	/=	除法与赋值复合
	%=	取余与赋值复合
	+=	加法与赋值复合
	-=	减法与赋值复合
	<<=	按位左移与赋值复合
	>>=	按位右移与赋值复合
	&=	按位与与赋值复合
	|=	按位或与赋值复合
	^=	按位异或与赋值复合
	&*=	带溢出乘法与赋值复合
	&+=	带溢出加法与赋值复合
	&-=	带溢出减法与赋值复合
	&<<=	忽略溢出按位左移与赋值复合
	&>>=	忽略溢出按位右移与赋值复合
	.&=	向量逐点按位与与赋值复合
	.|=	向量逐点按位或与赋值复合
	.^=	向量逐点按位异或与赋值复合

表 3-1 中的运算符处于操作数中间，称这类运算符为中缀运算符。除表 3-1 中的中缀运算符外，还有 7 个可用作前缀的运算符和 1 个可用作后缀的运算，如表 3-2 所示。

表 3-2　Swift 语言的前缀和后缀运算符(设 a 为操作数)

运算符性质	运　算　符	含　　义
前缀运算符	!	逻辑非
	.!	向量逐点逻辑非
	～	按位取反
	＋	正
	－	负
	..<	"..< a"表示从首位置至 a 的前一个位置的范围
	...	"...a"表示从首位置至 a 表示的位置的范围
后缀运算符	...	"a..."表示从 a 表示的位置至最后一个位置的范围

注意区分表 3-1 和表 3-2 中表示区间的运算符"..."和"..<"。除了 Swift 标准库运算符外，用户可以自定义运算符，默认自定义运算符的优先级属于 DefaultPrecedence，略高于三目运算符；可以为自定义运算符指定优先级，表 3-1 中的各个优先级号对应的优先级组名如表 3-3 所示。

表 3-3　优先级号与优先级组名对应关系表

优先级号	优先级组名	优先级号	优先级组名
1	BitwiseShiftPrecedence	7	ComparisonPrecedence
2	MultiplicationPrecedence	8	LogicalConjunctionPrecedence
3	AdditionPrecedence	9	LogicalDisjunctionPrecedence
4	RangeFormationPrecedence	自定义	DefaultPrecedence
5	CastingPrecedence	10	TernaryPrecedence
6	NilCoalescingPrecedence	11	AssignmentPrecedence

下面将介绍常用的运算符。

3.1　算术运算符

算术运算符包括加法"＋"、减法"－"、乘法"＊"、除法"/"和取余"%"等，其中，对于加法、减法和乘法操作的运算符在计算过程中将自动进行越界管理(对于除法和取余不会溢出)，当发生溢出时报错。可容许溢出的加法"&＋"、减法"&－"和乘法"&＊"运算符将自动舍弃计算中溢出的部分。加法、减法、乘法和除法运算符可适用于整型和浮点型，而取余运算只能适用于整型数。注意，加法"＋"还可用于字符串连接操作。

下面以整型数为例，介绍算术运算符的用法，如程序段 3-1 所示。

程序段 3-1　算术运算符的用法

```
1    import Foundation
2
3    let arr1 = Array(repeating: 22, count: 5)
4    let arr2 = [Int](6...10)
5    var arr3 : [Int] = []
6    print("arr1 = \(arr1)")
7    print("arr2 = \(arr2)")
8    arr3.append(arr1[0] *arr2[0])
9    arr3.append(arr1[1] / arr2[1])
10   arr3.append(arr1[2] + arr2[2])
11   arr3.append(arr1[3] - arr2[3])
```

```
12  arr3.append(arr1[4] % arr2[4])
13  print("arr3 = \(arr3)")
14  let a : Int8 = 100
15  let b : Int8 = 120
16  let c : Int8 = a & *b
17  print(c & 0x7F)
18  print(Int(a) *Int(b)& 0x7F)
```

程序段 3-1 的执行结果如图 3-1 所示。

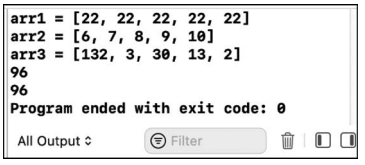

图 3-1　程序段 3-1 的执行结果

现在结合图 3-1 分析程序段 3-1 的执行过程。在程序段 3-1 中，第 3 行“let arr1＝Array(repeating:22, count:5)”定义数组常量 arr1，具有 5 个相同的元素 22。第 4 行“let arr2＝[Int](6...10)”定义数组常量 arr2，为“[6, 7, 8, 9, 10]”。第 5 行“var arr3:[Int]＝[]”定义数组变量 arr3，为空数组。

第 6 行“print("arr1＝\(arr1)")”输出数组 arr1，得到“arr1＝[22, 22, 22, 22, 22]”。第 7 行“print("arr2＝\(arr2)")”输出数组 arr2，得到“arr2＝[6, 7, 8, 9, 10]”。

第 8 行“arr3.append(arr1[0]＊arr2[0])”将 arr1[0]与 arr2[0]相乘，其积添加到 arr3 中。第 9 行“arr3.append(arr1[1]/arr2[1])”将 arr1[1]除以 arr2[1]的商添加到 arr3 中。第 10 行“arr3.append(arr1[2]＋arr2[2])”将 arr1[2]与 arr2[2]的和添加到 arr3 中。第 11 行“arr3.append(arr1[3]－arr2[3])”将 arr1[3]与 arr2[3]的差添加到 arr3 中。第 12 行“arr3.append(arr1[4]％arr2[4])”将 arr1[4]除以 arr2[4]的余数添加到 arr3 中。第 13 行“print("arr3＝\(arr3)")”输出数组 arr3，得到“arr3＝[132, 3, 30, 13, 2]”。

第 14 行“let a:Int8＝100”定义常量 a 为有符号 8 位整数，赋值为 100；第 15 行“let b:Int8＝120”定义常量 b 为有符号 8 位整数，赋值为 120。第 16 行“let c:Int8＝a &＊ b”定义常量 c 为有符号 8 位整数(范围为－128～127)，赋值为 a 与 b 的容许溢出的乘法，这里，由于操作数均为有符号 8 位整数，故乘法将自动舍弃第 7 位以上的位数(注：最低位为第 0 位)，且将第 7 位视为符号位。第 16 行中如果使用“＊”，将报溢出错误。

第 17 行“print(c & 0x7F)”将 c 与 0x7F 按位与，即将 c 的最高位清零，得到 96。第 16～17 行相当于第 18 行“print(Int(a) ＊ Int(b)& 0x7F)”，即将 a 和 b 转换为整数，作普通乘法，再与 0x7F 按位与，得到 96。

最后，再强调一下取余操作。在 Swift 语言中，求整数 a 除以整数 b 的余数 c，使用表达式 c＝a％b。这个运算与 C++语言中的取余操作有区别，这里相当于求 a 中最多包含的 b 的个数 n，然后，c＝a－n＊b。例如，30％7＝30％(－7)＝2，－30％7＝－30％(－7)＝－2，这是因为 30＝4＊7＋2＝(－4)＊(－7)＋2，－30＝(－4)＊7＋(－2)＝4＊(－7)＋(－2)。可以将“c＝a％b”理解为“c＝a－|a|％|b|＊sign(a＊b)＊b”，其中，“|a|”表示 a 的绝对值，“sign(a＊b)”表示 a 与 b 的乘积的符号。

3.2　关系运算符和条件运算符

关系运算符包括小于“<”、小于或等于“<＝”、大于“>”、大于或等于“>＝”、等于“＝＝”和不等于“!＝”等。关系运算符连接操作数得到关系表达式，关系表达式的返回值为布尔型量(true 或 false)。

条件运算符为“?:”，是 Swift 语言中唯一的三目运算符，对于形式“a?b:c”而言，若 a 为

真，则返回表达式 b 的结果，否则返回表达式 c 的结果。这里 a 必须为布尔量或返回值为布尔量的关系表达式。

下面借助程序段 3-2 介绍关系运算符和条件运算符的用法。


视频讲解

程序段 3-2　关系运算符和条件运算符实例

```
1    import Foundation
2
3    let str1 = "Hello Swift."
4    let str2 = "Hello World."
5    let n1 = 15, n2 = 18, d1 = 3.4, d2 = 5.6
6    print(str1 > str2)
7    print(str1 >= str2)
8    print(str1 < str2)
9    print(str1 <= str2)
10   print(n1 == n2)
11   print(n1 != n2)
12   print(d1>d2 ?d1 : d2)
```

在程序段 3-2 中，第 3 行“let str1＝"Hello Swift.""定义字符串常量 str1，赋值为“Hello Swift.”。第 4 行“let str2＝"Hello World.""定义字符串常量 str2，赋值为“Hello World.”。第 5 行“let n1＝15，n2＝18，d1＝3.4，d2＝5.6”定义两个整型常量 n1 和 n2 以及两个双精度浮点型常量 d1 和 d2，依次赋值为 15、18、3.4 和 5.6。

第 6 行“print(str1 > str2)”输出关系表达式“str1 > str2”的结果，由于“S”<“W”，故“str1 < str2”，这里输出 false。第 7 行“print(str1 >= str2)”输出关系表达式“str1 >＝str2”的结果，这里输出 false。第 8 行“print(str1 < str2)”输出关系表达式“str1 < str2”的结果，得到 true。第 9 行“print(str1 <= str2)”输出关系表达式“str1 <＝str2”的结果，得到 true。

第 10 行“print(n1＝＝n2)”输出关系表达式“n1＝＝n2”的结果，得到 false。第 11 行“print(n1!＝n2)”输出关系表达式“n1!＝n2”的结果，得到 true。

第 12 行“print(d1 > d2 ? d1:d2)”中，“d1 > d2 ? d1:d2”借助条件运算符计算 d1 和 d2 中的最大者，这里输出 5.6。

最后，需要补充说明两点：①关系运算符不能级联，例如，类似于“3 < 4 < 5”这种用法是错误的；②条件运算符可以嵌套，例如，形如“5 > 3 ? (7 > 8 ? 10:12): 13”这种用法是正确的。

3.3　逻辑运算符

在 Swift 语言中，逻辑常量仅有 2 个，即 true 与 false。逻辑运算符只能连接逻辑量，并得到一个新的逻辑量。常用的逻辑运算符为逻辑非“!”、逻辑与“&&”和逻辑或“||”，优先级从高至低依次为逻辑非>逻辑与>逻辑或。逻辑运算符的运算规则如表 3-4 所示。

表 3-4　逻辑运算符的运算规则

逻辑运算符	运 算 规 则
逻辑非“!”	!false＝true，!true＝false
逻辑与“&&”	false && false＝false && true＝true && false＝false， true && true＝true
逻辑或“||”	false || true＝true || false＝true || true＝true， false || false＝false

逻辑表达式主要用于条件控制，在 Swift 语言中，表示“控制条件”的表达式结果只能为逻辑量。程序段 3-3 展示了逻辑运算符的用法。


视频讲解

程序段 3-3 逻辑运算符用法实例

```
1    import Foundation
2
3    var a1,a2 : Int
4    a1 = -12
5    a2 = 7
6    var b1, b2 : Bool
7    b1 = a1 >= a2
8    b2 = a1 < a2
9    print("b1=\(b1),!b1=\(!b1),b2=\(b2),!b2=\(!b2)")
10   print("b1 && b2 = \(b1 && b2), b1 || b2 = \(b1 || b2)")
11   if b1
12   {
13       print("\(a1)>=\(a2)")
14   }
15   if b2
16   {
17       print("\(a1)<\(a2)")
18   }
```

程序段 3-3 的执行结果如图 3-2 所示。

```
b1=false,!b1=true,b2=true,!b2=false
b1 && b2 = false, b1 || b2 = true
-12<7
Program ended with exit code: 0
All Output ⌃   Filter
```

图 3-2 程序段 3-3 的执行结果

现在，结合图 3-2 分析程序段 3-3 的执行情况。在程序段 3-3 中，第 3 行“var a1,a2:Int”定义两个整型变量 a1 和 a2(默认均被初始化为 0)。第 4 行“a1＝－12”将－12 赋给 a1，注意，负号前有一个空格，在 Swift 语言中，尽可能在独立的语言元素(如变量、常量和运算符)间插入一个空格。第 5 行“a2＝7”将 7 赋给变量 a2。

第 6 行“var b1, b2:Bool”定义两个布尔型(即逻辑型)变量 b1 和 b2，默认初始化值均为 false。第 7 行“b1＝a1 >= a2”将关系表达式“a1 >= a2”的结果赋给 b1，这里为 false。第 8 行“b2＝a1 < a2”将关系表达式“a1 < a2”的结果赋给 b2，这里为 true。

第 9 行“print("b1＝\(b1),!b1＝\(!b1),b2＝\(b2),!b2＝\(!b2)")”输出 b1、b1 的逻辑非、b2 和 b2 的逻辑非，将得到“b1＝false, !b1＝true, b2＝true, !b2＝false”。第 10 行“print("b1 && b2＝\(b1 && b2), b1 || b2＝\(b1 || b2)")”输出 b1 与 b2 的逻辑与和逻辑或，得到“b1 && b2＝false, b1 || b2＝true”。

第 11～14 行为一个 if 结构，由于 b1 为假，第 12～14 行不执行。第 15～18 行也为一个 if 结构，由于 b2 为真，第 16～18 行被执行，其中第 17 行“print("\(a1)<\(a2)")”输出“－21＜7”。

注意：不能把布尔量作为整型量使用，一般借助条件运算符将布尔量转换为整型，例如，设 a 为布尔量，则“a ? 1:0”返回 1(若 a 为真)或 0(若 a 为假)。

3.4 位运算符与区间运算符

位运算符只能适用于整型，一般地，位运算符主要用于无符号整型。常用的位运算符有按位取反“～”、按位左移“<<”、按位右移“>>”、按位左移(忽略溢出)“&<<”、按位右移(忽略溢出)“&>>”、按位与“&”、按位或“|”和按位异或“^”等。其中，“按位取反”优先级最高；“按位

左移”“按位右移”“按位左移(忽略溢出)”和“按位右移(忽略溢出)”的优先优相同,低于“按位取反”运算符;然后,“按位与”运算符的优先级更低;最后,“按位或”与“按位异或”优先级相同,它们的优先级最低。

位运算符直接作用于操作数的各位,设 a 和 b 为无符号 8 位整型,不妨设 a 为 23,b 为 145,则表 3-5 给出了位运算符的计算情况。

表 3-5　位运算符计算结果(设 a=23(0b00010111),b=145(0b10010001))

位　运　算	计 算 结 果	位　运　算	计 算 结 果
~a	0b11101000	a& b	0b00010001
~b	0b01101110	a | b	0b10010111
a>>2	0b00000101	a ^ b	0b10000110
a<<2	0b01011100	(a&(1<<2))>>2	1
a>>9	0b00000000	a | (1<<6)	0b01010111
a&>>9	0b00001011	a ^(1<<5)	0b00110111
b>>2	0b00100100	a& ~(1<<2)	0b00010011
b<<2	0b01000100	a& 0x0F	0b00000111

位运算有一些特殊的用法,例如,右移一位相当于整数值除以 2;左移一位相当于整数值乘以 2;将一个整数 a 的第 n 位设为 1 而其余位不变,可用“a | (1<<n)”;将整数 a 的第 n 位设为 0 而其余位不变,可用“a & ~(1<<n)”;将整数 a 的第 n 位取反而其余位不变,可以用“a ^(1<<n)”。

区间运算符只有两种表示:“...”和“..<”。设 a 和 b 为索引,则区间运算符可有五种情况:①“a...b”表示从 a 至 b 的闭区间,包含两个端点;②“a...”表示从 a 至最后一个索引号的闭区间,包括两个端点;③“...b”表示从首索引至 b 的闭区间,包括两个端点;④“a..<b”表示从 a 至 b 的半开区间,包括 a 但不包括 b;⑤“..<b”表示从首索引至 b 的半开区间,包括首位置,但不包括 b 端点。

程序段 3-4 展示了位运算符和区间运算符的用法。

程序段 3-4　位运算符与区间运算符用法实例

```
1    import Foundation
2
3    let a : UInt8 = 23
4    let b : UInt8 = 145
5    let sa = String(a, radix: 2)
6    print("a: 0b" + String(repeating: "0", count: 8-sa.count)+sa)
7    let sb = String(b, radix: 2)
8    print("b: 0b" + String(repeating: "0", count: 8-sb.count)+sb)
9    var res = [~a,~b,a >> 2,a << 2, a >> 9, a &>> 9, b >> 2, b << 2, a & b, a | b, a ^ b]
10
11   for (i,e) in res.enumerated()
12   {
13       print("res["+String(format: "%2d", i)+"]: 0b" + String(repeating: "0",
         count: 8-String(e,radix: 2).count)+String(e,radix: 2))
14   }
15
16   var c1,c2,c3,c4: UInt8
17   c1 = (a &(1<<2)) >> 2
18   c2 = a | (1<<6)
19   c3 = a ^(1<<5)
```

```
20  c4 = a & ~(1<<2)
21  print("c1 = \(c1), c2 = \(c2), c3 = \(c3), c4 = \(c4)")
22  print("c1: 0b" + String(repeating: "0", count: 8-String(c1,radix: 2).count)+
    String(c1,radix: 2))
23  print("c2: 0b" + String(repeating: "0", count: 8-String(c2,radix: 2).count)+
    String(c2,radix: 2))
24  print("c3: 0b" + String(repeating: "0", count: 8-String(c3,radix: 2).count)+
    String(c3,radix: 2))
25  print("c4: 0b" + String(repeating: "0", count: 8-String(c4,radix: 2).count)+
    String(c4,radix: 2))
26
27  let arr = Array(10...19)
28  print("arr[3...5] = \(arr[3...5])")
29  print("arr[3..<5] = \(arr[3..<5])")
30  print("arr[3...]  = \(arr[3...])")
31  print("arr[...5]  = \(arr[...5])")
32  print("arr[..<5]) = \(arr[..<5])")
33  let str = "Hello Swift."
34  print("\(str[..<str.index(str.endIndex,offsetBy: -4)])")
```

程序段 3-4 的执行结果如图 3-3 所示。

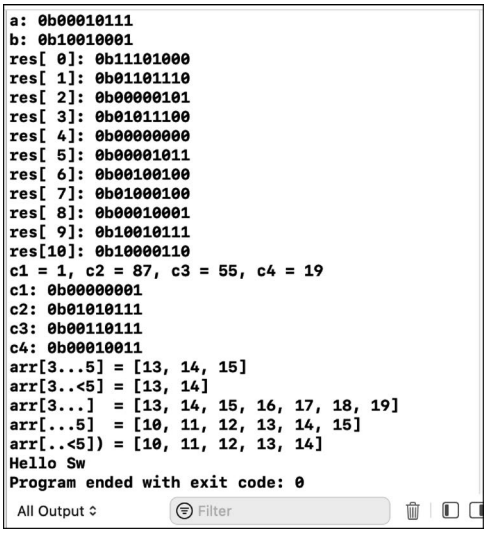

图 3-3　程序段 3-4 的执行结果

下面结合图 3-3 介绍程序段 3-4。在程序段 3-4 中，第 3 行“let a:UInt8＝23”定义无符号 8 位整型常量 a，赋值为 23；第 4 行“let b:UInt8＝145”定义无符号 8 位整型常量 b，赋值为 145。第 5 行“let sa＝String(a, radix:2)”将 a 转换为二进制数表示的字符串；第 6 行“print("a:0b"＋String(repeating:"0", count:8-sa.count)＋sa)”以 8 位长显示 a 的二进制数表示，输出“0b00010111”。第 7 行“let sb＝String(b, radix:2)”将 b 转换为二进制数表示的字符串；第 8 行“print("b:0b"＋String(repeating:"0", count:8-sb.count)＋sb)”以 8 位长显示 b 的二进制数表示，输出“0b10010001”。

第 9 行“var res＝[～a,～b,a >> 2,a << 2, a >> 9, a &>> 9, b >> 2, b << 2, a & b, a | b, a ^ b]”定义数组 res，其元素依次为 a 的按位取反值、b 的按位取反值、a 右移 2 位、a 左移 2 位、a 右移 9 位、a 忽略溢出右移 9 位、b 右移 2 位、b 左移 2 位、a 与 b、a 或 b、a 异或 b。第 11～14 行为一个 for-in 循环结构，用于输出数组 res。这里，“res.enumerated()”将数组的

每个元素序号和元素值打包成一个元组返回，第 11 行“for (i,e) in res.enumerated()”中 i 为元素序号(即下标值)，e 为 i 对应的元素值。第 13 行“print("res["＋String(format:"%2d", i)＋"]:0b"＋String(repeating:"0", count:8-String(e,radix:2).count)＋String(e,radix:2))”格式化输出 i 和 e 的值，例如，对于 res[0]得到“res[0]:0b11101000”。表 3-5 和图 3-3 中列出了数组 res 的所有结果。

第 16 行“var c1,c2,c3,c4:UInt8”定义无符号 8 位整型变量 c1、c2、c3 和 c4。第 17 行“c1＝(a &(1 << 2)) >> 2”计算 a 的第 2 位(注：从右向左从第 0 位算起)上的值，赋给 c1。第 18 行“c2＝a | (1 <<6)”将 a 的第 6 位置为 1，然后，赋给 c2。第 19 行“c3＝a ^(1 <<5)”将 a 的第 5 位取反后赋给 c3。第 20 行“c4＝a & ～(1 << 2)”将 a 的第 2 位清零后赋给 c4。第 21 行“print("c1＝\(c1)，c2＝\(c2)，c3＝\(c3)，c4＝\(c4)")”以十进制数格式输出 c1、c2、c3 和 c4 的值，得到“c1＝1，c2＝87，c3＝55，c4＝19”。第 22～25 行以二进制形式输出 c1、c2、c3 和 c4 的值，如图 3-3 所示。

第 27 行“let arr＝Array(10...19)”定义常量数组 arr，包含 10～19 共 10 个整数。第 28 行“print("arr[3...5]＝\(arr[3...5])")”输出 arr 的第 3～5 个元素，得到“arr[3…5]＝[13，14，15]”。第 29 行“print("arr[3..< 5]＝\(arr[3..< 5])")”输出 arr 的第 3～4 个元素，得到“arr[3..< 5]＝[13，14]”。第 30 行“print("arr[3...]＝ \(arr[3...])")”输出 arr 的第 3 个至最后一个元素，得到“arr[3...]＝[13，14，15，16，17，18，19]”。第 31 行“print("arr[...5]＝\(arr[...5])")”输出 arr 的首个至第 5 个元素，得到“arr[...5]＝[10，11，12，13，14，15]”。第 32 行“print("arr[..<5]＝ \(arr[..<5])")”输出 arr 的首个至第 4 个元素，得到“arr[..<5]＝[10，11，12，13，14]”。

第 33 行“let str＝"Hello Swift."”定义字符串常量为“Hello Swift.”。第 34 行“print(" \(str[..< str.index(str.endIndex,offsetBy:－4)])")”输出字符串 str 的首个字符至倒数第 5 个位置间的子串，得到“Hello Sw”。

3.5　赋值和复合赋值运算符

在 Swift 语言中，赋值运算符为“＝”。赋值运算符可以和算术运算符或位运算符复合使用，例如，a＝a ＋ b 可以简写为 a ＋＝ b，后者的“＋＝”为复合赋值运算符，这两个表达式都表示将 a 与 b 作加法运算，其结果赋给 a。常用的复合赋值运算符有“＊＝”“&＊＝”“/＝”“%＝”“＋＝”“&＋＝”“－＝”“&－＝”“<<＝”“& <<＝”“>>＝”“& >>＝”“&＝”“|＝”“^＝”等。

赋值和复合赋值运算符的优先级是所有运算符中级别最低的。注意区分赋值运算符“＝”与关系运算符中的等于运算符“＝＝”，赋值运算符将其右边表达式的计算结果赋给左边的变量或常量(注：常量只能在定义时赋值一次)，等于运算符用于判断其两端的表达式的结果是否相等，整个表达式的判定值为逻辑值。

程序段 3-5 是赋值和复合赋值运算符的应用实例。


视频讲解

程序段 3-5　赋值和复合赋值运算符实例

```
1    import Foundation
2
3    var sum = 0
4    for i in 1...100
5    {
6        sum += i
```

```
7    }
8    print("1+2+...+100 = \(sum)")
```

在程序段 3-5 中，第 3 行“var sum＝0”定义变量 sum，赋值为 0。第 4～7 行为一个 for-in 结构，i 从 1 遍历到 100，执行第 6 行 100 次。第 6 行“sum ＋＝ i”使用复合赋值运算符实现 sum 与 i 相加的和赋给 sum。第 8 行“print("1＋2＋...＋100＝\(sum)")”输出“1＋2＋...＋100＝5050”。

3.6 程序执行方式

程序主要有三种执行方式，即顺序执行、分支执行和循环执行。程序总的执行方式为顺序执行，在 Swift 语言中，程序入口为 main.swift，然后，按照程序指令的“先后”位置顺序执行。为了实现复杂的算法，Swift 语言带有分支和循环控制语句，可以实现程序代码的有条件执行和循环执行。下面首先回顾程序的顺序执行方式。

3.6.1 顺序执行方式

Swift 语言中，程序会按照语句的位置先后关系顺序执行，这是程序的基本执行方式，如程序段 3-6 所示，这个程序段实现了求三角形面积和周长。


视频讲解

程序段 3-6 顺序执行程序实例

```
1    import Foundation
2
3    let triangle = (3.4, 2.6, 4.7)
4    var area,peri : Double
5    peri = triangle.0+triangle.1+triangle.2
6    let s = peri/2.0
7    area = sqrt(s*(s-triangle.0)*(s-triangle.1)*(s-triangle.2))
8    print("perimeter: \(peri), area: \(String(format: "%5.2f", area))")
```

在程序段 3-6 中，语句按先后顺序依次执行。第 3 行“let triangle＝(3.4，2.6，4.7)”定义元组 triangle，包括了三角形的三条边长。第 4 行“var area，peri：Double”定义变量 area 和 peri，分别用于保存三角形的面积和周长。

第 5 行“peri＝triangle.0＋triangle.1＋triangle.2”计算三角形的周长，赋给 peri。第 6 行“let s＝peri/2.0”定义常量 s，保存周长的一半。第 7 行“area＝sqrt(s＊(s-triangle.0)＊(s-triangle.1)＊(s-triangle.2))”利用海伦公式计算三角形的面积，赋给 area。

第 8 行“print("perimeter：\(peri)，area：\(String(format："%5.2f"，area))")”输出三角形的周长和面积，得到“perimeter：10.7，area：4.32”。

3.6.2 分支执行方式

分支执行方式，又称选择执行方式或有条件执行方式。Swift 语言通过 if 结构、switch 结构和 guard 结构实现分支执行。if 结构和 switch 结构均可以嵌套(包括互相嵌套和多级嵌套)使用，例如，if 结构中包含另一个 if 结构、if 结构中包括一个 switch 结构、switch 结构中包含一个 if 结构等。

if 结构的基本形式有以下三种。

(1) 基本 if 结构。

```
if  条件表达式
{
    条件表达式为真时执行的语句或语句组
}
```

在上述基本 if 结构中,如果 if 后的“条件表达式”为真,则执行“{ }”括起来的语句;如果“条件表达式”为假,则跳至“}”后面的语句执行。

(2) 基本 if-else 结构。

```
if  条件表达式
{
    条件表达式为真时执行的语句或语句组
}
else
{
    条件表达式为假时执行的语句或语句组
}
```

在上述基本 if-else 结构中,当条件表达式为真时,执行 if 下的“{ }”括住的语句;否则,执行 else 下的“{ }”括住的语句。基本 if-else 结构实现了两路选择。

(3) 多选择 if-else 结构。

```
if  条件表达式 1
{
    条件表达式 1 为真时执行的语句或语句组
}
else if  条件表达式 2
{
    条件表达式 2 为真时执行的语句或语句组
}
.....//省略多个 elseif 块
else if  条件表达式 n
{
    条件表达式 n 为真时执行的语句或语句组
}
else
{
    上述条件表达式 1 至 n 均为假时执行的语句或语句组
}
```

上述多选择 if-else 结构中,else 块可以省略。注意,if 或 else 部分不能为空,如果没有语句,则需要插入 break。

下面举例说明 if 结构的用法,如程序段 3-7 所示,这个程序段实现了一元二次方程求根运算。


视频讲解

程序段 3-7　if 结构用法实例

```
1    import Foundation
2
3    print("Please input a:")
4    let a = Double(readLine() ??"1.0") ??1.0
5    print("Please input b:")
6    let b = Double(readLine() ??"2.0") ??2.0
7    print("Please input c:")
```

```
8    let c = Double(readLine() ??"1.0") ??1.0
9    var x1, x2 : Double
10   if abs(a) < 1E-8
11   {
12       print("Not a quadratic equation.")
13   }
14   else
15   {
16       let delta = b *b - 4*a *c
17       if delta > 1E-8
18       {
19           x1 = (-b + sqrt(delta))/(2.0*a)
20           x2 = (-b - sqrt(delta))/(2.0*a)
21           print("x1 = \(String(format: "%5.2f", x1)), x2 = \(String(format: "%5.
             2f", x2))")
22       }
23       else if abs(delta) < 1E-8
24       {
25           x1 = -b/(2.0*a)
26           print("x1 = x2 = \(String(format: "%5.2f", x1))")
27       }
28       else
29       {
30           x1 = -b/(2.0*a)
31           x2 = sqrt(-delta)/(2.0*a)
32           print("x1 = \(String(format: "%5.2f", x1)) + \(String(format: "%5.2f",
             x2))i", terminator: ", ")
33           print("x2 = \(String(format: "%5.2f", x1)) - \(String(format: "%5.2f",
             x2))i")
34       }
35   }
```

在程序段 3-7 中，第 3 行“print("Please input a:")”输出提示信息“Please input a:”。第 4 行“let a=Double(readLine() ?? "1.0") ?? 1.0”从键盘读入浮点型数值，赋给 a，如果输入非法，则 a 将赋值 1.0。第 5 行 print("Please input b:")”输出提示信息“Please input b:”。第 6 行“let b=Double(readLine() ?? "2.0") ?? 2.0”从键盘读入浮点型数值，赋给 b，如果输入非法，则 b 将赋值 2.0。第 7 行 print("Please input c:")”输出提示信息“Please input c:”。第 8 行“let c=Double(readLine() ?? "1.0") ?? 1.0”从键盘读入浮点型数值，赋给 c，如果输入非法，则 c 将赋值 1.0。

第 9 行“var x1, x2:Double”定义双精度浮点型变量 x1 和 x2，用于保存一元二次方程的两个根。第 10～35 行为一个 if-else 结构，在 else 部分嵌套了一个 if-else 结构。第 10 行“if abs(a) < 1E−8”表示如果 a 的绝对值小于 10^{-8}，则认为 a 近似为 0，则执行第 12 行“print("Not a quadratic equation.")”输出“Not a quadratic equation.”；否则，认为输入的 a 非 0，则执行第 15～35 行。

第 16 行“let delta=b * b− 4 * a * c”计算判别式 delta 的值。第 17～34 行为多选择的 if-else 结构，分为三种情况。

(1) delta 大于 0，即第 17 行为真，这里用“delta > 1E−8”表示 delta 大于 0。此时，一元二次方程具有两个不等的实根，第 19 行“x1=(−b + sqrt(delta))/(2.0 * a)”计算第一个实根 x1，第 20 行“x2=(−b−sqrt(delta))/(2.0 * a)”计算第二个实根 x2。第 21 行“print("x1=

\(String(format:"%5.2f", x1)), x2=\(String(format:"%5.2f", x2))")"输出两个实根的值。

(2) delta 等于 0,即第 23 行的条件"abs(delta) < 1E-8"为真,这里用 delta 的绝对值小于 10^{-8} 表示 delta 为 0。此时,一元二次方程具有两个相同的实根,第 25 行"x1=-b/(2.0 * a)"计算这个实根,第 26 行"print("x1=x2=\(String(format:"%5.2f", x1))")"输出两个相等的实根。

(3) 当上述两个情况都不满足时,即 delta 小于 0 时,执行第 30～33 行,此时一元二次方程具有一对共轭复根。第 30 行"x1=-b/(2.0 * a)"计算根的实部,赋给 x1;第 31 行"x2=sqrt(-delta)/(2.0 * a)"计算根的虚部,赋给 x2。第 32 行"print("x1=\(String(format:"%5.2f", x1)) + \(String(format:"%5.2f", x2))i",terminator:", ")"输出第一个复根。第 33 行"print("x2=\(String(format:"%5.2f", x1)) - \(String(format:"%5.2f", x2))i")"输出第二个复根。

程序段 3-7 的执行结果如图 3-4 所示。

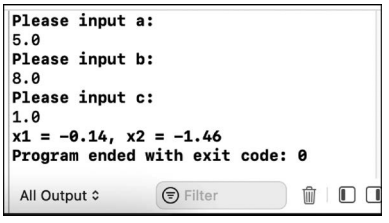

图 3-4　程序段 3-7 的执行结果

下面介绍 switch 结构的两种形式。

(1) 标准 switch 结构。

```
switch 表达式
{
    case 值 1:
        语句组 1
    case 值 2:
        语句组 2
    case 值 3:
        语句组 3
    …
    case 值 n:
        语句组 n
    default:
        语句组 n+1
}
```

在 switch 结构中,当"表达式"的值为"值 1"时,执行"语句组 1",然后,跳出 switch 结构;当"表达式"的值为"值 2"时,执行"语句组 2",然后,跳出 switch 结构;以此类推,当"表达式"的值为"值 n"时,执行"语句组 n";当"表达式"的值不为上述任一值时,执行"default"下的"语句组 n+1"。注意,switch 结构中的 case 部分必须涵盖所有的情况,如果不能包含所有情况,则必须添加"default"部分。

在 switch 结构中,若"表达式"的值为多个值,且执行同一个处理时,则在 case 中罗列这些值,并使用",,"分隔它们,形式如下所示:

```
case 值 m, 值 m+1, 值 m+2, 值 m+3:
    "表达式"为上述任一值时执行的语句组
```

此外,可以在 case 部分使用范围(或区间)运算符"..<"或"...",例如:

```
case  0..<100:
    当"表达式"的值为大于或等于 0 且小于 100 的整数时执行的语句组
```

(2) 具有"值绑定"和 where 子句的 switch 结构。

```
switch 表达式
```

```
{
    case let x where 条件表达式 1:
        语句组 1
    case let y where 条件表达式 2:
        语句组 2
    …
    case _:
        语句组 n+1
}
```

在上述 switch 结构中，当“条件表达式 1”为真时，“表达式”的值赋给 x，然后，执行“语句组 1”，这里的“x”的作用域为“语句组 1”，称为将 x 绑定在“语句组 1”中。如果 x 的值在“语句组 1”中可被改变，则使用“var x where 条件表达式 1”。当“条件表达式 1”为假时，判断“条件表达式 2”的值，若其为真，则将“表达式”的值赋给 y，并执行“语句组 2”，以此类推；如果上述所有条件都不满足时，则“表达式”将匹配“_”（其中，“_”表示任意值），并执行“语句组 n＋1”。

在带有“值绑定”的 switch 结构中，where 子句可根据情况使用，是可选项。一般使用元组作为 switch 的表达式，可实现多个值的匹配，下面以二元元组为例，介绍一种典型形式：

```
switch (e1,e2)                                    // e1 和 e2 为元组的两个元素
{
    case let (x, 0):
        当 e2 为 0 时执行的语句组，其中 e1 赋给 x
    case let (x,y) where x>y:
        将 e1 赋给 x、e2 赋给 y，当 x>y 时执行的语句组
    case let (x,y):
    将 e1 赋给 x、将 e2 赋给 y，然后执行这时的语句组
}
```

在上述 switch 结构中，首先判断给定的元组“(e1, e2)”是否匹配形式“let(x, 0)”，即 e2 为 0 的情况(e1 赋给 x)，如果匹配成功，则执行“当 e2 为 0 时执行的语句组，其中 e1 赋给 x”。如果匹配不成功，则匹配“let (x,y) where x＞y”，即“将 e1 赋给 x、将 e2 赋给 y”且 x 大于 y 的情况，如果匹配成功，则执行“将 e1 赋给 x、e2 赋给 y，当 x＞y 时执行的语句组”。如果匹配不成功，则执行“case let (x,y)”部分，这种情况包含了其上述 case 匹配不成功的所有情况，此时将 e1 赋给 x、将 e2 赋给 y，执行“将 e1 赋给 x、将 e2 赋给 y，然后执行这时的语句组”，这里无须 default 部分。

在带有“值绑定”和 where 子句的 switch 结构中，case 的“值”也可以组合使用，例如：

```
switch 表达式
{
    case let x where x>10 && x<60, let x where x<-10 && x>-60:
        语句组 1
    case _:
        语句组 2
}
```

在上述 switch 结构中，首先将“表达式”的值匹配 case 的条件“let x where x＞10 && x＜60，let x where x＜－10 && x＞－60”，即将“表达式”的值赋给 x，如果 x 大于 10 且小于 60，或者 x 小于－10 且大于－60，则执行“语句组 1”；否则，执行“语句组 2”。

注意：case 部分不能为空，如果 case 部分没有语句，则需要插入 break。

程序段 3-8 介绍了 switch 结构的用法。

程序 3-8　switch 结构用法实例


```
1    import Foundation
2
3    let i = Int.random(in: 1...21)
4    switch i
5    {
6    case 1..<7:
7        print(i,"is a small number.")
8    case 7, 14,21:
9        print(i,"is a big prize.")
10   case let x where x % 2==0:
11       print(x,"is an even number.")
12   default:
13       print(i,"is an odd number.")
14   }
15
16   let (j,k)=(Int.random(in: 1...100),Double.random(in: 0...1))
17   switch (j,k)
18   {
19   case (let x, let y) where y>0.5,(let x,let y) where Double(x)*y>80:
20       print("x=",x,", y=",String(format: "%4.2f", y),", y>0.5 or x*y>80")
21   case (let x,let y) where x<50,(let x,let y) where Double(x)*y<20:
22       print("x=",x,", y=",String(format: "%4.2f", y),", x<50 or x*y<20")
23   case let (_,y):
24       print("j=",j,", y=",String(format: "%4.2f", y))
25   }
```

在程序段 3-8 中，第 3 行“let i＝Int. random(in:1...21)”生成一个 1～21 的伪随机整数，赋给常量 i。第 4～14 行为一个 switch 结构，根据 i 的值做出选择：首先，与第 6 行“case 1..＜7”进行匹配，即判断 i 的值是否在 1～7(不含后者)之间，若是，则执行第 7 行“print(i,"is a small number.")”输出 i 的值和字符串“is a small number.”，之后跳出 switch 结构，去执行第 16 行。如果第 6 行匹配失败，则与第 8 行“case 7，14,21”相匹配，即判断 i 的值是否为 7、14 或 21，如果是，则执行第 9 行“print(i,"is a big prize.")”输出 i 的值和字符串“is a big prize.”，之后跳至第 16 行执行。如果第 8 行匹配失败，则与第 10 行“case let x where x ％ 2＝＝0”匹配，即将 i 的值赋给 x，判断 x 是否为偶数，如果 x 为偶数，则执行第 11 行“print(x,"is an even number.")”输出 x 的值和字符串“is an even number.”。如果 x 不为偶数，则执行第 12～13 行，即 default 部分，输出 i 的值和字符串“is an odd number.”。

第 16 行“let (j,k)＝(Int. random(in:1...100),Double. random(in:0...1))”定义一个二元元组，包含一个 1～100 的伪随机整数和一个 0～1 的伪随机双精度浮点数。第 17～25 行为一个 switch 结构，根据元组(j,k)的值做出选择，共有三种情况：①第 19 行“case (let x, let y) where y＞0.5,(let x,let y) where Double(x) * y＞80”，表示将 j 的值赋给 x，将 k 的值赋给 y，这里包含了两个条件，即 y 大于 0.5 或者 x 与 y 的积大于 80；②第 21 行“case (let x,let y) where x＜50,(let x,let y) where Double(x) * y＜20”，表示将 j 的值赋给 x，将 k 的值赋给 y，这里也包含了两个条件，即 x 的值小于 50 或者 x 与 y 的积小于 20；③第 23 行“case let (_, y):”，这里不关注 j 的值，只是将 k 的值赋给 y。第 17 行的 switch 语句依次与上述三种情况进行匹配，然后，执行匹配成功的情况下的语句，执行完后跳出 switch 结构。

注意：switch 结构的各个 case 部分无须添加 break，这一点和 C 语言不同；如果某个 case

执行完后，希望其相邻的下一个 case 部分被直接执行(即无须判断这个 case 的条件部分)，则在该 case 的执行语句后添加语句 fallthrough。

除了 if 结构和 switch 结构外，还有一种分支结构，即 guard 结构。guard 结构类似于 if 结构，但是 guard 结构必须有 else 部分，并且 guard 结构一般只用于函数中，保证某些条件为真，其典型结构如下：

```
guard 条件表达式
else
{
    语句组
    return
}
```


视频讲解

程序段 3-9 是继程序段 3-8 的部分代码，展示 guard 结构的用法。

程序段 3-9 guard 结构用法实例

```
26  func show(_ x: Int)
27  {
28      guard x>=0
29      else
30      {
31          print("x is negative.")
32          return
33      }
34      print(String(format:"x=%4d",x))
35  }
36  show(10)
37  show(-5)
```

在程序段 3-9 中，第 26 行“func show(_ x:Int)”定义了一个函数 show，具有一个整型参数 x，函数用法将在第 4 章中介绍。第 27～35 行为 show 函数体，第 28～33 行为一个 guard 结构，第 28 行“guard x>=0”判断 x 是否大于或等于 0，如果为真，则执行 guard 结构之后的语句，即第 34 行语句；如果为假，则执行第 30～33 行，输出字符串“x is negative.”，调用 return 跳出函数 show。

第 36 行“show(10)”由于参数 10 为正数，故调用 show 函数输出“x=10”；第 37 行“show(-5)”由于参数-5 为负数，故调用 show 函数输出“x is negative.”。

除了上述分支控制结构外，还有 continue 和 break 语句可用于控制程序跳转，这两个语句不但可以用于 if 分支结构和 switch 分支结构，也可以用于 while 循环结构和 for 循环结构，故将这两个语句放在 3.6.3 节循环执行方式时介绍。此外，异常处理 throw 语句也可以实现程序跳转，将在第 7 章中介绍。

本节最后介绍一下借助 if 结构实现系统版本检查的方法。

在 if 结构中，可以通过 available 内置方法指定程序代码的最低运行平台，如程序段 3-10 所示。


视频讲解

程序段 3-10 设定程序代码运行的最低平台

```
1   if #available(iOS 16.1.1,macOS 13.2.1, *)
2   {
3       print("The program can be executed.")
4   }
5   else
```

```
6    {
7        print("The program need to be updated.")
8    }
9
10   @available(macOS 13.2.1,*)
11   struct student
12   {
13       var name = "John"
14   }
15   if #available(macOS 13.2.1, *)
16   {
17       let s = student()
18       print(s.name)
19   }
20   else
21   {
22       print("student cannot be used.")
23   }
```

在程序段 3-10 中，第 1 行“if #available(iOS 16.1.1，macOS 13.2.1，*)”表示如果程序运行在 iOS 系统版本 16.1.1 及以上平台或者 macOS 系统版本 13.2.1 及以上平台时，执行第 3 行“print("The program can be executed.")”，输出信息“The program can be executed.”，否则（第 5 行“else”），执行第 7 行“print("The program need to be updated.")”，输出信息“The program need to be updated.”。第 1 行中的“*”表示没有其他的平台。在我们使用的 MacBook 笔记本电脑上，由于其系统为 macOS 13.2.1，故执行第 1～8 行代码时，将输出信息“The program can be executed.”。

第 10 行“@available(macOS 13.2.1，*)”称为装饰器，也称“语法糖”，表示其下相邻的结构工作在 macOS 13.2.1 系统及其以上平台上，这里表示第 11～14 行的 student 结构工作在 macOS 13.2.1 系统及其以上平台上。第 15～23 行为一个 if 结构，第 15 行“if #available(macOS 13.2.1，*)”判断如果平台为 macOS 13.2.1 及其以上系统时，执行第 17～18 行，定义一个结构体常量 s，输出 s 的成员 name；否则执行第 22 行“print("student cannot be used.")”输出提示信息“student cannot be used.”。

对程序做版本检查的好处在于使程序随着系统版本的升级而同步更新，保证程序使用最新的 API（应用程序接口）函数。

3.6.3　循环执行方式

在 Swift 语言中，主要有两种循环执行控制方式：for-in 结构和 while 结构。while 结构又细分为当型 while 结构和直到型 while 结构，后者称为 repeat-while 结构。下面首先介绍 for-in 结构。

循环控制方式 for-in 结构可用于区间中的整数值遍历、字符串中的字符遍历、字典中的元素遍历等，常用的形式如下。

（1）典型 for-in 结构。

```
for 元素 in 范围
{
    语句组
}
```

上述 for-in 结构中，遍历“范围”中的各个“元素”，对于每个“元素”执行“语句组”一次。这里无须为“元素”定义变量类型，自动根据“范围”中的数组类型设定“元素”的数据类型。

如果“元素”使用“_”替换，则表示不关注“范围”中的具体元素，只关注循环次数。表示“范围”的方法可借助运算符“...”和“..<”，或者使用函数 stride(from:to:by:)或 stride(from:through:by:)实现，两个 stride 的区别在于前者不包含“to”参数指定的边界，而后者包含“through”参数指定的边界。例如：stride(from:1, to:10, by:3)表示 1～10、步长为 3 生成的序列，即 1、4、7，不包含 10；stride(from:1, through:10, by:3)表示 1～10、步长为 3 生成的序列，即 1、4、7、10，包含 10。

for-in 结构中的“范围”可以为字符串或字典。

(2) 用于字典遍历的 for-in 结构。

```
for 二元元组 in 字典
{
    语句组
}
```

在上述 for-in 结构中，使用“二元元组”作为字典中每个元素的返回值，例如，将二元元组表示为“(k,v)”，则 k 将对应着字典元素的键值，v 将对应着同一个字典元素的值。如果 k 用“_”表示，则将只关心字典元素的值，而不关心它的键值。如果 v 用“_”表示，则将只关心字典元素的键值，而不关心它的值。


视频讲解

程序段 3-11 介绍了 for-in 结构的用法。

程序段 3-11 for-in 结构的用法实例

```
1    import Foundation
2    var s = 1
3    for e in 1...10
4    {
5        s *= e
6    }
7    print("10!=",s)
8
9    s = 1
10   for _ in 1...10
11   {
12       s *= 2
13   }
14   print("2^10=",s)
15
16   s = 0
17   for e in stride(from: 1, to: 100, by: 2)
18   {
19       s += e
20   }
21   print("1+3+...+99 = ",s)
22
23   s = 0
24   for e in stride(from: 1, through: 100, by: 1)
25   {
26       if e % 2 == 0
27       {
28           continue
```

```
    }
    s += e
}
print("1+3+...+99 = ",s)

s = 0
for e in 1...200
{
    if e % 2 == 0
    {
        continue
    }
    if e>=100
    {
        break
    }
    s += e
}
print("1+3+...+99 = ",s)

s = 0
mylabel: for e in 1...100
{
    if e % 2 == 0
    {
        continue mylabel
    }
    if e >= 100
    {
        break mylabel
    }
    s += e
}
print("1+3+...+99 = ",s)

let p = "helloworld"
var c : String = ""
print("plain =",p)
for e in p.unicodeScalars
{
    c += String(UnicodeScalar((e.value-97+4) % 26 + 97)!)
}
print("cipher=",c)

let dic = ["Apple":3.5,"Pear":2.7,"Banana":5.2]
for (k,v) in dic
{
    print(k,v,terminator: ", ")
}
print()

var t = 0.0
for (_,v) in dic
{
    t += v
```

```
84   }
85   print("Total price:",t)
```

程序段 3-11 的执行结果如图 3-5 所示。

```
10!= 3628800
2^10= 1024
1+3+...+99 =  2500
1+3+...+99 =  2500
1+3+...+99 =  2500
1+3+...+99 =  2500
plain = helloworld
cipher= lippsasvph
Apple 3.5, Banana 5.2, Pear 2.7,
Total price:11.40
Program ended with exit code: 0
All Output ⇕   Filter
```

图 3-5　程序段 3-11 的执行结果

下面结合图 3-5 介绍程序段 3-11 的执行过程。在程序段 3-11 中，第 2 行“var s=1”定义整型变量 s，赋初值为 1。第 3～6 行为一个 for-in 结构，第 3 行“for e in 1...10”表示 e 在 1～10 的 10 个整数上遍历，即 e 依次取值 1、2、……、10，循环执行第 5 行“s *= e”，这里为计算 10 的阶乘。第 7 行“print("10!=",s)”输出 10 的阶乘的值。

第 9 行“s=1”令变量 s 的值为 1。第 10～13 行为一个 for-in 结构，第 10 行“for _ in 1...10”中使用“_”作为循环变量，没有指定循环变量的名称，表示该循环只关注循环次数，这里循环 10 次，计算第 12 行“s *= 2”，即计算 2 的 10 次方。第 14 行“print("2^10=",s)”输出 2 的 10 次方的值。

第 16 行“s=0”将 0 赋给变量 s。第 17～20 行为一个 for-in 结构，第 17 行“for e in stride(from:1, to:100, by:2)”这里 e 遍历的序列为 1、3、5、……、99，对于每个遍历值执行第 19 行“s+=e”，即计算 100 以内的正奇数的和。第 21 行“print("1+3+...+99=",s)”输出字符串“1+3+...+99=”和 s 的值。这里循环变量 e 的作用范围为 for-in 结构，故在其他的 for-in 结构中仍然可以使用同名的局部变量 e。

第 23～32 行实现了与第 16～21 行相同的功能，这里为演示 continue 语句的用法。第 23 行“s=0”将 0 赋给变量 s。第 24～31 行为一个 for-in 结构，第 24 行“for e in stride(from:1, through:100, by:1)”表示循环变量 e 在范围 1、2、……、100 上遍历取值，对每个 e，执行第 26～30 行，第 26～29 行为一个 if 结构，第 26 行“if e % 2 == 0”如果 e 为偶数，则执行第 28 行“continue”，表示跳过 for-in 结构中 continue 后面的语句回到第 24 行执行条件判断，这里跳过第 30 行回到第 24 行，即忽略循环变量 e 为偶数的情况。第 32 行“print("1+3+...+99=",s)”输出字符串“1+3+...+99=”和 s 的值。

第 34～47 行实现了与第 16～21 行相同的功能，即计算 100 以内正奇数的和，这里为了演示 break 语句的用法。第 41～44 行为一个 if 结构，如果第 41 行“if e>=100”的条件为真，则执行第 43 行 break 语句，跳出它所在的 for-in 结构，这里跳转到第 47 行执行。

现在总结一下 continue 和 break 的特点：①continue 和 break 均可以用于 if、switch、for-in 和 while 结构中；②在循环体中遇到 continue，将跳过循环体内 continue 之后的语句，转到循环开头执行条件判断，开始下一次循环；③在循环体内遇到 break 语句，将跳出该循环体，转到循环体外的下一条语句执行；④对于嵌套的循环体而言，continue 和 break 语句仅对它所在的循环体有效，当有多个嵌套的循环体时，为了明确 continue 和 break 作用的循环体，可以为它们所在的循环体开头添加标号，指示 continue 和 break 的跳转，这个仅是为了方便程序阅读，第 49～62 行中演示了这种用法。

第 49～62 行实现了与第 16～21 行相同的功能。第 50～61 行为一个 for-in 结构，第 50 行“mylabel:for e in 1...100”在 for 循环体头部添加了标号 mylabel，第 54 行 continue mylabel 表示 continue 执行时要跳转的标号为 mylabel；第 58 行 break mylabel 表示 break 要执行时跳出的循环体为 mylabel 指示的循环体。

第 64 行“let p="helloworld""定义常量 p，赋值为“helloworld”。第 65 行“var c:String="""定义字符串变量 c，赋值为空串。第 66 行“print("plain=",p)”输出字符串“plain=”和字符串 p 的值。第 67～70 行为一个 for-in 结构，这里实现了凯撒密码，即一个字符用其后的第 4 个字符替换，第 67 行“for e in p.unicodeScalars”表示循环变量 e 在 p 的 Unicode 码形式的字符串中遍历各个字符，对于每个 e，执行第 69 行“c+=String(UnicodeScalar((e.value-97+4)%26+97)!)”，这里“e.value”返回 e 的 ASCII 值，97 为字符 a 的 ASCII 值，UnicodeScalar 函数返回以整数值参数作为 ASCII 值对应的字符，这里将每个 e 变换后的字符转换为字符串添加到 c 的尾部。第 71 行“print("cipher=",c)”，输出加密后的密文文本 c。第 69 行中的 97 可以使用“UnicodeScalar("a").value”替换，可避免去查字符 a 的 ASCII 值，这里，“UnicodeScalar("a").value”返回字符 a 的 ASCII 值。

第 73 行“let dic=["Apple":3.5,"Pear":2.7,"Banana":5.2]”定义字典 dic。第 74～77 行为一个 for-in 结构，第 74 行“for (k,v) in dic”中，元组(k,v)遍历 dic 字典的每个元素，其中，k 存储遍历到的字典元素的键值，v 存储遍历到的字典元素的值，对于每个元组(k,v)，执行第 76 行“print(k,v,terminator:",")”，输出遍历到的键值对。第 78 行“print()”输出一个空行。

第 80 行“var t=0.0”定义双精度浮点型变量 t，赋值为 0.0。第 81～84 行为一个 for-in 结构，第 81 行“for(_,v) in dic”中，只关注从字典 dic 中遍历到的元素的值，将这个值赋给 v，对于每次遍历，执行第 83 行“t+=v”，这里表示计算字典中所有水果的单价和。第 85 行“print("Total price:",t)”输出字符串“Total price:”和 t 的值。

上述介绍的 for-in 结构的特点在于循环变量的取值范围是确定的，可以准确地推断出循环的次数。而当循环次数不可知时，一般不使用 for-in 结构，而使用 while 循环控制结构。while 结构分为标准 while 结构和 repeat-while 结构两种，其语法如下。

(1) 标准 while 结构，即当型 while 结构。

```
while 条件表达式
{
    条件表达式为真时执行的语句组
}
```

在上述结构中，当“条件表达式”为真时，循环执行“条件表达式为真时执行的语句组”，直到“条件表达式”为假时跳出 while 循环体。在“条件表达式为真时执行的语句组”中，一般包含有调整“条件表达式”结果的语句。在标准 while 结构中，如果“条件表达式”为假，则“条件表达式为真时执行的语句组”可能一次也得不到执行。

(2) repeat-while 结构，即直到型 while 结构。

```
repeat
{
    语句组
}while 条件表达式
```

在上述结构中，先执行一次“语句组”，再判断“条件表达式”的值，若其为真，则循环执行“语句组”，直到“条件表达式”为假，跳出 repeat-while 循环体。在 repeat-while 结构中，表示循环体的“语句组”至少可以被执行一次。

程序段 3-12 介绍了 while 结构的用法，实现了欧几里得求两个整数的最大公约数和最小公倍数的算法。欧几里得算法的基本原理：设 a 和 b 为两个整数，不妨设 a>b，则 gcd(a,b)=gcd(b, a mod b)，即 a 与 b 的最大公约数等于 b 与 a 模 b 的最大公约数，通过反复求模运算，

最后可以表示为 gcd(r,0),则 r 为 a 与 b 的最大公约数,最小公倍数为 a * b/r。现在发现这个 2000 多年前的算法是求两数的最大公约数最快的方法。


视频讲解

程序段 3-12 while 结构用法实例

```
1    import Foundation
2
3    var a0 = 6525
4    var b0 = 81450
5    var a,b : Int
6    if a0<b0
7    {
8        (a0,b0)=(b0,a0)
9    }
10   (a,b)=(a0,b0)
11   var r = 1
12   while r>0
13   {
14       r = a % b
15       a = b
16       b = r
17   }
18   print("gcd(\(a0),\(b0))=\(a)")
19   print("lcm(\(a0),\(b0))=\(a0*b0/a)")
20
21   (a,b)=(a0,b0)
22   repeat
23   {
24       r = a % b
25       a = b
26       b = r
27   }while(r>0)
28   print("gcd(\(a0),\(b0))=\(a)")
29   print("lcm(\(a0),\(b0))=\(a0*b0/a)")
30
31   (a,b)=(a0,b0)
32   while true
33   {
34       r = a % b
35       a = b
36       b = r
37       if r==0
38       {
39           break
40       }
41   }
42   print("gcd(\(a0),\(b0))=\(a)")
43   print("lcm(\(a0),\(b0))=\(a0*b0/a)")
```

程序段 3-12 的执行结果如图 3-6 所示。

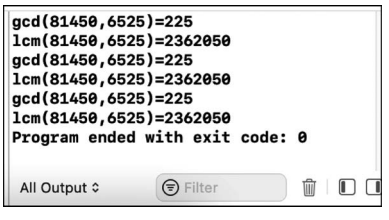

图 3-6 程序段 3-12 的执行结果

下面结合图 3-6 介绍程序段 3-12 的执行过程。在程序段 3-12 中,第 3 行“var a0＝6525”定义整型变量 a0,赋初值为 6525。第 4 行“var b0＝81450”定义整型变量 b0,赋初值为 81450。第 5 行“var a,b:Int”定义整型变量 a 和 b。

第 6～9 行为一个 if 结构，第 6 行“if a0＜b0”若 a0 小于 b0，则执行第 8 行“(a0,b0)＝(b0,a0)”将 a0 和 b0 的值对换。第 10 行“(a,b)＝(a0,b0)”将 a0 赋给 a，将 b0 赋给 b。

第 11 行“var r＝1”定义整型变量 r，赋初值为 1。第 12～17 行为一个 while 结构，实现欧几里得算法，第 12 行“while r＞0”当 r 大于 0 时，循环执行第 14～16 行，这里的 r 保存 a 除以 b 的余数，如第 14 行“r＝a ％ b”所示，然后，第 15 行“a＝b”将 b 赋给 a，第 16 行“b＝r”将 r 赋给 b，进行下一次循环，直到余数 r 为 0。第 18 行“print("gcd(\(a0),\(b0))＝\(a)")”输出 a0 和 b0 的最大公约数，第 19 行“print("lcm(\(a0),\(b0))＝\(a0＊b0/a)")”输出 a0 和 b0 的最小公倍数。

第 21～29 行实现了与第 10～19 行相同的功能，这里使用了 repeat-while 结构。第 21 行“(a,b)＝(a0,b0)”将 a0 赋给 a，将 b0 赋给 b。第 22～27 行为一个 repeat-while 结构，循环执行第 24～26 行直到 r 小于或等于 0。第 28、29 行依次输出 a0 和 b0 的最大公约数和最小公倍数。

第 31～43 行实现了与第 10～19 行相同的功能。这里使用了 while 结构，第 31 行“(a,b)＝(a0,b0)”将 a0 赋给 a，将 b0 赋给 b；第 32 行“while true”表示这是一个无限循环，循环执行第 34～40 行，在无限循环体中，添加了第 37～40 行的一个 if 结构，判断 r 的值是否为 0(第 37 行“if r＝＝0”)，如果 r 为 0，则执行第 39 行 break，跳出 while 循环体，跳至第 42 行执行，第 42 行“print("gcd(\(a0),\(b0))＝\(a)")”输出 a0 和 b0 的最大公约数；第 43 行“print("lcm(\(a0),\(b0))＝\(a0＊b0/a)")”输出 a0 和 b0 的最小公倍数。

3.7　本章小结

本章详细介绍了 Swift 语言的运算符，讨论了运算符的优先级以及常用运算符的用法，重点阐述了算术运算符、关系运算符、条件运算符、逻辑运算符、位运算符、区间运算符、赋值与复合赋值运算符等的用法。本章还详细介绍了程序流程控制方式，即顺序执行、分支执行和循环执行方式，重点讲述了 if 结构、if-else 结构、switch 结构、for-in 结构和 while 结构等，借助本章的内容可以编写实现任意计算机数值算法的代码。

习题

1. Swift 语言常用的运算符有哪些？讨论这些运算符的优先级情况。

2. 简述 Swift 语言的位运算符的用法。

3. 编程实现 3n＋1 猜想，即输入一个整数，如果该整数为奇数，则将其乘以 3 加上 1；如果其为偶数，则将其除以 2，如此循环下去，必能得到 1。

4. 编程输入一个整数 n，输出该整数的二进制形式。

5. 编程求一元二次方程 $ax^2+bx+c=0$ 的根。

6. 编程计算两个正整数的最大公约数和最小公倍数。

7. 输入一个正整数 n，求它的素数因子。例如，n＝80，则其素数因子为“2,2,2,2,5”。