Kotlin Grammer

deepwaterooo

2021 年 12 月 24 日

目录

1 空安全

1.1 可空类型与非空类型

- Kotlin 的类型系统旨在消除来自代码空引用的危险，也称为《十亿美元的错误》。
- 许多编程语言（包括 Java）中最常见的陷阱之一，就是访问空引用的成员会导致空引用异常。在 Java 中，这等同于 NullPointerException 或简称 NPE。
- Kotlin 的类型系统旨在从我们的代码中消除 NullPointerException。NPE 的唯一可能的原因可能是：

  - 显式调用 throw NullPointerException()；
  - 使用了下文描述的!! 操作符；
  - 有些数据在初始化时不一致，例如当：
    * 传递一个在构造函数中出现的未初始化的 this 并用于其他地方（“泄漏 this”）；
    * 超类的构造函数调用一个开放成员，该成员在派生中类的实现使用了未初始化的状态；
  - Java 互操作：
    * 企图访问平台类型的 null 引用的成员；
    * 用于具有错误可空性的 Java 互操作的泛型类型，例如一段 Java 代码可能会向 Kotlin 的 MutableList<String> 中加入 null，这意味着应该使用 MutableList<String?> 来处理它；
    * 由外部 Java 代码引发的其他问题。

- 在 Kotlin 中，类型系统区分一个引用可以容纳 null （可空引用）还是不能容纳（非空引用）。例如，String 类型的常规变量不能容纳 null：

```
var a: String = "ab"
var a: String = "abc" // 默认情况下，常规初始化意味着非空
a = null // 编译错误
```

- 如果要允许为空，我们可以声明一个变量为可空字符串，写作 String?：

```
var b: String? = "abc" // 可以设置为空
b = null // ok
print(b)
```

- 现在，如果你调用 a 的方法或者访问它的属性，它保证不会导致 NPE，这样你就可以放心地使用：

```
val l = a.length
```

- 但是如果你想访问 b 的同一个属性，那么这是不安全的，并且编译器会报告一个错误：

```
val l = b.length // 错误：变量“b”可能为空
```

- 但是我们还是需要访问该属性，对吧？有几种方式可以做到。

1.2 在条件中检测 null

- 首先，你可以显式检测 b 是否为 null，并分别处理两种可能：

```
val l = if (b != null) b.length else -1
```

- 编译器会跟踪所执行检测的信息，并允许你在 if 内部调用 length。同时，也支持更复杂（更智能）的条件：

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
// String of length 6
```

- 请注意，这只适用于 b 是不可变的情况（即在检测和使用之间没有修改过的局部变量，或者不可覆盖并且有幕后字段的 val 成员），因为否则可能会发生在检测之后 b 又变为 null 的情。况

1.3 安全的调用

- 你的第二个选择是安全调用操作符，写作?.：

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // 无需安全调用
```

- 如果 b 非空，就返回 b.length，否则返回 null，这个表达式的类型是 Int?。
- 安全调用在链式调用中很有用。例如，如果一个员工 Bob 可能会（或者不会）分配给一个部门，并且可能有另外一个员工是该部门的负责人，那么获取 Bob 所在部门负责人（如果有的话）的名字，我们写作：

```
bob?.department?.head?.name
```

- 如果任意一个属性（环节）为空，这个链式调用就会返回 null。
- 如果要只对非空值执行某个操作，安全调用操作符可以与 let 一起使用：

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // 输出 Kotlin 并忽略 null
}
```

- 安全调用也可以出现在赋值的左侧。这样，如果调用链中的任何一个接收者为空都会跳过赋值，而右侧的表达式根本不会求值：

```
// 如果 `person` 或者 `person.department` 其中之一为空，都不会调用该函数：
person?.department?.head = managersPool.getManager()
```

1.4 Elvis 操作符

- 当我们有一个可空的引用 b 时，我们可以说“如果 b 非空，我使用它；否则使用某个非空的值”：

```
val l: Int = if (b != null) b.length else -1
val l: Int = if (b != null) b.length else -1
```

- 除了完整的 if-表达式，这还可以通过 Elvis 操作符表达，写作?:：

```
val l = b?.length ?: -1
val l = b?.length ?: -1
```

- 如果?: 左侧表达式非空，elvis 操作符就返回其左侧表达式，否则返回右侧表达式。请注意，当且仅当左侧为空时，才会对右侧表达式求值。
- 请注意，因为 throw 和 return 在 Kotlin 中都是表达式，所以它们也可以用在 elvis 操作符右侧。这可能会非常方便，例如，检测函数参数：

```
fun foo(node: Node): String? { // <<<==== 返回值可为空
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ……
}
```

1.5 !! 操作符：非空断言运算符（!!）

- 第三种选择是为 NPE 爱好者准备的：非空断言运算符（!!）将任何值转换为非空类型，若该值为空则抛出异常。我们可以写 b!!，这会返回一个非空的 b 值（例如：在我们例子中的 String）或者如果 b 为空，就会抛出一个 NPE 异常：

```
val l = b!!.length
```

- 因此，如果你想要一个 NPE，你可以得到它，但是你必须显式要求它，否则它不会不期而至。

1.6 安全的类型转换

如果对象不是目标类型，那么常规类型转换可能会导致 ClassCastException。另一个选择是使用安全的类型转换，如果尝试转换不成功则返回 null：

```
val aInt: Int? = a as? Int
```

1.7 可空类型的集合

- 如果你有一个可空类型元素的集合，并且想要过滤非空元素，你可以使用 filterNotNull 来实现：

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

2 Kotlin 中常量和静态方法

2.1 常量

2.1.1 Java 中：

```
class StaticDemoActivity {
     public static final String LOAN_TYPE = "loanType";
     public static final String LOAN_TITLE = "loanTitle";
}
```

2.1.2 Kotlin 中：

```
class StaticDemoActivity {
    companion object {
        val LOAN_TYPE = "loanType"
        val  LOAN_TITLE = "loanTitle"
    }
}
class StaticDemoActivity {
    companion object StaticParams{
        val  LOAN_TYPE = "loanType"
        val  LOAN_TITLE = "loanTitle"
    }
}
class StaticDemoActivity {
    companion object {
        const val LOAN_TYPE = "loanType"
        const val LOAN_TITLE = "loanTitle"
    }
}
```

- 注：const 关键字用来修饰常量，且只能修饰 val，不能修饰 var, companion object 的名字可以省略，可以使用 Companion 来指代

2.1.3 引用常量（这里的引用只针对于 **java** 引用 **kotlin** 代码）

- TestEntity 类引用 StaticDemoActivity 中的常量

```
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.Companion.getLOAN_TITLE();
    }
}
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.StaticParams.getLOAN_TITLE();
    }
}
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.LOAN_TITLE;
        String type= StaticDemoActivity.LOAN_TYPE;
    }
}
```

2.2 静态方法

- Java 代码：

```
class StaticDemoActivity {
    public static void test(){
    }
}
```

- Kotlin 中：

```
    class StaticDemoActivity {
        companion object {
            fun test(){
            }
        }
    }
    class StaticDemoActivity {
        companion object StaticParams{
            fun test() {
            }
        }
    }
```

2.2.1 引用静态方法（这里的引用只针对于 **java** 引用 **kotlin** 代码）

- TestEntity 类引用 StaticDemoActivity 中的静态方法

```
class TestEntity {
    public TestEntity () {
        StaticDemoActivity.Companion.test();
    }
}
class TestEntity {
    public TestEntity () {
        StaticDemoActivity.StaticParams.test();
    }
}
```

- companion object {} 中用来修饰静态常量，或者静态方法，单例等等

3 Kotlin 中的 object 与 companion object 的区别

- 区别：
  - companion object 类中只能一个声明周期跟类同步，Only one companion object is allowed per class
  - objcet 没有限制声明更多用在对象声明，对象表达式使用。
    - 可以声明在类里也可以声明在顶级包下 top-level declaration
- 使用原则：
  - 如果想写工具类的功能，直接创建文件，写 top-level「顶层」函数。（声明在包下）
  - 如果需要继承别的类或者实现接口，就用 object 或 companion object。

3.1 一、object 关键字

- object 关键字可以表达两种含义：一种是对象表达式, 另一种是对象声明。

3.1.1 1、对象表达式

- 继承一个匿名对象

```
val textView = findViewById<TextView>(R.id.tv)
textView.setOnClickListener(object : OnClickListener {
        override fun onClick(p0: View?) {
            Toast.makeText(this@TestActivity, " 点击事件生效", Toast.LENGTH_LONG)
        }

})
```

- 上面代码其实就是我们经常要给 view 设置的点击事件，OnClickListener 事件是一个匿名类的对象，用 object 来修饰。

3.1.2 2、对象声明

- 用 object 修饰的类为静态类，里面的方法和变量都为静态的。

1. 2.1 直接声明类

```
object DemoManager {
    private val TAG = "DemoManager"
    fun a() {
        Log.e(TAG," 此时 object 表示 声明静态内部类")
    }
}
```

2. 2.2 声明静态内部类
   - 类内部的对象声明，没有被 inner 修饰的内部类都是静态的

```
class DemoManager{
    object MyObject {
        fun a() {
            Log.e(TAG," 此时 object 表示 直接声明类")
        }
    }
}
```

- 如果需要调用 a() 方法
- kotlin 中调用

```
fun init() {
    MyObject.a()
}
```

- java 中调用

```
MyObject.INSTANCE.a();
```

3.2 二、companion object

- companion object 修饰为伴生对象, 伴生对象在类中只能存在一个，类似于 java 中的静态方法 Java 中使用类访问静态成员，静态方法。

```
companion object {
    private val TAG = "DemoManager"
    fun b() {
        Log.e(TAG," 此时 companion objec t 表示 伴生对象")
    }
}
```

- kotlin 中调用

```
fun init(){
   b()
}
```

- java 中调用

```
DemoManager.Companion.b();
```

- companion object 相关的内容可以查阅 Kotlin 中常量和静态方法这篇文章，在这里不多在具体描述。

3.3 三、在 companion object 中如何调用外部的成员变量

3.3.1 3.1 为什么 companion object 中调用不到外部成员变量

```
class DemoManager {
    private val MY_TAG = "DemoManager"
    fun init(){
       b()
    }
    companion object {
        fun b() {
            Log.e(MY_TAG," 此时 companion objec t 表示 伴生对象")
        }
    }
}
```

- 在上面代码中 MY_{TAG} 是不会被调用到的。
- 原理很简单：

- 在 java 中我们写一个静态方法，如果需要调用成员变量，是无法调用到的

```
private String TAG = "MainActivity";
public static void init(){
    Log.e(TAG,"init() ");
}
```

- 只有将 TAG 修改为静态成员变量才能调用到

```
private static String TAG = "MainActivity";
public static void init(){
    Log.e(TAG,"init() ");
}
```

- 由此可以看出来，java 中静态方法调用成员变量，要求成员变量必须是静态的，在 kotlin 中也是一样，所以当 companion object 中调用非静态的成员变量也是调用不到的。

3.3.2 3.2 怎样解决才能调用到呢？

```
companion object {
    private val MY_TAG = "DemoManager"
    fun b() {
        Log.e(MY_TAG," 此时 companion objec t 表示 伴生对象")
    }
}
```

- 将所引用的成员变量也修饰静态的，这样就可以引用到了。
- 再透彻一点,偏源码与反编译一点儿的:https://www.cnblogs.com/webor2006/p/11210181.html
- https://juejin.cn/post/6844903816446345224

4 伴生对象：A few facts about Companion objects（写得比较深和透彻一点儿）

- Kotlin 给 Java 开发者带来最大改变之一就是废弃了 static 修饰符。与 Java 不同的是在 Kotlin 的类中不允许你声明静态成员或方法。相反，你必须向类中添加 Companion 对象来包装这些静态引用: 差异看起来似乎很小，但是它有一些明显的不同。
- 首先，companion 伴生对象是个实际对象的单例实例。你实际上可以在你的类中声明一个单例，并且可以像 companion 伴生对象那样去使用它。这就意味着在实际开发中，你不仅仅只能使用一个静态对象来管理你所有的静态属性！companion 这个关键字实际上只是一个快捷方式，允许你通过类名访问该对象的内容 (如果伴生对象存在一个特定的类中，并且只是用到其中的方法或属性名称，那么伴生对象的类名可以省略不写)。就编译而言，下面的 testCompanion () 方法中的三行都是有效的语句。

```
class TopLevelClass {
    companion object {
        fun doSomeStuff() {
        }
    }
    object FakeCompanion {
        fun doOtherStuff() {
        }
    }
```

```
}
fun testCompanion() {
    TopLevelClass.doSomeStuff()
    TopLevelClass.Companion.doSomeStuff()
    TopLevelClass.FakeCompanion.doOtherStuff()
}
```

- 为了兼容的公平性，companion 关键字还提供了更多选项，尤其是与 Java 互操作性相关选项。果您尝试在 Java 类中编写相同的测试代码，调用方式可能会略有不同：

```
public void testCompanion() {
    TopLevelClass.Companion.doSomeStuff();
    TopLevelClass.FakeCompanion.INSTANCE.doOtherStuff();
}
```

- 区别在于: Companion 作为 Java 代码中静态成员开放 (实际上它是一个对象实例，但是由于它的名称是以大写的 C 开头，所以有点存在误导性)，而 FakeCompanion 引用了我们的第二个单例对象的类名。在第二个方法调用中，我们需要使用它的 INSTANCE 属性来实际访问 Java 中的实例 (你可以打开 IntelliJ IDEA 或 AndroidStudio 中的"Show Kotlin Bytecode" 菜单栏，并点击里面"Decompile" 按钮来查看反编译后对应的 Java 代码)

- 在这两种情况下 (不管是 Kotlin 还是 Java)，使用伴生对象 Companion 类比 FakeCompanion 类那种调用语法更加简短。此外，由于 Kotlin 提供一些注解，可以让编译器生成一些简短的调用方式，以便于在 Java 代码中依然可以像在 Kotlin 中那样简短形式调用。

- @JvmField 注解，例如告诉编译器不要生成 getter 和 setter, 而是生成 Java 中成员。在伴生对象的作用域内使用该注解标记某个成员，它产生的副作用是标记这个成员不在伴生对象内部作用域，而是作为一个 Java 最外层类的静态成员存在。从 Kotlin 的角度来看，这没有什么太大区别，但是如果你看一下反编译的字节代码，你就会注意到伴生对象以及他的成员都声明和最外层类的静态成员处于同一级别。

- 另一个有用的注解 @JvmStatic. 这个注解允许你调用伴生对象中声明的方法就像是调用外层的类的静态方法一样。但是需要注意的是：在这种情况下，方法不会和上面的成员一样移出伴生对象的内部作用域。因为编译器只是向外层类中添加一个额外的静态方法，然后在该方法内部又委托给伴生对象。

- 一起来看一下这个简单的 Kotlin 类例子:

```
class MyClass {
    companion object {
        @JvmStatic
        fun aStaticFunction() {}
    }
}
```

- 这是相应编译后的 Java 简化版代码:

```
public class MyClass {
    public static final MyClass.Companion Companion = new MyClass.Companion();
    fun aStaticFunction() {//外层类中添加一个额外的静态方法
        Companion.aStaticFunction();//方法内部又委托给伴生对象的 aStaticFunction 方法
    }
    public static final class Companion {
         public final void aStaticFunction() {}
    }
}
```

- 这里存在一个非常细微的差别，但在某些特殊的情况下可能会出问题。例如，考虑一下 Dagger 中的 module(模块)。当定义一个 Dagger 模块时，你可以使用静态方法去提升性能，但是如果你选择这样做，如果您的模块包含静态方法以外的任何内容，则编译将失败。由于 Kotlin 在类中既包含静态方法，也保留了静态伴生对象，因此无法以这种方式编写仅仅包含静态方法的 Kotlin 类。
- 但是不要那么快放弃! 这并不意味着你不能这样做，只是它需要一个稍微不同的处理方式：在这种特殊的情况下，你可以使用 Kotlin 单例 (使用 object 对象表达式而不是 class 类) 替换含有静态方法的 Java 类并在每个方法上使用 @JvmStatic 注解。如下例所示：在这种情况下，生成的字节代码不再显示任何伴生对象，静态方法会附加到类中。

```
@Module
object MyModule {
    @Provides
    @Singleton
    @JvmStatic
    fun provideSomething(anObject: MyObject): MyInterface {
        return myObject
    }
}
```

- 这又让你再一次明白了伴生对象仅仅是单例对象的一个特例。但它至少表明与许多人的认知是相反的，你不一定需要一个伴生对象来维护静态方法或静态变量。你甚至根本不需要一个对象来维护，只要考虑顶层函数或常量：它们将作为静态成员被包含在一个自动生成的类中 (默认情况下，例如 MyFileKt 会作为 MyFile.kt 文件生成的类名，一般生成类名以 Kt 为后缀结尾)
- 我们有点偏离这篇文章的主题了，所以让我们继续回到伴生对象上来。现在你已经了解了伴生对象实质就是对象，也应该意识到它开放了更多的可能性，例如继承和多态。
- 这意味着你的伴生对象并不是没有类型或父类的匿名对象。它不仅可以拥有父类，而且它甚至可以实现接口以及含有对象名。它不需要被称为 companion。这就是为什么你可以这样写一个 Parcelable 类：

```
class ParcelableClass() : Parcelable {
    constructor(parcel: Parcel) : this()
    override fun writeToParcel(parcel: Parcel, flags: Int) {}
    override fun describeContents() = 0
    companion object CREATOR : Parcelable.Creator<ParcelableClass> {
        override fun createFromParcel(parcel: Parcel): ParcelableClass = ParcelableClass(parcel)
        override fun newArray(size: Int): Array<ParcelableClass?> = arrayOfNulls(size)
    }
}
```

- 这里, 伴生对象名为 CREATOR，它实现了 Android 中的 Parcelable.Creator 接口，允许遵守 Parcelable 约定，同时保持比使用 @JvmField 注释在伴随对象内添加 Creator 对象更直观。Kotlin 中引入了 @Parcelize 注解，以便于可以获得所有样板代码，但是在这不是重点...
- 为了使它变得更简洁，如果你的伴生对象可以实现接口，它甚至可以使用 Kotlin 中的代理来执行此操作：

```
class MyObject {
    companion object : Runnable by MyRunnable()
}
```

- 这将允许您同时向多个对象中添加静态方法！请注意，伴生对象在这种情况下甚至不需要作用域体，因为它是由代理提供的。
- 最后但同样重要的是，你可以为伴生对象定义扩展函数! 这就意味着你可以在现有的类中添加静态方法或静态属性，如下例所示：

```
class MyObject {
    companion object
    fun useCompanionExtension() {
        someExtension()
    }
}
fun MyObject.Companion.someExtension() {}//定义扩展函数
```

- 这样做有什么意义？我真的不知道。虽然 Marcin Moskala 建议使用此操作将静态工厂方法以 Companion 的扩展函数的形式添加到类中。
- 总而言之，伴生对象不仅仅是为了给缺少 static 修饰符的使用场景提供解决方案:
- 它们是真正的 Kotlin 对象，包括名称和类型，以及一些额外的功能。
- 他们甚至可以不用于仅仅为了提供静态成员或方法场景。可以有更多其他选择，比如他们可以用作单例对象或替代顶层函数的功能。
- 与大多数场景一样，Kotlin 意味着在你设计过程需要有一点点转变，但与 Java 相比，它并没有真正限制你的选择。如果有的话，也会通过提供一些新的、更简洁的方式让你去使用它。
- https://cloud.tencent.com/developer/article/1381584 这个某天上午的时候再读一遍就可以了
- http://www.4k8k.xyz/article/u013064109/89199478 这个没读，扫一眼