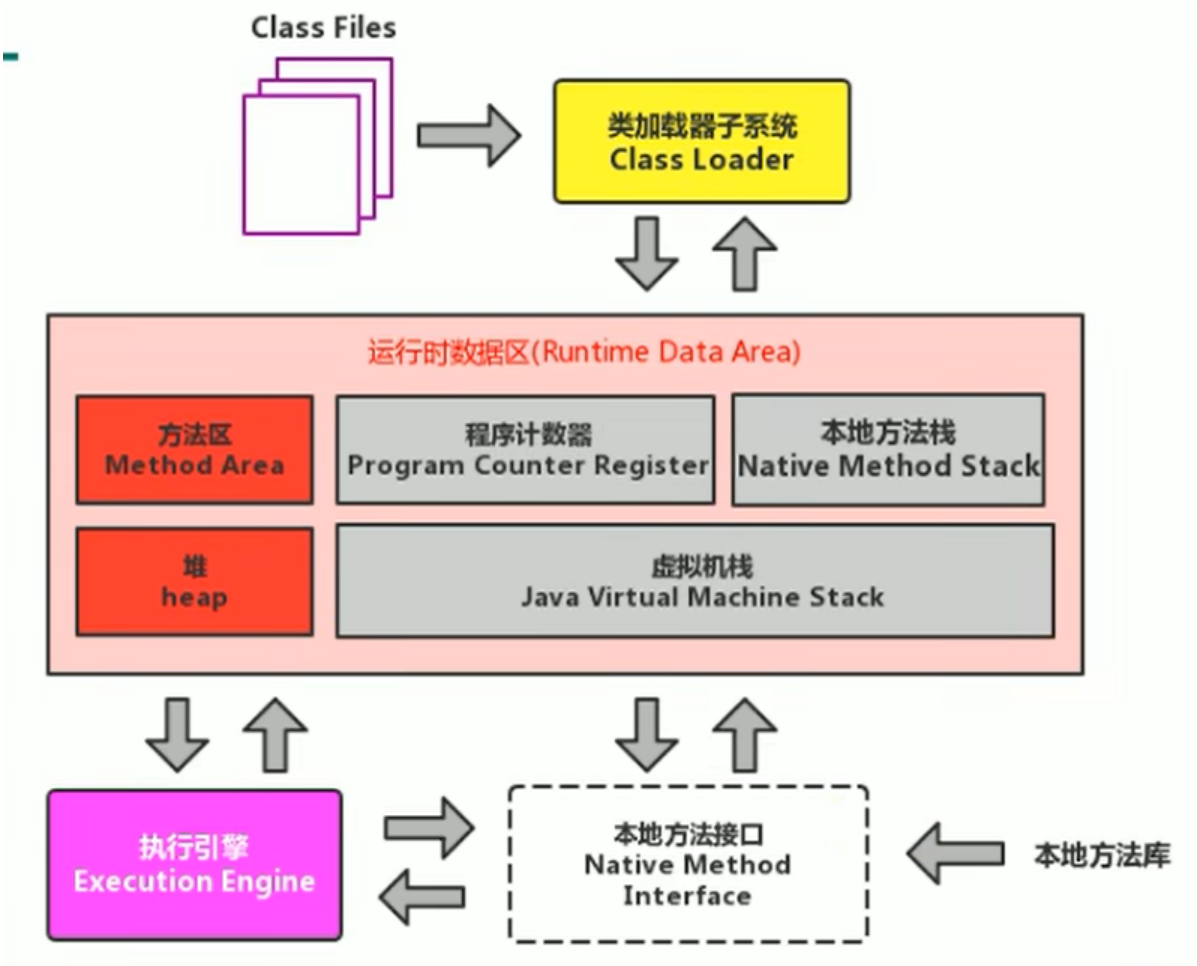
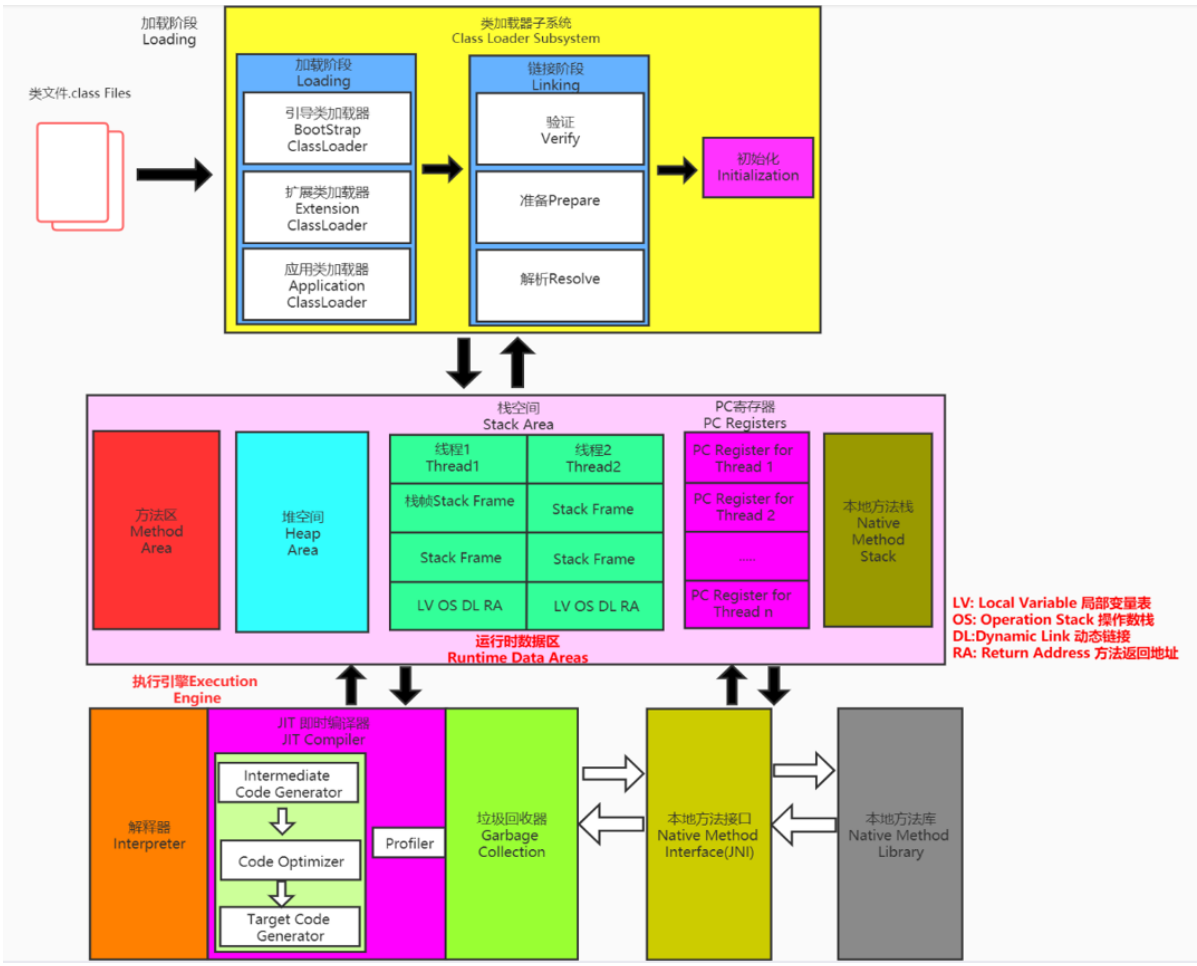
二、类加载子系统

1、内存结构概述

1.1 简单内存结构示意图



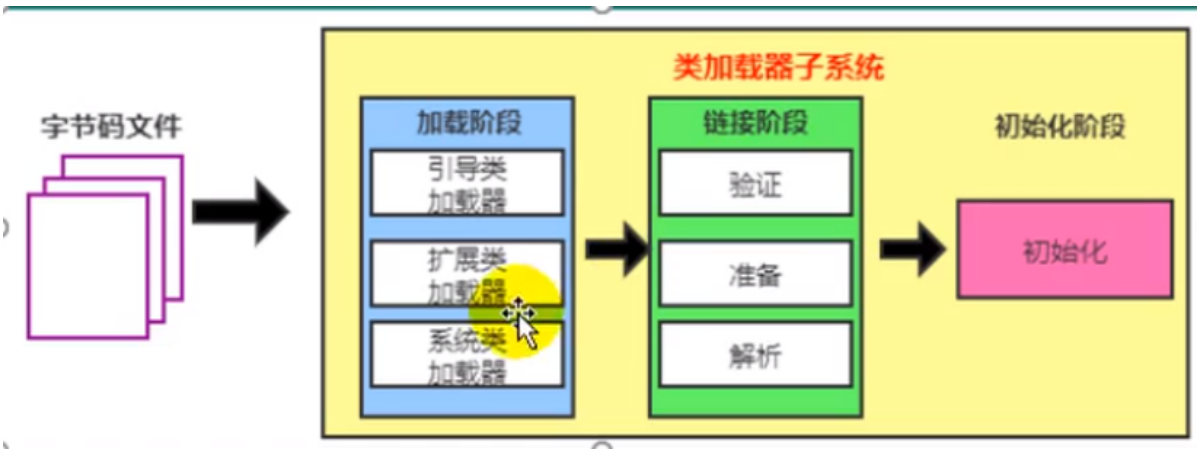
1.2 详细内存结构图



2、类加载器与类的加载过程

2.1 类加载子系统作用

1. 类加载子系统图示



2. 类加载子系统的作用

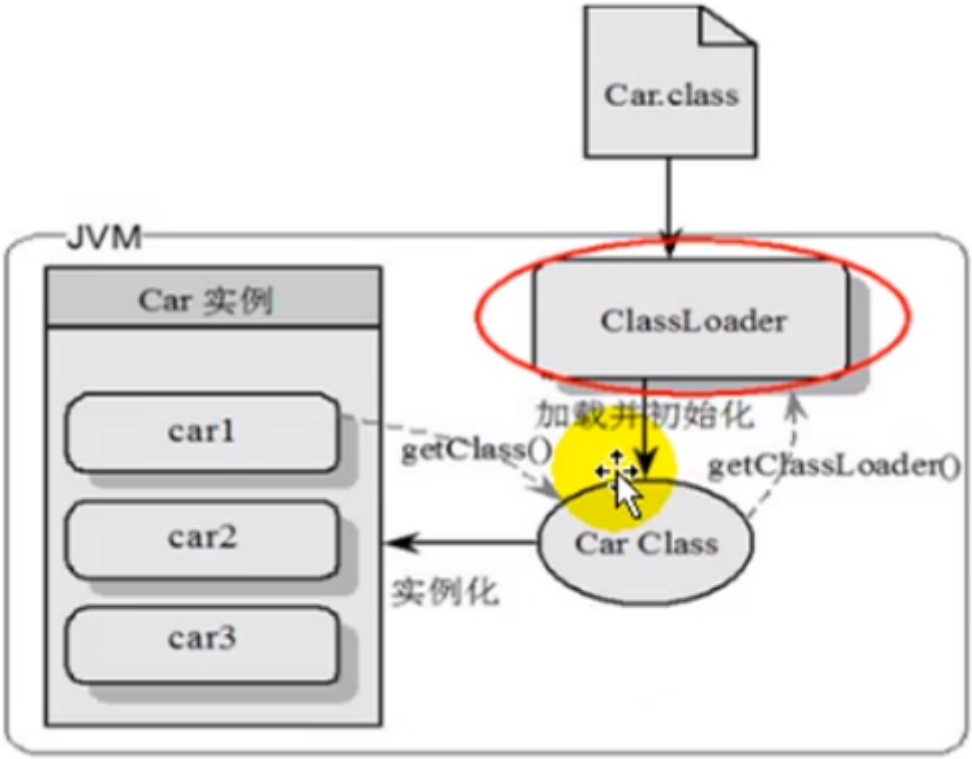
（1）类加载器子系统负责从文件系统或者网络中加载Class文件，class文件（任何语言可以生成符合java虚拟机规范的class文件）在文件开头有特定的文件标识。

（2）ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定。

（3）加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中还会存放运行时常量池信息，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）

2.2 类加载器ClassLoader角色

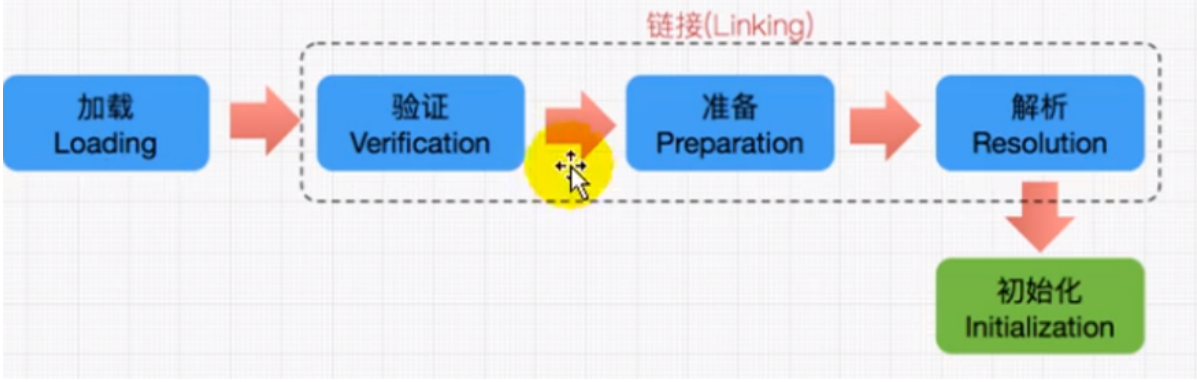
具体例子如下



- 此时我们有一个Car类的字节码文件Car.class,该字节码文件（class file）保存在本地硬盘上（对于字节码文件，可以将其理解为设计师画在纸上的模版），而最终这个模版在执行的时候是要加载到JVM当中来根据这个文件实例化出n各一模一样的实例。
- class file 加载到JVM中，被称为DNA元数据模版（即上图中的Car Class），放在方法区中（其是通过类加载器实现的）。可以通过调用getClassLoader获得当前DNA元数据模版的类加载器，可以通过其构造方法，创建对应的几个Car实例（类的对象），Car实例可以通过getClass方法获取其Class对象。
- 在.class文件 -> JVM -> 最终成为元数据模板，此过程就要一个运输工具（类装载器 Class Loader），扮演一个快递员的角色。

2.3 类加载过程图示

类的加载过程可以分为加载（Loading）、链接（Linking）、以及初始化（initialization），如下图所示



2.4 类的加载过程具体详解

#. 扩展：

IDEA中提供插件jclasslib Bytecode viewer，用来反编译字节码文件

1. 加载（Loading）

- 通过一个类的全限定名获取定义此类的二进制字符流

  获取加载.class文件的方式有如下几种：

  - 从本地系统重直接加载
  - 通过网络获取，典型场景Web Applet（Web Applet通常指的是一个小型的程序，通常是以Java Applet的形式嵌入到网页中运行的。）
  - 从zip压缩包中读取，成为日后jar、war格式的基础
  - 运行时计算生成，使用最多的是：动态代理技术（Proxy）
  - 由其他文件生成，典型场景：JSP应用（Java Server Pages (JSP) 在运行时会被编译成 Java Servlet，然后再由 Servlet 容器执行）
  - 从专用数据库中提取.class文件，比较少见
  - 从加密文件中获取，典型的防Class文件被反编译的保护措施
- 将这个字节流所代表的静态存储结构转化为方法区（jdk7及以前称为永久代，之后称为元数据）的运行时数据结构
- 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

2. 链接（Lingking）

- 验证（Verify）
  - 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信息（字节码文件可以被修改），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（如java的字节码文件其开头一般为ca fe ba be）、元数据验证、字节码验证、符号引用验证。
- 准备（Prepare）
  - 为类变量(被static修饰的变量，即静态变量)分配内存并且设置该类变量的默认初始值（整型：0，浮点：0.0，……数据类型不同，数据的初始值也不同），即零值
  - 这里不包含用final修饰的static（即常量），因为final在编译的时候就会分配对应的值，在准备阶段会显示初始化（如final staitc Integer a = 5，那么此时就会在准备阶段为a复制为5）
  - 这里不会为实例变量分配初始化（此时还只是类的加载过程，还没有创建类的对象），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

- 解析（Resolve）
  - 将常量池内的符号引用转换为直接引用的过程（通过javap -v 字节码文件.class 进行反编译字节码文件）
  - 解析操作会在JVM执行完初始化后再执行
  - 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（Initialization）

- 初始化阶段就是执行`类构造器方法<clinit>()`（注意不是实例化类的那个构造器）的过程

  此方法不需定义，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，并且构造器方法中指令按语句在源文件中出现的顺序执行（注意，如果程序中没有声明类变量和静态代码块，则不会默认生成()类构造器方法）：

```
public class Test{
    // 静态变量1
    private static int num1 = 1;

    // 静态代码块
    static{
        num1 = 2;
        num2 = 20;
        /*
        注意点：非法的前向引用
        System.out.println(Test.num1);  // 合法
        System.out.println(Test.num2);  // 不合法
        */
    }

    // 静态变量1
    private static int num2 = 10;

    public static void main(String[] args){
        System.out.println(Test.num1);
        System.out.println(Test.num2);
    }

}
```

  首先在类加载的连接（即Linking）过程，为类变量num1和num2分配内存并且设置该类变量的默认初始值，即num1 = 0，num2=0

  在initalization的时候，执行()构造方法，其复制的过程依次是：

  num1: 0 -> 1 -> 2

  num2:0 -> 20 -> 10

- ()不同于类的构造器（即实例化对象的那个构造器）(关联:类构造器对应虚拟机下的())，类构造器中的方法，会在()方法中
- 若该类具有父类，JVM会保证子类的()执行前，父类的()已经执行完毕

```
public class Test{

    static class Father {
        public static int A = 1;
        static{
            A = 2;
        }
    }

    static class Son extends Father {
        public static int B = A;
    }

    private static int num2 = 10;

    public static void main(String[] args){
        System.out.println(Son.B);
    }

}
```

说明：当我们需要执行main方法的时候，会将Test类加载到内存中，在main方法中调用Son.B（Son类的静态变量B）的时候，会将Son类加载进来，但是在执行Son类加载前，会加载其父类，中间设计到父类的加载、连接、初始化，在Son加载时候，B初始化时候此时A已经是2了，因此Son.B的答案是2

- 虚拟机必须保证一个类的()方法在多线程下被同步加锁

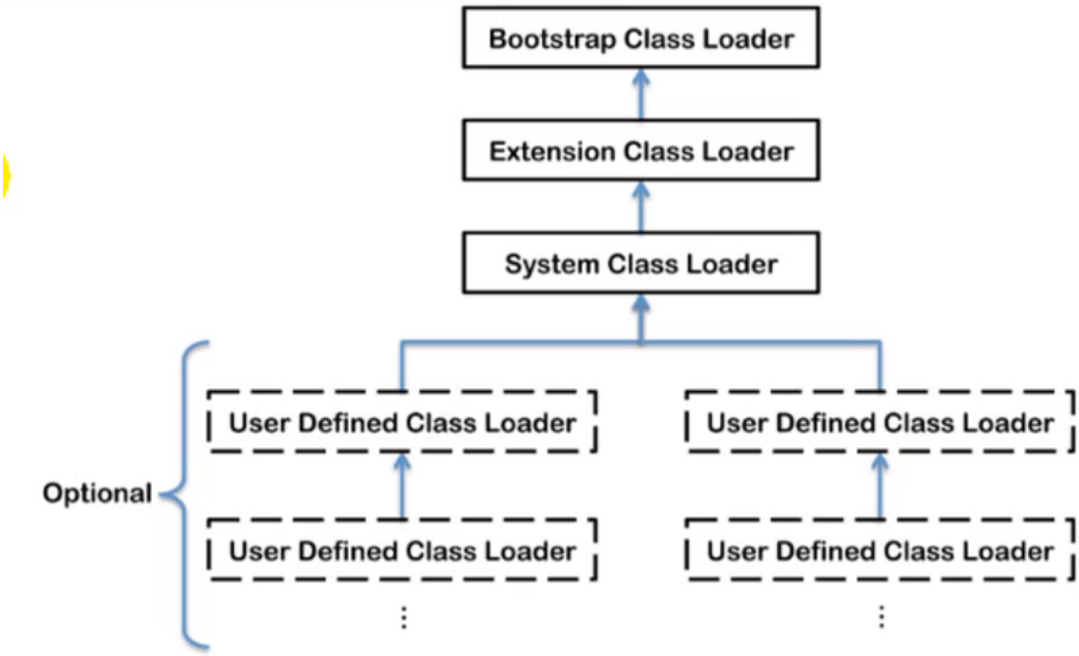
3、 类加载器分类

3.1 几种类加载器的说明

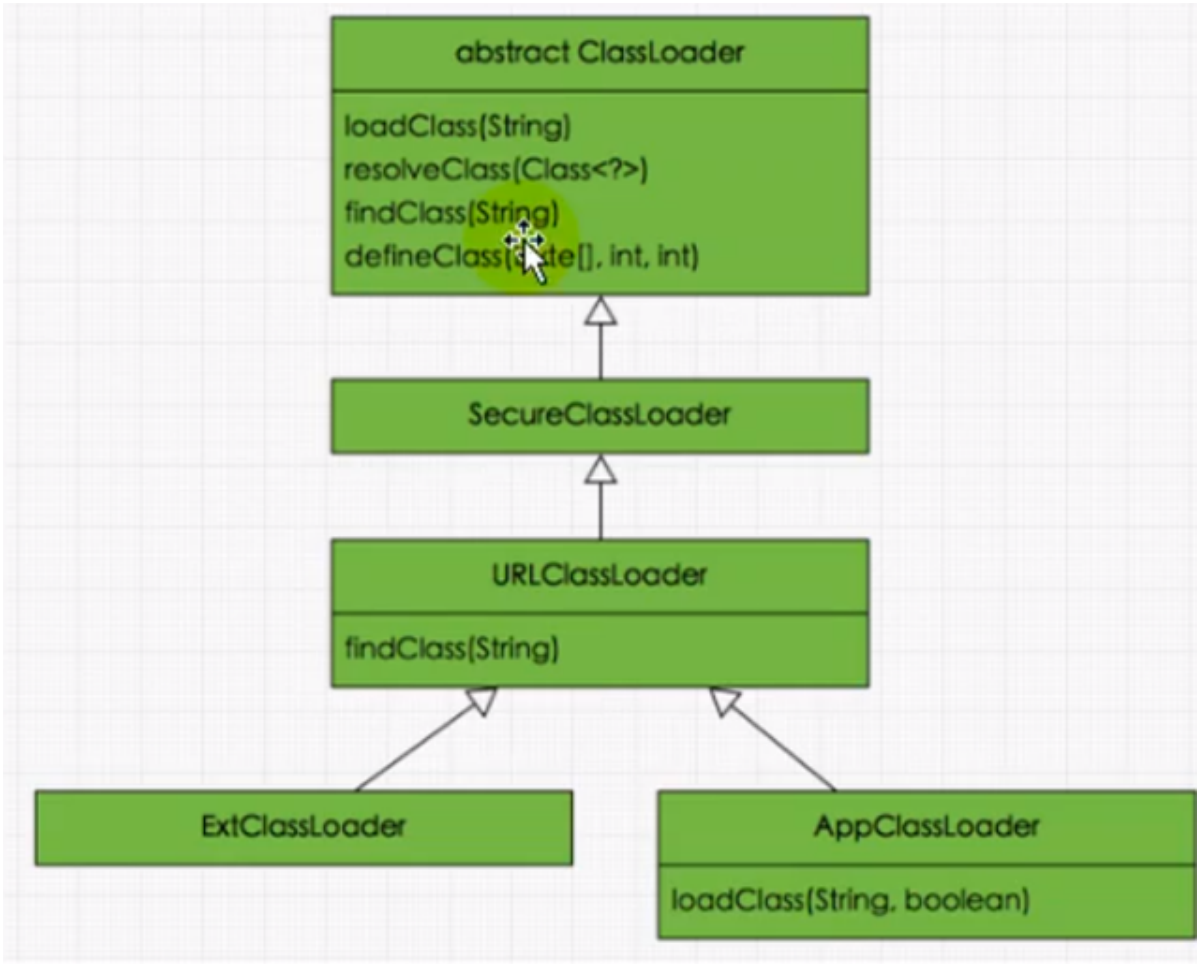
JVM支持两种类型的类加载器，分别为引导类加载器（Bootstrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）

从概念上讲，自定义类加载器一般指的使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。因此对于如下，Bootstrap ClassLoader属于引导类加载器，其他Extension ClassLoader和System ClassLoader都间接集成自抽象类ClassLoader类（他们都是Launcher类的内部类，两者属于并列关系），属于自定义类加载器。

注意，如下图表示最常见的类加载器，其四者之间的关系是包含关系，不是上层下层，也不是子父类的继承关系。



他们的继承关系如下图所示



对于这些加载器的获取相关代码示例如下所示（jdk8版本）

```
/**
 * @author banana
 * @create 2024-06-05 21:34
 */
public class ClassLoaderTest {
    public static void main(String[] args) {
```

```
        // 他们的关系如图目录 /A/B/C.text, 是一种包含关系，只不过可以通过getPatent方法获取
上层

        // 获取系统类加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(systemClassLoader); //
sun.misc.Launcher$AppClassLoader@18b4aac2

        // 获取系统类加载器上层：扩展类加载器
        ClassLoader extClassLoader = systemClassLoader.getParent();
        System.out.println(extClassLoader); //
sun.misc.Launcher$ExtClassLoader@6842775d

        // 获取扩展类加载器的上层(引导类加载器): 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：默认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(Java的核心类库)都是使用引导类加载器进行加载
        ClassLoader classLoader1 = String.class.getClassLoader();
        System.out.println(classLoader1);   // null
    }
}
```

注意：在Java平台的发展过程中，类加载器的实现确实发生了一些变化。在早期版本的JDK中，通常会使用`sun.misc.Launcher`作为主要的类加载器。但是，在较新的版本中，特别是从Java 9开始，引入了模块化系统，其中类加载器的组织和实现发生了变化。在Java 9及其之后的版本中，类加载器的实现被重新组织到了`jdk.internal.loader`包中。这一变化是为了更好地支持模块化系统，并提供更加可靠和安全的类加载机制。`sun.misc.Launcher`仍然存在，但它在这些版本中更多地扮演了一个过渡或支持的角色，而不是主要的类加载器实现。因此，如果你发现类加载器的实现变成了`jdk.internal.loader`，那么这很可能是因为你在使用Java 9或更新的版本。这种变化是为了使Java平台更加健壮和可维护，并更好地适应现代的开发需求。

```
package com.yjy.JVMTEST;

/**
 * @author banana
 * @create 2024-06-05 21:34
 */
public class ClassLoaderTest {
    public static void main(String[] args) {

        // 他们的关系如图目录 /A/B/C.text, 是一种包含关系，只不过可以通过getPatent方法获取
上层

        // 获取系统类加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
```

```
        System.out.println(systemClassLoader); //
jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc

        // 获取系统类加载器上层：扩展类加载器
        ClassLoader extClassLoader = systemClassLoader.getParent();
        System.out.println(extClassLoader); //
jdk.internal.loader.ClassLoaders$PlatformClassLoader@3567135c

        // 获取扩展类加载器的上层(引导类加载器)：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：默认使用系统类加载器进行加载
        ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
        System.out.println(classLoader);    //
jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc

        // String类(Java的核心类库)都是使用引导类加载器进行加载
        ClassLoader classLoader1 = String.class.getClassLoader();
        System.out.println(classLoader1);   // null
    }
}
```

3.2 引导类、扩展类、系统类加载器的使用以及演示

（1）引导类加载器（启动类加载器，Bootstrap ClassLoader）

- 该类加载器是使用C/C++语言实现的，嵌套在JVM内部
- 它用来加载Java的核心库（JAVA_HOME/jre/lib/rt.jar、resources.jar或sun.boot.class.path路径下的内容，都称为核心类），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
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器
- 出于安全考虑，Bootstrap启动类加载器只加载名为java、javax、sun等开头的类

获取BootstrapClassLoader能够加载的api路径代码示例：

```
import sun.misc.Launcher;

import java.net.URL;

/**
 * @author banana
 * @create 2024-06-05 22:50
 */
public class ClassLoaderTest1 {
    public static void main(String[] args) {

        // 获取BootstrapClassLoader能够加载的api路径
        URL[] urLs = Launcher.getBootstrapClassPath().getURLs();
        for (URL urL : urLs) {
            System.out.println(urL.toExternalForm());
```

```
        }
    }
}
```

结果：

```
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/resources.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/rt.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/sunrsasign.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/jsse.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/jce.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/charsets.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/lib/jfr.jar
file:/C:/Program%20Files/Java/jdk1.8.0_91/jre/classes

Disconnected from the target VM, address: '127.0.0.1:57420', transport: 'socket'

Process finished with exit code 0
```

我们从上面任意选一个jar包，然后将其解压后，获取其中的字节码文件(.class),然后通过类对象的getClassLoader（）方法去获取其类加载器，可以发现都是BootstrapClassLoader。

（2）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iava.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ire/lib/ext子日录(扩展目录)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。

获取扩展类加载器能够加载的api路径代码示例

```java
import sun.misc.Launcher;

import java.net.URL;

/**
 * @author banana
 * @create 2024-06-05 22:50
 */
public class ClassLoaderTest1 {
    public static void main(String[] args) {

        // 获取扩展类加载器能够加载的api路径代码示例
        String extDirs = System.getProperty("java.ext.dirs");
        for(String path : extDirs.split(";")) {
            System.out.println(path);
        }
```

```
    }
}
```

结果：

```
C:\Program Files\Java\jdk1.8.0_91\jre\lib\ext
C:\Windows\Sun\Java\lib\ext
```

我们从上面任意选一个jar包，然后将其解压后，获取其中的字节码文件(.class),然后通过类对象的getClassLoader（）方法去获取其类加载器，可以发现都是扩展类加载器

（3）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
    - 环境变量classpath

      使用1.5以上版本的JDK，完全可以不用设置ClassPath环境变量，即使不设置ClassPath环境变量，也完全可以正常编译和运行Java程序。

      ClassPath环境变量的作用：当使用"java Java类名"来运行Java程序的时候，我们的想法是JRE到当前路径下搜索Java类，但是1.4以前版本的JDK没有设计这个功能，这意味着即使当前路径（其执行java Java类名的当前目录）已经包含了HelloWorld.class，并在当前路径下执行"java HelloWorld"，系统一样提示找不到HelloWorld类。如果使用1.4以前版本的JDK，则需要在CLASSPATH环境变量中添加一点(.)，用以告诉JRE需要在当前路径下搜索Java类。

      编译和运行Java程序还需要JDK的lib路径下dt.jar和tools.jar文件中的java类，因此还需要把这两个文件添加到CLASSPATH环境变量里。

      因此，如果使用1.4以前版本的JDK来编译和运行Java程序，常常需要设置CLASSPAT环境变量的值为

      ```
      .;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar
      ```

      其中%JAVA_HOME%代表JDK的安装目录。
      后来的JRE会自动搜索当前路径下的类文件，而且使用Java的编译和运行工具时，系统可以自动加载dt.jar和tools.jar文件中的Java类，因此不再需要设置CLASSPATH环境变量。

      当然，即使使用1.5以上版本的JDK，也可以设置CLASSPATH环境变量，一旦设置了该环境变量，JRE将会按照该环境变量指定的路径来搜索Java类。这意味着如果CLASSPATH环境变量中不包括一点(.)，也就是没有包含当前路径，JRE不会在当前路径下搜索Java类。

      如果想在运行Java程序时临时指定JRE搜索Java类的路径，则可以使用-classpath选项，即按如下格式来运行Java命令：

      ```
      java -classpath dir1;dir2;dir3;...;dirN Java类
      ```

-classpath选项的值可以是一系列的路径，多个路径之间在windows平台上以分号(;)隔开，在Linux平台上则以冒号(:)隔开。
如果在运行Java程序时指定了-classpath选项的值，JRE将严格按-classpath选项所指定的路径来搜索Java类，既不会在当前路径下搜索Java类，CLASSPAH环境变量所制定的搜索路径也不再有效。如果想使CLASSPATH环境变量指定的搜索路径有效，而且还会在当前路径下搜索Java类，则可以按如下格式来运行Java程序：

```
java -classpath %CLASSPATH%;.;dir1;dir2;dir3;...;dirN Java类
```

上面命令通过%CLASSPATH%来引用CLASSPATH环境变量的值，并在-classpath选项的值里添加了一点，强制JRE在当前路径下搜索Java类。

- 系统属性java.class.path

在java命令行中，可以使用-classpath参数或简写形式-cp来设置系统属性，如`java -classpath /path/to/directory:/path/to/jar/file.jar MyClass`；在java代码中，可以使用`System.setProperty("java.class.path", "/path/to/directory:/path/to/jar/file.jar")`来动态设置，其设置的值是系统类加载器加载字节码class的路径。可以通过如下代码进行打印，其打印内容为系统启动时的加载所有class的路径（包括pom中的嵌套的jar，嵌套的JAR通常指的是在项目的依赖项中引入的JAR文件）

```
public class SystemProperty {
    private static final String JAVA_CLASS_PATH = "java.class.path";

    public static void main(String[] args) {
        String javaClassPath = "";
        Properties properties = System.getProperties();
        for(String property : properties.stringPropertyNames()) {
            System.out.println(property + " = " +
System.getProperty(property));
            if(StringUtils.equals(property, JAVA_CLASS_PATH)){
                javaClassPath = property;
            }
        }


 System.out.println("==========================================================
==================");

        //java.class.path
        String[] javaClassPaths =
System.getProperty(javaClassPath).split(";");
        System.out.println(JAVA_CLASS_PATH + ":");
        for (String classPath : javaClassPaths){
            System.out.println(classPath);
        }
    }
```

打印结果

```
~/jdk/jre/lib/*.jar(12个)
~/jdk/jre/lib/ext/*.jar(13个)
~/idea_rt.jar(1个)
~/.m2/*.jar(m个)
~/target/classes/*.jar(n个)
```

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

(4) 用户自定义类加载器

- 在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，来定制类的加载方式。
- 为什么要自定义类加载器
  - 隔离加载类

    在某些情况下，需要在同一个应用程序中加载不同版本的同一个类，或者加载来自不同源的类，这时就需要使用自定义类加载器来实现隔离加载。比如，在模块化框架中，可能需要加载不同模块中的类，而这些模块可能有不同的依赖关系或版本要求
  - 修改类加载的方式

    默认的类加载机制可能无法满足特定的加载需求，例如需要在类加载时进行特殊处理或添加额外的逻辑。自定义类加载器可以让开发者更灵活地控制类加载的方式，从而实现一些定制化的需求
  - 扩展加载源

    默认的类加载器通常只能从本地文件系统或网络中加载类，但有时可能需要从其他源加载类，比如数据库、内存中的数据结构等。自定义类加载器可以帮助扩展加载源，使得类可以从更多不同的地方加载。
  - 防止源码泄漏

    在某些应用场景中，可能需要保护代码的安全性，防止源码泄漏。通过自定义类加载器，可以对类加载过程进行控制，从而加密或混淆类文件，增强代码的安全性，避免源码泄漏的风险。
- 用户自定义加载器实现步骤
  - 开发人员可以通过继承抽象类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
  - 在JDK1.2之前，在自定义类加载器时，总会去继承classLoader类并重写loadClass()方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadclass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中，示例如下

```
/**
 * @author banana
 * @create 2024-06-06 0:28
 */
public class customClassLoader extends ClassLoader {
    @Override
    protected Class<?> findClass(String name) throws
ClassNotFoundException {
```

```
        try{
            byte[] result = getClassFromCustomPath(name);

            if(result == null) {
                throw new FileNotFoundException();
            } else {
                return defineClass(name, result, 0, result.length);
            }
        }catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        return super.findClass(name);
    }

    private byte[] getClassFromCustomPath(String name) {
        // 从自定义路径中加载指定类：细节略
        // 如果字节码文件加密，这里需要有解密逻辑
        return null;
    }

}
```

- 在编写自定义类加载器时，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()方法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。

4、关于ClassLoader

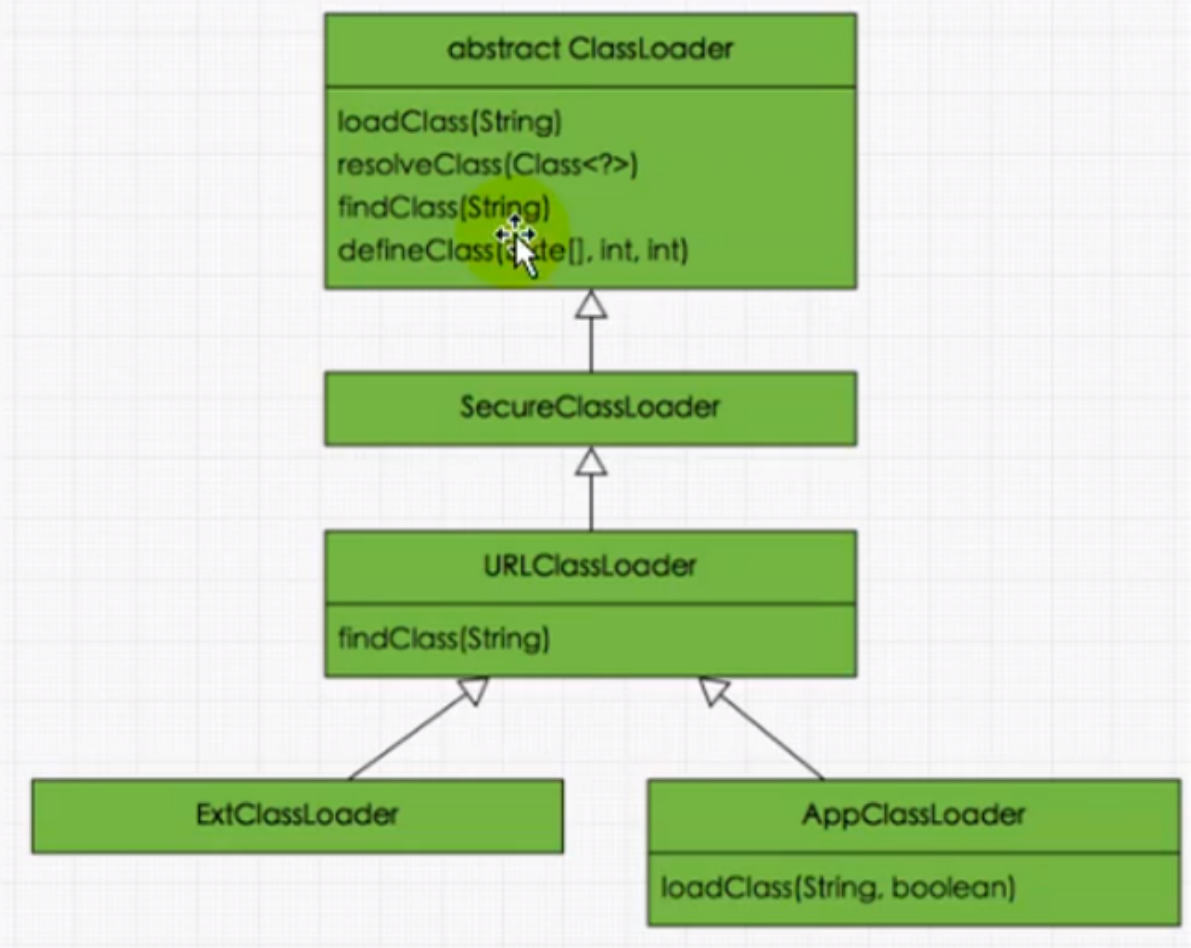
4.1 介绍

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)

4.2 方法介绍（都不是抽象方法）

方法名称	描述
getParent()	返回该类加载器的超类加载器
loadClass(String name)	加载名称为name的类，返回结果为java.lang.Class 类的实例
findClass(String name)	查找名称为name的类，返回结果为java.lang.Class 类的实例
findLoadedClass(String name)	查找名称为name的已经被加载过的类，返回结果为java.lang.Class类的实例
defineClass(String name,byte[]b.intoff.int len)	把字节数组b中的内容转换为一个Java类,返回结果为java.lang.Class 类的实例
resolveClass(Class<?>c)	连接指定的一个Java类

4.3 继承关系



sun.misc.Launcher是一个java虚拟机的入口应用

4.4 获取ClassLoader的方法

方式一：获取当前类的ClassLoader
clazz.getClassLoader()
方式二：获取当前线程上下文的ClassLoader
Thread.currentThread().getContextClassLoader()
方式三：获取系统的ClassLoader
ClassLoader.getSystemClassLoader()
方式四：获取调用者的ClassLoader
DriverManager.getCallerClassLoader()

5、双亲委派机制

5.1 说明

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它的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模式，即把请求交由父类处理它是一种任务委派模式。

5.2 示例说明

1. 示例一

创建一个java.lang包，并在其中定义一个String类，内容如下所示

```
package java.lang;

/**
 * @author banana
 * @create 2024-06-11 22:42
 */
public class String {

    // 声明一个静态方法，用于判断该类是否被加载
    // 因为类加载的时候会调用其静态方法
    static {
        System.out.println("我是自定义的String类的静态代码块");
    }

}
```

创建StringTest，进行测试，测试内容如下所示

```
package com.yjy.JVMTEST;

import java.lang.String;
/**
 * @author banana
 * @create 2024-06-11 22:45
 */
public class StringTest {

    public static void main(String[] args) {

        // 调用java.lang.String的默认构造器，创建java.lang.String的实例
        java.lang.String str = new java.lang.String();

    }
}
```

运行结果（并没有调用自定义String类的静态方法）

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:63479', transport: 'socket'
Disconnected from the target VM, address: '127.0.0.1:63479', transport: 'socket'

Process finished with exit code 0
```

原因是在main方法中，调用new java.lang.String()方法时，类加载器回去加载java.lang.String,然而根据双亲委派机制，首先回去引导类加载器中寻找该类，Bootstrap启动类加载器只加载名为java、javax、sun等开头的类，因此由引导类加载器完成java.lang.String类的加载。我们可以通过getClassLoader方法验证一下

```
 // 调用java.lang.String的默认构造器，创建java.lang.String的实例
java.lang.String str = new java.lang.String();
System.out.println(str.getClass().getClassLoader());
```

结果如下所示为null，因为引导类加载器是通过C/C+=实现。

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:63694', transport: 'socket'
null
Disconnected from the target VM, address: '127.0.0.1:63694', transport: 'socket'

Process finished with exit code 0
```

同样我们可以通过如下命令，看一下StringTest的类加载器

```
StringTest stringiTest = new StringiTest();
System.out.println(stringiTest.getClass().getClassLoader());
```

其结果是系统类加载器（我们自定义的类都是通过系统类加载器去进行加载的）

```
sun.misc.Launcher$AppClassLoader@18b4aac2
```

2. 示例二

在示例一的基础之上，我们在自定义的String类型中，通过main方法区打印一段话

```
package java.lang;

/**
 * @author banana
 * @create 2024-06-11 22:42
 */
public class String {

    // 声明一个静态方法，用于判断该类是否被加载
    // 因为类加载的时候会调用其静态方法
    static {
        System.out.println("我是自定义的String类的静态代码块");
    }


    public static void main(String[] args) {
        System.out.println("hello world");
    }

}
```

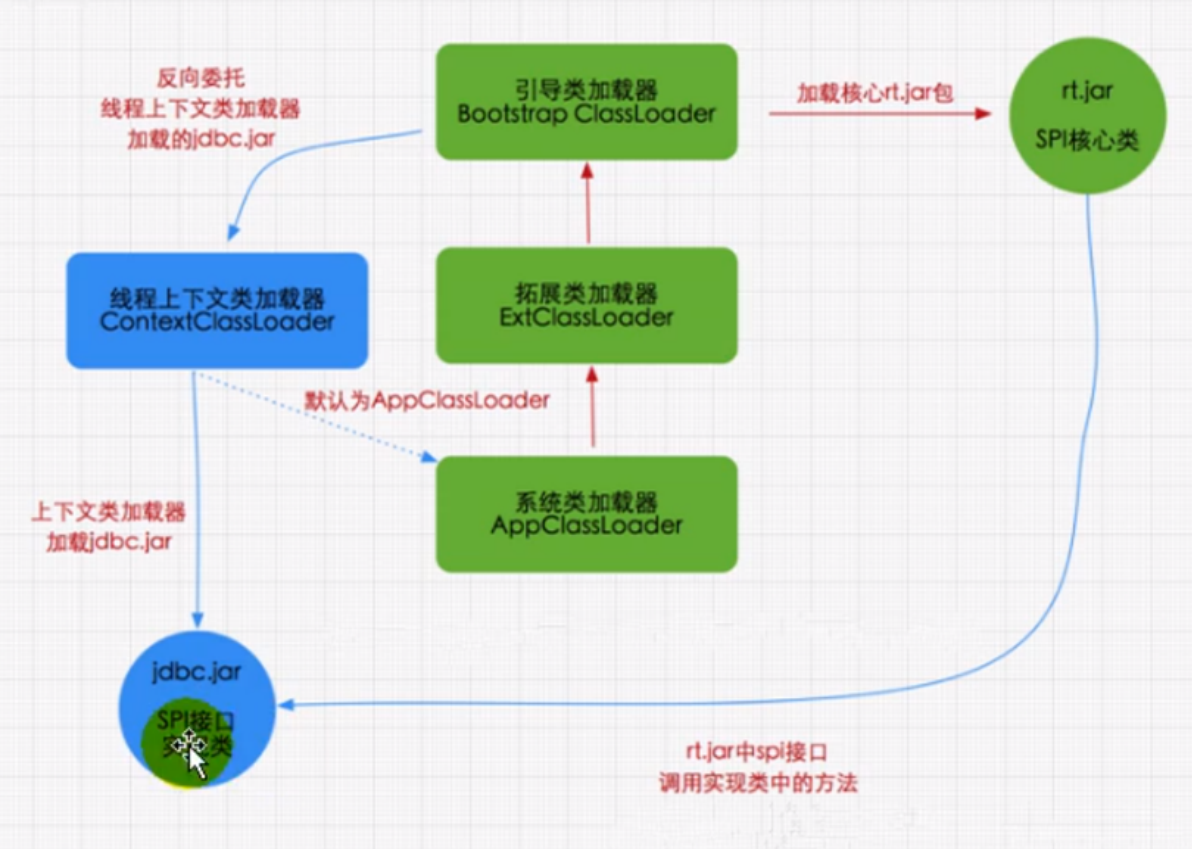
结果

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:64108', transport: 'socket'
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
   public static void main(String[] args)
否则 JavaFX 应用程序类必须扩展javafx.application.Application
Disconnected from the target VM, address: '127.0.0.1:64108', transport: 'socket'

Process finished with exit code 1
```

其原因很简单，是因为java.lang.String类是通过引导类加载器进行加载的的，其加载的使java核心中的那个java.lang.String类，这个类中没有main方法，所以报该错误。

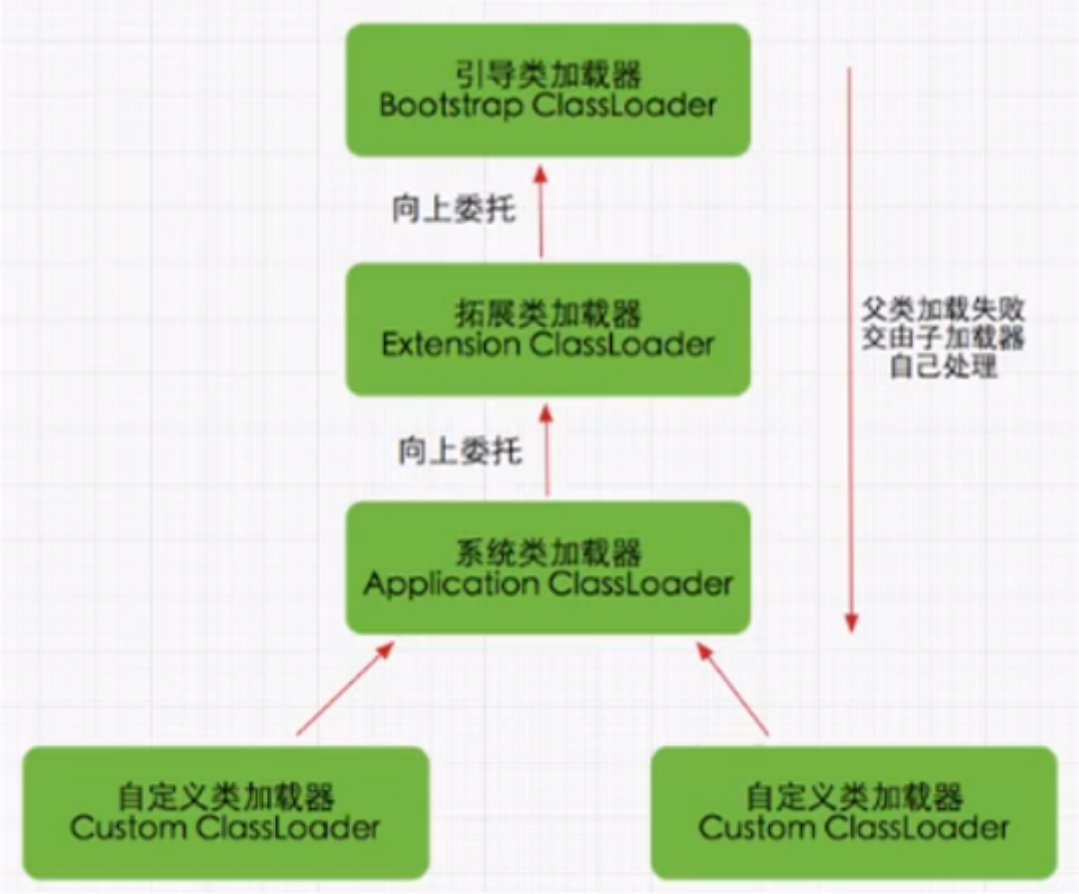
3. 示例三



程序中用到了SPI接口，SPI接口属于核心API，使用双亲委派机制进行加载，首先是通过引导类加载器去加载SPI核心jar包rt.jar,这里面会存在一些接口（interface），其具体实现类由第三方实现，例如我们要加载第三方的jdbc.jar包，其不属于系统的核心api，应该由系统类加载器去进行加载，引导类加载器，反向委派（即向下传递），一直到系统类加载器，其是通过线程的ContextClassLoader（即系统类加载器），来加载SPI接口的具体实现类jdbc.jar。即引导类加载器加载核心jar包，系统类加载器加载第三方的jar包api等。

5.3 双亲委派工作原理

- 如果一个类加载器收到了类加载请求，它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
- 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启动类加载器
- 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。



5.4 双亲委派机制的优势

- 避免类的重复加载
- 保护线程安全，防止核心API被随意篡改
    - 自定义类：java.lang.String (对现在的程序代码造成了破坏)

> 具体危害：
>
> **安全风险**
>
> - **恶意代码注入**：攻击者可以创建恶意版本的核心类（如`java.lang.String`），并将其注入到应用程序中。这种恶意版本的类可以包含后门、篡改数据或者窃取敏感信息。
> - **破坏系统完整性**：自定义的核心类可以绕过Java提供的安全机制和沙箱模型，进行不当的文件访问、网络连接等操作。
>
> **稳定性和一致性问题**
>
> - **类冲突**：不同加载器加载同名但内容不同的类会导致类冲突和不可预测的行为。例如，不同模块或库依赖于各自版本的核心类，可能导致方法不匹配或抛出异常。
> - **调试困难**：当问题发生时，很难排查和调试，因为多个版本的核心类共存会使错误信息混乱，难以确定问题的来源。
>
> **维护性问题**
>
> - **版本管理复杂化**：Java核心类库由Oracle（现在是甲骨文公司）或OpenJDK项目维护和更新，如果允许自定义核心类，开发者需要自己管理这些类的版本和更新，增加了维护难度。
> - **API契约破坏**：自定义核心类可能不遵循官方API的契约和规范，导致应用程序行为不一致，甚至在不同环境中出现不同的结果。

> **影响Java生态系统**
>
> - **标准库失去统一性**：Java生态系统依赖于一套统一的标准库，如果每个应用都可以自定义核心类，整个生态系统的兼容性和互操作性将受到极大挑战。
> - **开发成本增加**：开发者需要花费更多时间和精力来确保自定义核心类和标准库之间的兼容性，这无疑增加了开发成本。
>
> 实际例子
>
> 举个具体的例子，如果允许自定义`java.lang.String`类，一个恶意版本的`String`类可能会修改常见的字符串操作方法（如`equals`、`hashCode`、`substring`等），导致：
>
> - 安全漏洞：例如，通过在`equals`方法中植入恶意代码，可以在比较字符串时执行未授权操作。
> - 数据篡改：在对字符串进行操作时，无形中改变数据内容，导致数据完整性被破坏。
> - 程序崩溃：通过引入恶意逻辑，使得在处理字符串时抛出异常，从而导致程序崩溃。

- 自定义类：java.lang.guagua(核心java.lang下不存在guagua，但是运行的时候也会报错，报错内内容：java.lang.SecurityException:Prohibited package name: java.lang,可以理解为核心的java.lang包访问需要权限，不能在该包名下定义自己的类，如果允许，可能会对引导类加载器造成影响，将其整挂掉)

> 危害：
>
> **安全风险**：恶意的`java.lang.guagua`类可能包含恶意代码，用于执行未授权操作、窃取敏感信息或者破坏系统完整性。
>
> **类冲突**：自定义的`java.lang.guagua`类可能与其他模块或库中的类发生冲突，导致不可预测的行为。
>
> **稳定性问题**：自定义的`java.lang.guagua`类可能破坏Java核心类库的API契约，导致应用程序行为不一致。
>
> **影响Java生态系统**：自定义的`java.lang.guagua`类可能影响整个Java生态系统的兼容性和一致性，增加开发成本和维护难度。

5.5 安全沙箱机制

如5.2中的示例二所示

自定义string类，但是在加载自定义string类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载idk自带的文件(rt.jar包中java\lang\string.class)，报错信息说没有main方法，就是因为加载的是rt.jar包中的string类。这样可以保证对java核心源代码的保护，这就是沙箱安全机制。

6、其他

6.1 在JVM中表示两个Class对象是否为同一个类存在的两个必要条件

- 类的完整名称必须一致，包括包名
- 加载这个类的类加载器（ClassLoader，指ClassLoader实例对象）必须相同

  换句话说，在JVM中，即使这两个类对象(class对象)来源同一个Class文件，被同一个虚拟机所加载，但只要加载它们的ClassLoader实例对象不同，那么这两个类对象也是不相等的。

6.2 对类加载器的引用

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需要保证这两个类型的类加载器是相同的。

6.3 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

主动使用，又分为七种情况:

- 创建类的实例
- 访问某个类或接口的静态变量，或者对该静态变量赋值
- 调用类的静态方法
- 反射(比如:Class.forName(“xxx"))
- 初始化一个类的子类
- Java虚拟机启动时被标明为启动类的类
- 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。

其实就是看是否会调用的类的初始化。