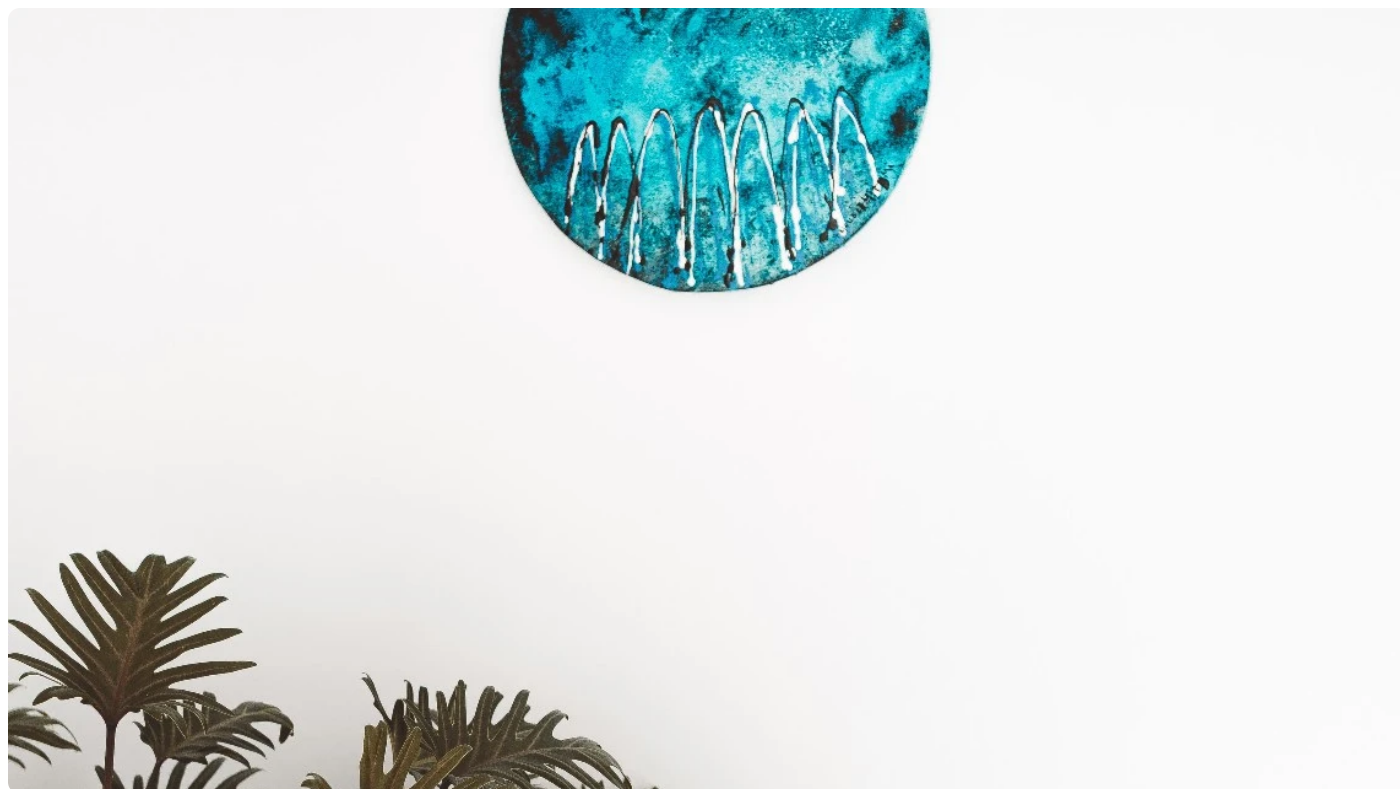


# 春节刷题计划（三）| 一题双解，搞定求解方程

2022-02-02 朱涛

《朱涛 · Kotlin编程第一课》

课程介绍 >



讲述：朱涛

时长 10:51 大小 9.95M



你好，我是朱涛。初二过年好！

在上节课里，我给你留了一个作业，那就是：用 Kotlin 来完成 [LeetCode 的 640 号题《求解方程》](#)。那么这节课，我就来讲讲我的解题思路，我们互相学习。

这道题也非常容易理解，程序的输入是一个“一元一次方程”，我们需要根据输入的方程，计算出正确的结果。根据输入方程的不同，结果可能有三种情况：

- **方程仅有一个解**，这时，我们只需要按照格式返回结果即可，比如输入“ $2x=4$ ”，那么输出就应该是“ $x=2$ ”。
- **方程有无数个解**，比如输入“ $x=x$ ”，那么输出就应该是“Infinite solutions”。
- **方程无解**，比如输入“ $x=x+5$ ”，那么输出结果就应该是“No solution”。

另外，对于程序的**输入格式**，其实我们还有几个问题需要弄清楚。只有弄清楚了这些问题，我们才能开始写代码：

- 方程当中的未知数只会用  $x$  表示，不会是  $y$ ，也不会是大写的“ $X$ ”。
- 方程当中不会出现空格，比如“ $2x=4$ ”，不会出现“ $2x = 4$ ”的情况。
- 方程当中只会有加减法，不会出现乘除法。
- 方程当中的数字，一定是整数，不会出现分数、小数。
- 输入的方程一定是一个正确的方程，不会出现“ $x=...$ ”之类的脏数据。

好，问题的细节都弄清楚了，下面我们来分析一下解题的思路。

对于这种简单的一元一次方程的解法，其实我们在小学就学过了，概括起来，就是分为三个步骤。

- 第一步，**移项**。将含有  $x$  的式子全部移到等式的左边，将数字全部都移到等式的右边。另外，移项的时候符号要变。比如“ $3x-4=x+2$ ”这个方程，移项以后，就会变成这样：“ $3x-x=2+4$ ”。
- 第二步，**合并同类项**。这里其实就是将等式的左边与右边合并起来，对于“ $3x-x=2+4$ ”这个式子，合并完以后，就会变成“ $2x=6$ ”。
- 第三步，**系数化为一**。这时候，我们就需要拿右边的数字，除以左边的系数。比如上面的式子“ $2x=6$ ”，系数化为一之后，就会变成“ $x=3$ ”，这就是我们想要的方程解。当然，这只是方程只有一个解的情况，其实在系数化为一之前，还存在其他的情况，比如“ $x=x+5$ ”最终会变成“ $0=5$ ”，这时候左边是零，右边不是零，这时候就代表方程无解；对于“ $2x=2x$ ”这样的方程，它最终会变成“ $0=0$ ”，这种两边都等于零的情况，就代表了方程有无数个解。

好，如何求解方程的思路我们已经知道了，那么代码该如何写呢？这里，我们仍然有两种解法，这两种解法的思路是一致的，只是其中一种是偏命令式的，另一种是偏函数式的。

这里，我照样是制作了一张动图，给你展示下程序运行的整体思路：

# 求解方程

$$x+5-3+x=6+x-2$$

## 解法一：命令式

首先，我们按照前面分析的思路，把待实现的程序分为以下几个步骤：

```
1 fun solveEquation(equation: String): String {  
2     // ① 分割等号  
3     // ② 遍历左边的等式，移项，合并同类项  
4     // ③ 遍历右边的等式，移项，合并同类项  
5     // ④ 系数化为一，返回结果  
6 }
```

[复制代码](#)

根据注释，我们很容易就能完成其中①、④两个步骤的代码：

```
1 fun solveEquation(equation: String): String {  
2     // ① 分割等号  
3     val list = equation.split("=")  
4  
5     // ② 遍历左边的等式，移项，合并同类项  
6     // ③ 遍历右边的等式，移项，合并同类项  
7  
8     // ④ 系数化为一，返回结果  
9     return when {  
10         leftSum == 0 && rightSum == 0 -> "Infinite solutions"  
11         leftSum == 0 && rightSum != 0 -> "No solution"
```

[复制代码](#)

```
12         else -> "x=${rightSum / leftSum}"
13     }
14 }
```

现在，关键还是在于②、③两个步骤的代码。这里，`list[0]`其实就代表了左边的式子，`list[1]`就代表了右边的式子。

按照之前的思路分析，我们其实用两个 `for` 循环，分别遍历它们，然后顺便完成移项与合并同类项就行了。具体的代码如下：

 复制代码

```
1 var leftSum = 0
2 var rightSum = 0
3
4 val leftList = splitByOperator(list[0])
5 val rightList = splitByOperator(list[1])
6
7 // ② 遍历左边的等式，移项，合并同类项
8 leftList.forEach {
9     if (it.contains("x")) {
10         leftSum += xToInt(it)
11     } else {
12         rightSum -= it.toInt()
13     }
14 }
15
16 // ③ 遍历右边的等式，移项，合并同类项
17 rightList.forEach{
18     if (it.contains("x")) {
19         leftSum -= xToInt(it)
20     } else {
21         rightSum += it.toInt()
22     }
23 }
```

这段代码的逻辑其实也比较清晰了，`leftList`、`rightList` 是根据“+”、“-”分割出来的元素。在完成分割以后，我们再对它们进行了遍历，从而完成了移项与合并同类项。

并且，这里我们还用到了另外两个方法，分别是 `splitByOperator()`、`xToInt()`，它们具体的代码如下：

 复制代码

```

1 private fun splitByOperator(list: String): List<String> {
2     val result = mutableListOf<String>()
3     var temp = ""
4     list.forEach {
5         if (it == '+' || it == '-') {
6             if (temp.isNotEmpty()) {
7                 result.add(temp)
8             }
9             temp = it.toString()
10        } else {
11            temp += it
12        }
13    }
14
15    result.add(temp)
16    return result
17 }
18
19 private fun xToInt(x: String) =
20     when (x) {
21         "x",
22         "+x" -> 1
23         "-x" -> -1
24         else -> x.replace("x", "").toInt()
25     }

```

从以上代码中，我们可以看到 `splitByOperator()` 就是使用“+”、“-”作为分隔符，将字符串类型的式子，分割成一个个的元素。而 `xToInt()` 的作用则是为了提取 `x` 的系数，比如“2x”，提取系数以后，就是“2”；而“-2x”的系数就是“-2”。

最后，我们再来看看整体的代码：

 复制代码

```

1 fun solveEquation(equation: String): String {
2     // ① 分割等号
3     val list = equation.split("=")
4
5     var leftSum = 0
6     var rightSum = 0
7
8     val leftList = splitByOperator(list[0])
9     val rightList = splitByOperator(list[1])
10
11    // ② 遍历左边的等式，移项，合并同类项
12    leftList.forEach {
13        if (it.contains("x")) {
14            leftSum += xToInt(it)

```

```

15     } else {
16         rightSum -= it.toInt()
17     }
18 }
19
20 // ③ 遍历右边的等式，移项，合并同类项
21 rightList.forEach{
22     if (it.contains("x")) {
23         leftSum -= xToInt(it)
24     } else {
25         rightSum += it.toInt()
26     }
27 }
28
29 // ④ 系数化为一，返回结果
30 return when {
31     leftSum == 0 && rightSum == 0 -> "Infinite solutions"
32     leftSum == 0 && rightSum != 0 -> "No solution"
33     else -> "x=${rightSum / leftSum}"
34 }
35 }
36
37 // 根据“+”、“-”分割式子
38 private fun splitByOperator(list: String): List<String> {
39     val result = mutableListOf<String>()
40     var temp = ""
41     list.forEach {
42         if (it == '+' || it == '-') {
43             if (temp.isNotEmpty()) {
44                 result.add(temp)
45             }
46             temp = it.toString()
47         } else {
48             temp += it
49         }
50     }
51
52     result.add(temp)
53     return result
54 }
55
56 // 提取x的系数：“-2x” ->“-2”
57 private fun xToInt(x: String) =
58     when (x) {
59         "x",
60         "+x" -> 1
61         "-x" -> -1
62         else -> x.replace("x", "").toInt()
63     }

```

至此，偏命令式的代码就完成了，接下来我们看看偏函数式的代码该怎么写。

## 解法二：函数式

这里你要注意了，函数式的思路呢，和命令式的思路其实是一样的。解方程的步骤是不会变的，仍然是移项、合并同类项、系数化为一。只不过，对比前面的实现方式，我们这里会更多地借助 **Kotlin** 的标准库函数。

首先，我们来看看第一部分的代码怎么写：

 复制代码

```
1 fun solveEquation(equation: String): String {
2     val list = equation
3         .replace("-", "+-") // 预处理逻辑
4         .split("=")
5
6     // 用“+”分割字符串
7     val leftList = list[0].split("+")
8     val rightList = list[1].split("+")
9
10    // 省略
11 }
```

这里，为了可以直接使用 **Kotlin** 的库函数 `split` 来实现算式的分割，我使用了一种**数据预处理**的办法。你可以看到，在上面代码的注释处，`replace("-", "+-")` 的作用是将算式当中的所有“-”替换成“+-”，这就是预处理。经过这个预处理后，我们就可以直接使用 `split("+")` 来分割算式了。

为了体现这个细节，我这里也做了一个动图，你可以看看：

求解方程

$$x+5-3+x=6+x-2$$

预处理

这样一来，我们得到的 `leftList`、`rightList` 其实就是干净的、独立的数字和  $x$  式子了。以“ $x+5-3+x=6+x-2$ ”为例，`leftList=["x","5","-3","x"]`，而`rightList=["6","x","-2"]`。

既然它们两者都是普通的集合，那么我们接下来，就完全可以借助 **Kotlin** 强大的库函数来做剩下的事情了。我们只需要将所有  $x$  的式子挪到左边，所有数字挪到右边，然后合并，最后系数化为一即可。大致代码如下：

 复制代码

```
1 leftList
2     .filter { it.hasX() }
3     .map { xToInt(it) } // ①
4     .toMutableList()
5     .apply {
6         rightList
7             .filter { it.hasX() }
8             .map { xToInt(it).times(-1) } // ②
9             .let { addAll(it) }
10    }.sum() // ③
11    .let { leftSum = it }
12
13 rightList
14     .filter { it.isNumber() }
15     .map { it.toInt() } // ④
16     .toMutableList()
17     .apply {
18         leftList
19             .filter { it.isNumber() }
20             .map { it.toInt().times(-1) } // ⑤
21             .let { addAll(it) }
22    }.sum() // ⑥
23    .let { rightSum = it }
24
25 // 返回结果
26 return when {
27     leftSum == 0 && rightSum == 0 -> "Infinite solutions"
28     leftSum == 0 && rightSum != 0 -> "No solution"
29     else -> "x=${rightSum / leftSum}" // ⑦
30 }
```

上面这段代码中，一共有 6 个注释，我们一个个看：

- 注释①，我们提取出了左边式子里所有  $x$  的系数，这里不需要移项，因为它本来就在左边。



- 注释②，我们提取了右边式子里所有  $x$  的系数，由于这里涉及到移项，因此需要变号，这里我们通过乘以一个“-1”来实现的。
- 注释③，我们将所有  $x$  的系数合并到了一起，得到了左边  $x$  的系数之和。
- 注释④，我们收集了右边式子里所有的数字，这里也不需要移项，因为它本来就在右边。
- 注释⑤，我们收集了左边式子里所有的数字，这里要移项，所以要变号。
- 注释⑥，我们将所有数字求和了。
- 注释⑦，如果方程有解的话，我们通过“rightSum / leftSum”就可以计算出来了。

另外，以上代码其实还涉及到三个辅助的函数，需要我们自己实现，它们的逻辑都很简单：

 复制代码

```
1 private fun String.isNumber(): Boolean =
2     this != "" && !this.contains("x")
3
4 private fun String.hasX(): Boolean =
5     this != "" && this.contains("x")
6
7 // 提取x的系数：“-2x” ->“-2”
8 private fun xToInt(x: String) =
9     when (x) {
10         "x" -> 1
11         "-x" -> -1
12         else -> x.replace("x", "").toInt()
13     }
```

`xToInt()` 这个函数和之前的逻辑是相似的，`isNumber()` 和 `hasX()` 这两个扩展函数，它们是用来判断式子是纯数字、还是含有  $x$  的，这是因为我们要把  $x$  放到等式左边，而数字要放到等式右边。

最后，我们再来看看整体的代码：

 复制代码

```
1 fun solveEquation(equation: String): String {
2     val leftSum: Int
3     val rightSum: Int
4
5     val list = equation
6         .replace("-", "+-") // 预处理数据
7         .split("=")
```

```

8      val leftList = list[0].split("+")
9      val rightList = list[1].split("+")
10
11
12      // 求出所有x的系数之和
13      leftList
14          .filter { it.hasX() }
15          .map { xToInt(it) }
16          .toMutableList()
17          .apply {
18              rightList
19                  .filter { it.hasX() }
20                  .map { xToInt(it).times(-1) }
21                  .let { addAll(it) }
22          }.sum()
23          .let { leftSum = it }
24
25      // 求出所有数字之和
26      rightList
27          .filter { it.isNumber() }
28          .map { it.toInt() }
29          .toMutableList()
30          .apply {
31              leftList
32                  .filter { it.isNumber() }
33                  .map { it.toInt().times(-1) }
34                  .let { addAll(it) }
35          }.sum()
36          .let { rightSum = it }
37
38      // 返回结果
39      return when {
40          leftSum == 0 && rightSum == 0 -> "Infinite solutions"
41          leftSum == 0 && rightSum != 0 -> "No solution"
42          else -> "x=${rightSum / leftSum}"
43      }
44  }
45
46  private fun String.isNumber(): Boolean =
47      this != "" && !this.contains("x")
48
49  private fun String.hasX(): Boolean =
50      this != "" && this.contains("x")
51
52      // 提取x的系数: "-2x" -> "-2"
53  private fun xToInt(x: String) =
54      when (x) {
55          "x" -> 1
56          "-x" -> -1
57          else -> x.replace("x", "").toInt()
58      }

```

## 小结

这节课，我们用两种方式实现了 [LeetCode 的 640 号题《求解方程》](#)。这两种解法的核心思路其实是一致的，不过前者是偏命令式的，后者是偏函数式的。而你要清楚，即使它们是用的一种思路，也仍然是各有优劣的。


- 解法一，命令式的代码，它的时间复杂度和空间复杂度要稍微好一些，但总体差距不大，所以不一定能体现出运行时的差异。这种方式的劣势在于，逻辑相对复杂，可读性稍差，且编码过程中容易出错。
- 解法二，偏函数式的代码，它的优势在于，代码逻辑相对清晰，并且，由于运用了大量 Kotlin 库函数，没那么容易出错。


## 小作业

好，最后，我还是给你留一个小作业，请你用 Kotlin 来完成 [LeetCode 的 592 号题《分数加减运算》](#)，下节课我也会给出我的答案。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [春节刷题计划（二）| 一题三解，搞定版本号判断](#)

下一篇 [春节刷题计划（四）| 一题三解，搞定分式加减法](#)

## 精选留言 (4)

 写留言



白乾涛

2022-03-02

```

fun solveEquation(equation: String): String { // x+5-3+2x=6+x-2
    var xCount = 0 // 移到左边的 x 系数之和
    var addValue = 0 // 移到右边的数字之和
    val equalIndex = equation.indexOf('=') // 等号的位置
    var i = 0
    while (i < equation.length) {
        val fromIndex = i
        if (i == 0) {
            i++
        }
        for (j in i until equation.length) {
            val c: Char = equation[j]
            if (c == '+' || c == '-' || c == '=') {
                break
            }
            i++
        }
        var subString = equation.substring(if (fromIndex == 0) fromIndex else fromIndex - 1, i)
        subString = if (subString.startsWith("=")) subString.substring(1) else subString
        println("值为: $subString")
        if (subString.endsWith("x")) {
            subString = subString.substring(0, subString.length - 1)
            val tempCount = if (subString.isEmpty()) 1 // x
            else {
                if (subString.length == 1 && (subString.startsWith("+") || subString.startsWith("-")))
                { // +x 或 -x
                    if (subString[0] == '+') 1 else -1
                } else subString.toInt() // +5x 或 5x 或 53x 或 -2x
            }
            xCount = if (i > equalIndex) xCount - tempCount else xCount + tempCount // 左正右
            负
        } else if (subString.isNotEmpty()) { // 过滤掉 = 产生的一个空字符串
            val tempValue = subString.toInt()
            addValue = if (i > equalIndex) addValue + tempValue else addValue - tempValue // 左
            负右正
        }
        i++
    }
    println("结果: ${xCount}x = $addValue")
    return if (xCount == 0) if (addValue == 0) "Infinite solutions" else "No solution"
}

```

```
else "x=" + addValue / xCount
```

```
}
```

作者回复: 很符合直觉的思路, 不过略显繁琐。



**Geek\_Adr**

2022-02-20

符号处理复杂了, 其它与 @郑峰 略同

```
// 分数的数据结构 symbol为-1 1
```

```
// 注意分子/分母为正整数
```

```
data class Fraction(val numerator: Int, val denominator: Int, val symbol: Int)
```

```
fun fractionAddition(expression: String): String {
```

```
    return expression.replace("-", "+-") // "-"前增加+, 方便split处理
```

```
    .split("+") // 按"+"分隔
```

```
    .filter { it.isNotBlank() } // 去掉可能为空的部分
```

```
    .map { // 处理成分数实例
```

```
        var symbol = if (it.startsWith("-")) -1 else 1
```

```
        val ss = it.replace("-", "").split("/")
```

```
        Fraction(ss[0].toInt(), ss[1].toInt(), symbol)
```

```
    }.run {
```

```
        // 算出分母的最小公倍数, 作为分母
```

```
        val denominator = map { it.denominator }.reduce { a, b -> lcm(a, b) }
```

```
        // 按最小公倍数计算分子结果
```

```
        var numerator = map { it.symbol * it.numerator * denominator / it.denominator }.red
```

```
uce { a, b -> a + b }
```

```
        val symbol = if (numerator < 0) -1 else 1
```

```
        numerator *= symbol // 转正
```

```
        val gcd = gcd(numerator, denominator) // 结果可能可约分
```

```
        Fraction(numerator / gcd, denominator / gcd, symbol)
```

```
    }.run { "${if (symbol < 0) "-" else ""}${numerator}/${denominator}" }
```

```
}
```

```
// 最小公倍数
```

```
private fun lcm(m: Int, n: Int): Int {
```

```
    return m * n / gcd(m, n)
```

```
}
```

```
// 最大公约数
```

```
private fun gcd(m: Int, n: Int): Int {  
    return if (m % n == 0) n else gcd(n, m % n)  
}
```

作者回复: 代码写的挺好的, 大家可以参考看看。



**qinsi**

2022-02-02

尝试用正则分割

```
```kotlin  
// 根据“+”、“-”分割式子  
private fun splitByOperator(list: String) =  
    list.split(Regex("(?=[+-])")).filter { it.isNotEmpty() }  
```
```

或者

```
```kotlin  
// 根据“+”、“-”分割式子  
private fun splitByOperator(list: String): Sequence<String> =  
    Regex("""[+-]?(\d*x|\d+)""").findAll(list).map { it.value }  
```
```

作者回复: 这种思路也很巧妙, 值得大家学习。



**郑峰**

2022-02-02

```
```Kotlin  
fun fractionAddition(expression: String): String {  
    // Split the expression to several pairs of numerator and denominator  
    val numbers = expression  
        .replace("-", "+-")  
        .split("+")  
        .filter { it.isNotEmpty() }  
        .map { it.split("/").take(2).map(String::toInt) }
```

```
// Calculate the lcm of all denominators
val rawDenominator = numbers.map { it[1] }.fold(1) { x, y -> lcm(x, y) }
// Calculate the sum of all numerators
val rawNumerator = numbers.sumOf { it[0] * rawDenominator / it[1] }

// Reformat numerator and denominator through their gcd
val gcd = abs(gcd(rawNumerator, rawDenominator))
val denominator = rawDenominator / gcd
val numerator = rawNumerator / gcd
return "$numerator/$denominator"
}

fun gcd(x: Int, y: Int): Int = if (y == 0) x else gcd(y, x % y)
fun lcm(x: Int, y: Int): Int = x * y / gcd(x, y)
...
```

作者回复: 这代码很不错，比我提供的解法更加的简洁。这种不拘泥与特定编程范式，并且融合双方优势的写法，看起来真的很舒服。

共 2 条评论 >

