第2章

数据类型

如果把写程序比喻成盖房子，那么各种变量相当于各种建筑材料，建材包括砖头、水泥、沙子等，程序变量也分为不同的数据类型，例如整型数、浮点数、数组、字符串以及更高级的各种容器等。建筑内容有诸如砌砖头、搅拌泥沙等处理操作，数据类型也拥有自己的常见操作，如转换、修改、查询等。下面就依次介绍如何使用 Kotlin 的各种数据类型。

2.1 基本数据类型

每个编程语言都离不开基本的数据类型，包括整型、浮点型、布尔型等，当然 Kotlin 也不例外。虽然基本数据类型的概念是老生常谈，但是 Kotlin 声明基本变量究竟有哪些特别之处呢？本节从基本变量类型开始，逐步探讨这些数据类型的常见用法。

2.1.1 基本类型的变量声明

Kotlin 的基本数据类型跟其他高级语言的分类一样，包括整型、长整型、浮点型、双精度、布尔型、字符型、字符串这几种常见类型，具体的类型名称说明见表 2-1。

表 2-1 Kotlin 与 Java 的基本数据类型对比

基本数据类型名称	Kotlin 的数据类型	Java 的数据类型
整型	Int	int 和 Integer
长整型	Long	long 和 Long
浮点型	Float	float 和 Float
双精度	Double	double 和 Double
布尔型	Boolean	boolean 和 Boolean

（续表）

基本数据类型名称	Kotlin 的数据类型	Java 的数据类型
字符型	Char	char
字符串	String	String

看起来很熟悉是不是，Kotlin 原来这么简单。可是如果你马上敲出变量声明的代码，便会发现编译有问题。比如声明一个最简单的整型变量，按 Java 的写法是下面这样：

```
int i=0;
```

倘若按照 Java 的规则来书写 Kotlin 代码，就是下面这行代码：

```
Int i=0;
```

然而 Android Studio 立即提示编译不通过，刚开始学 Kotlin 便掉到坑里，看来要认真对待 Kotlin，不能这么轻易让它坑蒙拐骗了。正确的 Kotlin 声明变量的代码是下面这样的：

```
var i:Int = 0
```

前面的 var 表示后面是一个变量声明语句，接着是“变量名:变量类型”的格式声明，而不是常见的“变量类型 变量名”这种格式。至于后面的分号，则看该代码行后面是否还有其他语句，如果变量声明完毕直接回车换行，那么后面无须带分号；如果没有回车换行，而是添加其他语句，那么变量声明语句要带上分号。

2.1.2　简单变量之间的转换

Kotlin 变量的另一个重要特点是类型转换，在 Java 开发中，如 int、long、float、double 类型的变量可以直接在变量名前面加上诸如（int）、（long）、（float）、（double）这种表达式进行强制类型转换；对于 int（整型）和 char（字符型）这两种类型，甚至都无须转换类型，直接互相赋值即可。但在 Kotlin 中，不允许通过 Java 的前缀表达式来强制转换类型，只能调用类型转换函数输出其他类型的变量，表 2-2 是常见的几种类型转换函数的说明。

表 2-2　Kotlin 的数据类型转换函数的说明

Kotlin 的数据类型转换函数	转换函数说明
toInt	转换为整型数
toLong	转换为长整型
toFloat	转换为浮点数
toDouble	转换为双精度数
toChar	转换为字符
toString	转换为字符串

接下来通过实际代码观察一下类型转换的过程，测试用到的类型转换的 Kotlin 代码片段如下所示：

```
    val origin:Float = 65.0f
    tv_origin.text = origin.toString()
    var int:Int
    btn_int.setOnClickListener { int=origin.toInt();
tv_convert.text=int.toString() }
    var long:Long
    btn_long.setOnClickListener { long=origin.toLong();
tv_convert.text=long.toString() }
    var float:Float
    btn_float.setOnClickListener { float=origin.toDouble().toFloat();
tv_convert.text=float.toString() }
    var double:Double
    btn_double.setOnClickListener { double=origin.toDouble();
tv_convert.text=double.toString() }
    var boolean:Boolean
    btn_boolean.setOnClickListener { boolean=origin.isNaN();
tv_convert.text=boolean.toString() }
    var char:Char
    btn_char.setOnClickListener { char=origin.toChar();
tv_convert.text=char.toString() }
```

各种类型转换的操作结果如图 2-1～图 2-3 所示，其中图 2-1 展示转换为整型的界面效果，图 2-2 展示转换为双精度的界面效果，图 2-3 展示转换为字符型的界面效果。

图 2-1 浮点型转换为整型

图 2-2 浮点型转换为双精度

图 2-3 浮点型转换为字符型

注意到上述类型转换代码的第一行变量声明语句以 val 开头，而其余的变量声明语句均以 var 开头，这是为什么呢？其实 val 和 var 的区别在于，前者修饰过的变量只能在第一次声明时赋值，后续不能再赋值；而后者修饰过的变量在任何时候都允许赋值。方便记忆的话，可以把 val 看作是 Java 里的 final 关键字；至于 var，Java 里面没有对应的关键字，就当它是例行公事好了。

2.2 数　　组

2.1 节介绍了基本数据类型在 Kotlin 中的用法，不过这只针对单个变量，如果要求把一组相同类型的变量排列起来，形成一个变量数组，那又该如何声明和操作呢？本节就来谈谈 Kotlin 对数组的常见用法。

2.2.1　数组变量的声明

在 Java 中声明数组跟在 C 语言中声明是一样的，以整型数组为例，声明数组并加以初始化的语句如下所示：

```
int[] int_array = new int[] {1, 2, 3};
```

其他基本类型的数组声明与之类似，只要把 int 替换为 long、float、double、boolean、char 其中之一即可。但在 Kotlin 中，声明并初始化一个整型数组的语句是下面这样的：

```
var int_array:IntArray = intArrayOf(1, 2, 3)
```

两相对比，对于整型数组的声明，Kotlin 与 Java 之间有以下区别：

（1）Kotlin 另外提供了新的整型数组类型，即 IntArray。

（2）分配一个常量数组，Kotlin 调用的是 intArrayOf 方法，并不使用 new 关键字。

推而广之，其他基本类型的数组也各有自己的数组类型，以及对应分配常量数组的初始化方法，详细的对应关系说明见表 2-3。

表 2-3　Kotlin 基本数据类型名称及其初始化方法对应关系

Kotlin 的基本数组类型	数组类型的名称	数组类型的初始化方法
整型数组	IntArray	intArrayOf
长整型数组	LongArray	longArrayOf
浮点数组	FloatArray	floatArrayOf
双精度数组	DoubleArray	doubleArrayOf
布尔型数组	BooleanArray	booleanArrayOf
字符数组	CharArray	charArrayOf

下面是这些基本类型数组的初始化代码例子：

```
var long_array:LongArray = longArrayOf(1, 2, 3)
var float_array:FloatArray = floatArrayOf(1.0f, 2.0f, 3.0f)
var double_array:DoubleArray = doubleArrayOf(1.0, 2.0, 3.0)
var boolean_array:BooleanArray = booleanArrayOf(true, false, true)
var char_array:CharArray = charArrayOf('a', 'b', 'c')
```

不知读者有没有注意到，上面的 Kotlin 数组类型不包括字符串数组，而 Java 是允许使用字符串数组的，声明字符串数组的 Java 代码示例如下：

```
String[] string_array = new String[] {"How", "Are", "You"};
```

但在 Kotlin 这里，并不存在名为 StringArray 的数组类型，因为 String 是一种特殊的基本数据类型。要想在 Kotlin 中声明字符串数组，得使用 Array<String>类型，也就是把“String”用尖括号包起来。同时，分配字符串数组的方法也相应变成了 arrayOf，下面是声明字符串数组的 Kotlin 代码：

```
var string_array:Array<String> = arrayOf("How", "Are", "You")
```

这种字符串数组的声明方式是不是很熟悉？看起来就跟 Java 里面的 ArrayList 用法差不多，都是在尖括号中间加入数据结构的类型。同理，其他类型的数组变量也能通过“Array<数据类型>”的方式来声明，像前面介绍的整型数组，其实可以使用类型 Array<Int>，以此类推，改造之后的各类型数组变量的声明代码如下所示：

```
var int_array:Array<Int> = arrayOf(1, 2, 3)
var long_array:Array<Long> = arrayOf(1, 2, 3)
var float_array:Array<Float> = arrayOf(1.0f, 2.0f, 3.0f)
var double_array:Array<Double> = arrayOf(1.0, 2.0, 3.0)
var boolean_array:Array<Boolean> = arrayOf(true, false, true)
var char_array:Array<Char> = arrayOf('a', 'b', 'c')
```

2.2.2 数组元素的操作

现在声明数组和对数组初始化的代码都有了，接下来还需要对数组做进一步的处理，常见的处理包括获取数组长度、获取指定位置的数组元素等，这些操作在 Kotlin 与 Java 之间的区别包括：

（1）对于如何获取数组长度，Java 使用.length，而 Kotlin 使用.size。

（2）对于如何获取指定位置的数组元素，Java 通过方括号加下标来获取，比如“int_array[0]”指的是得到该数组的第一个元素；Kotlin 也能通过方括号加下标来获取指定元素，不过 Kotlin 还拥有 get 和 set 两个方法，通过 get 方法获取元素值，通过 set 方法修改元素值，看起来就像在操作 ArrayList 队列。

下面是 Kotlin 操作字符串数组的示例代码：

```
//声明字符串数组
var string_array:Array<String> = arrayOf("How", "Are", "You")
btn_string.setOnClickListener {
    var str:String = ""
    var i:Int = 0
    while (i<string_array.size) {
        str = str + string_array[i] + ", "
        //数组元素可以通过下标访问，也可通过 get 方法访问
        //str = str + string_array.get(i) + ", "
        i++
    }
    tv_item_list.text = str
}
```

上述代码的演示效果如图 2-4 所示，可以看到字符串数组内部的各元素都被逗号分隔开了。

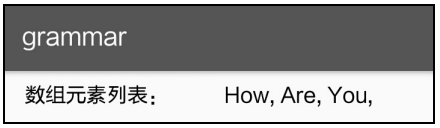

图 2-4 Kotlin 操作字符串数组的演示界面

2.3　字　符　串

2.2 节介绍了数组的声明和操作，其中包括字符串数组的用法。注意到 Kotlin 的字符串类型名称跟 Java 一样都叫 String，那么字符串在 Kotlin 和 Java 中的用法有哪些差异呢？这便是本节所要阐述的内容了。

2.3.1　字符串与基本类型的转换

首先要说明的是字符串类型与基本变量类型之间的转换方式，在前面的“2.1.2 简单变量之间的转换”中，提到基本数据类型的变量可以通过 toString 方法转换为字符串类型。反过来，字符串类型又该如何转换为基本变量类型呢？表 2-4 展示使用 Kotlin 和 Java 编码将字符串转换为基本数据类型的对照方式说明。

表 2-4　字符串转换为其他数据类型的 Kotlin 与 Java 方式对比

字符串转换目标	Kotlin 的转换方式	Java 的转换方式
字符串转整型	字符串变量的 toInt 方法	Integer.parseInt（字符串变量）
字符串转长整型	字符串变量的 toLong 方法	Long.parseLong（字符串变量）
字符串转浮点数	字符串变量的 toFloat 方法	Float.parseFloat（字符串变量）
字符串转双精度数	字符串变量的 toDouble 方法	Double.parseDouble（字符串变量）
字符串转布尔型	字符串变量的 toBoolean 方法	Boolean.parseBoolean（字符串变量）
字符串转字符数组	字符串变量的 toCharArray 方法	字符串变量的 toCharArray 方法

就表 2-4 的转换情况来看，Java 的实现方式比较烦琐，既需要其他类型的类名，又需要该类型的转换方法。而在 Kotlin 这边，转换类型相对简单，并且与基本数据类型之间的转换形式保持一致，即都是采取“to***()”的形式。显而易见，Kotlin 对字符串的类型转换方式更友好，也更方便记忆。

2.3.2　字符串的常用方法

当然，转换类型只是字符串的基本用法，还有更多处理字符串的其他用法，比如查找子串、替换子串、截取指定位置的子串、按特定字符分隔子串等，在这方面 Kotlin 基本兼容 Java 的相关方法。对于查找子串的操作，二者都调用 indexOf 方法；对于截取指定位置子串的操作，二者都调用 substring 方法；对于替换子串的操作，二者都调用 replace 方法；对于按特定字符分隔子串的操作，二者都调用 split 方法。

下面是 Kotlin 使用 indexOf 和 substring 方法进行子串查找和截取字符串的代码例子：

```
//截取小数点之前的字符串，即取整操作
val origin:String = tv_origin.text.toString()
```

```
var origin_trim:String = origin
if (origin_trim.indexOf('.') > 0) {
    origin_trim = origin_trim.substring(0, origin_trim.indexOf('.'))
}
```

在这些字符串处理方法里面，唯一区别是 split 方法的返回值，在 Java 中，split 方法返回的是 String 数组，即 String[]；但在 Kotlin 中，split 方法返回的是 String 队列，即 List<String>。下面是 Kotlin 使用 split 方法的示例代码：

```
//根据点号将源串分割为字符串队列，并将分割结果显示在界面上
btn_split.setOnClickListener {
    var strList:List<String> = origin.split(".")
    var strResult:String = ""
    for (item in strList) {
        strResult = strResult + item + ", "
    }
    tv_convert.text = strResult
}
```

分割字符串的界面效果如图 2-5 所示，可以看到源字符串里面的点号都被替换为逗号，字符串末尾也多了一个逗号。

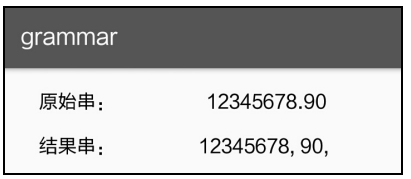

图 2-5　Kotlin 调用字符串方法的演示界面

若想获取字符串某个位置的字符，这个看似简单的需求，采取 Java 实现时却有点烦琐，因为只能调用 substring 方法去截取指定位置的字符串，具体的 Java 代码如下所示：

```
String result = origin.substring(number, number+1);
tv_convert.setText(result);
```

通过 Kotlin 实现上述需求就简单多了，因为 Kotlin 允许直接通过下标访问字符串指定位置的字符，下面是访问字符串指定位置的 Kotlin 代码例子：

```
tv_convert.text = origin[number].toString()
```

同时，Kotlin 也支持字符串变量通过 get 方法获取指定位置上的字符，代码如下：

```
tv_convert.text = origin.get(number).toString()
```

如此一来，Kotlin 的字符串定位代码不但更加精炼，而且可读性也增强了。

2.3.3　字符串模板及其拼接

Kotlin 对字符串带来的便利并不限于此，举个例子，若 Java 把几个变量拼接成字符串，则要么用加号强行拼接，要么用 String.format 函数进行格式化。可是前者的拼接加号时常会跟数值相加的加号混淆；而后者的格式化还得开发者死记硬背，如%d、%f、%s、%c、%b 等格式转换符，实在令人头痛。对于字符串格式化这个痛点，Kotlin 恰如其分地进行了优化，何必引入这些麻烦的格式转换符呢？直接在字符串中加入“$变量名”即可表示此处引用该变量的值，岂不妙哉！

心动不如行动，赶紧动起手来，看看 Kotlin 如何格式化字符串，先来看一个示例代码：

```
        btn_format.setOnClickListener { tv_convert.text = "字符串值为 $origin" }
```

这里要注意，符号$后面跟变量名，系统会自动匹配最长的变量名。比如下面这行代码，打印出来的是变量 origin_trim 的值，而不是 origin 的值：

```
        btn_format.setOnClickListener { tv_convert.text = "字符串值为
$origin_trim" }
```

另外，有可能变量会先进行运算，再把运算结果拼接到字符串中。此时，需要用大括号把运算表达式给括起来，具体代码如下所示：

```
        btn_length.setOnClickListener { tv_convert.text = "字符串长度为
${origin.length}" }
```

在上述的 Kotlin 格式化代码中，美元符号$属于特殊字符，因此不能直接打印它，必须经过转义才可以打印。转义的办法是使用“${'***'}”表达式，该表达式外层的“${''}”为转义声明，内层的“***”为需要原样输出的字符串，所以通过表达式“${'$'}”即可打印一个美元符号，示例代码如下：

```
        btn_dollar.setOnClickListener { tv_convert.text = "美元金额为
${'$'}$origin" }
```

如果只是对单个美元符号做转义，也可直接在符号$前面加个反斜杆，即变成“\$”，修改后的代码如下所示：

```
        btn_dollar.setOnClickListener { tv_convert.text = "美元金额为
\$$origin" }
```

然而一个反斜杆仅仅对一个字符进行转义，倘若要对一个字符串做转义，也就是把某个字符串的所有字符原样输出，那么只能采用形如“${'***'}”的表达式，该表达式利用单引号把待转义的字符串包起来，好处是能够保留该字符串内部的所有特殊字符。

2.4　容　　器

Kotlin 号称全面兼容 Java，于是 Java 的容器类仍可在 Kotlin 中正常使用，包括大家熟悉的队列 ArrayList、映射 HashMap 等。不过 Kotlin 作为一门全新的语言，肯定要有自己的容器类，不然哪天 Java 跟 Kotlin 划清界限，那麻烦就大了。本节就对 Kotlin 的几种容器类进行详细的说明。

2.4.1　容器的基本操作

与 Java 类似，Kotlin 也拥有三类基本的容器，分别是集合 Set、队列 List、映射 Map，然后每类容器又分作只读与可变两种类型，这是为了判断该容器能否进行增、删、改等变更操作。Kotlin 对变量的修改操作很慎重，每个变量在定义的时候就必须指定能否修改，比如添加 val 修饰表示该

变量不可修改，添加 var 修饰表示该变量允许修改。至于容器则默认为只读容器，如果需要允许修改该容器变量，就需要加上 Mutable 前缀形成新的容器，比如 MutableSet 表示可变集合，MutableList 表示可变队列，MutableMap 表示可变映射，只有可变的容器才能够对其内部元素进行增、删、改操作。

既然集合 Set、队列 List、映射 Map 三者都属于容器，那么它们必定拥有相同的容器方法，这些公共方法具体说明如下。

- isEmpty：判断该容器是否为空。
- isNotEmpty：判断该容器是否非空。
- clear：清空该容器。
- contains：判断该容器是否包含指定元素。
- iterator：获取该容器的迭代器。
- count：获取该容器包含的元素个数，也可通过 size 属性获得元素数量。

另外，Kotlin 允许在声明容器变量时就进行初始赋值，如同对数组变量进行初始化那样。而 Java 的容器类是无法同时声明并初始化的，由此可见 Kotlin 的这点特性给开发者带来很大便利。下面是一个初始化 List 队列的 Kotlin 代码例子：

```
        val satellites:List<String> = listOf("水星", "金星", "地球", "火星",
"木星", "土星")
```

当然，不同容器的初始化方法有所区别，各种容器与其初始化方法的对应关系见表 2-5。

表 2-5 Kotlin 的容器及其初始化方法的对应关系

Kotlin 的容器	容器名称	容器的初始化方法
只读集合	Set	setOf
可变集合	MutableSet	mutableSetOf
只读队列	List	listOf
可变队列	MutableList	mutableListOf
只读映射	Map	mapOf
可变映射	MutableMap	mutableMapOf

以上介绍了 Kotlin 容器的基本用法，更具体的增、删、改、查等操作则有所不同，接下来分别说明这三类 6 种容器的详细使用。

2.4.2 集合 Set/MutableSet

集合是一种最简单的容器，它具有以下特性：

（1）容器内部的元素不按顺序排列，因此无法按照下标进行访问。

（2）容器内部的元素存在唯一性，通过哈希值校验是否存在相同的元素，若存在，则将其覆盖。

因为 Set 是只读集合，初始化赋值后便不可更改，所以元素变更的方法只适用于可变集合

MutableSet，但 MutableSet 的变更操作尚有以下限制：

（1）MutableSet 的 add 方法仅仅往集合中添加元素，由于集合是无序的，因此不知道添加的具体位置。

（2）MutableSet 没有修改元素值的方法，一个元素一旦被添加，就不可被修改。

（3）MutableSet 的 remove 方法用于删除指定元素，但无法删除某个位置的元素，这是因为集合内的元素不是按顺序排列的。

对于集合的遍历操作，Kotlin 提供了好几种方式，有熟悉的 for-in 循环、迭代器遍历，还有新面孔 forEach 遍历，这三种集合遍历的用法说明如下。

1. for-in 循环

与 Java 类似，通过 for 语句加上 in 条件即可轻轻松松依次取出集合中的所有元素。下面是运用 for-in 循环的代码例子：

```
    val goodsMutSet:Set<String> = setOf("iPhone8", "Mate10", "小米 6", "OPPO R11",
"vivo X9S", "魅族 Pro6S")
    btn_set_for.setOnClickListener {
        var desc = ""
        //使用 for-in 语句循环取出集合中的每条记录
        for (item in goodsMutSet) {
            desc = "${desc}名称：${item}\n"
        }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

上述代码对应的界面效果如图 2-6 所示，可见初始化时输入的 6 部手机都通过遍历操作打印了出来。

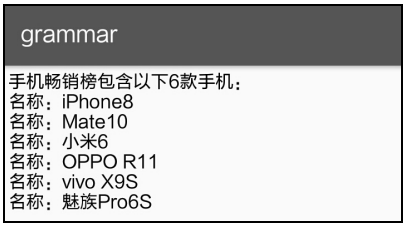

图 2-6　Kotlin 集合的遍历结果

2. 迭代器遍历

迭代器与指针的概念有点接近，它自身并非具体的元素，而是指向元素的存放地址，所以迭代器遍历其实是遍历所有元素的地址。迭代器通过 hasNext 方法判断是否还存在下一个节点，如果不存在下一节点，就表示已经遍历完毕，它通过 next 方法获得下一个节点的元素，同时迭代器自身改为指向该元素的地址。下面是运用迭代器遍历的代码例子：

```
    btn_set_iterator.setOnClickListener {
        var desc = ""
        val iterator = goodsMutSet.iterator()
```

```
        //如果迭代器还存在下一个节点，就继续取出下一个节点的记录
        while (iterator.hasNext()) {
            val item = iterator.next()
            desc = "${desc}名称：${item}\n"
        }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

3. forEach 遍历

无论是 for-in 循环还是迭代器遍历，其实都脱胎于 Java 已有的容器遍历操作，代码书写上不够精炼。为了将代码精简到极致，Kotlin 给容器创造了 forEach 方法，明确指定该方法就是要依次遍历容器内部的元素。forEach 方法在编码时采用匿名函数的形式，内部使用 it 代表每个元素，下面是运用 forEach 遍历的代码例子：

```
    btn_set_foreach.setOnClickListener {
        var desc = ""
        //forEach 内部使用 it 指代每条记录
        goodsMutSet.forEach { desc = "${desc}名称：${it}\n" }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

结合以上有关 Set/MutableSet 的用法说明，可以发现集合在实战中存在诸多不足，主要包括以下几点：

（1）集合不允许修改内部元素的值。
（2）集合无法删除指定位置的元素。
（3）不能通过下标获取指定位置的元素。

鉴于集合的以上缺点难以克服，故而实际开发基本用不到集合，大多数场合用的是它的两个兄弟——队列和映射。

2.4.3 队列 List/MutableList

队列是一种元素之间按照顺序排列的容器，它与集合的最大区别在于多了次序管理。不要小看这个有序性，正因为队列建立了秩序规则，所以它比集合多提供了如下功能（注意，凡是涉及增、删、改的，都必须由 MutableList 来完成）：

（1）队列能够通过 get 方法获取指定位置的元素，也可以直接通过下标获得该位置的元素。
（2）MutableList 的 add 方法每次都是把元素添加到队列末尾，也可指定添加的位置。
（3）MutableList 的 set 方法允许替换或者修改指定位置的元素。
（4）MutableList 的 removeAt 方法允许删除指定位置的元素。
（5）队列除了拥有跟集合一样的三种遍历方式（for-in 循环、迭代器遍历、forEach 遍历）外，还多了一种按元素下标循环遍历的方式，具体的下标遍历代码例子如下：

```
    val goodsMutList:List<String> = listOf("iPhone8", "Mate10", "小米 6", "OPPO R11",
"vivo X9S", "魅族 Pro6S")
    btn_for_index.setOnClickListener {
        var desc = ""
        //indices 是队列的下标数组。如果队列大小为 10，下标数组的取值就为 0～9
        for (i in goodsMutList.indices) {
            val item = goodsMutList[i]
            desc = "${desc}名称：${item}\n"
        }
        tv_list_result.text = "手机畅销榜包含以下${goodsMutList.size}款手机：\n$desc"
    }
```

上面按下标遍历队列的代码对应的运行界面如图 2-7 所示，可见队列中保存的 6 部手机都通过遍历显示了出来。

图 2-7　Kotlin 队列的遍历结果

（6）MutableList 提供了 sort 系列方法用于给队列中的元素重新排序，其中 sortBy 方法表示按照指定条件升序排列，sortByDescending 方法表示按照指定条件降序排列。下面是一个给队列排序的代码例子（含升序和降序）：

```
    var sortAsc = true
    btn_sort_by.setOnClickListener {
        if (sortAsc) {
            //sortBy 表示升序排列，后面跟的是排序条件
            goodsMutList.sortBy { it.length }
        } else {
            //sortByDescending 表示降序排列，后面跟的是排序条件
            goodsMutList.sortByDescending { it.length }
        }
        var desc = ""
        for (item in goodsMutList) {
            desc = "${desc}名称：${item}\n"
        }
        tv_list_result.text = "手机畅销榜已按照${if (sortAsc) "升序" else "降序"}重
新排列：\n$desc"
        sortAsc = !sortAsc
    }
```

队列进行排序操作后的演示效果如图 2-8 和图 2-9 所示，其中图 2-8 展示按升序排列后的手机信息界面，图 2-9 展示按降序排列后的手机信息界面。

图 2-8　队列进行升序排列后的结果界面

图 2-9　队列进行降序排列后的结果界面

2.4.4　映射 Map/MutableMap

映射内部保存的是一组键值对（Key-Value），也就是说，每个元素都由两部分构成，第一部分是元素的键，相当于元素的名字；第二部分是元素的值，存放着元素的详细信息。元素的键与值是一一对应的关系，相同键名指向的键值是唯一的，所以映射中每个元素的键名各不相同，这个特性使得映射的变更操作与队列存在以下不同之处（注意，增、删操作必须由 MutableMap 来完成）：

（1）映射的 containsKey 方法判断是否存在指定键名的元素，containsValue 方法判断是否存在指定键值的元素。

（2）MutableMap 的 put 方法不单单是添加元素，而是智能的数据存储。每次调用 put 方法时，映射会先根据键名寻找同名元素，如果找不到就添加新元素，如果找得到就用新元素替换旧元素。

（3）MutableMap 的 remove 方法是通过键名来删除元素的。

（4）调用 mapOf 和 mutableMapOf 方法初始化映射时，有两种方式可以表达单个键值对元素，其一是采取“键名 to 键值”的形式，其二是采取 Pair 配对方式，形如“Pair(键名, 键值)”。下面是这两种初始化方式的代码例子：

```
    //to 方式初始化映射
    var goodsMap: Map<String, String> = mapOf("苹果" to "iPhone8", "华为" to "Mate10",
"小米" to "小米 6", "欧珀" to "OPPO R11", "步步高" to "vivo X9S", "魅族" to "魅族 Pro6S")
    //Pair 方式初始化映射
    var goodsMutMap: MutableMap<String, String> = mutableMapOf(Pair("苹果",
"iPhone8"), Pair("华为", "Mate10"), Pair("小米", "小米 6"), Pair("欧珀", "OPPO R11"),
Pair("步步高", "vivo X9S"), Pair("魅族", "魅族 Pro6S"))
```

映射的遍历与集合类似，也有 for-in 循环、迭代器遍历、forEach 遍历三种遍历手段。但是由于映射的元素是一个键值对，因此它的遍历方式与集合稍有不同，详述如下：

1. for-in 循环

for-in 语句取出来的是映射的元素键值对，若要获取该元素的键名，还需访问元素的 key 属性；若要获取该元素的键值，还需访问元素的 value 属性。下面是在映射中运用 for-in 循环的代码例子：

```
btn_map_for.setOnClickListener {
    var desc = ""
    //使用 for-in 语句循环取出映射中的每条记录
    for (item in goodsMutMap) {
        //item.key 表示该配对的键，即厂家名称；item.value 表示该配对的值，即手机名称
        desc = "${desc}厂家：${item.key}，名称：${item.value}\n"
    }
    tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
}
```

上述通过 for-in 语句遍历映射的运行效果如图 2-10 所示，可见映射内部每个元素的键名与键值都被获取到了。

图 2-10　Kotlin 映射的遍历结果

2. 迭代器遍历

映射的迭代器通过 next 函数得到下一个元素，接着需访问该元素的 key 属性获取键名，访问该元素的 value 属性获取键值。下面是在映射中运用迭代器遍历的代码例子：

```
btn_map_iterator.setOnClickListener {
    var desc = ""
    val iterator = goodsMutMap.iterator()
    //如果迭代器还存在下一个节点，就继续取出下一个节点的记录
    while (iterator.hasNext()) {
        val item = iterator.next()
        desc = "${desc}厂家：${item.key}，名称：${item.value}\n"
    }
    tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
}
```

3. forEach 遍历

映射的 forEach 方法内部依旧采用匿名函数的形式，同时把元素的 key 和 value 作为匿名函数的输入参数。不过映射的 forEach 函数需要 API 24 及以上版本支持，开发时注意修改编译配置。下面是在映射中运用 forEach 遍历的代码例子：

```
btn_map_foreach.setOnClickListener {
    var desc = ""
    //映射的 forEach 函数需要 API 24 及以上版本支持
    //forEach 内部使用 key 指代每条记录的键，使用 value 指代每条记录的值
```

```
        goodsMap.forEach { key, value -> desc = "${desc}厂家：${key}，名称：
${value}\n" }
        tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
        //tv_map_result.text = "Map 的 forEach 函数需要 API 24 及以上版本支持"
    }
```

2.5 小　　结

本章介绍了 Kotlin 开发涉及的几种常见数据类型，包括以整型、浮点型、布尔型、字符型为代表的基本数据类型，还有这些基本数据类型数组的概念和运用，以及字符串的各种常见用法，最后是几种容器的常见操作方式。

通过本章的学习，读者应能掌握以下技能：

（1）学会 Kotlin 对基本数据类型的变量定义以及变量之间的类型转换。

（2）学会 Kotlin 对基本类型数组的声明方式以及数组变量的常见用法。

（3）学会 Kotlin 对字符串的各种处理操作以及字符串模板的书写格式。

（4）学会 Kotlin 对容器的声明方式及其增、删、改、查操作，包括集合、队列、映射三种基本容器。