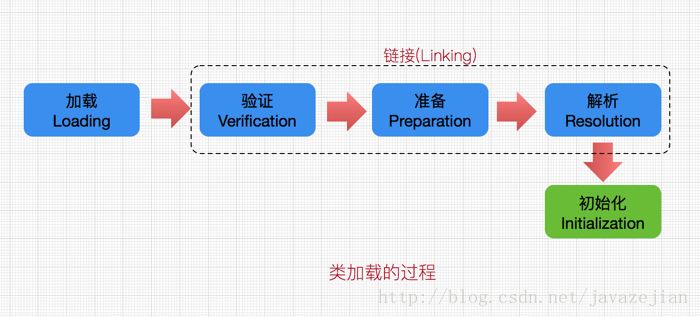
**装载验证流程**

每个编写的”.java”拓展名类文件都存储着需要执行的程序逻辑，这些”.java”文件经过[Java编译器](https://so.csdn.net/so/search?q=Java%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020" \t "_blank)编译成拓展名为”.class”的文件，  
”.class”文件中保存着Java代码经转换后的虚拟机指令，当需要使用某个类时，虚拟机将会加载它的”.class”文件，并创建对应的class对象，  
将class文件加载到虚拟机的内存，这个过程称为类加载，这里我们需要了解一下类加载的过程，如下：



**加载**

类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象

**链接**

**验证**

目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种验证，文件格式验证，  
元数据验证，字节码验证，符号引用验证。

* 文件格式的验证

是否以0xCAFEBABE开头，版本号是否合理

* 元数据验证

是否有父类，继承了final类？非抽象类实现了所有的抽象方法

* 字节码验证 (很复杂)

运行检查，栈数据类型和操作码数据参数吻合，跳转指令指定到合理的位置

* 符号引用验证

常量池中描述类是否存在，访问的方法或字段是否存在且有足够的权限

**准备**

为类变量(即static修饰的字段变量)分配内存并且设置该类变量的初始值即0(如static int i=5;这里只将i初始化为0，  
至于5的值将在初始化时赋值)，这里不包含用final修饰的static，因为final在编译的时候就会分配了，注意这里不会为实例变量分配初始化，  
类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

分配内存，并为类设置初始值 （方法区中）

* public static int v=1;
* 在准备阶段中，v会被设置为0
* 在初始化的中才会被设置为1
* 对于static final类型，在准备阶段就会被赋上正确的值
* public static final int v=1;

**解析**

主要将常量池中的符号引用替换为直接引用的过程。符号引用就是一组符号来描述目标，可以是任何字面量，  
而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，  
接口方法解析(这里涉及到字节码变量的引用，如需更详细了解，可参考《深入Java虚拟机》)。

**初始化**

类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量(如前面只初始化了默认值的static变量将会在这个阶段赋值，  
成员变量也将被初始化)。

* 执行类构造器
  + static变量 赋值语句
  + static{}语句
* 子类的调用前保证父类的被调用
* 是线程安全的

这便是类加载的5个过程，而类加载器的任务是根据一个类的全限定名来读取此类的二进制字节流到JVM中，  
然后转换为一个与目标类对应的java.lang.Class对象实例，在虚拟机提供了3种类加载器，  
引导（Bootstrap）类加载器、扩展（Extension）类加载器、系统（System）类加载器（也称应用类加载器）

**加载器**

**启动（Bootstrap）类加载器**

启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，  
它负责将 <JAVA\_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中，  
注意必由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，  
即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类)。

**扩展（Extension）类加载器**

扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher$ExtClassLoader类，由Java语言实现的，  
是Launcher的静态内部类，它负责加载<JAVA\_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定位路径中的类库，  
开发者可以直接使用标准扩展类加载器。

|  |
| --- |
| *//ExtClassLoader类中获取路径的代码*  private static File[] getExtDirs() {  *//加载<JAVA\_HOME>/lib/ext目录中的类库*  String s = System.getProperty("java.ext.dirs");  File[] dirs;  if (s != null) {  StringTokenizer st =  new StringTokenizer(s, File.pathSeparator);  int count = st.countTokens();  dirs = new File[count];  for (int i = 0; i < count; i++) {  dirs[i] = new File(st.nextToken());  }  } else {  dirs = new File[0];  }  return dirs;  } |

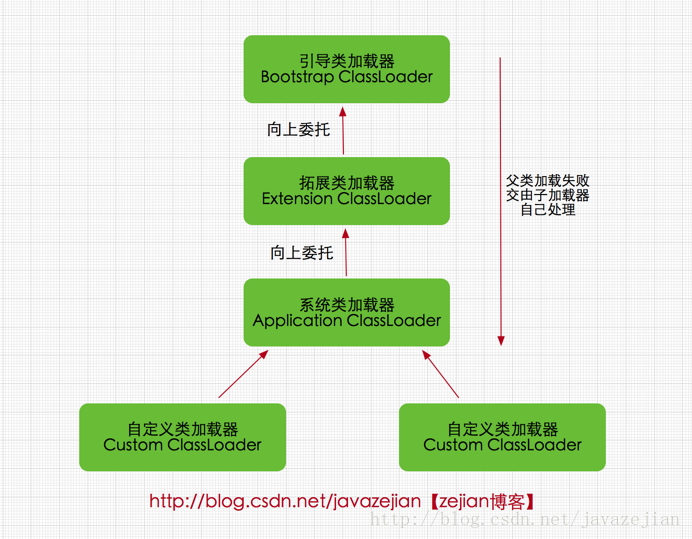
**系统（System）类加载器**

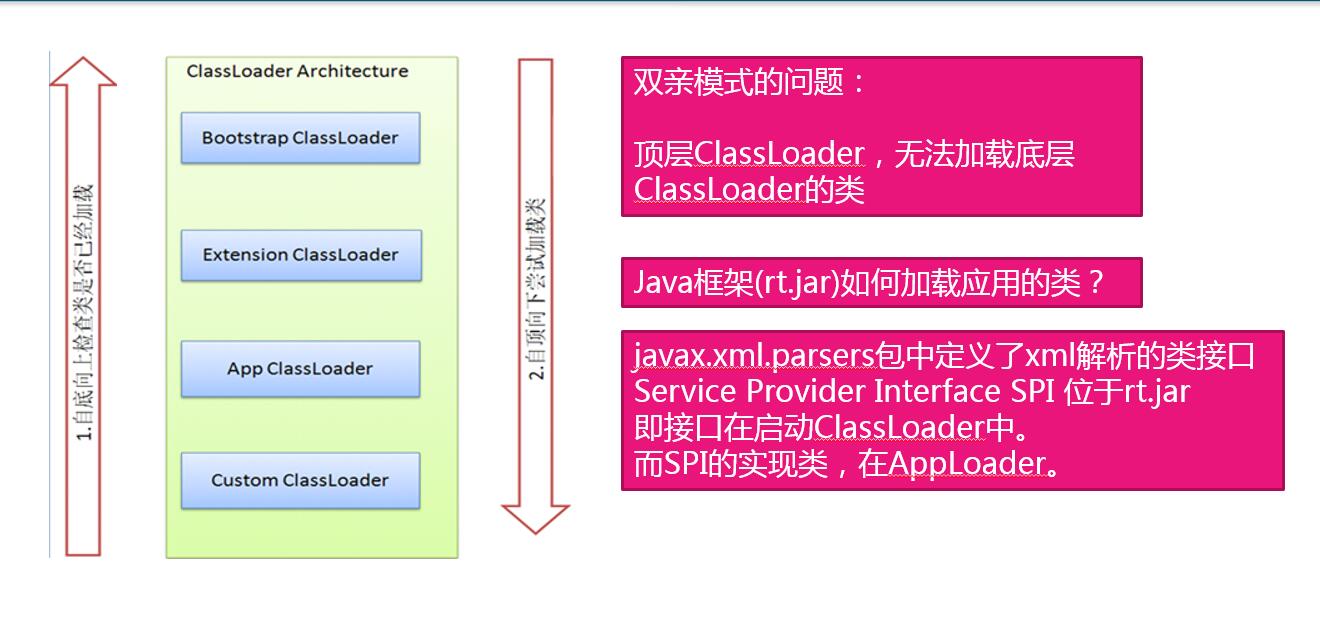
也称应用程序加载器是指 Sun公司实现的sun.misc.Launcher$AppClassLoader。  
它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，  
开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认的类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。  
在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，  
需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，  
而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它一种任务委派模式，下面我们进一步了解它。

**双亲委派模式**

**双亲委派模式工作原理**

双亲委派模式要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器，  
请注意双亲委派模式中的父子关系并非通常所说的类继承关系，而是采用组合关系来复用父类加载器的相关代码，  
类加载器间的关系如下：





双亲委派模式是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，  
而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，  
如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，  
即每个儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完成，这不就是传说中的实力坑爹啊？  
那么采用这种模式有啥用呢?

**双亲委派模式优势**

采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，  
当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。其次是考虑到安全因素，java核心api中定义类型不会被随意替换，  
假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，  
发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。  
可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，  
经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，  
最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出如下异常

java.lang.SecurityException: Prohibited package name: java.lang

**编写自己的类加载器**

通过前面的分析可知，实现自定义类加载器需要继承ClassLoader或者URLClassLoader，继承ClassLoader则需要自己重写findClass()方法并编写加载逻辑，  
继承URLClassLoader则可以省去编写findClass()方法以及class文件加载转换成字节码流的代码。那么编写自定义类加载器的意义何在呢？

* 当class文件不在ClassPath路径下，默认系统类加载器无法找到该class文件，  
  在这种情况下我们需要实现一个自定义的ClassLoader来加载特定路径下的class文件生成class对象。
* 当一个class文件是通过网络传输并且可能会进行相应的加密操作时，需要先对class文件进行相应的解密后再加载到JVM内存中，  
  这种情况下也需要编写自定义的ClassLoader并实现相应的逻辑。
* 当需要实现热部署功能时(一个class文件通过不同的类加载器产生不同class对象从而实现热部署功能)，  
  需要实现自定义ClassLoader的逻辑。

**自定义File类加载器**

这里我们继承ClassLoader实现自定义的特定路径下的文件类加载器并加载编译后DemoObj.class，源码代码如下

|  |
| --- |
| public class DemoObj {  @Override  public String toString() {  return "I am DemoObj";  }  } |

|  |
| --- |
| */\*\**  *\* Created by zejian on 2017/6/21.*  *\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]*  *\*/*  public class FileClassLoader extends ClassLoader {  private String rootDir;  public FileClassLoader(String rootDir) {  this.rootDir = rootDir;  }  */\*\**  *\* 编写findClass方法的逻辑*  *\* @param name*  *\* @return*  *\* @throws ClassNotFoundException*  *\*/*  @Override  protected Class<?> findClass(String name) throws ClassNotFoundException {  *// 获取类的class文件字节数组*  byte[] classData = getClassData(name);  if (classData == null) {  throw new ClassNotFoundException();  } else {  *//直接生成class对象*  return defineClass(name, classData, 0, classData.length);  }  }  */\*\**  *\* 编写获取class文件并转换为字节码流的逻辑*  *\* @param className*  *\* @return*  *\*/*  private byte[] getClassData(String className) {  *// 读取类文件的字节*  String path = classNameToPath(className);  try {  InputStream ins = new FileInputStream(path);  ByteArrayOutputStream baos = new ByteArrayOutputStream();  int bufferSize = 4096;  byte[] buffer = new byte[bufferSize];  int bytesNumRead = 0;  *// 读取类文件的字节码*  while ((bytesNumRead = ins.read(buffer)) != -1) {  baos.write(buffer, 0, bytesNumRead);  }  return baos.toByteArray();  } catch (IOException e) {  e.printStackTrace();  }  return null;  }  */\*\**  *\* 类文件的完全路径*  *\* @param className*  *\* @return*  *\*/*  private String classNameToPath(String className) {  return rootDir + File.separatorChar  + className.replace('.', File.separatorChar) + ".class";  }  public static void main(String[] args) throws ClassNotFoundException {  String rootDir="/Users/zejian/Downloads/Java8\_Action/src/main/java/";  *//创建自定义文件类加载器*  FileClassLoader loader = new FileClassLoader(rootDir);  try {  *//加载指定的class文件*  Class<?> object1=loader.loadClass("com.zejian.classloader.DemoObj");  System.out.println(object1.newInstance().toString());  *//输出结果:I am DemoObj*  } catch (Exception e) {  e.printStackTrace();  }  }  } |

显然我们通过getClassData()方法找到class文件并转换为字节流，并重写findClass()方法，  
利用defineClass()方法创建了类的class对象。在main方法中调用了loadClass()方法加载指定路径下的class文件，由于启动类加载器、  
拓展类加载器以及系统类加载器都无法在其路径下找到该类，因此最终将有自定义类加载器加载，即调用findClass()方法进行加载。  
如果继承URLClassLoader实现，那代码就更简洁了，如下：

|  |
| --- |
| */\*\**  *\* Created by zejian on 2017/6/21.*  *\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]*  *\*/*  public class FileUrlClassLoader extends URLClassLoader {  public FileUrlClassLoader(URL[] urls, ClassLoader parent) {  super(urls, parent);  }  public FileUrlClassLoader(URL[] urls) {  super(urls);  }  public FileUrlClassLoader(URL[] urls, ClassLoader parent, URLStreamHandlerFactory factory) {  super(urls, parent, factory);  }  public static void main(String[] args) throws ClassNotFoundException, MalformedURLException {  String rootDir="/Users/zejian/Downloads/Java8\_Action/src/main/java/";  *//创建自定义文件类加载器*  File file = new File(rootDir);  *//File to URI*  URI uri=file.toURI();  URL[] urls={uri.toURL()};  FileUrlClassLoader loader = new FileUrlClassLoader(urls);  try {  *//加载指定的class文件*  Class<?> object1=loader.loadClass("com.zejian.classloader.DemoObj");  System.out.println(object1.newInstance().toString());  *//输出结果:I am DemoObj*  } catch (Exception e) {  e.printStackTrace();  }  }  } |

非常简洁除了需要重写构造器外无需编写findClass()方法及其class文件的字节流转换逻辑。

**自定义网络类加载器**

自定义网络类加载器，主要用于读取通过网络传递的class文件（在这里我们省略class文件的解密过程），并将其转换成字节流生成对应的class对象，如下

|  |
| --- |
| */\*\**  *\* Created by zejian on 2017/6/21.*  *\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]*  *\*/*  public class NetClassLoader extends ClassLoader {  private String url;*//class文件的URL*  public NetClassLoader(String url) {  this.url = url;  }  @Override  protected Class<?> findClass(String name) throws ClassNotFoundException {  byte[] classData = getClassDataFromNet(name);  if (classData == null) {  throw new ClassNotFoundException();  } else {  return defineClass(name, classData, 0, classData.length);  }  }  */\*\**  *\* 从网络获取class文件*  *\* @param className*  *\* @return*  *\*/*  private byte[] getClassDataFromNet(String className) {  String path = classNameToPath(className);  try {  URL url = new URL(path);  InputStream ins = url.openStream();  ByteArrayOutputStream baos = new ByteArrayOutputStream();  int bufferSize = 4096;  byte[] buffer = new byte[bufferSize];  int bytesNumRead = 0;  *// 读取类文件的字节*  while ((bytesNumRead = ins.read(buffer)) != -1) {  baos.write(buffer, 0, bytesNumRead);  }  *//这里省略解密的过程.......*  return baos.toByteArray();  } catch (Exception e) {  e.printStackTrace();  }  return null;  }  private String classNameToPath(String className) {  *// 得到类文件的URL*  return url + "/" + className.replace('.', '/') + ".class";  }  } |

比较简单，主要是在获取字节码流时的区别，从网络直接获取到字节流再转车字节数组然后利用defineClass方法创建class对象，  
如果继承URLClassLoader类则和前面文件路径的实现是类似的，无需担心路径是filePath还是Url，  
因为URLClassLoader内的URLClassPath对象会根据传递过来的URL数组中的路径判断是文件还是jar包，  
然后根据不同的路径创建FileLoader或者JarLoader或默认类Loader去读取对于的路径或者url下的class文件。

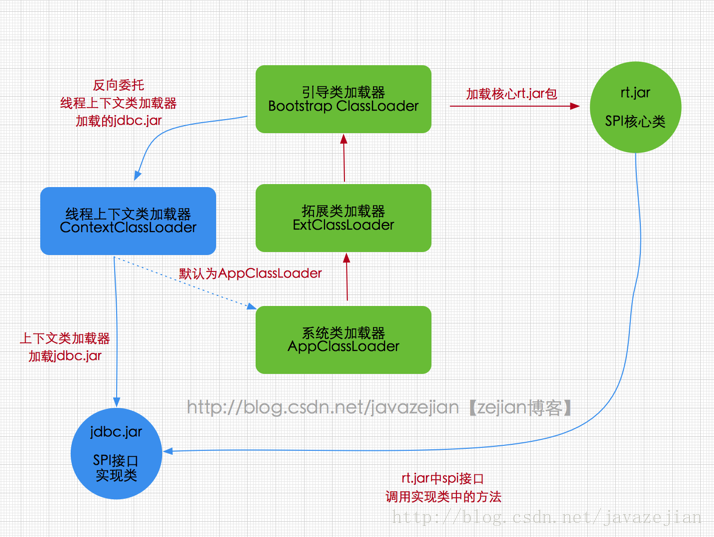
**热部署类加载器**

所谓的热部署就是利用同一个class文件不同的类加载器在内存创建出两个不同的class对象(关于这点的原因前面已分析过，即利用不同的类加载实例)，  
由于JVM在加载类之前会检测请求的类是否已加载过(即在loadClass()方法中调用findLoadedClass()方法)，如果被加载过，则直接从缓存获取，  
不会重新加载。注意同一个类加载器的实例和同一个class文件只能被加载器一次，多次加载将报错，  
因此我们实现的热部署必须让同一个class文件可以根据不同的类加载器重复加载，以实现所谓的热部署。  
实际上前面的实现的FileClassLoader和FileUrlClassLoader已具备这个功能，但前提是直接调用findClass()方法，  
而不是调用loadClass()方法，因为ClassLoader中loadClass()方法体中调用findLoadedClass()方法进行了检测是否已被加载  
，因此我们直接调用findClass()方法就可以绕过这个问题，当然也可以重新loadClass方法，但强烈不建议这么干。利用FileClassLoader类测试代码如下：

|  |
| --- |
| public static void main(String[] args) throws ClassNotFoundException {  String rootDir="/Users/zejian/Downloads/Java8\_Action/src/main/java/";  *//创建自定义文件类加载器*  FileClassLoader loader = new FileClassLoader(rootDir);  FileClassLoader loader2 = new FileClassLoader(rootDir);  try {  *//加载指定的class文件,调用loadClass()*  Class<?> object1=loader.loadClass("com.zejian.classloader.DemoObj");  Class<?> object2=loader2.loadClass("com.zejian.classloader.DemoObj");  System.out.println("loadClass->obj1:"+object1.hashCode());  System.out.println("loadClass->obj2:"+object2.hashCode());  *//加载指定的class文件,直接调用findClass(),绕过检测机制，创建不同class对象。*  Class<?> object3=loader.findClass("com.zejian.classloader.DemoObj");  Class<?> object4=loader2.findClass("com.zejian.classloader.DemoObj");  System.out.println("loadClass->obj3:"+object3.hashCode());  System.out.println("loadClass->obj4:"+object4.hashCode());  */\*\**  *\* 输出结果:*  *\* loadClass->obj1:644117698*  *loadClass->obj2:644117698*  *findClass->obj3:723074861*  *findClass->obj4:895328852*  *\*/*  } catch (Exception e) {  e.printStackTrace();  }  } |

**双亲委派模型的破坏者-线程上下文类加载器**

在Java应用中存在着很多服务提供者接口（Service Provider Interface，SPI），这些接口允许第三方为它们提供实现，  
如常见的 SPI 有 JDBC、JNDI等，这些 SPI 的接口属于 Java 核心库，一般存在rt.jar包中，由Bootstrap类加载器加载，  
而 SPI 的第三方实现代码则是作为Java应用所依赖的 jar 包被存放在classpath路径下，  
由于SPI接口中的代码经常需要加载具体的第三方实现类并调用其相关方法，但SPI的核心接口类是由引导类加载器来加载的，  
而Bootstrap类加载器无法直接加载SPI的实现类，同时由于双亲委派模式的存在，Bootstrap类加载器也无法反向委托AppClassLoader加载器SPI的实现类。  
在这种情况下，我们就需要一种特殊的类加载器来加载第三方的类库，而线程上下文类加载器就是很好的选择。  
线程上下文类加载器（contextClassLoader）是从 JDK 1.2 开始引入的，我们可以通过java.lang.Thread类中的getContextClassLoader()和  
setContextClassLoader(ClassLoader cl)方法来获取和设置线程的上下文类加载器。如果没有手动设置上下文类加载器，  
线程将继承其父线程的上下文类加载器，初始线程的上下文类加载器是系统类加载器（AppClassLoader）,  
在线程中运行的代码可以通过此类加载器来加载类和资源，如下图所示，以jdbc.jar加载为例



从图可知rt.jar核心包是有Bootstrap类加载器加载的，其内包含SPI核心接口类，由于SPI中的类经常需要调用外部实现类的方法，  
而jdbc.jar包含外部实现类(jdbc.jar存在于classpath路径)无法通过Bootstrap类加载器加载，  
因此只能委派线程上下文类加载器把jdbc.jar中的实现类加载到内存以便SPI相关类使用。  
显然这种线程上下文类加载器的加载方式破坏了“双亲委派模型”，它在执行过程中抛弃双亲委派加载链模式，使程序可以逆向使用类加载器，  
当然这也使得Java类加载器变得更加灵活。为了进一步证实这种场景，不妨看看DriverManager类的源码，DriverManager是Java核心rt.jar包中的类，  
该类用来管理不同数据库的实现驱动即Driver，它们都实现了Java核心包中的java.sql.Driver接口，如mysql驱动包中的com.mysql.jdbc.Driver，  
这里主要看看如何加载外部实现类，在DriverManager初始化时会执行如下代码

|  |
| --- |
| *//DriverManager是Java核心包rt.jar的类*  public class DriverManager {  *//省略不必要的代码*  static {  loadInitialDrivers();*//执行该方法*  println("JDBC DriverManager initialized");  }  *//loadInitialDrivers方法*  private static void loadInitialDrivers() {  sun.misc.Providers()  AccessController.doPrivileged(new PrivilegedAction<Void>() {  public Void run() {  *//加载外部的Driver的实现类*  ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);  *//省略不必要的代码......*  }  });  } |





Thread. setContextClassLoader()

* 上下文加载器
* 是一个角色
* 用以解决顶层ClassLoader无法访问底层ClassLoader的类的问题
* 基本思想是，在顶层ClassLoader中，传入底层ClassLoader的实例

双亲模式的破坏

* 双亲模式是默认的模式，但不是必须这么做
* Tomcat的WebappClassLoader 就会先加载自己的Class，找不到再委托parent
* OSGi的ClassLoader形成网状结构，根据需要自由加载Class