视频： <https://www.bilibili.com/video/av47756459>

资料： 深入理解java虚拟机

# 工具使用

jconsole

jvisualvm

# 类加载

## 加载过程

在java代码中，类型的加载 连接 初始化过程都是在程序运行期间完成的。（加载的是类的class文件，并且是程序运行期间加载，说明只有在被调用后，才会去加载这个类文件）。

完整过程

1. 加载

把二进制形式的java类型读入到java虚拟机中

2.连接

2.1 验证

验证字节码文件的正确性

2.2 准备

为类变量分配内存，设置默认值，但是在到达初始化之前，类变量都没有初始化为真证的初始化值。

2.3 解析

解析过程就是在类型的常量池中寻找类，接口，字段，和方法的符号引用，把这些符号引用替换为直接引用的过程。

3.初始化

为类变量赋值。

4.类实例化

为新对象分配内存

为实例变量赋予默认值

为实例变量赋予正确的初始值

java编译器为她编译的每一个类都至少生成了一个实例化初始方法，在java的class文件中，这个实例初始化方法被称为<init>,针对源代码中的每一个类的构造方法，java编译器都产生一个<init>方法。

5.垃圾回收与对象终结

## 类的加载 连接 初始化 使用 卸载

**1.**

加载：

查找并加载类的二进制数据。

连接：

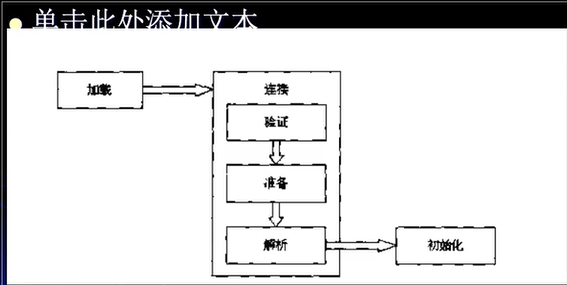
验证：确保被加载的类的正确性

准备：为类的静态变量分配内存，并将其初始化为默认值

解析：把类中的符号引用转变为直接引用

初始化：

把类的静态变量赋予正确的初始值。



**2.**

java程序对类的使用方式可以分为两种

主动使用

被动使用

所有的java虚拟机实现必须在每个类或者接口被java程序 **‘首次主动使用’** 时才**初始化**他java们。

主动使用：

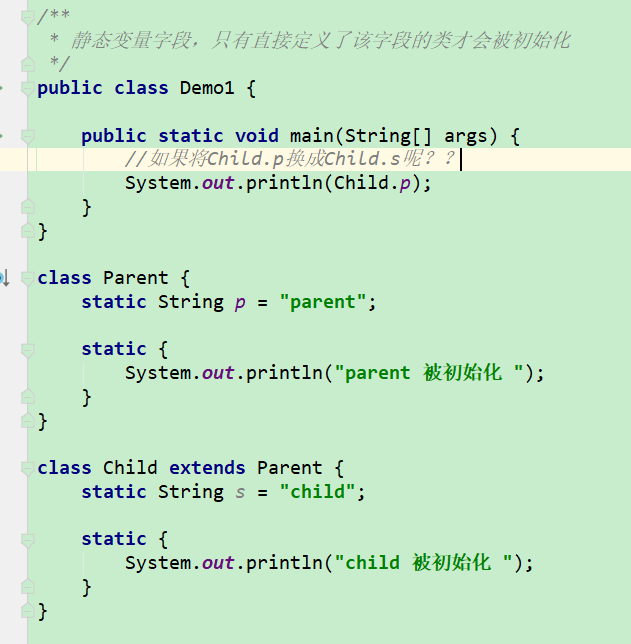
1. **创建类的实例**
2. **访问某个类或者接口的静态变量，或者对该静态变量赋值。**
3. **调用类的静态方法。**
4. **反射（Class.forName（“”））**
5. **初始化类的子类**
6. **java虚拟机启动时被标明为启动类的类**

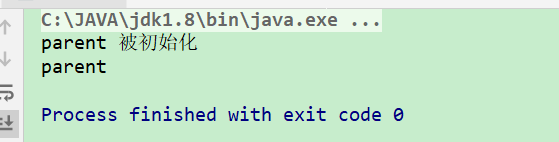
**案例：**

**static代码块会在类被初始化的时候去执行，通过这个代码块的打印情况，判断类的初始化情况。**

**案例1：**

**对于静态字段，在继承体系中只有直接定义了该字段的类才会被初始化。**





**思考：**

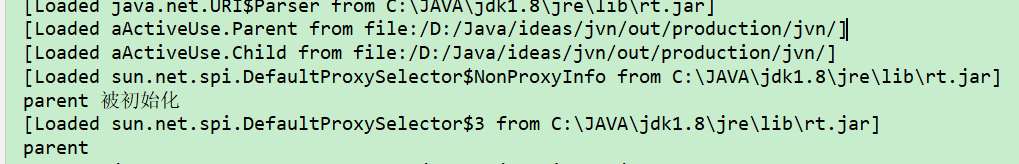
**1. 如果换成Child.s输出结果是什么？**

**父类和子类都会被初始化，使用了静态字段，是对类的主动使用，所以Child类会被初始化，Parent类是Child父类，子类的初始化会将父类进行初始化，所有都会有输出。**

**2.如果执行Child.p，Child不会被初始化，是否会被加载？**

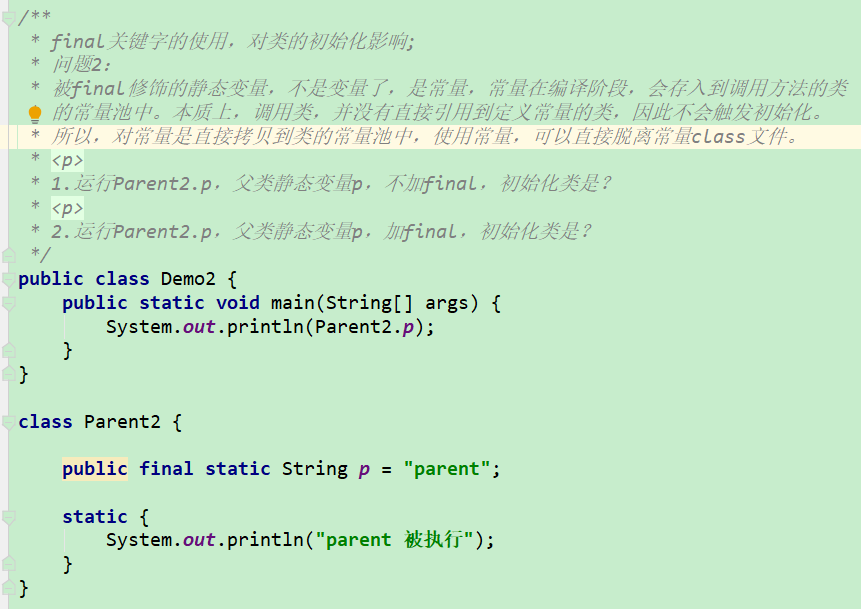
**会，通过虚拟机参数查看，加载的类。**

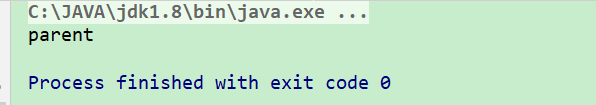
*-XX：+TraceClassLoading 用于追踪类的加载信息打印*



**案例2：**

**对于常量，使用常量，不会导致类初始化**

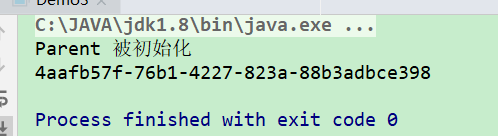




**案例3：**

**对于编译期间确定的常量值，类不需要初始化，对于运行期间的常量值，需要初始化。**





**案例4：**

**数组本质是jvm在运行期间动态生成的类型，不会引发类的初始化。**



**案例5：**

**静态变量中，初始化值，被覆盖问题**

|  |
| --- |
| */\*\*  \* 1. b放在位置1，结果是什么？  \* <p>  \* 2. b放在位置2，结果是什么？  \*/* **public class** Demo6 {   **public static void** main(String[] args) {  Singleton s = Singleton.*getInstance*();  System.***out***.println(Singleton.*a*);  System.***out***.println(Singleton.*b*);   } }  */\*\*  \* 初始化顺序，对于静态变量，从上到下的顺序进行初始化  \*/* **class** Singleton {   *//连接-准备： 为静态变量分配内存，并默认值。a=0，instance=null，b=0；   //初始化，按照从上向下的顺序。    //a没有赋值，默认为0* **public static int** *a*;   *//位置1  // public static int b = 0;   //给静态变量初始化，则a=1，b=1;* **private static** Singleton *instance* = **new** Singleton();   **private** Singleton() {  *a*++;  *b*++;  }   *//给变量b，初始化，b=0覆盖b1；  //位置2* **public static int** *b* = 0;   **public static** Singleton getInstance() {  **return** *instance*;  } } |

被动使用：

除了主动使用的情况外，其余的情况是类的被动使用，都不会导致类的初始化，但类可能会被加载。

**3.**

**类的加载：**

类的加载是指将类的class文件中的二进制数据读入到内存中，将其放在运行时的数据区的方法区内，然后在内存中创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

（读取class到内存中，放在方法区中，创建一个Class对象）

**方式：**

本地直接加载

网络下载.class

从zip，jar中加载

将java源文件动态编译为.class文件。（动态代理，或者jsp文件）

## 2.3 类的加载

* 类的加载的最终产品是位于内存中的Class对象。
* Class对象封装了类在方法区的数据结构，并且向java程序员提供了访问方法区内的数据结构的接口。

1. 有两种类型的类加载器：

java虚拟机自带的加载器，

根类加载器（BootStrap）

扩展类加载器（Extension）

系统（应用）类加载器 （System）

用户自定义的类加载器

java.lang.ClassLoader的子类。

用户可以定制类的加载方法。

2.类加载器并不需要等到某个类被“首次主动使用”时再加载它。

jvm规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）。

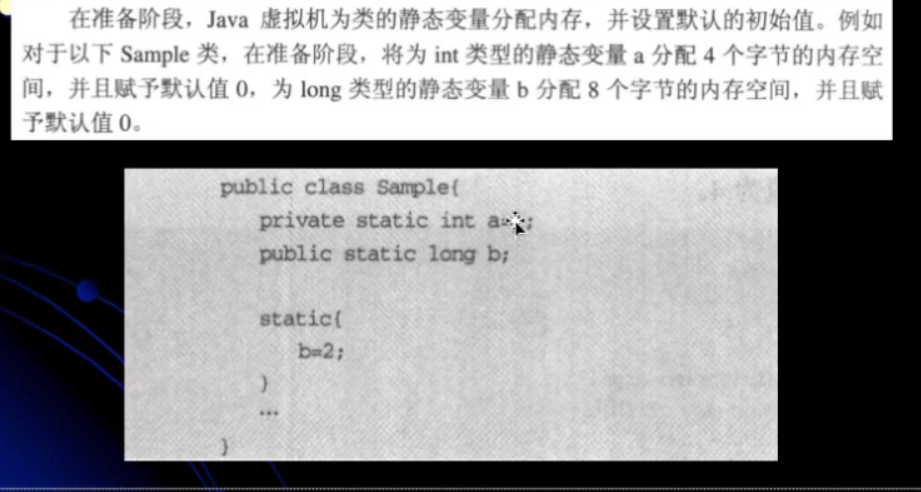
如果这个类，一直没有被程序主动使用，那么这个类加载器就不会报告错误。

## 2.4 类的验证

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

## 2.5 类的准备

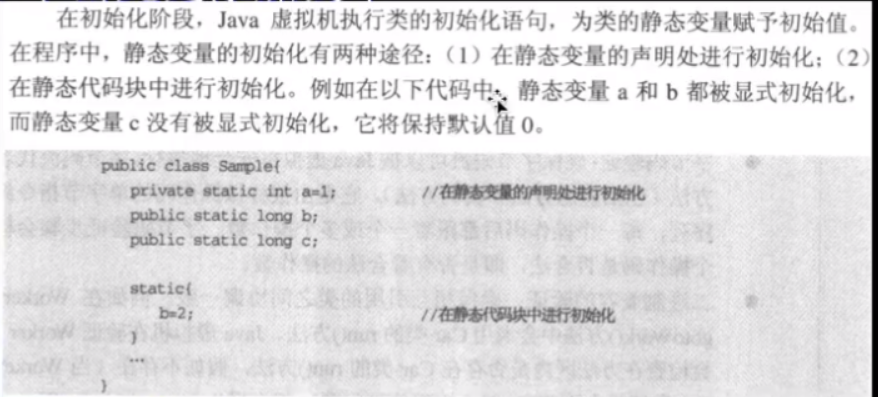
在准备阶段，java虚拟机为类的静态变量分配内存，并设置默认的初始值。

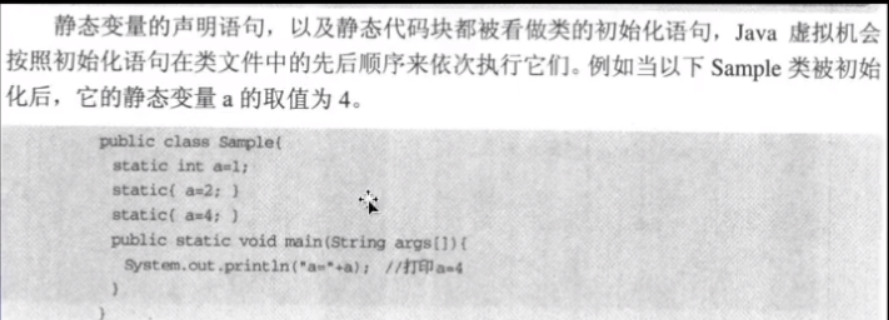


## 2.6 类的初始化

在初始化阶段，java虚拟机执行类的初始化语句，为类的静态变量赋予初始值。

<https://github.com/mashenghao/jvn/tree/master/src/aActiveUse>



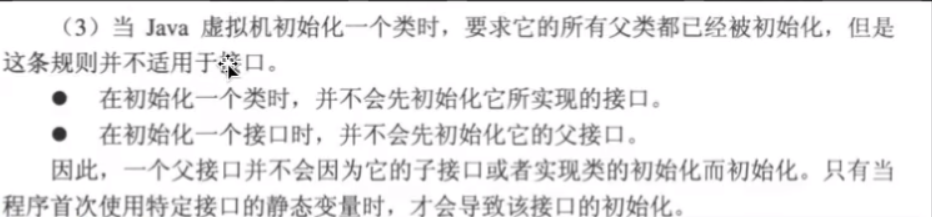


初始化步骤：

如果一个要主动使用，这个类将进行初始化。

* 加入这个类还没有被加载和连接，那就先进行加载和连接，
* 如果这个类存在直接父类，并且这个父类还没有被初始化，那就先初始化直接父类。
* 如果类中存在初始化语句，那就一次执行这些初始化语句。

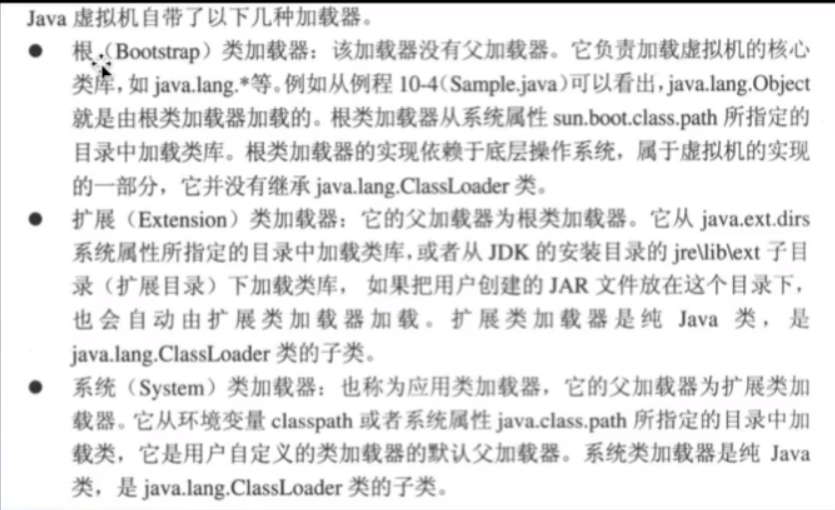
注意：



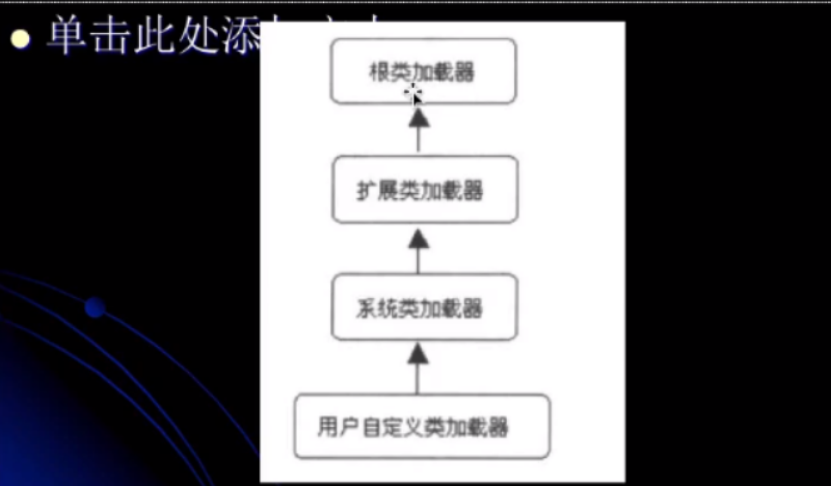
# 3. 类加载器

## 3.1 描述

类加载器是用来把类加载到java虚拟机中的，从1.2开始，类的加载过程采用父亲委托机制，这种机制能更好的保证java平台的安全。在此委托机制中，除了java虚拟机自带的根加载器外，其余的加载器都有且只有一个父加载器。当java程序请求加载器loader1加载Sample类时，loader1首先委托自己的父加载器去加载Sample类，若父加载器能加载，则由父加载器完成加载任务，否则才由加载器loader1本身加载Sample类。

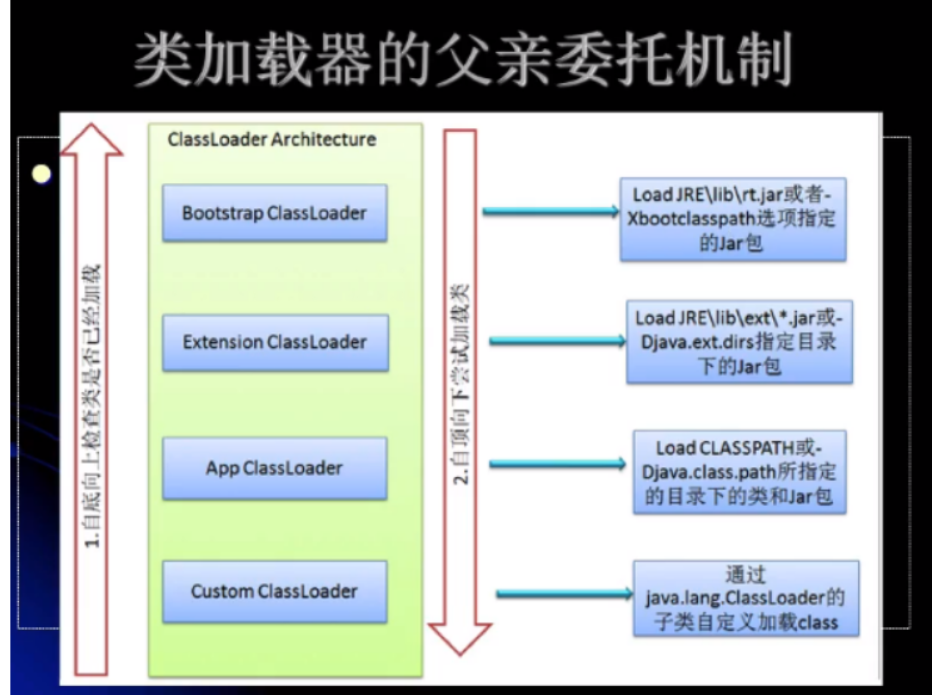


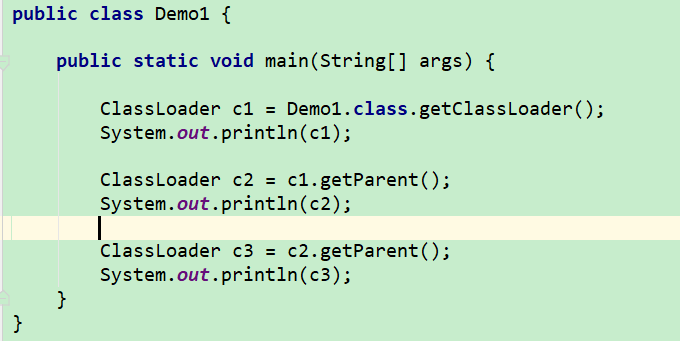
## 3.2 双亲委托机制

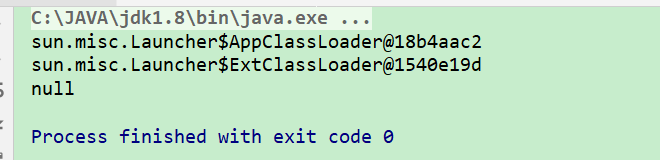


双亲委托机制：

如果一个类需要加载器去加载，如果加载器有父类，先让父类去加载，如此向上追溯，直到根类加载器，然后根类加载器尝试去加载，加载成功则结束，加载失败，又往下，一层层的尝试去加载，最后没有加载成功，则报错（classnotFound）。







java中有三个类加载器，如何区分那个类该由那个类加载器来完成，java采用了委托模型机制，这个机制简单来讲，就是“类装载器有载入需求时，会先请求其Parent使用其搜索路径帮忙载入，如果Parent找不到，那么才由自己的搜索路径搜索类”。

所以，对于一个类的加载，会先让父类去加载。图中可知Demo1的类加载器是AppClassLoader加载器加载的，AppClassloader的Parent加载器是ExtClassLoader，但ExtClassLoader的Parent为null，按照正常应该是Bootstrap loader。 原因是BootStrap Loader是由c++而写的，依据java的观点来看，逻辑上并不存在Bootstrap Loader的类实体。所以在java程序代码里打印就会输出null。

好处：

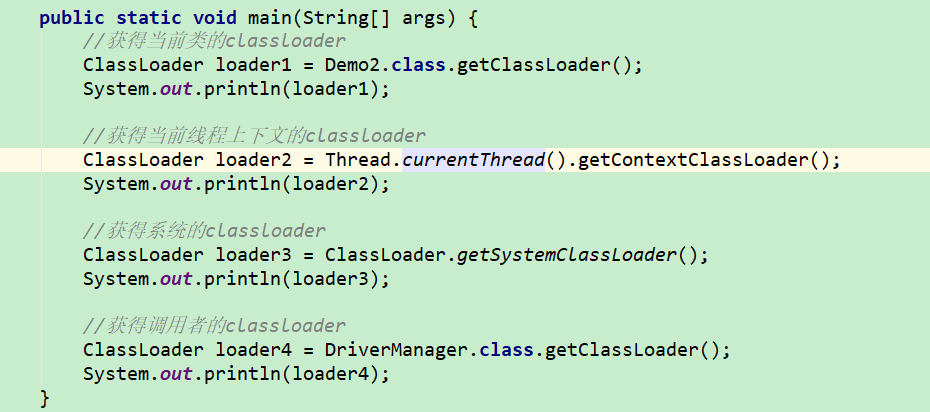
1.可以确保java核心库的类型安全，比如使用java.lang.Object这个类，所有的类都会使用，如果这个类由自定义的类加载器完成，在jvm中就有可能存在不同命名空间内的Object，这些类之间还是不兼容的，相互不可见的。借助于双亲委托机制，java的核心类库都有核心类库加载完成，这样使jvm中只有一个版本的核心类库。

2.确保java核心类库不会被自定义的类替代。

3.不同的类加载器可以为相同名称的类创建额外的命名空间。相同名称的类可以并存在jvm中，只需要用不同的类加载器去加载他们即可，不同的类加载器加载的类之间是不兼容的，这就相当于在jvm内部创建了一个有一个java类空间，框架中使用多。

## 3.3 不同加载类加载器作用与加载动作分析

1.获取加载器的途径



2. 数组类型和原始数据类型加载器

数组类型的Class对象不是由类加载器创建的，而是jvm在运行时创建的，数组的类加载器是数组中元素的类加载器。对于原始数据类型，没有类加载器。

## 3.4 自定义实现类加载器

<https://github.com/mashenghao/jvn/blob/master/src/bClassLoader/Demo4.java>

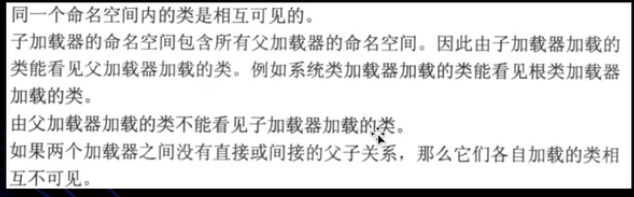
|  |
| --- |
| **package** bClassLoader;  **import** java.io.\*;  */\*\*  \** ***@program:*** *jvn  \** ***@Date:*** *2019/7/9 10:09  \** ***@Author:*** *mahao  \** ***@Description:自定义实现类加载器，自定义的加载器去加载class文件下的文件*** *\*/* **public class** MyClassloader **extends** ClassLoader {   **private** String **classLoaderName**;  **private** String **fileExtention** = **".class"**;   **public** MyClassloader(ClassLoader parent, String classLoaderName) {  **super**(parent);  **this**.**classLoaderName** = classLoaderName;  }   **public** MyClassloader(String classLoaderName) {  **super**(ClassLoader.*getSystemClassLoader*());  **this**.**classLoaderName** = classLoaderName;  }    @Override  **protected** Class<?> findClass(String name) **throws** ClassNotFoundException {  **byte**[] bytes = loadClassData(name);  **return** defineClass(name, bytes, 0, bytes.**length**);  }   **private byte**[] loadClassData(String name) {  InputStream in = **null**;  **try** {  in = **new** FileInputStream(**"out\\production\\jvn\\"** + **new** File(name.replace(**"."**, **"\\"**)+**fileExtention**));   ByteArrayOutputStream out = **new** ByteArrayOutputStream();  **int** n = 0;  **while** ((n = in.read()) != -1) {  out.write(n);  }  **byte**[] buf = out.toByteArray();  **return** buf;  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  in.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return null**;  }    @Override  **public** String toString() {  **return "MyClassloader{"** +  **"classLoaderName='"** + **classLoaderName** + **'\''** +  **", fileExtention='"** + **fileExtention** + **'\''** +  **'}'**;  }   **public static void** main(String[] args) **throws** ClassNotFoundException {  ClassLoader classLoader = **new** MyClassloader(ClassLoader.*getSystemClassLoader*().getParent(), **"myClassloader"**);  Class<?> aClass = classLoader.loadClass(**"aActiveUse.Singleton"**);  System.***out***.println(aClass);  } } |

## 3.5 命名空间

* 每个类加载器都有自己的命名空间，命名空间由该类加载器及所有父加载器所加载的类组成。
* 在同一个命名空间中，不会出现两个完整名字（包括类的包名）相同的类
* 在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的类。
* 重要说明：

1.子加载器所加载的类能够访问父加载器所加载的类

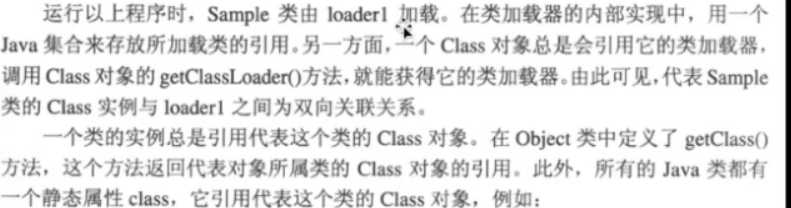
2.父加载器所加载的类不能访问子加载器所加载的类。



<https://github.com/mashenghao/jvn/blob/master/src/bClassLoader/Demo6.java>

## 3.6类的卸载

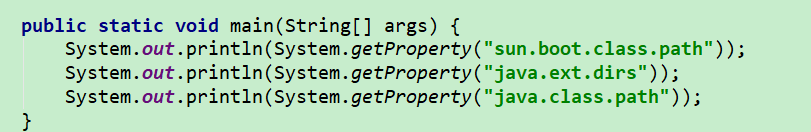
* 当sample类被加载、连接、初始化后，他的生命周期开始。当代变Sample类的class对象不再被引用，即不可触及时，Class对象就会结束生命周期，Sample类在方法区的数据也会被卸载，从而结束Sample类的声明周期。
* 一个类何时结束生命周期，取决于代表他的class对象何时结束周期。
* 由java虚拟机自带的类加载器加载的类，在虚拟机的声明周期中，始终不会被卸载，Java虚拟机自带的类包括根类加载器，扩展类加载器，系统加载类。Java虚拟机本身会始终引用这些类加载器，而这些类加载器始终会引用他们所加载的类的Class对象，因而这些Class对象始终是可以触及的。
* 由用户自定义的类加载器所加载的类是可以被卸载的。



* 查看类的卸载，将class对象，置为空的引用。

|  |
| --- |
| * *\*\*  \* 使用自定义的类加载器实现类的卸载  \*/* **public class** Demo5 {   */\*\*  \* 当类加载器和Class对象，都没有引用指向时，就会对类进行卸载，jvm自带的类加载器，由于始终，  \* 会由java虚拟机引用，这些加载器又会始终引用他们加载的class对象。但是，自定义的类加载则可能都会有  \* 指向为空的情况，所以是可以卸载的。  \*  \** ***@param args*** *\** ***@throws*** *Exception  \*/* **public static void** main(String[] args) **throws** Exception {  ClassLoader classLoader1 = **new** MyClassloader(ClassLoader.*getSystemClassLoader*().getParent(), **"myClassloader"**);  Class<?> aClass = classLoader1.loadClass(**"aActiveUse.Singleton"**);  System.***out***.println(aClass.hashCode());  */\*aClass = null; 方法一  classLoader1 = null;\*/   /\*方式二\*/* classLoader1 = ClassLoader.*getSystemClassLoader*();  aClass = classLoader1.loadClass(**"aActiveUse.Singleton"**);  System.*gc*();   } } |

## 3.7 类加载器各自加载的文件目录



## 3.8 不同命名空间内的类实例，是不相等的

*/\*\*  
 \* public void setUser(Demo8User demo8User) {  
 \* this.user = demo8User;  
 \* System.out.println("111111111111111");  
 \* }  
 \*  
 \* public void setUser(Object obj) {  
 \* this.user = (Demo8User) obj;  
 \* System.out.println("222222222222");  
 \*  
 \* }  
 \*  
 \* 在加载的类中，设置了两个方法，测试前提是clazz1 和 clazz2 有不同的命名空间。  
 \* 1.当获取方法setUser(Object obj)，为user赋值Object对象，当参数是obj2，是另一个空间的对象实例，则会发生错误  
 \* Caused by: java.lang.ClassCastException: bean.Demo8User cannot be cast to bean.Demo8User，可以看到是  
 \* 实例名相同，但由于类加载器不同，导致实例无法相互转换。如果传入参数是obj1则可以正常运行，因为是同一命名空间的  
 \* 同一类型。  
 \* <p>  
 \* 2.当获取setUser(Demo8User obj)时，如果传入的参数是Method method = clazz1.getMethod("setUser", Demo8User.class);，  
 \* 则会出现错误Caused by: java.lang.ClassNotFoundException: bean.Demo8User，因为这里的Demo8User.class，是loader2加载的类，  
 \* 所以，找不到clazz2对象，要想获取方法，需要制定参数的class为clazz1，即loader1加载的class对象。  
 \* <p>  
 \* 3.保证2的加载正确，测试method.invoke(obj1, obj？);当放入obj1和obj2之间会有什么不同，  
 \* obj1,显然执行正常，方法时clazz1对象的，参数也是clazz1的实例所以执行正常  
 \* obj2，执行错误，会报非法参数，java.lang.IllegalArgumentException: argument type mismatch，因为clazz1对象  
 \* 的方法接受的是clazz1对象实例，而传入的是clazz2的对象实例。  
 \*  
 \*/*

<https://github.com/mashenghao/jvn/blob/master/src/bClassLoader/Demo8.java>

## 3.9平台特定的启动类加载器

启动类加载器是由c++而写的，它内建与jvm中，会加载java.lang.classLoader以及其他平台类，当jvm启动时，一块特殊的机器码会运行，他会加载扩展类加载器和系统类加载器，这块机器码就是启动类加载器。

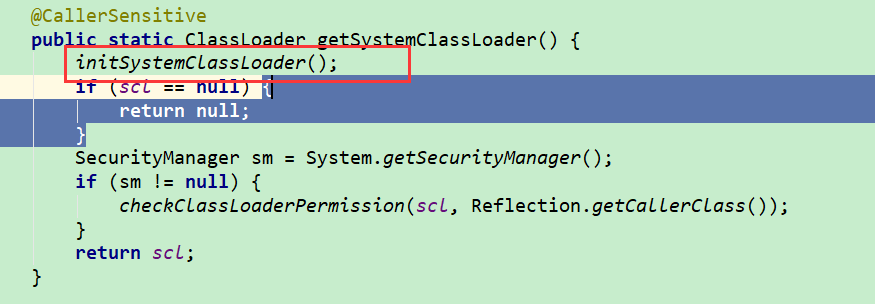
启动类加载器不是java类，其他的加载器都是java类，启动类加载器特定于平台的及其指令，负责开启整个加载过程。

启动类加载器加载了其他加载器和一些java核心库，加载第一个纯java加载器类就是启动器类加载器的作用。

## 3.10 Lanucher和ClassLoader源码分析

* Lanucher源码查看，openjdk或者反编译都可以
* 通过ClassLoader.getSystemClassLoader()方法读懂SystemClassLoader 和 ExtClassLoader的创建
* 自定义实现系统默认的加载器，即参数（java..system.class.loader）如何起作用。

1. 获取ClassLoader

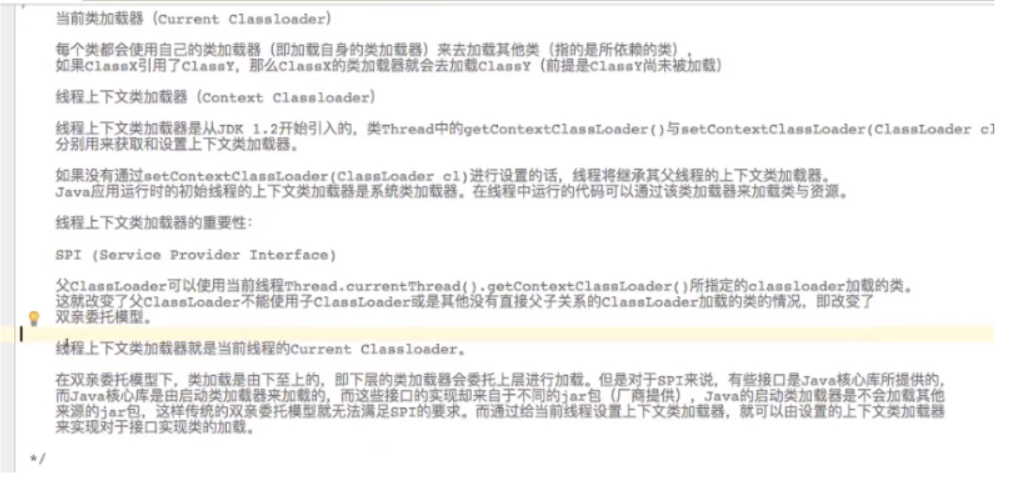


2.初始化SystemClassloader和ExtClassLoader

分析地址：

<https://blog.csdn.net/mahao25/article/details/95376891>

## 3.11 线程上下文类加载器分析与实现





# 附件：

## 1. 助记符：

getstatic 获取静态值

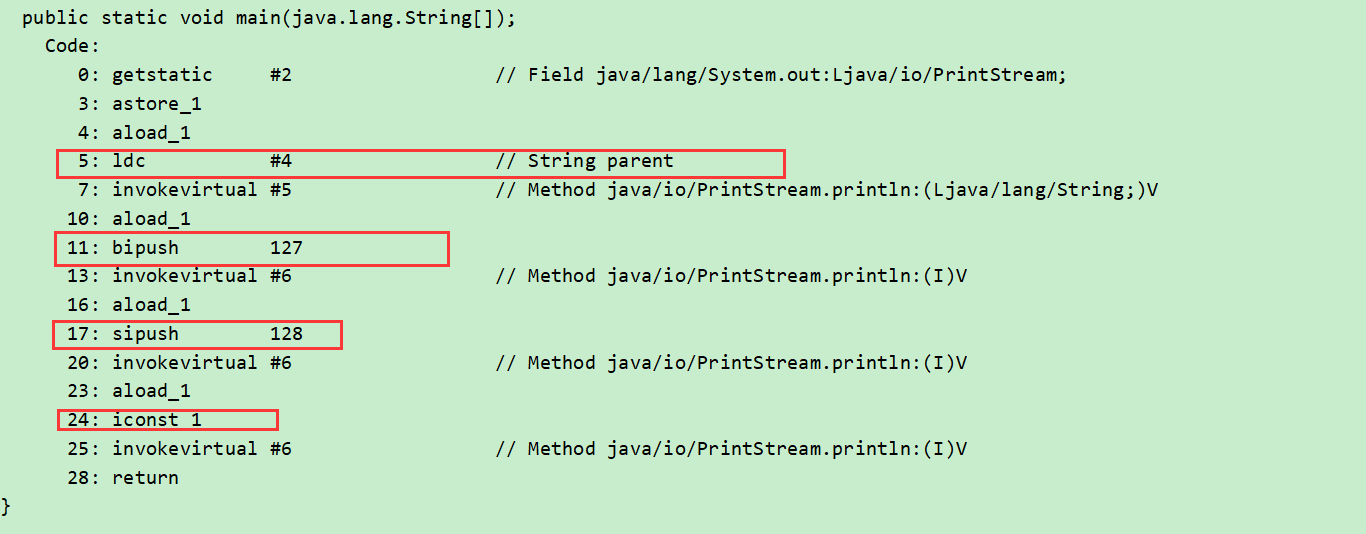
putstatic 放入静态值

invokestatic 创建对象

案例： aActiveUse.Demo2

|  |
| --- |
| **public class** Demo2 {  **public static void** main(String[] args) {  PrintStream p = System.***out***;  p.println(Parent2.***p***);  p.println(Parent2.***s***);  p.println(Parent2.***i***);  p.println(Parent2.***m***);  } }  **class** Parent2 {   **public final static** String ***p*** = **"parent"**;   **public final static byte *s*** = 127;   **public final static short *i*** = 128;   **public final static int *m*** = 1;   **static** {  System.***out***.println(**"parent 被执行"**);  } } |

反编译结果：



ldc是将int float 或是String类型的常量值从常量池汇中推送至栈顶

bipush是将byte类型常量推送到栈顶（-128-127）

sipush是将short常量值（-32768-32767）推送至栈顶

iconst\_1表示将int类型1推送至栈顶。

anewarray 创建一个引用类型的数据，String ， 类

newarray 创建一个原始类型的数组， int float char

## 2.虚拟机参数

格式：

-XX:+<option> 开启option选项

-XX:-<option> 关闭option选项

-XX:<option>=<value>表示将option的值设置为value

## 3.反编译

javap -c 类名