# JVM——Java类加载机制总结

1. **类加载器的组织结构**

类加载器ClassLoader是具有层次结构的，也就是父子关系。其中，Bootstrap是所有类加载器的父亲。

1. Bootstrap ClassLoader：启动类加载器

最顶层的加载类。当运行Java虚拟机时，这个类加载器被创建，它负责加载虚拟机的核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。

**注：**（另外需要注意的是可以通过启动jvm时指定-Xbootclasspath和路径来改变Bootstrap ClassLoader的加载目录。比如java-Xbootclasspath/a:path被指定的文件追加到默认的bootstrap路径中。）

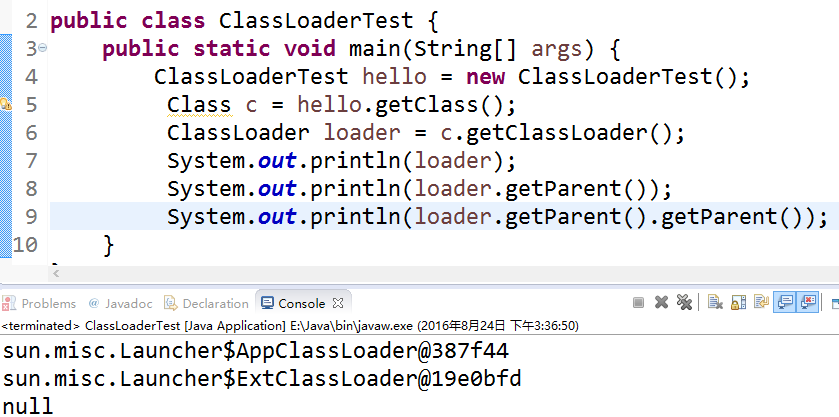
1. Extension ClassLoader：标准扩展类加载器

用于加载除了基本API之外的一些拓展类。（加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。）

1. AppClass Loader：加载应用程序和程序员自定义的类。

**加载顺序：**

1）Bootstrap ClassLoader  
2）Extension ClassLoader  
3）AppClassLoader



每个类加载器都有一个父加载器，比如加载ClassLoaderTest.class是由AppClassLoader完成，那么AppClassLoader也有一个父加载器，怎么样获取呢？很简单，通过getParent方法。

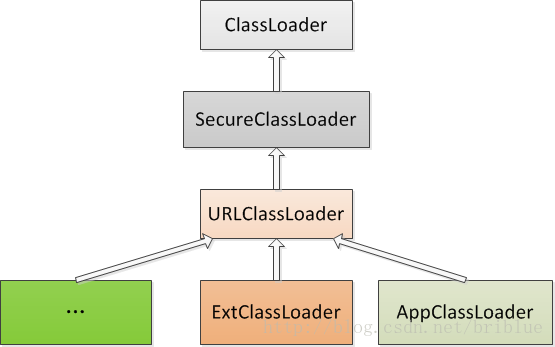
从运行结果可以看出加载器之间的父子关系，ExtClassLoader的父Loader返回了null。原因是BootstrapLoader（启动类加载器）是用C语言实现的，找不到一个确定的返回父Loader的方式。

static class ExtClassLoader extends URLClassLoader{}

static class AppClassLoader extends URLClassLoader{}

可以看见ExtClassLoader和AppClassLoader同样继承自URLClassLoader，但上面一小节代码中，为什么调用AppClassLoader的getParent()代码会得到ExtClassLoader的实例呢？先从URLClassLoader说起，这个类又是什么？

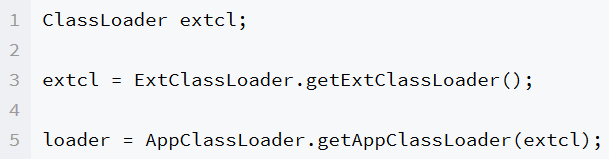
先上一张类的继承关系图



URLClassLoader的源码中并没有找到getParent()方法。这个方法在ClassLoader.java中。

查看源码可知getParent()实际上返回的就是一个ClassLoader对象parent，parent的赋值是在ClassLoader对象的构造方法中，它有两个情况：

1. 由外部类创建ClassLoader时直接指定一个ClassLoader为parent。
2. 由getSystemClassLoader()方法生成，也就是在sun.misc.Laucher通过getClassLoader()获取，也就是AppClassLoader。直白的说，一个ClassLoader创建时如果没有指定parent，那么它的parent默认就是AppClassLoader。



代码说明了AppClassLoader的parent是一个ExtClassLoader实例。

ExtClassLoader并没有直接找到对parent的赋值。它调用了它的父类也就是URLClassLoder的构造方法并传递了3个参数。

**Bootstrap ClassLoader是由C++编写的。**

Bootstrap ClassLoader是由C/C++编写的，它本身是虚拟机的一部分，所以它并不是一个JAVA类，也就是无法在java代码中获取它的引用，JVM启动时通过Bootstrap类加载器加载rt.jar等核心jar包中的class文件，之前的int.class,String.class都是由它加载。然后呢，我们前面已经分析了，JVM初始化sun.misc.Launcher并创建Extension ClassLoader和AppClassLoader实例。并将ExtClassLoader设置为AppClassLoader的父加载器。Bootstrap没有父加载器，但是它却可以作用一个ClassLoader的父加载器。比如ExtClassLoader。这也可以解释之前通过ExtClassLoader的getParent方法获取为Null的现象。具体是什么原因，很快就知道答案了。

**2.类的加载机制**

类被加载到虚拟机内存包括加载、链接、初始化几个阶段。其中链接又细化分为验证、准备、解析。（注：解析阶段在某些情况下可以在初始化阶段之后再开始，这是为了支持Java的运行时绑定。各个阶段的作用整理如下）

**2.1加载阶段**

加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，也可以由用户自定义的类加载器完成，开发人员可以通过定义类加载器去控制字节流的获取方式。

1. 通过类的全名产生对应类的二进制数据流。
2. 将这些二进制数据流转换为方法区的运行时数据结构。
3. 创建代表这个类的java.lang.Class对象。作为方法区这些数据的访问入口。

**2.2链接阶段（实现Java的动态性的重要一步）**

1. 验证：验证阶段的主要目的是确保class文件字节流的正确性，要验证比如class文件格式规范、这个类是否继承了final类，不能把一个父类对象赋值给子类数据类型等等。
2. 准备：准备阶段为方法区中的静态变量分配内存空间。并将其赋值为初始值，所有原始类型的值都为0。如float为0f、int为0、boolean为false、引用类型为null。
3. 解析：解析阶段把符号引用解析为直接引用。

符号引用是一个字符串，它唯一标识一个类、一个字段、一个方法等目标。

而直接引用对于类变量、类方法指的是指向方法区的指针，然后对于实例方法、实例对象来说就是偏移量，比如一个实例方法，子类中方法表中的偏移量和父类是一致的，这个偏移量可以确定某个方法的位置

**2.3初始化**

到了初始化阶段，才是真正执行用户定义的程序代码。在初始化阶段就是执行类构造器方法的过程，工作包括赋值类变量、静态语句块的合并。

//定义在静态语句块之后的变量可以赋值，但不能访问

public class Test{

static{

i=0; //给变量赋值，可以通过编译

System.out.print(i); //这句编译器会提示非法向前引用

}

static int i=1;

}

初始化过程会被触发的条件汇总：

1. 使用new关键字实例化对象、访问一个类的静态字段、静态方法的时候。
2. 对类进行反射调用的时候。
3. 当初始化子类时，如果发现其父类还没有进行过初始化。则进行父类的初始化。

**【关于构造器方法拓展知识】**

（1）类构造器<clinit>()方法与类的构造函数不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

（2）由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。

（3）<clinit>()方法不是必须的，如果一个类中没有静态语句，那么编译器可以不为这个类生成<clinit>()方法。

（4）接口中不能使用静态语句块，和类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

（5）虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步，可能会导致阻塞。

**3.类加载的三种方法**

1. 由new关键字创建一个类的实例。
2. 调用Class.forName()方法，通过反射加载类。
3. 调用某个ClassLoader实例的loadClass()方法。

三者的区别汇总如下：

1. 方法1和方法2都是使用的当前类加载器。方法3是由用户指定的类加载器加载。
2. 方法1是静态方法，2、3是动态加载。
3. 对于两种动态加载，如果程序需要类被初始化，就必须使用Class.forName(name)的方式。

Class.forName(className);

//实际上是调用的是：

Class.forName(className,true,this.getClass().getClassLoader()); //第二个参为true即默认 类需要初始化，初始化会触发目标对象静态块的执行和静态变量的初始化

ClassLoader.loadClass(className);

//实际上调用的是:

ClassLoader.loadClass(name,false); //第二个参数即默认得到的class还没有进行链接， 意味着不进行初始化等系列操作，即静态代码块不会执行