第 3 章　语句和表达式

程序是由一个或者多个语句构成的，而程序的最小执行单位也是语句，所以语句在编程中占据了很重要的位置。在 Swift 中，最常见到的语句是由表达式构成的语句。本章将详细讲解常见表达式和语句的使用。

3.1　语　　句

在 Swift 语言中，语句的构成是很简单的，如下：

```
表达式
```

或者是：

```
表达式;
```

一般建议使用第一种形式。在 Swift 语言中有两种语句：一种是简单语句；一种是控制流语句。简单语句最为常见，它由表达式或者声明组成。控制流语句用于控制程序中语句的执行顺序。Swift 有 3 种类型的控制流语句：循环语句、分支语句和控制转移语句。这部分内容在后面章节具体讲解。下面首先讲解简单语句的重要构成部分——表达式。

3.2　运算符与表达式

Swift 语言提供了丰富的运算符。它们是一些特殊的符号或短语，用来查、改或组合值。程序会针对一个或者多个运算符进行运算。而由这些运算符构成的式子被称为表达式。本节将为读者讲解 Swift 语言常用的运算符以及运算符所对应的表达式。

3.2.1　常用术语——元

元表示运算符所使用的目标数值个数（即操作数）。根据数值个数的不同，运算符分为一元运算符、二元运算符、三元运算符。它们的详细说明如表 3-1 所示。

表 3-1　常用运算符

运　算　符	说　　明
一元运算符	它对一个目标进行操作。一元运算符分为一元前缀运算符和一元后缀运算符。其中，一元前缀运算符出现在目标的前面（例如-b）；一元后缀运算符出现在目标的后面（例如 b--）。

续表

运　算　符	说　　明
二元运算符	它对两个目标进行操作，并且是中缀（即在两个操作数之间）
三元运算符	它对三个目标进行操作。与 C 语言一样，Swift 也只有一个三元运算符，即三元跳转运算符（a?b:c）

3.2.2　赋值运算符和表达式

赋值运算符一般使用“=”表示，由“=”号连接起来的式子被称为赋值表达式。它的功能就是计算右边表达式的值，再赋予左边的变量。其语法形式如下：

```
变量=表达式
```

其中，表达式可以是一个变量、常量，也可以是一个表达式。在第 2 章中，所有的示例都使用到了赋值运算符。注意：如果右边分配的值具有多个值的元组，其元组中的元素可以被分配给多个常量和变量。代码如下：

```
let (x,y)=(5,3)
println(x,y)
```

运行结果如下：

```
(5, 3)
Program ended with exit code: 0
```

3.2.3　算术运算符和表达式

算术运算是基本的运算方式。Swift 针对所有的数字类型支持 4 种标准算术运算符，如表 3-2 所示。

表 3-2　标准算术运算符

运算符名称	运算符	功　　能
加法运算符	+	将两个数相加
减法运算符	–	将两个数相减
乘法运算符	*	将两个数相乘
除法运算符	/	将两个数相除

注意：不同于 C 和 Objective-C，默认情况下 Swift 的算术运算符不允许值溢出。并且算术运算符属于从左到右的结合性。使用标准算术运算符连接起来的式子被称为算术表达式，其语法形式如下：

```
操作数　算术运算符　操作数
```

【示例 3-1】下面就使用 4 种算术运算符实现两个操作数的加、减、乘、除，并将结果输出。代码如下：

```
import Foundation
var a=10
var b=5
```

```
var c=a+b                                    //加法运算
var d=a-b                                    //减少运算
var e=a*b                                    //乘法运算
var f=a/b                                    //除法运算
println(c)
println(d)
println(e)
println(f)
```

运行结果如下所示：

```
15
5
50
2
Program ended with exit code: 0
```

算术运算的处理有以下 6 个规律，需要遵循。

（1）加法运算符对于字符串也一样适用。产生的作用为连接字符串，代码如下：

```
import Foundation
var str1="Hello"
var str2="World"
var a=str1+str2
println(a)
```

运行结果如下所示：

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

（2）在乘法中，当两个操作数都为正数时，所得的结果也为正数；当两个操作数都为负数时，所得的结果也为正数；当两个操作数其中有一个为正数，一个为负数时，所得的结果就为负数。代码如下：

```
import Foundation
var a=10
var b=(-5)
var c=a*a
var d=b*b
var e=a*b
println(c)
println(d)
println(e)
```

运行结果如下所示：

```
100
25
-50
Program ended with exit code: 0
```

（3）在除法中，当两个操作数都为正数时，所得的结果也为正数；当两个操作数都为负数时，所得的结果也为正数；当两个操作数其中有一个为正数，一个为负数时，所得的结果就为负数。代码如下：

```
import Foundation
var a=10
```

```
var b=(-5)
var c=a/a
var d=b/b
var e=a/b
println(c)
println(d)
println(e)
```

运行结果如下所示：

```
1
1
-2
Program ended with exit code: 0
```

（4）在进行除法运算时，当两个操作数都为整数时，所得的结果也为整数，即发生了整除运算；当两个操作数有一个为浮点数时，所得的结果也为浮点数。代码如下：

```
import Foundation
var a1=9/4
println(a1)
var a2=9.00/(-4)
println(a2)
```

运行结果如下：

```
2
-2.25
Program ended with exit code: 0
```

注意：9/4 的结果为 2，这相当于算术运算的商。

（5）在进行除法运算时，除数不可以为 0，否则就会出现错误，如：

```
import Foundation
var x=0
var y=10/0
println(y)
```

由于除数为 0，导致程序出现以下的错误信息：

```
Division by zero
```

（6）可以将多个算术运算符组合起来使用，代码如下：

```
import Foundation
let a=10+2*3/4%2
println(a)
```

运行结果如下：

```
11
Program ended with exit code: 0
```

注意：当有多个算术运算符时，需要注意它的运算优先级别，其中*、/的优先级最高，其次是%，最后是+、–。所以在计算上面代码中的算术表达式时，首先需要计算 2*3，结果为 6；其次计算 6/4，结果为 1；然后计算 1%2，结果为 1；最后计算 10+1，其结果为 11。

3.2.4　取余运算符和表达式

取余运算符（%）在其他语言中被称为模数运算符。使用取余运算符连接起来的式子被称为取余表达式，其语法形式如下：

```
操作数 % 操作数
```

【示例 3-2】下面就使用取余运算符%计算 9%4 的余数，并输出结果。代码如下：

```
import Foundation
var a=9%4
println(a)
```

运行结果如下所示：

```
1
Program ended with exit code: 0
```

取余运算有以下 3 条规则需要注意。

（1）余数的符号规则。在进行取余操作时，当两个操作数为正数时，所得的结果也为正数；当两个操作数都为负数时，所得的结果也为负数；当被除数为负数，除数为正数时，所得的结果就为负数；当被除数为正数，除数为负数时，所得的结果就为正数；代码如下：

```
import Foundation
var a1=9%4
println(a1)
var a2=(-9)%(-4)
println(a2)
var a3=(-9)%(4)
println(a3)
var a4=(9)%(-4)
println(a4)
```

运行结果如下所示：

```
1
-1
-1
1
Program ended with exit code: 0
```

（2）在进行取余操作时，两个操作数除了可以是整数外，还可以是浮点数。这一点是 Swift 特有的特性。代码如下：

```
import Foundation
var a=8%2.5
println(a)
```

运行结果如下所示：

```
0.5
Program ended with exit code: 0
```

（3）在进行取余运算时，除数不可以为 0，否则就会出现错误，如：

```
import Foundation
```

```
var x=0
var y=10%0
println(y)
```

由于除数为 0，导致程序出现以下的错误信息：

```
Division by zero
```

3.2.5　自增自减运算符和表达式

在很多语言中都提供了自增自减运算符，Swift 语言也不例外。它是作为增加或减少一个变量的值的快捷方式。使用自增自减运算符连接起来的式子被称为自增自减表达式。

1．自增运算符与表达式

自增运算符（++）的作用是使变量的值自增 1。自增运算符可以分为两种：一种是前缀自增运算符；一种是后缀自增运算符。使用自增运算符连接起来的式子被称为自增表达式。自增表达式也分为了两种：一种是前缀自增表达式，一种是后缀自增表达式。

（1）前缀自增表达式的语法形式如下：

```
++运算分量
```

【示例 3-3】下面使用前缀自增运算符实现自加 1，并输出。代码如下：

```
import Foundation
var a=0;
println(a)
println(++a)
```

运行结果如下：

```
0
1
Program ended with exit code: 0
```

注意：在本示例中只输出 a 的结果，就为 0，如果要输出++a 的结果，需要考虑它的功能，即在返回其值以前自动加 1，所以结果就为 1。

（2）后缀自增表达式的语法形式如下：

```
运算分量++
```

读者可以将其看作是 a=a+1 简写为 a++的形式。

【示例 3-4】下面就使用后缀自增运算符实现自加 1，并输出。代码如下：

```
import Foundation
var a=0
println(a)
println(a++)
println(a)
```

运行结果如下所示：

```
0
0
```

```
1
Program ended with exit code: 0
```

在本示例中只输出 a 的结果，就为 0，如果要输出 a++的结果，需要考虑它的功能，即在返回其值以后自动加 1，所以首先返回现在 a 的值，即执行 a++时输出 0，然后再为 a 自动加 1，所以再一次输出 a 时，结果就为 1。

前缀自增运算符与后缀自增运算符总结：

- 如果运算符写在变量之前，它在返回其值之前对变量值加 1。
- 如果运算符写在变量之后，它在返回其值之后对变量值加 1。

2．自减运算符

自减运算符（--）的作用是使变量的值自减 1。自减运算符和自增运算符一样，也分为了两种：一种是前缀自减运算符；一种是后缀自减运算符。使用自减运算符连接起来的式子被称为自减表达式。其中表达式也被分为了两种：一种是前缀自减表达式，一种是后缀自减表达式。

（1）前缀自减表达式的语法形式如下：

```
--运算分量
```

【示例 3-5】下面就是使用前缀自减运算符实现自减 1，并输出。代码如下：

```
import Foundation
var a=2;
println(a)
println(--a)
```

运行结果如下所示：

```
2
1
Program ended with exit code: 0
```

在本示例中只输出 a 的结果，就为 2，如果要输出--a 的结果，需要考虑它的功能，即在返回其值以前自动减 1，所以结果就为 1。

（2）后缀自减表达式的语法形式如下：

```
运算分量--
```

读者可以将其看作是 a=a–1 简写为 a--的形式。

【示例 3-6】下面就是使用后缀自减运算符实现自减 1 的功能，并输出。代码如下：

```
import Foundation
var a=2;
println(a)
println(a--)
println(a)
```

运行结果如下所示：

```
2
2
1
Program ended with exit code: 0
```

在本示例中只输出 a 的结果，就为 2，如果要输出 a--的结果，需要考虑它的功能，即在返回其值以后自动减 1。所以，首先返回现在 a 的值，即执行 a--时输出 2，然后再为 a 自减 1，所以再一次输出 a 时，结果就为 1。

3.2.6　一元减运算符

在一个操作数之前加一个“–”号，此“–”号就被叫作一元减运算符。它的作用是将正数变为负数，将负数变为正数。由一元减运算符连接起来的式子被称为一元减表达式。其语法形式如下：

```
-操作数;
```

【示例 3-7】下面使用一元减运算符将数值 3 变为负数，再将变为负数的 3 变为正数，并输出。代码如下：

```
import Foundation
let three = 3
let minusThree = -three
let plusThree = -minusThree
println(minusThree)
println(plusThree)
```

运行结果如下所示：

```
-3
3
Program ended with exit code: 0
```

3.2.7　一元加运算符

在操作数以前加一个“+”号，此“+”号就被叫作一元加运算符，它基本上没有什么作用。只是为了对齐代码，尤其是使用一元减运算符时。由一元加运算符连接起来的式子被称为一元加表达式。其语法形式如下：

```
+操作数;
```

【示例 3-8】下面是使用一元加运算符的示例。代码如下：

```
import Foundation
let minusSix = -6
let alsoMinusSix = +minusSix
println(alsoMinusSix)
```

运行结果如下所示：

```
-6
Program ended with exit code: 0
```

3.2.8　位运算符

内存存储数据的基本单位为字节，一个字节由 8 个位组成。在二进制系统中，每个 0

或者 1 就是一个位，也称为比特位。位运算就是对二进制进行的运算。在 Swift 中有专门对位运算使用的运算符——位运算符。位运算符通常在诸如图像处理和创建设备驱动等底层开发中使用。使用它可以单独操作数据结构中原始数据的比特位。在使用一个自定义的协议进行通信的时候，运用位运算符来对原始数据进行编码和解码是非常有效的。位运算符的总结如表 3-3 所示。

表 3-3　位运算符

位运算符符号	位运算符名称	作　　用
&	按位与	两个相应的二进制位都为 1，则该位为 1，否则为 0
|	按位或	两个相应的二进制位中只有一个为 1，则该位为 1
^	按位异或	两个相应的二进制位值相同则为 0，否则为 1
~	取反	将二进制数按位取反，即 0 变 1，1 变 0
<<	左移	将一个数的各二进制位全部左移 N 位，右补 0
>>	右移	将一个数的各二进制位全部右移 N 位，对于无符号位，高位补 0

对位运算符中的&、|、^、~的作用进行分析，如图 3.1 所示。

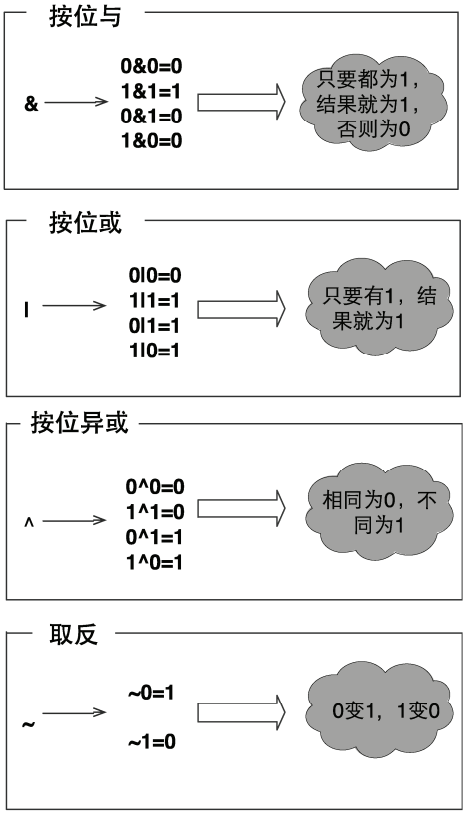

图 3.1　位运算的说明

【示例 3-9】以下就是使用&对 0b11111100 和 0b00111111 进行按位与运算并输出。代码如下：

```
import Foundation
var a:UInt8=0b11111100
```

```
var b:UInt8=0b00111111
var c=a&b
println(c)
```

运行结果如下所示：

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

它的工作方式如图 3.2 所示。

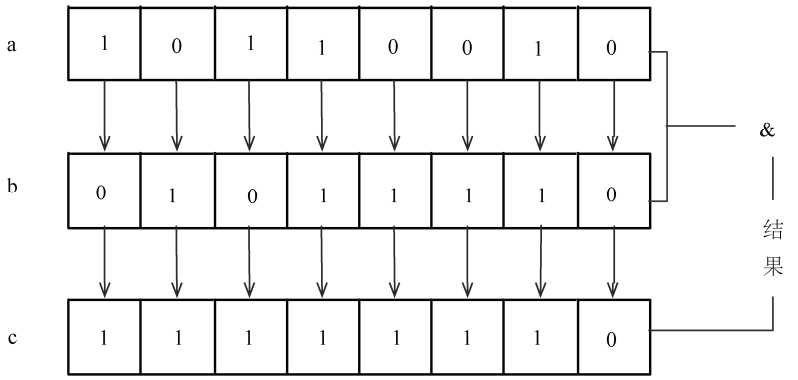

图 3.2　按位与工作方式

【示例 3-10】以下就是使用|对 0b10110010 和 0b01011110 进行按位或运算并输出。代码如下：

```
import Foundation
var a:UInt8=0b10110010
var b:UInt8=0b01011110
var c=a|b
println(c)
```

运行结果如下所示：

```
254
Program ended with exit code: 0
```

它的工作方式如图 3.3 所示。

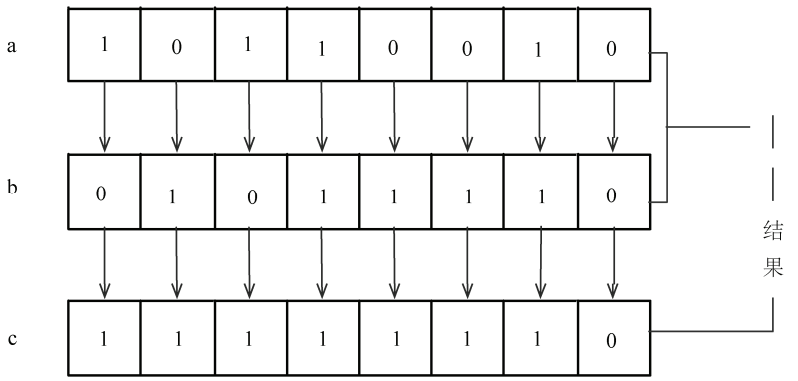

图 3.3　按位或工作方式

【示例 3-11】以下就是使用^对 0b00010100 和 0b00000101 进行按位异或运算并输出。代码如下：

```
import Foundation
var a:UInt8=0b00010100
var b:UInt8=0b00000101
var c=a^b
println(c)
```

运行结果如下所示：

```
17
Program ended with exit code: 0
```

它的工作方式如图 3.4 所示。

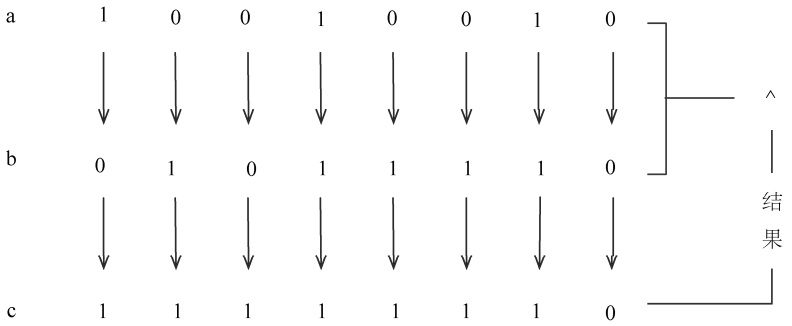

图 3.4　按位异或工作方式

【示例 3-12】以下就是使用~对 0b10110010 进行按位取反运算并输出。代码如下：

```
import Foundation
var a:UInt8=0b10110010
var b = ~a
println(b)
```

运行结果如下所示：

```
77
Program ended with exit code: 0
```

它的工作方式如图 3.5 所示。

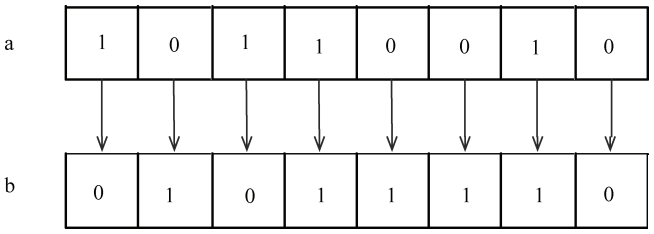

图 3.5　按位取反工作方式

1．左移运算符

左移运算符（<<）会将一个数字的各比特位按一定的位数向左移动。其中，左移分为

了无符号整型左移和有符号整型左移两种。下面就详细讲解这两种左移方式。

（1）无符号整型左移

无符号整型左移是将一个数字的比特位向左移动指定的位置，其中左边被移出整型存储边界的位数直接抛弃，右边空白的位用 0 填补。图 3.6 展示了无符号整型 00001111 左移 4 位的过程。

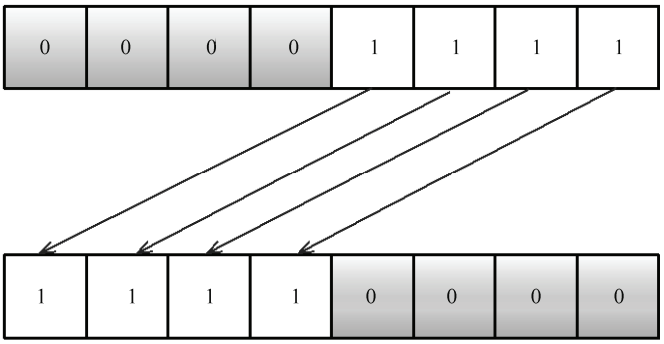

图 3.6　左移

其中，第一行中的前 4 位被移出，第二行中的后 4 位是补的空位。

【示例 3-13】以下将通过<<将 15 向左移动 4 位，代码如下：

```
import Foundation
let a=15
let b=a<<4                                  //左移
println("\(a)向左移动 4 位后变为了\(b)")
```

对与此代码的分析过程可以参考图 3.6，运行结果如下所示：

```
15 向左移动 4 位后变为了 240
Program ended with exit code: 0
```

注意：左移的效果相当于将一个整数乘以一个因子为 2 的整数。向左移动一个整数的比特位相当于将这个数乘以 2。以下的运算就是将整数 15 的比特位向左移动 4 位。

```
2*2*2*2*15=240
```

（2）有符号整型左移

有符号整型通过比特位的第一位来表示的，如果第一位为 0 表示正数，如果为 1 表示负数。其余的比特位（称为数值位）用来存储实值。有符号正数和无符号正数在计算机里存储的结果是一样的，如图 3.7 所示是+4 的二进制存储。

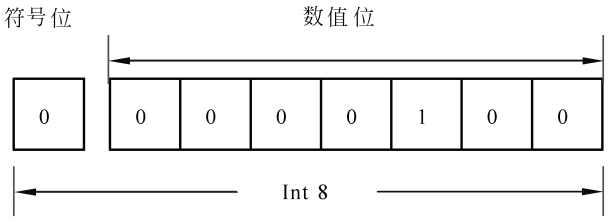

图 3.7　+4 的存储

负数的存储（二进制存储）就有一些复杂了，这里需要有一个运算，即 2 的 n 次方减去负数的绝对值，其中 n 为数值位的位数。如–4，它在 Int8 中有 7 位数值位，所以它的二进制存储为：

```
2^7-4=124
```

然后再将 124 转为二进制数，就是–4 的二进制存储，如图 3.8 所示。

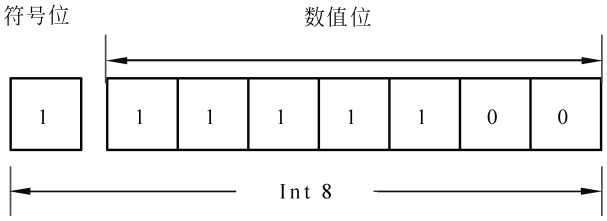

图 3.8　–4 的存储

除了可以使用 2 的 n 次方减去负数的绝对值，计算负数的二进制存储外，还可以使用取反加 1 的方法（即补码）实现负数的二进制的存放，还是以–4 为例，首先，获取–4 的原码，即 00000100，然后进行取反：

```
00000100 取反 11111011
```

最后将取反的值加 1：

```
11111011+1 为 11111100
```

得到的 11111100 为–4 的二进制存储。了解了正数和负数的表示后，再来看有符号整型的左移：对于一个正数来说它的左移就是无符号整型的左移，对于负数来说，它也是一样的，即左移 1 位时乘以 2。

【示例 3-14】以下将–4 左移一位。代码如下：

```
import Foundation
var a=(-4)
var b=a<<1
println("向左移动 1 位后变为了\(b)")
```

运行结果如下所示：

```
向左移动 1 位后变为了-8
Program ended with exit code: 0
```

2．右移运算符

右移运算符（>>）会将一个数字的各比特位按一定的位数向右移动。其中，右移分为了无符号整型右移和有符号整型右移。以下就详细讲解这两种右移方式。

（1）无符号整型右移

无符号整型右移是将一个数字的比特位向右移动指定的位置，其中右边被移出整型存储边界的位数直接抛弃，左边空白的位用 0 填补。图 3.9 就展示了无符号整型 00001111 右移 2 位的过程。

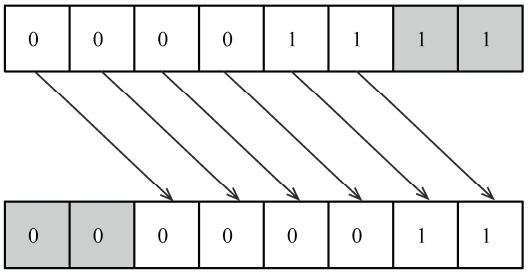

图 3.9　右移

其中，第一行中的后 2 位被移出，第二行中的前 2 位是补的空位。

【示例 3-15】以下将通过>>将 15 向右移动 2 位，代码如下：

```
import Foundation
let a=15
let b=a>>2                          //右移
println("\(a)向右移动 2 位后变为了\(b)")
```

运行结果如下：

```
15 向右移动 2 位后变为了 3
Program ended with exit code: 0
```

右移的效果相当于将一个整数除以一个因子为 2 的整数。向右移动一个整数的比特位相当于将这个数除以 2。如以下的运算就是将整数 15 的比特位向右移动 2 位。

```
15/2/2=3
```

（2）有符号整型右移

对于有符号整型右移来说，正数的右移和无符号整型的右移是一样的，但是对于负数来说是有分别的，它需要使用符号位去填充空白位。如图 3.10 是将–4 向右移动 1 位的操作。

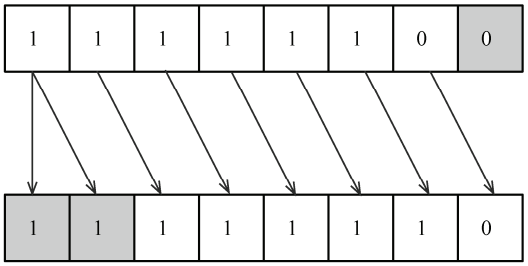

图 3.10　负符整型的右移

【示例 3-16】以下将–4 向右移动了 1 位。代码如下：

```
import Foundation
var a=(-4)
var b=a>>1
println("向右移动 1 位后变为了\(b)")
```

运行结果如下：

```
向右移动 1 位后变为了-2
Program ended with exit code: 0
```

3.2.9　溢出运算符

一般情况下，为一个整型变量/常量赋值时都不会超出它的承载范围。当超出时，Swift 也不会让程序通过，它会报错。如以下的代码：

```
import Foundation
var a=Int8.max
a=a+1
println(a)
```

a 的值为 127，这是 Int8 的最大范围，再为 a 加 1 就超出了 Int8 所能承受的范围，所以就会出现错误（这一点在数据类型中提到过）。如果开发者有意进行溢出操作，可以使用溢出运算符进行。在 Swift 中提供了 5 种针对整型的溢出运算符，如表 3-4 所示。

表 3-4　溢出运算符

溢出运算符	说明
&+	溢出加法
&–	溢出减法
&*	溢出乘法
&/	溢出除法
&%	溢出取余

1．值的上溢出

上溢出就是当一个值到达可以承载的最大值后，如果再一次进行加或者乘运算，就会导致新值的上溢出。

【示例 3-17】使用溢出加来实现 UInt8 在获取最大值后，实现加 1 的运算。代码如下：

```
import Foundation
var a=UInt8.max
a=a &+ 1
println(a)
```

UInt8 所能承载的最大值为 255（二进制 11111111），然后为这个最大值使用&+加 1，此时 UInt8 就无法表达这个新值的二进制了，就会导致溢出，如图 3.11 所示。

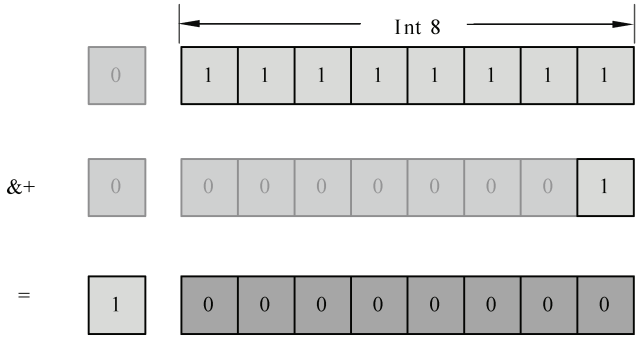

图 3.11　值的上溢出

在图 3.11 中可以看到，新值的承载范围内的那部分为 00000000，也就是 0。运行结果如下所示：

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

2．值的下溢出

下溢出因为数值太小而越界，当一个值到达可以承载的最小值后，如果再一次进行减运算，就会导致新值的下溢出。

【示例 3-18】使用溢出减来实现 UInt8 在获取最小值后，实现减 1 的运算。代码如下：

```
import Foundation
var a=UInt8.min
a=a &- 1
println(a)
```

UInt8 所能承载的最小值为 0（二进制 00000000），然后为这个最小值使用&–减 1，此时得到的二进制为 11111111（即 255），如图 3.12 所示。

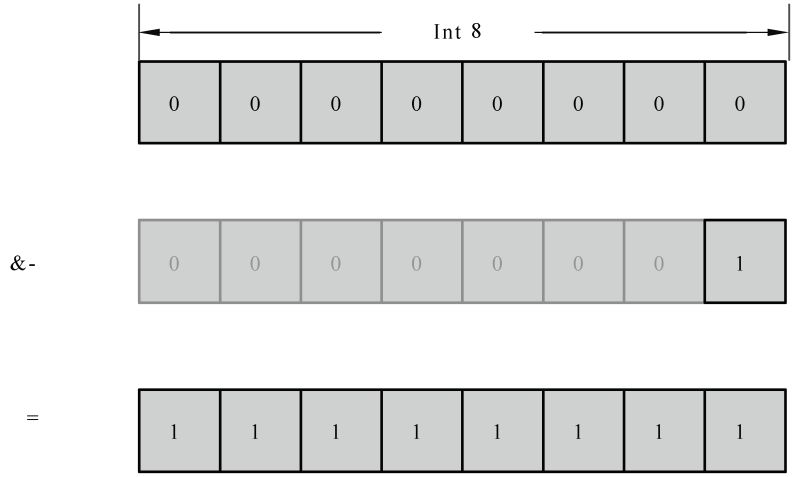

图 3.12　值的下溢出

运行结果如下所示：

```
255
Program ended with exit code: 0
```

【示例 3-19】使用溢出减来实现 Int8 在获取最小值后，实现减 1 的运算。代码如下：

```
import Foundation
var a=Int8.min
a=a &- 1
println(a)
```

有符号整型也有下溢出，Int8 所能承载的最小值为–128（二进制 10000000），然后为这个最小值使用&–减 1，此时得到的二进制为 01111111（即 127），如图 3.13 所示。

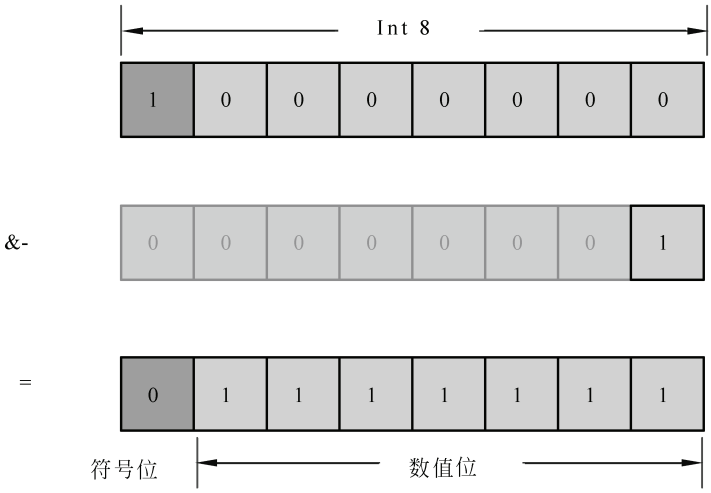

图 3.13　有符号型的下溢出

运行结果如下所示：

```
127
Program ended with exit code: 0
```

3．除0溢出

在进行除法运算时，要求除数不可以为 0，否则就会出现错误，使用溢出除法（&/可以进行除 0 的操作），代码如下：

```
import Foundation
var x=0
var y=10 &/ 0
println(y)
```

运行结果如下：

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

3.2.10　比较运算符和表达式

比较运算符是用来对两个操作数进行大小比较的。在 Swift 语言中，提供了 6 种比较运算符，这 6 种运算符的总结如表 3-5 所示。

表 3-5　比较运算符

运　算　符	运算符名称	功　　能	实　　例	结　　果
<	小于	若 a<b，结果为 true，否则为 false	2<3	true
<=	小于等于	若 a<=b，结果为 true，否则为 false	7<=3	false
>	大于	若 a>b，结果为 true，否则为 false	7>3	true
>=	大于等于	若 a>=b，结果为 true，否则为 false	3>=3	true
==	等于	若 a==b，结果为 true，否则为 false	7==3	false
!=	不等于	若 a!=b，结果为 true，否则为 false	7!=3	true

注意：比较运算符也是有优先级别的，其中最高的是>、>=、<、<=，其次是==、!=。

比较运算符通常用在条件语句中（对于条件语句会在后面的章节中作讲解）。使用比较运算符连接起来的式子被称为比较表达式。其语法形式如下：

```
表达式　比较运算符　表达式
```

注意：表达式可以是一个运算符，比较表达式返回的类型为 BOOl（布尔类型）。

【示例 3-20】下面就使用比较运算符>、<、!=对 10 和 5 进行比较，并输出。代码如下：

```
import Foundation
let a=10
let b=5
let c=a>b
println(c)
let d=a<b
println(d)
let e=a != b
println(e)
```

运行结果如下：

```
true
false
true
Program ended with exit code: 0
```

Swift 提供了两个恒等运算符（===和!==），用它来测试两个对象引用是否来自于同一个对象实例。（对于这两个运算符我们会在集合类型中作讲解。）

3.2.11　三元条件运算符

三元条件运算符（?:）是一种特殊的运算符，主要由 3 部分组成，它一般用于对条件的求值。使用三元条件运算符连接起来的式子被称为三元条件表达式。其语法形式以及执行流程如图 3.14 所示。

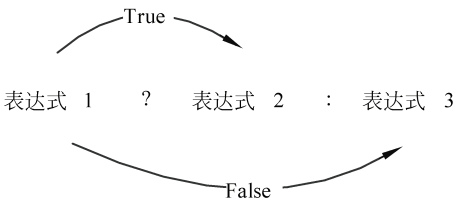

图 3.14　语法形式以及执行流程

在图 3.14 中，当表达式 1 的值为真时，结果为表达式 2 的结果；当表达式 1 的值为假时，结果为表达式 3 的结果。三元条件运算符其实是下面代码（关于 if else 语句会在后面的章节中做一个详细的讲解）的简化：

```
if 表达式 1 {
     表达式 2
```

```
} else {
     表达式 3
}
```

【示例 3-21】下面使用三元条件运算符实现比较 5 和 3 哪个较大，并输出较大的值。代码如下：

```
import Foundation
let max=5>3 ?5 : 3
println(max)
```

运行结果如下所示：

```
5
Program ended with exit code: 0
```

如果将以上代码使用 if else 语句编写，代码如下：

```
if 5>3{
   println("5")
}else{
   println("3")
}
```

运行结果如下所示：

```
5
Program ended with exit code: 0
```

从使用三元条件运算符编写的代码和使用 if else 语句编写的代码可以看出：三元条件运算符要比 if else 语句编写的代码简单。所以三元条件运算符提供了一个高效的写法。但是请小心使用三元条件运算符，如果过度使用会导致阅读代码更难。要避免多个实例的三元条件运算符组合成一个复合语句。

3.2.12　逻辑运算符和表达式

在一些编程中，一个语句往往需要满足多个条件才可以执行。这时就需要将这多个语句进行组合。逻辑运算符就可以把这多个语句进行组合，从而实现更复杂的语句。在 Swift 语言中包括 3 种逻辑运算符，如表 3-6 所示。

表 3-6　逻辑运算符

逻辑运算符	名　　称
&&	逻辑与
||	逻辑或
!	逻辑非

使用逻辑运算符连接起来的式子被称为逻辑表达式。其语法形式如下：

```
条件表达式　逻辑运算符　条件表达式
```

其中，逻辑表达式返回的值也是 Bool（布尔值）。下面就针对这 3 种逻辑运算符逐一讲解。

1．逻辑与（&&）

逻辑与运算符使用&&表示。使用逻辑与运算符关联起来的式子被称为逻辑与表达式，其语法形式如下：

```
条件表达式 1　&&　条件表达式 2
```

其中，只有当条件表达式 1 和条件表达式 2 的值都为 true 时，逻辑与表达式的值才为 true；如果条件表达式 1 或者条件表达式 2 中有一个值为 false 时，逻辑与表达式的值就为 false。

【示例 3-22】下面使用逻辑与运算符对两个表达式进行操作并输出结果。代码如下：

```
import Foundation
let a = 7<10 && 8>5
println(a)
let b = 7>10 && 8>5
println(b)
```

运行结果如下所示：

```
true
false
Program ended with exit code: 0
```

对于以上的代码，a 输出了 true，是由于逻辑与运算符关联的两个条件表达式的值都为真；b 输出 false，是由于逻辑与运算符关联的两个条件表达式，其中有一个（7>10）为假。

2．逻辑或（||）

逻辑或运算符使用||表示，其使用逻辑或运算符关联起来的式子被称为逻辑或表达式，其语法形式如下：

```
条件表达式 1　||　条件表达式 2
```

其中，只要当条件表达式 1 或者条件表达式 2 其中一个值为 true 时，逻辑或表达式的值就为 true。只有当条件表达式 1 和 2 的值都为 false 时，逻辑或表达式的值才为 false。

【示例 3-23】下面使用逻辑或运算符对两个表达式进行操作并输出结果。代码如下：

```
import Foundation
let a = 7<10 || 8>5
println(a)
let b = 7>10 || 8>5
println(b)
let c = 7>10 || 8<5
println(c)
```

运行结果如下所示：

```
true
true
false
Program ended with exit code: 0
```

对于以上的代码，a 输出了 true，是由于逻辑或运算符关联的两个条件表达式都为真；b 输出 true，是由于逻辑或运算符关联的两个条件表达式，其中有一个（8>5）为真；c 输出了 false，是由于逻辑或运算符关联的两个条件表达式都为假。

3．逻辑非（!）

逻辑非运算符使用!表示，其使用逻辑非运算符关联起来的式子被称为逻辑非表达式，其语法形式如下：

```
!条件表达式
```

其中，当条件表达式的值为 true 时，逻辑非表达式的值就为 false。当条件表达式的值为 false 时，逻辑非表达式的值就为 true。

【示例 3-24】下面使用逻辑非运算符对两个表达式进行操作并输出结果。代码如下：

```
import Foundation
let a = !(7<10)
println(a)
let b = !(7>10)
println(b)
```

运行结果如下：

```
false
true
Program ended with exit code: 0
```

这里总结一下逻辑运算符的运算规则，如表 3-7 所示。

表 3-7　逻辑运算符运算规则表

a	b	!a	!b	a&&b	a||b
真（true）	真（true）	假（false）	假（false）	真（true）	真（true）
真（true）	假（false）	假（false）	真（true）	假（false）	真（true）
假(false）	真（true）	真（true）	假（false）	假（false）	真（true）
假(false）	假（false）	真（true）	真（true）	假（false）	假（false）

有时为了满足某种需求，需要将多个逻辑运算符组合，来创建更长的复合表达式，代码如下：

```
import Foundation
let a=7<10
let b=8<5
let c=8<9
let d = a&&b||c
println(d)
```

运行结果如下所示：

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

注意：在本示例中使用到了多个逻辑运算符，此时需要考虑逻辑运算符的运算优先级别。其中优先级最高的是!，其次是&&，最后是||。根据优先级的高低，首先计算 a&&b，

其返回的结果为假，再使用返回的值与 c 做逻辑或的操作，由于 c 的值为真，所以最后的结果就为真。

3.2.13　范围运算符

Swift 包含两个范围运算符，能快捷地表达一系列的值。

1．封闭范围运算符

封闭范围运算符定义了一个范围，使用封闭范围运算符连接起来的式子被称为封闭范围表达式。其语法形式如下：

```
(操作数 1...操作数 2)
操作数 1...操作数 2
```

注意：加括号和不加括号都是可以的。其中范围从操作数 1~操作数 2，并且包括操作数 1 和操作数 2。操作数 1 必须要小于操作数 2。

【示例 3-25】以下程序就是使用了封闭范围运算符将 1~5 的值输出。代码如下：

```
import Foundation
for index in (1...5) {
   println("\(index) times is \(index)")
}
```

运行结果如下所示：

```
1 times is 1
2 times is 2
3 times is 3
4 times is 4
5 times is 5
Program ended with exit code: 0
```

注意：(1…5)是封闭范围运算符，所以输出的值需要包括 1 和 5。

2．半封闭范围运算符

半封闭的范围运算符也是定义了一个范围。但是它包含第一个值，而不包含最终值。用半封闭范围运算符连接起来的式子被称为半封闭范围表达式。其语法形式如下：

```
(操作数 1..<操作数 2)
操作数 1..<操作数 2
```

其中，范围从操作数 1~操作数 2，但是只包括操作数 1，不包括操作数 2。加括号和不加括号都是可以的。

【示例 3-26】以下就使用半封闭范围运算符实现对数组元素的输出。代码如下：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
println(count)
for i in 0..<count {
```

```
    println("Person \(i ) is called \(names[i])")
}
```

运行结果如下所示：

```
4                                   //count 的值
Person 0 is called Anna
Person 1 is called Alex
Person 2 is called Brian
Person 3 is called Jack
Program ended with exit code: 0
```

在本示例中，数组包含 4 个项目，但是由于它是一个半封闭的范围，所以输出的项目的索引值可以到达 3。

3.2.14　复合赋值运算符和表达式

在多数语言中，都有复合赋值运算符，在 Swift 语言中也不例外。它是由赋值运算符和其他的一些运算符组合起来的。其中，复合赋值运算符的种类、使用方法以及功能如表 3-8 所示。

表 3-8　复合赋值运算符

符　　号	使 用 方 法	等 效 形 式	功　　能
*=	a*=b	a=a*b	乘后赋值
/=	a/=b	a=a/b	除后赋值
%=	a%=b	a=a%b	取余后赋值
+=	a+=b	a=a+b	加后赋值
–=	a–=b	a=a–b	减后赋值
<<=	a<<=b	a=a<<b	左移后赋值
>>=	a>>=b	a=a>>b	右移后赋值
&=	a&=b	a=a&b	按位与后赋值
^=	a^=b	a=a^b	按位异或后赋值
|=	a|=b	a=a|b	按位或后赋值

由这些复合赋值运算符连接起来的式子被称为复合赋值表达式。其语法形式如下：

```
变量　复合赋值运算符　表达式
```

【示例 3-27】以下程序就是使用+=、–=、*=、/=复合赋值运算符实现的运算。代码如下：

```
import Foundation
var a=100
a+=10                                   //a+10 赋值给 a 即 100+10 赋值给 a
println(a)                              //输出 110
a-=10                                   //a-10 赋值给 a 即 110-10 赋值给 a
println(a)                              //输出此时 a 的值 100
a*=10                                   //a*10 赋值给 a 即 100*10 赋值给 a
println(a)                              //输出此时 a 的值 1000
a/=2                                    //a/2 赋值给 a 即 1000/2 赋值给 a
println(a)                              //输出此时 a 的值 500
a%=3                                    //a%3 赋值给 a 即 500/2 赋值给 a
```

```
println(a)                                    //输出此时 a 的值 2
```

运行结果如下：

```
110
100
1000
500
2
Program ended with exit code: 0
```

以本示例中的 a+=10 为例，它其实实现了两个功能：一个功能是做加法运算（a+10），一个功能是做赋值运算（将 a+10 的结果赋值给 a）。需要注意，复合赋值运算符是不返回值的，例如：

```
var b=+=3
```

由于复合赋值运算符没有返回值，所以就会出现错误，其错误信息如下：

```
Consecutive statements on a line must be separated by ';'
Expected expression after unary operator
Type annotation missing in pattern
Use of unresolved identifier '=+='
```

3.2.15　求字节运算符和表达式

由于不同的计算机所支持的数据类型长度也是不一样的，所以就提供了一个用来计算数据类型所占的字节数的运算符——sizeof。由 sizeof 运算符连接起来的式子被称为求字节表达式。其语法形式如下：

```
sizeof(数据类型)
```

【示例 3-28】下面使用 sizeof 求字节运算符对整型、浮点型所占的字节数进行获取，并输出。代码如下：

```
import Foundation
let a=sizeof(Int)
println(a)
let b=sizeof(Float)
println(b)
```

运行结果如下所示：

```
8
4
Program ended with exit code: 0
```

3.2.16　强制解析

在将某一个变量或者常量的类型定义为可选类型后，它们所代表的值是不可以直接进行运算的，如以下的代码，其功能是实现两个操作数的减法运算。

```
import Foundation
var value:Int?=5
```

```
var a=value-3
println(a)
```

在此代码中，由于在变量 value 的数据类型定义成了可选类型，导致程序在进行减法运算时，出现以下的错误：

```
Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?
```

怎么将可选类型的值进行运算呢？这时就需要使用强制解析，强制解析就是使用一个“!”感叹号运算符。它的使用形式如下：

```
可选类型的变量名/常量名!
```

【示例 3-29】以下使用强制解析实现对可选类型 value 的加法运算，代码如下：

```
import Foundation
var value:Int?=10
var a=value!+10                                    //强制解析后实现加法运算
println(a)
```

在此代码中，由于 value 是一个可选类型，所以需要强制解析，将 value 的值解析出来实现运算。运行结果如下：

```
20
Program ended with exit code: 0
```

注意：只有可选类型的常量或者变量才需要强制解析。例如以下的代码实现的功能是两个操作数的乘法运算：

```
import Foundation
var value:Int=10
var a=value!*10
println(a)
```

在此代码中，由于 value 不是可选类型，但使用了强制解析，导致程序出现了以下的错误：

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

3.3　类 型 转 换

在编程中，经常会遇到数据类型不一样的问题。例如在进行赋值运算时，左边的数据类型和右边的数据类型不一致，这时就需要进行类型转换。在 Switf 中类型转换都是显式转换，不存在隐式转换。在本节中将会讲解两个关于数字的转换方式：整数的转换、整数和浮点数直接的转换。

3.3.1　整数的转换

在整数中类型分为了 8 种。当它们中的两种或者两种以上出现在一个表达式中时，就需要进行类型转换。其转换的语法形式如下：

```
整数的数据类型(整数类型的变量/常量)
```

其中，整数的数据类型有 UInt8、UInt16、UInt32、UInt64、Int8、Int16、Int32 和 Int64。

【示例 3-30】下面实现将 UInt8 类型的数据转换为 UInt16 类型的数据。代码如下：

```
import Foundation
let a:UInt8=2
let b:UInt16=2000
let c=UInt16(a)+b                                    //转换
println(c)
```

运行结果如下所示：

```
2002
Program ended with exit code: 0
```

在本示例中，a 是一个 UInt8 类型，b 是一个 UInt16 类型，所以它们之间是不可以直接进行加法运算的。如果直接进行运算，就会出现以下错误：

```
Cannot invoke '+' with an argument list of type '(UInt8, UInt16)'
```

这时需要对 UInt8 进行类型转换，将其转换为 UInt16 类型，才可以进行加法运算（注意：转换的时候必须是将范围小的向范围大的类型上转换）。

3.3.2　整数和浮点数的转换

整数除了在整数类型之间进行转换外，还可以和浮点数进行转换。首先来了解整数转换为浮点数，其语法形式如下：

```
浮点数类型(整数的变量/常量)
```

注意：浮点数类型有两种，一种是 Float，一种是 Double。

【示例 3-31】下面实现将整数转换为 Double 类型的浮点数。代码如下：

```
import Foundation
let a=3
let b=10.555555
let c=Double(a)+b                          //将整数转换为 Double 类型的数据
println(c)
let d:Float=10.22
let e=Float(a)+d                           //将整数转为 Float 类型的数据
println(e)
```

运行结果如下所示：

```
13.555555
13.22
Program ended with exit code: 0
```

整数和浮点数之间的转换是相互的，整数可以转换为浮点数，同样浮点数也可以转换为整数，其语法形式如下：

```
整数的数据类型(浮点数类型的变量/常量)
```

【示例 3-32】下面就实现将浮点数转换为整数。代码如下：

```
import Foundation
let Pi=3.14159
println(Pi)
let IntPi=Int(Pi)
println(IntPi)
```

运行结果如下所示：

```
3.14159
3
Program ended with exit code: 0
```