Java 类加载器揭秘

什么是 ClassLoader

Java 代码要想运行，首先需要将源代码进行编译生成 .class 文件，然后 JVM 加载 .class 字节码文件到内存，而 .class 文件是怎样被加载到 JVM 中的就是Java ClassLoader 要做的事情。

那么 .class 文件什么时候会被类加载器加载到 JVM 中那？比如执行 new 操作时候；当我们使用 Class.forName(“包路径+类名”)、 Class.forName(“包路径+类名”,ClassLoader)、ClassLoader.loadClass(“包路径+类名”)的时候就触发了类加载器去类加载对应的路径去查找 *.class，并创建 Class 对象。另外需要注意的是除去 new 操作外，其他几种方式加载字节码到内存后只是生产一个 Class 对象，要产生具体的对象实例还需要使用 Class 对象 .newInstance() 函数来创建。

Java 原生的三种 ClassLoader

AppClassloader

应用类加载器，又称系统类加载器。它负责在 JVM 启动时，加载来自在命令java中的-classpath或者java.class.path系统属性或者 CLASSPATH 操作系统属性所指定的 JAR 类包和类路径。调用 ClassLoader.getSystemClassLoader() 可以获取该类加载器。如果没有特别指定，则用户自定义的任何类加载器都将该类加载器作为它的父加载器，这点通过 java.lang.ClassLoader 的无参构造函数可以证明，代码如下。

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

执行以下代码即可获得 classpath 加载路径。

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

另外我们写的含有 main 函数的类的加载就是使用该类加载器进行加载的，证明如下：

```
    public static void main(String[] args) {
        ClassLoader cl = App2.class.getClassLoader();
        System.out.println(cl);
        System.out.println(cl.getParent());
    }
```

运行结果如下。

```
sun.misc.Launcher$AppClassLoader@4554617c
sun.misc.Launcher$ExtClassLoader@677327b6
```

并且可以看出 AppClassLoader 的父加载器是 ExtClassLoader，那么 ExtClassLoader 是什么呢？

ExtClassloader

扩展类加载器，主要负责加载 Java 的扩展类库，默认加载 `JAVA_HOME/jre/lib/ext/` 目录下的所有 Jar 包或者由 `java.ext.dirs` 系统属性指定的 Jar 包。放入这个目录下的 Jar 包对 AppClassloader 加载器都是可见的（因为 ExtClassloader 是 AppClassloader 的父加载器，并且 Java 类加载器采用了委托机制)。那么 ExtClassloader 的类扫描路径都有哪些？

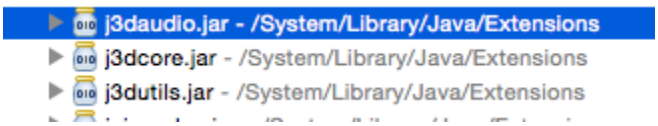
执行如下代码。

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

运行结果如下。

```
/Users/zhuizhumengxiang/Library/Java/Extensions:/Library/Java/Jav
aVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/ext:/Libr
ary/Java/Extensions:/Network/Library/Java/Extensions:/System/Libr
ary/Java/Extensions:/usr/lib/java
```

具体比如下面有关 3D视频的一些 Jar：



这里我们随便找一个 `javax.media.j3d.Font3D` 类，来看看它的类加载器是谁。

```
    public static void main(String[] args) {
        ClassLoader cl =
```

```
javax.media.j3d.Font3D.class.getClassLoader();
        System.out.println(cl);
        System.out.println(cl.getParent());
    }
```

执行结果如下。

```
sun.misc.Launcher$ExtClassLoader@135fbaa4
null
```

另外可知 ExtClassLoader 的父加载器为 null。

BootstrapClassloader

引导类加载器，又称启动类加载器，是最顶层的类加载器，主要用来加载 Java 核心类，如 rt.jar、resources.jar、charsets.jar 等。

需要注意的是它不是 java.lang.ClassLoader 的子类，而是由 JVM 自身实现的，该类为 C 语言实现，所以严格来说它不属于 Java 类加载器范畴，Java 程序访问不到该加载器。

通过下面代码我们可以查看该加载器查找类的扫描路径。

```
public void test() {
        URL[] urls =
sun.misc.Launcher.getBootstrapClassPath().getURLs();
        for (int i = 0; i < urls.length; i++) {
            System.out.println(urls[i].toExternalForm());
        }
    }
```

执行结果如下。

```
<terminated> App2 (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2018年2月9日 上午10:08:25)
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/resources.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/rt.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/sunrsasign.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/jsse.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/jce.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/charsets.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/lib/jfr.jar
file:/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/jre/classes
```

String 类就是 rt.jar 里面提供的，这个类我们经常用，下面我们看下 String 类的类加载器是什么。

```
    public static void main(String[] args) {
        ClassLoader cl = String.class.getClassLoader();
```

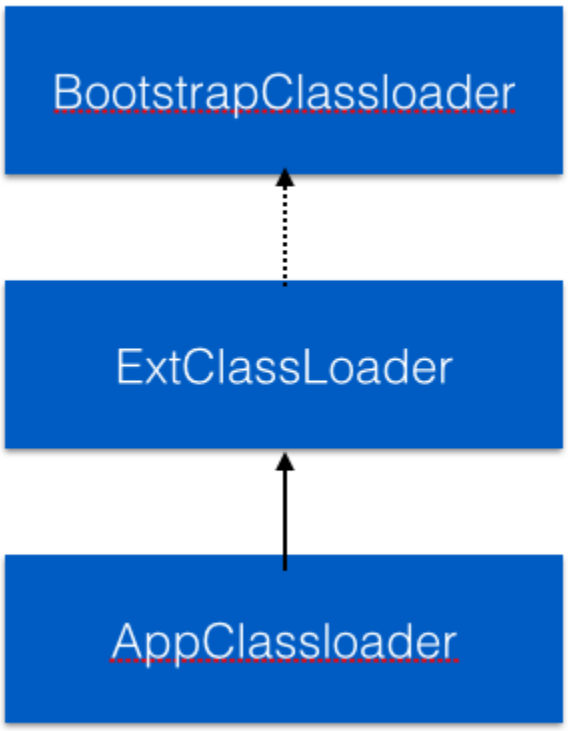
```
            System.out.println(cl);
        }
```

执行结果如下。

可知由于 BootstrapClassloader 对 Java 不可见，所以返回了 null，我们也可以通过看某一个类的加载器是否为 null 来作为判断该类是不是使用 BootstrapClassloader 进行加载的依据。另外上面提到 ExtClassLoader 的 父加载器返回的 null，那是否说明 ExtClassLoader 的父加载器是 BootstrapClassloader呢？

三种加载器关系

首先用一张图来表示三张图的关系如下：



- AppClassloader 的父加载器是 ExtClassloader。
- ExtClassloader 的父加载器为 null，但是要注意的是 ExtClassloader 的父加载器并不是 BootstrapClassloader。

类加载器的构造

下面从源码来分析下 JVM 是如何构建内置 Classloader 的，具体是 rt.jar 包里面 sun.misc.Launcher类。代码如下。

```
public Launcher()
      {
        ExtClassLoader localExtClassLoader;
        try
        {  //（1）首先创建了ExtClassLoader
          localExtClassLoader =
ExtClassLoader.getExtClassLoader();
        }
        catch (IOException localIOException1)
        {
          throw new InternalError("Could not create extension
class loader");
        }
        try
        {  //（2）然后以ExtClassloader作为父加载器创建了
AppClassLoader
          this.loader =
AppClassLoader.getAppClassLoader(localExtClassLoader);
        }
        catch (IOException localIOException2)
        {
          throw new InternalError("Could not create application
class loader");
        }  //（3）这个是个特殊的加载器后面会讲到，这里只需要知道默认下线程
上下文加载器为appclassloader

Thread.currentThread().setContextClassLoader(this.loader);

        .................
      }
```

代码（1）首先创建了 ExtClassLoader 类加载器，下面我们看看具体创建过程，打开 ExtClassLoader.getExtClassLoader() 的代码，如下。

```
public static ExtClassLoader getExtClassLoader()
      throws IOException
    {
      File[] arrayOfFile = getExtDirs();
      try
      {
        (ExtClassLoader)AccessController.doPrivileged(new
PrivilegedExceptionAction()
        {
          public Launcher.ExtClassLoader run()
            throws IOException
          {
            int i = this.val$dirs.length;
            for (int j = 0; j < i; j++) {
              MetaIndex.registerDirectory(this.val$dirs[j]);
```

```
            }
            //(5)
            return new Launcher.ExtClassLoader(this.val$dirs);
          }
        });
      }
      catch (PrivilegedActionException
localPrivilegedActionException)
      {
        throw
((IOException)localPrivilegedActionException.getException());
      }
    }

   //(6)
    private static File[] getExtDirs()
    {
      String str = System.getProperty("java.ext.dirs");
      File[] arrayOfFile;
      if (str != null)
      {
        StringTokenizer localStringTokenizer = new
StringTokenizer(str, File.pathSeparator);

        int i = localStringTokenizer.countTokens();
        arrayOfFile = new File[i];
        for (int j = 0; j < i; j++) {
          arrayOfFile[j] = new
File(localStringTokenizer.nextToken());
        }
      }
      else
      {
        arrayOfFile = new File[0];
      }
      return arrayOfFile;
    }
```

从代码（6）可知，ExtClassLoader 类加载类扫描路径为 “java.ext.dirs“`。下面我们看看代码（5），它说明 ExtClassLoader 的父加载器为 null，打开 Launcher.ExtClassLoader 的代码如下。

```
    public ExtClassLoader(File[] paramArrayOfFile)
      throws IOException
    {
     //(7)第一个参数，就是父加载器的设置，这里传递了null。
      super(null, Launcher.factory);
      SharedSecrets.getJavaNetAccess()
        .getURLClassPath(this).initLookupCache(this);
    }
```

代码（2）以创建的 ExtClassloader 作为父加载器创建了 AppClassLoader，下面看下 AppClassLoader.getAppClassLoader 的代码，如下。

```
public static ClassLoader getAppClassLoader(final ClassLoader
paramClassLoader)
      throws IOException
    {  //(8)
      String str = System.getProperty("java.class.path");
      final File[] arrayOfFile = str == null ? new File[0] :
Launcher.getClassPath(str);

      (ClassLoader)AccessController.doPrivileged(new
PrivilegedAction()
      {
        public Launcher.AppClassLoader run()
        {
          URL[] arrayOfURL = this.val$s == null ? new URL[0] :
Launcher.pathToURLs(arrayOfFile);

          return new Launcher.AppClassLoader(arrayOfURL,
paramClassLoader);
        }
      });
    }
```

由代码（8）可知 AppClassLoader 类加载扫描路径为为 “java.class.path“`。下面看下 AppClassLoader 的父加载器的设置，看下 Launcher.AppClassLoader 的代，如下。

```
 AppClassLoader(URL[] paramArrayOfURL, ClassLoader
paramClassLoader)
    {
     //paramClassLoader就是ExtClassloader
      super(paramClassLoader, Launcher.factory);
      this.ucp =
SharedSecrets.getJavaNetAccess().getURLClassPath(this);
      this.ucp.initLookupCache(this);
    }
```

代码（3）创建了一个与线程相关的类加载器，这个后面会讲到。

类加载器原理

Java 类加载器使用的是委托机制，也就是一个类加载器在加载一个类时候会首先尝试让父类加载器来加载。那么问题来了，为啥使用这种方式？

第一，这样可以避免重复加载，当父类加载器已经加载了该类的时候，就没有必要子 ClassLoader 再加载一次。

第二，考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随便使用自定义的类来动态替代 Java 核心 API 中类实现，比如我们自己写了个 String 类，包路径+类名与 rt.jar 里面的一样，如果不用委托机制，那么当 JVM 加载 String 类的时候会使用 AppClassLoader 加载我们自己定义的 String 类而不会去加载 rt.jar 里面的了。使用双亲委托则，当 JVM 加载 String 类的时候， AppClassLoader 会委托父类加载器 ExtClassLoader 来加载，ExtClassLoader 又会委托给 Bootstrcp ClassLoader 来加载，这样就不会加载自定义的 String 类了。

下面我们从源码看如何实现委托机制。

```
protected Class<?> loadClass(Stringname,boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // 首先从jvm缓存查找该类
            Class c = findLoadedClass(name); // (1)
            if (c ==null) {
                longt0 = System.nanoTime();
                try {  //然后委托给父类加载器进行加载
                    if (parent !=null) {
                        c = parent.loadClass(name,false);  (2)
                    } else {  //如果父类加载器为null,则委托给BootStrap
加载器加载
                        c = findBootstrapClassOrNull(name);  (3)
                    }
                } catch (ClassNotFoundExceptione) {
                    // ClassNotFoundException thrown if class not
found
                    // from the non-null parent class loader
                }

                if (c ==null) {
                    // 若仍然没有找到则调用findclass查找
                    // to find the class.
                    longt1 = System.nanoTime();
                    c = findClass(name);  (4)

                    // 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);  //（5）
                }
                returnc;
            }
        }
```

代码（1）表示从 JVM 缓存查找该类，如果该类之前被加载过，则直接从 JVM 缓存返回该类。

代码（2）表示如果 JVM 缓存不存在该类，则看当前类加载器是否有父加载器，如果有的话则委托父类加载器进行加载，否者调用（3），委托 BootStrapClassloader 进行加载，如果还是没有找到，则调用当前 Classloader 的 findclass 方法进行查找。

代码（5）则是当字节码加载到内存后进行链接操作，对文件格式和字节码验证，并为 static 字段分配空间并初始化，符号引用转为直接引用，访问控制，方法覆盖等，本文对这些不进入深入探讨。

从上面源码知道要想修改类加载委托机制，实现自己的载入策略，可以通过覆盖 ClassLoader 的 findClass 方法或者覆盖 loadClass 方法来实现。

一种特殊的类加载器 ContextClassLoader

ContextClassLoader 是一种与线程相关的类加载器，类似 ThreadLocal，每个线程对应一个上下文类加载器。在使用时，一般都用下面的经典结构。

```
//获取当前线程上下文类加载器
ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
try {//设置当前线程上下文类加载器为targetTccl
    Thread.currentThread().setContextClassLoader(targetTccl);
    //doSomething
    doSomething();
} finally {//设置当前线程上下文加载器为原始加载器
    Thread.currentThread().setContextClassLoader(classLoader);
}
```

首先获取当前线程的线程上下文类加载器并保存到方法栈，然后设置当前线程上下文类加器为自己的类加载器。

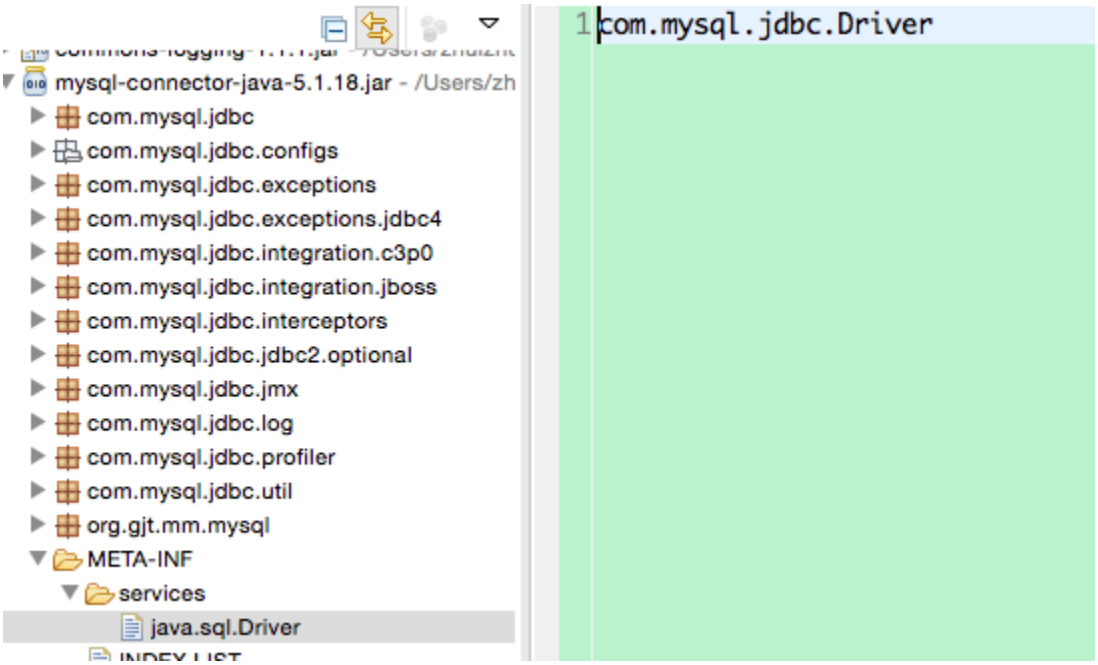
doSomething 里面则调用了 Thread.currentThread().getContextClassLoader()，获取当前线程上下文类加载器做某些事情。

最后在设置当前线程上下文类加载器为老的类加载器。

那么这其中的奥秘和使用场景是什么？我们知道 Java 默认的类加载机制是委托机制，但是有些时候需要破坏这种机制。

具体来说，比如 Java 中的 SPI（Service Provider Interface）是面向接口编程的，服务规则提供者会在 JRE 的核心 API 里面提供服务访问接口，而具体实现则由其他开发商提供。我们知道 Java 核心 API，比如rt.jar包，是使用 Bootstrap CalssLoader 加载的，而用户提供的 Jar 包再由 Appclassloader 加载。并且我们知道如果一个类由类加载器 A 加载，那么这个类依赖类也是由相同的类加载器加载。那么 Bootstrap ClassLoader 加载了服务提供者在 rt.jar 里面提供的搜索开发商提供的实现类的 API 类（ServiceLoader），那么这些 API 类里面依赖的类应该也是由 Bootstrap CalssLoader 来加载。而上面说了用户提供的 Jar 包再由 Appclassloader 加载，所以需要一种违反双亲委派模型的方法，线程上下文类加载器就是为了解决这个问题。

下面使用 JDBC 4 来具体说明，JDBC 4 是基于 SPI 机制来发现驱动提供商提供的实现类，提供者只需在 JDBC 实现的 Jar 的 `META-INF/services/java.sql.Driver` 文件里指定实现类的方式暴露驱动提供者。例如 MySQL 实现的 Jar，如下。



而 MySQL 的驱动如下实现了 java.sql.Driver。

```
public class com.mysql.jdbc.Driver extends
com.mysql.jdbc.NonRegisteringDriver implements java.sql.Driver
```

OK，下面我们写个测试代码，看看具体是如何工作的。

```
    public static void main(String[] args) {
                //(1)
        ServiceLoader<Driver> loader =
ServiceLoader.load(Driver.class);
                //(2)
        Iterator<Driver> iterator = loader.iterator();
```

```
            while (iterator.hasNext()) {
                Driver driver = (Driver) iterator.next();
                System.out.println("driver:" + driver.getClass() +
    ",loader:" + driver.getClass().getClassLoader());
            }
            //(3)
            System.out.println("current thread contextloader:" +
    Thread.currentThread().getContextClassLoader());
                    //(4)
            System.out.println("ServiceLoader loader:" +
    ServiceLoader.class.getClassLoader());
        }

    }
```

然后引入 MySQL 驱动的 Jar 包，执行结果如下。

```
driver:class
com.mysql.jdbc.Driver,loader:sun.misc.Launcher$AppClassLoader@455
4617c
current thread
contextloader:sun.misc.Launcher$AppClassLoader@4554617c
ServiceLoader loader:null
```

从执行结果可以知道 ServiceLoader 的加载器为 Bootstarp，因为这里输出了 null，并且从该类在 rt.jar 里面，也可以证明。

当前线程上下文类加载器为 AppClassLoader。而 com.mysql.jdbc.Driver 则使用 AppClassLoader 加载。我们知道如果一个类中引用了另外一个类，那么被引用的类也应该由引用方类加载器来加载，而现在则是引用方 ServiceLoader 使用 BootStarpClassloader 加载，被引用方则使用子加载器 APPClassLoader 来加载了，是不是很诡异。

下面我们来看下 ServiceLoader 的 load 方法源码。

```
public final class ServiceLoader<S> implements Iterable<S> {
    public static <S> ServiceLoader<S> load(Class<S> service) {
        // （5）获取当前线程上下文加载器
        ClassLoader cl =
Thread.currentThread().getContextClassLoader();
        return ServiceLoader.load(service, cl);
    }

    public static <S> ServiceLoader<S> load(Class<S> service,
ClassLoader loader) {
        return new ServiceLoader<>(service, loader);
    }
       //(6)
```

```
    private ServiceLoader(Class<S> svc, ClassLoader cl) {
        service = svc;
        loader = cl;
        reload();
    }
```

代码（5）获取了当前线程上下文加载器，这里是 AppClassLoader。

代码（6）传递该类加载器到新构造的 ServiceLoader 的成员变量 loader。那么这个 loader 什么时候使用的呢？下面我们看下 LazyIterator 的 next() 方法。

```
            public S next() {
              if (acc == null) {
                  return nextService();
              } else {
                  PrivilegedAction<S> action = new
PrivilegedAction<S>() {
                      public S run() { return nextService(); }
                  };
                  return AccessController.doPrivileged(action,
acc);
              }
          }
```

```
  private S nextService() {
   ...
    try {
        //（7）使用loader类加载器加载
        c = Class.forName(cn, false, loader);
    } catch (ClassNotFoundException x) {
        fail(service,
             "Provider " + cn + " not found");
    }
   ...
    }
```

代码（7）使用 loader 也就是 AppClassLoader 加载具体的驱动实现类。至于 cn 是怎么来的，读者可以参见 LazyIterator 的 hasNext() 方法。

到这里再回想下，ContextClassLoader 的作用是为了破坏 Java 类加载委托机制，JDBC 规范定义了一个 JDBC 接口，然后使用 SPI 机制提供的一个叫做 ServiceLoader 的 Java 核心 API（rt.jar 里面提供）用来扫描服务实现类，服务实现者提供的 Jar，比如 MySQL 驱动则是放到我们的 classpath 下面，从上文知道默认线程上下文类加载器就是 AppClassLoader，所以例子里面没有显示在调用 ServiceLoader 前设置线程上下文类加载器为 AppClassLoader，ServiceLoader 内部则获取当前线程上下文类加载器（这里为

AppClassLoader）来加载服务实现者的类，这里加载了 classpath 下的MySQL 的驱动实现。

读者可以尝试在调用 ServiceLoader 的 load 方法前设置线程上下文类加器为 ExtclassLoder，代码如下：

```
Thread.currentThread().setContextClassLoader(main函数所在的
类.class.getClassLoader().getParent());
```

然后再运行本例子，设置后 ServiceLoader 内部则获取当前线程上下文类加载器为 ExtclassLoder，然后会尝试使用 ExtclassLoder 去查找 JDBC 驱动实现，而 ExtclassLoder 扫描类的路径为 JAVA_HOME/jre/lib/ext/，而这下面没有驱动实现的 Jar，所以不会查找到驱动。

总结下，当父类加载器需要加载子类加载器中的资源时，可以通过设置和获取线程上下文类加载器来实现。

Tomcat ClassLoader

本文tomcat版本为 apache-tomcat-8.5.12。

Tomcat classloader 的构造

首先我们打开 Tomcat 的源码 Bootstrap 类的 initClassLoaders 方法，代码如下。

```
        private void initClassLoaders() {
          try {
              //(1)创建commonLoader
              commonLoader = createClassLoader("common", null);
              if( commonLoader == null ) {
                  commonLoader=this.getClass().getClassLoader();
              }
              //(2)创建catalinaLoader，父类加载器为commonLoader
              catalinaLoader = createClassLoader("server",
    commonLoader);
              //(3)创建sharedLoader，父类加载器为commonLoader
              sharedLoader = createClassLoader("shared",
    commonLoader);
          } catch (Throwable t) {
            ...
          }
      }
```

代码（1）创建 commonLoader，这里父加载器传递为 null，但是内部会使用默认的类加载器 AppClassLoader 作为父加载器。

代码（2）创建 catalinaLoader，父类加载器为 commonLoader。

代码（3）创建 sharedLoader，父类加载器为 commonLoader。

现在我们已经知道这三个加载器关系了，下面看下 createClassLoader 是如何构造类加载器的。代码如下。

```
    private ClassLoader createClassLoader(String name,
ClassLoader parent)
        throws Exception {

        //(4)
        String value = CatalinaProperties.getProperty(name +
".loader");
        if ((value == null) || (value.equals("")))
            return parent;

        //(5)
        value = replace(value);
        List<Repository> repositories = new ArrayList<>();
        ...

        return ClassLoaderFactory.createClassLoader(repositories,
parent);
```

代码（4）获取 catalina.properties 中配置项分别为：common.loader、server.loader、shared.loader，用来设置对应类加载器扫描类的路径。默认内容为：

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

所以根据上面代码可知 commonLoader、serverLoader、sharedLoader 是同一个 ClassLoader。

代码（5）根据 common.loader、server.loader、shared.loader 的配置装饰类加载器所需的扫描路径。

具体创建类加载器的是 ClassLoaderFactory.createClassLoader，看下面的代码。

```
 public static ClassLoader createClassLoader(List<Repository>
repositories,
```

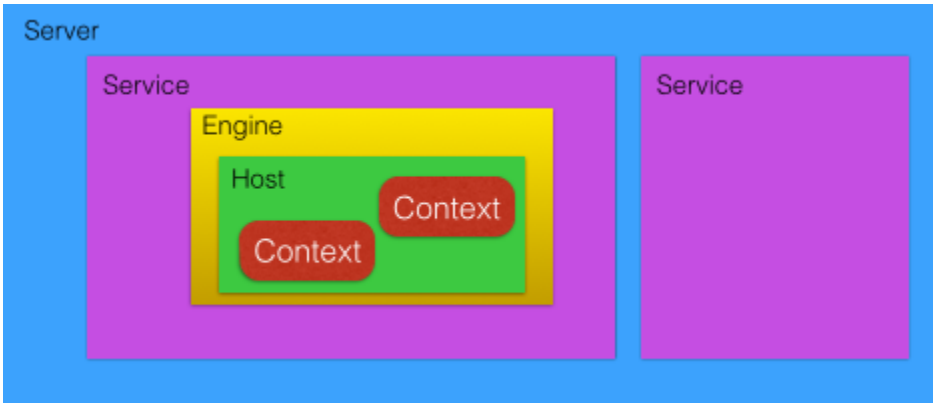
```
                                                    final ClassLoader
parent)
        throws Exception {
        ...
        //(6)
        return AccessController.doPrivileged(
                new PrivilegedAction<URLClassLoader>() {
                    @Override
                    public URLClassLoader run() {
                        if (parent == null)
                            return new URLClassLoader(array);
                        else
                            return new URLClassLoader(array,
parent);
                    }
                });
    }
```

由代码（6）可知创建 commonLoader 时 parent==null，这时候使用了单个参数的 URLClassLoader 创建了 URLClassLoader 类加载器作为 commonLoader，而 URLClassLoader 默认的父加载器为 AppClassLoader。

那么 URLClassLoader 为何物？其实我们上面讲解的 AppClassloader 和 ExtClassLoder 都是继承自 URLClassLoader。

到这里总结下，默认情况下 Tomcat `commonLoader`，`sharedLoader`，`catalinaLoader` 是同一个加载器，其类查找路径都是同一个地方。其实 catalinaLoader 主要工作是加载 Tomcat 本身启动所需要的类，而 sharedLoader 是下文将要讲解的 webAppclassloader 的父类，所以作用是加载所有应用都需要的类，而 commonLoader 作为 sharedL oader、catalinaLoader 的父类，自然设计目的是为了加载二者共享的类。所以如果能恰当的使用 Tomcat 中设计的这种策略，修改 catalina.properites 中三种加载器类加载路径，就会真正达到这种设计效果。

ok，下面看看 WebAppclassloader，我们知道在 Tomcat 中可以部署多个应用，每个应用使用自己独立的一个 WebAppclassloader 来加载应用，下面看看 WebAppclassloader 是如何创建的，为了能更好的切入 WebAppclassloader 创建的地方，这里先简单介绍下 Tomcat 的容器内部构造，如下图所示。



如上图，其中 Engine 是最大的容器，默认实现为 StandardEngineEngine，里面可以有若干个 Host。Host 的默认实现为 StandardHost，Host 的父容器为 Engine。每个 Host 容器里面有若干 Context 容器，默认为 StandardContext，context 容器的父容器为 Host，每个 Context 代表一个应用。而创建 WebAppclassloader 的就是在 StandardContext 的 startInternal 方法，代码如下。

```
    protected synchronized void startInternal() throws
LifecycleException {
        ...
        //(7)
        if (getLoader() == null) {
            WebappLoader webappLoader = new
WebappLoader(getParentClassLoader());
            webappLoader.setDelegate(getDelegate());
            setLoader(webappLoader);
        }

        ...
        try {
            if (ok) {
                //(8)
                Loader loader = getLoader();
                if (loader instanceof Lifecycle) {
                    ((Lifecycle) loader).start();
                }
        ...
            }
        }
     }
```

代码（1）创建了 WebappLoader 对象，并调用 getParentClassLoader() 获取到 sharedLoader。

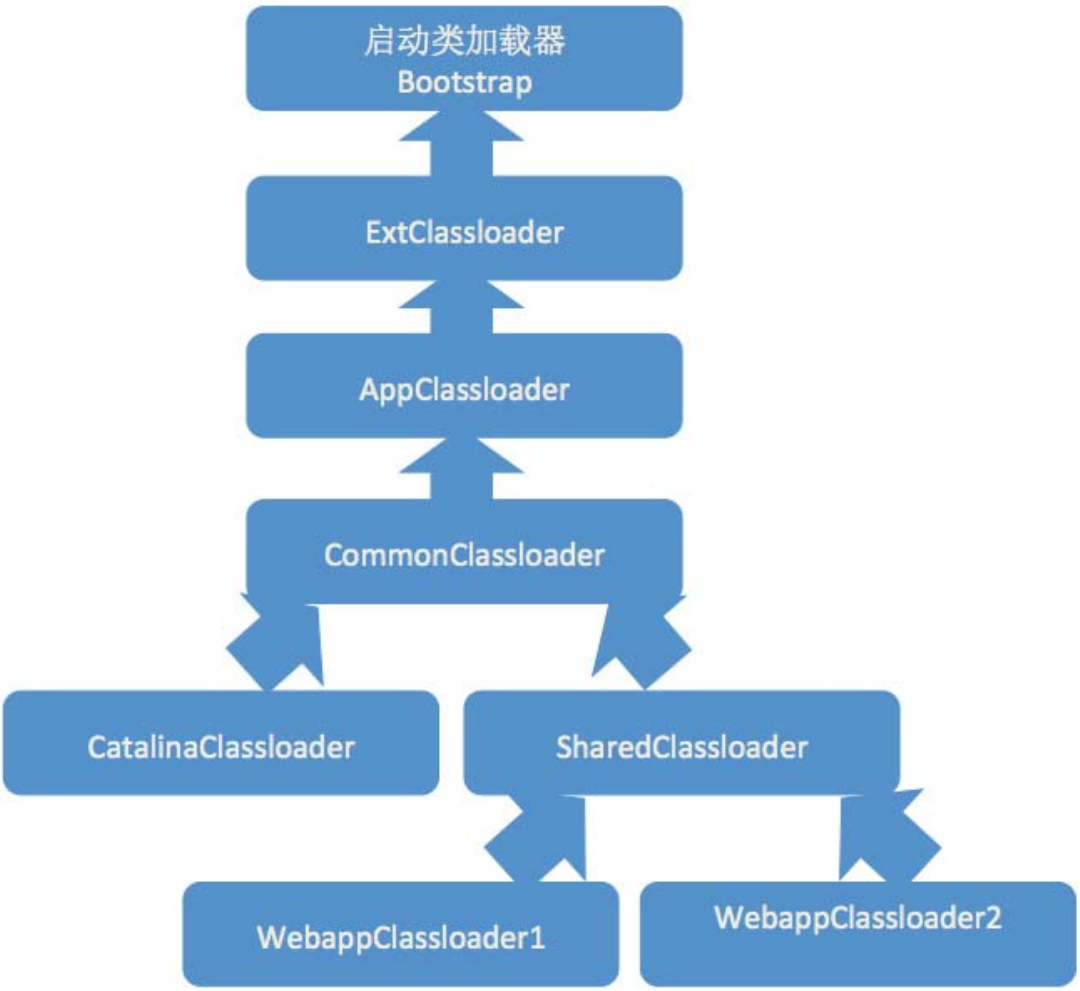
代码（8）则调用 WebappLoader 的 start 方法创建当前应用的 WebappClassLoader 加载器，下面具体看看代码。

```
protected void startInternal() throws LifecycleException {

        ...
        try {
            //(8)
            classLoader = createClassLoader();
            classLoader.setResources(context.getResources());
            classLoader.setDelegate(this.delegate);
        }
        ...
 }
```

```
    private WebappClassLoaderBase createClassLoader()
        throws Exception {

        //(9)
        Class<?> clazz = Class.forName(loaderClass);
        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;
    }
```

代码（9）创建并实例化一个 webappClassLoader，并设置父类加载器为 sharedLoader。需要注意的是 Tomcat 8 里面的 `loaderClass = "org.apache.catalina.loader.ParallelWebappClassLoader"` 而不再是 `"org.apache.catalina.loader.WebappClassLoader"`。

至此创建了应用的类加载器，由于每个 standardcontext 对应一个 Web 应用，所以不同的应用都有不同的
WebappClassLoader，共同点是他们的父加载器都是 sharedLoader。下面是 Tomcat 类加载器关系图。



WebappClassLoader 类加载原理

下面说说 WebappClassLoader 类加载器加载流程，具体是看 loadClass 方法。

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

        synchronized (getClassLoadingLock(name)) {
            if (log.isDebugEnabled())
                log.debug("loadClass(" + name + ", " + resolve +
")");
            Class<?> clazz = null;

            // Log access to stopped class loader
            checkStateForClassLoading(name);

            // (10) 首先检查WebappClassloader缓存中是否已经加载该类
            clazz = findLoadedClass0(name);
            if (clazz != null) {
                if (log.isDebugEnabled())
                    log.debug("  Returning class from cache");
                if (resolve)
                    resolveClass(clazz);
                return (clazz);
```

```
            }

            // (11) 看jvm缓存是否已经加载该类
            clazz = findLoadedClass(name);
            if (clazz != null) {
                if (log.isDebugEnabled())
                    log.debug("  Returning class from cache");
                if (resolve)
                    resolveClass(clazz);
                return (clazz);
            }

            //（12）为了避免webapp覆盖Java SE classes，这里尝试使用
ExtClassLoader进行加载
            String resourceName = binaryNameToPath(name, false);
            ClassLoader javaseLoader = getJavaseClassLoader();
            boolean tryLoadingFromJavaseLoader;
            try {

                tryLoadingFromJavaseLoader =
(javaseLoader.getResource(resourceName) != null);
            } catch (ClassCircularityError cce) {
                tryLoadingFromJavaseLoader = true;
            }
            //(13)
            if (tryLoadingFromJavaseLoader) {
                try {
                    clazz = javaseLoader.loadClass(name);
                    if (clazz != null) {
                        if (resolve)
                            resolveClass(clazz);
                        return (clazz);
                    }
                } catch (ClassNotFoundException e) {
                    // Ignore
                }
            }

            //(14)
            boolean delegateLoad = delegate || filter(name,
true);

            // (14.1) 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) {
                    // Ignore
                }
            }

            // (15) webapp本地搜索
            if (log.isDebugEnabled())
                log.debug("  Searching local repositories");
            try {
                clazz = findClass(name);
                if (clazz != null) {
                    if (log.isDebugEnabled())
                        log.debug("  Loading class from local
repository");
                    if (resolve)
                        resolveClass(clazz);
                    return (clazz);
                }
            } catch (ClassNotFoundException e) {
                // Ignore
            }

            // (16) Delegate to parent unconditionally
            if (!delegateLoad) {
                if (log.isDebugEnabled())
                    log.debug("  Delegating to parent classloader
at end: " + 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) {
                    // Ignore
                }
            }
        }

        throw new ClassNotFoundException(name);
    }
```

代码（10）尝试在 webapp 的缓存里面查找，如果存在则直接返回。

代码（11）尝试在 JVM 缓存查找，如果存在则直接返回。

代码（12）、（13）为了避免 webapp 覆盖 Java SE classes，这里尝试使用 ExtClassLoader 进行加载。Tomcat 8 对这个做了个优化，之前是直接调用 loadClass 方法进行尝试加载，如果不存在则抛出 ClassNotFoundException 异常，这个异常虽然会被 catch调用，但是抛出 ClassNotFoundException 异常的代价非常高，所以 Tomcat 8 做了个优化，就是先调用 getResource 判断要加载的类在 ExtClassLoader 搜索路径下是否存在（getResource 的调用不会产生昂贵的代价），存在才调用 loadClass 进行加载。

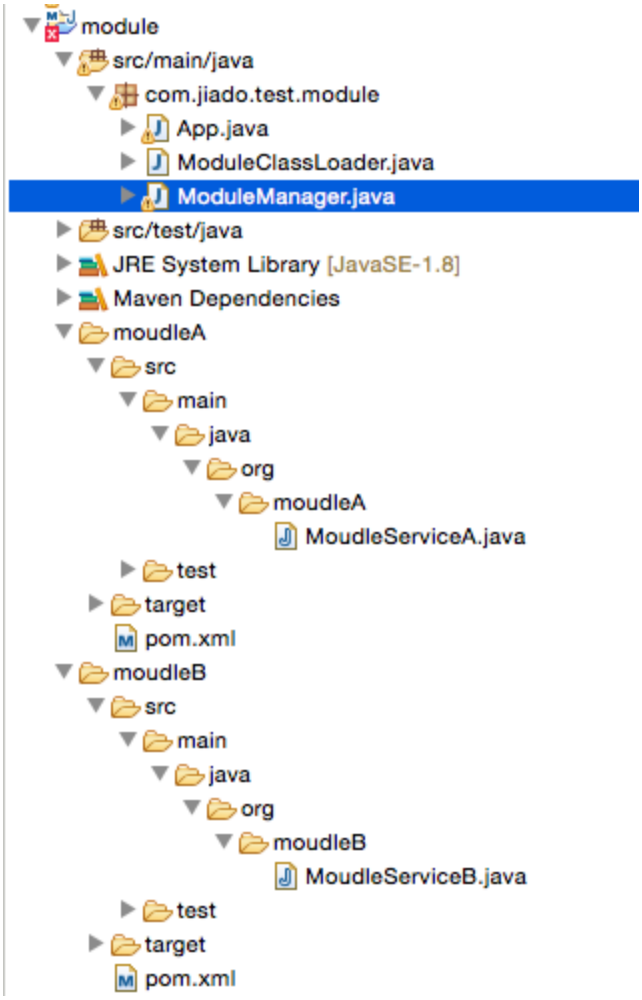
代码（14）如果设置了委托机制则委托给父类加载进行加载。

代码（15），调用 findClass 在 Web 应用的 lib 目录下进行查找。

代码（16）表示，如果步骤(（14）没有设置委托则这时候再让父加载器进行加载，这个时候也是违背类加载器委托模型的一个例子。

使用自定义类加载器实现模块隔离

本节我们实现一个简单模块隔离功能，每个模块使用一个类加载器进行加载，每个模块独立作为一个 Jar 包对外提供功能。我们把每个模块 Jar 里面的类加载到一个内存 map，然后再创建一个自定义类加载器，重写 loadClass 方法，当加载类时候优先在内存 map 里面查找，如果存在则直接返回，否则调用默认加载方法。如下图所示。



本 Demo 使用 Maven 的聚合功能，里面含有两个子模块，分别为 moudleA 和 moudleB。主工程里面并没有添加这两个子模块的依赖。 moudleA 和 moudleB 里面分别有一个 service 类。

```
public class MoudleServiceA {
    public String sayHelloServiceA(){
        return "---sayHelloServiceA---";
    }
}
```

```
public class MoudleServiceB {
    public String sayHelloServiceB(){
        return "---sayHelloServiceB---";
    }
}
```

分别在两个子模块里面执行 mvn clean package ，生成 moudleB-0.0.1-SNAPSHOT.jar 和 moudleA-0.0.1-SNAPSHOT.jar。

主工程里面的 ModuleManager 是核心类，实现了从不同模块加载 Jar 里面的类到内存 map 的功能，核心代码如下。

```
public  class ModuleManager {

    private ExecutorService executor =
Executors.newFixedThreadPool(8);
    private Map<String,Class> cache = new ConcurrentHashMap<>();
    private List<String> moudleList= new ArrayList<String>();

    private String getClassName(JarEntry entry) {
        String entryName = entry.getName();

        if (!entryName.endsWith(".class")) {
            return null;
        }
        if (entryName.charAt(0) == '/') {
            entryName = entryName.substring(1);
        }
        entryName = entryName.replace("/", ".");
        return entryName.substring(0, entryName.length() - 6);
    }

    public void init(){

        System.out.println("----begin load  All module----");
        List<Future<String>> futureList = new
ArrayList<Future<String>>();
        for(String moudle:moudleList){
            Future<String> future = executor.submit(new
Callable<String>() {

                @Override
                public String call() throws Exception {
                    try{
                        URL[] moduleUrl = new URL[]{new
URL("file://"+ moudle)};
                        @SuppressWarnings("resource")
                        URLClassLoader classLoader = new
URLClassLoader(moduleUrl );
                        @SuppressWarnings("resource")
                        JarFile jar = new JarFile(new
File(moudle));
                         Enumeration<JarEntry> entries =
jar.entries();
                            while (entries.hasMoreElements()) {
                                JarEntry entry =
entries.nextElement();
                                String className =
getClassName(entry);
                                if (className == null ) {
                                    continue;
                                }
```

```
                                    try {
                                        Class<?> clazz =
classLoader.loadClass(className);
                                        cache.put(className, clazz);
                                    } catch (Throwable t) {

//System.out.println(t.getLocalizedMessage());
                                    }
                                }
                            }catch(Exception e){

System.out.println(e.getLocalizedMessage());
                            }

                            return moudle;
                        }

                    });
                    futureList.add(future);
                }

                for(Future<String> future:futureList){
                    try {
                    String moduleName = future.get();
                    System.out.println("---load moudle " + moduleName + "
ok" );
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } catch (ExecutionException e) {
                        e.printStackTrace();
                    }
                }
                executor.shutdown();
                System.out.println("----end load All module----");

            }
}
```

其中 executor 是个线程池，用来并发加载多个模块 Jar。

cache 用来存放所有 Jar 里面的 Class 对象。

moudleList 是需要加载的模块的 Jar 的路径。

init 方法是核心方法，其中 future 的 call 方法针对每个模块新建了一个 URLClassLoader 加载器用来加载该模块的类，并把加载的类放入内存 map。

类 ModuleClassLoader 是自定义的一个类加载器，代码如下。

```
public class ModuleClassLoader extends URLClassLoader {

    private ModuleManager manager = new ModuleManager();

    public ModuleClassLoader(URL[] urls,ModuleManager manager) {
        super(urls);
        this.manager = manager;
    }
    protected Class<?> loadClass(String name, boolean resolve)
throws ClassNotFoundException {

        if(manager.getCache().containsKey(name)){
            return manager.getCache().get(name);
        }else{
            return super.loadClass(name, resolve);
        }
    }
}
```

重写了 loadClass，优先从内存 map 查找要加载的类。

App类是一个测试类，代码如下。

```
    public static void main(String[] args) throws
ClassNotFoundException, NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException,
InvocationTargetException, InstantiationException {

        System.out.println("Hello World!");
        ModuleManager manager = new ModuleManager();
        List<String> jarList = new ArrayList<String>();
        jarList.add(

"/Users/zhuizhumengxiang/workspace/mytool/moudleexample/module/mo
udleA/target/moudleA-0.0.1-SNAPSHOT.jar");
        jarList.add(

"/Users/zhuizhumengxiang/workspace/mytool/moudleexample/module/mo
udleB/target/moudleB-0.0.1-SNAPSHOT.jar");
        manager.setMoudleList(jarList);
        manager.init();

        ModuleClassLoader classLoader = new
ModuleClassLoader(((URLClassLoader)
App.class.getClassLoader()).getURLs(),manager);

        Class serviceA =
classLoader.loadClass("org.moudleA.MoudleServiceA");
        Method sayHelloServiceA =
```

```
        serviceA.getMethod("sayHelloServiceA");
                String result = (String)
        sayHelloServiceA.invoke(serviceA.newInstance(), null);
                System.out.println(result);
            }
```

首先创建了一个 ModuleManager 对象，设置需要加载模块的 Jar 列表，然后调用 init 方法使用不同的类加载器加载不同的 Jar 里面的 class 到内存 map。

然后创建了一个 ModuleClassLoader 的实例，其内部维护这内存 map，当使用 `classLoader.loadClass("org.moudleA.MoudleServiceA");` 加载 MoudleServiceA 时候，会有些去内存 map 看是否有该类，如果发现有，则返回 MoudleServiceA 的 Class 对象，然后创建一个 MoudleServiceA 的实例，并通过反射调用了 sayHelloServiceA 方法。

总结

Java 类加载器作为 Java 家族核心成员，了解它的原理，能灵活自定义类加载器去实现自己的功能显得尤为重要。