彻底搞懂Kotlin的高阶函数、匿名函数、Lambda表达式


HurryYu_YZH 关注

0.651 2020.08.04 18:42:18 字数 6,046 阅读 1,353

在Kotlin还没成为Android开发首选语言之前，我们一直是使用Java语言来开发Android应用。不过对于Lambda来说，大家应该并不陌生，因为Android Studio也可以支持JAVA 8中的Lambda表达式，只需要在app的 `build.gradle` 中添加以下代码：

```
android {
    // ...
    compileOptions {
        targetCompatibility JavaVersion.VERSION_1_8
        sourceCompatibility JavaVersion.VERSION_1_8
    }
}
```

这个时候，如果你之前给一个按钮加点击事件的代码是这样写的：

```
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

Android Studio会给你一个提示：

image-20200720174724650.png

这个提示是告诉我们，可以将传入的这个匿名内部类替换成Lambda表达式，通常遇到这种情况，我们可以使用快捷键 `alt + Enter` 让IDE快速帮我们完成转换，转换完的代码如下：

```
btn.setOnClickListener(v -> {
});
```

真香，5行变2行了。然后呢？我们就编写点击事件里面的代码了，至于这个Lambda，我们只知道有这么个东西，还是依靠Android Studio帮我们完成的转换，你要让我直接在 `setOnClickListener` 里面直接手写传入Lambda？不好意思，并不习惯。

这就导致了：虽然之前我们可能就在Java中使用过Lambda，但并不知道Lambda到底是什么，只知道这样可以消除IDE的提示并让代码更加精简。

而在学习Kotlin的时候，Lambda成为了一个很重要的知识点，而且又出现了高阶函数、匿名函数、函数类型、函数引用等等这些我们可能在Java中听都没听过的词，学习起来就更加吃力了，甚至因此而放弃Kotlin。

本文的初衷就是希望把Kotlin中这部分内容讲解清楚。

一、函数类型

1.1、初探函数类型

函数类型对于Java开发者来说是个新概念，因为在Java中并不存在函数这个类型。我们都知道一个变量是需要有一个类型的，在Java中，`int a = 1;`表示的是将1赋值给类型为int，名字叫a的变量，它是java八大基本类型之一，除了基本类型外，剩下的就是引用类型了，比如：`Person person = new Person();`，我们可以说person的类型是Person，它是指向Person对象的引用。

提到字符串，我们都知道它的类型是String；看到一个方法接收一个Person类型的形参，我们都知道要传入一个Person实例。而Kotlin中新增的函数类型并不能用一个泛指的类型代替，比如我们不能把它的类型规定为Function，虽然Function这个单词"很函数"。

这是因为描述一个函数必须要确定它【接收哪些类型的参数】以及【返回值类型是什么】，因此在描述一个函数类型的时候，我们也必须要明确指出这个函数的参数列表以及它的返回值类型。

接收一个int类型的参数并返回String的函数 与 不接收任何参数，返回值类型是int的函数，它们都可以是函数类型，但它们是两个不同的函数类型。好比String和Person都是类类型，但是它们却是两个不同的类。

现在我们来总结一下：**函数类型必须要具有函数的参数和返回值类型，它没有一个固定&通用的表示方法，因为每个函数的参数和返回值类型都可能是不同的，故函数类型更像是【一些】类型而不是【一个】类型。**

1.2、函数类型的书写格式

现在我们已经知道了函数类型必须要具有函数的参数和返回值类型，那这些信息该怎么表示呢？在Kotlin中，你让我定义一个Int类型的变量多简单啊：

```
val a: Int = 0
```

但是你让我定义一个函数类型的变量？前半部分还会写`val f: ??? = ???`后面就直接懵了。就算我们再说得具体一点，接收一个Int类型参数返回String类型的函数类型如何表示？还是不会，这是因为我们现在还不知道书写的格式，下面我们就一起来看看函数类型的书写格式：

> 所有函数类型都有一个圆括号括起来的参数类型列表，以及一个返回类型：`(A, B) -> C`，参数列表与返回值类型之间通过 -> 符号连接。`(A, B) -> C`表示接收类型分别为`A`与`B`两个参数并返回一个`C`类型值的函数类型。

知道了这个格式后，我们很快就可以把上面的例子代码补全：

```
val f: (Int) -> String
```

`(Int) -> String`这就是一个函数类型。另外需要注意的是，即使函数类型的参数列表为空，也必须保留小括号，例如`() -> String`；即使函数类型的返回值类型为Unit，也不可省略Unit不写，例如：`(Int) -> Unit`。

1.3、函数类型变量的赋值

函数类型的变量接收的不就是个函数吗？我们只要将符合这个函数类型的函数赋值给它不就行了？比如如下代码：

```
val block: (Int, Int) -> Int = sum

fun sum(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber + secondNumber
}
```

首先我们定义了一个函数类型的变量block，它的类型为：`(Int,Int) -> Int`，不难看出是一个接收两个Int类型参数并返回Int类型值的函数类型。

显然下面的sum函数是符合这个函数类型的，因此我们认为将函数sum赋值给函数类型的变量block是理所应当的，即 `val block: (Int,Int) -> Int = sum` 是没有任何问题的。然而这样做却是错误的。

这不禁让人怀疑：难道不能将函数赋值给函数类型的变量？那函数类型设计来是干什么的？

我们都知道，除了基本类型以外，我们只能将对象赋值给变量，比如：`Person p = new Person()`，等号右边创建了一个Person对象，而等号左边的p则是指向这个对象的引用，在平时我们也习惯性地称它是Person类型的对象。

那函数类型呢？同样需要将函数类型的对象赋值给函数类型的变量啊，**而函数是对象吗？不是，函数是函数类型的对象吗？更不是。那函数是什么？函数就是我们认知的那个函数，用于存放一段代码**。因此我们怎么可能将函数赋值给函数类型的变量呢？

那...怎么办？

1.3.1、使用双冒号(::)创建函数引用的本质

通过上面的学习我们知道，函数类型的变量需要接收函数类型的对象，那怎么才能得到函数类型的对象呢？Kotlin为我们提供了双冒号(::)来创建函数类型对象的引用。

`val block: (Int,Int) -> Int = ::sum` 只需这样写，就没有问题了，但是请必须确保sum函数的参数列表类型以及返回值类型与函数类型 `(Int,Int) -> Int` 完全保持一致。

那 `sum` 与 `::sum` 到底有什么区别呢？如果在代码中单纯的写 `sum` 将没有任何意义，因为它只是一个函数的名字，除非你在后面加上括号表示调用。而 `::sum` 会基于这个函数创建一个函数类型的对象，这个对象和原来的函数没有直接的联系，只是完全拥有了这个函数的功能而已。

只有类才能创建对象，那么函数类型的对象对应的类是什么呢？我们新建一个文件：`FunctionTest.kt`，并键入如下代码：

```
val block: (Int, Int) -> Int = ::sum

fun sum(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber + secondNumber
}
```

编译后，我们得到了 `FunctionTestKt.class` 和 `FunctionTestKt$block$1.class` 两个字节码文件。我们先来反编译 `FunctionTestKt.class`：

```
public final class FunctionTestKt
{
    @NotNull
    private static final Function2<Integer, Integer, Integer> block;

    @NotNull
    public static final Function2<Integer, Integer, Integer> getBlock() {
        return FunctionTestKt.block;
    }

    public static final int sum(final int firstNumber, final int secondNumber) {
        return firstNumber + secondNumber;
    }

    static {
        block = (Function2)FunctionTestKt$block.FunctionTestKt$block$1.INSTANCE;
    }
}
```

看到这里已经十分清楚了，我们在Kotlin中定义的那个函数类型的变量block被转换成了 `Function2<Integer, Integer, Integer> block`，而这个Function2是一个泛型接口，代码如下：

```
public interface Function2<in P1, in P2, out R> : kotlin.Function<R> {
    public abstract operator fun invoke(p1: P1, p2: P2): R
}
```

其中这个数字2表示的是此接口的泛型支持两个类型的入参和一个返回值类型，刚好和我们定义的函数类型一致。既然block被转换成了接口类型的变量，那能赋值给它的，当然是实现了此接口的类的实例，我们来看看静态代码块种的代码：

```
static {
    block = (Function2)FunctionTestKt$block.FunctionTestKt$block$1.INSTANCE;
}
```

`FunctionTestKt$block$1` 是Kotlin自动帮我们生成的类，我们反编译看看代码：

```
static final class FunctionTestKt$block$1 extends FunctionReference implements Function2<Ir
    public static final FunctionTestKt$block$1 INSTANCE;

    public final int invoke(final int p1, final int p2) {
        return FunctionTestKt.sum(p1, p2);
    }

    public final KDeclarationContainer getOwner() {
        return Reflection.getOrCreateKotlinPackage((Class)FunctionTestKt.class, "lambda");
    }

    public final String getName() {
        return "sum";
    }

    public final String getSignature() {
        return "sum(II)I";
    }

    static {
        FunctionTestKt$block$1.INSTANCE = new FunctionTestKt$block$1();
    }
}
```

这个类继承自 `FunctionReference` 并实现了 `Function2<Integer, Integer, Integer>`，自然也必须实现接口中的invoke方法，由于 `Function2` 支持两个泛型入参，因此invoke方法也有两个Int类型的参数并返回一个Int类型的值，而它具体的实现就是直接调用了 `sum` 函数，为什么是调用 `sum` 函数?

```
val block: (Int, Int) -> Int = ::sum
```

因为我们是基于sum这个函数利用 `::` 创建的函数类型的对象啊，而这个对象实际上就是继承自 `FunctionReference` 且实现了 `Function2` 接口的类的实例，只不过这一切都是Kotlin帮我们完成的。

如果我们的函数类型是这样的：`(String, Int, String) -> Int`，Kotlin也有内置的接口来转换，它的名字叫 `Function3`，这个3表示支持3个入参。那一共可以支持多少呢？答案是22个入参：

```
/** A function that takes 22 arguments. */
public interface Function22<in P1, in P2, in P3, in P4, in P5, in P6, in P7, in P8, in
    /** Invokes the function with the specified arguments. */
    public operator fun invoke(p1: P1, p2: P2, p3: P3, p4: P4, p5: P5, p6: P6, p7: P7,
}
```

也就是说函数类型的入参不能多于22个。

1.4、函数类型的对象怎么使用

```
val block: (Int, Int) -> Int = ::sum

fun sum(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber + secondNumber
}
```

现在我们知道了，这个block实际上是一个函数类型的对象，那怎么使用呢？其实它和函数的使用方式是一样的，我们怎么使用函数就可以怎么使用它。

使用函数？不就是调用吗？ `sum(1, 2)` 这我可以理解，毕竟sum是一个函数，但是 `block(1, 2)` 这样也行？block可是一个函数类型的对象啊，函数类型的对象后面可以加括号调用？

没错，只有函数类型的对象可以，其实这是Kotlin为我们提供的一个语法糖，它本质上还是会去调用invoke函数，也就是说：

```
val block: (Int, Int) -> Int = ::sum

// 以下调用都是有返回值的,因为(Int, Int) -> Int这个函数类型是有返回值的
block(1, 2)
// 等价于
block.invoke(1, 2)
// 等价于
(::sum)(1, 2)
// 等价于
(::sum).invoke(1, 2)
```

Kotlin使用这种（函数类型对象能够使用括号访问）语法让我们感觉，嗯，block就是sum函数的替身，而实际上它是一个对象，一个函数类型的对象。

二、高阶函数

2.1、什么是高阶函数

高阶函数本身没有什么可说的地方，我们先来看看到底什么是高阶函数？

> 一个函数的参数列表中存在函数类型的参数或是函数的返回值类型为函数类型，那么这个函数就叫做高阶函数

如果觉得上面的文字不好理解，我们直接上代码：

```
private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
    return emptyList()
}
```

这就是一个高阶函数，为什么？因为它的参数列表中存在一个函数类型的参数predicate。别忘了，即使函数的参数中没用函数类型的参数，但它的返回值类型是函数类型，这个函数同样是一个高阶函数。这有什么用？大家应该都是写Android的吧，在Android中如果你想给一个按钮加上点击事件，那么肯定需要这样写：

```
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

这个OnClickListener是一个接口，里面包含了一个方法：

```
public interface OnClickListener {
    void onClick(View v);
}
```

有没有想过为什么需要传一个匿名类过去？因为Framework会在该View被点击的时候调用我们在匿名类中实现的那个onClick方法，以便于执行我们自己的处理逻辑。

要是我们能直接将方法传过去就好了...梦该醒了，这是Java，只能通过接口 + 匿名内部类这种折中的方案来实现这样的需求。

而Kotlin有了函数类型，又有了高阶函数，意味着我们可以传入一段代码给函数了，函数可以在适当的时候执行我们传入的那段代码，完美的解决了这一痛点。

2.2、高阶函数的使用案例

前面我们编写过一个高阶函数：

```
private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
    return emptyList()
}
```

目前这个函数里面什么逻辑也没写，从函数的名字推断，这个函数应该要实现的功能是过滤苹果集合，筛选出符合要求的AppleBean并返回。

我们先来看看AppleBean的代码：

```
data class AppleBean(val color:Int, val weight:Int)
```

它包含了苹果的颜色和苹果的重量两个属性。现在有如下三个需求：

- 找出所有重量大于6的苹果
- 找出颜色为0xFF0000的苹果
- 找出颜色为0xFF0000且重量大于6的苹果

虽然都是过滤找出符合要求的苹果，但是条件是不一样的。这个怎么做呢？最简单的方法就是写三个循环，每个循环中针对不同的条件对苹果集合进行筛选。但是我们完全可以使用高阶函数和函数类型优雅的实现这个功能：

```
private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
    val destination = mutableListOf<AppleBean>()
    for (appleBean in appleList) {
        if (predicate(appleBean)){
            destination.add(appleBean)
        }
    }
    return destination
}
```

可以看到，filterApple函数主要的功能只是遍历集合，以及把符合条件的苹果添加到新的集合中返回，至于筛选的条件，filterApple函数并不知道，而是完全取决于predicate这个函数类型的参数，只要predicate返回true，则符合条件。这样我们可以在调用函数时动态地传入不同的过滤规则，代码如下：

```
fun main() {
    filterApple(appleList, ::filterColorPredicate)
    filterApple(appleList, ::filterWeightPredicate)
    filterApple(appleList, ::filterColorAndWeightPredicate)
}

private fun filterColorPredicate(appleBean: AppleBean): Boolean = appleBean.color == 0

private fun filterWeightPredicate(appleBean: AppleBean): Boolean = appleBean.weight >
```

```
9
10 private fun filterColorAndWeightPredicate(appleBean: AppleBean): Boolean = appleBean.c
11
12 private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
13     val destination = mutableListOf<AppleBean>()
14     for (appleBean in appleList) {
15         if (predicate(appleBean)) {
16             destination.add(appleBean)
17         }
18     }
19     return destination
20 }
21
```

首先我们定义了三个函数，这三个函数的参数类型以及返回值类型都与 `predicate: (AppleBean) -> Boolean)` 这个函数类型一致，这三个函数分别实现了三种不同的过滤规则。

在 `main` 函数中，我们调用 `filterApple` 这个函数时，第二个参数会传入一个函数类型的参数，而通过前面的学习我们知道，需要传入的必须是这个函数类型的对象，因此我们使用了双冒号创建了函数类型的对象。

这...太麻烦了吧，我每次调用高阶函数，传入函数类型的时候，还必须要先定义一个函数，然后再使用 `::函数名` 传入吗？其实不必这么麻烦的，现在我们学习的只不过是其中的一种方式而已。下面我们就要来学习更加简单的方式：匿名函数。

三、匿名函数

3.1、什么是匿名函数

匿名函数顾名思义就是没有名字的函数，那这种没有名字的函数我们怎么调用呢？答案是无法直接调用。匿名函数可以赋值给一个变量，或者当作实参直接传递给一个函数类型的形参。

上面的话是什么意思呢？我们用代码来解释。首先匿名函数是没用名字的，因此我们可以这样来定义一个匿名函数：

```
1 fun(appleBean: AppleBean): Boolean = appleBean.weight > 6
```

这就是一个匿名函数，可见它和普通函数并没有太大区别，唯一不同的就是没有了函数名。没有名字怎么调用呢？前面说了，不能直接调用，只能将它赋值给一个变量，或者当作实参直接传递给一个函数类型的形参。

3.2、匿名函数赋值给变量的本质

匿名函数能够赋值给变量：

```
1 val filterWeightFunPredicate =
2             fun(appleBean: AppleBean): Boolean = appleBean.weight
```

一旦我们将一个匿名函数赋值给一个变量，那个这个变量其实就是一个函数类型的对象。根据这个匿名函数的参数类型和返回值类型可知，它对应的函数类型为： `(AppleBean) -> Boolean` ，又由于filterWeightPredicateFun已经是此函数类型的对象，或者说此函数类型对象的引用，因此我们可以将filterWeightPredicateFun传入filterApple这个高阶函数的第二个参数：

```
1 val filterWeightPredicateFun =
2                         fun(appleBean: AppleBean): Boolean = appleBean.weight > 6
3 filterApple(appleList, filterWeightPredicateFun)
```

当然，我们也可以直接将匿名函数传入，相当于将匿名函数直接赋值给函数类型的参数：

```
1 filterApple(appleList, fun(appleBean: AppleBean): Boolean = appleBean.weight > 6)
```

看起来很爽，好像真的传入了一个函数一样。而实际上你传递的是一个对象，一个函数类型的对象。因为**匿名函数它本来就不是函数，而是一个函数类型的对象**。

你以为这样写就是最简洁的了？不！你忘记Lambda了？

四、Lambda表达式

终于学到了最后一个知识点——Lambda。其实Lambda表达式才是与高阶函数的绝配，平时我们给高阶函数中的函数类型参数传递值时，一般都会选择传入Lambda表达式，因为它足够简洁与强大。

其实Lambda表达式的本质是匿名函数，而匿名函数的本质是函数类型的对象。因此，**Lambda表达式、匿名函数、双冒号+函数名这三个东西，都是函数类型的对象，他们都能够赋值给变量以及当作函数的参数传递！**

4.1、Lambda表达式的格式

现在我们先来学习Lambda表达式的格式：

- Lambda表达式被大括号包围着
- Lambda表达式的参数在 `->` 的左边，如果没有参数，则只保留函数体
- Lambda表达式的函数体在 `->` 的后面
- Lambda表达式的返回类型值总为函数体最后一行代码的返回值类型

下面我们通过代码来看看Lambda表达式到底是什么样子。

4.1.1、无参数，无返回值的Lambda表达式

```
val test01Lambda = {
    print("无参数,无返回值")
}
```

我们同样将Lambda表达式赋值给了变量，这个变量是函数类型（ `() -> Unit` ）的对象。

4.1.2、有参数，无返回值的Lambda表达式

```
val test02Lambda = { name: String ->
    print("有参数,无返回值, 参数值为: $name")
}
```

这个变量是函数类型（ `(String) -> Unit` ）的对象。

如果我们手动给这个变量指明了类型，那么Lambda的参数类型还可以不写：

```
val test02Lambda: (String) -> Unit = { name ->
    print("有参数,无返回值, 参数值为: $name")
}
```

这个时候，Kotlin可以自动为我们推断出Lambda中这个参数的类型是String类型。

如果Lambda表达式的参数只有一个，我们甚至连这个参数都可以省略不写，那...我想使用这个参数的时候怎么办呢？我们可以用 `it` 来代替：

```
val test02Lambda: (String) -> Unit = {
    print("有参数,无返回值, 参数值为: $it")
}
```

嗯，真香。

4.1.3、有参数，有返回值的Lambda表达式

```
val test03Lambda = { doubleValue: Double ->
    print("parameter is Double,Value is:$doubleValue")
    print("now parse Double into String")
    doubleValue.toString()
}
```

Lambda表达式的最后一行代码将作为返回值，因此它对应的函数类型为： `(Double) -> String` 。同样，如果变量已经确切的指定了类型，则Lambda表达式的参数类型可以省略，又由于只有一个参数，所以连参数都可以省略，Kotlin将使用it代替这个参数名。

4.2、Lambda表达式与高阶函数

现在大家已经对Lambda表达式的写法了如指掌了，现在是时候来看看Lambda表达式如何与高阶函数配合使用了。还记得之前的那个筛选苹果的例子吗？我们定义了一个高阶函数，它的名字叫：filterApple，代码如下：

```
private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
    val destination = mutableListOf<AppleBean>()
    for (appleBean in appleList) {
        if (predicate(appleBean)) {
            destination.add(appleBean)
        }
    }
    return destination
}
```

这个高阶函数的第二个参数接收了一个函数类型的参数，在之前，我们分别使用了两种方式进行调用：

1. 双冒号的方式：

```
fun main() {
    filterApple(appleList, ::filterColorPredicate)
}

private fun filterColorPredicate(appleBean: AppleBean): Boolean = appleBean.color ==
```

显然这种方式太麻烦了，必须要定义一个与函数类型相匹配的函数。

2. 匿名函数的方式：

```
filterApple(appleList, fun(appleBean: AppleBean): Boolean = appleBean.weight > 6)
```

不错，挺简洁。

下面，我们将通过传入Lambda表达式的方式对这个高阶函数进行调用，你一定会爱上Lambda表达式的！

4.2.1、完整的写法

先来看看最完整的写法：

```
filterApple(appleList, { appleBean: AppleBean -> appleBean.weight > 6 })
```

唉，也没比匿名函数好到那里去啊！别急我们接着往下看。

4.2.2、简化的写法

我们怎么知道哪个参数需要传入Lambda表达式呢？当然是看被调用的这个高阶函数的定义啊，还记的这个高阶函数的样子吗？

```
private fun filterApple(appleList: List<AppleBean>, predicate: (AppleBean) -> Boolean)
    // ...
}
```

显然第二个形参是一个函数类型的参数，并且已经明确的指出了形参的函数类型：`(AppleBean) -> Boolean`（实际上也必须明确指出，否则报错）。根据我们前面所学习的知识，这种情况下，我们可以直接省略Lambda表达式中的参数类型，Kotlin会根据上下文自动推断：

```
filterApple(appleList, { appleBean -> appleBean.weight > 6 })
```

嗯！香。

4.2.3、再简化的写法

由于这个函数类型只需要一个参数，因此我们还可以省略参数的名字，Kotlin会使用it代替，这在之前的Lambda讲解中，都是讲过的：

```
filterApple(appleList, { it.weight > 6 })
```

嗯！真香。

4.2.4、再再简化的写法

Lambda表达式与高阶函数配合使用，还有两个特别爽的地方。

第一个是：如果Lambda表达式作为函数的最后一个参数传入，那么它可以单独放在调用函数的括号后面：

```
filterApple(appleList) { it.weight > 6 }
```

第二个是：如果函数只接收一个函数类型的参数，我们传入Lambda表达式时，连函数调用的括号都可以去掉：

```
fun main() {
    test { num1, num2 ->
        // ...
    }
}

private fun test(block: (Int, Int) -> Unit) {
    // ...
}
```

嗯！真TM香！（嘶哑）

五、总结

本文介绍了函数类型、高阶函数、匿名函数以及Lambda表达式的本质，通过学习我们知道了：

- 函数不能直接传递，传递的实际上是函数类型的变量
- 双冒号、匿名函数、Lambda表达式的本质实际上都是函数类型的对象
- 高阶函数没什么神奇的，只不过是参数列表或返回值类型存在函数类型的函数
- Lambda很方便，日常使用最多的就是Lambda与高阶函数配合使用
- Lambda的各种简便写法

如果你认真阅读了此文，并从中获取到了知识，对你有帮助，那我编写此文的目的之一也就达到了。再次感谢你的阅读。

[参考文章]

1. Kotlin 的 Lambda 表达式，大多数人学得连皮毛都不算
2. Kotlin官方文档