链滴

虚拟机类的加载机制 (二)

作者：dddygin
原文链接：https://ld246.com/article/1579137875283
来源网站：链滴

虚拟机类的加载机制(二)

类加载器

类加载器虽然只用于实现类的加载动作。但是对于任意一个类，都需要由加载它的类加载器和这个类身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立名称空间（即类和加载器成唯一性）。**其中比较两个类是否“相等”，只有在这两个类来源于同个Class文件而且被同一个虚机加载才能说相等。**

这里的“相等”，包括代表类的Class对象的equals()方法，isAssignableFrom（）、isInstance()返的结果。

```
package dddy.gin.jvm.chapter07;

import java.io.IOException;
import java.io.InputStream;

/**
 * 不同类加载器对instanceof关键字的影响
 * @author gin
 */
public class ClassLoader01 {
    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 = 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);
            }
        }
    };
    Object obj = myLoader
            .loadClass("dddy.gin.jvm.chapter07.ClassLoader01")
            .getDeclaredConstructor()
            .newInstance();
    System.out.println(obj.getClass());
    System.out.println(obj instanceof dddy.gin.jvm.chapter07.ClassLoader01);
  }
}
```

运行结果为：

```
class dddy.gin.jvm.chapter07.ClassLoader01
false
```

从结果可以看出我们用myLoder加载器加载的类和系统加载器加载的类名字虽然相同但是加载器不同是不相等的。

双亲委派模型

在Java中主要可以分为两种不同的类加载器：第一种是**启动类加载器（Bootstrap ClassLoader）**，个类加载器使用C++语言实现；**第二种是所有其他的类加载器**，这些类加载器都有Java语言实现，独于虚拟机外部，并且继承于java.lang.ClassLoader。

如下图是类加载器双亲委派模型：

其中**启动类加载器（Bootstrap ClassLoader）**：这个类加载器负责将存放在<
AVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径
的，**并且是虚拟机识别的类库**（比如可以识别rt.jar；如果名字不符的类库即使放在lib目录也不会加载加载到虚拟内存中。

扩展类加载器（Extension ClassLoader）：这个加载器由sun.misc.Launcher$ExtClassLoader实，负责加载<JAVA_HOME>\
ib\ext目录中的，或者被java.ext.dirs系统变量指定的路径中的所有类库，开发者可以直接使用扩展载器。

应用程序类加载器（Application ClassLoader）：这个类加载器由sun.misc.Launcher.$App-Classoader实现，它负责加载用户路径ClassPath上指定的类库，开发者可以直接使用这个类加载器，如果用没有自定义类加载器，一般情况下就是这个为程序的默认类加载器。

上图展示的是双亲委派模型（Parents Delegation Model）。**双亲委派模型**要求除了顶层的启动类载器外，其余的类加载器都应当有自己的父类加载器。这里的“父子关系”一般不会以继承关系来实，而都是使用“组合关系”复用父加载器的代码。

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，因此所有的加载请最终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（它的搜范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

为什么需要双亲委派模型，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t.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载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，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ClssPath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，应程序也将会变得一片混乱。

破坏双亲委派模型

因为双亲委派模型不是一个强制性的约束模型，是java设计者推荐给开发者的类加载实现方式。其中三次比较大规模的“被破坏”的情况

1. **JDK1.2发布之前**，因为双亲委派模型是JDK1.2之后才被引入的，而类加载器和抽象类java.lang.ClasLoader则JDK1.0就已经存在，为了向前兼容。在JDK1.2之后java.lang.ClassLoader引入了findClas（）方法，来解决向前兼容的问题。

2. **JNDI破坏双亲委派模型** **， 由于越基础的类由越上层的加载器进行加载。

若加载的基础类中需要回调用户代码，而这时顶层的类加载器无法识别这些用户代码，怎么办呢？这就需要破坏双亲委派模型了。 **

- **JNDI破坏双亲委派模型**

JNDI是Java标准服务，它的代码由启动类加载器去加载。但是JNDI需要回调独立厂商实现的代码，类加载器无法识别这些回调代码（SPI）。

为了解决这个问题，引入了一个**线程上下文类加载器。** 可通过Thread.setContextClassLoader()设置。

利用线程上下文类加载器去加载所需要的SPI代码，即父类加载器请求子类加载器去完成类加载的过，而破坏了双亲委派模型

- **Spring破坏双亲委派模型**

Spring要对用户程序进行组织和管理，而用户程序一般放在WEB-INF目录下，由WebAppClassLoade类加载器加载，而Spring由Common类加载器或Shared类加载器加载。

那么Spring是如何访问WEB-INF下的用户程序呢?

使用**线程上下文类加载器。** Spring加载类所用的classLoader都是通过Thread.currentThread().getCntextClassLoader()获取的。当线程创建时会默认创建一个AppClassLoader类加载器（对应Tomcat的WebAppclassLoader类加载器）： setContextClassLoader(AppClassLoader)。

利用这个来加载用户程序。即任何一个线程都可通过getContextClassLoader()获取到WebAppclassLader。

3. **追求程序的动态性**，如代码热替换（HotSwap）、模块热部署（Hot Deployment），在OSGI环下，类加载器不再是双亲委派模型中的树型结构，而是进一步发展成为更加复杂的网状结构，收到类载请求时不再是双亲委派模型。

参考：

1. https://blog.csdn.net/luzhensmart/article/details/82665122
2. 周志明. 深入理解Java虚拟机：JVM高级特性与最佳实践 . 机械工业出版社.