

Class 类文件结构

Java 跨平台的基础

各种不同平台的虚拟机与所有平台都统一使用的程序存储格式——字节码（ByteCode）是构成平台无关性的基石，也是语言无关性的基础。Java 虚拟机不和包括 Java 在内的任何语言绑定，它只与“Class 文件”这种特定的二进制文件格式所关联，Class 文件中包含了 Java 虚拟机指令集和符号表以及若干其他辅助信息。

Class 类的本质

任何一个 Class 文件都对应着唯一的一个类或接口的定义信息，但反过来说，Class 文件实际上它并不一定以磁盘文件的形式存在。

Class 文件是一组以 8 位字节为基础单位的二进制流。

Class 文件格式

各个数据项目严格按照顺序紧凑地排列在 Class 文件之中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。

Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：无符号数和表。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表。

Class 文件格式详解

Class 的结构不像 XML 等描述语言，由于它没有任何分隔符号，所以在其中的数据项，无论是顺序还是数量，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。

按顺序包括：

魔数与 Class 文件的版本

每个 Class 文件的头 4 个字节称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的 Class 文件。使用魔数而不是扩展名来进行识别主要是基于安全方面的考虑，因为文件扩展名可以随意地改动。文件格式的制定者可以自由地选择魔数值，只要这个魔数值还没有被广泛采用过同时又不会引起混淆即可。

紧接着魔数的 4 个字节存储的是 Class 文件的版本号：第 5 和第 6 个字节是次版本号（MinorVersion），第 7 和第 8 个字节是主版本号（Major Version）。Java 的版本号是从 45 开始的，JDK 1.1 之后的每个 JDK 大版本发布主版本号向上加 1 高版本的 JDK 能向下兼容以前版本的 Class 文件，但不能运行以后版本的 Class 文件，即使文件格式并未发生任何变化，虚拟机也必须拒绝执行超过其版本号的 Class 文件。

常量池

常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 `u2` 类型的数据，代表常量池容量计数值（`constant_pool_count`）。与 Java 中语言习惯不一样的是，这个容量计数是从 1 而不是 0 开始的

常量池中主要存放两大类常量：字面量（**Literal**）和符号引用（**Symbolic References**）。字面量比较接近于 Java 语言层面的常量概念，如文本字符串、声明为 `final` 的常量值等。而符号引用则属于编译原理方面的概念，包括了下面三类常量：

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

访问标志

用于识别一些类或者接口层次的访问信息，包括：这个 **Class** 是类还是接口；是否定义为 `public` 类型；是否定义为 `abstract` 类型；如果是类的话，是否被声明为 `final` 等

类索引、父类索引与接口索引集合

这三项数据来确定这个类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。由于 Java 语言不允许多重继承，所以父类索引只有一个，除了 `java.lang.Object` 之外，所有的 Java 类都有父类，因此除了 `java.lang.Object` 外，所有 Java 类的父类索引都不为 0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 `implements` 语句（如果这个类本身是一个接口，则应当是 `extends` 语句）后的接口顺序从左到右排列在接口索引集合中

字段表集合

描述接口或者类中声明的变量。字段（**field**）包括类级变量以及实例级变量。

而字段叫什么名字、字段被定义为什么数据类型，这些都是无法固定的，只能引用常量池中的常量来描述。

字段表集合中不会列出从超类或者父接口中继承而来的字段，但有可能列出原本 Java 代码之中不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

方法表集合

描述了方法的定义，但是方法里的 Java 代码，经过编译器编译成字节码指令后，存放在属性表集合中的方法属性表集合中一个名为“**Code**”的属性里面。

与字段表集合相类似的，如果父类方法在子类中没有被重写（**Override**），方法表集合中就不会出现来自父类的方法信息。但同样的，有可能会由编译器自动添加的方法，最典型的便是类构造器“`<clinit>`”方法和实例构造器“`<init>`”

属性表集合

存储 **Class** 文件、字段表、方法表都自己的属性表集合，以用于描述某些场景专有的信息。如方法的代码就存储在 **Code** 属性表中。

字节码指令

Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字（称为操作码，**Opcode**）以及跟随其后的零至多个代表此操作所需参数（称为操作数，**Operands**）而构成。

由于限制了 Java 虚拟机操作码的长度为一个字节（即 0~255），这意味着指令集的操作码总数不可能超过 256 条。

大多数的指令都包含了其操作所对应的数据类型信息。例如：

`iload` 指令用于从局部变量表中加载 `int` 型的数据到操作数栈中，而 `fload` 指令加载的则是 `float` 类型的数据。

大部分的指令都没有支持整数类型 `byte`、`char` 和 `short`，甚至没有任何指令支持 `boolean` 类型。大多数对于 `boolean`、`byte`、`short` 和 `char` 类型数据的操作，实际上都是使用相应的 `int` 类型作为运算类型

阅读字节码作为了解 Java 虚拟机的基础技能，请熟练掌握。请熟悉并掌握常见指令即可。

加载和存储指令

用于将数据在栈帧中的局部变量表和操作数栈之间来回传输，这类指令包括如下内容。

将一个局部变量加载到操作栈：`iload`、`iload_<n>`、`lload`、`lload_<n>`、`fload`、`fload_<n>`、`dload`、`dload_<n>`、`aload`、`aload_<n>`。

将一个数值从操作数栈存储到局部变量表：`istore`、`istore_<n>`、`lstore`、`lstore_<n>`、`fstore`、`fstore_<n>`、`dstore`、`dstore_<n>`、`astore`、`astore_<n>`。

将一个常量加载到操作数栈：`bipush`、`sipush`、`ldc`、`ldc_w`、`ldc2_w`、`aconst_null`、`iconst_m1`、`iconst_<i>`、`lconst_<l>`、`fconst_<f>`、`dconst_<d>`。

扩充局部变量表的访问索引的指令：`wide`。

运算或算术指令

用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。

加法指令：`iadd`、`ladd`、`fadd`、`dadd`。

减法指令：`isub`、`lsub`、`fsub`、`dsub`。

乘法指令：`imul`、`lmul`、`fmul`、`dmul` 等等

类型转换指令

可以将两种不同的数值类型进行相互转换，

Java 虚拟机直接支持以下数值类型的宽化类型转换（即小范围类型向大范围类型的安全转换）：

`int` 类型到 `long`、`float` 或者 `double` 类型。

`long` 类型到 `float`、`double` 类型。

`float` 类型到 `double` 类型。

处理窄化类型转换（Narrowing Numeric Conversions）时，必须显式地使用转换指令来完成，这些转换指令包括：`i2b`、`i2c`、`i2s`、`l2i`、`f2i`、`f2l`、`d2i`、`d2l` 和 `d2f`。

创建类实例的指令：

`new`。

创建数组的指令：

`newarray`、`anewarray`、`multianewarray`。

访问字段指令：

getfield、putfield、getstatic、putstatic。

数组存取相关指令

把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload。

将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore。

取数组长度的指令：arraylength。

检查类实例类型的指令：

instanceof、checkcast。

操作数栈管理指令

如同操作一个普通数据结构中的堆栈那样，Java 虚拟机提供了一些用于直接操作操作数栈的指令，包括：将操作数栈的栈顶一个或两个元素出栈：pop、pop2。

复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、dup_x2、dup2_x2。

将栈最顶端的两个数值互换：swap。

控制转移指令

控制转移指令可以让 Java 虚拟机有条件或无条件地从指定的位置指令而不是控制转移指令的下一条指令继续执行程序，从概念模型上理解，可以认为控制转移指令就是在有条件或无条件地修改 PC 寄存器的值。控制转移指令如下。

条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpeq、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne。

复合条件分支：tableswitch、lookupswitch。

无条件分支：goto、goto_w、jsr、jsr_w、ret。

方法调用指令

invokevirtual 指令用于调用对象的实例方法，根据对象的实际类型进行分派（虚方法分派），这也是 Java 语言中最常见的方法分派方式。

invokeinterface 指令用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。

invokespecial 指令用于调用一些需要特殊处理的实例方法，包括实例初始化方法、私有方法和父类方法。

invokestatic 指令用于调用类方法（static 方法）。

invokedynamic 指令用于在运行时动态解析出调用点限定符所引用的方法，并执行该方法，前面 4 条调用指令的分派逻辑都固化在 Java 虚拟机内部，而 invokedynamic 指令的分派逻辑是由用户所设定的引导方法决定的。

方法调用指令与数据类型无关。

方法返回指令

是根据返回值的类型区分的，包括 `ireturn`（当返回值是 `boolean`、`byte`、`char`、`short` 和 `int` 类型时使用）、`lreturn`、`freturn`、`dreturn` 和 `areturn`，另外还有一条 `return` 指令供声明为 `void` 的方法、实例初始化方法以及类和接口的类初始化方法使用。

异常处理指令

在 Java 程序中显式抛出异常的操作（`throw` 语句）都由 `athrow` 指令来实现

同步指令

有 `monitorenter` 和 `monitorexit` 两条指令来支持 `synchronized` 关键字的语义

类加载机制

概述

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7 个阶段。其中验证、准备、解析 3 个部分统称为连接（Linking）

于初始化阶段，虚拟机规范则是严格规定了有且只有 5 种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

- 1）遇到 `new`、`getstatic`、`putstatic` 或 `invokestatic` 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 `new` 关键字实例化对象的时候、读取或设置一个类的静态字段（被 `final` 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
- 2）使用 `java.lang.reflect` 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 3）当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 4）当虚拟机启动时，用户需要指定一个要执行的主类（包含 `main()` 方法的那个类），虚拟机会先初始化这个主类。
- 5）当使用 JDK 1.7 的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后的解析结果 `REF_getStatic`、`REF_putStatic`、`REF_invokeStatic` 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

注意：

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

常量 `HELLOWORLD`，但其实在编译阶段通过常量传播优化，已经将此常量的值“hello world”存储到了 `NotInitialization` 类的常量池中，以后 `NotInitialization` 对常量 `ConstClass.HELLOWORLD` 的引用实际都被转化为 `NotInitialization` 类对自身常量池的引用了。

也就是说，实际上 `NotInitialization` 的 `Class` 文件之中并没有 `ConstClass` 类的符号引用入口，这两个类在编译成 `Class` 之后就不存在任何联系了。

加载阶段

虚拟机需要完成以下 3 件事情：

- 1) 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 3) 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口。

验证

是连接阶段的第一步，这一阶段的目的是为了确保 `Class` 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。但从整体上看，验证阶段大致上会完成下面 4 个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

准备阶段

是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被 `static` 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 `Java` 堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
public static int value=123;
```

那变量 `value` 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 `Java` 方法，而把 `value` 赋值为 123 的 `putstatic` 指令是程序被编译后，存放于类构造器 `<clinit>()` 方法之中，所以把 `value` 赋值为 123 的动作将在初始化阶段才会执行。表 7-1 列出了 `Java` 中所有基本数据类型的零值。

假设上面类变量 `value` 的定义变为：`public static final int value=123;`

编译时 `Javac` 将会为 `value` 生成 `ConstantValue` 属性，在准备阶段虚拟机就会根据 `ConstantValue` 的设置将 `value` 赋值为 123。

解析阶段

是虚拟机将常量池内的符号引用替换为直接引用的过程

类初始化阶段

是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 `Java` 程序代码在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器 `<clinit>()` 方法的过程。`<clinit>()` 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（`static{}` 块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。`<clinit>()` 方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成 `<clinit>()` 方法。

虚拟机会保证一个类的 `<clinit>()` 方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的 `<clinit>()` 方法，其他线程都需要阻塞等待，直到活动线程执行 `<clinit>()` 方法完毕。如果在一个类的 `<clinit>()` 方法中有耗时很长的操作，就可能造成多个进程阻塞。

类加载器

如何自定义类加载器，看代码

系统的类加载器

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的 Class 对象的 equals () 方法、isAssignableFrom () 方法、isInstance () 方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况。

在自定义 ClassLoader 的子类时候，我们常见的会有两种做法，一种是重写 loadClass 方法，另一种是重写 findClass 方法。其实这两种方法本质上差不多，毕竟 loadClass 也会调用 findClass，但是从逻辑上讲我们最好不要直接修改 loadClass 的内部逻辑。我建议的做法是只在 findClass 里重写自定义类的加载方法。

loadClass 这个方法是实现双亲委托模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委托模型框架内进行小范围的改动，不破坏原有的稳定结构。同时，也避免了自己重写 loadClass 的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择。

双亲委派模型

从 Java 虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++ 语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

启动类加载器（Bootstrap ClassLoader）：这个类加载器负责将存放在 <JAVA_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 null 代替即可。

扩展类加载器（Extension ClassLoader）：这个加载器由 sun.misc.Launcher\$ExtClassLoader 实现，它负责加载 <JAVA_HOME>\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader）：这个类加载器由 sun.misc.Launcher\$AppClassLoader 实现。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader () 方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认类加载器。

我们的应用程序都是由这 3 种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 `java.lang.Object`，它存放在 `rt.jar` 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 `Object` 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 `java.lang.Object` 的类，并放在程序的 `ClassPath` 中，那系统中将会出现多个不同的 `Object` 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

Tomcat 类加载机制

Tomcat 本身也是一个 java 项目，因此其也需要被 JDK 的类加载机制加载，也就必然存在引导类加载器、扩展类加载器和应用(系统)类加载器。

`Common ClassLoader` 作为 `Catalina ClassLoader` 和 `Shared ClassLoader` 的 parent，而 `Shared ClassLoader` 又可能存在多个 children 类加载器 `WebApp ClassLoader`，一个 `WebApp ClassLoader` 实际上就对应一个 Web 应用，那 Web 应用就有可能存在 Jsp 页面，这些 Jsp 页面最终会转成 class 类被加载，因此也需要一个 Jsp 的类加载器。

需要注意的是，在代码层面 `Catalina ClassLoader`、`Shared ClassLoader`、`Common ClassLoader` 对应的实体类实际上都是 `URLClassLoader` 或者 `SecureClassLoader`，一般我们只是根据加载内容的不同和加载父子顺序的关系，在逻辑上划分为这三个类加载器；而 `WebApp ClassLoader` 和 `JasperLoader` 都是存在对应的类加载器类的。

当 tomcat 启动时，会创建几种类加载器：

1 Bootstrap 引导类加载器 加载 JVM 启动所需的类，以及标准扩展类（位于 `jre/lib/ext` 下）

2 System 系统类加载器 加载 tomcat 启动的类，比如 `bootstrap.jar`，通常在 `catalina.bat` 或者 `catalina.sh` 中指定。位于 `CATALINA_HOME/bin` 下。

3 Common 通用类加载器 加载 tomcat 使用以及应用通用的一些类，位于 `CATALINA_HOME/lib` 下，比如 `servlet-api.jar`

4 webapp 应用类加载器 每个应用在部署后，都会创建一个唯一的类加载器。该类加载器会加载位于 `WEB-INF/lib` 下的 jar 文件中的 class 和 `WEB-INF/classes` 下的 class 文件。

栈帧

见 PPT

栈帧详解

见 PPT

方法调用详解

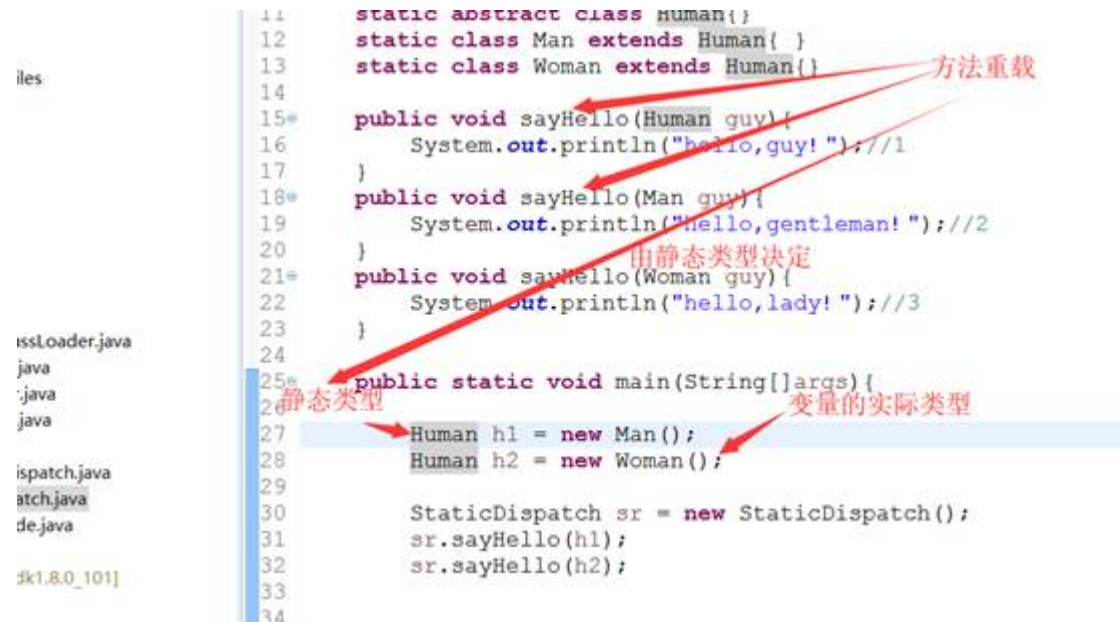
解析

调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析。

在 Java 语言中符合“编译期可知，运行期不可变”这个要求的方法，主要包括静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了它们都不可能通过继承或别的方式重写其他版本，因此它们都适合在类加载阶段进行解析。

静态分派

多见于方法的重载。



```
11 static abstract class Human{
12 static class Man extends Human{ }
13 static class Woman extends Human{ }
14
15 public void sayHello(Human guy){
16     System.out.println("hello,guy! "); //1
17 }
18 public void sayHello(Man guy){
19     System.out.println("Hello,gentleman! "); //2
20 }
21 public void sayHello(Woman guy){
22     System.out.println("hello,lady! "); //3
23 }
24
25 public static void main(String[] args){
26     Human h1 = new Man();
27     Human h2 = new Woman();
28
29     StaticDispatch sr = new StaticDispatch();
30     sr.sayHello(h1);
31     sr.sayHello(h2);
32 }
33
34
```

Annotations in the image:

- 方法重载 (Method Overload) points to the three `sayHello` methods.
- 由静态类型决定 (Determined by static type) points to the parameter `Human guy` in the first `sayHello` method.
- 静态类型 (Static Type) points to the `Human` type in the variable declarations `Human h1` and `Human h2`.
- 变量的实际类型 (Actual Type of the variable) points to the `new Man()` and `new Woman()` instantiations.

“Human”称为变量的静态类型（Static Type），或者叫做的外观类型（Apparent Type），后面的“Man”则称为变量的实际类型（Actual Type），静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

代码中定义了两个静态类型相同但实际类型不同的变量，但虚拟机（准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，因此，在编译阶段，Javac 编译器会根据参数的静态类型决定使用哪个重载版本，所以选择了 `sayHello (Human)` 作为调用目标。所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。静态分派的典型应用是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。

动态分派

静态类型同样都是 `Human` 的两个变量 `man` 和 `woman` 在调用 `sayHello ()` 方法时执行了不同的行为，并且变量 `man` 在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同。

在实现上，最常用手段就是为类在方法区中建立一个虚方法表。虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口地址。PPT 图中，Son 重写了来自 `Father` 的全部方法，因此 `Son` 的方法表没有指向 `Father` 类型数据的箭头。但

是 Son 和 Father 都没有重写来自 Object 的方法，所以它们的方法表中所有从 Object 继承来的方法都指向了 Object 的数据类型。

基于栈的字节码解释执行引擎

Java 编译器输出的指令流，基本上]是一种基于栈的指令集架构，指令流中的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与

基于寄存器的指令集，最典型的就是 x86 的二地址指令集，说得通俗一些，就是现在我们主流 PC 机中直接支持的指令集架构，这些指令依赖寄存器进行工作。

举个最简单的例子，分别使用这两种指令集计算“1+1”的结果，基于栈的指令集会是这样子的：

```
iconst_1
```

```
iconst_1
```

```
iadd
```

```
istore_0
```

两条 iconst_1 指令连续把两个常量 1 压入栈后，iadd 指令把栈顶的两个值出栈、相加，然后把结果放回栈顶，最后 istore_0 把栈顶的值放到局部变量表的第 0 个 Slot 中。

如果基于寄存器，那程序可能会是这个样子：

```
mov eax, 1
```

```
add eax, 1
```

mov 指令把 EAX 寄存器的值设为 1，然后 add 指令再把这个值加 1，结果就保存在 EAX 寄存器里面。

基于栈的指令集主要的优点就是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。栈架构指令集的主要缺点是执行速度相对来说会稍慢一些。所有主流物理机的指令集都是寄存器架构也从侧面印证了这一点。

