链滴

虚拟机之类的加载机制

作者：zwxbest
原文链接：https://ld246.com/article/1544532276040
来源网站：链滴

<h2 id="类的生命周期">类的生命周期</h2>
加载，验证，准备，解析，初始化，使用，卸载
解析和初始化的相对顺序不固定，解析可以在初始化之后，叫做动态解析或者动态绑定，对应继承或接口的运行时。
<h2 id="虚拟机规范种的5种必须初始化的情况">虚拟机规范种的 5 种必须初始化的情况</h2>

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，则需要先触发其初始化。new 就是 new，getstatic、putstatic 是操作静态成员，invokestatic 是用静态方法
对类反射调用
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)，虚拟机会先初始这个主类。
使用 JDK1.7 的动态语言支持时，比如 Groovy，如果一个 java.lang.invoke.MethodHandle 实最后的解析结果 REF_getStatic、REF_putStatic、REF」nvokeStatic 的方法句柄，并且这个方法句所对应的类没有进行过初始化，则需要先触发其初始化。

<h2 id="题目">题目</h2>
<h3 id="通过子类引用父类的静态字段-不会导致子类初始化">通过子类引用父类的静态字段，不会致子类初始化</h3>
<pre><code class="highlight-chroma">public static class Parent
{
    static {
        System.out.pr ntln("parent init");
    }
    public static int v=100;
}

public static class hild extends Parent
{
    static {
        System.out.pr ntln("child init");
    }

}
public static void main(String[] args) {
    System.out.print n(Child.v);
}
</code></pre>
输出如下，在-XX:+TraceClassLoading 模式下，子类和父类都被加载了，但是只有父类被初始了。
<pre><code class="highlight-chroma">[Loaded class_loader_demo.ClassLoader1$Parent from file:/F:/Github/Demo/jvm/out/pro uction/jvm/]
[Loaded class_loa er_demo.ClassLoader1$Child from file:/F:/Github/Demo/jvm/out/production/jvm/]

parent init
100
</code></pre>
会加载，但不会初始化
<pre><code class="highlight-chroma">public static class Parent
{
    static {
        System.out.println("parent init");
    }
}
public static void main(String[] args) {
    Parent[] parents = new Parent[10];
}
</code></pre>
<pre><code class="highlight-chroma">public static class FinalClass
{
    public static final String constStr="FINAL";
    static {
        System.out.println("init");
    }
}

public static class UseFinalField
{
    public static void main(String[] args) {
        System.out.println(FinalClass.constStr);
    }
}
</code></pre>
结果如下,常量被放到常量池里了
<pre><code class="highlight-chroma">FINAL
</code></pre>
<h3 id="继承下初始化的顺序">继承下初始化的顺序</h3>
在定义成员的时候赋值语句是在 cinit 的最前面执行

parent 的定义赋值
parent cinit
parent 构造函数
child 的定义赋值
child cinit
child 构造函数

<h3 id="接口和类初始化的区别">接口和类初始化的区别</h3>

当一个类在初始化时，要求具父类全部都已经初始化过了，但是一个接口在初始化时，并不要求父接口全部都完成了初始化，只有在真正使用到父接门的时候（如引用接口中定义的常量）才会初始。
<h3 id="类加载器分类">类加载器分类</h3>

bootstrap classloader C 编写，加载 java 核心类，包括 ext 和 appclassloader,java 中没有对的，打印 String.getClassLoader 会是 null
extclassloader java 编写，加载/lib/ext 中的类
appclassloader 加载 classpath 变量中的类，通常自定义的类就是由它加载
自定义类加载器

<h3 id="双亲委派模型">双亲委派模型</h3>
看这个词我以为自定义类加载器要有爹又有妈呢，其实就是一个职责链模式。
<h3 id="loadclass--和forname--的区别">loadclass（）和 forname（）的区别</h3>
loaderClass 默认构造函数调用的是 resolve-false，不会类进行解析，因为不会初始化
forName 第二个参数 initialize 为 true，会初始化。
<h3 id="双亲委派工作机制">双亲委派工作机制</h3>
自底向顶检查，自顶向底加载。
根据双亲委派模式，在加载类文件时，子加载器首先会将加载请求委托给它的父加载器。
父加载器会检测自己是否已经加载过该类，如果己加载则加载过程结束；如果未加载则请求继
续向上传递，直到 BootstrapClassLoader。如果在请求向上委托的过程中，始终未检测到该类己

加载，则从 Boots 位叩 ClassLoader 开始尝试从其对应路径中加载该类文件，如果加载失败则由

子加载器继续尝试加载，直至发起加载请求的子加载器位为止。
双亲委派模式可以保证两点：一是子加载器可以使用父加载器己加载的类，而父加载器无
法使用子加载器己加载的类：二是父加载器己加载过的类无法被子加载器再次加载。这样就可
以保证 NM 的安全性和稳定性。

原文链接：虚拟机之类的加载机制