链滴

Java 类加载机制

作者：someone33881

原文链接：https://ld246.com/article/1542543007844

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20181110.jpg?imageView2/1/w/960/h/520/interlace/1/q/100">
关键词：**类的装载、****类生命周期、类加载过程、**类装载器、双亲委派模型</st
ong>
一、什么类的装载
    在很多其他文章或书中，一般都用“加载”这个词语，在这里我们用“<strong
装载”进行区分，以更好地加强理解；
    在这里，装载为表示 JVM 读取 class 文件二进制数据并生成 Class 对象的过程<
p>
    所谓装载类，就是 JVM 将类的.class 文件中二进制数据
取到内存（运行时数据区的方法区）中，并在内存（堆区）中创建 java.lang.Class 对象的过程。其
，堆区的 Class 对象是封装了类在方法区内的数据结构，向 java 程序员提供了
作方法区内数据结构的接口，为类装载的最终产物。
    装载类的时机：一个类的装载，并不需要等到
类被使用时才被装载，JVM 允许类装载器预先装载可能将要被使用的类；且在预加载过程
若.class 文件缺失或错误，类装载器并非一定会报告错误，只有该类被程序主动使用时才会报告错误（
inkageError）。
    装载类的途径：本地系统中的.class 文件、网络中下载的.c
ass 文件、zip/jar 等归档文件中加载的.class 文件、专有数据库中提取的.class 文件、java 源文件动
编译生成的.class 文件
注意：
    确切地说，上述描述的所谓类的装载（常常也会被称做加载）只
类生命周期整个类加载过程（因此，为了与类生命周期的全部过程-加载过程进行区分，本
称之为“装载”）中的第一个阶段，也即是获取类的二进
字节流的一个动作，称之为加载阶段；；
    注意区分用词“类的装载阶段与类的加载过程”！！
二、类生命周期
    类生命周期，也即类的加载过程包
装载(也有称为加载阶段，以便与整个加载过程进行区分！)、验证、准备、解析、初始化</s
rong>五个阶段；
    其中，装载阶段、验证、准备、初始化这
个阶段开始的顺序是确定的(注意，这里并不是说按顺序进行或完成，仅是说开
的顺序，通常进行是交叉混合的也即在一个阶段的执行过程中调用或激活另一
阶段)；而解析，则可以是在初始化阶段之后才开始，这是为了支持 java 的动态绑定(运行
绑定)
    1、装载阶段
    也即前文第一部分中所描述的过程，简言之“查找并读取类的二进制
据到内存静态方法区，并在内存堆区中创建 Class 对象”；JVM 主要完成三件事情：

读取二进制字节流数据（by 类的全限定名）
将字节流所代表的静态存储结构转化为运行时数据结构(-> 静态方法区)<
li>
生成 Class 对象(-> 堆区)

PS：开发人员可以使用系统提供的类装载器完成装载，也可以自定义类装载器完成装载（一般就
是自定义一个类文件二进制数据的读取功能，如对网络传输中加密的类文件），确切地说是自定义类
载中的读取方法！
    2、连接阶段（验证、准备、解析）
    （1）验证：确保二进制字节流数据符合 JVM
求，不会存在危害 JVM 虚拟机的安全问题
    主要完成四个检验动作，验证文件格式（cla
s 文件格式规范）、验证元数据（字节码描述的信息是否符合 java 语言规范）
验证字节码（程序语义是否合法、符合逻辑）、验证符号引用
g>（确保后续解析动作能够正确执行）

    验证阶段非常重要，但不是必须的，可通过**-Xverifynone 参数关闭验证以缩类加载时间**
    （2）准备：在方法区内为类变量(静态变量)分内存，并设置初始值！注意：

仅是类变量，而非实例变量（实例变量为对象实例化与对象一起在堆内存中分配）
初始值****通常是变量所属数据类型默认的零值(0、0、null、false 等)，而非 java 代码中显示赋予的值
如果是静态常量(final 和 static 修饰），也即该类字段的字段属性表中存在 ConstantValue 属性，那么就会被初始化为 ConstantValue 属性所指定的值因此，静态常量在声明时必须显示地赋值，否则编译无法时不能通过 => 编译时：基本数据类型类变量和全局变量可以不显示赋值，局部变量必须显示地为其赋值），也即 static final 常量在编译就将其结果放入了调用它的类的常量池中

    （3）解析：将常量池中的符号引用替换为直引用的过程

该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、调用点限定符这 7 类符号引用进行
符号引用：一组描述目标的符号，可以是任意字面量
直接引用：直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄

    3、初始化阶段
    为类变量赋予正确的初始值（声明时指定初始值；或静态代码块中为其指定赋值）
    初始化时机：类被主动使用时，才会执行初始化动作，如

创建类的实例时（new）
访问某个类或接口的静态变量，或对该静态变量赋值时
调用某个类的静态方法时
反射
初始化某个类时，其父类也会被初始化
JVM 启动时被标明为启动类的类（如 Test 类），或直接使用 java.exe 命令运行的某主类

    类初始化的步骤：
    （1）、若该类还没被加载和连接，则先加载该类
    （2）、若该类的直接父类还没被初始化，则先初始化父类
    （3）、若该类中有初始化语句，则依次执行该类的初始化语句
    其中，1、2、3 属于类的加载过程
    4、使用 - 卸载 - 结束生命周期
三、类装载器（--对应二的 1-加载阶段）
    java 的父类加载器并不是通过继承关系实现的，而是通过组合实现的/strong>
**   对于 Hotspot 虚拟机而言，类加载器分两类：**
（1）启动类加载器（C++ 实现，其他虚拟机也有是 Java 实现的），为虚拟机的一部分；
（2）所有其他的类加载器（Java 实现），独立于虚拟机之外且全部继承于抽象类 ClassLoader 均由启动类加载器加载到内存中之后才能去加载其他的类
  ** 对于 javaer 而言，类加载器分三类：**
（1）启动类加载器 Bootstrap ClassLoader：由 C++ 实现（Hotspot）负责加载路径 $JAVA_HOME**\jre\lib 或被-Xbootclasspath 参数定的路径，并且能被虚拟机识别的类库（如rt.jar，包 java.下的所有类*）启动类加载器无法被 java 程序直接引用。
（2）扩展类加载器 ExtClassLoader：由 sun.misc.Launcher<span class="

anguage-math">ExtClassLoader实现，负责加载路径JAVA_HOME\jre\lib\ex或被 java.ext.dirs 系统变量指定的路径中的所有类库（*包 javax.下的所有类）；可以直接使用扩展类加载器。
（3）**应用类加载器 AppClassLoader：又 sun.misc.Launcher$AppClassLoader 现，负责加载用户路径（ClassPath）**所指定的类；可以直接使用应用类加载器，默认类加载器
=> 自定义类加载器：一般都是通过继承 ClassLoader 类，重写 findClass 方法，核心是对字节码文件的获取
    在执行非置信代码之前，对类的数字签名进行自动验证；
    动态创建需要的定制化构建类；
    特定的 Class 二进制文件源加载的，如网络或数据库等
四、类装载机制（--对应二的 1-装载载阶段）
    1、父类委托 - 双亲委派模型：接收到类加载请求的类加器，首先自己不会去尝试加载该类而是将请求委托给父加载器去完成，依次向上；因此，所有类加载求最终都会被传递到顶层的启动类加载器中，若父加载器在其加载路径中找不到所需要的类，子加载才会尝试从自己的类路径中去加载该类。
<p>   注：其实所谓的双亲模型，指的就是在加载类的时候要<strong先经过父类的判断是否存在。</p>
    2、**全盘负责：**当一个类加载器负责某个类 Class 的加载时，那么该 Class 依赖的和引用的其他 Class 也将由该类加载器负责加载（除非被显示地使用另一个类加载器加载）【全盘负责仍然是基于父类委托的，也即如果该类加载器需要加载 Class 所依赖的 ClassA 那么也是先于父类委托先通过父类判断是否已经加载 ClassA，然后再决定是否由该类加载器去加载的】
    3、**缓存机制：**也即，所有被加载过的 Class 都会被缓存；当程需要使用某个 Class 时，则类加载器先从缓存区去寻找，若缓存区中不存在才会去读取二制数据并将其转换成 Class 对象且存入缓存中（=> 因此，java 程序修改了一个 Class，必须重启 VM，修改才会生效！）
五、双亲委派模型（--对应二的 1-装载阶段）
    也即第四部分中的父类委托
双亲委派模型的过程：
    当 AppClassLoader 加载一个 ClassA 时，会先把该类的加载请求委派给父类载器 ExtClassLoader；
    当 ExtClassLoader 收到需要加载的 ClassA 时，会先把该类的加载请求委派给类加载 BoostrapClassLoader；
   当 BoostrapClassLoader 收到需要加载的 ClassA 时，如果在 $JAVA_HOME/jr/lib 下未找到 ClassA 即加载失败，则会使用 ExtClassLoader 去加载；
    若 ExtClassLoader 也加载失败，则会使用 AppClassLoader 加载（仍然加载败，则抛 ClassNotFoundException 异常）。
双亲委派模型的意义：
    重复问题：防止内存中出现多份同样的字节码
    安全问题：保护核心类库能够被 Bootstrap 和 Ext 所加载，以防止出现自义核心类库被 App 加载而覆盖了掉真正的库，保证 java 程序安全稳定运行。