目录

Kotlin 相关分享

—张鹿鹿

1 Kotlin 基础

1.1 基础语法

1.1.1 Kotlin 文件

创建一个以 .kt 结尾的文件，注意这是 Kotlin 的文件格式：

1.1.2 var 和 val

其中用 `var` 表示一个变量， `val` 表示一个不可变的变量，注意是不可变的变量而不是一个常量（后面会有详细介绍）
我们注意到在 Kotlin 中，变量名是写在前面的，变量的类型是写在后面的，中间用`:`分割，特别的，如果你的类型是编译器可以推断出来的就不用写类型。比如：

1.1.3 空安全

需要强调一点，Kotlin 是具有空安全类型的，上面代码中的声明的 Int 和 String 都是不可为空的，如果强制设置 null，编译器将会报错：

如果有一个可能为空的变量可以在后面添加 ?，这里则可以使用 String? 的类型，需要特别注意的是，String 和 String? 是两种不同的类型，所以不可随意互相赋值：

如果非要要赋值的话，可以使用 `!!`，表示该变量不可能为空，一旦为空则报 `KotlinNullPointerException`

如上面代码所述，反之赋值是可以的，因为 String? 类型包含了 String 类型（换句话说，String? 可为空也可不为空）

1.1.4 函数

从上面的 main 函数可以看出来，kotlin 中以 `fun` 关键字表示我要声明的是一个函数，函数的参数与变量声明类似，`变量名 : 变量类型`，如果需要返回值，则直接在后面添加 `: 返回值类型` ，例如：

1.1.5 字符串的模板语法

在 Kotlin 中可以使用 `$变量`，直接拼接字符串，例如：

1.2 与 Java 互调

1.2.1 Kotlin 文件中的函数

首先让大家感到非常不适应的一点是，Kotlin 的函数居然可以直接写在文件中，而不用写在类中，但如果你知道 Kotlin 的代码编译以后依旧是 JVM 平台的一个 class，这一点应该就很好理解了。

例如，我们在 Utils.kt 中写如下函数：

在 Java 的类中可以做如下的调用：

从调用可以看出，Kotlin 文件中的函数编译后直接转换为对应文件名 + Kt 类中以 public static 开头的方法。

1.2.2 object 关键字

还有一种 Kotlin 中特殊的语法，object + 类声明，这种写法是在 Kotlin 中创建一个内部所有方法和变量都是类似静态的类的写法：

如果在 Kotlin 代码中则可以直接调用：

如果在 Java 代码调用，需要使用以下方法：

> 简单分析下，其实构造方法变成了私有方法，暴露了一个静态的当前实例 INSTANCE ，通过它我们可以访问到其中的方法。
>
> 通过 Java 代码调用我们能够看出， object 关键字的类其实是 Kotlin 单例的一种写法。
>
> 特别的: object 还会修饰为伴生对象，后面将会说到。

1.2.3 Class 和 KClass

使用 Java 时我们有的时候传递某个类的 class，但是在 Kotlin 中调用 Java 类的 class 需要在后面追加 .java，例如我们有一个 JavaMain.java 的 Java 类，我们使用它的 class 时则需要这样调用：JavaMain::class.java。

为什么会有这样的特殊语法呢？这是因为 Kotlin 的 class 和 Java 的 class 格式是不一致的。
在 Kotlin 中，所有的类并不是编译成 Class，而是编译成 KClass，例如：

1.2.4 与 Java 在关键字上的冲突

例如在 Java 类中有一个变量 in ：

而 in 在 Kotlin 中是一个关键字，如果在 Kotlin 中调用就会报错

正确调用方法应该是将该变量使用两个反引号包裹(`)

1.2.5 Kotlin 没有封装类

首先看看以下代码。

我们用 Java 创建一个接口，如下：

之后我们创建一个 A.java 实现这个接口，并在每个方法中打印对应的类型，int 和 Integer。

最后，我们在 Kotlin 代码中调用它，我们来看下它的执行结果：

你会发现只调用了 int 参数的方法，通过这个示例可以看出来 Kotlin 是没有封装类的概念的，为了进一步说明，我们让 Kotlin 的类来实现这个接口：

你会发现编译器会报方法名重复的错误，删掉后正常

> 上述例子中如果非要调用 Integer 参数的方法，当然也是能做到的，可以通过反射的方法。

1.2.6 Kotlin 空值敏感

还是通过一个示例来说明，写一个包含如下静态方法的 Java 类：

之后在 Kotlin 代码中调用这个方法，其中赋值方式有三种，分别是让编译器推断、赋值给不为空的 String 类型以及赋值给可为空的 String? 类型。

首先看第一种编译器推断类型的方式，当我们使用编译器查看类型时你会发现它返回的是 String! 的类型，这种类型我们不能直接声明，这是 Kotlin 与 Java 互调时的一种特有的类型，也可以说成是 Java 的 String 类型。

之后我们执行 main 函数，看到代码 10 行位置报错，说明为空的值给不可为空的 String 类型赋值会报错！

当我们注释掉 fmt2 代码继续执行，你会发现编译器会正常编译通过，难道这就说明第 1、3 种方式是可靠的呢？我们尝试调用 fmt1 的某个方法：

发现会报 NullPointerException 错误。为什么会这样呢？ 因为 String! 类型 Kotlin 编译器会认为这是兼容类型，你只能是临时地使用，但是如果你调用它的话，就会像 Java 一样去执行，所以就报了 Java 的空指针异常。

反观 fmt3 就不会报错，这就是 Kotlin 的空安全！

> 【注】当我们调用 Java 的方法返回类对象的时候，如果你不确定返回值是否可能为空，你一定要赋值给一个可空类型，这样才能利用 Kotlin 的空安全！

1.2.7 Kotlin 没有静态变量与静态方法

由于 Kotlin 中没有静态变量和静态方法，我们可以通过上一节中提到的 object 关键字声明的类来间接实现类似 Java 中的静态方法。

如果让其变成真正的静态方法，可以通过使用 @JvmStatic 注解使其变成 Java 的静态方法，这时就可以直接在 Java 类中调用。

1.3 构造函数

1.3.1 类的声明

在前面的文章中简单的介绍过类的声明，我们来回顾下。

与 Java 类似，同样使用 `class` 关键字来声明一个类，后面紧跟着类名，如果该类有父类的话使用 `:` 分割，如果该类需要实现接口，则直接使用 `,` 跟在后面即可，而不是用 `implement` 关键字。接口和父类没有先后顺序。

特别的，Kotlin 中如果没有显式的声明一个父类，那么它的父类为 **Any**，而不是 **Object**。

Kotlin 会对所有的类默认添加 `public final` 修饰符，如果不需要 `final` 来修饰的话，则需要使用 `open` 关键字修饰 Kotlin 的类， `open` 就表示这个类“不 final”。

1.3.2 构造函数

在 Kotlin 中分为主构造函数和次级构造函数，其中主构造函数只能有一个，次构造函数可以有多个。

如果构造函数只有一个且没有参数，则可以直接省略不写，如上面的示例中 MainActivity 省略了无参构造函数，父类后面的括号表示调用父类无参数的构造函数。

如果构造函数需要参数，则可以在类名后添加参数，如果需要在构造函数中执行语句的话，则需要添加 init 代码块，将语句写在代码块中，像这种直接跟在类名后面的构造函数就称之为**主构造函数**。（init 代码块是可以写多个的）。

如果需要多个构造函数，则可以使用 `constructor` 关键字声明**次级构造函数**，特别地，如果声明了次级构造函数，主构造函数的无参隐藏的特性就会失效。例如示例中的类，就没有无参构造函数。

如果主构造函数和次级构造函数都存在时，次级构造函数要使用 `this` 关键字调用主构造函数：

1.3.3 执行顺序

通过示例我们就可以看出，init 代码块执行顺序受声明的顺序影响，且优先于次级构造函数。

1.4 访问修饰符

在 Kotlin 中有四种访问符：

- public

- protected
- private
- internal

前面三种有 Java 基础的读者肯定都比较了解。public 具有最大的访问权限，可以访问任意路径的类、接口、成员变量；protected 表示子类可以访问它修饰的父类成员变量；private 表示访问权限仅限于类的内部；而 internal 是 Kotlin 特有的访问修饰符，表示一个**模块**内都能访问到这个对象。

模块是什么概念呢？我们在 Android Studio 中看下：

通过 File -> New -> New Module 创建出来的就是一个模块。

> internal 修饰符更多用于项目中的结构化扩展以及模块化等场景。

1.5 比较对象

1.5.1 Java 比较对象

我们知道 Java 在比较对象的时候，是通过 == 来判断两个对象是否完全相同，通过 equals 方法判断两个对象的值是否相同：

```
public static void main(String[] args) {
    String string = "string";
    String newString = new String("string");
    System.out.println("string == newString :" + (string == newString));
    System.out.println("string.equals(newString) :" + string.equals(newString));
}
```

执行结果：

```
string == newString :false
string.equals(newString) :true
```

1.5.2 Kotlin 比较对象

在 Kotlin 中是不需要通过 equals 方法判断两个对象的值是否相同，而是通过 ==，如果判断对象是否相同则需要使用 ===，Kotlin 与 Java 在比较对象上的区别如图：

Kotlin 与 Java 比较对象

> 其实大家只要记住这张图就会避免出现比较对象的错误。

接下来写一个 Kotlin 比较对象的示例：

需要注意的是，Kotlin 中除了直接赋值字符串以外没办法直接使用字符串去构造一个 String 对象，需要通过 StringBuilder 或者传入一个 ByteArray 对象来间接的构造对象:

```
fun main() {
    val string = "string"
    val newString = String("string".toByteArray())
    println("string == newString: ${string == newString}")
    println("string === newString: ${string === newString}")
}
```

执行结果：

```
string == newString: true
string === newString: false
```

1.6 伴生对象

在 Java 中我们经常使用类似示例中的工具类，可以直接通过 类名.方法名 来调用。

但是在 Kotlin 中是没有静态方法的，解决的办法有两种,一种方法就是前面提到过的使用 @JvmStatic 注解去注释它，第二种方法就是使用伴生对象的方式创建：

伴生对象一定要写在一个类的内部，作为这个类的一个伴生对象存在，伴生对象使用 companion object 两个关键字来声明。在 Kotlin 中就可以像 Java 调用静态方法一样，直接使用 类名.方法名 来使用。

而在 Java 中则需要通过静态变量 Companion 来调用，实际上，这个 Companion 是编译器帮我们生成的一个对象，用来访问内部的方法和变量。

1.6.1 单例

前面的文章中介绍过 Kotlin 中一种单例的写法，其实更推荐的写法是使用伴生对象和 object 关键字结合的方式，示例如下：

1.7 数据类

数据类是 Kotlin 中很特殊的一种类，它可以将我们类中的成员变量自动的生成 getter/setter 方法，以及我们经常需要重写的 toString()、hashCode()、equals()、copy() 方法，而不需要像 Java Bean 一样需要我们手动去重写这些方法。

数据类的声明只需要在类的前面添加 data 关键字。

需要注意一点的是，数据类是 final 类型的，不能添加 open 关键字去修饰它！

1.8 枚举

1.8.1 枚举类

Kotlin 中也有枚举类，它的枚举类与 Java 中的使用是一致的：

但是在 Kotlin 中我们很少使用枚举类，而是使用它更加强大的“枚举类”，称之为密闭类。

1.8.2 密闭类

使用 `sealed` 关键字修饰一个类即可，另外密闭类是可以有子类的，但是密闭类的子类必须和密闭类写在同一个文件中，所以通常会把密闭类的子类写在类本身里面。它的用法也与枚举类的用法一致，示例如下：

密闭类最大一个特性在于它是可以有扩展它的子类的，并且它的子类也可以成为密闭类的一个选项，示例如下：

1.9 循环

下面的示例是我们最熟悉的循环语法，但是这种语法在 Kotlin 中是不能使用的。

不过没有关系，在 Kotlin 中有更多适合的语法供我们选择。

以下列出 5 种 Kotlin 中常用的循环语法：

1.9.1 常用的循环语法

第一种，其中声明一个 i 用于迭代，之后 1..10 表示的是从 1 到 10 的闭区间（1 <= x <=10)

当我们把循环 1 到 10 改成 10 到 1 时，发现编译器发出警告，说我们的区间是空的，你的意思是不是使用 `downTo` 关键字。这就说明，关键字 `..` 只能用递增的循环，如果使用递减的循环应使用 `downTo`。

第二种，1 until 10 表示的是从 1 到 10 的半开区间（1 <= x < 10)

我们看下 `until` 的原型声明：

在 Kotlin 库中找到 kotlin.ranges._Ranges.kt 类，会发现其实是给 Int、Long、Byte 等等声明了对应的扩展函数：

从它的执行步骤来看：

第一步检查你传入的值是否是一个合法的值，否则直接返回 EMPTY，如果合法的话其实就是将 to 值减 1 后调用了 `..` 的函数，我们点开 `..` 函数，发现它就是对应的 rangeTo 函数。

第三种，上面已经提到，10 downTo 1 表示遍历从 10 到 1 的闭区间（10 <= x <= 1)

查看 `downTo` 对应扩展函数也能清楚的知道它的含义，从当前的 this 到 to 值按照步长 -1 循环递减。

第四种，带有步长 step 的循环，相当于 Java for 循环中的 count += 2。

同样的，调用的还是上面提到的方法，只不过加入了步长。

第五种，Kotlin 提供了海量的扩展函数，其中有一个扩展函数 repeat 可以用来循环：

实际上，这个函数其实就是封装了一层 for-in 形式的循环，允许你传入一个 Lambda 闭包用来执行对应的代码。

1.9.2 遍历集合

我们知道循环大多数用于集合，针对集合 for-in 形式的循环可直接把 list 中的元素取出：

特别的，Kotlin 还允许我们使用解构的形式来获取对应的 index 和 value：

需要注意的是，如果我们使用的是 map 的话是可以直接使用解构的（后面会提到），但是如果使用的是 list，则需要先调用 list 的 withIndex 方法，它可以为我们返回一个迭代器：

迭代器的泛型参数是 IndexedValue，它实际上是一个声明了 index 和 对应 value 的数据类！这样就解释了 list 使用 withIndex 来生成可解构的迭代器。

2 Kotlin 进阶

2.1 查看 Kotlin 对应的 Java 文件

2.2 函数嵌套

2.2.1 函数语法

2.2.1.1 函数声明

即使 Kotlin 是一门面向对象的编程语言，它依然保留了函数这样的概念，不像 Java 中，仅仅只有方法。

在开始前先回顾下函数的声明：

在 kotlin 中以 `fun` 关键字表示我要声明的是一个函数，函数的参数与变量声明类似，`变量名 ： 变量类型`，如果需要返回值，则直接在后面添加 `： 返回值类型` ，例如：

2.2.1.2 函数参数默认值

另外，Kotlin 中允许函数的参数有默认值的。例如下面代码，控制台则输出 “Default Name”。

2.2.1.3 函数体省略

如果一个函数的函数体只要一个语句的话，我们是可以直接将这个语句赋值给这个函数的：

2.2.2 函数嵌套

与内部类有些类似，内部函数可以访问外部函数的局部变量，例如代码中的 str，内部函数是可以访问的。

通过上面的示例，我们可以简单总结一下它的使用场景：

1. 在某些条件下会触发递归的函数
2. 不希望被外部函数访问到的函数

> 【注】需要注意的是，在一般情况下我们是不推荐使用嵌套函数的，因为这样会大大降低代码的可读性。

2.3 扩展函数

> Kotlin 中有一个非常大的优势，就是可以静态的给一个类扩展它的成员方法，以及成员变量

首先扩展函数也是一个函数，所以也需要 `fun` 关键字进行声明，后面紧跟着的是你需要扩展的类的类名， 比如示例中是给 File 类扩展一个成员方法，之后是一个 `.` 它用来分割类名和函数名，后面紧跟着的是扩展函数名，后面则与普通函数声明无异，参数、返回值以及函数体。下面我们看到的示例实际上是 Kotlin 标准库中 File 的扩展函数。

如何使用呢？很简单，在 Kotlin 中，可以直接在类的对象上面调用这个扩展函数：

特别地，在 Java 代码调用时

首先，我们需要 new 一个 Java 的对象，其次我们需要注意是，这个扩展函数并不是这个类本身的函数，而是我们扩展出来的，它在编译的时候会被编译到那个类所对应的 class 中去。所以在 Java 中我们实际上是调用 FileKt 这个类中的扩展方法，另外还需要一点注意的是，扩展函数的第一个参数是需要扩展的类的对象即示例中的 file 对象，示例如下：

扩展函数需要大家注意的一点就是，它是静态的给一个类添加成员变量和成员方法。

> 扩展函数的主要用途是对一些第三方 SDK，或者说那个类是你不能控制的，你要想给它新增些你需要用到的方法时会用到的函数。

2.3.1 扩展函数的静态解析

为什么是静态的呢？下面给出一个示例。

示例中，open 表示“不 final”，即可以继承，声明一个 open 的 Animal，Dog 类继承自 Animal，给 Animal 和 Dog 添加扩展函数 name，并给 Animal 添加扩展函数 printName 执行 Dog().printName(Dog())：

它的执行结果我们发现是 animal，而不是我们的 dog，为什么是这样的呢？这是因为 Kotlin 的扩展函数实际上是静态的给一个类添加的方法，它是不具备运行时的**多态效应**的。

我们看一下这个类编译之后的样子：

你会发现扩展函数会被编译成 `public static` 修饰的方法，且第一个参数是需要扩展的类的对象。

示例中需要注意的是，printName 扩展函数接受两个参数，第一个是需要扩展的类 Animal 对象另外一个是我们声明的扩展函数的参数。

最后，我们看 main 方法中我们的 Dog 对象被强转成 Animal 传递给 printName，printName 中 调用的 name 方法选择的是带有 Animal 参数的方法，而不是我们想象的带有 Dog 参数的 name 方法。

即扩展函数不具备运行时的多态效应。

2.4 Lambda 闭包

我们先看一个 Java 中非常常见的例子，就是创建一个子线程：

我们知道 Java 8 也是有 Lambda 支持的，它可以把 Runnable 对象省略成一个 `() ->` 的表达形式。

接下来，我们看下 Kotlin 的 Lambda 形式，与 Java 8 Lambda 类似，只不过 `->` 放在了 `{}` 的内部。

同时 Kotlin 的 Lambda 语法还有很多的特性，它允许你省略很多没有用的信息。

比如，**如果你的 Lambda 是没有参数的**，你是可以省略 `->` 符号的。示例中，Runnable 是没有参数的，可以写成如下形式：

接着，**如果 Lambda 是函数的最后一个参数**，可以将大括号放在小括号外面；

如果函数只有一个参数且这个参数是 Lambda，可以省略小括号。

这就是 Kotlin Lambda 中最简单的结构：

2.4.1 Lambda 闭包声明

接下来看下单独声明一个闭包的时候它对应的写法：

首先给闭包声明一个变量名，同时用闭包声明给它赋值，闭包也是可以有参数的，参数声明与变量声明基本相同，`变量名: 变量类型 -> Lambda 闭包体`。

> 通过示例可以看出，闭包调用分为两种形式，一种是直接使用小括号添加参数调用，另外一种方式是调用它的 invoke 方法。

2.4.2 Lambda 闭包原理

首先，写一个带有一个参数的闭包：

接下来我们看下它对应的 Java 源码：

从转换的 Java 代码可以看出来，我们所写的闭包被转换成了 Function1 的类，实际上 Function1 是 Kotlin 标准库中的接口，我们打开 Functions.kt 文件：

翻到最后你会发现总共有 23 个接口，从 Function0 到 Function22，可以看出来接口名后面所跟的数字表示的是当前闭包所能接收的参数的个数，上面的示例中，我们传入了一个 Int 型参数，所以使用的是 Function1 接口。
难道只能接收 22 个参数以下闭包？下面我们再来看个示例：

我们声明了带有 23 个参数的闭包，之后看下 Java 源码：

会发现它使用 FunctionN 接口，使用数组作为参数巧妙地解决了 22 个以上参数的问题。

好，那我看下这个 FunctionN.kt 文件：

需要注意的是，在 Kotlin 1.3 之前确实只支持 22 个参数的闭包，而在 1.3 之后可以支持 N 个参数。

2.5 高阶函数

高阶函数就是指函数或者 Lambda 的参数又是一个函数或者 Lambda。

通过下面示例简单的了解下。首先，高阶函数也是函数，所以同样需要使用 `fun` 关键字来修饰，示例中的高阶函数 onlyIf，需要接收两个参数，第一个参数为 Boolean 的变量，第二个参数是参数为空返回值为 Unit 的函数。

其中 Unit 就是一个没有返回值的函数的类型，同样的示例中的 main 函数也会有个隐藏的 Unit 返回值类型，如果作为函数进行声明通常是可以省略的（ main 函数），但是如果作为参数则必须要显式的声明（ onlyIf 的第二个参数）。

2.5.1 函数是“一等公民”

在 Kotlin 中函数是“一等公民”，可以直接使用对象加两个冒号（`::`）进行分割，后面紧跟着方法名，使用这种方式来引用一个函数声明，而不是像一个对象一样使用 `.` 来执行函数。可以通过示例体会一下：

> 【注】在高阶函数的参数中要想作为参数传递给高阶函数时，必须传的是函数的声明，如果传的是直接执行一个函数的话，那么实际上传递的是函数的返回值！

2.6 用内联优化代码

Kotlin 的 Lambda 表达式会被编译成匿名内部类的形式，如果在代码中有大量重复的 Lambda 表达式的话，会生成很多无用对象。

可以使用 inline 关键字修饰方法，这样的方法在编译时会把方法的调用拆解为语句的调用，进而减少创建不必要的对象。

拿上面提到过的例子，加上 inline 关键字修饰，之后我们查看一下它对应的 Java 源码，会发现函数中的代码直接被拆解开进行执行了。

> 需要注意的是，过度使用 inline 关键字会增加编译器的负担，同时使代码块变得很庞大，查找问题会变得麻烦。因此，inline 关键字通常只会修饰高阶函数而不会滥用它。

2.7 解构

在 Kotlin 中允许直接将一个类拆解之后并分别赋值，就像示例中的 User 对象一样，可以赋值给一个拆解开的 age 和 name 变量。

解构的格式固定，val/var 括号后的第一个值与 User 类中的 component1 方法对应，相应的第二个值与 component2 方法对应，且都需要使用 operator 关键字修饰。当然完全可以再声明一个 component3，与 User 对象的变量个数无关，例如：

打印结果：

> operator 表示将一个函数标记为重载一个操作符或者实现一个约定。

特别的，对于数据类，也就是之前提到的以 `data` 关键字修饰的类，会默认为每个字段生成 componentX 方法。

2.7.1 应用场景

解构更常用的场景是在遍历 Map 时。

例如，声明一个 Map 对象，关键字 `to` 的左侧表示 key，右侧表示对应的 value，在遍历时可以直接使用解构将里面的 key 和 value 取出，非常方便：

2.8 Kotlin 集合操作符

> 如果大家有用过 RxJava 一定对操作符的概念并不陌生，RxJava 允许我们对数据做一系列的链式调用，在每一步中改变数据的格式，最终得到我们想要的数据。Kotlin 中原生的为集合添加了海量的操作符，基本上 RxJava 有的操作符 Kotlin 都会有语言层面的支持。

附上整体代码：

```
fun main() {
    val a = arrayOf("4", "0", "7", "i", "f", "w", "0", "9")
    val index = arrayOf(5, 3, 9, 4, 8, 3, 1, 9, 2, 1, 7)
    index
        .filter { it < a.size }
        .map { a[it] }
        .reduce{s, s1 -> "$s$s1" }
        .also { println("密码是：$it") }
}
```

首先还是声明两个数组，一个是作为参数输出的字符串数组，另外一个是下标数组。

之后就可以直接操作集合，而不是像 RxJava 一样经过 flatMap。

filter 等价于 RxJava 的 filter，过滤下标大于 String 数组的值。

```
  .filter { it < a.size }
```

map 等价于 RxJava 的 map，做一次数据类型变换。取出对应下标的元素。

```
.map { a[it] }
```

reduce 等价于 RxJava 的 reduce，做一次数据合并。这里使用 Kotlin 的模板字符串的方式将两字符串合并。

```
.reduce{s, s1 -> "$s$s1" }
```

最后，使用 also 输出结果，相当于 RxJava 的 subscribe。

```
.also { println("密码是：$it") }
```

我们首先直观上看，Kotlin 的代码量要远远小于 RxJava，这里面一个原因是 Kotlin 不需要 flatmap 操作符去做变换，第二个原因是 Kotlin 原生支持了 Lambda 表达式，所以我们不需要像 Java 一样每一个参数都去显式的 new 一个接口对象。

> 如果我们想有一个自定的集合操作符，该怎么做呢？

2.8.1 自定义集合操作符

其实 Kotlin 的所有操作符都是一个 inline 的扩展函数来实现的。

比如这里我们定一个 convert，它的功能实际上跟 map 的功能是一致的，将我们输入的参数转换成另外一种类型的数据输出，它扩展的是一个迭代器，也就是说我们所有的集合都可以使用这个扩展函数：

```
inline fun <T, E> Iterable<T>.convert(action: (T) -> E): MutableList<E> {
    val list: MutableList<E> = mutableListOf()
    for (item in this) list.add(action(item))
    return list
}

fun main() {
    val list: List<Int> = listOf(1, 2, 3, 4, 5)
    list.convert { it + 1 }
        .forEach {
            print("$it ")
        }
}
```

为了方便理解，我们把作用域函数和集合操作符统称为 Kotlin 的操作符。

2.8.2 操作符的实现原理

作用域函数与集合操作符的原理是完全一致的，这里以集合操作符作为一个例子来看整个作用域函数和集合操作符的运行原理。

来看下这个 Kotlin 内置的操作符 forEach，它其实是对泛型为 T 的 Iterable 的一个扩展函数，其实内部就是执行了 for-in 形式的循环，this 表示当前 T 泛型的 Iterable 对象，循环执行 action 方法，也就是我们传入的 Lambda 闭包。

```
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

同样的，之前写过的自定义扩展函数 convert，也是这样的原理：

```
inline fun <T, E> Iterable<T>.convert(action: (T) -> E): MutableList<E> {
    val list: MutableList<E> = mutableListOf()
```

```
    for (item in this) list.add(action(item))
    return list
}
```

所以基本上不管是作用域函数还是之前提到的集合操作符，它的本质都是**扩展函数**，或者类似于扩展函数的形式为我们的代码进行一系列的扩展操作。

2.8.3 常用操作符

下面为大家列出了 Kotlin 中集合常用的操作符，这些操作符没有必要去记它，只需要根据编译器的提示去使用即可。

元素操作类

操作符	描述
contains	判断是否有指定元素
elementAt	返回对应的元素，越界会抛 ArrayIndexOutOfBoundsException
firstOrNull	返回符合条件的第一个元素，没有返回 null
lastOrNull	返回符合条件的最后一个元素，没有返回 null
indexOf	返回指定元素的下标，没有返回 -1
singleOfNull	返回符合条件的单个元素，如有没有符合或者超过一个，返回 null

判断类

操作符	描述
any	判断集合中是否有满足条件的元素
all	判断集合中的元素是否都满足条件
none	判断集合中是否都不满足条件，是则返回 true
count	查询集合中满足条件的元素个数
reduce	从第一项到最后一项进行累计

过滤类

操作符	描述
filter	过滤所有满足条件的元素
filterNot	过滤所有不满足条件的元素
filterNotNull	过滤 null
take	返回前 n 个元素

转换类

操作符	描述
map	转换成另一个集合（与之前提到的 convert 类似）
mapIndexed	除了转换成另一个集合，还可以拿到 Index 下标
mapNotNull	执行转换前过滤掉 null 的元素

操作符	描述
flatMap	自定义逻辑合并两个集合
groupBy	按照某个条件分组，返回 map

排序类

操作符	描述
reversed	反序
sorted	升序
sortedDescending	降序
sortedBy	自定义排序

2.9 作用域函数

2.9.1 作用域函数是什么

作用域函数是 Kotlin 内置的可以对数据做一系列变换的函数。它们与集合的操作符非常的相似，但是集合的操作符只能用于集合的数据变换，而作用域函数可以应用于所有对象，它可以对所有对象做一系列的操作。

在 Kotlin 中常用的作用域函数有五个：

```
run {...}
with(T) {...}
let {...}
apply {...}
also {...}
```

2.9.2 作用域函数的使用

还是以代码示例的形式讲述，开始前先做一个数据类 User：

```
data class User(var name: String)
fun main() {
    val user = User("Changer0")
    //...
}
```

let 与 run

let 与 run 都会返回闭包的执行结果，区别在于 let 有闭包参数，而 run 没有闭包参数，其中 run 可以通过 this 来获取是谁来调用 run 的，也就是说 this 指代了外层的调用对象。这是他们的唯一区别。

```
val letResult = user.let { user: User -> "let: ${user.name}" }
println(letResult)
val runResult = user.run {"let: ${this.name}" }
println(runResult)
```

特别的，对于 let 函数，根据 Kotlin 的 Lambda 规则如果只有一个参数时可以省略参数不写，使用 it 来代替：

```
val letResult = user.let { "let: ${it.name}" }
```

also 与 apply

also 与 apply 都不返回闭包的执行结果，与上面类似，区别在于 also 有闭包参数，而 apply 没有闭包参数。

```
user.also {
    println("also:${it.name}")
}
user.apply {
    println("apply:${this.name}")
}
```

那他们闭包的返回值结果是什么呢？

打开该作用域函数的声明发现，其实这个作用域函数是对泛型 T 做的扩展函数，对于 also/apply 返回的都是它本身：

```
public inline fun <T> T.also(block: (T) -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    block(this)
    return this
}
```

也就是说，我们可以连续的去调用这个作用域函数，适合链式操作某个对象：

```
user.also {
    println("also:${it.name}")
}.apply {
    println("apply:${this.name}")
}.name
```

takeIf 与 takeUnless

takeIf 与 takeUnless 主要是用于判断。

我们看下 takeIf 作用域函数，发现闭包只能返回一个 Boolean 类型的值，并且会根据你传入的 Lambda 表达式的执行结果来做判断，如果执行结果为 true 则返回当前对象，否则会返回 null，这也就是为什么 takeIf 的返回值为 T? 。

```
public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {
    contract {
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (predicate(this)) this else null
}
```

通常在使用时，在闭包后面使用 ?. 继续执行该对象不为空时的代码，后面通过 ?: 执行该对象为空时的代码：

```
user.takeIf { it.name.length > 0}?.also { println("name:${it.name}") } ?: println("name 为空")
```

takeUnless 则与 takeIf 完全相反，如果 Lambda 表达式执行结果为 false 返回当前对象，否则返回 null：

```
public inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {
    contract {
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (!predicate(this)) this else null
}
```

repeat

repeat 函数在之前已经提到过，其实是对 for-in 形式的循环做的封装，不再做具体的介绍：

```
public inline fun repeat(times: Int, action: (Int) -> Unit) {
    contract { callsInPlace(action) }

    for (index in 0 until times) {
        action(index)
    }
}
```

with

查看 with 函数发现与其他作用域函数不同，它不是扩展函数，而是一个顶级的函数，传入参数 receiver 和 闭包 block，返回 receiver 执行闭包的结果，返回值类型为泛型 R。

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return receiver.block()
}
```

with 一般用于对某个对象的整体赋值，这点在 Android 开发中尤其突出，例如对某个 View 属性的整体赋值，可以使用 with。

```
with(user) {
    this.name = "Changer1007"
}
```

3 Kotlin 协程

> 站在巨人的肩膀上做个笔记，摘录自：https://kaixue.io/kotlin-coroutines-1

3.1 协程是什么

> 协程的概念并没有官方的或者统一的定义，协程原本是一个跟线程非常类似的用于处理多任务的概念，是一种编程思想，并不局限于特定的语言。

那在 Kotlin 中的协程是什么呢？

其实就是一套有 Kotlin 官方提供的**线程 API**。就像 Java 的 Executor 和 Android 的 AsyncTask，Kotlin 协程也对 Thread 相关的 API 做了一套封装，让我们不用过多关心线程也可以方便地写出并发操作，这就是 Kotlin 的协程。

3.2 协程的好处

> 既然 Java 有了 Executor，Android 又有了 Handler 和 AsyncTask 来解决线程间通信，而且现在还有 RxJava，那用协程干嘛呢？协程好在哪儿呢？

协程的好处，本质上跟其他的线程 API 一样，都是为了方便使用，但由于它借助了 Kotlin 的语言优势所以比起其他的实现方案要更方便一点。

协程最基本的功能就是并发也就是多线程，用协程你可以把任务切到后台执行：

```
launch(Dispatchers.IO) {
```

```
    //耗时
}
```

切到前台：

```

launch(Dispatchers.Main) {
   //更新 UI
}
```

这种写法很简单，但它不能算是协程直接使用 Thread 的优势，因为 Kotlin 专门添加了一个函数来简化对线程的直接使用：

```
thread {
    ...
}
```

而 Kotlin 最大的好处是在于你可以把运行在不同线程的代码，写在同一个代码块里，上下两行代码，线程切走再切回来，这是在写 Java 时绝对做不到的。它**可以用看起来同步的方式写出异步代码**，这就是 Kotlin 最有名的 非阻塞式挂起。例如：

```
CoroutineScope(Dispatchers.Main).launch {
    val bitmap = suspendingGetBitmap()// 网络请求，后台线程
    imageView.setImageBitmap(bitmap)// 更新 UI 主线程
}
```

Java 中我们处理这样的场景需要使用回调来解决，一旦逻辑复杂很容易会出现两层或 n 层回调，这就陷入了回调地狱。

而 Kotlin 的协程就完全消除了回调！另外大家不要忘了，回调式可不止多了几个缩进那么简单，它也限制了我们的能力。

比如，我们有个需求，他需要分别执行两次网络请求，然后把结果合并后再展示到界面，按照正常思路这两个接口应该同时发起请求，然后把结果做融合。

但是这种场景如果使用 Java 的回调式就会比较吃力，可能会把两个接口做串行的请求，但这很明显是垃圾代码！（暂时先不考虑 RxJava）

而使用协程可轻松实现，依然是上下两行代码：

```
launch(Dispatchers.Main) {
    val avatar = async { getAvatar() }//获取用户头像
    val logo = async { getLogo() }//获取 Logo
    mergeShowUI(avatar.await(), logo.await())//合并展示
}
```

所以由于 Kotlin 消除了并发任务之间协作的难度，协程可以让我们轻松的写出复杂的并发代码。这些就是协程优势所在！

> async 会在后面的章节中介绍。

3.3 协程的基本使用

3.3.1 配置协程

要想在 Android 工程中使用 Kotlin 协程，需要配置相应的依赖：

根目录下的 build.gradle 配置版本：

```
buildscript {
    ...
```

```
    ext.kotlin_coroutines = '1.3.1'
    ...
}
```

app 下的 build.gradle:

```
dependencies {
    ...
    //                                      依赖协程核心库
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$kotlin_coroutines"
    //                                      依赖当前平台所对应的平台库
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$kotlin_coroutines"
    ...
}

```

3.3.2 创建协程

> 上面提到的 launch 函数不是一个顶层函数，是不能直接使用的，可以通过下面的三种方法来创建：

```
// 方法一，使用 runBlocking 顶层函数
runBlocking {
    getImage(imageId)
}

// 方法二，使用 GlobalScope 单例对象
//            可以直接调用 launch 开启协程
GlobalScope.launch {
    getImage(imageId)
}

// 方法三，自行通过 CoroutineContext 创建一个 CoroutineScope 对象
//                                    需要一个类型为 CoroutineContext 的参数
val coroutineScope = CoroutineScope(context)
coroutineScope.launch {
    getImage(imageId)
}
```

- 方法一通常用于单元测试场景，业务开发一般不会用它，因为它是线程阻塞的；
- 方法二和方法一的区别在于不会线程阻塞。但在 Android 中并不推荐这种用法，因为它的生命周期与 Application 一致，且不可取消；
- 方法三为推荐方式，可以通过传入的 context 参数来管理协程的生命周期（注：这里的 context 和 Android 里的不是一个）

3.3.3 使用协程

接下来开始介绍协程的具体使用，最简单的方式上面已经介绍了：

```
launch(Dispatchers.IO) {
    ...
    getImage(imageId)
    ...
}
```

这个 launch 函数表示含义就是我要创建一个新的协程，并在指定的线程上运行它，这个被创建的协程是谁？就是你传给 launch 的那些代码：

```
...
```

```
    getImage(imageId)
    ...
```

这段连续的代码就是协程。

所以我们就清楚了什么时候用协程，就是**当你切线程或者指定线程的时候**。

例如，子线程中获取图片，主线程中更新 UI：

```
launch(Dispatchers.IO) {
    val image = getImage(imageId)
    launch(Dispatchers.Main) {
        iv.setImageBitmap(image)
    }
}
```

？？？什么情况，还是回调地狱？？？

如果只用 launch 函数协程并不比直接使用线程更加方便，但是协程里有一个很厉害的函数 withContext().

这个函数可以指定线程来执行代码，并且在执行完成之后 **自动把线程切回来** 继续执行。

```
launch(Dispatchers.Main) {
    val image = withContext(Dispatchers.IO){
        getImage(imageId)
    }
    iv.setImageBitmap(image)
}
```

这种写法跟刚才看起来区别不大。 但是如果有了更多的线程切换区别就体现出来了，由于有了自动切回来的功能，协程消除了并发代码在协作时的嵌套，直接写成了上下关系的代码就能让多线程之间进行协作，这就是协程。

> 站在巨人的肩膀上做个笔记，摘录自：https://kaixue.io/kotlin-coroutines-2

3.4 协程挂起

3.4.1 前言

上面中我们提到了下面的示例，它使用了 async 关键字来创建一个协程。

```
launch(Dispatchers.Main) {
    val avatar = async { getAvatar() }//获取用户头像
    val logo = async { getLogo() }//获取 Logo
    mergeShowUI(avatar.await(), logo.await())//合并展示
}
```

我们来看下 launch 和 async 的区别：

- 相同点：它们都可以用来启动一个协程，返回的都是 Coroutine；
- 不同点：async 返回的 Coroutine 多实现了 Deferred 接口。它的意思就是延迟也就是结果稍后才能拿到，调用 `Deferred.await()` 就可以得到结果了。

我们看下 `await` 函数签名如下：

```
//       挂起关键字
public suspend fun await(): T
```

我们看到了suspend关键字，它就是是本篇文章**协程挂起**的主角。

3.4.2 挂起的本质

协程中「挂起」的对象到底是什么？挂起线程，还是挂起函数？都不对，**我们挂起的对象是协程**。

还记得协程是什么吗？启动一个协程可以使用 launch 或者 async 函数，**协程其实就是这两个函数中闭包的代码块**。

launch，async 或者其他函数创建的协程，在执行到某一个 suspend 函数的时候，这个协程会被「suspend」，也就是被挂起。

那此时又是从哪里挂起？**从当前线程挂起**。换句话说，就是**这个协程从正在执行它的线程上脱离**。需要注意，不是这个协程停下来了！是脱离，当前线程不再管这个协程要去做什么了。也可以理解为：**当线程执行到协程的 suspend 函数的时候，暂时不继续执行协程代码了。**

我们先让时间静止，然后兵分两路，分别看看这两个互相脱离的线程和协程接下来将会发生什么事情：

线程：

协程的代码块中，线程执行到了 suspend 函数这里的时候，就直接执行完毕然后返回了！完毕之后线程就该干嘛就干嘛了，如果是后台线程可能就去执行其他任务了，要是**主线程则会继续刷新页面**。

如果你启动一个主线程的协程：

```
coroutineScope.launch {
    val bitmap = suspendingGetBitmap()// 网络请求，后台线程
    imageView1.setImageBitmap(bitmap)// 更新 UI 主线程
}
```

相当于会往你的主线程 post 一个新任务，这个任务就是你的协程代码，当这个协程被挂起的时候实质上你 post 的这个任务就提前结束了（虚线代码直接略过）。

```
handler.post {
        协程代码，遇到 suspend 函数直接执行结束
    -----------------------------------
    |val bitmap = suspendingGetBitmap()|
    |imageView1.setImageBitmap(bitmap) |
    -----------------------------------
}
```

看完线程，再看下协程做了什么。

协程：

协程的代码在到达 suspend 函数的时候被掐断，接下来协程会从这个 suspend 函数开始继续往下执行，不过是在**指定的线程**。

谁指定的？是 suspend 函数指定的，比如我们这个例子中，函数内部的 withContext 传入的 Dispatchers.IO 所指定的 IO 线程。

```
private suspend fun suspendingGetBitmap(): Bitmap {
    return withContext(Dispatchers.IO) {
        val url = URL("https://gitee.com/luluzhang/ImageCDN/raw/master/blog/20200420120447.png")
        val openConnection = url.openConnection() as HttpURLConnection
        BitmapFactory.decodeStream(openConnection.inputStream)
    }
}

```

Dispatchers 调度器，它可以将协程限制在一个特定的线程执行，或者将它分派到一个线程池，或者让它不受限制地运行。

常用的 Dispatchers 调用器有三种：

- Dispatchers.Main：Android 中的主线程
- Dispatchers.IO：针对磁盘和网络 IO 进行了优化，适合 IO 密集型的任务，比如：读写文件，操作数据库以及网络请求
- Dispatchers.Default：适合 CPU 密集型的任务，比如计算

回到我们的协程，它从 suspend 函数开始脱离启动它的线程，继续执行在 Dispatchers 所指定的 IO 线程。

紧接着在 suspend 函数执行完成之后，协程为我们做的最爽的事就来了：会**自动帮我们把线程再切回来**。

切回来是什么意思呢？

示例中我们的协程原本运行在主线程中，当代码遇到 suspend 函数时，发生线程切换，协程代码会在 Dispatchers 指定的 IO 线程中运行，运行结束后，协程会帮我们再 post 一个任务，让剩下的代码回到主线程执行。

从上面两个角度分析后，我们再对协程的挂起做一个解释：

协程在执行到有 suspend 标记的函数的时候，会被 suspend, 也就是被挂起，而所谓的被挂起，就是切个线程；不过区别在于，**挂起函数在执行完成之后，协程会重新切回它原先的线程**。再简单来讲，在 Kotlin 中所谓的挂起，就是一个**稍后会被自动切回来的线程调度操作**。

> 另外，这个「切回来」的动作，在 Kotlin 里叫做 **resume**，恢复。

通过刚才分析我们了解到：挂起之后是需要恢复的。而恢复这个功能是协程的，如果你不在协程里面恢复这个功能就没法实现，这就是为什么**挂起函数必须在协程中或者另一个挂起函数中被调用**的原因。

3.4.3 挂起是如何做到的

首先，我们可以自定义一个挂起函数：

```
private suspend fun suspendingFun() {
    println("Current Thread: ${Thread.currentThread().name}")
}

Logcat 输出
I/System.out: Current Thread: main
```

输出结果还是主线程。嗯？为什么没切线程？因为它不知道往哪儿切，需要我们告诉它。

对比之前的示例，不同之处在于 withContext。

```
private suspend fun suspendingGetBitmap(): Bitmap {
    return withContext。(Dispatchers.IO) {
        ...
    }
}
```

其实通过 withContext 源码可以知道，它本身就是一个挂起函数，它接收一个 Dispatcher 参数，依赖这个 Dispatcher 参数的指示，你的协程被挂起，然后切到别的线程。

所以这个 **suspend，其实并不是起到把任何协程挂起，或者说切换线程的作用**。还需要你在挂起函数里面去调用另外一个挂起函数，而且里面的**这个挂起函数需要是直接或间接调用协程自带的、内部实现了协程挂起代码的挂起函数**，让它来真正做挂起，也就是线程切换的工作。

3.4.4 suspend 存在的意义

通过上面了解到，suspend 并不能真正的实现挂起，那它有什么作用呢？

它其实是一个提醒，是函数创建者对函数调用者的提醒。 提醒调用者我是一个耗时函数，请在协程中调用。

所以我们知道，suspend 关键字的定位就不是用来去操作挂起的，挂起操作靠的是挂起函数里面的实际代码，而不是这个关键字。

例如，我们写一个挂起函数，但是不在内部调用别的挂起函数：

Android Studio 会给你一个提醒，认为这个 suspend 关键字是多余的。因为其内部并没有调用其他挂起函数，也无需在协程中运行。

3.4.5 自定义挂起函数

3.4.5.1 什么时候自定义

其实原则上如果你的函数比较耗时就应该写成挂起函数。哪些操作会比较耗时呢？

耗时操作一般分为两类：**I/O 操作和 CPU 计算工作** 。比如文件的读写、网络交互、图片的模糊处理，都是耗时的，通通可以把它们写进 suspend 函数里。

另外这个「耗时」还有一种特殊情况，就是这件事本身做起来并不慢，但它需要等待，比如 5 秒钟之后再做这个操作。这种也是 suspend 函数的应用场景。

3.4.5.2 如何写

非常简单，给函数加上 suspend 关键字，然后用 withContext 把函数的内容包住就可以了：

```
1  private suspend fun getAvatar() = withContext(Dispatchers.IO) {
2      val url = URL("https://gitee.com/luluzhang/ImageCDN/raw/master/blog/20200626111013.jpg")
       val openConnection = url.openConnection() as HttpURLConnection
3      BitmapFactory.decodeStream(openConnection.inputStream)
4  }
5
```

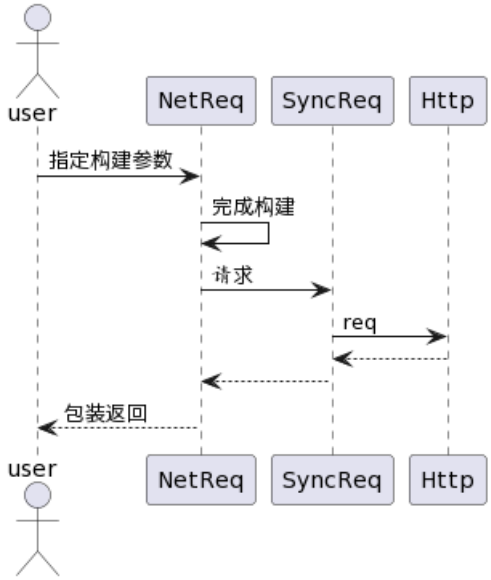
当然并不是只有 withContext 这一个函数来辅助我们实现自定义的 suspend 函数，比如还有一个挂起函数叫 delay，它的作用是等待一段时间后再继续往下执行代码。

使用它就可以实现刚才提到的等待类型的耗时操作:

```
1  suspend fun suspendUntilDone() {
2    while (!done) {
3      delay(5)
4    }
5  }
```

3.5 工程中协程使用

我在咱们工程简单封装了下协程的网络请求供大家使用



3.5.1 示例代码

```
lifecycleScope.launch(Dispatchers.Main) {
    val response = NetReq(Bean::class.java)
        .url(xx)//协议地址
        .addHeader(k, v)//额外Header
        .loadWithCoroutine()//启动协程

    val bean = response.data
    if (bean == null) {
        Log.e(TAG, "Error: ${response.e?.message}")
        return@launch
    }
    //TODO do something
}
```

目前暴露方法:

方法	含义
构造方法	返回 Bean 类型, 根据泛型目前支持四类: InputStream, String,JSONObject,JSONArray,自定义 GSON Bean
url	协议地址
addHeader	除系统 Header 外, 额外 Header
get	明确 Get 请求
postString	发起 String 类型的 post 请求
addFormFile	添加 Form 文件
noUseSysHeader	指定不使用系统 Header
loadWithCoroutine	协程加载
loadWithLiveData	返回 LiveData, 通过 LiveData 监听, 可传入已有 LiveData

3.5.2 注意事项

如果当前在页面组件内（eg：Activity、Fragment），请使用 lifecycleScope 的协程作用域，例如：

```
lifecycleScope.launch {
    //do something

}
```