**参考**

https://www.xiaoniuhululu.com/2022-06-07\_ClassLoader\_and\_parental\_delegation\_mechanism

**引子**

大家想必都有过平时开发springboot 项目的时候稍微改动一点代码，就得重启，就很烦

网上一般介绍 2种方式 spring-boot-devtools,或者通过JRebel插件 来实现"热部署"

热部署就是当应用正在运行时，修改应用不需要重启应用。

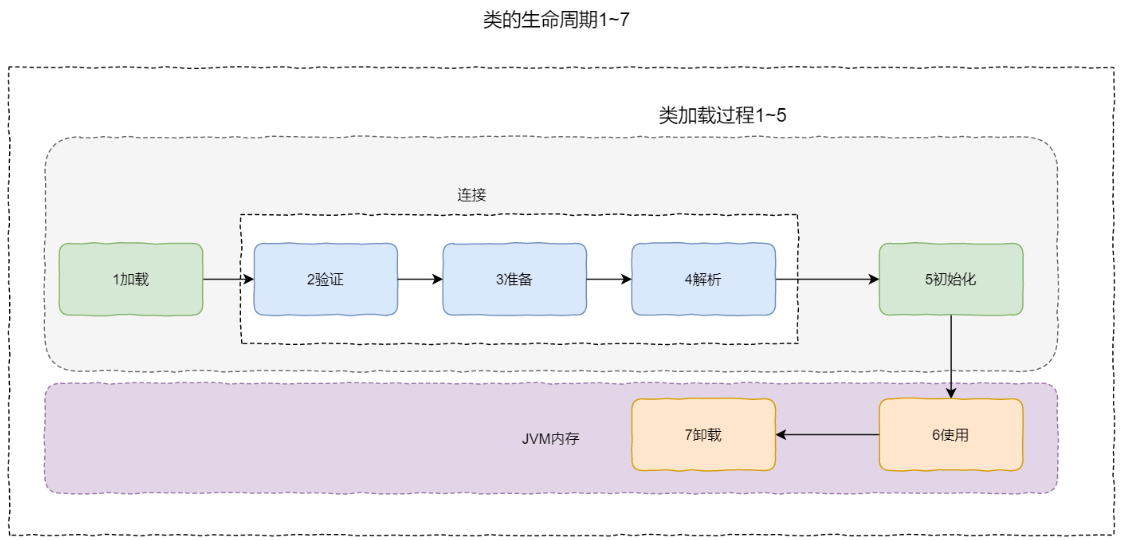
其中 spring-boot-devtools其实是自动重启，主要是节省了我们手动点击重启的时间，不算真正意义上的热部署。JRebel插件啥都好，就是需要收费

但如果平时我们在调试debug的情况下，只是在方法块内代码修改了一下，我们还得重启项目，就很浪费时间。这个时候我们其实可以直接build ，不重启项目，即可实现热部署。

## 类加载的过程

虽然我们写的时候是简单的一句，但是JVM内部的实现过程却是复杂的：

* 将硬盘上指定位置的Person.class文件加载进内存
* 执行main方法时，在栈内存中开辟了main方法的空间（压栈-进栈），然后在main方法的栈区分配了一个变量zhang。
* 执行new，在堆内存中开辟一个 实体类的 空间，分配了一个内存首地址值
* 调用该实体类对应的构造函数，进行初始化（如果没有构造函数，Java会补上一个默认构造函数）。
* 将实体类的 首地址赋值给zhang，变量zhang就引用了该实体。（指向了该对象）



1. 加载

**加载**阶段：指的是将类对应的.class文件中的二进制字节流读入到内存中，将这个字节流转化为方法区的运行时数据结构，然后在堆区创建一个java.lang.Class 对象，作为对方法区中这些数据的访问入口

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是我们最可以控制的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义类加载器来完成加载。这个我们文章后面再详细讲

1. 验证

验证阶段：校验字节码文件正确性。这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

这部分对开发者而言是无法干预的，以下内容了解即可

验证阶段大致会完成4个阶段的检验动作：

* **文件格式验证**：验证字节流是否符合Class文件格式的规范；例如：是否以0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
* **元数据验证**：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。
* **字节码验证**：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
* **符号引用验证**：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

1. 准备

准备阶段：为类变量(static 修饰的变量)分配内存，并将其初始化为默认值

注意此阶段**仅仅**是为**类变量 即静态变量**分配内存，并将其初始化为默认值

1. 解析

解析阶段： 是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

**符号引用**就是一组符号来描述目标，可以是任何字面量。

**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。

这个阶段了解一下即可

1. 初始化

直到初始化阶段，Java虚拟机才真正开始执行类中编写的Java程序代码，将主导权移交给应用程序。

初始化阶段 是类加载过程的最后一个步骤，之前介绍的几个类加载的动作里，除了在加载阶段用户应用程序可以通过自定义类加载器的方式局部参与外，其余动作都完全由Java虚拟机来主导控 制。

Java程序对类的使用方式可分为两种：主动使用与被动使用。一般来说只有当对类的首次主动使用的时候才会导致类的初始化，所以主动使用又叫做类加载过程中“初始化”开始的时机。

类实例初始化方式，主要是以下几种：

1、创建类的实例，也就是new的方式

2、访问某个类或接口的静态变量，或者对该静态变量赋值

3、调用类的静态方法

4、反射（如Class.forName("com.test.Person")）

5、初始化某个类的子类，则其父类也会被初始化

6、Java虚拟机启动时被标明为启动类的类（JavaTest），还有就是Main方法的类会 首先被初始化

1. 使用

当JVM完成初始化阶段之后，JVM便开始从入口方法开始执行用户的程序代码

1. 卸载

当用户程序代码执行完毕后，JVM便开始销毁创建的Class对象，最后负责运行的JVM也退出内存

在如下几种情况下，Java虚拟机将结束生命周期

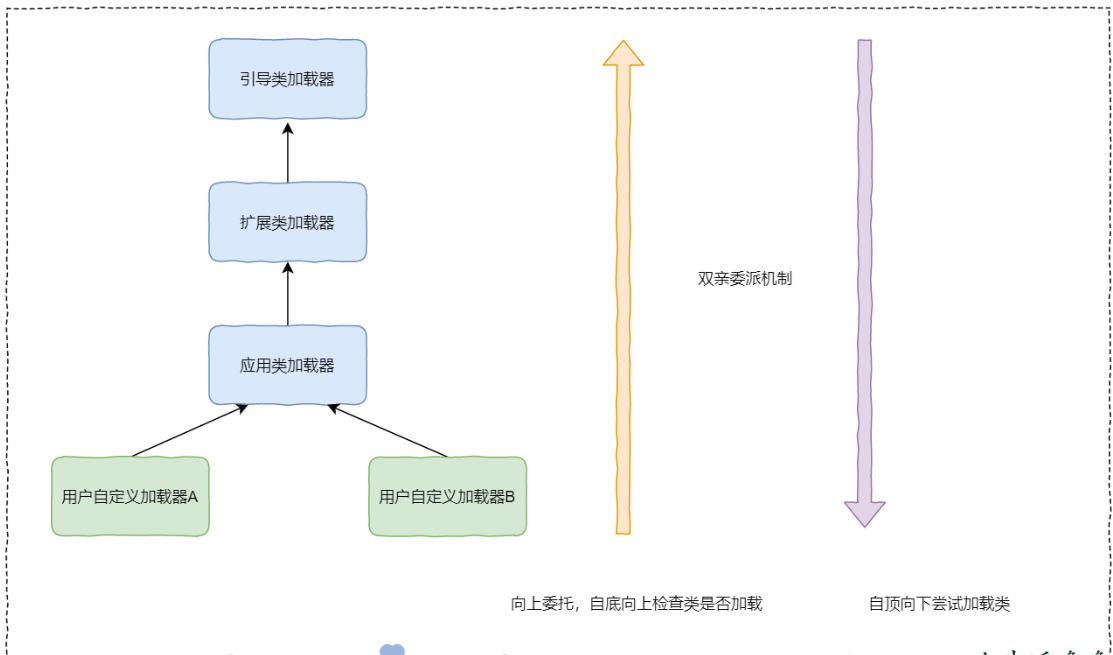
* 执行了System.exit()方法
* 程序正常执行结束
* 程序在执行过程中遇到了异常或错误而异常终止
* 由于操作系统出现错误而导致Java虚拟机进程终止

## 类加载器 与 双亲委派机制

上文类加载过程中，是需要类加载器的参与，**类加载器在Java中非常重要**，它使得 Java 类可以被动态加载到 Java 虚拟机中并执行

那什么是类加载器？通过一个类的全限定名来获取描述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

Java虚拟机支持类加载器的种类：主要包括3中： 引导类加载器（Bootstrap ClassLoader）、扩展类加载器（Extension ClassLoader）、应用类加载器（系统类加载器，AppClassLoader)，另外我们还可以自定义加载器-用户自定义类加载器



1. 引导类加载器（Bootstrap ClassLoader）：BootStrapClassLoader是由c++实现的。引导类加载器加载java运行过程中的**核心类库**JRE＼lib＼rt.jar，sunrsasign.jar， charsets.jar， jce.jar， jsse.jar， plugin.jar 以及存放 在JRE＼classes里的类，也就是JDK提供的类等常见的比如：Object、Stirng、List等
2. 扩展类加载器（Extension ClassLoader）：它用来加载/jre/lib/ext目录以及java.ext.dirs系统变量指定的类路径下的类。
3. 应用类加载器（AppClassLoader)：它主要加载应用程序ClassPath下的类（包含jar包中的类）。它是java应用程序默认的类加载器。其实就是加载我们一般开发使用的类
4. 用户自定义类加载器： 用户根据自定义需求，自由的定制加载的逻辑，只需**继承应用类加载器AppClassLoader**，负责加载用户自定义路径下的class字节码文件
5. 线程上下文类加载器：除了以上列举的三种类加载器，其实还有一种比较特殊的类型就是线程上下文类加载器。ThreadContextClassLoader可以是上述类加载器的任意一种，这个我们下文再细说



sun.misc.Launcher$AppClassLoader@18b4aac2  
sun.misc.Launcher$ExtClassLoader@5caf905d  
null

结果显示分别打印应用类加载器、扩展类加载器和引导类加载器

由于 引导类加载器 是由c++实现的，所以并不存在一个Java的类，因此会打印出null

我们还可以看到结果里面打印了 sun.misc.Launcher，这个是什么东东？

其实**Launcher**是JRE中用于启动程序入口main()的类，我们看下Launcher的源码：

public class Launcher {

private static Launcher launcher = new Launcher();

private static String bootClassPath =

System.getProperty("sun.boot.class.path");

public static Launcher getLauncher() {

return launcher;

}

private ClassLoader loader;

public Launcher() {

// Create the extension class loader

ClassLoader extcl;

try {

extcl = ExtClassLoader.getExtClassLoader(); //加载扩展类类加载器

} catch (IOException e) {

throw new InternalError(

"Could not create extension class loader", e);

}

// Now create the class loader to use to launch the application

try {

loader = AppClassLoader.getAppClassLoader(extcl);//加载应用程序类加载器,并设置parent为extClassLoader

} catch (IOException e) {

throw new InternalError(

"Could not create application class loader", e);

}

Thread.currentThread().setContextClassLoader(loader); //设置AppClassLoader为线程上下文类加载器

}

/\*

\* Returns the class loader used to launch the main application.

\*/

public ClassLoader getClassLoader() {

return loader;

}

/\*

\* The class loader used for loading installed extensions.

\*/

static class ExtClassLoader extends URLClassLoader {}

/\*\*

\* The class loader used for loading from java.class.path.

\* runs in a restricted security context.

\*/

static class AppClassLoader extends URLClassLoader {}

其中loader = AppClassLoader.getAppClassLoader(extcl);的核心方法源码如下：

private ClassLoader(Void unused, ClassLoader parent) {

this.parent = parent;//设置parent

if (ParallelLoaders.isRegistered(this.getClass())) {

parallelLockMap = new ConcurrentHashMap<>();

package2certs = new ConcurrentHashMap<>();

assertionLock = new Object();

} else {

// no finer-grained lock; lock on the classloader instance

parallelLockMap = null;

package2certs = new Hashtable<>();

assertionLock = this;

}

}

通过以上源码我们可以知晓：

1. Launcher的ClassLoader是BootstrapClassLoader，在Launcher创建的同时，还会同时创建ExtClassLoader，AppClassLoader（并设置其parent为extClassLoader）。其中代码中 “sun.boot.class.path”是BootstrapClassLoader加载的jar包路径。
2. 这几种类加载器 都遵循 双亲委派机制

双亲委派机制说的其实就是，当一个类加载器收到一个类加载请求时，会去判断有没有加载过，如果加载过直接返回，否则该类加载器会把请求先委派给父类加载器。每个类加载器都是如此，只有在父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。

双亲委派模式优势：

1. 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, 这样保证了每个类只被加载一次。
2. 保护程序安全，防止核心API被随意篡改，比如 java核心api中定义类型不会被随意替换

我们这里看一个例子：

我们新建一个自己的类“String”放在src/java/lang目录下



## ClassLoader

ClassLoader 是一个抽象类，负责加载类，像 ExtClassLoader，AppClassLoader **都是由该类派生出来**，实现不同的类装载机制。这块的源码太多了，就不贴了

我们来看下 它的核心方法loadClass()，传入需要加载的**类名**，它会帮你加载:



loadClass()源码 展示了，一般加载.class文件大致流程：

1. 先去缓存中 检查是否已经加载该类，有就直接返回，避免重复加载；没有就下一步
2. 遵循 双亲委派机制，来加载.class文件
3. 上面两步都失败了，调用findClass()方法，让当前类加载器加载

注意：由于ClassLoader类是抽象类，而抽象类是无法通过new创建对象的，所以它最核心的findClass()方法，没有具体实现，只抛了一个异常，而且是protected的，这是应用了模板方法模式，具体的findClass()方法丢给子类实现, 所以继承的子类得**重写**该方法。

## 自定义类加载器

### 编写一个自定义的类加载器

那我们仿照 ExtClassLoader，AppClassLoader 来实现一个自定义的类加载器，我们同样是继承ClassLoader类

编写一个测试类TestPerson



接着 编写一个自定义类加载器MyTestClassLoader：

public class MyTestClassLoader extends ClassLoader {

final String classNameSpecify = "TestPerson";

public MyTestClassLoader() {

}

public MyTestClassLoader(ClassLoader parent)

{

super(parent);

}

protected Class<?> findClass(String name) throws ClassNotFoundException

{

File file = getClassFile(name);

try

{

byte[] bytes = getClassBytes(file);

Class<?> c = this.defineClass(name, bytes, 0, bytes.length);

return c;

}

catch (Exception e)

{

e.printStackTrace();

}

return super.findClass(name);

}

private File getClassFile(String name)

{

File file = new File("D:\\ideaProjects\\src\\main\\java\\com\\zj\\ideaprojects\\test2\\"+ classNameSpecify+ ".class");

return file;

}

private byte[] getClassBytes(File file) throws Exception

{

// 这里要读入.class的字节，因此要使用字节流

FileInputStream fis = new FileInputStream(file);

FileChannel fc = fis.getChannel();

ByteArrayOutputStream baos = new ByteArrayOutputStream();

WritableByteChannel wbc = Channels.newChannel(baos);

ByteBuffer by = ByteBuffer.allocate(1024);

while (true)

{

int i = fc.read(by);

if (i == 0 || i == -1)

break;

by.flip();

wbc.write(by);

by.clear();

}

fis.close();

return baos.toByteArray();

}

//我们这边要打破双亲委派模型，重写整个loadClass方法

@Override

public Class<?> loadClass(String name) throws ClassNotFoundException {

Class<?> c = findLoadedClass(name);

if (c == null && name.contains(classNameSpecify)){//指定的类，不走双亲委派机制，自定义加载

c = findClass(name);

if (c != null){

return c;

}

}

return super.loadClass(name);

}

}

最后在编写一个测试controller:

@RestController

public class TestClassController {

@RequestMapping(value = "testClass",method = {RequestMethod.GET, RequestMethod.POST})

public void testClassLoader() throws ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException, InvocationTargetException {

MyTestClassLoader myTestClassLoader = new MyTestClassLoader();

Class<?> c1 = Class.forName("com.zj.ideaprojects.test2.TestPerson", true, myTestClassLoader);

Object obj = c1.newInstance();

System.out.println("当前类加载器："+obj.getClass().getClassLoader());

obj.getClass().getMethod("print").invoke(obj);

}

}

先找到TestPerson所在的目录， 执行命令： javac TestPerson，生成TestPerson.class

这里没有使用idea的build,是因为我们代码的class读取路径 是写死了的，不走默认CLASSPATH

D:\ideaProjects\src\main\java\com\zj\ideaprojects\test2\TestPerson.class

我们然后用postman调用testClassLoader()测试接口

结果：

当前类加载器：com.zj.ideaprojects.test2.MyTestClassLoader@1d75e392

hello my name is: xiao ming

然后修改TestPerson，将name 改为 “xiao niu”

public class TestPerson {

String name = "xiao niu";

public void print(){

System.out.println("hello my name is: "+ name);

}

}

然后在当前目录 重新编译, 执行命令： javac TestPerson，会在当前目录重新生成TestPerson.class  
不重启项目，直接用postman 直接调这个测试接口  
结果：

当前类加载器：com.zj.ideaprojects.test2.MyTestClassLoader@7091bd27

hello my name is: xiao niu

这样就实现了“热部署”！！！

### 为什么我们这边要打破双亲委派机制？

如果不打破的话,结果 当前类加载器会显示”sun.misc.Launcher$AppClassLoader”，原因是由于idea启动项目的时候会自动帮我们编译，将class放到 CLASSPATH路径下。其实可以把默认路径下的.class删除也行。这里也是为了展示如何打破双亲委派机制，才如此实现的。

官方推荐我们自定义类加载器时，遵循双亲委派机制。但是凡事得看实际需求嘛

### 自定义类加载器时，如何打破双亲委派机制

通过上面的例子我们可以看出：  
1、如果不想打破双亲委派机制，我们自定义类加载器，那么只需要重写findClass方法即可

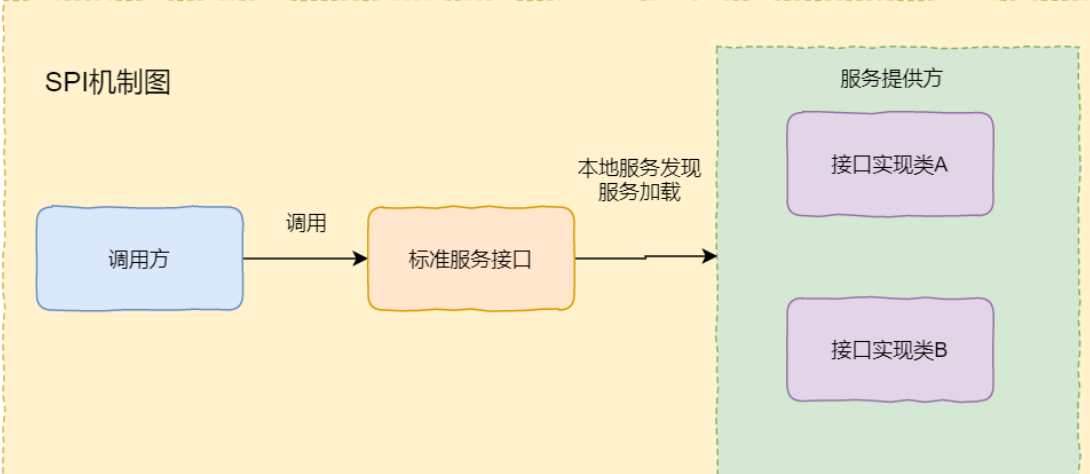
2、如果想打破双亲委派机制，我们自定义类加载器，那么还得重写整个loadClass方法

## SPI机制 与 线程上下文类加载器

如果你阅读到这里，你会发现双亲委派机制的各种好处，但万物都不是绝对正确的，我们需要一分为二地看待问题。

在某些场景下双亲委派制过于局限，所以有时候必须打破双亲委派机制来达到目的。比如 ：SPI机制、线程上下文类加载器

1. SPI(Service Provider Interface)服务提供接口。它是jdk内置的一种服务发现机制，**将装配的控制权移到程序之外**，在模块化设计中这个机制尤其重要，其核心思想就是 **让服务定义与实现分离**、**解耦**。



1. 线程上下文类加载器（context class loader）是可以**破坏Java类加载委托机制**，使程序可以逆向使用类加载器，使得java类加载体系显得更灵活。

Java 应用运行的初始线程的上下文类加载器是应用类加载器，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。Java.lang.Thread中的方法getContextClassLoader()和 setContextClassLoader(ClassLoader cl)用来获取和设置线程的上下文类加载器。如果没有通过 setContextClassLoader(ClassLoader cl)方法进行设置的话，线程将继承其父线程的上下文类加载器。

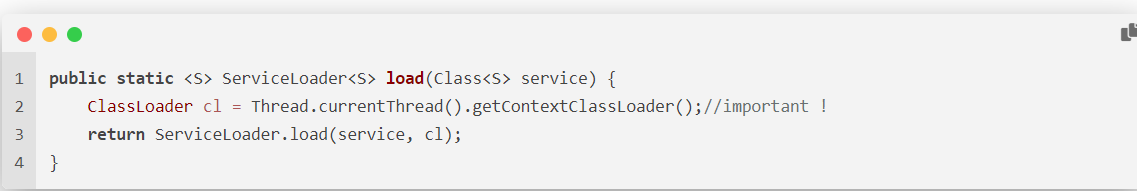
SPI机制在框架的设计上应用广泛，下面举几个常用的例子：

### JDBC

平时获取jdbc,我们可以这样：  
Connection connection =DriverManager.getConnection("jdbc://localhost:3306");

我们读DriverManager的源码发现：其实就是查询classPath下，所有META-INF下给定Class名的文件，并将其内容返回，使用迭代器遍历，这里遍历的内部使用Class.forName加载了类。

其中有一处非常重要 ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);我们看下它的实现：



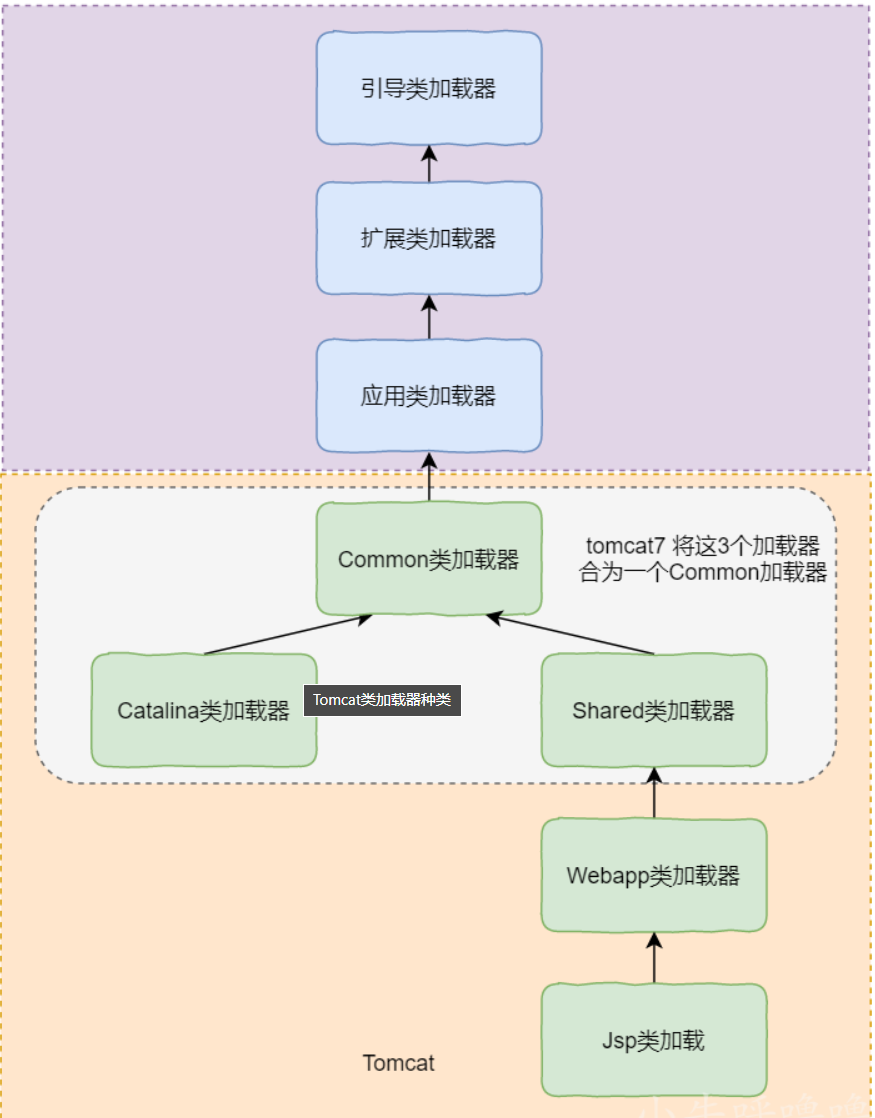
我们可以看出JDBC，DriverManager类和ServiceLoader类都是属于核心库 rt.jar 的，它们的类加载器是Bootstrap ClassLoader类加载器。而具体的数据库驱动相关功能却是**第三方提供的**，第三方的类不能被引导类加载器（Bootstrap ClassLoader）加载。

所以java.util.ServiceLoader类进行动态装载时，使用了线程的上下文类加载器(ThreadContextClassLoader)**让父级类加载器能通过调用子级类加载器来加载类**，这**打破了双亲委派机制**。

### Tomcat

Tomcat是web容器，我们把war包放到 tomcat 的webapp目录下，这意味着一个tomcat可以部署多个应用程序。

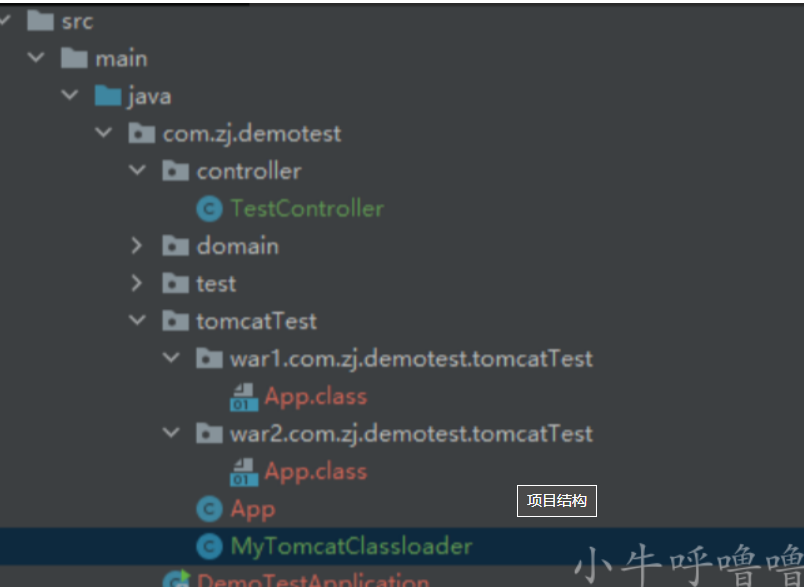
不同的应用程序可能会依赖同一个第三方类库的不同版本，但是不同版本的类库中某一个类的全路径名可能是一样的。防止出现一个应用中加载的类库会影响另一个应用的情况。如果采用默认的双亲委派类加载机制，那么是无法加载多个相同的类。



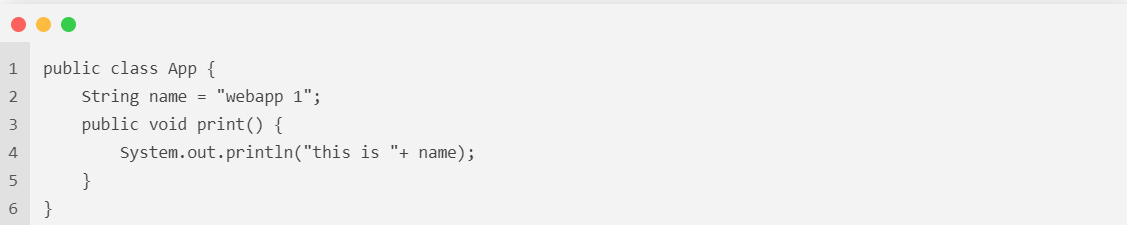
1. 如果Tomcat本身的依赖和Web应用还需要共享，Common类加载器（CommonClassLoader）来装载实现共享
2. Catalina类加载器(CatalinaClassLoader) 用来 隔绝Web应用程序与Tomcat本身的类
3. Shared类加载器（SharedClassLoader）：如果WebAppClassLoader自身没有加载到某个类，那就委托SharedClassLoader去加载
4. **WebAppClassLoader： 为了实现隔离性，优先加载 Web 应用自己定义的类，所以没有遵照双亲委派的约定，**每一个应用自己的类加载器**WebAppClassLoader**(多个应用程序，就有多个WebAppClassLoader)**，**负责优先加载本身的目录下的class文件**，**加载不到时再交给CommonClassLoader以及上层的ClassLoader进行加载**，**这破坏了双亲委派机制。
5. Jsp类加载器（JasperLoader）：实现热部署的功能，修改文件不用重启就自动重新装载类库。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HotSwap功能。

我们来模拟一下tomcat 多个版本代码共存：

这边的例子换了个电脑，所以目录结构、路径与上面的例子有点变化



我们先编写 App类

javac App生成的App.class 放入 tomcatTest\war1\com\zj\demotest\tomcatTest 目录下

将name改为webapp 2，重新生成的App.class 放入 tomcatTest\war2\com\zj\demotest\tomcatTest 目录下

然后我们编写类加载器：

public class MyTomcatClassloader extends ClassLoader {

private String classPath;

public MyTomcatClassloader(String classPath) {

this.classPath = classPath;

}

@Override

protected Class<?> findClass(String name) throws ClassNotFoundException

{

File file = getClassFile(name);

try

{

byte[] bytes = getClassBytes(file);

Class<?> c = this.defineClass(name, bytes, 0, bytes.length);

return c;

}

catch (Exception e)

{

e.printStackTrace();

}

return super.findClass(name);

}

private File getClassFile(String name)

{

name = name.replaceAll("\\.", "/");

File file = new File(classPath+ "/"+ name + ".class");//拼接路径，找到class文件

return file;

}

private byte[] getClassBytes(File file) throws Exception

{

// 这里要读入.class的字节，因此要使用字节流

FileInputStream fis = new FileInputStream(file);

FileChannel fc = fis.getChannel();

ByteArrayOutputStream baos = new ByteArrayOutputStream();

WritableByteChannel wbc = Channels.newChannel(baos);

ByteBuffer by = ByteBuffer.allocate(1024);

while (true)

{

int i = fc.read(by);

if (i == 0 || i == -1) {

break;

}

by.flip();

wbc.write(by);

by.clear();

}

fis.close();

return baos.toByteArray();

}

//我们这边要打破双亲委派模型，重写整个loadClass方法

@Override

public Class<?> loadClass(String name) throws ClassNotFoundException {

Class<?> c = findLoadedClass(name);

if (c == null && name.contains("tomcatTest")){//指定的目录下的类，不走双亲委派机制，自定义加载

c = findClass(name);

if (c != null){

return c;

}

}

return super.loadClass(name);

}

}

最后编写测试controller:

@RestController

public class TestController {

@RequestMapping(value = "/testTomcat",method = {RequestMethod.GET, RequestMethod.POST})

public void testclass() throws ClassNotFoundException, IllegalAccessException, InstantiationException, NoSuchMethodException, InvocationTargetException {

MyTomcatClassloader myTomcatClassloader = new MyTomcatClassloader("D:\\GiteeProjects\\study-java\\demo-test\\src\\main\\java\\com\\zj\\demotest\\tomcatTest\\war1");

Class cl = myTomcatClassloader.loadClass("com.zj.demotest.tomcatTest.App");

Object obj = cl.newInstance();

System.out.println("当前类加载器："+obj.getClass().getClassLoader());

obj.getClass().getMethod("print").invoke(obj);

MyTomcatClassloader myTomcatClassloader22 = new MyTomcatClassloader("D:\\GiteeProjects\\study-java\\demo-test\\src\\main\\java\\com\\zj\\demotest\\tomcatTest\\war2");

Class cl22 = myTomcatClassloader22.loadClass("com.zj.demotest.tomcatTest.App");

Object obj22 = cl22.newInstance();

System.out.println("当前类加载器："+obj22.getClass().getClassLoader());

obj22.getClass().getMethod("print").invoke(obj22);

}

}

然后postman 调一下这个接口,  
结果：

当前类加载器：com.zj.demotest.tomcatTest.MyTomcatClassloader@18fbb876

this is webapp 1

当前类加载器：com.zj.demotest.tomcatTest.MyTomcatClassloader@5f7ed4a9

this is webapp 2

我们发现2个同样的类能共存在同一个JVM中，互不影响。

注意： **同一个JVM内，2个相同的包名和类名的对象是可以共存的，前提是他们的类加载器不一样。所以我们要判断多个类对象是否是同一个，除了要看包名和类名相同，还得注意他们的类加载器是否一致**

### SpringBoot Starter

springboot自动配置的原因是因为使用了@EnableAutoConfiguration注解。

当程序包含了EnableAutoConfiguration注解，那么就会执行下面的方法，然后会加载所有spring.factories文件，将其内容封装成一个map，spring.factories其实就是一个名字特殊的properties文件。

在spring-boot应用启动时，会调用**loadFactoryNames**方法，其中传递的一个参数就是：org.springframework.boot.autoconfigure.EnableAutoConfiguration

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {

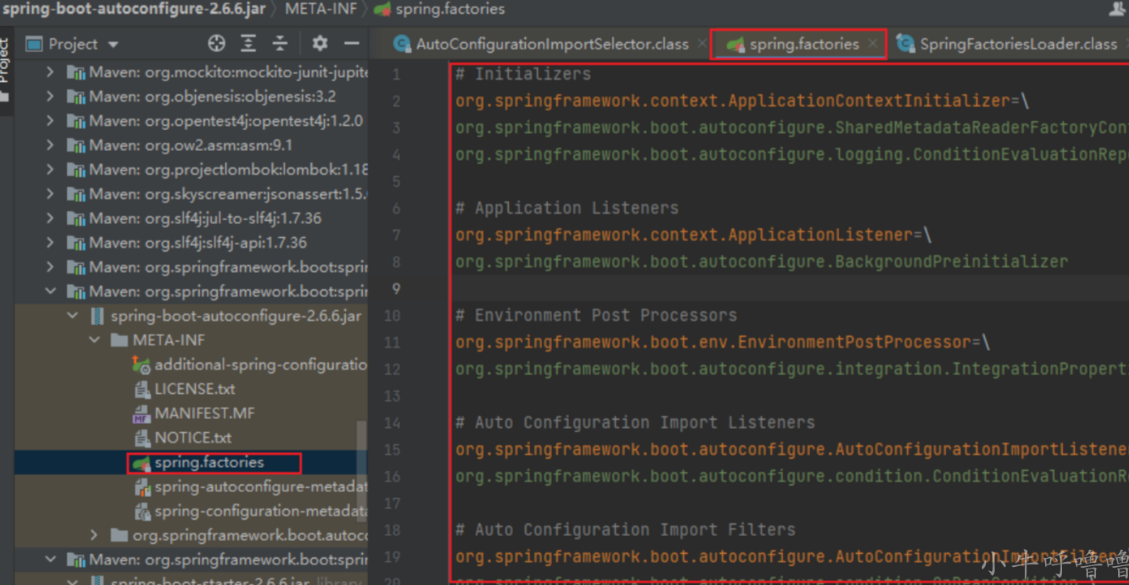
List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());

Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packaging, make sure that file is correct.");

return configurations;

}

META-INF/spring.factories会被读取到。



它还使用了this.getBeanClassLoader() 获取类加载器。所以我们立刻明白了文章一开始的例子，SpringBoot项目直接build项目，不重启项目，就能实现热部署效果。

## 尾语

类加载器是 Java 语言的一个创新，它使得动态安装和更新软件组件成为可能。同时我们应该了解双亲委派机制的优缺点和应用场景，这些可能比较难但对于我们来说却很重要。