[阿里面试题，深入理解Java类加载机制](https://blog.csdn.net/tzllxya/article/details/90412771)

**类的生命周期**

包括以下 7 个阶段：

* **加载（Loading）**
* **验证（Verification）**
* **准备（Preparation）**
* **解析（Resolution）**
* **初始化（Initialization）**
* 使用（Using）
* 卸载（Unloading）

其中解析过程在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的动态绑定。

**类初始化时机**

虚拟机规范中并没有强制约束何时进行加载，但是规范严格规定了有且只有下列五种情况必须对类进行初始化（加载、验证、准备都会随着发生）：

* 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。最常见的生成这 4 条指令的场景是：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。
* 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行初始化，则需要先触发其初始化。
* 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。
* 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；
* 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为 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；

以上 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。被动引用的常见例子包括：

* 通过子类引用父类的静态字段，不会导致子类初始化。

System.out.println(SubClass.value); *// value 字段在 SuperClass 中定义*

* 通过数组定义来引用类，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚拟机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

SuperClass[] sca = new SuperClass[10];

* 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

System.out.println(ConstClass.HELLOWORLD);

**类加载过程**

包含了加载、验证、准备、解析和初始化这 5 个阶段。

**1. 加载**

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

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存储结构。
3. 在内存中生成一个代表这个类的 Class 对象，作为方法区这个类的各种数据的访问入口。

其中二进制字节流可以从以下方式中获取：

* 从 ZIP 包读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础。
* 从网络中获取，这种场景最典型的应用是 Applet。
* 运行时计算生成，这种场景使用得最多得就是动态代理技术，在 java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 的代理类的二进制字节流。
* 由其他文件生成，典型场景是 JSP 应用，即由 JSP 文件生成对应的 Class 类。
* 从数据库读取，这种场景相对少见，例如有些中间件服务器（如 SAP Netweaver）可以选择把程序安装到数据库中来完成程序代码在集群间的分发。 ...

**2. 验证**

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。

* 文件格式验证：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。
* 元数据验证：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。
* 字节码验证：通过数据流和控制流分析，确保程序语义是合法、符合逻辑的。
* 符号引用验证：发生在虚拟机将符号引用转换为直接引用的时候，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。

**3. 准备**

类变量是被 static 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存。

实例变量不会在这阶段分配内存，它将会在对象实例化时随着对象一起分配在 Java 堆中。（实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次）

初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

public static int value = 123;

如果类变量是常量，那么会按照表达式来进行初始化，而不是赋值为 0。

public static final int value = 123;

**4. 解析**

将常量池的符号引用替换为直接引用的过程。

**5. 初始化**

初始化阶段才真正开始执行类中的定义的 Java 程序代码。初始化阶段即虚拟机执行类构造器 <clinit>() 方法的过程。

在准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，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它资源。

<clinit>() 方法具有以下特点：

* 是由编译器自动收集类中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，编译器收集的顺序由语句在源文件中出现的顺序决定。特别注意的是，静态语句块只能访问到定义在它之前的类变量，定义在它之后的类变量只能赋值，不能访问。例如以下代码：

1. public class Test {
2. static {
3. i = 0; *// 给变量赋值可以正常编译通过*
4. System.out.print(i); *// 这句编译器会提示“非法向前引用”*
5. }
6. static int i = 1;
7. }

* 与类的构造函数（或者说实例构造器 <init>()）不同，不需要显式的调用父类的构造器。虚拟机会自动保证在子类的 <clinit>() 方法运行之前，父类的 <clinit>() 方法已经执行结束。因此虚拟机中第一个执行 <clinit>() 方法的类肯定为 java.lang.Object。
* 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于子类的变量赋值操作。例如以下代码：

1. static class Parent {
2. public static int A = 1;
3. static {
4. A = 2;
5. }
6. }
8. static class Sub extends Parent {
9. public static int B = A;
10. }
12. public static void main(String[] args) {
13. System.out.println(Sub.B); *// 输出结果是父类中的静态变量 A 的值，也就是 2。*
14. }

* <clinit>() 方法对于类或接口不是必须的，如果一个类中不包含静态语句块，也没有对类变量的赋值操作，编译器可以不为该类生成 <clinit>() 方法。
* 接口中不可以使用静态语句块，但仍然有类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。
* 虚拟机会保证一个类的 <clinit>() 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 <clinit>() 方法，其它线程都会阻塞等待，直到活动线程执行 <clinit>() 方法完毕。如果在一个类的 <clinit>() 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

**类加载器**

实现类的加载动作。在 Java 虚拟机外部实现，以便让应用程序自己决定如何去获取所需要的类。

**类与类加载器**

两个类相等：类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。

这里的相等，包括类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为 true，也包括使用 instanceof 关键字做对象所属关系判定结果为 true。

**类加载器分类**

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

* 启动类加载器（Bootstrap ClassLoader），这个类加载器用 C++ 实现，是虚拟机自身的一部分；
* 所有其他类的加载器，这些类由 Java 实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

* 启动类加载器（Bootstrap ClassLoader）此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启动类加载器，直接使用 null 代替即可。
* 扩展类加载器（Extension ClassLoader）这个类加载器是由 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将 <JAVA\_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
* 应用程序类加载器（Application ClassLoader）这个类加载器是由 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。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。

**双亲委派模型**

应用程序都是由三种类加载器相互配合进行加载的，如果有必要，还可以加入自己定义的类加载器。

下图展示的类加载器之间的层次关系，称为类加载器的双亲委派模型（Parents Delegation Model）。该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有自己的父类加载器。这里类加载器之间的父子关系一般通过组合（Composition）关系来实现，而不是通过继承（Inheritance）的关系实现。

A screenshot of a cell phone

Description automatically generated

**（一）工作过程**

一个类加载器首先将类加载请求传送到父类加载器，只有当父类加载器无法完成类加载请求时才尝试加载。

**（二）好处**

使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而是的基础类得到统一。

例如 java.lang.Object 存放在 rt.jar 中，如果编写另外一个 java.lang.Object 的类并放到 ClassPath 中，程序可以编译通过。因为双亲委派模型的存在，所以在 rt.jar 中的 Object 比在 ClassPath 中的 Object 优先级更高，因为 rt.jar 中的 Object 使用的是启动类加载器，而 ClassPath 中的 Object 使用的是应用程序类加载器。正因为 rt.jar 中的 Object 优先级更高，因为程序中所有的 Object 都是这个 Object。

**（三）实现**

以下是抽象类 java.lang.ClassLoader 的代码片段，其中的 loadClass() 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 ClassNotFoundException，此时尝试自己去加载。

1. public abstract class ClassLoader {
2. *// The parent class loader for delegation*
3. private final ClassLoader parent;
5. public Class<?> loadClass(String name) throws ClassNotFoundException {
6. return loadClass(name, false);
7. }
9. protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
10. synchronized (getClassLoadingLock(name)) {
11. *// First, check if the class has already been loaded*
12. Class<?> c = findLoadedClass(name);
13. if (c == null) {
14. try {
15. if (parent != null) {
16. c = parent.loadClass(name, false);
17. } else {
18. c = findBootstrapClassOrNull(name);
19. }
20. } catch (ClassNotFoundException e) {
21. *// ClassNotFoundException thrown if class not found*
22. *// from the non-null parent class loader*
23. }
25. if (c == null) {
26. *// If still not found, then invoke findClass in order*
27. *// to find the class.*
28. c = findClass(name);
29. }
30. }
31. if (resolve) {
32. resolveClass(c);
33. }
34. return c;
35. }
36. }
38. protected Class<?> findClass(String name) throws ClassNotFoundException {
39. throw new ClassNotFoundException(name);
40. }
41. }

**自定义类加载器实现**

FileSystemClassLoader 是自定义类加载器，继承自 java.lang.ClassLoader，用于加载文件系统上的类。它首先根据类的全名在文件系统上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 defineClass() 方法来把这些字节代码转换成 java.lang.Class 类的实例。

java.lang.ClassLoader 类的方法 loadClass() 实现了双亲委派模型的逻辑，因此自定义类加载器一般不去重写它，而是通过重写 findClass() 方法。

1. public class FileSystemClassLoader extends ClassLoader {
3. private String rootDir;
5. public FileSystemClassLoader(String rootDir) {
6. this.rootDir = rootDir;
7. }
9. protected Class<?> findClass(String name) throws ClassNotFoundException {
10. byte[] classData = getClassData(name);
11. if (classData == null) {
12. throw new ClassNotFoundException();
13. } else {
14. return defineClass(name, classData, 0, classData.length);
15. }
16. }
18. private byte[] getClassData(String className) {
19. String path = classNameToPath(className);
20. try {
21. InputStream ins = new FileInputStream(path);
22. ByteArrayOutputStream baos = new ByteArrayOutputStream();
23. int bufferSize = 4096;
24. byte[] buffer = new byte[bufferSize];
25. int bytesNumRead;
26. while ((bytesNumRead = ins.read(buffer)) != -1) {
27. baos.write(buffer, 0, bytesNumRead);
28. }
29. return baos.toByteArray();
30. } catch (IOException e) {
31. e.printStackTrace();
32. }
33. return null;
34. }
36. private String classNameToPath(String className) {
37. return rootDir + File.separatorChar
38. + className.replace('.', File.separatorChar) + ".class";
39. }
40. }