第5章

数　值

Kotlin 提供了多种类型来处理数值的计算，包括多种整数类型和**浮点数**（**floating point number**）类型（也称为带小数点的数值）等。

本章学习 Kotlin 如何处理这两种类型的变量。bounty-board 项目在本章中不会进行任何改动；相反，将使用 REPL 对代码进行评估。第 6 章中再继续 bounty-board 项目的开发。

5.1　数值类型

Kotlin 可以支持多种数值类型。无论面向的是哪个平台，这些数值类型的规则都是不变的。从 Kotlin 1.5 版本开始，Kotlin 中的数值类型就包含两种：**有符号**（**signed**）和**无符号**（**unsigned**）。有符号数值既可以表示正数也可以表示负数。无符号数值只能表示正数。我们首先讨论有符号数值，在 5.6 节再讨论无符号数值。

除了数值是否有符号之外，Kotlin 的数值类型之间的关键区别还包括：数值类型在内存中分配的空间大小有所不同，也就是说，能表示的最小值和最大值不同。

如果熟悉 Java，这些规则应该就不会陌生。Kotlin 数值类型的规则与 Java 是相同的。对于那些熟悉 JavaScript 的读者来说，可能会惊讶地看到 Java 的数值类型有很多种，而 JavaScript 仅有一种数值类型。Kotlin 的每种数值类型都有其自身的含义。

对于打算使用 Kotlin/Native 的用户来说，也应注意每种类型所分配的内存大小。不管面向哪个平台，Kotlin 数值类型都会被分配相同大小的内存（在 C 语言中，根据程序编译方式的不同，其 Int 类型被分配的内存大小并不相同）。

表 5.1 给出了 Kotlin 中的一些数值类型、每种类型占用的内存大小以及支持的最大值和最小值。

表 5.1　常用数值类型

类　型	位　数	最 大 值	最 小 值
Byte	8	127	－128
Short	16	32 767	－32 768
Int	32	2 147 483 647	－2 147 483 648
Long	64	9 223 372 036 854 775 807	－9 223 372 036 854 775 808
Float	32	3.4028235E38	1.4E-45
Double	64	1.7976931348623157E308	4.9E-324

不同数值类型所占的比特位数与其最大值和最小值之间是紧密联系的。计算机采用固定位数的二

进制形式来存储整数，每个比特位存储一位二进制的 0 或 1。

为了表示数值，Kotlin 根据数值类型的不同分配不同数量的比特位。对于有符号数值，最左边的一位表示符号(0 表示正，1 表示负)。剩余的位分别表示 2 的幂次方，最右边的位是 2^0。要计算二进制数的值，需将各位上的数按 2 的幂次方相加。

图 5.1 给出了数值 42 的二进制形式。

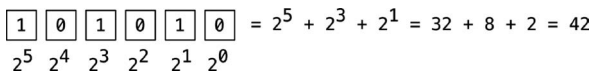

图 5.1 数值 42 的二进制形式

由于数值类型 **Int** 占 32 位，所以 **Int** 类型可以存储的最大数，以二进制形式表示就是 31 个 1(最左边的一位代表符号)。所以，将所有 2 的幂次方相加得到 2 147 483 647，这是 Kotlin 中 **Int** 类型所能表示的最大值。

因为数值类型所占的位数决定了可以表示的最大值和最小值，所以两种类型之间的区别就在于可用于表示数字的位数的多少。由于数值类型 **Long** 占 64 位而不是 32 位，因此 **Long** 类型可以表示更大的数值(2^{63})。

对于数值类型 **Short** 和 **Byte** 来说，此处所谓的长与短(long and short)，在表示传统数值时，既不常用 **Short** 类型也不常用 **Byte** 类型。这两种类型主要用于一些特殊情形，并支持互操作性，通常与遗留程序(legacy programs)一起使用。

例如，当从文件读取数据流或处理图形时，可能就会用到 **Byte** 类型(彩色像素通常表示为 3 字节，每字节代表 RGB 中的一种颜色)。在与不支持 32 位指令的 CPU 的本机代码交互时，有时会用到 **Short** 类型。但是，大多数情形下，整数用 **Int** 类型表示，如果需要表示更大的数值，则需使用 **Long** 类型。

5.2 整数

在第 2 章中已经介绍了整数就是没有小数点的数值，在 Kotlin 中可以用 **Int** 类型表示。**Int** 类型很适合用于表示“物”的数量或进行计数，如玩家的技能水平、经验值、蜂蜜酒的剩余量或玩家拥有的金币和银币的数量等。

为了获得更多关于 **Int** 类型的第一手经验，下面将使用 REPL 执行一些算术运算。单击 Tools→Kotlin→Kotlin REPL 命令选项。

在 REPL 中，输入程序清单 5.1 所示的运算，并先预测一下计算结果。

程序清单 5.1 执行整数的算术运算(REPL)

```
2 + 4 * 5
```

按下组合键 Ctrl+Enter 运行以上表达式，REPL 的输出结果为 22。

如上所示，Kotlin 的乘法运算符(*、/和%)优先于加法运算符(+、-)，与普通数学中的运算优先级是一致的。也就是说，在加 2 之前，先对乘法 4 * 5 进行求解。

如果需要指定不同的运算顺序，可以使用括号对运算进行分组。正如在第 3 章中看到的，Kotlin 会首先计算嵌套在括号内的表达式。

现在，在 REPL 中输入程序清单 5.2 所示的运算。同样，不妨先预测一下计算结果。

程序清单 5.2 执行整数的除法运算(REPL)

```
9 / 5
```

计算以上的表达式，可能期望的结果为 1.8。但是，REPL 实际输出的结果却是 1。当用一个整数除以另一个整数时，结果仍会是整数。如果整数除法运算的结果不是整数，Kotlin 将截断小数点后的所有数字。同样的，如果要 REPL 计算 9/−5 的值，结果将是−1。

整数除法运算的结果总是会四舍五入。这种截断操作是悄无声息的，因此，如果小数点后的数字对应用程序来说很重要，在执行整数除法时需要格外小心。

计算除法余数的一种方法是使用**模运算符**（**modulus operator**）%，也称为**余数运算符**（**remainder operator**），当一个数除以另一个数时，该运算符会得到其余数。例如，9%5 将返回结果 4。

以上适用于数值为整数时的运算。对于十进制数值，可以使用浮点类型。

5.3 浮点数

在 Kotlin 中，**Float** 类型和 **Double** 类型是两种可以表示十进制数字的数值类型。这些数值也称为浮点数，因为小数点可以出现在任何位置，也就是说，小数点是“浮动的”（而不是固定的），这取决于数值的数量级。

Double 类型是双精度浮点数（double-precision floating point number）的缩写。**Double** 类型使用的位数是常规 **Float** 类型的 2 倍，由此而得名，并且它可以更精确地存储十进制数。

Kotlin 中的浮点数也可以用来表示一些特殊值，如无穷大、负无穷大和 NaN（Not A Number 的缩写）。这些值通常在执行非法或未定义的操作时返回，如除以零（返回无穷大或负无穷大）或负数的平方根（返回 NaN）。可以通过在代码中引用 Double.POSITIVE_INFINITY（或 Float.POSITVE_INFINITY）、Double.NEGATIVE_ININITY（或 Float.NEGATIVE_INFINTITY）和 Double.NaN（或 Float.NaN）来访问这些特殊值。

在 REPL 中，重新审视 9/5 这一整数除法表达式。为了使用浮点除法表示这个表达式，需要告知 Kotlin 这些数值是浮点数而不是整数，一种方法是采用小数点表示这些数值，具体如程序清单 5.3 所示。

程序清单 5.3 执行浮点除法（REPL）

```
9.0 / 5.0
```

计算此表达式。REPL 输出 kotlin.Double＝1.8，该结果很符合预期。注意，此表达式的类型是 **Double** 类型。默认情形下，Kotlin 更喜欢用 **Double** 类型，但可以通过在数值中添加 f 后缀来明确要求将数值视为 **Float** 类型：9.0f/5.0f。

注意：如果使用 f 后缀指定数值是浮点数，也可以省略两个数值中的.0。实际上，甚至可以将相同的运算表示为 9f/5，因为 Kotlin 会在至少有一个操作数是浮点数时使用浮点除法来运算。

之前提到过浮点数是有“精度”的。要想了解其含义，在 REPL 中输入以下表达式。

程序清单 5.4 造成浮点数的精度出错（REPL）

```
0.01f * 5
```

直观地说，这个表达式应该返回 0.05。但是，当计算该表达式时，REPL 会输出一个结果为 0.0499999997 的值。现在采用程序清单 5.5 的代码比较一下这个结果是否等于 0.05。

程序清单 5.5 检查浮点数的精确相等性（REPL）

```
0.01f * 5 == 0.05f
```

执行该语句，REPL 输出的结果将会是 false。为什么呢？

整数的每一位都有特定的含义，永远不会改变。但是对于浮点数来说，情况就不是那么简单了。从二进制的角度来看，浮点数由一个符号位和两个附加的位集组成：第一个位集确定数值大小的指数；第二个位集确定所表示的数值的有效位数。

由此可见，浮点数不能精确地表示每个数值，它们只是**近似值**(**approximation**)。虽然 0.05 可以用 **Float** 类型精确表示，但 0.01 不能，最接近 0.01 的 **Float** 类型存储值是 0.009999999776482582。当做乘法 0.01f * 5 时，精度的损失会影响结果，从而导致不准确的(但非常接近的!)结果。

为了避免这种精度问题，可以采用以下几个选项。

(1) **首选 Double 类型而不是 Float 型**：通过使用更高精度的浮点类型，就可以避免浮点精度错误的问题，但代价是内存使用量增加。注意，使用 **Double** 类型仍然不足以完全避免这个问题(例如，在 REPL 中计算 10.1-5.9)。

(2) **四舍五入浮点数值**(**round floating point values**)：如果确切知道一个浮点数应该有多少位小数，则可以相应地进行四舍五入。Kotlin 提供的 API 可以输出具有特定小数位数的十进制值，并且有 **round**()函数可以将数值四舍五入到最接近的整数。

如果将 **round**()函数与乘法和除法运算相结合，则可以四舍五入到特定的小数位数。例如，round (number * 100)/100 会将 number 的值四舍五入到两位小数。

(3) **使用精度更高的其他数据类型**(**prefer another data type with higher precision**)：如果需要存储的是一个关键的十进制值，而四舍五入和精度损失是不可接受的(例如，假设正在开发一款银行软件)，可能需要一个更强大的数据类型。

有时，可以使用 **Int** 类型表示可能是十进制的数据类型。例如，假设想存储用户的银行账户余额，可以使用 **Int** 类型而不是 **Double** 类型跟踪以美分为单位的值。

作为最后的选择，在面向 JVM 的应用中，可以使用 **BigDecimal** 类型。**BigDecimal** 类型在执行四舍五入和运算方面更加强大。与基本数值类型相比，它通过增加复杂性回避了精度错误。**BigDecimal** 类型在存储数值和执行算术运算时也会比浮点数用到更多的资源(如果 Kotlin 代码面向的是 iOS 或 macOS，那么 **Decimal** 类型是 **BigDecimal** 类型的等效类)。

5.4 格式化双精度数值

回到 bounty-board 奇幻游戏世界中。假设想要追踪 Madrigal 在游戏中拥有的货币数量，可编写如下代码：

```
val currentBalance = 1120.40
println(currentBalance)
```

运行此代码，输出 Madrigal 的银行账户余额为 1120.4，且未标明货币单位。此时，更好的做法是对余额进行类似货币的格式化，使其看起来更像是货币。对于北美国家而言，余额应该显示为 1120.40 美元。可以使用 **format**()函数对一个双精度数值进行格式化处理，包括添加货币或其他符号、千位分隔符以及显示的小数位数等。

首先确定需要的小数位数。在 REPL 中运行程序 5.6 所示的代码。

注意：使用了点语法(dot syntax)来调用 **format** 函数。每当调用作为类型定义的一部分的函数时，都可以使用点语法。

程序清单 5.6 格式化一个双精度数值(REPL)

```
val currentBalance = 1120.40
```

```
println(" % .2f".format(currentBalance))
```

REPL 的输出为 1120.40，看起来可读性更好了。

在对 **format**()函数的调用中指定了一个**格式字符串**(**format string**)“%.2f”。格式字符串使用特殊的字符序列定义数据的格式。此处定义的特定格式字符串指定要将浮点数四舍五入到小数第二位，然后将格式化后的值作为实际参数传递给 **format**()函数。

这些格式字符串使用的是与 Java、C/C++、Ruby 及许多其他语言中的标准字符串格式相同的样式。关于格式字符串规范的详细内容，可参考 Java API 文档。

若想添加逗号和美元符号，可以将格式字符串修改为“$ %,.2f”，如程序清单 5.7 所示。

程序清单 5.7 添加货币格式(REPL)

```
val currentBalance = 1120.40
println(" $ %,.2f".format(currentBalance))
```

使用 **format**()函数有几个注意事项。首先，这样做有可能会使应用程序的本地化变得困难。假如将 Madrigal 的余额显示在用户所在地的语言环境中，那么就需要将美元符号替换为用户所在地适用的货币符号。此外，许多国家使用逗号作为小数点，使用句点作为千位分隔符，这也会增加本地化工作的复杂性。其次，**format**()函数仅在面向 JVM 时可用(本书编写时)。如果需要面向其他平台，则需要使用不同的方法来格式化数值。

为了解决以上两个问题，可以使用特定于平台的格式化 API。在 Java 中，使用 **NumberFormat** 类可获得相同的效果，代码如下：

```
val currentBalance = 1120.40
val formatter = NumberFormat.getCurrencyInstance()
val formattedBalance = formatter.format(currentBalance)
println("Madrigal's life savings: " + formattedBalance)
```

注意：*若想在 REPL 中运行此代码，还需要添加* importjava.text.NumberFormat *行。*

这样，将自动将 Madrigal 的储蓄金额转换为适合用户的格式化字符串，具体取决于用户地区的偏好设置。

除了 Java 风格之外，Android 在 android.icu.text 包中有自己的 **NumberFormat** 类。这两个 **NumberFormat** 类均可以用于获取类似于区域设置的货币格式的实例。同样，如果 Kotlin 代码面向的是 iOS 或 macOS，可以使用 **NSNumberFormatter** 类。同时，对于 Kotlin/JS，可以使用 **Intl.NumberFormat** 类来满足数值格式的需求。

本书第六部分介绍如何将 **NSNumberFormatter** 类和 **Intl.NumberFormat** 类与 Kotlin/Native 和 Kotlin/JS 结合起来使用。

5.5 在数值类型之间进行转换

有时，需要在浮点数和整数之间进行转换。例如，若使用经验值表示玩家的技能水平，而不是像 bounty-board 项目中那样直接跟踪玩家的水平，代码如下所示：

```
var experiencePoints = 460.25
val playerLevel = experiencePoints / 100
```

变量 playerLevel 的类型将被推断为 **Double** 类型，当运行此代码时，其值将被设置为 4.6025。但这可能并不是程序开发者想要的。无论经验值是 400 点还是 499 点，从确定任务难度的角度来说，4 级技

能水平只需用 4 来表示。为进一步说明该细节，可以将变量 playerLevel 设置为 **Int** 类型，而不是 **Double** 类型。

若想要进行此转换，可以对表达式调用 **toInt**()，具体如程序清单 5.8 所示。

程序清单 5.8　将 Double 类型转换为 Int 类型(REPL)

```
var experiencePoints = 460.25
val playerLevel = (experiencePoints / 100).toInt()
println(playerLevel)
```

运行此代码，REPL 输出结果为 4。当以此方式将 **Double** 类型转换为 **Int** 类型时，遵循的是与整数除法相同的规则：小数点后的值将被截断，数值四舍五入为零。

该行为有时被称为**精度损失**(**loss of precision**)。因为想用整数表示包含小数部分的双精度数值，而整数能表示的精度不够，所以，可能会损失部分原始数据。

有时，这种截断是不可取的。在有些情形下，需要将 **Double** 类型**四舍五入**(**rounded**)为 **Int** 类型，而不是转换。幸运的是，Kotlin 中还有一个 **roundToInt**()函数可以实现这一点。将程序清单 5.9 中的代码输入 REPL，查看该函数的运行情况。

程序清单 5.9　将 Double 类型四舍五入为 Int 类型(REPL)

```
import kotlin.math.roundToInt
val distanceToObjective = 4.6
println("The objective is about " + distanceToObjective.roundToInt() + " miles away")
```

运行以上代码，可以看到输出为 The objective is about 5 miles away。Kotlin 输出的是 5 而不是 5.0，表示该值是整数而不是浮点数。该函数只需一步就将 **Double** 类型数值四舍五入并转换为 **Int** 类型。这样，根据具体的需要，**roundToInt**()函数就成了 **toInt**()函数的替代。

注意：如果需要将 **Int** 类型转换为 **Double** 类型，可以使用相应的 **toDouble**()函数。当调用此函数时，将返回小数点后为零的双精度值。

本章已经介绍了 Kotlin 的数值类型，并了解了 Kotlin 如何处理两大类数值：整数和双精度数。还介绍了如何在不同类型之间进行转换，以及每种类型可表示的数值大小。在第 6 章将介绍 Kotlin 的字符串。

5.6　好奇之处：无符号数

Kotlin 1.5 版本引入了无符号数值类型作为一种稳定的语言特性。这与迄今为止见过的数值类型非常相似，唯一的区别是无符号数不能表示负数。表 5.2 给出了 Kotlin 中的无符号数值类型。

表 5.2　Kotlin 中的无符号数值类型

类　型	位　数	最 大 值	最 小 值
UByte	8	255	0
UShort	16	65 535	0
UInt	32	4 294 967 295	0
ULong	64	18 446 744 073 709 551 615	0

这些无符号数值类型与本章之前介绍的有符号数值类型之间存在相似之处，但也有不同之处，包括浮点数值类型 **Float** 和 **Double** 没有对应的无符号类型，本节末尾会解释其中的原因。

每一种整数类型(**Byte**、**Short**、**Int** 和 **Long**)都有一个对应的无符号类型(**UByte**、**UShort**、**UInt** 和

ULong)。分配给这些有符号数和无符号数类型的位数是相同的,例如 **Int** 类型和 **UInt** 类型均占 32 位。所有无符号数的最小值均为 0,但是,其最大值远高于对应的有符号数。

无符号数值类型使用起来比有符号数值要麻烦一些。将变量 playerLevel 声明为 **UInt** 类型并赋值为 5,代码如下所示:

```
var playerLevel: UInt = 5.toUInt()
```

也可以在数值之后加上字母 u,以将其标记为无符号数,如下所示:

```
var playerLevel: UInt = 5u
```

如果需要,可以删除显式类型信息(: UInt),但必须使用无符号后缀或调用 **toUInt**()函数将数字标记为无符号数。

Kotlin 不会在有符号类型和无符号类型之间进行隐式转换。这也会影响对无符号类型的操作,例如:

```
var playerLevel = 5u
val levelsToAdd = 1
playerLevel += 1.toUInt()              // Adding a UInt to a UInt is allowed
playerLevel += 1u                      // Also allowed (shorthand for the line above)
playerLevel += 1                       // Compiler error: you must convert 1 to a UInt first
playerLevel += levelsToAdd             // Compiler error: cannot add an Int and a UInt

print(playerLevel * 10u)               // Allowed
print(playerLevel * 10)                // Compiler error
```

因为 Kotlin 不会自动在有符号和无符号类型之间进行转换,所以,要么程序中的大量变量是无符号数,要么根据需要使用 **toUInt**()和 **toInt**()函数来回进行转换。

在一些特定的情形中,无符号数可能非常有用,例如,当需要保证某变量为正数,对函数参数强制执行规则,或者使用的无符号数是平台特定类型时。但无符号数并非无懈可击,如果不小心,仍然有可能在程序中出现意外的情形。例如,假设将变量 playerLevel 设置为 **UInt** 类型并将其赋值为 −1,会怎么样呢:

```
val playerLevel = (-1).toUInt()
println(playerLevel)                        // Prints 4294967295
```

如果从任何数值类型可以表示的最小值中减去 1,则将会“翻转回”(rolls over)该类型所能表示的最大值,这称为**整数下溢**(**integer underflow**)。本例中,**UInt** 类型或任何无符号类型可以表示的最小值均为 0,因此减去 1 会得到可能的最大 **UInt** 类型值为 4 294 967 295。这种意外的结果是将有符号类型和无符号类型一起使用的缺陷之一。

注意:当处理的有符号数非常大时,也可能发生类似的情形,称为**整数溢出**(**integer overflow**)。如果在 Int.MAX_VALUE 上加 1,则会换行并得到 Int.MIN_VALU。

为什么无符号数的类型与本章中学习的有符号数值的类型不同呢?为什么要如此大费周章呢?在 Kotlin 中,无符号整数的实现方式与有符号整数不同。实际上,无符号整数是用有符号整数实现的。

当使用到无符号整数时,实际上是告诉 Kotlin 使用有符号整数,但将其比特位看作无符号整数的比特位。与使用有符号整数相比,这样做的好处是不会产生任何内存损失,并且在对无符号整数执行操作时,性能差异几乎可以忽略不计。

那么,为什么 Kotlin 不支持无符号浮点数呢?无符号浮点数是对其有符号变体的重新解释,尽管可以重新解释浮点数值并改变符号位的含义,但这是非常不符合常规的做法,而且 Kotlin 不支持这种操作。

由于执行浮点数运算的复杂性，计算机有专门的组件可高效地执行这些操作。如果要重新赋予符号位的含义，将无法使用这些硬件加速器，从而导致应用程序的性能下降。为了避免该问题，几乎没有一种带有无符号数的语言支持无符号浮点数。

因为 Kotlin 中的无符号类型是用有符号类型来实现的，所以，有符号类型往往被视为高人一等。与无符号类型相比，有符号类型用到的时候要多很多。同时，它们还有一些古怪之处（例如，有时 IntelliJ 在 REPL 中输出结果时，会将 **UInt** 类型视为 **Int** 类型）。

是否使用无符号类型，以及在何处使用，取决于自己。在 bounty-board 项目中选择不使用它们，但是可以认为有一些变量，如 playerLevel，应该是无符号的，因为它们永远不应该是负数。

5.7 好奇之处：位运算

早些时候，学习了数值的二进制表示，随时可以得到一个数值的二进制表示。例如，求解整数 42 的二进制表示如下：

```
42.toString(radix = 2)
"101010"
```

注意：参数 radix 表示输出数值所需的基数。通过指定 2，要求计算以 2 为基数或二进制的数值。默认基数为 10，将会输出十进制数值。另一个常用的基数为 16，将返回十六进制的字符串。

Kotlin 提供了用于对数值执行二进制操作的函数，称为按位操作，包括可能在其他语言中熟悉的操作，如 Java、C 和 JavaScript 等。表 5.3 给出了 Kotlin 中常用的二进制运算。

表 5.3 二进制运算

函　数	描　述	示　例
shl(bitcount)	按位计数向左移动位	`42.shl(2)` *10101000*
shr(bitcount)	按位计数向右移动位	`42.shr(2)` *1010*
inv()	按位反转	`42.inv()` *11111111111111111111111111010101*
xor(number)	比较两个二进制数值，并对相应的位执行逻辑“异或”运算	`42.xor(33)` *001011*
and(number)	比较两个二进制数值，并对相应的位执行逻辑“与”运算	`42.and(10)` *1010*