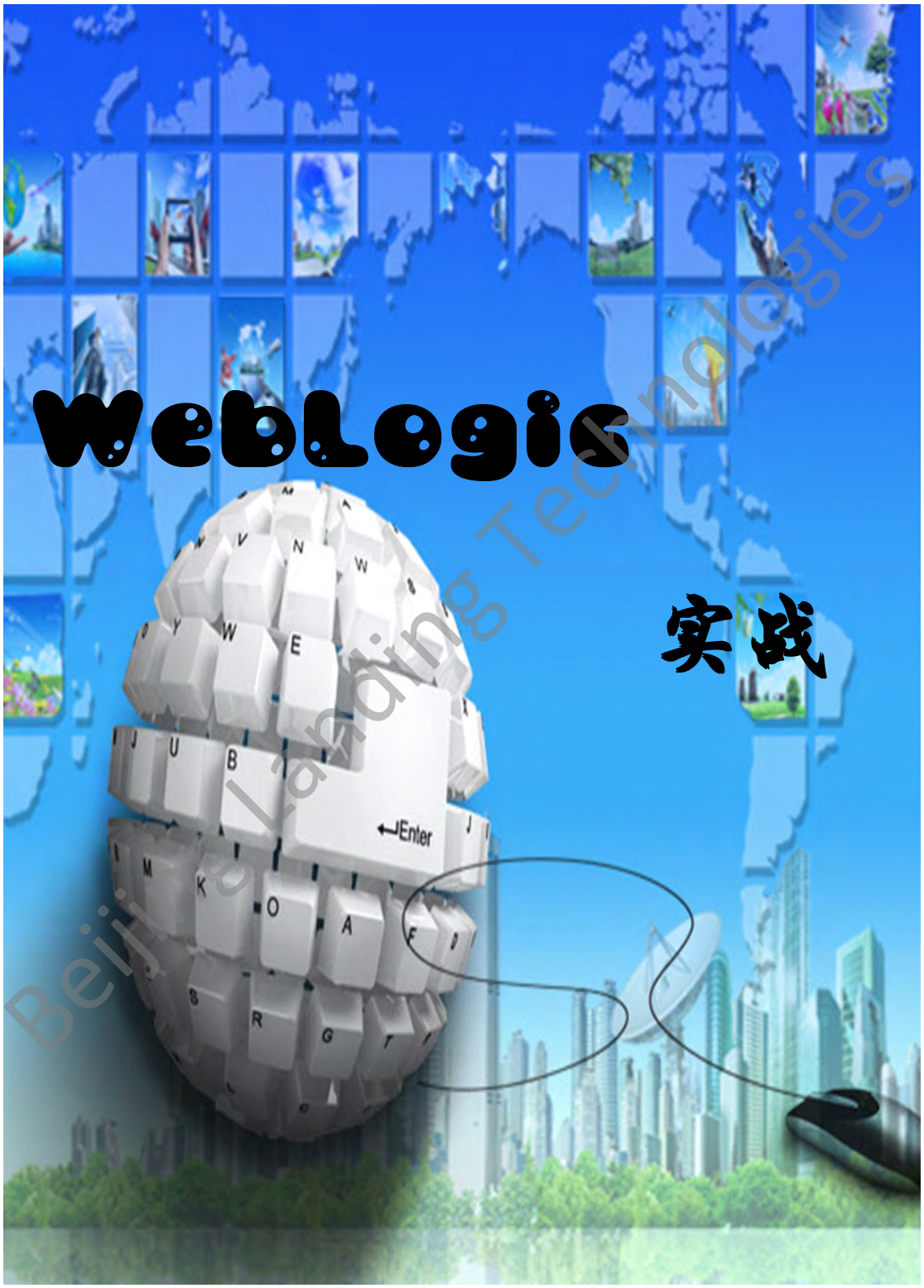
联动北方
LANDING

企业级 IT 运维宝典系列



联动北方 出品

目　录

第 6 章 Java 虚拟机相关知识

6.1 JVM 简介

Java Virtual Machine（Java 虚拟机），它是一个虚构出来的计算机,是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java 虚拟机有自己完善的硬件架构,如处理器、堆栈、寄存器等,还具有相应的指令系统。JVM 屏蔽了与具体操作系统平台相关的信息,使得 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

不过需要说明的是，Java 虚拟机在执行字节码时,实际上最终还是把字节码解释成具体平台上的机器指令执行。

6.2 常见 JDK 的内存机制

下面以 Sun 的 JDK 为例，讲述 JDK 一般的内存机制。

Java 的内存由三个代组成，如下图所示：

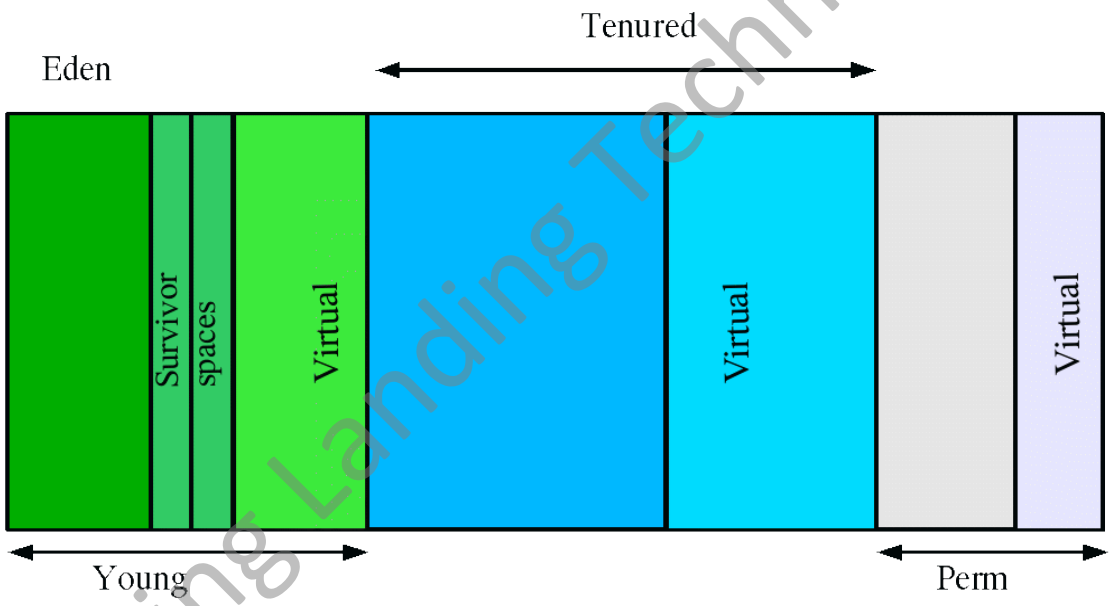


图 6-1

Java 内存的这三个代分别为：

1. Young（年青代，也叫 new 区）：

Young 还可以分为 Eden 区和两个 Survivor 区（from 和 to，这两个 Survivor 区大小严格一至），新的对象实例总是首先放在 Eden 区，Survivor 区作为 Eden 区和 Tenure(old)的缓冲，可以向 Tenure(old)转移活动的对象实例。

2. Tenured（保持代，也叫 old 区）

Tenure 中存放生命周期长久的实例对象，但是里面的对象也是会被回收掉。

这里，Young 和 Tenured 就组成了 Java 的所谓堆内存，也就是大家经常提到的 heap 内存，或 heap 区。

3. Perm（永久代）

Perm（perm）则是非堆内存的组成部分。主要存放加载的 Class 类级对象如 class 本身，method、field 等等。

那图中标示的 Virtual 区又是干什么的呢?

其实，在 JVM 启动时，就已经保留了固定的内存空间给 Heap 内存，这部分内存并不一定都会被 JVM 使用，但是可以确定的是这部分保留的内存不会被其他进程使用，这部分内存大小由“-Xmx”参数指定。

影响这个的另一个参数是“-Xms”，如果“-Xms”指定的值比“-Xmx”的小，那么两者的差值就是 Virtual 内存值。

随着程序的运行，Eden 区、Tenured 区和 Perm 区会逐渐使用保留的 Virtual 空间。

6.3 GC 概述

什么是 GC 呢？GC 是 Garbage Collection 的缩写，就是垃圾回收机制，在 Java 中开发人员无法使用指针来自由的管理内存，GC 是 JVM 对内存（实际上就是对象）进行管理的方式。

GC 使得 Java 开发人员摆脱了繁琐的内存管理工作，让程序的开发更有效率。

关于 GC 更详细的信息，请参考后续“虚拟机 GC 及其相关问题”章节。

6.4 JVM 中的 ClassLoader

6.4.1 ClassLoader 概述

类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。

与 C 或 C++编写的程序不同，Java 程序是由许多独立的类文件组成的，每个文件对应于一个 Java 类。此外，这些类文件并非立即全部装入内存，而是根据程序需要装入内存的。ClassLoader 便是 JVM 中将类装入内存的那个零件。用户可以定制自己的 ClassLoader。JVM 中缺省的 ClassLoader 可以完成将本地文件系统装入类文件的任务。用户定制的 ClassLoader 可以拥有 JVM 缺省的 ClassLoader 所不具有的功能。例如：用户可以使用自己创建的 ClassLoader 从非本地硬盘或者从网络装入可执行内容。

JVM 在运行时会产生三个 ClassLoader，Bootstrap ClassLoader、Extension ClassLoader、App ClassLoader。BootStrap ClassLoader 使用 C++编写，屏幕上打印它时为 null。它用来在 JVM 启动时自动加载核心类库，运行过程中不能修改 Bootstrap 加载路径。Extension ClassLoader 用来加载 JRE\lib\ext 目录下的库文件，JRE\classes 目录下的类。App ClassLoader 用来加载 CLASSPATH 变量定制路径下的类。

三者的层次关系如下：

```
BootStrap ClassLoader
|------ Extension ClassLoader
          |------- App ClassLoader
```

示例 6-1

ClassLoader 加载类用的是委托模型，加载一个类时，首先 BootStrap 先进行寻找，找不到再由 ExtClassLoader 寻找，最后才是 AppClassLoader。

6.4.2 Java 类加载

类加载器是 Java 语言的基本模块。类加载器是 Java 虚拟机（JVM）的一部分，它会将类加载到内存中；类加载器负责在运行时查找和加载类文件。每个成功的 Java 编程人员都需要了解类加载器及其行为。本部分概述 Java 类加载器。

6.4.2.1 Java 类加载器层次结构

类加载器包含具有父类加载器和子类加载器的层次结构。父类加载器和子类加载器之间的关系类似于超类和子类之间的对象关系。

下面是三种 JVM 提供的类加载器：

- 引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。；
- 扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类；
- 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。

除了引导类加载器之外，所有的类加载器都有一个父类加载器，通过 getParent() 方法可以得到。对于系统提供的类加载器来说，系统类加载器的父类加载器是扩展类加载器，而扩展类加载器的父类加载器是引导类加载器；对于开发人员编写的类加载器来说，其父类加载器是加载此类加载器 Java 类的类加载器。因为类加载器 Java 类如同其它的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加载器的父类加载器是系统类加载器。

注意：在 WebLogic Server 之外的上下文中，BEA 所指的“系统类路径类加载器”通常被称作“应用程序类加载器”。在 WebLogic Server 中讨论类加载器时，BEA 使用术语“系统”，从而与 J2EE 应用程序或库相关的类加载器（BEA 将其称作“应用程序类加载器”）区分开来。

6.4.2.2 加载类

类加载器加载类时由于用到了代理模式，类加载器会首先代理给其它类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器，有可能不是同一个。

真正完成类的加载工作是通过调用 defineClass()来实现的；而启动类的加载过程是通过调用 loadClass() 来实现的。前者称为一个类的定义加载器（defining loader），后者称为初始加载器（initiating loader）。

在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其它类的初始加载器。如类 com.example.Outer 引用了类 com.example.Inner，则由类 com.example.Outer 的定义加载器负责启动类 com.example.Inner 的加载过程。

类加载器在成功加载某个类之后，会把得到的 java.lang.Class 类的实例缓存起来。下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加

载。也就是说，对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass() 方法不会被重复调用。

在 WebLogic Server 中可以对与 Web 应用程序关联的类加载器进行配置，使其首先在本地进行检查，然后再要求其父类加载器提供该类。这样，Web 应用程序能够使用其自己版本的第三方类，即便这些类也可能包含于 WebLogic Server 产品中。

下一章节将对此详细讨论。

6.4.2.3 prefer-web-inf-classes 元素

WebLogic 的专有 Web 应用部署描述符 weblogic.xml 中包含 <prefer-web-inf-classes> 元素（<container-descriptor> 元素的子元素）。默认情况下，该元素设置为 false；将该元素设置为 true 会颠反类加载器委托模型，以便优先加载 Web 应用程序中的类定义，其次才加载更高一级的类加载器中的类定义。这使得 Web 应用程序能够使用自己版本的第三方类，这些类也可能包含于 WebLogic Server 中。

使用此功能时必须谨慎，不要混淆通过 Web 应用程序类定义创建的实例与通过服务器定义创建的实例。如果混淆这些实例，则会引发 ClassCastException。

缺省情况下，类加载器使用 delegation 模型确保已加载的类得到重用。不过，对于 Web 应用程序，可通过将元素 prefer-web-inf-classes 设置为 true；这样 Web 应用程序将使用第三方类的本地版本，而该类可能是 WebLogic Server 的一部分；这意味着，将先加载 Web 应用程序的 WEB-INF 目录类，然后再加载由应用程序类加载器或系统类加载器加载的类。

启用 prefer-web-inf-classes 容易导致类的加载来源发生混淆；比如使用单独加载的类创建类实例，就可能会抛出 ClassCastException，因此切勿混淆使用本地类创建的实例与使用服务器的类创建的实例。

6.4.2.4 更改正在运行的程序中的类

通过 WebLogic Server，您可以在服务器运行期间部署更新版本的应用程序模块，例如 EJB。这个过程即是所谓的热部署或热重新部署，它与类加载紧密相关。

Java 类加载器没有用于取消部署或卸载一组类的标准机制，也不能加载新版本的类。为更新正在运行的虚拟机中的类，必须将已加载该更改类的类加载器替换为新的类加载器。替换类加载器时，必须重新加载通过该类加载器（或该类加载器的任何子类加载器）加载的所有类。这些类的所有实例都必须重新实例化。

在 WebLogic Server 中，每个应用程序都具有类加载器的层次结构，它们是系统类加载器的子类加载器。这些层次结构允许单独重新加载应用程序或部分应用程序，而不会影响系统的其他部分。

WebLogic Server 应用程序类加载中将讨论此主题。

6.4.3 WebLogic ClassLoader

WebLogic 中发布一个应用一般的目录是：

```
myapplication
|---APP-INF       //放在这个目录下的 lib 和 classes 不能实例化 webapp
     |---lib      //放 ejb 和 webapp 公用的 jar 包
     |---classes  //放 ejb 和 webapp 公用的类
|---META-INF
     |---application.xml
|---mywebapp
```

```
        |---WEB-INF
        |---lib
        |---classes //放 class 类
        |---web.xml
  |---ejb.jar       //ejb 的 jar 包
```

示例 6-2

对应的各层级 ClassLoader 如下：

```
BootStrap ClassLoader
|--- Extension ClassLoader
              |---WebLogic Service System ClassLoader
                  |---Filtering ClassLoader
                     |---Application ClassLoader
                        |---Web Application ClassLoader
                           |---Jsp ClassLoader
```

示例 6-3

Application ClassLoader 用来加载 EJB JARS、APP-INF/lib、APP-INF/classes、EJB JARS 中 ClassPath 变量定制的路径下的类。

Web Application ClassLoader 用来加载 WAR、WAR 中 ClassPath 变量定制的路径下的类。

WebLogic 下 ClassLoader 用的也是委托模型，首先 BootStrap 先进行寻找，找不到再由 ExtClassLoader 寻找，然后再由 AppClassLoader 一级一级往下找。这样的分层结构有一个好处，就是在 Jsp，Servlet 中可以直接访问 EJB 的接口。这种上层装载 EJB,下层装载 servlet 等，最下面装载 jsp 文件的结构，使得经常变动的 jsp，servlet 等可以被重新装载而不会涉及到 EJB 层。

在 Weblogic 中可以通过修改配置文件来修改这种加载顺序，在 weblogic.xml 中加入以下代码段：

```
<container-descriptor>
<prefer-web-inf-classess>true</ prefer-web-inf-classess>
</container-descriptor>
```

示例 6-4

上面提到用户为拓展功能可以定制自己的 ClassLoader。Weblogic 中自定义的 ClassLoader 在 Weblogic-application.xml 中描述。

例如：

```
<classloader-struts>
<module-ref>
<module-uri>ejba.jar</module-uri>
</module-ref>
<module-ref>
<module-uri>webc.war</module-uri>
</module-ref>
<classloader-structure>
<module-ref>
<module-uri>weba.war</module-uri>
</module-ref>
```

```
</classloader-structure>
<classloader-structure>
<module-ref>
<module-uri>ejbc.jar</module-uri>
</module-ref>
<module-ref>
<module-uri>webb.war</module-uri>
</module-ref>
<classloader-structure>
<module-ref>
<module-uri>webd.war</module-uri>
</module-ref>
</classloader-structure>
<classloader-structure>
<module-ref>
<module-uri>ejbb.jar</module-uri>
</module-ref>
</classloader-structure>
</classloader-structure>
</classloader-structure>
```

示例 6-5

所定义的 ClassLoader 的层次结构如图：

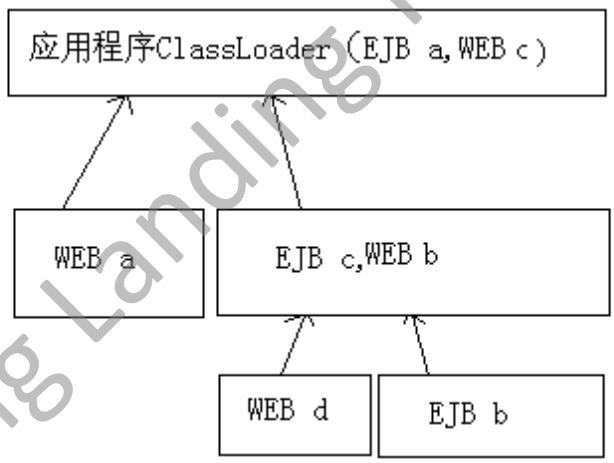


图 6-2

6.4.4 了解 WebLogic Server 对 应用程序类加载的机制

6.4.4.1 WebLogic Server 中应用程序类加载的概述

WebLogic Server 类加载以应用程序的概念为核心。

应用程序通常打包为企业归档（EAR）文件，其中包含应用程序类，EAR 文件中的所有内容均被视为同一个应用程序的组成部分。

下面的内容可以作为 EAR 的一部分，也可以作为独立应用程序进行加载：

- Enterprise JavaBean（EJB）JAR 文件；

- Web 应用程序 WAR 文件；
- 资源适配器 RAR 文件。

如果分别部署 EJB 和 Web 应用程序，会将它们视为两个应用程序。如果将它们一起部署在 EAR 文件中，则它们是一个应用程序。可以将模块共同部署于一个 EAR 文件内，以便系统将它们视为同一个应用程序的各部分。

每个应用程序都有其自己的类加载器层次结构；该层次结构的父级是系统类路径类加载器。这可以隔离应用程序，以使应用程序 A 无法查看应用程序 B 的类加载器或类。在层次结构类加载器中，不存在同级或同伴的概念。应用程序代码只能看到与该应用程序（或模块）关联的类加载器所加载的类，以及应用程序（或模块）类加载器的父类加载器所加载的类。

这允许 WebLogic Server 在同一个 JVM 中承载多个隔离的应用程序。

6.4.4.2 应用程序类加载器层次结构

部署应用程序时，WebLogic Server 自动创建类加载器的层次结构。该层次结构的根类加载器将加载应用程序中的所有 EJB JAR 文件。将针对每个 Web 应用程序 WAR 文件创建子类加载器。

由于 Web 应用程序通常会调用 EJB，所以 WebLogic Server 应用程序类加载器体系结构允许 JavaServer Page（JSP）文件和 servlet 查看其父类加载器中的 EJB 接口。这种体系结构还允许在不重新部署 EJB 层的情况下重新部署 Web 应用程序。实际上，通常会更改 JSP 文件和 Servlet，而不更改 EJB 层。

下图说明此 WebLogic Server 应用程序类加载的概念：

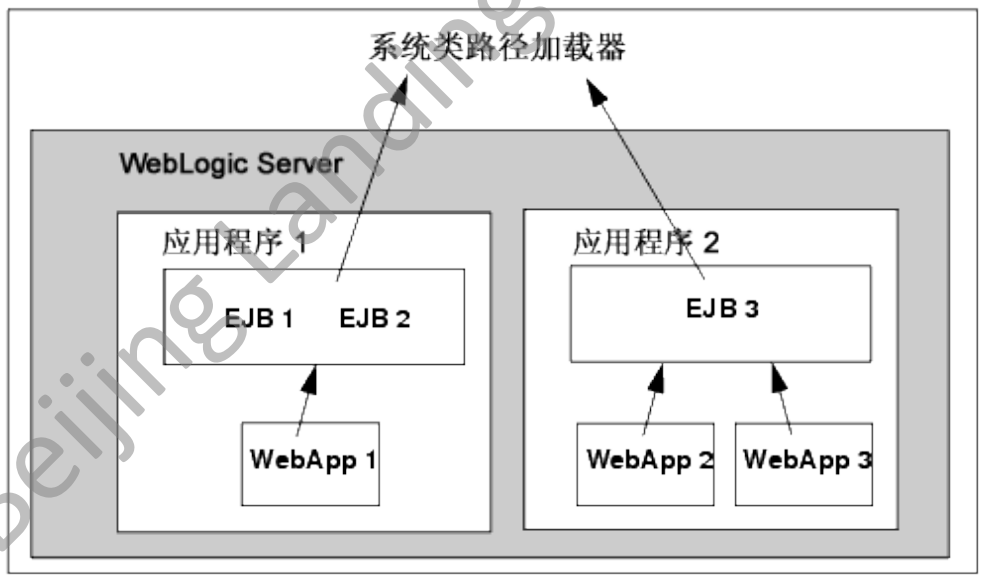


图 6-3

如果您的应用程序包含使用 EJB 的 Servlet 和 JSP：

- 将 Servlet 和 JSP 打包在 WAR 文件中；
- 将 Enterprise JavaBean 打包在 EJB JAR 文件中；
- 将 WAR 和 JAR 文件打包在 EAR 文件中；

- 部署 EAR 文件。

尽管可以分别部署 WAR 和 JAR 文件，但如果将它们共同部署于 EAR 文件内，则会生成允许 Servlet 和 JSP 查找 EJB 类的类加载器安排。如果分别部署 WAR 和 JAR 文件，WebLogic Server 将为它们创建同级类加载器。这表明您必须在 WAR 文件中包含 EJB Home 接口和远程接口，且 WebLogic Server 必须针对 EJB 调用使用 RMI 存根和骨架类，如同 EJB 客户端和实现类位于不同 JVM 中一样。

下一部分应用程序类加载和按值传递或按引用传递中将详细讨论该概念。

注意：Web 应用程序类加载器中包含 Web 应用程序的所有类，但 JSP 类除外。JSP 类包含其自己的类加载器，它是 Web 应用程序类加载器的子类加载器。从而允许分别重新加载各个 JSP。

6.4.4.3 自定义模块类加载器层次结构

可以为应用程序创建自定义类加载器层次结构，从而更好地控制类是否可见以及是否可重新加载。可以通过在 weblogic-application.xml 部署描述符文件中定义 classloader-structure 元素来实现自定义。

下图说明如何 WebLogic 应用程序默认的类加载器组织结构。具有应用程序级类加载器，它可以加载所有 EJB 类。对于每个 Web 模块，都有适用于该模块的类的独立子类加载器（为了简化起见，下图中不说明 JSP 类加载器）。

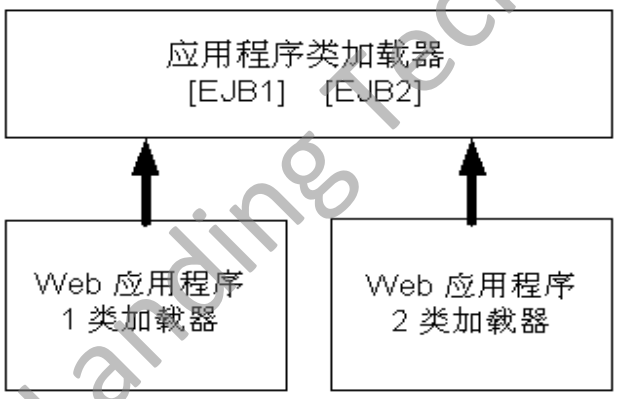


图 6-4 标准类加载器层次结构

该层次结构对于多数应用程序都具有最佳效果，因为在调用 EJB 时，它支持使用 call-by-reference 语义。它还允许独立重新加载 Web 模块，而不影响其他模块。而且，它允许其中一个 Web 模块中运行的代码加载任任 EJB 模块中的类。这很方便，因为 Web 模块不必包含其使用的 EJB 的接口。

注意：严格来说，其中某些优点并不与 J2EE 规范严格符合。

创建自定义模块类加载器的功能提供了声明备用类加载器组织的一种机制，从而支持下列操作：

- 独立地重新加载各个 EJB 模块；
- 重新加载要一起重新加载的模块组；
- 颠倒特定 Web 模块和 EJB 模块之间的父子关系；
- EJB 模块之间的名称空间分隔。

6.4.4.4 声明类加载器层次结构

可以在 WebLogic 特定的应用程序部署描述符 weblogic-application.xml 中声明类加载器层次结构。

该声明的 DTD 如下（清单声明类加载器层次结构）：

```
<!ELEMENT classloader-structure (module-ref*, classloader-structure*)>
<!ELEMENT module-ref (module-uri)>
<!ELEMENT module-uri (#PCDATA)>
```

示例 6-6

weblogic-application.xml 中的顶级元素包含一个可选的 classloader-structure 元素。如果不指定该元素，则将使用标准类加载器。同时，如果某特定模块未包含于定义中，则将按照标准层次结构的定义为其分配一个类加载器。即 EJB 模块与应用程序“根”类加载器关联，而 Web 应用程序模块具有自己的类加载器。

通过 classloader-structure 元素，可以嵌套 classloader-structure 节，以便描述类加载器的任意层次结构，当前限制为三级嵌套。最外层的条目指明应用程序类加载器。对于未列出的模块，将使用标准层次结构。

注意：此定义 scheme 中不包含 JSP 类加载器。JSP 将始终加载至其所属的 Web 模块所关联的类加载器的子类加载器中。

下面是一个类加载器声明（在 weblogic-application.xml 中的 classloader-structure 元素中定义）的示例：

```
<classloader-structure>
   <module-ref>
         <module-uri>ejb1.jar</module-uri>
   </module-ref>
   <module-ref>
         <module-uri>web3.war</module-uri>
   </module-ref>
   <classloader-structure>
         <module-ref>
               <module-uri>web1.war</module-uri>
         </module-ref>
   </classloader-structure>
   <classloader-structure>
         <module-ref>
               <module-uri>ejb3.jar</module-uri>
         </module-ref>
         <module-ref>
               <module-uri>web2.war</module-uri>
         </module-ref>
         <classloader-structure>
               <module-ref>
                     <module-uri>web4.war</module-uri>
               </module-ref>
         </classloader-structure>
         <classloader-structure>
               <module-ref>
                     <module-uri>ejb2.jar</module-uri>
               </module-ref>
```

```
            </classloader-structure>
        </classloader-structure>
    </classloader-structure>
```

示例 6-7 类加载器声明示例

嵌套的组织结构指明类加载器的层次结构，以上各节将形成下图所示的层次结构：

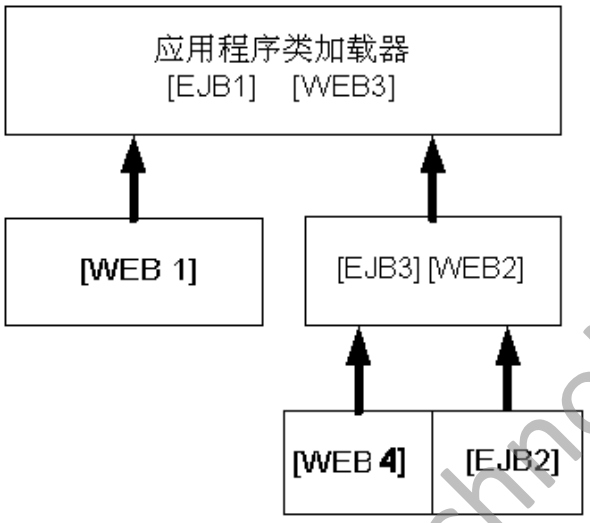


图 6-5 类加载器层次结构示例

6.4.4.5 用户定义类加载器的限制

通过用户定义的类加载器限制，可以更好地控制可重新加载的类，还可以提供模块内的类的可见性。此功能主要供开发人员使用。它适用于迭代开发，但不建议在生产应用中使用此功能的重新加载功能，因为如果更新中包含无效元素，则可能会损坏正在运行的应用程序。

可以在生产中应用名称空间隔离和类可见性的自定义类加载器安排。但程序员应明确知晓，J2EE 规范要求应用程序不依赖于任何给定的类加载器组织。

某些类加载器层次结构会导致应用程序内的模块的行为如同两个独立应用程序中的模块。例如，如果将 EJB 置于其自己的类加载器中，以便对其分别加载，则将获得 call-by-value 语义，而不是 BEA 在标准类加载器层次结构中提供的 call-by-reference 优化。

同时请注意，如果使用自定义层次结构，可能会遇到过期引用而结束。因此，如果重新加载 EJB 模块，也应该重新加载其调用模块。

6.4.4.6 实现类的单个 EJB 类加载器

WebLogic Server 允许重新加载单独的 EJB 模块，而不要求同时重新加载其他模块，也不必重新部署整个 EJB 模块。此功能类似于目前在 WebLogic Server Servlet 容器中重新加载 JSP 的方式。

由于 EJB 类是通过接口调用的，所以可能将单独的 EJB 实现类加载到其自己的类加载器中。这样，可以单独地重新加载这些类，而不必重新部署整个 EJB 模块。下图显示单个 EJB 模块的类加载器层次结构的外观。模块包含两个 EJB（Foo 和 Bar）。

这是前一部分中描述的应用程序一般层次结构的子树：

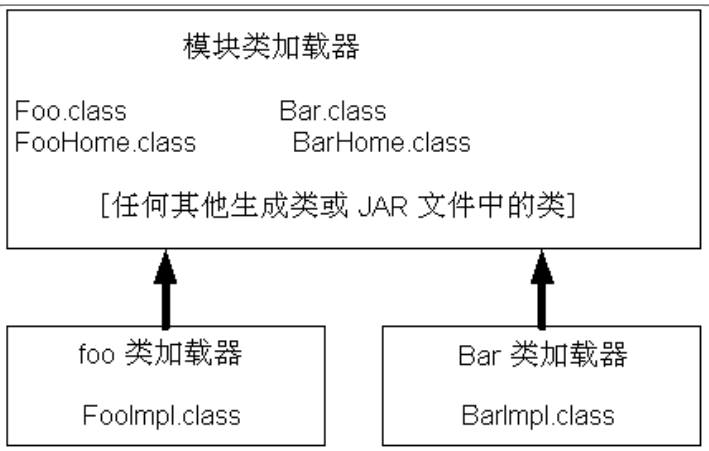


图 6-6 单个 EJB 模块的示例类加载器层次结构

要对文件（相对于所展开应用程序的根）执行部分更新，请使用以下命令行（执行部分文件更新）：

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy myejb/foo.class
```

示例 6-8

在“-redeploy”命令后，提供要更新的文件（相对于已展开的应用程序的根）列表。这可能是特定元素（如上）或模块（或任何一组元素和模块）的路径。

例如（提供要更新的相对文件的列表）：

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy mywar myejb/foo.class anotherejb
```

示例 6-8

给定一组要更新的文件，系统将尝试计算重新部署所需要的最少内容。如果仅重新部署 EJB 实现类，则将仅重新部署该类。如果指定整个 EJB（在上例中为 anotherejb）或如果更改并更新 EJB Home 接口，则必须重新部署整个 EJB 模块。

该重新部署可能会导致其他模块也重新部署，具体取决于类加载器层次结构。特别是，如果其他模块共享 EJB 类加载器，或加载到该 EJB 的类加载器（如 WebLogic Server 标准类加载器模块中）的子类加载器中，则这些模块也将重新加载。

6.4.4.7. 应用程序类加载和按值传递或按引用传递

现代编程语言使用两种常见参数传递模型：“按值传递”和“按引用传递”。通过“按值传递”，将为每个方法调用复制参数和返回值。通过“按引用传递”，将向方法传递实际对象的指针（或引用）。因为“按引用传递”可以避免复制对象，所以它可以提高性能，但它也提供了修改被传递参数状态的一种方法。

WebLogic Server 中包含一种优化方法，能够提高服务器内远程方法接口(RMI)调用的性能。它不使用“按值传递”以及 RMI 子系统的编组和解组工具，而是由服务器通过“按引用传递”直接调用 Java 方法。此机制大大提高了性能，还可用于 EJB 2.0 本地接口。

RMI 调用优化和按引用调用仅适用于调用方和被调用方位于同一个应用程序内的情况。同样，这也与类加载器相关。由于应用程序具有自己的类加载器层次结构，任何应用程序类都会在两个类加载器中定义，如果您尝试在应用程序之间进行分配，则会引发 ClassCastException 错误。为解决这个问题，WebLogic Server 在应用程序之间使用“按值传递”，即使应用程序位于同一个 JVM 中，也会使用该方法。

注意：应用程序之间的调用比同一应用程序内部的调用慢。如果将模块统一部署为一个 EAR 文件，则可以快速调用 RMI，还支持使用 EJB 2.0 本地接口。

6.4.4.8 使用筛选类加载器

在 WebLogic Server 中，系统类路径中的全部.jar 文件由 WebLogic Server 系统类加载器加载。服务器实例中运行的所有应用程序都在应用程序类加载器（系统类加载器的子类加载器）中加载。在系统类加载器的此实现中，应用程序不能使用系统类加载器中已有的不同版本的第三方 jar 包。每个子类加载器都会要求父类加载器（系统类加载器）提供特定的类，但不能加载父类加载器所见的类。

例如，如果$CLASSPATH 以及应用程序 EAR 中都包含名为 com.foo.Baz 的类，则将加载$CLASSPATH 中的类，而不加载 EAR 中的类。由于 weblogic.jar 位于$CLASSPATH 中，应用程序无法替代任何 WebLogic Server 系统的类。

以下部分定义并介绍如何使用筛选类加载器：

- 什么是筛选类加载器；
- 配置 FilteringClassLoader ；
- 资源加载顺序。

6.4.4.8.1 什么是筛选类加载器

FilteringClassLoader 提供一种机制，可用于对部署描述符进行配置，以明确指定：特定的包应始终从应用程序中加载，而不应由系统类加载器加载。这样，可以使用其他版本的应用程序，例如 Xerces 和 Ant。

FilteringClassLoader 位于应用程序类加载器和系统之间。它是系统类加载器的子类加载器，也是应用程序类加载器的父类加载器。FilteringClassLoader 侦听 loadClass(String className) 方法，并将 className 与 weblogic-application.xml 文件中指定的一列包进行比较。如果该包匹配 className，则 FilteringClassLoader 会引发 ClassNotFoundException，该异常通知应用程序类加载器从应用程序中加载类。

6.4.4.8.2 配置 FilteringClassLoader

要配置 FilteringClassLoader，从而指定应从应用程序中加载特定的包，需要向 weblogic-application.xml 文件中添加 prefer-application-packages 描述符元素，该文件详细列出要从应用程序中加载的包。

下例指定 org.apache.log4j.*和 antlr.*包应从应用程序中加载，而不从系统加载器中加载：

```
<prefer-application-packages>
  <package-name>org.apache.log4j.*</package-name>
  <package-name>antlr.*</package-name>
</prefer-application-packages>
```

示例 6-9

6.4.4.8.3 资源加载顺序

资源加载顺序是 java.lang.ClassLoader 的 getResource()方法和 getResources()方法返回资源的顺序。启用筛选时，该顺序与禁用筛选时的顺序稍有不同。启用筛选表示

FilteringClassLoader 中包含一个或多个包模式。不使用任何筛选（默认）时，将按照类加载器树的自上而下的顺序收集资源。

例如，如果 Web(1)请求资源，资源分组的顺序为：系统(3)、应用程序(2)和 Web(1)。

使用系统类加载器：

系统(3) --> 应用程序(2) --> Web (1)

为使说明更明确，给定一个资源 /META-INF/foo.xml，它位于所有类加载器中且会返回下列 URL：

```
META-INF/foo.xml - from the System ClassLoader (3)
META-INF/foo.xml - from the App ClassLoader (2)
META-INF/foo.xml - from the Web ClassLoader (1)
```

示例 6-10

启用筛选时，将返回从 FilteringClassLoader（应用程序类加载器）的子类加载器到该调用类加载器中的资源，然后再返回系统类加载器中的资源。比如，如果同一资源位于所有类加载器(D)、(B)和(A)中，当 Web 类加载器请求该资源时，则将以如下顺序获得这些资源：

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the Web ClassLoader (A)
META-INF/foo.xml - from the System ClassLoader (D)
```

示例 6-11

注意：返回资源时将依据 FilteringClassLoader 下的默认 J2EE 委托模型。只将 FilteringClassLoader 的父类加载器中的资源追加到所返回枚举值的末尾。

使用筛选类加载实现：

系统 (D) --> FilteringClassLoader (filterList := x.y.*) (C)

--> 应用程序(B) --> Web(A)

如果应用程序类加载器请求同一资源，将获得以下顺序：

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the System ClassLoader (D)
```

示例 6-12

getResource() 只返回第一个描述符，getResourceAsStream() 返回第一个资源的 InputStream。

6.4.5 解析模块和应用程序之间的类引用

应用程序可能使用多个不同的 Java 类，包括 EJB、Servlet、JSP、实用程序类以及第三方包。WebLogic Server 通过不同的类加载器部署应用程序，从而维护独立性，并简化动态重新部署和取消部署。因此，您应以正确的方式打包应用程序类，以使每个模块都可以访问其所依赖的类。在某些情况下，可能必须在多个应用程序或模块中包含一组类。本部分描述 WebLogic Server 如何使用多个类加载器，以便您能够成功地临时保存应用程序。

6.4.5.1 关于资源适配器类

现在每个资源适配器都使用其自己的类加载器加载类（类似 Web 应用程序）。结果，诸如 Web 应用程序和 EJB 等与应用程序归档（EAR 文件）中的资源适配器一起打包的模块无法访问资源适配器中的类。如果需要此类访问，则必须将资源适配器类放置在 APP-INF/classes 下。还可以归档这些类（使用 JAR 实用工具），并将它们放置在应用程序归档的 APP-INF/lib 下。

确保 WebLogic Server 系统类路径中不包含任何资源适配器特定的类。如果 Web 模块（例如，EJB 或 Web 应用程序）需要使用资源适配器特定的类，则您必须将这些类绑定在相应模块的归档文件（例如，EJB 的 JAR 文件或 Web 应用程序的 WAR 文件）中。

6.4.5.2 打包共享的实用工具类

WebLogic Server 在 EAR 文件提供了一个存储共享实用工具类的位置。将实用工具 JAR 文件放置在 APP-INF/lib 目录下，将各个类放置在 APP-INF/classes 目录下（不要将 JAR 文件放置在/classes 目录下，或将类放置在/lib 目录下）。这些类将加载到应用程序的根类加载器中。

通过此功能，不必将实用工具类放置在系统类路径中，也不必将类放置在 EJB JAR 文件内（取决于标准 WebLogic Server 类加载器层次结构）。请注意，使用此功能与使用前一部分所述的清单类路径稍有不同。使用此功能，可以在应用程序内共享类定义。而使用清单类路径，只是简单扩展了引用模块的类路径，这表明，每个模块都有各自的类副本。

6.4.5.3 清单类路径

J2EE 规范提供清单类路径条目，模块可以使用该方式指定其需要辅助类的 JAR。仅当您的 EJB JAR 或 WAR 文件中包含其他支持 JAR 文件时，才需使用清单类路径条目。在这种情况下，当您创建 JAR 或 WAR 文件时，您应包含清单文件，其中指明引用所需 JAR 文件的 Class-Path 元素。

下面是简单的清单文件，其中引用 utility.jar 文件：

```
Manifest-Version: 1.0 [CRLF]
Class-Path: utility.jar [CRLF]
```

示例 6-13

在清单文件的第一行，必须始终包含 Manifest-Version 特性，再另起一行（CR | LF |CRLF），然后是 Class-Path 特性。

清单 Class-Path 条目指相对于对这些条目进行定义的当前归档的其他归档。该结构支持多个 WAR 文件和 EJB JAR 文件共享公用库 JAR。例如，如果 WAR 文件中包含的一个清单条目为 y.jar，该条目应位于 WAR 文件旁边（而不是在其中），如下：

```
/<directory>/x.war
/<directory>/y.jars
```

示例 6-14

清单文件自身应位于 META-INF/MANIFEST.MF 下的归档中。

6.4.5.4 在系统类路径中添加 JAR

WebLogic Server 包含一个 lib 子目录，它位于域目录下，可以用于当服务器启动时向 WebLogic Server 系统类路径中添加一个或多个 JAR 文件。lib 子目录用于存放更改频繁的 JAR 文件，服务器中部署的所有或大多数应用程序或 WebLogic Server 本身都需要该子目录。

例如，可以使用 lib 目录来存储域中所有部署都需要的第三方实用工具类。还可以使用它将修补程序应用到 WebLogic Server。

不建议将 lib 目录用于一般用途，例如，在域中部署的一个或两个应用程序之间共享 JAR，或共享需要定期更新的 JAR。如果更新 lib 目录中的 JAR，必须重新引导该域中的所有服务器，应用程序才能实现该更改。如果需要在若干应用程序之间共享 JAR 文件或 J2EE 模块，请使用创建共享 J2EE 库和可选包中描述的 J2EE 库功能。

要使用 lib 库共享 JAR，请执行下列操作：

1、 关闭域中的所有服务器；

2、 将要共享的 JAR 文件复制到域目录的 lib 子目录中。

例如：

```
mkdir c:\bea\weblogic90\samples\domains\wl_server\lib
cp c:\3rdpartyjars\utility.jar
c:\bea\weblogic90\samples\domains\wl_server\lib
```

示例 6-15

注意：启动过程中，WebLogic Server 必须具有读取 lib 目录的权限。

注意：管理服务器不会自动将 lib 目录下的文件复制到远程计算机上的受管服务器上。如果您的受管服务器的物理域目录与管理服务器的不同，则必须将 JAR 文件手工复制到受管服务器计算机的 domain_name/lib 目录上。

启动域中的管理服务器和所有受管服务器。WebLogic Server 将在 lib 目录中发现的 JAR 文件追回到系统类路径中，多个文件将按照字母顺序添加。