1.1 简介

Kotlin 是一种在 Java 虚拟机上运行的静态类型编程语言，被称之为 Android 世界的Swift，由 JetBrains 设计开发并开源。

Kotlin 可以编译成Java字节码，也可以编译成 JavaScript，方便在没有 JVM 的设备上运行。

在Google I/O 2017中，Google 宣布 Kotlin 成为 Android 官方开发语言。

JetBrains这家公司非常牛逼，开发了很多著名的软件，他们在使用Java的过程中发现java比较笨重不方便，所以就开发了kotlin，kotlin是一种全栈的开发语言，可以用它进行开发web、web后端、Android等。

## 为什么选择 Kotlin？

* 简洁: 大大减少样板代码的数量。
* 安全: 避免空指针异常等整个类的错误。
* 互操作性: 充分利用 JVM、Android 和浏览器的现有库。
* 工具友好: 可用任何 Java IDE 或者使用命令行构建。

1.2环境搭建

1. IntelliJ IDEA——纯编译环境
2. Android Studio——带模拟器环境

1.3基本语法

包声明

代码文件的开头一般为包的声明

package com.runoob.main

import java.util.\*

fun test() {}

class Runoob {}

kotlin源文件不需要相匹配的目录和包，源文件可以放在任何文件目录。

以上例中 test() 的全名是 com.runoob.main.test、Runoob 的全名是 com.runoob.main.Runoob。

如果没有指定包，默认为 default 包。

函数声明

函数定义使用关键字 fun，参数格式为：参数 : 类型

fun sum(a: Int, b: Int): Int { // Int 参数，返回值 Int

return a + b

}

也支持表达式作为函数体，返回类型会自动推断，如下：

fun sum(a: Int, b: Int) = a + b

public fun sum(a: Int, b: Int): Int = a + b // public 方法则必须明确写出返回类型

无返回值的函数(类似Java中的void)：

fun printSum(a: Int, b: Int): Unit {

print(a + b)

}

// 如果是返回 Unit类型，则可以省略(对于public方法也是这样)：

public fun printSum(a: Int, b: Int) {

print(a + b)

}

可变长参数函数

函数的变长参数可以用 vararg 关键字进行标识：

fun vars(vararg v:Int){

for(vt in v){

print(vt)

}

}

// 测试

fun main(args: Array<String>) {

vars(1,2,3,4,5) // 输出12345

}

lambda(匿名函数)

// 测试

fun main(args: Array<String>) {

val sumLambda: (Int, Int) -> Int = {x,y -> x+y}

println(sumLambda(1,2)) // 输出 3

}

1.4常亮与变量

可变变量定义：var 关键字

var <标识符> : <类型> = <初始化值>

不可变变量定义：val 关键字，只能赋值一次的变量(类似Java中final修饰的变量)

val <标识符> : <类型> = <初始化值>

常量与变量都可以没有初始化值,但是在引用前必须初始化

编译器支持自动类型判断,即声明时可以不指定类型,由编译器判断。

val a: Int = 1

val b = 1 // 系统自动推断变量类型为Int

val c: Int // 如果不在声明时初始化则必须提供变量类型

c = 1 // 明确赋值

var x = 5 // 系统自动推断变量类型为Int

x += 1 // 变量可修改

注释

和java保持一致，唯一的区别是块注释支持嵌套

字符串模板

$ 表示一个变量名或者变量值

$varName 表示变量值

${varName.fun()} 表示变量的方法返回值:

var a = 1

// 模板中的简单名称：

val s1 = "a is $a"

a = 2

// 模板中的任意表达式：

val s2 = "${s1.replace("is", "was")}, but now is $a"

NULL检查机制

Kotlin的空安全设计对于声明可为空的参数，在使用时要进行空判断处理，有两种处理方式：

字段后加!!像Java一样抛出空异常，

字段后加?可不做处理返回值为 null 或 配合 ?: 做空判断处理

//类型后面加?表示可为空

var age: String? = "23"

//抛出空指针异常

val ages = age!!.toInt()

//不做处理返回 null

val ages1 = age?.toInt()

//age为空返回-1

val ages2 = age?.toInt() ?: -1

当一个引用可能为 null 值时, 对应的类型声明必须明确地标记为可为 null。

当 str 中的字符串内容不是一个整数时, 返回 null:

fun parseInt(str: String): Int? {

return str.toIntOrNull()

}

fun printProduct(arg1: String, arg2: String) {

val x = parseInt(arg1)

val y = parseInt(arg2)

// 直接使用 `x \* y` 会导致错误, 因为它们可能为 null

if (x != null && y != null) {

// 在进行过 null 值检查之后, x 和 y 的类型会被自动转换为非 null 变量

println(x \* y)

}

else {

println("'$arg1' or '$arg2' is not a number")

}

}

fun main() {

printProduct("6", "7")

printProduct("a", "7")

printProduct("a", "b")

}

1.5 类型检测及自动类型转换

我们可以使用 is 运算符检测一个对象是否是指定类型的实例(类似于 Java 中的 instanceof 关键字)。

当 is 检测通过时，Kotlin 会自动将 obj 视为指定类型，因此在 if 语句的分支内不需要显式地进行类型转换。这被称为智能类型转换。

fun main() {

val obj: Any = "Hello, Kotlin"

if (obj is String) {

println("字符串长度: ${obj.length}") // 在这里 `obj` 已被智能转换为 `String`

} else {

println("不是字符串类型")

}

}

fun getStringLength(obj: Any): Int? {

// 在 `&&` 运算符的右侧, `obj` 的类型会被自动转换为 `String`

if (obj is String && obj.length > 0)

return obj.length

return null

}

1.5区间

区间表达式由具有操作符形式 .. 的 rangeTo 函数辅以 in 和 !in 形成。

区间是为任何可比较类型定义的，但对于整型原生类型，它有一个优化的实现。以下是使用区间的一些示例:

for (i in 1..4) print(i) // 输出“1234”

for (i in 4..1) print(i) // 什么都不输出

if (i in 1..10) { // 等同于 1 <= i && i <= 10

println(i)

}

// 使用 step 指定步长

for (i in 1..4 step 2) print(i) // 输出“13”

for (i in 4 downTo 1 step 2) print(i) // 输出“42”

// 使用 until 函数排除结束元素

for (i in 1 until 10) { // i in [1, 10) 排除了 10

println(i)

}