**虚拟机类加载机制**

**虚拟机的类加载机制**

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型

Java语言与那些在编译时需要进行连接工作的语言不同，在java语言里面，类型的加载、连接和初始化都是在程序运行期间完成的，稍微增加了一些性能开销，但为java应用程序提供了高度了灵活性，java里天生可以动态扩展的语言特性就是依赖运行期间动态加载和动态连接完成的

**类加载的时机**

类从被加载到虚拟机内存到卸载出内存为止，整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载，其中验证、准备、解析共同构成连接操作。

加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的：类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定；解析阶段在某些情况可以在初始化阶段之后再开始，这是为了支持java语言的运行时绑定(也称为动态绑定或晚期绑定)。

什么情况下需要开始类加载过程的第一个阶段：加载？Java虚拟机规范中并没有进行强制约束，这点可以交给虚拟机的具体实现来自由把握。但对于初始化阶段，虚拟机规范则是明确规定了有且只有5种情况必须对类进行“初始化”

(1)遇到new、getstatic、putstatic或invokestatic四条字节码指令时，如果类没有进行初始化，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：使用new关键字实例化对象的时候、读取或设置类的一个静态字段的时候(被final修饰、已在编译期把结果放入常量池的静态字段除外)，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的1.7动态语言支持时，如果一个Java.lang.invoke.MethodHandle实例最后的解析结果是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

接口的加载过程和类加载过程稍微有一些不同，接口也有初始化过程，这点和是一致的，接口中不能有static{}语句块，但编译器仍然会为接口生成”<clinit>()”类构造器，用于初始化接口中所定义的成员变量。当一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用父接口的时候才会被初始化

**类加载的过程**

**1.加载**

加载是类加载过程的一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

(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

加载阶段完成之后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构。然后在内存中实例化一个java.lang.Class类的对象(并没有明确规定是在java堆中)，对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里面

**数组类的加载**

数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。但数组类与类加载器仍然有很密切的联系，因为数组类的元素类型(指的是数组去掉所有维度的类型)最终是要靠类加载器去创建，一个数组类C的创建过程遵循以下规则：

(1)如果数组的组件类型是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识

(2)如果数组的组件类型不是引用类型，Java虚拟机会把数组C标记为与引导类加载器关联

(3)数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public

**2.连接过程之验证**

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

验证阶段大致会完成下面四个阶段的检验动作：文本格式验证、元数据验证、字节码验证、符号引用验证

**1.文件格式验证**

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，这一阶段可能包括下面这些验证点：

(1)是否以魔数0xCAFEBABE开头

(2)主次版本号是否在当前虚拟机处理范围之内

(3)常量池的常量中是否有不被支持的常量类型(检查常量tag标志)

(4)指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量

(5)CONSTANT\_Utf8\_info型的常量中是否有不符合UTF8编码的数据

(6)Class文件中各个部分及文件本身是否有被删除的或附加的其他信息

…

该验证阶段的主要目的是保证输入的字节流能正确地解析出存储于方法区之内，格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制方法的，只有通过这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的三个验证都是基于方法区的存储结构进行的，不会再直接操作字节流

**2.元数据验证**

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的符合Java语言规范的要求，这个阶段可能包括的验证点如下：

(1)这个类是否有父类

(2)如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法

(3)这个类的父类是否继承了不允许被继承的类(被final修饰的类)

(4)类中的字段、方法是否与父类产生矛盾

第二阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合Java语言规范的元数据信息

**3.字节码验证**

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析、保证被校验类的方法在运行时不会做出危害虚拟机安全的事件

**4.符号引用验证**

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个动作将在连接的第三阶段—解析阶段中发生。符号引用验证可以看做是对类自身以外的信息进行匹配性校验，通常需要校验一下内容：

(1)符号引用中通过字符串描述的全限定名是否能找到对应的类

(2)在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段

(3)符号引用中的类、字段、方法的访问性是否可以被当前类访问

符号验证的目的是确保解析动作能够正常执行。验证阶段是一个非常重要的、但不一定是必要的阶段。

**2.连接阶段之准备**

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，这些变量所用的内存都将在方法区中进行分配。这里有两个容易产生混淆的概念需要强调一下：

(1)这时候进行内存分配的仅仅包括类变量，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

(2)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。相对地还会有一些“特殊情况”：如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，假设上面类变量value定义变为：

Public static final int value=123;

编译时将会为value生成ConstantValue属性，在准备阶段虚拟机将会根据ConstantValue的设置将value赋值为123.

**2.连接阶段之解析**

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。符号引用在class文件中以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现。

解析阶段中所说的直接引用和符号引用有什么关联？

(1)符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们所接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中

(2)直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用和虚拟机实现的内存布局是相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标不定已经在内存中存在。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行，分别对应于常量池的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info、CONSTANT\_MethodType\_info、CONSTANT\_MethodHandle\_info、CONSTANT\_InvokeDynamic\_info七种常量类型。下面针对类、接口、字段、类方法解析进行阐述：

(1)类或接口的解析

(2)字段解析

(3)类方法解析

(4)接口方法解析

**3.初始化**

类初始化阶段是类加载过程的最后一步，只有到初始化阶段，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：初始化阶段是执行类构造器<clinit>()方法的过程。关于类构造器<clinit>()方法的特点：

(1)<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，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量将访问不到(java支持向前引用：在定义类、接口、方法、变量之前使用它们)

(2)<clinit>()方法与类的构造函数(或者说实例构造器<init()方法>)不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕，因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是java.lang.Object

(3)由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作

(4)<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，那么可以不为这个类生成<clinit>()方法

(5)接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都能生成<clinit>()方法，但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法，只有当父类接口中定义的变量使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行<clinit>（）方法

(6)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，那么只有一个线程去执行这个类的<clinit>()方法，其他线程都会被阻塞等待，直到活动线程执行<clinit>()方法完毕

**类加载器**

**1.类与类加载器**

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。

**2.类加载器分类**

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，并且全部都继承自抽象类java.lang.ClassLoader

绝大部分Java程序都会使用到以下3种系统提供的类加载器：

(1)启动类加载器(Bootstrap ClassLoader):该类加载器负责将存放在<JAVA\_HOME>\lib目录中的、并且被虚拟机识别的类库加载到虚拟机内存中，如rt.jar。

(2)扩展类加载器(Extension ClassLoader):该加载器由sun.misc.Launcher$ExtClassLoader实现，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

(3)应用程序类加载器(Application ClassLoader):该类加载器由sun.misc.Launcher$ApplicationClassLoader实现，也称系统类加载器

**3.双亲委派模型**

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求时，子加载器才会去尝试自己加载。

**为什么要使用双亲委派模型？**

双亲委派模型对于保证Java程序的稳定性运作很重要，但它的实现却非常简单，实现双亲委派的代码都集中在java.lang.ClassLoader的loadClass()方法之中

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类在程序中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，并没有放在程序的ClassPath中，那系统中的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

**Q1:接口中为什么不能定义静态方法？**

A1:接口中的方法都是没有方法体的，假设将接口中的方法都定义成public static abstract void method(),因为是静态方法，那么可以直接通过类名.method()调用，但又因为没有方法体，添加static毫无意义

**Q2:接口中为什么不能使用静态语句块？**

A2:

**Q3:接口中的常量为什么定义成public static final？**

A3:《thinking in java》中说：接口规定了可对一个特定的对象发出哪些请求，即接口是用来描述一个对象有哪些方法的，所以接口不应该涉及任何数据相关的东西，因此个人认为就算接口中不容许有变量存在也是合理的，但JDK的设计者容许接口中存在public static final 的变量也是可以理解的，因为这种类型的变量在编译时会执行宏替换，直接替换成对应的常量值，这就和你没有定义常量变量，直接将值写在使用常量变量的地方一样

**Q4：执行宏替换的条件？**

A4：使用final修饰；在定义final变量时指定了初始值；初始值在编译期间就能确定