第 3 章

Kotlin 快速入门

Kotlin 是开发 Android 应用程序的主要开发语言，是 Android 应用开发的基础。Kotlin 是由 JetBrains 研发的跨平台静态编程语言，既可以编译为 Java 字节码，也可以编译为 JavaScript，以便提供跨平台能力。2016 年，Kotlin 发布了其 1.0 版本。短短 1 年后的 2017 年，谷歌就将其作为 Android 应用开发的一级开发语言。那么 Kotlin 为什么具有如此的魅力呢？这主要归功于 Kotlin 如下主要特性。

(1) Kotlin 的语法更加现代：Kotlin 和新生的 Python、Go 等一同成为现代编程语言的明星。这意味着 Kotlin 更加简洁，拥有众多好用的语法糖，如 Lambda 表达式、尾随闭包、扩展函数等。

(2) Kotlin 更加安全：空值安全特性能够避免许多空指针异常的调试。

(3) Kotlin 支持协程(Coroutine)：通过协程可以轻松地构建异步非阻塞代码，而这在移动开发中非常常用。

(4) Kotlin 是多范式编程语言：不仅具有完整的面向对象编程特性，也提供了函数式编程方法。

本章介绍 Kotlin 的基本特性和基本用法，主要面向没有任何 Kotlin 开发基础的读者参阅，以便能够快速进入 ArcGIS Maps SDK 的学习中。如果读者有 Kotlin 编程基础，则可跳过本章。本章核心知识点如下：

- Kotlin 基础语法
- 函数和 Lambda 表达式
- 字符串
- 集合类型
- 面向对象编程
- 空安全

3.1 Kotlin 基本语法

在 Android 编程中，也可以同时使用 Kotlin 和 Java 两种语言构建应用程序。不过，似

乎在互联网上 Java 拥有更好的“群众基础”，学习资料也更多，那么为什么还要学习 Kotlin 呢？这主要有以下两个原因：

（1）Kotlin 具有更加现代的语法，更简洁、更好用，已经成为 Android 开发的首选（第一）语言。越来越多的 Android 开发者使用 Kotlin 语言。

（2）ArcGIS Maps SDK 已经不再支持 Java 编程语言，仅支持 Kotlin 语言编程。

实际上，Kotlin 是参照 Java 开发者编程思维而研发的编程语言，因此可以在 Kotlin 中寻找到 Java 的影子。对于有 Java 语言基础的开发者，能够很容易地切换到 Kotlin 编程应用中。本节介绍 Kotlin 的变量、函数等基本语法。

3.1.1　运行和调试 Kotlin 代码

为了学习 Kotlin 语言，需要选择合适的 IDE（集成开发环境）。通过以下方式可以编写 Kotlin 程序：

（1）使用 Android Studio。由于 Android Studio 无法创建纯 Kotlin 语言的项目，只能在 Android 工程中创建 Kotlin 源代码文件。

（2）使用独立的 IDE，如 IntelliJ IDEA、Visual Studio Code 或 Vim 等。IntelliJ IDEA 是官方推荐的 Kotlin 语言 IDE，可以在 https://www.jetbrains.com/zh-cn/idea/download 网站下载使用。

（3）在 play.kotlinlang.org 网站上编写 Kotlin 语言程序，如图 3-1 所示。

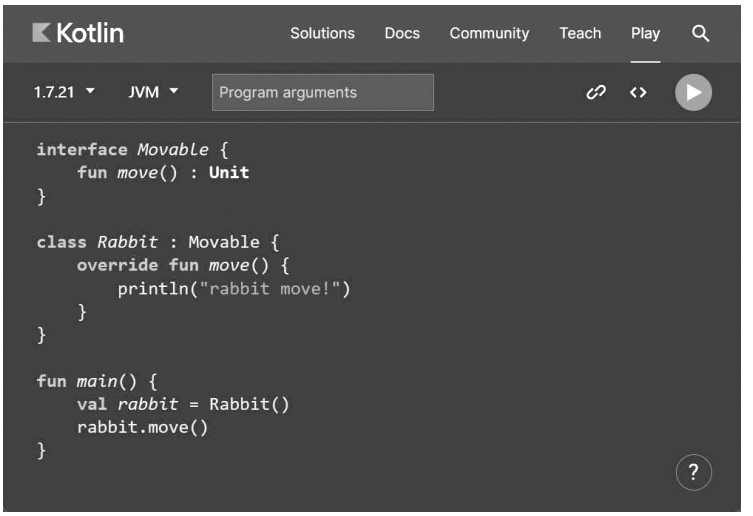

图 3-1　在线编写 Kotlin 程序

本节主要介绍在 Android Studio 中创建 Kotlin 源文件的方法。创建或打开任意的 Android 工程，在工程窗格中的任意包（Package）上右击，选择 New→Kotlin File/Class，此时会弹出 New Kotlin Class/File 对话框，如图 3-2 所示。

在 Name 文本框中输入任意名称（如 kotlin），并选择类型为 File，按 Enter 键即可创建

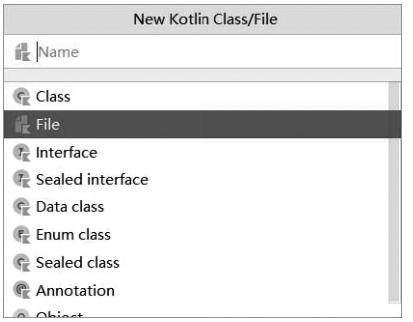

图 3-2 创建 Kotlin 类/文件

kotlin.kt 文件。此时,即可在该文件中编写 Kotlin 语言代码。在默认情况下,该文件中包含了包的声明,代码如下:

```
package edu.hebtu.kotlintest
```

下面,添加 Kotlin 程序的主函数 main(程序的入口),输出“hello, kotlin!”字符串,代码如下:

```
package edu.hebtu.kotlintest

fun main() {
    println("hello, kotlin!")
}
```

随后,单击 fun main 左侧的▶按钮,并在弹出的菜单中选择 Run 'KotlinKt'选项即可运行程序,如图 3-3 所示。

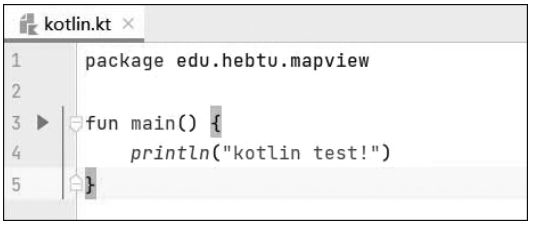

图 3-3 运行 Kotlin 程序

程序运行后,程序的输出结果会显示在 Run 窗体中,如图 3-4 所示。

图 3-4 Kotlin 程序运行结果

开发者可以沿用上述搭建好的开发环境进行学习和调试。

3.1.2　常量和变量

常量和变量用于承载程序中的各种数据，本节介绍常量和变量的定义，并实现简单的基本数学运算。

1. 常量和变量的基本用法

在 Kotlin 中，常量用 val 关键字(value 的简写)定义，其值在初始化后不能再次修改；变量用 var 定义(variable 的简写)，值在初始化后可再修改。

注意　本章所有的代码均可在本书配套实例代码 code/chapter03 中找到，并且在后文中的每个实例前均包括文件位置的注释，如 code/chapter03/example3_1.kt 表示该代码文件的名称为 example3_1.kt。

分别定义整型常量 constant 和浮点型变量 variable，并输出相应的值，代码如下：

```
//code/chapter03/example3_1.kt
fun main() {
    val constant =100                          //声明并初始化 constant 整型常量
    println("The constant is " +constant)      //输出 constant 值
    var variable =100.0                        //声明并初始化 variable 浮点型变量
    println("The variable is " +variable)      //输出 variable 值
}
```

在 main 函数中，每行为一个语句。这些语句会按照先后顺序执行。在上述程序中，首先定义了 constant 常量，并初始化为 100，然后输出该 constant 常量的值。再后，定义了 variable 变量，并初始化为 100.0，最后输出该 variable 变量的值。常量 constant 的值在初始化后就无法通过赋值表达式修改了。

注意　两个斜线“//”表示注释，该行后面的任何字符串不参与程序的编译。通过注释可以说明代码的意义，记录编程的过程。除了“//”注释以外，还可以使用“/* 注释内容 */”的方式实现多行注释。

编译并运行程序，输出的结果如下：

```
The constant is 100
The variable is 100.0
```

2. 基本数据类型

Kotlin 中包含了 Int、Long 等 8 种基本数据类型，这 8 种基本数据类型和 Java 中的基本数据类型相对应，如表 3-1 所示。

表 3-1 Kotlin 和 Java 基本数据类型

数据类型	Java	Kotlin	值的大小（字节数）	字面量（以十进制为例）
字节型	byte	Byte	1	无字面量，可使用整型向上转型
短整型	short	Short	2	无字面量，可使用整型向上转型
整型	int	Int	4	直接使用数字，如 30
长整型	long	Long	8	数字后接 L，如 30L
单精度浮点型	float	Float	4	浮点值后接 f(或 F)，如 3.0f、12.4e2F
双精度浮点型	double	Double	8	直接使用浮点值，如 3.2e10
布尔型	boolean	Boolean	1	true 或 false
字符型	char	Char	2	用单引号表示，如'a'

和 Java 不同的是，Kotlin 数据类型关键字的首字母为大写，而 Java 数据类型关键字均为小写。

定义这些不同类型的变量，代码如下：

```
//code/chapter03/example3_2.kt
fun main() {
    var oneByte : Byte =10                  //字节型
    println(oneByte is Byte)
    var short : Short =1024                 //短整型
    println(short is Short)
    var int =1314520                        //整型
    println(int is Int)
    var long =80000000000000L               //长整型
    println(long is Long)
    var pi =3.1415926f                      //单精度浮点型
    println(pi is Float)
    var lightSpeed =2.99792458e8            //双精度浮点型
    println(lightSpeed is Double)
    var bool =false                         //布尔型
    println(bool is Boolean)
    var ch ='e'                             //字符型
    println(ch is Char)
}
```

通过 is 关键字可以判断变量或常量的类型，其返回值为布尔型的判断结果。编译并执行上述程序会在控制台中输出 8 个 true。

注意 Kotlin 中，数值支持十进制、十六进制(以 0x 开头)，但是不支持八进制。

在声明常量和变量时，如果没有同时初始化，则必须声明其类型，代码如下：

```
//code/chapter03/example3_3.kt
fun main() {
    val constant : Int                          //声明 constant 整型常量
    constant =100                               //初始化 constant
    println("The constant is " +constant)       //输出 constant 值
    var variable : Double                       //声明 variable 双精度浮点型变量
    variable =100.0                             //初始化 variable
    println("The variable is " +variable)       //输出 variable 值
}
```

编译并运行程序，输出的结果如下：

```
The constant is 100
The variable is 100.0
```

3. 数学运算

对于整型（包括字节型、短整型、整型和长整型）和浮点型（包括单精度浮点型和双精度浮点型）来讲，可以通过＋、－、＊、/和％等运算符完成加、减、乘、除和取余等基本运算，但是，参与计算的变量（或常量、值）必须为同一种类型，这主要分为以下 3 种情况。

（1）参与计算的值均为整型：自动将精度较低的整型转换为精度较高的整型后参与计算。如 Short 和 Int 类型的值参与计算，那么会将 Short 类型的值转换为 Int 类型的值后再进行计算。

（2）参与计算的值均为浮点型：与整型类似，自动将精度较低的浮点型转换为精度较高的浮点型后参与计算。

（3）整型值和浮点型值参与计算：无法计算会导致编译报错。此时，可以通过类型转换的方式将整型值转换为浮点型值，或者将浮点型值转换为整型值后再进行计算。

Kotlin 是一门静态强类型编程语言，因此对于任意的整型或者浮点型来讲都需要进行显式转换。在实际操作中，可使用如表 3-2 所示的函数进行类型转换。

表 3-2　整型和浮点型的类型转换函数

函　　数	描　　述
toByte()：Byte	转换为字节型
toShort()：Short	转换为短整型
toInt()：Int	转换为整型
toLong()：Long	转换为长整型
toFloat()：Float	转换为单精度浮点型
toDouble()：Double	转换为双精度浮点型
toChar()：Char	转换为字符型

例如，计算整型值 657260 和浮点型值 2.0 之间的乘法，代码如下：

```
//code/chapter03/example3_4.kt
fun main() {
    val a =657260                //整型
    val b =2.0                   //浮点型
    //将 b 转换为整型后参与计算
    println(a * b.toInt())
}
```

由于常量 a 和 b 为不同类型，因此需要通过类型转换函数将其中一个常量的类型转换为另一个常量的类型。这里将浮点型常量 b 转换为整型后再进行乘法运算。编译并运行程序，输出的结果如下：

```
1314520
```

3.1.3 函数

函数是业务逻辑的载体，本节介绍函数及其调用、函数重载的方法。

1. 函数

函数通过 fun 关键字定义，包括函数头和函数体两个主要部分，如图 3-5 所示。

函数头包含了函数名称、函数参数列表和函数返回定义。

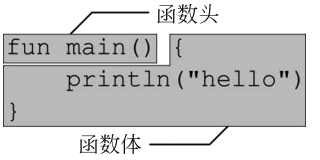

图 3-5 函数由函数头和函数体组成

(1) 函数名称：通过标识符定义。在 Kotlin 中，标识符可以由下画线、字母和数字组成，但是首字符不能是数字。函数名称为 main 的函数为程序的入口函数。对于一个程序(包)而言，只能存在一个 main 函数。

(2) 参数列表：用于定义函数的输入，在函数名称后的小括号内定义。在上述的 main 函数中，没有参数列表，在调用该函数时无须函数输入。

(3) 函数返回：用于定义函数的输出。当没有定义函数的返回时，函数返回 Unit 对象。

函数体包含了函数具体的业务逻辑，也是函数的主体部分。定义两个整型相乘的函数 multiply，通过参数列表传入两个整型值，并返回这两个值的乘积，代码如下：

```
//code/chapter03/example3_5.kt
//主函数
fun main() {
    //计算 10 * 5
    val res =multiply(10, 5)
    //输出结果
    println("10 * 5 =" +res)
}
```

```
//两数相乘函数
fun multiply(a : Int, b : Int) : Int {
    //通过 return 关键字返回结果
    return a * b
}
```

在 main 函数中通过函数调用表达式 multiply(10, 5)调用了 multiply 函数，其中 10 和 5 分别为该函数的两个参数 a 和 b。随后执行 multiply 函数中的函数体，计算其值为 50，并将该值作为返回值返回，因此表达式 multiply(10, 5)的值为 50，并将其赋值给 res 变量，整个调用过程如图 3-6 所示。

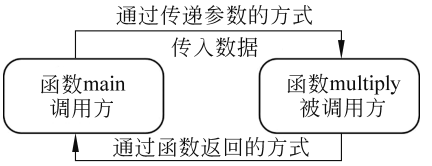

图 3-6　函数的调用

编译并运行程序，输出的结果如下：

```
10 * 5 =50
```

2. 函数的重载

Kotlin 函数支持函数重载。同名函数可以具有不同的参数列表，通常用于针对不同的输入完成相同或者类似的功能。例如，实现两个 add 重载函数，分别用于整型值的加法和浮点型的加法，代码如下：

```
//code/chapter03/example3_6.kt
fun main() {
    println("1 +1 =" +add(1, 1))
    println("10 +6 =" +add(10.0f, 6.0f))
}

fun add(a : Int, b : Int) : Int {
    val sum =a +b
    return sum
}

fun add(a : Float, b : Float) : Float {
    val sum =a +b
    return sum
}
```

编译并运行程序，输出的结果如下：

```
1 +1 =2
10 +6 =16.0
```

许多 Kotlin 内置函数设计了函数重载。例如，用于输出的 println 函数就是重载函数，其输出参数的数据类型可以为任意基本数据类型，也可以是字符串。值得注意的是，println 函数输出信息后会在其末尾添加一个换行符，因此通过 println 函数输出的信息会处在不同的行，然而，print 函数也可以实现类型的输出功能，但是并不会在其末尾添加换行符。

3.1.4 Lambda 表达式

Lambda 表达式实际上可以理解为匿名函数。Lambda 表达式使用花括号{}包围，其中包含了参数列表、双线箭头＝＞和表达式体 3 个主要部分，其基本结构如下：

```
{ 参数列表->表达式体 }
```

例如，用于两个整型值加法运算的 Lambda 表达式的代码如下：

```
{ v1 : Int, v2 : Int ->
    println("计算 v1 和 v2 的和!")
v1 +v2
}
```

与普通函数相比，Lambda 表达式没有名称，只是一个独立的代码块。此时，可以将该 Lambda 表达式理解为函数类型的值，可以赋值给任意的函数类型。

函数类型的由参数类型列表、单线箭头符号->和返回类型组成，其基本形式如下：

```
(参数类型列表) ->返回类型
```

上述 Lambda 表达式的函数类型为(Int, Int) -＞ Int。例如，定义(Int, Int) -＞ Int 类型的函数 func，然后将上述 Lambda 表达式赋值给 func，代码如下：

```
//code/chapter03/example3_7.kt
fun main(){
    val func : (Int, Int) ->Int ={ v1 : Int, v2 : Int ->
        println("计算${v1}和${v2}的和!")
        v1 +v2
    }
    val res =func(2, 3)
    println("结果为 : " +res)
}
```

定义 func 函数类型的常量后，即可通过 func 函数名对该 Lambda 表达式进行调用了。编译并运行程序，输出的结果如下：

```
计算 2 和 3 的和!
结果为 : 5
```

如此一来，函数就能够和其他的数据类型具有相同的地位了，这就是“函数是一等公民”的思想。

3.1.5　协程

在 Android 应用程序中，默认存在主线程。由于主线程负责管理 UI 界面刷新，因此异步代码不建议也不能在主线程中实现。此时，开发者使用线程方式管理异步代码，但是本节介绍另一种轻量化的异步管理方式：协程。

1. 协程

协程是 Kotlin 非常具有特色的语言特性。传统的异步编程通常使用多线程编程方法，线程由操作系统调度，而协程则由编程语言进行调度，更加简洁和高效。在日常使用中，可以将协程理解为更加轻量级的线程。通过协程可以实现更加流畅的应用。

注意　协程特性并不在 Kotlin 的标准库中，需要使用 Coroutine 库才能使用。读者可以在包含协程特性的 Android 工程中练习本节代码。

通过 Global.launch 函数可以创建一个线程，代码如下：

```
//code/chapter03/example3_8.kt
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch {
        println("协程作用域")
    }
    Thread.sleep(1000)          //线程睡眠 1s
}
```

在 launch 函数后的 Lambda 表达式中是独立的协程作用域。这部分代码和 Lamdba 外部的其他代码是异步的。

之所以在主函数中睡眠 1 秒线程，是为了保证在程序结束前执行完协程中的业务逻辑。如果需要确保执行完协程中的代码后再结束，则可以使用 runBlocking 函数，代码如下：

```
runBlocking {
    println("协程作用域")
}
```

此时，无论如何协程中的代码一定会执行完毕。通过 runBlocking 可以创建多个子线程(Global.launch 函数只能创建顶层协程)，代码如下：