链滴

JVM_01 类加载子系统

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1601446833569

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20200109.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="JVM架构图">JVM 架构图</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="170ec7cd31581d15.webp" d
ta-src="https://b3logfile.com/file/2020/09/170ec7cd31581d15-61d698dd.webp?imageView2
2/interlace/1/format/jpg">
<h2 id="1-类加载子系统作用">1.类加载子系统作用</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="QQ 截图 20200930113542.p
g" data-src="https://b3logfile.com/file/2020/09/QQ%E6%88%AA%E5%9B%BE202009301135
2-69242ba6.png?imageView2/2/interlace/1/format/jpg">

类加载子系统负责从文件系统或者网络中加载 Class 文件，class 文件在文件开头有特定的文件
识；
ClassLoader 只负责 class 文件的加载，至于它是否可以运行，则由 Execution Engine 决定</li

加载的类信息存放于一块成为方法区的内存空间。除了类信息之外，方法区还会存放运行时常量
信息，可能还包括字符串字面量和数字常量（这部分常量信息是 Class 文件中常量池部分的内存映射

<h3 id="1-1类加载器ClassLoader角色">1.1 类加载器 ClassLoader 角色</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="QQ 截图 20200930113907.p
g" data-src="https://b3logfile.com/file/2020/09/QQ%E6%88%AA%E5%9B%BE202009301139
7-a6d06708.png?imageView2/2/interlace/1/format/jpg">

class file 存在于本地硬盘上，可以理解为设计师画在纸上的模板，而最终这个模板在执行的时候
要加载到 JVM 当中来根据这个文件实例化出 n 个一模一样的实例。
class file 加载到 JVM 中,被称为 DNA 元数据模板,放在方法区。
在.class 文件-> JVM -> 最终成为元数据模板,此过程就要一一个运输工具 (类装载器 Class
oader) ,扮演一个快递员的角色。

<h3 id="1-2加载">1.2 加载</h3>

通过一个类的全限定明获取定义此类的二进制字节流；
将这个字节流所代表的的静态存储结构转化为方法区的运行时数据；
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
/li>

<img src="https://ld246.com/images/img-loading.svg" alt="170ec7daeca85a52.png" dat
-src="https://b3logfile.com/file/2020/09/170ec7daeca85a52-2bc0245d.png?imageView2/2/in
erlace/1/format/jpg">
<h3 id="1-3-链接">1.3 链接</h3>
<h4 id="1-3-1-验证-">1.3.1 验证：</h4>

目的在于确保 Class 文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不
危害虚拟机自身安全。
主要包括四种验证，文件格式验证，源数据验证，字节码验证，符号引用验证。

<h4 id="1-3-2-准备-">1.3.2 准备：</h4>

为类变量分配内存并且设置该类变量的默认初始值，即零值；
这里不包含用 final 修饰的 static，因为 final 在编译的时候就会分配了，准备阶段会显式初始化

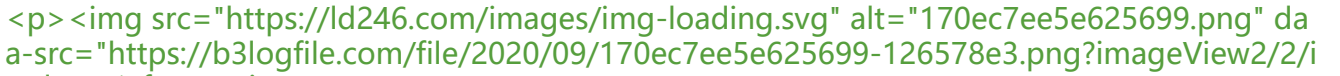
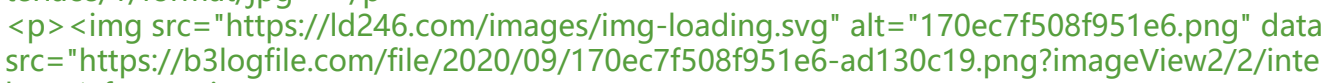
这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到
java 堆中。

<h4 id="1-3-3-解析-">1.3.3 解析：</h4>

将常量池内的符号引用转换为直接引用的过程。
事实上，解析操作网晚会伴随着 jvm 在执行完初始化之后再执行
符号引用就是一组符号来描述所引用的目标。符号应用的字面量形式明确定义在《java 虚拟机规》的 class 文件格式中。直接引用就是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
解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的 CONSTAT_Class_info/CONSTANT_Fieldref_info、CONSTANT_Methodref_info 等。

<h3 id="1-4初始化">1.4 初始化</h3>

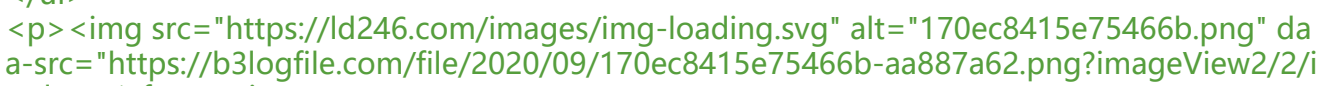
初始化阶段就是执行类构造器方法 clinit（）的过程。
此方法不需要定义，是 javac 编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语合并而来。<code>我们注意到如果没有静态变量c，那么字节码文件中就不会有clinit方法</code></i>

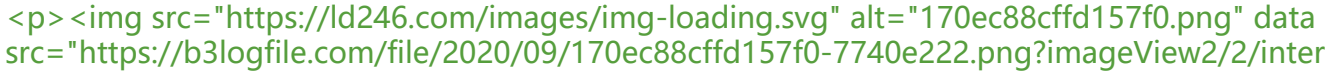
构造器方法中指令按语句在源文件中出现的顺序执行

clinit()不同于类的构造器。（关联：构造器是虚拟机视角下的 init()）
若该类具有父类，jvm 会保证子类的 clinit()执行前，父类的 clinit()已经执行完毕

虚拟机必须保证一个类的 clinit()方法在多线程下被同步加锁。


<h2 id="2-类加载器分类">2.类加载器分类</h2>

JVM 支持两种类型的加载器，分别为引导类加载器（BootStrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）
从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为定义类加载器。
无论类加载器的类型如何划分，在程序中我们最常见的类加载器始终只有三个，如下所示：



ace/1/format/jpg"></p>
<h3 id="2-1-自定义类与核心类库的加载器">2.1 自定义类与核心类库的加载器</h3>

对于用户自定义类来说：使用系统类加载器 AppClassLoader 进行加载
java 核心类库都是使用引导类加载器 BootStrapClassLoader 加载的

<pre><code class="highlight-chroma">/**
 * ClassLoader加载
 */
public class Class oaderTest {
    public static vo d main(String[] args) {
        //获取系统类 载器
        ClassLoader ystemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.pr ntln(systemClassLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2

        //获取其上层 扩展类加载器
        ClassLoader xtClassLoader = systemClassLoader.getParent();
        System.out.pr ntln(extClassLoader);//sun.misc.Launcher$ExtClassLoader@610455d6

        //获取其上层 获取不到引导类加载器
        ClassLoader ootStrapClassLoader = extClassLoader.getParent();
        System.out.pr ntln(bootStrapClassLoader);//null

        //对于用户自 义类来说：使用系统类加载器进行加载
        ClassLoader c assLoader = ClassLoaderTest.class.getClassLoader();
        System.out.pr ntln(classLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2

        //String 类 用引导类加载器进行加载的 -->java核心类库都是使用引导类加载器加载的
        ClassLoader c assLoader1 = String.class.getClassLoader();
        System.out.pr ntln(classLoader1);//null

    }
}
</code></pre>
<h3 id="2-2-虚拟机自带的加载器">2.2 虚拟机自带的加载器</h3>

① 启动类加载器（引导类加载器，BootStrap ClassLoader）

这个类加载使用 C/C++ 语言实现的，嵌套在 JVM 内部
它用来加载 java 的核心库（JAVA_HOME/jre/lib/rt.jar/resources.jar 或 sun.boot.class.path 径下的内容），用于提供 JVM 自身需要的类
并不继承自 java.lang.ClassLoader,没有父加载器
加载拓展类和应用程序类加载器，并指定为他们的父加载器
处于安全考虑，BootStrap 启动类加载器只加载包名为 java、javax、sun 等开头的类

② 拓展类加载器（Extension ClassLoader）

java 语言编写，由 sun.misc.Launcher$ExtClassLoader 实现。
派生于 ClassLoader 类
父类加载器为启动类加载器
从 java.ext.dirs 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 jre/lib/ext 子目录扩展目录）下加载类库。如果用户创建的 JAR 放在此目录下，也会由拓展类加载器自动加

③ 应用程序类加载器（系统类加载器，AppClassLoader）

java 语言编写， 由 sun.misc.Launcher$AppClassLoader 实现。
派生于 ClassLoader 类
父类加载器为拓展类加载器
它负责加载环境变量 classpath 或系统属性 java.class.path 指定路径下的类库
该类加载器是程序中默认的类加载器，一般来说，java 应用的类都是由它完成加载
通过 ClassLoader#getSystemClassLoader()方法可以获取到该类加载器


```
/**
 * 虚拟机自带加载

 */
public class ClassLoaderTest1 {
    public static void main(String[] args) {
        System.out.println("********启动类加载器*********");
        URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs();
        //获取BootStrapClassLoader能够加载的api路径
        for (URL e:urls){
            System.out.println(e.toExternalForm());
        }

        //从上面的路
```

中随意选择一个类 看看他的类加载器是什么
        //Provider位
/jdk1.8.0_171.jdk/Contents/Home/jre/lib/jsse.jar 下，引导类加载器加载它
        ClassLoader c
assLoader = Provider.class.getClassLoader();
        System.out.pr
ntln(classLoader);//null

        System.out.pr
ntln("********拓展类加载器********");
        String extDirs
= System.getProperty("java.ext.dirs");
        for (String pa
h : extDirs.split(";")){
            System.out
println(path);
        }
        //从上面的路
中随意选择一个类 看看他的类加载器是什么:拓展类加载器
        ClassLoader c
assLoader1 = CurveDB.class.getClassLoader();
        System.out.pr
ntln(classLoader1);//sun.misc.Launcher$ExtClassLoader@4dc63996
    }
}

</code></pre>
<h3 id="2-3-用户自定义类加载器">2.3 用户自定义类加载器</h3>
在 Java 的日常应用程序开发中，类的加载几乎是由上述 3 种类加载器相互配合执行的，在必要
，我们还可以自定义类加载器，来定制类的加载方式。
为什么要自定义类加载器？

隔离加载类
修改类加载的方式
拓展加载源
防止源码泄漏

用户自定义类加载器实现步骤:

开发人员可以通过继承抽象类 java. lang. ClassLoader 类的方式，实现自己的类加载器，以满足
些特殊的需求。
在 JDK1.2 之前，在自定义类加载器时，总会去继承 ClassLoader 类并重写 loadClass()方法，
而实现自定义的类加载类，但是在 JDK1.2 之后已不再建议用户去覆盖 loadClass()方法，而是建议把
定义的类加载逻辑写在 findClass()方法中。
在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承 URLClassLoader 类，这
就可以避免自己去编写 findClass()方法及其获取字节码流的方式，使自定义类加载器编写更加简洁。
/li>

<h2 id="3-ClassLoader的常用方法及获取方法">3 ClassLoader 的常用方法及获取方法</h2>
<h3 id="3-1-ClassLoader类-它是一个抽象类-其后所有的类加载器都继承自ClassLoader-不包括启
类加载器-">3.1 ClassLoader 类，它是一个抽象类，其后所有的类加载器都继承自 ClassLoader（不
括启动类加载器）</h3>
<table>
<thead>

<tr>
<th>方法名称</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParent（）</td>
<td>返回该类加载器的超类加载器</td>
</tr>
<tr>
<td>loadClass（String name）</td>
<td>加载名称为 name 的类，返回结果为 java.lang.Class 类的实例</td>
</tr>
<tr>
<td>findClass（String name）</td>
<td>查找名称为 name 的类，返回结果为 java.lang.Class 类的实例</td>
</tr>
<tr>
<td>findLoadedClass（String name）</td>
<td>查找名称为 name 的已经被加载过的类，返回结果为 java.lang.Class 类的实例</td>
</tr>
<tr>
<td>defineClass（String name，byte[] b,int off,int len）</td>
<td>把字节数组 b 中的内容转换为一个 Java 类，返回结果为 java.lang.Class 类的实例</td>
</tr>
<tr>
<td>resolveClass（Class<?> c）</td>
<td>连接指定的一个 java 类</td>
</tr>
</tbody>
</table>
<h3 id="3-2-ClassLoader继承关系">3.2 ClassLoader 继承关系</h3>
拓展类加载器和系统类加载器间接继承于 ClassLoader 抽象类

<h3 id="3-3-获取ClassLoader的途径">3.3 获取 ClassLoader 的途径</h3>

获取当前类的 ClassLoader：class.getClassLoader()
获取当前线程上下文的 ClassLoader：Thread. currentThread() . getContextClassLoder ()
获取系统的 ClassLoader：ClassLoader . getSystemClassLoader ()
获取调用者的 ClassLoader：Drive rManager . getCallerClassLoader ()</i>

<h2 id="4--双亲委派机制">4. 双亲委派机制</h2>
Java 虚拟机对 class 文件采用的是按需加载的方式，也就是说当需要使用该类时才会它的 class 文件加载到内存生成的 class 对象。而且加载某个类的 class 文件时，java 虚拟机采用的双亲委派模式，即把请求交由父类处理，它是一种任务委派模式
<h3 id="4-1-双亲委派机制工作原理">4.1 双亲委派机制工作原理</h3>

<img src="https://ld246.com/images/img-loading.svg" alt="170ec8cbbe16af0c.png" dat
-src="https://b3logfile.com/file/2020/09/170ec8cbbe16af0c-7a09dde6.png?imageView2/2/in
erlace/1/format/jpg">
如图，虽然我们自定义了一个 java.lang 包下的 String 尝试覆盖核心类库中的 String，但是由
双亲委派机制，启动加载器会加载 java 核心类库的 String 类（BootStrap 启动类加载器只加载包名为
java、javax、sun 等开头的类），而核心类库中的 String 并没有 main 方法
<img src="https://ld246.com/images/img-loading.svg" alt="170ec8d5358f0991.png" dat
-src="https://b3logfile.com/file/2020/09/170ec8d5358f0991-3da69d27.png?imageView2/2/in
erlace/1/format/jpg">
<h3 id="4-2-双亲委派机制的优势">4.2 双亲委派机制的优势</h3>

避免类的重复加载
保护程序安全，防止核心 API 被随意篡改

自定义类：java.lang.String
自定义类：java.lang.MeDsh（java.lang 包需要访问权限，阻止我们用包名自定义类）

<img src="https://ld246.com/images/img-loading.svg" alt="170ec8ddb41c5559.png" da
a-src="https://b3logfile.com/file/2020/09/170ec8ddb41c5559-7431e3b8.png?imageView2/2/
nterlace/1/format/jpg">
<img src="https://ld246.com/images/img-loading.svg" alt="QQ 截图 20200930141305.p
g" data-src="https://b3logfile.com/file/2020/09/QQ%E6%88%AA%E5%9B%BE202009301413
5-cf51730d.png?imageView2/2/interlace/1/format/jpg">
<h2 id="5--沙箱安全机制">5. 沙箱安全机制</h2>
自定义 String 类，但是在加载自定义 String 类的时候回率先使用引导类加载器加载
而引导类加载器在加载过程中会先加载 jdk 自带的文件（rt.jar 包中的 java\lang\String.class）,报错
息说没有 main 方法就是因为加载的是 rt.jar 包中的 String 类。这样可以保证对 java 核心源代码的
护，这就是 沙箱安全机制
<h2 id="6-其他">6.其他</h2>

在 jvm 中表示两个 class 对象是否为同一个类存在的两个必要条件

类的完整类名必须一致，包括包名
加载这个类的 ClassLoader（指 ClassLoader 实例对象）必须相同

换句话说，在 jvm 中，即使这两个类对象（class 对象）来源同一个 Class 文件，被同一个虚拟
所加载，但只要加载它们的 ClassLoader 实例对象不同，那么这两个类对象也是不相等的.

<h3 id="对类加载器的引用">对类加载器的引用</h3>
JVM 必须知道一个类型是有启动类加载器加载的还是由用户类加载器加载的。如果一个类型由
户类加载器加载的，那么 jvm 会将这个类加载器的一个引用作为类型信息的会议部分保存
方法区中。当解析一个类型到另一个类型的引用的时候，JVM 需要保证两个类型的加载器
相同的。
<h3 id="类的主动使用和被动使用">类的主动使用和被动使用</h3>
java 程序对类的使用方式分为：主动使用和被动使用

主动使用，分为七种情况：

创建类的实例

访问某各类或接口的静态变量，或者对静态变量赋值

调用类的静态方法

反射 比如 Class.forName(com.dsh.jvm.xxx)

初始化一个类的子类

java 虚拟机启动时被标明为启动类的类

JDK 7 开始提供的动态语言支持：
java.lang.invoke.MethodHandle 实例的解析结果 REF_getStatic、REF_putStatic、REF_invokStatic 句柄对应的类没有初始化，则初始化

除了以上七种情况，其他使用 java 类的方式都被看作是对类的被动使用，都不会导致的初始化。
