# **JVM**

# 0开篇

方法论

1. 学习优秀的人（从人学习）

缺点：不是自亲身经历，不够深刻

1. 有输入，有输出

做笔记，思维导图，做项目，分享给别人（这点最好）

# 1JVM介绍和HotSpot机讲解

# 2垃圾收集器

# 3类加载机制

## 3.1类加载概述

类加载五步骤

1. 类型加载：查找并加载类的二进制数据。从磁盘上加载字节码文件至内存中
2. 连接：将不同类关联起来，确定连接关系

2.1 验证：确保被加载的类的正确性

2.2 准备：为累的静态变量分配内存，并将其初始化为默认值

2.3 解析： 把勒种的符号引用（间接引用）转换为直接引用

1. 初始化：为类的静态变量赋予正确的初始值
2. 使用
3. 卸载

加载，连接，初始化都是运行期完成的

如下几种情况下，java虚拟机将结束生命周期：

1. 执行了System.exit();
2. 程勋正常执行结束
3. 程序在执行过程中遇到了异常或错误而异常终止
4. 由于操作系统出现错误而导致Java虚拟机进程终止

Java程序对类的使用方式分为两种

1. 主动使用（七种情形）

1.1创建类的实例

1.2访问某个类或接口的静态变量，或者对该静态变量赋值

1.3调用类的静态方法

1.4 反射（如Class.forName(“”)）

1.5 初始化一个类的子类。初始化子类会初始化父类

1.6 Java虚拟机启动时被标明启动类的类（Java Test）

1.7 JDK1.7 开始提供的动态语言支持：java.lang.invoke.MethodHandle实例解析结果REF\_getStatic,REF\_pubStatic,REF\_invokeStatic句柄对应的类没有初始化，则初始化

除了以上其中情况，其他使用Java类的方式都被看做是对类的被动使用，都不会导致类的初始化

1. 被动使用

所有的Java虚拟机实现必须在每个类或接口被Java程序“首次主动使用”时才初始化他们

类的加载：类的加载指的是将类.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后再内存中创建一个java.lang.Class对象（规范并未说明Class对象位于哪里，HotSpot虚拟机将其放在了方法区中）用来封装类在方法区内的数据结构

加载.class文件的方式

1. 从本地系统中直接加载
2. 通过网络下载.class问件
3. 从zip，jar等归档文件中加载.class文件
4. 从专有数据库中提取.class文件
5. 将java源文件动态编译为.class文件

## 3.2代码示例

代码例子1

|  |
| --- |
| package com.gardenia.jvm.classloader;  public class Mytest1 {   public static void main(String[] args) {  */\*\*  \* 执行结果  \* MyParent1 static block  \* hello word  \*/* System.*out*.println(MyChild1.*str*);  } }  class MyParent1 {  public static String *str* = "hello word";  static {  System.*out*.println("MyParent1 static block ");  } }  class MyChild1 extends MyParent1 {  static {  System.*out*.println("MyChild1 static block");  } }  上述代码，访问父类的str字段，输出父类静态代码块，父类静态字段，而子类完全没有初始化动作，说明：  *1.对于静态字段来说，只有指定定义了该字段的类才会被初始化 2.当一个雷在初始化时，要求其父类全部都已经初始化完毕了 3.子类MyChild1类被加载*  启动-XX:+TraceClassLoading 用与追踪类的加载信息    代码吗块二  在子类中增加静态属性str2，并且访问它  package com.gardenia.jvm.classloader;  public class Mytest1 {  public static void main(String[] args) {  */\*\*  \* 执行结果  \* MyParent1 static block   \* MyChild1 static block  \* welcome  \*/*  System.*out*.println(MyChild1.*str2*);  } }  class MyParent1 {  public static String *str* = "hello word";  static {  System.*out*.println("MyParent1 static block ");  } }  class MyChild1 extends MyParent1 {  public static String *str2* = "welcome";  static {  System.*out*.println("MyChild1 static block");  } }  上述：说明。访问子类，父类先初始化。 |

虚拟机设置启动参数的格式

-XX:+<option> 表示开启option选项

-XX:-<option> 表示关闭option选项

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

代码例子2

|  |
| --- |
| 片段1  package com.gardenia.jvm.classloader;  public class MyTest2 {  public static void main(String[] args) {  System.*out*.println(MyParent2.*str*);  }  }  class MyParent2{  public static String *str* = "hello world";  static{  System.*out*.println("MyParent2 static block");  } }  结果：    片段2  package com.gardenia.jvm.classloader;  public class MyTest2 {  public static void main(String[] args) {  System.*out*.println(MyParent2.*str*);  }  }  class MyParent2{  public static final String *str* = "hello world";  static{  System.*out*.println("MyParent2 static block");  } }  结果：    片段一和片段二是差别是字段str一个被final修饰，  final修饰：常量在编译阶段会存入到调用这个常量的方法所在的类的常量池中，本质上，调用类并没有直接引用到定义常量的类，因此并不会触发定义常量的类的初始化  注意：这里是指将常量存放到了MyTest2的常量池中，之后的MyTest2与MyParent2就没有任何关系了；甚至，将MyParent2的class文件删除 |

助记符：

ldc表示将int，float或是String类型的常量值从常量池中推送至栈顶



|  |
| --- |
| 片段1  package com.gardenia.jvm.classloader;  public class MyTest2 {  public static void main(String[] args) {  System.*out*.println(MyParent2.*s*);  }  }  class MyParent2{  public static final String *str* = "hello world";  public static final short *s* = 7;  static{  System.*out*.println("MyParent2 static block");  } }  bipush:表示将单字节(-128到127)的常量值推送至栈顶    片段2  package com.gardenia.jvm.classloader;  public class MyTest2 {  public static void main(String[] args) {  System.*out*.println(MyParent2.*i*);  }  }  class MyParent2{  public static final String *str* = "hello world";  public static final short *s* = 7;  public static final int *i* = 128;  static{  System.*out*.println("MyParent2 static block");  } }  sipush:将一个短整型常量值(-32768到32767)推送至栈顶    片段3  package com.gardenia.jvm.classloader;  public class MyTest2 {  public static void main(String[] args) {  System.*out*.println(MyParent2.*m*);  }  }  class MyParent2{  public static final String *str* = "hello world";  public static final short *s* = 7;  public static final int *i* = 128;  public static final int *m* = 1;  static{  System.*out*.println("MyParent2 static block");  } }  iconst\_1:表示短整型的iconst\_1-iconst\_5 （1-5）数字6开始使用bipush    片段4  package com.gardenia.jvm.classloader;  import java.util.UUID;  public class MyTest3 {   public static void main(String[] args) {  System.*out*.println(MyParent3.*str*);   }  }  class MyParent3{   public static final String *str* = UUID.*randomUUID*().toString();   static {  System.*out*.println("MyParent3 static code");  }  }    当一个常量值并非编译时可以确定的，那么其值就不会被放到调用类的常量池中，  这时在程序运行时，会导致主动使用这个常量所在的类，显然会导致这个类被初始化  片段5  package com.gardenia.jvm.classloader;  public class MyTest4 {  public static void main(String[] args) {  // 创建对象实例，对MyParent4主动使用  MyParent4 myParent4 = new MyParent4();  // 只加载一次静态代码块  MyParent4 myParent5 = new MyParent4();   } }  class MyParent4{  static {  System.*out*.println("MyParent4 static block");  } }    创建对象，首次使用，静态代码块加载一次  片段6  package com.gardenia.jvm.classloader;  public class MyTest4 {  public static void main(String[] args) { // // 创建对象实例，对MyParent4主动使用 // MyParent4 myParent4 = new MyParent4(); // // 只加载一次静态代码块 // MyParent4 myParent5 = new MyParent4();    // 数组的创建不会加载静态代码块  // 创建的实例不是MyParent4的类型  MyParent4[] myParent4s = new MyParent4[1];  // [Lcom.gardenia.jvm.classloader.MyParent4;  // 代码没有显示的数组类型，由jvm在运行期创建出来的  System.*out*.println(myParent4s.getClass());   } }  class MyParent4{  static {  System.*out*.println("MyParent4 static block");  } }  对于数组实例来书，其类型是由JVM在运行期动态生成的，动态生成的类型，其父类型就是object  对于数组来说，JavaDoc经常讲构成数组的元素为Component，实际上就是讲数组降低一个维度后的类型。  片段7  package com.gardenia.jvm.classloader;  public class MyTest4 {  public static void main(String[] args) { // // 创建对象实例，对MyParent4主动使用 // MyParent4 myParent4 = new MyParent4(); // // 只加载一次静态代码块 // MyParent4 myParent5 = new MyParent4();    // 数组的创建不会加载静态代码块  // 创建的实例不是MyParent4的类型  MyParent4[] myParent4s = new MyParent4[1];  // [Lcom.gardenia.jvm.classloader.MyParent4;  // 代码没有显示的数组类型，由jvm在运行期创建出来的  System.*out*.println(myParent4s.getClass());  System.*out*.println(myParent4s.getClass().getSuperclass());   System.*out*.println("========");  int[] ints = new int[1];  System.*out*.println(ints.getClass());  System.*out*.println(ints.getClass().getSuperclass());   char[] chars = new char[1];  System.*out*.println(chars.getClass());   boolean[] booleans = new boolean[1];  System.*out*.println(booleans.getClass());   short[] shorts = new short[1];  System.*out*.println(shorts.getClass());   byte[] bytes = new byte[1];  System.*out*.println(bytes.getClass());    } }  class MyParent4{  static {  System.*out*.println("MyParent4 static block");  } }      片段8  package com.gardenia.jvm.classloader;  */\*\*  \* 当一个接口在初始化时，并不要求其父接口都完成了初始化  \* 只有在真正使用到父接口的时候（如引用接口中所定义的常量时），才会初始化  \*  \*  \*/* public class MyTest5 {   public static void main(String[] args) {   System.*out*.println(MyChild5.*b*);   }  }  interface MyParent5 {  public static final int *a* = 5; }  interface MyChild5 extends MyParent5 {  public static final int *b* = 6; }  片段9  package com.gardenia.jvm.classloader;  */\*\*  \* 静态代码块从上往下执行  \*  \*  \*  \*/* public class MyTest6 {   public static void main(String[] args) {  Singleton instance = Singleton.*getInstance*();  System.*out*.println("counter1=" + Singleton.*counter1*);  System.*out*.println("counter2=" + Singleton.*counter2*);  }  }  class Singleton {   public static int *counter1* = 1;// 准备阶段，赋初始值0，初始化阶段，赋值为1   // 放置在构造函数后面之后，结果就变成了counter2=0 // public static int counter2 = 0;   // 准备阶段，赋初始值null，初始化阶段，赋值new Singleton()  // 这时调用构造函数，count1++结果为2，counter2默认值加1结果为1，  // 但是在初始化阶段执行到 public static int counter2 = 0;时，又给counter2赋值为0  private static Singleton *singleton* = new Singleton();   private Singleton() {  *counter1*++;  *counter2*++;//准备阶段的意义  System.*out*.println(*counter1*);  System.*out*.println(*counter2*);  }   public static int *counter2* = 0;   public static Singleton getInstance() {  return *singleton*;  }   } |

## 3.3类的生命周期

开始

装载类HelloApp

classLoader装载顺利

抛出异常

连接

初始化HelloApp

调用HelloApp.main()

结束

N

N

Y

|  |  |
| --- | --- |
| 验证  准备  解析  初始化  类实例化  垃圾回收和对象终结 | 加载：就是把二级制形式的java类型读入java虚拟机中  验证：  准备：为类变量分配内存，设置默认值，但是在达到初始化之前，类变量都没有初始化为真正的初始值  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成直接引用的过程  初始化：为类变量赋予正确的初始值  类实例化：  为新的对象分配内存  为实例变量赋默认值  为实例变量赋正确的初始值  java编译器为他编译的每一类都至少生成一个实例初始化方法，在java的class文件中，这个实例初始化方法被称为”<init>”,针对源代码中每一个类的构造方法，java编译器都产生一个<init>方法 |

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内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口

## 3.4类加载器类型

两种类型的类加载器：

1. 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):

1. 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式：

类的加载：

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，那么类加载器就不会报告错误。

类的验证：

类被加载后，就进入链接阶段。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。

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。例如对于以下Sample类，在准备阶段，将为int类型的静态变量a分配4个字节的内存空间，并且赋予默认值0，为long类型的静态变量分配8个字节的内存空间，并且赋予默认值0；

Public class Sample{

private static int a = 1; //在静态变量的声明处进行初始化

public static long b;

static {

b=2;// 在金泰代码块中进行初始化

}

}

类的初始化：

加入这个类还没有被加载和连接诶，那就先进行加载和连接

加入类存在直接父类，并且这个父类还没有被初始化，那就先初始化直接父类

加入类中存在初始化预计，那就依次执行这些初始化语句

类的初始化时机：

主动使用（七种情形）

1.1创建类的实例

1.2访问某个类或接口的静态变量，或者对该静态变量赋值

1.3调用类的静态方法

1.4 反射（如Class.forName(“”)）

1.5 初始化一个类的子类。初始化子类会初始化父类

1.6 Java虚拟机启动时被标明启动类的类（Java Test）

1.7 JDK1.7 开始提供的动态语言支持：java.lang.invoke.MethodHandle实例

当Java虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则不适用接口：

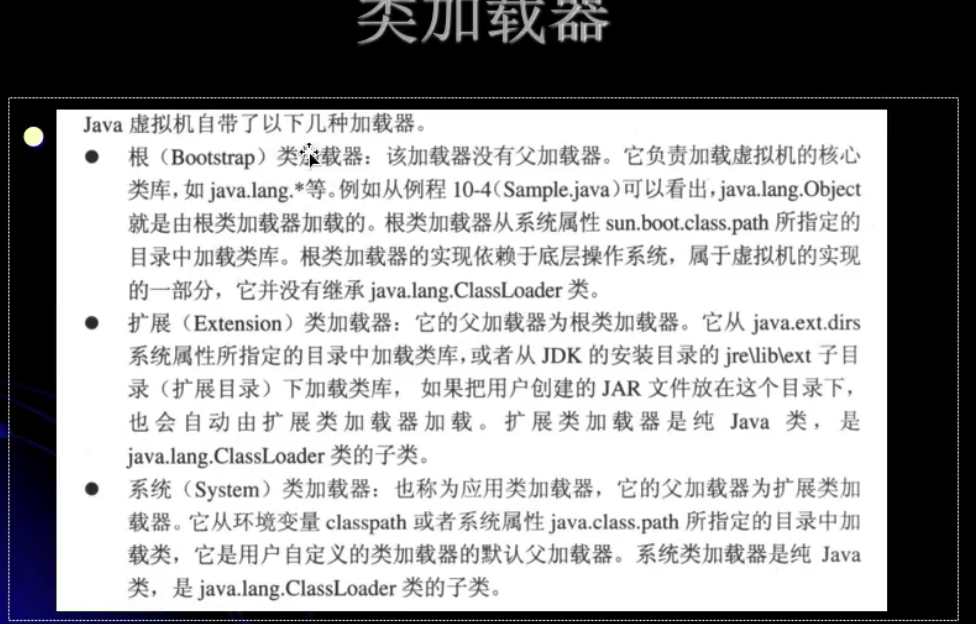
在初始化一个类时，并不会先初始化它所实现的接口

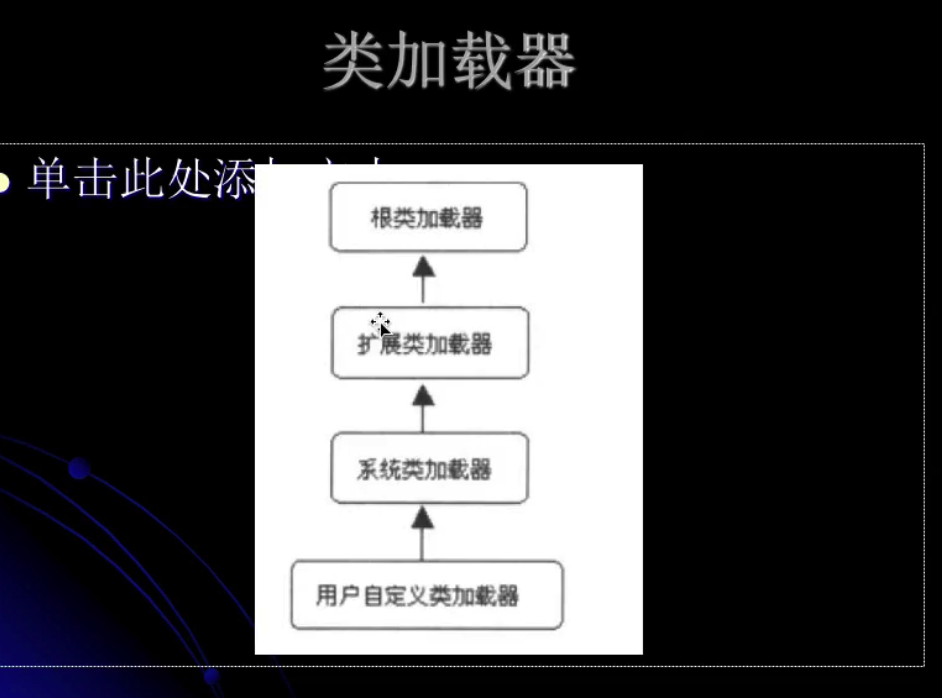
在初始化一个接口时，并不会先初始化它的父接口

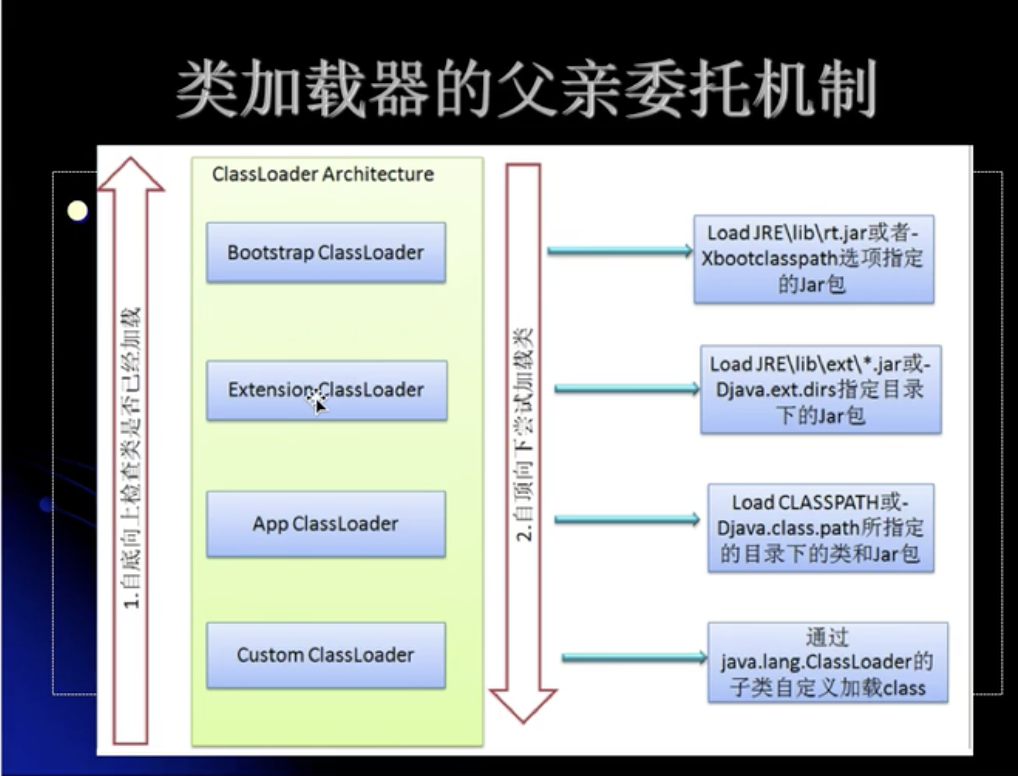
因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化

调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化

类加载器：类加载器用来把类加载到Java虚拟机中，从Jdk1.2版本开始，类的加载过程采用双亲委托机制，这种机制能更好地保证Java平台的安全。在此委托机制中，出看Java虚拟机自带的根类加载器以外，其余的类加载器都有且只有一个父类加载器。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，否则此案有架子器loader1本身加载Sample类。







Bootstrap ClassLoader /启动类加载器

$JAVA\_HOME中jre/lib/rt.jar 里所有的class，由C++实现。不是CLassLoader子类

Extension ClassLoader /扩展类加载器

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME 中 jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

App ClassLoader/ 系统类加载器

负责加载classpath中指定的jar包及目录中class

若有一个类加载器能够成功加载Test类，那么这个类加载器被称为定义类加载器，所有能成功返回Class对象应用的类加载器（包括定义类加载器）都被称为初始类加载器

栗子

|  |
| --- |
| 片段1  package com.gardenia.jvm.classloader;  public class MyTest7 {    public static void main(String[] args) throws ClassNotFoundException {   // 是有根类加载的  Class<?> clazz = Class.*forName*("java.lang.String");  // null  System.*out*.println(clazz.getClassLoader());   //AppClassLoader 系统类加载器  Class<?> clazz2 = Class.*forName*("com.gardenia.jvm.classloader.C");  //sun.misc.Launcher$AppClassLoader@18b4aac2  System.*out*.println(clazz2.getClassLoader());  }  }  class C{  }  片段2  package com.gardenia.jvm.classloader;  // XX:+TraceClassLoading class Parent{  static int *a* = 3;   static {  System.*out*.println("Parent static block");  } }  class Child extends Parent{  static int *b* = 4;   static {  System.*out*.println("Child static block");  }  }  public class MyTest9 {   static {  System.*out*.println("MyTest9 static block");  }   public static void main(String[] args) {  System.*out*.println(Child.*b*);  }  }  结果如下    片段3  package com.gardenia.jvm.classloader;   class Parent3 {  static int *a* = 3;   static {  System.*out*.println("Parent3 static block");  }   static void doSomething() {  System.*out*.println("do something");  } }  class Child3 extends Parent3 {   static {  System.*out*.println("Child3 static block");  }  }  public class MyTest11 {   public static void main(String[] args) {  System.*out*.println(Child3.*a*);  //Parent3 static block  //3  //do something  Child3.*doSomething*();   }  }  片段5  package com.gardenia.jvm.classloader;   class CL{  static {  System.*out*.println("Class CL");  } }   */\*\*  \* 调用ClassLoder类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化  \*/* public class MyTest12 {   public static void main(String[] args) throws Exception {  ClassLoader loader = ClassLoader.*getSystemClassLoader*();  Class<?> clazz = loader.loadClass("com.gardenia.jvm.classloader.CL");  System.*out*.println(clazz);  System.*out*.println("-----------");  // 反射使类初始化  clazz = Class.*forName*("com.gardenia.jvm.classloader.CL");  System.*out*.println(clazz);  }  }    片段6 启动类，扩展类，系统类（应用类）加载器  package com.gardenia.jvm.classloader;  */\*\*  \* sun.misc.Launcher$AppClassLoader@18b4aac2 系统类  \* sun.misc.Launcher$ExtClassLoader@61bbe9ba 扩展类  \* null 根类  \*/* public class MyTest13 {    public static void main(String[] args) {  ClassLoader classLoader = ClassLoader.*getSystemClassLoader*();  System.*out*.println(classLoader);   //sun.misc.Launcher$AppClassLoader@18b4aac2 系统类  //sun.misc.Launcher$ExtClassLoader@61bbe9ba 扩展类  //null 根类  while (null != classLoader){  classLoader = classLoader.getParent();  System.*out*.println(classLoader);  }  }  }    片段7：根加载器和应用加载器示例  package com.gardenia.jvm.classloader;   public class MyTest15 {   public static void main(String[] args) {   String[] strings = new String[2];  // null 根加载器  System.*out*.println(strings.getClass().getClassLoader());   System.*out*.println("----------");  MyTest15[] myTest15s = new MyTest15[2];  //sun.misc.Launcher$AppClassLoader@18b4aac2  System.*out*.println(myTest15s.getClass().getClassLoader());  System.*out*.println("----------");  int[] ints = new int[2];  // null 没有加载器  System.*out*.println(ints.getClass().getClassLoader());  System.*out*.println("----------");  Integer[] integers = new Integer[2];  // null 根加载器：加载jdk rt.jar下的包里的类  System.*out*.println(ints.getClass().getClassLoader());    }   }  片段8 自定义类加载器  package com.gardenia.jvm.classloader;  import jdk.internal.util.xml.impl.Input;  import java.io.ByteArrayOutputStream; import java.io.File; import java.io.FileInputStream; import java.io.InputStream;  public class MyTest16 extends ClassLoader {   private String classLoaderName;   private final String fileExtension = ".class";   public MyTest16(String classLoaderName) {  super();// 系统类加载器当做该类加载器的父加载器  this.classLoaderName = classLoaderName;  }   public MyTest16(ClassLoader parent, String classLoaderName) {  super(parent);  this.classLoaderName = classLoaderName;  }   @Override  public String toString() {  return "[" + classLoaderName + "]";  }   @Override  protected Class<?> findClass(String className) throws ClassNotFoundException {  byte[] data = this.loadClassData(className);  return this.defineClass(className, data, 0, data.length);  }   private byte[] loadClassData(String name) {  InputStream is = null;  byte[] data = null;  ByteArrayOutputStream baos = null;  try {  this.classLoaderName = this.classLoaderName.replace(".", "/");  is = new FileInputStream(new File(name + this.fileExtension));  baos = new ByteArrayOutputStream();  int ch = 0;  while (-1 != (ch = is.read())) {  baos.write(ch);  }  data = baos.toByteArray();  } catch (Exception e) {  e.printStackTrace();  } finally {  try {  is.close();  baos.close();  } catch (Exception ex) {  ex.printStackTrace();  }  }  return data;  }    public static void test(ClassLoader classLoader) throws Exception {  Class<?> clazz = classLoader.loadClass("com.gardenia.jvm.classloader.Mytest1");  Object object = clazz.newInstance();  System.*out*.println(object);  }   public static void main(String[] args) throws Exception {  MyTest16 loader1 = new MyTest16("loader1");  // com.gardenia.jvm.classloader.Mytest1@61bbe9ba  *test*(loader1);  } } |

# 4线程与锁

# 5JIT与GC日志生成与分析

# 6虚拟机监控工具等

# 7其他知识点