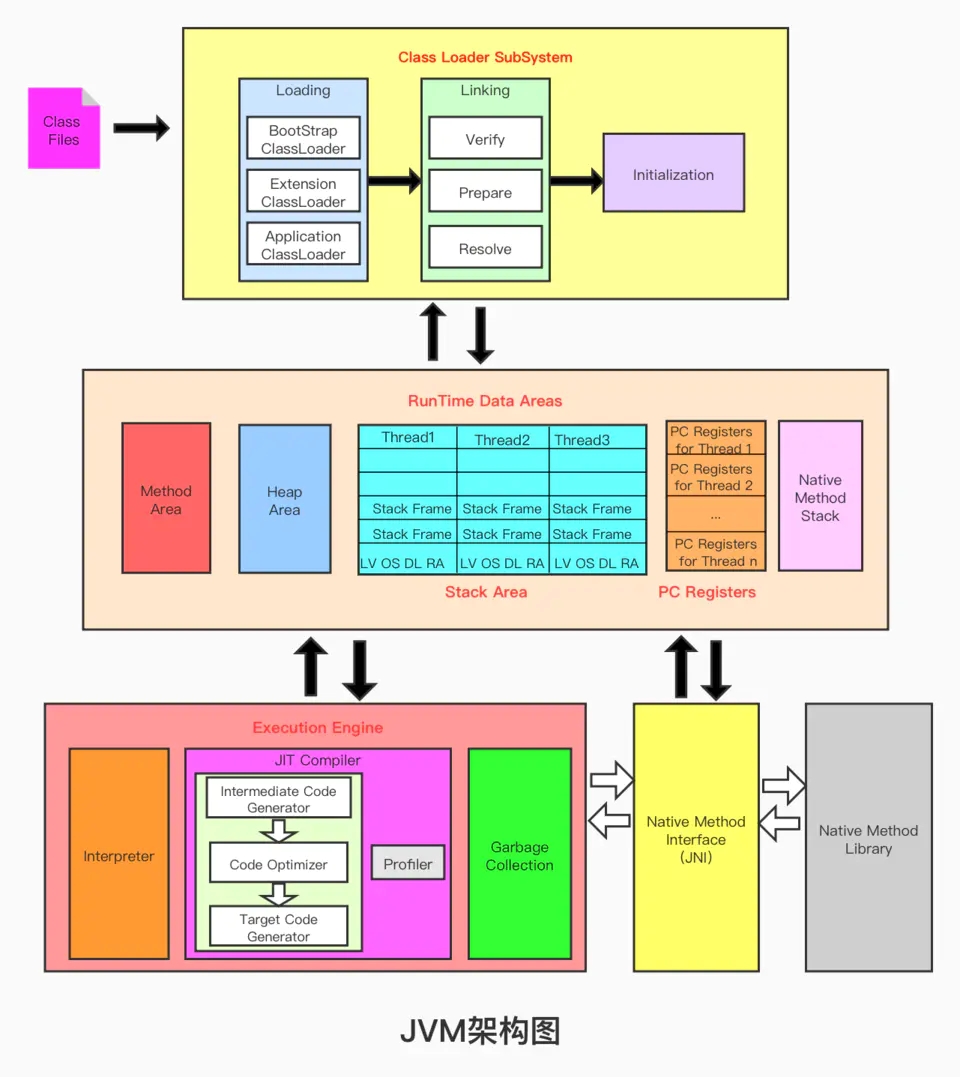
# [26]JVM架构图

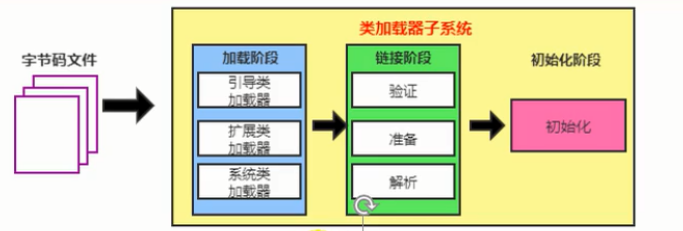


**Java源代码-->编译-->class文件--->类加载子系统--->运行时数据区--->执行引擎**

类加载器和执行引擎是JVM必须要有的

# [27]类加载器子系统以及类加载过程

## 类加载子系统



* 类加载子系统分为三个阶段

加载阶段

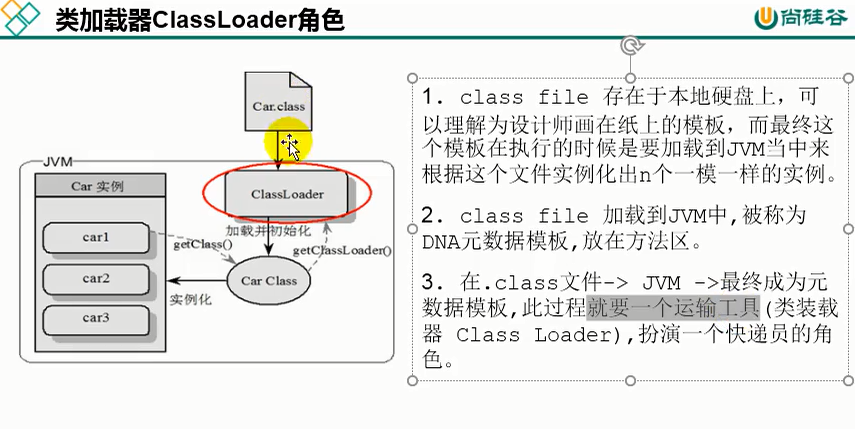
链接阶段：验证->准备->解析

初始化阶段

* 类加载子系统的任务

1. 负责加载.class文件，加载的类信息存放在方法区中；除了类信息，方法区还会存放运行时常量池信息，这一块对应.class文件中的常量池，还包括字符串常量和数字常量，这部分常量信息是.class文件中常量池部分的内存映射
2. ClassLoader只负责加载class文件，不负责运行，是否可以运行由执行引擎决定。（媒婆只负责说媒，生孩子（运行）靠你自己）

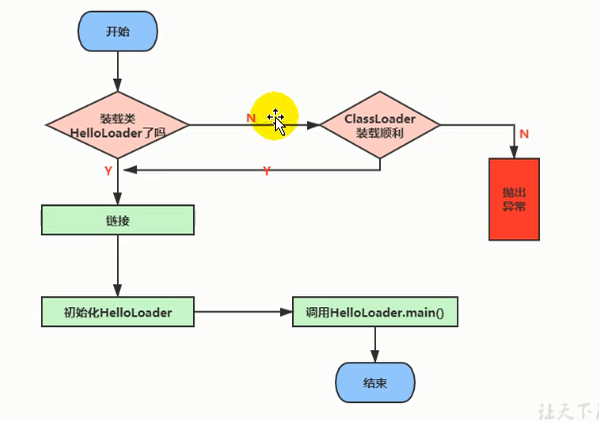
* 类加载器的角色



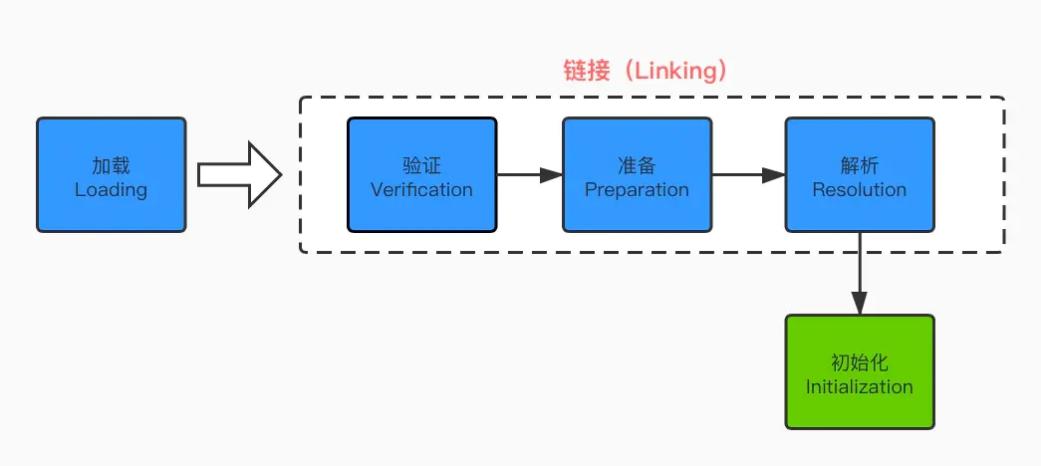
1. .class文件相当于设计师画在图纸上的模型，加载到JVM内存中根据这个图纸实例化出n个一摸一样的实例
2. .class文件加载到JVM中，叫做DNA元数据模板，存在方法区
3. .class文件加载到JVM,然后形成DNA元数据模板，这个过程就需要类加载器来完成
4. 图中可以看到实例对象和元数据模板以及类加载器三者的关系：
   1. 对象.getClass()获取Class对象
   2. Class对象可以获取类加载器

* 类加载过程演示：





## 2.类加载过程[28]

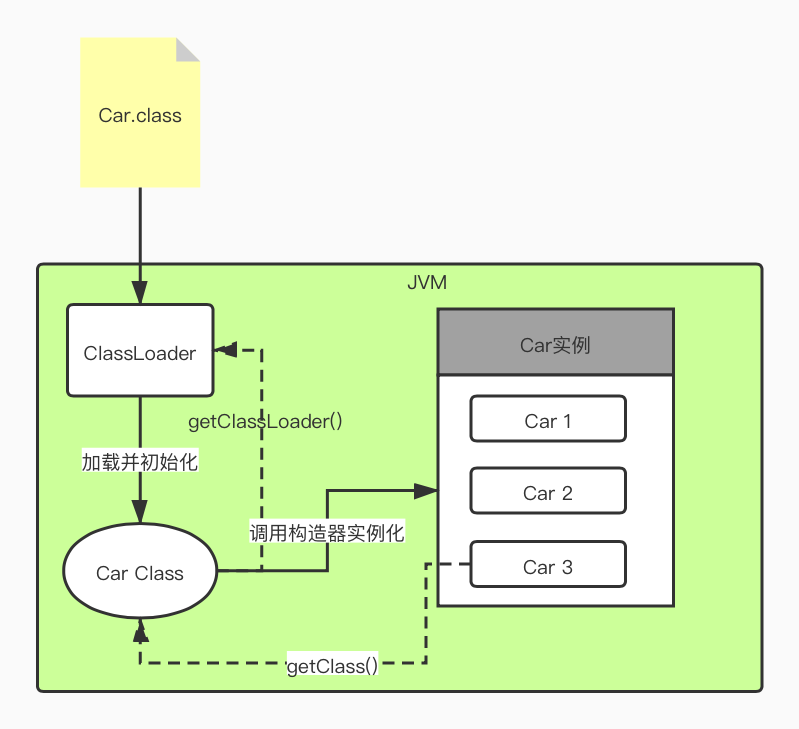


### (1)加载

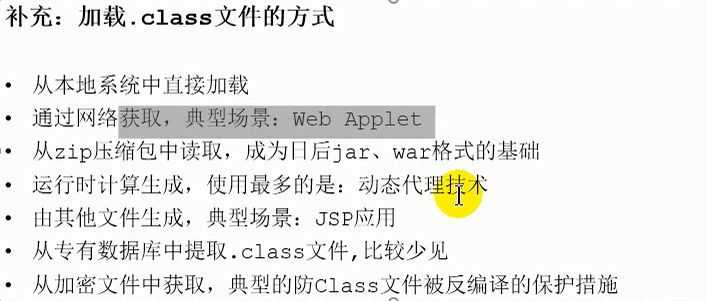
1.通过一个类的全限定明获取定义此类的二进制字节流；

2.将这个字节流所代表的的**静态存储结构**转化为方法区的**运行时数据**；

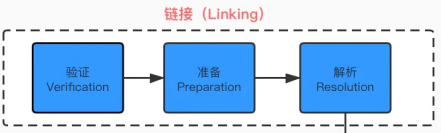
3.加载在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口



* .class文件的获取方式

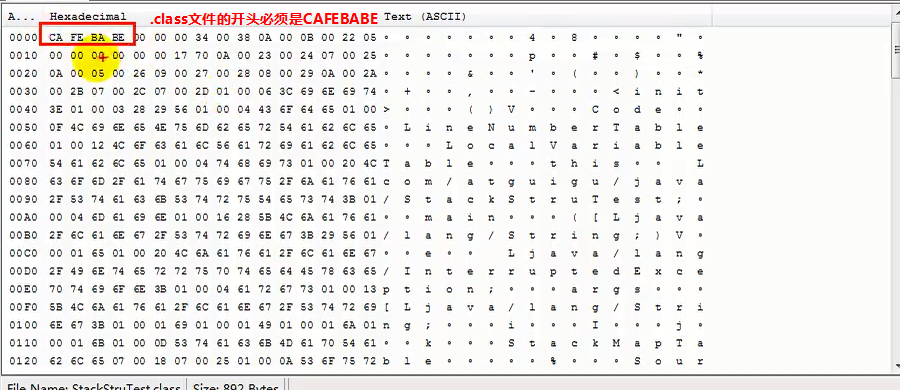


### (2)链接



#### 验证

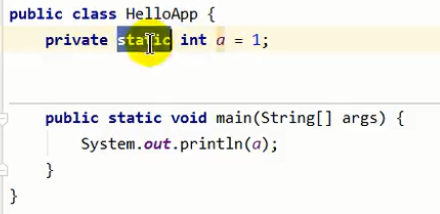
* 目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。
* 主要包括四种验证，文件格式验证，源数据验证，字节码验证，符号引用验证。



#### 准备

准备阶段针对类变量做初始化工作

* 为类变量(static **非final变量**)分配内存并且设置该类变量的默认初始值；
* 对于final修饰的staic变量，因为final在编译的时候就会分配了，准备阶段会显式初始化；
* 准备阶段不会为实例变量分配内存和初始化，类变量是分配在方法区中，而实例变量是随着对象一起分配到java堆中。



Public static int value1= 123;

Public static final int value2 = 123;

Value1在准备阶段过后值为0

Value2在准备阶段过后为123

#### 解析

解析阶段针对符号引用

* 解析的目的：将常量池内的**符号引用**转换为**直接引用**。

事实上，解析操作往往伴随着jvm在执行完初始化之后再执行

说明：

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、相对偏移量或一个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、CONSTANT\_Methodref\_info等。

### (3)初始化

定义: 初始化阶段就是执行类构造器方法clinit()的过程。

(1)此类构造器方法不需要定义，是javac编译器自动收集类中的所有**类变量的显示赋值和静态代码块中的语句合并而来**。编译器收集的顺序是由语句在java原文件中出现的顺序决定。

代码验证

|  |
| --- |
|  |

(2)静态代码块只能访问到定义在静态代码块之前的变量，定义在其后面的变量，在静态代码块中可以赋值，但是不能访问。

代码验证：

|  |
| --- |
| 可以在声明之前赋值：      但是不能在声明之前调用： |

之所以静态代码块可以直接赋值，因为静态变量在准备阶段就已经赋予默认值了，然后这里是解析阶段，会按照其赋值顺序进行赋值。

(3)<clinit>()方法对于接口和类来说不是必须的，如果类中没有静态属性和静态代码块，那么字节码文件中就不会有clinit方法

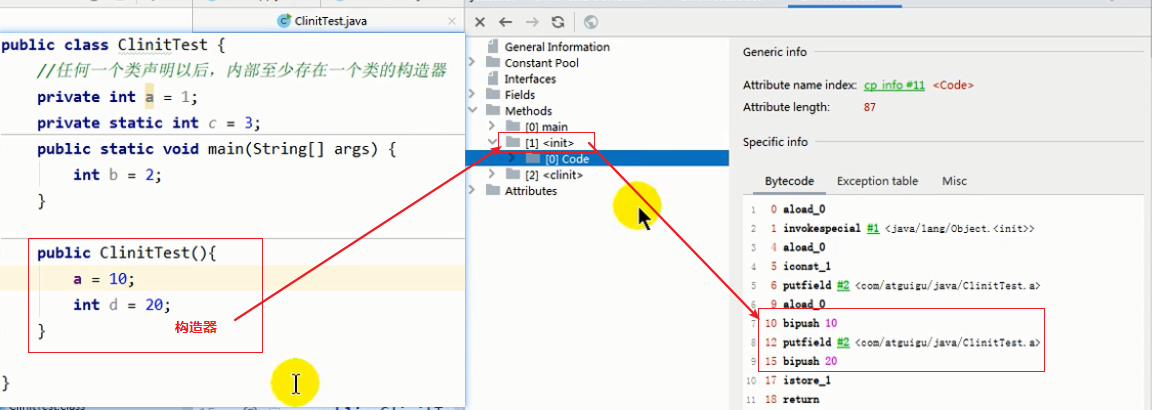
代码验证：

|  |
| --- |
|  |

现象解释：

<clinit>()与<init>()方法不同，<clinit>()不需要显式的调用父类构造器。而<init>()是对构造器方法的封装，一个构造器对应一个<init>()方法，该方法只有在调用构造器创建对象的时候使用。

而每个类中，至少有一个构造器，因此每个类都会有<init>()方法。



(4)父类初始化于子类初始化之前，父类<clinit>()在子类<clinit>()之前执行。

jvm会保证子类的<clinit>()执行前，父类的<clinit>()已经执行完毕，因此可以推断出第一个被执行的<clinit>()一定是java.lang.Object

* + 父类一定在子类前初始化
  + 父类的静态代码块、静态变量的赋值语句一定在子类的静态代码块、静态变量的赋值语句前执行。

代码验证：

|  |
| --- |
|  |

(5)虚拟机必须保证一个类的clinit()方法在多线程下被同步加锁。

即一个类只需被clinit一次，之后该类的内部信息就被存储在方法区。

代码验证：

|  |
| --- |
| 可以看出来static代码块只被执行了一次，因为类初始化只一次。 |

# 类加载器分类

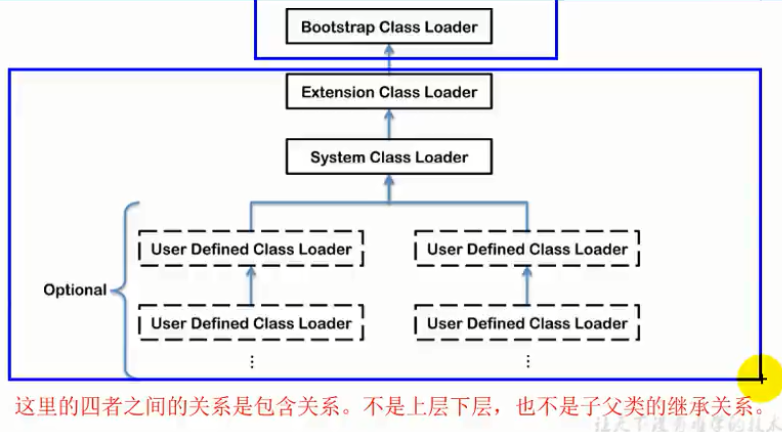
## 按照JVM规范分类

* JVM规范声明了两种类型的加载器：

1. 引导类加载器(BootStrap ClassLoader) C/C++实现
2. 自定义类加载器 Java实现

从概念上来讲，自定义类加载器一般指的是程序中由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。

除了Bootstrap Class Loader是C/C++实现的，因此不派生于抽象类ClassLoader，属于引导类加载器；Extension和System Class Loader都是派生于ClassLoader，都是自定义类加载器。



## 按照加载内容分类

### 引导类加载器（Bootstrap Classloader）

* 又称为根类加载器
* 加载内容

负责加载Java的核心库(JAVA\_HOME/jre/lib/**rt.ja**r等或sun.boot.class.path路径下的内容)

出于安全考虑，BootStrap启动类加载器只加载包名为java、javax、sun等开头的类

用于提供JVM自身需要的类

* 实现方式

用原生代码(C/C++)来实现的，嵌套在JVM内部，并不继承自java.lang.ClassLoder，没有父加载器，所以通过Java代码获取引导类加载器对象将会得到null

* 还可以加载自定义类加载器

加载拓展类和应用程序类加载器，并指定为他们的父加载器，即ClassLoader

|  |
| --- |
| @Test public void test01(){  *// 普通类的类加载器* Class clazz = \_01\_.class;  ClassLoader loader = clazz.getClassLoader();  System.*out*.println(loader);*//sun.misc.Launcher$AppClassLoader@18b4aac2*  *//通过rt.jar包下的类来获取其类加载器* Class<String> stringClass = String.class;  ClassLoader classLoader = stringClass.getClassLoader();  System.*out*.println(classLoader);*//null* } |

### 扩展类加载器（Extension ClassLoader）

1.java语言编写 ，由sun.misc.Launcher$ExtClassLoader实现。

2.派生于java.lang.ClassLoader类

3.负责加载java.ext.dirs路径下的类库，或JDK的安装目录的jre/lib/**ext子目录**下的类库。如果用户创建的JAR放在此目录下，也会由拓展类加载器自动加载

### 应用程序类加载器（Application Classloader）

1.java语言编写， 由sun.misc.Launcher$AppClassLoader实现。

2.派生于java.lang.ClassLoader类

3.它负责加载环境变量classpath或系统属性 java.class.path下的类库

classPath= 项目路径bin文件夹 或者 配置了的环境变量classPath

4.该类加载器是程序中默认的类加载器，一般来说，java应用的类都是由它来完成加载

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

### 自定义类加载器

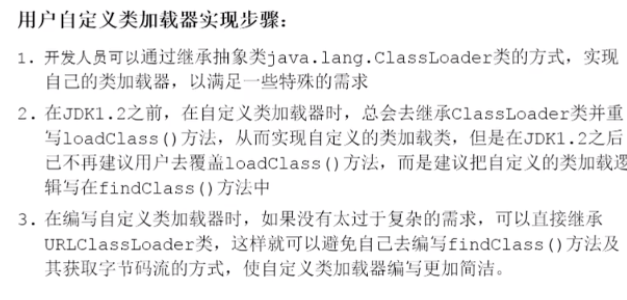
开发人员可以通过继承java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求.

1.隔离加载类

2.修改类加载的方式

3.拓展加载源

4.防止源码泄漏



### 查看类加载器

|  |
| --- |
| /\*\*  \* ClassLoader加载  \*/  public class ClassLoaderTest {  public static void main(String[] args) {  //获取系统类加载器  ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();  System.out.println(systemClassLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2  //获取其上层 扩展类加载器  ClassLoader extClassLoader = systemClassLoader.getParent();  System.out.println(extClassLoader);//sun.misc.Launcher$ExtClassLoader@610455d6  //获取其上层 获取不到引导类加载器  ClassLoader bootStrapClassLoader = extClassLoader.getParent();  System.out.println(bootStrapClassLoader);//null  //对于用户自定义类来说：使用系统类加载器进行加载  ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();  System.out.println(classLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2  //String 类使用引导类加载器进行加载的 -->java核心类库都是使用引导类加载器加载的  ClassLoader classLoader1 = String.class.getClassLoader();  System.out.println(classLoader1);//null获取不到间接证明了String 类使用引导类加载器进行加载的  }  } |

### 查看类加载器能加载哪些类

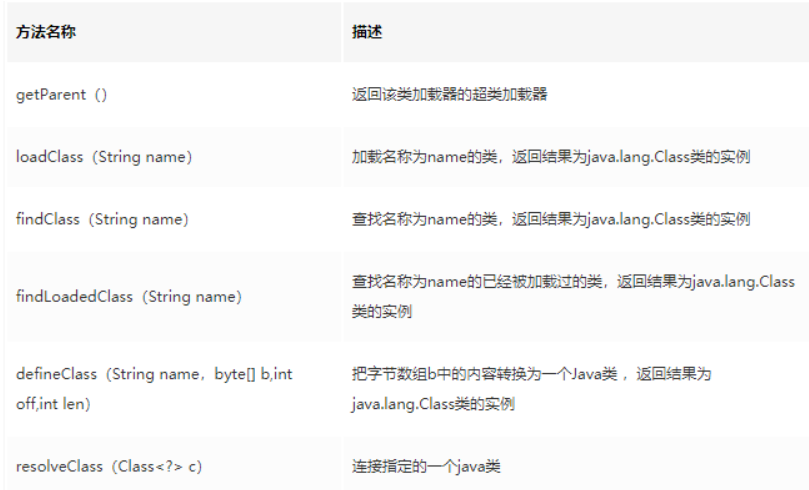
|  |
| --- |
| /\*\*  \* 虚拟机自带加载器  \*/  public class ClassLoaderTest1 {  public static void main(String[] args) {  System.out.println("\*\*\*\*\*\*\*\*启动类加载器\*\*\*\*\*\*\*\*\*");  URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs();  //获取BootStrapClassLoader能够加载的api路径  for (URL e:urls){  System.out.println(e.toExternalForm());  }  //从上面的路径中随意选择一个类 看看他的类加载器是什么  //Provider位于 /jdk1.8.0\_171.jdk/Contents/Home/jre/lib/jsse.jar 下，引导类加载器加载它  ClassLoader classLoader = Provider.class.getClassLoader();  System.out.println(classLoader);//null  System.out.println("\*\*\*\*\*\*\*\*拓展类加载器\*\*\*\*\*\*\*\*");  String extDirs = System.getProperty("java.ext.dirs");  for (String path : extDirs.split(";")){  System.out.println(path);  }  //从上面的路径中随意选择一个类 看看他的类加载器是什么:拓展类加载器  ClassLoader classLoader1 = CurveDB.class.getClassLoader();  System.out.println(classLoader1);//sun.misc.Launcher$ExtClassLoader@4dc63996  }  } |



java.util.concurrent包的内容，发现java.time.、java.util.、java.nio.、java.lang.、java.text.、java.sql.、java.math.\*等等都在rt.jar包下

# [34]ClassLoader的常用方法及获取方法

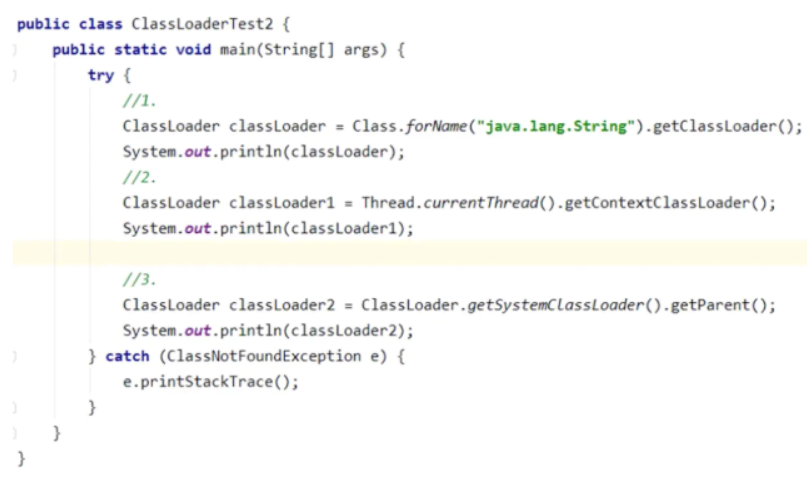
ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自ClassLoader（不包括启动类加载器）



## **ClassLoader继承关系**



## **代码示例**

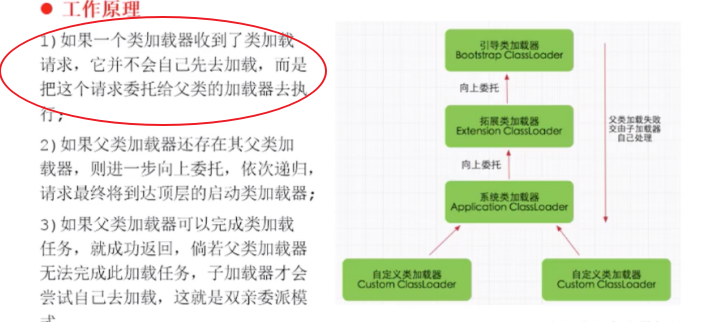


# [35]双亲委派机制

1.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时才会将她的class文件加载到内存生成的class对象。

2.JVM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，它是一种**任务委派模式**

* **工作原理**



买了个苹果，问妈妈吃不吃，妈妈问奶奶吃不吃，奶奶说太硬了吃不动，交给妈妈吃，妈妈说太酸了，交给自己吃。

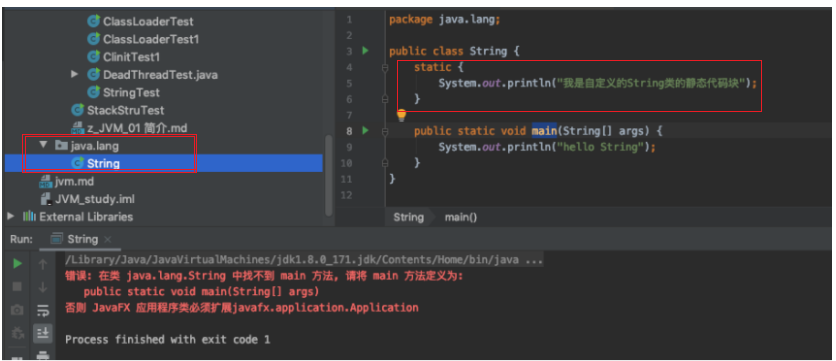
* **代码示例**

**例1：**

|  |
| --- |
| 自定义java.lang.String类，发现Test并没有打印静态代码块中内容，说明没有加载自定义String类： |

**例2：**

虽然我们自定义了一个java.lang包下的String尝试覆盖核心类库中的String，但是由于双亲委派机制，启动加载器会加载java核心类库的String类（BootStrap启动类加载器只加载包名为java、javax、sun等开头的类），而核心类库中的String并没有main方法，因此报如下错误：



# [36]双亲委派机制的优势

1. 避免类的重复加载

一旦被加载了，就不会被别的类加载器加载了

1. 保护程序安全，防止核心API被随意修改【和核心类库同名的类不会被加载】

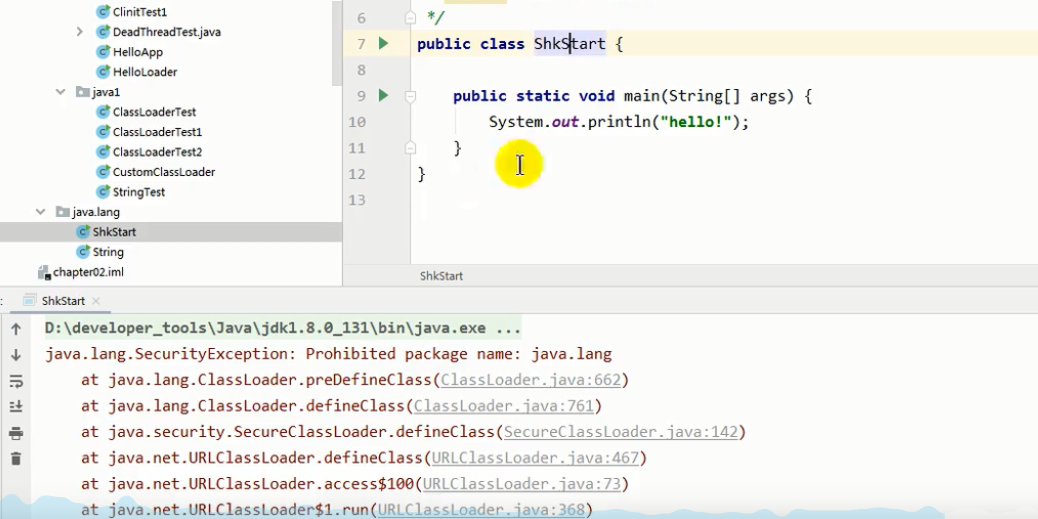
启动类加载器可以抢在标准扩展类装载器之前去装载类，而标准扩展类装载器可以抢在类路径加载器之前去装载那个类，类路径装载器又可以抢在自定义类加载器之前去加载它，所以Java虚拟机先从最可信的Java核心API查找类型，这是为了防止不可靠的类扮演被信任的类。试想一 下，网络上有个名叫java.lang.Integer的类，它是某个黑客为了想混进java.lang包所起的名字，实际上里面含有恶意代码，但是这种伎俩在双亲模式加载体系结构下是行不通的，因为网络类加载器在加载它的时候，它首先调用双亲类加载器，这样一直向上委托，直到启动类加载器，而启动类加载器在核心Java API里发现了这个名字的类，所以它就直接加载Java核心API的java.lang.Integer类，然后将这个类返回，所以自始自终网络上的 java.lang.Integer的类是不会被加载的。【自定义全限定名完全相同的类不会被加载，只会被父类加载器检测到该名称属于自己的加载范围，从而加载自己的类】

1. 保证核心API包的访问权限【插入到核心类库jar包需要经过运行时包的同意】

但是如果这个移动代码不是去试图替换一个被信任的类（就是前面说的那种情况），而是想在一个被信任的包中插入一个全新的类型，情况会怎样呢？比如一个名为 java.lang.Virus的类，经过双亲委托模式，最终类装载器试图从网络上下载这个类，因为网络类装载器的双亲们都没有这个类（当然没有了，因为 是病毒嘛）。假设成功下载了这个类，那你肯定会想，Virus和lang下的其他类同在java.lang包下，暗示这个类是Java API的一部分，那么是不是也拥有修改Java.lang包中数据的权限呢？答案当然不是，因为要取得访问和修改java.lang包中的权 限，java.lang.Virus和java.lang下其他类必须是属于同一个运行时包的，什么是**运行时包**？运行时包是指由同一个类装载器装载的、属 于同一个包的、多个类型的集合。考虑一下，java.lang.Virus和java.lang其他类是同一个类装载器装载的吗？不是 的！java.lang.Virus是由网络类装载器装载的！

**代码示例：**

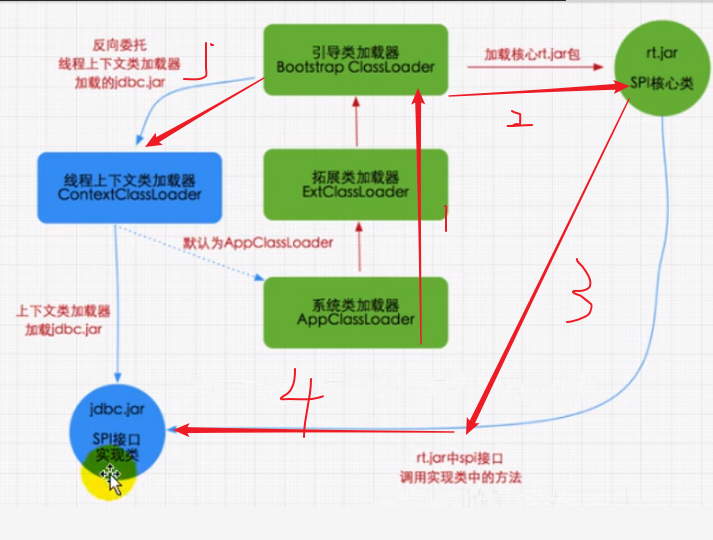
在自定义java.lang包中定义一个自定义类：



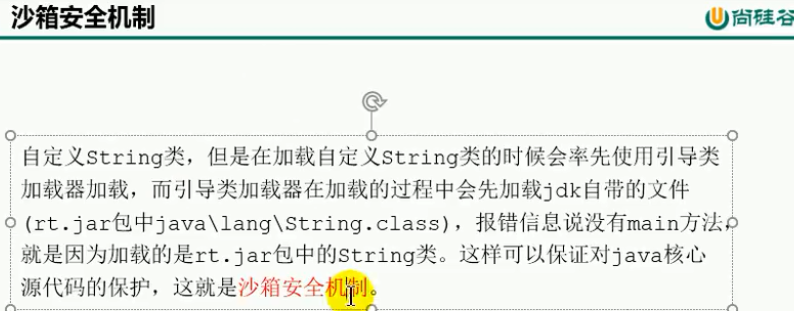
由于是java.lang包，因此由BootStrap类加载器加载，但是该类加载器没有发现有此类，因此报安全异常。

## 双亲委派机制在SPI中的应用

某个应用程序由双亲委派机制找到引导类加载器，首先调用rt.jar包中的SPI核心，但由于SPI核心当中有各种各样的接口需要被实现（这里指具体的服务提供商），这里我们以JDBC.jar为例，jdbc.jar可以为我们提供具体的实现。那么这时我们需要反向委托，找到线程上下文类加载器去加载jdbc.ja**(线程上下文类加载器属于系统类加载器)**

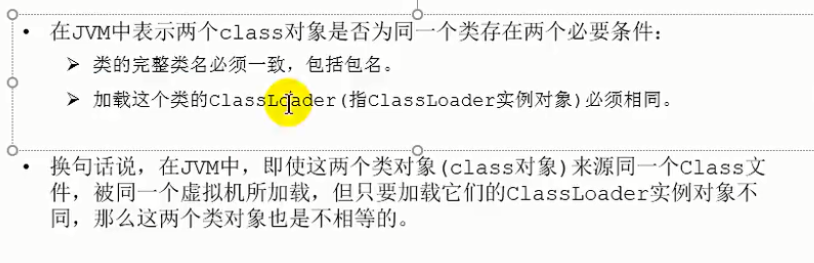


# [37]沙箱安全机制



# [39]JVM内存中Class对象唯一性确定

* **Class唯一性确定**



* **Class对象对类加载器的引用**

JVM必须知道一个类型是有启动类加载器加载的还是由用户类加载器加载的。如果一个类型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区中**。当解析一个类型到另一个类型的引用的时候，JVM需要保证两个类型的加载器是相同的。（动态链接中会再次说明）

# 类的主动使用和被动使用

java程序对类的使用方式分为：主动使用和被动使用，二者的区别在于是否调用了<clinit>()方法

主动使用则会在类加载系统中的第三阶段initialization即初始化阶段调用了<clinit>()方法。而被动使用不会去调用

* **主动使用，分为七种情况**

1.创建类的实例

2.访问某各类或接口的静态变量，或者对静态变量赋值

3.调用类的静态方法

4.反射 比如Class.forName(com.dsh.jvm.xxx)

5.初始化一个类的子类

6.java虚拟机启动时被标明为启动类的类

7.JDK 7 开始提供的动态语言支持：java.lang.invoke.MethodHandle实例的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic句柄对应的类没有初始化，则初始化

除了以上七种情况，其他使用java类的方式都被看作是对类的被动使用，都不会导致类的初始化。

# 破坏双亲委派模型

## \*类加载时机

Jvm规范中没有规定什么时候对类进行加载，但是对类的初始化时机进行了严格的规定。

加载 验证 准备 初始化 这四个阶段是按顺序执行的，因此可以根据初始化时机来判断类加载时机。

### \*什么时候进行初始化？【主动引用】

总的来说，在遇到**new、getstatic、putstatic、invokestatic**这四个字节码指令时，如果类没有进行过初始化，那么就必须先触发其初始化。

**也就是说在创建对象、设置静态属性、获取静态属性、调用静态方法的时候。**

1. 用new实例化对象的时候
2. 用java.lang.reflect包的方法对类型进行反射调用的时候
3. 读取或者设置一个类型的静态字段。 但是被final修饰的静态字段在编译期间就把值放进了常量池，此时不会触发
4. 初始化类的时候，发现父类还没有初始化，会触发父类的初始化
5. main方法所在的类，在jvm启动的时候会先初始化这个类
6. 拥有default方法的接口，其实现类要初始化，会先初始化这个接口。

### \*什么时候不进行初始化？【被动引用】

1. 通过子类调用父类的父类的静态字段，不会导致子类初始化。
2. 创建元素为该类类型A元素的数组的时候，只会触发A[]类型的初始化，而不会触发A类型的初始化。
3. **调用该类的final static属性时，不会触发此类的初始化。**

**Final static的属性，在声明的时候就赋值了**

**Static的属性和在静态代码块中赋值是一样的效果，在类初始化的时候触发赋值**

# 类加载器

类加载器的功能：”通过一个类的全限定名来获取描述此类的二进制字节码流”

## 类与类加载器

### 类加载器的类型

从JVM的角度来看，只存在两种类加载器：

1. 启动类加载器，此类加载器**由C++实现**，**是JVM的组成部分**（Bootstrap ClassLoader）
2. 其他类加载器，这些类加载器**由Java实现**，**独立于JVM之外，都继承抽象类java.lang.ClassLoader**

从程序员角度来看，类加载器可以细分：

1. **启动类加载器：**<JAVA\_HOME>\lib目录中的，此类加载器不能索引到，为null

BootStrap启动类加载器只加载包名为java、javax、sun等开头的类

**2.扩展类加载器：**Extension ClassLoader负责加载<JAVA\_HOME>\lib\ext目录

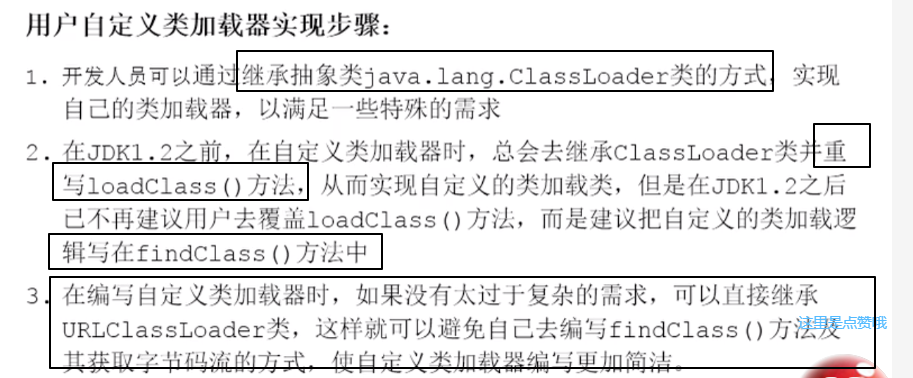
**3.应用程序类加载器：**Application ClassLoader

这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值。

一般也称为系统类加载器。负责加载用户类路径上所指定的类库，开发者可以直接适用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

|  |
| --- |
| /\*\*  \* ClassLoader加载  \*/  public class ClassLoaderTest {  public static void main(String[] args) {  //获取系统类加载器  ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();  System.out.println(systemClassLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2  //获取其上层 扩展类加载器  ClassLoader extClassLoader = systemClassLoader.getParent();  System.out.println(extClassLoader);//sun.misc.Launcher$ExtClassLoader@610455d6  //获取其上层 获取不到引导类加载器  ClassLoader bootStrapClassLoader = extClassLoader.getParent();  System.out.println(bootStrapClassLoader);//null  //对于用户自定义类来说：使用系统类加载器进行加载  ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();  System.out.println(classLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2  //String 类使用引导类加载器进行加载的 -->java核心类库都是使用引导类加载器加载的  ClassLoader classLoader1 = String.class.getClassLoader();  System.out.println(classLoader1);//null获取不到间接证明了String 类使用引导类加载器进行加载的  }  } |

### 自定义类加载器



自定义类加载器的目的：

1.隔离加载类

2.修改类加载的方式

3.拓展加载源

4.防止源码泄漏