1. Jconsole
2. Jvisualvm
3. Jmap
4. 类加载
   1. Java代码中，类型的加载，连接与初始化过程都是在程序运行期间完成的
   2. 加载（字节码加载到jvm中），连接（类之间的调用关系，符号引用转换直接引用）
   3. 初始化
5. Java虚拟机结束生命周期

1), 执行了System.exit()

2), 程序正常结束

3), 程序执行过程中遇到异常或错误而异常终止

4),操作系统故障

6， 类的加载,连接与初始化

1),加载： 查找并加载类的二进制数据

2),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（整型 0, Boolean false …）

解析: 把类中的符号引用转换为直接引用

3),初始化：为类的静态变量赋予正确的初始值

7， 类的使用与卸载

8，类的加载，连接，初始化

Java 程序对类的使用方式可分为两种

1. 主动使用
2. 被动使用

所有的Java虚拟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

如下是对类的主动使用，除此之外都是被动使用

1,**创建类的实例**

2,**访问某个类或接口的静态变量，或者对该静态变量赋值**

3,**调用类的静态方法**

4,**反射（Class.forName(“com……”)）**

5,**初始化一个类的子类**

6,java **虚拟机被标记为启动类的类**

7,jdk7 开始提供的动态语言的支持：

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。

9，类的加载

1，类的加载指的是将类的.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对象用来封装类再方法区的数据结构

2，加载.class文件的方式

1》从本地系统直接加载

2》通过网络下载.class

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

5》将java源文件动态编译为.class文件

加载： 就是把二进制形式的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。但是在到达初始化之前，类变量都没有初始化为真正的初始化值，

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编译器为它编译的每个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，而且向java 程序员提供了访问方法区内的数据机构的接口

1. 有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 or app

2）用户自定义的类加载器

Java.lang.ClassLoader 类的子类

用户可以定制类的加载方式

1. 类加载器加载时并不需要类的首次使用。

Jvm 规范允许类加载器在预料某个类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文件缺失或者存在错误，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

类的初始化步骤

1. 假如这个类还没有被加载和连接，那就先进行加载和连接
2. 假如类存在直接父类，而且这个父类还没有被初始化，那就先初始化直接父类
3. 假如类中存在初始化语句，那就依次执行这些初始化语句

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，但是这条规则并不适用于接口

1）， 初始化一个类时，并不会先初始化它所实现的接口。

2）， 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。

|  |
| --- |
| **public class** MyTest10 {  **public static void** main(String[] args) {  System.***out***.println(MyChild10.*b*);  } }  **interface** MyParent10{  **public static** Thread ***thread*** = **new** Thread(){  {  System.***out***.println(**"MyParent10 invoked!"**);  }  }; } **class** MyChild10 **implements** MyParent10{  **static int** *b*= 5; } |

**类的初始化时机**

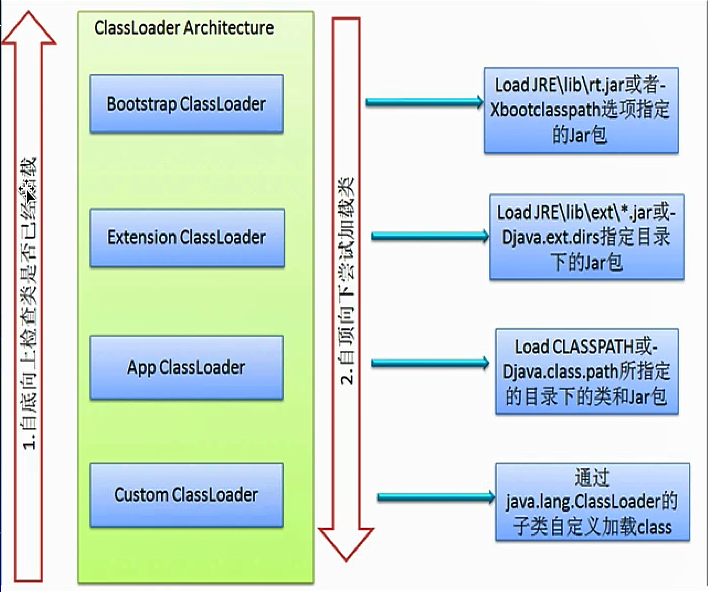
只有当程序访问的静态变量或者静态方法确实在当前类或者当前接口中定义时，才可以认为是对类或接口的主动使用，掉用ClassLoader类的loadClass 方法加载一个类，并不是对类的主动使用，不会导致类的初始化

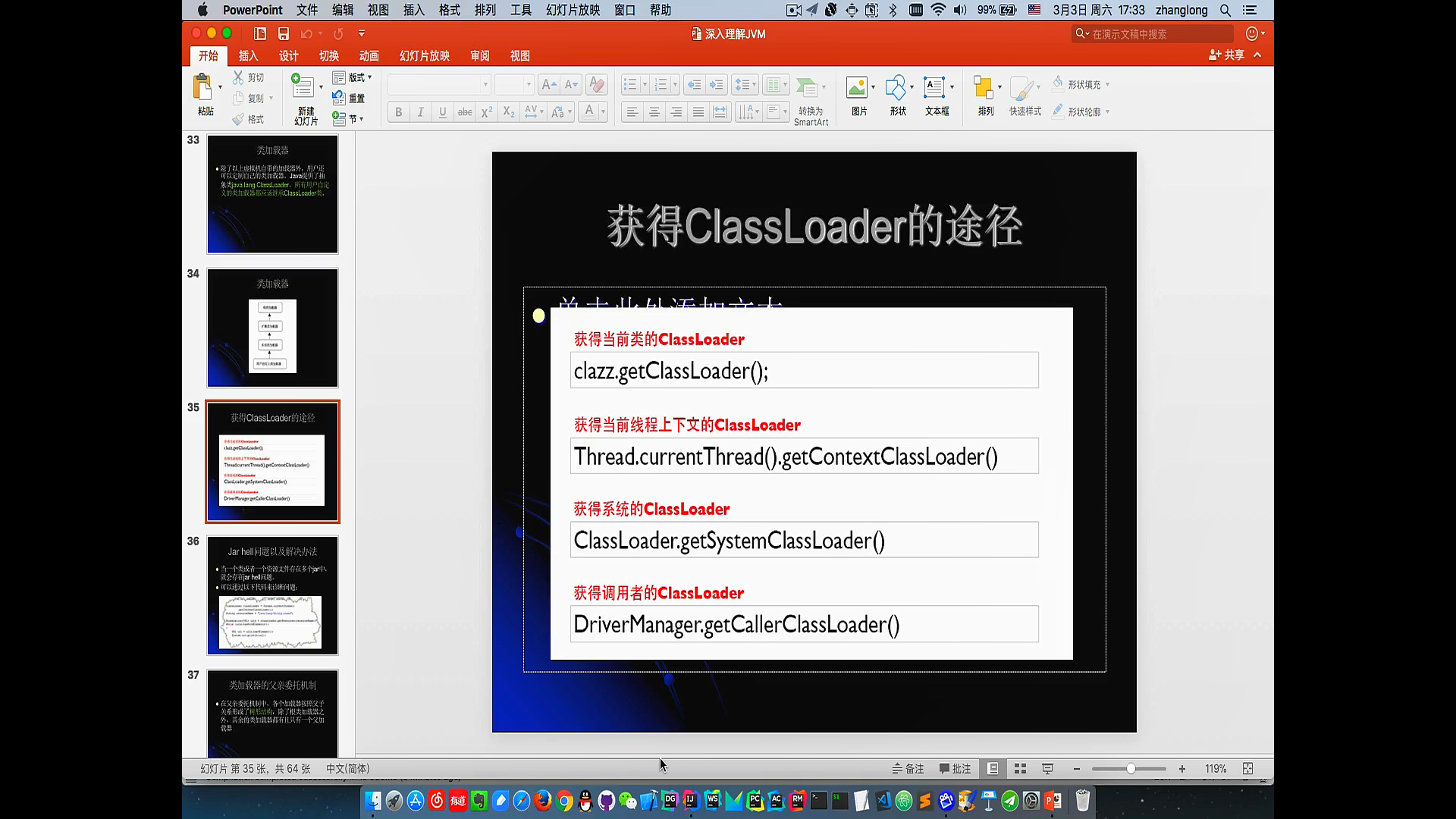
|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  System.***out***.println(Demo2.*str*);  } } **class** Demo1{  **static** String *str*=**"xxxxx"**;  **static**{  System.***out***.println(**"Demo1 ............"**);  } }  **class** Demo2 **extends** Demo1{  **static** {  System.***out***.println(**"Demo2 ............."**);  } }  Demo1 ............  xxxxx |

**类加载器**

1. 根类加载器： 该加载器没有父类加载器。它负责加载虚拟机的核销类库，如java.lang.\* 等。根类加载器从系统属性sun.boot.class.path 所指定的目录中加载类库。根类加载器的实现依赖于底层操作系统，属于虚拟机的实现的一部分，它并没有继承java.lang.ClassLoader类
2. 拓展类加载器： 它的父加载器为根类加载器。它从java.ext.dirs系统属性所指定的目录中加载类库，或者从jdk的安装目录的 jre\lib\ext 子目录下加载类库，如果把用户创建的jar文件放在这个目录下，也会自动由拓展类加载器加载，拓展类加载器是纯java类，是java.lang.ClassLoader类的子类
3. 系统类加载器：也称为应用类加载器，它的父加载器为扩展类加载器，它从环境变量classpath或者系统属性java.class.path所指定的目录中加载类，它是用户自定义的类的加载器的默认父加载器。系统类加载器是纯java类，是java.lang.ClassLoader类的子类

**类加载器的父亲委托机制**





|  |
| --- |
| java.lang  public abstract class ClassLoader  extends Object  A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class. Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a "class file" of that name from a file system.  Every Class object contains a reference to the ClassLoader that defined it.  Class objects for array classes are not created by class loaders, but are created automatically as required by the Java runtime. The class loader for an array class, as returned by Class.getClassLoader() is the same as the class loader for its element type; if the element type is a primitive type, then the array class has no class loader.  Applications implement subclasses of ClassLoader in order to extend the manner in which the Java virtual machine dynamically loads classes.  Class loaders may typically be used by security managers to indicate security domains.  The ClassLoader class uses a delegation model to search for classes and resources. Each instance of ClassLoader has an associated parent class loader. When requested to find a class or resource, a ClassLoader instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself. The virtual machine's built-in class loader, called the "bootstrap class loader", does not itself have a parent but may serve as the parent of a ClassLoader instance.  Class loaders that support concurrent loading of classes are known as parallel capable class loaders and are required to register themselves at their class initialization time by invoking the ClassLoader.registerAsParallelCapable method. Note that the ClassLoader class is registered as parallel capable by default. However, its subclasses still need to register themselves if they are parallel capable. In environments in which the delegation model is not strictly hierarchical, class loaders need to be parallel capable, otherwise class loading can lead to deadlocks because the loader lock is held for the duration of the class loading process (see loadClass methods).  Normally, the Java virtual machine loads classes from the local file system in a platform-dependent manner. For example, on UNIX systems, the virtual machine loads classes from the directory defined by the CLASSPATH environment variable.  However, some classes may not originate from a file; they may originate from other sources, such as the network, or they could be constructed by an application. The method defineClass converts an array of bytes into an instance of class Class. Instances of this newly defined class can be created using Class.newInstance.  The methods and constructors of objects created by a class loader may reference other classes. To determine the class(es) referred to, the Java virtual machine invokes the loadClass method of the class loader that originally created the class.  For example, an application could create a network class loader to download class files from a server. Sample code might look like:  ClassLoader loader = new NetworkClassLoader(host, port);  Object main = loader.loadClass("Main", true).newInstance();  . . .    The network class loader subclass must define the methods findClass and loadClassData to load a class from the network. Once it has downloaded the bytes that make up the class, it should use the method defineClass to create a class instance. A sample implementation is:  class NetworkClassLoader extends ClassLoader {  String host;  int port;    public Class findClass(String name) {  byte[] b = loadClassData(name);  return defineClass(name, b, 0, b.length);  }    private byte[] loadClassData(String name) {  // load the class data from the connection  . . .  }  }    Binary names  Any class name provided as a String parameter to methods in ClassLoader must be a binary name as defined by The Java™ Language Specification.  Examples of valid class names include:  "java.lang.String"  "javax.swing.JSpinner$DefaultEditor"  "java.security.KeyStore$Builder$FileBuilder$1"  "java.net.URLClassLoader$3$1"    Since:  1.0  See Also:  resolveClass(Class) |

**关于命名空间**

**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无法访问到子加载器所加载的类**

助记符号：getstatic/putstatic/invokestatic

Ldc:将int,float,String 类型的常量值从常量池中推送至栈顶

Bipush:将 -128~127 常量值从常量池中推送至栈顶

Sipush:将短整形常量值推送至栈顶(-32767~32767)

Iconst\_1:1

Iconst\_2:2

Iconst\_3:3

Iconst\_4:4

Iconst\_5:5

anewarray:表示创建一个引用类型的（类，接口，数组），并将其引用值压入栈顶

newarray:表示创建一个指定的原始类型（如， int，float,char 等）的数组，并将其引用压入栈顶

