# 深入理解JVM

## 类的加载

在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

提供了更大的灵活性，增加了更多的可能性

### 类的加载过程

**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，将其放在运行时方法区内(jdk1.8后为Meatspace)，然后在内存中创建一个java.lang.Class的对象（JVM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了）用来封装类在方区内的数据结构

### Java虚拟机与程序的生命周期

类的加载深入剖析：Java虚拟机与程序的生命周期

在以下几种情况下，Java虚拟机将结束生命周期

1. 执行了System.exit()方法
2. 程序正常执行结束
3. 程序在执行中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

### 类的生命周期

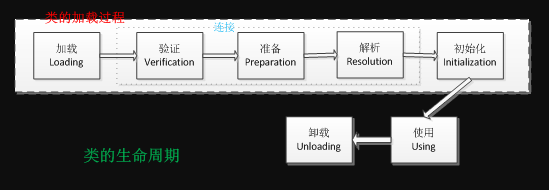
1. 加载：查找并加载类的二进制数据
2. 连接

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：把类的符合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：



### Java程序对类

Java程序对类的使用方式可分为两种

* + 主动使用
  + 被动使用

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

主动使用（七种）：

1. 创建类的实例
2. 访问某个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
5. 初始化一个类的子类
6. Java虚拟机启动时被表明为启动类的类（Java Test）
7. JDK1.7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\_puStatic，REF\_invokeStatic句柄对应的类没有初始化，则初始化

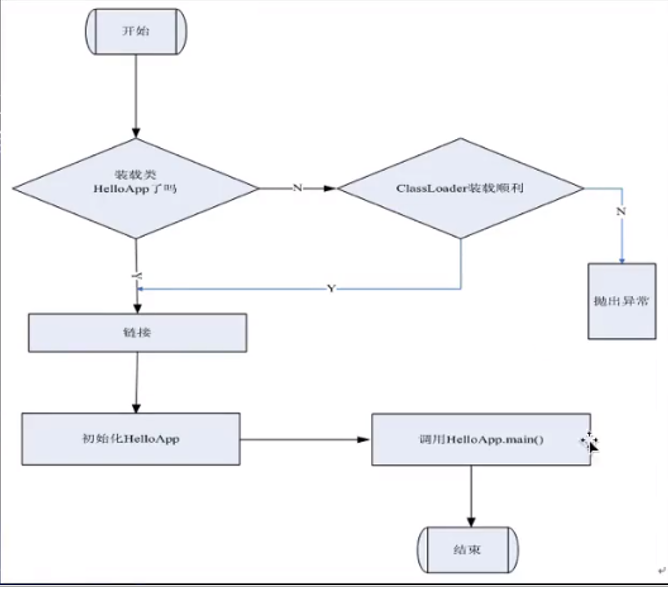
除了以上七种情况，其他使用Java类的方式都被看作是对类的被动使用，都不会导致类的初始化

### 加载.class文件的方式

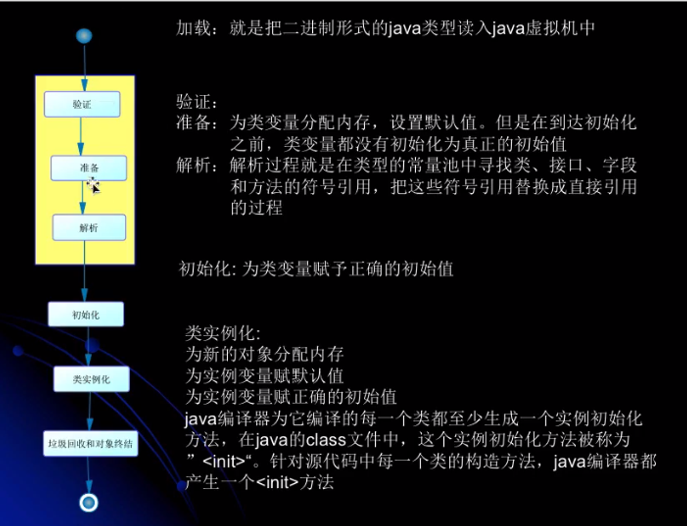
加载.class文件的方式

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

### 类加载流程



### 类加载与类的实例化



### Class对象

* 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并且向Java程序员提供了访问方法区内的数据结构的接口

### 类加载器

类加载器可以分为两类：

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 App）

2、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

### 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.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

### 验证

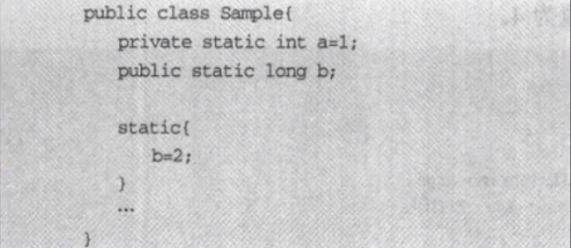
类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

类的验证的内容：

* + 类文件的结构检查
  + 语义检查
  + 字节码验证
  + 二进制兼容性的验证

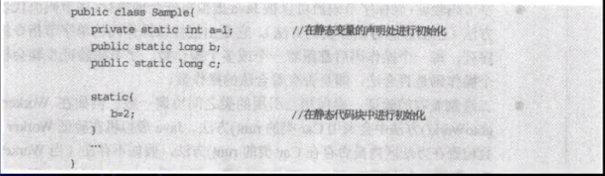
### 准备

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字节的内存空间，并且赋予默认值0，为long类的静态变量b分配8个字节的内容空间，并且赋予默认值0.

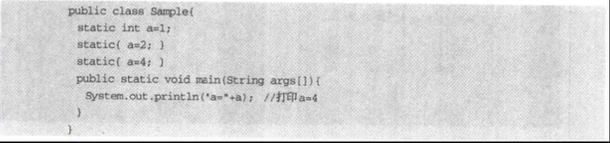


### 初始化

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。在程序中，静态变量的初始化有两种途径：（1）在静态变量的声明处进行初始化；（2）在静态代码块中进行初始化。例如在以下代码中，静态变量a和b都被显式初始化，而静态变量c没有被显式初始化，它将保持默认值0。



静态变最的声明语句，以及静态代码块都被看做类的初始化语句，Java虚拟机会按照初始化语句在类文件中的先后顺序来依次执行它们。例如当以下Sample类被初始化后，它的静态变量a的取值为4。



### 初始化步骤

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，并且这个父类还没有被初始化，那就先初始化直接父类
3. 假如类存在初始化语句，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，重要）

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（如CIass.forName('com.test.Test"))
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(JavaTest）

除了上述七种情形，其他使用Java类的方式都被看作是被动使用，不会导致类的初始化。

**注意：**

**1、**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。

* 在初始化一个类时，并不会先初始化它所实现的接口。
* 在初始化一个接囗时，并不会先初始化它的父接囗。

因此，一个父接口并不会因为它的了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。

2、只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用

3、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拟机中。从JDK1.2版本开始，类的加载过程采用父亲委托机制，这种机制能更好地保证Java平台的安全。在此委托机制中，除了Java靠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。

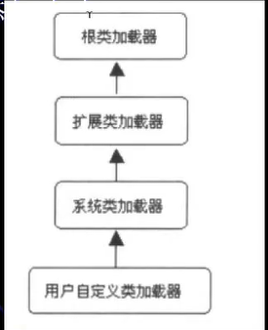
### 类加载器分类

Java虚拟机自带了以下几种加载器：

* 根（Bootstrap）类加载器：该加载器没有父加载器。它负责加载虚拟机的核心类库，如java.lang.\*等。例如从例程10-4(sample.java)可以看出，java.lang.Object就是由根类加载器加载的。根类加我器从系统属性sun.bootclass.path所指定的目录中加线类库。根类加载器的实现依赖于底层操作糸统，属于虚拟机的实现的一部分，它并没有继承java.lang.ClassLoader类。
* 扩展（Extension)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厍，或者从JDK的安装目录的jre\lib\ext子目录（扩展目录）下加载类库，如果把用户创建的JAR文件放在这个目录下，也会自动由扩展类加载器加载。扩展类加载器是纯Java类，是java.lang.ClassLoader类的子类。
* 系统（System）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。

除了以上虚拟机自带的加载器外，用户还可以定制自己的类加载器。Java提供了抽象类java.lang.ClassLoader，所有用户自定义的类加载器都应该继承ClassLoader类。

### 类加载器关系



### 获得CIassLoader的途径

**获得当前类的ClassLoader**

Clazz.getClassLoader();

**获得当前线程上下文的Classloader**

Thread.currentThread().getContextCIassLoader()

**获得系统的ClassLoader**

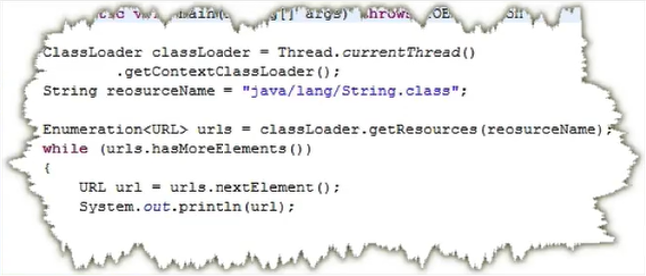
CIassLoader.getSystemClassLoader()

获得调用者的ClassLoader

DriverManager.getCaIIerClassLoader()

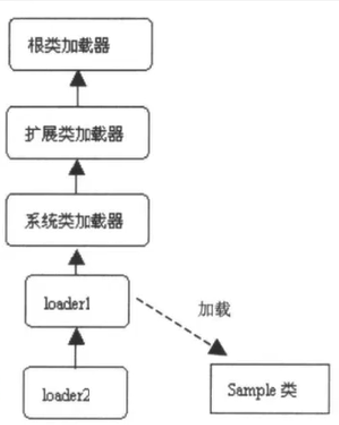
### Jar hell问题以及解决办法

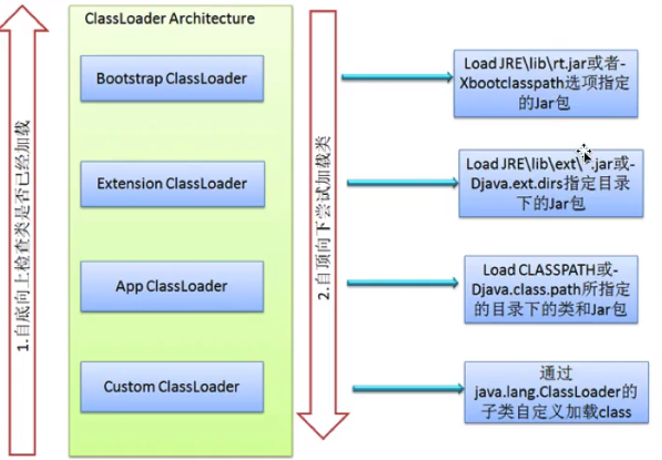
当一个类或者一个资源文件存在多个jar中，就会存在jarhell问题。可以通过以下代码来诊断问题：



### 类加载器的父亲委托机制

在父亲委托机制中，各个加载器按照父子关系形成了**树形结构**，除了根类加载器之外，其余的类加载器都有且只有一个父加载器





* 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
* AppC lassLoader/系统类加载器  
  负责加载classpath中指定的jar包及目录中class

注意：

1、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

有能成功返回Class对象引用的类加载器（包括定义类加载器）都被称为初始类加载器；

假loaderl实际加载了Samp|e类，则loaderl为Sample类的定义类加载器，loader2与loaderl为Sample类的初始类加载器

2、需要指出的是，加载器之间的父子关系实际上指的是加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。在子加载器对象中包装了一个父加载器对象。例如以下loaderl和loader2都是MyClassLoader类的实例，并且]oade2包装了loader1，loaderl是loader2的父加载器。

ClassLoader loaderl=new MyClassLoader();

//参数loaderl将作为loader2的父加载器

ClassLoader loader2=new MyClassLoader(loader1);

3、父亲委托机制的优点是能够提高软件系统的安全性。因为在此机制下，用户自定义

的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码

代替由父加载器加载的可靠代码．例如，java.langObject类总是由根类加载器加载，其

他任何用户自定义的类加载器都不可能加载含有恶意代码的java.lang.Object类

### 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，不会出现类的完整名字（包括类的包名）相同的两个类
* 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）相同的两个类

### 创建用户自定义的类加载器

要创建用户自己的类加载器，只需要扩展java.lang.ClassLoader类，然后覆盖它的findClass(String name)方法即可，该方法根据参数指定的类的名字，返回对应的Class对象的引用。

### 不同类加载器的命名空间关系

* 同一个命名空间内的类是相互见的。
* 子加载器的命名空间包含所有父加载器的命名空间。因此由子加载器加载的能看见父加载器加载的类．例如系统类加载器加载的类能看见根类加载器加载的类。
* 由父加载器加载的类不能看见子加载器加载的类。
* 如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见

## 类的卸载

* 当MySample类被加载、连接和初始化后，它的生命周期就开始了。当代表MySample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，MySample类在方法区内的数据也会被卸载，从而结束MySample类的生命周期。
* 一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期
* 由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。前面己经介绍过，Java虚拟机自带的类加载器包括根类加载器、扩展类加载器和系统类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的的Class对象，因此这些Class对象始终是可触及的。
* 由用户自定义的类加载器所加载的类是可以被卸载的
* 运行以上程序时，Sample类由loaderl加载。在类加载器的内部实现中，用一个Java集合来存放所加载类的引用。另一方面，一个Class对象总是会引用它的类加载器，调用Class对象的getClassLoader()方法，就能获得它的类加载器。由此可见，代表Sample类的CIass实例与loaderl之间为双向关联关系。
* 一个类的实例总是引用代表这个类的Class对象。在Object类中定义了getClass()方法，这个方法返回代表对象所属类的Class对象的引用。此外，所有的Java类都有一个静态属性class，它引用代表这个类的Class对象