Kotlin速成手册

快速入手Kotlin

Kotlin是Android开发者广泛使用的一种编程语言。本文是一个Kotlin速成手册，让你快速使用起来。

变量声明

Kotlin使用两个不同的关键字声明变量：val和var。

- 用val修饰不可变的常量(类似Java中的final)
- 用var修改可变变量

在下面的示例中，count是Int类型的变量，其初始值为10：

```
var count: Int = 10
```

Int是一种表示整数的类型，是可以用Kotlin表示的众多数字类型之一。与其他语言类似，您还可以根据数字数据使用Byte、Short、Long、Float和Double。

var关键字意味着您可以根据需要重新分配要计数的值。例如，可以将count的值从10更改为15：

```
var count: Int = 10
count = 15
```

不过，有些值是不应该改变的。考虑一下名为languageName的字符串。如果要确保languageName始终保持值“Kotlin”，则可以使用val关键字声明languageName：

```
val languageName: String = "Kotlin"
```

这些关键字允许您明确说明可以更改的内容。根据需要利用它们。如果变量引用必须可重新分配，则将其声明为var。否则，请使用val。

类型推断

继续前面的示例，当您将初始值赋给languageName时，Kotlin编译器可以根据所赋值的类型推断类型。

由于“Kotlin”的值是String类型，编译器推断languageName也是一个String。注意，Kotlin是一种静态类型语言。这意味着类型在编译时被解析，并且永远不会更改。

在下面的示例中，languageName被推断为String，因此不能调用不属于String类的任何函数：

```
val languageName = "Kotlin"
val upperCaseName = languageName.toUpperCase()

// Fails to compile
languageName.inc()
```

toupperCase()是一个只能对String类型的变量调用的函数。由于Kotlin编译器已将languageName推断为String，因此可以安全地调用toupperCase()。inc()是一个Int运算符函数，因此不能对String调用它。Kotlin的类型推断方法既简洁又安全。

空安全

在某些语言中，可以在不提供初始显式值的情况下声明引用类型变量。在这些情况下，变量通常包含一个null。默认情况下，Kotlin变量不能包含null。这意味着以下代码段无效：

```
// Fails to compile
val languageName: String = null
```

要使变量保持null，它必须是可为空的类型。您可以指定一个变量作为可null的，必须在类型后面加个?，如下面的例子所示：

```
val languageName: String? = null
```

对于String?类型，可以为languageName指定String值或null。

必须小心处理可null的变量，否则可能会出现可怕的NullPointerException异常。例如，在Java中，如果试图对空值调用方法，程序将崩溃。
Kotlin提供了许多安全处理可为空变量的机制。有关更多信息，请参阅 Common Kotlin patterns in Android: Nullability.

条件语句

Kotlin有几种实现条件逻辑的机制。其中最常见的是if-else语句。如果if关键字旁边括号中的表达式计算结果为true，则执行该分支中的代码（即紧随其后的代码，该代码用大括号括起来）。否则，将执行else分支中的代码。

```
if (count == 42) {
    println("I have the answer.")
} else {
    println("The answer eludes me.")
}
```

可以使用else if表示多个条件。这使您可以在单个条件语句中表示更细粒度、更复杂的逻辑，如下例所示：

```
if (count == 42) {
    println("I have the answer.")
} else if (count > 35) {
    println("The answer is close.")
} else {
    println("The answer eludes me.")
}
```

条件语句对于表示有状态的逻辑很有用，但是您可能会发现在编写它们时会重复自己的语句。在上面的示例中，只需在每个分支中打印一个字符串。为了避免这种重复，Kotlin提供了条件表达式。最后一个示例可以重写如下：

```
val answerString: String = if (count == 42) {
    "I have the answer."
} else if (count > 35) {
    "The answer is close."
} else {
    "The answer eludes me."
}

println(answerString)
```

隐式地，每个条件分支在其最后一行返回表达式的结果，因此不需要使用return关键字。因为这三个分支的结果都是String类型，所以if-else表达式的结果也是String类型。在本例中，answerString从if-else表达式的结果中分配一个初始值。类型推断可用于省略answerString的显式类型声明，但为了清楚起见，通常最好包含它。

注意：Kotlin不包含传统的 三元运算符，而是倾向于使用条件表达式。

随着条件语句的复杂性增加，您可以考虑用一个when表达式替换if-else表达式，如下面的示例所示：

```
val answerString = when {
    count == 42 -> "I have the answer."
    count > 35 -> "The answer is close."
    else -> "The answer eludes me."
}

println(answerString)
```

when表达式中的每个分支都由条件、箭头（->）和结果表示。如果箭头左侧的条件计算结果为true，则返回右侧表达式的结果。注意，执行不会从一个分支到下一个分支。when表达式示例中的代码在功能上与前一示例中的代码等效，但可以说更易于阅读。

Kotlin的条件句突出了它的一个更强大的功能：智能转换。您可以使用条件语句检查变量是否包含对空值的引用，而不是使用safe call运算符或not null断言运算符来处理可空值，如下例所示：

```
val languageName: String? = null
if (languageName != null) {
    // No need to write languageName?.toUpperCase()
    println(languageName.toUpperCase())
}
```

在条件分支中，languageName可以被视为非null。Kotlin足够聪明，能够认识到执行分支的条件是languageName不包含null，因此您不必在该分支中将languageName视为可null。此智能强制转换可用于null检查、type checks类型检查或满足contract的任何条件。

函数

可以将一个或多个表达式分组到一个function中。不必每次需要结果时都重复相同的表达式序列，您可以将表达式包装到函数中，然后调用该函数。

要声明函数，请使用fun关键字后跟函数名。接下来，定义函数接受的输入类型（如果有的话），并声明它返回的输出类型。函数体是定义调用函数时调用的表达式的地方。

在前面的例子的基础上，这里有一个完整的Kotlin函数：

```
fun generateAnswerString(): String {
    val answerString = if (count == 42) {
        "I have the answer."
    } else {
        "The answer eludes me"
    }

    return answerString
}
```

上面示例中的函数名为generateAnswerString。它不需要任何输入。它输出String类型的结果。若要调用函数，请使用其名称，后跟调用运算符()。在下面的示例中，answerString变量使用generateAnswerString()的结果初始化。

```
val answerString = generateAnswerString()
```

函数可以将参数作为输入，如下例所示：

```
fun generateAnswerString(countThreshold: Int): String {
    val answerString = if (count > countThreshold) {
        "I have the answer."
```

```
    } else {
        "The answer eludes me."
    }

    return answerString
}
```

在声明函数时，可以指定任意数量的参数及其类型。在上面的示例中，generateAnswerString()接受一个名为countThreshold的Int类型的参数。在函数中，可以使用该参数的名称来引用该参数。

调用此函数时，必须在函数调用的括号中包含参数：

```
val answerString = generateAnswerString(42)
```

简化函数声明

generateanswerstring()是一个相当简单的函数。函数声明一个变量，然后立即返回。从函数返回单个表达式的结果时，可以跳过声明局部变量，方法是直接返回函数中包含的if else表达式的结果，如下例所示：

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

还可以用赋值运算符替换return关键字：

```
fun generateAnswerString(countThreshold: Int): String = if (count >
countThreshold) {
```

```
        "I have the answer"
    } else {
        "The answer eludes me"
    }
```

匿名函数

不是每个函数都需要一个名称。有些功能更直接地由它们的输入和输出来识别。这些函数称为匿名函数。您可以保留对匿名函数的引用，稍后使用此引用调用匿名函数。您也可以像传递其他引用类型一样，在应用程序周围传递引用。

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

与命名函数一样，匿名函数可以包含任意数量的表达式。函数的返回值是最终表达式的结果。

在上面的示例中，stringLengthFunc包含对匿名函数的引用，该函数接受字符串作为输入，并返回输入字符串的长度作为int类型的输出。因此，该函数的类型被表示为（string）->Int。但是，此代码不调用该函数。要检索函数的结果，必须像调用命名函数一样调用它。调用stringLengthFunc时必须提供字符串，如下例所示：

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

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

高阶函数

函数可以将另一个函数作为参数。使用其他函数作为参数的函数称为高阶函数。这种模式对于组件之间的通信非常有用，就像在Java中使用回调接口一样。
下面是一个高阶函数的例子：

```
fun stringMapper(str: String, mapper: (String) -> Int): Int {
    // Invoke function
    return mapper(str)
}
```

函数的作用是：获取一个String和一个function，该function从传递给它的String中派生一个Int值。
可以通过传递String和满足其他输入参数的function来调用stringMapper()，即接受String作为输入并输出Int的函数，如下例所示：

```
stringMapper("Android", { input ->
    input.length
})
```

如果匿名函数是在函数上定义的最后一个参数，则可以将其传递到用于调用函数的括号之外，如下例所示：

```
stringMapper("Android") { input ->
    input.length
}
```

匿名函数可以在Kotlin标准库中找到。了解更多信息，见 Higher-Order Functions and Lambdas.

类

到目前为止提到的所有类型都内置到Kotlin编程语言中。如果要添加自己的自定义类型，可以使用class关键字定义类，如下例所示：

```
class Car
```

属性

类使用属性表示状态。 属性(property)是类级变量，可以包括getter、setter和backing字段。由于汽车需要轮子来驱动，因此可以将轮子对象列表添加为汽车的属性，如下例所示：

```
class Car {
    val wheels = listOf<Wheel>()
}
```

注意，wheels是一个public val，这意味着可以从Car类外部访问wheels，并且不能重新分配它。如果要获取Car的实例，必须首先调用其构造函数。从那里，您可以访问它的任何可访问属性。

```
val car = Car() // construct a Car
val wheels = car.wheels // retrieve the wheels value from the Car
```

如果要自定义wheels，可以定义自定义构造函数，指定如何初始化类属性：

```
class Car(val wheels: List<Wheel>)
```

在上面的示例中，类构造函数将List<wheel>作为构造函数参数，并使用该参数初始化其wheels属性。

类函数和封装

类使用函数来建模行为。函数可以修改状态，帮助您只公开希望公开的数据。这种访问控制是一个更大的面向对象的概念（称为封装）的一部分。

在下面的示例中，doorLock属性对car类之外的任何内容都是私有的。要解锁汽车，必须调用unlockDoor()函数传入有效的密钥，如下例所示：

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

如果希望自定义属性的引用方式，可以提供自定义getter和setter。例如，如果希望在限制对其setter的访问时公开属性的getter，则可以将该setter指定为private：

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    val gallonsOfFuelInTank: Int = 15
        private set

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

通过属性和函数的组合，可以创建对所有类型对象建模的类。

互操作性

Kotlin最重要的特性之一是它与Java的流畅互操作性。因为Kotlin代码编译成JVM字节码，所以Kotlin代码可以直接调用Java代码，反之亦然。这意味着您可以直接利用Kotlin现有的Java库。此外，大多数Android APIs都是用Java编写的，您可以直接从Kotlin调用它们。

Kotlin是一种灵活、实用的语言，有着越来越多的支持和动力。如果你还没有试过，我们鼓励你试一试。下一步，
请查看Kotlin官方文档以及如何在Android应用程序中应用Kotlin常见模式的指南。

微信扫一扫
关注该公众号