# JVM

## 我们写的Java代码到底是如何运行起来的？

针对这个问题，我们来一步一步的分析。

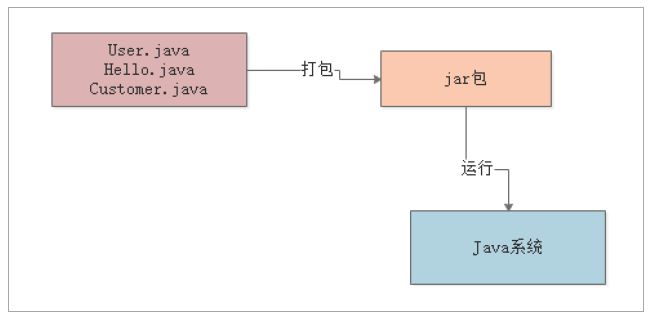
首先假设咱们写好了一份Java代码，那这份Java代码中，是不是会包含很多的“.java”为后缀的代码文件？比如User.java，OrderService.java，CustomerManager.java其实咱们Java程序员平时在Eclipse、Intellij IDEA等开发工具中，就有很多类似这样的Java源代码文件。

那么大家现在思考一下，当我们写好这些“.java”后缀的代码文件之后，接下来你要部署到线上的机器上去运行，你会怎么做？

一般来说，都是把代码给打成“.jar”后缀的jar包，或者是“.war”后缀的war包，是不是？

然后呢，就是把你打包好的jar包或者是war包给放到线上机器去部署。

这个部署就有很多种途径了，但是最基本的一种方式，就是通过Tomcat这类容器来部署代码，也可以是你自己手动通过 “java”命令来运行一个jar包中的代码。咱们先用下面这张图，回忆一下这个顺序。



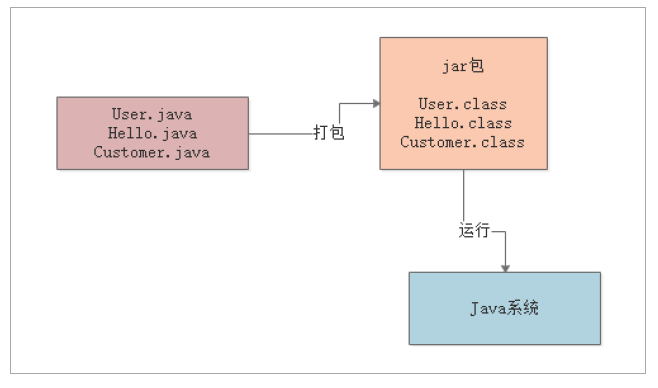
但是实际上这里有一个非常关键的步骤，那就是“编译”

也就是说，在我们写好的“.java”代码打包的过程中，一般就会把代码编译成“.class”后缀的字节码文件，比如 “User.class”，“Hello.class”，”Customer.class“。

然后这个“.class”后缀的字节码文件，他才是可以被运行起来的！

所以首先，无论大家对JVM机制是否熟悉，咱们都先来回顾一下这个编译的过程，以及“.class”字节码文件的概念。

来看看下图，一起来感受一下：



接着我们可能就要思考下一个问题：

对于编译好的这些“.class”字节码，是怎么让他们运行起来的呢？

这个时候就需要使用诸如“java -jar”之类的命令来运行我们写好的代码了。

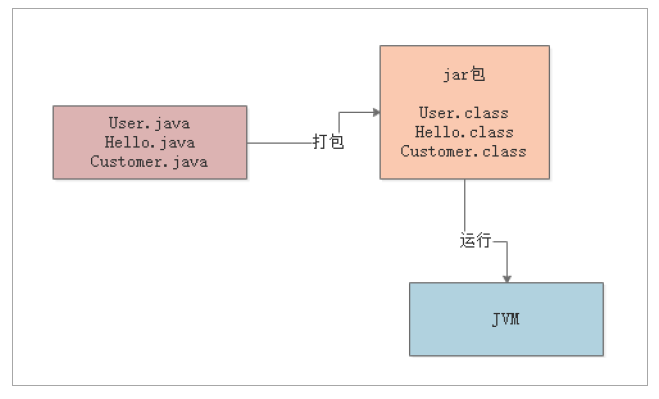
此时一旦你采用“java”命令，实际上此时就会启动一个JVM进程。

这个JVM就会来负责运行这些“.class”字节码文件，也就相当于是负责运行我们写好的系统。

所以平时我们写好的某个系统在一台机器上部署的时候，你一旦启动这个系统，其实就是启动了一个JVM，由它来负责运行这台机器上运行的这个系统。

对这个概念，大家一定要先搞清楚。

我们还是用一张图来展示一下，相信大家图文结合，会理解的更好。



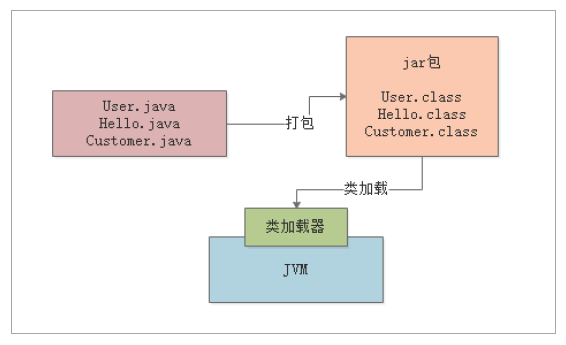
接着下一步，JVM要运行这些“.class”字节码文件中的代码，那是不是首先得把这些“.class”文件中包含的各种类给加载进来？

这些“.class”文件不就是我们写好的一个一个的类吗？对不对？

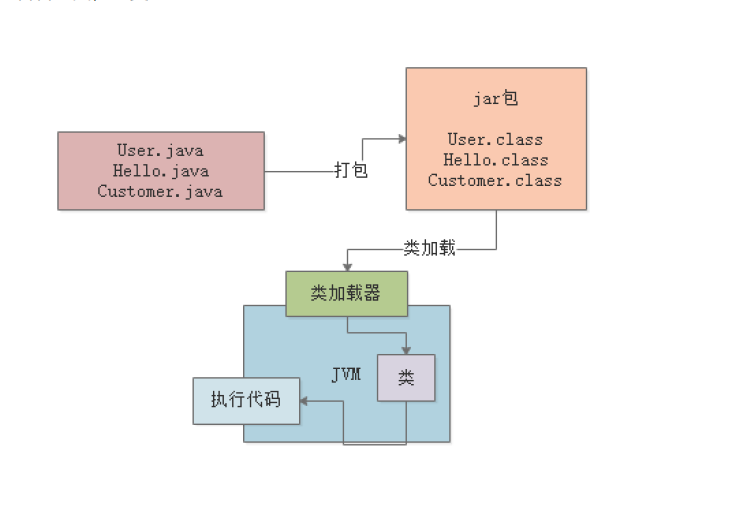
此时就会有一个“类加载器”的概念。

此时会采用类加载器把编译好的那些“.class”字节码文件给加载到JVM中，然后供后续代码运行来使用。

我们再看下图。



接着，最后一步，JVM就会基于自己的字节码执行引擎，来执行加载到内存里的我们写好的那些类了比如你的代码中有一个“main()”方法，那么JVM就会从这个“main()”方法开始执行里面的代码。他需要哪个类的时候，就会使用类加载器来加载对应的类，反正对应的类就在“.class”文件中。大家最后看看下面的图。



从写“.java”后缀的源代码开始，一步一步梳理了以下的流程：

写好的代码编译成“.class”后缀的字节码文件JVM是个什么东西

JVM跟我们平时运行在机器上的系统之间是什么关系类加载器的概念

针对加载进内存的类进行代码的执行

一个类从加载到使用，一般会经历下面的这个过程：

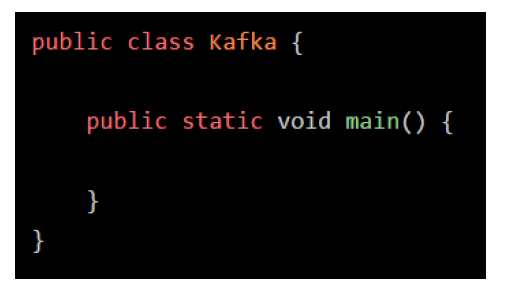
加载 -> 验证 -> 准备 -> 解析 -> 初始化 -> 使用 -> 卸载

JVM在执行我们写好的代码的过程中，一般在什么情况下会去加载一个类呢？

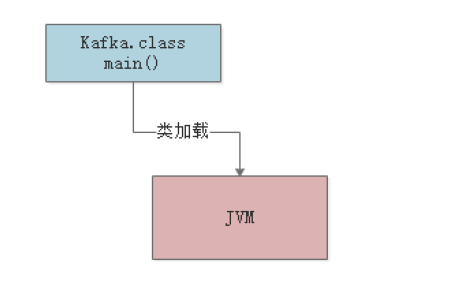
也就是说，啥时候会从“.class”字节码文件中加载这个类到JVM内存里来。

其实答案非常简单，就是在你的代码中用到这个类的时候。举个简单的例子，比如下面你有一个类（Kafka.class），里面有一个“main()”方法作为主入口。

那么一旦你的JVM进程启动之后，它一定会先把你的这个类（Kafka.cass）加载到内存里，然后从“main()”方法的入口代码开始执行。



坚持一步一图，大家先看看下图，感受一下：

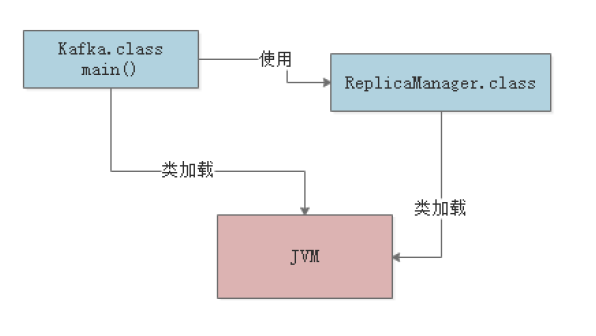


接着假设上面的代码中，出现了如下的这么一行代码：



这时可能大家就想了，你的代码中明显需要使用“ReplicaManager”这个类去实例化一个对象，此时必须得把“ReplicaManager.class”字节码文件中的这个类加载到内存里来，所以这个时候就会触发JVM通过类加载器，从“ReplicaManager.class”字节码文件中加载对应的类到内存里来使用，这样代码才能跑起来。

我们来看下面的图：



简单概括一下：

首先你的代码中包含“main()”方法的主类一定会在JVM进程启动之后被加载到内存，开始执行你的“main()”方法中的代码

接着遇到你使用了别的类，比如“ReplicaManager”，此时就会从对应的“.class”字节码文件加载对应的类到内存里来。

验证、准备、初始化

（1）验证阶段

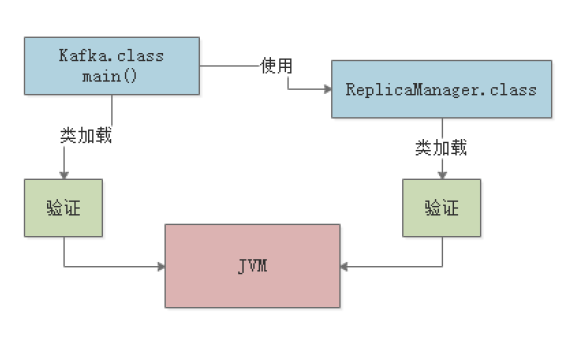
简单来说，这一步就是根据Java虚拟机规范，来校验你加载进来的“.class”文件中的内容，是否符合指定的规范。

这个相信很好理解，假如说，你的“.class”文件被人篡改了，里面的字节码压根儿不符合规范，那么JVM是没法去执行这个字节码

的！

所以把“.class”加载到内存里之后，必须先验证一下，校验他必须完全符合JVM规范，后续才能交给JVM来运行。

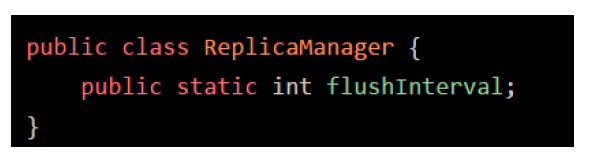
下面用一张图，展示了这个过程：



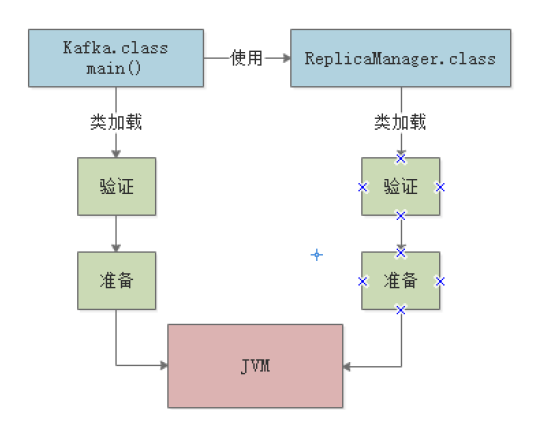
（2）准备阶段

这个阶段其实也很好理解，咱们都知道，我们写好的那些类，其实都有一些类变量

比如下面的这个“ReplicaManager”类：



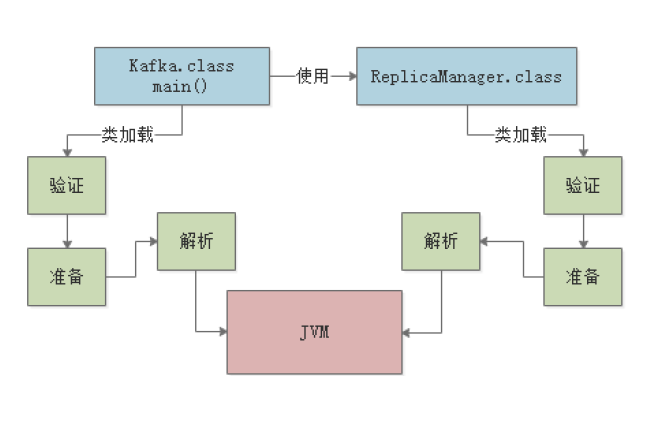
假设你有这么一个“ReplicaManager”类，他的“ReplicaManager.class”文件内容刚刚被加载到内存之后，会进行验证，确认这个字节码文件的内容是规范的接着就会进行准备工作。这个准备工作，其实就是给这个“ReplicaManager”类分配一定的内存空间，然后给他里面的类变量（也就是static修饰的变量）分配内存空间，来一个默认的初始值，比如上面的示例里，就会给“flushInterval”这个类变量分配内容空间，给一个“0”这个初始值。整个过程，如下图所示：



（3）解析阶段

这个阶段干的事儿，实际上是把符号引用替换为直接引用的过程，其实这个部分的内容很复杂，涉及到JVM的底层

画图展示一下：



（4）三个阶段的小结

其实这三个阶段里，最核心的大家务必关注的，就是“准备阶段”

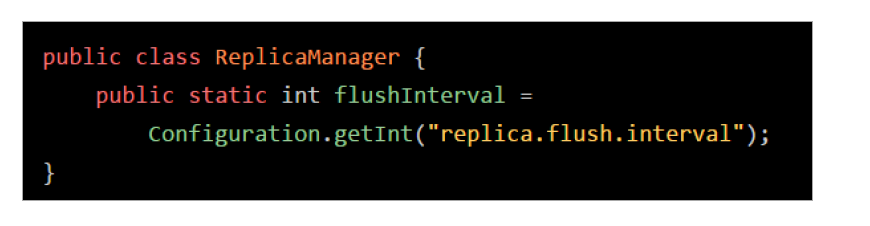
因为这个阶段是给加载进来的类分配好了内存空间，类变量也分配好了内存空间，并且给了默认的初始值。

核心阶段：初始化

在准备阶段时，就会把我们的“ReplicaManager”类给分配好内存空间

另外他的一个类变量“flushInterval”也会给一个默认的初始值“0”，那么接下来，在初始化阶段，就会正式执行我们的类初始化的代码了。

那么什么是类初始化的代码呢？我们来看看下面这段代码：



大家可以看到，对于“flushInterval”这个类变量，我们是打算通过Configuration.getInt("replica.flush.interval")这段代码来获取一个值，并且赋值给他的

但是在准备阶段会执行这个赋值逻辑吗？

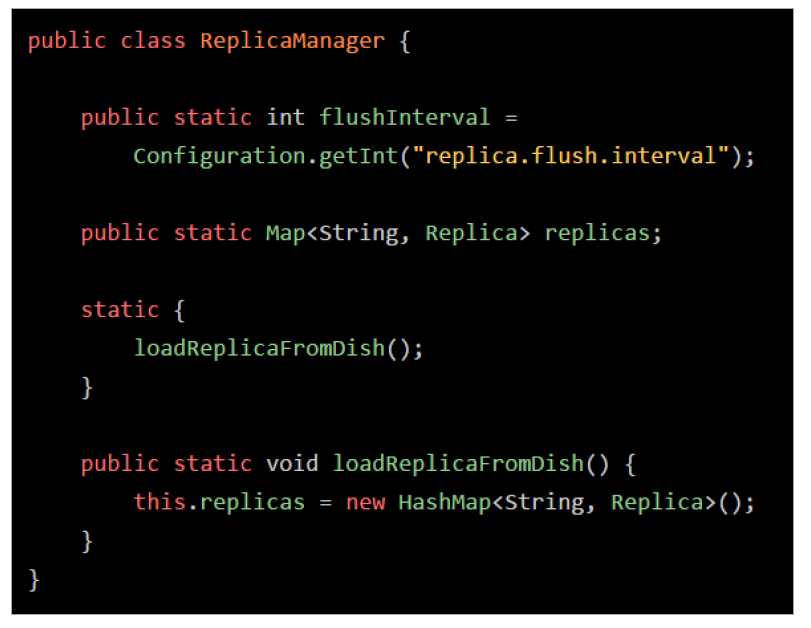
NO！在准备阶段，仅仅是给“flushInterval”类变量开辟一个内存空间，然后给个初始值“0”罢了。

那么这段赋值的代码什么时候执行呢？答案是在“初始化”阶段来执行。

在这个阶段，就会执行类的初始化代码，比如上面的 Configuration.getInt("replica.flush.interval") 代码就会在这里执行，完成一个配置项的读取，然后赋值给这个类变量“flushInterval”。

另外比如下图的static静态代码块，也会在这个阶段来执行。

类似下面的代码语义，可以理解为类初始化的时候，调用“loadReplicaFromDish()”方法从磁盘中加载数据副本，并且放在静态变量“replicas”中：



那么搞明白了类的初始化是什么，就得来看看类的初始化的规则了。

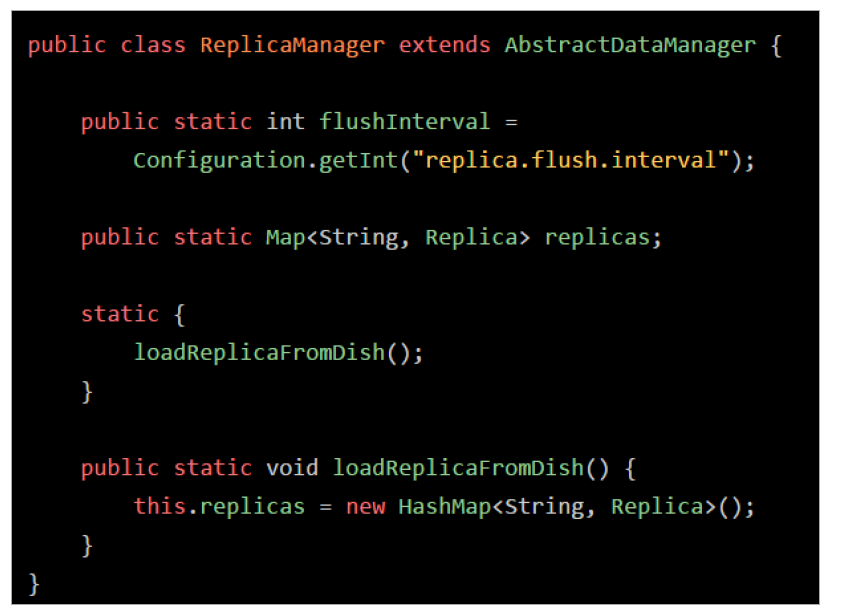
什么时候会初始化一个类？

一般来说有以下一些时机：比如“new ReplicaManager()”来实例化类的对象了，此时就会触发类的加载到初始化的全过程，把这个类准备好，然后再实例化一个对象出来；

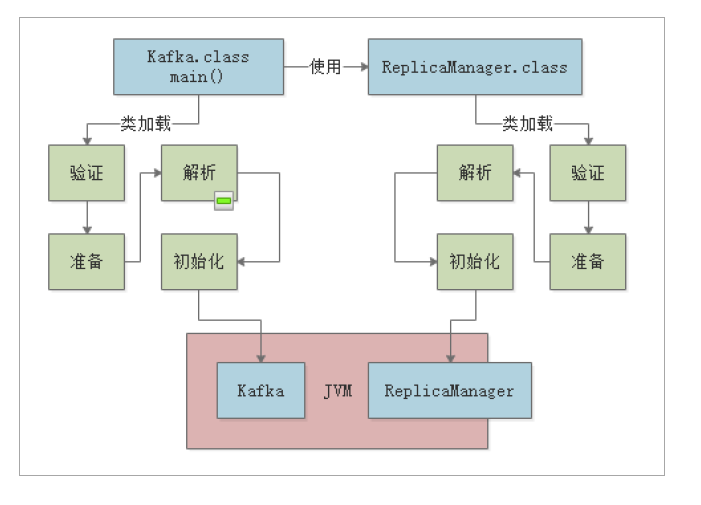
或者是包含“main()”方法的主类，必须是立马初始化的。

此外，这里还有一个非常重要的规则，就是如果初始化一个类的时候，发现他的父类还没初始化，那么必须先初始化他的父类

比如下面的代码：



如果你要“new ReplicaManager()”初始化这个类的实例，那么会加载这个类，然后初始化这个类，但是初始化这个类之前，发现AbstractDataManager作为父类还没加载和初始化，那么必须先加载这个父类，并且初始化这个父类。这个规则，大家必须得牢记，再来一张图，借助图片来进行理解：



类加载器和双亲委派机制

现在相信大家都搞明白了整个类加载从触发时机到初始化的过程了，接着给大家说一下类加载器的概念

因为实现上述过程，那必须是依靠类加载器来实现的

那么Java里有哪些类加载器呢？简单来说有下面几种：

（1）启动类加载器

Bootstrap ClassLoader，他主要是负责加载我们在机器上安装的Java目录下的核心类的

相信大家都知道，如果你要在一个机器上运行自己写好的Java系统，无论是windows笔记本，还是linux服务器，是不是都得装一下JDK？

那么在你的Java安装目录下，就有一个“lib”目录，大家可以自己去找找看，这里就有Java最核心的一些类库，支撑你的Java系统的运行。

所以一旦你的JVM启动，那么首先就会依托启动类加载器，去加载你的Java安装目录下的“lib”目录中的核心类库。

（2）扩展类加载器

Extension ClassLoader，这个类加载器其实也是类似的，就是你的Java安装目录下，有一个“lib\ext”目录

这里面有一些类，就是需要使用这个类加载器来加载的，支撑你的系统的运行。

那么你的JVM一旦启动，是不是也得从Java安装目录下，加载这个“lib\ext”目录中的类？

（3）应用程序类加载器

Application ClassLoader，这类加载器就负责去加载“ClassPath”环境变量所指定的路径中的类

其实你大致就理解为去加载你写好的Java代码吧，这个类加载器就负责加载你写好的那些类到内存里。

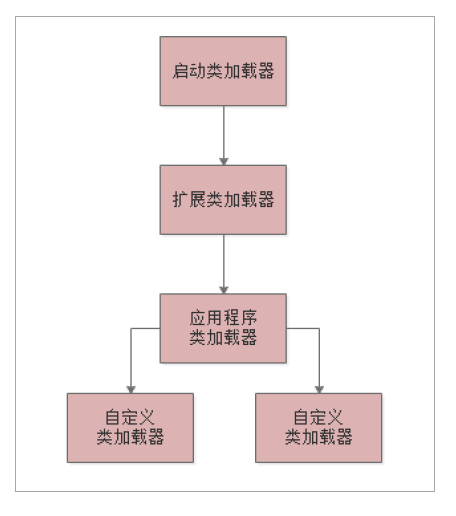
（4）自定义类加载器

除了上面那几种之外，还可以自定义类加载器，去根据你自己的需求加载你的类。

（5）双亲委派机制

JVM的类加载器是有亲子层级结构的，就是说启动类加载器是最上层的，扩展类加载器在第二层，第三层是应用程序类加载器，最后一层是自定义类加载器。

大家看下图：



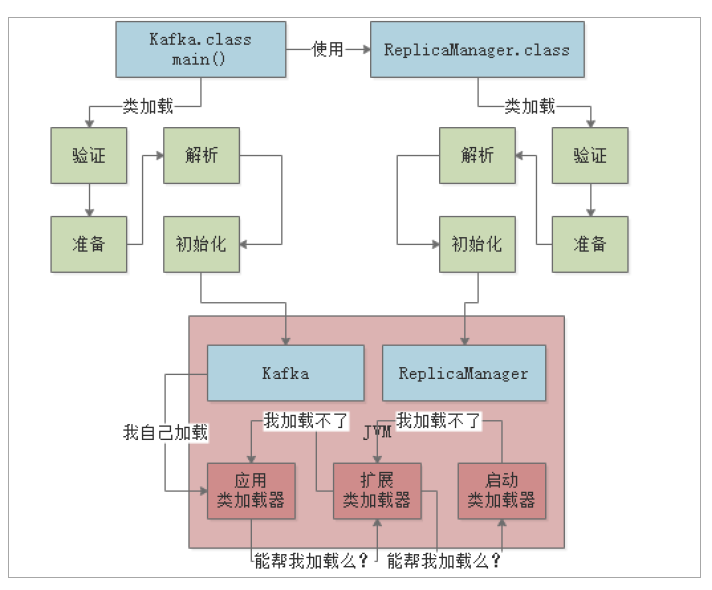
然后，基于这个亲子层级结构，就有一个双亲委派的机制

什么意思呢？

就是假设你的应用程序类加载器需要加载一个类，他首先会委派给自己的父类加载器去加载，最终传导到顶层的类加载器去加载

但是如果父类加载器在自己负责加载的范围内，没找到这个类，那么就会下推加载权利给自己的子类加载器。

来一张图图，感受一下类加载器的双亲委派模型。



JVM的内存区域划分：

1. 存放类的方法区：这个方法区是在JDK1.8以前的版本里，代表JVM中的一块区域。主要放从“.class”文件里加载进来的类，还会有一些类似常量池的东西放在这个区域里。但是在JDK1.8以后，这块区域的名字改了，叫做“Metaspace”,可以认为是“元数据空间”这样的意思。当然这里主要还是存放我们自己写的各种类相关的信息。

方法区触发垃圾回收的几个条件：

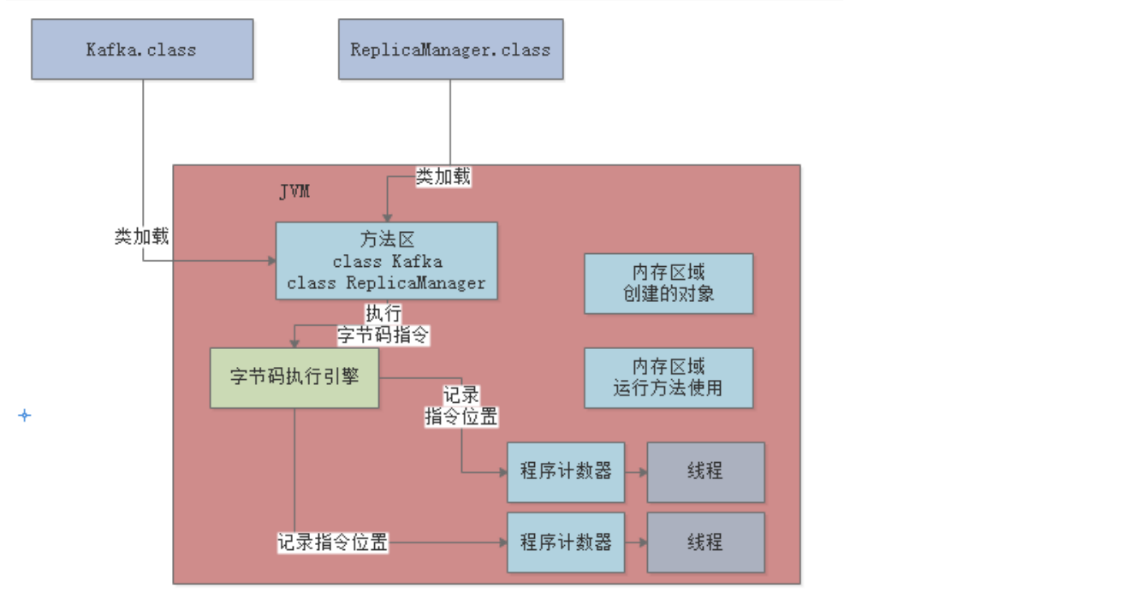
①首先该类的所有实例对象都已经从Java堆内存里被回收，

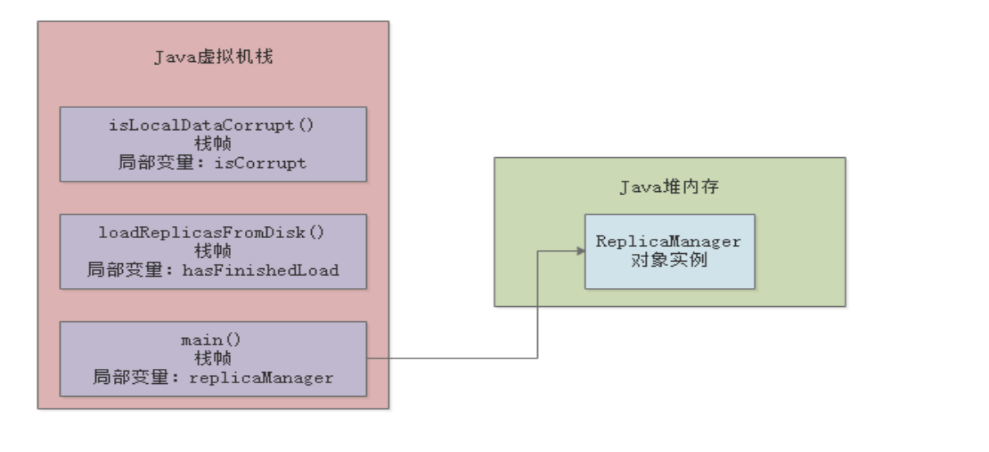
②其次加载这个类的ClassLoader已经被回收

③最后，对该类的Class对象没有任何引用，满足以上三个条件就可以回收该类了

1. 程序计数器：执行字节码指令的时候，JVM里就需要一个特殊的内存区域了，那就是“程序计数器”。这个程序计数器就是用来记录当前执行的字节码指令的位置的，也就是记录目前执行到了哪一条字节码指令。
2. Java虚拟机栈：JVM必须有一块区域是来保存每个方法内的局部变量等数据的，这个区域就是Java虚拟机栈每个线程都有自己的Java虚拟机栈。
3. Java堆内存：存放我们在代码中创建的各种对象的区域。

JVM是支持多线程的





咱们平时代码里创建出来的对象，一般就是两种：

一种是短期存活的，分配在Java堆内存之后，迅速使用完就会被垃圾回收

另外一种是长期存活的，需要一直生存在Java堆内存里，让程序后续不停的去使用

第一种短期存活的对象，是在Java堆内存的新生代里的。

第二种长期存活的对象，是在Java堆内存的老年代里的。

大部分的正常对象，都是优先在新生代分配内存的。