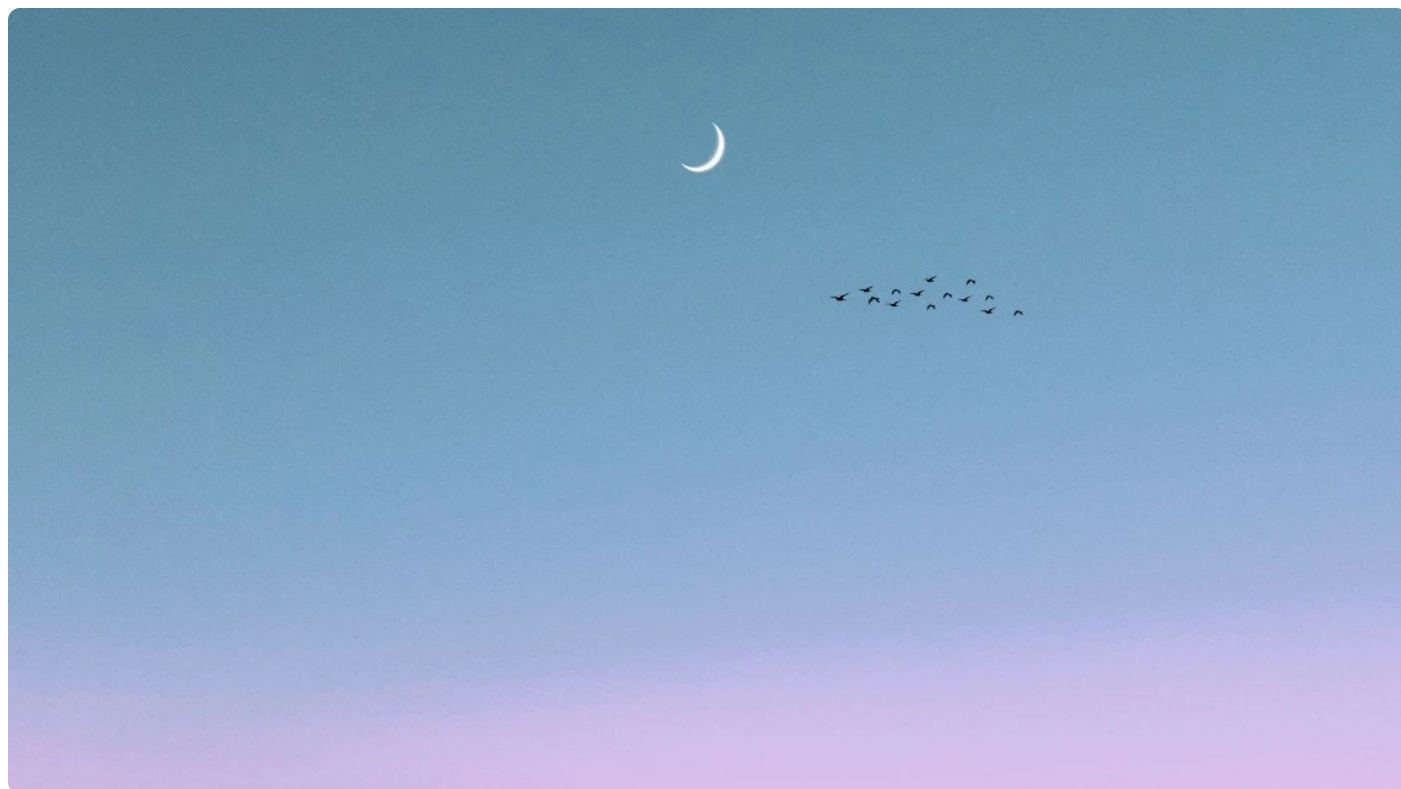


# 答疑（一）| Java和Kotlin到底谁好谁坏？

2022-03-25 朱涛

《朱涛·Kotlin编程第一课》

[课程介绍 >](#)



讲述：朱涛

时长 09:11 大小 8.42M



你好，我是朱涛。

由于咱们课程的设计理念是简单易懂、贴近实际工作，所以我在课程内容的讲述上也会有一些侧重点，进而也会忽略一些细枝末节的知识点。不过，我看到很多同学都在留言区分享了自己的见解，算是对课程内容进行了很好的补充，这里给同学们点个赞，感谢你的仔细思考和认真学习。

另外，我看到不少同学提出的很多问题也都非常有价值，有些问题非常有深度，有些问题非常有实用性，有些问题则非常有代表性，这些问题也值得我们再一起探讨下。因此，这一次，我们来一次集中答疑。

## Java 和 Kotlin 到底谁好谁坏？

很多同学看完 [开篇词](#) 以后，可能会留下一种印象，就是貌似 Java 就是坏的，Kotlin 就是好的。但其实在我看来，语言之间是不存在明确的优劣之分的。“XX 是世界上最好的编程语言”这种说法，也是没有任何意义的。

不过，虽然语言之间没有优劣之分，但在特定场景下，还是会有更优选择的。比如说，站在 Android 开发的角度上看，Kotlin 就的确要比 Java 强很多；但如果换一个角度，服务端开发，Kotlin 的优势则并不明显，因为 Spring Boot 之类的框架对 Java 的支持已经足够好了；甚至，如果我们再换一个角度，站在性能、编译期耗时的视角上看，Kotlin 在某些情况下其实是略逊于 Java 的。

如果用发展的眼光来看待这个问题的话，其实这个问题根本不重要。Kotlin 是一门基于 JVM 的语言，它更像是站在了巨人的肩膀上。**Kotlin 的设计思路就是“扬长避短”**。Java 的优点，Kotlin 都可以拿过来；Java 的缺点，Kotlin 尽量都把它扔掉！这就是为什么很多人会说：Kotlin 是一门更好的 Java 语言（Better Java）。

在开篇词里，我曾经提到过 Java 的一些问题：语法表现力差、可读性差，难维护、易出错、并发难。而这并不是说 Java 有多么不好，我想表达的其实是这两点：

- **Java 太老了**。Java 为了自身的兼容性，它的语法很难发展和演进，这才导致它在几十年后的今天看起来“语法表现力差”。
- **不是 Java 变差了，而是 Kotlin 做得更好了**。因为 Kotlin 的理念就是扬长避短，因此，在 Java 特别容易出错的领域，Kotlin 做了足够多的优化，比如内部类默认静态，比如不允许隐式的类型转换，比如挂起函数优化异步逻辑，等等。

所以，Kotlin 一定就比 Java 好吗？结论是并不一定。但在大部分场景下，我会愿意选 Kotlin。

## Double 类型字面量

在 Java 当中，我们会习惯性使用“1F”代表 Float 类型，“1D”代表 Double 类型。但是这一行为在 Kotlin 当中其实会略有不同，而我发现，很多同学都会下意识地把 Java 当中的经验带入到 Kotlin（当然也包括我）。

```
2 val i = 1F // Float 类型
3 val j = 1.0 // Double 类型
4 val k = 1D // 报错!!
5
```

实际上，在 Kotlin 当中，要代表 Double 类型的字面量，我们只需要在数字末尾加上小数位即可。“1D”这种写法，在 Kotlin 当中是不被支持的，我们需要特别注意一下。

## 逆序区间

在🔗第 1 讲里，我曾提到过：如果我们想要逆序迭代一个区间，不能使用“6...0”这种写法，因为这种写法的区间要求是：右边的数字大于等于左边的数字。

 复制代码

```
1 // 代码段2
2
3 fun main() {
4     for (i in 6..0) {
5         println(i) // 无法执行
6     }
7 }
```

在我们实际工作中，我们也许不会直接写出类似代码段 2 这样的逻辑，但是，当我们的区间范围变成变量以后，这个问题就没那么容易被发现了。比如我们可以看看下面这个例子：

 复制代码

```
1 // 代码段3
2
3 fun main() {
4     val start = calculateStart() // 6
5     val end = calculateEnd() // 0
6     for (i in start..end) {
7         println(i)
8     }
9 }
```

在这段代码中，如果 end 小于 start，我们就很难通过读代码发现问题了。所以在实际的开发工作中，我们其实应该慎重使用“start...end”的写法。如果我们不管是正序还是逆序都需要迭代的话，这时候，我们可以考虑封装一个全局的顶层函数：

```

1 // 代码段4
2
3 fun main() {
4     fun calculateStart(): Int = 6
5     fun calculateEnd(): Int = 0
6
7     val start = calculateStart()
8     val end = calculateEnd()
9     for (i in fromTo(start, end)) {
10         println(i) // end 小于start, 无法执行
11     }
12 }
13
14 fun fromTo(start: Int, end: Int) =
15     if (start <= end) start..end else start downTo end

```

在上面的 `fromTo()` 当中，我们对区间的边界进行了简单的判断，如果左边界小于右边界，我们就使用逆序的方式迭代。

## 密封类优势

在 [第 2 讲](#) 中，有不少同学觉得密封类不是特别好理解。在课程里，我们是拿密封类与枚举类进行对比来说明讲解的。我们知道，**所谓枚举，就是一组有限数量的值**。枚举的使用场景往往是某种事物的某些状态，比如，电视机有开关的状态，人类有女性和男性，等等。在 Kotlin 当中，同一个枚举，在内存当中是同一份引用。

```

1 enum class Human {
2     MAN, WOMAN
3 }
4
5 fun main() {
6     println(Human.MAN == Human.MAN)
7     println(Human.MAN === Human.MAN)
8 }
9
10 输出
11 true
12 true

```

那么**密封类**，其实是对枚举的一种补充。枚举类能做的事情，密封类也能做到：

```

1 sealed class Human {
2     object MAN: Human()
3     object WOMAN: Human()
4 }
5
6 fun main() {
7     println(Human.MAN == Human.MAN)
8     println(Human.WOMAN === Human.WOMAN)
9 }
10
11 输出
12 true
13 true

```

所以，密封类，也算是用了枚举的思想。但它跟枚举不一样的地方是：**同一个父类的所有子类**。举个例子，我们在 IM 消息当中，就可以定义一个 **BaseMsg**，然后剩下的就是具体的消息子类型，比如文字消息 **TextMsg**、图片消息 **ImageMsg**、视频消息 **VideoMsg**，这些子类消息的种类肯定是有限的。

而密封类的好处就在于，对于每一种消息类型，它们都可以携带各自的数据。

```

1 // 代码段5
2
3 sealed class BaseMsg {
4     //          密封类可以携带数据
5     //          ↓
6     data class TextMsg(val text: String) : BaseMsg()
7     data class ImageMsg(val url: String) : BaseMsg()
8     data class VideoMsg(val url: String) : BaseMsg()
9 }

```

所以我们可以说：**密封类，就是一组有限数量的子类**。针对这里的子类，我们可以让它们创建不同的对象，这一点是枚举类无法做到的。

那么，**使用密封类的第一个优势**，就是如果我们哪天扩充了密封类的子类数量，所有密封类的使用处都会智能检测到，并且给出报错：

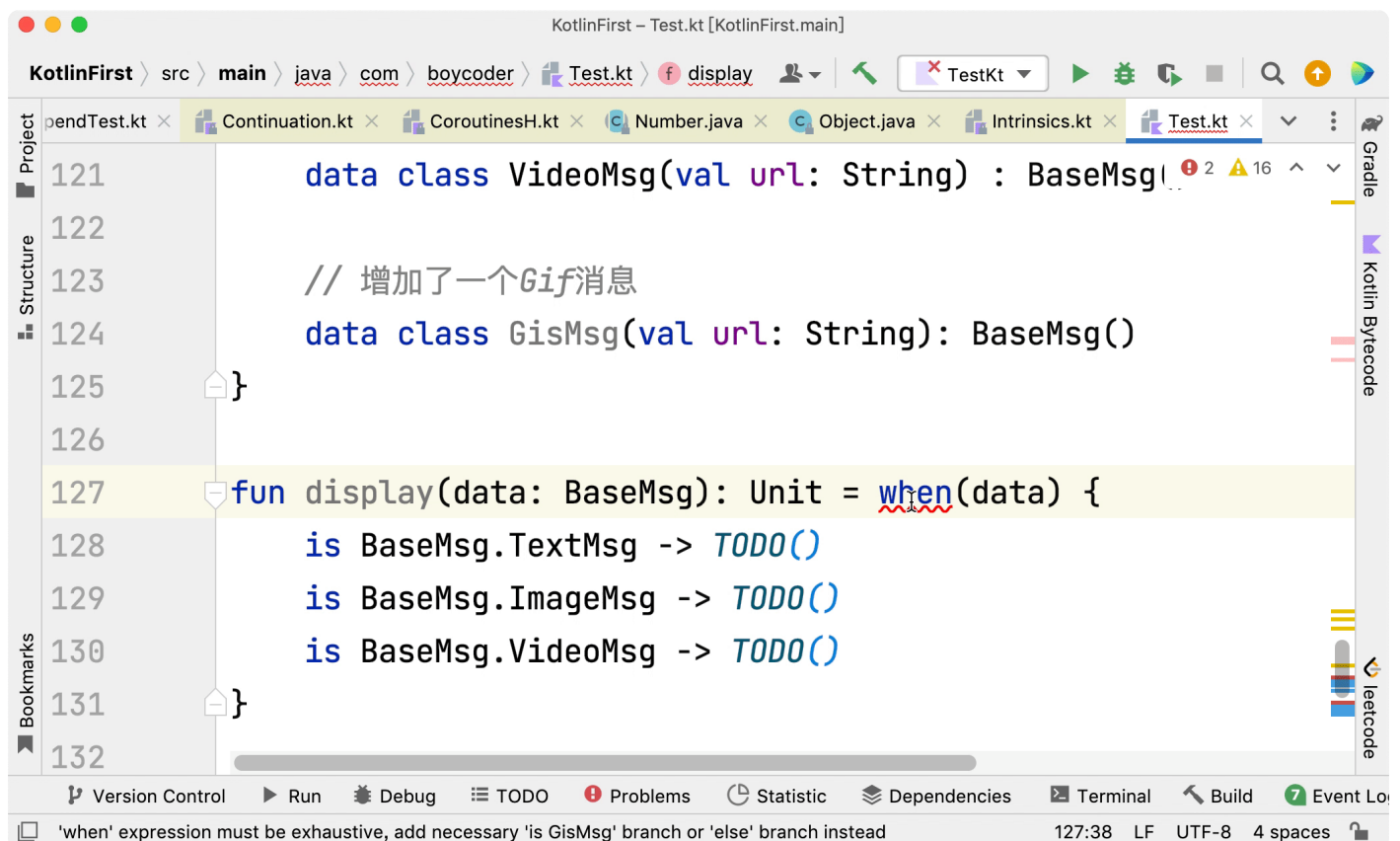
```

1 // 代码段6
2
3
4 sealed class BaseMsg {
5     data class TextMsg(val text: String) : BaseMsg()
6     data class ImageMsg(val url: String) : BaseMsg()
7     data class VideoMsg(val url: String) : BaseMsg()
8
9     // 增加了一个Gif消息
10    data class GisMsg(val url: String): BaseMsg()
11 }
12
13 // 报错!!
14 fun display(data: BaseMsg): Unit = when(data) {
15     is BaseMsg.TextMsg -> TODO()
16     is BaseMsg.ImageMsg -> TODO()
17     is BaseMsg.VideoMsg -> TODO()
18 }

```

上面的代码会报错，因为 `BaseMsg` 已经有 4 种子类型了，而 `when` 表达式当中只枚举了 3 种情况，所以它会报错。

**使用密封类的第二个优势**在于，当我们扩充了子类型以后，IDE 可以帮我们快速补充分支类型：



不过，还有一点需要特别注意，那就是 **else** 分支。一旦我们在枚举密封类的时候使用了 **else** 分支，那我们前面提到的两个密封类的优势就会不复存在！

 复制代码

```
1 sealed class BaseMsg {
2     data class TextMsg(val text: String) : BaseMsg()
3     data class ImageMsg(val url: String) : BaseMsg()
4     data class VideoMsg(val url: String) : BaseMsg()
5
6     // 增加了一个Gif消息
7     data class GifMsg(val url: String): BaseMsg()
8 }
9
10 // 不会报错
11 fun display(data: BaseMsg): Unit = when(data) {
12     is BaseMsg.TextMsg -> TODO()
13     is BaseMsg.ImageMsg -> TODO()
14     // 注意这里
15     else -> TODO()
16 }
```

请留意这里的 **display()** 方法，当我们只有三种消息类型的时候，我们可以在枚举了 **TextMsg**、**ImageMsg** 以后，使得 **else** 就代表 **VideoMsg**。不过，一旦后续增加了 **GifMsg** 消息类型，这里的逻辑就会出错。而且，在这种情况下，我们的编译器还不会提示报错！

因此，在我们使用枚举或者密封类的时候，一定要慎重使用 **else** 分支。

## 枚举类的 **valueOf()**

另外，在使用 Kotlin 枚举类的时候，还有一个坑需要我们特别注意。在 [🔗 第 4 讲](#) 实现的第一个版本的计算器里，我们使用了 **valueOf()** 尝试解析了操作符枚举类。而这只是理想状态下的代码，实际上，正确的方式应该使用 2.0 版本当中的方式。

 复制代码

```
1 val help = """
2 -----
3 使用说明：
4 1. 输入 1 + 1，按回车，即可使用计算器；
5 2. 注意：数字与符号之间要有空格；
6 3. 想要退出程序，请输入：exit
7 -----""".trimIndent()
8
9 fun main() {
```

```

10     while (true) {
11         println(help)
12
13         val input = readLine() ?: continue
14         if (input == "exit") exitProcess(0)
15
16         val inputList = input.split(" ")
17         val result = calculate(inputList)
18
19         if (result == null) {
20             println("输入格式不对")
21             continue
22         } else {
23             println("$input = $result")
24         }
25     }
26 }
27
28 private fun calculate(inputList: List<String>): Int? {
29     if (inputList.size != 3) return null
30
31     val left = inputList[0].toInt()
32     //                                注意这里
33     //                                ↓
34     val operation = Operation.valueOf(inputList[1])?: return null
35     val right = inputList[2].toInt()
36
37     return when (operation) {
38         Operation.ADD -> left + right
39         Operation.MINUS -> left - right
40         Operation.MULTI -> left * right
41         Operation.DIVI -> left / right
42     }
43 }
44
45 enum class Operation(val value: String) {
46     ADD("+"),
47     MINUS("-"),
48     MULTI("*"),
49     DIVI("/")
50 }

```

请留意上面的代码注释，这个 `valueOf()` 是无法正常工作的。Kotlin 为我们提供的这个方法，并不能为我们解析枚举类的 `value`。

 复制代码

```

1 fun main() {
2     // 报错

```



```

3     val wrong = Operation.valueOf("+")
4     // 正确
5     val right = Operation.valueOf("ADD")
6 }

```

出现这个问题的原因就在于，**Kotlin 提供的 valueOf() 就是用于解析“枚举变量名称”的。**

这是一个非常常见的使用误区，不得不说，Kotlin 在这个方法的命名上并不是很好，导致开发者十分容易用错。Kotlin 提供的 valueOf() 还不如说是 nameOf()。

而如果我们希望可以根据 value 解析出枚举的状态，我们就需要自己动手。最简单的办法，就是使用**伴生对象**。在这里，我们只需要将 2.0 版本当中的逻辑挪进去即可：

 复制代码

```

1 enum class Operation(val value: String) {
2     ADD("+"),
3     MINUS("-"),
4     MULTI("*"),
5     DIVI("/");
6
7     companion object {
8         fun realValueOf(value: String): Operation? {
9             values().forEach {
10                 if (value == it.value) {
11                     return it
12                 }
13             }
14             return null
15         }
16     }
17 }

```

对应的，在我们尝试解析操作符的时候，我们就不再使用 Kotlin 提供的 valueOf()，而是使用自定义的 realValueOf() 了：

 复制代码

```

1 val help = """
2 -----
3 使用说明：
4 1. 输入 1 + 1，按回车，即可使用计算器；
5 2. 注意：数字与符号之间要有空格；
6 3. 想要退出程序，请输入：exit
7 -----""".trimIndent()

```

```

8 fun main() {
9     while (true) {
10         println(help)
11
12         val input = readLine() ?: continue
13         if (input == "exit") exitProcess(0)
14
15         val inputList = input.split(" ")
16         val result = calculate(inputList)
17
18         if (result == null) {
19             println("输入格式不对")
20             continue
21         } else {
22             println("$input = $result")
23         }
24     }
25 }
26
27 private fun calculate(inputList: List<String>): Int? {
28     if (inputList.size != 3) return null
29
30     val left = inputList[0].toInt()
31     // 变化在这里
32     // ↓
33     val operation = Operation.valueOf(inputList[1])?: return null
34     val right = inputList[2].toInt()
35
36     return when (operation) {
37         Operation.ADD -> left + right
38         Operation.MINUS -> left - right
39         Operation.MULTI -> left * right
40         Operation.DIVI -> left / right
41     }
42 }
43

```

因此，对于枚举，我们在使用 `valueOf()` 的时候一定要足够小心！因为它解析的根本就不是 `value`，而是 `name`。

## 小结

在我看来，专栏是“作者说，读者听”的过程，而留言区则是“读者说，作者听”的过程。这两者结合在一起之后，我们才能形成一个更好的沟通闭环。今天的这节答疑课，就是我在倾听了你的声音后，给到你的回应。


所以，如果你在学习的过程中遇到了什么问题，请一定要提出来，我们一起交流和探讨，共同进步。

## 思考题

请问你在使用 Kotlin 的过程中，还遇到过哪些问题？请在留言区提出来，我们一起交流。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 实战：让 KtHttp 支持 Flow

下一篇 25 | 集合操作符：你也会“看完就忘”吗？

## 精选留言 (5)

 写留言



白乾涛

2022-03-26

Kotlin 舍弃了 Java 中很多容易出错的语法，那为什么引入了 in 却又不支持 6...0 这种写法呢？

这不就是另一个容易出错的语法吗？

作者回复: 我赞同，只能说 Kotlin 还不够好吧！

当然这不是一个特别大的问题，因为：6..0 这样的写法，它到底执行还是不执行，其实都是可以接受的，只是 Kotlin 选择了后者。



 1



7Promise

2022-03-25

kotlin中要考虑集合是否可变其实有时候也是麻烦的事情

作者回复: 是的。



👍 1



张国庆

2022-03-25

使用kotlin是不是包体积会比Java大

作者回复: 会稍微大一点点, 但不会很多。



👍 1



focus on

2022-03-28

大佬能多讲讲flow吗, 随着flow取代livedata, 而且Android上的StateFlow和SharedFlow不好理解

作者回复: 好的, 记下了, 请留意后续的加餐。



Paul Shan

2022-03-27

Java 和Kotlin很难直接比较, 因为这两个语言是诞生在不同年代。不过倒是可以从Kotlin的诞生看看两者的区别。Kotlin是Jetbrains公司开发的, Jetbrains是Java的重度使用者, 开发跨平台的IDE, 这个世界上比Jetbrains公司更擅长Java的公司, 怕是不多。Jetbrains选择研发一门新语言本身就说明, 现阶段Java不是Jetbrains的最优选择, Jetbrains估计受够了Java的短板, 所以才要在Java的基础上迭代。Kotlin在Java的基础上开发的, 所以更为简洁顺手。个人觉得将来Kotlin Multiplatform比Kotlin Backend成功的概率更大一些, 虽然Kotlin Backend技术上和Kotlin Android没什么差别。这主要是因为Jetbrains是一家精通UI开发的公司, 后端开发并非Jetbrains的强项。

Kotlin是Jetbrains俄罗斯团队研发的(Kotlin名字来自于圣彼得堡旁边的小岛), 俄乌战争开打以后, Jetbrains就无限期关停了在俄罗斯的研发和销售业务, 这给Kotlin Multiplatform等项目蒙上阴影。从Jetbrains的Channel上看, 战争开打以来, 视频更新明显减少。请问老师, 俄乌战争给Kotlin带来的影响会短期过去, 还是会成为长期挥之不去的阴影?

作者回复: 很棒的见解! Kotlin Multiplatform的潜力确实比后端要大很多。

关于俄乌战争对Kotlin的影响：Jetbrains的创始人由于自身立场，选择了制裁俄罗斯，停止对俄罗斯提供服务，也停止了俄罗斯的研发团队。不过，总的来说，Jetbrains是一个全球化的公司，我相信这个影响只是短期的。

