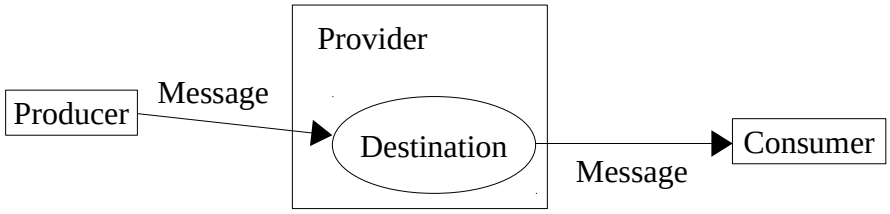
Оглавление

Описание JMS

MOM

Для обмена информацией между слабо-связанными и разнородными системами удобно использовать асинхронный обмен сообщениями: клиенты не обязаны знать ничего ни друг про друга, ни про то, каким образом обрабатываются или генерируются их сообщения на другом конце канала связи. Единственное, что требуется от клиентов в этой схеме — знать формат сообщений и согласовать промежуточный пункт назначения, где они будут их принимать и куда будут отправлять. Примерами таких клиентов могут быть системы, работающие в разном темпе, например, одна регулярно, другая — периодически.

Класс интеграционных систем, обеспечивающих подобный обмен сообщениями, называется MOM (Message Oriented Middleware). Идея MOM состоит в создании общего посредника между клиентами, который и берет на себя все проблемы обмена. Этот посредник называется поставщиком сообщений (Message Provider или Message Broker). Клиент, который отправляет сообщение поставщику, называется производителем сообщения (Message Producer), а клиент, который принимает от поставщика сообщение — потребителем (Message Consumer). Промежуточное хранилище сообщений и порядок обращения к нему через поставщика обобщаются в понятии источника (Message Destination).



Система обмена сообщениями (Message Service) является реализацией MOM, устанавливающей правила доступа к поставщику, регламентирует создание, отправку и получение сообщений. Основными требованиями к системам обмена сообщений являются:

- асинхронность
- слабая связанность клиентов
- надежность обмена

Асинхронность означает временную независимость клиентов обмена, в т.ч. их on-line/off-line режимы. Слабая связанность клиентов обусловлена наличием посредника, устанавливающего относительно универсальные правила приема и отправки сообщений. Надежность системы обмена означает способность обеспечивать доставку сообщений в асинхронных режимах работы, т.е. в условиях задержек или off-line состояний клиентов, при сбоях передачи или обработки сообщений.

JMS

Штатная система сообщений в JEE описывается API JMS, появившимся в 1998 году.

JMS (Java Message Service) – стандартное API системы обмена сообщениями в JEE. JMS 1.1 был штатной системой сообщений JEE до версии JEE7, где его сменил JMS 2.0. Вообще API JMS вылазит за пределы JEE-контейнеров и требует от вендоров также и реализации функционала для работы в JSE.

JMS является вендорно-независимым API, которое может использоваться для доступа к более общему классу EIS (Enterprise Information System, промышленные системы обмена сообщениями). JMS играет роль JDBC, обеспечивая приложению вендорно-нейтральный доступ к EIS через стандартное JMS API.

В JEE-контейнеры исторически большинство вендоров встраивало собственную систему сообщений, оставляя возможность интеграции внешних продуктов. Несмотря на это, экспертной группой было решено гарантировать разработчикам EJB наличие реализации штатной JMS, обеспечивающей отправку и прием сообщений. Все реализации контейнеров EJB 3.0 должны поддерживать JMS-поставщика.

Реализацию JMS также называют брокером (Message Broker) или Message Router. Все типы EJB могут использовать JMS для отправки сообщений. Эти сообщения далее могут потребляться другими приложениями или MDB.

В архитектуре JMS используются следующие термины:

- JMS-клиент (JMS Client) — приложение, использующее JMS, объединяет и производителя и потребителя сообщений;
- JMS-производитель (JMS Producer) – JMS-клиент, отправляющий сообщения;
- JMS-потребитель (JMS Consumer) – JMS-клиент, принимающий сообщения;
- JMS-поставщик (JMS Provider) — реальная система, которая управляет отправкой и доставкой сообщений согласно JMS API;
- JMS-сообщение (JMS Message) — объект, который JMS-клиент отправляет или получает от поставщика сообщений;
- JMS-источник (JMS Destination) — хранилище поставщика для сообщений, которые ему отправляют производители и принимают потребители. В JMS существует 2 типа источников: темы (Topic) и очереди (Queue).
- Администрируемые объекты JMS — фабрики соединений (Connection Factory) и источники (Destination), которые создаются брокером административно, настраиваются и помещаются в JNDI-дерево для последующих JNDI-запросов или внедрения в run-time;
- JMS-приложение — бизнес-система, состоящая из одного или более JMS-клиента и одного (обычно) или более JMS-поставщика.

Rem: любой JMS-клиент может одновременно быть и производителем и потребителем сообщений.

Rem: различные Message Broker появились десятки лет назад, наиболее популярными были IBM MQ Series. JMS появился относительно недавно, он специально разрабатывался для

обмена различными сообщениями между java-приложениями.

JMS 2.0 появился в JEE7, сменив JMS 1.1. Из нового появилась поддержка интерфейса AutoCloseable для администрируемых объектов. К классическому API добавлен новый упрощенный API (JMSContext и т.д.). Добавлены системные исключения JMSRuntimeExcepiton. Добавлены методы асинхронной отправки сообщений с использованием интерфейса прослушивателя MessageListener.

API JMS требует реализации, т.е. JMS-поставщика. Эталонным (RI) JMS-поставщиком является система Open Message Queue (Open MQ), входящая в состав сервера приложений GlassFish.

Асинхронность JMS

Принципиальной чертой JMS является асинхронность. В отличие от стандартного взаимодействия с сессионными EJB по RMI или HTTP (WS-представление EJB), где каждый вызов блокирует поток до полного выполнения запроса, JMS-клиент не ждет ответа на посланный им запрос. Т.е. под асинхронностью подразумевается временная независимость производителя и потребителя сообщения, при этом связь клиентов с источником может быть как асинхронной, так и синхронной.

Клиенты-производители отправляют запросы в т.н. темы (topic) или очереди (queue), откуда их могут получить другие клиенты-потребители. При этом ответственность за доставку лежит целиком на брокере, а отправитель продолжает выполнять свою работу безотносительно потребителя.

В некоторых случаях использование JMS выгоднее использования RMI. Получив от JMS-поставщика по JNDI JMS-соединение, EJB-компонент может использовать JMS для доставки сообщений другим Java-приложениям асинхронным образом.

В качестве примера можно рассмотреть регистрацию: пусть приложению требуется оповещать новых пользователей при их регистрации, но при этом не отвлекаться на это. Регистрацией может заниматься некоторый сессионный EJB — UserRegistrationEJB, отсылающий JMS-сообщения о завершении некоторой системе, слушающей регистрационные события. При этом сам UserRegistrationEJB может сразу же продолжать работу, не дожидаясь подтверждения приема или ответа. В качестве системы, прослушивающей регистрационные события-сообщения, может выступать MDB, другой EJB или вообще приложение на другой платформе, заинтересованное в оповещении о результате регистрации. Пример может включать рассылку электронных писем для подтверждения регистрации, включение пользователя в списки рассылки, причем все это может делаться отдельно от основного потока регистрации, не влияя на него.

JMS-клиенты не общаются напрямую, а используют для этого JMS-поставщика. Общение между JMS-клиентами и JMS-поставщиком уже более полноценно.

Контекст безопасности не передается от отправителя сообщений к получателю. Хотя JMS-поставщик может аутентифицировать пользователя по его сообщению, он не передаст эту информацию потребителю. Это связано с тем, что архитектура MOM рассчитана на разнородных производителей, которые могут работать на множестве платформ и использовать совершенно разные схемы безопасности, поэтому стандартизовать это в JEE тяжело.

Транзакции также не распространяются от отправителя получателю, т.к. неясно, кто будет получать очередное сообщение. А распределенная по тысяче получателей транзакция не является разумной. К тому же неясно, какие задержки будут при получении, будет ли вообще доступен получатель и т.д. Такое поведение с возможностью подвисания противоречит концепции оперативного выполнения транзакции, в процессе которой могут блокироваться общеприкладные и системные ресурсы. Но распределенные транзакции возможны между JMS-клиентами и JMS-поставщиком. Например, JMS-поставщик может отменить рассылку сообщений, если у отправителя произошел сбой и он решил откатить транзакцию.

Модели обмена JMS

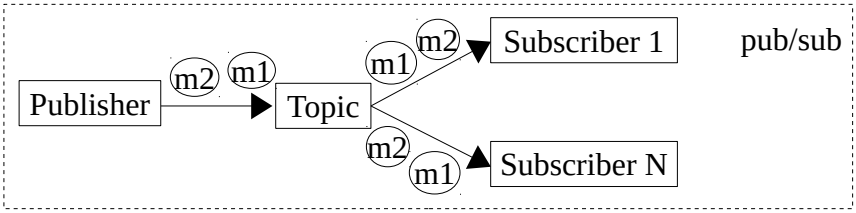
В JMS существует 2 модели обмена сообщениями (в спеке JMS 1.1 это messaging domains).

- publish-and-subsribe (pub/sub) — публикация/подписка
- point-to-point (p2p) — точка-точка

С JMS 1.1 введено общее API для обеих моделей: pub/sub и p2p.

Модель pub/sub (JMS Topic)

Модель pub/sub используется для рассылок 1:N.



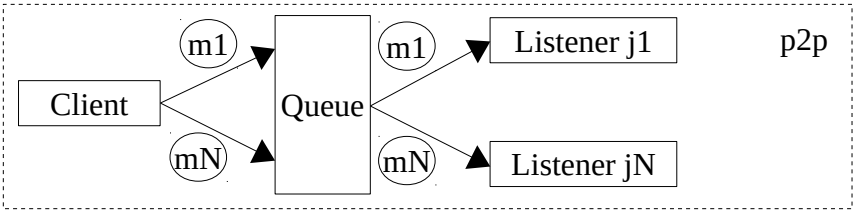
В модели публикация/подписка (pub/sub) производитель (producer) отправляет сообщение в виртуальный канал, называемый темой (topic). Потребители (consumers) могут подписать на эту тему и автоматически получать из нее сообщения. Такую систему рассылки называют еще основанной на проталкивании (push-based): потребителю не нужно слушать или запрашивать тему (topic) для получения сообщения, он делегирует это самой теме (topic) поставщика.

Производители сообщений не зависят от их потребителей. Потребляющие JMS-

клиенты, использующие тему (topic), могут устанавливать длительную подписку, в течении которой могут отсоединяться и подсоединяться вновь, собирая непрочитанные сообщения. Причем сообщения каждому из подписчиков приходят в неопределенном порядке

Модель p2p (JMS Queue)

Модель p2p используется для сообщений, обрабатываемых однократно 1:1.



Модель точка-точка (p2p) позволяет клиентам отправлять и принимать сообщения в синхронном и асинхронном режимах через виртуальный канал, называемый очередью (queue). Модель точка-точка (p2p) по типу является системой опросного типа (pull/polling-based model), где сообщение надо запрашивать из очереди. К очереди может быть подключено множество потребителей, но каждое сообщение может быть прочитано только одним из них, после чего оно удаляется. При этом название «очередь» имеет весьма символический характер: в JMS-спецификации не оговорен порядок распределения доступа потребителей к очереди, JMS гарантирует лишь то, что каждое сообщение будет читать единственный потребитель.

Производитель и потребитель никак не связаны во времени, потребитель может получать сообщения, присланные до его создания.

Схемы использования JMS

Модель pub/sub применяется там, где производителю нужно передавать копии сообщения n>>1 пользователям. И при этом его не должна заботить степень готовности потребителей принять эти сообщения. Также используя различные темы (topic) можно организовывать фильтрацию сообщений даже в схемах общения 1:1. При этом потребитель организовывает прием каждого вида сообщений через отдельный свой прослушиватель onMessage(). Классическим примером pub/sub является обычная почтовая рассылка подписчикам каких-то новостей.

Модель p2p применяется для схемы общения 1:1, между определенными клиентами, когда каждое сообщение имеет реальное значение. Также может иметь значение диапазон и вариативность данных в сообщениях. Радикальное отличие p2p от pub/sub в однократном потреблении каждого сообщения. Это может быть особенно важно тогда, когда необходимо обрабатывать сообщения отдельно, но последовательно. Классическим примером очереди сообщений (p2p) является минное поле, где на каждую мину может наступить любой, но только один раз. Или менеджер, победивший

в драке за очередной звонок клиента.

Модель запрос-ответ предполагает определенную (событийную) синхронизацию: после доставки запроса на 1-м этапе обмена, производитель и потребитель меняются местами с целью доставки ответа. Часто ее реализуют при помощи p2p или pub/sub. Например, для ответа организуется ответная очередь, которую слушает производитель запроса, привязка ответа к запросу производится меткой сообщений correlationID.

Rem: спека JMS 1.1 не регламентирует конкретно как должны быть реализованы модели pub/sub и p2p. Каждая из моделей может использовать систему push или pull, но концептуально для pub/sub ближе push, а для p2p ближе pull.

Применение JMS. Администрируемые объекты

Администрируемые объекты — объекты, которые создаются вне приложения (административно) для дальнейшего программного применения в run-time. Брокер (реализация JMS) создает и настраивает эти объекты один раз, после чего помещает их в JNDI-дерево, доступное клиентам. Существует 2 вида администрируемых объектов:

- Connection Factory – фабрики соединений, которые используются клиентами для подключений к источникам
- Destination – источники, которые получают, хранят и распространяют сообщения клиентов. В JMS существуют источники 2-х видов: Topic (тема, pub/sub) и Queue (очередь, p2p).

Такие объекты создаются вне приложения при помощи JMS-поставщика (сервера приложений в JEE). В JEE (WEB и EJB контейнерах) допускается также их задание при помощи метаинформации с последующим созданием JEE-контейнером в deploy-time .

Создание администрируемых объектов JMS в консоли GlassFish

Можно сделать это через консоль Glassfish: перейти в его каталог bin, запустить там консольный скрипт asadmin и выполнить в нем команды:

```
asadmin> create-jms-resource –restype javax.jms.ConnectionFactory xxx/ExFactory
asadmin> create-jms-resource –restype javax.jms.Topic xxx/ExTopic
asadmin> create-jms-resource –restype javax.jms.Queue xxx/ExQueue
```

Для просмотра созданных администрируемых объектов (ресурсов JMS) можно использовать команду:

```
asadmin> list-jms-resources
```

Задание администрируемых объектов JMS аннотациями

Администрируемые объекты в JMS 2.0 также можно определять посредством метаинформации, в аннотациях @javax.jms.JMSConnectionFactoryDefinitiion и @javax.jms.JMSDestinationDefinition, помечающих класс MDB :

```
Ex: пример в Glassfish 4.1

@JMSDestinationDefinition(name = "java:app/ExTopic", interfaceName = "javax.jms.Topic",
resourceAdapter = "jmsra", destinationName = "ExTopic")
@JMSConnectionFactoryDefinition(name = "jms/example/ExConnectionFactory", className =
"javax.jms.ConnectionFactory")
@MessageDriven(…)
public class ...
```

Rem: GlassFish 4.1 при использовании этих аннотаций требует применения портируемых глобальных JNDI-имен. JSE-клиент почему-то не мог найти через JNDI-запрос созданные таким образом объекты.

Клиентский доступ к администрируемым объектам JMS

Клиент получает доступ к этим объектам через интерфейсы JMS запросами JNDI или внедрением. Т.е. работает с ними через прокси.

Объекты, осуществляющие обмен, производятся администрируемыми объектами «по-ссылке» (?). Т.е. они также являются прокси соответствующих объектов брокера. И клиентские обращения к их методам — это обращение к методам соответствующих объектов JMS-поставщика, делегировавшего управление клиенту.

Применение JMS. JMS API

JMS API служит для соединения приложения с JMS-поставщиком. JMS API позволяет программно создавать, модифицировать, отправлять и принимать асинхронным образом JMS-сообщения, содержащие текст или другие объекты.

Классический интерфейс JMS позволяет соединиться с JMS-поставщиком (ConnectionFactory и Connection), организовать сессию (Session), в которой будет производиться создание сообщений (Message) и обмен ими (MessageProducer, MessageConsumer) через источник (Destination). По аналогии с базами данных JMS-поставщик играет роль РБД, а ConnectionFactory, порождаемый ею Connection и Session являются аналогом JDBC, предоставляющими транзакционно-управляемые соединения.

В JMS 2.0 в дополнение к классическому API добавлен упрощенный. Главным образом введен интерфейс однопоточного обмена JMSContext, который заменил собою классические Connection и Session. Потребитель JMSConsumer получил дополнительную возможность асинхронного приема сообщений из источника посредством регистрации прослушивателя MessageListener. Также упрощенный API

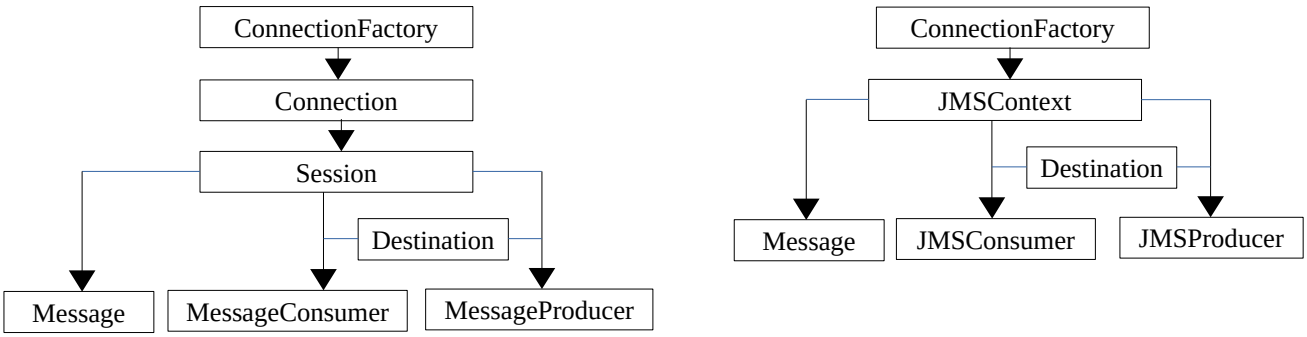
реализует шаблон текучего интерфейса, что позволяет одной строкой описывать настройку, прием или отправку сообщения.

Rem: также доступен еще и устаревший API (не рассматривается), разделенный на модели pub/sub (Topic) и p2p (Queue).

Состав API JMS (пакет javax.jms):

Классический API (JMS 1.1)	**Упрощенный API (JMS 2.0)**
ConnectionFactory	ConnectionFactory
Connection	JMSContext
Session	JMSContext
Destination	Destination
Message	Message
MessageConsumer	JMSConsumer
MessageProducer	JMSProducer
JMSException	JMSRuntimeException

Схемы создания объектов JMS в классическом и упрощенных API:



Rem: клиентские объекты Connection, Session, Message, MessageProducer, MessageConsumer, JMSContext, JMSConsumer, JMSProducer, порожденные от прокси фабрики и источника, сами являются прокси соответствующих объектов JMS-поставщика. Обращение к их методам — в реальности обращение к JMS-поставщику.

ConnectionFactory (фабрика соединений)

Интерфейс javax.jms.ConnectionFactory описывает JMS-фабрику соединений — администрируемый объект, инкапсулирующий параметры конфигурации JMS-поставщика. Эта фабрика создает объекты классического интерфейса javax.jms.Connection или упрощенного интерфейса JMSContext, программно связывающие приложение с JMS-поставщиком.

Получить объект фабрики (прокси объекта фабрики JMS-поставщика) можно через внедрение в MBean или прямого JNDI-запроса.

```
Ex: примеры получения экземпляров фабрики ConnectionFactory

@Resource(name="ExFactoryName")
private ConnectionFactory connectionFactory;
…
@Resource(lookup="jndiName")
private ConnectionFactory connectionFactory;

@Inject
@JMSConnectionFactory(“jndiName”)
private ConnectionFactory connectionFactory;
…
javax.naming.Context ctx = new InitialContext(…);
ConnectionFactory connectionFactory = (ConnectionFactory) javax.naming.Context.lookup("jndiName");
```

Destination (источник)

Интерфейс javax.jms.Destination описывает администрируемый объект, содержащий информацию (тип и адрес) о источнике JMS-поставщика, который играет роль буфера, в котором накапливаются все потребляемые и производимые клиентами сообщения. Т.о. Destination работает как виртуальный канал между приложениями, позволяющий разделить их и абстрагироваться от сети.

В JMS существует два типа источников — Topic (тема, pub/sub) и Queue (очередь, p2p). Получить объект темы или очереди (прокси объекта темы или очереди JMS-поставщика) можно через внедрение в MBean при помощи @Resource или прямого JNDI-запроса.

```
Ex: примеры получения экземпляров очереди

@Resource(name="ExQueueName")
private Destination queue;
…
@Resource(mappedName="jndiName")
private Destination queue;
…
javax.naming.Context ctx = new InitialContext(…);
Destination queue = (Destination) ctx.lookup("jndiName");
```

Классическое API. Connection & Session (соединение и сессия)

Фабрика ConnectionFactory вызовами createConnection(...) умеет создавать объекты классического интерфейса javax.jms.Connection, которые отвечают за реальное соединение с JMS-поставщиком. Connection спроектирован потокобезопасным для параллельного доступа, поскольку каждое соединение дорого с т.з. вычислительных

ресурсов.

Для организации работы в рамках одного потока, соединением Connection создается объект сессии, описываемый интерфейсом javax.jms.Session. Одному соединению Connection может соответствовать одна или несколько сессий (по одной на каждый поток):

- в EJB-контейнере и WEB-контейнере многопоточная работа не допускается, т.е. соединение Connection может создавать активные (не закрытые close()) сессии только по-одиночке, иначе будет выброшено JMSException;
- в JSE или ACC-контейнере, при необходимости параллельной посылки и приема сообщений в одном соединении, для каждого потока надо создавать свою сессию.

Объект Session порождает объекты Message, MessageProducer и MessageConsumer, при помощи которых организуется обмен сообщениями. Сессионные операции обмена могут группироваться в единицу работы (UOW) и выполняться в рамках транзакции, также могут быть определены различные режимы подтверждения приема сообщений.

Это поведение зависит от окружения и параметров createSession(boolean transacted, int acknowledgeMode) создания соединением сессии. Первый параметр определяет транзакционное поведение, второй задает режим подтверждения:

- в EJB или WEB контейнерах при наличии активной глобальной контейнерной транзакции (CMT) параметры создания сессии игнорируются контейнером, все сообщения отправляются, а прием входящих автоматически подтверждается только при ее фиксации, при откате же ее ничего не посылается и не подтверждается. В отсутствии активной CMT, подтверждение приема зависит от допустимых режимов - AUTO_ACKNOWLEDGE и DUPS_OK_ACKNOWLEDGE, посылка сообщений зависит от наличия BMT.
- в JSE или ACC-контейнере в режиме transacted = true JMS-поставщик автоматом создает локальную транзакцию при получении клиентом объекта сессии Session. Этой транзакцией можно управлять посредством вызовов commit() и rollback() интерфейса сессии. При этом отправка сообщений и автоматическое подтверждение будут проводиться только при фиксации транзакции, а при ее откате ничего делаться не будет. В случае нетранзакционного режима transacted = false отправка сообщений пойдет без задержек, а подтверждение будет определяться допустимыми режимами: AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE

```
Connection con = connectionFactory.createConnection();
Session session = connect.createSession(transacted, acknowledgeMode);
```

Значения параметра транзакционного режима — transacted (применим в JSE и ACC-контейнере):

- true – создание локальной транзакции
- false – отказ от транзакции

Значения параметра режима подтверждения сообщений — acknowledgeMode:

- Session.AUTO_ACKNOWLEDGE — немедленное (трудолюбивое, eager) подтверждение. В режиме автоподтверждения успешный прием сообщения вызовом receive() интерфейса потребителя или асинхронным прослушивателем MessageListener потребителя будет подтверждаться автоматически без задержек.
- Session.DUPS_OK_ACKNOWLEDGE — отложенное (ленивое, lazy) подтверждение. В режиме подтверждения с допустимым дублированием допускается автоматическое подтверждение приема с задержкой, что может вызвать в ответ на простой повторную посылку дубля сообщения JMS-поставщиком. Используется для разгрузки сети, потребитель должен уметь работать с дублями
- Session.CLIENT_ACKNOWLEDGE — клиентское подтверждение, путем программного вызова метода acknowledge() объекта Message клиентом.

Rem: в спеке EJB 3.2 рекомендуется использовать createSession() для EJB и WEB контейнеров при наличии CMT-транзакции. Поскольку параметры будут проигнорированы контейнером. BMT-транзакция не включается в транзакцию доставки, но распространяется на JMS-поставщика при отправке сообщений.

Перед получением сообщений потребитель должен запустить соединение вызовом start() объекта Connection, а для временной приостановки приема сообщений — использовать вызов stop(). Закрыть соединение можно вызовом close(). Начиная с JMS 2.0 (EJB 3.2, JEE7) Connection и Session реализуют интерфейс java.lang.AutoCloseable, т.е. объекты Connection можно использовать в конструкции try-с-ресурсами и не закрывать соединения вручную вызовом close().

Пример автономного JMS-потребления с автозакрытием в try:

```
try(Connection con = connectionFactory.createConnection();
    Session session = connect.createSession(true,0)) {
  …
}
```

Классическое API. MessageProducer (производитель сообщений)

Объект javax.jms.MessageProducer создается объектом Session. Он отправляет сообщения от вызывающей стороны к принимающей, которая описывается объектом источника (Destination).

Ex: отправка текстового сообщения в источник

```
MessageProducer mp = session.createProducer(destination);
```

```
TextMessage tm = session.createTextMessage();
tm.setText("Text example!");
mp.send(tm);
```

Классическое API. MessageConsumer (потребитель сообщений)

Объект javax.jms.MessageConsumer создается объектом Session, при создании указывается объект источника Destination. Любой JMS-клиент, подписанный на тему (Topic) будет получать копии сообщений, а для JMS-клиентов, стоящих в очереди (Queue) будут изыматься из нее "оригиналы".

```
Ex: явное получение сообщений из очереди путем циклического опроса источника

MessageConsumer consumer = session.createConsumer(destination);
connection.start();
while(true){
  TextMessage message = (TextMessage) consumer.receive();
  System.out.println(tm.getText());
}
```

Упрощенное API. JMSContext (контекст JMS 2.0)

Этот интерфейс упрощенного API JMS 2.0 является гибридом классических Connection и Session. Т.е. реализует однопоточное соединение с JMS-поставщиком, как раз подходящее для EJB и WEB контейнеров.

Объект JMSContext может быть получен клиентом одним из вызовов производящих методов createContext(...) администрируемого объекта ConnectionFactory. В EJB и WEB контейнерах также можно внедрить объект JMSContext при помощи @Inject.

Rem: внедренный или произведенный фабрикой клиентский объект JMSContext будет являться прокси, представляющим объект JMS-поставщика. Т.е. все обращения к его методам будут транслироваться JMS-поставщику.

В вызов createContext(int sessionMode) можно передать целочисленный параметр сессионного режима sessionMode, который объединяет классические режимы transacted и acknowledgeMode:

- для JSE и ACC-контейнера доступны режимы sessionMode, определенные константами JMSContext: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE. В транзакционном режиме JMS-поставщик оборачивает в локальную транзакцию отправку и подтверждение сообщений, которые будут выполняться только при успешной ее фиксации через вызов commit() интерфейса контекста, а в случае сбоя и отката вызовом rollback() ничего делаться не будет. Остальные режимы нетранзакционные, сообщения будут отправляться в них без задержки, а прием

сообщений будет подтверждаться перед JMS-поставщиком также, как и для аналогичных режимов классического Session.
- в EJB или WEB контейнерах при наличии активной глобальной контейнерной транзакции (CMT) параметр sessionMode игнорируются контейнером, все сообщения отправляются, а прием входящих автоматически подтверждается только при ее фиксации, при откате же ее ничего не посылается и не подтверждается. В отсутствии активной CMT, подтверждение приема зависит от допустимых режимов - AUTO_ACKNOWLEDGE и DUPS_OK_ACKNOWLEDGE, посылка сообщений зависит от наличия BMT.

Далее объект JMSContext может использоваться для получения объектов Message стандартных типов, производителя JMSProducer и потребителя JMSConsumer, работающих в одном потоке и являющихся представителями (прокси) соответствующих объектов JMS-поставщика.

Для начала приема сообщений используется вызов start(), для приостановки — вызов стоп, для закрытия контекста — вызов close() интерфейса JMSContext.

В транзакционном режиме локальную транзакцию, открытую при создании объекта JMSContext, фиксирует вызов commit(), а откатывает вызов rollback().

Упрощенное API. JMSProducer

Объект javax.jms.JMSProducer создается вызовом createProducer() интерфейса JMSContext. Он позволяет задать заголовок get/set[Xxx] и свойства get/set[Xxx]Property сообщения и отправить его (Message) в источник (Destination), используя вызов send(…). Отдельное задание свойств и заголовка сообщения в производителе позволяет использовать их для заполнения самого сообщения на стороне JMS-поставщика, поскольку клиентский производитель является прокси JMS-поставщика, которому в реальности и перенаправляются все вызовы.

```
context.createProducer().send(queue, "Message send " + new Date());
```

Упрощенное API. JMSConsumer

Объект javax.jms.JMSConsumer создается одним из вызовов createConsumer(…) интерфейса JMSContext, куда можно передать в качестве параметров источник и селектор сообщений. Для явного (синхронного) приема сообщения из источника используется вызов receive(…), вызов типизированного метода <T> T receiveBody(Class<T> c) принимает сообщение и отдает его тело, приведенное к указанному типу. Для асинхронного (в другом потоке) принятия сообщений можно зарегистрировать вызовом setMessageListener(MessageListener ml) прослушиватель сообщений, чей метод onMessage() и будет дергаться JMS-поставщиком при доставке сообщений потребителю.

Ex: фрагмент циклического опроса источника потребителем в JSE

```
while (true) {
  String message = context.createConsumer(destination).receive Body (String.class);
}
```

Ex: асинхронное потребление сообщений через прослушиватель в JSE

```
// Класс асинхронного прослушивателя
public class Listener implements MessageListener {
@Override
  public void onMessage(Message message) {
    System.out.println("Asynchronous message is received: " + message.getBody(String.class));
  }
}

// Фрагмент задания прослушивателя сообщений Listener в потребителе
try (JMSContext context = connectionFactory.createContext()) {
  context.createConsumer(queue).setMessageListener(new Listener());
}
```

Message (сообщение)

Единицей обмена являются объекты интерфейса javax.jms.Message, называемые сообщениями. Производитель посылает их в источник, а потребитель их оттуда получает. Каждое сообщение JMS можно условно разбить на 3 части:

- заголовок (header) — совокупность предопределенных именованных свойств (get/set) Message, в которых хранится стандартизованная информация для доставки и идентификации сообщения
- свойства (properties) — пары имя-значение, несущие произвольную прикладную информацию простых типов, в частности, позволяющие источнику фильтровать сообщения по своим значениям
- тело (body) — полезная прикладная нагрузка сообщения, записанная в одном из допустимых форматов

Все части (свойства) заголовка доступны через get/set-методы Message. Они либо устанавливаются клиентом вручную в объекте Message, либо JMS-поставщиком в клиентском вызове (через прокси производителя) его метода send(), либо самостоятельно JMS-поставщиком.

Ex: установка в заголовке JMSCorrelationID клиентом вручную

```
Message msg = jmsContext.createTextMessage(“Test text”);
msg.setJMSCorrelationID(“xxx”);
```

Ex: установка в заголовке приоритета через интерфейс производителя

```
jmsContext.createProducer().setPriority(4).send(destination, msg);
```

Часть заголовка	**Описание**	**Установщик**
JMSDestination	источник	send(...)
JMSDeliveryMode	Указывает JMS-поставщику режим доставки: PERSISTENT — сохранять сообщение в ROM в рамках операции send() для восстановления при сбоях; NON_PERSISTENT — экономный режим, не сохраняющий сообщение в ROM, гарантирующий одноразовость (без дублей при сбое)	send(...)
JMSMessageID	UID сообщения, посылаемого поставщиком. Уникальность должна обеспечиваться в рамках сообщений, передаваемых подсоединенными роутерами (брокерами) сообщений используемого JMS-поставщика. UID должен ставить клиент, некоторые JMS-поставщики допускают null и ставят его сами для экономии трафика	send(...)
JMSTimeStamp	Время поступления сообщения поставщику, msec.	send(...)
JMSCorrelationID	Метка для связывания сообщений. Полезна в режиме запрос-ответ	клиент
JMSReplyTo	источник (Destination) для перенаправления ответа. Полезен в режиме запрос-ответ, для указания адресата ответа	клиент
JMSRedelivered	Флаг повторной доставки, сообщающий потребителю, что, возможно, это сообщение уже посылалось ему ранее, но тогда подтверждения приема не последовало	поставщик
JMSType	Вендор-специфический идентификатор типа сообщения	клиент
JMSExpiration	Время истечения сообщения, устанавливаемое JMS-поставщиком путем сложения текущего времени получения и времени жизни сообщения, установленного клиентом	send(...)
JMSPriority	Приоритет сообщения. Может влиять на скорость доставки сообщений JMS-поставщиком. Всего 10 уровней: 0-4 нормальные, 5-9 срочные	send(...)

Свойства Message — это дополнительные типизированные свойства, дополняющие стандартные именованные свойства заголовка сообщения. Они несут прикладную информацию простых типов boolean, byte, short, int, long, float, double, String. Доступ к ним обеспечивают методы Message setXxxProperty(name,value) и getXxxProperty(name), где Xxx — один из простых типов, name — строковое имя, а value — значение типа Xxx.

```
message.setFloatProperty(“price”,12.5);
float price = message.getFloatProperty(“price”);
```

Тело Message — полезная нагрузка сообщения в заданном формате: текст, параметры, сериализуемые объекты, байтовые потоки и т.п. Для каждого типа предусмотрен свой тип сообщения: TextMessage, MapMessage, ObjectMessage, StreamMessage, BytesMessage.

Они создаются в JMSContext и Session вызовами createXxxMessage, где Xxx — Text, Map, Object, Strem, Bytes. И заполняются вызовами соответствующих типу set-методов.

```
Ex: фрагмент создания сообщения для регистрации пользователя

//DTO User
User user = getUser();
MessageProducer producer = session.createProducer(topic);
MapMessage mapMsg = session.createMapMessage();
mapMsg.setInt("userId", user.getUserId());
mapMsg.setString("firstname", user.getFirstname());
mapMsg.setString("email", user.getEmail());
producer.send(mapMsg);
```

```
…
ObjectMessage objectMsg = jmsContext.createObjectMessage();
ObjectMsg.setObject(user);
producer.send(objectMsg);
```

При получении сообщения Message из источника, в классическом API надо привести его к надлежащему типу и получить тело вызовом однjuj из соответствующих get-методов. В упрощенном интерфейсе Message JMS 2.0 появился типизированный метод извлечения тела <T>T getBody(Class<T> c). При этом тело сообщения доступно только для чтения (?)объект Message является прокси объекта JMS-поставщика(?).

```
(?) Примеры получения тел сообщений в классическом API JMS 1.1
ObjectMessage objectMsg = (ObjectMessage) message;
User user = (User) objectMsg.getObject();
TextMessage textMsg = (TextMessage) message;
String text = textMsg.getText();
```

```
(?) Примеры получения тела сообщений в упрощенном API JMS 2.0
User user = (User) message.getBody(Serializable.class);
String text = message2.getBody(String.class);
```

Сообщение StreamMessage передает в качестве полезной нагрузки (тела) содержимое своего потока. Сообщение BytesMessage передает массив байт, который можно интерпретировать как черный ящик.

Применение JMS. Производство и отправка сообщений

Классическое JMS 1.1 API в JSE

Для начала клиенту надо получить через запросы JNDI администрируемые JMS-поставщиком объекты фабрики соединений ConnectionFactory и источника Destination.

Затем при помощи них создаются и настраиваются объекты Connection, Session, MessageProducer, Message, которые на самом деле являются прокси JMS-поставщика.

В объект соединения Connection при создании можно передать учетную информацию. Объект сессии Session порождается соединением и обеспечивает однопоточную отправку сообщений путем создания экземпляров производителей и сообщений. При создании сессии указывается ее транзакционный режим и режим подтверждения приема сообщений.

В объект производителя MessageProducer может зашиваться при создании адрес источника. Через производитель можно передать JMS-поставщику часть информации о заголовке и свойствах сообщений, при помощи которых поставщик будет дозаполнять отправленные в источник сообщения. Объект сообщения Message частично заполняется клиентом (тело, свойства, часть заголовка), частично JMS-поставщиком, при перенаправлении ему клиентского вызова send(…) производителя MessageProducer.

Сообщения передаются производителем в источник по-разному в зависимости от транзакционного режима соединения: либо сразу вызовом send() для transacted = false, либо для transacted = true при фиксации транзакции вызовом commit(), а при ее откате вызовом rollback() не передается ничего.

После окончания отправки сообщений стоит закрыть соединение Connection вызовом close() или использовать конструкцию try-with-resources, доступную с JEE7. При использовании классического API JMS 1.1 надо обрабатывать проверяемые исключения класса JMSException.

Ex.: стиль JMS 1.1 в JSE

```
public class Producer {
  public static void main(String[] args) {
     try {
      Context jndiContext = new InitialContext(...);
      ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("xxx/Factory");
      Destination queue = (Destination) jndiContext.lookup("xxx/Queue");
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageProducer producer = session.createProducer(queue);
      TextMessage message = session.createTextMessage("Mesage is sended " + new Date());
      producer.send(message);
      connection.close();
    } catch (NamingException | JMSException ex) {
       ex.printStackTrace();
```

```
    }
  }
}
```

Упрощенное JMS 2.0 API в JSE

Аналогично классике, клиент должен получить через JNDI администрируемые JMS-поставщиком объекты фабрики соединений ConnectionFactory и источника Destination.

Затем при помощи них создаются и настраиваются объекты JMSContext, JMSProducer, Message, которые на самом деле являются прокси объектов JMS-поставщика.

В объект однопоточного контекста JMSContext при его создании вызовом createContext(int sessionMode) фабрики зашивается сессионный режим sessionMode: транзакционное поведение, либо различные нетранзакционные режимы. Контекст создает производителя JMSProducer. Производитель может передать JMS-поставщику часть информации о заголовке и свойствах сообщений, при помощи которых поставщик будет дозаполнять отправленные в источник сообщения. Объект сообщения Message частично заполняется клиентом (тело, свойства, часть заголовка), частично JMS-поставщиком через клиентский вызов send(…) интерфейса производителя.

Сообщения передаются производителем в источник по-разному в зависимости от транзакционного режима сессии. Для sessionMode = SESSION_TRANSACTED отправка происходит при фиксации транзакции вызовом commit(), а при ее откате вызовом rollback() не передается ничего. Для остальных режимов отправка идет сразу при вызове send().

После окончания отправки сообщений стоит закрыть соединение JMSContext вызовом close() или использовать конструкцию try-with-resources, доступную с JEE7. При работе с упрощенным API JMS 2.0 необходимость обработки непроверяемых JMSRuntimeException отпадает.

Ex.: стиль JMS 2.0 в JSE

```
public class Producer {
  public static void main(String[] args) {
    try {
      Context jndiContext = new InitialContext(...);
      ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("xxx/Factory");
      Destination queue = (Destination) jndiContext.lookup("xxx/Queue");
      try (JMSContext context = connectionFactory.createContext()) {
        context.createProducer().send(queue, "Message is sended " + new Date());
      }
    } catch (NamingException ex) {
      ex.printStackTrace();
    }
  }
```

```
}
```

Упрощенное JMS API в JEE. @Resource

Принцип работы подобен варианту в JSE, но проще за счет автоматизации.

Контейнеры JEE дают возможность заменить JNDI-вызовы внедрением. Для внедрения администрируемых JMS-поставщиком фабрики соединений и источника можно применять специализированную аннотацию @Resource, появившуюся в JEE5. Это позволяет разработчику компонента освободиться от тягот настройки JNDI-контекста и от обработки исключений NamingException, поскольку отвечает за внедрение контейнер.

Более серьезные отличия в транзакциях для EJB и WEB контейнера. При наличии активной глобальной (JTA) транзакции (CMT или BMT) все сообщения отправляются при ее фиксации, при откате же ее ничего не посылается. В отсутствии активной транзакции отправка происходит без задержек. При этом запрещено ручное управление локальной транзакцией JMS-поставщика через вызовы интерфейса контекста commit() и rollback().

Rem: ACC-контейнер в управлении транзакциями приравнивается к JSE

Ex.: стиль JMS 2.0 в JEE7. Ловить исключения JNDI NamingException уже не надо

```
@Stateless
public class ProducerEJB {
  @Resource(lookup=”xxx/Factory”)
  private ConnectionFactory connectionFactory;
  @Resource(lookup=”xxx/Queue”)
  private Destination destination;

  public void sendMessage() {
    try(JMXContext context = connectionFactory.createContext()) {
        context.createProducer().send(destination, "Message is sended " + new Date());
    }
  }
}
```

Упрощенное JMS API в JEE. @Inject, @JMSConnectionFactory, @JMSPasswordCredential

Создание производителя сообщений на основе CDI MBean с использованием @Inject, доступной с JEE6, аналогично стилю JEE5 с @Resource, но еще сильнее автоматизирует и упрощает работу.

Внедрять через @Inject можно непосредственно JMSContext, но для выбора конкретной фабрики придется использовать костыль — дополнительную специальную аннотацию

@javax.jms.JMSConnectionFactory, где указывается строковое JNDI-имя фабрики. Кроме этого, для внедрения защищенного JMS-поставщиком контекста можно передать учетную информацию посредством @javax.jms.JMSPasswordCredential. Сессионный режим можно задать @javax.jms.JMSSessionMode

Ex.: стиль JMS 2.0 в CDI MBean JEE7. Внедряется непосредственно контекст, его жизненным циклом заведует контейнер и try-c-ресурсами больше не нужен

```
public class ProducerEJB {
  @Inject
  @JMSConnectionFactory(”xxx/Factory”)
  @JMSPasswordCredential(userName=”xxx”,password=”yyy”)
  @JMSSessionMode(JMSContext.DUPS_OK_ACKNOWLEDGE)
  private JMSContext context;
  @Resource(lookup=”xxx/Queue”)
  private Destination destination;

  public void sendMessage() {
     context.createProducer().send(destination, "Message is sended " + new Date());
  }
}
```

Применение JMS. Прием сообщений

Классическое JMS API в JSE

Для начала клиенту надо получить через запросы JNDI администрируемые JMS-поставщиком объекты фабрики соединений ConnectionFactory и источника Destination.

Затем при помощи них создаются и настраиваются объекты Connection, Session, MessageProducer, Message, которые на самом деле являются прокси JMS-поставщика.

В объект соединения при создании может зашиваться сессионный режим и учетная информация. Объект сессии Session порождается соединением и обеспечивает однопоточный прием сообщений путем создания экземпляров потребителей. В объект потребителя MessageConsumer зашивается при создании адрес источника.

Получение сообщений от источника в классическом API имеет синхронную природу: запускается оно вызовом start() соединения, сообщения получаются синхронно в циклическом опросе источника при помощи вызова receive() потребителя.

Сообщения принимаются потребителем сразу, далее надо подтвердить прием JMS-поставщику. Делается это по-разному в зависимости от параметров создания сессии вызовом соединения createSession(boolean transacted, int acknowledgeMode): transacted определяет транзакционное поведение, acknowledgeMode задает режим подтверждения. В режиме transacted = true JMS-поставщик автоматом создает локальную транзакцию при получении клиентом объекта Session, которой можно

управлять посредством сессионных вызовов commit() и rollback(). При этом автоматическое подтверждение будет проводиться только при фиксации транзакции, а при ее откате ничего делаться не будет. В случае нетранзакционного режима transacted = false, подтверждение будет определяться допустимыми режимами: AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE. В режиме автоподтверждения успешный прием сообщения вызовом receive() будет подтверждаться автоматически без задержек. В режиме подтверждения с допустимым дублированием допускается автоматическое подтверждение приема с задержкой, что может вызвать в ответ на простой повторную посылку дубля сообщения JMS-поставщиком. Используется для разгрузки сети, потребитель должен уметь работать с дублями. В режиме клиентского подтверждения, подтверждение делается путем программного вызова метода acknowledge() объекта Message клиентом.

После окончания приема сообщений стоит закрыть соединение Connection вызовом close() или использовать конструкцию try-with-resources, доступную с JMS 2.0. При использовании классического API JMS 1.1 надо обрабатывать проверяемые исключения класса JMSException.

Ex.: прием сообщений, стиль JMS 1.1 в JSE7

```
public class Consumer {
  public static void main(String[] args) {
     try {
      Context jndiContext = new InitialContext(...);
      ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("xxx/Factory");
      Destination queue = (Destination) jndiContext.lookup("xxx/Queue");
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageConsumer consumer = session.createConsumer(queue);
      connection.start();
      while(true) {
        TextMessage message = (TextMessage) consumer.receive();
        System.out.println("Message is received: " + message.getText());
      }
    } catch (NamingException | JMSException ex) {
       ex.printStackTrace();
    }
  }
}
```

Упрощенное JMS API в JSE

В упрощенном API JMS 2.0 существует 2 модели приема сообщений потребителем из источника: синхронная и асинхронная. Синхронный прием подобен классической модели приема, когда сообщения явно запрашиваются у источника вызовом receive() и клиент ждет результата. Отличие асинхронного приема в том, что JMS-поставщику через потребителя указывается специальный объект прослушивателя сообщений MessageListener, который принимает сообщения и обрабатывает их в параллельном

потоке, т.е. асинхронно.

Сообщения принимаются сразу , способ подтверждения задается сессионным режимом sessionMode контекста JMSContext. Для sessionMode = SESSION_TRANSACTED, подтверждение будет происходить при фиксации транзакции JMS-поставщика через вызов commit() интерфейса контекста, а при ее откате rollback() ничего подтверждаться не будет. Остальные режимы являются нетранзакционными. В режиме sessionMode = AUTO_ACKNOWLEDGE подтверждение будет происходит без задержек после успешного приема сообщения вызовом receive(). В режиме sessionMode = DUPS_OK_ACKNOWLEDGE подтверждение сообщений допускает задержки, которые могут вызвать повторную отправку JMS-поставщиком дублей. В режиме клиентского подтверждения CLIENT_ACKNOWLEDGE подтверждение делается путем программного вызова метода acknowledge() объекта Message клиентом.

Ex.: Синхронный прием сообщений. Стиль JMS 2.0 в JSE7: проброс непроверяемых JMSRuntimeException происходит автоматически; try-with-resource делегирует слежение за ж.ц. JMSContext JSE; содержимое тела получается сразу вызовом типизированного метода receiveBody()

```
public class Consumer {
  public static void main(String[] args) {
    try {
      Context jndiContext = new InitialContext();
      ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("xxx/Factory");
      Destination queue = (Destination) jndiContext.lookup("xxx/Queue");
      try (JMSContext context = connectionFactory.createContext()) {
        context.start();
        while(true) {
          String message = (TextMessage)context.createConsumer(queue).receiveBody(String.class);
          System.out.println(“Message is received: ”+message);
        }
      }
    } catch (NamingException ex) {
      ex.printStackTrace();
    }
  }
}
```

Ex: стиль JMS 2.0. Асинхронное потребление сообщений через прослушиватель в JSE

```
// Класс асинхронного прослушивателя
public class Listener implements MessageListener {
@Override
  public void onMessage(Message message) {
    System.out.println("Asynchronous message is received: " + message.getBody(String.class));
  }
}

// Класс потребителя с асинхронным приемом сообщений из источника
public class Consumer {
```

```
  public void static main(String[] args) {
    try {
      Context jndiContext = new InitialContext();
      ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup(“xxx/Factory”);
      Destination destination = (Destination) jndiContext.lookup(“xxx/Topic”);
      try(JMSContext context = connectionFactory.createJMSContext()){
        context.createConsumer(destination).setMessageListener(new Listener());
      }
    } catch(NamingException ex){
       ex.printStackTrace();
    }
  }
}
```

Упрощенное JMS API в JEE

В JEE для асинхронного потребления сообщений создан MDB. Этот компонент является вершиной автоматизации обмена сообщениями в JEE, в дополнение, обладая всеми службами EJB. MDB работают с JMS из коробки (а кроме них и с любыми EIS, оснащенными JCA-адаптером). Настраивается MDB посредством метаинформации, обычно аннотациями.

Но для наглядности можно сделать асинхронный потребитель JMS на основе обычного CDI MBean:

```
Ex.: стиль JMS 2.0 в CDI MBean JEE7. Внедряется непосредственно контекст, его жизненным
циклом заведует контейнер и try-с-ресурсами больше не нужен

public class ConsumerBean {
  @Inject
  @JMSConnectionFactory(”xxx/Factory”)
  @JMSPasswordCredential(userName=”xxx”,password=”yyy”)
  @JMSSessionMode(JMSContext.DUPS_OK_ACKNOWLEDGE)
  private JMSContext context;
  @Resource(lookup=”xxx/Queue”)
  private Destination destination;
  @Inject
   private Listener listener;

  public void receiveMessage() {
     context.createConsumer(destination).setMessageListener(listener);
  }
}

// Класс асинхронного прослушивателя, зарегистрированный как CDI MBean
public class Listener implements MessageListener {
@Override
  public void onMessage(Message message) {
    System.out.println("Asynchronous message is received: " + message.getBody(String.class));
```

```
  }
}
```

Контейнеры JEE дают возможность заменить JNDI-вызовы внедрением. В CDI MBean через @Inject можно внедрять сразу объект контекста JMSContext, для уточнения фабрики соединений и учетной информации надо применять уточняющие @JMSConnectionFactory, @JMSPasswordCredential, @JMSSessionMode. Источник Destination внедряется через старую @Resource. Это позволяет забыть про настройку JNDI-контекста и избавиться от обработки исключений NamingException, поскольку отвечает за внедрение контейнер. Также это перекладывает на CDI-движок слежение за ж.ц. контекста, т.е. избавляет от try-c-ресурсами или проблемы закрытия контекста вызовом close().

Более серьезные отличия в транзакциях для EJB и WEB контейнера. При наличии активной глобальной контейнерной транзакции (CMT) параметр sessionMode игнорируются контейнером, прием входящих сообщений автоматически подтверждается только при ее фиксации, а при откате ничего не подтверждается. В отсутствии активной CMT или при использовании BMT, подтверждение приема зависит от допустимых режимов - AUTO_ACKNOWLEDGE и DUPS_OK_ACKNOWLEDGE. В режиме AUTO_ACKNOWLEDGE подтверждение будет происходить немедленно после приема сообщения вызовом receive(). В режиме DUPS_OK_ACKNOWLEDGE подтверждение приема сообщений допускается с задержкой, что может вызвать повторную отправку дубля сообщения JMS-поставщиком. При этом запрещено ручное управление локальной транзакцией JMS-поставщика через вызовы интерфейса контекста commit() и rollback().

Rem: ACC-контейнер приравнивается в данной задаче к JSE, т.к. не поддерживает JTA-транзакции.

Применение JMS. Настройка обмена

Настройки обмена проводятся на стороне JMS-поставщика через клиентские прокси производителя и потребителя. При этом упрощенное API JMS 2.0 реализует текучий интерфейс, что удобно в написании однострочных выражений для приема или отправки сообщений.

Фильтрация сообщений. MessageSelector

При обмене сообщениями потребитель может настроить фильтрацию входящих сообщений по заголовку и свойствам javax.jms.Message. Это может быть полезно для разделения подписчиков темы при рассылках. Или для записи логов только важных сообщений.

Делается это при помощи строковых логических выражений, называемых селекторами и использующих синтаксис оператора WHERE SQL92. Операндами выступают

свойства и части заголовка простых типов, литералы. Допускаются логические операторы (NOT, AND, OR), сравнения (=, <>, >,<,>=,<=), арифметические (+,-,*,/) и выражения ([NOT] BETWEEN, [NOT] IN, [NOT] LIKE, IS [NOT] NULL). (?)

Селектор можно задать при создании через интерфейс Session или JMSContext вызовами createConsumer(Destination d, String messageSelector) потребителя MessageConsumer или JMSConsumer, соответственно. Фильтрация проводится JMS-поставщиком, поскольку реальный объект потребителя создается на стороне JMS-поставщика (клиенту возвращается его прокси).

Ex.: примеры задания фильтров через интерфейс потребителя

```
// Фильтрация по части заголовка JMSPriority: прием сообщения нормального приоритета
jmsContext.createConsumer(topic, “JMSPriority < 6”).receive();
// Фильтрация сообщения по части заголовка JMSPriority и прикладному свойству price
jmsContext.createConsumer(queue, “JMSPriority < 6 AND price < 10“).receive();
// Фильтрация по прикладному свойству price
jmsContext.createConsumer(queue, “price BETWEEN 10 AND 20“).receive();
```

Установка времени жизни сообщения. TimeToLive

Клиент-отправитель может задать время жизни сообщения в миллисекундах вызовом setTimeToLive(Long timeToLive) через интерфейс производителя. JMS-поставщик использует это значение для определения момента устаревания сообщения путем сложения времени жизни и момента получения сообщения в вызове send(). После устаревания, сообщения утилизируются и никуда не отправляются. Это может быть полезно для разгрузки системы при интенсивном обмене.

Ex.: пример задания времени жизни через интерфейс производителя

```
// Установка времени жизни сообщения в 1000мсек = 1сек
jmsContext.createProducer().setTimeToLive(1000).send(destination,message);
```

Задание стойкости доставки. DeliveryMode

Клиент-отправитель может задать JMS-поставщику режим стойкости доставки сообщений потребителю. Делается это вызовом setDeliveryMode(int deliveryMode) через интерфейс поставщика. Существует 2 режима стойкости — PERSISTENT, NON_PERSISTENT, определенные в классе DeliveryMode. Значение PERSISTENT требует от JMS-поставщика во время вызова send() сериализовать копию сообщения в ROM, при сбоях доставки (задержке отклика-подтверждения потребителя) сообщение может быть десериализовано и отправлено вновь, возможны дубли, (?)при этом часть заголовка JMSRedelivered = true(?). Режим NON_PERSISTENT предписывает не дублировать отправку при сбоях, тем самым гарантируя отсутствие дублей.

Ex.: пример задания «одноразовой» доставки через интерфейс производителя

```
jmsContext.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(queue,msg);
```

Задание режима подтверждения. AcknowledgeMode

В JMS существует понятие подтверждения доставки сообщения (acknowledgment). JMS-клиент уведомляет JMS-поставщика о том, что он получил сообщение. Если такого подтверждения нет, то JMS-поставщик может отправить сообщение вновь.

Режим подтверждение доставки сообщений зависит от среды выполнения JSE/JEE и параметров создания сессии или контекста.

В JEE при наличии активной CMT-транзакции, всю заботу о подтверждении доставки берет на себя JEE-контейнер: при фиксации транзакции доставка подтверждается, при откате неподтверждается. В JSE в транзакционном режиме сессии (transacted = true) или контекста (sessionMode = SESSION_TRANSACTED) JMS-поставщик автоматически создает локальную транзакцию, ее подтверждение происходит при фиксации вызовами commit() сессии или контекста, а при откате методом rollback() подтверждения не происходит.

В нетранзакционных режимах в JSE или при отсутствии активной JTA-транзакции в JEE (в т.ч. при использовании BMT), подтверждение зависит от режима. В режиме AUTO_ACKNOWLEDGE подтверждение происходит сразу после обработки сообщения потребителем: завершения вызова receive() интерфейса потребителя или отработки onMessage() прослушивателя MessageListener. В режиме DUPS_OK_ACKNOWLEDGE допускается задержка в подтверждении приема сообщения. При этом JMS-поставщик может повторно отправить дубль сообщения с частью заголовка JMSRedelivered = true в ответ на простой. Используется этот режим для разгрузки сети, потребитель должен быть готов к обработке дублей. Режим CLIENT_ACKNOWLEDGE допустим только в JSE, т.к. в JEE ответственность за подтверждение берет на себя JEE-контейнер. При этом клиент самостоятельно подтверждает успешный прием вызовом acknowledge() JMS-поставщика через интерфейс входящего сообщения Message.

В JEE для указания сессионного режима можно использовать @javax.jms.JMSSessionMode.

Ex.: пример задания режима подтверждения с дублями в JEE, при приеме сообщения проверка на дубль

```
public class ConsumerBean {
  @Inject
  @JMSConnectionFactory("xxx/Factory")
  @JMSSessionMode(JMSContext.DUPS_OK_ACKNOWLEDGE)
  private JMSContext jmsContext;
  @Resource(lookup = "xxx/Queue")
  private Destination queue;
```

```
  public void receiveMessage() {
    Message msg = jmsContext.createConsumer(queue).receive();
    if(msg.getJMSRedelivered()) return;
    System.out.println("Original message received: "+msg.getBody(String.class));
  }
}
```

Задание длительности подписки. SubscriptionDurability

В модели pub/sub (Topic) предполагается, что рассылка делается on-line подписчикам. Если же потребитель был off-line, то он не получит сообщение. Для того, чтобы потребители рассылочной схемы Topic могли получать off-line сообщения, им нужна длительная подписка. В JMS определено 2 типа подписки: Durable и NonDurable.

NonDurable – тип краткосрочной подписки, при которой теряются все сообщения, не переданные из-за разрыва связи. Это позволяет повысить производительность, вместе с тем ухудшает надежность обработки сообщений. Это подписка по-умолчанию.

Durable – тип длительной подписки, при которой JMS-поставщик сохраняет сообщения, которые не были отправлены потребителю в результате разрыва связи. Как только это соединение будет восстановлено, JMS-поставщик передаст все сохраненные сообщения. Длительная подписка создается вызовами createDurableConsumer(…) сессии или контекста. Длительная подписка доступна только одному потребителю, идентифицируемому JMS-поставщиком по clientID, заданному через интерфейсы сессии или контекста, и имени подписки, заданному при создании потребителя. После создания, такая подписка начинает сохранят off-line сообщения вплоть до вызова метода unsubscribe(String name).

```
jmsContext.setClientID("xxx");
jmsContext.createDurableConsumer(topic,"xxx/DurableTopic").receive();
```

Задание приоритета доставки сообщения

Приоритет сообщения может влиять на скорость доставки сообщений JMS-поставщиком. Существует всего 10 уровней приоритетности: 0-4 нормальные, 5-9 срочные. Уровень приоритета можно выставить JMS-поставщику через интерфейс производителя: вызовом setPriority(int n) установить значение приоритета, тогда при вызове send() JMS-поставщик впишет его в часть заголовка JMSPriority отправляемого сообщения.

```
jmsContext.createProducer().setPriority(1).send(topic,"text message");
```

Применение JMS MDB

MDB (Message Driven Bean, управляемый сообщениями компонент) – это асинхронный

прослушиватель сообщений, являющийся EJB. В терминах MOM это асинхронный потребитель.

Штатной системой обмена сообщениями для MDB является JMS. JMS MDB использует модель асинхронного потребления, реализуя интерфейс MessageListener. Взаимодействие с JMS-поставщиком в плане доставки сообщения максимально переложено на EJB-контейнер, в коде MDB можно сосредоточиться на обработке самого сообщения. Также EJB-контейнер обеспечивает транзакционное выполнение операций MDB с JMS-поставщиком при помощи глобальных (распределенных) JTA-транзакций. Настраивать обмен сообщениями можно в MDB при помощи метаинформации (аннотаций).

Класс JMS MDB должен отвечать общим требованиям к классам EJB, помечаться @javax.jms.MessageDriven, реализовывать интерфейс javax.jms.MessageListener (либо определять этот интерфейс через атрибут аннотации).

Ex: пример MDB, занимающегося обработкой писем из темы (xxx/Topic2) с подтверждением в ответную очередь (xxx/Queue3). Определена фильтрация входных сообщений по селектору, используется ленивый режим подтверждения с возможными дублями Dups-ok-acknowledge. Неявно определена глобальная контейнерная транзакция (CMT), откат которой в случае сбоя приведет к повторной отправке сообщения JMS-поставщиком и отмене отправки ответного сообщения в ответную очередь.

```
@MessageDriven(mappedName = "", activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "xxx/Topic2"),
  @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
  @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Dups-ok-
acknowledge"),
  @ActivationConfigProperty(propertyName="messageSelector", propertyValue="MessageFormat =
'Version 3.4' AND JMSPriority < 6 AND price BETWEEN 10 AND 20 ")
})
public class ExMDB implements MessageListener {
  private final Logger logger = Logger.getLogger(this.getClass().getName());
  @Inject
  @JMSConnectionFactory(“xxx/Factory”)
  private JMSContext jmsContext;
  @Resource(lookup=”xxx/Queue3”)
  private Destination queue;

  public void onMessage(Message message) {
    try {
      // Извлечение заголовков из сообщения
      MapMessage mapMessage = (MapMessage)message;
      int userId = mapMessage.getInt("userId");
      String email = mapMessage.getInt("email");
      // Обработка тела сообщения
      …
      // Отправка ответного письма по адресу JMSReplyTo входного message
      Destination topic = message.getJMSReplyTo();
```

```
    // Установка ID ответа для связи с входным сообщением в заголовок сообщения
    Message msg = jmsContext.createTextObject("Message successfully processed");
    msg.setJMSCorrelationID(message.getJMSCorrelationID());
    jmsContext.createProducer().send(queue, msg);
   } catch(Exception ex) {
     logger.log(Level.SEVERE,"Consumption error",ex);
     throw new JMSRuntimeException("Manually thrown exception");
   }
  }
}
```

Ругательства

- Message Service = система сообщений, отвечающая за их отправку и доставку
- Message Broker = Message Service
- MDB = Message Driven Bean = управляемый сообщениями компонент
- EIS = Enterprise Information System — промышленная информационная система
- JCA = Java Connector Architecture = java-архитектура соединителя — JEE-стандарт, описывающий соединение JEE-серверов приложений с EIS.
- JCA Resource Adapter = Resource Adapter = JCA Adapter — компонент, инкапсулирующий алгоритмы соединения с определенной EIS по правилам спецификации JCA.
- JEE-коннектор = JCA Resource Adapter
- MOM = Message Oriented Middleware
- Message Provider = поставщик сообщений — посредник, организующий обмен сообщениями между клиентами MOM
- Message Service = служба сообщений — абстракция MOM, регламентирующая порядок доступа к поставщику, создание, отправку и получение сообщений.
- Message Broker = конкретная реализация Message Service
- Message Router = Message Broker
- JMS = Java Message Service = служба сообщений JEE
- JMS-клиент — приложение, использующее JMS.
- JMS-поставщик — система, которая управляет отправкой и доставкой сообщений согласно JMS API.
- JMS-приложение — бизнес-система, состоящая из одного или более JMS-клиента и одного (обычно) или более JMS-поставщика.
- JMS-производитель (producer) – JMS-клиент, отправляющий сообщения.
- JMS-потребитель (consumer) – JMS-клиент, принимающий сообщения.

Литература

Enterprise JavaBeans 3.1 (6th edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Куправк, Майкл

Ремижан, стр. nnn

http://www.interface.ru/home.asp?artId=21612

http://www.ibm.com/developerworks/ru/library/j-jca1/index.html

https://glassfish.java.net/docs/4.0/mq-tech-over.pdf

https://www.ibm.com/support/knowledgecenter/SSEQTP_7.0.0/com.ibm.websphere.nd.doc/info/ae/ae/cmb_trans.html