1. Kotlin配置

1、`Android` 中使用 `Kotlin`

1. 需要在项目的根节点下的 `build.gradle` 添加依赖

   classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.50"

2. 在 `Module` 的 `build.gradle` 下添加相关插件和依赖

- 添加插件

  apply plugin: 'kotlin-android' // 扩展插件，用来在代码中直接使用控件ID来进行操作 apply plugin: 'kotlin-android-extensions'

- 添加依赖

  implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.3.50"

2、将现有的 `Java` 代码转为 `Kotlin` 代码

选中要转换的 `Java` 类, 选择顶部工具栏中的 `Code -> Convert Java File to Kotlin File`

3、查看 `Kotlin` 代码转为 `Java` 之后大概是什么代码

1. 打开 `Kotlin` 文件
2. 选择顶部工具栏 `Tools -> Kotlin -> Show Kotlin Bytecode`
3. 在右侧的区域中点击左上角的 `Decompile`

2.函数*变量*类*美剧*类型转换_控制流

```
/**
 * Kotlin 中作用域默认是 public,
 *
 * 声明类，方法 默认都是 final 的，如果需要让别的类继承或重写，需要加 open 关键字
 *
 * 在一个类里面声明另一个类，默认两个类是没有关系的，和 Java 的内部类加了static 一样，如果要变成内
 */

/**
 * 直接将方法写在文件里面，称为顶层函数，这里不用加public，默认就是public
 * 函数接收一个 String 数组
```

```
 *
 * 返回值类型默认是Unit，相当于java中的 void
 *
 * 这里返回值写了Int，表示返回值是Int，Kotlin中只有大写的类型，没有java中的Int等小写的基础数据类
 */
//fun main(args: Array<String>): Int {
//    println(max(1, 2))
//    return 1
//}

/**
 * 定义方法
 *
 * 这里的返回值可以进行推断出来，可以省略
 *
 * 方法参数可以有默认值
 *
 * 这里 if 语句的最后一行默认当做返回值
 */
fun max(x: Int = 0, y: Int) = if (x > y) x else y

/**
 * 顶层变量，声明变量可以省略类型，可以推断出来
 */

// 只有get, final
val name = "Jack"
val name2: String = "Tom"
val name3: String? = null
val name4
    get() = "Bob"

// 有 set/get
var age = 20
var age2: Int = 20
var age3 = 0
    get() {
        return field
    }
    set(value) {
        field = value
    }

fun testField() {
    // 声明变量可以
    var nickName = "Jason"
    // 字符串拼接使用 $，这里的获取值是静态获取，当走完这一行之后就已经确定了
    // 哪怕在后面修改了nickName的值，str的值也不会改变
```

```
    val str = "name: $name age: $age nickName: $nickName"
    println(str)
    nickName = "Danny"
    println("nickName: $nickName str: $str")
}

/**
 * 在Activity中声明变量一般还会用到 lateinit 和 by lazy{}
 */

// 不可空对象声明的时候需要赋值，但是需要findViewById
lateinit var tv: TextView
// 用到的时候才回去new出来这个对象
val strList by lazy { ArrayList<String>() }

/**
 * Kotlin中的字符串
 */
val strHello = "Hello"
val strWorld = "World"
val strHelloWorld = "$strHello $strWorld"
val strMultiLine = """
<activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
""".trimMargin()


fun testStr() {
    // ello World
    println(strHelloWorld.substring(1))
    // lo World
    println(strHelloWorld.substringAfter('l'))
    // d
    println(strHelloWorld.substringAfterLast('l'))
    // He
    println(strHelloWorld.substringBefore("l"))
    // Hello Wor
    println(strHelloWorld.substringBeforeLast('l'))

    // Kotlin为String提供了很多的扩展函数，比如各种replace等等
}

/**
```

```
 * 这里的参数如果不加可见性修饰符就只是作为参数，否则是类的成员
 *
 * 如果类里面没有函数，可以省略类的大括号
 *
 * 这里变量访问的时候都是访问的get/set方法
 *
 * Person("test").name 实际上是调用的 getName 方法，有点类似 groovy
 */
class Person(var name: String?)

/**
 * 空安全
 */
fun testNull(person: Person?) {
    person ?: doSomething() // 如果person为空做一些事
    person ?: return // 如果person为空直接返回

    val person2: Person? = null
    val length = person2?.name?.length ?: 0
    val length2 = person2?.name?.length ?: getLength()
}

fun doSomething() {}
fun getLength() = 0

/**
 * ==: 在 Kotlin 中 == 比较的是值，就相当于java中的equals
 * ===: 比较的是内存地址
 */
fun testEquals() {
    val str1 = String("Hello World.".toCharArray())
    val str2 = String("Hello World.".toCharArray())
    val str3 = "Hello World."
    val str4 = "Hello World."
    println(str1 == str2)
    println(str1 === str2)
    println(str3 === str4)
}

//fun main(args: Array<String>) {
//    testEquals()
//}


/**
 * enum 是一个软关键字，只有跟在 class 前面才是一个关键字
 */
```

```
// 这里不是关键字
val enum = 1

enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    // 如果要在枚举中定义函数，需要用分号将上面的常量和方法分隔开
    // 这可能是kotlin中唯一需要写分号的地方了

    fun rgb() = r + g + b
}

fun test类型推断() {
    // Kotlin 中有类型推断
    val view: View? = null
    if (view is ImageView) {
        view.setImageDrawable(ColorDrawable(android.graphics.Color.RED))
    }

    if (view !is TextView) {
        // 这里不是TextView
    } else {
        view.text = "这里是TextView"
    }
}

/**
 * 控制语句
 */
fun testControlStatement(color: Color) {
    // Java 中的 switch 被换成了 when
    when (color) {
        Color.RED -> println("red")
        // 如果有多个条件，逗号隔开
        Color.GREEN, Color.BLUE -> println("green or blue")
        // 如果有多行，可以加大括号
        else -> {
            println("other 1")
            println("other 2")
        }
    }

    when {
        1 + 1 == 3 -> println("11")
        test() -> println("22")
        else -> println("33")
```

```
    }
}

fun test() = true

fun max2(x: Int = 0, y: Int): Int = when {
    x > y -> x
    else -> y
}

fun testFor() {
    // 这里的区间是 [0, 10]，注意两边都是闭区间，总共执行11次循环
    for (i in 0..10) {
    }

    // [0, 10)
    for (i in 0 until 10) {
    }

    // 从 0 到 10，每次 +2  += 2
    for (i in 0 until 10 step 2) {
    }

    // 从 10 到 0，每次 -2  -= 2
    for (i in 10 downTo 0 step 2) {
    }

    // 这里的 0..10 是一个 IntRange
    val intRange: IntRange = 1..10

    // 跳出
    outLoop@ for (i in intRange) {
        if (i == 5) {
            continue@outLoop
        }
        innerLoop@ for (j in intRange) {
            if (i == 5) {
                break@innerLoop
            }
        }
    }
}
```

3.数组*集合*扩展函数*扩展属性*vararg*infix*析构

```
fun testArray() {
```

```
    // 基础类型的数组要使用 intArrayOf 或者 floatArrayOf 等

    // int[]
    val array = intArrayOf(1, 2, 3)
    // Integer[]
    val array1 = arrayOf(1, 2, 3)
    // Integer[]
    val array2 = Array(3) { it }

    // 取值
    val num1 = array[0]
    val num2 = array.last()
    val num3 = array.component1()
    val num4 = array.elementAt(0)

    // 遍历
    for (i in array) {

    }

    array.forEach {
        // 这里默认参数名是it，可以自己改  x ->
        println(it)
    }

    array.forEachIndexed { index, value ->
        println("index: $index value: $value")
    }

    // 转为String
    array.joinToString()
}

fun testList() {
    val list = listOf<Int>(1, 2, 3)
    val list2 = arrayListOf<Int>(1, 2, 3)

    // 这里的 List 只能进行读取，不能进行增删
    val list3: List<Int> = listOf()

    // 可变的集合可以随意的增删
    val list4: MutableList<Int> = mutableListOf()

    list4.filter { it > 3 }
        .take(3)
        .map { "$it" }
        .flatMap { it.toCharArray().toList() }
        .sorted()
```

```
    // 一个满足就返回
    list4.any { it > 5 }
    // 所有满足的
    list4.all { it > 5 }
    list4.min()
    list4.max()
    list4.sum()
}

/**
 * 扩展方法
 * 比如一般用到字符串都会写一个StringUtil，这个时候就可以用扩展函数替代
 *
 * inline 表示在编译之后会把inline方法里面的内容复制到调用方法的地方
 */
inline fun String.show() {
    Log.d("xxx", "show ${toString()}")
}

/**
 * 扩展属性
 */
val String.lastChar: Char
    get() = get(length - 1)

/**
 * 可变参数：vararg 伪关键字，和 emum 一样
 */
fun test(vararg str: String) {
    val vararg = str[0]
    str.size
    println(str.joinToString(","))
}

/**
 * 方法默认值
 */
fun testFunc(name: String = "", sex: String = "") {
}

fun testFunc2() {
    // 这个时候传参默认是会给到name
    testFunc("男")
    // 如果想给到sex怎么办呢
    testFunc(sex = "男")

    // 通过这样指定参数名的方式，可以节省查看方法参数的时间，而且更加规范
```

```
}

/**
 * 中缀表达式
 */
infix fun Int.加(that: Int): Int = this + that

fun testExtensionFunction() {
    "abc".show()
    "abc".lastChar

    // 中缀调用 infix: 1 to 1
    val x = 1 加 1
}

fun testMap() {
    val map = mapOf<Int, Int>()
    val map2 = mapOf(Pair(1, 1), Pair(2, 2))
    val map3 = mapOf(1 to 1, 2 to 2, "c" to 2)

    map3[1]
    map3["c"]
}

data class User(
    val name: String,
    val age: Int
)

fun test析构() {
    // 从集合析构
    val xxx = "androidx.recyclerview:recyclerview:1.1.0"
    val (group, name, version) = xxx.split(":")

    // 从map析构
    val map = mapOf<Int, Int>(1 to 1, 2 to 2, 3 to 3)
    for ((k, v) in map) {
    }
    map.mapValues {
    }
    map.mapValues { (k, v) ->
    }

    // 从这里析构
    val (sex, age) = Pair("男", 1)
    val (a, b, c) = Triple("a", 2, Color.RED)

    // 从对象析构
```

```
    val user = User("name", 20)
    val (name2, age2) = user

    val userArray = arrayOf<User>()
    for ((name3, age3) in userArray) {

    }
}

# 4. this_异常

class MyActivity : Activity() {

    private val mRecyclerView by lazy { RecyclerView(this) }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        mRecyclerView.run {
            layoutManager = GridLayoutManager(context, 3).apply {
                orientation = RecyclerView.HORIZONTAL

                this@apply.spanCount

                this@run.hasFixedSize()

                // 获取外面的，要获取java的class需要 MainActivity::class.java
                this@MyActivity.startActivity(Intent(context, MainActivity::class.ja
            }
        }
    }
}

class MyException(message: String) : Exception(message)

//// throw 表达式的类型是特殊类型 Nothing.该类型没有值,而是用于标记永远不能达到的代码位置
fun fail(message: String): Nothing {
    throw MyException(message)
}

fun testException() {
    // fail("fail")

    // try-表达式的返回值是 try 块中的最后一个表达式或者是（所有）catch 块中的最后一个表达式。
    // finally 块中的内容不会影响表达式的结果
    val iStr: String? = try {
        "hello"
        throw MyException("error")
```

```
    } catch (e: MyException) {
        "catch 中的值"
        null
    } finally {
        "finally中的值"
    }
    println(iStr)

    //  下面两种方式效果一样
    // val s = iStr?.length ?: throw MyException("iStr.length is null")
    // println(s)

    // val ss = iStr?.length ?: fail("iStr.length is null")
    // println(ss)

    // val x = null            // x 具有类型 Nothing?
    // val l = listOf(null)    // l 具有类型 List<Nothing?>
}

//fun main(args: Array<String>) {
//    testException()
//}
```

5.单例

```
// 最简单的单例
object Singleton {
    var name: String = ""
    fun test() {}
}

fun testSingleton() {
    Singleton.name = "testName"
    Singleton.test()
}

/**
 * 在Kotlin中一般实现接口或者抽象类都是用object
 */
fun testObject() {
    val tv: TextView? = null
    tv?.setOnClickListener(object : View.OnClickListener {
        override fun onClick(v: View?) {
        }
    })
    // 如果接口只有一个方法，可以直接用大括号
```

```
    tv?.setOnClickListener { }

    tv?.addTextChangedListener(object : TextWatcher {
        override fun afterTextChanged(s: Editable?) {}
        override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, aft
        override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count:
    })
}

/**
 * 有时候Android中需要全局单例Application
 */
class MyApp : Application() {

    companion object {
        lateinit var INSTANCE: MyApp
    }

    override fun onCreate() {
        super.onCreate()
        INSTANCE = this
    }
}

/**
 * 可以直接声明顶层变量加 lazy
 */
val testList by lazy { ArrayList<String>() }

/**
 * 加锁的，类似java的DCL
 */
class DoubleCheckSingleton private constructor() {

    companion object {

        // kotlin 写法
        val INSTANCE_1 by lazy(mode = LazyThreadSafetyMode.SYNCHRONIZED) {
            DoubleCheckSingleton()
        }

        // 翻译Java写法
        private var INSTANCE_2: DoubleCheckSingleton? = null

        // 如果要给方法加锁要用
        // @Synchronized
        fun getInstance(): DoubleCheckSingleton {
            if (null == INSTANCE_2) {
```

```
                synchronized(DoubleCheckSingleton::class) {
                    if (null == INSTANCE_2) {
                        INSTANCE_2 = DoubleCheckSingleton()
                    }
                }
            }
            // 这里用!! 表示强制转换为非空的
            return INSTANCE_2!!
        }
    }
}

/**
 * 内部类的方式
 */
class InnerStaticSingleton private constructor() {

    companion object {
        fun getInstance() = Holder.INSTANCE
    }

    private object Holder {
        val INSTANCE = InnerStaticSingleton()
    }
}
```

6.Java调用Kotlin

1、kotlin编译后会生成一个 `类名 + Kt` ,如果不想使用这个名字，可以自己自定义

```
在文件第一行添加注解 @file:JvmName("名字")
例如 @file:JvmName("Hello")
```

2、访问顶层属性

```
var name = "1"
```

通过 `文件名Kt` 来调用set/get方法

```
String name = JavaClassKt.getName();
```

如果想直接通过 "文件名.属性名" 的方式

- 方法一: 通过添加@JvmField属性可以做到

```
@JvmField
var name1 = "1"

那么在java里    就可以
String name1 = JavaClassKt.name1;
```

- 方法二: 使用 `const` 关键字

```
// const 只能用来修饰 val , 相当于java中的 public static final
const val name2 = "1"

Stringg name2 = JavaClassKt.name2;
```

3、合并文件

```
假如有
AUtils.kt,里面有个 a 方法
 和
 BUtils.kt,里面有个 b 方法

给AUtils.kt设置名字叫Utils(通过@file:JvmName("Utils"))
给BUtils.kt设置名字叫Utils(通过@file:JvmName("Utils"))

这个时候通过Utils调用方法的时候只能调用 a 方法，但是我还想要调用 b 方法

就需要在 @file:JvmName("Utils") 下面添加 @file:JvmMultifileClass，表示可以合并

在AUtils.kt 和 BUtils.kt 里面都添加
```

4、扩展方法

```
fun String.test() {}
fun Int.test() {}

在java 中调用的时候需要通过 文件名.test(参数就是扩展类型)
比如
Hello.test("string");
Hello.test(1);
```

5、相同签名的不同泛型的方法

```
fun List<String>.test() {}
fun List<Int>.test() {}

上面这两个方法因为java中的签名一样，都是List，所以有问题

就需要修改下方法名

@JvmName("test1")
fun List<String>.test() {}
fun List<Int>.test() {}
```

`java` 中调用

```
// 相同签名的不同泛型
Hello.test(new ArrayList<Integer>());
Hello.test1(new ArrayList<String>());
```

6、方法默认值

```
fun func(name: String = "", id: Int = 1) {}

kotlin中两个参数都有默认值，在java中调用的话两个参数都需要传
```

如果在 `java` 中不传参数的话需要添加注解

```
@JvmOverloads
fun func(name: String = "", id: Int = 1) {}

这个时候就可以调用了

// 有默认值的方法
Hello.func();
Hello.func("");
Hello.func("", 1);
```

7、类

```
class A(name: String) {
    val name1 = name
    @JvmField
    val name2 = name

    companion object AA {
        val id = "2"

        @JvmField
        val sex = "男"
    }
}
```

`java` 中调用

```
// 类
A a = new A("name");
n = a.getName1();
n = a.name2;
String id = A.AA.getId();
String sex = A.sex;
```

8、object

```
object B {
    val id = "1"
}
```

`java` 中调用

```
// object
id = B.INSTANCE.getId();
```

9、异常

```
fun ex() {
    throw IOException("xxx")
}
```

在 `kotlin` 中抛出异常， `java` 中是没有提示要处理的，如果需要提示可以添加注解

```
@Throws(IOException::class)
fun ex() {
    throw IOException("xxx")
}
```

`java` 中

```
// 异常
try {
    Hello.ex();
} catch (IOException e) {
    e.printStackTrace();
}
```

7. Kotlin调用Java

1、属性名或者方法名是关键字

```
在 kotlin 中 is、object等都是关键字，但是 java 中不是

public static void is() {}
public static void object() {}
```

在 `kotlin` 中调用的时候需要添加 `名字`

```
// is  object 是关键字
JavaTest.`is`()
JavaTest.`object`()
```

2、 `Java` 中数组与可变长度参数

`Java` 中的可变参数其实是一个数组

```
public static void array(String[] args) {
    array2(args);
}
public static void array2(String... args) {}
```

在 `Kotlin` 中调用

```
// java数组与可变长度参数
JavaTest.array(arrayOf<String>("a", "b"))
// 这里要前面加一个 * 号，展开操作符，相当于参数就是 "a","b"
JavaTest.array2(*arrayOf("a", "b"))
```

3、 `Java Object`

```
Java 中类的父类是 Object，Object 里面有 wait() 等
```

如果要在 Kotlin 中调用 wait()

```
// 调用java object里的方法
val o = JavaTest() as java.lang.Object
o.wait()
```

4、操作符重载

```
/**
 * invoke 在kotlin中是操作符
 */
 public void invoke() {}
```

`invoke` 在 `kotlin` 中是操作符重载方法的方法名，则可以使用操作符调用相应的 `java` 方法

```
val jt = JavaTest()
jt.invoke()
JavaTest()()
jt()
```

8.委托

```
class MyList<T> : List<T> by ArrayList<T>()

interface A {
    fun sayHello()
}

class Users : A {
    override fun sayHello() {
        println("Hello")
    }

    val name = "1"
}

// 我要有 A 接口的功能，但是不想自己做，交给了另一个人帮我做，就是 by  委托
class User2 : A by Users()

fun main() {
    val aaa by lazy { "" }
    User2().sayHello()
}
```