**第9章 类加载及执行子系统的案例与实战**

Class文件格式与执行引擎这部分，用户的程序能直接影响的内容很少。Class文件的存储格式、类型何时加载、如何连接，以及虚拟机如何执行字节码指令等都是由虚拟机直接控制的行为，用户程序无法对其进行改变。

能通过程序进行操作的，主要是**字节码生成**与**类加载器**这两部分的功能。但仅仅这两点功能，就已经出现了许多值得欣赏和借鉴的思路，这些思路后来成为了许多常用功能和程序实现的基础。

类加载器

一、Tomcat：正统的类加载器架构

一个功能健全的Web服务器，需要解决以下几个问题：

（1）部署在同一个服务器上的两个Web应用程序所使用的Java类库可以实现相互隔离，这是最基本的需求。

（2）部署在同一个服务器上的两个Web应用程序所使用的Java类库可以互相共享。

（3）服务器需要尽可能地保证自身的安全不受部署的Web应用程序影响。

（4）支持JSP应用的服务器，需要支持JSP生成类的热替换（HotSwap）。即修改后无需重启。

由于存在以上问题，所以一个ClassPath就无法满足需求。各种Web服务器都提供了好几个ClassPath路径供用户存放第三方类库，这些路径一般以“lib”或“classes”命名。被放置在不同路径中的类库，具备不同的访问范围和服务对象。通常，每一个目录都会有一个相应的自定义类加载器去加载放置在里面的Java类库。

Tomcat具体规划用户类库结构和类加载器如下：

有3组目录（“/common/\*”、“/server/\*”、“/shared/\*”）可以存放Java类库，另外还可以加上Web应用程序自身的目录“/WEB-INF/\*”，一共4组，把Java类库放置在这些目录中的含义分别如下：

/common 目录：类库可被**Tomcat和所有的Web应用程序**共同使用

/server 目录：类库可被Tomcat使用，对所有的Web应用程序都不可见。

/shared 目录：类库可被所有的Web应用程序共同使用，但对Tomcat自己不可见。

/WebApp/WEB-INF目录：类库仅仅可以被**此Web应用程序**使用，对Tomcat和其他Web应用程序都不可见。

**启动类加载器**

**Bootstrap ClassLoader**

JDK

**扩展类加载器**

**Extension ClassLoader**

**应用程序类加载器**

**Application ClassLoader**

**Jsp类加载器**

**JasperClassLoader**

**WebApp类加载器**

**WebAppClassLoader**

**Catalina类加载器**

**CatalinaClassLoader**

**Shared类加载器**

**SharedClassLoader**

**Common类加载器**

**CommonClassLoader**

**加载/common/\*类库**

**加载/shared/\*类库**

Tomcat

**加载/server/\*类库**

**加载/WebApp/WEB-INF类库**

**加载某个JSP的Class**

其中WebApp类加载器和Jsp类加载器通常会存在多个实例，每一个Web应用程序对应一个WebApp类加载器，每一个JSP文件对应一个Jsp类加载器。

从图中委派关系可知，CommonClassLoader类加载器能加载的类都可以被CatalinaClassLoader和SharedClassLoader使用，而CatalinaClassLoader和SharedClassLoader自己能加载的类则与对方相互隔离。而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的哪一个Class，它出现的目的就是为了被丢弃：当服务器检测到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。

对于Tomcat的6.x版本，只有指定了tomcat/conf/catalina.properties配置文件的server.loader和share.loader项后才会真正建立CatalinaClassLoader和SharedClassLoader的实例，否则会用到这两个类加载器的地方都会用CommonClassLoader的实例代替，而默认的配置文件中没有设置这两个loader项目，所以Tomcat 6.x顺利成章地把/common、/server和/shared三个目录默认合并到一起变成一个/lib目录，这个目录里的类库相当于以前/common目录中类库的作用。如果默认设置不能满足需要，用户可以通过修改配置文件指定的server.loader和share.loader的方式重新启用Tomcat 5.x的加载器架构。

二、OSGi：灵活的类加载器架构

Java程序社区观点：“学习JEE规范，去看JBoss源码；学习类加载器，就去看OSGi源码”。

OSGi最著名的应用案例就是Eclipse IDE；

OSGi中的每个模块（Bundle）与普通的java类库区别并不大，两者一般都以JAR格式进行封装，并且内部存储的都是Java Package和Class。

但是一个Bundle可以声明它所依赖的Java Package（Import-Package），也可以声明它允许导出发布的Java Package（Export-Package）。在OSGi里，Bundle之间的依赖关系从传统的上层模块依赖底层模块转变为平级模块之间的依赖，而且类库的可见性能得到非常精确的控制。**一个模块里只有被Export过的Package才可能由外界访问，其他的Package和Class将会隐藏起来**。

除了更精确的模块划分和可见性控制外，引入OSGi的另外一个重要理由是，基于OSGi的程序很可能可以实现模块级的热插拔功能，当程序升级更新或调试除错时，可以只停用、重新安装然后启用程序的其中一部分，这对企业级程序开发来说意义重大。

OSGi之所以有如上的特点，原因在于其灵活的类加载器架构。OSGi的Bundle类加载器之间只有规则，没有固定的委派关系。例如，某个Bundle声明了一个它依赖的Package，如果有其他Bundle声明发布了这个Package，那么所有对这个Package的类加载器动作都会委派给发布它的Bundle类加载器去完成。不涉及某个具体的Package时，各个Bundle加载器都是平级关系。只有具体使用某个Package和Class的时候，才会根据Package导入导出定义来构造Bundle间的委派和依赖。

另外，一个Bundle类加载器为其他Bundle提供服务时，会根据Export-Package列表严格控制访问范围。如果一个类存在于Bundle的类库中，但是没有被Export，那么这个Bundle的类加载器能找到这个类，但绝不会提供给其他Bundle使用，而且OSGi平台也不会把其他Bundle的类加载器请求分配给这个Bundle来处理。

eg：有Bundle A、Bundle B、Bundle C三个模块，且依赖关系如下：

Bundle A：声明发布了package A，依赖了java.\*的包。

Bundle B：声明依赖了package A和package C，同时也依赖了java.\*的包。

Bundle C：声明依赖了package A和package C，同时也依赖了java.\*的包。

**父类加载器**

**ParentCLassClassLoader**

**BundleA类加载器**

**BundleACLassClassLoader**

**BundleB类加载器**

**BundleBCLassClassLoader**

**BundleC类加载器**

**BundleCCLassClassLoader**

OSGi网状结构

类加载时可能进行的查找规则如下：  
（1）以java.\*开头的类，委派给父类加载器加载；

（2）否则，委派列表名单内的类，委派给父类加载器加载；

（3）否则，Import列表中的类，委派给Export这个类的Bundle的类加载器加载；

（4）否则，查找当前Bundle的Classpath，使用自己的类加载器加载

（5）否则，查找当前是否在自己的Fragment Bundle中，如果是，则委派给Fragment Bundle的类加载器加载。

（6）否则，查找Dynamic Import列表的Bundle，委派给对应Bundle的类加载器加载。

（7）否则，类查找失败。

如果Bundle A依赖Bundle B的Package B，而Bundle B又依赖于Bundle A的Package A，这两个Bundle进行类加载时就很容易发生死锁。

具体原因为：当Bundle A加载Package B的类时，首先需要锁定当前类加载器的实例对象（java.lang.ClassLoader.loadClass()是一个synchronized方法），然后把请求委派给Bundle B的加载器去处理，而此时Bundle B也正好想加载Package A的类，它也先锁定自己的加载器再去请求Bundle A的加载器处理，这样两个加载器都在等待对方处理自己的请求，而对方处理完之前自己又一直处于同步锁定的状态，因此它们就互相死锁，永远无法完成加载请求。

三、字节码生成技术与动态代理的实现

关于字节码生成：

（1）javassist、CGLib、ASM等字节码类库

（2）javac

（3）Web服务器中的JSP编译器

（4）编译时植入的AOP框架

（5）反射

**（6）动态代理技术**

相关类： java.lang.reflect.Proxy

java.lang.reflect.InvocationHandler接口

相关应用：Spring内部通过动态代理对Bean进行组织管理

优势：实现了在原始类和接口还未知的时候，便确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景之中。

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class DynamicProxyTest {

interface IHello {

void sayHello();

}

/\*\*

\* 委托类/被代理类

\*/

static class Hello implements IHello {

@Override

public void sayHello(){

System.out.println("hello world");

}

}

/\*\*

\* InvocationHandler接口的实现类

\*/

static class DynamicProxy implements InvocationHandler {

Object originalObj;//委托类对象成员

Object bind(Object originalObj){

this.originalObj = originalObj;

return Proxy.newProxyInstance(

originalObj.getClass().getClassLoader(),

originalObj.getClass().getInterfaces(),

this);//返回生成的代理类对象

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("welcome");

return method.invoke(originalObj, args);

}

}

public static void main(String[] args) {

IHello hello = (IHello) new DynamicProxy().bind(new Hello());

//返回了一个实现IHello接口，并且代理了new Hello()实例行为的代理对象

hello.sayHello();

//输出：

//welcome

//hello world

System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");

}

}

四、Retrotranslator：跨越JDK版本

JDK1.5版本是对Java程序编写习惯改变最大，新加入的特性有:自动装箱、泛型、动态注解、枚举、变长参数、遍历循环（foreach循环）、静态导入等。但如果原有程序架构版本较老，为了使用新版本JDK的特性进行开发，而又不影响和改动老版本的程序架构，可以使用一种名为“Java逆向移植”的工具，其中比较突出的是Retrotranslator。