## 1. 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，这就是虚拟机的类加载机制。

## 2. 类加载的时机。

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）。

## 3. 有且只有 5 种情况必须立即对类进行“初始化”。

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。

## 4. 主动引用和被动引用。

这 5 种场景中的行为称为对一个类进行主动引用。除此之外，所有引用类的方式都不会触发初始化，称为被动引用。

## 5. 接口和类初始化的区别。

当一个类在初始化时，要求其父类全部都已经初始化过了，但是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，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

## 6. 类加载的过程。

**加载：**“加载”是“类加载”（Class Loading）过程的一个阶段。在加载阶段，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：**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。
**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。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并存储于方法区之内。
**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，以保证其描述的信息符合 Java 语言规范的要求。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，这个转化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（常量池中的各种符号引用）的信息进行匹配性校验。符号引用验证的目的是确保解析动作能正常执行，如果无法通过符号引用验证，那么将会抛出一个 java.lang.IncompatibleClassChangeError 异常的子类。

**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，这些变量所使用的内存都将在方法区中进行分配。

**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。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。

**初始化：**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（或者说是字节码）。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。

## 7. 数组类的加载阶段过程。

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关系，因为数组类的元素类型（Element Type，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（下面简称为 C）创建过程就遵循以下规则：
1）如果数组的组件类型（Component Type，指的是数组去掉一个维度的类型）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（这一个类必须与类加载器一起确定唯一性）。
2）如果数组的组件类型不是引用类型（例如 int[]数组），Java 虚拟机将会把数组 C 标记为与引导类加载器关联。
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。

## 8. public static int value=123；初始化操作步骤。

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。

## 9. ConstantValue 属性的作用。

在“通常情况”下初始值是零值，那相对的会有一些“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 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。

## 10. 符号引用和直接引用。

**符号引用（Symbolic References）：**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，但是它们能接受的符号引用必须都是一致的，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。
**直接引用（Direct References）：**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。直接引用是和虚拟机实现的内存布局相关的，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，那引用的目标必定已经在内存中存在。

## 11. invokedynamic 指令。

当碰到某个前面已经由 invokedynamic 指令触发过解析的符号引用时，并不意味着这个解析结果对于其他 invokedynamic 指令也同样生效。因为 invokedynamic 指令的目的本来就是用于动态语言支持（目前仅使用 Java 语言不会生成这条字节码指令），它所对应的引用称为“动态调用点限定符”（Dynamic Call SiteSpecifier），这里“动态”的含义就是必须等到程序实际运行到这条指令的时候，解析动作才能进行。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

## 12. 类或接口的解析。

假设当前代码所处的类为 D，如果要把一个从未解析过的符号引用 N 解析为一个类或接口 C 的直接引用，那虚拟机完成整个解析的过程需要以下 3 个步骤：
1）如果 C 不是一个数组类型，那虚拟机将会把代表 N 的全限定名传递给 D 的类加载器去加载这个类 C。在加载过程中，由于元数据验证、字节码验证的需要，又可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就宣告失败。
2）如果 C 是一个数组类型，并且数组的元素类型为对象，也就是 N 的描述符会是类似“[Ljava/lang/Integer”的形式，那将会按照第 1 点的规则加载数组元素类型。如果 N 的描述符如前面所假设的形式，需要加载的元素类型就是“java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。
3）如果上面的步骤没有出现任何异常，那么 C 在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认 D 是否具备对 C 的访问权限。如果发现不具备访问权限，将抛出 java.lang.IllegalAccessError 异常。

## 13. 字段解析。

要解析一个未被解析过的字段符号引用，首先将会对字段表内 class_index [2] 项中索引的 CONSTANT_Class_info 符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失败。如果解析成功完成，那将这个字段所属的类或接口用 C 表示，虚拟机规范要求按照如下步骤对 C 进行后续字段的搜索。
1）如果 C 本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
2）否则，如果在 C 中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
3）否则，如果 C 不是 java.lang.Object 的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
4）否则，查找失败，抛出 java.lang.NoSuchFieldError 异常。如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出 java.lang.IllegalAccessError 异常。

## 14. 类方法解析。

类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的 class_index [3] 项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用 C 表示这个类，接下来虚拟机将会按照如下步骤进行后续的类方法搜索。
1）类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现 class_index 中索引的 C 是个接口，那就直接抛出 java.lang.IncompatibleClassChangeError 异常。
2）如果通过了第 1 步，在类 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
3）否则，在类 C 的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

4）否则，在类 C 实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类 C 是一个抽象类，这时查找结束，抛出 java.lang.AbstractMethodError 异常。
5）否则，宣告方法查找失败，抛出 java.lang.NoSuchMethodError。
最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证，如果发现不具备对此方法的访问权限，将抛出 java.lang.IllegalAccessError 异常。

## 15. 接口方法解析。

接口方法也需要先解析出接口方法表的 class_index [4] 项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用 C 表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索。
1）与类方法解析不同，如果在接口方法表中发现 class_index 中的索引 C 是个类而不是接口，那就直接抛出 java.lang.IncompatibleClassChangeError 异常。
2）否则，在接口 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
3）否则，在接口 C 的父接口中递归查找，直到 java.lang.Object 类（查找范围会包括 Object 类）为止，看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
4）否则，宣告方法查找失败，抛出 java.lang.NoSuchMethodError 异常。由于接口中的所有方法默认都是 public 的，所以不存在访问权限的问题，因此接口方法的符号解析应当不会抛出 java.lang.IllegalAccessError 异常。

## 16. ＜clinit＞（）方法。

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（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，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，在前面的静态语句块可以赋值，但是不能访问。
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，那么编译器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会生成 < 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。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 < clinit > （）方法。

## 17. 比较两个类是否“相等”。

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。如果没有注意到类加载器的影响，在某些情况下可能会产生具有迷惑性的结果。

## 18. 加载器介绍。

启动类加载器（Bootstrap ClassLoader）：前面已经介绍过，这个类将器负责将存放在 < 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 $App-ClassLoader 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的类加载器。

## 19. 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，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。双亲委派模型对于保证 Java 程序的稳定运作很重要，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，先检查是否已经被加载过，若没有加载则调用父加载器的 loadClass（）方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass（）方法进行加载。