Red Hat Fuse 7.13

Apache Karaf 事务指南

为 Apache Karaf 容器编写事务应用程序

Last Updated: 2024-06-12

Red Hat Fuse 7.13 Apache Karaf 事务指南

为 Apache Karaf 容器编写事务应用程序

法律通告

摘要

为 Fuse 开发事务感知应用程序

目录

前言

本指南提供有关实施 Fuse 事务应用程序的信息和说明。该信息组织如下：

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看我们的 CTO Chris Wright 信息。

第 1 章 事务简介

本章介绍了一些基本事务概念以及在事务管理器中重要的服务严重性。该信息组织如下：

1.1. 什么是事务？

事务建模是一个操作，其概念上包含一个步骤（例如，将资金从帐户 A 转移到帐户 B），但必须作为一系列步骤实施。此类操作容易受到系统故障的影响，因为失败可能会使一些步骤无法完成，这会使系统处于不一致的状态。例如，考虑将资金从 A 转移到帐户 B 的操作。假设系统在去除帐户 A 后失败，但在信用帐户 B 之前失败。因此，结果会消失。

为确保此操作可靠，请将其实施为 *事务*。事务保证了可靠的执行，因为它是原子、一致、隔离和持久化。这些属性称为事务的 ACID 属性。

1.2. 事务的 ACID 属性

事务的 *ACID* 属性定义如下：

- **Atomic**-a 事务是所有或无任何内容的步骤。当事务完成时，单独更新会被编译，并同时提交或中止（回滚）。
- **致**- 一个事务是将系统从一个一致状态移到另一个一致状态的工作单元。
- **isolated**- 在事务执行时，其部分结果会与其他实体隐藏。
- **durable**- 事务的结果在提交事务后马上失败。

1.3. 关于事务客户端

事务客户端 是一个 API 或对象，可让您启动和结束事务。通常，事务客户端会公开 **开始**、**提交** 或**回滚** 事务的操作。

在标准 hundreds 应用中，**javax.transaction.UserTransaction** 接口会公开事务客户端 API。在 Spring Framework 的上下文中，Spring Boot，**org.springframework.transaction.PlatformTransactionManager** 接口会公开一个事务客户端 API。

1.4. 事务术语的描述

下表定义了一些重要的事务术语：

术语	描述
demarcation	事务处理指的是启动和结束事务。结束事务意味着事务中完成的工作是提交或回滚。解译可以是显式的，例如，调用事务客户端 API 或隐式（例如，每当消息从事务端点轮询时）。详情请查看 第 9 章 *编写使用事务的 Camel 应用程序*。
Resources	*资源是* 可以处理持久或永久更改的计算机系统的任何组件。在实践中，资源几乎始终是在数据库中分层的数据库或服务，例如具有持久性的消息服务。然而，其他类型的资源是可有效的。例如，自动化 Teller 机器(ATM)是一种资源。客户从计算机物理接受 cash 后，无法撤销事务。
事务管理器	*事务管理器* 负责跨一个或多个资源协调事务。在很多情况下，事务管理器内置在一个资源中。例如，企业级数据库通常包含一个事务管理器，能够管理更改该数据库中内容的事务。涉及 **多个资源** 的事务通常需要 **一个外部** 事务管理器。
事务上下文	*事务上下文是* 一个对象，它封装了跟踪事务所需的信息。事务上下文的格式完全取决于相关的事务管理器实现。事务上下文至少包含唯一的事务标识符。
分布式事务	分布式事务指的是分布式系统中的事务，其中事务范围跨越多个网络节点。支持分布式事务的基本先决条件是支持以规范格式传输事务上下文的网络协议。**分布式事务不在 Apache Camel 事务范围内。**另请参阅：第 3.2.3 节 “关于分布式事务管理器”。
X/Open XA 标准	X/Open XA 标准描述了将资源与事务管理器集成的接口。要管理包含多个资源的事务，参与资源必须支持 XA 标准。支持 XA 标准的资源会公开一个特殊对象(*XA 交换机*)，它允许事务管理器（或事务处理监视器）控制资源事务。XA 标准支持 1 阶段提交协议和 2 阶段提交协议。

1.5. 管理修改多个资源的事务

对于涉及 **单个资源** 的事务，通常可以使用内置在资源中的事务管理器。对于 **涉及多个** 资源的事务，需要使用外部事务管理器或事务处理(TP)监控。在这种情况下，资源必须通过注册 XA 交换机来与事务管理器集成。

协议之间有一个重要的区别，用于提交在单个资源系统上运行的事务，以及用于提交在多个资源系统上操作的事务的协议：

- **1-phase commit**-is 用于单资源系统。此协议在单个步骤中提交事务。
- **2-phase commit**-is 用于多个资源系统。此协议在两个步骤中提交一个事务。

在一个事务中包含多个资源会带来系统故障的风险，在某些资源上提交事务，而不是所有资源。这会使系统处于不一致的状态。2 阶段提交协议旨在消除这一风险。它确保系统重启后 **始终** 可以恢复到一致的状态。

1.6. 事务和线程之间的关系

为了了解事务处理，非常感谢事务和线程之间的基本关系非常重要： **事务是特定于线程的。**也就是说，当事务启动时，它会附加到特定的线程。（从技术上，创建 *事务上下文* 对象并与当前线程相关联）。此

时，线程中的所有活动都会在这个事务范围内发生。其他线程中的活动 **不会** 属于这个事务的范围。但是，任何其他线程中的活动都可以在某些其他事务范围内。

这个事务和线程之间的关系意味着：

- 只要在单独的线程中创建每个 **事务，应用程序就可以同时处理多个** 事务。
- **注意在事务中创建子线程**。如果您在事务中，并且您创建了一个新的线程池，例如通过调用 **threads** （） Camel DSL 命令，新的线程 **不在** 原始事务范围内。
- **注意处理步骤，根据前面点上给出的相同原因，隐式创建新的线程**。
- **事务范围通常不会跨越路由段扩展**。也就是说，如果一个路由片段 **以(*JoinEndpoint*)** 结尾，另一个路由片段 **从(*JoinEndpoint*)** 开始，这些路由片段 **通常不** 属于相同的事务。然而，有一些例外。

注意

有些高级事务管理器实现可让您自由地将事务上下文与线程分离和附加。例如，这样可以将事务上下文从一个线程移到另一个线程。在某些情况下，也可以将单个事务上下文附加到多个线程。

1.7. 关于事务服务质量

在选择实施您的交易系统的产品时，提供各种数据库产品和交易管理器，有些免费收费和一些商业系统。所有这些都不支持事务处理，但这些产品支持的服务质量有显著的差异。本节提供了比较不同交易产品的可靠性和复杂程度时需要考虑的功能的简单指南。

1.7.1. 由资源提供的服务质量

以下特性决定了资源的服务质量：

- 第 1.7.1.1 节 “事务隔离级别”
- 第 1.7.1.2 节 “支持 XA 标准”

1.7.1.1. 事务**隔离**级别

ANSI SQL 定义四个 *事务隔离级别*，如下所示：

SERIALIZABLE

事务完全相互隔离。也就是说，在提交事务前，一个事务都不会影响任何其他事务。这个隔离级别被描述为可 **序列** 的，因为其效果如同在所有事务都执行后执行，但实际上，资源通常会优化算法，以便允许一些事务同时进行。

REPEATABLE_READ

每次事务读取或更新数据库时，都会获得读取或写入锁定，直到事务结束为止。这提供了几乎完美的隔离。但是，有一种情况是隔离并不完美。考虑使用 **WHERE** 子句读取行范围的 SQL **SELECT** 语句。如果在第一个事务运行时，另一个事务向这个范围添加一个行，则第一个事务可以查看这个新行（如果它重复 **SELECT** 调用(*读取*)。

READ_COMMITTED

在事务结束前，写入锁定会被保存。在事务结束前，读取锁定 **不会被** 保存。因此，重复读取可能会提供不同的结果，因为其他事务所提交的更新对持续事务可见。

READ_UNCOMMITTED

在事务结束前，读取锁定或写锁都不会被保存。因此，脏读取是可能的。脏就绪是，当其他事务中未提交的更改对持续的事务可见。

数据库通常不支持所有不同的事务隔离级别。例如，一些可用的数据库只支持 **READ_UNCOMMITTED**。另外，一些数据库以与 ANSI 标准不同的方式实施事务隔离级别。隔离是一个复杂的问题，它涉及使用数据库性能权衡（例如，请参见 Wikipedia 中的隔离）。

1.7.1.2. 支持 XA 标准

要使资源参与涉及多个资源的事务，它需要支持 X/Open XA 标准。务必检查资源的 XA 标准的实现是否受到任何特殊限制的影响。例如，XA 标准的一些实现仅限于单个数据库连接，这意味着一次只有一个线程可以处理涉及该资源的事务。

1.7.2. 事务管理器提供的服务质量

以下功能决定了事务管理器的服务质量：

- 第 1.7.2.1 节 “支持 suspend/resume 和 attach/detach”.
- 第 1.7.2.2 节 “支持多个资源”.
- 第 1.7.2.3 节 “分布式事务”.
- 第 1.7.2.4 节 “事务监控”.
- 第 1.7.2.5 节 “从失败中恢复”.

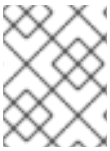
1.7.2.1. 支持 suspend/resume 和 attach/detach

有些事务管理器支持操作事务上下文和应用程序线程之间关联的高级功能，如下所示：

- suspend **/resume current transaction**- 可让您临时暂停当前事务上下文，而应用在当前线程中执行一些非事务工作。
- **attach/detach transaction context**- 可让您将事务上下文从一个线程移到另一个线程，或者扩展事务范围使其包含多个线程。

1.7.2.2. 支持多个资源

事务管理器的主要区别是能够支持多个资源。这通常支持 XA 标准，其中事务管理器为资源提供了注册其 XA 交换机的方法。



注意

严格说，XA 标准不是您可以用来支持多个资源的唯一方法，但它是最实际的一个。替代方案通常涉及编写繁琐（和关键）自定义代码，以实施通常由 XA 交换机提供的算法。

1.7.2.3. 分布式事务

有些事务管理器能够管理范围在分布式系统中包含多个节点的事务。通过使用特殊的协议（如 WS-AtomicTransactions 或 CORBA OTS）将事务上下文从节点传播到节点。

1.7.2.4. 事务监控

高级事务管理器通常提供可视化工具来监控待处理事务的状态。此类工具在系统失败后特别有用，它可以帮助识别和解决处于不确定状态（硬例外）的事务。

1.7.2.5. 从失败中恢复

在出现系统故障(crash)时，事务管理器与其稳健性相关的显著区别。事务管理器使用的关键策略是将数据写入持久日志，然后执行事务的每个步骤。如果出现故障，日志中的数据可用于恢复事务。一些交易经理会比其他人更仔细地实施此策略。例如，高端事务管理器通常会复制持久性事务日志，并允许每个日志存储在单独的主机机器上。

第 2 章 开始使用 KARAF 上的事务(OSGI)

本节介绍使用事务访问 Artemis JMS 代理的 Camel 应用程序。该信息组织如下：

- 第 2.1 节 “先决条件”
- 第 2.2 节 “构建 camel-jms 项目”
- 第 2.3 节 “camel-jms 项目的解释”

2.1. 先决条件

此 Camel 应用程序的实现需要满足以下先决条件：

- 外部 AMQ 7 JMS 消息代理必须正在运行。
  以下示例代码运行 **amq-broker-7.1.0-bin.zip** 的独立（非 Docker）版本。执行创建并运行 **amq7** 实例：

```
$ pwd
/data/servers/amq-broker-7.1.0

$ bin/artemis create --user admin --password admin --require-login amq7
Creating ActiveMQ Artemis instance at: /data/servers/amq-broker-7.1.0/amq7

Auto tuning journal ...
done! Your system can make 27.78 writes per millisecond, your journal-buffer-timeout will be 36000

You can now start the broker by executing:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis" run

Or you can run the broker in the background using:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis-service" start

$ amq7/bin/artemis run

        __  __  ____    ____          _
      /\   |  \/  |/ __ \  |  _ \        | |
     /  \  | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
    / /\ \ | |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
   / ____ \| |  | | |__| | | |_) | | | (_) |   <  __/ |
  /_/    \_\_|  |_|\____\_\ |____/|_|  \___/|_|\_\___|_|

 Red Hat JBoss AMQ 7.1.0.GA


018-05-02 16:37:19,294 INFO  [org.apache.activemq.artemis.integration.bootstrap] AMQ101000: Starting ActiveMQ Artemis Server
...
```

- 客户端库是必需的。Artemis 库在 Maven Central 或红帽存储库中提供。例如，您可以使用：
  - **mvn:org.apache.activemq/artemis-core-client/2.4.0.amq-710008-redhat-1**

- **mvn:org.apache.activemq/artemis-jms-client/2.4.0.amq-710008-redhat-1**

或者，Artemis/AMQ 7 客户端库也可以作为 Karaf 功能安装，例如：

- **karaf@root()> feature:install artemis-jms-client artemis-core-client**

- 需要一些提供 Karaf shell 命令或专用 Artemis 支持的功能：

```
karaf@root()> feature:install jms pax-jms-artemis pax-jms-config
```

- 所需的 Camel 功能包括：

```
karaf@root()> feature:install camel-jms camel-blueprint
```

2.2. 构建 CAMEL-JMS 项目

您可以从 Fuse Software Downloads 页面下载 **快速入门**。

将 zip 文件的内容提取到本地文件夹中，例如一个名为 **quickstarts** 的新文件夹。

然后，您可以构建并安装 **/camel/camel-jms** 示例作为 OSGi 捆绑包。此捆绑包包含 Camel 路由的蓝图 XML 定义，用于将消息发送到 AMQ 7 JMS 队列。

在以下示例中，**$FUSE_HOME** 是解压缩 Fuse 分发的位置。构建此项目：

1. 调用 Maven 以构建项目：

```
$ cd quickstarts

$ mvn clean install -f camel/camel-jms/
```

2. 创建 JMS 连接工厂配置，使得 **javax.jms.ConnectionFactory** 服务在 OSGi 运行时中发布。为此，请将 **quickstarts/camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg** 复制到 **$FUSE_HOME/etc** 目录中。将处理此配置来创建正常工作的连接工厂。例如：

```
$ cp camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg ../etc/
```

3. 验证发布的连接工厂：

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
----------------------------
 felix.fileinstall.filename = file:$FUSE_HOME/etc/org.ops4j.connectionfactory-amq7.cfg
 name = artemis
 osgi.jndi.service.name = artemis
 password = admin
 pax.jms.managed = true
 service.bundleid = 251
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 436
 service.pid = org.ops4j.connectionfactory.d6207fcc-3fe6-4dc1-a0d8-0e76ba3b89bf
 service.scope = singleton
 type = artemis
```

```
 url = tcp://localhost:61616
 user = admin
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:info -u admin -p admin artemis
Property │ Value
─────────┼──────────────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
──────────────────────────────────────────────────────
df2501d1-aa52-4439-b9e4-c0840c568df1
DLQ
ExpiryQueue
```

4. 安装捆绑包：

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/7.0.0.redhat-SNAPSHOT
Bundle ID: 256
```

5. 确认它正在正常工作：

```
karaf@root()> camel:context-list
 Context             Status           Total #       Failed #     Inflight #   Uptime
 -------             ------           -------       --------     ----------   ------
 jms-example-context   Started              0              0              0   2 minutes
karaf@root()> camel:route-list
 Context             Route              Status            Total #       Failed #     Inflight #   Uptime
 -------             -----              ------            -------       --------     ----------   ------
 jms-example-context   file-to-jms-route   Started              0              0              0   2 minutes
 jms-example-context   jms-cbr-route       Started              0              0              0   2 minutes
```

6. Camel 路由启动后，您可以在 Fuse 安装中看到目录 **work/jms/input**。将您在这个 quickstart 的 **src/main/data** 目录中找到的文件复制到新创建的 **work/jms/input** 目录中。

7. 稍等片刻，您将在 **work/jms/output** 目录下找到国家按国家组织相同的文件：

   - **order1.xml**,**order2.xml** 和 **order4.xml** in **work/jms/output/others**
   - **work/jms/output/us**中的 **order3.xml** 和 **order5.xml**
   - **order6.xml** in **work/jms/output/fr**

8. 查看日志以查看业务日志记录：

```
2018-05-02 17:20:47,952 | INFO  | ile://work/jms/input | file-to-jms-route              | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Receiving order order1.xml
2018-05-02 17:20:48,052 | INFO  | umer[incomingOrders] | jms-cbr-route                  | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Sending order order1.xml to another
country
2018-05-02 17:20:48,053 | INFO  | umer[incomingOrders] | jms-cbr-route                  | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Done processing order1.xml
```

9. 查看队列是动态创建的：

```
karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
————————————————————————————————————————
DLQ
17767323-937f-4bad-a403-07cd63311f4e
ExpiryQueue
incomingOrders
```

10. 检查 Camel 路由统计信息：

```
karaf@root()> camel:route-info jms-example-context file-to-jms-route
Camel Route file-to-jms-route
 Camel Context: jms-example-context
 State: Started
 State: Started


Statistics
 Exchanges Total: 1
 Exchanges Completed: 1
 Exchanges Failed: 0
 Exchanges Inflight: 0
 Min Processing Time: 67 ms
 Max Processing Time: 67 ms
 Mean Processing Time: 67 ms
 Total Processing Time: 67 ms
 Last Processing Time: 67 ms
 Delta Processing Time: 67 ms
 Start Statistics Date: 2018-05-02 17:14:17
 Reset Statistics Date: 2018-05-02 17:14:17
 First Exchange Date: 2018-05-02 17:20:48
 Last Exchange Date: 2018-05-02 17:20:48
```

2.3. CAMEL-JMS 项目的解释

Camel 路由使用以下端点 URI：

```
<route id="file-to-jms-route">
...
    <to uri="jms:queue:incomingOrders?transacted=true" />
</route>

<route id="jms-cbr-route">
    <from uri="jms:queue:incomingOrders?transacted=true" />
...
</route>
```

jms 组件使用此片断进行配置：

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <reference interface="javax.jms.ConnectionFactory" />
```

```
    </property>
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

在 **transactionManager** 引用时：

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

正如您所见，JMS 连接工厂和 **PlatformTransactionManager** 的 Spring 接口仅引用。不需要在蓝图 XML *中定义* 它们。*这些服务* 由 Fuse 本身公开。

您已看到 **javax.jms.ConnectionFactory** 是使用 **etc/org.ops4j.connectionfactory-amq7.cfg** 创建的。

事务管理器是：

```
karaf@root()> service:list org.springframework.transaction.PlatformTransactionManager
[org.springframework.transaction.PlatformTransactionManager]
----------------------------------------------------------
 service.bundleid = 21
 service.id = 527
 service.scope = singleton
Provided by :
 Red Hat Fuse :: Fuse Modules :: Transaction (21)
Used by:
 Red Hat Fuse :: Quickstarts :: camel-jms (256)
```

检查注册实际事务管理器的其他接口：

```
karaf@root()> headers 21

Red Hat Fuse :: Fuse Modules :: Transaction (21)
------------------------------------------------
...
Bundle-Name = Red Hat Fuse :: Fuse Modules :: Transaction
Bundle-SymbolicName = fuse-pax-transx-tm-narayana
Bundle-Vendor = Red Hat
...

karaf@root()> bundle:services -p 21

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
objectClass = [org.osgi.service.cm.ManagedService]
service.bundleid = 21
service.id = 519
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
 ----
objectClass = [javax.transaction.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 520
service.scope = singleton
 ----
```

```
objectClass = [javax.transaction.TransactionSynchronizationRegistry]
provider = narayana
service.bundleid = 21
service.id = 523
service.scope = singleton
 ----
objectClass = [javax.transaction.UserTransaction]
provider = narayana
service.bundleid = 21
service.id = 524
service.scope = singleton
 ----
objectClass = [org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
provider = narayana
service.bundleid = 21
service.id = 525
service.scope = singleton
 ----
objectClass = [org.ops4j.pax.transx.tm.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 526
service.scope = singleton
 ----
objectClass = [org.springframework.transaction.PlatformTransactionManager]
service.bundleid = 21
service.id = 527
service.scope = singleton
```

事务管理器可从这些接口获得：

- **javax.transaction.TransactionManager**
- **javax.transaction.TransactionSynchronizationRegistry**
- **javax.transaction.UserTransaction**
- **org.jboss.narayana.osgi.jta.ObjStoreBrowserService**
- **org.ops4j.pax.transx.tm.TransactionManager**
- **org.springframework.transaction.PlatformTransactionManager**

您可以在您需要的任何上下文中使用它们。例如，**camel-jms** 需要初始化 **org.apache.camel.component.jms.JmsConfiguration.transactionManager** 字段。这就是示例使用的原因：

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

例如：

```
<reference id="transactionManager" interface="javax.transaction.TransactionManager" />
```

第 3 章 用于配置和引用事务管理器的接口

JSR 和 Spring Boot 各自提供一个事务客户端接口，用于在 Fuse 中配置事务管理器，并在已部署的应用中使用事务管理器。配置之间有明确的区别，即管理任务和引用，这是开发任务。应用程序开发人员负责将应用指向之前配置的事务管理器。

3.1. 事务管理器的作用

事务管理器是应用程序的一部分，负责跨一个或多个资源协调事务。事务管理器的职责如下：

- demarcation - 使用 begin、commit 和 rollback 方法开始和结束事务。
- 管理事务上下文 - 事务上下文包含事务管理器跟踪事务所需的信息。事务管理器负责创建事务上下文并将其附加到当前线程中。
- 协调多个资源之间的事务 - 企业级的事务通常能够跨多个资源协调事务。此功能要求使用 XA 协议注册和管理 2 阶段提交协议和资源。请参阅 第 1.7.1.2 节 “支持 XA 标准”。
  这是一个高级功能，不受到所有事务管理器的支持。
- 从失败中恢复 - 事务管理器负责确保在系统故障且应用失败时不会处于不一致状态。在某些情况下，可能需要人工干预才能将系统恢复到一致的状态。

3.2. 关于本地、全局和分布式事务管理器

事务管理器可以是 local、global 或 distributed。

3.2.1. 关于本地事务管理器

本地事务管理器 是一个事务管理器，只能协调单个资源的事务。本地事务管理器的实施通常嵌入到资源本身中，应用程序使用的事务管理器是围绕此内置事务管理器的精简包装器。

例如，Oracle 数据库有一个内置事务管理器，它支持 decation 操作（使用 SQL **BEGIN**、**COMMIT**、or **zFCPLBACK** 语句，或使用原生 Oracle API）以及各种级别的事务隔离。可以通过 JDBC 导出对 Oracle 事务管理器的控制，应用程序使用此 JDBC API 来划分事务。

在此上下文中，了解构成资源的内容非常重要。例如，如果您使用 JMS 产品，JMS 资源是 JMS 产品的单个正在运行的实例，而不是单个队列和主题。此外，如果以不同方式访问相同的底层资源，有时似乎是多个资源可能实际上是一个资源。例如，您的应用可能会直接访问关系数据库（通过 JDBC）和间接访问（通过对象关系映射工具，如 Hibernate）。在这种情况下，涉及相同的底层事务管理器，因此应可以在相同的事务中注册这两个代码片段。

注意

无法保证每个情况下都可以正常工作。虽然原则上可能，但设计 Spring Framework 或其他打包程序层的一些细节可能会阻止它在实践中工作。

应用可能具有许多不同的本地事务管理器相互独立工作。例如，您可以有一个 Camel 路由来处理 JMS 队列和主题，其中 JMS 端点引用 JMS 事务管理器。另一个路由可以通过 JDBC 访问关系数据库。但是您不能在同一路由中组合 JDBC 和 JMS 访问，并且它们都参与同一事务。

3.2.2. 关于全局事务管理器

全局事务管理器是一个事务管理器，它可以协调多个资源上的事务。当您无法依赖内置于资源本身的事务管理器时，这是必需的。外部系统有时被称为事务处理监控器(TP monitor)，能够跨不同资源协调事务。

以下是在多个资源上运行的事务的先决条件：

- 全局事务管理器或 TP monitor - 实现 2 阶段提交协议的外部事务系统，用于协调多个 XA 资源。
- 支持 *XA 标准* 的资源 - 要参与 2 阶段提交，资源必须支持 XA 标准。请参阅 第 1.7.1.2 节 “支持 XA 标准”。实际上，这意味着资源能够导出 XA *交换机* 对象，它可以完全控制到外部 TP 监控器的事务。

提示

Spring Framework 本身不提供 TP 监控器来管理全局事务。但是，它确实支持与 OSGi 提供的 TP monitor 集成，或与 gRPC 提供的 TP monitor 集成（其中集成由 JtaTransactionManager 类实施）。因此，如果您将应用程序部署到带有完整事务支持的 OSGi 容器中，您可以在 Spring 中使用多个事务资源。

3.2.3. 关于分布式事务管理器

通常，服务器直接连接到事务中涉及的资源。但是，在分布式系统中，偶尔需要通过 Web 服务连接到仅间接公开的资源。在这种情况下，您需要一个能够支持分布式事务的 TP 监控器。有几个标准可用于描述如何支持各种分布式协议的事务，例如，适用于 Web 服务的 WS-AtomicTransactions 规格。

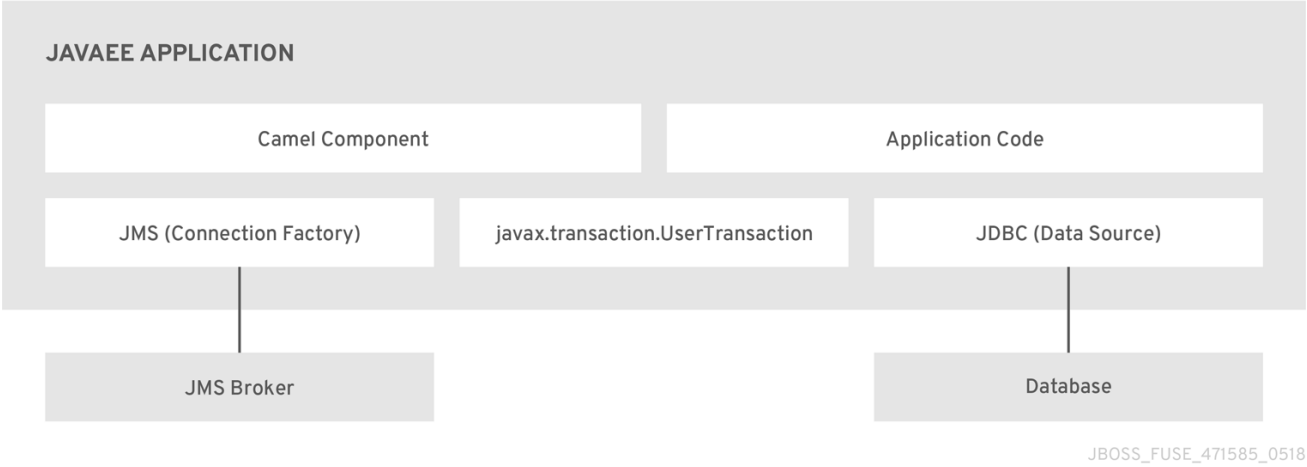
3.3. 使用>=< 事务客户端

使用 802-1 时，与事务管理器交互的最资金和标准方法是 Java Transaction API (JTA)接口 **javax.transaction.UserTransaction**。规范用法是：

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
ut.begin();

// Access transactional, JTA-aware resources such as database and/or message broker

ut.commit(); // or ut.rollback()
```

从 JNDI (Java 命名和目录接口)获取用户 **事务** 实例是获取事务客户端的一种方式。在 interaction 环境中，您可以访问事务客户端，例如使用 CDI （上下文和依赖项注入）。

下图显示了 typica developer Camel 应用。



图显示 Camel 代码和应用程序代码都可以访问：

- **javax.transaction.UserTransaction** 实例，可以使用 Spring **TransactionTemplate** 类直接从应用程序或内部处理事务处理。
- 数据库通过 JDBC API 直接或使用 Spring 的 **JdbcTemplate**，或者使用 **camel-jdbc** 组件。
- 使用 Spring 的 **JmsTemplate** 类或使用 **camel-jms** 组件直接通过 JMS API 进行消息代理。

使用 **javax.transaction.UserTransaction** 对象时，您不需要了解正在使用的实际事务管理器，因为您只直接使用事务客户端。（请参阅 第 1.3 节 “关于事务客户端”。）Spring 和 Camel 采用不同的方法，因为它在内部使用 Spring 的事务设施。

iwl Application

在典型的>=< 情景中，应用被部署到 ausearch 应用服务器，通常作为 **WAR** 或 **EAR** 存档。通过 JNDI 或 CDI，应用程序可以访问 **javax.transaction.UserTransaction** 服务的实例。然后，使用这个事务客户端实例来划分事务。在事务中，应用执行 JDBC 和/或 JMS 访问。

Camel 组件和应用程序代码

它们代表了执行 JMS/JDBC 操作的代码。Camel 具有自己的高级方法来访问 JMS/JDBC 资源。应用程序代码可以直接使用给定的 API。

JMS Connection Factory

这是 **javax.jms.ConnectionFactory** 接口，用于获取 **javax.jms.Connection** 的实例，然后是 **javax.jms.Session**（或 JMS 2.0 中的 **javax.jms.JmsContext**）。这可以由应用程序直接使用，或者在 Camel 组件中间接使用，这些组件可以在内部使用 **org.springframework.jms.core.JmsTemplate**。应用程序代码和 Camel 组件都不需要此连接工厂的详细信息。连接工厂在应用服务器中配置。您可以在 regex 服务器中看到此配置。Fuse 等 OSGi 服务器类似。系统管理员独立于应用程序配置连接工厂。通常，连接工厂实现池功能。

JDBC 数据源

这是 **javax.sql.DataSource** 接口，用于获取 **java.sql.Connection** 的实例。与 JMS 一样，此数据源可以直接或间接使用。例如，**camel-sql** 组件在内部使用 **org.springframework.jdbc.core.JdbcTemplate** 类。与 JMS 一样，应用程序代码和 Camel 都不需要此数据源的详细信息。配置是在应用服务器内或 OSGi 服务器内完成的，方法是使用 第 4 章 *配置 Narayana 事务管理器* 中描述的方法。

3.4. 使用 SPRING BOOT 事务客户端

Spring Framework （和 Spring Boot）的一个主要目标是使 QPC API 更易于使用。所有主要的 *vanilla* API 将其部分在 Spring 框架(Spring Boot)中。这些不是给定 API **的替代方案** 或替换，而是添加更多配置选项或更一致的用法的打包程序，例如处理异常。

下表与 Spring 相关的接口匹配给定了>=< API：

iwl API	Spring utility	配置为
JDBC	**org.springframework.jdbc.core.JdbcTemplate**	**javax.sql.DataSource**
JMS	**org.springframework.jms.core.JmsTemplate**	**javax.jms.ConnectionFactory**
JTA	**org.springframework.transaction.support.TransactionTemplate**	**org.springframework.transaction.PlatformTransactionManager**

JdbcTemplate 和 JmsTemplate 分别直接使用 javax.sql.DataSource 和 javax.jms.ConnectionFactory。但是 TransactionTemplate 使用 PlatformTransactionManager 的 Spring 接口。在这里，Spring 并不只是 改进 QPC，而是将 binutils 客户端 API 替换为自己的 API。

Spring 将 javax.transaction.UserTransaction 视为接口，对于真实情况来说非常简单。另外，因为 javax.transaction.UserTransaction 没有区分本地、单一资源事务和全局多资源事务，因此 org.springframework.transaction.PlatformTransactionManager 的实现使开发人员更自由。

以下是 Spring Boot 的规范 API 用法：

```
// Create or get from ApplicationContext or injected with @Inject/@Autowired.
JmsTemplate jms = new JmsTemplate(...);
JdbcTemplate jdbc = new JdbcTemplate(...);
TransactionTemplate tx = new TransactionTemplate(...);

tx.execute((status) -> {
   // Perform JMS operations within transaction.
   jms.execute((SessionCallback<Object>)(session) -> {
      // Perform operations on JMS session
      return ...;
   });
   // Perform JDBC operations within transaction.
   jdbc.execute((ConnectionCallback<Object>)(connection) -> {
      // Perform operations on JDBC connection.
      return ...;
   });
   return ...;
});
```

在上例中，所有三种 *模板* 都仅实例化，但它们也可以从 Spring 的 ApplicationContext 获取，也可以使用 @Autowired 注解注入。

3.4.1. 使用 Spring PlatformTransactionManager 接口

如前文所述，javax.transaction.UserTransaction 通常从 QPC 应用中的 JNDI 获取。但是，Spring 为许多场景提供明确的这个接口实现。您不需要完整的 JTA 场景，有时应用程序只需要访问单个资源，如 JDBC。

通常，org.springframework.transaction.PlatformTransactionManager 是 Spring 事务客户端 API，它提供经典事务客户端操作：begin、commit 和 rollback。换句话说，这个接口提供了在运行时控制事务的基本方法。

注意

任何事务系统的其他关键方面都是实施事务资源的 API。但是，事务资源通常由底层数据库实施，因此该事务编程方面很少会考虑应用程序。

3.4.1.1. PlatformTransactionManager 接口的定义

```
public interface PlatformTransactionManager {

   TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException;

   void commit(TransactionStatus status) throws TransactionException;

   void rollback(TransactionStatus status) throws TransactionException;
}
```

3.4.1.2. 关于 TransactionDefinition 接口

您可以使用 TransactionDefinition 接口指定新创建的事务的特征。您可以指定隔离级别和新事务的传播策略。详情请查看 第 9.4 节 “事务传播策略”。

3.4.1.3. TransactionStatus 接口的定义

您可以使用 TransactionStatus 接口来检查当前事务的状态，即与当前线程关联的事务，并标记当前的事务进行回滚。这是接口定义：

```
public interface TransactionStatus extends SavepointManager, Flushable {

   boolean isNewTransaction();

   boolean hasSavepoint();

   void setRollbackOnly();

   boolean isRollbackOnly();

   void flush();

   boolean isCompleted();
}
```

3.4.1.4. 平台TransactionManager 接口定义的方法

PlatformTransactionManager 接口定义以下方法：

getTransaction()

创建新的事务，并通过传递定义新事务的特征的 TransactionDefinition 对象来将其与当前的线程相关联。这与许多其他事务客户端 API 的 begin （） 方法类似。

commit()

提交当前事务，从而对已注册资源进行所有待处理的更改。

rollback()

回滚当前事务，从而撤销对注册资源的所有待处理更改。

3.4.2. 使用事务管理器的步骤

通常，您不直接使用 PlatformTransactionManager 接口。在 Apache Camel 中，您通常使用事务管理器，如下所示：

1. 创建事务管理器的实例。Spring 中有几个不同的实现，请参阅 第 3.4 节 “使用 Spring Boot 事务客户端”。
2. 将事务管理器实例传递到 Apache Camel 组件或路由中的 transacted （） DSL 命令。然后，事务组件或 transacted （） 命令负责处理事务。详情请查看 第 9 章 *编写使用事务的 Camel 应用程序*。

3.4.3. 关于 Spring PlatformTransactionManager 实现

本节概述了 Spring Framework 提供的事务管理器实施。这些实施分为两类：本地事务管理器和全局事务管理器。

从 Camel 开始：

- camel-jms 组件使用的 org.apache.camel.component.JmsConfiguration 对象需要一个 org.springframework.transaction.PlatformTransactionManager 接口的实例。
- org.apache.camel.component.sql.SqlComponent 在内部使用 org.springframework.jdbc.core.JdbcTemplate 类，并且此 JDBC 模板也与 org.springframework.transaction.PlatformTransactionManager 集成。

正如您所见，*必须有 这个接口的一些* 实现。根据具体情况，您可以配置所需的平台事务管理器。

3.4.3.1. 本地平台TransactionManager 实现

以下列表总结了 Spring Framework 提供的本地事务管理器实施。这些事务管理器只支持一个资源。

org.springframework.jms.connection.JmsTransactionManager

此事务管理器实施能够管理 *单个* JMS 资源。您可以连接到任意数量的队列或主题，但只有在它们属于同一底层 JMS 消息传递产品实例时。此外，您无法在事务中列出任何其他类型的资源。

org.springframework.jdbc.datasource.DataSourceTransactionManager

此事务管理器实施能够管理 *单个* JDBC 数据库资源。您可以更新任意数量的不同的数据库表，*但仅在* 它们属于同一底层数据库实例时。

org.springframework.orm.jpa.JpaTransactionManager

此事务管理器实施能够管理 Java Persistence API (jpa)资源。但是，无法同时列出事务中任何其他种类的资源。

org.springframework.orm.hibernate5.HibernateTransactionManager

此事务管理器实施能够管理 Hibernate 资源。但是，无法同时列出事务中任何其他种类的资源。此外，比原生 Hibernate API 首选 JPA API。

另外还有其他常用的，平台TransactionManager 的实现。

3.4.3.2. 全局 PlatformTransactionManager 实现

Spring Framework 提供了一个全局事务管理器实现，供 OSGi 运行时使用。org.springframework.transaction.jta.JtaTransactionManager 支持事务中的多个资源的操作。此事务管理器支持 XA 事务 API，并可在事务中列出多个资源。要使用此事务管理器，您必须将应用程序部署到 OSGi 容器或 iwl 服务器。

虽然 PlatformTransactionManager 的单资源实现是实际 *实现*，但JtaTransactionManager 更是标准 javax.transaction.TransactionManager 的实际实现的打包程序。

这就是为什么最好在一个环境中使用平台事务管理器的 Jta TransactionManager 实现（通过 JNDI 或 CDI）一个已经配置的 javax.transaction.TransactionManager 实例，通常是 javax.transaction.UserTransaction。通常，这两个 JTA 接口都通过单个对象/服务来实施。

以下是配置/使用 JtaTransactionManager 的示例：

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
TransactionManager tm = (TransactionManager) context.lookup("java:/TransactionManager");

JtaTransactionManager jta = new JtaTransactionManager();
jta.setUserTransaction(ut);
jta.setTransactionManager(tm);

TransactionTemplate jtaTx = new TransactionTemplate(jta);

jtaTx.execute((status) -> {
   // Perform resource access in the context of global transaction.
   return ...;
});
```

在上例中，实际为 JTA 对象(UserTransaction 和 TransactionManager)的实例是从 JNDI 获取的。在 OSGi 中，它们也可以从 OSGi 服务注册表获取。

3.5. 事务客户端和事务管理器之间的 OSGI 接口

在 description of the client API 和 Spring Boot 事务客户端 API 的描述后，查看 OSGi 服务器（如 Fuse）中的关系会很有帮助。OSGi 的一个功能是全局服务 registry，可用于：

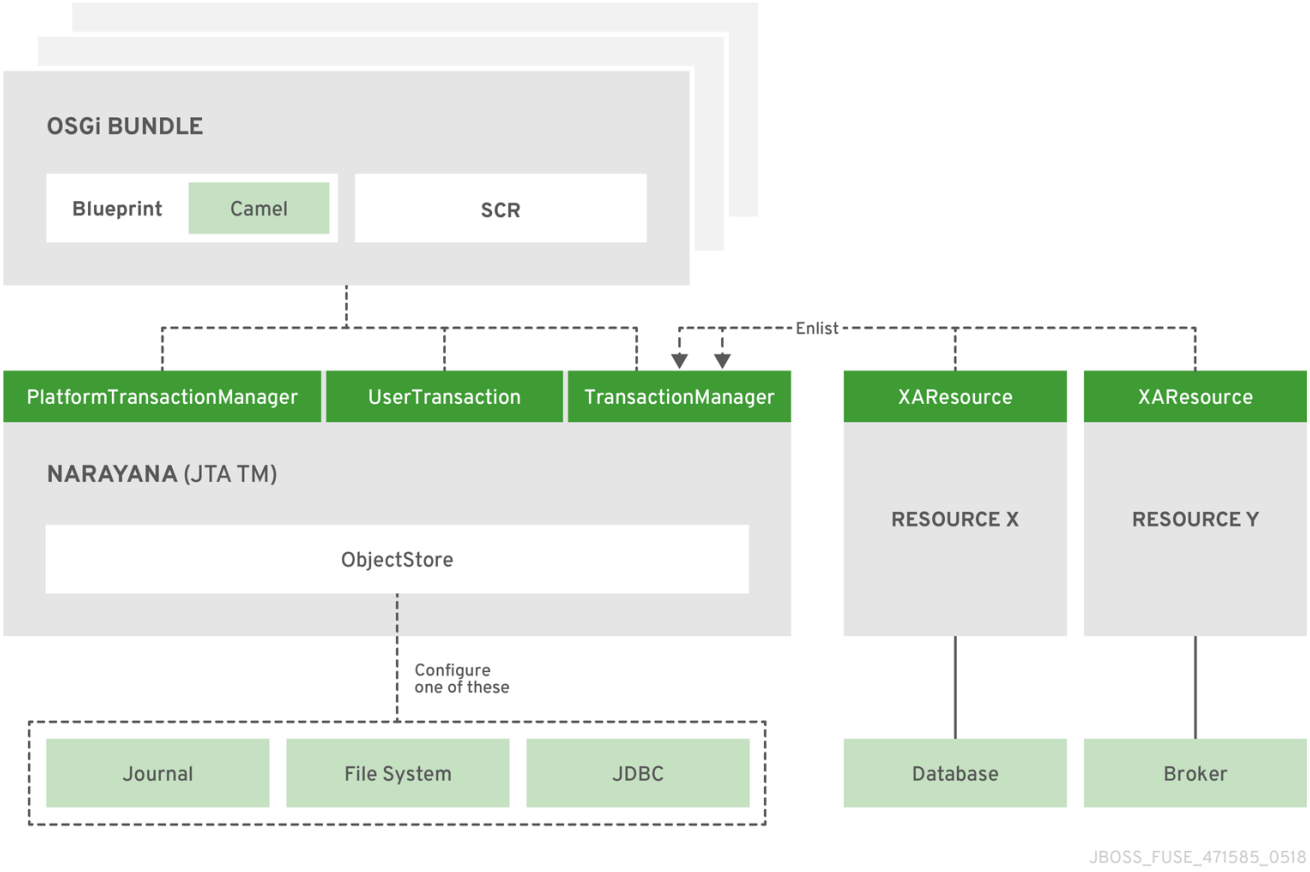
- 通过过滤或接口查找服务。
- 使用给定接口和属性注册服务。

同样，在>=< 应用程序服务器中部署的应用程序使用 JNDI (*服务 locator* 方法)获取对 javax.transaction.UserTransaction 的引用，或者使它们被 CDI*注入*（*依赖项注入* 方法）注入，您可以使用以下方法获取相同的引用（直接或间接）：

- 调用 org.osgi.framework.BundleContext.getServiceReference （） 方法(*服务 locator*)。
- 将它们注入到 Blueprint 容器中。

- 使用 Service 组件运行时(SCR)注解(*依赖项注入*)。

下图显示了在 OSGi 运行时中部署的 Fuse 应用程序。应用程序代码和/或 Camel 组件使用其 API 获取事务管理器、数据源和连接工厂的引用。



应用程序（捆绑包）与 OSGi 注册表中注册的服务交互。通过 *接口执行访问权限*，这与应用程序应相关。

在 Fuse 中，实施（直接或通过小包装器）事务的客户端接口的基本对象是 org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager。您可以使用以下接口访问事务管理器：

- javax.transaction.TransactionManager
- javax.transaction.UserTransaction
- org.springframework.transaction.PlatformTransactionManager
- org.ops4j.pax.transx.tm.TransactionManager

您可以直接使用这些接口，或通过选择框架或库（如 Camel）来隐式使用它们。

有关在 Fuse 中配置 org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager 的方法的详情，请参考 第 4 章 *配置 Narayana 事务管理器*。本指南中的后续章节将构建本章中的信息，并描述了如何配置和使用其他服务，如 JDBC 数据源和 JMS 连接工厂。

第 4 章 配置 NARAYANA 事务管理器

在 Fuse 中，内置的全局事务管理器是 JBoss Narayana Transaction Manager，后者是企业应用平台 (EAP) 7 使用的同一事务管理器。

在 OSGi 运行时中，与用于 Karaf 的 Fuse 一样，额外的集成层由 PAX TRANSX 项目提供。

以下主题讨论 Narayana 配置：

- 第 4.1 节 “关于 Narayana 安装”
- 第 4.2 节 “支持的事务协议”
- 第 4.3 节 “关于 Narayana 配置”
- 第 4.4 节 “配置日志存储”

4.1. 关于 NARAYANA 安装

Narayana 事务管理器在以下接口下的 OSGi 捆绑包中公开使用，以及一些额外的支持接口：

- **javax.transaction.TransactionManager**
- **javax.transaction.UserTransaction**
- **org.springframework.transaction.PlatformTransactionManager**
- **org.ops4j.pax.transx.tm.TransactionManager**

7.13.0.fuse-7_13_0-00012-redhat-00001 分发使这些接口在启动时可用。

pax-transx-tm-narayana 功能包含一个覆盖捆绑包，用于嵌入 Narayana：

```
karaf@root()> feature:info pax-transx-tm-narayana
Feature pax-transx-tm-narayana 0.3.0
Feature has no configuration
Feature has no configuration files
Feature depends on:
  pax-transx-tm-api 0.0.0
Feature contains followed bundles:
  mvn:org.jboss.fuse.modules/fuse-pax-transx-tm-narayana/7.0.0.fuse-000191-redhat-1 (overriden
from mvn:org.ops4j.pax.transx/pax-transx-tm-narayana/0.3.0)
Feature has no conditionals.
```

fuse-pax-transx-tm-narayana 捆绑包提供的服务有：

```
karaf@root()> bundle:services fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
[org.osgi.service.cm.ManagedService]
[javax.transaction.TransactionManager]
[javax.transaction.TransactionSynchronizationRegistry]
[javax.transaction.UserTransaction]
```

```
[org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
[org.ops4j.pax.transx.tm.TransactionManager]
[org.springframework.transaction.PlatformTransactionManager]
```

由于此捆绑包注册 org.osgi.service.cm.ManagedService，所以它会跟踪并响应 CM 配置中的更改：

```
karaf@root()> bundle:services -p fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
objectClass = [org.osgi.service.cm.ManagedService]
service.bundleid = 21
service.id = 232
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
...
```

默认的 org.ops4j.pax.transx.tm.narayana PID 为：

```
karaf@root()> config:list '(service.pid=org.ops4j.pax.transx.tm.narayana)'
----------------------------------------------------------------
Pid:            org.ops4j.pax.transx.tm.narayana
BundleLocation: ?
Properties:
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
/data/servers/7.13.0.fuse-7_13_0-00012-redhat-00001/data/narayana

com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
/data/servers/7.13.0.fuse-7_13_0-00012-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
/data/servers/7.13.0.fuse-7_13_0-00012-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean.recoveryBackoffPeriod = 10
   felix.fileinstall.filename = file:/data/servers/7.13.0.fuse-7_13_0-00012-redhat-
00001/etc/org.ops4j.pax.transx.tm.narayana.cfg
   service.pid = org.ops4j.pax.transx.tm.narayana
```

概述：

- **用于 Karaf 的 Fuse 包括 Narayana 事务管理器的全功能全球。**
- **事务管理器在各种客户端接口(JTA、Spring-tx、PAX JMS)下正确公开。**
- **您可以使用标准 OSGi 方法 Configuration Admin 来配置 Narayana，该方法可在 org.ops4j.pax.transx.tm.narayana 中提供。**

- 默认配置在 $FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg 中提供。

4.2. 支持的事务协议

Narayana 事务管理器 是 EAP 中使用的 JBoss/红帽产品。Narayana 是一个交易工具包，为使用各种标准交易协议开发的应用程序提供支持：

- JTA
- JTS
- web-Service Transactions
- REST Transactions
- STM
- XATMI/TX

4.3. 关于 NARAYANA 配置

pax-transx-tm-narayana 捆绑包包括 jbossts-properties.xml 文件，它提供事务管理器的不同方面的默认配置。所有这些属性都可以在 $FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg 文件中直接覆盖，或使用 Configuration Admin API 覆盖。

Narayana 的基本配置是通过各种 EnvironmentBean 对象实现的。每个 bean 都可以通过使用具有不同前缀的属性来配置。下表提供了使用的配置对象和前缀的概述：

配置 Bean	属性前缀
com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean	com.arjuna.ats.arjuna.coordinator
com.arjuna.ats.arjuna.common.CoreEnvironmentBean	com.arjuna.ats.arjuna
com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqJournalEnvironmentBean	com.arjuna.ats.arjuna.hornetqjournal
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean	com.arjuna.ats.arjuna.objectstore
com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean	com.arjuna.ats.arjuna.recovery
com.arjuna.ats.jdbc.common.JDBCEnvironmentBean	com.arjuna.ats.jdbc
com.arjuna.ats.jta.common.JTAEnvironmentBean	com.arjuna.ats.jta
com.arjuna.ats.txoj.common.TxojEnvironmentBean	com.arjuna.ats.txoj.lockstore

*前缀*可以简化配置。但是，您通常使用以下格式之一：

NameEnvironmentBean.propertyName（首选格式）或

完全限定-class-name.field-name

例如，考虑 com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase 字段。它可以通过使用 com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase 属性进行配置，也可以使用更简单(preferred)表单 CoordinatorEnvironment.commitOnePhase 进行配置。有关如何设置属性以及可配置 Bean 的完整详情，请参阅 Narayana 产品文档。

一些 Bean （如 ObjectStoreEnvironmentBean ）可能会多次配置，每个指定实例都为不同的目的提供配置。*在这种情况下，实例的名称在前缀（上述任意位置）和 field-name 之间使用。例如，可以使用命名的属性配置名为 communicationStore 的 ObjectStore 实例的对象存储类型：*

- ***com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType***
- ***ObjectStoreEnvironmentBean.communicationStore.objectStoreType***

4.4. 配置日志存储

最重要的配置是对象日志存储的类型和位置。com.arjuna.ats.arjuna.objectstore.ObjectStoreAPI 接口通常有三个实现：

com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqObjectStoreAdaptor

在内部使用 org.apache.activemq.artemis.core.journal.Journal 存储。

com.arjuna.ats.internal.arjuna.objectstore.jdbc.JDBCStore

使用 JDBC 来保留 TX 日志文件。

com.arjuna.ats.internal.arjuna.objectstore.FileSystemStore （及专用实施）

使用基于文件的自定义日志存储。

默认情况下，Fuse 使用 com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore，它是 FileSystemStore 的专用实现。

Narayana 有三个存储保留了事务/对象日志：

- ***defaultStore***
- ***communicationStore***
- ***stateStore***

如需了解更多详细信息，请参阅 Narayana 文档中的状态管理。

这三种存储的默认配置是 ：

```
# default store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
```

```
# communication store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
# state store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
```

ShadowNoFileLockStore 使用基础目录(objectStoreDir)和特定存储的目录(localOSRoot)配置。

Narayana 文档 指南 **中包含许多配置选项。但是，Narayana 文档指出，对配置选项的规范引用是各种 EnvironmentBean 类的 Javadoc。**

第 5 章 使用 NARAYANA 事务管理器

本节通过实施 javax.transaction.UserTransaction 接口、org.springframework.transaction.PlatformTransactionManager 接口或 javax.transaction.Transaction 接口来使用 Narayana 事务管理器的详细信息。您选择使用哪个接口取决于应用程序的需求。在本章的结尾处，可以讨论列出 XA 资源的问题解决办法。该信息组织如下：

- 第 5.1 节 “使用 UserTransaction 对象”
- 第 5.2 节 “使用 TransactionManager 对象”
- 第 5.3 节 “使用 Transaction 对象”
- 第 5.4 节 “解决 XA enlistment 问题”

有关 Java 事务 API 详情，请查看 Java Transaction API (JTA) 1.2 规格和 Javadoc。

5.1. 使用 USERTRANSACTION 对象

实施 javax.transaction.UserTransaction 接口以进行事务处理。也就是说，用于开始、提交或回滚事务。这是您很可能直接在应用程序代码中使用的 JTA 接口。但是，UserTransaction 接口只是分解事务的方法之一。有关您可以分离事务的不同方法的讨论，请参阅 第 9 章 编写使用事务的 Camel 应用程序。

5.1.1. UserTransaction 接口的定义

JTA UserTransaction 接口定义如下：

```
public interface javax.transaction.UserTransaction {

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);
}
```

5.1.2. 用户事务方法的描述

UserTransaction 接口定义了以下方法：

begin()

启动新的事务，并将其与当前线程相关联。如果有任何 XA 资源与此事务相关联，则事务会隐式成为 XA 事务。

commit()

正常完成当前事务，以便所有待处理的更改变得永久。提交后，不再有与当前线程关联的事务。

注意

但是，如果当前事务仅标记为回滚，则当称为 commit （） 时实际会回滚事务。

rollback()

立即中止事务，以便丢弃所有待处理的更改。回滚后，不再有与当前线程关联的事务。

setRollbackOnly()

修改当前事务的状态，以便回滚是唯一可能的结果，但尚未执行回滚。

getStatus()

返回当前事务的状态，可以是以下整数值之一，如 javax.transaction.Status 接口中定义的：

- STATUS_ACTIVE
- STATUS_COMMITTED
- STATUS_COMMITTING
- STATE_MARKED_ROLLBACK
- STATUS_NO_TRANSACTION
- STATUS_PREPARED
- STATUS_PREPARING
- STATUS_ROLLEDBACK
- STATUS_ROLLING_BACK
- STATUS_UNKNOWN

setTransactionTimeout()

自定义当前事务的超时时间，以秒为单位指定。如果在指定的超时时间内没有解决事务，事务管理器会自动回滚。

5.2. 使用 TRANSACTIONMANAGER 对象

使用 javax.transaction.TransactionManager 对象的最常见方法是将它传递给框架 API，例如，传递给 Camel JMS 组件。这可让框架在您事务处理后进行查找。有时，您可能想要直接使用 TransactionManager 对象。当您需要访问高级事务 API，如 suspend （） 和 resume （） 方法时，这非常有用。

5.2.1. TransactionManager 接口的定义

JTA TransactionManager 接口有以下定义：

```
interface javax.transaction.TransactionManager {

   // Same as UserTransaction methods

   public void begin();

   public void commit();
```

```
    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);

    // Extra TransactionManager methods

    public Transaction getTransaction();

    public Transaction suspend() ;

    public void resume(Transaction tobj);
}
```

5.2.2. TransactionManager 方法的描述

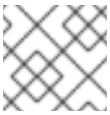
TransactionManager 接口支持 UserTransaction 接口中找到的所有方法。您可以使用 TransactionManager 对象进行事务处理。另外，TransactionManager 对象支持以下方法：

getTransaction()

获取对当前事务的引用，这是与当前线程关联的事务。如果没有当前的事务，此方法会返回 null。

suspend()

将当前事务从当前线程中分离，并返回对事务的引用。调用此方法后，当前线程不再有事务上下文。您在此时之后执行的所有工作都不再在事务环境中完成。



注意

并非所有事务管理器都支持挂起事务。但是，Narayana 支持此功能。

resume()

将暂停的事务重新附加到当前线程上下文。调用此方法后，事务上下文会被恢复，并在此时在事务环境中完成的所有工作。

5.3. 使用 TRANSACTION 对象

如果您要暂停/恢复事务，或者需要明确列出资源，您可能需要直接使用 javax.transaction.Transaction 对象。如 第 5.4 节 “解决 XA enlistment 问题” 所述，框架或容器通常会自动处理资源。

5.3.1. Transaction 接口的定义

JTA Transaction 接口具有以下定义：

```
interface javax.transaction.Transaction {

    public void commit();

    public void rollback();
```

```
    public void setRollbackOnly();

    public int getStatus();

    public boolean enlistResource(XAResource xaRes);

    public boolean delistResource(XAResource xaRes, int flag);

    public void registerSynchronization(Synchronization sync);
}
```

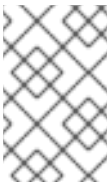
5.3.2. Transaction 方法的描述

commit （）, rollback （）, setRollbackOnly （）, 和 getStatus （） 方法的行为与 UserTransaction 接口的对应方法相同。实际上， UserTransaction 对象是一个方便的打包程序，用于检索当前事务，然后在 Transaction 对象上调用对应的方法。

另外， Transaction 接口定义了以下方法， 它在 UserTransaction 接口中没有对应的方法：

enlistResource()

将 XA 资源与当前事务相关联。



注意

这个方法是 XA 事务上下文中的关键重要。使用当前事务代表 XA 事务的功能， 可以精确列出多个 XA 资源。另一方面，明确列出资源是微不足道的资源， 您通常希望您的框架或容器为您完成此操作。例如， 请参阅 第 5.4 节 “解决 XA enlistment 问题”。

delistResource()

解除指定资源与事务相关的关联。flag 参数可以采用 javax.transaction.Transaction 接口中定义的以下整数值之一：

- TMSUCCESS
- TMFAIL
- TMSUSPEND

registerSynchronization()

使用当前事务注册 javax.transaction.Synchronization 对象。Synchronization 对象仅在提交准备阶段之前收到回调，并在事务完成后接收回调。

5.4. 解决 XA ENLISTMENT 问题

要列出 XA 资源的标准 JTA 方法是将 XA 资源明确添加到当前 javax.transaction.Transaction 对象， 它代表当前的事务。换句话说，每次新事务启动时， 您必须明确列出 XA 资源。

5.4.1. 如何列出 XA 资源

使用 事务 列出 XA 资源涉及在事务中调用enlistResource （） 方法。例如，给定一个 TransactionManager 对象和 XAResource 对象， 您可以按如下方式列出 XAResource 对象：

```
// Java
import javax.transaction.Transaction;
import javax.transaction.TransactionManager;
import javax.transaction.xa.XAResource;
...
// Given:
// 'tm' of type TransactionManager
// 'xaResource' of type XAResource

// Start the transaction
tm.begin();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaResource);

// Do some work...
...

// End the transaction
tm.commit();
```

列出资源的技巧方面是必须在每个新事务上列出资源，在开始使用资源前必须列出该资源。如果您明确列出资源，则可能会出现容易出错的代码，这些代码使用 enlistResource （） 调用。此外，有时很难在正确的位置调用 enlistResource （），例如，如果您使用一个隐藏了一些事务详细信息的框架时，会出现这种情况。

5.4.2. 关于自动加入

使用支持 XA 资源的自动连接的功能，而不是明确列出 XA 资源。例如，在使用 JMS 和 JDBC 资源的情况下，标准技术是使用支持自动加入的打包程序类。

JDBC 和 JMS 访问的常见模式是：

1. 应用程序代码需要 javax.sql.DataSource for JDBC access 和javax.jms.ConnectionFactory for JMS 获取 JDBC 或 JMS 连接。
2. 在应用程序/OSGi 服务器中，会注册这些接口的数据库或代理特定实现。
3. application/OSGi 服务器将 database/broker 特定工厂嵌套成通用、池、清单工厂。

这样，应用程序代码仍然使用 javax.sql.DataSource 和 javax.jms.ConnectionFactory，但在访问它们时内部存在额外的功能，这通常涉及：

- 连接池 - 而不是在每次创建新连接时都创建新的连接，而是使用重新初始化的连接池。池的另一个 方面可能是连接的定期验证。
- JTA enlistment - 在返回 java.sql.Connection (JDBC) 或 javax.jms.Connection (JMS) 的实例之前，如果实际连接对象是 true XA 资源，则会注册实际连接对象。注册会在 JTA 事务中发生（如果可用）。

通过自动协助，应用程序代码不必更改。

有关 JDBC 数据源和 JMS 连接工厂的池和配套打包程序的更多信息，请参阅 第 6 章 使用 JDBC 数据源 和 第 7 章 使用 JMS 连接工厂。

第 6 章 使用 JDBC 数据源

以下主题讨论 Fuse OSGi 运行时中使用的 JDBC 数据源：

6.1. 关于连接接口

用于执行数据操作的最重要对象是 java.sql.Connection 接口的实现。从 Fuse 配置的角度来看，了解如何获取连接对象非常重要。

包含相关对象的库有：

- postgresql: **mvn:org.postgresql/postgresql/42.2.5**
- MySQL: **mvn:mysql/mysql-connector-java/5.1.34**

现有实现（包含在驱动程序 JAR 中）提供：

- postgresql: **org.postgresql.jdbc.PgConnection**
- mysql: **com.mysql.jdbc. JDBC4Connection**（请参阅 **com.mysql.jdbc.Driver**）的各种 **connect*（）方法。**

这些实现包含特定于数据库的逻辑，用于执行 DML、DDL 和简单的事务管理。

在理论上，可以手动创建这些连接对象，但有两个 JDBC 方法用于隐藏详情以提供干净的 API：

- **java.sql.Driver.connect （）** - 这个方法在独立应用程序中被长时间使用。
- **javax.sql.DataSource.getConnection （）** - 这是使用工厂模式的首选方法。类似的方法用于从 JMS 连接工厂获取 JMS 连接。

此处不讨论驱动程序管理器方法。它足以说明此方法只是给定连接对象的普通构造器之上的小层。

除了有效实现特定于数据库的通信协议的 java.sql.Connection 外，还有另外两个专用的连接接口：

- **javax.sql.PooledConnection** 代表物理连接。您的代码不会与这个池的连接直接交互。反之，使用 **getConnection （）** 方法获取的连接。这种间接允许管理应用服务器级别的连接池。使用 **getConnection （）** 获取的连接通常是代理。当此类代理连接关闭时，物理连接不会关闭，而是在受管连接池中再次可用。

- *javax.sql.XAConnection 允许获取与 javax.transaction.TransactionManager 一起使用的 XA 感知连接的 javax.transaction.xa.XAResource 对象。由于 javax.sql.XAConnection 扩展 javax.sql.PooledConnection，它还提供 'getConnection （） 方法，它提供对具有典型 DML/DQL 方法的 JDBC 连接对象的访问。*

6.2. JDBC 数据源概述

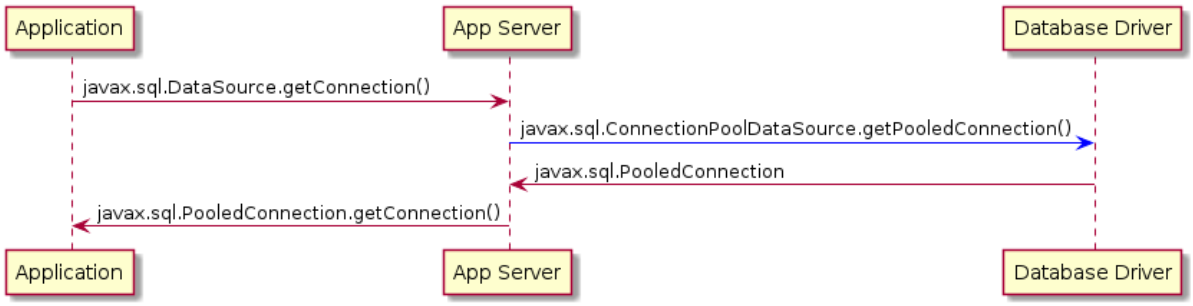
JDBC 1.4 标准引入了 javax.sql.DataSource 接口，它充当 java.sql.Connection 对象的工厂。通常，此类数据源绑定到 JNDI 注册表，并位于内或注入到 servlet 或 EJB 等 Java EE 组件中。主要方面是，这些数据源是在 应用服务器 内配置的，并根据名称在部署的应用程序 中引用。

以下 连接 对象具有自己的数据源 ：

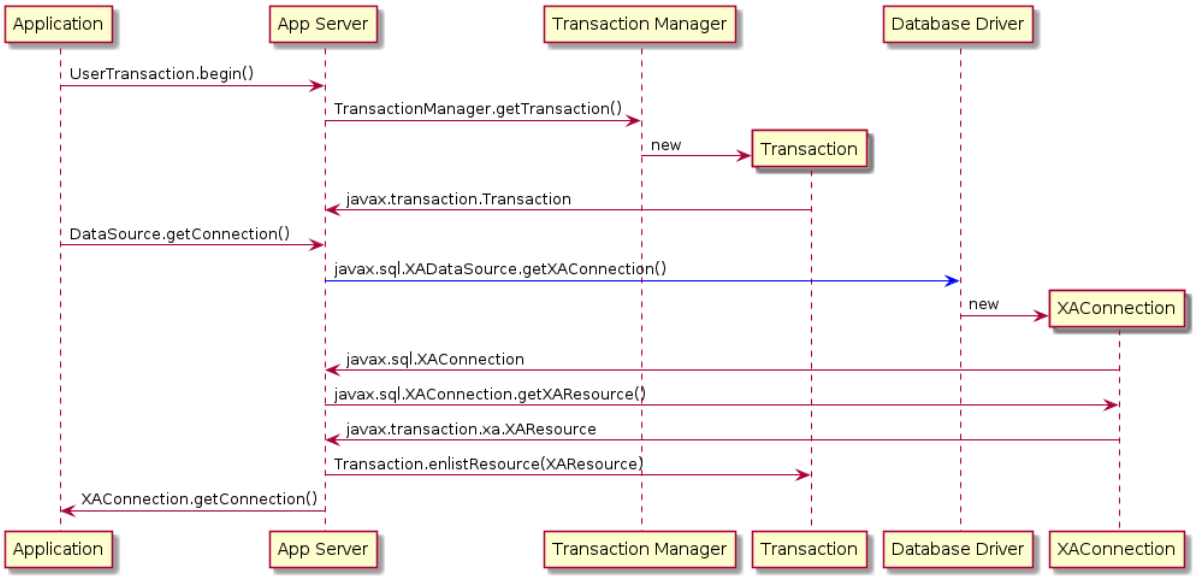
数据源	连接
javax.sql.DataSource	java.sql.Connection
javax.sql.ConnectionPoolDataSource	javax.sql.PooledConnection
javax.sql.XADataSource	javax.sql.XAConnection

以上每个 数据源 之间最重要的区别如下：

- *javax.sql.DataSource 最重要的是一个 工厂-like 对象用于获取 java.sql.Connection 实例。大多数 javax.sql.DataSource 实施通常执行 连接池 的事实不应更改图片。这是应当供应用程序代码使用的唯一接口。您实施以下哪个项无关紧要：*
  - *直接 JDBC 访问*
  - *JPA persistence 单元配置(< jta-data-source> 或 <non-jta-data-source>)*
  - *流行的库，如 Apache Camel 或 Spring Framework*
- *javax.sql.ConnectionPoolDataSource 最重要的是一个通用（特定于数据库）连接池/数据源和特定于数据库的数据源之间的 桥接。它可能被视为 SPI 接口。应用程序代码通常处理从 JNDI 获取并由应用服务器（可能使用 commons-dbcp2等库）实现的通用 javax.sql.DataSource 对象。在另一端，应用程序代码不直接与 javax.sql.ConnectionPoolDataSource 的接口。它用于应用服务器和特定于数据库的驱动程序。以下序列图显示了：*



- *javax.sql.XADataSource 是获取 javax.sql.XAConnection 和 javax.transaction.xa.XAResource 的方法。与 javax.sql.ConnectionPoolDataSource 相同，它在应用服务器和数据库特定驱动程序之间使用。以下是不同执行器的稍修改图表，这一次包括 JTA Transaction Manager：*



如上两个图所示，您与 App Server 交互，它是一个通用的实体，您可以在其中配置 javax.sql.DataSource 和 javax.transaction.UserTransaction 实例。此类实例可以通过 JNDI 访问，也可以使用 CDI 或其他依赖项机制注入。

重要

重要点是，即使应用使用 XA 事务和/或连接池，应用与 javax.sql.DataSource 交互，而不是另外两个 JDBC 数据源接口。

6.2.1. 特定于数据库和通用数据源

JDBC 数据源实现分为两个类别：

- *通用 javax.sql.DataSource 实现，例如：*
  - *Apache Commons DBCP (2)*
  - *Apache Tomcat JDBC （基于 DBCP）*
- *javax.sql.DataSource,javax.sql.XADataSource, 和 javax.sql.ConnectionPoolDataSource的数据库特定实现*

通用 javax.sql.DataSource 实现无法自行创建特定于数据库的连接，这可能会造成混淆。即使 通用 数据源可以使用 java.sql.Driver.connect （） 或 java.sql.DriverManager.getConnection （），它通常最好使用特定于数据库的 javax.sql.DataSource 实现配置此 通用 数据源。

当 通用 数据源要与 JTA 交互时，必须将其配置为 javax.sql.XADataSource 的特定数据库实施。

为关闭图像，一般 数据源 通常不需要 javax.sql.ConnectionPoolDataSource 的特定数据库实现来执行连接池。现有池通常在没有标准 JDBC 接口的情况下处理池(javax.sql.ConnectionPoolDataSource 和 javax.sql.PooledConnection)，而是使用自己的自定义实现。

6.2.2. 有些通用数据源

考虑一个知名、通用数据源、Apache Commons DBCP (2) 示例。

javax.sql.XADataSource 实现

DBCP2 不包含 javax.sql.XADataSource 的任何实施，这是预期的。

javax.sql.ConnectionPoolDataSource implementations

DBCP2 包括了 javax.sql.ConnectionPoolDataSource 的实现：org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS。它通过调用 java.sql.DriverManager.getConnection （） 来创建 javax.sql.PooledConnection 对象。这个池不应该直接使用，它应该被视为 驱动程序的适配器 ：

- *不要提供自己的 javax.sql.ConnectionPoolDataSource 实现*
- *您需要根据 JDBC 建议对 连接池使用*

如上图所示，驱动程序直接提供 javax.sql.ConnectionPoolDataSource，或提供 org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS 适配器的帮助，而 DBCP2 则通过以下之一实现 应用服务器 合同：

- *org.apache.commons.dbcp2.datasources.PerUserPoolDataSource*
- *org.apache.commons.dbcp2.datasources.SharedPoolDataSource*

这两个池都会在配置阶段获取 javax.sql.ConnectionPoolDataSource 实例。

这是 DBCP2 中最重要的、最有趣的部分：

javax.sql.DataSource implementations

要实现连接池功能，您不必遵循 JDBC 建议 来使用 javax.sql.ConnectionPoolDataSource → javax.sql.PooledConnection SPI。

以下是 DBCP 2 的一般数据源列表：

- *org.apache.commons.dbcp2.BasicDataSource*
- *org.apache.commons.dbcp2.managed.BasicManagedDataSource*
- *org.apache.commons.dbcp2.PoolingDataSource*
- *org.apache.commons.dbcp2.managed.ManagedDataSource*

这里有两个 axes:

基本 与 池

这个 axis 决定 池配置 方面。

两种类型的数据源都执行 java.sql.Connection 对象的池。唯一的 区别是：

- *使用 bean 属性配置 基本 数据源，如 maxTotal 或 minIdle，用来配置 org.apache.commons.pool2.impl.GenericObjectPool 的内部实例。*
- *池 数据源配置有外部创建的/配置 org.apache.commons.pool2.ObjectPool。*

受管 与非管理

这个 axis 决定 连接创建 方面和 JTA 行为：

- *非管理的基本数据源通过使用 内部 java.sql.Driver.connect （） 创建 java.sql.Connection 实例。*
  非管理的池 数据源使用传递的 org.apache.commons.pool2.ObjectPool 对象创建 java.sql.Connection 实例。

- 受管池 数据源将 java.sql.Connection 实例嵌套在 org.apache.commons.dbcp2.managed.ManagedConnection 对象内，确保在 JTA 上下文中调用 javax.transaction.Transaction.enlistResource （）。但仍然从池配置的任何 org.apache.commons.pool2.ObjectPool 对象获取实际连接。
  受管的基本 数据源可为您配置专用的 org.apache.commons.pool2.ObjectPool。相反，配置特定于数据库的现有 javax.sql.XADataSource 对象就足够了。bean 属性用于创建 org.apache.commons.pool2.impl.GenericObjectPool 的内部实例，后者将传递到 受管池 数据源的内部实例(org.apache.commons.dbcp2.managed.ManagedDataSource)。

注意

DBCP2 唯一无法执行的操作是 XA 事务恢复。DBCP2 在活跃的 JTA 事务中正确列出 XAResources，但它没有执行恢复。这应该单独完成，配置通常特定于所选的事务管理器实施（如 Narayana）。

6.2.3. 要使用的模式

推荐的模式是：

- 创建或获取 特定于数据库的 javax.sql.DataSource 或 javax.sql.XADataSource 实例，其中包含特定于数据库的配置(URL、凭证等)，可以创建 connection/XA 连接。
- 创建或获取 特定于数据库的 javax.sql.DataSource 实例（内部配置有上述、特定于数据库的数据源），以及特定于数据库的配置（连接池、事务管理器等）。
- 使用 javax.sql.DataSource 获取 java.sql.Connection 的实例并执行 JDBC 操作。

以下是 规范 示例：

```
// Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
PGXADataSource postgresql = new org.postgresql.xa.PGXADataSource();
// Database-specific configuration
postgresql.setUrl("jdbc:postgresql://localhost:5432/reportdb");
postgresql.setUser("fuse");
postgresql.setPassword("fuse");
postgresql.setCurrentSchema("report");
postgresql.setConnectTimeout(5);
// ...

// Non database-specific, pooling, enlisting javax.sql.DataSource
BasicManagedDataSource pool = new
org.apache.commons.dbcp2.managed.BasicManagedDataSource();
// Delegate to database-specific XADatasource
pool.setXaDataSourceInstance(postgresql);
// Delegate to JTA transaction manager
pool.setTransactionManager(transactionManager);
// Non database-specific configuration
pool.setMinIdle(3);
pool.setMaxTotal(10);
pool.setValidationQuery("select schema_name, schema_owner from
information_schema.schemata");
// ...

// JDBC code:
javax.sql.DataSource applicationDataSource = pool;
```

```
try (Connection c = applicationDataSource.getConnection()) {
    try (Statement st = c.createStatement()) {
        try (ResultSet rs = st.executeQuery("select ...")) {
            // ....
```

在 Fuse 环境中，有许多配置选项，且不需要使用 DBCP2。

6.3. 配置 JDBC 数据源

如 OSGi 事务架构 中所述，必须在 OSGi 服务注册表中注册一些服务。正如您可以使用 javax.transaction.UserTransaction 接口（例如 javax.transaction.UserTransaction 接口）找到（查找）事务管理器实例一样，您可以使用 javax.sql.DataSource 接口与 JDBC 数据源执行相同的操作。要求是：

- 特定于数据库的数据源，可与目标数据库通信
- 通用数据源，您可以在其中配置池，以及可能进行事务管理(XA)

在 OSGi 环境中，如 Fuse，如果数据源作为 OSGi 服务注册，则可以从应用程序访问。从根本上讲，它按以下方式完成：

```
org.osgi.framework.BundleContext.registerService(javax.sql.DataSource.class,
                                                 dataSourceObject,
                                                 properties);
org.osgi.framework.BundleContext.registerService(javax.sql.XADataSource.class,
                                                 xaDataSourceObject,
                                                 properties);
```

注册这些服务的方法有两种：

- 使用 jdbc:ds-create Karaf console 命令发布数据源。这是配置方法。
- 使用 Blueprint、SOSOS Declarative Services (SCR)或仅 BundleContext.registerService （） API 调用等方法发布数据源。这个方法需要一个包含代码和/或元数据的专用 OSGi 捆绑包。这是 the_deployment method_。

6.4. 使用 OSGI JDBC 服务

chapter 125 of the OSGi Enterprise R6 规范在 org.osgi.service.jdbc 软件包中定义了单一接口。这是 OSGi 处理数据源的方式：

```
public interface DataSourceFactory {

    java.sql.Driver createDriver(Properties props);

    javax.sql.DataSource createDataSource(Properties props);

    javax.sql.ConnectionPoolDataSource createConnectionPoolDataSource(Properties props);

    javax.sql.XADataSource createXADataSource(Properties props);
}
```

如前文所述，可以从 java.sql.Driver 直接获取普通 java.sql.Connection 连接。

Generic org.osgi.service.jdbc.DataSourceFactory

org.osgi.service.jdbc.DataSourceFactory 的最简单实现是 org.ops4j.pax.jdbc.impl.DriverDataSourceFactory 由 mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0 捆绑包提供。所有这些都跟踪捆绑包，其中可能包括标准 Java™ ServiceLoader 工具的 /META-INF/services/java.sql.Driver 描述符。如果您安装任何标准 JDBC 驱动程序，pax-jdbc 捆绑包注册一个 DataSourceFactory，它可以通过 java.sql.Driver.connect （） 调用来获取连接。

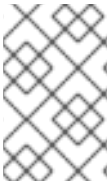
```
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 223
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 224
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 225
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 226

karaf@root()> bundle:services -p org.postgresql.jdbc42

PostgreSQL JDBC Driver JDBC42 (225) provides:
----------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = org.postgresql.Driver
osgi.jdbc.driver.name = PostgreSQL JDBC Driver
osgi.jdbc.driver.version = 42.2.5
service.bundleid = 225
service.id = 242
service.scope = singleton

karaf@root()> bundle:services -p com.mysql.jdbc

Oracle Corporation's JDBC Driver for MySQL (226) provides:
-----------------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 243
service.scope = singleton
-----
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 244
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = org.postgresql.Driver
 osgi.jdbc.driver.name = PostgreSQL JDBC Driver
 osgi.jdbc.driver.version = 42.2.5
 service.bundleid = 225
```

```
 service.id = 242
 service.scope = singleton
Provided by :
 PostgreSQL JDBC Driver JDBC42 (225)

[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
 service.id = 243
 service.scope = singleton
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)

[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
 service.id = 244
 service.scope = singleton
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)
```

在上述命令中，javax.sql.DataSource 服务仍未注册，但您更接近一个步骤。以上中间的 org.osgi.service.jdbc.DataSourceFactory 服务可用于获取：

- *java.sql.Driver*
- *javax.sql.DataSource 通过将属性传递：url、用户和密码 到createDataSource （） 方法。*

您无法从非特定于数据库的 pax- jdbc 捆绑包创建的通用 org.sql.XADataSource 或 javax.sql. XADataSource 获取 javax.sql.ConnectionPoolDataSource 或 javax.sql.XADataSource。



注意

mvn:org.postgresql/postgresql/42.2.5 捆绑包可以正确地实现 OSGi JDBC 规格，并使用实施的所有方法注册 org.osgi.service.jdbc.DataSourceFactory 实例，包括创建 XA 和 ConnectionPool 数据源的方法。

特定于数据库的 org.osgi.service.jdbc.DataSourceFactory 实现

还有其他捆绑包，例如：

- *mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0*
- *mvn:org.ops4j.pax.jdbc/pax-jdbc-db2/1.3.0*
- *...*

这些捆绑包注册特定于数据库的 org.osgi.service.jdbc.DataSourceFactory 服务，它可以返回所有类型的工厂，包括 javax.sql.ConnectionPoolDataSource 和 javax.sql.XADataSource。例如：

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 227

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.mysql

OPS4J Pax JDBC MySQL Driver Adapter (227) provides:
---------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = mysql
service.bundleid = 227
service.id = 245
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 227
 service.id = 245
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (227)
```

6.4.1. PAX- JDBC 配置服务

使用 pax-jdbc （或 pax-jdbc-mysql, pax-jdbc-oracle, ...）捆绑包时，您可以注册 org.osgi.service.jdbc.DataSourceFactory 服务，可用于获取给定数据库的数据源（请参阅 第 6.2.1 节 “特定于数据库和通用数据源”）。但是还没有实际的数据源。

mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0 捆绑包提供了一个受管服务工厂，它有两个操作：

- 跟踪 org.osgi.service.jdbc.DataSourceFactory OSGi 服务，以调用其方法：

  ```
  public DataSource createDataSource(Properties props);
  public XADataSource createXADataSource(Properties props);
  public ConnectionPoolDataSource createConnectionPoolDataSource(Properties
  props);
  ```

- 跟踪 org.ops4j.datasource 工厂 PID，以收集以上方法所需的属性。如果您使用 Configuration Admin 服务的任何方法创建 工厂配置，例如，通过创建一个 ${karaf.etc}/org.ops4j.datasource-mysql.cfg 文件，您可以执行最终步骤来公开特定于数据库的实际数据源。

以下是从全新的 Fuse 安装开始的详细 规范 逐步指南。

注意

您明确安装捆绑包而不是功能，以精确显示需要哪些捆绑包。为方便起见，PAX JDBC 项目提供了多种数据库产品和配置方法的功能。

1. 使用 /META-INF/services/java.sql.Driver 安装 JDBC 驱动程序：

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
```

2. **安装 OSGi JDBC 服务捆绑包和 pax-jdbc-mysql 捆绑包，该捆绑包注册 中介org.osgi.service.jdbc.DataSourceFactory:**

```
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 225
 service.id = 242
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (225)
```

3. **安装 pax-jdbc 捆绑包和 pax-jdbc-config 捆绑包，该捆绑包跟踪 org.osgi.service.jdbc.DataSourceFactory 服务和org.ops4j.datasource factory PIDs:**

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.config

OPS4J Pax JDBC Config (228) provides:
--------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 228
service.id = 245
service.pid = org.ops4j.datasource
service.scope = singleton
```

4. **创建 工厂配置 （假设 MySQL 服务器正在运行）：**

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set user fuse
karaf@root()> config:property-set password fuse
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
```

```
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
  dataSourceName = mysqlds
  felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
  osgi.jdbc.driver.name = mysql
  password = fuse
  service.factoryPid = org.ops4j.datasource
  service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
  url = jdbc:mysql://localhost:3306/reportdb
  user = fuse
```

5. *检查* ***pax-jdbc-config*** *是否将配置处理到* ***javax.sql.DataSource*** *服务中：*

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 password = fuse
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
 service.scope = singleton
 url = jdbc:mysql://localhost:3306/reportdb
 user = fuse
Provided by :
 OPS4J Pax JDBC Config (228)
```

您现在有一个特定于数据库的实际（还没有池）数据源。您可以已经注入需要它的位置。例如，您可以使用 Karaf 命令查询数据库：

```
karaf@root()> feature:install -v jdbc
Adding features: jdbc/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jdbc:ds-list
Mon May 14 08:46:22 CEST 2018 WARN: Establishing SSL connection without server's identity
verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL
connection must be established by default if explicit option isn't set. For compliance with existing
applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to
explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server
certificate verification.
Name    | Product | Version | URL                          | Status
--------+---------+---------+------------------------------+-------
mysqlds | MySQL   | 5.7.21  | jdbc:mysql://localhost:3306/reportdb | OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
Mon May 14 08:46:46 CEST 2018 WARN: Establishing SSL connection without server's identity
verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL
connection must be established by default if explicit option isn't set. For compliance with existing
```

```
applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to
explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server
certificate verification.
date                | summary   | name   | details                  | id | email
--------------------+-----------+--------+--------------------------+----+-----------------
2018-02-20 08:00:00.0 | Incident 1 | User 1 | This is a report incident 001 | 1 | user1@redhat.com
2018-02-20 08:10:00.0 | Incident 2 | User 2 | This is a report incident 002 | 2 | user2@redhat.com
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3 | user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4 | user4@redhat.com
```

在上例中，您可以看到 MySQL 警告。这不是一个问题。可以提供任何属性（不仅是 OSGi JDBC 特定属性）：

```
karaf@root()> config:property-set --pid org.ops4j.datasource.a7941498-9b62-4ed7-94f3-
8c7ac9365313 useSSL false

karaf@root()> jdbc:ds-list
Name    | Product | Version | URL                          | Status
--------+---------+---------+------------------------------+-------
mysqlds | MySQL   | 5.7.21  | jdbc:mysql://localhost:3306/reportdb | OK
```

6.4.2. 处理的属性摘要

admin factory PID **配置中的属性**传递给**相关的 org.osgi.service.jdbc.DataSourceFactory** 实现。

generic

org.ops4j.pax.jdbc.impl.DriverDataSourceFactory properties:

- ***url***
- ***user***
- ***password***

DB2

org.ops4j.pax.jdbc.db2.impl.DB2DataSourceFactory 属性包括这些实现类**的所有** bean **属性：**

- ***com.ibm.db2.jcc.DB2SimpleDataSource***
- ***com.ibm.db2.jcc.DB2ConnectionPoolDataSource***
- ***com.ibm.db2.jcc.DB2XADataSource***

PostgreSQL

Nnative **org.postgresql.osgi.PGDataSourceFactory 属性包括 org.postgresql.PGProperty 中指定的所有属性。**

HSQLDB

org.ops4j.pax.jdbc.hsqldb.impl.HsqldbDataSourceFactory properties:

- *url*
- *user*
- *password*
- *databaseName*
- *的所有 bean 属性*
  - *org.hsqldb.jdbc.JDBCDataSource*
  - *org.hsqldb.jdbc.pool.JDBCPooledDataSource*
  - *org.hsqldb.jdbc.pool.JDBCXADataSource*

SQL Server 和 Sybase

org.ops4j.pax.jdbc.jtds.impl.JTDSDataSourceFactory 属性包括 net.sourceforge.jtds.jdbcx.JtdsDataSource 的所有 bean 属性。

SQL Server

org.ops4j.pax.jdbc.mssql.impl.MSSQLDataSourceFactory 属性：

- *url*
- *user*
- *password*
- *databaseName*
- *serverName*
- *portnumber*
- *的所有 bean 属性*
  - *com.microsoft.sqlserver.jdbc.SQLServerDataSource*
  - *com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource*
  - *com.microsoft.sqlserver.jdbc.SQLServerXADataSource*

MySQL

org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory properties:

- *url*
- *user*
- *password*
- *databaseName*

- ***serverName***
- ***portnumber***
- ***的所有 bean 属性***
  - ***com.mysql.jdbc.jdbc2.optional.MysqlDataSource***
  - ***com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource***
  - ***com.mysql.jdbc.jdbc2.optional.MysqlXADataSource***

Oracle

***org.ops4j.pax.jdbc.oracle.impl.OracleDataSourceFactory** properties:*

- ***url***
- ***databaseName***
- ***serverName***
- ***user***
- ***password***
- ***的所有 bean 属性***
  - ***oracle.jdbc.pool.OracleDataSource***
  - ***oracle.jdbc.pool.OracleConnectionPoolDataSource***
  - ***oracle.jdbc.xa.client.OracleXADataSource***

SQLite

***org.ops4j.pax.jdbc.sqlite.impl.SqliteDataSourceFactory** properties:*

- ***url***
- ***databaseName***
- ***org.sqlite.SQLiteDataSource的所有 bean 属性***

6.4.3. pax-jdb-config 捆绑包如何处理属性

pax-jdbc-config 捆绑包处理前缀为 jdbc 的属性。所有这些属性都将删除这个前缀，剩余的名称将被传递。

下面是一个全新的 Fuse 安装开始的示例：

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
```

```
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   felix.fileinstall.filename = file:/data/servers/7.13.0.fuse-7_13_0-00012-redhat-
00001/etc/org.ops4j.datasource-mysql.cfg
   jdbc.password = fuse
   jdbc.url = jdbc:mysql://localhost:3306/reportdb
   jdbc.useSSL = false
   jdbc.user = fuse
   osgi.jdbc.driver.name = mysql
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (228)
```

***pax-jdbc-config** 捆绑包需要这些属性：*

- ***osgi.jdbc.driver.name***
- ***dataSourceName***
- ***dataSourceType***

查找并调用相关的 org.osgi.service.jdbc.DataSourceFactory 方法。前缀为 jdbc. 的属性会被传递（在删除前缀后），例如 org.osgi.service.jdbc.DataSourceFactory.createDataSource (properties)。但是，会添加这些属性，而不删除前缀，如 javax.sql.DataSource OSGi 服务的属性。

6.5. 使用 JDBC 控制台命令

Fuse 提供 jdbc 功能，其中包含 jdbc: 范围中的 shell 命令。前面的示例演示了使用 jdbc:query。另外，还有一些命令隐藏了创建配置管理员配置的需求。*

从全新的 Fuse 实例开始，您可以使用通用 DataSourceFactory 服务注册数据库特定数据源，如下所示：

```
karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-factories
Name │ Class │ Version
─────┼───────┼────────

karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 228

karaf@root()> jdbc:ds-factories
Name          │ Class                          │ Version
──────────────┼────────────────────────────────┼──────
com.mysql.jdbc │ com.mysql.jdbc.Driver              │ 5.1.34
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

以下是注册 MySQL 特定 DataSourceFactory 服务的示例：

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install pax-jdbc-mysql

karaf@root()> la -l|grep mysql

232 │ Active   │  80 │ 5.1.34              │ mvn:mysql/mysql-connector-java/5.1.34

233 │ Active   │  80 │ 1.3.0               │ mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0

karaf@root()> jdbc:ds-factories
Name          │ Class                          │ Version
──────────────┼────────────────────────────────┼──────
com.mysql.jdbc │ com.mysql.jdbc.Driver              │ 5.1.34
mysql         │ com.mysql.jdbc.Driver              │
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

以上表可能会造成混淆，但如上所述，只有 pax-jdbc-database 捆绑包之一可以注册 org.osgi.service.jdbc.DataSourceFactory 实例，可以创建不只是委派至 java.sql.Driver.connect () 的 standard/XA/connection 池数据源。

以下示例创建并检查 MySQL 数据源：

```
karaf@root()> jdbc:ds-create -dt DataSource -dn mysql -url 'jdbc:mysql://localhost:3306/reportdb?
useSSL=false' -u fuse -p fuse mysqlds

karaf@root()> jdbc:ds-list
Name    | Product | Version | URL                                          | Status
--------+---------+---------+----------------------------------------------+-------
mysqlds | MySQL   | 5.7.21  | jdbc:mysql://localhost:3306/reportdb?useSSL=false | OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
date                | summary    | name   | details                 | id | email
--------------------+------------+--------+-------------------------+----+-----------------
2018-02-20 08:00:00.0 | Incident 1 | User 1 | This is a report incident 001 | 1  |
user1@redhat.com
2018-02-20 08:10:00.0 | Incident 2 | User 2 | This is a report incident 002 | 2  |
user2@redhat.com
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3  |
user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4  |
user4@redhat.com

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
FactoryPid:     org.ops4j.datasource
BundleLocation: mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   osgi.jdbc.driver.name = mysql
   password = fuse
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
   url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
   user = fuse
```

可以看到，将为您创建 org.ops4j.datasource factory PID。但是，它不会自动存储在 ${karaf.etc} 中，可以使用 config:update。

6.6. 使用加密配置值

pax-jdbc-config 功能能够处理加密值的配置管理配置。常用的解决方案是使用 Jasypt 加密服务，这些服务也供 Blueprint 使用。

如果任何 org.jasypt.encryption.StringEncryptor 服务在任何 alias service 属性中注册，您可以在数据源工厂 PID 和使用加密的密码中拒绝它。下面是一个示例：

```
felix.fileinstall.filename = */etc/org.ops4j.datasource-mysql.cfg
```

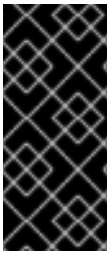
```
dataSourceName = mysqlds
dataSourceType = DataSource
decryptor = my-jasypt-decryptor
osgi.jdbc.driver.name = mysql
url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
user = fuse
password = ENC(<encrypted-password>)
```

用于查找解密器服务的服务过滤器是 （& (objectClass=org.jasypt.encryption.StringEncryptor) (alias=<alias>)，其中 < alias > 是数据源配置工厂 PID 的 decryptor 属性的值。

6.7. 使用 JDBC 连接池

本节介绍了使用 JDBC 连接池，然后演示如何使用这些连接池模块：

- pax-jdbc-pool-dbcp2
- pax-jdbc-pool-narayana
- pax-jdbc-pool-transx



重要

本章介绍了数据源管理内部的详细信息。虽然提供了关于 DBCP2 连接池功能的信息，但请记住，此连接池提供正确的 JTA 编码功能，但不提供 XA 恢复功能。

为确保 XA 恢复 已就位，请使用 pax-jdbc-pool-transx 或 pax-jdbc-pool-narayana 连接池模块。

6.7.1. 使用 JDBC 连接池简介

前面的示例演示了如何注册特定于数据库的数据源 工厂。由于 数据源 本身是连接的工厂，org.osgi.service.jdbc.DataSourceFactory 可能会被视为元工厂，它应该能够生成三种类型的数据源，以及作为 bonus，一个 java.sql.Driver:

- *javax.sql.DataSource*
- *javax.sql.ConnectionPoolDataSource*
- *javax.sql.XADataSource*

例如，pax-jdbc-mysql 注册一个 org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory，它会产生：

- *javax.sql.DataSource → com.mysql.jdbc.jdbc2.optional.MysqlDataSource*
- *javax.sql.ConnectionPoolDataSource → com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource*
- *javax.sql.XADataSource → com.mysql.jdbc.jdbc2.optional.MysqlXADataSource*
- *java.sql.Driver → com.mysql.jdbc.Driver*

PostgreSQL 驱动程序本身实施 OSGi JDBC 服务并生成：

- *javax.sql.DataSource → org.postgresql.jdbc2.optional.PoolingDataSource （如果指定了池相关的属性）或 org.postgresql.jdbc2.optional.SimpleDataSource*
- *javax.sql.ConnectionPoolDataSource → org.postgresql.jdbc2.optional.ConnectionPool*
- *javax.sql.XADataSource → org.postgresql.xa.PGXADataSource*
- *java.sql.Driver → org.postgresql.Driver*

如 规范数据源示例 所示，任何池、通用 数据源都要在 JTA 环境中工作，则需要一个数据库特定 数据源来实际获取(XA)连接。

我们已有后者，我们需要实际的、通用、可靠的连接池。

规范数据源示例 演示了如何配置特定数据库数据源的通用池。pax-jdbc-pool indices 捆绑包与上述 org.osgi.service.jdbc.DataSourceFactory 服务平稳工作。

正如 OSGI Enterprise R6 JDBC 规范提供了 org.osgi.service.jdbc.DataSourceFactory 标准接口，pax-jdbc-pool-common 提供专有 org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory 接口：

```
public interface PooledDataSourceFactory {

   javax.sql.DataSource create(org.osgi.service.jdbc.DataSourceFactory dsf, Properties config)

}
```

这个界面完全符合之前介绍的重要备注，值得重复：

重要

即使应用使用 XA 事务和/或连接池，应用与 javax.sql.DataSource 交互，而不是另外两个 JDBC 数据源接口。

这个界面只是从特定于数据库且非池数据中创建池数据源。或者更精确地，它是一个 数据源工厂 (meta factory)，它将特定于数据库数据源的工厂转换为池数据源的工厂。

注意

没有这样可防止应用使用 org.osgi.service.jdbc.DataSource 对象为 javax.sql.DataSource 对象配置池，该服务 已经为 javax.sql.DataSource 对象返回池。

下表显示了哪些捆绑包注册池数据源工厂。在表格中，o.o.p.j.p 的实例代表 org.ops4j.pax.jdbc.pool。

捆绑包（Bundle）	PooledDataSourceFactory	池密钥
pax-jdbc-pool-narayana	o.o.p.j.p.narayana.impl.Dbcp(XA)PooledDataSourceFactory	narayana
pax-jdbc-pool-dbcp2	o.o.p.j.p.dbcp2.impl.Dbcp(XA)PooledDataSourceFactory	dbcp2

捆绑包（Bundle）	PooledDataSourceFactory	池密钥
pax-jdbc-pool-transx	o.o.p.j.p.transx.impl.Transx(Xa)PooledDataSourceFactory	transx

以上捆绑包只安装数据源，而不是数据源本身。应用需要调用 javax.sql.DataSource create (org.osgi.service.jdbc.DataSourceFactory dsf, Properties config) 方法的内容。

6.7.2. 使用 dbcp2 连接池模块

有关通用数据源的部分提供了 如何使用和配置 Apache Commons DBCP 模块的示例。本节演示了如何在 Fuse OSGi 环境中执行此操作。

考虑 第 6.4.1 节 "PAX- JDBC 配置服务" 捆绑包。除了跟踪以下内容外：

- *org.osgi.service.jdbc.DataSourceFactory services*
- *org.ops4j.datasource factory PIDs*

该捆绑包还会跟踪 org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory 实例，它们由 pax-jdbc-pool reading 捆绑包之一注册。

如果 工厂配置包含 pool 属性，则 pax-jdbc-config 捆绑包注册的最终数据源是我们的数据库特定数据，但如果 pool=dbcp2 则嵌套在以下之一：

- *org.apache.commons.dbcp2.PoolingDataSource*
- *org.apache.commons.dbcp2.managed.ManagedDataSource*

这与 通用数据源示例 一致。除了 pool 属性和布尔值 xa 属性外，它选择非 xa 或 xa 数据源，org.ops4j.datasource factory PID 可能会包含 前缀 属性：

- *pool prerequisites*
- *factory prerequisites*

其中，每个属性都使用哪个 pax-jdbc-pool suppress 捆绑包被使用。对于 DBCP2，它是：

- *pool mdadm: bean 属性的 org.apache.commons.pool2.impl.GenericObjectPoolConfig (xa 和非 xa 场景)*
- *factoryadtrust: bean 属性是 org.apache.commons.dbcp2.managed.PoolableManagedConnectionFactory (xa) 或 org.apache.commons.dbcp2.PoolableConnectionFactory (non-xa)*

6.7.2.1. BasicDataSource 的配置属性

下表列出了 BasicDataSource 的通用配置属性。

参数	默认	描述

参数	默认	描述
username		要传递给 JDBC 驱动程序的连接用户名以建立连接。
password		要传递给 JDBC 驱动程序的连接密码以建立连接。
url		要传递给 JDBC 驱动程序的连接 URL 以建立连接。
driverClassName		要使用的 JDBC 驱动程序的完全限定 Java 类名称。
initialSize	0	池启动时创建的初始连接数。
maxTotal	8	可以同时从这个池分配的最大活跃连接数，或者没有限制的负数。
maxIdle	8	池中可以保持闲置的最大连接数，没有额外的连接被释放，或者没有限制。
minIdle	0	池中可以保持闲置的最小连接数量，而不创建额外的连接，或零来创建任何连接。
maxWaitMillis	无限期	池将等待的最大毫秒数（当没有可用连接时），在抛出异常前返回连接的最大毫秒数，或 -1 无限期等待。
validationQuery		用于从这个池验证连接的 SQL 查询，然后再将它们返回到调用者。如果指定，此查询必须是一个 SQL SELECT 语句，该语句至少返回一行。如果没有指定，通过调用 isValid （）方法来验证连接。
validationQueryTimeout	没有超时	连接验证查询失败前的超时时间（以秒为单位）。如果设置为正值，则此值将通过用于执行验证查询的声明的 setQueryTimeout 方法传递给驱动程序。
testOnCreate	false	指明对象在创建后是否会被验证。如果对象无法验证，则触发对象的浏览尝试将失败。
testOnBorrow	true	指明对象是否在从池中分离前验证对象。如果对象无法验证，它将从池中丢弃，我们将尝试浏览另一个对象。
testOnReturn	false	指明对象在返回到池之前是否会被验证。

参数	默认	描述
testWhileIdle	false	指明对象是否会被闲置对象驱除（若有）验证。如果对象无法验证，它将从池中丢弃。
timeBetweenEvictionRunsMillis	-1	运行空闲对象驱除器线程之间休眠的毫秒数。当非正数时，不会运行闲置对象驱除器线程。
numTestsPerEvictionRun	3	每次运行空闲对象驱除器线程期间要检查的对象数量（若有）。
minEvictableIdleTimeMillis	1000 * 60 * 30	对象在空闲对象驱除前可能处于空闲时间的最小时间（若有）。

6.7.2.2. 如何配置 DBCP2 池的示例

以下是一个现实示例（除了使用 SSL=false）配置 DBCP2 池(org.ops4j.datasource-mysql factory PID)，它使用 jdbc.-prefixed 属性的方便语法：

```
# Configuration for pax-jdbc-config to choose and configure specific
org.osgi.service.jdbc.DataSourceFactory
dataSourceName = mysqlds
dataSourceType = DataSource
osgi.jdbc.driver.name = mysql
jdbc.url = jdbc:mysql://localhost:3306/reportdb
jdbc.user = fuse
jdbc.password = fuse
jdbc.useSSL = false

# Hints for pax-jdbc-config to use org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
pool = dbcp2
xa = false

# dbcp2 specific configuration of org.apache.commons.pool2.impl.GenericObjectPoolConfig
pool.minIdle = 10
pool.maxTotal = 100
pool.initialSize = 8
pool.blockWhenExhausted = true
pool.maxWaitMillis = 2000
pool.testOnBorrow = true
pool.testWhileIdle = false
pool.timeBetweenEvictionRunsMillis = 120000
pool.evictionPolicyClassName = org.apache.commons.pool2.impl.DefaultEvictionPolicy

# dbcp2 specific configuration of org.apache.commons.dbcp2.PoolableConnectionFactory
factory.maxConnLifetimeMillis = 30000
factory.validationQuery  = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
```

在上面的配置中，pool 和 xa 键是提示（服务过滤器属性）来选择一个注册的 org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory 服务。对于 DBCP2，这是：

```
karaf@root()> feature:install pax-jdbc-pool-dbcp2

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.pool.dbcp2

OPS4J Pax JDBC Pooling DBCP2 (230) provides:
----------------------------------------
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 337
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 338
service.scope = singleton
xa = true
```

为了完整性，以下是一个完整的示例，其连接池配置添加到上例中。同样，这假设您是从全新的 Fuse 安装开始。

1. *安装 JDBC 驱动程序：*

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
```

2. *安装 jdbc、pax-jdbc-mysql 和 pax-jdbc-pool-dbcp2 功能：*

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install jdbc pax-jdbc-mysql pax-jdbc-pool-dbcp2

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 232
 service.id = 328
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (232)

karaf@root()> service:list org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
----------------------------------------------------------
 pool = dbcp2
```

```
 service.bundleid = 233
 service.id = 324
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)

[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
----------------------------------------------------------
 pool = dbcp2
 service.bundleid = 233
 service.id = 332
 service.scope = singleton
 xa = true
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)
```

3. *创建工厂配置：*

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:property-set pool dbcp2
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.minIdle 2
karaf@root()> config:property-set pool.maxTotal 10
karaf@root()> config:property-set pool.blockWhenExhausted true
karaf@root()> config:property-set pool.maxWaitMillis 2000
karaf@root()> config:property-set pool.testOnBorrow true
karaf@root()> config:property-set pool.testWhileIdle alse
karaf@root()> config:property-set pool.timeBetweenEvictionRunsMillis 120000
karaf@root()> config:property-set factory.validationQuery 'select schema_name from
information_schema.schemata'
karaf@root()> config:property-set factory.validationQueryTimeout 2
karaf@root()> config:update
```

4. *检查 **pax-jdbc-config** 是否将配置处理到 **javax.sql.DataSource** 服务中：*

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
```

```
osgi.jdbc.driver.name = mysql
osgi.jndi.service.name = mysqlds
pax.jdbc.managed = true
pool.blockWhenExhausted = true
pool.maxTotal = 10
pool.maxWaitMillis = 2000
pool.minIdle = 2
pool.testOnBorrow = true
pool.testWhileIdle = alse
pool.timeBetweenEvictionRunsMillis = 120000
service.bundleid = 225
service.factoryPid = org.ops4j.datasource
service.id = 338
service.pid = org.ops4j.datasource.fd7aa3a1-695b-4342-b0d6-23d018a46fbb
service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (225)
```

5. ***使用数据源：***

```
karaf@root()> jdbc:query mysqlds 'select * from incident'
date                  | summary    | name   | details                    | id | email
----------------------+------------+--------+----------------------------+----+-----------------
2018-02-20 08:00:00.0 | Incident 1 | User 1 | This is a report incident 001 | 1  | user1@redhat.com
2018-02-20 08:10:00.0 | Incident 2 | User 2 | This is a report incident 002 | 2  | user2@redhat.com
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3  | user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4  | user4@redhat.com
```

6.7.3. 使用 **narayana** 连接池模块

pax-jdbc-pool-narayna 模块几乎执行所有操作为 pax-jdbc-pool-dbcp2。它为 XA 和非 XA 场景安装 DBCP2 特定的 org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory。唯一的区别在于，XA 场景中有一个额外的集成点。org.jboss.tm.XAResourceRecovery OSGi 服务被注册为由 com.arjuna.ats.arjuna.recovery.RecoveryManager （这是 Narayana 事务管理器的一部分）。

6.7.4. 使用 **transx** 连接池模块

pax-jdbc-pool-transx 捆绑包基础，它对 pax-transx-jdbc 捆绑包上的 org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory 服务实施。pax-transx-jdbc 捆绑包使用 org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder 工具创建 javax.sql.DataSource 池。这是一个 JCA (Java™ Connector Architecture)解决方案，稍后 进行介绍。

6.8. 将数据源部署为工件

本章介绍了 OSGi JDBC 服务，显示 pax-jdbc 捆绑包帮助注册数据库特定和通用数据源，以及如何从 OSGi 服务和配置管理配置的角度来看。虽然配置 两种类型的数据源可以通过使用 Configuration Admin factory PID （来自 pax-jdbc-config 捆绑包的帮助）来完成，但通常最好使用 部署方法。

在部署方法中，javax.sql.DataSource 服务由应用程序代码直接注册，通常是在 Blueprint 容器中注册。蓝图 XML 可以是普通 OSGi 捆绑包的一部分，可通过使用 mvn: URI 进行安装，并存储在 Maven 存储库（本地或远程）。通过将捆绑包与 Configuration Admin 配置进行比较，更易于版本控制。

pax-jdbc-config 捆绑包版本 1.3.0 为数据源配置添加部署方法。应用程序开发人员注册 javax.sql. (XA) DataSource 服务（通常使用 Bluerpint XML）并指定服务属性。pax-jdbc-config 捆绑包检测到此类注册的数据库特定数据源，以及（使用服务属性）将服务嵌套在通用、非特定于数据库的连接池中。

为了完整起见，以下是使用 Blueprint XML 的三种部署方法。Fuse 提供了快速入门下载，其中包含 Fuse 的不同方面的不同示例。您可以从 Fuse Software Downloads 页面下载快速入门 zip 文件。

将快速入门 zip 文件的内容提取到本地文件夹。

在以下示例中，quickstarts/persistence 目录被称为 $PQ_HOME。

- 第 6.8.1 节 “手动部署数据源”
- 第 6.8.2 节 “数据源的工厂部署”
- 第 6.8.3 节 “数据源的混合部署”

6.8.1. 手动部署数据源

这个示例手动部署数据源使用基于 docker 的 PostgreSQL 安装。在此方法中，不需要 pax-jdbc-config。应用程序代码负责注册特定于数据库和通用数据源。

需要这三个捆绑包：

- mvn:org.postgresql/postgresql/42.2.5
- mvn:org.apache.commons/commons-pool2/2.5.0
- mvn:org.apache.commons/commons-dbcp2/2.1.1

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Fuse/Karaf exports this service from fuse-pax-transx-tm-narayana bundle
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non database-specific, generic, pooling, enlisting javax.sql.DataSource
-->
<bean id="pool" class="org.apache.commons.dbcp2.managed.BasicManagedDataSource">
   <property name="xaDataSourceInstance" ref="postgresql" />
   <property name="transactionManager" ref="tm" />
```

```
    <property name="minIdle" value="3" />
    <property name="maxTotal" value="10" />
    <property name="validationQuery" value="select schema_name, schema_owner from
information_schema.schemata" />
</bean>

<!--
    Expose datasource to use by application code (like Camel, Spring, ...)
-->
<service interface="javax.sql.DataSource" ref="pool">
    <service-properties>
        <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
    </service-properties>
</service>
```

以上蓝图 XML 片段与 规范数据源示例 匹配。以下是显示如何使用它的 shell 命令：

```
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 233
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 224
karaf@root()> install -s mvn:org.apache.commons/commons-dbcp2/2.1.1
Bundle ID: 225
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-manual.xml
Bundle ID: 226

karaf@root()> bundle:services -p 226

Bundle 226 provides:
--------------------
objectClass = [javax.sql.DataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 226
service.id = 242
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 226
service.id = 243
service.scope = singleton

karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-list
Name          | Product    | Version                | URL
| Status
--------------+------------+------------------------+-------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
```

```
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────┼───────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding(
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                │ summary   │ name   │ details                │ id │ email
───────────────────────────────┼──────────────┼─────────┼──────────────────
────────────┼─────┼──────────────────
2018-02-20 08:00:00 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
2018-02-20 08:10:00 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │ user3@redhat.com
2018-02-20 08:30:00 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │ user4@redhat.com
```

如上表所示，Blueprint 捆绑包导出 **javax.sql.DataSource** 服务，该服务是一个通用的、非特定于数据库的连接池。特定于数据库的 **javax.sql.XADataSource** 捆绑包没有 作为 OSGi 服务注册，因为 Blueprint XML 没有明确的 **< service ref="postgresql">** 声明。

6.8.2. 数据源的工厂部署

数据源的工厂部署以规范的方式使用 **pax-jdbc-config** 捆绑包。这与 Fuse 6.x 中推荐的方法有点不同，该方法需要指定池配置作为服务属性。

以下是 Blueprint XML 示例：

```
<!--
   A database-specific org.osgi.service.jdbc.DataSourceFactory that can create
DataSource/XADataSource/
   /ConnectionPoolDataSource/Driver using properties. It is registered by pax-jdbc-* or for
example
   mvn:org.postgresql/postgresql/42.2.5 bundle natively.
-->
<reference id="dataSourceFactory"
        interface="org.osgi.service.jdbc.DataSourceFactory"
        filter="(osgi.jdbc.driver.class=org.postgresql.Driver)" />

<!--
   Non database-specific org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory that can
create
   pooled data sources using some org.osgi.service.jdbc.DataSourceFactory. dbcp2 pool is
registered
```

```
    by pax-jdbc-pool-dbcp2 bundle.
-->
<reference id="pooledDataSourceFactory"
        interface="org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory"
        filter="(&amp;(pool=dbcp2)(xa=true))" />

<!--
    Finally, use both factories to expose pooled, xa-aware data source.
-->
<bean id="pool" factory-ref="pooledDataSourceFactory" factory-method="create">
    <argument ref="dataSourceFactory" />
    <argument>
        <props>
            <!--
                Properties needed by postgresql-specific org.osgi.service.jdbc.DataSourceFactory.
                Cannot prepend them with 'jdbc.' prefix as the DataSourceFactory is implemented
directly
                by PostgreSQL driver, not by pax-jdbc-* bundle.
            -->
            <prop key="url" value="jdbc:postgresql://localhost:5432/reportdb" />
            <prop key="user" value="fuse" />
            <prop key="password" value="fuse" />
            <prop key="currentSchema" value="report" />
            <prop key="connectTimeout" value="5" />
            <!-- Properties needed by dbcp2-specific
org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory -->
            <prop key="pool.minIdle" value="2" />
            <prop key="pool.maxTotal" value="10" />
            <prop key="pool.blockWhenExhausted" value="true" />
            <prop key="pool.maxWaitMillis" value="2000" />
            <prop key="pool.testOnBorrow" value="true" />
            <prop key="pool.testWhileIdle" value="false" />
            <prop key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
            <prop key="factory.validationQueryTimeout" value="2" />
        </props>
    </argument>
</bean>

<!--
    Expose data source for use by application code (such as  Camel, Spring, ...).
-->
<service interface="javax.sql.DataSource" ref="pool">
    <service-properties>
        <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
    </service-properties>
</service>
```

本例使用 工厂 Bean 使用数据源创建数据源。您不需要显式引用 javax.transaction.TransactionManager 服务，因为这由 XA 感知的 PooledDataSourceFactory 在内部进行跟踪。

以下是相同的示例，但在 Fuse/Karaf shell 中。

注意

要让原生 org.osgi.service.jdbc.DataSourcFactory 捆绑包注册，请安装 mvn:org.osgi/org.osgi.service.jdbc/1.0.0，然后安装* PostgreSQL *驱动程序。

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-factory-dbcp2.xml
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
objectClass = [javax.sql.DataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-factory-dbcp2.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 337
service.scope = singleton

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                       │ URL
│ Status
────────────────┼────────────┼───────────────────────────────┼──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┼─────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&databaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&binaryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&loginTimeout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding(
```

```
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false | OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                | summary    | name   | details                      | id | email
--------------------+------------+--------+------------------------------+----+-----------------
2018-02-20 08:00:00 | Incident 1 | User 1 | This is a report incident 001 | 1  | user1@redhat.com
2018-02-20 08:10:00 | Incident 2 | User 2 | This is a report incident 002 | 2  | user2@redhat.com
2018-02-20 08:20:00 | Incident 3 | User 3 | This is a report incident 003 | 3  | user3@redhat.com
2018-02-20 08:30:00 | Incident 4 | User 4 | This is a report incident 004 | 4  | user4@redhat.com
```

如上表所示，Blueprint 捆绑包导出 javax.sql.DataSource 服务，该服务是一个通用的、非特定于数据库的连接池。特定于数据库的 javax.sql.XADataSource 没有 作为 OSGi 服务注册，因为 Blueprint XML 没有明确的 < service ref="postgresql"> 声明。

6.8.3. 数据源的混合部署

在数据源的混合部署中，pax-jdbc-config 1.3.0 捆绑包使用服务属性 在池数据源中打包特定于数据库的数据源的另一个方法。这个方法与 Fuse 6.x 中的工作方式匹配。

以下是 Blueprint XML 示例：

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Expose database-specific data source with service properties.
   No need to expose pooling, enlisting, non database-specific javax.sql.DataSource. It is
registered
   automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking.
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
   <service-properties>
      <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source
inside connection pool -->
      <entry key="pool" value="dbcp2" />
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
      <!-- Other properties that configure given connection pool, as indicated by pool=dbcp2 --
>
      <entry key="pool.minIdle" value="2" />
      <entry key="pool.maxTotal" value="10" />
      <entry key="pool.blockWhenExhausted" value="true" />
      <entry key="pool.maxWaitMillis" value="2000" />
      <entry key="pool.testOnBorrow" value="true" />
      <entry key="pool.testWhileIdle" value="false" />
```

```
        <entry key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
        <entry key="factory.validationQueryTimeout" value="2" />
    </service-properties>
</service>
```

在上例中，只会手动注册特定于数据库的数据源。pool=dbcp2 服务属性是数据源跟踪器的提示，它由 pax-jdbc-config 捆绑包管理。具有这个 service 属性的数据源服务嵌套在池数据源中，本例中为 pax-jdbc-pool-dbcp2。

以下是 Fuse/Karaf shell 中的相同示例：

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-
discovery.xml
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
factory.validationQuery = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
objectClass = [javax.sql.XADataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = postgresql
pool = dbcp2
pool.blockWhenExhausted = true
pool.maxTotal = 10
pool.maxWaitMillis = 2000
pool.minIdle = 2
pool.testOnBorrow = true
pool.testWhileIdle = false
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-discovery.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 338
service.scope = singleton

karaf@root()> service:list javax.sql.XADataSource
[javax.sql.XADataSource]
------------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
 osgi.service.blueprint.compname = postgresql
 pool = dbcp2
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
```

```
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 233
 service.id = 336
 service.scope = bundle
Provided by :
 Bundle 233
Used by:
 OPS4J Pax JDBC Config (224)

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
 osgi.service.blueprint.compname = postgresql
 pax.jdbc.managed = true
 pax.jdbc.service.id.ref = 336
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 224
 service.id = 337
 service.ranking = 1000
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (224)

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                    │ URL
 │ Status
────────────────────────┼──────────────────┼───────────────────────────────────────────────┼──
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────┼──────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&bi
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
```

```
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncodingC
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckSe
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false | OK
jdbc/postgresql | PostgreSQL | 10.3 (Debian 10.3-1.pgdg90+1) |
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncodingC
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckSe
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false | OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident'
date                | summary    | name   | details                    | id | email
--------------------+------------+--------+----------------------------+----+-----------------
2018-02-20 08:00:00 | Incident 1 | User 1 | This is a report incident 001 | 1  | user1@redhat.com
2018-02-20 08:10:00 | Incident 2 | User 2 | This is a report incident 002 | 2  | user2@redhat.com
2018-02-20 08:20:00 | Incident 3 | User 3 | This is a report incident 003 | 3  | user3@redhat.com
2018-02-20 08:30:00 | Incident 4 | User 4 | This is a report incident 004 | 4  | user4@redhat.com
```

如此列表中所示，jdbc:ds-list 输出中有两个数据源，即原始数据源和打包程序数据源。

javax.sql.XADataSource 从 Blueprint 捆绑包注册，并声明了 pool = dbcp2 属性。

javax.sql.DataSource 从 pax-jdbc-config 捆绑包中注册，并：

- **没有 pool = dbcp2 属性（注册打包程序数据源时已被删除）。**
- **具有 service.ranking = 1000 属性，因此当它始终是根据名称查找数据源时的首选版本。**
- **具有 pax.jdbc.managed = true 属性，因此不会尝试再次打包它。**
- **具有 pax.jdbc.service.id.ref = 336 属性，以指示嵌套在连接池中的原始数据源服务。**

6.9. 将数据源与 JAVA™ 持久性 API 搭配使用

从事务管理的角度来看，了解数据源如何与 Java™ Persistence API (uildDefaults)搭配使用。本小节并不描述了 JPA 规范本身的详细信息，也没有有关 Hibernate 的详细信息，这是最已知的 JPA 实施。相反，本节演示了如何将 JPA 持久单位指向数据源。

6.9.1. 关于数据源引用

META-INF/persistence.xml 描述符（请参阅 JPA 2.1 规格8.2.1.5 jta-data-source, non-jta-data-source）定义了两种数据源引用：

- **<JTA-data-source> - 这是对支持 JTA 事务的JTA 数据源的 JNDI 引用。**
- **<non-jta-data-source> - 这是对支持 JTA 事务外的JTA 数据源的引用。此数据源通常也用于初始化阶段，例如，使用 hibernate.hbm2ddl.auto 属性将 Hibernate 配置为自动创建数据库 schema。**

这两个数据源与 javax.sql.DataSource 或 javax.sql.XADataSource! 这是开发 JPA 应用程序时常见的误解。这两个 JNDI 名称都必须引用 JNDI-bound javax.sql.DataSource 服务。

6.9.2. 引用 JNDI 名称

当您注册带有 osgi.jndi.service.name 属性的 OSGi 服务时，它会在 OSGi JNDI 服务中绑定。在 OSGi 运行时（如 Fuse/Karaf）中，JNDI 不是 name → value 对的简单字典。在 OSGi 中通过 JNDI 名称引用对象涉及服务查找和其他更为复杂的 OSGi 机制，如服务 hook。

在全新的 Fuse 安装中，以下列表演示了如何在 JNDI 中注册数据源：

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set osgi.jndi.service.name jdbc/mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> feature:install jndi

karaf@root()> jndi:names
JNDI Name                 | Class Name
--------------------------+-------------------------------------------------------------------
osgi:service/jndi         | org.apache.karaf.jndi.internal.JndiServiceImpl
osgi:service/jdbc/mysqlds | com.mysql.jdbc.jdbc2.optional.MysqlDataSource
```

正如您所见，数据源位于 osgi:service/jdbc/mysqlds JNDI 名称下。

但是，如果将 JPA 在 OSGi 中，您必须使用 完整的 JNDI 名称。以下是指定数据源引用的 META-INF/persistence.xml 片段示例：

```
<jta-data-source>
   osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</jta-data-source>
<non-jta-data-source>
   osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</non-jta-data-source>
```

如果没有上述配置，您可能会遇到这个错误：

> *Persistence unit "pu-name" refers to a non OSGi service DataSource*

第 7 章 使用 JMS 连接工厂

本章论述了如何在 OSGi 中使用 JMS 连接工厂。从根本上，您使用以下方法实现它：

```
org.osgi.framework.BundleContext.registerService(javax.jms.ConnectionFactory.class,
                                                 connectionFactoryObject,
                                                 properties);
org.osgi.framework.BundleContext.registerService(javax.jms.XAConnectionFactory.class,
                                                 xaConnectionFactoryObject,
                                                 properties);
```

注册此类服务的方法有两种：

- 使用 jms:create Karaf console 命令发布连接工厂。这是配置方法。
- 使用 Blueprint、SOSOS Declarative Services (SCR)或仅 BundleContext.registerService （） API 调用等方法发布连接工厂。这个方法需要一个包含代码和/或元数据的专用 OSGi 捆绑包。这是 部署方法。

详情包括在以下主题中：

- 第 7.1 节 “关于 OSGi JMS 服务”
- 第 7.2 节 “关于 PAX-JMS 配置服务”
- 第 7.3 节 “使用 JMS 控制台命令”
- 第 7.4 节 “使用加密配置值”
- 第 7.5 节 “使用 JMS 连接池”
- 第 7.6 节 “以工件的形式部署连接工厂”

7.1. 关于 OSGI JMS 服务

处理 JDBC 数据源的 OSGi 方法与两个接口相关：

- standard org.osgi.service.jdbc.DataSourceFactory
- proprietary org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory

对于 JMS，请考虑以下几点：

- 专有 org.ops4j.pax.jms.service.ConnectionFactoryFactory 与标准 OSGi JDBC org.osgi.service.jdbc.DataSourceFactory相同
- 专有 org.ops4j.pax.jms.service.PooledConnectionFactoryFactory Factory 与专有 pax-jdbc org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory相同

对于专用的、特定于代理的 org.ops4j.pax.jms.service.ConnectionFactoryFactory 实现，有如下捆绑包：

- mvn:org.ops4j.pax.jms/pax-jms-artemis/1.0.0
- mvn:org.ops4j.pax.jms/pax-jms-ibmmq/1.0.0

- *mvn:org.ops4j.pax.jms/pax-jms-activemq/1.0.0*

这些捆绑包注册特定于代理的 org.ops4j.pax.jms.service.ConnectionFactoryFactory 服务，它可以返回 javax.jms.ConnectionFactory 和 javax.jms.XAConnectionFactory 等 JMS 工厂。例如：

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
----------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
-------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

7.2. 关于 PAX-JMS 配置服务

mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0 捆绑包提供了一个 Managed Service Factory，它执行三个操作：

- *跟踪 org.ops4j.pax.jms.service.ConnectionFactoryFactory OSGi 服务，以调用其方法：*

```
public ConnectionFactory createConnectionFactory(Map<String, Object> properties);

public XAConnectionFactory createXAConnectionFactory(Map<String, Object>
properties);
```

- *跟踪 org.ops4j.connectionfactory factory PIDs，以收集以上方法所需的属性。如果您使用可用于 Configuration Admin 服务的任何方法创建 工厂配置，例如，通过创建一个 ${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg 文件，您可以执行最后一步来公开特定于代理的连接工厂。*

- *跟踪 javax.jms.ConnectionFactory 和 javax.jms.XAConnectionFactory 服务，将它们嵌套在 JMS 连接工厂内。*

详情包括在以下主题中：

- *第 7.2.1 节 “为 AMQ 7.1 创建连接工厂”*
- *第 7.2.2 节 “为 IBM MQ 8 或 IBM MQ 9 创建连接工厂”*
- *第 7.2.4 节 “处理的属性摘要”*

7.2.1. 为 AMQ 7.1 创建连接工厂

以下是为 Artemis 代理创建连接因素的详细 规范、逐步指南。

1. **使用 pax-jms-artemis 功能和 pax-jms-config 功能安装 Artemis 驱动程序：**

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
---------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

2. **创建 工厂配置 ：**

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis # "name" property may
be used too
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or
XAConnectionFactory
karaf@root()> config:property-set jms.url tcp://localhost:61616
karaf@root()> config:property-set jms.user admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set jms.consumerMaxRate 1234
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
  connectionFactoryType = ConnectionFactory
  felix.fileinstall.filename = file:${karar.etc}/org.ops4j.connectionfactory-artemis.cfg
  jms.consumerMaxRate = 1234
  jms.password = admin
  jms.url = tcp://localhost:61616
  jms.user = admin
  osgi.jndi.service.name = jms/artemis
```

```
  service.factoryPid = org.ops4j.connectionfactory
  service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
  type = artemis
```

3. *检查 **pax-jms-config 是否处理到 javax.jms.ConnectionFactory 服务中的配置：***

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
 jms.consumerMaxRate = 1234
 jms.password = admin
 jms.url = tcp://localhost:61616
 jms.user = admin
 osgi.jndi.service.name = jms/artemis
 pax.jms.managed = true
 service.bundleid = 248
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 342
 service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Config (248)
```

注意

如果您指定了额外的 Artemis 配置，特别是 protocol=amqp，则将使用 QPID JMS 库而不是 Artemis JMS 客户端。还必须将 amqp:// 协议用于 jms.url 属性。

4. ***测试连接。***

现在，您有一个特定于代理（还没有池）的连接工厂，您可以在需要时注入。例如，您可以使用 jms 功能中的 Karaf 命令：

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property | Value
---------+-----------------------------------------
product  | ActiveMQ
version  | 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                          | Content      | Charset | Type | Correlation ID | Delivery Mode |
```

```
Destination          | Expiration | Priority | Redelivered | ReplyTo | Timestamp
---------------------+------------+----------+-------------+---------+-----------------------------
ID:2b6ea56d-574d-11e8-971a-7ee9ecc029d4 | Hello Artemis | UTF-8 |  |  | Persistent
| ActiveMQQueue[DEV.QUEUE.1] | Never | 4 | false |  | Mon May 14 10:02:38
CEST 2018
```

以下列表显示了在切换协议时会发生什么：

```
karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
   jms.consumerMaxRate = 1234
   jms.password = fuse
   jms.url = tcp://localhost:61616
   jms.user = fuse
   osgi.jndi.service.name = jms/artemis
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
   type = artemis

karaf@root()> config:edit org.ops4j.connectionfactory.312eb09a-d686-4229-b7e1-2ea38a77bb0f
karaf@root()> config:property-set protocol amqp
karaf@root()> config:property-delete user
karaf@root()> config:property-set username admin # mind the difference between artemis-jms-client
and qpid-jms-client
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:update

karaf@root()> jms:info -u admin -p admin jms/artemis
Property | Value
---------+-----------------------
product  | QpidJMS
version  | 0.30.0.redhat-1

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID | Content | Charset | Type | Correlation ID | Delivery Mode | Destination |
Expiration | Priority | Redelivered | ReplyTo | Timestamp
-----------+---------+---------+------+----------------+---------------+-------------+
           | Hello Artemis | UTF-8 |  |  | Persistent | DEV.QUEUE.1 | Never | 4
| false |  | Mon May 14 10:02:38 CEST 2018
```

7.2.2. 为 IBM MQ 8 或 IBM MQ 9 创建连接工厂

本节介绍如何连接到 IBM MQ 8 和 IBM MQ 9。虽然 pax-jms-ibmmq 安装相关的 pax-jms 捆绑包，但由于许可原因，IBM MQ 驱动程序不会被安装。

1. *进入 https://developer.ibm.com/messaging/mq-downloads/*
2. *登录。*
3. *点您要安装的版本，例如，点 IBM MQ 8.0 Client 或 IBM MQ 9.0 Client。*
4. *在出现的页面中，在底部，在下载版本表中点击您想要的版本。*
5. *在下一页中，选择具有 IBM-MQ-Install-Java-All 后缀的最新版本。例如，下载 8.0.0.10-WS-MQ-Install-Java-All 或 9.0.0.4-IBM-MQ-Install-Java-All。*
6. *提取下载的 JAR 文件的内容。*
7. *执行 bundle:install 命令。例如，如果您将内容提取到 /home/Downloads 目录中，则输入如下命令：*

```
`bundle:install -s wrap:file:////home/Downloads/9.0.0.4-IBM-MQ-Install-Java-All/ibmmq9/wmq/JavaSE/com.ibm.mq.allclient.jar`.
```

8. *按如下所示创建连接工厂：*

   a. *安装 pax-jms-ibmmq ：*

```
karaf@root()> feature:install pax-jms-ibmmq

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-ibmmq

OPS4J Pax JMS IBM MQ Support (239) provides:
----------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 239
service.id = 346
service.scope = singleton
type = ibmmq
```

   b. *创建 工厂配置 ：*

```
karaf@root()> config:edit --factory --alias ibmmq org.ops4j.connectionfactory
karaf@root()> config:property-set type ibmmq
karaf@root()> config:property-set osgi.jndi.service.name jms/mq9 # "name" property may be used too
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or XAConnectionFactory
karaf@root()> config:property-set jms.queueManager FUSEQM
karaf@root()> config:property-set jms.hostName localhost
karaf@root()> config:property-set jms.port 1414
karaf@root()> config:property-set jms.transportType 1 # com.ibm.msg.client.wmq.WMQConstants.WMQ_CM_CLIENT
karaf@root()> config:property-set jms.channel DEV.APP.SVRCONN
karaf@root()> config:property-set jms.CCSID 1208 # com.ibm.msg.client.jms.JmsConstants.CCSID_UTF8
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
```

```
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-ibmmq.cfg
   jms.CCSID = 1208
   jms.channel = DEV.APP.SVRCONN
   jms.hostName = localhost
   jms.port = 1414
   jms.queueManager = FUSEQM
   jms.transportType = 1
   osgi.jndi.service.name = jms/mq9
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
   type = ibmmq
```

c. *检查 **pax-jms-config 是否处理到 javax.jms.ConnectionFactory 服务中的配置：***

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
------------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:/data/servers/7.13.0.fuse-7_13_0-00012-redhat-
00001/etc/org.ops4j.connectionfactory-ibmmq.cfg
 jms.CCSID = 1208
 jms.channel = DEV.APP.SVRCONN
 jms.hostName = localhost
 jms.port = 1414
 jms.queueManager = FUSEQM
 jms.transportType = 1
 osgi.jndi.service.name = jms/mq9
 pax.jms.managed = true
 service.bundleid = 237
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 347
 service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
 service.scope = singleton
 type = ibmmq
Provided by :
 OPS4J Pax JMS Config (237)
```

d. ***测试连接：***

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/mq9

karaf@root()> jms:info -u app -p fuse jms/mq9
Property │ Value
─────────┼──────────────────────────────
product  │ IBM MQ JMS Provider
```

```
version │ 8.0.0.0

karaf@root()> jms:send -u app -p fuse jms/mq9 DEV.QUEUE.1 "Hello IBM MQ 9"

karaf@root()> jms:browse -u app -p fuse jms/mq9 DEV.QUEUE.1
Message ID                                │ Content          │ Charset │ Type │
Correlation ID │ Delivery Mode │ Destination        │ Expiration │ Priority │ Redelivered
│ ReplyTo │ Timestamp
──────────────────────────────────────────┼──────────────────┼─────────┼──────┼
──────────────┼───────────────┼────────────────────┼────────────┼──────────┼────────────
─────────┼──────────────────────────
ID:414d512046555345514d202020202020c940f95a038b3220 │ Hello IBM MQ 9
│ UTF-8   │         │                  │ Persistent   │ queue:///DEV.QUEUE.1 │ Never      │ 4
│ false      │         │ Mon May 14 10:17:01 CEST 2018
```

您还可以检查消息是否从 IBM MQ Explorer 发送或从 Web 控制台发送。

7.2.3. 在 Apache Karaf 上的 Fuse 中使用 JBoss A-MQ 6.3 客户端

您可以从 Fuse Software Downloads 页面下载 Fuse 快速入门。

将快速启动 zip 文件的内容提取到本地文件夹，例如一个名为 quickstarts 的文件夹。

您可以构建并安装 quickstarts/camel/camel-jms 示例作为 OSGi 捆绑包。此捆绑包包含 Camel 路由的蓝图 XML 定义，用于将消息发送到 JBoss A-MQ 6.3 JMS 队列。为 JBoss A-MQ 6.3 代理创建连接工厂的步骤如下：

7.2.3.1. 先决条件

- **已安装 Maven 3.3.1 或更高版本。**
- **您已在机器上安装了 Red Hat Fuse。**
- **您已在计算机上安装了 JBoss A-MQ Broker 6.3。**
- **您已从客户门户网站下载并解压缩了 Karaf quickstarts zip 文件中的 Fuse。**

7.2.3.2. 流程

1. **导航到 quickstarts/camel/camel-jms/src/main/resources/OSGI-INF/blueprint/ 目录。**
2. **将以下 bean 替换为 camel-context.xml 文件中的 id="jms" ：**

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <reference interface="javax.jms.ConnectionFactory" />
    </property>
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

使用以下部分来实例化 JBoss A-MQ 6.3 连接工厂：

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
```

```
        <property name="connectionFactory" ref="activemqConnectionFactory"/>
        <property name="transactionManager" ref="transactionManager"/>
 </bean>
 <bean id="activemqConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
  <property name="userName" value="admin"/>
  <property name="password" value="admin"/>
 </bean>
```

JBoss A-MQ 6.3 连接工厂配置为连接到在 tcp://localhost:61616 侦听的代理。默认情况下，JBoss A-MQ 使用 IP 端口值 61616。连接工厂也配置为使用 userName/password 凭证 admin/admin。确保此用户已在代理协调中启用（或者您可以在此处自定义这些设置以匹配代理配置）。

3. 保存 camel-context.xml 文件。

4. 构建 camel-jms 快速启动：

```
$ cd quickstarts/camel/camel-jms
$ mvn install
```

5. 成功安装 Quickstart 后，进入到 $FUSE_HOME/ 目录，并运行以下命令来在 Apache Karaf 服务器上启动 Fuse：

```
$ ./bin/fuse
```

6. 在 Apache Karaf 实例的 Fuse 上，安装 activemq-client 功能和 camel-jms 功能：

```
karaf@root()> feature:install activemq-client
karaf@root()> feature:install camel-jms
```

7. 安装 camel-jms quickstart 捆绑包：

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/{$fuseversion}
```

其中，将 {$fuseversion} 替换为您刚才构建的 Maven 工件的实际版本(consult the camel-jms quickstart README 文件)。

8. 启动 JBoss A-MQ 6.3 代理（您需要安装 JBoss A-MQ 6.3）。打开另一个终端窗口，再导航到 JBOSS_AMQ_63_INSTALLDIR 目录：

```
$ cd JBOSS_AMQ_63_INSTALLDIR
$ ./bin/amq
```

9. Camel 路由启动后，您可以在 Fuse 安装中看到目录 work/jms/input。将您在这个 quickstart 的 src/main/data 目录中找到的文件复制到新创建的 work/jms/input 目录中。

10. 稍等片刻，您将在 work/jms/output 目录下找到国家按国家组织相同的文件：

```
order1.xml, order2.xml and order4.xml in work/jms/output/others
order3.xml and order5.xml in work/jms/output/us
order6.xml in work/jms/output/fr
```

11. 使用 log:display 检查业务日志记录：

```
Receiving order order1.xml

Sending order order1.xml to another country

Done processing order1.xml
```

7.2.4. 处理的属性摘要

Configuration Admin factory PID 中的属性传递到相关的 org.ops4j.pax.jms.service.ConnectionFactoryFactory 实现。

- ActiveMQ
  org.ops4j.pax.jms.activemq.ActiveMQConnectionFactoryFactory （仅限JMS 1.1）

  传递给 org.apache.activemq.ActiveMQConnectionFactory.buildFromMap （） 方法的属性

- artemis
  org.ops4j.pax.jms.artemis.ArtemisConnectionFactoryFactory

  如果 protocol=amqp，属性将传递到 org.apache.qpid.jms.util.util.PropertyUtil.setProperties （） 方法，以配置 org.apache.qpid.jms.JmsConnectionFactory 实例。

  否则，org.apache.activemq.artemis.utils.uri.BeanSupport.setData （） 为 org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory 实例调用。

- IBM MQ
  org.ops4j.pax.jms.ibmmq.MQConnectionFactoryFactory

  处理 com.ibm.mq.jms.MQConnectionFactory 或 com.ibm.mq.jms.MQXAConnectionFactory 的 bean 属性。

7.3. 使用 JMS 控制台命令

Apache Karaf 提供 jms 功能，它包括 jms:* 范围内的 shell 命令。您已看到一些使用这些命令检查手动配置的连接工厂的示例。另外，还有一些命令隐藏了创建配置管理员配置的需求。

从全新的 Fuse 实例开始，您可以注册特定于代理的连接工厂。以下列表显示了从 Karaf 安装 jms 功能，以及从 pax-jms -jms-jms 安装pax-jms-artemis 的 jms 功能：

```
karaf@root()> feature:install jms pax-jms-artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------------

karaf@root()> service:list javax.jms.ConnectionFactory # should be empty

karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
----------------------------------------------------
 service.bundleid = 250
 service.id = 326
 service.scope = singleton
```

```
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (250)
```

下表显示了如何创建和检查 Artemis 连接工厂：

```
karaf@root()> jms:create -t artemis -u admin -p admin --url tcp://localhost:61616 artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property | Value
---------+-----------------------------------------
product  | ActiveMQ
version  | 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                           | Content       | Charset | Type | Correlation ID | Delivery Mode |
Destination              | Expiration | Priority | Redelivered | ReplyTo | Timestamp
-------------------------------------+---------------+---------+------+----------------+---------------
-------------------------+------------+----------+-------------+---------+--------------------------
ID:7a944470-574f-11e8-918e-7ee9ecc029d4 | Hello Artemis | UTF-8   |      |                | Persistent
| ActiveMQQueue[DEV.QUEUE.1] | Never      | 4        | false       |         | Mon May 14 10:19:10
CEST 2018

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0
Properties:
   name = artemis
   osgi.jndi.service.name = jms/artemis
   password = admin
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
   type = artemis
   url = tcp://localhost:61616
   user = admin
```

正如您所见，将为您创建 org.ops4j.connectionfactory factory PID。但是，它不会自动存储在 ${karaf.etc} 中，可以使用 config:update。无法指定其他属性，但您可以在稍后添加它们。

7.4. 使用加密配置值

与 pax-jdbc-config 捆绑包一样，您可以使用 Jasypt 来加密属性。

如果有任何 org.jasypt.encryption.StringEncryptor 服务在任何 alias service 属性中注册，您可以在连接工厂 PID 中引用它，并使用加密的密码。以下是一个示例：

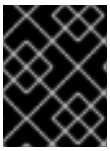
```
felix.fileinstall.filename = */etc/org.ops4j.connectionfactory-artemis.cfg
name = artemis
type = artemis
decryptor = my-jasypt-decryptor
url = tcp://localhost:61616
user = fuse
password = ENC(<encrypted-password>)
```

用于查找解密器服务的服务过滤器是 （& (objectClass=org.jasypt.encryption.StringEncryptor) (alias=<alias>)，其中 < alias > 是连接工厂配置工厂 PID 的 decryptor 属性的值。

7.5. 使用 JMS 连接池

本节讨论 JMS 连接/会话池选项。少于 JDBC 的选择数量要少。这些信息被组织到以下主题：

- *第 7.5.1 节 “使用 JMS 连接池简介”*
- *第 7.5.2 节 “使用 pax-jms-pool-pooledjms 连接池模块”*
- *第 7.5.3 节 “使用 pax-jms-pool-narayana 连接池模块”*
- *第 7.5.4 节 “使用 pax-jms-pool-transx 连接池模块”*



重要

要使用 XA 恢复，您应该使用 pax-jms-pool-transx 或 pax-jms-pool-narayana 连接池模块。

7.5.1. 使用 JMS 连接池简介

目前，您已注册了一个特定于代理的连接工厂。由于 连接工厂 本身是连接工厂的工厂，org.ops4j.pax.jms.service.ConnectionFactoryFactory 服务可能会被视为 meta factory。它应该能够生成两种类型的连接工厂：

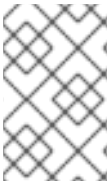
- *javax.jms.ConnectionFactory*
- *javax.jms.XAConnectionFactory*

pax-jms-pool busybox 捆绑包与 org.ops4j.pax.jms.service.ConnectionFactoryFactory 服务平稳工作。这些捆绑包提供了 org.ops4j.pax.jms.service.PooledConnectionFactoryFactory 的实现，它们可用于使用一组属性和原始 org.ops4j.pax.jms.service.ConnectionFactoryFactory 来创建池连接工厂。例如：

```
public interface PooledConnectionFactoryFactory {

    ConnectionFactory create(ConnectionFactoryFactory cff, Map<String, Object> props);

}
```

下表显示了哪些捆绑包注册池连接工厂。在表中，o.o.p.j.p 代表 org.ops4j.pax.jms.pool。

捆绑包（Bundle）	PooledConnectionFactoryFactory	池密钥
pax-jms-pool-pooledjms	o.o.p.j.p.pooledjms.PooledJms(XA) PooledConnectionFactoryFactory	pooledjms
pax-jms-pool-narayana	o.o.p.j.p.narayana.PooledJms(XA)PooledConnectionFactoryFactory	narayana
pax-jms-pool-transx	o.o.p.j.p.transx.Transx(XA)PooledConnectionFactoryFactory	transx



注意

pax-jms-pool-narayana 工厂名为 PooledJms (XA) PooledConnectionFactoryFactory，因为它基于 pooled-jms 库。它为 XA 恢复添加与 Narayana 事务管理器的集成。

以上捆绑包只安装连接工厂。不安装连接工厂的捆绑包。因此，需要一些操作来调用 javax.jms.ConnectionFactory org.ops4j.pax.jms.service.PooledConnectionFactoryFactory.create（）方法。

7.5.2. 使用 pax-jms-pool-pooledjms 连接池模块

了解如何使用 pax-jms-pool-pooledjms 捆绑包可帮助您仅使用 pax-jms-pool-pooledjms 捆绑包，以及 pax-jms-pool-pool -pool-narayna 捆绑包，几乎执行几乎所有操作为 pax-jms-pool-pooledjms。

pax-jms-config 捆绑包跟踪以下内容：

- *org.ops4j.pax.jms.service.ConnectionFactoryFactory services*
- *org.ops4j.connectionfactory factory PIDs*
- *org.ops4j.pax.jms.service.PooledConnectionFactoryFactory 的实例由 pax-jms-pool 114 捆绑包之一注册。*

如果工厂配置包含 池 属性，则由 pax-jms-config 捆绑包注册的最终连接工厂是特定于代理的连接工厂。如果 pool=pooledjms，则连接工厂将嵌套在以下之一：

- *org.messaginghub.pooled.jms.JmsPoolConnectionFactory (xa=false)*
- *org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory (xa=true)*

除了 pool 属性（以及布尔值 xa 属性外，它选择其中一个非xa/xa 连接工厂），org.ops4j .connectionfactory factory PID 可能会包含前缀为 pool 的属性。

对于 pooled-jms 库，将使用这些前缀属性（在删除前缀后）来配置实例：

- *org.messaginghub.pooled.jms.JmsPoolConnectionFactory, 或*
- *org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory*

以下列表是 pooled-jms 池(org.ops4j.connectionfactory-artemis factory PID)的真实配置，它使用带有 jms.-prefixed 属性的便捷语法：

```
# configuration for pax-jms-config to choose and configure specific
org.ops4j.pax.jms.service.ConnectionFactoryFactory
name = jms/artemis
connectionFactoryType = ConnectionFactory
jms.url = tcp://localhost:61616
jms.user = fuse
jms.password = fuse
# org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory specific coniguration
jms.callTimeout = 12000
# ...

# hints for pax-jms-config to use selected org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
pool = pooledjms
xa = false

# pooled-jms specific configuration of org.messaginghub.pooled.jms.JmsPoolConnectionFactory
pool.idleTimeout = 10
pool.maxConnections = 100
pool.blockIfSessionPoolIsFull = true
# ...
```

在上述配置中，池和 xa 键是提示（服务过滤器属性）来选择一个注册的 org.ops4j.pax.jms.service.PooledConnectionFactoryFactory 服务。对于 pooled-jms 库，它是：

```
karaf@root()> feature:install pax-jms-pool-pooledjms

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-pool-pooledjms

OPS4J Pax JMS MessagingHub JMS Pool implementation (252) provides:
----------------------------------------------------------------------
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 331
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 335
service.scope = singleton
xa = true
```

以下是创建和配置连接池的步骤的完整示例：

1. **安装所需的功能：**

   ```
   karaf@root()> feature:install -v pax-jms-pool-pooledjms pax-jms-artemis
   Adding features: pax-jms-pool-pooledjms/[1.0.0,1.0.0]
   ...
   ```

2. **安装 jms 功能：**

   ```
   karaf@root()> feature:install jms
   ```

```
karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
----------------------------------------------------
 service.bundleid = 249
 service.id = 327
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (249)

karaf@root()> service:list org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
 pool = pooledjms
 service.bundleid = 251
 service.id = 328
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)

[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
 pool = pooledjms
 service.bundleid = 251
 service.id = 333
 service.scope = singleton
 xa = true
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)
```

3. *创建 **工厂配置** ：*

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set connectionFactoryType ConnectionFactory
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set protocol amqp # so we switch to
org.apache.qpid.jms.JmsConnectionFactory
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:property-set jms.username admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set pool pooledjms
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.idleTimeout 10
karaf@root()> config:property-set pool.maxConnections 123
karaf@root()> config:property-set pool.blockIfSessionPoolIsFull true
karaf@root()> config:update
```

4. *检查 **pax-jms-config 是否处理到 javax.jms.ConnectionFactory 服务中的配置：***

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
```

```
connectionFactoryType = ConnectionFactory
felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
jms.password = admin
jms.url = amqp://localhost:61616
jms.username = admin
osgi.jndi.service.name = jms/artemis
pax.jms.managed = true
pool.blockIfSessionPoolIsFull = true
pool.idleTimeout = 10
pool.maxConnections = 123
protocol = amqp
service.bundleid = 250
service.factoryPid = org.ops4j.connectionfactory
service.id = 347
service.pid = org.ops4j.connectionfactory.fc1b9e85-91b4-421b-aa16-1151b0f836f9
service.scope = singleton
type = artemis
Provided by :
 OPS4J Pax JMS Config (250)
```

5. ***使用连接工厂：***

```
karaf@root()> jms:connectionfactories
JMS Connection Factory
--------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property | Value
---------+------------------------------
product  | QpidJMS
version  | 0.30.0.redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                                | Content       | Charset | Type | Correlation ID |
Delivery Mode | Destination | Expiration | Priority | Redelivered | ReplyTo | Timestamp
------------------------------------------+---------------+---------+------+----------------+
--------------+-------------+------------+----------+-------------+---------+--------------
--------------------------------------------
ID:64842f99-5cb2-4850-9e88-f50506d49d20:1:1:1-1 | Hello Artemis | UTF-8   |      |
| Persistent    | DEV.QUEUE.1 | Never      | 4        | false       |         | Mon May 14
12:47:13 CEST 2018
```

7.5.3. 使用 pax-jms-pool-narayana 连接池模块

pax-jms-pool-narayna 模块几乎执行所有操作为 pax-jms-pool-pooledjms。它安装特定于池的 org.ops4j.pax.jms.service.PooledConnectionFactoryFactory，适用于 XA 和非 XA 场景。唯一的区别是在 XA 场景中，有一个额外的集成点。org.jboss.tm.XAResourceRecovery OSGi 服务被注册为由 com.arjuna.ats.arjuna.recovery.RecoveryManager 获取。

7.5.4. 使用 pax-jms-pool-transx 连接池模块

pax-jms-pool-transx 模块提供了 org.ops4j.pax.jms.service.PooledConnectionFactoryFactory 服务的实现，它基于 pax-transx-jms 捆绑包。pax-transx-jms 捆绑包使用 org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder 工具来创建 javax.jms.ConnectionFactory 池。这是 第 8.3 节 “关于 pax-transx 项目” 中讨论的 JCA (Java™ Connector Architecture)解决方案。

7.6. 以工件的形式部署连接工厂

本主题讨论了真实建议。

在 部署方法 中，javax.jms.ConnectionFactory 服务由应用代码直接注册。通常，此代码位于 Blueprint 容器中。蓝图 XML 可以是普通 OSGi 捆绑包的一部分，可通过使用 mvn: URI 进行安装，并存储在 Maven 存储库（本地或远程）。与 Configuration Admin 配置相比，版本控制（如捆绑包）更容易。

pax-jms-config 版本 1.0.0 捆绑包为连接工厂 配置添加部署方法。应用程序开发人员注册 javax.jms.(XA) ConnectionFactory 服务（通常使用 Bluerpint XML）并指定服务属性。然后 pax-jms-config 检测到已注册的、特定于代理的连接工厂，以及（使用服务属性）将服务嵌套在通用的、特定于代理的连接池中。

以下是使用 Blueprint XML 的三种 部署方法 ：

- *第 7.6.1 节 “手动部署连接工厂”*
- *第 7.6.2 节 “连接工厂部署”*
- *第 7.6.3 节 “连接工厂的混合部署”*

7.6.1. 手动部署连接工厂

在此方法中，不需要 pax-jms-config 捆绑包。应用程序代码负责注册特定于代理和通用连接池。

```
<!--
   Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
   <argument value="tcp://localhost:61616" />
   <property name="callTimeout" value="2000" />
   <property name="initialConnectAttempts" value="3" />
</bean>

<!--
   Fuse exports this service from fuse-pax-transx-tm-narayana bundle.
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non broker-specific, generic, pooling, enlisting javax.jms.ConnectionFactory
-->
<bean id="pool" class="org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory">
   <property name="connectionFactory" ref="artemis" />
   <property name="transactionManager" ref="tm" />
   <property name="maxConnections" value="10" />
   <property name="idleTimeout" value="10000" />
</bean>
```

```
<!--
   Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
   <service-properties>
      <!-- Giving connection factory a name using one of these properties makes identification
easier in jms:connectionfactories: -->
      <entry key="osgi.jndi.service.name" value="jms/artemis" />
      <!--<entry key="name" value="jms/artemis" />-->
      <!-- Without any of the above, name will fall back to "service.id" -->
   </service-properties>
</service>
```

以下是显示如何使用它的 shell 命令：

```
karaf@root()> feature:install artemis-core-client artemis-jms-client
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 244
karaf@root()> install -s mvn:org.messaginghub/pooled-jms/0.3.0
Bundle ID: 245
karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-manual.xml
Bundle ID: 246

karaf@root()> bundle:services -p 246

Bundle 246 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 246
service.id = 340
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 246
service.id = 341
service.scope = singleton

karaf@root()> feature:install jms

karaf@root()> jms:connectionfactories
JMS Connection Factory
————————————————————
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

如上方列表所示，Blueprint 捆绑包导出 **javax.jms.ConnectionFactory** 服务，它是一个通用的、非特定于代理的连接池。特定于代理的 **javax.jms.XAConnectionFactory** 没有 作为 OSGi 服务注册，因为 Blueprint XML 没有明确的 **< service ref="artemis">** 声明。

7.6.2. 连接工厂部署

此方法演示了以规范的方式使用 **pax-jms-config**。这与推荐用于 Fuse 6.x 的方法有点不同，其要求是将池配置指定为服务属性。

以下是 Blueprint XML 示例：

```
<!--
   A broker-specific org.ops4j.pax.jms.service.ConnectionFactoryFactory that can create
(XA)ConnectionFactory
   using properties. It is registered by pax-jms-* bundles
-->
<reference id="connectionFactoryFactory"
       interface="org.ops4j.pax.jms.service.ConnectionFactoryFactory"
       filter="(type=artemis)" />

<!--
   Non broker-specific org.ops4j.pax.jms.service.PooledConnectionFactoryFactory that can
create
   pooled connection factories with the help of
org.ops4j.pax.jms.service.ConnectionFactoryFactory

   For example, pax-jms-pool-pooledjms bundle registers
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
   with these properties:
    - pool = pooledjms
    - xa = true|false (both are registered)
-->
<reference id="pooledConnectionFactoryFactory"
       interface="org.ops4j.pax.jms.service.PooledConnectionFactoryFactory"
       filter="(&amp;(pool=pooledjms)(xa=true))" />

<!--
   When using XA connection factories, javax.transaction.TransactionManager service is not
needed here.
   It is used internally by xa-aware pooledConnectionFactoryFactory.
-->
<!--<reference id="tm" interface="javax.transaction.TransactionManager" />-->

<!--
   Finally, use both factories to expose the pooled, xa-aware, connection factory.
-->
<bean id="pool" factory-ref="pooledConnectionFactoryFactory" factory-method="create">
   <argument ref="connectionFactoryFactory" />
   <argument>
      <props>
         <!--
            Properties needed by artemis-specific
org.ops4j.pax.jms.service.ConnectionFactoryFactory
         -->
         <prop key="jms.url" value="tcp://localhost:61616" />
         <prop key="jms.callTimeout" value="2000" />
```

```
        <prop key="jms.initialConnectAttempts" value="3" />
        <!-- Properties needed by pooled-jms-specific
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory -->
        <prop key="pool.maxConnections" value="10" />
        <prop key="pool.idleTimeout" value="10000" />
      </props>
    </argument>
</bean>

<!--
   Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
    <service-properties>
      <!-- Giving connection factory a name using one of these properties makes identification
easier in jms:connectionfactories: -->
      <entry key="osgi.jndi.service.name" value="jms/artemis" />
      <!--<entry key="name" value="jms/artemis" />-->
      <!-- Without any of the above, name will fall back to "service.id" -->
    </service-properties>
</service>
```

上例使用 factory beans，通过使用连接工厂(...)创建连接工厂。不需要明确引用 javax.transaction.TransactionManager 服务，因为这由 XA 感知的池 ConnectionFactoryFactory 在内部跟踪。

以下是在 Fuse/Karaf shell 中查找它的方式：

```
karaf@root()> feature:install jms pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-
factory-pooledjms.xml
Bundle ID: 253
karaf@root()> bundle:services -p 253

Bundle 253 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 253
service.id = 347
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-factory-pooledjms.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 253
service.id = 348
service.scope = singleton

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/artemis
```

```
karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

如上方列表所示，Blueprint 捆绑包导出 **javax.jms.ConnectionFactory** 服务，它是一个通用的、非特定于代理的连接池。特定于代理的 **javax.jms.XAConnectionFactory** 没有 作为 OSGi 服务注册，因为 Blueprint XML 没有明确的 **< service ref="artemis">** 声明。

7.6.3. 连接工厂的混合部署

pax-jms-config 1.0.0 捆绑包使用服务属性在池连接工厂中添加其他特定于代理的连接工厂。这个方法与用于在 Fuse 6.x 中工作的方式匹配。

以下是 Blueprint XML 示例：

```
<!--
    Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
    <argument value="tcp://localhost:61616" />
    <property name="callTimeout" value="2000" />
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties.
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory.
It will be registered
    automatically by pax-jms-config with the same properties as this <service>, but with a
higher service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory
inside connection pool -->
        <entry key="pool" value="pooledjms" />
        <!-- <service>/@id attribute does not propagate, but name of the connection factory is
required using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally by e.g., pax-jms-pool-pooledjms -->
        <entry key="pool.maxConnections" value="10" />
        <entry key="pool.idleTimeout" value="10000" />
    </service-properties>
</service>
```

在上例中，您可以看到只手动注册特定于代理的连接工厂。**pool=pooledjms** 服务属性是连接工厂跟踪器的提示，它由 **pax-jms-config** 捆绑包管理。使用这个服务属性的连接工厂服务嵌套在池连接工厂中，在本例中为 **pax-jms-pool-pooledjms**。

以下是在 Fuse/Karaf shell 中查找它的方式：

```
karaf@root()> feature:install jms pax-jms-config pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-discovery.xml
Bundle ID: 254

karaf@root()> bundle:services -p 254

Bundle 254 provides:
--------------------
objectClass = [javax.jms.XAConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = artemis
pool = pooledjms
pool.idleTimeout = 10000
pool.maxConnections = 10
service.bundleid = 254
service.id = 349
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-discovery.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 254
service.id = 351
service.scope = singleton

karaf@root()> service:list javax.jms.XAConnectionFactory
[javax.jms.XAConnectionFactory]
-------------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pool = pooledjms
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 254
 service.id = 349
 service.scope = bundle
Provided by :
 Bundle 254
Used by:
 OPS4J Pax JMS Config (251)

karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pax.jms.managed = true
 pax.jms.service.id.ref = 349
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 251
 service.id = 350
 service.ranking = 1000
 service.scope = singleton
```

```
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property | Value
---------+-----------------------------------------
product  | ActiveMQ
version  | 2.4.0.amq-711002-redhat-1
```

在上例中，jms:connectionfactories 仅显示一个服务，因为此命令会删除重复的名称。两个服务由数据源混合部署中的 jdbc:ds-list 出示。

javax.jms.XAConnectionFactory 从 Blueprint 捆绑包注册，它声明了 pool = pooledjms 属性。

javax.jms.ConnectionFactory 从 pax-jms-config 捆绑包中注册，并：

- **它没有 pool = pooledjms 属性。在注册 wrapper 连接工厂时，它已被删除。**
- **它具有 service.ranking = 1000 属性，因此当它始终是首选的版本，例如，根据名称查找连接工厂。**
- **它具有 pax.jms.managed = true 属性，因此不会尝试再次打包它。**
- **它具有 pax.jms.service.id.ref = 349 属性，它指示嵌套在连接池中的原始连接工厂服务。**

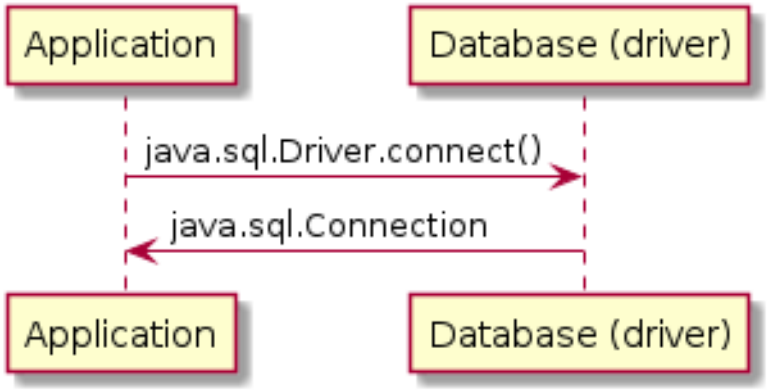
第 8 章 关于 JAVA 连接器架构

创建 JCA 规格是为了（有其他事项） 一般是 拥有这三位参与者的场景：

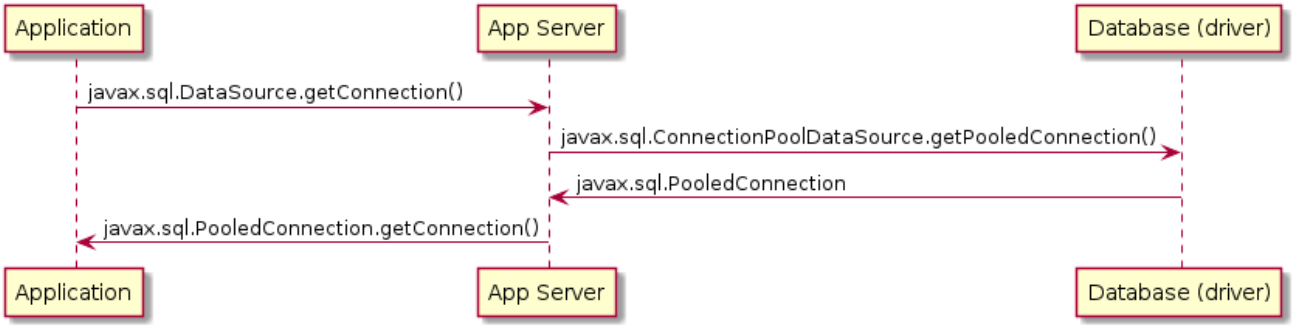
- *一个外部系统，如数据库或 通常是 EIS 系统*
- *developer1 应用服务器*
- *部署的应用程序*

8.1. 简单的 JDBC 模拟

在最简单的场景中，只有应用程序和数据库，您有：



添加公开 javax.sql.DataSource 的应用服务器，您有以下几项（无需重新调用 XA 等数据源的不同方面）：



8.2. 使用 JCA 概述

JCA 常规化 数据库驱动程序 的概念，方法是在驱动程序和应用服务器 之间添加 双向通信。驱动程序成为由 javax. resource.spi.ResourceAdapter 表示的资源适配器。

有两个重要接口：

- *javax.resource.spi.ManagedConnectionFactory 由资源适配器实施。*
- *javax.resource.spi.ConnectionManager 由应用服务器实现。*

ManagedConnectionFactory 接口有两个目的：

- *Object createConnectionFactory (ConnectionManager cxManager) 方法可用于为给定 EIS（或数据库或消息代理）生成 连接工厂，供应用程序代码使用。返回的对象 可以是：*

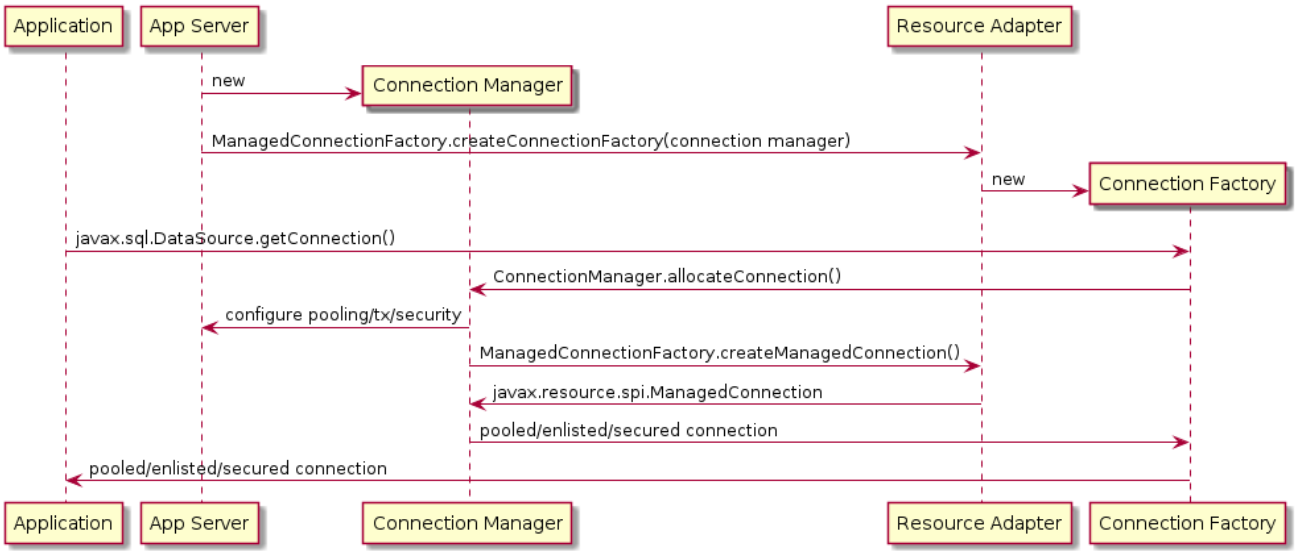
- 通用 javax.resource.cci.ConnectionFactory（此处未描述，请参阅 JCA 1.6，第 17 章：常见客户端接口）
- EIS 特定的连接工厂，如知名 javax.sql.DataSource 或 javax.jms.ConnectionFactory。这是由 pax-transx- jdbc 和 pax-transx -jms 捆绑包使用的连接工厂类型。

- javax.resource.spi.ManagedConnection ManagedConnectionFactory.createManagedConnection （）方法由应用服务器使用，创建与 EIS/database/broker 的实际物理连接。

ConnectionManager 由应用服务器实现，并由资源适配器使用。它是首先执行 QoS 操作（池、安全性、事务管理），最后委托资源适配器的 ManagedConnectionFactory 来创建 ManagedConnection 实例的应用服务器。流程类似如下：

1. 应用程序代码使用从 ManagedConnectionFactory.createConnectionFactory （）返回的对象创建并公开的连接工厂。它可以是通用的 CCI 接口，如 javax.sql.DataSource。
2. 此连接工厂不自行创建连接，而是委托给 ConnectionManager.allocateConnection （）传递资源适配器- 特定的 ManagedConnectionFactory
3. 由应用服务器实现的 ConnectionManager 创建支持对象、管理事务、池等，最终从传递 ManagedConnectionFactory 获取物理（管理）连接。
4. 应用程序代码获得连接，通常是应用服务器创建的 wrapper/proxy，最终委托给资源适配器的特定物理连接。

下图中，应用服务器创建了特定于 EIS 的非CCI连接工厂。只需 - 访问 EIS （这里：数据库）是使用 javax.sql.DataSource 接口完成的，驱动程序的任务是提供物理连接，而应用服务器则将其嵌套在（通常）执行池/备份的代理内（通常）



8.3. 关于 PAX-TRANSX 项目

pax-transx 项目在 OSGi 中提供对 JTA/JTS 事务管理的支持，以及 JDBC 和 JMS 的资源池。它将关闭 pax-jdbc 和 pax-jms 之间的差距。

- Pax-jdbc 为 javax.sql. (XA) ConnectionFactory 服务添加配置选项和发现，并附带一些 JDBC 池实施
- Pax-jms 对 javax.jms. (XA) ConnectionFactory 服务执行相同的操作，并提供了一些 JMS 池实施

- **Pax-transx** 为 javax.transaction.TransactionManager 实施和（最终）为 **javax.transaction.TransactionManager** 实现添加配置选项和发现，提供基于 JCA 的 JDBC/JMS 连接管理，支持池和交易器。

关于 JDBC 连接池 和 JMS 连接池 的部分仍然有效。使用基于 JCA 的池的唯一更改是在注册 JDBC 数据源和 JMS 连接工厂时使用 **pool=transx** 属性。

- **Pax-jdbc-pool-transx** 使用 **org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder** from **pax-transx-jdbc**
- **Pax-jms-pool-transx** 使用 **org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder** from **pax-transx-jms**

虽然池化数据源/连接工厂是以 构建器样式 创建的（没有 Java™ bean 属性），但 JDBC 支持这些属性：

- **name**
- **userName**
- **password**
- **commitBeforeAutocommit**
- **preparedStatementCacheSize**
- **transactionIsolationLevel**
- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**
- **connectionTimeout**
- **idleTimeout**
- **maxLifetime**

JMS 支持这些属性：

- **name**
- **userName**
- **password**
- **clientID**
- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**

- *connectionTimeout*
- *idleTimeout*
- *maxLifetime*

XA 恢复需要 用户名和密码 属性才能工作（就像使用 aries.xa.username 和 aries.xa.password 属性）在 Fuse 6.x 中。

在蓝图中使用此 JDBC 配置(mind pool=transx)：

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Expose database-specific data source with service properties
   No need to expose pooling, enlisting, non database-specific javax.sql.DataSource - it'll be
registered
   automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
   <service-properties>
      <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source
inside connection pool -->
      <entry key="pool" value="transx" />
      <!-- <service>/@id attribute doesn't propagate, but name of the datasource is required
using one of: -->
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
      <!-- or: -->
      <!--<entry key="dataSourceName" value="jdbc/postgresql" />-->
      <!-- Other properties, that normally are needed by e.g., pax-jdbc-pool-transx -->
      <entry key="pool.maxPoolSize" value="13" />
      <entry key="pool.userName" value="fuse" />
      <entry key="pool.password" value="fuse" />
   </service-properties>
</service>
```

使用蓝图中的这个 JMS 配置(mind pool=transx)：

```
<!--
   Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
```

```
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
    <argument index="0" value="tcp://localhost:61616" />
    <!-- credentials needed for JCA-based XA-recovery -->
    <argument index="1" value="admin" />
    <argument index="2" value="admin" />
    <property name="callTimeout" value="2000" />
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory - it'll be registered
    automatically by pax-jms-config with the same properties as this <service>, but with higher service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory inside connection pool -->
        <entry key="pool" value="transx" />
        <!-- <service>/@id attribute doesn't propagate, but name of the connection factory is required using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally are needed e.g., pax-jms-pool-transx -->
        <entry key="pool.maxPoolSize" value="13" />
        <entry key="pool.userName" value="admin" />
        <entry key="pool.password" value="admin" />
    </service-properties>
</service>
```

您有一个 JDBC 数据源和 JMS 连接工厂，该工厂利用基于 JCA 的资源管理。基于 transx 的池可以正确地与 pax-transx-tm-narayana 与 XA 恢复有关。

所需功能有：

- *pax-jdbc-pool-tranx*
- *pax-jms-pool-tranx*
- *pax-transx-jdbc*

- ***pax-transx-jms***

- ***Pax-jms-artemis （使用 A-MQ 7 时）***

在配置了三个、可用的引用类型后，您可以编写一个应用程序。三种类型的服务有：

- 一个事务管理器，它是以下接口之一：
  - javax.transaction.UserTransaction
  - javax.transaction.TransactionManager
  - org.springframework.transaction.PlatformTransactionManager
- 至少有一个 JDBC 数据源实施 javax.sql.DataSource. 接口。通常，有多个数据源。
- 至少一个实施 javax.jms.ConnectionFactory 接口的 JMS 连接工厂。通常，有多个。

这部分论述了与管理事务、数据源和连接工厂相关的特定于 Camel 的配置。

注意

本节论述了几个与 Spring 相关的概念，如 SpringTransactionPolicy。Spring XML DSL 和 Blueprint XML DSL 之间存在明显区别，它们是定义 Camel 上下文的 XML 语言。Spring XML DSL 现在在 Fuse 中被弃用。但是，Camel 事务机制仍然在内部使用 Spring 库。

这里的大部分信息都不依赖于所使用的 PlatformTransactionManager 类型。如果 PlatformTransactionManager 是 Narayana 事务管理器，则使用完整的 JTA 事务。如果 PlatformTransactionManager 定义为本地 Blueprint < bean>，例如 org.springframework.jms.connection.JmsTransactionManager，则使用本地事务。

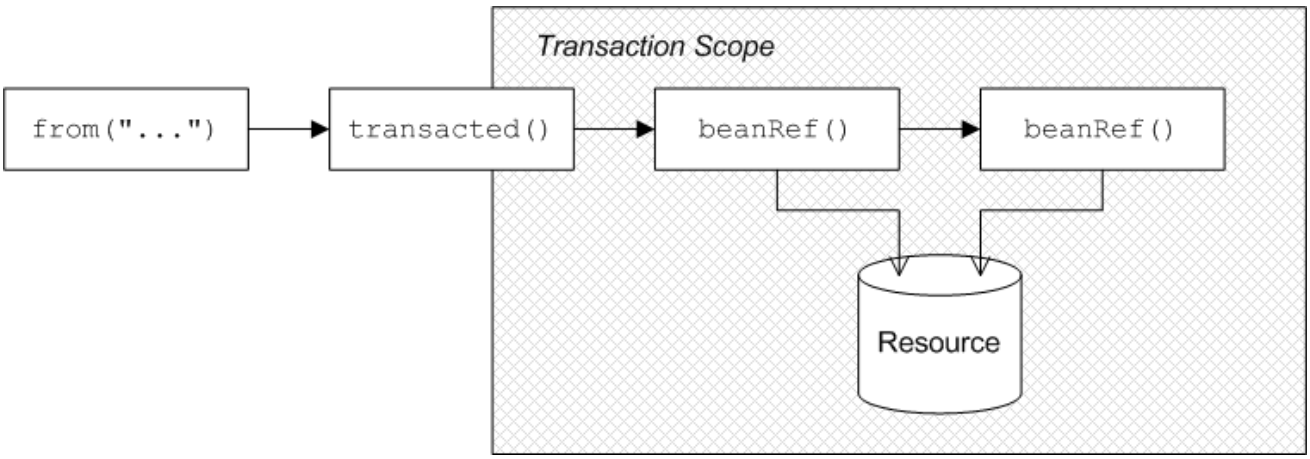
事务处理指的是启动、提交和回滚事务的步骤。本节介绍通过编程和配置控制事务处理的机制。

- 第 9.1 节 “通过标记路由进行事务处理”
- 第 9.2 节 “按事务端点划分”
- 第 9.3 节 “声明事务的划分”
- 第 9.4 节 “事务传播策略”
- 第 9.5 节 “错误处理和回滚”

9.1. 通过标记路由进行事务处理

Apache Camel 提供了在路由中启动事务的简单机制。在 Java DSL 中插入 transacted （） 命令，或者在 XML DSL 中插入 <transacted /> 标签。

图 9.1. 通过标记路由来划分



翻译处理器取消处理，如下所示：

1. 当交换进入转换处理器时，转换的处理器调用默认事务管理器来开始事务，并将事务附加到当前线程。
2. 当交换到达剩余的路由结束时，转换的处理器调用事务管理器来提交当前的事务。

9.1.1. 使用 JDBC 资源的路由示例

图 9.1 “通过标记路由来划分” 显示通过向路由中添加 transacted （） DSL 命令进行事务的路由示例。遵循 transacted （） 节点的所有路由节点都包含在事务范围内。在本例中，以下两个节点访问 JDBC 资源：

9.1.2. Java DSL 中的路由定义

以下 Java DSL 示例演示了如何通过使用 transacted （） DSL 命令标记路由来定义事务路由：

```
import org.apache.camel.builder.RouteBuilder;

class MyRouteBuilder extends RouteBuilder {
   public void configure() {
      from("file:src/data?noop=true")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit");
   }
}
```

在本例中，file 端点读取一些 XML 格式文件，这些文件描述了资金从一个帐户传输到另一个帐户的传输。第一个 bean （） 调用将指定资金总和计入 beneficiary 帐户，然后第二个 bean （） 调用从发送者的帐户中减去指定资金总和。两个 bean （） 调用都会导致对数据库资源的更新。假设数据库资源通过事务管理器绑定到事务，例如： 第 6 章 使用 JDBC 数据源。

9.1.3. Blueprint XML 中的路由定义

前面的路由也可以在 Blueprint XML 中表示。& lt;transacted /> 标签将路由标记为事务，如以下 XML 所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <route>
         <from uri="file:src/data?noop=true" />
         <transacted />
         <bean ref="accountService" method="credit" />
         <bean ref="accountService" method="debit" />
      </route>
   </camelContext>

</blueprint>
```

9.1.4. 默认事务管理器和转换的策略

为分离事务，转换的处理器必须与特定的事务管理器实例关联。要保存您必须在每次调用 transacted（） 时都指定事务管理器，转换的处理器会自动选择一个可靠的默认值。例如，如果您的配置中只有一个事务管理器实例，则转换的处理器隐式选择此事务管理器，并使用它来处理事务。

转换器的处理器也可以配置有 transacted 策略，该策略是 TransactedPolicy 类型，它封装了传播策略和事务管理器（详情请参阅 第 9.4 节 “事务传播策略”）。以下规则用于选择默认的事务管理器或事务策略：

1. 如果只有一个 bean 是 org.apache.camel.spi.TransactedPolicy 类型，则使用此 bean。

注意

TransactedPolicy 类型是 SpringTransactionPolicy 类型的基本类型，如 第 9.4 节 “事务传播策略” 中所述。因此，这里引用的 bean 可以是 SpringTransactionPolicy bean。

2. 如果存在类型为 org.apache.camel.spi.TransactedPolicy 的 bean，其 ID 为 ,PROPAGATION_REQUIRED，则使用此 bean。

3. 如果只有一个 bean of org.springframework.transaction.PlatformTransactionManager 类型，则使用此 bean。

您还可以通过向 transacted （） 提供 bean ID 作为参数来显式指定 bean。请参阅 第 9.4.4 节 “Java DSL 中带有 PROPAGATION_NEVER 策略的示例路由”。

9.1.5. 事务范围

如果您将转换的处理器插入到路由中，则事务管理器每次通过此节点时都会创建一个新的事务。事务的范围定义如下：

- 事务仅与当前线程关联。
- 事务范围包括所有遵循转换处理器的路由节点。

任何在 transacted 处理器之前的路由节点都不在事务中。但是，如果路由以事务端点开头，则路由中的所有节点都位于事务中。请参阅 第 9.2.5 节 “路由开始时的事务端点”。

考虑以下路由：它不正确，因为 transacted () DSL 命令错误地出现在访问数据库资源的 first bean () 调用后：

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .bean("accountService", "credit")
                .transacted()  // <-- WARNING: Transaction started in the wrong place!
                .bean("accountService", "debit");
    }
}
```

9.1.6. 事务路由中没有线程池

务必要了解，给定的事务只与当前线程关联。您不能在事务路由中间创建线程池，因为新线程中的处理不会参与当前的事务。例如，以下路由被绑定到导致问题：

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .threads(3)  // WARNING: Subthreads are not in transaction scope!
                .bean("accountService", "credit")
                .bean("accountService", "debit");
    }
}
```

由于 threads () DSL 命令与 transacted 路由不兼容，如前面的路由是一定的破坏。即使 threads () 调用在 transacted () 调用前面，路由也不会按预期工作。

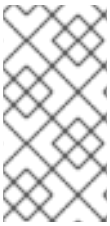
9.1.7. 将路由拆分为片段

如果您想要将路由拆分为片段，并且每个路由片段都参与当前事务，您可以使用 direct: 端点。例如，要将交换发送到单独的路由片段，具体取决于传输量大（不等于 100）还是小（不等于 100），您可以使

用 choice （） DSL 命令和直接端点，如下所示：

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
   ...
   public void configure() {
      from("file:src/data?noop=true")
            .transacted()
            .bean("accountService", "credit")
            .choice().when(xpath("/transaction/transfer[amount > 100]"))
            .to("direct:txbig")
            .otherwise()
            .to("direct:txsmall");

      from("direct:txbig")
            .bean("accountService", "debit")
            .bean("accountService", "dumpTable")
            .to("file:target/messages/big");

      from("direct:txsmall")
            .bean("accountService", "debit")
            .bean("accountService", "dumpTable")
            .to("file:target/messages/small");
   }
}
```

以 direct:txbig 和以 direct:txsmall 开头的片段都参与当前的事务，因为直接端点是同步的。这意味着，片段在与第一个路由片段相同的线程中执行，因此它们包含在相同的事务范围内。



注意

您不能使用 seda 端点来加入路由片段。seda 消费者端点创建一个新的线程（或线程），以执行路由片段（异步处理）。因此，片段不会参与原始的事务。

9.1.8. 资源端点

当 Apache Camel 组件显示为路由的目标时，以下 Apache Camel 组件充当资源端点，例如，如果它们出现在 to （） DSL 命令中。也就是说，这些端点可以访问事务资源，如数据库或持久队列。资源端点可以参与当前的事务，只要它们与启动当前事务的转换处理器相同的事务管理器关联。

- ActiveMQ

- AMQP
- 休眠
- iBatis
- JavaSpace
- JBI
- JCR
- JDBC
- JMS
- JPA
- LDAP

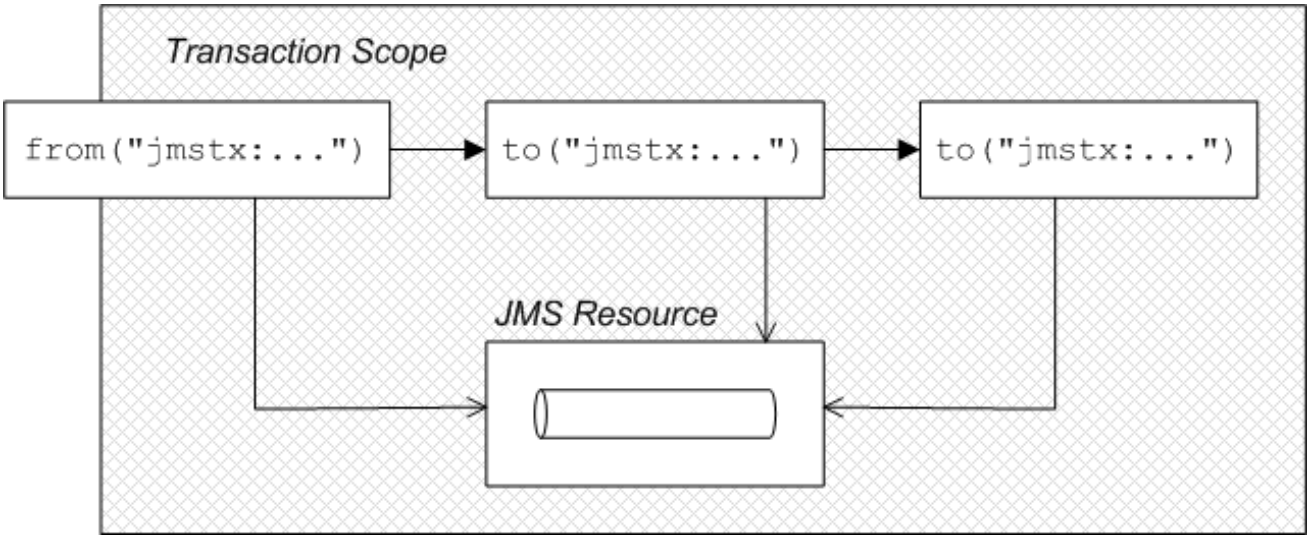
9.1.9. 使用资源端点的示例路由

以下示例显示了具有资源端点的路由。这将向两个不同的 JMS 队列发送资金传输订单。信用 队列处理了接收方帐户的顺序。解封 队列处理以取消发送者帐户的顺序。只有存在对应的 debit 时才应有学分。因此，您要将 enqueueing 操作放在单个事务中。如果事务成功，则信用订购和下序都将排队。如果发生错误，则既不会排队任何顺序。

```
from("file:src/data?noop=true")
        .transacted()
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

9.2. 按事务端点划分

如果路由开始时的消费者端点访问资源，则 transacted （） 命令不使用，因为它在交换轮询后启动事务。换句话说，事务开始太晚，以在事务范围内包括消费者端点。在这种情况下，正确的方法是使端点本身负责启动事务。能够管理事务的端点被称为 事务端点。



事务端点有两种不同的模型，如下所示：

- 常规情况 - 通常，事务端点取消处理事务，如下所示：

  1. 当交换到达端点时，或者端点成功轮询交换时，端点调用其关联的事务管理器以开始事务。
  2. 端点会将新事务附加到当前线程。
  3. 当交换到达路由结束时，事务端点调用事务管理器来提交当前的事务。

- 带有 InOut 交换的 JMS 端点 - 当 JMS 消费者端点收到 InOut 交换，并且此交换路由到另一个 JMS 端点，这必须被视为特殊情况。问题是，如果您尝试在单个事务中包括整个请求/回复交换，则路由可能会死锁。

9.2.1. 带有 JMS 端点的路由示例

第 9.2 节 “按事务端点划分” 显示通过路由开始时存在事务端点（在 from （） 命令中）存在的路由示例。所有路由节点都包含在事务范围内。在本例中，路由中的所有端点访问 JMS 资源。

9.2.2. Java DSL 中的路由定义

以下 Java DSL 示例演示了如何通过启动使用事务端点的路由来定义事务路由：

```
from("jmstx:queue:giro")
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

在上例中，事务范围包含端点 jmstx:queue:giro,jmstx:queue:credits, 和 jmstx:queue:debits。如果事务成功，则会从上级队列永久删除交换，并推送到信用 队列和 去位 队列。如果交易失败，交换不会放入信用队列，并且将交换推送回巨队列。 默认情况下，JMS 会自动尝试更新消息。JMS 组件 bean jmstx 必须明确配置为使用事务，如下所示：

```
<blueprint ...>
    <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
        <property name="configuration" ref="jmsConfig" />
    </bean>

    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="jmsConnectionFactory" />
        <property name="transactionManager" ref="jmsTransactionManager" />
        <property name="transacted" value="true" />
    </bean>
    ...
</blueprint>
```

在上例中，jmsTransactionManager 的事务管理器实例 与 JMS 组件关联，transacted 属性设为 true，以启用 InOnly Exchanges 的事务处理。

9.2.3. Blueprint XML 中的路由定义

前面的路由可以在 Blueprint XML 中表示，如下所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <from uri="jmstx:queue:giro" />
            <to uri="jmstx:queue:credits" />
            <to uri="jmstx:queue:debits" />
        </route>
    </camelContext>

</blueprint>
```

9.2.4. 不需要 DSL transacted （） 命令

在以事务端点开头的路由中不需要 transacted （） DSL 命令。但是，假设默认事务策略是 PROPAGATION_REQUIRED （请参阅 第 9.4 节 “事务传播策略”），通常不会损害包含 transacted （） 命令，如下例所示：

```
from("jmstx:queue:giro")
        .transacted()
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

但是，这个路由可能会以意外的方式的行为，例如，如果在 Blueprint XML 中创建了具有非默认传播策略的单一 TransactedPolicy bean。请参阅 第 9.1.4 节 “默认事务管理器和转换的策略”。因此，通常不要将 transacted （） DSL 命令包含在以事务端点开头的路由中。

9.2.5. 路由开始时的事务端点

以下 Apache Camel 组件在路由启动时显示为事务端点（例如，如果它们出现在 from （） DSL 命令中）。也就是说，这些端点可以配置为充当事务客户端，它们也可以访问事务资源。

- ActiveMQ
- AMQP
- JavaSpace
- JMS
- JPA

9.3. 声明事务的划分

使用 Blueprint XML 时，您还可以通过在 Blueprint XML 文件中声明事务策略来分离事务。通过将适当的事务策略应用到 bean 或 bean 方法，例如 Required 策略，您可以确保在调用该特定 bean 或 bean 方法时启动事务。在 bean 方法的末尾，事务被提交。这个方法类似于事务在企业 Java Beans 中处理的方式。

OSGi 声明性事务允许您在 Blueprint 文件中的以下范围中定义事务策略：

- 第 9.3.1 节 “bean-level 声明”
- 第 9.3.2 节 “顶级声明”

另请参阅： 第 9.3.3 节 “tx:transaction 属性的描述”。

9.3.1. bean-level 声明

要在 bean 级别上声明事务策略，请插入 tx:transaction 元素作为 bean 元素的子级，如下所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

   <bean id="accountFoo" class="org.jboss.fuse.example.Account">
      <tx:transaction method="*" value="Required" />
      <property name="accountName" value="Foo" />
   </bean>

   <bean id="accountBar" class="org.jboss.fuse.example.Account">
      <tx:transaction method="*" value="Required" />
      <property name="accountName" value="Bar" />
   </bean>

</blueprint>
```

在前面的示例中，所需的事务策略应用于 accountFoo bean 的所有方法，以及 accountBar bean，其中 method 属性指定通配符 *，以匹配所有 bean 方法。

9.3.2. 顶级声明

要在顶层声明事务策略，请插入 tx:transaction 元素作为 blueprint 元素的子，如下所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

   <tx:transaction bean="account*" value="Required" />

   <bean id="accountFoo" class="org.jboss.fuse.example.Account">
      <property name="accountName" value="Foo" />
```

```
    </bean>

    <bean id="accountBar" class="org.jboss.fuse.example.Account">
        <property name="accountName" value="Bar" />
    </bean>

</blueprint>
```

在上例中，Required 事务策略应用到 ID 与模式 匹配的每个 bean 的所有方法。

9.3.3. tx:transaction 属性的描述

tx:transaction 元素支持以下属性：

Bean

（仅限顶级）指定事务策略应用到的 bean ID （命名空间或空格）列表。例如：

```
<blueprint ...>
    <tx:transaction bean="accountFoo,accountBar" value="..." />
</blueprint>
```

您还可以使用通配符字符 *，它在每个列表条目中最多可能会出现一次。例如：

```
<blueprint ...>
    <tx:transaction bean="account*,jms*" value="..." />
</blueprint>
```

如果省略 bean 属性，则默认为 *（匹配蓝图文件中所有非syntic Bean）。

方法

（顶级和 bean-level）指定事务策略应用到的方法名称列表(comma 或空格分开)。例如：

```
<bean id="accountFoo" class="org.jboss.fuse.example.Account">
    <tx:transaction method="debit,credit,transfer" value="Required" />
    <property name="accountName" value="Foo" />
</bean>
```

您还可以使用通配符字符 *，它在每个列表条目中最多可能会出现一次。

如果省略了 method 属性，则默认为 * （匹配适用 Bean 中的所有方法）。

value

（顶级和 bean-level）指定事务策略。策略值与 EJB 3.0 规格中定义的策略具有相同的语义，如下所示：

- 必需 - 支持当前事务；如果不存在，创建一个新事务。
- 必需 - 支持当前事务；如果没有当前事务，则抛出异常。
- RequiresNew - 创建新的事务，挂起当前事务（如果存在）。
- 支持 - 支持当前事务；如果没有存在，则非事务执行。
- NotSupported - 不支持当前事务，而是始终以非事务执行。
- Never - 不支持当前事务；如果当前事务存在，抛出异常。

9.4. 事务传播策略

如果要影响事务客户端创建新事务的方式，您可以使用 JmsTransactionManager 并为它指定一个事务策略。特别是，Spring 事务策略允许您为事务指定传播行为。例如，如果事务客户端是创建新的事务，它检测到已与当前线程关联的事务，如果它继续并创建新的事务，请暂停旧的事务？或者应该让现有交易接管？通过指定事务策略上的传播行为，可以控制这些类型的行为。

事务策略在蓝图 XML 中作为 Bean 进行实例化。然后，您可以通过将其 bean ID 作为 transacted （） DSL 命令的参数来引用事务策略。例如，如果要启动受行为的事务(PROPAGATION_REQUIRES_NEW)，您可以使用以下路由：

```
from("file:src/data?noop=true")
    .transacted("PROPAGATION_REQUIRES_NEW")
    .bean("accountService","credit")
    .bean("accountService","debit")
    .to("file:target/messages");
```

其中 PROPAGATION_REQUIRES_NEW 参数指定使用 PROPAGATION_REQUIRES_NEW 行为配置的事务策略 bean 的 bean ID。请参阅 第 9.4.3 节 “在蓝图 XML 中定义策略 Bean”。

9.4.1. 关于 Spring 事务策略

Apache Camel 可让您使用 org.apache.camel.spring.spi.SpringTransactionPolicy 类来定义 Spring 事务策略，这基本上是一个围绕原生 Spring 类的打包程序。SpringTransactionPolicy 类封装了两部分数据：

- 对 PlatformTransactionManager 类型的事务管理器的引用
- 传播行为

例如，您可以使用 PROPAGATION_MANDATORY 行为实例化 Spring 事务策略 bean，如下所示：

```
<blueprint ...>
  <bean id="PROPAGATION_MANDATORY
"class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager" />
    <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
  </bean>
  ...
</blueprint>
```

9.4.2. propagation 行为的描述

Spring 支持以下传播行为。这些值最初在 JavaEE 支持的传播行为上建模：

PROPAGATION_MANDATORY

支持当前的事务。如果没有当前事务，则抛出异常。

PROPAGATION_NESTED

如果存在当前的事务，请在嵌套的事务中执行，否则行为类似于 PROPAGATION_REQUIRED。

注意

所有事务管理器都不支持嵌套事务。

PROPAGATION_NEVER

不支持当前的事务。如果当前事务存在，则抛出异常。

PROPAGATION_NOT_SUPPORTED

不支持当前的事务。始终以非事务方式执行。

注意

此策略需要暂停当前的事务，此功能不受所有事务管理器的支持。

PROPAGATION_REQUIRED

（默认）支持当前事务。如果不存在，则创建一个新名称。

PROPAGATION_REQUIRES_NEW

创建新的事务，并暂停当前事务（如果存在）。

注意

所有事务管理器都不支持挂起事务。

PROPAGATION_SUPPORTS

支持当前的事务。如果不存在，则以非事务方式执行。

9.4.3. 在蓝图 XML 中定义策略 Bean

以下示例演示了如何为所有支持的传播行为定义事务策略 Bean。为方便起见，每个 bean ID 与传播行为值的指定的值匹配，但实践中，您可以使用您要用于 bean ID 的任何值。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
    <bean id="PROPAGATION_MANDATORY "
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
    </bean>

    <bean id="PROPAGATION_NESTED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_NESTED" />
    </bean>

    <bean id="PROPAGATION_NEVER"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_NEVER" />
    </bean>

    <bean id="PROPAGATION_NOT_SUPPORTED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_NOT_SUPPORTED" />
    </bean>

    <!-- This is the default behavior. -->
    <bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
    </bean>

    <bean id="PROPAGATION_REQUIRES_NEW"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW" />
    </bean>

    <bean id="PROPAGATION_SUPPORTS"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_SUPPORTS" />
    </bean>

</blueprint>
```

注意

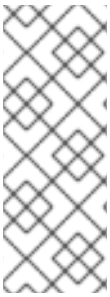
如果要将任何这些 bean 定义粘贴到您自己的蓝图 XML 配置中，请记得自定义对事务管理器的引用。也就是说，将对 txManager 的引用替换为您的事务管理器 bean 的实际 ID。

9.4.4. Java DSL 中带有 PROPAGATION_NEVER 策略的示例路由

演示该事务策略对事务有一些影响的简单方式是将 PROPAGATION_NEVER 策略插入到现有事务的中间，如以下路由所示：

```
from("file:src/data?noop=true")
        .transacted()
        .bean("accountService","credit")
        .transacted("PROPAGATION_NEVER")
        .bean("accountService","debit");
```

以这种方式使用，PROPAGATION_NEVER 策略可避免中止每个事务，从而导致回滚。您应该能轻松查看对应用的影响。



注意

请记住，传递给 transacted （） 的字符串值是一个 bean ID，而不是传播行为名称。在本例中，选择 bean ID 与传播行为名称相同，但这并非总是如此。例如，如果您的应用程序使用多个事务管理器，则最终可能会有多个具有特定传播行为的策略 Bean。在这种情况下，您无法在传播行为后简单地命名 Bean。

9.4.5. 蓝图 XML 中带有 PROPAGATION_NEVER 策略的路由示例

前面的路由可以在 Blueprint XML 中定义，如下所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <from uri="file:src/data?noop=true" />
            <transacted />
            <bean ref="accountService" method="credit" />
            <transacted ref="PROPAGATION_NEVER" />
            <bean ref="accountService" method="debit" />
        </route>
    </camelContext>

</blueprint>
```

9.5. 错误处理和回滚

虽然您可以在事务路由中使用标准 Apache Camel 错误处理技术，但了解例外和事务处理之间的交互非常重要。特别是，您需要考虑抛出异常通常会导致事务回滚。请参见以下主题：

- 第 9.5.1 节 “如何回滚事务”
- 第 9.5.2 节 “如何定义死信队列”
- 第 9.5.3 节 “捕获事务的例外”

9.5.1. 如何回滚事务

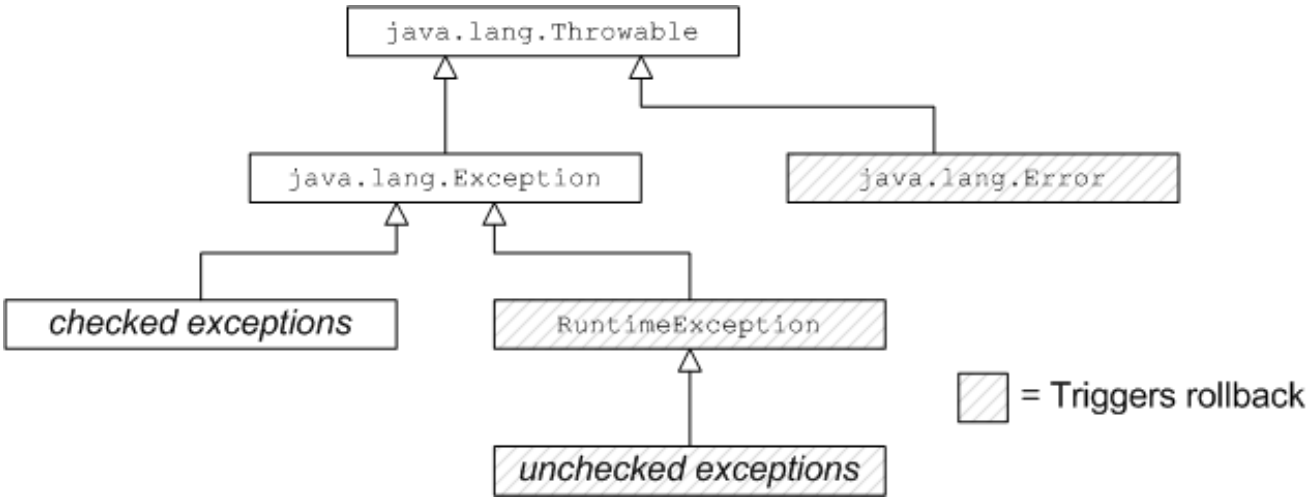
您可以使用以下方法之一回滚事务：

- 第 9.5.1.1 节 “使用运行时例外来触发回滚”
- 第 9.5.1.2 节 “使用 rollback （） DSL 命令”
- 第 9.5.1.3 节 “使用 markRollbackOnly （） DSL 命令”

9.5.1.1. 使用运行时例外来触发回滚

回滚 Spring 事务的最常见方法是抛出运行时（未检查）异常。换句话说，例外是 java.lang.RuntimeException 的一个实例或子类。java.lang.Error 类型的 Java 错误也会触发事务回滚。另一方面，检查异常不会触发回滚。

下图总结了 Java 错误和异常对事务的影响，其中触发回滚的类是灰色的。



注意

Spring 框架还提供 XML 注解系统，允许您指定应或不应触发回滚的例外。详情请参阅 Spring 参考指南中的"回滚"。

警告

如果在事务中处理运行时异常，也就是说，在异常有机会被仔细处理事务时，事务会被回滚。详情请查看 第 9.5.2 节 “如何定义死信队列”。

9.5.1.2. 使用 rollback （） DSL 命令

如果要在 transacted 路由的中间触发回滚，您可以通过调用 rollback （） DSL 命令进行此操作，它会抛出 org.apache.camel.RollbackExchangeException 异常。换句话说，rollback （） 命令使用标准方法抛出运行时异常来触发回滚。

例如，假设您确定在帐户服务应用程序中应有关于资金传输大小的绝对限制。您可以使用以下示例中的代码超过 100 时触发回滚：

```
from("file:src/data?noop=true")
   .transacted()
   .bean("accountService","credit")
   .choice().when(xpath("/transaction/transfer[amount > 100]"))
      .rollback()
   .otherwise()
      .to("direct:txsmall");

from("direct:txsmall")
   .bean("accountService","debit")
   .bean("accountService","dumpTable")
   .to("file:target/messages/small");
```

注意

如果在前面的路由中触发回滚，它将在无限循环中捕获。这是因为 rollback （） 抛出 RollbackExchangeException 异常会在路由开始时传播到 文件 端点。File 组件有一个内置的可靠性功能，可导致它重新发送抛出异常的任何交换。在重新发送课程后，交换仅触发另一个回滚，从而导致无限循环。下一个示例演示了如何避免这种无限循环。

9.5.1.3. 使用 markRollbackOnly （） DSL 命令

markRollbackOnly （） DSL 命令允许您在不抛出异常的情况下强制当前事务回滚。当抛出异常具有不必要的副作用时，这很有用，比如 第 9.5.1.2 节 “使用 rollback （） DSL 命令” 中的示例。

以下示例演示了如何使用 markRollbackOnly （） 命令替换 rollback （） 命令来修改 第 9.5.1.2 节 “使用 rollback （） DSL 命令” 中的示例。此版本的路由解决了无限循环的问题。在这种情况下，当资金传输量超过 100 时，当前的交易将被回滚，但不会抛出异常。由于文件端点没有收到异常，所以它不会重试交换，因此失败的事务会被静默丢弃。

以下代码回滚异常，但 markRollbackOnly （） 命令：

```
from("file:src/data?noop=true")
   .transacted()
   .bean("accountService","credit")
   .choice().when(xpath("/transaction/transfer[amount > 100]"))
      .markRollbackOnly()
   .otherwise()
      .to("direct:txsmall");
...
```

上述路由实施不是理想选择。虽然路由会完全回滚事务（以一致状态保存数据库），并避免无限循环的缺陷，但它不会保留失败事务的任何记录。在现实应用程序中，您通常希望跟踪任何失败的事务。例如，您可能希望向相关客户写入一个字母，以解释事务无法成功的原因。跟踪失败事务的便捷方法是向路由添加死信队列。

9.5.2. 如何定义死信队列

要跟踪失败的事务，您可以定义一个 onException （） 子句，该子句将相关的交换对象分解为死信队列。但是，当在事务上下文中使用时，您需要注意如何定义 onException （） 子句，因为处理异常和事务处理之间的潜在交互。以下示例显示了定义 onException （） 子句的正确方法，假设您需要抑制重新增长异常。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
   ...
   public void configure() {
      onException(IllegalArgumentException.class)
         .maximumRedeliveries(1)
         .handled(true)
         .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
         .markRollbackOnly(); // NB: Must come *after* the dead letter endpoint.
```

```
        from("file:src/data?noop=true")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

在前面的示例中，onException （） 配置为捕获 IllegalArgumentException 异常，并将终止交换发送到死信文件 deadLetters.xml。当然，您可以更改此定义，以捕获应用程序出现的任何异常。改进行为和事务回滚行为的异常由 onException （） 子句中的以下特殊设置控制：

- handled (true) - 阻止重新增长异常。在这个特定示例中，rethrown 异常不可取，因为它会在重新传播到文件端点时触发无限循环。请参阅 第 9.5.1.3 节 “使用 markRollbackOnly （） DSL 命令”。然而，在某些情况下，可能可以接受重新增长异常（例如，如果路由开始时的端点没有实现重试功能）。

- markRollbackOnly （） - 在不抛出异常的情况下标记当前回滚的事务。请注意，在将交换路由到死信队列的 to （） 命令后插入此 DSL 命令非常重要。否则，交换永远不会到达死信队列，因为 markRollbackOnly （） 中断了处理链。

9.5.3. 捕获事务的例外

处理事务路由中的异常不是使用 onException （） 的简单方法是使用路由上的 doTry （） 和 doCatch （） 子句。例如，以下代码演示了如何捕获和处理事务路由中的 IllegalArgumentException，而无需在无限循环中捕获的风险。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
            .doTry()
                .to("direct:split")
            .doCatch(IllegalArgumentException.class)
                .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .end();

        from("direct:split")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
```

```
            .to("file:target/messages");
    }
}
```

在本例中，路由被分成两个片段。第一个片段（来自 file:src/data 端点）接收传入的交换，并使用 doTry （） 和 doCatch （） 执行异常处理。第二个片段（来自 direct:split 端点）执行所有事务处理。如果在这个事务片段中发生异常，它会首先将所有事务传播到 transacted （） 命令，从而导致当前事务被回滚，然后由第一个路由段中的 doCatch （） 子句捕获。doCatch （） 子句不会重新增长异常，因此可以避免文件端点不会执行任何重试和无限循环。