**一、ClassLoader基本概念**

平常用文本编辑器或者是IDEA编写的程序都是.java格式的文件，这是最基础的源码。Java程序不是一个可执行文件，而是由许多独立的类文件组成的，每一个文件对应一个Java类。

**类加载器分类**

从JVM的角度分

1. Bootstrap ClassLoader由c++语言实现，属于JVM的一部分。
2. 其他类加载器由Java语言实现，继承自java.lang.ClassLoader

从Java应用程序的角度分

1. Bootstrap ClassLoader，负责加载jre/lib目录下的核心类库
2. Extension ClassLoader，负责加载jre/lib/ext目录下的核心类库
3. AppClassLoader，负责加载app classpath目录下的类库
4. 用户自定义ClassLoader

**二、ClassLoader加载流程**

加载顺序：Bootstrap ClassLoader->ExtClassLoader->AppClassLoader

当一个程序运行的时候，JVM启动，运行Bootstrap ClassLoader，该类加载器加载Java核心API（ExtClassLoader和AppClassLoader也在此时被加载），然后调用ExtClassLoader加载扩展API，最后AppClassLoader加载CLASSPATH目录下定义的Class。

public class Launcher {  
 private static URLStreamHandlerFactory factory = new Launcher.Factory();  
 private static Launcher launcher = new Launcher();  
 private static String bootClassPath = System.getProperty("sun.boot.class.path");  
 private ClassLoader loader;  
 private static URLStreamHandler fileHandler;  
  
 public static Launcher getLauncher() {  
 return launcher;  
 }  
  
 public Launcher() {  
 // 创建ExtClassLoader

Launcher.ExtClassLoader var1;  
 try {  
 var1 = Launcher.ExtClassLoader.getExtClassLoader();  
 } catch (IOException var10) {  
 throw new InternalError("Could not create extension class loader");  
 }

// 创建用于启动应用程序的类加载器  
 try {  
 this.loader = Launcher.AppClassLoader.getAppClassLoader(var1);  
 } catch (IOException var9) {  
 throw new InternalError("Could not create application class loader");  
 }  
  
 // 设置AppClassLoader为线程上下文类加载器

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

}

}

public ClassLoader getClassLoader() {  
 return this.loader;  
 }

static class ExtClassLoader extends URLClassLoader {}

static class AppClassLoader extends URLClassLoader {}

从源码可以看出：

* Launcher初始化了ExtClassLoader和AppClassLoader。
* Launcher中并没有看见BootstrapClassLoader，但通过System.getProperty("sun.boot.class.path")得到了字符串bootClassPath,这个应该就是BootstrapClassLoader加载的jar包路径。

自此我们知道了BootstrapClassLoader、ExtClassLoader、AppClassLoader实际是查阅响应的环境属性sun.boot.class.path、java.ext.path、java.class.path来加载资源文件的。

测试：

先创建一个Test.java，然后再编写一个ClassLoaderTest.java，获取Test.java文件的类加载器，然后打印出来ClassLoader is:sun.misc.Launcher$AppClassLoader@18b4aac2，说明自己编写的类文件是由AppClassLoader加载的。那么int.class或者String.class的加载是由谁完成的呢？

ClassLoader cl01 = int.class.getClassLoader();  
System.***out***.println(cl01.toString());

运行报错：

Exception in thread "main" java.lang.NullPointerException

at com.sglm.ClassLoaderTest.main(ClassLoaderTest.java:11)

提示空指针异常，int.class是由BootstrapClassLoader加载的。

**三、每个类加载器都有一个父加载器**

每个类加载器都有一个父类加载器，比如加载Test.class是由AppClassLoader完成，那么AppClassLoader也有一个父加载器，通过getParent()方法获取。

System.***out***.println("ClassLoader is:" + classLoader.getParent().toString());

运行结果如下：

ClassLoader is:sun.misc.Launcher$ExtClassLoader@4141d797

这个说明，AppClassLoader的父类加载器是ExtClassLoader。那么ExtClassLoader的父类加载器又是谁呢？

System.***out***.println("ClassLoader is:" + classLoader.getParent().getParent().toString());

运行结果又是一个空指针异常。要解释这一点，需要看下面的一个基础前提。

1. **父类加载器不是父类**

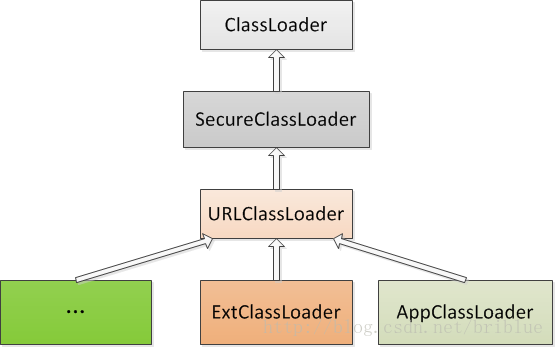
先前粘贴过ExtClassLoader和AppClassLoader的代码。

static class ExtClassLoader extends URLClassLoader {}

static class AppClassLoader extends URLClassLoader {}

由此可见ExtClassLoader和AppClassLoader同样继承自URLClassLoader，但上面一小节代码中，为什么调用AppClassLoader的getParent()代码会得到ExtClassLoader的实例呢？先从URLClassLoader说起。

类关系继承图如下：



URLClassLoader的源码中并没有找到getParent()方法。这个方法在ClassLoader.java中。

答案已经很明了了，**ExtClassLoader的parent为null**。

AppClassLoader的parent是ExtClassLoader，ExtClassLoader的parent是null。这符合我们之前编写的测试代码。

不过，还是有疑问的我们只看到ExtClassLoader和AppClassLoader的创建，那么BootstrapClassLoader呢？

还有，ExtClassLoader的父加载器为null,但是BootstrapClassLoader却可以当成它的父加载器这又是为何呢？

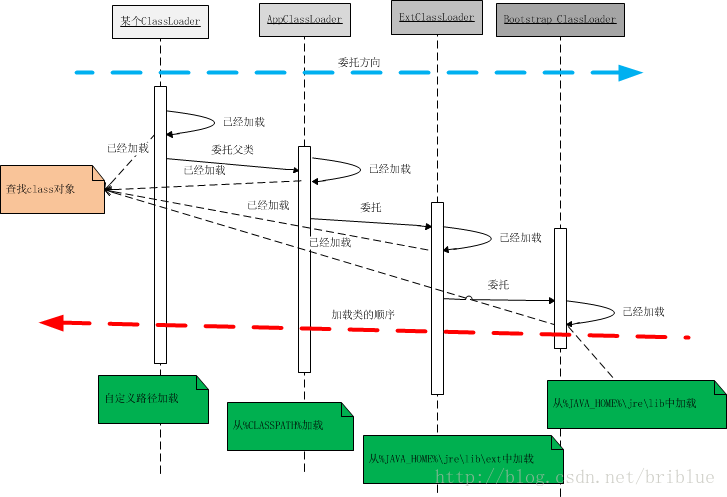
**五、BootstrapClassLoader是由C++编写的**

Bootstrap ClassLoader是由C/C++编写的，它本身是虚拟机的一部分，所以它并不是一个JAVA类，也就是无法在java代码中获取它的引用，J**VM启动时通过Bootstrap类加载器加载rt.jar等核心jar包中的class文件，之前的int.class,String.class都是由它加载**。然后呢，我们前面已经分析了，**JVM初始化sun.misc.Launcher并创建Extension ClassLoader和AppClassLoader实例**。并将ExtClassLoader设置为AppClassLoader的父加载器。Bootstrap没有父加载器，但是它却可以作用一个ClassLoader的父加载器。比如ExtClassLoader。这也可以解释之前通过ExtClassLoader的getParent方法获取为Null的现象。具体是什么原因，很快就知道答案了。

**六、双亲委托**

一个类加载器查找class和resource时，是通过“委托模式”进行的，它**首先判断这个class是不是已经加载成功，如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap ClassLoader，如果BootstrapClassLoader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身去查找这些对象。**这种机制就叫做双亲委托。

整个流程可以如下图所示：



可以看到2根箭头，蓝色的代表类加载器向上委托的方向，**如果当前的类加载器没有查询到这个class对象已经加载就请求父加载器（不一定是父类）进行操作，然后以此类推。直到Bootstrap ClassLoader。如果Bootstrap ClassLoader也没有加载过此class实例，那么它就会从它指定的路径中去查找，如果查找成功则返回，如果没有查找成功则交给子类加载器，也就是ExtClassLoader,这样类似操作直到终点**，也就是我上图中的红色箭头示例。

用序列描述一下：

1）一个AppClassLoader查找资源时，先看看缓存是否有，缓存有从缓存中获取，否则委托给父加载器。

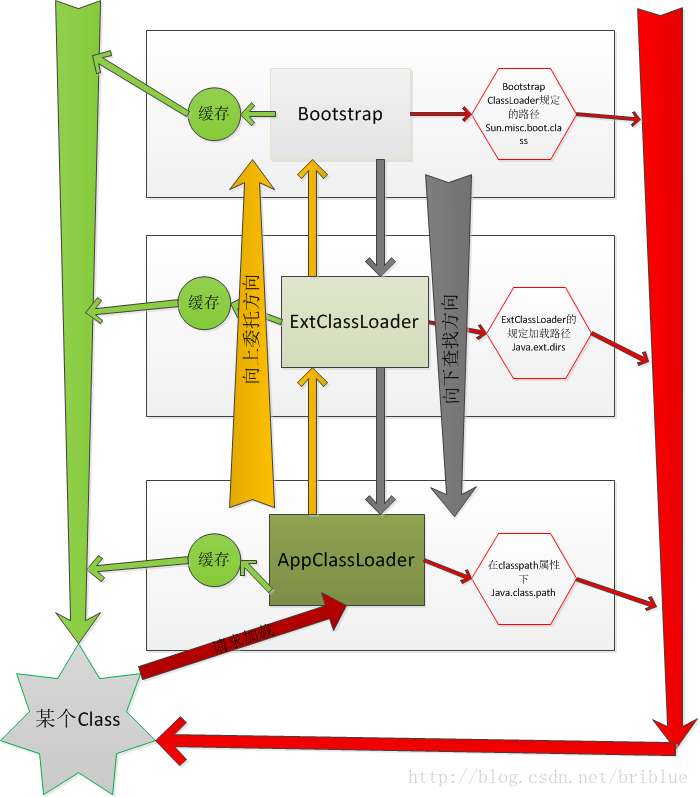
2）递归，重复第1部的操作。

3）如果ExtClassLoader也没有加载过，则由Bootstrap ClassLoader出面，它首先查找缓存，如果没有找到的话，就去找自己的规定的路径下，也就是sun.mic.boot.class下面的路径。找到就返回，没有找到，让子加载器自己去找。

4）Bootstrap ClassLoader如果没有查找成功，则ExtClassLoader自己在java.ext.dirs路径中去查找，查找成功就返回，查找不成功，再向下让子加载器找。

5）ExtClassLoader查找不成功，AppClassLoader就自己查找，在java.class.path路径下查找。找到就返回。如果没有找到就让子类找，如果没有子类会怎么样？抛出各种异常。

上面的序列，详细说明了双亲委托的加载流程。我们可以发现委托是从下向上，然后具体查找过程却是自上至下。



上面已经详细介绍了加载过程，但具体为什么是这样加载，我们还需要了解几个个重要的方法loadClass()、findLoadedClass()、findClass()、defineClass()。

1. **重要方法**
2. loadClass()

protected Class<?> loadClass(String name, boolean resolve)  
 throws ClassNotFoundException

上面是方法原型，一般实现这个方法的步骤是：

1. 执行**findLoadedClass(String)**去检测这个class是不是已经加载过了。
2. 执行父加载器的**loadClass()**方法。如果父加载器为null，则JVM内置的加载器去替代，也就是BootstrapClassLoader。这也解释了ExtClassLoader的parent为null，但仍然说BootstrapClassLoader是它的父加载器。
3. 如果向上委托父加载器没有加载成功，则通过**findClass(String)**查找。

如果class在上面的步骤中找到了，参数resolve又是true的话盘，那么loadClass()又会调用resolveClass(Class)这个方法来生成最终的Class对象。我们可以从源代码看出这个步骤。

protected Class<?> loadClass(String name, boolean resolve)  
 throws ClassNotFoundException  
{  
 synchronized (getClassLoadingLock(name)) {  
 // First, check if the class has already been loaded  
 Class<?> c = findLoadedClass(name);  
 if (c == null) {  
 long t0 = System.nanoTime();  
 try {  
 if (parent != null) {  
 // 父加载器不为空则调用父加载器的loadClass

c = parent.loadClass(name, false);  
 } else {

// 父加载器为空则调用BootstrapClassLoader  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 // ClassNotFoundException thrown if class not found  
 // from the non-null parent class loader  
 }  
  
 if (c == null) {  
 // If still not found, then invoke findClass in order  
 // to find the class.  
 long t1 = System.nanoTime();

// 父类加载器没有找到，则调用findClass  
 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) {

// 调用resolveClass()  
 resolveClass(c);  
 }  
 return c;  
 }  
}

代码解释了双亲委托。另外，要注意的是如果要编写一个classLoader的子类，也就是自定义一个classLoader，建议覆盖**findClass()**方法，而不要直接改写**loadClass()**方法。

ExtClassLoader的parent为null，所以它向上委拖时，系统会为它指定BootstrapClassLoader。

**八、自定义ClassLoader**

不管是BootstrapClassLoader还是ExtClassLoader等，这些类加载器都只是加载指定的目录下的jar包或者资源。如果在某种情况下，需要动态加载一些东西，比如从D盘某个文件夹加载一个class文件，或者从网络上下载class主内容然后再进行加载，这样需要我们自定义一个classLoader。

自定义步骤：

1. 编写一个类继承自ClassLoader抽象类。
2. 复写它的findClass()方法。
3. 在findClass()方法中调用defineClass()（这个方法在在编写自定义classLoader的时候非常重要，它能将class二进制内容转换成Class对象，如果不符合要求会抛出各种异常）。

注：一个classLoader创建时如果没有指定parent，那么它的parent默认就是AppClassLoader。因为这样就能保证它能访问我系统内置加载器加载成功的class文件。

1. **自定义ClassLoader示例之DiskClassLoader**

要求：需要一个自定义的classLoader，默认加载路径为D:\lib下的jar包和资源。

（1）编写一个测试用的类文件，Test.java。

1. 然后将它编译为class文件Test.class放到D:\lib这个路径下。
2. 编写DiskClassLoader的代码

在findClass()方法中定义了查找class的方法，然后数据通过definedClass()生成了Class对象。

（4）编写测试代码

BootstrapClassLoader、ExtClassLoader、AppClassLoader都是加载指定路径下的jar包。如果要突破这种限制，实现自己某些特殊的需求，我们就得自定义ClassLoader，自己指定加载的路径，可是磁盘、内存、网络或者其它。

**九、ContextClassLoader线程上下文类加载器**

ContextClassLoader其实是一个概念。

*/\* The context ClassLoader for this thread \*/*

private ClassLoader contextClassLoader;

public void setContextClassLoader(ClassLoader cl) {  
 SecurityManager sm = System.getSecurityManager();  
 if (sm != null) {  
 sm.checkPermission(new RuntimePermission("setContextClassLoader"));  
 }  
 contextClassLoader = cl;  
}

public ClassLoader getContextClassLoader() {  
 if (contextClassLoader == null)  
 return null;  
 SecurityManager sm = System.getSecurityManager();  
 if (sm != null) {  
 ClassLoader.checkClassLoaderPermission(contextClassLoader,  
 Reflection.getCallerClass());  
 }  
 return contextClassLoader;  
}

contextClassLoader只是一个成员变量，通过setContextClassLoader()方法设置，通过getContextClassLoader()设置。

每个Thread都有一个相关联的ClassLoader，默认是AppClassLoader()。并且子线程默认使用父线程的ClassLoader除非子线程特别设置。

**十、ContextClassLoader的运用时机**

**总结**

1. classLoader用来加载class文件的。
2. 系统内置的classLoader通过双亲委托来加载指定路径下的class和资源。
3. 可以自定义classLoader一般覆盖findClass()方法。
4. ContextClassLoader与线程相关，可以获取和设置，可以绕过双亲委托的机制。