Red Hat JBoss Enterprise Application Platform 7.4

开发指南

针对红帽 JBoss 企业应用平台开发 Jakarta EE 应用程序的说明.

Last Updated: 2024-02-10

Red Hat JBoss Enterprise Application Platform 7.4 开发指南

针对红帽 JBoss 企业应用平台开发 Jakarta EE 应用程序的说明.

法律通告

摘要

本文档提供快速开发安全且可扩展的 Jakarta EE 应用的说明和信息。您将了解如何 设置开发环境，使用 Maven 存储库，并在部署 中进行类加载。本文档还包含有关以下的详细信息： 日志记录 远程 JNDI 查找 在 Web 应用程序中集群 Jakarta 上下文和依赖注入 Jakarta EE APIs, such as jakarta Transactions and jakarta Persistence

目录

提供有关 JBOSS EAP 文档的反馈

要报告错误或改进文档，请登录到 Red Hat JIRA 帐户并提交问题。如果您没有 Red Hat Jira 帐户，则会提示您创建一个帐户。

流程

1. 单击以下链接 **以创建 ticket**。
2. 请包含 **文档 URL、章节编号 并描述问题**。
3. 在 **Summary** 中输入问题的简短描述。
4. 在 **Description** 中提供问题或功能增强的详细描述。包括一个指向文档中问题的 URL。
5. 点 **Submit** 创建问题，并将问题路由到适当的文档团队。

使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 CTO Chris Wright 信息。

第 1 章 开始开发应用程序

1.1. 关于 JAKARTA EE

1.1.1. Jakarta EE 8

JBoss EAP 7 是一款适用于 Jakarta EE Web 配置文件和 Jakarta EE 平台规格的 Jakarta EE 8 兼容实施。

有关 Jakarta EE 8 的详情，请参阅 关于 Jakarta EE。

1.1.2. Jakarta EE 配置文件概述

Jakarta EE 定义不同的配置文件。每个配置集是一个 API 子集，表示适合特定类应用程序的配置。

Jakarta EE 8 定义 Web 配置文件和平台配置文件的规格。产品可以选择以任何组合实施平台、Web 配置文件或一个或多个自定义配置文件。

- Jakarta EE Web 配置文件包含选定的 API 子集，旨在对 Web 应用程序开发很有用。
- Jakarta EE 平台配置文件包括 Jakarta EE 8 Web 配置文件定义的 API，以及用于企业应用程序开发的完整 Jakarta EE 8 API 集。

JBoss EAP 7.4 是兼容 Web 配置文件和完整平台规范的 Jakarta EE 8 实施。

如需 Jakarta EE 8 API 的完整列表，请参阅 Jakarta EE 规范。

1.2. 设置开发环境

1. 下载并安装 Red Hat CodeReady Studio。
   具体步骤，请参阅 Red Hat CodeReady Studio *安装指南*中的使用安装程序独立安装 CodeReady Studio。

2. 在红帽 CodeReady Studio 中设置 JBoss EAP 服务器。
   具体步骤，请参阅 *CodeReady Studio 工具入门指南* 中的通过 IDE 下载、安装和设置 JBoss EAP。

1.3. 在 RED HAT CODEREADY STUDIO 中配置注解处理

在 Eclipse 中，注解处理(AP)默认关闭。如果您的项目生成实施类，这可能会导致 **java.lang.ExceptionInInitializerError** 异常，后跟 ***CLASS_NAME*** **（实施未找到）** 错误消息。

您可以通过以下方法之一解决这些问题：您可以为 单个项目启用注解处理，也可以 为所有 Red Hat CodeReady Studio 项目 全局启用注解处理。

为单**个项目启用注解处理**
若要为特定项目启用注释处理，您必须将值设为 **jdt_ apt_apt的 m2e.apt.activation 属性** 添加到项目的 **pom.xml** 文件中。

```
<properties>
  <m2e.apt.activation>jdt_apt</m2e.apt.activation>
</properties>
```

您可以在 JBoss EAP 附带的 **logging-tools** 和 **kitchensink- ml quickstarts 的 pom.xml** 文件中找到此技术的示例。

在 Red Hat CodeReady Studio 中启用注解处理全局

1. 选择 **Window → Preferences**。
2. 展开 **Maven**，然后选择 **Annotation Processing**。
3. 在 **Select Annotation Processing Mode** 下，选择 **Automatically configure JDT APT（构建速度更快，但结果可能与 Maven 构建不同）**，然后单击 **Apply 和 Close**。

1.4. 配置默认欢迎 WEB 应用程序

JBoss EAP 包含一个默认的 **Welcome** 应用，默认显示在端口 **8080** 的根上下文中。

此默认 **Welcome** 应用可替换为您自己的 Web 应用。这可以通过以下两种方式之一进行配置：

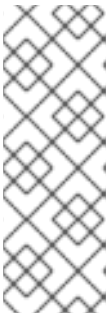
- 更改 **welcome-content** 文件处理程序
- 更改 **default-web-module**

您还可以 禁用欢迎内容。

更改 welcome-content 文件处理程序

1. 修改现有的 **welcome-content** 文件处理程序路径，以指向新部署。

```
/subsystem=undertow/configuration=handler/file=welcome-content:write-attribute(name=path,value="/path/to/content")
```



注意

或者，您也可以创建供服务器的 root 使用的其他文件处理程序。

```
/subsystem=undertow/configuration=handler/file=NEW_FILE_HANDLER:add(path="/path/to/content")
/subsystem=undertow/server=default-server/host=default-host/location=\/:write-attribute(name=handler,value=NEW_FILE_HANDLER)
```

2. 重新加载服务器以使更改生效。

```
reload
```

更改 default-web-module

1. 将部署的 Web 应用映射到服务器的根目录.

```
/subsystem=undertow/server=default-server/host=default-host:write-attribute(name=default-web-module,value=hello.war)
```

2. 重新加载服务器以使更改生效。

```
reload
```

禁用默认欢迎 Web 应用程序

1. 删除 **default-host 的位置** 条目 / 来禁用 welcome 应用。

```
/subsystem=undertow/server=default-server/host=default-host/location=\/:remove
```

2. 重新加载服务器以使更改生效。

```
reload
```

第 2 章 在 JBOSS EAP 中使用 MAVEN

2.1. 了解 MAVEN

2.1.1. 关于 Maven 存储库

Apache Maven 是 Java 应用程序开发中使用的分布式构建自动化工具，用于创建、管理和构建软件项目。Maven 使用名为 Project Object Model 或 POM 文件的标准配置文件来定义项目并管理构建流程。poms 描述模块和组件依赖项，使用 XML 文件描述生成的项目打包和输出的构建顺序和目标。这可确保以正确、一致的方式构建项目。

Maven 使用存储库可实现此目的。Maven 存储库存储 Java 库、插件和其他构建构件。默认公共存储库是 Maven 2 Central Repository，但存储库可以是私有和内部存储库，目标为在开发团队之间共享通用工件。也可从第三方获取存储库。JBoss EAP 包括一个 Maven 存储库，其中包含 Jakarta EE 开发人员通常用于在 JBoss EAP 上构建应用程序的许多要求。若要将项目配置为使用此存储库，请参阅 配置 JBoss EAP Maven 存储库。

有关 Maven 的更多信息，请参阅 Welcome to Apache Maven 。

如需有关 Maven 存储库的更多信息，请参阅 Apache Maven Project - 存储库简介 。

2.1.2. 关于 Maven POM 文件

Project Object Model 或 POM 文件是 Maven 用于构建项目的配置文件。这是一个 XML 文件，其中包含项目的信息以及如何构建项目，包括源位置、测试和目标目录的位置、项目依赖项、插件存储库及其可执行的目标。它还可以包含有关项目的其他详细信息，包括版本、描述、开发人员、邮件列表、许可证等。**pom.xml** 文件需要一些配置选项，并将默认为所有其他选项。

pom.xml 文件的 schema 可以在 http://maven.apache.org/maven-v4_0_0.xsd 找到。

有关 POM 文件的更多信息，请参阅 Apache Maven Project POM 参考 。

Maven POM 文件的最低要求
pom.xml 文件的最低要求如下：

- 项目根目录
- modelVersion
- GroupId - 项目组的 ID
- artifactId - 工件的 ID（项目）
- Version - 指定组下的工件版本

示例：基本 pom.xml 文件

基本的 **pom.xml** 文件可能类似如下：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

2.1.3. 关于 Maven 设置文件

Maven **settings.xml** 文件包含 Maven 的特定用户配置信息。它包含的信息不得通过 **pom.xml** 文件分发，如开发人员身份、代理信息、本地存储库位置，以及特定于用户的其他设置。

有两个位置可以找到 **settings.xml** ：

- **在 Maven 安装中：** 设置文件可以在 **$M2_HOME/conf/** 目录中找到。这些设置称为 **全局** 设置。默认的 Maven 设置文件是可复制的模板，并用作用户设置文件的起点。
- **在用户安装中：** 设置文件可以在 **${user.home}/.m2/** 目录中找到。如果 Maven 和用户 **settings.xml** 文件都存在，则内容将合并。如果存在重叠，用户的 **settings.xml** 文件优先。

示例：Maven 设置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-7.4.0-maven-repository/maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-eap-maven-plugin-repository</id>
          <url>file:///path/to/repo/jboss-eap-7.4.0-maven-repository/maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <!-- Optionally, make the repository active by default -->
    <activeProfile>jboss-eap-maven-repository</activeProfile>
  </activeProfiles>
</settings>
```

settings.xml 文件的 schema 可在 http://maven.apache.org/xsd/settings-1.0.0.xsd 找到。

2.1.4. 关于 Maven Repository Manager

存储库管理器是一种可让您轻松管理 Maven 存储库的工具。存储库管理器在多个方面很有用：

- 它们提供在您的组织和远程 Maven 存储库之间配置代理的功能。这带来了诸多优势，包括更快、更高效的部署，以及更高水平的控制由 Maven 下载的内容。
- 它们为您自己的构件提供部署目的地，允许整个企业的不同开发团队相互协作。

有关 Maven 存储库管理器的更多信息，请参阅 最佳实践 - 使用存储库管理器。

常用的 Maven 存储库管理器

Sonatype Nexus

有关 Nexus 的更多信息，请参阅 Sonatype Nexus 文档 。

Artifactory

有关 Artifactory 的更多信息，请参阅 JFrog Artifactory 文档 。

Apache Archiva

有关 Apache Archiva 的更多信息，请参阅 Apache Archiva：构建工件存储库管理器。

注意

在通常使用存储库管理器的企业环境中，Maven 应当使用此管理器查询所有项目的所有构件。由于 Maven 使用所有声明的存储库来查找缺少的工件，如果无法找到它查找的内容，它将尝试在存储库中 （内置父级 POM 中定义的）中查找它。要覆盖此 **中央** 位置，您可以使用 **central** 添加定义，以便默认存储库 **中央** 现在是您的存储库管理器。这非常适用于已建立的项目，但对于清洁或"新"项目，这会导致问题，因为它会造成问题，因为它会造成对已建立的 **依赖项**。

2.2. 安装 MAVEN 和 JBOSS EAP MAVEN 存储库

2.2.1. 下载和安装 Maven

按照以下步骤下载和安装 Maven：

- 如果您使用 Red Hat CodeReady Studio 构建和部署应用程序，请跳过此步骤。Maven 随 Red Hat CodeReady Studio 一起发布。
- 如果您使用 Maven 命令行构建应用并将其部署到 JBoss EAP，您必须下载并安装 Maven。
  1. 转到 Apache Maven Project - 下载 Maven 并下载您的操作系统的最新发行版。
  2. 如需有关如何为您的操作系统下载和安装 Apache Maven 的信息，请参阅 Maven 文档。

2.2.2. 下载 JBoss EAP Maven 存储库

您可以使用任一方法下载 JBoss EAP Maven 存储库：

- 下载 JBoss EAP Maven 存储库 ZIP 文件 。
- 使用 Offliner 应用下载 JBoss EAP Maven 存储库 。

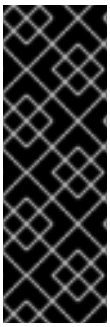
2.2.2.1. 下载 JBoss EAP Maven Repository ZIP 文件

按照以下步骤下载 JBoss EAP Maven 存储库：

1. 在红帽客户门户上登录到 JBoss EAP 下载页面。
2. 在 **Version** 下拉菜单中选择 **7.4**。
3. 在列表中找到 红帽 **JBoss 企业应用平台 7.4 Maven Repository**条目，然后单击 **Download** 以下载包含存储库的 ZIP 文件。
4. 将 ZIP 文件保存到所需的目录中。
5. 提取 ZIP 文件。

2.2.2.2. 使用 Offliner 应用下载 JBoss EAP Maven 存储库

Offliner 应用可作为替代选项下载使用红帽 Maven 存储库开发 JBoss EAP 应用的 Maven 构件。



重要

使用 Offliner 应用下载 JBoss EAP Maven 存储库的过程仅作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

1. 在红帽客户门户上登录到 JBoss EAP 下载页面。
2. 在 **Version** 下拉菜单中选择 **7.4**。
3. 在列表中找到 红帽 **JBoss 企业应用平台 7.4 Maven Repository Offliner Content List**条目，然后单击 **Download**。
4. 将文本文件保存到所需的目录中。

注意

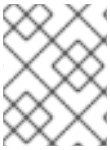
此文件不包含许可证信息。Offliner 应用下载的构件具有与 Maven 存储库 ZIP 文件中指定的相同许可证，该文件随 JBoss EAP 分发。

5. 从 Maven 中央存储库下载 Offliner 应用。
6. 使用以下命令运行 Offliner 应用程序：

```
$ java -jar offliner.jar -r http://repository.redhat.com/ga/ -d DOWNLOAD_FOLDER jboss-eap-7.4.0-maven-repository-content-with-sha256-checksums.txt
```

JBoss EAP Maven 存储库的构件下载到 ***DOWNLOAD_FOLDER*** 目录中。

有关运行 Offliner 应用程序的更多信息，请参阅 Offliner 文档。



注意

生成的 JBoss EAP Maven 存储库将包含与 JBoss EAP Maven 存储库 ZIP 文件中当前提供的相同内容。它不包含 Maven Central 存储库中可用的构件。

2.2.3. 安装 JBoss EAP Maven 存储库

您可以使用在线可用的 JBoss EAP Maven 存储库，或者使用列出的任一方法之一下载并安装到本地：

- 在您的本地文件系统上安装 JBoss EAP Maven 存储库。有关详细信息，请参阅本地 安装 JBoss EAP Maven 存储库。
- 在 Apache Web 服务器上安装 JBoss EAP Maven 存储库。如需更多信息，请参阅 安装 JBoss EAP Maven 存储库以用于 Apache httpd。
- 使用 Nexus Maven 存储库管理器安装 JBoss EAP Maven 存储库。如需更多信息，请参阅使用 Nexus Maven 存储库管理器进行存储库管理。

2.2.3.1. 本地安装 JBoss EAP Maven 存储库

使用此选项将 JBoss EAP Maven 存储库安装到本地文件系统。这易于配置，并允许您在本地计算机上快速启动并运行。

重要

此方法可帮助您熟悉使用 Maven 进行开发，但不建议在团队生产环境中使用。

在下载新的 Maven 存储库之前，先删除位于 **.m2/** 目录下的缓存的 **repository/** 子目录，然后再尝试使用它。

将 JBoss EAP Maven 存储库安装到本地文件系统中：

1. 确保已将 JBoss EAP Maven 存储库 ZIP 文件 下载到本地文件系统。
2. 解压缩您选择的本地文件系统中的 文件。
   这会创建一个新的 **jboss-eap-7.4.0-maven-repository/** 目录，其中包含名为 **maven-repository/** 的子目录中 Maven 存储库。

重要

如果要使用旧的本地存储库，您必须在 Maven **settings.xml** 配置文件中单独配置它。每个本地存储库都必须在自己的 **<repository>** 标签中配置。

2.2.3.2. 安装 JBoss EAP Maven 存储库以用于 Apache httpd

安装用于 Apache httpd 的 JBoss EAP Maven 存储库对于多用户和跨团队开发环境是一个不错的选择，因为任何能够访问 Web 服务器的开发人员也可以访问 Maven 存储库。

在安装 JBoss EAP Maven 存储库之前，您必须首先配置 Apache httpd。具体步骤请查看 Apache HTTP 服务器项目 文档。

1. 确保已将 JBoss EAP Maven 存储库 ZIP 文件 下载到本地文件系统。
2. 将 文件解压缩到可在 Apache 服务器上访问的目录中。

3. 配置 Apache 以允许在创建的目录中浏览读取访问权限和浏览目录。
此配置允许多用户环境访问 Apache httpd 上的 Maven 存储库。

2.3. 使用 MAVEN REPOSITORY

2.3.1. 配置 JBoss EAP Maven 存储库

概述

有两种方法可以指示 Maven 在您的项目中使用 JBoss EAP Maven Repository：

- 您可以在 Maven 全局或用户设置中配置存储库。
- 您可以在项目的 POM 文件中配置存储库。

使用 Maven 设置配置 JBoss EAP Maven 存储库
这是推荐的方法。与共享服务器上的存储库管理器或存储库一起使用的 Maven 设置提供更好的项目的控制和可管理性。设置还支持使用替代镜像将特定存储库的所有查找请求重定向到存储库管理器，而不更改项目文件。有关镜像的详情请参考 http://maven.apache.org/guides/mini/guide-mirror-settings.html。

这种配置方法适用于所有 Maven 项目，只要 POM 文件不包含存储库配置。

本节论述了如何配置 Maven 设置。您可以配置 Maven 安装全局设置或用户的安装设置。

配置 Maven 设置文件

1. 找到您的操作系统的 Maven **settings.xml** 文件。它通常位于 **${user.home}/.m2/** 目录中。
   - 对于 Linux 或 Mac，这是 **~/.m2/**。
   - 对于 Windows，这是 **\Documents 和 Settings\.m2\ 或 \Users\.m2**。
2. 如果您找不到 **settings.xml** 文件，请将 **${user . home}/.m2/conf/ 目录中的 settings**.xml 文件复制到 **${user.home}/.m2/** 目录中。
3. 将以下 XML 复制到 **settings.xml** 文件的 **<profiles> 元素** 中。确定 JBoss EAP 存储库的 URL，并将 ***JBOSS_EAP_REPOSITORY_URL*** 替换为此存储库。

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
 <id>jboss-enterprise-maven-repository</id>
 <repositories>
  <repository>
   <id>jboss-enterprise-maven-repository</id>
   <url>JBOSS_EAP_REPOSITORY_URL</url>
   <releases>
    <enabled>true</enabled>
   </releases>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </repository>
 </repositories>
 <pluginRepositories>
  <pluginRepository>
```

```
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

以下是访问在线 JBoss EAP Maven 存储库的示例配置：

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

4. 将以下 XML 复制到 **settings.xml** 文件的 **<activeProfiles> 元素** 中。

```
<activeProfile>jboss-enterprise-maven-repository</activeProfile>
```

5. 如果在 Red Hat CodeReady Studio 运行时修改 **settings.xml** 文件，您必须刷新用户设置。
   a. 在菜单中，选择 **Window → Preferences**。
   b. 在 **"首选项 "**窗口中，展开 **Maven**，然后选择 **"用户设置 "**。
   c. 单击 **Update Settings** 按钮，以刷新 Red Hat CodeReady Studio 中的 Maven 用户设置。

重要

如果您的 Maven 存储库包含过时的工件，则构建或部署项目时可能会遇到以下 Maven 错误消息之一：

- 缺少工件 *ARTIFACT_NAME*
- [ERROR] 在项目 *PROJECT_NAME* 上执行目标失败; 无法解析 *PROJECT_NAME*的依赖项

要解决这个问题，请删除本地存储库的缓存版本，以强制下载最新的 Maven 工件。缓存的存储库位于此处： **${user.home}/.m2/repository/**

使用 Project POM 配置 JBoss EAP Maven 存储库

警告

您应该避免使用这种配置方法，因为它会覆盖配置的项目的全局和用户 Maven 设置。

如果您决定使用项目 POM 文件配置存储库，您必须仔细规划。Transitively POM 是这类配置的问题，因为 Maven 必须查询外部存储库查找缺少的工件，这会减慢构建过程。它还可能导致您失去对工件来自哪里的控制。

注意

存储库的 URL 将取决于存储库所在的位置：在文件系统或 Web 服务器上。有关如何安装存储库的详情，请参考： 安装 JBoss EAP Maven 存储库。以下是每个安装选项的示例：

文件系统

file:///path/to/repo/jboss-eap-maven-repository

Apache Web Server

http://intranet.acme.com/jboss-eap-maven-repository/

Nexus 存储库管理器

https://intranet.acme.com/nexus/content/repositories/jboss-eap-maven-repository

配置项目的 POM 文件

1. 在文本编辑器中打开项目的 **pom.xml** 文件。
2. 添加以下存储库配置：如果文件中已有 **<repositories>** 配置，请在其中添加 **<repository>** 元素。确保将 **<url>** 改为 实际的存储库位置。

```
<repositories>
  <repository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>JBOSS_EAP_REPOSITORY_URL</url>
```

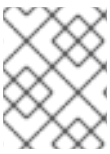
```
    <layout>default</layout>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </snapshots>
  </repository>
</repositories>
```

3. 添加以下插件存储库配置：如果文件中已有 **<pluginRepositories>** 配置，请在其中添加 **<pluginRepository>** 元素。

```
<pluginRepositories>
  <pluginRepository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>JBOSS_EAP_REPOSITORY_URL</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

确定 JBoss EAP 存储库的 URL
存储库 URL 取决于存储库所在的位置。您可以将 Maven 配置为使用以下任一存储库位置：

- 要使用在线 JBoss EAP Maven 存储库，请指定以下 URL ：https://maven.repository.redhat.com/ga/
- 若要使用本地文件系统上安装的 JBoss EAP Maven 存储库，您必须下载存储库，然后使用本地文件路径作为 URL。例如： *file:///path/to/repo/jboss-eap-7.4.0-maven-repository/maven-repository/*
- 如果您在 Apache Web 服务器上安装软件仓库，则存储库 URL 将类似以下示例：*http://intranet.acme.com/jboss-eap-7.4.0-maven-repository/maven-repository/*
- 如果您使用 Nexus Repository Manager 安装 JBoss EAP Maven 存储库，则 URL 类似如下内容：*https://intranet.acme.com/nexus/content/repositories/jboss-eap-7.4.0-maven-repository/maven-repository/*



注意

对于文件服务器上的存储库，可使用常用协议（如 **http://** ）访问远程存储库，如 HTTP 服务器或 **file://** 上的存储库。

2.3.2. 配置 Maven 以用于 Red Hat CodeReady Studio

构建应用程序并将其部署至红帽 JBoss 企业应用平台所需的构件和依赖关系托管在公共存储库中。构建应用程序时，您必须指示 Maven 使用此存储库。如果您计划使用 Red Hat CodeReady Studio 构建和部署应用，本节将介绍配置 Maven 的步骤。

Maven 随 Red Hat CodeReady Studio 一起发布，因此不需要单独安装。但是，您必须配置 Maven，以供 Java EE Web 项目向导使用，以部署到 JBoss EAP。以下流程演示如何通过从 Red Hat CodeReady Studio 编辑 Maven 配置文件来配置 Maven 以用于 JBoss EAP。

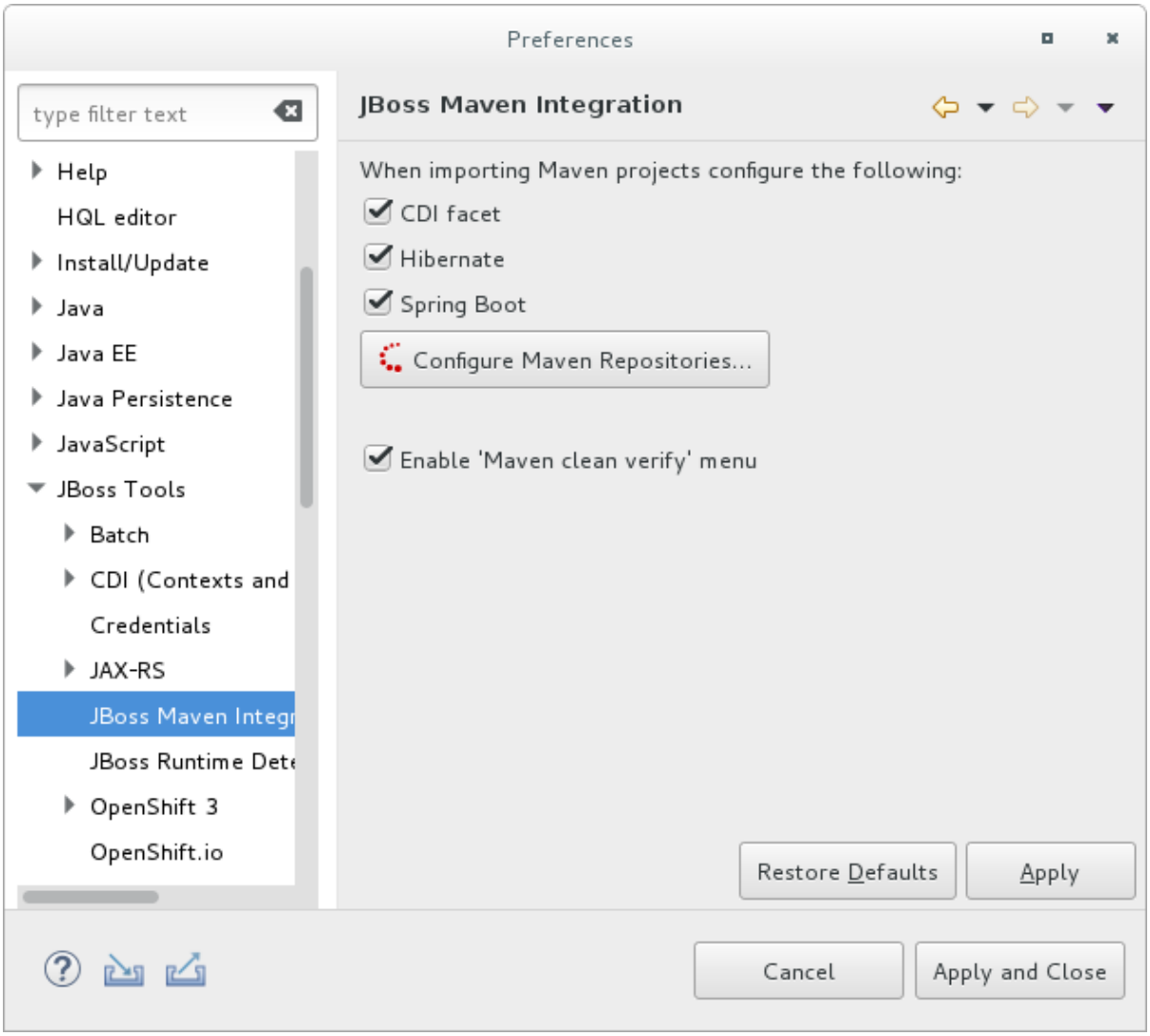
注意

如果您在 Red Hat CodeReady Studio 中将 **Target runtime** 设为 **7.4** 或更新的运行时版本，则您的项目与 Jakarta EE 8 规范兼容。

在 Red Hat CodeReady Studio 中配置 Maven

1. 点击 **Window → Preferences**，展开 **JBoss Tools** 并选择 **JBoss Maven Integration**。

图 2.1. Preferences 窗口中的 JBoss Maven 集成 Pane



2. 单击 **Configure Maven Repositories**。

3. 单击 **Add Repository** 以配置 JBoss 企业 Maven 存储库。按如下所示完成 **Add Maven Repository** 对话框：

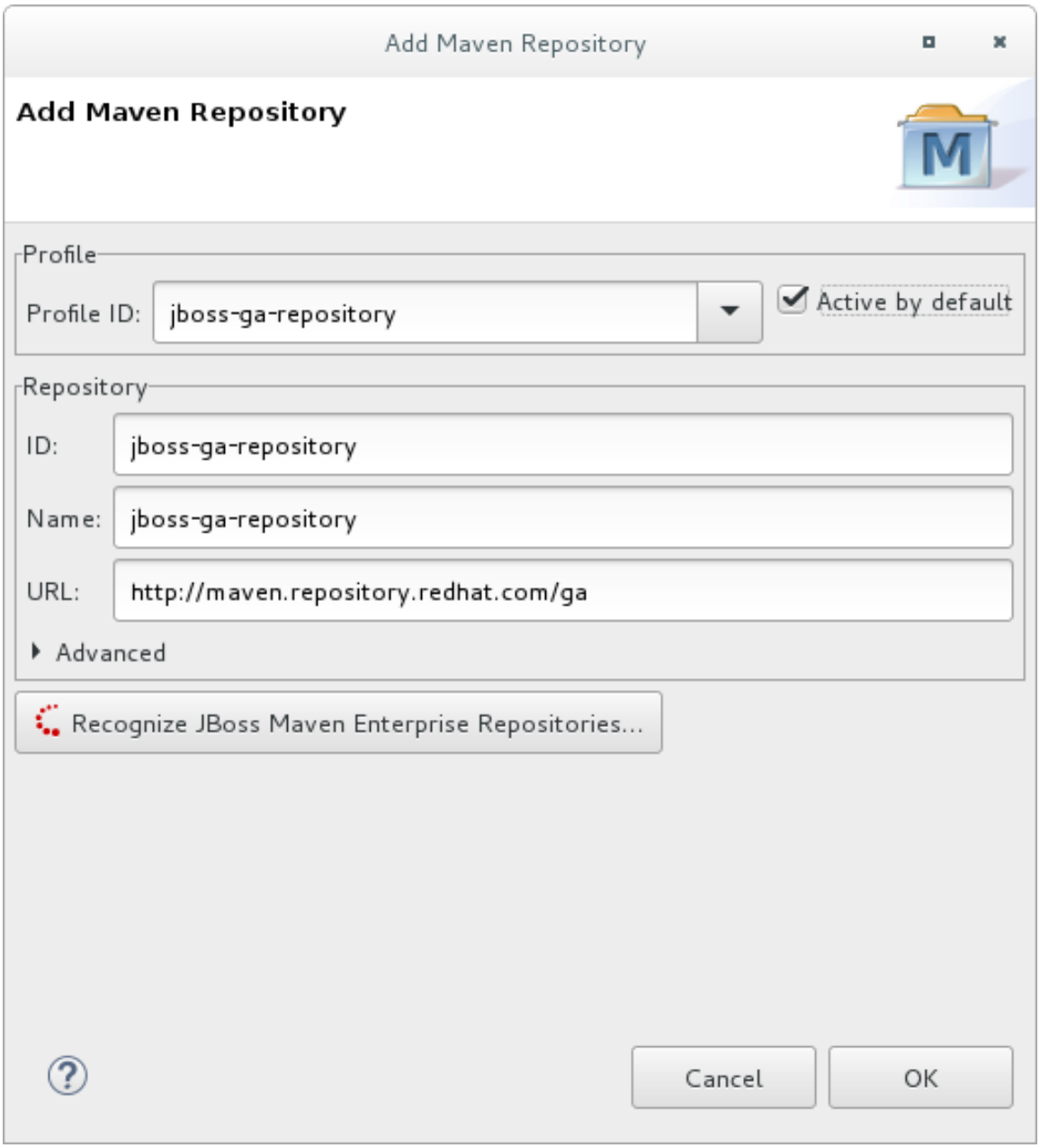
   a. 将 **Profile ID**、**存储库 ID** 和 **Repository Name** 值设置为 **jboss-ga-repository**。

b. 将 **Repository URL** 值设置为 **http://maven.repository.redhat.com/ga**

c. 单击 **默认活动** 复选框，以启用 Maven 存储库。

d. 点击 **OK**。

图 2.2. 添加 Maven Repository



4. 检查存储库并点击 **Finish**。

5. 系统将提示您显示消息 ***"是否确定要更新文件 MAVEN_HOME/settings.xml?"***。单击 **Yes** 以更新设置。单击 **确定** 关闭该对话框。

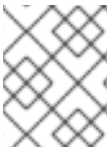
JBoss EAP Maven 存储库现在已配置为与红帽 CodeReady Studio 搭配使用。

2.3.3. 管理项目依赖项

这部分论述了在红帽 JBoss 企业应用平台上对材料(BOM)POM 的使用情况。

BOM 是一个 Maven **pom.xml** (POM)文件，用于指定模块的所有运行时依赖项的版本。版本依赖项列在文件的依赖项管理部分中。

项目使用 BOM，方法是将其 **groupId:artifactId:version** (GAV)添加到项目的 **pom.xml** 文件的依赖项管理部分，并指定 **<scope>import</scope>** 和 **<type>pom</type>** 元素值。



注意

在很多情况下，POM 项目文件中的依赖关系使用 **提供** 的范围。这是因为这些类是在运行时由应用服务器提供的，而且不需要将它们与用户应用打包。

支持的 Maven Artifacts
作为产品构建流程的一部分，JBoss EAP 的所有运行时组件都是从受控环境中的源代码构建的。这有助于确保二进制工件不包含任何恶意代码，并且它们在产品的生命周期内得到支持。这些工件可以通过 **-redhat** 版本限定符（如 **1.0.0-redhat-1** ）轻松识别。

将受支持的构件添加到构建配置 **pom.xml** 文件可确保构建使用正确的二进制构件来进行本地构建和测试。请注意，带有 **-redhat** 版本的构件不一定是受支持的公共 API 的一部分，将来的修订可能会改变。有关公共支持的 API 的详情，请查看发行版中包含的 Javadoc 文档。

例如，要使用受支持的 Hibernate 版本，请在构建配置中添加类似如下的内容：

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.3.1.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

请注意，上例中包含 **<version/>** 字段的值。不过，建议您使用 Maven 依赖项管理来配置依赖项版本。

依赖项管理
Maven 包括管理整个构建过程中直接和传递依赖关系版本的机制。有关使用依赖项管理的一般信息，请参阅 Apache Maven 项目： 依赖机制简介。

在您的构建中直接使用一个或多个支持的红帽依赖关系不能保证所有构建的传递依赖项都会被完全支持。Maven 构建通常使用来自 Maven 中央存储库和其他 Maven 存储库的构件源组合。

JBoss EAP Maven 存储库中包含了依赖性管理 BOM，用于指定所有支持的 JBoss EAP 二进制构件。此 BOM 可以在构建中使用，以确保 Maven 将受支持的 JBoss EAP 依赖项的优先级放在构建中的所有直接和传递依赖关系上。换句话说，传输依赖关系将管理到正确的受支持的依赖版本（若适用）。此 BOM 的版本与 JBoss EAP 版本匹配。

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>eap-runtime-artifacts</artifactId>
      <version>7.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
```

```
    ...
  </dependencies>
</dependencyManagement>
```

注意

在 JBoss EAP 7 中，此 BOM 的名称已从 **eap6 支持的工件改**为 **eap -runtime-artifacts**。此更改的目的是更清楚，此 POM 中的工件是 JBoss EAP 运行时的一部分，但不一定是受支持的公共 API 的一部分。些 JAR 包含内部 API 和功能，它们可能会在版本间有所变化。

JBoss EAP Jakarta EE Specs BOM
jboss-jakartaee-8.0 BOM 包含 JBoss EAP 使用的 Jakarta EE 规范 API JAR。

要在项目中使用此 BOM，请首先在 POM 文件的 **dependencyManagement** 部分中添加 **jboss-jakartaee-8.0** BOM 依赖项，为 **groupId** 指定 **org.jboss.spec**，然后添加应用所需特定 API 的依赖项。这些依赖项不需要版本，并使用 **提供** 的范围，因为 **jboss-jakartaee-8.0** BOM 中包含了这些 API。

以下示例使用 **jboss-jakartaee-8.0** BOM 的 **1.0.0.Alpha1** 版本添加 Servlet 和 Jakarta 服务器页面 API 的依赖项。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-jakartaee-8.0</artifactId>
      <version>1.0.0.Alpha1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet.jsp</groupId>
    <artifactId>jboss-jsp-api_2.3_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```

注意

JBoss EAP 为大部分产品组件的 API 打包并提供 BOM。许多此类 BOM 都方便地打包到一个更大的 **jboss-eap-jakartaee8 BOM** 中，组**Id** 为 **org.jboss.bom**。**jboss-jakartaee-8.0** BOM（组为 **org. jboss.spec** ）包含在这个较大的 BOM 中。这意味着，如果您使用在此 BOM 中打包的其他 JBoss EAP 依赖项，您只需将一个 **jboss-eap-jakartaee8 BOM BOM** 添加到项目的 POM 文件中，而不是单独添加 **jboss-jakartaee-8.0** 和其他 BOM 依赖项。

JBoss EAP BOMs 可用于应用程序开发
下表列出了可用于应用程序开发的 Maven BOMs。

表 2.1. JBoss BOMs

BOM Artifact ID	使用案例
eap-runtime-artifacts	支持的 JBoss EAP 运行时构件.
jboss-eap-jakartaee8	支持的 JBoss EAP Jakarta EE 8 API 外加其他 JBoss EAP API JAR.
jboss-eap-jakartaee8-with-tools	**jboss-eap-jakartaee8** 加上 Arquillian 等开发工具.

注意

JBoss EAP 6 中的这些 BOM 合并到更少的 BOM 中，以简化大多数用例的使用。Hibernate、日志记录、事务、消息传递和其他公共 API JAR 现在包含在 **jboss-eap-jakartaee8** BOM 中，而不是为每个情况要求单独的 BOM。

以下示例使用 **jboss-eap-jakartaee8** BOM 的 **7.4.0.GA** 版本。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```

JBoss EAP 客户端 BOM
客户端 BOMs 不创建依赖项管理部分或定义依赖项。相反，它们是其他 BOM 的聚合，用于打包远程客户端用例所需的一组依赖项。

wildfly-ejb-client-bom、**wildfly-jms-client -bom** 和 **wildfly-jaxws-client-bom** 由 **jboss-eap-jakartaee8** BOM 管理，因此您无需管理项目依赖项中的版本。

以下是如何将 **wildfly-ejb-client-bom**、**wildfly- jms-client-bom** 和 **wildfly-jaxws-client-bom** 依赖项添加到项目的示例：

```
<dependencyManagement>
  <dependencies>
    <!-- JBoss stack of the Jakarta EE APIs and related components.  -->
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-ejb-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-jms-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-jaxws-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  ...
</dependencies>
```

有关 Maven 依赖项和 BOM POM 文件的更多信息，请参阅 Apache Maven Project - 依赖机制简介。

第 3 章 类加载和模块

3.1. 简介

3.1.1. 类加载和模块概述

JBoss EAP 使用模块化类加载系统来控制已部署应用的类路径。这个系统比传统的分层类加载器系统提供更多灵活性和控制力。开发人员对其应用可用的类有精细的控制，并且可以配置部署来忽略应用服务器提供的类，而非自有的类。

模块类加载器将所有 Java 类划分为称为模块的逻辑组。每个模块可以定义对其他模块的依赖关系，以便将该模块中的类添加到其自己的类路径中。由于每个部署的 JAR 和 WAR 文件都被视为模块，因此开发人员可以通过在其应用中添加模块配置来控制其应用的类路径的内容。

3.1.2. 在 Deployment 中加载类

为了进行类加载，JBoss EAP 将所有部署视为模块。它们称为动态模块。根据部署类型，类加载行为会有所不同。

WAR 部署

WAR 部署被视为单个模块。**WEB-INF/lib** 目录中的类被视为与 **WEB-INF/classes** 目录中的类相同。WAR 中打包的所有类将加载使用相同的类加载器。

EAR 部署

EAR 部署由多个模块组成，由以下规则定义：

1. EAR 的 **lib/** 目录是名为父模块的单个模块。
2. EAR 中的每个 WAR 部署都是一个模块。
3. EAR 内的 Jakarta Enterprise Beans JAR 部署均是一个模块。

Subdeployment 模块（如 EAR 中的 WAR 和 JAR 部署）对父模块具有自动依赖关系。但是，它们之间没有自动依赖关系。这称为子部署隔离，可以为每个部署或整个应用服务器禁用。

子部署模块之间的显式依赖关系可以通过与任何其他模块相同的方法添加。

3.1.3. 类加载优先级

JBoss EAP 模块类加载程序使用优先级系统来防止类加载冲突。

在部署期间，会为每个部署及其各个依赖项创建完整的软件包和类列表。该列表根据类加载优先级规则排序。在运行时加载类时，类加载程序会搜索此列表，并加载第一个匹配项。这可以防止部署类路径中的相同类和软件包的多个副本相互冲突。

类加载器按照以下顺序加载从高到低的顺序：

1. **隐式依赖项：** 这些依赖关系由 JBoss EAP 自动添加，如 Jakarta EE API。这些依赖项具有最高的类加载器优先级，因为它们包含 JBoss EAP 提供的常见功能和 API。
   如需了解有关每个隐式 依赖项的完整详情，请参阅 Implicit 模块 依赖项。
2. **明确的依赖关系：** 这些依赖关系使用应用的 **MANIFEST.MF** 文件或新的可选 JBoss 部署描述符 **jboss-deployment-structure.xml** 文件手动添加到应用配置中。

请参阅 为部署添加 Explicit 模块 依赖关系以了解如何添加显式依赖项。

3. **本地资源：** 这些是打包在部署本身中的类文件，如 **WEB-INF/classes** 或 **WEB-INF/lib** 目录。
4. 部署 **间依赖关系：** 它们是 EAR 部署中其他部署的依赖关系。这可以包括在 EAR 的 **lib** 目录中，或者在其他 Jakarta Enterprise Beans jars 中定义的类中。

3.1.4. jboss-deployment-structure.xml

jboss-deployment-structure.xml 文件是 JBoss EAP 的可选部署描述符。此部署描述符可对部署中的类加载进行控制。

此部署描述符的 XML 架构位于 ***EAP_HOME*/docs/schema/jboss-deployment-structure-1_2.xsd** 下的产品安装目录中。

可使用此部署描述符执行的关键任务有：

- 定义显式模块依赖项.
- 防止特定隐式依赖项加载.
- 从该部署的资源定义其他模块.
- 更改 EAR 部署中的子部署隔离行为.
- 向 EAR 中的模块添加其他资源根.

3.2. 为部署添加 EXPLICIT 模块依赖性

可以在应用中添加显式模块依赖项，将这些模块的类添加到部署时应用的类路径。

注意

JBoss EAP 自动为部署添加一些依赖性。详情请参阅 Implicit 模块依赖项。

先决条件

1. 您要添加模块依赖项的工作软件项目。
2. 您必须知道添加为依赖项的模块的名称。如需 JBoss EAP 随附的静态模块列表，请参阅 包含的模块。如果模块是另一部署，请参阅 JBoss EAP *配置指南* 中的 Dynamic Module Naming，以确定模块名称。

可以使用两种方法配置依赖关系：

- 向部署的 **MANIFEST.MF** 文件中添加条目。
- 向 **jboss-deployment-structure.xml** 部署描述符中添加条目。

添加依赖配置到 MANIFEST.MF
可以将 Maven 项目配置为在 **MANIFEST.MF** 文件中创建所需的依赖项条目。

1. 如果项目没有，请创建名为 **MANIFEST.MF** 的文件。对于 Web 应用(WAR)，将此文件添加到 **META-INF/** 目录中。对于 Jakarta Enterprise Beans 存档(JAR)，将其添加到 **META-INF/** 目录中。

2. 使用逗号分隔的依赖模块名称列表在 **MANIFEST.MF** 文件中添加依赖项条目：

```
Dependencies: org.javassist, org.apache.velocity, org.antlr
```

- 要使依赖项可选，请在依赖项条目中的模块名称中附加 **可选** ：

```
Dependencies: org.javassist optional, org.apache.velocity
```

- 通过将导出附加到依赖项条目中的模块名称，可以 **导出** 依赖项：

```
Dependencies: org.javassist, org.apache.velocity export
```

- 当模块依赖项包含在注解扫描期间处理的注解时，如声明 Jakarta Enterprise Beans 拦截器时，需要 **annotations** 标志。否则，就无法在部署中使用模块中声明的 Jakarta Enterprise Beans 拦截器。当还需要注解扫描时，还有其他情况涉及注解扫描。

```
Dependencies: org.javassist, test.module annotations
```

- 默认情况下，无法访问依赖项的 **META-INF** 中的项目。**服务** 依赖项使来自 **META-INF/services** 的项目可以访问，以便可以加载模块中的 **服务**。

```
Dependencies: org.javassist, org.hibernate services
```

- 若要扫描 **beans.xml** 文件并将其生成的 bean 用于应用程序，可以使用 **meta-inf** 依赖项。

```
Dependencies: org.javassist, test.module meta-inf
```

添加依赖配置到 jboss-deployment-structure.xml

1. 如果应用没有，请创建名为 **jboss-deployment-structure.xml** 的新文件，并将它添加到项目中。此文件是一个 XML 文件，它的根元素为 **<jboss-deployment-structure>**。

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

对于 Web 应用(WAR)，将此文件添加到 **WEB-INF/** 目录中。对于 Jakarta Enterprise Beans 存档(JAR)，将其添加到 **META-INF/** 目录中。

2. 在文档根目录和 **< dependencies> 元素内创建一个 <deployment >** 元素。

3. 在 **<dependencies>** 节点中，为每个模块依赖项添加一个模块元素。将 **name** 属性设置为模块的名称。

```
<module name="org.javassist" />
```

- 若要使依赖项成为可选的，可以添加 **optional** 属性到模块条目，值设为 **true**。此属性的默认值为 **false**。

```
<module name="org.javassist" optional="true" />
```

- 可以通过将 **export** 属性添加到值为 **true** 的模块条目来导出依赖项。此属性的默认值为 **false**。

```
<module name="org.javassist" export="true" />
```

- 当模块依赖项包含需要在注解扫描过程中处理的注解时，将使用 **annotations** 标记。

```
<module name="test.module" annotations="true" />
```

- **服务** 依赖项指定是否以及如何使用此依赖项中找到 **的服务**。默认值为 **none**。指定此属性的 **导入** 值等同于在导入过滤器列表末尾添加一个过滤器，其中包括 dependencies 模块的 **META-INF/services** 路径。为此属性设置 export 值等同于 **导出** 过滤器列表上的相同操作。

```
<module name="org.hibernate" services="import" />
```

- **META-INF** 依赖项指定是否以及如何使用此依赖项中的 **META-INF** 条目。默认值为 **none**。为此属性指定 **导入** 值等同于在导入过滤器列表末尾添加一个过滤器，其中包括 dependencies 模块的 **META-INF/**** 路径。为此属性设置 export 值等同于 **导出** 过滤器列表上的相同操作。

```
<module name="test.module" meta-inf="import" />
```

示例：带有两个依赖项的 **jboss-deployment-structure.xml** 文件

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.javassist" />
      <module name="org.apache.velocity" export="true" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

JBoss EAP 在部署时将指定模块中的类添加到应用的类路径中。

创建 Jandex Index

annotations 标志要求模块包含 Jandex 索引。在 JBoss EAP 7.4 中，这会自动生成。但是，出于性能的原因，仍建议手动添加索引，因为自动扫描可能是消耗 CPU 并增加部署时间的较长进程。

要手动添加索引，请创建新的"index JAR"以添加到模块中。使用 Jandex JAR 构建索引，然后将它插入到新的 JAR 文件中。在当前的实施中，当索引添加到模块内的 JAR 文件中时，根本不执行扫描。

创建 Jandex 索引：

1. 创建索引：

```
java -jar modules/system/layers/base/org/jboss/jandex/main/jandex-jandex-2.0.0.Final-redhat-1.jar $JAR_FILE
```

2. 创建临时工作空间：

```
mkdir /tmp/META-INF
```

3. 将索引文件移到工作目录

```
mv $JAR_FILE.ifx /tmp/META-INF/jandex.idx
```

a. 选项 1：在新 JAR 文件中包括索引

```
jar cf index.jar -C /tmp META-INF/jandex.idx
```

然后，将 JAR 放置到模块目录中，再编辑 **module.xml** 以将它添加到资源根目录中。

b. 选项 2：将索引添加到现有 JAR

```
java -jar /modules/org/jboss/jandex/main/jandex-1.0.3.Final-redhat-1.jar -m $JAR_FILE
```

4. 告知模块导入以使用注解索引，以便注解扫描可以找到注解。

a. 选项 1：如果您要使用 **MANIFEST.MF** 添加模块依赖项，请在模块名称后添加 **注解**。例如，更改：

```
Dependencies: test.module, other.module
```

为

```
Dependencies: test.module annotations, other.module
```

b. 选项 2：如果您要根据模块依赖关系添加使用 **jboss-deployment-structure.xml** 添加 **注释 ="true"** 的模块依赖关系。

注意

如果应用想要使用静态模块内类中定义的带注解的 Jakarta EE 组件，则需要注释索引。在 JBoss EAP 7.4 中，会自动生成静态模块的注解索引，因此您无需创建它们。但是，您必须告知模块导入以使用注释，方法是将依赖项添加到 **MANIFEST.MF** 或 **jboss-deployment-structure.xml** 文件。

3.3. 使用 MAVEN 生成 MANIFEST.MF 条目

使用 Maven JAR、Jakarta Enterprise Beans 或 WAR 包装插件的 Maven 项目可以生成 **MANIFEST.MF** 文件，其中包含 **依赖性** 条目。这不会自动生成依赖项列表，而是使用 **pom .xml 中指定的详细信息创建 MANIFEST. MF** 文件。

在使用 Maven 生成 **MANIFEST.MF** 条目前，您需要：

- 一个有效的 Maven 项目，它使用 JAR、Jkarta Enterprise Beans 或 WAR 插件（**maven-jar-plugin**、**maven-ejb-plugin** 或 **maven-war-plugin**）。
- 您必须知道项目的模块依赖项的名称。有关 JBoss EAP 随附 的静态模块列表，请参阅包含的模块。如果模块是另一部署，则参考 JBoss EAP *配置指南中* 的动态模块命名 来确定模块名称。

生成 MANIFEST.MF 文件包含模块依赖项

1. 将以下配置添加到项目的 **pom.xml** 文件中的打包插件配置：

```
<configuration>
  <archive>
    <manifestEntries>
      <Dependencies></Dependencies>
```

```
      </manifestEntries>
    </archive>
</configuration>
```

2. 将模块依赖项列表添加到 **<Dependencies>** 元素中。使用 **MANIFEST.MF** 文件中添加依赖项时使用的相同格式：

```
<Dependencies>org.javassist, org.apache.velocity</Dependencies>
```

可选 和 **导出** 属性也可以在这里使用：

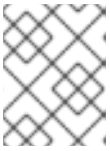
```
<Dependencies>org.javassist optional, org.apache.velocity export</Dependencies>
```

3. 使用 Maven 装配目标构建项目：

```
[Localhost ]$ mvn assembly:single
```

使用 assemble 目标构建项目时，最终的存档包含包含指定模块依赖项的 **MANIFEST.MF** 文件。

示例： **pom.xml 中的**配置模块依赖项



注意

此处的示例显示了 WAR 插件，但它也与 JAR 和 Jakarta Enterprise Beans 插件（maven-jar-plugin 和 maven-ejb-plugin）兼容。

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <configuration>
      <archive>
        <manifestEntries>
          <Dependencies>org.javassist, org.apache.velocity</Dependencies>
        </manifestEntries>
      </archive>
    </configuration>
  </plugin>
</plugins>
```

3.4. 防止一个模块被加载

您可以配置可部署应用，以防止加载隐式依赖项。当应用包含与应用服务器作为隐式依赖项提供的库或框架的不同版本时，这非常有用。

先决条件

- 您要将隐式依赖项排除在其中的工作软件项目。
- 您必须知道要排除的模块的名称。如需隐式 依赖项及其条件的列表，请参阅 Implicit 模块 依赖项列表。

将依赖项排除配置添加到 jboss-deployment-structure.xml

1. 如果应用没有，请创建名为 **jboss-deployment-structure.xml** 的新文件，并将它添加到项目中。这是一个 XML 文件，它的根元素为 **<jboss-deployment-structure>**。

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

   对于 Web 应用(WAR)，将此文件添加到 **WEB-INF/** 目录中。对于 Jakarta Enterprise Beans 存档(JAR)，将其添加到 **META-INF/** 目录中。

2. 在文档根目录和 **< exclusions> 元素内创建一个 <deployment >** 元素。

```
<deployment>
  <exclusions>

  </exclusions>
</deployment>
```

3. 在 excludes 元素中，为每个要排除的模块添加 **<module> 元素**。将 **name** 属性设置为模块的名称。

```
<module name="org.javassist" />
```

   示例：扩展两个模块

```
<jboss-deployment-structure>
  <deployment>
    <exclusions>
      <module name="org.javassist" />
      <module name="org.dom4j" />
    </exclusions>
  </deployment>
</jboss-deployment-structure>
```

3.5. 从 DEPLOYMENT 中排除子系统

排除子系统具有与移除子系统相同的效果，但它仅适用于单个部署。您可以通过编辑 **jboss-deployment-structure.xml** 配置文件，从部署中排除子系统。

排除子系统

1. 编辑 **jboss-deployment-structure.xml** 文件。

2. 在 **<deployment>** 标签中添加以下 XML:

```
<exclude-subsystems>
  <subsystem name="SUBSYSTEM_NAME" />
</exclude-subsystems>
```

3. 保存 **jboss-deployment-structure.xml** 文件。

该子系统的部署单元处理器将不再在部署上运行。

示例： **jboss-deployment-structure.xml** 文件

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jaxrs" />
    </exclude-subsystems>
    <exclusions>
      <module name="org.javassist" />
    </exclusions>
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <resources>
      <resource-root path="my-library.jar" />
    </resources>
  </deployment>
  <sub-deployment name="myapp.war">
    <dependencies>
      <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <local-last value="true" />
  </sub-deployment>
  <module name="deployment.myjavassist" >
    <resources>
     <resource-root path="javassist.jar" >
       <filter>
         <exclude path="javassist/util/proxy" />
       </filter>
     </resource-root>
    </resources>
  </module>
  <module name="deployment.javassist.proxy" >
    <dependencies>
      <module name="org.javassist" >
        <imports>
          <include path="javassist/util/proxy" />
          <exclude path="/**" />
        </imports>
      </module>
    </dependencies>
  </module>
</jboss-deployment-structure>
```

3.6. 在部署中使用类加载器编程

3.6.1. 在部署中以编程方式载入类和资源

您可以以编程方式查找或加载应用程序代码中的类和资源。您选择的方法取决于多个因素。本节介绍可用的方法，并提供了有关何时使用它们的指导。

使用 Class.forName（）方法加载类

您可以使用 **Class.forName（）** 方法以编程方式加载和初始化类。这个方法有两个签名：

- **class.forName(String className)** ：
  此签名仅取一个参数，即您需要加载的类的名称。通过此方法签名，当前类的类加载程序将加载类并默认初始化新加载的类。

- **class.forName(String className, boolean initial, ClassLoader loader loader)** ：
  此签名需要三个参数：类名称、布尔值，指定是否初始化类，以及加载该类的 **ClassLoader**。

建议使用三个参数签名来以编程方式加载类。通过此签名，您可以控制您是否希望在加载时初始化目标类。获取和提供类加载器也更高效，因为 JVM 不需要检查调用堆栈来确定要使用的类加载程序。假设包含代码的类名为 **CurrentClass**，您可以使用 CurrentClass **.class.getClassLoader（）** 方法获取类 loader。

以下示例提供了类加载程序来加载并初始化 **TargetClass** 类：

```
Class<?> targetClass = Class.forName("com.myorg.util.TargetClass", true,
CurrentClass.class.getClassLoader());
```

使用给定名称查找所有资源

如果您知道资源的名称和路径，直接加载资源的最佳方法是使用标准 Java 开发套件(JDK) 类**或 Class Loader** API。

- 加载单个资源。
  若要加载位于与部署中的类或其他类相同的目录中的单个资源，您可以使用 **Class.getResourceAsStream（）** 方法。

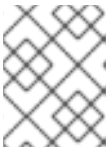
  ```
  InputStream inputStream =
  CurrentClass.class.getResourceAsStream("targetResourceName");
  ```

- 加载单个资源的所有实例。
  要加载对您部署的类 loader 可见的单个资源的所有实例，请使用 **Class.getClassLoader（）.getResources(String resourceName)** 方法，其中 **resourceName** 是资源的完全限定路径。此方法为类加载器使用给定名称访问的资源返回所有 **URL** 对象的枚举。然后，您可以使用 **openStream（）** 方法迭代 URL 数组以打开各个流。

  下例加载资源的所有实例，并迭代结果。

  ```
  Enumeration<URL> urls =
  CurrentClass.class.getClassLoader().getResources("full/path/to/resource");
  while (urls.hasMoreElements()) {
      URL url = urls.nextElement();
      InputStream inputStream = null;
      try {
          inputStream = url.openStream();
          // Process the inputStream
          ...
      } catch(IOException ioException) {
          // Handle the error
      } finally {
          if (inputStream != null) {
              try {
                  inputStream.close();
  ```

```
            } catch (Exception e) {
                // ignore
            }
        }
    }
}
```



注意

由于 URL 实例是从本地存储加载的，因此不需要使用 **openConnection（）** 或其他相关方法。流更易于使用并最小化代码的复杂性。

- 从类加载器加载类文件。
  如果已经载入了一个类，您可以使用以下语法载入与该类对应的类文件：

```
InputStream inputStream =
CurrentClass.class.getResourceAsStream(TargetClass.class.getSimpleName() + ".class");
```

如果还没有载入该类，则必须使用类加载器并转换路径：

```
String className = "com.myorg.util.TargetClass"
InputStream inputStream =
CurrentClass.class.getClassLoader().getResourceAsStream(className.replace('.', '/') +
".class");
```

3.6.2. 编程化部署中的资源

JBoss 模块库为迭代所有部署资源提供了多个 API。JBoss 模块 API 的 Java 文档位于以下位置 ：http://docs.jboss.org/jbossmodules/1.3.0.Final/api/.要使用这些 API，您必须在 **MANIFEST.MF** 中添加以下依赖项：

```
Dependencies: org.jboss.modules
```

务必要注意，尽管这些 API 提供了更大的灵活性，但它们的运行也比直接路径查找慢得多。

本节介绍了您可以编程化地迭代应用程序代码中的资源的一些方法。

- 列出部署中和所有导入中的资源。
  有时无法通过确切的路径查找资源。例如，确切的路径可能未知，或者您可能需要检查给定路径中的多个文件。在本例中，JBoss 模块库为迭代所有部署资源提供了多个 API。您可以使用以下两种方法之一迭代部署中的资源：
  - 迭代单一模块中找到的所有资源。
    ModuleClassLoader.iterateResources（） 方法迭代此模块类加载程序中的所有资源。此方法采用两个参数：要搜索的起始目录名和一个布尔值，指定它是否应递归到子目录中。

    以下示例演示了如何获取 ModuleClassLoader 并获取 **bin/** 目录中资源的迭代器，并递归到子目录。

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources = moduleClassLoader.iterateResources("bin",true);
```

生成的迭代器可用于检查每个匹配资源并查询其名称和大小（如果可用），打开可读流，或获取资源的 URL。

- 迭代单一模块和导入的资源中找到的所有资源。
  Module.iterateResources（） 方法迭代此模块类加载程序中的所有资源，包括导入模块中的资源。此方法返回的集合比之前的方法大得多。这个方法需要一个参数，它是将结果缩小到特定模式的过滤器。或者，也可以提供 PathFilters.acceptAll（）来返回整个集合。

  以下示例演示了如何在此模块中查找整个资源集合，包括导入。

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.acceptAll());
```

- 查找与某一模式匹配的所有资源。
  如果您只需要在部署中或部署的完整导入集中找到特定资源，则需要过滤资源迭代。JBoss 模块过滤 API 为您提供了多个工具来实现这一目标：

  - 检查完整的依赖项。
    如果您需要检查完整的依赖项集合，您可以使用 **Module.iterateResources（）** 方法的 **PathFilter** 参数检查每个资源的匹配名称。

  - 检查部署依赖项。
    如果您只需要在部署中查看，请使用 **ModuleClassLoader.iterateResources（）** 方法。但是，您必须使用其他方法过滤生成的迭代器。**PathFilters.filtered（）** 方法可以提供本例中资源迭代器的过滤视图。**PathFilters** 类包含许多静态方法，用于创建和编写可执行各种功能的过滤器，包括查找子路径或精确匹配项或匹配 Ant 风格的"glob"模式。

- 过滤资源的其他代码示例。
  以下示例演示了如何根据不同的条件过滤资源。

示例：查找所有文件命名 的消息.properties（ 在您的部署中）

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/messages.properties"),
moduleClassLoader.iterateResources("", true));
```

示例：在您的部署和导入中查找所有文件命名 的消息.properties

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.match("**/message.properties"));
```

示例：查找任何在您的部署中命名 my-resources 的所有文件

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources = PathFilters.filtered(PathFilters.match("**/my-
```

```
resources/**"), moduleClassLoader.iterateResources("", true));
```

示例：查找部署和导入中 的所有文件命名 的消息 或错误

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.any(PathFilters.match("**/messages"),
PathFilters.match("**/errors"));
```

示例：查找部署中特定软件包中的所有文件

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("path/form/of/packagename", false);
```

3.7. 类加载和子部署

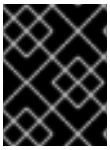
3.7.1. 企业存档中的模块和类加载

企业存档(EAR)不作为单一模块加载，如 JAR 或 WAR 部署。它们作为多个唯一模块载入。

以下规则决定了 EAR 中存在的模块：

- EAR 存档根目录中的 **lib/** 目录的内容是一个模块。这称为父模块。
- 每个 WAR 和 Jakarta Enterprise Beans JAR 子部署都是一个模块。这些模块具有与任何其他模块相同的行为，以及对父模块的隐式依赖项。
- Subdeployments 对父模块和其他任何非 WAR 子部署具有隐式依赖项。

存在对非 WAR 子部署的隐式依赖关系，因为 JBoss EAP 默认禁用子部署类加载器隔离。无论子部署类加载器隔离如何，对父模块的依赖关系都会保留。



重要

没有子部署获得对 WAR 子部署的隐式依赖性。任何子部署都可以像任何其他模块一样，为另一个子部署配置明确的依赖关系。

如果需要严格兼容性，则可启用 Subdeployment 类加载器隔离。这可以在单个 EAR 部署或所有 EAR 部署中启用。Jakarta EE 规范建议，便携式应用不应依赖于子部署相互访问，除非在每个子部署的 **MANIFEST.MF** 文件中明确声明依赖项为 **Class-Path** 条目。

3.7.2. 子部署类加载器隔离

企业存档(EAR)中的每个子部署是一个具有自己的类加载器的动态模块。默认情况下，子部署可以访问其他子部署的资源。

如果子部署不允许访问其他子部署的资源，则可以启用严格的子部署隔离。

3.7.3. 在 EAR 中启用子部署类加载器隔离

此任务演示了如何在 EAR 部署中使用特殊的部署描述符在 EAR 部署中启用子部署类加载器隔离。这不需要对应用服务器进行任何更改，也不会影响任何其他部署。

重要

即使子部署类加载器隔离已被禁用，也无法添加 WAR 部署作为依赖项。

1. 添加部署描述符文件。
   将 **jboss-deployment-structure.xml** 部署描述符文件添加到 EAR 的 **META-INF** 目录中（如果它尚不存在），并添加以下内容：

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

2. 添加 **<ear-subdeployments-isolated>** 元素。
   如果 jboss- **deployment-structure.xml 文件的内容不存在，请将 <ear -subdeployments-isolated> 元素 添加到 jboss-deployment-structure.xml** 文件中 **。**

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

现在，为这个 EAR 部署启用了 Subdeployment 类装载程序隔离。这意味着，EAR 的子部署不会对每个非 WAR 子部署具有自动依赖项。

3.7.4. 在企业归档中配置子部署之间的会话共享

JBoss EAP 提供配置企业存档(EAR)在 EAR 中包含的 WAR 模块子部署之间共享会话的功能。默认情况下禁用此功能，必须在 EAR 的 **META-INF/jboss-all.xml** 文件中明确启用。

重要

由于此功能不是标准 servlet 功能，如果启用此功能，您的应用程序可能无法移植。

要在 EAR 中启用 WAR 之间的会话共享，您需要在 EAR 的 **META -INF/jboss-all.xml 中声明 shared-session-config** 元素：

示例： META-INF/jboss-all.xml

```
<jboss xmlns="urn:jboss:1.0">
  ...
  <shared-session-config xmlns="urn:jboss:shared-session-config:2.0">
  </shared-session-config>
  ...
</jboss>
```

shared-session-config 元素用于为 EAR 中的所有 WAR 配置共享会话管理器。如果存在 **shared-session-config** 元素，则 EAR 内的所有 WAR 将共享相同的会话管理器。此处所做的更改将影响 EAR 中包含的所有 WAR。

3.7.4.1. 共享会话配置选项参考

示例： META-INF/jboss-all.xml

```
<jboss xmlns="urn:jboss:1.0">
   <shared-session-config xmlns="urn:jboss:shared-session-config:2.0">
    <distributable/>
      <max-active-sessions>10</max-active-sessions>
      <session-config>
         <session-timeout>0</session-timeout>
         <cookie-config>
            <name>JSESSIONID</name>
            <domain>domainName</domain>
            <path>/cookiePath</path>
            <comment>cookie comment</comment>
            <http-only>true</http-only>
            <secure>true</secure>
            <max-age>-1</max-age>
         </cookie-config>
         <tracking-mode>COOKIE</tracking-mode>
      </session-config>
      <replication-config>
         <cache-name>web</cache-name>
         <replication-granularity>SESSION</replication-granularity>
      </replication-config>
   </shared-session-config>
</jboss>
```

元素	描述
shared-session-config	共享会话配置的根元素.如果 **META-INF/jboss-all.xml** 中存在，则 EAR 中包含的所有部署 WAR 都将共享单个会话管理器。
distributable	指定应使用可分布式会话管理器。从 schema 版本 2.0 开始，默认使用一个不可分发的会话管理器。对于版本 1.0，distributable 会话管理器仍然是默认的会话管理器。
max-active-sessions	允许的最大会话数.
session-config	包含 EAR 中包含的所有已部署 WAR 的会话配置参数。
session-timeout	为 EAR 中包含的部署的 WAR 中创建的所有会话超时间隔定义默认的会话超时间隔。指定的超时必须以整分钟表示。如果超时为 *0* 或以下，容器可确保会话的默认行为永不超时。如果没有指定此元素，容器必须设置其默认超时期限。
cookie-config	包含由 EAR 中包含的部署的 WAR 创建的会话跟踪 Cookie 的配置。
name	分配给 EAR 中所含部署的 WAR 创建的任何会话跟踪 Cookie 的名称。默认值为 **JSESSIONID**。

元素	描述
domain	分配给 EAR 中所含部署的 WAR 创建的任何会话跟踪 Cookie 的域名。
路径	分配给 EAR 中所含部署的 WAR 创建的任何会话跟踪 Cookie 的路径。
注释	分配给 EAR 中所含部署的 WAR 创建的任何会话跟踪 Cookie 的注释。
http-only	指定 EAR 中包含的已部署 WAR 创建的任何会话跟踪 Cookie 是否标记为 **HttpOnly**。
安全	指定任何由 EAR 中包含的部署的 WAR 创建的会话跟踪 Cookie 是否会被标记为安全，即使发起对应会话的请求使用普通 HTTP 而不是 HTTPS。
max-age	分配给由 EAR 中包含的部署的 WAR 创建的任何会话跟踪 Cookie 的生命周期（以秒为单位）。默认值为 **-1**。
tracking-mode	定义由 EAR 中包含的部署的 WAR 创建的会话的跟踪模式。
replication-config	包含 HTTP 会话集群配置。
cache-name	这个选项仅适用于集群。它指定 Infinispan 容器的名称，以及用于存储会话数据的缓存。默认值（如果未明确设置）由应用服务器决定。要在缓存容器内使用特定的缓存，请使用 **container.cache** 形式，如 **web.dist**。如果 name 不限定，则使用指定容器的默认缓存。
复制 -granularity	这个选项仅适用于集群。它决定了会话复制粒度级别。可能的值有 **SESSION** 和 **ATTRIBUTE**，其中 **SESSION** 是默认值。 如果使用了 **SESSION** 粒度，则在请求范围内修改了任何会话属性，则会复制所有会话属性。如果对象引用由多个会话属性共享，则需要此策略。但是，如果会话属性足够大和/或不常修改，则这可能会低效，因为无论是否修改了所有属性，都必须复制所有属性。 如果使用 **ATTRIBUTE** 粒度，则仅复制在请求范围内修改的属性。如果对象引用由多个会话属性共享，则此策略不合适。如果会话属性足够大和/不频繁地修改，则这比 **SESSION** 粒度更高。

3.8. 在自定义模块中部署标签库描述符(TLD)

如果您有多个应用使用通用标签库描述符(TLD)，请将 TLD 与应用分隔开，使它们位于一个中央和唯一的位置。这实现了对 TLD 的轻松添加和更新，无需更新使用它们的每一个应用。

这可以通过创建自定义 JBoss EAP 模块来实现，该模块包含 TLD JAR，并在应用中声明对该模块的依赖关系。如需更多信息，请参阅 模块和依赖项。

注意

确保至少有一个 JAR 包含 TLD，并且 TLD 打包在 **META-INF** 中。

在自定义模块中部署 TLD

1. 使用管理 CLI 连接您的 JBoss EAP 实例，再执行以下命令来创建包含 TLD JAR 的自定义模块：

```
module add --name=MyTagLibs --resources=/path/to/TLDarchive.jar
```

重要

使用 **模块** 管理 CLI 命令添加和删除模块，仅作为技术预览提供。此命令不适合在受管域中使用，或在远程连接管理 CLI 时使用。在生产环境中，应当手动添加和删除模块。如需更多信息，请参阅 手动创建自定义模块 并手动 删除 JBoss EAP *配置指南中* 的自定义模块部分。

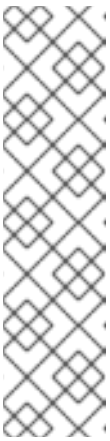
技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

如果 TLD 与需要依赖项的类打包，请使用 **--dependencies** 选项确保在创建自定义模块时指定这些依赖项。

在创建模块时，您可以通过使用特定于文件系统的分隔符来指定多个 JAR 资源。

- Linux - **：**示例, **--resources=<path-to-jar>:<path-to-another-jar>**
- 对于 Windows - **;**.示例, **--resources=<path-to-jar>;<path-to-another-jar>**



注意

--resources

除非使用 **--module-xml,** 否则此模块是必需的。它列出了文件系统路径（通常 JAR 文件），它由特定于文件系统的路径分隔符分隔，如 **java.io.File.pathSeparatorChar**。指定的文件将复制到创建的模块的 目录中。

--resource-delimiter

它是 user-defined 路径分隔符，用于 resources 参数。如果存在此参数，则命令解析器将在此处使用值，而不使用特定于文件系统的路径分隔符。这允许在跨平台脚本中使用 **模块** 命令。

2. 在您的应用中，使用 Add a Explicit Module 依赖项到 Deployment 中所述的方法声明新的 MyTagLibs 自定义模块的 依赖关系。

重要

在声明依赖项时，确保您也导入 **META-INF**。例如，**MANIFEST.MF**:

```
Dependencies: com.MyTagLibs meta-inf
```

或者，对于 **jboss-deployment-structure.xml**，请使用 **meta-inf** 属性。

3.9. 按部署显示模块

按部署显示模块

您可以使用 **list-modules** 管理操作来显示每个部署的模块列表。

```
:list-modules
```

示例：为单机服务器按部署显示模块

```
/deployment=ejb-in-ear.ear:list-modules
```

```
/deployment=ejb-in-ear.ear/subdeployment=ejb-in-ear-web.war:list-modules
```

示例：显示受管域的部署模块

```
/host=master/server=server-one/deployment=ejb-in-ear.ear:list-modules
```

```
/host=master/server=server-one/deployment=ejb-in-ear.ear/subdeployment=ejb-in-ear-web.war:list-modules
```

此操作在紧凑视图中显示列表。

示例：标准列表输出

```
[standalone@localhost:9990 /] /deployment=sample-ear-1.0.ear:list-modules
 {
    "outcome" => "success",
    "result" => {
        "system-dependencies" => [
            {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8"},
            {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jsr310"},
            {"name" => "ibm.jdk"},
            {"name" => "io.jaegertracing.jaeger"},
            {"name" => "io.opentracing.contrib.opentracing-tracerresolver"},
            ...
        ],
        "local-dependencies" => [
            {"name" => "deployment.ejb-in-ear.ear.ejb-in-ear-ejb.jar"},
             ...
        ],
        "user-dependencies" => [
            {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8"},
            {"name" => "org.hibernate:4.1"},
```

```
            ...
        ]
    }
}
```

使用 **verbose=[false*|true]** 属性将产生更详细的列表。

示例：详细列表输出

```
[standalone@localhost:9990 /] /deployment=sample-ear-1.0.ear:list-modules(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "system-dependencies" => [
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8",
                "optional" => true,
                "export" => false,
                "import-services" => true
            },
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jsr310",
                "optional" => true,
                "export" => false,
                "import-services" => true
            },
            ...
        ],
        "local-dependencies" => [
            {
              "name" => "deployment.ejb-in-ear.ear.ejb-in-ear-ejb.jar",
              "optional" => false,
              "export" => false,
              "import-services" => true
            },
            ...
        ],
        "user-dependencies" => [
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8",
                "optional" => false,
                "export" => false,
                "import-services" => false
            },
            {
                "name" => "org.hibernate:4.1",
                "optional" => false,
                "export" => false,
                "import-services" => false
            },
            ...
```

下表描述了输出中提供的信息的类别：

表 3.1. list-modules Operation 的输出列表类别

类别	描述
system-dependencies	由 server 隐式添加.
local-dependencies	由部署的其他部分添加。
user-dependencies	由用户通过 **MANIFEST.MF 或 deployment-structure.xml** 文件定义。

3.10. 类加载参考

3.10.1. 隐式模块依赖项

下表列出了自动添加到部署中的模块，这些模块作为依赖项和触发依赖项的条件。

表 3.2. 隐式模块依赖项

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
应用程序客户端	• org.omg.api • org.jboss.xnio		
batch	• javax.batch.api • org.jberet.jberet-core • org.wildfly.jberet		
Jakarta Bean 验证	• org.hibernate.validator • javax.validation.api		
Core Server	• javax.api • sun.jdk • org.jboss.vfs • ibm.jdk		
DriverDependenciesProcessor		• javax.transaction.api	

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
EE	• org.jboss.invocation (except org.jboss.invocation.proxy.classloading) • org.jboss.as.ee（除 org.jboss.as.ee.component.serialization, org.jboss.as.ee.concurrent, org.jboss.as.ee.concurrent.handle） • org.wildfly.naming • javax.annotation.api • javax.enterprise.concurrent.api • javax.interceptor.api • javax.json.api • javax.resource.api • javax.rmi.api • javax.xml.bind.api • javax.api • org.glassfish.javax.el • org.glassfish.javax.enterprise.concurrent		
jakarta Enterprise Beans 3	• javax.ejb.api • javax.xml.rpc.api • org.jboss.ejb-client • org.jboss.iiop-client • org.jboss.as.ejb3	• org.wildfly.iiop-openjdk	

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
IIOP	• org.omg.api • javax.rmi.api • javax.orb.api		
Jakarta RESTful Web 服务 (RESTEasy)	• javax.xml.bind.api • javax.ws.rs.api • javax.json.api • org.jboss.resteasy.resteasy-atom-provider • org.jboss.resteasy.resteasy-crypto • org.jboss.resteasy.resteasy-validator-provider • org.jboss.resteasy.resteasy-jaxrs • org.jboss.resteasy.resteasy-jaxb-provider • org.jboss.resteasy.resteasy-jackson2-provider • org.jboss.resteasy.resteasy-jsapi • org.jboss.resteasy.resteasy-json-p-provider • org.jboss.resteasy.resteasy-multipart-provider • org.jboss.resteasy.resteasy-yaml-provider • org.codehaus.jackson.jackson-core-asl	• org.jboss.resteasy.resteasy-cdi	部署中存在 Jakarta RESTful Web Services 注释。

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
Jakarta Connectors	• javax.resource.api	• javax.jms.api • javax.validation.api • org.jboss.ironjacamar.api • org.jboss.ironjacamar.impl • org.hibernate.validator	部署资源适配器(RAR)存档。
Jakarta Persistence(Hibernate)	• javax.persistence.api	• org.jboss.as.jpa • org.jboss.as.jpa.spi • org.javassist	部署描述符中存在 **@PersistenceUnit** 或 **@PersistenceContext** 注解，或 **<persistence-unit-ref>** 或 **<persistence-context-ref>** 元素。 JBoss EAP 将持久性提供程序名称映射到模块名称.如果您在 **persistence.xml** 文件中命名特定的提供程序，则会为适当的模块添加依赖项。如果这不是所需的行为，您可以使用 **jboss-deployment-structure.xml** 文件将它排除。
Jakarta Server Faces		• javax.faces.api • com.sun.jsf-impl • org.jboss.as.jsf • org.jboss.as.jsf-injection	已添加到 EAR 应用程序. 仅当 **web.xml 文件没有指定值为 true 的 context-param.** jboss **.jbossfaces.WAR_BUNDLES_JSF_IMPL** 时，才会添加到 WAR 应用。
JSR-77	• javax.management.j2ee.api		

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
日志记录	• org.jboss.logging • org.apache.commons.logging • org.apache.logging.log4j.api • org.apache.log4j • org.slf4j • org.jboss.logging.jul-to-slf4j-stub		
mail	• javax.mail.api • javax.activation.api		
消息传递	• javax.jms.api	• org.wildfly.extension.messaging-activemq	
PicketLink Federation		• org.picketlink	
POJO	• org.jboss.as.pojo		
SAR		• org.jboss.modules • org.jboss.as.system-jmx • org.jboss.common-beans	部署具有 **jboss-service.xml** 的 SAR 存档。
Seam2		• org.jboss.vfs	.

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
安全性	• org.picketbox • org.jboss.as.security • javax.security.jacc.api • javax.security.auth.message.api		
ServiceActivator		• org.jboss.msc	
事务	• javax.transaction.api	• org.jboss.xts • org.jboss.jts • org.jboss.narayana.compensations	
Undertow	• javax.servlet.jstl.api • javax.servlet.api • javax.servlet.jsp.api • javax.websocket.api	• io.undertow.core • io.undertow.servlet • io.undertow.jsp • io.undertow.websocket • io.undertow.js • org.wildfly.clustering.web.api	
Web 服务	• javax.jws.api • javax.xml.soap.api • javax.xml.ws.api	• org.jboss.ws.api • org.jboss.ws.spi	如果这不是应用客户端类型，它将添加条件依赖项。

负责添加依赖性的子系统	Always Added 的软件包依赖项	带有条件地添加的软件包依赖项	触发依赖性添加的条件
weld（Jakarta 上下文和依赖注入）	• javax.enterprise.api • javax.inject.api	• javax.persistence.api • org.javassist • org.jboss.as.weld • org.jboss.weld.core • org.jboss.weld.probe • org.jboss.weld.api • org.jboss.weld.spi • org.hibernate.validator.cdi	部署中存在 **beans.xml** 文件。

3.10.2. 包括的模块

有关所含模块的完整列表以及是否被支持，请参阅红帽客户门户网站 中的 Red Hat JBoss Enterprise Application Platform 7 包含的模块。

第 4 章 日志记录

4.1. 关于日志记录

记录的是记录来自应用的一系列消息的练习，该应用提供应用活动的记录或日志。

在调试应用以及在生产环境中维护应用的系统管理员时，日志消息为开发人员提供了重要的信息。

大多数现代 Java 日志框架也包含详细信息，如消息的确切时间和来源。

4.1.1. 支持的应用程序日志记录框架

JBoss LogManager 支持以下日志记录框架：

- JBoss Logging（包含在 JBoss EAP 中）
- Apache Commons Logging
- 用于 Java 的简单日志记录 Facade(SLF4J)
- Apache log4j
- Java SE Logging (java.util.logging)

JBoss LogManager 支持以下 API：

- JBoss Logging
- Commons-logging
- SLF4J
- Log4j
- Log4j2
- java.util.logging

JBoss 日志管理器还支持以下 SPI：

- java.util.logging Handler
- Log4j 附录

注意

如果您使用 **Log4j API** 和 **Log4J 附录**，则对象将在通过之前转换为 **字符串**。

4.2. 使用 JBOSS LOGGING FRAMEWORK 进行日志记录

4.2.1. 关于 JBoss Logging

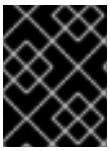
JBoss Logging 是包含在 JBoss EAP 中的应用程序日志框架。它提供了一种向应用添加日志的简便方法。您可以向应用添加代码，以使用框架以定义的格式发送日志消息。当应用部署到应用服务器时，服务器可以捕获这些消息，并根据服务器的配置显示或写入到 文件中。

JBoss Logging 提供以下功能：

- 种创新且易于使用的日志记录器。类型日志记录器是一个带有 **org.jboss.logging.annotations.MessageLogger** 注解的日志记录器接口。例如，请参阅 创建国际化的日志记录器、消息和例外。
- 完全支持国际化和本地化.转换器在属性文件中处理消息捆绑包，而开发人员则使用接口和注释。详情请查看 国际化和本地化。
- 构建时间工具，用于生成用于生产类型日志记录器和类型日志器的运行时生成以进行开发的 类型日志记录器。

4.2.2. 使用 JBoss Logging 将日志记录添加到应用程序

此流程演示了如何使用 JBoss Logging 向应用添加日志。



重要

如果使用 Maven 构建项目，您必须将 Maven 配置为使用 JBoss EAP Maven 存储库。如需更多信息，请参阅 配置 JBoss EAP Maven 存储库。

1. JBoss Logging JAR 文件必须位于应用程序的构建路径中。
   - 如果使用红帽代码Ready Studio 构建，请从 **Project** 菜单中选择 **Properties**，然后选择 **Targeted Runtimes** 并确保选中 JBoss EAP 的运行时。

     注意

     如果您在 Red Hat CodeReady Studio 中将 **Target runtime** 设为 **7.4** 或更新的运行时版本，则您的项目与 Jakarta EE 8 规范兼容。

   - 如果使用 Maven 构建项目，请确保将 **jboss-logging** 依赖项添加到项目的 **pom.xml** 文件中，以访问 JBoss Logging 框架：

     ```
     <dependency>
      <groupId>org.jboss.logging</groupId>
      <artifactId>jboss-logging</artifactId>
      <version>3.3.0.Final-redhat-1</version>
      <scope>provided</scope>
     </dependency>
     ```

     jboss-eap-jakartaee8 BOM 管理 **jboss-logging** 的版本。如需了解更多详细信息，请参阅 管理项目依赖项。有关登录应用的工作示例，请参见 JBoss EAP 附带 **的日志** 快速入门。

   您不需要将 JAR 包含在您的构建的应用中，因为 JBoss EAP 将它们提供给已部署的应用。

2. 对于您要添加日志的每个类：
   a. 为您要使用的 JBoss Logging 类命名空间添加导入语句。您至少需要以下导入：

      ```
      import org.jboss.logging.Logger;
      ```

b. 创建 **org.jboss.logging.Logger** 实例，并通过调用静态方法 **Logger.getLogger(Class)** 进行初始化。建议将它创建为各个类的一个实例变量。

```
private static final Logger LOGGER = Logger.getLogger(HelloWorld.class);
```

3. 在您要发送日志消息的代码中调用 **Logger** 对象方法。
日志器具 有许多不同的方法，具有不同参数以用于不同类型的消息。使用以下方法发送带有对应日志级别和消息参数的日志消息，**作为** 字符串：

```
LOGGER.debug("This is a debugging message.");
LOGGER.info("This is an informational message.");
LOGGER.error("Configuration file not found.");
LOGGER.trace("This is a trace message.");
LOGGER.fatal("A fatal error occurred.");
```

有关 JBoss Logging 方法的完整列表，请参阅 Logging API 文档。

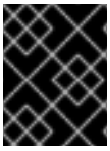
以下示例从属性文件加载应用的自定义配置。如果未找到指定的文件，则会记录 **aERROR** 级别的日志消息。

示例：使用 JBoss Logging 的应用程序日志记录

```
import org.jboss.logging.Logger;
public class LocalSystemConfig
{
  private static final Logger LOGGER = Logger.getLogger(LocalSystemConfig.class);

  public Properties openCustomProperties(String configname) throws
CustomConfigFileNotFoundException
  {
    Properties props = new Properties();
    try
    {
      LOGGER.info("Loading custom configuration from "+configname);
      props.load(new FileInputStream(configname));
    }
    catch(IOException e) //catch exception in case properties file does not exist
    {
      LOGGER.error("Custom configuration file ("+configname+") not found. Using defaults.");
      throw new CustomConfigFileNotFoundException(configname);
    }

    return props;
  }
}
```

4.2.3. 将 Apache Log4j2 API 添加到应用程序中

您可以使用 Apache Log4j2 API 而不是 Apache Log4j API 将应用程序日志记录消息发送到您的 JBoss LogManager 实施。



重要

JBoss EAP 7.4 发行版本支持 Log4J2 API，但不支持 Apache Log4j2 Core 实现、**org.apache.logging.log4j:log4j-core** 或其配置文件。

流程

1. 将 **org.apache.logging.log4j:log4j-api** 作为依赖项添加到项目 **pom.xml** 文件。

   将 org.apache.logging.log4j:log4j-api 添加到 pom.xml 文件示例。

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${version.org.apache.logging.log4j}</version>
    <scope>provided</scope>
</dependency>
```

   注意

   log4j-api Maven 依赖项是指 Apache Log4j2 API。 **log4j** Maven 依赖项指的是 Apache Log4j API。

   记录应用程序消息时，您会将该消息发送到您的 JBoss Log Manager 实施。

2. *可选：* 要排除 **org.apache.logging.log4j.api 模块**，必须从 **jboss-deployment-structure.xml** 文件中排除该模块，或者将 **add-logging-api-dependencies** 属性设置为 **false**。

4.2.4. 创建 Log4j2 LogManager 实现

您可以通过在项目的 **pom.xml** 文件中包含 Log4j2 API，在应用程序中使用 Log4j2 LogManager。另外，您必须在项目的 **pom.xml** 文件中包括对应的 Log4j2 LogManager 版本。

流程

1. 通过排除 **jboss-deployment-structure.xml** 文件中的 **org.apache.logging.log4j.api** 模块依赖项来禁用 Log4j **日志记录** 依赖项。

2. 将 **log4j-api** 依赖项和 **log4j2** 依赖项添加到项目 **pom.xml** 文件。

   将 log4j-api 依赖项和 log4j2 依赖项添加到 pom.xml 文件示例。

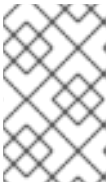
```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${version.org.apache.logging.log4j}</version>
</dependency>

<dependency>
    <groupId>org.apache.logging.log4j2</groupId>
    <artifactId>log4j2-core</artifactId>
    <version>${version.org.apache.logging.log4j}</version>
</dependency>
```

注意

log4j-api Maven 依赖项是指 Apache Log4j2 API。 **log4j** Maven 依赖项指的是 Apache Log4j API。

记录应用消息时，您会将该消息发送到 Log4j2 LogManager 实施。

3. *可选：* 要排除 **org.apache.logging.log4j.api 模块**，您必须从 **jboss-deployment-structure.xml** 文件中排除该模块，或者将 **add-logging-api-dependencies** 属性设为 **false**。然后，您必须将 **log4j2-api** 和 **log4j2-core** 添加到项目 **pom.xml** 文件。

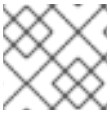


注意

如果对 **jboss-deployment-structure.xml** 文件进行更改，请对部署应用更改。如果您对 **add-logging-api-dependencies** 属性进行更改，请对所有已部署的应用程序应用更改。

4.3. 按部署日志记录

通过按部署日志记录，开发人员可以提前为其应用配置日志配置。部署应用时，根据定义的配置开始日志记录。通过此配置创建的日志文件仅包含有关应用程序行为的信息。



注意

如果未执行按部署的日志配置，则所有应用和服务器都使用来自 **logging** 子系统的配置。

与使用整个系统日志记录相比，这种方法具有优缺点。优点是 JBoss EAP 实例的管理员不需要配置服务器日志记录之外的任何其他日志记录。个缺点是每个部署的日志配置仅在服务器启动时读取，因此在运行时无法更改。

4.3.1. 将 Per-deployment Logging 添加到应用程序

要为应用配置每个部署日志记录，请将 **logging.properties** 配置文件添加到您的部署中。建议使用这个配置文件，因为它可以用于任何记录 facade，其中 JBoss 日志管理器是底层日志管理器。

添加配置文件的目录取决于部署方法。

- 对于 EAR 部署，将日志记录配置文件复制到 **META-INF/** 目录中。
- 对于 WAR 或 JAR 部署，请将日志配置文件复制到 **WEB-INF/classes/** 目录中。

注意

如果您使用 **Simple Logging Facade for Java(SLF4J)或 Apache log4j**，则适合使用 **logging.properties** 配置文件。如果您使用的是 Apache log4j 附加程序，则需要配置文件 **log4j.properties**。配置文件 **jboss-logging.properties** 仅支持传统部署。

配置 logging.properties
服务器引导时使用 **logging.properties** 文件，直到 **logging** 子系统启动为止。如果您的配置中没有包括 **logging** 子系统，则服务器将此文件中的配置用作整个服务器的日志记录配置。

JBoss 日志管理器配置选项

日志记录器选项

日志记录器选项

- **loggers=<category>[,<category>,...]** - 指定要配置以逗号分隔的日志记录器类别列表。此处未列出的任何类别均不会从以下属性配置：
- **logger.<category>.level=<level> -** 指定类别级别。级别可以是有效级别之一。如果未指定，将继承最接近的父项的级别。
- **logger.<category>.handlers=<handler>[,<handler>,...]** - 指定附加到此日志记录器的处理程序名称的逗号分隔列表。处理程序必须在同一属性文件中配置。
- **logger.<category>.filter=<filter>** - 为类别指定过滤器。
- **logger.<category>.useParentHandlers=(true|false)** - 指定日志消息是否应该与父处理程序匹配。默认值为 **true**。

处理程序选项

- **handler.<name>=<className>** - 指定要实例化的处理程序的类名称。此选项是必需的。

注意

表 4.1. 可能的类名称：

Name	关联的类
控制台（Console）	**org.jboss.logmanager.handlers.ConsoleHandler**
File	**org.jboss.logmanager.handlers.FileHandler**
periodic	**org.jboss.logmanager.handlers.PeriodicRotatingFileHandler**
Size	**org.jboss.logmanager.handlers.SizeRotatingFileHandler**
定期大小	**org.jboss.logmanager.handlers.PeriodicSizeRotatingFileHandler**
Syslog	**org.jboss.logmanager.handlers.SyslogHandler**
Async	**org.jboss.logmanager.handlers.AsyncHandler**

自定义 处理程序可以有任何关联的类或模块。用户可在 **logging** 子系统中定义自己的日志处理程序。

如需更多信息，请参阅 JBoss EAP *配置指南中* 的日志处理程序。

- **handler.<name>.level=<level>** - 限制这个处理器的级别。如果未指定，则保留默认值 ALL。

- **handler.<name>.encoding=<encoding>** - 指定字符编码（如果这个处理器类型支持）。如果未指定，则使用特定于处理程序的默认值。
- **handler.<name>.errorManager=<name> -** 指定要使用的错误管理器名称。错误管理器必须在同一属性文件中配置。如果未指定，则不配置错误管理器。
- **handler.<name>.filter=<name>** - 为类别指定过滤器。有关定义过滤器的详细信息，请参阅过滤器表达式。
- **handler.<name>.formatter=<name> -** 如果这个处理器类型支持，指定要使用的格式器的名称。格式器必须在同一属性文件中配置。如果没有指定，大多数处理程序类型将不会记录消息。
- **handler.<name>.properties=<property>[,<property>,...]** - 指定用于额外配置的 JavaBean-style 属性列表。进行粗略类型内省，以确定给定属性的相应转换。
  如果 JBoss 日志管理器中的所有文件处理程序 **都需要** 在 **fileName** 之前设置。在 **handler.<name>.properties** 中显示属性的顺序是设置属性的顺序。
- **handler.<name>.constructorProperties=<property>[,<property>,...]** - 指定应用作构造参数的属性列表。进行粗略类型内省，以确定给定属性的相应转换。
- **handler.<name>.<property>=<value> -** 设置 named 属性的值。
- **handler.<name>.module=<name> -** 指定处理器所在的模块的名称。

如需更多信息，请参阅 JBoss EAP *配置指南中* 的日志处理程序属性。

错误管理器选项

- **errorManager.<name>=<className>** - 指定要实例化错误管理器的类名称。此选项是必需的。
- **errorManager.<name>.properties=<property>[,<property>,...]** - 指定额外配置的 JavaBean-style 属性列表。进行粗略类型内省，以确定给定属性的相应转换。
- **errorManager.<name>.<property>=<value> -** 设置 named 属性的值。

格式选项

- **formatter.<name>=<className>** - 指定要实例化格式器的类名称。此选项是必需的。
- **formatter.<name>.properties=<property>[,<property>,...]** - 指定额外配置的 JavaBean-style 属性列表。进行粗略类型内省，以确定给定属性的相应转换。
- **formatter.<name>.constructorProperties=<property>[,<property>,...]** - 指定应用作构造参数的属性列表。进行粗略类型内省，以确定给定属性的相应转换。
- **formatter.<name>.<property>=<value> -** 设置 named 属性的值。

以下示例显示了记录到控制台的 **logging.properties** 文件的最小配置。

示例：最小 logging.properties 配置

```
# Additional logger names to configure (root logger is always configured)
# loggers=

# Root logger level
logger.level=INFO
```

```
# Root logger handlers
logger.handlers=CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%K{level}%d{HH:mm:ss,SSS} %-5p %C.%M(%L) [%c] %s%e%n
```

4.4. 日志记录配置集

日志记录配置集是独立的日志配置集合，可以分配给已部署的应用。与常规的 **logging** 子系统一样，日志记录配置文件可以定义处理程序、类别和根日志记录器，但它不能引用其他配置文件或主要 **日志记录** 子系统中的配置。日志配置文件的设计模仿 **logging** 子系统以方便配置。

通过日志记录配置文件，管理员可以创建专用于一个或多个应用的日志记录配置，而不影响任何其他日志配置。由于每个配置集都在服务器配置中定义，因此可以更改日志记录配置，而无需重新部署受影响的应用。

如需更多信息，请参阅 JBoss EAP 配置 *指南中的配置日志配置文件*。

每个日志记录配置集都可以有：

- 唯一的名称。此值是必需的。
- 任意数量的日志处理程序.
- 任何数量的日志类别。
- 最多一个根日志记录器.

应用可以使用 **Logging-Profile** 属性在其 **MANIFEST.MF** 文件中指定要使用的日志记录配置文件。

4.4.1. 在应用程序中指定日志配置集

应用指定要在其 **MANIFEST.MF** 文件中使用的日志配置集。

注意

您必须知道服务器上设置的日志配置集的名称，供这个应用程序使用。

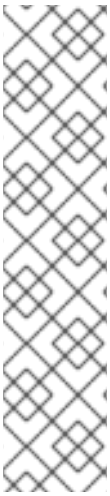
若要添加日志配置文件到应用，可编辑 **MANIFEST.MF** 文件。

- 如果您的应用程序没有 **MANIFEST.MF** 文件，请使用以下内容创建一个文件来指定日志记录配置文件名称：

```
Manifest-Version: 1.0
Logging-Profile: LOGGING_PROFILE_NAME
```

- 如果您的应用程序已有 **MANIFEST.MF** 文件，请添加以下行来指定日志配置文件名称：

```
Logging-Profile: LOGGING_PROFILE_NAME
```



注意

如果您使用 Maven 和 **maven-war-plugin**，请将 **MANIFEST.MF** 文件放在 **src/main/resources/META-INF/** 中，并将以下配置添加到 **pom.xml** 文件中：

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

部署应用时，它将将指定的日志记录配置文件中的配置用于其日志消息。

有关如何使用它配置日志配置文件和应用的示例，请参阅 JBoss EAP *配置指南中* 的日志配置文件配置示例。

4.5. 国际化和本地化

4.5.1. 简介

4.5.1.1. 关于国际化

国际化是设计软件的过程，以便能够适应不同的语言和区域，无需修改工程。

4.5.1.2. 关于本地化

本地化是通过添加特定于区域或语言的国际化软件和文本翻译的过程。

4.5.2. JBoss Logging 工具国际化和本地化

JBoss Logging Tools 是一种 Java API，支持日志消息、异常消息和通用字符串的国际化和本地化。除了提供转换机制外，JBoss 日志工具还为每一日志消息提供唯一标识符支持。

国际化的消息和异常是作为使用 **org.jboss.logging.annotations** 注解标注的接口内的方法定义创建的。实施接口是不需要的；JBoss 记录工具在编译时可以做到这一点。定义之后，您可以使用这些方法记录消息或获取代码中的异常对象。

通过为每个捆绑包创建属性文件（包含特定语言和区域的翻译），JBoss Logging 工具创建的国际化日志记录和异常接口可以本地化。JBoss Logging Tools 可以为每个捆绑包生成模板属性文件，然后可由转换器编辑。

JBoss Logging Tools 为您项目中的每个相应翻译属性文件创建每个捆绑包的实施。您要做的只是使用捆绑包中定义的方法，JBoss Logging Tools 可确保为您的当前区域设置调用正确的实施。

消息 ID 和项目代码是每个日志消息前面的唯一标识符。可以在文档中使用这些唯一标识符，以方便查找日志消息的相关信息。通过适当的文档，日志消息的含义可以从标识符确定，无论消息所写所用的语言是什么。

JBoss Logging 工具包括对以下功能的支持：

MessageLogger

org.jboss.logging.annotations 软件包中的此接口用于定义国际化的日志消息。消息日志记录器接口标有 **@MessageLogger**。

MessageBundle

此界面可用于定义通用可翻译消息和带有国际化消息的 Exception 对象。消息捆绑包不用于创建日志消息。消息捆绑包接口标有 **@MessageBundle**。

国际化日志消息

这些日志消息通过在 **MessageLogger** 中定义方法来创建。该方法必须使用 **@LogMessage 和 @Message** 注释标注，并且必须使用 **@Message** 的 value 属性指定日志消息。通过在属性文件中提供翻译，国际化日志消息会进行本地化。

JBoss Logging Tools 在编译时为每次编译时生成所需的日志记录类，并在运行时调用当前区域设置的正确方法。

国际化例外

国际化的异常是从 MessageBundle 中定义的方法返回的异常对象。可以为这些消息捆绑包添加注解，以定义默认的异常消息。如果在当前区域设置的匹配属性文件中找到一条消息，则默认消息将被替换为转换。国际化的异常也可以分配有项目代码和消息 ID。

国际化的信息

国际化的消息是从 **MessageBundle** 中定义的方法返回的字符串。可以给返回 Java String 对象的消息捆绑包方法添加注释，以定义该字符串的默认内容，称为消息。如果在当前区域设置的匹配属性文件中找到一条消息，则默认消息将被替换为转换。

翻译属性文件

转换属性文件是 Java 属性文件，包含来自一个区域设置、国家/地区和变体的一个接口的消息转换。JBoss 日志工具使用转换属性文件来生成返回消息的类。

JBoss Logging Tools 项目代码

项目代码是标识消息组的字符字符串。它们显示在每个日志消息的开头，并在消息 ID 的前面显示。项目代码通过 **@MessageLogger** 注释的 projectCode 属性来定义。

注意

有关新日志消息项目代码前缀的完整列表，请参阅 JBoss EAP 7.4 中使用的项目 代码。

JBoss Logging 工具消息 ID

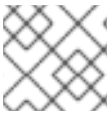
消息 ID 是结合项目代码时唯一标识日志消息的数字。消息 ID 显示在每条日志消息的开头，并附加到消息的项目代码中。消息 ID 通过 **@Message** 注释的 ID 属性来定义。

JBoss EAP 附带的 **logging-tools** 快速入门是一个简单的 Maven 项目，提供 JBoss Logging 工具的许多功能的有效示例。以下代码示例取自 **logging-tools** quickstart。

4.5.3. 创建国际化的日志记录器、消息和例外

4.5.3.1. 创建国际化的日志消息

您可以通过创建 **MessageLogger** 接口，使用 JBoss Logging Tools 创建国际化日志消息。



注意

本节不涵盖所有可选功能或日志消息的本地化。

1. 如果您还没有这样做，请将您的 Maven 设置配置为使用 JBoss EAP Maven 存储库。
   如需更多信息，请参阅使用 Maven 设置配置 JBoss EAP Maven 存储库。

2. 配置项目的 **pom.xml** 文件，以使用 JBoss Logging 工具。
   详情请查看 JBoss Logging Tools Maven 配置。

3. 通过添加 Java 接口到项目来创建消息日志记录器接口，以包含日志消息定义。
   指出用于描述它将定义的日志消息的接口。日志消息接口有以下要求：

   - 它必须标有 **@org.jboss.logging.annotations.MessageLogger**。
   - 另外，还可扩展 **org.jboss.logging.BasicLogger**。
   - 接口必须定义一个字段，该字段是与接口相同的类型的消息日志记录器。使用 **@org.jboss.logging. Logger 的 getMessageLogger （）** 方法执行此操作。

     示例：创建消息日志器

     ```
     package com.company.accounts.loggers;

     import org.jboss.logging.BasicLogger;
     import org.jboss.logging.Logger;
     import org.jboss.logging.annotations.MessageLogger;

     @MessageLogger(projectCode="")
     interface AccountsLogger extends BasicLogger {
       AccountsLogger LOGGER = Logger.getMessageLogger(
           AccountsLogger.class,
           AccountsLogger.class.getPackage().getName() );
     }
     ```

4. 向接口添加一个方法定义，以用于每一日志消息。
   请以描述性方式命名每个方法，使其表示的日志消息。每个方法都有以下要求：

   - 该方法必须返回 **void**。
   - 它必须标有 **@org.jboss.logging.annotation.LogMessage** 注释。
   - 它必须标有 **@org.jboss.logging.annotations.Message** 注释。
   - 默认日志级别为 **INFO**。
   - **@org.jboss.logging.annotations.Message** 的 value 属性包含默认日志消息，如果没有可用的翻译则使用此消息。

     ```
     @LogMessage
     @Message(value = "Customer query failed, Database not available.")
     void customerQueryFailDBClosed();
     ```

5. 在您的代码中添加调用，以从其中记录消息来调用，从而调用方法。
无需创建接口实施，注释处理器会在编译项目时为您执行此操作。

```
AccountsLogger.LOGGER.customerQueryFailDBClosed();
```

自定义日志记录器从 **BasicLogger** 子级，因此也可以使用 **BasicLogger** 的日志记录方法。不需要创建其他日志记录器来记录非国际化的消息。

```
AccountsLogger.LOGGER.error("Invalid query syntax.");
```

6. 该项目现在支持一个或多个可本地化的国际化日志记录器。

注意

JBoss EAP 附带的 **logging-tools** 快速入门是一个简单的 Maven 项目，它提供了一个如何使用 JBoss Logging Tools 的工作示例。

4.5.3.2. 创建和使用国际化的信息

此流程演示了如何创建和使用国际化的信息。

注意

本节不涵盖所有可选功能或对这些消息进行本地化的流程。

1. 如果您还没有这样做，请将您的 Maven 设置配置为使用 JBoss EAP Maven 存储库。如需更多信息，请参阅使用 Maven 设置配置 JBoss EAP Maven 存储库。

2. 配置项目的 **pom.xml** 文件，以使用 JBoss Logging 工具。详情请查看 JBoss Logging Tools Maven 配置。

3. 为异常创建一个接口。JBoss 日志工具在接口中定义国际化的消息。为每个接口命名其包含的消息的描述性。接口有以下要求：

   - 必须将其声明为 **公开**。
   - 它必须标有 **@org.jboss.logging.annotations.MessageBundle**。
   - 接口必须定义与接口相同的消息捆绑包的字段。

   示例：创建一个 消息捆绑包 接口

```
@MessageBundle(projectCode="")
public interface GreetingMessageBundle {
   GreetingMessageBundle MESSAGES =
Messages.getBundle(GreetingMessageBundle.class);
}
```

注意

调用 **Messages.getBundle(GreetingMessagesBundle.class)** 等同于调用 **Messages.getBundle（GreetingMessagesBundle.class, Locale.getDefault（））**。

locale **.getDefault（）** 获取 Java 虚拟机此实例的默认区域设置的当前值。Java 虚拟机会根据主机环境设置启动过程中的默认区域设置。如果未明确指定区域设置，它将被许多区域敏感方法使用。它可以通过 **setDefault** 方法更改。

如需更多信息，请参阅《JBoss EAP *配置指南*》中设置服务器的默认 区域。

4. 向每条消息的接口添加一个方法定义。请以描述性方式命名每个方法，使其表示的消息。每个方法都有以下要求：

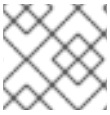
   - 它必须返回类型为 **String** 的对象。
   - 它必须标有 **@org.jboss.logging.annotations.Message** 注释。
   - **@org.jboss.logging.annotations.Message** 的 value 属性必须设置为 default 消息。这是如果没有可用的翻译，则会使用的消息。

   ```
   @Message(value = "Hello world.")
   String helloworldString();
   ```

5. 在应用程序中调用需要获取消息的接口方法：

   ```
   System.out.println(helloworldString());
   ```

该项目现在支持可本地化的国际化消息字符串。



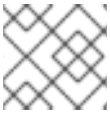
注意

请参阅 JBoss EAP 附带的 **logging-tools** quickstart，以获取完整的工作示例。

4.5.3.3. 创建国际化例外

您可以使用 JBoss Logging Tools 创建和使用国际化的异常。

以下说明假定您要为使用 Red Hat CodeReady Studio 或 Maven 构建的现有软件项目添加国际化的异常。



注意

本节不涵盖这些例外的所有可选功能或本地化过程。

1. 配置项目的 **pom.xml** 文件，以使用 JBoss Logging 工具。详情请查看 JBoss Logging Tools Maven 配置。
2. 为异常创建一个接口。JBoss 日志工具在接口中定义了国际化的异常。为每个接口命名其定义的异常的描述性。接口有以下要求：

   - 必须将其声明为 **公开**。

- 它必须标有 **@MessageBundle**。
- 接口必须定义与接口相同的消息捆绑包的字段。

  示例：创建一个 例外捆绑包 接口

```
@MessageBundle(projectCode="")
public interface ExceptionBundle {
   ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);
}
```

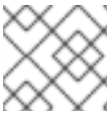
3. 为每个例外向接口添加一个方法定义。以描述性方式命名每个方法表示的异常。每个方法都有以下要求：
   - 它必须返回 **例外** 对象或子类型 **例外**。
   - 它必须标有 **@org.jboss.logging.annotations.Message** 注释。
   - **@org.jboss.logging.annotations.Message** 的 value 属性必须设置为默认异常消息。这是如果没有可用的翻译，则会使用的消息。
   - 如果返回的异常具有除消息字符串外还需要参数的构造器，则必须使用 **@Param** 注释在方法定义中提供这些参数。这些参数的类型和顺序必须与在异常的结构中相同。

```
@Message(value = "The config file could not be opened.")
IOException configFileAccessError();

@Message(id = 13230, value = "Date string '%s' was invalid.")
ParseException dateWasInvalid(String dateString, @Param int errorOffset);
```

4. 在您需要获得其中一个例外的地方，调用代码中的接口方法。方法不会抛出异常，它们返回异常对象，然后您可以抛出它。

```
try {
   propsInFile=new File(configname);
   props.load(new FileInputStream(propsInFile));
}
catch(IOException ioex) {
  //in case props file does not exist
   throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
}
```

该项目现在支持国际化的异常，可以本地化。



注意

请参阅 JBoss EAP 附带的 **logging-tools** quickstart，以获取完整的工作示例。

4.5.4. 本地化国际化日志记录器、消息和例外

4.5.4.1. 使用 Maven 生成新转换属性文件

使用 Maven 构建的项目可以为每个 **MessageLogger 和 Message Bundle** 生成空的转换属性文件。这些文件随后可用作新的转换属性文件。

以下步骤演示了如何配置 Maven 项目以生成新的转换属性文件。

先决条件

- 您必须已有一个正常工作的 Maven 项目。
- 必须已为 JBoss Logging 工具配置该项目。
- 该项目必须包含一个或多个定义国际化日志消息或异常的接口。

生成转换属性文件

1. 通过将 **-AgenereatedTranslationFilePath** 编译器参数添加到 Maven 编译器插件配置来添加 Maven 配置，并为它分配创建新文件的路径。
   此配置会在 Maven 项目的 **target/generated-translation-files** 目录中创建新文件。

   示例：定义转换文件路径

   ```
   <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <version>2.3.2</version>
     <configuration>
       <source>1.6</source>
       <target>1.6</target>
       <compilerArgument>
       -AgeneratedTranslationFilesPath=${project.basedir}/target/generated-translation-files
       </compilerArgument>
       <showDeprecation>true</showDeprecation>
     </configuration>
   </plugin>
   ```

2. 使用 Maven 构建项目：

   ```
   $ mvn compile
   ```

   为标有 **@MessageBundle 或 @Message Logger** 的每个接口创建了一个属性文件。

   - 新文件创建在与声明每个接口的 Java 软件包对应的子目录中。
   - 每一新文件采用以下模式命名，其中 **INTERFACE_NAME** 是用于生成文件的接口的名称。

     ```
     INTERFACE_NAME.i18n_locale_COUNTRY_VARIANT.properties
     ```

现在可将生成的文件复制到您的项目中，作为新翻译的基础。

注意

请参阅 JBoss EAP 附带的 **logging-tools** quickstart，以获取完整的工作示例。

4.5.4.2. 转换国际化的日志记录器、例外或消息

属性文件可用于使用 JBoss 日志工具为接口中定义的日志和异常消息提供翻译。

以下步骤演示了如何创建和使用转换属性文件，并且假定您已拥有为国际化异常或日志消息定义的一个或多个接口的项目。

先决条件

- 您必须已有一个正常工作的 Maven 项目。
- 必须已为 JBoss Logging 工具配置该项目。
- 该项目必须包含一个或多个定义国际化日志消息或异常的接口。
- 项目必须配置为生成模板转换属性文件。

转换国际化的日志记录器、例外或消息

1. 运行以下命令来创建模板转换属性文件：

```
$ mvn compile
```

2. 将您要从创建它们的目录中转换的接口模板复制到项目的 **src/main/resources** 目录中。属性文件必须与要转出的接口位于同一个软件包中。
3. 重命名复制的模板文件，以指明它将包含的语言。例如：**GreeterLogger.i18n_fr_FR.properties**。
4. 编辑新转换属性文件的内容，使其包含适当的翻译：

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

5. 重复复制模板的过程，并根据捆绑包中的每个转换进行修改。

该项目现在包含一个或多个消息或日志记录器捆绑包的转换。构建项目会生成适当的类，以使用提供的转换记录日志消息。无需显式调用特定语言的方法或提供参数，JBoss 日志工具会自动将正确的类用于应用服务器的当前区域设置。

生成类的源代码可以在 **target/generated-sources/annotations/** 下查看。

4.5.5. 自定义国际化日志消息

4.5.5.1. 添加消息 ID 和项目代码到日志消息

此流程演示了如何将消息 ID 和项目代码添加到使用 JBoss Logging Tools 创建的国际化日志消息中。日志消息中必须同时显示项目代码和消息 ID。如果消息同时没有项目代码和消息 ID，则不会显示任何消息。

先决条件

1. 您必须已拥有含有国际化日志消息的项目。详情请查看 创建国际化的日志消息。
2. 您需要知道您将要使用的项目代码。您可以使用单个项目代码，或者为每个接口定义不同的项目代码。

添加消息 ID 和项目代码到日志消息

1. 利用附加至自定义日志记录器接口的 **@MessageLogger** 注释的 projectCode 属性，指定接口的项目代码。接口中定义的所有消息都将使用该项目代码。

```
@MessageLogger(projectCode="ACCNTS")
interface AccountsLogger extends BasicLogger {

}
```

2. 利用附加至消息方法的 **@Message** 注释的 **id** 属性，指定每条消息的消息 ID。

```
@LogMessage
@Message(id=43, value = "Customer query failed, Database not available.")  void
customerQueryFailDBClosed();
```

3. 如果日志消息同时关联了消息 ID 和项目代码，这些日志消息会将它们添加到记录的消息中。

```
10:55:50,638 INFO  [com.company.accounts.ejb] (MSC service thread 1-4) ACCNTS000043:
Customer query failed, Database not available.
```

4.5.5.2. 指定消息的日志级别

JBoss Logging Tools 定义的消息的默认日志级别为 **INFO**。可以使用附加至日志记录方法的 **@LogMessage** 注释的 **level** 属性来指定不同的日志级别。使用以下步骤指定不同的日志级别。

1. 将 **level** 属性添加到日志消息方法定义的 **@LogMessage** 注释。

2. 利用 **level** 属性，分配此消息的日志级别。**level** 的有效值是在 **org.jboss.logging.Logger.Level**:**DEBUG**、**ERROR**、**FATAL**、**INFO**、**TRACE** 和 **WARN** 中定义的六个常量。

```
import org.jboss.logging.Logger.Level;

@LogMessage(level=Level.ERROR)
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

在上例中调用日志记录方法将在 at **ERROR** 级别上生成一条日志消息。

```
10:55:50,638 ERROR  [com.company.app.Main] (MSC service thread 1-4)
 Customer query failed, Database not available.
```

4.5.5.3. 使用参数自定义日志消息

自定义日志记录方法可以定义参数。这些参数用于传递要在日志消息中显示的其他信息。在日志消息中出现参数时，使用显式或普通索引方式在消息本身中指定。

使用参数自定义日志消息

1. 向方法定义中添加任何类型的参数。无论类型如何，参数的 String 表示法就是消息中显示的内容。

2. 添加对该日志消息的参数引用。参考可以使用显式或普通索引。

   - 要使用普通索引，请在您希望显示每个参数的消息字符串中插入 **%s** 字符。**%s** 的第一个实例将插入第一个参数，第二个实例将插入第二个参数，以此类推。

   - 要使用显式索引，请在消息中插入 **%#$s** 字符，其中 # 代表您要显示的参数数。

使用显式索引时，消息中的参数引用的顺序与方法中定义的不同顺序不同。这对于可能需要对参数进行不同排序的转换消息来说非常重要。

重要

参数数量必须与指定消息中参数的引用数量匹配，否则代码不会编译。标有 **@Cause 注释的参数** 不包含在参数数中。

以下是使用普通索引的消息参数示例：

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

以下是使用显式索引的消息参数示例：

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.5.4. 指定一个例外作为日志消息的 Causeion

JBoss Logging Tools 允许将自定义日志记录方法的一个参数定义为消息的原因。此参数必须是 **可浏览** 类型或其任何子类，并且标上 **@Cause 注释**。此参数无法像其他参数一样在日志消息中引用，并在日志消息后显示。

以下步骤演示了如何使用 **@Cause 参数** 更新日志记录方法以指示"阻碍"异常：假设您已创建了您要向其添加此功能的国际化日志消息。

指定一个例外作为日志消息的 Causeion

1. 向方法添加类型为 **可** 浏览或其子类的参数。

   ```
   @LogMessage
   @Message(id=404, value="Loading configuration failed. Config file:%s")
   void loadConfigFailed(Exception ex, File file);
   ```

2. 添加 **@Cause** 注释到 参数。

   ```
   import org.jboss.logging.annotations.Cause

   @LogMessage
   @Message(value = "Loading configuration failed. Config file: %s")
   void loadConfigFailed(@Cause Exception ex, File file);
   ```

3. 调用 方法。在您的代码中调用该方法时，必须传递一个正确类型的对象，并在日志消息后显示该方法。

```
try
{
   confFile=new File(filename);
   props.load(new FileInputStream(confFile));
}
catch(Exception ex) //in case properties file cannot be read
{
    ConfigLogger.LOGGER.loadConfigFailed(ex, filename);
}
```

如果代码引发 **FileNotFoundException** 类型异常，则以上代码示例的输出如下：

```
10:50:14,675 INFO [com.company.app.Main] (MSC service thread 1-3) Loading configuration failed.
Config file: customised.properties
java.io.FileNotFoundException: customised.properties (No such file or directory)
   at java.io.FileInputStream.open(Native Method)
   at java.io.FileInputStream.<init>(FileInputStream.java:120)
   at com.company.app.demo.Main.openCustomProperties(Main.java:70)
   at com.company.app.Main.go(Main.java:53)
   at com.company.app.Main.main(Main.java:43)
```

4.5.6. 自定义国际化例外

4.5.6.1. 添加消息 ID 和项目代码到例外消息

消息 ID 和项目代码是国际化异常所显示的每条消息的前面的唯一标识符。通过这些识别代码，可以创建对应用中所有异常消息的引用。这允许用户查找以自己不理解的语言编写的异常消息的含义。

以下步骤演示了如何将消息 ID 和项目代码添加到使用 JBoss Logging Tools 创建的国际化异常消息中。

先决条件

1. 您必须已有具有国际化例外的项目。详情请查看 创建国际化例外。
2. 您需要知道您将要使用的项目代码。您可以使用单个项目代码，或者为每个接口定义不同的项目代码。

添加消息 ID 和项目代码到例外消息

1. 使用附加至异常捆绑包接口的 **@MessageBundle 注释的 projectCode** 属性来指定项目代码。接口中定义的所有消息都将使用该项目代码。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
   ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);
}
```

2. 使用与定义异常的方法关联的 **@Message** 注释的 **id** 属性，指定各个异常的消息 ID。

```
@Message(id=143, value = "The config file could not be opened.")
IOException configFileAccessError();
```

重要

包含项目代码和消息 ID 的消息在消息前面显示。如果消息同时没有项目代码和消息 ID，则也不会显示。

示例：国际化例外

这个异常捆绑包接口示例使用 "ACCTS" 的项目代码。它包含一个异常方法，其 ID 为 "143"。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
   ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);

   @Message(id=143, value = "The config file could not be opened.")
   IOException configFileAccessError();
}
```

可以使用以下代码获取并抛出异常对象：

```
throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
```

这会显示类似如下的异常信息：

```
Exception in thread "main" java.io.IOException: ACCTS000143: The config file could not be opened.
at com.company.accounts.Main.openCustomProperties(Main.java:78)
at com.company.accounts.Main.go(Main.java:53)
at com.company.accounts.Main.main(Main.java:43)
```

4.5.6.2. 使用参数自定义例外消息

定义例外的异常捆绑包方法可以指定参数，以传递要在异常消息中显示的其他信息。在消息本身中使用显式或普通索引来指定参数在异常消息中的确切位置。

使用参数自定义例外消息

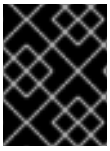
1. 向方法定义中添加任何类型的参数。无论类型如何，参数的 String 表示法就是消息中显示的内容。

2. 添加对异常消息的参数引用。参考可以使用显式或普通索引。

   - 要使用普通索引，请在您希望显示每个参数的消息字符串中插入 **%s** 字符。**%s** 的第一个实例将插入第一个参数，第二个实例将插入第二个参数，以此类推。

   - 要使用显式索引，请在消息中插入 **%#$s** 字符，其中 # 代表您要显示的参数数。

使用显式索引时，消息中的参数引用的顺序与方法中定义的不同顺序不同。这对于可能需要对参数进行不同排序的转换消息来说非常重要。



重要

参数数量必须与指定消息中参数的引用数量匹配，否则代码将不会编译。标有 **@Cause 注释的参数** 不包含在参数数中。

示例：使用机构索引

```
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

示例：使用 Explicit Indexes

```
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.6.3. 将一个例外指定为 Another Exception 的 Cause of Exception

异常捆绑包方法返回的异常可能会有另一个例外指定为根本原因。这可以通过向方法添加 参数并使用 **@Cause 标注 参数。**此参数用于传递导致的异常，且无法在异常消息中引用。

下列步骤演示了如何使用 **@Cause 参数** 从异常捆绑包更新方法以指示导致的异常。假设您已创建了要添加此功能的异常捆绑包。

1. 向方法添加类型为 **可** 浏览或其子类的参数。

```
@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(Throwable cause, String msg);
```

2. 添加 **@Cause** 注释到 参数。

```
import org.jboss.logging.annotations.Cause

@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(@Cause Throwable cause, String msg);
```

3. 调用 interface 方法以获取异常对象。最常见的用例是从 **catch** 块中抛出新的异常，并将捕获的异常指定为原因。

```
try
{
  ...
}
catch(Exception ex)
{
  throw ExceptionBundle.EXCEPTIONS.calculationError(
                              ex, "calculating payment due per day");
}
```

以下是将异常指定为另一例外的原因的示例。此异常捆绑包定义了一个返回类型 **ArithmeticException** 异常的方法。

```
@MessageBundle(projectCode = "TPS")
interface CalcExceptionBundle
```

```
{
    CalcExceptionBundle EXCEPTIONS = Messages.getBundle(CalcExceptionBundle.class);

    @Message(id=328, value = "Error calculating: %s.")
    ArithmeticException calcError(@Cause Throwable cause, String value);
}
```

以下示例演示了引发异常的操作，因为它尝试将整数除以零。这个异常会被捕获，使用第一个异常作为原因创建新的异常。

```
int totalDue = 5;
int daysToPay = 0;
int amountPerDay;

try
{
   amountPerDay = totalDue/daysToPay;
}
catch (Exception ex)
{
   throw CalcExceptionBundle.EXCEPTIONS.calcError(ex, "payments per day");
}
```

以下是从上例生成的异常消息：

```
Exception in thread "main" java.lang.ArithmeticException: TPS000328: Error calculating: payments
per day.
    at com.company.accounts.Main.go(Main.java:58)
    at com.company.accounts.Main.main(Main.java:43)
Caused by: java.lang.ArithmeticException: / by zero
    at com.company.accounts.Main.go(Main.java:54)
    ... 1 more
```

4.5.7. JBoss Logging 工具参考

4.5.7.1. JBoss Logging Tools Maven 配置

以下步骤将 Maven 项目配置为使用 JBoss Logging 和 JBoss Logging 工具进行国际化。

1. 如果您还没有这样做，请将您的 Maven 设置配置为使用 JBoss EAP 存储库。如需更多信息，请参阅使用 Maven 设置配置 JBoss EAP Maven 存储库。
   在项目的 **pom.xml** 文件的 **<dependencyManagement>** 部分中包含 **jboss-eap-jakartaee8** BOM。

```
<dependencyManagement>
  <dependencies>
    <!-- JBoss distributes a complete set of Jakarta EE APIs including
      a Bill of Materials (BOM). A BOM specifies the versions of a "stack" (or
      a collection) of artifacts. We use this here so that we always get the correct versions of
artifacts.
      Here we use the jboss-javaee-7.0 stack (you can
      read this as the JBoss stack of the Jakarta EE APIs). You can actually
      use this stack with any version of JBoss EAP that implements Jakarta EE. -->
    <dependency>
```

```
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  <dependencies>
<dependencyManagement>
```

2. 将 Maven 依赖项添加到项目的 **pom.xml** 文件中：

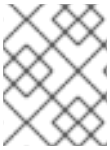
   a. 添加 **jboss-logging** 依赖项，以访问 JBoss Logging 框架。

   b. 如果您计划使用 JBoss Logging 工具，还要添加 **jboss-logging-processor** 依赖项。这两个依赖关系都在上一步中添加的 JBoss EAP BOM 中提供，因此每个依赖项的 scope 元素可以设置为 **提供** ：

```
<!-- Add the JBoss Logging Tools dependencies -->
<!-- The jboss-logging API -->
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <scope>provided</scope>
</dependency>
<!-- Add the jboss-logging-tools processor if you are using JBoss Tools  -->
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging-processor</artifactId>
  <scope>provided</scope>
</dependency>
```

3. maven-compiler-plugin 必须至少是 **3.1** 版本，并且针对目标及生成的 **1.8** 源进行配置。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```



注意

有关配置为使用 JBoss Logging **工具的 pom.xml** 文件的完整工作示例，请查看 JBoss EAP 附带的 **logging-tools** quickstart。

4.5.7.2. 翻译属性文件格式

用于 JBoss Logging Tools 中消息转换的属性文件是标准 Java 属性文件。文件的格式是 **java.util.Properties** 类文档 中描述的简单面向行的 **key=value** 对格式。

文件名格式具有以下格式：

-

```
InterfaceName.i18n_locale_COUNTRY_VARIANT.properties
```

- **interfaceName** 是转换应用到的接口的名称。
- **locale**、**CO** UNT **RY 和 VARIANT** 识别该转换应用到的区域设置。
- **locale** 和 **COUNTRY** 分别使用 ISO-639 和 ISO-3166 语言及国家/地区代码指定语言和区域。**COUNTRY** 是可选的。
- **VARIANT** 是一个可选标识符，可用于识别仅适用于特定操作系统或浏览器的翻译。

转换文件中包含的属性是正在转换的接口中方法的名称。属性的分配值是转换。如果方法过载，则通过附加点以及参数数到名称来指示这一点。转换方法只能通过提供不同数量的参数来过载。

示例：转换属性文件

文件名： **GreeterService.i18n_fr_FR_POSIX.properties.**

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

4.5.7.3. JBoss Logging 工具注释参考

以下注释在 JBoss Logging 中定义，用于日志消息、字符串和异常的国际化和本地化。

表 4.2. JBoss Logging 工具注释

注解	目标	描述	属性
@MessageBundle	Interface	将接口定义为消息捆绑包。	projectCode
@MessageLogger	Interface	将接口定义为消息日志记录器。	projectCode
@Message	方法	可以在消息捆绑包和消息日志记录器中使用。在消息捆绑包中，它将方法定义为返回本地化 String 或 Exception 对象的方法。在消息日志记录器中，它将方法定义为本地化日志记录器。	值, id
@LogMessage	方法	在消息日志记录器中将某一方法定义为记录方法。	level（默认 **INFO**）
@Cause	参数	将参数定义为传递例外作为日志消息或其他例外的原因的参数。	-
@Param	参数	将参数定义为传递到例外构造器的参数。	-

4.5.7.4. JBoss EAP 中使用的项目代码

下表列出了 JBoss EAP 7.4 中使用的所有项目代码，以及它们所属的 Maven 模块。

表 4.3. JBoss EAP 中使用的项目代码

Maven 模块	项目代码
AppClient	WFLYAC
批处理/扩展-jberet	WFLYBATCH
批处理/扩展	WFLYBATCH-DEPRECATED
batch/jberet	WFLYBAT
bean-validation	WFLYBV
controller-client	WFLYCC
controller	WFLYCTL
clustering/common	WFLYCLCOM
clustering/ejb/infinispan	WFLYCLEJBINF
群集/infinispan/extension	WFLYCLINF
clustering/jgroups/extension	WFLYCLJG
Cluster/server	WFLYCLSV
clustering/web/infinispan	WFLYCLWEBINF
connector	WFLYJCA
deployment-repository	WFLYDR
deployment-scanner	WFLYDS
domain-http	WFLYDMHTTP
域管理	WFLYDM
EE	WFLYEE
ejb3	WFLYEJB
嵌入式	WFLYEMB

Maven 模块	项目代码
host-controller	WFLYDC
host-controller	WFLYHC
iiop-openjdk	WFLYIIOP
io/subsystem	WFLYIO
jaxrs	WFLYRS
jdr	WFLYJDR
jmx	WFLYJMX
jpa/hibernate5	JIPI
jpa/spi/src/main/java/org/jipijapa/JipiLogger.java	JIPI
jpa/subsystem	WFLYJPA
jsf/subsystem	WFLYJSF
jsr77	WFLYEEMGMT
启动程序	WFLYLNCHR
legacy/jacorb	WFLYORB
传统/消息传递	WFLYMSG
legacy/web	WFLYWEB
logging	WFLYLOG
mail	WFLYMAIL
management-client-content	WFLYCNT
messaging-activemq	WFLYMSGAMQ
mod_cluster/extension	WFLYMODCLS
命名	WFLYNAM

Maven 模块	项目代码
network	WFLYNET
patching	WFLYPAT
PicketLink	WFLYPL
platform-mbean	WFLYPMB
pojo	WFLYPOJO
process-controller	WFLYPC
协议	WFLYPRT
远程	WFLYRMT
request-controller	WFLYREQCON
rTS	WFLYRTS
sar	WFLYSAR
security-manager	WFLYSM
安全	WFLYSEC
server	WFLYSRV
system-jmx	WFLYSYSJMX
threads	WFLYTHR
事务	WFLYTX
undertow	WFLYUT
webservices/server-integration	WFLYWS
weld	WFLYWELD
xts	WFLYXTS

第 5 章 远程 JNDI 查找

5.1. 将对象注册到 JAVA 命名和目录接口

Java 命名和目录接口是目录服务的 Java API，允许 Java 软件客户端使用名称来发现和查找对象。

如果注册到 Java 命名和目录接口的对象需要由远程 Java 命名和目录接口客户端（例如在单独的 JVM 中运行的客户端）查找，则必须在 **java:jboss/exported** 上下文下注册。

例如，如果 **messaging-activemq** 子系统中的 Jakarta 消息队列必须公开给远程 Java 命名和目录接口客户端，则必须使用 **java:jboss/exported/jms/queue/myTestQueue 将它** 注册到 Java 命名和目录接口。远程 Java 命名和目录接口客户端可以按照 name **jms/queue/myTestQueue 进行** 查找。

示例：在 standalone-full(-ha).xml 中配置Queue

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <jms-queue name="myTestQueue" entries="java:jboss/exported/jms/queue/myTestQueue"/>
    ...
  </server>
</subsystem>
```

5.2. 配置远程 JNDI

远程 JNDI 客户端可以根据名称从 JNDI 连接和查找对象。若要使用远程 JNDI 客户端查找对象，它必须在其类路径 **中包含 jboss-client.jar**。**jboss-client.jar** 位于 ***EAP_HOME*/bin/client/jboss-client.jar**。

以下示例演示了如何在远程 JNDI 客户端中从 JNDI 查找 **myTestQueue** 队列：

示例： MDB 资源适配器的配置

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote+http://HOST_NAME:8080");
context = new InitialContext(properties);
Queue myTestQueue = (Queue) context.lookup("jms/queue/myTestQueue");
```

5.3. JNDI INVOCATION OVER HTTP

通过 HTTP 进行 JNDI 调用包括两个不同的部分：客户端实施和服务器端实施。

5.3.1. 客户端实施

客户端实施与远程命名实施类似，但基于使用 Undertow HTTP 客户端的 HTTP。

连接管理是隐式而非直接的，使用与现有远程命名实施中使用的缓存方法。连接池根据连接参数进行缓存。如果指定超时期间不使用它们，则会丢弃它们。

要将远程 JNDI 客户端应用程序配置为使用 HTTP 传输，您必须在 HTTP 传输实现上添加以下依赖关系：

```
<dependency>
    <groupId>org.wildfly.wildfly-http-client</groupId>
    <artifactId>wildfly-http-naming-client</artifactId>
</dependency>
```

要执行 HTTP 调用，您必须使用 **http** URL 方案，并包含 HTTP 调用器的上下文名称 **wildfly-services**。例如：如果您使用 **remote+http://localhost:8080** 作为目标 URL，为了使用 HTTP 传输，您必须将其 **更新为** http://localhost:8080/wildfly-services。

5.3.2. 服务器端实施

服务器端实施与现有的远程命名实施类似，但具有 HTTP 传输。

要配置服务器，您必须在 **undertow** 子系统中使用的每个虚拟主机上启用 **http-invoker**。这在标准配置中默认启用。如果禁用，您可以使用以下管理 CLI 命令重新启用它：

```
/subsystem=undertow/server=default-server/host=default-host/setting=http-invoker:add(http-
authentication-factory=myfactory, path="/wildfly-services")
```

http-invoker 属性采用两个参数：默认为 **/wildfly-services 的路径** 和 **http-authentication-factory**，必须是 Elytron **http-authentication-factory 的** 引用。

注意

任何旨在使用 **http-authentication-factory** 的部署都必须使用 Elytron 安全性和与指定 HTTP 身份验证工厂对应的相同安全域。

第 6 章 WEB 应用程序中的集群

6.1. 会话复制

6.1.1. 关于 HTTP 会话复制

会话复制可确保集群中节点的故障切换不会破坏可分布式应用的客户端会话。群集中的每个节点共享有关持续会话的信息，并在节点消失时接管会话。

会话复制是 mod_cluster、mod_jk、mod_proxy、ISAPI 和 NSAPI 集群提供高可用性的机制。

6.1.2. 在应用程序中启用会话复制

若要利用 JBoss EAP 高可用性(HA)功能并启用 Web 应用的集群，您必须将应用配置为可分布。如果您的应用未标记为 distributable，则永远不会分发其会话。

使应用程序可分发

1. 在应用程序的 **web.xml** 描述符文件的 **< web-app>** 标签**中添加 < distributable/> 元素** ：

   示例：可分布式应用程序的最小配置

   ```
   <?xml version="1.0"?>
   <web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                               http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
           version="3.0">

       <distributable/>

   </web-app>
   ```

2. 接下来，如果需要，修改默认的复制行为。如果要更改影响会话复制的任何值，您可以在 **应用程序的 WEB -INF/jboss-web.xml 文件中的 <jboss-web> 元素 中** 覆盖它们。对于给定元素，仅在您想要覆盖默认值时包含它。

   示例：& lt;replication-config> Values

   ```
   <jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
   http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">
     <replication-config>
       <replication-granularity>SESSION</replication-granularity>
     </replication-config>
   </jboss-web>
   ```

<replication-granularity> 参数决定复制数据的粒度。它默认为 **SESSION**，但可以设置为 **ATTRIBUTE**，以提高大多数属性保持不变的会话的性能。

<replication-granularity> 的有效值可以是 ：

- **SESSION** ：默认值。如果任何属性为脏属性，则会复制整个会话对象。如果对象引用由多个会话属性共享，则需要此策略。共享对象引用在远程节点上维护，因为整个会话都以一个单元序列化。
- **ATTRIBUTE** ：这仅适用于会话中的脏属性和某些会话数据，如最后一次访问的时间戳。

不可变会话属性
对于 JBoss EAP 7，会话更改或会话的任何可变属性访问时会触发会话复制。会话属性会被假定为可变，除非以下条件之一为 true：

- 该值是一个已知的不可变值：
  - **null**
  - **java.util.Collections.EMPTY_LIST**,**EMPTY_MAP**,**EMPTY_SET**
- 值类型是或实现一个已知的不可变类型：
  - **Java.lang.Boolean, Character**,**Byte**,**Short**,**Integer**,**Long**,**Float**,**Double**
  - **java.lang.Class**、**num**、**StackTraceElement**、**String**
  - **java.io.File**,**java.nio.file.Path**
  - **java.math.BigDecimal**, **BigInteger**, **MathContext**
  - **java.net.Inet4Address**, **Inet6Address**, **InetSocketAddress**, **URI**, **URL**
  - **java.security.Permission**
  - **java.util.Currency**,**Locale**,**TimeZone**,**UUID**
  - **Java.time.Clock**,**Duration**,**Instant**,**LocalDate**,**LocalDateTime**,**LocalTime**,**MonthDay**,**Period**,**Year**,**YearMonth**,**ZoneId**,**ZoneOffset**,**ZonedDateTime**
  - **java.time.chrono.ChronoLocalDate**, **Chronology**, **Era**
  - **java.time.format.DateTimeFormatter**, **DecimalStyle**
  - **java.time.temporal.TemporalField**, **TemporalUnit**, **ValueRange**, **WeekFields**
  - **java.time.zone.ZoneOffsetTransition**, **ZoneOffsetTransitionRule**, **ZoneRules**
- 值类型带有以下注解：
  - **@org.wildfly.clustering.web.annotation.Immutable**
  - **@net.jcip.annotations.Immutable**

6.1.3. 会话属性 marshalling

通过减少通过网络发送的字节数或保留到存储的字节数，最小化单个会话属性的复制或持久性有效负载可能会直接提高性能。通过使用 web 应用程序，您可以使用以下方法优化会话属性的总结：

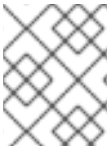
- 您可以自定义 Java Development Kit(JDK)序列化逻辑。
- 您可以实施自定义外部化器。

externalizer 实施 **org.wildfly.clustering.marshalling.Externalizer** 接口，用于指定类的 marshalling。外部化器不仅会直接从对象读取或写入输入/输出流，还执行以下操作：

- 允许应用程序在未实现 **java.io.Serializable**的会话中存储对象
- 消除序列化对象类描述符及其状态的需要

示例

```
public class MyObjectExternalizer implements
org.wildfly.clustering.marshalling.Externalizer<MyObject> {

    @Override
    public Class<MyObject> getTargetClass() {
        return MyObject.class;
    }

    @Override
    public void writeObject(ObjectOutput output, MyObject object) throws IOException {
        // Write object state to stream
    }

    @Override
    public MyObject readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
        // Construct and read object state from stream
        return ...;
    }
}
```



注意

服务加载器机制会在部署过程中动态加载外部化器。实施必须在名为 **/META-INF/services/org.wildfly.clustering.marshalling.Externalizer** 的文件中进行枚举。

6.2. HTTP 会话加密和激活

6.2.1. 关于 HTTP 会话清理和激活

传递是 控制内存使用情况的过程，方法是从内存中移除相对未使用的会话，同时将其存储在持久存储中。

激活 是在从持久存储中检索到传递的数据并重新放入内存中时。

在 HTTP 会话生命周期的不同时间进行传递：

- 当容器请求创建新会话时，如果当前活动会话的数量超过可配置的限制，服务器会尝试释放一些会话以便为新会话腾出空间。
- 部署 Web 应用并且新部署 Web 应用会话管理器获取在其他服务器上活跃的会话备份副本时，可能会传递会话。

如果活跃会话的数量超过可配置的最大值，则将传递会话。

会话始终使用最早使用(LRU)算法进行传递。

6.2.2. 在应用程序中配置 HTTP Session Passivation

HTTP 会话传递是在应用的 **WEB-INF/jboss-web.xml** 和 **META-INF/jboss-web.xml 文件中配置** 的。

示例： jboss-web.xml 文件

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">

  <max-active-sessions>20</max-active-sessions>
</jboss-web>
```

<max-active-sessions> 元素指定允许的最大活跃会话数，用于启用会话引用。如果会话创建将导致活跃会话数超过 **<max-active-sessions>**，则会话管理器已知的最旧的会话将传递，以便为新会话腾出空间。

注意

内存中的会话总数包括从此节点上未访问的其他群集节点复制的会话。设置 **<max-active-sessions>** 时 请考虑这一点。从其他节点复制的会话数量还取决于是否启用了 **REPL** 或 **DIST** 缓存模式。在 In **REPL** 缓存模式中，每一会话复制到每个节点。在 **DIST** 缓存模式中，每个会话仅复制到由 owner **参数** 指定的节点数。有关配置会话 缓存模式 的信息，请参阅 JBoss EAP *配置指南* 中的缓存模式。例如，考虑一个 8 节点集群，其中每个节点处理来自 100 个用户的请求。使用 **REPL** 缓存模式时，每个节点都会将 800 个会话存储在内存中。启用 **DIST** 缓存模式后，每个节点的默认 **所有者** 设置将 200 **个** 会话存储在内存中。

6.3. 集群服务的公共 API

JBoss EAP 7 引入了一种经过优化的公共集群 API，供应用程序使用。新服务设计为轻量、可注入且无外部依赖项。

org.wildfly.clustering.group.Group

组服务提供了一种机制，可用于查看 JGroups 频道的集群拓扑，并在拓扑更改时获得通知。

```
@Resource(lookup = "java:jboss/clustering/group/channel-name")
private Group channelGroup;
```

org.wildfly.clustering.dispatcher.CommandDispatcher

CommandDispatcherFactory 服务提供创建分配程序的机制，用于在群集的节点上执行命令。生成的 **CommandDispatcher** 是命令特征模拟以前的 JBoss EAP 版本中基于事件的 **GroupRpcDispatcher**。

```
@Resource(lookup = "java:jboss/clustering/dispatcher/channel-name")
private CommandDispatcherFactory factory;

public void foo() {
    String context = "Hello world!";
    // Exclude node1 and node3 from the executeOnCluster
    try (CommandDispatcher<String> dispatcher = this.factory.createCommandDispatcher(context))
{
        dispatcher.executeOnGroup(new StdOutCommand(), node1, node3);
```

```
    }
}

public static class StdOutCommand implements Command<Void, String> {
    @Override
    public Void execute(String context) {
        System.out.println(context);
        return null;
    }
}
```

6.4. HA 单例服务

群集单例服务也称为高可用性(HA)单例，是在群集的多个节点上部署的服务。该服务仅在其中一个节点上提供。运行单例服务的节点通常称为 *master* 节点。

当 *master* 节点出现故障或关闭时，将从剩余的节点中选择另一个 master，并在新 master 上重新启动该服务。除了一个 master 已停止并且另一个主机尚未接管时的简短间隔之外，该服务由一个（仅一个）节点提供。

HA Singleton ServiceBuilder API

JBoss EAP 7 引入了一个新的公共 API，用于构建单例服务，显著简化流程。

SingletonServiceConfigurator 实施会安装其服务，以便异步启动，防止 Modular Service Container(MSC)死锁。

HA Singleton Service Election 策略

如果哪个节点应启动 HA 单例，您可以在 **ServiceActivator** 类中设置选择策略。

JBoss EAP 提供两种选择策略：

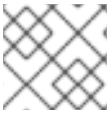
- 简单选择策略
  简单的选择策略会根据相对年龄选择 master 节点。所需的年龄在 location 属性中配置，这是可用节点列表中的索引，其中：
  - 位置 = 0 - 代表最旧的节点.这是默认值。
  - 位置 = 1 - 是指第 2 个最早的位置，以此类推。

  位置也可能为负数，表示最年轻的节点。
  - 位置 = -1 - 是指最年轻的节点.
  - 位置 = -2 - 是指第二大节点，以此类推。
- 随机选择策略
  随机选择策略选择随机成员作为单例服务的提供商。

HA 单例服务首选项

HA 单例服务选择策略可以选择性地指定一个或多个首选服务器。此首选服务器在可用时将成为该策略下所有单例应用的主服务器。

您可以通过节点名称或通过出站套接字绑定名称来定义首选项。



注意

节点首选项总是优先于选择策略的结果。

默认情况下，JBoss EAP 高可用性配置提供一个简单的选择策略，名为 **default**，无首选服务器。您可以通过创建自定义策略和定义首选服务器来设置首选项。

仲裁

当存在网络分区时，单例服务可能出现潜在问题。在这种情况下，也称脑裂情景，节点子集无法互相通信。每组服务器都将另一组中的服务器视为故障，并作为存活的群集继续工作。这可能会导致数据一致性问题。

JBoss EAP 允许您在选择策略中指定仲裁，以防止出现脑裂情形。在进行单例供应商选择前，仲裁指定要存在的最少节点数。

典型的部署场景使用 N/2 + 1 的仲裁，其中 N 是预期的集群大小。这个值可以在运行时更新，并且会立即影响任何活跃的单例服务。

HA Singleton Service Election Listener

选择新的主单例服务提供商后，会触发任何注册的 **SingletonElectionListener**，向群集的每个成员通知新主提供程序。以下示例演示了 **SingletonElectionListener** 的用法：

```
public class MySingletonElectionListener implements SingletonElectionListener {
    @Override
    public void elected(List<Node> candidates, Node primary) {
        // ...
    }
}

public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        String containerName = "foo";
        SingletonElectionPolicy policy = new MySingletonElectionPolicy();
        SingletonElectionListener listener = new MySingletonElectionListener();
        int quorum = 3;
        ServiceName name = ServiceName.parse("my.service.name");
        // Use a SingletonServiceConfiguratorFactory backed by default cache of "foo" container
        Supplier<SingletonServiceConfiguratorFactory> factory = new
ActiveServiceSupplier<SingletonServiceConfiguratorFactory>(context.getServiceRegistry(),
ServiceName.parse(SingletonDefaultCacheRequirement.SINGLETON_SERVICE_CONFIGURATOR_
FACTORY.resolve(containerName)));
        ServiceBuilder<?> builder = factory.get().createSingletonServiceConfigurator(name)
            .electionListener(listener)
            .electionPolicy(policy)
            .requireQuorum(quorum)
            .build(context.getServiceTarget());
        Service service = new MyService();
        builder.setInstance(service).install();
    }
}
```

创建 HA 单例服务应用

以下是作为集群范围的单例创建和部署应用所需的步骤的缩写示例。本例演示了一个查询服务，它会定期查询单例服务来获取运行它的节点的名称。

要查看单例行为，您必须将应用部署到至少两台服务器。无论单例服务是否在同一节点上运行，还是远程获取该值，都是透明的。

1. 创建 **SingletonService** 类。**getValue（）** 方法由查询服务调用，提供有关在其上运行的节点的信息。

```
class SingletonService implements Service {
    private Logger LOG = Logger.getLogger(this.getClass());
    private Node node;

    private Supplier<Group> groupSupplier;
    private Consumer<Node> nodeConsumer;

    SingletonService(Supplier<Group> groupSupplier, Consumer<Node> nodeConsumer) {
      this.groupSupplier = groupSupplier;
      this.nodeConsumer = nodeConsumer;
    }

    @Override
    public void start(StartContext context) {
        this.node = this.groupSupplier.get().getLocalMember();

        this.nodeConsumer.accept(this.node);

        LOG.infof("Singleton service is started on node '%s'.", this.node);
    }

    @Override
    public void stop(StopContext context) {
        LOG.infof("Singleton service is stopping on node '%s'.", this.node);

        this.node = null;
    }
}
```

2. 创建查询服务。它调用单例服务的 **getValue（）** 方法以获取运行它的节点的名称，然后将结果写入服务器日志。

```
class QueryingService implements Service {
    private Logger LOG = Logger.getLogger(this.getClass());
    private ScheduledExecutorService executor;

    @Override
    public void start(StartContext context) throws {
        LOG.info("Querying service is starting.");

        executor = Executors.newSingleThreadScheduledExecutor();
        executor.scheduleAtFixedRate(() -> {

        Supplier<Node> node = new PassiveServiceSupplier<>
(context.getController().getServiceContainer(),
SingletonServiceActivator.SINGLETON_SERVICE_NAME);
         if (node.get() != null) {
            LOG.infof("Singleton service is running on this (%s) node.", node.get());
         } else {
            LOG.infof("Singleton service is not running on this node.");
```

```
        }

        }, 5, 5, TimeUnit.SECONDS);
    }

    @Override
    public void stop(StopContext context) {
        LOG.info("Querying service is stopping.");

        executor.shutdown();
    }
}
```

3. 实施 **SingletonServiceActivator** 类，以构建和安装单例服务和查询服务。

```
public class SingletonServiceActivator implements ServiceActivator {

    private final Logger LOG = Logger.getLogger(SingletonServiceActivator.class);

    static final ServiceName SINGLETON_SERVICE_NAME =
            ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service");
    private static final ServiceName QUERYING_SERVICE_NAME =
            ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service.querying");

    @Override
    public void activate(ServiceActivatorContext serviceActivatorContext) {
        SingletonPolicy policy = new ActiveServiceSupplier<SingletonPolicy>(
                serviceActivatorContext.getServiceRegistry(),
                ServiceName.parse(SingletonDefaultRequirement.POLICY.getName())).get();

        ServiceTarget target = serviceActivatorContext.getServiceTarget();
        ServiceBuilder<?> builder =
policy.createSingletonServiceConfigurator(SINGLETON_SERVICE_NAME).build(target);
        Consumer<Node> member = builder.provides(SINGLETON_SERVICE_NAME);
        Supplier<Group> group =
builder.requires(ServiceName.parse("org.wildfly.clustering.default-group"));
        builder.setInstance(new SingletonService(group, member)).install();

        serviceActivatorContext.getServiceTarget()
                .addService(QUERYING_SERVICE_NAME, new QueryingService())
                .setInitialMode(ServiceController.Mode.ACTIVE)
                .install();


serviceActivatorContext.getServiceTarget().addService(QUERYING_SERVICE_NAME).setInst
ance(new QueryingService()).install();

        LOG.info("Singleton and querying services activated.");
    }
}
```

4. 在 **META-INF/services/** 目录中创建一个名为 **org.jboss.msc.service.ServiceActivator** 的文件，其中包含 **ServiceActivator** 类的名称，例如 **org.jboss.as.quickstarts.ha.singleton.service.SingletonServiceActivator**。

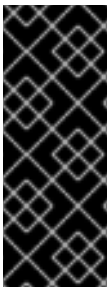
请参阅 JBoss EAP 随附的 **ha-singleton-service** 快速入门，以获取完整的工作示例。此快速入门还提供了第二个示例，它演示了安装有备份服务的单例服务。备份服务在所有不选择运行单例服务的节点中运行。最后，此快速入门还演示了如何配置几个不同的选择策略。

6.5. HA 单例部署

您可以将应用程序部署为单例部署。当部署到一组集群服务器时，单例部署仅在任何给定时间部署到单个节点上。如果部署活跃状态的节点停止或失败，则部署会在另一节点上自动启动。

在以下情况下，可以在多个节点上部署单例部署：

- 由于配置问题或网络问题，给定节点上的一组集群服务器无法建立连接。
- 使用非 HA 配置，如以下配置文件：
  - 一个 **standalone.xml** 配置，它支持 Jakarta EE 8 Web 配置文件或 **standalone-full.xml** 配置，支持 Jakarta EE 8 Full Platform 配置文件。
  - **domain.xml** 配置，由默认域配置文件或全域配置文件组成。



重要

默认情况下，非 HA 配置不启用 singleton 子系统。如果您使用此默认配置，则忽略 **singleton-deployment.xml** 文件，以推进应用部署成功。

不过，使用非 HA 配置可能会导致 **jboss-all.xml** 描述符文件出现错误。为避免这些错误，请将单个部署添加到 **singleton-deployment.xml** 描述符中。然后，您可以使用任何配置集类型部署应用。

控制 HA 单例行为的策略由新的 单 例子系统管理。部署可以指定特定的单例策略，也可以使用默认的子系统策略。

部署使用 **META-INF/singleton-deployment.xml** 部署描述符将自身识别为单例部署，该描述符作为部署覆盖应用到现有部署中。此外，必要的单例配置也可嵌入到现有的 **jboss-all.xml** 文件中。

定义或选择单例部署
要将部署定义为单例部署，请在应用存档 **中包含 META-INF/singleton-deployment.xml** 描述符。

如果 Maven WAR 插件已存在，您可以将插件迁移到 **META-INF** 目录：** **/src/main/webapp/META-INF**。

流程

- 如果应用部署在 EAR 文件中，请将位于 **jboss -all.xml 文件中的 singleton-deployment.xml 描述符或单例部署** 元素移到 **META-INF** 目录的顶级。

  示例：单例部署描述符

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
  ```

- 要将应用部署添加为 WAR 文件或 JAR 文件，请将 **singleton-deployment.xml** 描述符移到应用存档中的 /**META-INF** 目录的顶级。

  示例：带有特定单例策略的单例部署描述符

```
<?xml version="1.0" encoding="UTF-8"?>
<singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-deployment:1.0"/>
```

- 可选：要在 **jboss -all.xml 文件中定义单例部署**，请将 **jboss-all.xml** 描述符移到应用存档中的 **/META-INF** 目录的顶级。

  示例：定义 jboss -all.xml 中的单例部署

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss xmlns="urn:jboss:1.0">
   <singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
</jboss>
```

- 可选：使用单例策略在 **jboss -all.xml 文件中定义 singleton- deployment**。将 **jboss-all.xml** 描述符移到应用存档中 **/META-INF** 目录的顶级。

  示例：使用特定 的单例策略定义 jboss-all.xml 中的单例部署

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss xmlns="urn:jboss:1.0">
   <singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-
deployment:1.0"/>
</jboss>
```

创**建单例部署**

JBoss EAP 提供两种选择策略：

- 简单选择策略

  simple-election-policy 选择要在其上部署给定应用的特定成员（通过 location **属性** 表示）。**位置** 属性决定从按照降序年龄排序的候选列表中选择节点索引，其中 **0** 代表节点最旧的，**1** 表示第二个最旧的节点，-1 **表示节点最年轻，-2 则表示** 第二个较年轻的节点，以此类推。如果指定的位置超过候选项的数量，则应用模态操作。

  示例：使用 管理 CLI 创建一个带有 简单选择策略 和位置集的新单例策略

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-
policy=simple:add(position=-1)
run-batch
```

注意

要将新创建的策略 **my-new-policy** 设置为默认值，请运行这个命令：

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

示例：使用 standalone -ha.xml 配置一个简单选择策略，将位置设置为 -1

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-new-policy">
    <singleton-policy name="my-new-policy" cache-container="server">
```

```
        <simple-election-policy position="-1"/>
      </singleton-policy>
    </singleton-policies>
  </subsystem>
```

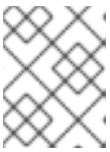
- 随机选择策略
  random-election-policy 选择要在其上部署给定应用的随机成员。

  示例：使用管理 CLI 创建带有 随机选择策略 的新单例策略

```
batch
/subsystem=singleton/singleton-policy=my-other-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-other-new-policy/election-policy=random:add()
run-batch
```

  示例：使用 standalone -ha.xml 配置随机选择策略

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```



注意

在尝试添加策略之前，需要定义 cache-container 的 default-cache 属性。如果不这样做，如果您使用的是自定义缓存容器，您可能会得到错误消息。

首选项
此外，任何单例选择策略都可指明群集的一个或多个成员的首选。首选项可以使用节点名称或使用出站套接字绑定名称来定义。节点首选项总是接管选择策略的结果。

示例：使用管理 CLI 在现有单例策略中指定首选项

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)

/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-
preferences, value=binding1)
```

示例：使用管理 CLI，使用 simple-election-policy 和 name-preferences 创建一个新的单例策略

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-policy=simple:add(name-preferences=
[node1, node2, node3, node4])
run-batch
```

注意

要将新创建的策略 **my-new-policy** 设置为默认值，请运行这个命令：

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

示例：使用 standalone -ha.xml 为 socket-binding-preferences 配置随机选择策略

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy>
        <socket-binding-preferences>binding1 binding2 binding3 binding4</socket-binding-
preferences>
      </random-election-policy>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

定义仲裁
在单例部署中，网络分区尤其有问题，因为它们可以触发多个单例供应商来同时运行同一部署。为了防止这种情况，单例策略可以定义一个仲裁，在单例供应商选择前需要存在最少的节点数。典型的部署场景使用 N/2 + 1 的仲裁，其中 N 是预期的集群大小。这个值可以在运行时更新，并使用对应的单例策略立即影响任何单例部署。

示例：在 standalone-ha.xml 文件中 仲裁

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="default">
    <singleton-policy name="default" cache-container="server" quorum="4">
      <simple-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

示例：使用管理 CLI 仲裁

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

有关将应用中打包 为**使用单例部署的单例的完整工作示例，**请参见 **JBoss EAP 附带的 ha-singleton-deployment** 快速入门。

使用 CLI 确定主要单例服务提供商
singleton 子系统为从特定单例策略创建的每一个单例部署或服务公开运行时资源。这有助于您使用 CLI 确定主要单例提供程序。

您可以查看当前作为单例供应商的集群成员的名称。例如：

```
/subsystem=singleton/singleton-policy=default/deployment=singleton.jar:read-
attribute(name=primary-provider)
{
```

```
    "outcome" => "success",
    "result" => "node1"
}
```

您还可以查看在其上安装单例部署或服务的节点的名称。例如：

```
/subsystem=singleton/singleton-policy=default/deployment=singleton.jar:read-
attribute(name=providers)
{
    "outcome" => "success",
    "result" => [
        "node1",
        "node2"
    ]
}
```

6.6. APACHE MOD_CLUSTER-MANAGER 应用程序

6.6.1. 关于 mod_cluster-manager 应用程序

mod_cluster-manager 应用是一个管理网页，可在 Apache HTTP 服务器中使用。它用于监控连接的 worker 节点并执行各种管理任务，如启用或禁用上下文，以及在集群中配置 worker 节点的负载平衡属性。

探索 mod_cluster-manager 应用
mod_cluster-manager 应用可用于在工作程序节点上执行各种管理任务。

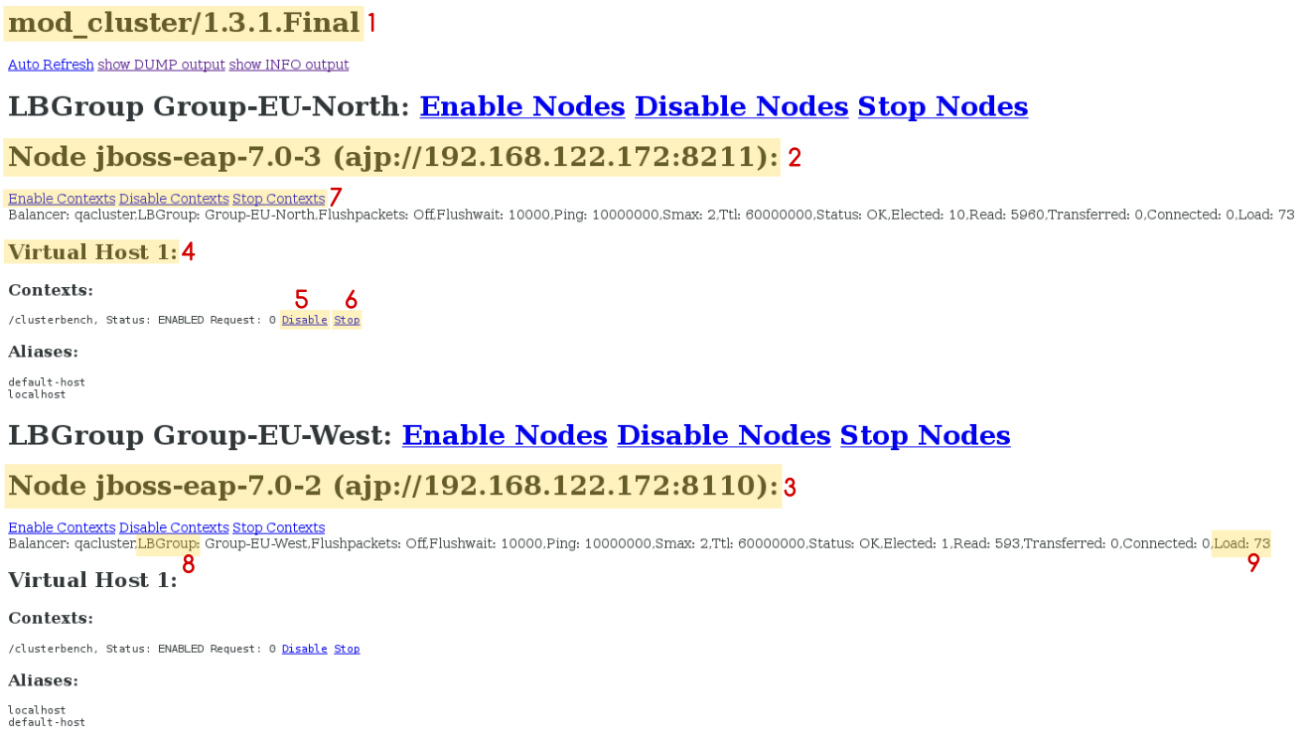


图 - mod_cluster 管理网页

- [1] **mod_cluster/1.3.1.Final** ：mod_cluster 原生库的版本。
- [2] AJP **://192.168.122.204:8099** ：使用的协议（如 AJP、HTTP 或 HTTPS）、工作程序节点的主机名或 IP 地址，以及端口。

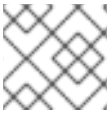
- [3] **JBoss-eap-7.0-2** ：工作程序节点的 JVMRoute。
- [4] **虚拟主机 1**：在 worker 节点上配置的虚拟主机。
- [5] disable：一个管理选项，可用于 **禁用** 在特定上下文上创建新会话。但是，持续会话不会被禁用，也不会保持完好。
- [6] **stop** ：一个管理选项，可用于停止将会话请求路由到上下文。剩余的会话将故障转移到另一节点，除非 **sticky-session-force** 属性设为 **true**。
- [7] **启用上下文禁用上下文停止上下文** ：可在整个节点上执行的操作。选择其中一个选项会影响其所有虚拟主机中节点的所有上下文。
- [8] **负载平衡组(LBGroup)** ： **负载平衡-group** 属性在 JBoss EAP 配置中的 **modcluster** 子系统中设置，将所有工作程序节点分组到自定义负载平衡组中。负载平衡组(LBGroup)是一个信息字段，提供关于所有集合负载平衡组的信息。如果没有设置此字段，则所有 worker 节点都分组到一个默认负载均衡组中。

注意

这只是一个信息字段，因此无法用于设置 **负载平衡-group** 属性。该属性必须在 JBoss EAP 配置的 **modcluster** 子系统中设置。

- [9] **负载（值）** ：worker 节点上的负载因子。负载因素的评估如下：

> -load > 0 : A load factor with value 1 indicates that the worker node is overloaded. A load factor of 100 denotes a free and not-loaded node.
> -load = 0 : A load factor of value 0 indicates that the worker node is in standby mode. This means that no session requests will be routed to this node until and unless the other worker nodes are unavailable.
> -load = -1 : A load factor of value -1 indicates that the worker node is in an error state.
> -load = -2 : A load factor of value -2 indicates that the worker node is undergoing CPing/CPong and is in a transition state.



注意

对于 JBoss EAP 7.4，也可以使用 Undertow 作为负载平衡器。

6.7. 分布式 WEB 会话配置的 DISTRIBUTABLE-WEB 子系统

distributable-web 子系统可促进灵活且可分布的 Web 会话配置。子系统定义一组可分布式 Web 会话管理配置文件。这些配置文件中的一个被指定为默认配置集。它定义可分布式 Web 应用的默认行为。例如：

```
[standalone@embedded /] /subsystem=distributable-web:read-attribute(name=default-session-management)
{
    "outcome" => "success",
    "result" => "default"
}
```

默认会话管理将 Web 会话数据存储在 Infinispan 缓存中，如下例所示：

```
[standalone@embedded /] /subsystem=distributable-web/infinispan-session-
```

```
management=default:read-resource
{
    "outcome" => "success",
    "result" => {
        "cache" => undefined,
        "cache-container" => "web",
        "granularity" => "SESSION",
        "affinity" => {"primary-owner" => undefined}
    }
}
```

本例中使用的属性和可能的值有：

- **缓存** ：关联的 cache-container 中的缓存。Web 应用的缓存基于此缓存的配置。如果未定义，则使用相关缓存容器的默认缓存。
- **cache-container** ：Infinispan **子系统** 中定义的 cache-container，将会话数据存储到其中。
- **粒度** ：定义会话管理器如何将会话映射到单个缓存条目。可能的值有：
  - **SESSION** ：将所有会话属性存储在单个缓存条目中。比 **ATTRIBUTE** 粒度更高，但会保留任何跨目录对象引用。
  - **ATTRIBUTE** ：将各个会话属性存储在单独的缓存条目中。比 **SESSION** 粒度更高，但不会保留任何跨目录对象引用。
- **affinity** ：定义 Web 请求必须针对服务器具有的关联性。关联的 Web 会话的关联性决定了生成路由以附加到会话 ID 的算法。可能的值有：
  - **affinity=none** ：Web 请求没有与任何节点的关联性。如果应用服务器内不维护 Web 会话状态，可使用此选项。
  - **affinity=local** ：Web 请求与最后处理会话请求的服务器关联。此选项与粘性会话行为对应。
  - **affinity=primary-owner** ：Web 请求与会话的主所有者关联。这是这个分布式会话管理器的默认关联性。如果不分发或复制后备缓存，其行为与 **affinity=local** 相同。
  - **affinity=ranked：** Web 请求对列表中的第一个可用成员具有关联性，其中包括主用户和备份所有者，以及最后处理会话的成员的关联。
  - **affinity=ranked 分隔符** ：用于在编码会话标识符中分隔单个路由的分隔符。
  - **affinity=ranked max 路由** ：编码到会话标识符的最大路由数。

您必须在负载均衡器中启用等级会话关联，才能使会话与多个排序的路由关联。如需更多信息，请参阅 JBoss EAP *配置指南* 中的负载平衡器中启用等级的会话关联性。

您可以通过按名称引用会话管理配置文件或提供特定于部署的会话管理配置来覆盖默认的可分布式会话管理行为。如需更多信息，请参阅 覆盖默认分布式会话管理行为。

6.7.1. 在远程 Red Hat Data Grid 中存储 Web 会话数据

distributable-web 子系统可以配置为使用 HotRod 协议将 Web 会话数据存储在远程红帽数据网格群集中。在远程集群中存储 Web 会话数据可让缓存层独立于应用服务器进行扩展。

配置示例：

```
[standalone@embedded /]/subsystem=distributable-web/hotrod-session-
management=ExampleRemoteSessionStore:add(remote-cache-container=datagrid, cache-
configuration=__REMOTE_CACHE_CONFIG_NAME__, granularity=ATTRIBUTE)
{
    "outcome" => "success"
}
```

本例中使用的属性和可能的值有：

- **remote-cache-container** ：Infinispan 子系统 中定义的远程缓存容器，用于存储 Web 会话数据。
- **缓存配置** ：红帽数据网格集群中的缓存配置名称.新创建的特定于部署的缓存基于此配置。如果没有找到与该名称匹配的远程缓存配置，则会在远程容器中创建新的缓存配置。
- **粒度** ：定义会话管理器如何将会话映射到单个缓存条目。可能的值有：
  - **SESSION** ：将所有会话属性存储在单个缓存条目中。比 **ATTRIBUTE** 粒度更高，但会保留任何跨目录对象引用。
  - **ATTRIBUTE** ：将各个会话属性存储在单独的缓存条目中。比 **SESSION** 粒度更高，但不会保留任何跨目录对象引用。

第 7 章 JAKARTA 上下文和依赖注入

7.1. JAKARTA 上下文和依赖注入简介

7.1.1. 关于 Jakarta 上下文和依赖注入

Jakarta Contexts 和 Dependency Injection 2.0 是一种规范，旨在使 Jakarta 企业 Bean 3 能够用作 Jakarta 服务器 Faces 管理 Bean。Jakarta Contexts 和 Dependency Injection 统一了两种组件模型，并可显著简化 Java 中基于 Web 的应用程序的编程模型。有关 Jakarta 上下文和依赖注入 2.0 的详细信息，请参阅 Jakarta 上下文和依赖注入 2.0 规范。

JBoss EAP 包括 Weld，它是一个 Jakarta 上下文和依赖注入 2.0 兼容规范。

注意

weld 是 Jakarta EE 平台的 Jakarta Contexts 和 Dependency Injection 的兼容实施。Jakarta Contexts 和 Dependency Injection 是一种 Jakarta EE 标准，适用于依赖注入和上下文生命周期管理。此外，雅加达上下文和依赖注入是 Jakarta EE 中最重要的部分之一。

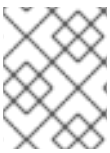
Jakarta 上下文和依赖注入的好处
Jakarta 上下文和依赖注入的好处包括：

- 使用注释替换大量代码，从而简化和缩小代码库。
- 灵活性，允许您禁用和启用注入和事件、使用备用 Bean 以及轻松注入非文本和依赖注入对象。
- 另外，如果您需要自定义配置，使其与默认值不同，允许您在 **META -INF/ 或 WEB-INF/** 目录中包含 **beans.xml** 文件。该文件可以为空。
- 简化打包和部署，减少您在部署中需要添加的 XML 数量。
- 通过上下文提供生命周期管理.您可以将注入与请求、会话、对话或自定义上下文相关联。
- 提供 type-safe 依赖项注入，与基于字符串的注入相比，它更安全且更易于调试。
- 将拦截器与 Bean 分离.
- 提供复杂事件通知.

7.2. 使用 JAKARTA 上下文和依赖注入来开发应用程序

Jakarta Contexts 和 Dependency Injection 在开发应用程序、重用代码、在部署或运行时修改代码以及单元测试方面为您提供巨大的灵活性。

我们ld 为应用程序开发提供了一个特殊模式。启用后，可以使用某些内置工具，协助 Jakarta Contexts 和 Dependency Injection 应用程序的开发。



注意

开发模式不应在生产环境中使用，因为它可能会对应用的性能造成负面影响。在部署到生产环境之前，务必禁用开发模式。

为 **Web 应用程序启用开发模式：**

对于 web 应用程序，将 servlet 初始化参数 **org.jboss.weld.development** 设置为 **true** ：

```
<web-app>
  <context-param>
    <param-name>org.jboss.weld.development</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

使用管理 CLI 为 JBoss EAP 启用开发模式：

通过将 **development-mode** 属性设置为 **true**，可以为部署的所有应用程序全局启用 Weld 开发模式：

```
/subsystem=weld:write-attribute(name=development-mode,value=true)
```

7.2.1. 默认 Bean 发现模式

Bean 存档的默认 bean 发现模式被 **标注**。此类 Bean 存档 **被认为是隐式 Bean 存档**。

如果 **注解** 了 bean 发现模式，则：

- 不会发现没有 **bean 定义注解** 且不是 Bean 类会话 Bean 类的 Bean 类。
- 不在会话 Bean 上以及 bean 类没有 Bean 定义注释的制作者方法不会发现。
- 不在会话 Bean 上以及 bean 类没有 Bean 定义注解的制作者字段不会被发现。
- 不在会话 Bean 上且 bean 类没有 Bean 定义注解的推理器方法不会被发现。
- 不在会话 Bean 上以及 bean 类没有 Bean 定义注解的观察方法不会发现。

重要

只有将发现模式设置为 **all** 时，Contexts 和 Dependency Injection 部分中的所有示例才有效。

Bean 定义注解
Bean 类可以具有一个 **定义注解的** Bean，以便能够将其放置在应用程序中的任何位置，如 bean 存档中所定义。Bean 类中带有 Bean 定义注解，这被认为是隐式 Bean。

Bean 定义注解集合包含：

- **@ApplicationScoped**、**@SessionScoped**、**@ConversationScoped** 和 **@RequestScoped** 注释.
- 所有其他普通范围类型.
- **@interceptor** 和 **@Decorator** 注释.
- 所有 stereotype 注释，即标有 **@Stereotype 的** 注释。
- **@Dependent** 范围注释.

如果在 Bean 类上声明其中一个注释，则 bean 类被认为具有 Bean 定义注释。

示例：Bean 定义注解

```
@Dependent
public class BookShop
      extends Business
      implements Shop<Book> {
   ...
}
```

注意

为确保与其他 JSR-330 实施以及 Jakarta 上下文和依赖注入规范兼容，除 **@Dependent** 外，所有伪作用注释都不是定义注释。但是，stereotype 注释（包括伪范围注释）是 Bean 定义注解。

7.2.2. 从扫描过程中排除 Bean

排除过滤器由 bean 归档的 **beans.xml** 文件中的 **<exclude> 元素定**义为 **<scan> 元素的** 子项。默认情况下，排除过滤器处于活动状态。如果 exclude 过滤器包含以下内容，则 exclude 过滤器将变为不活跃：

- 名为 **<if-class-available>** 的子元素及 **name** 属性，并且 bean 存档的类加载器无法为该名称加载类，或者
- 名为 **<if-class-not-available>** 的子元素及 **name** 属性，bean 存档的类加载器可以为该名称加载类，或者
- 名为 **<if-system-property>** 且带有 **name** 属性的子元素，没有为该名称定义系统属性，或者
- 名为 **<if-system-property>** 的子元素带有 **name** 属性和 value 属性，没有为该名称定义具有该值的系统属性。

如果过滤器活跃，则该类型不在发现中，且：

- 被发现的类型的完全限定名称与 exclude 过滤器的 name 属性的值匹配，或者
- 被发现类型的软件包名称与 exclude 过滤器的 name 属性的值匹配，或者后缀为 ".*"。
- 被发现类型的软件包名称以 exclude 过滤器的 name 属性值开头

例 7.1. 示例： beans.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">

  <scan>
    <exclude name="com.acme.rest.*" /> (1)

    <exclude name="com.acme.faces.**"> (2)
      <if-class-not-available name="javax.faces.context.FacesContext"/>
    </exclude>

    <exclude name="com.acme.verbose.*"> (3)
      <if-system-property name="verbosity" value="low"/>
    </exclude>
```

```
        <exclude name="com.acme.ejb.**"> 4
            <if-class-available name="javax.enterprise.inject.Model"/>
            <if-system-property name="exclude-ejbs"/>
        </exclude>
    </scan>

</beans>
```

1. 第一个排除过滤器将排除 **com.acme.rest** 软件包中的所有类。
2. 第二个排除过滤器将排除 **com.acme.faces** 软件包中的所有类，以及任何子软件包，但仅当 Jakarta Server Faces 不可用时。
3. 如果系统属性 详细 **程度较低**，则第三个排除过滤器将排除 **com.acme.verbose** 软件包中的所有类。
4. 如果系统属性 **exclude- ejbs** 设置了任何值，并且同时 **javax.enterprise.inject.Model** 类也可用于类加载器，则第四个排除过滤器将排除 **com.acme. ej b** 软件包中的所有类，以及任何子软件包。

注意

使用 **@Vetoed** 标注 Jakarta EE 组件是安全的，防止它们被视为 Bean。事件不会触发标有 **@Vetoed 的** 任何类型，也不会在标有 **@Vetoed** 的软件包中触发。如需更多信息，请参见 **@Vetoed**。

7.2.3. 使用注入来扩展实施

您可以使用注入来添加或更改现有代码的功能。

以下示例为现有类添加了转换功能，并假定您已有一个 **Welcome** 类，它具有方法 **buildPhrase**。**buildPhrase** 方法将城市名称用作参数，并输出一个短语，如"Welcome to Boston!"。

这个示例将假设的转换 **器** 对象注入到 Welcome 类**中。Translator 对象可以是 Jakarta Enterprise Beans 无状态 Bean 或其他类型的 bean，它可以将句子从一个语言转换为另一种语言。在这个示例中，转换 器用于 转译整个问候语，而不修改原始 Welcome 类。Translator 在调用 buildPhrase 方法之前注入。**

示例：将一个翻译器 Bean 注入 欢迎 类

```
public class TranslatingWelcome extends Welcome {

    @Inject Translator translator;

    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
```

```
    }
    ...
}
```

7.3. 不明确或不满意的依赖关系

当容器无法将注入解析为一个 Bean 时，存在不确定的依赖关系。

当容器无法将注入解析到任何 Bean 时，存在不满意的依赖关系。

容器执行以下步骤来尝试解决依赖项：

1. 它解决了所有实现注入点 Bean 类型的限定注解。
2. 它过滤掉禁用的 Bean。禁用的 Bean 是 @Alternative bean，没有被显式启用。

如果依赖项模糊或不满意，容器将中止部署并引发异常。

要修复模糊的依赖关系，请参阅使用 Qualifier 来解析 Ambiguous Injection。

7.3.1. 限定符

限定符是注解，用于避免在容器可以解析多个 Bean 时产生模糊的依赖关系，它们适合注入点。在注入点上声明的限定符提供一组合格的 Bean，后者声明相同的限定符。

限定条件必须以保留和目标声明，如下例所示。

示例：定义 @Synchronous 和 @Asynchronous Qualifiers

```
@Qualifier
```

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Synchronous {}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Asynchronous {}
```

示例：使用 @Synchronous 和 @Asynchronous Qualifiers

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
  public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
  public void process(Payment payment) { ... }
}
```

'@Any'

每当 bean 或注入点没有明确声明限定符时，容器会假定限定符 @Default。您不时需要声明一个注入点而无需指定限定符。这也有限定条件。所有 beans 都具有限定符 @Any。因此，通过在注入点显式指定 @Any，您可以抑制默认限定符，而不限制有资格注入的 Bean。

当您想迭代某个 Bean 类型的所有 bean 时，这尤其有用。

```
import javax.enterprise.inject.Instance;
...

@Inject

void initServices(@Any Instance<Service> services) {

  for (Service service: services) {

    service.init();
```

```
    }
}
```

每个 bean 都具有限定符 @Any，即使它没有明确声明此限定符。

每个事件也都有限定符 @Any，即使它不是明确声明这个限定符的。

```
@Inject @Any Event<User> anyUserEvent;
```

@Any 限定符允许注入点引用所有 bean 或特定 bean 类型的所有事件。

```
@Inject @Delegate @Any Logger logger;
```

7.3.2. 使用 Qualifier 解决 Ambiguous Injection

您可以使用限定符解决模糊注入问题。阅读有关 Ambiguous 或 Unsatisfidened 依赖项 模糊注入的更多信息。

以下示例模糊不清，并且有两个 Welcome 实施， 一种是转换，另一个是没有实现。需要指定注入以使用转换 Welcome。

示例：Ambiguous Injection

```
public class Greeter {
  private Welcome welcome;

  @Inject
  void init(Welcome welcome) {
    this.welcome = welcome;
  }
  ...
}
```

使用 Qualifier 解析 Ambiguous Injection

1. 要解决模糊注入，请创建一个名为 @Translating 的限定注释：

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE,METHOD,FIELD,PARAMETERS})
public @interface Translating{}
```

2. 使用 @ Translating 注释给转换 Welcome 标注：

```
@Translating
public class TranslatingWelcome extends Welcome {
   @Inject Translator translator;
   public String buildPhrase(String city) {
      return translator.translate("Welcome to " + city + "!");
   }
   ...
}
```

3. 请求注入中的 Welcome。您必须明确请求合格的实施，类似于工厂方法模式。这种不确定性是在注入点解决的。

```
public class Greeter {
  private Welcome welcome;
  @Inject
  void init(@Translating Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

7.4. 受管 BEAN

Jakarta EE 在 Jakarta 托管 Bean 规范 中建立了通用定义。对于 Jakarta EE，受管 Bean 定义为容器管理的 Bean，具有最少的编程限制，否则被缩写为 POJO（Plain Old Java 对象）已知的。它们支持一组基本服务，如资源注入、生命周期回调和拦截器。配套规范（如 Jakarta Enterprise Beans 和 Jakarta Contexts 和 Dependency Injection）构建在这个基本模型之上。

除极少数外，几乎每个没有参数的构造器或带有注释 @Inject 的构造器都是 bean。这包括每个 JavaBean 和每个 Jakarta Enterprise Beans 会话 bean。

7.4.1. Bean 类

受管 Bean 是 Java 类。对于 Jakarta EE，受管 Bean 的基本生命周期和语义按照 Jakarta Managed

Beans 1.0 规范 进行定义。您可以通过注解 beanagedBean 类来明确声明受管 Banaged Bean，但在上下文和依赖注入中，您不需要这样做。根据规范，上下文和依赖注入容器将满足以下条件的任何类视为受管 Bean：

- 这不是一个非静态内类。
- 它是具体类，或者带有 @Decorator 标注。
- 它没有使用 Jakarta Enterprise Beans 组件定义注释，也不声明为 ejb-jar.xml 文件中的 Jakarta Enterprise Beans bean 类。
- 它不实施接口 javax.enterprise.inject.spi.Extension。
- 它具有不含参数的构造器，或者带有 @Inject 标注的构造器。
- 它不标有 @Vetoed，也不在标有 @Vetoed 的包中标注。

受管 Bean 的无限制 Bean 类型包含 Bean 类、每个超级类，以及它直接或间接实施的所有接口。

如果受管 Bean 具有公共字段，则必须具有默认范围 @Dependent。

@Vetoed

您可以对类进行处理，从而不会安装本课程定义的 beans 或观察方法：

```
@Vetoed
public class SimpleGreeting implements Greeting {
    ...
}
```

在这个代码中，SimpleGreeting Bean 不视为注入。

软件包中的所有 bean 可以被禁止注入：

```
@Vetoed
package org.sample.beans;

import javax.enterprise.inject.Vetoed;
```

此代码在 org.sample .beans 软件包中的 package-info. java 代码将阻止此软件包中的所有 bean 注入。

Jakarta EE 组件（如无状态 Jakarta Enterprise Beans 或 Jakarta RESTful Web Services 资源端点）可以标记为 @Vetoed，以防止它们被视为 bean。将 @Vetoed 注释添加到所有持久实体可防止 BeanManager 将实体作为 Jakarta 上下文和依赖注入 Bean 管理。当实体使用 @Vetoed 标注时，不会进行注入。其背后的原因是防止 BeanManager 执行可能导致 Jakarta Persistence 供应商中断的操作。

7.4.2. 使用上下文和依赖注入将对象注入 Bean

如果在应用程序中检测到上下文和依赖注入组件，则上下文和依赖注入会自动激活。如果要自定义配置使其与默认值不同，您可以在部署存档中包含 META-INF/beans.xml 文件或 WEB-INF/beans.xml 文件。

将对象注入其他对象

1. 要获取类的实例，请在 bean 中为字段标上 @Inject:

```
public class TranslateController {
  @Inject TextTranslator textTranslator;
  ...
```

2. 直接使用注入的对象方法。假设 TextTranslator 有一个方法 转换 ：

```
// in TranslateController class

public void translate() {
  translation = textTranslator.translate(inputText);
}
```

3. 在 Bean 的构造器中使用注入。您可以将对象注入到 Bean 的构造器中，作为使用工厂或服务定位器创建对象的替代方法：

```
public class TextTranslator {

  private SentenceParser sentenceParser;
  private Translator sentenceTranslator;
```

```
  @Inject
  TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
    this.sentenceParser = sentenceParser;
    this.sentenceTranslator = sentenceTranslator;
  }

  // Methods of the TextTranslator class
  ...
}
```

4. 使用 Instance(<T>) 接口以编程方式获取实例。当使用 bean 类型进行参数化时，Instance 接口可以返回 TextTranslator 的实例。

```
@Inject Instance<TextTranslator> textTranslatorInstance;
...
public void translate() {
  textTranslatorInstance.get().translate(inputText);
}
```

将对象注入 Bean 时，所有对象的方法和属性都可用于您的 Bean。如果您注入到 bean 的构造器中，则调用 bean 的构造器时会创建注入对象的实例，除非注入引用已存在的实例。例如，如果在会话生命周期内注入会话范围 Bean，则不会创建新的实例。

7.5. 上下文和范围

在上下文和依赖注入方面，上下文是一个存储区域，存放与特定范围关联的 bean 实例。

范围是 bean 和上下文之间的链接。作用域/上下文组合可以具有特定的生命周期。有一些预定义的范围，您可以自行创建。预定义的范围示例有 @RequestScoped、@SessionScoped 和 @ConversationScope。

表 7.1. 可用范围

影响范围	描述
@Dependent	Bean 绑定到持有该引用的 Bean 的生命周期。注入 Bean 的默认范围为 **@Dependent**。
@ApplicationScoped	Bean 绑定到应用程序的生命周期。
@RequestScoped	Bean 绑定到请求的生命周期。
@SessionScoped	Bean 绑定到会话的生命周期。

影响范围	描述
@ConversationScoped	Bean 受对话生命周期约束。对话范围介于请求和会话的长度之间，并由申请控制。
自定义范围	如果上述上下文不满足您的需求，您可以定义自定义范围。

7.6. 名为 BEANS

您可以使用 @Named 注释来命名 bean。通过命名 bean，您可以直接在 Jakarta Server Faces 和 Jakarta Expression 语言中使用它。

@Named 注释采用可选参数，即 bean 名称。如果省略此参数，ban 名称默认为 Ban 的类名称，其第一个字母转换为小写。

7.6.1. 使用命名 Bean

使用 @Named 注解配置 Bean 名称

1. **使用 @Named 注释，为 bean 分配名称。**

```
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;

  @Inject
  void init (Welcome welcome) {
    this.welcome = welcome;
  }

  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

在上面的示例中，如果没有指定名称，则默认名称为 greeterBean。

2. **在 Jakarta Server Faces 视图中使用名为 bean。**

```
<h:form>
  <h:commandButton value="Welcome visitors" action="#{greeter.welcomeVisitors}"/>
</h:form>
```

7.7. BEAN 生命周期

此任务显示如何在请求的生命周期内保存 Bean。

注入 Bean 的默认范围为 @Dependent。这意味着 Bean 的生命周期取决于拥有该参考的 Bean 的生命周期。还有其他一些范围，您可以定义自己的范围。如需更多信息，请参阅 上下文和范围。

管理 Bean 生命周期

1. 使用所需范围给 Bean 标注。

```
@RequestScoped
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;
  private String city; // getter & setter not shown
  @Inject   void init(Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase(city));
  }
}
```

2. 当您在 Jakarta Server Faces 视图中使用 Ban 时，它会保留 状态。

```
<h:form>
  <h:inputText value="#{greeter.city}"/>
  <h:commandButton value="Welcome visitors" action="#{greeter.welcomeVisitors}"/>
</h:form>
```

您的 bean 保存在与您指定的范围相关的上下文中，只要适用范围，则会持续。

7.7.1. 使用 Producer 方法

制作者方法 是一种充当 bean 实例来源的方法。如果指定的上下文中没有实例，方法声明本身会描述 bean，容器调用获取 bean 实例的方法。制作者方法使应用可以完全控制 bean 实例化流程。

本节介绍如何使用制作者方法生成各种不是注入 Bean 的不同对象。

示例：使用 Producer 方法

通过使用制作者方法而非替代方案，部署后允许多形性。

示例中的 @Preferred 注释是限定符注释。有关 限定符 的更多信息，请参阅质量。

```
@SessionScoped
public class Preferences implements Serializable {
  private PaymentStrategyType paymentStrategy;
  ...
  @Produces @Preferred
  public PaymentStrategy getPaymentStrategy() {
     switch (paymentStrategy) {
        case CREDIT_CARD: return new CreditCardPaymentStrategy();
        case CHECK: return new CheckPaymentStrategy();
        default: return null;
     }
  }
}
```

以下注入点具有与 producer 方法相同的类型和限定注释，因此它会使用常见的上下文和依赖注入规则解析为制作者方法。容器调用制作者方法来获取实例来为这个注入点提供服务。

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

示例：将范围分配给一个 Producer 方法

制作者方法的默认范围为 @Dependent。如果您将范围分配给 Bean，它将绑定到适当的上下文。本例中的 producer 方法每个会话仅调用一次。

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
  ...
}
```

示例：在 Producer 方法中使用 Injection

应用程序直接实例化的对象无法利用依赖项注入，也没有拦截器。但是，您可以使用依赖项注入制作者方法来获取 bean 实例。

```
@Produces @Preferred @SessionScoped
```

```
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          CheckPaymentStrategy cps ) {
  switch (paymentStrategy) {
    case CREDIT_CARD: return ccps;
    case CHEQUE: return cps;
    default: return null;
  }
}
```

如果您将请求作用域 Bean 注入会话范围制作者中，则制作者方法会将当前的请求作用范围实例提升到会话范围中。这几乎肯定不是所需的行为，因此，以这种方式使用制作者方法时要谨慎。

注意

producer 方法的范围不继承自声明制作者方法的 bean。

借助制作者方法，您可以注入非 Bean 对象并动态更改代码。

7.8. 替代 BEAN

替代方案是其实施特定于特定客户端模块或部署方案的 Bean。

默认情况下，@ Alternative be an 被禁用。通过编辑 Bean .xml 文件，为特定的 Bean 存档启用它们。但是，此激活仅适用于该存档中的 bean。您可以使用 @Priority 注释为整个应用启用替代选项。

示例：定义替代方案

这种替代方案定义了 PaymentProcessor 类的实施，该类同时使用 @Synchronous 和 @Asynchronous 替代方案：

```
@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

  public void process(Payment payment) { ... }

}
```

示例：使用 beans.xml启用 @Alternative

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <alternatives>
      <class>org.mycompany.mock.MockPaymentProcessor</class>
  </alternatives>
</beans>
```

声明所选替代方案

@Priority 注释允许为整个应用启用替代选项。可以为应用程序指定优先级：

- **将 @Priority 注释放在受管 Bean 或会话 Bean 的 Bean 类上，或者**
- **通过将 @Priority 注释放置到 bean 类，以声明制作者方法、字段或资源。**

7.8.1. 使用替代方案覆盖 Injection

您可以使用备用 Bean 来覆盖现有的 Bean。它们可以想象成是插入同一角色的类的方法，但功能不同。默认情况下禁用它们。

此任务显示如何指定和启用替代方案。

覆盖 Injection

此任务假定项目中已有一个 TranslatingWelcome 类，但您想要使用"mock" TranslatingWelcome 类覆盖它。测试部署就是如此，因此无法使用 真实的转换器 Bean。

1. **定义备选方案.**

```
@Alternative
@Translating
public class MockTranslatingWelcome extends Welcome {
  public String buildPhrase(string city) {
    return "Bienvenue Ã  " + city + "!");
  }
}
```

2. 通过将完全限定的类名称添加到您的 META-INF/beans.xml 或 WEB-INF/beans.xml 文件来激活替换实施。

```
<beans>
 <alternatives>
  <class>com.acme.MockTranslatingWelcome</class>
 </alternatives>
</beans>
```

现在，使用替代的实现而不是原始实施。

7.9. 挪威

在许多系统中，使用架构模式会产生一组重复的 Bean 角色。stereotype 允许您识别此类角色，并声明具有该角色的 Bean 在中央位置的一些常见元数据。

stereotype 封装了以下任意组合：

- 默认范围。
- 组拦截器绑定。

stereotype 也可以指定：

- 所有默认 bean EL 名称的 Bean。
- 所有的 Bean 都是替代的 Bean。

Bean 可以声明零、一或多任。stereotype 是打包了几个其他注释的 @Stereotype 注释。stereotype 注解可应用到 bean 类、制作者方法或字段。

从 stereotype 中继承范围的类可以覆盖该 stereotype，并直接在 bean 上指定范围。

此外，如果 stereotype 具有 @Named 注释，则它所放入的任何 Bean 都具有默认的 Bean 名称。如果 Bean 上直接指定了 @Named 注释，则 bean 可以覆盖此名称。有关指定 Bean 的更多信息，请参阅命名 Bean。

7.9.1. 使用 Stereotypes

无需考虑，注释就会变得模糊不清。此任务向您展示如何使用强制减少杂乱情况并简化代码。

示例：注解过滤器

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

定义和使用 Stereotypes

1. 定义 stereotype。

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
 ...
}
```

2. 使用 stereotype。

```
@BusinessComponent
public class AccountManager {
  public boolean transfer(Account a, Account b) {
```

```
    ...
  }
}
```

7.10. 观察方法

发生事件时，观察方法会收到通知。

上下文和依赖注入还提供事务观察方法，它在完成之前或交易完成阶段接收事件通知。

7.10.1. 触发和观察事件

示例：Fire a Event

以下代码显示了在方法中注入和使用的事件：

```
public class AccountManager {
  @Inject Event<Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

示例：Fire a Event with a Qualifier

您可以使用限定符为事件注入添加注解，使其更具体。有关 限定符 的更多信息，请参阅质量。

```
public class AccountManager {
  @Inject @Suspicious Event <Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

示例：观察事件

若要观察事件，可使用 @Observes 注释。

```
public class AccountObserver {
  void checkTran(@Observes Withdrawal w) {
```

```
    ...
  }
}
```

您可以使用限定符来仅观察特定类型的事件。

```
public class AccountObserver {
  void checkTran(@Observes @Suspicious Withdrawal w) {
    ...
  }
}
```

7.10.2. 事务观察器

在交易开始阶段之前或之后，交易处理接收事件通知。事务处理在有状态对象模型中很重要，因为状态通常保留比单个原子交易更长的时间。

交易有五种类型：

- IN_PROGRESS ：默认情况下，会立即调用。
- AFTER_SUCCESS ：观察器在事务完成阶段后会被调用，但前提是事务成功完成。
- AFTER_FAILURE ：观察者在事务完成阶段后会被调用，但前提是事务无法成功完成。
- AFTER_COMPLETION ：观察器在事务完成后被调用。
- BEFORE_COMPLETION ：在事务完成阶段前调用观察者。

以下观察方法刷新应用程序上下文中缓存的查询结果集，但只有在更新 Category 树的操作成功时才会刷新：

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS)
CategoryUpdateEvent event) { ... }
```

假设您缓存了 Jakarta Persistence 查询结果，请在应用程序范围中设置，如下例所示：

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@ApplicationScoped @Singleton

public class Catalog {
  @PersistenceContext EntityManager em;
  List<Product> products;
  @Produces @Catalog
  List<Product> getCatalog() {
    if (products==null) {
      products = em.createQuery("select p from Product p where p.deleted = false")
        .getResultList();
    }
    return products;
  }
}
```

有时 会创建 或删除产品。发生这种情况时，您需要刷新 产品 目录。但是，您必须等待事务成功完成，然后才能进行此刷新。

以下是创建和删除 Products 触发事件的 bean 示例：

```
import javax.enterprise.event.Event;

@Stateless

public class ProductManager {
  @PersistenceContext EntityManager em;
  @Inject @Any Event<Product> productEvent;
  public void delete(Product product) {
    em.delete(product);
    productEvent.select(new AnnotationLiteral<Deleted>(){}).fire(product);
  }

  public void persist(Product product) {
    em.persist(product);
    productEvent.select(new AnnotationLiteral<Created>(){}).fire(product);
  }
  ...
}
```

Catalog 现在可在成功完成事务后观察事件：

```
import javax.ejb.Singleton;
```

```
@ApplicationScoped @Singleton
public class Catalog {
  ...
  void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
    products.add(product);
  }

  void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
    products.remove(product);
  }

}
```

7.11. 拦截器

拦截器允许您在 Bean 的业务方法中添加功能，而无需直接修改 bean 的方法。拦截器在 Bean 的任何业务方法之前执行。拦截器定义为 Jakarta Enterprise Beans 规范的一部分。

Jakarta Contexts 和 Dependency Injection 允许您使用注解将拦截器绑定到 bean 来增强此功能。

拦截点

- 业务方法拦截器：商业方法拦截器适用于 Bean 的客户调用 Bean 方法。
- 生命周期回调拦截器：生命周期回调拦截器适用于容器调用的生命周期回调。
- 超时方法拦截器：超时方法拦截器应用于容器的 Jakarta Enterprise Beans 超时方法的调用。

启用 Interceptors

默认情况下，禁用所有拦截器。您可以使用 bean 存档的 beans.xml 描述符启用拦截器。但是，此激活仅适用于该存档中的 bean。

您可以使用 @Priority 注释为整个应用启用拦截器。

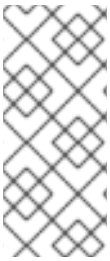
示例：在 beans.xml中启用 Interceptors

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2.0.xsd">
  <interceptors>
    <class>org.mycompany.myapp.TransactionInterceptor</class>
  </interceptors>
</beans>
```

使 XML 声明解决了两个问题：

- 它可让您在系统中指定拦截器的排序，以确保确定行为。
- 它可让您在部署时启用或禁用拦截器类。

在使用 beans.xml 文件启用拦截器之前，使用 @Priority 启用的拦截器会被调用。



注意

由 @Priority 启用拦截器并且同时被 beans.xml 文件调用会导致无法移植的行为。因此，应避免这种启用组合，以便在不同 Jakarta 上下文和依赖注入实施之间保持一致的行为。

7.11.1. 使用 Jakarta Contexts 和 Dependency Injection 的 Interceptors

Jakarta Contexts 和 Dependency Injection 可以简化拦截器代码，更轻松地应用于您的业务代码。

如果没有 Jakarta 上下文和依赖注入，拦截器有两个问题：

- Bean 必须直接指定拦截器实施。
- 应用程序中的每个 bean 都必须以正确的顺序指定完整的拦截器集合。这使得在应用程序范

围内添加或移除拦截器会非常耗时且容易出错。

使用 Jakarta 上下文和依赖注入的 Interceptors

1. 定义拦截器绑定类型。

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Secure {}
```

2. 标记拦截器实施。

```
@Secure
@Interceptor
public class SecurityInterceptor {
  @AroundInvoke
  public Object aroundInvoke(InvocationContext ctx) throws Exception {
    // enforce security ...
    return ctx.proceed();
    }
}
```

3. 在您的开发环境中使用拦截器。

```
@Secure
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

4. 通过将其添加到 META-INF/beans.xml 或 WEB-INF/beans.xml 文件，在部署中启用拦截器。

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

拦截器按照列出的顺序应用。

7.12. DECORATORS

解码器截获来自特定 Java 接口的调用，并了解与该接口连接的所有语义。解码器可用于为某些业务问题建模，但并不具有拦截器的一般性。decorator 是一种 Bean 甚至抽象类，用于实施它解密的类型，并标上 @Decorator 标注。要在 Jakarta 上下文和依赖注入应用中调用 decorator，必须在 beans.xml 文件中指定。

示例：通过 beans.xml清理一个 Decorator

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <decorators>
      <class>org.mycompany.myapp.LargeTransactionDecorator</class>
  </decorators>
</beans>
```

该声明主要有两个目的：

- 它可让您在系统中为解码器指定排序，以确保确定性行为。
- 它可让您在部署时启用或禁用 decorator 类。

解码器必须具有一个 @Delegate 注入点，才能获得对解码对象的引用。

示例：解码器类

```
@Decorator
public abstract class LargeTransactionDecorator implements Account {

  @Inject @Delegate @Any Account account;
  @PersistenceContext EntityManager em;
```

```
    public void withdraw(BigDecimal amount) {
      ...
    }

    public void deposit(BigDecimal amount);
      ...
    }
}
```

您可以使用 @Priority 注释为整个应用启用 decorator。

在使用 beans.xml 文件启用解码器之前，调用使用 @Priority 启用的 decorators。较低优先级的值先调用。

注意

通过 @Priority 启用 decorator 并且同时被 beans.xml 调用，这会造成不可移植的行为。因此，应避免这种启用组合，以便在不同的上下文和依赖注入实施之间保持一致的行为。

7.13. 便携式扩展

上下文和依赖注入旨在成为框架、扩展以及与其他技术集成的基础。因此，上下文和依赖注入会公开一组 SPI，供开发人员使用可移植扩展到上下文和依赖注入。

扩展可以提供以下类型的功能：

- 与业务流程管理引擎集成.
- 与第三方框架（如 Spring、Seam、GWT 或 Wicket）集成.
- 基于上下文和依赖注入编程模型的新技术.

根据 Jakarta Contexts 和 Dependency Injection 规格，可移植扩展可以通过以下方式与容器集成：

- 提供自己的 Bean、拦截器和解码器到容器。
- 使用依赖项将依赖项注入到自己的对象中。注入服务。
- 为自定义范围提供上下文实施。
- 使用来自另一个源的元数据添加或覆盖基于注解的元数据。

如需更多信息，请参阅 Weld 文档中的可移植扩展。

7.14. BEAN 代理

注入 Bean 的客户端通常不包含对 bean 实例的直接引用。除非 bean 是一个依赖对象，范围为 @Dependent，否则容器必须使用代理对象重定向所有注入的 Bean 引用。

bean 代理（可称为客户端代理）负责确保接收方法调用的 bean 实例是与当前上下文关联的实例。客户端代理还允许 Bean 绑定到上下文（如会话上下文），从而无需递归地将其他注入的 Bean 序列化到磁盘。

由于 Java 的限制，容器无法代理一些 Java 类型。如果使用其中一个类型声明的注入点解析为具有 @Dependent 范围以外的 Bean，容器将中止部署。

某些 Java 类型不能由容器代理。包括：

- 无参数的非专用构造器的类.
- 被声明为 最终 或具有 最终 方法的类.

- 数组和原语类型.

7.15. 在注入中使用代理

当 bean 的生命周期相互不同时，代理用于注入。代理是运行时创建的 Bean 的子类，并覆盖 bean 类的所有非私有方法。代理将调用转发到实际 bean 实例。

在本例中，Pay mentProcessor 实例没有直接注入到 Shop。相反，代理会被注入，调用 processPayment（） 方法时，代理会查找当前的 PaymentProcessor be an 实例，并对其调用 processPayment（） 方法。

示例：Proxy Injection

```
@ConversationScoped
class PaymentProcessor
{
  public void processPayment(int amount)
  {
    System.out.println("I'm taking $" + amount);
  }
}

@ApplicationScoped
public class Shop
{

  @Inject
  PaymentProcessor paymentProcessor;

  public void buyStuff()
  {
    paymentProcessor.processPayment(100);
  }
}
```

第 8 章 JBOSS EAP MBEAN 服务

受管 Bean 有时只称为 MBean，是利用依赖注入创建的 JavaBean 类型。MBean 服务是 JBoss EAP 服务器的核心构建模块。

8.1. 编写 JBOSS MBEAN 服务

编写依赖于 JBoss 服务的自定义 MBean 服务需要服务接口方法模式。JBoss MBean 服务接口方法模式由一组生命周期操作组成，在 可创建、启动、停止 和销毁 自身时通知 MBean 服务。

您可以使用以下任一方法管理依赖项状态：

- 如果您想在 MBean 上调用特定方法，请在 MBean 界面中声明这些方法。这种方法允许您的 MBean 实施以避免对 JBoss 特定课程的依赖。
- 如果您不担心对 JBoss 特定类的依赖，则您可以扩展您的 MBean 接口和 ServiceMBean Support 类。ServiceMBeanSupport 类提供创建、启动和停止等服务生命周期方法的实施。要处理 start（） 事件等特定事件，您需要覆盖 ServiceMBeanSupport 类提供的 startService（） 方法。

8.1.1. 标准 MBean 示例

本节开发两个在服务存档(.sar)中打包在一起的 MBean 服务示例。

ConfigServiceMBean 接口声明了具体的方法，如 start、getTimeout 和 stop 方法，以正确 启动、保留 和停止 MBean，而无需使用任何 JBoss 特定类。ConfigService 类实施 ConfigServiceMBean 接口，从而实施该接口内使用的方法。

PlainThread 类扩展 ServiceMBeanSupport 类，并实施 PlainThreadMBean 接口。PlainThread 启动线程并使用 ConfigServiceMBean.getTimeout（） 来确定线程应休眠的时长。

示例：MBean 服务类

```
package org.jboss.example.mbean.support;
public interface ConfigServiceMBean {
   int getTimeout();
```

```
    void start();
    void stop();
}
package org.jboss.example.mbean.support;
public class ConfigService implements ConfigServiceMBean {
    int timeout;
    @Override
    public int getTimeout() {
        return timeout;
    }
    @Override
    public void start() {
        //Create a random number between 3000 and 6000 milliseconds
        timeout = (int)Math.round(Math.random() * 3000) + 3000;
        System.out.println("Random timeout set to " + timeout + " seconds");
    }
    @Override
    public void stop() {
        timeout = 0;
    }
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBean;
public interface PlainThreadMBean extends ServiceMBean {
    void setConfigService(ConfigServiceMBean configServiceMBean);
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBeanSupport;
public class PlainThread extends ServiceMBeanSupport implements PlainThreadMBean {
    private ConfigServiceMBean configService;
    private Thread thread;
    private volatile boolean done;
    @Override
    public void setConfigService(ConfigServiceMBean configService) {
        this.configService = configService;
    }
    @Override
    protected void startService() throws Exception {
        System.out.println("Starting Plain Thread MBean");
        done = false;
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (!done) {
                        System.out.println("Sleeping....");
                        Thread.sleep(configService.getTimeout());
                        System.out.println("Slept!");
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
```

```
        thread.start();
    }
    @Override
    protected void stopService() throws Exception {
        System.out.println("Stopping Plain Thread MBean");
        done = true;
    }
}
```

jboss-service.xml 描述符演示了 ConfigService 类如何使用注入标签 注入 PlainThread 类。inject tag 在 PlainThreadMBean 和 ConfigServiceMBean 之间建立一个依赖关系，因此 PlainThreadMBean 可以轻松地使用 ConfigServiceMBean。

示例： jboss-service.xml 服务描述符

```
<server xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:jboss:service:7.0 jboss-service_7_0.xsd"
      xmlns="urn:jboss:service:7.0">
 <mbean code="org.jboss.example.mbean.support.ConfigService"
name="jboss.support:name=ConfigBean"/>
 <mbean code="org.jboss.example.mbean.support.PlainThread"
name="jboss.support:name=ThreadBean">
  <attribute name="configService">
   <inject bean="jboss.support:name=ConfigBean"/>
  </attribute>
 </mbean>
</server>
```

编写 MBeans 示例后，您可以将类和 jboss-service.xml 描述符打包在服务存档(.sar)的 META-INF/ 文件夹中。

8.2. 部署 JBOSS MBEAN 服务

示例：在受管域中部署和测试 MBean

使用以下命令，在受管域中部署示例 MBeans(ServiceMBeanTest.sar)：

```
deploy ~/Desktop/ServiceMBeanTest.sar --all-server-groups
```

示例：在单机服务器上部署和测试 MBean

使用以下命令，在单机服务器中构建和部署示例 **MBeans(ServiceMBeanTest.sar)**：

```
deploy ~/Desktop/ServiceMBeanTest.sar
```

示例：取消部署 MBeans 归档

使用以下命令取消部署 **MBeans** 示例：

```
undeploy ServiceMBeanTest.sar
```

第 9 章 JAKARTA CONCURRENCY

Jakarta Concurrency 是一种 API，可满足 Jakarta EE 应用环境规格中的 Java SE 并发实用程序。它在 Jakarta Concurrency 规范中定义。JBoss EAP 允许您创建、编辑和删除 Jakarta Concurrency 实例，从而使这些实例随时可供应用使用。

Jakarta Concurrency 通过拉取现有上下文的应用线程并在自己的线程中使用这些线程，帮助扩展调用上下文。默认情况下，这种调用上下文的扩展包括类加载、JNDI 和安全上下文。

Jakarta Concurrency 的类型包括：

- 上下文服务
- 管理的线程事实
- 受管执行器服务
- 管理的调度执行器服务

示例： standalone.xml中的 Jakarta Concurrency

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
        <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
        <concurrent>
            <context-services>
                <context-service name="default" jndi-name="java:jboss/ee/concurrency/context/default"
use-transaction-setup-provider="true"/>
            </context-services>
            <managed-thread-factories>
                <managed-thread-factory name="default" jndi-
name="java:jboss/ee/concurrency/factory/default" context-service="default"/>
            </managed-thread-factories>
            <managed-executor-services>
                <managed-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/executor/default" context-service="default" hung-task-
threshold="60000" keepalive-time="5000"/>
            </managed-executor-services>
            <managed-scheduled-executor-services>
                <managed-scheduled-executor-service name="default" jndi-
```

```
name="java:jboss/ee/concurrency/scheduler/default" context-service="default" hung-task-
threshold="60000" keepalive-time="3000"/>
                </managed-scheduled-executor-services>
            </concurrent>
            <default-bindings context-service="java:jboss/ee/concurrency/context/default"
datasource="java:jboss/datasources/ExampleDS" managed-executor-
service="java:jboss/ee/concurrency/executor/default" managed-scheduled-executor-
service="java:jboss/ee/concurrency/scheduler/default" managed-thread-
factory="java:jboss/ee/concurrency/factory/default"/>
</subsystem>
```

9.1. 上下文服务

上下文服务(javax.enterprise.concurrent.ContextService)允许您从现有对象构建上下文代理。在将调用传输到原始对象之前，上下文代理会准备调用上下文，供其他 Jakarta Concurrency 实用程序在创建或调用上下文时使用。

上下文服务并发程序的属性包括：

- Name ：所有上下文服务中的唯一名称。
- jnDI-name ：定义应将上下文服务放置在 JNDI 中。
- use-transaction-setup-provider: Optional.指明在调用代理对象时，上下文服务构建的上下文代理是否应该在上下文中暂停事务。其值默认为 false，但默认上下文服务的值为 true。

有关使用上下文服务并发实用程序的示例，请参见上面的示例。

示例：添加新上下文服务

```
/subsystem=ee/context-service=newContextService:add(jndi-
name=java:jboss/ee/concurrency/contextservice/newContextService)
```

示例：更改上下文服务

```
/subsystem=ee/context-service=newContextService:write-attribute(name=jndi-name,
value=java:jboss/ee/concurrency/contextservice/changedContextService)
```

此操作需要重新加载。

示例：删除上下文服务

```
/subsystem=ee/context-service=newContextService:remove()
```

此操作需要重新加载。

9.2. 管理的线程事实

托管线程工厂 (javax.enterprise.concurrent.ManagedThreadFactory) 并发实用工具允许 Jakarta EE 应用创建 Java 线程。JBoss EAP 处理托管的线程实例，因此 Jakarta EE 应用程序无法调用任何与生命周期相关的方法。

管理的线程工厂并发实用程序的属性包括：

- context-service ：所有受管线程工厂中的唯一名称。
- JNDI -name ：定义应在 JNDI 中放置托管线程工厂的位置。
- priority ：可选。表示工厂创建的新线程的优先级，默认为 5。

示例：添加新管理的线程事实

```
/subsystem=ee/managed-thread-factory=newManagedTF:add(context-
service=newContextService, jndi-
name=java:jboss/ee/concurrency/threadfactory/newManagedTF, priority=2)
```

示例：更改受管线程事实

```
/subsystem=ee/managed-thread-factory=newManagedTF:write-attribute(name=jndi-name,
value=java:jboss/ee/concurrency/threadfactory/changedManagedTF)
```

此操作需要重新加载。类似地，您也可以更改其他属性。

示例：删除受管线程事实

```
/subsystem=ee/managed-thread-factory=newManagedTF:remove()
```

此操作需要重新加载。

9.3. 受管执行器服务

受管执行器服务 (javax.enterprise.concurrent.ManagedExecutorService) 允许 Jakarta EE 应用提交异步执行任务。JBoss EAP 处理托管执行器服务实例，因此 Jakarta EE 应用程序无法调用任何与生命周期相关的方法。

受管 executor 服务并发程序的属性包括：

-

context-service ：可选.按名称引用现有上下文服务。如果指定，引用的上下文服务会在向 executor 提交任务时捕获调用上下文，然后在执行任务时使用该调用上下文。

- JNDI -name ：定义受管线程工厂应放置在 JNDI 中。
- max-threads ：定义执行者使用的最大线程数。如果未定义，则使用来自 core-threads 的值。
- thread-factory ：按名称引用现有托管线程工厂，以处理内部线程的创建。如果没有指定，则内部将创建和使用具有默认配置的管理线程工厂。
- core-threads ：定义执行者要使用的最少线程数。如果未定义此属性，则根据处理器数量计算默认值。不建议值为 0。有关如何使用此值来确定排队策略的详细信息，请参阅 queue-length 属性。
- keepalive-time ：定义内部线程可以空闲的时间，以毫秒为单位。属性默认值为 60000。
- queue-length ：指定 executor 的任务队列容量。值 0 表示直接移交和可能的拒绝。如果此属性未定义或设置为 Integer.MAX_VALUE，这表示应当使用一个未绑定的队列。所有其他值指定准确的队列大小。如果使用未绑定队列或直接移交，则需要大于 0 的内核 线程 值。
- hung-task-threshold ：定义时间（以毫秒为单位），之后任务被托管执行器服务视为挂起并且强制中止。如果值为 0（ 默认值），则任务永远不会被视为挂起。
- 长时间运行任务 ：建议优化长时间运行任务的执行，默认值为 false。
- reject-policy ：定义要在 executor 拒绝任务时使用的策略。属性值可以是默认的 ABORT，这意味着应抛出异常，或 RETRY_ABORT，这意味着执行者会在抛出异常之前尝试再次提交一次异常。

示例：添加新的托管执行器服务

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:add(jndi-
name=java:jboss/ee/concurrency/executor/newManagedExecutorService, core-threads=7,
thread-factory=default)
```

示例：更改受管执行器服务

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:write-
attribute(name=core-threads,value=10)
```

此操作需要重新加载。同样，您也可以更改其他属性。

示例：删除受管执行器服务

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:remove()
```

此操作需要重新加载。

9.4. 管理的调度执行器服务

受管计划执行器服务 (javax.enterprise.concurrent.ManagedScheduledExecutorService) 允许 Jakarta EE 应用为异步执行调度任务。JBoss EAP 处理托管的调度执行器服务实例，因此 Jakarta EE 应用程序无法调用任何与生命周期相关的方法。

受管 executor 服务并发程序的属性包括：

- context-service ：按名称引用现有上下文服务。如果指定，引用的上下文服务会在向 executor 提交任务时捕获调用上下文，然后在执行任务时使用该调用上下文。
- hung-task-threshold ：定义时间（以毫秒为单位），之后任务被托管的执行执行者服务视为挂起并且强制中止。如果值为 0（默认值），则任务永远不会被视为挂起。

- keepalive-time ：定义内部线程可以空闲的时间，以毫秒为单位。属性默认值为 60000。
- reject-policy ：定义要在 executor 拒绝任务时使用的策略。属性值可能是默认的 ABORT，这意味着应抛出异常，或 RETRY_ABORT，这意味着执行者将尝试再次提交一次，然后再抛出异常。
- core-threads ：定义计划执行器使用的最少线程数。
- JNDI -name ：定义受管调度执行器服务应放在 JNDI 中的位置。
- 长时间运行任务 ：建议优化长时间运行任务的执行，默认值为 false。
- thread-factory ：按名称引用现有托管线程工厂，以处理内部线程的创建。如果没有指定，则内部将创建和使用具有默认配置的管理线程工厂。

示例：添加新管理的已调度执行器服务

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:add(jndi-
name=java:jboss/ee/concurrency/scheduledexecutor/newManagedScheduledExecutorService,
core-threads=7, context-service=default)
```

此操作需要重新加载。

示例：更改受管的已调度执行器服务

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:write-attribute(name=core-threads,
value=10)
```

此操作需要重新加载。类似地，您可以更改其他属性。

示例：删除受管的已调度执行器服务

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:remove()
```

此操作需要重新加载。

9.5. 受管执行器服务和受管调度执行器服务的运行时统计信息

您可以通过查看管理 CLI 属性生成的运行时统计信息，监控受管 executor 服务的性能和管理调度的执行者服务。您可以查看单机服务器的运行时统计信息，或者查看映射到主机的个别服务器的运行时统计信息。

重要

domain.xml 配置不包括运行时统计管理 CLI 属性的资源，因此您无法使用管理 CLI 属性来查看受管域的运行时统计信息。

表 9.1. 显示用于监控受管执行器服务和托管调度执行器服务性能的管理 CLI 属性。

属性	描述
active-thread-count	主动执行任务的近似线程数量。
completed-task-count	已完成执行的任务约为总数。
hung-thread-count	挂起的 executor 线程数量。
max-thread-count	执行器线程的最大数量.
current-queue-size	executor 任务队列的当前大小。

属性	描述
task-count	已提交执行的任务的近似总数。
thread-count	执行器线程的当前数量。

查看在单机服务器上运行的受管 executor 服务的运行时统计信息示例：

```
[standalone@localhost:9990 /] /subsystem=ee/managed-executor-service=default:read-
resource(include-runtime=true,recursive=true)
```

在单机服务器上运行的受管调度执行器服务的运行时统计数据示例。

```
[standalone@localhost:9990 /] /subsystem=ee/managed-scheduled-executor-service=default:read-
resource(include-runtime=true,recursive=true)
```

查看在映射到主机的服务器上运行的受管 executor 服务的运行时统计信息示例：

```
[domain@localhost:9990 /] /host=<host_name>/server=<server_name>/subsystem=ee/managed-
executor-service=default:read-resource(include-runtime=true,recursive=true)
```

在映射到主机的服务器上运行的受管计划执行器服务的运行时统计数据示例。

```
[domain@localhost:9990 /] /host=<host_name>/server=<server_name>/subsystem=ee/managed-
scheduled-executor-service=default:read-resource(include-runtime=true,recursive=true)
```

其他资源

- 有关创建受管执行器服务的信息，请参阅 **JBoss EAP** *开发指南* 中的 托管执行器服务。
- 有关创建受管调度执行器服务的信息，请参阅 **JBoss EAP** *开发指南* 中的 托管调度执行器服务。

第 10 章 UNDERTOW

10.1. UNDERTOW 处理程序简介

Undertow 是一种 Web 服务器，设计用于阻塞和非阻塞任务。它取代了 JBoss EAP 7 中的 JBoss Web。它的一些主要功能是：

- high performance
- 可嵌入
- Servlet 4.0
- WebSockets
- 反向代理

请求生命周期

客户端连接到服务器时，Undertow 会创建 a io.undertow.server.HttpServerConnection。当客户端发送请求时，它将由 Undertow 解析器解析，然后生成的 io.undertow.server.HttpServerExchange 传递到 root 处理程序。当 root 处理程序完成后，可能会出现以下四种情况之一：

- 交易完成。

  如果请求和响应通道已完全读取或写入，则交易被视为已完成。对于没有内容的请求，如 GET 和 HEAD，请求方将自动视为完全读取。当处理程序写出完整响应并且关闭并完全清空响应通道时，读取端被视为已完成。如果交换已完成，则不执行任何操作。

- root 处理程序通常返回而不完成交换。

  在这种情况下，交换是通过调用 HttpServerExchange.endExchange（） 来完成的。

- root 处理程序返回一个例外。

在本例中，设置了 500 的响应代码，并使用 HttpServerExchange.endExchange（） 结束交换。

- root 处理程序可以在 HttpServerExchange.dispatch（） 被调用后返回，或者在同步 IO 启动后返回。

  在这种情况下，分配的任务将提交到分配执行器，或者如果在请求或响应通道上启动了异步 IO，则会启动此操作。在这两种情况下，交换都不会完成。处理完后，将由您的 async 任务来完成交换。

目前，最常用的 HttpServerExchange.dispatch（） 是从不允许阻止操作的 IO 线程中执行，变为 worker 线程，这确实允许阻止操作。

示例： Dispatching to a Worker Thread

```
public void handleRequest(final HttpServerExchange exchange) throws Exception {
    if (exchange.isInIoThread()) {
      exchange.dispatch(this);
      return;
    }
    //handler code
}
```

由于在调用堆栈返回后实际不会分配交换，您可以确保一次在交换中不再激活多个线程。交换是不安全的。但是，只要两个线程都不会尝试一次修改它，就可以在多个线程之间传递它。

结束 Exchange

有两种方法可以结束交换：完全读取请求频道并在响应频道中调用 shutdownWrites（），然后清除它，或者调用 HttpServerExchange.endExchange（）。调用 endExchange（） 时，Undertow 将检查内容是否已生成。如果存在，它将只排空请求频道并关闭并清空响应频道。如果没有，且在交换上注册了任何默认响应监听器，则 Undertow 将获得生成默认响应的机会。这种机制是生成默认错误页面的方式。

有关配置 Undertow 的更多信息，请参阅 JBoss EAP *配置指南中的配置* Web 服务器。

10.2. 将现有 UNDERTOW 处理程序用于部署

Undertow 提供了一组默认的处理程序，可用于部署到 JBoss EAP 的任何应用。

要将处理程序与部署搭配使用，您需要添加 WEB-INF/undertow-handlers.conf 文件。

示例： WEB-INF/undertow-handlers.conf 文件

```
allowed-methods(methods='GET')
```

所有处理程序也可以取一个可选 predicate 来在特定情况下应用该处理程序。

示例：使用可选 Predicate 的 WEB-INF/undertow-handlers.conf 文件

```
path('/my-path') -> allowed-methods(methods='GET')
```

上例将仅将 allowed-methods 处理程序应用到路径 /my-path。

Undertow Handler 默认参数

些处理程序具有 default 参数，允许您在处理程序定义中指定该参数的值，而不使用名称。

示例： 使用默认参数的 WEB-INF/undertow-handlers.conf 文件

```
path('/a') -> redirect('/b')
```

您还可以更新 WEB-INF/jboss-web.xml 文件，使其包含一个或多个处理程序的定义，但首选使用 WEB-INF/undertow-handlers.conf。

示例： WEB-INF/jboss-web.xml 文件

```
<jboss-web>
   <http-handler>
      <class-name>io.undertow.server.handlers.AllowedMethodsHandler</class-name>
      <param>
         <param-name>methods</param-name>
         <param-value>GET</param-value>
      </param>
   </http-handler>
</jboss-web>
```

提供的 Undertow 处理程序的完整列表可在 Provided Undertow Handlers 参考中找到。

10.3. 创建自定义处理程序

定义自定义处理程序的方法有两种：

1. 使用 WEB-INF/jboss-web.xml 文件
2. 在 WEB-INF/undertow-handlers.conf 中

使用 WEB-INF/jboss-web.xml 文件定义自定义处理程序

自定义处理程序可以在 WEB-INF/jboss-web.xml 文件中定义。

示例：在 WEB-INF/jboss-web.xml中定义自定义处理程序

```
<jboss-web>
   <http-handler>
      <class-name>org.jboss.example.MyHttpHandler</class-name>
```

```
    </http-handler>
</jboss-web>
```

示例： HttpHandler Class

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void handleRequest(HttpServerExchange exchange) throws Exception {
        // do something
        next.handleRequest(exchange);
    }
}
```

也可以使用 WEB-INF/jboss-web.xml 文件为自定义处理程序设置参数。

示例：在 WEB-INF/jboss-web.xml 中定义参数

```
<jboss-web>
    <http-handler>
        <class-name>org.jboss.example.MyHttpHandler</class-name>
        <param>
            <param-name>myParam</param-name>
            <param-value>foobar</param-value>
        </param>
    </http-handler>
</jboss-web>
```

要使这些参数发挥作用，处理程序类需要具有对应的集合器。

示例：在 Handler 中定义设置方法

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;
    private String myParam;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void setMyParam(String myParam) {
        this.myParam = myParam;
    }

    public void handleRequest(HttpServerExchange exchange) throws Exception {
        // do something, use myParam
        next.handleRequest(exchange);
    }
}
```

在 WEB-INF/undertow-handlers.conf 文件中定义自定义处理程序

除了使用 WEB-INF/jboss-web.xml 定义处理程序外，也可以在 WEB-INF/undertow-handlers.conf 文件中定义。

```
myHttpHandler(myParam='foobar')
```

要让 WEB-INF/undertow-handlers.conf 中定义的处理器正常工作，需要创建两个因素：

1. HandlerBuilder 实施，它为 undertow-handlers.conf 定义对应的语法位，并负责创建 HttpHandler （封装在 HandlerWrapper 中）。

   示例： HandlerBuilder 类

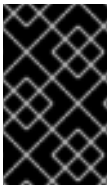
```
package org.jboss.example;

import io.undertow.server.HandlerWrapper;
import io.undertow.server.HttpHandler;
import io.undertow.server.handlers.builder.HandlerBuilder;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

public class MyHandlerBuilder implements HandlerBuilder {
    public String name() {
        return "myHttpHandler";
    }

    public Map<String, Class<?>> parameters() {
        return Collections.<String, Class<?>>singletonMap("myParam", String.class);
    }

    public Set<String> requiredParameters() {
        return Collections.emptySet();

    }

    public String defaultParameter() {
        return null;

    }

    public HandlerWrapper build(final Map<String, Object> config) {
        return new HandlerWrapper() {
            public HttpHandler wrap(HttpHandler handler) {
                MyHttpHandler result = new MyHttpHandler(handler);
                result.setMyParam((String) config.get("myParam"));
                return result;
            }
        };
    }
}
```

2. 文件中的条目。META-INF/services/io.undertow.server.handlers.builder.HandlerBuilder. 此文件必须位于类路径上，例如，在 WEB-INF/classes 中。

```
org.jboss.example.MyHandlerBuilder
```

10.4. 开发自定义 HTTP 机制

当 Elytron 用于保护 Web 应用时，可以实施可以使用 elytron 子系统注册的自定义 HTTP 身份验证机制。然后，也可以覆盖部署中的配置，以利用这种机制，而无需修改部署。



重要

所有自定义 HTTP 机制都需要实施 HttpServerAuthenticationMechanism 接口。

通常，对于 HTTP 机制，会调用 evaluation Request 方法来处理在 HTTPServerRequest 对象中传递的请求。机制处理请求并使用请求中的以下回调方法之一来指示结果：

- authenticationComplete - 机制成功验证请求。
- Authentication Failed - 身份验证尝试失败，但失败。
- Authentication InProgress - 开始身份验证，但还需要额外往返。
- badRequest - 此机制的身份验证失败验证请求。
- noAuthenticationInProgress - 机制没有尝试任何身份验证阶段。

在创建了实施 HttpServerAuthenticationMechanism 接口的自定义 HTTP 机制后，下一步是创建一个工厂来返回此机制的实例。工厂必须实施 HttpAuthenticationFactory 接口。实施工厂中最重要的步骤是重复检查请求的机制的名称。如果工厂无法创建所需的机制，则工厂返回 null 非常重要。机制工厂也可以考虑传输的映射中的属性，以确定它是否能创建请求的机制。

有两种不同的方法可用来公告可供使用的机制工厂。

- 第一种方法是实施 java.security.Provider，其支持的每种机制都注册为一次 HttpAuthenticationFactory。
- 第二种方法是使用 java.util.ServiceLoader 来发现工厂。要做到这一点，META-INF/services 下应当添加一个名为

org.wildfly.security.http.HttpServerAuthenticationMechanismFactory 的文件。此文件中唯一需要的内容是工厂实施的完全限定类名称。

然后，这个机制可以作为可被使用的模块安装到应用程序服务器中：

```
module add --name=org.wildfly.security.examples.custom-http --resources=/path/to/custom-http-mechanism.jar --dependencies=org.wildfly.security.elytron,javax.api
```

其他资源

- 如需更多信息，请参阅 模块和依赖项。

使用自定义 HTTP 机制

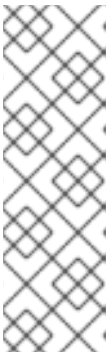
1. 添加自定义模块。

```
/subsystem=elytron/service-loader-http-server-mechanism-factory=custom-factory:add(module=org.wildfly.security.examples.custom-http)
```

2. 添加 http-authentication-factory，将机制工厂绑定到将用于身份验证 的安全域。

```
/subsystem=elytron/http-authentication-factory=custom-mechanism:add(http-server-mechanism-factory=custom-factory,security-domain=ApplicationDomain,mechanism-configurations=[{mechanism-name=custom-mechanism}])
```

3. 更新 application-security-domain 资源，以使用新的 http-authentication-factory。



注意

部署应用时，它默认使用 other 安全域。因此，您需要向应用添加映射，以将其映射到 Elytron HTTP 身份验证工厂。

```
/subsystem=undertow/application-security-domain=other:add(http-authentication-factory=application-http-authentication)
```

现在，可以更新 application-security-domain 资源以使用新的 http-authentication-factory。

```
/subsystem=undertow/application-security-domain=other:write-attribute(name=http-
authentication-factory,value=custom-mechanism)

/subsystem=undertow/application-security-domain=other:write-attribute(name=override-
deployment-config,value=true)
```

请注意，上述命令将覆盖部署配置。这意味着，即使部署已配置为使用不同的机制，也会使用 http-authentication-factory 中的机制。因此，可以在不需要修改部署本身的情况下覆盖部署中的配置，从而利用自定义机制。

4. **重新加载服务器**

```
reload
```

第 11 章 JAKARTA TRANSACTIONS

11.1. 概述

11.1.1. Jakarta 交易概述

简介

本节提供了对 Jakarta 交易的基本了解。

- 关于 Jakarta Transactions
- 事务生命周期
- Jakarta 交易事务示例

11.2. 事务概念

11.2.1. 关于交易

事务由两个或多个操作组成，它们都必须成功或全部失败。成功的结果是提交，失败的结果为回滚。在回滚中，每个成员的状态在尝试提交之前恢复为其状态。

设计良好的事务的典型标准是原子、一致、隔离和可达(ACID)。

11.2.2. 关于交易的 ACID 属性

ACID 是缩写词，代表 原子性、一致性、隔离 和持久性。此术语通常用于数据库或事务操作。

原子性

要使事务成为原子性，所有事务成员都必须做出相同的决定。他们要么全部提交，要么全部回滚。如果原子性中断，结果将被称为启发式的结果。

致性

一致性意味着写入数据库的数据保证为有效的数据，就数据库架构而言。数据库或其他数据源必

须始终处于一致状态。不一致状态的一个示例是，在操作中止前写入一半数据。一致的状态是写入所有数据，或者写入操作在无法完成时回滚。

隔离

隔离意味着交易所执行的数据必须在修改之前锁定，以防止超出事务范围的进程修改数据。

持久性

持久性意味着，在交易成员收到指示提交后的外部故障时，所有成员都可以在解决故障时继续提交事务。此故障可能与硬件、软件、网络或其他任何相关系统相关。

11.2.3. 关于交易协调员或交易管理器

在使用 JBoss EAP 进行事务处理方面，交易量和事务管理器 ™ 大多可以互换。交易审核员这一术语通常用于分布式 JTS 交易。

在 Jakarta Transactions 事务中，TM 在 JBoss EAP 内运行，并在两阶段提交协议期间与交易参与者通信。

TM 告知事务参与者是提交还是回滚其数据，具体取决于其他交易参与者的结果。这样，它会确保事务遵循 ACID 标准。

- 关于交易参与者
- 关于交易的 ACID 属性
- 关于 2-Phase 提交协议

11.2.4. 关于交易参与者

交易参与者是指交易中能够提交或回滚状态的任何资源。它通常是数据库或 Jakarta 消息传递代理，但通过实施交易界面，应用程序代码也可以充当交易参与者。交易的每个参与者均独立决定是否提交或回滚其状态，并且只有在所有参与者都能够提交整个交易成功时才可行。否则，每位参与者都会回滚其状态，整个交易都会失败。TM 协调提交或回滚操作，并确定事务的结果。

11.2.5. 关于 Jakarta Transactions

Jakarta Transactions 是 Jakarta EE Spec 的一部分。它在 Jakarta Transactions 1.3 规范 中定义。

Jakarta Transactions 的实施通过 TM 实施，TM 由 Narayana 项目针对 JBoss EAP 应用服务器进行介绍。TM 允许应用程序通过单一全球交易分配各种资源，如数据库或 Jakarta 消息传递代理。全局事务被称为 XA 事务。通常具有 XA 功能的资源包含在此类交易中，但非 XA 资源也可以成为全球交易的一部分。有几个优化可帮助非 XA 资源作为 XA 功能资源的行为。如需更多信息，请参阅单阶段提交LRCO优化。

在本文档中，术语 Jakarta Transactions 指的是两个方面：

1. Jakarta Transactions，由 Jakarta EE 规范定义。
2. 它指示 TM 如何处理事务。

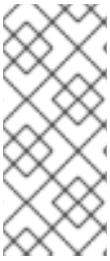
TM 在 Jakarta Transactions 事务模式中工作，数据在内存中共享，事务上下文由远程 Jakarta Enterprise Beans 调用传输。在管理事务模式中，通过发送通用对象请求代理架构(CORBA)消息和 IIOP 调用传输事务上下文来共享数据。两种模式都支持在多个 JBoss EAP 服务器之间进行事务分发。

- 关于分布式交易
- 关于 XA 数据源和 XA 事务

11.2.6. 关于 JTS

JTS 是对象交易服务(OTS)到 Jakarta 的映射。Jakarta EE 应用使用 Jakarta Transactions 来管理事务。然后，当交易经理切换到 JTS 模式时，Jakarta Transactions 与对象事务服务事务实施交互。JTS 工作于 IIOP 协议。使用 JTS 的事务管理器使用称为对象请求代理(ORB)的进程相互通信，使用称为通用对象请求代理架构(CORBA)的通信标准。如需更多信息，请参阅 JBoss EAP 配置 *指南中的 ORB* 配置。

从应用角度使用 Jakarta Transactions 时，JTS 事务的行为方式与 Jakarta Transactions 交易相同。



注意

JBoss EAP 中包含的 JTS 实施支持分布式事务。完全合规 JTS 事务的区别在于与外部第三方 ORB 的互操作性。JBoss EAP 不支持此功能。支持的配置仅在多个 JBoss EAP 容器之间分发事务。

11.2.7. 关于 XML 事务服务

XML 交易服务(XTS)组件支持在业务交易中协调私有和公共 Web 服务。通过使用 XTS，您可以以受控且可靠的方式协调复杂业务交易。XTS API 支持基于 WS-Coordination、WS-Atomic Transaction 和 WS-Business Activity 协议的事务协调模型。

11.2.7.1. XTS 使用的协议概述

WS-协调(WS-C)规范定义了一个框架，它允许插入不同的协调协议以协调客户端、服务和参与者之间的工作。

WS-Transaction(WS-T)协议包括交易协调协议、WS-Atomic Transaction(WS-AT)和 WS-Business Activity(WS-BA)协议，它们利用 WS-C 提供的协调框架。WS-T 是为统一现有的传统交易处理系统而开发的，允许它们相互可靠的通信。

11.2.7.2. Web 服务-原子事务流程

原子事务(AT)旨在支持适合 ACID 语义的短持续时间交互。在 AT 范围内，Web 服务通常采用桥接来访问由 WS-T 控制的 XA 资源，如数据库和消息队列。当交易终止时，参与者将 AT 的结果决定传播到 XA 资源，每个参与者均采取适当的提交或回滚行动。

11.2.7.2.1. Atomic 事务过程

1. 要启动 AT，客户端应用程序首先找到支持 WS-T 的 WS-C 激活器 Web 服务。
2. 客户端向服务发送 WS-C CreateCoordinationContext 消息，并将 http://schemas.xmlsoap.org/ws/2004/10/wsat 指定为其协调类型。
3. 客户端从激活服务接收适当的 WS-T 上下文。
4. 对 CreateCoordinationContext 消息的响应（事务上下文 ）将其 reconcileType 元素设置为 WS-AT 命名空间 http://schemas.xmlsoap.org/ws/2004/10/wsat。它还提到了原子交易协调

器端点，即 WS-C 注册服务，以供参与者参加。

5. 客户端通常继续调用 Web 服务并完成事务，要么提交 Web 服务执行的所有更改，要么回滚。为了能够驱动这一完成，客户端必须通过向注册服务发送寄存器消息（在协调环境中返回端点），将自身注册为完成协议的参与者。

6. 且注册完成，客户端应用程序便会与 Web 服务交互，以完成其业务级别工作。每次调用业务 Web 服务时，客户端都会将事务上下文插入到 SOAP 标头块中，使得每次调用都会隐式地限制在交易范围内。支持 WS-AT 感知 Web 服务的工具包提供工具来连接 SOAP 标头块中的上下文与后端操作。这可确保通过 Web 服务进行的修改在与客户端相同的交易范围内进行，并接受交易协调员提交或回滚。

7. 完成所有必要的应用程序工作后，客户端即可终止事务，以永久更改服务状态。完成参与者指示协调员尝试提交或回滚交易。提交或回滚操作完成后，会将状态返回给参与者以指明交易的结果。

详情请查看 Naryana 项目文档中 的 WS-Co ordination。

11.2.7.2.2. WS-AT 与 Microsoft .NET 客户端的互操作性

X ts 子系统可能会遇到与 Microsoft .NET 客户端通信的问题，因为 WS-AT 规范实施的 .NET 实施存在差异。WS-AT 规范的 .NET 实施强制任何调用异步。

若要实现与 .NET 客户端的互操作性，JBoss EAP xts 子系统中提供了异步注册选项。XTS 异步注册默认为禁用，您应仅在需要时启用它。

要启用与 .NET 客户端的 WS-AT 互操作性，请使用以下管理 CLI 命令：

```
/subsystem=xts:write-attribute(name=async-registration, value=true)
```

11.2.7.3. Web 服务-业务活动过程

Web 服务 - 业务活动(WS-BA)定义 Web 服务应用程序协议，使现有的业务处理和工作流系统能够包装其专有机制，跨实施和业务界限进行互操作。

WS-AT 协议模型不同，其中参与者仅在询问时告知其状态，WS-BA 中的子活动可以直接向协调者指定结果，而不必等待请求。参与者可以选择退出活动，或在任何时间点通知协调人员失败。此功能在任务

失败时很有用，因为通知可用于修改目标并向前驱动处理，而不必等待事务结束时确定故障。

11.2.7.3.1. WS-BA 流程

1. 请求服务来开展工作。

2. 只要这些服务能够撤销任何工作，它们都会通知 WS-BA，以防 WS-BA 之后决定取消工作。如果 WS-BA 出现故障。它可以指示服务执行其 撤销 行为。

WS-BA 协议采用基于补偿的交易模式。当业务活动的参与者完成其工作时，可以选择退出活动。这种选择不允许任何后续回滚。或者，参与者也可以完成自己的活动，向协调者发出信号，如果稍后有一名参与者通知了失败，可以弥补他们已经完成的工作。在后者的情况下，协调会要求每个未退出的参与者弥补失败，让他们有机会执行他们认为适当的任何补偿措施。如果所有参与者在没有失败的情况下退出或完成，协调会通知每位已完成的参与者活动已经结束。

详情请查看 Naryana 项目文档中 的 WS-Co ordination。

11.2.7.4. 事务桥接概述

事务 Bridging 描述了链接 Jakarta EE 和 WS-T 域的过程。事务桥组件 txbridge 提供双向支持，使得任何类型的事务都可以包含设计用于其他类型的业务逻辑。网桥所使用的技术是交集和协议映射的组合。

在交易桥中，交集的协调员注册到现有交易中并执行协议映射的其他任务；即，它似乎是其本土交易类型的资源，而似乎其子方是其本机交易类型的协调者，即使这些交易类型有所不同。

事务网桥驻留在 org.jboss.jbossts.txbridge 及其子软件包中。它由两组不同的类组成，一个用于在各个方向上桥接。

如需了解更多详细信息，请参阅 Naryana 项目文档中的 TXBridge 指南。

11.2.8. 关于 XA 资源和 XA 事务

XA 代表 eXtended 架构，由 X/Open Group 开发以定义使用多个后端数据存储的交易。XA 标准描述了全局 TM 和本地资源管理器之间的接口。XA 允许多个资源（如应用服务器、数据库、缓存和消息队列）参与同一事务，同时保留所有四个 ACID 属性。四个 ACID 属性中有一个是原子性，这意味着，如果

其中一个参与者未能提交更改，其他参与者中止交易，并将其状态恢复为与交易发生前相同的状态。XA 资源是可参与 XA 全球交易的资源。

XA 事务是可跨越多个资源的事务。它涉及一个协调 TM，与一个或多个数据库或其他事务资源一起涉及单一全球 XA 事务。

11.2.9. 关于 XA 恢复

TM 实施 X/Open XA 规范并支持跨多个 XA 资源进行 XA 事务。

XA 恢复是确保更新或回滚受交易进程影响的所有资源的过程，即使任何资源属于交易参与者崩溃或不可用。在 JBoss EAP 范围内，事务 子系统为使用它们的任何 XA 资源或子系统（如 XA 数据源、Jakarta 消息传递消息队列和 Jakarta Connectors 资源适配器）提供了 XA 恢复机制。

XA 恢复发生，无需用户干预.如果 XA 恢复失败，日志输出中会记录错误。如果需要帮助，请联系红帽全球支持服务。XA 恢复过程由定期恢复线程驱动，该线程默认每两分钟启动一次。定期恢复线程处理所有未完成事务。

注意

完成一个 in-doubt 事务的恢复可能需要 4 到 8 分钟，因为它可能需要多次运行恢复过程。

11.2.10. XA 恢复过程的限制

XA 恢复有以下限制：

- 可能无法从成功提交的事务中清除事务日志。

  如果 JBoss EAP 服务器在 XAResource 提交方法后崩溃并提交事务，但在协调员更新日志之前，您可能会在重启服务器时在日志中看到以下警告消息：

  ```
  ARJUNA016037: Could not find new XAResource to use for recovering non-serializable
  XAResource XAResourceRecord
  ```

  这是因为在恢复时，JBoss 事务管理器(TM)会在日志中看到事务参与者并尝试重试提交。最终，JBoss TM 假设资源已提交，不再重试提交。在这种情况下，您可以在提交事务时安全地忽

略此警告，而且不会丢失数据。

要防止警告，请将 com.arjuna.ats.jta.xaAssumeRecoveryComplete 属性 值设置为 true。当新的 XAResource 实例无法来自任何注册的 XAResource Recovery 实例时，就会检查此属性。当设置为 true 时，恢复假设上一提交尝试成功，并且可以在不进一步恢复尝试的情况下从日志中移除实例。此属性必须谨慎使用，因为它是全局的，使用不正确时可能会导致 XAResource 实例处于未提交状态。

注意

JBoss EAP 7.4 实施了增强功能，以便在成功提交事务后清除事务日志，以上情况不应频繁发生。

- 当服务器在 XAResource.prepare（） 结束时崩溃时，不会在 JTS 事务中调用回滚。

如果 JBoss EAP 服务器在完成 XAResource.prepare（） 方法调用后崩溃，则所有参与的 XAResource 实例都会处于就绪状态，并在服务器重启后保持这种方式。事务不会被回滚，资源在事务超时或数据库管理员手动回滚资源并清除事务日志前保持锁定。如需更多信息，请参阅 https://issues.jboss.org/browse/JBTM-2124

- 对已提交的事务可进行定期恢复。

当服务器负载过量时，服务器日志可能包含以下警告信息，后跟一个 stacktrace：

```
ARJUNA016027: Local XARecoveryModule.xaRecovery got XA exception
XAException.XAER_NOTA: javax.transaction.xa.XAException
```

在负载过重时，交易所花费的处理时间可能与定期恢复过程的活动时间重叠。定期恢复过程检测到仍在进行中的事务，并尝试启动回滚，但实际交易将继续完成。在定期恢复尝试但回滚失败时，它会在服务器日志中记录回滚失败。以后的发行版本中会解决此问题的根本原因，但目前可以使用一个临时解决方案。

通过将 com.arjuna.ats.jta.orphanSafetyInterval 属性设置为大于默认值 10000 毫秒，以增大恢复过程的两个阶段之间的间隔。建议值为 40000 毫秒。请注意，这不会解决这个问题。相反，它会降低发生该信号的可能性，并在日志中显示警告消息。如需更多信息，请参阅 https://developer.jboss.org/thread/266729

11.2.11. 关于 2-Phase 提交协议

两阶段提交(2PC)协议引用一种算法来确定事务的结果。2PC 由事务管理器(TM)驱动，作为完成 XA 事务的过程。

第 1 阶段：准备

在第一阶段，交易参与者通知交易协调员是他们可以提交交易还是必须回滚。

第 2 阶段：提交

在第二阶段，交易协调员决定整个交易应提交或回滚。如果任一参与者无法提交，交易必须回滚。否则，事务可以提交。协调会指示有关要做什么的资源，他们在完成此操作时通知协调员。此时，事务已完成。

11.2.12. 关于交易超时

为了保持原子性并遵循用于事务的 ACID 标准，事务的某些部分可以长时间运行。事务参与者在提交时需要锁定作为队列中数据库表或消息一部分的 XA 资源。TM 需要等待每个交易参与者回听，然后才能指示他们全部提交或回滚。硬件或网络故障可能会导致资源被无限期锁定。

事务超时可以与事务关联，以控制其生命周期。如果在事务提交前通过超时阈值或回滚，超时会导致自动回滚事务。

您可以为整个事务子系统配置默认超时值，或者您可以禁用默认超时值并根据每个事务指定超时。

11.2.13. 关于分布式交易

分布式事务是指与多个 JBoss EAP 服务器上的参与者进行的一项交易。JTS 规范要求 JTS 事务能够分布到来自不同供应商的应用程序服务器中。Jakarta Transactions 没有定义，但 JBoss EAP 支持 JBoss EAP 服务器中的分布式 Jakarta Transactions 事务。

注意

不支持不同供应商的服务器之间的事务分配。

注意

在其他应用程序服务器供应商文档中，您可能会发现术语分布式事务意味着 XA 事务。在 JBoss EAP 文档环境中，分布式事务指的是在多个 JBoss EAP 应用服务器中分发的交易。包含不同资源（如数据库资源和 Jakarta 消息传递资源）的事务在本文档中被称为 XA 事务。如需更多信息，请参阅 关于 JTS 和关于 XA 数据源和 XA 事务。

11.2.14. 关于 ORB 便携 API

对象请求代理(ORB)是一个向交易参与者、投资者、资源和其他跨多个应用服务器分发的其他服务发送和接收消息的过程。ORB 使用标准化接口描述语言(IDL)来通信和解释消息。通用对象请求代理架构(CORBA)是 JBoss EAP 中 ORB 使用的 IDL。

使用 ORB 的主要服务类型是分布式 Jakarta Transactions 系统，采用 JTS 规范。其他系统，特别是传统系统，可以选择使用 ORB 进行通信，而非其他机制，如远程 Jakarta Enterprise Beans 或 Jakarta 企业 Web 服务或 Jakarta RESTful Web 服务。

ORB 可移植性 API 提供了与 ORB 交互的机制。此 API 提供了获取对 ORB 引用的方法，以及将应用置于侦听来自 ORB 的传入连接的模式中。API 中的部分方法不受所有 ORB 支持。在这些情况下，会抛出异常。

API 由两个不同的类组成：

- com.arjuna.orbportability.orb
- com.arjuna.orbportability.oa

有关 ORB Portability API 中包含的方法和属性的详细信息，请参阅 红帽客户门户 上的 JBoss EAP Java 文档捆绑包。

11.3. 事务优化

11.3.1. 交易优化概述

JBoss EAP 交易经理(TM)包括多个优化功能，供您的应用利用：

优化有助于在特定情况下增强两阶段提交协议。通常，TM 会启动全局事务，该事务通过两阶段提交。但在某些情况下，当您优化这些事务时，TM 不需要进行完整的 2 阶段提交，因此进程速度更快。

下面详细介绍了 TM 使用的不同优化。

- 关于单阶段提交 LRCO 优化(1PC)
- 关于presumed-Abort Optimization
- 关于只读优化

11.3.2. 关于单阶段提交 LRCO 优化(1PC)

单阶段提交(1PC)

尽管交易时常遇到 2 阶段提交协议(2PC)，但在某些情况下不要求或无法满足这两个阶段。在这些情况下，您可以使用单一阶段提交(1PC)协议。当只有一个 XA 或非 XA 资源是全球交易的一部分时，使用单一阶段commnit 协议。

准备阶段通常会锁定资源，直到第二阶段处理为止。单阶段提交意味着跳过准备阶段，并且仅对资源处理提交。如果没有指定，则当全局交易仅包含一个参与者时，会自动使用单阶段提交优化。

最后资源提交优化(LRCO)

在非 XA 数据源参与 XA 事务的情况下，使用名为 Last Resource Commit Optimization(LRCO)的优化。虽然此协议允许大多数事务正常完成，但某些类型的错误可能会导致交易结果不一致。因此，仅将此方法用作最后的手段。

非 XA 资源在准备阶段结束时处理，并尝试进行提交。如果提交成功，则会写入事务日志，其余的资源将进入提交阶段。如果最后一个资源未能提交，则会回滚事务。

当在事务中使用一个本地 TX 数据源时，LRCO 会自动应用到它。

在以前的版本中，在 XA 事务中添加非 XA 资源是通过 LRCO 方法实现的。但是 LRCO 中存在一个故障窗口。使用 LRCO 方法将非 XA 资源添加到 XA 事务的步骤如下：

1. 准备 XA 事务。
2. 提交 LRCO.
3. 写入事务日志。
4. 提交 XA 事务。

如果流程在步骤 2 和步骤 3 间崩溃，这可能会导致数据不一致，且您无法提交 XA 事务。数据不一致的原因是提交了 LRCO 非 XA 资源，但没有记录有关准备 XA 资源的信息。恢复管理器将在服务器启动后回滚资源。提交标记资源(CMR)消除了此限制，并允许将非 XA 资源可靠地加入 XA 事务。

注意

CMR 属于 LRCO 优化的特殊情形，仅用于数据源。它并不适用于所有非 XA 资源。

- 关于 2-Phase 提交协议

11.3.2.1. 提交可标记资源

概述

使用 Commit Markable Resource(CMR)接口配置资源管理器的访问权限可确保在 XA(2PC)事务中可靠地加入非 XA 数据源。LRCO 算法的实施使非 XA 资源完全可以恢复。

要配置 CMR，您必须：

1. 在数据库中创建表。
2. 启用数据源可连接。
3. 添加对 事务 子系统的引用。

在数据库中创建表

事务只能包含一个 CMR 资源。您可以使用类似以下示例的 SQL 创建表。

```
SELECT xid,actionuid FROM _tableName_ WHERE transactionManagerID IN (String[])
DELETE FROM _tableName_ WHERE xid IN (byte[[]])
INSERT INTO _tableName_ (xid, transactionManagerID, actionuid) VALUES (byte[],String,byte[])
```

以下是用于为各种数据库管理系统创建表的 SQL 语法示例。

示例：Sybase Create Table Syntax

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
```

示例：甲骨文创建表语法

```
CREATE TABLE xids (xid RAW(144), transactionManagerID varchar(64), actionuid RAW(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

示例：IBM 创建表语法

```
CREATE TABLE xids (xid VARCHAR(255) for bit data not null, transactionManagerID
varchar(64), actionuid VARCHAR(255) for bit data not null)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

示例：SQL Server Create Table Syntax

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

示例：PostgreSQL 创建表语法

```
CREATE TABLE xids (xid bytea, transactionManagerID varchar(64), actionuid bytea)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

示例：MariaDB 创建表语法

```
CREATE TABLE xids (xid BINARY(144), transactionManagerID varchar(64), actionuid BINARY(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

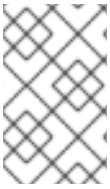
示例：Myntax(Myntax)

```
CREATE TABLE xids (xid VARCHAR(255), transactionManagerID varchar(64), actionuid
VARCHAR(255))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

启用数据源可连接

默认情况下，sources 禁用 CMR 功能。若要启用它，您必须创建或修改数据源配置，并确保 connectable 属性设为 true。以下是服务器 XML 配置文件的 datasources 部分的示例：

```
<datasource enabled="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" jta="true" use-java-context="true" connectable="true"/>
```



注意

这个功能不适用于 XA 数据源。

您还可以使用管理 CLI 启用资源管理器作为 CMR，如下所示：

```
/subsystem=datasources/data-source=ConnectableDS:add(enabled="true", jndi-
name="java:jboss/datasources/ConnectableDS", jta="true", use-java-context="true",
connectable="true", connection-url="validConnectionURL", exception-sorter-class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter", driver-name="mssql")
```

此命令会在服务器配置文件的 datasources 部分中生成以下 XML：

```
<datasource jta="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" enabled="true" use-java-context="true" connectable="true">
  <connection-url>validConnectionURL</connection-url>
  <driver>mssql</driver>
  <validation>
    <exception-sorter class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter"/>
  </validation>
</datasource>
```

注意

数据源必须定义一个有效的驱动程序。上例使用 mssql 作为 driver-name，但 mssql 驱动程序不存在。详情请参阅 JBoss EAP *配置指南* 中的 MySQL 数据源示例。

注意

在数据源配置 中使用 exception-sorter-class-name 参数。详情请参阅 JBoss EAP *配置 指南 中的数据源配置示例。*

更新现有资源以使用新的 CMR 功能

如果您只需要更新现有数据源以使用 CMR 功能，只需修改 connectable 属性：

```
/subsystem=datasources/data-source=ConnectableDS:write-attribute(name=connectable,value=true)
```

添加对事务子系统的引用

transaction 子系统 标识 CMR 能够通过进入 transaction 子系统 配置部分的条目实现的数据源，如

下所示：

```
<subsystem xmlns="urn:jboss:domain:transactions:5.0">
   ...
   <commit-markable-resources>
      <commit-markable-resource jndi-name="java:jboss/datasources/ConnectableDS">
         <xid-location name="xids" batch-size="100" immediate-cleanup="false"/>
      </commit-markable-resource>
      ...
   </commit-markable-resources>
</subsystem>
```

使用管理 CLI 可以实现相同的结果：

```
/subsystem=transactions/commit-markable-
resource=java\:jboss\/datasources\/ConnectableDS/:add(batch-size=100,immediate-
cleanup=false,name=xids)
```

注意

您必须在 事务 子系统中添加 CMR 引用后重新启动服务器。

11.3.3. 关于presumed-Abort Optimization

如果交易要回滚，它可以在本地记录此信息并通知所有参与方。该通知只是礼貌，对交易结果没有影响。联系了所有参与者后，可以删除有关交易的信息。

如果随后发生交易状态请求，则不会提供任何信息。在这种情况下，请求者假定事务已中止并回滚。这种假定的优化意味着，在决定提交交易之前，不需要使参与者的任何信息永久保留，因为在此之前的任何故障都将被视为交易的中止。

11.3.4. 关于只读优化

当要求参与者准备时，可以向协调者表明其在交易期间没有修改任何数据。不需要向这类参与者告知交易结果，因为参与者的承诺不会影响交易。提交协议的第二阶段中可以省略该只读参与者。

11.4. TRANSACTION OUTCOMES

11.4.1. 关于交易结果

事务有三种可能的结果：

Commit（提交）

如果每个交易参与者都可提交，交易协调员将指示他们这样做。如需更多信息，请参阅 关于交易提交。

回滚（Rollback）

如果任何交易参与者无法提交，或者交易协调员无法指示参与者提交，交易将被回滚。如需更多信息，请参阅关于交易回滚。

启发式结果

如果一些事务参与者提交和其他回滚，则称为启发式结果。启发性成果需要人为干预。如需更多信息，请参阅关于 Heuristic Outcomes。

11.4.2. 关于交易提交

交易参与者提交时，将其新状态变为持久状态。新状态由参与者承担交易涉及的工作创建。最常见的示例是事务成员向数据库写入记录。

提交后，交易信息将从交易协调员中删除，新编写的状态现在是持久状态。

11.4.3. 关于交易回滚

交易参与者回滚，恢复其状态以在交易开始前反映状态。回滚后，其状态与事务从未启动的情况相同。

11.4.4. 关于 Heuristic Outcomes

启发性结果或非混合性的结果是交易参与者的决定不同于交易经理的决定的情况。启发性结果可能会导致系统完整性丢失，通常需要人工干预来解决它们。不要编写依赖于它们的代码。

启发式结果通常在 2 阶段提交(2PC)协议的第二阶段发生。在个别情况下，可能会在 1PC 中产生这个结果。它们通常由底层服务器基础硬件或通信子系统失败所致。

启发式结果可能源自各种子系统或资源的超时结果，即使事务管理器和完整崩溃恢复也是如此。在需要某种形式的分布式协议的任何系统中，系统的某些部分在全局结果上存在差异的情况。

启发式结果有四种：

Heuristic rollback

提交操作无法提交资源，但所有参与者都可以回滚，因此依然能实现原子性成果。

Heuristic commit

尝试回滚操作失败，因为所有参与者都已完成。例如，如果协调员能够成功准备交易，但随后决定回滚（例如，未能更新其日志），则会出现这种情况。此时，参与者可能决定提交。

Heuristic mix

某些参与者已提交，另一些已回滚。

启发式政府.

些更新的内容未知。对于已知的对象，它们或者全部已提交，或者全部已回滚。

- 关于 2-Phase 提交协议

11.4.5. JBoss 事务错误和例外

有关 UserTransaction 类方法引发异常的详情，请查看 UserTransaction API Javadoc。

11.5. 交易生命周期概述

11.5.1. 事务生命周期

有关 Jakarta Transactions 的更多信息，请参阅关于 Jakarta Transactions。

当资源要求参与事务时，会启动一系列事件。事务管理器(TM)是一个驻留于应用程序服务器和管理事务的过程。交易参与者是指参与交易的对象。资源是数据源、雅加达消息传递连接工厂或其他 Jakarta Connectors 连接器连接。

1. 应用将启动新的事务。

要开始事务，应用从 Java Naming 和 Directory 界面获取类 UserTransaction 实例，或者从注释获取 Jakarta Enterprise Beans 实例。UserTransaction 界面包括启动、提交和回滚顶级事务的方法。新创建的事务自动与其调用线程关联。Jakarta Transactions 不支持嵌套交易，因此所有事务都是顶级事务。

当调用 UserTransaction.begin（） 方法时，Jakarta Enterprise Beans 将启动事务。此事务的默认行为可能通过使用 TransactionAttribute 注解或 ejb.xml 描述符而受到影响。在该点之后使用的任何资源都与该事务相关联。如果加入多个资源，事务将变为 XA 事务，并在提交时参与两阶段提交协议。

注意

默认情况下，事务由 Jakarta 企业 Beans 中的应用容器驱动。这称为 *容器管理事务(CMT)。*要让事务用户驱动，请将 交易管理 更改为 *Bean Managed Transaction(BMT)。*在 BMT 中，用户可使用 UserTransaction 对象来管理事务。

2. 应用修改其状态。

在下一步中，应用将执行自己的工作并更改其状态，仅对占用的资源进行更改。

3. 应用决定提交或回滚。

当应用完成更改其状态时，它将决定要提交还是回滚。它调用适当的方法，即 UserTransaction.commit（） 或 UserTransaction.rollback（）。对于 CMT，此过程会自动驱动，而对于 BMT，则必须显式调用 UserTransaction 的方法提交或回滚。

4. TM 会从其记录中删除事务。

在提交或回滚完成后，TM 会清理其记录并从事务日志中删除有关事务的信息。

故障恢复

如果资源、事务参与者或应用服务器崩溃或不可用，则 交易管理器 会在解决底层故障并且资源再次可用时处理恢复。这个过程会自动进行。如需更多信息，请参阅 XA 恢复。

11.6. 事务子系统配置

transactions 子系统 允许您配置事务管理器选项，如统计、超时值和事务记录。您还可以管理事务和查看事务统计。

如需更多信息，请参阅《JBoss EAP *配置指南* 》中的配置事务。

11.7. 练习中的事务使用

11.7.1. 事务使用情况概述

当您需要在应用中使用事务时，以下步骤非常有用。

- 控制事务
  - 开始交易
  - 提交事务
  - 回滚事务
- 在交易中处理 Heuristic Outcome
- 处理事务错误
- 事务参考

11.7.2. 控制事务

简介

本流程列表概述了控制应用中使用 Jakarta Transactions API 的不同方法。

- 开始交易

- 提交事务
- 回滚事务

11.7.2.1. 开始交易

此流程演示了如何开始新交易。无论您运行配置了 Jakarta Transactions 或 JTS 的事务管理器™，API 都相同。

1. 获取 UserTransaction 实例.

   如果 Jakarta Enterprise Beans 使用 @TransactionManagement(TransactionManagement)注释 (TransactionManagementType.BEAN) 注释，则可以使用 Java 命名和目录接口、注入或 Jakarta Enterprise Beans 上下文获取实例。

   - 使用 Java 命名和目录界面获取实例。

     ```
     new InitialContext().lookup("java:comp/UserTransaction")
     ```

   - 使用注入获取实例。

     ```
     @Resource UserTransaction userTransaction;
     ```

   - 使用 Jakarta Enterprise Beans 上下文获取实例。

     - 在无状态/状态 Bean 中：

       ```
       @Resource SessionContext ctx;
       ctx.getUserTransaction();
       ```

     - 在消息驱动型 Bean 中：

       ```
       @Resource MessageDrivenContext ctx;
       ctx.getUserTransaction()
       ```

2. 连接到数据源后，请致电 UserTransaction.begin( )。

```
try {
   System.out.println("\nCreating connection to database: "+url);
   stmt = conn.createStatement();  // non-tx statement
   try {
      System.out.println("Starting top-level transaction.");
      userTransaction.begin();
      stmtx = conn.createStatement(); // will be a tx-statement
      ...
   }
}
```

结果

事务开始。在提交或回滚事务之前，数据源的所有用途都是事务性。

有关完整示例，请参阅 Jakarta Transactions 交易示例。

注意

Jakarta Enterprise Beans（用于 CMT 或 BMT）的好处之一是，容器管理事务处理的所有内部，也就是说，您可以免于处理作为 JBoss EAP 容器中 XA 事务处理一部分的事务处理或事务分配。

11.7.2.1.1. 嵌套事务

嵌套交易允许应用创建嵌入在现有事务中的事务。在此模型中，多个子事务可以递归地嵌入到事务中。子事务可以提交或回滚，无需提交或回滚父事务。但是，提交操作的结果取决于所有交易先锋的承诺。

有关具体实施的信息，请参阅 Narayana 项目文档。

嵌套事务仅在与 JTS 规范一起使用时才可用。嵌套事务不是 JBoss EAP 应用服务器的支持功能。此外，许多数据库供应商不支持嵌套交易，因此请在向应用添加嵌套事务前咨询您的数据库供应商。

11.7.2.2. 提交事务

此流程演示了如何使用 Jakarta Transactions 进行交易。

先决条件

您必须先开始事务，然后才能提交。有关如何开始交易的详情，请参考 开始交易。

1. **对 UserTransaction 调用 commit（） 方法。**

   当您在 UserTransaction 上调用 commit（）方法时，TM 会尝试提交事务。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value) {
   EntityManager entityManager = entityManagerFactory.createEntityManager();
   try {
      userTransaction.begin();
      <!-- Perform some data manipulation using entityManager -->
      ...
      // Commit the transaction
      userTransaction.commit();
   } catch (Exception ex) {
      <!-- Log message or notify Web page -->
      ...
      try {
         userTransaction.rollback();
      } catch (SystemException se) {
         throw new RuntimeException(se);
      }
      throw new RuntimeException(ex);
   } finally {
      entityManager.close();
   }
}
```

2. **如果使用容器管理事务(CMT)，则不需要手动提交。**

   如果将 Bean 配置为使用容器管理交易，则容器将根据您在代码中配置的注解来管理您的事务生命周期。

```
@PersistenceContext
private EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateTable(String key, String value)
  <!-- Perform some data manipulation using entityManager -->
  ...
}
```

结果

您的数据源提交和您的事务终止，或者抛出异常。

注意

有关完整示例，请参阅 Jakarta Transactions 交易示例。

11.7.2.3. 回滚事务

此流程演示了如何使用 Jakarta Transactions 回滚事务。

先决条件

您必须先开始事务，然后才能回滚。有关如何开始交易的详情，请参考 开始交易。

1. 在 UserTransaction 上调用 rollback（） 方法。

当您在 UserTransaction 上调用 rollback（） 方法时，TM 会尝试回滚事务并将数据返回到之前的状态。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
   EntityManager entityManager = entityManagerFactory.createEntityManager();
   try {
      userTransaction.begin():
      <!-- Perform some data manipulation using entityManager -->
        ...
       // Commit the transaction
      userTransaction.commit();
   } catch (Exception ex) {
      <!-- Log message or notify Web page -->
      ...
      try {
         userTransaction.rollback();
      } catch (SystemException se) {
         throw new RuntimeException(se);
      }
      throw new RuntimeException(e);
   } finally {
      entityManager.close();
   }
}
```

-

2. 如果使用容器管理事务(CMT)，则不需要手动回滚事务。

   如果将 Bean 配置为使用容器管理交易，则容器将根据您在代码中配置的注解来管理您的事务生命周期。

注意

如果抛出 RuntimeException，则 CMT 会出现回滚。您还可以显式调用 setRollbackOnly 方法以获取回滚。或者，将 @ApplicationException(rollback=true)用于回滚应用异常。

结果

您的事务由 TM 回滚。

注意

有关完整示例，请参阅 Jakarta Transactions 交易示例。

11.7.3. 在交易中处理 Heuristic Outcome

启发式事务结果不常见，通常具有特殊的原因。神秘一词意味着"手动"，这就是通常必须处理这些结果的方式。有关启发式事务结果的更多信息，请参阅关于 Heuristic Outcomes。

此流程演示了如何使用 Jakarta Transactions 处理交易的启发式结果。

1. 事务中的启发式成果的原因是资源经理承诺可以提交或回滚，然后无法履行承诺。这可能是因为第三方组件、第三方组件和 JBoss EAP 之间的集成层或 JBoss EAP 本身存在问题。

   目前，导致启发式错误的最常见两个原因是环境中的瞬态故障，以及处理资源管理器的编码错误。

2. 通常，如果您的环境中出现瞬态故障，您通常会在发现启发性错误前了解它。这可能是因为网络中断、硬件故障、数据库故障、电源中断或许多其他因素造成的。

如果在压力测试期间测试环境中发现了启发性的结果，这意味着您的测试环境存在缺点。

警告

JBoss EAP 自动恢复出现故障时处于非修复状态的交易，但不试图恢复启发式交易。

3. 如果您的环境中没有明显失败，或者启发式结果很容易再现，这可能是因为编码错误。您必须联系第三方供应商，以确定解决方案是否可用。

   如果您怀疑问题在 JBoss EAP 本身的交易经理中，您必须提交支持票据。

4. 您可以使用管理 CLI 尝试手动恢复事务。如需更多信息，请参阅在 *JBoss EAP 上管理交易的 恢复事务 参与者* 一节。

5. 手动解决事务结果的过程取决于故障的确切情况。根据您的环境执行以下步骤：

   a. 确定涉及哪些资源管理器。

   b. 检查事务管理器和资源管理器的状态。

   c. 在一个或多个涉及的组件中手动强制进行日志清理和数据协调。

6. 在测试环境中，或者如果您不关注数据的完整性，请删除事务日志并重新启动 JBoss EAP 将会去除启发式结果。默认情况下，事务日志位于单机服务器的 *EAP_HOME*/standalone/data/tx-object-store/ 目录中，或者位于受管域中的 *EAP_HOME*/domain/servers/*SERVER_NAME*/data/tx-object-store/ 目录中。对于受管域，SERVE *R_NAME* 是指参与服务器组的单个服务器的名称。

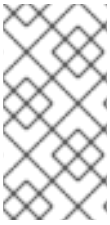
注意

事务日志的位置还取决于使用的对象存储，以及为 **object-store- relative-to** 和 **object-store -path** 参数设置的值。对于文件系统日志，如标准 **shadow** 和 **Apache ActiveMQ Artemis** 日志，将使用默认的目录位置，但在使用 **JDBC** 对象存储时，事务日志存储在数据库中。

11.7.4. Jakarta Transactions 事务错误处理

11.7.4.1. 处理事务错误

事务错误很难解决，因为它们通常依赖于时间。以下是排除错误的一些常见错误和观点：



注意

这些规则不适用于启发性错误。如果您遇到启发性错误，请参阅 在交易中处理 Heuristic Outcome，并联系红帽全球支持服务以获得帮助。

事务超时，但业务逻辑线程未注意到

当 Hibernate 无法获取用于延迟加载的数据库连接时，这种类型的错误通常会列出自身。如果频繁发生，您可以延长超时值。有关 配置事务管理器 的详情，请查看 JBoss EAP *配置指南*。

如果这不可行，您或许能够调整外部环境以更快地执行，或者将代码重组为更高效。如果您遇到超时问题，请联系红帽全球支持服务。

事务已在线程上运行，或者您收到 NotSupportedException 异常

NotSupportedException 异常 通常表示您试图嵌套 Jakarta Transactions 事务，且不受支持。如果您没有尝试嵌套事务，则可能会在线程池任务中启动另一个事务，但无需暂停或终止事务即可完成任务。

应用通常使用 UserTransaction，后者自动处理此问题。如果是这样，则框架可能存在问题。

如果您的代码确实 直接使用 事务 管理器 或交易方法，请注意提交或回滚事务时的以下行为：如果您的代码使用 TransactionManager 方法来控制您的事务，则提交或回滚事务会从当前线程中解除事务关联。但是，如果您的代码使用 事务 方法，则事务可能不会与正在运行的线程关联，而且您需要手动将其从线程中取消关联，然后再将其返回到线程池。

您无法获取第二个本地资源

如果您尝试将第二个非 XA 资源放入事务中，则会出现这个错误。如果您在事务中需要多个资源，则必须是 XA。

11.8. 事务参考

11.8.1. Jakarta 交易的事务示例

本例演示了如何开始、提交和回滚 Jakarta Transactions 事务。您需要调整连接和数据源参数以适合您的环境，并在数据库中设置两个测试表。

```
public class JDBCExample {
   public static void main (String[] args) {
      Context ctx = new InitialContext();
      // Change these two lines to suit your environment.
      DataSource ds = (DataSource)ctx.lookup("jdbc/ExampleDS");
      Connection conn = ds.getConnection("testuser", "testpwd");
      Statement stmt = null; // Non-transactional statement
      Statement stmtx = null; // Transactional statement
      Properties dbProperties = new Properties();

      // Get a UserTransaction
      UserTransaction txn = new InitialContext().lookup("java:comp/UserTransaction");

      try {
         stmt = conn.createStatement();  // non-tx statement

         // Check the database connection.
         try {
            stmt.executeUpdate("DROP TABLE test_table");
            stmt.executeUpdate("DROP TABLE test_table2");
         }
         catch (Exception e) {
            throw new RuntimeException(e);
            // assume not in database.
         }

         try {
            stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b INTEGER)");
            stmt.executeUpdate("CREATE TABLE test_table2 (a INTEGER,b INTEGER)");
         }
         catch (Exception e) {
            throw new RuntimeException(e);
         }

         try {
            System.out.println("Starting top-level transaction.");

            txn.begin();

            stmtx = conn.createStatement(); // will be a tx-statement
```

```
// First, we try to roll back changes

System.out.println("\nAdding entries to table 1.");

stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");

ResultSet res1 = null;

System.out.println("\nInspecting table 1.");

res1 = stmtx.executeQuery("SELECT * FROM test_table");

while (res1.next()) {
   System.out.println("Column 1: "+res1.getInt(1));
   System.out.println("Column 2: "+res1.getInt(2));
}
System.out.println("\nAdding entries to table 2.");

stmtx.executeUpdate("INSERT INTO test_table2 (a, b) VALUES (3,4)");
res1 = stmtx.executeQuery("SELECT * FROM test_table2");

System.out.println("\nInspecting table 2.");

while (res1.next()) {
   System.out.println("Column 1: "+res1.getInt(1));
   System.out.println("Column 2: "+res1.getInt(2));
}

System.out.print("\nNow attempting to rollback changes.");

txn.rollback();

// Next, we try to commit changes
txn.begin();
stmtx = conn.createStatement();
System.out.println("\nAdding entries to table 1.");
stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");
ResultSet res2 = null;

System.out.println("\nNow checking state of table 1.");

res2 = stmtx.executeQuery("SELECT * FROM test_table");

while (res2.next()) {
   System.out.println("Column 1: "+res2.getInt(1));
   System.out.println("Column 2: "+res2.getInt(2));
}

System.out.println("\nNow checking state of table 2.");

stmtx = conn.createStatement();

res2 = stmtx.executeQuery("SELECT * FROM test_table2");

while (res2.next()) {
```

```
                System.out.println("Column 1: "+res2.getInt(1));
                System.out.println("Column 2: "+res2.getInt(2));
            }

            txn.commit();
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);

        }
    }
    catch (Exception sysEx) {
        sysEx.printStackTrace();
        System.exit(0);
    }
  }
}
```

11.8.2. 事务 API 文档

事务 Jakarta Transactions API 文档可作为 Javadoc 在以下位置找到：

- UserTransaction - https://jakarta.ee/specifications/platform/8/apidocs/javax/transaction/UserTransaction.html

如果您使用 Red Hat CodeReady Studio 来开发应用程序，API 文档会包括在 Help 菜单中。

第 12 章 JAKARTA PERSISTENCE

12.1. 关于 JAKARTA PERSISTENCE

Jakarta Persistence 是一种 Jakarta EE 规范，用于访问、保留和管理 Java 对象或类和关系数据库之间的数据。Jakarta Persistence 规范确认透明对象或关系映射模式的兴趣和成功。它标准化任何对象或关系持久性机制所需的基本 API 和元数据。

注意

Jakarta Persistence 本身只是一种规范，而非产品；无法自行执行持久性或其他任何产品。Jakarta Persistence 仅仅是一组接口，需要实施。

12.2. 创建简单的 JPA 应用程序

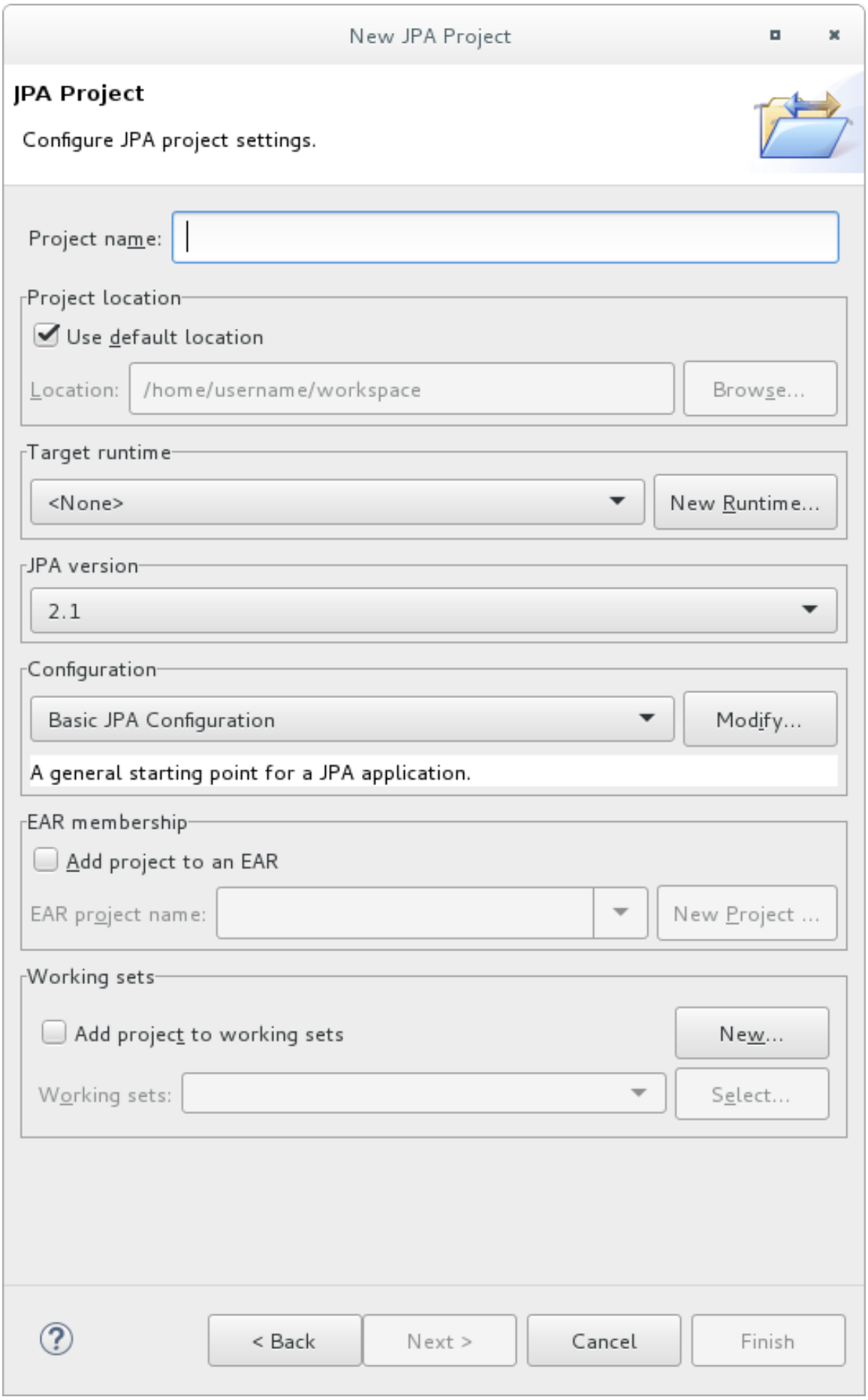
按照以下步骤在红帽 CodeReady Studio 中创建简单的 JPA 应用。

流程

1. 在红帽 CodeReady Studio 中创建 JPA 项目。

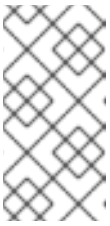
    a. 在 Red Hat CodeReady Studio 中，点击 File → New → Project。在列表中找到 JPA，对其进行扩展，然后选择 JPA Project。您将看到以下对话框：

图 12.1. 新的 JPA 项目对话框



b. 输入 项目名称。

c. 选择一个 目标运行时。如果没有目标运行时，请按照以下说明来定义新的服务器和运行时： 在 IDE 中下载、安装和设置 JBoss EAP （在 *CodeReady Studio 工具入门* 指南中）。



注意

如果您在 Red Hat CodeReady Studio 中将 Target runtime 设为 7.4 或更新的运行时版本，则您的项目与 Jakarta EE 8 规范兼容。

d. 在 JPA 版本 下，确保选择了 2.1。

e. 在 Configuration 下，选择 基本的 JPA 配置。

f. 点 Finish。

g. 若有提示，请选择您是否希望将此类项目与 JPA 透视图窗口相关联。

2. 创建和配置新的持久性设置文件。

a. 在 Red Hat CodeReady Studio 中打开 EJB 3.x 项目。

b. 在 Project Explorer 面板中，右键单击项目根目录。

c. 选择 New → Other...。

d. 从 XML 文件夹选择 XML File，然后单击 Next。

e. 选择 ejbModule/META-INF/ 文件夹作为父目录。

f.

将文件命名为 **persistence.xml**，再单击 **Next**。

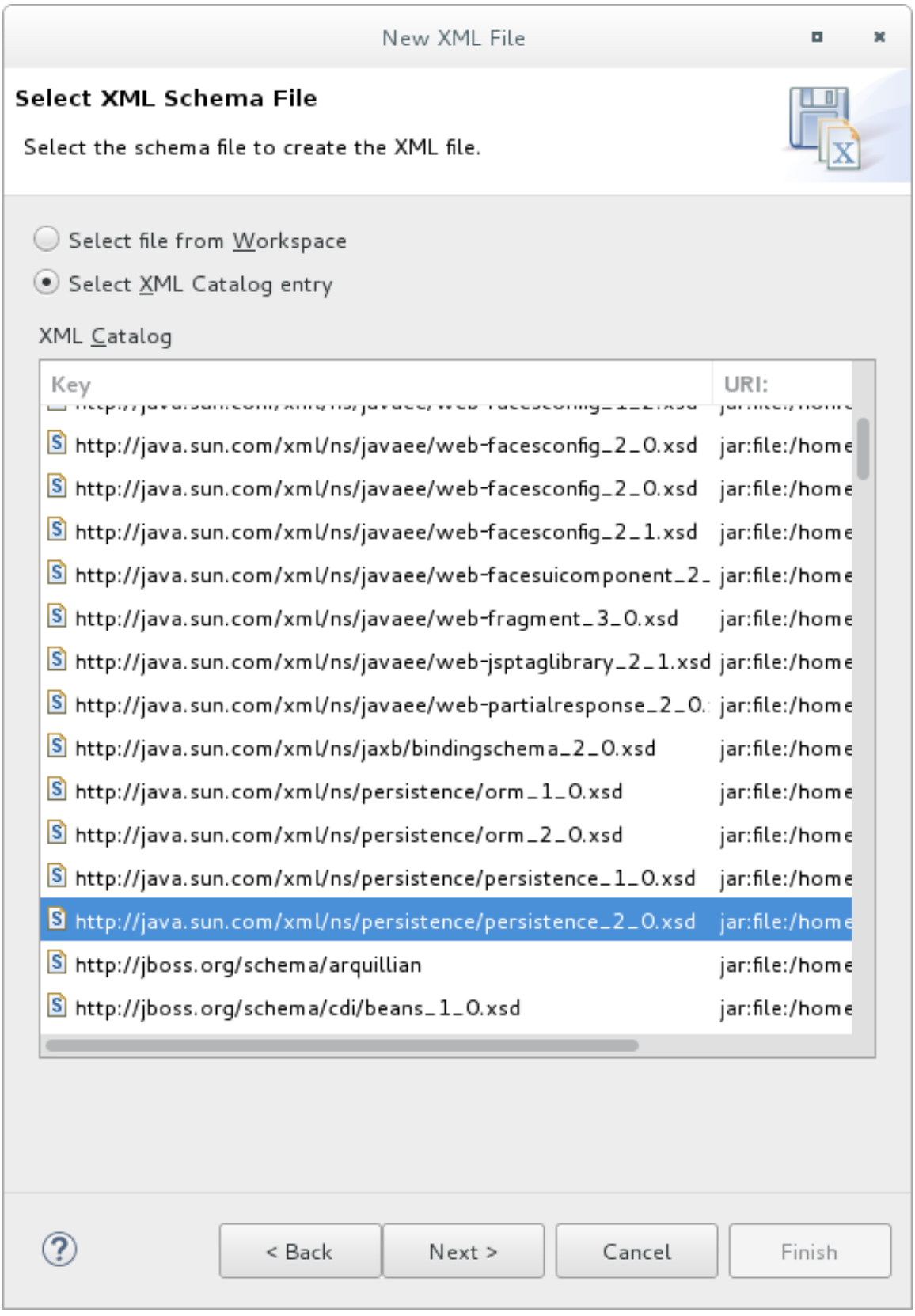
g.

从 **XML** 架构文件选择 **Create XML** 文件，然后单击 **Next**。

h.

从 **Select XML Catalog** 条目 列表中选择 **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd**，然后点击 **Next**。

图 12.2. 持久性 XML 架构



i. 单击 **Finish** 以创建该文件。**persistence.xml** 已创建在 **META-INF/** 文件夹中，并可随时配置。

示例： Persistence 设置文件

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd"
  version="2.2">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>TestApp.jar</jar-file>
    <class>org.test.Test</class>
    <shared-cache-mode>NONE</shared-cache-mode>
    <validation-mode>CALLBACK</validation-mode>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

12.3. JAKARTA PERSISTENCE ENTITIES

建立从应用程序到数据库的连接后，您可以开始将数据库中的数据映射到 Java 对象。用于针对数据库表映射的 Java 对象称为实体对象。

实体与其它实体有关系，它们通过对象-关系元数据来表示。可以使用注释直接在实体类文件中指定对象-关系元数据，也可以在应用中包含的名为 persistence.xml 的 XML 描述符文件中指定。

Java 对象到数据库的高级映射如下：

- Java 类映射到数据库表。
- Java 实例映射到数据库行。
- Java 字段映射到数据库列。

12.4. 持久性上下文

Jakarta Persistence 持久上下文包含由持久提供商管理的实体。持久上下文充当第一级别的事务缓存，以便与数据源交互。它管理实体实例及其生命周期。加载的实体在返回到应用之前被置于持久上下文中。实体更改也会置于持久性上下文中，以便在交易提交时保存到数据库中。

容器管理的持久性上下文的生命周期可以限定为事务（称为事务范围的持久性上下文），或者具有超过单个事务（称为扩展持久性上下文）的生命周期范围。PersistenceContextType 属性（具有 枚举 数据类型）用于为容器管理的实体管理器定义持久性上下文生命周期范围。持久上下文生命周期范围是在创建实体管理器 实例时定义的。

12.4.1. transaction-Scoped Persistence Context

事务范围的持久性上下文与活跃的 Jakarta Transactions 事务配合。事务提交时，持久性上下文刷新到数据源；实体对象会被分离，但可能仍被应用代码引用。预期要保存到数据源的所有实体更改必须在交易期间进行。当实体 管理器调用完成后，会在事务之外读取的实体 分离。

12.4.2. 扩展持久性上下文

扩展的持久性上下文跨越多个事务，允许数据修改排队而无需活跃的 Jakarta Transactions 事务。容器管理的扩展持久性上下文只能注入有状态会话 Bean 中。

12.5. JAKARTA PERSISTENCE ENTITYMANAGER

Jakarta Persistence 实体管理器代表与持久上下文的连接。您可以使用实体管理器从持久上下文定义的数据库读取并写入数据库。

持久性上下文通过 javax.pers istence 软件包中的 Java 注释 @Persistence Context 提供。实体管理器通过 Java 类 javax.persistence.EntityManager 提供。在任何受管 Bean 中，实体管理器 实例都可以注入，如下所示：

示例：实体管理器注入

```
@Stateless
public class UserBean {
   @PersistenceContext
   EntityManager entitymanager;
   ...
}
```

12.5.1. application-Managed EntityManager

应用管理的实体管理器提供对底层持久性提供商 org.hibernate.jpa.HibernatePersistenceProvider 的直接访问。应用程序管理的实体管理器的范围从应用创建并持续到应用关闭的时间。您可以使用 @PersistenceUnit 注释将持久性单元注入 javax.persistence.EntityManagerFactory 接口，该界面返回应用管理的实体管理器。

当您的应用需要访问特定持久性单元中不通过 Jakarta Transactions 实例传播的持久性上下文时，可以使用应用管理 的实体管理器。在这种情况下，每个实体 管理器 实例都会创建一个新的隔离持久性上下文。您的应用程序将明确创建和销毁 实体管理器 实例及其关联的 PersistenceContext。当您无法直接注入实体管理器实例时，也可使用应用管理 的实体管理器，因为 实体管理器 实例不是线程安全。EntityManagerFactory 实例是 thread-safe。

示例：应用程序管理的实体管理器

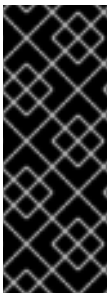
```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
}
catch (Exception e) {
    utx.rollback();
}
```

12.5.2. 容器管理实体管理器

容器管理的实体管理器管理应用的底层持久性提供商。它们可以使用事务范围的持久性上下文或扩展持久性上下文。容器管理的实体管理器根据需要创建底层持久性提供商的实例。每次创建新的底层持久性提供商 org.hibernate.jpa.HibernatePersistenceProvider 实例时，都会创建一个新的持久性上下文。

12.6. 使用实体管理器

当您的 persistent .xml 文件位于 /META-INF 目录中时，将加载实体管理器并具有与数据库的有效连接。EntityManager 属性可用于将实体管理器绑定到 JNDI，以及添加、更新、删除和查询实体。



重要

如果您计划将安全管理器与 Hibernate 一起使用，请注意，Hibernate 仅在 JBoss EAP 服务器引导 实体管理器Factory 时支持它。当应用程序引导 EntityManagerFactory 或 SessionFactory 时不支持它。有关安全 管理器的更多信息，请参阅 *如何配置服务器安全性的* Java 安全 管理器。

12.6.1. 将实体管理器绑定到 JNDI

默认情况下，JBoss EAP 不将 EntityManagerFactory 绑定到 JNDI。您可以通过设置 jboss.entity.manager .factory.jndi.name 属性，在应用的 persistence. xml 文件中显式配置此配置。此属性的值应当是您要将 EntityManagerFactory 绑定到的 JNDI 名称。

您还可以使用 jboss.entity.manager.jndi.name 属性将容器管理的事务范围实体管理器绑定到 JNDI。

示例：将 实体管理器 和 EntityManagerFactory 绑定到 JNDI

```
<property name="jboss.entity.manager.jndi.name" value="java:/MyEntityManager"/>
<property name="jboss.entity.manager.factory.jndi.name"
value="java:/MyEntityManagerFactory"/>
```

示例：使用实体 管理器保存实体

```
public User createUser(User user) {
    entityManager.persist(user);
    return user;
}
```

示例：使用实体 管理器更新实体

```
public void updateUser(User user) {
   entityManager.merge(user);
}
```

示例：使用实体 管理器删除实体

```
public void deleteUser(String user) {
   User user = findUser(username);
   if (user != null)
      entityManager.remove(user);
}
```

示例：使用实体 管理器查询实体

```
public User findUser(String username) {
   CriteriaBuilder builder = entityManager.getCriteriaBuilder();
   CriteriaQuery<User> criteria = builder.createQuery(User.class);
   Root<User> root = criteria.from(User.class);
   TypedQuery<User> query = entityManager
      .createQuery(criteria.select(root).where(
         builder.equal(root.<String> get("username"), username)));
   try {
      return query.getSingleResult();
   }
   catch (NoResultException e) {
      return null;
   }
}
```

12.7. 部署 PERSISTENCE 单元

持久性单元是逻辑分组，包括：

- 实体管理器工厂及其实体管理器的配置信息。
- 由实体管理器管理的类.
- 映射元数据，指定类到数据库的映射.

persistence.xml 文件包含持久性单元配置，包括数据源名称。JAR 文件或 /META-INF/ 目录包含 persistence.xml 文件的目录被称为持久性单元的根目录。

在 Jakarta EE 环境中，持久性单元的根必须是以下之一：

- EJB-JAR 文件
- WAR 文件的 /WEB-INF/classes/ 目录
- WAR 文件的 /WEB-INF/lib/ 目录中的 JAR 文件
- EAR 库目录中的 JAR 文件
- 应用程序客户端 JAR 文件

示例： Persistence 设置文件

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd"
  version="2.2">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

```
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>TestApp.jar</jar-file>
    <class>org.test.Test</class>
    <shared-cache-mode>NONE</shared-cache-mode>
    <validation-mode>CALLBACK</validation-mode>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

12.8. 第二级缓存

12.8.1. 关于第二级缓存

第二级缓存是在应用会话外保留信息的本地数据存储。缓存由持久性提供商管理，通过将数据与应用分开来改进运行时。

JBoss EAP 支持用于以下目的的缓存：

- Web Session Clustering
- 有状态的 Session Bean Clustering
- SSO 集群
- Hibernate 第二级缓存
- Jakarta Persistence second-level Cache

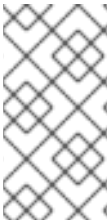
警告

每个缓存容器定义一个 repl 和一个 dist 缓存。用户应用不应直接使用这些缓存。

12.8.1.1. 默认的二级缓存供应商

Infinispan 是 JBoss EAP 的默认第二级缓存提供程序。Infinispan 是带有可选架构的分布式内存中键/值数据存储，在 Apache License 2.0 下提供。

12.8.1.1.1. 在 Persistence 单元中配置二级缓存



注意

为确保与将来的 JBoss EAP 版本兼容，应使用 Infinispan 子系统（而非 persistence.xml 属性覆盖）自定义缓存配置。

您可以使用 persistence 单元 的 shared-cache-mode 元素来配置第二级缓存。

1. 请参阅 创建一个 Simple Jakarta Persistence Application，以便在 Red Hat CodeReady Studio 中创建 persistence.xml 文件。

2. 在 persistence.xml 文件中添加以下内容：

```
<persistence-unit name="...">
  (...) <!-- other configuration -->
  <shared-cache-mode>SHARED_CACHE_MODE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
  </properties>
</persistence-unit>
```

SHARED_CACHE_MODE 元素可以使用以下值：

- **ALL ：所有实体都应被视为可缓存。**
- **新增 ：任何实体都不应被视为可缓存。**
- **ENABLE_SELECTIVE ：仅标记为可缓存的实体才应被视为可缓存。**
- **DISABLE_SELECTIVE ：除明确标记为不可缓存的实体外的所有实体都应被视为可缓存。**
- **UNSPECIFIED ：行为未定义。适用特定于提供商的默认值。**

示例：使用 persistence.xml更改 实体 和 local-query 缓存的属性

```
<persistence ... version="2.2">
   <persistence-unit ...>
      ...
      <properties>
         <!-- Values below are not recommendations. Appropriate values should be determined
based on system use/capacity. -->

         <!-- entity default overrides -->
         <property name="hibernate.cache.infinispan.entity.memory.size" value="5000"/>
         <property name="hibernate.cache.infinispan.entity.expiration.max_idle" value="300000"/> <!--
5 minutes -->
         <property name="hibernate.cache.infinispan.entity.expiration.lifespan" value="1800000"/> <!--
30 minutes -->
         <property name="hibernate.cache.infinispan.entity.expiration.wake_up_interval"
value="300000"/>  <!-- 5 minutes -->

         <!-- local-query default overrides -->
         <property name="hibernate.cache.infinispan.query.memory.size" value="5000"/>
         <property name="hibernate.cache.infinispan.query.expiration.max_idle" value="300000"/> <!--
5 minutes -->
         <property name="hibernate.cache.infinispan.query.expiration.lifespan" value="1800000"/> <!--
30 minutes -->
         <property name="hibernate.cache.infinispan.query.expiration.wake_up_interval"
value="300000"/> <!-- 5 minutes -->
      </properties>
   </persistence-unit>
</persistence>
```

表 12.1. 实体 和本地查询缓存 的属性

属性	描述
memory.size	表示 object-memory 大小。
expiration.max_idle	表示在缓存中维护缓存条目的最大空闲时间（以毫秒为单位）。
expiration.lifespan	表示缓存条目过期的最大生命周期span（以毫秒为单位）。默认为 60 秒。无限期限可以使用 -1 指定。
expiration.wake_up_interval	表示后续运行之间的间隔（以毫秒为单位）从缓存中清除过期的条目。可以使用 -1 禁用过期时间。

第 13 章 JAKARTA BEAN 验证

13.1. 关于 JAKARTA BEAN 验证

Jakarta Bean 验证是一种验证 Java 对象中数据的模型。该模型使用内置和自定义注解约束来确保应用程序数据的完整性。它还提供方法和构造器验证，以确保参数和返回值受到约束。该规范记录在 Jakarta Bean Validation 2.0 规范中。

Hibernate 验证器是 Jakarta Bean 验证的 JBoss EAP 实施。它也是 Jakarta Bean Validation 2.0 规范的参考实施。

JBoss EAP 完全遵循 Jakarta Bean Validation 2.0 规范。Hibernate 验证器还为规范提供了其他功能。

若要开始使用 Jakarta Bean Validation，请参见 JBoss EAP 附带的 bean-validation 快速入门。有关如何下载和运行快速入门的详情，请参考 JBoss EAP 入门 *指南中的使用快速入门示例*。

JBoss EAP 7.4 包括 Hibernate 验证器 6.0.x。

Hibernate 验证器 6.0.x 的特性

- Jakarta Bean Validation 2.0 定义用于实体和方法验证的元数据模型和 API。

  元数据的默认源是注释，它可以利用 XML 覆盖和扩展元数据。

  API 不与任何特定的应用层或编程模型绑定。它可用于服务器端应用程序编程和丰富的客户端 Swing 应用程序开发。

- 除了漏洞修复外，此 Hibernate 验证器版本还包含大多数常见用例的许多性能改进。

- 自版本 1.1 起，Jakarta Bean Validation 约束也可用于使用 Jakarta Bean 验证 API 的任意 Java 类型方法的参数和返回值。

- Hibernate 验证器 6.0.x 和 Jakarta Bean Validation 2.0 需要 Java 8 或更高版本。

如需更多信息，请参阅 Hibernate 验证器 6.0.17.Final - JSR 380 参考实施：参考指南。

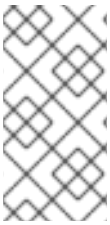
13.2. 验证约束

13.2.1. 关于验证约束

验证约束应用到 Java 元素，如字段、属性或 bean。约束通常具有一组用于设置限值的属性。有预定义的限制，可以创建自定义限制。每个约束都以注释的形式表示。

Hibernate 验证器的内置验证限制如下：Hibernate 验证器约束。

13.2.2. Hibernate 验证器约束



注意

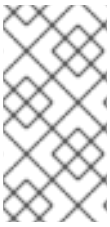
如果适用，应用程序级别的限制会导致创建数据库级别限制，下表中的 Hibernate 元数据影响 列中进行了描述。

Java 特定的验证约束

下表包含 Java 规范中定义的验证约束，包含在 javax.validation.constraints 软件包中。

注解	属性类型	运行时检查	Hibernate 元数据影响
@AssertFalse	布尔值	检查方法是否评估为 false。用于以代码而不是注释表示的约束。	none.
@AssertTrue	布尔值	检查方法是否评估为 true。用于以代码而不是注释表示的约束。	none.
@Digits(integerDigits=1)	数字的数字或字符串表示	检查该属性是否为具有最多 **整数** 数字和 **分分** 数字的数字。	定义列精度和规模.
@Future	日期或日历	检查日期是否在将来。	none.
@Max(value=)	数字的数字或字符串表示	检查值是否小于或等于 max。	在列中添加检查限制。

注解	属性类型	运行时检查	Hibernate 元数据影响
@Min(value=)	数字的数字或字符串表示	检查值是否大于或等于 Min。	在列中添加检查限制。
@NotNull		检查值是否不为空。	列不为空.
@Past	日期或日历	检查日期是否过去。	在列中添加检查限制。
@Pattern(regexp="regexp", flag=) or @Patterns({@Pattern(...)})	字符串	检查 属性是否与给定匹配标记的正则表达式匹配。请参阅 **java.util.regex.Pattern**。	none.
@Size(min=, max=)	数组、集合、映射	检查元素大小是否介于 min 和 max 之间，两个值都包含。	none.
@Valid	对象	对关联的对象递归执行验证。如果对象是 Collection 或数组，则元素会递归验证。如果对象是 Map，则值元素会递归验证。	none.



注意

参数 @Valid 是 Jakarta Bean Validation 规范的一部分，即使它位于 javax.validation.constraints 软件包中。

Hibernate 验证器特定的验证约束

下表包含特定于供应商的验证约束，它们是 org.hibernate.validator.constraints 软件包的一部分。

注解	属性类型	运行时检查	Hibernate 元数据影响
@Length(min=, max=)	字符串	检查字符串长度是否与范围匹配。	列长度将设置为最大值。
@CreditCardNumber	字符串	检查字符串是否为格式良好的信用卡号码，并派生了 Luhn 算法。	none.

注解	属性类型	运行时检查	Hibernate 元数据影响
@EAN	字符串	检查字符串是否为格式正确的 EAN 或 UPC-A 代码。	none.
@Email	字符串	检查字符串是否符合电子邮件地址规范。	none.
@NotEmpty		检查字符串是否不为空或空。检查连接是否不为空或为空。	列对于字符串不是空的。
@Range(min=, max=)	数字的数字或字符串表示	检查值是否介于 min 和 max 之间，两个值都包含。	在列中添加检查限制。

13.2.3. 使用 Jakarta Bean 验证自定义约束

Jakarta Bean Validation API 定义一组标准约束注释，如 @NotNull、@Size 等。但是，如果这些预定义的限制不够，您可以轻松创建针对特定验证要求量身定制的自定义限制。

创建 Jakarta Bean Validation 自定义约束要求您 创建一个约束注解 并实施约束验证器。以下缩写的代码示例取自 JBoss EAP 附带的 bean-validation-custom-constraint 快速入门：看到快速入门的完整工作示例。

13.2.3.1. 创建约束注解

以下示例显示了使用类 AddressValidator 中定义的一组自定义约束来验证实体 Person 的 personAddress 字段。

1. **创建实体 Person。**

示例： Person Class

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

@Entity
@Table(name = "person")
public class Person implements Serializable {
```

```
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    @Column(name = "person_id")
    private Long personId;

    @NotNull

    @Size(min = 4)
    private String firstName;

    @NotNull
    @Size(min = 4)
    private String lastName;

    // Custom Constraint @Address for bean validation
    @NotNull
    @Address
    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
    private PersonAddress personAddress;

    public Person() {

    }

    public Person(String firstName, String lastName, PersonAddress address) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.personAddress = address;
    }

    /* getters and setters omitted for brevity*/
}
```

2. 创建约束验证器文件。

示例： 地址 接口

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
import javax.validation.Constraint;
import javax.validation.Payload;

// Linking the AddressValidator class with @Address annotation.
@Constraint(validatedBy = { AddressValidator.class })
// This constraint annotation can be used only on fields and method parameters.
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
public @interface Address {

   // The message to return when the instance of MyAddress fails the validation.
   String message() default "Address Fields must not be null/empty and obey
character limit constraints";

   Class<?>[] groups() default {};

   Class<? extends Payload>[] payload() default {};
}
```

示例： PersonAddress 类

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "person_address")
public class PersonAddress implements Serializable {

   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "person_id", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Long personId;

   private String streetAddress;
   private String locality;
   private String city;
   private String state;
```

```
    private String country;
    private String pinCode;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Person person;

    public PersonAddress() {

    }

    public PersonAddress(String streetAddress, String locality, String city, String state,
String country, String pinCode) {
        this.streetAddress = streetAddress;
        this.locality = locality;
        this.city = city;
        this.state = state;
        this.country = country;
        this.pinCode = pinCode;
    }

    /* getters and setters omitted for brevity*/
}
```

13.2.3.2. 实施约束验证器

在定义了注释后，您需要创建一个能够使用 @Address 注释验证元素的约束验证器。要做到这一点，实施接口 ConstraintValidator，如下所示：

示例：地址验证器 类

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.jboss.as.quickstarts.bean_validation_custom_constraint.PersonAddress;

public class AddressValidator implements ConstraintValidator<Address, PersonAddress> {

    public void initialize(Address constraintAnnotation) {
    }

    /**
     * 1. A null address is handled by the @NotNull constraint on the @Address.
     * 2. The address should have all the data values specified.
     * 3. Pin code in the address should be of at least 6 characters.
     * 4. The country in the address should be of at least 4 characters.
```

```
     */
    public boolean isValid(PersonAddress value, ConstraintValidatorContext context) {
        if (value == null) {
            return true;
        }

        if (value.getCity() == null || value.getCountry() == null || value.getLocality() == null
            || value.getPinCode() == null || value.getState() == null || value.getStreetAddress() ==
null) {
            return false;
        }

        if (value.getCity().isEmpty()
            || value.getCountry().isEmpty() || value.getLocality().isEmpty()
            || value.getPinCode().isEmpty() || value.getState().isEmpty() ||
value.getStreetAddress().isEmpty()) {
            return false;
        }

        if (value.getPinCode().length() < 6) {
            return false;
        }

        if (value.getCountry().length() < 4) {
            return false;
        }

        return true;
    }
}
```

13.3. JAKARTA BEAN 验证配置

您可以在位于 /META-INF 目录的 validation.xml 文件中使用 XML 描述符配置 Jakarta Bean 验证。如果在类路径中存在此文件，则在创建 ValidatorFactory 时应用其配置。

示例：Jakarta Bean 验证配置文件

下例显示了 validation.xml 文件的几个配置选项。所有设置都是可选的。这些选项也可以使用 javax.validation 软件包进行配置。

```
<validation-config xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">

    <default-provider>
        org.hibernate.validator.HibernateValidator
```

```
    </default-provider>
    <message-interpolator>
        org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpolator
    </message-interpolator>
    <constraint-validator-factory>
        org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
    </constraint-validator-factory>

    <constraint-mapping>
        /constraints-example.xml
    </constraint-mapping>

    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

节点 **default-provider** 可用于选择 **Jakarta Bean Validation** 提供程序。如果类路径上有多个提供程序，这很有用。**message-interpolator** 和 **constraint-validator-factory** 属性用于自定义接口 **MessageInterpolator** 和 **ConstraintValidator Factory** 所使用的实现，它们在 **javax.validation** 软件包中定义。**constraint-mapping** 元素列出了包含实际约束配置的其他 **XML** 文件。

第 14 章 创建 JAKARTA WEBSOCKET 应用程序

Jakarta WebSocket 协议提供 Web 客户端和服务器之间的双向通信。客户端和服务器之间的通信基于事件，与基于轮询的方法相比，可以更快地处理和缩小带宽。Jakarta WebSocket 使用 JavaScript API 以及使用 Jakarta WebSocket 规格 的客户端 Jakarta WebSocket 端点，可用于 Web 应用。

首先作为 HTTP 连接在客户端和服务器之间建立连接。然后，客户端使用 Upgrade 标头请求 Jakarta WebSocket 连接。然后，所有通信都是相同的 TCP/IP 连接的全双工，而数据开销最少。由于每条消息都不包含不必要的 HTTP 标头内容，Jakarta WebSocket 通信需要更小的带宽。其结果是适用于应用程序的低延迟通信路径，需要实时响应。

JBoss EAP Jakarta WebSocket 实施为服务器端点提供全面的依赖注入支持，但它不为客户端端点提供上下文和依赖注入服务。

Jakarta WebSocket 应用程序需要以下组件和配置更改：

- 启用了 HTML 客户端或 Jakarta WebSocket 的 Java 客户端。您可以在此位置验证 HTML 客户端浏览器支持 ：http://caniuse.com/#feat=websockets
- Jakarta WebSocket 服务器端点类。
- 将项目依赖项配置为声明对 Jakarta WebSocket API 的依赖关系。

创建 Jakarta WebSocket 应用程序

以下代码示例取自 JBoss EAP 附带的 websocket-hello 快速入门。它是 Jakarta WebSocket 应用的一个简单示例，它打开连接、发送消息并关闭连接。它不实施任何其他功能或包含任何错误处理，这是真实应用所需要的。

1. 创建 JavaScript HTML 客户端。

   以下是 Jakarta WebSocket 客户端的示例。它包含这些 JavaScript 功能：

   - connect（） ：此功能创建通过 Jakarta WebSocket URI 的 Jakarta WebSocket 连接。资源位置与服务器端点类中定义的资源匹配。此功能还会截获和处理 Jakarta WebSocket onopen、onmessage、onerror 和 onclose。

- sendMessage（） ：此功能获取表单中输入的名称，创建消息并使用 WebSocket.send（）命令发送。

- disconnect（） ：此功能会发出 WebSocket.close（）命令。

- displayMessage（） ：此功能将页面上的显示消息设置为 Jakarta WebSocket endpoint 方法返回的值。

- DisplayStatus（） ：此函数显示 Jakarta WebSocket 连接状态。

示例：应用程序 index.html Code

```
<html>
  <head>
    <title>WebSocket: Say Hello</title>
    <link rel="stylesheet" type="text/css" href="resources/css/hello.css" />
    <script type="text/javascript">
      var websocket = null;
      function connect() {
        var wsURI = 'ws://' + window.location.host + '/websocket-
hello/websocket/helloName';
        websocket = new WebSocket(wsURI);
        websocket.onopen = function() {
           displayStatus('Open');
           document.getElementById('sayHello').disabled = false;
           displayMessage('Connection is now open. Type a name and click Say Hello
to send a message.');
        };
        websocket.onmessage = function(event) {
           // log the event
           displayMessage('The response was received! ' + event.data, 'success');
        };
        websocket.onerror = function(event) {
           // log the event
           displayMessage('Error! ' + event.data, 'error');
        };
        websocket.onclose = function() {
           displayStatus('Closed');
           displayMessage('The connection was closed or timed out. Please click the
Open Connection button to reconnect.');
           document.getElementById('sayHello').disabled = true;
        };
      }
      function disconnect() {
        if (websocket !== null) {
```

```
            websocket.close();
            websocket = null;
        }
        message.setAttribute("class", "message");
        message.value = 'WebSocket closed.';
        // log the event
      }
      function sendMessage() {
        if (websocket !== null) {
            var content = document.getElementById('name').value;
            websocket.send(content);
        } else {
            displayMessage('WebSocket connection is not established. Please click the
Open Connection button.', 'error');
        }
      }
      function displayMessage(data, style) {
        var message = document.getElementById('hellomessage');
        message.setAttribute("class", style);
        message.value = data;
      }
      function displayStatus(status) {
        var currentStatus = document.getElementById('currentstatus');
        currentStatus.value = status;
      }
    </script>
  </head>
  <body>
    <div>
      <h1>Welcome to Red Hat JBoss Enterprise Application Platform!</h1>
      <div>This is a simple example of a Jakarta WebSocket implementation.</div>
      <div id="connect-container">
        <div>
          <fieldset>
            <legend>Connect or disconnect using websocket :</legend>
            <input type="button" id="connect" onclick="connect();" value="Open Connection"
/>
            <input type="button" id="disconnect" onclick="disconnect();" value="Close
Connection" />
          </fieldset>
        </div>
        <div>
            <fieldset>
              <legend>Type your name below, then click the `Say Hello` button
:</legend>
              <input id="name" type="text" size="40" style="width: 40%"/>
              <input type="button" id="sayHello" onclick="sendMessage();" value="Say Hello"
disabled="disabled"/>
            </fieldset>
        </div>
        <div>Current WebSocket Connection Status: <output id="currentstatus"
class="message">Closed</output></div>
        <div>
          <output id="hellomessage" />
        </div>
      </div>
```

```
    </div>
  </body>
</html>
```

2. 创建 Jakarta WebSocket 服务器端点。

您可以使用以下任一方法之一创建 Jakarta WebSocket 服务器端点：

- 编程端点 ：端点扩展端点类。
- 标注 的端点 ：端点类使用注释来与 Jakarta WebSocket 事件交互。代码比编程端点更容易。

以下代码示例使用注解的端点方法并处理以下事件：

- @ServerEndpoint 注释将此类识别为 Jakarta WebSocket 服务器端点，并指定路径。
- 打开 Jakarta WebSocket 连接时会触发 @OnOpen 注释。
- 收到消息时会触发 @OnMessage 注释。
- 当 Jakarta WebSocket 连接关闭时，会触发 @OnClose 注释。

示例：Jkarta WebSocket Endpoint Code

```
package org.jboss.as.quickstarts.websocket_hello;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
```

```
@ServerEndpoint("/websocket/helloName")
public class HelloName {

   @OnMessage
   public String sayHello(String name) {
      System.out.println("Say hello to '" + name + "'");
      return ("Hello" + name);
   }

   @OnOpen
   public void helloOnOpen(Session session) {
      System.out.println("WebSocket opened: " + session.getId());
   }

   @OnClose
   public void helloOnClose(CloseReason reason) {
      System.out.println("WebSocket connection closed with CloseCode: " +
reason.getCloseCode());
   }
}
```

3. 在项目 POM 文件中声明 Jakarta WebSocket API 依赖项。

   如果使用 Maven，请将以下依赖项添加到 pom.xml 项目：

示例：Maven 依赖性

```
<dependency>
  <groupId>org.jboss.spec.javax.websocket</groupId>
  <artifactId>jboss-websocket-api_1.1_spec</artifactId>
  <scope>provided</scope>
</dependency>
```

JBoss EAP 随附的快速入门包括额外的 Jakarta WebSocket 客户端和端点代码示例。

第 15 章 JAKARTA 授权

15.1. 关于 JAKARTA 授权

Jakarta 授权是一种标准，它定义容器与授权服务提供商之间的合同，从而实施供容器使用的供应商。有关规格的详情，请参阅 Jakarta 授权规范。

JBoss EAP 在 security 子系统的安全功能内实施对 Jakarta 授权的支持。

15.2. 配置 JAKARTA 授权安全性

您可以通过正确配置安全域来配置 Jakarta 授权，然后修改 jboss-web.xml 以包含所需的参数。

将 Jakarta 身份验证添加到安全域

要将 Jakarta 授权支持添加到安全域，请将 Jakarta Authorization 授权策略添加到安全域的授权堆栈，并设置了 所需 标志。以下是具有 Jakarta 授权支持的安全域示例：不过，建议您从管理控制台或管理 CLI 配置安全域，而不是直接修改 XML。

示例：使用 Jakarta 身份验证的安全域

```
<security-domain name="jacc" cache-type="default">
    <authentication>
        <login-module code="UsersRoles" flag="required">
        </login-module>
    </authentication>
    <authorization>
        <policy-module code="JACC" flag="required"/>
    </authorization>
</security-domain>
```

配置 Web 应用程序以使用 Jakarta 身份验证

jboss-web.xml 文件位于部署的 WEB-INF/ 目录中，包含 Web 容器的覆盖和其他特定于 JBoss 的配置。要使用 Jakarta Authorization-enabled 安全域，您需要包含 <security-domain> 元素，同时将 <use-jboss-authorization> 元素设置为 true。以下 XML 配置为使用上面的 Jakarta Authorization 安全域：

示例：使用 Jakarta 身份验证安全域

```
<jboss-web>
    <security-domain>jacc</security-domain>
    <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

配置 Jakarta Enterprise Beans 应用程序以使用 Jakarta 身份验证

将 Jakarta Enterprise Beans 配置为使用安全域，并使用 Jakarta 授权与 Web 应用不同。对于 Jakarta Enterprise Beans，您可以在 ejb-jar.xml 描述符中声明方法权限或方法组。在 <ejb-jar> 元素中，任何子 <method-permission> 元素都包含有关 Jakarta 授权角色的信息。详情请查看以下示例配置。EJBMethodPermission 类是 Jakarta EE API 的一部分，并记录在 EJBMethodPermission 类中。

示例：Jkarta Enterprise Beans 中的 Jakarta 身份验证方法权限

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles can access any method of the
EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

您还可以使用安全域限制 Jakarta Enterprise Beans 的身份验证和授权机制，就如对 Web 应用执行的操作一样。安全域在 <security> 子元素的 jboss-ejb3.xml 描述符中声明。除了安全域外，您还可以指定 <run-as-principal>，它更改了 Jakarta Enterprise Beans 运行的主体。

示例：Jkarta Enterprise Beans 中的安全域文化

```
<ejb-jar>
   <assembly-descriptor>
      <security>
      <ejb-name>*</ejb-name>
      <security-domain>myDomain</security-domain>
      <run-as-principal>myPrincipal</run-as-principal>
      </security>
   </assembly-descriptor>
</ejb-jar>
```

使用 elytron 子系统启用 Jakarta 授权

在传统安全子系统中禁用 Jakarta 身份验证

默认情况下，应用服务器使用传统 安全 子系统来配置 Jakarta 授权策略提供商和工厂。默认配置映射到 PicketBox 中的实施。

要使用 Elytron 管理 Jakarta 授权配置，或您要安装到应用服务器的任何其他策略，您必须首先在旧安全 子系统中禁用 Jakarta 授权。为此，您可以使用以下管理 CLI 命令：

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

如果不这样做，可能会导致服务器日志中出现以下错误： MSC000004: Failure 在服务 org.wildfly.security.policy: java.lang.StackOverflowError.

定义 Jakarta 身份验证策略提供程序

elytron 子系统根据 Jakarta 授权规范提供内置策略提供程序。要创建策略供应商，您可以执行以下管理 CLI 命令：

```
/subsystem=elytron/policy=jacc:add(jacc-policy={})

reload
```

为 Web 部署启用 Jakarta 身份验证

定义了 Jakarta Authorization 策略供应商后，您可以执行以下命令为 web 部署启用 Jakarta Authorization：

```
/subsystem=undertow/application-security-domain=other:add(security-
domain=ApplicationDomain,enable-jacc=true)
```

以上命令为应用定义了一个默认安全域（如果 jboss-web.xml 文件中未提供）。如果您已经定义了 application-security-domain，只需要启用 Jakarta 授权，您可以执行以下命令：

```
/subsystem=undertow/application-security-domain=my-security-domain:write-attribute(name=enable-jacc,value=true)
```

为 Jakarta Enterprise Beans 部署启用 Jakarta 身份验证

定义了 Jakarta Authorization 策略供应商后，您可以通过执行以下命令为 Jakarta Enterprise Beans 部署启用 Jakarta Authorization：

```
/subsystem=ejb3/application-security-domain=other:add(security-domain=ApplicationDomain,enable-jacc=true)
```

以上命令为 Jakarta Enterprise Beans 定义了一个默认安全域。如果您已经定义了 application-security-domain，而只希望启用 Jakarta 授权，您可以按如下方式执行命令：

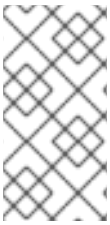
```
/subsystem=ejb3/application-security-domain=my-security-domain:write-attribute(name=enable-jacc,value=true)
```

创建自定义 Elytron 策略提供程序

当您需要自定义 java.security.Policy 时，会使用自定义策略提供程序，例如在您要与一些外部授权服务集成以检查权限时。若要创建自定义策略提供程序，您需要实施 java.security.Policy，创建并插入具有该实施的自定义模块，并在 elytron 子系统中使用模块的实施。

```
/subsystem=elytron/policy=policy-provider-a:add(custom-policy={class-name=MyPolicyProviderA, module=x.y.z})
```

如需更多信息，请参阅 策略提供程序属性。



注意

在大多数情况下，您可以使用 Jakarta 授权策略提供程序，因为它应当属于任何符合 Jakarta EE 规范的应用服务器。

第 16 章 JAKARTA 身份验证

16.1. 关于 JAKARTA 身份验证安全性

Jakarta Authentication 是 Java 应用程序的可插拔接口。有关规格的详情，请查看 Jakarta 身份验证规范。

16.2. 配置 JAKARTA 身份验证

您可以通过在安全域中添加 <authentication-jaspi> 元素来验证 Jakarta 身份验证供应商。这个配置与标准验证模块类似，但登录模块元素被包括在 <login-module-stack> 元素中。配置的结构为：

示例： authentication-jaspi Element的结构

```
<authentication-jaspi>
  <login-module-stack name="...">
    <login-module code="..." flag="...">
      <module-option name="..." value="..."/>
    </login-module>
  </login-module-stack>
  <auth-module code="..." login-module-stack-ref="...">
    <module-option name="..." value="..."/>
  </auth-module>
</authentication-jaspi>
```

登录模块本身的配置方式与标准身份验证模块相同。

基于 Web 的管理控制台不公开 JASPI 身份验证模块的配置。您必须完全停止 JBoss EAP 运行的实例，然后直接将配置添加到 *EAP_HOME*/domain/configuration/domain.xml 文件或 *EAP_HOME*/standalone/configuration/standalone.xml 文件。

16.3. 使用 ELYTRON 配置 JAKARTA 身份验证安全性

从 JBoss EAP 7.3 开始，elytron 子系统从 Jakarta 身份验证中提供了 Servlet 配置集的实施。这样可以更紧密地与 Elytron 提供的安全功能集成。

为 Web 应用程序启用 Jakarta 身份验证

若要为 Web 应用启用 Jakarta 身份验证集成，需要将 Web 应用与 Elytron http-authentication-factory 或 security-domain 关联。通过这样做，可以为部署安装 Elytron 安全处理程序，并且为部署激活 Elytron 安全框架。

为部署激活 Elytron 安全框架时，会在处理请求时查询全局注册的 AuthConfigFactory。它将识别用于该部署的 AuthConfigProvider 是否已注册。如果找到了 AuthConfigProvider，则将使用 JASPI 身份验证，而不是部署的身份验证配置。如果找不到 AuthConfigProvider，则将改为使用部署的身份验证配置。这可能会导致三种可能性之一：

- 使用 http-authentication-factory 中的身份验证机制.
- 使用 web.xml 中指定的机制.
- 如果应用尚未定义任何机制，则不执行身份验证。

对 AuthConfigFactory 进行的任何更新都会立即可用。这意味着，如果注册了 a AuthConfigProvider，并且与现有应用匹配，它将立即开始使用，而无需重新部署应用。

部署到 JBoss EAP 的所有 Web 应用都有一个安全域，将按照以下顺序解决：

1. （来自部署描述符或注释）。
2. undertow 子系统 的 default-security-domain 属性上定义的值。
3. 默认为 其他.

注意

假定此安全域是对 **PicketBox** 安全域的引用，因此激活的最后一步是确保在 **undertow** 子系统中使用 **application-security-domain** 资源将它映射到 **Elytron**。

此映射可执行以下操作之一：

- 直接引用 **elytron** 安全域，例如：

```
/subsystem=undertow/application-security-domain=MyAppSecurity:add(security-domain=ApplicationDomain)
```

- 引用 **http-authentication-factory** 资源来获取身份验证机制的实例，例如：

```
/subsystem=undertow/application-security-domain=MyAppSecurity:add(http-authentication-factory=application-http-authentication)
```

启用 **Jakarta** 身份验证集成的最小步骤是：

1. 将 **undertow** 子系统上的 **default-security-domain** 属性保留为未定义，以使其默认为 **other**。
2. 添加从其他 到 **Elytron** 安全域的 **application-security-domain** 映射。

下列步骤中与部署关联的安全域是安全域，它将包装到要 传递到 用于身份验证的 **ServerAuthModule** 实例中。

其他选项

在 **application-security-domain** 资源中添加了两个额外的属性，以便进一步控制 **Jakarta** 身份验证行为。

表 16.1. 添加到 application-security-domain 资源的属性

属性	描述
enable-jaspi	可以设置为 **false**，以使用此映射为所有部署禁用 Jakarta 身份验证支持。

属性	描述
integrated-jaspi	默认情况下，所有身份都是从安全域加载的。如果设为 **false**，则会改为创建 ad-hoc 身份。

子系统配置

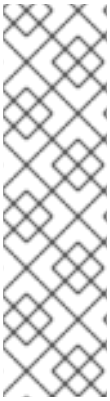
要注册将导致 AuthConfigProvider 返回部署的配置，一种方法是在 elytron 子系统中注册 jaspi-configuration。

以下命令演示了如何添加包含两个 ServerAuth 模块定义的配置。

```
/subsystem=elytron/jaspi-configuration=simple-configuration:add(layer=HttpServlet, application-context="default-host /webctx", description="Elytron Test Configuration", server-auth-modules=[{class-name=org.wildfly.security.examples.jaspi.SimpleServerAuthModule, module=org.wildfly.security.examples.jaspi, flag=OPTIONAL, options={a=b, c=d}}, {class-name=org.wildfly.security.examples.jaspi.SecondServerAuthModule, module=org.wildfly.security.examples.jaspi}])
```

这将永久保留以下配置：

```
<jaspi>
    <jaspi-configuration name="simple-configuration" layer="HttpServlet" application-context="default-host /webctx" description="Elytron Test Configuration">
        <server-auth-modules>
            <server-auth-module class-name="org.wildfly.security.examples.jaspi.SimpleServerAuthModule" module="org.wildfly.security.examples.jaspi" flag="OPTIONAL">
                <options>
                    <property name="a" value="b"/>
                    <property name="c" value="d"/>
                </options>
            </server-auth-module>
            <server-auth-module class-name="org.wildfly.security.examples.jaspi.SecondServerAuthModule" module="org.wildfly.security.examples.jaspi"/>
        </server-auth-modules>
    </jaspi-configuration>
</jaspi>
```



注意

name 属性只是允许在管理模型中引用资源的名称。

层 和应用上下文 属性用于将此配置注册到 AuthConfigFactory。可以省略这两个属性，允许通配符匹配。description 属性也是可选的，用于向 AuthConfigFactory 提供描述。

在 配置中，可以使用下列属性定义一个或多个 server-auth-module 实例：

- class-name - ServerAuthModule 的完全限定类名称。
- 模块 - 要从中加载 ServerAuthModule 的模块。
- 标志 - 用于指示此模块如何与其他模块相比运行的控制标志。
- 选项 - 初始化时要传递到 服务器AuthModule 中的配置选项。

以这种方式定义的配置立即注册到 AuthConfigFactory。任何使用与 层 和应用上下文 匹配的 Elytron 安全框架的现有部署都将立即开始使用此配置。

编程配置

Jakarta 身份验证规范中定义的 API 允许应用程序动态注册 custom AuthConfigProvider 实例。但是，该规范不提供要使用的实际实施，也不提供创建实现实例的任何标准方法。Elytron 项目包含一个简单的实用程序，可供部署用于帮助实现此目标。

以下代码示例演示了如何使用此 API 注册与上述 子系统配置中所示的配置类似的配置。

```
String registrationId =
org.wildfly.security.auth.jaspi.JaspiConfigurationBuilder.builder("HttpServlet",
servletContext.getVirtualServerName() + " " + servletContext.getContextPath())
    .addAuthModuleFactory(SimpleServerAuthModule::new, Flag.OPTIONAL,
Collections.singletonMap("a", "b"))
    .addAuthModuleFactory(SecondServerAuthModule::new)
.register();
```

例如，此代码可以在 Servlet 的 init（） 方法中执行，以注册特定于该 部署的AuthConfigProvider。在此代码示例中，还通过咨询 ServletContext 汇编了应用程序上下文。

register（） 方法返回生成的注册 ID，这也可用于直接从 AuthConfigFactory 中删除此注册。

与 Subsystem 配置 一样，此调用也立即生效，适用于所有使用 Elytron 安全框架的 Web 应用。

身份验证过程

根据 undertow 子系统中 application-security-domain 资源的配置，传递到 ServerAuthModule 的 CallbackHandler 可在以下任一模式下运行：

- 集成模式
- 非整合模式

集成模式

在集成模式下操作时，尽管 ServerAuthModule 实例将处理实际身份验证，但使用该 SecurityDomain 引用的 Security Realms，从引用的 Security Domain 中加载生成的身份 。在这种模式中，仍可以覆盖要在 servlet 容器中分配的角色。

此模式的优势在于，ServerAuthModules 能够利用 Elytron 配置来加载身份，从而无需了解这些位置即可加载存储在常规位置的身份， 如数据库和 LDAP。此外，还可以应用其他 Elytron 配置，如角色和权限映射。引用的 SecurityDomain 也可以在其他位置引用，如 SASL 身份验证或其他非 JASPI 应用，它们都由一个通用的身份存储库支持。

表 16.2. 集成模式下 CallbackHandlers 方法的操作。

操作	描述
PasswordValidationCallback	用户名和密码将与 SecurityDomain 一起使用，以执行身份验证。如果成功，将有一个经过身份验证的身份。

操作	描述
CallerPrincipalCallback	此 **回调** 用于建立授权身份或在请求到达 Web 应用后可用的身份。  **注意** 如果已通过 **PasswordValidationCallback** 建立了经过身份验证的身份，则此 **Callback 将** 解释为作为运行请求。在这种情况下，要执行授权检查，以确保身份验证的身份被授权以以此 **回调** 中指定的身份运行。如果 **PasswordValidationCallback** 没有建立经过身份验证的身份，则假定 **ServerAuthModule** 处理了身份验证步骤。 如果收到带有 null Principal 和名称的 **回调**，则： ● 如果已经建立了经过身份验证的身份，则将以该身份执行授权。 ● 如果没有建立身份，将执行匿名身份的授权。 在执行对匿名身份的授权时，**SecurityDomain** 必须已配置为授予匿名身份的 **LoginPermission**。
GroupPrincipalCallback	在这种模式中，使用安全域中配置的属性加载、角色解码和角色映射来建立身份。如果收到此 **回调**，则使用指定的组来确定分配给该身份的角色。请求将位于 servlet 容器中，这些角色仅在 servlet 容器中可见。

不集成模式

在非集成模式下运行时，ServerAuthModules 完全负责所有身份验证和身份管理。指定 的回调 可用于建立身份。生成的身份将在 SecurityDomain 上创建，但它独立于存储在引用的 SecurityRealms 中的任何身份。

此模式的优势在于，完全处理身份的 JASPI 配置可以部署到应用服务器，而无需简单的 SecurityDomain 定义。此 SecurityDomain 不需要实际包含运行时将使用的身份。此模式的缺点在于，ServerAuthModule 现在负责所有身份处理，可能会导致实施更加复杂。

表 16.3. 在非集成模式下， CallbackHandlers 方法的操作。

操作	描述
PasswordValidationCallback	此模式中不支持 **Callback**。此模式的目的是让 **ServerAuthModule** 独立于引用的 **SecurityDomain 运行**。请求验证密码并不合适。

操作	描述
CallerPrincipalCallback	此 **回调** 用于为生成的身份建立 Principal。由于 **ServerAuthModule** 正在处理所有身份检查要求，因此不会执行任何检查来验证安全域中是否存在身份并且不执行授权检查。 如果使用 null Principal 和 name 收到 **Callback**，则身份将作为匿名身份建立。由于 **ServerAuthModule** 正在做出决策，因此不会使用 **SecurityDomain** 执行授权检查。
GroupPrincipalCallback	由于该身份是在未从 **SecurityDomain** 加载的情况下创建的，所以默认情况下不会分配任何角色。如果收到此 **回调**，则在请求位于 servlet 容器中时，将生成组并分配给生成的身份。这些角色仅在 servlet 容器中可见。

validateRequest

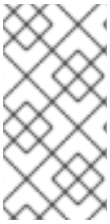
在调用 ServerAuthContext 上的 validateRequest 期间，将按照定义的顺序调用各个 ServerAuthModule 实例。也可以为每个模块指定 control 标志。此标志定义应当如何解释响应，以及处理是否应继续进入下一个服务器身份验证模块或立即返回。

控制标记

无论配置是在 elytron 子系统内提供的，还是使用 JaspiConfigurationBuilder API，都可将控制标志与每一 ServerAuthModule 关联。如果未指定，则默认为 REQUIRED。标志具有以下含义，具体取决于它们的结果：

标记	AuthStatus.SEND_SUCCESS	AuthStatus.SEND_FAILURE, AuthStatus.SEND_CONTINUE
必填	验证将继续至其余模块。只要满足其余模块的要求，该请求将被允许继续授权。	验证将继续至其余模块；但是，无论它们的结果如何，验证都无法成功，控制也会返回到客户端。
必需	验证将继续至其余模块。只要满足其余模块的要求，该请求将被允许继续授权。	请求将立即返回到客户端。
足够	验证被认定为成功且已完成，如果以前的 **Required 或 Requisite** 模块没有返回 **AuthStatus**，但 **AuthStatus.SUCCESS**.请求将继续授权受保护的资源。	验证将继续向下列出剩余模块。只有没有 **REQUIRED 或 REQUI SITE** 模块时，这个状态才会影响决定。

标记	AuthStatus.SEND_SUCCESS	AuthStatus.SEND_FAILURE, AuthStatus.SEND_CONTINUE
选填	只要任何 **所需的** 或 **Requisite** 模块尚未返回 **SUCCESS，验证将继续对剩余的模块进行验证**。这将足以使验证被视为成功，并使请求进入授权阶段和安全的资源。	验证将继续向下列出剩余模块。只有没有 **REQUIRED 或 REQUI SITE** 模块时，这个状态才会影响决定。



注意

对于所有 ServerAuthModule 实例，如果它们抛出 AuthException，则立即向客户端报告错误，而无需进一步调用模块。

secureResponse

在调用 secureResponse 时，调用每个 ServerAuthModule，但这一次以相反的顺序调用，其中模块仅在 secureResponse 中执行操作。如果模块强化了 validateResponse 中的 操作，则模块负责跟踪此操作。

control 标志对 secureResponse 处理 没有影响。当以下过程之一为 true 时，处理结束：

- **所有 ServerAuthModule 实例都已调用。**
- **模块返回 AuthStatus.SEND_FAILURE。**
- **模块抛出 AuthException.**

SecurityIdentity Creation

身份验证过程完成后，由于 回调到 Call backHandler，将创建用于 部署的 SecurityDomain 的 org.wildfly.security.auth.server.SecurityIdentityIdentity。根据 Callbacks，这是直接从 SecurityDomain 加载的身份，也可以是回调描述的临时身份。此 安全身份将与 请求相关联，方式与其它验证机制相同。

第 17 章 JAKARTA 安全

17.1. 关于 JAKARTA 安全

Jakarta Security 定义用于身份验证和身份存储的插件接口，以及提供编程安全性的可注入类型 SecurityContext 接口。有关规格的详情，请参阅 Jakarta 安全规范。

17.2. 使用 ELYTRON 配置 JAKARTA SECURITY

使用 elytron 子系统启用 Jakarta 安全

Jakarta Security 中定义的 SecurityContext 界面使用 Jakarta Authorization 策略供应商来访问当前的身份验证身份。要让您的部署使用 SecurityContext 界面，您必须配置 elytron 子系统来管理 Jakarta 授权配置，并定义默认的 Jakarta 授权策略提供商。

1. 在传统 安全 子系统中禁用 Jakarta 授权。如果 Jakarta 授权已配置为由 Elytron 管理，请跳过这一步。

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

2. 在 etlyron 子系统中定义 Jakarta 授权策略提供程序，再重新加载服务器。

```
/subsystem=elytron/policy=jacc:add(jacc-policy={})
reload
```

为 Web 应用启用 Jakarta 安全性

要为 Web 应用启用 Jakarta Security，需要将 Web 应用与 Elytron http-authentication-factory 或 security-domain 关联。这将安装 Elytron 安全处理程序，并为部署激活 Elytron 安全框架。

启用 Jakarta 安全的最低步骤是：

1. 将 undertow 子系统上的 default-security-domain 属性保留为未定义，以使其默认为 other。

2. 添加从其他 到 Elytron 安全域的 application-security-domain 映射：

```
/subsystem=undertow/application-security-domain=other:add(security-
domain=ApplicationDomain, integrated-jaspi=false)
```

当 integrated-jaspi 设置为 false 时，会动态创建临时身份。

Jakarta 安全基于 Jakarta 身份验证构建。有关配置 Jakarta 身份验证的详情，请参考 使用 Elytron 配置 Jakarta 身份验证安全性。

第 18 章 JAKARTA BATCH 应用开发

JBoss EAP 支持 Jakarta Batch，在 Jakarta EE 规范中定义。有关 Jakarta Batch 的详情，请参考 Jakarta Batch。

JBoss EAP 中的 batch-jberet 子系统有助于进行批处理配置和监控。

若要将应用配置为在 JBoss EAP 上使用批处理，您必须指定 所需的依赖项。其他用于批处理的 JBoss EAP 功能包括作业规格语言(JSL)继承和 批处理属性注入。

18.1. 所需的批处理依赖项

若要将批处理部署到 JBoss EAP，需要在应用的 pom.xml 中声明批处理所需的一些其他依赖项。下面显示了这些必需的依赖关系的示例。大多数依赖项已设置为 提供 范围，因为它们已包含在 JBoss EAP 中。

示例： pom.xml Batch Batch 依赖项

```
<dependencies>
    <dependency>
        <groupId>org.jboss.spec.javax.batch</groupId>
        <artifactId>jboss-batch-api_1.0_spec</artifactId>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.spec.javax.annotation</groupId>
        <artifactId>jboss-annotations-api_1.2_spec</artifactId>
        <scope>provided</scope>
    </dependency>

    <!-- Include your application's other dependencies. -->
    ...
</dependencies>
```

18.2. 作业规格语言(JSL)继承

JBoss EAP batch-jberet 子系统的一项功能是能够使用作业规格语言(JSL)继承来提取您的作业定义中的一些常见部分。

在相同作业 XML 文件中继承步骤和流

父元素（如 step 和 stream）标有属性 abstract ="true"，以将其从直接执行中排除。子元素包含指向父 元素的父属性。

```
<job id="inheritance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <!-- abstract step and flow -->
    <step id="step0" abstract="true">
        <batchlet ref="batchlet0"/>
    </step>

    <flow id="flow0" abstract="true">
        <step id="flow0.step1" parent="step0"/>
    </flow>

    <!-- concrete step and flow -->
    <step id="step1" parent="step0" next="flow1"/>

    <flow id="flow1" parent="flow0"/>
</job>
```

从不同的作业 XML 文件中继承步骤

子元素（如 step 和 job）包含：

- a jsl-name 属性，它指定作业 XML 文件名， 不包含父元素。
- 父 属性，它指向由 jsl-name 指定的作业 XML 文件中的父元素。

父元素标有属性 abstract ="true"，以将其从直接执行中排除。

示例： chunk-child.xml

```
<job id="chunk-child" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="chunk-child-step" parent="chunk-parent-step" jsl-name="chunk-parent">
    </step>
</job>
```

示例： block-parent.xml

```
<job id="chunk-parent" >
   <step id="chunk-parent-step" abstract="true">
      <chunk checkpoint-policy="item" skip-limit="5" retry-limit="5">
         <reader ref="R1"></reader>
         <processor ref="P1"></processor>
         <writer ref="W1"></writer>

         <checkpoint-algorithm ref="parent">
            <properties>
               <property name="parent" value="parent"></property>
            </properties>
         </checkpoint-algorithm>
         <skippable-exception-classes>
            <include class="java.lang.Exception"></include>
            <exclude class="java.io.IOException"></exclude>
         </skippable-exception-classes>
         <retryable-exception-classes>
            <include class="java.lang.Exception"></include>
            <exclude class="java.io.IOException"></exclude>
         </retryable-exception-classes>
         <no-rollback-exception-classes>
            <include class="java.lang.Exception"></include>
            <exclude class="java.io.IOException"></exclude>
         </no-rollback-exception-classes>
      </chunk>
   </step>
</job>
```

18.3. 批量属性注入

JBoss EAP batch-jberet 子系统的一项功能是能够让作业 XML 文件中定义的属性，注入到批处理工件类中的字段。可以利用 @Inject 和 @ BatchProperty 注释，将作业 XML 文件中定义的属性注入到字段中。

inject 字段可以是以下任一 Java 类型：

- java.lang.String
- java.lang.StringBuilder
- java.lang.StringBuffer
- 任何原语类型及其打包程序类型：
  - 布尔值,布尔值
  - int, Integer
  - double,Double
  - 长，长
  - Char、character
  - 浮点值，Float
  - Short
  - 字节， Byte
- java.math.BigInteger
- java.math.BigDecimal

- java.net.URL
- java.net.URI
- java.io.File
- java.util.jar.JarFile
- java.util.Date
- java.lang.Class
- java.net.Inet4Address
- java.net.Inet6Address
- java.util.List,List<?> , List<String>
- java.util.Set,Set<?> , Set<String>
- java.util.Map, Map<?, ?>, Map<String, String>, Map<String, ?>
- java.util.logging.Logger
- java.util.regex.Pattern
- javax.management.ObjectName

还支持以下数组类型：

- java.lang.String[]
- 任何原语类型及其打包程序类型：
  - boolean[], Boolean[]
  - int[]、Integer[]
  - double[], Double[]
  - 长[]、Long[]
  - char[]、Cha racter[]
  - float[], Float[]
  - short[], Short[]
  - 字节[]、Byte[]
- java.math.BigInteger[]
- java.math.BigDecimal[]
- java.net.URL[]

- java.net.URI[]
- java.io.File[]
- java.util.jar.JarFile[]
- java.util.zip.ZipFile[]
- java.util.Date[]
- java.lang.Class[]

下面是使用批处理属性注入的一些示例：

- 将数字注入到批处理类中，作为各种类型
- 将数字序列注入到批处理类中，作为各种阵列
- 将类属性注入批处理类
- 为属性注入注解的字段分配默认值

将数字注入到批处理类中，作为各种类型

示例：作业 XML 文件

```
<batchlet ref="myBatchlet">
   <properties>
      <property name="number" value="10"/>
   </properties>
</batchlet>
```

示例：Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int number;  // Field name is the same as batch property name.

    @Inject
    @BatchProperty (name = "number")  // Use the name attribute to locate the batch property.
    long asLong;  // Inject it as a specific data type.

    @Inject
    @BatchProperty (name = "number")
    Double asDouble;

    @Inject
    @BatchProperty (name = "number")
    private String asString;

    @Inject
    @BatchProperty (name = "number")
    BigInteger asBigInteger;

    @Inject
    @BatchProperty (name = "number")
    BigDecimal asBigDecimal;
}
```

将数字序列注入到批处理类中，作为各种阵列

示例：作业 XML 文件

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="weekDays" value="1,2,3,4,5,6,7"/>
    </properties>
</batchlet>
```

示例：Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int[] weekDays;  // Array name is the same as batch property name.

    @Inject
    @BatchProperty (name = "weekDays")  // Use the name attribute to locate the batch
property.
    Integer[] asIntegers;  // Inject it as a specific array type.

    @Inject
    @BatchProperty (name = "weekDays")
    String[] asStrings;

    @Inject
    @BatchProperty (name = "weekDays")
    byte[] asBytes;

    @Inject
    @BatchProperty (name = "weekDays")
    BigInteger[] asBigIntegers;

    @Inject
    @BatchProperty (name = "weekDays")
    BigDecimal[] asBigDecimals;

    @Inject
    @BatchProperty (name = "weekDays")
    List asList;

    @Inject
    @BatchProperty (name = "weekDays")
    List<String> asListString;

    @Inject
    @BatchProperty (name = "weekDays")
    Set asSet;

    @Inject
    @BatchProperty (name = "weekDays")
    Set<String> asSetString;
}
```

将类属性注入批处理类

示例：作业 XML 文件

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="myClass" value="org.jberet.support.io.Person"/>
    </properties>
</batchlet>
```

示例：Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    private Class myClass;
}
```

为属性注入注解的字段分配默认值

如果作业 XML 文件中未定义目标批处理属性，您可以为工件 Java 类中的字段分配默认值。如果将 target 属性解析为有效的值，它将注入到该字段；否则，不会注入值，并使用默认的字段值。

示例：Artifact Class

```
/**
 Comment character. If commentChar batch property is not specified in job XML file, use the
default value '#'.
 */
@Inject
@BatchProperty
private char commentChar = '#';
```

第 19 章 配置客户端

19.1. 使用 WILDFLY-CONFIG.XML 文件配置客户端

JBoss EAP 7.1 引入了 wildfly-config.xml 文件，旨在将所有客户端配置统一到一个配置文件中。

下表介绍了可以使用 JBoss EAP 中的 wildfly-config.xml 文件实现的客户端和配置类型，以及指向每种配置引用模式链接的链接。

客户端配置	架构位置/配置信息
身份验证客户端	该架构引用在 ***EAP_HOME*/docs/schema/elytron-client-1_2.xsd** 上的产品安装中提供。 模式也发布在 http://www.jboss.org/schema/jbossas/elytron-client-1_2.xsd。
	如需更多信息，请参阅使用 **wildfly-config.xml** 文件的客户端身份验证配置来了解配置示例。 如需其他信息，请参阅 *如何使用 Elytron 客户端配置客户端身份验证，并了解如何为 JBoss EAP 配置身份管理*。
Jakarta Enterprise Beans 客户端	该架构引用在 ***EAP_HOME*/docs/schema/wildfly-client-ejb_3_0.xsd** 产品安装中提供。 模式也发布在 http://www.jboss.org/schema/jbossas/wildfly-client-ejb_3_0.xsd。
	如需更多信息和示例 配置，请参阅使用 **wildfly-config.xml** 文件的 Jakarta Enterprise Beans 客户端 配置。 另一个简单示例位于 JBoss EAP *迁移指南* 的 Migrate a Jakarta Enterprise Beans 客户端至 Elytron 部分中。
HTTP 客户端	该架构引用在 ***EAP_HOME*/docs/schema/wildfly-http-client_1_0.xsd** 上的产品安装中提供。 模式也发布在 http://www.jboss.org/schema/jbossas/wildfly-http-client_1_0.xsd。

客户端配置	架构位置/配置信息
	 **注意** 此功能仅 作为技术预览提供。 如需更多信息，请参阅使用 **wildfly-config.xml** 文件的 HTTP 客户端配置来了解配置示例。
远程客户端	该架构参考在 ***EAP_HOME*/docs/schema/jboss-remoting_5_0.xsd** 产品安装中提供。 模式也发布在 http://www.jboss.org/schema/jbossas/jboss-remoting_5_0.xsd。
	如需更多信息，请参阅使用 **wildfly-config.xml 文件删除** 客户端配置 以了解更多信息和示例配置。
XNIO worker 客户端	该架构参考在 ***EAP_HOME*/docs/schema/xnio_3_5.xsd** 产品安装中提供。 模式也发布在 http://www.jboss.org/schema/jbossas/xnio_3_5.xsd。
	如需更多信息，请参阅使用 **wildfly-config.xml** 文件配置默认 XNIO 工作程序 配置。

19.1.1. 使用 wildfly-config.xml 文件的客户端身份验证配置

您可以使用位于 urn:elytron: client:1.2 命名空间中的 authentication- client 元素，以使用 wildfly-config.xml 文件配置客户端身份验证信息。这部分论述了如何使用这个元素配置客户端身份验证。

authentication-client Elements 和 Attributes

authentication-client 元素可选择性地包含以下顶层子元素，及其子元素：

- **credential-stores**
  - **credential-store**
    - **供应商**

- global
- use-service-loader

- 属性
- protection-parameter-credentials
  - key-store-reference
  - credential-store-reference
  - clear-password
  - key-pair
    - public-key-pem
    - private-key-pem
  - 证书
    - public-key-pem
  - bearer-token
  - oauth2-bearer-token

        - client-credentials
        - resource-owner-credentials

- key-stores
    - key-store
        - file
        - load-from
        - resource
        - key-store-clear-password
        - key-store-credential

- authentication-rules
    - 规则
        - match-no-user
        - match-user
        - match-protocol

- rewrite-user-name-regex
- SASL-mechanism-selector
- set-mechanism-properties
  - 属性
- 凭证
  - key-store-reference
  - credential-store-reference
  - clear-password
  - key-pair
  - 证书
  - public-key-pem
  - bearer-token
  - oauth2-bearer-token
- set-authorization-name

- 供应商
  - global
  - use-service-loader
- use-provider-sasl-factory
- use-service-loader-sasl-factory

- net-authenticator
- ssl-context-rules
  - 规则
    - match-no-user
    - match-user
    - match-protocol
    - match-host
    - match-path
    - match-port

- match-urn
- match-domain-name
- match-abstract-type

- ssl-contexts
  - default-ssl-context
  - ssl-context
    - key-store-ssl-certificate
    - trust-store
    - cipher-suite
    - 协议
    - provider-name
    - certificate-revocation-list
    - 供应商
      - global

- use-service-loader

- 供应商
  - global
  - use-service-loader

credential-stores

此可选元素定义从配置的其他位置引用的凭据存储，作为在配置中嵌入凭据的替代方法。

它可以包含任意数量 的凭据存储 元素。

示例： 凭证存储配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <credential-stores>
      <credential-store name="..." type="..." provider="..." >
        <attributes>
          <attribute name="..." value="..." />
        </attributes>
        <protection-parameter-credentials>...</protection-parameter-credentials>
      </credential-store>
    </credential-stores>
  </authentication-client>
</configuration>
```

credential-store

此元素定义从配置的其他位置引用的凭据存储。

它具有以下属性：

属性名称	属性描述
name	凭据存储的名称。此属性是必需的。
type	凭据存储的类型。此属性是可选的。
provider	用于加载凭据存储的 **java.security.Provider** 的名称。此属性是可选的。

它可包含一个以及仅包含以下各个子元素之一：

- **供应商**
- **属性**
- **protection-parameter-credentials**

属性

此元素定义用于初始化凭据存储的配置属性，并可根据需要重复。

示例： 属性 配置

```
<attributes>
  <attribute name="..." value="..." />
</attributes>
```

protection-parameter-credentials

此元素包含一个或多个凭据，组合成一个保护参数，以便在初始化凭据存储时使用。

它可包含以下一个或多个子元素，它们依赖于凭证存储实现：

- key-store-reference
- credential-store-reference
- clear-password
- key-pair
- 证书
- public-key-pem
- bearer-token
- oauth2-bearer-token

示例： Protection-parameter-credentials 配置

```
<protection-parameter-credentials>
  <key-store-reference>...</key-store-reference>
  <credential-store-reference store="..." alias="..." clear-text="..." />
  <clear-password password="..." />
  <key-pair public-key-pem="..." private-key-pem="..." />
  <certificate private-key-pem="..." pem="..." />
  <public-key-pem>...</public-key-pem>
  <bearer-token value="..." />
  <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</protection-parameter-credentials>
```

key-store-reference

此元素 目前未由 JBoss EAP 中的任何身份验证机制使用，它定义对密钥存储的引用。

它具有以下属性：

属性名称	属性描述
key-store-name	密钥存储名称。此属性是必需的。
Alias	从引用的密钥存储加载的条目的别名。这仅适用于仅包含单个条目的密钥存储，可以省略。

它可以包含其中一个，并且只能包含以下子元素之一：

- key-store-clear-password
- credential-store-reference
- key-store-credential

示例： 键存储引用 配置

```
<key-store-reference key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-reference>
```

credential-store-reference

此元素定义对凭据存储的引用。

它具有以下属性：

属性名称	属性描述
Storage	凭据存储名称。
Alias	从引用的凭据存储加载的条目的别名。这仅适用于仅包含单个条目的密钥存储，可以省略。
clear-text	明文密码。

clear-password

此元素定义明文密码。

key-pair

此元素 目前未由 JBoss EAP 中的任何身份验证机制使用，它定义一个公钥和私钥对。

它可以包含下列子元素：

- public-key-pem
- private-key-pem

public-key-pem

此元素 目前未由 JBoss EAP 中的任何身份验证机制使用，它定义 PEM 编码的公钥。

private-key-pem

这个元素定义了 PEM 编码的私钥。

certificate

此元素 目前未由 JBoss EAP 中的任何身份验证机制使用，它指定了一个证书。

它具有以下属性：

属性名称	属性描述
private-key-pem	PEM 编码的私钥。
pem	对应的证书。

bearer-token

此元素定义 bearer 令牌。

oauth2-bearer-token

此元素定义 OAuth 2 bearer 令牌。

它具有以下属性：

属性名称	属性描述
token-endpoint-uri	令牌端点的 URI。

它可包含一个以及仅包含以下各个子元素之一：

- client-credentials
- resource-owner-credentials

client-credentials

此元素定义客户端凭据。

它具有以下属性：

属性名称	属性描述
client-id	客户端 ID。此属性是必需的。

属性名称	属性描述
client-secret	客户端机密。此属性是必需的。

resource-owner-credentials

此元素定义资源所有者凭据。

它具有以下属性：

属性名称	属性描述
name	资源名称。此属性是必需的。
pasword	密码.此属性是必需的。

key-stores

此可选元素定义从配置的其他位置引用的密钥存储。

示例： 密钥存储配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="...">
        <!-- The following 3 elements specify where to load the keystore from. -->
        <file name="..." />
        <load-from uri="..." />
        <resource name="..." />
        <!-- One of the following to specify the protection parameter to unlock the keystore. -->
        <key-store-clear-password password="..." />
        <key-store-credential>...</key-store-credential>
      </key-store>
    </key-stores>
    ...
  </authentication-client>
</configuration>
```

key-store

此可选元素定义从配置的其他位置引用的密钥存储。

key-store 具有下列属性：

属性名称	属性描述
name	密钥存储的名称。此属性是必需的。
type	密钥存储类型，如 **JCEKS**。此属性是必需的。
provider	用于加载凭据存储的 **java.security.Provider** 的名称。此属性是可选的。
wrap-passwords	如果为 true，**密码** 将被换行。通过采用明确的密码内容，在 UTF-8 中对密码进行编码，并将生成的字节存储为 secret 密钥来存储密码。默认值为 **false**。

它必须正好包含下列元素之一，这些元素定义从中加载密钥存储的位置：

- **file**
- **load-from**
- **resource**

它还必须包含下列元素之一，以指定在初始化密钥存储时要使用的保护参数。

- **key-store-clear-password**
- **key-store-credential**

file

此元素指定密钥存储文件的名称。

它具有以下属性：

属性名称	属性描述
name	文件的完全限定文件名和名称。

load-from

此元素指定密钥存储文件的 URI。

它具有以下属性：

属性名称	属性描述
uri	密钥存储文件的 URI。

resource

此元素指定要从 线程 上下文类加载器加载的资源名称。

它具有以下属性：

属性名称	属性描述
name	资源名称。

key-store-clear-password

此元素指定明文密码。

它具有以下属性：

属性名称	属性描述
password	明文密码。

key-store-credential

此元素指定对另一个密钥存储的引用，该存储获取一个条目，用作访问此密钥存储的保护参数。

key-store-credential 元素具有以下属性：

属性名称	属性描述
key-store-name	密钥存储名称。此属性是必需的。
Alias	从引用的密钥存储加载的条目的别名。这仅适用于仅包含单个条目的密钥存储，可以省略。

它可以包含其中一个，并且只能包含以下子元素之一：

- **key-store-clear-password**
- **credential-store-reference**
- **key-store-credential**

示例： key-store-credential 配置

```
<key-store-credential key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-credential>
```

authentication-rules

此元素定义与出站连接匹配的规则，以应用适当的身份验证配置。需要 身份验证配置 时，访问资源的 URI 以及可选的抽象类型授权与配置中定义的规则匹配，以识别应使用哪些 身份验证配置。

此元素可以包含一个或多个子 规则 元素。

示例： authentication-rules 配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    ...
    <authentication-rules>
      <rule use-configuration="...">
        ...
      </rule>
    </authentication-rules>
    ...
  </authentication-client>
</configuration>
```

rule

此元素定义与出站连接匹配的规则，以应用适当的身份验证配置。

它具有以下属性：

属性名称	属性描述
use-configuration	规则匹配时选择的身份验证配置。

身份验证配置规则匹配独立于 SSL 上下文规则匹配。身份验证规则结构与 SSL 上下文规则结构相同，只是它引用身份验证配置，而 SSL 上下文规则引用 SSL 上下文。

它可以包含下列子元素：

- match-no-user
- match-user

- match-protocol
- match-host
- match-path
- match-port
- match-urn
- match-domain-name
- match-abstract-type

示例：身份验证 的规则 配置

```
<rule use-configuration="...">
    <!-- At most one of the following two can be defined. -->
    <match-no-user />
    <match-user name="..." />
    <!-- Each of the following can be defined at most once. -->
    <match-protocol name="..." />
    <match-host name="..." />
    <match-path name="..." />
    <match-port number="..." />
    <match-urn name="..." />
    <match-domain name="..." />
    <match-abstract-type name="..." authority="..." />
</rule>
```

match-no-user

当 URI 中没有 嵌入用户信息时， 此规则将匹配。

match-user

当 URI 中嵌入的 user-info 与此元素中指定的 name 属性匹配时，此规则将匹配。

match-protocol

当 URI 中的协议与此元素 中指定的 协议匹配时，此规则匹配。

match-host

当 URI 中指定的主机名与此元素中指定的 主机名 匹配时，此规则匹配。

match-path

当 URI 中指定的路径与此元素中指定的路径匹配时，此规则 将 匹配此规则。

match-port

当 URI 中指定的端口号与此元素中指定的端口 号 匹配时，此规则匹配。这仅与 URI 中指定的编号匹配，而不与从协议派生的任何默认端口号匹配。

match-urn

当 URI 的特定方案部分与此元素中指定的 name 属性匹配时，此规则匹配。

match-domain-name

当 URI 协议是 domain 时，此规则匹配，并且 URI 的特定部分与此元素中指定的 name 属性匹配。

match-abstract-type

当抽象类型与 name 属性匹配并且权威与此元素 中指定的 authority 属性匹配时，此规则将匹配此规则。

authentication-configurations

此元素定义将由身份验证规则选择的指定身份验证配置。

它可以包含一个或多个 配置 元素。

示例： 身份验证配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-configurations>
      <configuration name="...">
        <!-- Destination Overrides. -->
        <set-host name="..." />
        <set-port number="..." />
        <set-protocol name="..." />
        <!-- At most one of the following two elements. -->
        <set-user-name name="..." />
        <set-anonymous />
        <set-mechanism-realm name="..." />
        <rewrite-user-name-regex pattern="..." replacement="..." />
        <sasl-mechanism-selector selector="..." />
        <set-mechanism-properties>
          <property key="..." value="..." />
        </set-mechanism-properties>
        <credentials>...</credentials>
        <set-authorization-name name="..." />
        <providers>...</providers>
        <!-- At most one of the following two elements. -->
        <use-provider-sasl-factory />
        <use-service-loader-sasl-factory module-name="..." />
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```

配置

此元素定义将由身份验证规则选择的指定身份验证配置。

它可以包含下列子元素：

- **可选 的 set-host-name、set-port-number 和 set-protocol 元素可以覆盖目的地。**
- **可选 的 set-user-name 和 set-anonymous 元素是互斥的，可用于设置身份验证的名称或切换到匿名身份验证。**
- **接下来是 set-mechanism-realm-name、rewrite -user-name-regex、sasl-mechanism-selector、set-mechanism-properties、凭证、set-authorization-name 和 提**

供程序 可选元素。

- 最后两个可选 use-provider-sasl-factory 和 use-service-loader-sasl-factory 元素互斥，并定义如何发现 SASL 机制工厂进行身份验证。

set-host-name

此元素覆盖经过身份验证的调用的主机名。

它具有以下属性：

属性名称	属性描述
name	主机名。

set-port-number

此元素覆盖经过身份验证的调用的端口号。

它具有以下属性：

属性名称	属性描述
数字	端口号。

set-protocol

此元素覆盖经过身份验证的调用的协议。

它具有以下属性：

属性名称	属性描述
name	协议.

set-user-name

此元素设置用于身份验证的用户名。它不应与 set-anonymous 元素一起使用。

它具有以下属性：

属性名称	属性描述
name	用于身份验证的用户名。

set-anonymous

元素用于切换到匿名身份验证。它不应与 set-user-name 元素搭配使用。

set-mechanism-realm-name

此元素指定 SASL 机制将在需要时选择的域的名称。

它具有以下属性：

属性名称	属性描述
name	域的名称。

rewrite-user-name-regex

此元素定义正则表达式模式和替换，以重写用于身份验证的用户名。

它具有以下属性：

属性名称	属性描述
pattern	正则表达式模式.
replace	用于重写用于身份验证的用户名的替代方法。

SASL-mechanism-selector

此元素使用 org.wildfly.security.sasl.Sasl.SaslMechanismSelector.fromString(string) 方法的语法指定 SASL 机制选择器。

它具有以下属性：

属性名称	属性描述
selector	SASL 机制选择器。

有关 sasl-mechanism-selector 所需的语法的更多信息，请参阅 *如何为 JBoss EAP 配置服务器安全性的* sasl-mechanism-selector Grammar。

set-mechanism-properties

此元素可以包含一个或多个要传递到身份验证机制的属性 元素。

属性

此元素定义要传递到身份验证机制的属性。

它具有以下属性：

属性名称	属性描述
key	属性名称。
值	属性值。

credentials

此元素定义一个或多个可在身份验证期间使用的凭据。

它可包含以下一个或多个子元素，它们依赖于凭证存储实现：

- key-store-reference
- credential-store-reference

- clear-password
- key-pair
- 证书
- public-key-pem
- bearer-token
- oauth2-bearer-token。

这些子元素与 protected -parameter-credentials 元素中包含的子元素相同。有关详细信息和示例配置，请参见 Protection -parameter-credentials 元素。

set-authorization-name

此元素指定在与身份验证身份不同的授权时使用的名称。

它具有以下属性：

属性名称	属性描述
name	用于授权的名称。

use-provider-sasl-factory

此元素指定此配置中继承或定义的 java.security.Provider 实例，用于查找可用的 SASL 客户端工厂。此元素不应与 use-service-loader-sasl-factory 元素一起使用。

use-service-loader-sasl-factory

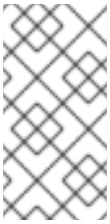
此元素指定用于利用服务加载器发现机制发现 SASL 客户端工厂的模块。如果没有指定模块，则使用载入该配置的类加载程序。此元素不应与 use-provider-sasl-factory 元素一起使用。

它具有以下属性：

属性名称	属性描述
module-name	模块的名称。

net-authenticator

此元素不包含任何配置。如果存在，org.wildfly.security.auth.util.ElytronAuthenticator 已注册到 java.net.Authenticator.setDefault(Authenticator)。当 JDK API 用于需要身份验证的 HTTP 调用时，可以使用 Elytron 身份验证客户端配置进行身份验证。



注意

由于 JDK 在第一个调用 JVM 之间缓存身份验证，因此最好仅在不需要不同凭据的单机进程上使用此方法。

ssl-context-rules

此可选元素定义 SSL 上下文规则。需要 ssl-context 时，访问资源的 URI 以及可选的抽象类型和抽象类型授权与配置中定义的规则匹配，以确定应使用哪些 ssl-context。

此元素可以包含一个或多个子 规则 元素。

示例： ssl-context-rules 配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <ssl-context-rules>
      <rule use-ssl-context="...">
        ...
      </rule>
    </ssl-context-rules>
    ...
  </authentication-client>
</configuration>
```

rule

此元素定义要在 SSL 上下文定义上匹配的规则。

它具有以下属性：

属性名称	属性描述
use-ssl-context	规则匹配时选择的 SSL 上下文定义。

SSL 上下文规则匹配独立于身份验证规则匹配。SSL 上下文规则结构与身份验证配置规则结构相同，只是它引用 SSL 上下文，而身份验证规则引用身份验证配置。

它可以包含下列子元素：

- **match-no-user**
- **match-user**
- **match-protocol**
- **match-host**
- **match-path**
- **match-port**
- **match-urn**
- **match-domain-name**

- match-abstract-type

示例：SSL 上下文 的规则 配置

```
<rule use-ssl-context="...">
  <!-- At most one of the following two can be defined. -->
  <match-no-user />
  <match-user name="..." />
  <!-- Each of the following can be defined at most once. -->
  <match-protocol name="..." />
  <match-host name="..." />
  <match-path name="..." />
  <match-port number="..." />
  <match-urn name="..." />
  <match-domain name="..." />
  <match-abstract-type name="..." authority="..." />
</rule>
```

ssl-contexts

此可选元素定义将由 SSL 上下文规则选择的 SSL 上下文定义。

示例： ssl-contexts 配置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <ssl-contexts>
      <default-ssl-context name="..."/>
      <ssl-context name="...">
        <key-store-ssl-certificate>...</key-store-ssl-certificate>
        <trust-store key-store-name="..." />
        <cipher-suite selector="..." />
        <protocol names="... ..." />
        <provider-name name="..." />
        <providers>...</providers>
        <certificate-revocation-list path="..." maximum-cert-path="..." />
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

default-ssl-context

此元素取 javax.net.ssl.SSLContext.getDefault（） 返回的 SSL Context 返回，并为它分配一个名称，以便它能够从 ssl-context-rules 引用。此元素可以重复，这意味着可以使用不同的名称来引用默认 SSL 上下文。

ssl-context

此元素定义用于连接的 SSL 上下文。

它可选择性地包含以下各个子元素之一：

- key-store-ssl-certificate
- trust-store
- cipher-suite
- 协议
- provider-name
- 供应商
- certificate-revocation-list

key-store-ssl-certificate

此元素定义对密钥存储中用于此 SSL 上下文的密钥存储条目的引用。

它具有以下属性：

属性名称	属性描述
key-store-name	密钥存储名称。此属性是必需的。
Alias	从引用的密钥存储加载的条目的别名。这仅适用于仅包含单个条目的密钥存储，可以省略。

它可包含以下子元素：

- key-store-clear-password
- credential-store-reference
- key-store-credential

此结构与 key-store-credential 配置中使用的结构几乎相同，但此处获取了密钥和证书的条目除外。但是，嵌套 的 key-store-clear-password 和 key-store-credential 元素仍然提供用于解锁条目的保护参数。

示例： key-store-ssl-certificate Configuration

```
<key-store-ssl-certificate key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-ssl-certificate>
```

trust-store

此元素是对用于初始化 TrustManager 的密钥存储的引用。

它具有以下属性：

属性名称	属性描述
key-store-name	密钥存储名称。此属性是必需的。

cipher-suite

这个元素为启用的密码套件配置过滤器。

它具有以下属性：

属性名称	属性描述
selector	用于过滤密码套件的选择器。选择器使用由 org.wildfly.security.ssl.CipherSuiteSelector.fromString(selector) 方法创建的 OpenSSL 样式密码列表字符串的格式。

示例： 使用默认过滤的 cipher-suite 配置

```
<cipher-suite selector="DEFAULT" />
```

protocol

此元素定义要支持的协议的空格分隔列表。有关可用协议列表，请参阅 *如何为 JBoss EAP 配置服务器安全性* 的客户端-ssl-context Attributes 表。红帽建议您使用 TLSv1.2。

provider-name

确定了可用的提供程序后，仅使用此元素上定义的名称的供应商。

certificate-revocation-list

此元素同时定义了证书撤销列表的路径和认证路径中可能存在的非自签发中间证书的最大数量。通过存在此元素，可以根据证书撤销列表检查同级的证书。

它具有以下属性：

属性名称	属性描述
路径	认证列表的路径.此属性是可选的。
maximum-cert-path	认证路径中可能存在的非自签发中间证书的最大数量。此属性是可选的。

供应商

此元素定义 java.security.Provider 实例在需要时如何定位。

它可以包含下列子元素：

- **global**
- **use-service-loader**

由于 authentication-client 的配置部分相互独立，因此可以在以下位置配置此元素：

示例： 供应商 配置的位置

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <providers />
    ...
    <credential-stores>
      <credential-store name="...">
        ...
        <providers />
      </credential-store>
    </credential-stores>
    ...
    <authentication-configurations>
      <authentication-configuration name="...">
        ...
        <providers />
      </authentication-configuration>
    </authentication-configurations>
    ...
    <ssl-contexts>
      <ssl-context name="...">
        ...
        <providers />
```

```
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

提供程序 配置适用于定义它的元素，以及它的任何子元素，除非被覆盖。子元素中的 提供程序 规范会覆盖其任何父元素中指定的 提供程序。如果没有指定 提供程序 配置，默认行为等同于以下内容，这赋予 Elytron 提供商高于任何全局注册供应商的优先级，但也允许使用全局注册的供应商：

示例： 供应商 配置

```
<providers>
  <use-service-loader />
  <global />
</providers>
```

global

此空元素指定使用 java.security.Security.getProviders（） 方法调用加载的全局提供程序。

use-service-loader

此空元素指定使用指定模块加载的供应商。如果没有指定模块，则使用加载身份验证客户端的类加载程序。

重要

任何 JBoss EAP 身份验证机制当前未使用的元素

Elytron 客户端配置中的 credentials 元素的以下子元素目前未用于 JBoss EAP 中的任何身份验证机制。它们可以用于您自己的自定义身份验证机制实施，但它们不受支持。

1. key-pair
2. public-key-pem
3. key-store-reference
4. certificate

19.1.2. 使用 wildfly-config.xml 文件的 Jakarta Enterprise Beans 客户端配置

您可以使用 jboss-ejb-client 元素（位于 urn:jboss:wildfly-client-ejb:3.0 命名空间）来配置 Jakarta Enterprise Beans 客户端连接、全局拦截器和使用 wildfly-config.xml 文件调用超时。本节论述了如何使用这个元素配置 Jakarta Enterprise Beans 客户端。

jboss-ejb-client Elements and Attributes

jboss-ejb-client 元素可以选择性地包含以下三个顶级子元素，及其子元素：

- invocation-timeout
- global-interceptors
  - 拦截器
- 连接

- 连接
  - 拦截器
    - 拦截器

invocation-timeout

此可选元素指定 Jakarta Enterprise Beans 调用超时。它具有以下属性：

属性名称	属性描述
秒	Jakarta Enterprise Beans 握手或方法调用请求/回复周期的超时时间（以秒为单位）。此属性是必需的。 如果方法的执行用时超过超时时间，调用将引发 **java.util.concurrent.TimeoutException** ；但是，服务器端不会被中断。

global-interceptors

此可选元素指定全局 Jakarta Enterprise Beans 客户端拦截器。它可以包含任意数量的 拦截器 元素。

拦截器

此元素用于指定 Jakarta Enterprise Beans 客户端拦截器。它具有以下属性：

属性名称	属性描述
class	实施 **org.jboss.ejb.client.EJBClientInterceptor** 接口的类的名称。此属性是必需的。
module	用于加载拦截器类的模块名称。此属性是可选的。

连接

此元素用于指定 Jakarta Enterprise Beans 客户端连接。它可以包含任意数量 的连接 元素。

连接

此元素用于指定 Jakarta Enterprise Beans 客户端连接。它可以选择包含 拦截器 元素。它具有

以下属性：

属性名称	属性描述
uri	连接的目标 URI。此属性是必需的。

拦截器

此元素用于指定 Jakarta Enterprise Beans 客户端拦截器，可以包含任意数量的 拦截器 元素。

wildfly-config.xml 文件中的 Jakarta Enterprise Beans 客户端配置示例

下例中使用 wildfly-config.xml 文件中的 jboss-ejb-client 元素配置 Jakarta Enterprise Beans 客户端连接、全局拦截器和调用超时：

```
<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <invocation-timeout seconds="10"/>
        <connections>
            <connection uri="remote+http://10.20.30.40:8080"/>
        </connections>
        <global-interceptors>
            <interceptor class="org.jboss.example.ExampleInterceptor"/>
        </global-interceptors>
    </jboss-ejb-client>
...
</configuration>
```

19.1.3. 使用 wildfly-config.xml 文件配置 HTTP 客户端

以下是如何使用 wildfly-config.xml 文件配置 HTTP 客户端的示例：

```
<configuration>
...
    <http-client xmlns="urn:wildfly-http-client:1.0">
        <defaults>
            <eagerly-acquire-session value="true" />
            <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
        </defaults>
    </http-client>
...
</configuration>
```

重要

使用 wildfly-config.xml 文件的 HTTP 客户端配置仅作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

19.1.4. 使用 wildfly-config.xml 文件调整客户端配置

您可以使用 endpoint 元素（位于 urn:jboss-remoting:5.0 命名空间中）来配置使用 wildfly-config.xml 文件的远程客户端。本节论述了如何使用这个元素配置远程客户端。

端点元素和属性

endpoint 元素可以选择性地包含以下两个顶级子元素，以及它们的子元素。

- 供应商
  - provider
- 连接
  - 连接

它还具有以下属性：

属性名称	属性描述
name	端点名称。此属性是可选的。如果没有提供，则端点名称会从系统的主机名派生（如果可能）。

供应商

此可选元素指定远程端点的传输提供程序。它可以包含任意数量 的供应商 元素。

provider

此元素定义远程传输提供程序。它具有以下属性：

属性名称	属性描述
scheme	与此提供程序对应的主 URI 方案。此属性是必需的。
Alias	也为此提供程序识别的其他 URI 方案名称的空格分隔列表。此属性是可选的。
module	包含提供程序实施的模块的名称。此属性是可选的。如果没有提供，加载 JBoss 远程的类加载程序将搜索提供商类。
class	实施传输提供程序的类的名称。此属性是可选的。如果未提供，则使用 **java.util.ServiceLoader** 功能来搜索提供程序类。

连接

此可选元素指定远程端点的连接。它可以包含任意数量 的连接 元素。

连接

此元素定义远程端点的连接。它具有以下属性：

属性名称	属性描述
destination	端点的目标 URI。此属性是必需的。
read-timeout	在对应套接字上读取操作的超时时间（以秒为单位）。此属性是可选的；但只有在定义了 **heartbeat-interval** 时才应提供它。
write-timeout	写入操作的超时时间（以秒为单位）。此属性是可选的；但只有在定义了 **heartbeat-interval** 时才应提供它。
ip-traffic-class	定义用于此连接流量的数字 IP 流量类。此属性是可选的。
tcp-keepalive	确定是否使用 TCP keepalive 的布尔值设置.此属性是可选的。
heartbeat-interval	检查心跳时要使用的间隔，以毫秒为单位。此属性是可选的。

在 wildfly-config.xml 文件中删除客户端配置示例

以下是使用 wildfly-config.xml 文件配置远程客户端的示例：

```
<configuration>
  ...
  <endpoint xmlns="urn:jboss-remoting:5.0">
    <connections>
      <connection destination="remote+http://10.20.30.40:8080" read-timeout="50" write-timeout="50"
heartbeat-interval="10000"/>
    </connections>
  </endpoint>
  ...
</configuration>
```

19.1.5. 使用 wildfly-config.xml 文件进行默认的 XNIO 工作程序配置

您可以使用 worker 元素（位于 urn:xnio:3.5 命名空间中），使用 wildfly-config.xml 文件配置 XNIO 工作程序。本节论述了如何使用这个元素配置 XNIO worker 客户端。

Worker Elements and Attributes

worker 元素可选择性地包含以下顶层子元素，及其子元素：

- daemon-threads
- worker-name
- pool-size
- task-keepalive
- io-threads
- stack-size
- outbound-bind-addresses

- bind-address

daemon-threads

这个可选元素指定 worker 和任务线程应该是守护进程线程。此元素没有内容。它具有以下属性：

属性名称	属性描述
值	指定 worker 和任务线程是否应是守护进程线程的布尔值。值 **true** 表示 worker 和任务线程应当是守护进程线程。值 **false** 表示它们不应是守护进程线程。此属性是必需的。 如果未提供此元素，则假设值为 **true**。

worker-name

此元素定义工作程序的名称。worker 名称会出现在线程转储和 Jakarta Management 中。此元素没有内容。它具有以下属性：

属性名称	属性描述
值	worker 的名称。此属性是必需的。

pool-size

此可选元素定义 worker 任务线程池的最大大小。此元素没有内容。它具有以下属性：

属性名称	属性描述
max-threads	正整数，用于指定应创建的线程的最大数量。此属性是必需的。

task-keepalive

此可选元素在任务线程过期前建立任务线程的 keep-alive 时间。它具有以下属性：

属性名称	属性描述
值	一个正整数，指定空闲线程处于活动状态所需的最少秒数。此属性是必需的。

io-threads

此可选元素决定了应当维护多少个 I/O 选择器线程。通常，这个数字应该是可用内核数的倍数的小常数。它具有以下属性：

属性名称	属性描述
值	指定 I/O 线程数量的正整数。此属性是必需的。

stack-size

此可选元素为 worker 线程建立所需的最小线程堆栈大小。这个元素只应在非常特殊的情况下定义，其中密度很高。它具有以下属性：

属性名称	属性描述
值	指定所请求堆栈大小的正整数，以字节为单位。此属性是必需的。

outbound-bind-addresses

此可选元素指定用于出站连接的绑定地址。每个绑定地址映射均包含一个目标 IP 地址块，以及用于该块内目的地连接的绑定地址和可选端口号。它可以包含任意数量的 bind-address 元素。

bind-address

此可选元素定义单独的绑定地址映射。它具有以下属性：

属性名称	属性描述
匹配	要匹配的 IP 地址块，采用 CIDR 表示法。
bind-address	如果地址块匹配，要绑定到的 IP 地址。此属性是必需的。
bind-port	如果地址块匹配，要绑定到的端口号。这个值解译为 0，即它绑定到任何端口。此属性是可选的。

wildfly-config.xml 文件中的 XNIO Worker 配置示例

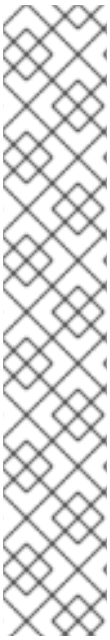
以下是如何使用 wildfly-config.xml 文件配置默认 XNIO 工作程序的示例。

```
<configuration>
  ...
  <worker xmlns="urn:xnio:3.5">
    <io-threads value="10"/>
```

```
    <task-keepalive value="100"/>
  </worker>
  ...
</configuration>
```

附录 A. 参考资料

A.1. 提供的 UNDERTOW 处理程序



注意

若要获得处理程序的完整列表，您必须在与 JBoss EAP 安装中的 Undertow 核心匹配的版本中检查 Undertow 核心的源 JAR 文件。您可以从 JBoss EAP Maven 存储库 下载 Undertow 核心源 JAR 文件，然后引用 /io/undertow/server/handlers/ 目录中可用的处理程序。

您可以通过搜索 server.log 文件中在 JBoss EAP 服务器启动期间打印的 INFO 消息来验证 JBoss EAP 的当前安装中使用的 Undertow 核心版本，类似于以下示例中所示的信息：

```
INFO [org.wildfly.extension.undertow] (MSC service thread 1-1) WFLYUT0003:
Undertow 1.4.18.Final-redhat-1 starting
```

AccessControlListHandler

类 **Name: io.undertow.server.handlers.AccessControlListHandler**

名称： access-control

可以根据远程对等点的属性来接受或拒绝请求的处理程序。

表 A.1. 参数

Name	描述
acl	ACL 规则.这个参数是必需的。
attribute	交换属性字符串.这个参数是必需的。
default-allow	指定处理程序是否默认接受或拒绝请求的布尔值.默认值为 **false**。

AccessLogHandler

类 **Name: io.undertow.server.handlers.accesslog.AccessLogHandler**

名称： access-log

访问日志处理程序.此处理程序根据提供的格式字符串生成访问日志消息，并将这些消息传递到提供的 AccessLogReceiver 中。

此处理程序可以记录通过 ExchangeAttribute 机制提供的任何属性。

此工厂为以下模式生成令牌处理程序：

表 A.2. pattern

pattern	描述
%a	远程 IP 地址
%A	本地 IP 地址
%b	发送字节，除 HTTP 标头或 **-** 如果没有发送字节
%B	发送的字节数，HTTP 标头除外
%h	远程主机名
%H	请求协议
%l	远程逻辑用户名 from **identd** （始终返回 **-**）
%m	请求方法
%p	本地端口
%q	查询字符串（除 **?** 字符外）
%r	请求的第一行
%s	响应的 HTTP 状态代码
%t	日期和时间，采用通用日志格式格式
%u	经过身份验证的远程用户
%U	请求的 URL 路径

pattern	描述
%v	本地服务器名称
%D	处理请求所需的时间，以毫秒为单位
%T	处理请求的时间（以秒为单位）
%I	当前 Request 线程名称（稍后与堆栈追踪进行比较）
common	**%h %l %u %t "%r" %s %b**
combined	**%h %l %u %t "%r" %s %b "%{i,Referer}" "%{i,User-Agent}"**

也支持从 Cookie、传入标头或会话中写入信息。

它使用 Apache 语法建模：

- **用于传入标头的 %{I,xx}**
- **用于传出响应标头的 %{O,xxx}**
- **特定 Cookie %{C,xxx}**
- **%{R,xxx} where xxx 是 ServletRequest中的属性**
- **%{s,xxx} where xxx 是 HttpSession中的属性**

表 A.3. 参数

Name	描述
格式	用于生成日志消息的格式。这是 默认参数。

AllowedMethodsHandler

将某些 HTTP 方法列入白名单的处理程序。只有允许的方法集中具有方法的请求才可以继续。

类 Name: io.undertow.server.handlers.AllowedMethodsHandler

name: allowed-methods

表 A.4. 参数

Name	描述
方法	允许的方法，如 GET、POST 和PUT 等。这是 默认参数。

BlockingHandler

启动阻止请求的 HttpHandler。如果线程当前在分配的 I/O 线程中运行。

类 Name: io.undertow.server.handlers.BlockingHandler

name：block ing

此处理程序没有参数。

ByteRangeHandler

范围请求的处理程序.这是一个通用处理程序，可以处理对固定内容长度的任何资源的范围请求，例如设置了 content-length 标头的任何资源。这不一定是处理范围请求的最有效方式，因为会生成完整的内容然后丢弃。目前，此处理程序只能处理简单、单一范围的请求。如果请求多个范围，则忽略 Range 标头。

类 Name: io.undertow.server.handlers.ByteRangeHandler

名称： 字节-range

表 A.5. 参数

Name	描述
send-accept-ranges	是否发送接受范围的布尔值.这是 默认参数。

CanonicalPathHandler

此处理程序将相对路径转换为规范路径。

类名称： io.undertow.server.handlers.CanonicalPathHandler

名称： canonical-path

此处理程序没有参数。

DisableCacheHandler

通过浏览器和代理禁用响应缓存的处理程序。

类 Name: io.undertow.server.handlers.DisableCacheHandler

name: disable-cache

此处理程序没有参数。

DisallowedMethodsHandler

将某些 HTTP 方法列入黑名单的处理程序。

class Name: io.undertow.server.handlers.DisallowedMethodsHandler

name: disallowed-methods

表 A.6. 参数

Name	描述
方法	禁止的方法，如 **GET**、**POST** 和**PUT** 等。这是 默认参数。

EncodingHandler

此处理程序作为内容编码实施的基础。编码处理程序作为委托添加到此处理程序，具有指定的服务器端优先级。

q 值将用于确定正确的处理程序。如果请求没有 q 值，服务器会选择优先级最高的处理程序作为要使用的编码。

如果没有处理程序匹配，则假设身份编码。如果因为 q 值为 0 而明确禁止身份编码，处理程序会设置响应代码 406(Not Acceptable) 并返回。

类 **Name: io.undertow.server.handlers.encoding.EncodingHandler**

名称： 压缩

此处理程序没有参数。

FileErrorPageHandler

从磁盘提供 文件的处理程序，以充当错误页面。默认情况下，此处理程序不提供任何响应代码，您必须配置它响应的代码。

类 **Name: io.undertow.server.handlers.error.FileErrorPageHandler**

name: error-file

表 A.7. 参数

Name	描述
file	文件的位置，以用作错误页面.

Name	描述
response-codes	导致重定向到定义错误页面文件的响应代码列表。

HttpTraceHandler

处理 HTTP 追踪请求的处理程序。

类 Name: io.undertow.server.handlers.HttpTraceHandler

name: trace

此处理程序没有参数。

IPAddressAccessControlHandler

可以根据远程对等点的 IP 地址接受或拒绝请求的处理程序。

类 Name: io.undertow.server.handlers.IPAddressAccessControlHandler

名称： ip-access-control

表 A.8. 参数

Name	描述
acl	代表访问控制列表的字符串。这是 默认参数。
failure-status	代表状态代码的整数，以便在被拒绝的请求时返回。
default-allow	代表默认是否允许的布尔值.

JDBCLogHandler

类 Name: io.undertow.server.handlers.JDBCLogHandler

名称： jdbc-access-log

表 A.9. 参数

Name	描述
格式	指定 JDBC 日志模式。默认值为 **common**。您还可以使用 **combined**，将 VirtualHost、request 方法、引用器和用户代理信息添加到日志消息。
datasource	要日志的数据源的名称。此参数是必需的，它是 默认参数。
tableName	表名称.
remoteHostField	远程主机地址.
userField	用户名.
timestampField	时间戳.
virtualHostField	VirtualHost.
methodField	方法.
queryField	查询.
statusField	状态.
bytesField	字节.
refererField	推荐者.
userAgentField	用户代理.

LearningPushHandler

构建浏览器请求的资源缓存的处理程序，并使用服务器推送在受支持时推送它们。

类 **Name: io.undertow.server.handlers.LearningPushHandler**

名称： learning-push

表 A.10. 参数

Name	描述
max-age	代表缓存条目最长时间的整数。
max-entries	代表最大缓存条目数的整数

LocalNameResolvingHandler

执行 DNS 查找以解析本地地址的处理程序。当前端服务器发送 X-forwarded-host 标头或使用 AJP 时，可以创建未解析的本地地址。

类 Name: io.undertow.server.handlers.LocalNameRe resolveHandler

Name: resolve-local-name

此处理程序没有参数。

PathSeparatorHandler

将 URL 中的非斜杠分隔符字符转换为斜杠的处理程序。通常，这将反斜杠转换为 Windows 系统上的斜杠。

类 Name: io.undertow.server.handlers.PathSeparatorHandler

名称： path-separator

此处理程序没有参数。

PeerNameResolvingHandler

执行反向 DNS 查找的处理程序，以解析对等地址。

类 Name: io.undertow.server.handlers.PeerNameRe resolveHandler

Name: resolve-peer-name

此处理程序没有参数。

ProxyPeerAddressHandler

将对等地址设置为 X-Forwarded-For 标头值的处理程序。这只在始终设置此标头的代理后面使用，否则攻击者便可以伪造其对等地址。

类 Name: io.undertow.server.handlers.ProxyPeerAddressHandler

name: proxy-peer-address

此处理程序没有参数。

RedirectHandler

通过 302 重定向重定向到指定位置的重定向处理程序。该位置指定为交换属性字符串。

类名称： io.undertow.server.handlers.RedirectHandler

name： redirect

表 A.11. 参数

Name	描述
值	重定向的目标.这是 默认参数。

RequestBufferingHandler

缓冲所有请求数据的处理程序。

类 Name: io.undertow.server.handlers.RequestBufferingHandler

名称： buffer-request

表 A.12. 参数

Name	描述
buffers	定义最大缓冲区数的整数。这是 默认参数。

RequestDumpingHandler

将交换转储到日志的处理程序。

类 **Name: io.undertow.server.handlers.RequestDumpingHandler**

名称： dump-request

此处理程序没有参数。

RequestLimitingHandler

限制并发请求的最大数量的处理程序。超出限制的请求将阻止，直到上一个请求完成为止。

类 **Name: io.undertow.server.handlers.RequestLimitingHandler**

名称： request-limit

表 A.13. 参数

Name	描述
requests	代表最大并发请求数的整数。这是 默认参数，是必需的。

ResourceHandler

服务资源的处理程序。

类 **Name: io.undertow.server.handlers.resource.ResourceHandler**

name: resource

表 A.14. 参数

Name	描述
位置	资源位置.这是 默认参数，是必需的。
allow-listing	布尔值，确定是否允许目录列表.

ResponseRateLimitingHandler

将下载率限制为一定数量的字节/时间的处理程序.

类 **Name: io.undertow.server.handlers.ResponseRateLimitingHandler**

Name: response-rate-limit

表 A.15. 参数

Name	描述
字节	限制下载率的字节数。这个参数是必需的。
time	限制下载率的时间（以秒为单位）。这个参数是必需的。

SetHeaderHandler

设置固定响应标头的处理程序。

类 **Name: io.undertow.server.handlers.SetHeaderHandler**

name: header

表 A.16. 参数

Name	描述
标头	标头属性的名称.这个参数是必需的。
值	标头属性的值.这个参数是必需的。

SSLHeaderHandler

在连接上设置 SSL 信息的处理程序基于以下标头：

- SSL_CLIENT_CERT
- SSL_CIPHER
- SSL_SESSION_ID

如果此处理程序存在于链中，它将始终覆盖 SSL 会话信息，即使这些标头不存在。

此处理程序 *必须* 仅用于反向代理后面的服务器，其中反向代理已配置为始终 *为每个* 请求设置这些标头，如果没有 SSL 信息，则使用这些名称剥离现有的标头。否则，恶意客户端可能会欺骗 SSL 连接。

类 Name: io.undertow.server.handlers.SSLHeaderHandler

name: ssl-headers

此处理程序没有参数。

StuckThreadDetectionHandler

此处理程序检测到需要很长时间处理的请求，这可能表示处理线程卡时卡住。

类 Name: io.undertow.server.handlers.StuckThreadDetectionHandler

Name: stuck-thread-detector

表 A.17. 参数

Name	描述

Name	描述
threshhold	以秒为单位的整数值，用于确定请求处理所需的时间的阈值。默认值为 **600**（10 分钟）。这是 默认参数。

URLDecodingHandler

将 URL 和查询参数解码到指定的 charset 的处理程序。如果使用此处理程序，您必须将 UndertowOptions.DECODE_URL 参数设置为 false。

这不如使用在 UTF-8 解码器中构建的解析器的效率。除非您需要解码 UTF-8 以外的内容，否则您应该改为依赖解析器解码。

类 **Name: io.undertow.server.handlers.URLDecodingHandler**

name: url-decoding

表 A.18. 参数

Name	描述
charset	charset 可解码.这是 默认的参数，是必需的。

A.2. 持久性单元属性

持久性单元定义支持下列属性，这些属性可以从 persistence.xml 文件进行配置：

属性	描述
jboss.as.jpa.providerModule	persistence 提供程序模块的名称。默认为 **org.hibernate**。如果持久提供程序与应用一起打包，则应为应用名称。
jboss.as.jpa.adapterModule	帮助 JBoss EAP 与持久提供商配合工作的集成类名称。
jboss.as.jpa.adapterClass	集成适配器的类名称。
jboss.as.jpa.managed	设置为 **false**，以禁用容器管理的 Jakarta Persistence 对 persistence 单元的访问。默认值为 **true**。

属性	描述
jboss.as.jpa.classtransformer	设置为 **false**，为 persistence 单元禁用类转换器。默认值为 **true**，允许类转换。 Hibernate 还需要 persistence 单元属性 **hibernate.ejb.use_class_enhancer** 才会启用 类 转换。
jboss.as.jpa.scopedname	指定要使用的合格应用程序范围的持久性单元名称。默认情况下，这会集中设置为应用名称和持久单元名称。**hibernate.cache.region_prefix** 默认为将 **jboss.as.jpa.scopedname** 设置为的任何内容。确保将 **jboss.as.jpa.scopedname** 值设置为同一应用服务器实例上部署的其他应用尚未使用的值。
jboss.as.jpa.deferdetach	控制非 Jakarta 事务线程中使用的事务范围持久性上下文是在每次实体 **管理器** 调用后分离加载的实体，还是在持久性上下文关闭时分离。默认值为 **false**。如果设为 **true**，则分离将延迟到上下文关闭为止。
wildfly.jpa.default-unit	设置为 **true**，以选择应用中的默认持久性单元。这可用于注入持久性上下文，但不指定 **unitName**，但在 **persistence.xml** 文件中指定多个持久性单元。
wildfly.jpa.twophasebootstrap	持久性供应商允许一个双阶段持久性单元 bootstrap，改进了 Jakarta Persistence 与 Jakarta Contexts 和 Dependency Injection 的集成。将 **wildfly.jpa.twophasebootstrap** 值设置为 **false**，可为包含该值的持久性单元禁用两阶段 bootstrap。
wildfly.jpa.allowdefaultdatasourceuse	设置为 **false**，以防止持久性单元使用默认数据源。默认值为 **true**。这只适用于没有指定数据源的持久性单元。
wildfly.jpa.hibernate.search.module	控制要包含在类路径中的 Hibernate Search 版本。默认值为 **auto** ；其他有效值为 **none**，或者是完整模块标识符来使用替代版本。

A.3. 策略提供程序属性

表 A.19. policy-provider Attributes

属性	描述
custom-policy	自定义策略提供程序定义。
jacc-policy	个策略提供程序定义，用于设置 Jakarta 授权和相关服务。

表 A.20. custom-policy Attributes

属性	描述
class-name	**java.security.Policy** 实施的名称，用于引用策略提供程序。

属性	描述
module	要从中加载提供程序的模块名称。

表 A.21. jacc-policy Attributes

属性	描述
policy	**java.security.Policy** 实施的名称，用于引用策略提供程序。
configuration-factory	**javax.security.jacc.PolicyConfigurationFactory** 实施的名称引用策略配置工厂提供程序。
module	要从中加载提供程序的模块名称。

A.4. JAKARTA EE 配置文件和技术参考

下表按类别列出了 Jakarta EE 技术，并注意它们是否包含在 Web Profile 或 Full Platform 配置文件中。

- Jakarta EE Web Application Technologies
- Jakarta EE Enterprise Application Technologies
- Jakarta EE Web Services Technologies
- Jakarta EE 管理和安全技术

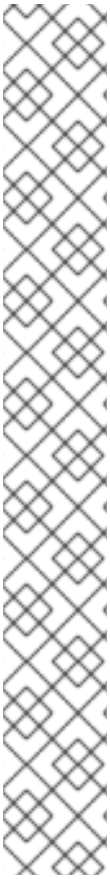
有关规格，请参阅 Jakarta EE 规范。

表 A.22. Jakarta EE Web Application Technologies

技术	Web 配置文件	完整平台
Jakarta WebSocket 1.1	✔	✔
jakarta JSON Binding 1.0	✔	✔

技术	Web 配置文件	完整平台
Jakarta JSON 处理 1.1	✔	✔
Jakarta Servlet 4.0	✔	✔
Jakarta Server Faces 2.3	✔	✔
Jakarta Expression Language 3.0	✔	✔
Jakarta 服务器页面 2.3	✔	✔
Jakarta 标准标签库 1.2 [1]	✔	✔

[1] 个额外的 Jakarta 标准标签库信息：



注意

JBoss EAP 中存在一个已知的安全风险，其中 Jakarta 标准标签库允许处理不受信任的 XML 文档中的外部实体引用，这些参考可以访问主机系统上的资源，并可能允许任意代码执行。

为避免这种情况，必须使用系统属性 org.apache.taglibs.standard.xml.accessExternalEntity 运行 JBoss EAP 服务器，通常具有空字符串作为值。这可以通过两种方式完成：

- **配置系统属性并重新启动服务器.**

```
org.apache.taglibs.standard.xml.accessExternalEntity
```

- **将 -Dorg.apache.taglibs.standard.xml.accessExternalEntity="" 作为参数传递给 standalone.sh 或 domain.sh 脚本。**

表 A.23. Jakarta EE Enterprise Application Technologies

技术	Web 配置文件	完整平台
Jakarta Batch 1.0		✔

技术	Web 配置文件	完整平台
jakarta Concurrency 1.0		✔
Jakarta 上下文和依赖注入 2.0	✔	✔
Jakarta 上下文和依赖注入 1.0	✔	✔
Jakarta Bean Validation 2.0	✔	✔
Jakarta Managed Beans 1.0	✔	✔
Jakarta Enterprise Beans 3.2		✔
Jakarta Interceptors 1.2	✔	✔
Jakarta Connectors 1.7		✔
Jakarta Persistence 2.2	✔	✔
Jakarta 注释 1.3		✔
雅加达消息传递 2.0		✔
Jakarta Transactions 1.2	✔	✔
雅加达邮件 1.6		✔

表 A.24. Jakarta EE Web Services Technologies

技术	Web 配置文件	完整平台
Jakarta RESTful Web Services 2.1		✔
jakarta 企业 Web 服务 1.3		✔
Java 平台 2.1 的 Web 服务元数据		✔
Jakarta XML RPC 1.1（可选）		
Jakarta XML 注册表 1.0（可选）		

表 A.25. Jakarta EE 管理和安全技术

技术	Web 配置文件	完整平台
Jakarta 安全 1.0	✔	✔
Jakarta 身份验证 1.1	✔	✔
Jakarta Authorization 1.5		✔
Jakarta Deployment 1.2（可选）		✔
Jakarta Management 1.1		✔
雅加达调试支持其他语言 1.0		✔

更新于 2024-02-09