支持依赖修复的热部署技术

李海骋 曹 春 吕 军 陶先平

(南京大学计算机软件新技术国家重点实验室 南京 210023) (南京大学计算机系 南京 210023)

摘 要 热部署机制是主流应用服务器的典型功能之一。但是目前的应用服务器仅仅支持独立应用的热部署,并不能满足具有依赖注入的复杂企业级应用在线更新的需求。如果在线更新部分模块,会出现程序调用失效的问题,并会导致整个应用平台的失效。为了解决这个问题,介绍一种支持依赖修复的热部署技术。在首次部署应用的各模块时,用该技术建立模块之间的依赖关系。而在其更新时,通过查找依赖关系,找出受到更新影响的模块,修复依赖并进行局部的热部署,避免重启应用服务器的代价。最后通过实验表明,该热部署技术可以保证依赖注入下的应用正确性;在实际工程应用的场景下,该技术也能够大幅度提升应用服务器的性能和运行效率。

关键词 热部署,依赖修复,应用服务器

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.11.028

Hot Deployment with Dependency Reconstruction

LI Hai-cheng CAO Chun LV Jun TAO Xian-ping

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

(Depart of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract The hot deployment mechanism is a typical feature of mainstream application servers. But current application servers only support hot deployment of standalone applications, which cannot satisfy the requirement of complicated enterprise applications with dependency injection. Failures will occur when some modules are updated online, which will result in failure of the whole application platform. To solve the problem, a technology of hot deployment with dependency reconstruction was introduced. We created dependencies between modules when each module of applications is deployed at the first time. On its updating, we found out modules affected by the update, reconstructed dependency and carried out partial hot deployment, avoiding the cost of restarting application server. Experiments show that our technology of hot deployment can ensure the correctness of the application with dependency injection and operating efficiency of application servers will be highly improved in the scenario of real-world applications.

Keywords Hot deployment, Dependency reconstruction, Application server

1 概述

随着构件化软件开发技术的发展,应用服务器[1]在企业级应用开发中起着不可忽视的作用。在J2EE领域,JBoss[2]是其中发展最为快速的开源应用服务器,并具有程序热部署和热卸载的能力。通过该能力,可以在无需重启应用服务器的情况下,实现部署或更新应用程序的功能。它的优势在于大幅度提升了应用服务器的部署效率,增强了应用软件部署和更新的灵活性。该机制是主流中间件平台和应用服务器的典型功能之一。

然而,热部署的内在表现是对于独立应用的重新加载,其基本假设是被热部署的软件封装是封闭或自包含的,如果存在模块间相互依赖关系则会导致调用失效。例如,热部署机制对处理具有依赖注入[3]关系的模块,在线更新的支持并不好,更新被依赖方模块会导致程序调用失效,影响系统的可靠性[4]。实际应用开发过程中,模块化软件是软件工程中的基本要求,这种存在依赖的软件封装是普遍存在的。即使单个企业级应用也经常会被划分成几个甚至更多的模块,开发人员会将不同的存在依赖的业务组件独立封装,而当这些存在相互依赖关系的模块需要进行更新时,就可能产生依赖关系的丢失,产生程序调用失效的问题。因此,目前的热部署机制并不能满足企业级应用的需求。

本文主要介绍一种支持依赖修复的热部署技术。通过该技术可以在组件(EJB[5])部署时构建依赖关系表,当被依赖

到稿日期:2013-09-16 返修日期:2013-11-09 本文受国家高技术研究发展计划(863计划)(2013AA01A213),国家自然科学基金:集成项目可信软件理论、方法集成与综合试验平台(91318301),国家自然科学基金:环境显示化中的涌现上下文研究(61073031),国家自然科学青年基金:面向网构软件的情境感知和自适应体系结构研究(61100037),江苏省科技支撑项目:基于云计算的智慧城市开发应用平台(BE2012123)资助。

李海骋(1990-),男,硕士生,主要研究方向为软件工程和软件中间件,E-mail:lhc_happy@sina.cn;曹 春(1978-),男,博士,副教授,主要研究方向为软件架构、中间件和软件动态更新技术等;吕 军(1987-),男,硕士,高级工程师;陶先平(1970-),男,博士,教授,博士生导师,主要研究方向为软件Agent技术、软件中间件技术、网构软件方法学等。

方组件更新时查找依赖并自动更新依赖方组件，最终完成依赖的修复，解决程序调用失效的问题。从而，提升了应用服务器在依赖注入下的热部署功能。

本文第 2 节对具有依赖注入的应用在线更新产生的调用失效问题进行了分析；第 3 节介绍了对 JBoss AS5.1 的热部署功能扩展的设计和实现；第 4 节通过几个简单的 EJB 应用，对原版本的 JBoss AS5.1 和扩展后的 JBoss AS5.1 进行实验对比和评价；第 5 节讨论了相关工作；最后总结全文并展望未来工作。

2 问题分析

热部署可以在不重启应用服务器的情况下，部署和更新应用程序，依靠的是它内在的类加载机制。这种类加载机制针对的就是单模块、独立应用的重新部署。对于多模块具有依赖注入关系的应用，在线更新被依赖模块所产生的问题也是由应用服务器的类加载机制所引起的。

2.1 热部署机制与类加载机制

图 1 所示是标准的类加载器架构，是最基本的层次类加载模型。它根据 Java 编程语言和 Java 虚拟机规范，作为实现类加载器的一个范例，在 J2EE 连接器架构上就采用了这种类加载机制。这种类加载器架构包括一个主类加载器，已经加载的类将被放入缓存中。当后续请求到来时，主类加载器会查找缓存中类是否已被加载，并加载没有被加载过的类。然而，当类被重部署时，若可以在缓存中找到原版本中已加载的类，那么就无法加载新版本的类。所以重部署之前，必须对主类加载器进行重部署，销毁原版本的类加载信息。但是，应用服务器中已经部署好的其它应用会因为主类加载器的重部署而丢失本不应该丢失的类加载信息，导致其它已经部署好的应用运行失效。因此图 1 所示的类加载机制不能解决应用的热部署问题。

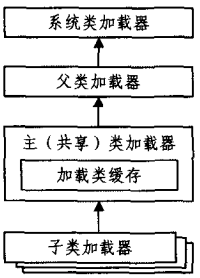

图 1 标准的类加载器架构

为了追求程序的动态性，实现热部署，标准的类加载器架构就会被破坏。图 2 中的类加载器架构提供了共享委托类加载器和类加载器链的方式，有效地解决了图 1 架构下不能热部署的问题。共享委托类加载器将类委托给类加载器链来加载。可以看出在这条链中，每一个模块都有自己的类加载器实例和缓存。当后续请求到来时，只需要查找类加载器链即可。这种机制下，每个模块的模块类加载器都是独立的，重部署类的时候，可直接删除链中相应的模块类加载器，在类加载器链中创建一个新版本的模块类加载器，以保证版本的一致性，完成重部署工作[6]。

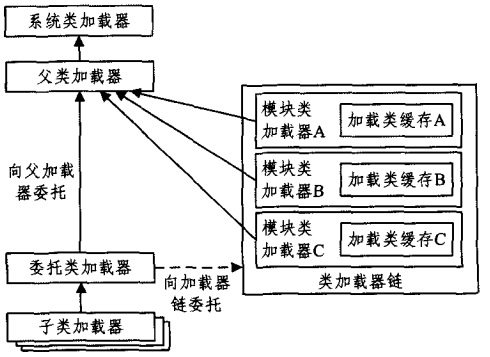

图 2 支持独立应用热部署的类加载器架构

2.2 依赖丢失及调用失效原因

目前的应用服务器一般都具有热部署能力，它们采用的类加载机制也都类似图 2 所示的类加载器架构，以支持应用的热部署和热卸载。如果应用以单模块的形式部署，在线更新时，模块类加载器同样会随着应用的更新而更新，应用的正确性不会受到任何影响。

但是，当存在相互依赖关系的多模块应用需要更新被依赖的模块时，就会产生调用失效的问题。这是因为在图 2 所示的类加载器架构下，类加载器链中的多个模块类加载器会将应用中的各个模块分别进行加载。当被依赖模块在线更新时，它首次部署时的模块类加载器将被销毁，取而代之的是新的模块类加载器。而依赖模块首次部署时的模块类加载器仍然存在，其中所依赖的却是已被销毁的旧版本的信息，表现出来就是依赖的丢失。所以当依赖方模块再次调用被依赖方模块时，一定会导致调用失效。

3 设计与实现

3.1 案例设计

实际应用开发过程中，存在依赖的软件封装是普遍存在的。其具体表现在一个应用的不同业务组件独立封装，并同时部署在应用服务器中。图 3 所示是一个简单的含有依赖注入关系的应用案例。在应用服务器 JBoss AS5.1 中部署这两个 EJB 时，就产生了依赖关系。这个依赖体现在 Session Bean 中的业务执行时，需要对 Entity Bean 中的实体进行操作。那么在线更新有两种情况：1）更新 Session Bean，在这种情况下重部署后，EJB 应用仍然能正常运行；2）更新 Entity Bean，在这种情况下，简单的重新部署会导致应用的异常，原因如 2.2 节所述。

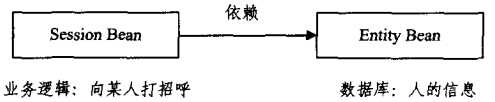

图 3 一个简单的有依赖关系的 EJB 应用

3.2 方案设计

为支持依赖修复的热部署，如果被依赖方进行了重部署，依赖方也必须进行重部署，这即是设计方案的核心思想。3.1 节所描述的应用案例运行在 JBoss AS5.1 上时，我们需要修改 JBoss AS5.1 的源代码，以提升其在依赖注入下在线更新的热部署能力。图 4 所示的是实现这一技术的逻辑框图，主要体现在部署的流程上。我们将部署分为首次部署和重部署两部分，它们原本在部署流程上是没有任何区别的。为了实

现依赖的修复，在被依赖方重部署的同时，重部署依赖方，需要实现两大步骤：(1)首次部署时，通过依赖注入器获得 EJB 之间的依赖关系，并构建反向依赖映射表；(2)当有 EJB 被重新部署，在这个 EJB 部署到最后阶段时，根据反向依赖映射表，找到依赖方 EJB 的路径，并对依赖方进行重新部署，以保证业务的正确性。

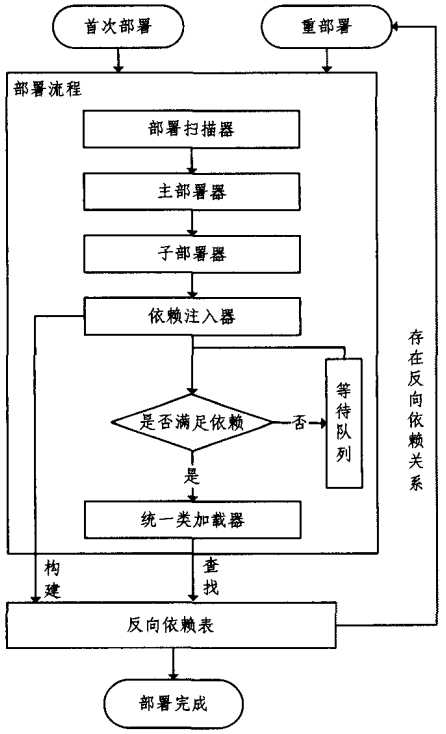

图 4　设计方案逻辑框图

3.3　技术难点

根据问题分析和设计方案，实现中的技术难点主要有两个，分别对应实现的两个步骤：(1)在首次部署时，如何捕获所注入的依赖以便管理；(2)当有 EJB 被重新部署时，根据反向依赖映射表找到其依赖方，在何时可以对依赖方进行重部署。

这两个技术难点源于应用服务器(JBoss AS5.1)本身的复杂性。因为解决这两个问题需要熟悉应用服务器启动运行的整个流程，特别是应用服务器部署 EJB 的流程。而作为能够支持企业级应用的应用服务器，它的内部设计和实现都具有一定的复杂性。图 4 所示的设计方案逻辑框图中包含着应用服务器部署 EJB 阶段的流程。分析应用服务器部署 EJB 阶段的源代码，我们可以更清晰地细化设计方案，解决技术难点。

在部署过程中，*URLDeploymentScanner* 负责热部署的执行，对应图 4 中的部署扫描器。它作为一个 MBean，在 JBoss 启动时被创建。JBoss 启动时，会调用它的 *createService* 方法，最终创建一个 *ScannerThread* 线程。每隔一段时间，该线程就会调用 *URLDeploymentScanner* 的 *scan* 方法。*scan* 方法用来检查 deploy 目录下的变化，并调用主部署器 *MainDeployer* 的相应方法来进行重新部署和销毁等操作。主部署器将为部署单元查找能够部署它的子部署器 *SubDeployer*，它们也都是在 JBoss 启动时创建。每个子部署器都被注册到主部署器中，这样主部署器就能找到它们。例如，EJB 组件对应的子部署器就是 *EJBDeployer*。接着，主部署器将部署单元添加到部署等待队列中。若某些部署单元由于依赖关系不满足，而不能被部署，就将它临时存起来。一旦依赖关系满足，主部署器便从队列中将其取出，并重新部署。然后调用部署单元的 *init* 方法进行初始化[7]。主部署器还要为部署单元创建一个统一类加载器。这个统一类加载器在支持热部署的类加载器架构中表现为模块类加载器，每一个模块都对应一个模块类加载器实例，在 2.1 节已经具体说明。

分析部署流程，在 3.1 节 Session Bean 依赖 Entity Bean 的案例中，Entity Bean 总是先进行部署。在随后部署 Session Bean 的过程中，通过依赖注入器中的 *inject* 方法完成依赖注入，将依赖传递给了对象，也就是说在 Session Bean 的实现中，可以方便地使用 Entity Bean 中的对象。这种依赖注入是一种软件设计模式，除去了硬编码的依赖，让运行时刻改变依赖成为了可能。

3.4　功能实现

根据部署流程的分析，在依赖注入器中注入了 Session Bean 依赖 Entity Bean 的关系，我们通过 JNDI 的方式，构建反向依赖的映射表，利用 HashMap 保存 Session Bean 的路径和 Session Bean 所依赖的所有实体单元的集合。在依赖注入器中，inject 方法可以获得依赖信息，当所有的依赖注入完成后，反向依赖表也全部构建完成。

Session Bean 依赖 Session Bean 的情况与上述 Session Bean 依赖 Entity Bean 的情况类似。同样通过 JNDI 的方式，构建反向依赖映射表，HashMap 保存的是 Session Bean 的路径和所依赖接口名称的集合。

另一方面，*DeployersImpl* 类是部署的主要流程。部署过程存在多个阶段 *stageName*，每个阶段中的部署环境 *deploymentContext* 记录了部署所需要的信息，随着部署阶段一步步地执行下去，它的内容会不断扩充。最后当 Installed 阶段执行后，可以认为这个 Bean 已经部署完成。为实现依赖修复，在我们设计的部署架构下，一旦 Bean 处在 Installed 阶段，就会在该阶段即将结束时，查找在依赖注入器中建立的 JNDI 反向依赖映射表，对反向依赖的 EJB 进行重新部署，修复丢失的依赖关系。在第一次部署时，也会执行查找反向依赖映射表，同样会对反向依赖的 EJB 进行重新部署。但是，这种重复的部署并不影响应用部署的正确性。

4　实验结果与评价

4.1　实验案例

根据设计方案和表 1 所列的实验环境，并考虑到 EJB 之间有 Session Bean 依赖 Entity Bean 和 Session Bean 依赖 Session Bean 的情况，分别写出了两个 EJB 应用的例子进行实验。

表 1　实验配置环境

JDK 操作系统	Ubuntu 12.04
版本	java version "1.6.0_24"
集成开发环境	Eclipse IDE for Java EE Developers
应用服务器版本	JBoss AS 5.1

· greeting 案例

该案例由 3 部分组成：一个 Session Bean，它有两个可调用的接口 *insertPerson* 和 *greeting*，用于实现业务过程；一个 Entity Bean 实现持久化，管理 *person* 对象，每个 *person* 对象有 *name* 和 *age* 两个属性；一个测试用的 client 端。在版本 1

中，我们假定存在 bug，将 *age* 初始化为 100，并且不接受任何的输入，在版本 2 中修正了该错误。案例中 client 端调用 JBoss JNDI 上 *GreeterBean/remote* 的相关方法，对 Session Bean 和 Entity Bean 进行测试，输入〈frj，24〉，最后控制台打印输出“frj(24)：Nice to meet U!”。该案例如 3.1 节中图 3 所示。

· division 案例

该案例由 3 个部分组成：一个 Session Bean 用于实现除法的逻辑过程；另一个 Session Bean 用于检查被除数是否为 0；一个测试用的 client 端。在 client 中输入 A 和 B 两个数，返回 A/B 的结果。在计算 A/B 的结果之前，先检查 B 是否为 0。如果 B 为 0，则返回一个 *EJBException*，在控制台打印提示。该案例如图 5 所示。

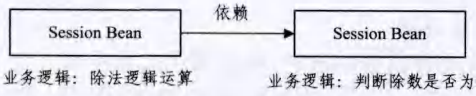

图 5　division 案例

4.2　实验结果

我们以 greeting 案例为例，通过两个版本的 JBoss 做对比实验，旧版本是 JBoss AS 5.1版本，新版本是在 JBoss AS 5.1 的基础上，修改了部分源码，添加了反向依赖查找功能的新 JBoss 环境。执行 java -jar run.jar -c aejb 命令启动 JBoss，aejb 是自定义的环境，EJB 应用将部署在 aejb 文件夹下的 server/deploy 目录下。在旧版本的 JBoss 下，部署 greeting 案例中的 EJB，并在终端运行客户端 GreeterClient.jar，如图 6 所示，显示出打招呼的人名和他的年龄。

```
happy@happy:~/JBOSS/TEST-JAR$ java -jar GreeterClient.jar
frj(100):Nice to meet U!

happy@happy:~/JBOSS/TEST-JAR$
```

图 6　运行 Greeting 案例客户端

旧版本 JBoss 更新被依赖方 Entity Bean，运行终端给出卸载和部署信息，完成 Entity Bean 的重部署。再次运行客户端，出现空指针的错误，如图 7 所示。

```
happy@happy:~/JBOSS/TEST-JAR$ java -jar GreeterClient.jar
Exception in thread "main" javax.ejb.EJBException: java.lang.NullPointerExcepti
n
        at org.jboss.ejb3.tx.Ejb3TxPolicy.handleExceptionInOurTx(Ejb3TxPolicy.j
va:77)
        at org.jboss.aspects.tx.TxPolicy.invokeInOurTx(TxPolicy.java:83)
        at org.jboss.aspects.tx.TxInterceptor$Required.invoke(TxInterceptor.jav
:190)
        at org.jboss.aop.joinpoint.MethodInvocation.invokeNext(MethodInvocation
```

图 7　旧版本 JBoss 更新 EJB 后运行时出错

所出现的错误正如之前的分析，被依赖方更新后，依赖方并不能指向新的被依赖者。而新版本的 JBoss，通过反向依赖表查找依赖并重部署的过程，修复了丢失的依赖关系，从而解决了调用失效问题。图 8 所示为与旧版本 JBoss 相比，在新版本 JBoss 的控制台上，在线更新 Enity Bean 后重部署 Session Bean 的消息输出。

```
18:29:58,650 INFO  [SessionSpecContainer] Starting jboss.j2ee:jar=GreeterSession
.jar,name=GreeterBean,service=EJB3
18:29:58,664 INFO  [EJBContainer] STARTED EJB: org.artemisprojects.aware.aejb.sa
mples.greetperson.session.GreeterBean ejbName: GreeterBean
18:29:58,670 INFO  [JndiSessionRegistrarBase] Binding the following Entries in G
lobal JNDI:

        GreeterBean/remote - EJB3.x Default Remote Business Interface
        GreeterBean/remote-org.artemisprojects.aware.aejb.samples.greetperson.se
ssion.Greeter - EJB3.x Remote Business Interface
```

图 8　新版本 JBoss 更新后的重部署信息

因此使用新版本 JBoss，更新后再次运行客户端，EJB 应用运行正常，显示出更新后打招呼的人的年龄，从 100 更新成了 24，如图 9 所示。

```
happy@happy:~/JBOSS/TEST-JAR$ java -jar GreeterClient.jar
frj(100):Nice to meet U!

happy@happy:~/JBOSS/TEST-JAR$ java -jar GreeterClient.jar
frj(24):Nice to meet U!

happy@happy:~/JBOSS/TEST-JAR$
```

图 9　新版本 JBoss 下应用更新前后的显示

同理，在新版本 JBoss 下 division 案例同样也在更新部分 EJB 后正常运行，保证了应用业务逻辑的正确性。

4.3　实验评价

将 4.1 节中描述的两个案例进行实验，并对比实验结果。在新的部署架构下，构建了反向依赖表的新版本 JBoss 在热部署的性能上更具有优势。它通过对依赖的修复，可以支持依赖注入下的热部署，即对于具有依赖注入关系的 EJB 应用，在线更新被依赖方可以保证应用的正确性。

除了应用在线更新的正确性评价外，对两个版本的应用服务器还需要进行性能评价。这里的性能评价主要体现在应用服务器的启动和部署更新的效率上，以应用服务器的平均启动、平均部署更新时间为评价标准。因为在实际的企业级应用中，不可能是单一的 Session Bean 依赖 Session Bean，或者是 Session Bean 依赖 Entity Bean 的情况，4.1 节的案例只能说明新版本 JBoss 在更新后的应用正确性。评价新版本 JBoss 的性能必须通过具有依赖注入关系，并更接近于企业级的应用案例来说明。

在软件体系结构设计中，分层式结构是最为重要的一种结构。其中 3 层结构最为常见，它从上至下分别为：表示层、业务逻辑层和数据访问层。如图 10 所示，这是一个 3 层结构的 EJB 应用，最下层的 4 个组件就是 Entity Bean，负责数据访问。在实际应用中，由于软件工程自身的复杂性，4 层以上的依赖注入关系是十分少见的。事实上，企业级应用更多的是多业务逻辑组件的两层依赖注入关系的应用，如图 11 所示。

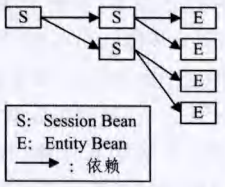

图 10　三层结构的应用案例

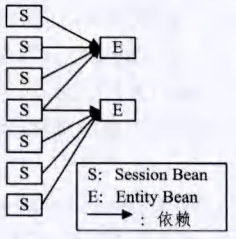

图 11　多业务逻辑组件的应用案例

性能评价中采用的就是图 10、图 11 中所示的应用案例。对这两个应用案例中的 6 个 Entity Bean 和 10 个 Session Bean 分别进行在线更新，并重复 5 次。在新旧两个版本的 JBoss 中检验在线更新后的应用正确性，统计部署和更新所用的平均时间。两个版本应用服务器的性能评价如表 2 所列，其中平均启动时间是指每个版本应用服务器正常启动 10 次的平均所需时间，平均更新时间是指图 10、图 11 案例中的总共 16 个 EJB 分别在线更新 5 次，共计 80 次的平均更新所需时间。

表 2　两个版本 JBoss 的性能评价

应用服务器版本	JBoss AS 5.1	扩展后 JBoss AS 5.1
平均启动时间	57.5s	58.1s
平均更新时间	3.4s	5.6s
更新后正确性	不能保证	可以保证

从性能评价上看，新版本 JBoss 主要针对具有依赖注入关系的 EJB 应用，在线更新可以保证应用的正确性。在运行效率上，新版本 JBoss 的启动时间比旧版本 JBoss 稍有增加，这是由新版本启动在对 EJB 应用的首次部署中，会执行构建依赖表的操作而导致的。在更新 EJB 的过程中，新版本 JBoss 所需时间比旧版本 JBoss 有大幅度增加，实际上这多出来的时间很大一部分是部署依赖方的时间消耗，即根据依赖表，反向查找到依赖方信息，对其进行重部署的时间。尽管如此，新版本整体上增加的时间相比重新启动应用服务器的时间要小得多，因为重启应用服务器过程中，最耗时的是部署服务器自身服务组件的过程。

5　相关工作

尽管目前很多研究工作致力于对热部署的研究，但是大部分工作集中在对分布式异构环境下的热部署[8-11]，其中的很多研究工作都是基于 OGSA(Open Grid Service Architecture，开放网格服务架构)，以服务为中心，解决动态服务创建、管理服务生命周期等问题。同样，在依赖注入方面，大部分工作集中在依赖注入的本身机制和在不同环境下的表现[12-15]，这一方面的工作可以涉及分布式应用、设计模式、软件维护等多个领域。相对而言，我们的工作提出了一种能够支持依赖修复的热部署技术，增强了应用服务器的热部署能力。

利用组件技术提升应用服务器的能力并不是一个新思想。先前许多工作都致力于对应用服务器性能提升的研究，例如在运行时刻动态加载应用服务器所需服务[16]、将应用需求作为一组功能用以重构中间件[17]等。由于 JBoss 是一个可扩展、动态可重构的开源应用服务器[18]，我们基于所提出的热部署技术，扩展并提升了 JBoss AS 5.1 的热部署功能，并进行实验和评价。

结束语　热部署机制是主流中间件平台和应用服务器的典型功能之一。然而热部署自身的局限性，并不能满足具有依赖注入的企业级应用的需求。本文提出了一种支持依赖修复的热部署技术，通过该技术最终解决在线更新后可能出现的程序调用失效问题，提升了应用服务器在依赖注入下的热部署功能。

本文的工作还可以做更好的优化。下一步，将对依赖方的更新操作放置在调用时刻进行，如果没有调用任务，就不会进行耗时的更新操作。更进一步，依赖方的更新操作实质是原版本的重部署，只是为了依赖修复，原版本中的其它信息并没有发生改变，所以重部署中大部分过程都是重复的操作。将来工作中，我们将在代码层面上对依赖进行修复，做到代码级别上的动态更新[19]，以更高效的方式修复依赖，提升应用服务器的热部署能力。

参 考 文 献

[1] Ottinger J. What is an App Server? [EB/OL]. http://www.theserverside.com/news/1363671/What-is-an-App-Server

[2] JBoss Application Server. The JBossGroup[EB/OL]. http://www.jboss.org

[3] Yang H Y, Tempero E, Melton H. An empirical study into use of dependency injection in java[C]// ASWEC 2008. 19th Australian Conference on Software Engineering, 2008. IEEE, 2008: 239-247

[4] Huang G, Wang W, Liu T, et al. Simulation-based analysis of middleware service impact on system reliability: Experiment on Java application server[J]. Journal of Systems and Software, 2011, 84(7): 1160-1170

[5] JSR 220: Enterprise JavaBeansTM 3.0[EB/OL]. http://www.jcp.org/en/jsr/detail? id=220

[6] Thyagarajan S M, Gangadharan B P, Padakandla S. Hot deployment of shared modules in an application server: U.S. Patent 7, 721, 277[P]. 2010-5-18

[7] Olliges S. Runtime Reconfiguration in J2EE Systems[D]. Diplomarbeit, University of Oldenburg, 2005

[8] Florian V, Neagu G, Preda S. An OGSA Compliant Environment for eScience Service Management[C]// 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC). IEEE, 2010: 381-386

[9] Dornemann K, Freisleben B. Discovering Grid Resources and Deploying Grid Services Using Peer-to-Peer Technologies[C]// International Conference on Advanced Information Networking and Applications Workshops, 2009 (WAINA'09). IEEE, 2009: 292-297

[10] Friese T, Smith M, Freisleben B. Hot service deployment in an ad hoc grid environment[C]// Proceedings of the 2nd international conference on Service oriented computing. ACM, 2004: 75-83

[11] Abdellatif T, Kornas J, Stefani J B. Reengineering J2EE servers for automated management in distributed environments[J]. Distributed Systems Online, IEEE, 2007, 8(11): 1

[12] Heinrich M, Grüneberger F J, Springer T, et al. Enriching Web applications with collaboration support using dependency injection[M]// Web Engineering. Springer Berlin Heidelberg, 2012: 473-476

[13] Rajam S, Cortez R, Vazhenin A, et al. Enterprise service bus dependency injection on mvc design patterns[C]// TENCON 2010-2010 IEEE Region 10 Conference. IEEE, 2010: 1015-1020

[14] Fowler, Martin. Inversion of control containers and the dependency injection pattern[EB/OL]. http://martinfowler.com/articles/injection.html

[15] Razina E, Janzen D S. Effects of dependency injection on maintainability[C]// Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA. 2007: 7

[16] Li Y, Zhou M, You C, et al. Enabling on demand deployment of middleware services in componentized middleware[M]// Component-Based Software Engineering. Springer Berlin Heidelberg, 2010: 113-129

[17] Zhang C, Gao D, Jacobsen H A. Towards just-in-time middleware architectures[C]// Proceedings of the 4th international conference on Aspect-oriented software development. ACM, 2005: 63-74

[18] Fleury M, Reverbel F. The JBoss extensible server [C]// Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware. Springer-Verlag New York, Inc., 2003: 344-373

[19] Gu T, Cao C, Xu C, et al. Javelus: A Low Disruptive Approach to Dynamic Software Updates[C]// APSEC. 2012: 527-536