1、类加载过程

多个java文件经过编译打包生成可运行jar包，最终由java命令运行某个主类的main函数启动程序，这里首先需要通过**类加载器**把主类加载到JVM。

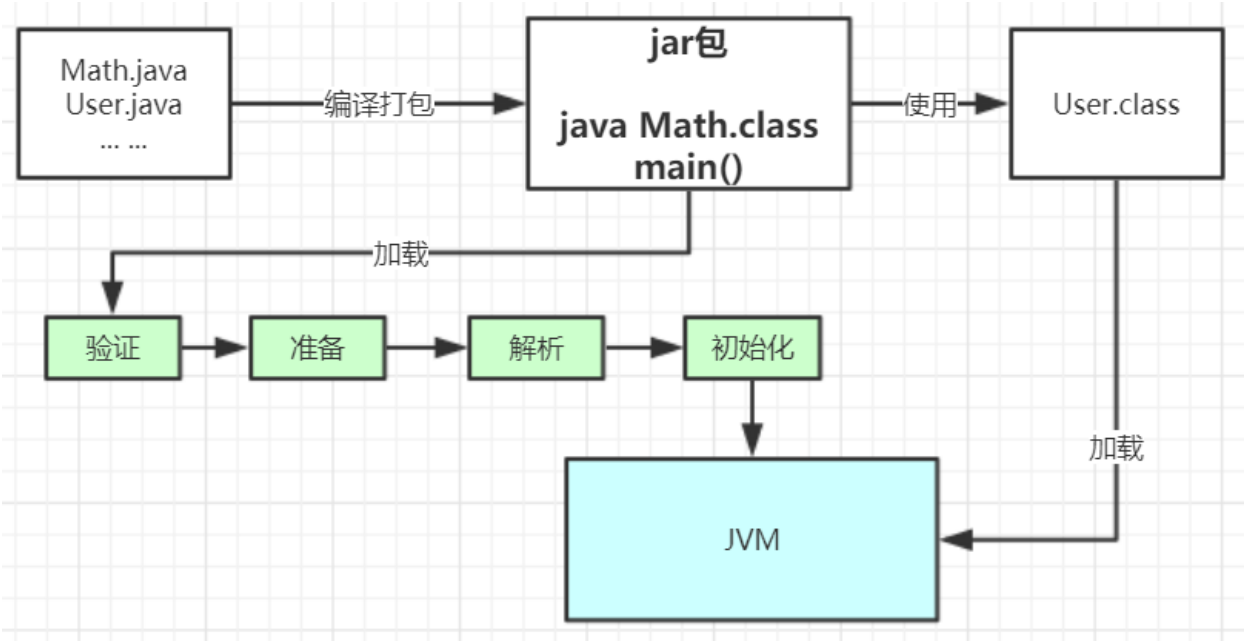
主类在运行过程中如果使用到其它类，会逐步加载这些类。

注意，jar包里的类不是一次性全部加载的，是使用到时才加载。

类加载到使用整个过程有如下几步：

加载 >> 验证 >> 准备 >> 解析 >> 初始化 >> 使用 >> 卸载

- 加载：在硬盘上查找并通过IO读入字节码文件，使用到类时才会加载，例如调用类的main()方法，new对象等等
- 验证：校验字节码文件的正确性
- 准备：给类的静态变量分配内存，并赋予默认值
- 解析：将符号引用替换为直接引用，该阶段会把一些静态方法(符号引用，比如main()方法)替换为指向数据所存内存的指针或句柄等(直接引用)，这是所谓的**静态链接**过程(类加载期间完成)，**动态链接**是在程序运行期间完成的将符号引用替换为直接引用，下节课会讲到动态链接
- **初始化**：对类的静态变量初始化为指定的值，执行静态代码块



2、类加载器和双亲委派机制

上面的类加载过程主要是通过类加载器来实现的，Java里有如下几种类加载器

- 启动类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
- 应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
- 自定义加载器：负责加载用户自定义路径下的类包

看一个类加载器示例：

```
1  public class TestJDKClassLoader {
2   public static void main(String[] args){
3   System.out.println(String.class.getClassLoader());
4   System.out.println(com.sun.crypto.provider.DESKeyFactory.class.getClassL
oader().getClass().getName());
5
System.out.println(TestJDKClassLoader.class.getClassLoader().getClass().get
Name());
6
System.out.println(ClassLoader.getSystemClassLoader().getClass().getName())
7   }
8  }
9
10  运行结果：
11  null //启动类加载器是C++语言实现，所以打印不出来
12  sun.misc.Launcher$ExtClassLoader
13  sun.misc.Launcher$AppClassLoader
14  sun.misc.Launcher$AppClassLoader
```

自定义一个类加载器示例：

自定义类加载器只需要继承 java.lang.ClassLoader 类，该类有两个核心方法，一个是loadClass(String, boolean)，实现了**双亲委派机制**，大体逻辑

1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。

2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）.或者是调用bootstrap类加载器来加载。

3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载。

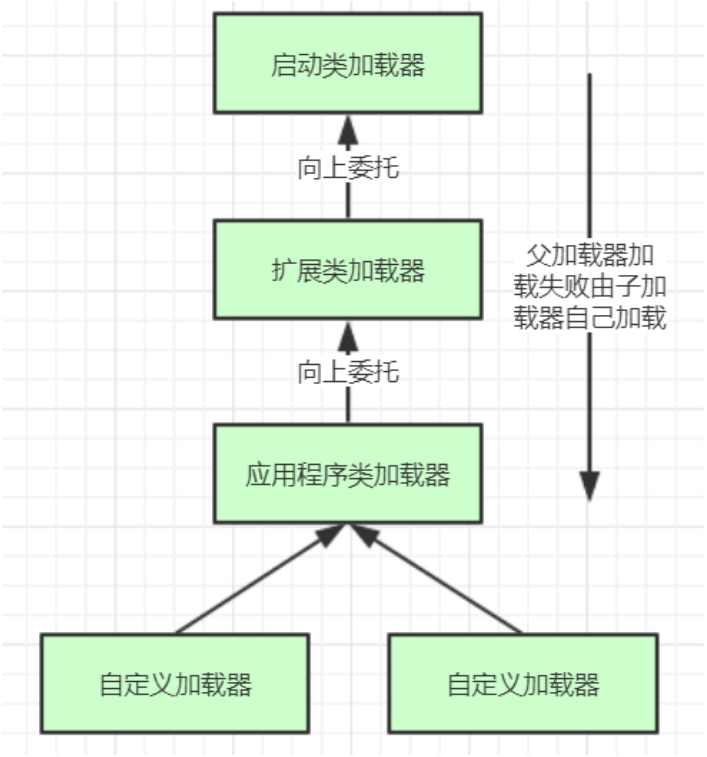
还有一个方法是findClass，默认实现是抛出异常，所以我们自定义类加载器主要是**重写findClass方法。**

```
public class MyClassLoaderTest {
 static class MyClassLoader extends ClassLoader {
 private String classPath;

 public MyClassLoader(String classPath) {
 this.classPath = classPath;
 }

 private byte[] loadByte(String name) throws Exception {
 name = name.replaceAll("\\.", "/");
 FileInputStream fis = new FileInputStream(classPath + "/" + name
 + ".class");
 int len = fis.available();
 byte[] data = new byte[len];
 fis.read(data);
 fis.close();
 return data;
 }

 protected Class<?> findClass(String name) throws ClassNotFoundException
{
 try {
 byte[] data = loadByte(name);
 //defineClass将一个字节数组转为Class对象，这个字节数组是class文件读取后最终
的字节数组。
 return defineClass(name, data, 0, data.length);
 } catch (Exception e) {
 e.printStackTrace();
 throw new ClassNotFoundException();
 }
 }

 }

 public static void main(String args[]) throws Exception {
 MyClassLoader classLoader = new MyClassLoader("D:/test");
 Class clazz = classLoader.loadClass("com.tuling.jvm.User1");
 Object obj = clazz.newInstance();
 Method method= clazz.getDeclaredMethod("sout", null);
```

```
38  method.invoke(obj, null);
39  System.out.println(clazz.getClassLoader().getClass().getName());
40  }
41 }
42
43 运行结果：
44 =======自己的加载器加载类调用方法=======
45 com.tuling.jvm.MyClassLoaderTest$MyClassLoader
```

双亲委派机制

JVM类加载器是有亲子层级结构的，如下图



这里类加载其实就有一个**双亲委派机制**，加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。

比如我们的Math类，最先会找应用程序类加载器加载，应用程序类加载器会先委托扩展类加载器加载，扩展类加载器再委托启动类加载器，顶层启动类加载器在自己的类加载路径里找了半天没找到Math类，则向下退回加载Math类的请求，扩展类加载器收到回复就自己加载，在自己的类加载路径里找了半天也没找到Math类，又向下退回Math类的加载请求给应用程序类加载器，应用程序类加载器于是在自己的类加载路径里找Math类，结果找到了就自己加载了。。

双亲委派机制说简单点就是，先找父亲加载，不行再由儿子自己加载

为什么要设计双亲委派机制?

- 沙箱安全机制：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改
- 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，保证**被加载类的唯一性**

看一个类加载示例：

```
package java.lang;

public class String {
 public static void main(String[] args) {
 System.out.println("**************My String Class**************");
 }
}

运行结果：
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
 public static void main(String[] args)
否则 JavaFX 应用程序类必须扩展javafx.application.Application
```

再来一个沙箱安全机制示例，尝试打破双亲委派机制，用自定义类加载器加载我们自己实现的 java.lang.String.class

```
public class MyClassLoaderTest {
 static class MyClassLoader extends ClassLoader {
 private String classPath;

 public MyClassLoader(String classPath) {
 this.classPath = classPath;
 }

 private byte[] loadByte(String name) throws Exception {
 name = name.replaceAll("\\.", "/");
 FileInputStream fis = new FileInputStream(classPath + "/" + name
 + ".class");
 int len = fis.available();
 byte[] data = new byte[len];
 fis.read(data);
 fis.close();
 return data;
```

```
18
19  }
20
21  protected Class<?> findClass(String name) throws ClassNotFoundException
{
22  try {
23  byte[] data = loadByte(name);
24  return defineClass(name, data, 0, data.length);
25  } catch (Exception e) {
26  e.printStackTrace();
27  throw new ClassNotFoundException();
28  }
29  }
30
31  /**
32  * 重写类加载方法，实现自己的加载逻辑，不委派给双亲加载
33  * @param name
34  * @param resolve
35  * @return
36  * @throws ClassNotFoundException
37  */
38  protected Class<?> loadClass(String name, boolean resolve)
39  throws ClassNotFoundException {
40  synchronized (getClassLoadingLock(name)) {
41  // First, check if the class has already been loaded
42  Class<?> c = findLoadedClass(name);
43
44  if (c == null) {
45  // If still not found, then invoke findClass in order
46  // to find the class.
47  long t1 = System.nanoTime();
48  c = findClass(name);
49
50  // this is the defining class loader; record the stats
51  sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
52  sun.misc.PerfCounter.getFindClasses().increment();
53  }
54  if (resolve) {
55  resolveClass(c);
56  }
57  return c;
```

```
58  }
59  }
60  }
61
62  public static void main(String args[]) throws Exception {
63  MyClassLoader classLoader = new MyClassLoader("D:/test");
64  //尝试用自己改写类加载机制去加载自己写的java.lang.String.class
65  Class clazz = classLoader.loadClass("java.lang.String");
66  Object obj = clazz.newInstance();
67  Method method= clazz.getDeclaredMethod("sout", null);
68  method.invoke(obj, null);
69  System.out.println(clazz.getClassLoader().getClass().getName());
70  }
71 }
72
73 运行结果：
74 java.lang.SecurityException: Prohibited package name: java.lang
75  at java.lang.ClassLoader.preDefineClass(ClassLoader.java:659)
76  at java.lang.ClassLoader.defineClass(ClassLoader.java:758)
```

打破双亲委派

以Tomcat类加载为例，Tomcat 如果使用默认的双亲委派类加载机制行不行?

我们思考一下：Tomcat是个web容器， 那么它要解决什么问题：

1. 一个web容器可能需要部署两个应用程序，不同的应用程序可能会**依赖同一个第三方类库的不同版本**，不能要求同一个类库在同一个服务器只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。

2. 部署在同一个web容器中**相同的类库相同的版本可以共享。**否则，如果服务器有10个应用程序，那么要有10份相同的类库加载进虚拟机。

3. **web容器也有自己依赖的类库，不能与应用程序的类库混淆。**基于安全考虑，应该让容器的类库和程序的类库隔离开来。

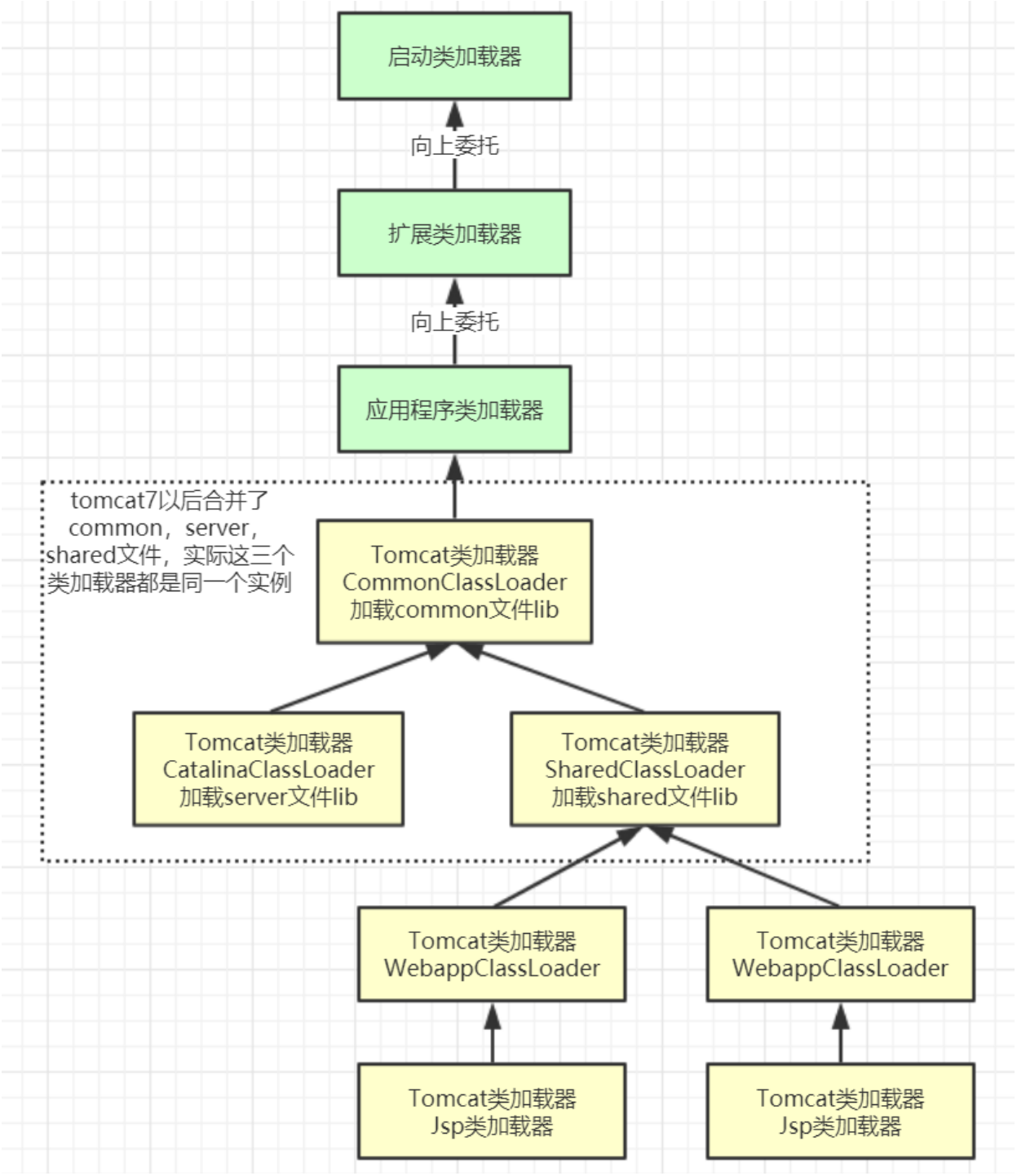
4. web容器要支持jsp的修改，我们知道，jsp 文件最终也是要编译成class文件才能在虚拟机中运行，但程序运行后修改jsp已经是司空见惯的事情， web容器需要支持 jsp 修改后不用重启。

再看看我们的问题：Tomcat 如果使用默认的双亲委派类加载机制行不行?

答案是不行的。为什么?

第一个问题，如果使用默认的类加载器机制，那么是无法加载两个相同类库的不同版本的，默认的类加器是不管你是什么版本的，只在乎你的全限定类名，并且只有一份。第二个问题，默认的类加载器是能够实现的，因为他的职责就是保证**唯一性。**
第三个问题和第一个问题一样。
我们再看第四个问题，我们想我们要怎么实现jsp文件的热加载，jsp 文件其实也就是class文件，那么如果修改了，但类名还是一样，类加载器会直接取方法区中已经存在的，修改后的jsp是不会重新加载的。那么怎么办呢？我们可以直接卸载掉这jsp文件的类加载器，所以你应该想到了，每个jsp文件对应一个唯一的类加载器，当一个jsp文件修改了，就直接卸载这个jsp类加载器。重新创建类加载器，重新加载jsp文件。

Tomcat自定义加载器详解



tomcat的几个主要类加载器：

- commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
- catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
- sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
- WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见；

从图中的委派关系中可以看出：
CommonClassLoader能加载的类都可以被CatalinaClassLoader和SharedClassLoader使用，从而实现了公有类库的共用，而CatalinaClassLoader和SharedClassLoader自己能加载的类则与对方相互隔离。
WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离。
而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的热加载功能。

tomcat 这种类加载机制违背了java 推荐的双亲委派模型了吗？答案是：违背了。
我们前面说过，双亲委派机制要求除了顶层的启动类加载器之外，其余的类加载器都应当由自己的父类加载器加载。
很显然，tomcat 不是这样实现，tomcat 为了实现隔离性，没有遵守这个约定，**每个webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器，打破了双亲委派机制。**