深入探讨 Java 类加载器

JVM设计者把类加载阶段中的“通过'类全名'来获取定义此类的.class文件中的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。

**实现这个动作的代码模块称为“类加载器”**。

**1.类与类加载器**

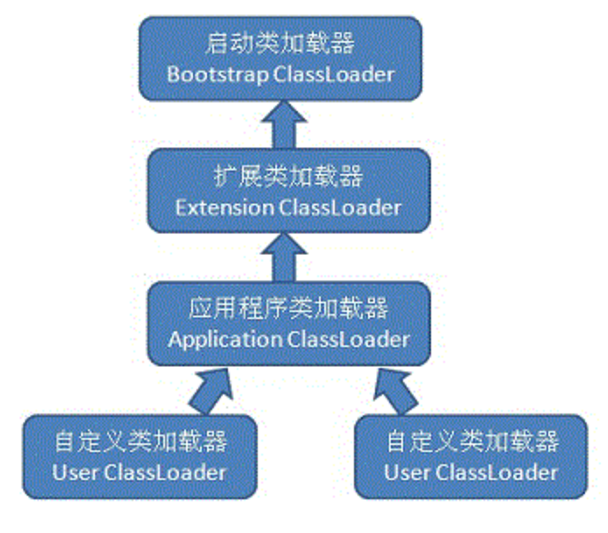
对于任何一个类，都需要由加载它的类加载器和这个类来确立其在JVM中的唯一性。也就是说，两个类来源于同一个Class文件，并且被同一个类加载器加载，这两个类才相等。

**2.双亲委派模型**

从虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），该类加载器使用C++语言实现，属于虚拟机自身的一部分。另外一种就是所有其它的类加载器，这些类加载器是由Java语言实现，独立于JVM外部，并且全部继承自抽象类java.lang.ClassLoader。

从Java开发人员的角度来看，大部分Java程序一般会使用到以下三种系统提供的类加载器：  
1)启动类加载器（Bootstrap ClassLoader）：负责加载JAVA\_HOME\lib目录中并且能被虚拟机识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。**该类加载器无法被Java程序直接引用。**  
2)扩展类加载器（Extension ClassLoader）：该加载器主要是负责加载JAVA\_HOME\lib\，**该加载器可以被开发者直接使用。**  
3)应用程序类加载器（Application ClassLoader）：该类加载器也称为系统类加载器，它负责加载用户类路径（Classpath）上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器**。之前我们在环境变量中配置的classpath就是指定AppClassLoader的类加载路径。**

我们的应用程序都是由这三类加载器互相配合进行加载的，我们也可以加入自己定义的类加载器。这些类加载器之间的关系如下图所示：



如上图所示的类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parent Delegation Model）。该模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过**组合（Composition）**关系来复用父加载器的代码。

**双亲委派模型的工作过程为：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。**

1. 在rt.jar包中的java.lang.ClassLoader类中，我们可以查看类加载实现过程的代码，具体源码如下：**protected** **synchronized** Class loadClass(String name, **boolean** resolve)
2. **throws** ClassNotFoundException {
3. // 首先检查该name指定的class是否有被加载
4. Class c = findLoadedClass(name);
5. **if** (c == **null**) {
6. **try** {
7. **if** (parent != **null**) {
8. // 如果parent不为null，则调用parent的loadClass进行加载
9. c = parent.loadClass(name, **false**);
10. } **else** {
11. // parent为null，则调用BootstrapClassLoader进行加载
12. c = findBootstrapClass0(name);
13. }
14. } **catch** (ClassNotFoundException e) {
15. // 如果仍然无法加载成功，则调用自身的findClass进行加载
16. c = findClass(name);
17. }
18. }
19. **if** (resolve) {
20. resolveClass(c);
21. }
22. **return** c;
23. }

通过上面代码可以看出，双亲委派模型是通过loadClass()方法来实现的，根据代码以及代码中的注释可以很清楚地了解整个过程其实非常简单：先检查是否已经被加载过，如果没有则调用父加载器的loadClass()方法，如果父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载器加载失败，**则先抛出ClassNotFoundException，然后再调用自己的findClass()方法进行加载**。

**3.自定义类加载器**

若要实现自定义类加载器，**只需要继承java.lang.ClassLoader 类，并且重写其findClass()方法即可**。java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例。如tomcat、jboss都会根据j2ee规范自行实现ClassLoader.除此之外，ClassLoader 还负责加载 Java 应用所需的资源，如图像文件和配置文件等，ClassLoader 中与加载类相关的方法如下：

| **方法** | **说明** |
| --- | --- |
| getParent() | 返回该类加载器的父类加载器。 |
| loadClass(String name) | 加载名称为 name的类，返回的结果是 java.lang.Class类的实例。 |
| findClass(String name) | 查找名称为 name的类，返回的结果是 java.lang.Class类的实例。 |
| findLoadedClass(String name) | 查找名称为 name的已经被加载过的类，返回的结果是 java.lang.Class类的实例。 |
| defineClass(String name, byte[] b, int off, int len) | 把字节数组 b中的内容转换成 Java 类，返回的结果是 java.lang.Class类的实例。这个方法被声明为 final的。 |
| resolveClass(Class<?> c) | 链接指定的 Java 类。 |

注意：在JDK1.2之前，类加载尚未引入双亲委派模式，因此实现自定义类加载器时常常重写loadClass方法，提供双亲委派逻辑，从JDK1.2之后，双亲委派模式已经被引入到类加载体系中，自定义类加载器时不需要在自己写双亲委派的逻辑，因此不鼓励重写loadClass方法**，而推荐重写findClass方法。**

## 双亲委派模式优势

采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。

其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出如下异常

**几点思考**

1. Java虚拟机的第一个类加载器是Bootstrap，这个加载器很特殊，它不是Java类，因此它不需要被别人加载，**它嵌套在Java虚拟机内核里面**，也就是JVM启动的时候Bootstrap就已经启动，它是用C++写的二进制代码（不是字节码），它可以去加载别的类。

这也是我们在测试时为什么发现System.class.getClassLoader()结果为null的原因，这并不表示**System这个类没有类加载器**，而是它的加载器比较特殊，是BootstrapClassLoader，由于它不是Java类，因此获得它的引用肯定返回null。

1. 委托机制具体含义   
   当Java虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？
   * 首先当前线程的类加载器去加载线程中的第一个类（假设为类A）。   
     注：当前线程的类加载器可以通过Thread类的getContextClassLoader()获得，也可以通过setContextClassLoader()自己设置类加载器。
   * 如果类A中引用了类B，Java虚拟机将使用加载类A的类加载器去加载类B。
   * 还可以直接调用ClassLoader.loadClass()方法来指定某个类加载器去加载某个类。
2. 委托机制的意义 — 防止内存中出现多份同样的字节码   
   比如两个类A和类B都要加载System类：
   * 如果不用委托而是自己加载自己的，那么类A就会加载一份System字节码，然后类B又会加载一份System字节码，这样内存中就出现了两份System字节码。
   * 如果使用委托机制，会递归的向父类查找，也就是首选用Bootstrap尝试加载，如果找不到再向下。这里的System就能在Bootstrap中找到然后加载，如果此时类B也要加载System，也从Bootstrap开始，此时Bootstrap发现已经加载过了System那么直接返回内存中的System即可而不需要重新加载，这样内存中就只有一份System的字节码了。