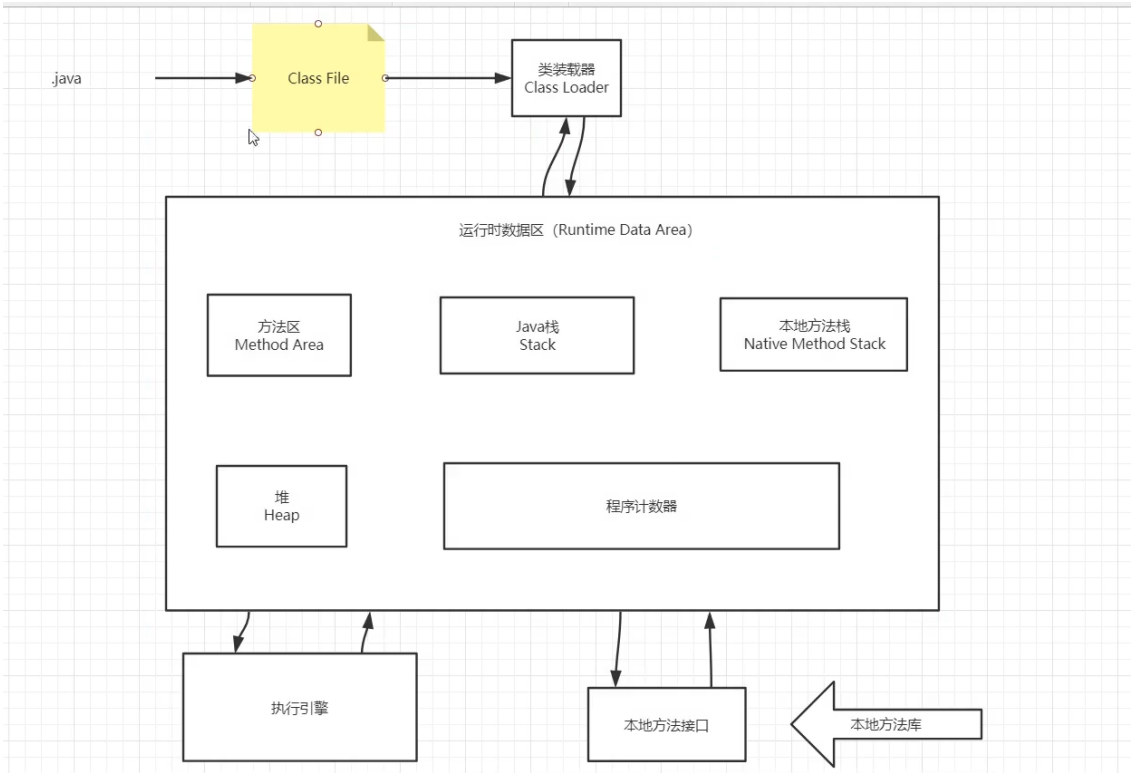
JVM类加载器与双亲委派

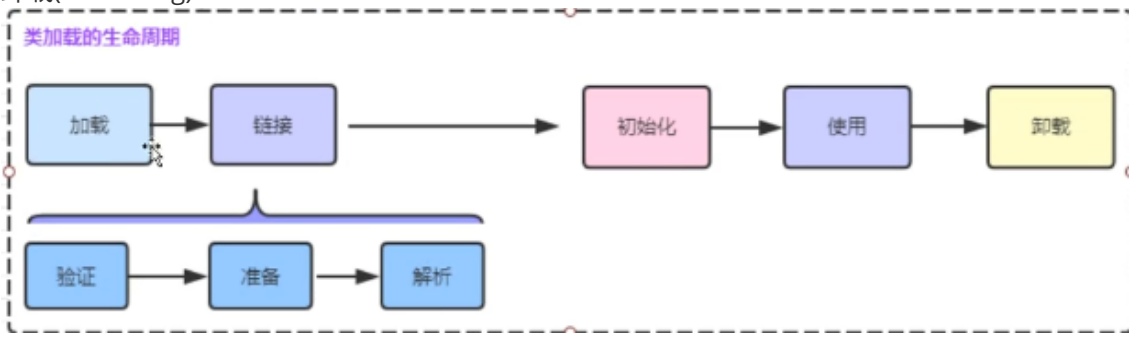
- 学习视频：学习视频



类的生命周期

> 一个类从加载到jvm内存，到从jvm内存卸载，生命周期可分为七个阶段。

1. 加载(Loading)：classpath，jar包，网络，磁盘位置下的类的class以二进制字节流读进来，在内存中生成一个代表这个类的Class类对象放入元空间。可以自定义类加载器。
2. 验证(Verification)：验证Class文件的字节流中包含的信息是否符合java虚拟机规范
3. 准备(Preparation)：类变量赋默认初始值
4. 解析(Resolution)：把符合引用翻译为直接引用
5. 初始化(Initialization)
6. 使用(Using)
7. 卸载(Unloading)



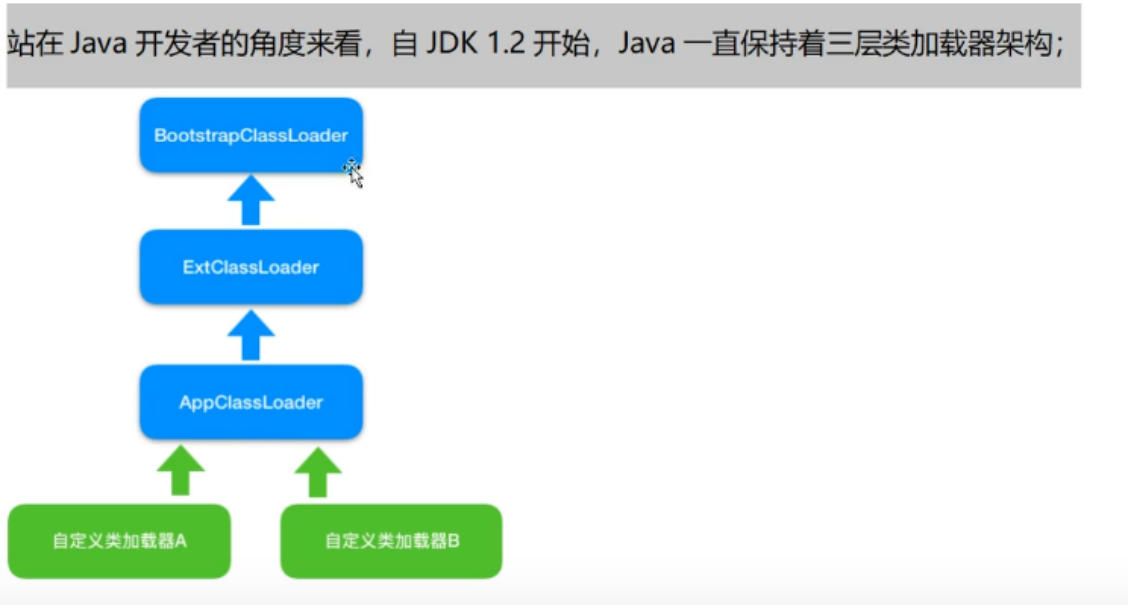
类加载器

- 什么是类加载器

> 在类加载阶段，通过一个类的全限定名来获取描述该类的二进制字节流的这个动作的'代码'被称为'类加载器'，这个动作可以自动有实现

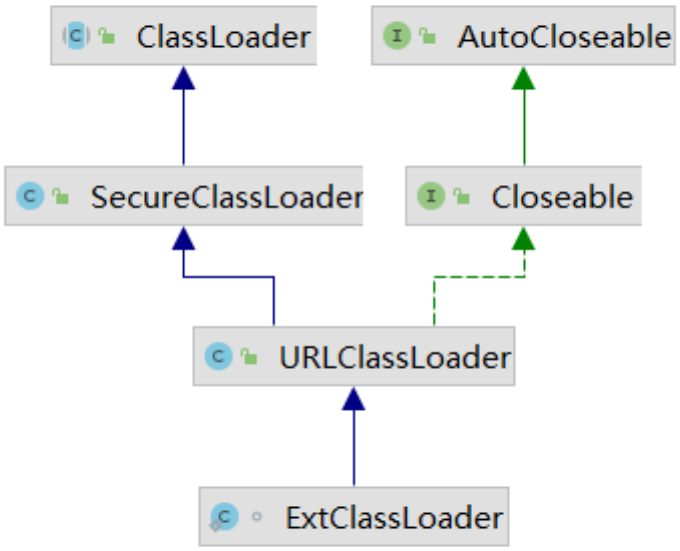
- jvm有哪些类加载器

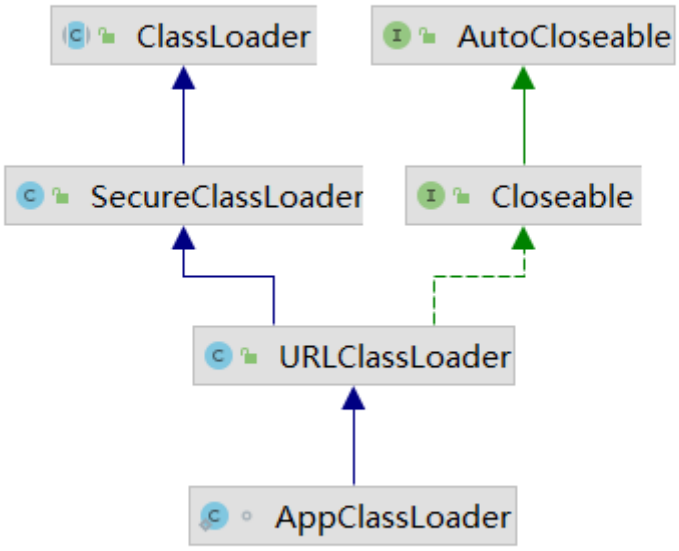
1. 启动类（根）加载器：（Bootstrap Classloader），使用C++实现，是虚拟机自身的一部分
2. 其他类加载器：由java语言实现，全部继承自抽象类`java.lang.CLassLoader`
3. jdk的三层类加载结构



- 不同类加载器加载哪些文件

1. 启动类加载器：<JAVA_HOME>\jre\lib\rt.jar,resources.jar,charsets.jar，被-Xbootclasspath参数所指定的路径中存放的类库
2. 扩展类加载器：<JAVA_HOME>\jre\lib\ext，被java.ext.dirs系统变量所指定的路径中所有的类库。
3. 应用程序类加载器(Application Classloader)：系统类加载器，加载用户类路径（classPath）上所有的类库。
   注意：三层加载器不是类的继承关系。`appclassloader`和`extclassloader`继承自`urlclassload`，二者之间没有继承关系。

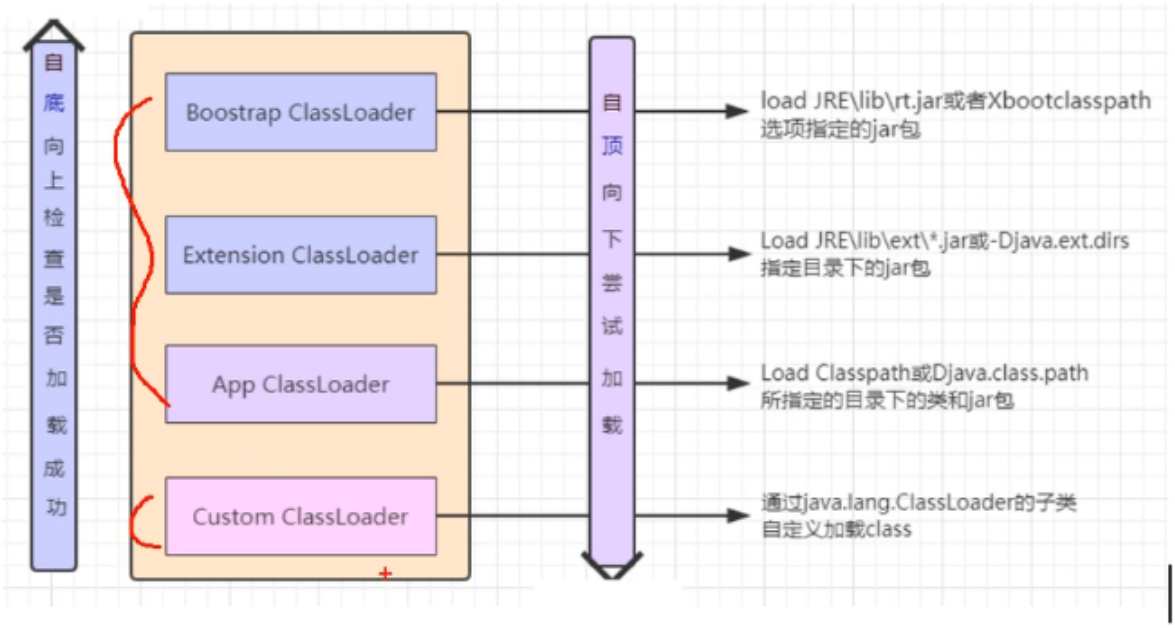





```
package com;

public class example {
    public static void main(String[] args) {
        example example = new example();

        Class<? extends com.example> aClass = example.getClass();
```

```
        ClassLoader classLoader = aClass.getClassLoader();
        System.out.println(classLoader); //AppClassLoader

        System.out.println(classLoader.getParent());//ExtClassLoader
        System.out.println(classLoader.getParent().getParent());//null  1. 不存在或java程序获取不到，根加载器使用CPP系的，无法获取
        System.out.println(classLoader.getParent().getParent().getParent());
    }
}
//////////////////////////////////////////////////
sun.misc.Launcher$AppClassLoader@18b4aac2
sun.misc.Launcher$ExtClassLoader@4554617c
null
Exception in thread "main" java.lang.NullPointerException
   at com.example.main(example.java:14)
```

双亲委派模型

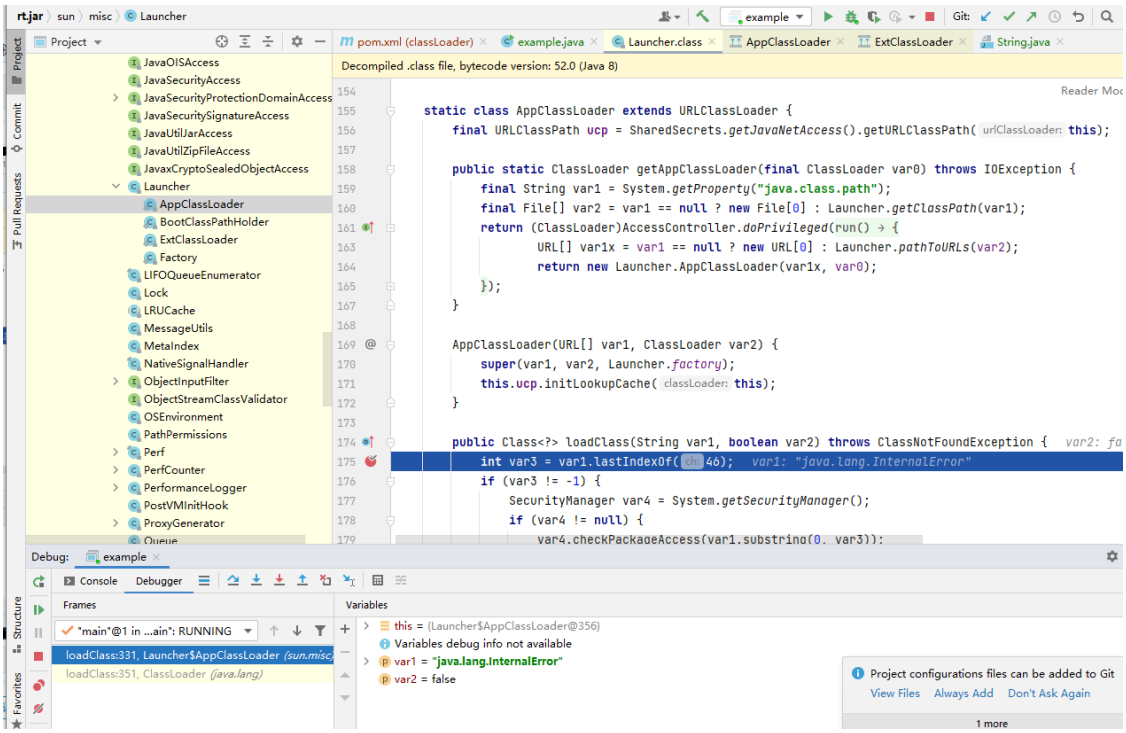


双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到最顶层的启动类加载器中，只有当上一层类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到这个类）时，下一层类加载器才会尝试自己去加载；

> 当尝试加载`java.lang.String`类的时候，首先是`App ClassLoader`委派给`Ext ClassLoader`，然后`Ext ClassLoader`也不加载而是委派给`BootStrap ClassLoader`进行加载，之后`Bootstrap Loader`尝试加载，如果加载失败再交还`Ext classLoader`进行加载，如果依然失败再交给`App Classloader`。

- 打破双亲委派模型

> 自定义类加载器，重写其中的loadClass方法，使其不进行双亲委派。



```
public Class<?> loadClass(String var1, boolean var2) throws
ClassNotFoundException {
            int var3 = var1.lastIndexOf(46);
            if (var3 != -1) {
                SecurityManager var4 = System.getSecurityManager();
                if (var4 != null) {
                    var4.checkPackageAccess(var1.substring(0, var3));
                }
            }

            if (this.ucp.knownToNotExist(var1)) {
                Class var5 = this.findLoadedClass(var1);
                if (var5 != null) {
                    if (var2) {
                        this.resolveClass(var5);
                    }

                    return var5;
                } else {
                    throw new ClassNotFoundException(var1);
                }
            } else {
                return super.loadClass(var1, var2); //调用父类加载器
（java.lang.CLassloader）
            }
        }
```

> 当调用`super.loadClass(var1, var2)`去找父类加载的时候，最终找到的是`java.lang.CLassloader`，再这个`java.lang.CLassloader#loadClass()`方法中实现了双亲委派。

```
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;
        }
    }
```

> `if (parent != null)`会判断这个父加载器是否存在，如果存在，则使用父加载器，其中注意的是parent是jvm指定的，并不是子父类的继承关系。

- 类加载过程

  1、首先是调用`public Class<?> loadClass(String name)`方法，通过public方法调用保护方法`protected Class<?> loadClass(String name, boolean resolve)`

  2、在protected loadClass方法中，第406行会调用一个findLoadedClass方法判断当前类是否已经加载。

  如果类已经加载，直接返回当前类的类对象。

  3、如果创建当前ClassLoader时传入了父类加载器(new ClassLoader(父类加载器))就使用父类加载器加载TestHelloWorld类，否则使用JVM的Bootstrap ClassLoader加载。

  4、如果通过类加载器没有办法加载类，则会通过findClass方法尝试加载类。

  5、如果当前的ClassLoader没有重写findClass方法，则会直接返回类不存在。跟进findClass方法进行查看。

如果当前类重写了findClass方法并通过传入的com.anbai.sec.classloader.TestHelloWorld类名找到了对应的类字节码，那么应该调用defineClass方法去JVM中注册该类。

6、如果调用loadClass的时候传入的resolve参数为true，那么还需要调用resolveClass方法链接类,默认为false。

```
432             if (resolve) {
433                 resolveClass(c);
434             }
```

7、返回一个JVM加载后的java.lang.Class类对象

自定义ClassLoader

> java.lang.ClassLoader是所有的类加载器的父类，java.lang.ClassLoader有非常多的子类加载器，比如我们用于加载jar包的java.net.URLClassLoader其本身通过继承java.lang.ClassLoader类，重写了findClass方法从而实现了加载目录class文件甚至是远程资源文件。

```
package com.sec.classloader;

import java.lang.reflect.Method;

/*
尝试自定义ClassLoader
如果一个TestHelloWorld类根本不存在，我们可以通过自定义类加载器重写findClass方法，然后调用defineClass方法
的时候传入TestHelloWorld类的字节码，来像JVM中定义一个TestHelloWorld类，
最后通过反射机制就可以调用TestHelloWorld类的hello方法了。
package com.anbai.sec.classloader;
 * Creator: yz
 * Date: 2019/12/17

//public class TestHelloWorld {
//
//    public String hello() {
//        return "Hello World~";
//    }
//
//}
*/
public class selfClassLoader extends ClassLoader {
    private static String
testClassName="com.anbai.sec.classloader.TestHelloWorld";
    //testHelloWorld类的字节码
    private static byte[] testClassBytes = new byte[]{
            -54, -2, -70, -66, 0, 0, 0, 51, 0, 17, 10, 0, 4, 0, 13, 8, 0, 14, 7,
0, 15, 7, 0,
            16, 1, 0, 6, 60, 105, 110, 105, 116, 62, 1, 0, 3, 40, 41, 86, 1, 0,
4, 67, 111, 100,
            101, 1, 0, 15, 76, 105, 110, 101, 78, 117, 109, 98, 101, 114, 84,
97, 98, 108, 101,
            1, 0, 5, 104, 101, 108, 108, 111, 1, 0, 20, 40, 41, 76, 106, 97,
118, 97, 47, 108,
            97, 110, 103, 47, 83, 116, 114, 105, 110, 103, 59, 1, 0, 10, 83,
111, 117, 114, 99,
```

```
            101, 70, 105, 108, 101, 1, 0, 19, 84, 101, 115, 116, 72, 101, 108,
108, 111, 87, 111,
            114, 108, 100, 46, 106, 97, 118, 97, 12, 0, 5, 0, 6, 1, 0, 12, 72,
101, 108, 108, 111,
            32, 87, 111, 114, 108, 100, 126, 1, 0, 40, 99, 111, 109, 47, 97,
110, 98, 97, 105, 47,
            115, 101, 99, 47, 99, 108, 97, 115, 115, 108, 111, 97, 100, 101,
114, 47, 84, 101, 115,
            116, 72, 101, 108, 108, 111, 87, 111, 114, 108, 100, 1, 0, 16, 106,
97, 118, 97, 47, 108,
            97, 110, 103, 47, 79, 98, 106, 101, 99, 116, 0, 33, 0, 3, 0, 4, 0,
0, 0, 0, 0, 2, 0, 1,
            0, 5, 0, 6, 0, 1, 0, 7, 0, 0, 0, 29, 0, 1, 0, 1, 0, 0, 0, 5, 42,
-73, 0, 1, -79, 0, 0, 0,
            1, 0, 8, 0, 0, 0, 6, 0, 1, 0, 0, 0, 7, 0, 1, 0, 9, 0, 10, 0, 1, 0,
7, 0, 0, 0, 27, 0, 1,
            0, 1, 0, 0, 0, 3, 18, 2, -80, 0, 0, 0, 1, 0, 8, 0, 0, 0, 6, 0, 1, 0,
0, 0, 10, 0, 1, 0, 11,
            0, 0, 0, 2, 0, 12
    };
    public Class<?> findClass(String name) throws ClassNotFoundException{
        //只处理testHelloWorld类
        if (name.equals(testClassName)){
            return defineClass(testClassName,testClassBytes,0,
testClassBytes.length);
        }
        return super.findClass(name);  //如果不是testHelloWorld返回父类的findClass方
法，即类不存在
    }

    public static void main(String[] args) {
        selfClassLoader loader=new selfClassLoader();
        try {
            Class testClass=loader.loadClass(testClassName); //调用loadClass方法加
载类，返回一个Class类对象
            Object testInstance=testClass.newInstance();//// 反射创建
TestHelloWorld类，等价于 TestHelloWorld t = new TestHelloWorld();
            Method method=testInstance.getClass().getMethod("hello");
            String str=(String) method.invoke(testInstance);
            System.out.println(str);

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

> 利用自定义类加载器我们可以在webshell中实现加载并调用自己编译的类对象，比如本地命令执行漏洞调用自定义类字节码的native方法绕过RASP检测，也可以用于加密重要的Java类字节码(只能算弱加密了)。

- loadClass，findClass，defineClass区别

1. loadClass主要进行类加载的方法，默认的双亲委派机制在这个方法中实现，当我们需要打破双亲委派机制时可以通过重写loadClass方法
2. findClass根据名称或位置加载.class字节码

3. defineClass把字节码转换为Class类对象。

URLClassLoader

```
package com.sec.classloader;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.net.URL;
import java.net.URLClassLoader;

/*
URLClassLoader继承了ClassLoader，URLClassLoader提供了加载远程资源的能力
在写漏洞利用的payload或者webshell的时候我们可以使用这个特性来加载远程的jar来实现远程的类方法调用。
 */
public class urlClassLoaderDemo {
    public static void main(String[] args) {
        try {
            // 定义远程加载的jar路径
            URL url = new URL("http://localhost/java/calc.jar");

            // 创建URLClassLoader对象，并加载远程jar包
            URLClassLoader ucl = new URLClassLoader(new URL[]{url});

            // 定义需要执行的系统命令
            String cmd = "whoami";

            // 通过URLClassLoader加载远程jar包中的CMD类
            Class cmdClass = ucl.loadClass("calc");

            // 调用CMD类中的exec方法，等价于: Process process = CMD.exec("whoami");
            Process process = (Process) cmdClass.getMethod("exec",
String.class).invoke(null, cmd);

            // 获取命令执行结果的输入流
            InputStream in = process.getInputStream();
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            byte[] b = new byte[1024];
            int a = -1;

            // 读取命令执行结果
            while ((a = in.read(b)) != -1) {
                baos.write(b, 0, a);
            }

            // 输出命令执行结果
            System.out.println(baos.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

`TemplatesImpl`加载字节码

说明

> 利用`classloader`中的`defineClass`直接加载字节码。每一个类加载器最后都是通过`defineClass`方法来加载字节码。在`TemplatesImpl`类中有实现自定义的`defineClass`，可以通过这个`TemplatesImpl`类来加载我们自己的代码。调用链：TemplatesImpl#getOutputProperties() -> TemplatesImpl#newTransformer() ->TemplatesImpl#getTransletInstance() -> TemplatesImpl#defineTransletClasses()-> TransletClassLoader#defineClass()

```
package SecurityRambling.TemplatesImpl_;


import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;

import javax.xml.transform.TransformerConfigurationException;
import java.io.*;
import java.lang.reflect.Field;
import java.util.Base64;

public class TemplatesImpl_ {
    public static void main(String[] args) throws
TransformerConfigurationException, IOException {
        String str="yv66vgAAADQAIQoABgASCQATABQIABUKABYAFwcAG" +

 "AcAGQEACXRyYW5zZm9ybQEAcihMY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3h" +

 "zbHRjLORPTTtbTGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW5OZXJuYWwvc2VyaWFsaXplci9TZ" +

 "XJpYWxpemFOaW9uSGFuZGxlcjspVgEABENvZGUBAA9MaW5lTnVtYmVyVGFibGUBAApFeGNlcHRpb"
+

 "25zBwAaAQCmKExjb2Ovc3VuL29yZy9hcGFjaGUveGFsYW4vaW5OZXJuYWwveHNsdGMvRE9NOOxjb"
+

 "2Ovc3VuL29yZy9hcGFjaGUveG1sL2ludGVybmFsL2ROObS9EVE1BeGlzSXRlcmFOb3I7TGNvbS9zdW"
+

 "4vb3JnL2FwYWNoZS94bWwvaW5OZXJuYWwvc2VyaWFsaXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjsp
" +

 "VgEABjxpbmlOPgEAAygpVgEAClNvdXJjZUZpbGUBABdIZWxsb1RlbXBsYXRlc0ltcGwuamF2YQwADg
AP" +

 "BwAbDAAcAB0BABNIZWxsbyBUZW1wbGFOZXNJbXBsBwAeDAAfACABABJIZWxsb1RlbXBsYXRlc0ltcG
wBAEBjb" +

 "2Ovc3VuL29yZy9hcGFjaGUveGFsYW4vaW5OZXJuYWwveHNsdGMvcnVudGltZS9BYnNOcmFjdFRyYW5
zbGV0AQA" +

 "5Y29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjL1RyYW5zbGV0RXhjZXB0aW9uA
QAQamF2Y" +

 "S9sYW5nL1N5c3RlbQEAA291dAEAFUxqYXZhL2lvL1ByaW50U3RyZWFtOwEAE2phdmEvaW8vUHJpbnR
TdHJlYW0" +
```

```
 "BAAdwcmludGxuAQAVKExqYXZhL2xhbmcvU3RyaW5nOylWACEABQAGAAAAAAADAAEABwAIAAIACQAAABkAAAADAAA" +

 "AAbEAAAABAAoAAAAGAAEAAAAIAAsAAAAEAAEADAABAAcADQACAAkAAAAZAAAABAAAAAGxAAAAAQAKAAAABgABAAAA" +

 "CgALAAAABAABAAwAAQAOAA8AAQAJAAAALQACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAEACgAAAA4AAwAAAA0ABAAO" +
            "AAwADwABABAAAAACABE=";
        //System.out.println(str);
        byte[] code = Base64.getDecoder().decode(str);
        byteToFile(code);
        String s = fileToBase64();
        System.out.println(s);
        //System.out.println(new String(code));
        TemplatesImpl templates = new TemplatesImpl();
        TemplatesImpl_.setFieldValue(templates,"_name","HelloTemplatesImpl");
        TemplatesImpl_.setFieldValue(templates,"_bytecodes",new byte[][]{code});
        TemplatesImpl_.setFieldValue(templates,"_tfactory",new TransformerFactoryImpl());
        templates.newTransformer();
    }

    private static void setFieldValue(Object obj,String FieldName,Object setObj){
        try {
            Field field = obj.getClass().getDeclaredField(FieldName);
            field.setAccessible(true);
            field.set(obj,setObj);
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    private static byte[] fileToByte() throws IOException {  //文件转字节
        File file = new File("E:\\技术文章\\自己的\\代码审计\\java\\ysoserial\\src\\main\\java\\SecurityRambling\\TemplatesImpl_\\test.class");
        FileInputStream inputStream = new FileInputStream(file);
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        byte[] bytes = new byte[1024];
        int n;
        while ((n=inputStream.read(bytes))!=-1){
            byteArrayOutputStream.write(bytes,0,n);
        }
        inputStream.close();
        byteArrayOutputStream.close();
        return byteArrayOutputStream.toByteArray();
    }

    private static void byteToFile(byte[] bytes) throws IOException{  //字节转文件
        if(bytes.length == 0){
            return;
        }
```

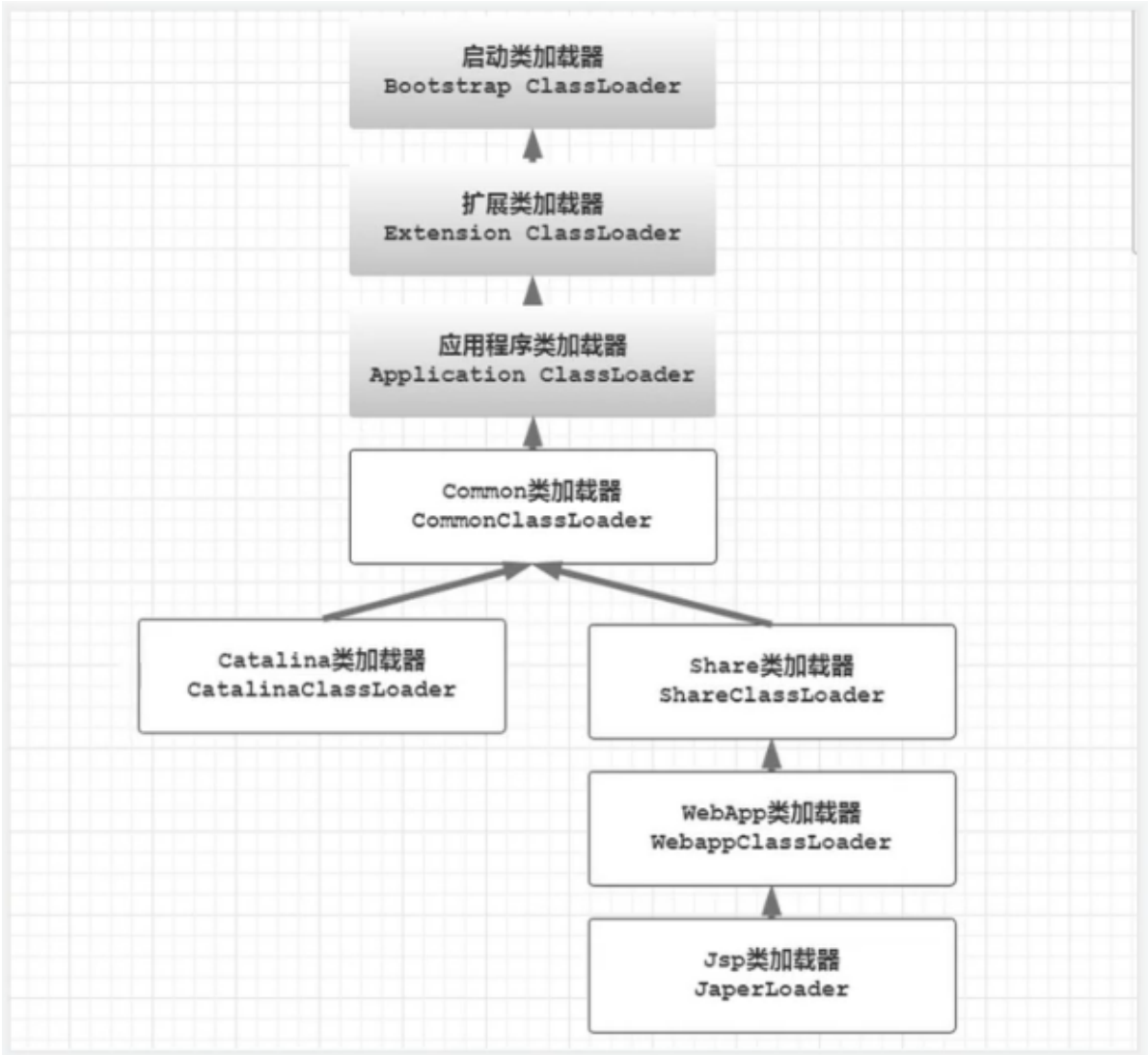
```
        File file = new File("E:\\技术文章\\自己的\\代码审计
\\java\\ysoserial\\src\\main\\java\\SecurityRambling\\TemplatesImpl_\\test.class"
);
        FileOutputStream fileOutputStream = new FileOutputStream(file);
        BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(fileOutputStream);
        bufferedOutputStream.write(bytes);
        bufferedOutputStream.close();
        fileOutputStream.close();
    }

    private static String fileToBase64() throws IOException{  //文件转base64编码
        byte[] bytes = fileToByte();
        byte[] encode = Base64.getEncoder().encode(bytes);
        return new String(encode);
    }
}
```

几点疑问

1. 利用`TemplatesImpl`加载字节码之后是不会自动创建类对象的，这个对象需要显示的创建。在上面的`demo`当中创建对象的代码位于`getTransletInstance`方法当中。通过反射创建。

```
446     private Translet getTransletInstance()
447         throws TransformerConfigurationException {
448         try {
449             if (_name == null) return null;
450
451             if (_class == null) defineTransletClasses();
452
453             // The translet needs to keep a reference to all its auxiliary
454             // class to prevent the GC from collecting them
455             AbstractTranslet translet = (AbstractTranslet) _class[_transletIndex].newInstance();
456             translet.postInitialization();
457             translet.setTemplates(this);
458             translet.setServicesMechnism(_useServicesMechanism);
459             translet.setAllowedProtocols(_accessExternalStylesheet);
460             if (_auxClasses != null) {
461                 translet.setAuxiliaryClasses(_auxClasses);
462             }
463
464             return translet;
465         }
```

tomcat的类加载机制



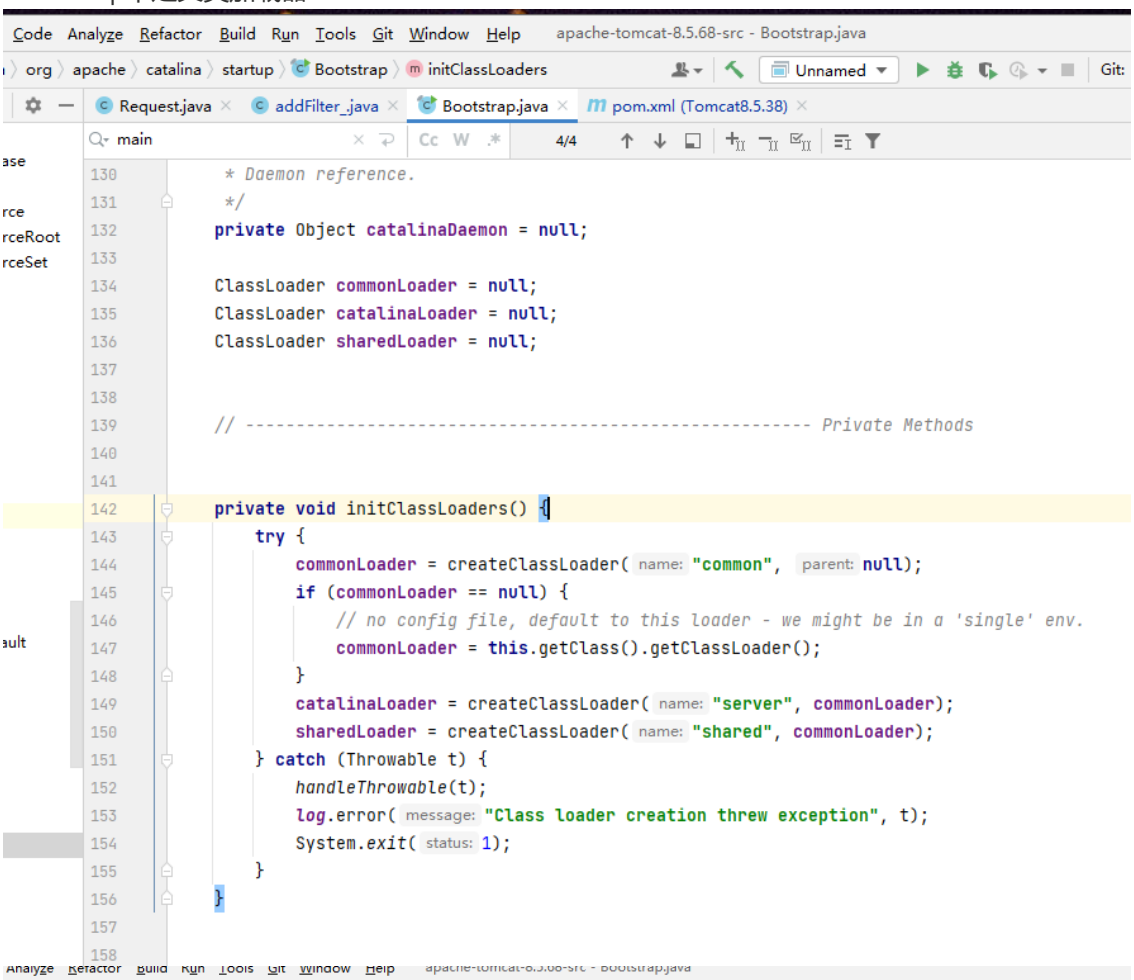
可以看到，在原来的 Java 的类加载机制基础上，Tomcat 新增了 3 个基础类加载器和每个 Web 应用的类加载器+JSP 类加载器；

3 个基础类加载器在 conf/catalina.properties 中进行配置：

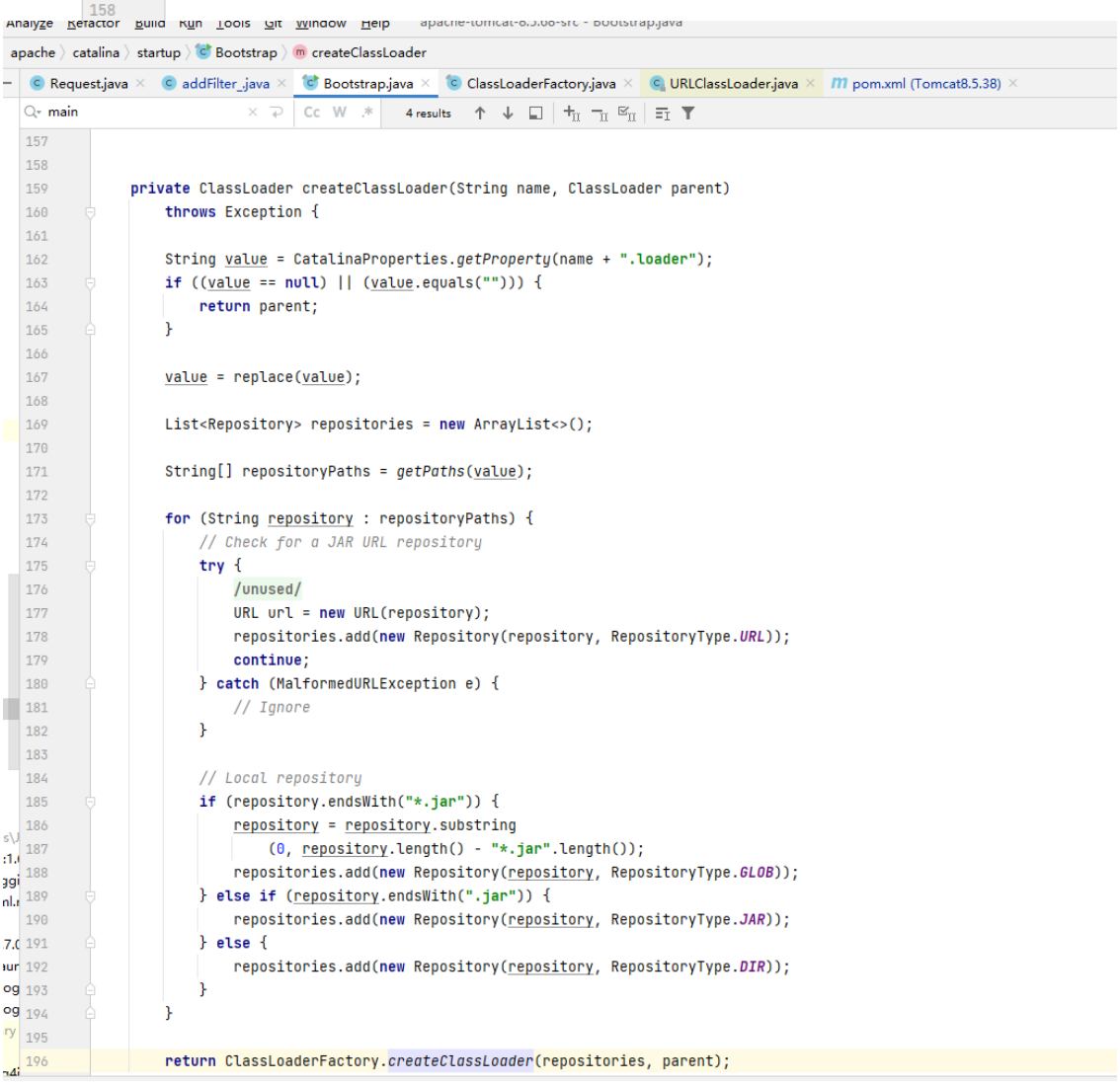
```
common.loader="${catalina.base}/lib","${catalina.base}/lib/*.jar","${catalina.hom
e}/lib","${catalina.home}/lib/*.jar"
server.loader=
shared.loader=
```

Tomcat 自定义了 WebAppClassLoader 类加载器，打破了双亲委派的机制，即如果收到类加载的请求，首先会尝试自己去加载，如果找不到再交给父加载器去加载，目的就是为了优先加载 Web 应用自己定义的类，我们知道 ClassLoader 默认的 loadClass 方法是以双亲委派的模型进行加载类的，那么 Tomcat 打破了这个规则，重写了 loadClass 方法，我们可以看到

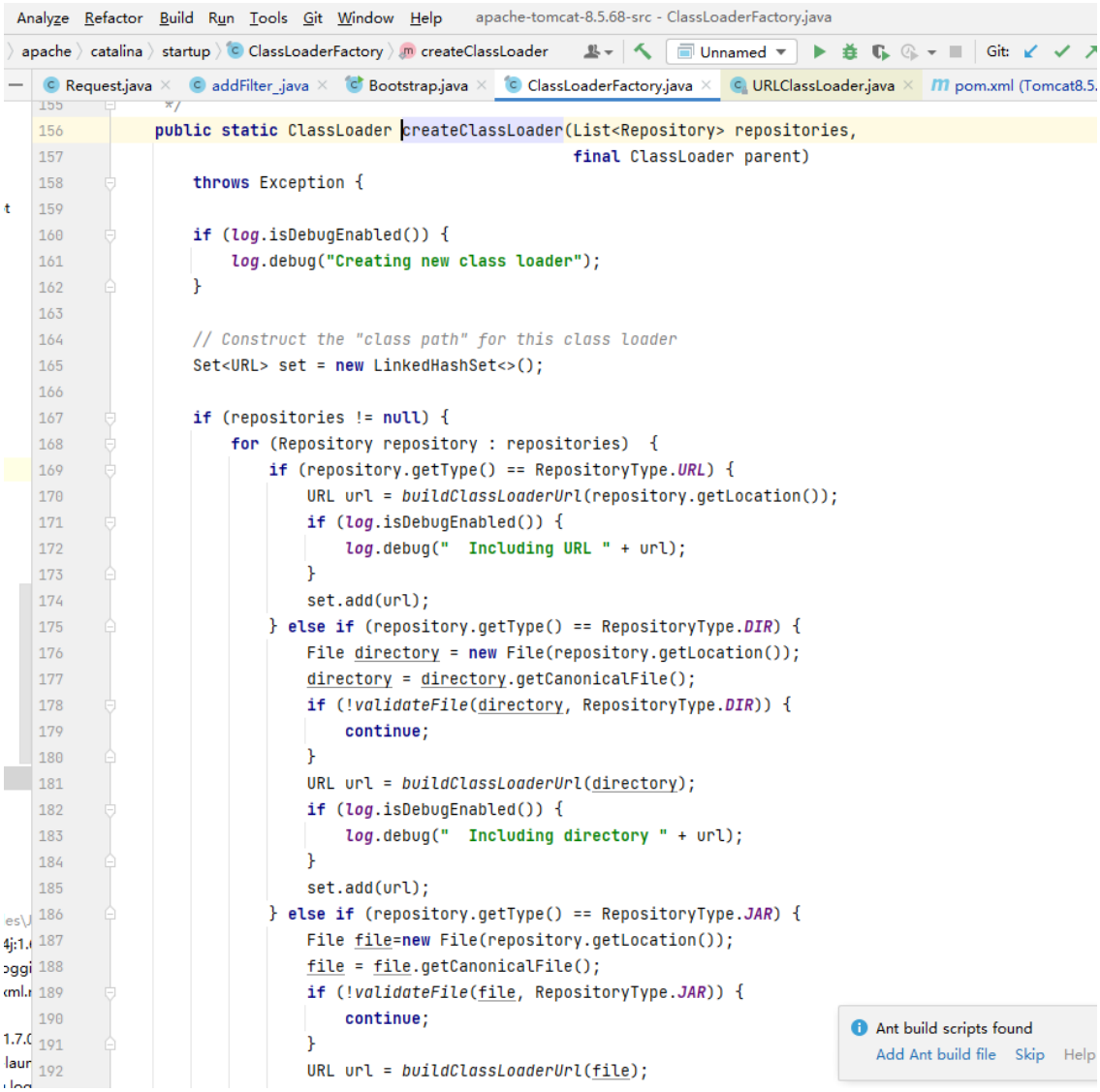
- BootStrap中定义类加载器



```
130      * Daemon reference.
131      */
132     private Object catalinaDaemon = null;
133
134     ClassLoader commonLoader = null;
135     ClassLoader catalinaLoader = null;
136     ClassLoader sharedLoader = null;
137
138
139     // -------------------------------------------------------- Private Methods
140
141
142     private void initClassLoaders() {
143         try {
144             commonLoader = createClassLoader( name: "common",  parent: null);
145             if (commonLoader == null) {
146                 // no config file, default to this loader - we might be in a 'single' env.
147                 commonLoader = this.getClass().getClassLoader();
148             }
149             catalinaLoader = createClassLoader( name: "server", commonLoader);
150             sharedLoader = createClassLoader( name: "shared", commonLoader);
151         } catch (Throwable t) {
152             handleThrowable(t);
153             log.error( message: "Class loader creation threw exception", t);
154             System.exit( status: 1);
155         }
156     }
157
158
```



```
157
158
159     private ClassLoader createClassLoader(String name, ClassLoader parent)
160         throws Exception {
161
162         String value = CatalinaProperties.getProperty(name + ".loader");
163         if ((value == null) || (value.equals(""))) {
164             return parent;
165         }
166
167         value = replace(value);
168
169         List<Repository> repositories = new ArrayList<>();
170
171         String[] repositoryPaths = getPaths(value);
172
173         for (String repository : repositoryPaths) {
174             // Check for a JAR URL repository
175             try {
176                 /unused/
177                 URL url = new URL(repository);
178                 repositories.add(new Repository(repository, RepositoryType.URL));
179                 continue;
180             } catch (MalformedURLException e) {
181                 // Ignore
182             }
183
184             // Local repository
185             if (repository.endsWith("*.jar")) {
186                 repository = repository.substring
187                     (0, repository.length() - "*.jar".length());
188                 repositories.add(new Repository(repository, RepositoryType.GLOB));
189             } else if (repository.endsWith(".jar")) {
190                 repositories.add(new Repository(repository, RepositoryType.JAR));
191             } else {
192                 repositories.add(new Repository(repository, RepositoryType.DIR));
193             }
194         }
195
196         return ClassLoaderFactory.createClassLoader(repositories, parent);
```


Analyze Refactor Build Run Tools Git Window Help apache-tomcat-8.5.68-src - ClassLoaderFactory.java
apache catalina startup ClassLoaderFactory createClassLoader
Unnamed
Git:
Request.java addFilter_java Bootstrap.java ClassLoaderFactory.java URLClassLoader.java pom.xml (Tomcat8.5.
*/
public static ClassLoader createClassLoader(List<Repository> repositories,
final ClassLoader parent)
throws Exception {
if (log.isDebugEnabled()) {
log.debug("Creating new class loader");
}
// Construct the "class path" for this class loader
Set<URL> set = new LinkedHashSet<>();
if (repositories != null) {
for (Repository repository : repositories) {
if (repository.getType() == RepositoryType.URL) {
URL url = buildClassLoaderUrl(repository.getLocation());
if (log.isDebugEnabled()) {
log.debug(" Including URL " + url);
}
set.add(url);
} else if (repository.getType() == RepositoryType.DIR) {
File directory = new File(repository.getLocation());
directory = directory.getCanonicalFile();
if (!validateFile(directory, RepositoryType.DIR)) {
continue;
}
URL url = buildClassLoaderUrl(directory);
if (log.isDebugEnabled()) {
log.debug(" Including directory " + url);
}
set.add(url);
} else if (repository.getType() == RepositoryType.JAR) {
File file=new File(repository.getLocation());
file = file.getCanonicalFile();
if (!validateFile(file, RepositoryType.JAR)) {
continue;
}
URL url = buildClassLoaderUrl(file);
Ant build scripts found
Add Ant build file Skip Help

```
                    File file = new File(directory, s);
                    file = file.getCanonicalFile();
                    if (!validateFile(file, RepositoryType.JAR)) {
                        continue;
                    }
                    if (log.isDebugEnabled()) {
                        log.debug("    Including glob jar file "
                                + file.getAbsolutePath());
                    }
                    URL url = buildClassLoaderUrl(file);
                    set.add(url);
                }
            }
        }
    }

    // Construct the class loader itself
    final URL[] array = set.toArray(new URL[0]);
    if (log.isDebugEnabled()) {
        for (int i = 0; i < array.length; i++) {
            log.debug("  location " + i + " is " + array[i]);
        }
    }

    return AccessController.doPrivileged(
            new PrivilegedAction<URLClassLoader>() {
                @Override
                public URLClassLoader run() {
                    if (parent == null) {
                        return new URLClassLoader(array);
                    } else {
                        return new URLClassLoader(array, parent);
                    }
                }
            });
}
```

- 为什么tomcat要打破双亲委派模型

1.15 为什么 Tomcat 要破坏双亲委派模型？

Tomcat 是 web 容器，那么一个 web 容器可能需要部署多个应用程序；

1、部署在同一个 Tomcat 上的两个 Web 应用所使用的 Java 类库要相互隔离；

2、部署在同一个 Tomcat 上的两个 Web 应用所使用的 Java 类库要互相共享；

3、保证 Tomcat 服务器自身的安全不受部署的 Web 应用程序影响；

4、需要支持 JSP 页面的热部署和热加载；