Java 类加载器

一、概述

虚拟机设计团队把类加载阶段中的"通过一个类的全限定名来获取描述此类的二进制字节流"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".

类加载器可以说是 Java 语言的一项创新,也是 Java 语言流行的重要原因之一，它最初是为了满足 Java Applet 的需求而开发出来的. 虽然目前 Java Applet 技术基本上已经"死掉"，但类加载器却在类层次划分，OSGI，热部署，代码加密等领域大放异采用，成为了 Java 技术体系中一块重要的基石，可谓失之桑榆，收之东隅.

二、类与类加载器

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远远不限制于类加载阶段. 对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性,每一个类加载器，都拥有一个独立的类名称空间. 这句话可以表达得更通俗一些 ：比较两个类是否"相等"，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等.

这里所指的"相等"，包括代表类的 Class 对象的 equals()方法，isAssignableFrom()方法，isInstance()方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况. 如果没有到类加载器的影响，在某些情况下可能会产生具有迷惑性的结果，下面代码演示了不同的类加载器对 instanceof 关键字运算的结果的影响:

```
package com.atguigu.test;
/**
    类加载器与 instanceof 关键字演示
*/
public class ClassLoaderTest {

    public static void main(String[] args) throws Exception {
        ClassLoader myLoader = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name)
                                        throws ClassNotFoundException {
                try {
                    String fileName = name.substring(name.lastIndexOf(".") + 1)
+ ".class";
```

```
                    InputStream is = null;

                    is = getClass().getResourceAsStream(fileName);
                    if (is == null) {
                        return super.loadClass(name);
                    }
                    byte[] b = new byte[is.available()];
                    is.read(b);
                    return defineClass(name, b, 0, b.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException(name);
                }
            }
        };

        Class clazz = null;
        Object obj = null;
        clazz =myLoader.loadClass("com.atguigu.test.ClassLoaderTest");
        obj = clazz.newInstance();
        System.out.println(clazz);
        System.out.println(obj instanceof com.atguigu.test.ClassLoaderTest);
    }

}
```

运行结果:

```
class com.atguigu.test.ClassLoaderTest
false
```

代码中构造了一个简单的类加载器，尽管很简单，但是对于这个演示来说还是够用了，它可以加载与自己在同一路径下的 class 文件. 我们使用这个类加载器去加载了一个名为"com.atguigu.test.ClassLoaderTest"的类，并实例化了这个类的对象. 两行输出结果中，从第一句可以看出，这个对象确实是类 com.atguigu.test.ClassLoaderTest 实例化出来的对象，但从第二句可以发现，这个对象与类 com.atguigu.test.ClassLoaderTest 所属的类型检查的时候却返回了 false，这是因为虚拟机存在了两个 ClassLoaderTest 类，一个是由系统应用程序类加载器加载的，另外一个是由我们自定义的类加载器加载的，虽然都来自于同一个 class 文件，但依然是两个独立的类，做对象所属类型检查时结果自然为 false.

三、双亲委派模型：

从 Java 虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器(Bootstrap ClassLoader)，这个类加载器使用 C++语言实现(这里只限于 HotSpot,

像 MRP, Maxine 等虚拟机，整个虚拟机本身都是由 Java 编写的，自然 Bootstrap ClassLoader 也是由 Java 语言而不是 C++实现的，退一步讲，除了 HotSpot 以外的其他两个高性能虚拟机 JRockit 和 J9 都有一个代表 Bootstrap ClassLoader 的 java 类存在，但是关键方法的实现仍然是使用 JNI 回调到 C，不是 C++的实现上，这个 Bootstrap ClassLoader 的实例也无法被用户获取到.)，是虚拟机自身的一部分;另一种就是所有其他的类加载器，这些类加载器都由 java 语言实现,独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader.

从 java 开发人员的角度来看，类加载器还可以划分得更细致一些，绝大多数 java 程序都会使用到以下 3 种系统提供的类加载器.

1) 启动类加载器(Bootstrap ClassLoader)：这个类加载器负责将存放在 JAVA_HOME\lib 目录中的,或者被-Xbootclasspath 参数所指定的路径中的,并且是虚拟机识别的(仅按照文件名识别，如 rt.jar 名字不符合的类库即使放在 lib 目录中也不会被加载) 类库加载到虚拟机内存中. 启动类加载器无法被 java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器,那直接使用 null 代替即可. 下面的代码就是 java.lang.ClassLoader.getClassLoader()方法的代码片段:

```
public ClassLoader getClassLoader() {
    ClassLoader cl = getClassLoader();
    if (cl == null)
        return null;
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        ClassLoader ccl = ClassLoader.getCallerClassLoader();
        if (ccl != null && ccl != cl && !cl.isAncestor(ccl)) {

sm.checkPermission(SecurityConstants.GET_CLASSLOADER_PERMISSION);
        }
    }
    return cl;
}
```

2) 扩展类加载器(Extension ClassLoader)：这个加载器由 sun.misc.Launcher$ExtClassLoader 实现，它负责加载 JAVA_HOME/lib/ext 目录中的,或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器.

3) 应用程序类加载器(Application ClassLoader)： 这个类加载器由 sun.misc.Launcher$AppClassLoader 实现. 由于这个类加载器是 ClassLoader 中的 getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器. 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这个类加载器,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.

我们的应用程序都是由这 3 种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器. 这些类加载器之间的关系是

启动类加载器
扩展类加载器
应用程序类加载器
自定义类加载器 1，自定义类加载器 2 ....

上面的类是下面的类的父类. 这种关系,称为类加载器的双亲委派模型(Parents Delegation Model). 双亲委派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. 这里类加载器之间的父子关系一般不会以继承(Inheritance)的关系来实现，而是都使用组合(Composition)关系来复用父加载器的代码.

类加载器的双亲委派模型在 JDK1.2 期间被引入并被广泛应用于之后几乎所有的 java 程序中,但它并非不是一个强制性的约束模型,而是 java 设计者推荐给开发者的一种类加载器实现方式.

双亲委派模型的工作过程是 ：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，每一个层次的类加载器都是如此，因此所有的加载请求最都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求(它的搜索范围内没有找到所需的类)时，子加载器才会尝试自己去加载.

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 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. 例如类 Java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，由各个类加载器自行去加载的话,如果用户自己编写了一个称为 java.lang.Object 的类,并放在程序的 classpath 中,那系统中将会出现多个不同的 Object 类, java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱. 如果我们尝试编写一个与 rt.jar 类库中已有类重名的 java 类，将会发现可以正常编译,但永远无法被加载运行. 即使自定义了自己的类加载器，强行用 defineClass()方法去加载一个以"java.lang"开头的类也不会成功. 如果尝试这样做的话，将会收到一个由虚拟机自己抛出异常"java.lang.SecurityException:Prohibited package name:java.lang"

双亲委派模型对于保证 java 程序的稳定动作很重要，但它的实现却非常简单,实现双亲委派的代码都集中在 java.lang.ClassLoader 的 loadClass()方法之中，如下所示，逻辑清晰，先检查是否已经被加载过,若没有加载则调用父加载器的 loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器. 如果父类加载失败,抛出 ClassNotFoundException 异常后，再调用自己的 findClass()方法进行加载.

代码如下:

```
protected synchronized Class<?> loadClass(String name, boolean resolve)
                                        throws ClassNotFoundException {
    // 首先，检查请求的类是否已经被加载过了
    Class c = findLoadedClass(name);
```

```
        if (c == null) {
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // 如果父类加载器抛出异常
                // 说明父类加载器无法完成加载请求
            }
            if (c == null) {
                // 在父类加载器无法加载的时候，再调用本身的 findClass 方法
                c = findClas(name);
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }

}
```