Kotlin Android 环境搭建 Kotlin 基本数据类型

Kotlin 基础语法

Kotlin 文件以 .kt 为后缀。

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。

默认导入

有多个包会默认导入到每个 Kotlin 文件中：

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

函数定义

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
}
```

表达式作为函数体，返回类型自动推断：

```
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)

lambda表达式使用实例：

```
// 测试
fun main(args: Array<String>) {
    val sumLambda: (Int, Int) -> Int = {x,y -> x+y}
    println(sumLambda(1,2))  // 输出 3
}
```

定义常量与变量

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

Kotlin 支持单行和多行注释，实例如下：

```
// 这是一个单行注释

/* 这是一个多行的
   块注释。 */
```

与 Java 不同, Kotlin 中的块注释允许嵌套。

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，另一种字段后加?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? {
  // ...
}
```

以下实例演示如何使用一个返回值可为 null 的函数:

```
fun main(args: Array<String>) {
  if (args.size < 2) {
    print("Two integers expected")
    return
  }
  val x = parseInt(args[0])
  val y = parseInt(args[1])
  // 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
    print(x * y)
  }
}
```

类型检测及自动类型转换

我们可以使用 is 运算符检测一个表达式是否某类型的一个实例(类似于Java中的instanceof关键字)。

```
fun getStringLength(obj: Any): Int? {
  if (obj is String) {
    // 做过类型判断以后，obj会被系统自动转换为String类型
    return obj.length
  }

  //在这里还有一种方法，与Java中instanceof不同，使用!is
  // if (obj !is String){
  //   // XXX
  // }

  // 这里的obj仍然是Any类型的引用
  return null
}
```

或者

```
fun getStringLength(obj: Any): Int? {
  if (obj !is String)
    return null
  // 在这个分支中，`obj` 的类型会被自动转换为 `String`
  return obj.length
}
```

甚至还可以

```
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
}
```

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)
}
```

实例测试

```
fun main(args: Array<String>) {
    print("循环输出：")
    for (i in 1..4) print(i) // 输出“1234”
    println("\n----------------")
    print("设置步长：")
    for (i in 1..4 step 2) print(i) // 输出“13”
    println("\n----------------")
    print("使用 downTo：")
    for (i in 4 downTo 1 step 2) print(i) // 输出“42”
    println("\n----------------")
    print("使用 until：")
    // 使用 until 函数排除结束元素
    for (i in 1 until 4) {   // i in [1, 4) 排除了 4
        print(i)
    }
    println("\n----------------")
}
```

输出结果：

```
循环输出：1234
----------------
设置步长：13
----------------
使用 downTo：42
----------------
使用 until：123
----------------
```

← Kotlin Android 环境搭建 Kotlin 基本数据类型 →

点我分享笔记