不学习底层知识可能不会阻碍你称为一个称职的程序员，但也许会阻碍你成为一个优秀的程序员。我所理解的底层知识，是指编程或开发所依赖的平台（或者框架、工具）的知识。对于 Java 开发者来说，虚拟机、字节码就是其底层知识。

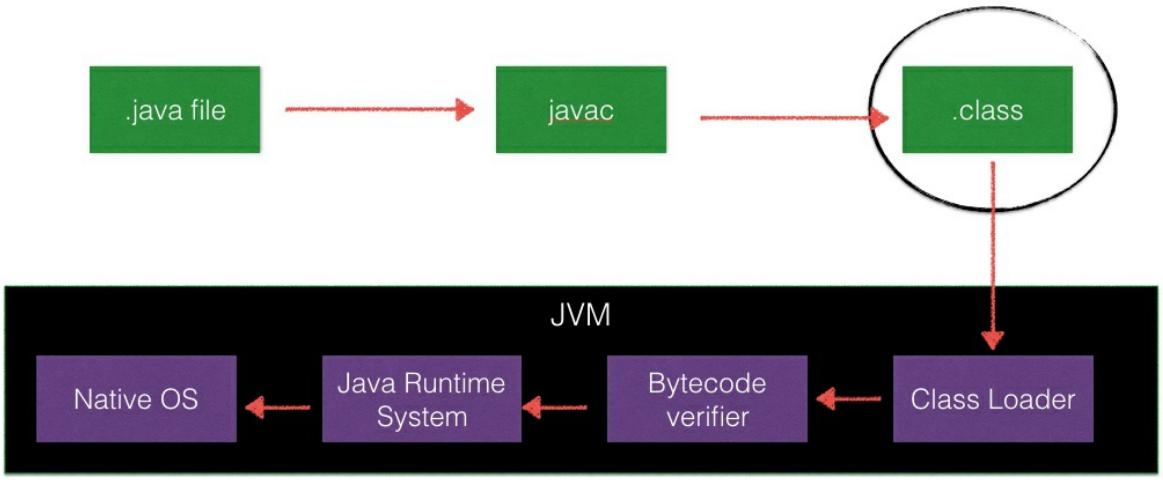
这篇文章我们以输出 "Hello, World" 来开始字节码之旅，如果之前没有怎么接触过字节码的话，这篇文章应该能够让你对字节码有一个最基本的认识。

0x01 java 文件如何变成 .class 文件

新建一个 Hello.java 文件，源码如下：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Java 从源文件到执行的过程如下图所示



JDK 工具 javac（java 编译器）帮我们完成了源文件编译成 JVM 可识别的 class 文件的工作。
在命令行中执行javac Hello.java，可以看到生成了Hello.class 文件。用xxd 命令以 16 进制的方式查看这个 class 文件。

```
xxd Hello.class
00000000: cafe babe 0000 0034 0022 0a00 0600 1409  .......4."......
00000010: 0015 0016 0800 170a 0018 0019 0700 1a07  ................
00000020: 001b 0100 063c 696e 6974 3e01 0003 2829  .....<init>...()
00000030: 5601 0004 436f 6465 0100 0f4c 696e 654e  V...Code...LineN
00000040: 756d 6265 7254 6162 6c65 0100 124c 6f63  umberTable...Loc
00000050: 616c 5661 7269 6162 6c65 5461 626c 6501  alVariableTable.
00000060: 0004 7468 6973 0100 074c 4865 6c6c 6f3b  ..this...LHello;
```

0x02 魔数 0xCAFEBABE

class 文件的头四个字节称为魔数（Magic Number），可以看到 class 的**魔数**为 0xCAFEBABE。很多文件都以魔数来进行文件类型的区分，比如 PDF 文件的魔数是%PDF-(16进制0x255044462D)，png 文件的魔数是.png（0x89504E47）。文件格式的制定者可以自由的选择魔数值，只要魔数值还没有被广泛的采用过且不会引起混淆即可。

Java 早期开发者选用了这样一个浪漫气息的魔数，高司令有解释这一段 轶事 (http://mishadoff.com/blog/java-magic-part-2-0xcafebabe/)。这个魔数值在 Java 还成为 Oak 语言的时候就已经确定下来了。

这个魔数是 JVM 识别 .class 文件的标志，虚拟机在加载类文件之前会先检查这四个字节，如果不是 0xCAFEBABE 则拒绝加载该文件，更多关于字节码格式的说明，我们会在后面的文章中慢慢介绍。

0x03 javap 详解

类文件是二进制块，想直接与它打交道比较艰难，但是很多情况下我们必须理解类文件。比如服务器上的接口出了 bug，重新打包部署以后问题并没有解决，为了找出原因你可能需要看一下部署以后的 class 文件究竟是不是我们想要的。还有一种情况跟你合作的开发商跑路了，只给你留下一堆编译过的代码，没有源代码，当出 bug 时我们需要研究这些类文件，看问题出在哪里。
好在 JDK 提供了专门用来分析类文件的工具：javap，用来方便的窥探 class 文件内部的细节。javap 有比较多的参数选项，其中`-c -v -l -p -s`是最常用的。

```
Usage: javap <options> <classes>
where possible options include:
  -help  --help  -?        Print this usage
message
  -version                 Version information
  -v  -verbose             Print additional
information
  -l                       Print line number and
local variable tables
  -public                  Show only public
classes and members
  -protected               Show protected/public
classes and members
  -package                 Show
package/protected/public classes
                           and members (default)
  -p  -private             Show all classes and
members
  -c                       Disassemble the code
  -s                       Print internal type
signatures
  -sysinfo                 Show system info
(path, size, date, MD5 hash)
                           of class being
processed
  -constants               Show final constants
  -classpath <path>        Specify where to find
user class files
  -cp <path>               Specify where to find
user class files
  -bootclasspath <path>    Override location of
bootstrap class files
```

- **-c选项**

最常用的选项是-c，可以对类进行反编译。执行javap -c Hello的输出结果如下

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3  // String Hello, World
       5: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

上面代码前面的数字表示从方法开始算起的字节码偏移量

- 3 ~ 7 行：可以看到虽然没有写 Hello 类的构造器函数，编译器会自动加上一个默认构造器函数
- 5 行：aload_0 这个操作码是 aload_x 格式操作码中的一个。它们用来把**对象引用**加载到操作数栈。 x 表示正在被访问的局部变量数组的位置。在这里的 0 代表什么呢？我们知道非静态的函数都有第一个默认参数，那就是 **this**，这里的 aload_0 就是把 this 入栈

- 6 行：invokespecial #1，invokespecial 指令调用实例初始化方法、私有方法、父类方法，#1 指的是常量池中的第一个，这里是方法引用`java/lang/Object."<init>":()V`，也即构造器函数
- 7 行：return，这个操作码属于 ireturn、lreturn、freturn、dreturn、areturn 和 return 操作码组中的一员，其中 i 表示 int，返回整数，同类的还有 l 表示 long，f 表示 float，d 表示 double，a 表示 对象引用。没有前缀类型字母的 return 表示返回 void

到此为止，默认构造器函数就讲完了，接下来，我们来看 9 ~ 14 行的 main 函数

- 11 行：getstatic #2，getstatic 获取指定类的静态域，并将其值压入栈顶，#2 代表常量池中的第 2 个，这里表示的是`java/lang/System.out:Ljava/io/PrintStream;`，其实就是java.lang.System 类的静态变量 out（类型是 PrintStream）
- 12 行：ldc #3、，ldc 用来将常量从运行时常量池压栈到操作数栈，#3 代表常量池的第三个（字符串 Hello, World）
- 13 行：invokevirtual #4，invokevirutal 指令调用一个对象的实例方法，#4 表示 PrintStream.println(String) 函数引用，并把栈顶两个元素出栈

过程如下图

```
System.out.println("Hello, World");
```

- **-p 选项**

默认情况下，javap 会显示访问权限为 public、protected 和默认（包级 protected）级别的方法，加上 -p 选项以后可以显示 private 方法和字段

- **-v 选项**

javap 加上 -v 参数的输出更多详细的信息，比如栈大小、方法参数的个数。

```
public Hello();
    stack=1, locals=1, args_size=1

public static void main(java.lang.String[]);
    stack=2, locals=1, args_size=1
```

为什么Hello()和main()的args_size都等于1呢？明明 Hello 的构造器函数没有参数的呀？
对于非静态函数，this对象会作为函数的隐式第一个参数，所以Hello()的args_size=1
对于静态main函数，不需要this对象，它的参数就是String[] args这个数组，也等于1

- **-s选项**

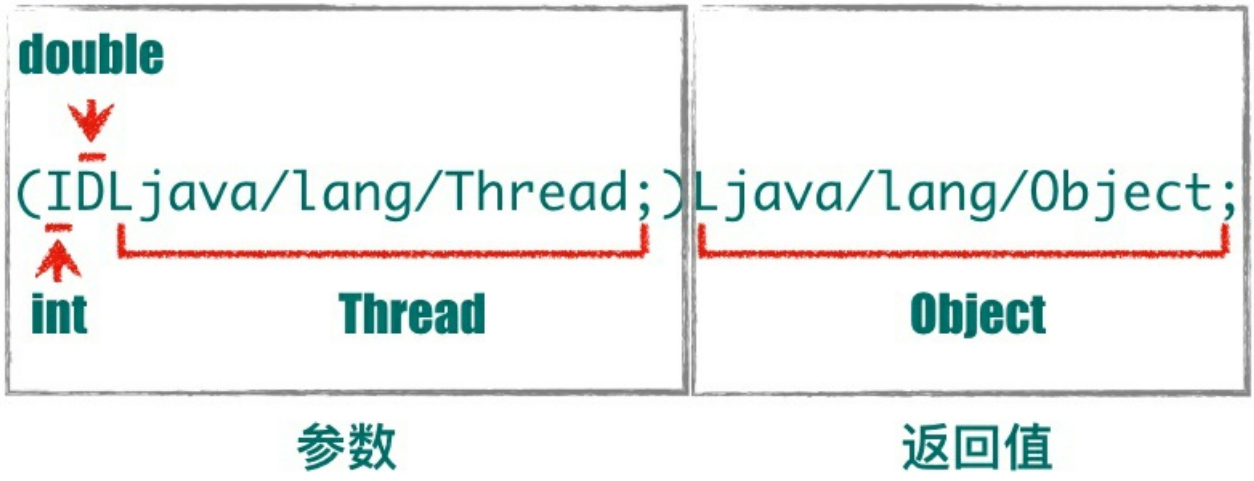
javap 还有一个好用的选项 -s，可以输出签名的类型描述符。我们可以看下 Hello.java 所有的方法签名

```
javap -s Hello
Compiled from "Hello.java"
public class Hello {
  public Hello();
    descriptor: ()V
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
}
```

可以看到 main 函数的方法签名是 ([Ljava/lang/String;)V。JVM 内部使用方法签名与我们日常阅读的方法签名不太一样，但是后面会频繁遇到，主要分为两部分字段描述符和方法描述符。

字段描述符（Field Descriptor），是一个表示类、实例或局部变量的语法符号，它的表示形式是紧凑的，比如 int 是用 I 表示的。完整的类型描述符如下表

方法描述符（Method Descriptor）表示一个方法所需参数和返回值信息，表示形式为(ParameterDescriptor*) ReturnDescriptor。
ParameterDescriptor 表示参数类型，ReturnDescriptor表示返回值信息，当没有返回值时用V表示。比如方法Object foo(int i, double d, Thread t)的描述符为 (IDLjava/lang/Thread;)Ljava/lang/Object;



0x04 小结

这篇文章讲解了一个输出 "Hello, World" 的字节码的细节，一起来回顾一下要点：

- 第一，class 文件的魔数是具有浪漫气息的 0xCAFEBABE；
- 第二，我们讲解了字节码分析的利器 javap 的各个参数详细的用法。第三，讲解了字段描述符与方法描述符在 JVM 层面的表示规则，方便我们后面文章的理解。

0x05 思考

最后，给你留一个思考题，javap 的 `-l` 参数有什么用?

欢迎你在留言区留言，和我一起讨论。