Kotlin 是一门让人觉得惊喜的语言，2017 年 Google I/O 大会上，Google 宣布将 Kotlin 作为 Android 开发的头等语言以后，Kotlin 得到了大量的关注和快速的发展，我们后端开发也在此时进行了第一时间的跟进。Kotlin 代码更加简洁、类型推断、不变性、null 安全、函数式编程、协程等特性，都非常好用，而且能够与 Java 无缝互相调用，迁移成本几乎为零。与其说 Kotlin 是一门新语言，不如说是 Java 上最流行的库。这些好用语法层面的特性的背后都是华丽的语法糖，写 Kotlin 很爽是因为编译器把那些繁琐的东西帮我们都做了。

> 哪有什么岁月静好，不过是有人替你负重前行，

0x01 main 是怎么回事

在 Java 中，main 函数必须要写在一个 class 里面，但是 Kotlin 中却不用这样，比如我们新建了一个 MyTestMain.kt 文件，写入一个 main 函数

```
fun main(args: Array<String>) {
    println("hello kotlin")
}
```

用 kotlinc 编译一下，会发现生成一个类文件 MyTestMainKt.class

```
public final class MyTestMainKt {
  public static final void
main(java.lang.String[]);
    Code:
       0: aload_0
       1: ldc           #9                  //
String args
       3: invokestatic  #15                 //
Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNo
tNull:(Ljava/lang/Object;Ljava/lang/String;)V
       6: ldc           #17                 //
String hello kotlin
       8: astore_1
       9: getstatic     #23                 //
Field java/lang/System.out:Ljava/io/PrintStream;
      12: aload_1
      13: invokevirtual #29                 //
Method java/io/PrintStream.println:
(Ljava/lang/Object;)V
      16: return
}
```

人肉翻译一下

```
public final class MyTestMainKt {
    public static final void main(String[] args)
{
        Intrinsics.checkParameterIsNotNull(args,
"args");
        String str = "hello kotlin";
        System.out.println(str);
    }
}
```

在 Kotlin1.3 版本中，我们甚至可以省略掉 main 函数的参数，更加简洁

```
fun main() {
    println("hello kotlin")
}
```

0x02 object：易如反掌创建单例

在准备面试的过程中，你一定准备过单例模式的 N 中写法，比如饿汉式、懒汉式、单线程写法、双重检查锁写法、枚举

下面这种就是最简单的一种 eager 模式单例

```
public class SingleObject {
   private static SingleObject instance = new
SingleObject();
   private SingleObject() {}
   public static SingleObject getInstance(){
      return instance;
   }
}
```

object 关键字天生为单例而生，只用如下简单的做法就可以实现了上面代码 饿汉式单例模式同样的功能

```
object MySingleton {
}
```

它是如何做到的呢?
用 kotlinc 把上面的源码编译成字节码kotlinc MySingleton.kt

```
static {};
     0: new           #2                  // class MySingleton
     3: dup
     4: invokespecial #25                 // Method "<init>":()V
     7: astore_0
     8: aload_0
     9: putstatic     #27                 // Field INSTANCE:LMySingleton;
    12: return
}
```

- 0 ~ 7：是我们前面介绍对象初始化操作里面非常经典的操作，new-dup-invokespecial-astore，看到这个现在就要形成条件反射，这就是新建一个对象存储到局部变量表的过程。可以理解为对应 Java 中代码MySingleton localMySingleton = new MySingleton()
- 8 ~ 9：是把刚刚新建的变量从局部变量表中捞出来存储到类的静态变量 INSTANCE 中

人肉翻译成 Java 代码就是

```
public final class MySingleton {
    public static final MySingleton INSTANCE;
    static {
        MySingleton localMySingleton = new
MySingleton();
        INSTANCE = localMySingleton;
    }
}
```

0x03 扩展方法

Kotlin 的扩展方法比 Java 要灵活多了，

```
ExtensionTest.kt

class MyClass(val i: Int)
fun MyClass.plusOne() = this.i + 1
fun main(args: Array<String>) {
    val obj = MyClass(1)
    println(obj.plusOne())
}
```

那 Kotlin 编译器会怎么实现这样一个特性呢？先来看下 MyClass类有没有做修改,使用 javap MyClass.class 会发现 MyClass 并没有发现 plusOne 函数的踪迹，这也比较符合常理，因为扩展方法往往是后期动态新增的，直接修改 MyClass 类不太合适，剩下的一个类就是 ExtensionTestKt 了，查看一下字节码

```
public static final int plusOne(MyClass);
    Code:
       0: aload_0
       1: ldc           #9                  //
String receiver$0
       3: invokestatic  #15                 //
Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNo
tNull:(Ljava/lang/Object;Ljava/lang/String;)V
       6: aload_0
       7: invokevirtual #21                 //
Method MyClass.getI:()I
      10: iconst_1
      11: iadd
      12: ireturn
```

可以看到 ExtensionTestKt 新增了一个 plusOne 函数，函数参数是 MyClass 对象，上面的字节码非常简单，人肉翻译机翻译一下是这样

```
public static final int plusOne(MyClass
$receiver) {
    Intrinsics.checkParameterIsNotNull($receiver,
"receiver$0");
    return $receiver.getI() + 1;
}
```

main 调用翻译成 Java 代码

```
MyClass obj = new MyClass(1);
int i = ExtensionTestKt.plusOne(obj);
System.out.println(i);
```

所以 Kotlin 就是扩展函数代码所在的类新建了一个静态的函数，把要扩展的类作为静态函数的第一个参数传入进来，简化而言就是这样：func obj.extension -> OtherClass.extension(obj)
Kotlin 就是用这样一种非常简单轻量的方式实现了函数扩展。

0x04 高级 for 循环

```
for (i in 100 downTo 1 step 2) {
    println(i)
}
输出:
100
98
...
2
```

对应字节码

```
public static final void foo();
Code:
   0: bipush        100
   2: iconst_1
   3: invokestatic  #57                 // Method
kotlin/ranges/RangesKt.downTo:
(II)Lkotlin/ranges/IntProgression;
   6: iconst_2
   7: invokestatic  #61                 // Method
kotlin/ranges/RangesKt.step:
(Lkotlin/ranges/IntProgression;I)Lkotlin/ranges/I
ntProgression;

  10: dup
  11: dup
```

```
  12: invokevirtual #67                 // Method
kotlin/ranges/IntProgression.getFirst:()I
  15: istore_0
  16: invokevirtual #70                 // Method
kotlin/ranges/IntProgression.getLast:()I
  19: istore_1
  20: invokevirtual #73                 // Method
kotlin/ranges/IntProgression.getStep:()I
  23: istore_2

  24: iload_0
  25: iload_1
  26: iload_2
  27: ifle          36
  30: if_icmpgt     58
  33: goto          39
  36: if_icmplt     58

  39: getstatic     #39                 // Field
java/lang/System.out:Ljava/io/PrintStream;
  42: iload_0
  43: invokevirtual #76                 // Method
java/io/PrintStream.println:(I)V
  46: iload_0
  47: iload_1
  48: if_icmpeq     58
  51: iload_0
  52: iload_2
  53: iadd
  54: istore_0
  55: goto          39
  58: return
}
```

我们把局部变量表放出来，你能否通过上面的字节码人肉翻译出 Java 代码呢?

下标	0	1	2
别名	first	last	step
值	100	2	-2

局部变量表

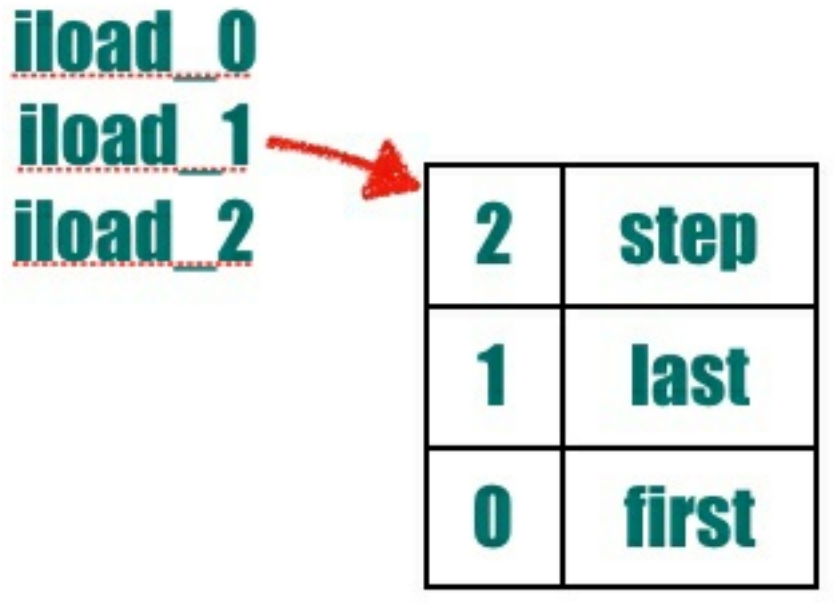
我们逐行翻译一下

- 0 ~ 7：调用了 kotlin 标准库的两个函数，翻译一下大概如下 `IntProgression progression = RangesKt.step(RangesKt.downTo(100, 1), 2);` 这里要注意，last 的初始化值为 2（不是我们代码中的 1），不是我写错了，是 Kotlin 根据 first last step 初始值算出来的最终迭代退出的值，后面会有用

- 10 ~ 23：初始化一些变量为后面循环做准备，这里有三个变量分别是循环开始值（记为 first）、循环结束值（记为 last）、循环 step（记为 step）

- 24 ~ 30：做一些明显不符合条件的跳出。比如 step 大于 0 的情况下，first 应该小于等于 last，step 小于等于 0 的情况下，fist 应该大于等于 last

- 24 ~ 26：加载三个变量到操作数栈上



- 27 行：ifle 指令表示小于等于 0 则跳转 36 行，这里是判断 step 小于等于 0 的情况下，继续进行 first 和 last 的比较。执行完，操作数栈如下

- 30 行与 36 行使用 if_icmpgt 和 if_icmplt 对栈顶的两个变量进行比较（也即first 和 last），如果不合法直接跳出
- 39 ~ 55：while 循环处理。39 ~ 43 打印 first 的值，然后对局部变量表 0 和 1 位置的变量进行比较是否相等，这里是进行 first 和 last 是否相等的判断，如果相等，则退出循环。如果不等，对 first += step 操作

人肉字节码翻译机的结果如下：

```
public static void foo() {
    IntProgression progression =
RangesKt.step(RangesKt.downTo(100, 1), 2);
    int first = progression.getFirst(); // first:
100
    int last = progression.getLast();   // last :
2
    int step = progression.getStep();   // step :
-2
    if (step > 0) {
        if (first > last) {
            return;
        }
    } else if (first < last) {
        return;
    }

    while (true) {
        System.out.println(first);
        if (first == last) {
            return;
        }

        first += step;
    }
}
```

在 while 循环中，注意循环退出的条件是判断 first 是否与 last 相等，而不是 first 是否小于 last，就是因为在 IntProgression 初始化的时候就已经做好了 last 的计算，可以用效率更高的等于在循环中进行比较判断。

0x05 小结

这篇文章我们讲了 Kotlin 语法糖在字节码层面的实现细节，一起来回顾一下要点：第一，没有被任何类包裹的 main 函数在编译后自动生成一个临时类包含了上面的静态 main 函数。第二，object 对象即单例的写法实际上是一个 eager 模式的单例实现。第三，Kotlin 扩展方法实际上是生成了一个静态方法，把对象作为静态方法的参数传入，调用扩展方法实际上是调用了另外一个类的静态方法。第四，我们讲了一下 Kotlin 高级 for 循环的例子，其底层是可以理解为是用 while 语句来实现。

0x06 思考

留一个作业：Kotlin 声明函数时可以指定默认参数值，这样可以避免创建重载的函数，你可以从字节码的角度分析一下具体的实现吗?

欢迎你在留言区留言，和我一起讨论。