第 4 章

基本运算符与程序流程控制

上帝创造了整数，所有其余的数都是人造的。

—— 利奥波德·克罗内克

世界上所有的运算无外乎是由计算过程与结果两部分组成，无论这个结果是否符合预期目标。在编程中，运算由表达式表示，而量值和运算符共同构成了表达式。Swift 语言对运算符的支持可谓强大，其除了支持一些 C 语言与 Objective-C 语言中常用的运算符之外，还提供了一些十分有特点的运算符，例如空合并运算符、区间运算符等。除此之外，Swift 语言还支持对运算符进行重载与自定义操作，开发者可以根据自己的需要为系统的运算符提供新的运算方法，甚至自定义自己的运算符。

程序存在的意义就是帮助人们实现解题思路和进行重复性的计算，然而任何复杂问题的解决过程都不会是从上到下线性完成的，对程序流程的控制能力是编程语言强大的关键所在。Swift 语言中提供了强大的程序流程控制语句，无论是循环结构、选择结构还是跳转结构，开发者都可以十分方便地运用，并且 Swift 语言的语句设计也更加简洁与优美，通过本章的学习，读者将会更深刻地体会到这一点。

通过本章，你将学习到：

- 各种运算符的应用。
- 运算符的优先级与结合性。
- 使用 for-in 结构进行循环遍历。
- 使用 while 与 repeat-while 结构进行条件循环。
- 使用 if 与 if-else 结构进行选择判断。
- 使用 switch-case 结构进行多分支选择。
- 使用跳转语句灵活控制程序流程。

4.1　初识运算符

运算符的作用在于将量值或者表达式结合在一起进行计算。Swift 语言中的运算符按照其操作数的个数可以分为 3 类：

- 一元运算符：一元运算符作用于一个操作数，其可以出现在操作数的前面，例如正负运算符“+”“-”及逻辑非运算符“!”。
- 二元运算符：二元运算符作用于两个操作数之间，例如加减运算符“+”“-”等。
- 三元运算符：三元运算符作用于 3 个操作数之间，最经典的三元运算符为问号冒号运算符，其可以方便地实现简单的判断选择结构。

4.1.1　赋值运算符

赋值运算符应该是在编程中出现频率最高的运算符之一，在对任何量值进行赋值时，都需要使用到赋值运算符“=”。需要注意，“=”在 Swift 语言中是赋值运算符，并不是相等运算符，对于一些编程初学者，很容易将相等运算符“==”与赋值运算符“=”混淆使用。使用 Xcode 开发工具创建一个命名为 Operational 的 playground 文件，赋值运算符示例如下：

```
//字符串赋值
var str = "Hello, playground"
//整型赋值
var count = 5
//元组赋值
var group = (1,2,"12")
//Bool 赋值
var bol = true
```

提　示

赋值运算符用于值的传递，其结果是量值被赋了具体的值。相等运算符则用于比较操作，其会返回一个 Bool 类型的逻辑值。

4.1.2　基本算术运算符

基本算术运算符用于进行一些基本的数学运算，例如加、减、乘、除等。需要注意的是，Swift 语言从 2.2 版本之后删除了自增运算符“++”与自减运算符“--”，目前版本的 Swift 语言中不可以再使用这两个运算符。Swift 中支持的基本算术运算符示例如下：

```
//相加运算
1+2
//相减运算
```

```
2-1
//相乘运算
2*2
//相除运算
4/2
//取余运算
4%3
```

提 示

取余运算符必须在整数间进行运算时使用。

将赋值运算符与基本算术运算符结合使用可以组合成复合赋值运算符，复合赋值运算符可以在一个表达式中完成一项基本运算与赋值的复合操作，示例如下：

```
var tmp=1
//加赋值复合运算
tmp+=3  //tmp = tmp +3
//减赋值复合运算
tmp-=3  //tmp = tmp -3
//乘赋值复合运算
tmp*=3  //tmp = tmp *3
//除赋值复合运算
tmp/=3  //tmp = tmp /3
//取余赋值复合运算
tmp%=3  //tmp = tmp %3
```

除了上面提到的那些运算符，正负运算符和数学中的正负号作用类似，负运算符会改变数据的正负性，正运算符会保持数据的正负性。示例如下：

```
var a = 1
var b = -2
+b //-2
-a //-1
```

提 示

自增与自减运算符在 Swift 2.0 及之前版本可以使用，Swift 2.2 版本后，基于代码可读性与减少歧义的考虑，移除了这两个运算符。

4.1.3 基本逻辑运算符

基本算术运算符进行数学上的算术操作，基本逻辑运算符进行逻辑运算操作。可以简单理解为，逻辑运算即是生活中所定义的真与假。系统定义的基本逻辑运算符会返回一个 Bool 类型的逻辑值，因此，基本逻辑运算符组成的逻辑表达式在 if 判断语句中会经常用到。

Swift 中支持的基本逻辑运算符有逻辑与运算符“&&”、逻辑或运算符“||”、逻辑非运算符“!”三种。逻辑运算只在逻辑值（Bool 类型值）之间进行，与、或、非三种运算中前两者为二元运算符，需要有两个 Bool 类型的操作数。非运算符为一元运算符，需要一个 Bool 类型的操作数，

它们有如下特点：

- 与：两个操作数都为真，结果才为真，有一个操作数为假则结果为假。
- 或：两个操作数有一个为真则结果为真，两个操作数都为假则结果为假。
- 非：操作数为真则结果为假，操作数为假则结果为真。

示例如下：

```
var p1 = true
var p2 = false
//与运算 false
p1&&p2
//或运算 true
p1||p2
//非运算 false
!p1
```

提　示

与 Objective-C 语言不同，Swift 语言中逻辑运算的操作数必须为严格的 Bool 类型。

4.1.4　比较运算符

Swift 中的比较运算符用于两个操作数之间的比较运算，其会返回一个 Bool 类型的逻辑值。基本的比较运算符有等于比较运算符“==”、小于比较运算符“<”、大于比较运算符“>”、不等于比较运算符“!=”、小于等于比较运算符“<=”以及大于等于比较运算符“>=”。示例如下：

```
1==2 //等于比较          返回 false
1<2  //小于比较          返回 true
1>2  //大于比较          返回 false
1 != 2 //不等于比较        返回 true
1<=2  //小于等于比较      返回 true
1>=2  //大于等于比较      返回 false
```

对于 Swift 中元组的比较操作，读者需要注意：首先，要比较的元组中元素个数和对应位置的元素类型必须相同；其次，元组中每一个元素必须支持比较运算操作。示例代码如下：

```
var tp1 = (3,4,"5")
var tp2 = (2,6,"9")
var tp3 = ("1",4,5)
tp1<tp2            //将返回 false
```

在上面的代码中，元组实例 tp1 与元组实例 tp2 中元素个数和对应类型相同，且所有元素都支持比较运算操作，所以 tp1 与 tp2 可以进行比较运算。tp1 与 tp3 虽然元素个数相同，但是 tp1 的第 1 个元素为整型，tp3 的第一个元素为字符串类型，类型不同则不能进行比较运算。Swift 在进行元组间比较运算的时候会遵守这样一个原则：从第 1 个元素开始比较，如果比较出了结果，就不再进行后面元素的比较运算，直接返回结果 Bool 值，如果没有比较出结果，那么继续依次比较后面的元素，直到比较出结果为止。

4.1.5 条件运算符

条件运算符（三目运算符）是一种三元运算符，其可以简便实现代码中的条件选择逻辑。例如，如下代码为一个简单的条件选择语句示例：

```
var m = 3
var n = 6
if m>n {
   print("m>n")
}else{
   print("m<=n")
}
```

上面的代码对变量 m 和 n 进行了比较，并将比较结果进行打印，如果使用条件运算符，上面的逻辑可以简写成如下模样：

```
print(m>n ? "m>n":"m<=n")
```

上面的代码在语法上可以简化为如下格式：条件?成立的代码:不成立的代码。

可以看到，条件运算符（三目运算符）只使用了一句代码就完成了 if-else 结构需要多句代码才能完成的工作，其编写效率很高。我们来分解一下条件运算符的组成结构，首先条件运算符需要有 3 个操作数，其中第 1 个操作数必须为一个条件语句或者为一个 Bool 类型的值，第 2 个和第 3 个操作数可以是任意类型的值或者是一个有确定值的表达式，3 个操作数由问号“？”和冒号“：”进行分割，当问号前的操作数值为真时，条件运算符运算的结果为冒号前的操作数的值，当问号前的操作数值为假时，条件运算符运算的结果为冒号后的操作数的值。

4.2 Swift 语言中两种特殊的运算符

前边介绍过，Optional 可选值类型是 Swift 语言的一大特点。Swift 语言中的空合并运算符也是专门为 Optional 值类型所设计的。除了空合并运算符，Swift 语言中的区间运算符也十分强大易用，熟练使用这些运算符将极大地提高开发效率。

4.2.1 空合并运算符

可选值类型（Optional）是 Swift 语言的一个独特之处，空合并运算符就是针对可选类型而设计的运算符。首先来看一段示例代码：

```
var q:Int? = 8
var value:Int
if q != nil {
   value = q!
}else{
   value = 0
```

```
}
```

上面的示例就是一个简单的 if-else 的选择结构，利用 4.1.5 节介绍的条件运算符（三目运算符），可以将上面的代码简写成如下形式：

```
var q:Int? = 8
var value:Int
value = (q != nil) ? (q!) : 0
```

使用条件运算符改写后的代码简单很多。Swift 语言还提供了空合并运算符来更加简洁地处理这种 Optional 类型值的条件选择结构，空合并运算符由“??”表示。上面的代码可以改写成如下形式：

```
//空合并运算符
var q:Int? = 8
var value:Int
value = q ?? 0
```

使用空合并运算符改写后的代码更加简洁。空合并运算符“??”是一个二元运算符。其需要两个操作数，第一个操作数必须为一个 Optional 值，如果此 Optional 值不为 nil，则将其进行拆包操作，并作为空合并运算的运算结果。如果此 Optional 值为 nil，则会将第二个操作数作为空合并操作运算的结果返回。使用空合并操作符来处理有关 Optional 值的选择逻辑将十分方便。

4.2.2　区间运算符

在 C 语言中，关于范围概念的描述常常会使用一个表达式来表示，例如：

```
//表示大于 0 小于 10 的范围
index>0 && index<10
```

在 Objective-C 语言中提供了 NSRange 这样一个结构体来描述范围，虽然直观了许多，但开发者在使用时需要构造 NSRange 实例，使用起来就略显烦琐。Swift 中除了支持 Range 结构体来描述范围外，还提供了一个区间运算符来快捷直观地表示范围区间。示例如下：

```
//创建范围 >=0 且<=10 的闭区间
var range1 = 0...10
//创建范围>=0 且<10 的半开区间
var range2 = 0..<10
```

也可以通过“~=”运算符来检查某个数字是否包含于范围中，示例如下：

```
//8 是否在 range1 中
print(range1 ~= 8)  //输出 true
```

区间运算符最常见于 for-in 循环结构中，开发者常常会使用区间运算符来定义循环次数，示例如下：

```
//a...b 为闭区间写法
for index in 0...3 {
   print(index)
```

```
}
//a..<b 为左闭右开区间
for index in 0..<3 {
    print(index)
}
```

提 示

在 for-in 循环结构中，如果 in 关键字后面是一个集合，则变量 index 会自动获取集合中的元素；如果 in 关键字后面是一个范围，则 index 获取到的是从左向右依次遍历到的范围索引数。

4.3 循 环 结 构

在程序编写中，常常会有大量而重复的操作，而这也是计算机计算的优势所在，对于开发者来说，循环结构就是为执行大量而重复代码块诞生的。Swift 语言主要提供了 for-in 遍历、while 与 repeat-while 条件循环 3 种循环结构。

4.3.1 for-in 循环结构

读者对于 for-in 结构并不陌生，在前面章节中介绍的很多内容都提前使用了 for-in 结构进行演示。如果读者了解 C/Objective-C 语言，那么这里就要注意了，在 C/Objective-C 语言中也支持 for-in 这种循环结构，但是其被称为快速遍历，用它来进行的循环操作将是无序的。Swift 语言中的 for-in 结构则强大很多，既可以进行无序的循环遍历，也可以进行有序的循环遍历。

在 Swift 2.2 及以上版本中，在循环结构中做的最大更改就是删除了经典的 for()循环结构，许多 C/Objective-C 语言的开发者可能会不太习惯，事实上，Swift 语言中的 for-in 结构已经完全可以代替曾经的 for()循环，并且比 for()循环的实现更加简洁优美，同 for()循环一同移除的还有“++”和“--”运算符。

使用 Xcode 开发工具创建一个命名为 ControlFlow 的 playground，进行循环代码的测试。要使用 for-in 结构进行有序的循环遍历，需要配合区间运算符，并且指定一个循环次数变量，示例如下：

```
//将打印 1，2，3，4，5
for index in 1...5 {
    print(index)
}
```

for-in 结构中需要两个参数：第 2 个参数既可以是一个集合类型的实例，也可以是一个范围区间；第 1 个参数为捕获参数，每次从第 2 个参数中遍历出的元素便会赋值给它，可供开发者在循环结构中直接使用。

如果在进行 for-in 循环遍历的时候，开发者并不需要捕获到的值，可以使用匿名参数来接收。Swift 中使用“_”符号来表示匿名参数，示例如下：

```
//如果不需要获取循环中的循环次序，可以使用如下方式
var sum=0;
for _ in 1...3 {
    sum += 1
}
```

对集合的遍历是 for-in 循环最常用的场景之一，这些在前边讲解集合类型的章节中有详细的介绍，简单的示例代码如下：

```
//遍历集合类型
var collection1:Array = [1,2,3,4]
var collection2:Dictionary = [1:1,2:2,3:4,4:4]
var collection3:Set = [1,2,3,4]
for obj in collection1 {
    print(obj)
}
for (key , value) in collection2 {
    print(key,value)
}
for obj in collection3 {
    print(obj)
}
```

4.3.2　while 与 repeat-while 条件循环结构

while 与 repeat-while 结构在 C/Objective-C 语言中也支持，并且功能基本一致，只是 Swift 语言将 do-while 结构修改为 repeat-while。

在开发中，经常会遇到需要进行条件循环的需求，例如模拟水池蓄水的过程，每次蓄水 1/10，当蓄满水后停止蓄水。while 循环结构可以十分方便地创建这类循环代码，示例如下：

```
var i=0
//当 i 不小于 10 时跳出循环
while i<10 {
    print("while",i)
    //进行 i 的自增加
    i+=1
}
```

在 while 循环结构中，while 关键字后面需要填写一个逻辑值或者以逻辑值为结果的表达式作为循环条件，如果逻辑值为真，则程序会进入 while 循环体。执行完循环体的代码后进行循环条件的判断，如果循环条件依然为真则会再次进入循环体，否则循环结束。由于 while 循环是根据循环条件来判断是否进入循环体的，如果循环条件一直成立，就会无限循环，因此在使用 while 循环的时候，注意要在循环体中对循环条件进行修改，且修改的结果是循环条件将不成立，否则会出现死循环。

上面演示的 while 结构会先进行条件判断再进行循环体的执行。repeat-while 结构则会先执行一次循环体再进行循环条件的判断。图 4-1 中列出了两种结构的异同。

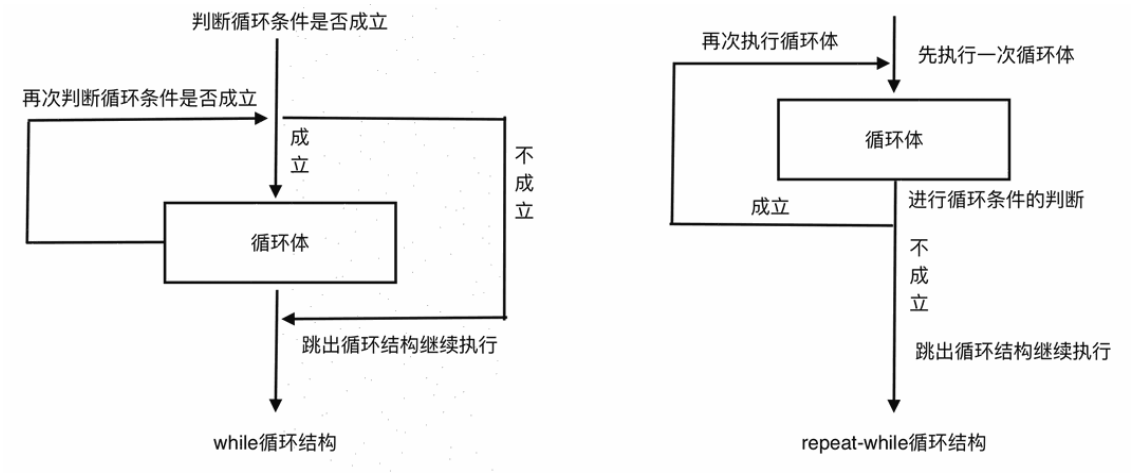

图 4-1　while 结构与 repeat-while 结构的异同

repeat-while 循环结构示例代码如下：

```
var j=0
//先执行一次循环体，再判断循环条件是否成立
repeat {
    print("repeat while")
    j+=1
} while j<10
```

4.4　条件选择与多分支选择结构

在开发中最重要的两种流程结构是循环结构与分支结构：循环结构用于处理大量的重复操作，分支结构用于处理由条件差异而产生的代码分支路径。Swift 语言中提供的分支结构有 if 结构、if-else 结构与 switch-case 结构。

4.4.1　if 与 if-else 条件选择结构

if 与 if-else 结构语句是 Swift 语言中用于最基础的分支结构语句，开发者可以使用单个的 if 语句进行单条件分支，也可以使用 if 与 else 组合来实现多条件分支，示例如下：

```
var c = 10
//进行 if 条件判断
if c<10 {
    print(c)
}
//进行 if-else 组合
if c>10 {
    c-=10
    print(c)
```

```
}else{
    print(c)
}
//进行 if-else 多分支组合
if c>0&&c<10 {
    print(c)
}else if c<=0 {
    c = -c
    print(c)
}else if c<=10&&c<20{
    c-=10
    print(c)
}else{
    print("bigger")
}
```

提　示

（1）需要注意，if 关键字后面跟的条件必须为严格意义上的逻辑值或者结果为逻辑值的表达式。这点 Swift 语言和 C/Objective-C 语言有一定差异。

（2）if-else 组合结构中的每个分支是互斥的，只能有一个分支的代码被执行，条件的判断顺序会从上到下进行，直到找到一个判断条件为真的分支或者最后一个 else 语句。同时，开发者也可以不加单独的 else 语句，这种情况下，如果没有条件成立的分支，则任何分支都不会被执行，程序会继续向后执行。

4.4.2　switch-case 多分支选择结构

switch 语句也被称为开关选择语句，它是通过匹配的方式来选择要执行的代码块，Swift 语言中的 switch 语句更加强大，不像 C/Objective-C 中的 switch 语句只能进行 int 类型值的匹配，Swift 语言中的 switch 语句可以进行任意数据类型的匹配，并且 case 子句的语法和扩展都更加灵活。使用 switch 结构进行字符分支匹配的示例如下：

```
//使用 switch 语句进行字符分支匹配
switch charac {
case "a":
    print("chara is a")
case "b":
    print("chara is b")
case "c":
    print("chara is c")
default ://default 用于处理其他额外情况
    print("no charac")
}
```

如上代码所示，switch 关键字后面需要填写要进行分支匹配的元素，在 switch 结构中通过子句 case 的列举进行元素值的匹配，匹配成功后会执行相应 case 子句中的代码，如上的代码将打印字符串“chara is b”。default 为 switch 语句中的默认匹配语句，即如果前面所有的 case 子句都没

有匹配成功，则会执行 default 中的代码。开发者也可以将 default 子句省略，这时如果所有 case 子句都没有匹配上，则会跳过 switch 结构，直接执行后面的代码。

还有一点读者需要注意，在 C/Objective-C 语言中，case 语句不会因匹配成功而中断，如果不进行手动控制，switch 结构中的 case 子句会依次进行匹配执行。举个例子，如果第 1 个 case 子句匹配成功，第 2 个 case 子句也匹配成功，则第 1 个 case 子句和第 2 个 case 子句中的代码都会执行，因此在 C/Objective-C 程序开发中，开发者一般会在每个 case 子句后面添加 break 关键字进行手动中断。Swift 语句优化了这一点，一个 case 语句匹配成功后，会自动跳出 switch 结构，如果不加特殊处理，switch 结构中的分支只会被执行一个或者一个也不执行。

switch-case 结构也支持开发者在一个 case 子句中编写多个匹配条件，程序在执行到这个 case 子句时，只要有一个条件匹配成功，就会执行此 case 下的代码，示例如下：

```
//同一个 case 中可以包含多个分支
switch charac {
case "a","b","c" :
   print("chara is word")
case "1","2","3" :
   print("chara is num")
default :
   print("no charac")
}
```

case 子句的匹配条件也可以是一个区间范围，当要匹配的参数在这个区间范围内时就会执行此 case 下的代码，示例如下：

```
//在 case 中也可以使用一个范围
var num = 3
switch num {
case 1...3 :
   print("1<=num<=3")
case 4 :
   print("chara is num")
default :
   print("no charac")
}
```

从上面的示例中可以了解，Swift 语言中的 switch-case 结构十分灵活强大，如果将 switch-case 结构和元组结合使用，开发者在编写代码时将更加灵活多变。首先，对于元组类型参数的匹配，case 子句可以进行选择匹配和优化匹配，示例如下：

```
//使用 Switch 语句进行元组的匹配
var tuple = (0,0)
switch tuple {
   //进行完全匹配
case (0,1):
   print("Sure")
   //进行选择性匹配
case (_,1):
   print("Sim")
   //进行元组元素的范围匹配
```

```
case(0...3,0...3):
    print("SIM")
default:
    print("")
}
```

如上代码所示，在进行元组的匹配时，有 3 种方式可以选择。第 1 种方式是完全匹配，即元组中所有元素都必须完全相等，才算匹配成功；第 2 种方式是选择匹配，即开发者只需要指定元组中的一些元素进行匹配，不需要关心的元素可以使用匿名参数标识符来代替，这种方式下只要指定的参数都相等就算匹配成功；第 3 种方式是范围匹配，即相应位置指定的范围包含需匹配元组相应位置的值就算匹配成功。其中第 2 种匹配方式可以和第 3 种匹配方式组合使用。

Swift 语言中的 case 子句中还可以捕获 switch 元组的参数，在相应的 case 代码块中可以直接使用捕获到的参数，这在开发中可以简化所编写的代码，示例如下：

```
var tuple = (0,0)
//进行数据绑定
switch tuple {
    //对元组中的第一个元素进行捕获
case (let a,1):
    print(a)
case (let b,0):
    print(b)
    //捕获元组中的两个元素，let(a,b) 与 (let a,let b)意义相同
case let(a,b):
    print(a,b)
default:
    print("")
}
```

这里读者需要注意，要捕获的元素并不能起到匹配的作用，例如元组 tuple 中有两个元素，如果 case 条件为(let a,1)，则在进行匹配时会匹配 tuple 中第 2 个参数，如果匹配成功就会将 tuple 元组的第 1 个参数的值传递给 a 常量，并且执行此 case 中的代码块，在这个代码块中，开发者可以直接使用常量 a。因此，要捕获的元素在匹配时实际上充当着匿名标识符的作用，如上代码所示的第 3 个 case 子句，其条件为 let(a,b)，实际上这个条件始终会被匹配成功。并且，如果开发者对元组中的所有元素都进行了捕获，在代码表现上，可以写作(let a,let b)，也可以直接捕获整个元组，写作 let(a,b)，这两种方式只是写法上有差异，在使用时并无差别。

switch-case 结构的参数捕获语法在使用起来为开发者带来了不少的便利。然而其也有一个问题，它将所有要捕获的元素都作为匿名参数来进行匹配，有时候并不是开发者想要的结果。例如上面的 tuple 元组示例，开发者需要捕获元组中的第 1 个元素，同时又需要与第 1 个元素相关的条件成立时再使 case 子句匹配成功，针对这种情况，Swift 也提供了相应的办法来处理，可通过在 case 语句中追加 where 条件的方式来实现上述需求：

```
//对于进行了数据捕获的 Switch-case 结构，可以使用 where 关键字来进行条件判断
switch tuple {
case (let a,1):
    print(a)
    //当元组中的两个元素都等于 0 时才匹配成功，并且捕获第一个元素的值
```

```
case (let b,0) where b==0:
    print(b)
//当元组中的两个元素相同时，才会进入下面的 case
case let(a,b) where a==b:
    print(a,b)
default:
    print("")
}
```

4.5 Swift 语言中的流程跳转语句

跳转语句可以提前中断循环结构，也可以人为控制选择结构的跳转，使代码的执行更加灵活多变。Swift 中提供了大量的流程跳转语句供开发者使用，熟悉这些语句的结构与特点可以使开发效率大大提高。Swift 中提供的流程跳转语句主要有 continue、break、fallthrough、return、throw、guard。

continue 语句用于循环结构中，其作用是跳过本次循环，直接开始下次循环。这里需要注意，continue 的作用并不是跳出循环结构，而是跳过本次循环，直接执行下一个循环周期。示例如下：

```
for index in 0...9 {
    if  index == 6 {
        continue
    }
    print("第\(index)次循环")
}
```

上面的示例代码将跳过 index 等于 6 时的代码块，在打印信息中会缺少 index 等于 6 时的打印输出。需要注意的是，continue 语句默认的操作范围是直接包含它的这一层循环结构，如果代码中嵌套了多层循环结构，continue 语句会跳过本次循环。那么，如果想要实现不跳过本次循环，而是直接跳至开发者指定的那一层循环结构，该如何写呢？示例如下：

```
MyLabel:for indexI in 0...2 {
    for indexJ in 0...2 {
        if indexI == 1 {
            continue MyLabel
        }
        print("第\(indexI)\(indexJ)次循环")
    }
}
```

以上代码创建了两层循环结构，在内层循环中使用 continue 语句进行跳转，MyLabel 是外层循环的标签，因此这里的 continue 跳转将会跳出 indexI 等于 1 时的外层循环，直接开始 indexI 等于 2 的循环操作。

break 语句是中断语句，也可以用于循环结构中。和 continue 语句不同的是，break 语句会直接中断直接包含它的循环结构，即当循环结构为一层时，如果循环并没有执行完成，则后面所有的循

环都将被跳过。如果有多层循环结构，程序会直接中断直接包含它的循环结构，继续执行该循环结构外层的循环结构，示例如下：

```
for index in 0...9 {
    if  index == 6 {
        break
    }
    print("第\(index)次循环")
}
```

上面的代码在 index 等于 6 时使用了 break 语句进行中断，第 5 次循环后的所有打印信息都将被跳过。break 语句默认将中断直接包含它的循环结构，同样也可以使用指定标签的方式来中断指定的循环结构，示例如下：

```
MyLabel:for indexI in 0...2 {
    for indexJ in 0...2 {
        if indexI == 1 {
            break MyLabel
        }
        print("第\(indexI)\(indexJ)次循环")
    }
}
```

break 语句也可以用于 switch 结构中。在 switch 结构中，break 语句将直接中断后面所有的匹配过程，直接跳出 switch 结构。在 Swift 语言中，switch-case 选择匹配结构默认就是 break 操作，故开发者不必手动添加 break 代码。

fallthrough 语句是 Swift 中特有的一种流程控制语句。前面提到过，当 Swift 语言中的 switch-case 结构匹配到一个 case 后会自动中断后面所有 case 的匹配操作。如果在实际开发中需要 switch-case 结构不自动进行中断操作，可以使用 fallthrough 语句，示例如下：

```
var tuple = (0,0)
switch tuple {
case (0,0):
    print("Sure")
    //fallthrough 会继续执行下面的 case
    fallthrough
case (_,0):
    print("Sim")
    fallthrough
case(0...3,0...3):
    print("SIM")
default:
    print("")
}
```

如上示例代码将会打印 Sure、Sim 和 SIM。

return 语句对于读者来说应该十分熟悉，其在函数中用于返回结果值，也可以用于提前结束无返回值类型的函数。当然，return 语句的应用场景也不只局限于函数中，在闭包中也可以使用 return 进行返回。关于函数的相关知识，会在后面的章节做详细的介绍，这里只做简单演示，示例如下：

```
//有返回值函数的返回
func myFunc()->Int{
    return 0
}
//无返回值函数的返回
func myFunc(){
    return
}
```

throw 语句用于异常抛出，throw 语句抛出的异常如果不进行捕获处理，也会使程序中断。Swift 语言中有抛出异常和处理异常的代码结构，在后面章节中会详细介绍，这里也只做简单演示。在函数中抛出异常的代码示例如下：

```
//定义异常类型
enum MyError:Error{
    case errorOne
    case errorTwo
}
func  newFunc() throws{
    //抛出异常
    throw MyError.errorOne
}
```

guard-else 结构语句是 Swift 2.0 之后新加入的一种语法结构，Swift 团队创造它的目的在于使代码的结构和逻辑更加清晰。在实际开发中，尤其是在函数的编写中，经常会遇到这样的场景：就是当参数符合某个条件时，函数才能正常执行，否则直接通过 return 来终止函数的执行，如果不使用 guard-else 结构，示例代码如下：

```
func myFuncTwo(param:Int)  {
    if param <= 0  {
        return
    }
    print("其他操作")
}
```

上面的代码结构在逻辑上并不那么优美，开发者的原意是当 param 参数大于 0 时才执行函数中的操作，在2.0之前却使用了相反的逻辑来中断函数，当然，开发者也可以将函数实现为如下形式：

```
func myFuncTwo(param:Int)  {
    if param > 0  {
         print("其他操作")
    }
}
```

经过修改后，代码逻辑清晰了许多，然而还是有一些问题。如果这个函数中需要做的操作很多，那么所有条件判断的代码都将写在 if 语句块中，代码结构就会显得杂乱无章。guard-else 语句就是为了优化这种情况而产生的。guard-else 语句也被称为守护语句，顾名思义，其作用就是确保某个条件成立才允许其后的代码执行。示例如下：

```
func myFuncTwo(param:Int)  {
    guard param>0 else {
        return
    }
    print("其他操作")
}
```

4.6　练习及解析

（1）将下列描述翻译成 Swift 表达式。

小李买了 5 支铅笔、1 块橡皮、3 本作业本和 11 个书签。每支铅笔 2 元，每块橡皮 3 元，每本作业本 2.5 元，每个书签 0.5 元，计算小李花了多少钱。

解析：

```
//共 26 元
var sum = 5*2+1*3+3*2.5+11*0.5
```

（2）设计一个表达式来生成 1~7 之间的随机数。

解析：

```
// arc4random()为 Swift 标准函数库中的随机数生成函数
var rand = arc4random()%7+1
```

（3）对语、数、外 3 门科目进行测试，当 3 门科目的成绩都大于 60 且总分不小于 200 分时，成绩才为合格，使用 Swift 表达式来描述上述逻辑。

解析：

```
var Language=60
var Math=65
var English=70
if Language>60 && Math>60 && English>60 && (Language+Math+English)>200 {
    print("合格")
}
```

（4）编写闰年判断的表达式。

闰年：① 能够被 400 整除。

② 能够被 4 整除但是不能够被 100 整除。

解析：

```
var year = 2016
if year%400==0 || ((year%4==0) && (year%100 != 0)) {
    print("闰年")
}
```

（5）学校乒乓球比赛需要每班出一名主选手和一名辅助选手参赛，比赛分为上、下两场，上半场主选手得分超过 30 分则下半场需要辅助选手进行，否则下半场依然由主选手进行，使用条件运算符（三目运算符）描述下半场出赛的选手。

解析：

```
var mark = 40
var people = mark>30 ? "主选手" : "辅助选手"
```

（6）打印如下图案：

```
**********
*????????*
*????????*
**********
```

解析：

```
for indexH in 1...4 {
    //每行有 10 列符号
    print("")
    for indexV in 1...10 {
        //第一行和最后一行为*
        if indexH==1 || indexH==4 {
            print("*", separator: "", terminator: "")
        }else{
            //第一列和最后一列为*
            if indexV==1 || indexV==10 {
                print("*", separator: "", terminator: "")
            }else{
                //其余为?
                print("?",separator: "",terminator: "")
            }
        }
    }
}
```

print()函数会自动在打印末尾添加换行符，使用带 3 个参数的 print()函数，并且将后两个参数设置为空字符串，以屏蔽 print 函数的自动换行功能。

（7）打印出所有的“水仙花数”。所谓“水仙花数”，是指一个 3 位数，其各位数字的立方和等于该数本身。

解析：

```
for item in 100...999 {
    //获取个位数字
    var dig = item%10
    //获取十位数字
    var tens = item/10%10
    //获取百位数字
    var hundred = item/100
    //获取结果，这里可以考虑用 pow(Double,Double) 函数代替 dig*dig*dig
    var sum = dig*dig*dig + tens*tens*tens + hundred*hundred*hundred

    if sum == item {
        print(item)
    }
```

```
}
```

（8）猴子吃桃问题：猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上想再吃时，就只剩下一个桃子了。求第一天共摘了多少个桃子。

解析：

```
var count = 1
for day in 1...9 {
    count = (count+1)*2
}
print(count)
```

（9）两个乒乓球队进行比赛，每队各出三人。甲队为 p1、p2、p3 三人，乙队为 q1、q2、q3 三人。抽签决定了比赛名单后，有人向队员打听比赛的名单。p1 说他不和 q1 比，p3 说他不和 q1、q3 比，请编写程序列出三对赛手的名单。

解析：

```
//标识甲队
var p1 = 1
var p2 = 2
var p3 = 3
//标识乙队
var q1 = 0
var q2 = 0
var q3 = 0
for indexI in 1...3 {
    q1 = indexI
    for indexJ in 1...3 {
        q2 = indexJ
        for indexK in 1...3 {
            q3 = indexK
            if indexI != indexJ && indexI != indexK && indexJ != indexK {
                if q1 != p1 && p3 != q1 && p3 != q3 {
                    print(q1,q2,q3)
                }
            }
        }
    }
}
//输出 2,3,1
```

（10）求 1+2!+3!+...+20!的和。

解析：

```
var sumC = 0
for var index in 1...20 {
    var tmp = 1
    while index > 0{
        tmp *= index
```

```
            index -= 1
        }
        sumC+=tmp
    }
    print(sumC)
    //输出 2561327494111820313
```

（11）打印倒金字塔：

```
  * * * * * * *
    * * * * *
      * * *
        *
```

解析：

```
        for indexJ in 1...7 {
            if indexJ < indexI{
                //先打印左侧空格
                print(" ", separator: "", terminator: "")
            }else if indexJ+(indexI-1)<=7 {
                //再打印*
                print("*",separator: "",terminator: "")
            }
        }
        //换行
        print("")
    }
```

4.7 模 拟 面 试

（1）编程中的流程控制结构有哪几种，分别用于什么场景？

回答要点提示：

① 编程中主要的流程结构有顺序结构、分支结构、循环结构、跳转与中断结构。

② 在编写代码时，我们的核心思路和代码的主流程都是线性的，代码是一行一行向下执行的，这就是我们最常用的顺序结构。分支结构是程序逻辑的重要描述方式，输入的不同，不同的运行场景都会对程序执行的结果产生影响，这时我们需要使用分支结构来处理。循环结构用来处理大量重复的工作。跳转和中断结构使得分支和循环结构更加灵活可控。

核心理解内容：

理解各种程序流程控制的方法，能够在开发中根据实际场景灵活使用各种流程控制结构。

（2）运算符是一门编程语言的基础，Swift 中有哪些特殊的运算符？

回答要点提示：

① Swift 是一门非常强大的语言，在 Swift 中开发者可以根据需要对运算符进行重载，也可以进行运算符的自定义。

② Swift 语言中提供了空合并运算符来对 Optional 值进行快捷的条件运算。

③ 在 Swift 语言中，区间运算符也是一种十分有特点的运算符，使用它可以方便地创建区间与范围，在集合遍历、字符串和数组的截取中都十分有用。

核心理解内容：

熟悉 Swift 中的运算符重载和自定义的方法，熟练使用 Swift 原生定义的各种运算符。

第5章

函数与闭包技术

所谓科学，包括逻辑和数学在内，都是有关时代的函数，所有科学连同它的理想和成就统统都是如此。

——穆尔

任何复杂的系统都是由许多简单的系统组合演化而来的，在编程中更是如此，任何复杂的功能都是由一些简单的功能组合演化而来的。函数是高级语言共有的代码特性。在数学中，函数是一种特定的算法映射，自变量的改变将引起因变量的改变。在编程中，函数的实质是完成特定功能的代码块，只是此代码块有一个名称，开发者可以通过函数名来调用函数完成特定的需求和功能。

Swift 语言中提供了十分灵活的方式来创建和调用函数。实际上在 Swift 语言中，每个函数都有特定的类型，函数的类型取决于参数和返回值。另外，在 Swift 语言中函数可以进行嵌套。

闭包的功能与函数类似，也是有一定功能的代码块，在 Objective-C 语言中，与之相似的语法结构被称为 block 结构。闭包与函数有着密不可分的关系：函数是有名称的功能代码块，闭包在大多数情况下是没有名称的功能代码块；在语法结构上，闭包与函数也有很大的差异。由于对闭包语法的支持，Swift 语言更加强大而灵活。本章将向读者介绍在 Swift 语言中闭包结构的应用与简化技巧。

通过本章，你将学习到：

- 函数的创建与调用。
- 函数的类型与嵌套。
- 函数的 inout 参数。
- 编写可变参数函数。
- 了解闭包的应用场景及设计思路。
- 对闭包结构进行简化。

- 特定条件下使用后置闭包。
- 逃逸与非逃逸闭包的应用。
- 自动闭包的应用。

5.1　函数的基本应用

在数学中，函数有 3 要素：定义域、对应关系和值域。在编程中，抛开函数的实现，在声明函数时也有 3 要素：参数、返回值和函数名。参数和返回值决定函数的类型。参数数量和类型完全相同，同时返回值类型也相同的函数为同类型函数。在 Swift 语言中，使用 func 关键字来声明函数，一个完整的函数声明和实现应该符合如下格式：

```
func methodName(param1,param2,…)->returnValue{实现部分}
```

methodName 需要编写为函数名称，之后跟的小括号中需要设置函数的参数类型和个数，多个参数使用逗号进行分割。参数列表后面使用符号“->”来连接返回值类型，到此，函数的声明部分就完成了，如果要对函数进行实现，在后面追加大括号，里面为函数的实现代码。如果一个函数没有返回值，也可以将参数列表后面的部分省略。在调用函数时，直接使用函数名来进行调用。

5.1.1　函数的创建与调用

使用 Xcode 开发工具创建一个命名为 Function 的 playground 文件，在其中编写如下示例代码：

```
//编写一个函数，功能为传入一个数字，判断其是否大于 10，并将结果返回
func isMoreThanTen(count:Int)->Bool {
    if count>10 {
        return true
    }else{
        return false
    }
}
//进行函数的调用
//将返回 false
isMoreThanTen(count: 9)
//将返回 true
isMoreThanTen(count: 11)
```

参数列表中的参数需要指定参数名和参数类型，也可以编写无参的函数，为空即可，示例代码如下：

```
//编写无参的函数
func myFunc1()->String{
    return "无参函数"
}
//将返回字符串"无参函数"
```

```
myFunc1()
```

如果函数也不需要返回值，可以选择返回 Void 或者直接省略返回值部分，示例代码如下：

```
//编写无参无返回值的函数
func myFunc2() -> Void {
    print("无参无返回值")
}
func myFunc3() {
    print("省略返回值")
}
myFunc2()
myFunc3()
```

还有一种情况比较特殊，原则上函数的返回值只能是一个，而在实际开发中如果需要返回多个值通常会采用复合类型来处理。在 Objective-C 语言中，不支持元组类型，要进行多个值的返回时会采用返回数组或者字典的方式。在 Swift 语言中，可以用元组来达到这样的效果，模拟一个数据查询的函数，这个函数将通过传入一个数据 ID 来进行数据查询操作，并返回查询状态和具体的数据，示例代码如下：

```
//模拟数据查询函数
func searchData(dataID:String)->(succsee:Bool,data:String){
    //模拟一个查询结果和数据实体
    let result = true
    let data = "数据实体"
    return (result,data)
}
if searchData(dataID: "1101").succsee {
    //查询成功
    print(searchData(dataID: "1101").data)
}
```

Swift 语言中的函数还有一个使用技巧。开发者可以通过返回 Optional 可选值类型来标识函数执行是否成功，在调用函数时使用 if-let 结构做安全性检查，示例代码如下：

```
//返回 Optional 类型值的函数
func myFunc4(param:Int)->Int?{
    guard param>100 else{
        return nil
    }
    return param-100
}
if let tmp = myFunc4(param: 101) {
    print(tmp)
}
```

5.1.2 关于函数的参数名

有过编程经验的读者可能会发现各个编程语言都有一些特点。以函数的参数名为例，在

Objective-C 中实际上函数的参数名是隐含于函数名称中的，示例如下：

```
//Objective-C 语言中函数的风格
-(void)getDataFromDataID:(NSString*)dataID{
}
//对函数进行调用
[self getDataFromDataID:@"1101"];
```

Objective-C 这种风格的函数写法有一个很大的优点，即开发者在调用函数时根据函数名中的信息就可以推断出参数的意义。如上代码所示，getDataFromDataID 很容易使开发者联想到此参数需要传递数据的 ID 值。这里会产生一个问题，函数名将变得非常冗长，编码界面将变得十分拥挤。在 Java 中，参数名是直接添加在参数列表中的，示例如下：

```
//Java 语言中函数的风格
private void getMyData(String dataID){
}
getMyData("1101");
```

通过比较 Java 与 Objective-C 的函数风格，可以发现 Java 语言要简练得多，但同时也有缺陷：在调用函数时，函数参数列表中的参数并没有一个参数名标识，这样开发者在调用函数或者检查代码时不能一目了然地明白各个参数的意义。在参数很多的情况下，这个问题就变得尤为突出。

Swift 语言中的函数风格借鉴了 Objective-C 与 Java 的优势和劣势，引入了参数的内部命名与外部命名概念。内部命名在函数实现时使用，外部命名在函数调用时使用。在上面所有例子编写的函数中，参数名都是内部命名，开发者若不设置参数的外部命名，则默认函数参数的外部命名与内部命名相同。因此开发者在调用函数时，传入的参数前面都有一个参数名标注，示例如下：

```
//多参数函数，默认内部命名与外部命名相同
func myFunc5(param1: Int,param2: Int,param3: Int) {
    //这里使用的 param1、param2、param3 是参数的内部命名
    param1+param2+param3
}
//调用函数的参数列表中使用的 param1、param2 和 param3 为外部命名
myFunc5(param1: 1, param2: 2, param3: 3)
```

在声明函数时，也可以在内部命名的前面再添加一个名称作为参数的外部命名，示例如下：

```
//为函数的参数添加外部命名
func myFunc6(out1 param1: Int,out2 param2: Int,out3 param3: Int) {
    //这里使用的 param1、param2、param3 是参数的内部命名
    param1+param2+param3
}
//调用函数时，参数将被外部命名标识，这里的 out1、out2、out3 为函数参数的外部命名
myFunc6(out1: 1, out2: 2, out3: 3)
```

有了 Swift 中参数内部名称与外部名称的语法规则，开发者可以十分灵活地编写函数。参数的外部名称会在调用函数时标识参数，这样既简化了函数名，也能很好地帮助开发者理解每个参数的意义，并且这种语法的优势在进行函数重载操作时会更大。在后面讲解函数重载的章节中，读者就能体会到。

Swift 语言也支持省略函数参数的外部名称，默认函数参数的外部名称与内部名称相同，开发

者可以使用匿名变量标识符“_”来对外部名称进行省略，示例如下：

```
//省略外部名称的函数参数列表
func myFunc7(_ param1:Int,_ param2:Int , _ param3:Int){
    param1+param2+param3
}
//在调用函数时，不再标识参数名称
myFunc7(1, 2, 3)
```

5.1.3 函数中参数的默认值、不定数量参数与 inout 类型参数

在进行函数调用时，每个参数都必须要传值，这句话其实并不十分准确，应该说每个参数都必须有值。除了在调用时为参数传值外，Swift 语言中函数的参数也支持设置默认值。需要注意的是，如果函数的某个参数设置了默认值，那么开发者在调用函数的时候既可以传此参数的值，也可以不传此参数的值，但是参数的位置要严格对应。示例如下：

```
//默认参数 param2 的值为 10，param3 的值为 5
func myFunc8(param1:Int,param2:Int = 10 ,param3:Int = 5)  {
    param1+param2+param3
}
//对每个参数都进行传值
myFunc8(param1: 1, param2: 1, param3: 1)
//只对没有设置默认值的参数传值
myFunc8(param1: 10)
func myFunc9(param1:Int,param2:Int=10 ,param3:Int)  {
    param1+param2+param3
}
//对应的参数位置要一致
myFunc9(param1: 10,param3:10)
```

在开发中还有一种情况也十分常见，有时候开发者需要编写参数个数不定的函数。例如，打印函数 print()，其中传入参数的数量就是不确定的。对于这类函数的编写，Swift 也对它做了很好的支持。编写一个函数，传入不定个数的整数值，将其相加后的结果打印出来，代码如下：

```
//编写参数数量不定的函数
func myFunc10(param:Int...){
    var sum=0;
    for count in param {
        sum+=count
    }
    print(sum)
}
//传递参数的个数可以任意
myFunc10(param: 1,2,3,4,5)
myFunc10(param: 12,2,3)
```

实际上，在 Swift 语言中某个参数类型的后面追加符号“…”，就会将此参数设置为数量可变。在函数内部，开发者传递的值会被包装成一个集合类型赋值给对应参数。需要注意，传递的

参数类型必须相同，并且可以传递多组数量可变的参数，不同参数之间参数类型可以不同，示例如下：

```
func myFunc11(param1:Int...,param2:String)  {
    var sum=0;
    for count in param1 {
        sum+=count
    }
    print("\(param2):\(sum)")
}
myFunc11(param1: 1,2,3, param2: "hello")
```

Swift 语言支持设置函数参数的默认值，支持传递数量不定的参数，如果开发者在编写代码时灵活运用函数，就可以达到事半功倍的效果。

关于 Swift 语言的参数传递，还有这样一个特点：传递的如果是值类型的参数，那么参数值在传递进函数内部时会将原值复制为一份常量，且在函数内不可以修改。关于值类型和引用类型的相关知识，后面章节会详细介绍，这里读者只需要了解：类属于引用类型，而基本数据类型、枚举和结构体都属于值类型。对于值类型参数，如果开发者在函数内部修改参数的值，编译器会直接报错，示例代码如下：

```
//错误示范
//func myFunc12(param:Int){
//   param+=1
//}
```

如果在开发中真的需要在函数内部修改传递参数的变量的值，可以将此参数声明为 inout 类型，示例代码如下：

```
//在函数内部修改参数变量的值
func myFunc12(param:inout Int){
    param+=1
}
var para = 10;
myFunc12(param: &para)
//将打印 11
print(para)
```

注意，在上面的演示代码中将参数 param 声明为了 inout 类型，在传参时需要使用“&”符号（这个符号将传递参数变量的内存地址）。

5.2　函数的类型与函数嵌套

前面章节有提到，Swift 语言中每一个函数都有其特定的类型。因此，开发者也可以像声明普通变量那样来声明一个函数变量，同样也可以对此变量进行赋值、调用等操作。将函数作为数据类型这种语言设计思路有强大的优势，这将允许开发者将一个函数作为另一个函数的参数或者返回

值，大大增强了编程的灵活性。

函数变量的声明及赋值示例代码如下：

```
//声明一个函数变量
var addFunc:(Int,Int)->Int
//对函数变量进行赋值
addFunc = {(param1:Int,param2:Int) in return param1+param2}
//调用函数变量
addFunc(2,3)
```

函数变量的类型由参数和返回值决定，参数和返回值相同的函数类型就相同。上面示例代码中对函数变量的赋值采用了闭包的方式，闭包的实质是一段有具体功能的代码块，其结构为{(param1,param2,…) in 代码块}，其最外面由大括号包围，内部小括号为参数列表，in 为闭包关键字，之后需要编写实现相应功能的代码。关于闭包的更多内容，后面章节会详细介绍。

也可以通过一个函数来对函数变量进行赋值，示例如下：

```
var addFunc:(Int,Int)->Int
func myFunc13(param1:Int,param2:Int) -> Int {
    return param2+param1
}
addFunc = myFunc13
addFunc(1,2)
```

函数也可以作为另一个函数的参数，示例代码如下：

```
//参数 param 的类型为函数类型(Int,Int)->Int
func myFunc14(param:(Int,Int)->Int) {
    print(param(1,2))
}
//将 addFunc 函数作为参数传递进 myFunc14 函数
myFunc14(param: addFunc)
```

如上代码所示，这种将函数作为参数的编程方式应用十分广泛，在 Objective-C 语言中，这种语法结构被称为 block。在处理一些回调操作时，例如网络回调、子线程异步处理回调等场景中，使用这种编程方式将十分简洁优美。

函数可以作为参数，同样其也可以作为返回值来使用，示例代码如下：

```
//声明一个函数变量
var addFunc:(Int,Int)->Int
func myFunc15() -> (Int,Int)->Int {
    return {(param1:Int,param2:Int)in
        return param1+param2
    }
}
//使用 addFunc 变量获取返回值
addFunc = myFunc15()
//进行调用
addFunc(1,2)
```

上面的演示代码中，在函数内部创建了闭包并将其返回，由于 Swift 语言是支持进行函数嵌

套的，实际上开发者也可以在函数内部再次创建函数，示例如下：

```
func myFunc16() -> (Int,Int)->Int {
    func subFunc(param1:Int,param2:Int)->Int{
        return param1+param2
    }
    return subFunc
}
```

提　示

函数也有其作用域。所谓嵌套函数，是指在函数内部再次创建一个子函数，子函数只能在父函数内部调用，不可以在父函数外部调用，但是可以作为返回值传递到父函数外部。

5.3　理解闭包结构

闭包结构对于编程初学者来说可能会难于理解一些。在学习关于闭包的过程中，读者首先应该理解闭包的结构与实质。

5.3.1　闭包的语法结构

使用 Xcode 开发工具创建一个命名为 Closures 的 playground 文件，本章将在其中进行代码的演练。5.2 节中向读者介绍了有关函数的内容，函数的设计思路是将有一定功能的代码块包装在一起，通过函数名实现复用。闭包和函数有着类似的作用，然而闭包的设计大多数情况下并不是为了代码的复用，而是传递功能代码块和处理回调结构。首先，一个完整的函数包含函数名、参数列表、返回值和函数体，示例如下：

```
//标准函数 这个函数的功能是计算某个整数的平方
func myFunc(param:Int)->Int{
    return param*param
}
```

将上面函数的功能使用闭包来实现，代码如下：

```
//闭包的实现方式
let myClosures = {(param:Int)->Int in
    return param*param
}
```

上面的代码创建了一个名为 myClosures 的闭包常量，闭包在语法上有这样的标准结构：{(参数列表)->返回值 in 闭包体}。首先闭包的最外层由大括号包围，内部由闭包关键字 in 来进行分割，关键字 in 前面为闭包结构的参数列表和返回值，其书写规则与函数一致，in 关键字后面为闭包体，用于实现具体功能。上面示例的闭包和函数原理上完全相同，并且闭包也可以像函数一样被调用，示例代码如下：

```
//对函数进行调用 将返回 9
myFunc(param: 3)
//对闭包进行调用 将返回 9
myClosures(3)
```

与函数不同的是，闭包的返回值是可以省略的，在闭包体中，如果有 return 返回，则闭包会自动将 return 的数据类型作为返回值类型，上面的闭包代码也可以简写为如下样式：

```
//闭包的实现方式
let myClosures = {(param:Int) in
    return param*param
}
```

5.3.2 通过实现一个排序函数来深入理解闭包

在实践中分析与解决问题是学习编程的一条捷径。本节将带领读者通过分析问题、探讨解决方案、进行初步实现、优化实现方式等一步步深入了解闭包的用法。学习这种分析解决问题的方式在编程中十分有益，其思路如图 5-1 所示。

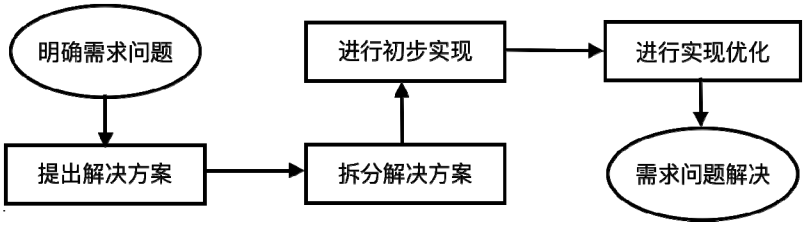

图 5-1 分析与解决问题思路

在实际开发中，开发者经常会遇到不同的排序需求，例如对商品价格排序、文章热度排序、消息时间先后排序、学生成绩排序等。很多情况下，开发者要排序的对象并不是简单的数字类型值或者字符串类型值，而是自定义的复杂对象，也就是开发者常用的类。对于这种类型的排序需求，应该如何实现呢？首先应该明确需求问题：

（1）应该实现一个函数，来对数组类型排序。
（2）数组中的元素可以是任意的复杂类型。

实现根据复杂类型数据中的某一个属性进行排序，例如学生的成绩。

（1）针对上面提出的需求问题，设计初步的实现思路。
（2）以通用的数组类型作为函数的参数。

若要实现对自定义复杂类型的排序操作，需要将排序算法作为参数传入函数。
编写函数的结构示例如下：

```
func mySort(array:Array<Any>,sortClosure:(Int,Int)->Bool) -> Array<Any> {
    return array
}
```

提　示

Any 在 Swift 语言中代表任意类型。

mySort()函数中需要传入两个参数，一个是要进行排序的数组数据，另一个是一个闭包排序方法，这个闭包有两个 Int 类型的参数，表示数组中两个相邻的元素。第 1 个参数表示前一个元素，第 2 个参数表示后一个元素，这个闭包有一个 Bool 类型的返回值，返回 true 则表示正向排序，即参数中的第 1 个元素和第 2 个元素不交换位置，返回 false 表示逆向排序，即参数中的第 1 个元素和第 2 个元素交换位置。之后，根据上面的分析来对 mySort 函数进行实现，代码如下：

```
func mySort(array:inout Array<Any>,sortClosure:(Int,Int)->Bool) -> Array<Any> {
    //冒泡排序算法
    for indexI in array.indices {
        //最后一个元素直接返回
        if indexI == array.count-1 {
            break
        }
        //冒泡排序
        for indexJ in 0...((array.count-1)-indexI-1){
            //调用传递进来的闭包算法
            if sortClosure(indexJ,indexJ+1) {

            }else{
                //进行元素交换
                swap(&array[indexJ], &array[indexJ+1])
            }
        }
    }
    return array
}
```

如上代码所示，使用了冒泡排序算法来进行排序操作，而具体两个元素的排序规则是由闭包 sortClosure 来实现的。swap()函数是 Swift 语言中的一个交换函数，用来实现数组元素的交换，由于需要对原数组数据进行操作，需要使用 inout 类型的数组参数。

先使用整型数组来对编写的排序函数进行测试，代码如下：

```
var  array:Array<Any> = [1,4,3,5,7,5,4,2,7]
mySort(array: &array,sortClosure: {(index:Int, nextIndex:Int) -> Bool in
    return (array[index] as! Int) > (array[nextIndex] as! Int)
})
print(array)
```

提　示

as!的作用是类型转换。

编写一个自定义的类来进行排序测试，示例如下：

```
//编写一个学生类
class Student {
```

```
        //学生成绩
        let achievement:Int
        //学生姓名
        let name:String
        //构造方法
        init(name:String,achievement:Int){
            self.achievement = achievement
            self.name=name
        }
    }
    //创建 4 个学生
    let stu1 = Student(name: "小王", achievement: 89)
    let stu2 = Student(name: "小李", achievement: 69)
    let stu3 = Student(name: "小张", achievement: 81)
    let stu4 = Student(name: "小孙", achievement: 93)
    //将学生放入数组
    var stuArr:Array<Any> = [stu1,stu2,stu3,stu4]
    //进行排序
    mySort(array: &stuArr, sortClosure : { (index: Int, nextIndex: Int) -> Bool
in
        return (stuArr[index] as! Student).achievement > (stuArr[nextIndex] as!
Student).achievement
    })
```

以上代码模拟了一个学生类，每一个学生对象由名字和分数组成，闭包实现了对学生分数的排序规则。

5.4 将闭包作为参数传递时的写法优化

通过前面章节的学习，读者应该可以感受到 Swift 语言在设计上追求的简洁与高效，开发者在将闭包作为参数传递进函数时，也可以在标准形式上做许多优化。我们依然以学生数组排序的代码为例，省略闭包返回值类型的写法如下：

```
    //省略返回值
    mySort(array: &stuArr, sortClosure: { (index, nextIndex) in
        return (stuArr[index] as! Student).achievement > (stuArr[nextIndex] as!
Student).achievement
    })
```

闭包作为函数参数时的参数类型可以省略，是因为在函数声明时，闭包参数中已经指定了参数的类型，编译器可以进行自动推断。

如果闭包只有一行代码组成，return 关键字也可以进行省略，默认会将此行代码的执行结果返回。需要注意，只有将闭包作为函数的参数才可以如此简化，示例如下：

```
    //省略 return
    mySort(array:&stuArr, sortClosure: { (index, nextIndex) in
        (stuArr[index] as! Student).achievement > (stuArr[nextIndex] as!
```

```
Student).achievement
    })
```

经过简化后的闭包结构已经简洁了很多，其实还可以继续简化。如上代码中，开发者使用 index 和 nextIndex 来标识闭包中的参数，实际上，当此闭包作为函数的参数时，闭包的参数列表会自动创建一组参数，参数名会以$0、$1 这样的结构依次类推。因此，开发者也可以使用编译器默认生成的参数名而不必指定参数名。表现在代码写法上，开发者也可以将参数列表和闭包关键字 in 省略，优化后的代码如下所示：

```
    mySort(array: &stuArr, sortClosure: {
        (stuArr[$0] as! Student).achievement > (stuArr[$1] as!
Student).achievement
    })
```

一步步简化后的代码与最开始的代码模样有很大的不同，Swift 语言在这些细节上的处理使开发者可以十分灵活地编写代码。然而这些代码的简化操作对于初学者来说可能会难于理解，读者务必要将本节的简化过程熟练应用，在开发中需要使用各种各样的闭包时，才能游刃有余。

5.5　后置闭包、逃逸闭包与自动闭包

闭包常常会作为函数的参数来使用，函数在调动时，参数是写在小括号中的参数列表中的，而闭包又是一个写在大括号中的代码块，如此的嵌套写法在视觉上十分不直观。因此，Swift 语言中提供了后置闭包的写法。当函数中的最后一个参数为闭包参数时，在调用函数时，开发者可以将闭包结构脱离出函数的参数列表，追加在函数的尾部，增强代码的可读性，示例如下：

```
    //原结构
    mySort(array: &stuArr, sortClosure: {
    (stuArr[$0] as! Student).achievement > (stuArr[$1] as! Student).achievement
    })
    //后置闭包结构
    mySort(array: &stuArr){
    (stuArr[$0] as! Student).achievement > (stuArr[$1] as! Student).achievement
    }
```

后置闭包的语法简化了代码的结构，这里面还有一个小技巧，如果一个函数只有一个参数，且这个参数是一个闭包类型的参数，则开发者在调用函数时，使用后置闭包的写法可以直接将函数的参数列表省略，示例代码如下：

```
    //只有一个闭包参数的函数
    func myFunc(closure:(Int,Int)->Bool) {

    }
    //进行闭包的后置 可以省略参数列表
    myFunc {
        $0>$1
    }
```

以上示例代码几乎是闭包的最简形式了。

当闭包传递进函数时，系统会为此闭包进行内存的分配。在 Swift 语言中，还有逃逸闭包与非逃逸闭包这样的概念。所谓逃逸闭包，是指函数内的闭包在函数执行结束后在函数外依然可以进行使用，非逃逸闭包是指当函数的生命周期结束后，闭包也将被销毁。换句话说，非逃逸闭包只能在函数内部使用，在函数外部不能够使用。默认情况下函数参数中的闭包都为非逃逸闭包，这样做的优点是可以提高代码性能，节省内存消耗，开发者可以根据实际需求将闭包参数声明成逃逸闭包。

提　示

非逃逸闭包也不可以作为返回值返回，如果这么做，编译器会抛出一个错误。

将闭包声明为非逃逸类型，需要使用@noescape 修饰。需要注意的是，在最新版本的 Xcode 开发工具中，这个关键字已经不需要再使用，参数中的闭包默认都是非逃逸的，示例代码如下：

```
//只有一个闭包参数的函数，将此闭包声明为非逃逸的，此闭包既不可作为返回值返回也不可赋值给外部变量
//在 Xcode8.1 中会有警告 这个关键字可以省略
func myFunc(closure: @noescape (Int,Int)->Bool){

}
```

提　示

逃逸类型的闭包常用于异步操作中，例如一个后台请求完成后要执行闭包回调，需要使用逃逸类型。

不是所有的闭包都需要显式创建，Swift 语言中还有一种语法，其可以实现对简单闭包的自动生成，这种闭包通常称为自动闭包。需要注意，自动闭包参数的使用有严格的条件，首先此闭包不能够有参数，其次在调用函数传参时，此闭包的实现只能由一句表达式组成，闭包的返回值即为此表达式的值，自动闭包参数由@autoclosure 来声明，示例代码如下：

```
//将闭包参数声明为自动闭包
func myFunc2(closure: @autoclosure ()->Bool)  {

}
//调用函数时，直接传入一个表达式即可，编译器会自动生成闭包参数
myFunc2(1+2+3>10)
```

自动闭包默认为非逃逸的，若要使用逃逸类型的闭包参数，需要声明如下：

```
//将闭包参数声明为自动闭包，逃逸闭包
func myFunc2(closure: @autoclosure @escaping ()->Bool)  {
}
```

5.6　练习及解析

（1）编写一个计算阶乘的函数。

解析：

```
func funcOne(param:Int) -> Int {
    guard param>0 else{
        return 0
    }
    var tmp = param
    var result = 1
    while tmp>0 {
        result *= tmp
        tmp -= 1
    }
    return result
}
funcOne(param: 5)
```

（2）编写函数，其功能是：判断输入的字符是否为数字字符。如果是，则输出 true，否则输出 false。

解析：

```
func funcTwo(param:Character) -> Bool {
    if param <= "9" && param >= "0" {
        return true
    }else{
        return false
    }
}
funcTwo(param: "9")
```

（3）编写函数，其功能是：将两个两位数的正整数 a、b 合并成一个整数 c，合并规则是将 a 的十位和个位分别放在 c 的千位和个位，将 b 的十位和个位分别放在 c 的百位和十位。

解析：

```
func funcThree(param1:Int,param2:Int) -> Int {
    //param1 的个位数字
    let tmpa1 = param1%10
    //param1 的十位数字
    let tmpa2 = param1/10%10
    //param2 的个位数字
    let tmpb1 = param2%10
    //param2 的十位数字
    let tmpb2 = param2/10%10
    return tmpa2*1000+tmpb2*100+tmpb1*10+tmpa1
}
funcThree(param1: 45, param2: 12)
```

（4）编写函数，将字符串中的大写字母变成对应的小写字母，将小写字母变成对应的大写字母，其他字符不变。

解析：

```
func funcFour(param:String) -> String {
```

```
    var str = ""
    for char in param.characters {
        if char >= "a" && char <= "z" {
            //swift3.0
            str.append(String(char).uppercased())
        }else if char >= "A" && char <= "Z" {
             str.append(String(char).lowercased())
        }else{
            str.append(char)
        }
    }
    return str
}
funcFour(param: "How Are You?")
```

（5）编写函数，输入一个大于 0 的数字，将不大于这个数字的所有正奇数的和与正偶数的和以元组的形式返回。

解析：

```
func funcFive(param:Int) -> (Int,Int) {
    guard param>0 else {
        return (0,0)
    }
    //奇数和
    var sum1 = 0
    //偶数和
    var sum2 = 0
    var tmp = param
    while tmp>0 {
        if tmp%2==0 {
            sum2+=tmp
        }else{
            sum1+=tmp
        }
        tmp-=1
    }
    return (sum1,sum2)
}
funcFive(param: 10)
```

（6）编写函数，输入不定个数的一组整数值，统计其中正数和负数的个数，0 不纳入统计。

解析：

```
func funcSix(param:Int...) -> (Int,Int) {
    //正数个数
    var sum1 = 0
    //负数个数
    var sum2 = 0
    for index in param {
        if index>0 {
            sum1 += 1
```

```
        }else if index<0 {
            sum2 += 1
        }
    }
    return (sum1,sum2)
}
funcSix(param: 1,2,-1,-4,2,3,5,0,-12)
```

（7）编写函数，输入圆的半径，返回圆的周长和面积。

```
func funcSeven(param:Double) -> (Double,Double) {
    //周长
    let l = M_PI * 2 * param
    //面积
    let  s = M_PI * param * param
    return (l,s)

}
funcSeven(param: 3)
```

（8）编写函数，输入不定个数的整数，将其中的最大值和最小值返回。

```
func funcEight(param:Int...) -> (Int,Int) {
    return (param.maxElement()!,param.minElement()!)
}
funcEight(param: 1,2,-5,5,13,64,-8)
```

（9）使用闭包的风格模拟 Dictionary 数据的遍历。

解析：

```
//创建一个字典示例
let dic:Dictionary = [1:"1",2:"2",3:"3",4:"4",5:"5"]
//创建一个函数 通过闭包来传递遍历结果
func MyEnumDic(dic:Dictionary<Int,String>,closure:(_key:Int,_value:String)->Bool){
//遍历字典
    for item in dic {
        //执行闭包代码
        if closure(key: item.0,value: item.1) {
           //如果闭包返回值为 true 则中断遍历
            return
        }
    }
}
MyEnumDic(dic: dic) { (key, value) -> Bool in
    if key == 3 {
        //开发者自己控制中断遍历的逻辑
        print(value)
        return true
    }
    print(value)
    return false
}
```

本题中的代码设计十分巧妙，MyEnumDic(dic:,closure:)函数将每次遍历字典的结果传递给闭包，具体这些结果应怎样使用，完全交由闭包中的逻辑来做，并且通过闭包的返回值控制字典遍历是否结束，当开发者找到自己需要的值后，在闭包中返回 true 即可提前中断字典的遍历，提高代码的运行性能。

5.7 模 拟 面 试

（1）怎么理解函数？

回答要点提示：

① 从表面上看，函数其实是一组代码的组合，因此函数也被称为完成特殊功能的代码块。

② 函数的三要素是函数名、参数和返回值。

③ 在 Swift 语言中，函数的定义十分灵活，可以定义带默认值的函数，可以定义参数个数不定的函数，函数的参数名也可以自由地设置内部名称、外部名称甚至匿名。

④ 在定义函数时，可以将其理解为小功能单元，切记避免函数过于冗长。

核心理解内容：

熟练使用系统函数，熟练掌握函数的定义方法，理解函数参数和返回值的意义。

（2）什么是闭包？

回答要点提示：

① 闭包是 Swift 中的一种高级语法结构，闭包的核心是在其中使用的局部变量会被额外复制或引用，使这些变量脱离其作用域后依然有效。

② 闭包的功能与函数十分类似，其也是完成特定功能的代码块。可以将闭包当作对象使用，将其赋值给指定的变量，并且可以使用此变量直接调用闭包。

③ 和函数一样，闭包也有参数和返回值。

④ 闭包可以作为函数的参数或返回值。

⑤ 在 Swift 中，闭包有逃逸闭包与非逃逸闭包之分，对于逃逸闭包，函数内的闭包在函数外依然有效，对于非逃逸闭包，离开函数后闭包将失效。

⑥ Swift 语言中的闭包写法十分灵活，可以使用各种技巧来编写出非常简洁的闭包代码，例如后置闭包技巧、省略参数类型技巧、省略返回值类型技巧等。

核心理解内容：

在实际开发中，闭包的使用非常频繁，网络回调、自定义算法块、界面间传值等都会使用到闭包，加强练习掌握闭包知识是学习 Swift 方法的重中之重。

第6章

高级运算符与枚举

控制复杂性是计算机编程的本质。

——Brian Kernighan

本书在前面的章节主要针对 Swift 语言中的运算符进行介绍，除了算术运算符、逻辑运算符等基础的运算符外，Swift 语言还提供了许多关于运算符的高级使用技巧。开发者甚至可以重新实现系统的运算符或者自定义特殊功能的运算符。

枚举是 Swift 语言中一种略微复杂的数据类型。枚举和类、结构体一样，也是开发者可以进行定义的一种数据模型，熟悉 Objective-C 语言的读者知道，在 Objective-C 语言中枚举类型数据实际上就是一种整型数据，在 Swift 语言中则不同，枚举类型就是一种独立的数据类型。Swift 语言中的枚举语法很有特点，相比于 Objective-C 语言，Swift 语言中的枚举更灵活。比如：读者可以对枚举值设置原始值、相关值来扩展枚举的功能等。

通过本章，你将学习到：

- 位运算符的使用。
- 溢出运算符的意义。
- 对运算符进行重载操作。
- 自定义运算符。
- 枚举的创建与使用场景。
- 枚举原始值及相关值的应用。
- 递归枚举与递归函数的结合使用。

6.1 位运算符与溢出运算符

很多编程语言都支持位运算符，位运算符的主要功能是对二进制数据进行位运算等操作。而溢出运算符是 Swift 语言独有的，由于 Swift 语言对代码安全性的注重，正常的运算是不允许出现溢出行为的。开发者使用溢出运算符可以控制是否允许溢出运算。

6.1.1 位运算符的应用

在计算机中，数据都是以二进制的形式存储的，位运算也是专门针对二进制数据的一种运算方式。在 Swift 语言中，开发者在创建数值变量时可以通过追加“0b”前缀的方式将数值设置为二进制。使用 Xcode 开发工具创建一个命名为 AdvancedOperators 的 playground 文件，在其中进行代码演示。创建一个 UInt8 类型的变量，将十进制数 8 以二进制的方式赋值，示例如下：

```
//十进制数 8
var a:UInt8 = 0b1000
```

前面介绍过 UInt8 类型，其为 8 位的无符号整型，也就是说，任何一个 UInt8 类型的变量都是采用 8 个二进制位来存储数据的。因此读者可以理解为 a 变量实际存储的数据为 00001000。位运算的实质就是对数据的每一个二进制位进行逻辑运算。

Swift 语言支持 C/Objective-C 语言中的全部位运算符，其中包括按位取反运算、按位与运算、按位或运算、按位异或运算、按位左移运算以及按位右移运算。

按位取反运算符“~”的作用是将数据的每一位都进行取反操作，即如果当前位为 0，则运算后变为 1，如果当前位为 1，则运算后变为 0。对上面创建的变量 a 进行按位取反运算后，其存储的数据将变为 11110111，即十进制数 247，示例如下：

```
//运算后 a 的值变为十进制数 247
a = ~a
```

按位与运算符“&”需要有两个操作数，其作用是将两个操作数相同的位进行逻辑与运算。即如果两个对应位的值都为 1，则运算后此位结果为 1；如果其中有一个位的值为 0，则运算后此位结果为 0。示例如下：

```
//使用二进制数 11110000 与 a 进行按位与运算，运算结果为 11110000，即十进制数 240
a = 0b11110000&a
```

按位或运算符“|”需要有两个操作数，其作用是将两个操作数相同的位进行逻辑或运算。即如果两个对应位的值有一个为 1，则运算后此位结果为 1；如果两个对应位的值都为 0，则运算后此位结果为 0。示例如下：

```
//使用二进制数 11111111 与 a 进行按位或运算，结果为 11111111，即十进制数 255
a = 0b11111111|a
```

按位异或运算符“^”需要有两个操作数，其作用是将两个操作数相同的位进行逻辑异或运

算。即如果两个对应位的值相同，则运算后此位的值为 0；如果两个对应位的值不同，则运算后结果为 1。示例如下：

```
//使用二进制数 11110000 与 a 进行按位异或运算，结果为 00001111，即十进制数 15
a = 0b11110000^a
```

按位左移运算符“<<”用于将数据每一位上的值进行左移操作，示例如下：

```
//将 a 按位左移 1 位，结果为 00011110，即十进制数 30
a = a<<1
```

与按位左移运算对应，按位右移运算符“>>”用于将数据每一位上的值进行右移操作，示例如下：

```
//将 a 按位右移 1 位，结果为 00001111，即十进制数 15
a = a>>1
```

提　示

进行按位左移或者右移运算的时候，有可能出现丢失数据位的情况。例如，对于 UInt8 类型，将二进制数据 00001111 向左移动 6 位结果为 11000000，同样，将二进制数据 11110000 向右移动 6 位结果为 00000011。

6.1.2　溢出运算符

Swift 语言十分注重安全性，在编写代码时，如果出现了数据溢出，就会直接出现运行时错误。而溢出运算符这种设计将代码的不确定性降低了很多。所谓溢出，是指超出数据所属类型的最大值或者最小值。以 UInt8 来说，其最大值为二进制数 11111111，即十进制数 255。如果将 UInt8 类型的变量设置为 255 后再让其加 1，就会出现数据溢出，示例如下：

```
var b:UInt8 = 255
//代码运行到这里会出现错误，运行直接中断
b = b+1
```

如果开发者确实需要溢出操作而不是无意留下的小错误，Swift 语言中也提供了支持溢出操作的溢出操作符，示例如下：

```
var b:UInt8 = 255
//进行支持溢出的加操作，b 的值将变为 0
b = b &+ 1
//进行支持溢出的减操作，b 的值将变为 255
b = b &- 1
//进行支持溢出的乘操作，b 的值将变为 254
b = b &* 2
```

提　示

对二进制数据进行乘 2 运算，实质就是对二进制数据进行左移一位的运算。例如，二进制数据 11111111*2=11111110。

6.2 运算符的重载与自定义

准确来说，重载运算符和自定义运算符都是开发者自定义运算符功能的手段，二者的差别在于重载运算符是给系统已经存在的运算符追加新的功能，而自定义运算符是完全自定义一个系统不存在的运算符来实现特定的运算功能。

6.2.1 重载运算符

读者在认识重载运算符前，首先应该清楚重载的概念。重载的概念最初是针对函数的，对同一个函数名，设置不同的参数类型以实现不同的功能被称为函数的重载。在 Objective-C 中函数参数的名称是包含在函数名里的，因此从严格意义上讲，Objective-C 语言并不存在函数的重载操作。Swift 语言则不同，我们可以通过对函数重载应用的一个小例子来理解重载的意义。

实现一个整数的加法函数十分简单，示例如下：

```
func addFuncInt(param1:Int,param2:Int)->Int{
    return param1+param2
}
addFuncInt(param1: 3, param2: 4)
```

上面的示例代码用来进行整型数据的加法是完全可以的，但是如果需要进行浮点型数据的加法就会出现问题，开发者如果直接将浮点数传入 addFunc()函数中，编译器就会直接报类型错误，这时你可能想要创建一个针对浮点数的加法运算，示例如下：

```
func addFuncDouble(param1:Double,param2:Double)->Double{
    return param1+param2
}
addFuncDouble(param1: 0.5, param2: 0.3)
```

这样是解决了问题，但是这种设计思路十分糟糕，实现相同功能的函数，由于参数的不同被生生切成了两个，其实开发者可以使用相同的函数名 addFunc()，通过重载实现不同类型参数的计算，示例如下：

```
//创建整型的加法函数
func addFunc(param1:Int,param2:Int) -> Int {
    return param1+param2
}
//重载 addFunc 实现浮点型数据的加法
func addFunc(param1:Double,param2:Double) -> Double {
    return param1+param2
}
//再次重载 addFunc  实现字符串类型的加法
func addFunc(param1:String,param2:String)->String{
    return param1+param2
}
```

如上代码，通过重载的方式对不同数据类型实现了加法操作，并且在调用加法函数的时候，开发者只需要记住这一个函数名即可，这就大大增强了代码的统一性。

类比于函数的重载，运算符的重载是指在系统已经存在的运算符上扩展新的功能。其实在前面章节中使用的加号运算符“+”就是通过重载实现的，开发者可以直接使用“+”运算符进行整型数据、浮点型数据甚至字符串类型数据的相加操作。下面我们通过自定义一个圆形的类，通过重载加号运算符“+”来实现支持对圆形类实例的相加操作。

设计圆形类如下，其中有两个属性，分别表示圆形半径与圆心：

```
class Circle{
    //圆心
    var center:(Double,Double)
    //半径
    var radius:Double
    init(center:(Double,Double),radius:Double){
        self.center = center
        self.radius = radius
    }
}
```

定义两个Circle实例进行相加操作时，应执行这样的运算：两个Circle实例相加返回一个新的Circle 实例，并且这个新的 Circle 实例的圆心为第一个 Circle 操作数的圆心，新的 Circle 实例的半径为两个操作数 Cirlcle 实例半径的和，重载加法运算符如下：

```
func +(param1:Circle,param2:Circle) -> Circle {
    return Circle(center: param1.center, radius: param1.radius+param2.radius)
}
```

可以发现，重载运算符的语法格式与函数十分相似。实际上，运算符就是通过函数的方式定义的。

提　示

在某些场景下，运算符也的确可以像函数一样来使用，例如在一个函数参数中传入闭包时，也可以直接传入某个功能类型的运算符，示例如下：

```
func myFunc(closure:(Circle,Circle)->Circle) {
}
//将重载的加法运算符传入
myFunc(closure: +)
```

注　意

Swift 语言中还有一个覆写的概念。覆写是指子类对父类中的属性和方法进行适合自身的重新实现，和重载意义完全不同，读者在后面的学习中会遇到，注意不要将这两个概念混淆。

6.2.2 自定义运算符

重载运算符是为已经存在的系统运算符扩展新的功能。开发者也可以通过自定义系统不存在的运算符来实现特殊的需求，例如 Swift 语言从 2.2 版本开始移除了“++”“--”运算符，这里我们可以通过自定义运算符来添加自加运算符“++”，示例如下：

```
//自定义前缀运算符++
prefix operator ++
//进行自定义运算符实现
prefix func ++(param:Int)->Int{
    return param+1
}
```

自定义运算符分为两个步骤，首先开发者需要将要定义的运算符进行声明，如上代码中的 prefix operator ++。在声明运算符的结构中，prefix 的作用是运算符的类型，可以使用 prefix 关键字将其声明为前缀运算符，也可以使用 infix 关键字将其声明为中缀运算符、postfix 关键字将其声明为后缀运算符。在进行运算符的实现时，后缀和前缀运算符只能有一个参数，参数在 func 关键字前需要表明要实现的运算符类型，而中缀运算符需要有两个参数且 func 关键字前不需要额外标明，示例如下：

```
//自定义前缀运算符++
prefix operator ++
//进行自定义运算符实现
prefix func ++(param:Int)->Int{
    return param+1
}
//将返回 6
++5
//自定义中缀运算符
infix operator ++
func ++(param1:Int,parma2:Int)->Int{
    return param1*param1+parma2*parma2
}
//将返回 41
5++4
//自定义后缀运算符
postfix operator ++
postfix func ++(param1:Int) -> Int {
    return param1+param1
}
//将返回 10
5++
```

提　示

前缀运算符是指在只有一个操作数且在使用运算符进行运算时，运算符需要出现在操作数的前面；中缀运算符需要有两个操作数，且在进行运算时运算符需要出现在两个操作数的中间；后缀运算符只能有一个操作数，在运算时后缀运算符需要出现在操作数的后面。

需要注意，Swift 语言中提供了许多 Unicode 字符，可用于运算符的自定义，但是也有一些规则，自定义运算符常使用如下字符作为开头：/、=、-、+、！、*、%、<、>、&、|、^、?、~。开发者也可以使用点“.”来进行运算符的定义。当开发者的自定义运算符中有使用到符号“.”的时候需要注意：如果“.”出现在自定义运算符的开头，则运算符中可以出现多个符号“.”，例如“.+.”；如果自定义运算符中的符号“.”不在开头，那么这个自定义运算符中只允许出现一个符号“.”。

提　示

Swift 语言中也有一些保留符号，它们不可以单独被重载和自定义。保留符号为=、->、//、/*、*/、.、<、>、&、?、!。

6.3　运算符的优先级与结合性

在小时候学习数学时，老师总会强调四则运算中的先乘除后加减、从左向右计算等规则。其实在 Swift 语言编程中，也有这样的规则存在。例如，进行如下混合运算：

```
/*
 运算结果 23
 过程如下：
 2*10 = 20
 20*3 = 60
 60/4 = 15
 8+15 = 23
 */
8+2*10*3/4
```

从上面演示的代码中可以看出，Swift 语言中的四则运算也是先进行乘除运算后进行加减运算，运算顺序为从左向右。其实在 Swift 语言的运算符体系中，有着优先级与结合性的概念，运算符的优先级决定同一行代码中出现多种运算符时的计算顺序，运算符的结合性决定运算符是从左向右运算还是从右向左运算。任何运算符都有默认的优先级，开发者自定义的运算符也是如此，优先级高的运算符优先执行。对于结合性而言，由于前缀运算符与后缀运算符都只有一个操作数，因此它只对中缀运算符有意义。

表 6-1 和表 6-2 列出了 Swift 中所有运算符的相关信息。

表 6-1　Swift 语言中的系统前缀运算符

运算符	功能	运算符	功能
！	逻辑非运算	+	取正运算
~	按位取反运算	-	取负运算

表 6-2 Swift 语言中的系统中缀运算符

运算符	功能	结合性	优先级
<<	按位左移运算	无	160
>>	按位右移运算	无	160
*	乘法运算符	左结合	150
/	除法运算符	左结合	150
%	取余运算符	左结合	150
&*	溢出乘法运算符	左结合	150
&	按位与运算符	左结合	150
+	加法运算符	左结合	140
-	减法运算符	左结合	140
&+	溢出加法运算符	左结合	140
&-	溢出减法运算符	左结合	140
|	按位或运算符	左结合	140
^	按位非运算符	左结合	140
..<	右开区间运算符	无	135
…	闭区间运算符	无	135
is	类型检查运算符	左结合	132
as，as?，as!	类型转换运算符	左结合	132
??	空合并运算符	右结合	131
<	小于运算符	无	130
<=	小于等于运算符	无	130
>	大于运算符	无	130
>=	大于等于运算符	无	130
==	等于运算符	无	130
!=	不等于运算符	无	130
===	等同运算符	无	130
!==	不等同运算符	无	130
~=	模式匹配运算符	无	130
&&	逻辑与运算符	左结合	120
||	逻辑或运算符	左结合	110
?:	条件运算符	右结合	100
=	赋值运算符	右结合	90
*=	复合乘法赋值运算符	右结合	90
/=	复合除法赋值运算符	右结合	90
%=	复合取余赋值运算符	右结合	90
+=	复合加法赋值运算符	右结合	90
-=	复合减法赋值运算符	右结合	90
<<=	复合按位左移赋值运算符	右结合	90
>>=	复合按位右移赋值运算符	右结合	90
&=	复合按位与运算赋值运算符	右结合	90

（续表）

运算符	功能	结合性	优先级
|=	复合按位或运算赋值运算符	右结合	90
^=	复合按位异或运算赋值运算符	右结合	90
&&=	复合逻辑与运算赋值运算符	右结合	90
||=	复合逻辑或运算赋值运算符	右结合	90

上面两个表格中列举了 Swift 语言系统定义的所有运算符相关信息，无须专门记忆，在实际开发中需要使用时再来查表即可。其实更多情况下，开发者会直接使用小括号来决定表达式的执行顺序，这样代码也会更加直观。

在重载运算符操作时，并不会改变原运算符的结合性和优先级，但对于自定义运算符，开发者可以设置其结合性与优先级，示例如下：

```
infix operator ++{associativity left precedence 140}
```

associativity 关键字用于声明运算符的结合性，可以选择 left 或者 right 来定义成左结合性或者右结合性，precedence 关键字用于声明运算符的优先级。

6.4　枚举类型的创建与应用

Swift 语言中使用 enum 关键字来进行枚举的创建，使用 Xcode 开发工具创建一个命名为 Enum 的 playground 文件，在其中创建一个姓氏类型的枚举，如下所示：

```
//创建一个姓氏类型枚举
enum Surname {
    //使用 case 进行枚举值的定义
    case 张
    case 王
    case 李
    case 赵
}
```

上面的代码创建了一个姓氏枚举类型，这个枚举类型中定义了 4 个枚举值，分别是张、王、李、赵，上面的写法将 4 个枚举值分别在 4 个 case 语句中定义，开发者也可以在 1 个 case 子句中完成多个枚举值的定义，示例如下：

```
//创建一个姓氏枚举类型
enum Surname {
    //在一条 case 语句中定义多个枚举值
    case 张,王,李,赵
}
```

在使用时，枚举和其他类型一样，开发者可以在声明变量时将变量的类型指定为某个枚举类型，也可以通过对变量初始化来使编译器自动推断出变量的类型。枚举中定义的枚举值在使用时，开发者可以使用点语法来获取，示例如下：

```
//创建一个姓氏枚举类型的变量
var sur:Surname
//对 sur 变量进行赋值
sur=Surname.张
```

实际上，如果一个变量的类型已经确认为某个枚举类型，那么开发者再进行变量赋值的时候是可以将枚举类型省略掉的，直接使用点语法获取枚举值即可，示例如下：

```
//对 sur 进行修改
sur = .王
```

在开发中，枚举类型会经常与 switch-case 结合使用以实现选择结构，这种方式实现的选择结构代码清晰统一，对于开发者来说十分有益，示例如下：

```
//创建一个姓氏枚举类型的变量
var sur:Surname
//对 sur 变量进行复制
sur=Surname.张
//对 sur 进行修改
sur = .王
//对枚举类型的变量进行 switch 选择结构
switch sur {
    case .张:
        print("姓氏张")
    case .王:
        print("姓氏王")
    case .李:
        print("姓氏李")
    case .赵:
        print("姓氏赵")
}
```

6.5 枚举的原始值与相关值

枚举的原始值特性可以将枚举值与另一种数据类型进行绑定，相关值则可以为枚举值关联一些其他数据。通过相关值，开发者可以实现复杂的枚举类型。

6.5.1 枚举的原始值

上节中创建的枚举其实并没有声明一个原始值类型，Swift 语言中的枚举支持开发者声明一个原始类型，并将某个已经存在的类型的值与枚举值进行绑定，枚举指定原始值类型的语法与继承的语法有些类似，示例如下：

```
//为枚举类型指定一个原始值类型
enum CharEnum:Character{
    //通过赋值的方式来为枚举值设置一个原始值
```

```
    case a = "a"
    case b = "b"
    case c = "c"
    case d = "d"
}
```

如果开发者要指定枚举的原始值类型为 Int 类型，也可以只设置第一个枚举值的原始值，其后的枚举值的原始值会在第一个枚举值原始值的基础上依次递增，示例如下：

```
enum IntEnum:Int {
    //第一个枚举值的原始值设置为 1
    case 一 = 1
    //默认原始值为 2
    case 二
    //默认原始值为 3
    case 三
    //默认原始值为 4
    case 四
}
```

通过枚举类型中的 rawValue 属性来获取枚举的原始值，示例如下：

```
//创建枚举变量
var char = CharEnum.a
//获取 char 枚举变量的原始值 "a"
var rawValue = char.rawValue
```

在枚举变量初始化时，开发者可以使用枚举类型加点语法的方式，如果这个枚举有指定的原始值，也可以通过枚举值的原始值来完成枚举实例的构造，示例如下：

```
//通过原始值构造枚举变量 一
var intEnum = IntEnum(rawValue: 1)
```

需要注意，通过原始值进行枚举实例的构造时是有可能构造失败的，因为开发者传入的原始值不一定会对应到某一个枚举值。因此这个方法实际上返回的是一个 Optional 类型的可选值，如果构造失败，则会返回 nil。

6.5.2　枚举的相关值

Swift 语言在很多方面的设计都比其他编程语言更加灵活与现代，枚举的相关值语法最能够体现这一特点。

枚举类型的设计思路是帮助开发者将一些简单的同类数据进行整合。举个例子，在游戏类软件的开发中经常会使用到各种物理模型，以形状为例，开发者通常会定义一系列的枚举值作为物理形状的枚举，如圆形、三角形、矩形等，示例如下：

```
//定义形状枚举
enum Shape {
    //圆形
    case circle
```

```
        //矩形
        case rect
        //三角形
        case triangle
    }
```

上面的代码进行了形状的定义，但是有一个问题，这种枚举值的定义方式只适合简单数据类型的定义，而不同的形状可能需要不同的参数。例如圆形需要圆心和半径来确定，矩形需要中心点与宽高来确定，三角形需要 3 个顶点来确定。如果对枚举类型进行实例化，可以根据不同的形状设置不同的参数，那么在使用时对开发者来说将十分方便，在 Swift 语言中，对枚举设置相关值就可以完成这样的需求。

在定义枚举值的时候，开发者可以为其设置一个参数列表，这个参数列表被称为枚举的相关值，示例如下：

```
    //定义形状枚举
    enum Shape {
        //圆形 设置圆心和半径为相关值
        case circle(center:(Double,Double),radius:Double)
        //矩形 设置中心、宽、高为相关值
        case rect(center:(Double,Double),width:Double,height:Double)
        //三角形 设置 3 个顶点为相关值
        case triangle(point1:(Double,Double),point2:(Double,Double),
point3:(Double,Double))
    }
```

在创建有相关值枚举的时候，开发者需要提供参数列表中所需要的参数，示例如下：

```
    //创建圆形枚举实例 此圆的圆心为(0,0)，半径为 3
    var circle = Shape.circle(center: (0, 0), radius: 3)
    //创建矩形枚举实例 此矩形的中心点为(1,1)，宽度为 10，高度为 15
    var rect = Shape.rect(center: (1, 1), width: 10, height: 15)
    //创建三角形枚举实例 此三角形的 3 个顶点为(2,2),(3,3),(2,5)
    var triangle = Shape.triangle(point1: (2, 2), point2: (3, 3), point3: (2, 5))
```

在 switch-case 结构语句中，匹配到枚举后，可以通过参数捕获的方式来获取枚举实例的相关值，这里捕获到的相关值参数可以在开发者的代码中使用，示例如下：

```
    //写一个匹配函数 参数为 Shape 枚举类型
    func shapeFunc(param:Shape){
        switch param {
            //进行参数捕获
        case let .circle(center,radius):
            print("此圆的圆心为：\(center)，半径为：\(radius)")
        case let .rect(center,width,height):
            print("此矩形的中心为：\(center)，宽为：\(width)，高为：\(height)")
        case let .triangle(point1,point2,point3):
            print("此三角形的 3 个顶点分别为：\(point1)，\(point2)，\(point3)")
        }
    }
    shapeFunc(param: circle)
    shapeFunc(param: rect)
```

```
shapeFunc(param: triangle)
```

6.5.3　递归枚举

递归枚举是 Swift 语言枚举相关语法中比较难于理解的一个语法，但是如果可以将其完全掌握，则可以编写出结构十分优美的代码。要完全明白递归枚举的意义与使用，首先需要明白两点——递归与枚举实质。

递归是一种代码算法技巧，并不区分语言，各种高级语言都可以实现自己的递归算法。简单来说，递归就是程序调用自身的编程技巧。针对函数来说，递归函数就是在函数内部进行了此函数本身的调用。读者需要注意一点，递归算法效率十分高，但是其性能资源的耗费也十分严重。在大多数情况下，开发者应该尽量避免使用递归。前面的章节中曾经使用过循环结构来计算正整数的阶乘，例如 5!=5*4*3*2*1=120，这里我们要使用递归算法来实现一个正整数的阶乘，代码如下：

```
//使用递归算法来实现计算正整数的阶乘
func mathsFunc(param:Int)->Int{
    let tmp = param-1
    if tmp>0 {
        //递归
        return mathsFunc(param: tmp) * param
    }else{
        return 1
    }
}
mathsFunc(param: 5)
```

函数的功能是进行数据计算，递归函数只是使用递归的算法来进行数据的计算。枚举则不同，枚举的功能是数据的描述。例如 6.5.2 节中创建的形状枚举，其中只是对几种形状的数据结构进行描述和定义，它并不具有数据计算的功能。递归枚举其实就是使用递归的方式来进行数据描述。

使用枚举描述加、减、乘、除四则表达式示的例代码如下：

```
//使用枚举来模拟加减乘除四则运算
enum Expression {
    //表示加法运算 两个相关值 param1 与 param2 代表进行加法运算的两个参数
    case add(param1:Int,param2:Int)
    //表示减法运算 两个相关值 param1 与 param2 代表进行减法运算的两个参数
    case sub(param1:Int,param:Int)
    //表示乘法运算 两个相关值 param1 与 param2 代表进行乘法运算的两个参数
    case mul(param1:Int,param2:Int)
    //表示除法运算 两个相关值 param1 与 param2 代表进行除法运算的两个参数
    case div(param1:Int,param2:Int)
}
```

使用上面创建的枚举来描述四则运算表达式，示例如下：

```
//表示表达式 5+5
var exp1 = Expression.add(param1: 5, param2: 5)
//表示表达式 10-5
var exp2 = Expression.sub(param1: 10, param2: 5)
```

```
//表示表达式 5*5
var exp3 = Expression.mul(param1: 5, param2: 5)
//表示表达式 10/2
var exp4 = Expression.div(param1: 10, param2: 2)
```

这里读者需要注意，变量 exp1、exp2、exp3、exp4 只是四则运算表达式的描述，并没有运算功能。可以简单地理解为：Expression 枚举模拟的是一种四则运算表达式类型，如果要进行运算，开发者还需要实现具体的功能函数。

可以发现，Expression 能够描述的表达式只是单运算表达式，不能够进行复合表达式的描述，例如对于((5+5)*2-8)/2 表达式的描述。分析这类复合表达式，其实质只是将单运算表达式作为计算的参数传入另一个单运算表达式。类比于 Swift 语言中的枚举，一个枚举值的相关值类型可以设置为这个枚举本身的类型，通过这种递归的方式就可以实现复合表达式的描述，将前面创建的 Expression 枚举修改如下：

```
//使用枚举来模拟加减乘除四则运算
enum Expression {
    //描述单个数字
    case num(param:Int)
    //表示加法运算 将 Expression 作为相关值参数类型
    indirect case add(param1:Expression,param2:Expression)
    //表示减法运算
    indirect case sub(param1:Expression,param2:Expression)
    //表示乘法运算
    indirect case mul(param1:Expression,param2:Expression)
    //表示除法运算
    indirect case div(param1:Expression,param2:Expression)
}
```

使用 indirect 关键字修饰的枚举值表示这个枚举值是可递归的，即此枚举值中的相关值可以使用其枚举类型本身。使用修改后的 Expression 枚举来描述复合表达式((5+5)*2-8)/2 的代码如下：

```
//创建单值 5
var num5 = Expression.num(param: 5)
//进行表达式 5+5 描述
var exp1 = Expression.add(param1: num5, param2: num5)
//创建单值 2
var num2 = Expression.num(param: 2)
//进行表达式(5+5)*2 的描述
var exp2 = Expression.mul(param1: exp1, param2: num2)
//创建单值 8
var num8 = Expression.num(param: 8)
//进行表达式(5+5)*2-8 的描述
var exp3 = Expression.sub(param1: exp2, param2: num8)
//进行表达式((5+5)*2-8)/2 的描述
var expFinal = Expression.div(param1: exp3, param2: num2)
```

最后得到的变量 expFinal 就是对((5+5)*2-8)/2 的描述。另外，读者可以为这四则表达式枚举类型 Expression 实现一个函数来进行运算，在开发中将描述与运算结合，能够编写出十分优美的代码，处理递归枚举通常会采用递归函数，函数方法实现示例如下：

```
//这个递归函数的作用是将 Expression 描述的表达式进行运算 结果返回
func expressionFunc(param:Expression) -> Int {
    switch param {
        //单值直接返回
    case let .num(param):
        return param
    case let .add(param1, param2):
        //返回加法运算结果
       return expressionFunc(param: param1)+expressionFunc(param: param2)
    case let .sub(param1, param2):
        //返回减法运算结果
        return expressionFunc(param: param1)-expressionFunc(param:param2)
    case let .mul(param1, param2):
        //返回乘法运算结果
        return expressionFunc(param: param1)*expressionFunc(param: param2)
        //返回除法运算结果
    case let .div(param1, param2):
        return expressionFunc(param: param1)/expressionFunc(param: param2)
    }
}
//进行((5+5)*2-8)/2 运算 结果为 6
expressionFunc(param: expFinal)
```

关于递归枚举还有一点需要注意，如果一个枚举中所有的枚举值都是可递归的，开发者可以直接将整个枚举类型声明为可递归的，示例如下：

```
//使用枚举来模拟加减乘除四则运算
indirect enum Expression {
    //描述单个数字
    case num(param:Int)
    //表示加法运算 将 Expression 作为相关值参数类型
    case add(param1:Expression,param2:Expression)
    //表示减法运算
    case sub(param1:Expression,param2:Expression)
    //表示乘法运算
    case mul(param1:Expression,param2:Expression)
    //表示除法运算
    case div(param1:Expression,param2:Expression)
}
```

6.6　练习及解析

（1）模拟 C 语言通过自定义运算符的方式实现前缀自增、前缀自减、后缀自增、后缀自减运算符。

解析：

```
//定义前缀自增运算符
```

```
prefix operator ++
//定义后缀自增运算符
postfix operator ++
//定义前缀自减运算符
prefix operator --
//定义后缀自减运算符
postfix operator --
//进行实现
prefix func ++( param: inout Int)->Int{
    param+=1
    return param
}
postfix func ++(param: inout Int)->Int{
    param+=1
    return param-1
}
prefix func --( param: inout Int)->Int{
    param-=1
    return param
}
postfix func --(param: inout Int)->Int{
    param-=1
    return param+1
}
```

（2）Swift 语言中的加法运算符不能支持对区间范围的相加操作，重载加法运算符，使其支持区间的追加，例如(0…5)+5 计算后的结果为区间 0…10。

解析：

```
func +(param:ClosedRange<Int>,param2:Int)->ClosedRange<Int>{
    return param.lowerBound...param.upperBound+param2

}
//将得到 0…10
var newRange = 0...5+5
```

（3）自定义新后缀运算符“*!”，其功能是对某个数进行阶乘计算。

解析：

```
postfix operator *!{}
postfix func *! (param:Int)->Int{
    var result = 1
    var tmp = param
    while tmp>0 {
        result *= tmp
        tmp-=1
    }
    return result
}
//得到计算结果为 120
5*!
```

（4）模拟设计一个交通工具枚举，将速度与乘坐价钱作为枚举的相关值。

解析：

```
enum Transport{
    case car(price:Int,speed:Float)
    case boat(price:Int,speed:Float)
    case airport(price:Int,speed:Float)
}
//创建一个汽车交通工具，价钱为 2，速度为 80
var car = Transport.car(price: 2, speed: 80)
```

6.7 模 拟 面 试

（1）Swift 语言中有“++”和“--”运算符吗？

回答问题要点：

① 在 Swift 的初期版本中是有“++”和“--”这两个运算符的，在 Swift 2.2 版本之后这两个运算符被移除了。

② 自增和自减运算符是类 C 语言中非常烦琐的一个运算符，很多时候这两个运算符对初学者造成了很大的困扰，编写的代码易读性差并且所实现的功能完全可以用其他方式实现。

③ 虽然 Swift 的原生框架中将这两个运算符移除了，但是如果真的需要，开发者依然可以使用 Swift 中的自定义运算符技术来重新实现这两个运算符。

核心理解内容：

了解“++”和“--”运算符在类 C 语言中的简单用法和作用。熟练使用 Swift 语言中的自定义运算符技术。

（2）怎样理解枚举？Swift 中的枚举有怎样的特别之处？

回答问题要点：

① 枚举也是一种数据类型，数据类型的作用就是用来描述数据，枚举通常用来描述一组简单的、属性一致的数据。

② 和其他编程语言不同，枚举在 Swift 中被设计得非常强大，其可以通过继承于某个数据类型来为每一个枚举值指定原始值，其也可以在定义枚举值时定义一组与之有联系的相关值。通过相关值，枚举可以描述数据的灵活性大大增强了。

核心理解内容：

理解枚举的基本用法，熟悉原始值与相关值的意义，能够简单使用递归枚举的技巧编写代码。