原 [置顶] 一看你就懂，超详细java中的ClassLoader详解

标签： jvm java classloader 类加载器 双亲委托

2017-02-10 19:26 6535人阅读 评论(28) 收藏 举报

分类： Android笔记（22）

目录(?) [+]

本篇文章已授权微信公众号 guolin_blog （郭霖）独家发布

ClassLoader翻译过来就是类加载器，普通的Java开发者其实用到的不多，但对于某些框架开发者来说却非常常见。理解ClassLoader的加载机制，也有利于我们编写出更高效的代码。ClassLoader的具体作用就是将class文件加载到jvm虚拟机中去，程序就可以正确运行了。但是，jvm启动的时候，并不会一次性加载所有的class文件，而是根据需要去动态加载。想想也是的，一次性加载那么多jar包那么多class，那内存不崩溃。本文的目的也是学习ClassLoader这种加载机制。

备注：本文篇幅比较长，但内容简单，大家不要恐慌，安静地耐心翻阅就是

Class文件的认识

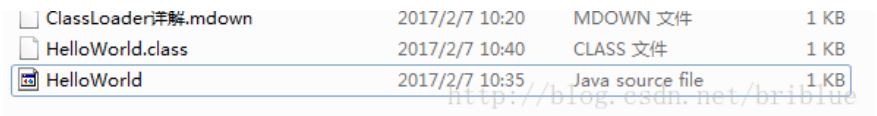
我们都知道在Java中程序是运行在虚拟机中，我们平常用文本编辑器或者是IDE编写的程序都是.java格式的文件，这是最基础的源码，但这类文件是不能直接运行的。如我们编写一个简单的程序HelloWorld.java

```
1 public class HelloWorld{
2
3     public static void main(String[] args){
4         System.out.println("Hello world!");
5     }
6 }
```

如图：

然后，我们需要在命令行中进行java文件的编译

```
1 javac HelloWorld.java
```



可以看到目录下生成了.class文件

我们再从命令行中执行命令：

```
1 java HelloWorld
```

上面是基本代码示例，是所有入门JAVA语言时都学过的东西，这里重新拿出来是想让大家将焦点回到class文件上，class文件是字节码格式文件，java虚拟机并不能直接识别我们平常编写的.java源文件，所以需要令转换成.class文件。另外，如果用C或者Python编写的程序正确转换成.class文件后，java虚拟机也行的。更多信息大家可以参考这篇。

了解了.class文件后，我们再来思考下，我们平常在Eclipse中编写的java程序是如何运行的，也就是我们自己编写的各种类是如何被加载到jvm(java虚拟机)中去的。

你还记得java环境变量吗？

初学java的时候，最害怕的就是下载JDK后要配置环境变量了，关键是当时不理解，所以战战兢兢地照着书籍上或者是网络上的介绍进行操作。然后下次再弄的时候，又忘记了而且是必忘。当时，心里的想法很气愤的，想着是--这东西一点也不人性化，为什么非要自己配置环境变量呢？太不照顾菜鸟和新手了，很多菜鸟就是因为卡在环境变量的配置上，遭受了太多的挫败感。

因为我是在Windows下编程的，所以只讲Window平台上的环境变量，主要有3个：**JAVA_HOME**、**PATH**、**CLASSPATH**。

JAVA_HOME

指的是你JDK安装的位置，一般默认安装在C盘，如

```
1 C:\Program Files\Java\jdk1.8.0_91
```

PATH

将程序路径包含在PATH当中后，在命令行窗口就可以直接键入它的名字了，而不再需要键入它的全路径,比如上面代码中我用的到 `javac` 和 `java` 两个命令。
一般的

```
1 PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%PATH%;
```

也就是在原来的PATH路径上添加JDK目录下的bin目录和jre目录的bin.

CLASSPATH

```
1 CLASSPATH=.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar
```

一看就是指向jar包路径。
需要注意的是前面的 `.;` ，`.` 代表当前目录。

环境变量的设置与查看

设置可以右击我的电脑，然后点击属性，再点击高级，然后点击环境变量，具体不明白的自行查阅文档。

查看的话可以打开命令行窗口

```
1
2 echo %JAVA_HOME%
3
```

```
4 echo %PATH%
5
6 echo %CLASSPATH%
```

好了，扯远了，知道了环境变量，特别是CLASSPATH时，我们进入今天的主题Classloader.

JAVA类加载流程

Java语言系统自带有三个类加载器:
- **Bootstrap ClassLoader** 最顶层的加载类，主要加载核心类库，%JRE_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。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。我们可以打开我的电脑，在上面的目录下查看，看看这些jar包是不是存在于这个目录。
- **Extention ClassLoader** 扩展的类加载器，加载目录%JRE_HOME%\lib\ext目录下的jar包和clas
加载 `-D java.ext.dirs` 选项指定的目录。
- **Appclass Loader也称为SystemAppClass** 加载当前应用的classpath的所有类。

我们上面简单介绍了3个ClassLoader。说明了它们加载的路径。并且还提到了 `-Xbootclasspath` 和 `-D java.ext.dirs` 这两个虚拟机参数选项。

加载顺序？

我们看到了系统的3个类加载器，但我们可能不知道具体哪个先行呢？
我可以先告诉你答案
1. Bootstrap CLassloder
2. Extention ClassLoader
3. AppClassLoader

为了更好的理解，我们可以查看源码。
看sun.misc.Launcher,它是一个java虚拟机的入口应用。

```
1  public class Launcher {
2      private static Launcher launcher = new Launcher();
3      private static String bootClassPath =
4          System.getProperty("sun.boot.class.path");
5
6      public static Launcher getLauncher() {
7          return launcher;
8      }
9
10     private ClassLoader loader;
11
12     public Launcher() {
13         // Create the extension class loader
14         ClassLoader extcl;
15         try {
16             extcl = ExtClassLoader.getExtClassLoader();
17         } catch (IOException e) {
18             throw new InternalError(
19                 "Could not create extension class loader", e);
20         }
21
22         // Now create the class loader to use to launch the application
23         try {
24             loader = AppClassLoader.getAppClassLoader(extcl);
25         } catch (IOException e) {
26             throw new InternalError(
27                 "Could not create application class loader", e);
28         }
29
30         //设置AppClassLoader为线程上下文类加载器，这个文章后面部分讲解
31         Thread.currentThread().setContextClassLoader(loader);
32     }
33
34     /*
```

```
     * Returns the class loader used to launch the main application.
     */
    public ClassLoader getClassLoader() {
        return loader;
    }
    /*
     * The class loader used for loading installed extensions.
     */
    static class ExtClassLoader extends URLClassLoader {}

/**
     * The class loader used for loading from java.class.path.
     * runs in a restricted security context.
     */
    static class AppClassLoader extends URLClassLoader {}
```


源码有精简，我们可以得到相关的信息。

1. Launcher初始化了ExtClassLoader和AppClassLoader。

2. Launcher中并没有看见BootstrapClassLoader，但通过`System.getProperty("sun.boot.class.path")`得到了字符串`bootClassPath`,这个应该就是BootstrapClassLoader加载的jar包路径。

我们可以先代码**测试**一下`sun.boot.class.path`是什么内容。

```
System.out.println(System.getProperty("sun.boot.class.path"));
```

得到的结果是：

```
C:\Program Files\Java\jre1.8.0_91\lib\resources.jar;
C:\Program Files\Java\jre1.8.0_91\lib\rt.jar;
C:\Program Files\Java\jre1.8.0_91\lib\sunrsasign.jar;
C:\Program Files\Java\jre1.8.0_91\lib\jsse.jar;
C:\Program Files\Java\jre1.8.0_91\lib\jce.jar;
C:\Program Files\Java\jre1.8.0_91\lib\charsets.jar;
C:\Program Files\Java\jre1.8.0_91\lib\jfr.jar;
C:\Program Files\Java\jre1.8.0_91\classes
```

可以看到，这些全是JRE目录下的jar包或者是class文件。

ExtClassLoader源码

如果你有足够的好奇心，你应该会对它的源码感兴趣

```
/*
     * The class loader used for loading installed extensions.
     */
    static class ExtClassLoader extends URLClassLoader {

        static {
            ClassLoader.registerAsParallelCapable();
        }

        /**
         * create an ExtClassLoader. The ExtClassLoader is created
         * within a context that limits which files it can read
         */
        public static ExtClassLoader getExtClassLoader() throws IOException
        {
            final File[] dirs = getExtDirs();

            try {
                // Prior implementations of this doPrivileged() block supplied
                // aa synthesized ACC via a call to the private method
                // ExtClassLoader.getContext().

                return AccessController.doPrivileged(
                    new PrivilegedExceptionAction<ExtClassLoader>() {
                        public ExtClassLoader run() throws IOException {
                            int len = dirs.length;
                            for (int i = 0; i < len; i++) {
```

```
                                MetaIndex.registerDirectory(dirs[i]);
                            }
                            return new ExtClassLoader(dirs);
                        }
                    });
            } catch (java.security.PrivilegedActionException e) {
                throw (IOException) e.getException();
            }
        }

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

......
    }
```


我们先前的内容有说过，可以指定 `-D java.ext.dirs` 参数来添加和改变ExtClassLoader的加载路径。这里我们通过可以编写测试代码。

```
System.out.println(System.getProperty("java.ext.dirs"));
```

结果如下：

```
C:\Program Files\Java\jre1.8.0_91\lib\ext;C:\Windows\Sun\Java\lib\ext
```

AppClassLoader源码

```
/**
     * The class loader used for loading from java.class.path.
     * runs in a restricted security context.
     */
    static class AppClassLoader extends URLClassLoader {


        public static ClassLoader getAppClassLoader(final ClassLoader extcl)
            throws IOException
        {
            final String s = System.getProperty("java.class.path");
            final File[] path = (s == null) ? new File[0] : getClassPath(s);


            return AccessController.doPrivileged(
                new PrivilegedAction<AppClassLoader>() {
                    public AppClassLoader run() {
                    URL[] urls =
                        (s == null) ? new URL[0] : pathToURLs(path);
                    return new AppClassLoader(urls, extcl);
                    }
                });
        }

        ......
    }
```

可以看到AppClassLoader加载的就是 `java.class.path` 下的路径。我们同样打印它的值。

```
System.out.println(System.getProperty("java.class.path"));
```

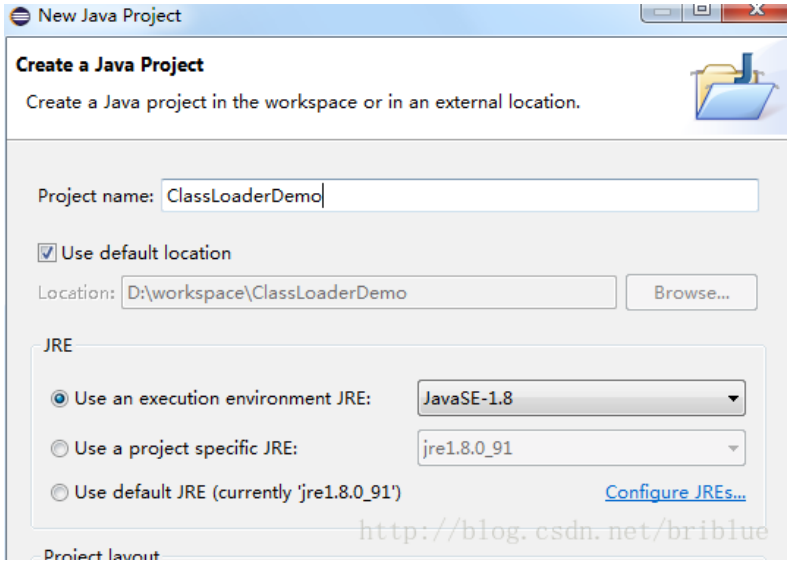
结果：

```
D:\workspace\ClassLoaderDemo\bin
```

这个路径其实就是当前java工程目录bin，里面存放的是编译生成的class文件。

好了，自此我们已经知道了BootstrapClassLoader、ExtClassLoader、AppClassLoader实际是查阅相应的环境属性 `sun.boot.class.path` 、 `java.ext.dirs` 和 `java.class.path` 来加载资源文件的。

接下来我们探讨它们的加载顺序，我们先用Eclipse建立一个java工程。



然后创建一个 `Test.java` 文件。

```
public class Test{}
```

然后，编写一个ClassLoaderTest.java文件。

```

public class ClassLoaderTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        ClassLoader cl = Test.class.getClassLoader();

        System.out.println("ClassLoader is:"+cl.toString());

    }

}
```

我们获取到了Test.class文件的类加载器，然后打印出来。结果是：

```
ClassLoader is:sun.misc.Launcher$AppClassLoader@73d16e93
```

也就是说明Test.class文件是由AppClassLoader加载的。

这个Test类是我们自己编写的，那么int.class或者是String.class的加载是由谁完成的呢？
我们可以在代码中尝试

```
public class ClassLoaderTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        ClassLoader cl = Test.class.getClassLoader();

        System.out.println("ClassLoader is:"+cl.toString());
```

```
8
9           cl = int.class.getClassLoader();
10
11          System.out.println("ClassLoader is:"+cl.toString());
12
13      }
14
15  }
```

运行一下，却报错了

```
1  ClassLoader is:sun.misc.Launcher$AppClassLoader@73d16e93
2  Exception in thread "main" java.lang.NullPointerException
3      at ClassLoaderTest.main(ClassLoaderTest.java:15)
```


提示的是空指针，意思是int.class这类基础类没有类加载器加载？

当然不是！
int.class是由Bootstrap ClassLoader加载的。要想弄明白这些，我们首先得知道一个前提。

每个类加载器都有一个父加载器

每个类加载器都有一个父加载器，比如加载Test.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。比如代码可以这样编写：

```
1  ClassLoader cl = Test.class.getClassLoader();
2
3  System.out.println("ClassLoader is:"+cl.toString());
4  System.out.println("ClassLoader\'s parent is:"+cl.getParent().toString());
```

运行结果如下：

```
1  ClassLoader is:sun.misc.Launcher$AppClassLoader@73d16e93
2  ClassLoader's parent is:sun.misc.Launcher$ExtClassLoader@15db9742
```

这个说明，AppClassLoader的父加载器是ExtClassLoader。那么ExtClassLoader的父加载器又是谁呢？

```
1  System.out.println("ClassLoader is:"+cl.toString());
2  System.out.println("ClassLoader\'s parent is:"+cl.getParent().toString());
3  System.out.println("ClassLoader\'s grand father is:"+cl.getParent().getParent().toString());
```

运行如果：

```
1  ClassLoader is:sun.misc.Launcher$AppClassLoader@73d16e93
2  Exception in thread "main" ClassLoader's parent is:sun.misc.Launcher$ExtClassLoader@15db9742
3  java.lang.NullPointerException
4      at ClassLoaderTest.main(ClassLoaderTest.java:13)
```

又是一个空指针异常，这表明ExtClassLoader也没有父加载器。那么，为什么标题又是每一个加载器都有一个父加载器呢？这不矛盾吗？为了解释这一点，我们还需要看下面的一个基础前提。

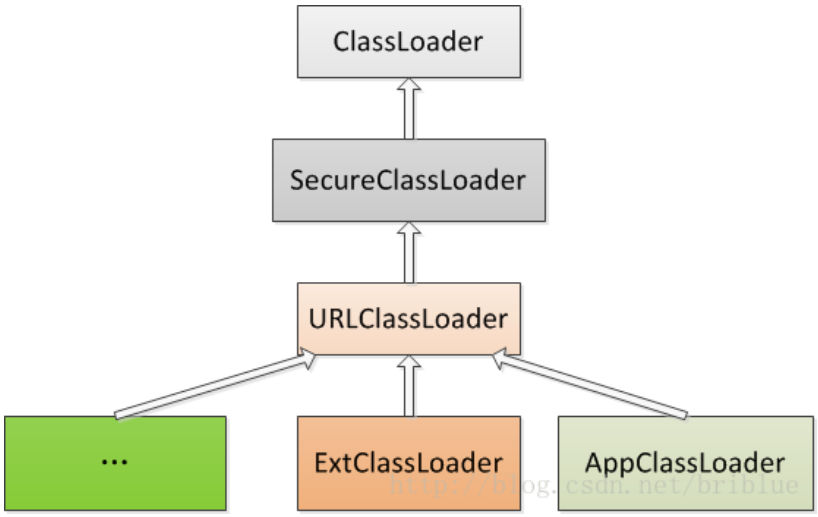
父加载器不是父类

我们先前已经粘贴了ExtClassLoader和AppClassLoader的代码。

```
1  static class ExtClassLoader extends URLClassLoader {}
2  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。

```
1  public abstract class ClassLoader {
2
3  // The parent class loader for delegation
4  // Note: VM hardcoded the offset of this field, thus all new fields
5  // must be added *after* it.
6  private final ClassLoader parent;
7  // The class loader for the system
8      // @GuardedBy("ClassLoader.class")
9  private static ClassLoader scl;
10
11 private ClassLoader(Void unused, ClassLoader parent) {
12     this.parent = parent;
13     ...
14 }
15 protected ClassLoader(ClassLoader parent) {
16     this(checkCreateClassLoader(), parent);
17 }
18 protected ClassLoader() {
19     this(checkCreateClassLoader(), getSystemClassLoader());
20 }
21 public final ClassLoader getParent() {
22     if (parent == null)
23         return null;
24     return parent;
25 }
26 public static ClassLoader getSystemClassLoader() {
27     initSystemClassLoader();
28     if (scl == null) {
29         return null;
30     }
31     return scl;
32 }
33
34 private static synchronized void initSystemClassLoader() {
35     if (!sclSet) {
36         if (scl != null)
37             throw new IllegalStateException("recursive invocation");
38         sun.misc.Launcher l = sun.misc.Launcher.getLauncher();
39         if (l != null) {
40             Throwable oops = null;
41             //通过Launcher获取ClassLoader
42             scl = l.getClassLoader();
43             try {
44                 scl = AccessController.doPrivileged(
45                     new SystemClassLoaderAction(scl));
46             } catch (PrivilegedActionException pae) {
47                 oops = pae.getCause();
48                 if (oops instanceof InvocationTargetException) {
49                     oops = oops.getCause();
```

```
                }
            }
            if (oops != null) {
                if (oops instanceof Error) {
                    throw (Error) oops;
                } else {
                    // wrap the exception
                    throw new Error(oops);
                }
            }
        }
        sclSet = true;
    }
}
}
```


我们可以看到 `getParent()` 实际上返回的就是一个ClassLoader对象parent，parent的赋值是在Clas[illegible]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。

我们主要研究的是ExtClassLoader与AppClassLoader的parent的来源，正好它们与Launcher类有关，我们上面已经粘贴过Launcher的部分代码。

```
public class Launcher {
    private static URLStreamHandlerFactory factory = new Factory();
    private static Launcher launcher = new Launcher();
    private static String bootClassPath =
        System.getProperty("sun.boot.class.path");

    public static Launcher getLauncher() {
        return launcher;
    }

    private ClassLoader loader;

    public Launcher() {
        // Create the extension class loader
        ClassLoader extcl;
        try {
            extcl = ExtClassLoader.getExtClassLoader();
        } catch (IOException e) {
            throw new InternalError(
                "Could not create extension class loader", e);
        }

        // Now create the class loader to use to launch the application
        try {
        //将ExtClassLoader对象实例传递进去
            loader = AppClassLoader.getAppClassLoader(extcl);
        } catch (IOException e) {
            throw new InternalError(
                "Could not create application class loader", e);
        }

public ClassLoader getClassLoader() {
        return loader;
    }
static class ExtClassLoader extends URLClassLoader {

        /**
         * create an ExtClassLoader. The ExtClassLoader is created
         * within a context that limits which files it can read
         */
        public static ExtClassLoader getExtClassLoader() throws IOException
        {
            final File[] dirs = getExtDirs();
```

```
44
45              try {
46                  // Prior implementations of this doPrivileged() block supplied
47                  // aa synthesized ACC via a call to the private method
48                  // ExtClassLoader.getContext().
49
50                  return AccessController.doPrivileged(
51                      new PrivilegedExceptionAction<ExtClassLoader>() {
52                          public ExtClassLoader run() throws IOException {
53                              //ExtClassLoader在这里创建
54                              return new ExtClassLoader(dirs);
55                          }
56                      });
57              } catch (java.security.PrivilegedActionException e) {
58                  throw (IOException) e.getException();
59              }
60          }
61
62
63          /*
64           * Creates a new ExtClassLoader for the specified directories.
65           */
66          public ExtClassLoader(File[] dirs) throws IOException {
67              super(getExtURLs(dirs), null, factory);
68
69          }
70          }
71      }
```


我们需要注意的是

```
1  ClassLoader extcl;
2
3  extcl = ExtClassLoader.getExtClassLoader();
4
5  loader = AppClassLoader.getAppClassLoader(extcl);
```

代码已经说明了问题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。

```
1  public ExtClassLoader(File[] dirs) throws IOException {
2              super(getExtURLs(dirs), null, factory);
3  }
```

对应的代码

```
1  public  URLClassLoader(URL[] urls, ClassLoader parent,
2                         URLStreamHandlerFactory factory) {
3      super(parent);
4  }
```

答案已经很明了了，ExtClassLoader的parent为null。

上面张贴这么多代码也是为了说明AppClassLoader的parent是ExtClassLoader，ExtClassLoader的parent是null。这符合我们之前编写的测试代码。

不过，细心的同学发现，还是有疑问的我们只看到ExtClassLoader和AppClassLoader的创建，那么BootstrapClassLoader呢？

还有，ExtClassLoader的父加载器为null,但是Bootstrap CLassLoader却可以当成它的父加载器这又是为何呢？

我们继续往下进行。

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。

双亲委托

双亲委托。

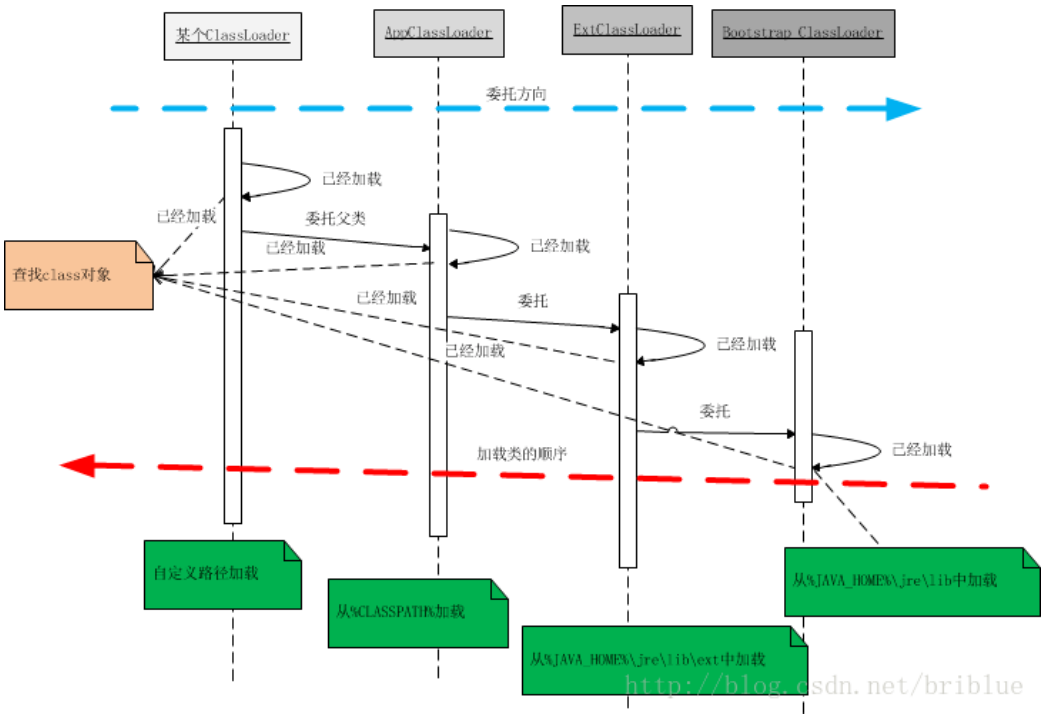
我们终于来到了这一步了。

一个类加载器查找class和resource时，是通过“委托模式”进行的，它首先判断这个class是不是已

如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap Class Loader 如果Bootstrap classloader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身去查找

机制就叫做双亲委托。

整个流程可以如下图所示：



这张图是用时序图画出来的，不过画出来的结果我却自己都觉得不理想。

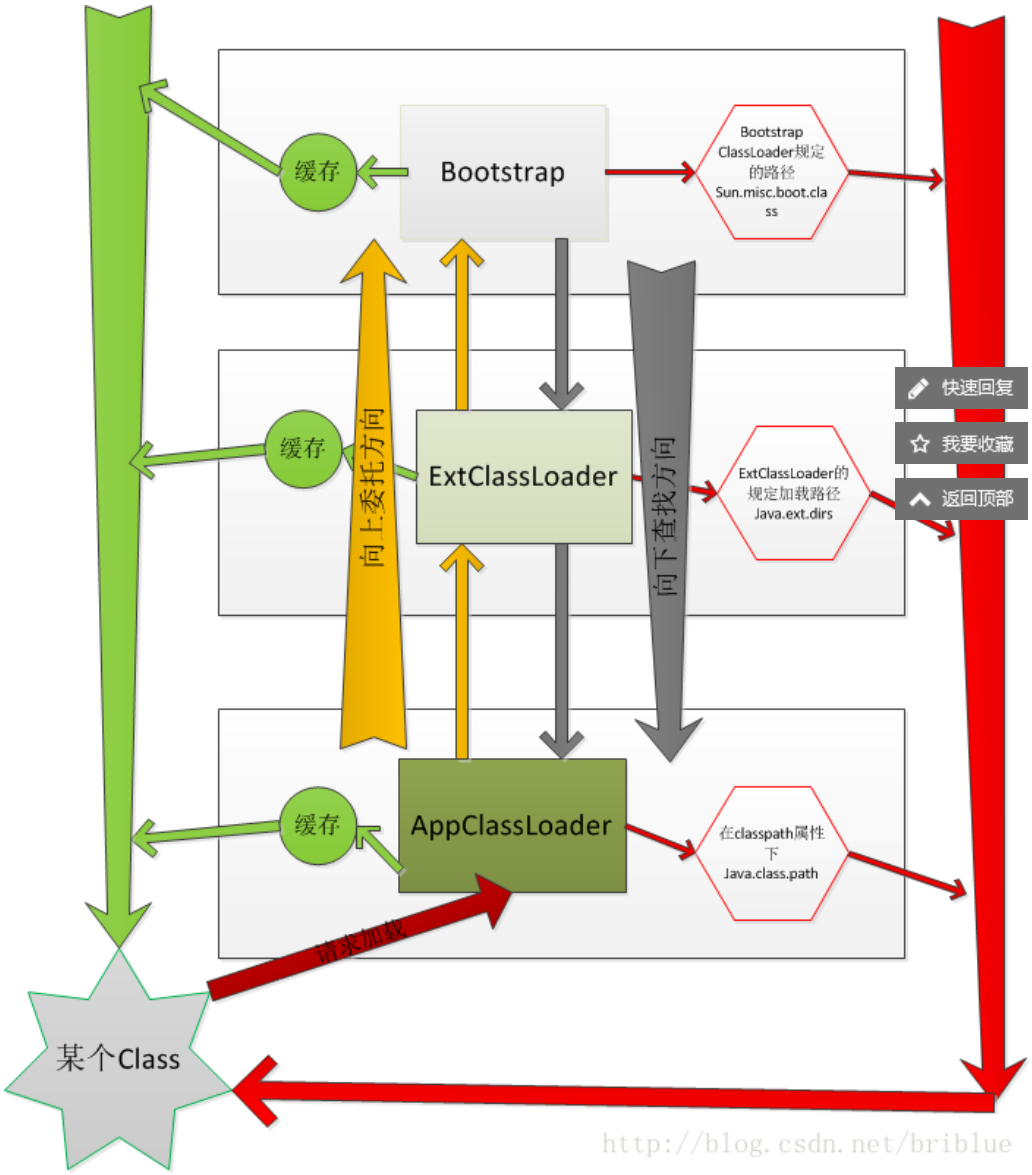
大家可以看到2根箭头，蓝色的代表类加载器向上委托的方向，如果当前的类加载器没有查询到这个class对象已经加载就请求父加载器（不一定是父类）进行操作，然后以此类推。直到Bootstrap ClassLoader。如果Bootstrap ClassLoader也没有加载过此class实例，那么它就会从它指定的路径中去查找，如果查找成功则返回，如果没有查找成功则交给子类加载器，也就是ExtClassLoader,这样类似操作直到终点，也就是我上图中的红色箭头示例。

用序列描述一下：

1. 一个AppClassLoader查找资源时，先看看缓存是否有，缓存有从缓存中获取，否则委托给父加载器。
2. 递归，重复第1部的操作。
3. 如果ExtClassLoader也没有加载过，则由Bootstrap ClassLoader出面，它首先查找缓存，如果没有找到的话，就去找自己的规定的路径下，也就是 `sun.mic.boot.class` 下面的路径。找到就返回，没有找到，让子加载器自己去找。
4. Bootstrap ClassLoader如果没有查找成功，则ExtClassLoader自己在 `java.ext.dirs` 路径中去查找，查找成功就返回，查找不成功，再向下让子加载器找。
5. ExtClassLoader查找不成功，AppClassLoader就自己查找，在 `java.class.path` 路径下查找。找到就返回。如果没有找到就让子类找，如果没有子类会怎么样？抛出各种异常。

上面的序列，详细说明了双亲委托的加载流程。**我们可以发现委托是从下向上，然后具体查找过程却是自上至下。**

我说过上面用时序图画的让自己不满意，现在用框图，最原始的方法再画一次。



上面已经详细介绍了加载过程，但具体为什么是这样加载，我们还需要了解几个个重要的方法loadClass()、findLoadedClass()、findClass()、defineClass()。

重要方法

loadClass()

JDK文档中是这样写的，通过指定的全限定类名加载class，它通过同名的loadClass(String,boolean)方法。

```
protected Class<?> loadClass(String name,
                             boolean resolve)
                      throws ClassNotFoundException
```

上面是方法原型，一般实现这个方法的步骤是

1. 执行 `findLoadedClass(String)` 去检测这个class是不是已经加载过了。
2. 执行父加载器的 `loadClass` 方法。如果父加载器为null，则jvm内置的加载器去替代，也就是Bootstrap ClassLoader。这也解释了ExtClassLoader的parent为null,但仍然说Bootstrap ClassLoader是它的父加载器。
3. 如果向上委托父加载器没有加载成功，则通过 `findClass(String)` 查找。

如果class在上面的步骤中找到了，参数resolve又是true的话，那么 `loadClass()` 又会调用 `resolveClass(Class)` 这个方法来生成最终的Class对象。 我们可以从源代码看出这个步骤。

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
```

```
            // 首先，检测是否已经加载
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        //父加载器不为空则调用父加载器的loadClass
                        c = parent.loadClass(name, false);
                    } else {
                        //父加载器为空则调用Bootstrap Classloader
                        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();
                    //父加载器没有找到，则调用findclass
                    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) {
                //调用resolveClass()
                resolveClass(c);
            }
            return c;
        }
    }
```


代码解释了双亲委托。

另外，要注意的是如果要编写一个classLoader的子类，也就是自定义一个classloader，建议覆盖 `findClass()` 方法，而不要直接改写 `loadClass()` 方法。
另外

```
if (parent != null) {
    //父加载器不为空则调用父加载器的loadClass
    c = parent.loadClass(name, false);
} else {
    //父加载器为空则调用Bootstrap Classloader
    c = findBootstrapClassOrNull(name);
}
```

前面说过ExtClassLoader的parent为null，所以它向上委托时，系统会为它指定Bootstrap ClassLoader。

自定义ClassLoader

不知道大家有没有发现，不管是Bootstrap ClassLoader还是ExtClassLoader等，这些类加载器都只是加载指定的目录下的jar包或者资源。如果在某种情况下，我们需要动态加载一些东西呢？比如从D盘某个文件夹加载一个class文件，或者从网络上下载class主内容然后再进行加载，这样可以吗？

如果要这样做的话，需要我们自定义一个classloader。

自定义步骤

1. 编写一个类继承自ClassLoader抽象类。
2. 复写它的 `findClass()` 方法。
3. 在 `findClass()` 方法中调用 `defineClass()` 。

defineClass()

这个方法在编写自定义classloader的时候非常重要，它能将class二进制内容转换成Class对象，如果不符合要求的会抛出各种异常。

注意点：

一个ClassLoader创建时如果没有指定parent，那么它的parent默认就是AppClassLoader。

上面说的是，如果自定义一个ClassLoader，默认的parent父加载器是AppClassLoader，因为这样就能够保证它能访问系统内置加载器加载成功的class文件。

自定义ClassLoader示例之DiskClassLoader。

假设我们需要一个自定义的classloader,默认加载路径为 `D:\lib` 下的jar包和资源。

我们写编写一个测试用的类文件，Test.java

Test.java

```
package com.frank.test;

public class Test {

    public void say(){
        System.out.println("Say Hello");
    }

}
```

然后将它编译过年class文件Test.class放到 `D:\lib` 这个路径下。

DiskClassLoader

我们编写DiskClassLoader的代码。

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;


public class DiskClassLoader extends ClassLoader {

    private String mLibPath;

    public DiskClassLoader(String path) {
        // TODO Auto-generated constructor stub
        mLibPath = path;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        // TODO Auto-generated method stub

        String fileName = getFileName(name);

        File file = new File(mLibPath,fileName);

        try {
            FileInputStream is = new FileInputStream(file);

            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            int len = 0;
            try {
                while ((len = is.read()) != -1) {
                    bos.write(len);
                }
```


```
            } catch (IOException e) {
                e.printStackTrace();
            }

            byte[] data = bos.toByteArray();
            is.close();
            bos.close();

            return defineClass(name,data,0,data.length);

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        return super.findClass(name);
    }

    //获取要加载 的class文件名
    private String getFileName(String name) {
        // TODO Auto-generated method stub
        int index = name.lastIndexOf('.');
        if(index == -1){
            return name+".class";
        }else{
            return name.substring(index)+".class";
        }
    }

}
```


我们在 `findClass()` 方法中定义了查找class的方法，然后数据通过 `defineClass()` 生成了Class对象。

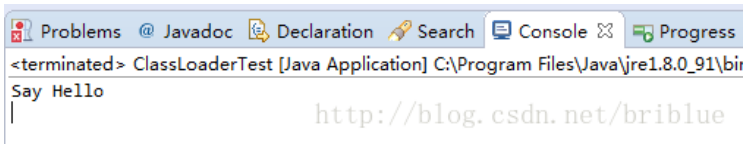
测试

现在我们要编写测试代码。我们知道如果调用一个Test对象的say方法，它会输出”Say Hello”这条字符串。但现在是我们把Test.class放置在应用工程所有的目录之外，我们需要加载它，然后执行它的方法。具体效果如何呢？我们编写的DiskClassLoader能不能顺利完成任务呢？我们拭目以待。

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class ClassLoaderTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        //创建自定义classloader对象。
        DiskClassLoader diskLoader = new DiskClassLoader("D:\\lib");
        try {
            //加载class文件
            Class c = diskLoader.loadClass("com.frank.test.Test");

            if(c != null){
                try {
                    Object obj = c.newInstance();
                    Method method = c.getDeclaredMethod("say",null);
                    //通过反射调用Test类的say方法
                    method.invoke(obj, null);
                } catch (InstantiationException | IllegalAccessException
                        | NoSuchMethodException
                        | SecurityException |
                        IllegalArgumentException |
                        InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }

    }

}
```

我们点击运行按钮，结果显示。



可以看到，Test类的say方法正确执行，也就是我们写的DiskClassLoader编写成功。

回首

讲了这么大的篇幅，自定义ClassLoader才姗姗来迟。 很多同学可能觉得前面有些啰嗦，但我按照自己的思路，我觉得还是有必要的。因为我是围绕一个关键字进行讲解的。

关键字是什么？

关键字 路径

- 从开篇的环境变量
- 到3个主要的JDK自带的类加载器
- 到自定义的ClassLoader

它们的关联部分就是路径，也就是要加载的class或者是资源的路径。
BootStrap ClassLoader、ExtClassLoader、AppClassLoader都是加载指定路径下的jar包。如果我们要突破这种限制，实现自己某些特殊的需求，我们就得自定义ClassLoader，自己指定加载的路径，可以是磁盘、内存、网络或者其它。

所以，你说路径能不能成为它们的关键字？

当然上面的只是我个人的看法，可能不正确，但现阶段，这样有利于自己的学习理解。

自定义ClassLoader还能做什么？

突破了JDK系统内置加载路径的限制之后，我们就可以编写自定义ClassLoader，然后剩下的就叫给开发者你自己了。你可以按照自己的意愿进行业务的定制，将ClassLoader玩出花样来。

玩出花之Class解密类加载器

常见的用法是将Class文件按照某种加密手段进行加密，然后按照规则编写自定义的ClassLoader进行解密，这样我们就可以在程序中加载特定了类，并且这个类只能被我们自定义的加载器进行加载，提高了程序的安全性。

下面，我们编写代码。

1.定义加密解密协议

加密和解密的协议有很多种，具体怎么定看业务需要。在这里，为了便于演示，我简单地将加密解密定义为异或运算。当一个文件进行异或运算后，产生了加密文件，再进行一次异或后，就进行了解密。

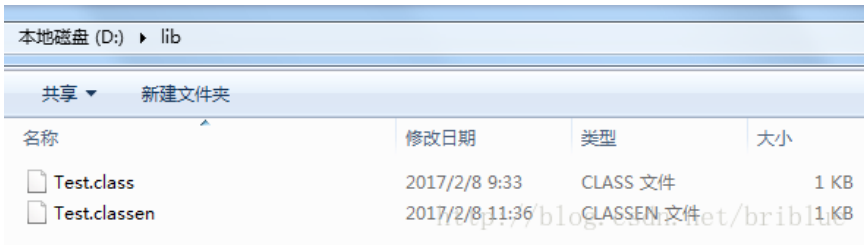
2.编写加密工具类

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;


public class FileUtils {
```

```
9
10      public static void test(String path){
11          File file = new File(path);
12          try {
13              FileInputStream fis = new FileInputStream(file);
14              FileOutputStream fos = new FileOutputStream(path+"en");
15              int b = 0;
16              int b1 = 0;
17              try {
18                  while((b = fis.read()) != -1){
19                      //每一个byte异或一个数字2
20                      fos.write(b ^ 2);
21                  }
22                  fos.close();
23                  fis.close();
24              } catch (IOException e) {
25                  // TODO Auto-generated catch block
26                  e.printStackTrace();
27              }
28          } catch (FileNotFoundException e) {
29              // TODO Auto-generated catch block
30              e.printStackTrace();
31          }
32      }
33
34  }
```


我们再写测试代码

```
1  FileUtils.test("D:\\lib\\Test.class");
```



然后可以看见路径 `D:\\lib\\Test.class` 下Test.class生成了Test.classen文件。

编写自定义classloader，DeClassLoader

```
1  import java.io.ByteArrayOutputStream;
2  import java.io.File;
3  import java.io.FileInputStream;
4  import java.io.IOException;
5
6
7  public class DeClassLoader extends ClassLoader {
8
9      private String mLibPath;
10
11     public DeClassLoader(String path) {
12         // TODO Auto-generated constructor stub
13         mLibPath = path;
14     }
15
16     @Override
17     protected Class<?> findClass(String name) throws ClassNotFoundException {
18         // TODO Auto-generated method stub
19
20         String fileName = getFileName(name);
21
22         File file = new File(mLibPath,fileName);
23
24         try {
25             FileInputStream is = new FileInputStream(file);
26
27             ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

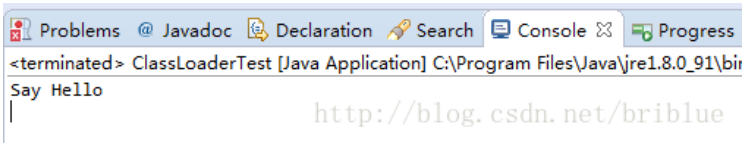

```
            int len = 0;
            byte b = 0;
            try {
                while ((len = is.read()) != -1) {
                    //将数据异或一个数字2进行解密
                    b = (byte) (len ^ 2);
                    bos.write(b);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

            byte[] data = bos.toByteArray();
            is.close();
            bos.close();

            return defineClass(name,data,0,data.length);

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        return super.findClass(name);
    }

    //获取要加载 的class文件名
    private String getFileName(String name) {
        // TODO Auto-generated method stub
        int index = name.lastIndexOf('.');
        if(index == -1){
            return name+".classen";
        }else{
            return name.substring(index+1)+".classen";
        }
    }

}
```


测试

我们可以在ClassLoaderTest.java中的main方法中如下编码：

```
DeClassLoader diskLoader = new DeClassLoader("D:\\lib");
        try {
            //加载class文件
            Class c = diskLoader.loadClass("com.frank.test.Test");

            if(c != null){
                try {
                    Object obj = c.newInstance();
                    Method method = c.getDeclaredMethod("say",null);
                    //通过反射调用Test类的say方法
                    method.invoke(obj, null);
                } catch (InstantiationException | IllegalAccessException
                        | NoSuchMethodException
                        | SecurityException |
                        IllegalArgumentException |
                        InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

查看运行结果是：



可以看到了，同样成功了。现在，我们有两个自定义的ClassLoader:DiskClassLoader和DeClassLoader，我们可以尝试一下，看看DiskClassLoader能不能加载Test.classen文件也就是Test.class加密后的文件。

我们首先移除 `D:\\lib\\Test.class` 文件，只剩下一下Test.classen文件，然后进行代码的测试。

```
DeClassLoader diskLoader1 = new DeClassLoader("D:\\lib");
        try {
            //加载class文件
            Class c = diskLoader1.loadClass("com.frank.test.Test");

            if(c != null){
                try {
                    Object obj = c.newInstance();
                    Method method = c.getDeclaredMethod("say",null);
                    //通过反射调用Test类的say方法
                    method.invoke(obj, null);
                } catch (InstantiationException | IllegalAccessException
                        | NoSuchMethodException
                        | SecurityException |
                        IllegalArgumentException |
                        InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        DiskClassLoader diskLoader = new DiskClassLoader("D:\\lib");
        try {
            //加载class文件
            Class c = diskLoader.loadClass("com.frank.test.Test");

            if(c != null){
                try {
                    Object obj = c.newInstance();
                    Method method = c.getDeclaredMethod("say",null);
                    //通过反射调用Test类的say方法
                    method.invoke(obj, null);
                } catch (InstantiationException | IllegalAccessException
                        | NoSuchMethodException
                        | SecurityException |
                        IllegalArgumentException |
                        InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

运行结果：

```
Problems @ Javadoc Declaration Search Console Progr
<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jre1.8.0_9
Say Hello
java.io.FileNotFoundException: D:\lib\Test.class (系统找不到指定的文件。)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at DiskClassLoader.findClass(DiskClassLoader.java:26)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at ClassLoaderTest.main(ClassLoaderTest.java:45)
java.lang.ClassNotFoundException: com.frank.test.Test
        at java.lang.ClassLoader.findClass(Unknown Source)
        at DiskClassLoader.findClass(DiskClassLoader.java:49)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at ClassLoaderTest.main(ClassLoaderTest.java:45)
```


我们可以看到。DeClassLoader运行正常，而DiskClassLoader却找不到Test.class的类,并且它也无Test.classen文件。

Context ClassLoader 线程上下文类加载器

前面讲到过Bootstrap ClassLoader、ExtClassLoader、AppClassLoader，现在又出来这么一个类加载器，这是为什么？

前面三个之所以放在前面讲，是因为它们是真实存在的类，而且遵从”双亲委托“的机制。而ContextClassLoader其实只是一个概念。

查看Thread.java源码可以发现

```
1  public class Thread implements Runnable {
2
3  /* The context ClassLoader for this thread */
4     private ClassLoader contextClassLoader;
5
6     public void setContextClassLoader(ClassLoader cl) {
7         SecurityManager sm = System.getSecurityManager();
8         if (sm != null) {
9             sm.checkPermission(new RuntimePermission("setContextClassLoader"));
10        }
11        contextClassLoader = cl;
12    }
13
14    public ClassLoader getContextClassLoader() {
15        if (contextClassLoader == null)
16            return null;
17        SecurityManager sm = System.getSecurityManager();
18        if (sm != null) {
19            ClassLoader.checkClassLoaderPermission(contextClassLoader,
20                                                   Reflection.getCallerClass());
21        }
22        return contextClassLoader;
23    }
24 }
```

contextClassLoader只是一个成员变量，通过 `setContextClassLoader()` 方法设置，通过 `getContextClassLoader()` 设置。

每个Thread都有一个相关联的ClassLoader，默认是AppClassLoader。并且子线程默认使用父线程的ClassLoader除非子线程特别设置。

我们同样可以编写代码来加深理解。
现在有2个SpeakTest.class文件，一个源码是

```
1  package com.frank.test;
2
3  public class SpeakTest implements ISpeak {
4
5      @Override
```


```
    public void speak() {
        // TODO Auto-generated method stub
        System.out.println("Test");
    }

}
```

它生成的SpeakTest.class文件放置在`D:\\lib\\test`目录下。

另外ISpeak.java代码

```
 package com.frank.test;

public interface ISpeak {
    public void speak();

}
```


然后，我们在这里还实现了一个SpeakTest.java

```
package com.frank.test;

public class SpeakTest implements ISpeak {

    @Override
    public void speak() {
        // TODO Auto-generated method stub
        System.out.println("I\' frank");
    }

}
```

它生成的SpeakTest.class文件放置在`D:\\lib`目录下。

然后我们还要编写另外一个ClassLoader，DiskClassLoader1.java这个ClassLoader的代码和DiskClassLoader.java代码一致，我们要在DiskClassLoader1中加载位置于`D:\\lib\\test`中的SpeakTest.class文件。

测试代码：

```
DiskClassLoader1 diskLoader1 = new DiskClassLoader1("D:\\lib\\test");
Class cls1 = null;
try {
//加载class文件
 cls1 = diskLoader1.loadClass("com.frank.test.SpeakTest");
System.out.println(cls1.getClassLoader().toString());
if(cls1 != null){
    try {
        Object obj = cls1.newInstance();
        //SpeakTest1 speak = (SpeakTest1) obj;
        //speak.speak();
        Method method = cls1.getDeclaredMethod("speak",null);
        //通过反射调用Test类的speak方法
        method.invoke(obj, null);
    } catch (InstantiationException | IllegalAccessException
            | NoSuchMethodException
            | SecurityException |
            IllegalArgumentException |
            InvocationTargetException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
} catch (ClassNotFoundException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

DiskClassLoader diskLoader = new DiskClassLoader("D:\\lib");
System.out.println("Thread "+Thread.currentThread().getName()+" classloader: "+Thread.current
```


```
new Thread(new Runnable() {

    @Override
    public void run() {
        System.out.println("Thread "+Thread.currentThread().getName()+" classloader: "+Thread

        // TODO Auto-generated method stub
        try {
            //加载class文件
        //  Thread.currentThread().setContextClassLoader(diskLoader);
            //Class c = diskLoader.loadClass("com.frank.test.SpeakTest");
            ClassLoader cl = Thread.currentThread().getContextClassLoader();
            Class c = cl.loadClass("com.frank.test.SpeakTest");
            // Class c = Class.forName("com.frank.test.SpeakTest");
            System.out.println(c.getClassLoader().toString());
            if(c != null){
                try {
                    Object obj = c.newInstance();
                    //SpeakTest1 speak = (SpeakTest1) obj;
                    //speak.speak();
                    Method method = c.getDeclaredMethod("speak",null);
                    //通过反射调用Test类的say方法
                    method.invoke(obj, null);
                } catch (InstantiationException | IllegalAccessException
                        | NoSuchMethodException
                        | SecurityException |
                        IllegalArgumentException |
                        InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}).start();
```


结果如下：

```
<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\ja
DiskClassLoader1@6d06d69c
Test
Thread main classloader: sun.misc.Launcher$AppClassLoader@73d16e93
Thread Thread-0 classloader: sun.misc.Launcher$AppClassLoader@73d16e93
java.lang.ClassNotFoundException: com.frank.test.SpeakTest
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at ClassLoaderTest$1.run(ClassLoaderTest.java:88)
        at java.lang.Thread.run(Unknown Source)
```


我们可以得到如下的信息：

1. DiskClassLoader1加载成功了SpeakTest.class文件并执行成功。
2. 子线程的ContextClassLoader是AppClassLoader。
3. AppClassLoader加载不了父线程当中已经加载的SpeakTest.class内容。

我们修改一下代码，在子线程开头处加上这么一句内容。

```
Thread.currentThread().setContextClassLoader(diskLoader1);
```

结果如下：

```
<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jre1.8.0_9
DiskClassLoader1@6d06d69c
Test
Thread main classloader: sun.misc.Launcher$AppClassLoader@73d16e93
Thread Thread-0 classloader: DiskClassLoader1@6d06d69c
DiskClassLoader1@6d06d69c
Test
```

可以看到子线程的ContextClassLoader变成了DiskClassLoader。

继续改动代码：

```
1  Thread.currentThread().setContextClassLoader(diskLoader);
```

结果：

```
<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jre1.8.0_91\
DiskClassLoader1@6d06d69c
Test
Thread main classloader: sun.misc.Launcher$AppClassLoader@73d16e93
Thread Thread-0 classloader: DiskClassLoader@4f0e0bdb
DiskClassLoader@4f0e0bdb
I'm frank!
```


可以看到DiskClassLoader1和DiskClassLoader分别加载了自己路径下的SpeakTest.class文件，并且它们的类名是一样的 `com.frank.test.SpeakTest` ，但是执行结果不一样，因为它们的实际内容不一样。

Context ClassLoader的运用时机

其实这个我也不是很清楚，我的主业是**Android**，研究ClassLoader也是为了更好的研究Android。网上的答案说是适应那些Web服务框架软件如Tomcat等。主要为了加载不同的APP，因为加载器不一样，同一份class文件加载后生成的类是不相等的。如果有同学想多了解更多的细节，请自行查阅相关资料。

总结

1. ClassLoader用来加载class文件的。
2. 系统内置的ClassLoader通过双亲委托来加载指定路径下的class和资源。
3. 可以自定义ClassLoader一般覆盖findClass()方法。
4. ContextClassLoader与线程相关，可以获取和设置，可以绕过双亲委托的机制。

下一步

1. 你可以研究ClassLoader在Web容器内的应用了，如Tomcat。
2. 可以尝试以这个为基础，继续学习Android中的ClassLoader机制。

引用

我这篇文章写了好几天，修修改改，然后加上自己的理解。参考了下面的这些网站。

1. grepcode ClassLoader源码
2. http://blog.csdn.net/xyang81/article/details/7292380
3. http://blog.csdn.net/irelandken/article/details/7048817
4. https://docs.oracle.com/javase/7/docs/api/java/net/URLClassLoader.html

上一篇 Android绘图Canvas十八般武器之Shader详解及实战篇(下)
下一篇 AndroidStudio不自动添加新创建的文件到VCS

我的同类文章

Android笔记（22）

- 可能是最通俗易懂的 Java 位.. 2017-04-20 阅读 140
- 小甜点，RecyclerView 之 It.. 2017-04-13 阅读 147
- RecyclerView探索之通过Ite... 2017-04-17 阅读 998
- Android绘图Canvas十八般... 2016-12-16 阅读 768

- Android绘图Canvas十八般… 2016-12-15 阅读 536
- ClipDrawable让开发变得更… 2016-12-09 阅读 156
- 通信协议之Protocol buffer(J… 2016-11-16 阅读 2439
- 我眼中最好用的Android日志.. 2016-11-01 阅读 248
- Android常用加密手段之MD… 2016-10-31 阅读 1673
- OKHTTP之缓存配置详解 2016-10-25 阅读 2596

更多文章

参考知识库

Android知识库
33758 关注 | 2806 收录

Python知识库
23286 关注 | 1612 收录

Java 知识库
26380 关注 | 1457 收录

.NET知识库
3841 关注 | 839 收录

Oracle知识库
4986 关注 | 252 收录

微信开发知识库
20437 关注 | 861 收录

软件测试知识库
4621 关注 | 318 收录

Java SE知识库
26026 关注 | 527 收录

Java EE知识库
18048 关注 | 1334 收录

猜你在找

- 从此不求人:自主研发一套PHP前端开发框架
- Hadoop生态系统零基础入门
- HTML 5移动开发从入门到精通
- Swift视频教程(第七季)
- C++语言基础
- Java ClassLoader 原理详细分析
- iOS NSError详解 NSError错误code对照表 自定义定制
- Android自定义ListView轻松实现上下拉刷新一看就懂一
- Linux静态库与动态库详解一看就懂
- C++中模板使用详解---写得很棒一看就懂

查看评论

14楼 qzjqzjqzj 2017-04-28 08:33发表

写得很好了！！

13楼 俺总在笑 2017-04-20 18:14发表

文章写的很详细，通俗易懂。多谢博主！学习了！

Re: frank909 2017-04-20 19:30发表

回复俺总在笑： 谢谢你的支持。也希望你能阅读我另外的博文，并提供建议感想。

12楼 大-树 2017-04-06 17:36发表

很是通俗易懂，像我渣渣好久没有看过这么爽的文章了

11楼 zhiyinqiao6737 2017-04-02 19:18发表

Class.forName("com.frank.test.SpeakTest")
博主你这样写能成功加载吗

Re: frank909 2017-04-05 11:00发表

回复zhiyinqiao6737：
可以的，这都是我亲手敲出来的例子。

10楼 phei03022324 2017-03-28 15:43发表

您好，我是电子工业出版社的编辑，请问您有计划出版图书吗？

Re: frank909 2017-03-30 08:22发表

回复phei03022324：谢谢您的肯定，我没有这个计划。

9楼 Courage_Yeah 2017-03-27 11:59发表

很好的文章，虽然之前有点了解，不过模模糊糊，但现在看了伪代码印象更深刻了，谢谢楼主

8楼 Hitomis丶 2017-03-18 20:46发表

博主，想问下，你是看的什么书啊，知识点梳理的很好

Re: frank909 2017-03-20 16:08发表

回复Hitomis丶：我看的是别人写的博客 源码 还有张孝详老师的视频。然后自己思考了下，怎么写才能让别人比较能看懂。

Re: Hitomis丶 2017-03-22 19:46发表

期待博主以后能有更多像这样的blog

7楼 qq_35463672 2017-03-15 14:26发表

博主可以说下怎么在launcher里面加sysout然后能看到输出的吗

Re: frank909 2017-03-15 14:46发表

回复qq_35463672：Launcher是JDK中的代码。如果你要调试的话，就要自己编译jdk。那是另外一个知识了。

Re: qq_35463672 2017-03-15 20:02发表

回复frank909：用的linux系统吗

Re: frank909 2017-03-16 08:45发表

回复qq_35463672：
http://blog.csdn.net/iam333/article/details/40585419

你看这篇文章

6楼 Big_Centaur 2017-03-15 10:07发表

写的很好，感谢楼主。

5楼 江湖人称某二代 2017-03-11 18:45发表

专门找到原链接来留言的，真的很谢谢你的这篇文章，让之前对热更新之类的技术有了更清晰的认识，再次感谢。博主有考虑写个系列吗？

Re: frank909 2017-03-15 08:42发表

回复江湖人称某二代：谢谢你的有心支持。我是打算写一个系列的，研究这个也是为了准备研究Android中的热更新。但是，写这个很费时。所以没有那么快出第二篇。

Re: 江湖人称某二代 2017-03-16 16:30发表

回复frank909：博主这个系列可以考虑用简书写，大家可以用微信赞赏。我觉得知识就是财富，是应该的。

4楼 Nancy1992 2017-03-10 14:05发表

我们可以看到。DeClassLoader运行正常，而DiskClassLoader却找不到Test.class的类,并且它也无法加载Test.classen文件。你说这句话那里的图，失败的原因似乎是你把文件给删除了。。你这里其实是想告诉读者必须得解密才行，所以应该是搞错了吧?

Re: frank909 2017-03-14 20:47发表

回复Nancy1992：对的 我的意思是必须解密才能得到那个类

3楼 Nancy1992 2017-03-10 13:49发表

从郭霖的公众号看过来的，必须支持一下。但是希望博主能够稍微添加一小节来解释下，双亲委派机制的意义在哪里，过程的话你说的已经很清楚了我也明白了。感谢！

Re: frank909 2017-03-10 13:54发表

回复Nancy1992：双亲的意义在于不需要重复加载类。如果一个类加载器的父类加载器能够加载某个类，那么这个类加载器也能加载这个类。有空我再把文章修改一下，说清楚这个问题。

2楼 伦家不知道 2017-03-09 16:03发表

原谅我要骂人！！
这他妈真的是一篇强文

Re: frank909 2017-03-09 16:31发表

回复伦家不知道：看到前面6个字 我的小心脏可哆嗦了一番。谢谢你的支持与肯定。

1楼 yyyu_ 2017-03-09 09:58发表

很详细，正好最近在看JVM，谢谢博主，学习了。

Re: frank909 2017-03-09 10:25发表

回复yyyu_：写的详细是因为我写了整整一周，反复修改。怕别人看不懂，其实在这样反复思考的过程中可以发现自己的思路和理解更深更透彻了。写博客是为了分享，更重要的是写的过程中自己的认知也进步了。最后，谢谢你的支持和鼓励。我们一起学习，一起进步。今年我也打算把jvm知识内容学习完。