从 Java 到 Kotlin，再从 Kotlin 回归 Java

开源中国 2018-06-11

协作翻译

原文：From Java to Kotlin and Back Again
链接：https://allegro.tech/2018/05/From-Java-to-Kotlin-and-Back-Again.html
译者：边城, Tocy, kevinlinkai, 雪落无痕xdj

背景介绍

作者是 Allegro 的一名技术人员。Allegro 拥有超过 50 个开发团队。可以自由选择 他们的 PaaS 所支持的技术。他们主要使用 Java、Kotlin、Python 和 Golang 进行编码。本文中提出的观点来自作者的经验。

Kotlin 很流行，Kotlin 很时髦。Kotlin 为你提供了编译时 null-safety 和更少的 boilerplate。当然，它比 Java 更好，你应该切换到 Kotlin。等等，或者你不应该如此？在开始使用 Kotlin 编写之前，请阅读一个项目的故事。

尝试 Kotlin 后，我们正用 Java10 重写

我有我最喜欢的JVM语言集。Java的/main和Groovy的/test对我来说是组好的组合。2017年夏季，我的团队开始了一个新的微服务项目，我们就像往常一样谈论了语言和技术。

在Allegro有几个支持Kotlin的团队，而且我们也想尝试新的东西，所以我们决定试试Kotlin。由于Kotlin中没有Spock的替代品，我们决定继续在/test中使用Groovy（Spek没有Spock好用）。在2018年的冬天，每天与Kotlin相伴的几个月后，我们总结出了正反两面，并且得出Kotlin使我们的生产力下降的结论。我们开始用Java重写这个微服务。

这有几个原因：

- 名称遮蔽
- 类型推断
- 编译时空值安全
- 类字面量
- 相反的类型声明
- 伴生对象
- 集合文字面量
- Maybe？ 不
- 数据类
- 开放类
- 陡峭的学习曲线

名称遮掩

这是 Kotlin 让我感到最大惊喜的地方。看看这个函数：

```
fun inc(num : Int) {
    val num = 2
    if (num > 0) {
        val num = 3
    }
    println ("num: " + num)
}
```

当你调用 inc(1) 的时候会输出什么呢？在 Kotlin 中方法参数是一个值，所以你不能改变 num 参数。这是好的语言设计，因为你不应该改变方法的参数。但是你可以用相同的名称定义另一个变量，并按照你想要的方式初始化。现在，在这个方法级别的范围中你拥有两个叫做 num 的变量。当然，同一时间你只能访问其中一个num，所以 num 的值会改变。将军，无解了。

在 if 主体中，你可以添加另一个 num，这并不令人震惊（新的块级别作用域）。

好的，在 Kotlin 中，inc(1) 输出 2。但是在Java中，等效代码将无法通过编译。

```
void inc(int num) {
    int num = 2; //error: variable 'num' is already defined in the scope
    if (num > 0) {
        int num = 3; //error: variable 'num' is already defined in the scope
    }
    System.out.println ("num: " + num);
}
```

名称遮蔽不是 Kotlin 发明的。这在编程语言中着很常见。在 Java 中，我们习惯用方法参数来遮蔽类中的字段。

```
public class Shadow {
    int val;

    public Shadow(int val) {
        this.val = val;
    }
}
```

在 Kotlin 中，遮蔽有点过分了。当然，这是 Kotlin 团队的一个设计缺陷。IDEA 团队试图把每一个遮蔽变量都通过简洁的警告来向你展示，以此修复这个问题：Name shadowed。两个团队都在同一家公司工作，所以或许他们可以相互交流并在遮蔽问题上达成一致共识？我感觉 —— IDEA 是对的。我无法想象存在这种遮蔽了方法参数的有效用例。

类型推断

在 Kotlin 中，当你申明一个 var 或者 val 时，你通常让编译器从右边的表达式类型中猜测变量类型。我们将其称做局部变量类型推断，这对程序员来说是一个很大的改进。它允许我们在不影响静态类型检查的情况下简化代码。

例如，这段 Kotlin 代码：

```
var a = "10"
```

将由 Kotlin 编译器翻译成：

```
var a : String = "10"
```

它曾经是胜过Java的真正优点。我故意说曾经是，因为——有个好消息——Java10 已经有这个功能了，并且Java10现在已经可以使用了。

Java10 中的类型涂端：

```
var a = "10";
```

公平的说，我需要补充一点，Kotlin在这个领域仍然略胜一筹。你也可以在其他上下文中使用类型推断，例如，单行方法。

更多关于Java10 中的局部变量类型推断。

编译时空值安全

Null-safe类型是Kotlin的杀手级特征。这个想法很好。在Kotlin，类型是默认的非空值。如果您需要一个可空类型，您需要添加？符号，例如：

```
val a: String? = null      // ok

val b: String = null       // 编译错误
```

如果您在没有空检查的情况下使用可空变量，那么Kotlin将无法编译，例如：

```
println (a.length)          // compilation error
println (a?.length)         // fine, prints null
println (a?.length ?: 0)    // fine, prints 0
```

一旦你有了这两种类型，non-nullable T 和nullable T?，您可以忘记Java中最常见的异常——NullPointerException。真的吗?不幸的是，事情并不是那么简单。

当您的Kotlin代码必须与Java代码一起使用时，事情就变得很糟糕了（库是用Java编写的，所以我猜它经常发生）。然后，第三种类型就跳出来了——T!它被称为平台类型，它的意思是T或T?，或者如果我们想要精确，T!意味着具有未定义空值的T类型。这种奇怪的类型不能用Kotlin来表示，它只能从Java类型推断出来。T!会误导你，因为它放松了对空的限制，并禁用了Kotlin的空值安全限制。

看看下面的Java方法：

```
public class Utils {
    static String format(String text) {
        return text.isEmpty() ? null : text;
    }
}
```

现在，您想要从Kotlin调用format(string)。您应该使用哪种类型来使用这个Java方法的结果？好吧，你有三个选择。

第一种方法。你可以使用字符串，代码看起来很安全，但是会抛出空指针异常。

```
fun doSth(text: String) {
    val f: String = Utils.format(text)       // compiles but assignment can throw NPE at runtime
    println ("f.len : " + f.length)
}
```

你需要用增加判断来解决这个问题：

```
fun doSth(text: String) {
    val f: String = Utils.format(text) ?: ""  //
    println ("f.len : " + f.length)
}
```

第二种方法。您可以使用String?，然后你的程序就是空值安全的了。

```
fun doSth(text: String) {
    val f: String? = Utils.format(text)   // safe
    println ("f.len : " + f.length)       // compilation error, fine
    println ("f.len : " + f?.length)      // null-safe with ? operator
}
```

第三种方法。如果你让Kotlin做了令人难以置信的局部变量类型推断呢？

```
fun doSth(text: String) {
    val f = Utils.format(text)          // f type inferred as String!
    println ("f.len : " + f.length)     // compiles but can throw NPE at runtime
}
```

坏主意。这个Kotlin的代码看起来很安全，也可以编译通过，但是允许空值在你的代码中不受约束的游走，就像在Java中一样。

还有一个窍门，!!操作符。使用它来强制推断f类型为String类型：

```
fun doSth(text: String) {
    val f = Utils.format(text)!!        // throws NPE when format() returns null
    println ("f.len : " + f.length)
}
```

在我看来，Kotlin的类型系统中所有这些类似scala的东西!，？和!!，实在是太复杂了。为什么Kotlin从Java的T类型推断到T!而不是T?呢？似乎Java互操作性破坏了Kotlin的杀手特性——类型推断。看起来您应该显式地声明类型（如T?），以满足由Java方法填充的所有Kotlin变量。

类字面量

在使用Log4j或Gson之类的Java库时，类字面量是很常见的。

在Java中，我们用.class后缀来写类名：

```
Gson gson = new GsonBuilder().registerTypeAdapter(LocalDate.class, new LocalDateAdapter()).create();
```

在Groovy中，类字面量被简化为本质。你可以省略.class，不管它是Groovy还是Java类都没关系。

```
def gson = new GsonBuilder().registerTypeAdapter(LocalDate, new LocalDateAdapter()).create()
```

Kotlin区分了Kotlin和Java类，并为其准备了不同的语法形式：

```
val kotlinClass : KClass<LocalDate> = LocalDate::class
val javaClass : Class<LocalDate> = LocalDate::class.java
```

所以在Kotlin，你不得不写：

```
val gson = GsonBuilder().registerTypeAdapter(LocalDate::class.java, LocalDateAdapter()).create()
```

这真是丑爆了。

相反顺序的类型声明

在C系列编程语言中，有一个标准的声明类型的方式。即先写出类型，再写出声明为该类型的东西（变量、字段、方法等）。

在Java中如下表示：

```
int inc(int i) {
    return i + 1;
}
```

在Kotlin中则是相反顺序的表示：

```
fun inc(i: Int): Int {
    return i + 1
}
```

这让人觉得恼火，因为：

首先，你得书写或者阅读介于名称和类型之间那个讨厌的冒号。这个多余的字母到底起什么作用？为什么要把名称和类型分隔开？我不知道。不过我知道这会加大使用Kotlin的难度。

第二个问题。在阅读一个方法声明的时候，你最先想知道的应该是方法的名称和返回类型，然后才会去了解参数。

在 Kotlin 中，方法的返回类型远在行末，所以可能需要滚动屏幕来阅读：

```
private fun getMetricValue(kafkaTemplate : KafkaTemplate<String, ByteArray>, metricName : String) : Double {
    ...
}
```

另一种情况，如果参数是按分行的格式写出来的，你还得去寻找返回类型。要在下面这个方法定义中找到返回类型，你需要花多少时间?

```
@Bean
fun kafkaTemplate(
        @Value("\${interactions.kafka.bootstrap-servers-dc1}") bootstrapServersDc1: String,
        @Value("\${interactions.kafka.bootstrap-servers-dc2}") bootstrapServersDc2: String,
        cloudMetadata: CloudMetadata,
        @Value("\${interactions.kafka.batch-size}") batchSize: Int,
        @Value("\${interactions.kafka.linger-ms}") lingerMs: Int,
        metricRegistry : MetricRegistry
): KafkaTemplate<String, ByteArray> {

    val bootstrapServer = if (cloudMetadata.datacenter == "dc1") {
        bootstrapServersDc1
    }
    ...
}
```

关于相反顺序的第三个问题是限制了IDE的自动完成功能。在标准顺序中，因为是从类型开始，所以很容易找到类型。一旦确定了类型，IDE 就可以根据类型给出一些与之相关的变量名称作为建议。这样就可以快速输入变量名，不像这样：

```
MongoExperimentsRepository repository
```

即时在 Intellij 这么优秀的 IDE 中为 Kotlin 输入这样的变量名也十分不易。如果代码中存在很多 Repository，就很难在自动完成列表中找到匹配的那一个。换句话说，你得手工输入完整的变量名。

```
repository : MongoExperimentsRepository
```

伴生对象

一个 Java 程序员来到 Kotlin 阵营。

“嗨，Kotlin。我是新来的，有静态成员可用吗？”他问。

“没有。我是面向对象的，而静态成员不是面向对象的，” Kotlin回答。

“好吧，但我需要用于 MyClass 日志记录器，该怎么办？”

“没问题，可以使用伴生对象。”

“伴生对象是什么鬼？”

“它是与类绑定的一个单例对象。你可以把日志记录器放在伴生对象中，”Kotlin 如此解释。

“明白了。是这样吗？”

```
class MyClass {
    companion object {
        val logger = LoggerFactory.getLogger(MyClass::class.java)
    }
}
```

“对！“

“好麻烦的语法，”这个程序看起来有些疑惑，“不过还好，现在我可以像这样——MyClass.logger——调用日志记录了吗？就像在 Java 中使用静态成员那样？”

“嗯……是的，但是它不是静态成员！它只是一个对象。可以想像那是一个匿名内部类的单例实现。而实际上，这个类并不是匿名的，它的名字是 Companion，你可以省略这个名称。明白吗？这很简单。”

我很喜欢对象声明的概念——单例是种很有用的模式。从从语言中去掉静态成员就不太现实了。我们在Java中已经使用了若干年的静态日志记录器，这是非常经典的模式。因为它只是一个日志记录器，所以我们并不关心它是否是纯粹的面向对象。只要它起作用，而且不会造成损害就好。

有时候，我们必须使用静态成员。古老而友好的 public static void main() 仍然是启动 Java 应用的唯一方式。在没有Google的帮助下尝试着写出这个伴生对象。

```
class AppRunner {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            SpringApplication.run(AppRunner::class.java, *args)
        }
    }
}
```

集合字面量

在 Java 中初始化列表需要大量的模板代码：

```
import java.util.Arrays;
...

List<String> strings = Arrays.asList("Saab", "Volvo");
```

初始化 Map 更加繁琐，所以不少人使用 Guava：

```
import com.google.common.collect.ImmutableMap;
...

Map<String, String> string = ImmutableMap.of("firstName", "John", "lastName", "Doe");
```

我们仍然在等待 Java 产生新语法来简化集合和映射表的字面表达。这样的语法在很多语言中都自然而便捷。

JavaScript:

```
const list = ['Saab', 'Volvo']
const map = {'firstName': 'John', 'lastName' : 'Doe'}
```

Python:

```
list = ['Saab', 'Volvo']
map = {'firstName': 'John', 'lastName': 'Doe'}
```

Groovy:

```
def list = ['Saab', 'Volvo']
def map = ['firstName': 'John', 'lastName': 'Doe']
```

简单来说，简洁的集合字面量语法在现代编程语言中倍受期待，尤其是初始化集合的时候。Kotlin 提供了一系列的内建函数来代替集合字面量：listOf()、mutableListOf()、mapOf()、hashMapOf()，等

等。

Kotlin:

```
val list = listOf("Saab", "Volvo")
val map = mapOf("firstName" to "John", "lastName" to "Doe")
```

映射表中的键和值通过 to 运算符关联在一起，这很好，但是为什么不使用大家都熟悉的冒号(:)？真是令人失望！

Maybe? 不

函数式编程语言（比如 Haskell）没有空(null)。它们提供 Maybe Monad（如果你不清楚 Monad，请阅读这篇由 Tomasz Nurkiewicz 撰写文章）。

在很久以前，Scala 就将 Maybe 作为 Option 引入 JVM 世界，然后在 Java 8 中被采用，成为 Optional。现在 Optional 广泛应用于 API 边界，用于处理可能含空值的返回类型。

Kotlin 中并没有与 Optional 等价的东西。看起来你应该使用 Kotlin 的可空类型封装。我们来研究一下这个问题。

通常，在使用 Optional 时，你会先进行一系列空安全的转换，最后来处理空值。
比如在 Java 中：

```
public int parseAndInc(String number) {
    return Optional.ofNullable(number)
                   .map(Integer::parseInt)
                   .map(it -> it + 1)
                   .orElse(0);
}
```

在 Kotlin 中也没问题，使用 let 功能：

```
fun parseAndInc(number: String?): Int {
    return number.let { Integer.parseInt(it) }
                 .let { it -> it + 1 } ?: 0
}
```

可以吗？是的，但并不是这么简单。上面的代码可能会出错，从 parseInt() 中抛出 NPE。只有值存在的时候才能执行 Monad 风格的 map()，否则，null 只会简单的传递下去。这就是 map() 方便的原因。然后不幸的是，Kotlin 的 let 并不是这样工作的。它只是从左往右简单地执行调用，不在乎是否是空。

因此，要让这段代码对空安全，你必须在 let 前添加 ?:

```
fun parseAndInc(number: String?): Int {
    return number?.let { Integer.parseInt(it) }
                  ?.let { it -> it + 1 } ?: 0
}
```

现在，比如 Java 和 Kotlin 两个版本的可读性，你更喜欢哪一个？

想了解更多关于 Optional 的知识，可以阅读 Stephen Colebourne 的博客。

数据类

Data classes（数据类）是Kotlin在实现Value Objects（又名DTO）时为减少Java中不可避免的boilerplate的方法。

例如，在Kotlin中，你仅需编写Value Object的精髓：

```
data class User(val name: String, val age: Int)
```

同时Kotlin生成了equals(), hashCode(), toString()以及copy()的实现。

在实现简单的DTO时它非常有用。但请记住，数据类带有严重的局限性 - 它们是final的。你无法扩展Data类或将其抽象化。所以很可能，你不会在核心领域模型中使用它们。

这个局限性不是Kotlin的错。没有办法在不违反Liskov原则的情况下生成正确的基于值的equals()实现。这就是为什么Kotlin不允许Data类继承的原因。

开放(Open)类

Kotlin 中的类默认是封闭（final）的。如果你想从某个类扩展，你就必须为它的声明添加 open 修饰符。

继承语法就像这样：

```
open class Base

class Derived : Base()
```

Kotlin 把 extends 关键字改为：运算符，而这个运算符已经用于分隔变量名及其类型。这是想回归 C++ 语法吗？对于我来说，这让人感到困惑。

这里最具争议的话题是默认封闭。可能是因为 Java 程序员过度使用了继承，可能你应该在允许类被继承之前想想清楚。不过我们生活在框架和时代，而框架往往喜欢 AOP。Spring 使用一些库（cglib、jassist）为 Java Bean 生成动态代理。Hibernate 会扩展实体类来实现懒加载。

如果使用 Spring，你有两个选择。可以在所有 Bean 类前添加 open 声明（很繁琐），或者使用这个巧妙的编译插件：

```
buildscript {
    dependencies {
        classpath group: 'org.jetbrains.kotlin', name: 'kotlin-allopen', version: "$versions.kotlin"
    }
}
```

陡峭的学习曲线

如果你认为你可以快速学习Kotlin，因为你已经知道Java了 - 那么你错了。Kotlin会让你深陷其中。事实上，Kotlin的语法更接近Scala。这是赌上全部。你将不得不忘记Java并切换到完全不同的语言。

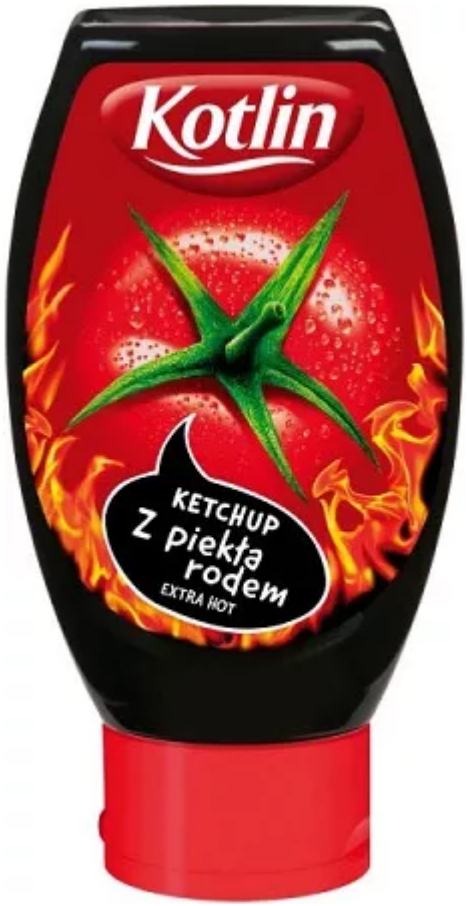
相反，学习Groovy是一个愉快的旅程。Groovy亲手引领你。Java代码是正确的Groovy代码，因此你可以从将.java文件扩展名更改为.groovy扩展名开始。每次你学习新的Groovy功能时，你都可以做出决定。你喜欢它还是喜欢用Java的方式？ 这简直棒极了。

最后的想法

学习新技术就像投资。我们投入时间，然后此技术应该给予（我们）回报。我不是说Kotlin是一种糟糕的语言。 我只是说按照我们的状况，其成本超过了收益。

关于Kotlin的趣事

在波兰，Kotlin是番茄酱中最畅销的品牌之一。这个名字冲突不是任何人的错，但很有趣。Kotlin听起来像Heinz发音一样。



推荐阅读

Linus 又开怼：有时候标准就是一坨屎！
“王者对战”之 MySQL 8 vs PostgreSQL 10
传微软将全资收购 GitHub，价格达 50 亿美元或更高
C++ 协程的近况、设计与实现中的细节和决策
吃透这套架构演化图，从零搭建 Web 网站也不难！