# sun.misc.URLClassPath 解析

原创

N3verL4nd 于 2020-02-23 14:59:53 发布 2629 收藏 1

分类专栏： Java学习笔记



本文链接：https://blog.csdn.net/x_iya/article/details/104460158



Java学习笔记 专栏收录该内容

173 篇文章 16 订阅

订阅专栏

作用：Java 中加载类和查找资源的路径

jdk8

http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/8477fd88653c/src/share/classes/sun/misc/URLClassPath.java

jdk9

http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/9b93380c8445/src/java.base/share/classes/jdk/internal/loader/URLClassPath.java

jdk10

http://hg.openjdk.java.net/jdk/jdk10/file/b09e56145e11/src/java.base/share/classes/jdk/internal/loader/URLClassPath.java

jdk11

http://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/java.base/share/classes/jdk/internal/loader/URLClassPath.java

jdk12

https://hg.openjdk.java.net/jdk/jdk12/file/06222165c35f/src/java.base/share/classes/jdk/internal/loader/URLClassPath.java

jdk13

http://hg.openjdk.java.net/jdk/jdk13/file/0368f3a073a9/src/java.base/share/classes/jdk/internal/loader/URLClassPath.java

本文针对 jdk8

## 属性

```
// 保存所有的查找路径
private ArrayList<URL> path = new ArrayList<>();
   Stack<URL> urls = new Stack<>();
 // 初始为空，根据 urls 生成 loaders
   ArrayList<Loader> loaders = new ArrayList<>();
 // 判断路径和 jar 包是否有相应的 Loader，没有才会放入 loaders 和 lmap
   HashMap<String, Loader> lmap = new HashMap<>();
```

## 构造方法

```java
/**
 * 根据给定的 URL 创建 URLClassPath
 * 使用给定的 URL 顺序查找 Resource 和 jar
 * 如果 URL 以 '/' 结尾则被认定为路径，否则被认定为 jar 文件
 */
public URLClassPath(URL[] urls, URLStreamHandlerFactory factory, AccessControlContext acc) {
    for (int i = 0; i < urls.length; i++) {
        path.add(urls[i]);
    }
    push(urls);
    if (factory != null) {
        jarHandler = factory.createURLStreamHandler("jar");
    }
    if (DISABLE_ACC_CHECKING) {
        this.acc = null;
    } else {
        this.acc = acc;
    }
}

public URLClassPath(URL[] urls) { this(urls, null, null); }


public URLClassPath(URL[] urls, AccessControlContext acc) { this(urls, null, acc); }
```

URLClassPath 在哪里被使用到

```
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();
    }
    acc = AccessController.getContext();
    ucp = new URLClassPath(urls, factory, acc);
}

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

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

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

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

jdk1.8中 AppClassLoader 和 ExtClassLoader 都继承于 URLClassLoader，生成 UrlClassPath 的时候：

AppClassLoader 传入"**java.class.path**"对应的路径

ExtClassLoader 传入"**java.ext.dirs**" 对应的路径

```java
// 儿子找爹可以通过 *.class.getClassLoader()实现，爹找儿子属于 JVM 实现层级上的东西，随时可能在版本迭代中被删除。
private static native int[] getLookupCacheForClassLoader(ClassLoader loader, String name);

private synchronized int[] getLookupCache(String name) {
    if (lookupCacheURLs == null || !lookupCacheEnabled) {
        return null;
    }
    int[] cache = getLookupCacheForClassLoader(lookupCacheLoader, name);
    if (cache != null && cache.length > 0) {
        int maxindex = cache[cache.length - 1]; // cache[] is strictly ascending.
        if (!ensureLoaderOpened(maxindex)) {
            if (DEBUG_LOOKUP_CACHE) {
                System.out.println("Expanded loaders FAILED " + loaders.size() + " for maxindex=" + maxindex);
            }
            return null;
        }
    }
    return cache;
}

public Resource getResource(String name, boolean check) {
    if (DEBUG) {
        System.err.println("URLClassPath.getResource(\"" + name + "\")");
    }
    Loader loader;
    int[] cache = getLookupCache(name);
    for (int i = 0; (loader = getNextLoader(cache, i)) != null; i++) {
        Resource res = loader.getResource(name, check);
        if (res != null) {
            return res;
        }
    }
    return null;
}

private synchronized Loader getNextLoader(int[] cache, int index) {
    if (closed) {
        return null;
    }
    if (cache != null) {
        if (index < cache.length) {
            Loader loader = loaders.get(cache[index]);
            if (DEBUG_LOOKUP_CACHE) {
                System.out.println("HASCACHE: Loading from : " + cache[index] + " = " + loader.getBaseURL());
            }
            return loader;
        } else {
            return null; // finished iterating over cache[]
        }
    } else {
        return getLoader(index);
    }
}

private synchronized Loader getLoader(int index) {
    if (closed) {
        return null;
    }
    // Expand URL search path until the request can be satisfied or the URL stack is empty.
```

```
    while (loaders.size() < index + 1) {
        // Pop the next URL from the URL stack
        URL url;
        synchronized (urls) {
            if (urls.empty()) {
                return null;
            } else {
                url = urls.pop();
            }
        }
        // Skip this URL if it already has a Loader. (Loader may be null in the case where URL has not been open
ed but is referenced by a JAR index.)
        String urlNoFragString = URLUtil.urlNoFragString(url);
        if (lmap.containsKey(urlNoFragString)) {
            continue;
        }
        // Otherwise, create a new Loader for the URL.
        Loader loader;
        try {
            loader = getLoader(url);
            // If the loader defines a local class path then add the URLs to the list of URLs to be opened.

            URL[] urls = loader.getClassPath();
            if (urls != null) {
                push(urls);
            }
        } catch (IOException e) {
            // Silently ignore for now...
            continue;
        } catch (SecurityException se) {
            // Always silently ignore. The context, if there is one, that this URLClassPath was given during con
struction will never have permission to access the URL.
            if (DEBUG) {
                System.err.println("Failed to access " + url + ", " + se);
            }
            continue;
        }
        // Finally, add the Loader to the search path.
        validateLookupCache(loaders.size(), urlNoFragString);
        loaders.add(loader);
        lmap.put(urlNoFragString, loader);
    }
    if (DEBUG_LOOKUP_CACHE) {
        System.out.println("NOCACHE: Loading from : " + index);
    }
    return loaders.get(index);
}

// 根据 url 生成 Loader
private Loader getLoader(final URL url) throws IOException {
    try {
        return java.security.AccessController.doPrivileged(
                new java.security.PrivilegedExceptionAction<Loader>() {
                    @Override
                    public Loader run() throws IOException {
                        String file = url.getFile();
                        if (file != null && file.endsWith("/")) {
                            if ("file".equals(url.getProtocol())) {
                                return new FileLoader(url);
```

```
                            } else {
                                return new Loader(url);
                            }
                        } else {
                            return new JarLoader(url, jarHandler, lmap, acc);
                        }
                    }
                }, acc);
    } catch (java.security.PrivilegedActionException pae) {
        throw (IOException) pae.getException();
    }
}
// 新添加进去的导致查找缓存失效
private synchronized void validateLookupCache(int index, String urlNoFragString) {

    if (lookupCacheURLs != null && lookupCacheEnabled) {

        if (index < lookupCacheURLs.length && urlNoFragString.equals(URLUtil.urlNoFragString(lookupCacheURLs[ind
ex]))) {
            return;

        }

        if (DEBUG || DEBUG_LOOKUP_CACHE) {

            System.out.println("WARNING: resource lookup cache invalidated " + "for lookupCacheLoader at " + ind
ex);
        }

        disableAllLookupCaches(); // lookupCacheEnabled = false;

    }

}

// 操作 urls 的几个方法
// getLoader(int index) 中

// 增加查找路径
public synchronized void addURL(URL url) {
    if (closed) {
        return;
    }
    synchronized (urls) {
        if (url == null || path.contains(url)) {
            return;
        }
        // 在 Stack 的头部添加元素
        urls.add(0, url);
        path.add(url);
        if (lookupCacheURLs != null) {
            // 查找缓存是静态的("java.class.path"、"java.ext.dirs"), 动态添加的查找缓存导致其不可使用

            disableAllLookupCaches();
        }
    }
}

// java.net.URLClassLoader
private final URLClassPath ucp;
```

```
private final URLClassPath ucp;
// java.net.URLClassLoader#addURL
protected void addURL(URL url) {
    ucp.addURL(url);
}
```

getLookupCache 查找缓存是为了提高查找速度，作用如下：
如果 lookupCacheURLs包 含{a.jar，b.jar，c.jar，d.jar}，并且包“ foo”仅存在于a.jar和c.jar中，则getLookupCache（“ foo / Bar.class”）将返回 {0，2}。
作用和 sun.misc.MetaIndex 差不多
关于：VM.getSavedProperty(“sun.cds.enableSharedLookupCache”)

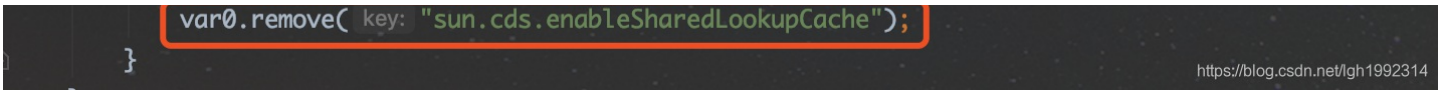
> 在系统初始化的时候，JVM调用java.lang.System.initializeSystemClass()来初始化System类，而它进一步调用的java.lang.System.initProperties()会调用到JVM的JVM_InitProperties()函数，来从JVM那边把-D参数都拿到System.properties里。
>
> 紧接着，initializeSystemClass()又调用sun.misc.VM.saveAndRemoveProperties()来保存一份“干净”的system properties备份以便JDK内部使用，避免JDK的内部行为受到运行时用户代码对System.properties的修改所干扰。同时saveAndRemoveProperties()也会从System.properties移除一些只供JDK内部使用、用户代码不应该看到的系统属性，例如设置NIO direct memory大小限制的sun.nio.MaxDirectMemorySize属性。这些被移除的系统属性在sun.misc.VM.savedProps里都留有备份。 https://www.iteye.com/topic/1132986

sun.cds.enableSharedLookupCache 是 JVM 实现层面的参数。

```
public static void saveAndRemoveProperties(Properties var0) {
    if (booted) {
        throw new IllegalStateException("System initialization has completed");
    } else {
        savedProps.putAll(var0);
        String var1 = (String)var0.remove( key: "sun.nio.MaxDirectMemorySize");
        if (var1 != null) {
            if (var1.equals("-1")) {
                directMemory = Runtime.getRuntime().maxMemory();
            } else {
                long var2 = Long.parseLong(var1);
                if (var2 > -1L) {
                    directMemory = var2;
                }
            }
        }

        var1 = (String)var0.remove( key: "sun.nio.PageAlignDirectMemory");
        if ("true".equals(var1)) {
            pageAlignDirectMemory = true;
        }

        var1 = var0.getProperty("sun.lang.ClassLoader.allowArraySyntax");
        allowArraySyntax = var1 == null ? defaultAllowArraySyntax : Boolean.parseBoolean(var1);
        var0.remove( key: "java.lang.Integer.IntegerCache.high");
        var0.remove( key: "sun.zip.disableMemoryMapping");
        var0.remove( key: "sun.java.launcher.diag");
```




**URLClassPath 查找缓存相关的代码在 jdk9 中已经被删除**

http://openjdk.5641.n7.nabble.com/RFR-M-8061651-Interface-to-the-Lookup-Index-Cache-to-improve-URLClassPath-search-time-td204636.html

## 参考

https://www.iteye.com/topic/1132986