

# JVM 基础 - 类字节码详解

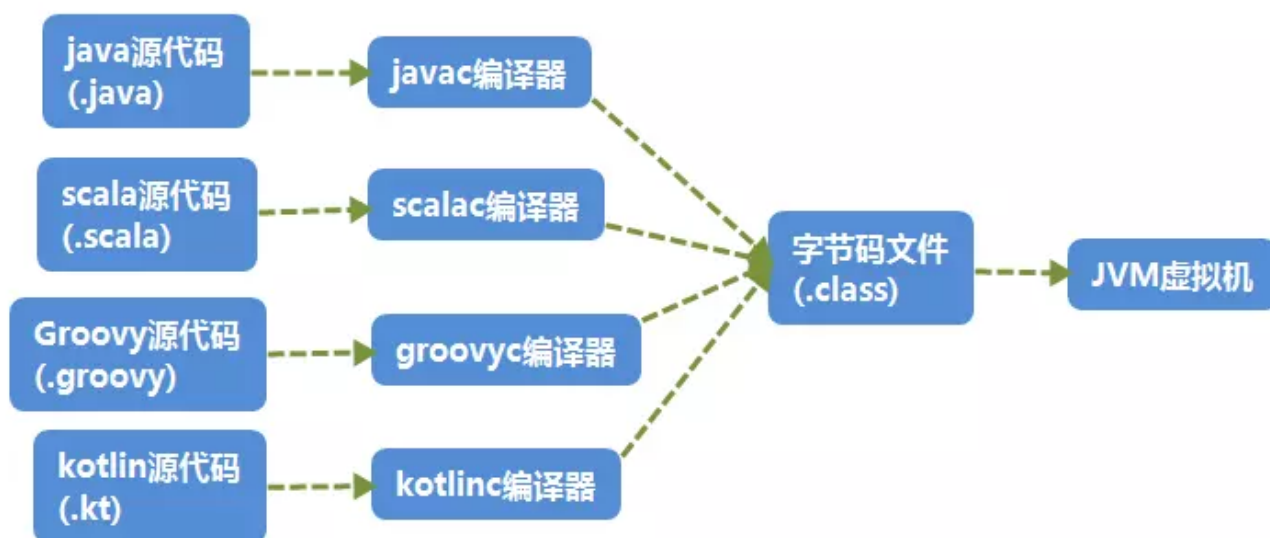
源代码通过编译器编译为字节码，再通过类加载子系统进行加载到JVM中运行。

## 多语言编译为字节码在JVM运行

计算机是不能直接运行java代码的，必须要先运行java虚拟机，再由java虚拟机运行编译后的java代码。这个编译后的java代码，就是本文要介绍的java字节码。

为什么jvm不能直接运行java代码呢，这是因为在cpu层面看来计算机中所有的操作都是一个个指令的运行汇集而成的，java是高级语言，只有人类才能理解其逻辑，计算机是无法识别的，所以java代码必须要先编译成字节码文件，jvm才能正确识别代码转换后的指令并将其运行。

- Java代码间接翻译成字节码，储存字节码的文件再交由运行于不同平台上的JVM虚拟机去读取执行，从而实现一次编写，到处运行的目的。
- JVM也不再只支持Java，由此衍生出了许多基于JVM的编程语言，如Groovy, Scala, Kotlin等等。



## Java字节码文件

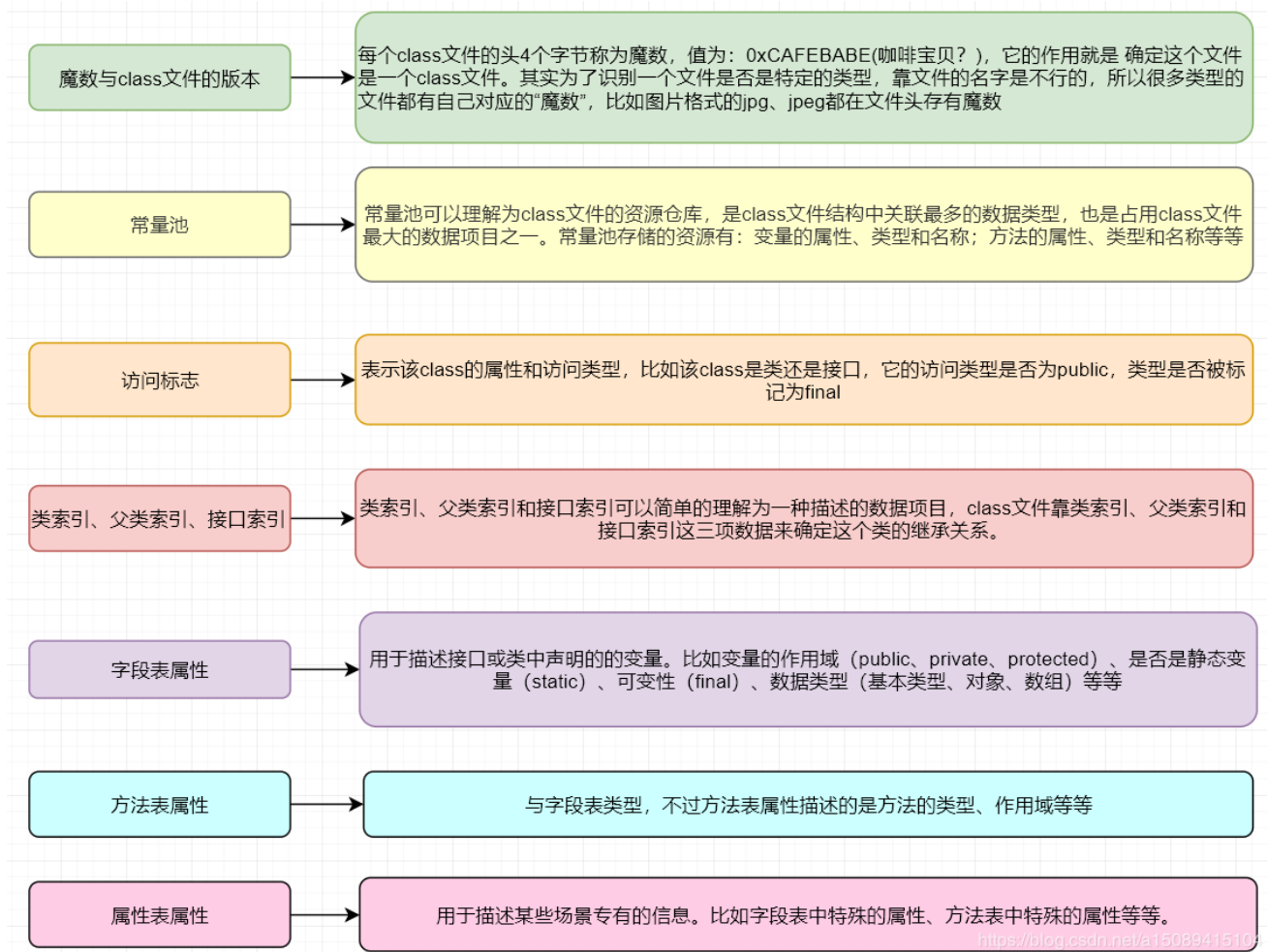
class文件本质上是一个以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑的排列在class文件中。jvm根据其特定的规则解析该二进制数据，从而得到相关信息。

Class文件采用一种伪结构来存储数据，它有两种类型：无符号数和表。这里暂不详细的讲。

本文将通过简单的java例子编译后的文件来理解。

# Class文件的结构属性

在理解之前先从整体看下java字节码文件包含了哪些类型的数据：



## 从一个例子开始

下面以一个简单的例子来逐步讲解字节码。

```
//Main.java
public class Main {

    private int m;

    public int inc() {
        return m + 1;
    }
}
```

通过以下命令,可以在当前所在路径下生成一个Main.class文件。

```
javac Main.java
```

以文本的形式打开生成的class文件，内容如下：

```

cafe babe 0000 0034 0013 0a00 0400 0f09
0003 0010 0700 1107 0012 0100 016d 0100
0149 0100 063c 696e 6974 3e01 0003 2829
5601 0004 436f 6465 0100 0f4c 696e 654e
756d 6265 7254 6162 6c65 0100 0369 6e63
0100 0328 2949 0100 0a53 6f75 7263 6546
696c 6501 0009 4d61 696e 2e6a 6176 610c
0007 0008 0c00 0500 0601 0010 636f 6d2f
7268 7974 686d 372f 4d61 696e 0100 106a
6176 612f 6c61 6e67 2f4f 626a 6563 7400
2100 0300 0400 0000 0100 0200 0500 0600
0000 0200 0100 0700 0800 0100 0900 0000
1d00 0100 0100 0000 052a b700 01b1 0000
0001 000a 0000 0006 0001 0000 0003 0001
000b 000c 0001 0009 0000 001f 0002 0001
0000 0007 2ab4 0002 0460 ac00 0000 0100
0a00 0000 0600 0100 0000 0800 0100 0d00
0000 0200 0e

```

- 文件开头的4个字节("cafe babe")称之为 魔数，唯有以"cafe babe"开头的class文件方可被虚拟机所接受，这4个字节就是字节码文件的身份识别。
- 0000是编译器jdk版本的次版本号0，0034转化为十进制是52,是主版本号，java的版本号从45开始，除1.0和1.1都是使用45.x外,以后每升一个大版本，版本号加一。也就是说，编译生成该class文件的jdk版本为1.8.0。

通过java -version命令稍加验证, 可得结果。

```

Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)

```

继续往下是常量池... 知道是这么分析的就可以了，然后通过工具反编译字节码文件继续去看。

## 反编译字节码文件

使用到java内置的一个反编译工具javap可以反编译字节码文件, 用法: javap <options> <classes>

其中<options>选项包括:

-help --help -?	输出此用法消息
-version	版本信息
-v -verbose	输出附加信息
-l	输出行号和本地变量表
-public	仅显示公共类和成员
-protected	显示受保护的/公共类和成员
-package	显示程序包/受保护的/公共类和成员 (默认)
-p -private	显示所有类和成员
-c	对代码进行反汇编
-s	输出内部类型签名
-sysinfo	显示正在处理的类的系统信息 (路径, 大小, 日期, MD5 散列)
-constants	显示最终常量
-classpath <path>	指定查找用户类文件的位置
-cp <path>	指定查找用户类文件的位置
-bootclasspath <path>	覆盖引导类文件的位置

输入命令javap -verbose -p Main.class查看输出内容:

```
Classfile /E:/JavaCode/TestProj/out/production/TestProj/com/rhythm7/Main.class
  Last modified 2018-4-7; size 362 bytes
  MD5 checksum 4aed8540b098992663b7ba08c65312de
  Compiled from "Main.java"
public class com.rhythm7.Main
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #4.#18      // java/lang/Object."<init>":()V
  #2 = Fieldref           #3.#19      // com/rhythm7/Main.m:I
  #3 = Class               #20        // com/rhythm7/Main
  #4 = Class               #21        // java/lang/Object
  #5 = Utf8                m
  #6 = Utf8                I
  #7 = Utf8                <init>
  #8 = Utf8                ()V
  #9 = Utf8                Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lcom/rhythm7/Main;
  #14 = Utf8               inc
  #15 = Utf8               ()I
  #16 = Utf8               SourceFile
  #17 = Utf8               Main.java
  #18 = NameAndType        #7:#8      // "<init>":()V
  #19 = NameAndType        #5:#6      // m:I
  #20 = Utf8               com/rhythm7/Main
  #21 = Utf8               java/lang/Object
{
  private int m;
    descriptor: I
    flags: ACC_PRIVATE

  public com.rhythm7.Main();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: return
    LineNumberTable:
      line 3: 0
    LocalVariableTable:
      Start Length Slot Name Signature
        0      5     0  this Lcom/rhythm7/Main;

  public int inc();
    descriptor: ()I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
        0: aload_0
        1: getfield      #2                  // Field m:I
        4: iconst_1
        5: iadd
        6: ireturn
```

```
LineNumberTable:
  line 8: 0
LocalVariableTable:
  Start  Length  Slot  Name   Signature
      0       7     0  this   Lcom/rhythm7/Main;
}
SourceFile: "Main.java"
```

## 字节码文件信息

开头的7行信息包括:Class文件当前所在位置，最后修改时间，文件大小，MD5值，编译自哪个文件，类的全限定名，jdk次版本号，主版本号。

然后紧接着的是该类的访问标志：ACC\_PUBLIC, ACC\_SUPER，访问标志的含义如下：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	是否为Public类型
ACC_FINAL	0x0010	是否被声明为final，只有类可以设置
ACC_SUPER	0x0020	是否允许使用invokespecial字节码指令的新语义。
ACC_INTERFACE	0x0200	标志这是一个接口
ACC_ABSTRACT	0x0400	是否为abstract类型，对于接口或者抽象类来说，次标志值为真，其他类型为假
ACC_SYNTHETIC	0x1000	标志这个类并非由用户代码产生
ACC_ANNOTATION	0x2000	标志这是一个注解
ACC_ENUM	0x4000	标志这是一个枚举

## 常量池

Constant pool意为常量池。

常量池可以理解成Class文件中的资源仓库。主要存放的是两大类常量：字面量(Literal)和符号引用(Symbolic References)。字面量类似于java中的常量概念，如文本字符串，final常量等，而符号引用则属于编译原理方面的概念，包括以下三种：

- 类和接口的全限定名(Fully Qualified Name)
- 字段的名称和描述符号(Descriptor)
- 方法的名称和描述符

不同于C/C++，JVM是在加载Class文件的时候才进行的动态链接，也就是说这些字段和方法符号引用只有在运行期转换后才能获得真正的内存入口地址。当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建或运行时解析并翻译到具体的内存地址中。直接通过反编译文件来查看字节码内容：

```
#1 = Methodref      #4.#18      // java/lang/Object."<init>":()V
#4 = Class          #21          // java/lang/Object
#7 = Utf8           <init>
#8 = Utf8           ()V
#18 = NameAndType    #7:#8          // "<init>":()V
#21 = Utf8           java/lang/Object
```

**第一个常量**是一个方法定义，指向了第4和第18个常量。以此类推查看第4和第18个常量。最后可以拼接成第一个常量右侧的注释内容：

```
java/lang/Object."<init>":()V
```

这段可以理解为该类的实例构造器的声明，由于Main类没有重写构造方法，所以调用的是父类的构造方法。此处也说明了Main类的直接父类是Object。该方法默认返回值是V,也就是void，无返回值。

**第二个常量**同理可得：

```
#2 = Fieldref      #3.#19      // com/rhythm7/Main.m:I
#3 = Class          #20          // com/rhythm7/Main
#5 = Utf8           m
#6 = Utf8           I
#19 = NameAndType    #5:#6          // m:I
#20 = Utf8           com/rhythm7/Main
```

复制代码此处声明了一个字段m，类型为I, I即是int类型。关于字节码的类型对应如下：

标识字符	含义
	B
C	基本类型char
D	基本类型double
F	基本类型float
I	基本类型int
J	基本类型long
S	基本类型short
Z	基本类型boolean
V	特殊类型void
L	对象类型，以分号结尾，如Ljava/lang/Object;

对于数组类型，每一位使用一个前置的[字符来描述，如定义一个java.lang.String[][]类型的维数组，将被记录为[[Ljava/lang/String;

# 方法表集合

在常量池之后的是对类内部的方法描述，在字节码中以表的集合形式表现，暂且不管字节码文件的16进制文件内容如何，我们直接看反编译后的内容。


```
private int m;  
descriptor: I  
flags: ACC_PRIVATE
```

此处声明了一个私有变量m，类型为int，返回值为int

```
public com.rhythm7.Main();  
descriptor: ()V  
flags: ACC_PUBLIC  
Code:  
  stack=1, locals=1, args_size=1  
  0: aload_0  
  1: invokespecial #1           // Method java/lang/Object."<init>":()V  
  4: return  
LineNumberTable:  
  line 3: 0  
LocalVariableTable:  
  Start  Length  Slot  Name  Signature  
    0       5     0   this  Lcom/rhythm7/Main;
```

这里是构造方法：Main()，返回值为void，公开方法。

code内的主要属性为：

- **stack**: 最大操作数栈，JVM运行时会根据这个值来分配栈帧(Frame)中的操作栈深度。此处为1
- **locals**: 局部变量所需的存储空间，单位为Slot，Slot是虚拟机为局部变量分配内存时所使用的最小单位，为4个字节大小。方法参数(包括实例方法中的隐藏参数this)，显示异常处理器的参数(try catch中的catch块所定义的异常)，方法体中定义的局部变量都需要使用局部变量表来存放。值得一提的是，locals的大小并不一定等于所有局部变量所占的Slot之和，因为局部变量中的Slot是可以重用的。
- **args\_size**: 方法参数的个数，这里是1，因为每个实例方法都会有一个隐藏参数this
- **attribute\_info**: 方法体内容，0,1,4为字节码"行号"，该段代码的意思是将第一个引用类型本地变量推送至栈顶，然后执行该类型的实例方法，也就是常量池存放的第一个变量，也就是注释里的"java/lang/Object.""V"，然后执行返回语句，结束方法。
- **LineNumberTable**: 该属性的作用是描述源码行号与字节码行号(字节码偏移量)之间的对应关系。可以使用 -g:none 或 -g:lines 选项来取消或要求生成这项信息，如果选择不生成LineNumberTable，当程序运行异常时将无法获取到发生异常的源码行号，也无法按照源码的行数来调试程序。
- **LocalVariableTable**: 该属性的作用是描述帧栈中局部变量与源码中定义的变量之间的关系。可以使用 -g:none 或 -g:vars 来取消或生成这项信息，如果没有生成这项信息，那么当别人引用这个方法时，将无法获取到参数名称，取而代之的是arg0, arg1这样的占位符。start 表示该局部变量在哪一行开始可见，length表示可见行数，Slot代表所在帧栈位置，Name是变量名称，然后是类型签名。

同理可以分析Main类中的另一个方法"inc()":

方法体内的内容是：将this入栈，获取字段#2并置于栈顶，将int类型的1入栈，将栈内顶部的两个数值相加，返回一个int类型的值。

# 类名

最后很显然是源码文件：

```
SourceFile: "Main.java"
```

## 再看两个示例

### 分析try-catch-finally

通过以上一个最简单的例子，可以大致了解源码被编译成字节码后是什么样子的。下面利用所学的知识点来分析一些Java问题：

```
public class TestCode {  
    public int foo() {  
        int x;  
        try {  
            x = 1;  
            return x;  
        } catch (Exception e) {  
            x = 2;  
            return x;  
        } finally {  
            x = 3;  
        }  
    }  
}
```

试问当不发生异常和发生异常的情况下，foo()的返回值分别是多少。

```
javac TestCode.java  
javap -verbose TestCode.class
```

查看字节码的foo方法内容：

```
public int foo();  
descriptor: ()I  
flags: ACC_PUBLIC  
Code:  
    stack=1, locals=5, args_size=1  
    0: iconst_1 //int型1入栈 ->栈顶=1  
    1: istore_1 //将栈顶的int型数值存入第二个局部变量 ->局部2=1  
    2: iload_1 //将第二个int型局部变量推送至栈顶 ->栈顶=1  
    3: istore_2 ///!!将栈顶int型数值存入第三个局部变量 ->局部3=1  
  
    4: iconst_3 //int型3入栈 ->栈顶=3  
    5: istore_1 //将栈顶的int型数值存入第二个局部变量 ->局部2=3  
    6: iload_2 ///!!将第三个int型局部变量推送至栈顶 ->栈顶=1  
    7: ireturn //从当前方法返回栈顶int数值 ->1  
  
    8: astore_2 // ->局部3=Exception  
    9: iconst_2 // ->栈顶=2  
    10: istore_1 // ->局部2=2  
    11: iload_1 //->栈顶=2
```



```

12: istore_3 ///!! ->局部4=2

13: iconst_3 // ->栈顶=3
14: istore_1 // ->局部1=3
15: iload_3 ///!! ->栈顶=2
16: ireturn // -> 2

17: astore      4 //将栈顶引用型数值存入第五个局部变量=any
19: iconst_3 //将int型数值3入栈 -> 栈顶3
20: istore_1 //将栈顶第一个int数值存入第二个局部变量 -> 局部2=3
21: aload      4 //将局部第五个局部变量(引用型)推送至栈顶
23: athrow //将栈顶的异常抛出
Exception table:
   from    to  target type
     0      4      8   Class java/lang/Exception //0到4行对应的异常，对应#8中储存的异常
     0      4     17    any //Exception之外的其他异常
     8     13     17    any
    17     19     17    any

```

在字节码的4,5, 以及13,14中执行的是同一个操作, 就是将int型的3入操作数栈顶, 并存入第二个局部变量。这正是我们源码在finally语句块中内容。也就是说, JVM在处理异常时, 会在每个可能的分支都将finally语句重复执行一遍。

通过一步步分析字节码, 可以得出最后的运行结果是:

- 不发生异常时: return 1
- 发生异常时: return 2
- 发生非Exception及其子类的异常, 抛出异常, 不返回值

以上例子来自于《深入理解Java虚拟机 JVM高级特性与最佳实践》, 关于虚拟机字节码指令表, 也可以在《深入理解Java虚拟机 JVM高级特性与最佳实践-附录B》中获取。

## kotlin 函数扩展的实现

kotlin提供了扩展函数的语言特性, 借助这个特性, 我们可以给任意对象添加自定义方法。

以下示例为Object添加"sayHello"方法

```

//SayHello.kt
package com.rhythm7

fun Any.sayHello() {
    println("Hello")
}

```

编译后, 使用javap查看生成SayHelloKt.class文件的字节码。

```

Classfile /E:/JavaCode/TestProj/out/production/TestProj/com/rhythm7/SayHelloKt.class
Last modified 2018-4-8; size 958 bytes
MD5 checksum 780a04b75a91be7605cac4655b499f19
Compiled from "SayHello.kt"
public final class com.rhythm7.SayHelloKt
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_FINAL, ACC_SUPER
Constant pool:

```

```

//省略常量池部分字节码
{
    public static final void sayHello(java.lang.Object);
    descriptor: (Ljava/lang/Object;)V
    flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
    Code:
        stack=2, locals=2, args_size=1
        0: aload_0
        1: ldc          #9              // String $receiver
        3: invokestatic  #15             // Method
        kotlin/jvm/internal/Intrinsics.checkNotNull:(Ljava/lang/Object;Ljava/lang/String;)V
        6: ldc          #17              // String Hello
        8: astore_1
        9: getstatic    #23              // Field java/lang/System.out:Ljava/io/PrintStream;
        12: aload_1
        13: invokevirtual #28              // Method java/io/PrintStream.println:
        (Ljava/lang/Object;)V
        16: return
    LocalVariableTable:
        Start Length Slot Name Signature
        0      17     0 $receiver Ljava/lang/Object;
    LineNumberTable:
        line 4: 6
        line 5: 16
    RuntimeInvisibleParameterAnnotations:
        0:
        0: #7()
}
SourceFile: "SayHello.kt"

```

观察头部发现,kotlin为文件SayHello生成了一个类, 类名"com.rhythm7.SayHelloKt".

由于我们一开始编写SayHello.kt时并不希望SayHello是一个可实例化的对象类, 所以, SayHelloKt是无法被实例化的, SayHelloKt并没有任何一个构造器。

再观察唯一的一个方法: 发现Any.sayHello()的具体实现是静态不可变方法的形式:

```
public static final void sayHello(java.lang.Object);
```

所以当我们在其他地方使用Any.sayHello()时, 事实上等同于调用java的SayHelloKt.sayHello(Object)方法。

顺便一提的是, 当扩展的方法为Any时, 意味着Any是non-null的, 这时, 编译器会在方法体的开头检查参数的非空, 即调用 kotlin.jvm.internal.Intrinsics.checkNotNull(Object value, String paramName) 方法来检查传入的Any类型对象是否为空。如果我们扩展的函数为Any?.sayHello(), 那么在编译后的文件中则不会有这段字节码的出现。