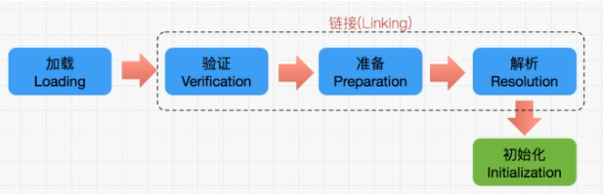
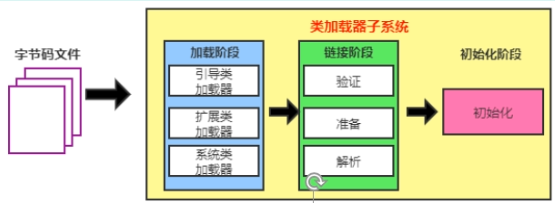
# 概述和概览

1. 类文件（class file）存在本地硬盘上，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
2. 类文件（class file） 加载到JVM中，被称为DNA元数据模板，放在**方法区**
3. 在 **.class文件 -> JVM -> 元数据模板** 这个过程中需要一个运输工具（类装载器 Class Loader），扮演一个快递员的角色





· 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标识（魔数 CAFEBABE）

· ClassLoader 只负责class文件的加载，至于它是否可以运行，则由Execution Engine（执行引擎）决定

· 加载的类信息存放在方法区，除了**类的信息**外，方法区中还会存放着运行时常量池信息，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**）。

# 2、类加载的三个阶段

## 1、加载

使用类加载器加载一个类

1、通过一个类的全限定名获取定义此类的二进制字节流

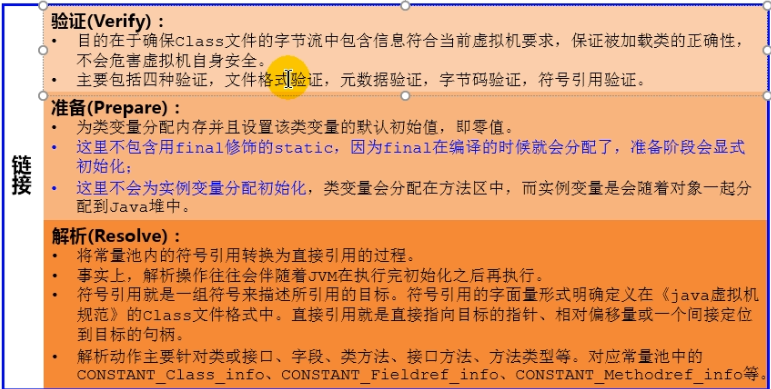
2、将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构

3、在内存（堆）中生成一个代表这个类的 java.lang.Class对象，**作为方法区中这个类的各种数据的访问入口**

加载 .class 文件的方式

1. 从本地系统磁盘中直接加载
2. 通过网络获取
3. 从zip 压缩包中读取，成为日后jar，war格式的基础
4. 运行时计算生成。使用最多的是动态代理技术
5. 由其他文件生成，典型场景jsp应用
6. 从专有数据库中提取.class 文件，比较少见
7. 从加密文件中获取，典型的放Class 文件被反编译的额保护措施

## 2、链接



### 验证

·目的是确保Class文件的字节流中包含信息符合当前虚拟机的要求，保证被加载类的正确性，不会危害虚拟机自身的安全。

·主要包含四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证

### 准备

1、为类变量分配**内存**并且设置该类的**默认初始值**，即零值，真正赋值是在初始化阶段。

2、这里不包含用final 修饰的static 变量即常量，因为**final 在编译**的时候就会分配内存了，在这里（准备阶段）就会显式的初始化。

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

### 3、解析

·**将.class文件的常量池中的符号引用转换为直接引用的过程。**

·事实上解析操作往往会伴随着着JVM在执行完初始化之后再执行

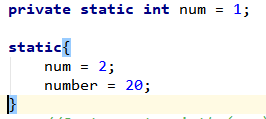
·**符号引用就是一组符号来描述所引用的目标**，符号引用的**字面量**形式明确定义在《java虚拟机规范》的Class 文件格式中，**直接引用就是直接指向目标的指针**，相对偏移量，或间接定位到目标的句柄

·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 CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info等

## 初始化

·**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**中的语句合并而来，如图



·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。（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

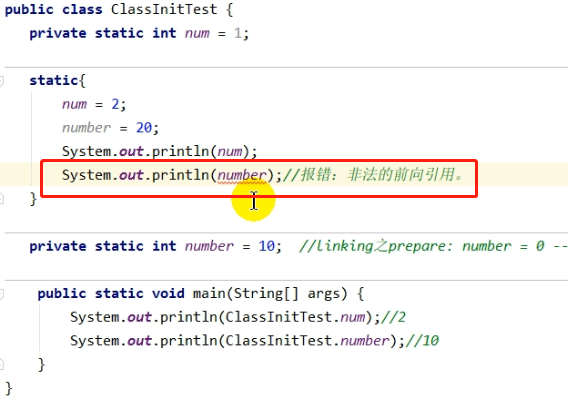
**赋值顺序问题**



number在链接阶段准备阶段已经被加载进了内存了，也做了默认初始化，赋值为0 ,相当于number = 0

所以在初始化阶段中，number按照顺序 先被赋值为 20 后被赋值为10

**调用问题**



可以赋值但是不可以调用

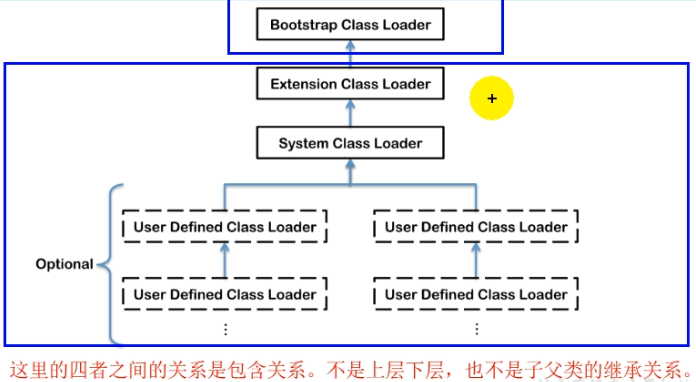
# 类加载器的分类

· JVM支持**两种**类型的类加载器，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有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**都划分为自定义类加载器

·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3个

如图所示



他们之间的关系相当于目录的关系

sun.misc.Launcher 是一个java虚拟机的入口应用

## 引导类加载器

引导类加载器（Bootstrap ClassLoader）

·这个类加载器使用c/c++语言实现的，嵌套在JVM的内部

·它用来加载JAVA的核心库（JAVA\_HOME/jre/lib/rt.jar、resources.jar、sun.boot.class.path路径下的内容），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

·加载扩展类加载器和应用程序加载器，并制定为他们的父 类加载器

·出于安全考虑，引导类加载器值加载报名为java,javax,sun 等开头的类。

## 扩展类加载器

扩展类加载器(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 子目录下加载类库，如果用户创建的JAR放在此目录下，也会自动由扩展类加载器加载

## 系统类加载器

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

·父类加载器为扩展类加载器

·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应用的类都是由他来完成的

·通过ClassLoader#getSystemClassLoader() 方法可以获取到该类加载器

# 为什么要自定义类加载器

在java的日常应用程序开发中，类的加载几乎是由上述三种类加载器相互配合执行的，在必要的时候，我们还可以自定义类加载器，来定制类的加载方式

为什么要自定义类的加载器呢

·隔离加载类

·修改类的加载方式

·扩展加载源

·防止源码泄露

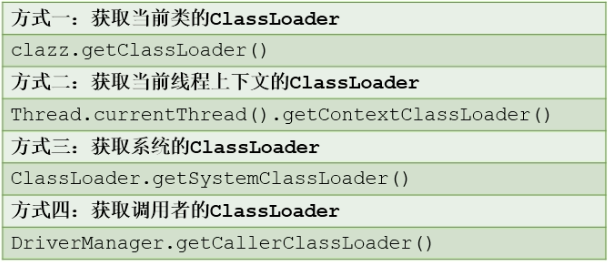
# 自定义类加载器的实现步骤

1. 继承抽象类java.lang.ClassLoader 类的方式，实现自己的类加载器
2. 在JDK 12 前，在自定义类加载器时，总会去继承**ClassLoader**抽象类并重写**loadClass**()，从而实现自定义的类加载器，但是在JDK12之后已不再建议用户去覆盖loadClass() ，而是建议把自定义的类加载逻辑写在**findClass**() 方法中。
3. 在编写自动定义类加载器时，如果没有太过于复杂的需求，可以直接继承**URLClassLoader** 类，**这样就可以避免自己去编写findClass()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。

# 7、Classsloader的api



# 8、获取classloader 的途径

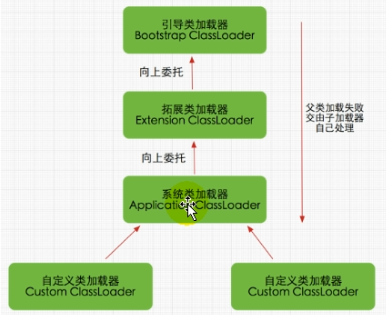


# 双亲委派机制

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的时候才会将他的class文件加载到内存生成class对象，而且加载某个类的class文件时，java虚拟机采用的是双亲委派模式，即把请求交由父类处理。

***双亲委派模式***

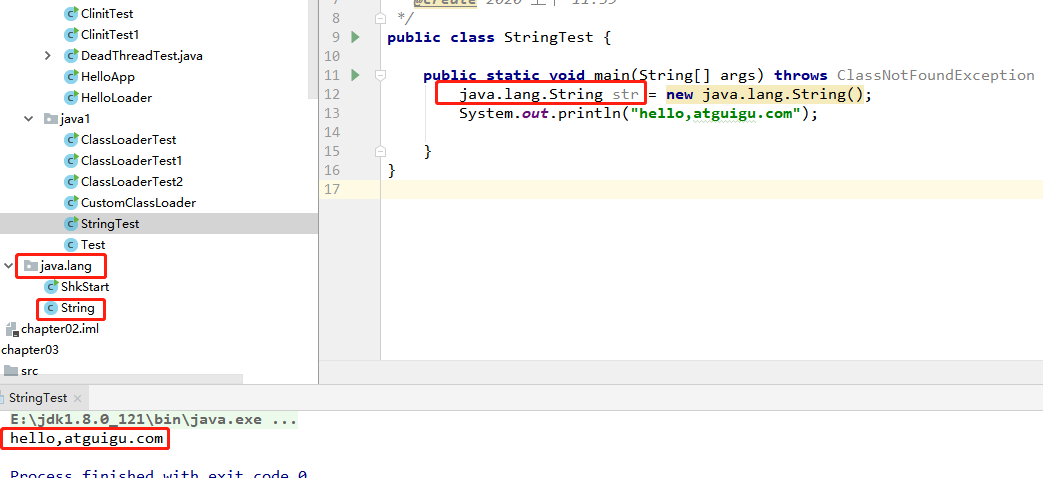
1. 如果一个类加载器收到了**类加载请求**，他并不会自己先去加载，而是把这个请求委托给**父类**的加载器去执行。
2. 如果父类加载器还存在其父类加载器，**则进一步向上委托**，依次递归，请求最终将到达顶层的引导类加载器
3. 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式



**实例**：在自己的工程里面建一个java.lang.String 类，最后加载出来的是rt.jar 中的java.lang.String类

因为这个类在加载的时候，appclassloader向上委托 extclassloader。 extclassloader 向上委托bootstrapclassloader ，最后由bootstrapclassloader加载了位于 rt.jar中的 String 类

如图：

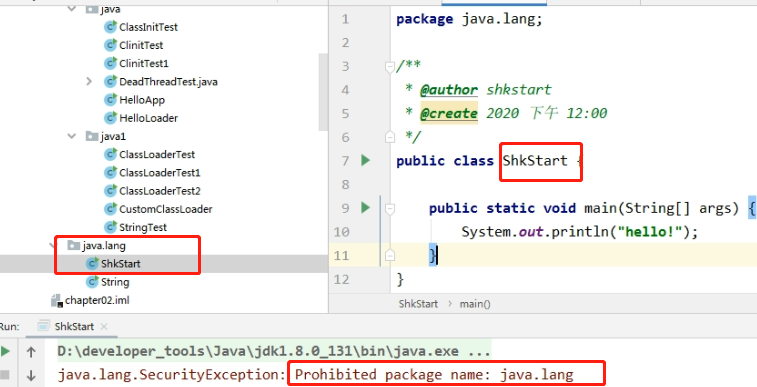


***意义：***

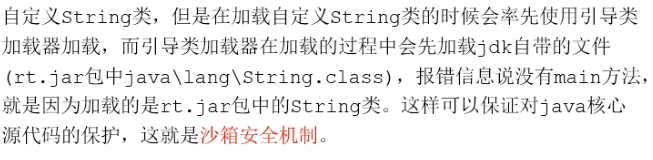
1. 避免类的重复加载
2. 保护程序安全，防止核心API被随意篡改

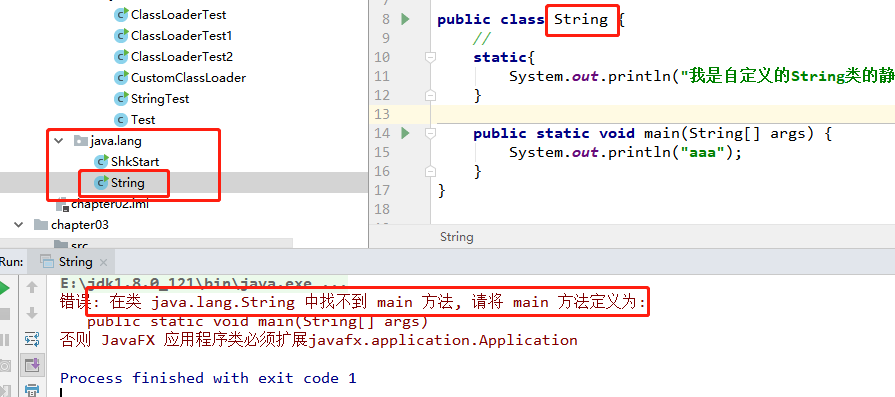
出于安全的考虑

防止自定义类对引导类加载器本身造成破坏



# 沙箱安全机制





# 其他

在JVM中表示两个class对象是否是同一个类的两个必要条件

1. 类的完整类名必须一致
2. 加载这个类的classLoader（指ClassLoader实例对象）必须相同

JVM必须知道一个类型是由启动加载器加载的还是由用户类加载器加载的，如果一个类型是由用户类加载器加载的，那么JVM会**将这个类加载器的一个引用作为类型信息的一部分保存在方法去中（标识这个类是由哪个类加载器加载的）。**当解析一个类型到另一个类型的引用的时候，JVM需要保证这两个类型的类加载器是相同的

