第3章 运 算 符

运算符是一种特殊的符号。通过运算符，可以实现对变量或常量的值进行操作，包括值的检查、值的改变及值的合并。本章从最基本的赋值运算符开始，依次介绍算术运算符、关系运算符、逻辑运算符、三元运算符以及区间运算符。

3.1 赋值运算符

赋值运算符“=”用来初始化或者改变一个变量的值。如图 3.1 所示，变量 str 通过赋值运算符初始化为字符串"Hello，playground"。变量 newStr 通过赋值运算符初始化为变量 str 的值。

```
var str = "Hello, playground"                "Hello, playground"
var newStr = str                             "Hello, playground"

let (a,b,c) = (1,2,3)

print(a)                                     "1\n"
print(b)                                     "2\n"
print(c)                                     "3\n"
```

图 3.1 赋值运算符

元组也可以通过赋值运算符，对其中的所有元素一次性赋值。如图 3.1 所示，元组(a，b，c)被初始化赋值为(1，2，3)，相当于 a、b、c 分别被赋值为 1、2、3。

3.2 算术运算符

Swift 中所有的数值类型都支持最基本的四则运算：加(+)、减(-)、乘(*)、除(/)。图 3.2 定义了两个整型变量 a 和 b，然后打印出 a 和 b 进行四则运算的结果。字符串类型也可以相加，从而合并成一个新的字符串，通常称之为拼接运算。

```
var a : Int = 8                          8
var b : Int = 2                          2

print(a+b)                               "10\n"
print(a-b)                               "6\n"
print(a*b)                               "16\n"
print(a/b)                               "4\n"

var str1 : String = "Hello,"             "Hello,"
var str2 : String = "world!"             "world!"

print(str1+str2)                         "Hello,world!\n"
```

图 3.2 加、减、乘、除运算符

3.2.1 求余运算符

求余运算符“%”也叫取模运算符，其计算两个数相除后的余数。只支持整型求余运算，如图 3.3 所示。

```
var a : Int = 9                          9
var b : Int = 2                          2

a % b                                    1
```

图 3.3 求余运算符

3.2.2 自增自减运算符

Swift 4 中去掉了以前版本中的自增运算符“++”和自减运算符“--”，取而代之的是“+=”和“-=”。原来的自增/自减运算符是从 C 语言中搬过来，使用起来比较灵活，初学者较难理解。另外，自增运算符 “++” 主要用于 for 循环语句中。而在 Swift 4 中，for-in 循环结构已经不再需要使用“++”了。

Swift 4 中的自增/自减运算符示例如图 3.4 所示。

```
var a = 3                    3

a += 1                       4
let b  = a                   4

a -= 1                       3
let c = a                    3
```

图 3.4　Swift 4 中的自增/自减运算符示例

3.3　关系运算符

关系运算符就是用来比较两个值之间关系的运算符，运算结果为布尔值，即 true 或者 false。关系运算符包括等于(==)、不等于(!=)、大于(>)、小于(<)、大于或等于(>=)、小于或等于(<=)。如图 3.5 所示，定义变量 a、b，并赋初值，然后进行各种关系运算，运算结果显示在右侧。关系运算符的运算结果为布尔值，所以关系运算符适用于条件语句。

```
var a = 3, b = 6

a == b                       false
a != b                       true
a >  b                       false
a <  b                       true
a >= b                       false
a <= b                       true

if a == b {
    print("a is \(a), b is \(b), they are equal.")
} else {
    print("a is \(a), b is \(b), they are not
        equal!")                 "a is 3, b is 6, they are not equal!\n"
}
```

图 3.5　关系运算符

3.4 逻辑运算符

逻辑运算符包括与(&&)、或(||)、非(!),操作数为布尔型。

逻辑非是一元运算符,即操作数为1个,其真值表见表3.1。

表3.1 逻辑非运算真值表

a的布尔值	!a的布尔值
true	false
false	true

图3.6定义了电灯开关的状态变量on,初始值为false,即表示灯关着。在if语句中,通过!on判断灯是否关着,如果是,则打印相应信息。

```
var on : Bool = false                    false

if !on {
    print("the light is off")            "the light is off\n"
}
```

图3.6 逻辑非运算符

逻辑与是二元运算符,操作数为2个,其真值表为表3.2。

表3.2 逻辑与运算符真值表

a的布尔值	b的布尔值	a&&b的布尔值
true	true	true
true	false	false
false	true	false
false	false	false

图3.7定义了控制电灯开关的函数,需要传入两个参数,分别为布尔变量sunshine(表示是否为白天)和indoors(表示是否家里有人)。表示电灯开关的布尔变量为on,on为true的条件是没有阳光,并且家里有人。将sunshine赋值为false,表示没有阳光。同时,将indoors赋值为true,表示家里有人。调用函数lightControl(),

将变量 sunshine 和 indoors 传入，控制台的输出结果为“turn light on”，即打开电灯。

```
var on : Bool = false

var sunshine : Bool
var indoors : Bool

func lightControl(sunshine:Bool,indoors:Bool) {

    on = (!sunshine)&&indoors

    if !on {
        print("turn light off")
    } else {
        print("turn light on")
    }
}
sunshine = false
indoors = true
lightControl(sunshine:sunshine, indoors:indoors)
```

```
false

true

"turn light on\n"

false
true
```

图 3.7　电灯开关控制(逻辑与)

逻辑或是二元运算符，操作数是 2 个，真值表见表 3.3。

表 3.3　逻辑或运算符真值表

a 的布尔值	b 的布尔值	a||b 的布尔值
true	true	true
true	false	true
false	true	true
false	false	false

如图 3.8 所示，用逻辑或实现电灯开关的功能，需要对前面的代码做如下修改：将变量名 on 改为 off，当 off 为 true 的时候表示应该关灯。将 on 的逻辑表达式改为 off 的逻辑表达式，它表示有阳光的时候，或者没人在家的时候，将灯关了。程序运行的输出结果完全一致。可以看出，逻辑与和逻辑或的表达式是可以进行逻辑等价转换的。

```
var off : Bool = false                                          false

var sunshine : Bool
var indoors : Bool

func lightControl(sunshine:Bool,indoors:Bool) {

    off = sunshine || (!indoors)                                false

    if off {
        print("turn light off")
    } else {
        print("turn light on")                                  "turn light on\n"
    }
}
sunshine = false                                                false
indoors = true                                                  true
lightControl(sunshine:sunshine, indoors:indoors)
```

图 3.8　电灯开关控制(逻辑或)

3.5　三元运算符

前面介绍了多个一元运算符和二元运算符,本节介绍一种三元运算符,即三元条件运算符。

三元条件运算符有 3 个操作数,格式为

```
question ? answer1 : answer2
```

其语义为:如果 question 为 true,则返回 answer1;否则返回默认值 answer2。

实际上,三元运算符是一种简写方式,其语义与下面的 if 语句等价。

```
if question: {
    answer1
} else {
    answer2
}
```

图 3.9 定义了 3 个常量 pageHeight、contentHeight、bottomHeight,分别表示页面高度、正文内容高度、底部栏高度,另外还定义了一个布尔型变量 hasHeader,用来表示页面中是否要包含一个头部栏。页面高度由正文内容高度、底部栏高度和头部栏

高度相加得到。如果 hasHeader 为 true 时，头部栏按照 30 算，否则头部栏按照 10 算。这就可以通过三元条件运算符便捷地实现。

```
let contentHeight = 100
let bottomHeight = 20
var hasHeader = true
let pageHeight = contentHeight + bottomHeight +
    (hasHeader ? 30 : 10)
```

```
100
20
true
150
```

图 3.9　三元条件运算符

三元条件运算符可以简洁地表示有条件的二选一语义，但是三元条件运算符的缺点是可读性较差。因此，在表达复杂的组合逻辑时，尽量不使用三元条件运算符。

3.6　区间运算符

区间运算符包括闭区间运算符和半闭区间运算符。

闭区间运算符 a...b 表示一个从 a 到 b 的所有值的区间(包括 a 和 b)。闭区间运算符在循环语句中很有用。如图 3.10 所示，在 for-in 循环中，变量 i 取 0～3 的值(包括 0、1、2、3)，循环执行 4 次。

半闭区间运算符 a..＜b 表示一个从 a 到 b 的所有值的区间(包括 a，但不包括 b)。如图 3.11 所示，在 for-in 循环中，变量 i 取值为 0、1、2，循环执行 3 次。

```
for i in 0...3 {
    print(i)
}
```

```
(4 times)
```

图 3.10　闭区间运算符

```
for i in 0..<3 {
    print(i)
}
```

```
(3 times)
```

图 3.11　for-in 循环

本章知识点

本章介绍了 6 类常用的运算符，分别为赋值运算符、算术运算符(加、减、乘、除、求余、自增、自减)、关系运算符(等于、不等于、大于、小于、大于或等于、小于或等于)、逻辑运算符(与、或、非)、三元运算符以及区间运算符(闭区间运算符和半闭区间运算符)。

第4章 字 符 串

字符串是一组有序字符的集合。字符串在移动应用中是一个非常重要的数据类型，在与用户交互的过程中，绝大多数信息都是以字符串的形式呈现的，如个人信息、新闻内容、表格信息等。本章介绍字符串的定义、操作、比较等。

4.1 字符串的定义

4.4.1 字符和字符串

计算机的底层语言是用0/1表示的。计算机在理解字符时，是通过将其映射到对应的数字编码实现的。定义字符用关键字Character表示。字符类型只能保存一个字符，要表示多个字符，就要字符串类型。字符串是一组有序字符的集合，用String关键字表示。图4.1定义了3个字符常量，分别为字母、数字和符号，另外还定义了这3个字符组成的字符串。定义字符类型的常量或变量时，必须显式定义。编译器无法通过类型推断区别该常量或变量是字符类型，还是字符串类型，因为这两种类型的赋值都是用双引号""实现的。

```
let characterAlphabet: Character = "a"                    "a"
let characterNumber: Character = "9"                      "9"
let characterOperator: Character = "*"                    "*"

let str = "a*9"                                           "a*9"
```

图4.1 字符和字符串的声明

4.1.2　多行文本的赋值

需要输入多行文本时，Swift 提供了一个快捷的方法。如图 4.2 所示，通过三重双引号，可以将多行文本按照原有的换行格式输入并赋值给一个字符串常量 multilinesString。

```
let multilinesString = """
    Do the most simple people,
    the most happy way:
    We often heart will feel tired, just want to too much;
    We always say life trival, is actually I don't understand taste;
    Our business is busy, often is not satisfied;
    We are always competitive, is actually his vanity is too strong.
    Don't compare, heart is cool, life is so simple.
"""
print(multilinesString)
```

图 4.2　多行文本的赋值

4.1.3　字符串初始化

构建长字符串时，通常的做法是先用空字符串作为初始值，然后逐步增加字符串长度。图 4.3 定义了两个字符串变量 emptyStr 和 anotherEmptyStr，其中 emptyStr 的初始化是通过直接赋值空字符串实现的，anotherEmptyStr 则是通过 String()实现，两者的效果一致。

```
var emptyStr = ""                    ""
var anotherEmptyStr = String()       ""
```

图 4.3　空字符串

初始化字符串后，可以通过字符串的方法.isEmpty 检查字符串是否为空，如图 4.4 所示。

```
if emptyStr.isEmpty {
    print("This String is empty")        "This String is empty\n"
}
```

图 4.4　空字符串的检查

4.1.4 值类型

字符串类型是值类型。对字符串常量或者变量进行赋值时，是将字符串的值进行了复制，而不是字符串的指针，也就是重新开辟了一片存储空间用来存储常量或变量的字符串，与被复制的字符串分别位于不同的存储空间。因此，用来赋值的字符串变量的值不会随着被赋值的字符串变量值的改变而改变。

如图 4.5 所示，对字符串变量 str2 赋值为 str1。str2 的值是通过复制 str1 的值得到的，在系统内存中，这两个相同的字符串分别存储在不同的存储空间里。将 str2 的值改为"hello China"后，只是修改了保存 str2 的内存单元，str1 的值不受任何影响。

```
var str1 = "hello world"                    "hello world"
var str2 = str1                             "hello world"
str2 = "helllo China"                       "helllo China"
print(str1)                                 "hello world\n"
```

图 4.5 值类型

练习题

1. 定义两个字符串变量，分别表示大学校名和学院名，并用两种方式赋初始值为空字符串。

2. 在第 1 题的基础上判断学院名是否为空。如果为空，则打印提示信息；如果不为空，则打印输出学院名。

3. 在第 1 题的基础上给大学校名赋值一个具体的校名，然后判断其是否为空。如果为空，则打印提示信息；如果不为空，则打印输出校名。

4. 在第 3 题的基础上定义一个新的字符串变量 myUniversity，并将 3 题的大学校名赋值给 myUniversity。打印 3 题中的校名和 myUniversity，然后将 myUniversity 赋值为"Tsinghua University"，再将两个校名打印出来。比较两次打印的结果，并说明原因。

4.2 字符串的操作

下面介绍几个常用的字符串方法。

4.2.1　遍历字符串

在 Swift 中遍历字符串是非常简单的事情，不需要通过字符串的 characters 属性访问。如图 4.6 所示，如果要取出字符串中的字符，通过 for-in 循环直接遍历字符串本身，就可以获得字串符中的每个字符，

```
var welcomeString = "Hello, world!"
for charInString in welcomeString {
    print(charInString)
}
```

```
"Hello, world!"

(13 times)
```

图 4.6　遍历字符串

4.2.2　字符数的统计

如图 4.7 所示，要统计字符串中的字符数，直接调用字符串的 count()方法即可。

```
let countString = welcomeString.count
```

```
13
```

图 4.7　字符数的统计

4.2.3　字符串的连接运算

字符串和字符都可以通过加法运算符(+)进行连接，从而得到一个新的字符串，如图 4.8 所示。

```
var str1 = "Hello"
var str2 = "world"
var character1 = ","
var character2 = "!"

let newString = str1 + character1 + str2 + character2
```

```
"Hello"
"world"
","
"!"

"Hello,world!"
```

图 4.8　字符串连接

4.2.4　插入字符串

如果要向一个字符串中插入一个常量或变量，可以采用格式：\(constantName or variableName)。如图 4.9 所示，整型常量 insertedNum 的值被插入到字符串中。

```
let insertedNum = 888

let insertedString = "The number \(insertedNum) is
    inserted into this string!"
```

```
888

"The number 888 is inserted into this string!"
```

图 4.9　插入字符串

4.2.5　字符串的大小写

另外，还可以通过字符串的方法 uppercased()和 lowercased()分别取得字符串的大写和小写版本，如图 4.10 所示。

```
let theString = "It's my world!"
print("the uppercaseString is \(String(describing:
    theString.uppercased))")
print("the lowercaseString is \(String(describing:
    theString.lowercased))")
```

```
"It's my world!"
"the uppercaseString is (Function)\n"

"the lowercaseString is (Function)\n"
```

图 4.10　字符串的大小写

4.2.6　字符串索引

字符串中的字符可以通过字符串索引获取。在其他编程语言中，字符串的索引一般都为整型，而 Swift 的字符串索引则为一种特殊的类型 String. Index。这使得字符串索引的相关操作复杂一些。

要获取字符串中某个位置的字符时，首先要获取字符串中特殊位置的索引，如起始位置或结束位置。图 4.11 定义了一个字符串常量 courseTitle，用来保存课程名，并赋值为"Swift"。然后，通过字符串的方法 startIndex()获得字符串的起始位置索引，并赋值给常量 firstIndex，它的类型为 String. Index。最后，通过 firstIndex 获取字符串中起始位置的字符"S"。

```
let courseTitle = "Swift"

let firstIndex = courseTitle.startIndex
let firstCharacter = courseTitle[firstIndex]
```

```
"Swift"

String.Index
"S"
```

图 4.11　字符串起始位置索引

这里要特别注意，字符串中最后一个字符的位置在结束位置前，如果根据结束位

置索引获取字符，会导致严重错误。图 4.12 定义了一个常量 lastIndex，赋值为字符串的结束位置索引。然后，根据该索引获取对应位置的字符，结果导致系统出错。

```
let lastIndex = courseTitle.endIndex

let lastCharacter = courseTitle[lastIndex]   error: Execution was int...   error
```

图 4.12　字符串结束位置索引

取出字符串结束位置字符的正确方式是根据字符串结束位置前的索引，获取字符串的最后一个字符，如图 4.13 所示。

```
//let lastIndex = courseTitle.endIndex
let lastIndex = courseTitle.index(before: courseTitle.endIndex)
let lastCharacter = courseTitle[lastIndex]
```

String.Index
"t"

图 4.13　获取结束位置字符的正确方式

在起始位置或结束位置的索引基础上，可以通过指定偏移量获取其他位置的字符。如图 4.14 所示，定义常量 middleIndex 存储字符串的中间位置的索引。这里以字符串起始位置为基准，向后偏移量为 2，即字符串的第 3 个字符的位置，也就是中间位置，然后再根据 middleIndex 索引获取中间位置的字符"i"。

```
let middleIndex = courseTitle.index(courseTitle.startIndex,
    offsetBy: 2)
let middleCharacter = courseTitle[middleIndex]
```

String.Index
"i"

图 4.14　获取字符串中任意位置的字符

4.2.7　字符串的子串

字符串的子串就是从字符串中截取一部分，形成一个新的字符串。例如，姓名是一个人基本信息中常见的字符串，它的子串"姓"和"名"就是有用的两个子串。如何获取字符串中的特定子串呢？图 4.15 定义了表示学生姓名的字符串常量 studentName，并赋了初始值。要获取学生的姓和名，可以先得到字符串中空格的字符串索引。表示名的子串就是字符串中从起始位置到空格位置的子串。注意，起始位置属于子串的一部分，而空格则不属于子串，因此这里使用符号“<”表示小于该索引值。

这里使用了区间标识[startPosition, endPosition]，表示从 startPosition 到

```
let studentName = "Liang Zhang"                                    "Liang Zhang"
let spaceIndex = studentName.firstIndex(of: " ")!                  String.Index
let firstName = studentName[studentName.startIndex..<spaceIndex]   "Liang"
```

图 4.15　获取姓名中的子串

endPosition 之间的所有元素，包括 startPosition 和 endPosition。其中，startPosition 为区间的起始位置索引，endPosition 为区间的结束位置索引。

如果 startPosition 为字符串的起始位置，或者 endPosition 为字符串的结束位置，则可采用半开区间写法，默认表示从起始位置到特定位置，或者从特定位置到结束位置。如图 4.16 所示，学生的名用常量 firstName2 表示，姓用常量 surname 表示。这两个子串分别是从起始位置到空格和从空格到结束的位置，均采用半开区间表示子串的范围。

```
let firstName2 = studentName[..<spaceIndex]                        "Liang"
let surname = studentName[studentName.index(after: spaceIndex)...] "Zhang"
```

图 4.16　半开区间表示子串的范围

检查子串的类型声明发现，子串并不是字符串类型，而是一种特殊的类型 String.SubSequence。当要将子串作为字符串使用时，需要将其强制转换为字符串，如图 4.17 所示。子串之所以不同于字符串，是因为子串与其对应的字符串指向了同一块物理空间，这个空间存储整个字符串的内容，当然也包括子串的内容。当子串被强制转换为字符串后，就重新开辟一块新的物理空间，用来存储子串的内容。

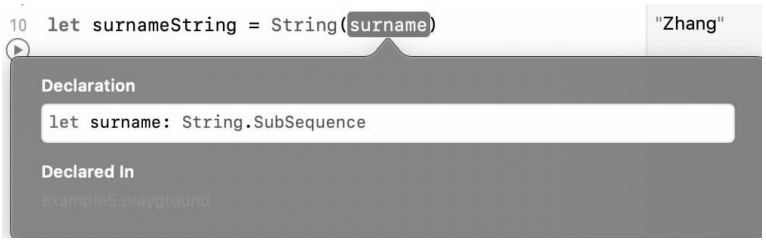

图 4.17　字符串子串的类型

4.2.8　字符串的比较

可以从 3 个维度比较字符串，即字符串相等、字符串前缀相等和字符串后缀相等。

如果两个字符串中的字符以相同顺序出现，并且完全相同，则两个字符串相等，如图 4.18 所示。

```
var compareStr1 = "It is compare string."
var compareStr2 = "It is compare string."
if compareStr1 == compareStr2 {
    print("they are equal!")
}
```

```
"It is compare string."
"It is compare string."

"they are equal!\n"
```

图 4.18　字符串相等

要判断字符串是否含有特定的字符串前缀，可以使用方法 hasPrefix()。通过该方法，可以便捷地判断一个字符串是否包含特定的字符串前缀，具体格式为

```
theString.hasPrefix(thePrefix)
```

图 4.19 定义了字符串常量 bookInfo1。通过 hasPrefix()方法判断该字符串是否包含特定的前缀"History book"。

```
let bookInfo1 = "History book: world history"

if bookInfo1.hasPrefix("History book") {
    print("book1 is a history book.")
}
```

```
"History book: world history"

"book1 is a history book.\n"
```

图 4.19　判断字符串的特定前缀

要判断字符串中是否含有特定的字符串后缀，可以使用方法 hasSuffix()，具体格式为

```
theString.hasSuffix(theSuffix)
```

图 4.20 定义了字符串常量 bookInfo2，并检索了该字符串是否包含特定的后缀"history"。

```
let bookInfo2 = "History book: China history"

if bookInfo2.hasSuffix("history") {
    print("book2 is a history book.")
}
```

```
"History book: China history"

"book2 is a history book.\n"
```

图 4.20　判断字符串的特定后缀

练习题

1. 定义一个字符串变量 myHobbies，并将其赋值为你的兴趣爱好，至少为 3 个。遍历 myHobbies 字符串中的每个字符并打印，查看循环执行的次数。

2. 在第 1 题的基础上调用字符串的字符数统计方法，查看字符数是否和 1 题的循环执行次数一致，为什么？

3. 定义两个字符串常量 str1 和 str2，分别赋值为"My hobbies" "is"。定义一个字符串变量 info，赋值为 3 个字符串 str1、str2、myHobbies 连接的结果。打印字符串 info。

4. 在第 3 题的基础上向字符串 info 中的单词之间插入空格，并重新打印输出。

5. 打印输出字符串 info 的大写和小写版本。

6. 将字符串变量 myHobbies 中的第一个字母和最后一个字母取出并打印。myHobbies 中有 3 个兴趣，将第二个兴趣的首字母，通过字符串索引打印出来，要求分别从字符串的起始位置和结束位置向第二个兴趣的首字母检索。

7. 运用字符串子串中的相关知识将 myHobbies 中的 3 个兴趣分别保存到 3 个新的字符串中，并打印输出，注意每个兴趣中都不要含有多余的空格。

8. 比较第 7 题中的第一、二个兴趣的字符串是否相等，并打印输出结果。检索第一个兴趣中是否含有特定字符串前缀，检索第二个兴趣中是否含有特定字符串后缀，并打印输出结果。

本章知识点

1. 字符和字符串的定义、多行文本的赋值方法、字符串的初始化及空字符串检测、字符串的值类型特点。

2. 字符串的常用操作包括遍历字符串、字符数的统计、字符串的连接运算、插入字符串、字符串的大小写、字符串索引、字符串的子串、字符串的比较（字符串相等、字符串前缀相等、字符串后缀相等）。

参考代码

【4.1 节　字符串的定义】

第 1 题的参考代码-字符串的定义如图 4.21 所示。

```
var schoolName = ""
var universityName = String()
```

图 4.21　第 1 题的参考代码-字符串的定义

第 2 题的参考代码-字符串的定义如图 4.22 所示。

```
if schoolName.isEmpty {
    print("School Name is empty!")
} else {
    print("My School is \(schoolName)")
}
```

图 4.22　第 2 题的参考代码-字符串的定义

第 3 题的参考代码-字符串的定义如图 4.23 所示。

```
universityName = "Beijing University of Aeronautic and
    Astronautic"
if universityName.isEmpty {
    print("University Name is empty!")
} else {
    print("My University is \(universityName)")
}
```

图 4.23　第 3 题的参考代码-字符串的定义

第 4 题的参考代码-字符串的定义如图 4.24 所示。

```
var myUniversity = ""
myUniversity = universityName
print("My university is \(myUniversity), University Name
    is \(universityName)")

myUniversity = "Tsinghua University"
print("My university is \(myUniversity), University Name
    is \(universityName)")
```

图 4.24　第 4 题的参考代码-字符串的定义

【4.2 节　字符串的操作】

第 1 题的参考代码-字符串的操作如图 4.25 所示。

```
var myHobbies = "Music Basketball Philosophy"

for char in myHobbies {
    print(char)
}
```

图 4.25　第 1 题的参考代码-字符串的操作

第 2 题的参考代码-字符串的操作如图 4.26 所示。

```
print("The number of characters in myHobbies is \(myHobbies.count)")
```

图 4.26　第 2 题的参考代码-字符串的操作

第 3 题的参考代码-字符串的操作如图 4.27 所示。

```
let str1 = "My hobbies", str2 = "is"
var info = str1 + str2 + myHobbies
print(info)
```

图 4.27　第 3 题的参考代码-字符串的操作

第 4 题的参考代码-字符串的操作如图 4.28 所示。

```
let space = " "
info = str1 + space + str2 + space + myHobbies
print(info)
```

图 4.28　第 4 题的参考代码-字符串的操作

第 5 题的参考代码-字符串的操作如图 4.29 所示。

```
print(info.uppercased())
print(info.lowercased())
```

图 4.29　第 5 题的参考代码-字符串的操作

第 6 题的参考代码-字符串的操作如图 4.30 所示。

```
let firstIndex = myHobbies.startIndex
let firstChar = myHobbies[firstIndex]

let lastIndex = myHobbies.index(before: myHobbies.endIndex)
let lastChar = myHobbies[lastIndex]

var middleIndex = myHobbies.index(firstIndex, offsetBy: 6)
var middleChar = myHobbies[middleIndex]
middleIndex = myHobbies.index(lastIndex, offsetBy: -20)
middleChar = myHobbies[middleIndex]
```

图 4.30　第 6 题的参考代码-字符串的操作

第 7 题的参考代码-字符串的操作如图 4.31 所示。

```
var spaceIndex = myHobbies.firstIndex(of: " ")!
let hobby1st = myHobbies[firstIndex..<spaceIndex]

firstIndex = myHobbies.index(after: spaceIndex)
let restOfMyHobbies = myHobbies[firstIndex...]
spaceIndex = restOfMyHobbies.firstIndex(of: " ")!

let hobby2nd = restOfMyHobbies[restOfMyHobbies.startIndex..<spaceIndex]
firstIndex = restOfMyHobbies.index(after: spaceIndex)
let hobby3rd = restOfMyHobbies[firstIndex...]

print("The first hobby is \(hobby1st)")
print("The second hobby is \(hobby2nd)")
print("The third hobby is \(hobby3rd)")
```

图 4.31　第 7 题的参考代码-字符串的操作

第 8 题的参考代码-字符串的操作如图 4.32 所示。

```
//compare hobby
if hobby1st == hobby2nd {
    print("The two hobbies are equal.")
}

if hobby1st.hasPrefix("musi") {
    print("The first hobby: \(hobby1st) has prefix: musi")
} else {
    print("The first hobby: \(hobby1st) hasn't prefix: musi")
}

if hobby2nd.hasSuffix("sophy") {
    print("The 1ast hobby: \(hobby2nd) has suffix: sophy")
} else {
    print("The 1ast hobby: \(hobby2nd) hasn't suffix: sophy")
}
```

图 4.32　第 8 题的参考代码-字符串的操作