重新看待Jar包冲突问题及解决方案

ar包冲突是老生常谈的问题，几乎每一个Java程序猿都不可避免地遇到过，并且也都能想到通常的原因一般是同一个Jar包由于maven传递依赖等原因被引进了多个不同的版本而导致，可采用依赖排除、依赖管理等常规方式来尝试解决该问题，但这些方式真正能彻底解决该冲突问题吗？答案是否定的。笔者之所以将文章题目起为“重新看待”，是因为之前对于Jar包冲突问题的理解仅仅停留在前面所说的那些，直到在工作中遇到的一系列Jar包冲突问题后，才发现并不是那么简单，对该问题有了重新的认识，接下来本文将围绕Jar包冲突的问题本质和相关的解决方案这两个点进行阐述。

Jar包冲突问题

一、冲突的本质

Jar包冲突的本质是什么？Google了半天也没找到一个让人满意的完整定义。其实，我们可以从Jar包冲突产生的结果来总结，在这里给出如下定义（此处如有不妥，欢迎拍砖~-~）：

> Java应用程序因某种因素，加载不到正确的类而导致其行为跟预期不一致。

具体来说可分为两种情况：1）应用程序依赖的同一个Jar包出现了多个不同版本，并选择了错误的版本而导致JVM加载不到需要的类或加载了错误版本的类，为了叙述的方便，笔者称之为**第一类Jar包冲突问题**；2）同样的类（类的全限定名完全一样）出现在多个不同的依赖Jar包中，即该类有多个版本，并由于Jar包加载的先后顺序导致JVM加载了错误版本的类，称之为**第二类Jar包问题**。这两种情况所导致的结果其实是一样的，都会使应用程序加载不到正确的类，那其行为自然会跟预期不一致了，以下对这两种类型进行详细分析。

1.1 同一个Jar包出现了多个不同版本

随着Jar包迭代升级，我们所依赖的开源的或公司内部的Jar包工具都会存在若干不同的版本，而版本升级自然就避免不了类的方法签名变更，甚至于类名的更替，而我们当前的应用程序往往依赖特定版本的某个类 M ，由于maven的传递依赖而导致同一个Jar包出现了多个版本，当maven的仲裁机制选择了错误的版本时，而恰好类 M在该版本中被去掉了，或者方法签名改了，导致应用程序因找不到所需的类 M或找不到类 M中的特定方法，就会出现第一类Jar冲突问题。可总结出该类冲突问题发生的以下三个必要条件：

- 由于maven的传递依赖导致依赖树中出现了同一个Jar包的多个版本
- 该Jar包的多个版本之间存在接口差异，如类名更替，方法签名更替等，且应用程序依赖了其中有变更的类或方法
- maven的仲裁机制选择了错误的版本

1.2 同一个类出现在多个不同Jar包中

同样的类出现在了应用程序所依赖的两个及以上的不同Jar包中，这会导致什么问题呢？我们知道，同一个类加载器对于同一个类只会加载一次（多个不同类加载器就另说了，这也是解决Jar包冲突的一个思路，后面会谈到），那么当一个类出现在了多个Jar包中，假设有 A 、 B 、 C 等，由于Jar包依赖的路径长短、声明的先后顺序或文件系统的文件加载顺序等原因，类加载器首先从Jar包 A 中加载了该类后，就不会加载其余Jar包中的这个类了，那么问题来了：如果应用程序此时需要的是Jar包 B 中的类版本，并且该类在Jar包 A 和 B 中有差异（方法不同、成员不同等等），而JVM却加载了Jar包 A 的中的类版本，与期望不一致，自然就会出现各种诡异的问题。

从上面的描述中，可以发现出现不同Jar包的冲突问题有以下三个必要条件：

- 同一个类 **M** 出现在了多个依赖的Jar包中，为了叙述方便，假设还是两个： **A** 和 **B**
- Jar包 **A** 和 **B** 中的该类 **M** 有差异，无论是方法签名不同也好，成员变量不同也好，只要可以造成实际加载的类的行为和期望不一致都行。如果说Jar包 **A** 和 **B** 中的该类完全一样，那么类加载器无论先加载哪个Jar包，得到的都是同样版本的类 **M** ，不会有任何影响，也就不会出现Jar包冲突带来的诡异问题。
- 加载的类 **M** 不是所期望的版本，即加载了错误的Jar包

二、冲突的产生原因

2.1 maven仲裁机制

当前maven大行其道，说到第一类Jar包冲突问题的产生原因，就不得不提maven的依赖机制了。传递性依赖是Maven2.0引入的新特性，让我们只需关注直接依赖的Jar包，对于间接依赖的Jar包，Maven会通过解析从远程仓库获取的依赖包的pom文件来隐式地将其引入，这为我们开发带来了极大的便利，但与此同时，也带来了常见的问题——版本冲突，即同一个Jar包出现了多个不同的版本，针对该问题Maven也有一套仲裁机制来决定最终选用哪个版本，但Maven的

选择往往不一定是我们所期望的，这也是产生Jar包冲突最常见的原因之一。先来看下Maven的仲裁机制：

- 优先按照依赖管理<dependencyManagement>元素中指定的版本声明进行仲裁，此时下面的两个原则都无效了
- 若无版本声明，则按照“短路径优先”的原则（Maven2.0）进行仲裁，即选择依赖树中路径最短的版本
- 若路径长度一致，则按照“第一声明优先”的原则进行仲裁，即选择POM中最先声明的版本

从maven的仲裁机制中可以发现，除了第一条仲裁规则（这也是解决Jar包冲突的常用手段之一）外，后面的两条原则，对于同一个Jar包不同版本的选择，maven的选择有点“一厢情愿”了，也许这是maven研发团队在总结了大量的项目依赖管理经验后得出的两条结论，又或者是发现根本找不到一种统一的方式来满足所有场景之后的无奈之举，可能这对于多数场景是适用的，但是它不一定适合我——当前的应用，因为每个应用都有其特殊性，该依赖哪个版本，maven没办法帮你完全搞定，如果你没有规规矩矩地使用<dependencyManagement>来进行依赖管理，就注定了逃脱不了第一类Jar包冲突问题。

2.1 Jar包的加载顺序

对于第二类Jar包冲突问题，即多个不同的Jar包有类冲突，这相对于第一类问题就显得更为棘手。为什么这么说呢？在这种情况下，两个不同的Jar包，假设为 A、 B，它们的名称互不相同，甚至可能完全不沾边，如果不是出现冲突问题，你可能都不会发现它们有共有的类！对于A、B这两个Jar包，maven就显得无能为力了，因为maven只会为你针对同一个Jar包的不同版本进行仲裁，而这俩是属于不同的Jar包，超出了maven的依赖管理范畴。此时，当A、B都出现在应用程序的类路径下时，就会存在潜在的冲突风险，即A、B的加载先后顺序就决定着JVM最终选择的类版本，如果选错了，就会出现诡异的第二类冲突问题。

那么Jar包的加载顺序都由哪些因素决定的呢？具体如下：

- Jar包所处的加载路径，或者换个说法就是加载该Jar包的类加载器在JVM类加载器树结构中所处层级。由于JVM类加载的双亲委派机制，层级越高的类加载器越先加载其加载路径下的类，顾名思义，引导类加载器（bootstrap ClassLoader，也叫启动类加载器）是最先加载其路径下Jar包的，其次是扩展类加载器（extension ClassLoader），再次是系统类加载器（system ClassLoader，也就是应用加载器appClassLoader），Jar包所处加载路径的不同，就决定了它的加载顺序的不同。比

如我们在eclipse中配置web应用的resin环境时，对于依赖的Jar包是添加到Bootstrap Entries中还是User Entries中呢，则需要仔细斟酌下咯。

- 文件系统的文件加载顺序。这个因素很容易被忽略，而往往又是因环境不一致而导致各种诡异冲突问题的罪魁祸首。因tomcat、resin等容器的ClassLoader获取加载路径下的文件列表时是不排序的，这就依赖于底层文件系统返回的顺序，那么当不同环境之间的文件系统不一致时，就会出现有的环境没问题，有的环境出现冲突。例如，对于Linux操作系统，返回顺序则是由iNode的顺序来决定的，如果说测试环境的Linux系统与线上环境不一致时，就极有可能出现典型案例：测试环境怎么测都没问题，但一上线就出现冲突问题，规避这种问题的最佳办法就是尽量保证测试环境与线上一致。

三、冲突的表象

Jar包冲突可能会导致哪些问题？通常发生在编译或运行时，主要分为两类问题：一类是比较直观的也是最为常见的错误是抛出各种运行时异常，还有一类就是比较隐晦的问题，它不会报错，其表现形式是应用程序的行为跟预期不一致，分条罗列如下：

- **java.lang.ClassNotFoundException，即java类找不到。这类典型异常通常是由于，没有在依赖管理中声明版本，maven的仲裁的时候选取了错误的版本，而这个版本缺少我们需要的某个class而导致该错误。例如httpclient-4.4.jar升级到httpclient-4.36.jar时，类org.apache.http.conn.ssl.NoopHostnameVerifier被去掉了，如果此时我们本来需要的是4.4版本，且用到了NoopHostnameVerifier这个类，而maven仲裁时选择了4.6，则会导致ClassNotFoundException异常。**
- **java.lang.NoSuchMethodError，即找不到特定方法，第一类冲突和第二类冲突都可能导致该问题——加载的类不正确。若是第一类冲突，则是由于错误版本的Jar包与所需要版本的Jar包中的类接口不一致导致，例如antlr-2.7.2.jar升级到antlr-2.7.6.Jar时，接口antlr.collections.AST.getLine()发生变动，当maven仲裁选择了错误版本而加载了错误版本的类AST，则会导致该异常；若是第二类冲突，则是由于不同Jar包含有的同名类接口不一致导致，典型的案例：Apache的commons-lang包，2.x升级到3.x时，包名直接从commons-lang改为commons-lang3，部分接口也有所改动，由于包名不同和传递性依赖，经常会出现两种Jar包同时在classpath下，org.apache.commons.lang.StringUtils.isBlank就是其中有差异的接口之一，由于Jar包的加载顺序，导致加载了错误版本的StringUtils类，就可能出现NoSuchMethodError异常。**

- **java.lang.NoClassDefFoundError，java.lang.LinkageError等，原因和上述雷同，就不作具体案例分析了。**
- **没有报错异常，但应用的行为跟预期不一致。这类问题同样也是由于运行时加载了错误版本的类导致，但跟前面不同的是，冲突的类接口都是一致的，但具体实现逻辑有差异，当我们加载的类版本不是我们需要的实现逻辑，就会出现行为跟预期不一致问题。这类问题通常发生在我们自己内部实现的多个Jar包中，由于包路径和类名命名不规范等问题，导致两个不同的Jar包出现了接口一致但实现逻辑又各不相同的同名类，从而引发此问题。**

解决方案

一、问题排查和解决

1. 如果有异常堆栈信息，根据错误信息即可定位导致冲突的类名，然后在eclipse中CTRL+SHIFT+T或者在idea中CTRL+N就可发现该类存在于多个依赖Jar包中
2. 若步骤1无法定位冲突的类来自哪个Jar包，可在应用程序启动时加上JVM参数-verbose:class或者-XX:+TraceClassLoading，日志里会打印出每个类的加载信息，如来自哪个Jar包
3. 定位了冲突类的Jar包之后，通过mvn dependency:tree -Dverbose -Dincludes=<groupId>:<artifactId>查看是哪些地方引入的Jar包的这个版本
4. 确定Jar包来源之后，如果是第一类Jar包冲突，则可用**<excludes>**排除不需要的Jar包版本或者在依赖管理**<dependencyManagement>**中申明版本；若是第二类Jar包冲突，如果可排除，则用**<excludes>**排掉不需要的那个Jar包，若不能排，则需考虑Jar包的升级或换个别的Jar包。当然，除了这些方法，还可以从类加载器的角度来解决该问题，可参考博文——如果jar包冲突不可避免，如何实现jar包隔离，其思路值得借鉴。

二、有效避免

从上一节的解决方案可以发现，当出现第二类Jar包冲突，且冲突的Jar包又无法排除时，问题变得相当棘手，这时候要处理该冲突问题就需要较大成本了，所以，最好的方式是**在冲突发生之前能有效地规避之！**就好比数据库死锁问题，死锁避免和死锁预防就显得相当重要，若是等到真正发生死锁了，常规的做法也只能是回滚并重启部分事务，这就捉襟见肘了。那么怎样才能有效地规避Jar包冲突呢？

2.1 良好的习惯：依赖管理

对于第一类Jar包冲突问题，通常的做法是用<excludes>排除不需要的版本，但这种做法带来的问题是每次引入带有传递性依赖的Jar包时，都需要一一进行排除，非常麻烦。maven为此提供了集中管理依赖信息的机制，即依赖管理元素<dependencyManagement>，对依赖Jar包进行统一版本管理，一劳永逸。通常的做法是，在parent模块的pom文件中尽可能地声明所有相关依赖Jar包的版本，并在子pom中简单引用该构件即可。

来看个示例，当开发时确定使用的httpclient版本为4.5.1时，可在父pom中配置如下：

```
1. ...
2.   <properties>
3.     <httpclient.version>4.5.1</httpclient.version>
4.   </properties>
5.   <dependencyManagement>
6.     <dependencies>
7.       <dependency>
8.         <groupId>org.apache.httpcomponents</groupId>
9.         <artifactId>httpclient</artifactId>
10.          <version>${httpclient.version}</version>
11.      </dependency>
12.    </dependencies>
13.  </dependencyManagement>
14. ...
```

然后各个需要依赖该Jar包的子pom中配置如下依赖：

```
1. ...
2.   <dependencies>
3.     <dependency>
4.       <groupId>org.apache.httpcomponents</groupId>
5.       <artifactId>httpclient</artifactId>
6.     </dependency>
7.   </dependencies>
8. ...
```

2.2 冲突检测插件

对于第二类Jar包冲突问题，前面也提到过，其核心在于同名类出现在了多个不同的Jar包中，如果人工来排查该问题，则需要逐个点开每个Jar包，然后相互对比看有没同名的类，那得多么浪费精力啊？！好在这种费时费力的体力活能交给程序去干。maven-enforcer-plugin，这个强大的maven插件，配合extra-enforcer-rules工具，能自动扫描Jar包将冲突检测并打印出来，汗颜的是，笔者工作之前居然都没听过有这样一个插件的存在，也许是没遇到像工作中这样的冲突问题，算是涨姿势了。其原理其实也比较简单，通过扫描Jar包中的class，记录每个class对应的Jar包列表，如果有多个即是冲突了，故不必深究，我们只需要关注如何用它即可。

在**最终需要打包运行的应用模块pom**中，引入maven-enforcer-plugin的依赖，在build阶段即可发现问题，并解决它。比如对于具有parent pom的多模块项目，需要将插件依赖声明在应用模块的pom中。这里有童鞋可能会疑问，为什么不把插件依赖声明在parent pom中呢？那样依赖它的应用子模块岂不是都能复用了？这里之所以强调“打包运行的应用模块pom”，是因为冲突检测针对的是最终集成的应用，关注的是应用运行时是否会出现冲突问题，而每个不同的应用模块，各自依赖的Jar包集合是不同的，由此而产生的<ignoreClasses>列表也是有差异的，因此只能针对应用模块pom分别引入该插件。

先看示例用法如下：

```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.4.1</version>
  <executions>
    <execution>
      <id>enforce</id>
      <configuration>
        <rules>
          <dependencyConvergence/>
        </rules>
      </configuration>
      <goals>
        <goal>enforce</goal>
```

```
      </goals>
    </execution>
    <execution>
      <id>enforce-ban-duplicate-classes</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <banDuplicateClasses>
            <ignoreClasses>
              <ignoreClass>javax.*</ignoreClass>
              <ignoreClass>org.junit.*</ignoreClass>
              <ignoreClass>net.sf.cglib.*</ignoreClass>
              <ignoreClass>org.apache.commons.logging.*</ignoreClass>
              <ignoreClass>org.springframework.remoting.rmi.RmiInvocationHandler</ignoreClass>
            </ignoreClasses>
            <findAllDuplicates>true</findAllDuplicates>
          </banDuplicateClasses>
        </rules>
        <fail>true</fail>
      </configuration>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>extra-enforcer-rules</artifactId>
      <version>1.0-beta-6</version>
    </dependency>
  </dependencies>
</plugin>
```

maven-enforcer-plugin是通过很多预定义的标准规则（standard rules）和用户自定义规则，来约束maven的环境因素，如maven版本、JDK版本等等，它有很多好用的特性，具体可参见官网。而Extra Enforcer Rules则是*MojoHaus*项目下的针对maven-enforcer-plugin而开发的提供额外规则的插件，这其中就包含前面所提的重复类检测功能，具体用法可参见官网，这里就不详细叙述了。

典型案例

第一类Jar包冲突

这类Jar包冲突是最常见的也是相对比较好解决的，已经在三、冲突的表象这节中列举了部分案例，这里就不重复列举了。

第二类Jar包冲突

Spring2.5.6与Spring3.x

Spring2.5.6与Spring3.x，从单模块拆分为多模块，Jar包名称（artifactId）也从spring变为spring-submoduleName，如spring-context、spring-aop等等，并且也有少部分接口改动（Jar包升级的过程中，这也是在所难免的）。由于是不同的Jar包，经maven的传递依赖机制，就会经常性的存在这俩版本的Spring都在classpath中，从而引发潜在的冲突问题。