一、三种类加载器详解

类加载器通常可以分为三种：

启动类加载器(BootstrapClassLoader)

扩展类加载器(ExtClassLoader)

应用程序类加载器(AppClassLoader）

1、启动类加载器

启动类加载器是由c++实现的，是虚拟机的一部分，主要负责加载jvm自身需要的类，即负责加载JAVAHOME下的核心类库。打印下启动类加载器的加载路径，代码如下：

URLClassPath path = Launcher.*getBootstrapClassPath*();  
for(URL url : path.getURLs()){  
 *log*.info(url.getPath());  
}

输出结果

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/resources.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/rt.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/sunrsasign.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/jsse.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/jce.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/charsets.jar

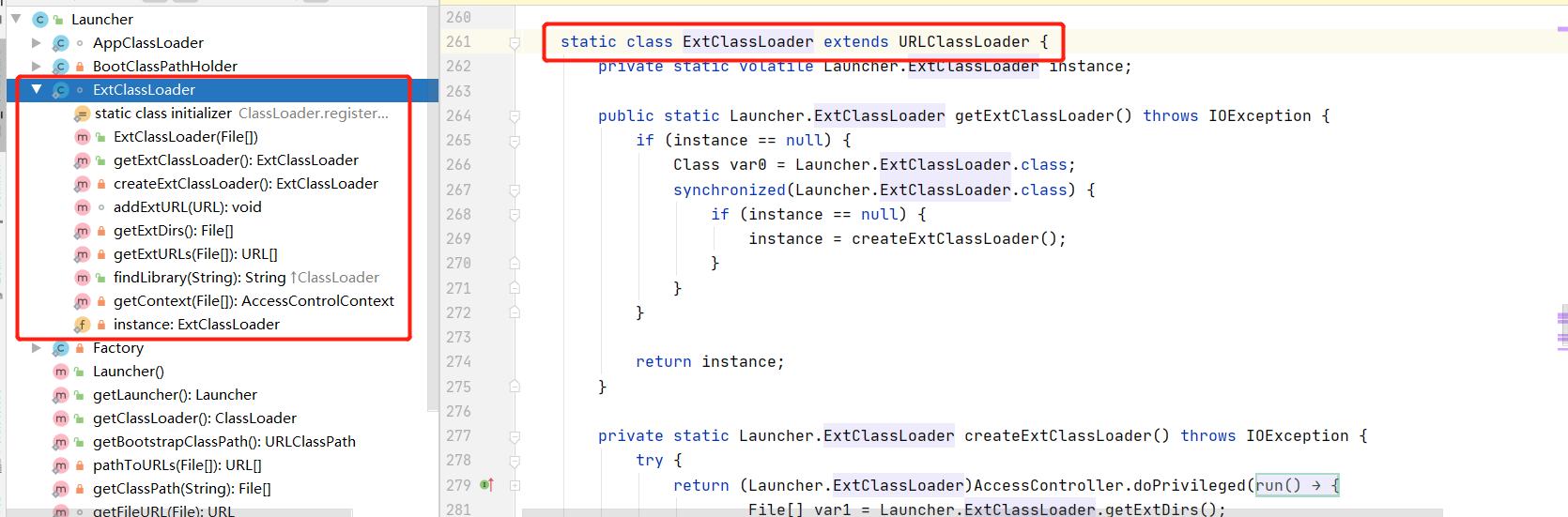
/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/jfr.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/classes

因是c++直接实现的启动类加载器，所以这是唯一没有继承java中ClassLoder的类加载器。扩展类加载器和应用程序类加载器都继承了ClassLoder

2、扩展类加载器

扩展类加载器是由java实现的，具体实现类是sun.misc. Launcher$ExtClassLoader，如下图：



打印一下扩展类加载器的实例化对象来确认下扩展类加载的实现

URLClassLoader extClassLoader = (URLClassLoader)ClassLoader.*getSystemClassLoader*().getParent();  
*log*.info (extClassLoader);

输出结果：sun.misc.Launcher$ExtClassLoader@1753acfe

打印下扩展类加载器的加载路径，代码如下：

URLClassLoader extClassLoader = (URLClassLoader)ClassLoader.*getSystemClassLoader*().getParent();  
for(URL url : extClassLoader.getURLs()) {  
 *log*.info(url.getPath());  
}

输出结果：

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/access-bridge-64.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/cldrdata.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/dnsns.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/jaccess.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/jfxrt.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/localedata.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/nashorn.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/sunec.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/sunjce\_provider.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/sunmscapi.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/sunpkcs11.jar

/C:/Program%20Files/Java/jdk1.8.0\_201/jre/lib/ext/zipfs.jar

扩展类记载器加载路径的代码在sun.misc. Launcher$ExtClassLoader的getExtDirs函数中，代码如下：

private static File[] getExtDirs() {  
 String var0 = System.getProperty("java.ext.dirs");  
 File[] var1;  
 if (var0 != null) {  
 StringTokenizer var2 = new StringTokenizer(var0, File.pathSeparator);  
 int var3 = var2.countTokens();  
 var1 = new File[var3];  
 for(int var4 = 0; var4 < var3; ++var4) {  
 var1[var4] = new File(var2.nextToken());  
 }  
 } else {  
 var1 = new File[0];  
 }  
 return var1;  
}

java.ext.dirs这个目录下的所有jar包都被加载了，那问题来了，扩展类加载器是负责加载这个路径下的所有jar包，还是只负责规定好的jar包呢，让我们试一下，在这个路径下放一个其他的jar包，我放了一个自己的的工具类jar包mg-common-1.0.0.jar到这个目录下，再次执行上边的代码，发现输出结果中包含了新加的jar包。所以**扩展类加载器负责加载java.ext.dirs对应目录下的所有jar包。**

**注意：扩展类加载器的父级是启动类加载器，不过通过扩展类加载器的getParent()方法取到的返回结果为null，这是因为启动类加载器不是java实现的。**

3、应用程序类加载器

应用程序类记载器同样也是java代码实现的，类加载器是sun.misc. Launcher$AppClassLoader

的实例对象，如下图：



可以通过如下代码来确认

URLClassLoader appClassLoader = (URLClassLoader)ClassLoader.*getSystemClassLoader*();  
System.*out*.println(appClassLoader);

输出结果：sun.misc.Launcher$AppClassLoader@18b4aac2

输出下系统默认的应用程序类加载的加载路径，如下：

URLClassLoader appClassLoader = (URLClassLoader)ClassLoader.*getSystemClassLoader*();  
for(URL url : appClassLoader.getURLs()) {  
 *log*.info(url.getPath());  
}

输出结果太多，就不列出来了，感兴趣的可以试下。

获取应用程序加载器的代码中，我们获取的ClassLoader.*getSystemClassLoader*()这个方法返回的类加载器，这个函数的加载器是从哪里来得呢，进到函数中可以看到执行了initSystemClassLoader函数，这个函数获取sun.misc. Launcher这个类的加载器

private static synchronized void initSystemClassLoader() {  
 if (!*sclSet*) {  
 if (*scl* != null)  
 throw new IllegalStateException("recursive invocation");  
 sun.misc.Launcher l = sun.misc.Launcher.*getLauncher*();  
 if (l != null) {  
 Throwable oops = null;  
 *scl* = l.getClassLoader();  
 try {  
 *scl* = AccessController.*doPrivileged*(  
 new SystemClassLoaderAction(*scl*));  
 } catch (PrivilegedActionException pae) {  
 oops = pae.getCause();  
 if (oops instanceof InvocationTargetException) {  
 oops = oops.getCause();  
 }  
 }  
 if (oops != null) {  
 if (oops instanceof Error) {  
 throw (Error) oops;  
 } else {  
 *// wrap the exception* throw new Error(oops);  
 }  
 }  
 }  
 *sclSet* = true;  
 }  
}

应用程序的类加载器的父类加载器是扩展类加载器，可以调用getParent（）函数获取到扩展类加载器。

刚才说的三种类加载器说完了，那自定义的类加载器怎么算呢，自定义类加载器既然是自定义的，加载范围也是自定义的了，不过默认自定义类记载器的父级是通过刚才我们使用过的getSystemClassLoader函数获取的，代码如下**ClassLoader() {this(checkCreateClassLoader(), getSystemClassLoader());}**

现在三种类加载器准备好了，现在有一个类如果三种类加载器都加载一次，会有什么问题，怎么解决的呢，接来下让我们一起来看下双亲委派模式吧。

三、双亲委派模式

有这么多的类加载器，如果多个加载器加载了同一个class对象会不会有问题啊，先在E:/cltmp/放一个MgDemoSample 类的class文件，执行下如下代码

URL url = new URL("file:/E:/cltmp/");  
 URLClassLoader classLoader1 = new URLClassLoader(new URL[]{url});  
 Class cl1 = classLoader1.loadClass("MgDemoSample");  
 URLClassLoader classLoader2 = new URLClassLoader(new URL[]{url});  
 Class cl2 = classLoader2.loadClass("MgDemoSample");System.*out*.println(cl1);  
 System.*out*.println(cl2);  
 System.*out*.println(cl1.equals(cl2));

输出结果：false

同一个类被不通的类加载器，会被识别为两个不同的类对象（类的唯一标识是 类加载器+类名），所以类加载重复加载同一个类不仅仅是资源的浪费，还会引起java核心代码库的安全问题。可以想象一下在系统中加载了多个被重写过的String类会有什么结果。那怎么解决这个问题呢，这就需要用到**双亲委派模式**这个加载机制了。启动类加载器、扩展类加载器、应用程序类加载器和自定义类加载器的关系如下图：



在双亲委派模式下怎么避免类的重复加载呢，**在加载class对象时，类加载器会首先去尝试父级类加载器中加载，如果父级类加载器加器反馈无法加载这个类，类加载器才会尝试自己去加载**。类加载器都有自己的父级，在尝试让父级类加载器加载这个操作是一个**递归的过程**，一直会找到扩展类加载器这一级，然后扩展类记载器会找到启动类记载器尝试加载。可以通过ClassLoder这个类的loadClass这个函数来了解这个加载过程，代码如下：

protected Class<?> loadClass(String name, boolean resolve)  
 throws ClassNotFoundException  
{  
 synchronized (getClassLoadingLock(name)) {  
 *// First, check if the class has already been loaded* Class<?> c = findLoadedClass(name);  
 if (c == null) {  
 long t0 = System.*nanoTime*();  
 try {  
 if (parent != null) {  
 c = parent.loadClass(name, false);  
 } else {  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 *// ClassNotFoundException thrown if class not found  
 // from the non-null parent class loader* }  
  
 if (c == null) {  
 *// If still not found, then invoke findClass in order  
 // to find the class.* long t1 = System.*nanoTime*();  
 c = findClass(name);  
  
 *// this is the defining class loader; record the stats* sun.misc.PerfCounter.*getParentDelegationTime*().addTime(t1 - t0);  
 sun.misc.PerfCounter.*getFindClassTime*().addElapsedTimeFrom(t1);  
 sun.misc.PerfCounter.*getFindClasses*().increment();  
 }  
 }  
 if (resolve) {  
 resolveClass(c);  
 }  
 return c;  
 }  
}

在上一个章节中已经提过自定义类加载默认使用系统类加载器，所以自定义类加载也是在双亲委派模式这个加载流程中。

到这里类加载机制可以告一段落了，有问题或者建议可以留言。