Class Loader

--- CoreJava Advanced , Moduel 5

ORACLE HIGH-END IT TRAINING , JAVA ADVANCED

内容提要

- 从一个诡异的问题说起
- 类的生命周期
- 类加载器

第一个测试

单例类

```
public class Singleton1 {

        private static Singleton1 singleton = new Singleton1();
        public static int a;
        public static int b = 0;

        private Singleton1() {
                super();
                a++;
                b++;
        }

        public static Singleton1 getInstence() {
                return singleton;
        }

}
```

第一个测试

测试类

```
public class MyTest1 {
        public static void main(String[] args) {
                Singleton1 mysingleton = Singleton1.getInstence();
                System.out.println(mysingleton.a);
                System.out.println(mysingleton.b);
        }
}
```

第一个测试

测试类

```
public class MyTest1 {
        public static void main(String[] args) {
                Singleton1 mysingleton = Singleton1.getInstence();
                System.out.println(mysingleton.a);
                System.out.println(mysingleton.b);
        }
}
```

第二个测试

单例类

```
public class Singleton2 {

        public static int a;
        public static int b = 0;
        public static Singleton2 singleton = new Singleton2();

        private Singleton2() {
                super();
                a++;
                b++;
        }

        public static Singleton2 GetInstence() {
                return singleton;
        }

}
```

第二个测试

测试类

```
public class MyTest2 {
        public static void main(String[] args) {
                Singleton2 mysingleton = Singleton2.GetInstence();
                System.out.println(mysingleton.a);
                System.out.println(mysingleton.b);
        }
}
```

JVM的生命周期

- **一个 JVM 实例是操作系统的一个进程**
  - 当调用 java 命令运行某个 Java 程序时
  - 该命令 (java) 会导致启动一个 JVM 进程
    - Windows 下在 任务管理器 中可以找到名为 javaw.exe 的进程
    - Linux 下在类似任务管理器的程序中或者用命令可以查看到名为 java 的进程
  - 所运行的 Java 程序的所有线程、所有变量都处于该 JVM 进程中

- **当系统出现以下情况时，JVM 进程将被终止**
  - 正在运行的 Java 程序正常运行结束
  - 程序中显式调用 System 或 Runtime 中的 exit() 结束程序
  - 程序执行过程中遇到未捕获的异常或错误而结束
  - 程序所在的平台强制结束了 JVM 进程

JVM的生命周期

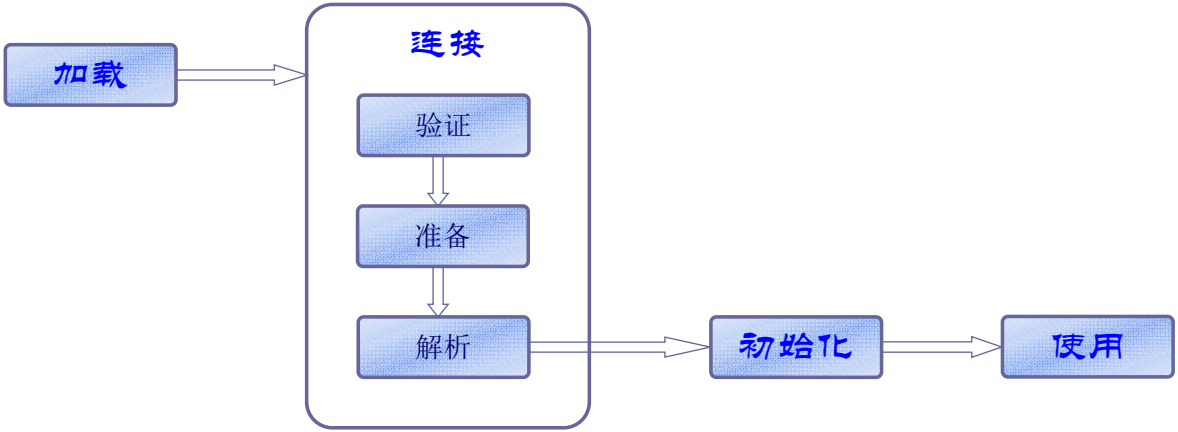
JVM进程结束后其在内存中的状态将会丢失

```
public class JvmProcessStatus {

        public static int count = 0 ;

        public static void main(String[] args) {
                JvmProcessStatus.count ++ ;
                System.out.println( JvmProcessStatus.count );
        }

}
```

```
public class Test {
        public static void main(String[] args) {
                JvmProcessStatus.count ++ ;
                System.out.println( JvmProcessStatus.count );
        }

}
```

类的生命周期

类的生命周期的开始



类的生命周期

类的加载

- 把类对应的 .class 文件中的二进制数据读入内存中
  - 这些二进制数据被存放在运行时数据区的方法区内
- 然后根据加载的二进制数据在堆区创建一个 java.lang.Class 对象
  - 该 Class 对象也是类加载阶段的最终产品
    - 该 Class 对象封装类该类在方法区存放的数据结构
    - 向 Java 程序提供了访问该类在方法区内的数据结构的接口

类的生命周期

类的加载

- JVM 可以从多种来源加载类的二进制数据
  - 从本地文件系统中加载类的 .class 文件(最常见方式)
  - 通过网络加载类的 .class 文件
  - 从 zip 、jar 或其他类型的归档文件中提取 .class 文件
  - 从一个专有数据库中提取 .class 文件
  - 把一个 Java 源文件动态编译为 .class 文件
- 类的加载是由类加载器完成的
  - 加载器分为两种：
    - JVM 自带的加载器，包括启动类加载器、扩展类加载器、系统类加载器
    - 用户自定义的类加载器，一般都是 java.lang.ClassLoader 类的子类的实例
  - 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 规范允许在预料某个类将要被使用时预先加载它
    - 如果预先加载该类时遇到 .class 文件缺失或其它错误，那么在首次主动使用该类时抛出 LinkageError 错误；如果该类始终未使用过，则不抛出该错误

类的生命周期

类的连接

- 当类被加载后，就进入到连接阶段
  - 连接是把已读入到内存的类的二进制数据合并到Java运行时环境(JRE)中去
  - 连接又分为三个阶段：验证、准备、解析
- 类的验证
  - 验证的目的是保证类有正确的内部结构，并且与其它类协调一致
    - 如果 JVM 检查到错误，就会抛出 Error 对象
  - 验证主要包括以下内容
    - 类文件的结构检查: 确保文件遵循 Java 文件的固定格式
    - 语义检查: 确保类本身符合 Java 语言的语法规定
    - 字节码验证: 确保字节码流可以被 JVM 安全地执行
      - 字节码流代表 Java 方法(含静态和非静态)，它是被称作操作码的单字节指令组成的序列，每个操作码后都跟着一个或多个操作数
      - 字节码验证会检查每个操作码是否合法，即是否有合法的操作数
    - 二进制兼容的验证: 确保相互引用的类之间协调一致
      - 比如 A 类中调用 B 类的 b() 方法，检查 B 中是否有 b() 方法存在

类的生命周期

类的连接

类的准备

- 在准备阶段，JVM 为类的静态变量分配内存，并设置默认的初始值

```
public class Example1 {

    private static int intVar = 8985 ;

    public static long longVar ;
    public static short s = 0 ;

    static{
        longVar = 9527 ;
    }

}
```

在准备阶段，对 Example1 的操作:

为 intVar 分配 4 个字节的内存空间，并赋予其默认值 0 ;

为 longVar 分配 8 个字节的内存空间，并赋予其默认值 0L ;

类的生命周期

类的连接

类的解析

- 在解析阶段，JVM 会把类的二进制数据中的符号引用替换为直接引用
  - 下面代码中 printSize( ) 方法调用了 List 的 size() 方法，在该类对应的二进制数据中包含了一个 对 List 中 size() 方法的符号引用
    - 这个符号引用由 size() 方法的全名和相关描述符组成
  - 解析阶段，JVM 会把这个符号引用替换为一个指针
    - 这个指针指向 List 的 size() 方法在方法区的内存位置
    - 这个指针即为直接引用

```
public  void  printSize( List list ){
    System.out.println( list.size() );  //在当前类的二进制数据中表示为符号引用
}
```

类的生命周期

类的初始化

- 初始化阶段，JVM执行类的初始化语句，为静态变量赋予初始值
  - 静态变量的初始化途径
    - 在静态变量的声明处进行初始化
    - 在静态代码块中进行初始化
  - 静态变量的声明语句及静态代码块都应该被视为类的初始化语句
    - JVM 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
  - 这里的静态变量指不能作为编译时常量的静态变量
    - Java 编译器 和 JVM 对 编译时常量有特殊处理方式(见后面类的初始化时机)
- 类初始化的一般步骤
  - 如果该类还没有被加载和连接，那么先加载和连接该类
  - 如果该类存在直接父类，但该父类还未初始化，则先初始化其直接父类
  - 如果该类中存在初始化语句，则依次执行这些初始化语句

类的生命周期

类的初始化时机

- JVM 只有在首次主动使用某个类或接口时才会初始化它
- 只有 6 种活动被看作是程序对类或接口的主动使用：
  - 创建类的实例
    - 包括使用 new 语句、通过反射、克隆、反序列化等手段来创建实例
  - 调用类的静态方法
  - 访问某个类或接口的静态变量，或对该静态变量赋值
  - 调用 Java API 中的某些反射方法
    - 比如 Class.forName("java.util.ArrayList") 可以导致 ArrayList 被初始化
  - 初始化一个类的子类 (会导致初始化本类)
  - JVM 启动时被标明为启动类的类
    - 如果使用 java ListTest，则 ListTest 就是启动类，JVM 会先初始化它
- 除了上述 6 中活动外，其它一律视作被动使用
- 被动使用不会导致本类的初始化

类的生命周期

类的初始化时机

被动使用的特例

- 程序中对编译时常量的使用视作对类的被动使用
  - 对于 final 修饰的变量，如果编译时就能确定其取值，即被看作 编译时常量
    - 编译时常量如：public static final int a = 2 * 3；
    - JVM 的加载和连接阶段，不会在方法区内为某个类的编译时常量分配内存
  - 对于 final 修饰的变量，如果编译时就不能确定其取值，则不被看作编译时常量
    - 非编译时常量如：public static final long time = System.currentTimeMillis()；
    - 使用该类型的静态变量将导致当前类被初始化（主动使用）
- JVM初始化某个类时，要求其所有父类都已经被初始化
  - 该规则不适用 于 接口 类型
  - 一个接口不会因为其子接口或实现类的初始化而初始化
  - 除非使用了该接口的静态属性
- 只有当程序访问的静态变量或静态方法的确在当前类或接口定义时，才能看作是对类或接口的主动使用
  - 如果使用了 Sub.method()，而 method() 是继承自 Base，则只初始化 Base 类
- 调用 ClassLoader 的 loadClass( ) 加载一个类，不属于对类的主动使用

类的生命周期

类的卸载

- 当代表某个类的 Class 对象不再被引用，即不可触及时，
- 该 Class 对象就会结束生命周期
- 其所对应的方法区内的数据也会被卸载，本类生命周期结束
- JVM 自带的类加载器所加载的类，在JVM 结束前不会被卸载

类加载器

类加载器用来把类加载到 JVM 中

- 从 JDK 1.2 版本开始，类的加载过程采用父亲委托机制
  - 设 loader 要加载 A 类，则 loader 首先委托自己的父加载器去加载 A 类，如果父加载器能加载 A 类则由父加载器加载，否则才由 loader 本身来加载 A 类
  - 这种机制能更好地保证 Java 平台的安全性
- 父亲委托机制中，每个类加载器都有且只有一个父加载器
  - 除了 JVM 自带的根类加载器（Bootstrap Loader）

类加载器

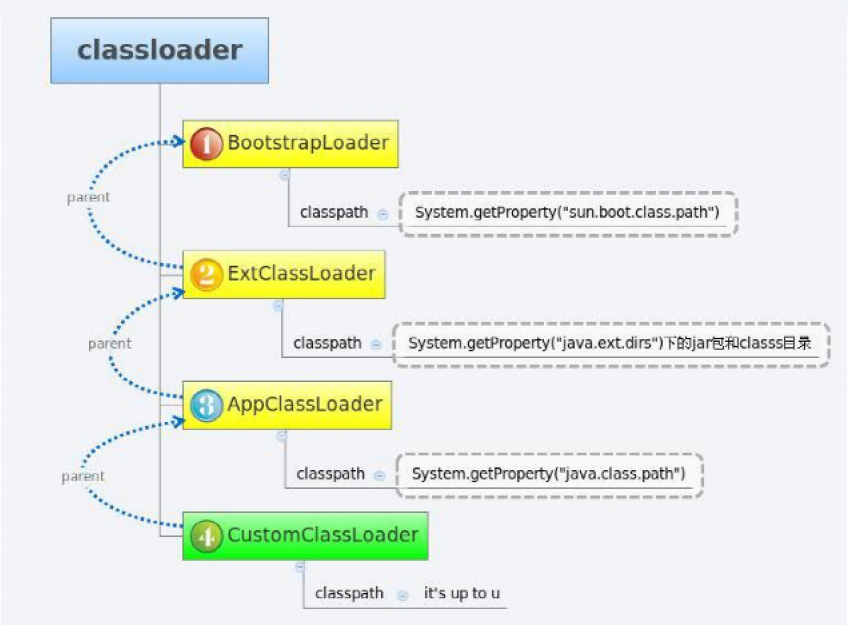
JVM 自带的类加载器

- 根类加载器 (BootstrapLoader)
  - 负责加载虚拟机的核心类库，比如 java.lang.* 等
    - 从 系统属性 sun.boot.class.path 所指定的目录中加载类库
  - 该加载器没有父加载器，它属于 JVM 的实现的一部分 (用C++实现)
- 扩展类加载器 (ExtClassLoader)
  - 其父加载器为 BootstrapLoader 类的一个实例
  - 该加载器负责从 java.ext.dirs 系统属性所指定的目录中加载类库
  - 或者从 JDK_HOME/jre/lib/ext 目录中加载类库
  - 该加载器对应的类是纯 Java 类，其父类是 java.lang.ClassLoader
- 系统类加载器 (AppClassLoader)
  - 也称作应用类加载器，其父加载器默认为 ExtClassLoader 类的一个实例
  - 负责从 CLASSPATH 或 系统属性 java.class.path 所指定的目录中加载类库
  - 它是用户自定义类加载器的默认父加载器
  - 其父类也是 java.lang.ClassLoader

类加载器

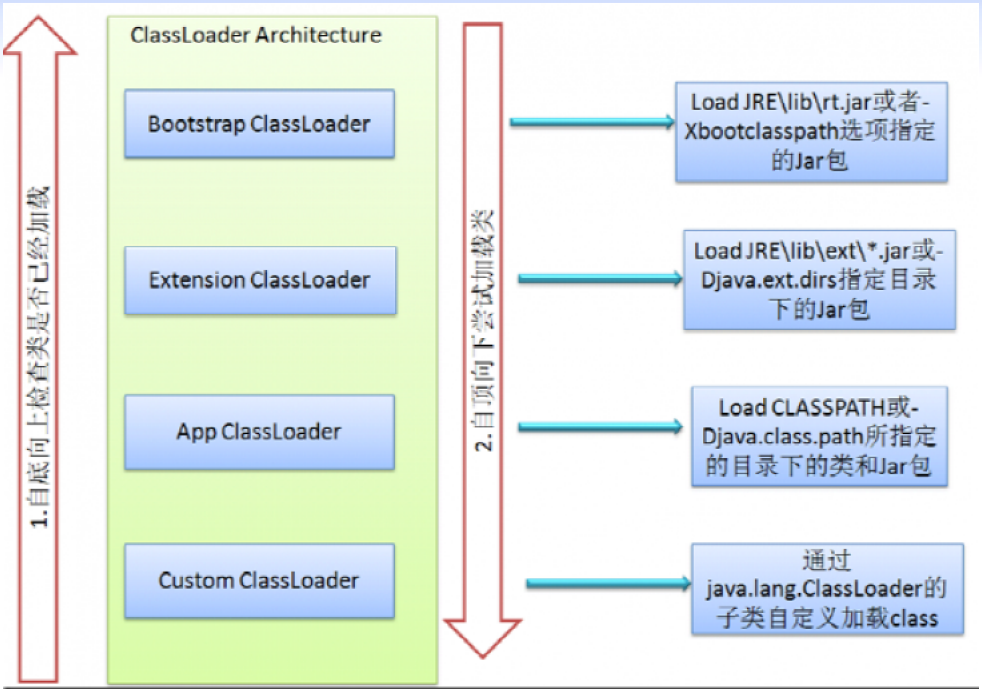
类加载器的层次

- 注意这里的层次关系不是类与类的继承关系



类加载器

各层次的类加载器加载的类



类加载器

获得各层次的类加载器

```
public class ClassLoaderTest1 {
  public static void main(String[] args) {
    //获得当前类的类加载器
    ClassLoader loader = ClassLoaderTest1.class.getClassLoader();
    //逐层向上寻找 父 加载器
    while( loader != null ){
      System.out.println( loader.getClass().getName() );
      loader = loader.getParent();
    }
    System.out.println( loader ); //当某个类加载器没有父加载器时即为根加载器
  }
}
```

输出结果为:

```
sun.misc.Launcher$AppClassLoader
sun.misc.Launcher$ExtClassLoader
null
```

类加载机制

- JVM 的类加载机制主要有以下三种：
  - 全盘负责
    - 当一个类加载器负责加载某个 类 时，该 类 所依赖和引用的其它类也将由当前的类加载器负责载入，除非显式使用了另外一个类加载器来载入
  - 父类委托
    - 先让父加载器加载某个类，只有父加载器无法加载该类时子加载器才加载
  - 缓存机制
    - 使用缓存把所有的被加载过的 类 缓存起来，当程序中需要用到某个类时，类加载器先从缓存中搜寻该 类，如果缓存中不存在该类，系统将读取该类对应的二进制数据并转换成 Class 对象并存入 cache 中
    - 这正是修改源文件后只有重启一个 JVM 才能看到修改后的执行效果的原因

自定义类加载器

- JVM 允许开发者开发自己的类加载器
  - 扩展 java.lang.ClassLoader 类即可
  - 重写其中的方法
- ClassLoader 中的关键方法
  - Class loadClass(String name)
    - 该方法为 ClassLoader 的入口点，根据指定二进制名称来加载类
  - Class findClass(String name)
    - 根据二进制名称来查找类 (一般重写该方法即可)
  - Class defineClass(String name, byte[] b, int off, int len)
    - 根据加载到的二进制数据返回一个 Class 对象

URLClassLoader

- java.net.URLClassLoader 类
  - 是 ClassLoader 的 URL 版实现
  - 该类也是 系统类加载器类 和 扩展类加载器类 的父类
    - 这两个类继承了 该类，该类又继承了 java.security.SecureClassLoader
    - 而 java.security.SecureClassLoader 则继承了 ClassLoader
  - URLClassLoader 功能比较强大
    - 可以从本地文件系统中获取二进制文件来加载类
    - 也可以从远程主机获取二进制文件来加载类
- 常用构造
  - URLClassLoader(URL[] urls)
  - URLClassLoader(URL[] urls , ClassLoader parent)
- 获得实例的静态方法
  - static URLClassLoader newInstance(URL[] urls)
  - static URLClassLoader newInstance(URL[] urls, ClassLoader parent)

问渠那得清如许，为有源头活水来

Oracle Academy