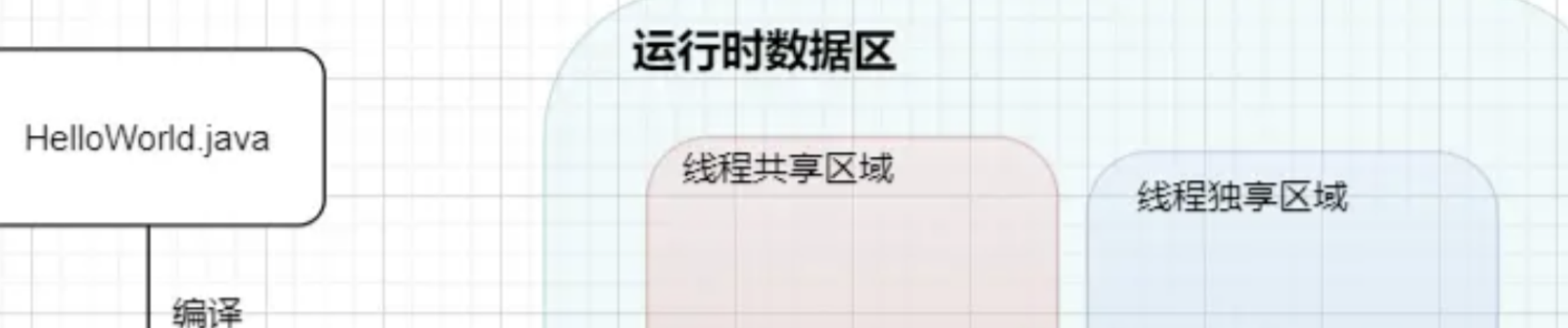
> 主要讲述Java类的加载过程。

前言

其实类的加载过程应该放到JVM的第一个系列，因为我接触JVM是从JVM内存结构开始，所以就先通过2篇文章完成这块内容的讲解。对于Java类的加载过程，我感觉里面的内容其实不多，也不那么重要，就把网上的资料简单整理了一下，作为了解即可。

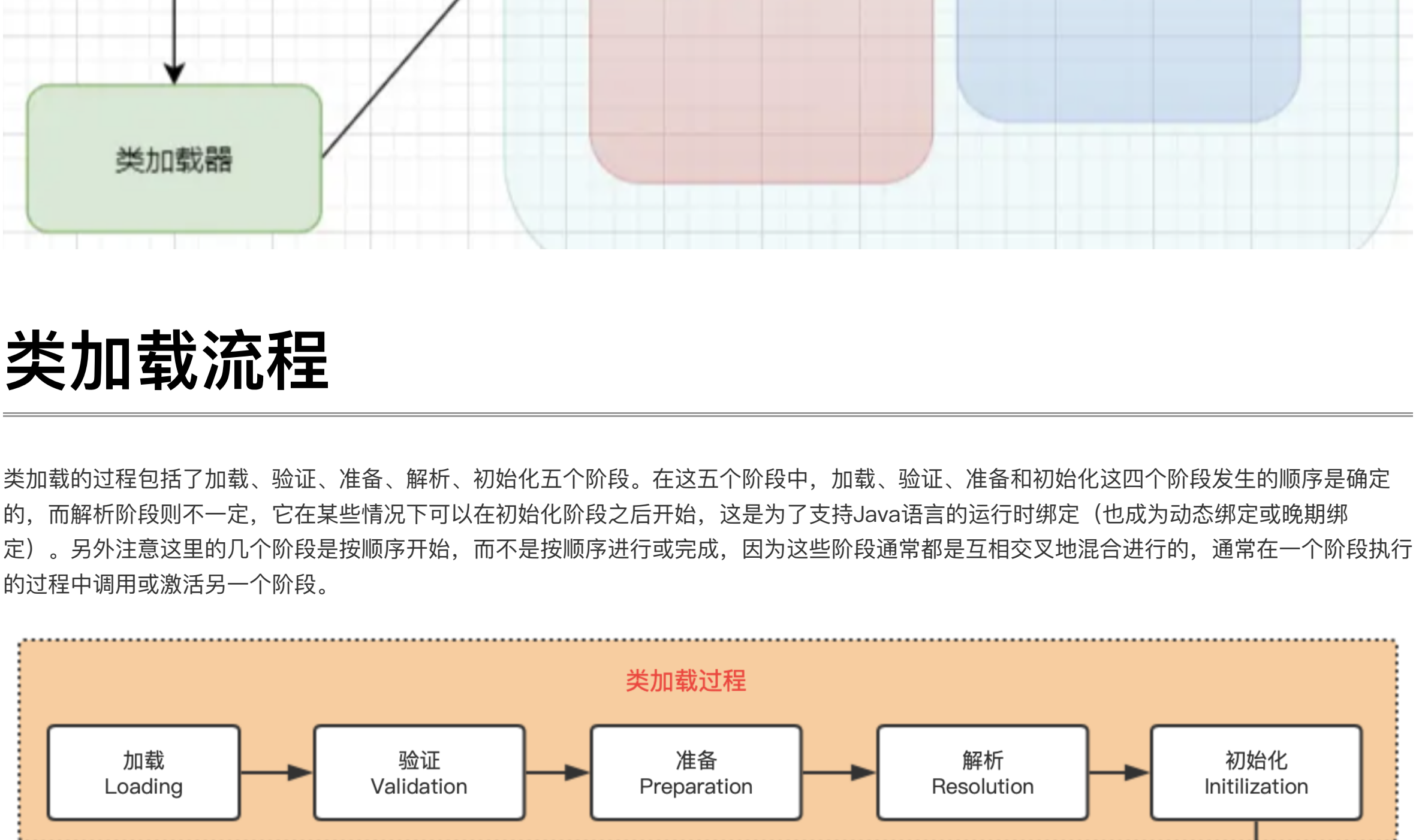
简介

如果 JVM 想要执行这个 .class 文件，我们需要将其装进一个类加载器 中，它就像一个搬运工一样，会把所有的 .class 文件全部搬进JVM里面来。



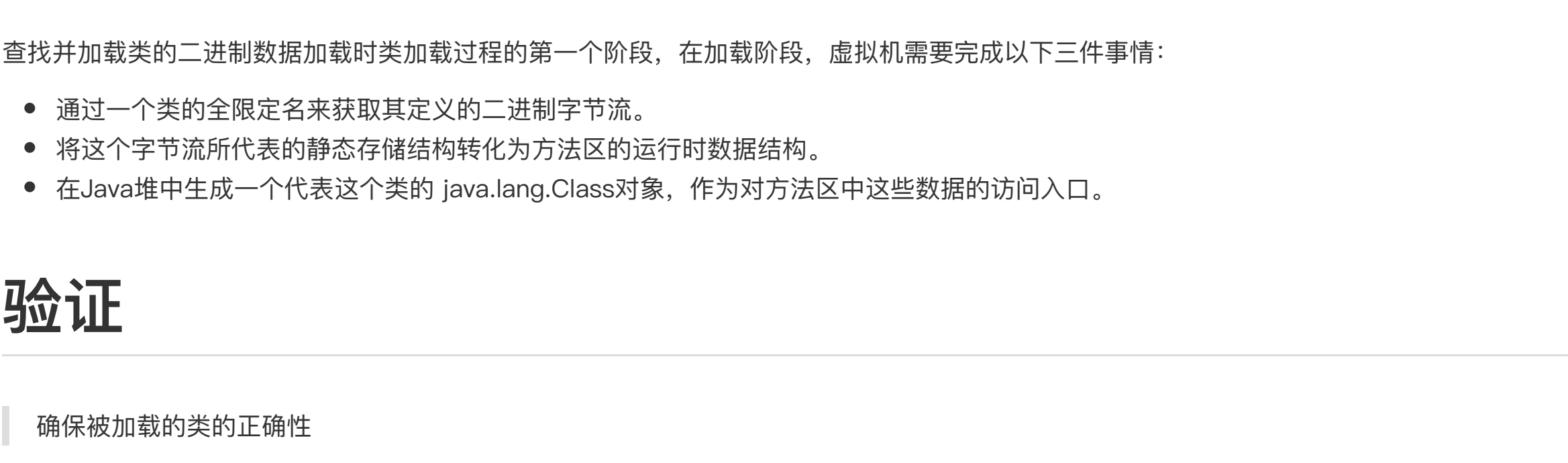
重点知识：

- Java文件经过编译后变成 .class 字节码文件
- 字节码文件通过类加载器被搬运到 JVM 虚拟机中
- 虚拟机主要的5大块：方法区，堆都为线程共享区域，有线程安全问题，栈和本地方法栈和计数器都是独享区域，不存在线程安全问题，而 JVM 的调优主要就是围绕堆，栈两大块进行。



类加载流程

类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。



加载

查找并加载类的二进制数据加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

- 通过一个类的全限定名来获取其定义的二进制字节流。
- 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数据结构。
- 在Java堆中生成一个代表这个类的 java.lang.Class对象，作为对方法区中这些数据的访问入口。

验证

> 确保被加载的类的正确性

验证是连接阶段的第一步，这一阶段的目的是为了确保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。验证阶段大致会完成4个阶段的检验动作：

- 文件格式验证：验证字节流是否符合Class文件格式的规范；例如：是否以 0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
- 元数据验证：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了 java.lang.Object之外。
- 字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
- 符号引用验证：确保解析动作能正确执行。

准备

> 为类的静态变量分配内存，并将其初始化为默认值

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。
- 这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。
- 如果类字段的字段属性表中存在 ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。

> 假设一个类变量的定义为：publicstaticintvalue=3，那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，value赋值为3的动作将在初始化阶段才会执行。

解析

> 把类中的符号引用转换为直接引用

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，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、方法句柄和调用点限定符7类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。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柄。

初始化

初始化其实就是一个赋值的操作，它会执行一个类构造器的方法。由编译器自动收集类中所有变量的赋值动作，此时准备阶段时的那个static int a = 3 的例子，在这个时候就正式赋值为3

卸载

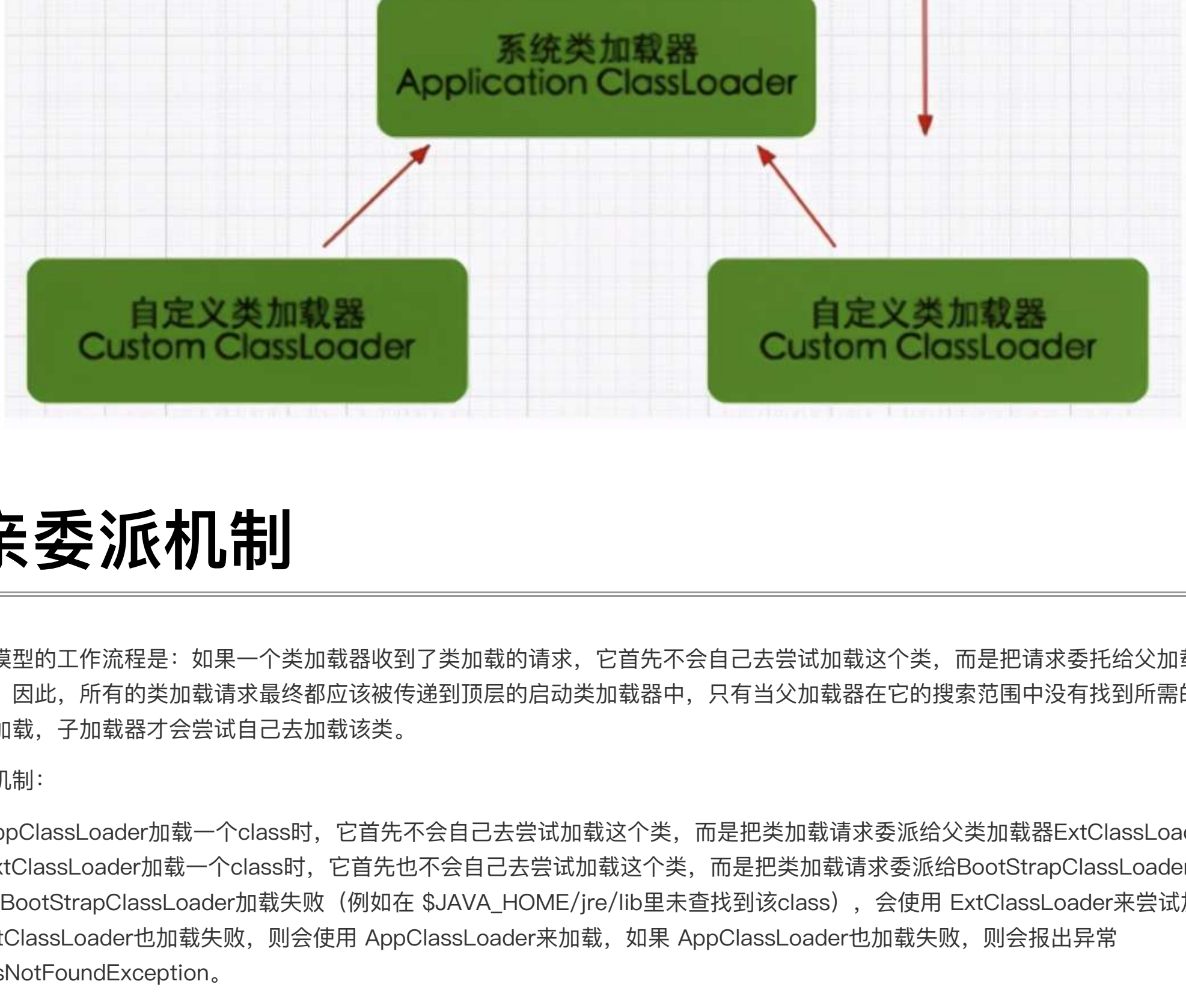
GC将无用对象从内存中卸载，Java虚拟机将结束生命周期：

- 执行了 System.exit()方法
- 程序正常执行结束
- 程序在执行过程中遇到了异常或错误而异常终止
- 由于操作系统出现错误而导致Java虚拟机进程终止

类加载器的加载顺序

加载一个Class类的顺序也是有优先级的，类加载器从最底层开始往上的顺序是这样的：

- BootStrap ClassLoader：rt.jar
- Extention ClassLoader: 加载扩展的jar包
- App ClassLoader：指定的classpath下面的jar包
- Custom ClassLoader：自定义的类加载器



双亲委派机制

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

双亲委派机制：

- 当 AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。
- 当 ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。
- 如果 BootStrapClassLoader加载失败（例如在 $JAVA_HOME/jre/lib里未查找到该class），会使用 ExtClassLoader来尝试加载；
- 若ExtClassLoader也加载失败，则会使用 AppClassLoader来加载，如果 AppClassLoader也加载失败，则会报出异常 ClassNotFoundException。

后记

这篇文章基本都是“八股文”，干货不多，但是它又是JVM系列不可或缺的部分，反正就是给我一种“食之无味弃之可惜”的感觉，怎么说呢，还是没有JVM内存知识看的带劲、来的实际，所以仅供了解即可。

学习交流

可以扫下面二维码，关注「楼仔」公众号。

长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏🏻👏🏻👏🏻

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。