<https://mp.weixin.qq.com/s/7Dod-NcukPjYuQizGEdZdg、>

# Java 能否自定义一个类叫 java.lang.System？

# 缘起：一个面试题

最近在上下班地铁刷博客,无意刷到一个面试题,号称很多程序员的烈士公墓：

java 能否自己写一个类叫 java.lang.System

博主也提供了相关的答案：

一般情况下是不可以的,但是可以通过特殊的处理来达到目的,这个特殊的处理就是自己写个类加载器来加载自己写的这个 java.lang.System 类。

然后随手又刷了几个,基本雷同,看到的博客都是在讲 java 类加载的双亲委托机制, 一个类在需要加载时,会向上委托,直到最上层的 bootstrapClassLoader ,然后最上层的 bootstrapClassLoader 如果能在自己对应的目录加载就加载,不能就向下查找。

而 bootstrapClassLoader 会加载系统默认的 System 类,所以我们自定义的就不会被加载。

但是我们自定义一个类加载器加载特定路径的,避开 jvm 默认的三个类加载器的加载路径,就可以使我们的自定义 System 类被加载。

**可是真的是这样吗？**

为了弄清楚这个问题,我又看了下类加载。

# 什么是类加载

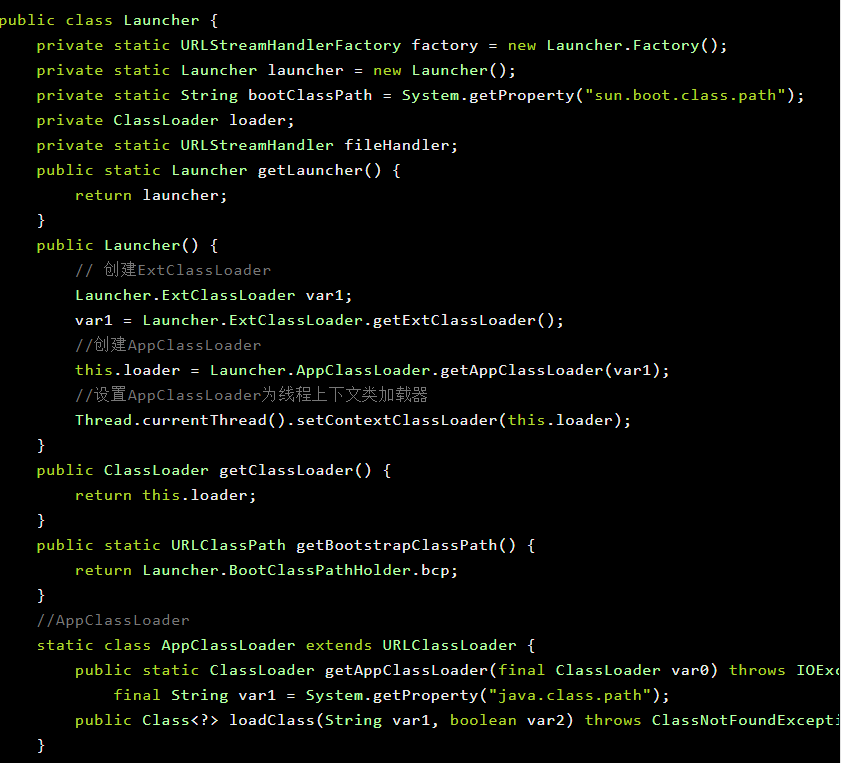
* 类加载指的是将类 Class 文件读入内存，并为之创建一个 java.lang.Class 对象, class 文件被载入到了内存之后，才能被其它 class 所引用
* jvm 启动的时候，并不会一次性加载所有的 class 文件，而是根据需要去动态加载
* java 类加载器是 jre 的一部分,负责动态加载 java 类到 java 虚拟机的内存

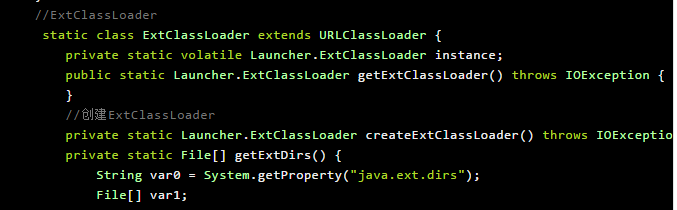
**类的唯一性由类加载器和类共同决定**

还了解到系统的三种类加载器：

* **AppClassLoader** : 也称为 SystemAppClass 加载当前应用的 classpath 的所有类。
* **ExtClassLoader** : 扩展的类加载器，加载目录 %JRE\_HOME%\lib\ext 目录下的 jar 包和 class 文件。还可以加载 -D java.ext.dirs 选项指定的目录。
* **BoostrapClassLoader** : 最顶层的加载类，主要加载核心类库， %JRE\_HOME%\lib 下的 rt.jar、resources.jar、charsets.jar 和 class 等。另外需要注意的是可以通过启动 jvm 时指定 -Xbootclasspath 和路径来改变 Bootstrap ClassLoader 的加载目录。比如 java -Xbootclasspath/a:path 被指定的文件追加到默认的 bootstrap 路径中。

瞄一眼源码,在Launcher类中





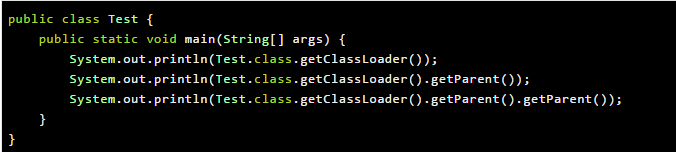
**这段源码有以下几点**

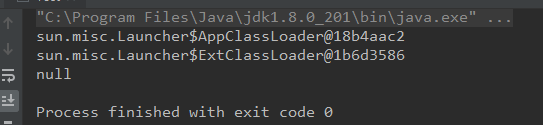
* Launcher 类在构造函数初始化了 ExtClassLoader 和 AppClassLoader 并设置 AppClassLoader 为线程上下文类加载器。
* 代码里面没有告诉我们 BoostrapClassLoader 从哪里来的,但却为其指定了要加载 class 文件的路径 sun.boot.class.path 。
* BoostrapClassLoader 是由 c++ 编写的,内嵌在 jvm 中,所以不能显示的看到他的存在【这个不是从源码中得到】。

# 实践出真知

我们通过代码来检验下上面的理论。

## 类加载器的父子关系

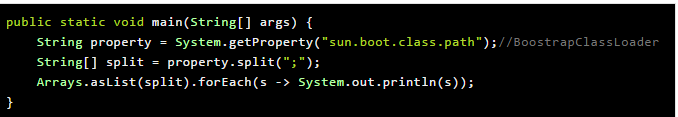


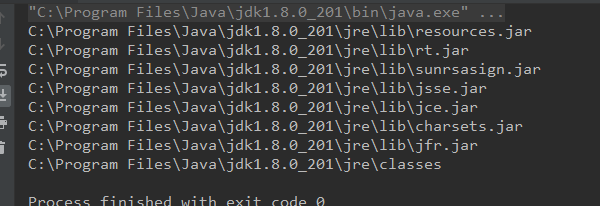


这段代码我们可以看到类加载器的父子关系, APPClassLoader->ExtClassLoader->BoostrapClassLoader , 但是 BoostrapClassLoader 无法显示的获取到,只能看到是个 null 。

## 源码中的路径到底加载哪些目录

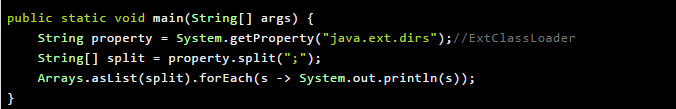
sun.boot.class.path

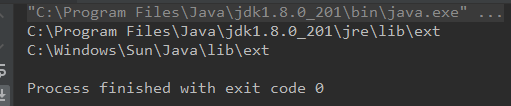




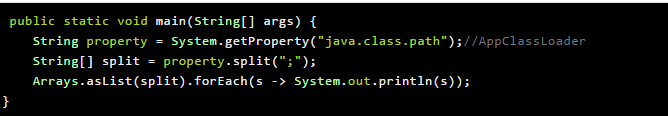
**可以看到是** jre/lib **目录下一些核心 jar**

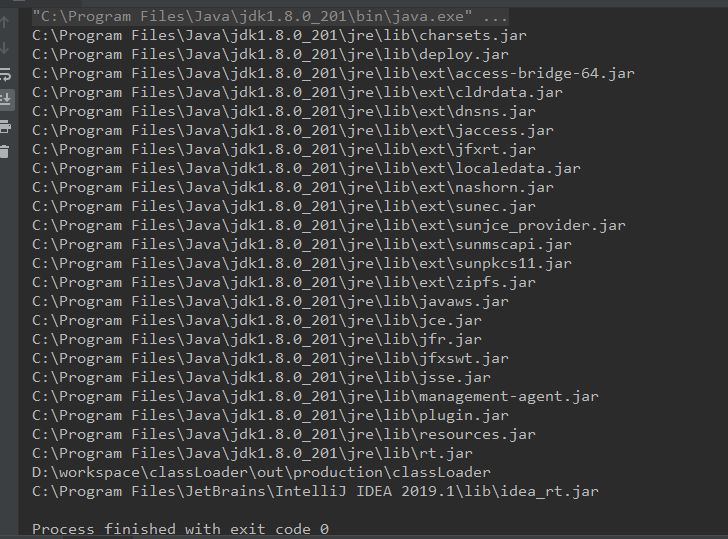
* java.ext.dirs





java.class.path

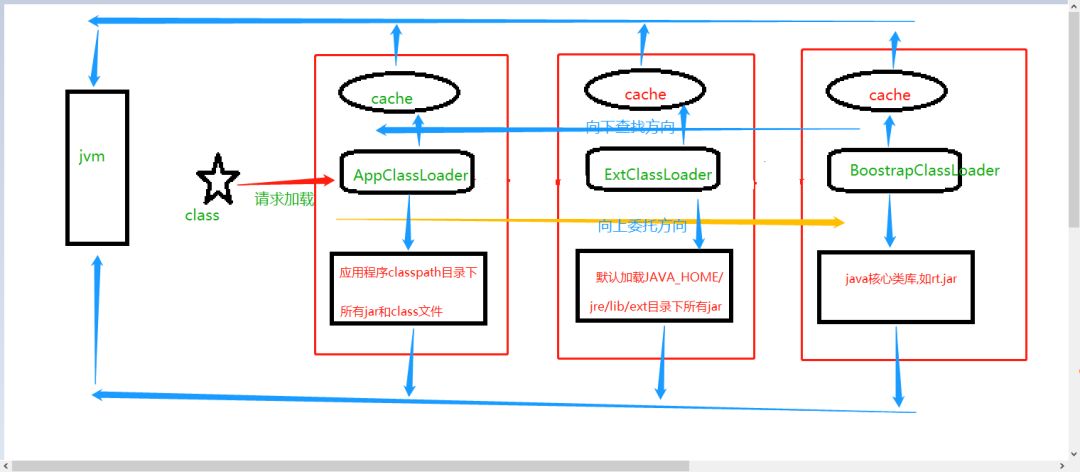




**可以看到,各个加载器加载的对应路径和前面的介绍是吻合的**

## 类加载的双亲委托机制

这里直接来一张图（processon 图库满了，这个先将就下）：



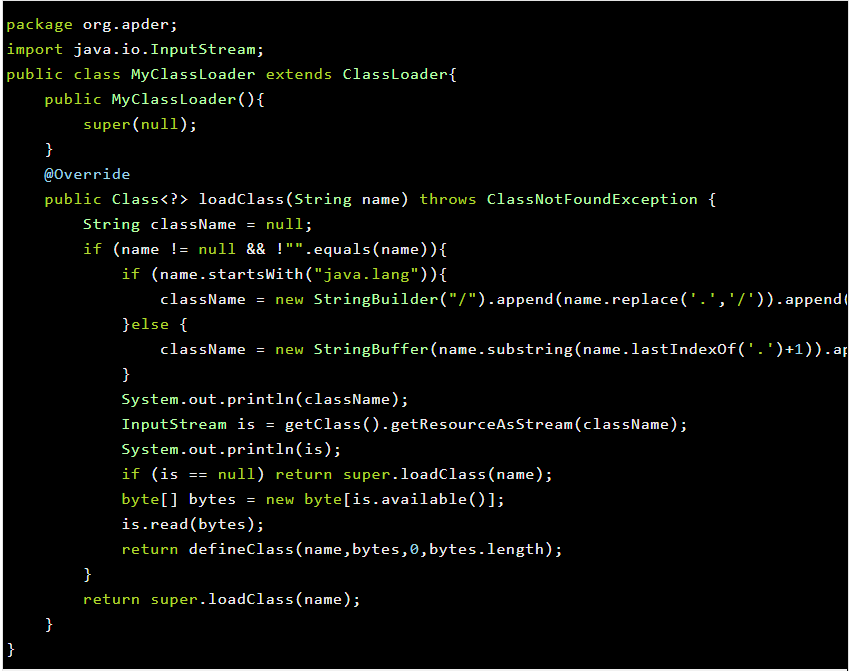
**如果看不太懂可以看下以下解释**

* 一个 class 文件发送请求加载,会先找到自定义的类加载器,当然这里没画出来。
* APPClassLoader 得到加载器请求后,向上委托交给 ExtClassLoader , ExtClassLoader 同理会交给 BoostrapClassLoader ,这是**向上委托方向**。
* 最终到达 BoostrapClassLoader ,**会先在缓存中找,没有就尝试在自己能加载的路径去加载**,找不到就交给 ExtClassLoader ,同理一直到用户自定义的 ClassLoader ,这就是**向下查找方向**。
* **前面说的类的唯一性由类和类加载器共同决定**, 这样保证了确保了类的唯一性。

**弄清楚这些,我们可以开始验证自定义的类加载器是否可以加载我们自定义的这个System类了**

# 自定义类加载器

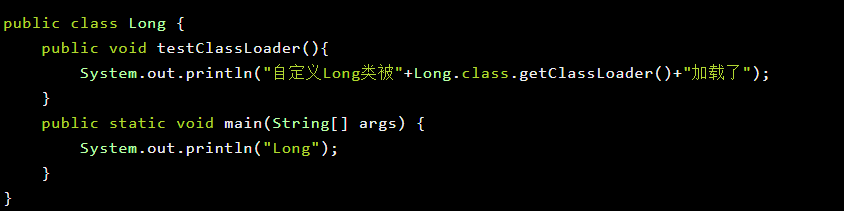
* 新建一个 MyClassLoader 继承 ClassLoader ,并重写 loadclass 方法



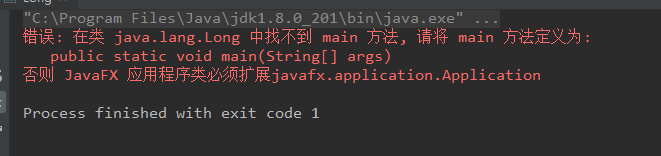
这里的代码很容易看懂,就不赘述了。

* 测试

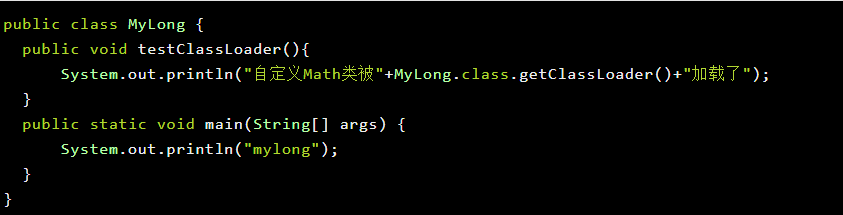
由于 System 需要用于打印获取结果,这里就用同属 lang 包的 Long 类：



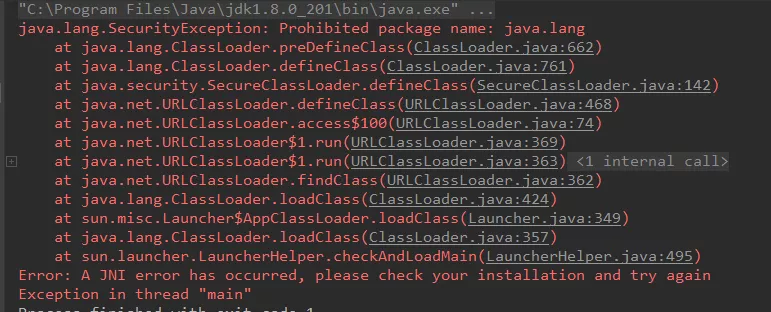
运行自定义 Long 类中 main 方法 报错如下：



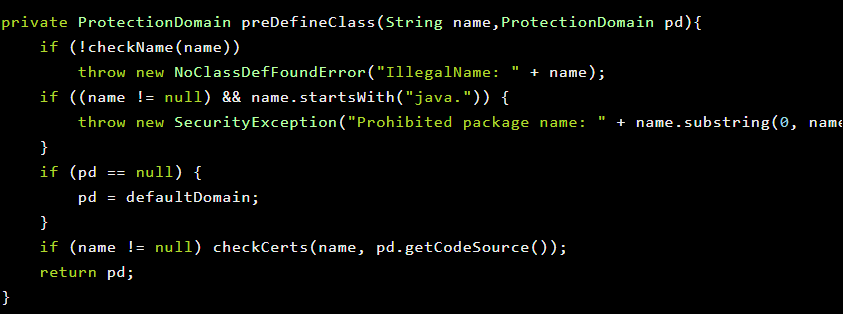
出错原因很简单,这个自定义的 Long 类申请加载后,会被委托到 BoostrapClassLoader,BoostrapClassLoader 会在向下查找的过程中找到 rt.jar 中的 java.lang.Long 类并加载,执行 main 方法时,找不到 main 方法,所以报找不到 main 方法。



我们再定义一个自定义的 java.lang.MyLong 类,执行 main 方法,报错如下



很明显的堆栈信息，禁止使用的包名 java.lang ,我们点进去 preDefineClass 看看：



可以看到,当如果类的全路径名以 java. 开头时,就会报错,看到这里,开头的答案你是否有了结果呢？

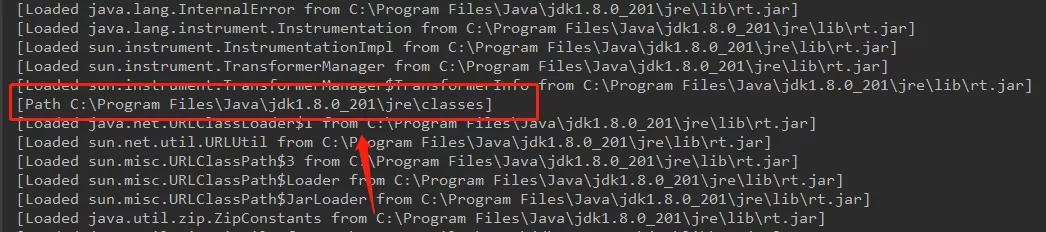
**我们梳理一下过程,如果用自定义的类加载器加载我们自定义的类**

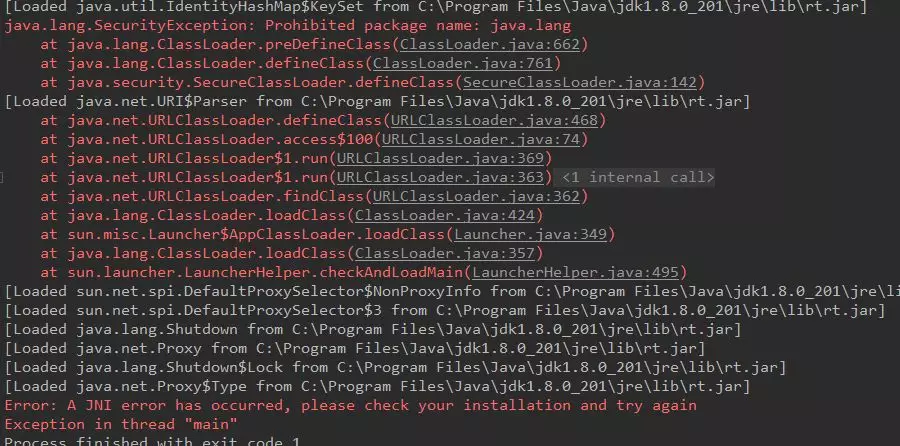
* 会调用自定义类加载器的 loadClass 方法。
* 而我们自定义的 classLoader 必须继承 ClassLoader,loadClass 方法会调用父类的 defineClass 方法。
* **而父类的这个 defineClass 是一个 final 方法,无法被重写**
* 所以自定义的 classLoader 是无论如何也不可能加载到以 java. 开头的类的。

到这里,最开始的问题已经有了答案。我们无法自定义一个叫 java.lang.System 的类。

## 思考

如果我把 MyLong 打成 jar 放到 BoostrapClassLoader 的加载路径呢？让 BoostrapclassLoader 去加载,具体操作如下,在 jdk 的 jre 目录下创建 classes 目录,然后把 MyLong.jar 复制进去,再通过 vmOptions 追加这个 classes 目录以使 BoostrapClassLoader 加载：





可以看到仍然加载不了,如果能加载，在控制台是会有 load 信息的,如果不是 java.lang.Long ,是可以跨过 APPClassLoader 和 ExtClassLoader 来让 boostraPClassloader 来加载的,这里就不演示了,操作很简单。

下面是vm参数



由一个面试题引起的类加载器思考,既然已经写到这里,干脆把线程上下文类加载器也一并学习了。

# 拓展线程上下文类加载器

为什么不和前面三种类加载器放在一起说呢,这个线程上下文类加载器只是一个概念,是一个成员变量,而前三种是确切存在的,是一个类,我们来看一下 Thread 的源码：

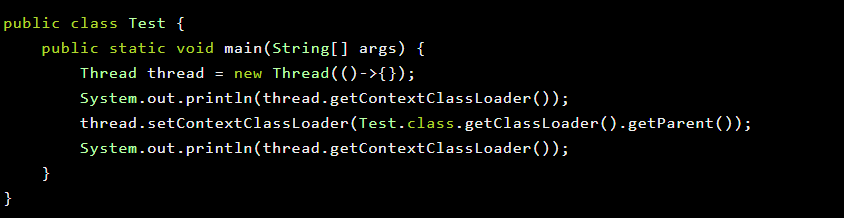


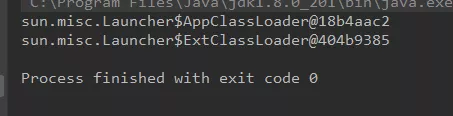
## 特点

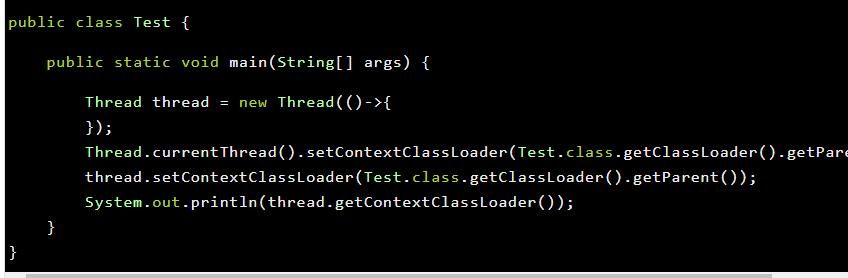
* 线程上下文类加载器是一个成员变量,可以通过相应的方法来设置和获取。
* 每个线程都有一个线程类加载器,默认是 AppClassLoader 。
* 子线程默认使用父线程的 ClassLoader ,除非子线程通过上面的 setContextClassLoader 来设置。

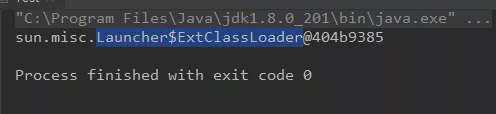
## 测试

针对以上两点简单测试一下：









可以证明以上三点

# 总结

* java 三种类加载器
* 一条主线-----**路径**
* 一个机制->双亲委托
* 两个方向->向上委托,向下查找