Java & Tomcat ClassLoader 了解

入口点 main.c 文件源码

该文件位于 jdk/src/share/bin/main.c，其源码如下：

```
int
main(int argc, char **argv)
{
        …
        margc = argc;
        margv = argv;
        return JLI_Launch(margc, margv,
                                    sizeof(const_jargs) / sizeof(char *), const_jargs,
                                    sizeof(const_appclasspath) / sizeof(char *), const_appclasspath,
                                    FULL_VERSION,
                                    DOT_VERSION,
                                    (const_progname != NULL) ? const_progname : *margv,
                                    (const_launcher != NULL) ? const_launcher : *margv,
                                    (const_jargs != NULL) ? JNI_TRUE : JNI_FALSE,
                                    const_cpwildcard, const_javaw, const_ergo_class);
}
```

这里主要是处理一些平台相关参数并调用 JLI_Launch 函数。

JLI_Launch 函数

该文件位于 jdk/src/share/bin/java.c，其源码如下：

```
/*
 * Entry point.
 */
int
JLI_Launch(int argc, char ** argv,              /* main argc, argc */
        int jargc, const char** jargv,          /* java args */
        int appclassc, const char** appclassv,  /* app classpath */
        const char* fullversion,                /* full version defined */
        const char* dotversion,                 /* dot version defined */
        const char* pname,                      /* program name */
        const char* lname,                      /* launcher name */
        jboolean javaargs,                      /* JAVA_ARGS */
        jboolean cpwildcard,                    /* classpath wildcard*/
        jboolean javaw,                         /* windows-only javaw */
        jint ergo                               /* ergonomics class policy */
)
{
    int mode = LM_UNKNOWN;
    char *what = NULL;
    char *cpath = 0;
    char *main_class = NULL;
    int ret;
    InvocationFunctions ifn;
    jlong start, end;
    char jvmpath[MAXPATHLEN];
    char jrepath[MAXPATHLEN];
    char jvmcfg[MAXPATHLEN];

    _fVersion = fullversion;
    _dVersion = dotversion;
    _launcher_name = lname;
    _program_name = pname;
    _is_java_args = javaargs;
    _wc_enabled = cpwildcard;
    _ergo_policy = ergo;

    InitLauncher(javaw);
    DumpState();
    if (JLI_IsTraceLauncher()) {
        int i;
        printf("Command line args:\n");
        for (i = 0; i < argc ; i++) {
            printf("argv[%d] = %s\n", i, argv[i]);
        }
        AddOption("-Dsun.java.launcher.diag=true", NULL);
    }

    /*
     * Make sure the specified version of the JRE is running.
     *
     * There are three things to note about the SelectVersion() routine:
     *  1) If the version running isn't correct, this routine doesn't
     *     return (either the correct version has been exec'd or an error
     *     was issued).
     *  2) Argc and Argv in this scope are *not* altered by this routine.
     *     It is the responsibility of subsequent code to ignore the
     *     arguments handled by this routine.
     *  3) As a side-effect, the variable "main_class" is guaranteed to
     *     be set (if it should ever be set).  This isn't exactly the
     *     poster child for structured programming, but it is a small
     *     price to pay for not processing a jar file operand twice.
     *     (Note: This side effect has been disabled.  See comment on
     *     bugid 5030265 below.)
     */
    SelectVersion(argc, argv, &main_class);

    CreateExecutionEnvironment(&argc, &argv,
                               jrepath, sizeof(jrepath),
                                    jvmpath, sizeof(jvmpath),
                                    jvmcfg,   sizeof(jvmcfg));

    ifn.CreateJavaVM = 0;
    ifn.GetDefaultJavaVMInitArgs = 0;

    if (JLI_IsTraceLauncher()) {
        start = CounterGet();
    }

    if (!LoadJavaVM(jvmpath, &ifn)) {
        return(6);
    }

    if (JLI_IsTraceLauncher()) {
        end    = CounterGet();
    }

    JLI_TraceLauncher("%ld micro seconds to LoadJavaVM\n",
             (long)(jint)Counter2Micros(end-start));

    ++argv;
    --argc;

    if (IsJavaArgs()) {
        /* Preprocess wrapper arguments */
        TranslateApplicationArgs(jargc, jargv, &argc, &argv);
        if (!AddApplicationOptions(appclassc, appclassv)) {
            return(1);
        }
    } else {
        /* Set default CLASSPATH */
        cpath = getenv("CLASSPATH");
        if (cpath == NULL) {
            cpath = ".";
        }
        SetClassPath(cpath);
    }

    /* Parse command line options; if the return value of
     * ParseArguments is false, the program should exit.
     */
    if (!ParseArguments(&argc, &argv, &mode, &what, &ret, jrepath))
    {
        return(ret);
    }

    /* Override class path if -jar flag was specified */
    if (mode == LM_JAR) {
        SetClassPath(what);         /* Override class path */
    }

    /* set the -Dsun.java.command pseudo property */
    SetJavaCommandLineProp(what, argc, argv);

    /* Set the -Dsun.java.launcher pseudo property */
    SetJavaLauncherProp();

    /* set the -Dsun.java.launcher.* platform properties */
    SetJavaLauncherPlatformProps();

    return JVMInit(&ifn, threadStackSize, argc, argv, mode, what, ret);
}
```

JVMInit 函数

该函数位于 jdk/src/share/bin/java_md_macosx.c，其源码如下：

```
/* This class is made for performSelectorOnMainThread when java main
 * should be launched on main thread.
 * We cannot use dispatch_sync here, because it blocks the main dispatch queue
 * which is used inside Cocoa
 */
@interface JavaLaunchHelper : NSObject {   // 定义类
    int _returnValue;   // 属性
}
- (void) launchJava:(NSValue*)argsValue; // 方法
- (int) getReturnValue;
@end

@implementation JavaLaunchHelper

- (void) launchJava:(NSValue*)argsValue
{
    _returnValue = JavaMain([argsValue pointerValue]);   // 方法的具体实现
}

- (int) getReturnValue
{
    return _returnValue;
}

@end
```

上面代码是定义一个 JavaLaunchHelper 类，其 launchJava 函数是调用 JavaMain([argsValue pointerValue]); 函数。

```
// MacOSX we may continue in the same thread
int
JVMInit(InvocationFunctions* ifn, jlong threadStackSize,
                     int argc, char **argv,
                     int mode, char *what, int ret) {
    if (sameThread) {
        JLI_TraceLauncher("In same thread\n");
        // need to block this thread against the main thread
        // so signals get caught correctly
        JavaMainArgs args;
        args.argc = argc;
        args.argv = argv;
        args.mode = mode;
        args.what = what;
        args.ifn   = *ifn;
        int rslt;
        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        {
            JavaLaunchHelper* launcher = [[[JavaLaunchHelper alloc] init] autorelease];
            [launcher performSelectorOnMainThread:@selector(launchJava:)
                                      withObject:[NSValue valueWithPointer:(void*)&args]
                                   waitUntilDone:YES];
            rslt = [launcher getReturnValue];
        }
        [pool drain];
        return rslt;
    } else {
        return ContinueInNewThread(ifn, threadStackSize, argc, argv, mode, what, ret);
    }
}
```

上述代码调用了 JavaLaunchHelper 的 getReturnValue 函数，即最终调用 JavaMain 函数。

JavaMain 函数

该函数位于 jdk/src/share/bin/java.c，其源码如下：

```
int JNICALL
JavaMain(void * _args)
{
    JavaMainArgs *args = (JavaMainArgs *)_args;
    int argc = args->argc;
    char **argv = args->argv;
    int mode = args->mode;
    char *what = args->what;
    InvocationFunctions ifn = args->ifn;

    JavaVM *vm = 0;
```

```
JNIEnv *env = 0;
jclass mainClass = NULL;
jmethodID mainID;
jobjectArray mainArgs;
int ret = 0;
jlong start, end;

RegisterThread();

/* Initialize the virtual machine */
start = CounterGet();
if (!InitializeJVM(&vm, &env, &ifn)) {    // init & create JVM
        JLI_ReportErrorMessage(JVM_ERROR1);
        exit(1);
}

if (showSettings != NULL) {
        ShowSettings(env, showSettings);
        CHECK_EXCEPTION_LEAVE(1);
}

if (printVersion || showVersion) {
        PrintJavaVersion(env, showVersion);
        CHECK_EXCEPTION_LEAVE(0);
        if (printVersion) {
                LEAVE();
        }
}

/* If the user specified neither a class name nor a JAR file */
if (printXUsage || printUsage || what == 0 || mode == LM_UNKNOWN) {
        PrintUsage(env, printXUsage);
        CHECK_EXCEPTION_LEAVE(1);
        LEAVE();
}

FreeKnownVMs();    /* after last possible PrintUsage() */

if (JLI_IsTraceLauncher()) {
        end = CounterGet();
        JLI_TraceLauncher("%ld micro seconds to InitializeJVM\n",
                     (long)(jint)Counter2Micros(end-start));
}

/* At this stage, argc/argv have the application's arguments */
if (JLI_IsTraceLauncher()){
        int i;
        printf("%s is '%s'\n", launchModeNames[mode], what);
        printf("App's argc is %d\n", argc);
        for (i=0; i < argc; i++) {
            printf("        argv[%2d] = '%s'\n", i, argv[i]);
        }
    }
    ret = 1;

    /*
     * Get the application's main class.
     *
     * See bugid 5030265.  The Main-Class name has already been parsed
     * from the manifest, but not parsed properly for UTF-8 support.
     * Hence the code here ignores the value previously extracted and
     * uses the pre-existing code to reextract the value.  This is
     * possibly an end of release cycle expedient.  However, it has
     * also been discovered that passing some character sets through
     * the environment has "strange" behavior on some variants of
     * Windows.  Hence, maybe the manifest parsing code local to the
     * launcher should never be enhanced.
     *
     * Hence, future work should either:
     *     1)   Correct the local parsing code and verify that the
     *          Main-Class attribute gets properly passed through
     *          all environments,
     *     2)   Remove the vestages of maintaining main_class through
     *          the environment (and remove these comments).
     */
    mainClass = LoadMainClass(env, mode, what);
    CHECK_EXCEPTION_NULL_LEAVE(mainClass);
    PostJVMInit(env, mainClass, vm);
    /*
     * The LoadMainClass not only loads the main class, it will also ensure
     * that the main method's signature is correct, therefore further checking
     * is not required. The main method is invoked here so that extraneous java
     * stacks are not in the application stack trace.
     */
    mainID = (*env)->GetStaticMethodID(env, mainClass, "main",
                                       "([Ljava/lang/String;)V");
    CHECK_EXCEPTION_NULL_LEAVE(mainID);

    /* Build platform specific argument array */
    mainArgs = CreateApplicationArgs(env, argv, argc);
    CHECK_EXCEPTION_NULL_LEAVE(mainArgs);

    /* Invoke main method. */
    (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);

    /*
```

```
     * The launcher's exit code (in the absence of calls to
     * System.exit) will be non-zero if main threw an exception.
     */
    ret = (*env)->ExceptionOccurred(env) == NULL ? 0 : 1;
    LEAVE();
}
```

LoadMainClass 函数

该函数位于 jdk/src/share/bin/java.c，其源码如下：

```
/*
 * Loads a class and verifies that the main class is present and it is ok to
 * call it for more details refer to the java implementation.
 */
static jclass
LoadMainClass(JNIEnv *env, int mode, char *name)
{
    jmethodID mid;
    jstring str;
    jobject result;
    jlong start, end;
    jclass cls = GetLauncherHelperClass(env);
    NULL_CHECK0(cls);
    if (JLI_IsTraceLauncher()) {
        start = CounterGet();
    }
    NULL_CHECK0(mid = (*env)->GetStaticMethodID(env, cls,
                "checkAndLoadMain",
                "(ZILjava/lang/String;)Ljava/lang/Class;"));

    str = NewPlatformString(env, name);
    result = (*env)->CallStaticObjectMethod(env, cls, mid, USE_STDERR, mode, str);

    if (JLI_IsTraceLauncher()) {
        end   = CounterGet();
        printf("%ld micro seconds to load main class\n",
               (long)(jint)Counter2Micros(end-start));
        printf("----%s----\n", JLDEBUG_ENV_ENTRY);
    }

    return (jclass)result;
}
```

```
jclass
GetLauncherHelperClass(JNIEnv *env)
{
    if (helperClass == NULL) {
            NULL_CHECK0(helperClass = FindBootStrapClass(env,
                        "sun/launcher/LauncherHelper"));
    }
    return helperClass;
}
```

剩下就简单了

LauncherHelper. checkAndLoadMain

这是一个 java 方法，终于可以进入 JAVA 的世界了。

```
/**
     * This method does the following:
     * 1. gets the classname from a Jar's manifest, if necessary
     * 2. loads the class using the System ClassLoader
     * 3. ensures the availability and accessibility of the main method,
     *    using signatureDiagnostic method.
     *    a. does the class exist
     *    b. is there a main
     *    c. is the main public
     *    d. is the main static
     *    c. does the main take a String array for args
     * 4. and off we go......
     *
     * @param printToStderr
     * @param isJar
     * @param name
     * @return
     */
    public static Class<?> checkAndLoadMain(boolean printToStderr,
                                            int mode, String what) {
        final PrintStream ostream = (printToStderr) ? System.err : System.out;
        final ClassLoader ld = ClassLoader.getSystemClassLoader();
        // get the class name
        String cn = null;
        switch (mode) {
            case LM_CLASS:
                cn = what;
                break;
            case LM_JAR:
                cn = getMainClassFromJar(ostream, what);
                break;
            default:
                // should never happen
                throw new InternalError("" + mode + ": Unknown launch mode");
        }
        cn = cn.replace('/', '.');
        Class<?> c = null;
        try {
            c = ld.loadClass(cn);
        } catch (ClassNotFoundException cnfe) {
            abort(ostream, cnfe, "java.launcher.cls.error1", cn);
        }
        getMainMethod(ostream, c);
        return c;
    }
```

Java 在方法区中的类数据

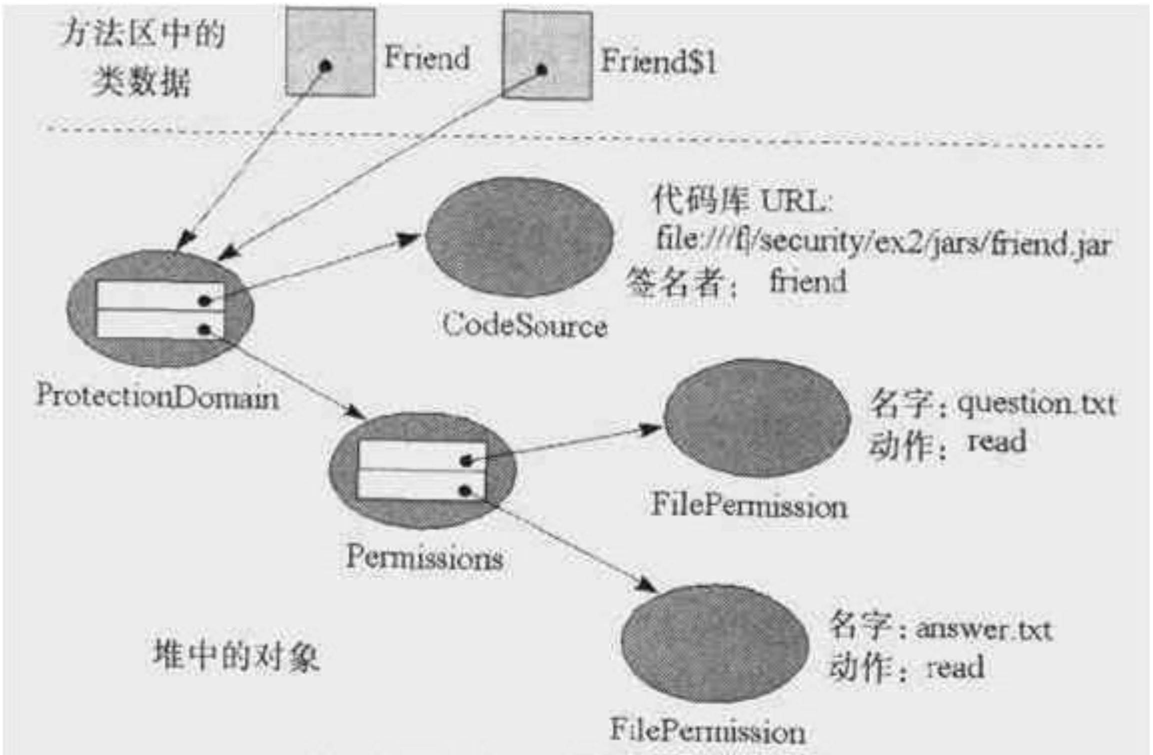


图3-5 保护域、代码来源以及权限

Tomcat 中 classloader 分析

Tomcat 在启动时定义了三个 classloader:

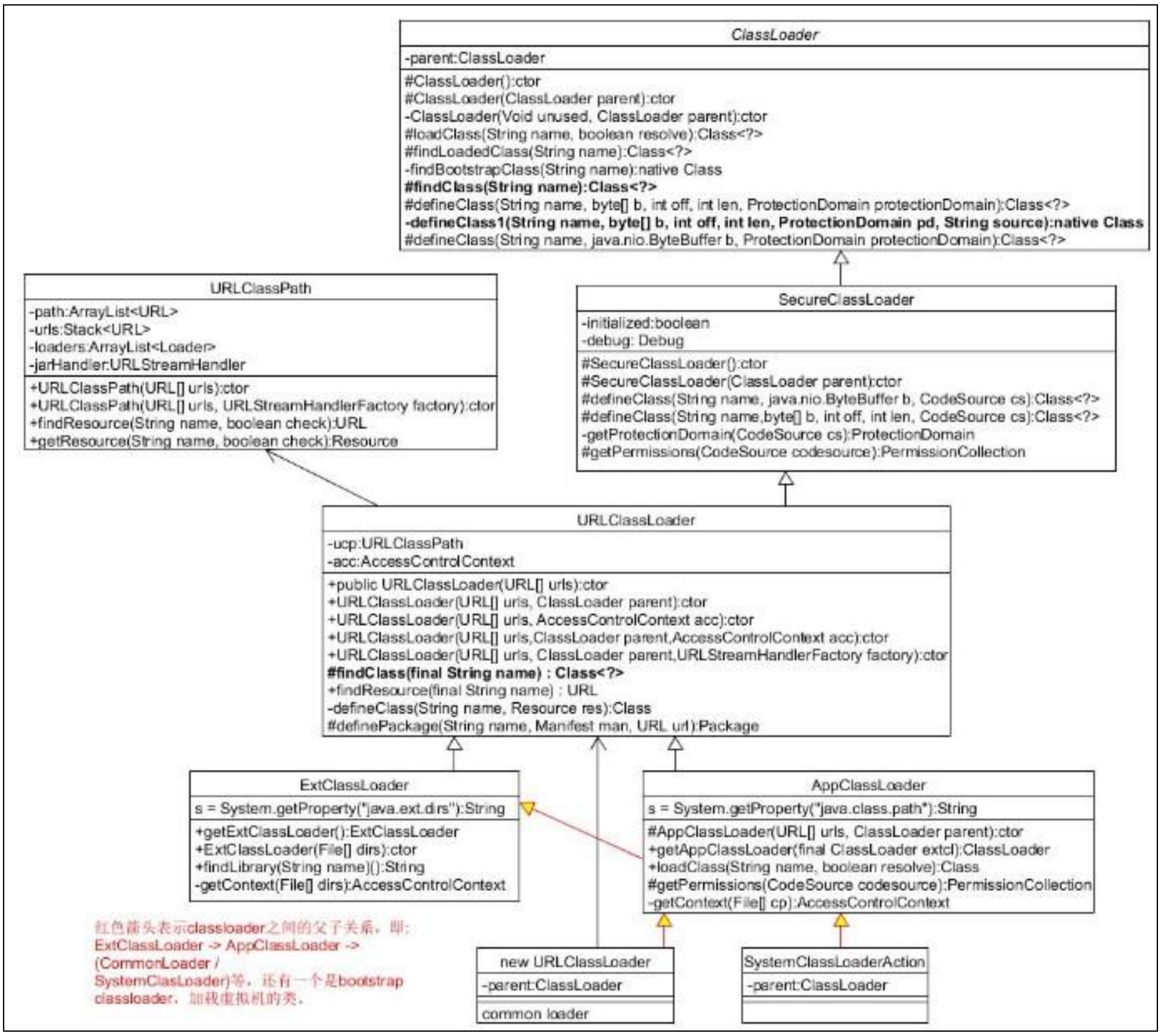
```
protected ClassLoader commonLoader = null;
protected ClassLoader catalinaLoader = null;
protected ClassLoader sharedLoader = null;
```

其中 commonLoader = createClassLoader("common", null); 其步骤如下:

1. 意思是读取 catalina.properties 的属性 common 值。

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

2. 将得到的值对应的目录(xx/lib)、文件(xx/lib/abc.jar) 和通配符(xx/lib/*.jar)转化为 List<Repository>数据。
3. 将 2 得到的 repositories 传入 ClassLoaderFactory.*createClassLoader*(repositories, parent); 这里 parent = null.
4. 将 repositories 转化为文件 / 目录对应的 url 数组。根据 parent 是否为 null，分别调用 new URLClassLoader(array) or new URLClassLoader(array, parent)得到 commonloader 对象.
5. URLClassLoader 类图如下。


ClassLoader
-parent:ClassLoader
#ClassLoader():ctor
#ClassLoader(ClassLoader parent):ctor
-ClassLoader(Void unused, ClassLoader parent):ctor
#loadClass(String name, boolean resolve):Class<?>
#findLoadedClass(String name):Class<?>
-findBootstrapClass(String name):native Class
#findClass(String name):Class<?>
#defineClass(String name, byte[] b, int off, int len, ProtectionDomain protectionDomain):Class<?>
-defineClass1(String name, byte[] b, int off, int len, ProtectionDomain pd, String source):native Class
#defineClass(String name, java.nio.ByteBuffer b, ProtectionDomain protectionDomain):Class<?>
URLClassPath
-path:ArrayList<URL>
-urls:Stack<URL>
-loaders:ArrayList<Loader>
-jarHandler:URLStreamHandler
+URLClassPath(URL[] urls):ctor
+URLClassPath(URL[] urls, URLStreamHandlerFactory factory):ctor
+findResource(String name, boolean check):URL
+getResource(String name, boolean check):Resource
SecureClassLoader
-initialized:boolean
-debug: Debug
#SecureClassLoader():ctor
#SecureClassLoader(ClassLoader parent):ctor
#defineClass(String name, java.nio.ByteBuffer b, CodeSource cs):Class<?>
#defineClass(String name,byte[] b, int off, int len, CodeSource cs):Class<?>
-getProtectionDomain(CodeSource cs):ProtectionDomain
#getPermissions(CodeSource codesource):PermissionCollection
URLClassLoader
-ucp:URLClassPath
-acc:AccessControlContext
+public URLClassLoader(URL[] urls):ctor
+URLClassLoader(URL[] urls, ClassLoader parent):ctor
+URLClassLoader(URL[] urls, AccessControlContext acc):ctor
+URLClassLoader(URL[] urls,ClassLoader parent,AccessControlContext acc):ctor
+URLClassLoader(URL[] urls, ClassLoader parent,URLStreamHandlerFactory factory):ctor
#findClass(final String name) : Class<?>
+findResource(final String name) : URL
-defineClass(String name, Resource res):Class
#definePackage(String name, Manifest man, URL url):Package
ExtClassLoader
s = System.getProperty("java.ext.dirs"):String
+getExtClassLoader():ExtClassLoader
+ExtClassLoader(File[] dirs):ctor
+findLibrary(String name)():String
-getContext(File[] dirs):AccessControlContext
AppClassLoader
s = System.getProperty("java.class.path"):String
#AppClassLoader(URL[] urls, ClassLoader parent):ctor
+getAppClassLoader(final ClassLoader extcl):ClassLoader
+loadClass(String name, boolean resolve):Class
#getPermissions(CodeSource codesource):PermissionCollection
-getContext(File[] cp):AccessControlContext
红色箭头表示classloader之间的父子关系，即：ExtClassLoader -> AppClassLoader -> (CommonLoader / SystemClasLoader)等。还有一个是bootstrap classloader，加载虚拟机的类。
new URLClassLoader
-parent:ClassLoader
common loader
SystemClassLoaderAction
-parent:ClassLoader

6. 从上述类图可知，commonloader 就是一个 URLClassLoader 对象，其继承自 ClassLoader，因此我们在自定义类加载器中只要考虑如何找到二进制文件(.class)并读入内存。具体将.class 转化为 Class<?>对象，可交由 ClassLoader 相关方法完成。
7. 因 parent = null, 分析 new URLClassLoader(array)代码。 其第一步是通过构造函数的 super() 不断调用父类构造函数。最终调用的代码为：

```
protected ClassLoader() {
        this(checkCreateClassLoader(), getSystemClassLoader());
    }
```

8. 上步 parent 为 null，我们会使用 *getSystemClassLoader*() 得到的 ClassLoader 作为 parent classloader. 该函数的核心代码如下(这段代码只执行一次，成功后设置为 ture,下次直接拿，不用再执行)：

```
sun.misc.Launcher l = sun.misc.Launcher.getLauncher();
if ( l != null) {
    scl = l.getClassLoader();
    …
    scl = AccessController.doPrivileged(new SystemClassLoaderAction(scl));
    …
}
```

含义为：通过 Launcher 得到 l.getClassLoader()，然后使用这个 classloader 来 new 一个 SystemClassLoaderAction 对象出来。

8.1 SystemClassLoaderAction 代码比较简单，如下：

```
class SystemClassLoaderAction implements PrivilegedExceptionAction<ClassLoader> {
    private ClassLoader parent;
    SystemClassLoaderAction(ClassLoader parent) {
        this.parent = parent;
    }
    public ClassLoader run() throws Exception {
        String cls = System.getProperty("java.system.class.loader");
        if (cls == null) {
            return parent;
        }
        Constructor ctor = Class.forName(cls, true, parent)
            .getDeclaredConstructor(new Class[] { ClassLoader.class });
        ClassLoader sys = (ClassLoader) ctor.newInstance(
            new Object[] { parent });
        Thread.currentThread().setContextClassLoader(sys);
        return sys;
    }
}
```

就是设置 parent 为传进来的 parent classloader, 因为是父 classloader 先加载，因此父类完成不了自己也就完成不了。当然有个 run 方法，可以根据 “java.system.class.loader” 变量通过反射来自定义自己的 classloader.

9. Launcher 是重点，这里定义了 ExtClassLoader 和 AppClassLoader 两个 JVM 自己的 classloader.

```
public Launcher() {
    …
    ClassLoader extcl = ExtClassLoader.getExtClassLoader();
    ClassLoader  loader = AppClassLoader.getAppClassLoader(extcl);
    Thread.currentThread().setContextClassLoader(loader);
}
```

9.1 ExtClassLoader 加载 System.*getProperty*("java.ext.dirs") 路径下的资源后通过 new ExtClassLoader 得到新的对象。

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

其中 factory 类似回调函数，可以在处理 ClassLoader 的 ucp 变量时（ucp = new URLClassPath(urls, factory);）通过 factory 指定 url 处理类型，如 file, jar 等，即如何将这些类型文件转成 InputStream，供后续处理。

9.2 ExtClassLoader 继承 URLClassLoader，代码如下：

```
public URLClassLoader(URL[] urls, ClassLoader parent, URLStreamHandlerFactory factory) {
        super(parent);
        // this is to make the stack depth consistent with 1.1
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkCreateClassLoader();
        }
        ucp = new URLClassPath(urls, factory);
        acc = AccessController.getContext();
    }
```

一个 URLClassPath 实例化如下：

```
public URLClassPath(URL[] urls, URLStreamHandlerFactory factory) {
        for (int i = 0; i < urls.length; i++) {
            path.add(urls[i]);
        }
        push(urls);
        if (factory != null) {
            jarHandler = factory.createURLStreamHandler("jar");
        }
    }
```

这里代码如下：

```
private static String PREFIX = "sun.net.www.protocol";

public URLStreamHandler createURLStreamHandler(String protocol) {
            String name = PREFIX + "." + protocol + ".Handler";
            try {
                Class c = Class.forName(name);
                return (URLStreamHandler)c.newInstance();
             …
```

9.3 最终在 ClassLoader 类里实例化该 classloader,设置 parent loader. 由 9.1 可知，其 parent 为 null.

9.4 AppClassLoader 实例化类似 ExtClassLoader，只是其 parent 为上步的 ExtClassLoader 对象。

注：ExtClassLoader 加载的资源在 String s = System.*getProperty*("java.ext.dirs"); 路径下。
AppClassLoader 加载的资源在 System.*getProperty*("java.class.path"); 路径下。

10. Launcher 的 getClassLoader()返回的是 loader，即 AppClassLoader 对象。
11. 有了上述 parent 后，就可以调用 ClassLoader(Void unused, ClassLoader parent) 构造函数实例化一个用户自定义的 classloader 啦。

总结一下 classloader 的父子关系：
URLClassLoader(common classloader) → AppClassLoader → ExtClassLoader → null
Bootstrap Classloader.

下面是一些 classloader 的信息：
```
java.system.class.loader: null
System class loader: sun.misc.Launcher$AppClassLoader@2259e205
loader : sun.misc.Launcher$AppClassLoader@2259e205
ploader : sun.misc.Launcher$ExtClassLoader@3b05c7e1
==============================================
commonLoader is java.net.URLClassLoader@1822b7f8
ploader : sun.misc.Launcher$AppClassLoader@2259e205
ploader : sun.misc.Launcher$ExtClassLoader@3b05c7e1
ploader : null
```

commonLoader.loadClass("org.apache.catalina.startup.Catalina") 分析

该代码位于 Bootstrap.java 的 init() 方法中。详细分析如下：

1. 该代码是通过 commonLoader 加载的，查询 URLClassLoader 的 loadClass(String name)如下。
2. 根据继承关系，loadClass 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); // (1)
            if (c == null) { // (2)
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false); // (2.1)
                    } else {
                        c = findBootstrapClassOrNull(name); // (2.2)
                    }
                } 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;
        }
    }
```

(1) 首先查找当前类加载器是否已加载了类

(2) 如果未找到，则

  (2.1) 如果父 classloader 存在，则循环调用父加载器从(1)继续查找。

  (2.2) 一直网上，直到 parent 的 classloader 为 null。 则调用 findBootstrapClassOrNull 在启动类 classloader 继续查找。

  如此这般循环，上述类似一个栈，从栈底到栈顶依次为：

```
URLClassLoader(p) -> AppClassLoader(p) -> ExtClassLoader(n)
                                                   Bootstrap ClassLoader
1->2->2.1            1->2->2.1            1->2->2.2       2.2 findBootstrapClassOrNull
```

其中(p)表示 parent 非 null,(n)表示 parent 为 null. BootstrapClassLoader 跟其他无直接父子关系，但经过上述可知，对于一个未加载过的类，查找的最上层即 Bootstrap classloader.

注： AppClassLoader 重载了 loadClass 方法，会对 class 所在的 package 做权限检查，然后继续调用父类的 loadClass 方法。

(3) 如果(2.2) 仍然未查找到需加载的类。说明该类未加载，且不是 bootstrap classloader 加载。

3. 经过 2 为找到加载的 class，则调用 findClass(String name)加载该 class.
   3.1 bootstrap classloader 未加载，则轮到 ExtClassLoader 的 findClass 执行。
   3.2 ExtClassLoader 本身无 findClass(String name)方法，则调用父类(URLClassLoader)的 **findClass** 方法，具体代码：

```
protected Class<?> findClass(final String name) throws ClassNotFoundException {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<Class>() {
                    public Class run() throws ClassNotFoundException {
                        String path = name.replace('.', '/').concat(".class");
                        Resource res = ucp.getResource(path, false);
                        if (res != null) {
                            try {
                                return defineClass(name, res);
                            } catch (IOException e) {
                                throw new ClassNotFoundException(name, e);
                            }
                        } else {
                            throw new ClassNotFoundException(name);
                        }
                    }
                }, acc);
        } catch (java.security.PrivilegedActionException pae) {
            throw (ClassNotFoundException) pae.getException();
        }
    }
```

代码含义为：根据 class 的 full name，将其转化为本地磁盘的相对路径的文件，这样就可以转化为 Resource，然后调用 defineClass(String name, Resource res)来具体加载了。

注： ExtClassLoader 的加载路径为 System.*getProperty*("java.ext.dirs");

   3.3 如果 ExtClassLoader 的 findClass 没找到 class 文件进行加载。则用 AppClassLoader 的 findClass 方法加载。这里 AppClassLoader 同样没有 findClass 方法，因此处理流程类似 ExtAppClassLoader，使用父类方法处理。

注： AppClassLoader 的加载路径为 System.*getProperty*("java.class.path");

   3.4 如果 AppClassLoader 的 findClass 没有找到 class 文件进行加载。则进入步骤 4.
   比较幸运的是 java.class.path 的路径包含 “D:\dev\workspace_luna\tomcat8015\target\classes”，在这个路径下可以找到对应的 class 文件，因此我们使用 AppClassLoader 对类进行加载。

4. 步骤 3 处理的一个细节就是调用 defineClass 对找到的 class 加载并生成 Class<?> 对象。根据类图可知仅 URLClassLoader 及其父类有这个方法。 而具体的 class loader 譬如 ExtClassLoader、AppClassLoader 以及自定义的 class loader 都不需要单独写这样的方法。 这样的好处是处理简单也不容易出错。

因此：自定义的 class loader 主要功能是如何找到相关的资源。 居然加载并转化为 Class 就交给父类处理。

5. URLClassLoader 的 defineClass 代码如下：

```
        处理 class 所在的 package 已经定义.
         …
        java.nio.ByteBuffer bb = res.getByteBuffer(); // 资源转化为 ByteBuffer 对象
        if (bb != null) { // 处理 Class 在方法区的一些属性。
            // Use (direct) ByteBuffer:
            CodeSigner[] signers = res.getCodeSigners();
            CodeSource cs = new CodeSource(url, signers);
            sun.misc.PerfCounter.getReadClassBytesTime().addElapsedTimeFrom(t0);
            return defineClass(name, bb, cs); // 交由父类 security classloader 处理
        } else {
            byte[] b = res.getBytes(); // 不是 ByteBuffer 类型，直接处理二进制
            // must read certificates AFTER reading bytes.
            CodeSigner[] signers = res.getCodeSigners();
            CodeSource cs = new CodeSource(url, signers);
            sun.misc.PerfCounter.getReadClassBytesTime().addElapsedTimeFrom(t0);
            return defineClass(name, b, 0, b.length, cs);
        }
```

父类 SecureClassLoader 主要增加了 ProtectionDomain 的信息，继续交由父类 ClassLoader 处理。

```
protected final Class<?> defineClass(String name, java.nio.ByteBuffer b, CodeSource cs) {
        return defineClass(name, b, getProtectionDomain(cs));
    }

Or

protected final Class<?> defineClass(String name, byte[] b, int off, int len,CodeSource cs) {
        return defineClass(name, b, off, len, getProtectionDomain(cs));
    }
```

6. ClassLoader 的 defineClass 处理

```
protected final Class<?> defineClass(String name, byte[] b, int off, int len,
                    ProtectionDomain protectionDomain) throws ClassFormatError {
        // 1. 确认非 java. 开头，且 name是否合理，如： z, I, [java/lang/Object; 等。
        // 2. 如果 ProtectionDomain 为null,则给个默认值
        // 3. 根据name得到其package,查找package的证书，如果第一次则put到map里。将package查到的
        //     证书和class的证书比较，应该相等。否则抛出异常。
        protectionDomain = preDefineClass(name, protectionDomain);

        Class c = null;
        // 得到文件所在目录or路径
        // eg: file:/D:/dev/workspace_luna/tomcat8015/target/classes/
        String source = defineClassSourceLocation(protectionDomain);

        try {
            c = defineClass1(name, b, off, len, protectionDomain, source);
        } catch (ClassFormatError cfe) {
            c = defineTransformedClass(name, b, off, len, protectionDomain, cfe,
                                        source);
        }

        postDefineClass(c, protectionDomain); // 设置 class的 signer.
        return c;
    }
```

上述 defineClass1 就是调用本地 C 写的 native 方法。

7. 本地方法调用 defineClass1 源码如下：

```
JNIEXPORT jclass JNICALL
Java_java_lang_ClassLoader_defineClass1(JNIEnv *env,
                                        jobject loader,
                                        jstring name,
                                        jbyteArray data,
                                        jint offset,
                                        jint length,
                                        jobject pd,
                                        jstring source)
{
    jbyte *body;
    char *utfName;
    jclass result = 0;
    char buf[128];
    char* utfSource;
    char sourceBuf[1024];

    if (data == NULL) {
        JNU_ThrowNullPointerException(env, 0);
        return 0;
    }

    /* Work around 4153825. malloc crashes on Solaris when passed a
     * negative size.
     */
    if (length < 0) {
        JNU_ThrowArrayIndexOutOfBoundsException(env, 0);
        return 0;
    }
    // 分配内存空间
    body = (jbyte *)malloc(length);

    if (body == 0) { // 没有内存可分配，则抛出 OOM 异常
        JNU_ThrowOutOfMemoryError(env, 0);
        return 0;
    }
```

```
将Byte类型数组某一区域复制到缓冲区中，参数说明如下：
  env: JNI 接口指针
  data: Java 指针
  offset:: 起始下标
  length: 要复制的元素个数
  body: 目的缓存区
(*env)->GetByteArrayRegion(env, data, offset, length, body);

if ((*env)->ExceptionOccurred(env))
    goto free_body;

if (name != NULL) {
    // 将 name 转换为unicode 的字符串
    utfName = getUTF(env, name, buf, sizeof(buf));
    // 类名为空，表示JVM内存分配失败，说明没有内存了，抛出OOM 异常
    if (utfName == NULL) {
        JNU_ThrowOutOfMemoryError(env, NULL);
        goto free_body;
    }
    VerifyFixClassname(utfName);
} else {
    utfName = NULL;
}

if (source != NULL) {
    // 文件路径同样转unicode格式，且不成功就抛出OOM异常。
    utfSource = getUTF(env, source, sourceBuf, sizeof(sourceBuf));
    if (utfSource == NULL) {
        JNU_ThrowOutOfMemoryError(env, NULL);
        goto free_utfName;
    }
} else {
    utfSource = NULL;
}
// C定义的jclass即Java 的 Class对象
result = JVM_DefineClassWithSource(env, utfName, loader, body, length, pd, utfSource);

if (utfSource && utfSource != sourceBuf)
```

```
        free(utfSource);

 free_utfName:
    if (utfName && utfName != buf)
        free(utfName);

 free_body:
    free(body);
    return result; // 返回 jclass 对象
}
```

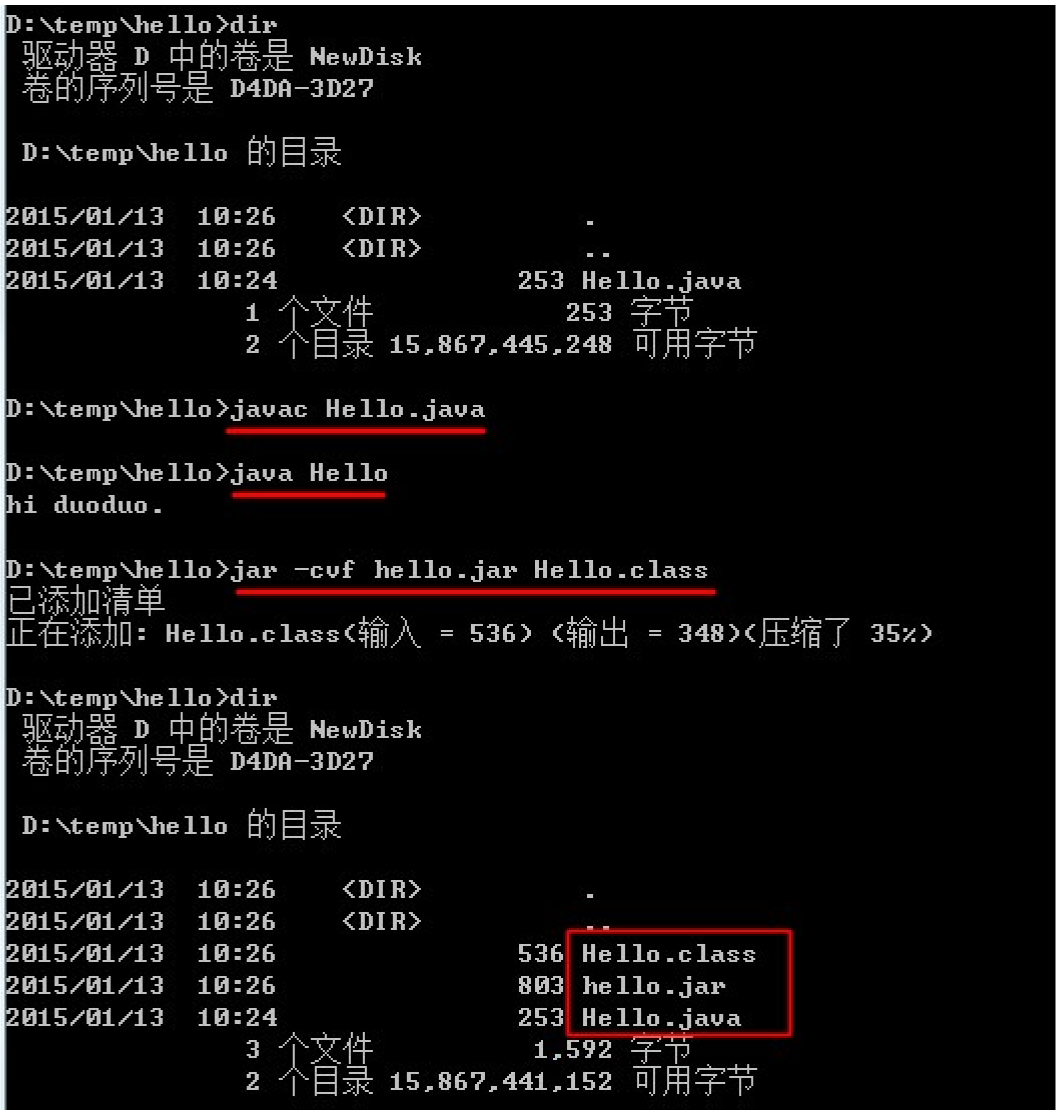
何时真正使用 comm class loader 加载 class 呢？

1. 准备

   编写自己的代码 Hello.java 如下：

```
public class Hello {
    public String say() {
        System.out.println("hi duoduo.");
        return "hi duoduo.";
    }
    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.say();
    }
}
```

   代码很简单，处理命令如下，分别得到 Hello.class 和 hello.jar 包。



```
D:\temp\hello>dir
 驱动器 D 中的卷是 NewDisk
 卷的序列号是 D4DA-3D27

 D:\temp\hello 的目录

2015/01/13  10:26    <DIR>          .
2015/01/13  10:26    <DIR>          ..
2015/01/13  10:24               253 Hello.java
               1 个文件            253 字节
               2 个目录 15,867,445,248 可用字节

D:\temp\hello>javac Hello.java

D:\temp\hello>java Hello
hi duoduo.

D:\temp\hello>jar -cvf hello.jar Hello.class
已添加清单
正在添加: Hello.class(输入 = 536) (输出 = 348)(压缩了 35%)

D:\temp\hello>dir
 驱动器 D 中的卷是 NewDisk
 卷的序列号是 D4DA-3D27

 D:\temp\hello 的目录

2015/01/13  10:26    <DIR>          .
2015/01/13  10:26    <DIR>          ..
2015/01/13  10:26               536 Hello.class
2015/01/13  10:26               803 hello.jar
2015/01/13  10:24               253 Hello.java
               3 个文件          1,592 字节
               2 个目录 15,867,441,152 可用字节
```

2. 将 hello.jar 扔到 $TOMCAT_HOME/lib 中。
3. 执行代码

```
Class<?> helloClass = commonLoader.loadClass("Hello");
```

```
System.out.println("Hello is loaded by " + helloClass.getClassLoader());
```

这里 loadClass 的步骤和上例一样，只是到 AppClassLoader 时还没在给定的路径找到 Hello.class 文件，因此进入 comm class loader 对应的路径查找 Hello.class 文件。 根据上述配置文件可知， comm class loader 查询 $TOMCAT_HOME/lib 目录，这样可以找到 hello.jar 并通过 “sun.net.www.protocol.jar.Handler”处理 jar 文件来加载里面的 Hello.class.

4. 相关代码及执行结果

```
System.out.println("commonLoader : " + commonLoader);
System.out.println("commonLoader parent : " + commonLoader.getParent());

Class<?> helloClass = commonLoader.loadClass("Hello");
System.out.println("Hello is loaded by " + helloClass.getClassLoader());

Class<?> startupClass = commonLoader.loadClass("org.apache.catalina.startup.Catalina");
System.out.println("Catalina is loaded by " + startupClass.getClassLoader());
```

输出如下：

```
commonLoader : java.net.URLClassLoader@51a23566
commonLoader parent : sun.misc.Launcher$AppClassLoader@57f530d8
Hello is loaded by java.net.URLClassLoader@51a23566
Catalina is loaded by sun.misc.Launcher$AppClassLoader@57f530d8
```

事实上，上节因为是在 eclipse 里调试 tomcat 源码，所以才有了 java.class.path 的路径包含“D:\dev\workspace_luna\tomcat8015\target\classes”，最终 Catalina 通过 AppClassLoader 来加载的。在线上的 tomcat 项目中，指定 $TOMCAT_HOME 后， $TOMCAT_HOME/lib 会落在 common loader 加载的路径内，通过查询 tomcat-xxx.jar 可以找到其中的 Cataline.class，使用 common loader 来加载和实例化 tomcat 对应的资源和对象。

编写自己的 ClassLoader

经过上述分析。已经对 classloader 基本了解了。写一个自己的试试看。 其实主要就是确定从哪里加载神马样子的资源。 简单起见，从 tomcat 根目录加载 Hello.class 资源即可。

1. 修改 conf/catalina.properties，加入如下参数。 表示从 $TOMCAT_HOME/jack 目录加载资源。
   `jack.loader="jack"`
2. Tomcat 项目的 Bootstrap.java 定义 classload 变量
   `protected ClassLoader jackLoader = null;`
3. initClassLoaders 函数增加创建 jackLoader 的代码
   `jackLoader = createJackClassLoader("jack", null);`
4. createJackClassLoader 函数具体代码如下：

```
private ClassLoader createJackClassLoader(String name,  final ClassLoader parent) {
        String value = CatalinaProperties.getProperty(name + ".loader");
        if ((value == null) || (value.equals("")))
            return parent;

        System.out.println("value : " + value);

        Set<URL> set = new LinkedHashSet<>();
        String[] repositoryPaths = getPaths(value);
        for (int i = 0; i < repositoryPaths.length; i++) {
            File directory = new File(repositoryPaths[i]);
            try {
                directory = directory.getCanonicalFile();
                URL url = directory.toURI().toURL();
                System.out.println("url: " + url);
                set.add(url);
            } catch (IOException e) {
            }
        }

        final URL[] array = set.toArray(new URL[set.size()]);


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

即将定义的目录转化为 URL[]，然后传入 URLClassLoader 构造函数来 new 一个新的 ClassLoader 对象。

5. 创建 $TOMCAT_HOME/jack 目录，将前文得到的 Hello.class 文件拷贝到该目录。
6. 测试代码和结果如下
   代码：

```
System.out.println("commonLoader : " + commonLoader);
System.out.println("commonLoader parent : " + commonLoader.getParent());

Class<?> helloClass = commonLoader.loadClass("Hello");
System.out.println("Hello is loaded by " + helloClass.getClassLoader());

System.out.println("jackLoader : " + jackLoader);
System.out.println("jackLoader parent : " + jackLoader.getParent());
Class<?> jackHelloClass = jackLoader.loadClass("Hello");
System.out.println("jackHelloClass is loaded by " + jackHelloClass.getClassLoader());

Class<?> startupClass = commonLoader.loadClass("org.apache.catalina.startup.Catalina");
System.out.println("Catalina is loaded by " + startupClass.getClassLoader());
```

   结果：

```
commonLoader : java.net.URLClassLoader@6cb05409
commonLoader parent : sun.misc.Launcher$AppClassLoader@57f530d8

Hello is loaded by java.net.URLClassLoader@6cb05409

jackLoader : java.net.URLClassLoader@3f4e8936
jackLoader parent : sun.misc.Launcher$AppClassLoader@57f530d8
jackHelloClass is loaded by java.net.URLClassLoader@3f4e8936

Catalina is loaded by sun.misc.Launcher$AppClassLoader@57f530d8
```

7. 将 Hello.class 从$TOMCAT_HOME/jack 目录移除，再次执行步骤 6，会得到如下异常。进一步说明用户自定义的 class loader 是从 $TOMCAT_HOME/jack 加载 Hello.class 的。

```
jackLoader parent : sun.misc.Launcher$AppClassLoader@2259e205
java.lang.ClassNotFoundException: Hello
    at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
    at org.apache.catalina.startup.Bootstrap.init(Bootstrap.java:372)
    at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:565)
```

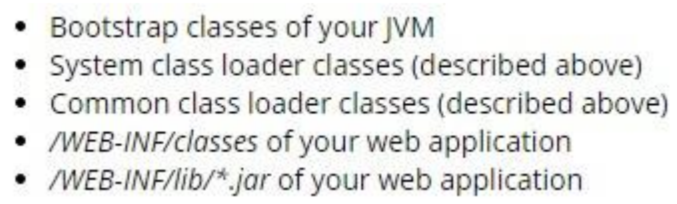
Tomcat WebApp 的 ClassLoader

前面所说的 class loader 都是按照传统父类先加载的顺序查找.class 文件并加载的原则。如果在 tomcat 中部署了多个 webapp,加载 class 资源可以通过如下两种途径之一：

- 将每个 webapp 资源（eg: webapps/WebX/classes, webapps/Webx/lib）设置到 classpath 中，对每个新增的资源需要每次设置，极其不方便。
- 设置类似通配符的匹配方式，还是限制了灵活性。

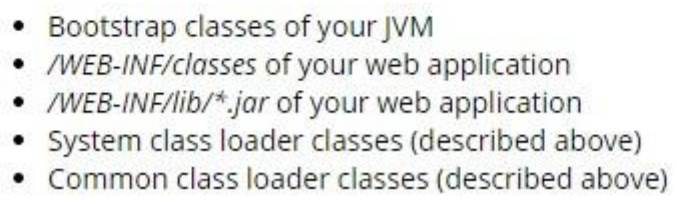
不管那种方式，按照父类先加载的原则，这些 webapp 都会通过相同的 classloader 加载 class 资源。因此各 webapp 之间的对象可能可以互相访问（classloader 相同情况下，public 等是可以互相访问的)。

解决这个问题的办法是，引入 Thread classloader 概念，为每个 WebX 启动一个线程，同时定义该线程的的 context classloader，**默认情况下为 System classloader**. 这样我们可以认为加载 classloader 的顺序从



- Bootstrap classes of your JVM
- System class loader classes (described above)
- Common class loader classes (described above)
- */WEB-INF/classes* of your web application
- */WEB-INF/lib/*.jar* of your web application

变成



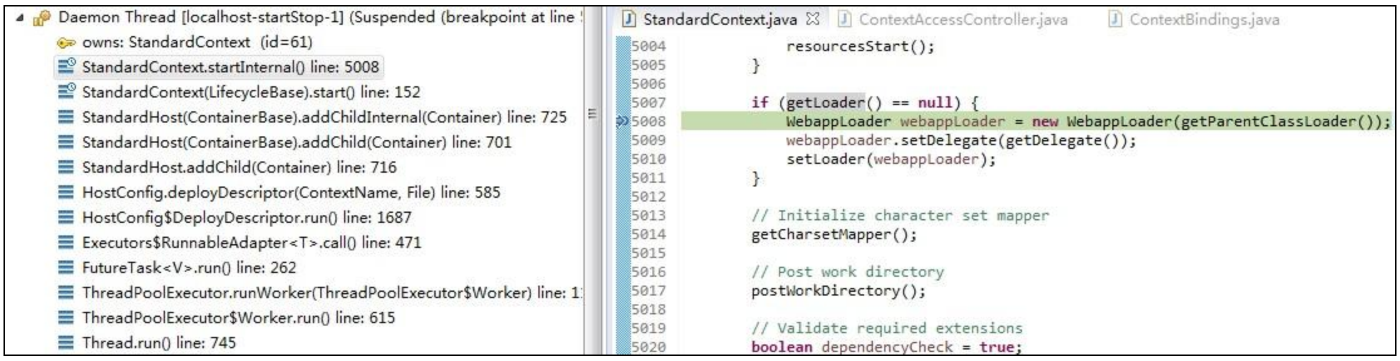
- Bootstrap classes of your JVM
- */WEB-INF/classes* of your web application
- */WEB-INF/lib/*.jar* of your web application
- System class loader classes (described above)
- Common class loader classes (described above)

针对 Tomcat 实际工作步骤应该如下(2,3 通常选一, 目的是为了创建 4 即 thread context class loader)，一旦建好后，都是先从 thead context classloader 开始找 classes. 即：

- Bootstrap classloader → JVM 相关的 classes 加载
- Ext/App classloader→即 Ext 和 System classloader （eclipse debug 时加载 tomcat 相关 classes）
- Tomcat Common classloader → 即 Tomcat 相关 classes （实际线上是通过该步骤加载的）
- Thread context classloader → 加载该线程定义的 classes (Tomcat 为 WebappClassloader)，即 /WEB-INF/classes 和 /WEB-INF/lib/*.jar 资源。
- Ext / App 等 classloader 加载 clases.

根据 tomcat 设计，每个 web 都是 host container 的 StandardContext child,加载的时候是通过开开一个线程启动的，设置的 classloade 为 WebappClassloader 即可，这样每个 WebX 加载的 webapp 自身的资源(WEB-INF/classes, WEB-INF/lib/*.jar)就可以隔离开来了。具体实现过程如下：

1. 设置 web app 的 classloader 代码如下：



```
Daemon Thread [localhost-startStop-1] (Suspended (breakpoint at line
    owns: StandardContext  (id=61)
    StandardContext.startInternal() line: 5008
    StandardContext(LifecycleBase).start() line: 152
    StandardHost(ContainerBase).addChildInternal(Container) line: 725
    StandardHost(ContainerBase).addChild(Container) line: 701
    StandardHost.addChild(Container) line: 716
    HostConfig.deployDescriptor(ContextName, File) line: 585
    HostConfig$DeployDescriptor.run() line: 1687
    Executors$RunnableAdapter<T>.call() line: 471
    FutureTask<V>.run() line: 262
    ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1
    ThreadPoolExecutor$Worker.run() line: 615
    Thread.run() line: 745
```

StandardContext.java ContextAccessController.java ContextBindings.java

```
5004            resourcesStart();
5005        }
5006
5007        if (getLoader() == null) {
5008            WebappLoader webappLoader = new WebappLoader(getParentClassLoader());
5009            webappLoader.setDelegate(getDelegate());
5010            setLoader(webappLoader);
5011        }
5012
5013        // Initialize character set mapper
5014        getCharsetMapper();
5015
5016        // Post work directory
5017        postWorkDirectory();
5018
5019        // Validate required extensions
5020        boolean dependencyCheck = true;
```

Host 启动的时候，触发 HostConfig listener 调用如下代码：

```
/**
 * Deploy applications for any directories or WAR files that are found
 * in our "application root" directory.
 */
protected void deployApps() {
    File appBase = host.getAppBaseFile();
    File configBase = host.getConfigBaseFile();
    String[] filteredAppPaths = filterAppPaths(appBase.list());
    // Deploy XML descriptors from configBase
    deployDescriptors(configBase, configBase.list());
    // Deploy WARs
    deployWARs(appBase, filteredAppPaths);
    // Deploy expanded folders
    deployDirectories(appBase, filteredAppPaths);
}
```

如上图左边在新线程执行 HostConfig$DeployDecriptor.run()方法，这是内部类的新线程，执行了外部类 HostConfig. deployDescriptor()方法。参考上述调用栈，针对每个 webapp，创建一个 StandardContext 表示，这是实现了 LifeCycle 方法。在其生命周期的 start()方法中，实现了如上方法。

2. 第一次执行到 1 图示的代码中，StandardContext.loader = null,因此执行：

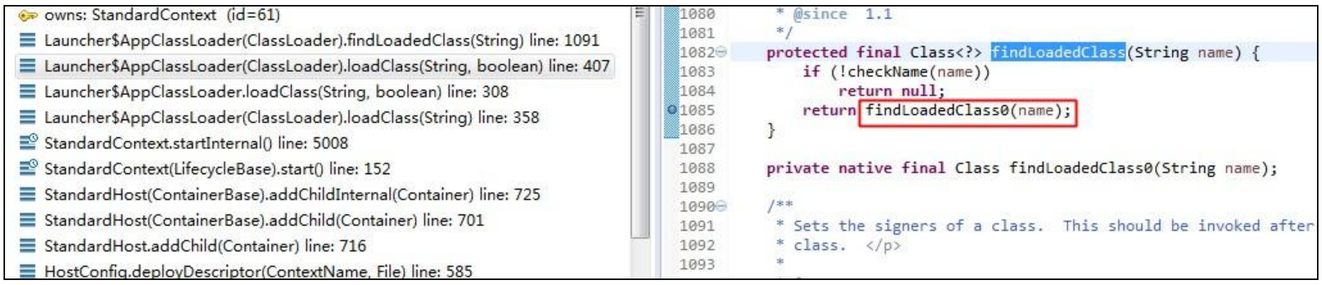
```
/**
 * The Loader implementation with which this Container is associated.
 */
private Loader loader = null;


…


if (getLoader() == null) {
    WebappLoader webappLoader = new WebappLoader(getParentClassLoader());
    webappLoader.setDelegate(getDelegate());
    setLoader(webappLoader);
}
```

2.1 针对 new WebappLoader(),这是第一次调用，因此我们需要通过 classloader 加载。目前的 classloader 如下：

Thread.currentThread().getContextClassLoader(); ➔ java.net.URLClassLoader@67826710

其实这就是 tomcat 的 commonloader 内容 （commonloader: java.net.URLClassLoader@67826710）

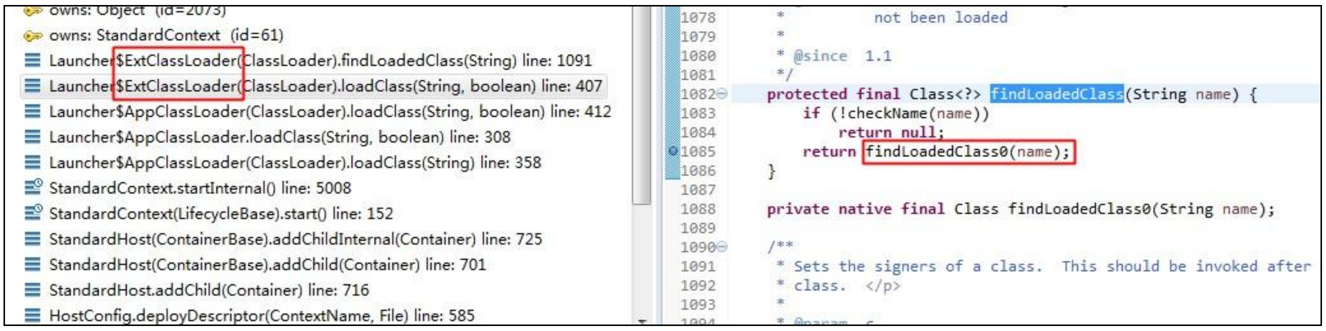
2.2 方法调用栈如下：



```
owns: StandardContext  (id=61)
Launcher$AppClassLoader(ClassLoader).findLoadedClass(String) line: 1091
Launcher$AppClassLoader(ClassLoader).loadClass(String, boolean) line: 407
Launcher$AppClassLoader.loadClass(String, boolean) line: 308
Launcher$AppClassLoader(ClassLoader).loadClass(String) line: 358
StandardContext.startInternal() line: 5008
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
```

```
     * @since  1.1
     */
    protected final Class<?> findLoadedClass(String name) {
        if (!checkName(name))
            return null;
        return findLoadedClass0(name);
    }

    private native final Class findLoadedClass0(String name);

    /**
     * Sets the signers of a class.  This should be invoked after
     * class.  </p>
     *
```

继续

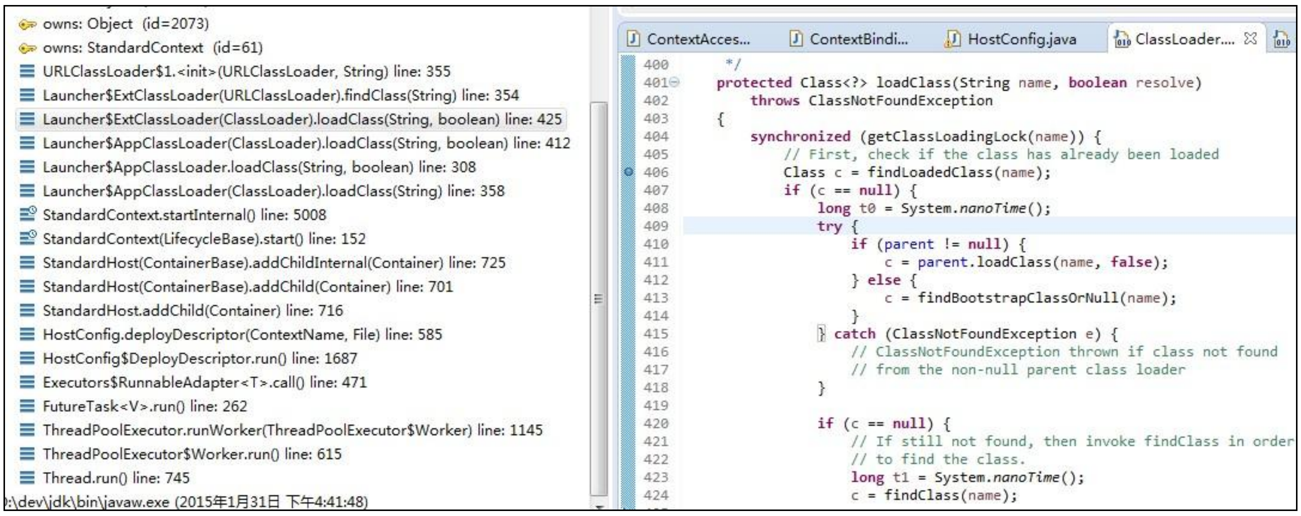


```
owns: Object  (id=2073)
owns: StandardContext  (id=61)
Launcher$ExtClassLoader(ClassLoader).findLoadedClass(String) line: 1091
Launcher$ExtClassLoader(ClassLoader).loadClass(String, boolean) line: 407
Launcher$AppClassLoader(ClassLoader).loadClass(String, boolean) line: 412
Launcher$AppClassLoader.loadClass(String, boolean) line: 308
Launcher$AppClassLoader(ClassLoader).loadClass(String) line: 358
StandardContext.startInternal() line: 5008
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
```

```java
     *         not been loaded
     *
     * @since  1.1
     */
    protected final Class<?> findLoadedClass(String name) {
        if (!checkName(name))
            return null;
        return findLoadedClass0(name);
    }

    private native final Class findLoadedClass0(String name);

    /**
     * Sets the signers of a class.  This should be invoked after
     * class.  </p>
     *
```

看到不同点了吗？ 从 AppClassLoader 变成 ExtClassLoader.

2.3 如 2.2 执行，我们看到详细加载 class 代码如下：

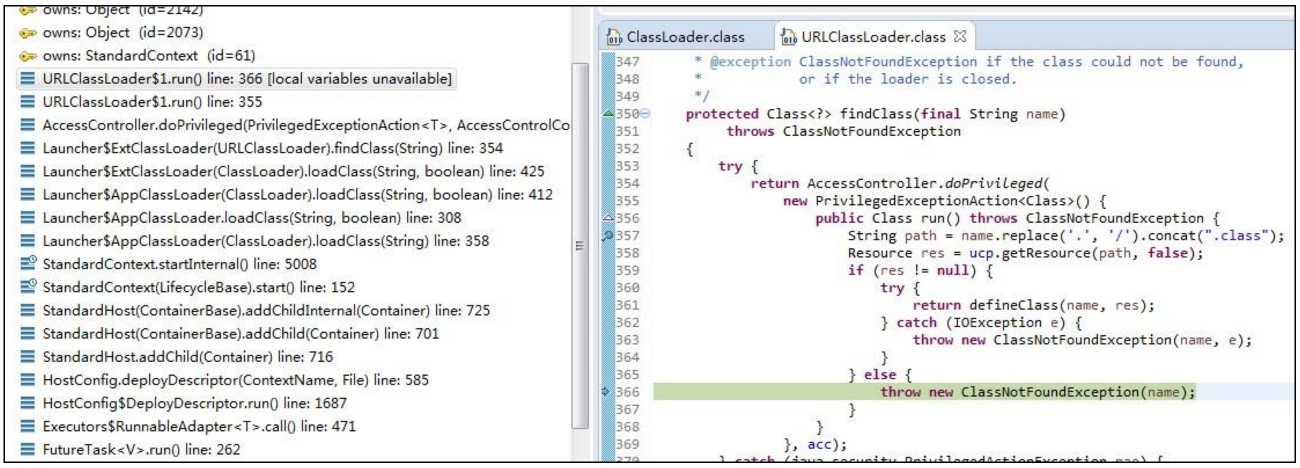


```
owns: Object  (id=2073)
owns: StandardContext  (id=61)
URLClassLoader$1.<init>(URLClassLoader, String) line: 355
Launcher$ExtClassLoader(URLClassLoader).findClass(String) line: 354
Launcher$ExtClassLoader(ClassLoader).loadClass(String, boolean) line: 425
Launcher$AppClassLoader(ClassLoader).loadClass(String, boolean) line: 412
Launcher$AppClassLoader.loadClass(String, boolean) line: 308
Launcher$AppClassLoader(ClassLoader).loadClass(String) line: 358
StandardContext.startInternal() line: 5008
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
HostConfig$DeployDescriptor.run() line: 1687
Executors$RunnableAdapter<T>.call() line: 471
FutureTask<V>.run() line: 262
ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1145
ThreadPoolExecutor$Worker.run() line: 615
Thread.run() line: 745
:\dev\jdk\bin\javaw.exe (2015年1月31日 下午4:41:48)
```

ContextAcces... ContextBindi... HostConfig.java ClassLoader....

```java
400      */
401     protected Class<?> loadClass(String name, boolean resolve)
402         throws ClassNotFoundException
403     {
404         synchronized (getClassLoadingLock(name)) {
405             // First, check if the class has already been loaded
406             Class c = findLoadedClass(name);
407             if (c == null) {
408                 long t0 = System.nanoTime();
409                 try {
410                     if (parent != null) {
411                         c = parent.loadClass(name, false);
412                     } else {
413                         c = findBootstrapClassOrNull(name);
414                     }
415                 } catch (ClassNotFoundException e) {
416                     // ClassNotFoundException thrown if class not found
417                     // from the non-null parent class loader
418                 }
419
420                 if (c == null) {
421                     // If still not found, then invoke findClass in order
422                     // to find the class.
423                     long t1 = System.nanoTime();
424                     c = findClass(name);
```

根据图示 409-418 代码，加载 class WebappLoader 先后通过 AppClassLoader -> ExtClassLoader -> Bootstrap 这几个 classloader，但是还没找到。 这时候停留在 ExtClassLoader 对象上，因此在 424 行调用了 ExtClassLoader 的 findClass(name)方法。其执行结果如下：

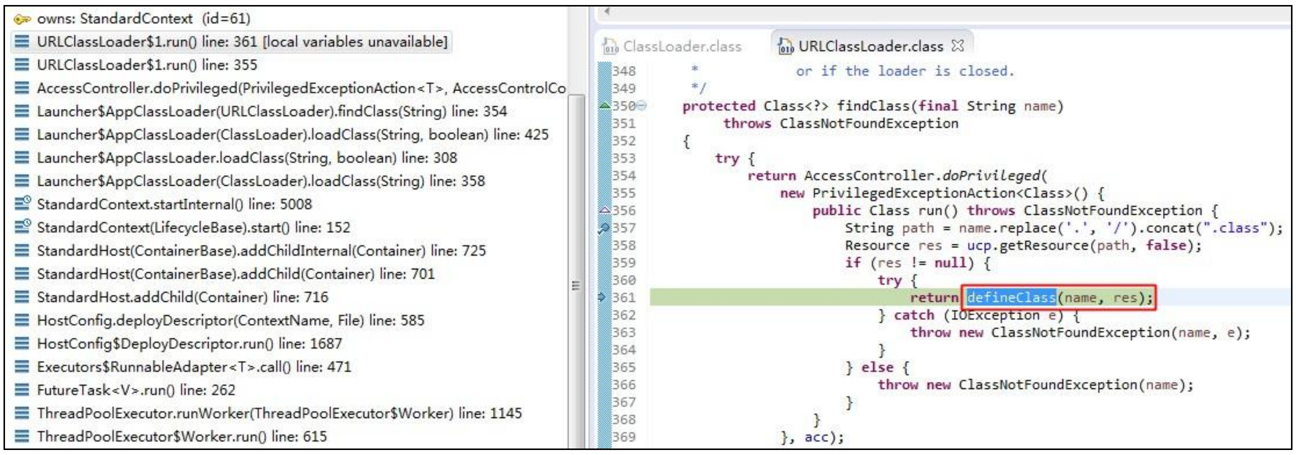


```
owns: Object  (id=2142)
owns: Object  (id=2073)
owns: StandardContext  (id=61)
URLClassLoader$1.run() line: 366 [local variables unavailable]
URLClassLoader$1.run() line: 355
AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlCo
Launcher$ExtClassLoader(URLClassLoader).findClass(String) line: 354
Launcher$ExtClassLoader(ClassLoader).loadClass(String, boolean) line: 425
Launcher$AppClassLoader(ClassLoader).loadClass(String, boolean) line: 412
Launcher$AppClassLoader.loadClass(String, boolean) line: 308
Launcher$AppClassLoader(ClassLoader).loadClass(String) line: 358
StandardContext.startInternal() line: 5008
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
HostConfig$DeployDescriptor.run() line: 1687
Executors$RunnableAdapter<T>.call() line: 471
FutureTask<V>.run() line: 262
```

ClassLoader.class URLClassLoader.class

```java
     * @exception ClassNotFoundException if the class could not be found,
     *            or if the loader is closed.
     */
    protected Class<?> findClass(final String name)
         throws ClassNotFoundException
    {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<Class>() {
                    public Class run() throws ClassNotFoundException {
                        String path = name.replace('.', '/').concat(".class");
                        Resource res = ucp.getResource(path, false);
                        if (res != null) {
                            try {
                                return defineClass(name, res);
                            } catch (IOException e) {
                                throw new ClassNotFoundException(name, e);
                            }
                        } else {
                            throw new ClassNotFoundException(name);
                        }
                    }
                }, acc);
```

表示通过 ExtClassLoader 没有找到其加载路径里有 WebappLoader 对应的 class 文件。继续使用 AppClassLoader 加载该文件，执行结果如下：



```
owns: StandardContext  (id=61)
URLClassLoader$1.run() line: 361 [local variables unavailable]
URLClassLoader$1.run() line: 355
AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlCo
Launcher$AppClassLoader(URLClassLoader).findClass(String) line: 354
Launcher$AppClassLoader(ClassLoader).loadClass(String, boolean) line: 425
Launcher$AppClassLoader.loadClass(String, boolean) line: 308
Launcher$AppClassLoader(ClassLoader).loadClass(String) line: 358
StandardContext.startInternal() line: 5008
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
HostConfig$DeployDescriptor.run() line: 1687
Executors$RunnableAdapter<T>.call() line: 471
FutureTask<V>.run() line: 262
ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1145
ThreadPoolExecutor$Worker.run() line: 615
```

```
     *          or if the loader is closed.
     */
    protected Class<?> findClass(final String name)
         throws ClassNotFoundException
    {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<Class>() {
                    public Class run() throws ClassNotFoundException {
                        String path = name.replace('.', '/').concat(".class");
                        Resource res = ucp.getResource(path, false);
                        if (res != null) {
                            try {
                                return defineClass(name, res);
                            } catch (IOException e) {
                                throw new ClassNotFoundException(name, e);
                            }
                        } else {
                            throw new ClassNotFoundException(name);
                        }
                    }
                }, acc);
```

这里可以看到在其路径找到了对应 class 文件，调用 defineClass()方法，最终使用底层 C 写的代码将.class 文件加载到方法区并生成对应的 Class<?>对象。

2.4 当前该 Thead 的 classloader 如 2.1 所示。

2.5 WebappLoader 有下面定义：

```
private WebappClassLoaderBase classLoader = null;
```

因此我们也要通过 Thead 的 context classloader 加载该类，具体加载过程跟 WebappLoader 相同。

3. 针对步骤 2 的 getParentClassLoader 方法，其代码如下：

```
/**
     * Return the parent class loader (if any) for this web application.
     * This call is meaningful only <strong>after</strong> a Loader has
     * been configured.
     */
    @Override
    public ClassLoader getParentClassLoader() {
        if (parentClassLoader != null)  // 默认设置为 null
            return (parentClassLoader);
        if (getPrivileged()) { // server.xml 默认配置为 true
            return this.getClass().getClassLoader();
        } else if (parent != null) {
            return (parent.getParentClassLoader());
        }
        return (ClassLoader.getSystemClassLoader());
    }
```

因此使用了当前类的 classloader,即 StandardContext 的 AppClassLoader. 其值为:

sun.misc.Launcher$AppClassLoader@2259e205

注： 该值与 **common classloader** 不同，因为这里是 **eclipse debug** 代码，**JVM** 启动的 **classpath** 包含了 **$TOMCAT/target/classes** 路径，因此 **StandardContext** 的 **classloader** 为 sun.misc.Launcher$AppClassLoader@2259e205,实际线上 tomcat 的 classpath 不会有上述路径,就是 common classloader.

系统启动时存在的 classloader 如下：

```
系统启动时的classloader如下：
=================================== START =========================================
>>>>>>>>>>>> commonloader: java.net.URLClassLoader@67826710
```

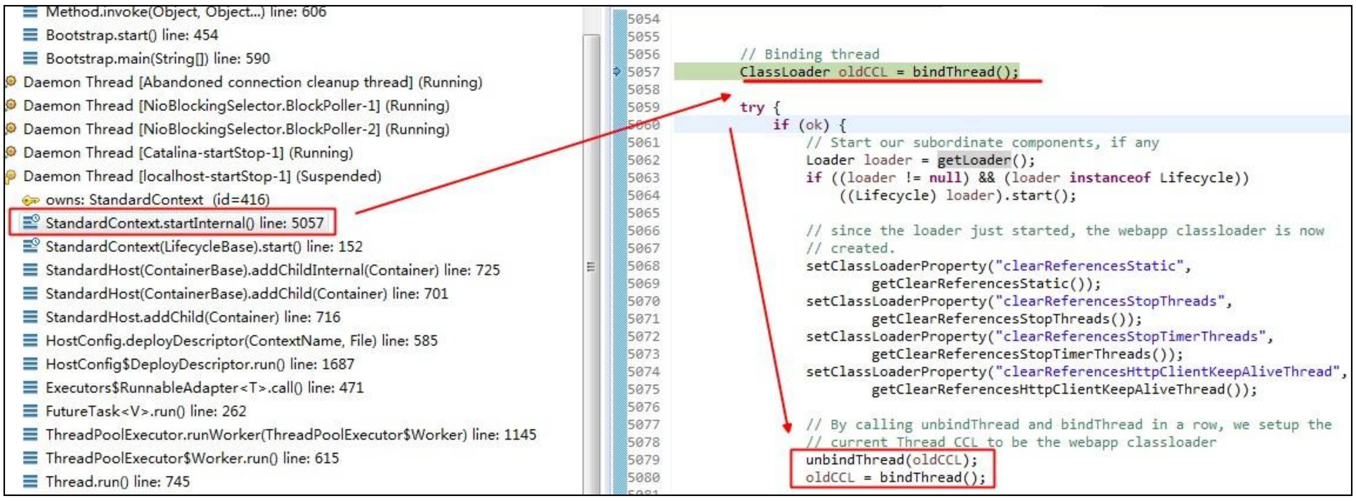
```
>>>>>>>>>>>> app classloader: sun.misc.Launcher$AppClassLoader@2259e205
>>>>>>>>>>>> ext classloader: sun.misc.Launcher$ExtClassLoader@3b05c7e1
```

4. new WebappLoader 时设置了 loaderClass 为：

```
/**
     * The Java class name of the ClassLoader implementation to be used.
     * This class should extend WebappClassLoaderBase, otherwise, a different
     * loader implementation must be used.
     */
    private String loaderClass = WebappClassLoader.class.getName();
```

设置 parentClassLoader 为步骤 3 得到的 classloader.

5. 最终将上面得到的 loader 设置给 context.
   注意：上面只是设置了一个对象，用于保存 classloader 信息的，真正的 classloader 在后面设置。
6. 看看下面的代码逻辑：



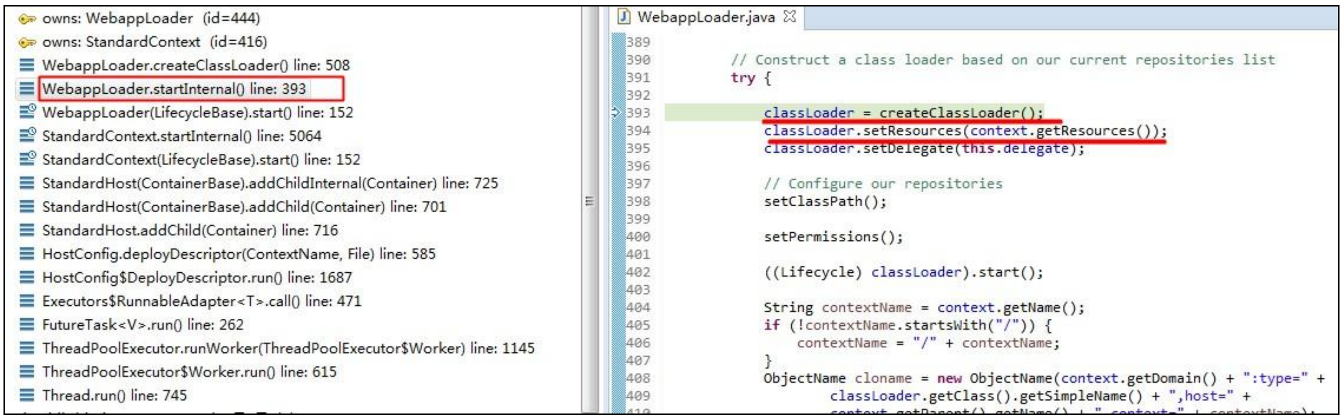
```
Method.invoke(Object, Object...) line: 606
Bootstrap.start() line: 454
Bootstrap.main(String[]) line: 590
Daemon Thread [Abandoned connection cleanup thread] (Running)
Daemon Thread [NioBlockingSelector.BlockPoller-1] (Running)
Daemon Thread [NioBlockingSelector.BlockPoller-2] (Running)
Daemon Thread [Catalina-startStop-1] (Running)
Daemon Thread [localhost-startStop-1] (Suspended)
  owns: StandardContext  (id=416)
  StandardContext.startInternal() line: 5057
  StandardContext(LifecycleBase).start() line: 152
  StandardHost(ContainerBase).addChildInternal(Container) line: 725
  StandardHost(ContainerBase).addChild(Container) line: 701
  StandardHost.addChild(Container) line: 716
  HostConfig.deployDescriptor(ContextName, File) line: 585
  HostConfig$DeployDescriptor.run() line: 1687
  Executors$RunnableAdapter<T>.call() line: 471
  FutureTask<V>.run() line: 262
  ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1145
  ThreadPoolExecutor$Worker.run() line: 615
  Thread.run() line: 745
```

```
        // Binding thread
        ClassLoader oldCCL = bindThread();

        try {
            if (ok) {
                // Start our subordinate components, if any
                Loader loader = getLoader();
                if ((loader != null) && (loader instanceof Lifecycle))
                    ((Lifecycle) loader).start();

                // since the loader just started, the webapp classloader is now
                // created.
                setClassLoaderProperty("clearReferencesStatic",
                        getClearReferencesStatic());
                setClassLoaderProperty("clearReferencesStopThreads",
                        getClearReferencesStopThreads());
                setClassLoaderProperty("clearReferencesStopTimerThreads",
                        getClearReferencesStopTimerThreads());
                setClassLoaderProperty("clearReferencesHttpClientKeepAliveThread",
                        getClearReferencesHttpClientKeepAliveThread());

                // By calling unbindThread and bindThread in a row, we setup the
                // current Thread CCL to be the webapp classloader
                unbindThread(oldCCL);
                oldCCL = bindThread();
```

其中 bindThread() 即 `ClassLoader oldContextClassLoader = bind(false, null);`具体代码如下：

```
@Override
    public ClassLoader bind(boolean usePrivilegedAction, ClassLoader originalClassLoader) {
        Loader loader = getLoader();
        ClassLoader webApplicationClassLoader = null;
        // loader  已经设置到  StandardContext中，设置webApplicationClassLoader  为其classloader
        if (loader != null) {
            webApplicationClassLoader = loader.getClassLoader();
        }

        if (originalClassLoader == null) {
            if (usePrivilegedAction) {
                PrivilegedAction<ClassLoader> pa = new PrivilegedGetTccl();    //  当前线程  classloader
                originalClassLoader = AccessController.doPrivileged(pa);
            } else {
                originalClassLoader = Thread.currentThread().getContextClassLoader();
            }
        }

        if (webApplicationClassLoader == null ||
                webApplicationClassLoader == originalClassLoader) {
            // Not possible or not necessary to switch class loaders. Return null to indicate this.
            return null;
        }

        ThreadBindingListener threadBindingListener = getThreadBindingListener();

        if (usePrivilegedAction) {
            PrivilegedAction<Void> pa = new PrivilegedSetTccl(webApplicationClassLoader);
            AccessController.doPrivileged(pa);
        } else {
            Thread.currentThread().setContextClassLoader(webApplicationClassLoader);
        }
        if (threadBindingListener != null) {
            try {
                threadBindingListener.bind();
            } catch (Throwable t) {
                ExceptionUtils.handleThrowable(t);
                log.error(sm.getString(
                        "standardContext.threadBindingListenerError", getName()), t);
            }
        }

        return originalClassLoader;
    }
```

7. 步骤 6 图的 5064 行代码 start 了 loader，其代码有：



```
owns: WebappLoader  (id=444)
owns: StandardContext  (id=416)
WebappLoader.createClassLoader() line: 508
WebappLoader.startInternal() line: 393
WebappLoader(LifecycleBase).start() line: 152
StandardContext.startInternal() line: 5064
StandardContext(LifecycleBase).start() line: 152
StandardHost(ContainerBase).addChildInternal(Container) line: 725
StandardHost(ContainerBase).addChild(Container) line: 701
StandardHost.addChild(Container) line: 716
HostConfig.deployDescriptor(ContextName, File) line: 585
HostConfig$DeployDescriptor.run() line: 1687
Executors$RunnableAdapter<T>.call() line: 471
FutureTask<V>.run() line: 262
ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1145
ThreadPoolExecutor$Worker.run() line: 615
Thread.run() line: 745
```

WebappLoader.java

```
389
390         // Construct a class loader based on our current repositories list
391         try {
392
393             classLoader = createClassLoader();
394             classLoader.setResources(context.getResources());
395             classLoader.setDelegate(this.delegate);
396
397             // Configure our repositories
398             setClassPath();
399
400             setPermissions();
401
402             ((Lifecycle) classLoader).start();
403
404             String contextName = context.getName();
405             if (!contextName.startsWith("/")) {
406                 contextName = "/" + contextName;
407             }
408             ObjectName cloname = new ObjectName(context.getDomain() + ":type=" +
409                     classLoader.getClass().getSimpleName() + ",host=" +
```

其中 393 行代码即通过反射方法实例化了 StandardContext 的 loaderClass 对象，并调用其带 parent classloader 的构造函数，代码如下：

```
/**
     * Create associated classLoader.
     */
    private WebappClassLoaderBase createClassLoader()
        throws Exception {
        Class<?> clazz = Class.forName(loaderClass);   // WebappClassLoader
        WebappClassLoaderBase classLoader = null;
        if (parentClassLoader == null) {
            parentClassLoader = context.getParentClassLoader();
        }
        Class<?>[] argTypes = { ClassLoader.class };
        Object[] args = { parentClassLoader };
        Constructor<?> constr = clazz.getConstructor(argTypes);
        classLoader = (WebappClassLoaderBase) constr.newInstance(args);
        return classLoader;
    }
```

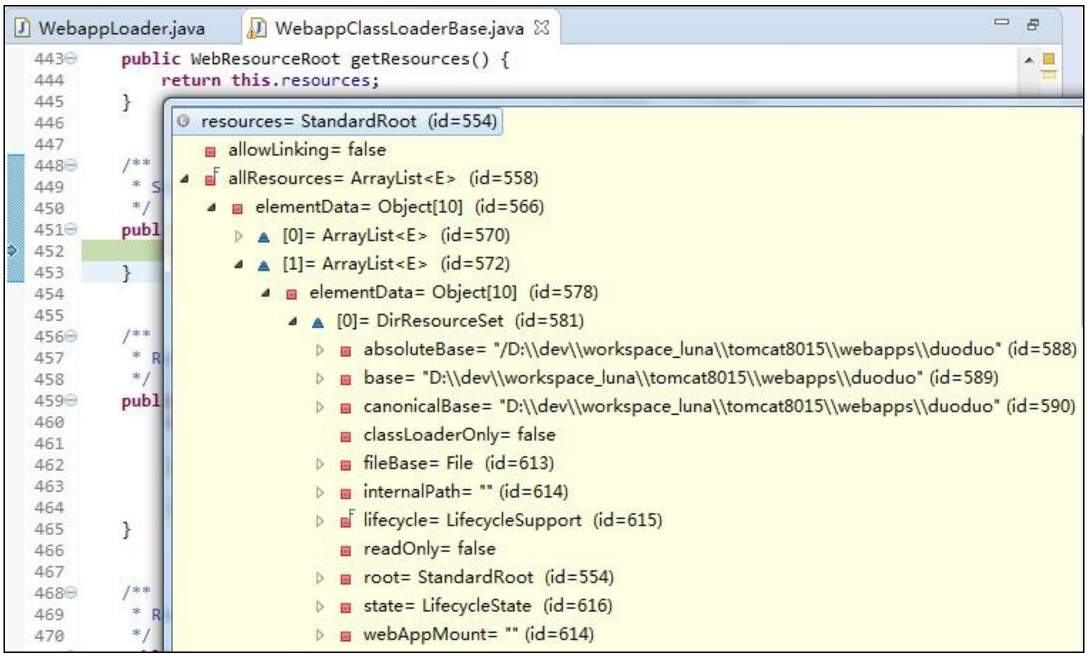
这个 classloader 限制了下述包名是不可以加载的。

```
/**
     * Regular expression of package names which are not allowed to be loaded
     * from a webapp class loader without delegating first.
     */
    protected final Matcher packageTriggersDeny = Pattern.compile(
            "^javax\\.el\\.|" +
            "^javax\\.servlet\\.|" +
            "^org\\.apache\\.(catalina|coyote|el|jasper|juli|naming|tomcat)\\."
            ).matcher("");
```

其中，下面的 package 可以从 webapp 加载

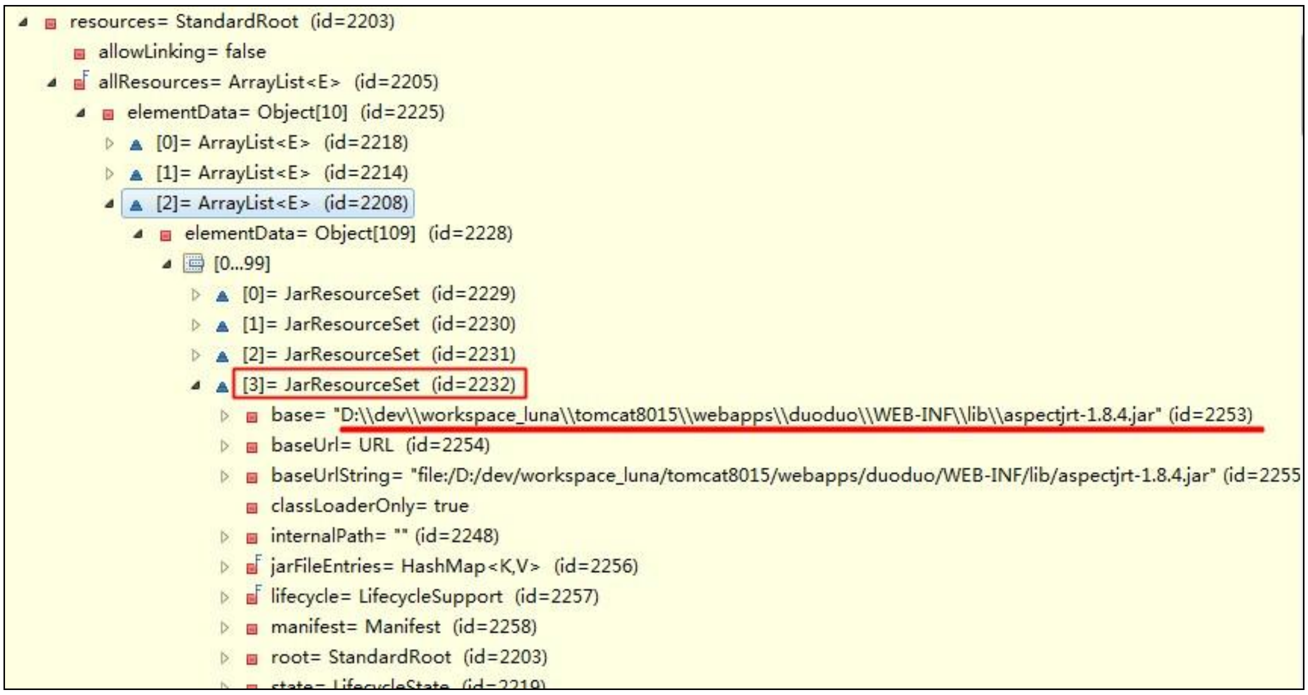
```
/**
     * Regular expression of package names which are allowed to be loaded from a
     * webapp class loader without delegating first and override any set by
     * {@link #packageTriggersDeny}.
     */
    protected final Matcher packageTriggersPermit =
            Pattern.compile("^javax\\.servlet\\.jsp\\.jstl\\.|" +
                    "^org\\.apache\\.tomcat\\.jdbc\\.").matcher("");
```

394 行的 resource 即 StandardRoot，是一个标准的 WebResourceRoot，其包含了 webappclassloader 可以加载的 class 路径，典型的为 webapps/WebX 路径，在其下面可以查找相对路径 WEB-INF/classes 和 WEB-INF/lib/*.jars 资源. 一个例子如下：


WebappLoader.java   WebappClassLoaderBase.java

```
    public WebResourceRoot getResources() {
        return this.resources;
    }

    /**
     * S
     */
    publ

    }

    /**
     * R
     */
    publ

    }

    /**
     * R
     */
```

```
resources= StandardRoot  (id=554)
    allowLinking= false
  ◢ allResources= ArrayList<E>  (id=558)
    ◢ elementData= Object[10]  (id=566)
      ▷ [0]= ArrayList<E>  (id=570)
      ◢ [1]= ArrayList<E>  (id=572)
        ◢ elementData= Object[10]  (id=578)
          ◢ [0]= DirResourceSet  (id=581)
            ▷ absoluteBase= "/D:\\dev\\workspace_luna\\tomcat8015\\webapps\\duoduo" (id=588)
            ▷ base= "D:\\dev\\workspace_luna\\tomcat8015\\webapps\\duoduo" (id=589)
            ▷ canonicalBase= "D:\\dev\\workspace_luna\\tomcat8015\\webapps\\duoduo" (id=590)
              classLoaderOnly= false
            ▷ fileBase= File  (id=613)
            ▷ internalPath= "" (id=614)
            ▷ lifecycle= LifecycleSupport  (id=615)
              readOnly= false
            ▷ root= StandardRoot  (id=554)
            ▷ state= LifecycleState  (id=616)
            ▷ webAppMount= "" (id=614)
```



```
◢ resources= StandardRoot (id=2203)
    allowLinking= false
  ◢ allResources= ArrayList<E> (id=2205)
    ◢ elementData= Object[10] (id=2225)
      ▷ [0]= ArrayList<E> (id=2218)
      ▷ [1]= ArrayList<E> (id=2214)
      ◢ [2]= ArrayList<E> (id=2208)
        ◢ elementData= Object[109] (id=2228)
          ◢ [0...99]
            ▷ [0]= JarResourceSet (id=2229)
            ▷ [1]= JarResourceSet (id=2230)
            ▷ [2]= JarResourceSet (id=2231)
            ◢ [3]= JarResourceSet (id=2232)
              ▷ base= "D:\\dev\\workspace_luna\\tomcat8015\\webapps\\duoduo\\WEB-INF\\lib\\aspectjrt-1.8.4.jar" (id=2253)
              ▷ baseUrl= URL (id=2254)
              ▷ baseUrlString= "file:/D:/dev/workspace_luna/tomcat8015/webapps/duoduo/WEB-INF/lib/aspectjrt-1.8.4.jar" (id=2255
                classLoaderOnly= true
              ▷ internalPath= "" (id=2248)
              ▷ jarFileEntries= HashMap<K,V> (id=2256)
              ▷ lifecycle= LifecycleSupport (id=2257)
              ▷ manifest= Manifest (id=2258)
              ▷ root= StandardRoot (id=2203)
```

设置 delegate 为 false 表示不用传统的 parent 优先的顺序加载 class，而是：bootstrap -> webappclassloaer -> ExtClassloader -> SystemClassloader -> common classloder.

8. Webappclassloader 也是 LifeCycle 的对象，也可以在生命周期启动，其主要逻辑为：

```
// 根据传入的路径 "/WEB-INF/classes" 及classloader加载的资源路径 (webapp/WebX绝对路径)
// 构造实际要查找资源的位置，即：
// file:/D:/dev/workspace_luna/tomcat8015/webapps/duoduo/WEB-INF/classes/
WebResource classes = resources.getResource("/WEB-INF/classes");
if (classes.isDirectory() && classes.canRead()) {
    localRepositories.add(classes.getURL());
}
// 这里是将 "/WEB-INF/lib"目录下每个 jar 文件转化成本地文件资源，实际查找资源为:
// file:/D:/dev/workspace_luna/tomcat8015/webapps/duoduo/WEB-INF/lib/abc.jar
// file:/D:/dev/workspace_luna/tomcat8015/webapps/duoduo/WEB-INF/lib/bcd.jar
WebResource[] jars = resources.listResources("/WEB-INF/lib");
for (WebResource jar : jars) {
    if (jar.getName().endsWith(".jar") && jar.isFile() && jar.canRead()) {
        localRepositories.add(jar.getURL());
        jarModificationTimes.put(
                jar.getName(), Long.valueOf(jar.getLastModified()));
    }
}
```

即设置 /WEB-INF/classes 和 /WEB-INF/lib 的资源到 localRepositories 中去。

9. 后续工作就简单了，设置好了 webappclassloader 后，通过 unbind & bind Thread 就可以将当前 thread 的 context classloader 设置为 webappclassloader. 具体代码参考步骤 6 描述。

10. 设置好了 classloader 后，下一个主要工作是实现 findClass & loadClass 业务逻辑。加载 class 的入口：

```
/**
     * Load the class with the specified name.  This method searches for
     * classes in the same manner as <code>loadClass(String, boolean)</code>
     * with <code>false</code> as the second argument.
     *
     * @param name The binary name of the class to be loaded
     *
     * @exception ClassNotFoundException if the class was not found
     */
    @Override
    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return (loadClass(name, false));
    }
```

11. LoadClass 主要工作为:
    - 11.1 根据 class 名字转换为对应文件路径和名字，如: `org.apache.tomcat.util.descriptor.web.ServletDef` ⇨ `/org/apache/tomcat/util/descriptor/web/ServletDef.class`
    - 11.2 在当前线程的 classloader 即: WebappClassloader 查找是否加载了该资源。这里有个 map 容器，

```
/**
     * The cache of ResourceEntry for classes and resources we have loaded,
     * keyed by resource path, not binary name. Path is used as the key since
     * resources may be requested by binary name (classes) or path (other
     * resources such as property files) and the mapping from binary name to
     * path is unambiguous but the reverse mapping is ambiguous.
     */
    protected final Map<String, ResourceEntry> resourceEntries =
            new ConcurrentHashMap<>();
```

每次新加载的资源都会放到这里。如果找到则说明已加载。返回该资源继续，否则走 11.3.

    - 11.3 没找到，调用 `findLoadedClass(name)`;通过 `native` 方法继续查找是否已经加载过了。找到返回，没找到继续 11.4.
    - 11.4 使用扩展类（即该 classloader 最非 null 的 parent classloader，具体可用 String.class.getClassloader -> parent 得到）加载该资源。找到则返回，否则继续 11.5.
    - 11.5 看看 classloader 的 delegate 是否为 true,或者不是步骤 7 的按 package 过滤的 class，则调用 parent classloader（AppClassLoader）加载该资源。这里 `org.apache.tomcat.util.descriptor.web.ServletDef` 的却不在，调用 `parent` 的 `classloader` 加载。找到则返回。否则查找本地资源。

```
             boolean delegateLoad = delegate || filter(name);
            // (1) Delegate to our parent if requested
            if (delegateLoad) {
                if (log.isDebugEnabled())
                    log.debug("  Delegating to parent classloader1 " + parent);
                try {
                    clazz = Class.forName(name, false, parent);
                    if (clazz != null) {
                        if (log.isDebugEnabled())
                            log.debug("  Loading class from parent");
                        if (resolve)
                            resolveClass(clazz);
                        return (clazz);
                    }
                } catch (ClassNotFoundException e) {
                }
            }
```

11.6 概括一下，class 加载顺序如下：

11.7

12. 概括一下，class 加载顺序如下：

入口：调用 webappclassloader 的 loadClass (className)开始。

12.1 看看是否已加载，如已加载则返回。

12.2 WebappCloader 有个容器 Map<String, ResourceEntry> resourceEntries = new ConcurrentHashMap<>();是否已包含，如果包含则返回。

12.3 通过 native 方式看看是否已加载到方法区，如果加载了则返回。

12.4 通过 system classloader 加载，确保 webapp 不会覆盖 J2SE 的 classes.

12.5 看看 delegate 是否为 true，或者 class 是否被过滤掉。如果条件为 true,则委托父类加载，走正常流程。否则 12.6

12.6 查询本地资源。调用 webappclassloader 的 findClass(className).

findClass(className) 流程为：

12.7 如果是 class，则调用 getResource("/WEB-INF/classes" + path, true, true); 查找。即到 "/WEB-INF/classes" 查找这个资源（前文的 StandardRoot 介绍）。

注：实际上这里查了/WEB-INF/classes 和/WEB-INF/lib/*.jar 文件，找到返回为 WebResource（如 JarResource 等）。

12.8 如果找到，则根据 WebResource 构造 ReourceEntry，放入到 12.2 的缓存容器中去。放的时候发觉如果已经有了，则用已经有的。

```
entry = new ResourceEntry();
entry.source = resource.getURL();
entry.codeBase = resource.getCodeBase();
entry.lastModified = resource.getLastModified();

entry.binaryContent = binaryContent;
entry.certificates = resource.getCertificates();
entry.manifest = resource.getManifest();

学习 ClassFileTransformer 用法，可类似实现 AOP 功能。
```

12.9 上节的 entry 其实带了 loadedClass 属性，这是要 load 的 class，如果有则返回。否则：

12.10根据文件名先保证 package 已设置好。然后调用 ClassLoad 的下面方法定义 class：

```
clazz = defineClass(name, entry.binaryContent, 0,
                     entry.binaryContent.length,
                     new CodeSource(entry.codeBase, entry.certificates));
```

12.11如上可知，最终 define class 还是通用历史逻辑。 Class 加载成功后，继续设置 entry 属性如下：

```
// Now the class has been defined, clear the elements of the local
           // resource cache that are no longer required.
           entry.loadedClass = clazz;
           entry.binaryContent = null;
           entry.codeBase = null;
           entry.manifest = null;
           entry.certificates = null;
           // Retain entry.source in case of a getResourceAsStream() call on
           // the class file after the class has been defined.
```

12.12如果上述还未找到，且允许通过父类查找。则使用父类的 findClass(name)即 URLClassLoader 查找加载。
12.13如果上述还没找到，且不允许 delegate 的情况，这时候可以通过父类继续加载资源了。找到则返回，否则抛出 ClassNotFound 异常。

13. 卡卡

最后的总结

Java 两个环境变量：
{java.ext.dirs} 可选包扩展机制：供各厂商扩展 JavaSE 用的。默认为：$JAVA_HOME/jre/lib/ext
{java.endorsed.dirs} 包升级替换机制：供用户升级 JavaSE 代码用的。默认为：$JAVA_HOME/jre/lib/endorsed
一个是新增，一个是重写。

Class loader 概念：

- Bootstra classloader：启动类加载器。JVM 启动的时候自动加载，具体加载机制由 C 实现，用来加载"sun.boot.class.path"路径下的资源。可以通过如下代码看看都加载了啥：

```
URL[] urls=sun.misc.Launcher.getBootstrapClassPath().getURLs();
for (int i = 0; i < urls.length; i++) {
   System.out.println(urls[i].toExternalForm());
}
➔
file:/D:/dev/jdk/jre/lib/resources.jar
file:/D:/dev/jdk/jre/lib/rt.jar
file:/D:/dev/jdk/jre/lib/sunrsasign.jar
file:/D:/dev/jdk/jre/lib/jsse.jar
file:/D:/dev/jdk/jre/lib/jce.jar
file:/D:/dev/jdk/jre/lib/charsets.jar
file:/D:/dev/jdk/jre/lib/jfr.jar
file:/D:/dev/jdk/jre/classes
```

- ExtClassLoader:扩展类加载器。主要由各厂商实现对 JavaSE 类的扩展,譬如重写 java.util.ArrayList 方法等。放在 System.*getProperty*("java.ext.dirs") 路径下，默认为 $JAVA_HOME/jre/lib/ext 目录下。默认的扩展目录对所有从同一个 JRE 中启动的 JVM 都是通用的，所以放入这个目录的 JAR 类包对所有的 JVM 和 system classloader 都是可见的。Eg:

```
java.ext.dirs: D:\dev\jdk\jre\lib\ext;C:\windows\Sun\Java\lib\ext
```

- AppClassloader：系统类加载器。加载 System.*getProperty*("java.class.path")下资源。 Eg:

```
java.class.path: D:\dev\workspace_luna\simple-sample\target\classes
```

- UserSelfDefinedClassLoader：用户自定义类加载器。如 Tomcat 定义加载 tomcat 自身代码的 common classloader 以及加载 webapp 资源的 webapp classloader 等。

Oracle的 JDK代码中，ExtClassLoader & AppClassLoader 均由oracle厂商在 sum.misc.Launcher 中实现，代码如下：

```
/**
 * This class is used by the system to launch the main application.
Launcher */
```

```
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();  // java.ext.dirs
        } catch (IOException e) {
            throw new InternalError(
                "Could not create extension class loader");
        }
        // Now create the class loader to use to launch the application
        try {
            loader = AppClassLoader.getAppClassLoader(extcl); // java.class.path
        } catch (IOException e) {
            throw new InternalError(
                "Could not create application class loader");
        }
        // Also set the context class loader for the primordial thread.
        Thread.currentThread().setContextClassLoader(loader);
        // Finally, install a security manager if requested
        String s = System.getProperty("java.security.manager");
        if (s != null) {
            SecurityManager sm = null;
            if ("".equals(s) || "default".equals(s)) {
                sm = new java.lang.SecurityManager();
            } else {
                try {
                    sm = (SecurityManager)loader.loadClass(s).newInstance();
                } catch (IllegalAccessException e) {
                } catch (InstantiationException e) {
                } catch (ClassNotFoundException e) {
                } catch (ClassCastException e) {
                }
            }
            if (sm != null) {
                System.setSecurityManager(sm);
            } else {
                throw new InternalError(
                    "Could not create SecurityManager: " + s);
            }
        }
    }
    /*
     * Returns the class loader used to launch the main application.
     */
    public ClassLoader getClassLoader() {
        return loader;
    }
```

`Map<String, ResourceEntry> resourceEntries = new ConcurrentHashMap<>();`

1. 根据class name得到对应 .class 的 path,看看 resourceEntries 里面是否可以找到。【**webappclassloader本地**】
2. 根据name通过该classloader的findLoadedClass0(String name)从jvm native看看是否已加载。【**JVM本地**】
3. 根据ExtClassLoader查找对应路径资源，如果找到则用这个classloader加载。【**$JAVA_HOME/lib/ext目录**】
4. 根据是否delegate给父类：
   4.1 如果delegate给父类，则用parent classloader按传统方式加载class. 这里最多到系统类。上面被2，3处理。如果未找到，则走4.2即通过自身classloder查找加载。
   4.2 否则，调用webappclassloader的findClass(name)通过该classloader新定义。如果未找到，则走5.【**WEB-INF/...**】
   如果走4.2，则可以忽略掉common classloader 和 system classloader (AppClassloader)加载过程。
5. 用parent classloader按传统方式加载class，类似4.1. 【**common loader / java.class.path等**】

以上过程，找到class加载后立即返回，如果经过5还没有找到，则抛出 ClassNotFoundException 异常。
通过 4.2 可以实现资源的动态加载功能。