# 《Android Kotlin-GET STARTED》

## 原文地址

https://developer.android.com/kotlin/androidbasics

## 从头开始学习android和kotlin

## Android Kotlin的基础

这门课程教没有编程经验的人如何构建简单的Android应用程序。在此过程中，您将学习编程的基础知识和Kotlin编程语言的基础知识。您将使用Android Studio构建一个Android应用程序集合，开始向您展示Android的可能性。

[课程地址](https://developer.android.com/courses/android-basics-kotlin/course)

## 学习kotlin程序语言

kotlin是一种被android开发者广泛使用的编程语言。这个主题可以作为一个kotlin速成课程，帮助您快速入门和运行。

### 变量声明

Kotlin使用两个不同的关键字来声明变量:val和var。

* 使用val声明变量的时候你不能去修改这个变量的值。
* 使用var声明变量的时候你可以去修改这个变量的值。

例子如下，count是一个var声明的Int类型的变量，这个变量的值初始化分配成10.

|  |
| --- |
| var count: Int = 10 |

Int是一种表示整数的类型，整数是可以用Kotlin表示的许多数字类型之一。与其他语言类似，您也可以根据您的数字数据使用Byte、Short、Long、Float和Double。

这个var关键词你可以重新分配count当你需要的时候，例如，你能改变这个count的值从10到15。

|  |
| --- |
| var count: Int = 10 count = 15 |

不过，有些值是不需要更改的。考虑一个名为languageName的字符串。如果你想确保languageName总是保存一个值“Kotlin”，那么你可以使用val关键字来声明languageName:

|  |
| --- |
| val languageName: String = "Kotlin" |

这些变量允许你明确的说明哪些变量是允许修改的。在合适的时候使用他们会让你更有优势。如果这个变量参数必须能重新被分配那么就应该声明为var，反之则为val。

#### 类型推断

书接上文，当你初始化分配languageName的值，这个kotlin编译器能推断根据你赋予的值推断出相应类型。

因为“Kotlin”的值是字符串类型的，所以编译器推断languageName也是一个字符串。请注意，Kotlin是一种静态类型语言。这意味着该类型是在编译时解析的，并且永远不会改变。

关注下面的例子，languageName被推断出是一个String类型，所以你不能够调用任何不属于String类的方法：

|  |
| --- |
| val languageName = "Kotlin" val upperCaseName = languageName.toUpperCase()  // Fails to compile languageName.inc() |

toUpperCase()方法只能String类型的变量调用，因为Kotlin编译器推断出languageName是一个String类型，你能安全的调用toUpperCase()方法。然后inc()方法是Int类的操作函数，所以String类型不能去调用。Kotln的类型推断方法给予了你简洁性和类型安全性。

#### 空安全

在某些语言中，你可以声明引用变量而不提供一个初始化值，在这些情况下，这个变量通常包含了一个空值，Kotlin变量不能保存空值，这意味着以下代码片段无效:

|  |
| --- |
| // Fails to compile val languageName: String = null |

变量保存null，它必须是一个可以保存null的类型，你可以在变量添加一个后缀？声明了一个特殊的变量来保存null，展示代码例子：

|  |
| --- |
| val languageName: String? = null |

用这个String？是你可以为这个languageName变量分配string或者null。

您必须小心处理可为空的变量，否则可能会出现可怕的NullPointerException异常。例如，在Java中，如果试图在空值上调用方法，程序就会崩溃。

Kotlin提供了许多安全使用可空变量机制，更多信息请查看: [android中普通的kotlin模式](https://developer.android.com/kotlin/common-patterns" \l "nullability)

### 控制语句

Kotlin提供了几种实现条件逻辑的机制。其中最常见的是if-else语句。如果在If关键字旁边用圆括号括起来的表达式求值为true，则执行该分支中的代码(即紧接在后面的用花括号括起来的代码)。否则，执行else分支中的代码。

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

如果你代表了多种条件可以使用else if，这个允许你代表更多的颗粒，一个的条件语句复杂的逻辑，例子如下：

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

条件声明有益代表逻逻辑状态，但是你可能发现你你重复写了许多代码，例如上面的例子，你只需要在每个分之中简单的打印一下语句，避免重复，Kotlin官方提供了经验，比如下面这个写法例子：

|  |
| --- |
| 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) |

每个条件控制语句都隐式的在最后一行返回表达式结果，因此不需要return关键词，因为三个分支的结果类型都式String，这个if-else表达式结果类型也是String，在这个例子中，从if-else表达式的结果为answerString赋了一个初始值，类型推断器能够省略answerString的显式类型声明，但是为了看起来更清晰一点我们通常最好包含它。

注意：Kotlin不支持三元运算符，而是支持条件表达式。

随着你的条件表达式复杂度的增加，你可能需要考虑你的if-else表达式用when表达式代替，例子如下：

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

一个when表达式可以代表一种情况，一个箭头（->），一个结果，如果这个条件箭头左边结果是true，然后这个箭头右边就会返回结果。注意，从分支不会在往下执行了。这个when表达式代码在功能上和上一个例子的if-else表达式等价，但是更方便阅读。

Kotlin的条件语句突出了一个强大的特征-智能铸造（smart casting），与其使用安全调用操作符又或者是非空检查符来进行可空变量的检查，你还不如使用条件控制语句来检查你的变量是否保存了空。例子如下：

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

在条件分支中，languageName可以被视为不能为空变量，Kotlin足够智能，能够识别出条件分支执行这个分支languageName不能保存空的值，所以你不需要将languageName视为空值。这个智能铸造（smart casting）会进行非空检查工作，类型检查工作，或者满足合同的任何条件。

### 函数

可以将一个或多个表达式分组到一个函数中。与每次需要一个结果时重复相同的一系列表达式不同，您可以将表达式包装在一个函数中并调用该函数。

声明一个函数，使用fun这个关键词在这个函数名前面。下一步，定义你函数输入类型，如果有，则声明它返回的输出类型。函数体是定义函数被调用时调用的表达式的地方。

基于前面的例子，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。它不接受任何输入。它输出一个String类型的结果。要调用一个函数，请使用它的名称，后跟调用操作符(())。在下面的示例中，用generateAnswerString()的结果初始化answerString变量。

|  |
| --- |
| val answerString = generateAnswerString() |

函数可以接受参数输入，例子如下：

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

什么是声明函数，你能指定任何数量参数类型。在上面的例子中，generateAnswerString()接受一个名为countThreshold的Int类型的参数，在函数内，你能通过名称来调用参数。

当调用这个函数的时候，你必须在函数调用的括号中包含一个实参:

|  |
| --- |
| val answerString = generateAnswerString(42) |

#### 简约的函数声明

generateAnswerString()是一个及其简洁的函数，这个函数声明了一个变量，然后立即返回一个结果。当函数返回单个表达式的结果时，你可以跳过声明局部变量，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."  } } |

你能够用赋值运算符来代替return关键词：

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

#### 匿名函数

并不是每一个函数都需要名称，一些函数能直接通过它的输入和输出来直接调用。这些函数被成为匿名函数。你能保持这些匿名函数的引用，使用这些引用以后能调用这些匿名函数。还可以在应用周围传递引用。与其他引用类型一样。

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

与命名函数一样，匿名函数能包含任意数量的表达式，这个匿名函数返回的结果就是最后一行表达式。

在上面的例子中，stringLengthFunc包含了一个函数的引用，该函数接受一个字符串作

输入，返回输入字符串的长度类型是Int。因为如此，这个函数类型表示输入一个String

回一个Int。这个代码没用调用任何函数，然后，要检索这个函数的执行结果，你必须像

用命名函数一样去调用它，你必须提供一个String字符串在你调用stringLengthFunc函

的时候。例子如下：

|  |
| --- |
| val stringLengthFunc: (String) -> Int = { input ->  input.length } val stringLength: Int = stringLengthFunc("Android") |

#### 高阶函数

一个函数能被另一个函数作为参数，函数能被其他函数作为参数调用高阶函数。在某种程度在这种模式对组件之间的通信是有益的，就像你可能在Java中使用回调接口一样。

这里有一个高阶函数例子：

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

stringMapper()函数接受一个字符串和一个函数，该函数从传递给它的字符串派生出Int值。你能传递一个满足参数的字符串来调用stringMapper()函数，即是接受一个String的输入值和输出一个Int值。例子如下：

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

如果这个匿名函数最后一个参数定义的是一个函数，你能够在函数外部传递一个插入来调用函数，例子如下：

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

匿名函数可以在整个Kotlin标准库中找到。更多信息请查看，[higher-order functions and lambdas](https://kotlinlang.org/docs/lambdas.html)

### 类

到目前为止提到的所有类型都内置在Kotlin编程语言中,如果你有需要你可以构建自己的自定义类，你能定义类的关键词，例子如下：

|  |
| --- |
| class Car |

#### 属性

类属性代表状态，一个属性是类水平变量包含类一个get方法，一个set方法，和一个后备场。自从车子需要轮子以后，你可以给Car类添加一个Wheel集合属性，例子如下：

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

注意这个Wheels是一个公告的val变量，这意味这Wheels能被Car类之外进入，以及它是不能被重新分配的。如果你想获取Car的实例，你必须首次调用构造函数。从这儿，你能访问任何你能访问的属性。

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

如果你想自定义你的Wheels，你能定义一个自定义的构造函数来指定你类属性的初始化：

|  |
| --- |
| class Car(val wheels: List<Wheel>) |

在上面这个例子中，这个类的构造函数赋予了一个List<Wheel>作为构造函数的参数，初始化这个类时你需要传递一个wheels属性。

#### 类函数和封装

类使用函数来建模行为，函数能够修改状态，帮助你暴露唯一的数据你希望暴露的数据，这种访问控制是更大的面向对象概念封装的一部分。

在下面的例子中，doorLock属性对Car类之外的任何东西都是私有的。要解锁汽车，你必须调用unlockDoor()函数传入一个有效的密钥，如下面的例子所示:

|  |
| --- |
| 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  } } |

如果您想自定义如何引用属性，您可以提供自定义getter和setter。例如，如果你想公开属性的getter，同时限制对其setter的访问，你可以将setter指定为private:

|  |
| --- |
| class Car(val wheels: List<Wheel>) {  private val doorLock: DoorLock = ...  var 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，您可以创建为所有类型的对象建模的类。

### 可操作性

Kotlin最重要的特性之一是它与Java的流畅互操作性,因为Kotlin代码编译器是在JVM字节码之下的。你的Kotlin代码被立即转换成Java代码和vice-versa。这意味着现有的Java库能直接调用Kotlin。此外，Android主要的api都是用Java写的，你能够直接用Kotiln去调用它们。

## Android中常见的Kotlin模式

本篇主题主要集中在Android开发时使用Kotlin最通常的方面。

### （工作和fragment ）Work with fragments

下面几节使用片段示例来强调Kotlin的一些最佳特性。

#### 继承

你能在Kotlin中使用class关键词来声明类，在下面的例子中，LoginFragment是Fragment的一个子类。你能使用 : 符号来表明子类和父类的继承关系：

|  |
| --- |
| class LoginFragment : Fragment() |

在这个声明的类中，LoginFragment代表调用类它父类Fragment的构造方法。在LoginFragment中你可以覆盖大量生命周期回调来响应你Fragment中状态更改。覆盖这个函数使用override关键词，例子如下：

|  |
| --- |
| override fun onCreateView(  inflater: LayoutInflater,  container: ViewGroup?,  savedInstanceState: Bundle? ): View? {  return inflater.inflate(R.layout.login\_fragment, container, false) } |

引用父类中的函数，使用use这个关键词，例子如下：

|  |
| --- |
| override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState) } |

#### 可空性和初始化

在之前的例子中，覆盖的方法中的一些参数的类型以问号作为后缀。这表明传递参数给这些参数可以为空。必须保证可空性安全。[handle their nullability safely](https://kotlinlang.org/docs/reference/null-safety.html)

在Kotlin中，你声明对象的时候必须声明对象的属性。这意味着在你获取一个类的实例的时候，你能直接引用访问任何允许访问的属性。Fragment中的视图（View）对象。然而，直到调用Fragment#onCreateView之前还没准备好膨胀。所以你需要一个延期执行View初始化的方法。

这个lateinit关键词是能延期初始化属性。当你使用lateinit的时候，你应该尽快初始化你的属性。

下面的例子演示了如何在onViewCreated中使用lateinit来分配视图对象:

|  |
| --- |
| class LoginFragment : Fragment() {  private lateinit var usernameEditText: EditText  private lateinit var passwordEditText: EditText  private lateinit var loginButton: Button  private lateinit var statusTextView: TextView  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState)  usernameEditText = view.findViewById(R.id.username\_edit\_text)  passwordEditText = view.findViewById(R.id.password\_edit\_text)  loginButton = view.findViewById(R.id.login\_button)  statusTextView = view.findViewById(R.id.status\_text\_view)  }  ... } |

注意:如果您在属性初始化之前访问它，Kotlin会抛出一个UninitializedPropertyAccessException。

#### SAM转换

你能在Android中引用这个OnClickListener来监听点击事件。Button对象包含来一个setOnClickListener函数可以获取到一个OnClickListener引用。

OnClickListener是一个抽象方法,onClick(),你必须引用它。因为setOnClickListener()总是获取一个OnClickListener作为参数，因为这些OnClickListener总是抽象方法。这个引用在Kotlin中能用匿名函数代表使用。

这个过程称为单个抽象方法转换，或SAM转换。[Single Abstract Method conversion](https://kotlinlang.org/docs/java-interop.html" \l "sam-conversions)

SAM转换能够让你的代码相当简洁，下面这个例子向你展示了如果使用SAM转换在Button中引用OnClickListener：

|  |
| --- |
| loginButton.setOnClickListener {  val authSuccessful: Boolean = viewModel.authenticate(  usernameEditText.text.toString(),  passwordEditText.text.toString()  )  if (authSuccessful) {  // Navigate to next screen  } else {  statusTextView.text = requireContext().getString(R.string.auth\_failed)  } } |

在你使用loginButton的点击事件时会通过setOnClickListener来执行里面的匿名函数代码。

#### 伴生对象（companion object）

伴生对象提供了一种机制，用于定义在概念上链接到类型但不绑定到特定对象的变量或函数。伴生对象类似于Java中的静态变量或静态方法。

在下面这个例子中，TAG是一个String常量。你不需要每个LoginFragment实例的字符串的唯一实例，所以你应该在一个同伴对象中定义它:

|  |
| --- |
| class LoginFragment : Fragment() {  ...  companion object {  private const val TAG = "LoginFragment"  } } |

你能文件顶部定义TAG，但是这个文件顶部可能还定义了大量的变量，函数，和类。伴生对象能够帮助你联系这些变量，函数和类定义，而不引用该类的任何特定实例。

#### 属性委托（property delegation）

属性初始化的时候，你可以重复一些Android的常见模式，比如访问片段中的ViewModel，避免执行重复代码，你能使用Kotlin委托语法。

|  |
| --- |
| private val viewModel: LoginViewModel by viewModels() |

属性委托提供了一个通用的实现，你可以在整个应用程序中重用它。Android KTX为您提供了一些属性委托。例如，viewModels检索当前片段范围内的ViewModel。

属性委托使用反射，这会增加一些性能开销。折衷的办法是使用简洁的语法，从而节省开发时间。

### 为空性（nullability）

Kotlin提供了严格的为空性规则维持着类型安全在你整个app中。在Kotlin不能引用默认为空的对象。这个变量再一次为空值，你必须声明这个变量是可为空的类型也就是添加后缀 ? 在这个类型声明后面。

在下面这个例子中，展示了非法的代码，name是一个String类型不能为空的变量：

|  |
| --- |
| val name: String = null |

允许变量为空，你必须使用可以为空的String类型，Strring?，例子如下：

|  |
| --- |
| val name: String? = null |

#### 互操作行（interoperability）

Kotlin有严格的规则来保证你的代码安全和更加简洁。这些规则降低了NullPointerException导致应用程序崩溃的可能性。它们减少了代码中需要进行的空检查的次数。

常常，当你在写android app的时候你必须调用non-kotlin代码，大部分android api都是用Java程序语言写的。

可空性是Java和Kotlin在行为上的一个关键区别。Java对可空性语法不那么严格。

在这个例子中，Account类有几个属性，你可以通过name调用String属性。Java没有Kotlin这样的可空行规则，相反，依赖于可选的nullability注释来显式声明是否可以分配空值。

因为Android框架主要是用Java编写的，所以在调用没有nullability注释的api时，可能会遇到这种情况。

#### 平台类型

如果使用Kotlin引用在Java Account类中定义未注释的name成员，在Kotlin中编译器不会知道是否String映射到String或String?。这种歧义是通过一个平台类型String!来表示的。

在Kotlin编译器中String!没有特殊含义，String!可以代表String或者String?，编译器允许您为任意一种类型赋值。请注意，如果将类型表示为String并分配空值，则有抛出NullPointerException的风险。

为了解决这个问题，你在Java中写代码时应该使用nullability注释，这些注释能够帮助Java和Kotlin开发者。

下面这个例子，下面是在Java中定义的Account类：

|  |
| --- |
| public class Account implements Parcelable {  public final String name;  public final String type;  private final @Nullable String accessId;  ... } |

成员变量之一accessId带有@Nullable注释，表明它可以保存空值。然后，Kotlin会将accessId视为一个String?。

要指出一个变量永远不能为空，请使用@NonNull注释:

|  |
| --- |
| public class Account implements Parcelable {  public final @NonNull String name;  ... } |

在这个方案中，在Kotlin中name是被认为一个不能为空的String。

Nullability注释包含在所有新的Android api和许多现有的Android api中。许多Java库都添加了nullability注释，以更好地支持Kotlin和Java开发人员。

#### 处理可空性（handling nullability）

如果您不确定Java类型，你应该认为它是空的。在这个例子中，在这个Account类name成员是没有注释的，您应该假定它是一个可空字符串。

如果希望调整名称，使其值不包括开头或结尾的空格，则可以使用Kotlin的trim函数。你能用几种不同的方法去安全修改一个String?。其中一种方法是使用非空断言操作符 , !! ,如下面这个例子：

|  |
| --- |
| val account = Account("name", "type") val accountName = account.name!!.trim() |

这个 !! 操作符对它左侧对任何内容都视为非空，因为这个原因，您将name视为一个非空字符串。如果这个结果左侧展示了一个空，那么你的app就会抛出一个NullPointerException异常。这个操作符既快速又简单，但是应该谨慎使用，因为它可以在代码中重新引入NullPointerException的实例。

一个安全选择是使用安全操作符，? ，例子如下：

|  |
| --- |
| val account = Account("name", "type") val accountName = account.name?.trim() |

使用safe-call操作符，如果name不为空，则name?.trim()的结果是一个没有前导或尾部空白的name值。如果name是空，然后这个name?.trime()结果也会是空。当你执行这个方法的时候你的app从不抛出空指针异常(NullPointerException)。