类加载器及其加载原理

作者：vcjmhg
原文链接：https://ld246.com/article/1618630095863
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180629.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="概述">概述</h2>
在之前的文章"类的加载流程"讲了一个 Class 文件从加载到卸载整个生命周期的过程，并且提到
非数组类在加载阶段是可控性最强的"。而这个优点很大程度上都是类加载器所带了的，因而本篇文
就着重讲一下类加载器的加载机制与加载原理。
首先我们思考一个问题：什么是类加载器？
简单来说就是加载类的二进制字节流的工具，那它是如何找到所要加载类
具体位置呢？
答案就是通过类的全限定名。
因而我们可以这样说，类加载器就是用来完成“通过一个类的全限定名来获取描述该
的二进制字节流”这一加载动作的代码。
<h2 id="类加载器的作用">类加载器的作用</h2>
顾名思义，类加载器的作用是实现类的加载动作。但它的作用就仅限于此
？
答案必然是否定的。
我们首先看下边这一段代码：
<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">public c
ass ClassLoaderTest <span class="highlight-o"
{

    <span class="h
ghlight-kd">public static <span class="highlig
t-kt">void main(</
pan>String[] <span c
ass="highlight-n">args) <span class="highlight
kd">throws Exception <span class="highlight-o
>{
        <span class=
highlight-c1">//创建自定义的类加载器并重写loadClass方法
<span cla
s="highlight-c1">   ClassLoader <span class=
highlight-n">myLoader = <span class="highligh
-k">new ClassLoader
) {
            <span clas
="highlight-nd">@Override
            <span clas
="highlight-kd">public Class<span class="highl
ght-o"><?> loadClass<span class="highlig
t-o">(String name</
pan>) throws <spa
 class="highlight-n">ClassNotFoundException {
                <span c
ass="highlight-k">try {
                    <span
class="highlight-n">String fileName <span clas
="highlight-o">= name<span class="highlight-
">.substring(</spa
>name.<span class=
highlight-na">lastIndexOf(<span class="highlig
t-s">".") +
span class="highlight-mi">1) <span class="highl

```
ght-o">+</span> <span class="highlight-s">".class"</span><span class="highlight-o">;</
pan>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-n">InputStream</span> <span class="highlight-n">is</span> <span class
"highlight-o">=</span> <span class="highlight-n">getClass</span><span class="highlight
o">().</span><span class="highlight-na">getResourceAsStream</span><span class="highl
ght-o">(</span><span class="highlight-n">fileName</span><span class="highlight-o">);<
span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-k">if</span> <span class="highlight-o">(</span><span class="highlight-n
>is</span> <span class="highlight-o">==</span> <span class="highlight-kc">null</span
<span class="highlight-o">)</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                    <s
an class="highlight-k">return</span> <span class="highlight-kd">super</span><span clas
="highlight-o">.</span><span class="highlight-na">loadClass</span><span class="highlig
t-o">(</span><span class="highlight-n">name</span><span class="highlight-o">);</span

</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-kt">byte</span><span class="highlight-o">[]</span> <span class="highli
ht-n">b</span> <span class="highlight-o">=</span> <span class="highlight-k">new</sp
n> <span class="highlight-kt">byte</span><span class="highlight-o">[</span><span clas
="highlight-n">is</span><span class="highlight-o">.</span><span class="highlight-na">a
ailable</span><span class="highlight-o">()];</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-n">is</span><span class="highlight-o">.</span><span class="highlight-n
">read</span><span class="highlight-o">(</span><span class="highlight-n">b</span><s
an class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-k">return</span> <span class="highlight-n">defineClass</span><span cla
s="highlight-o">(</span><span class="highlight-n">name</span><span class="highlight-o
>,</span> <span class="highlight-n">b</span><span class="highlight-o">,</span> <span
lass="highlight-mi">0</span><span class="highlight-o">,</span> <span class="highlight-n
>b</span><span class="highlight-o">.</span><span class="highlight-na">length</span>
span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span> <span class="highlight-k">catch</span> <span class="highligh
-o">(</span><span class="highlight-n">IOException</span> <span class="highlight-n">e<
span><span class="highlight-o">)</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-k">throw</span> <span class="highlight-k">new</span> <span class="hi
hlight-n">ClassNotFoundException</span><span class="highlight-o">(</span><span class
"highlight-n">name</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class=
highlight-o">};</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class=
highlight-c1">//使用自定义类加载器加载对象
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>    <span class="highlight-n">Object</span> <span class="high
```

ight-n">obj = myLoder.loadClass("test.ClassLoaderTest").newInstance();

        System.out.println(obj.getClass();
        System.out.println(obj instanceof test.ClassLoaderTest);
    }
}
</code></pre>
代码运行结果如下：
<pre><code class="language-text highlight-chroma">class test.ClassLoaderTest
false
</code></pre>
上述代码基本上所做的事情就是，创建了一个自定义的类加载器，然后使用这个类加载器加载了 <code>ClassLoaderTest</code> 类，并以该类为模板创建了对象。
从输出结果上看，新创建的对象 <code>obj</code> 确实是以 <code>test.ClassLoaderTest</code> 为类模板创建的，但为何在判断是否是 <code>test.ClassLoaderTest</code> 的实例对象时结果是 false 呢？
这是因为在 Java 中一个类的唯一性不仅和类本身相关而且和加载它的类加载器相关，也就是说：任何一个类都必须由加载它的类加载器和这个本身一起确定其在 Java 中的唯一性，每一个类加载器都有一个独立类命名空间。
换句话说，如果想要比较两个类是否相等时，只有这两个类是同一个类加载器的前提下才有意义否则，即使这两个类是同一个 Class 文件，被同一个 Java 虚拟机加载，只要加载它们的类加载器不，这两个类必然就不相等。
这里类的相等与否到底有何影响呢？
这里的相等包含了的 Class 对象的 <code>equals()</code> 方法、<code>isAssignableForm()</code>、<code>isInstance()</code> 方法返回结果，也包括了使用 <code>instanceof</code> 关键字进行从属判断的各种情况。
通过上面的解释，我们应该就懂了为何例子程序中，在使用 <code>instanceof</code> 判断时候返回结果是 false。因为在例子程序中，Java 虚拟中同时存在两个 <code>test.ClassLoaderTest</code> 类，一个是由虚拟机的"应用程序类加载器"加载的，另一个是由自定加载器加载的，但在 Java 虚拟机中仍然是两个独立的类，因而在做类型检查时返回结果是 false。</>
<h2 id="类的加载机制">类的加载机制</h2>
前边我们说了类加载器是参与类唯一性的判断的，并且我们的 Java 虚拟机是有多个类加载器的因而这里就会有一个问题：多个类加载器在加载类的时候是如何进行协调的？它是如何解重复加载这一问题的？

说到这里，我们不得不提一下，类加载器的一个加载机制--<code>双亲委派机制</code>。但正式聊 <code>双亲委派机制</code> 之前，我们有必要了解一下类加载器的类别和它们之间的一层次关系。
<h2 id="类加载器的种类与关系">类加载器的种类与关系</h2>
类加载器的层次关系图，如下所示：

从图中可以看到，系统提供的类加载器主要有三个：

BootstrapClassLoader(启动类加载器) ：用于加载系统类库中类，是最顶层的加载类，由 C++ 实现，负责加载 <code>%JAVA_HOME%/lib</code> 目录下的 jar 包和类或者或被 <code>-Xbootclasspath</code> 参数指定的路径中的所有类。<strng>注意该加载器无法被 Java 程序直接引用，若在自定义加载器中需要委派给启动类加载器加载，直接返回 null 即可。
ExtensionClassLoader(扩展类加载器) ：用于加载系统类库扩类，主要负责加载目录 <code>%JRE_HOME%/lib/ext</code> 目录下的 jar 包和类，被 <code>java.ext.dirs</code> 系统变量所指定的路径下的 jar 包。但在 JDK9 之后，这种扩展机被 <code>模块化</code> 的天然扩展能力所取代。
AppClassLoader(应用程序类加载器) ：面向我们用户的加载器，负责加载前应用 classpath 下的所有 jar 包和类。一般情况下，该加载器是默认加载器。/li>

最后是自定义类加载器(User Class Loader)：这个加载器是需要在继承 <code>ClassLoader</code> 的自定义加载器类 <code>UserClassLoader</code> 中重写 <code>findClass()</code> 来实现的。
此处可能会有些疑问，既然应用程序类加载器都已经是默认的加载器了？那自定义类载器还有什么意义呢？
主要原因是 <code>应用程序加载器</code> 它只能加载在 classpath 下的所有类</trong>，但不在 classpath 下的 class 文件是无法被加载的，比如通过网络远程传输过来的 class 文（远程调用）或者我们在桌面上有一个 class 文件，希望在运行的过程中被加和使用，在这两种情况下，应用程序类加载器是无法加载的，此时如果想要加载必须靠自义类。
为了说明这一点，我们可以看一个例子：
首先我们定义一个待加载的普通类，放置在 <code>com.test</code> 中：
<pre><code class="language-java highlight-chroma">package com.test;

public class Test {
    public void hello() {
   System.out.println</span("我是Test，由 " <spn class="highlight-o">+ getClass().getClassLoader().getClass<span class="highlight-o">(

            + " 加载进来的");
    }
}
</code></pre>
将编译生成后的 class 文件移动到其他位置，非当前项目的 <code>classpath</code> 下面，例位置如下：

注意：
<blockquote>
如果你是直接在当前项目里面创建，待 <code>Test.java</code> 编译后，请把 <code>Test.class</code> 文件拷贝走，再将 <code>Test.java</code> 删除。因为如果 <code>Test.class</code> 存放在当前项目中，根据双亲委派模型可知，会通过 <code>sun.misc.Launcher$AppClassLoader</code> 类加载器加载。为了让我们自定义的类加载器加载，我们把 Test.class 文件放入到其他目。
</blockquote>
如果此时我们想调用 <code>Test</code> 类，因为该类不在项目的 <code>classpath</code> 下，因而无法通过 <code>系统加载器</code> 进行加载，只能通过用户自定义加载器
写一个用户自定义加载器，内容如下：
<pre><code class="language-java highlight-chroma">/**
 * 自定义类加载器，加载自定义位置下的class文件
 * @author vcjmhg
 *
 */
public class UserDefineClassLoader {
    static class MyClassLoader extends ClassLoader {
        private String classPath;

        public MyClassLoader(String classPath) {
            this.classPath = classP</code></pre>

th;
        <span clas
="highlight-o">}

        <span clas
="highlight-kd">private byte<span class="highl
ght-o">[] loadByte(
/span>String name<
pan class="highlight-o">) throws <span class=
highlight-n">Exception {
            <span c
ass="highlight-n">name = <span class="highli
ht-n">name.replac
All("\\."<spa
n class="highlight-o">, "/"<span class="highligh
-o">);
            <span c
ass="highlight-n">FileInputStream fis <span cla
s="highlight-o">= new <span class="highlight-
">FileInputStream(cl
ssPath + "/"
+ name
                <s
an class="highlight-o">+ ".class"<span class="h
ghlight-o">);
            <span c
ass="highlight-kt">int len <span class="highlig
t-o">= fis.<
pan class="highlight-na">available();
            <span c
ass="highlight-kt">byte[] <span class="highligh
-n">data = new</sp
n> byte[<span clas
="highlight-n">len];
            <span c
ass="highlight-n">fis.<span class="highlight-na
>read(data
span class="highlight-o">);
            <span c
ass="highlight-n">fis.<span class="highlight-na
>close();
            <span c
ass="highlight-k">return data<span class="high
ight-o">;

        <span clas
="highlight-o">}

        <span clas
="highlight-kd">protected Class<span class="h
ghlight-o"><?> findClass<span class="high
ight-o">(String nam
) throws <
pan class="highlight-n">ClassNotFoundException {</sp

```
n>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-k">try</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-kt">byte</span><span class="highlight-o">[]</span> <span class="highli
ht-n">data</span> <span class="highlight-o">=</span> <span class="highlight-n">loadBy
e</span><span class="highlight-o">(</span><span class="highlight-n">name</span><sp
n class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-k">return</span> <span class="highlight-n">defineClass</span><span cla
s="highlight-o">(</span><span class="highlight-n">name</span><span class="highlight-o
>,</span> <span class="highlight-n">data</span><span class="highlight-o">,</span> <sp
n class="highlight-mi">0</span><span class="highlight-o">,</span> <span class="highligh
-n">data</span><span class="highlight-o">.</span><span class="highlight-na">length</s
an><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span> <span class="highlight-k">catch</span> <span class="highligh
-o">(</span><span class="highlight-n">Exception</span> <span class="highlight-n">e</s
an><span class="highlight-o">)</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-n">e</span><span class="highlight-o">.</span><span class="highlight-n
">printStackTrace</span><span class="highlight-o">();</span>
</span></span><span class="highlight-line"><span class="highlight-cl">                <span
class="highlight-k">throw</span> <span class="highlight-k">new</span> <span class="hi
hlight-n">ClassNotFoundException</span><span class="highlight-o">();</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class=
highlight-o">};</span>
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class=
highlight-kd">public</span> <span class="highlight-kd">static</span> <span class="highli
ht-kt">void</span> <span class="highlight-nf">main</span><span class="highlight-o">(<
span><span class="highlight-n">String</span> <span class="highlight-n">args</span><s
an class="highlight-o">[])</span> <span class="highlight-kd">throws</span> <span class=
highlight-n">Exception</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
="highlight-n">MyClassLoader</span> <span class="highlight-n">classLoader</span> <sp
n class="highlight-o">=</span> <span class="highlight-k">new</span> <span class="highl
ght-n">MyClassLoader</span><span class="highlight-o">(</span><span class="highlight-s
>"C:\\Users\\vcjmhg\\Desktop\\Test"</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
="highlight-n">Class</span> <span class="highlight-n">clazz</span> <span class="highlig
t-o">=</span> <span class="highlight-n">classLoader</span><span class="highlight-o">.
/span><span class="highlight-na">loadClass</span><span class="highlight-o">(</span><
pan class="highlight-s">"com.test.Test"</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
="highlight-n">Object</span> <span class="highlight-n">obj</span> <span class="highlig
t-o">=</span> <span class="highlight-n">clazz</span><span class="highlight-o">.</span
<span class="highlight-na">newInstance</span><span class="highlight-o">();</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span clas
```

原文链接：类加载器及其加载原理

="highlight-n">Method helloMethod <span cla
s="highlight-o">= clazz<span class="highlight-
">.getDeclaredMethod<span class="highlight-
">("hello", <
pan class="highlight-kc">null);
  <span clas
="highlight-n">helloMethod.<span class="highl
ght-na">invoke(obj
/span>, null<span c
ass="highlight-o">);
  <span class=
highlight-o">}
<span class="high
ight-o">}
</code></pre>
上述代码，基本意思就是：定义了一个类加载器 <code>MyClassLoader</code> 可以对指定
件夹下的 class 文件进行加载，在加载完成之后通过反射创建了一个 <code>obj</code> 对象并调
了其 hello()方法。关于自定义加载器定义方法可以参考后续小节的"如何自定义类加载器?"
运行结果如下：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">我是Test，由 class com.test.UserDefineClassLoader$MyClassLoader 加载进
的
</code></pre>
上边的例子，就解释了自定义类加载器的作用：它可以按照用户要求加载指定的 <code>class
件</code>，无论 <code>class文件</code> 是通过网络传过来，还是本地的某个路径，都可以实
加载。
<h2 id="双亲委派机制">双亲委派机制</h2>
其实从前边那个类的层次关系图中，，我们就可以简单了解双亲委派模型加载机制：
当一个类加载器收到类加载请求时，首先不会自己尝试加载这个类，而是把这个请求
派给父类加载器去完成，每一层的类加载器都是如此，因此所有的
载请求最终都会传送到顶层的启动类加载器中，只有父加载器无法完成这个加
请求（它的搜索范围内，找不到所需的类）时，子加载器才会尝试自己去完成加载。
结合自定义加载器，整个类的加载流程如下图所示：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210416113915-4qhgv50.png?imageV
ew2/2/interlace/1/format/jpg">

当我们的 <code>自定义类加载器</code> 要加载一个类的时候，会首先判断给定的类是否被
载过，如果已经被加载过则不再加载，可以直接使用；如果没被加载，它也不会直接加载</
trong>而是把加载任务委派给父加载器也就是 <code>AppClassLoader</code> 加载器。
<code>AppClassLoader</code> 在收到委派的加载任务后，也不直接加载，也会做一个<strong
g>自己是否加载过的判断，，如果没有将将加载任务委派给 <code>ExtClassLoader</c
de>。
<code>ExtClassLoader</code> 收到委派的任务后，在自己没有加载过该类的情况下，会将
载任务委派给 <code>BootstrapClassLoader</code>，由于 <code>BootstrapClassLoader</co
e> 是顶层加载器没有父加载器，因而 <code>BootstrapClassLoader</code> 会开始尝试自己加
，如果说需要加载的类位于其加载范围（比如 <code>-Xbootclasspath</code> 参数指定的加载
），则直接返回加载结果。否则下沉到子类加载器进行加载，直到底层的 <code>自定义类加载器</
ode>
要注意，如果所有的类加载器最终都无法加载，会抛出一个 <code>ClassNotFoundException<
code>

到这里可能就会有小伙伴有疑问了：这玩意有什么用？为什么要这样设计呢？
<h3 id="双亲委派机制有什么用-">双亲委派机制有什么用？</h3>

这种加载机制一个显而易见的好处就是 Java 中的类随着它的它的类加载器起具备了具有优先级的层级结构。比如类 <code>java.lang.Object</code> 它存放在 <code>rt.jar</code> 需要被顶层启动类加载器所加载，因而 <cod>Object</code> 类在程序的各种类加载器的环境中都能保证是同一个类，避了重复加载的情况发生，而这也避免了危险代码植入的风险（比如恶意替换 <code>java.lang.Objec</code> 类）。
双亲委派模型对于 Java 程序的稳定性极其重要，但其实现却异常简单。
<h3 id="双亲委派机制如何实现的-">双亲委派机制如何实现的？</h3>
用以实现双亲委派的代码只有短短十多行，全部集中在 <code>java.lang.ClassLoader</code> 的 <code>loadClass()</code> 方法之中，具体实现代码如下：
<pre><code class="language-java highlight-chroma"><span c ass="highlight-cl">//name：被加载类的全限定名
<span cla s="highlight-c1">// resolove:是否连接了已加载的类
<span cla s="highlight-c1">protected Class<?> <span class="highlight-n >loadClass(String</ pan> name, <span c ass="highlight-kt">boolean resolve<span class "highlight-o">) throws ClassNotFoundException
{
  synchronized (getClassLoadingLock(name)) {</sp n>
  <span class= highlight-c1">// 首先检查类是否已经被加载了
<span cla s="highlight-c1"> Class<?> c = /span> findLoadedClass(</s an>name);
  //未被加载的情况下，尝试用父类加载器进行加载
<span cla s="highlight-c1"> if (c == < pan class="highlight-kc">null) {
  try {
  <span class= highlight-c1">//存在父类加载器的情况下继续向上委托
<span cla s="highlight-c1"> if (parent !=</ pan> null) <span cl ss="highlight-o">{
  c = <span class="highlight

```
n">parent</span><span class="highlight-o">.</span><span class="highlight-na">loadClas
</span><span class="highlight-o">(</span><span class="highlight-n">name</span><span
class="highlight-o">,</span> <span class="highlight-kc">false</span><span class="highlig
t-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span> <span class="highlight-k">else</span> <span class="highlight
o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
highlight-c1">//使用顶层加载器BootstrapClassLoader进行加载
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>                <span class="highlight-n">c</span> <span class="hi
hlight-o">=</span> <span class="highlight-n">findBootstrapClassOrNull</span><span clas
="highlight-o">(</span><span class="highlight-n">name</span><span class="highlight-o
>);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span clas
="highlight-o">}</span> <span class="highlight-k">catch</span> <span class="highlight-o
>(</span><span class="highlight-n">ClassNotFoundException</span> <span class="highli
ht-n">e</span><span class="highlight-o">)</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-c1">// 父类加载器无法加载的情况下抛出ClassNotFoundException异常
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>                <span class="highlight-c1">// 说明父类加载器无法完成
载请求
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>            <span class="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">            <span clas
="highlight-k">if</span> <span class="highlight-o">(</span><span class="highlight-n">c
/span> <span class="highlight-o">==</span> <span class="highlight-kc">null</span><sp
n class="highlight-o">)</span> <span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span c
ass="highlight-c1">// 父类加载器无法加载的情况下，调用本加载器的findClass()方法着手进行加载
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>                <span class="highlight-n">c</span> <span class="highl
ght-o">=</span> <span class="highlight-n">findClass</span><span class="highlight-o">(
/span><span class="highlight-n">name</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span clas
="highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-c1">//如果resolve标记为true，则在加载操作完成后执行链接操作
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>        <span class="highlight-k">if</span> <span class="highlight
o">(</span><span class="highlight-n">resolve</span><span class="highlight-o">)</span>
<span class="highlight-o">{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">            <span clas
="highlight-n">resolveClass</span><span class="highlight-o">(</span><span class="highl
ght-n">c</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
highlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
```

highlight-k">return c

    <span class="h
ghlight-o">}
<span class="high
ight-o">}
</code></pre>
这段代码逻辑非常简单：先检查类是否被加载，如果没有被加载则委托父类加载器进行加载，若类加载器也无法加载则调用 <code>findClass()</code> 方法进行进行加载。
<h2 id="如何自定义类加载器-">如何自定义类加载器?</h2>
实现一个自定义类加载器有两种情况：
<h3 id="第一种-不破坏-双亲委派机制-">第一种 不破坏“双亲委派机制”</h3>
从 <code>loadClass()</code> 的代码中可以看到，类加载的最后一步就是调用 <code>findC
ass()</code> 方法，因而如果要实现一个自定义类加载器需要首先继承 <code>ClassLoader</cod
> 类，然后重写其 <code>findClass()</code> 方法。
我们可以首先看下 <code>findClass()</code> 的默认实现：
<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">protected <span class="highlight-n
>Class<?> find
lass(String
span class="highlight-n">name) <span class="h
ghlight-kd">throws ClassNotFoundException <
pan class="highlight-o">{
        <span class=
highlight-k">throw new <span class="highlight
n">ClassNotFoundException(<span class="highl
ght-n">name);
<span class="high
ight-o">}
</code></pre>
可以看出，抽象类 <code>ClassLoader</code> 的 <code>findClass()</code> 函数默认只抛出异常的，因此要自定义类加载器必须要重写 <code>findClass()</code> 方法，根据传入的字串（指定类文件路径）生成对应的 Class 对象。
那如何生成一个 Class 对象呢？
很简单，<code>Java</code> 提供了 <code>defineClass()</code> 方法，通过这个方法，们可以把一个字节数组转为 Class 对象。
<code>defineClass()</code> 方法的默认实现如下：
<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">protected <span class="highlight-kd
>final Class<?&gt
 defineClass(</span
String name<span cl
ss="highlight-o">, byte<span class="highlight-
">[] b, <spa
class="highlight-kt">int off<span class="highli
ht-o">, int len</spa
>)
        <span class=
highlight-kd">throws ClassFormatError <span c
ass="highlight-o">{
        <span class=
highlight-k">return defineClass<span class="hi
hlight-o">(name,</
pan> b, <span class

"highlight-n">off, le
, null<span
class="highlight-o">);
<span class="high
ight-o">}
</code></pre>
当重写 <code>findClass()</code> 首先要获取到 Class 文件的字节流数据，然后将字节流数
传递给 <code>defineClass()</code> 方法最终获取到一个 Class 对象，至此
不破坏双亲委派机制的情况下，就完成了自定义类加载器。具体实现可以参考前边的"例子"，该自定
类加载器的基本思路就是重写了 <code>findClass()</code> 方法。
<h3 id="第二种--破坏双亲委托机制-">第二种 “破坏双亲委托机制” </h3>
可能在某些场景下，需要使用自定义加载器加载一些特殊的类文件，比如位于 classpath 路径下
一些类文件，如果直接重写 <code>findClass()</code> 方法，由于 <code>双亲委派机制</code>
该类必然会被 <code>AppClassLoader</code> 所加载。因而若要实现这样的类加载器，必须要重
<code>loadClass()</code> 方法。
<h2 id="如何破坏双亲委托-">如何破坏双亲委托？</h2>
因为 <code>双亲委派机制</code> 并不是一个强制性约束的模型，而是 Java 设计者推荐给
发者们的类加载器实现方式，因而在"模块化"出现之前，双亲委托模型主要出现了 3 次被较大规模 “
破坏” 的情况：
第一次：为了保证兼容性：
由于 <code>双亲委派模型</code> 是在 JDK1.2 之后才被引入的，而类加载器这一概念 <cod
>java.lang.ClasssLoader</code> 这一概念在 Java 第一个版本中便存在了。因而为了保证向前兼
性，兼用 JDK1.2 之前已经存在的自定义类加载器代码，Java 设计者在引入双
委派模型的时候做了一些妥协，不直接以技术手段避免 <code>loadClass()</code> 被覆盖的可能
，而是将 <code>双亲委派模型</code> 的逻辑代码写在 <code>loadClass()</code> 方法中。
且引导用户在编写自定义类加载器时，尽量重写新添加的 <code>findClass()</code> 方法，而不
覆盖 <code>loadClass()</code> 方法。
第二次：为了实现 SPI 技术
某些情况下，基础类型需要调用用户的代码，比如 JNDI 技术（对资源几种查找和管理的技术）
它需要调用其他厂商实现并部署在应用程序的 ClassPath 下的 JNDI 服务提供者接口 SPI 的代码，但
启动类绝不可能认识和加载这些代码，那该怎么办呢？
为了解决该问题，Java 设计团队设计了一个线程上下文加载器（<code>Thread Context Class
oader</code>）。这个类加载器可以通过 <code>java.lang.Thread</code> 类的 <code>setCon
extLoader()</code> 方法进行设计，如果创建线程时，没有设置它将从父线程中继承一个，如果在
用程序的全局范围内都没有设置过的话，那么这个类加载器默认就是应用程序类加载器。
有了这些上下文类加载器之后，JNDI 服务使用这个线程上下文加载器去加载所需要的 SPI 代码
这是一种父加载器请求自类加载器的类加载行为。
第三次：用户对应用程序动态性的追求所导致的
为了追求应用程序的动态性，IBM 在 2008 年提出了 OSGI 技术，用来实现模块化的热部署。</
>
其实现热部署的原理如下：
OSGI 实现模块化热部署的关键是它自定义的类加载器机制的实现，每一个程序模块（Bundle）
有一个自己的类加载器，当需要替换一个 Bundle 时，就把 Bundle 连同类加载器一起替换掉以实现
码的热替换。在 OSGI 环境下，类加载器不再是双亲委派模型推荐的树状结构
而逐步发展成了网状结构，当收到请求加载时，OSGI 将按照如下顺序进行类搜索：

将以 <code>java.*</code> 开头的类，委派给父类加载器加载
否则，将委派列表名单的类，委派给父加载器进行加载
否则，将 Import 列表中的类委派给 Export 这个类的 Bundle 的类加载器进行加载
否则，查找当前 Bundle 的 ClassPath，使用自己的类加载器进行加载
否则，查找类是否在自己的 Fragment Bundle 中，如果在，则委派给 Fragment Bundle 的类
载器进行加载
否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器加载
否则，类查找失败

从上边流程中我们可以看到，只有前两条符合双亲委派模型，其他的均不符合。
<h2 id="总结">总结</h2>
本文主要讲了常用的类加载器，比如启动类加载器、扩展类加载器、应用类加载器以及自定义类载器，详细介绍了类加载器在加载一个类时的原理以及加载所使用的双亲委派机制。以及使用双亲委机制的好处以及破坏该机制的一些情况。
<h2 id="引用">引用</h2>

Java 自定义类加载器与双亲委模型
深入理解 jvm 虚拟机 第三版
类加载器
