一文吃透 Kotlin 中眼花缭乱的函数家族

小虾米君 郭霖 2022-10-11 08:00 发表于江苏

/ 今日科技快讯 /

近日，中国裁判文书网显示，沐瞳科技起诉腾讯商业诋毁纠纷一案于近日终审宣判。上海知识产权法院认定，腾讯构成商业诋毁，责令其赔偿原告沐瞳科技经济损失及合理费用等共计22万元，并以书面形式就涉案商业诋毁行为对相关方进行澄清。

沐瞳科技成立于2014年，在东南亚游戏市场成绩斐然。其上线于2016年6月的《无尽对决》在东南亚MOBA（多人在线战术竞技游戏）市场上超越腾讯的《王者荣耀》，在多个国家和地区成为霸主。截至今年6月，《无尽对决》的月活用户突破1亿。2021年3月，沐瞳科技被字节跳动旗下游戏业务品牌朝夕光年收购。

/ 作者简介 /

本篇文章转自TechMerger的博客，文章主要分享了对 Kotlin 中的函数内容相关的整理，相信会对大家有所帮助！

原文地址：

> https://juejin.cn/post/7151626949965086734

/ 前言 /

料想 Kotlin 开发者对于其庞大繁杂的函数家族必深有感触：包括但不限于简化函数、lambda 表达式、匿名函数、高阶函数、扩展函数、内联函数、闭包、顶层函数、局部函数、

运算符重载函数等。

细看这些叫法，貌似用都会用。但要论其个中区别乃至实现原理，则难以说得明白。本文将通过格式、用法和反编译后的代码对这些函数进行整体地盘点和对比，期望能为大家理清这些函数之间的关系。

目录前瞻：

/ 简化函数 /

Kotlin 中定义方法的时候，如果函数体是单个表达式，可以进行函数简化。

```
fun generateAnswerString(count: Int, countThreshold: Int): String {
    return if (count > countThreshold) {
        "I have the answer."
    } else {
        "The answer eludes me."
    }
}
```

简化的形式是直接赋值：不仅可以省略花括号，还可以省略返回类型以及 return 关键字。

这样子的写法很接近于日常表达习惯，简单明了 ~

```
fun generateAnswerString(count: Int, countThreshold: Int) =
    if (count > countThreshold) {
        "I have the answer"
    } else {
        "The answer eludes me"
    }
```

反编译之后发现其采用的三元运算符的写法。

```
@NotNull
public final String generateAnswerString(int count, int countThreshold) {
    return count > countThreshold ? "I have the answer." : "The answer eludes me.";
}
```

/ lambda 表达式和匿名函数 /

Kotlin 中并不要求函数都拥有名称，只声明其必要的输入类型、输出类型以及表达式即可完成函数的定义。

比如：String 即为输入参数类型，Int 为输出类型，花括号内为 lambda 表达式：

- it 是隐式参数名称，也可任意拟定，比如这里用 input 指定。如果不需要参数的话也可以省略
- 如果有不需要的参数一般用下划线_取代其名称
- input.length 为计算逻辑以及表达式返回值

```
    (String) -> Int = { input ->
        input.length
    }
```

但没有名称的函数无法直接调用，我们还得为这个匿名函数指定函数引用，使其可以像属性一样被传递、被灵活调用。

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}
```

定义了名为 stringLength 的函数引用后，函数没有真正的执行，还需要后续的调用。比如：

```
val stringLength: Int = stringLengthFunc("Android")
```

反编译看下。

stringLengthFunc 的函数引用事实上是 Kotlin 中预设的 Function1 接口的实现变量，函数的调用会 invoke 到 Function1 的实现体，即包装了真实表达式的逻辑。

```
public final class Test {
  @NotNull
  private final Function1 stringLengthFunc;
  ...

  public Test() {
    this.stringLength = ((Number)this.stringLengthFunc.invoke("Android")).intValue();
  }
}
```

需要留意的是匿名函数当然可以像普通函数一样无参数、无返回值。无参数的时候 () 内留空即可，无返回值的话返回类型写作 Unit。

```
  val printTime: () -> Unit = {
     Log.d("Test", "current:${System.currentTimeMillis()}")
  }
```

既然匿名函数既然可以像属性一样传递，那么自然可以作为参数传递给其他函数，这就要引出下个话题：高阶函数。

在高阶函数调用的场景里还可以见到匿名函数的另一种形式：无需实例化，直接将函数体传入。

/ 高阶函数 /

高阶函数是将函数用作参数或返回值的函数。支持高阶函数是 Kotlin 函数式编程的一大特性，这在 Kotlin 源码中有大量的使用。

函数作为参数

Kotlin 中函数执行的时候如果需要回调参数继续处理，则无需像 Java 那样定义接口，而是直接将函数作为参数传入。

如下的 stringMapper 即为高阶函数，mapper 即为函数参数的引用名称。

```
class Temp {
   fun stringMapper(input: String, mapper: (String) -> Int): Int {
      return mapper(input)
   }
}
```

匿名函数参数

首先这个函数参数的传入可以是匿名函数的引用，比如：

```
class Temp {
   val stringLengthFunc: (String) -> Int = {
      input -> input.length
   }
   ...
   fun main() {
      stringMapper("Android", stringLengthFunc)
   }
}
```

当然，如果不实例化匿名函数也是可以的，在使用函数的时候将 lambda 表达式直接传入，等待调用。

```
class Temp {
   ...
   fun main() {
      stringMapper("Android", { input ->
         input.length }
      )
   }
}
```

如果传入的 lambda 表达式是最后一个参数的话，可单独拎出，更加简洁。

```
class Temp {
   ...
   fun main() {
      stringMapper("Android"
      ) { input ->
         input.length
      }
   }
}
```

如果高阶函数只有一个函数参数的话，调用的时候可直接省略圆括号。

```
class Temp {
   val temp = "ddd"

   fun stringMapperNew(mapper: (String) -> Int): Int {
      return mapper(temp)
   }

   fun main() {
      stringMapperNew { input ->
         input.length
      }
   }
}
```

反编译后可以看到高阶函数的函数参数实际传入的 Function1 接口的实例，函数参数的执行也是接口的回调。

```
  public final int stringMapper(@NotNull String input, @NotNull Function1 mapper) {
    ...
    return ((Number)mapper.invoke(input)).intValue();
  }
```

具名函数参数

除了传入匿名函数的方法体或引用，还可以传入普通函数的名称作为参数。写法稍稍不同，::functionName 的形式。

```
class Temp {
  private fun stringLengthInner(input: String) = input.length

  fun main() {
    stringMapper("Android", ::stringLengthInner)
  }
}
```

函数作为返回值

如果函数并非想要知道处理结果，只想获得处理方法的话，可以将返回值定义成匿名函数的规格，并在 return 里写上匿名函数的实现。

```
fun stringMapperFunction(input: String): (String) -> Int {
  return {
    val newString = input.substring(
      input.indexOf("start")
    )
    newString.length
  }
}
```

同样的看下实现，即返回的类型是 Function1 接口，结果是实现该接口的匿名内部类。

```
@NotNull
public final Function1 stringMapperFunction(@NotNull final String input) {
  Intrinsics.checkNotNullParameter(input, "input");
  return (Function1)(new Function1() {
    public Object invoke(Object var1) {
      return this.invoke((String)var1);
    }

    public final int invoke(@NotNull String it) {
      Intrinsics.checkNotNullParameter(it, "it");
      String var3 = input;
      int var4 = StringsKt.indexOf$default((CharSequence)input, "start", 0, false, 6, (O
      String var10000 = var3.substring(var4);
      Intrinsics.checkNotNullExpressionValue(var10000, "this as java.lang.String).subs
      String newString = var10000;
      return newString.length();
    }
  });
}
```

与 let 等函数的关系

let 等函数是结合了 inline 函数、扩展函数的高阶函数，以 let 为例看下源码：

```
@kotlin.internal.InlineOnly
public inline fun <T, R> T.let(block: (T) -> R): R {
   contract {
      callsInPlace(block, InvocationKind.EXACTLY_ONCE)
   }
   return block(this)
}
```

可以看到：

- 其实是 T 类型的扩展函数，接受引用名称为 block 、参数类型为 T、返回类型为 R 的匿名函数
- 函数本身又返回和匿名函数参数相同的 R 类型
- 方法体内构建完 contract 之后即调用 block 函数并返回

这样的话，任意对象调用该函数传入的方法体，将拥有等同于对象本身的参数。正如前面所说，lambda 表达式默认用 it 作为其默认引用（当然也可以自定义参数名称），并且 let 方法最终返回的就是方法体的返回类型。

```
   fun main() {
      val lastResult = stringMapper( ... ).let {
           it -> "$it-done"
      }
   }
```

既然 let 能接受匿名函数体，自然也可以接受具名函数传入。

```
class Test {
   private fun stringLengthInner2(input: Int) = "$input-done"

   fun main() {
      val lastResult2 = stringMapper( ... ).let(::stringLengthInner2)
   }
}
```

除了 let，再瞅一眼 also 函数的源码，进行些对比以加深理解：

```
@kotlin.internal.InlineOnly
@SinceKotlin("1.1")
public inline fun <T> T.also(block: (T) -> Unit): T {
   contract {
      callsInPlace(block, InvocationKind.EXACTLY_ONCE)
   }
   block(this)
   return this
}
```

通过观察其定义，不难理解，also 函数将拥有如下特性：

- 方法体将拥有和 let 一样的 it 参数
- 但方法体不存在返回值，即和 let 不同，最后一个表达式的结果无法被 also 直接沿用
- 而且 also 函数返回的是执行函数的对象本身

inline 函数和扩展函数的原理将会在后面阐述，作为高级、匿名函数部分的原理比较简单，就是按照 let 等源码的顺序调用匿名函数并按约定返回相应类型即可。

/ 扩展函数和扩展属性 /

扩展函数

Kotlin 可以实现扩展一个类的新功能而无需继承该类。比如可以为一个不能修改的第三方库中的类编写一个新的函数， 这个新增的函数就像那个原始类本来就有的函数一样，可以用普通的方法调用。这种机制称为扩展函数。

来看一个典型的扩展函数写法：

```
fun String.lastChar(): Char = this[length - 1]
```

- String. 表示扩展的目标类
- lastChar 即函数名

- Char 即函数返回类型
- this 代表当前类的实例，并非必须、可省略
- [...] 即函数体

在 Kotlin 和 Java 中不同的调用方法

Kotlin 中直接调用：

```
class Test {
   ...
   fun main() {
      ...
      Log.d("test", "last char:${"Ellison".lastChar()}")
   }
}

fun String.lastChar(): Char = this[length - 1]
```

Java 中则是像静态类一样调用该扩展方法，要注意两点：

1. Java 中当静态方法调用它，类名为扩展函数所存在的 Kt 文件名 + Kt。此处即为 TestKt
2. 调用函数传入的第一个参数为实例，其后为函数参数

```
public class TestJava {
   static void main(String[] args) {
      Log.d("test", "last char" + TestKt.lastChar("Ellison"));
   }
}
```

泛型扩展函数

对于泛型类的也可以拥有扩展函数，只不过需要在声明的函数前指定泛型，否则无法扩展成功。

比如给 MutableMap 添加新的函数：

```
fun <K, V>  MutableMap<K, V> .putLast(): V? {
```

```
    ...
}
```

与 apply 等函数的关系

以 apply 函数，了解下源码中扩展函数的应用：

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

apply 和 let 等函数一样是扩展自任意类型对象的函数，因为泛型的缘故在函数前添加了 <T> 的声明、block 参数的 T.()->Unit 指的是带有指向 T 实例的 this 的参数并无返回值。

T.()->Unit 这种带有接收者的参数形式被称为函数字面值，其和扩展函数的形式有点像，但并不是。通过反编译之后会发现它仍然属于匿名函数的范畴，通过 Function2 接口实现，只不过传入的参数是 T 其本身。

apply 函数返回 T 类型，内部则是调用 block() 传入 T 对象，进行处理之后，返回对象本身。

从如下的 run 函数的源码可以看出与 apply 之间的区别：其函数参数和返回值均是 R 类型，这将导致像 let 和 also 一样的不同点：

- apply 总是返回的是对象本身
- run 返回的是函数结果

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

扩展属性

扩展属性提供了一种方法能通过属性语法进行访问的 API 来扩展。尽管它们被叫做属性，但是它们不能拥有任何状态，它不能添加额外的字段到现有的 Java 对象实例。

比如下面的为 List 添加一个 last 属性用于获取列表的最后一个元素，this 可以省略。

注意：泛型仍要声明在扩展属性前。

```
val <T> List<T>.last: T get() = get(size - 1)

val listString = listOf("Android Q", "Android N", "Android M")

fun main() {
   println("listString.last${listString.last}")
}
```

与 KTX 的关系

KTX 是专门为 Android 库设计的 Kotlin 扩展程序，以提供简洁易用的 Kotlin 代码，其中部分 KTX 采用了扩展属性的写法，比如 viewModelScope。

它向 ViewModel 类扩展了 viewModelScope 属性，供 ViewModel 中便捷地使用 CoroutineScope：绑定至 Dispatchers.Main，并且会在清除 ViewModel 后自动取消。

```
public val ViewModel.viewModelScope: CoroutineScope
   get() {
      val scope: CoroutineScope? = this.getTag(JOB_KEY)
      if (scope != null) {
         return scope
      }
      return setTagIfAbsent(
         JOB_KEY,
         CloseableCoroutineScope(SupervisorJob() + Dispatchers.Main.immediate)
      )
   }
```

伴生对象扩展函数和属性

如果一个类定义了伴生对象，那么我们也可以为伴生对象定义扩展函数与属性，并且就可以和伴生对象一样使用类名直接访问：

```
class Job {
   companion object {}
}

class Test {
   fun main() {
      Job.print("Extension for Companion object.")
   }
}

fun Job.Companion.print(summary: String) {
   Log.d("Test", "Job:$summary")
}
```

原理

看下上述 String.lastChar() 扩展函数反编译后的代码。

即在 TestKt Class 内生成了同名的静态函数，接收的参数即为目标 Class 即 String 实例，内部将调用扩展函数的函数体。

```
public final class TestKt {
  public static final char lastChar(@NotNull String $this$lastChar) {
    Intrinsics.checkNotNullParameter($this$lastChar, "$this$lastChar");
    return $this$lastChar.charAt($this$lastChar.length() - 1);
  }
}
```

再看下 viewModelScope KTX 的反编译来研究下扩展属性的原理。

同样在 XXXKt 的 ViewModelKt Class 内生成了静态方法，不过名称为 getXXX 形式，其接收的参数为 ViewModel 实例，内部执行 get() 的逻辑并返回。

```
public final class ViewModelKt {
  @NotNull
```

```
  public static final CoroutineScope getViewModelScope(@NotNull ViewModel $this$viewModel
    Intrinsics.checkNotNullParameter($this$viewModelScope, "$this$viewModelScope");
    CoroutineScope scope = (CoroutineScope)$this$viewModelScope.getTag("androidx.lifecyc
    if (scope != null) {
      return scope;
    } else {
      Object var10000 = $this$viewModelScope.setTagIfAbsent("androidx.lifecycle.ViewMode
      Intrinsics.checkNotNullExpressionValue(var10000, "setTagIfAbsent(\n     …Main.imm
      return (CoroutineScope)var10000;
    }
  }
}
```

不要滥用

扩展函数、扩展属性虽好但不要滥用，因为会造成一些弊端：

- 扩展函数无法像普通函数那样进行函数引用
- 多个接受者隐式的访问可能会令人困惑
- 当修改引用接受者的时候，不清楚是修改的是扩展接受者还是调度接受者
- 对于经验较少的开发人员来说，看到成员扩展可能是违反直觉，可读性很差

/ 内联函数 /

inline 函数在调用它的地方，会把这个函数方法体中的所以代码移动到调用的地方，而不是通过方法间压栈进栈的方式。一定程度上可以代码效率。

比如如下的代码：

```
class TestInline {
  fun test() {
    highLevelFunction("Android") {
      it.length
    }
  }

  private fun highLevelFunction(input: String, mapper: (String) -> Int): Int {
    Log.d("TestInline", "highLevelFunction input:$input")
    return mapper(input)
```

```
    }
}
```

highLevelFunction 函数没有添加 inline 的话，反编译之后可以看到 test 函数调用 highLevelFunction 的时候传入了 Function1 接口实例。

```
public final class TestInline {
  public final void test() {
    this.highLevelFunction("Android", (Function1) ...);
  }

  private final int highLevelFunction(String input, Function1 mapper) {
    Log.d("TestInline", Intrinsics.stringPlus("highLevelFunction input:", input));
    return ((Number)mapper.invoke(input)).intValue();
  }
}
```

当添加了 inline 修饰再看下反编译的代码，会发现 highLevelFunction 函数的内容被编译进了 test 函数内。

```
public final class TestInline {
  public final void test() {
    String input$iv = "Android";
    int $i$f$highLevelFunction = false;
    Log.d("TestInline", "highLevelFunction input:" + input$iv);
    int var5 = false;
    input$iv.length();
  }
}
```

但并非所有的函数都适合 inline 标注，强行标注的话会收到 IDE 的警告：

Expected performance impact of inlining '...' can be insignificant. Inlining works best for functions with lambda parameters.

是否使用 inline 函数，可以简单参考如下：

1. 不带参数，或是带有普通参数的函数，不建议使用 inline
2. 带有 lambda 函数参数的函数，建议使用 inline

另外，inline 还可以让函数参数里面的 return 生效。因为平常的高阶函数调用传入方法体不允许 return，但如果该高阶函数标注了 inline 就可以直接 return 整个外部函数。

```
class TestInline {
    fun test() {
        highLevelFunction("Android") {
            it.length
            return // 可以 return 整个 test 函数，后续的 Log 不再输出
        }
        Log.d("TestInline", "tested")
    }

    private inline fun highLevelFunction(input: String, mapper: (String) -> Int): Int {
        ...
    }
}
```

反编译之后会发现，由于 return 的存在，后续的 Log 代码压根没参与编译。

/ 闭包 /

前面提到的 lambda 表达式或匿名函数可以访问其闭包，即便是作用域以外的局部变量，甚至可以进行修改。

比如下面的 stringMapper 的 lambda 参数内可以直接访问和修改外部的 sum 变量。

```
    fun test() {
        var sum = 0

        stringMapper("Android") {
            sum += it.length
            ...
        }

        print(sum)
    }
```

反编译后可以看到传入 stringMapper 高阶函数的是 Function1 接口的实现即匿名内部类，匿名内部类拷贝 sum 引用进行数值的修改操作。

```
public final void test() {
  final IntRef sum = new IntRef();
  sum.element = 0;
  this.stringMapper("Android", (Function1)(new Function1() {
    public Object invoke(Object var1) {
      return this.invoke((String)var1);
    }

    public final int invoke(@NotNull String it) {
      Intrinsics.checkNotNullParameter(it, "it");
      IntRef var10000 = sum;
      var10000.element += it.length();
      return it.length();
    }
  }));
  int var3 = sum.element;
  System.out.print(var3);
}
```

如果高阶函数同时也是内联函数，那么实现较为直接，即在外部函数内直接操作变量即可。

```
public final void test() {
  int sum = 0;
  String input$iv = "Android";
  int $i$f$stringMapper = false;
  int var7 = false;
  int sum = sum + input$iv.length();
  input$iv.length();
  System.out.print(sum);
}
```

/ 顶层函数 /

Kotlin 允许在文件内直接定义函数，这个方法可以被称为顶层函数。

```
// Test.kt
fun topFunction(string: String) {
   println("this is top function for $string")
}
```

这种函数可以在 Kotlin 中被直接调用，无需指定其实例或类名。

```
class TestInline {
   fun test() {
      ...
      topFunction("Ellison")
   }
}
```

在 Java 中调用该顶层函数的话是和扩展函数一样的形式：

```
public class TestJava {
   static void main(String[] args) {
      TestKt.topFunction("Ellison");
   }
}
```

通过反编译会发现，原理跟扩展函数一样，其通过静态函数实现：

```
public final class TestKt {
  ...
  public static final void topFunction(@NotNull String string) {
    Intrinsics.checkNotNullParameter(string, "string");
    String var1 = "this is top function for " + string;
    System.out.println(var1);
  }
}
```

调用处反编译的实现也可想而知。

```
public final class TestInline {
  public final void test() {
    TestKt.topFunction("Ellison");
  }
}
```

留意下和扩展函数的区别：

1. 定义的时候无需指定目标类名
2. 调用的时候自然无需指定实例

/ 局部函数 /

除了允许在文件顶层定义函数外，Kotlin 还允许在现有函数内定义嵌套函数，称为局部函数。

而且该函数还可以访问闭包。

```
fun magic(): Int {
    val v1 = (0..100).random()

    fun foo(): Int {
        return v1 * v1
    }

    return foo()
}
```

通过反编译发现和闭包是一样的实现：

```
public final int magic() {
    byte var2 = 0;
    IntRange var3 = new IntRange(var2, 100);
    final int v1 = RangesKt.random(var3, (Random)Random.Default);
    <undefinedtype> $fun$foo$1 = new Function0() {
        public Object invoke() {
            return this.invoke();
        }

        public final int invoke() {
            return v1 * v1;
        }
    };
    return $fun$foo$1.invoke();
}
```

当然如果没有访问闭包的话。

```
fun magic(): Int {
    val v1 = (0..100).random()

    fun foo(v: Int): Int {
        return v * v
```

```
    }

    return foo(v1)
}
```

实现稍稍区别：

```
public final int magic() {
  byte var2 = 0;
  IntRange var3 = new IntRange(var2, 100);
  int v1 = RangesKt.random(var3, (Random)Random.Default);
  <undefinedtype> $fun$foo$1 = null.INSTANCE;
  return $fun$foo$1.invoke(v1);
}
```

/ 运算符重载函数 /

Kotlin 允许使用 operator 关键字对已有函数进行重载，达到扩展函数参数、改写函数逻辑等目的。

比如如下给 Int 类型的 minus 方法扩展了支持 Person 参数的重载函数，这样的话数字即可与 Person 类型直接进行 - 号运算。

```
data class Person(var name: String, var age: Int)

operator fun Int.minus(p: Person) = this - p.age

fun testOperator() {
    val person1 = Person("A", 3)
    println("testInt+person1=${5 - person1}")
}
```

如果不添加 operator 运算符的话，上述 5 - person1 的写法会无法通过编译，因为 - 运算符不识别 Person 类型。因为这种写法就变成了向 Int 类添加了接收 Person 参数的 minus 方法而已，使用的话就得要改成对象调用函数的形式：

```
...
fun Int.minus(p: Person) = this - p.age

fun testOperator() {
```

```
    val person1 = Person("A", 3)
    println("testInt+person1=${5.minus(person1)}")
}
```

除此之外还可以重载 get()、compareTo() 等函数，在这里还想额外谈下运算符重载函数在解构声明方面的用处。

解构声明

Kotlin 有时会把一个对象解构成很多变量，使用起来会很方便：

```
fun testDeco() {
    val (msg, code) = Result("good", 1)
    println("msg:${msg} code:${code}")
}

inner class Result(
    private val msg: String,
    private val code: Int
) {
    operator fun component1() = msg

    operator fun component2() = code
}
```

componentN() 函数是 Kotlin 中约定的获取解构声明变量的对应运算符，这里面需要使用 operator 进行描述以重载。

解构声明的变量如果不需要使用的话，可以用 _ 来代替，比如：

```
val (_, status) = getResult()
```

解构声明在 Kotlin 中使用非常普遍，比如遍历一个映射（map）最好的方式：

```
for ((key, value) in map) {
  ...
}
```

operator 原理

反编译解构声明的示例代码，可以看到事实上定义了多个变量，每个变量按照顺序调用目标类中实现的 componentN() 函数进行赋值。

```
public final void testDeco() {
    Test.Result var3 = new Test.Result("good", 1);
    Object msg = var3.component1();
    int code = var3.component2();
    String var4 = "msg:" + msg + " code:" + code;
    System.out.println(var4);
}

public final class Result {
    private final String msg;
    private final int code;

    @NotNull
    public final Object component1() {
      return this.msg;
    }

    public final int component2() {
      return this.code;
    }
    ...
  }
```

/ 结语 /

函数在 Kotlin 语法中极为重要，了解其特点和原理对于灵活编程非常必要，再次回顾下各函数的异同及原理。

函数	特点	原理
lambda 表达式	花括号内的函数体，更加简洁、便捷	通过 Kotlin 中预设的 `Function1` 接口实现
匿名函数	定义没有名称的函数引用，供高阶函数使用	同上
高阶函数	接收函数参数或返回函数引用	接收或返回 `Function1` 接口实例
扩展函数	给目标类添加函数或属性	生成 `XXXKt` 类添加 `静态函数` ，参数包括目标实例和其他参数
内联函数	节省匿名函数的内存消耗、提高效率 + return 整个函数	编译期将内敛函数**代码移动到调用**的地方
闭包	lambda 表达式或匿名函数可以直接访问和修改外部函数变量	以匿名内部类形式拷贝外部的对象引用进行修改
顶层函数	无需声明类直接在文件内定义函数，使用时直接调用函数	类似扩展函数的静态函数方式
局部函数	在函数内声明函数并可以访问闭包	和闭包类似的实现或 `lambda` 表达式，取决于是否使用闭包
运算符重载函数	扩展运算符改写参数或逻辑	实现运算符替代逻辑，以函数的形式调用
解构声明	用变量组合的形式进行定义方便直接使用	用运算符重载函数实现 `componentN` 运算符返回目标实例

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

协程简史，一文讲清楚协程的起源、发展和实现

PermissionX 1.7发布，全面支持Android 13运行时权限

欢迎关注我的公众号

学习技术或投稿

长按上图，识别图中二维码即可关注

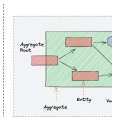
文章已于2022-10-11修改

阅读原文

喜欢此内容的人还喜欢

如何用 DDD 给 DDD 建模，破解 DDD 的魔法?

phodal



10 个 Python 脚本来自动化你的日常任务

Python大数据分析

CTF-Anubis HackTheBox 渗透测试（二）

红日安全